Babel:
A refactoring tool

NICOLÒ RIBAUDO

Babel team

BABEL

What is Babel?

What is Babel?

Babel is a JavaScript compiler

It’s a JavaScript to JavaScript compiler

```
player.level ??= 70_000;
```

```
(_player$level = player.level) != null
    ? _player$level
    : player.level = 70000;
```

@NicoloRibaudo

HOLY JS/...

Москва
8-9 ноября, 2019

What is Babel?

Babel is a JavaScript compiler

5

INNOPOLIS

ManyChat

Deutsche Bank

КОНТУР

Райффайзен БАНК

1:18 / 59:40

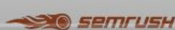
@NicoloRibaudo

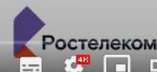
@babel/how-to

https://www.youtube.com/watch?v=UeVq_U5obnE

@NicoloRibaudo

5





1:18 / 59:40

What is Babel?

What is Babel?

Babel is a *customizable* JavaScript compiler

So... what can I do?

So... what can I do?

- Compile modern ECMAScript syntax to older (and more supported) syntax
- Compile TypeScript, Flow and JSX to plain JavaScript

So... what can I do?

- Compile modern ECMAScript syntax to older (and more supported) syntax
- Compile TypeScript, Flow and JSX to plain JavaScript

- Minify your code

So... what can I do?

- Compile modern ECMAScript syntax to older (and more supported) syntax
- Compile TypeScript, Flow and JSX to plain JavaScript

- Minify your code

- Statically evaluate parts of your program

So... what can I do?

- Compile modern ECMAScript syntax to older (and more supported) syntax
- Compile TypeScript, Flow and JSX to plain JavaScript

- Minify your code

- Statically evaluate parts of your program

- Perform static analysis on your code

So... what can I do?

- Compile modern ECMAScript syntax to older (and more supported) syntax
- Compile TypeScript, Flow and JSX to plain JavaScript

- Minify your code

- Statically evaluate parts of your program

- Perform static analysis on your code

- Run one-time transforms and automate refactorings

So... what can I do?

- Run one-time transforms and automate refactorings

Codemods

One-time transforms and automate refactorings

Codemods

Codemods

Refactors... happen.

Codemods

Refactors... happen.

Sometimes ...

... they are self-contained, without impacting other parts or your program

And they take ...

... a few hours

Codemods

Refactors... happen.

Sometimes ...

... they are self-contained, without impacting other parts or your program

... they affects your whole codebase

And they take ...

... a few hours

... a week

Codemods

Refactors... happen.

Sometimes ...	And they take ...
... they are self-contained, without impacting other parts or your program	... a few hours
... they affects your whole codebase	... a week
... you need to introduce changes accross applications maintained by different teams	... many weeks, maybe months months

Codemods

Refactors... happen.

Sometimes ...	And they take ...
... they are self-contained, without impacting other parts or your program	... a few hours
... they affects your whole codebase	... a week
... you need to introduce changes accross applications maintained by different teams	... many weeks, maybe months months
... you maintain a popular library that is used by many users outside of your company	... ?

@NicoloRibaudo

What can you use codemods for?

~~What can you use codemods for?~~
What did we use them for in Babel?

~~What can you use codemods for?~~
What did we use them for in Babel?

- Migrate our tests to Jest 🔗

```
  it("options", function() {
    const string = "original;";
    assert.deepEqual(
      newTransform(string).ast,
    expect(newTransform(string).ast).toEqual(
      babel.transform(string, { ast: true }).ast,
    );
    assert.equal(newTransform(string).code, string);
    expect(newTransform(string).code).toBe(string);
  });
```

@NicoloRibaudo

~~What can you use codemods for?~~
What did we use them for in Babel?

- Migrate our tests to Jest 🔗
- Remove unused `catch` bindings 🔗

```diff
  try {
    fs.statSync(filename);
    return true;
- } catch (err) {
+ } catch {
    reutrn false;
  }
}
```

~~What can you use codemods for?~~

What did we use them for in Babel?

- Migrate our tests to Jest 🔗
- Remove unused `catch` bindings 🔗
- Remove the `resolve` dependency 🔗

```
import buildDebug from "debug";
import resolve from "resolve";
import path from "path";
@@ -96,14 +95,18 @@ function resolveStandardizedName(
  try {
    return resolve.sync(standardizedName, { basedir: dirname });
    return require.resolve(standardizedName, {
      paths: [dirname],
    });
  } catch (e) {
```

@NicoloRibaudo

~~What can you use codemods for?~~

What did we use them for in Babel?

- Migrate our tests to Jest 🔗
- Remove unused `catch` bindings 🔗
- Remove the `resolve` dependency 🔗
- Migrate from Flow to TypeScript 🔗

```
export type TemplateOpts = {|
  parser: ParserOpts,
  placeholderWhitelist: Set<string> | void,
  placeholderPattern: RegExp | false | void,
  preserveComments: boolean | void,
  syntacticPlaceholders: boolean | void,
|};
export type TemplateOpts = {
  parser: ParserOpts;
  placeholderWhitelist?: Set<string>;
  placeholderPattern?: RegExp | false;
  preserveComments?: boolean;
  syntacticPlaceholders?: boolean;
};
```

@NicoloRibaudo

~~What can you use codemods for?~~
What other companies use them for?

~~What can you use codemods for?~~
What other companies use them for?

- Facebook publishes codemods to migrate away from legacy React versions 🔗

@NicoloRibaudo

~~What can you use codemods for?~~

What other companies use them for?

- Facebook publishes codemods to migrate away from legacy React versions 🔗

- Gatsby provides codemods to migrate to new versions 🔗 🔗

~~What can you use codemods for?~~

What other companies use them for?

- Facebook publishes codemods to migrate away from legacy React versions 🔗
- Gatsby provides codemods to migrate to new versions 🔗 🔗
- Next.js provides codemods to upgrade when a feature is deprecated 🔗

@NicoloRibaudo

Compilers vs Codemods

Compilers vs Codemods

Compilers vs Codemods

The usual Babel transforms are based on *strictly defined semantics*

Compilers vs Codemods

The usual Babel transforms are based on *strictly defined semantics*

```
person?.address.city?.size
```

```
person == null
  ? void 0
  : (_tmp = person.address.city) == null
    ? void 0
    : _tmp.size
```

Compilers vs Codemods

The usual Babel transforms are based on *strictly defined semantics*

```
person?.address.city?.size
```

```
person == null
  ? void 0
  : (_tmp = person.address.city) == null
    ? void 0
    : _tmp.size
```

Codemods are based on *assumptions* about your coding style

Compilers vs Codemods

The usual Babel transforms are based on *strictly defined semantics*

```
person?.address.city?.size
```

```
person == null
  ? void 0
  : (_tmp = person.address.city) == null
    ? void 0
    : _tmp.size
```

Codemods are based on *assumptions* about your coding style

```
person
  && person.address.city
  && person.address.city.size
```

```
person?.address.city?.size
```

@NicoloRibaudo

Compilers vs Codemods

The usual Babel transforms are based on *strictly defined semantics*

```
person?.address.city?.size
```

```
person == null
  ? void 0
  : (_tmp = person.address.city) == null
    ? void 0
    : _tmp.size
```

Codemods are based on *assumptions* about your coding style

```
person
  && person.address.city
  && person.address.city.size
```

```
person?.address.city?.size
```

```
false?.address.city?.size
```

@NicoloRibaudo

Compilers vs Codemods

The usual Babel transforms are based on *strictly defined semantics*

Codemods are based on *assumptions* about your coding style

Compilers vs Codemods

The usual Babel transforms are based on *strictly defined semantics*

They need to be precise and infallible.

Developers should be able to trust the compiler output without checking it every time.

Codemods are based on *assumptions* about your coding style

Compilers vs Codemods

The usual Babel transforms are based on *strictly defined semantics*

Codemods are based on *assumptions* about your coding style

They need to be precise and infallible.

Developers should be able to trust the compiler output without checking it every time.

They need to work *in most cases*, but it's not necessary that they work *always*.

Compilers vs Codemods

The usual Babel transforms are based on *strictly defined semantics*

Codemods are based on *assumptions* about your coding style

They need to be precise and infallible.

Developers should be able to trust the compiler output without checking it every time.

They need to work *in most cases*, but it's not necessary that they work *always*.

They rely on human intervention.

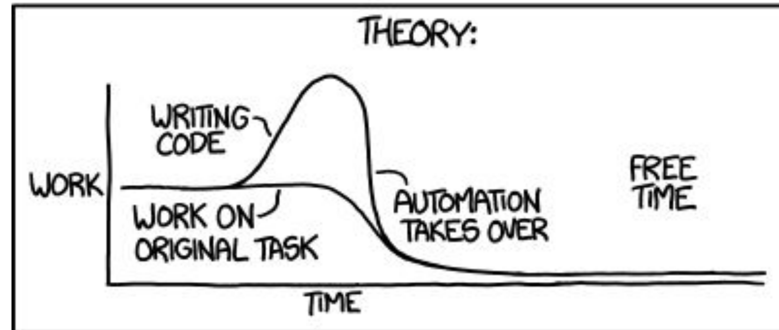
@NicoloRibaudo

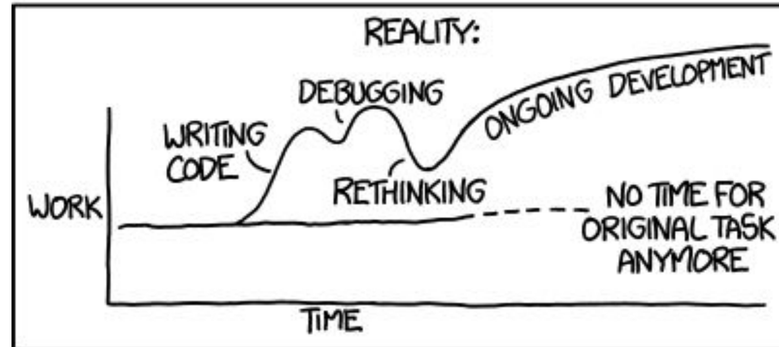
Human-assisted automated transforms

Why don't we just create the perfect codemod?

