

BABEL

Automágicamente



Contenido

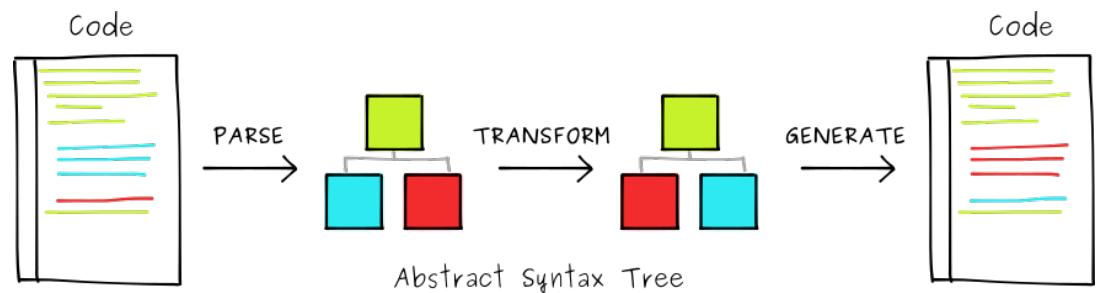
- ¿Que es Babel?
- Que es el AST (Árbol de Sintaxis Abstracto)
- Babel como transpilador
- Anatomía de un Plugin
- Demo!
- Técnicas de refactor
- Que es un Codemod
- Babel como herramienta de refactor
- Mantener un codebase al día y libre de legacy
- Demo!
- Enfrentandose a breaking changes
- Demo!



¿Que es Babel?

¿Que es Babel?

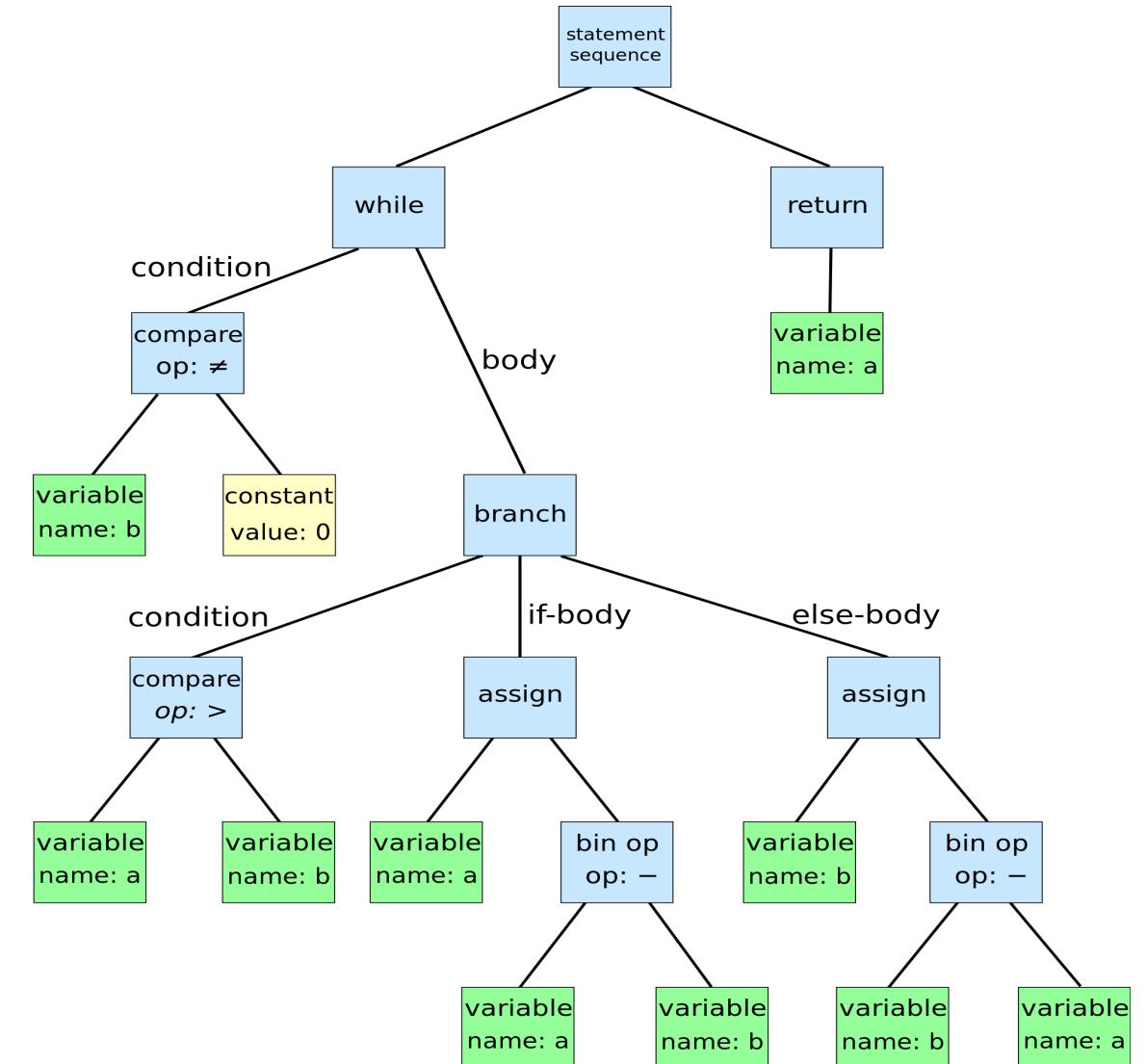
- [Babel](#) es una herramienta que se suele usar para transpilar código javascript en versiones modernas como ES6 o typescript a versiones más antiguas de javascript para compatibilidad con navegadores.
- Primero, el [parser](#) de Babel convierte el código JavaScript a un [Árbol de sintaxis abstracto \(AST\)](#).
- A continuación, el [traverser](#) de Babel, usa ese AST para explorar y modificar el código utilizando la config que previamente le proporcionamos.
- Y por último, el [generator](#) de Babel traducira este AST a código otra vez, legible por el navegador.



