# The art of tree-shaking

Eric Fennis - DEVWorld 2024

# About me

Living in:

🇳🇱 The Netherlands

Creator of:

Lucide

Follow me:

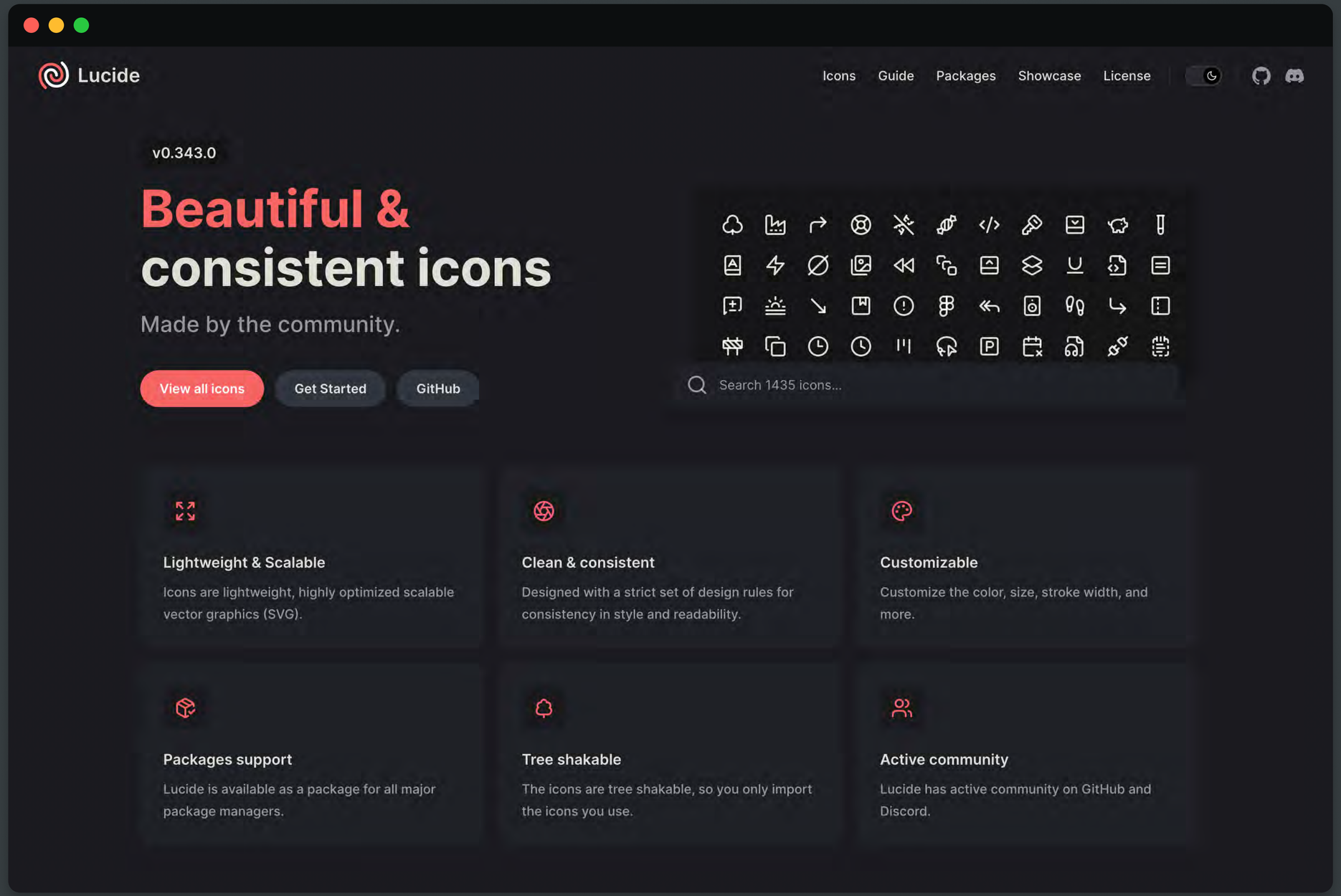
@ericfennis

Eric Fennis

Senior Frontend Engineer

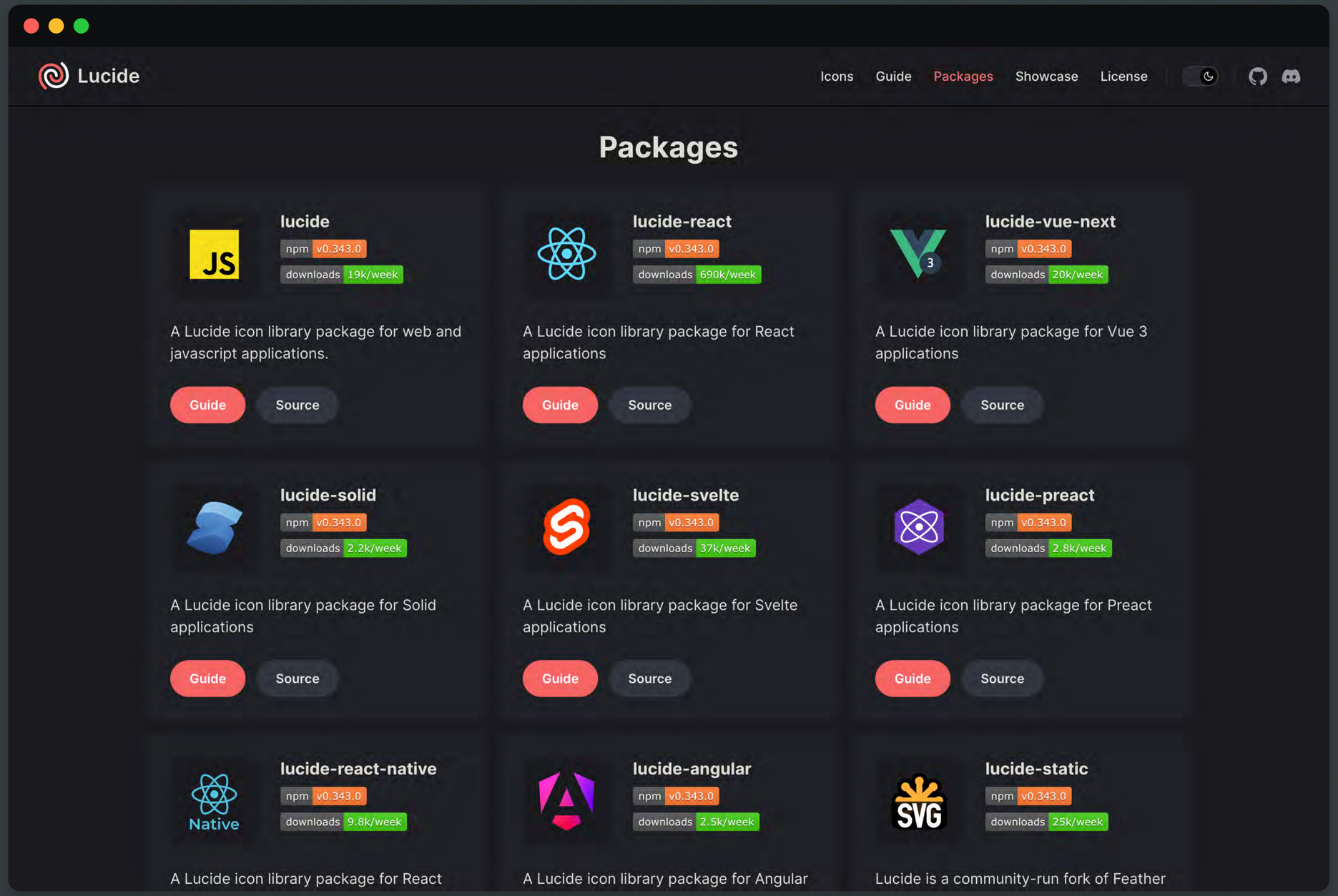
nedap

When building a web-app with Lucide you shouldn't include all 1400+ icons. This harms performance because the bundle size will be quite big. Tree-shaking is helping us with this.



All packages from Lucide are fully tree-shakable.

I've written most of the Lucide packages.



# Tree-shaking

# What is tree-shaking?

- A Form of dead-code elimination
