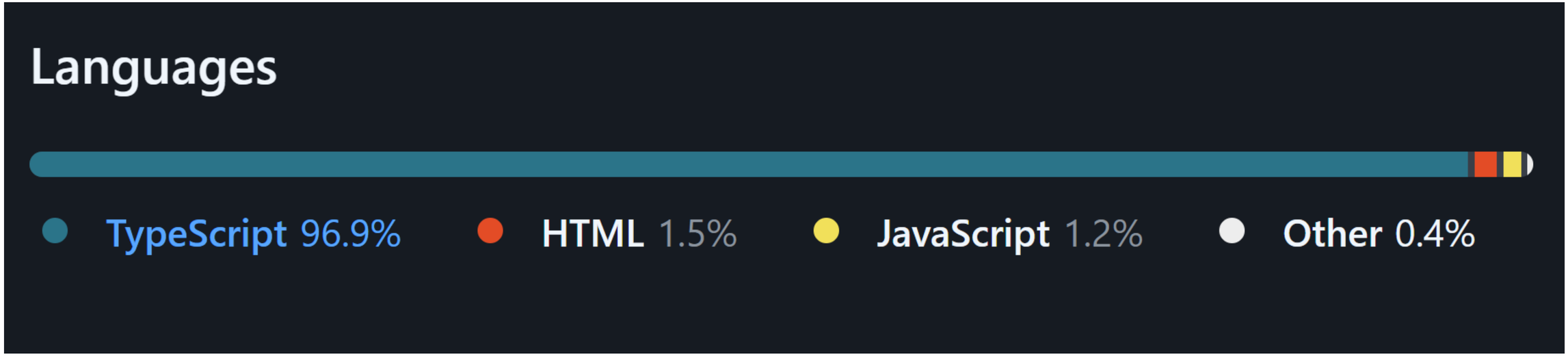
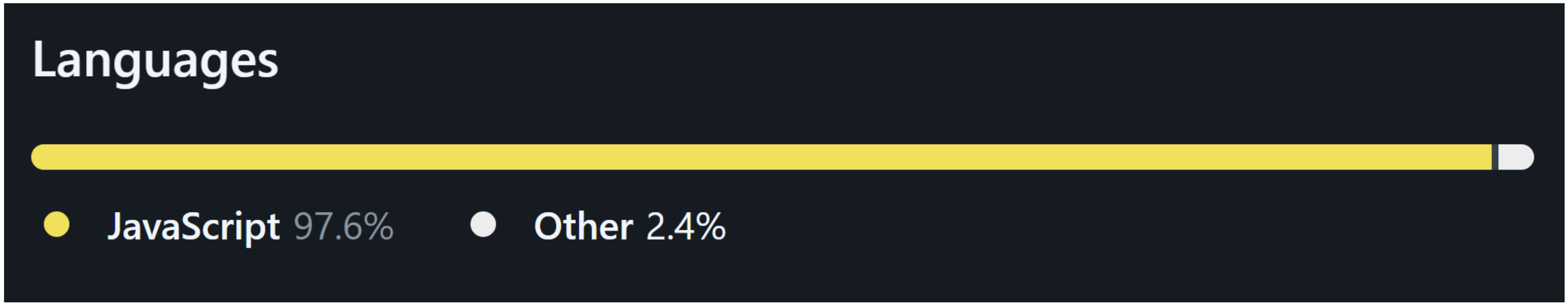
Vue 3 is built using typescript.