Human-assisted automated transforms

"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"





https://xkcd.com/1319/

Human-assisted automated transforms

```
const fs = require("fs");
const { join } = require("path");
```

Human-assisted automated transforms

```
const fs = require("fs");
const { join } = require("path");
```

```
import fs from "fs";
import { join } from "path";
```

Human-assisted automated transforms

```
const fs = require("fs");
const { join } = require("path");

exports.readLocal = (file) => {
  return fs.readFileSync(
    join(__dirname, file)
  );
};
```

```
import fs from "fs";
import { join } from "path";
```

@NicoloRibaudo

Human-assisted automated transforms

```js
const fs = require("fs");
const { join } = require("path");

exports.readLocal = (file) => {
  return fs.readFileSync(
    join(__dirname, file)
  );
};
```

```js
import fs from "fs";
import { join } from "path";

export const readLocal = (file) => {
  return fs.readFileSync(
    join(__dirname, file)
  );
}
```

@NicoloRibaudo

Human-assisted automated transforms

```
const fs = require("fs");
const { join } = require("path");

exports.readLocal = (file) => {
  return fs.readFileSync(
    join(__dirname, file)
  );
};

// "writeLocal"
exports[`${fs.write.name}Local`] =
  (file, contents) => {
    return fs.writeFileSync(
      join(__dirname, file), contents
    );
  };
```

```
import fs from "fs";
import { join } from "path";

export const readLocal = (file) => {
  return fs.readFileSync(
    join(__dirname, file)
  );
}
```

Human-assisted automated transforms

```js
const fs = require("fs");
const { join } = require("path");

exports.readLocal = (file) => {
  return fs.readFileSync(
    join(__dirname, file)
  );
};

// "writeLocal"
exports[`${fs.write.name}Local`] =
  (file, contents) => {
    return fs.writeFileSync(
      join(__dirname, file), contents
    );
  };
```

```js
import fs from "fs";
import { join } from "path";

export const readLocal = (file) => {
  return fs.readFileSync(
    join(__dirname, file)
  );
}

// "writeLocal"
export const UNKNOWN =
  (file, contents) => {
    return fs.writeFileSync(
      join(__dirname, file), contents
    );
  };
```

@NicoloRibaudo

Human-assisted automated transforms

```diff
- const fs = require("fs");
- const { join } = require("path");
+ import fs from "fs";
+ import { join } from "path";
 
- exports.readLocal = (file) => {
+ export const readLocal = (file) => {
   return fs.readFileSync(
     join(__dirname, file)
   );
- };
+ }
 
 // "writeLocal"
- exports[`${fs.write.name}Local`] =
+ export const UNKNOWN =
   (file, contents) => {
     return fs.writeFileSync(
       join(__dirname, file), contents
```

@NicoloRibaudo

Alternatives

Alternatives

`Regular Expressions`

Good for simple, self-contained refactors

Alternatives

`JSCodeshift`

The most popular tool to build JavaScript codemods

Alternatives

`JSCodeshift`

The most popular tool to build JavaScript codemods

- Provides a jQuery-like API to navigate the AST

@NicoloRibaudo

Alternatives

`JSCodeshift`

The most popular tool to build JavaScript codemods

- Provides a jQuery-like API to navigate the AST
- Includes different tools (`@babel/parser` for parsing, `recast` for printing)

Alternatives

`JSCodeshift`

The most popular tool to build JavaScript codemods

- Provides a jQuery-like API to navigate the AST
- Includes different tools (`@babel/parser` for parsing, `recast` for printing)
- It can be used with any AST transform tool you like (Babel included!)

Alternatives

`JSCodeshift`

The most popular tool to build JavaScript codemods

So… why Babel?

Alternatives

`JSCodeshift`

The most popular tool to build JavaScript codemods

So… why Babel?

- It provides a massive set of built-in utilities to analyze the AST and the Scope

Alternatives

`JSCodeshift`

The most popular tool to build JavaScript codemods

So... why Babel?

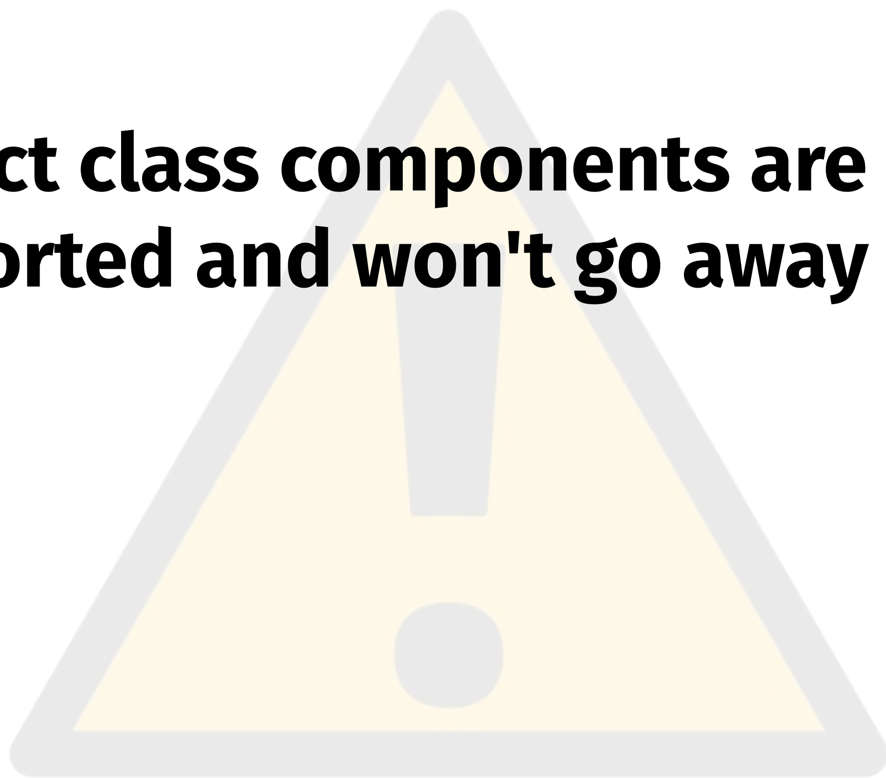
- It provides a massive set of built-in utilities to analyze the AST and the Scope
- You can re-use the same knowledge to modify your build process with custom plugins

A practical example

A practical example

Transforming React class components to functions with hooks

React class components are still supported and won't go away soon.



React class components are still supported and won't go away soon.

This codemod is for illustrative purposes, you don't actually need to do it!

```
class Counter extends React.Component {
  state = { name: "Unknown", count: 0 };

  incrementCount = () => {
    this.setState((state) => ({ count: state.count - 1 }));
  };

  decrementCount = () => {
    this.setState((state) => ({ count: state.count - 1 }));
  };

  updateName = (e) => {
    this.setState({ name: e.target.value });
  };

  render() {
    return (
      <div>
        <label>
          ${this.props.label}:
          <input type="text" defaultValue={this.state.name}
                 onChange={this.updateName} />
        </label>

        <button onClick={this.decrementCount}>-1</button>
        <span>{this.state.count}</span>
        <button onClick={this.incrementCount}>+1</button>
      </div>
    );
  }
}
```

```
const Counter = (props) => {
  const [name, setName] = useState("Unknown");
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount((count) => count - 1);
  };

  const decrementCount = () => {
    setCount((count) => count - 1);
  };

  const updateName = (e) => {
    setName(e.target.value);
  };

  return (
    <div>
      <label>
        ${props.label}:
        <input type="text" defaultValue={name}
               onChange={updateName} />
      </label>

      <button onClick={decrementCount}>-1</button>
      <span>{count}</span>
      <button onClick={incrementCount}>+1</button>
    </div>
  );
};
```

AST Explorer

A plugin development playground

AST Explorer: a plugin playground

AST Explorer | Snippet | JavaScript | </> @babel/parser | Transform | default | ?

Parser: @babel/parser-7.11.5
Transformer: babelv7-7.12.3

```
1 class MyComponent {
2 }
```

Tree | JSON | 14ms

☑ Autofocus ☑ Hide methods ☐ Hide empty keys ☑ Hide location data ☑ Hide type keys

```
- program: Program {
    sourceType: "script"
    interpreter: null
  - body: [
    - ClassDeclaration {
      + id: Identifier {name}
        superClass: null
      - body: ClassBody {
          body: [ ]
```

```
1 export default function codemod({ types: t, template }) {
2
3   // ...
4
5   return {
6     visitor: {
7
8       // ...
9
10    }
11  };
12 }
```

Prettier

```
1 class MyComponent {}
```

Built with React, Babel, Font Awesome, CodeMirror, Express, and webpack | GitHub | Build: 677cc64

@NicoloRibaudo

AST Explorer: a plugin playground

AST Explorer | Snippet | JavaScript | </> @babel/parser | Transform | default | ?

Parser: @babel/parser-7.11.5
Transformer: babelv7-7.12.3

Input code

```
1 class MyComponent {
2 }
```

Input AST

Tree | JSON | 14ms

☑ Autofocus ☑ Hide methods ☐ Hide empty keys ☑ Hide location data ☑ Hide type keys

```
- program: Program {
    sourceType: "script"
    interpreter: null
  - body: [
    - ClassDeclaration {
      + id: Identifier {name}
        superClass: null
      - body: ClassBody {
          body: [ ]
```

Babel plugin

Prettier

```
1 export default function codemod({ types: t, template }) {
2
3   // ...
4
5   return {
6     visitor: {
7
8       // ...
9
10    }
11  };
12 }
```

Output code

```
1 class MyComponent {}
```

Built with React, Babel, Font Awesome, CodeMirror, Express, and webpack | GitHub | Build: 677cc64

@NicoloRibaudo

Steps

Steps

1. Detecting React class components

@NicoloRibaudo

Steps

1. Detecting React class components
2. Transforming the `render()` method

Steps

1. Detecting React class components
2. Transforming the `render()` method
3. Adding support for `props`

@NicoloRibaudo

Steps

1. Detecting React class components
2. Transforming the `render()` method
3. Adding support for `props`
4. Adding support for `state`

@NicoloRibaudo

Steps

1. Detecting React class components
2. Transforming the `render()` method
3. Adding support for `props`
4. Adding support for `state`
    a. *Not* adding support for complex state (with `useReducer()`)

Complexity

@NicoloRibaudo

Steps

Complexity

1. Detecting React class components
2. Transforming the `render()` method
3. Adding support for `props`
4. Adding support for `state`
    a. *Not* adding support for complex state (with `useReducer()`)
    b. Analytics: Should we support `defaultProps`?

