Proven Patterns for Building Vue.js Apps

Ben Hong

@bencodezen

VUECONF TORONTO - 2020

Introduction


UNIVERSAL'S
ISLANDS OF ADVENTURE
QUKSILVER
JURASSIC PARK
RIVER ADVENTURE
UNIVERSAL'S
ISLANDS OF ADVENTURE..
November 12, 2017
TM & © Universal Studios and Amblin


QUKSILVER
JURASSIC PARK
RIVER ADVENTURE
UNIVERSAL'S
ISLANDS OF ADVENTURE™
November 12, 2017
TM & © Universal Studios and Amblin

My background

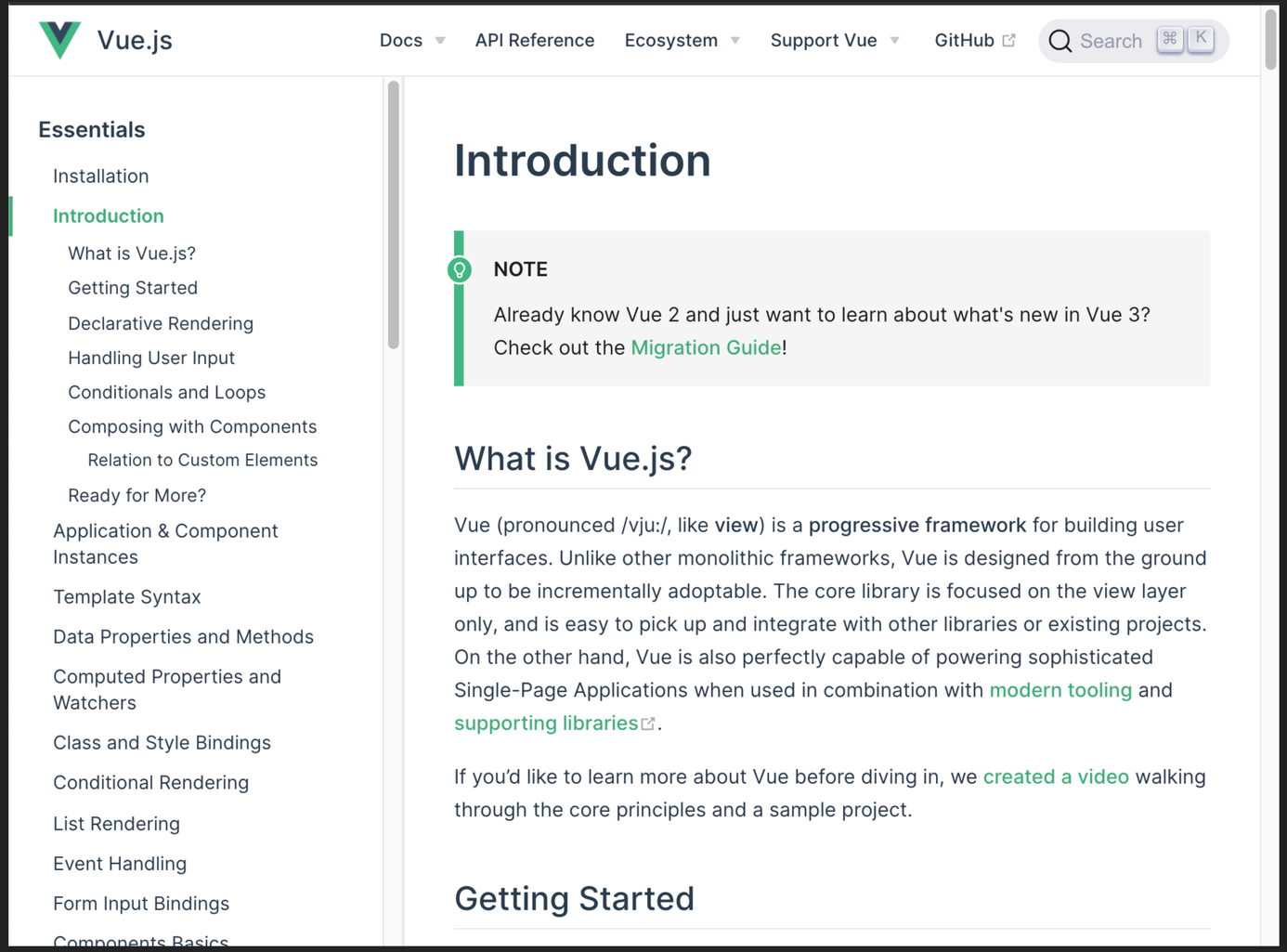
What's my experience?

BEN HONG

Senior DX Engineer

Netlify

@bencodezen



Zoom Onboarding

A little bit about you…

WORKSHOP FORMAT

FORMAT

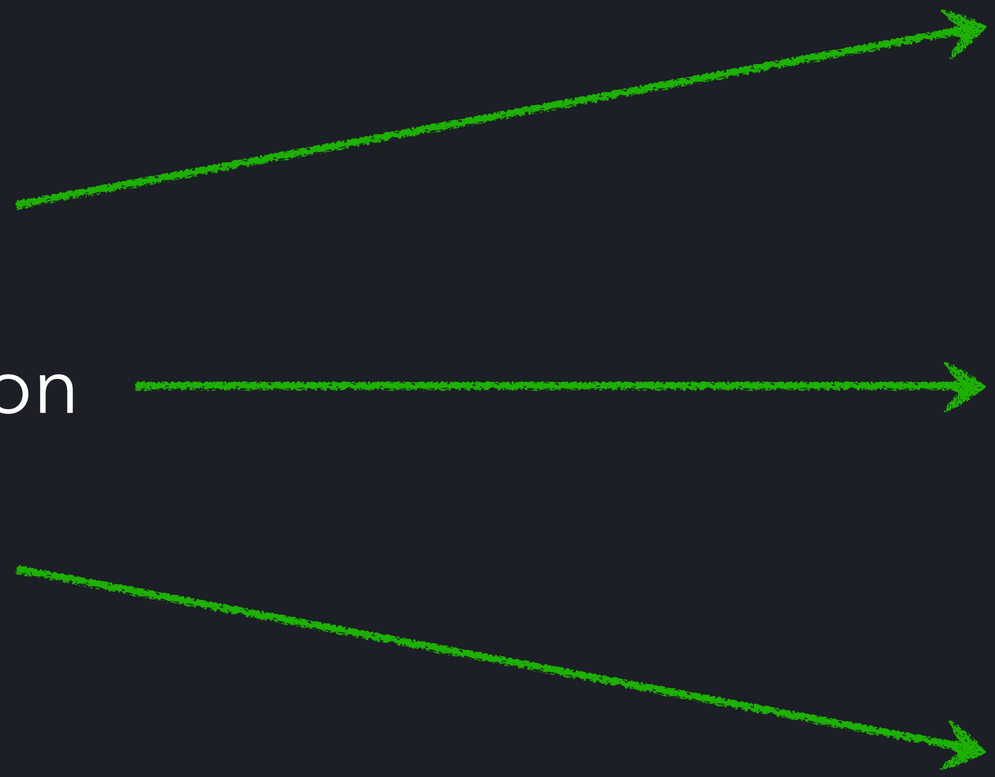
1.Learn

2.Question

3.Apply

FORMAT

1. Learn
2. Question
3. Apply



Explanations

Examples

Clarifications

What-ifs

Exercises

Experiment

SCHEDULE

Time Slot (EST)	Event
9:00AM - 10:30AM	Part 1
10:30AM - 10:45AM	☕ Coffee Break ☕
10:45AM - 12:00PM	Part 2
12:00PM - 1:00PM	🥪 Lunch 🥗
1:00PM - 2:45PM	Part 3
2:45PM - 3:00PM	☕ Coffee Break ☕
3:00PM - 4:30PM	Part 4: Accessibility
4:30PM - 5:00PM	Final Thoughts + Q&A

Resources

Vue Enterprise Boilerplate GitHub Repo

https://github.com/chrisvfritz/vue-enterprise-boilerplate

Proven Patterns GitHub Repo

https://github.com/bencodezen/proven-patterns-workshop

Your App!

Before we get started...

PARTICIPATION TIPS

Please **no recording**.

Out of respect for you and your classmates' privacy

PARTICIPATION TIPS

Raise your hand for **questions** at **any time**!

There are no wrong questions.

This workshop is designed to be interactive.

PARTICIPATION TIPS

All slides and examples will be **public**.

No need to hurry and copy down examples.

PARTICIPATION TIPS

If you need a **break**, please take one!

There is going to be a lot of information

I encourage you to take notes to stay engaged

But seriously, if you need a break, take time for yourself.

PARTICIPATION TIPS

All code is **compromise**.

I encourage you to question and/or disagree.

Your opinion and experience matter too.

Choose what works best for you and your team.

Questions?

Languages

BEST PRACTICES

JavaScript

JavaScript or TypeScript?

JavaScript or TypeScript?

“The vast majority of bugs encountered at work are not often due to type violations. If bugs is what you're aiming to solve: linting, tests, and code reviews are the way forward - none of which are made easier by TypeScript by itself.”

For a more in depth explanation, visit this link

JavaScript or TypeScript?

- TypeScript in Vue 2 works, but has some pain points, particularly with integrations like Vue

- Vue 3 has been rewritten from the ground up with TypeScript, which means support for TypeScript will be dramatically improved; but it still an **opt-in feature and not required**

JavaScript or TypeScript?

- As time goes on, there will most likely be big advancements in TypeScript tooling support (especially for Vue 3).

- That said, the key to remember here is asking yourself what problem you're trying to solve by using TypeScript.

Vue CLI Modern Mode

Vue CLI Modern Mode

- With Babel, Vue CLI is able to leverage all the newest language features in ES2015+, this meant that we were shipping transpiled and polyfilled bundles in order to support older browsers.
- By adding the flag to your build command, this allows Vue CLI to produce two versions of your app:

  - One modern bundle that targets modern browsers
  - And one legacy bundle targeting older browsers that do not

- The result is that even for a Hello World app, resulted in the bundle being 16% smaller. Not to mention that modern bundles will also result in significantly faster parsing and evaluation, improving your app's loading performance.
-

For a more in depth explanation, visit this link

Polyfill.io

- A fantastic service that can help you support older browser:
  - Define which browser features you need to have for your application
  - It will return only the polyfills needed by the requesting browser

BEST PRACTICES

HTML

Templates vs Render Functions

Templates vs Render Functions

- Most of the time you want to be sticking with templates
- Templates are fully declarative (typically only 1 right way to do something)
- Render functions are more flexible and can sometimes be easier to maintain

Functional Components

Functional Components (v2)

Stateless and instance-less

(no data, computed, etc and no lifecycle)

```
<template functional>
  <!-- ... -->
</template>
```

or

```
export default {
  functional: true,
  render (h, context) {
    // ...
  }
}
```

Functional Components (v2)

Improve performance

(especially for components rendered many times, e.g. with a v-for)

