Automating design token migrations with codemods

Steve Dodier-Lazaro

07/11/2023

Back Market

Design tokens

What they are

- Collection of design values like colors, spacings, font properties
- Each value is associated with a name
- Token names encode the relations between tokens

```
$bg-static-default-low: #fff;
$text-static-default-low: #555;
$text-static-default-hi: #111;
```

Design tokens

Their purpose

- They unify language between designers and engineers
- They help apply a brand identity consistently across products

Design tokens are presented as a tool that lets you make future brand changes without effort.

Design tokens

Token systems are not immutable

- Token systems have implicit usage rules and can't be compatible with every brand (e.g. Material Design)
- When rebranding, the system often needs to be adjusted
- Token systems also change when we learn from our mistakes (indexed spacings, overly simplistic color systems)

Today's topic

When tokens change, we often need to migrate component code.

Today's topic

When tokens change, we often need to migrate component code.

Manually.

Today's topic

When tokens change, we often need to migrate component code.

Manually?

Brand migrations with tokens

- Tokens force us to normalise how we declare style
- This results in uniform CSS code
- Migrating tokenised code is mostly about renaming variables

Example with Tailwind

```
typography-2
bg-primary
text-primary
hover:bg-primary-hover

```

```
body-1
bg-action-primary
text-onaction-primary
hover:bg-action-primary-hover
focus:bg-action-primary-focus
```

Brand migrations with tokens

- Tokens force us to normalise how we declare style
- This results in uniform CSS code
- Migrating tokenised code is mostly about renaming variables

Example with Tailwind

```
typography-2
bg-primary
text-primary
hover:bg-primary-hover

```

```
body-1
bg-action-primary
text-onaction-primary
hover:bg-action-primary-hover
focus:bg-action-primary-focus
```

Brand migrations with tokens

- Tokens force us to normalise how we declare style
- This results in uniform CSS code
- Migrating tokenised code is mostly about renaming variables

Example with Tailwind

```
typography-2
bg-primary
text-primary
hover:bg-primary-hover
```

```
body-1
bg-action-primary
text-onaction-primary
hover:bg-action-primary-hover
focus:bg-action-primary-focus
```

Brand migrations with tokens

- Tokens force us to normalise how we declare style
- This results in uniform CSS code
- Migrating tokenised code is mostly about renaming variables

Example with Tailwind

```
typography-2
bg-primary
text-primary
hover:bg-primary-hover
```

```
body-1
bg-action-primary
text-onaction-primary
hover:bg-action-primary-hover
focus:bg-action-primary-focus
```

Building blocks for token migrations

Many situations with token changes are simple to deal with.

Token value change	Change the token value; no migration to do
Token addition	Find hardcoded values matching the new tokens, and replace them with tokens
Token removal	Replace references to the tokens with their values
Replacing tokens with other ones	Find where old tokens were used, and replace them with the right tokens in the new system

Things get muddier in practice

Conflicts

Actual token migrations combine these simple scenarios. There are name and value conflicts between old and new tokens, so the migration must be done in a specific order.

Things get muddier in practice

Conflicts

Actual token migrations combine these simple scenarios. There are name and value conflicts between old and new tokens, so the migration must be done in a specific order.

One-to-many changes

Sometimes, an token in the old system must be replaced by different tokens based on context (e.g. is the component using the token interactive?).

Enter codemodding

Codemod engines find and replace patterns in programs, based on their grammar.

We can use them to migrate tokens with an understanding of their surrounding context.

Codemodding 101

- Codemod engines operate on the grammar of source code files
- They build an **Abstract Syntax Tree** (AST) of the file
- They provide an API for you to search and edit the AST

- Codemods are script that use a codemodding engine's API
- For every migration, you write a new codemod script