Que es el AST

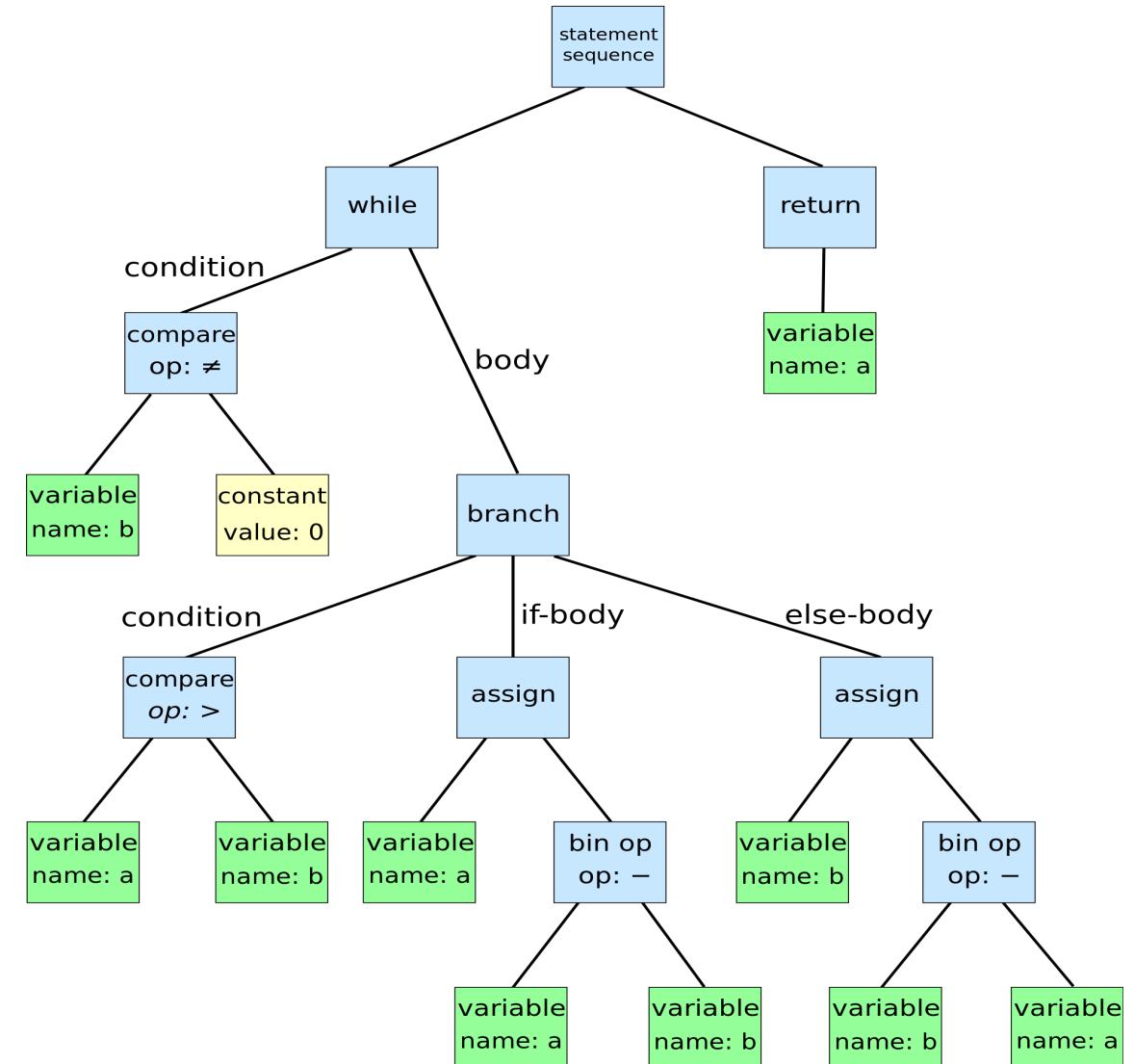
“ An abstract syntax tree is a tree representation of the abstract syntactic structure of source code written in a programming language. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. ”

- [Wikipedia](#)



Que es el AST

- Abstract Syntax Trees son estructuras de datos muy utilizadas en compiladores, generalmente como resultado de la fase de análisis de sintaxis.
- AST se usa intensamente durante el análisis semántico, donde el compilador verifica el uso correcto de los elementos del programa y el lenguaje.
- Frente al código fuente, AST no incluye signos de puntuación y delimitadores no esenciales (llaves, punto y coma, paréntesis, etc.).



AST en JavaScript

Todos estos proyectos usan AST y puede que no lo supierais



webpack



Prettier



ESLint

TS TypeScript

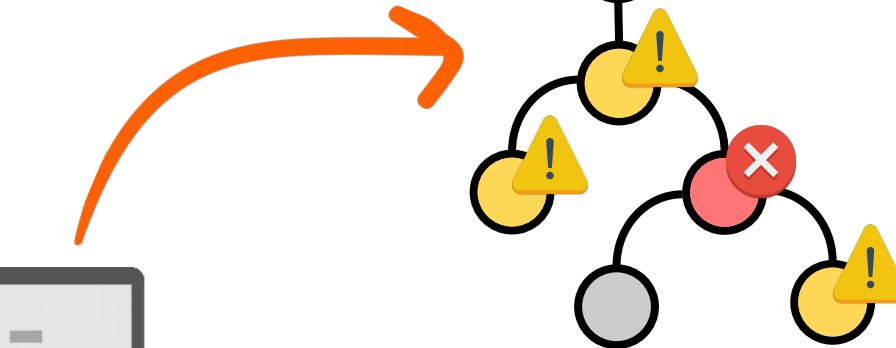


AST en JavaScript

Las reglas de Eslint están escritas con AST para analizar tu código

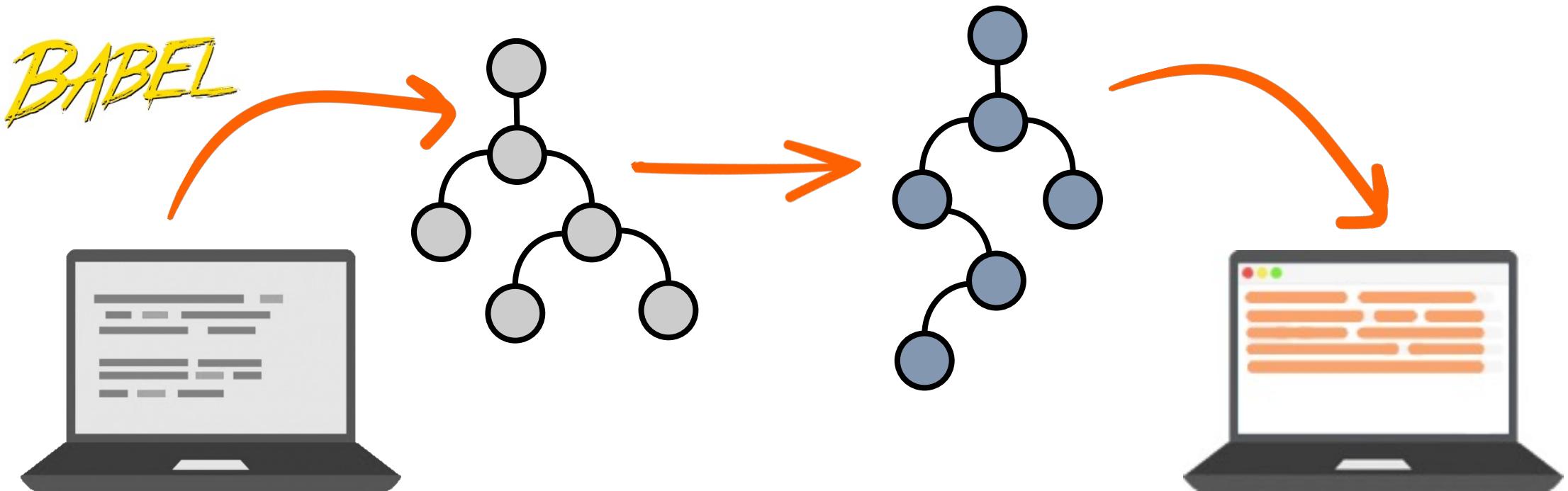


ESLint



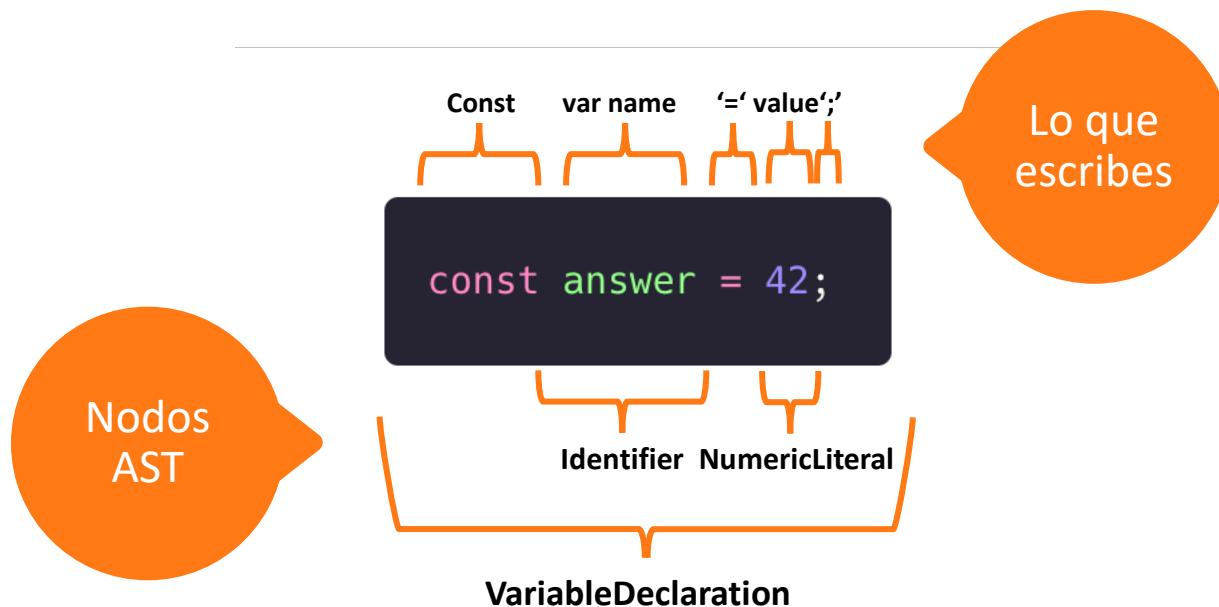
AST en JavaScript

Babel parsea tu código a AST, hace transformaciones al árbol e imprime el código de vuelta.