Steps

Complexity

1. Detecting React class components
2. Transforming the `render()` method
3. Adding support for `props`
4. Adding support for `state`
   a. *Not* adding support for complex state (with `useReducer()`)
   b. Analytics: Should we support `defaultProps`?
5. Adding support for class fields

Steps

Complexity

1. Detecting React class components
2. Transforming the `render()` method
3. Adding support for `props`
4. Adding support for `state`
    a. *Not* adding support for complex state (with `useReducer()`)
    b. Analytics: Should we support `defaultProps`?
5. Adding support for class fields
6. Injecting imports to hooks

@NicoloRibaudo

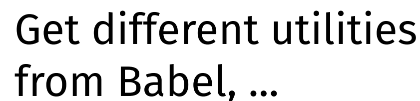
The anatomy of a Babel plugin

```
function codemod({ types: t, template }) {

  // ...

  return {
    visitor: {

      // ...

    }
  };
}
```

@NicoloRibaudo

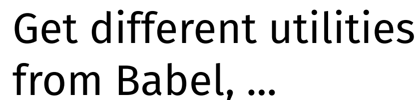
The anatomy of a Babel plugin

```
function codemod({ types: t, template }) {

  // ...

  return {
    visitor: {

      // ...

    }
  };
}
```



The anatomy of a Babel plugin

```
function codemod({ types: t, template }) {

  // ...

  return {
    visitor: {

      // ...

    }
  };
}
```



The anatomy of a Babel plugin

```
function codemod({ types: t, template }) {

  // ...

  return {
    visitor: {

      // ...

    }
  };
}
```

Get different utilities from Babel, ...

... define helper functions to analyze and transform ...

... and intercept the AST nodes to handle.

1. Detecting React class components

```
class MyComponent extends React.Component {
  // ...
}
```

1. Detecting React class components

```
class MyComponent extends React.Component {
  // ...
}
```

```
- ClassDeclaration {
    + id: Identifier {name}
    - superClass: MemberExpression {
        - object: Identifier {
            name: "React"
          }
          computed: false
        - property: Identifier = $node {
            name: "Component"
          }
      }
    - body: ClassBody {
        body: [ ]
      }
  }
```

@NicoloRibaudo

1. Detecting React class components

```
- ClassDeclaration {
  + id: Identifier {name}
  - superClass: MemberExpression {
    - object: Identifier {
        name: "React"
      }
      computed: false
    - property: Identifier  = $node {
        name: "Component"
      }
    }
  - body: ClassBody {
      body: [ ]
    }
}
```

@NicoloRibaudo

1. Detecting React class components

```
return {
  visitor: {

    // ...

  }
};
```

```
- ClassDeclaration {
  + id: Identifier {name}
  - superClass: MemberExpression {
    - object: Identifier {
        name: "React"
      }
      computed: false
    - property: Identifier = $node {
        name: "Component"
      }
    }
  - body: ClassBody {
      body: [ ]
    }
}
```

@NicoloRibaudo

1. Detecting React class components

```
return {
  visitor: {
    ClassDeclaration(path) {
      if (!isReactComponent(path.node)) {
        return;
      }

      alert("We have a React component!");
    }
  }
};
```

```
- ClassDeclaration {
    + id: Identifier {name}
    - superClass: MemberExpression {
        - object: Identifier {
              name: "React"
          }
          computed: false
        - property: Identifier = $node {
              name: "Component"
          }
      }
    - body: ClassBody {
          body: [ ]
      }
  }
```

@NicoloRibaudo

1. Detecting React class components

```
function isReactComponent(node) {
  return t.matchesPattern(
    node.superClass,
    "React.Component"
  );
}
return {
  visitor: {
    ClassDeclaration(path) { /* ... */ }
  }
};
```

```
- ClassDeclaration {
  + id: Identifier {name}
  - superClass: MemberExpression {
    - object: Identifier {
        name: "React"
      }
      computed: false
    - property: Identifier = $node {
        name: "Component"
      }
    }
  - body: ClassBody {
      body: [ ]
    }
}
```

@NicoloRibaudo

2. Extracting the `render()` method

2. Extracting the render( ) method

```
class Counter extends Component {
  render() {
    return <p>It works!</p>;
  }
}
```

@NicoloRibaudo

2. Extracting the render( ) method

```
class Counter extends Component {
  render() {
    return <p>It works!</p>;
  }
}
```

```
- ClassDeclaration  {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody  {
    - body:  [
      - ClassMethod  {
          static: false
        - key: Identifier  = $node {
            name: "render"
          }
          computed: false
          kind: "method"
          generator: false
          async: false
          params: [ ]
        - body: BlockStatement  {
          - body:  [
            + ReturnStatement {argument}
            ]
            directives: [ ]
          }
```

@NicoloRibaudo

2. Extracting the render( ) method

```
function findMethod(node, name) {




}

findMethod(path.node, "render");
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassMethod {
          static: false
        - key: Identifier = $node {
            name: "render"
          }
          computed: false
          kind: "method"
          generator: false
          async: false
          params: [ ]
        - body: BlockStatement {
          - body: [
              + ReturnStatement {argument}
            ]
            directives: [ ]
          }
```

@NicoloRibaudo

2. Extracting the render( ) method

```
function findMethod(node, name) {
  const elem = node.body.body




}

findMethod(path.node, "render");
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassMethod {
          static: false
        - key: Identifier = $node {
            name: "render"
          }
          computed: false
          kind: "method"
          generator: false
          async: false
          params: [ ]
        - body: BlockStatement {
          - body: [
            + ReturnStatement {argument}
            ]
            directives: [ ]
          }
```

@NicoloRibaudo

2. Extracting the render( ) method

```
function findMethod(node, name) {
  const elem = node.body.body.find(el =>
    t.isClassMethod(el)



  );

}

findMethod(path.node, "render");
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassMethod {
          static: false
        - key: Identifier = $node {
            name: "render"
          }
          computed: false
          kind: "method"
          generator: false
          async: false
          params: [ ]
        - body: BlockStatement {
          - body: [
              + ReturnStatement {argument}
            ]
            directives: [ ]
          }
```

@NicoloRibaudo

2. Extracting the render( ) method

```
function findMethod(node, name) {
  const elem = node.body.body.find(el =>
    t.isClassMethod(el, {
      static: false, computed: false
    })

  );

}

findMethod(path.node, "render");
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassMethod {
          static: false
        - key: Identifier = $node {
            name: "render"
          }
          computed: false
          kind: "method"
          generator: false
          async: false
          params: [ ]
        - body: BlockStatement {
          - body: [
              + ReturnStatement {argument}
            ]
            directives: [ ]
          }
```

@NicoloRibaudo

2. Extracting the render( ) method

```
function findMethod(node, name) {
  const elem = node.body.body.find(el =>
    t.isClassMethod(el, {
      static: false, computed: false
    })
    && t.isIdentifier(el.key, { name })
  );

}

findMethod(path.node, "render");
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassMethod {
          static: false
        - key: Identifier = $node {
            name: "render"
          }
          computed: false
          kind: "method"
          generator: false
          async: false
          params: [ ]
        - body: BlockStatement {
          - body: [
              + ReturnStatement {argument}
            ]
            directives: [ ]
          }
```

@NicoloRibaudo

2. Extracting the render( ) method

```
function findMethod(node, name) {
  const elem = node.body.body.find(el =>
    t.isClassMethod(el, {
      static: false, computed: false
    })
    && t.isIdentifier(el.key, { name })
  );
  if (elem) return elem.body.body;
}

findMethod(path.node, "render");
```

```
- ClassDeclaration {
   + id: Identifier {name}
   + superClass: Identifier {name}
   - body: ClassBody {
      - body: [
         - ClassMethod {
              static: false
            - key: Identifier = $node {
                 name: "render"
              }
              computed: false
              kind: "method"
              generator: false
              async: false
              params: [ ]
            - body: BlockStatement {
               - body: [
                  + ReturnStatement {argument}
                 ]
                 directives: [ ]
              }
```

@NicoloRibaudo

2. Extracting the render( ) method

```
function findMethod(node, name) {
  const elem = node.body.body.find(el =>
    t.isClassMethod(el, {
      static: false, computed: false
    })
    && t.isIdentifier(el.key, { name })
  );
  if (elem) return elem.body.body;
}

findMethod(path.node, "render");
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassMethod {
          static: false
        - key: Identifier = $node {
            name: "render"
          }
          computed: false
          kind: "method"
          generator: false
          async: false
          params: [ ]
        - body: BlockStatement {
          - body: [
            + ReturnStatement {argument}
            ]
            directives: [ ]
          }
```

@NicoloRibaudo

2. Extracting the render( ) method

```
visitor: {
  ClassDeclaration(path) {
    if (!isReactComponent(path.node)) return;

    path.replaceWith(template.ast`
      const ${path.node.id} = (props) => {
        ${findMethod(path.node, "render")};
      };
    `);
  },
},
```

@NicoloRibaudo

2. Extracting the render( ) method

```
visitor: {
  ClassDeclaration(path) {
    if (!isReactComponent(path.node)) return;

    path.replaceWith(template.ast`
      const ${path.node.id} = (props) => {
        ${findMethod(path.node, "render")};
      };
    `);
  },
},
```

```
class Counter extends Component {
  render() {
    return <p>It works!</p>;
  }
}
```

