View online

Download PDF



Compiling and bundling JS, the painless way

Paolo Insogna

Node.js TSC, Principal Engineer @ **Platformatic**

Fight your fears!



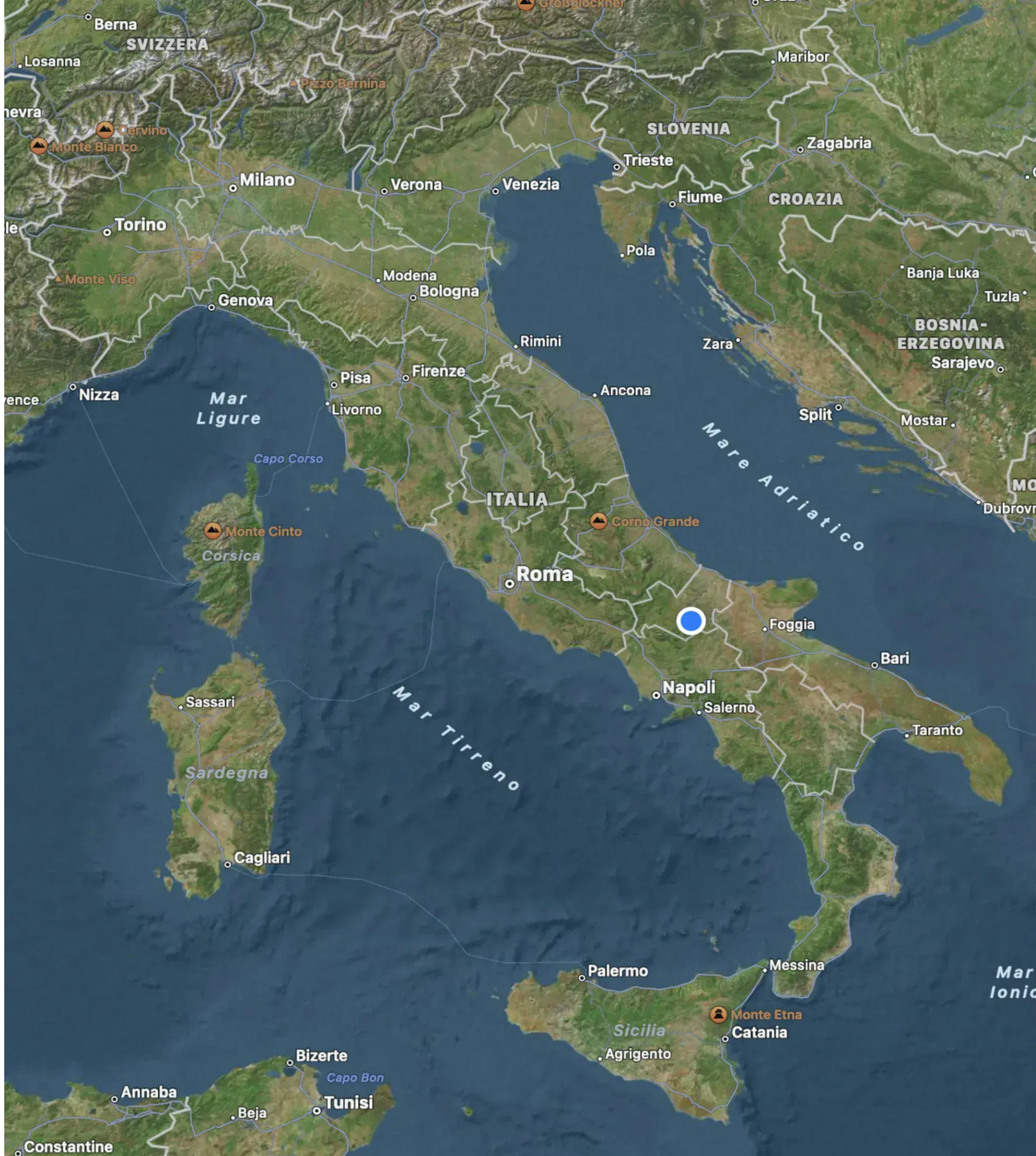
Hello, I'm **Paolo**!



Node.js Technical Steering Committee Member

Platformatic Principal Engineer

`paoloinsogna.dev` `ShogunPanda` `p_insogna` `pinsogna`



First of all, let's give credits!

This talk has originally been written by my colleague and friend **Michele Riva**.

Whatever goes wrong today, please complain directly to him on Twitter!

@MicheleRivaCode



Compiling and bundling JavaScript is often a pain ...





...but it should not!