Hello World AST

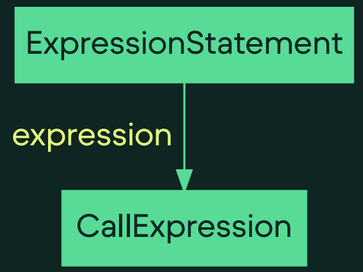
```
console.log('hello world');
```

Hello World AST

```
console.log('hello world');
```

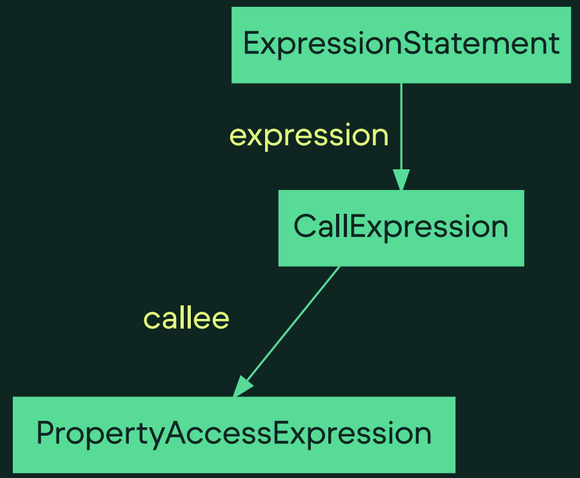
Hello World AST

```
console.log('hello world');
```



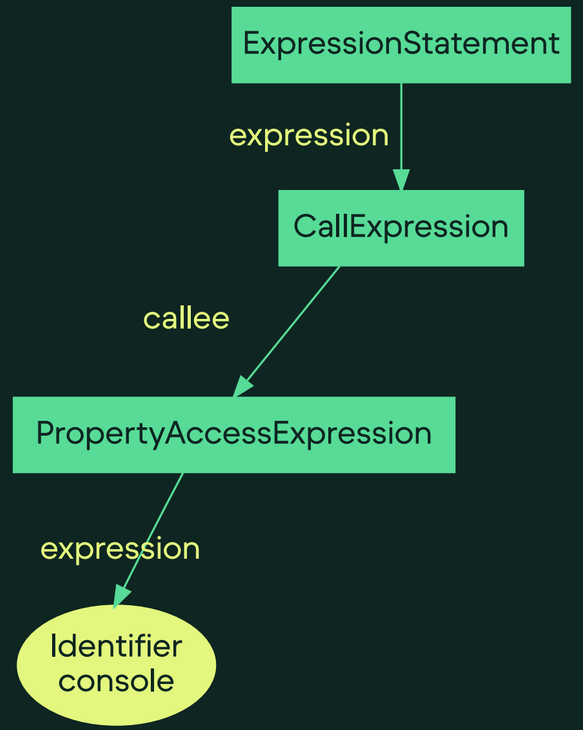
Hello World AST

```
console.log('hello world');
```



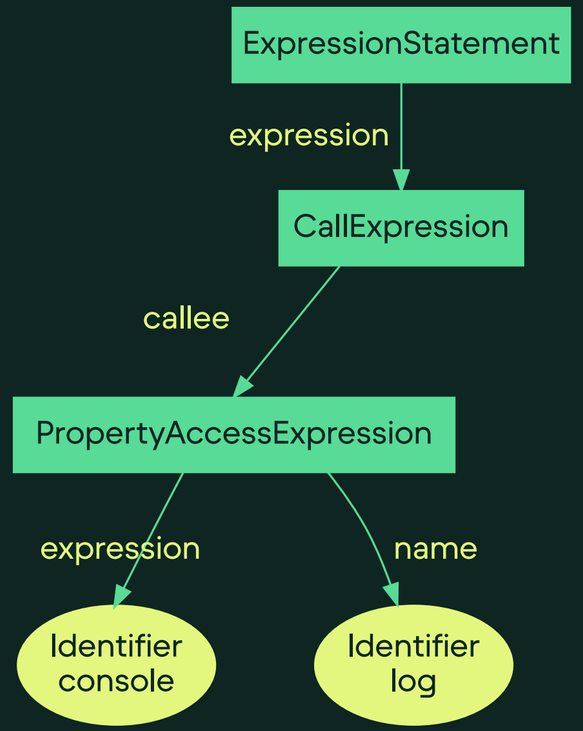
Hello World AST

```
console.log('hello world');
```