- Part of the build process
- Performed by bundlers like:


PARCEL

# Why tree-shaking?

- Removes unused modules
- Reduce your bundle size
- Improve load performance
- Improve Core Web Vitals

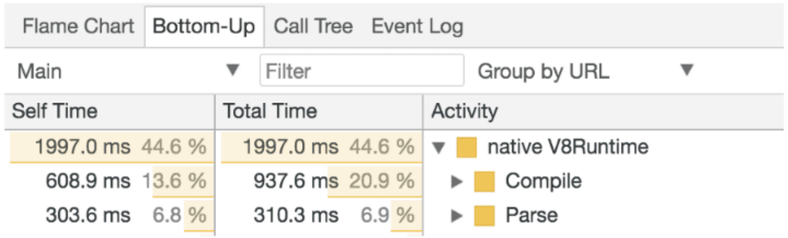
# JavaScript bytes !== JPEG bytes

~170KB ~170KB

## Network Transmission



## Resource Processing

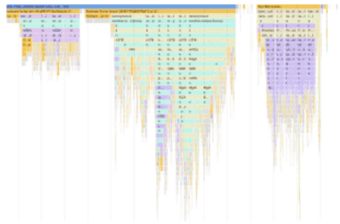




~2s in Parse/Compile

0.064s in Image Decode

~1.5s in Execution




0.028s in Rasterize Paint

@addyosmani - 170KB of (compressed) JS vs. JPEG bytes over a slow 3G network on a Moto G4. JS needing parsed is even larger once decompressed.

