ReScript Documentation

This document contains the complete official documentation of ReScript version 11 as of December 2023.

SECTION OFFICIAL-WEBSITE

SECTION GENTYPE

SECTION introduction

---

title: "Introduction" description: "GenType - Interoperability between ReScript and TypeScript / Flow" canonical: "/docs/gentype/latest/introduction"

GenType

`genType` is a code generation tool that lets you export ReScript values and types to use in TypeScript (TS), and import TS values and types into ReScript.

Converter functions between the two runtime representations are generated when required based on the type of the values. In particular, conversion of rescript-react components both ways is supported, with automatic generation of the wrappers.

Here's an article describing how to use `genType` as part of a migration strategy where a tree of components is gradually converted to ReScript bottom-up (old article containing Reason / BuckleScript): Adopting Reason: strategies, dual sources of truth, and why genType is a big deal.

The implementation of genType performs a type-directed transformation of ReScript programs after compilation. The transformed programs operate on data types idiomatic to JS.

For example, a ReScript function operating on a ReScript variant `type t = | A(int) | B(string)` (which is represented as custom objects with tags at runtime) is exported to a JS function operating on the corresponding JS object of type `{ tag: "A"; value: number } | { tag: "B"; value: string }`.

A Quick Example

Let's assume we are working on a TypeScript codebase and we want to integrate a single rescript-react component.

We want to be able to import the rescript-react component like any other React component in our existing TS code, but we also want to preserve all the ReScript types in the TS type system (and convert incompatible values if necessary).

That's exactly what genType was made for!

First we'll set up a rescript-react component:

```
/* src/MyComp.res */

@genType
type color =
  | Red
  | Blue;

@genType
@react.component
let make = (~name: string, ~color: color) => {
  let colorStr =
    switch (color) {
    | Red => "red"
    | Blue => "blue"
    };

  <div className={"color-" ++ colorStr}> {React.string(name)} </div>;
};
```

On a successful compile, `genType` will convert `src/MyComp.res` to a TS file called `src/MyComp.gen.tsx` which will look something like this:

```
// src/MyComp.gen.tsx

/* TypeScript file generated from MyComp.res by genType. */
/* eslint-disable import/first */
```

```
import * as React from 'react';

const $$toRE818596289: { [key: string]: any } = {"Red": 0, "Blue": 1};

// tslint:disable-next-line:no-var-requires
const MyCompBS = require('./MyComp.bs');

// tslint:disable-next-line:interface-over-type-literal
export type color = "Red" | "Blue";

// tslint:disable-next-line:interface-over-type-literal
export type Props = { readonly color: color; readonly name: string };

export const make: React.ComponentType<{ readonly color: color; readonly name: string }> = function MyComp(Arg1:
 any) {
  const $props = {color:$$toRE818596289[Arg1.color], name:Arg1.name};
  const result = React.createElement(MyCompBS.make, $props);
  return result
};
```

genType automatically maps the `color` variant to TS via a string union type `color = "Red" | "Blue"`, and also provides all the converters to convert between the ReScript & TS representation as well.

Therefore way we can seamlessly use ReScript specific data structures within TS without writing the converter code by hand!

Within our TypeScript application, we can now import and use the React component in the following manner:

```
// src/App.ts
import { make as MyComp } from "./MyComp.gen.tsx";

const App = () => {
  return (<div>
    <h1> My Component </h1>
    <MyComp color="Blue" name="ReScript & TypeScript" />
  </div>);
};
```

That's it for our quick example.

For detailed information, head to the Getting Started or Usage section.

Development

Since ReScript v10.1, genType is part of the compiler's GitHub repository.

SECTION getting-started

---

title: "Getting Started" description: "How to get started with genType in your ReScript projects" canonical: "/docs/gentype/latest/getting-started"

Getting Started

Setup

Since compiler v10.1, there's no need to install anything. For compiler 10.0 or older, install the binaries via `npm` (or `yarn`):

```
npm install gentype --save-dev

### Verify installed gentype binary
npx gentype --help
```

Add a `gentypeconfig` section to your `rescript.json` (See Configuration for details):

```
"gentypeconfig": {
    "language": "typescript",
    "shims": {},
    "generatedFileExtension": ".gen.tsx",
    "module": "es6",
    "debug": {
      "all": false,
      "basic": false
    }
}
```

Configuration

Every `genType` powered project requires a configuration item `"gentypeconfig"` at top level in the project's `rescript.json`. The configuration has following structure:

```
//...
"gentypeconfig": {
  "language": "typescript",
  "generatedFileExtension": ".gen.tsx",
  "module": "es6" | "commonjs",
  "shims": {
    "ReasonReact": "ReactShim"
  }
}
```

- **generatedFileExtension**
- File extension used for genType generated files (defaults to `.gen.tsx`)
- **language**
- `"typescript"` : the `language` setting is not required from compiler v10.1
- **module**
- Module format used for the generated `*.gen.tsx` files (supports `"es6"` and `"commonjs"`)
- **shims**
- Required only if one needs to export certain basic ReScript data types to JS when one cannot modify the sources to add annotations (e.g. exporting ReScript lists), and if the types are not first-classed in genType.
- Example: `Array<string>` with format: `"RescriptModule=JavaScriptModule"`

Adding Shims

A shim is a TS file that provides user-provided definitions for library types.

Configure your shim files within `"gentypeconfig"` in your `rescript.json`, and add relevant `.shims.ts` files in a directory which is visible by ReScript e.g. `src/shims/`. An example shim to export ReactEvent can be found here.

Testing the Whole Setup

Open any relevant `*.res` file and add `@genType` annotations to any bindings / values / functions to be used from JavaScript. If an annotated value uses a type, the type must be annotated too. See e.g. Hooks.res.

Save the file and rebuild the project via `npm run bs:build` or similar. You should now see a `*.gen.tsx` file with the same name (e.g. `MyComponent.res` -> `MyComponent.gen.tsx`).

Any values exported from `MyComponent.res` can then be imported from JS. For example:

```
import MyComponent from "./components/MyComponent.gen";
```

Examples

We prepared some examples to give you an idea on how to integrate `genType` in your own project. Check out the READMEs of the listed projects.

- typescript-react-example

Experimental features

These features are for experimentation only. They could be changed/removed any time, and not be considered breaking changes.

- Export object and record types as interfaces. To activate, add `"exportInterfaces": true` to the configuration. The types are also renamed from `name` to `Iname`.
- Emit prop types for the untyped back-end. To activate, add `"propTypes": true` and `"language": "untyped"` to the configuration.

Limitations

- **in-source = true**. Currently only supports ReScript projects with in-source generation and `.bs.js` file suffix.
- **Limited namespace support**. Currently there's limited namespace support, and only `namespace:true` is possible, not e.g. `namespace:"custom"`.

SECTION supported-types

---

title: "Supported Types" description: "Supported types and value convertion in GenType" canonical: "/docs/gentype/latest/supported-types"

Supported Types

Some types and values in ReScript do not map directly to JavaScript and need to be converted whenever a value crosses the boundary. This document gives an overview on how `genType`'s convertion works on different types.

Int

ReScript values e.g. `1`, `2`, `3` are unchanged. So they are exported to JS values of type `number`.

Float

ReScript values e.g. `1.0`, `2.0`, `3.0` are unchanged. So they are exported to JS values of type `number`.

String

ReScript values e.g. `"a"`, `"b"`, `"c"` are unchanged. So they are exported to JS values of type `string`.

Optionals

ReScript values of type e.g. `option<int>`, such as `None`, `Some(0)`, `Some(1)`, `Some(2)`, are exported to JS values `null`, `undefined`, `0`, `1`, `2`. The JS values are unboxed, and `null`/`undefined` are conflated. So the option type is exported to JS type `null` or `undefined` or `number`.

Nullables

ReScript values of type e.g. `Js.Nullable.t<int>`, such as `Js.Nullable.null`, `Js.Nullable.undefined`, `Js.Nullable.return(0)`, `Js.Nullable.return(1)`, `Js.Nullable.return(2)`, are exported to JS values `null`, `undefined`, `0`, `1`, `2`. The JS values are identical: there is no conversion unless the argument type needs conversion.

Records

ReScript record values of type e.g. `{x: int}` such as `{x: 0}`, `{x: 1}`, are exported to JS values of type `{x: number}` without runtime conversion.

Since records are immutable by default, their fields will be exported to readonly property types in Flow/TS. Mutable fields are specified in ReScript by e.g. `{mutable mutableField: string}`.

The `@as` annotation can be used to change the name of a field on the JS side of things. So e.g. `{@as("y") x: int}` is exported as JS type `{y: number}`.

If one field of the ReScript record has option type, this is exported to an optional JS field. So for example ReScript type `{x: option<int>}` is exported as JS type `{x?: number}`.

Objects

ReScript object values of type e.g. `{. "x":int}` such as `{"x": 0}`, `{"x": 1}`, `{"x": 2}`, are exported as identical JS object values `{x:0}`, `{x:1}`, `{x:2}`. This requires no conversion. So they are exported to JS values of type `{x:number}`. A conversion is required only when the type of some field requires conversions.

Since objects are immutable by default, their fields will be exported to readonly property types in Flow/TS. Mutable fields are specified in ReScript by e.g. `{ @set "mutableField": string }`.

It is possible to mix object and option types, so for example the ReScript type `{. "x":int, "y":option<string>}` exports to JS type `{x:number, ?y: string}`, requires no conversion, and allows option pattern matching on the ReScript side.

Tuples

ReScript tuple values of type e.g. `(int, string)` are exported as identical JS values of type `[number, string]`. This requires no conversion, unless one of types of the tuple items does. While the type of ReScript tuples is immutable, there's currently no mature enforcement in TS/Flow, so they're currenty exported to mutable tuples.

Variants

Ordinary variants (with capitalized cases, e.g. `| A | B(int)`) and polymorphic variants (with a backtick, e.g. `| `A | `B(int)`) are represented in the same way, so there's no difference from the point of view of JavaScript. Polymorphic variants don't have to be capitalized.

Variants can have an *unboxed*, or a *boxed* representation. The unboxed representation is used when there is at most one case with a payload, and that payload has object type; otherwise, a boxed representation is used. Object types are arrays, objects, records and tuples.

Variants without payloads are essentially sequences of identifiers. E.g. type `@genType type days = Monday | Tuesday`. The corresponding JS representation is `"Monday"`, `"Tuesday"`. Similarly, polymorphic variant type `@genType type days = [#Monday | #Tuesday]` has the same JS representation.

When at most one variant case has a payload, and if the payload is of object type, e.g. `Unnamed | Named({. "name": string, "surname": string})` then the representation is unboxed: JS values are e.g. `"Unnamed"` and `{name: "hello", surname: "world"}`. Similarly for polymorphic variants. Note that this unboxed representation does not use the label `"Named"` of the variant case with payload, because that value is distinguished from the other payload-less cases by its type: an object.

If there is more than one case with payload, or if the single payload has not type object, a boxed representation is used. The boxed representation has shape `{tag: "someTag", value: someValue}`. For example, type `| A | B(int) | C(string)` has values such as `"A"` and `{tag: "B", value: 42}` and `{tag: "C", value: "hello"}`. Polymorhphic variants are treated similarly. Notice that payloads for polymorphic variants are always unary: `` `Pair(int,int) `` has a single payload of type `(int,int)`. Instead, ordinary variants distinguish between unary `Pair((int,int))` and binary `Pair(int,int)` payloads. All those cases are represented in JS as `{tag: "Pair", value: [3, 4]}`, and the conversion functions take care of the different ReScript representations.

The `@genType.as` annotation can be used to modify the name emitted for a variant case on the JS side. So e.g. `| @genType.as("Arenamed") A` exports ReScript value `A` to JS value `"Arenamed"`. Boolean/integer/float constants can be expressed as `| @genType.as(true) True` and `| @genType.as(20) Twenty` and `| @genType.as(0.5) Half`. Similarly for polymorphic variants. The `@genType.as` annotation can also be used on variants with payloads to determine what appears in `{ tag: ... }`.

For more examples, see Variants.res and VariantsWithPayload.res.

NOTE: When exporting/importing values that have polymorphic variant type, you have to use type annotations, and cannot rely on type inference. So instead of `let monday = `Monday`, use `let monday : days = `Monday`. The former does not work, as the type checker infers a type without annotations.

Arrays

Arrays with elements of ReScript type `t` are exported to JS arrays with elements of the corresponding JS type. If a conversion is required, a copy of the array is performed.

Immutable arrays are supported with the additional ReScript library ImmutableArray.res/.resi, which currently needs to be added to your project. The type `ImmutableArray.t<+'a>` is covariant, and is mapped to readonly array types in TS/Flow. As opposed to TS/Flow, `ImmutableArray.t` does not allow casting in either direction with normal arrays. Instead, a copy must be performed using `fromArray` and `toArray`.

Functions and Function Components

ReScript functions are exported as JS functions of the corresponding type. So for example a ReScript function `foo : int => int` is exported as a JS function from numbers to numbers.

If named arguments are present in the ReScript type, they are grouped and exported as JS objects. For example `foo : (~x:int, ~y:int) => int` is exported as a JS function from objects of type `{x:number, y:number}` to numbers.

In case of mixed named and unnamed arguments, consecutive named arguments form separate groups. So e.g. `foo : (int, ~x:int, ~y:int, int, ~z:int) => int` is exported to a JS function of type `(number, {x:number, y:number}, number, {z:number}) => number`.

Function components are exported and imported exactly like normal functions. For example:

```
@genType
@react.component
let make = (~name) => React.string(name);
```

Imported Types

It's possible to import an existing TS/Flow type as an opaque type in ReScript. For example,

```
@genType.import("./SomeFlowTypes") type weekday
```

defines a type which maps to `weekday` in `SomeFlowTypes.js`. See for example Types.res and SomeFlowTypes.js.

Recursive Types

Recursive types which do not require a conversion are fully supported. If a recursive type requires a conversion, only a shallow

conversion is performed, and a warning comment is included in the output. (The alternative would be to perform an expensive conversion down a data structure of arbitrary size). See for example Types.res.

First Class Modules

ReScript first class modules are converted from their array ReScript runtime representation to JS Object types. For example,

```
module type MT = {
  let x: int
  let y: string
}
module M = {
  let y = "abc"
  let x = 42
}

@genType
let firstClassModule: module(MT) = module(M)
```

is exported as a JS object of type

```
{"x": number, "y": string}
```

Notice how the order of elements in the exported JS object is determined by the module type `MT` and not the module implementation `M`.

Polymorphic Types

If a ReScript type contains a type variable, the corresponding value is not converted. In other words, the conversion is the identity function. For example, a ReScript function of type `{payload: 'a} => 'a` must treat the value of the payload as a black box, as a consequence of parametric polymorphism. If a typed back-end is used, the ReScript type is converted to the corresponding generic type.

Exporting Values from Polymorphic Types with Hidden Type Variables

For cases when a value that contains a hidden type variable needs to be converted, a function can be used to produce the appropriate output:

Doesn't work

```
@genType
let none = None
```

```
export const none: ?T1 = OptionBS.none; // Errors out as T1 is not defined
```

Works

```
@genType
let none = () => None
```

```
const none = <T1>(a: T1): ?T1 => OptionBS.none;
```

Promises

Values of type `Js.Promise.t<arg>` are exported to JS promises of type `Promise<argJS>` where `argJS` is the JS type corresponding to `arg`. If a conversion for the argument is required, the conversion functions are chained via `.then(promise => ...)`.

SECTION usage

Usage

`genType` operates on two kinds of entities: *types* and *values*. Each can be *exported* from ReScript to JS, or *imported* into ReScript from JS. The main annotation is `@genType`, which by default means *export*.

Export and Import Types

The following exports a function type `callback` to JS:

```
@genType
type callback = ReactEvent.Mouse.t => unit
```

To instead import a type called `complexNumber` from JS module `MyMath.ts` (or `MyMath.js`), use the `@genType.import` annotation:

```
@genType.import("./MyMath")
type complexNumber
```

This imported type will be treated as opaque by ReScript.

Export and Import Values

To export a function `callback` to JS:

```
@genType
let callback = _ => Js.log("Clicked");
```

To rename the function and export it as `CB` on the JS side, use

```
@genType
@genType.as("CB")
let callback = _ => Js.log("Clicked");
```

or the more compact

```
@genType("CB")
let callback = _ => Js.log("Clicked");
```

To import a function `realValue` from JS module `MyMath.ts` (or `MyMath.js`):

```
@genType.import("./MyMath") /* JS module to import from. */
/* Name and type of the JS value to import. */
external realValue: complexNumber => float = "realValue";
```

> **Note:** With genType < 2.17.0 or bucklescript < 5.0.0, one had to add a line with `@bs.module` and the current file name. See the older README.

Because of the `external` keyword, it's clear from context that this is an import, so you can also just use `@genType` and omit `.import`.

To import a default JS export, use a second argument to `@genType.import` e.g. `@genType.import(("./MyMath", "default"))`.

Similarly, to import a value with a different JS name, use e.g. `@genType.import(("./MyMath", "ValueStartingWithUpperCaseLetter"))`.

To import nested values, e.g. `Some.Nested.value`, use e.g. `@genType.import(("./MyMath", "Some.Nested.value"))`.

Interface (.resi) and Implementation (.res) files

If both `Foo.resi` and `Foo.res` exist, the annotations are taken from `Foo.resi`. The same happens with local modules: if present, the module type gets precedence.

The behaviour can be overridden by adding annotation `@genType.ignoreInterface` at the top of `Foo.resi`. Use case: expose implementation details to JS but not to ReScript.

Type Expansion and @genType.opaque

If an exported type `persons` references other types in its definition, those types are also exported by default, as long as they are defined in the same file:

```
type name = string
type surname = string
type person = {name: name, surname: surname}

@genType
type persons = array<person>;
```

If however you wish to hide from JS the fact that `name` and `surname` are strings, you can do it with the `@genType.opaque` annotation:

```
@genType.opaque
type name = string

@genType.opaque
type surname = string

type person = {
  name,
  surname,
};

@genType
type persons = array<person>;
```

Renaming, @genType.as, and object mangling convention.

NOTE: Starting from ReScript 7.0.0, `@genType.as` on record fields will be discouraged, as it incurs a runtime conversion cost. Use a runtime free `@as` instead.

NOTE: Starting from ReScript 11.0.0, the object mangling is removed.

By default, entities with a given name are exported/imported with the same name. However, you might wish to change the appearence of the name on the JS side.

For example, to use a reserved keyword `type` as a record field:

```
@genType
type shipment = {
  date: float,
  @genType.as("type")
  type_: string,
}
```

Object field names follow ReScript's mangling convention:

```
Remove trailing "__" if present.
Otherwise remove leading "_" when followed by an uppercase letter, or keyword.
```

This means that the analogous example with objects is:

```
@genType
type shipment = {
  "date": float,
  "_type": string,
}
```

or the equivalent `"type__": string`.

Functions and function components also follow the mangling convention for labeled arguments:

```
@genType
let exampleFunction = (~_type) => "type: " ++ _type

@genType
@react.component
let exampleComponent = (~_type) => React.string("type: " ++ _type)
```

It is possible to use `@genType.as` for functions, though this is only maintained for backwards compatibility, and cannot be used on function components:

```
@genType
let functionWithGenTypeAs =
  (~date: float) => @genType.as("type") (~type_: string) => ...
```

NOTE: For technical reasons, it is not possible to use `@genType.as` on the first argument of a function.

Dependent Projects / Libraries

ReScript dependencies are specified in `bs-dependencies`. For example, if the dependencies are `"bs-dependencies": ["somelibrary"]` and `somelibrary` contains `Common.res`, this looks up the types of `foo` in the library:

```
@genType
let z = Common.foo;
```

Scoped packages of the form e.g. `@demo/somelibrary` are also supported.

NOTE: The library must have been published with the `.gen.ts` files created by genType.

SECTION BLOG

SECTION 2022-08-25-release-10-0-0

---

author: rescript-team date: "2022-08-25" previewImg: static/blog/grid_0.jpeg title: ReScript 10.0 badge: release description: | The first community powered release.

ReScript version 10 is available! Version 10 is a culmination of over a year's worth of work, bringing faster builds, improving JS interop, and including a bunch of bug fixes. It's also the first fully community powered release, with contributions from over 20 community members.

```
npm install rescript@10
```

All changes are listed here. Let's take a tour of a few of the features we're extra excited about.

Faster builds with native M1 support

Users with M1 chips should see a notable speedup, as the new ReScript version has full native support for M1.

Better ergonomics with Unicode support in regular strings

You can now use Unicode characters directly in regular strings. This will now produce what you'd expect:

```
let str = "Î£"
```

You can also pattern match on Unicode characters:

```
switch someCharacter {
  | 'Î£' => "what a fine Unicode char"
  | _ => "Unicode is fun"
}
```

Experimental optional record fields

Previously, a record would always have to define all its optional fields:

```
type user = {
  name: string,
  age: option<int>
}

let userWithoutAge = {
  name: "Name",
  age: None,
}

let userWithAge = {
  name: "Name",
  age: Some(34),
}
```

For small records like the one above, this is typically fine. But for records with many fields, the friction of having to always set all optional fields explicitly adds up. This release has a new experimental feature called optional record fields, allowing you to rewrite the above to this instead:

```
type user = {
  name: string,
  age?: int
}

// No need to set `age` unless it should have a value
let userWithoutAge = {
  name: "Name",
}

let userWithAge = {
  name: "Name",
  age: 34
}
```

Other than drastically improving the experience when working with large records with optional fields, this also has implications for bindings. For example, binding to JS APIs with large configuration objects is now more ergonomic. This feature also paves the way for other exciting features coming in the next release, such as a more idiomatic representation of React components.

What's next

Version 10 brings the building blocks needed for a number of exciting new features that'll be available in the next version. Features ranging from native support for async/await, to a new version of the JSX integration, making it leaner and more flexible. You'll hear more about this soon.

Upgrade guide

Please see the detailed changelog for a list of breaking changes. Each breaking change lists suggestions on how to upgrade your project. This can be out of your control in case of dependencies. In that case, please raise issues with the maintainers of those libraries.

One special word for PPXs, in particular for PPX authors: As mentioned in the changelog, some PPXs may give an error `"Attributes not allowed here"`. The solution is to adapt the PPXs following the example of `rescript-relay` in https://github.com/zth/rescript-relay/pull/372.

Acknowledgements

We would like to thank everyone from the community who volunteered their precious time to suport this project with contributions of any kind, from documentation, to PRs, to discussions in the forum. In particular, thank you @cknitt, @TheSpyder, @mattdamon108, @DZakh, @fhammerschmidt, @amiralies, @Minnozz, @Zeta611, @jchavarri, @nkrkv, @whitchapman, @ostera, @benadamstyles,

@cannorin, @ClaireNeveu, @kevinbarabash, @JsonKim, @Sehun0819, @glennsl, @namenu, @a-c-sreedhar-reddy.

SECTION 2023-05-17-enhanced-ergonomics-for-record-types

---

author: rescript-team date: "2023-05-17" title: Enhanced Ergonomics for Record Types badge: roadmap description: | A tour of new capabilities coming to ReScript v11

> This is the second post covering new capabilities that'll ship in ReScript v11. You can check out the first post on better interop with customizable variants here.

Records are a fundamental part of ReScript, offering a clear and concise definition of complex data structures, immutability by default, great error messages, and support for exhaustive pattern matching.

Even though records are generally preferable for defining structured data, there are still a few ergonomic annoyances, such as...

1. Existing record types can't be extended, which makes them hard to compose
2. Functions may only accept record arguments of the exact record type (no explicit sub-typing)

To mitigate the limitations above, one would need to retreat to structural objects to allow more flexible object field sharing and sub-typing, at the cost of more complex type errors and no pattern matching capabilities.

We think that records are a much more powerful data structure though, so we want to encourage more record type usage for these scenarios. This is why ReScript v11 will come with two new big enhancements for record types: **Record Type Spread** and **Record Type Coercion**.

Let's dive right into the details and show-case the new language capabilities.

Record Type Spread

As stated above, there was no way to share subsets of record fields with other record types. This means one had to copy / paste all the fields between the different record definitions. This was often tedious, error-prone and made code harder to maintain, especially when working with records with many fields.

In ReScript v11, you can now spread one or more record types into a new record type. It looks like this:

```
type a = {
  id: string,
  name: string,
}

type b = {
  age: int
}

type c = {
  ...a,
  ...b,
  active: bool
}
```

`type c` will now be:

```
type c = {
  id: string,
  name: string,
  age: int,
  active: bool,
}
```

Record type spreads act as a 'copy-paste' mechanism for fields from one or more records into a new record. This operation inlines the fields from the spread records directly into the new record definition, while preserving their original properties, such as whether they are optional or mandatory. It's important to note that duplicate field names are not allowed across the records being spread, even if the fields share the same type.

Needless to say, this feature offers a much better ergonomics when working with types with lots of fields, where variations of the same underlying type are needed.

Use case: Extending the Built-in DOM Nodes

This feature can be particularly useful when extending DOM nodes. For instance, in the case of the animation library Framer Motion, one could easily extend the native DOM types with additional properties specific to the library, leading to a more seamless and type-safe integration.

This is how you could bind to a `div` in Framer Motion with the new record type spreads:

```
type animate = {} // definition omitted for brevity

type divProps = {
  // Note: JsxDOM.domProps is a built-in record type with all valid DOM node attributes
  ...JsxDOM.domProps,
  initial?: animate,
  animate?: animate,
  whileHover?: animate,
  whileTap?: animate,
}

module Div = {
  @module("framer-motion") external make: divProps => Jsx.element = "div"
}
```

You can now use `<Div />` as a `<motion.div />` component from Framer Motion and your type definition is quite simple and easy to maintain.

Record Type Coercion

Record type coercion gives us more flexibility when passing around records in our application code. In other words, we can now coerce a record `a` to be treated as a record `b` at the type level, as long as the original record `a` contains the same set of fields in `b`. Here's an example:

```
type a = {
  name: string,
  age: int,
}

type b = {
  name: string,
  age: int,
}

let nameFromB = (b: b) => b.name

let a: a = {
  name: "Name",
  age: 35,
}

let name = nameFromB(a :> b)
```

Notice how we *coerced* the value `a` to type `b` using the coercion operator `:>`. This works because they have the same record fields. This is purely at the type level, and does not involve any runtime operations.

Additionally, we can also coerce records from `a` to `b` whenever `a` is a super-set of `b` (i.e. `a` containing all the fields of `b`, and more). The same example as above, slightly altered:

```
type a = {
  id: string,
  name: string,
  age: int,
  active: bool,
}

type b = {
  name: string,
  age: int,
}

let nameFromB = (b: b) => b.name

let a: a = {
  id: "1",
  name: "Name",
  age: 35,
  active: true,
}

let name = nameFromB(a :> b)
```

Notice how `a` now has more fields than `b`, but we can still coerce `a` to `b` because `b` has a subset of the fields of `a`.

In combination with optional record fields, one may coerce a mandatory field of an `option` type to an optional field:

```
type a = {
  name: string,

  // mandatory, but explicitly typed as option<int>
  age: option<int>,
}
```

```
type b = {
  name: string,
  // optional field
  age?: int,
}

let nameFromB = (b: b) => b.name

let a: a = {
  name: "Name",
  age: Some(35),
}

let name = nameFromB(a :> b)
```

The last example was rather advanced; the full feature set of record type coercion will later on be covered in a dedicated document page.

Record Type Coercion is Explicit

Records are nominally typed, so it is not possible to pass a record `a` as record `b` without an explicit type coercion. This conscious design decision prevents accidental type matching on shapes rather than records, ensuring predictable and more robust type checking results.

Try it out!

Feel free to check out the v11 alpha version on our online playground, or install the alpha release via npm: `npm i rescript@11.0.0-alpha.6`.

This release is mainly for feedback purposes and not intended for production usage.

Conclusion

The introduction of Record Type Spreads and Coercion in ReScript v11 will greatly improve the handling of record types. We're eager to see how you'll leverage these new language features in your ReScript projects.

Happy coding!

SECTION 2023-04-17-improving-interop

---

author: rescript-team date: "2023-04-17" title: Better interop with customizable variants badge: roadmap description: | A tour of new capabilities coming in ReScript v11

ReScript v11 is around the corner, and it comes packed with new features that will improve interop with JavaScript/TypeScript. Recently we've made some changes to the runtime representation of variants that'll allow you to use variants for a large number of new interop scenarios, zero cost. This is important, because variants are *the* feature of ReScript, enabling great data modeling, pattern matching and more.

- **Customizable runtime representation.** We're making the runtime representation of variants customizable. This will allow you to cleanly map variants to external data and APIs in many more cases than before.
- **Zero cost bindings to discriminated unions.** Variants with inline records will map cleanly to JavaScript/TypeScript discriminated unions.
- **Unboxed (untagged) variants.** We also introduce untagged variants - variants where the underlying runtime representation can be a primitive, without a specific discriminator. This will let you cleanly map to things like heterogenous array items, nullable values, and more.

Let's dive into the details.

Tagged variants

Variants with payloads have always been represented as a tagged union at runtime. Here's an example:

```
type entity = User({name: string}) | Group({workingName: string})

let user = User({name: "Hello"})
```

This is represented as:

```
var user = {
  TAG: /* User */ 0,
  name: "Hello",
};
```

However, this has been problematic when binding to external data because there has been no way to customize the discriminator (the `TAG` property) or how its value is represented for each variant case (`0` representing `User` here). This means that unless your external data is

modeled the exact same way as above, which is unlikely, you'd be forced to convert to the structure ReScript expects at runtime.

To illustrate this, let's imagine we're binding to an external union that looks like this in TypeScript:

```
type LoadingState =
  | { state: "loading"; ready: boolean }
  | { state: "error"; message: string }
  | { state: "done"; data: Data };
```

Currently, there's no good way to use a ReScript variant to represent this type without resorting to manual and error-prone runtime conversion. However, with the new functionality, binding to the above with no additional runtime cost is easy:

```
@tag("state")
type loadingState = | @as("loading") Loading({ready: bool}) | @as("error") Error({message: string}) | @as("done"
) Done({data: data})

let state = Error({message: "Something went wrong!"})
```

This will compile to:

```
var state = {
  state: "error",
  message: "Something went wrong!",
};
```

Let's break down what we've done to make this work:

- The `@tag` attribute lets you customize the discriminator (default: `TAG`). We're setting that to `"state"` so we map to what the external data looks like.
- Each variant case has an `@as` attribute. That controls what each variant case is discriminated on (default: the variant case name as string). We're setting all of the cases to their lowercase equivalent, because that's what the external data looks like.

The end result is clean and zero cost bindings to the external data, in a way that previously would require manual runtime conversion.

Now, let's look at a few more real-world examples.

Binding to TypeScript enums

```
// direction.ts
/** Direction of the action. */
enum Direction {
  /** The direction is up. */
  Up = "UP",

  /** The direction is down. */
  Down = "DOWN",

  /** The direction is left. */
  Left = "LEFT",

  /** The direction is right. */
  Right = "RIGHT",
}

export const myDirection = Direction.Up;
```

Previously, you'd be forced to use a polymorphic variant for this if you wanted clean, zero-cost interop:

```
type direction = [#UP | #DOWN | #LEFT | #RIGHT]
@module("./direction.js") external myDirection: direction = "myDirection"
```

Notice a few things:

- We're forced to use the names of the enum payload, meaning it won't fully map to what you'd use in TypeScript
- There's no way to bring over the documentation strings, because polymorphic variants are structural, so there's no one source definition for them to look for docstrings on. This is true *even* if you annotate with your explicitly written out polymorphic variant definition.

With the new runtime representation, this is how you'd bind to the above enum instead:

```
/** Direction of the action. */
type direction =
  | /** The direction is up. */
  @as("UP")
  Up

  | /** The direction is down. */
  @as("DOWN")
  Down

  | /** The direction is left. */
```

```
  @as("LEFT")
  Left

  | /** The direction is right. */
  @as("RIGHT")
  Right

@module("./direction.js") external myDirection: direction = "myDirection"
```

Now, this maps 100% to the TypeScript code, including letting us bring over the documentation strings so we get a nice editor experience.

String literals

The same logic is easily applied to string literals from TypeScript, only here the benefit is even larger, because string literals have the same limitations in TypeScript that polymorphic variants have in ReScript.

```
// direction.ts
type direction = "UP" | "DOWN" | "LEFT" | "RIGHT";
```

There's no way to attach documentation strings to string literals in TypeScript, and you only get the actual value to interact with.

With the new customizable variants, you could bind to the above string literal type easily, but add documentation, and change the name you interact with in ReScript. And there's no runtime cost.

Untagged variants

We've also implemented support for *untagged variants*. This will let you use variants to represent values that are primitives and literals in a way that hasn't been possible before.

We'll explain what this is and why it's useful by showing a number of real world examples. Let's start with a simple one on how we can now represent a heterogenous array.

```
@unboxed type listItemValue = String(string) | Boolean(bool) | Number(float)

let myArray = [String("Hello"), Boolean(true), Boolean(false), Number(13.37)]
```

Here, each value will be *unboxed* at runtime. That means that the variant payload will be all that's left, the variant case name wrapping the payload itself will be stripped out and the payload will be all that remains.

It, therefore, compiles to this JS:

```
var myArray = ["hello", true, false, 13.37];
```

This was previously possible to do, leveraging a few tricks, when you didn't need to potentially read the values from the array again in ReScript. But, if you wanted to read back the values, you'd have to do a number of manual steps.

In the above example, reaching back into the values is as simple as pattern matching on them.

Let's look at a few more examples of what untagged variants enable.

Pattern matching on nullable values

Previously, any value that might be `null` would need to be explicitly converted to an option by using for example `Null.toOption` before you could use pattern matching on it. Here's a typical example of how that could look:

```
type userAge = {ageNum: Null.t<int>}

type rec user = {
  name: string,
  age: Null.t<userAge>,
  bestFriend: Null.t<user>,
}

let getBestFriendsAge = user =>
  switch user.bestFriend->Null.toOption {
  | Some({age}) =>
    switch age->Null.toOption {
    | None => None
    | Some({ageNum}) => ageNum->Null.toOption
    }
  | None => None
  }
```

As you can see, you need to convert each level of nullables explicitly, which makes it hard to fully utilize pattern matching. With the new unboxed variant representation, we'll instead be able to do this:

```
// The type definition below is inlined here to examplify, but this definition will live in [Core](https://githu
b.com/rescript-association/rescript-core) and be easily accessible
```

```
module Null = {
  @unboxed type t<'a> = Present('a) | @as(null) Null
}

type userAge = {ageNum: Null.t<int>}

type rec user = {
  name: string,
  age: Null.t<userAge>,
  bestFriend: Null.t<user>,
}

let getBestFriendsAge = user =>
  switch user.bestFriend {
  | Present({age: Present({ageNum: Present(ageNum)})}) => Some(ageNum)
  | _ => None
  }
```

> Notice how `@as` now allows us to say that an unboxed variant case should map to a specific underlying *primitive*. `Present` has a type variable, so it can hold any type. And since it's an unboxed type, only the payloads `'a` or `null` will be kept at runtime. That's where the magic comes from.

We can now utilize pattern matching fully without needing to do any conversion.

This has a few implications:

- Dealing with external data, that is often nullable and seldom guaranteed to map cleanly to `option` without needing conversion, becomes much easier and zero cost.
- Special handling like [@return(nullable)] becomes redundant. This is good also because the current functionality does not work in all cases. The new functionality will work anywhere.

Decoding and encoding JSON idiomatically

With unboxed variants, we have everything we need to define a JSON type:

```
@unboxed
type rec json =
  | @as(false) False
  | @as(true) True
  | @as(null) Null
  | String(string)
  | Number(float)
  | Object(Js.Dict.t<json>)
  | Array(array<json>)

let myValidJsonValue = Array([String("Hi"), Number(123.)])
```

Here's an example of how you could write your own JSON decoders easily using the above, leveraging pattern matching:

```
@unboxed
type rec json =
  | @as(false) False
  | @as(true) True
  | @as(null) Null
  | String(string)
  | Number(float)
  | Object(Js.Dict.t<json>)
  | Array(array<json>)

type rec user = {
  name: string,
  age: int,
  bestFriend: option<user>,
}

let rec decodeUser = json =>
  switch json {
  | Object(userDict) =>
    switch (
      userDict->Dict.get("name"),
      userDict->Dict.get("age"),
      userDict->Dict.get("bestFriend"),
    ) {
    | (Some(String(name)), Some(Number(age)), Some(maybeBestFriend)) =>
      Some({
        name,
        age: age->Float.toInt,
        bestFriend: maybeBestFriend->decodeUser,
      })
    | _ => None
    }
  | _ => None
  }
```

```
let decodeUsers = json =>
  switch json {
  | Array(array) => array->Array.map(decodeUser)->Array.keepSome
  | _ => []
  }
```

Encoding that same structure back into JSON is also easy:

```
let rec userToJson = user => Object(
  Dict.fromArray([
    ("name", String(user.name)),
    ("age", Number(user.age->Int.toFloat)),
    (
      "bestFriend",
      switch user.bestFriend {
      | None => Null
      | Some(friend) => userToJson(friend)
      },
    ),
  ]),
)

let usersToJson = users => Array(users->Array.map(userToJson))
```

This can be extrapolated to many more cases.

Wrapping up

We hope you'll enjoy using these new capabilities. Some of them are a big leap forward for ReScript's interop with JavaScript and TypeScript, and we hope they will simplify many scenarios and open up a few new doors.

And last but not least, you can try ReScript v11 today by installing `npm i rescript@next`.

SECTION 2023-09-18-uncurried-mode

---

author: rescript-team date: "2023-09-18" title: Uncurried Mode badge: roadmap description: | A tour of new capabilities coming to ReScript v11

> This is the fourth post covering new capabilities that'll ship in ReScript v11. You can check out the first post on Better Interop with Customizable Variants, the second post on Enhanced Ergonomics for Record Types and the third post on First-class Dynamic Import Support.

Introduction

ReScript is a language that strives to keep its users free from experiencing runtime errors. Usually, when a program compiles, it will already do what the user described. But there is still a concept in the language that makes it easy to let some errors slip through. Currying!

Because of currying, partial application of functions is possible. That feature is always advertised as something really powerful. For instance,

```
let add = (a, b) => a + b
let addFive = add(5)
```

is shorter than having to write all remaining parameters again

```
let add = (a, b) => a + b
let addFive = (b) => add(5, b)
```

This comes at a price though. Here are some examples to show the drawbacks of currying:

- Errors because of changed function signatures have their impact at the use site. Consider this example, where the signature of the onChange function is extended with a labeled argument ```diff @react.component
- let make = (~onChange: string => option unit>) => {
- let make = (~onChange: (~a: int, string) => option unit>) => { React.useEffect(() => { // As partial application is allowed, there is no error here. let cleanup = onChange("change")

  ```
  // Here it errors with "This call is missing an argument of type (~a: int)"
  cleanup
  ```

  }) } `* If you wanted explicitly uncurry a function, you needed to annotate it with the uncurried dot.` rescript (. param) => () `* As ReScript could not fully statically analyze when to automatically uncurry a function over multiple files, it led to unnecessary `Curry.` calls in the emitted JavaScript code. * In the standard library (`Belt`), there are both curried and uncurried versions of the same function so you were required to`

think for yourself when to use the uncurried version and when only the curried one will work. * In combination with `ignore` / `let _ = ...`, curried can lead to unexpected behavior at runtime after adding a parameter to a function, because you are accidentally ignoring the result of a partial evaluation so that the function is not called at all. 1. Have a look at this simple function. It is assigned to `_` because we ignore the resulting `string` value.res let myCurriedFn = (~first) => first let _ = myCurriedFn(~first="Hello!") // ^ string 2. Now the function got a second parameter `~second`. Here, the resulting value is a function, which means it is not fully applied and thus never executed.res let myCurriedFn = (~first, ~second) => first ++ " " ++ second let _ = myCurriedFn(~first="Hello!") // ^ (~second: string) => string 3. One way to prevent such errors is to annotate the underscore with the function's return type:res let _: string = myCurriedFn(~first="Hello!") 4. However, the same issue arises when using the built-in ignore function, which cannot be annotated:res myCurriedFn(~first="Hello!")->ignore ```

Those are all only some small paper cuts, but all of them are intricacies that make the language harder to learn.

Uncurried mode

Starting with ReScript 11, your code will be compiled in uncurried mode. Yes, there is still a way to turn it off (see below), but we have decided to already default to this behavior to make it easier for newcomers. In uncurried mode, the introductory example yields an error:

```
let add = (a, b) => a + b
let addFive = add(5) // <-- Error:
// This uncurried function has type (. int, int) => int
// It is applied with 1 arguments but it requires 2.
```

to fix it, you have two options:

1. state the remaining parameters explicitly `rescript let add = (a, b) => a + b let addFive = (b) => add(5, b)`
2. or use the new explicit syntax for partial application `rescript let add = (a, b) => a + b let addFive = add(5, ...)`

The former approach helps library authors support both ReScript 11 and earlier versions.

No final unit anymore

We are happy to announce that with uncurried mode the "final unit" pattern is not necessary anymore, while you still can use optional or default parameters.

```
// old
let myFun = (~name=?, ())

// new
let myFun = (~name=?)
```

More wins

Furthermore, function calls in uncurried mode are now guaranteed to get compiled as simple JavaScript function calls, which is quite nice for readability of the generated code. It may also give you some (negligible) performance gains.

How to switch back to curried mode

While we strongly encourage all users to switch to the new uncurried mode, it is still possible to opt out. Just add a

```
{
  "uncurried": false
}
```

to your `bsconfig.json`, and your project will be compiled in curried mode again.

If you have uncurried mode off and still want to try it on a per-file basis, you can turn it on via

```
@@uncurried
```

at the top of a `.res` file.

Conclusion

Many thoughts have led to this decision, but we think this change is a great fit for a compile-to-JS language overall. If you are interested in the details, have a look at the corresponding forum post and its comments.

We hope that this new way of writing ReScript will make it both easier for beginners and also more enjoyable for the seasoned developers.

As always, we're eager to hear about your experiences with our new features. Feel free to share your thoughts and feedback with us on our issue tracker or on the forum.

Happy hacking!

SECTION 2023-06-05-first-class-dynamic-import-support

---

author: rescript-team date: "2023-06-05" title: First-class Dynamic Import Support badge: roadmap description: | A tour of new capabilities coming to ReScript v11

> This is the third post covering new capabilities that'll ship in ReScript v11. You can check out the first post on Better Interop with Customizable Variants and the second post on Enhanced Ergonomics for Record Types.

Introduction

When developing apps in JavaScript, every line of code eventually needs to be bundled up and shipped to the browser. As the app grows, it's usually a good idea to split up and load parts of the app code on demand as separate JS modules to prevent bundle bloat.

To accomplish this, browsers provide support for dynamic loading via the globally available `import()` function to allow code splitting and lazy loading and ultimately reducing initial load times for our applications.

Even though ReScript has been able to bind to `import` calls via `external` bindings, doing so was quite hard to maintain for to the following reasons:

1. An `import` call requires a path to a JS file. The ReScript compiler doesn't directly expose file paths for compiled modules, so the user has to manually find and rely on compiled file paths.
2. The return type of an `import` call needs to be defined manually; a quite repetitive task with lots of potential bugs when the imported module has changed.

Arguably, these kind of problems should ideally be tackled on the compiler level, since the ReScript compiler knows best about module structures and compiled JS file locations â€” so we finally decided to fix this.

Today we're happy to announce that ReScript v11 will ship with first-class support for dynamic imports as part of the language.

Let's have a look!

Import Parts of a Module

We can now use the `Js.import` function to dynamically import a value or function from a ReScript module. The import call will return a promise, resolving to the dynamically loaded value.

For example, imagine the following file `MathUtils.res`:

```
// MathUtils.res
let add = (a, b) => a + b
let sub = (a, b) => a - b
```

Now let's dynamically import the `add` function in another module, e.g. `App.res`:

```
// App.res
let main = async () => {
  let add = await Js.import(MathUtils.add)
  let onePlusOne = add(1, 1)

  RescriptCore.Console.log(onePlusOne)
}
```

This compiles to:

```
async function main() {
  var add = await import("./MathUtils.mjs").then(function(m) {
    return m.add;
  });

  var onePlusOne = add(1, 1);
  console.log(onePlusOne);
}
```

Notice how the compiler keeps track of the relative path to the module you're importing, as well as plucking out the value you want to use from the imported module.

Quite a difference compared to doing both of those things manually, right? Now let's have a look at a more concrete use-case with React components.

Use-case: Importing a React component

> **Note:** This section requires the latest @rescript/react bindings to be installed (*0.12.0-alpha.2 and above*).

Our dynamic import makes tasks like lazy loading React components a simple one-liner. First let's define a simple component as an example:

```
// Title.res
@react.component
let make = (~text) => {
  <div className="title">{text->React.string}</div>
}
```

Now let's dynamically import the `<Title/>` component by passing the result of our dynamic import to `React.lazy_`:

```
module LazyTitle = {
  let make = React.lazy_(() => Js.import(Title.make))
}

let titleJsx = <LazyTitle text="Hello!" />
```

That's all the code we need! The new `<LazyTitle />` component behaves exactly the same as the wrapped `<Title />` component, but will be lazy loaded via React's built in lazy mechanism.

Needless to say, all the code examples you've seen so far are fully type-safe.

Import a Whole Module

Sometimes it is useful to dynamically import the whole module instead. For example, you might have a collection of utility functions in a dedicated module that tend to be used together.

The syntax for importing a whole module looks a little different, since we are operating on the module syntax level; instead of using `Js.import`, you may simply `await` the module itself:

```
// App.res
let main = async () => {
  module Utils = await MathUtils

  let twoPlusTwo = Utils.add(2, 2)
  RescriptCore.Console.log(twoPlusTwo)
}
```

And, the generated JavaScript will look like this:

```
async function main() {
  var Utils = await import("./MathUtils.mjs");

  var twoPlusTwo = Utils.add(2, 2);
  console.log(twoPlusTwo);
}
```

The compiler correctly inserts the module's import path and stores the result in a `Utils` variable.

Try it out!

Feel free to try out our new dynamic import feature with the latest beta release:

```
npm install rescript@11.0.0-beta.1
```

Please note that this release is only intended for experiments and feedback purposes.

Conclusion

The most important take away of the new dynamic imports functionality in ReScript is that you'll never need to care about *where* what you're importing is located on the file system - the compiler already does it for you.

We hope that it will help shipping software with better end-user experience with faster load times and quicker app interaction, especially on slower network connections.

As always, we're eager to hear about your experiences with our new features. Feel free to share your thoughts and feedback with us on our issue tracker or on the forum.

Happy hacking!

SECTION 2023-02-02-release-10-1

---

author: rescript-team date: "2023-02-02" title: ReScript 10.1 badge: release description: | Async/await & better Promise support, JSX v4, and more!

Introduction

We are happy to announce ReScript 10.1!

ReScript is a robustly typed language that compiles to efficient and human-readable JavaScript. It comes with one of the fastest build toolchains and offers first class support for interoperating with ReactJS and other existing JavaScript code.

Use `npm` to install the newest 10.1 release:

```
npm install rescript

### or

npm install rescript@10.1
```

This version comes with two major language improvements we've all been waiting for. **async/await support** for an easy way to write asynchronous code in a synchronous manner, and a **new JSX transform** with better ergonomics, code generation and React 18 support.

Alongside the major changes, there have been many bugfixes and other improvements that won't be covered in this post.

Feel free to check the Changelog for all the details.

New `async` / `await` syntax

Async / await has arrived. Similar to its JS counterparts, you are now able to define `async` functions and use the `await` operator to unwrap a promise value. This allows writing asynchronous code in a synchronous fashion.

Example:

```
// Some fictive functionality that offers asynchronous network actions
@val external fetchUserMail: string => promise<string> = "GlobalAPI.fetchUserMail"
@val external sendAnalytics: string => promise<unit> = "GlobalAPI.sendAnalytics"

// We use the `async` keyword to allow the use of `await` in the function body
let logUserDetails = async (userId: string) => {
  // We use `await` to fetch the user email from our fictive user endpoint
  let email = await fetchUserMail(userId)

  await sendAnalytics(`User details have been logged for ${userId}`)

  Js.log(`Email address for user ${userId}: ${email}`)
}
```

To learn more about our async / await feature, check out the relevant manual section.

New `promise` builtin type and `Js.Promise2` module

In previous versions of ReScript, promises were expressed as a `Js.Promise.t<'a>` type, which was a little tedious to type. From now on, users may use the `promise<'a>` type instead.

Quick example of a `.resi` file using the new `promise` type:

```
// User.resi
type user

let fetchUser: string => promise<user>
```

Way easier on the eyes, don't you think? Note that the new `promise` type is fully compatible with `Js.Promise.t` (no breaking changes).

Additionally, we also introduced the `Js.Promise2` module as a stepping stone to migrate `Js.Promise` based code to a first-pipe (->) friendly solution. For the daily practise you'll almost always want to use `async` / `await` to handle promises.

(*Sidenote*: We are also well aware that our users want a solution to unify `Belt`, `Js` and `Js.xxx2` and have a fully featured "standard library" instead of adding more `Js.xxx2` modules. Good news is that we have a solution in the pipeline to fix this. `Js.Promise2` was introduced to ease the process later on and is not supposed to be the panacea of promise handling.)

If you are already using a third-party promise library like ryyppy/rescript-promise or similar, there's no need to migrate any existing code. Introduce `async` / `await` gradually in your codebase as you go.

New JSX v4 syntax

ReScript 10.1 now ships with JSX v4. Here's what's new:

- **Cleaner interop.** Due to recent improvements in the type checker, the `@react.component` transformation doesn't require any `makeProps` convention anymore. `make` functions will now be transformed into a `prop` type and a component function. That's it.
- **Two new transformation modes**. JSX v4 comes with a `classic` mode (= `React.createElement`) and `automatic` mode (= `jsx-`

`runtime` calls). The latter is the new default, moving forward with `rescript/react@0.11` and `React@18`.

- **Allow mixing JSX configurations on the project and module level.** Gradually mix and match JSX transformations and modes without migrating any old code!
- **Pass `prop` types** to `@react.component`. You can now fine tune `@react.component` with your specific prop type needs. Very useful for libraries and frameworks to define component interfaces.
- **Less boilerplate when using `React.Context`**. Check out our example for comparison.
- **Revisited props spread operator.** This will allow users to spread records in JSX without sacrificing their sanity. Note that this implementation has harder constraints than its JS counterpart. (requires `rescript/react@0.11` or higher)
- **Better type inference of props.** Type inference when passing e.g. variants that are defined in the same module as the component is much improved. With the earlier JSX version, you'd often need to write code like this in order for the compiler to understand which variant you're passing: `<Button variant=Button.Primary text="Click" />`. With JSX v4, you won't need to tell the compiler where the variant you're passing is located: `<Button variant=Primary text="Click" />`.