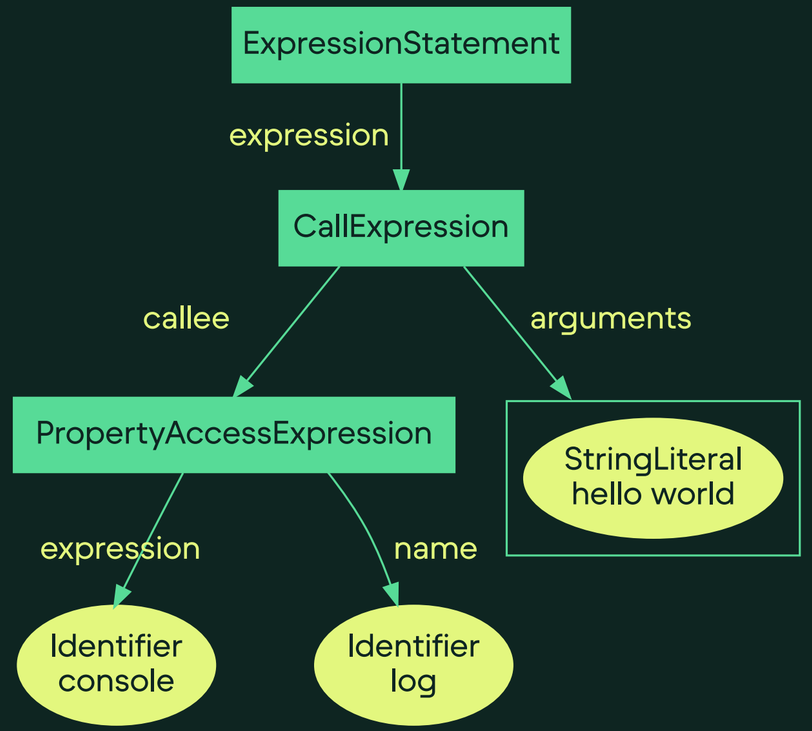
Hello World AST

```
console.log('hello world');
```



Hello World AST

```
console.log('hello world');
```



An example with size tokens

At Back Market, we used the following spacing tokens:

```
{
  "spacing": {
    "0": { "value": "0" },
    "px": { "value": "1px" },
    "1": { "value": "4px" },
    "2": { "value": "8px" },
    "3": { "value": "12px" },
    "4": { "value": "16px" },
    "5": { "value": "20px" },
    "6": { "value": "24px" },
    "7": { "value": "32px" },
    "8": { "value": "56px" },
    "9": { "value": "72px" }
  }
}
```

An example with size tokens

At Back Market, we used the following spacing tokens:

```
{
  "spacing": {
    "0": { "value": "0" },
    "px": { "value": "1px" },
    "1": { "value": "4px" },
    "2": { "value": "8px" },
    "3": { "value": "12px" },
    "4": { "value": "16px" },
    "5": { "value": "20px" },
    "6": { "value": "24px" },
    "7": { "value": "32px" },
    "8": { "value": "56px" },
    "9": { "value": "72px" }
  }
}
```

An example with size tokens

At Back Market, we used the following spacing tokens:

```
{
  "spacing": {
    "0": { "value": "0" },
    "px": { "value": "1px" },
    "1": { "value": "4px" },
    "2": { "value": "8px" },
    "3": { "value": "12px" },
    "4": { "value": "16px" },
    "5": { "value": "20px" },
    "6": { "value": "24px" },
    "7": { "value": "32px" },
    "8": { "value": "56px" },
    "9": { "value": "72px" }
  }
}
```

An example with size tokens

At Back Market, we used the following spacing tokens:

```
{
  "spacing": {
    "0": { "value": "0" },
    "px": { "value": "1px" },
    "1": { "value": "4px" },
    "2": { "value": "8px" },
    "3": { "value": "12px" },
    "4": { "value": "16px" },
    "5": { "value": "20px" },
    "6": { "value": "24px" },
    "7": { "value": "32px" },
    "8": { "value": "56px" },
    "9": { "value": "72px" }
  }
}
```