Can return multiple root nodes

boilerplate: src/components/nav-bar-routes.vue

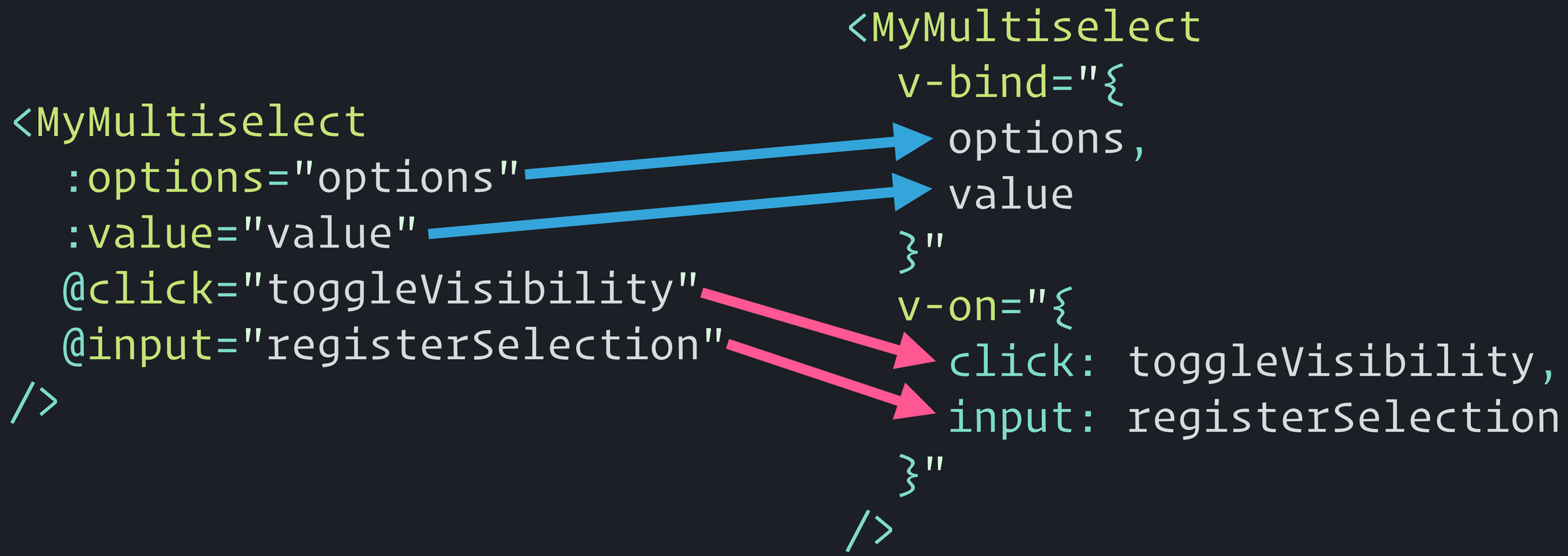
TECHNIQUE

```
v-bind="{ … }"
v-on="{ … }"
```

When working with multiple props or events, consider… the object syntax:

```
<MyMultiselect
  :options="options"
  :value="value"
  @click="toggleVisibility"
  @input="registerSelection"
/>
```

When working with multiple props consider...



BEST PRACTICES

CSS

CSS

Global CSS (usually) only in app.vue

Scope all component CSS
(using either scoped or module attributes)

```
<template>
  <p class="red">
    This should be red
  </p>
</template>


<style>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

TECHNIQUE

Scoped Styles

```
<template>
  <p class="red">
    This should be red
  </p>
</template>


<style scoped>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

```vue
<template>
  <p class="red">
    This should be red
  </p>
</template>

<style scoped>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

```vue
<template>
  <p class="red"
     data-f3f3eg9
  >
    This should be red
  </p>
</template>

<style scoped>
.red[data-f3f3eg9] {
  color: red;
}
.bold[data-f3f3eg9] {
  font-weight: bold;
}
</style>
```

TECHNIQUE

CSS Modules

```
<template>
  <p :class="$style.red">
    This should be red
  </p>
</template>

<style module>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

```
<template>
  <p :class="$style.red">
    This should be red
  </p>
</template>

<style module>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

```
<p class=“red-xhjd1">
  This should be red
</p>

<style>
.red-xhdj1 {
  color: red;
}
</style>
```

CSS Preprocessors

The lang attribute

```
<style lang="scss">
.my-class {
  & > .another-class {
    @extend %placeholder-class;

    &:hover {
      // ...
    }
  }
}
</style>
```

Questions?

Practice

In the boilerplate

- Add a `<label>` to the `_base-input-text.vue` component, then style that label with a CSS module class.
- Refactor the `_base-button.vue` component to use a render function instead of a template. Then, make it a functional component.

In your app

- Refactor a template to use a render function instead.
- Refactor a component to use CSS modules for scoping.
- Refactor an existing component that's rendered many times (e.g. with a `v-for`) to a functional component.

Coffee Break

Be back at 10:55AM!

DISCUSSION

Migrating from Vue 2 to Vue 3

DISCUSSION

Migrating from Vue 2 to Vue 3

- Vue 3 is ready for new projects that don’t need IE11 support.
- If you have lots of dependencies, wait until those are officially migrated
- Migration tool and IE11 combat build should be ready by the end of the year

Questions?

BEST PRACTICES

Components (Part 1)

BEST PRACTICE

Naming Components

Actual programming

Debating for 30 minutes on how to name a variable

Recommended **Naming Conventions**

Avoid single word components

~~Header.vue~~

~~Button.vue~~

~~Container.vue~~

Recommended **Naming Conventions**

AppPrefixedName.vue / **Base**PrefixedName.vue

Reusable, globally registered UI components.

`AppButton, AppModal, BaseDropdown, BaseInput`

ThePrefixedName.vue

Single-instance components where only 1 can be active at the same time.

`TheShoppingCart, TheSidebar, TheNavbar`

Recommended **Naming Convention**

Tightly coupled/related components

TodoList.vue

TodoListItem.vue

TodoListItemName.vue

1. Easy to spot relation
2. Stay next to each other in the file tree
3. Name starts with the highest-level words

More conventions:

https://vuejs.org/v2/style-guide/

- Single-file component filename casing
- Base component names
- Single-instance component names
- Tightly coupled component names

And more…

BEST PRACTICE

Naming Component Methods

Use descriptive names

❌ onInput

✅ updateUserName

Don’t assume where it will be called

```
updateUserName ($event) {
  this.user.name = $event.target.value
}
```

❌ Wrong

```
updateUserName (newName) {
  this.user.name = newName
}
```

✅ Correct

Prefer destructuring over multiple arguments

```
updateUser (userList, index, value, isOnline) {
  if (isOnline) {
    userList[index] = value
  } else {
    this.removeUser(userList, index)
  }
}
```

Prefer destructuring over multiple arguments

```
updateUser (userList, index, value, isOnline) {
  if (isOnline) {
    userList[index] = value
  } else {
    this.removeUser(userList, index)
  }
}
```

❌ Not recommended

```
updateUser ({ userList, index, value, isOnline }) {
  if (isOnline) {
    userList[index] = value
  } else {
    this.removeUser(userList, index)
  }
}
```

✅ Recommended

BEST PRACTICE

When to Refactor Your Components

Premature optimization is the root of all evil (or at least most of it) in programming.

- Donald Knuth

Data Driven Refactoring

Signs you need more components

- When your components are hard to understand
- You feel a fragment of a component could use its own state
- Hard to describe what what the component is actually responsible for

Components and how to find them?

- Look for similar visual designs
- Look for repeating interface fragments
- Look for multiple/mixed responsibilities
- Look for complicated data paths
- Look for *v-for* loops
- Look for large components

Questions?

PRO TIP

SFC Code Block Order

```
<template>
<!-- ... -->
</template>


<script>
export default {
// ...
}
</script>


<style>
/* ... */
</style>
```

```
<script>
export default {
// ...
}
</script>


<template>
<!-- ... -->
</template>

<style>
/* ... */
</style>
```

```
<script>
export default {
// ...
}
</script>
```

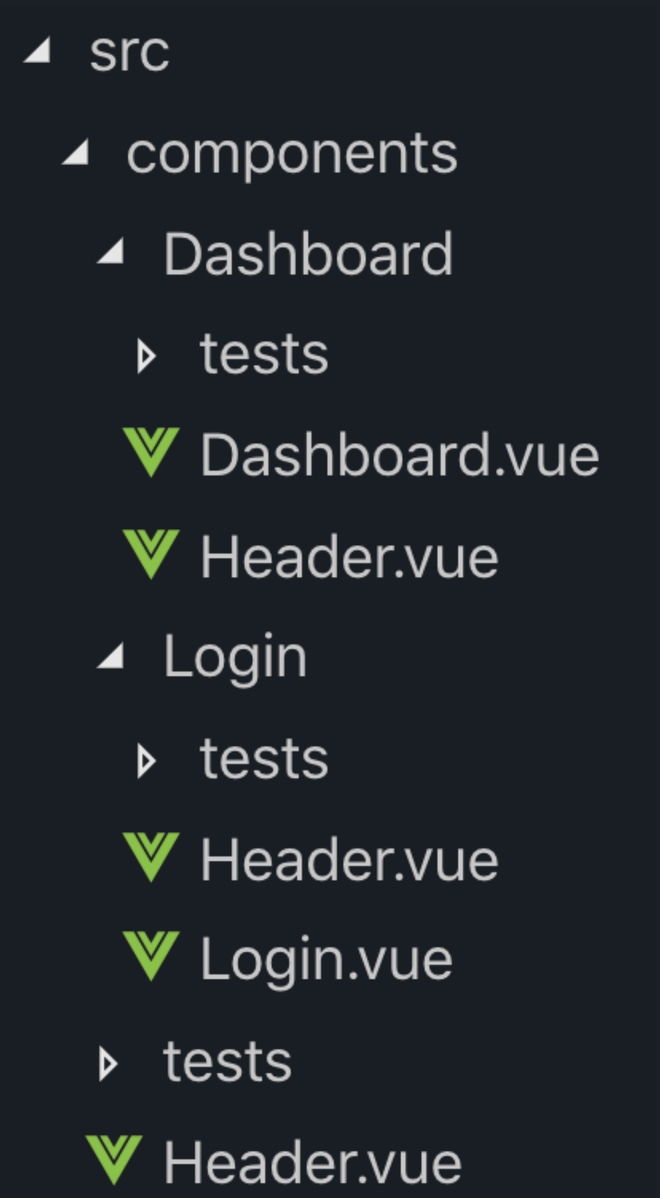
```
<template>
<!-- ... -->
</template>


