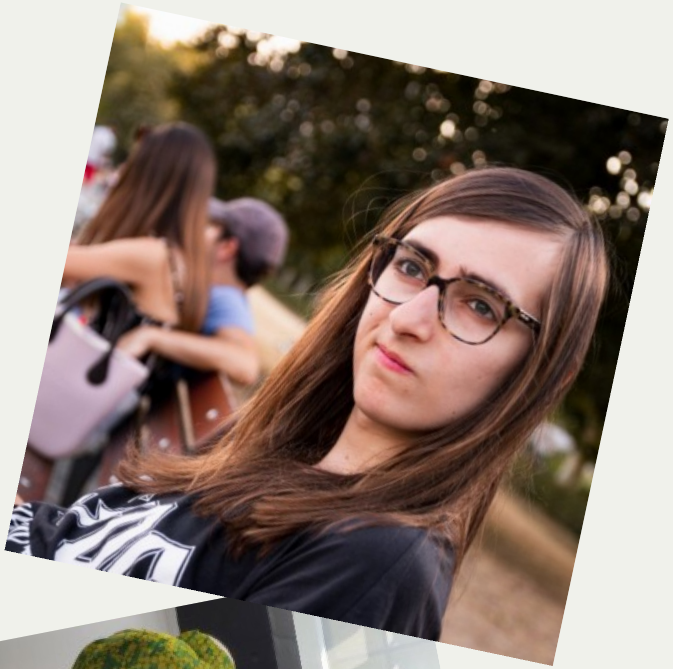
Mi componente, mis normas

@jnroji

FRONT FEST

@EliRP95



Elizabeth Rodriguez

Frontend at Sngular

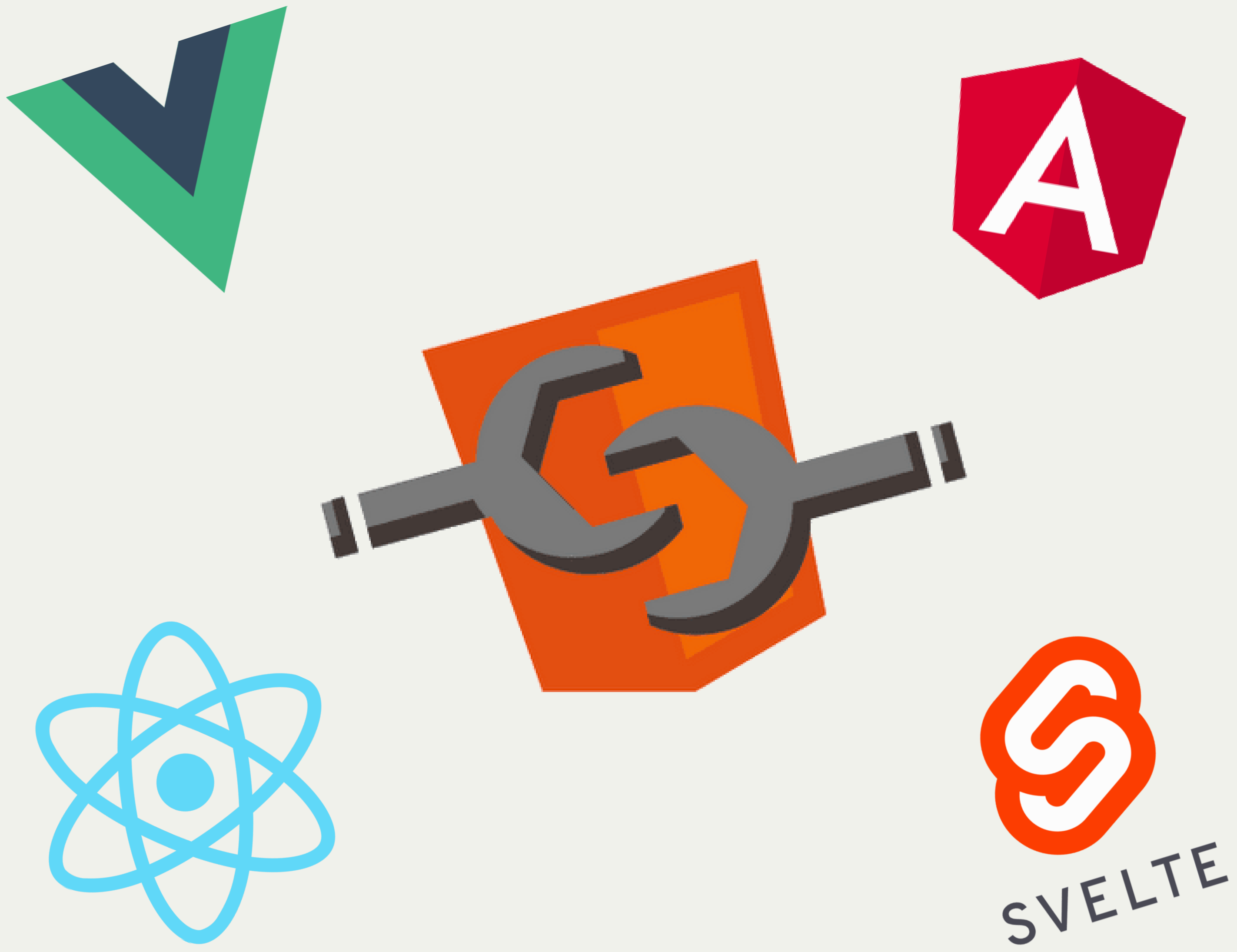
@EliRP95

Jon Rojí

Frontend at Sngular

@jnroji

FRONT FEST



FRONT FEST

Speaker notes

Hoy por hoy, casi todos los frontends nos hemos pasado a los componentes. Ya sean vue, angular, react, nativos u otros.

“ Why do I need a component system?

📚

Speaker notes

When developing a component library, independent to the framework used, it's important to have some things in mind that should be common to all component libraries. The first thing we have to ask ourselves is 'why do I need a component library?'. The most common answer to that is normally reusing code. Most use cases involve a scope of one application, but having reusability in mind from the start makes each component even more valuable the more places we put it in. And in case of expanding the app, diverging into different products or even rewriting the only app we have, having the component library separate to our app will help in the future. We don't want to rewrite the

CONTEXT AGNOSTIC

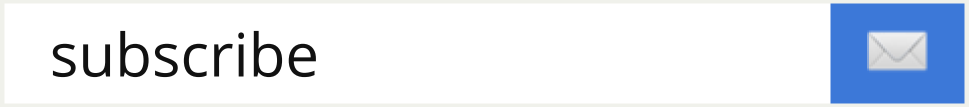
(work in an isolated environment)

FILL A GAP

(use slots and avoid overcharging)

Speaker notes

Know the scope of each component. Components have limited reach, not because they can't go further but because we get into a dangerous territory when we give our components functions that interfere with the outer scope of it. A good example could be a component that receives an array and shows some data, this component can have a delete button on it, and if the user clicks on it, should it delete anything? Probably not, maybe it's better for it to throw an event that the delete button has been clicked and then the container of this component, which has complete access to the source of the data we receive, should delete what's appropiate. If we don't take this into



CONTEXT AGNOSTIC

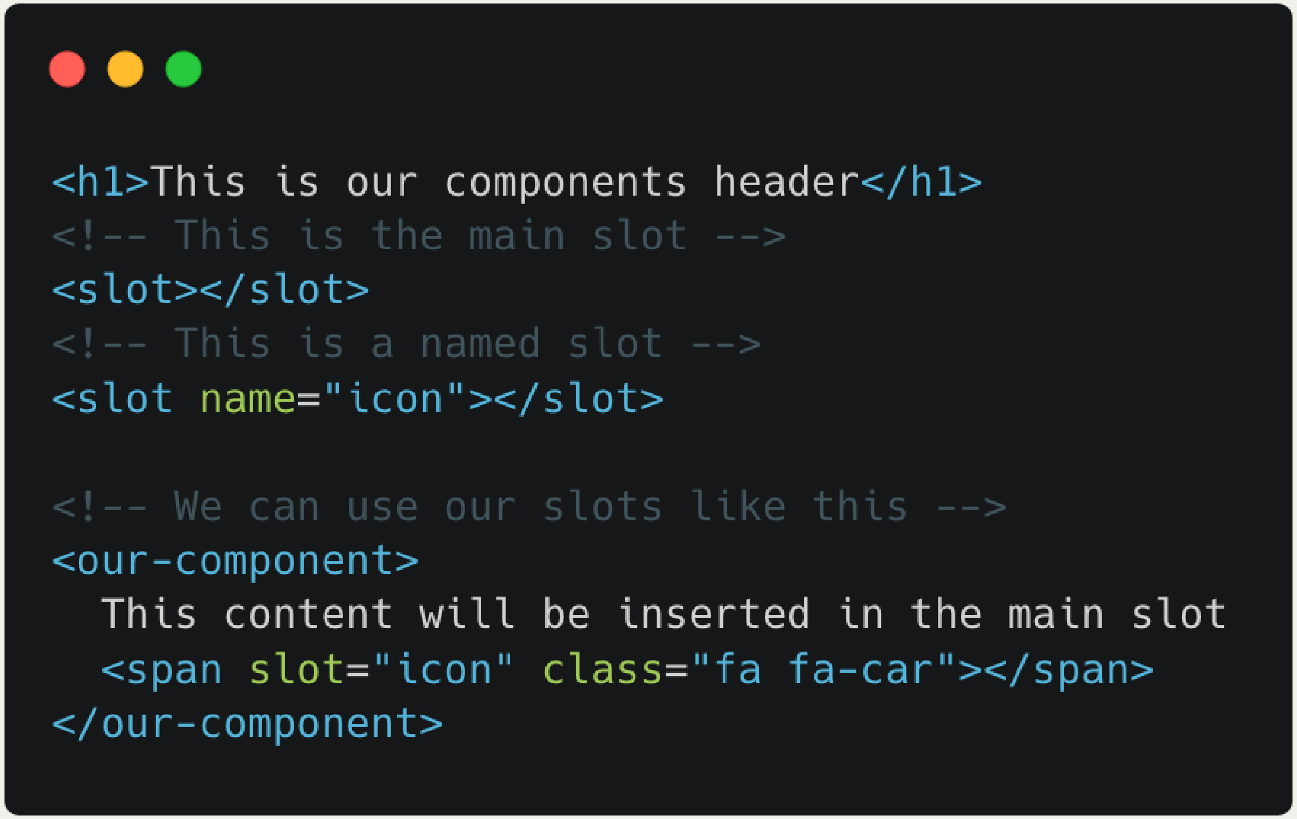
(work in an isolated environment)

FILL A GAP

(use slots and avoid overcharging)

Speaker notes

Know the scope of each component. Components have limited reach, not because they can't go further but because we get into a dangerous territory when we give our components functions that interfere with the outer scope of it. A good example could be a component that receives an array and shows some data, this component can have a delete button on it, and if the user clicks on it, should it delete anything? Probably not, maybe it's better for it to throw an event that the delete button has been clicked and then the container of this component, which has complete access to the source of the data we receive, should delete what's appropiate. If we don't take this into



```html
<h1>This is our components header</h1>
<!-- This is the main slot -->
<slot></slot>
<!-- This is a named slot -->
<slot name="icon"></slot>

<!-- We can use our slots like this -->
<our-component>
  This content will be inserted in the main slot
  <span slot="icon" class="fa fa-car"></span>
</our-component>
```

🔮 Using slots 🔮

DUMMY VS SMART

(don't be a smartass)