A bit of terminology: Compiling

“To change a computer program into a machine language”

Cambridge Dictionary

A bit of terminology: Transpiling

“To translate a source code into a different language source code”

Michele Riva

A bit of terminology: Bundling

“To pack all code and resources in a single source file or executable”

Me, a.k.a. Paolo Insogna

Transpilation

Why do we want to transpile our code?

We all have a loved language we'd like to see everywhere.



Adopt new language features

Runtimes update slower than the languages.

We don't want to wait.



Write once, run everywhere™

We want to write our scripts in our language despite of the runtime environment.

Who are you missing the most?

We all have a loved language we'd like to see everywhere.



Scala.js

Scala to Javascript



Opal

Ruby to Javascript

An example: transpilation of Scala.js

Feature	JavaScript ES5	JavaScript ES6	TypeScript	Scala.js
Interoperability				
Fully EcmaScript5 compatible	●	●	●	●
No compilation required	●	●	○	○
Use existing JS libraries	●	●	●	●
Language features				
Classes	○	●	●	●
Modules	○	●	●	●
Support for types	○	○	●	●
Strong type system	○	○	○	●
Extensive standard libraries	○	○	○	●
Optimizing compiler	○	○	○	●
Macros, to extend the language	○	○	○	●
IDE support				
Catch most errors in IDE	○	○	●	●
Easy and reliable refactoring	○	○	●	●
Reliable code completion	○	○	●	●

Where do we (mostly) run?

There are few known runtimes and browsers, way less than in the past.









There is a transpiler for everything...™



F#



Kotlin



Gleam



ReasonML



ELM



ReScript

… and that's thanks to LLVM!

The project is a collection of modular and reusable compiler and toolchains.



What are you missing the most?

We all want to adopt new language features even before they are finalised.

```
function toUpper(x) {
  return x.toUpperCase();
}

function addExclamationMarks(x) {
  return x + '!!!'
}

function scream(input) {
  return input
    |> toUpper
    |> addExclamationMarks
    |> console.log
}

scream('Hello world');
// HELLO WORLD!!!
```




```
"use strict";

function toUpper(x) {
  return x.toUpperCase();
}

function addExclamationMarks(x) {
  return x + '!!!';
}

function scream(input) {
  var _ref, _ref2, _input;

  return _ref = (
    _ref2 = (_input = input, toUpper(_input)),
    addExclamationMarks(_ref2)
  ), console.log(_ref);
}

scream('Hello world');
// HELLO WORLD!!!
```