<style>
/* ... */
</style>
```

```
<script>
export default {
// ...
}
</script>


<template>
<!-- ... -->
</template>


<style>
/* ... */
</style>
```

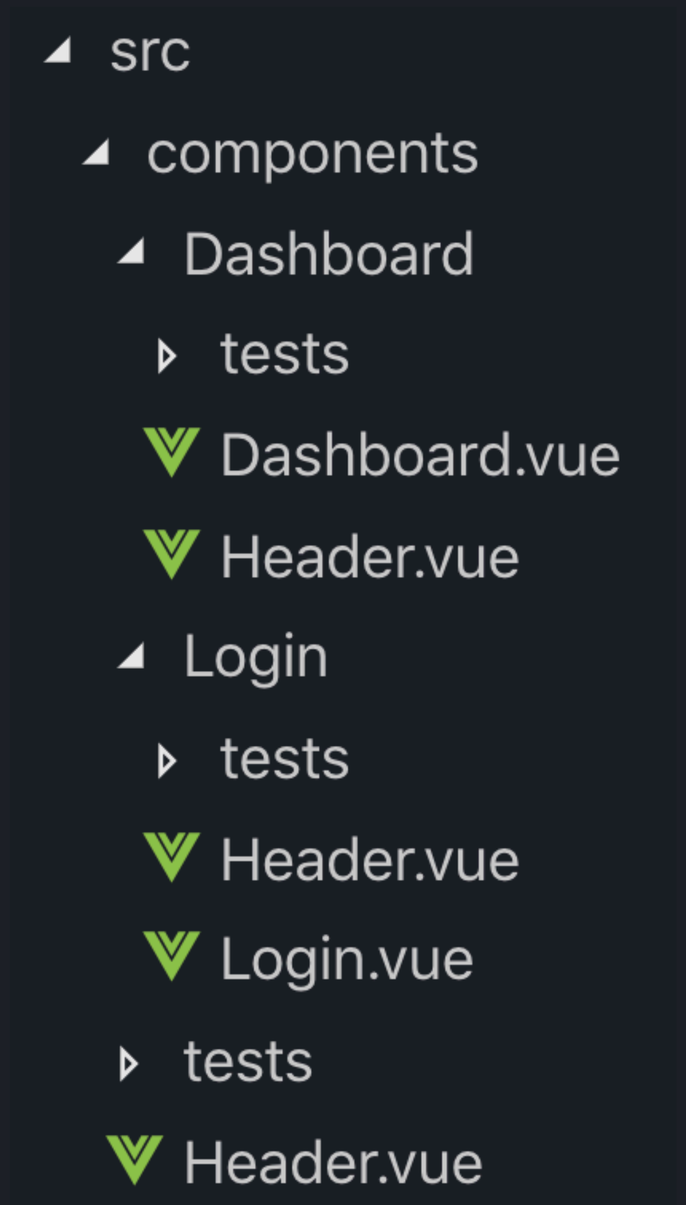
PRO TIP

How to Organize Component Files

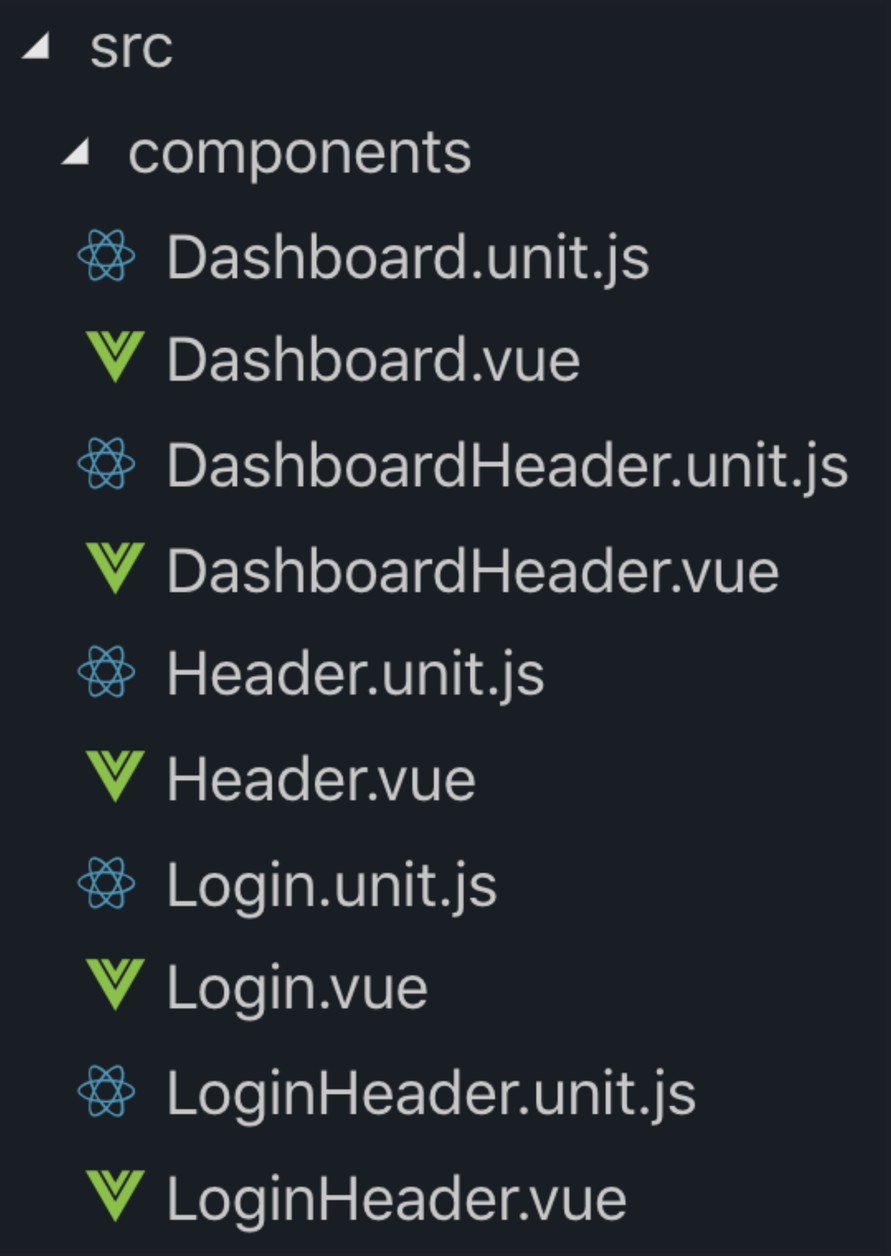
Nested Structure



Flat Structure


src
components
Dashboard
tests
Dashboard.vue
Header.vue
Login
tests
Header.vue
Login.vue
tests
Header.vue

VS


src
components
Dashboard.unit.js
Dashboard.vue
DashboardHeader.unit.js
DashboardHeader.vue
Header.unit.js
Header.vue
Login.unit.js
Login.vue
LoginHeader.unit.js
LoginHeader.vue

Component Organization

Flat makes refactoring easier

No need to update imports if components move

Flat makes finding files easier

Folders often leads to lazily named files

because they don’t have to be unique

PRO TIP

Register base components globally

Instead of every component having:

```
import BaseButton from ‘./components/BaseButton.vue'
import BaseIcon from ‘./components/BaseIcon.vue'
import BaseInput from ‘./components/BaseInput.vue'
```

Use this epic script by Chris Fritz:

https://vuejs.org/v2/guide/components-registration.html#Automatic-Global-Registration-of-Base-Components

```
import Vue from 'vue'
import upperFirst from 'lodash/upperFirst'
import camelCase from 'lodash/camelCase'

const requireComponent = require.context(
  './components',
  false,
  /Base[A-Z]\w+\.(vue|js)$/
)

requireComponent.keys().forEach(fileName => {
  const componentConfig = requireComponent(fileName)

  const componentName = upperFirst(
    camelCase(
      fileName
        .split('/')
        .pop()
        .replace(/\.\w+$/, '')
    )
  )

  // Register component globally
  Vue.component(
    componentName,
```

TECHNIQUE

Props

NavItem.vue

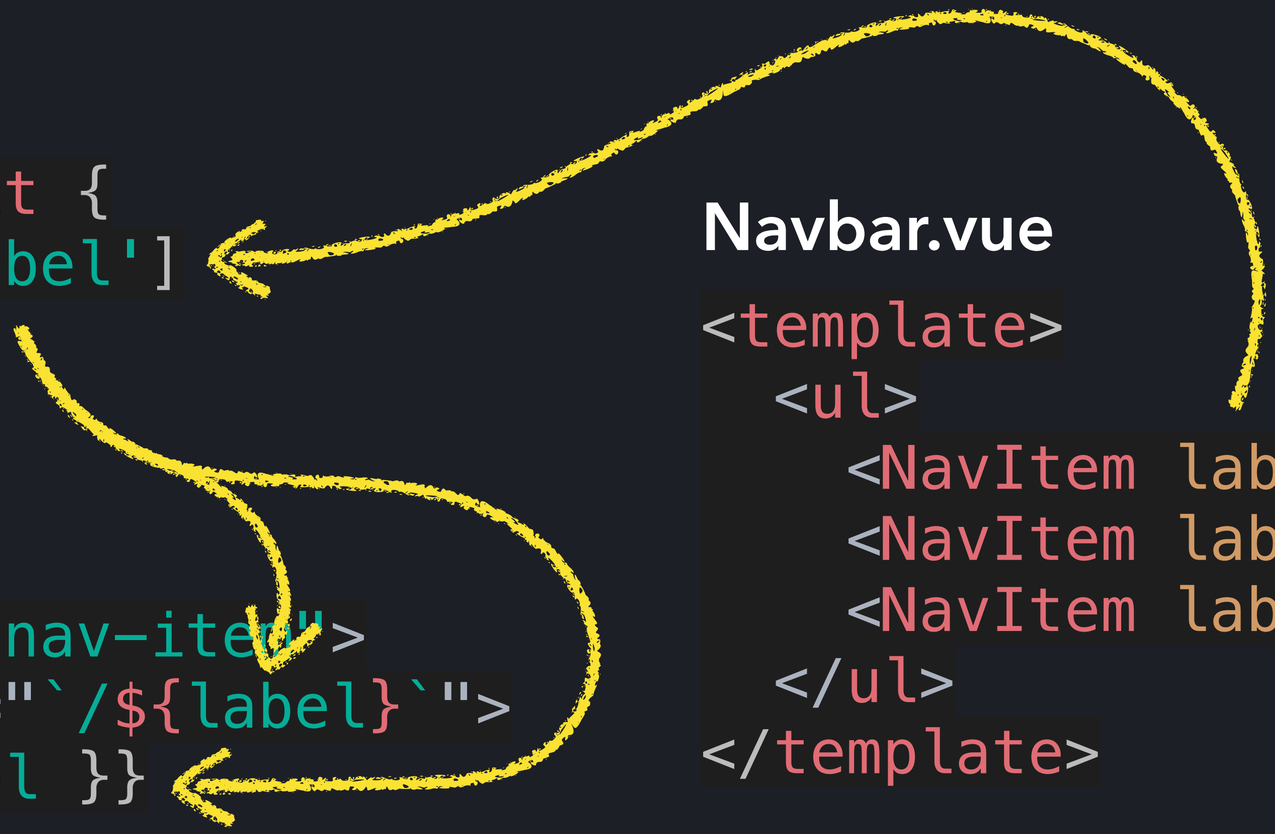
```
<script>
export default {
  props: ['label']
}
</script>

<template>
  <li class="nav-item">
    <a :href="`/${label}`">
      {{ label }}
    </a>
  </li>
</template>
```

NavItem.vue

```
<script>
export default {
  props: ['label']
}
</script>

<template>
  <li class="nav-item">
    <a :href="`/${label}`">
      {{ label }}
    </a>
  </li>
</template>
```

Navbar.vue

```
<template>
  <ul>
    <NavItem label="Home" />
    <NavItem label="About" />
    <NavItem label="Contact" />
  </ul>
</template>
```



NavItem.vue

```
<script>
export default {
  props: ['label']
}
</script>

<template>
  <li class="nav-item">
    <a :href="`/${label}`">
      {{ label }}
    </a>
  </li>
</template>
```

Navbar.vue

```
<template>
  <ul>
    <NavItem label="Home" />
    <NavItem label="About" />
    <NavItem label="Contact" />
  </ul>
</template>
```

NavItem.vue

```
<script>
export default {
  props: {
    label: {
      type: String,
      required: true,
      default: 'Home'
    }
  }
}
</script>
```

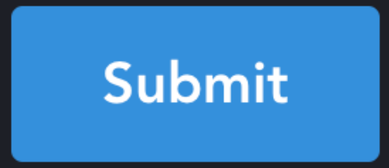
NavItem.vue

```
<script>
export default {
  props: {
    label: {
      type: String,
      default: 'Home'
    }
  }
}
</script>
```

Let’s do a
Coding Experiment

Task 1

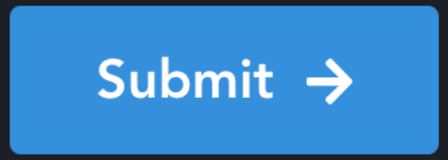
Create a button component that can display text specified in the parent component



Task 2

Allow the button to display an icon of choice

on the right side of the text

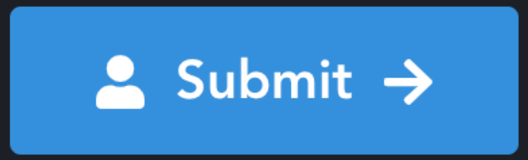



```
<AppIcon icon="arrow-right" class="ml-3"/>
```

This is the code responsible for displaying an arrow.

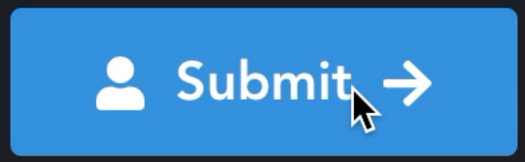
Task 3

Make it possible to have icons on either side or even both sides



Task 4

Make it possible to replace the content with a loading spinner

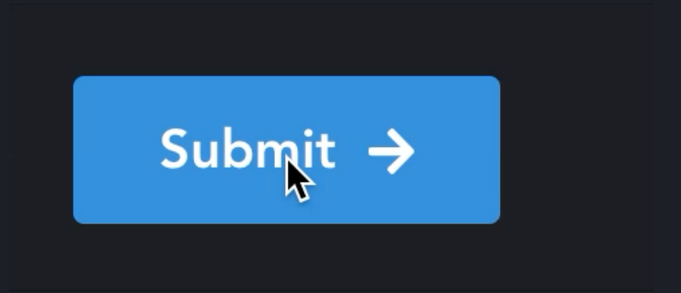



```
<PulseLoader color="#fff" size="12px"/>
```

This is the code responsible for displaying a spinner.

Task 5

Make it possible to replace an icon with a loading spinner



Possible solution

```
<template>
  <button type=“button" class=“nice-button“>
    {{ text }}
  </button>
</template>

<script>
export default {
  props: [‘text']
}
</script>
```

```
<template>
  <button type=“button" class=“nice-button“>
    <PulseLoader v-if="isLoading" color="#fff" size="12px">
    <template v-else>
      <template v-if="iconLeftName">
        <PulseLoader v-if="isLoadingLeft" color="#fff" size="6px">
        <AppIcon v-else :icon=“iconLeftName"/>
      </template>
      {{ text }}
      <template v-if="iconRightName">
        <PulseLoader v-if="isLoadingRight" color="#fff" size="6px">
        <AppIcon v-else :icon=“iconRightName”/>
      </template>
    </template>
  </button>
</template>

<script>
export default {
  props: ['text', 'iconLeftName', 'iconRightName', ‘isLoading’,
'isLoadingLeft', 'isLoadingRight']
}
</script>
```


OMG
PROPS EVERYWHERE!