2. Extracting the render( ) method

```
visitor: {
  ClassDeclaration(path) {
    if (!isReactComponent(path.node)) return;

    path.replaceWith(template.ast`
      const ${path.node.id} = (props) => {
        ${findMethod(path.node, "render")};
      };
    `);
  },
},
```

```
class Counter extends Component {
  render() {
    return <p>It works!</p>;
  }
}
```

```
const Counter = (props) => {
 return <p>It works!</p>;
};
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
class Counter extends Component {
 render() {
   return <p>Hi, {this.props.name}!</p>
 }
}
```

3. Rewriting `this.props` usage

```
class Counter extends Component {
  render() {
    return <p>Hi, {this.props.name}!</p>
  }
}
```

```
- expression: MemberExpression {
  - object: MemberExpression {
      object: ThisExpression { }
      computed: false
    - property: Identifier = $node {
        name: "props"
      }
    }
    computed: false
  - property: Identifier {
      name: "name"
    }
  }
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
class Counter extends Component {
  render() {
    return <p>Hi, {this.props.name}!</p>
  }
}
```

```
- expression: MemberExpression {
  - object: MemberExpression {
      object: ThisExpression { }
      computed: false
    - property: Identifier = $node {
        name: "props"
      }
    }
    computed: false
  - property: Identifier {
      name: "name"
    }
}
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
function rewritePropsUsage(path) {


}
```

```
- expression: MemberExpression  {
  - object: MemberExpression  {
        object: ThisExpression { }
        computed: false
      - property: Identifier  = $node {
          name: "props"
        }
    }
    computed: false
  - property: Identifier  {
        name: "name"
    }
}
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
function rewritePropsUsage(path) {
  path.traverse({
    MemberExpression(path) {




    },
  });
}
```

```
- expression: MemberExpression {
  - object: MemberExpression {
        object: ThisExpression { }
        computed: false
      - property: Identifier = $node {
          name: "props"
        }
    }
    computed: false
  - property: Identifier {
        name: "name"
    }
  }
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
function rewritePropsUsage(path) {
  path.traverse({
    MemberExpression(path) {
      const { node } = path;
      if (
        !node.computed



      ) {

      }
    },
  });
}
```

```
- expression: MemberExpression {
  - object: MemberExpression {
        object: ThisExpression { }
        computed: false
      - property: Identifier = $node {
          name: "props"
        }
    }
    computed: false
  - property: Identifier {
      name: "name"
    }
}
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
function rewritePropsUsage(path) {
  path.traverse({
    MemberExpression(path) {
      const { node } = path;
      if (
        !node.computed &&
        t.isThisExpression(node.object)

      ) {

      }
    },
  });
}
```

```
- expression: MemberExpression {
  - object: MemberExpression {
      object: ThisExpression { }
      computed: false
    - property: Identifier = $node {
        name: "props"
      }
    }
    computed: false
  - property: Identifier {
      name: "name"
    }
}
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
function rewritePropsUsage(path) {
  path.traverse({
    MemberExpression(path) {
      const { node } = path;
      if (
        !node.computed &&
        t.isThisExpression(node.object) &&
        t.isIdentifier(node.property, { name: "props" })
      ) {

      }
    },
  });
}
```

```
- expression: MemberExpression {
  - object: MemberExpression {
        object: ThisExpression { }
        computed: false
      - property: Identifier = $node {
          name: "props"
        }
    }
    computed: false
  - property: Identifier {
        name: "name"
    }
  }
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
function rewritePropsUsage(path) {
  path.traverse({
    MemberExpression(path) {
      const { node } = path;
      if (
        !node.computed &&
        t.isThisExpression(node.object) &&
        t.isIdentifier(node.property, { name: "props" })
      ) {
        path.replaceWith(t.identifier("props"));
      }
    },
  });
}
```

```
- expression: MemberExpression {
    - object: Identifier {
          name: "props"
      }
      computed: false
    - property: Identifier {
          name: "name"
      }
  }
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
visitor: {
  ClassDeclaration(path) {
    if (!isReactComponent(path.node)) return;




    path.replaceWith(template.ast`
      const ${path.node.id} = (props) => {
        ${findMethod(path.node, "render")};
      };
    `);
  },
},
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
visitor: {
  ClassDeclaration(path) {
    if (!isReactComponent(path.node)) return;

    rewritePropsUsage(path);

    path.replaceWith(template.ast`
      const ${path.node.id} = (props) => {
        ${findMethod(path.node, "render")};
      };
    `);
  },
},
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
visitor: {
  ClassDeclaration(path) {
    if (!isReactComponent(path.node)) return;

    rewritePropsUsage(path);

    path.replaceWith(template.ast`
      const ${path.node.id} = (props) => {
        ${findMethod(path.node, "render")};
      };
    `);
  },
},
```

```
class Counter extends Component {
  render() {
    return <p>
      Hi, {this.props.name}!
    </p>;
  }
}
```

@NicoloRibaudo

3. Rewriting `this.props` usage

```
visitor: {
  ClassDeclaration(path) {
    if (!isReactComponent(path.node)) return;

    rewritePropsUsage(path);

    path.replaceWith(template.ast`
      const ${path.node.id} = (props) => {
        ${findMethod(path.node, "render")};
      };
    `);
  },
},
```

```
class Counter extends Component {
  render() {
    return <p>
      Hi, {this.props.name}!
    </p>;
  }
}
```

```
const Counter = (props) => {
 return <p>Hi, {props.name}!</p>;
};
```

@NicoloRibaudo

4. Convert state = {...} to useState( )

```
class Counter extends Component {
  state = { count: 0 };

  render() {
    return <p>
      Total: {this.state.count}
    </p>;
  }
}
```

4. Convert state = {...} to useState( )

```
class Counter extends Component {
  state = { count: 0 };

  render() {
    return <p>
      Total: {this.state.count}
    </p>;
  }
}
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassProperty {
          static: false
        - key: Identifier {
            name: "state"
          }
          computed: false
        - value: ObjectExpression {
          - properties: [
            + ObjectProperty {method,
              shorthand, value}
            ]
          }
        }
      + ClassMethod {static, key, computed
    ]
```

@NicoloRibaudo

4. Convert state = {...} to useState( )

```
class Counter extends Component {
  state = { count: 0 };

  render() {
    return <p>
      Total: {this.state.count}
    </p>;
  }
}
```

```
- ClassDeclaration {
    + id: Identifier {name}
    + superClass: Identifier {name}
- value: ObjectExpression {
    - properties: [
        - ObjectProperty {
              method: false
            - key: Identifier = $node {
                  name: "count"
              }
              computed: false
              shorthand: false
            + value: NumericLiteral {extra, value}
          }
      ]
  }
          + ClassMethod {static, key, computed
        ]
```

@NicoloRibaudo

4. Convert state = {...} to useState( )

```
function findField(node, name) {
  const elem = node.body.body.find(el =>
    t.isClassProperty(el, {
      static: false, computed: false
    })
    && t.isIdentifier(el.key, { name })
  );
  if (elem) return elem.value;
}

findField(path.node, "state");
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassProperty {
          static: false
        - key: Identifier {
            name: "state"
          }
          computed: false
        - value: ObjectExpression {
          - properties: [
            + ObjectProperty {method, shorthand, value}
            ]
          }
        }
      + ClassMethod {static, key, computed
    ]
```

@NicoloRibaudo

4. Convert state = {...} to useState( )

```
class Counter extends Component {
  state = { count: 0 };

  render() {
    return <p>
      Total: {this.state.count}
    </p>;
  }
}
```

```
count ──▶ get  : Identifier { "count" }
          set  : Identifier { "setCount" }
          init : NumericLiteral { 0 }

...
```

4. Convert `state = {…}` to `useState()`

```
- ClassDeclaration {
    + id: Identifier {name}
    + superClass: Identifier {name}
```

```
- value: ObjectExpression {
    - properties: [
        - ObjectProperty {
              method: false
            - key: Identifier = $node {
                  name: "count"
              }
              computed: false
              shorthand: false
            + value: NumericLiteral {extra, value}
          }
      ]
  }
```

```
            + ClassMethod {static, key, computed
          ]
```

@NicoloRibaudo

4. Convert state = {...} to useState( )

```
function findInitialState(node) {
  const stateNode = findField(node, "state");
  if (!stateNode) return;







}
```

```
- ClassDeclaration  {
   + id: Identifier {name}
   + superClass: Identifier {name}
- value: ObjectExpression  {
   - properties:  [
      - ObjectProperty  {
           method: false
         - key: Identifier  = $node {
              name: "count"
           }
           computed: false
           shorthand: false
         + value: NumericLiteral {extra, value}
        }
     ]
  }
           + ClassMethod {static, key, computed
           ]
```

@NicoloRibaudo

4. Convert state = {...} to useState( )

```
function findInitialState(node) {
  const stateNode = findField(node, "state");
  if (!stateNode) return;

  const state = new Map();
  for (const prop of stateNode.properties) {




  }
  return state;
}
```

```
- ClassDeclaration  {
   + id: Identifier {name}
   + superClass: Identifier {name}
- value: ObjectExpression  {
   - properties:  [
      - ObjectProperty  {
           method: false
         - key: Identifier  = $node {
              name: "count"
           }
           computed: false
           shorthand: false
         + value: NumericLiteral {extra, value}
        }
     ]
  }
         + ClassMethod {static, key, computed
         ]
```

@NicoloRibaudo

4. Convert state = {...} to useState( )

```
function findInitialState(node) {
  const stateNode = findField(node, "state");
  if (!stateNode) return;

  const state = new Map();
  for (const prop of stateNode.properties) {
    state.set(prop.key.name, {
      get: t.identifier(prop.key.name),
      set: t.identifier(`set${upper(prop.key.name)}`),
      init: prop.value,
    });
  }
  return state;
}
```

```
- ClassDeclaration {
    + id: Identifier {name}
    + superClass: Identifier {name}
- value: ObjectExpression {
    - properties: [
        - ObjectProperty {
            method: false
          - key: Identifier = $node {
                name: "count"
            }
            computed: false
            shorthand: false
          + value: NumericLiteral {extra, value}
        }
    ]
}
        + ClassMethod {static, key, computed
    ]
```