There is no end to what we can achieve

We can even transpile features that likely will never get into the language.

```tsx
import { VFC } from 'react';

enum UserType {
  ADMIN     = "admin",
  EDITOR    = "editor",
  USER      = "user",
  ANONYMOUS = "guest"
}

type MyProps = {
  userType: UserType;
}

const MyComponent: VFC<MyProps> = ({ userType }) => {
  return (
    <div>
      User is of type: {props.userType}
    </div>
  )
}
```




```tsx
"use strict";

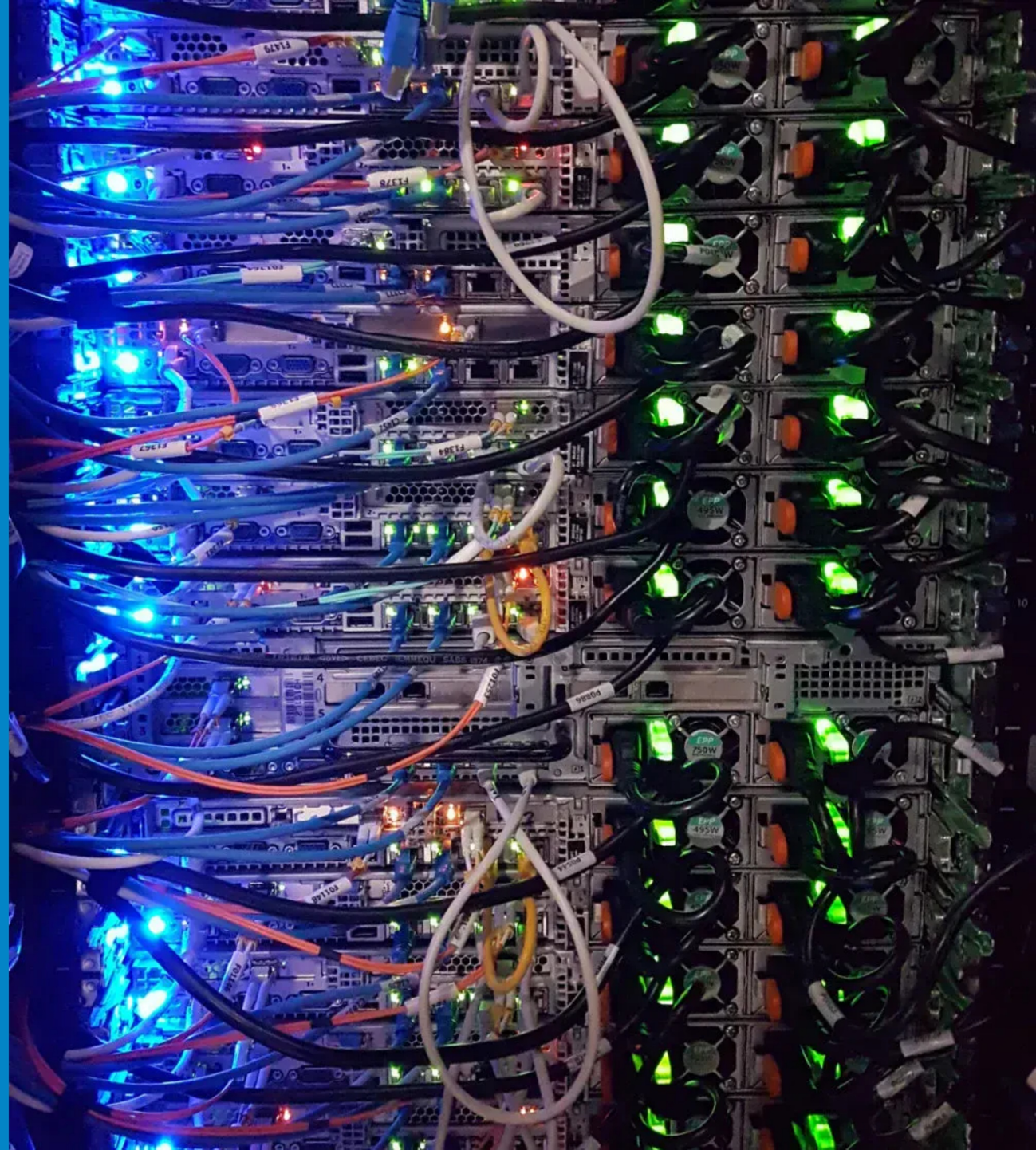
Object.defineProperty(exports, "__esModule", {
  value: true
});
var UserType;

(function (UserType) {
  UserType["ADMIN"] = "admin";
  UserType["EDITOR"] = "editor";
  UserType["USER"] = "user";
  UserType["ANONYMOUS"] = "guest";
})(UserType || (UserType = {}));

const MyComponent = ({ userType }) => {
  return /*#__PURE__*/ React.createElement(
    "div",
    null,
    "User is of type: ",
    props.userType
  );
};
```

Transpiling, in depth

Let's have a deep look on how a transpiler really works.



Generated code is not always readable...

```clojure
(defn simple-component []
  [:div
   [:p "I am a component!"]
   [:p.someclass
    "I have " [:strong "bold"]
    [:span {:style {:color "red"}} " and red "] "text."]])
```



```javascript
cljs.user.simple_component = (function cljs$user$simple_component(){
  return new cljs.core.PersistentVector(null, 3, 5, cljs.core.PersistentVector.EMPTY_NODE,
  [new cljs.core.Keyword(null,"div","div",(1057191632)),
  new cljs.core.PersistentVector(null, 2, 5, cljs.core.PersistentVector.EMPTY_NODE,
  [new cljs.core.Keyword(null,"p","p",(151049309)),"I am a component!"], null),
  new cljs.core.PersistentVector(null, 5, 5, cljs.core.PersistentVector.EMPTY_NODE,
  [new cljs.core.Keyword(null,"p.someclass","p.someclass",(-1904646929)),
  "I have ",new cljs.core.PersistentVector(null, 2, 5, cljs.core.PersistentVector.EMPTY_NODE,
  [new cljs.core.Keyword(null,"strong","strong",(269529000)),"bold"], null),
  new cljs.core.PersistentVector(null, 3, 5, cljs.core.PersistentVector.EMPTY_NODE,
  [new cljs.core.Keyword(null,"span","span",(1394872991)),
  new cljs.core.PersistentArrayMap(null, 1, [new cljs.core.Keyword(null,"style","style",(-496642736)),
  new cljs.core.PersistentArrayMap(null, 1, [
  new cljs.core.Keyword(null,"color","color",(1011675173)),"red"]
  , null)], null)," and red "], null),"text."], null)], null);
});
```