```
<template>
  <button type=“button" class=“nice-button“>
    <PulseLoader v-if="isLoading" color="#fff" size="12px">
    <template v-else>
      <template v-if="iconLeftName">
        <PulseLoader v-if="isLoadingLeft" color="#fff" size="6px">
        <AppIcon v-else :icon=“iconLeftName"/>
      </template>
      {{ text }}
      <template v-if="iconRightName">
        <PulseLoader v-if="isLoadingRight" color="#fff" size="6px">
        <AppIcon v-else :icon=“iconRightName"/>
      </template>
    </template>
  </button>
</template>

<script>
export default {
  props: ['text', 'iconLeftName', 'iconRightName', ‘isLoading’,
'isLoadingLeft', 'isLoadingRight']
}
</script>
```

Let’s call it the props-based solution

props-based solution

Is it wrong?

props-based solution

Is it wrong?

No.

It does the job.

props-based solution

Is it good, then?

props-based solution

Is it good, then?

Not exactly.

props-based solution

Problems

props-based solution

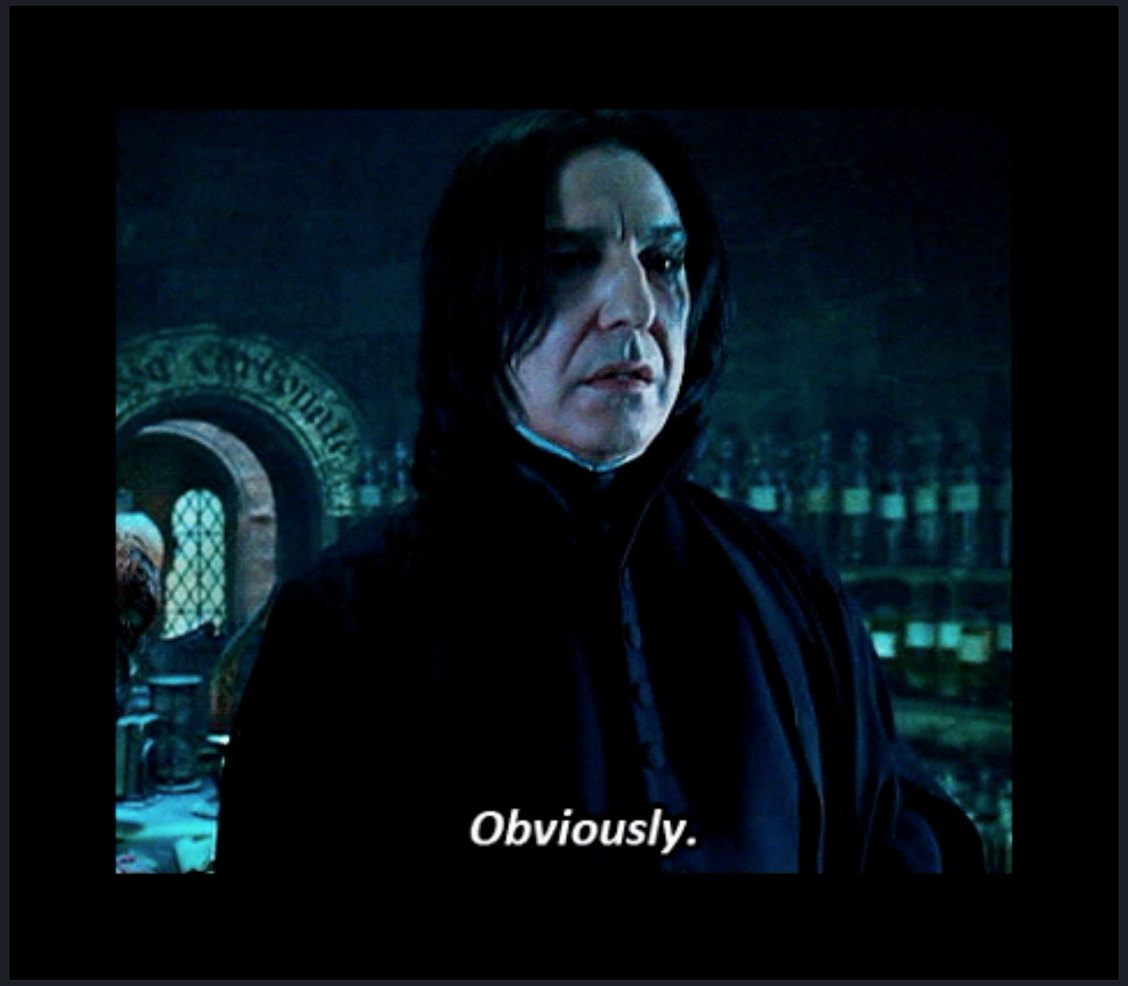
Problems

- New requirements increase complexity
- Multiple responsibilities
- Lots of conditionals in the template
- Low flexibility
- Hard to maintain

Is it good, then?

Not exactly.

Is there a better another alternative?



Is there a better another alternative?

Recommended solution

```
<template>
  <button type="button" class="nice-button">
    <slot />
  </button>
</template>
```

Lunch Break

Be back at 1:05PM!

BEST PRACTICES

Components (Part 2)

TECHNIQUE

<Component :is=“name”>

```
<template>
  <div>
    <Component :is=“clockType" v-bind=“clockProps”/>
  </div>
</template>

<script>
export default {
  components: { DigitalClock },
  computed: {
    clockType () {
      if (this.selectedClock === 'analog') {
         this.clockProps = {
            …analogProps
          }
        return () => import(`./components/${this.compName}`)
      } else {
        return 'DigitalClock'
      }
    }
  }
  // ...
}
</script>
```

<Component :is>

Becomes the component

specified by the **:is** prop.

Pros

- Extremely powerful and flexible
- Easy to use
- Can accept props
- Can accept asynchronous components
- Can change into different components
- You can make a router-view out of it

Cons

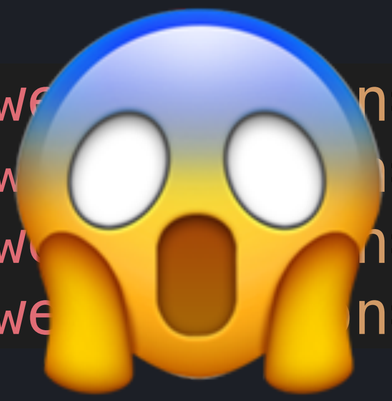
- Got to handle props carefully

DESIGN PATTERN

Vendor Components Wrapper

```
<template>
  <p>
    <FontAwesome icon=“water” />
    <FontAwesome icon="earth" />
    <FontAwesome icon=“fire" />
    <FontAwesome icon="air" />
  </p>
</template>
```

```
<template>
  <p>
    <FontAwesome icon=“water” />
    <FontAwesome icon="earth" />
    <FontAwesome icon=“fire" />
    <FontAwesome icon="air" />
  </p>
</template>
```



```
<template>
  <p>
    <FontAwesome icon=“water” />
    <FontAwesome icon="earth" />
    <FontAwesome icon=“fire" />
    <FontAwesome icon="air" />
  </p>
</template>
```

```
<template>
  <p>
    <FontAwesome icon=“water” />
    <FontAwesome icon="earth" />
    <FontAwesome icon=“fire" />
    <FontAwesome icon="air" />
  </p>
</template>
```

BaseIcon.vue

```
<template>
  <FontAwesomeIcon
    v-if="source === 'font-awesome'"
    :icon="name"
  />
  <span
    v-else
    :class="customIconClass"
  />
</template>
```

```
<template>
  <p>
    <BaseIcon icon=“earth” />
    <BaseIcon icon="fire" />
    <BaseIcon icon="water" />
    <BaseIcon icon=“air" />
  </p>
</template>
```

DESIGN PATTERN

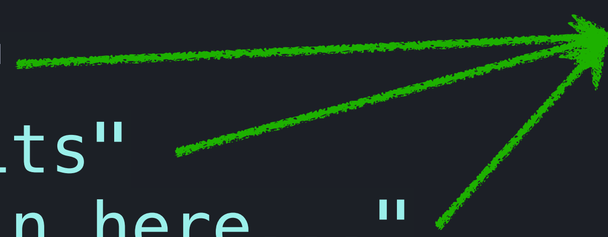
Transparent Components

When passing props, listeners, and attributes...

```
// BaseInput.vue
<template>
  <div>
    <input
      type="text"


    />
  </div>
</template>
```

When passing props, listeners, and attributes...

```
<BaseInput
  @input="filterData"
  label="Filter results"
  placeholder="Type in here..."
/>
```



```
// BaseInput.vue
<template>
  <div>
    <input
      type="text"
    />
  </div>
</template>
```

When passing props, listeners, and attributes...

```
<BaseInput
  @input="filterData"
  label="Filter results"
  placeholder="Type in here..."
/>
```

```
// BaseInput.vue
<template>
  <div>
    <input
      type="text"
    />
  </div>
</template>
```

When passing props, listeners, and attributes...

```
<template>
  <div>
    <input
      type="text"
      v-bind="{ …$attrs, …$props }"
      v-on="$listeners"
    />
  </div>
</template>

<script>
export default {
  inheritAttrs: false,
  // ...
}
</script>
```

Both props and attributes, as well as all listeners will be passed to this element instead.

Prevent Vue from assigning attributes to top-level element

When passing props, listeners, and attributes...

```
<template>
  <div>
    <input
      type="text"
      v-bind="{ …$attrs, …$props }"
      v-on="$listeners"
    />
  </div>
</template>

<script>
export default {
  inheritAttrs: false,
  // ...
}
</script>
```

When passing props, listeners, and attributes in v3...