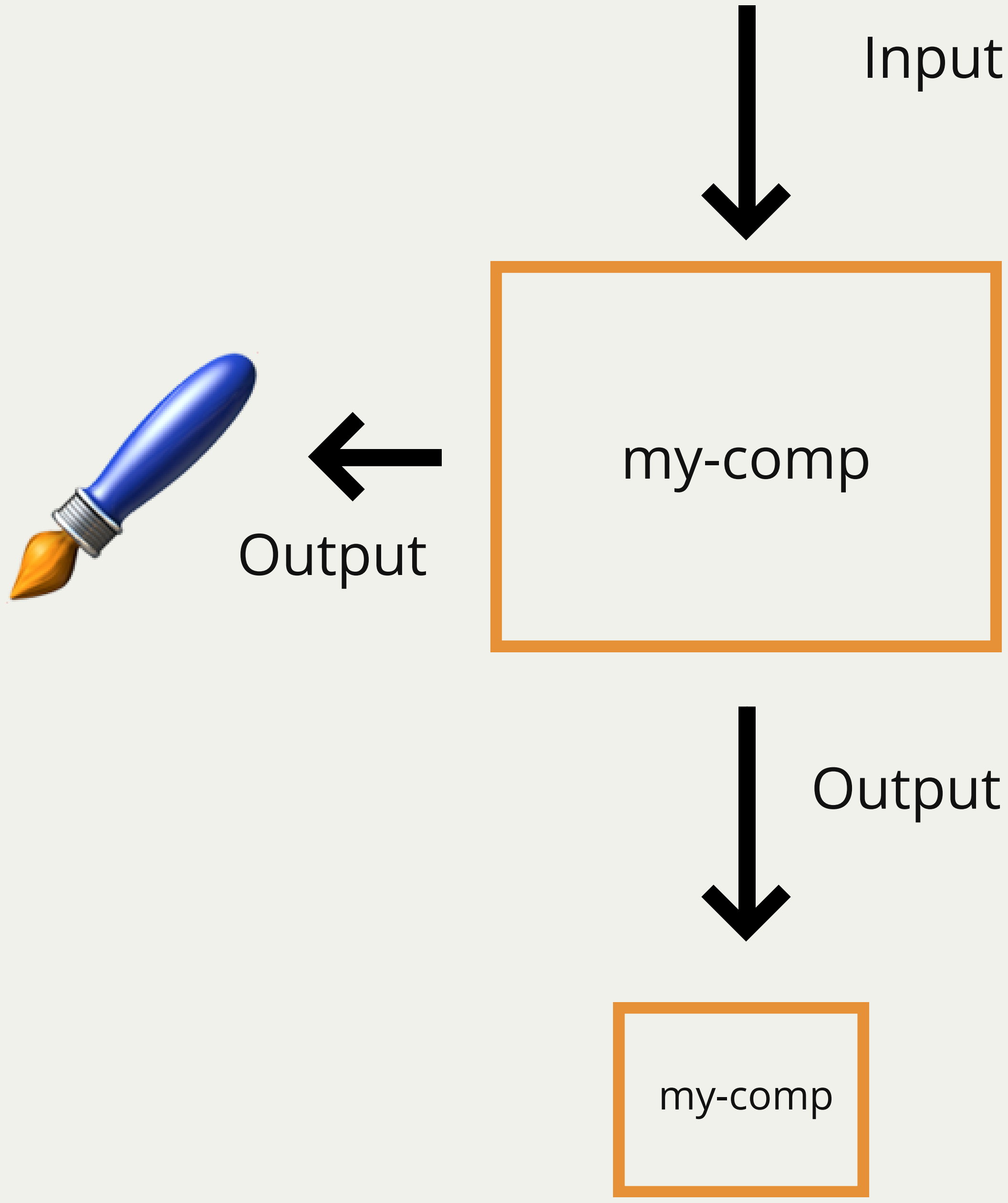
SINGLE RESPONSIBILITY

(even if the component is dumb)

Iterations are a **must**!

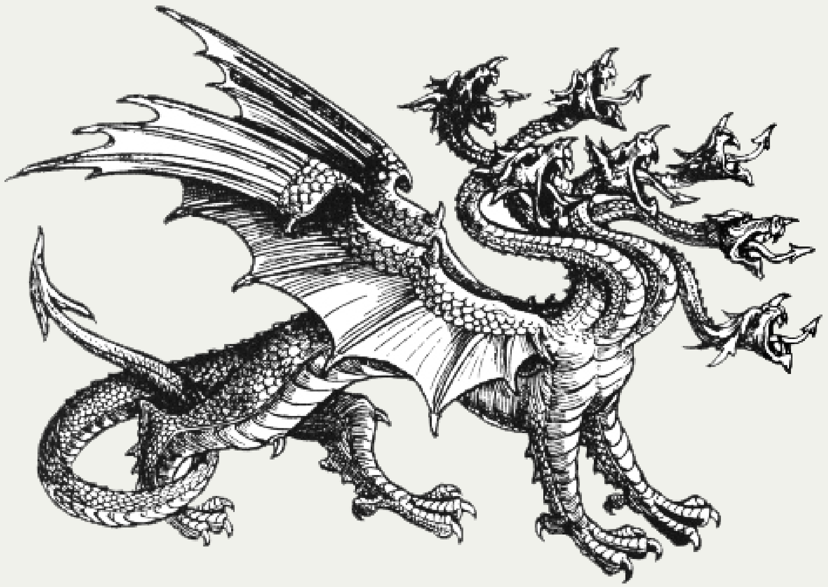
Speaker notes

Another thing we have to have in mind is that components can't be fully done after the first release. Specially when reusing components it's a given that things will need fine tuning, so having iterations planed after releasing the component is a good idea. It's probable that some needs are unintentionally ignored that are later detected in real world use. You can think that this is a problem caused by having the development agnostic to the project but I can assure you that having a component tied in some way to a project will make things much more difficult, in some cases resulting in having to redesign the entire component API.


Input
my-comp
Output
Output
my-comp

Speaker notes

Esta es la estructura habitual de un componente. Hablemos del framework que sea, siempre se presenta un modelo común. Unas propiedades de entrada, normalmente a través de atributos que se exponen en el propio tag, lo que pasa en el propio componente (lógica de negocio y peticiones de datos internas), y dos "Outputs", uno para renderizar el HTML, y aquellos que vayan a otros componentes. Esto "aisla" nuestro planteamiento, y podemos llegar a no ver más allá de donde trabajamos, a aislarnos del contexto, de tal manera que cada desarrollador haga piezas que no tienen nada que ver una con otra, o que nuestro entorno de desarrollo se vuelva un poco complejo.



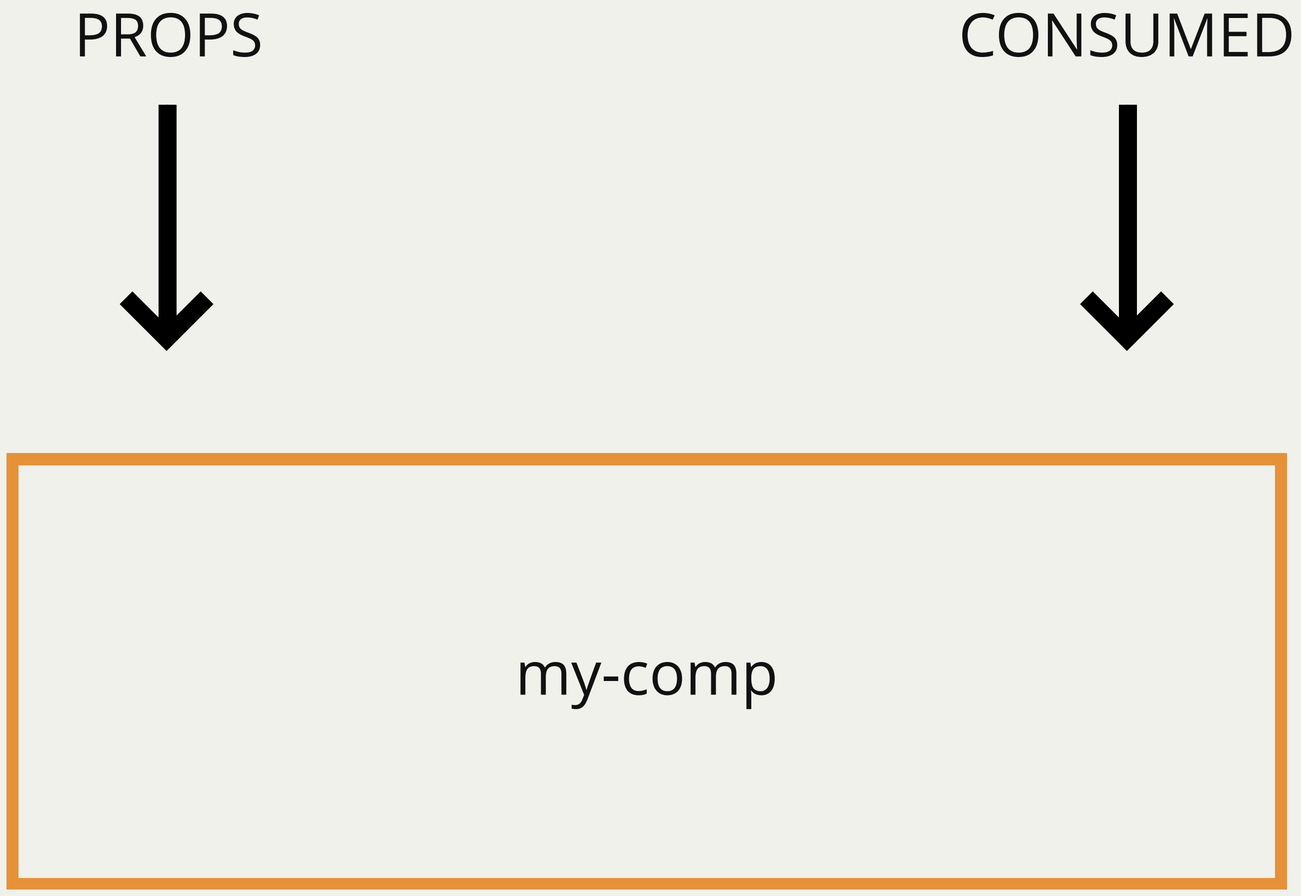
Lerna

Typescript

Speaker notes

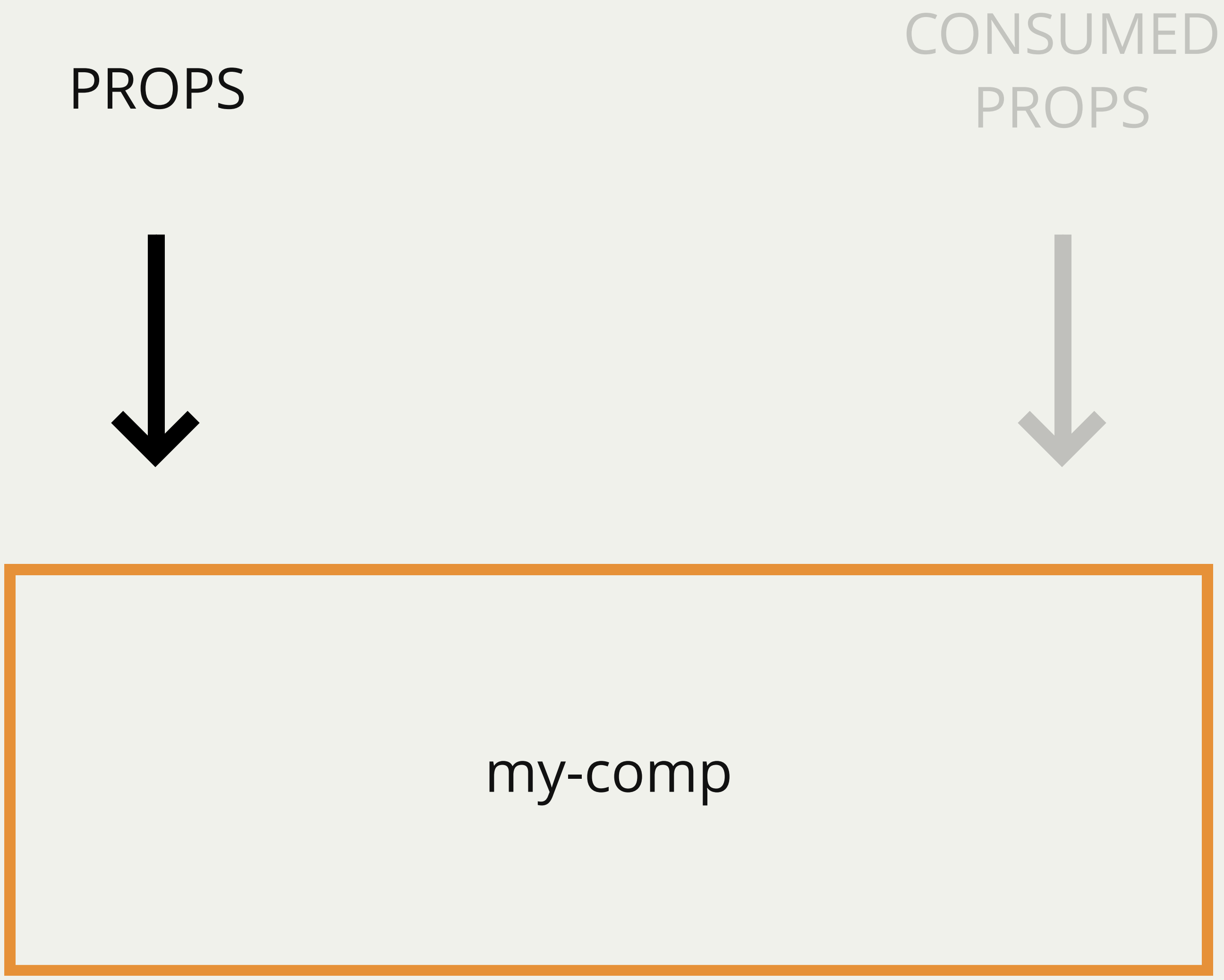
Monorepos can make our lives easier when working with a library of components. I'm going to recommend using Lerna, it's features are endless but it's main function is basicly making monorepos more manageable.