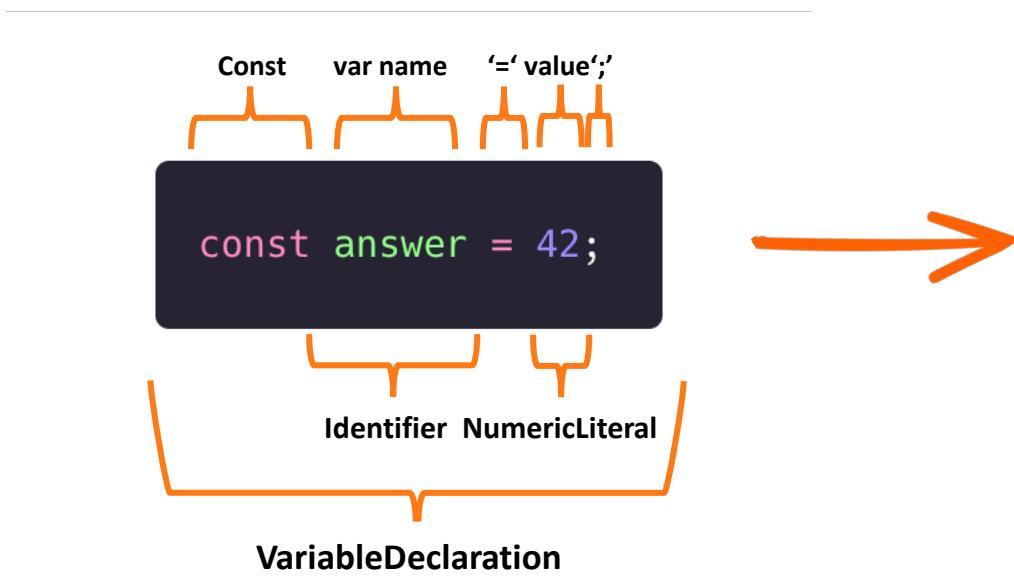


Cómo AST parsea el código

- Cada nodo del árbol representa una parte del código fuente.



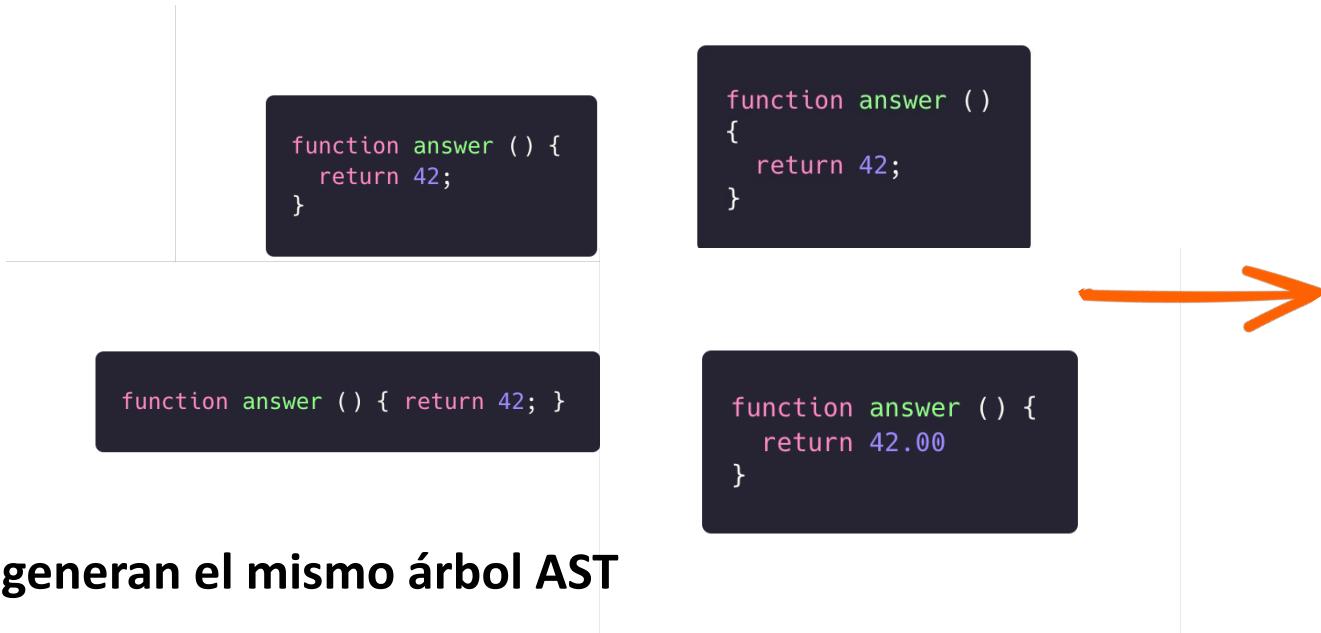
Qué pinta tiene un arbol AST



```
{
  "type": "VariableDeclaration",
  "kind": "const",
  "declarations": [
    {
      "type": "Identifiers",
      "id": {
        "type": "VariableDeclarator",
        "name": "answer"
      },
      "init": {
        "type": "NumericLiteral",
        "value": 42,
        "raw": "42"
      }
    }
}
```

Qué pinta tiene un arbol AST

Consideremos estos 4 ejemplos:



```
{  
    "type": "FunctionDeclaration",  
    "id": {  
        "type": "Identifier",  
        "name": "answer",  
    },  
    "params": [],  
    "body": [{  
        "type": "BlockStatement",  
        "body": [  
            {"type": "ReturnStatement",  
                "argument": {  
                    "type": "Literal",  
                    "value": 42,  
                    "raw": "42" // or "42.00"  
                }  
            }  
        ]  
    }]  
}
```

Ast explorer

Sandbox AST, herramienta online para visualizar y depurar el arbol AST con diferentes parseadores y transformadores.

The screenshot shows the AST Explorer interface. At the top, there are tabs for 'AST Explorer', 'Snippet' (with a file icon), 'JavaScript' (with a gear icon), and 'Transformer'. Below these are buttons for 'Parser' (@babel/parser) and 'Transformer' (@babel/transformer). The main area has two panes: 'Tree' (selected) and 'JSON'. In the 'Tree' pane, a code snippet is shown:

```
1 const n = 1;
```

The tree view shows the structure of this code. A node for 'n' is highlighted in yellow, indicating it is selected. The node details are as follows:

- id: Identifier {
 type: "Identifier"
 start: 6
 end: 7
 + loc: {start, end, filename, identifierName}
 name: "n"
}
- + init: NumericLiteral {type, start, end, loc, extra, ... +1}

The 'kind' of the node is "const". There are also sections for 'directives' and 'comments' which are currently empty.