```vue
<template>
  <div>
    <input type="text" v-bind="$attrs" />
  </div>
</template>


<script>
export default {
  inheritAttrs: false,
  // ...
}
</script>
```

BEST PRACTICES

Vuex

Tips for **using Vuex**

State

Tips for **using Vuex**

What data to put into Vuex?

- Data shared between components that might not be in direct parent-child relation
- Data that you want to keep between router views (for example lists of records fetched from the API)
  - Route params are more important though (as a source of truth)
- Any kind of global state
  - Examples: login status, user information, global notifications
- Anything if you feel it will make managing it simpler

Tips for **using Vuex**

What data **NOT** to put into Vuex?

- User Interface variables
  - Examples: `isDropdownOpen`, `isInputFocused`, `isModalVisible`
- Forms data.
- Validation results.
- Single records from the API
  - Think: `currentlyViewedProduct`

Tips for **using Vuex**

Getters

Tips for **using Vuex**

Do I need to **always** use a **getter** to return a simple fragment of state?

No.

Feel free to access state directly

```
this.$store.state.usersList
```

Use computed properties to return computed state

```
activeUsersList () {
  return this.$store.state.usersList.filter(
    user => user.isActive
  )
}
```

Tips for **using Vuex**

If you need to share derived Vuex state between components, make it a getter.

You should weigh the trade-offs and make decisions that fit the development needs of your app.

Tips for **using Vuex**

Use **mapState** and **mapGetters** helpers