- We can choose to have a component library with it's own versioning and, at the same time have each component with their own versioning too, for example, or you can choose not to.
- Lerna is aware when components depend on each other and will detect when a component versioning change is


PROPS
CONSUMED
my-comp

Speaker notes

La parte más importante de un componente, ya sea listo o tonto, son sus inputs, sus puntos de entrada. Esto marcará cómo se comporta nuestro componente en base a su programación. Si hablamos de los puntos de entrada de datos de nuestra aplicación, vamos a centrarnos en los dos más habituales, las props, o atributos expuestos, y los valores que consumen nuestros componentes a través del uso de APIS de terceros.



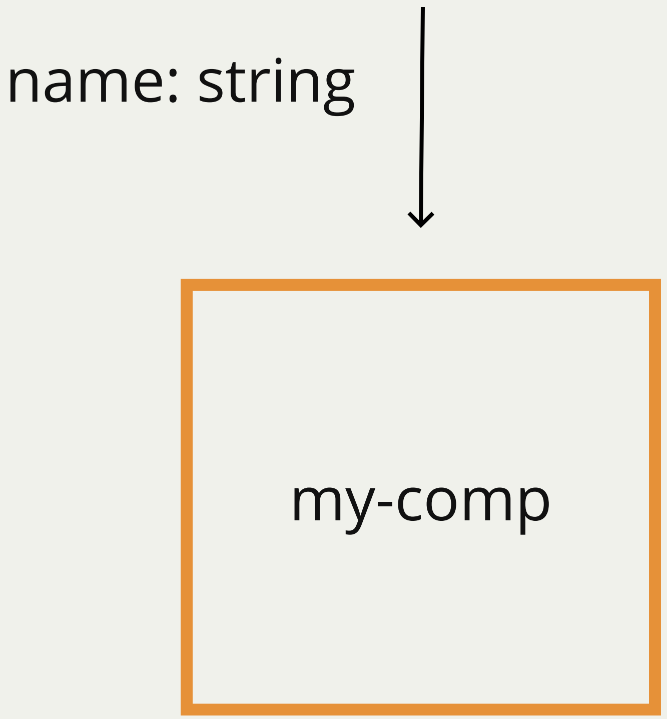
FRONT FEST

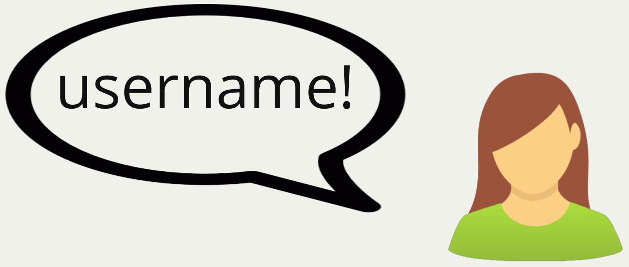
Speaker notes

Vamos a centrarnos primero en las props, la clave para conseguir componentes tontos

PROPS

CONSUMED PROPS



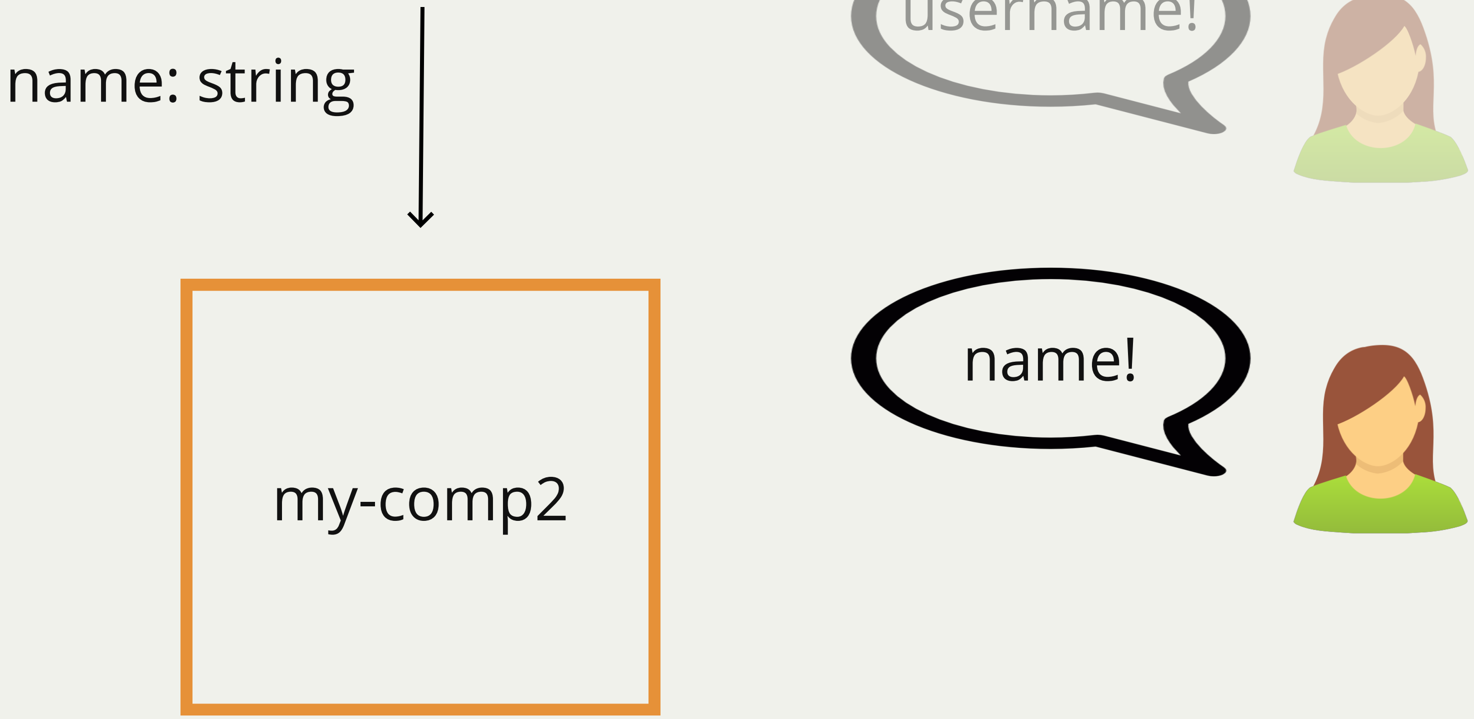


Speaker notes

Pongamos el caso de que existe un componente, con la necesidad de recibir un nombre, con un tipo string, y llega una desarrolladora. Esta decide que su componente, va a tener una propiedad username, que expondrá para el uso de el componente por parte de otros devs. Esta documenta la propiedad y publica el componente. Asunto resuelto

PROPS

CONSUMED PROPS



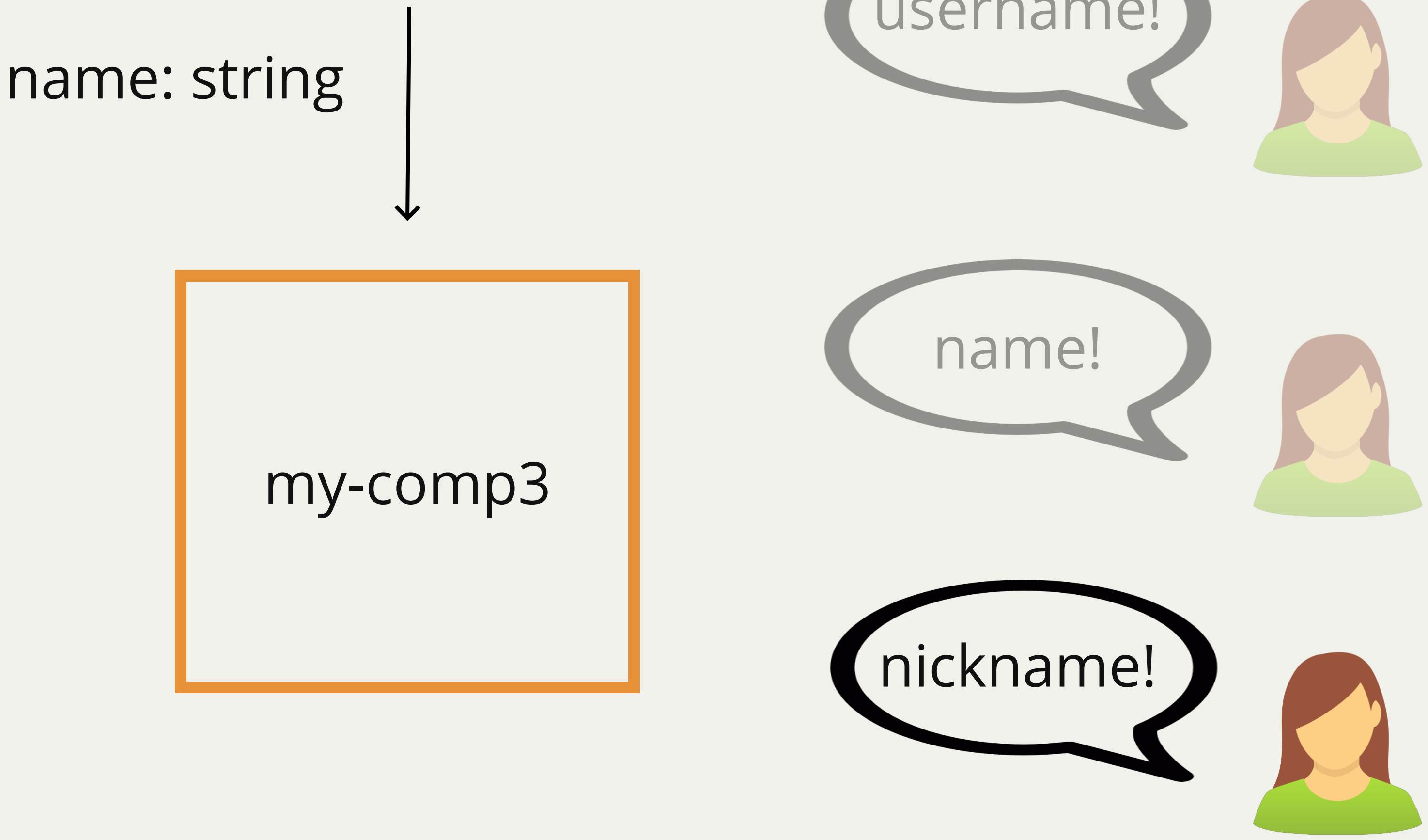
FRONT FEST

Speaker notes

El problema empieza a aparecer, cuando un segundo componente, hecho por otra desarrolladora, tiene la misma necesidad, un nombre en formato string. Esta segunda desarrolladora, que puede trabajar en remoto, o no comentarlo dado que no parece algo tan importante, llama a la propiedad simplemente name.