Code tells more than words, so here's a non-exhaustive code example to highlight the different JSX features. Make sure to also check out the JS output and play around with the code in our newest playground!

```
// Set the jsx configuration per module
@@jsxConfig({version: 4, mode: "automatic"})

module AutomaticModeExample = {
  // "automatic" mode will compile jsx to the React 18 compatible
  // jsx-runtime calls
  @@jsxConfig({version: 4, mode: "automatic"})

  @react.component
  let make = (~name) => {
    <div> {React.string(`Hello ${name}`)} </div>
  }
}

module ClassicModeExample = {
  // "classic" mode will compile jsx to React.createElement calls
  @@jsxConfig({version: 4, mode: "classic"})

  @react.component
  let make = (~name) => {
    <div> {React.string(`Hello ${name}`)} </div>
  }
}

module NoAttributeExample = {
  // No need for `makeProps` anymore
  type props = {name: string}

  let make = (props: props) => {
    <div> {React.string(`Hello ${props.name}`)} </div>
  }
}

module ReactInterfaceExample: {
  @react.component
  let make: (~name: string, ~age: int=?) => React.element
} = {
  @react.component
  let make = (~name, ~age=0) => {
    <div>
      {React.string(
        `Hello ${name}, you are ${Belt.Int.toString(age)} years old.`,
      )}
    </div>
  }
}

module PropTypeInjectionExample = {
  // Let's assume we have a prop type that we wanna enforce
  // as our labeled arguments
  type someoneElsesProps = {isHuman: bool}

  // Here we tell the `react.component` decorator what props to infer.
  // Useful for e.g. NextJS usage, or to create components that should
  // comply to certain library component interfaces
  @react.component(: someoneElsesProps)
  let make = (~isHuman) => {
    let msg = switch isHuman {
    | true => "hello human"
    | false => "hello fellow computer"
    }
    <div> {React.string(msg)} </div>
  }
}

module PropSpreadExample = {
  // Note: This will require @rescript/react 0.11 or later
  @@jsxConfig({version: 4, mode: "automatic"})
```

```
  @react.component
  let make = () => {
    let props = {NoAttributeExample.name: "World"}

    <NoAttributeExample {...props} />
  }
}

let root =
  <div>
    <AutomaticModeExample name="Automatic" />
    <ClassicModeExample name="Classic" />
    <NoAttributeExample name="NoAttribute" />
    <ReactInterfaceExample name="Interface" />
    <PropTypeInjectionExample isHuman=true />
    <PropSpreadExample />
  </div>
```

```
import * as React from "react";
import * as JsxRuntime from "react/jsx-runtime";

function Playground$AutomaticModeExample(props) {
  return JsxRuntime.jsx("div", {
              children: "Hello " + props.name + ""
            });
}

var AutomaticModeExample = {
  make: Playground$AutomaticModeExample
};

function Playground$ClassicModeExample(props) {
  return React.createElement("div", undefined, "Hello " + props.name + "");
}

var ClassicModeExample = {
  make: Playground$ClassicModeExample
};

function make(props) {
  return JsxRuntime.jsx("div", {
              children: "Hello " + props.name + ""
            });
}

var NoAttributeExample = {
  make: make
};

function Playground$ReactInterfaceExample(props) {
  var age = props.age;
  var age$1 = age !== undefined ? age : 0;
  return JsxRuntime.jsx("div", {
              children: "Hello " + props.name + ", you are " + String(age$1) + " years old."
            });
}

var ReactInterfaceExample = {
  make: Playground$ReactInterfaceExample
};

function Playground$PropTypeInjectionExample(props) {
  var msg = props.isHuman ? "hello human" : "hello fellow computer";
  return JsxRuntime.jsx("div", {
              children: msg
            });
}

var PropTypeInjectionExample = {
  make: Playground$PropTypeInjectionExample
};

function Playground$PropSpreadExample(props) {
  return JsxRuntime.jsx(make, {
              name: "World"
            });
}

var PropSpreadExample = {
  make: Playground$PropSpreadExample
};

var root = JsxRuntime.jsxs("div", {
      children: [
        JsxRuntime.jsx(Playground$AutomaticModeExample, {
              name: "Automatic"
```

```
            }),
        JsxRuntime.jsx(Playground$ClassicModeExample, {
              name: "Classic"
            }),
        JsxRuntime.jsx(make, {
              name: "NoAttribute"
            }),
        JsxRuntime.jsx(Playground$ReactInterfaceExample, {
              name: "Interface"
            }),
        JsxRuntime.jsx(Playground$PropTypeInjectionExample, {
              isHuman: true
            }),
        JsxRuntime.jsx(Playground$PropSpreadExample, {})
      ]
    });

export {
  AutomaticModeExample ,
  ClassicModeExample ,
  NoAttributeExample ,
  ReactInterfaceExample ,
  PropTypeInjectionExample ,
  PropSpreadExample ,
  root ,
}
```

How to migrate to JSX v4?

We provide a full migration guide with all the details of an migration.

Make sure to also check out the rescript-react changelog as well.

What's next?

Our contributors are already one step ahead and are currently working on improvements for the next major v11 release. Things that are currently being explored:

- Make uncurried functions the default. This will be a huge change in terms of how we do interop and will open completely new ways to interact with existing codebases. It will also allow us to improve tooling in ways that wouldn't have been possible in a curried language.
- Explorations for a community "standard library" that goes beyond `Belt` and `Js.*`. This will also involve disabling / removing global "Stdlib" modules that shouldn't be used (e.g. `Array`, `List`, etc).
- New tooling to generate markdown from docstrings (module, type and value level). This will be super simple, but very effective.
- Explorations for a localized documentation page (currently in a slowed-down exploration phase, but we will be getting there)

Check out the v11 milestone on our `rescript-lang` repo for more details on future improvements.

Acknowledgements

As always, we want to thank our contributors for building an amazing platform. Special thanks go out to mununki for building the new JSX v4 syntax. Amazing work!

That's it

We hope you enjoy the newest improvements as much as we do.

In case there's any issues / problems, make sure to report bugs to rescript-lang/rescript-compiler (language / syntax / jsx), rescript-lang/rescript-react (React 16 / 18 binding) or rescript-association/rescript-lang.org (documentation) repositories.

Also feel free to visit the ReScript forum to ask questions and connect with other ReScripters.

SECTION MANUAL

SECTION libraries

---

title: "Libraries & Publishing" description: "Install & publish ReScript packages" canonical: "/docs/manual/latest/libraries"

Libraries & Publishing

ReScript libraries are just like JavaScript libraries: published & hosted on NPM. You can reuse your `npm`, `yarn` and `package.json`-related tools to manage them!

Tips & Tricks

Publish

We recommend you to check in your compiled JavaScript output, for its various benefits. If not, then at least consider publishing the JavaScript output by un-ignoring them in your npmignore. This way, your published ReScript package comes with plain JavaScript files that JS users can consume. If your project's good, JS users might not even realize that they've installed a library written in ReScript!

In case your library is only consumed by JS users, you may want to check out our external stdlib configuration as well.

Find Libraries

Search `rescript`-related packages on NPM, or use our Package Index.

If you can't find what you're looking for, remember that **you don't need a wrapper** to use a JS library:

- Most JS data types, such as array and objects, map over cleanly to ReScript and vice-versa.
- You also have access to the familiar JS API.
- You can use a JavaScript library without needing to install dedicated binding libraries. Check the `external` page.

SECTION tuple

---

title: "Tuple" description: "Tuple types and values in ReScript" canonical: "/docs/manual/latest/tuple"

Tuple

Tuples are a ReScript-specific data structure that don't exist in JavaScript. They are:

- immutable
- ordered
- fix-sized at creation time
- heterogeneous (can contain different types of values)

```res example let ageAndName = (24, "Lil' ReScript") let my3dCoordinates = (20.0, 30.5, 100.0)

```js
var ageAndName = [24, "Lil' ReScript"];
var my3dCoordinates = [20.0, 30.5, 100.0];
```

Tuples' types can be used in type annotations as well. Tuple types visually resemble tuples values.

```res prelude let ageAndName: (int, string) = (24, "Lil' ReScript") // a tuple type alias type coord3d = (float, float, float) let my3dCoordinates: coord3d = (20.0, 30.5, 100.0)

```js
var ageAndName = [24, "Lil' ReScript"];
var my3dCoordinates = [20.0, 30.5, 100.0];
```

Note: there's no tuple of size 1. You'd just use the value itself.

Usage

To get a specific member of a tuple, destructure it:

```res example let (*, y, *) = my3dCoordinates // now you've retrieved y

```js
var y = 30.5;
```

The _ means you're ignoring the indicated members of the tuple.

Tuples aren't meant to be updated mutatively. You'd create new ones by destructuring the old ones:

```res example let coordinates1 = (10, 20, 30) let (c1x, *, *) = coordinates1 let coordinates2 = (c1x + 50, 20, 30)

```js
var coordinates1 = [10, 20, 30];
var c1x = 10;
var coordinates2 = [60, 20, 30];
```

Tips & Tricks

You'd use tuples in handy situations that pass around multiple values without too much ceremony. For example, to return many values:

```
let getCenterCoordinates = () => {
  let x = doSomeOperationsHere()
  let y = doSomeMoreOperationsHere()
  (x, y)
}

function getCenterCoordinates(param) {
  var x = doSomeOperationsHere(undefined);
  var y = doSomeMoreOperationsHere(undefined);
  return [x, y];
}
```

Try to keep the usage of tuple **local**. For data structures that are long-living and passed around often, prefer a **record**, which has named fields.

SECTION variant

---

title: "Variant" description: "Variant data structures in ReScript" canonical: "/docs/manual/latest/variant"

Variant

So far, most of ReScript's data structures might look familiar to you. This section introduces an extremely important, and perhaps unfamiliar, data structure: variant.

Most data structures in most languages are about "this **and** that". A variant allows us to express "this **or** that".

```res example type myResponse = | Yes | No | PrettyMuch

let areYouCrushingIt = Yes
```

```js
var areYouCrushingIt = /* Yes */0;
```

`myResponse` is a variant type with the cases `Yes`, `No` and `PrettyMuch`, which are called "variant constructors" (or "variant tag"). The `|` bar separates each constructor.

Note: a variant's constructors need to be capitalized.

Variant Needs an Explicit Definition

If the variant you're using is in a different file, bring it into scope like you'd do for a record:

```res example // Zoo.res type animal = Dog | Cat | Bird
```

```js
// Empty output
```

```
// Example.res
let pet: Zoo.animal = Dog // preferred
// or
let pet2 = Zoo.Dog
```

```
var pet = /* Dog */0;
var pet2 = /* Dog */0;
```

Constructor Arguments

A variant's constructors can hold extra data separated by comma.

```res prelude type account = | None | Instagram(string) | Facebook(string, int)
```

```js
// Empty output
```

Here, `Instagram` holds a `string`, and `Facebook` holds a `string` and an `int`. Usage:

```res example let myAccount = Facebook("Josh", 26) let friendAccount = Instagram("Jenny")
```

```js
var myAccount = {
  TAG: /* Facebook */1,
  _0: "Josh",
  _1: 26
};
```

```
var friendAccount = {
  TAG: /* Instagram */0,
  _0: "Jenny"
};
```

Labeled Variant Payloads (Inline Record)

If a variant payload has multiple fields, you can use a record-like syntax to label them for better readability:

```res example type user = | Number(int) | Id({name: string, password: string})

let me = Id({name: "Joe", password: "123"})
```

```js
var me = {
  TAG: /* Id */1,
  name: "Joe",
  password: "123"
};
```

This is technically called an "inline record", and only allowed within a variant constructor. You cannot inline a record type declaration anywhere else in ReScript.

Of course, you can just put a regular record type in a variant too:

```res example type u = {name: string, password: string} type user = | Number(int) | Id(u)

let me = Id({name: "Joe", password: "123"})
```

```js
var me = {
  TAG: /* Id */1,
  _0: {
    name: "Joe",
    password: "123"
  }
};
```

The output is slightly uglier and less performant than the former.

Variant Type Spreads

Just like with records, it's possible to use type spreads to create new variants from other variants:

```
type a = One | Two | Three
type b = | ...a | Four | Five
```

Type `b` is now:

```
type b = One | Two | Three | Four | Five
```

Type spreads act as a 'copy-paste', meaning all constructors are copied as-is from `a` to `b`. Here are the rules for spreads to work: - You can't overwrite constructors, so the same constructor name can exist in only one place as you spread. This is true even if the constructors are identical. - All variants and constructors must share the same runtime configuration - `@unboxed`, `@tag`, `@as` and so on. - You can't spread types in recursive definitions.

Note that you need a leading `|` if you want to use a spread in the first position of a variant definition.

Pattern Matching On Variant

See the Pattern Matching/Destructuring section later.

JavaScript Output

A variant value compiles to 3 possible JavaScript outputs depending on its type declaration:

- If the variant value is a constructor with no payload, it compiles to a string of the constructor name. Example: `Yes` compiles to `"Yes"`.
- If it's a constructor with a payload, it compiles to an object with the field `TAG` and the field `_0` for the first payload, `_1` for the second payload, etc. The value of `TAG` is the constructor name as string by default, but note that the name of the `TAG` field as well as the string value used for each constructor name can be customized.
- Labeled variant payloads (the inline record trick earlier) compile to an object with the label names instead of `_0`, `_1`, etc. The object will have the `TAG` field as per the previous rule.

Check the output in these examples:

```res example type greeting = Hello | Goodbye let g1 = Hello let g2 = Goodbye

type outcome = Good | Error(string) let o1 = Good let o2 = Error("oops!")

type family = Child | Mom(int, string) | Dad (int) let f1 = Child let f2 = Mom(30, "Jane") let f3 = Dad(32)

type person = Teacher | Student({gpa: float}) let p1 = Teacher let p2 = Student({gpa: 99.5})

type s = {score: float} type adventurer = Warrior(s) | Wizard(string) let a1 = Warrior({score: 10.5}) let a2 = Wizard("Joe")

```js
var g1 = "Hello";

var g2 = "Goodbye";

var o1 = "Good";

var o2 = {
  TAG: "Error",
  _0: "oops!"
};

var f1 = "Child";

var f2 = {
  TAG: "Mom",
  _0: 30,
  _1: "Jane"
};

var f3 = {
  TAG: "Dad",
  _0: 32
};

var p1 = "Teacher";

var p2 = {
  TAG: "Student",
  gpa: 99.5
};

var a1 = {
  TAG: "Warrior",
  _0: {
    score: 10.5
  }
};

var a2 = {
  TAG: "Wizard",
  _0: "Joe"
};
```

Tagged variants

- The `@tag` attribute lets you customize the discriminator (default: `TAG`).
- `@as` attributes control what each variant case is discriminated on (default: the variant case name as string).

Example: Binding to TypeScript enums

```
// direction.ts
/** Direction of the action. */
enum Direction {
  /** The direction is up. */
  Up = "UP",

  /** The direction is down. */
  Down = "DOWN",

  /** The direction is left. */
  Left = "LEFT",

  /** The direction is right. */
  Right = "RIGHT",
}

export const myDirection = Direction.Up;
```

You can bind to the above enums like so:

```
/** Direction of the action. */
type direction =
```
```

```
  | /** The direction is up. */
  @as("UP")
  Up

  | /** The direction is down. */
  @as("DOWN")
  Down

  | /** The direction is left. */
  @as("LEFT")
  Left

  | /** The direction is right. */
  @as("RIGHT")
  Right

@module("./direction.js") external myDirection: direction = "myDirection"
```

Now, this maps 100% to the TypeScript code, including letting us bring over the documentation strings so we get a nice editor experience.

String literals

The same logic is easily applied to string literals from TypeScript, only here the benefit is even larger, because string literals have the same limitations in TypeScript that polymorphic variants have in ReScript:

```
// direction.ts
type direction = "UP" | "DOWN" | "LEFT" | "RIGHT";
```

There's no way to attach documentation strings to string literals in TypeScript, and you only get the actual value to interact with.

Valid `@as` payloads

Here's a list of everything you can put in the `@as` tag of a variant constructor: - A string literal: `@as("success")` - An int: `@as(5)` - A float: `@as(1.5)` - True/false: `@as(true)` and `@as(false)` - Null: `@as(null)` - Undefined: `@as(undefined)`

Untagged variants

With *untagged variants* it is possible to mix types together that normally can't be mixed in the ReScript type system, as long as there's a way to discriminate them at runtime. For example, with untagged variants you can represent a heterogenous array:

```
@unboxed type listItemValue = String(string) | Boolean(bool) | Number(float)

let myArray = [String("Hello"), Boolean(true), Boolean(false), Number(13.37)]
```

Here, each value will be *unboxed* at runtime. That means that the variant payload will be all that's left, the variant case name wrapping the payload itself will be stripped out and the payload will be all that remains.

It, therefore, compiles to this JS:

```
var myArray = ["hello", true, false, 13.37];
```

In the above example, reaching back into the values is as simple as pattern matching on them.

Advanced: Unboxing rules

No overlap in constructors

A variant can be unboxed if no constructors have overlap in their runtime representation.

For example, you can't have `String1(string) | String2(string)` in the same unboxed variant, because there's no way for ReScript to know at runtime which of `String1` or `String2` that `string` belongs to, as it could belong to both. The same goes for two records - even if they have fully different shapes, they're still JavaScript `object` at runtime.

Don't worry - the compiler will guide you and ensure there's no overlap.

What you can unbox

Here's a list of all possible things you can unbox: - `string`: `String(string)` - `float`: `Number(float)`. Notice `int` cannot be unboxed, because JavaScript only has `number` (not actually `int` and `float` like in ReScript) so we can't disambiguate between `float` and `int` at runtime. - `bool`: `Boolean(bool)` - `array<'value>`: `List(array<string>)` - `promise<'value>`: `Promise(promise<string>)` - `Dict.t`: `Object(Dict.t<string>)` - `Date.t`: `Date(Date.t)`. A JavaScript date. - `Blob.t`: `Blob(Blob.t)`. A JavaScript blob. - `File.t`: `File(File.t)`. A JavaScript file. - `RegExp.t`: `RegExp(RegExp.t)`. A JavaScript regexp instance.

Again notice that the constructor names can be anything, what matters is what's in the payload.

> **Under the hood**: Untagged variants uses a combination of JavaScript `typeof` and `instanceof` checks to discern between unboxed constructors at runtime. This means that we could add more things to the list above detailing what can be unboxed, if there are useful enough use cases.

Pattern matching on unboxed variants

Pattern matching works the same on unboxed variants as it does on regular variants. In fact, in the perspective of ReScript's type system there's no difference between untagged and tagged variants. You can do virtually the same things with both. That's the beauty of untagged variants - they're just variants to you as a developer.

Here's an example of pattern matching on an unboxed nullable value that illustrates the above:

```
module Null = {
  @unboxed type t<'a> = Present('a) | @as(null) Null
}

type userAge = {ageNum: Null.t<int>}

type rec user = {
  name: string,
  age: Null.t<userAge>,
  bestFriend: Null.t<user>,
}

let getBestFriendsAge = user =>
  switch user.bestFriend {
  | Present({age: Present({ageNum: Present(ageNum)})}) => Some(ageNum)
  | _ => None
  }
```

No difference to how you'd do with a regular variant. But, the runtime representation is different to a regular variant.

> Notice how `@as` allows us to say that an untagged variant case should map to a specific underlying *primitive*. `Present` has a type variable, so it can hold any type. And since it's an unboxed type, only the payloads `'a` or `null` will be kept at runtime. That's where the magic comes from.

Decoding and encoding JSON idiomatically

With untagged variants, we have everything we need to define a native JSON type:

```
@unboxed
type rec json =
  | @as(null) Null
  | Boolean(bool)
  | String(string)
  | Number(float)
  | Object(Js.Dict.t<json>)
  | Array(array<json>)

let myValidJsonValue = Array([String("Hi"), Number(123.)])
```

Here's an example of how you could write your own JSON decoders easily using the above, leveraging pattern matching:

```
@unboxed
type rec json =
  | @as(null) Null
  | Boolean(bool)
  | String(string)
  | Number(float)
  | Object(Js.Dict.t<json>)
  | Array(array<json>)

type rec user = {
  name: string,
  age: int,
  bestFriend: option<user>,
}

let rec decodeUser = json =>
  switch json {
  | Object(userDict) =>
    switch (
      userDict->Dict.get("name"),
      userDict->Dict.get("age"),
      userDict->Dict.get("bestFriend"),
    ) {
    | (Some(String(name)), Some(Number(age)), Some(maybeBestFriend)) =>
      Some({
        name,
        age: age->Float.toInt,
        bestFriend: maybeBestFriend->decodeUser,
      })
```

```
    | _ => None
    }
  | _ => None
  }

let decodeUsers = json =>
  switch json {
  | Array(array) => array->Array.map(decodeUser)->Array.keepSome
  | _ => []
  }
```

Encoding that same structure back into JSON is also easy:

```
let rec userToJson = user => Object(
  Dict.fromArray([
    ("name", String(user.name)),
    ("age", Number(user.age->Int.toFloat)),
    (
      "bestFriend",
      switch user.bestFriend {
      | None => Null
      | Some(friend) => userToJson(friend)
      },
    ),
  ]),
)

let usersToJson = users => Array(users->Array.map(userToJson))
```

This can be extrapolated to many more cases.

Advanced: Catch-all Constructors

With untagged variants comes a rather interesting capability - catch-all cases are now possible to encode directly into a variant.

Let's look at how it works. Imagine you're using a third party API that returns a list of available animals. You could of course model it as a regular `string`, but given that variants can be used as "typed strings", using a variant would give you much more benefit:

```
type animal = Dog | Cat | Bird

type apiResponse = {
  animal: animal
}

let greetAnimal = (animal: animal) =>
  switch animal {
  | Dog => "Wof"
  | Cat => "Meow"
  | Bird => "Kashiiin"
  }
```

This is all fine and good as long as the API returns `"Dog"`, `"Cat"` or `"Bird"` for `animal`. However, what if the API changes before you have a chance to deploy new code, and can now return `"Turtle"` as well? Your code would break down because the variant `animal` doesn't cover `"Turtle"`.

So, we'll need to go back to `string`, loosing all of the goodies of using a variant, and then do manual conversion into the `animal` variant from `string`, right? Well, this used to be the case before, but not anymore! We can leverage untagged variants to bake in handling of unknown values into the variant itself.

Let's update our type definition first:

```
@unboxed
type animal = Dog | Cat | Bird | UnknownAnimal(string)
```

Notice we've added `@unboxed` and the constructor `UnknownAnimal(string)`. Remember how untagged variants work? You remove the constructors and just leave the payloads. This means that the variant above at runtime translates to this (made up) JavaScript type:

```
type animal = "Dog" | "Cat" | "Bird" | string
```

So, any string not mapping directly to one of the payloadless constructors will now map to the general `string` case.

As soon as we've added this, the compiler complains that we now need to handle this additional case in our pattern match as well. Let's fix that:

```
@unboxed
type animal = Dog | Cat | Bird | UnknownAnimal(string)

type apiResponse = {
  animal: animal
}
```

```
let greetAnimal = (animal: animal) =>
  switch animal {
  | Dog => "Wof"
  | Cat => "Meow"
  | Bird => "Kashiiin"
  | UnknownAnimal(otherAnimal) =>
    `I don't know how to greet animal ${otherAnimal}`
  }
```

```
function greetAnimal(animal) {
  if (!(animal === "Cat" || animal === "Dog" || animal === "Bird")) {
    return "I don't know how to greet animal " + animal;
  }
  switch (animal) {
    case "Dog" :
        return "Wof";
    case "Cat" :
        return "Meow";
    case "Bird" :
        return "Kashiiin";

  }
}
```

There! Now the external API can change as much as it wants, we'll be forced to write all code that interfaces with `animal` in a safe way that handles all possible cases. All of this baked into the variant definition itself, so no need for labor intensive manual conversion.

This is useful in any scenario when you use something enum-style that's external and might change. Additionally, it's also useful when something external has a large number of possible values that are known, but where you only care about a subset of them. With a catch-all case you don't need to bind to all of them just because they can happen, you can safely just bind to the ones you care about and let the catch-all case handle the rest.

Coercion

In certain situations, variants can be coerced to other variants, or to and from primitives. Coercion is always zero cost.

Coercing Variants to Other Variants

You can coerce a variant to another variant if they're identical in runtime representation, and additionally if the variant you're coercing can be represented as the variant you're coercing to.

Here's an example using variant type spreads:

```
type a = One | Two | Three
type b = | ...a | Four | Five

let one: a = One
let four: b = Four

// This works because type `b` can always represent type `a` since all of type `a`'s constructors are spread int
o type `b`
let oneAsTypeB = (one :> b)
```

Coercing Variants to Primitives

Variants that are guaranteed to always be represented by a single primitive at runtime can be coerced to that primitive.

It works with strings, the default runtime representation of payloadless constructors:

```
// Constructors without payloads are represented as `string` by default
type a = One | Two | Three

let one: a = One

// All constructors are strings at runtime, so you can safely coerce it to a string
let oneAsString = (one :> string)
```

If you were to configure all of your construtors to be represented as `int` or `float`, you could coerce to those too:

```
type asInt = | @as(1) One | @as(2) Two | @as(3) Three

let oneInt: asInt = One
let toInt = (oneInt :> int)
```

Advanced: Coercing `string` to Variant

In certain situtations it's possible to coerce a `string` to a variant. This is an advanced technique that you're unlikely to need much, but when you do it's really useful.

You can coerce a `string` to a variant when: - Your variant is `@unboxed` - Your variant has a "catch-all" `string` case

Let's look at an example:

```
@unboxed
type myEnum = One | Two | Other(string)

// Other("Other thing")
let asMyEnum = ("Other thing" :> myEnum)

// One
let asMyEnum = ("One" :> myEnum)
```

This works because the variant is unboxed **and** has a catch-all case. So, if you throw a string at this variant that's not representable by the payloadless constructors, like `"One"` or `"Two"`, it'll *always* end up in `Other(string)`, since that case can represent any `string`.

Tips & Tricks

Be careful not to confuse a constructor carrying 2 arguments with a constructor carrying a single tuple argument:

```res example type account = | Facebook(string, int) // 2 arguments type account2 = | Instagram((string, int)) // 1 argument - happens to be a 2-tuple

```js
// Empty output
```

Variants Must Have Constructors

If you come from an untyped language, you might be tempted to try `type myType = int | string`. This isn't possible in ReScript; you'd have to give each branch a constructor: `type myType = Int(int) | String(string)`. The former looks nice, but causes lots of trouble down the line.

Interop with JavaScript

This section assumes knowledge about our JavaScript interop. Skip this if you haven't felt the itch to use variants for wrapping JS functions yet.

Quite a few JS libraries use functions that can accept many types of arguments. In these cases, it's very tempting to model them as variants. For example, suppose there's a `myLibrary.draw` JS function that takes in either a `number` or a `string`. You might be tempted to bind it like so:

```res example // reserved for internal usage @module("myLibrary") external draw : 'a => unit = "draw"

type animal = | MyFloat(float) | MyString(string)

let betterDraw = (animal) => switch animal { | MyFloat(f) => draw(f) | MyString(s) => draw(s) }

betterDraw(MyFloat(1.5))

```js
var MyLibrary = require("myLibrary");

function betterDraw(animal) {
  MyLibrary.draw(animal._0);
}

betterDraw({
      TAG: "MyFloat",
      _0: 1.5
    });
```

Try not to do that, as this generates extra noisy output. Instead, use the `@unboxed` attribute to guide ReScript to generate more efficient code:

```res example // reserved for internal usage @module("myLibrary") external draw : 'a => unit = "draw"

@unboxed type animal = | MyFloat(float) | MyString(string)

let betterDraw = (animal) => switch animal { | MyFloat(f) => draw(f) | MyString(s) => draw(s) }

betterDraw(MyFloat(1.5))

```js
var MyLibrary = require("myLibrary");

function betterDraw(animal) {
  MyLibrary.draw(animal);
}
```
```

```
MyLibrary.draw(1.5);
```

Alternatively, define two `external`s that both compile to the same JS call:

```res example @module("myLibrary") external drawFloat: float => unit = "draw" @module("myLibrary") external drawString: string => unit = "draw"

```js
// Empty output
```

ReScript also provides a few other ways to do this.

Variant Types Are Found By Field Name

Please refer to this record section. Variants are the same: a function can't accept an arbitrary constructor shared by two different variants. Again, such feature exists; it's called a polymorphic variant. We'll talk about this in the future =).

Design Decisions

Variants, in their many forms (polymorphic variant, open variant, GADT, etc.), are likely *the* feature of a type system such as ReScript's. The aforementioned `option` variant, for example, obliterates the need for nullable types, a major source of bugs in other languages. Philosophically speaking, a problem is composed of many possible branches/conditions. Mishandling these conditions is the majority of what we call bugs. **A type system doesn't magically eliminate bugs; it points out the unhandled conditions and asks you to cover them***. The ability to model "this or that" correctly is crucial.

For example, some folks wonder how the type system can safely eliminate badly formatted JSON data from propagating into their program. They don't, not by themselves! But if the parser returns the `option` type `None | Some(actualData)`, then you'd have to handle the `None` case explicitly in later call sites. That's all there is.

Performance-wise, a variant can potentially tremendously speed up your program's logic. Here's a piece of JavaScript:

```
let data = 'dog'
if (data === 'dog') {
  ...
} else if (data === 'cat') {
  ...
} else if (data === 'bird') {
  ...
}
```

There's a linear amount of branch checking here (`O(n)`). Compare this to using a ReScript variant:

```res example type animal = Dog | Cat | Bird let data = Dog switch data { | Dog => Js.log("Wof") | Cat => Js.log("Meow") | Bird => Js.log("Kashiiin") }

```js
console.log("Wof");

var data = "Dog";
```

The compiler sees the variant, then

1. conceptually turns them into `type animal = "Dog" | "Cat" | "Bird"`
2. compiles `switch` to a constant-time jump table (`O(1)`).

SECTION build-pinned-dependencies

---

title: "Pinned Dependencies" metaTitle: "Pinned Dependencies" description: "Handling multiple packages within one ReScript project with pinned dependencies" canonical: "/docs/manual/latest/build-pinned-dependencies"

Pinned Dependencies

Usually we'd recommend to use ReScript in a single-codebase style by using one `rescript.json` file for your whole codebase.

There are scenarios where you still want to connect and build multiple independent ReScript packages for one main project though (`npm` workspaces-like "monorepos"). This is where `pinned-dependencies` come into play.

Package Types

Before we go into detail, let's first explain all the different package types recognized by the build system:

- Toplevel (this is usually the final app you are building, which has dependencies to other packages)

- Pinned dependencies (these are your local packages that should always rebuild when you build your toplevel, those should be listed in `bs-dependencies` and `pinned-dependencies`)
- Normal dependencies (these are packages that are consumed from npm and listed via `bs-dependencies`)

Whenever a package is being built (`rescript build`), the build system will build the toplevel package with its pinned-dependencies. So any changes made in a pinned dependency will automatically be reflected in the final app.

Build System Package Rules

The build system respects the following rules for each package type:

Toplevel - Warnings reported - Warn-error respected - Builds dev dependencies - Builds pinned dependencies - Runs custom rules - Package-specs like ES6/CommonJS overrides all its dependencies

Pinned dependencies - Warnings reported - Warn-error respected - Ignores pinned dependencies - Builds dev dependencies - Runs custom rules

Normal dependencies - Warnings, warn-error ignored - Ignores dev directories - Ignores pinned dependencies - Ignores custom generator rules

So with that knowledge in mind, let's dive into some more concrete examples to see our pinned dependencies in action.

Examples

Yarn workspaces

Let's assume we have a codebase like this:

```
myproject/
  app/
   - src/App.res
   - rescript.json
  common/
   - src/Header.res
   - rescript.json
  myplugin/
   - src/MyPlugin.res
   - rescript.json
  package.json
```

Our `package.json` file within our codebase root would look like this:

```
{
  "name": "myproject",
  "private": true,
  "workspaces": {
    "packages": [
      "app",
      "common",
      "myplugin"
    ]
  }
}
```

Our `app` folder would be our toplevel package, consuming our `common` and `myplugin` packages as `pinned-dependencies`. The configuration for `app/rescript.json` looks like this:

```
{
  "name": "app",
  "version": "1.0.0",
  "sources": {
    "dir" : "src",
    "subdirs" : true
  },
  /* ... */
  "bs-dependencies": [
    "common",
    "myplugin"
  ],
  "pinned-dependencies": ["common", "myplugin"],
  /* ... */
}
```

Now, whenever we are running `rescript build` within our `app` package, the compiler would always rebuild any changes within its pinned dependencies as well.

Important: ReScript will not rebuild any `pinned-dependencies` in watch mode! This is due to the complexity of file watching, so you'd need to set up your own file-watcher process that runs `rescript build` on specific file changes.

SECTION attribute

---

title: "Attribute (Decorator)" description: "Annotations in ReScript" canonical: "/docs/manual/latest/attribute"

Attribute (Decorator)

Like many other languages, ReScript allows annotating a piece of code to express extra functionality. Here's an example:

```
@inline
let mode = "dev"

let mode2 = mode
```

```
var mode2 = "dev";
```

The `@inline` annotation tells `mode`'s value to be inlined into its usage sites (see output). We call such annotation "attribute" (or "decorator" in JavaScript).

An attribute starts with `@` and goes before the item it annotates. In the above example, it's hooked onto the let binding.

Usage

> **Note:** In previous versions (< 8.3) all our interop related attributes started with a`bs.` prefix (`bs.module`, `bs.val`). Our formatter will automatically drop them in newer ReScript versions.

You can put an attribute almost anywhere. You can even add extra data to them by using them visually like a function call. Here are a few famous attributes (explained in other sections):

```
@@warning("-27")


@unboxed
type a = Name(string)

@val external message: string = "message"

type student = {
  age: int,
  @as("aria-label") ariaLabel: string,
}

@deprecated
let customDouble = foo => foo * 2

@deprecated("Use SomeOther.customTriple instead")
let customTriple = foo => foo * 3
```

1. `@@warning("-27")` is a standalone attribute that annotates the entire file. Those attributes start with `@@`. Here, it carries the data `"-27"`. You can find a full list of all available warnings here.
2. `@unboxed` annotates the type definition.
3. `@val` annotates the `external` statement.
4. `@as("aria-label")` annotates the `ariaLabel` record field.
5. `@deprecated` annotates the `customDouble` expression. This shows a warning while compiling telling consumers to not rely on this method long-term.
6. `@deprecated("Use SomeOther.customTriple instead")` annotates the `customTriple` expression with a string to describe the reason for deprecation.

For a list of all decorators and their usage, please refer to the Syntax Lookup page.

Extension Point

There's a second category of attributes, called "extension points" (a remnant term of our early systems):

```
%raw("var a = 1")
```

```
var a = 1
```

Extension points are attributes that don't *annotate* an item; they *are* the item. Usually they serve as placeholders for the compiler to implicitly substitute them with another item.

Extension points start with `%`. A standalone extension point (akin to a standalone regular attribute) starts with `%%`.

For a list of all extension points and their usage, please refer to the Syntax Lookup page.

SECTION generate-converters-accessors

---

title: "Generate Converters & Helpers" description: "All about the @deriving decorator, and how to generate code from types" canonical: "/docs/manual/latest/generate-converters-accessors"

Generate Converters & Helpers

Note: if you're looking for: - `@deriving(jsConverter)` for records - `@deriving({jsConverter: newType})` for records - `@deriving(jsConverter)` for polymorphic variants

These particular ones are no longer needed. Select a doc version lower than `9.0` in the sidebar to see their old docs.

When using ReScript, you will sometimes come into situations where you want to

- Automatically generate functions that convert between ReScript's internal and JS runtime values (e.g. variants).
- Convert a record type into an abstract type with generated creation, accessor and method functions.
- Generate some other helper functions, such as functions from record attribute names.

You can use the `@deriving` decorator for different code generation scenarios. All different options and configurations will be discussed on this page.

Note: Please be aware that extensive use of code generation might make it harder to understand your programs (since the code being generated is not visible in the source code, and you just need to know what kind of functions / values a decorator generates).

Generate Functions & Plain Values for Variants

Use `@deriving(accessors)` on a variant type to create accessor functions for its constructors.

```
@deriving(accessors)
type action =
  | Click
  | Submit(string)
  | Cancel;
```

```
function submit(param_0) {
  return /* Submit */[param_0];
}

var click = /* Click */0;

var cancel = /* Cancel */1;

exports.click  = click;
exports.submit = submit;
exports.cancel = cancel;
```

Variants constructors with payloads generate functions, payload-less constructors generate plain integers (the internal representation of variants).

Note: - The generated accessors are lower-cased. - You can now use these helpers on the JavaScript side! But don't rely on their actual values please.

Usage

```
let s = submit("hello"); /* gives Submit("hello") */
```

This is useful:

- When you're passing the accessor function as a higher-order function (which plain variant constructors aren't).
- When you'd like the JS side to use these values & functions opaquely and pass you back a variant constructor (since JS has no such thing).

Please note that in case you just want to *pipe a payload into a constructor*, you don't need to generate functions for that. Use the `->` syntax instead, e.g. `"test"->Submit`.

Generate Field Accessors for Records

Use `@deriving(accessors)` on a record type to create accessors for its record field names.

```
@deriving(accessors)
type pet = {name: string}

let pets = [{name: "bob"}, {name: "bob2"}]
```

```
pets
 ->Array.map(name)
 ->Array.joinWith("&")
 ->Js.log
```

```
function name(param) {
  return param.name;
}

var pets = [
  {
    name: "bob"
  },
  {
    name: "bob2"
  }
];

console.log(Belt_Array.map(pets, name).join("&"));
```

Generate Converters for JS Integer Enums and Variants

Use `@deriving(jsConverter)` on a variant type to create converter functions that allow back and forth conversion between JS integer enum and ReScript variant values.

```
@deriving(jsConverter)
type fruit =
  | Apple
  | Orange
  | Kiwi
  | Watermelon;
```

This option causes `jsConverter` to, again, generate functions of the following types:

```
let fruitToJs: fruit => int;

let fruitFromJs: int => option<fruit>;
```

For `fruitToJs`, each fruit variant constructor would map into an integer, starting at 0, in the order they're declared.

For `fruitFromJs`, the return value is an `option`, because not every int maps to a constructor.

You can also attach a `@as(1234)` to each constructor to customize their output.

Usage

```
@deriving(jsConverter)
type fruit =
  | Apple
  | @as(10) Orange
  | @as(100) Kiwi
  | Watermelon

let zero = fruitToJs(Apple) /* 0 */

switch fruitFromJs(100) {
| Some(Kiwi) => Js.log("this is Kiwi")
| _ => Js.log("received something wrong from the JS side")
}
```

Note: by using `@as` here, all subsequent number encoding changes. `Apple` is still `0`, `Orange` is `10`, `Kiwi` is `100` and `Watermelon` is **`101`**!

More Safety

Similar to the JS object \<-> record deriving, you can hide the fact that the JS enum are ints by using the same `newType` option with `@deriving(jsConverter)`:

```
@deriving({jsConverter: newType})
type fruit =
  | Apple
  | @as(100) Kiwi
  | Watermelon;
```

This option causes `@deriving(jsConverter)` to generate functions of the following types:

```
let fruitToJs: fruit => abs_fruit;

let fruitFromJs: abs_fruit => fruit;
```

For `fruitFromJs`, the return value, unlike the previous non-abstract type case, doesn't contain an `option`, because there's no way a bad value can be passed into it; the only creator of `abs_fruit` values is `fruitToJs`!

Usage

```
@deriving({jsConverter: newType})
type fruit =
  | Apple
  | @as(100) Kiwi
  | Watermelon

let opaqueValue = fruitToJs(Apple)

@module("myJSFruits") external jsKiwi: abs_fruit = "iSwearThisIsAKiwi"
let kiwi = fruitFromJs(jsKiwi)

let error = fruitFromJs(100) /* nope, can't take a random int */
```

Convert Record Type to Abstract Record

> **Note**: For ReScript >= v7, we recommend using plain records to compile to JS objects. This feature might still be useful for certain scenarios, but the ergonomics might be worse

Use `@deriving(abstract)` on a record type to expand the type into a creation, and a set of getter / setter functions for fields and methods.

Usually you'd just use ReScript records to compile to JS objects of the same shape. There is still one particular use-case left where the `@deriving(abstract)` convertion is still useful: Whenever you need compile a record with an optional field where the JS object attribute shouldn't show up in the resulting JS when undefined (e.g. `{name: "Carl", age: undefined}` vs `{name: "Carl"}`). Check the Optional Labels section for more infos on this particular scenario.

Usage Example

```
@deriving(abstract)
type person = {
  name: string,
  age: int,
  job: string,
};

@val external john : person = "john";
```

Note: the `person` type is **not** a record! It's a record-looking type that uses the record's syntax and type-checking. The `@deriving(abstract)` decorator turns it into an "abstract type" (aka you don't know what the actual value's shape).

Creation

You don't have to bind to an existing `person` object from the JS side. You can also create such `person` JS object from ReScript's side.

Since `@deriving(abstract)` turns the above `person` record into an abstract type, you can't directly create a person record as you would usually. This doesn't work: `{name: "Joe", age: 20, job: "teacher"}`.

Instead, you'd use the **creation function** of the same name as the record type, implicitly generated by the `@deriving(abstract)` annotation:

```
let joe = person(~name="Joe", ~age=20, ~job="teacher")
```

```
var joe = {
  name: "Joe",
  age: 20,
  job: "teacher"
};
```

Note how in the example above there is no JS runtime overhead.

Rename Fields

Sometimes you might be binding to a JS object with field names that are invalid in ReScript. Two examples would be `{type: "foo"}` (reserved keyword in ReScript) and `{"aria-checked": true}`. Choose a valid field name then use `@as` to circumvent this:

```
@deriving(abstract)
type data = {
  @as("type") type_: string,
  @as("aria-label") ariaLabel: string,
};

let d = data(~type_="message", ~ariaLabel="hello");
```

```
var d = {
  type: "message",
  "aria-label": "hello"
```

```
};
```

Optional Labels

You can omit fields during the creation of the object:

```
@deriving(abstract)
type person = {
  @optional name: string,
  age: int,
  job: string,
};

let joe = person(~age=20, ~job="teacher", ());
```

```
var joe = {
  age: 20,
  job: "teacher"
};
```

Optional values that are not defined, will not show up as an attribute in the resulting JS object. In the example above, you will see that `name` was omitted.

Note that the `@optional` tag turned the `name` field optional. Merely typing `name` as `option<string>` wouldn't work.

Note: now that your creation function contains optional fields, we mandate an unlabeled `()` at the end to indicate that you've finished applying the function.

Accessors

Again, since `@deriving(abstract)` hides the actual record shape, you can't access a field using e.g. `joe.age`. We remediate this by generating getter and setters.

Read

One getter function is generated per `@deriving(abstract)` record type field. In the above example, you'd get 3 functions: `nameGet`, `ageGet`, `jobGet`. They take in a `person` value and return `string`, `int`, `string` respectively:

```
let twenty = ageGet(joe)
```

Alternatively, you can use the Pipe operator (`->`) for a nicer-looking access syntax:

```
let twenty = joe->ageGet
```

If you prefer shorter names for the getter functions, we also support a `light` setting:

```
@deriving({abstract: light})
type person = {
  name: string,
  age: int,
}

let joe = person(~name="Joe", ~age=20)
let joeName = name(joe)
```

The getter functions will now have the same names as the object fields themselves.

Write

A `@deriving(abstract)` value is immutable by default. To mutate such value, you need to first mark one of the abstract record field as `mutable`, the same way you'd mark a normal record as mutable:

```
@deriving(abstract)
type person = {
  name: string,
  mutable age: int,
  job: string,
}
```

Then, a setter of the name `ageSet` will be generated. Use it like so:

```
let joe = person(~name="Joe", ~age=20, ~job="teacher");
ageSet(joe, 21);
```

Alternatively, with the Pipe First syntax:

```
joe->ageSet(21)
```

Methods

You can attach arbitrary methods onto a type (*any* type, as a matter of fact. Not just `@deriving(abstract)` record types). See Object Method in the "Bind to JS Function" section for more infos.

Tips & Tricks

You can leverage `@deriving(abstract)` for finer-grained access control.

Mutability

You can mark a field as mutable in the implementation (`.res`) file, while *hiding* such mutability in the interface file:

```
/* test.res */
@deriving(abstract)
type cord = {
  @optional mutable x: int,
  y: int,
};
```

```
/* test.resi */
@deriving(abstract)
type cord = {
  @optional x: int,
  y: int,
};
```

Tada! Now you can mutate inside your own file as much as you want, and prevent others from doing so!

Hide the Creation Function

Mark the record as `private` to disable the creation function:

```
@deriving(abstract)
type cord = private {
  @optional x: int,
  y: int,
}
```

The accessors are still there, but you can no longer create such data structure. Great for binding to a JS object while preventing others from creating more such object!

Use submodules to prevent naming collisions and binding shadowing

Oftentimes you will have multiple abstract types with similar attributes. Since ReScript will expand all abstract getter, setter and creation functions in the same scope where the type is defined, you will eventually run into value shadowing problems.

For example:

```
@deriving(abstract)
type person = {name: string}

@deriving(abstract)
type cat = {
  name: string,
  isLazy: bool,
};

let person = person(~name="Alice")

/* Error: This expression has type person but an expression was expected
   of type cat */
person->nameGet()
```

To get around this issue, you can use modules to group a type with its related functions and later use them via local open statements:

```
module Person = {
  @deriving(abstract)
  type t = {name: string}
}

module Cat = {
  @deriving(abstract)
  type t = {
    name: string,
    isLazy: bool,
  }
}
```

```
let person = Person.t(~name="Alice")
let cat = Cat.t(~name="Snowball", ~isLazy=true)

/* We can use each nameGet function separately now */
let shoutPersonName = {
  open Person
  person->nameGet->Js.String.toUpperCase
}

/* Note how we use a local `open Cat` expression to
 get access to Cat's nameGet function */
let whisperCatName = {
  open Cat
  cat->nameGet->Js.String.toLowerCase
}
```

Convert External into JS Object Creation Function

Use `@obj` on an `external` binding to create a function that, when called, will evaluate to a JS object with fields corresponding to the function's parameter labels.

This is very handy because you can make some of those labelled parameters optional and if you don't pass them in, the output object won't include the corresponding fields. Thus you can use it to dynamically create objects with the subset of fields you need at runtime.

For example, suppose you need a JavaScript object like this:

```
var homeRoute = {
  type: "GET",
  path: "/",
  action: () => console.log("Home"),
  // options: ...
};
```

But only the first three fields are required; the options field is optional. You can declare the binding function like so:

```
@obj
external route: (
  ~\"type": string,
  ~path: string,
  ~action: list<string> => unit,
  ~options: {..}=?,
  unit,
) => _ = ""
```

Note: the `= ""` part at the end is just a dummy placeholder, due to syntactic limitations. It serves no purpose currently.

This function has four labelled parameters (the fourth one optional), one unlabelled parameter at the end (which we mandate for functions with optional parameters, and one parameter (`\"type"`) that required quoting to avoid clashing with the reserved `type` keyword.

Also of interest is the return type: _, which tells ReScript to automatically infer the full type of the JS object, sparing you the hassle of writing down the type manually!

The function is called like so:

```
let homeRoute = route(
  ~\"type"="GET",
  ~path="/",
  ~action=_ => Js.log("Home"),
  (),
)
```

SECTION function

---

title: "Function" description: "Function syntax in ReScript" canonical: "/docs/manual/latest/function"

Function

Cheat sheet for the full function syntax at the end.

ReScript functions are declared with an arrow and return an expression, just like JS functions. They compile to clean JS functions too.

```res prelude
let greet = (name) => "Hello " ++ name
```

```js
function greet(name) {
  return "Hello " + name;
```

```
}
```

This declares a function and assigns to it the name `greet`, which you can call like so:

```res example greet("world!") // "Hello world!"

```js
greet("world!");
```

Multi-arguments functions have arguments separated by comma:

```res example let add = (x, y, z) => x + y + z add(1, 2, 3) // 6

```js
function add(x, y, z) {
  return (x + y | 0) + z | 0;
}
```

For longer functions, you'd surround the body with a block:

```res example let greetMore = (name) => { let part1 = "Hello" part1 ++ " " ++ name }

```js
function greetMore(name) {
  return "Hello " + name;
}
```

If your function has no argument, just write `let greetMore = () => {...}`.

Labeled Arguments

Multi-arguments functions, especially those whose arguments are of the same type, can be confusing to call.

```
let addCoordinates = (x, y) => {
  // use x and y here
}
// ...
addCoordinates(5, 6) // which is x, which is y?

function addCoordinates(x, y) {
  // use x and y here
}

addCoordinates(5, 6);
```

You can attach labels to an argument by prefixing the name with the `~` symbol:

```
let addCoordinates = (~x, ~y) => {
  // use x and y here
}
// ...
addCoordinates(~x=5, ~y=6)

function addCoordinates(x, y) {
  // use x and y here
}

addCoordinates(5, 6);
```

You can provide the arguments in **any order**:

```
addCoordinates(~y=6, ~x=5)

addCoordinates(5, 6);
```

The `~x` part in the declaration means the function accepts an argument labeled `x` and can refer to it in the function body by the same name. You can also refer to the arguments inside the function body by a different name for conciseness:

```
let drawCircle = (~radius as r, ~color as c) => {
  setColor(c)
  startAt(r, r)
  // ...
}

drawCircle(~radius=10, ~color="red")

function drawCircle(r, c) {
  setColor(c);
  return startAt(r, r);
}

drawCircle(10, "red");
```
```

As a matter of fact, `(~radius)` is just a shorthand for `(~radius as radius)`.

Here's the syntax for typing the arguments:

```
let drawCircle = (~radius as r: int, ~color as c: string) => {
  // code here
}

function drawCircle(r, c) {
  // code here
}
```

Optional Labeled Arguments

Labeled function arguments can be made optional during declaration. You can then omit them when calling the function.

```
// radius can be omitted
let drawCircle = (~color, ~radius=?) => {
  setColor(color)
  switch radius {
  | None => startAt(1, 1)
  | Some(r_) => startAt(r_, r_)
  }
}

var Caml_option = require("./stdlib/caml_option.js");

function drawCircle(color, radius) {
  setColor(color);
  if (radius === undefined) {
    return startAt(1, 1);
  }
  var r_ = Caml_option.valFromOption(radius);
  return startAt(r_, r_);
}
```

When given in this syntax, `radius` is **wrapped** in the standard library's `option` type, defaulting to `None`. If provided, it'll be wrapped with a `Some`. So `radius`'s type value is `None | Some(int)` here.

More on `option` type here.

Signatures and Type Annotations

Functions with optional labeled arguments can be confusing when it comes to signature and type annotations. Indeed, the type of an optional labeled argument looks different depending on whether you're calling the function, or working inside the function body. Outside the function, a raw value is either passed in (`int`, for example), or left off entirely. Inside the function, the parameter is always there, but its value is an option (`option<int>`). This means that the type signature is different, depending on whether you're writing out the function type, or the parameter type annotation. The first being a raw value, and the second being an option.

If we get back to our previous example and both add a signature and type annotations to its argument, we get this:

```
let drawCircle: (~color: color, ~radius: int=?) => unit =
  (~color: color, ~radius: option<int>=?) => {
    setColor(color)
    switch radius {
    | None => startAt(1, 1)
    | Some(r_) => startAt(r_, r_)
    }
  }

function drawCircle(color, radius) {
  setColor(color);
  if (radius !== undefined) {
    return startAt(radius, radius);
  } else {
    return startAt(1, 1);
  }
}
```

The first line is the function's signature, we would define it like that in an interface file (see Signatures). The function's signature describes the types that the **outside world** interacts with, hence the type `int` for `radius` because it indeed expects an `int` when called.

In the second line, we annotate the arguments to help us remember the types of the arguments when we use them **inside** the function's body, here indeed `radius` will be an `option<int>` inside the function.

So if you happen to struggle when writing the signature of a function with optional labeled arguments, try to remember this!

Explicitly Passed Optional

Sometimes, you might want to forward a value to a function without knowing whether the value is `None` or `Some(a)`. Naively, you'd do:

```
let result =
  switch payloadRadius {
  | None => drawCircle(~color)
  | Some(r) => drawCircle(~color, ~radius=r)
  }
```

```
var r = payloadRadius;

var result = r !== undefined
  ? drawCircle(color, Caml_option.valFromOption(r))
  : drawCircle(color);
```

This quickly gets tedious. We provide a shortcut:

```
let result = drawCircle(~color, ~radius=?payloadRadius)
```

```
var result = drawCircle(1, undefined);
```

This means "I understand `radius` is optional, and that when I pass it a value it needs to be an `int`, but I don't know whether the value I'm passing is `None` or `Some(val)`, so I'll pass you the whole `option` wrapper".

Optional with Default Value

Optional labeled arguments can also be provided a default value. In this case, they aren't wrapped in an `option` type.

```
let drawCircle = (~radius=1, ~color) => {
  setColor(color)
  startAt(radius, radius)
}
```

```
function drawCircle(radiusOpt, color) {
  var radius = radiusOpt !== undefined ? radiusOpt : 1;
  setColor(color);
  return startAt(radius, radius);
}
```

Recursive Functions

ReScript chooses the sane default of preventing a function to be called recursively within itself. To make a function recursive, add the `rec` keyword after the `let`:

```res example
let rec neverTerminate = () => neverTerminate()
```

```js
function neverTerminate(_param) {
  while(true) {
    _param = undefined;
    continue ;
  };
}
```

A simple recursive function may look like this:

```res example
// Recursively check every item on the list until one equals the item
// argument. If a match is found, return true, otherwise return false
let rec listHas = (list, item) => switch list {
| list{} => false
| list{a, ...rest} => a === item || listHas(rest, item)
}
```

```js
function listHas(_list, item) {
  while(true) {
    var list = _list;
    if (!list) {
      return false;
    }
    if (list.hd === item) {
      return true;
    }
    _list = list.tl;
    continue ;
  };
}
```

Recursively calling a function is bad for performance and the call stack. However, ReScript intelligently compiles tail recursion into a fast JavaScript loop. Try checking the JS output of the above code!

Mutually Recursive Functions

Mutually recursive functions start like a single recursive function using the `rec` keyword, and then are chained together with `and`:

```res example
let rec callSecond = () => callFirst() and callFirst = () => callSecond()
```

```js
function callSecond(_param) {
  while(true) {
    _param = undefined;
    continue ;
  };
}

function callFirst(_param) {
  while(true) {
    _param = undefined;
    continue ;
  };
}
```

Partial Application

Since 11.0

To partially apply a function, use the explicit `...` syntax.

```
let add = (a, b) => a + b
let addFive = add(5, ...)
```

```
function add(a, b) {
  return a + b | 0;
}

function addFive(extra) {
  return 5 + extra | 0;
}
```

Async/Await

Just as in JS, an async function can be declared by adding `async` before the definition, and `await` can be used in the body of such functions. The output looks like idiomatic JS:

```res example let getUserName = async (userId) => userId

let greetUser = async (userId) => { let name = await getUserName(userId)
"Hello " ++ name ++ "!" }
```

```js
async function greetUser(userId) {
  var name = await getUserName(userId);
  return "Hello " + name + "!";
}
```

The return type of `getUser` is inferred to be `promise<string>`. Similarly, `await getUserName(userId)` returns a `string` when the function returns `promise<string>`. Using `await` outside of an `async` function (including in a non-async callback to an async function) is an error.