```
computed: {
  ...mapState({
    userName: state => state.user.name
  }),
  ...mapGetters([
    'activeUsersList'
  ]),
  // local computed properties
}
```

Tips for **using Vuex**

Mutations & Actions

Tips for **using Vuex**

Use modules

https://vuex.vuejs.org/guide/modules.html

Questions?

Practice

In the boilerplate

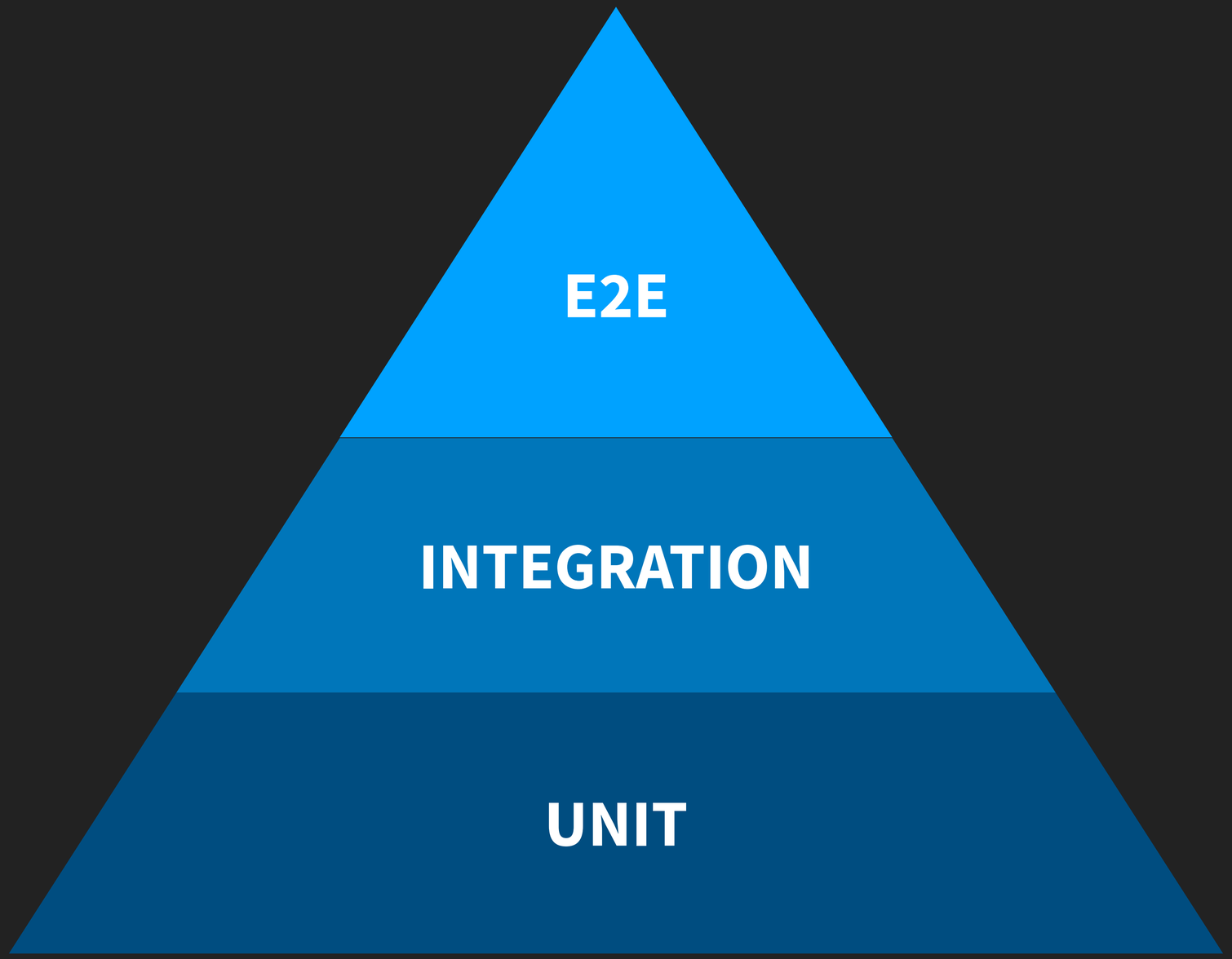
- Create a new **shoppingList** module to store shopping list items.
- Create helpers for the **shoppingList** module in **src/state/helpers.js**.
- In **src/router/views/home.vue**, import some state helpers to display, add, and remove items from the shopping list.

In your app

- Add Vuex to your app with a single **shoppingList** module, as described to the left.
- Automatically register all your Vuex modules.
- Refactor a module to be namespaced, updating all references to that module.
- Automatically dispatch an **init** action for your modules.

BEST PRACTICES

Testing


E2E
INTEGRATION
UNIT

UNIT

Jest

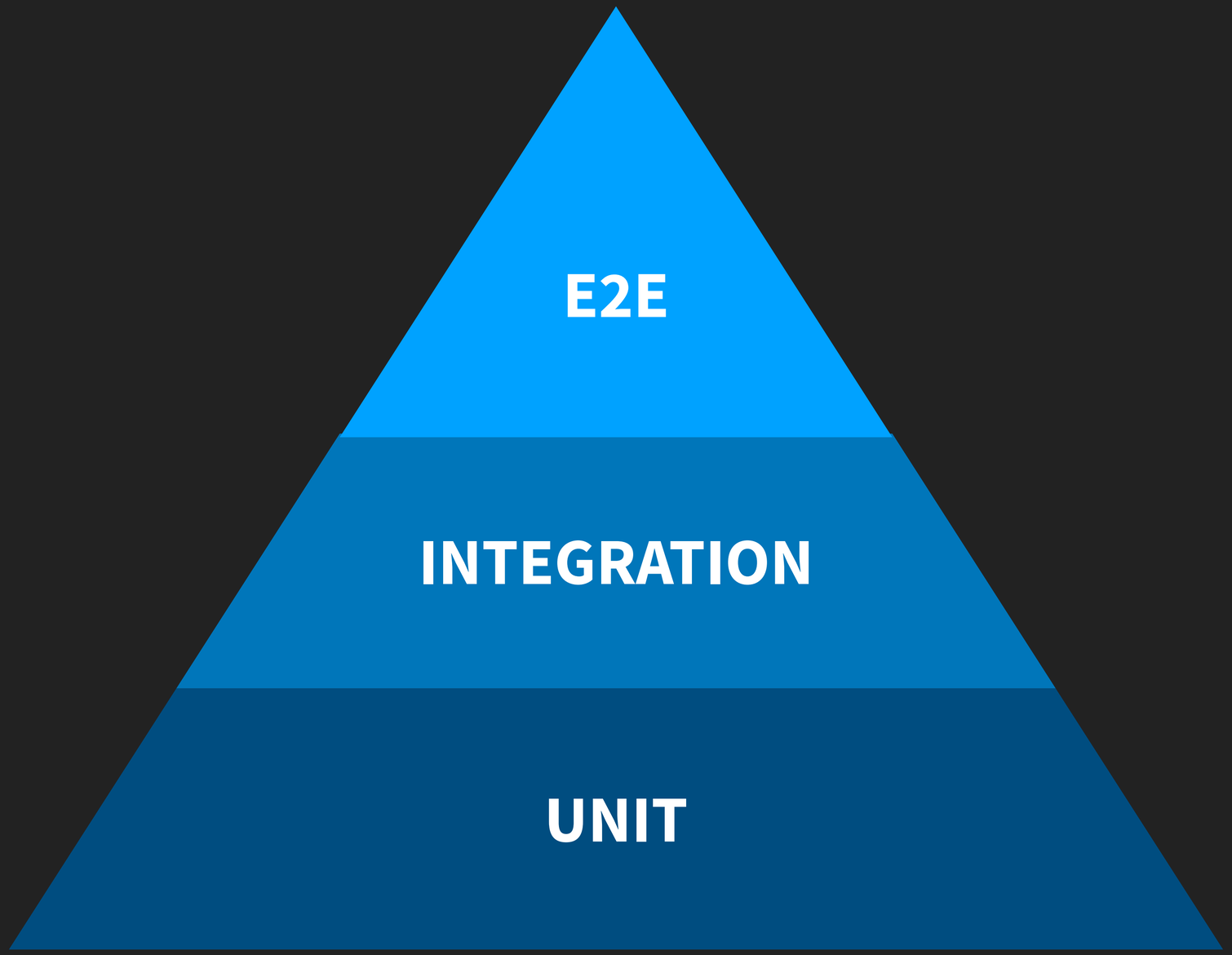
UNIT

What about Vue Test Utils?

COMPONENT

Testing Library

https://testing-library.com/

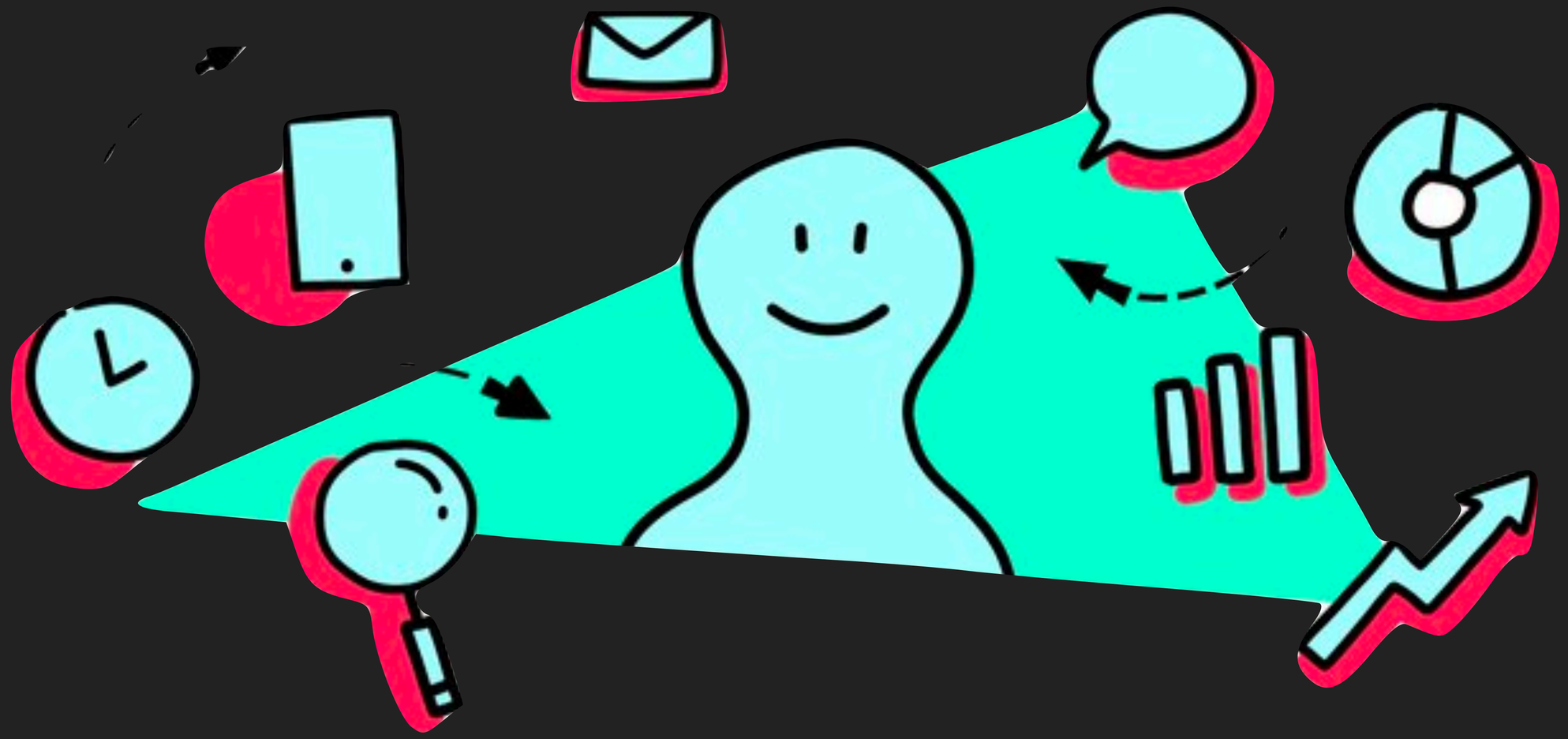

E2E
INTEGRATION
UNIT

/

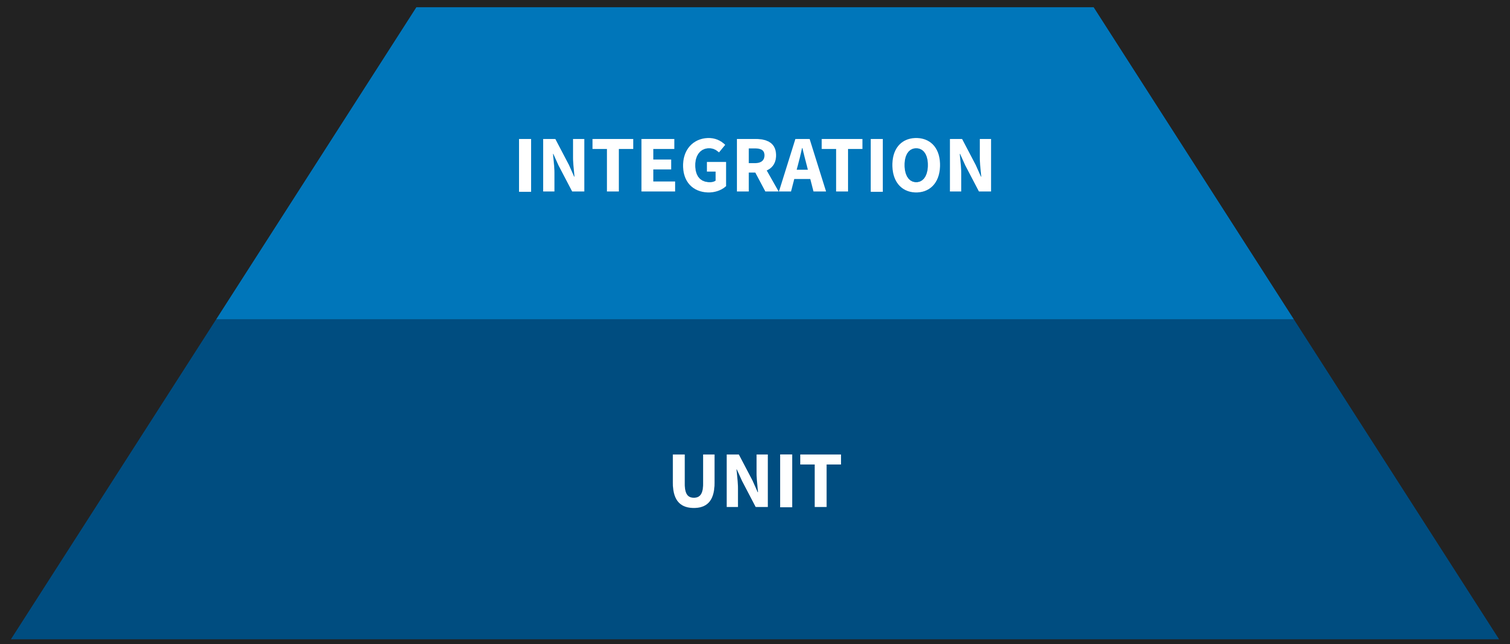
98.36% Statements 1859/1890 **93.92%** Branches 726/773 **100%** Functions 266/266 **99.73%** Lines 1816/1821 **37 statements, 21 branches** Ignored

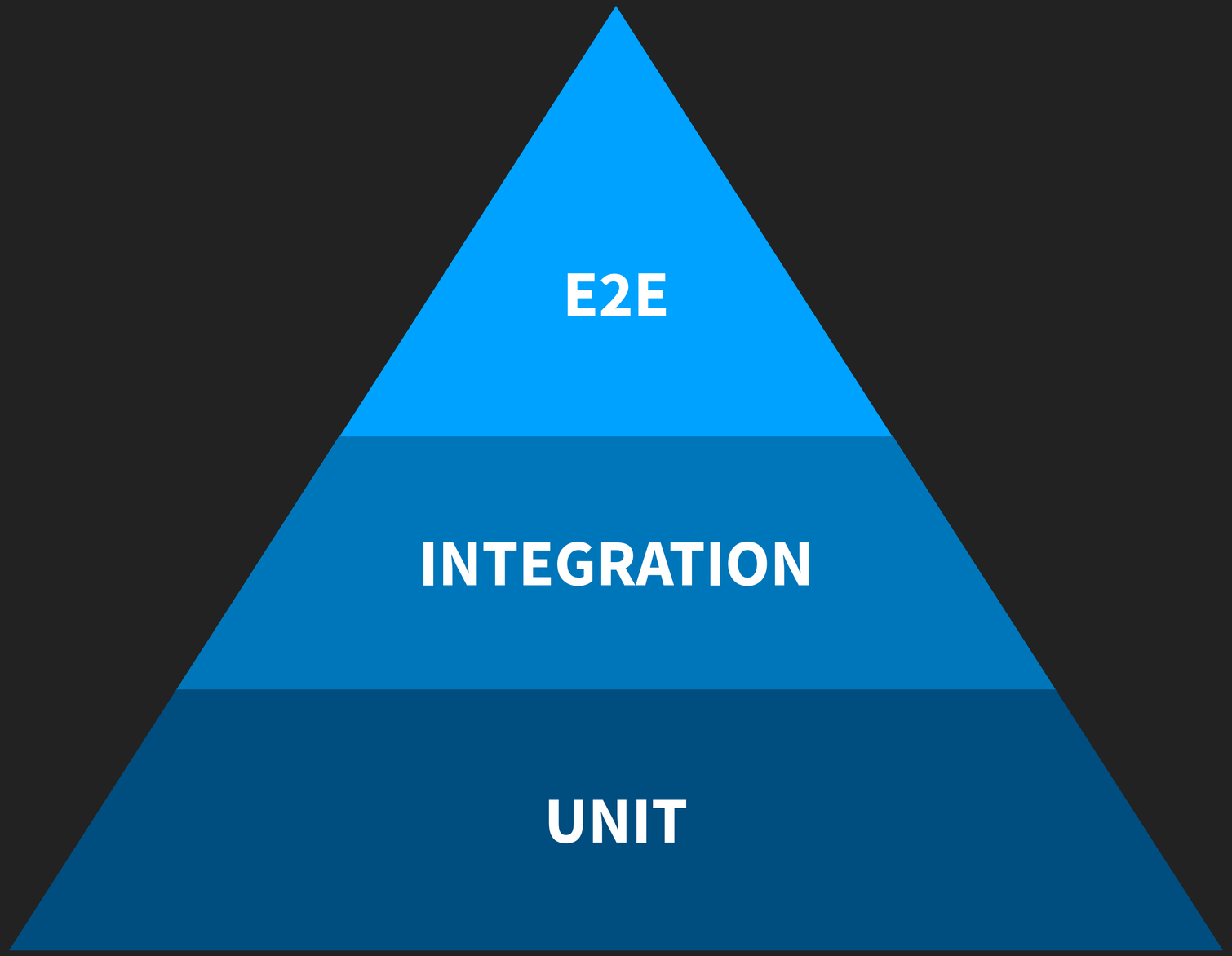
File		Statements		Branches		Functions		Lines	
express/		100%	1/1	100%	0/0	100%	0/0	100%	1/1
express/examples/auth/		93.75%	75/80	80.77%	21/26	100%	17/17	100%	71/71
express/examples/content-negotiation/		100%	32/32	100%	2/2	100%	13/13	100%	32/32
express/examples/cookie-sessions/		100%	12/12	100%	4/4	100%	1/1	100%	12/12
express/examples/cookies/		95.83%	/24	87.5%	7/8	100%	3/3	100%	22/22
express/examples/downloads/				87.5%	7/8	100%	3/3	100%	15/15
express/examples/ejs/				100%	2/2	100%	1/1	100%	11/11
express/examples/error-pages/				83.33%	10/12	100%	6/6	100%	34/34
express/examples/error/				66.67%	4/6	100%	4/4	100%	18/18
express/examples/markdown/		95.		66.67%	4/6	100%	5/5	100%	20/20
express/examples/multi-router/		100%	9/9	100%	2/2	100%	1/1	100%	9/9
express/examples/multi-router/controllers/		100%	14/14	100%	0/0	100%	4/4	100%	14/14
express/examples/mvc/		95.45%	42/44	80%	8/10	100%	4/4	100%	42/42
express/examples/mvc/controllers/main/		100%	2/2	100%	0/0	100%	1/1	100%	2/2
express/examples/mvc/controllers/pet/		94.12%	16/17	50%	1/2	100%	4/4	100%	16/16
express/examples/mvc/controllers/user-pet/		92.86%	13/14	50%	1/2	100%	1/1	100%	13/13
express/examples/mvc/controllers/user/		100%	21/21	100%	4/4	100%	6/6	100%	19/19
express/examples/mvc/lib/		97.96%	48/49	80%	20/25	100%	2/2	100%	46/46

All of these tools are great,
but we have forgotten the most critical metric
that determines whether we are successful.



The User!


INTEGRATION
UNIT

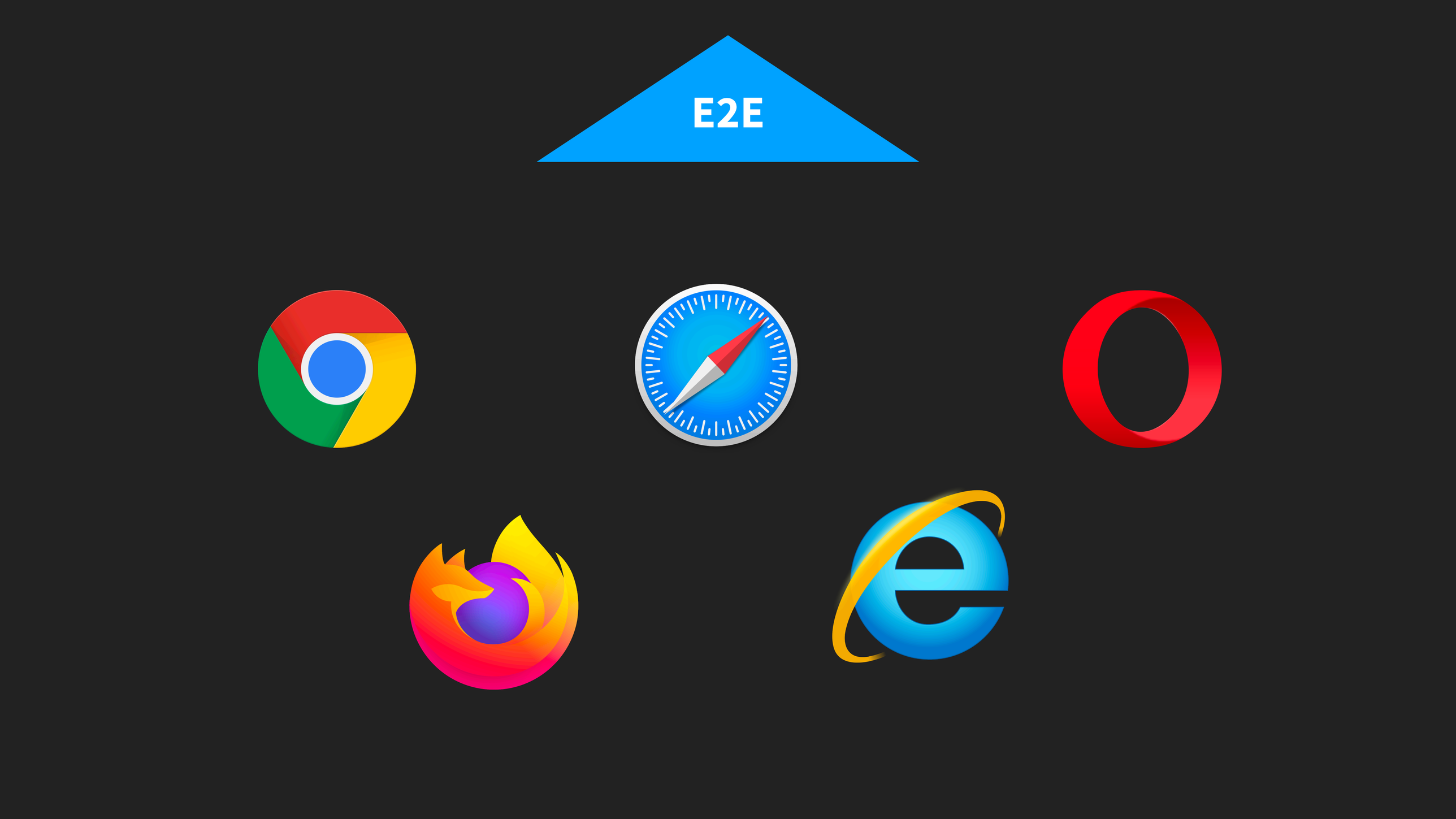

E2E
INTEGRATION
UNIT


E2E

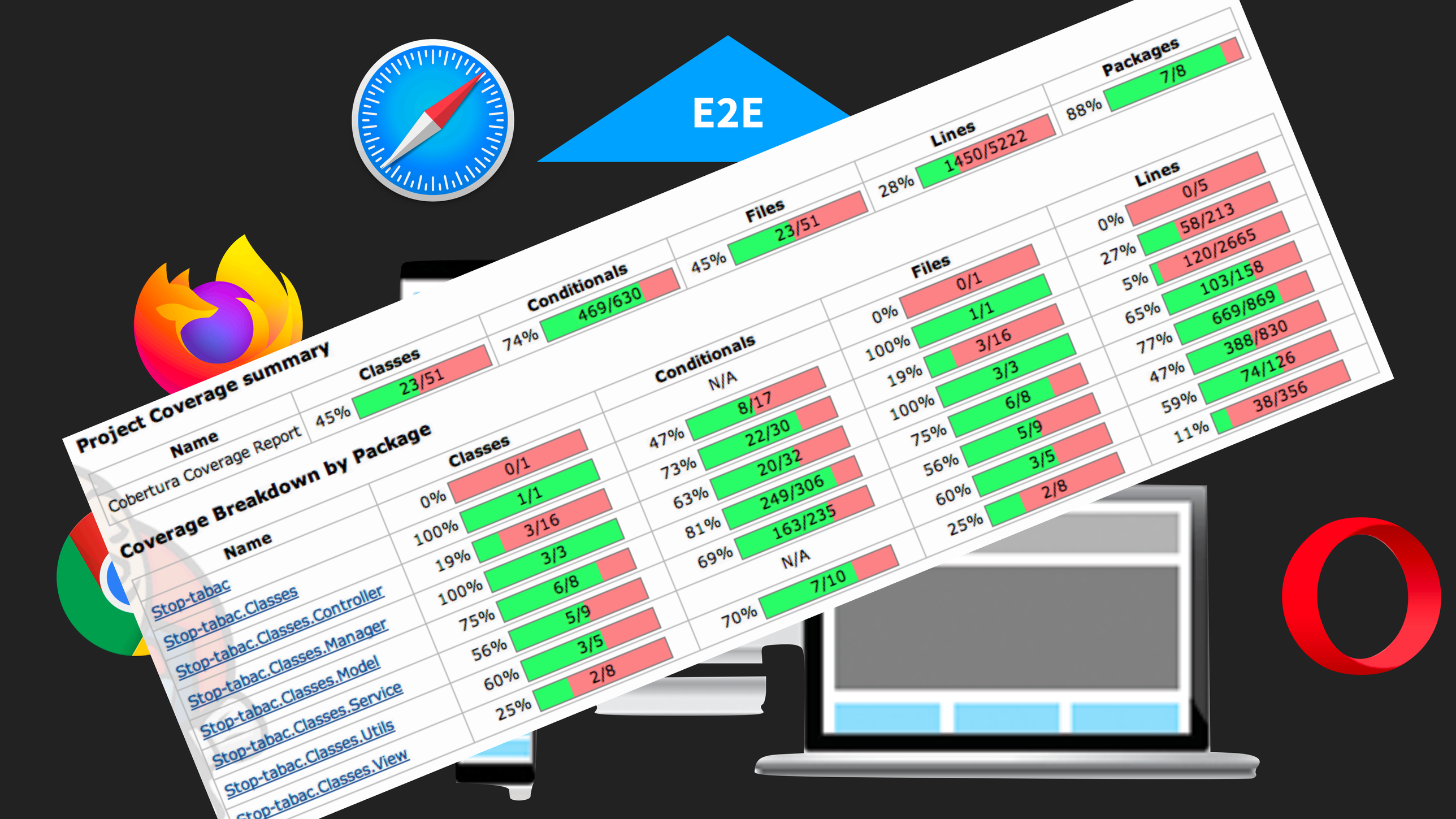
Does everything work?

Why do people hate e2e tests?