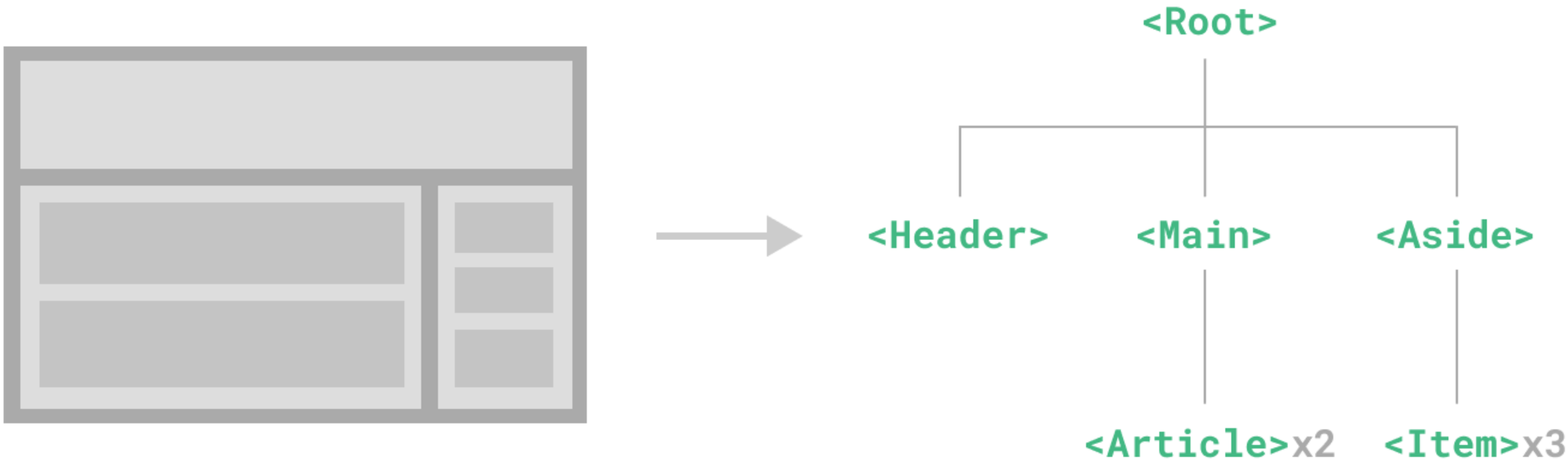
Vue 2 is built using javascript (with `Flow`) For now



What is a Vue Component?

Source: Vue.js docs

Components allow us to split the UI into independent and reusable pieces, and think about each piece in isolation. It’s common for an app to be organized into a tree of nested components:



Vue Component Typescript

Vue component contain different types depending on the context:

- Component Options (Not only Options API)

```
const MyComp = defineComponent({
  props: ['bar']
  data: ()=>  ({ foo: 1 }),
  computed: {
    odd() {
      return this.bar + this.foo;
    },
  },
});
```

- Render Component

```
<MyComp ref="comp" :bar="3" />
```

- Public Instance

```
this.$refs.comp.foo; // 1
```

Options

Options is the way we declare a component in Vue (using Options API, Composition-API, Function API or Class API*)

It requires some overloads for `defineComponent`:

- Function component
- No properties component
- Array declaration props component

```
defineComponent({ props: ["foo", "bar"] });
```

- Object props component

```
defineComponent({ props: { foo: Number } });
```

```
import { defineComponent } from "vue";

defineComponent({
  data: () => ({ foo: 1 }),
  mounted() {
    this.foo = 1;
  },
});
```

* plugin

Render Component

Render component is the TSX compatible usually used in the `<template>`

```ts
import { defineComponent, h } from "vue";
const Comp = defineComponent({ props: { foo: Number } });

h(Comp, { foo: 1 }); // equivalent to <Comp foo={1} />;
// @ts-expect-error foo should be a number
h(Comp, { foo: "1" });
```

For this to work it needs to a constructor and have `$props` (and other $ properties)

```ts
import { h } from "vue";
declare const Comp: { new (): { $props: { foo: number } } }; // Fake vue render component

// @ts-expect-error required prop
h(Comp, {});
// @ts-expect-error foo should be number
h(Comp, { foo: "1" });
h(Comp, { foo: 1 });
```

Public Instance

Component proxy (aka `this`, `getCurrentInstance().proxy` and `Template Ref`)

```
import { defineComponent } from "vue";
// Options
const MyComp = defineComponent({
  props: ["a"],
  mounted() {
    console.log(this.a);
  },
});
// Composition API
getCurrentInstance().proxy
// Template ref
<my-comp ref="(e)=>e">
```

defineComponent

> defineComponent is a utility that is primarily used for type inference when declaring components.