An example with size tokens

At Back Market, we used the following spacing tokens:

```
{
  "spacing": {
    "0": { "value": "0" },
    "px": { "value": "1px" },
    "1": { "value": "4px" },
    "2": { "value": "8px" },
    "3": { "value": "12px" },
    "4": { "value": "16px" },
    "5": { "value": "20px" },
    "6": { "value": "24px" },
    "7": { "value": "32px" },
    "8": { "value": "56px" },
    "9": { "value": "72px" }
  }
}
```

Can't insert `2px`

Can't insert `10px`

Can't insert `40px` or `48px`

An example with size tokens

We decided to switch to a new scale so that

- We would have a consistent relation between names and values
- The system would remain stable as we add new tokens
- `spacing-4` → `4px`, `spacing-12` → `12px` and so on

An example with size tokens

We decided to switch to a new scale so that

- We would have a consistent relation between names and values
- The system would remain stable as we add new tokens
- `spacing-4` → `4px`, `spacing-12` → `12px` and so on

Name conflicts between old and new system

- `spacing-4` used to mean `16px` and becomes `4px`
- `spacing-1` used to mean `4px` and becomes `1px`
- We had to rename `spacing-4` to `spacing-16` before `spacing-1`, etc.

CSS class substitution

Our stack is Vue and Tailwind, so we'll use the vue-sfcmod engine, and we'll target calls to Tailwind classes in two code paths.

- In HTML template, the content of `class` attributes

```
<Icon class="w-6 h-6 p-2" />
```

- In JS, the parameters of calls to `tw` tagged templates

```
const myClasses = computed(() => [
  tw`w-6 h-6 p-2`,
  hasIcon && tw`pr-5`,
  isDisabled && tw`bg-primary-disabled text-primary-disabled`,
])
```

We'll extract and rewrite CSS class lists in these code paths.

The HTML template codemod

```
function templateTransformer(ast, api, options) {
  api
    .findAstAttributes(ast, ({ name }) => name === 'class')
    .forEach((attr) =>
      api.updateAttribute(attr, ({ value }) => {
        return value
          ? { value: transformClasses(value.content) }
          : {}
      }),
    )

  return ast
}
```

The HTML template codemod

```
function templateTransformer(ast, api, options) {
  api
    .findAstAttributes(ast, ({ name }) => name === 'class')
    .forEach((attr) =>
      api.updateAttribute(attr, ({ value }) => {
        return value
          ? { value: transformClasses(value.content) }
          : {}
      }),
    )

  return ast
}
```

```
<Icon class="w-6 h-6 p-2" />
```

The HTML template codemod

```
function templateTransformer(ast, api, options) {
  api
    .findAstAttributes(ast, ({ name }) => name === 'class')
    .forEach((attr) =>
      api.updateAttribute(attr, ({ value }) => {
        return value
          ? { value: transformClasses(value.content) }
          : {}
      }),
    )

  return ast
}
```

```
<Icon class="w-24 h-24 p-8" />
```

The Vue script codemod

```
function jsTransformer(file, api) {
  const j = api.jscodeshift
  const root = j(file.source)

  root
    .find(j.TaggedTemplateExpression, { tag: { name: 'tw' } })
    .forEach((path) => {
      path.value.quasi.quasis.forEach((quasi) => {
        quasi.value.raw = transformClasses(quasi.value.raw)
      })
    })

  return root.toSource()
}
```

The Vue script codemod

```
function jsTransformer(file, api) {
  const j = api.jscodeshift
  const root = j(file.source)

  root
    .find(j.TaggedTemplateExpression, { tag: { name: 'tw' } })
    .forEach((path) => {
      path.value.quasi.quasis.forEach((quasi) => {
        quasi.value.raw = transformClasses(quasi.value.raw)
      })
    })

  return root.toSource()
}
```

```
const myClasses = computed(() => [ tw`w-6 h-6 p-2`, hasIcon && tw`pr-5` ])
```