The processing cost of parsing/compiling 170 KB of JavaScript vs decode time of an equivalently sized JPEG. (source).

# How does it work?

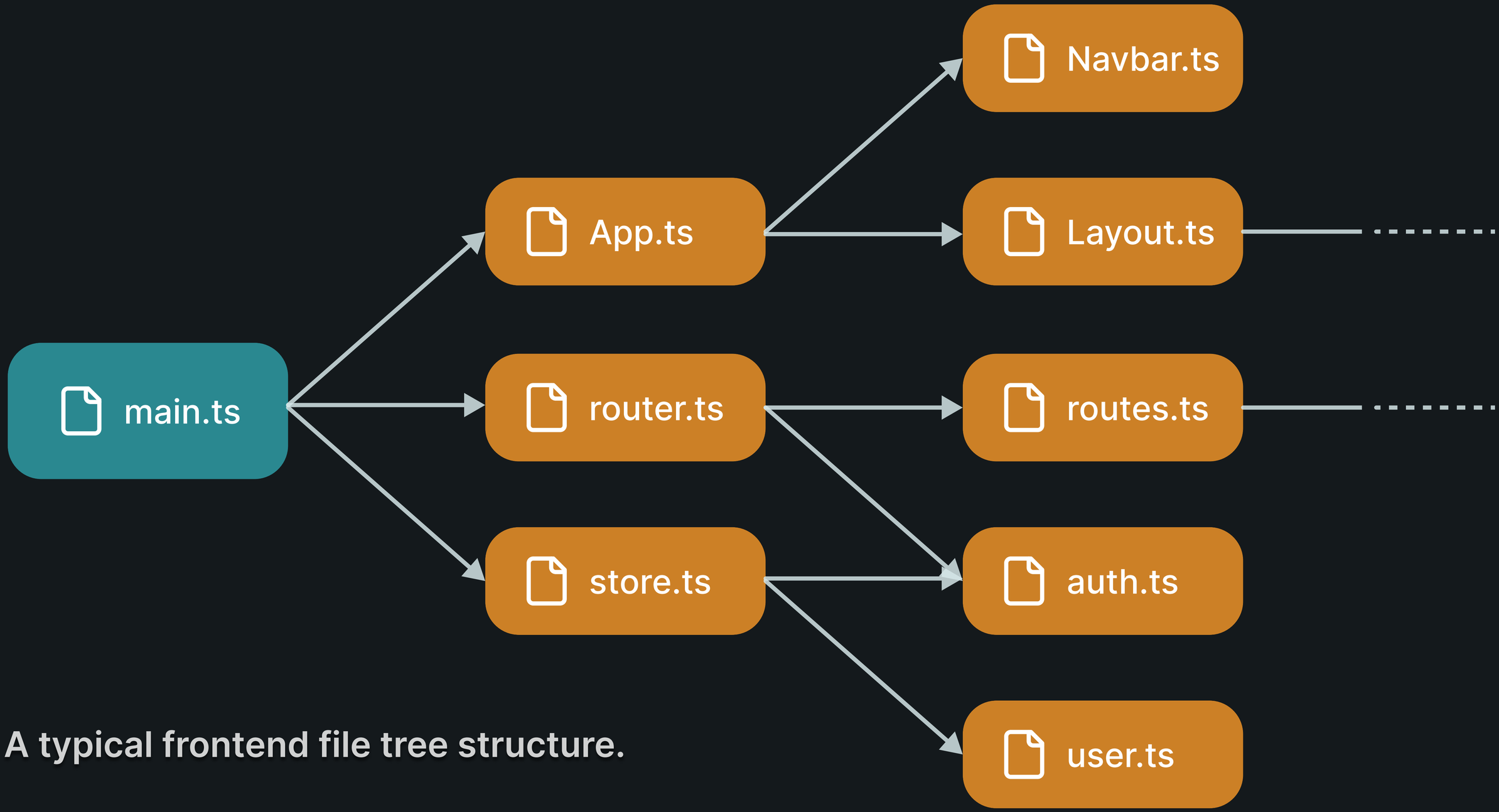
```
> npm run build
```


Tree-shaking is mostly done out of the box.
If everything is set up correctly.