@NicoloRibaudo

4. Convert state = {…} to useState( )

```
get  : Identifier { … }
set  : Identifier { … }
init : NumericLiteral { … }

...
```

```
path.replaceWith(template.ast`
  const ${componentId} = (props) => {

    ${findMethod(path, "render")};
  };
`);
```

@NicoloRibaudo

4. Convert state = {…} to useState( )

```
const state = findInitialState(path.node);
```

```
get  : Identifier { … }
set  : Identifier { … }
init : NumericLiteral { … }

...
```

```
path.replaceWith(template.ast`
  const ${componentId} = (props) => {

    ${findMethod(path, "render")};
  };
`);
```

@NicoloRibaudo

4. Convert state = {…} to useState( )

```
const state = findInitialState(path.node);
const useStateCalls = [];
for (let { get, set, init } of state.values())
  useStateCalls.push(template.ast`
    const [${get}, ${set}] = useState(${init})
  `);

path.replaceWith(template.ast`
  const ${componentId} = (props) => {

    ${findMethod(path, "render")};
  };
`);
```

```
get  : Identifier { … }
set  : Identifier { … }
init : NumericLiteral { … }

...
```

@NicoloRibaudo

4. Convert state = {…} to useState( )

```
const state = findInitialState(path.node);
const useStateCalls = [];
for (let { get, set, init } of state.values())
  useStateCalls.push(template.ast`
    const [${get}, ${set}] = useState(${init})
  `);

path.replaceWith(template.ast`
  const ${componentId} = (props) => {
    ${useStateCalls};
    ${findMethod(path, "render")};
  };
`);
```

```
[get  : Identifier { … }
 set  : Identifier { … }
 init : NumericLiteral { … }]

 ...
```

@NicoloRibaudo

4. Convert state = {…} to useState( )

```
const state = findInitialState(path.node);
const useStateCalls = [];
for (let { get, set, init } of state.values())
  useStateCalls.push(template.ast`
    const [${get}, ${set}] = useState(${init})
  `);

path.replaceWith(template.ast`
  const ${componentId} = (props) => {
    ${useStateCalls};
    ${findMethod(path, "render")};
  };
`);
```

```
class Counter extends Component {
  state = { count: 0 };
  render() {
    return /* ... */;
  }
}
```

@NicoloRibaudo

4. Convert state = {...} to useState( )

```
const state = findInitialState(path.node);
const useStateCalls = [];
for (let { get, set, init } of state.values())
  useStateCalls.push(template.ast`
    const [${get}, ${set}] = useState(${init})
  `);

path.replaceWith(template.ast`
  const ${componentId} = (props) => {
    ${useStateCalls};
    ${findMethod(path, "render")};
  };
`);
```

```
class Counter extends Component {
  state = { count: 0 };
  render() {
    return /* ... */;
  }
}
```

```
const Counter = (props) => {
 const [count, setCount] = useState(0);
 return /* ... */;
};
```

@NicoloRibaudo

4. Convert `this.state.count` to `count`

```
class Counter extends Component {
  state = { count: 0 };
  render() {
    return <p>Total: {this.state.count}</p>;
  }
}
```

```
const Counter = (props) => {
  const [count, setCount] = useState(0);
  return <p>Total: {count}</p>;
};
```

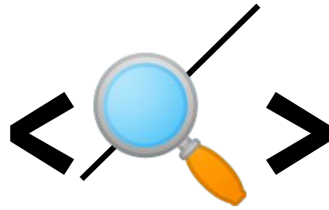
@NicoloRibaudo

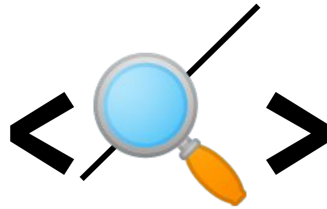
4. Convert `this.setState` to `setCount`

```
render() {
  const inc = () => this.setState(prev => ({
    count: prev.count + 1
  }));
  return <p onClick={inc}>Total: {this.state.count}</p>;
}
```

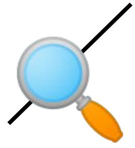
```
const Counter = (props) => {
  const [count, setCount] = useState(0);
  const inc = () => setCount(count => count + 1);
  return <p onClick={inc}>Total: {count}</p>;
};
```

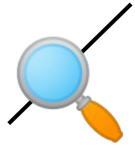
@NicoloRibaudo



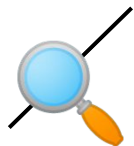


Does our codemod cover every case?

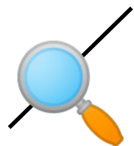
 Ignoring unsupported patterns

 Ignoring unsupported patterns

Complex state initialization

 Complex state initialization

```
class Counter extends Component {
  state = { count: 0, label: "Likes" };
  render() {
    return <p>
      {this.state.label}: {this.state.count}
    </p>;
  }
}
```

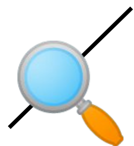


Complex state initialization

```
class Counter extends Component {
  state = { count: 0, label: "Likes" };
  render() {
    return <p>
      {this.state.label}: {this.state.count}
    </p>;
  }
}
```

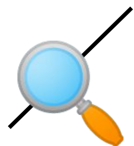
```
const Counter = (props) => {
   const [count, setCount] = useState(0);
   const [label, setLabel] = useState("Likes");

   return <p>{label}: {count}</p>;
};
```

@NicoloRibaudo

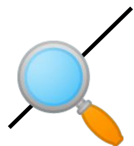
 Complex state initialization

```
class Counter extends Component {
  state = getInitialState();
  render() {
    return <p>
      {this.state.label}: {this.state.count}
    </p>;
  }
}
```

```
const Counter = (props) => {
   const [count, setCount] = ??????;
   const [label, setLabel] = ??????;

   return <p>{label}: {count}</p>;
};
```

 Complex state initialization

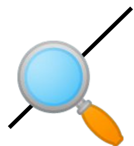
```
function findInitialState(node) {
  const stateNode = findField(node, "state");
  if (!stateNode) return;







  const state = new Map();
  for (const prop of stateNode.properties) {
    /* ... */
  }
  return state;
}
```

 Complex state initialization

```
function findInitialState(node) {
  const stateNode = findField(node, "state");
  if (!stateNode) return;






  const state = new Map();
  for (const prop of stateNode.properties) {
    /* ... */
  }
  return state;
}
```

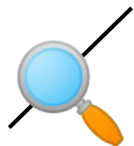
```
- value: ObjectExpression  {
  - properties:  [
    - ObjectProperty  {
        method: false
      - key: Identifier  = $node {
          name: "count"
        }
        computed: false
        shorthand: false
      + value: NumericLiteral {extra, value}
      }
    ]
  }
```

@NicoloRibaudo

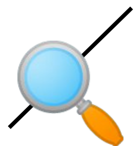
 Complex state initialization

```
function findInitialState(node) {
  const stateNode = findField(node, "state");
  if (!stateNode) return;

  if (!t.isObjectExpression(stateNode)) {



    return;
  }

  const state = new Map();
  for (const prop of stateNode.properties) {
    /* ... */
```

```
- value: ObjectExpression {
  - properties: [
    - ObjectProperty {
        method: false
      - key: Identifier = $node {
          name: "count"
        }
        computed: false
        shorthand: false
      + value: NumericLiteral {extra, value}
      }
    ]
  }
```

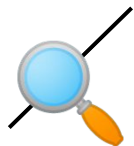
 Complex state initialization

```
function findInitialState(node) {
  const stateNode = findField(node, "state");
  if (!stateNode) return;

  if (!t.isObjectExpression(stateNode)) {
    t.addComment(
      stateNode.node, "leading", " @warning: Unable to refactor complex state initialization ",
    );
    return;
  }

  const state = new Map();
  for (const prop of stateNode.properties) {
    /* ... */
```



Complex state initialization

```
class Counter extends Component {
  state = getInisialState();
  render() {
    return <p>
      {this.state.label}: {this.state.count}
    </p>;
  }
}
```

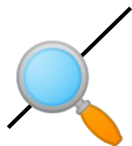
 Complex state initialization

```
class Counter extends Component {
  state
  rende
    ret
      {
    </p
  }
}
```

```
class Counter extends Component {
  state =
    /* @warning: Unable to refactor complex state initialization */
    getInisialState();

  render() {
    return <p>
      {this.state.label}: {this.state.count}
    </p>;
  }
}
```

@NicoloRibaudo

Analyzing a pattern's frequency

 Analyzing a pattern's frequency

Is it worth to implement support for `defaultProps`?

```
class Counter extends Component {
  static defaultProps = { name: "Nicolò" };

  render() {
    return <p>Hi, {this.props.name}!</p>;
  }
}
```

Analyzing a pattern's frequency

```
class Person {
 constructor(name) {
   this.name = name;
 }
}
```

```
class Btn extends Component {
  render() {
    return <button />
  }
```

```
class Btn extends Component {
 static defaultProps = {
   name: "Test",
 };

 render() {
   return <button />
 }
}
```

Analyzing a pattern's frequency

```
let reactClasses = 0, defaultProps = 0;

function defaultPropsAnalysis({ types: t }) {
  return {
    visitor: {



        // ...



    },
  };
```

```
class Person {
 constructor(name) {
   this.name = name;
 }
}
```

```
class Btn extends Component {
  render() {
    return <button />
  }
```

```
class Btn extends Component {
 static defaultProps = {
   name: "Test",
 };

 render() {
   return <button />
 }
}
```

@NicoloRibaudo

Analyzing a pattern's frequency

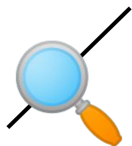
```
let reactClasses = 0, defaultProps = 0;

function defaultPropsAnalysis({ types: t }) {
  return {
    visitor: {
      ClassDeclaration(path) {


        // ...


      }
    },
  };
```