Ergonomic error handling

Error handling is done by simply using `try/catch`, or a switch with an `exception` case, just as in functions that are not async. Both JS exceptions and exceptions defined in ReScript can be caught. The compiler takes care of packaging JS exceptions into the builtin `JsError` exception:

```res example exception SomeReScriptException

let somethingThatMightThrow = async () => raise(SomeReScriptException)

let someAsyncFn = async () => { switch await somethingThatMightThrow() { | data => Some(data) | exception JsError(_) => None | exception SomeReScriptException => None } }
```

```js
var SomeReScriptException = /* @__PURE__ */Caml_exceptions.create("Example.SomeReScriptException");

async function someAsyncFn(param) {
  var data;
  try {
    data = await somethingThatMightThrow(undefined);
  }
  catch (raw_exn){
    var exn = Caml_js_exceptions.internalToOCamlException(raw_exn);
    if (exn.RE_EXN_ID === "JsError") {
      return ;
    }
    if (exn.RE_EXN_ID === SomeReScriptException) {
      return ;
```

```
    }
    throw exn;
  }
  return data;
}
```

The ignore() Function

Occasionally you may want to ignore the return value of a function. ReScript provides an `ignore()` function that discards the value of its argument and returns `()`:

```
mySideEffect()->Promise.catch(handleError)->ignore

Js.Global.setTimeout(myFunc, 1000)->ignore
```

```
$$Promise.$$catch(mySideEffect(), function (prim) {
  return handleError(prim);
});

setTimeout(function (prim) {
  myFunc();
}, 1000);
```

Tips & Tricks

Cheat sheet for the function syntaxes:

Declaration

```
// anonymous function
(x, y) => 1
// bind to a name
let add = (x, y) => 1

// labeled
let add = (~first as x, ~second as y) => x + y
// with punning sugar
let add = (~first, ~second) => first + second

// labeled with default value
let add = (~first as x=1, ~second as y=2) => x + y
// with punning
let add = (~first=1, ~second=2) => first + second

// optional
let add = (~first as x=?, ~second as y=?) => switch x {...}
// with punning
let add = (~first=?, ~second=?) => switch first {...}
```

With Type Annotation

```
// anonymous function
(x: int, y: int): int => 1
// bind to a name
let add = (x: int, y: int): int => 1

// labeled
let add = (~first as x: int, ~second as y: int) : int => x + y
// with punning sugar
let add = (~first: int, ~second: int) : int => first + second

// labeled with default value
let add = (~first as x: int=1, ~second as y: int=2) : int => x + y
// with punning sugar
let add = (~first: int=1, ~second: int=2) : int => first + second

// optional
let add = (~first as x: option<int>=?, ~second as y: option<int>=?) : int => switch x {...}
// with punning sugar
// note that the caller would pass an `int`, not `option<int>`
// Inside the function, `first` and `second` are `option<int>`.
let add = (~first: option<int>=?, ~second: option<int>=?) : int => switch first {...}
```

Application

```
add(x, y)

// labeled
add(~first=1, ~second=2)
// with punning sugar
add(~first, ~second)
```

```
// application with default value. Same as normal application
add(~first=1, ~second=2)

// explicit optional application
add(~first=?Some(1), ~second=?Some(2))
// with punning
add(~first?, ~second?)
```

With Type Annotation

```
// labeled
add(~first=1: int, ~second=2: int)
// with punning sugar
add(~first: int, ~second: int)

// application with default value. Same as normal application
add(~first=1: int, ~second=2: int)

// explicit optional application
add(~first=?Some(1): option<int>, ~second=?Some(2): option<int>)
// no punning sugar when you want to type annotate
```

Standalone Type Signature

```
// first arg type, second arg type, return type
type add = (int, int) => int

// labeled
type add = (~first: int, ~second: int) => int

// labeled
type add = (~first: int=?, ~second: int=?, unit) => int
```

In Interface Files

To annotate a function from the implementation file (`.res`) in your interface file (`.resi`):

```
res sig let add: (int, int) => int
```

The type annotation part is the same as the previous section on With Type Annotation.

Don't confuse `let add: myType` with `type add = myType`. When used in `.resi` interface files, the former exports the binding `add` while annotating it as type `myType`. The latter exports the type `add`, whose value is the type `myType`.

SECTION converting-from-js

---

title: "Converting from JS" description: "How to convert to ReScript with an existing JS codebase" canonical: "/docs/manual/latest/converting-from-js"

Converting from JS

ReScript offers a unique project conversion methodology which: - Ensures minimal disruption to your teammates (very important!). - Remove the typical friction of verifying conversion's correctness and performance guarantees. - Doesn't force you to search for pre-made binding libraries made by others. **ReScript doesn't need the equivalent of TypeScript's `DefinitelyTyped`.**

Step 1: Install ReScript

Run `npm install rescript` on your project, then imitate our New Project workflow by adding a `rescript.json` at the root. Then start `npx rescript build -w`.

Step 2: Copy Paste the Entire JS File

Let's work on converting a file called `src/main.js`.

```
const school = require('school');

const defaultId = 10;

function queryResult(usePayload, payload) {
  if (usePayload) {
    return payload.student;
  } else {
    return school.getStudentById(defaultId);
```

```
  }
}
```

First, copy the entire file content over to a new file called `src/Main.res` by using our `%%raw` JS embedding trick:

``res example %%raw(const school = require('school');

const defaultId = 10;

function queryResult(usePayload, payload) { if (usePayload) { return payload.student; } else { return school.getStudentById(defaultId); } } `)

```js
// Generated by ReScript, PLEASE EDIT WITH CARE
'use strict';

const school = require('school');

const defaultId = 10;

function queryResult(usePayload, payload) {
  if (usePayload) {
    return payload.student;
  } else {
    return school.getStudentById(defaultId);
  }
}

/*  Not a pure module */
```

Add this file to `rescript.json`:

```
  "sources": {
    "dir" : "src",
    "subdirs" : true
  },
```

Open an editor tab for `src/Main.bs.js`. Do a command-line `diff -u src/main.js src/Main.bs.js`. Aside from whitespaces, you should see only minimal, trivial differences. You're already a third of the way done!

Always make sure that at each step, you keep the ReScript output `.bs.js` file open to compare against the existing JavaScript file. Our compilation output is very close to your hand-written JavaScript; you can simply eye the difference to catch conversion bugs!

Step 3: Extract Parts into Idiomatic ReScript

Let's turn the `defaultId` variable into a ReScript let-binding:

```res example let defaultId = 10

%%raw(` const school = require('school');

function queryResult(usePayload, payload) { if (usePayload) { return payload.student; } else { return school.getStudentById(defaultId); } } `)

```js
// Generated by ReScript, PLEASE EDIT WITH CARE
'use strict';

const school = require('school');

function queryResult(usePayload, payload) {
  if usePayload {
    return payload.student
  } else {
    return school.getStudentById(defaultId)
  }
}

var defaultId = 10;

exports.defaultId = defaultId;
/*  Not a pure module */
```

Check the output. Diff it. Code still works. Moving on! Extract the function:

```
%%raw(`
const school = require('school');
`)

let defaultId = 10

let queryResult = (usePayload, payload) => {
```
```

```
  if usePayload {
    payload.student
  } else {
    school.getStudentById(defaultId)
  }
}
```

Format the code: `./node_modules/.bin/rescript format src/Main.res`.

We have a type error: "The record field student can't be found". That's fine! **Always ensure your code is syntactically valid first**. Fixing type errors comes later.

Step 4: Add `externals`, Fix Types

The previous type error is caused by `payload`'s record declaration (which supposedly contains the field `student`) not being found. Since we're trying to convert as quickly as possible, let's use our object feature to avoid needing type declaration ceremonies:

```
%%raw(`
const school = require('school');
`)

let defaultId = 10

let queryResult = (usePayload, payload) => {
  if usePayload {
    payload["student"]
  } else {
    school["getStudentById"](. defaultId)
  }
}
```

Now this triggers the next type error, that `school` isn't found. Let's use `external` to bind to that module:

```res example
@module external school: 'whatever = "school"

let defaultId = 10

let queryResult = (usePayload, payload) => {
  if usePayload {
    payload["student"]
  } else {
    school["getStudentById"](. defaultId)
  }
}
```

```js
// Generated by ReScript, PLEASE EDIT WITH CARE
'use strict';

var School = require("school");

function queryResult(usePayload, payload) {
  if (usePayload) {
    return payload.student;
  } else {
    return School.getStudentById(10);
  }
}

var defaultId = 10;

exports.defaultId = defaultId;
exports.queryResult = queryResult;
/* school Not a pure module */
```

We hurrily typed `school` as a polymorphic `'whatever` and let its type be inferred by its usage below. The inference is technically correct, but within the context of bringing it a value from JavaScript, slightly dangerous. This is just the interop trick we've shown in the `external` page.

Anyway, the file passes the type checker again. Check the `.bs.js` output, diff with the original `.js`; we've now converted a file over to ReScript!

Now, you can delete the original, hand-written `main.js` file, and grep the files importing `main.js` and change them to importing `Main.bs.js`.

(Optional) Step 5: Cleanup

If you prefer more advanced, rigidly typed `payload` and `school`, feel free to do so:

```res example
type school
type student
type payload = { student: student }

@module external school: school = "school"
@send external getStudentById: (school, int) => student = "getStudentById"
```

let defaultId = 10

let queryResult = (usePayload, payload) => { if usePayload { payload.student } else { school->getStudentById(defaultId) } }

```js
// Generated by ReScript, PLEASE EDIT WITH CARE
'use strict';

var School = require("school");

function queryResult(usePayload, payload) {
  if (usePayload) {
    return payload.student;
  } else {
    return School.getStudentById(10);
  }
}

var defaultId = 10;

exports.defaultId = defaultId;
exports.queryResult = queryResult;
/* school Not a pure module */
```

We've: - introduced an opaque types for `school` and `student` to prevent misusages their values - typed the payload as a record with only the `student` field - typed `getStudentById` as the sole method of `student`

Check that the `.bs.js` output didn't change. How rigidly to type your JavaScript code is up to you; we recommend not typing them too elaborately; it's sometime an endless chase, and produces diminishing returns, especially considering that the elaborate-ness might turn off your potential teammates.

Tips & Tricks

In the same vein of idea, **resist the urge to write your own wrapper functions for the JS code you're converting**. Use `externals`, which are guaranteed to be erased in the output. And avoid trying to take the occasion to convert JS data structures into ReScript-specific data structures like variant or list. **This isn't the time for that**.

The moment you produce extra conversion code in the output, your skeptical teammate's mental model might switch from "I recognize this output" to "this conversion might be introducing more problems than it solves. Why are we testing ReScript again?". Then you've lost.

Conclusion

- Paste the JS code into a new ReScript file as embedded raw JS code.
- Compile and keep the output file open. Check and diff against original JS file. Free regression tests.
- Always make sure your file is syntactically valid. Don't worry about fixing types before that.
- (Ab)use object accesses to quickly convert things over.
- Optionally clean up the types for robustness.
- Don't go overboard and turn off your boss and fellow teammates.
- Proudly display that you've conserved the semantics and performance characteristics during the conversion by showing your teammates the eerily familiar output.
- Get promoted for introducing a new technology the safer, mature way.

SECTION pattern-matching-destructuring

---

title: "Pattern Matching / Destructuring" description: "Pattern matching and destructuring complex data structures in ReScript" canonical: "/docs/manual/latest/pattern-matching-destructuring"

Pattern Matching / Destructuring

One of ReScript's **best** feature is our pattern matching. Pattern matching combines 3 brilliant features into one:

- Destructuring.
- `switch` based on shape of data.
- Exhaustiveness check.

We'll dive into each aspect below.

Destructuring

Even JavaScript has destructuring, which is "opening up" a data structure to extract the parts we want and assign variable names to them:

```res example let coordinates = (10, 20, 30) let (x, , ) = coordinates Js.log(x) // 10

```js
var coordinates = [10, 20, 30];
var x = 10;
console.log(10);
```

Destructuring works with most built-in data structures:

```
// Record
type student = {name: string, age: int}
let student1 = {name: "John", age: 10}
let {name} = student1 // "John" assigned to `name`

// Variant
type result =
  | Success(string)
let myResult = Success("You did it!")
let Success(message) = myResult // "You did it!" assigned to `message`
```

```
var student1 = {
  name: "John",
  age: 10
};
var name = "John";

var myResult = /* Success */{
  _0: "You did it!"
};
var message = "You did it!"

var myArray = [1, 2, 3];
if (myArray.length !== 2) {
  throw {
    RE_EXN_ID: "Match_failure",
    _1: [
      "playground.res",
      14,
      4
    ],
    Error: new Error()
  };
}
var item1 = myArray[0];
var item2 = myArray[1];

var myList = {
  hd: 1,
  tl: {
    hd: 2,
    tl: {
      hd: 3,
      tl: /* [] */0
    }
  }
};
// ...
```

You can also use destructuring anywhere you'd usually put a binding:

```res example type result = | Success(string) let displayMessage = (Success(m)) => { // we've directly extracted the success message // string by destructuring the parameter Js.log(m) } displayMessage(Success("You did it!"))

```js
function displayMessage(m) {
  console.log(m._0);
}

displayMessage(/* Success */{
  _0: "You did it!"
});
```

For a record, you can rename the field while destructuring:

```
let {name: n} = student1 // "John" assigned to `n`
```

```
var n = "John";
```

You *can* in theory destructure array and list at the top level too:

```
let myArray = [1, 2, 3]
let [item1, item2, _] = myArray
// 1 assigned to `item1`, 2 assigned to `item2`, 3rd item ignored

let myList = list{1, 2, 3}
```

```
let list{head, ...tail} = myList
// 1 assigned to `head`, `list{2, 3}` assigned to tail
```

But the array example is **highly disrecommended** (use tuple instead) and the list example will error on you. They're only there for completeness' sake. As you'll see below, the proper way of using destructuring array and list is using `switch`.

`switch` Based on Shape of Data

While the destructuring aspect of pattern matching is nice, it doesn't really change the way you think about structuring your code. One paradigm-changing way of thinking about your code is to execute some code based on the shape of the data.

Consider a variant:

```res prelude type payload = | BadResult(int) | GoodResult(string) | NoResult

```
```js
// Empty output
```

We'd like to handle each of the 3 cases differently. For example, print a success message if the value is `GoodResult(...)`, do something else when the value is `NoResult`, etc.

In other languages, you'd end up with a series of if-elses that are hard to read and error-prone. In ReScript, you can instead use the supercharged `switch` pattern matching facility to destructure the value while calling the right code based on what you destructured:

```res example let data = GoodResult("Product shipped!") switch data { | GoodResult(theMessage) => Js.log("Success! " ++ theMessage) | BadResult(errorCode) => Js.log("Something's wrong. The error code is: " ++ Js.Int.toString(errorCode)) | NoResult => Js.log("Bah.") }

```
```js
var data = {
  TAG: /* GoodResult */1,
  _0: "Product shipped!"
};

if (typeof data === "number") {
  console.log("Bah.");
} else if (data.TAG === /* BadResult */ 0) {
  console.log("Something's wrong. The error code is: " + "Product shipped!".toString());
} else {
  console.log("Success! Product shipped!");
}
```

In this case, `message` will have the value `"Success! Product shipped!"`.

Suddenly, your if-elses that messily checks some structure of the value got turned into a clean, compiler-verified, linear list of code to execute based on exactly the shape of the value.

Complex Examples

Here's a real-world scenario that'd be a headache to code in other languages. Given this data structure:

```res prelude type status = Vacations(int) | Sabbatical(int) | Sick | Present type reportCard = {passing: bool, gpa: float} type student = {name: string, status: status, reportCard: reportCard} type person = | Teacher({name: string, age: int}) | Student(student)

```
```js
// Empty output
```

Imagine this requirement:

- Informally greet a person who's a teacher and if his name is Mary or Joe.
- Greet other teachers formally.
- If the person's a student, congratulate him/her score if they passed the semester.
- If the student has a gpa of 0 and is on vacations or sabbatical, display a different message.
- A catch-all message for a student.

ReScript can do this easily!

```res prelude let person1 = Teacher({name: "Jane", age: 35})

let message = switch person1 { | Teacher({name: "Mary" | "Joe"}) => `Hey, still going to the party on Saturday?` | Teacher({name}) => // this is matched only if `name` isn't "Mary" or "Joe" `Hello ${name}.` | Student({name, reportCard: {passing: true, gpa}}) => `Congrats ${name}, nice GPA of ${Js.Float.toString(gpa)} you got there!` | Student({ reportCard: {gpa: 0.0}, status: Vacations(daysLeft) | Sabbatical(daysLeft) }) => `Come back in ${Js.Int.toString(daysLeft)} days!` | Student({status: Sick}) => `How are you feeling?` | Student({name}) => `Good luck next semester ${name}!` }

```
```js
var person1 = {
```

```
  TAG: /* Teacher */0,
  name: "Jane",
  age: 35
};

var message;

if (person1.TAG) {
  var match$1 = person1.status;
  var name = person1.name;
  var match$2 = person1.reportCard;
  message = match$2.passing
    ? "Congrats " +
      name +
      ", nice GPA of " +
      match$2.gpa.toString() +
      " you got there!"
    : typeof match$1 === "number"
    ? match$1 !== 0
      ? "Good luck next semester " + name + "!"
      : "How are you feeling?"
    : person1.reportCard.gpa !== 0.0
    ? "Good luck next semester " + name + "!"
    : "Come back in " + match$1._0.toString() + " days!";
} else {
  var name$1 = person1.name;
  switch (name$1) {
    case "Joe":
    case "Mary":
      message = "Hey, still going to the party on Saturday?";
      break;
    default:
      message = "Hello " + name$1 + ".";
  }
}
```

Note how we've: - drilled deep down into the value concisely - using a **nested pattern check** `"Mary" | "Joe"` and `Vacations | Sabbatical` - while extracting the `daysLeft` number from the latter case - and assigned the greeting to the binding `message`.

Here's another example of pattern matching, this time on an inline tuple.

```
type animal = Dog | Cat | Bird
let categoryId = switch (isBig, myAnimal) {
| (true, Dog) => 1
| (true, Cat) => 2
| (true, Bird) => 3
| (false, Dog | Cat) => 4
| (false, Bird) => 5
}
```

```
var categoryId = isBig ? (myAnimal + 1) | 0 : myAnimal >= 2 ? 5 : 4;
```

Note how pattern matching on a tuple is equivalent to a 2D table:

isBig \ myAnimal	**Dog**	**Cat**	**Bird**
true	1	2	3
false	4	4	5

Fall-Through Patterns

The nested pattern check, demonstrated in the earlier `person` example, also works at the top level of a `switch`:

```res prelude let myStatus = Vacations(10)

switch myStatus { | Vacations(days) | Sabbatical(days) => Js.log(`Come back in ${Js.Int.toString(days)} days!`) | Sick | Present => Js.log("Hey! How are you?") }
```

```
```js
var myStatus = {
  TAG: /* Vacations */0,
  _0: 10
};

if (typeof myStatus === "number") {
  console.log("Hey! How are you?");
} else {
  console.log("Come back in " + (10).toString() + " days!");
}
```

Having multiple cases fall into the same handling can clean up certain types of logic.
```

Ignore Part of a Value

If you have a value like `Teacher(payload)` where you just want to pattern match on the `Teacher` part and ignore the `payload` completely, you can use the _ wildcard like this:

```res example switch person1 { | Teacher(*) => *Js.log("Hi teacher")* | *Student*() => Js.log("Hey student") }

```js
if (person1.TAG) {
  console.log("Hey student");
} else {
  console.log("Hi teacher");
}
```

_ also works at the top level of the `switch`, serving as a catch-all condition:

```res example switch myStatus { | Vacations(_) => Js.log("Have fun!") | _ => Js.log("Ok.") }

```js
if (typeof myStatus === "number" || myStatus.TAG) {
  console.log("Ok.");
} else {
  console.log("Have fun!");
}
```

Do not abuse a top-level catch-all condition. Instead, prefer writing out all the cases:

```res example switch myStatus { | Vacations(*) => *Js.log("Have fun!")* | *Sabbatical*() | Sick | Present => Js.log("Ok.") }

```js
if (typeof myStatus === "number" || myStatus.TAG) {
  console.log("Ok.");
} else {
  console.log("Have fun!");
}
```

Slightly more verbose, but a one-time writing effort. This helps when you add a new variant case e.g. `Quarantined` to the `status` type and need to update the places that pattern match on it. A top-level wildcard here would have accidentally and silently continued working, potentially causing bugs.

If Clause

Sometime, you want to check more than the shape of a value. You want to also run some arbitrary check on it. You might be tempted to write this:

```res example switch person1 { | Teacher(_) => () // do nothing | Student({reportCard: {gpa}}) => if gpa < 0.5 { Js.log("What's happening") } else { Js.log("Heyo") } }

```js
if (person1.TAG) {
  if (person1.reportCard.gpa < 0.5) {
    console.log("What's happening");
  } else {
    console.log("Heyo");
  }
}
```

`switch` patterns support a shortcut for the arbitrary `if` check, to keep your pattern linear-looking:

```res example switch person1 { | Teacher(*) => () // do nothing | Student({reportCard: {gpa}}) if gpa < 0.5 => Js.log("What's happening")* | *Student*() => // fall-through, catch-all case Js.log("Heyo") }

```js
if (person1.TAG) {
  if (person1.reportCard.gpa < 0.5) {
    console.log("What's happening");
  } else {
    console.log("Heyo");
  }
}
```

Note: Rescript versions < 9.0 had a `when` clause, not an `if` clause.Â Rescript 9.0 changed `when` to `if`.Â (`when` may still work, but is deprecated.)

Match on Exceptions

If the function throws an exception (covered later), you can also match on *that*, in addition to the function's normally returned values.

```
switch List.find(i => i === theItem, myItems) {
```
```

```
| item => Js.log(item)
| exception Not_found => Js.log("No such item found!")
}
```

```
var exit = 0;

var item;

try {
  item = List.find(function(i) {
    return i === theItem;
  }, myItems);
  exit = 1;
}
catch (raw_exn){
  var exn = Caml_js_exceptions.internalToOCamlException(raw_exn);
  if (exn.RE_EXN_ID === "Not_found") {
    console.log("No such item found!");
  } else {
    throw exn;
  }
}

if (exit === 1) {
  console.log(item);
}
```

Match on Array

```res example let students = ["Jane", "Harvey", "Patrick"] switch students { | [] => Js.log("There are no students") | [student1] => Js.log("There's a single student here: " ++ student1) | manyStudents => // display the array of names Js.log2("The students are: ", manyStudents) }
```

```js
var students = ["Jane", "Harvey", "Patrick"];

var len = students.length;

if (len !== 1) {
  if (len !== 0) {
    console.log("The students are: ", students);
  } else {
    console.log("There are no students");
  }
} else {
  var student1 = students[0];
  console.log("There's a single student here: " + student1);
}
```

Match on List

Pattern matching on list is similar to array, but with the extra feature of extracting the tail of a list (all elements except the first one):

```res example let rec printStudents = (students) => { switch students { | list{} => () // done | list{student} => Js.log("Last student: " ++ student) | list{student1, ...otherStudents} => Js.log(student1) printStudents(otherStudents) } } printStudents(list{"Jane", "Harvey", "Patrick"})
```

```js
function printStudents(_students) {
  while(true) {
    var students = _students;
    if (!students) {
      return;
    }
    var otherStudents = students.tl;
    var student = students.hd;
    if (otherStudents) {
      console.log(student);
      _students = otherStudents;
      continue;
    }
    console.log("Last student: " + student);
    return;
  };
}

printStudents({
  hd: "Jane",
  tl: {
    hd: "Harvey",
    tl: {
      hd: "Patrick",
      tl: /* [] */0
```

```
    }
  }
});
```

Small Pitfall

Note: you can only pass literals (i.e. concrete values) as a pattern, not let-binding names or other things. The following doesn't work as expected:

```res example let coordinates = (10, 20, 30) let centerY = 20 switch coordinates { | (x, *centerY,* ) => Js.log(x) }

```js
var coordinates = [10, 20, 30];
var centerY = 20;

console.log(10);
```

A first time ReScript user might accidentally write that code, assuming that it's matching on `coordinates` when the second value is of the same value as `centerY`. In reality, this is interpreted as matching on coordinates and assigning the second value of the tuple to the name `centerY`, which isn't what's intended.

Exhaustiveness Check

As if the above features aren't enough, ReScript also provides arguably the most important pattern matching feature: **compile-time check of missing patterns**.

Let's revisit one of the above examples:

```
let message = switch person1 {
| Teacher({name: "Mary" | "Joe"}) =>
  `Hey, still going to the party on Saturday?`
| Student({name, reportCard: {passing: true, gpa}}) =>
  `Congrats ${name}, nice GPA of ${Js.Float.toString(gpa)} you got there!`
| Student({
    reportCard: {gpa: 0.0},
    status: Vacations(daysLeft) | Sabbatical(daysLeft)
  }) =>
  `Come back in ${Js.Int.toString(daysLeft)} days!`
| Student({status: Sick}) =>
  `How are you feeling?`
| Student({name}) =>
  `Good luck next semester ${name}!`
}
```

```
if (person1.TAG) {
  var match$1 = person1.status;
  var name = person1.name;
  var match$2 = person1.reportCard;
  if (match$2.passing) {
    "Congrats " + name + ", nice GPA of " + match$2.gpa.toString() + " you got there!";
  } else if (typeof match$1 === "number") {
    if (match$1 !== 0) {
      "Good luck next semester " + name + "!";
    } else {
      "How are you feeling?";
    }
  } else if (person1.reportCard.gpa !== 0.0) {
    "Good luck next semester " + name + "!";
  } else {
    "Come back in " + match$1._0.toString() + " days!";
  }
} else {
  switch (person1.name) {
    case "Joe":
    case "Mary":
      break;
    default:
      throw {
        RE_EXN_ID: "Match_failure",
        _1: [
          "playground.res",
          13,
          0
        ],
        Error: new Error()
      };
  }
}
```

Did you see what we removed? This time, we've omitted the handling of the case where `person1` is `Teacher({name})` when `name` isn't Mary or Joe.
```

Failing to handle every scenario of a value likely constitutes the majority of program bugs out there. This happens very often when you refactor a piece of code someone else wrote. Fortunately for ReScript, the compiler will tell you so:

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Some({name: ""})
```

BAM! You've just erased an entire category of important bugs before you even ran the code. In fact, this is how most of nullable values is handled:

```res example let myNullableValue = Some(5)

switch myNullableValue { | Some(_v) => Js.log("value is present") | None => Js.log("value is absent") }
```

```js
var myNullableValue = 5;

if (myNullableValue !== undefined) {
  console.log("value is present");
} else {
  console.log("value is absent");
}
```

If you don't handle the `None` case, the compiler warns. No more `undefined` bugs in your code!

Conclusion & Tips & Tricks

Hopefully you can see how pattern matching is a game changer for writing correct code, through the concise destructuring syntax, the proper conditions handling of `switch`, and the static exhaustiveness check.

Below is some advice:

Avoid using the wildcard _ unnecessarily. Using the wildcard _ will bypass the compiler's exhaustiveness check. Consequently, the compiler will not be able to notify you of probable errors when you add a new case to a variant. Try only using _ against infinite possibilities, e.g. string, int, etc.

Use the `if` clause sparingly.

Flatten your pattern-match whenever you can. This is a real bug remover. Here's a series of examples, from worst to best:

```res example let optionBoolToBool = opt => { if opt == None { false } else if opt === Some(true) { true } else { false } }
```

```js
function optionBoolToBool(opt) {
  if (opt === undefined) {
    return false;
  } else {
    return opt === true;
  }
}
```

Now that's just silly =). Let's turn it into pattern-matching:

```res example let optionBoolToBool = opt => { switch opt { | None => false | Some(a) => a ? true : false } }
```

```js
function optionBoolToBool(opt) {
  if (opt !== undefined && opt) {
    return true;
  } else {
    return false;
  }
}
```

Slightly better, but still nested. Pattern-matching allows you to do this:

```res example let optionBoolToBool = opt => { switch opt { | None => false | Some(true) => true | Some(false) => false } }
```

```js
function optionBoolToBool(opt) {
  if (opt !== undefined && opt) {
    return true;
  } else {
    return false;
  }
}
```

Much more linear-looking! Now, you might be tempted to do this:

```res example let optionBoolToBool = opt => { switch opt { | Some(true) => true | _ => false } }
```

```js
function optionBoolToBool(opt) {
  if (opt !== undefined && opt) {
    return true;
  } else {
    return false;
  }
}
```

Which is much more concise, but kills the exhaustiveness check mentioned above; refrain from using that. This is the best:

```res example let optionBoolToBool = opt => { switch opt { | Some(trueOrFalse) => trueOrFalse | None => false } }

```js
function optionBoolToBool(opt) {
  if (opt !== undefined) {
    return opt;
  } else {
    return false;
  }
}
```

Pretty darn hard to make a mistake in this code at this point! Whenever you'd like to use an if-else with many branches, prefer pattern matching instead. It's more concise and performant too.

SECTION overview

---

title: "Overview" metaTitle: "Language Features Overview" description: "A quick overview on ReScript's syntax" canonical: "/docs/manual/latest/overview"

Overview

Comparison to JS

Semicolon

JavaScript	ReScript
Rules enforced by linter/formatter	No semicolon needed!

Comments

JavaScript	ReScript
`// Line comment`	Same
`/* Comment */`	Same
`/** Doc Comment */`	`/** Before Types/Values */`
	`/*** Standalone Doc Comment */`

Variable

JavaScript	ReScript
`const x = 5;`	`let x = 5`
`var x = y;`	No equivalent (thankfully)
`let x = 5; x = x + 1;`	`let x = ref(5); x := x.contents + 1`

String & Character

JavaScript	ReScript
`"Hello world!"`	Same
`'Hello world!'`	Strings must use `"`
`"hello " + "world"`	`"hello " ++ "world"`
`` `hello ${message}` ``	Same

Boolean

JavaScript	ReScript
`true`, `false`	Same
```

JavaScript	ReScript
`!true`	Same
`||`, `&&`, `<=`, `>=`, `<`, `>`	Same
`a === b`, `a !== b`	Same
No deep equality (recursive compare)	`a == b`, `a != b`
`a == b`	No equality with implicit casting (thankfully)

Number

JavaScript	ReScript
`3`	Same *
`3.1415`	Same
`3 + 4`	Same
`3.0 + 4.5`	`3.0 +. 4.5`
`5 % 3`	`mod(5, 3)`

* JS has no distinction between integer and float.

Object/Record

JavaScript	ReScript
no types	`type point = {x: int, mutable y: int}`
`{x: 30, y: 20}`	Same
`point.x`	Same
`point.y = 30;`	Same
`{...point, x: 30}`	Same

Array

JavaScript	ReScript
`[1, 2, 3]`	Same
`myArray[1] = 10`	Same
`[1, "Bob", true]`	`(1, "Bob", true)` *

* Heterogenous arrays in JS are disallowed for us. Use tuple instead.

Null

JavaScript	ReScript
`null`, `undefined`	`None` *

* Again, only a spiritual equivalent; we don't have nulls, nor null bugs! But we do have an `option` type for when you actually need nullability.

Function

JavaScript	ReScript
`arg => retVal`	Same
`function named(arg) {...}`	`let named = (arg) => {...}`
`const f = function(arg) {...}`	`let f = (arg) => {...}`
`add(4, add(5, 6))`	Same

Async Function / Await

JavaScript	ReScript
`async (arg) => {...}`	Same
`async function named(arg) {...}`	`let named = async (arg) => {...}`
`await somePromise`	Same
`async (arg): Promise<string> => {...}`	`async (): string => {...}` (note the return type)

Blocks

JavaScript	ReScript
`` const myFun = (x, y) => { const doubleX = x + x; const doubleY = y + y; return doubleX + doubleY }; ```	``` let myFun = (x, y) => { let doubleX = x + x let doubleY = y + y doubleX + doubleY } ```

If-else

JavaScript	ReScript
`if (a) {b} else {c}`	`if a {b} else {c}` *
`a ? b : c`	Same
`switch`	`switch` but super-powered pattern matching!

* Our conditionals are always expressions! You can write `let result = if a {"hello"} else {"bye"}`

Destructuring

JavaScript	ReScript
`const {a, b} = data`	`let {a, b} = data`
`const [a, b] = data`	`let [a, b] = data` *
`const {a: aa, b: bb} = data`	`let {a: aa, b: bb} = data`

* Gives good compiler warning that `data` might not be of length 2.

Loop

JavaScript	ReScript
`for (let i = 0; i <= 10; i++) {...}`	`for i in 0 to 10 {...}`
`for (let i = 10; i >= 0; i--) {...}`	`for i in 10 downto 0 {...}`
`while (true) {...}`	`while true {...}`

JSX

JavaScript	ReScript
`<Comp message="hi" onClick={handler} />`	Same
`<Comp message={message} />`	`<Comp message />` *
`<input checked />`	`<input checked=true />`
No children spread	`<Comp>...children</Comp>`

* Argument punning!

Exception

JavaScript	ReScript
`throw new SomeError(...)`	`raise(SomeError(...))`
`try {a} catch (Err) {...} finally {...}`	`try a catch { | Err => ...}` *

* No finally.

Blocks

The last expression of a block delimited by `{}` implicitly returns (including function body). In JavaScript, this can only be simulated via an immediately-invoked function expression (since function bodies have their own local scope).

JavaScript	ReScript
`` let result = (function() { const x = 23; const y = 34; return x + y; })(); ```	``` let result = { let x = 23 let y = 34 x + y } ```

Common Features' JS Output

Feature	Example	JavaScript Output
String	`"Hello"`	`"Hello"`
String Interpolation	`` `Hello ${message}` ``	`"Hello " + message`
Character (disrecommended)	`'x'`	`120` (char code)
Integer	`23`, `-23`	`23`, `-23`
Float	`23.0`, `-23.0`	`23.0`, `-23.0`

Feature	Example	JavaScript Output
Integer Addition	`23 + 1`	`23 + 1`
Float Addition	`23.0 +. 1.0`	`23.0 + 1.0`
Integer Division/Multiplication	`2 / 23 * 1`	`2 / 23 * 1`
Float Division/Multiplication	`2.0 /. 23.0 *. 1.0`	`2.0 / 23.0 * 1.0`
Float Exponentiation	`2.0 ** 3.0`	`Math.pow(2.0, 3.0)`
String Concatenation	`"Hello " ++ "World"`	`"Hello " + "World"`
Comparison	`>, <, >=, <=`	`>, <, >=, <=`
Boolean operation	`!, &&, ||`	`!, &&, ||`
Shallow and deep Equality	`===, ==`	`===, ==`
List (disrecommended)	`list{1, 2, 3}`	`{hd: 1, tl: {hd: 2, tl: {hd: 3, tl: 0}}}`
List Prepend	`list{a1, a2, ...oldList}`	`{hd: a1, tl: {hd: a2, tl: theRest}}`
Array	`[1, 2, 3]`	`[1, 2, 3]`
Record	`type t = {b: int}; let a = {b: 10}`	`var a = {b: 10}`
Multiline Comment	`/* Comment here */`	Not in output
Single line Comment	`// Comment here`	Not in output

Note that this is a cleaned-up comparison table; a few examples' JavaScript output are slightly different in reality.

SECTION embed-raw-javascript

---

title: "Embed Raw JavaScript" description: "Utility syntax to for raw JS usage in ReScript" canonical: "/docs/manual/latest/embed-raw-javascript"

Embed Raw JavaScript

Paste Raw JS Code

First thing first. If you're ever stuck learning ReScript, remember that you can always just paste raw JavaScript code into our source file:

``res example %%raw(// look ma, regular JavaScript! var message = "hello"; function greet(m) { console.log(m) } `)

```js
// look ma, regular JavaScript!
var message = "hello";
function greet(m) {
  console.log(m)
}
```

The `%%raw` special ReScript call takes your code string and pastes it as-is into the output. **You've now technically written your first ReScript file!**

(The back tick syntax is a multiline string, similar to JavaScript's. Except for us, no escaping is needed inside the string. More on string in a later section.)

While `%%raw` lets you embed top-level raw JS code, `%raw` lets you embed expression-level JS code:

``res example let add = %raw(function(a, b) { console.log("hello from raw JavaScript!"); return a + b } `)

Js.log(add(1, 2))

```js
var add = function(a, b) {
  console.log("hello from raw JavaScript!");
  return a + b
};

console.log(add(1, 2));
```

The above code: - declared a ReScript variable `add`, - with the raw JavaScript value of a function declaration, - then called that function in ReScript.

If your boss is ever worried that your teammates can't adopt ReScript, just let them keep writing JavaScript inside ReScript files =).

Debugger

You can also drop a `%debugger` expression in a body:

```res example let f = (x, y) => { %debugger x + y }
```

```js
function f(x, y) {
  debugger;
  return x + y | 0;
}
```

Output:

```
function f(x, y) {
  debugger; // JavaScript developer tools will set an breakpoint and stop here
  x + y;
}
```

Tips & Tricks

Embedding raw JS snippets isn't the best way to experience ReScript, though it's also highly useful if you're just starting out. As a matter of fact, the first few ReScript projects were converted through:

- pasting raw JS snippets inside a file
- examining the JS output (identical to the old hand-written JS)
- gradually extract a few values and functions and making sure the output still looks OK

At the end, we get a fully safe, converted ReScript file whose JS output is clean enough that we can confidently assert that no new bug has been introduced during the conversion process.

We have a small guide on this iteration here. Feel free to peruse it later.

SECTION promise

---

title: "Promises" description: "JS Promise handling in ReScript" canonical: "/docs/manual/latest/promise"

Promise

> **Note:** Starting from ReScript 10.1 and above, we recommend using async / await when interacting with Promises.

`promise` type

Since 10.1

In ReScript, every JS promise is represented with the globally available `promise<'a>` type. For ReScript versions < 10.1, use its original alias `Js.Promise.t<'a>` instead.

Here's a usage example in a function signature:

```
// User.resi file

type user = {name: string}

let fetchUser: string => promise<user>
```

To work with promise values (instead of using `async` / `await`) you may want to use the built in `Js.Promise2` module.

Js.Promise2

A builtin module to create, chain and manipulate promises.

> **Note:** This is an intermediate replacement for the `Js.Promise` module. It is designed to work with the `->` operator and should be used in favour of it's legacy counterpart. We are aware that the `Belt`, `Js` and `Js.xxx2` situation is confusing; a proper solution will hopefully be part of our upcoming `v11` release.

Creating a promise

```
let p1 = Js.Promise2.make((~resolve, ~reject) => {
  // We use uncurried functions for resolve / reject
  // for cleaner JS output without unintended curry calls
  resolve(. "hello world")
})

let p2 = Js.Promise2.resolve("some value")

// You can only reject `exn` values for streamlined catch handling
exception MyOwnError(string)
```

```
let p3 = Js.Promise2.reject(MyOwnError("some rejection"))
```

Access the contents and transform a promise

```
let logAsyncMessage = () => {
  open Js.Promise2
  Js.Promise2.resolve("hello world")
  ->then(msg => {
    // then callbacks require the result to be resolved explicitly
    resolve("Message: " ++ msg)
  })
  ->then(msg => {
    Js.log(msg)

    // Even if there is no result, we need to use resolve() to return a promise
    resolve()
  })
  ->ignore // Requires ignoring due to unhandled return value
}
```

For comparison, the `async` / `await` version of the same code would look like this:

```
let logAsyncMessage = async () => {
  let msg = await Js.Promise2.resolve("hello world")
  Js.log(`Message: ${msg}`)
}
```

Needless to say, the async / await version offers better ergonomics and less opportunities to run into type issues.

Run multiple promises in parallel

In case you want to launch multiple promises in parallel, use `Js.Promise2.all`:

```
@val
external fetchMessage: string => promise<string> = "global.fetchMessage"

let logAsyncMessage = async () => {
  let messages = await Js.Promise2.all([fetchMessage("message1"), fetchMessage("message2")])

  Js.log(Js.Array2.joinWith(messages, ", "))
}
```

```
async function logAsyncMessage(param) {
  var messages = await Promise.all([
        global.fetchMessage("message1"),
        global.fetchMessage("message2")
      ]);
  console.log(messages.join(", "));
}

export {
  logAsyncMessage ,
}
```

Js.Promise module (legacy - do not use)

> **Note:** The `Js.Promise` bindings are following the outdated data-last convention from a few years ago. We kept those APIs for backwards compatibility. Either use `Js.Promise2` or a third-party promise binding instead.

ReScript has built-in support for JavaScript promises. The 3 functions you generally need are:

- `Js.Promise.resolve: 'a => Js.Promise.t<'a>`
- `Js.Promise.then_: ('a => Js.Promise.t<'b>, Js.Promise.t<'a>) => Js.Promise.t<'b>`
- `Js.Promise.catch: (Js.Promise.error => Js.Promise.t<'a>, Js.Promise.t<'a>) => Js.Promise.t<'a>`

Additionally, here's the type signature for creating a promise on the ReScript side:

```
Js.Promise.make: (
  (
    ~resolve: (. 'a) => unit,
    ~reject: (. exn) => unit
  ) => unit
) => Js.Promise.t<'a>
```

This type signature means that `make` takes a callback that takes 2 named arguments, `resolve` and `reject`. Both arguments are themselves uncurried callbacks (with a dot). `make` returns the created promise.

Usage

Using the pipe operator:

```res example let myPromise = Js.Promise.make((~resolve, ~reject) => resolve(. 2))

myPromise->Js.Promise.then_(value => { Js.log(value) Js.Promise.resolve(value + 2) }, *)->Js.Promise.then*(value => { Js.log(value) Js.Promise.resolve(value + 3) }, *)->Js.Promise.catch(err => { Js.log2("Failure!!", err) Js.Promise.resolve(-2) },* )

```js
var myPromise = new Promise(function (resolve, reject) {
  return resolve(2);
});

myPromise
  .then(function (value) {
    console.log(value);
    return Promise.resolve((value + 2) | 0);
  })
  .then(function (value) {
    console.log(value);
    return Promise.resolve((value + 3) | 0);
  })
  .catch(function (err) {
    console.log("Failure!!", err);
    return Promise.resolve(-2);
  });
```

SECTION bind-to-js-function

---

title: "Bind to JS Function" description: "JS interop with functions in ReScript" canonical: "/docs/manual/latest/bind-to-js-function"

Function

Binding a JS function is like binding any other value:

```res example // Import nodejs' path.dirname @module("path") external dirname: string => string = "dirname" let root = dirname("/User/github") // returns "User"

```js
var Path = require("path");
var root = Path.dirname("/User/github");
```

We also expose a few special features, described below.

Labeled Arguments

ReScript has labeled arguments (that can also be optional). These work on an `external` too! You'd use them to *fix* a JS function's unclear usage. Assuming we're modeling this:

```
// MyGame.js

function draw(x, y, border) {
   // suppose `border` is optional and defaults to false
}
draw(10, 20)
draw(20, 20, true)
```

It'd be nice if on ReScript's side, we can bind & call `draw` while labeling things a bit:

```res example @module("MyGame") external draw: (~x: int, ~y: int, ~border: bool=?) => unit = "draw"

draw(~x=10, ~y=20, ~border=true) draw(~x=10, ~y=20)

```js
var MyGame = require("MyGame");

MyGame.draw(10, 20, true);
MyGame.draw(10, 20, undefined);
```

We've compiled to the same function, but now the usage is much clearer on the ReScript side thanks to labels!

Note that you can freely reorder the labels on the ReScript side; they'll always correctly appear in their declaration order in the JavaScript output:

```res example @module("MyGame") external draw: (~x: int, ~y: int, ~border: bool=?) => unit = "draw"

draw(~x=10, ~y=20) draw(~y=20, ~x=10)

```js
```

```
var MyGame = require("MyGame");

MyGame.draw(10, 20, undefined);
MyGame.draw(10, 20, undefined);
```

Object Method

Functions attached to a JS objects (other than JS modules) require a special way of binding to them, using `send`:

```res example type document // abstract type for a document object @send external getElementById: (document, string) => Dom.element = "getElementById" @val external doc: document = "document"

let el = getElementById(doc, "myId")
```

```js
var el = document.getElementById("myId");
```

In a `send`, the object is always the first argument. Actual arguments of the method follow (this is a bit what modern OOP objects are really).

Chaining

Ever used `foo().bar().baz()` chaining ("fluent api") in JS OOP? We can model that in ReScript too, through the pipe operator.

Variadic Function Arguments

You might have JS functions that take an arbitrary amount of arguments. ReScript supports modeling those, under the condition that the arbitrary arguments part is homogenous (aka of the same type). If so, add `variadic` to your `external`.

```res example @module("path") @variadic external join: array => string = "join"

let v = join(["a", "b"])
```

```js
var Path = require("path");
var v = Path.join("a", "b");
```

`module` will be explained in Import from/Export to JS.

Modeling Polymorphic Function

Apart from the above special-case, JS function in general are often arbitrary overloaded in terms of argument types and number. How would you bind to those?

Trick 1: Multiple `externals`

If you can exhaustively enumerate the many forms an overloaded JS function can take, simply bind to each differently:

```res example @module("MyGame") external drawCat: unit => unit = "draw" @module("MyGame") external drawDog: (~giveName: string) => unit = "draw" @module("MyGame") external draw: (string, ~useRandomAnimal: bool) => unit = "draw"
```

```js
// Empty output
```

Note how all three externals bind to the same JS function, `draw`.

Trick 2: Polymorphic Variant + `unwrap`

If you have the irresistible urge of saying "if only this JS function argument was a variant instead of informally being either `string` or `int`", then good news: we do provide such `external` features through annotating a parameter as a polymorphic variant! Assuming you have the following JS function you'd like to bind to:

```
function padLeft(value, padding) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}
```

Here, `padding` is really conceptually a variant. Let's model it as such.

```res example @val external padLeft: ( string, @unwrap [ | #Str(string) | #Int(int) ]) => string = "padLeft" padLeft("Hello World",
```

#Int(4)) padLeft("Hello World", #Str("Message from ReScript: "))

```js
padLeft("Hello World", 4);
padLeft("Hello World", "Message from ReScript: ");
```

Obviously, the JS side couldn't have an argument that's a polymorphic variant! But here, we're just piggy backing on poly variants' type checking and syntax. The secret is the `@unwrap` annotation on the type. It strips the variant constructors and compile to just the payload's value. See the output.

Constrain Arguments Better

Consider the Node `fs.readFileSync`'s second argument. It can take a string, but really only a defined set: `"ascii"`, `"utf8"`, etc. You can still bind it as a string, but we can use poly variants + `string` to ensure that our usage's more correct:

```res example @module("fs") external readFileSync: ( ~name: string, @string [ | #utf8 | @as("ascii") #useAscii ], ) => string = "readFileSync"

readFileSync(~name="xx.txt", #useAscii)
```

```js
var Fs = require("fs");
Fs.readFileSync("xx.txt", "ascii");
```

- Attaching `@string` to the whole poly variant type makes its constructor compile to a string of the same name.
- Attaching a `@as("bla")` to a constructor lets you customize the final string.

And now, passing something like `"myOwnUnicode"` or other variant constructor names to `readFileSync` would correctly error.

Aside from string, you can also compile an argument to an int, using `int` instead of `string` in a similar way:

```res example @val external testIntType: ( @int [ | #onClosed | @as(20) #onOpen | #inBinary ]) => int = "testIntType" testIntType(#inBinary)
```

```js
testIntType(21);
```

`onClosed` compiles to `0`, `onOpen` to `20` and `inBinary` to **`21`**.

Unknown for type safety

It is best practice to inspect data received from untrusted external functions to ensure it contains what you expect. This helps avoid run-time crashes and unexpected behavior. If you're certain about what an external function returns, simply assert the return value as `string` or `array<int>` or whatever you want it to be. Otherwise use `unknown`. The ReScript type system will prevent you from using an `unknown` until you first inspect it and "convert" it using JSON parsing utilities or similar tools.

Consider the example below of two external functions that access the value of a property on a JavaScript object. `getPropertyUnsafe` returns an `'a`, which means "anything you want it to be." ReScript allows you to use this value as a `string` or `array` or any other type. Quite convenient! But if the property is missing or contains something unexpected, your code might break. You can make the binding more safe by changing `'a` to `string` or `option<'a>`, but this doesn't completely eliminate the problem.

The `getPropertySafe` function returns an `unknown`, which could be `null` or a `string` or anything else. But ReScript prevents you from using this value inappropriately until it has been safely parsed.

```res example @get_index external getPropertyUnsafe: ({..}, string) => 'a = "" @get_index external getPropertySafe: ({..}, string) => unknown = ""

let person = {"name": "Bob", "age": 12}

let greeting1 = "Hello, " ++ getPropertyUnsafe(person, "name") // works (this time!) // let greeting2 = "Hello, " ++ getPropertySafe(person, "name") // syntax error

#### Special-case: Event Listeners

One last trick with polymorphic variants:

<CodeTab labels={["ReScript", "JS Output"]}>

```res example
type readline

@send
external on: (
    readline,
    @string [
      | #close(unit => unit)
      | #line(string => unit)
```
```

```
    ]
  )
  => readline = "on"

let register = rl =>
  rl
  ->on(#close(event => ()))
  ->on(#line(line => Js.log(line)));
```

```
function register(rl) {
  return rl
    .on("close", function($$event) {})
    .on("line", function(line) {
      console.log(line);
    });
}
```

Fixed Arguments

Sometimes it's convenient to bind to a function using an `external`, while passing predetermined argument values to the JS function:

```res example @val external processOnExit: ( @as("exit") _, int => unit ) => unit = "process.on"

processOnExit(exitCode => Js.log("error code: " ++ Js.Int.toString(exitCode)) );
```

```js
process.on("exit", function (exitCode) {
  console.log("error code: " + exitCode.toString());
});
```

The `@as("exit")` and the placeholder _ argument together indicates that you want the first argument to compile to the string `"exit"`. You can also use any JSON literal with `as`: `@as(json`true`)`, `@as(json`{"name": "John"}`)`, etc.

Ignore arguments

You can also explicitly "hide" `external` function parameters in the JS output, which may be useful if you want to add type constraints to other parameters without impacting the JS side:

```
@val external doSomething: (@ignore 'a, 'a) => unit = "doSomething"

doSomething("this only shows up in ReScript code", "test")
```

```
doSomething("test");
```

Note: It's a pretty niche feature, mostly used to map to polymorphic JS APIs.

Modeling `this`-based Callbacks

Many JS libraries have callbacks which rely on this (the source), for example:

```
x.onload = function(v) {
  console.log(this.response + v)
}
```

Here, `this` would point to `x` (actually, it depends on how `onload` is called, but we digress). It's not correct to declare `x.onload` of type `(. unit) -> unit`. Instead, we introduced a special attribute, `this`, which allows us to type `x` as so:

```res example type x @val external x: x = "x" @set external setOnload: (x, @this ((x, int) => unit)) => unit = "onload" @get external resp: x => int = "response" setOnload(x, @this (o, v) => Js.log(resp(o) + v))
```

```js
x.onload = function (v) {
  var o = this;
  console.log((o.response + v) | 0);
};
```

`this` has its first parameter is reserved for `this` and for arity of 0, there is no need for a redundant `unit` type.

Function Nullable Return Value Wrapping

For JS functions that return a value that can also be `undefined` or `null`, we provide `@return(...)`. To automatically convert that value to an `option` type (recall that ReScript `option` type's `None` value only compiles to `undefined` and not `null`).