# Bundling steps

**Initialize**  **Build**  **Generate**

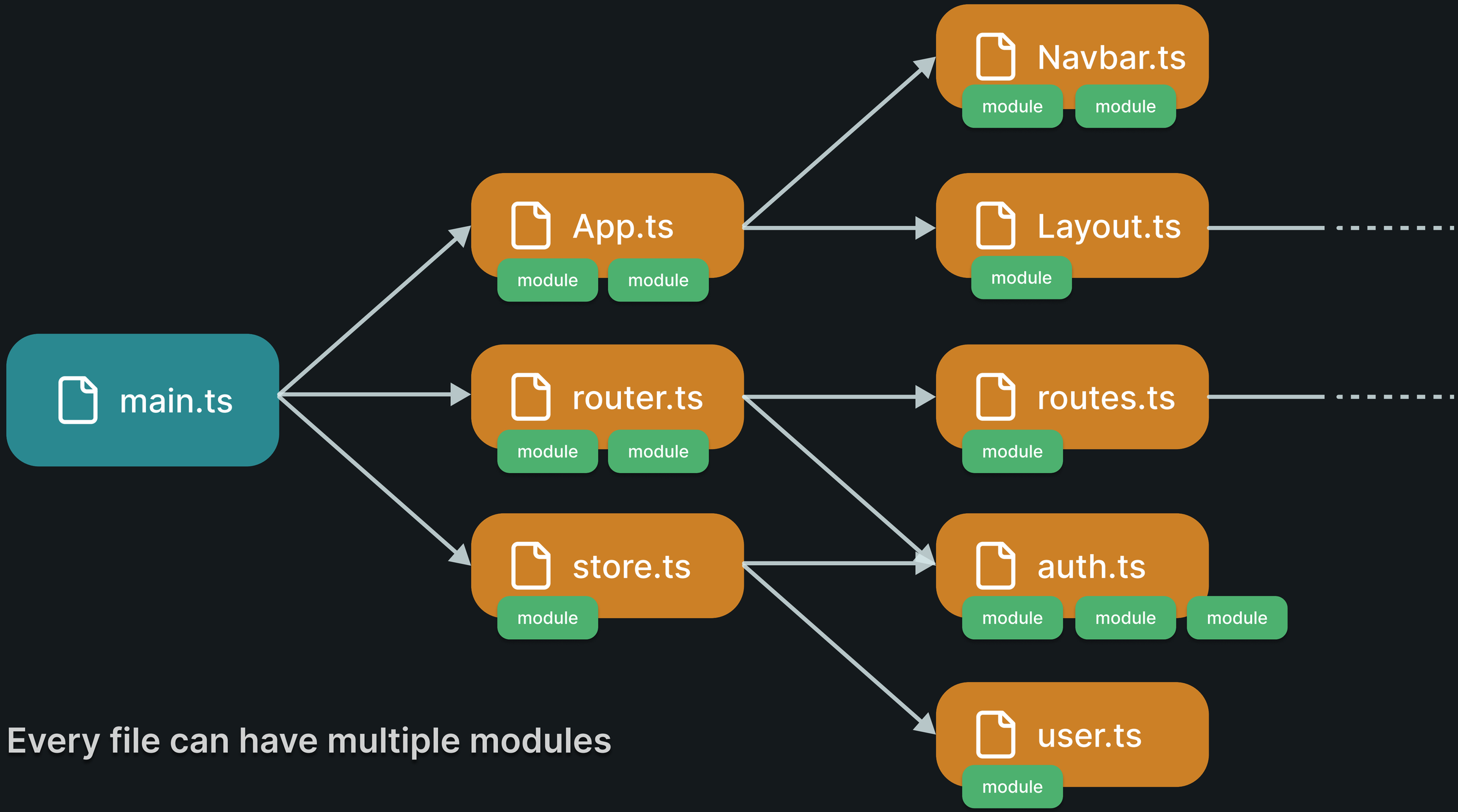
During bundle process
tree-shaking is happening in the build process.