- Take a long time to run
- "Flakey" (i.e., unreliable)


E2E


E2E


E2E
Project Coverage summary
Name
Classes
Conditionals
Files
Lines
Packages
Cobertura Coverage Report
45% 23/51
74% 469/630
45% 23/51
28% 1450/5222
88% 7/8
Coverage Breakdown by Package
Name
Classes
Conditionals
Files
Lines
Stop-tabac
0% 0/1
N/A
0% 0/1
0% 0/5
Stop-tabac.Classes
100% 1/1
47% 8/17
100% 1/1
27% 58/213
Stop-tabac.Classes.Controller
19% 3/16
73% 22/30
19% 3/16
5% 120/2665
Stop-tabac.Classes.Manager
100% 3/3
63% 20/32
100% 3/3
65% 103/158
Stop-tabac.Classes.Model
75% 6/8
81% 249/306
75% 6/8
77% 669/869
Stop-tabac.Classes.Service
56% 5/9
69% 163/235
56% 5/9
47% 388/830
Stop-tabac.Classes.Utils
60% 3/5
N/A
60% 3/5
59% 74/126
Stop-tabac.Classes.View
25% 2/8
70% 7/10
25% 2/8
11% 38/356


E2E
Project Coverage summary
Name
Classes
Conditionals
Files
Lines
Packages
Cobertura Coverage Report
45% 23/51
74% 469/
28% 1450/5222
88% 7/8
Coverage Breakdown by Package
Name
Classes
Files
Lines
Stop-tabac
0% 0/1
0% 0/1
0% 0/5
Stop-tabac.Classes
100% 1/1
00% 1/1
27% 58/213
Stop-tabac.Classes.Controller
19% 3/16
9% 3/16
5% 120/2665
Stop-tabac.Classes.Manager
100% 3/3
00% 3/3
65% 103/158
Stop-tabac.Classes.Model
75% 6/8
75% 6/8
77% 669/869
Stop-tabac.Classes.Service
56% 5/9
56% 5/9
47% 388/830
Stop-tabac.Classes.Utils
60% 3/5
60% 3/5
59% 74/126
Stop-tabac.Classes.View
25% 2/8
25% 2/8
11% 38/356
70%

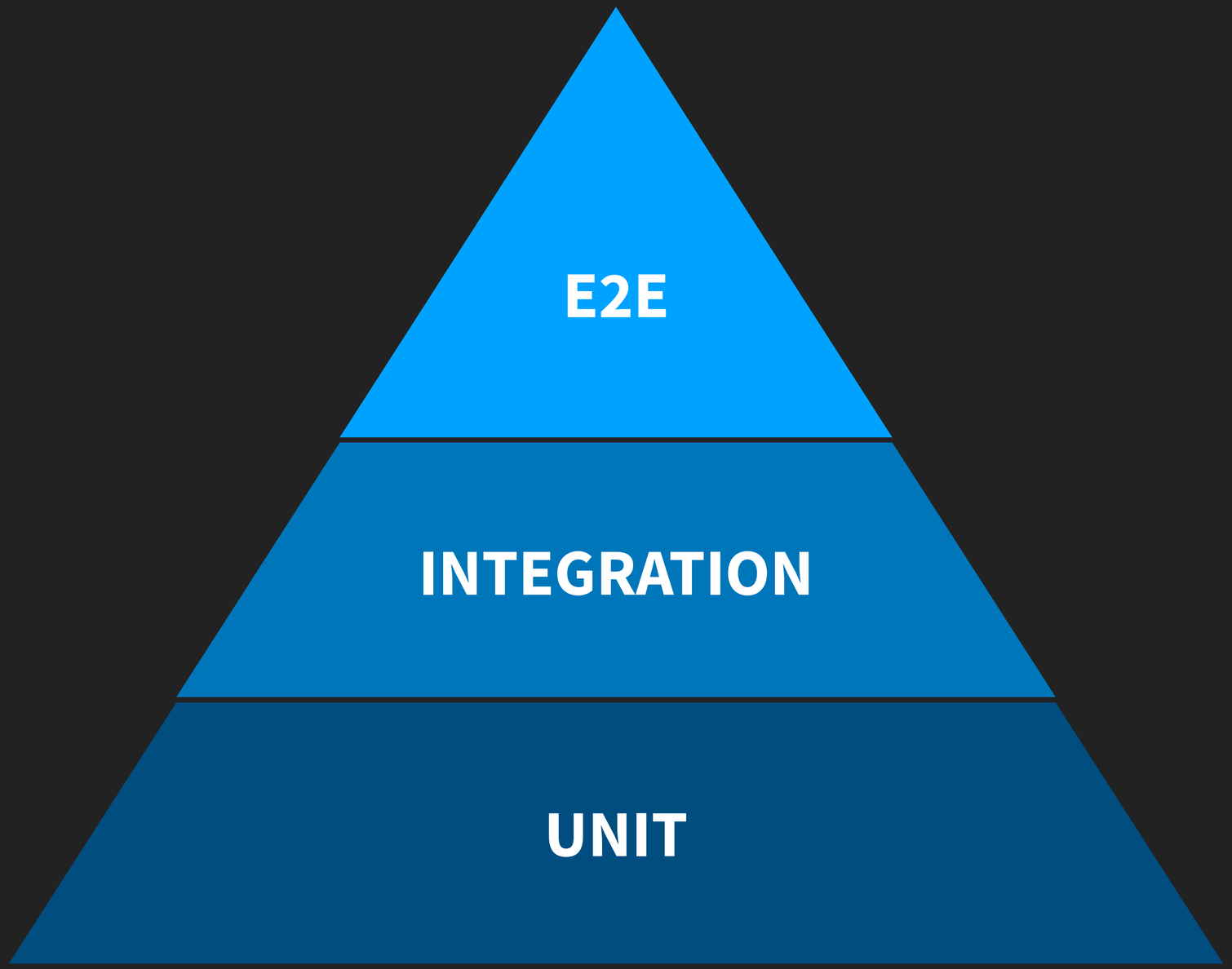
The Pareto Principle

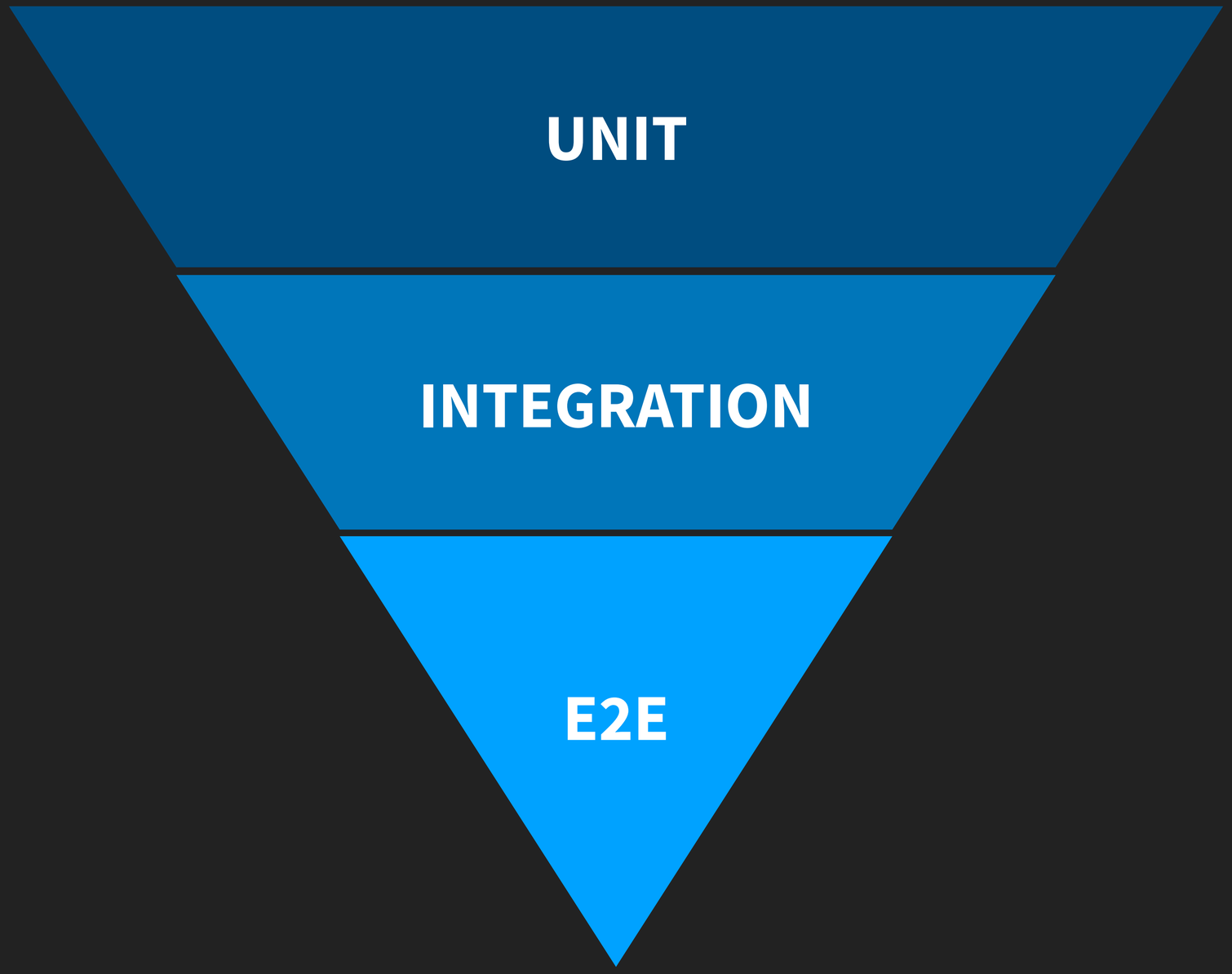
The Pareto Principle

The 80-20 Rule


20% EFFORT


20% EFFORT
80%
RESULTS


E2E
INTEGRATION
UNIT


UNIT
INTEGRATION
E2E

Most teams avoid E2E tests because...

- Take a long time to run
- "Flakey" (i.e., unreliable)

As a result, there are often no E2E tests at all...

If you can only have two tests for your application...

If you can only have two tests for your application...

#1. Can the user login?

#2. Can the user pay us?

Cypress

TestCafe

Nightwatch

Cypress

TestCafe

NEW FEATURE

Composition API

Vue 3.0 is here!

Live Coding

Counter App

Let’s Code Together

In the PPW repository

- Create a **TodoList** component

- It should render:
  - A text input for user’s to place text
  - All of the current tasks
  - Active Tasks
  - Completed Tasks

NEW FEATURE

Composition API

Vue 3.0 is here!

Also available in 2.0!

https://github.com/vuejs/composition-api

Questions?

COFFEE BREAK

Be back at 3:00PM!

ACCESSIBILLITY

Maria Lamardo

Final Thoughts

🎊 CONGRATULATIONS! 🎊

Questions?

Thanks everyone!

twitter.com/bencodezen

youtube.com/bencodezen

EXTRAS

BEST PRACTICES

Routing

Routes

mapping paths to view components

View components

definitions for page-level components

Layout components

markup shared between pages
(header, navbar, sidebar, etc)

View components

Can use in-component route guards

beforeRouteEnter, beforeRouteUpdate, beforeRouteLeave

(often better to use route-level guards instead)

Can access **$route** or accept props from params

(other components can also access **$route**, but usually shouldn't)