```res example type element type dom

@send @return(nullable) external getElementById: (dom, string) => option = "getElementById"

let test = dom => { let elem = dom->(getElementById("haha")) switch (elem) { | None => 1 | Some(_ui) => 2 } }
```

```js
function test(dom) {
  var elem = dom.getElementById("haha");
  if (elem == null) {
    return 1;
  } else {
    console.log(elem);
    return 2;
  }
}
```

`return(nullable)` attribute will automatically convert `null` and `undefined` to `option` type.

Currently 4 directives are supported: `null_to_opt`, `undefined_to_opt`, `nullable` and `identity`.

`identity` will make sure that compiler will do nothing about the returned value. It is rarely used, but introduced here for debugging purpose.

SECTION json

---

title: "JSON" description: "Interacting with JSON in ReScript" canonical: "/docs/manual/latest/json"

JSON

Parse

Bind to JavaScript's `JSON.parse` and type the return value as the type you're expecting:

```res example // declare the shape of the json you're binding to type data = {names: array}

// bind to JS' JSON.parse @scope("JSON") @val external parseIntoMyData: string => data = "parse"

let result = parseIntoMyData(`{"names": ["Luke", "Christine"]}`) let name1 = result.names[0]
```

```js
var result = JSON.parse("{\"names\": [\"Luke\", \"Christine\"]}");
var name1 = result.names[0];
```

Where `data` can be any type you assume the JSON is. As you can see, this compiles to a straightforward `JSON.parse` call. As with regular JS, this is convenient, but has no guarantee that e.g. the data is correctly shaped, or even syntactically valid. Slightly dangerous.

Stringify

Use `Js.Json.stringify`:

```res example Js.log(Js.Json.stringifyAny(["Amy", "Joe"]))
```

```js
console.log(JSON.stringify([
  "Amy",
  "Joe"
]));
```

Advanced

The Js.Json module provides slightly safer, low-level building blocks for power users who want to parse JSON on a per-field basis. See the examples in the API docs.

SECTION type

---

title: "Type" description: "Types and type definitions in ReScript" canonical: "/docs/manual/latest/type"

Type

Types are the highlight of ReScript! They are: - **Strong**. A type can't change into another type. In JavaScript, your variable's type might change when the code runs (aka at runtime). E.g. a `number` variable might change into a `string` sometimes. This is an anti-feature; it makes the code much harder to understand when reading or debugging. - **Static**. ReScript types are erased after compilation and don't exist at runtime. Never worry about your types dragging down performance. You don't need type info during runtime; we report all the information (especially all the type errors) during compile time. Catch the bugs earlier! - **Sound**. This is our biggest differentiator versus many other typed languages that compile to JavaScript. Our type system is guaranteed to **never** be wrong. Most type systems make a

guess at the type of a value and show you a type in your editor that's sometime incorrect. We don't do that. We believe that a type system that is sometime incorrect can end up being dangerous due to expectation mismatches. - **Fast**. Many developers underestimate how much of their project's build time goes into type checking. Our type checker is one of the fastest around. - **Inferred**. You don't have to write down the types! ReScript can deduce them from their values. Yes, it might seem magical that we can deduce all of your program's types, without incorrectness, without your manual annotation, and do so quickly. Welcome to ReScript =).

The following sections explore more of our type system.

Inference

This let-binding doesn't contain any written type:

```res example let score = 10 let add = (a, b) => a + b

```js
var score = 10;
function add(a, b) {
  return a + b | 0;
}
```

ReScript knows that `score` is an `int`, judging by the value `10`. This is called **inference**. Likewise, it also knows that the `add` function takes 2 `int`s and returns an `int`, judging from the `+` operator, which works on ints.

Type Annotation

But you can also optionally write down the type, aka annotate your value:

```res example let score: int = 10

```js
var score = 10;
```

If the type annotation for `score` doesn't correspond to our inferred type for it, we'll show you an error during compilation time. We **won't** silently assume your type annotation is correct, unlike many other languages.

You can also wrap any expression in parentheses and annotate it:

```
let myInt = 5
let myInt: int = 5
let myInt = (5: int) + (4: int)
let add = (x: int, y: int) : int => x + y
let drawCircle = (~radius as r: int): circleType => /* code here */
```

```
var myInt = 9;
function add(x, y) {
  return x + y | 0;
}
function drawCircle(r) {
  /* code here */
}
```

Note: in the last line, `(~radius as r: int)` is a labeled argument. More on this in the function page.

Type Alias

You can refer to a type by a different name. They'll be equivalent:

```res example type scoreType = int let x: scoreType = 10

```js
var x = 10;
```

Type Parameter (Aka Generic)

Types can accept parameters, akin to generics in other languages. The parameters' names **need** to start with `'`.

The use-case of a parameterized type is to kill duplications. Before:

```res example // this is a tuple of 3 items, explained next type intCoordinates = (int, int, int) type floatCoordinates = (float, float, float)

let a: intCoordinates = (10, 20, 20) let b: floatCoordinates = (10.5, 20.5, 20.5)

```js
var a = [10, 20, 20];
var b = [10.5, 20.5, 20.5];
```

After:

```res example type coordinates<'a> = ('a, 'a, 'a)

let a: coordinates = (10, 20, 20) let b: coordinates = (10.5, 20.5, 20.5)

```js
var a = [10, 20, 20];
var b = [10.5, 20.5, 20.5];
```

Note that the above codes are just contrived examples for illustration purposes. Since the types are inferred, you could have just written:

```res example let buddy = (10, 20, 20)

```js
var buddy = [10, 20, 20];
```

The type system infers that it's a `(int, int, int)`. Nothing else needed to be written down.

Type arguments appear in many places. Our `array<'a>` type is such a type that requires a type parameter.

``res example // inferred asarray` let greetings = ["hello", "world", "how are you"]

```js
// inferred as `array<string>`
var greetings = ["hello", "world", "how are you"];
```

If types didn't accept parameters, the standard library would need to define the types `arrayOfString`, `arrayOfInt`, `arrayOfTuplesOfInt`, etc. That'd be tedious.

Types can receive many arguments, and be composable.

```res example type result<'a, 'b> = | Ok('a) | Error('b)

type myPayload = {data: string}

type myPayloadResults<'errorType> = array>

let payloadResults: myPayloadResults = [ Ok({data: "hi"}), Ok({data: "bye"}), Error("Something wrong happened!") ]

```js
var payloadResults = [
  {
    TAG: /* Ok */0,
    _0: {data: "hi"}
  },
  {
    TAG: /* Ok */0,
    _0: {data: "bye"}
  },
  {
    TAG: /* Error */1,
    _0: "Something wrong happened!"
  }
];
```

Recursive Types

Just like a function, a type can reference itself within itself using `rec`:

```res example type rec person = { name: string, friends: array }

```js
// Empty output
```

Mutually Recursive Types

Types can also be *mutually* recursive through `and`:

```res example type rec student = {taughtBy: teacher} and teacher = {students: array}

```js
// Empty output
```

Type Escape Hatch

ReScript's type system is robust and does not allow dangerous, unsafe stuff like implicit type casting, randomly guessing a value's type, etc. However, out of pragmatism, we expose a single escape hatch for you to "lie" to the type system:

```
external myShadyConversion: myType1 => myType2 = "%identity"
```
```

```
// Empty output
```

This declaration converts a `myType1` of your choice to `myType2` of your choice. You can use it like so:

```res example external convertToFloat : int => float = "%identity" let age = 10 let gpa = 2.1 +. convertToFloat(age)

```js
var age = 10;
var gpa = 2.1 + 10;
```

Obviously, do **not** abuse this feature. Use it tastefully when you're working with existing, overly dynamic JS code, for example.

More on externals here.

Note: this particular `external` is the only one that isn't preceded by a `@` attribute.

SECTION newcomer-examples

---

title: "Newcomer Examples" description: "Quick examples for users new to ReScript" canonical: "/docs/manual/latest/newcomer-examples"

Newcomer Examples

An example is worth a thousand words.

This section is dedicated to newcomers trying to figure out general idioms & conventions. If you're a beginner who's got a good idea for an example, please suggest an edit!

Use the `option` type

```res example let possiblyNullValue1 = None let possiblyNullValue2 = Some("Hello")

switch possiblyNullValue2 { | None => Js.log("Nothing to see here.") | Some(message) => Js.log(message) }

```js
var possiblyNullValue1;
var possiblyNullValue2 = "Hello";

if (possiblyNullValue2 !== undefined) {
  console.log(possiblyNullValue2);
} else {
  console.log("Nothing to see here.");
}
```

Create a Parametrized Type

```res example type universityStudent = {gpa: float}

type response<'studentType> = { status: int, student: 'studentType, }

```js
// Empty output
```

Creating a JS Object

```res example let student1 = { "name": "John", "age": 30, }

```js
var student1 = {
  name: "John",
  age: 30,
};
```

Or using record:

```res example type payload = { name: string, age: int, }

let student1 = { name: "John", age: 30, }

```js
var student1 = {
  name: "John",
  age: 30,
};
```
```

Modeling a JS Module with Default Export

See here.

Checking for JS nullable types using the `option` type

For a function whose argument is passed a JavaScript value that's potentially `null` or `undefined`, it's idiomatic to convert it to an `option`. The conversion is done through the helper functions in ReScript's `Js.Nullable` module. In this case, `toOption`:

```res example let greetByName = (possiblyNullName) => { let optionName = Js.Nullable.toOption(possiblyNullName) switch optionName { | None => "Hi" | Some(name) => "Hello " ++ name } }
```

```js
function greetByName(possiblyNullName) {
  if (possiblyNullName == null) {
    return "Hi";
  } else {
    return "Hello " + possiblyNullName;
  }
}
```

This check compiles to `possiblyNullName == null` in JS, so checks for the presence of `null` or `undefined`.

SECTION browser-support-polyfills

---

title: "Browser Support & Polyfills" description: "Note on browser support in ReScript" canonical: "/docs/manual/latest/browser-support-polyfills"

Browser Support & Polyfills

ReScript compiles to JavaScript **ES5**, with the exception of optionally allowing to compile to ES6's module import & export.

For old browsers, you also need to polyfill TypedArray. The following standard library functions require it:

- `Int64.float_of_bits`
- `Int64.bits_of_float`
- `Int32.float_of_bits`
- `Int32.bits_of_float`

If you don't use these functions, you're fine. Otherwise, it'll be a runtime failure.

SECTION primitive-types

---

title: "Primitive Types" description: "Primitive Data Types in ReScript" canonical: "/docs/manual/latest/primitive-types"

Primitive Types

ReScript comes with the familiar primitive types like `string`, `int`, `float`, etc.

String

ReScript `string`s are delimited using **double** quotes (single quotes are reserved for the character type below).

```res example let greeting = "Hello world!" let multilineGreeting = "Hello world!"
```

```js
var greeting = "Hello world!";
var multilineGreeting = "Hello\n world!";
```

To concatenate strings, use `++`:

```res example let greetings = "Hello " ++ "world!"
```

```js
var greetings = "Hello world!";
```

String Interpolation

There's a special syntax for string that allows

- multiline string just like before
- no special character escaping
- Interpolation

```res example let name = "Joe"

let greeting = `Hello World ðŸ‘‹ ${name}`

```js
var name = "Joe";

var greeting = "Hello\nWorld\nðŸ‘‹\n" + name + "\n";
```

This is just like JavaScript's backtick string interpolation, except without needing to escape special characters.

Usage

See the familiar `String` API in the API docs. Since a ReScript string maps to a JavaScript string, you can mix & match the string operations in all standard libraries.

Tips & Tricks

You have a good type system now! In an untyped language, you'd often overload the meaning of string by using it as:

- a unique id: `var BLUE_COLOR = "blue"`
- an identifier into a data structure: `var BLUE = "blue" var RED = "red" var colors = [BLUE, RED]`
- the name of an object field: `person["age"] = 24`
- an enum: `if (audio.canPlayType() === 'probably') {...}` (à² _à²)
- other crazy patterns you'll soon find horrible, after getting used to ReScript's alternatives.

The more you overload the poor string type, the less the type system (or a teammate) can help you! ReScript provides concise, fast and maintainable types & data structures alternatives to the use-cases above (e.g. variants, in a later section).

Char

ReScript has a type for a string with a single letter:

```res example let firstLetterOfAlphabet = 'a'

```js
var firstLetterOfAlphabet = /* "a" */97;
```

Note: Char doesn't support Unicode or UTF-8 and is therefore not recommended.

To convert a String to a Char, use `String.get("a", 0)`. To convert a Char to a String, use `String.make(1, 'a')`.

Regular Expression

ReScript regular expressions compile cleanly to their JavaScript counterpart:

```res example let r = %re("/b/g")

```js
var r = /b/g;
```

A regular expression like the above has the type `Js.Re.t`. The Js.Re module contains the regular expression helpers you have seen in JS.

Boolean

A ReScript boolean has the type `bool` and can be either `true` or `false`. Common operations:

- `&&`: logical and.
- `||`: logical or.
- `!`: logical not.
- `<=`, `>=`, `<`, `>`
- `==`: structural equal, compares data structures deeply. `(1, 2) == (1, 2)` is `true`. Convenient, but use with caution.
- `===`: referential equal, compares shallowly. `(1, 2) === (1, 2)` is `false`. `let myTuple = (1, 2); myTuple === myTuple` is `true`.
- `!=`: structural unequal.
- `!==`: referential unequal.

ReScript's `true/false` compiles into a JavaScript `true/false`.

Integers
```

32-bits, truncated when necessary. We provide the usual operations on them: `+`, `-`, `*`, `/`, etc. See Js.Int for helper functions.

Be careful when you bind to JavaScript numbers! Since ReScript integers have a much smaller range than JavaScript numbers, data might get lost when dealing with large numbers. In those cases itâ€™s much safer to bind the numbers as **float**. Be extra mindful of this when binding to JavaScript Dates and their epoch time.

To improve readability, you may place underscores in the middle of numeric literals such as `1_000_000`. Note that underscores can be placed anywhere within a number, not just every three digits.

Floats

Float requires other operators: `+.`, `-.`, `*.`, `/.`, etc. Like `0.5 +. 0.6`. See Js.Float for helper functions.

As with integers, you may use underscores within literals to improve readability.

Unit

The `unit` type indicates the absence of a specific value. It has only a single value, `()`, which acts as a placeholder when no other value exists or is needed. It compiles to JavaScript's `undefined` and resembles the `void` type in languages such as C++. What's the point of such a type?

Consider the `Math.random` function. Its type signature is `unit => float`, which means it receives a `unit` as input and calculates a random `float` as output. You use the function like this - `let x = Math.random()`. Notice `()` as the first and only function argument.

Imagine a simplified `Console.log` function that prints a message. Its type signature is `string => unit` and you'd use it like this `Console.log("Hello!")`. It takes a string as input, prints it, and then returns nothing useful. When `unit` is the output of a function it means the function performs some kind of side-effect.

Unknown

The `unknown` type represents values with contents that are a mystery or are not 100% guaranteed to be what you think they are. It provides type-safety when interacting with data received from an untrusted source. For example, suppose an external function is supposed to return a `string`. It might. But if the documentation is not accurate or the code has bugs, the function could return `null`, an `array`, or something else you weren't expecting.

The ReScript type system helps you avoid run-time crashes and unpredicatable behavior by preventing you from using `unknown` in places that expect a `string` or `int` or some other type. The ReScript core libraries also provide utility functions to help you inspect `unknown` values and access their contents. In some cases you may need a JSON parsing library to convert `unknown` values to types you can safely use.

Consider using `unknown` when receiving data from external JavaScript functions

SECTION import-from-export-to-js

---

title: "Import from / Export to JS" description: "Importing / exporting JS module content in ReScript" canonical: "/docs/manual/latest/import-from-export-to-js"

Import from/Export to JS

You've seen how ReScript's idiomatic Import & Export works. This section describes how we work with importing stuff from JavaScript and exporting stuff for JavaScript consumption.

Note: due to JS ecosystem's module compatibility issues, our advice of keeping your ReScript file's compiled JS output open in a tab applies here **more than ever**, as you don't want to subtly output the wrong JS module import/export code, on top of having to deal with Babel/Webpack/Jest/Node's CommonJS \<-> ES6 compatibility shims.

In short: **make sure your bindings below output what you'd have manually written in JS**.

Output Format

We support 2 JavaScript import/export formats:

- CommonJS: `require('myFile')` and `module.exports = ...`.
- ES6 modules: `import * from 'MyReScriptFile'` and `export let ...`.

The format is configurable in via `rescript.json`.

Import From JavaScript

Import a JavaScript Module's Named Export

Use the `module` external:

```res example // Import nodejs' path.dirname @module("path") external dirname: string => string = "dirname" let root = dirname("/User/github") // returns "User"

```js
var Path = require("path");
var root = Path.dirname("/User/github");

import * as Path from "path";
var root = Path.dirname("/User/github");
```

Here's what the `external` does:

- `@module("path")`: pass the name of the JS module; in this case, `"path"`. The string can be anything: `"./src/myJsFile"`, `"@myNpmNamespace/myLib"`, etc.
- `external`: the general keyword for declaring a value that exists on the JS side.
- `dirname`: the binding name you'll use on the ReScript side.
- `string => string`: the type signature of `dirname`. Mandatory for `external`s.
- `= "dirname"`: the name of the variable inside the `path` JS module. There's repetition in writing the first and second `dirname`, because sometime the binding name you want to use on the ReScript side is different than the variable name the JS module exported.

Import a JavaScript Module As a Single Value

By omitting the string argument to `module`, you bind to the whole JS module:

```res example @module external leftPad: (string, int) => string = "./leftPad" let paddedResult = leftPad("hi", 5)

```js
var LeftPad = require("./leftPad");
var paddedResult = LeftPad("hi", 5);

import * as LeftPad from "./leftPad";
var paddedResult = LeftPad("hi", 5);
```

Depending on whether you're compiling ReScript to CommonJS or ES6 module, **this feature will generate subtly different code**. Please check both output tabs to see the difference. The ES6 output here would be wrong!

Import an ES6 Default Export

Use the value `"default"` on the right hand side:

```res example @module("./student") external studentName: string = "default" Js.log(studentName)

```js
import Student from "./student";
var studentName = Student;
```

Dynamic Import

Leveraging JavaScript's dynamic `import` to reduce bundle size and lazy load code as needed is easy in ReScript. It's also a little bit more convenient than in regular JavaScript because you don't need to keep track of file paths manually with ReScript's module system.

Dynamically Importing Parts of a Module

Use the `Js.import` function to dynamically import a specific part of a module. Put whatever `let` binding you want to import in there, and you'll get a `promise` back resolving to that specific binding.

Let's look at an example. Imagine the following file `MathUtils.res`:

```
let add = (a, b) => a + b
let sub = (a, b) => a - b
```

Now let's dynamically import the add function in another module, e.g. `App.res`:

```
// App.res
let main = async () => {
  let add = await Js.import(MathUtils.add)
  let onePlusOne = add(1, 1)

  Console.log(onePlusOne)
}

async function main() {
```
```

```
  var add = await import("./MathUtils.mjs").then(function(m) {
    return m.add;
  });

  var onePlusOne = add(1, 1);
  console.log(onePlusOne);
}
```

Dynamically Importing an Entire Module

The syntax for importing a whole module looks a little different, since we are operating on the module syntax level; instead of using `Js.import`, you may simply `await` the module itself:

```
// App.res
let main = async () => {
  module Utils = await MathUtils

  let twoPlusTwo = Utils.add(2, 2)
  Console.log(twoPlusTwo)
}
```

```
async function main() {
  var Utils = await import("./MathUtils.mjs");

  var twoPlusTwo = Utils.add(2, 2);
  console.log(twoPlusTwo);
}
```

Export To JavaScript

Export a Named Value

As mentioned in ReScript's idiomatic Import & Export, every let binding and module is exported by default to other ReScript modules (unless you use a `.resi` interface file). If you open up the compiled JS file, you'll see that these values can also directly be used by a *JavaScript* file too.

Export an ES6 Default Value

If your JS project uses ES6 modules, you're likely exporting & importing some default values:

```
// student.js
export default name = "Al";
```

```
// teacher.js
import studentName from 'student.js';
```

A JavaScript default export is really just syntax sugar for a named export implicitly called `default` (now you know!). So to export a default value from ReScript, you can just do:

```res example
// ReScriptStudent.res
let default = "Bob"
```

```js
var $$default = "Bob";

exports.$$default = $$default;
exports.default = $$default;
// informal transpiler-compatible marker of a default export compiled from ES6
exports.__esModule = true;
```

```
var $$default = "Bob";

export {
  $$default,
  $$default as default,
}
```

You can then import this default export as usual on the JS side:

```
// teacher2.js
import studentName from 'ReScriptStudent.js';
```

If your JavaScript's ES6 default import is transpiled by Babel/Webpack/Jest into CommonJS `require`s, we've taken care of that too! See the CommonJS output tab for `__esModule`.

SECTION mutation

---

title: "Mutation" description: "Imperative and mutative programming capabilities in ReScript" canonical: "/docs/manual/latest/mutation"

Mutation

ReScript has great traditional imperative & mutative programming capabilities. You should use these features sparingly, but sometimes they allow your code to be more performant and written in a more familiar pattern.

Mutate Let-binding

Let-bindings are immutable, but you can wrap it with a `ref`, exposed as a record with a single mutable field in the standard library:

```res prelude let myValue = ref(5)

```js
var myValue = {
  contents: 5
};
```

Usage

You can get the actual value of a `ref` box through accessing its `contents` field:

```res example let five = myValue.contents // 5

```js
var five = myValue.contents;
```

Assign a new value to `myValue` like so:

```res example myValue.contents = 6

```js
myValue.contents = 6;
```

We provide a syntax sugar for this:

```res example myValue := 6

```js
myValue.contents = 6;
```

Note that the previous binding `five` stays `5`, since it got the underlying item on the `ref` box, not the `ref` itself.

Note: you might see in the JS output tabs above that `ref` allocates an object. Worry not; local, non-exported `ref`s allocations are optimized away.

Tip & Tricks

Before reaching for `ref`, know that you can achieve lightweight, local "mutations" through overriding let bindings.

SECTION external

---

title: "External (Bind to Any JS Library)" description: "The external keyword" canonical: "/docs/manual/latest/external"

External (Bind to Any JS Library)

`external` is the primary ReScript features for bringing in and using JavaScript values.

`external` is like a let binding, but: - The right side of `=` isn't a value; it's the name of the JS value you're referring to. - The type for the binding is mandatory, since we need to know what the type of that JS value is. - Can only exist at the top level of a file or module.

```res example @val external setTimeout: (unit => unit, int) => float = "setTimeout"

```js
// Empty output
```

There are several kinds of `external`s, differentiated and/or augmented through the attribute they carry. This page deals with the general, shared mechanism behind most `external`s. The different are documented in their respective pages later. A few notable ones:

- `@val`, `@scope`: bind to global JS values.
- `@module`: bind to JS imported/exported values.
- `@send`: bind to JS methods.

You can also use our Syntax Lookup tool to find them.
```

Related: see also our list of external decorators.

Usage

Once declared, you can use an `external` as a normal value, just like a let binding.

Tips & Tricks

`external` + ReScript objects are a wonderful combination for quick prototyping. Check the JS output tab:

```res example // The type of document is just some random type 'a // that we won't bother to specify @val external document: 'a = "document"

// call a method document"addEventListener" })

// get a property let loc = document["location"]

// set a property document["location"]["href"] = "rescript-lang.org"
```

```js
document.addEventListener("mouseup", function(_event) {
  console.log("clicked!");
});

var loc = document.location;

document.location.href = "rescript-lang.org";
```

We've specified `document`'s type as `'a`, aka a placeholder type that's polymorphic. Any value can be passed there, so you're not getting much type safety (except the inferences at various call sites). However, this is excellent for quickly getting started using a JavaScript library in ReScript **without needing the equivalent of a repository of typed bindings** like TypeScript's `DefinitelyTyped` repo.

However, if you want to more rigidly bind to the JavaScript library you want, keep reading the next few interop pages.

Performance & Output Readability

`external`s declarations are inlined into their callers during compilation, **and completely disappear from the JS output**. This means any time you use one, you can be sure that you're not incurring extra JavaScript \<-> ReScript conversion cost.

Additionally, no extra ReScript-specific runtime is better for output readability.

> **Note:** do also use `external`s and the `@blabla` attributes in the interface files. Otherwise the inlining won't happen.

Design Decisions

ReScript takes interoperating with existing code very seriously. Our type system has very strong guarantees. However, such strong feature also means that, without a great interop system, it'd be very hard to gradually convert a codebase over to ReScript. Fortunately, our interop are comprehensive and cooperate very well with most existing JavaScript code.

The combination of a sound type system + great interop means that we get the benefits of a traditional gradual type system regarding incremental codebase coverage & conversion, without the downside of such gradual type system: complex features to support existing patterns, slow analysis, diminishing return in terms of type coverage, etc.

SECTION let-binding

---

title: "Let Binding" description: "Let binding syntax for binding to values in ReScript" canonical: "/docs/manual/latest/let-binding"

Let Binding

A "let binding", in other languages, might be called a "variable declaration". `let` *binds* values to names. They can be seen and referenced by code that comes *after* them.

```res example let greeting = "hello!" let score = 10 let newScore = 10 + score
```

```js
var greeting = "hello!";
var score = 10;
var newScore = 20;
```

Block Scope

Bindings can be scoped through `{}`.

```res example let message = { let part1 = "hello" let part2 = "world" part1 ++ " " ++ part2 } //part1andpart2` not accessible here!

```js
var message = "hello world";
```

The value of the last line of a scope is implicitly returned.

Design Decisions

ReScript's `if`, `while` and functions all use the same block scoping mechanism. The code below works **not** because of some special "if scope"; but simply because it's the same scope syntax and feature you just saw:

```
if displayGreeting {
  let message = "Enjoying the docs so far?"
  Js.log(message)
}
// `message` not accessible here!
```

```
if (displayGreeting) {
  console.log("Enjoying the docs so far?");
}
```

Bindings Are Immutable

Let bindings are "immutable", aka "cannot change". This helps our type system deduce and optimize much more than other languages (and in turn, help you more).

Binding Shadowing

The above restriction might sound unpractical at first. How would you change a value then? Usually, 2 ways:

The first is to realize that many times, what you want isn't to mutate a variable's value. For example, this JavaScript pattern:

```
var result = 0;
result = calculate(result);
result = calculateSomeMore(result);
```

...is really just to comment on intermediate steps. You didn't need to mutate `result` at all! You could have just written this JS:

```
var result1 = 0;
var result2 = calculate(result1);
var result3 = calculateSomeMore(result2);
```

In ReScript, this obviously works too:

```
let result1 = 0
let result2 = calculate(result1)
let result3 = calculateSomeMore(result2)
```

```
var result1 = 0;
var result2 = calculate(0);
var result3 = calculateSomeMore(result2);
```

Additionally, reusing the same let binding name overshadows the previous bindings with the same name. So you can write this too:

```
let result = 0
let result = calculate(result)
let result = calculateSomeMore(result)
```

```
var result = calculate(0);
var result$1 = calculateSomeMore(result);
```

(Though for the sake of clarity, we don't recommend this).

As a matter of fact, even this is valid code:

```res example let result = "hello" Console.log(result) // prints "hello" let result = 1 Console.log(result) // prints 1

```js
var result = 1;
console.log("hello");
console.log(1);
```

The binding you refer to is whatever's the closest upward. No mutation here! If you need *real* mutation, e.g. passing a value around, have it modified by many pieces of code, we provide a slightly heavier mutation feature.

Private let bindings

Private let bindings are introduced in the release 7.2.

In the module system, everything is public by default, the only way to hide some values is by providing a separate signature to list public fields and their types:

```
module A: {
  let b: int
} = {
  let a = 3
  let b = 4
}
```

`%%private` gives you an option to mark private fields directly

```
module A = {
  %%private(let a = 3)
  let b = 4
}
```

`%%private` also applies to file level modules, so in some cases, users do not need to provide a separate interface file just to hide some particular values.

Note interface files are still recommended as a general best practice since they give you better separate compilation units and also they're better for documentation.

Still, `%%private` is useful in the following scenarios:

- **Code generators.** Some code generators want to hide some values but it is sometimes very hard or time consuming for code generators to synthesize the types for public fields.
- **Quick prototyping.** During prototyping, we still want to hide some values, but the interface file is not stable yet, `%%private` provide you such convenience.

SECTION record

---

title: "Record" description: "Record types in ReScript" canonical: "/docs/manual/latest/record"

Record

Records are like JavaScript objects but:

- are immutable by default
- have fixed fields (not extensible)

Type Declaration

A record needs a mandatory type declaration:

```res prelude type person = { age: int, name: string, }

```js
// Empty output
```

Creation

To create a `person` record (declared above):

```res prelude let me = { age: 5, name: "Big ReScript" }

```js
var me = {
  age: 5,
  name: "Big ReScript"
};
```

When you create a new record value, ReScript tries to find a record type declaration that conforms to the shape of the value. So the `me` value here is inferred as of type `person`.

The type is found by looking above the `me` value. **Note**: if the type instead resides in another file or module, you need to explicitly indicate which file or module it is:

```res example // School.res type person = {age: int, name: string}

```js
// Empty output

// Example.res

let me: School.person = {age: 20, name: "Big ReScript"}
/* or */
let me2 = {School.age: 20, name: "Big ReScript"}

var me = {
  age: 20,
  name: "Big ReScript"
};
var me2 = {
  age: 20,
  name: "Big ReScript"
};
```

In both `me` and `me2` the record definition from `School` is found. The first one, `me` with the regular type annotation, is preferred.

Access

Use the familiar dot notation:

```res example let name = me.name

```js
var name = "Big ReScript";
```

Immutable Update

New records can be created from old records with the `...` spread operator. The original record isn't mutated.

```res example let meNextYear = {...me, age: me.age + 1}

```js
var meNextYear = {
  age: 21,
  name: "Big ReScript"
};
```

Note: spread cannot add new fields to the record value, as a record's shape is fixed by its type.

Mutable Update

Record fields can optionally be mutable. This allows you to efficiently update those fields in-place with the `=` operator.

```res example type person = { name: string, mutable age: int }

let baby = {name: "Baby ReScript", age: 5} baby.age = baby.age + 1 // `baby.age` is now 6. Happy birthday!

```js
var baby = {
  name: "Baby ReScript",
  age: 5
};

baby.age = baby.age + 1 | 0;
```

Fields not marked with `mutable` in the type declaration cannot be mutated.

JavaScript Output

ReScript records compile to straightforward JavaScript objects; see the various JS output tabs above.

Optional Record Fields

ReScript `v10` introduced optional record fields. This means that you can define fields that can be omitted when creating the record. It looks like this:

```res example type person = { age: int, name?: string }

```js
// Empty output
```

Notice how `name` has a suffixed `?`. That means that the field itself is *optional*.
```

Creation

You can omit any optional fields when creating a record. Not setting an optional field will default the field's value to `None`:

```res example type person = { age: int, name?: string }

let me = { age: 5, name: "Big ReScript" }

let friend = { age: 7 }
```

```js
var me = {
  age: 5,
  name: "Big ReScript"
};

var friend = {
  age: 7
};
```

This has consequences for pattern matching, which we'll expand a bit on soon.

Immutable Update

Updating an optional field via an immutable update above lets you set that field value without needing to care whether it's optional or not.

```res example type person = { age: int, name?: string }

let me = { age: 123, name: "Hello" }

let withoutName = { ...me, name: "New Name" }
```

```js
import * as Caml_obj from "./stdlib/caml_obj.js";

var me = {
  age: 123,
  name: "Hello"
};

var newrecord = Caml_obj.obj_dup(me);

newrecord.name = "New Name";

var withoutName = newrecord;
```

However, if you want to set the field to an optional value, you prefix that value with `?`:

```res example type person = { age: int, name?: string }

let me = { age: 123, name: "Hello" }

let maybeName = Some("My Name")

let withoutName = { ...me, name: ?maybeName }
```

```js
import * as Caml_obj from "./stdlib/caml_obj.js";

var me = {
  age: 123,
  name: "Hello"
};

var maybeName = "My Name";

var newrecord = Caml_obj.obj_dup(me);

newrecord.name = maybeName;

var withoutName = newrecord;
```

You can unset an optional field's value via that same mechanism by setting it to `?None`.

Pattern Matching on Optional Fields

Pattern matching, one of ReScript's most important features, has two caveats when you deal with optional fields.

When matching on the value directly, it's an `option`. Example:

```
type person = {
  age: int,
  name?: string,
}

let me = {
  age: 123,
  name: "Hello",
}

let isRescript = switch me.name {
| Some("ReScript") => true
| Some(_) | None => false
}
```

```
var isRescript;

isRescript = "Hello" === "ReScript" ? true : false;

var me = {
  age: 123,
  name: "Hello"
};
```

But, when matching on the field as part of the general record structure, it's treated as the underlying, non-optional value:

```
type person = {
  age: int,
  name?: string,
}

let me = {
  age: 123,
  name: "Hello",
}

let isRescript = switch me {
| {name: "ReScript"} => true
| _ => false
}
```

```
var isRescript;

isRescript = "Hello" === "ReScript" ? true : false;

var me = {
  age: 123,
  name: "Hello"
};
```

Sometimes you *do* want to know whether the field was set or not. You can tell the pattern matching engine about that by prefixing your option match with `?`, like this:

```
type person = {
  age: int,
  name?: string,
}

let me = {
  age: 123,
  name: "Hello",
}

let nameWasSet = switch me {
| {name: ?None} => false
| {name: ?Some(_)} => true
}
```

```
var nameWasSet = true;

var me = {
  age: 123,
  name: "Hello"
};
```

Record Type Spread

In ReScript v11, you can now spread one or more record types into a new record type. It looks like this:

```
type a = {
  id: string,
  name: string,
}

type b = {
```

```
  age: int
}

type c = {
  ...a,
  ...b,
  active: bool
}
```

`type c` will now be:

```
type c = {
  id: string,
  name: string,
  age: int,
  active: bool,
}
```

Record type spreads act as a 'copy-paste' mechanism for fields from one or more records into a new record. This operation inlines the fields from the spread records directly into the new record definition, while preserving their original properties, such as whether they are optional or mandatory. It's important to note that duplicate field names are not allowed across the records being spread, even if the fields have the same type.

Record Type Coercion

Record type coercion gives us more flexibility when passing around records in our application code. In other words, we can now coerce a record `a` to be treated as a record `b` at the type level, as long as the original record `a` contains the same set of fields in `b`. Here's an example:

```
type a = {
  name: string,
  age: int,
}

type b = {
  name: string,
  age: int,
}

let nameFromB = (b: b) => b.name

let a: a = {
  name: "Name",
  age: 35,
}

let name = nameFromB(a :> b)
```

Notice how we *coerced* the value `a` to type `b` using the coercion operator `:>`. This works because they have the same record fields. This is purely at the type level, and does not involve any runtime operations.

Additionally, we can also coerce records from `a` to `b` whenever `a` is a super-set of `b` (i.e. `a` containing all the fields of `b`, and more). The same example as above, slightly altered:

```
type a = {
  id: string,
  name: string,
  age: int,
  active: bool,
}

type b = {
  name: string,
  age: int,
}

let nameFromB = (b: b) => b.name

let a: a = {
  id: "1",
  name: "Name",
  age: 35,
  active: true,
}

let name = nameFromB(a :> b)
```

Notice how `a` now has more fields than `b`, but we can still coerce `a` to `b` because `b` has a subset of the fields of `a`.

In combination with optional record fields, one may coerce a mandatory field of an `option` type to an optional field:

```
type a = {
```

```
  name: string,

  // mandatory, but explicitly typed as option<int>
  age: option<int>,
}

type b = {
  name: string,
  // optional field
  age?: int,
}

let nameFromB = (b: b) => b.name

let a: a = {
  name: "Name",
  age: Some(35),
}

let name = nameFromB(a :> b)
```

Tips & Tricks

Record Types Are Found By Field Name

With records, you **cannot** say "I'd like this function to take any record type, as long as they have the field `age`". The following **won't work as intended**:

```
type person = {age: int, name: string}
type monster = {age: int, hasTentacles: bool}

let getAge = (entity) => entity.age

function getAge(entity) {
  return entity.age;
}
```

Instead, `getAge` will infer that the parameter `entity` must be of type `monster`, the closest record type with the field `age`. The following code's last line fails:

```
let kraken = {age: 9999, hasTentacles: true}
let me = {age: 5, name: "Baby ReScript"}

getAge(kraken)
getAge(me) // type error!
```

The type system will complain that `me` is a `person`, and that `getAge` only works on `monster`. If you need such capability, use ReScript objects, described here.

Optional Fields in Records Can Be Useful for Bindings

Many JavaScript APIs tend to have large configuration objects that can be a bit annoying to model as records, since you previously always needed to specify all record fields when creating a record.

Optional record fields, introduced in `v10`, is intended to help with this. Optional fields will let you avoid having to specify all fields, and let you just specify the one's you care about. A significant improvement in ergonomics for bindings and other APIs with for example large configuration objects.

Design Decisions

After reading the constraints in the previous sections, and if you're coming from a dynamic language background, you might be wondering why one would bother with record in the first place instead of straight using object, since the former needs explicit typing and doesn't allow different records with the same field name to be passed to the same function, etc.

1. The truth is that most of the times in your app, your data's shape is actually fixed, and if it's not, it can potentially be better represented as a combination of variant (introduced next) + record instead.
2. Since a record type is resolved through finding that single explicit type declaration (we call this "nominal typing"), the type error messages end up better than the counterpart ("structural typing", like for tuples). This makes refactoring easier; changing a record type's fields naturally allows the compiler to know that it's still the same record, just misused in some places. Otherwise, under structural typing, it might get hard to tell whether the definition site or the usage site is wrong.

SECTION bind-to-global-js-values

---

title: "Bind to Global JS Values" description: "JS interop with global JS values in ReScript" canonical: "/docs/manual/latest/bind-to-global-js-values"

Bind to Global JS Values

First, make sure the value you'd like to model doesn't already exist in our provided API.

Some JS values, like `setTimeout`, live in the global scope. You can bind to them like so:

```res example @val external setTimeout: (unit => unit, int) => float = "setTimeout" @val external clearTimeout: float => unit = "clearTimeout"

```js
// Empty output
```

(We already provide `setTimeout`, `clearTimeout` and others in the Js.Global module).

This binds to the JavaScript `setTimeout` methods and the corresponding `clearTimeout`. The `external`'s type annotation specifies that `setTimeout`:

- Takes a function that accepts `unit` and returns `unit` (which on the JS side turns into a function that accepts nothing and returns nothing aka `undefined`),
- and an integer that specifies the duration before calling said function,
- returns a number that is the timeout's ID. This number might be big, so we're modeling it as a float rather than the 32-bit int.

Tips & Tricks

The above isn't ideal. See how `setTimeout` returns a `float` and `clearTimeout` accepts one. There's no guarantee that you're passing the float created by `setTimeout` into `clearTimeout`! For all we know, someone might pass it `Math.random()` into the latter.

We're in a language with a great type system now! Let's leverage a popular feature to solve this problem: abstract types.

```res example type timerId @val external setTimeout: (unit => unit, int) => timerId = "setTimeout" @val external clearTimeout: timerId => unit = "clearTimeout"

let id = setTimeout(() => Js.log("hello"), 100) clearTimeout(id)

```js
var id = setTimeout(function (param) {
  console.log("hello");
}, 100);

clearTimeout(id);
```

Clearly, `timerId` is a type that can only be created by `setTimeout`! Now we've guaranteed that `clearTimeout` *will* be passed a valid ID. Whether it's a number under the hood is now a mere implementation detail.

Since `external`s are inlined, we end up with JS output as readable as hand-written JS.

Global Modules

If you want to bind to a value inside a global module, e.g. `Math.random`, attach a `scope` to your `val` external:

```res example @scope("Math") @val external random: unit => float = "random" let someNumber = random()

```js
var someNumber = Math.random();
```

you can bind to an arbitrarily deep object by passing a tuple to `scope`:

```res example @val @scope(("window", "location", "ancestorOrigins")) external length: int = "length"

```js
// Empty output
```

This binds to `window.location.ancestorOrigins.length`.

Special Global Values

Global values like `__filename` and `__DEV__` don't always exist; you can't even model them as an `option`, since the mere act of referring to them in ReScript (then compiled into JS) would trigger the usual `Uncaught ReferenceError: __filename is not defined` error in e.g. the browser environment.

For these troublesome global values, ReScript provides a special approach: `%external(a_single_identifier)`.

```res example switch %external(**DEV**) { | Some(_) => Js.log("dev mode") | None => Js.log("production mode") }

```js
var match = typeof __DEV__ === "undefined" ? undefined : __DEV__;
```
```

```
if (match !== undefined) {
  console.log("dev mode");
} else {
  console.log("production mode");
}
```

That first line's `typeof` check won't trigger a JS ReferenceError.

Another example:

```res example switch %external(__filename) { | Some(f) => Js.log(f) | None => Js.log("non-node environment") };

```js
var match = typeof (__filename) === "undefined" ? undefined : (__filename);

if (match !== undefined) {
  console.log(match);
} else {
  console.log("non-node environment");
}
```

SECTION inlining-constants

---

title: "Inlining Constants" description: "Inlining constants" canonical: "/docs/manual/latest/inlining-constants"

Inlining Constants

Sometime, in the JavaScript output, you might want a certain value to be forcefully inlined. For example:

```
if (process.env.mode === 'development') {
  console.log("Dev-only code here!")
}
```

The reason is that your JavaScript bundler (e.g. Webpack) might turn that into:

```
if ('production' === 'development') {
  console.log("Dev-only code here!")
}
```

Then your subsequent Uglifyjs optimization would remove that entire `if` block. This is how projects like ReactJS provide a development mode code with plenty of dev warnings, while ensuring that the uglified (minified) production code is free of those expensive blocks.

So, in ReScript, producing that example `if (process.env.mode === 'development')` output is important. This first try doesn't work:

```res example @val external process: 'a = "process"

let mode = "development"

if (process["env"]["mode"] === mode) { Js.log("Dev-only code here!") }

```js
var mode = "development";

if (process.env.mode === mode) {
  console.log("Dev-only code here!");
}
```

The JS output shows `if (process.env.mode === mode)`, which isn't what we wanted. To inline `mode`'s value, use `@inline`:

```res example @val external process: 'a = "process"

@inline let mode = "development"

if (process["env"]["mode"] === mode) { Js.log("Dev-only code here!") }

```js
if (process.env.mode === "development") {
  console.log("Dev-only code here!");
}
```

Now your resulting JS code can pass through Webpack and Uglifyjs like the rest of your JavaScript code, and that whole `console.log` can be removed.

The inlining currently only works for **string, float and boolean**.
```

Tips & Tricks

This is **not** an optimization. This is an edge-case feature for folks who absolutely need particular values inlined for a JavaScript post-processing step, like conditional compilation. Beside the difference in code that the conditional compilation might end up outputting, there's no performance difference between inlining and not inlining simple values in the eyes of a JavaScript engine.

SECTION introduction

---

title: "Introduction" description: "The hows and whys of ReScript" canonical: "/docs/manual/latest/introduction"

ReScript

Ever wanted a language like JavaScript, but without the warts, with a great type system, and with a lean build toolchain that doesn't waste your time?

ReScript looks like JS, acts like JS, and compiles to the highest quality of clean, readable and performant JS, directly runnable in browsers and Node.

This means you can pick up ReScript and access the vast JavaScript ecosystem and tooling as if you've known ReScript for a long time!

ReScript is the language for folks who don't necessarily love JavaScript, but who still acknowledge its importance.

Difference vs TypeScript

We respect TypeScript very much and think that it's a positive force in the JavaScript ecosystem. ReScript shares some of the same goals as TypeScript, but is different enough regarding some important nuances:

- TypeScript's (admittedly noble) goal is to cover the entire JavaScript feature set and more. **ReScript covers only a curated subset of JavaScript**. For example, we emphasize plain data + functions over classes, clean pattern matching over fragile `if`s and virtual dispatches, proper data modeling over string abuse, etc. JavaScript supersets will only grow larger over time; ReScript doesn't. *

- Consequently, TypeScript's type system is necessarily complex, pitfalls-ridden, potentially requires tweaking, sometimes slow, and requires quite a bit of noisy annotations that often feel like manual bookkeeping rather than clear documentation. In contrast, ReScript's type system:

- Is deliberately curated to be a simple subset most folks will have an easier time to use.

- Has **no** pitfalls, aka the type system is "sound" (the types will always be correct). E.g. If a type isn't marked as nullable, its value will never lie and let through some `undefined` value silently. **ReScript code has no null/undefined errors**.
- Is the same for everyone. No knobs, no bikeshedding opportunity.
- Runs extremely fast precisely thanks to its simplicity and curation. It's one of the fastest compiler & build system toolchains for JavaScript development.

- **Doesn't need type annotations**. Annotate as much or as little as you'd like. The types are inferred by the language (and, again, are guaranteed correct).

- Migrating to TypeScript is done "breadth-first," whereas migrating to ReScript is done "depth-first." You can convert your codebase to TypeScript by "turning it on" for all files and annotate here and there; but how much type safety did you gain? How do you measure it? Type errors can still slip in and out of the converted pieces. For ReScript, our interop features draw clear boundaries: there's pure ReScript code, and there's JS interop code. Every piece of converted ReScript code is 100% clean. You'd convert file by file and each conversion increases your safety monotonically.

* When you absolutely need to write or interoperate with free-for-all JavaScript, we expose enough escape hatches for you.

Other Highlights

Aside from the aforementioned simple, robust and fast type system, ReScript presents a few more advantages.

Faster than JavaScript

JavaScript's been aggressively optimized by talented engineers over a long span. Unfortunately, even for seasoned JS devs, it can be hard to know how to properly leverage JS's performance. ReScript's type system and compiler naturally guides you toward writing code that's very often performant by default, with good leverage of various Just-In-Time optimizations (hidden classes, inline caching, avoiding deopts, etc).

A widespread adage to write fast JavaScript code is to write as if there's a type system (in order to trigger JS engines' good optimization heuristics); ReScript gives you a real one and generates code that's friendly to optimizations by default.

High Quality Dead Code Elimination

The JavaScript ecosystem is very reliant on dependencies. Shipping the final product inevitably drags in a huge amount of code, lots of which the project doesn't actually use. These regions of dead code impact loading, parsing and interpretation speed. ReScript provides powerful dead code elimination at all levels:

- Function- and module-level code elimination is facilitated by the well-engineered type system and purity analysis.
- At the global level, ReScript generates code that is naturally friendly to dead code elimination done by bundling tools such as Rollup and Closure Compiler, after its own sophisticated elimination pass.
- The same applies for ReScript's own tiny runtime (which is written in ReScript itself).

Tiny JS Output

A `Hello world` ReScript program generates **20 bytes** of JS code. Additionally, the standard library pieces you require in are only included when needed.

Fast Iteration Loop

ReScript's build time is **one or two orders of magnitude** faster than alternatives. In its watcher mode, the build system usually finishes before you switch screen from the editor to the terminal tab (two digits of milliseconds). A fast iteration cycle reduces the need of keeping one's mental state around longer; this in turn allows one to stay in the flow longer and more often.

Readable Output & Great Interop

Unreadable JavaScript code generated from other compiled-to-js languages makes it so that it could be, practically speaking:

- Hard to debug (cryptic stack trace, mangled variable names)
- Hard to learn from (non-straightforward mapping of concepts from one language to another)
- Hard to profile for performance (unclear what runtime performance cost there is)
- Hard to integrate with existing hand-written JS code

ReScript's JS output is very readable. This is especially important while learning, where users might want to understand how the code's compiled, and to audit for bugs.

This characteristic, combined with a fully-featured JS interop system, allows ReScript code to be inserted into an existing JavaScript codebase almost unnoticed.

Preservation of Code Structure

ReScript maps one source file to one JavaScript output file. This eases the integration of existing tools such as bundlers and test runners. You can even start writing a single file without much change to your build setup. Each file's code structure is approximately preserved, too.

Conclusion

We hope the above gave you enough of an idea of ReScript and its differentiators. Feel free to try it online to get a feel!

SECTION control-flow

---

title: "If-Else & Loops" description: "If, else, ternary, for, and while" canonical: "/docs/manual/latest/control-flow"

If-Else & Loops

ReScript supports `if`, `else`, ternary expression (`a ? b : c`), `for` and `while`.

ReScript also supports our famous pattern matching, which will be covered in its own section

If-Else & Ternary

Unlike its JavaScript counterpart, ReScript's `if` is an expression; they evaluate to their body's content:

```
let message = if isMorning {
  "Good morning!"
} else {
  "Hello!"
}
```

```
var message = isMorning ? "Good morning!" : "Hello!";
```

Note: an `if-else` expression without the final `else` branch implicitly gives `()` (aka the `unit` type). So this:

```
if showMenu {
  displayMenu()
}

if (showMenu) {
  displayMenu();
}
```

is basically the same as:

```
if showMenu {
  displayMenu()
} else {
  ()
}

if (showMenu) {
  displayMenu()
}
```

Here's another way to look at it. This is clearly wrong:

```
let result = if showMenu {
  1 + 2
}
```

It'll give a type error, saying basically that the implicit `else` branch has the type `unit` while the `if` branch has type `int`. Intuitively, this makes sense: what would `result`'s value be, if `showMenu` was `false`?

We also have ternary sugar, but **we encourage you to prefer if-else when possible**.

```
let message = isMorning ? "Good morning!" : "Hello!"

var message = isMorning ? "Good morning!" : "Hello!";
```

`if-else` and ternary are much less used in ReScript than in other languages; Pattern-matching kills a whole category of code that previously required conditionals.

For Loops

For loops iterate from a starting value up to (and including) the ending value.

```
for i in startValueInclusive to endValueInclusive {
  Js.log(i)
}

for(var i = startValueInclusive; i <= endValueInclusive; ++i){
  console.log(i);
}
```

```res example // prints: 1 2 3, one per line for x in 1 to 3 { Js.log(x) }

```js
for(var x = 1; x <= 3; ++x){
  console.log(x);
}
```

You can make the `for` loop count in the opposite direction by using `downto`.

```
for i in startValueInclusive downto endValueInclusive {
  Js.log(i)
}

for(var i = startValueInclusive; i >= endValueInclusive; --i){
  console.log(i);
}
```

```res example // prints: 3 2 1, one per line for x in 3 downto 1 { Js.log(x) }

```js
for(var x = 3; x >= 1; --x){
  console.log(x);
}
```

While Loops

While loops execute its body code block while its condition is true.

```
while testCondition {
  // body here
}
```

```
while (testCondition) {
  // body here
}
```

Tips & Tricks

There's no loop-breaking `break` keyword (nor early `return` from functions, for that matter) in ReScript. However, we can break out of a while loop easily through using a mutable binding.