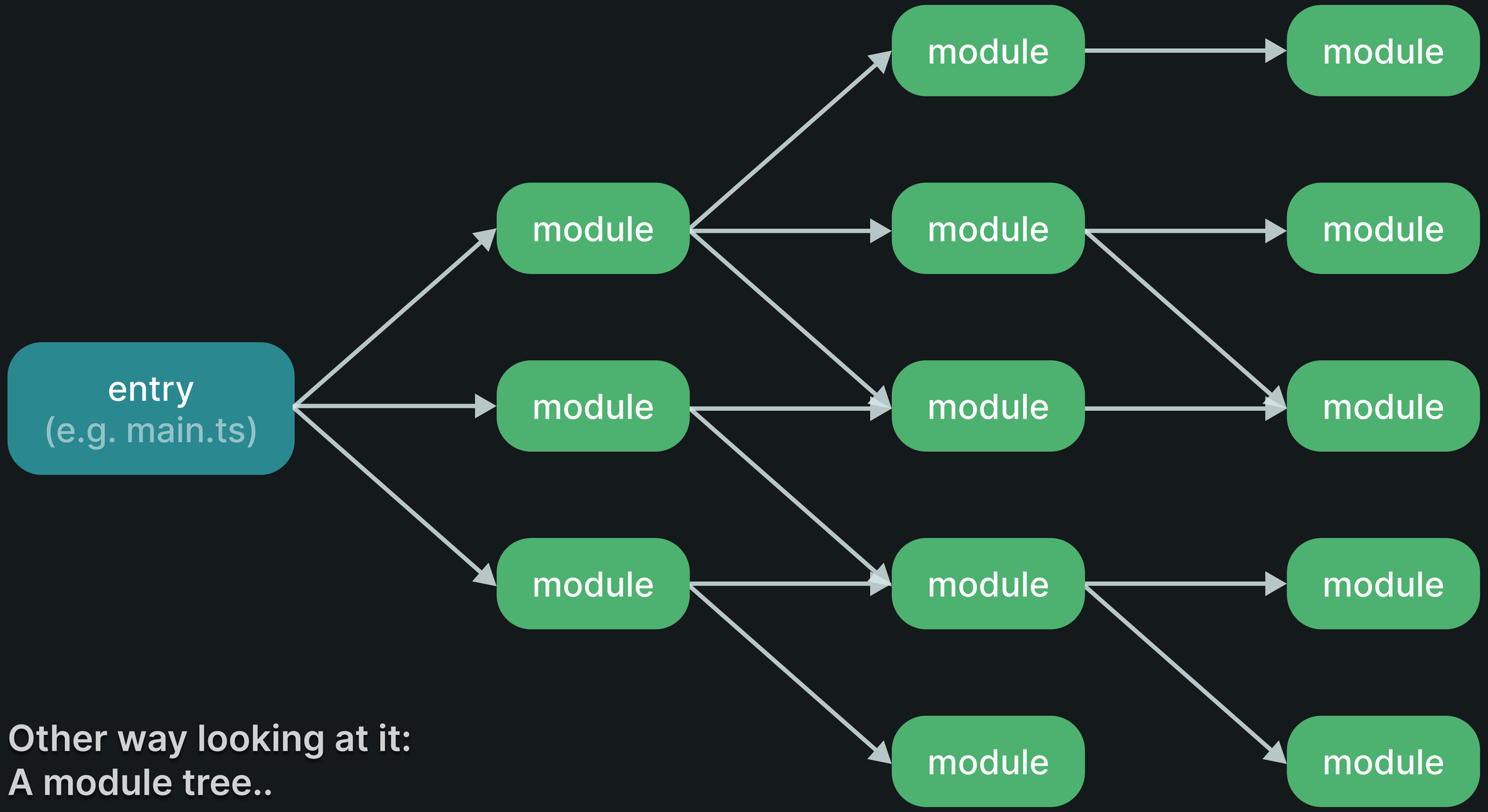

A typical frontend file tree structure.

```ts
// calculate-functions.ts

export const add = (a: number, b: number): number => {
  return a + b;
};

export const subtract = (a: number, b: number): number => {
  return a - b;
};

export const multiply = (a: number, b: number): number => {
  return a * b;
};

export const divide = (a: number, b: number): number => {
  return a / b;
};
```

Every time you use the “export” keyword in front of const, function, etc.
You create a new module within your file.




Every file can have multiple modules




Other way looking at it:
A module tree..

```
import { add } from './calculate-functions';

function calculateStuff() {
  const a = 10;
  const b = 5;

  const sum = add(a, b);

  return sum;
}
```

When importing one module at the top of a file.
During build time the bundler will analyze your file
and detect which module is used from that file.
Other unused modules will be removed from the module tree.
What we call tree-shaking!

# The bundle flow

## Initialize

- Load config
- Validate config
- Inject plugins
- Resolve entry files
- Start

## Build

- Reads entry file
- Resolve module imports
- Create module
- Build module
- Resolve module imports
- Add dependencies
- Create module tree

Process & analyze dependencies

## Generate

- Generate chunk graph
- Transform modules to chunks
- Generate assets
- Create bundle files

**Tree-shaking**

# Setup for tree-shaking

The next part includes the required setup to perform tree-shaking.

# Use ESM syntax

```
// Don't
const router = require('./router');

// Do
import router from './router';
```

Using ESM import syntax is “required”
to make tree-shaking work.

# Watch out for side-effects

```
// Side Effect free

// Pure module:
export const calculate = (a:number, b:number) => a + b;
```

Pure modules are tree-shakable
All the code in a module is scoped

```
// With side effects

window.results.number = 4

// Impure module:
export const calculate = (c:number) => window.results.number + c;
```

Impure modules contain code that relies
on factors external to the scope of execution.