```
import store from '@state/store'

export default [
  {
    path: '/',
    name: 'home',
    component: require('@views/home').default,
  },
  {
    path: '/login',
    name: 'login',
    component: require('@views/login').default,
    beforeEnter(routeTo, routeFrom, next) {
      // If the user is already logged in
      if (store.getters['auth/loggedIn']) {
        // Redirect to the home page instead
        next({ name: 'home' })
      } else {
        // Continue to the login page
        next()
      }
    },
  },
  // ...
}
```

Access the store

Inline **require**

In-route guards keep the views simple

Meta info for views

Page **<title>**, **<meta name="description">**, etc

vue-meta

```
export default {
  metaInfo() {
    return {
      title: this.user.name,
      meta: [
        {
          name: 'description',
          content: `The user profile for ${this.user.name}.`,
        },
      ],
    }
  },
  // ...
}
```

Handling 404s

src/router/views/404.vue

```
<script>
import Layout from '@layouts/main'

export default {
  components: { Layout },
  props: {
    resource: {
      type: String,
      default: '',
    },
  },
}
</script>

<template>
  <Layout>
    <h1>
      404
      <span v-if="resource">
        {{ resource }}
      </span>
      Not Found
    </h1>
  </Layout>
</template>
```

src/router/routes.js#76

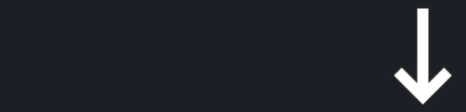
```
{
  path: '/404',
  name: '404',
  component: require('@views/404').default,
  props: true,
},
{
  path: '*',
  redirect: '404',
},
```

src/router/routes.js#54

```
next({ name: '404', params: { resource: 'User' } })
```

Lazy-loaded routes

```
{
  path: '/admin',
  name: 'admin-dashboard',
  component: require('@views/admin').default
}
```



```
{
  path: '/admin',
  name: 'admin-dashboard',
  component: () => import('@views/admin')
}
```

BEST PRACTICES

Components (Part 3)

PROBLEM

How to share the same functionality across different components?

TECHNIQUE

Mixins

https://vuejs.org/v2/guide/mixins.html

A mixin

```
const myMixin = {
  data () {
    return {
      foo: 'bar'
    }
  }
}

export default {
  mixins: [myMixin],
  // component code
}
```

Mixin as a function

```
const myMixin = (count) => ({
  data () {
    return {
      currentCount: count
    }
  }
})

export default {
  mixins: [myMixin(10)],
  // component code
}
```

Pros

- Relatively easy to use

Cons

- Possible properties name clashes.
- Can’t share template fragments
- Gets harder to track where things are coming from once there are more mixins

Should you never use mixins?

PROBLEM

How to pass data and methods
deep into the component tree?

TECHNIQUE

Provide/Inject

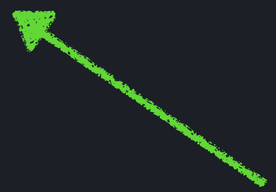
https://vuejs.org/v2/api/#provide-inject

Provide/Inject

```
export default {
  provide () {
    return {
      width: this.width, // will stay reactive
      key: ‘name’, // won’t be reactive
      fetchMore: this.fetchMore // methods can be passed
    }
  },
  data() {
    return {
      width: null,
    }
  },
  methods: {
    fetchMore () {
      // ...
    }
  }
}
```

Provide/Inject

```
export default {
  inject: ['width', 'key', ‘fetchMore'],
  props: {
    optionKey: {
      type: String,
      default () {
        return this.key
      }
    }
  }
}
```



Injected values can be used as

default props and data values

Pros

- Easy sharing data and methods with descendants
- Helps avoiding unnecessary props
- Components can choose which properties to inject
- Can be used to provide default props and data values

Cons

- Besides observable objects defined in data, other properties are not reactive
  - Example: computed properties won’t update
- Pretty clumsy usage, due to some properties staying reactive, where other don’t
- Requires complicated setup to make other properties reactive
- Better suited for plugins and component libraries rather than regular applications

Provide and inject are primarily provided for advanced plugin / component library use cases. It is NOT recommended to use them in generic application code.

PROBLEM

What if you only want to expose and/or manage data and methods, but no user interface?

“Provider” components

“Provider” components

```
<ApolloQuery
  :query="require('../graphql/HelloWorld.gql')"
  :variables="{ name }"
>
  <template v-slot="{ result: { loading, error, data } }">
    <!-- Loading -->
    <div v-if="loading" class="loading apollo">Loading...</div>
    <!-- Error -->
    <div v-else-if="error" class="error apollo">An error occured</div>
    <!-- Result -->
    <div v-else-if="data" class="result apollo">{{ data.hello }}</div>
```

From Vue Apollo by @akryum

“Provider” components

```
<ApolloQuery
  :query="require('../graphql/HelloWorld.gql')"
  :variables="{ name }"
>
  <template v-slot="{ result: { loading, error, data } }">
    <!-- Loading -->
    <div v-if="loading" class="loading apollo">Loading...</div>
    <!-- Error -->
    <div v-else-if="error" class="error apollo">An error occured</div>
    <!-- Result -->
    <div v-else-if="data" class="result apollo">{{ data.hello }}</div>
```

From Vue Apollo by @akryum

"Provider" components

```
<ApolloQuery
  :query="require('../graphql/HelloWorld.gql')"
  :variables="{ name }"
>
  <template v-slot="{ result: { loading, error, data } }">
    <!-- Loading -->
    <div v-if="loading" class="loading apollo">Loading...</div>
    <!-- Error -->
    <div v-else-if="error" class="error apollo">An error occured</div>
    <!-- Result -->
    <div v-else-if="data" class="result apollo">{{ data.hello }}</div>
```

From Vue Apollo by @akryum

“Provider” components

```
<ApolloQuery
  :query="require('../graphql/HelloWorld.gql')"
  :variables="{ name }"
>
  <template v-slot="{ result: { loading, error, data } }">
    <!-- Loading -->
    <div v-if="loading" class="loading apollo">Loading...</div>
    <!-- Error -->
    <div v-else-if="error" class="error apollo">An error occured</div>
    <!-- Result -->
    <div v-else-if="data" class="result apollo">{{ data.hello }}</div>
```

From Vue Apollo by @akryum