```res example let break = ref(false)

while !break.contents { if Js.Math.random() > 0.3 { break := true } else { Js.log("Still running") } }
```

```js
var $$break = {
  contents: false
};

while(!$$break.contents) {
  if (Math.random() > 0.3) {
    $$break.contents = true;
  } else {
    console.log("Still running");
  }
};
```

SECTION project-structure

---

title: "Project Structure" description: "Notes on project structure and other rough ReScript guidelines" canonical: "/docs/manual/latest/project-structure"

Project Structure

These are the existing, non-codified community practices that are currently propagated through informal agreement. We might remove some of them at one point, and enforce some others. Right now, they're just recommendations for ease of newcomers.

File Casing

Capitalized file names (aka first letter upper-cased).

Justification: Module names can only be capitalized. Newcomers often ask how a file maps to a module, and why `draw.res` maps to the module `Draw`, and sometimes try to refer to a module through uncapitalized identifiers. Using `Draw.res` makes this mapping more straightforward. It also helps certain file names that'd be awkward in uncapitalized form: `uRI.res`.

Ignore `.merlin` File

This is generated by the build system and you should not have to manually edit it. Don't check it into the repo.

Justification: `.merlin` is for editor tooling. The file contains absolute paths, which are also not cross-platform (e.g. Windows paths are different).

Folders

Try not to have too many nested folders. Keep your project flat, and have fewer files (reminder: you can use nested modules).

Justification: The file system is a *tree*, but your code's dependencies are a *graph*. Because of that, any file & folder organization is usually imperfect. While it's still valuable to group related files together in a folder, the time wasted debating & getting decision paralysis over these far outweight their benefits. We'll always recommend you to Get Work Done instead of debating about these issues.

Third-party Dependencies

Keep them to a minimum.

Justification: A compiled, statically typed language cannot model its dependencies easily by muddling along like in a dynamic language, especially when we're still piggy-backing on NPM/Yarn (to reduce learning overhead in the medium-term). Keeping dependencies simple & lean helps reduce possibility of conflicts (e.g. two diamond dependencies, or clashing interfaces).

Documentation

Have them. Spend more effort making them great (examples, pitfalls) and professional rather than *just* fancy-looking. Do use examples, and avoid using names such as `foo` and `bar`. There's always more concrete names (it's an example, no need to be abstract/generalized just

yet. The API docs will do this plentily). For blog posts, don't repeat the docs themselves, describe the *transition* from old to new, and why (e.g. "it was a component, now it's a function, because ...").

Justification: It's hard for newcomers to distinguish between a simple/decent library and one that's fancy-looking. For the sake of the community, don't try too hard to one-up each other's libraries. Do spread the words, but use your judgement too.

PPX & Other Meta-tools

Keep them to a minimum. PPX, unless used in renown cases (printer, accessors and serializer/deserializer generation), can cause big learning churn for newcomers; on top of the syntax, semantics, types, build tool & FFI that they already have to learn, learning per-library custom transformations of the code is an extra step. More invasive macros makes the code itself less semantically meaningful too, since the essence would be hiding somewhere else.

Paradigm

Don't abuse overly fancy features. Do leave some breathing room for future APIs but don't over-architect things.

Justification: Simple code helps newcomers understand and potentially contribute to your code. Contributing is the best way for them to learn. The extra help you receive might also surpass the gain of using a slightly more clever language trick. But do try new language tricks in some of more casual projects! You might discover new ways of architecting code.

Publishing

If it's a wrapper for a JS library, don't publish the JS artifacts. If it's a legit library, publish the artifacts in lib/js if you think JS consumers might use it. This is especially the case when you gradually convert a JS lib to ReScript while not breaking existing JS consumers.

Do put the keywords `"rescript"` in your package.json `keywords` field. This allows us to find the library much more easily for future purposes.

Justification: Be nice to JS consumers of your library. They're your future ReScripters.

SECTION build-external-stdlib

---

title: "External Stdlib" metaTitle: "External Stdlib" description: "Configuring an external ReScript stdlib package" canonical: "/docs/manual/latest/build-external-stdlib"

External Stdlib

Since 9.0

Your ReScript project depends on the `rescript` package as a `devDependency`, which includes our compiler, build system and runtime like `Belt`. However, you had to move it to `dependency` in `package.json` if you publish your code: - To Docker or other low-storage deployment devices. - For pure JS/TS consumers who probably won't install `bs-platform` in their own project.

In these cases, the size or mere presence of `bs-platform` can be troublesome, since it includes not just our necessary runtime like `Belt`, but also our compiler and build system.

To solve that, we now publish our runtime as a standalone package at `@rescript/std`, whose versions mirror `bs-platform`'s. Now you can keep `bs-platform` as a `devDependency` and have only `@rescript/std` as your runtime `dependency`.

This is an advanced feature. Please only use it in the aforementioned scenarios. If you already use a JS bundler with dead code elimination, you might not need this feature.

Configuration

Say you want to publish a JS-only ReScript 9.0 library. Install the packages like this:

```
npm install bs-platform@9.0.0 --save-dev
npm install @rescript/std@9.0.0
```

Then add this to `rescript.json`:

```
{
  // ...
  "external-stdlib" : "@rescript/std"
}
```

Now the compiled JS code will import using the path defined by `external-stdlib`. Check the JS output tab:

```
Array.forEach([1, 2, 3], num => Js.log(num))
```

```
// Note the require path starting with "@rescript/std".
var Belt_Array = require("@rescript/std/lib/js/belt_Array.js");

Belt_Array.forEach([1, 2, 3], function (num) {
  console.log(num);
});
```

Make sure the version number of `bs-platform` and `@rescript/std` match in your `package.json` to avoid running into runtime issues due to mismatching stdlib assumptions.

SECTION scoped-polymorphic-types

---

title: "Scoped Polymorphic Types" description: "Scoped Polymorphic Types in ReScript" canonical: "/docs/manual/latest/scoped-polymorphic-types"

Scoped Polymorphic Types

Scoped Polymorphic Types in ReScript are functions with the capability to handle arguments of any type within a specific scope. This feature is particularly valuable when working with JavaScript APIs, as it allows your functions to accommodate diverse data types while preserving ReScript's strong type checking.

Definition and Usage

Scoped polymorphic types in ReScript offer a flexible and type-safe way to handle diverse data types within specific scopes. This documentation provides an example to illustrate their usage in a JavaScript context.

Example: Logging API

Consider a logging example within a JavaScript context that processes various data types:

```
const logger = {
  log: (data) => {
    if (typeof data === "string") {
      /* handle string */
    } else if (typeof data === "number") {
      /* handle number */
    } else {
      /* handle other types */
    }
  },
};
```

In ReScript, we can bind to this function as a record with a scoped polymorphic function type:

```res prelude type logger = { log: 'a. 'a => unit }

@module("jsAPI") external getLogger: unit => logger = "getLogger"

The `logger` type represents a record with a single field `log`, which is a scoped polymorphic function type `'a
. 'a => unit`. The `'a` indicates a type variable that can be any type within the scope of the `log` function.

Now, we can utilize the function obtained from `getLogger`:

<CodeTab labels={["ReScript", "JS Output"]}>

```res example
let myLogger = getLogger()

myLogger.log("Hello, ReScript!")
myLogger.log(42)
```

```
var myLogger = JsAPI.getLogger();

myLogger.log("Hello, ReScript!");
myLogger.log(42);
```

In this example, we create an instance of the logger by calling `getLogger()`, and then we can use the `log` function on the `myLogger` object to handle different data types.

Limitations of Normal Polymorphic Types

Let's consider the same logging example in ReScript, but this time using normal polymorphic types:

```
type logger<'a> = { log: 'a => unit}

@module("jsAPI") external getLogger: unit => logger<'a> = "getLogger"
```
```

In this case, the `logger` type is a simple polymorphic function type `'a => unit`. However, when we attempt to use this type in the same way as before, we encounter an issue:

```
let myLogger = getLogger()

myLogger.log("Hello, ReScript!")
myLogger.log(42) // Type error!
```

The problem arises because the type inference in ReScript assigns a concrete type to the `logger` function based on the first usage. In this example, after the first call to `myLogger`, the compiler infers the type `logger<string>` for `myLogger`. Consequently, when we attempt to pass an argument of type `number` in the next line, a type error occurs because it conflicts with the inferred type `logger<string>`.

In contrast, scoped polymorphic types, such as `'a. 'a => unit`, overcome this limitation by allowing type variables within the scope of the function. They ensure that the type of the argument is preserved consistently within that scope, regardless of the specific value used in the first invocation.

Limitations of Scoped Polymorphic Types

Scoped polymorphic types work only when they are directly applied to let-bindings or record fields (as demonstrated in the logger example above). They can neither be applied to function bodies, nor to separate type definitions:

```
exception Abort

let testExn: 'a. unit => 'a = () => raise(Abort) // Works!

let testExn2 = (): 'a. 'a = raise(Abort) // Syntax error!
type fn = 'a. 'a => unit // Syntax error!
```

SECTION build-configuration

---

title: "Configuration" metaTitle: "Build System Configuration" description: "Details about the configuration of the ReScript build system (rescript.json)" canonical: "/docs/manual/latest/build-configuration"

Configuration

`rescript.json` (or `bsconfig.json` in versions prior ReScript 11) is the single, mandatory build meta file needed for `rescript`.

The complete configuration schema is [here](). We'll *non-exhaustively* highlight the important parts in prose below.

name, namespace

`name` is the name of the library, used as its "namespace". You can activate namespacing through `"namespace": true` in your `rescript.json`. Namespacing is almost **mandatory**; we haven't turned it on by default yet to preserve backward-compatibility.

Explanation: by default, your files, once used as a third-party dependency, are available globally to the consumer. E.g. if you have an `Util.re` and the consumer also has a file of the same name, they will clash. Turning on `namespace` avoids this by wrapping all your own project's files into an extra module layer; instead of a global `Util` module, the consumer will see you as `MyProject.Util`. **The namespacing affects your consumers, not yourself**.

Aka, in ReScript, "namespace" is just a fancy term for an auto-generated module that wraps all your project's files (efficiently and correctly, of course!) for third-party consumption.

We don't do folder-level namespacing for your own project; all your own file names must be unique. This is a constraint that enables several features such as fast search and easier project reorganization.

Note: the `rescript.json` `name` should be the same as the `package.json` `name`, to avoid confusing corner-cases. However, this means that you can't use a camelCased names such as `MyProject`, since `package.json` and npm forbid you to do so (some file systems are case-insensitive). To have the namespace/module as `MyProject`, write `"name": "my-project"`. ReScript will turn that into the camelCased name correctly.

Note on custom namespacing: if for some reason, you need a namespace that is different from what your `name` will produce, you can directly send a string to the `namespace` option. For example, if your package is a binding named `bs-some-thing`, you can use `"namespace": "some-thing"` to get `SomeThing` namespace instead of `BsSomeThing`.

sources

Your source files need to be specified explicitly (we don't want to accidentally drill down into some unrelated directories). Examples:

```
{
  "sources": ["src", "examples"]
}
```

```
{
  "sources": {
    "dir": "src",
    "subdirs": ["page"]
  }
}
```

```
{
  "sources": [
    "examples",
    {
      "dir": "src",
      "subdirs": true // recursively builds every subdirectory
    }
  ]
}
```

You can mark your directories as dev-only (for e.g. tests). These won't be built and exposed to third-parties, or even to other "dev" directories in the same project:

```
{
  "sources" : {
    "dir" : "test",
    "type" : "dev"
  }
}
```

You can also explicitly allow which modules can be seen from outside. This feature is especially useful for library authors who want to have a single entry point for their users.
Here, the file `src/MyMainModule.res` is exposed to outside consumers, while all other files are private.

```
{
  "sources": {
    "dir": "src",
    "public": ["MyMainModule"]
  },
}
```

bs-dependencies, bs-dev-dependencies

List of ReScript dependencies. Just like `package.json`'s dependencies, they'll be searched in `node_modules`.

Note that only sources marked with `"type":"dev"` will be able to resolve modules from `bs-dev-dependencies`.

pinned-dependencies

Since 8.4: List of pinned dependencies. A pinned dependency will always be rebuilt whenever you build a toplevel package (e.g. your main app) with `rescript`.

This is useful for working on multiple independent ReScript packages simultaneously. More usage details can be found in our dedicated pinned dependencies page.

external-stdlib

Since 9.0: This setting allows depending on an externally built stdlib package (instead of a locally built stdlib runtime). Useful for shipping packages that are only consumed in JS or TS without any dependencies to the ReScript development toolchain.

More details can be found on our external stdlib page.

js-post-build

Hook that's invoked every time a file is recompiled. Good for JS build system interop, but please use it **sparingly**. Calling your custom command for every recompiled file slows down your build and worsens the building experience for even third-party users of your lib.

Example:

```
{
  "js-post-build": {
    "cmd": "/path/to/node ../../postProcessTheFile.js"
  }
}
```

Note that the path resolution for the command (`node` in this case) is done so:

- `/myCommand` is resolved into `/myCommand`
- `package/myCommand` is resolved into `node_modules/package/myCommand`
- `./myCommand` is resolved into `myProjectRoot/myCommand`
- `myCommand` is just called as `myCommand`, aka a globally available executable. But note that ReScript doesn't read into your shell's

environment, so if you put e.g. `node`, it won't find it unless you specify an absolute path. Alternatively, add `#!/usr/local/bin/node` to the top of your script to directly call it without prepending `node`.

The command itself is called from inside `lib/bs`.

package-specs

Output to either CommonJS (the default) or ES6 modules. Example:

```
{
  "package-specs": {
    "module": "commonjs",
    "in-source": true
  }
}
```

- `"module": "es6-global"` resolves `node_modules` using relative paths. Good for development-time usage of ES6 in conjunction with browsers like Safari and Firefox that support ES6 modules today. **No more dev-time bundling**!
- `"in-source": true` generates output alongside source files. If you omit it, it'll generate the artifacts into `lib/js`. The output directory is not configurable otherwise.

This configuration only applies to you, when you develop the project. When the project is used as a third-party library, the consumer's own `rescript.json` `package-specs` overrides the configuration here, logically.

suffix

Since 11.0: The suffix can now be freely chosen. However, we still suggest you stick to the convention and use one of the following: - `".js"` - `".mjs"` - `".cjs"` - `".res.js"` - `".res.mjs"` - `".res.cjs"` - `".bs.js"` - `".bs.mjs"` - `".bs.cjs"`

Currently prefer `.bs.js` for now.

Design Decisions

Generating JS files with the `.bs.js` suffix means that, on the JS side, you can do `const myReScriptFile = require('./theFile.bs')`. The benefits:

- It's immediately clear that we're dealing with a generated JS file here.
- It avoids clashes with a potential `theFile.js` file in the same folder.
- It avoids the need of using a build system loader for ReScript files. This + in-source build means integrating a ReScript project into your pure JS codebase **basically doesn't touch anything in your build pipeline at all**.
- genType requires `bs.js` for compiled JS artifacts. If you are using `genType`, you need to use `bs.js` for now.

uncurried

Since 11.0: While we strongly encourage all users to use uncurried mode, it is still possible to opt out. Just set `"uncurried"` to `false` to get the old behavior back:

```
{
  "uncurried": false
}
```

More details can be found in the blogpost about "Uncurried Mode".

warnings

Selectively turn on/off certain warnings and/or turn them into hard errors. Example:

```
{
  "warnings": {
    "number": "-44-102",
    "error": "+5"
  }
}
```

Turn off warning `44` and `102` (polymorphic comparison). Turn warning `5` (partial application whose result has function type and is ignored) into a hard error.

The warning numbers are shown in the build output when they're triggered. See Warning Numbers for the complete list.

bsc-flags

Extra flags to pass to the compiler. For advanced usages.

Environment Variables

We heavily disrecommend the usage of environment variables, but for certain cases, they're justified.

Error Output Coloring: `NINJA_ANSI_FORCED`

This is mostly for other programmatic usage of `rescript` where outputting colors is not desired.

When `NINJA_ANSI_FORCED` is set to `1`: `rescript` produces color. When `NINJA_ANSI_FORCED` is set to `0`: `rescript` doesn't produce color. When `NINJA_ANSI_FORCED` is not set: `rescript` might or might not produce color, depending on a smart detection of where it's outputted.

> Note that the underlying compiler will always be passed `-color always`. See more details in this issue.

SECTION exception

---

title: "Exception" description: "Exceptions and exception handling in ReScript" canonical: "/docs/manual/latest/exception"

Exception

Exceptions are just a special kind of variant, thrown in **exceptional** cases (don't abuse them!).

Usage

```res prelude let getItem = (item: int) => if (item === 3) { // return the found item here 1 } else { raise(Not_found) }

let result = try { getItem(2) } catch { | Not_found => 0 // Default value if getItem throws }
```

```js
var Caml_js_exceptions = require("./stdlib/caml_js_exceptions.js");

function getItem(item) {
  if (item === 3) {
    return 1;
  }
  throw {
    RE_EXN_ID: "Not_found",
    Error: new Error()
  };
}

var result;

try {
  result = getItem(2);
}
catch (raw_exn){
  var exn = Caml_js_exceptions.internalToOCamlException(raw_exn);
  if (exn.RE_EXN_ID === "Not_found") {
    result = 0;
  } else {
    throw exn;
  }
}
```

Note that the above is just for demonstration purposes; in reality, you'd return an `option<int>` directly from `getItem` and avoid the `try` altogether.

You can directly match on exceptions *while* getting another return value from a function:

```res prelude switch list{1, 2, 3}->List.getExn(4) { | item => Js.log(item) | exception Not_found => Js.log("No such item found!") }
```

```js
var List = require("./stdlib/list.js");
var Caml_js_exceptions = require("./stdlib/caml_js_exceptions.js");

var exit = 0;

var item;

try {
  item = List.find((function (i) {
          return i === 4;
        }), {
        hd: 1,
        tl: {
          hd: 2,
          tl: {
            hd: 3,
```

```
            tl: /* [] */0
          }
        }
      });
  exit = 1;
}
catch (raw_exn){
  var exn = Caml_js_exceptions.internalToOCamlException(raw_exn);
  if (exn.RE_EXN_ID === "Not_found") {
    console.log("No such item found!");
  } else {
    throw exn;
  }
}

if (exit === 1) {
  console.log(item);
}
```

You can also make your own exceptions like you'd make a variant (exceptions need to be capitalized too).

```res example exception InputClosed(string) // later on raise(InputClosed("The stream has closed!"))

```js
var Caml_exceptions = require("./stdlib/caml_exceptions.js");

var InputClosed = Caml_exceptions.create("MyFile.InputClosed");

throw {
  RE_EXN_ID: InputClosed,
  _1: "The stream has closed!",
  Error: new Error()
};
```

Catching JS Exceptions

To distinguish between JavaScript exceptions and ReScript exceptions, ReScript namespaces JS exceptions under the `Js.Exn.Error(payload)` variant. To catch an exception thrown from the JS side:

Throw an exception from JS:

```
// Example.js

exports.someJsFunctionThatThrows = () => {
    throw new Error("A Glitch in the Matrix!");
}
```

Then catch it from ReScript:

```
// import the method in Example.js
@module("./Example")
external someJsFunctionThatThrows: () => unit = "someJsFunctionThatThrows"

try {
  // call the external method
  someJSFunctionThatThrows()
} catch {
| Js.Exn.Error(obj) =>
  switch Js.Exn.message(obj) {
  | Some(m) => Js.log("Caught a JS exception! Message: " ++ m)
  | None => ()
  }
}
```

The `obj` here is of type `Js.Exn.t`, intentionally opaque to disallow illegal operations. To operate on `obj`, do like the code above by using the standard library's `Js.Exn` module's helpers.

Raise a JS Exception

`raise(MyException)` raises a ReScript exception. To raise a JavaScript exception (whatever your purpose is), use `Js.Exn.raiseError`:

```res example let myTest = () => { Js.Exn.raiseError("Hello!") }

```js
var Js_exn = require("./stdlib/js_exn.js");

function myTest() {
  return Js_exn.raiseError("Hello!");
}
```

Then you can catch it from the JS side:

```
// after importing `myTest`...
try {
  myTest()
} catch (e) {
  console.log(e.message) // "Hello!"
}
```

Catch ReScript Exceptions from JS

The previous section is less useful than you think; to let your JS code work with your exception-throwing ReScript code, the latter doesn't actually need to throw a JS exception. ReScript exceptions can be used by JS code!

```res example exception BadArgument({myMessage: string})

let myTest = () => { raise(BadArgument({myMessage: "Oops!"})) }

```js
var Caml_exceptions = require("./stdlib/caml_exceptions.js");

var BadArgument = Caml_exceptions.create("Playground.BadArgument");

function myTest() {
  throw {
    RE_EXN_ID: BadArgument,
    myMessage: "Oops!",
    Error: new Error()
  };
}
```

Then, in your JS:

```
// after importing `myTest`...
try {
  myTest()
} catch (e) {
  console.log(e.myMessage) // "Oops!"
  console.log(e.Error.stack) // the stack trace
}
```

> Note: `RE_EXN_ID` is an internal field for bookkeeping purposes. Don't use it on the JS side. Use the other fields.

The above `BadArgument` exception takes an inline record type. We special-case compile the exception as `{RE_EXN_ID, myMessage, Error}` for good ergonomics. If the exception instead took ordinary positional arguments, l like the standard library's `Invalid_argument("Oops!")`, which takes a single argument, the argument is compiled to JS as the field `_1` instead. A second positional argument would compile to `_2`, etc.

Tips & Tricks

When you have ordinary variants, you often don't **need** exceptions. For example, instead of throwing when `item` can't be found in a collection, try to return an `option<item>` (`None` in this case) instead.

Catch Both ReScript and JS Exceptions in the Same `catch` Clause

```
try {
  someOtherJSFunctionThatThrows()
} catch {
| Not_found => ... // catch a ReScript exception
| Invalid_argument(_) => ... // catch a second ReScript exception
| Js.Exn.Error(obj) => ... // catch the JS exception
}
```

This technically works, but hopefully you don't ever have to work with such code...

SECTION editor-plugins

---

title: "Editor Plugins" description: "List of ReScript editor plugins" canonical: "/docs/manual/latest/editor-plugins"

Editor Plugins

- VSCode
- Sublime Text
- Vim/Neovim

Community Supported
```

We don't officially support these; use them at your own risk!

- Neovim Tree-sitter
- IDEA
- Emacs (only legacy `v8.0.0` Reason/OCaml syntax support)

SECTION build-configuration-schema

---

title: "Configuration Schema" metaTitle: "Build System Configuration Schema" description: "Schema exploration widget for the ReScript configuration file" canonical: "/docs/manual/latest/build-configuration-schema"

---

import dynamic from "next/dynamic";

export const Docson = dynamic(() => import("src/components/Docson").then((comp) => { return comp.make; }), { ssr: false, loading: () =>

Loading...
, });

export default function BuildConfigurationSchemaPage() { return ; }

SECTION reserved-keywords

---

title: "Reserved Keyword" description: "All reserved keywords in ReScript" canonical: "/docs/manual/latest/reserved-keywords"

Reserved Keywords

> **Note**: Some of these words are reserved purely for backward compatibility.
>
> If you *need* to use one of these names as binding and/or field name, see Use Illegal Identifier Names.

- `and`
- `as`
- `assert`
- `constraint`
- `else`
- `exception`
- `external`
- `false`
- `for`
- `if`
- `in`
- `include`
- `lazy`
- `let`
- `module`
- `mutable`
- `of`
- `open`
- `rec`
- `switch`
- `true`
- `try`
- `type`

- `when`
- `while`
- `with`

SECTION pipe

---

title: "Pipe" description: "The Pipe operator (->)" canonical: "/docs/manual/latest/pipe"

Pipe

ReScript provides a tiny but surprisingly useful operator `->`, called the "pipe", that allows you to "flip" your code inside-out. `a(b)` becomes `b->a`. It's a simple piece of syntax that doesn't have any runtime cost.

Why would you use it? Imagine you have the following:

```
validateAge(getAge(parseData(person)))
```

```
validateAge(getAge(parseData(person)));
```

This is slightly hard to read, since you need to read the code from the innermost part, to the outer parts. Use pipe to streamline it:

```
person
  ->parseData
  ->getAge
  ->validateAge
```

```
validateAge(getAge(parseData(person)));
```

Basically, `parseData(person)` is transformed into `person->parseData`, and `getAge(person->parseData)` is transformed into `person->parseData->getAge`, etc.

This works when the function takes more than one argument too.

```
a(one, two, three)
```

```
a(one, two, three);
```

is the same as

```
one->a(two, three)
```

```
a(one, two, three);
```

This also works with labeled arguments.

Pipes are used to emulate object-oriented programming. For example, `myStudent.getName` in other languages like Java would be `myStudent->getName` in ReScript (equivalent to `getName(myStudent)`). This allows us to have the readability of OOP without the downside of dragging in a huge class system just to call a function on a piece of data.

Tips & Tricks

Do **not** abuse pipes; they're a means to an end. Inexperienced engineers sometimes shape a library's API to take advantage of the pipe. This is backwards.

JS Method Chaining

This section requires understanding of [our binding API](#).

JavaScript's APIs are often attached to objects, and are often chainable, like so:

```
const result = [1, 2, 3].map(a => a + 1).filter(a => a % 2 === 0);
```

```
asyncRequest()
  .setWaitDuration(4000)
  .send();
```

Assuming we don't need the chaining behavior above, we'd bind to each case this using `@send` from the aforementioned binding API page:

```res prelude type request @val external asyncRequest: unit => request = "asyncRequest" @send external setWaitDuration: (request, int) => request = "setWaitDuration" @send external send: request => unit = "send"
```

```js
// Empty output
```

You'd use them like this:

```res example let result = Js.Array2.filter( Js.Array2.map([1, 2, 3], a => a + 1), a => mod(a, 2) == 0 )

send(setWaitDuration(asyncRequest(), 4000))
```

```js
var result = [1, 2, 3].map(function(a) {
  return a + 1 | 0;
}).filter(function(a) {
  return a % 2 === 0;
});

asyncRequest().setWaitDuration(4000).send();
```

This looks much worse than the JS counterpart! Clean it up visually with pipe:

```res example let result = [1, 2, 3] ->Js.Array2.map(a => a + 1) ->Js.Array2.filter(a => mod(a, 2) == 0)

asyncRequest()->setWaitDuration(4000)->send
```

```js
var result = [1, 2, 3].map(function(a) {
  return a + 1 | 0;
}).filter(function(a) {
  return a % 2 === 0;
});

asyncRequest().setWaitDuration(4000).send();
```

Pipe Into Variants

You can pipe into a variant's constructor as if it was a function:

```
let result = name->preprocess->Some
```

```
var result = preprocess(name);
```

We turn this into:

```
let result = Some(preprocess(name))
```

```
var result = preprocess(name);
```

Note that using a variant constructor as a function wouldn't work anywhere else beside here.

Pipe Placeholders

A placeholder is written as an underscore and it tells ReScript that you want to fill in an argument of a function later. These two have equivalent meaning:

```
let addTo7 = (x) => add3(3, x, 4)
let addTo7 = add3(3, _, 4)
```

Sometimes you don't want to pipe the value you have into the first position. In these cases you can mark a placeholder value to show which argument you would like to pipe into.

Let's say you have a function `namePerson`, which takes a `person` then a `name` argument. If you are transforming a person then pipe will work as-is:

```
makePerson(~age=47)
  ->namePerson("Jane")
```

```
namePerson(makePerson(47), "Jane");
```

If you have a name that you want to apply to a person object, you can use a placeholder:

```
getName(input)
  ->namePerson(personDetails, _)
```

```
var __x = getName(input);
namePerson(personDetails, __x);
```

This allows you to pipe into any positional argument. It also works for named arguments:

```
getName(input)
  ->namePerson(~person=personDetails, ~name=_)
```

```
var __x = getName(input);
namePerson(personDetails, __x);
```

Triangle Pipe (Deprecated)

You might see usages of another pipe, `|>`, in some codebases. These are deprecated.

Unlike `->` pipe, the `|>` pipe puts the subject as the last (not first) argument of the function. `a |> f(b)` turns into `f(b, a)`.

For a more thorough discussion on the rationale and differences between the two operators, please refer to the Data-first and Data-last comparison by Javier ChÃ¡varri

SECTION faq

---

title: "FAQ" description: "Frequently asked questions about ReScript and its ecosystem" canonical: "/docs/manual/latest/faq"

Frequently Asked Questions

What's the goal of this project?

We aim to provide the best typed language experience for the JavaScript platform.

Whatâ€™s the relationship with BuckleScript?

BuckleScript is ReScript's old branding, with a sharper focus on proper JS support and familiarity which we previously couldn't achieve to the degree we wanted, due to us needing to cater to various different crowds.

Whatâ€™s ReScript's relationship with OCaml?

We reuse and adjust the excellent type system and lots of other high quality components from OCaml for JS experience. Additionally, ReScript provides its own syntax, build system, IDE, backend, JS interop, extra language features, etc.

The ReScript toolchain is developed using OCaml, however, the version of ReScript is decoupled against the version of OCaml, ReScript compiler should build against any reasonable modern version of OCaml compiler.

For the majority of ReScript users, they don't need to learn OCaml or use OCaml toolchain to be productive in ReScript.

Whatâ€™s the relationship with Reason?

See here. Reason is a syntax layer for OCaml that BuckleScript also adopted. The current ReScript compiler also supports the old Reason syntax v3.6 for backward compatibility. We will support it for a long time to make sure existing users do not get breaking changes.

I come from Reason/OCaml. Will ReScript keep supporting X?

Please see our blog post on this matter.

Where can I see the docs in old Reason/OCaml syntax?

Switch the doc version to `v8.0.0` in the sidebar on the left!

Will ReScript support native compilation eventually?

Our focus is a solid JS story right now. In the future, if thereâ€™s strong demand, we might consider it.

Whatâ€™s the current state of ReScript?

Currently, we're actively working on the editor support.

When will we get the `async/await` keywords?

See our answer on the Async & Promise page's intro.

Why create a new syntax?

The existing Reason syntax is owned by a different team with a different vision. Reason aims to be 100% compatible with OCaml syntax and to support all versions of OCaml. In the last few years, we've drawn the conclusion that itâ€™s very hard to deliver such goal without sacrificing user experience. The other reason is that we feel itâ€™s better to have the same vision as a team so that we can make more coherent decisions.

Who is behind the project?

The ReScript team (Hongbo, Cheng, Cristiano, Maxim, Patrick, Ricky).

We have a new forum; will we also have our own Discord?

Not now. We've found that too much important information get casually passed in Discord then lost within the noise. We prefer folks to communicate on the forum. This is nicer to the less active members.

The team doesn't use the old Discord anymore. We encourage you to move your questions to the forum instead.

SECTION try

---

title: "Try" description: "Try ReScript via Command Line" canonical: "/docs/manual/latest/try"

Try Online

Our Playground lets you try ReScript online, and comes with ReScript-React and the new ReScript-Core standard library preinstalled.

SECTION api

Overview

ReScript ships 3 modules in its standard library.

- Js: bindings for all your familiar JavaScript APIs.
- Belt: extra collections and helpers not available in JavaScript.
- Dom: Dom related types and modules.

Usage heuristics:

- Default to using the `Js` module. Most of the APIs in it are runtime-free and compile down to clean, readable JavaScript, which is our priority.
- For other APIs that aren't available in regular JavaScript (and thus don't exist in our `Js` bindings), use Belt. For example, prefer `Js.Array2` over `Belt.Array`.
- The `Dom` module contains our standardized types used by various userland DOM bindings. Due to the complexity of DOM, we don't mind that you ignore this module and build your application-specific DOM bindings.

Note: we do not recommend other userland standard library alternatives (unless it's DOM bindings). These cause confusion and split points for the community.

SECTION async-await

---

title: "Async / Await" description: "Async / await for asynchronous operations" canonical: "/docs/manual/latest/async-await"

```res prelude @val external fetchUserMail: string => promise = "GlobalAPI.fetchUserMail" @val external sendAnalytics: string => promise = "GlobalAPI.sendAnalytics"

```

<!-- See https://github.com/cristianoc/rescript-compiler-experiments/pull/1#issuecomment-1131182023 for all asyn
c/await use-case examples -->

Async / Await

ReScript comes with `async` / `await` support to make asynchronous, `Promise` based code easier to read and writ
e. This feature is very similar to its JS equivalent, so if you are already familiar with JS' `async` / `await`,
 you will feel right at home.

How it looks

Let's start with a quick example to show-case the syntax:

<CodeTab labels={["ReScript", "JS Output"]}>

```res
// Some fictive functionality that offers asynchronous network actions
@val external fetchUserMail: string => promise<string> = "GlobalAPI.fetchUserMail"
@val external sendAnalytics: string => promise<unit> = "GlobalAPI.sendAnalytics"

// We use the `async` keyword to allow the use of `await` in the function body
let logUserDetails = async (userId: string) => {
  // We use `await` to fetch the user email from our fictive user endpoint
  let email = await fetchUserMail(userId)

  await sendAnalytics(`User details have been logged for ${userId}`)
```

```
  Js.log(`Email address for user ${userId}: ${email}`)
}

async function logUserDetails(userId) {
  var email = await GlobalAPI.fetchUserMail(userId);
  await GlobalAPI.sendAnalytics("User details have been logged for " + userId + "");
  console.log("Email address for user " + userId + ": " + email + "");
}
```

As we can see above, an `async` function is defined via the `async` keyword right before the function's parameter list. In the function body, we are now able to use the `await` keyword to explicitly wait for a `Promise` value and assign its content to a let binding `email`. You will probably notice that this looks very similar to `async` / `await` in JS, but there are still a few details that are specific to ReScript. The next few sections will go through all the details that are specific to the ReScript type system. #### Basics - You may only use `await` in `async` function bodies - `await` may only be called on a `promise` value - `await` calls are expressions, therefore they can be used in pattern matching (`switch`) - A function returning a `promise<'a>` is equivalent to an `async` function returning a value ``'a` (important for writing signature files and bindings) - `promise` values and types returned from an `async` function don't auto-collapse into a "flat promise" like in JS (more on this later) #### Types and `async` functions ##### `async` function type signatures Function type signatures (i.e defined in signature files) don't require any special keywords for `async` usage. Whenever you want to type an `async` function, use a `promise` return type.

```
// Demo.resi

let fetchUserMail: string => promise<string>
```

The same logic applies to type definitions in `.res` files: ```res example // function type type someAsyncFn = int => promise // Function type annotation let fetchData: string => promise = async (userId) => { await fetchUserMail(userId) }

```
**BUT:** When typing `async` functions in your implementation files, you need to omit the `promise<'a>` type:

```res
// This function is compiled into a `string => promise<string>` type.
// The promise<...> part is implicitly added by the compiler.
let fetchData = async (userId: string): string => {
  await fetchUserMail("test")
}
```

For completeness reasons, let's expand the full signature and inline type definitions in one code snippet:

```
// Note how the inline return type uses `string`, while the type definition uses `promise<string>`
let fetchData: string => promise<string> = async (userId: string): string {
  await fetchUserMail(userId)
}
```

Note: In a practical scenario you'd either use a type signature, or inline types, not both at the same time. In case you are interested in the design decisions, check out [this discussion](https://github.com/rescript-lang/rescript-compiler/pull/5913#issuecomment-1359003870). ##### Promises don't auto-collapse in async functions In JS, nested promises (i.e. `promise>`) will automatically collapse into a flat promise (`promise<'a>`). This is not the case in ReScript. Use the `await` function to manually unwrap any nested promises within an `async` function instead.

```
let fetchData = async (userId: string): string => {
  // We can't just return the result of `fetchUserMail`, otherwise we'd get a
  // type error due to our function return type of type `string`
  await fetchUserMail(userId)
}
```

Error handling You may use `try / catch` or `switch` to handle exceptions during async execution.

```
// For simulation purposes
let authenticate = async () => {
  raise(Js.Exn.raiseRangeError("Authentication failed."))
}

let checkAuth = async () => {
  try {
    await authenticate()
  } catch {
  | Js.Exn.Error(e) =>
    switch Js.Exn.message(e) {
    | Some(msg) => Js.log("JS error thrown: " ++ msg)
    | None => Js.log("Some other exception has been thrown")
    }
  }
}
```

Note how we are essentially catching JS errors the same way as described in our [Exception](exception#catch-rescript-exceptions-from-js) section. You may unify error and value handling in a single switch as well:

```
let authenticate = async () => {
  raise(Js.Exn.raiseRangeError("Authentication failed."))
}

let checkAuth = async () => {
  switch await authenticate() {
  | _ => Js.log("ok")
```
```

```
  | exception Js.Exn.Error(e) =>
    switch Js.Exn.message(e) {
    | Some(msg) => Js.log("JS error thrown: " ++ msg)
    | None => Js.log("Some other exception has been thrown")
    }
  }
}
```

Important: When using `await` with a `switch`, always make sure to put the actual await call in the `switch` expression, otherwise your `await` error will not be caught. #### Piping `await` calls You may want to pipe the result of an `await` call right into another function. This can be done by wrapping your `await` calls in a new `{}` closure. ```res example @val external fetchUserMail: string => promise = "GlobalAPI.fetchUserMail" let fetchData = async () => { let mail = {await fetchUserMail("1234")}->Js.String2.toUpperCase Js.log(`All upper-cased mail: ${mail}`) }

```js
async function fetchData(param) {
  var mail = (await GlobalAPI.fetchUserMail("1234")).toUpperCase();
  console.log("All upper-cased mail: " + mail + "");
}
```

Note how the original closure was removed in the final JS output. No extra allocations! #### Pattern matching on `await` calls `await` calls are just another kind of expression, so you can use `switch` pattern matching for more complex logic. ```res example @val external fetchUserMail: string => promise = "GlobalAPI.fetchUserMail" let fetchData = async () => { switch (await fetchUserMail("user1"), await fetchUserMail("user2")) { | (user1Mail, user2Mail) => { Js.log("user 1 mail: " ++ user1Mail) Js.log("user 2 mail: " ++ user2Mail) } | exception JsError(err) => Js.log2("Some error occurred", err) } }

```js
async function fetchData(param) {
  var val;
  var val$1;
  try {
    val = await GlobalAPI.fetchUserMail("user1");
    val$1 = await GlobalAPI.fetchUserMail("user2");
  }
  catch (raw_err){
    var err = Caml_js_exceptions.internalToOCamlException(raw_err);
    if (err.RE_EXN_ID === "JsError") {
      console.log("Some error occurred", err._1);
      return ;
    }
    throw err;
  }
  console.log("user 1 mail: " + val);
  console.log("user 2 mail: " + val$1);
}
```

`await` multiple promises We can utilize the `Js.Promise2` module to handle multiple promises. E.g. let's use `Js.Promise2.all` to wait for multiple promises before continuing the program:

```
let pauseReturn = (value, timeout) => {
  Js.Promise2.make((~resolve, ~reject) => {
    Js.Global.setTimeout(() => {
      resolve(. value)
    }, timeout)->ignore
  })
}

let logMultipleValues = async () => {
  let promise1 = pauseReturn("value1", 2000)
  let promise2 = pauseReturn("value2", 1200)
  let promise3 = pauseReturn("value3", 500)

  let all = await Js.Promise2.all([promise1, promise2, promise3])

  switch all {
  | [v1, v2, v3] => Js.log(`All values: ${v1}, ${v2}, ${v3}`)
  | _ => Js.log("this should never happen")
  }
}
```

JS Interop with `async` functions `async` / `await` practically works with any function that returns a `promise<'a>` value. Map your `promise` returning function via an `external`, and use it in an `async` function as usual. Here's a full example of using the MDN `fetch` API, using `async` / `await` to simulate a login:

```
// A generic Response type for typing our fetch requests
module Response = {
  type t<'data>
  @send external json: t<'data> => promise<'data> = "json"
}

// A binding to our globally available `fetch` function. `fetch` is a
// standardized function to retrieve data from the network that is available in
// all modern browsers.
@val @scope("globalThis")
external fetch: (
```

```
  string,
  'params,
) => promise<Response.t<{"token": Js.Nullable.t<string>, "error": Js.Nullable.t<string>}>> =
  "fetch"

// We now use our asynchronous `fetch` function to simulate a login.
// Note how we use `await` with regular functions returning a `promise`.
let login = async (email: string, password: string) => {
  let body = {
    "email": email,
    "password": password,
  }

  let params = {
    "method": "POST",
    "headers": {
      "Content-Type": "application/json",
    },
    "body": Js.Json.stringifyAny(body),
  }

  try {
    let response = await fetch("https://reqres.in/api/login", params)
    let data = await response->Response.json

    switch Js.Nullable.toOption(data["error"]) {
    | Some(msg) => Error(msg)
    | None =>
      switch Js.Nullable.toOption(data["token"]) {
      | Some(token) => Ok(token)
      | None => Error("Didn't return a token")
      }
    }
  } catch {
  | _ => Error("Unexpected network error occurred")
  }
}
```

SECTION bind-to-js-object

---

title: "Bind to JS Object" description: "Interop with JS objects in ReScript" canonical: "/docs/manual/latest/bind-to-js-object"

Bind to JS Object

JavaScript objects are a combination of several use-cases:

- As a "record" or "struct" in other languages (like ReScript and C).
- As a hash map.
- As a class.
- As a module to import/export.

ReScript cleanly separates the binding methods for JS object based on these 4 use-cases. This page documents the first three. Binding to JS module objects is described in the Import from/Export to JS section.

Bind to Record-like JS Objects

Bind Using ReScript Record

If your JavaScript object has fixed fields, then it's conceptually like a ReScript record. Since a ReScript record compiles to a clean JavaScript object, you can definitely type a JS object as a ReScript record!

```res example type person = { name: string, friends: array, age: int, }

@module("MySchool") external john: person = "john"

let johnName = john.name

```js
var MySchool = require("MySchool");

var johnName = MySchool.john.name;
```

External is documented here. `@module` is documented here.

If you want or need to use different field names on the ReScript and the JavaScript side, you can use the `@as` decorator:

```res example type action = { @as("type") type_: string }

let action = {type_: "ADD_USER"}

```js
var action = {
  type: "ADD_USER"
};
```

This is useful to map to JavaScript attribute names that cannot be expressed in ReScript (such as keywords).

It is also possible to map a ReScript record to a JavaScript array by passing indices to the `@as` decorator:

```
type t = {
  @as("0") foo: int,
  @as("1") bar: string,
}

let value = {foo: 7, bar: "baz"}
```

```
var value = [
  7,
  "baz"
];
```

Bind Using ReScript Object

Alternatively, you can use ReScript object to model a JS object too:

```res example type person = { "name": string, "friends": array, "age": int, }

@module("MySchool") external john: person = "john"

let johnName = john["name"]
```

```js
var MySchool = require("MySchool");

var johnName = MySchool.john.name;
```

Bind Using Special Getter and Setter Attributes

Alternatively, you can use `get` and `set` to bind to individual fields of a JS object:

```res example type textarea @set external setName: (textarea, string) => unit = "name" @get external getName: textarea => string = "name"

```js
```

You can also use `get_index` and `set_index` to access a dynamic property or an index:

```res example type t @new external create: int => t = "Int32Array" @get_index external get: (t, int) => int = "" @set_index external set: (t, int, int) => unit = ""

let i32arr = create(3) i32arr->set(0, 42) Js.log(i32arr->get(0))
```

```js
var i32arr = new Int32Array(3);
i32arr[0] = 42;
console.log(i32arr[0]);
```

Bind to Hash Map-like JS Object

If your JavaScript object:

- might or might not add/remove keys
- contains only values that are of the same type

Then it's not really an object, it's a hash map. Use Js.Dict, which contains operations like `get`, `set`, etc. and cleanly compiles to a JavaScript object still.

Bind to a JS Object That's a Class

Use `new` to emulate e.g. `new Date()`:

```res example type t @new external createDate: unit => t = "Date"

let date = createDate()
```
```

```js
var date = new Date();
```

You can chain `new` and `module` if the JS module you're importing is itself a class:

```res example type t @new @module external book: unit => t = "Book" let myBook = book()
```

```js
var Book = require("Book");
var myBook = new Book();
```

SECTION extensible-variant

---

title: "Extensible Variant" description: "Extensible Variants in ReScript" canonical: "/docs/manual/latest/extensible-variant"

Extensible Variant

Variant types are usually constrained to a fixed set of constructors. There may be very rare cases where you still want to be able to add constructors to a variant type even after its initial type declaration. For this, we offer extensible variant types.

Definition and Usage

```res example type t = ..

type t += Other

type t += | Point(float, float) | Line(float, float, float, float)
```

```js
var Caml_exceptions = require("./stdlib/caml_exceptions.js");

var Other = Caml_exceptions.create("Playground.Other");

var Point = Caml_exceptions.create("Playground.Point");

var Line = Caml_exceptions.create("Playground.Line");
```

The `..` in the type declaration above defines an extensible variant `type t`. The `+=` operator is then used to add constructors to the given type.

Note: Don't forget the leading `type` keyword when using the `+=` operator!

Pattern Matching Caveats

Extensible variants are open-ended, so the compiler will not be able to exhaustively pattern match all available cases. You will always need to provide a default _ case for every `switch` expression.

```
let print = v =>
  switch v {
  | Point(x, y) => Js.log2("Point", (x, y))
  | Line(ax, ay, bx, by) => Js.log2("Line", (ax, ay, bx, by))
  | Other
  | _ => Js.log("Other")
  }
```

```
function print(v) {
  if (v.RE_EXN_ID === Point) {
    console.log("Point", [v._1, v._2]);
  } else if (v.RE_EXN_ID === Line) {
    console.log("Line", [v._1, v._2, v._3, v._4]);
  } else {
    console.log("Other");
  }
}
```

Tips & Tricks

Fun fact: In ReScript, exceptions are actually extensible variants under the hood, so `exception UserError(string)` is equivalent to `type exn += UserError(string)`. It's one of the very few use-case where extensible variants make sense.

We usually recommend sticking with common variants as much as possible to reap the benefits of exhaustive pattern matching.

SECTION interop-cheatsheet

---

title: "Interop Cheatsheet" description: "Cheatsheet for various interop scenarios in ReScript" canonical: "/docs/manual/latest/interop-cheatsheet"

Interop Cheatsheet

This is a glossary with examples. All the features are described by later pages.

List of Decorators

> **Note:** In ReScript < 8.3, all our attributes started with the `bs.` prefix. This is no longer needed and our formatter automatically removes them in newer ReScript versions.

Attributes

- `@as`: here, here, here and here
- `@deriving`
- `@get`
- `@get_index`

- `@inline`
- `@int`

- `@module`
- `@new`
- `@obj`
- `@optional`
- `@return`
- `@send`: here and here
- `@scope`
- `@set`
- `@set_index`
- `@variadic`
- `@string`
- `@this`
- `@uncurry`
- `@unwrap`

- `@val`

- `@deprecated`

- `genType`
- `@JSX`
- `@react.component`: here and here
- `@warning`
- `@unboxed`

Extension Points

- `%debugger`
- `%external`

- `%raw`
- `%re`

Raw JS

```res example let add = %raw("(a, b) => a + b") %%raw("const a = 1")

```js
var add = ((a, b) => a + b);
const a = 1
```

Global Value

```res example @val external setTimeout: (unit => unit, int) => float = "setTimeout"

```js
// Empty output
```

Global Module's Value

```res example @val @scope("Math") external random: unit => float = "random"

let someNumber = random()

@val @scope(("window", "location", "ancestorOrigins")) external length: int = "length"
```

```js
var someNumber = Math.random();
```

Nullable

```res example let a = Some(5) // compiles to 5 let b = None // compiles to undefined
```

```js
var a = 5;
var b;
```

Handling a value that can be `undefined` and `null`, by ditching the `option` type and using `Js.Nullable.t`:

```res example let jsNull = Js.Nullable.null let jsUndefined = Js.Nullable.undefined let result1: Js.Nullable.t = Js.Nullable.return("hello") let result2: Js.Nullable.t = Js.Nullable.fromOption(Some(10)) let result3: option = Js.Nullable.toOption(Js.Nullable.return(10))
```

```js
var Caml_option = require("./stdlib/caml_option.js");
var Js_null_undefined = require("./stdlib/js_null_undefined.js");

var jsNull = null;
var jsUndefined;
var result1 = "hello";
var result2 = Js_null_undefined.fromOption(10);
var result3 = Caml_option.nullable_to_opt(10);
```

JS Object

- Bind to a JS object as a ReScript record.
- Bind to a JS object that acts like a hash map.
- Bind to a JS object that's a class.

Function

Object Method & Chaining

```res example @send external map: (array<'a>, 'a => 'b) => array<'b> = "map" @send external filter: (array<'a>, 'a => 'b) => array<'b> = "filter" [1, 2, 3] ->map(a => a + 1) ->filter(a => mod(a, 2) == 0) ->Js.log
```

```js
console.log(
  [1, 2, 3]
    .map(function (a) {
      return (a + 1) | 0;
    })
    .filter(function (a) {
      return a % 2 === 0;
    })
);
```

Variadic Arguments

```res example @module("path") @variadic external join: array => string = "join"
```

```js
// Empty output
```

Polymorphic Function

```res example @module("Drawing") external drawCat: unit => unit = "draw" @module("Drawing") external drawDog: (~giveName: string) => unit = "draw"
```

```js
// Empty output
```

```res example @val external padLeft: ( string, @unwrap [ | #Str(string) | #Int(int) ]) => string = "padLeft"

padLeft("Hello World", #Int(4)) padLeft("Hello World", #Str("Message from ReScript: "))
```

```js
padLeft("Hello World", 4);
```

```
padLeft("Hello World", "Message from ReScript: ");
```

JS Module Interop

See here

Dangerous Type Cast

Final escape hatch converter. Do not abuse.

```res example external convertToFloat: int => float = "%identity" let age = 10 let gpa = 2.1 +. convertToFloat(age)
```

```js
var age = 10;
var gpa = 2.1 + 10;
```

SECTION array-and-list

---

title: "Array & List" description: "Arrays and List data structures" canonical: "/docs/manual/latest/array-and-list"

Array and List

Array

Arrays are our main ordered data structure. They work the same way as JavaScript arrays: they can be randomly accessed, dynamically resized, updated, etc.

```res example let myArray = ["hello", "world", "how are you"]
```

```js
var myArray = ["hello", "world", "how are you"];
```

ReScript arrays' items must have the same type, i.e. homogeneous.

Usage

See the Js.Array API.

Access & update an array item like so:

```res example let myArray = ["hello", "world", "how are you"]

let firstItem = myArray[0] // "hello"

myArray[0] = "hey" // now ["hey", "world", "how are you"]

let pushedValue = Js.Array2.push(myArray, "bye")
```

```js
var myArray = ["hello", "world", "how are you"];

var firstItem = myArray[0];

myArray[0] = "hey";

var pushedValue = myArray.push("bye");
```

List

ReScript provides a singly linked list too. Lists are:

- immutable
- fast at prepending items
- fast at getting the head
- slow at everything else

```res example let myList = list{1, 2, 3}
```

```js
var myList = {
  hd: 1,
  tl: {
```

```
    hd: 2,
    tl: {
      hd: 3,
      tl: 0
    }
  }
};
```

Like arrays, lists' items need to be of the same type.

Usage

You'd use list for its resizability, its fast prepend (adding at the head), and its fast split, all of which are immutable and relatively efficient.

Do **not** use list if you need to randomly access an item or insert at non-head position. Your code would end up obtuse and/or slow.

The standard lib provides a List module.

Immutable Prepend

Use the spread syntax:

```res prelude let myList = list{1, 2, 3} let anotherList = list{0, ...myList}

```js
var myList = {
  hd: 1,
  tl: {
    hd: 2,
    tl: {
      hd: 3,
      tl: 0
    }
  }
};

var anotherList = {
  hd: 0,
  tl: myList
};
```

`myList` didn't mutate. `anotherList` is now `list{0, 1, 2, 3}`. This is efficient (constant time, not linear). `anotherList`'s last 3 elements are shared with `myList`!

Note that `list{a, ...b, ...c}` was a syntax error before compiler v10.1. In general, the pattern should be used with care as its performance and allocation overhead are linear (`O(n)`).

Access

`switch` (described in the pattern matching section) is usually used to access list items:

```res example let message = switch myList { | list{} => "This list is empty" | list{a, ...rest} => "The head of the list is the string " ++ Js.Int.toString(a) }

```js
var message = myList
  ? "The head of the list is the string " + (1).toString()
  : "This list is empty";
```

SECTION interop-with-js-build-systems

---

title: "Interop with JS Build Systems" description: "Documentation on how to interact with existing JS build systems" canonical: "/docs/manual/latest/interop-with-js-build-systems"

Interop with JS Build Systems

If you come from JS, chances are that you already have a build system in your existing project. Here's an overview of the role `rescript` would play in your build pipeline, if you want to introduce some ReScript code.

> **Please** try not to wrap `rescript` into your own incremental build framework. ReScript's compilation is very hard to get right, and you'll inevitably run into stale or badly performing builds (therefore erasing much of our value proposition) if you create your own meta layer on top.

Popular JS Build Systems

The JS ecosystem uses a few build systems: browserify, rollup, webpack, etc. The latter's probably the most popular of the three (as of 2019 =P). These build systems do both the compilation and the linking (aka, bundling many files into one or few files).

`rescript` only take care of the compilation step; it maps one `.res`/`.resi` file into one JS output file. As such, in theory, no build system integration is needed from our side. From e.g. the webpack watcher's perspective, the JS files ReScript generates are almost equivalent to your hand-written JS files. We also recommend **that you initially check in those ReScript-generated JS files**, as this workflow means:

- You can introduce ReScript silently into your codebase without disturbing existing infra.
- You have a **visual** diff of the performance & correctness of your JS file when you update the `.res` files and the JS artifacts change.
- You can let teammates hot-patch the JS files in emergency situations, without needing to first start learning ReScript.
- You can remove ReScript completely from your codebase and things will still work (in case your company decides to stop using us for whatever reason).

For what it's worth, you can also turn `rescript` into an automated step in your build pipeline, e.g. into a Webpack loader; but such approach is error-prone and therefore discouraged.

Tips & Tricks

You can make ReScript JS files look even more idiomatic through the in-source + bs suffix config in `rescript.json`:

```
{
  "package-specs": {
    "module": "commonjs", // or whatever module system your project uses
    "in-source": true
  },
  "suffix": ".bs.js"
}
```

This will:

- Generate the JS files alongside your ReScript source files.
- Use the file extension `.bs.js`, so that you can require these files on the JS side through `require('./MyFile.bs')`, without needing a loader.

Use Loaders on ReScript Side

"What if my build system uses a CSS/png/whatever loader and I'd like to use it in ReScript?"

Loaders are indeed troublesome; in the meantime, please use e.g. `%raw("require('./myStyles.css')")` at the top of your file. This just uses `raw` to compile the snippet into an actual JS require.

Getting Project's Dependencies

`rescript` generates one `MyFile.d` file per `MyFile` source file; you'll find them in `lib/bs`. These are human readable, machine-friendly list of the dependencies of said `MyFile`. You can read into them for your purpose (though mind the IO overhead). Use these files instead of creating your own dependency graph; we did the hard work of tracking the dependencies as best as possible (including inner modules, `open`s, module names overlap, etc).

Run Script Per File Built

See js-post-build. Though please use it sparingly; if you hook up a node.js script after each file built, you'll incur the node startup time per file!

SECTION use-illegal-identifier-names

---

title: "Use Illegal Identifier Names" description: "Handling (JS) naming collisions in ReScript" canonical: "/docs/manual/latest/use-illegal-identifier-names"

Use Illegal Identifier Names

Sometime, for e.g. a let binding or a record field, you might want to use: - A capitalized name. - A name that contains illegal characters (e.g. emojis, hyphen, space). - A name that's one of ReScript's reserved keywords.

We provide an escape hatch syntax for these cases:

```res example let \"my-ðŸŽ" = 10

type element = { \"aria-label": string }
```

let myElement = { \"aria-label": "close" }

let label = myElement.\"aria-label"

let calculate = (~\"Props") => { \"Props" + 1 }

```
```js
var my$$unknown$unknown$unknown$unknown = 10;

var myElement = {
  "aria-label": "close"
};

var label = myElement["aria-label"];

function calculate(Props) {
  return Props + 1 | 0;
}
```

See the output. **Use them only when necessary**, for interop with JavaScript. This is a last-resort feature. If you abuse this, many of the compiler guarantees will go away.