...but some transpilers are really good!

```
[@bs.config {jsx: 3}];

module Greeting = {
  [@react.component]
  let make = ( ) => {
    <button> {React.string("Hello!")} </button>
  };
};

ReactDOMRe.renderToElementWithId(<Greeting />, "preview");
```



```
// Generated by BUCKLESCRIPT, PLEASE EDIT WITH CARE
'use strict';

var React = require("react");
var ReactDOMRe = require("./stdlib/reactDOMRe.js");

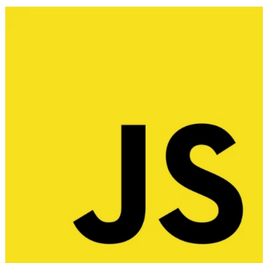
function _none_$Greeting(Props) {
  return React.createElement("button", undefined, "Hello!");
}

var Greeting = {
  make: _none_$Greeting
};

ReactDOMRe.renderToElementWithId(React.createElement(_none_$Greeting, { }), "preview");

exports.Greeting = Greeting;
/*  Not a pure module */
```

Each language has its own transpiler



Babel



TSC



BuckleScript

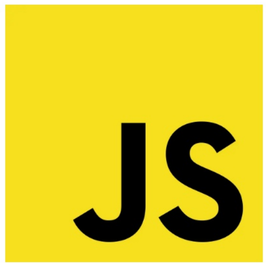


ClojureScript

No transpiler is perfect!



Problem #1: Transpilation time



Slow on large codebases



Slow on large codebases

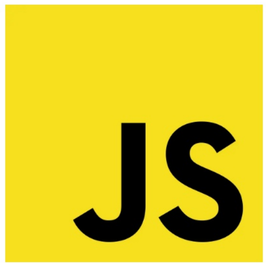


Really fast



Average

Problem #2: Output optimization



Well optimized



Quite optimized



Beautifully optimized

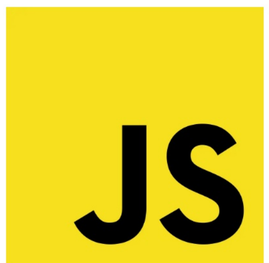
2



Awful

Let's focus on the popular one!

No intention to discriminate any language.



Babel

Quite fast, well optimized



TSC

Quite slow, quite optimized

How does a transpiler work?

Transpiling time grows proportionally to the source code.

1. **Parsing**
   The source code is parsed and converted to an Abstract Syntax Tree (AST)

2. **Transformation**
   The source language AST is traversed and mapped to the destination language AST.

3. **Code generation (codegen)**
   The destination language AST is converted to the destination source code.

Parsing step #1: Tokenization

The input is divided into tokens.



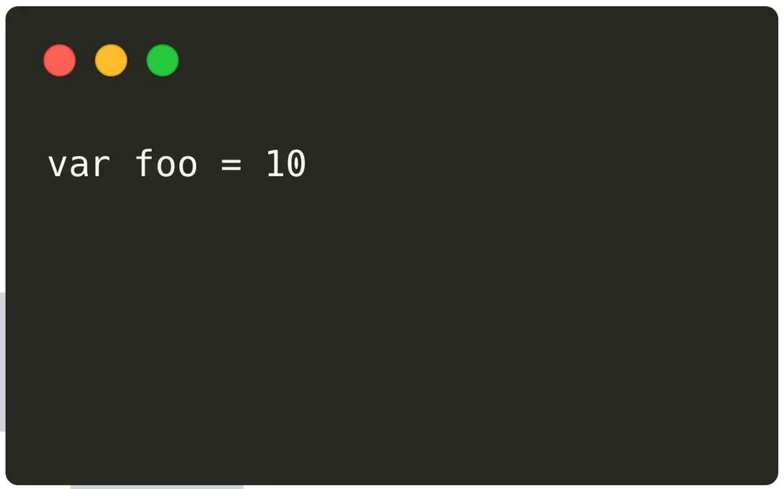
Input



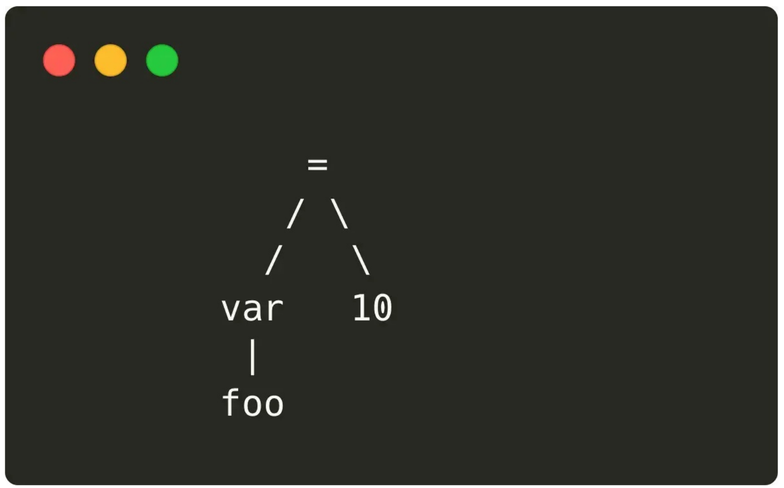
Output

Parsing step #2: Syntactical Analysis

The tokens are parsed and analysed.