Below the tree view, there is a code editor with a 'Prettier' button. The code in the editor is:

```
1 export default function (babel) {  
2   const { types: t } = babel;  
3  
4   return {  
5     name: "ast-transform", // not required  
6     visitor: {  
7       Identifier(path) {  
8         if (path.isIdentifier({ name: 'n' })) {  
9           path.node.name = 'x';  
10        }  
11      }  
12    },  
13  };  
14}  
15
```

On the right side of the interface, there is a URL bar with the address:

<https://astexplorer.net/#/gist/3d9ad4915461bde47b0e2673c1904e27/a98242aa9fa3fc97fee84d2de1a327b46a09304d>

Babel como transpilador

¿Como usar Babel para transformar Código?

- Necesitamos instalar [@babel/core](#) y usar sus dependencias:
@babel/parser, @babel/traverse
y @babel/generator.
- Código -> (parser) -> AST -> (traverse) ->
AST transformado -> (generator) ->
Código transformado.

```
import { parse } from '@babel/parser';
import traverse from '@babel/traverse';
import generate from '@babel/generator';

const code = 'const n = 1';

// parse the code -> ast
const ast = parse(code);

// transform the ast
traverse(ast, {
  enter(path) {
    if (path.isIdentifier({ name: 'n' })) {
      path.node.name = 'x';
    }
  },
});

// generate code <- ast
const output = generate(ast, code);
console.log(output.code); // 'const x = 1;'
```

Anatomia de un Plugin

- Solo con definir una function que devuelva un objeto respetando el Api que define Babel:
“{visitor:{ NodeType(path) { /*transformations*/ } }}” es suficiente. Ya estamos preparados para usar nuestro plugin. Podemos empaquetarlo y publicarlo en npm o directamente usarlo.

```
import babel from '@babel/core';

const code = 'const n = 1';

const output = babel.transformSync(code, {
  plugins: [
    function myCustomPlugin() {
      return {
        visitor: {
          Identifier(path) {
            if (path.isIdentifier({ name: 'n' })) {
              path.node.name = 'x';
            }
          },
        },
      };
    ],
  );
};

console.log(output.code); // 'const x = 1;'
```

LIVE DEMO

WHAT COULD GO WRONG?

memegenerator.net

Codemods

Técnicas de refactor

- Find-and-replace (**en un directorio completo**) para modificar ficheros de código.



Técnicas de refactor

- Find and replace (~~en un directorio completo~~) para modificar ficheros de código.
- Script con expresiones regulares usando grupos de captura.



Técnicas de refactor

- Find and replace (~~en un directorio completo~~) para modificar ficheros de código. 
- Script con expresiones regulares usando grupos de captura.



Técnicas de refactor

- Find and replace (~~en un directorio completo~~) para modificar ficheros de código. 
- Script con expresiones regulares usando grupos de captura. 
- Aquí es donde entran los “codemods” y AST

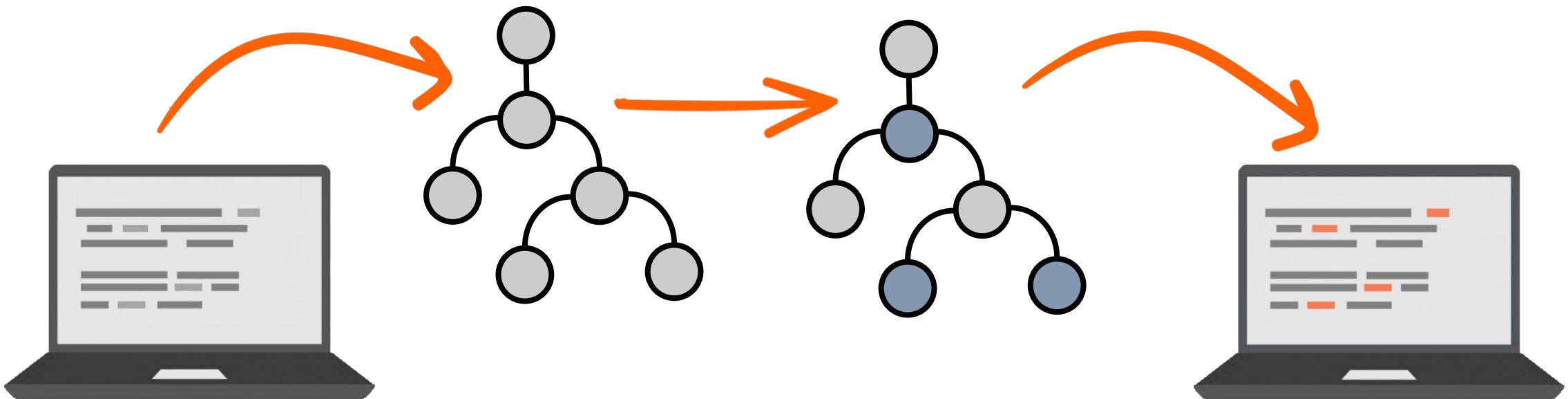


Que es un Codemod

- “Código que modifica código”
- Se suelen usar para:
 - Actualizar código fuente manteniendo el “code style” y convenciones.
 - Hacer cambios en nuestro código cuando se modifica un API.
 - Auto-fix del código fuente cuando un paquete público introduce breaking changes.

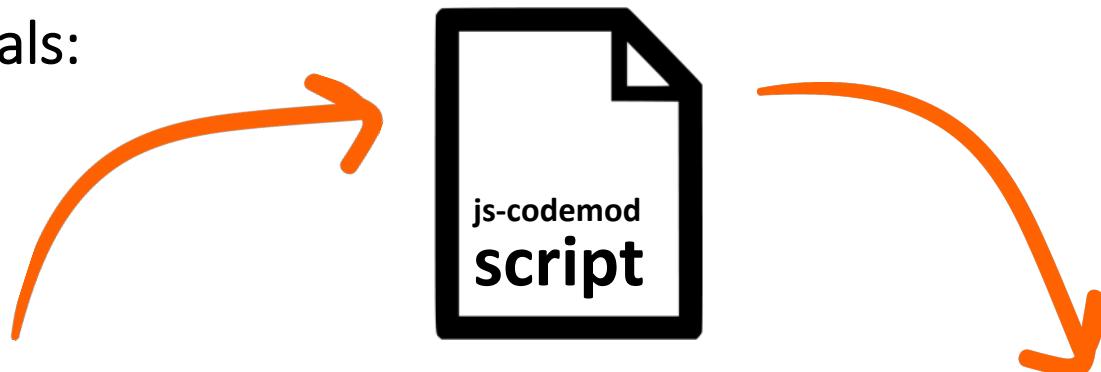
Babel como herramienta de refactor

Vamos a usar **codemods** para parsear nuestro código a AST, hacer transformaciones en el árbol sintáctico y volver a imprimir nuestro código al fichero original con **Recast sin perder el “code style”**



Casos de uso simples

- template-literals:

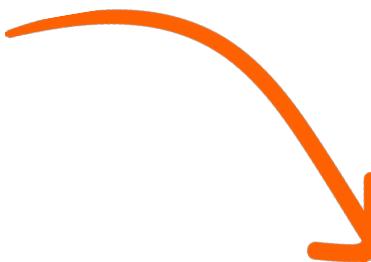
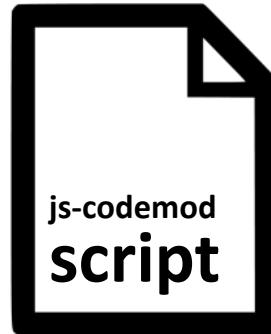
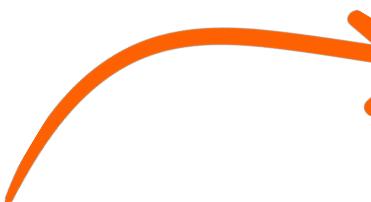


```
const belong = 'anywere ' + welcomeHome;
```

```
const belong = `anywere ${welcomeHome}`;
```