SECTION warning-numbers

---

title: "Warning Numbers" description: "Available compiler warning numbers in ReScript" canonical: "/docs/manual/latest/warning-numbers"

---

import { make as WarningTable } from "src/components/WarningTable.mjs";

Warning Numbers

You can configure which warnings the ReScript compiler generates in the build configuration or using the `@warning()` or the `@@warning()` decorator.

SECTION import-export

---

title: "Import & Export" description: "Importing / exporting in ReScript modules" canonical: "/docs/manual/latest/import-export"

Import & Export

Import a Module/File

Unlike JavaScript, ReScript doesn't have or need import statements:

```
// Inside School.res
let studentMessage = Student.message
```

```
var Student = require("./Student.bs");
var studentMessage = Student.message
```

The above code refers to the `message` binding in the file `Student.res`. Every ReScript file is also a module, so accessing another file's content is the same as accessing another module's content!

A ReScript project's file names need to be unique.

Export Stuff

By default, every file's type declaration, binding and module is exported, aka publicly usable by another file. **This also means those values, once compiled into JS, are immediately usable by your JS code**.

To only export a few selected things, use a `.resi` interface file.

Work with JavaScript Import & Export

To see how to import JS modules and export stuff for JS consumption, see the JavaScript Interop section's Import from/Export to JS.

SECTION module

```

title: "Module" description: "ReScript modules, module signatures and interface files" canonical: "/docs/manual/latest/module"

Module

Basics

Modules are like mini files! They can contain type definitions, `let` bindings, nested modules, etc.

Creation

To create a module, use the `module` keyword. The module name must start with a **capital letter**. Whatever you could place in a `.res` file, you may place inside a module definition's `{}` block.

```res example module School = { type profession = Teacher | Director

let person1 = Teacher let getProfession = (person) => switch person { | Teacher => "A teacher" | Director => "A director" } }
```

```js
function getProfession(person) {
  if (person) {
    return "A director";
  } else {
    return "A teacher";
  }
}

var School = {
  person1: /* Teacher */0,
  getProfession: getProfession
};
```

A module's contents (including types!) can be accessed much like a record's, using the `.` notation. This demonstrates modules' utility for namespacing.

```
let anotherPerson: School.profession = School.Teacher
Js.log(School.getProfession(anotherPerson)) /* "A teacher" */
```

```
var anotherPerson = /* Teacher */0;
console.log("A teacher");
```

Nested modules work too.

```res example module MyModule = { module NestedModule = { let message = "hello" } }

let message = MyModule.NestedModule.message
```

```js
var NestedModule = {
  message: message
};

var MyModule = {
  NestedModule: NestedModule
};

var message = MyModule.NestedModule.message;
```

opening a module

Constantly referring to a value/type in a module can be tedious. Instead, we can "open" a module and refer to its contents without always prepending them with the module's name. Instead of writing:

```
let p = School.getProfession(School.person1)
```

```
var p = School.getProfession(School.person1);
```

We can write:

```
open School
let p = getProfession(person1)
```

```
var p = School.getProfession(School.person1);
```

The content of `School` module are made visible (**not** copied into the file, but simply made visible!) in scope. `profession`, `getProfession` and `person1` will thus correctly be found.

Use `open` this sparingly, it's convenient, but makes it hard to know where some values come from. You should usually use `open` in a local scope:

```
let p = {
  open School
  getProfession(person1)
}
/* School's content isn't visible here anymore */

var p = School.getProfession(School.person1);
```

Use `open!` to ignore shadow warnings

There are situations where `open` will cause a warning due to existing identifiers (bindings, types) being redefined. Use `open!` to explicitly tell the compiler that this is desired behavior.

```
let map = (arr, value) => {
  value
}

// opening Js.Array2 would shadow our previously defined `map`
// `open!` will explicitly turn off the automatic warning
open! Js.Array2
let arr = map([1,2,3], (a) => { a + 1})
```

Note: Same as with `open`, don't overuse `open!` statements if not necessary. Use (sub)modules to prevent shadowing issues.

Destructuring modules

Since 9.0.2

As an alternative to `open`ing a module, you can also destructure a module's functions and values into separate let bindings (similarly on how we'd destructure an object in JavaScript).

```
module User = {
  let user1 = "Anna"
  let user2 = "Franz"
}

// Destructure by name
let {user1, user2} = module(User)

// Destructure with different alias
let {user1: anna, user2: franz} = module(User)

var user1 = "Anna";

var user2 = "Franz";

var User = {
  user1: user1,
  user2: user2
};
```

Note: You can't extract types with module destructuring â€” use a type alias instead (`type user = User.myUserType`).

Extending modules

Using `include` in a module statically "spreads" a module's content into a new one, thus often fulfill the role of "inheritance" or "mixin".

Note: this is equivalent to a compiler-level copy paste. **We heavily discourage `include`**. Use it as last resort!

```res example module BaseComponent = { let defaultGreeting = "Hello" let getAudience = (~excited) => excited ? "world!" : "world" }

module ActualComponent = { / *the content is copied over* / include BaseComponent / *overrides BaseComponent.defaultGreeting* / let defaultGreeting = "Hey" let render = () => defaultGreeting ++ " " ++ getAudience(~excited=true) }

```js
function getAudience(excited) {
  if (excited) {
    return "world!";
  } else {
    return "world";
  }
}

var BaseComponent = {
  defaultGreeting: "Hello",
  getAudience: getAudience
};

var defaultGreeting = "Hey";

function render(param) {
```
```

```
  return "Hey world!";
}

var ActualComponent = {
  getAudience: getAudience,
  defaultGreeting: defaultGreeting,
  render: render
};
```

Note: `open` and `include` are very different! The former brings a module's content into your current scope, so that you don't have to refer to a value by prefixing it with the module's name every time. The latter **copies over** the definition of a module statically, then also do an `open`.

Every `.res` file is a module

Every ReScript file is itself compiled to a module of the same name as the file name, capitalized. The file `React.res` implicitly forms a module `React`, which can be seen by other source files.

Note: ReScript file names should, by convention, be capitalized so that their casing matches their module name. Uncapitalized file names are not invalid, but will be implicitly transformed into a capitalized module name. I.e. `file.res` will be compiled into the module `File`. To simplify and minimize the disconnect here, the convention is therefore to capitalize file names.

Signatures

A module's type is called a "signature", and can be written explicitly. If a module is like a `.res` (implementation) file, then a module's signature is like a `.resi` (interface) file.

Creation

To create a signature, use the `module type` keyword. The signature name must start with a **capital letter**. Whatever you could place in a `.resi` file, you may place inside a signature definition's `{}` block.

```res example / *Picking up previous section's example* / module type EstablishmentType = { type profession let getProfession: profession => string }
```

```js
// Empty output
```

A signature defines the list of requirements that a module must satisfy in order for that module to match the signature. Those requirements are of the form:

- `let x: int` requires a `let` binding named `x`, of type `int`.
- `type t = someType` requires a type field `t` to be equal to `someType`.
- `type t` requires a type field `t`, but without imposing any requirements on the actual, concrete type of `t`. We'd use `t` in other entries in the signature to describe relationships, e.g. `let makePair: t => (t, t)` but we cannot, for example, assume that `t` is an `int`. This gives us great, enforced abstraction abilities.

To illustrate the various kinds of type entries, consider the above signature `EstablishmentType` which requires that a module:

- Declare a type named `profession`.
- Must include a function that takes in a value of the type `profession` and returns a string.

Note:

Modules of the type `EstablishmentType` can contain more fields than the signature declares, just like the module `School` in the previous section (if we choose to assign it the type `EstablishmentType`. Otherwise, `School` exposes every field). This effectively makes the `person1` field an enforced implementation detail! Outsiders can't access it, since it's not present in the signature; the signature **constrained** what others can access.

The type `EstablishmentType.profession` is **abstract**: it doesn't have a concrete type; it's saying "I don't care what the actual type is, but it's used as input to `getProfession`". This is useful to fit many modules under the same interface:

```
module Company: EstablishmentType = {
  type profession = CEO | Designer | Engineer | ...

  let getProfession = (person) => ...
  let person1 = ...
  let person2 = ...
}

function getProfession(person) {
  ...
}

var person1 = ...

var person2 = ...
```

```
var Company = {
  getProfession: getProfession,
  person1: person1,
  person2: person2
};
```

It's also useful to hide the underlying type as an implementation detail others can't rely on. If you ask what the type of `Company.profession` is, instead of exposing the variant, it'll only tell you "it's `Company.profession`".

Extending module signatures

Like modules themselves, module signatures can also be extended by other module signatures using `include`. Again, **heavily discouraged**:

```res example module type BaseComponent = { let defaultGreeting: string let getAudience: (~excited: bool) => string }

module type ActualComponent = { /* *the BaseComponent signature is copied over* */ include BaseComponent let render: unit => string }
```

```js
// Empty output
```

Note: `BaseComponent` is a module **type**, not an actual module itself!

If you do not have a defined module type, you can extract it from an actual module using `include (module type of ActualModuleName)`. For example, we can extend the `List` module from the standard library, which does not define a module type.

```res example module type MyList = { include (module type of List) let myListFun: list<'a> => list<'a> }
```

```js
// Empty output
```

Every `.resi` file is a signature

Similar to how a `React.res` file implicitly defines a module `React`, a file `React.resi` implicitly defines a signature for `React`. If `React.resi` isn't provided, the signature of `React.res` defaults to exposing all the fields of the module. Because they don't contain implementation files, `.resi` files are used in the ecosystem to also document the public API of their corresponding modules.

```res example /* *file React.res (implementation. Compiles to module React)* */ type state = int let render = (str) => str
```

```js
function render(str) {
  return str;
}
```

```res sig / *file React.resi (interface. Compiles to the signature of React.res)* / type state = int let render: string => string
```

```
#### Module Functions (functors)

Modules can be passed to functions! It would be the equivalent of passing a file
as a first-class item. However, modules are at a different "layer" of the
language than other common concepts, so we can't pass them to *regular*
functions. Instead, we pass them to special functions called "functors".

The syntax for defining and using functors is very much like the syntax
for defining and using regular functions. The primary differences are:

- Functors use the `module` keyword instead of `let`.
- Functors take modules as arguments and return a module.
- Functors *require* annotating arguments.
- Functors must start with a capital letter (just like modules/signatures).

Here's an example `MakeSet` functor, that takes in a module of the type
`Comparable` and returns a new set that can contain such comparable items.

<CodeTab labels={["ReScript", "JS Output"]}>

```res prelude
module type Comparable = {
  type t
  let equal: (t, t) => bool
}

module MakeSet = (Item: Comparable) => {
  // let's use a list as our naive backing data structure
  type backingType = list<Item.t>
  let empty = list{}
  let add = (currentSet: backingType, newItem: Item.t): backingType =>
    // if item exists
    if currentSet->List.some(x => Item.equal(x, newItem)) {
      currentSet // return the same (immutable) set (a list really)
```
```

```
    } else {
      list{
        newItem,
        ...currentSet // prepend to the set and return it
      }
    }
}

var List = require("./stdlib/list.js");

function MakeSet(Item) {
  var add = function(currentSet, newItem) {
    if (
      List.exists(function(x) {
        return Item.equal(x, newItem);
      }, currentSet)
    ) {
      return currentSet;
    } else {
      return {
        hd: newItem,
        tl: currentSet,
      };
    }
  };
  return {
    empty: /* [] */ 0,
    add: add,
  };
}
```

Functors can be applied using function application syntax. In this case, we're creating a set, whose items are pairs of integers.

```res example module IntPair = { type t = (int, int) let equal = ((x1: int, y1: int), (x2, y2)) => x1 == x2 && y1 == y2 let create = (x, y) => (x, y) }

/ *IntPair abides by the Comparable signature required by MakeSet* / module SetOfIntPairs = MakeSet(IntPair)
```

```js
function equal(param, param$1) {
  if (param[0] === param$1[0]) {
    return param[1] === param$1[1];
  } else {
    return false;
  }
}

function create(x, y) {
  return [x, y];
}

var IntPair = {
  equal: equal,
  create: create,
};

var SetOfIntPairs = {
  empty: /* [] */ 0,
  add: add,
};
```

Module functions types

Like with module types, functor types also act to constrain and hide what we may assume about functors. The syntax for functor types are consistent with those for function types, but with types capitalized to represent the signatures of modules the functor accepts as arguments and return values. In the previous example, we're exposing the backing type of a set; by giving `MakeSet` a functor signature, we can hide the underlying data structure!

```
module type Comparable = ...

module type MakeSetType = (Item: Comparable) => {
  type backingType
  let empty: backingType
  let add: (backingType, Item.t) => backingType
}

module MakeSet: MakeSetType = (Item: Comparable) => {
  ...
}

// Empty output
```

Exotic Module Filenames

Since 8.3

It is possible to use non-conventional characters in your filenames (which is sometimes needed for specific JS frameworks). Here are some examples:

- `src/Button.ios.res`
- `pages/[id].res`

Please note that modules with an exotic filename will not be accessible from other ReScript modules.

Tips & Tricks

Modules and functors are at a different "layer" of language than the rest (functions, let bindings, data structures, etc.). For example, you can't easily pass them into a tuple or record. Use them judiciously, if ever! Lots of times, just a record or a function is enough.

SECTION null-undefined-option

---

title: "Null, Undefined and Option" description: "JS interop with nullable and optional values in ReScript" canonical: "/docs/manual/latest/null-undefined-option"

Null, Undefined and Option

ReScript itself doesn't have the notion of `null` or `undefined`. This is a *great* thing, as it wipes out an entire category of bugs. No more `undefined is not a function`, and `cannot access someAttribute of undefined`!

However, the **concept** of a potentially nonexistent value is still useful, and safely exists in our language.

We represent the existence and nonexistence of a value by wrapping it with the `option` type. Here's its definition from the standard library:

```res example type option<'a> = None | Some('a)

```js
// Empty output
```

It means "a value of type option is either None (representing nothing) or that actual value wrapped in a Some".

Note how the `option` type is just a regular variant.

Example

Here's a normal value:

```res example let licenseNumber = 5

```js
var licenseNumber = 5;
```

To represent the concept of "maybe null", you'd turn this into an `option` type by wrapping it. For the sake of a more illustrative example, we'll put a condition around it:

```
let licenseNumber =
  if personHasACar {
    Some(5)
  } else {
    None
  }
```

```
var licenseNumber = personHasACar ? 5 : undefined;
```

Later on, when another piece of code receives such value, it'd be forced to handle both cases through pattern matching:

```
switch licenseNumber {
| None =>
  Js.log("The person doesn't have a car")
| Some(number) =>
  Js.log("The person's license number is " ++ Js.Int.toString(number))
}
```

```
var number = licenseNumber;

if (number !== undefined) {
  console.log("The person's license number is " + number.toString());
} else {
  console.log("The person doesn't have a car");
```

```
}
```

By turning your ordinary number into an `option` type, and by forcing you to handle the `None` case, the language effectively removed the possibility for you to mishandle, or forget to handle, a conceptual `null` value! **A pure ReScript program doesn't have null errors**.

Interoperate with JavaScript `undefined` and `null`

The `option` type is common enough that we special-case it when compiling to JavaScript:

```res example let x = Some(5)
```

```js
var x = 5;
```

simply compiles down to `5`, and

```res example let x = None
```

```js
var x;
```

compiles to `undefined`! If you've got e.g. a string in JavaScript that you know might be `undefined`, type it as `option<string>` and you're done! Likewise, you can send a `Some(5)` or `None` to the JS side and expect it to be interpreted correctly =)

Caveat 1

The option-to-undefined translation isn't perfect, because on our side, `option` values can be composed:

```res example let x = Some(Some(Some(5)))
```

```js
var x = 5;
```

This still compiles to `5`, but this gets troublesome:

```res example let x = Some(None)
```

```js
var Caml_option = require("./stdlib/caml_option.js");

var x = Caml_option.some(undefined);
```

(See output tab).

What's this `Caml_option.some` thing? Why can't this compile to `undefined`? Long story short, when dealing with a polymorphic `option` type (aka `option<'a>`, for any `'a`), many operations become tricky if we don't mark the value with some special annotation. If this doesn't make sense, don't worry; just remember the following rule:

- **Never, EVER, pass a nested `option` value (e.g. `Some(Some(Some(5)))`) into the JS side.**
- **Never, EVER, annotate a value coming from JS as `option<'a>`. Always give the concrete, non-polymorphic type.**

Caveat 2

Unfortunately, lots of times, your JavaScript value might be *both* `null` or `undefined`. In that case, you unfortunately can't type such value as e.g. `option<int>`, since our `option` type only checks for `undefined` and not `null` when dealing with a `None`.

Solution: More Sophisticated `undefined` & `null` Interop

To solve this, we provide access to more elaborate `null` and `undefined` helpers through the `Js.Nullable` module. This somewhat works like an `option` type, but is different from it.

Examples

To create a JS `null`, use the value `Js.Nullable.null`. To create a JS `undefined`, use `Js.Nullable.undefined` (you can naturally use `None` too, but that's not the point here; the `Js.Nullable.*` helpers wouldn't work with it).

If you're receiving, for example, a JS string that can be `null` and `undefined`, type it as:

```res example @module("MyConstant") external myId: Js.Nullable.t = "myId"
```

```js
// Empty output
```

To create such a nullable string from our side (presumably to pass it to the JS side, for interop purpose), do:

```res example @module("MyIdValidator") external validate: Js.Nullable.t => bool = "validate" let personId: Js.Nullable.t = Js.Nullable.return("abc123")

let result = validate(personId)

```js
var MyIdValidator = require("MyIdValidator");
var personId = "abc123";
var result = MyIdValidator.validate(personId);
```

The `return` part "wraps" a string into a nullable string, to make the type system understand and track the fact that, as you pass this value around, it's not just a string, but a string that can be `null` or `undefined`.

Convert to/from `option`

`Js.Nullable.fromOption` converts from a `option` to `Js.Nullable.t`. `Js.Nullable.toOption` does the opposite.

SECTION build-performance

---

title: "Performance" metaTitle: "Build Performance" description: "ReScript build performance and measuring tools" canonical: "/docs/manual/latest/build-performance"

Build Performance

ReScript considers performance at install time, build time and run time as a serious feature; it's one of those things you don't notice until you realize it's missing.

Profile Your Build

Sometime your build can be slow due to some confused infra setups. We provide an interactive visualization of your build's performance via `bstracing`:

```
./node_modules/.bin/bstracing
```

Run the above command at your ReScript project's root; it'll spit out a JSON file you can drag and drop into `chrome://tracing`.

import Image from "src/components/Image";

Under the Hood

ReScript itself uses a build system under the hood, called Ninja. Ninja is like Make, but cross-platform, minimal, focuses in perf and destined to be more of a low-level building block than a full-blown build system. In this regard, Ninja's a great implementation detail for `rescript`.

ReScript reads into `rescript.json` and generates the Ninja build file in `lib/bs`. The file contains the low-level compiler commands, namespacing rules, intermediate artifacts generation & others. It then runs `ninja` for the actual build.

The JS Wrapper

`rescript` itself is a Node.js wrapper which takes care of some miscellaneous tasks, plus the watcher. The lower-level, watcher-less, fast native `rescript` is called `rescript.exe`. It's located at `node_modules/rescript/{your-platform}/rescript.exe`.

If you don't need the watcher, you can run said `rescript.exe`. This side-steps Node.js' long startup time, which can be in the order of `100ms`. Our editor plugin finds and uses this native `rescript.exe` for better performance.

Numbers

Raw `rescript.exe` build on a small project should be around `70ms`. This doubles when you use the JS `rescript` wrapper which comes with a watcher, which is practically faster since you don't manually run the build at every change (though you should opt for the raw `rescript.exe` for programmatic usage, e.g. inserting rescript into your existing JS build pipeline).

No-op build (when no file's changed) should be around `15ms`. Incremental rebuild (described soon) of a single file in a project is around `70ms` too.

Cleaning the artifacts should be instantaneous.

Extreme Test
```

We've stress-tested `rescript.exe` on a big project of 10,000 files (2 directories, 5000 files each, first 5000 no dependencies, last 5000 10 dependencies on files from the former directory) using https://github.com/rescript-lang/build-benchmark, on a Retina Macbook Pro Early 2015 (3.1 GHz Intel Core i7).

- No-op build of 10k files: `800ms` (the minimum amount of time required to check the mtimes of 10k files).
- Clean build: \<3 minutes.
- Incremental build: depends on the number of the dependents of the file. No dependent means `1s`.

Stability

`rescript` is a file-based build system. We don't do in-memory build, even if that speeds up the build a lot. In-memory builds risk memory leaks, out-of-memory errors, corrupt halfway build and others. Our watcher mode stays open for days or months with no leak.

The watcher is also just a thin file watcher that calls `rescript.exe`. We don't like babysitting daemon processes.

Incrementality & Correctness

ReScript doesn't take whole seconds to run every time. The bulk of the build performance comes from incremental build, aka re-building a previously built project when a few files changed.

In short, thanks to our compiler and the build system's architecture, we're able to **only build what's needed**. E.g. if `MyFile.res` isn't changed, then it's not recompiled. You can roughly emulate such incrementalism in languages like JavaScript, but the degree of correctness is unfortunately low. For example, if you rename or move a JS file, then the watcher might get confused and not pick up the "new" file or fail to clean things up correctly, resulting in you needing to clean your build and restart anew, which defeats the purpose.

Say goodbye to stale build from your JavaScript ecosystem!

Speed Up Incremental Build

ReScript uses the concept of interface files (`.resi`) (or, equivalently, module signatures). Exposing only what you need naturally speeds up incremental builds. E.g. if you change a `.res` file whose corresponding `.resi` file doesn't expose the changed part, then you've reduced the amount of dependent files you have to rebuild.

Programmatic Usage

Unfortunately, JS build systems are usually the bottleneck for building a JS project nowadays. Having parts of the build blazingly fast doesn't matter much if the rest of the build takes seconds or literally minutes. Here are a few suggestions:

- Convert more files into ReScript =). Fewer files going through fewer parts of the JS pipeline helps a ton.
- Careful with bringing in more dependencies: libraries, syntax transforms (e.g. the unofficially supported PPX), build step loaders, etc. The bulk of these dragging down the editing & building experience might out-weight the API benefits they provide.

Hot Reloading

Hot reloading refers to maintaining a dev server and listening to file changes in a way that allows the server to pipe some delta changes right into the currently running browser page. This provides a relatively fast iteration workflow while working in specific frameworks.

However, hot reloading is fragile by nature, and counts on the occasional inconsistencies (bad state, bad eval, etc.) and the heavy devserver setup/config being less of a hassle than the benefits it provides. We err on the side of caution and stability in general, and decided not to provide a built-in hot reloading *yet*. **Note**: you can still use the hot reloading facility provided by your JS build pipeline.

SECTION jsx

---

title: "JSX" description: "JSX syntax in ReScript and React" canonical: "/docs/manual/latest/jsx"

JSX

Would you like some HTML syntax in your ReScript? If not, quickly skip over this section and pretend you didn't see anything!

ReScript supports the JSX syntax, with some slight differences compared to the one in ReactJS. ReScript JSX isn't tied to ReactJS; they translate to normal function calls:

Note for ReScriptReact readers: this isn't what ReScriptReact turns JSX into, in the end. See Usage section for more info.

Capitalized

```
<MyComponent name={"ReScript"} />

React.createElement(MyComponent, {
```

```
  name: "ReScript",
});
```

becomes

```
MyComponent.createElement(~name="ReScript", ~children=list{}, ())
```

```
React.createElement(MyComponent, {
  name: "ReScript",
});
```

Uncapitalized

```
<div onClick={handler}> child1 child2 </div>
```

```
React.createElement("div", {
  onClick: handler
}, child1, child2);
```

becomes

```
div(~onClick=handler, ~children=list{child1, child2}, ())
```

```
React.createElement("div", {
  onClick: handler
}, child1, child2);
```

Fragment

```
<> child1 child2 </>
```

```
React.createElement(React.Fragment, undefined, child1, child2);
```

becomes

```
list{child1, child2}
```

```
React.createElement(React.Fragment, undefined, child1, child2);
```

Children

```
<MyComponent> child1 child2 </MyComponent>
```

```
React.createElement(MyComponent, { children: null }, child1, child2);
```

This is the syntax for passing a list of two items, `child1` and `child2`, to the children position. It transforms to a list containing `child1` and `child2`:

```
MyComponent.createElement(~children=list{child1, child2}, ())
```

```
React.createElement(MyComponent.make, MyComponent.makeProps(null, undefined), child1, child2);
```

Note again that this isn't the transform for ReScriptReact; ReScriptReact turns the final list into an array. But the idea still applies.

So naturally, `<MyComponent> myChild </MyComponent>` is transformed to `MyComponent.createElement(~children=list{myChild}, ())`. I.e. whatever you do, the arguments passed to the children position will be wrapped in a list.

Usage

See ReScriptReact Elements & JSX for an example application of JSX, which transforms the above calls into a ReScriptReact-specific call.

Here's a JSX tag that shows most of the features.

```
<MyComponent
  booleanAttribute={true}
  stringAttribute="string"
  intAttribute=1
  forcedOptional=?{Some("hello")}
  onClick={handleClick}>
  <div> {React.string("hello")} </div>
</MyComponent>
```

```
React.createElement(MyComponent, {
  children: React.createElement("div", undefined, "hello"),
  booleanAttribute: true,
  stringAttribute: "string",
  intAttribute: 1,
  forcedOptional: "hello",
  onClick: handleClick
});
```

Departures From JS JSX

- Attributes and children don't mandate `{}`, but we show them anyway for ease of learning. Once you format your file, some of them go away and some turn into parentheses.
- Props spread is supported, but there are some restrictions (see below).
- Punning!

Spread Props

Since 10.1

JSX props spread is supported now, but in a stricter way than in JS.

```
<Comp {...props} a="a" />
```

```
React.createElement(Comp, {
  a: "a",
  b: "b"
});
```

Multiple spreads are not allowed:

```
<NotAllowed {...props1} {...props2} />
```

The spread must be at the first position, followed by other props:

```
<NotAllowed a="a" {...props} />
```

Punning

"Punning" refers to the syntax shorthand for when a label and a value are the same. For example, in JavaScript, instead of doing `return {name: name}`, you can do `return {name}`.

JSX supports punning. `<input checked />` is just a shorthand for `<input checked=checked />`. The formatter will help you format to the punned syntax whenever possible. This is convenient in the cases where there are lots of props to pass down:

```
<MyComponent isLoading text onClick />
```

```
React.createElement(MyComponent, {
  isLoading: true,
  text: text,
  onClick: onClick
});
```

Consequently, a JSX component can cram in a few more props before reaching for extra libraries solutions that avoids props passing.

Note that this is a departure from ReactJS JSX, which does **not** have punning. ReactJS' `<input checked />` desugars to `<input checked=true />`, in order to conform to DOM's idioms and for backward compatibility.

Tip & Tricks

For library authors wanting to take advantage of the JSX: the `@JSX` attribute is a hook for potential ppx macros to spot a function wanting to format as JSX. Once you spot the function, you can turn it into any other expression.

This way, everyone gets to benefit the JSX syntax without needing to opt into a specific library using it, e.g. ReScriptReact.

JSX calls supports the features of [labeled arguments]: optional, explicitly passed optional and optional with default.

SECTION lazy-values

---

title: "Lazy Value" description: "Data type for deferred computation in ReScript" canonical: "/docs/manual/latest/lazy-values"

Lazy Value

If you have some expensive computations you'd like to **defer and cache** subsequently, you can wrap it with `lazy`:

```res prelude @module("node:fs") external readdirSync: string => array = "readdirSync"

// Read the directory, only once let expensiveFilesRead = lazy(() => Js.log("Reading dir") readdirSync("./pages"))

```js
var Fs = require("fs");
```
```

```
var expensiveFilesRead = {
  LAZY_DONE: false,
  VAL: (function () {
    console.log("Reading dir");
    return Fs.readdirSync("./pages");
  })
};
```

Check the JS Output tab: that `expensiveFilesRead`'s code isn't executed yet, even though you declared it! You can carry it around without fearing that it'll run the directory read.

Note: a lazy value is **not** a shared data type. Don't rely on its runtime representation in your JavaScript code.

Execute The Lazy Computation

To actually run the lazy value's computation, use `Lazy.force` from the globally available `Lazy` module:

```res example // First call. The computation happens Js.log(Lazy.force(expensiveFilesRead)) // logs "Reading dir" and the directory content

// Second call. Will just return the already calculated result Js.log(Lazy.force(expensiveFilesRead)) // logs the directory content
```

```js
console.log(CamlinternalLazy.force(expensiveFilesRead));

console.log(CamlinternalLazy.force(expensiveFilesRead));
```

The first time `Lazy.force` is called, the expensive computation happens and the result is **cached**. The second time, the cached value is directly used.

You can't re-trigger the computation after the first `force` call. Make sure you only use a lazy value with computations whose results don't change (e.g. an expensive server request whose response is always the same).

Instead of using `Lazy.force`, you can also use pattern matching to trigger the computation:

```res example switch expensiveFilesRead { | lazy(result) => Js.log(result) }
```

```js
var result = CamlinternalLazy.force(expensiveFilesRead);
```

Since pattern matching also works on a `let` binding, you can also do:

```res example let lazy(result) = expensiveFilesRead Js.log(result)
```

```js
var result = CamlinternalLazy.force(expensiveFilesRead);
console.log(result);
```

Exception Handling

For completeness' sake, our files read example might raise an exception because of `readdirSync`. Here's how you'd handle it:

```res example let result = try { Lazy.force(expensiveFilesRead) } catch { | Not_found => [] // empty array of files }
```

```js
var result;

try {
  result = CamlinternalLazy.force(expensiveFilesRead);
} catch (raw_exn) {
  var exn = Caml_js_exceptions.internalToOCamlException(raw_exn);
  if (exn.RE_EXN_ID === "Not_found") {
    result = [];
  } else {
    throw exn;
  }
}
```

Though you should probably handle the exception inside the lazy computation itself.

SECTION unboxed

---

title: "Unboxed" description: "Unbox a wrapper" canonical: "/docs/manual/latest/unboxed"

Unboxed

Consider a ReScript variant with a single payload, and a record with a single field:

```
type name = Name(string)
let studentName = Name("Joe")

type greeting = {message: string}
let hi = {message: "hello!"}
```

```
var studentName = /* Name */{
  _0: "Joe"
};

var hi = {
  message: "hello!"
};
```

If you check the JavaScript output, you'll see the `studentName` and `hi` JS object, as expected (see the variant JS output and record JS output sections for details).

For performance and certain JavaScript interop situations, ReScript offers a way to unwrap (aka unbox) the JS object wrappers from the output for records with a single field and variants with a single constructor and single payload. Annotate their type declaration with the attribute `@unboxed`:

```
@unboxed
type name = Name(string)
let studentName = Name("Joe")

@unboxed
type greeting = {message: string}
let hi = {message: "hello!"}
```

```
var studentName = "Joe";

var hi = "hello!";
```

Check the new output! Clean.

Usage

Why would you ever want a variant or a record with a single payload? Why not just... pass the payload? Here's one use-case for variant.

Suppose you have a game with a local/global coordinate system:

```res example type coordinates = {x: float, y: float}

let renderDot = (coordinates) => { Js.log3("Pretend to draw at:", coordinates.x, coordinates.y) }

let toWorldCoordinates = (localCoordinates) => { { x: localCoordinates.x +. 10., y: localCoordinates.x +. 20., } }

let playerLocalCoordinates = {x: 20.5, y: 30.5}

renderDot(playerLocalCoordinates)
```

```js
function renderDot(coordinates) {
  console.log("Pretend to draw at:", coordinates.x, coordinates.y);
}

function toWorldCoordinates(localCoordinates) {
  return {
    x: localCoordinates.x + 10,
    y: localCoordinates.x + 20
  };
}

var playerLocalCoordinates = {
  x: 20.5,
  y: 30.5
};

renderDot(playerLocalCoordinates);
```

Oops, that's wrong! `renderDot` should have taken global coordinates, not local ones... Let's prevent passing the wrong kind of coordinates:

```res example type coordinates = {x: float, y: float} @unboxed type localCoordinates = Local(coordinates) @unboxed type worldCoordinates = World(coordinates)

let renderDot = (World(coordinates)) => { Js.log3("Pretend to draw at:", coordinates.x, coordinates.y) }

let toWorldCoordinates = (Local(coordinates)) => { World({ x: coordinates.x +. 10., y: coordinates.x +. 20., }) }
```

let playerLocalCoordinates = Local({x: 20.5, y: 30.5})

// This now errors! // renderDot(playerLocalCoordinates) // We're forced to do this instead: renderDot(playerLocalCoordinates->toWorldCoordinates)

```
```js
function renderDot(coordinates) {
  console.log("Pretend to draw at:", coordinates.x, coordinates.y);
}

function toWorldCoordinates(coordinates) {
  return {
    x: coordinates.x + 10,
    y: coordinates.x + 20
  };
}

var playerLocalCoordinates = {
  x: 20.5,
  y: 30.5
};

renderDot(toWorldCoordinates(playerLocalCoordinates));
```

Now `renderDot` only takes `worldCoordinates`. Through a nice combination of using distinct variant types + argument destructuring, we've achieved better safety **without compromising on performance**: the `unboxed` attribute compiled to clean, variant-wrapper-less JS code! Check the output.

As for a record with a single field, the use-cases are a bit more edgy. We won't mention them here.

SECTION installation

---

title: "Installation" description: "ReScript installation and setup instructions" canonical: "/docs/manual/latest/installation"

Installation

Prerequisites

- Node.js version >= 14
- npm (which comes with Node.js) or Yarn

New Project

```
git clone https://github.com/rescript-lang/rescript-project-template
cd rescript-project-template
npm install
npm run res:build
node src/Demo.bs.js
```

or use the create-rescript-app tool:

```
npm create rescript-app // Select basic template
cd <your-rescript-project-name>
npm run res:build
node src/Demo.bs.js
```

That compiles your ReScript into JavaScript, then uses Node.js to run said JavaScript. **We recommend you use our unique workflow of keeping a tab open for the generated `.bs.js` file**, so that you can learn how ReScript transforms into JavaScript. Not many languages output clean JavaScript code you can inspect and learn from!

During development, instead of running `npm run res:build` each time to compile, use `npm run res:dev` to start a watcher that recompiles automatically after file changes.

Integrate Into an Existing JS Project

If you already have a JavaScript project into which you'd like to add ReScript:

- Install ReScript locally: `sh npm install rescript`
- Create a ReScript build configuration at the root: `json { "name": "your-project-name", "sources": [ { "dir": "src", // update this to wherever you're putting ReScript files "subdirs": true } ], "package-specs": [ { "module": "es6", "in-source": true } ], "suffix": ".bs.js", "bs-dependencies": [] }` See Build Configuration for more details on `rescript.json`.
- Add convenience `npm` scripts to `package.json`: `json "scripts": { "res:build": "rescript", "res:dev": "rescript build -w" }`
```

Since ReScript compiles to clean readable JS files, the rest of your existing toolchain (e.g. Babel and Webpack) should just work!

Helpful guides:

- Converting from JS.
- Shared Data Types.
- Import from/Export to JS.

Integrate with a ReactJS Project

To start a rescript-react app, or to integrate ReScript into an existing ReactJS app, follow the instructions here.

SECTION polymorphic-variant

---

title: "Polymorphic Variant" description: "The Polymorphic Variant data structure in ReScript" canonical: "/docs/manual/latest/polymorphic-variant"

Polymorphic Variant

Polymorphic variants (or poly variant) are a cousin of variant. With these differences:

- They start with a `#` and the constructor name doesn't need to be capitalized.
- They don't require an explicit type definition. The type is inferred from usage.
- Values of different poly variant types can share the constructors they have in common (aka, poly variants are "structurally" typed, as opposed to "nominally" typed).

They're a convenient and useful alternative to regular variants, but should **not** be abused. See the drawbacks at the end of this page.

Creation

We provide 3 syntaxes for a poly variant's constructor:

```
let myColor = #red
let myLabel = #"aria-hidden"
let myNumber = #7
```

```
var myColor = "red";
var myLabel = "aria-hidden";
var myNumber = 7;
```

Take a look at the output. Poly variants are *great* for JavaScript interop. For example, you can use it to model JavaScript string and number enums like TypeScript, but without confusing their accidental usage with regular strings and numbers.

`myColor` uses the common syntax. The second and third syntaxes are to support expressing strings and numbers more conveniently. We allow the second one because otherwise it'd be invalid syntax since symbols like - and others are usually reserved.

Type Declaration

Although **optional**, you can still pre-declare a poly variant type:

```
// Note the surrounding square brackets, and ### for constructors
type color = [#red | #green | #blue]
```

These types can also be inlined, unlike for regular variant:

```
let render = (myColor: [#red | #green | #blue]) => {
  switch myColor {
  | #blue => Js.log("Hello blue!")
  | #red
  | #green => Js.log("Hello other colors")
  }
}
```

```
function render(myColor) {
  if (myColor === "green" || myColor === "red") {
    console.log("Hello other colors");
  } else {
    console.log("Hello blue!");
  }
}
```

Note: because a poly variant value's type definition is **inferred** and not searched in the scope, the following snippet won't error:

```
type color = [#red | #green | #blue]
```

```
let render = myColor => {
  switch myColor {
  | #blue => Js.log("Hello blue!")
  | #green => Js.log("Hello green!")
  // works!
  | #yellow => Js.log("Hello yellow!")
  }
}
```

```
function render(myColor) {
  if (myColor === "yellow") {
    console.log("Hello yellow!");
  } else if (myColor === "green") {
    console.log("Hello green!");
  } else {
    console.log("Hello blue!");
  }
}
```

That `myColor` parameter's type is inferred to be `#red`, `#green` or `#yellow`, and is unrelated to the `color` type. If you intended `myColor` to be of type `color`, annotate it as `myColor: color` in any of the places.

Constructor Arguments

This is similar to a regular variant's constructor arguments:

```
type account = [
  | #Anonymous
  | #Instagram(string)
  | #Facebook(string, int)
]

let me: account = #Instagram("Jenny")
let him: account = #Facebook("Josh", 26)
```

```
var me = {
  NAME: "Instagram",
  VAL: "Jenny"
};

var him = {
  NAME: "Facebook",
  VAL: [
    "Josh",
    26
  ]
};
```

Combine Types and Pattern Match

You can use poly variant types within other poly variant types to create a sum of all constructors:

```
type red = [#Ruby | #Redwood | #Rust]
type blue = [#Sapphire | #Neon | #Navy]

// Contains all constructors of red and blue.
// Also adds #Papayawhip
type color = [red | blue | #Papayawhip]

let myColor: color = #Ruby
```

```
var myColor = "Ruby";
```

There's also some special pattern matching syntax to match on constructors defined in a specific poly variant type:

```
// Continuing the previous example above...

switch myColor {
| #...blue => Js.log("This blue-ish")
| #...red => Js.log("This red-ish")
| other => Js.log2("Other color than red and blue: ", other)
}
```

```
var other = myColor;

if (other === "Neon" || other === "Navy" || other === "Sapphire") {
  console.log("This is blue-ish");
} else if (other === "Rust" || other === "Ruby" || other === "Redwood") {
  console.log("This is red-ish");
} else {
  console.log("Other color than red and blue: ", other);
}
```

This is a shorter version of:

```
switch myColor {
| #Sapphire | #Neon | #Navy => Js.log("This is blue-ish")
| #Ruby | #Redwood | #Rust => Js.log("This is red-ish")
| other => Js.log2("Other color than red and blue: ", other)
}
```

Structural Sharing

Since poly variants value don't have a source of truth for their type, you can write such code:

```
type preferredColors = [#white | #blue]

let myColor: preferredColors = #blue

let displayColor = v => {
  switch v {
  | #red => "Hello red"
  | #green => "Hello green"
  | #white => "Hey white!"
  | #blue => "Hey blue!"
  }
}

Js.log(displayColor(myColor))
```

```
var myColor = "blue";

function displayColor(v) {
  if (v === "white") {
    return "Hey white!";
  } else if (v === "red") {
    return "Hello red";
  } else if (v === "green") {
    return "Hello green";
  } else {
    return "Hey blue!";
  }
}

console.log(displayColor("blue"));
```

With a regular variant, the line `displayColor(myColor)` would fail, since it'd complain that the type of `myColor` doesn't match the type of `v`. No problem with poly variant.

JavaScript Output

Poly variants are great for JavaScript interop! You can share their values to JS code, or model incoming JS values as poly variants.

- `#red` and `#"I am red ðŸ˜ƒ"` compile to JavaScipt `"red"` and `"I am red ðŸ˜ƒ"`.
- `#1` compiles to JavaScript `1`.
- Poly variant constructor with 1 argument, like `Instagram("Jenny")` compile to a straightforward `{NAME: "Instagram", VAL: "Jenny"}`. 2 or more arguments like `#Facebook("Josh", 26)` compile to a similar object, but with `VAL` being an array of the arguments.

Bind to Functions

For example, let's assume we want to bind to `Intl.NumberFormat` and want to make sure that our users only pass valid locales, we could define an external binding like this:

```
type t

@scope("Intl") @val
external makeNumberFormat: ([#"de-DE" | #"en-GB" | #"en-US"]) => t = "NumberFormat"

let intl = makeNumberFormat(#"de-DE")
```

```
var intl = Intl.NumberFormat("de-DE");
```

The JS output is identical to handwritten JS, but we also get to enjoy type errors if we accidentally write `makeNumberFormat(#"de-DR")`.

More advanced usage examples for poly variant interop can be found in Bind to JS Function.

Bind to String Enums

Let's assume we have a TypeScript module that expresses following enum export:

```
// direction.js
enum Direction {
```

```
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}

export const myDirection = Direction.Up
```

For this particular example, we can also inline poly variant type definitions to design the type for the imported `myDirection` value:

```
type direction = [ #UP | #DOWN | #LEFT | #RIGHT ]
@module("./direction.js") external myDirection: direction = "myDirection"

var DirectionJs = require("./direction.js");

var myDirection = DirectionJs.myDirection;
```

Again: since we were using poly variants, the JS Output is practically zero-cost and doesn't add any extra code!

Extra Constraints on Types

The previous poly variant type annotations we've looked at are the regular "closed" kind. However, there's a way to express "I want at least these constructors" (lower bound) and "I want at most these constructors" (upper bound):

```
// Only #Red allowed. Closed.
let basic: [#Red] = #Red

// May contain #Red, or any other value. Open
// here, foreground will actually be inferred as [> #Red | #Green]
let foreground: [> #Red] = #Green

// The value must be, at most, one of #Red or #Blue
// Only #Red and #Blue are valid values
let background: [< #Red | #Blue] = #Red
```

Note: We added this info for educational purposes. In most cases you will not want to use any of this stuff, since it makes your APIs pretty unreadable / hard to use.

Closed `[`

This is the simplest poly variant definition, and also the most practical one. Like a common variant type, this one defines an exact set of constructors.

```
type rgb = [ #Red | #Green | #Blue ]

let color: rgb = #Green
```

In the example above, `color` will only allow one of the three constructors that are defined in the `rgb` type. This is usually the way how poly variants should be defined.

In case you want to define a type that is extensible, you'll need to use the lower / upper bound syntax.

Lower Bound `[>`

A lower bound defines the minimum set of constructors a poly variant type is aware of. It is also considered an "open poly variant type", because it doesn't restrict any additional values.

Here is an example on how to make a minimum set of `basicBlueTones` extensible for a new `color` type:

```
type basicBlueTone<'a> = [> #Blue | #DeepBlue | #LightBlue ] as 'a
type color = basicBlueTone<[#Blue | #DeepBlue | #LightBlue | #Purple]>

let color: color = #Purple

// This will fail due to missing minimum constructors:
type notWorking = basicBlueTone<[#Purple]>
```

Here, the compiler will enforce the user to define `#Blue | #DeepBlue | #LightBlue` as the minimum set of constructors when trying to extend `basicBlueTone<'a>`.

Note: Since we want to define an extensible poly variant, we need to provide a type placeholder `<'a>`, and also add `as 'a` after the poly variant declaration, which essentially means: "Given type `'a` is constraint to the minimum set of constructors (`#Blue | #DeepBlue | #LightBlue`) defined in `basicBlueTone`".

Upper Bound `[<`

The upper bound works in the opposite way than a lower bound: the extending type may only use constructors that are stated in the upper bound constraint.

Here another example, but with red colors:

```
type validRed<'a> = [< #Fire | #Crimson | #Ash] as 'a
type myReds = validRed<[#Ash]>

// This will fail due to unlisted constructor not defined by the lower bound
type notWorking = validRed<[#Purple]>
```

Coercion

You can convert a poly variant to a `string` or `int` at no cost:

```
type company = [#Apple | #Facebook]
let theCompany: company = #Apple

let message = "Hello " ++ (theCompany :> string)
```

```
var theCompany = "Apple";
var message = "Hello " + theCompany;
```

Note: for the coercion to work, the poly variant type needs to be closed; you'd need to annotate it, since otherwise, `theCompany` would be inferred as `[> #Apple]`.

Tips & Tricks

Variant vs Polymorphic Variant

One might think that polymorphic variants are superior to regular variants. As always, there are trade-offs:

- Due to their "structural" nature, poly variant's type errors might be more confusing. If you accidentally write `#blur` instead of `#blue`, ReScript will still error but can't indicate the correct source as easily. Regular variants' source of truth is the type definition, so the error can't go wrong.
- It's also harder to refactor poly variants. Consider this: `res let myFruit = #Apple let mySecondFruit = #Apple let myCompany = #Apple` Refactoring the first one to `#Orange` doesn't mean we should refactor the third one. Therefore, the editor plugin can't touch the second one either. Regular variant doesn't have such problem, as these 2 values presumably come from different variant type definitions.
- You might lose some nice pattern match checks from the compiler: ```res let myColor = #red

switch myColor { | #red => Js.log("Hello red!") | #blue => Js.log("Hello blue!") } `` Because there's no poly variant definition, it's hard to know whether the#blue` case can be safely removed.

In most scenarios, we'd recommend to use regular variants over polymorphic variants, especially when you are writing plain ReScript code. In case you want to write zero-cost interop bindings or generate clean JS output, poly variants are oftentimes a better option.

SECTION object

---

title: "Object" description: "Interoping with JS objects in ReScript" canonical: "/docs/manual/latest/object"

Object

ReScript objects are like records, but:

- No type declaration needed.
- Structural and more polymorphic, unlike records.
- Doesn't support updates unless the object comes from the JS side.
- Doesn't support pattern matching.

Although ReScript records compile to clean JavaScript objects, ReScript objects are a better candidate for emulating/binding to JS objects, as you'll see.

Type Declaration

Optional, unlike for records. The type of an object is inferred from the value, so you never really need to write down its type definition. Nevertheless, here's its type declaration syntax:

```res prelude type person = { "age": int, "name": string };

```js
// Empty output
```
```

Visually similar to record type's syntax, with the field names quoted.

Creation

To create a new object:

```res example let me = { "age": 5, "name": "Big ReScript" }

```js
var me = {
  "age": 5,
  "name": "Big ReScript"
};
```

Note: as said above, unlike for record, this `me` value does **not** try to find a conforming type declaration with the field `"age"` and `"name"`; rather, the type of `me` is inferred as `{"age": int, "name": string}`. This is convenient, but also means this code passes type checking without errors:

```
type person = {
  "age": int
};

let me = {
  "age": "hello!" // age is a string. No error.
}

var me = {
  "age": "hello!"
};
```

Since the type checker doesn't try to match `me` with the type `person`. If you ever want to force an object value to be of a predeclared object type, just annotate the value:

```
let me: person = {
  "age": "hello!"
}
```

Now the type system will error properly.

Access

```
let age = me["age"]

var age = me["age"];
```

Update

Disallowed unless the object is a binding that comes from the JavaScript side. In that case, use `=`

```res example type student = { @set "age": int, @set "name": string, } @module("MyJSFile") external student1: student = "student1"

student1["name"] = "Mary"

```js
var MyJSFile = require("MyJSFile");
MyJSFile.student1.name = "Mary";
```

Combine Types

You can spread one object type definition into another using `...`:

```res example type point2d = { "x": float, "y": float, } type point3d = { ...point2d, "z": float, }

let myPoint: point3d = { "x": 1.0, "y": 2.0, "z": 3.0, }

```js
var myPoint = {
  x: 1.0,
  y: 2.0,
  z: 3.0
};
```

This only works with object types, not object values!

Tips & Tricks

Since objects don't require type declarations, and since ReScript infers all the types for you, you get to very quickly and easily (and dangerously) bind to any JavaScript API. Check the JS output tab:
```

```res example // The type of document is just some random type 'a // that we won't bother to specify @val external document: 'a = "document"

// call a method document"addEventListener" })

// get a property let loc = document["location"]

// set a property document["location"]["href"] = "rescript-lang.org"

```js
document.addEventListener("mouseup", function(_event) {
  console.log("clicked!");
});
var loc = document.location;
document.location.href = "rescript-lang.org";
```

The `external` feature and the usage of this trick are also documented in the external section later. It's an excellent way to start writing some ReScript code without worrying about whether bindings to a particular library exists.

SECTION build-overview

---

title: "Overview" metaTitle: "Build System Overview" description: "Documentation about the ReScript build system and its toolchain" canonical: "/docs/manual/latest/build-overview"

Build System Overview

ReScript comes with a build system, `rescript`, that's fast, lean and used as the authoritative build system of the community.

Every ReScript project needs a build description file, `rescript.json`.

Options

See `rescript -help`:

```
â¯ rescript -help
Available flags
-v, -version  display version number
-h, -help     display help
Subcommands:
    build
    clean
    format
    convert
    help
Run rescript subcommand -h for more details,
For example:
    rescript build -h
    rescript format -h
The default `rescript` is equivalent to `rescript build` subcommand
```

Build Project

Each build will create build artifacts from your project's source files.

To build a project (including its dependencies / pinned-dependencies), run:

```
rescript
```

Which is an alias for `rescript build`.

To keep a build watcher, run:

```
rescript build -w
```

Any new file change will be picked up and the build will re-run.

Note: third-party libraries (in `node_modules`, or via `pinned-dependencies`) aren't watched, as doing so may exceed the node.js watcher count limit.