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

```
class Btn extends Component {
  render() {
    return <button />
  }
}
```

```
class Btn extends Component {
  static defaultProps = {
    name: "Test",
  };

  render() {
    return <button />
  }
}
```

@NicoloRibaudo

 Analyzing a pattern's frequency

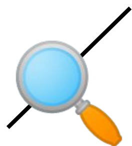
```js
let reactClasses = 2, defaultProps = 0;

function defaultPropsAnalysis({ types: t }) {
  return {
    visitor: {
      ClassDeclaration(path) {
        if (!isReactComponent(path.node)) return;
        reactClasses++;
        // ...

      }
    },
  };
```

```js
class Person {
 constructor(name) {
   this.name = name;
 }
}
```

```js
class Btn extends Component {
  render() {
    return <button />
  }
}
```

```js
class Btn extends Component {
 static defaultProps = {
   name: "Test",
 };

 render() {
   return <button />
 }
}
```

@NicoloRibaudo

 Analyzing a pattern's frequency

```js
let reactClasses = 2, defaultProps = 1;

function defaultPropsAnalysis({ types: t }) {
  return {
    visitor: {
      ClassDeclaration(path) {
        if (!isReactComponent(path.node)) return;
        reactClasses++;
        if (findField(path.node, "defaultProps"))
          defaultProps++;
      }
    },
  };
```

```js
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

```js
class Btn extends Component {
  render() {
    return <button />
  }
```

```js
class Btn extends Component {
  static defaultProps = {
    name: "Test",
  };

  render() {
    return <button />
  }
}
```

@NicoloRibaudo

Analyzing a pattern's frequency

```
glob("src/**/*.jsx").forEach(filename => {



});
```

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

```
class Btn extends Component {
  render() {
    return <button />
  }
```

```
class Btn extends Component {
  static defaultProps = {
    name: "Test",
  };

  render() {
    return <button />
  }
}
```

@NicoloRibaudo

Analyzing a pattern's frequency

```
glob("src/**/*.jsx").forEach(filename => {
  babel.transformFileSync(filename, {
    plugins: [defaultPropsAnalysis]
  });
});
```

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

```
class Btn extends Component {
  render() {
    return <button />
  }
}
```

```
class Btn extends Component {
  static defaultProps = {
    name: "Test",
  };

  render() {
    return <button />
  }
}
```

@NicoloRibaudo

Analyzing a pattern's frequency

```
glob("src/**/*.jsx").forEach(filename => {
  babel.transformFileSync(filename, {
    plugins: [defaultPropsAnalysis]
  });
});

console.log(`
Number of React classes: ${reactClasses}
Number ot classes using defaultProps: ${defaultProps}
(${(defaultProps / reactClasses)})
`);
```

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

```
class Btn extends Component {
  render() {
    return <button />
  }
```

```
class Btn extends Component {
  static defaultProps = {
    name: "Test",
  };

  render() {
    return <button />
  }
}
```

Analyzing a pattern's frequency

```
glob("src/**/*.jsx").forEach(filename => {
   babel.transformFileSync(filename, {
       plugins: [defaultPropsAnalysis]
   });
});

console.log(`
```

```
Number of React classes: 150
Number ot classes using defaultProps: 5
(0.03333333333333333)
```

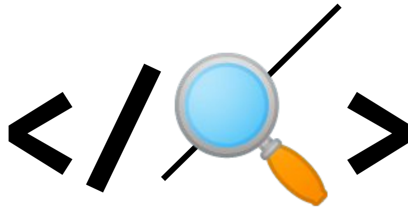
```
);
```

```
class Person {
 constructor(name) {
   this.name = name;
 }
}
```

```
class Btn extends Component {
  render() {
    return <button />
  }
```

```
class Btn extends Component {
 static defaultProps = {
   name: "Test",
 };

 render() {
   return <button />
 }
}
```

@NicoloRibaudo



Back to our codemod ...

Back to our codemod …

… assuming that we run some analysis and discovered that we almost only use arrow functions instead of class methods

5. Convert class fields to variables

```
class Counter extends Component {
  state = { count: 0 };

  inc = () =>
    this.setState(state => ({
      count: state.count + this.props.step,
    }));

  render() {
    return <p onClick={this.inc}>
      Total: {this.state.count}
    </p>;
  }
}
```

@NicoloRibaudo

5. Convert class fields to variables

```
class Counter extends Component {
  state = { count: 0 };

  inc = () =>
    this.setState(state => ({
      count: state.count + this.props.step,
    }));

  render() {
    return <p onClick={this.inc}>
      Total: {this.state.count}
    </p>;
  }
}
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassProperty {
          static: false
        - key: Identifier = $node {
            name: "inc"
          }
          computed: false
        + value: ArrowFunctionExpression
          {generator, async, params, body}
        }
      + ClassMethod {static, key, computed, kind,
        generator, ... +3}
      ]
    }
}
```

@NicoloRibaudo

5. Convert class fields to variables

```
class Counter extends Component {
  state = { count: 0 };

  inc = () =>
    this.setState(state => ({
      count: state.count + this.props.step,
    }));

  render() {
    return <p onClick={this.inc}>
      Total: {this.state.count}
    </p>;
  }
}
```

```
state ──► init : ObjectLiteral { ... }

inc   ──► init : ArrowFunctionExpression {}

...
```

@NicoloRibaudo

5. Convert class fields to variables

```
class Counter extends Component {
  state = { count: 0 };

  inc = () =>
    this.setState(state => ({
      count: state.count + this.props.step,
    }));

  render() {
    return <p onClick={this.inc}>
      Total: {this.state.count}
    </p>;
  }
}
```

```
inc    ——▶ init : ArrowFunctionExpression {}
...
```

@NicoloRibaudo

5. Convert class fields to variables

```
function findClassProperties(node) {
  const properties = new Map();









  return properties;
}
```

```
- ClassDeclaration {
    + id: Identifier {name}
    + superClass: Identifier {name}
    - body: ClassBody {
        - body: [
            - ClassProperty {
                  static: false
                - key: Identifier = $node {
                      name: "inc"
                  }
                  computed: false
                + value: ArrowFunctionExpression
                  {generator, async, params, body}
              }
            + ClassMethod {static, key, computed, kind,
              generator, ... +3}
          ]
      }
  }
```

@NicoloRibaudo

5. Convert class fields to variables

```
function findClassProperties(node) {
  const properties = new Map();
  for (const elem of node.body.body) {









  }
  return properties;
}
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassProperty {
          static: false
        - key: Identifier = $node {
            name: "inc"
          }
          computed: false
        + value: ArrowFunctionExpression
          {generator, async, params, body}
        }
      + ClassMethod {static, key, computed, kind,
        generator, ... +3}
      ]
    }
}
```

@NicoloRibaudo

5. Convert class fields to variables

```
function findClassProperties(node) {
  const properties = new Map();
  for (const elem of node.body.body) {
    if (
      !t.isClassProperty(elem, {
        computed: false, static: false,
      })
    ) continue;




  }
  return properties;
}
```

```
- ClassDeclaration  {
   + id: Identifier {name}
   + superClass: Identifier {name}
   - body: ClassBody  {
      - body:  [
         - ClassProperty  {
              static: false
            - key: Identifier  = $node {
                 name: "inc"
              }
              computed: false
            + value: ArrowFunctionExpression
              {generator, async, params, body}
           }
         + ClassMethod {static, key, computed, kind,
           generator, ... +3}
        ]
     }
  }
```

@NicoloRibaudo

5. Convert class fields to variables

```
function findClassProperties(node) {
  const properties = new Map();
  for (const elem of node.body.body) {
    if (
      !t.isClassProperty(elem, {
        computed: false, static: false,
      })
    ) continue;

    if (elem.key.name === "state") continue;


  }
  return properties;
}
```

```
- ClassDeclaration {
    + id: Identifier {name}
    + superClass: Identifier {name}
    - body: ClassBody {
        - body: [
            - ClassProperty {
                static: false
              - key: Identifier = $node {
                  name: "inc"
                }
                computed: false
              + value: ArrowFunctionExpression
                {generator, async, params, body}
              }
            + ClassMethod {static, key, computed, kind,
              generator, ... +3}
          ]
      }
  }
```

@NicoloRibaudo

5. Convert class fields to variables

```
function findClassProperties(node) {
  const properties = new Map();
  for (const elem of node.body.body) {
    if (
      !t.isClassProperty(elem, {
        computed: false, static: false,
      })
    ) continue;

    if (elem.key.name === "state") continue;

    properties.set(elem.key.name, elem.value);
  }
  return properties;
}
```

```
- ClassDeclaration {
  + id: Identifier {name}
  + superClass: Identifier {name}
  - body: ClassBody {
    - body: [
      - ClassProperty {
          static: false
        - key: Identifier = $node {
            name: "inc"
          }
          computed: false
        + value: ArrowFunctionExpression
          {generator, async, params, body}
        }
      + ClassMethod {static, key, computed, kind,
        generator, ... +3}
      ]
    }
}
```

@NicoloRibaudo

5. Convert class fields to variables

```
visotor: {
  ClassDeclaration(path) {

    // ...

    path.replaceWith(template.ast`
      const ${componentId} = (props) => {
        ${useStateCalls};

        ${findMethod(path, "render")};
      };
    `);
  }
}
```

@NicoloRibaudo

5. Convert class fields to variables

```
visotor: {
  ClassDeclaration(path) {

    // ...

    path.replaceWith(template.ast`
      const ${componentId} = (props) => {
        ${useStateCalls};

        ${findMethod(path, "render")};
      };
    `);
  }
}
```

```
inc ——▶ init : ArrowFunctionExpression {}

...
```

@NicoloRibaudo

5. Convert class fields to variables

```
inc ——→ init : ArrowFunctionExpression {}

...
```

```
// ...

path.replaceWith(template.ast`
  const ${componentId} = (props) => {
    ${useStateCalls};