Tree-shaking requires pure modules. If side-effects are detected
bundlers won’t perform tree-shaking properly.

# Declare sideEffects in package.json

```
// package.json
{
  "name": "my-project",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
  },
  "sideEffects": ["./src/script-with-side-effects.js"]
}
```

Files with side-effects can be declared in the package.json.
Bundler will read this file. Even wildcards are supported.

# Declare sideEffects in package.json

```
// package.json
{
  "name": "my-project",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
  },
  "sideEffects": false
}
```

If you are 100% sure you could set it to false.
Only recommend this when making packages.
Otherwise, some unwanted tree-shaking can happen.

# Writing tree-shakable code

The next part is code examples to write code
as tree-shakable as possible.
This can be done by writing your code as modular as possible.

```tsx
const Card = ({ header, children, footer }: CardProps) => (
  <div className="card">
    {header ? (
      <div className="card-header">
        {header}
      </div>
    ) : null}

    <div className="card-body">
      {children}
    </div>

    {footer ? (
      <div className="card-footer">
        {footer}
      </div>
    ) : null}
  </div>
);

export default Card;
```

A typical card component in React.
When writing a component with an if statement, that you may never use for example the “footer” part in your codebase this “dead” code always will be included in your build.

Yes, you can just remove this, but you may not be sure if this will be used in the future, or in larger codebases where these components are found often, you are not sure if it is used or not.

# Split up in smaller components

```
export const CardHeader = ({ children }: CardHeaderProps) => (
  <div className="card-header">
    {children}
  </div>
);

export const CardBody = ({ children }: CardBodyProps) => (
  <div className="card-body">
    {children}
  </div>
);

export const CardFooter = ({ children }: CardFooterProps) => (
  <div className="card-footer">
    {children}
  </div>
);

const Card = ({ children }: CardProps) => (
  <div className="card">
    {children}
  </div>
);

export default Card;
```

By splitting up this component into separate components and using the “export” keyword. All these components became separate modules, which can be tree-shaken!

# JSX dot notation, do not use it.

```tsx
export const CardHeader = () => ...

export const CardBody = () => ...

export const CardFooter = () => ...

const Card = ({ children }: CardProps) => (
  <div className="card">
    {children}
  </div>
);

Card.Header = CardHeader;
Card.Body = CardBody;
Card.Footer = CardFooter;

export default Card;
```

```tsx
const MyComponent = () => (
  <Card>
    <Card.Header>...</Card.Header>
    <Card.Body>...</Card.Body>
    <Card.Footer>...</Card.Footer>
  </Card>
);
```

Do not use the JSX dot notation. Yes, this is nice to use, but you blocking the bundler of tree-shaking this code.

Because all the other components became part of the Card module itself.

# Import all modules from a file.
# Will include all modules during build

```
import * as icons from 'lucide-react'

const Button = ({ children, onClick, icon }: ButtonProps) => {
  const Icon = icons[icon];

  return (
    <button className="button" onClick={onClick}>
      {icon && <Icon />}
      {children}
    </button>
  );
}

export default Button;
```

Caution when using “import * from ..”.

Importing all modules will significantly increase the bundle size of the application.

```
// Theme colors from tailwind.
export const ThemeColors = {
  inherit: 'inherit',
  current: 'currentColor',
  transparent: 'transparent',
  black: '#000',
  white: '#fff',
  slate: {
    '50': '#f8fafc',
    '100': '#f1f5f9',
    '200': '#e2e8f0',
    '300': '#cbd5e1',
    '400': '#94a3b8',
    '500': '#64748b',
    '600': '#475569',
    '700': '#334155',
    '800': '#1e293b',
    '900': '#0f172a',
    '950': '#020617',
  },
  gray: {
    '50': '#f9fafb',
    '100': '#f3f4f6',
    '200': '#e5e7eb',
    '300': '#d1d5db',
    '400': '#9ca3af',
    '500': '#6b7280',
    '600': '#4b5563',
    '700': '#374151',
    '800': '#1f2937',
    '900': '#111827',
    '950': '#030712',
```

# Avoid importing large static objects

This is just one giant module, and won't be tree-shaken if you use only a couple of properties.
Instead, just cherry-pick some.

```
export const ThemeColors = {
  black: '#000',
  white: '#fff',
  slate: '#64748b',
  gray:'#6b7280',
  zinc: '#71717a',
  neutral: '#737373',
  stone: '#78716c',
  red: '#ef4444',
  orange: '#f97316',
  amber: '#f59e0b',
  yellow: '#eab308',
};
```

# Same for json files.

```
import React from 'react';
import packageJson from '../../package.json';

const AppVersion = () => {
  const version = packageJson.version;

  return (
    <div className="app-version">
      App Version: {version}
    </div>
  );
};
```

If you for example include your package.json file in your app.
You may not know that you also include all other information in this file, like your “dependency” information.
This can be a security risk.