Casos de uso simples

- object-shorthand:

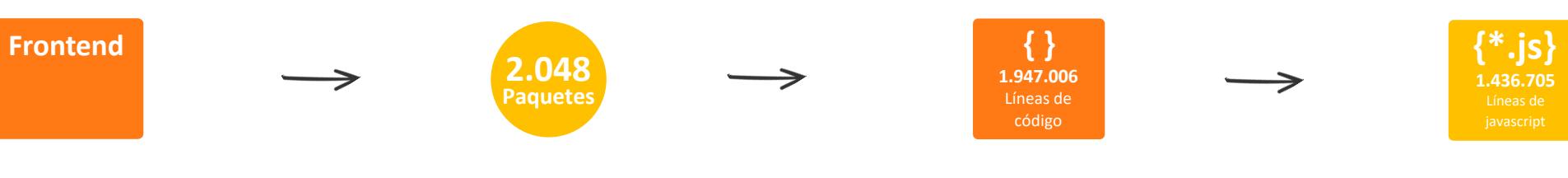


```
const things = {  
  belong: belong,  
  anywhere: function() {},  
};
```

```
const things = {  
  belong,  
  anywhere() {},  
};
```

Manteniendo código legacy

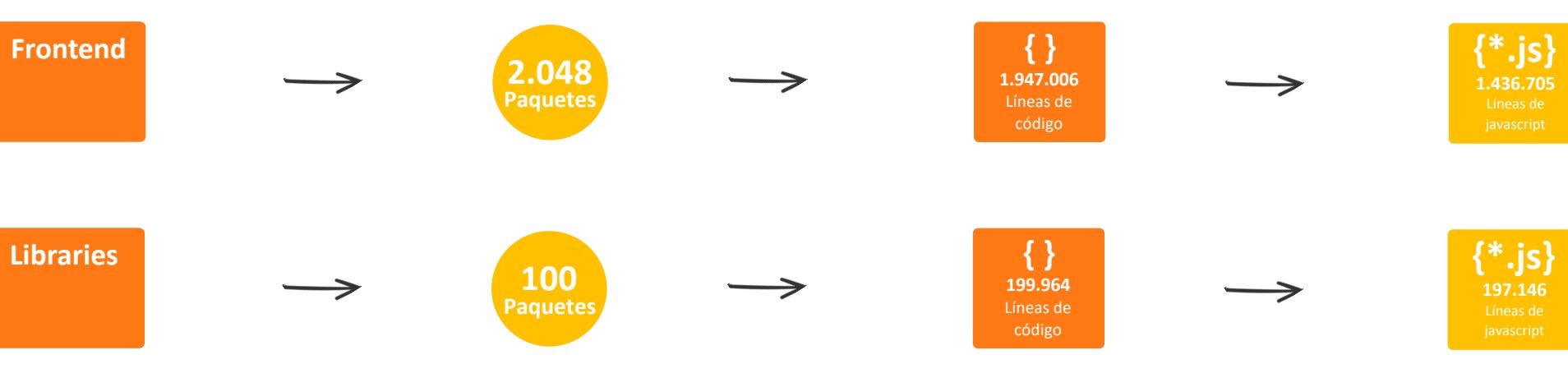
Ejemplo monorepo javascript monolítico



*node_modules excluido.

Manteniendo código legacy

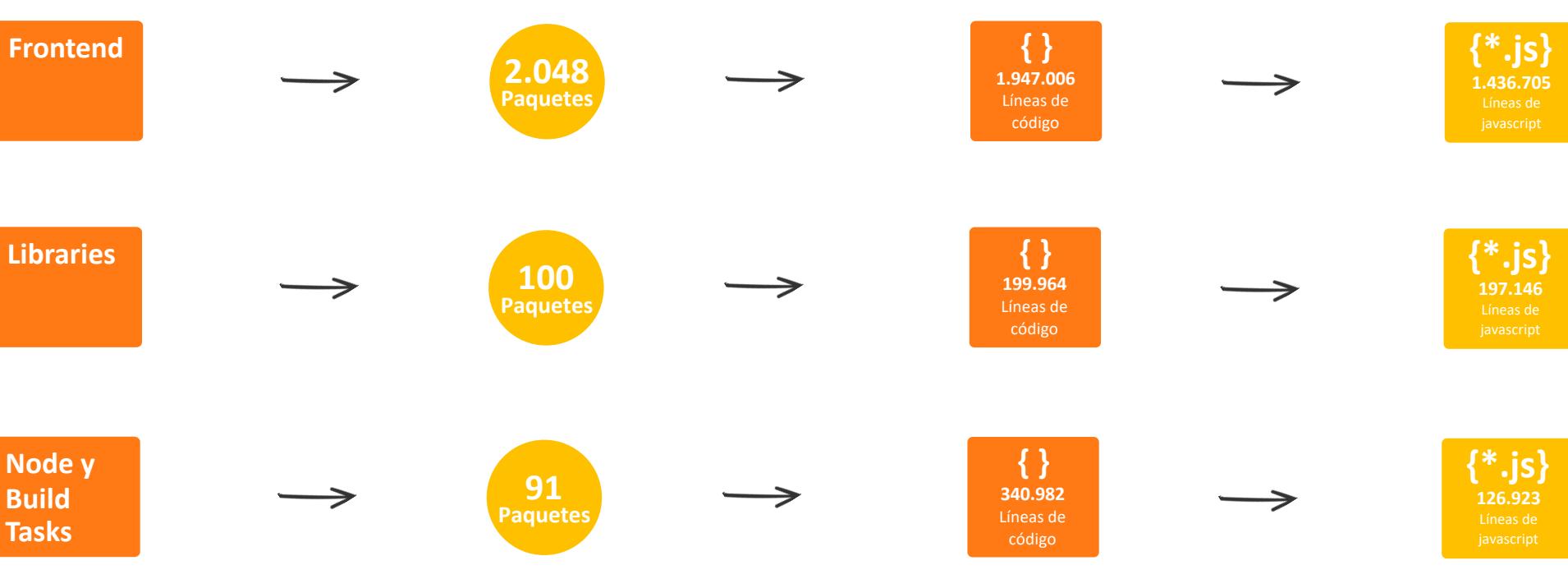
Ejemplo monorepo javascript monolítico



*node_modules excluido.

Manteniendo código legacy

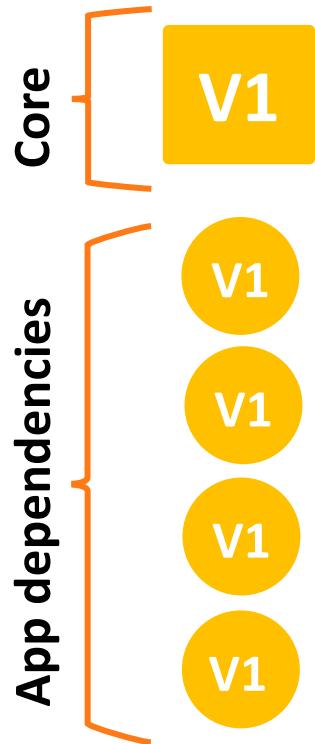
Ejemplo monorepo javascript monolítico



*node_modules excluido.