    ${findMethod(path, "render")};
  };
`);
```

@NicoloRibaudo

5. Convert class fields to variables

```
const vars = findClassProperties(path.node);

// ...



path.replaceWith(template.ast`
  const ${componentId} = (props) => {
    ${useStateCalls};

    ${findMethod(path, "render")};
  };
`);
```

```
inc  ──▶  init : ArrowFunctionExpression {}

...
```

@NicoloRibaudo

5. Convert class fields to variables

```
const vars = findClassProperties(path.node);
const varsDeclarations = [];
for (const [name, init] of vars)
  varsDeclarations.push(template.ast`
    const ${t.identifier(name)} = ${init};
  `);

path.replaceWith(template.ast`
  const ${componentId} = (props) => {
    ${useStateCalls};

    ${findMethod(path, "render")};
  };
`);
```

```
inc  ⟶  init : ArrowFunctionExpression {}

...
```

5. Convert class fields to variables

```
const vars = findClassProperties(path.node);
const varsDeclarations = [];
for (const [name, init] of vars)
  varsDeclarations.push(template.ast`
    const ${t.identifier(name)} = ${init};
  `);

path.replaceWith(template.ast`
  const ${componentId} = (props) => {
    ${useStateCalls};
    ${varsDeclarations};
    ${findMethod(path, "render")};
  };
`);
```

```
inc  ──▶  init : ArrowFunctionExpression {}
...
```

5. Convert class fields to variables

```
class Counter extends Component {
  state = { count: 0 };

  inc = () => this.setState(state => ({
    count: state.count + this.props.step,
  }));

  render() {
    return <p onClick={this.inc}>
      Total: {this.state.count}
    </p>;
  }
}
```

@NicoloRibaudo

5. Convert class fields to variables

```
class Counter extends Component {
  state = { count: 0 };

  inc = () => this.setState(state => ({
    count: state.count + this.props.step,
  }));

  render() {
    return <p onClick={this.inc}>
      Total: {this.state.count}
    </p>;
  }
}
```

```
const Counter = props => {
  const [count, setCount] = useState(0);

  const inc = () => setCount(
    count => count + props.step
  );

  return <p onClick={this.inc}>
    Total: {count}
  </p>;
};
```

@NicoloRibaudo

5. Convert class fields to variables

```
class Counter extends Component {
  state = { count: 0 };

  inc = () => this.setState(state => ({
    count: state.count + this.props.step,
  }));

  render() {
    return <p onClick={this.inc}>
      Total: {this.state.count}
    </p>;
  }
}
```

```
const Counter = props => {
  const [count, setCount] = useState(0);

  const inc = () => setCount(
    count => count + props.step
  );

  return <p onClick={this.inc}>
    Total: {count}
  </p>;
};
```

@NicoloRibaudo

5. Convert class fields to variables

```
class Counter extends Component {
  state = { count: 0 };

  inc = () => this.setState(state => ({
    count: state.count + this.props.step,
  }));

  render() {
    return <p onClick={this.inc}>
      Total: {this.state.count}
    </p>;
  }
}
```

```
- expression: MemberExpression {
    start: 191
    end: 199
  + loc: {start, end}
  - object: ThisExpression {
      start: 191
      end: 195
    + loc: {start, end}
    }
    computed: false
  - property: Identifier = $node {
      start: 196
      end: 199
    + loc: {start, end, identifierName}
      name: "inc"
    }
  }
```

@NicoloRibaudo

5. Convert class fields *usage* to variables

```
function rewriteVarsUsage(path, props) {


}
```

```
- expression: MemberExpression  {
    start: 191
    end: 199
  + loc: {start, end}
  - object: ThisExpression  {
      start: 191
      end: 195
    + loc: {start, end}
    }
    computed: false
  - property: Identifier  = $node {
      start: 196
      end: 199
    + loc: {start, end, identifierName}
      name: "inc"
    }
  }
```

@NicoloRibaudo

5. Convert class fields *usage* to variables

```
function rewriteVarsUsage(path, props) {
  path.traverse({
    MemberExpression(path) {






    }
  });
}
```

```
- expression: MemberExpression {
    start: 191
    end: 199
  + loc: {start, end}
  - object: ThisExpression {
      start: 191
      end: 195
    + loc: {start, end}
    }
    computed: false
  - property: Identifier = $node {
      start: 196
      end: 199
    + loc: {start, end, identifierName}
      name: "inc"
    }
}
```

@NicoloRibaudo

5. Convert class fields *usage* to variables

```
function rewriteVarsUsage(path, props) {
  path.traverse({
    MemberExpression(path) {
      const { node } = path;
      if (!t.isThisExpression(node.object)) return;
      if (node.computed) return;





    }
  });
}
```

```
- expression: MemberExpression  {
    start: 191
    end: 199
  + loc: {start, end}
  - object: ThisExpression  {
      start: 191
      end: 195
    + loc: {start, end}
    }
    computed: false
  - property: Identifier  = $node {
      start: 196
      end: 199
    + loc: {start, end, identifierName}
      name: "inc"
    }
  }
```

@NicoloRibaudo

5. Convert class fields *usage* to variables

```
function rewriteVarsUsage(path, props) {
  path.traverse({
    MemberExpression(path) {
      const { node } = path;
      if (!t.isThisExpression(node.object)) return;
      if (node.computed) return;

      const { name } = node.property;
      if (!props.has(name)) return;


    }
  });
}
```

```
- expression: MemberExpression {
    start: 191
    end: 199
  + loc: {start, end}
  - object: ThisExpression {
      start: 191
      end: 195
    + loc: {start, end}
    }
    computed: false
  - property: Identifier = $node {
      start: 196
      end: 199
    + loc: {start, end, identifierName}
      name: "inc"
    }
  }
```

@NicoloRibaudo

5. Convert class fields *usage* to variables

```
function rewriteVarsUsage(path, props) {
  path.traverse({
    MemberExpression(path) {
      const { node } = path;
      if (!t.isThisExpression(node.object)) return;
      if (node.computed) return;

      const { name } = node.property;
      if (!props.has(name)) return;

      path.replaceWith(t.identifier(name));
    }
  });
}
```

```
- expression: MemberExpression {
    start: 191
    end: 199
  + loc: {start, end}
  - object: ThisExpression {
      start: 191
      end: 195
    + loc: {start, end}
    }
    computed: false
  - property: Identifier = $node {
      start: 196
      end: 199
    + loc: {start, end, identifierName}
      name: "inc"
    }
  }
```

@NicoloRibaudo

5. Convert class fields *usage* to variables

```
class Counter extends Component {
  state = { count: 0 };

  inc = () => this.setState(state => ({
    count: state.count + this.props.step,
  }));

  render() {
    return <p onClick={this.inc}>
      Total: {this.state.count}
    </p>;
  }
}
```

```
const Counter = props => {
  const [count, setCount] = useState(0);

  const inc = () => setCount(
    count => count + props.step
  );

  return <p onClick={inc}>
    Total: {count}
  </p>;
};
```

@NicoloRibaudo

6. Inject `imports` for used hooks

6. Inject `imports` for used hooks

So far we only implemented local transforms, modifying a self-contained AST node:

`ClassDeclaration`

6. Inject `imports` for used hooks

So far we only implemented local transforms, modifying a self-contained AST node:

`ClassDeclaration`

We can use Babel's scope analysis and AST utilities to modify unrelated parts of the AST.

6. Inject `imports` for used hooks

```
class Counter extends Component {
  state = { count: 0 };
  render() {
    return <p>Total: {this.state.count}</p>;
  }
}
```

```
const Counter = (props) => {
 const [count, setCount] = useState(0);
 return <p>Total: {count}</p>;
};
```

@NicoloRibaudo

6. Inject imports for used hooks

```
class Counter extends Component {
  state = { count: 0 };
  render() {
    return <p>Total: {this.state.count}</p>;
  }
}
```

ReferenceError: useState is not defined

```
const Counter = (props) => {
 const [count, setCount] = useState(0);
 return <p>Total: {count}</p>;
};
```

@NicoloRibaudo

6. Inject imports for used hooks

```
class Counter extends Component {
  state = { count: 0 };
  render() {
    return <p>Total: {this.state.count}</p>;
  }
}
```

```
const Counter = (props) => {
 const [count, setCount] = useState(0);
 return <p>Total: {count}</p>;
};
```

@NicoloRibaudo

6. Inject imports for used hooks

```
class Counter extends Component {
  state = { count: 0 };
  render() {
    return <p>Total: {this.state.count}</p>;
  }
}
```

```
import { useState } from "react";
const Counter = (props) => {
 const [count, setCount] = useState(0);
 return <p>Total: {count}</p>;
};
```

@NicoloRibaudo

6. Inject imports for used hooks

```
function findReactImport(path) {




}
```

```
import * as _ from "lodash";
import React, { Component } from "react";

export function buildBtn(color) {
  class Btn extends Component {
    state = { x: 0 };

    render() {
      return (
        <button color={color} />
      );
    }
  }
  return Btn;
}
```

6. Inject imports for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
}
```

6. Inject imports for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
  let importPath = program.get("body")
    .filter(({ node }) => t.isImportDeclaration(node))









}
```

```
import * as _ from "lodash";
import React, { Component } from "react";

export function buildBtn(color) {
  class Btn extends Component {
    state = { x: 0 };

    render() {
      return (
        <button color={color} />
      );
    }
  }
  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
  let importPath = program.get("body")
    .filter(({ node }) => t.isImportDeclaration(node))
    .find(({ node }) => node.source.value === "react");



}
```

```
import * as _ from "lodash";
import React, { Component } from "react";

export function buildBtn(color) {
  class Btn extends Component {
    state = { x: 0 };

    render() {
      return (
        <button color={color} />
      );
    }
  }
  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
  let importPath = program.get("body")
    .filter(({ node }) => t.isImportDeclaration(node))
    .find(({ node }) => node.source.value === "react");

  if (importPath) return importPath;