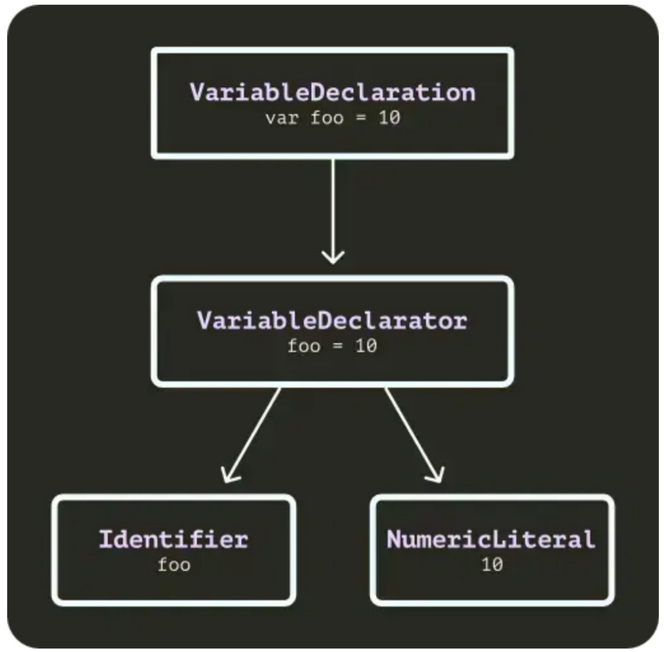
Input
(user code)



Parse Tree
(a.k.a. Concrete Syntax Tree)

Parsing step #3: Prepare the AST

The Parse Tree is converted into the Abstract syntax tree.



3

Parsing step #4: Build the AST

The Abstract Syntax Tree is built and returned to the developer.



4

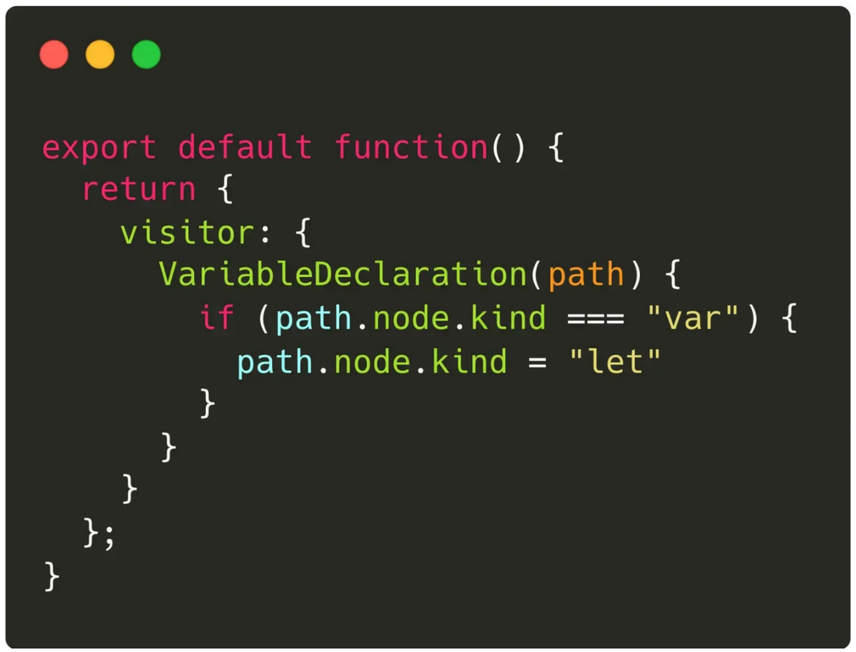
Traversing the AST

The transpiler exposes an API to allow developer to manipulate the AST.

```
export default function () {
  return {
    visitor: {
      VariableDeclaration(path) {
        console.log({ VariableDeclaration: path.node });
        // { type: "VariableDeclaration", kind: "var", ... }
      },
      Identifier(path) {
        console.log({ Identifier: path.node });
        // { type: "Identifier", name: "foo", ... }
      },
      NumericLiteral(path) {
        console.log({ NumericLiteral: path.node });
        // { type: "NumericLiteral", value: 10, ... }
      }
    }
  };
}
```