PROPS

CONSUMED PROPS

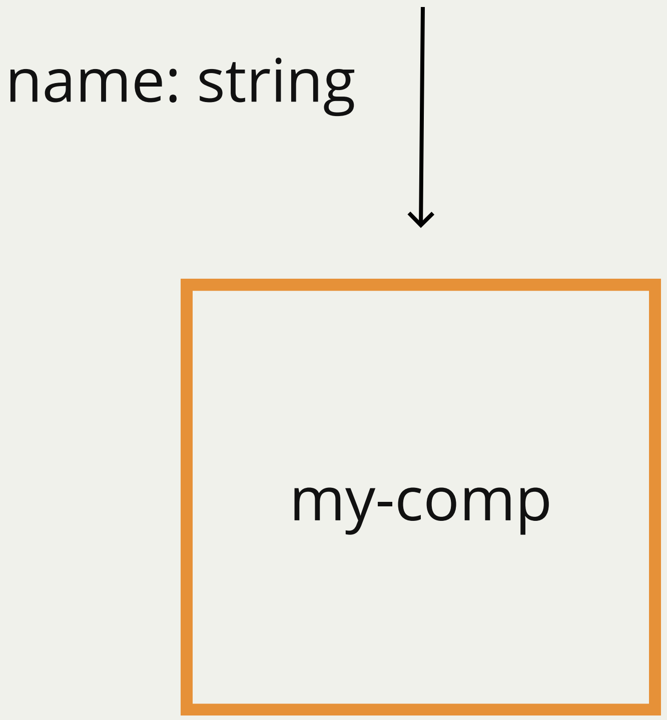


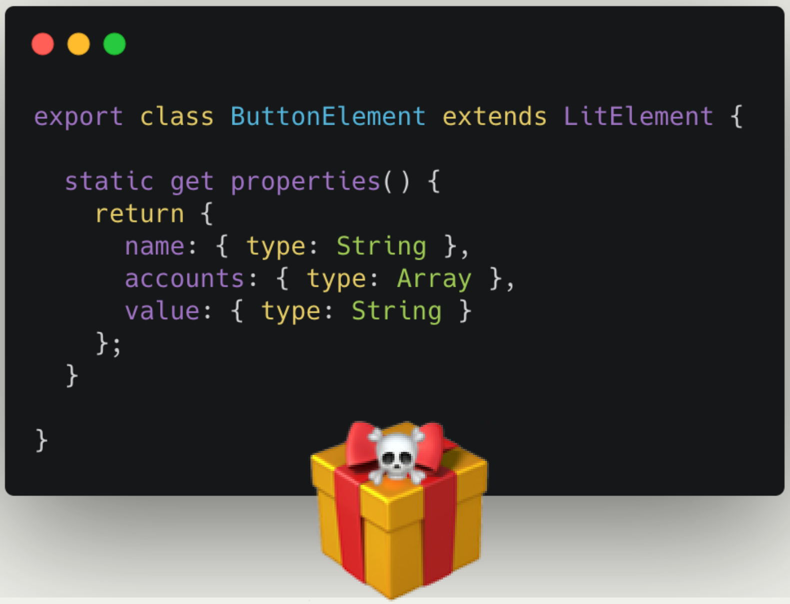
Speaker notes

Y finalmente tenemos una tercera que decide que el nombre adecuado es nickname. Es un caso bastante habitual entre equipos de desarrollo de componentes, en los que existe una necesidad, pero un dato tan poco relevante no es interesante de comentar en una daily.

PROPS

CONSUMED PROPS





```
export class ButtonElement extends LitElement {

  static get properties() {
    return {
      name: { type: String },
      accounts: { type: Array },
      value: { type: String }
    };
  }

}
```

Speaker notes

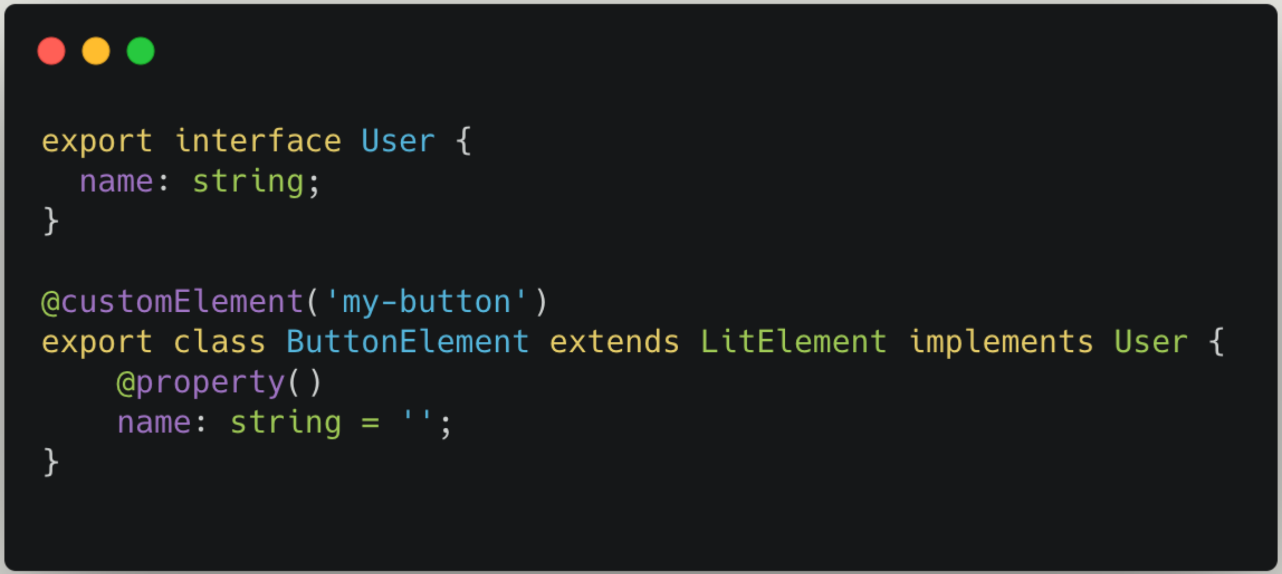
Al final, cada developer, prepara su componente con las propiedades necesarias, lo documenta y lo expone para su uso. Esto lo envía para que otro desarrollador, lo utilice al construir una vista, usando tanto el comp1, como el de las otras dos devs, provocando el resultado esperado

Speaker notes

Al final, el otro desarrollador, tiene que tener en cuenta tres propiedades, que van a recibir el mismo valor pero con el mismo nombre, tengo que acudir a la documentación de los 3 componentes, y ver como se llama cada propiedad, incluso, si es un objeto, mapearlo a 3 formatos de salida distintos, como podéis imaginaros, esto no es muy satisfactorio como experiencia de desarrollo.

PROPS

CONSUMED PROPS



```
export interface User {
  name: string;
}

@customElement('my-button')
export class ButtonElement extends LitElement implements User {
    @property()
    name: string = '';
}
```

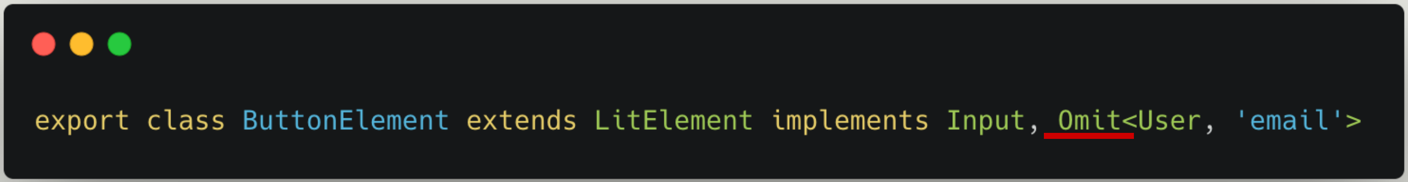
Speaker notes

Typescript, puede ayudarnos aquí para centrar el tiro. Si los 3 componentes comparten la implementación de una interfaz que, por ejemplo, define las propiedades del componente, esto ayudará bastante a la hora de mantener unas convenciones comunes usando componentes, en este caso, la clase ButtonElement, implementa el tipo User, obligando a que exista una propiedad que se llame name, dando un error de compilación si la misma no existe.

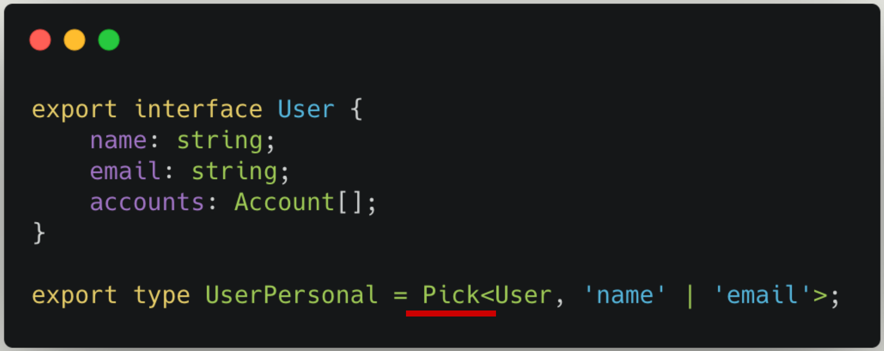
PROPS

CONSUMED PROPS

Utility types 💖



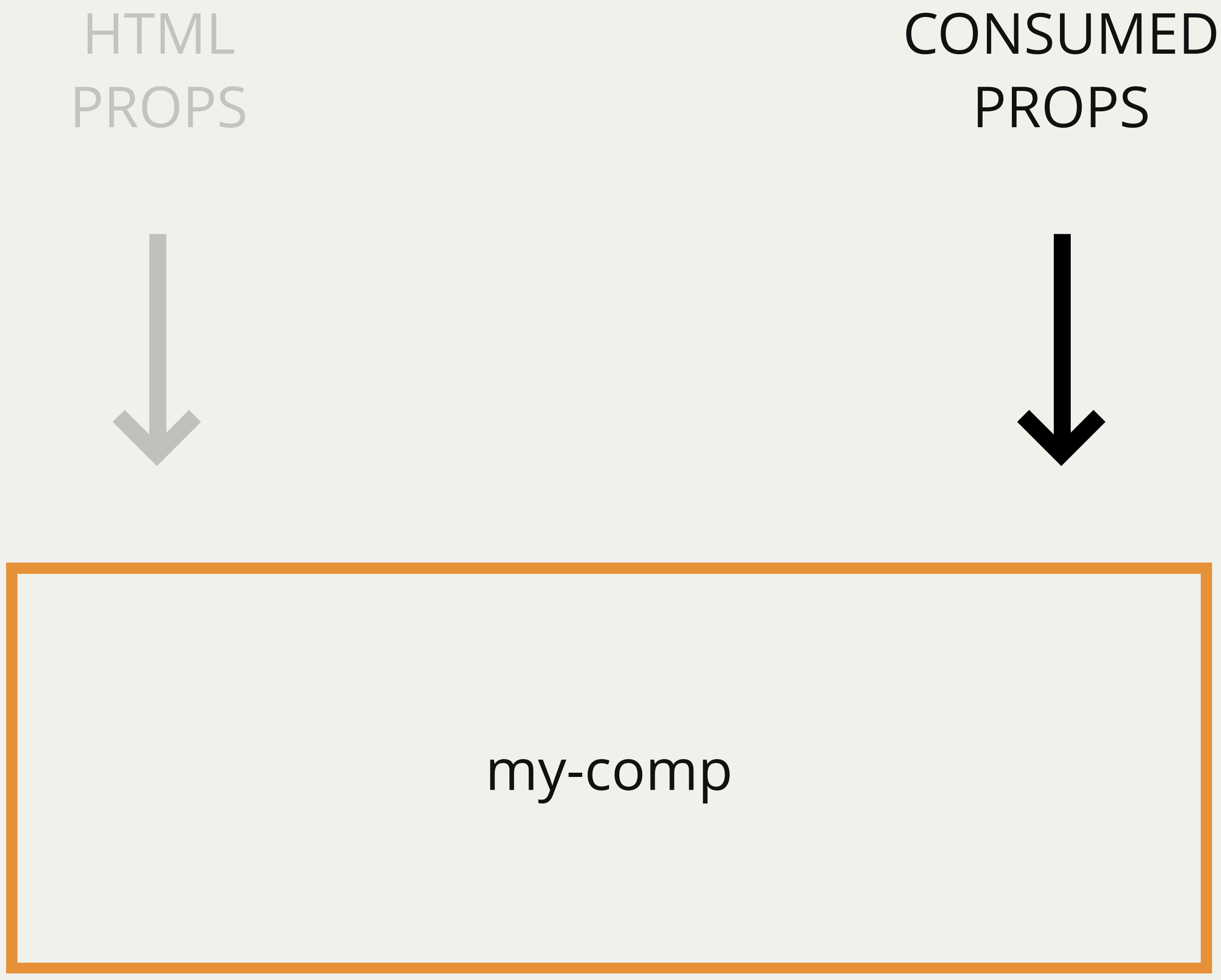
```
export class ButtonElement extends LitElement implements Input, Omit<User, 'email'>
```



```
export interface User {
    name: string;
    email: string;
    accounts: Account[];
}

export type UserPersonal = Pick<User, 'name' | 'email'>;
```

Speaker notes

Además de esto, si queremos implementar un subconjunto de las propiedades, o parte de esa interfaz, podemos utilizar los utility types de Typescript, que nos permiten crear nuevos tipos a raiz de la definición de una interfaz.