It has 4 overloads where only the **option** type changes:

- Functional Component

```
defineComponent((props) => h("div", "Hello World " + props.name));
```

- No properties component

```
defineComponent({
  render() {
    return h("div", "Hello World");
  },
});
```

- Array declaration props component

```
defineComponent({
  props: ["name"],
  render() {
    return h("div", "Hello World " + this.name);
  },
});
```

`defineComponent` Functional Component

- Generics:
  - Props: Props type
  - RawBindings: Type if what's returned is an object
- Setup: Render function
- Return: `DefineComponent` with props and with `RawBindings`

```ts
// overload 1: direct setup function
// (uses user defined props interface)
function defineComponent<Props, RawBindings = object>(
  setup: (
    props: Readonly<Props>,
    ctx: SetupContext
  ) => RawBindings | RenderFunction
): DefineComponent<Props, RawBindings>;
```

`defineComponent` Options API + Composition-API

Options API and Composition-API are handled by the same overloads, this allows the usage of both API on the same component:

```
defineComponent({
  setup: () => ({ foo: 1 }),
  mounted() {
    this.foo; //1
  },
});
```

To handle different props declaration we have 3 more overloads, all of them are built on top of ComponentOptionsBase

LegacyOptions

This are options focused for Options API specific.

- Props: Properties with type `PropType`
- D: aka `Data`, returned from `data()`
- C: aka `computed` options object
- M: aka `methods` options object
- Mixin: Extending based on mixin array
- Extends: Extending based on Extends
- Others: `Watch`, `provide`, `inject`, etc

```ts
interface LegacyOptions<
  Props,
  D,
  C extends ComputedOptions,
  M extends MethodOptions,
  Mixin extends ComponentOptionsMixin,
  Extends extends ComponentOptionsMixin
> {
  data?: (/* Props, is used here. omitted*/) => D;
  computed?: C;
  methods?: M;
  mixins?: Mixin[];
  extends?: Extends;
  watch?: ComponentWatchOptions;
  provide?: Data | Function;
  inject?: ComponentInjectOptions;

  // allow any custom options
  [key: string]: any;

  /* code omitted */
}
```

ComponentOptionsBase

Options used for OptionsAPI and Composition-API

- RawBindings: Return from `setup()`
- E & EE: Emit options & Emit options keys
- Defaults: Property defaults
- Setup
- Others

```ts
export interface ComponentOptionsBase<
  Props,
  RawBindings,
  D,
  C extends ComputedOptions,
  M extends MethodOptions,
  Mixin extends ComponentOptionsMixin,
  Extends extends ComponentOptionsMixin,
  E extends EmitsOptions,
  EE extends string = string,
  Defaults = {}
> extends LegacyOptions<Props, D, C, M, Mixin, Extends>,
    ComponentInternalOptions,
    ComponentCustomOptions {
  setup?: (
    props,
    ctx: SetupContext<E>
  ) => /**/ RawBindings | RenderFunction | void;
  name?: string;
  template?: string | object;
  render?: Function;
  components?: Record<string, Component>;
  directives?: Record<string, Directive>;
  inheritAttrs?: boolean;
  emits?: (E | EE[]) & ThisType<void>;
  expose?: string[];
}
```