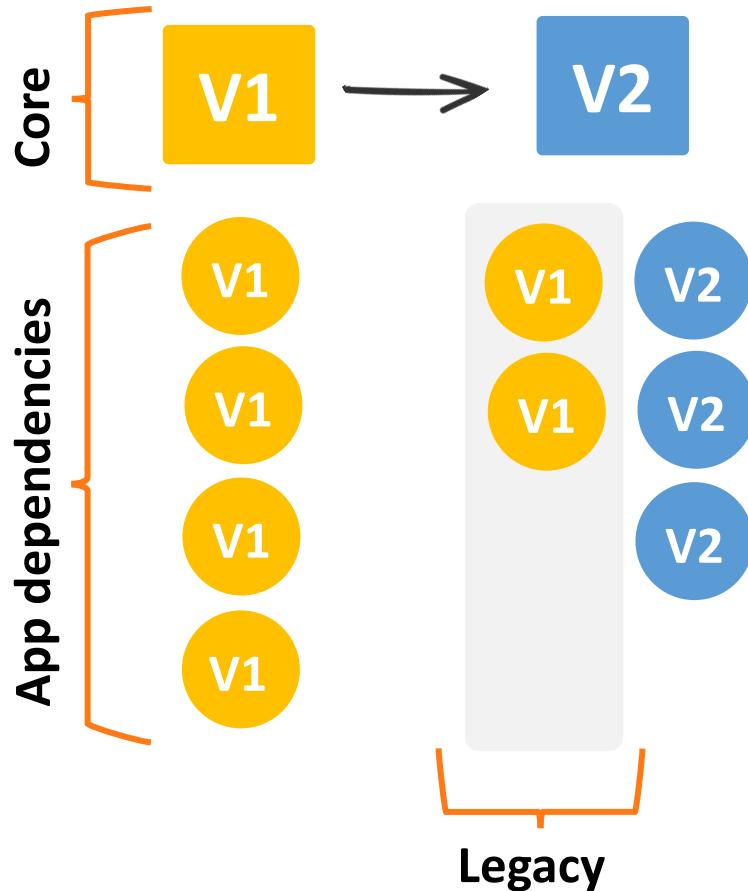
Manteniendo código legacy

Ejemplo monorepo javascript monolítico



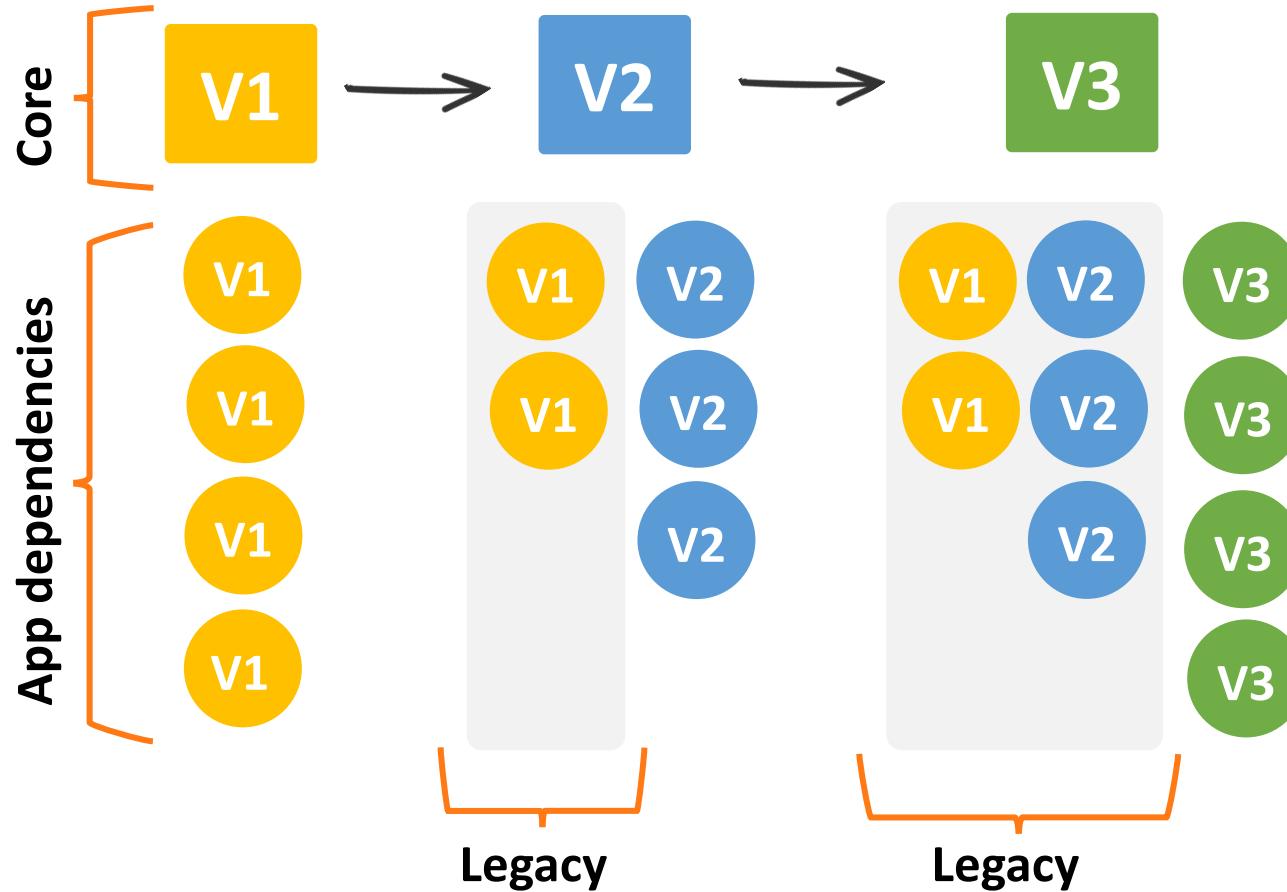
Manteniendo código legacy

Ejemplo monorepo javascript monolítico



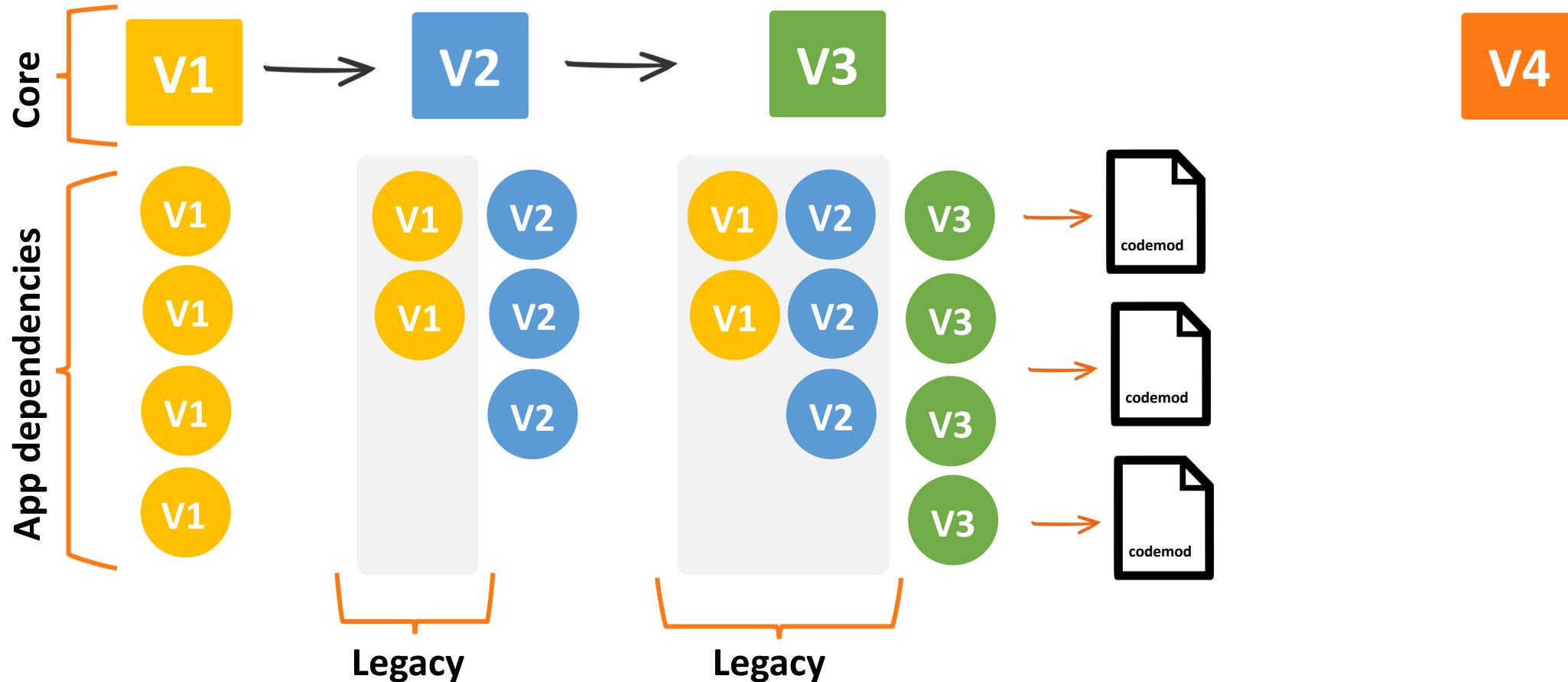
Manteniendo código legacy

Ejemplo monorepo javascript monolítico



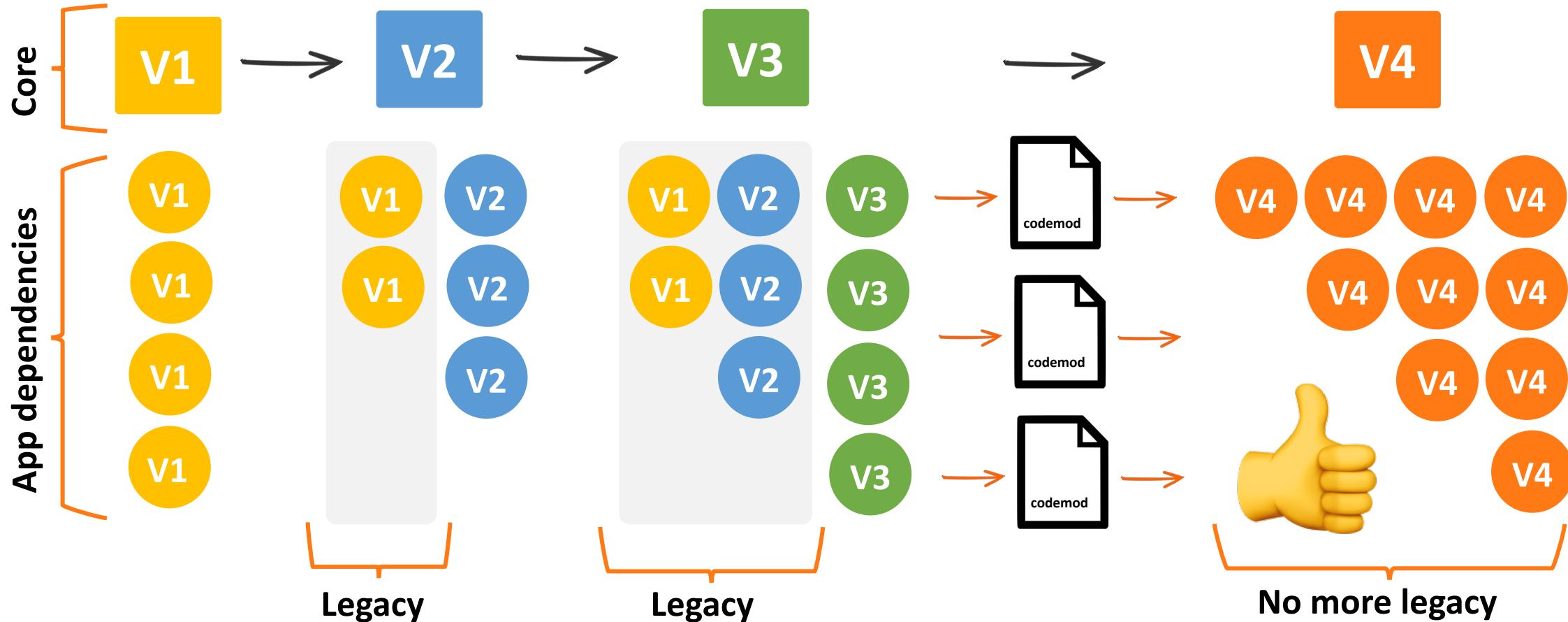
Manteniendo código legacy