The Vue script codemod

```
function jsTransformer(file, api) {
  const j = api.jscodeshift
  const root = j(file.source)

  root
    .find(j.TaggedTemplateExpression, { tag: { name: 'tw' } })
    .forEach((path) => {
      path.value.quasi.quasis.forEach((quasi) => {
        quasi.value.raw = transformClasses(quasi.value.raw)
      })
    })

  return root.toSource()
}
```

```
const myClasses = computed(() => [ tw`w-24 h-24 p-8`, hasIcon && tw`pr-20` ])
```

transformClass in plain English

1. We split the `class` string into individual CSS classes
2. We replace old tokens with new ones, avoiding conflicts
3. If a class uses a hardcoded value, we look for a replacement
4. We bundle all these new classes into a `class` string and return it

```
w-6  md:w-8  h-[40px]  p-1  md:p-4
```

transformClass in plain English

1. We split the `class` string into individual CSS classes
2. We replace old tokens with new ones, avoiding conflicts
3. If a class uses a hardcoded value, we look for a replacement
4. We bundle all these new classes into a `class` string and return it

```
w-6  md:w-8  h-[40px]  p-1  md:p-4
```

transformClass in plain English

1. We split the `class` string into individual CSS classes
2. We replace old tokens with new ones, avoiding conflicts
3. If a class uses a hardcoded value, we look for a replacement
4. We bundle all these new classes into a `class` string and return it

w-6 md:w-8 h-[40px] p-1 md:p-4

transformClass in plain English

1. We split the `class` string into individual CSS classes
2. We replace old tokens with new ones, avoiding conflicts
3. If a class uses a hardcoded value, we look for a replacement
4. We bundle all these new classes into a `class` string and return it

w-6 md:w-56 h-[40px] p-1 md:p-4

transformClass in plain English

1. We split the `class` string into individual CSS classes
2. We replace old tokens with new ones, avoiding conflicts
3. If a class uses a hardcoded value, we look for a replacement
4. We bundle all these new classes into a `class` string and return it

w-24 md:w-56 h-[40px] p-1 md:p-4

transformClass in plain English

1. We split the `class` string into individual CSS classes
2. We replace old tokens with new ones, avoiding conflicts
3. If a class uses a hardcoded value, we look for a replacement
4. We bundle all these new classes into a `class` string and return it

w-24 md:w-56 h-[40px] p-1 md:p-16

transformClass in plain English

1. We split the `class` string into individual CSS classes
2. We replace old tokens with new ones, avoiding conflicts
3. If a class uses a hardcoded value, we look for a replacement
4. We bundle all these new classes into a `class` string and return it

w-24 md:w-56 h-[40px] p-4 md:p-16

transformClass in plain English

1. We split the `class` string into individual CSS classes
2. We replace old tokens with new ones, avoiding conflicts
3. If a class uses a hardcoded value, we look for a replacement
4. We bundle all these new classes into a `class` string and return it

w-24 md:w-56 h-40 p-4 md:p-16

transformClass in plain English

1. We split the `class` string into individual CSS classes
2. We replace old tokens with new ones, avoiding conflicts
3. If a class uses a hardcoded value, we look for a replacement
4. We bundle all these new classes into a `class` string and return it

```
w-24  md:w-56  h-40  p-4  md:p-16
```

Data structures driving the transformation

Value map between old and tokens

```
const sizeMap = {
  9: '72',
  8: '56',
  7: '32',
  6: '24',
  5: '20',
  4: '16',
  3: '12',
  2: '8',
  1: '4',
  px: '1',
}
```

List of tokens to transform (or preserve)

`height`, `width`, `padding`, `margin`, `border-width`, `inset`, etc.

Okay, but...

Was it worth it?

Was it worth it? Yes!

First migration 💰

- ~5 days to create and optimise the codemod
- Over 2500 files migrated in the DS and Web apps
- Over 4000 token uses modified (estimated)

Was it worth it? Yes!

First migration 💰

- ~5 days to create and optimise the codemod
- Over 2500 files migrated in the DS and Web apps
- Over 4000 token uses modified (estimated)

Compound benefits 💰💰💰

- Token migrations can be boiled down to value mapping
- So the next migration codemods were trivial to write
  - Typography token changes
  - Migrating `rem` sizes from `10px` to `16px`
- Still migrating thousands of files when running them

Wrapping up

Codemods are a powerful tool for component library maintenance.

In the right situations, codemods can save you weeks of work. They help design tokens deliver on the promise of automation.

Thank you!