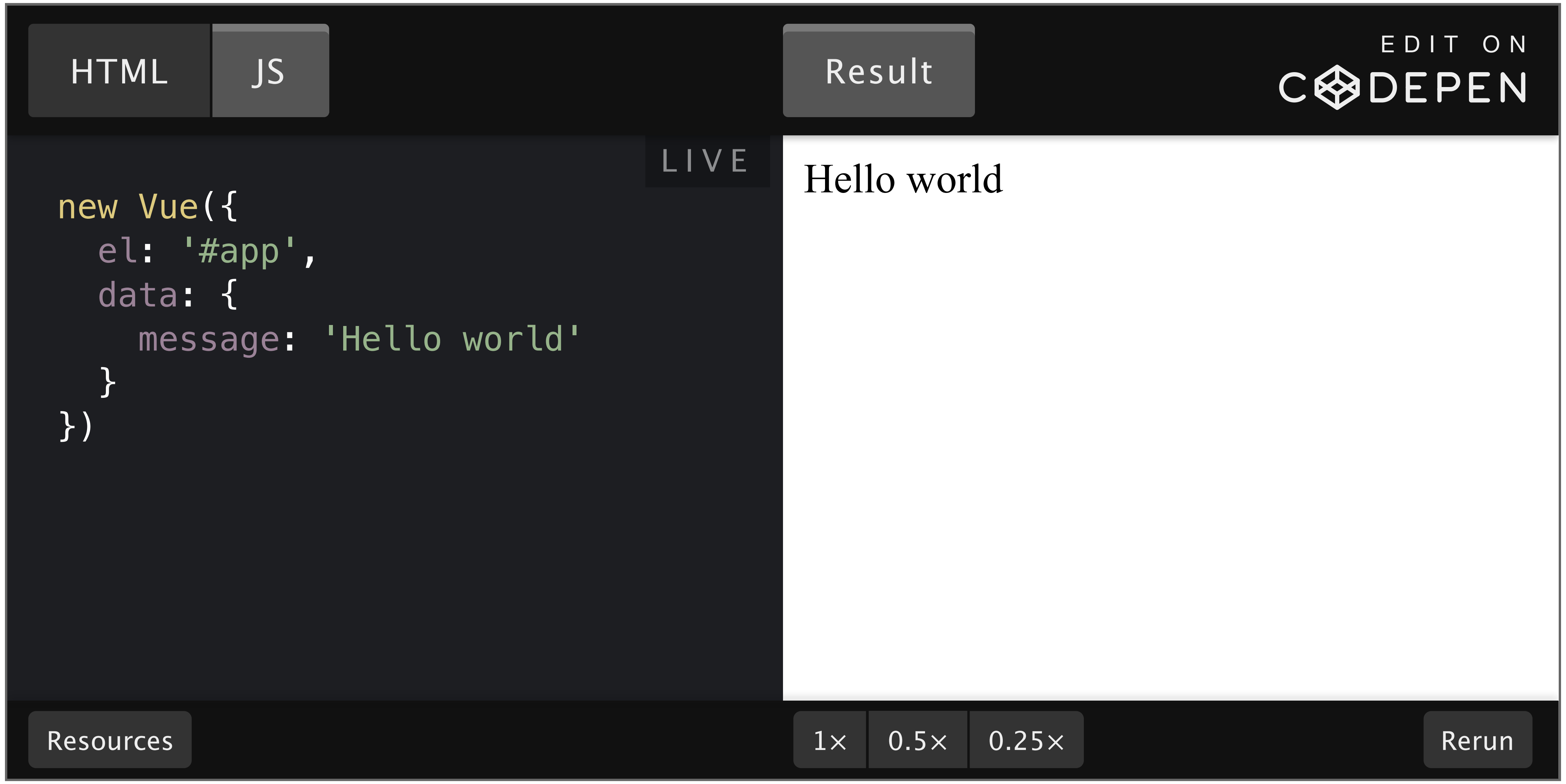
Vue.js & Nuxt.js

Frontend Made Declarative

What is Vue.js

- Progressive Framework for building UI
- Declarative Rendering, MVVM
- Components



```
new Vue({
  el: '#app',
  data: {
    message: 'Hello world'
  }
})
```

Hello world

Change `message` in `data` and see what happens.