Note 2: In case you want to set up a project in a JS-monorepo-esque approach (`npm` and `yarn` workspaces) where changes in your sub packages should be noticed by the build, you will need to define pinned dependencies in your main project's `rescript.json`. More details here.
```

Clean Project

If you ever get into a stale build for edge-case reasons, use:

```
rescript clean
```

This will clean your own project's build artifacts. To also clean the dependencies' artifacts:

```
rescript clean -with-deps
```

SECTION shared-data-types

---

title: "Shared Data Types" description: "Data types that share runtime presentation between JS and ReScript" canonical: "/docs/manual/latest/shared-data-types"

Shared Data Types

ReScript's built-in values of type `string`, `float`, `array` and a few others have a rather interesting property: they compile to the exact same value in JavaScript!

This means that if you're passing e.g. a ReScript string to the JavaScript side, the JS side can directly use it as a native JS string. It also means that you can import a JS string and pretend it's a native ReScript string.

Unlike most compiled-to-js languages, in ReScript, **you don't need to write data converters back and forth for most of our values**!

Shared, bidirectionally usable types: - String. ReScript strings are JavaScript strings, vice-versa. (Caveat: only our backtick string `` `hello ðŸ‘‹ ${personName}` `` supports unicode and interpolation). - Float. ReScript floats are JS numbers, vice-versa. - Array. In addition to the JS Array API, we provide our own Belt.Array API too. - Tuple. Compiles to a JS array. You can treat a fixed-sized, heterogenous JS array as ReScript tuple too. - Boolean. - Record. Record compiles to JS object. Therefore you can also treat JS objects as records. If they're too dynamic, consider modeling them on the ReScript side as a hashmap/dictionary `Js.Dict` or a ReScript object. - Object. ReScript objects are JavaScript objects, vice-versa. - Function. They compile to clean JS functions. - Module. ReScript files are considered top-level modules, and are compiled to JS files 1 to 1. Nested modules are compiled to JavaScript objects. - Polymorphic variants. - Unit. The `unit` type, which has a single value `()`, compiles to `undefined` too. Likewise, you can treat an incoming JS `undefined` as `()` if that's the only value it'll ever be.

Types that are slightly different than JS, but that you can still use from JS: - Int. **Ints are 32-bits**! Be careful, you can potentially treat them as JS numbers and vice-versa, but if the number's large, then you better treat JS numbers as floats. For example, we bind to Js.Date using `float`s. - Option. The `option` type's `None` value compiles into JS `undefined`. The `Some` value, e.g. `Some(5)`, compiles to `5`. Likewise, you can treat an incoming JS `undefined` as `None`. **JS `null` isn't handled here**. If your JS value can be `null`, use Js.Nullable helpers. - Exception. - Variant. Check the compiled JavaScript output of variant to see its shape. We don't recommend exporting a ReScript variant for pure JS usage, since they're harder to read as plain JS code, but you can do it. - List, which is just a regular variant.

Non-shared types (aka internal types):

- Character.
- Int64.
- Lazy values.
- Everything else.

Many of these are stable, which means that you can still serialize/deserialize them as-is without manual conversions. But we discourage actively peeking into their structure otherwise.

These types require manual conversions if you want to export them for JS consumption. For a seamless JS/TypeScript/Flow integration experience, you might want to use genType instead of doing conversions by hand.

SECTION REACT

SECTION components-and-props

---

title: Components and Props description: "Basic concepts for components and props in ReScript & React" canonical: "/docs/react/latest/components-and-props"

Components and Props

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. This page provides an introduction to the idea of components.

What is a Component?

A React component is a function describing a UI element that receives a `props` object as a parameter (data describing the dynamic parts of the UI) and returns a `React.element`.

The nice thing about this concept is that you can solely focus on the input and output. The component function receives some data and returns some opaque `React.element` that is managed by the React framework to render your UI.

> If you want to know more about the low level details on how a component interface is implemented, refer to the Beyond JSX page.

Component Example

Let's start with a first example to see how a ReScript React component looks like:

```
// src/Greeting.res
@react.component
let make = () => {
  <div>
    {React.string("Hello ReScripters!")}
  </div>
}
```

```
import * as React from "react";

function Greeting(props) {
  return React.createElement("div", undefined, "Hello ReScripters!");
}

var make = Greeting;
```

Important: Always make sure to name your component function `make`.

We've created a `Greeting.res` file that contains a `make` function that doesn't receive any props (the function doesn't receive any parameters), and returns a `React.element` that represents `<div> Hello ReScripters! </div>` in the rendered DOM.

You can also see in the the JS output that the function we created was directly translated into the pure JS version of a ReactJS component. Note how a `<div>` transforms into a `React.createElement("div",...)` call in JavaScript.

Defining Props

In ReactJS, props are usually described as a single `props` record. In ReScript, we use labeled arguments to define our props parameters instead. Here's an example:

```
// src/Article.res
@react.component
let make = (~title: string, ~visitorCount: int, ~children: React.element) => {
  let visitorCountMsg = "You are visitor number: " ++ Belt.Int.toString(visitorCount);
  <div>
    <div> {React.string(title)} </div>
    <div> {React.string(visitorCountMsg)} </div>
    children
  </div>
}
```

```
import * as React from "react";

function Article(props) {
  var visitorCountMsg = "You are visitor number: " + String(props.visitorCount);
  return React.createElement("div", undefined, React.createElement("div", undefined, props.title), React.createE
lement("div", undefined, visitorCountMsg), props.children);
}

var make = Article;
```

Optional Props

We can leverage the full power of labeled arguments to define optional props as well:

```
// Greeting.res
@react.component
let make = (~name: option<string>=?) => {
  let greeting = switch name {
  | Some(name) => "Hello " ++ name ++ "!"
  | None => "Hello stranger!"
  }
  <div> {React.string(greeting)} </div>
}
```

```
function Greeting(props) {
```

```
  var name = props.name;
  var greeting = name !== undefined ? "Hello " + name + "!" : "Hello stranger!";
  return React.createElement("div", undefined, greeting);
}
```

Note: The `@react.component` attribute implicitly adds the last `()` parameter to our `make` function for us (no need to do it ourselves).

In JSX, you can apply optional props with some special syntax:

```
let name = Some("Andrea")

<Greeting ?name />

var name = "Andrea";

React.createElement(Greeting, {
  name: name
});
```

Special Props `key` and `ref`

You can't define any props called `key` or `ref`. React treats those props differently and the compiler will yield an error whenever you try to define a `~key` or `~ref` argument in your component function.

Check out the corresponding Arrays and Keys and Forwarding React Refs sections for more details.

Handling Invalid Prop Names (e.g. keywords)

Prop names like `type` (as in `<input type="text" />`) aren't syntactically valid; `type` is a reserved keyword in ReScript. Use `<input type_="text" />` instead.

For `aria-*` use camelCasing, e.g., `ariaLabel`. For DOM components, we'll translate it to `aria-label` under the hood.

For `data-*` this is a bit trickier; words with `-` in them aren't valid in ReScript. When you do want to write them, e.g., `<div data-name="click me" />`, check out the React.cloneElement or React.createDOMElementVariadic section.

Children Props

In React `props.children` is a special attribute to represent the nested elements within a parent element:

```
let element = <div> child1 child2 </div>
```

By default, whenever you are passing children like in the expression above, `children` will be treated as a `React.element`:

```
module MyList = {
  @react.component
  let make = (~children: React.element) => {
    <ul>
      children
    </ul>
  }
}

<MyList>
  <li> {React.string("Item 1")} </li>
  <li> {React.string("Item 2")} </li>
</MyList>

function MyList(props) {
  return React.createElement("ul", undefined, props.children);
}

var MyList = {
  make: MyList
};

React.createElement(MyList, {
    children: null
  }, React.createElement("li", undefined, "Item 1"),
    React.createElement("li", undefined, "Item 2"));
```

Interestingly, it doesn't matter if you are passing just one element, or several, React will always collapse its children to a single `React.element`.

It is also possible to redefine the `children` type as well. Here are some examples:

Component with a mandatory `string` as children:

```
module StringChildren = {
  @react.component
```

```
  let make = (~children: string) => {
    <div>
      {React.string(children)}
    </div>
  }
}

<StringChildren> "My Child" </StringChildren>

// This will cause a type check error
<StringChildren/>
```

Component with an optional `React.element` as children:

```
module OptionalChildren = {
  @react.component
  let make = (~children: option<React.element>=?) => {
    <div>
      {switch children {
      | Some(element) => element
      | None => React.string("No children provided")
      }}
    </div>
  }
}

<div>
  <OptionalChildren />
  <OptionalChildren> <div /> </OptionalChildren>
</div>
```

Component that doesn't allow children at all:

```
module NoChildren = {
  @react.component
  let make = () => {
    <div>
      {React.string("I don't accept any children params")}
    </div>
  }
}

// The compiler will raise a type error here
<NoChildren> <div/> </NoChildren>
```

Children props are really tempting to be abused as a way to model hierarchies, e.g. `<List> <ListHeader/> <Item/> </List>` (`List` should only allow `Item` / `ListHeader` elements), but this kind of constraint is hard to enforce because all components end up being a `React.element`, so it would require notorious runtime checking within `List` to verify that all children are in fact of type `Item` or `ListHeader`.

The best way to approach this kind of issue is by using props instead of children, e.g. `<List header="..." items=[{id: "...", text: "..."}]/>`. This way it's easy to type check the constraints, and it also spares component consumers from memorizing and remembering component constraints.

The best use-case for `children` is to pass down `React.element`s without any semantic order or implementation details!

Props & Type Inference

The ReScript type system is really good at inferring the prop types just by looking at its prop usage.

For simple cases, well-scoped usage, or experimentation, it's still fine to omit type annotations:

```
// Button.res

@react.component
let make = (~onClick, ~msg, ~children) => {
  <div onClick>
    {React.string(msg)}
    children
  </div>
}
```

In the example above, `onClick` will be inferred as `ReactEvent.Mouse.t => unit`, `msg` as `string` and `children` as `React.element`. Type inference is especially useful when you just forward values to some smaller (privately scoped) functions.

Even though type inference spares us a lot of keyboard typing, we still recommend to explicitly type your props (just like with any public API) for better type visibility and to prevent confusing type errors.

Using Components in JSX

Every ReScript component can be used in JSX. For example, if we want to use our `Greeting` component within our `App` component, we

can do this:

```
// src/App.res

@react.component
let make = () => {
  <div>
    <Greeting/>
  </div>
}
```

```
var React = require("react");
var Greeting = require("./Greeting.js")

function App(Props) {
  return React.createElement("div", undefined, React.createElement(Greeting.make, {}));
}

var make = App;
```

Note: React components are capitalized; primitive DOM elements like `div` or `button` are uncapitalized. More infos on the JSX specifics and code transformations can be found in our JSX language manual section.

Handwritten Components

You don't need to use the `@react.component` decorator to write components that can be used in JSX. Instead you can write the `make` function with type `props` and these will always work as React components. But then you will have the issue with the component name being "make" in the React dev tools.

For example:

```
module Link = {
  type props = {href: string, children: React.element};

  let make = (props: props) => {
    <a href={props.href}>
     {props.children}
    </a>
  }
}

<Link href="/docs"> {React.string("Docs")} </Link>
```

```
function make(props) {
  return React.createElement(
    "a",
    { href: props.href },
    props.children
  );
}

var Link = {
  make: make,
};

React.createElement(make, {
  href: "/docs",
  children: "Docs",
});
```

More details on the `@react.component` decorator and its generated interface can be found in our Beyond JSX page.

Submodule Components

We can also represent React components as submodules, which makes it very convenient to build more complex UI without the need to create multiple files for each composite component (that's probably only used by the parent component anyways):

```
// src/Button.res
module Label = {
  @react.component
  let make = (~title: string) => {
    <div className="myLabel"> {React.string(title)} </div>
  }
}

@react.component
let make = (~children) => {
  <div>
    <Label title="Getting Started" />
    children
  </div>
}
```

The `Button.res` file defined above is now containing a `Label` component, that can also be used by other components, either by writing the fully qualified module name (`<Button.Label title="My Button"/>`) or by using a module alias to shortcut the full qualifier:

```
module Label = Button.Label

let content = <Label title="Test"/>
```

Component Naming

Because components are actually a pair of functions, they have to belong to a module to be used in JSX. It makes sense to use these modules for identification purposes as well. `@react.component` automatically adds the name for you based on the module you are in.

```
// File.res

// will be named `File` in dev tools
@react.component
let make = ...

// will be named `File$component` in dev tools
@react.component
let component = ...

module Nested = {
  // will be named `File$Nested` in dev tools
  @react.component
  let make = ...
};
```

If you need a dynamic name for higher-order components or you would like to set your own name you can use `React.setDisplayName(make, "NameThatShouldBeInDevTools").`

Tips & Tricks

- Start with one component file and utilize submodule components as your component grows. Consider splitting up in multiple files when really necessary.
- Keep your directory hierarchy flat. Instead of `article/Header.res` use `ArticleHeader.res` etc. Filenames are unique across the codebase, so filenames tend to be very specific `ArticleUserHeaderCard.res`, which is not necessarily a bad thing, since it clearly expresses the intent of the component within its name, and makes it also very easy to find, match and refactor across the whole codebase.

SECTION hooks-ref

---

title: useRef Hook description: "Details about the useRef React hook in ReScript" canonical: "/docs/react/latest/hooks-ref"

---

useRef

The `useRef` hooks creates and manages mutable containers inside your React component.

Usage

```
let refContainer = React.useRef(initialValue);

  var button = React.useRef(null);
  React.useRef(0);
```

`React.useRef` returns a mutable ref object whose `.current` record field is initialized to the passed argument (`initialValue`). The returned object will persist for the full lifetime of the component.

Essentially, a `React.ref` is like a "box" that can hold a mutable value in its `.current` record field.

You might be familiar with refs primarily as a way to access the DOM. If you pass a ref object to React with `<div ref={ReactDOM.Ref.domRef(myRef)} />`, React will set its `.current` property to the corresponding DOM node whenever that node changes.

However, `useRef()` is useful for more than the ref attribute. It's handy for keeping any mutable value around similar to how youâ€™d use instance fields in classes.

This works because `useRef()` creates a plain JavaScript object. The only difference between `useRef()` and creating a `{current: ...}` object yourself is that useRef will give you the same ref object on every render.

Keep in mind that `useRef` doesnâ€™t notify you when its content changes. Mutating the `.current` record field doesnâ€™t cause a re-render. If you want to run some code when React attaches or detaches a ref to a DOM node, you may want to use a callback ref instead.

More infos on direct DOM manipulation can be found in the Refs and the DOM section.

Examples

Managing Focus for a Text Input

```
// TextInputWithFocusButton.res

@send external focus: Dom.element => unit = "focus"

@react.component
let make = () => {
  let inputEl = React.useRef(Js.Nullable.null)

  let onClick = _ => {
    inputEl.current
    ->Js.Nullable.toOption
    ->Belt.Option.forEach(input => input->focus)
  }

  <>
    <input ref={ReactDOM.Ref.domRef(inputEl)} type_="text" />
    <button onClick> {React.string("Focus the input")} </button>
  </>
}
```

```
function TextInputWithFocusButton(Props) {
  var inputEl = React.useRef(null);
  var onClick = function (param) {
    return Belt_Option.forEach(Caml_option.nullable_to_opt(inputEl.current), (function (input) {
                  input.focus();

                }));
  };
  return React.createElement(React.Fragment, undefined, React.createElement("input", {
                  ref: inputEl,
                  type: "text"
                }), React.createElement("button", {
                  onClick: onClick
                }, "Focus the input"));
}
```

Using a Callback Ref

Reusing the example from our Refs and the DOM section:

```
// CustomTextInput.res

@send external focus: Dom.element => unit = "focus"

@react.component
let make = () => {
  let textInput = React.useRef(Js.Nullable.null)
  let setTextInputRef = element => {
    textInput.current = element;
  }

  let focusTextInput = _ => {
    textInput.current
    ->Js.Nullable.toOption
    ->Belt.Option.forEach(input => input->focus)
  }

  <div>
    <input type_="text" ref={ReactDOM.Ref.callbackDomRef(setTextInputRef)} />
    <input
      type_="button" value="Focus the text input" onClick={focusTextInput}
    />
  </div>
}
```

```
function CustomTextInput(Props) {
  var textInput = React.useRef(null);
  var setTextInputRef = function (element) {
    textInput.current = element;

  };
  var focusTextInput = function (param) {
    return Belt_Option.forEach(Caml_option.nullable_to_opt(textInput.current), (function (input) {
                  input.focus();

                }));
  };
  return React.createElement("div", undefined, React.createElement("input", {
                  ref: setTextInputRef,
                  type: "text"
                }), React.createElement("input", {
```

```
                  type: "button",
                  value: "Focus the text input",
                  onClick: focusTextInput
                }));
}
```

SECTION hooks-overview

---

title: Hooks & State Management Overview description: "Overview state management and hooks in ReScript and React" canonical: "/docs/react/latest/hooks-overview"

Hooks Overview

Hooks are an essential mechanism to introduce and manage state and effects in React components.

What is a Hook?

In the previous chapters we learned how React components are just a simple function representing UI based on specific prop values. For an application to be useful we still need a way to manipulate those props interactively either via user input or via requests loading in data from a server.

That's where Hooks come in. A Hook is a function that allows us to introduce component state and trigger side-effects for different tasks, such as HTTP requests, direct HTML DOM access, querying window sizes, etc.

In other words: **It allows us to "hook into" React features.**

Example: The `useState` Hook

Just for a quick look, here is an example of a `Counter` component that allows a user to click a button and increment an `count` value that will immediately be rendered on each button click:

```
// Counter.res
@react.component
let make = () => {
  let (count, setCount) = React.useState(_ => 0);

  let onClick = (_evt) => {
    setCount(prev => prev + 1)
  };

  let msg = "You clicked" ++ Belt.Int.toString(count) ++  "times"

  <div>
    <p>{React.string(msg)}</p>
    <button onClick> {React.string("Click me")} </button>
  </div>
}
```

```
function Counter(Props) {
  var match = React.useState(function () {
        return 0;
      });
  var setCount = match[1];
  var onClick = function (_evt) {
    return Curry._1(setCount, (function (prev) {
                  return prev + 1 | 0;
                }));
  };
  var msg = "You clicked" + String(match[0]) + "times";
  return React.createElement("div", undefined, React.createElement("p", undefined, msg), React.createElement("bu
tton", {
                  onClick: onClick
                }, "Click me"));
}
```

Here we are using the `React.useState` Hook. We call it inside a component function to add some local state to it. React will preserve this state between re-renders. `React.useState` returns a tuple: the current state value (`count`) and a function that lets you update it (`setCount`). You can call this function from an event handler or pass it down to other components to call the function.

The only argument to `React.useState` is a function that returns the initial state (`_ => 0`). In the example above, it is 0 because our counter starts from zero. Note that your state can be any type you want and `ReScript` will make sure to infer the types for you (only make sure to return an initial state that matches your type). The initial state argument is only used during the first render.

This was just a quick example on our first hook usage. We will go into more detail in a dedicated useState section.

Available Hooks

Note: All hooks are part of the `React` module (e.g. `React.useState`).

Basic Hooks:

- useState: Adds local state to your component
- useEffect: Runs side-effectual code within your component
- useContext: Gives your component to a React Context value

Additional Hooks:

- useReducer: An alternative to `useState`. Uses the state / action / reduce pattern.
- useRef: Returns a mutable React-Ref value

Rules of Hooks

Hooks are just simple functions, but you need to follow *two rules* when using them. ReScript doesn't enforce those rules within the compiler, so if you really want to enforce correct hooks conventions, you can use an eslint-plugin to check your compiled JS output.

Rule 1) Only Call Hooks at the Top Level

Donâ€™t call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function. By following this rule, you ensure that Hooks are called in the same order each time a component renders. Thatâ€™s what allows React to correctly preserve the state of Hooks between multiple `useState` and `useEffect` calls. (If youâ€™re curious, you can check out the in depth explanation in the ReactJS Hooks docs)

Rule 2) Only Call Hooks from React Functions

Don't call Hooks from regular functions. Instead, you can:

- âœ… Call Hooks from React function components.
- âœ… Call Hooks from custom Hooks (weâ€™ll learn about them in our custom hooks section).

By following this rule, you ensure that all stateful logic in a component is clearly visible from its source code.

SECTION refs-and-the-dom

---

title: Refs and the DOM description: "Using Refs and DOM elements in ReScript and React" canonical: "/docs/react/latest/refs-and-the-dom"

Refs and the DOM

Refs provide a way to access DOM nodes or React elements created within your `make` component function.

In the typical React dataflow, props are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an `React.element`, or it could be a `Dom.element`. For both of these cases, React provides an escape hatch.

A `React.ref` is defined like this:

```
type t<'value> = { mutable current: 'value }
```

> *Note that the `Ref.ref` should not to be confused with the builtin ref type, the language feature that enables mutation.*

When to use Refs

There are a few good use cases for refs:

- Managing state that *should not trigger* any re-render.
- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

Creating Refs

A React ref is represented as a `React.ref('value)` type, a container managing a mutable value of type `'value`. You can create this kind of ref with the React.useRef hook:

```
@react.component
let make = () => {
  let clicks = React.useRef(0);

  let onClick = (_) => {
    clicks.current = clicks.current + 1;
  };

  <div onClick>
    {Belt.Int.toString(clicks.current)->React.string}
  </div>
}
```

The example above defines a binding `clicks` of type `React.ref(int)`. Note how changing the value `clicks.current` doesn't trigger any re-rendering of the component.

Accessing Refs

When a ref is passed to an element during render, a reference to the node becomes accessible at the current attribute of the ref.

```
let value = myRef.current
```

The value of the ref differs depending on the type of the node: - When the ref attribute is used on an HTML element, the ref passed via `ReactDOM.Ref.domRef` receives the underlying DOM element as its current property (type of `React.ref<Js.Nullable.t<Dom.element>>`) - In case of interop, when the ref attribute is used on a custom class component (based on JS classes), the ref object receives the mounted instance of the component as its current (not discussed in this document). - **You may not use the ref attribute on component functions** because they donâ€™t have instances (we don't expose JS classes in ReScript).

Here are some examples:

Adding a Ref to a DOM Element

This code uses a `React.ref` to store a reference to an `input` DOM node to put focus on a text field when a button was clicked:

```
// CustomTextInput.res

@send external focus: Dom.element => unit = "focus"

@react.component
let make = () => {
  let textInput = React.useRef(Js.Nullable.null)

  let focusInput = () =>
    switch textInput.current->Js.Nullable.toOption {
    | Some(dom) => dom->focus
    | None => ()
    }

  let onClick = _ => focusInput()

  <div>
    <input type_="text" ref={ReactDOM.Ref.domRef(textInput)} />
    <input type_="button" value="Focus the text input" onClick />
  </div>
}

function CustomTextInput(Props) {
  var textInput = React.useRef(null);
  var onClick = function (param) {
    var dom = textInput.current;
    if (!(dom == null)) {
      dom.focus();
      return ;
    }

  };
  return React.createElement("div", undefined, React.createElement("input", {
                  ref: textInput,
                  type: "text"
                }), React.createElement("input", {
                  type: "button",
                  value: "Focus the text input",
                  onClick: onClick
                }));
}
```

A few things happened here, so let's break them down:

- We initialize our `textInput` ref as a `Js.Nullable.null`

- We register our `textInput` ref in our `<input>` element with `ReactDOM.Ref.domRef(textInput)`
- In our `focusInput` function, we need to first verify that our DOM element is set, and then use the `focus` binding to set the focus

React will assign the `current` field with the DOM element when the component mounts, and assign it back to null when it unmounts.

Refs and Component Functions

In React, you **can't** pass a `ref` attribute to a component function:

```
module MyComp = {
  @react.component
  let make = (~ref) => <input />
}

@react.component
let make = () => {
  let textInput = React.useRef(Js.Nullable.null)

  // This will **not** work
  <MyComp ref={ReactDOM.Ref.domRef(textInput)} />
}

// Compiler Error:
// Ref cannot be passed as a normal prop. Please use `forwardRef`
// API instead
```

The snippet above will not compile and output an error that looks like this: `"Ref cannot be passed as a normal prop. Please use forwardRef API instead."`.

As the error message implies, If you want to allow people to take a ref to your component function, you can use ref forwarding (possibly in conjunction with useImperativeHandle) instead.

Exposing DOM Refs to Parent Components

In rare cases, you might want to have access to a childâ€™s DOM node from a parent component. This is generally not recommended because it breaks component encapsulation, but it can occasionally be useful for triggering focus or measuring the size or position of a child DOM node.

we recommend to use ref forwarding for these cases. **Ref forwarding lets components opt into exposing any child componentâ€™s ref as their own**. You can find a detailed example of how to expose a childâ€™s DOM node to a parent component in the ref forwarding documentation.

Callback Refs

React also supports another way to set refs called â€œcallback refsâ€ (`React.Ref.callbackDomRef`), which gives more fine-grain control over when refs are set and unset.

Instead of passing a ref value created by `React.useRef()`, you can pass in a callback function. The function receives the target `Dom.element` as its argument, which can be stored and accessed elsewhere.

Note: Usually we'd use `React.Ref.domRef()` to pass a ref value, but for callback refs, we use `React.Ref.callbackDomRef()` instead.

The example below implements a common pattern: using the ref callback to store a reference to a DOM node in an instance property.

```
// CustomTextInput.res

@send external focus: Dom.element => unit = "focus"

@react.component
let make = () => {
  let textInput = React.useRef(Js.Nullable.null)
  let setTextInputRef = element => {
    textInput.current = element;
  }

  let focusTextInput = _ => {
    textInput.current
    ->Js.Nullable.toOption
    ->Belt.Option.forEach(input => input->focus)
  }

  <div>
    <input type_="text" ref={ReactDOM.Ref.callbackDomRef(setTextInputRef)} />
    <input
      type_="button" value="Focus the text input" onClick={focusTextInput}
    />
  </div>
}
```

```
function CustomTextInput(Props) {
  var textInput = React.useRef(null);
  var setTextInputRef = function (element) {
    textInput.current = element;

  };
  var focusTextInput = function (param) {
    return Belt_Option.forEach(Caml_option.nullable_to_opt(textInput.current), (function (input) {
                  input.focus();

                }));
  };
  return React.createElement("div", undefined, React.createElement("input", {
                  ref: setTextInputRef,
                  type: "text"
                }), React.createElement("input", {
                  type: "button",
                  value: "Focus the text input",
                  onClick: focusTextInput
                }));
}
```

React will call the ref callback with the DOM element when the component mounts, and call it with null when it unmounts.

You can pass callback refs between components like you can with object refs that were created with `React.useRef()`.

```
// Parent.res

@send external focus: Dom.element => unit = "focus"

module CustomTextInput = {
  @react.component
  let make = (~setInputRef) => {
    <div>
      <input type_="text" ref={ReactDOM.Ref.callbackDomRef(setInputRef)} />
    </div>
  }
}

@react.component
let make = () => {
  let textInput = React.useRef(Js.Nullable.null)
  let setInputRef = element => { textInput.current = element}

  <CustomTextInput setInputRef/>
}
```

```
function CustomTextInput(Props) {
  var setInputRef = Props.setInputRef;
  return React.createElement("div", undefined, React.createElement("input", {
                  ref: setInputRef,
                  type: "text"
                }));
}

var CustomTextInput = {
  make: CustomTextInput
};

function Parent(Props) {
  var textInput = React.useRef(null);
  var setInputRef = function (element) {
    textInput.current = element;

  };
  return React.createElement(CustomTextInput, {
              setInputRef: setInputRef
            });
}
```

In the example above, `Parent` passes its ref callback as an `setInputRef` prop to the `CustomTextInput`, and the `CustomTextInput` passes the same function as a special ref attribute to the `<input>`. As a result, the `textInput` ref in Parent will be set to the DOM node corresponding to the `<input>` element in the `CustomTextInput`.

SECTION arrays-and-keys

---

title: Arrays and Keys description: "Rendering arrays and handling keys in ReScript and React" canonical: "/docs/react/latest/arrays-and-keys"

Arrays and Keys

Whenever we are transforming data into an array of elements and put it in our React tree, we need to make sure to give every element an unique identifier to help React distinguish elements for each render. This page will explain the `key` attribute and how to apply it whenever we need to map data to `React.element`s.

Keys & Rendering Arrays

Keys help React identify which elements have been changed, added, or removed throughout each render. Keys should be given to elements inside the array to give the elements a stable identity:

```
let numbers = [1, 2, 3, 4, 5];

let items = Belt.Array.map(numbers, (number) => {
  <li key={Belt.Int.toString(number)}> {React.int(number)} </li>
})
```

The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often you would use IDs from your data as keys:

```
type todo = {id: string, text: string}

let todos = [
  {id: "todo1", text: "Todo 1"},
  {id: "todo2", text: "Todo 2"}
]

let items = Belt.Array.map(todos, todo => {
  <li key={todo.id}> {React.string(todo.text)} </li>
})
```

If you donâ€™t have stable IDs for rendered items, you may use the item index as a key as a last resort:

```res {1..3} let items = Belt.Array.mapWithIndex(todos, (i, todo) => { // Only do this if items have no stable id

- {todo.text}

});

##### Keys Must Only Be Unique Among Siblings

Keys used within arrays should be unique among their siblings. However they donâ€™t need to be globally unique.
We can use the same keys when we produce two different arrays:

```res {6,10,17,18,25,27}
type post = {id: string, title: string, content: string}

module Blog = {
  @react.component
  let make = (~posts: array<post>) => {
    let sidebar =
      <ul>
        {
          Belt.Array.map(posts, (post) => {
            <li key={post.id}>
              {React.string(post.title)}
            </li>
          })->React.array
        }
      </ul>

    let content = Belt.Array.map(posts, (post) => {
        <div key={post.id}>
          <h3>{React.string(post.title)}</h3>
          <p>{React.string(post.content)}</p>
        </div>
    });

      <div>
      {sidebar}
      <hr />
      {React.array(content)}
    </div>
  }
}

let posts = [
  {id: "1", title: "Hello World", content: "Welcome to learning ReScript & React!"},
  {id: "2", title: "Installation", content: "You can install reason-react from npm."}
]

let blog = <Blog posts/>
```

Rendering `list` Values
```

In case you ever want to render a `list` of items, you can do something like this:

```
type todo = {id: string, text: string}

@react.component
let make = () => {
  let todoList = list{
    {id: "todo1", text: "Todo 1"},
    {id: "todo2", text: "Todo 2"},
  }

  let items =
    todoList
    ->Belt.List.toArray
    ->Belt.Array.map(todo => {
      <li key={todo.id}> {React.string(todo.text)} </li>
    })

  <div> {React.array(items)} </div>
}
```

We use `Belt.List.toArray` to convert our list to an array before creating our `array<React.element>`. Please note that using `list` has performance impact due to extra conversion costs.

99% of the time you'll want to use arrays (seamless interop, faster JS code), but in some cases it might make sense to use a `list` to leverage advanced pattern matching features etc.

SECTION beyond-jsx

---

title: Beyond JSX description: "Details on how to use ReScript and React without JSX" canonical: "/docs/react/latest/beyond-jsx"

Beyond JSX

JSX is a syntax sugar that allows us to use React components in an HTML like manner. A component needs to adhere to certain interface conventions, otherwise it can't be used in JSX. This section will go into detail on how the JSX transformation works and what React APIs are used underneath.

Note: This section requires knowledge about the low level apis for creating elements, such as `React.createElement` or `ReactDOM.createDOMElementVariadic`.

> **Note:** This page assumes your `bsconfig.json` to be set to `"jsx": { "version": 4 }` to apply the right JSX transformations.

Component Types

A plain React component is defined as a `('props) => React.element` function. You can also express a component more efficiently with our shorthand type `React.component<'props>`.

Here are some examples on how to define your own component types (often useful when interoping with existing JS code, or passing around components):

```
// Plain function type
type friend = {name: string, online: bool}
type friendComp = friend => React.element

// Equivalent to
// ({padding: string, children: React.element}) => React.element
type props = {padding: string, children: React.element}
type containerComp = React.component<props>
```

The types above are pretty low level (basically the JS representation of a React component), but since ReScript React has its own ways of defining React components in a more language specific way, let's have a closer look on the anatomy of such a construct.

JSX Component Interface

A ReScript React component needs to be a (sub-)module with a `make` function and `props` type to be usable in JSX. To make things easier, we provide a `@react.component` decorator to create those functions for you:

```
module Friend = {
  @react.component
  let make = (~name: string, ~children) => {
    <div>
      {React.string(name)}
      children
    </div>
```

```
  }
}

module Friend = {
  type props<'name, 'children> = {
    name: 'name,
    children: 'children,
  }

  let make = ({name, children, _}: props<string, 'children>) => {
    ReactDOM.createDOMElementVariadic("div", [{React.string(name)}, children])
  }
}
```

In the expanded output:

- `props`: A generated record type that has fields according to the labeled arguments of the `make` function
- `make`: A converted `make` function that complies to the component interface `(props) => React.element`

Special Case React.forwardRef

The `@react.component` decorator also works for `React.forwardRef` calls:

```
module FancyInput = {
  @react.component
  let make = React.forwardRef((~className=?, ~children, ref) =>
    <div>
      // use ref here
    </div>
  )
}

// Simplified Output
type props<'className, 'children, 'ref> = {
  className?: 'className,
  children: 'children,
  ref?: 'ref,
}

let make = (
  {?className, children, _}: props<'className, 'children, ReactRef.currentDomRef>,
  ref: Js.Nullable.t<ReactRef.currentDomRef>,
) =>
  make(~className, ~children, ~ref, ())
```

As shown in the expanded output above, our decorator desugars the function passed to `React.forwardRef` in the same manner as a typical component `make` function. It also creates a `props` type with an optional `ref` field, so we can use it in our JSX call (`<FancyInput ref=.../>`).

So now that we know how the ReScript React component transformation works, let's have a look on how ReScript transforms our JSX constructs.

JSX Under the Hood

Whenever we are using JSX with a custom component ("capitalized JSX"), we are actually using `React.createElement` to create a new element. Here is an example of a React component without children:

```
<Friend name="Fred" age=20 />

// classic
React.createElement(Friend.make, {name: "Fred", age:20})

// automatic
React.jsx(Friend.make, {name: "Fred", age: 20})

React.createElement(Playground$Friend, { name: "Fred", age: 20 });
```

As you can see, it uses `Friend.make` to call the `React.createElement` API. In case you are providing children, it will use `React.createElementVariadic` instead (which is just a different binding for `React.createElement`):

```
<Container width=200>
  {React.string("Hello")}
  {React.string("World")}
</Container>

// classic
React.createElementVariadic(
  Container.make,
  {width: 200, children: React.null},
  [{React.string("Hello")}, {React.string("World")}],
)
```

```
// automatic
React.jsxs(
  Container.make,
  {width: 200, children: React.array([{React.string("Hello")}, {React.string("World")}])},
)

React.createElement(Container, { width: 200, children: null }, "Hello", "World");
```

Note that the `children: React.null` field has no relevance since React will only care about the children array passed as a third argument.

Dom Elements

"Uncapitalized JSX" expressions are treated as DOM elements and will be converted to `ReactDOM.createDOMElementVariadic` calls:

```
<div title="test"/>

// classic
ReactDOM.createDOMElementVariadic("div", ~props={title: "test"}, [])

// automatic
ReactDOM.jsx("div", {title: "test"})

React.createElement("div", { title: "test" });
```

The same goes for uncapitalized JSX with children:

```
<div title="test">
  <span/>
</div>

// classic
ReactDOM.createDOMElementVariadic(
  "div",
  ~props={title: "test"},
  [ReactDOM.createDOMElementVariadic("span", [])],
)

// automatic
ReactDOM.jsx("div", {title: "test", children: ?ReactDOM.someElement(ReactDOM.jsx("span", {}))})

React.createElement("div", { title: "test" }, React.createElement("span", undefined));
```

SECTION elements-and-jsx

---

title: Elements & JSX description: "Basic concepts for React elements and how to use them in JSX" canonical: "/docs/react/latest/elements-and-jsx"

Elements & JSX

Elements are the smallest building blocks of React apps. This page will explain how to handle `React.element`s in your React app with our dedicated JSX syntax.

> **Note:** This page assumes your `bsconfig.json` to be set to `"jsx": { "version": 4 }`, otherwise your JSX will not be transformed to its React specific form.

Element Basics

Let's start out by creating our first React element.

```
let element = <h1> {React.string("Hello World")} </h1>
```

The binding `element` and the expression `{React.string("Hello World")}` are both of type `React.element`, the fundamental type for representing React elements within a React application. An element describes what you see on the screen whenever you render your application to the DOM.

Let's say you want to create a function that handles another React element, such as `children`, you can annotate it as `React.element`:

```
let wrapChildren = (children: React.element) => {
  <div>
    <h1> {React.string("Overview")} </h1>
    children
  </div>
}

wrapChildren(<div> {React.string("Let's use React with ReScript")} </div>)
```

Understanding the definition of a `React.element` is essential since it is heavily used within the React APIs, such as `ReactDOM.Client.Root.render(..., element)`, etc. Be aware that JSX doesn't do any automatic `string` to `React.element` conversion for you (ReScript forces explicit type conversion). For example `<div> Hello World </div>` will not type-check (which is actually a good thing because it's also a huge source for subtle bugs!), you need to convert your `"Hello World"` with the `React.string` function first.

Fortunately our React bindings bring all necessary functionality to represent all relevant data types as `React.element`s.

Using Elements within JSX

You can compose elements into more complex structures by using JSX:

```
let greeting = React.string("Hello ")
let name = React.string("Stranger");


// element is also of type React.element
let element = <div className="myElement"> greeting name </div>
```

JSX is the main way to express your React application as a tree of elements.

Sometimes, when doing a lot of interop with existing ReactJS codebases, you'll find yourself in a situation where you can't use JSX syntax due to syntactic restrictions. Check out the Escape Hatches chapter later on for workarounds.

Creating Elements

Creating Elements from `string`, `int`, `float`, `array`

Apart from using JSX to create our React elements or React components, the `React` module offers various functions to create elements from primitive data types:

```
React.string("Hello") // new element representing "Hello"

React.int(1) // new element representing "1"

React.float(1.0) // new element representing "1.0"
```

It also offers `React.array` to represent multiple elements as one single element (useful for rendering a list of data, or passing children):

```
let element = React.array([
  React.string("element 1"),
  React.string("element 2"),
  React.string("element 3")
])
```

Note: We don't offer a `React.list` function because a `list` value would impose runtime overhead. ReScript cares about clean, idiomatic JS output. If you want to transform a `list` of elements to a single React element, combine the output of `Belt.List.toArray` with `React.array` instead.

Creating Null Elements

ReScript doesn't allow `element || null` constraints due to it's strongly typed nature. Whenever you are expressing conditionals where a value might, or might not be rendered, you will need the `React.null` constant to represent *Nothingness*:

```
let name = Some("Andrea")

let element = switch name {
  | Some(name) => <div> {React.string("Hello " ++ name)} </div>
  | None => React.null
}

<div> element </div>

var name = "Andrea";

var element = name !== undefined ? React.createElement("div", undefined, "Hello " + name) : null;

React.createElement("div", undefined, element);
```

Escape Hatches

Note: This chapter features low level APIs that are used by JSX itself, and should only be used whenever you hit certain JSX syntax limitations. More infos on the JSX internals can be found in our Beyond JSX section.

Creating Elements from Component Functions

Note: Details on components and props will be described in the next chapter.

Sometimes it's necessary to pass around component functions to have more control over `React.element` creation. Use the `React.createElement` function to instantiate your elements:

```
type props = {name: string}

let render = (myComp: props => React.element) => {
  <div> {React.createElement(myComp, {name: "Franz"})} </div>
}
```

This feature is often used when interacting with existing JS / ReactJS code. In pure ReScript React applications, you would rather pass a function that does the rendering for you (also called a "render prop"):

```
let render = (renderMyComp: (~name: string) => React.element) => {
  <div> {renderMyComp(~name="Franz")} </div>
}
```

Pass Variadic Children

There is also a `React.createElementVariadic` function, which takes an array of children as a third parameter:

```
type props = {title: string, children: React.element}

let render = (article: props => React.element) => {
  let children = [React.string("Introduction"), React.string("Body")]

  let props = {title: "Article #1", children: React.null}

  {React.createElementVariadic(article, props, children)}
}
```

```
function render(article) {
  var children = [
    "Introduction",
    "Body"
  ];
  var props = {
    title: "Article #1",
    children: null
  };
  return Caml_splice_call.spliceApply(React.createElement, [
              article,
              props,
              children
            ]);
}
```

Note: Here we are passing a prop `"children": React.null` to satisfy the type checker. React will ignore the children prop in favor of the children array.

This function is mostly used by our JSX transformations, so usually you want to use `React.createElement` and pass a children prop instead.

Creating DOM Elements

To create DOM elements (`<div>`, `<span>`, etc.), use `ReactDOM.createDOMElementVariadic`:

```
ReactDOM.createDOMElementVariadic("div", ~props={className: "card"}, [])
```

ReScript can make sure that we are only passing valid dom props. You can find an exhaustive list of all available props in the JsxDOM module.

Cloning Elements

Note: This is an escape hatch feature and will only be useful for interoping with existing JS code / libraries.

Sometimes it's required to clone an existing element to set, overwrite or add prop values to a new instance, or if you want to set invalid prop names such as `data-name`. You can use `React.cloneElement` for that:

```
let original = <div className="hello"/>

// Will return a new React.element with className set to "world"
React.cloneElement(original, {"className": "world", "data-name": "some name"});
```

```
var original = React.createElement("div", {
      className: "hello"
    });

React.cloneElement(original, {
      className: "world",
      "data-name": "some name"
```

```
  });
```

The feature mentioned above could also replicate `props spreading`, a practise commonly used in ReactJS codebases, but we strongly discourage the usage due to its unsafe nature and its incorrectness (e.g. adding undefined extra props to a component doesn't make sense, and causes hard to find bugs).

In ReScript, we rather pass down required props explicitly to leaf components or use a renderProp instead. We introduced JSX punning syntax to make the process of passing down props more convenient.

SECTION migrate-react

---

title: Migrate from JSX v3 description: "Migrate from JSX v3" canonical: "/docs/react/latest/migrate-react"

Migrate from JSX v3

JSX v4 introduces a new idiomatic record-based representation of components which is incompatible with v3. Because of this, either the entire project or dependencies need to be compiled in V4 mode, or some compatibility features need to be used to mix V3 and V4 in the same project. This page describes how to migrate from v3 to v4.

Configuration

Remove the existing JSX configuration from `bsconfig.json`:

```
{
  "reason": { "react-jsx": 3 }
}
```

Then add the new JSX configuration:

```
{
  "jsx": { "version": 4 }
}
```

Note JSX v4 requires the rescript compiler 10.1 or higher, and rescript-react version 0.11 or higher. In addition, react version 18.0 is required.

Classic and Automatic Mode

Classic mode is the default and generates calls to `React.createElement` just as with V3.

```
{
  "jsx": { "version": 4, "mode": "classic" }
}
```

Automatic mode is an experimental mode that generate calls to `_jsx` functions (similar to TypeScript's `react-jsx` mode)

```
{
  "jsx": { "version": 4, "mode": "automatic" }
}
```

File-level config

The top-level attribute `@@jsxConfig` is used to update the `jsx` config for the rest of the file (or until the next config update). Only the values mentioned are updated, the others are left unchanged.

```
@@jsxConfig({ version: 4, mode: "automatic" })

module Wrapper = {
  module R1 = {
    @react.component  // V4 and new _jsx transform
    let make = () => body
  }

  @@jsxConfig({ version: 4, mode: "classic" })

  module R2 = {
    @react.component  // V4 with `React.createElement`
    let make = () => body
  }
}

@@jsxConfig({ version: 3 })

@react.component  // V3
let make = () => body
```

v3 compatible mode

JSX v3 is still available with the latest version of compiler and rescript-react.

```
{
  "jsx": { "version": 3, "v3-dependencies": ["rescript-relay"] },
  "bsc-flags": ["-open ReactV3"]
}
```

To build certain dependencies in V3 compatibility mode, whatever the version used in the root project, use `"v3-dependencies"`. The listed dependencies will be built in V3 mode, and in addition `-open ReactV3` is added to the compiler options.

Migration of V3 components

Some components in existing projects are written in a way that is dependent on the v3 internal representation. Here are a few examples of how to convert them to v4.

`makeProps` does not exist in v4

Rewrite this:

```
// V3
module M = {
  @obj external makeProps: (~msg: 'msg, ~key: string=?, unit) => {"msg": 'msg} = ""

  let make = (~msg) => <div> {React.string(msg)} </div>
}
```

To this:

```
// V4
module M = {
  type props<'msg> = {msg: 'msg}
  let make = props => <div> {React.string(props.msg)} </div>
}
```

React.Context

Rewrite this:

```
module Context = {
  let context = React.createContext(() => ())

  module Provider = {
    let provider = React.Context.provider(context)

    @react.component
    let make = (~value, ~children) => {
      React.createElement(provider, {"value": value, "children": children}) // Error
    }
  }
}
```

To this:

```
module Context = {
  let context = React.createContext(() => ())

  module Provider = {
    let make = React.Context.provider(context)
  }
}
```

React.forwardRef (Discouraged)

Rewrite this:

```
module FancyInput = {
  @react.component
  let make = React.forwardRef((
    ~className=?,
    ~children,
    ref_, // argument
  ) =>
    <div>
      <input
        type_="text"
        ?className
        ref=?{ref_->Js.Nullable.toOption->Belt.Option.map(ReactDOM.Ref.domRef)}
      />
```

```
      children
    </div>
  )
}

@react.component
let make = () => {
  let input = React.useRef(Js.Nullable.null)

  <div>
    <FancyInput ref=input> // prop
      <button onClick> {React.string("Click to focus")} </button>
    </FancyInput>
  </div>
}
```

To this: In v3, there is an inconsistency between `ref` as prop and `ref_` as argument. With JSX V4, `ref` is only allowed as an argument.

```
module FancyInput = {
  @react.component
  let make = React.forwardRef((
    ~className=?,
    ~children,
    ref, // only `ref` is allowed
  ) =>
    <div>
      <input
        type_="text"
        ?className
        ref=?{ref->Js.Nullable.toOption->Belt.Option.map(ReactDOM.Ref.domRef)}
      />
      children
    </div>
  )
}

@react.component
let make = () => {
  let input = React.useRef(Js.Nullable.null)

  <div>
    <FancyInput ref=input>
      <button onClick> {React.string("Click to focus")} </button>
    </FancyInput>
  </div>
}
```

Mangling the prop name

The prop name was mangled automatically in v3, such as `_open` becomes `open` in the generated js code. This is no longer the case in v4 because the internal representation is changed to the record instead object. If you need to mangle the prop name, you can use the `@as` annotation.

Rewrite this:

```
module Comp = {
  @react.component
  let make = (~_open, ~_type) =>
    <Modal _open _type>
      <Description />
    </Modal>
}
```

To this:

```
module Comp = {
  @react.component
  let make =
    (@as("open") ~_open, @as("type") ~_type) =>
      <Modal _open _type>
        <Description />
      </Modal>
}
```

Bindings to JS components with optional props

Previously, you could wrap optional props with an explicit `option` when writing bindings to JS components. This approach functioned only due to an implementation detail of the ppx in JSX 3; it's not how to correctly write bindings to a function with optional arguments.

Rewrite this:

```
module Button = {
  @module("./Button") @react.component
```

```
  external make: (~text: option<string>=?) => React.element = "default"
}
```

To this:

```
module Button = {
  @module("./Button") @react.component
  external make: (~text: string=?) => React.element = "default"
}
```

SECTION hooks-context

---

title: useContext Hook description: "Details about the useContext React hook in ReScript" canonical: "/docs/react/latest/hooks-context"

---

useContext

Context provides a way to pass data through the component tree without having to pass props down manually at every level. The `useContext` hooks gives access to such Context values.

> **Note:** All the details and rationale on React's context feature can be found in here.

Usage

```
let value = React.useContext(myContext)
```

Accepts a `React.Context.t` (the value returned from `React.createContext`) and returns the current context value for that context. The current context value is determined by the value prop of the nearest `<MyContext.Provider>` above the calling component in the tree.

Examples

A Simple ThemeProvider

```
// App.res
module ThemeContext = {
  let context = React.createContext("light")

  module Provider = {
    let make = React.Context.provider(context)
  }
}

module ThemedButton = {
  @react.component
  let make = () => {
    let theme = React.useContext(ThemeContext.context)
    let (color, backgroundColor) = switch theme {
    | "dark" => ("#ffffff", "#222222")
    | "light" | _ => ("#000000", "#eeeeee")
    }

    let style = ReactDOMStyle.make(~color, ~backgroundColor, ())

    <button style> {React.string("I am a styled button!")} </button>
  }
}

module Toolbar = {
  @react.component
  let make = () => {
    <div> <ThemedButton /> </div>
  }
}

@react.component
let make = () => {
  <ThemeContext.Provider value="dark">
    <div> <Toolbar /> </div>
  </ThemeContext.Provider>
}

var context = React.createContext("light");

var make = context.Provider;

var Provider = {
  make: make
};
```

```
var ThemeContext = {
  context: context,
  Provider: Provider
};

function App$ThemedButton(props) {
  var theme = React.useContext(context);
  var match;
  switch (theme) {
    case "dark" :
        match = [
          "#ffffff",
          "#222222"
        ];
        break;
    case "light" :
        match = [
          "#000000",
          "#eeeeee"
        ];
        break;
    default:
      match = [
        "#000000",
        "#eeeeee"
      ];
  }
  var style = {
    backgroundColor: match[1],
    color: match[0]
  };
  return React.createElement("button", {
              style: style
            }, "I am a styled button!");
}

var ThemedButton = {
  make: App$ThemedButton
};

function App$Toolbar(props) {
  return React.createElement("div", undefined, React.createElement(App$ThemedButton, {}));
}

var Toolbar = {
  make: App$Toolbar
};

function App(props) {
  return React.createElement(make, {
              value: "dark",
              children: React.createElement("div", undefined, React.createElement(App$Toolbar, {}))
            });
}
```

SECTION hooks-state

---

title: useState Hook description: "Details about the useState React hook in ReScript" canonical: "/docs/react/latest/hooks-state"

---

useState

`React.useState` returns a stateful value, and a function to update it.

Usage

```
let (state, setState) = React.useState(_ => initialState)

var match = React.useState(function () {
      return initialState;
    });

var state = match[0];

var setState = match[1];
```

During the initial render, the returned state `state` is the same as the value passed as the first argument (initialState).

The `setState` function can be passed down to other components as well, which is useful for e.g. setting the state of a parent component by its children.

Examples

Using State for a Text Input

```
@react.component
let make = () => {
  let (text, setText) = React.useState(_ => "");

  let onChange = evt => {
    ReactEvent.Form.preventDefault(evt)
    let value = ReactEvent.Form.target(evt)["value"]
    setText(_prev => value);
  }

  <div>
    <input onChange value=text />
  </div>
};
```

Passing `setState` to a Child Component

In this example, we are creating a `ThemeContainer` component that manages a `darkmode` boolean state and passes the `setDarkmode` function to a `ControlPanel` component to trigger the state changes.

```
// ThemeContainer.res
module ControlPanel = {
  @react.component
  let make = (~setDarkmode, ~darkmode) => {
    let onClick = evt => {
      ReactEvent.Mouse.preventDefault(evt)
      setDarkmode(prev => !prev)
    }

    let toggleText = "Switch to " ++ ((darkmode ? "light" : "dark") ++ " theme")

    <div> <button onClick> {React.string(toggleText)} </button> </div>
  }
}

@react.component
let make = (~content) => {
  let (darkmode, setDarkmode) = React.useState(_ => false)

  let className = darkmode ? "theme-dark" : "theme-light"

  <div className>
    <section>
      <h1> {React.string("More Infos about ReScript")} </h1> content
    </section>
    <ControlPanel darkmode setDarkmode />
  </div>
}
```

```
function ControlPanel(Props) {
  var setDarkmode = Props.setDarkmode;
  var darkmode = Props.darkmode;
  var onClick = function (evt) {
    evt.preventDefault();
    return Curry._1(setDarkmode, (function (prev) {
                  return !prev;
                }));
  };
  var toggleText = "Switch to " + ((
      darkmode ? "light" : "dark"
    ) + " theme");
  return React.createElement("div", undefined, React.createElement("button", {
                  onClick: onClick
                }, toggleText));
}

function ThemeContainer(Props) {
  var content = Props.content;
  var match = React.useState(function () {
        return false;
      });
  var darkmode = match[0];
  var className = darkmode ? "theme-dark" : "theme-light";
  return React.createElement("div", {
              className: className
            }, React.createElement("section", undefined, React.createElement("h1", undefined, "More Infos about
ReScript"), content), React.createElement(Playground$ControlPanel, {
                  setDarkmode: match[1],
                  darkmode: darkmode
                }));
```

```
}
```

Note that whenever `setDarkmode` is returning a new value (e.g. switching from `true` -> `false`), it will cause a re-render for `ThemeContainer`'s `className` and the toggle text of its nested `ControlPanel`.

Uncurried Version

For cleaner JS output, you can use `React.Uncurried.useState` instead:

```
let (state, setState) = React.Uncurried.useState(_ => 0)

setState(. prev => prev + 1)
```

```
var match = React.useState(function () {
      return 0;
  });

var setState = match[1];

setState(function (prev) {
      return prev + 1 | 0;
    });
```

SECTION extensions-of-props

---

title: Extensions of props description: "Extensions of props in ReScript and React" canonical: "/docs/react/latest/spread-props"

Extensions of Props

> **Note:** This page assumes your `bsconfig.json` to be set to `"jsx": { "version": 4 }` to apply the right JSX transformations.

Spread props

JSX props spread is supported now, but in a stricter way than in JS.

```
<Comp {...props} a="a" />

React.createElement(Comp, {
  a: "a",
  b: "b"
});
```

Multiple spreads are not allowed:

```
<NotAllowed {...props1} {...props2} />
```

The spread must be at the first position, followed by other props:

```
<NotAllowed a="a" {...props} />
```

Shared props

You can control the definition of the `props` type by passing as argument to `@react.component` the body of the type definition of `props`. The main application is sharing a single type definition across several components. Here are a few examples:

```
type sharedprops<'x, 'y> = {x: 'x, y: 'y, z:string}

module C1 = {
  @react.component(:sharedProps<'a, 'b>)
  let make = (~x, ~y) => React.string(x ++ y ++ z)
}

module C2 = {
  @react.component(:sharedProps<string, 'b>)
  let make = (~x, ~y) => React.string(x ++ y ++ z)
}

module C3 = {
  type myProps = sharedProps<int, int>
  @react.component(:myProps)
  let make = (~x, ~y) => React.int(x + y)
}

type sharedprops<'x, 'y> = {x: 'x, y: 'y, z: string}

module C1 = {
```

```
  type props<'a, 'b> = sharedProps<'a, 'b>
  let make = ({x, y, _}: props<_>) => React.string(x ++ y ++ z)
}

module C2 = {
  type props<'b> = sharedProps<string, 'b>
  let make = ({x, y, _}: props<_>) => React.string(x ++ y ++ z)
}

module C3 = {
  type myProps = sharedProps<int, int>
  type props = myProps
  let make = ({x, y, _}: props) => React.int(x + y)
}
```

This feature can be used when the nominally different components are passed to the prop of `Screen` component in ReScript React Native Navigation.