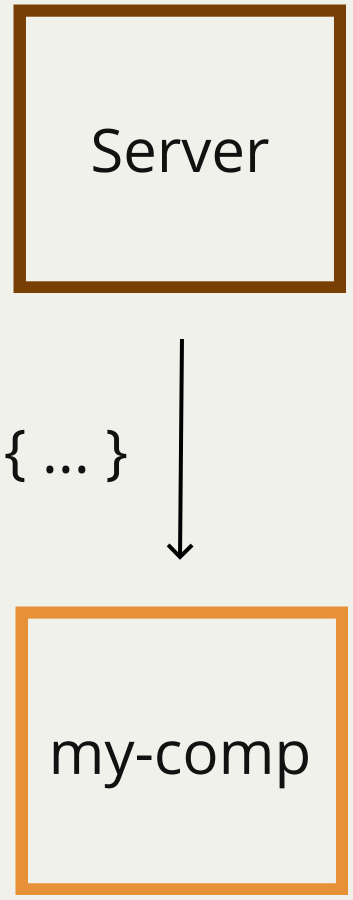

HTML
PROPS
CONSUMED
PROPS
my-comp

Speaker notes

Vamos a centrarnos ahora, en una de las bases para componentes listos, la recuperación de propiedades de back

HTML
PROPS

CONSUMED
PROPS

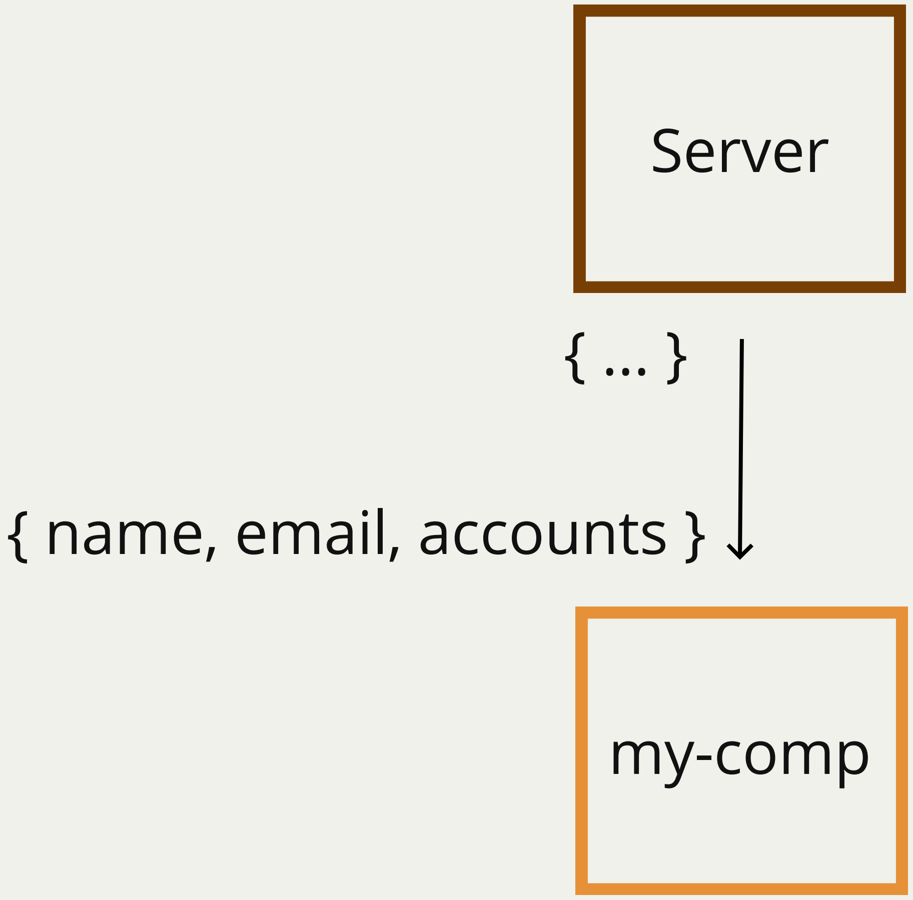


Speaker notes

Normalmente, nuestro componente de página, o de lógica, puede que requiera recuperar unos valores que envíe un servidor.

HTML PROPS

CONSUMED PROPS



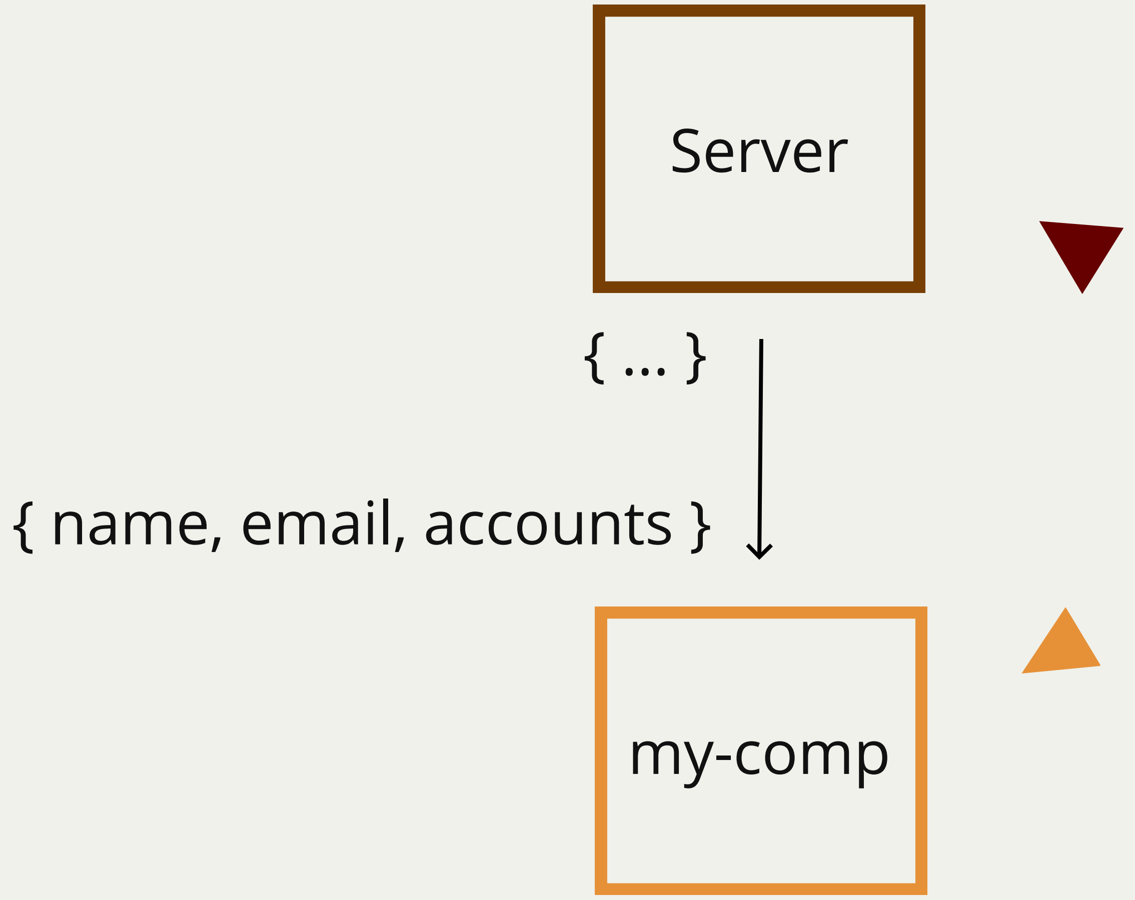
🙌 Contract 🙌

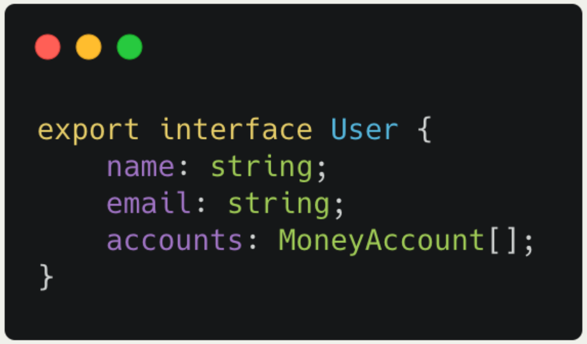
Speaker notes

Si queremos trabajar con una estructura de datos concreta, normalmente utilizamos un contrato, a través de algún tipo de documentación, o simplemente hablando, de como vamos a recibir y trabajar con ese objeto

HTML PROPS

CONSUMED PROPS





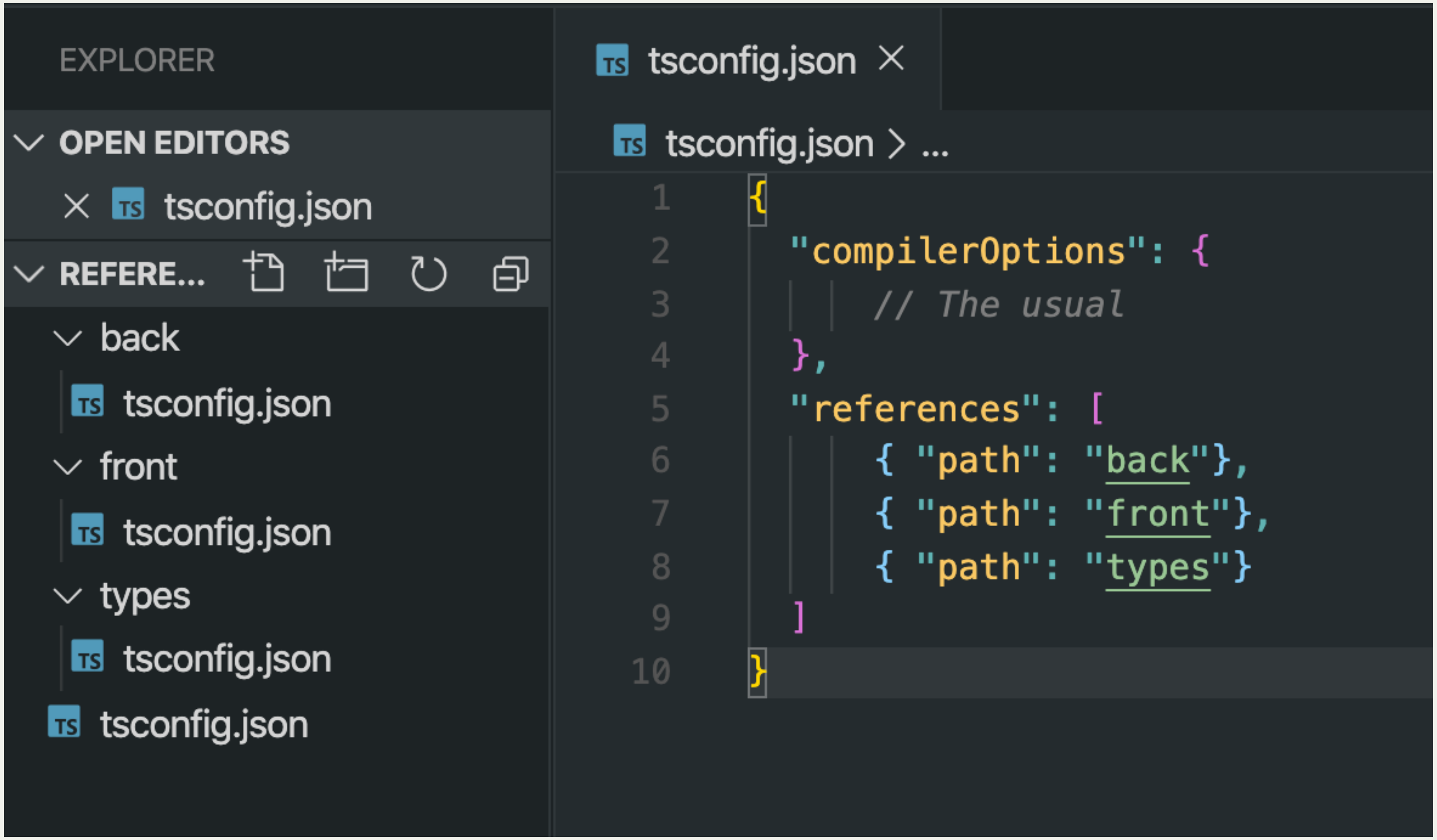
```
export interface User {
    name: string;
    email: string;
    accounts: MoneyAccount[];
}
```