Transforming the AST

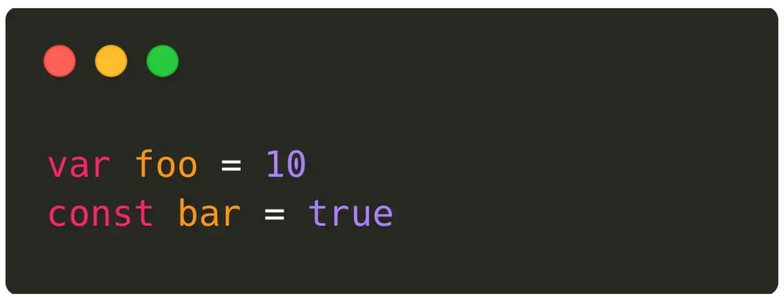
The traversing API allows to manipulate the AST.



```
export default function() {
  return {
    visitor: {
      VariableDeclaration(path) {
        if (path.node.kind === "var") {
          path.node.kind = "let"
        }
      }
    }
  };
}
```

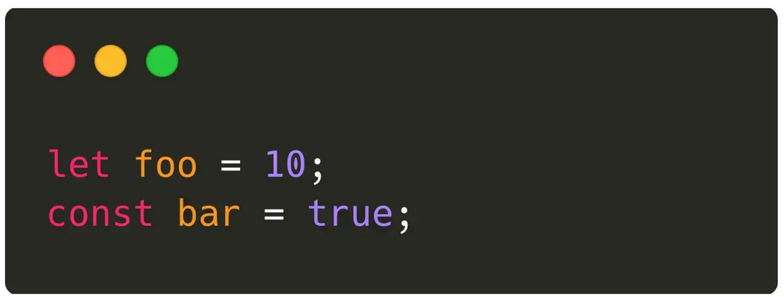
Code generation

The modified AST is used to generate the final transpiled code.



```
var foo = 10
const bar = true
```

Input



```
let foo = 10;
const bar = true;
```

Output

All your popular tools use this flow



Babel



Prettier



ESLint

A more complex example: the problem

jscodeshift can be used to easily migrate codebases.



```
import MyHeader from 'components/MyHeader';

export function MyApp(props) {
  return (
    <div>
      <MyHeader {...props.headerProps} />
      <p> Hello, {props.name}!</p>
    </div>
  )
}
```

Input



```
export function MyApp(props) {
  return (
    <div>
      <p> Hello, {props.name}!</p>
    </div>
  )
}
```

Desired output

A more complex example: the solution

```
export const parser = 'babel'

export default function transformer(file, api) {
  const j = api.jscodeshift;

  const withoutElement = j(file.source)
    .find(j.JSXElement)
    .forEach(function (path) {
      if (path.value.openingElement.name.name === "MyHeader") {
        path.prune();
      }
    })
    .toSource();

  const withoutImport = j(withoutElement)
    .find(j.ImportDefaultSpecifier)
    .forEach(function (path) {
      if (path.value.local.name === "MyHeader") {
        path.parentPath.parentPath.prune();
      }
    })
    .toSource();

  return withoutImport;
};
```

Bundling

Why do we want to bundle our code?

Bundling improves the developer experience. And it also benefits users.



To create a single executable file

A single file, especially if statically linked, is much easier to distribute and to install.



To serve a single JavaScript file

Serving only a single file reduces the overhead and improves the user experience.

The three horsemen

These are the mostly used bundlers, each with pros and cons.



Webpack

Hardest

Slower



Rollup

Easy

Slow



Parcel

Easiest

Fast

Is webpack still worth it?



Are there any better alternatives?



ESBuild



SWC



Vite



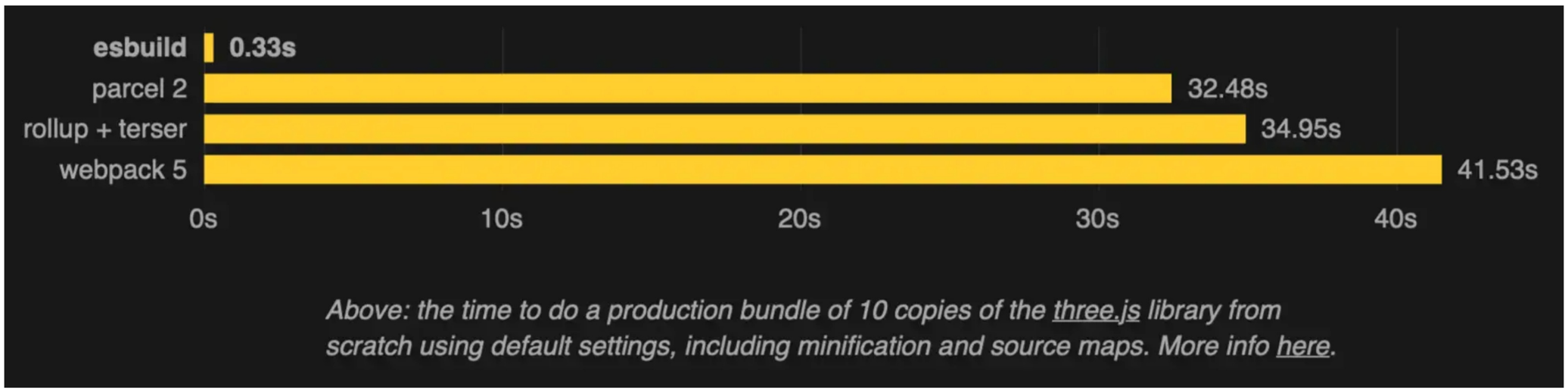
SnowPack

ESBuild



How fast ESBuild is?