Ejemplo monorepo javascript monolítico



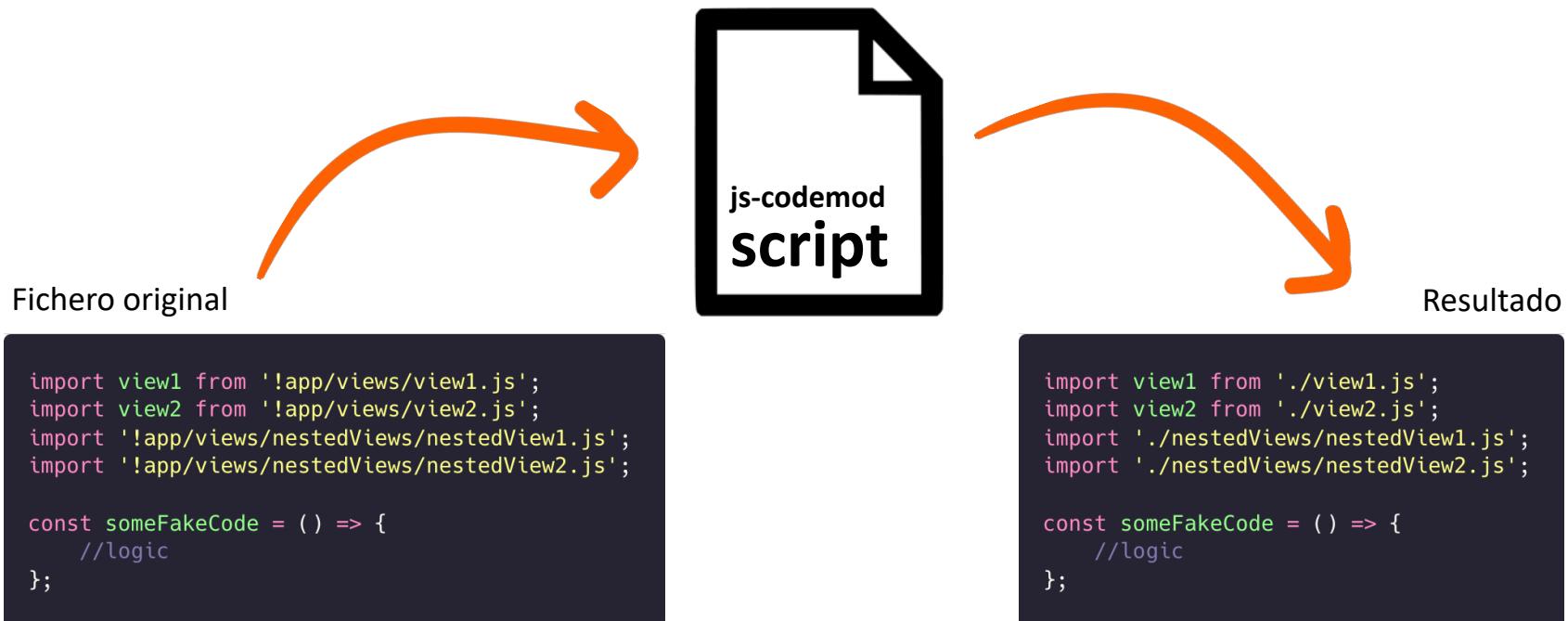
Manteniendo código legacy

Ejemplo monorepo javascript monolítico



Manteniendo código legacy

Imaginemos una app monolítica que hay que mantener, un refactor en las tareas de construcción, implica que el código fuente deje de tener rutas absolutas con alias y pase a tener paths relativos a los recursos:



Manteniendo código legacy

Imaginemos una app monolítica que hay que mantener, un refactor en las tareas de construcción, implica que el código fuente deje de tener rutas absolutas con aliases y pase a tener paths relativos a los recursos:

```
const pathToRelative = (importPath, filePath, currentAppPath) => {
  importPath = importPath.replace('!app', '');
  const currentPathFolder = path.dirname(filePath);
  const relativePathToFile = path.relative(currentPathFolder, path.resolve(currentAppPath, importPath));
  return `${relativePathToFile.startsWith('.') ? '' : './'}${relativePathToFile.replace(/\//g, '/')}`;
};

const removeAppAliases = filePaths => {
  filePaths.forEach(filePath => {
    const pathParts = filePath.replace(/\//g, '/').split('/');
    const currentAppPath = path.resolve(pathParts.slice(0, pathParts.indexOf('src') + 1).join('/'));
    const source = fse.readFileSync(filePath, 'utf8');
    const ast = parse(source, {
      parser: {
        parse: (source) =>
          parseSync(source, {
            sourceType: 'unambiguous',
            filename: filePath,
            parserOpts: {
              tokens: true,
            },
          }),
      },
    });
    traverse(ast, {
      StringLiteral: function(currentPath) {
        if (
          currentPath.node.value.startsWith('!app') &&
          currentPath.findParent((path) => path.isImportDeclaration())
        ) {
          currentPath.replaceWith(t.stringLiteral(pathToRelative(currentPath.node.value, filePath,
          currentAppPath)));
        }
      },
    });
    const output = filePath.replace('app', 'outputApp');
    fse.ensureFileSync(output);
    fse.writeFileSync(output, print(ast, { quote: 'single', useTabs: false, lineTerminator: '\n' }).code);
  });
};
```

SO YOU THINK

A LIVE DEMO IS WISE

memegenerator.net

Enfrentandose a breaking changes

Enfrentandose a breaking changes

¿Para que usan codemods algunas empresas?

Facebook publica codemods para ayudar a migrar React de versiones anteriores:

<https://github.com/reactjs/react-codemod>

Gatsby hace lo mismo:

<https://www.gatsbyjs.com/plugins/gatsby-codemods/>

<https://github.com/gatsbyjs/gatsby/pull/28112>

Next.js nos prove de codemods cuando una feature queda deprecada:

<https://nextjs.org/docs/advanced-features/codemods>

Enfrentandose a breaking changes

Pongamos un ejemplo: En ING tenemos proyecto con `lit-element`.

Tenemos un catalogo de componentes compartido llamado `ing-web`.

Hace unos meses, un componente de icono introdujo un breaking change para evitar posibles ataques xss:

```
// old (deprecated) approach
function render() {
  return html`
```

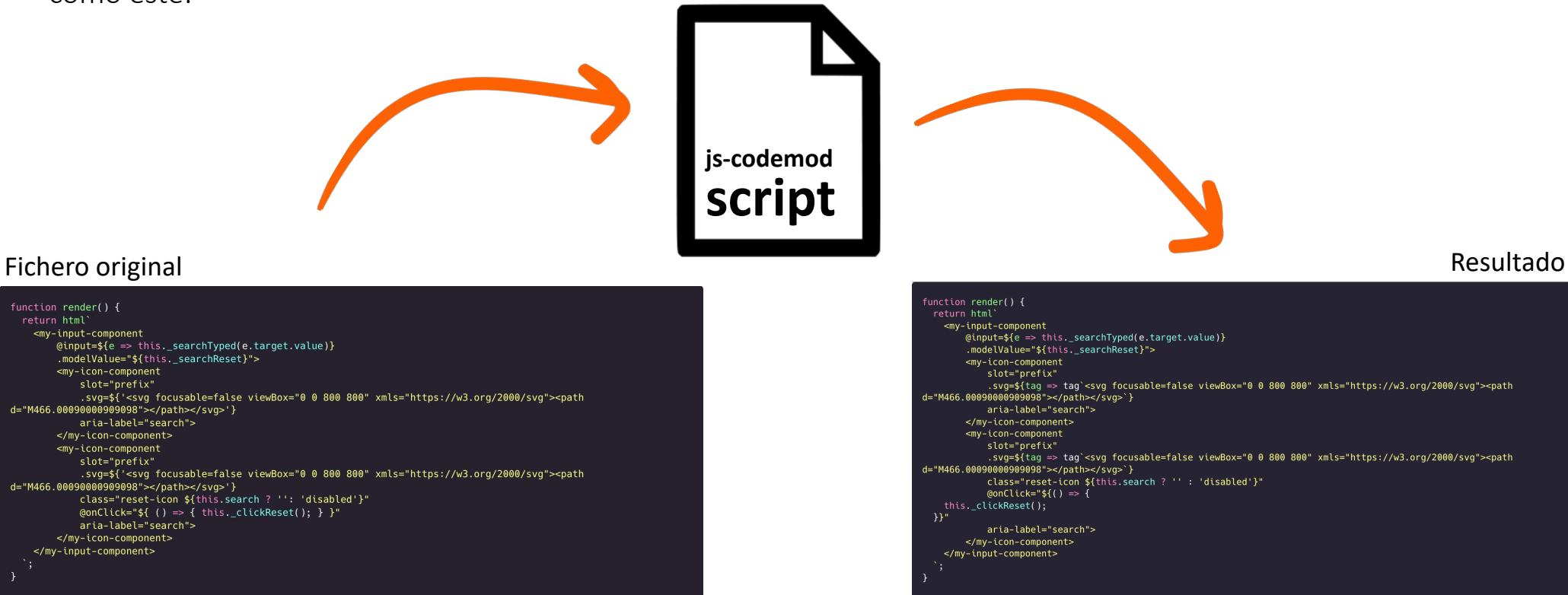
Enfrentandose a breaking changes

Imaginemos que tenemos un proyecto con multiples usos de `ing-icon`. Quieres actualizar la versión de `ing-web`, pero para eso hay que actualizar todos los atributos `.svg` antes.

```
function render() {
  return html`
```

Enfrentandose a breaking changes

Un aproach para que los equipos no tuvieran que refactorizar a mano sus proyectos habría sido distribuir un codemod como este:



Enfrentándose a breaking changes

Un aproach para que los equipos no tuvieran que refactorizar a mano sus proyectos habría sido distribuir un codemod como este:

```
export default function (babel) {
  const { types: t } = babel;

  return {
    name: "ast-transform", // not required
    visitor: {
      StringLiteral(path) {
        if (
          path.node.value.startsWith('<svg') &&
          path.findParent((path) => path.isTemplateLiteral()) &&
          path.findParent((path) => path.isTemplateLiteral()).findParent((path) =>
path.isTaggedTemplateExpression()).node.tag.name === 'html'
        ) {
          const svgLiteral = t.templateLiteral([t.templateElement({cooked:path.node.value, raw:path.node.value},
false)], []);
          const taggedTmplt = t.taggedTemplateExpression(t.identifier('tag'), svgLiteral);
          path.replaceWith(t.arrowFunctionExpression([t.identifier('tag')], taggedTmplt));
        }
      }
    }
  }
}
```

<https://astexplorer.net/#/gist/0d3a4c7fba27f094c36e167de3c181b5/373c260a4b9b0eb30fad308b7bcb11a4b2357633>



LIVE DEMO!

Recursos

<https://babeljs.io/docs/en/>

<https://github.com/jamiebuilds/babel-handbook>

<https://astexplorer.net/>

<https://github.com/benjamn/recast>

THANK YOU

COMPUTER MAN!

memegenerator.net