# If you want  for example the version, use ENV variables.

```
import React from 'react';

const AppVersion = () => {
  const version = process.env.APP_VERSION || '1.0.0';

  return (
    <div className="app-version">
      App Version: {version}
    </div>
  );
};

export default AppVersion;
```

In your bundle config, you can declare those environment variables.

# Using Classes

```
class Utils {
  static add(a: number, b: number): number {
    return a + b;
  }

  static subtract(a: number, b: number): number {
    return a - b;
  }

  static multiply(a: number, b: number): number {
    return a * b;
  }

  static divide(a: number, b: number): number {
    return a / b;
  }
}

export default Utils;
```

Only use classes for
object-oriented programming.
Not for static functions or variables.

Classes are just one giant module.
If you maybe use half of the functions the rest of the code will be included in your bundle.

# Using Classes

```ts
export class MyClient {
  static baseUrl: string = 'https://my-list.com/todo';

  static fetch(url: string, method: string, body?: any) {
    return fetch(`${this.baseUrl}/${url}`, {
      ..,
    })
  }

  static get(id: string) {
    return this.fetch(id, 'GET')
  }

  static post(body: Body) {
    return this.fetch('new', 'POST', body)
  }

  static put(id: string, body: Body) {
    return this.fetch(id, 'PUT', body)
  }

  static delete(id: string) {
    return this.fetch(id, 'DELETE')
  }
}
```

This more common example is a fetch client class. Imagine having 20+ client classes. Same with this example if you do use not all the code, the rest will be included in the code.
Instead, just create separate module functions. These are tree-shakable.

```ts
const baseUrl = 'https://my-list.com/todo';

const fetcher = (
  url: string,
  method: string,
  body?: any
) => fetch(`${baseUrl}/${url}`, {
  ...
});

export const getTodo = (id: string) => fetcher(id, 'GET');

export const createTodo = (body: Body) => fetcher('new', 'POST', body);

export const updateTodo = (id: string, body: Body) => fetcher(id, 'PUT', body);

export const deleteTodo = (id: string) => fetcher(id, 'DELETE');
```

# Use dependency injection

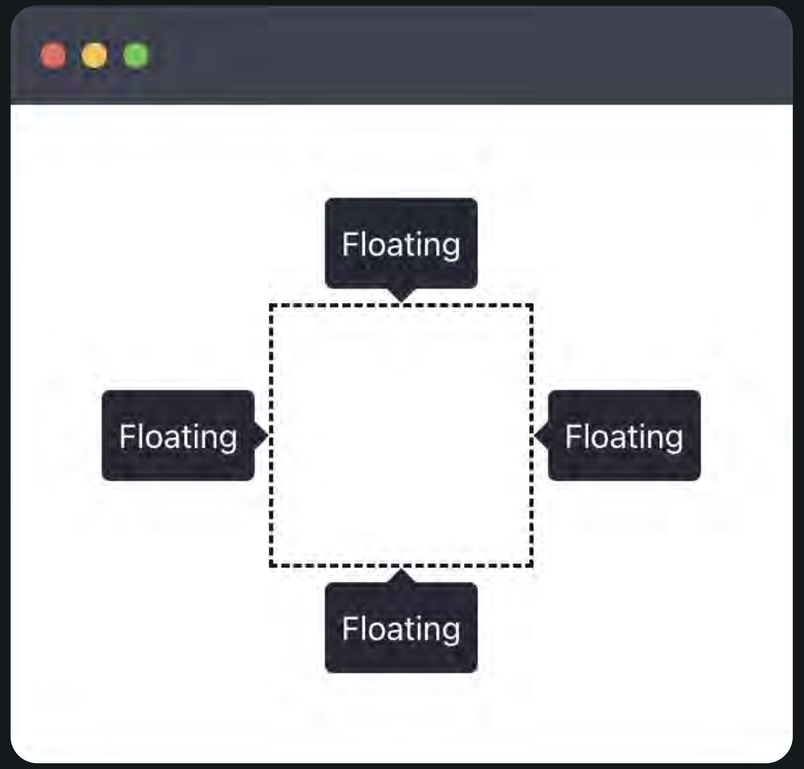
```
import { Input, Select, Checkbox } from '@/components';

const Page = () => {
  const formConfig = {
    fields: [
      ...
    ],
  };

  const handleFormSubmit = (formData: FormData) => {
    ...
  };

  return (
    <div>
      <FormGenerator
        formConfig={formConfig}
        onSubmit={handleFormSubmit}
        components={{
          text: Input,
          select: Select,
          checkbox: Checkbox,
        }}
      />
    </div>
  );
};
```

If you want to make a form component for example that can dynamically render forms based on input.