Component Options

- ComponentOptionsWithoutProps
- ComponentOptionsWithArrayProps
- ComponentOptionsWithObjectProps

```
type ComponentOptionsWithoutProps</* Generics */> = {
  props?: undefined;
} & ComponentOptionsBase</* Generics */> &
  ThisType<CreateComponentPublicInstance</* Generics */>>;

type ComponentOptionsWithArrayProps<PropNames /**/> = {
  props: PropsNames[];
} & ComponentOptionsBase</* Generics */> &
  ThisType<CreateComponentPublicInstance</* Generics */>>;

type ComponentOptionsWithObjectProps<PropsOptions /**/> = {
  props: PropsOptions & ThisType<void>;
} & ComponentOptionsBase</* Generics */> &
  ThisType<CreateComponentPublicInstance</* Generics */>>;
```

`defineComponent` overloads

- `Direct Setup Function`: Functional component
- `No props`
- `Array props`
- `Object props`
- Actual implementation

```ts
function defineComponent<Props, RawBindings = object>(
  setup: (
    props: Readonly<Props>,
    ctx: SetupContext
  ) => RawBindings | RenderFunction
): DefineComponent<Props, RawBindings>;

function defineComponent</* Generics */>(
  options: ComponentOptionsWithoutProps</**/>
): DefineComponent</**/>;

function defineComponent</* Generics */>(
  options: ComponentOptionsWithArrayProps</**/>
): DefineComponent</**/>;

function defineComponent</* Generics */>(
  options: ComponentOptionsWithObjectProps</**/>
): DefineComponent</**/>;

export function defineComponent(options: unknown) {
  return isFunction(options) ? { setup: options, name: opti
}
```

DefineComponent

Return type of `defineComponent`, containing Render Component and Public Instance

- `Render Component & Instance Component`
- `Options`
- Public props: VNodeProps, AllowedComponentProps and ComponentCustomProps

```
type DefineComponent</* Generics */> = ComponentPublicInsta
  CreateComponentPublicInstance</*Generics*/> & Props
> &
  ComponentOptionsBase</* Generics */> &
  PP;
```

Usage

```vue
<script setup>
  import { ref } from 'vue';
  import MyComponent from './MyComponent.vue';

  // Accessing component options *
  const ComponentOptions = MyComponent.props

  // Instance Component Type
  type MyComponentInstance = ReturnType<typeof MyComponent>

  // Instance Component
  const compEl = ref<MyComponentInstance | null>(null)
</script>
<template>
  <!-- Render Component -->
  <my-component ref="compEl">
</template>
```

- * Not yet correct typed, waiting for @vuejs/core#5416

Testing types

- You can use TSX or import h.
  - TSX allows greater control and better errors
  - h errors are a bit cryptic
- Volar and VueDX both use TSX syntax for Typechecking.

Augmenting Components

```ts
import { defineComponent, DefineComponent } from "vue";
interface KnownAttributes {
  special: number;
}
function augmentComponentProps<T extends DefineComponent<any, any, any>>(
  t: T
): T & { new (): { $props: KnownAttributes } } {
  return t;
}
const Comp = augmentComponentProps(
  defineComponent({
    props: {
      test: Number,
    },
  })
);

<Comp test={1} special={2} />;
// @ts-expect-error invalid special
<Comp test={1} special={"2"} />;
// @ts-expect-error invalid test
<Comp test={"1"} special={2} />;
```

- source

Level up with Generic props

```ts
import { defineComponent, DefineComponent } from "vue";
declare class GenericTypedProps<T> {
  declare $props: {
    value: T;
    onChange: (e: T) => void;
  };
}
function augmentComponentProps<T extends DefineComponent<any, any, any>>(
  t: T
): T & typeof GenericTypedProps {
  return t;
}
const Comp = augmentComponentProps(defineComponent({}));

<Comp value={2} onChange={(e) => e + 1} />;
// @ts-expect-error invalid change
<Comp value={{ a: 1 }} onChange={(e) => e + 1} />;
```

- source

Thank you

You can check this talk on

github.com/pikax/talks/tree/master/2022-04-28