}
```

```
import * as _ from "lodash";
import React, { Component } from "react";

export function buildBtn(color) {
  class Btn extends Component {
    state = { x: 0 };

    render() {
      return (
        <button color={color} />
      );
    }
  }
  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
  let importPath = program.get("body")
    .filter(({ node }) => t.isImportDeclaration(node))
    .find(({ node }) => node.source.value === "react");

  if (importPath) return importPath;



}
```

```
import * as _ from "lodash";

export function buildBtn(color) {
  class Btn extends Component {
    state = { x: 0 };

    render() {
      return (
        <button color={color} />
      );
    }
  }
  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
  let importPath = program.get("body")
    .filter(({ node }) => t.isImportDeclaration(node))
    .find(({ node }) => node.source.value === "react");

  if (importPath) return importPath;

  program.unshiftContainer(
    "body", template.ast`import "react"`
  );

}
```

```
import "react";
import * as _ from "lodash";

export function buildBtn(color) {
  class Btn extends Component {
    state = { x: 0 };

    render() {
      return (
        <button color={color} />
      );
    }
  }
  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
  let importPath = program.get("body")
    .filter(({ node }) => t.isImportDeclaration(node))
    .find(({ node }) => node.source.value === "react");

  if (importPath) return importPath;

  program.unshiftContainer(
    "body", template.ast`import "react"`
  );
  return program.get("body.0");
}
```

```
import "react";
import * as _ from "lodash";

export function buildBtn(color) {
  class Btn extends Component {
    state = { x: 0 };

    render() {
      return (
        <button color={color} />
      );
    }
  }
  return Btn;
}
```

@NicoloRibaudo

6. Inject `imports` for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
  let importPath = program.get("body")
    .filter(({ node }) => t.isImportDeclaration(node))
    .find(({ node }) => node.source.value === "react");

  if (importPath) return importPath;

  program.unshiftContainer(
    "body", template.ast`import "react"`
  );
  return program.get("body.0");
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
  let importPath = program.get("body")
    .filter(({ node }) => t.isImportDeclaration(node))
    .find(({ node }) => node.source.value === "react");

  if (importPath) return importPath;

  program.unshiftContainer(
    "body", template.ast`import "react"`
  );
  return program.get("body.0");
}
```

We are always working with *paths*, not with *nodes*.

@NicoloRibaudo

6. Inject `imports` for used hooks

```
function findReactImport(path) {
  const program = path.findParent((p) => p.isProgram());
  let importPath = program.get("body")
    .filter(({ node }) => t.isImportDeclaration(node))
    .find(({ node }) => node.source.value === "react");

  if (importPath) return importPath;

  program.unshiftContainer(
    "body", template.ast`import "react"`
  );
  return program.get("body.0");
}
```

We are always working with *paths*, not with *nodes*.

AST nodes are ergonomic for reading the AST, but when mutating it we must use *NodePaths* to let Babel track updates.

@NicoloRibaudo

6. Inject `imports` for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */



  }
}
```

```
import "react";
import * as _ from "lodash";

export function buildBtn(color) {
  class Btn extends Component {
    state = { x: 0 };

    render() {
      return (
        <button color={color} />
      );
    }
  }
  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */



  }
}
```

```
import "react";
import * as _ from "lodash";

export function buildBtn(color) {
  const Btn = props => {
    const [x, setX] = useState(0);

    return (
      <button color={color} />
    );
  }


  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */
    if (
       useStateCalls.length > 0

    ) {






    }
  }
}
```

```
import "react";
import * as _ from "lodash";

export function buildBtn(color) {
  const Btn = props => {
    const [x, setX] = useState(0);

    return (
      <button color={color} />
    );
  }


  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */
    if (
       useStateCalls.length > 0

    ) {
      findReactImport(path)




    }
  }
}
```

```
import "react";
import * as _ from "lodash";

export function buildBtn(color) {
  const Btn = props => {
    const [x, setX] = useState(0);

    return (
      <button color={color} />
    );
  }


  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */
    if (
       useStateCalls.length > 0

    ) {
      findReactImport(path)




    }
  }
}
```

```
import { useState } from "react";
```

```
- ImportDeclaration  {
    start: 6
    end: 39
  + loc: {start, end}
    importKind: "value"
  - specifiers:  [
     + ImportSpecifier {start, end, loc, imported, local}
    ]
  + source: StringLiteral {start, end, loc, extra, value}
  }
```

6. Inject imports for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */
    if (
      useStateCalls.length > 0

    ) {
      findReactImport(path)



    }
  }
}
```

```
import { useState } from "react";
```

```
- ImportD
    star
    end:
  + loc:
    impo
  - spec
    + I                                    , local}
    ]
  + sour                                   , value}
  }
```

```
- ImportSpecifier {
    start: 15
    end: 23
  + loc: {start, end}
  - imported: Identifier  = $node {
      start: 15
      end: 23
    + loc: {start, end, identifierName}
      name: "useState"
    }
  - local: Identifier {
      start: 15
      end: 23
    + loc: {start, end, identifierName}
      name: "useState"
    }
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */
    if (
      useStateCalls.length > 0

    ) {
      findReactImport(path)



    }
  }
}
```

```
import { useState as useState } from "react";
```

```
- ImportD
    star
    end:
  + loc:
    impo
  - spec
    + I
    ]
  + sour
}
```

```
- ImportSpecifier {
    start: 15
    end: 23
  + loc: {start, end}
  - imported: Identifier = $node {
      start: 15
      end: 23
    + loc: {start, end, identifierName}
      name: "useState"
    }
  - local: Identifier {
      start: 15
      end: 23
    + loc: {start, end, identifierName}
      name: "useState"
    }
}
```

, local}
, value}

@NicoloRibaudo

6. Inject imports for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */
    if (
       useStateCalls.length > 0

    ) {
      findReactImport(path)




    }
  }
}
```

```
import "react";
import * as _ from "lodash";

export function buildBtn(color) {
  const Btn = props => {
    const [x, setX] = useState(0);

    return (
      <button color={color} />
    );
  }


  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */
    if (
       useStateCalls.length > 0

    ) {
      findReactImport(path).pushContainer(
        "specifiers",
        t.importSpecifier(t.identifier("useState"),
                          t.identifier("useState"))
      );
    }
  }
}
```

```
import "react";
import * as _ from "lodash";

export function buildBtn(color) {
  const Btn = props => {
    const [x, setX] = useState(0);

    return (
      <button color={color} />
    );
  }



  return Btn;
}
```

@NicoloRibaudo

6. Inject imports for used hooks

```
visotor: {
  ClassDeclaration(path) {
    /* ... */
    if (
       useStateCalls.length > 0 &&
       !path.scope.hasBinding("useState")
    ) {
      findReactImport(path).pushContainer(
        "specifiers",
        t.importSpecifier(t.identifier("useState"),
                          t.identifier("useState"))
      );
    }
  }
}
```

```
import "react";
import * as _ from "lodash";

export function buildBtn(color) {
  const Btn = props => {
    const [x, setX] = useState(0);

    return (
      <button color={color} />
    );
  }


  return Btn;
}
```

@NicoloRibaudo

6. Inject `imports` for used hooks

```
class Counter extends Component {
  state = { count: 0 };
  render() {
    return <p>Total: {this.state.count}</p>;
  }
}
```

```
import { useState } from "react";
const Counter = (props) => {
 const [count, setCount] = useState(0);
 return <p>Total: {count}</p>;
};
```

@NicoloRibaudo

</Example>

Complementary tools

Complementary tools

Babel is great at doing two things:

Complementary tools

Babel is great at doing two things:

- Parsing JavaScript, proposals or language extensions

Complementary tools

Babel is great at doing two things:

- Parsing JavaScript, proposals or language extensions
- Providing tools to transform the code's AST

Complementary tools

Babel is great at doing two things:

- Parsing JavaScript, proposals or language extensions
- Providing tools to transform the code's AST

Codemods also need:

Complementary tools

Babel is great at doing two things:

- Parsing JavaScript, proposals or language extensions
- Providing tools to transform the code's AST

Codemods also need:

- A way to print the resulting AST, keeping the original formatting

Complementary tools

Babel is great at doing two things:

- Parsing JavaScript, proposals or language extensions
- Providing tools to transform the code's AST

Codemods also need:

- A way to print the resulting AST, keeping the original formatting
- A way to run the Babel plugin on the different files

Complementary tools

Codemods also need:

- A way to print the resulting AST, keeping the original formatting
- A way to run the Babel plugin on the different files

Complementary tools

Codemods also need:

- A way to print the resulting AST, keeping the original formatting
- A way to run the Babel plugin on the different files

@NicoloRibaudo

Complementary tools

recast

Codemods also need:

- A way to print the resulting AST, keeping the original formatting
- A way to run the Babel plugin on the different files

Complementary tools

recast

JSCodeshift's CLI

Codemods also need:

- A way to print the resulting AST, keeping the original formatting
- A way to run the Babel plugin on the different files

@NicoloRibaudo

Complementary tools

recast

JSCodeshift's CLI

A manual for loop that iterates over the files

Codemods also need:

- A way to print the resulting AST, keeping the original formatting
- A way to run the Babel plugin on the different files

Demo!

https://github.com/nicolo-ribaudo/conf-holyjs-moscow-2020

Babel:
A refactoring tool

NICOLÒ RIBAUDO

Babel team

Example steps

Original:
https://astexplorer.net/#/gist/43f16e179d959534ea7904c86b9cc26f/0921445ea2b1095c22f7be2ef5bbeed9236a57b8

1. Detect React class
2. Extract `render()` method
3. Rewrite `this.props` usage
4. Convert `state = { ... }` initialization to `useState`
5. Optimize `useState()` calls (observation: is it worth it?)
6. Rewrite `this.state.foo` reads
7. Transform class properties to local variables
8. Rewrite `this.setState({ ... })` to hooks
9. Rewrite `this.setState(callback)` to hooks (is it worth it?)
10. Add support for some lifecycle hooks

@NicoloRibaudo