Speaker notes

En el caso de TS esto se vuelve especialmente interesante, ya que podemos definir ese contrato, a través de una interfaz, con la que pueden trabajar tanto back (node) como front

HTML
PROPS

CONSUMED
PROPS



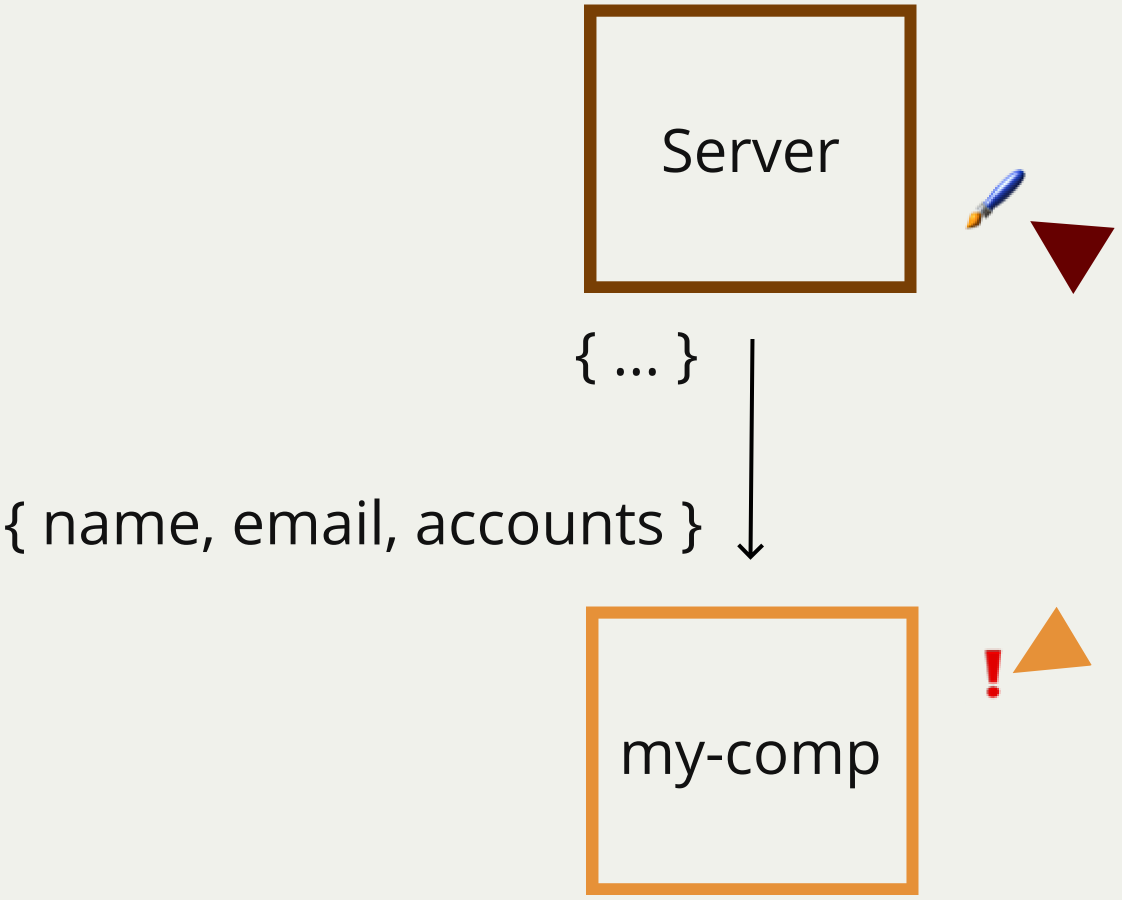
```
{
  "compilerOptions": {
    // The usual
  },
  "references": [
    { "path": "back"},
    { "path": "front"},
    { "path": "types"}
  ]
}
```

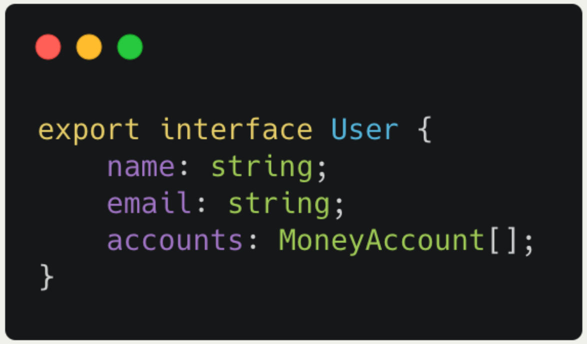
Speaker notes

Hay varias formas de hacerlo, por una parte podríamos usar un modulo de npm por separado, de tal forma que ambos paquetes dependan de nuestro paquete de tipos. Pero también podemos trabajar con Project references, un recurso que nos ofrece typescript para que el compilador pueda "recoger" tipos de otras carpetas o directorios

HTML
PROPS

CONSUMED
PROPS





```
export interface User {
    name: string;
    email: string;
    accounts: MoneyAccount[];
}
```

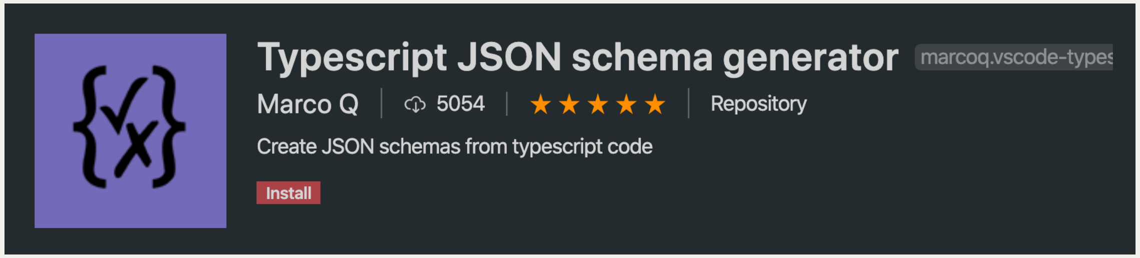
Speaker notes

De esta manera, si el servidor modifica un dato por su parte, se verá forzado a actualizar la interfaz para poder compilar, y esto hará que falle la compilación en nuestros componentes, advirtiendo a los mismos de que se ha producido un cambio.

HTML PROPS

CONSUMED PROPS

http://www.jsontots.com/


Typescript JSON schema generator marcoq.vscode-types

Marco Q | 5054 | ★★★★★ | Repository

Create JSON schemas from typescript code

https://github.com/vojtechhabarta/typescript-generator

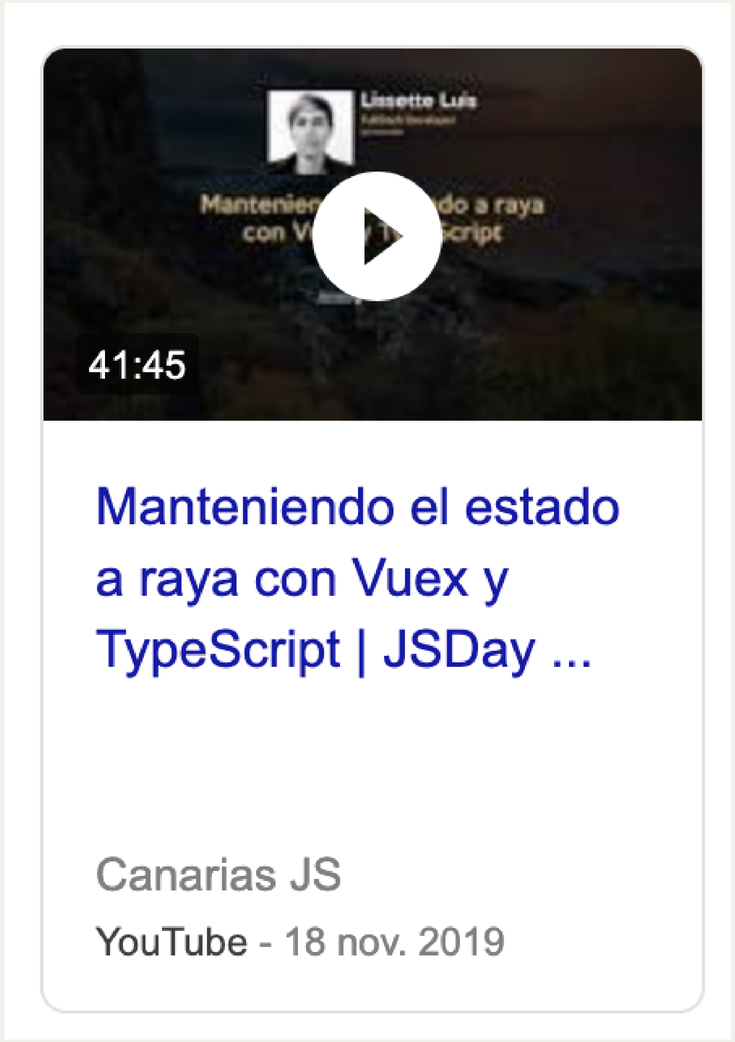
FRONT FEST

Speaker notes

Existen varias maneras de trabajar con servidores no javascript e interfaces. Estas son un ejemplo, pero ademas, existen numerosos paquetes para generar archivos .d.ts, que contengan los tipos de los lenguajes que trabajamos, como Java

HTML
PROPS

CONSUMED
PROPS



Manteniendo el estado a raya con Vuex y TypeScript | JSDay ...

Canarias JS

YouTube - 18 nov. 2019

TypeScript 💖 State Managers

react-redux-typescript-guide

Speaker notes

Existe otra fuente de datos habitual, como los state manage. No vamos a entrar en profundidad, aunque podeis imaginar la potencia que tiene a la hora de definir y utilizar acciones, y lo mucho que puede ayudarnos. Por ejemplo, al decir que una acción es de tal tipo, obligarnos a definir los parámetros requeridos por el reducer.