A lot, especially when comparing with webpack.



Ok, it's fast. What about configuration?

ESBuild CLI is definitely easy to use.

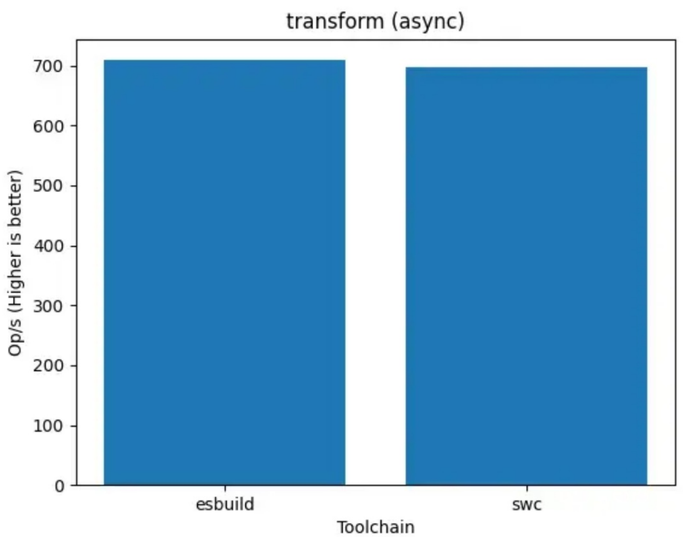


SWC

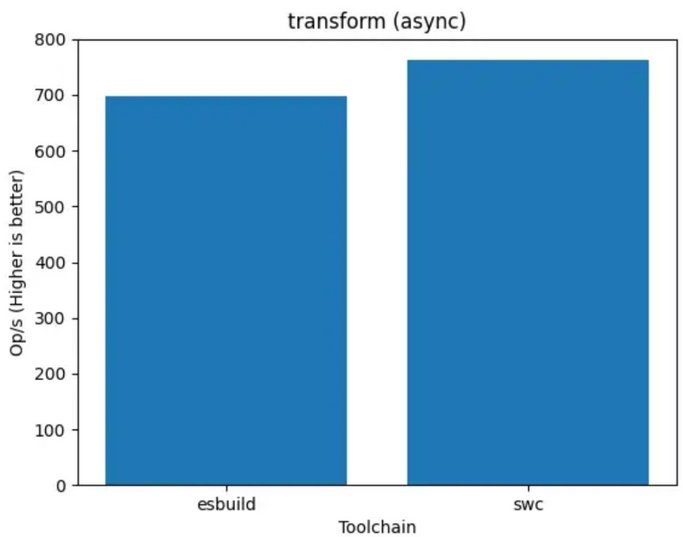


How does it compare to ESBuild?

SWC is sometimes faster than ESBuild.



ES2019



ES2020

An example of SWC configuration

```
{
  "jsc": {
    "parser": {
      "syntax": "ecmascript",
      "jsx": false,
      "dynamicImport": false,
      "privateMethod": false,
      "functionBind": false,
      "exportDefaultFrom": false,
      "exportNamespaceFrom": false,
      "decorators": false,
      "decoratorsBeforeExport": false,
      "topLevelAwait": false,
      "importMeta": false
    },
    "transform": null,
    "target": "es5",
    "loose": false,
    "externalHelpers": false,
    // Requires v1.2.50 or upper and requires target to be es2016 or upper.
    "keepClassNames": false
  }
}
```

SWC can run in a browser thanks to WASM



Vite



A little insights into Vite



Complete developer experience
It provides a webserver with Hot Module Reload support.



A bundler that doesn't actually bundle
It completely leverages ECMAScript Modules.



Still very fast
We will talk about this is in a bit.

Vite leverages existing tools

ESBuild and Rollup are used under the hood.