Instead of including all form fields within the FormGenerator component. We could use the code pattern “dependency injection” to make the form input elements tree-shakable by injecting them as separate modules.

This can also be done with other things like validation.

This example may be less convenient when the form config is coming from an external source like a database.


**Floating UI is a great example of using dependency injection.**
**This is a popular library for creating floating UI elements like tooltips.**

# Floating UI, useFloating function

This library can be used for floating and placing UI elements relative to a source element. Floating UI can be used in its minimal form. But if you want to extend it with extra features like “flip” behavior or an arrow for your tooltip, you can declare middleware.
This is dependency injection, only the used middleware plugins are included in the build.

```
import { useFloating, flip, arrow } from '@floating-ui/react';


const {refs, floatingStyles} = useFloating({
  placement: 'right',
  middleware: [
    flip(),
    arrow()
  ]
});
```

# Good to know

# Check you NPM packages

**Some packages are not tree-shakable**

- Axios
- Moment.js (maintenance mode)
- Lodash (use lodash-es)

These are popular libraries that still are written in CommonJS. There are great alternatives that are tree-shakable.

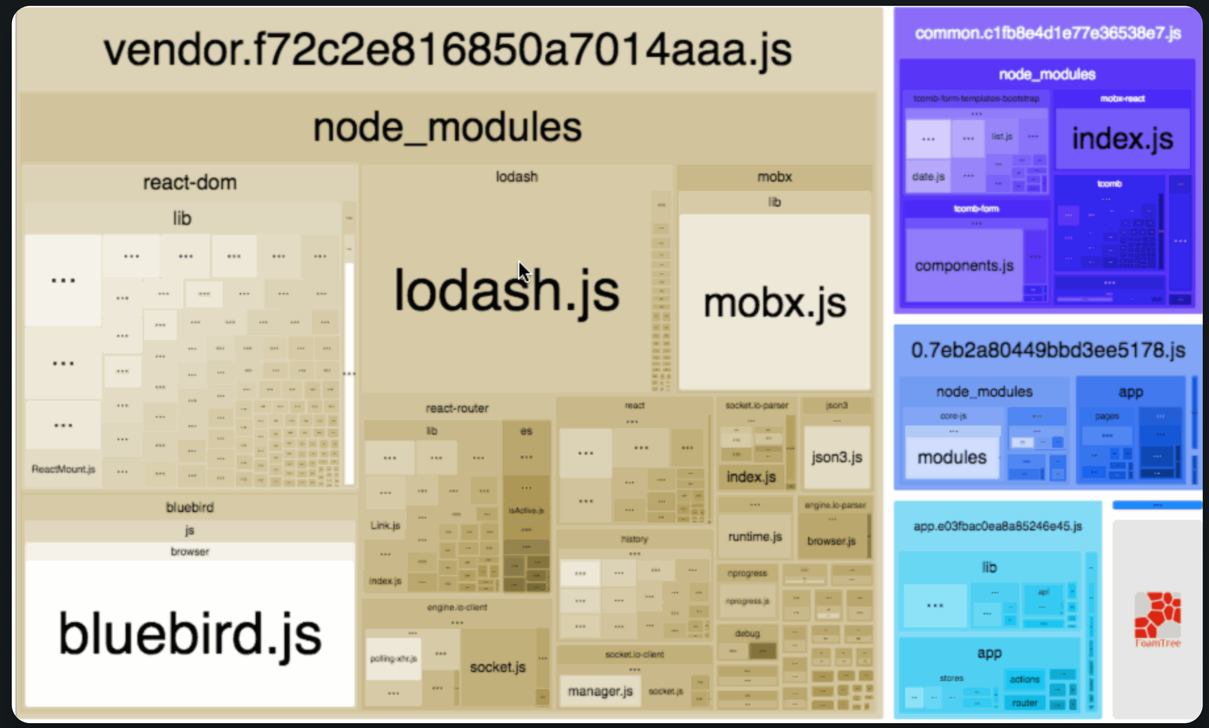
# Use bundle analyzer/visualiser

**For Vite/Rollup:**

- rollup-plugin-visualizer

**For Webpack:**

- webpack-bundle-analyzer




This plugin can generate a modules overview of
all the modules in your bundle. A great tool to check if things are tree-shaken

# Covering up

- Code need to be written as ESModules
- Watch out for side-effects
- Write your code as modular as possible
- Check you NPM packages
- Use a bundle analyser to check your output

# Happy Tree-shaking!

# Thank you!

Follow me:

@ericfennis

Eric Fennis

Senior Frontend Engineer

nedap