Decorators 💖

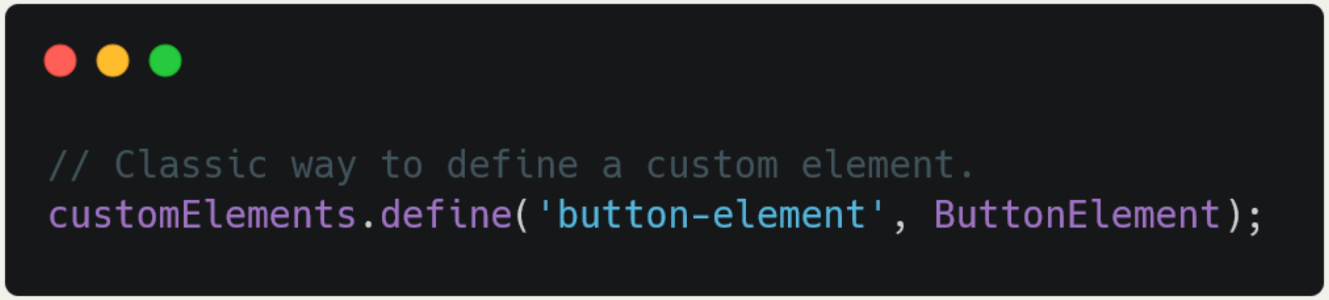
```
// Decorators are placed before the element they are influencing
@customElement('button-element');
export class ButtonElement extends LitElement{
  ...
}
```

FRONT FEST

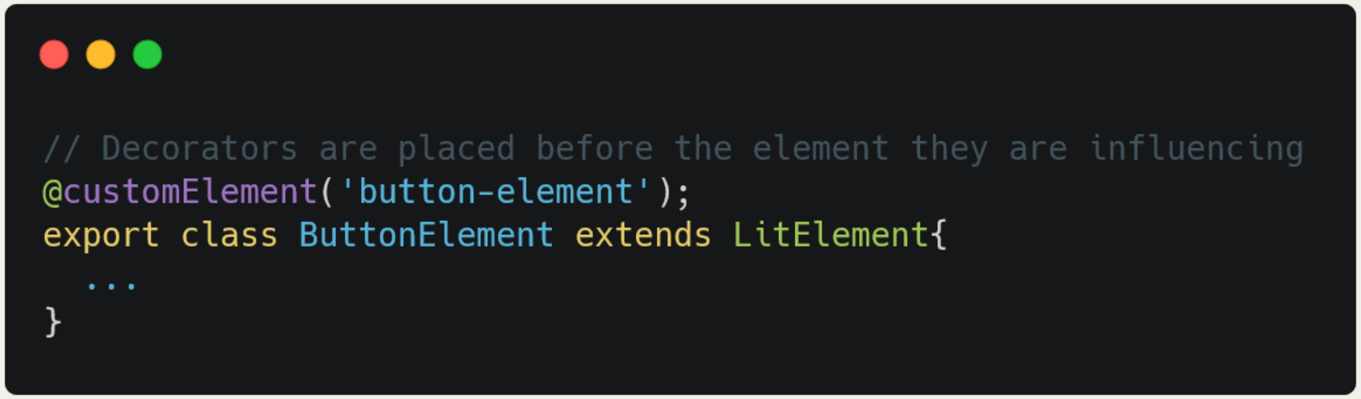
Speaker notes

Decorators are just functions that are equivalent to other funcions. It's important to note that they are just syntactic sugar for larger functions. They are nice because they make our code look cleaner but in terms of functionality they're just the same. When working with a big catalog of components it can make annoyingly repetitive parts of our code simpler, clearer to look at while still being self explanatory.

Decorators 💖



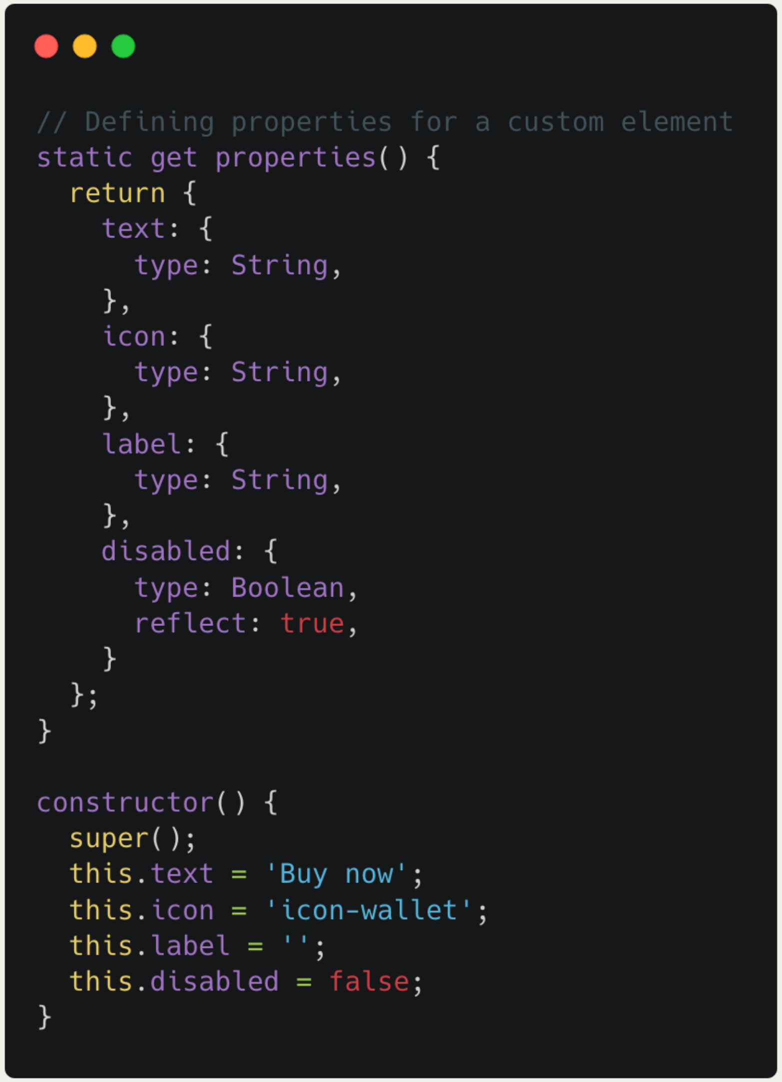
```
// Classic way to define a custom element.
customElements.define('button-element', ButtonElement);
```



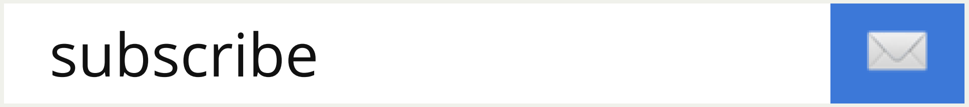
```
// Decorators are placed before the element they are influencing
@customElement('button-element');
export class ButtonElement extends LitElement{
  ...
}
```

FRONT FEST

DECORATORS



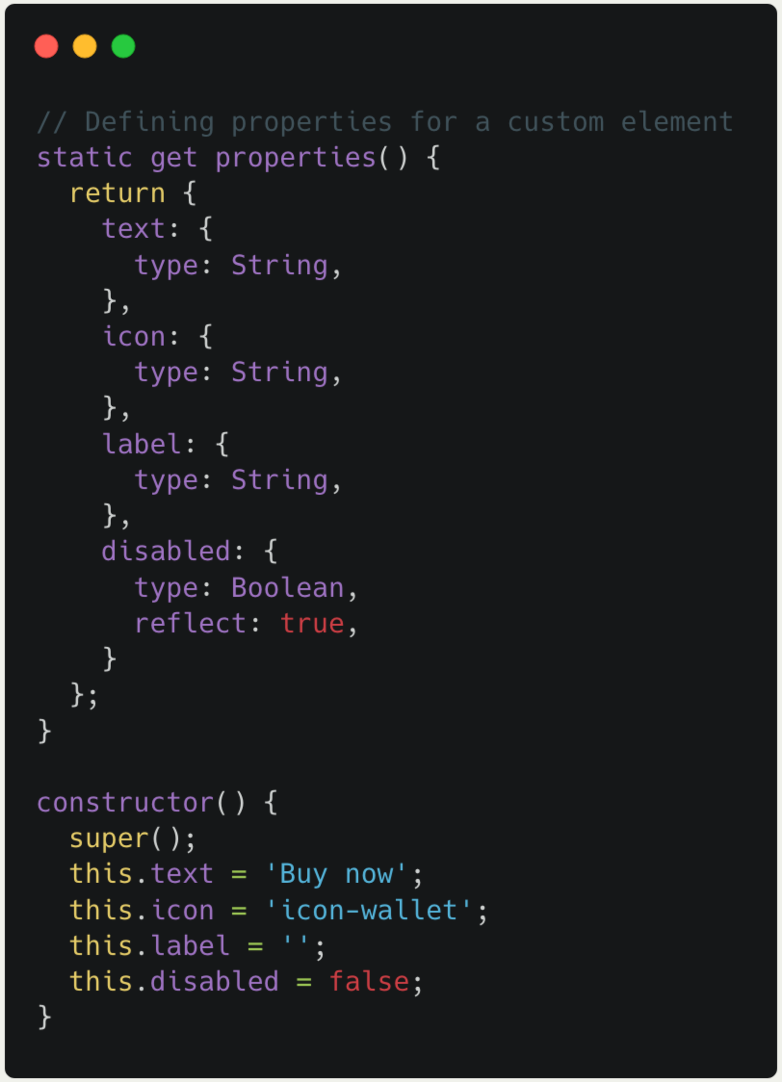
```
// Defining properties for a custom element
static get properties() {
  return {
    text: {
      type: String,
    },
    icon: {
      type: String,
    },
    label: {
      type: String,
    },
    disabled: {
      type: Boolean,
      reflect: true,
    }
  };
}

constructor() {
  super();
  this.text = 'Buy now';
  this.icon = 'icon-wallet';
  this.label = '';
  this.disabled = false;
}
```



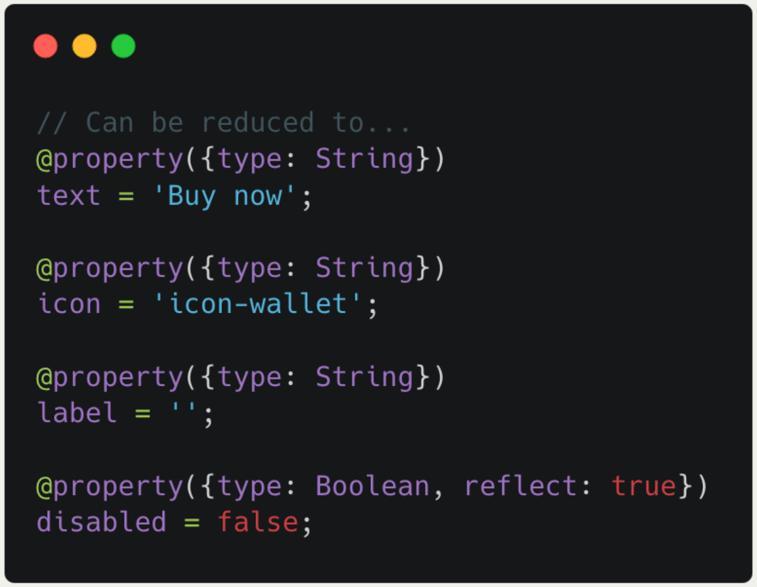
Speaker notes

Using decorators we can get a clearer and shorter view of a components class and property definitions. Since now everything is defined in the same place.