Snowpack



And now something completely different™

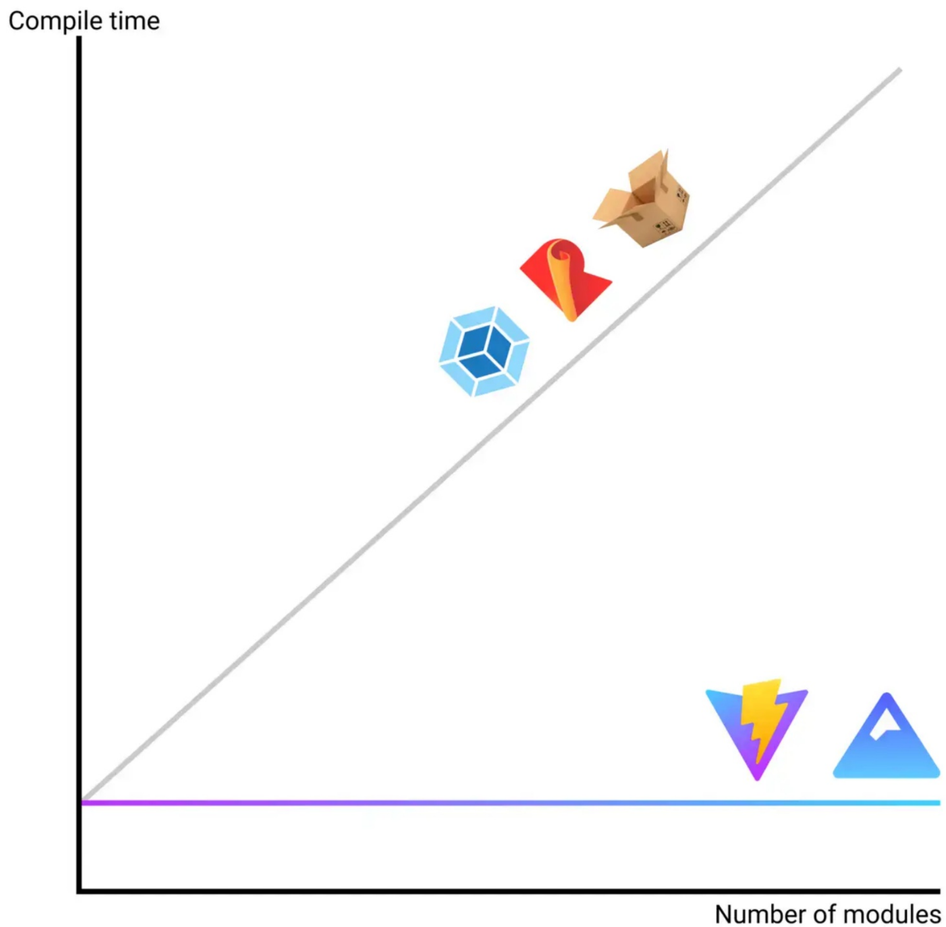
Skypack used snowpack to load module in the browsers without a bundler.

```
/*
 * Skypack CDN - canvas-confetti@1.4.0
 *
 * Learn more:
 *   📙 Package Documentation: https://www.skypack.dev/view/canvas-confetti
 *   📘 Skypack Documentation: https://www.skypack.dev/docs
 *
 * Pinned URL: (Optimized for Production)
 *   ▶ Normal: https://cdn.skypack.dev/pin/canvas-confetti@v1.4.0-
POmgSMO0U5q84otJfYlN/mode=imports/optimized/canvas-confetti.js
 *   ⏩ Minified: https://cdn.skypack.dev/pin/canvas-confetti@v1.4.0-
POmgSMO0U5q84otJfYlN/mode=imports,min/optimized/canvas-confetti.js
 *
 */

// Browser-Optimized Imports (Don't directly import the URLs below in your application!)
export * from '/-/canvas-confetti@v1.4.0-POmgSMO0U5q84otJfYlN/dist=es2020,mode=imports/optimized/canvas-
confetti.js';
export {default} from '/-/canvas-confetti@v1.4.0-POmgSMO0U5q84otJfYlN/dist=es2020,mode=imports/optimized/canvas-
confetti.js';
```

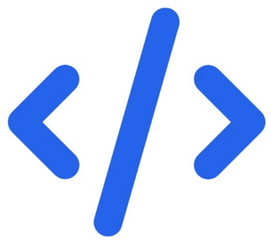
The greatest gain in new bundlers

New bundlers give O(1) compile time which is great when the number of modules grows.



Take home lessons

What can we learn from this long journey?



The future is no bundle

As ECMAScript module support increases, bundlers will not be needed anymore.



The future is bright

Big companies are sponsoring transpilers and bundlers and this ensures continuity.



Competition is good

Having multiple transpilers and bundlers competing to be the winner gives the best DX.

One last thing™

“Working hard and working smart can be two different things.”

Byron Dorgan





Thank you!

Paolo Insogna

Node.js TSC, Principal Engineer

@p_insogna

paolo.insogna@platformatic.dev

Platformatic