```
type screenProps = { navigation: navigation, route: route }

module Screen: {
  @react.component
  let make: (
    ~name: string,
    ~component: React.component<screenProps>,
    ...
  ) => React.element
}

<Screen
  name="SomeScreen"
  component={A.make} // This will cause a type check error
  ...
/>
<Screen
  name="SomeScreen"
  component={B.make} // This will cause a type check error
  ...
/>
```

This example cann't pass the type checker, because the props types of both components are nominally different. You can make the both components having the same props type by passing `screenProps` type as argument to `@react.component`.

```
module A = {
  @react.component(:screenProps)
  let make = (
    ~navigation: navigation,
    ~route: route
  ) => ...
}

module B = {
  @react.component(:screenProps)
  let make = (
    ~navigation: navigation,
    ~route: route
  ) => ...
}
```

SECTION introduction

---

title: Introduction description: "Introduction to ReScript & ReactJS" canonical: "/docs/react/latest/introduction"

ReScript & React

ReScript offers first class bindings for ReactJS and is designed and built by people using ReScript and React in large mission critical React codebases. The bindings are compatible with modern React versions (>= v18.0).

The ReScript philosophy is to compile as closely to idiomatic JS code as possible; in the case of ReactJS, we made no exception, so it's not only easy to transfer all the React knowledge to the ReScript platform, but also straightforward to integrate with existing ReactJS codebases and libraries.

All our documented examples can be compiled in our ReScript Playground as well.

Feature Overview

- No Babel plugins required (JSX is part of the language!)
- Comes with all essential React APIs for building production ready apps (`useState`, `useReducer`, `useEffect`, `useRef`,...)

- No component class API (all ReScript & React codebases are built on function components & hooks)
- Strong level of type safetiness and type inference for component props and state values
- GenType support for importing / exporting React components in TypeScript codebases

This documentation assumes basic knowledge about ReactJS.

Please note that even though we will cover many basic React concepts, it might still be necessary to have a look at the official ReactJS resources, especially if you are a complete beginner with React.

Development

- In case you are having any issues or if you want to help us out improving our bindings, check out our rescript-react GitHub repository.
- For doc related issues, please go to the rescript-lang.org repo.

SECTION hooks-reducer

---

title: useReducer Hook description: "Details about the useReducer React hook in ReScript" canonical: "/docs/react/latest/hooks-reducer"

---

useReducer

`React.useReducer` helps you express your state in an action / reducer pattern.

Usage

```
let (state, dispatch) = React.useReducer(reducer, initialState)
```

```
var match = React.useReducer(reducer, initialState);
```

An alternative to useState. Accepts a reducer of type `(state, action) => newState`, and returns the current `state` paired with a `dispatch` function `(action) => unit`.

`React.useReducer` is usually preferable to `useState` when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. `useReducer` also lets you optimize performance for components that trigger deep updates because you can pass dispatch down instead of callbacks.

Note: You will notice that the action / reducer pattern works especially well in ReScript due to its immutable records, variants and pattern matching features for easy expression of your action and state transitions.

Examples

Counter Example with `React.useReducer`

```
// Counter.res

type action = Inc | Dec
type state = {count: int}

let reducer = (state, action) => {
  switch action {
  | Inc => {count: state.count + 1}
  | Dec => {count: state.count - 1}
  }
}

@react.component
let make = () => {
  let (state, dispatch) = React.useReducer(reducer, {count: 0})

  <>
    {React.string("Count:" ++ Belt.Int.toString(state.count))}
    <button onClick={(_) => dispatch(Dec)}> {React.string("-")} </button>
    <button onClick={(_) => dispatch(Inc)}> {React.string("+")} </button>
  </>
}

function reducer(state, action) {
  if (action) {
    return {
            count: state.count - 1 | 0
          };
  } else {
    return {
            count: state.count + 1 | 0
          };
```

```
  }
}

function Counter(Props) {
  var match = React.useReducer(reducer, {
        count: 0
      });
  var dispatch = match[1];
  return React.createElement(React.Fragment, undefined, "Count:" + String(match[0].count), React.createElement("
button", {
                  onClick: (function (param) {
                      return Curry._1(dispatch, /* Dec */1);
                    })
                }, "-"), React.createElement("button", {
                  onClick: (function (param) {
                      return Curry._1(dispatch, /* Inc */0);
                    })
                }, "+"));
}
```

> React guarantees that dispatch function identity is stable and wonâ€™t change on re-renders. This is why itâ€™s safe to omit from the useEffect or useCallback dependency list.

Basic Todo List App with More Complex Actions

You can leverage the full power of variants to express actions with data payloads to parametrize your state transitions:

```
// TodoApp.res

type todo = {
  id: int,
  content: string,
  completed: bool,
}

type action =
  | AddTodo(string)
  | RemoveTodo(int)
  | ToggleTodo(int)

type state = {
  todos: array<todo>,
  nextId: int,
}

let reducer = (state, action) =>
  switch action {
  | AddTodo(content) =>
    let todos = Js.Array2.concat(
      state.todos,
      [{id: state.nextId, content: content, completed: false}],
    )
    {todos: todos, nextId: state.nextId + 1}
  | RemoveTodo(id) =>
    let todos = Js.Array2.filter(state.todos, todo => todo.id !== id)
    {...state, todos: todos}
  | ToggleTodo(id) =>
    let todos = Belt.Array.map(state.todos, todo =>
      if todo.id === id {
        {
          ...todo,
          completed: !todo.completed,
        }
      } else {
        todo
      }
    )
    {...state, todos: todos}
  }

let initialTodos = [{id: 1, content: "Try ReScript & React", completed: false}]

@react.component
let make = () => {
  let (state, dispatch) = React.useReducer(
    reducer,
    {todos: initialTodos, nextId: 2},
  )

  let todos = Belt.Array.map(state.todos, todo =>
    <li>
      {React.string(todo.content)}
      <button onClick={_ => dispatch(RemoveTodo(todo.id))}>
        {React.string("Remove")}
      </button>
```

```
      <input
        type_="checkbox"
        checked=todo.completed
        onChange={_ => dispatch(ToggleTodo(todo.id))}
      />
    </li>
  )

  <> <h1> {React.string("Todo List:")} </h1> <ul> {React.array(todos)} </ul> </>
}

function reducer(state, action) {
  switch (action.TAG | 0) {
    case /* AddTodo */0 :
        var todos = state.todos.concat([{
                id: state.nextId,
                content: action._0,
                completed: false
              }]);
        return {
                todos: todos,
                nextId: state.nextId + 1 | 0
              };
    case /* RemoveTodo */1 :
        var id = action._0;
        var todos$1 = state.todos.filter(function (todo) {
              return todo.id !== id;
            });
        return {
                todos: todos$1,
                nextId: state.nextId
              };
    case /* ToggleTodo */2 :
        var id$1 = action._0;
        var todos$2 = Belt_Array.map(state.todos, (function (todo) {
                if (todo.id === id$1) {
                  return {
                          id: todo.id,
                          content: todo.content,
                          completed: !todo.completed
                        };
                } else {
                  return todo;
                }
              }));
        return {
                todos: todos$2,
                nextId: state.nextId
              };

  }
}

var initialTodos = [{
    id: 1,
    content: "Try ReScript & React",
    completed: false
  }];

function TodoApp(Props) {
  var match = React.useReducer(reducer, {
        todos: initialTodos,
        nextId: 2
      });
  var dispatch = match[1];
  var todos = Belt_Array.map(match[0].todos, (function (todo) {
          return React.createElement("li", undefined, todo.content, React.createElement("button", {
                        onClick: (function (param) {
                            return Curry._1(dispatch, {
                                        TAG: /* RemoveTodo */1,
                                        _0: todo.id
                                      });
                          })
                      }, "Remove"), React.createElement("input", {
                        checked: todo.completed,
                        type: "checkbox",
                        onChange: (function (param) {
                            return Curry._1(dispatch, {
                                        TAG: /* ToggleTodo */2,
                                        _0: todo.id
                                      });
                          })
                      }));
        }));
  return React.createElement(React.Fragment, undefined, React.createElement("h1", undefined, "Todo List:"), Reac
t.createElement("ul", undefined, todos));
}
```

Lazy Initialization

```
let (state, dispatch) =
  React.useReducerWithMapState(reducer, initialState, initial)

var match = React.useReducer(reducer, initialState, init);
```

You can also create the `initialState` lazily. To do this, you can use `React.useReducerWithMapState` and pass an `init` function as the third argument. The initial state will be set to `init(initialState)`.

It lets you extract the logic for calculating the initial state outside the reducer. This is also handy for resetting the state later in response to an action:

```
// Counter.res

type action = Inc | Dec | Reset(int)
type state = {count: int}

let init = initialCount => {
  {count: initialCount}
}

let reducer = (state, action) => {
  switch action {
  | Inc => {count: state.count + 1}
  | Dec => {count: state.count - 1}
  | Reset(count) => init(count)
  }
}

@react.component
let make = (~initialCount: int) => {
  let (state, dispatch) = React.useReducerWithMapState(
    reducer,
    initialCount,
    init,
  )

  <>
    {React.string("Count:" ++ Belt.Int.toString(state.count))}
    <button onClick={_ => dispatch(Dec)}> {React.string("-")} </button>
    <button onClick={_ => dispatch(Inc)}> {React.string("+")} </button>
  </>
}

function init(initialCount) {
  return {
          count: initialCount
        };
}

function reducer(state, action) {
  if (typeof action === "number") {
    if (action !== 0) {
      return {
              count: state.count - 1 | 0
            };
    } else {
      return {
              count: state.count + 1 | 0
            };
    }
  } else {
    return {
            count: action._0
          };
  }
}

function Counter(Props) {
  var initialCount = Props.initialCount;
  var match = React.useReducer(reducer, initialCount, init);
  var dispatch = match[1];
  return React.createElement(React.Fragment, undefined, "Count:" + String(match[0].count), React.createElement("
button", {
                  onClick: (function (param) {
                      return Curry._1(dispatch, /* Dec */1);
                    })
                }, "-"), React.createElement("button", {
                  onClick: (function (param) {
                      return Curry._1(dispatch, /* Inc */0);
                    })
                }, "+"));
}
```

SECTION styling

---

title: Styling description: "Styling in ReScript & React" canonical: "/docs/react/latest/styling"

Styling

React comes with builtin support for inline styles, but there are also a number of third party libraries for styling React components. You might be comfortable with a specific setup, like:

- Global CSS / CSS modules
- CSS utility libraries (`tailwindcss`, `tachyons`, `bootstrap` etc.)
- CSS-in-JS (`styled-components`, `emotion`, etc.)

If they work in JS then they almost certainly work in ReScript. In the next few sections, we've shared some ideas for working with popular libraries. If you're interested in working with one you don't see here, search the package index or post in the forum.

Inline Styles

This is the most basic form of styling, coming straight from the 90s. You can apply a `style` attribute to any DOM element with our `ReactDOM.Style.make` API:

```
<div style={ReactDOM.Style.make(~color="#444444", ~fontSize="68px", ())} />
```

It's a labeled (therefore typed) function call that maps to the familiar style object `{color: '#444444', fontSize: '68px'}`. For every CSS attribute in the CSS specfication, there is a camelCased label in our `make` function.

Note that `make` returns an opaque `ReactDOM.Style.t` type that you can't read into. We also expose a `ReactDOM.Style.combine` that takes in two `style`s and combine them.

Escape Hatch: `unsafeAddProp`

The above `Style.make` API will safely type check every style field! However, we might have missed some more esoteric fields. If that's the case, the type system will tell you that the field you're trying to add doesn't exist. To remediate this, we're exposing a `ReactDOM.Style.unsafeAddProp` to dangerously add a field to a style:

```
let style =
  ReactDOM.Style.make(
    ~color="red",
    ~padding="10px",
    (),
  )->ReactDOM.Style.unsafeAddProp("-webkit-animation-name", "moveit")
```

Global CSS

Use a `%%raw` expression to import CSS files within your ReScript / React component code:

```
// in a CommonJS setup
%%raw("require('./styles/main.css')")

// or with ES6
%%raw("import './styles/main.css'")
```

CSS Modules

CSS modules can be imported like any other JS module. The imported value is a JS object, with attributes equivalent to each classname defined in the CSS file.

As an example, let's say we have a CSS module like this:

```
/* styles.module.css */

.root {
  color: red
}
```

We now need to create a module binding that imports our styles as a JS object:

```
// {..} means we are handling a JS object with an unknown
// set of attributes
@module external styles: {..} = "./styles.module.css"

// Use the obj["key"] syntax to access any classname within our object
let app = <div className={styles["root"]} />
```

Note: `{..}` is an open JS object type, which means the type checker will not type check correct classname usage. If you want to enforce compiler errors, replace `{..}` with a concrete JS object type, such as `{"root": string}`.

CSS Utility Libraries

Tailwind

CSS utility libraries like TailwindCSS usually require some globally imported CSS.

First, create your TailwindCSS main entrypoint file:

```
/* main.css */

@tailwind base;
@tailwind components;
@tailwind utilities;
```

Then, import your `main.css` file in your ReScript / React application:

```
// src/App.res

%%raw("import './main.css'")
```

Utilize ReScript's pattern matching and string interpolations to combine different classnames:

```
@react.component
let make = (~active: bool) => {
  let activeClass = if active {
    "text-green-600"
  }
  else {
    "text-red-600"
  }

  <div className={`border-1 border-black ${activeClass}`}>
    {React.string("Hello World")}
  </div>
}
```

> **Hint:** `rescript-lang.org` actually uses TailwindCSS under the hood! Check out our codebase to get some more inspiration on usage patterns.

CSS-in-JS

There's no way we could recommend a definitive CSS-in-JS workflow, since there are many different approaches on how to bind to CSS-in-JS libraries (going from simple to very advanced).

For demonstration purposes, let's create some simple bindings to e.g. `emotion` (as described here):

```
// src/Emotion.res

@module("@emotion/css") external css: {..} => string = "css"
@module("@emotion/css") external rawCss: string => string = "css"
@module("@emotion/css") external keyframes: {..} => string = "css"
@module("@emotion/css") external cx: array<string> => string = "cx"

@module("@emotion/css") external injectGlobal: string => unit = "injectGlobal"
```

This will give you straight-forward access to `emotion`'s apis. Here's how you'd use them in your app code:

```
let container = Emotion.css({
  "color": "#fff",
  "backgroundColor": "red"
})

let app = <div className={container} />
```

You can also use submodules to organize your styles more easily:

```
module Styles = {
  open Emotion
  let container = css({
    "color": "#fff",
    "backgroundColor": "red"
  })
  // your other declarations
}

let app = <div className={Styles.container} />
```

Please note that this approach will not check for invalid css attribute names. If you e.g. want to make sure that only valid CSS attributes

are being passed, you could define your `css` function like this as well:

```
@module("@emotion/css") external css: ReactDOM.Style.t => string = "css"

// Usage is slightly different (and probably less ergonomic)
let container = ReactDOM.Style.make(~padding="20px", ())->css;

let app = <div
  className={container}
/>
```

Here we used the already existing `React.Style.t` type to enforce valid CSS attribute names. Last but not least, you can also bind to functions that let you use raw CSS directly:

```
let container = Emotion.rawCss(`
  color: #fff;
  background-color: red;
`)

let app = <div className={container} />
```

Please keep in mind that there's a spectrum on how type-safe an API can be (while being more / less complex to handle), so choose a solution that fits to your team's needs.

SECTION hooks-custom

---

title: Build A Custom Hook description: "How to build your own hooks in ReScript & React" canonical: "/docs/react/latest/hooks-custom"

Build A Custom Hook

React comes with a few fundamental hooks out-of-the-box, such as `React.useState` or `React.useEffect`. Here you will learn how to build your own, higher-level hooks for your React use-cases.

Why Custom Hooks?

Custom hooks let you extract existing component logic into reusable, separate functions.

Let's go back to a previous example from our React.useEffect section where we built a component for a chat application that displays a message, indicating whether a friend is online or offline:

```res {16-31} // FriendStatus.res

module ChatAPI = { // Imaginary globally available ChatAPI for demo purposes type status = { isOnline: bool }; @val external subscribeToFriendStatus: (string, status => unit) => unit = "subscribeToFriendStatus"; @val external unsubscribeFromFriendStatus: (string, status => unit) => unit = "unsubscribeFromFriendStatus"; }

type state = Offline | Loading | Online;

@react.component let make = (~friendId: string) => { let (state, setState) = React.useState(_ => Offline)

React.useEffect(() => { let handleStatusChange = (status) => { setState(_ => { status.ChatAPI.isOnline ? Online : Offline }) }

ChatAPI.subscribeToFriendStatus(friendId, handleStatusChange);
setState(_ => Loading);

let cleanup = () => {
  ChatAPI.unsubscribeFromFriendStatus(friendId, handleStatusChange)
}

Some(cleanup)

})

let text = switch(state) { | Offline => friendId ++ " is offline" | Online => friendId ++ " is online" | Loading => "loading..." }

{React.string(text)}

}

```js
function FriendStatus(Props) {
  var friendId = Props.friendId;
  var match = React.useState(function () {
        return /* Offline */0;
```
```

```
      });
  var setState = match[1];
  React.useEffect(function () {
        var handleStatusChange = function (status) {
          return Curry._1(setState, (function (param) {
                        if (status.isOnline) {
                          return /* Online */2;
                        } else {
                          return /* Offline */0;
                        }
                      }));
        };
        subscribeToFriendStatus(friendId, handleStatusChange);
        Curry._1(setState, (function (param) {
                return /* Loading */1;
              }));
        return (function (param) {
                  unsubscribeFromFriendStatus(friendId, handleStatusChange);

                });
      });
  var text;
  switch (match[0]) {
    case /* Offline */0 :
        text = friendId + " is offline";
        break;
    case /* Loading */1 :
        text = "loading...";
        break;
    case /* Online */2 :
        text = friendId + " is online";
        break;

  }
  return React.createElement("div", undefined, text);
}
```

Now letâ€™s say that our chat application also has a contact list, and we want to render names of online users with a green color. We could copy and paste similar logic above into our `FriendListItem` component but it wouldnâ€™t be ideal:

```res {15-30} // FriendListItem.res type state = Offline | Loading | Online;

// module ChatAPI = {...}

type friend = { id: string, name: string };

@react.component let make = (~friend: friend) => { let (state, setState) = React.useState(_ => Offline)

React.useEffect(() => { let handleStatusChange = (status) => { setState(_ => { status.ChatAPI.isOnline ? Online : Offline }) }

ChatAPI.subscribeToFriendStatus(friend.id, handleStatusChange);
setState(_ => Loading);

let cleanup = () => {
  ChatAPI.unsubscribeFromFriendStatus(friend.id, handleStatusChange)
}

Some(cleanup)

})

let color = switch(state) { | Offline => "red" | Online => "green" | Loading => "grey" }

- {React.string(friend.name)}

}
```

```js
function FriendListItem(Props) {
  var friend = Props.friend;
  var match = React.useState(function () {
        return /* Offline */0;
      });
  var setState = match[1];
  React.useEffect(function () {
        var handleStatusChange = function (status) {
          return Curry._1(setState, (function (param) {
                        if (status.isOnline) {
                          return /* Online */2;
                        } else {
                          return /* Offline */0;
                        }
                      }));
        };
```

```
        subscribeToFriendStatus(friend.id, handleStatusChange);
        Curry._1(setState, (function (param) {
                return /* Loading */1;
              }));
        return (function (param) {
                  unsubscribeFromFriendStatus(friend.id, handleStatusChange);

                });
      });
  var color;
  switch (match[0]) {
    case /* Offline */0 :
        color = "red";
        break;
    case /* Loading */1 :
        color = "grey";
        break;
    case /* Online */2 :
        color = "green";
        break;

  }
  return React.createElement("li", {
              style: {
                color: color
              }
            }, friend.name);
}
```

Instead, weâ€™d like to share this logic between `FriendStatus` and `FriendListItem`.

Traditionally in React, weâ€™ve had two popular ways to share stateful logic between components: render props and higher-order components. We will now look at how Hooks solve many of the same problems without forcing you to add more components to the tree.

Extracting a Custom Hook

Usually when we want to share logic between two functions, we extract it to a third function. Both components and Hooks are functions, so this works for them too!

A custom Hook is a function whose name starts with â€useâ€ and that may call other Hooks. For example, `useFriendStatus` below is our first custom Hook (we create a new file `FriendStatusHook.res` to encapsulate the `state` type as well):

```
// FriendStatusHook.res

// module ChatAPI {...}

type state = Offline | Loading | Online

let useFriendStatus = (friendId: string): state => {
  let (state, setState) = React.useState(_ => Offline)

  React.useEffect(() => {
    let handleStatusChange = status => {
      setState(_ => {
        status.ChatAPI.isOnline ? Online : Offline
      })
    }

    ChatAPI.subscribeToFriendStatus(friendId, handleStatusChange)
    setState(_ => Loading)

    let cleanup = () => {
      ChatAPI.unsubscribeFromFriendStatus(friendId, handleStatusChange)
    }

    Some(cleanup)
  })

  state
}

function useFriendStatus(friendId) {
  var match = React.useState(function () {
        return /* Offline */0;
      });
  var setState = match[1];
  React.useEffect(function () {
        var handleStatusChange = function (status) {
          return Curry._1(setState, (function (param) {
                        if (status.isOnline) {
                          return /* Online */2;
                        } else {
                          return /* Offline */0;
                        }
```

```
                    }));
        };
        subscribeToFriendStatus(friendId, handleStatusChange);
        Curry._1(setState, (function (param) {
                return /* Loading */1;
              }));
        return (function (param) {
                  unsubscribeFromFriendStatus(friendId, handleStatusChange);

                }));
      });
  return match[0];
}
```

Thereâ€™s nothing new inside of it â€” the logic is copied from the components above. Just like in a component, make sure to only call other Hooks unconditionally at the top level of your custom Hook.

Unlike a React component, a custom Hook doesnâ€™t need to have a specific signature. We can decide what it takes as arguments, and what, if anything, it should return. In other words, itâ€™s just like a normal function. Its name should always start with use so that you can tell at a glance that the rules of Hooks apply to it.

The purpose of our `useFriendStatus` Hook is to subscribe us to a friendâ€™s status. This is why it takes `friendId` as an argument, and returns the online state like `Online`, `Offline` or `Loading`:

```
let useFriendStatus = (friendId: string): status {
  let (state, setState) = React.useState(_ => Offline);

  // ...

  state
}
```

Now letâ€™s use our custom Hook.

Using a Custom Hook

In the beginning, our stated goal was to remove the duplicated logic from the `FriendStatus` and `FriendListItem` components. Both of them want to know the online state of a friend.

Now that weâ€™ve extracted this logic to a useFriendStatus hook, we can just use it:

```res {6} // FriendStatus.res type friend = { id: string };

@react.component let make = (~friend: friend) => { let onlineState = FriendStatusHook.useFriendStatus(friend.id);

let status = switch(onlineState) { | FriendStatusHook.Online => "Online" | Loading => "Loading" | Offline => "Offline" }

React.string(status); }
```

```js
function FriendStatus(Props) {
  var friend = Props.friend;
  var onlineState = useFriendStatus(friend.id);
  var color;
  switch (onlineState) {
    case /* Offline */0 :
        color = "red";
        break;
    case /* Loading */1 :
        color = "grey";
        break;
    case /* Online */2 :
        color = "green";
        break;

  }
  return React.createElement("li", {
              style: {
                color: color
              }
            }, friend.name);
}
```

```res {4} // FriendListItem.res @react.component let make = (~friend: friend) => { let onlineState = FriendStatusHook.useFriendStatus(friend.id);

let color = switch(onlineState) { | Offline => "red" | Online => "green" | Loading => "grey" }

- {React.string(friend.name)}

}
```

```js
function FriendListItem(Props) {
  var friend = Props.friend;
  var onlineState = useFriendStatus(friend.id);
  var color;
  switch (onlineState) {
    case /* Offline */0 :
        color = "red";
        break;
    case /* Loading */1 :
        color = "grey";
        break;
    case /* Online */2 :
        color = "green";
        break;

  }
  return React.createElement("li", {
              style: {
                color: color
              }
            }, friend.name);
}
```

Is this code equivalent to the original examples? Yes, it works in exactly the same way. If you look closely, youâ€™ll notice we didnâ€™t make any changes to the behavior. All we did was to extract some common code between two functions into a separate function. Custom Hooks are a convention that naturally follows from the design of Hooks, rather than a React feature.

Do I have to name my custom Hooks starting with â€œuseâ€? Please do. This convention is very important. Without it, we wouldnâ€™t be able to automatically check for violations of rules of Hooks because we couldnâ€™t tell if a certain function contains calls to Hooks inside of it.

Do two components using the same Hook share state? No. Custom Hooks are a mechanism to reuse stateful logic (such as setting up a subscription and remembering the current value), but every time you use a custom Hook, all state and effects inside of it are fully isolated.

How does a custom Hook get isolated state? Each call to a Hook gets isolated state. Because we call useFriendStatus directly, from Reactâ€™s point of view our component just calls useState and useEffect. And as we learned earlier, we can call useState and useEffect many times in one component, and they will be completely independent.

Tip: Pass Information Between Hooks

Since Hooks are functions, we can pass information between them.

To illustrate this, weâ€™ll use another component from our hypothetical chat example. This is a chat message recipient picker that displays whether the currently selected friend is online:

```res {11,12,14-18,22} type friend = {id: string, name: string}

let friendList = [ {id: "1", name: "Phoebe"}, {id: "2", name: "Rachel"}, {id: "3", name: "Ross"}, ]

@react.component let make = () => { let (recipientId, setRecipientId) = React.useState(_ => "1") let recipientOnlineState = FriendStatusHook.useFriendStatus(recipientId)

let color = switch recipientOnlineState { | FriendStatusHook.Offline => Circle.Red | Online => Green | Loading => Grey }

let onChange = evt => { let value = ReactEvent.Form.target(evt)["value"] setRecipientId(value) }

let friends = Belt.Array.map(friendList, friend => {

})

<>

}
```

```js
var friendList = [
  {
    id: "1",
    name: "Phoebe"
  },
  {
    id: "2",
    name: "Rachel"
  },
  {
    id: "3",
    name: "Ross"
  }
```

```
];

function Playground(Props) {
  var match = React.useState(function () {
        return "1";
      });
  var setRecipientId = match[1];
  var recipientId = match[0];
  var recipientOnlineState = useFriendStatus(recipientId);
  var color;
  switch (recipientOnlineState) {
    case /* Offline */0 :
        color = /* Red */0;
        break;
    case /* Loading */1 :
        color = /* Grey */2;
        break;
    case /* Online */2 :
        color = /* Green */1;
        break;

  }
  var onChange = function (evt) {
    return Curry._1(setRecipientId, evt.target.value);
  };
  var friends = Belt_Array.map(friendList, (function (friend) {
          return React.createElement("option", {
                      key: friend.id,
                      value: friend.id
                    }, friend.name);
        }));
  return React.createElement(React.Fragment, undefined, React.createElement(Playground$Circle, {
                  color: color
                }), React.createElement("select", {
                  value: recipientId,
                  onChange: onChange
                }, friends));
}
```

We keep the currently chosen friend ID in the `recipientId` state variable, and update it if the user chooses a different friend in the `<select>` picker.

Because the useState Hook call gives us the latest value of the `recipientId` state variable, we can pass it to our custom `FriendStatusHook.useFriendStatus` Hook as an argument:

```
let (recipientId, setRecipientId) = React.useState(_ => "1")
let recipientOnlineState = FriendStatusHook.useFriendStatus(recipientId)
```

This lets us know whether the currently selected friend is online. If we pick a different friend and update the `recipientId` state variable, our `FriendStatus.useFriendStatus` Hook will unsubscribe from the previously selected friend, and subscribe to the status of the newly selected one.

Use Your Imagination

Custom Hooks offer the flexibility of sharing logic. You can write custom Hooks that cover a wide range of use cases like form handling, animation, declarative subscriptions, timers, and probably many more we havenâ€™t considered. Whatâ€™s more, you can build Hooks that are just as easy to use as Reactâ€™s built-in features.

Try to resist adding abstraction too early. It's pretty common that components grow pretty big when there is enough stateful logic handling involved. This is normal â€” donâ€™t feel like you have to immediately split it into Hooks. But we also encourage you to start spotting cases where a custom Hook could hide complex logic behind a simple interface, or help untangle a messy component.

SECTION rendering-elements

---

title: Rendering Elements description: "How to render React elements to the DOM" canonical: "/docs/react/latest/rendering-elements"

Rendering Elements

In our previous section React Elements & JSX we learned how to create and handle React elements. In this section we will learn how to put our elements into action by rendering them into the DOM.

As we mentioned before, a `React.element` describes what you see on the screen:

```
let element = <h1> {React.string("Hello World")} </h1>
```

Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.

Rendering Elements to the DOM

Let's assume we've got an HTML file that contains a `div` like this:

```
<div id="root"/>
```

We call this a â€œrootâ€ DOM node because everything inside it will be managed by React DOM.

Plain React applications usually have a single root DOM node. If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.

To render our React application into the `root` div, we need to do two things: - Find our DOM node with `ReactDOM.querySelector` - Render our React element to our queried `Dom.element` with `ReactDOM.render`

Here is a full example rendering our application in our `root` div:

```
// Dom access can actually fail. ReScript
// is really explicit about handling edge cases!
switch ReactDOM.querySelector("#root") {
| Some(rootElement) => {
    let root = ReactDOM.Client.createRoot(rootElement)
    ReactDOM.Client.Root.render(root, <div />)
  }
| None => ()
}
```

```
var root = document.querySelector("#root");

if (!(root == null)) {
  ReactDom.render(React.createElement("div", undefined, "Hello Andrea"), root);
}
```

React elements are immutable. Once you create an element, you canâ€™t change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

At this point we would need to understand a few more concepts, such as React components and state management to be able to update the rendered elements after the initial `ReactDOM.Client.Root.render`. For now, just imagine your React application as a tree of elements, whereby some elements may get replaced during the lifetime of your application.

React will automatically recognize any element changes and will only apply the DOM updates necessary to bring the DOM to the desired state.

SECTION hooks-effect

---

title: useEffect Hook description: "Details about the useEffect React hook in ReScript" canonical: "/docs/react/latest/hooks-effect"

---

useEffect

The *Effect* Hook lets you perform side effects in function components.

What are Effects?

Common examples for (side) effects are data fetching, setting up a subscription, and manually changing the DOM in React components.

There are two common kinds of side effects in React components: those that donâ€™t require cleanup, and those that do. We'll look into the distinction later on in our examples, but first let's see how the interface looks like.

Basic Usage

```
// Runs after every completed render
React.useEffect(() => {
  // Run effects
  None // or Some(() => {})
})


// Runs only once right after mounting the component
React.useEffect0(() => {
  // Run effects
  None // or Some(() => {})
})

// Runs everytime `prop1` has changed
React.useEffect1(() => {
  // Run effects based on prop1
```

```
  None
}, [prop1])

// Runs everytime `prop1` or `prop2` has changed
React.useEffect2(() => {
  // Run effects based on prop1 / prop2
  None
}, (prop1, prop2))

React.useEffect3(() => {
  None
}, (prop1, prop2, prop3));

// useEffect4...7 with according dependency
// tuple just like useEffect3

React.useEffect(function () { });
React.useEffect((function () { }), []);
React.useEffect((function () { }), [prop1]);
React.useEffect((function () { }), [ prop1, prop2 ]);
React.useEffect((function () { }), [ prop1, prop2, prop3 ]);
```

`React.useEffect` receives a function that contains imperative, possibly effectful code, and returns a value `option<unit => unit>` as a potential cleanup function.

A `useEffect` call may receive an additional array of dependencies (see `React.useEffect1` / `React.useEffect2...7`). The effect function will run whenever one of the provided dependencies has changed. More details on why this is useful down below.

Note: You probably wonder why `React.useEffect1` receives an `array`, and `useEffect2` etc require a `tuple` (e.g. `(prop1, prop2)`) for the dependency list. That's because a tuple can receive multiple values of different types, whereas an `array` only accepts values of identical types. It's possible to replicate `useEffect2` by doing `React.useEffect1(fn, [1, 2])`, on other hand the type checker wouldn't allow `React.useEffect1(fn, [1, "two"])`.

`React.useEffect` will run its function after every completed render, while `React.useEffect0` will only run the effect on the first render (when the component has mounted).

Examples

Effects without Cleanup

Sometimes, we want to run some additional code after React has updated the DOM. Network requests, manual DOM mutations, and logging are common examples of effects that donâ€™t require a cleanup. We say that because we can run them and immediately forget about them.

As an example, let's write a counter component that updates `document.title` on every render:

```
// Counter.res
module Document = {
  type t;
  @val external document: t = "document";
  @set external setTitle: (t, string) => unit = "title"
}

@react.component
let make = () => {
  let (count, setCount) = React.useState(_ => 0);

  React.useEffect(() => {
    open Document
    document->setTitle(`You clicked ${Belt.Int.toString(count)} times!`)
    None
  }, );

  let onClick = (_evt) => {
    setCount(prev => prev + 1)
  };

  let msg = "You clicked" ++ Belt.Int.toString(count) ++  "times"

  <div>
    <p>{React.string(msg)}</p>
    <button onClick> {React.string("Click me")} </button>
  </div>
}

function Counter(Props) {
  var match = React.useState(function () {
        return 0;
      });
  var setCount = match[1];
  var count = match[0];
  React.useEffect(function () {
```

```
        document.title = "You clicked " + String(count) + " times!";

      });
  var onClick = function (_evt) {
    return Curry._1(setCount, (function (prev) {
                  return prev + 1 | 0;
                }));
  };
  var msg = "You clicked" + String(count) + "times";
  return React.createElement("div", undefined,
    React.createElement("p", undefined, msg),
      React.createElement("button", {
                  onClick: onClick
                }, "Click me"));
}
```

In case we want to make the effects dependent on `count`, we can just use following `useEffect` call instead:

```
 React.useEffect1(() => {
    open Document
    document->setTitle(`You clicked ${Belt.Int.toString(count)} times!`)
    None
  }, [count]);
```

Now instead of running an effect on every render, it will only run when `count` has a different value than in the render before.

Effects with Cleanup

Earlier, we looked at how to express side effects that donâ€™t require any cleanup. However, some effects do. For example, we might want to set up a subscription to some external data source. In that case, it is important to clean up so that we donâ€™t introduce a memory leak!

Let's look at an example that gracefully subscribes, and later on unsubscribes from some subscription API:

```
// FriendStatus.res

module ChatAPI = {
  // Imaginary globally available ChatAPI for demo purposes
  type status = { isOnline: bool };
  @val external subscribeToFriendStatus: (string, status => unit) => unit = "subscribeToFriendStatus";
  @val external unsubscribeFromFriendStatus: (string, status => unit) => unit = "unsubscribeFromFriendStatus";
}

type state = Offline | Loading | Online;

@react.component
let make = (~friendId: string) => {
  let (state, setState) = React.useState(_ => Offline)

  React.useEffect(() => {
    let handleStatusChange = (status) => {
      setState(_ => {
        status.ChatAPI.isOnline ? Online : Offline
      })
    }

    ChatAPI.subscribeToFriendStatus(friendId, handleStatusChange);
    setState(_ => Loading);

    let cleanup = () => {
      ChatAPI.unsubscribeFromFriendStatus(friendId, handleStatusChange)
    }

    Some(cleanup)
  })

  let text = switch(state) {
    | Offline => friendId ++ " is offline"
    | Online => friendId ++ " is online"
    | Loading => "loading..."
  }

  <div>
    {React.string(text)}
  </div>
}

function FriendStatus(Props) {
  var friendId = Props.friendId;
  var match = React.useState(function () {
        return /* Offline */0;
      });
  var setState = match[1];
  React.useEffect(function () {
```

```
        var handleStatusChange = function (status) {
          return Curry._1(setState, (function (param) {
                        if (status.isOnline) {
                          return /* Online */2;
                        } else {
                          return /* Offline */0;
                        }
                      }));
        };
        subscribeToFriendStatus(friendId, handleStatusChange);
        Curry._1(setState, (function (param) {
                return /* Loading */1;
              }));
        return (function (param) {
                  unsubscribeFromFriendStatus(friendId, handleStatusChange);

                });
      });
  var text;
  switch (match[0]) {
    case /* Offline */0 :
        text = friendId + " is offline";
        break;
    case /* Loading */1 :
        text = "loading...";
        break;
    case /* Online */2 :
        text = friendId + " is online";
        break;

  }
  return React.createElement("div", undefined, text);
}
```

Effect Dependencies

In some cases, cleaning up or applying the effect after every render might create a performance problem. Let's look at a concrete example to see what `useEffect` does:

```
// from a previous example above
React.useEffect1(() => {
  open Document
  document->setTitle(`You clicked ${Belt.Int.toString(count)} times!`)
  None;
}, [count]);
```

Here, we pass `[count]` to `useEffect1` as a dependency. What does this mean? If the `count` is 5, and then our component re-renders with count still equal to 5, React will compare `[5]` from the previous render and `[5]` from the next render. Because all items within the array are the same (5 === 5), React would skip the effect. Thatâ€™s our optimization.

When we render with count updated to 6, React will compare the items in the `[5]` array from the previous render to items in the `[6]` array from the next render. This time, React will re-apply the effect because `5 !== 6`. If there are multiple items in the array, React will re-run the effect even if just one of them is different.

This also works for effects that have a cleanup phase:

```
// from a previous example above
React.useEffect1(() => {
  let handleStatusChange = (status) => {
    setState(_ => {
      status.ChatAPI.isOnline ? Online : Offline
    })
  }

  ChatAPI.subscribeToFriendStatus(friendId, handleStatusChange);
  setState(_ => Loading);

  let cleanup = () => {
    ChatAPI.unsubscribeFromFriendStatus(friendId, handleStatusChange)
  }

  Some(cleanup)
}, [friendId]) // Only re-subscribe if friendId changes
```

Important: If you use this optimization, make sure the array includes all values from the component scope (such as props and state) that change over time and that are used by the effect. Otherwise, your code will reference stale values from previous renders

If you want to run an effect and clean it up only once (on mount and unmount), use `React.useEffect0`.

If you are interested in more performance optmization related topics, have a look at the ReactJS Performance Optimization Docs for more detailed infos.

SECTION router

---

title: Router description: "Basic concepts for navigation and routing in ReScript & React" canonical: "/docs/react/latest/router"

Router

RescriptReact comes with a router! We've leveraged the language and library features in order to create a router that's:

- The simplest, thinnest possible.
- Easily pluggable anywhere into your existing code.
- Performant and tiny.

How does it work?

The available methods are listed here: - `RescriptReactRouter.push(string)`: takes a new path and update the URL. - `RescriptReactRouter.replace(string)`: like `push`, but replaces the current URL. - `RescriptReactRouter.watchUrl(f)`: start watching for URL changes. Returns a subscription token. Upon url change, calls the callback and passes it the `RescriptReactRouter.url` record. - `RescriptReactRouter.unwatchUrl(watcherID)`: stop watching for URL changes. - `RescriptReactRouter.dangerouslyGetInitialUrl()`: get `url` record outside of `watchUrl`. Described later. - `RescriptReactRouter.useUrl(~serverUrl)`: returns the `url` record inside a component.

> If you want to know more about the low level details on how the router interface is implemented, refer to the [RescriptReactRouter implementation].

Match a Route

There's no API! `watchUrl` gives you back a `url` record of the following shape:

```res prelude type url = { /* path takes window.location.pathname, like "/book/title/edit" and turns it intolist{"book", "title", "edit"}` */ *path: list,* /the url's hash, if any. The ### symbol is stripped out for you */ hash: string,* /the url's query params, if any. The ? symbol is stripped out for you */ search: string }

```js
// Empty output
```

So the url `www.hello.com/book/10/edit?name=Jane#author` is given back as:

```res prelude { path: list{"book", "10", "edit"}, hash: "author", search: "name=Jane" }

```js
// Empty output
```

Basic Example

Let's start with a first example to see how a ReScript React Router looks like:

```
// App.res
@react.component
let make = () => {
  let url = RescriptReactRouter.useUrl()

  switch url.path {
    | list{"user", id} => <User id />
    | list{} => <Home/>
    | _ => <PageNotFound/>
  }
}
```

```
import * as React from "react";
import * as User from "./User.bs.js";
import * as RescriptReactRouter from "@rescript/react/src/RescriptReactRouter.bs.js";
import * as Home from "./Home.bs.js";
import * as NotFound from "./NotFound.bs.js";

function App(Props) {
  var url = RescriptReactRouter.useUrl(undefined, undefined);
  var match = url.path;
  if (!match) {
    return React.createElement(Home.make, {});
  }
  if (match.hd === "user") {
    var match$1 = match.tl;
    if (match$1 && !match$1.tl) {
      return React.createElement(User.make, {
                  id: match$1.hd
```

```
                });
    }

  }
  return React.createElement(NotFound.make, {});
}

var make = App;

export {
  make ,

}
```

Directly Get a Route

In one specific occasion, you might want to take hold of a `url` record outside of `watchUrl`. For example, if you've put `watchUrl` inside a component's `didMount` so that a URL change triggers a component state change, you might also want the initial state to be dictated by the URL.

In other words, you'd like to read from the `url` record once at the beginning of your app logic. We expose `dangerouslyGetInitialUrl()` for this purpose.

Note: the reason why we label it as "dangerous" is to remind you not to read this `url` in any arbitrary component's e.g. `render`, since that information might be out of date if said component doesn't also contain a `watchUrl` subscription that re-renders the component when the URL changes. Aka, please only use `dangerouslyGetInitialUrl` alongside `watchUrl`.

Push a New Route

From anywhere in your app, just call e.g. `RescriptReactRouter.push("/books/10/edit#validated")`. This will trigger a URL change (without a page refresh) and `watchUrl`'s callback will be called again.

We might provide better facilities for typed routing + payload carrying in the future!

Note: because of browser limitations, changing the URL through JavaScript (aka pushState) cannot be detected. The solution is to change the URL then fire a "popState" event. This is what Router.push does, and what the event watchUrl listens to. So if, for whatever reason (e.g. incremental migration), you want to update the URL outside of `RescriptReactRouter.push`, just do `window.dispatchEvent(new Event('popState'))`.

Design Decisions

We always strive to lower the performance and learning overhead in RescriptReact, and our router design's no different. The entire implementation, barring browser features detection, is around 20 lines. The design might seem obvious in retrospect, but to arrive here, we had to dig back into ReactJS internals & future proposals to make sure we understood the state update mechanisms, the future context proposal, lifecycle ordering, etc. and reject some bad API designs along the way. It's nice to arrive at such an obvious solution!

The API also doesn't dictate whether matching on a route should return a component, a state update, or a side-effect. Flexible enough to slip into existing apps.

Performance-wise, a JavaScript-like API tends to use a JS object of route string -> callback. We eschewed that in favor of pattern-matching, since the latter in Rescript does not allocate memory, and is compiled to a fast jump table in C++ (through the JS JIT). In fact, the only allocation in the router matching is the creation of the url record!

SECTION forwarding-refs

---

title: Forwarding Refs description: "Forwarding Ref values in ReScript and React" canonical: "/docs/react/latest/forwarding-refs"

Forwarding Refs

Ref forwarding is a technique for automatically passing a [React.ref](#) through a component to one of its children. This is typically not necessary for most components in the application. However, it can be useful for some kinds of components, especially in reusable component libraries. The most common scenarios are described below.

Why Ref Forwarding?

Consider a FancyButton component that renders the native button DOM element:

```
// FancyButton.res

@react.component
let make = (~children) => {
  <button className="FancyButton">
```

```
    children
  </button>
}
```

React components hide their implementation details, including their rendered output. Other components using FancyButton **usually will not need** to obtain a ref to the inner button DOM element. This is good because it prevents components from relying on each otherâ€™s DOM structure too much.

Although such encapsulation is desirable for application-level components like `FeedStory` or `Comment`, it can be inconvenient for highly reusable â€œleafâ€ components like `FancyButton` or `MyTextInput`. These components tend to be used throughout the application in a similar manner as a regular DOM button and input, and accessing their DOM nodes may be unavoidable for managing focus, selection, or animations.

There are currently two strategies on forwarding refs through a component. In ReScript and React we strongly recommend **passing your ref as a prop**, but there is also a dedicated API called `React.forwardRef`.

We will discuss both methods in this document.

Forward Refs via Props

A `React.ref` can be passed down like any other prop. The component will take care of forwarding the ref to the right DOM element.

No new concepts to learn!

In the example below, `FancyInput` defines a prop `inputRef` that will be forwarded to its `input` element:

```
// App.res

module FancyInput = {
  @react.component
  let make = (~children, ~inputRef: ReactDOM.domRef) =>
    <div> <input type_="text" ref=inputRef /> children </div>
}

@send external focus: Dom.element => unit = "focus"

@react.component
let make = () => {
  let input = React.useRef(Js.Nullable.null)

  let focusInput = () =>
    input.current
    ->Js.Nullable.toOption
    ->Belt.Option.forEach(input => input->focus)

  let onClick = _ => focusInput()

  <div>
    <FancyInput inputRef={ReactDOM.Ref.domRef(input)}>
      <button onClick> {React.string("Click to focus")} </button>
    </FancyInput>
  </div>
}
```

We use the `ReactDOM.domRef` type to represent our `inputRef`. We pass our ref in the exact same manner as we'd do a DOM `ref` attribute (`<input ref={ReactDOM.Ref.domRef(myRef)} />`).

[Discouraged] React.forwardRef

Note: We discourage this method since it will likely go away at some point, and doesn't yield any obvious advantages over the previously mentioned ref prop passing.

`React.forwardRef` offers a way to "emulate a `ref` prop" within custom components. Internally the component will forward the passed `ref` value to the target DOM element instead.

This is how the previous example would look like with the `React.forwardRef` approach:

```
// App.res

module FancyInput = {
  @react.component
  let make = React.forwardRef((~className=?, ~children, ref) =>
    <div>
      <input
        type_="text"
        ?className
        ref=?{Js.Nullable.toOption(ref)->Belt.Option.map(ReactDOM.Ref.domRef)}
      />
      children
    </div>
```

```
  )
}

@send external focus: Dom.element => unit = "focus"

@react.component
let make = () => {
  let input = React.useRef(Js.Nullable.null)

  let focusInput = () =>
    input.current->Js.Nullable.toOption->Belt.Option.forEach(input => input->focus)

  let onClick = _ => focusInput()

  <div>
    <FancyInput className="fancy" ref=input>
      <button onClick> {React.string("Click to focus")} </button>
    </FancyInput>
  </div>
}
```

```
import * as React from "react";
import * as Belt_Option from "rescript/lib/es6/belt_Option.js";
import * as Caml_option from "rescript/lib/es6/caml_option.js";

var make = React.forwardRef(function (props, ref) {
  return React.createElement(
    "div",
    undefined,
    React.createElement("input", {
      ref: Belt_Option.map(
        ref == null ? undefined : Caml_option.some(ref),
        function (prim) {
          return prim;
        }
      ),
      className: props.className,
      type: "text",
    }),
    props.children
  );
});

var FancyInput = {
  make: make,
};

function App(props) {
  var input = React.useRef(null);
  var onClick = function (param) {
    Belt_Option.forEach(
      Caml_option.nullable_to_opt(input.current),
      function (input) {
        input.focus();
      }
    );
  };
  return React.createElement(
    "div",
    undefined,
    React.createElement(make, {
      className: "fancy",
      children: React.createElement(
        "button",
        {
          onClick: onClick,
        },
        "Click to focus"
      ),
      ref: input,
    })
  );
}
```

Note: Our `@react.component` decorator transforms our labeled argument props within our `React.forwardRef` function in the same manner as our classic `make` function.

This way, components using `FancyInput` can get a ref to the underlying `input` DOM node and access it if necessaryâ€”just like if they used a DOM `input` directly.

Note for Component Library Maintainers

When you start using ref forwarding in a component library, you should treat it as a breaking change and release a new major version of your library. This is because your library likely has an observably different behavior (such as what refs get assigned to, and

what types are exported), and this can break apps and other libraries that depend on the old behavior.

SECTION context

---

title: Context description: "Details about Context in ReScript and React" canonical: "/docs/react/latest/context"

Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

Why Context?

In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

Note: In ReScript, passing down props is way simpler than in TS / JS due to its JSX prop punning feature and strong type inference, so it's often preferrable to keep it simple and just do props passing. Less magic means more transparency!

When to Use Context

Context is designed to share data that can be considered â€œglobalâ€ for a tree of React components, such as the current authenticated user, theme, or preferred language. For example, in the code below we manually thread through a â€œthemeâ€ prop in order to style the Button component:

```
// src/App.res
type theme = Light | Dark;

module Button = {
  @react.component
  let make = (~theme) => {
    let className = switch theme {
      | Light => "theme-light"
      | Dark => "theme-black"
    };
    <button className> {React.string("Click me")} </button>
  }
}

module ThemedButton = {
  @react.component
  let make = (~theme) => {
    <Button theme />
  }
}

module Toolbar = {
  @react.component
  let make = (~theme) => {
    <div>
      <ThemedButton theme/>
    </div>
  }
}

@react.component
let make = () => {
  // We define the theme in the
  // toplevel App component and
  // pass it down
  <Toolbar theme=Dark/>
}
```

```
function Button(props) {
  var className = props.theme ? "theme-black" : "theme-light";
  return React.createElement("button", {
              className: className
            }, "Click me");
}

function ThemedButton(props) {
  return React.createElement(App$Button, {
              theme: props.theme
            });
}

function Toolbar(props) {
  return React.createElement("div", undefined, React.createElement(App$ThemedButton, {
```

```
                  theme: props.theme
                }));
}

function App(props) {
  return React.createElement(App$Toolbar, {
              theme: /* Dark */1
            });
}
```

Using context, we can avoid passing props through intermediate elements:

```
// src/App.res

module ThemeContext = {
  type theme = Light | Dark
  let context = React.createContext(Light)

  module Provider = {
    let make = React.Context.provider(context)
  }
}

module Button = {
  @react.component
  let make = (~theme) => {
    let className = switch theme {
    | ThemeContext.Light => "theme-light"
    | Dark => "theme-black"
    }
    <button className> {React.string("Click me")} </button>
  }
}

module ThemedButton = {
  @react.component
  let make = () => {
    let theme = React.useContext(ThemeContext.context)

    <Button theme />
  }
}

module Toolbar = {
  @react.component
  let make = () => {
    <div>
      <ThemedButton />
    </div>
  }
}

@react.component
let make = () => {
  <ThemeContext.Provider value=ThemeContext.Dark>
    <div>
      <Toolbar />
    </div>
  </ThemeContext.Provider>
}

var context = React.createContext(/* Light */0);

var make = context.Provider;

var Provider = {
  make: make
};

var ThemeContext = {
  context: context,
  Provider: Provider
};

function App$Button(props) {
  var className = props.theme ? "theme-black" : "theme-light";
  return React.createElement("button", {
              className: className
            }, "Click me");
}

var Button = {
  make: App$Button
};

function App$ThemedButton(props) {
  var theme = React.useContext(context);
```

```
  return React.createElement(App$Button, {
              theme: theme
            });
}

var ThemedButton = {
  make: App$ThemedButton
};

function App$Toolbar(props) {
  return React.createElement("div", undefined, React.createElement(App$ThemedButton, {}));
}

var Toolbar = {
  make: App$Toolbar
};

function App(props) {
  return React.createElement(make, {
              value: /* Dark */1,
              children: React.createElement("div", undefined, React.createElement(App$Toolbar, {}))
            });
}
```

SECTION installation

---

title: Installation description: "Installation and Setup" canonical: "/docs/react/latest/installation"

Installation

Requirements:

- `rescript@10.1` or later
- `react@18.0.0` or later

Add following dependency to your ReScript project (in case you don't have any project yet, check out the installation instructions in the manual):

```
npm install @rescript/react
```

Then add the following setting to your existing `bsconfig.json`:

```
{
  "jsx": { "version": 4, "mode": "classic" },
  "bs-dependencies": ["@rescript/react"]
}
```

> The new jsx transform of ReactJS is available with `mode: "automatic"`.

Note JSX v4 is newly introduced with the idiomatic record-based representation of a component. If your dependencies are not compatible with v4 yet, then you can use v3 in the same project. Check out the details in Migrate from v3

To test your setup, create a new `.res` file in your source directory and add the following code:

```
// src/Test.res
@react.component
let make = () => {
  <div> {React.string("Hello World")} </div>
}
```

Now run `npx rescript` and you should see a successful build.

Exposed Modules

After a successful installation, `@rescript/react` will make the following modules available in your project's global scope:

- `React`: Bindings to React
- `ReactDOM`: Bindings to the ReactDOM
- `ReactDOMServer`: Bindings to the ReactDOMServer
- `ReactEvent`: Bindings to React's synthetic events
- `ReactDOMStyle`: Bindings to the inline style API
- `RescriptReactRouter`: A simple, yet fully featured router with minimal memory allocations
- `RescriptReactErrorBoundary`: A component which handles errors thrown in its child components gracefully