DECORATORS

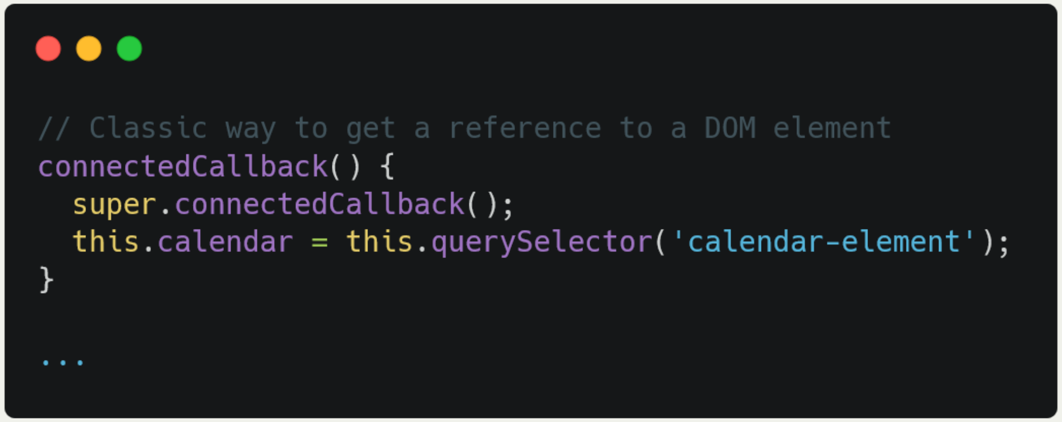


```
// Defining properties for a custom element
static get properties() {
  return {
    text: {
      type: String,
    },
    icon: {
      type: String,
    },
    label: {
      type: String,
    },
    disabled: {
      type: Boolean,
      reflect: true,
    }
  };
}

constructor() {
  super();
  this.text = 'Buy now';
  this.icon = 'icon-wallet';
  this.label = '';
  this.disabled = false;
}
```

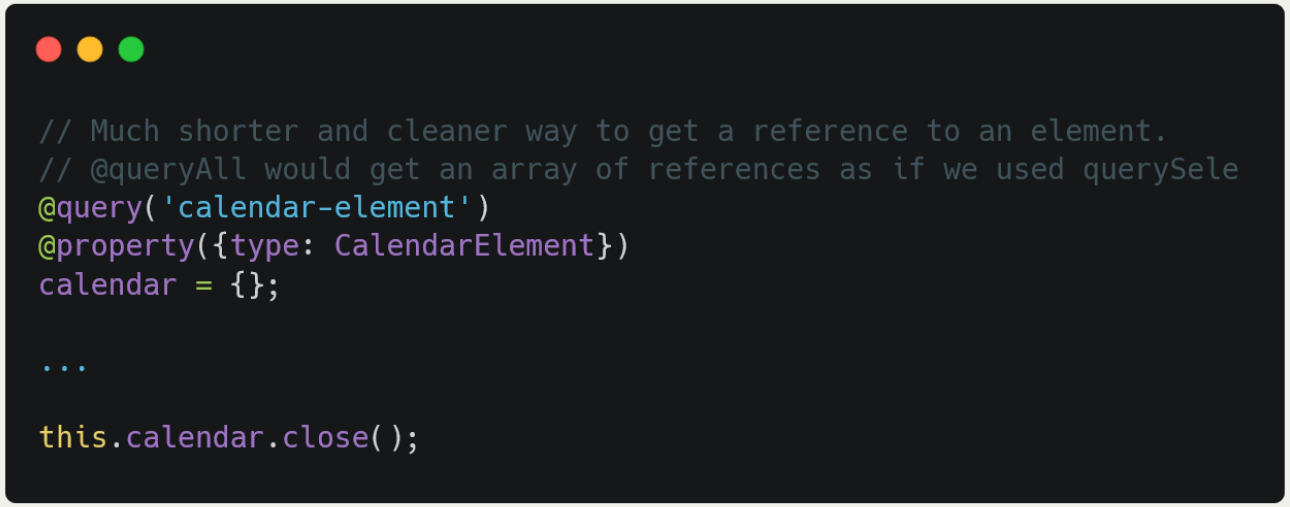


```
// Can be reduced to...
@property({type: String})
text = 'Buy now';

@property({type: String})
icon = 'icon-wallet';

@property({type: String})
label = '';

@property({type: Boolean, reflect: true})
disabled = false;
```

DECORATORS



```
// Classic way to get a reference to a DOM element
connectedCallback() {
  super.connectedCallback();
  this.calendar = this.querySelector('calendar-element');
}

...
```



```
// Much shorter and cleaner way to get a reference to an element.
// @queryAll would get an array of references as if we used querySele
@query('calendar-element')
@property({type: CalendarElement})
calendar = {};

...

this.calendar.close();
```

Speaker notes

One of the most useful decorators I have used is this one. There are times in a component were we need to have a reference to another element inside our component. For that we need a querySelector, which we can shorten with this decorator. This way we don't have the need to wait for the connectedCallback event to look for it and it's pretty clear what we're trying to get a reference from.

DECORATORS

Vue https://github.com/kaorun343/vue-property-decorator

LitElement https://lit-element.polymer-project.org/api/modules/_lib_decorators_.html

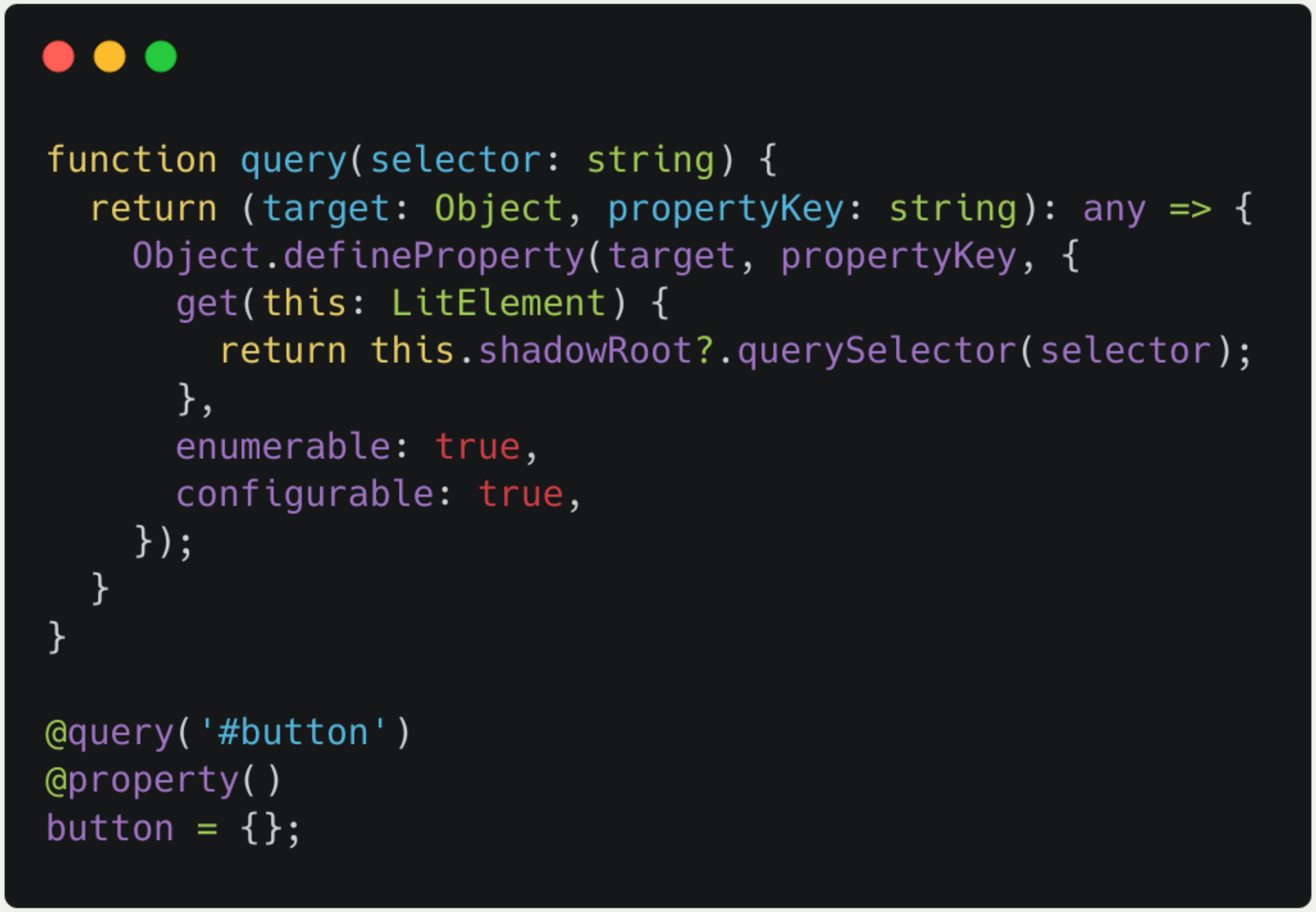
Angular - Native

React - Use your own

Speaker notes

Every framework has some kind of decorators, some of them are native to them but the others usually have some kind of popular repo you can check out.

🌟 MAKE YOUR OWN DECORATORS 🌟



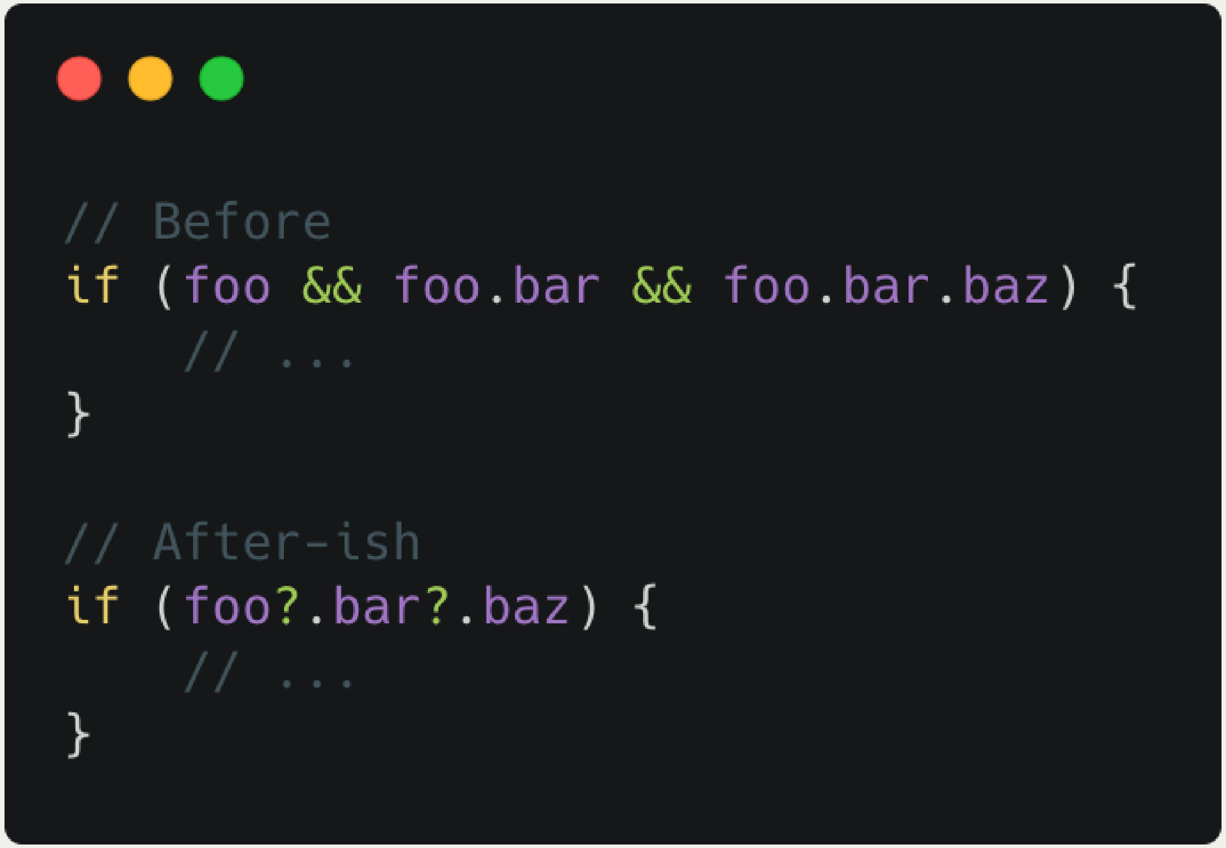
```
function query(selector: string) {
  return (target: Object, propertyKey: string): any => {
    Object.defineProperty(target, propertyKey, {
      get(this: LitElement) {
        return this.shadowRoot?.querySelector(selector);
      },
      enumerable: true,
      configurable: true,
    });
  }
}

@query('#button')
@property()
button = {};
```

FRONT FEST

Speaker notes

Here we can see a simplified declaration of the query decorator used before. Have in mind that decorators can only be used on and inside classes, and need a tsconfig flag turned on, called experimentalDecorators.

OPTIONAL CHAINING



```
// Before
if (foo && foo.bar && foo.bar.baz) {
    // ...
}

// After-ish
if (foo?.bar?.baz) {
    // ...
}
```

@jnroji

@EliRP95