MVVM

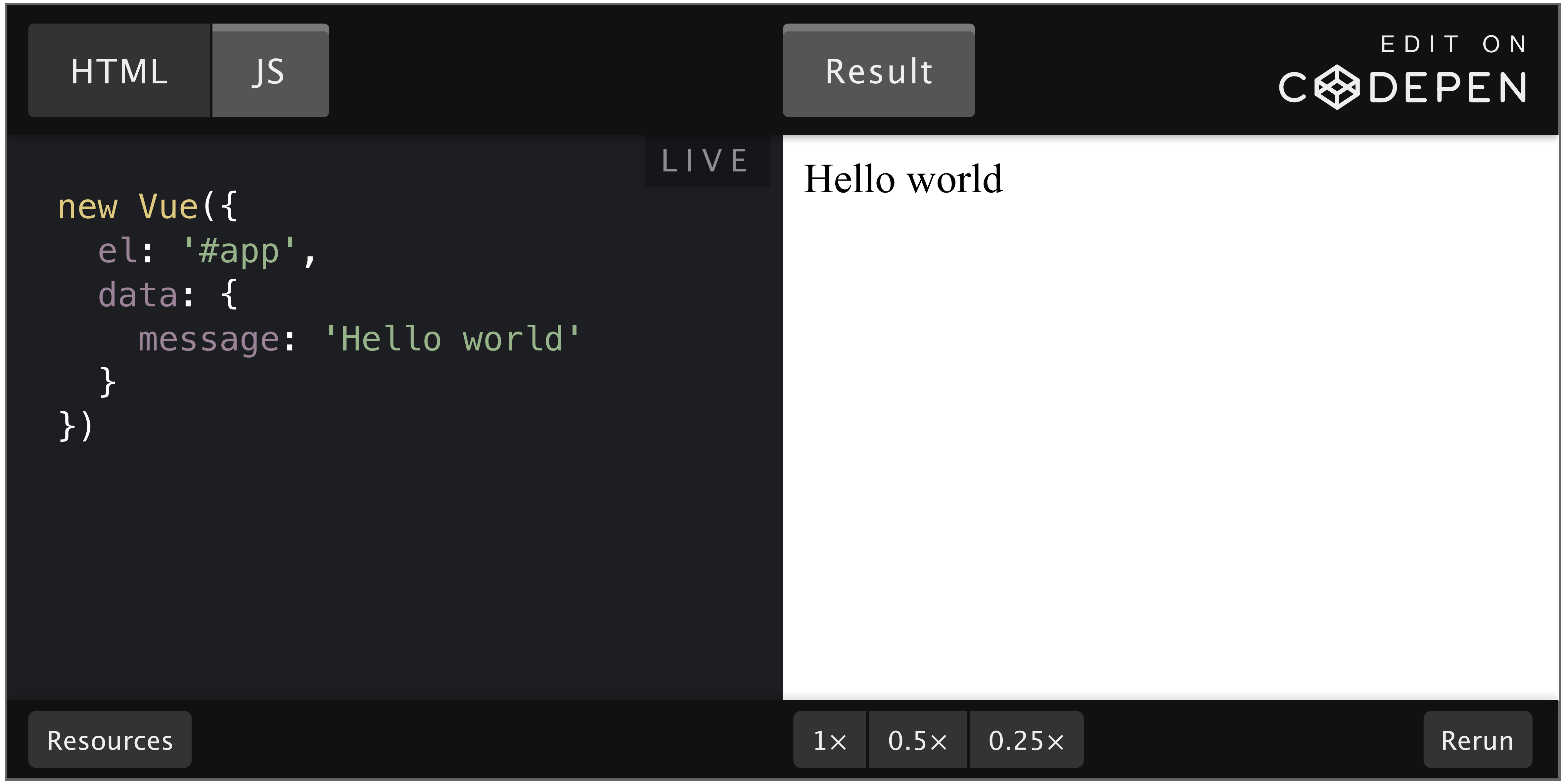
- Apply all modifications of data to the ViewModel
- The view code, i.e. DOM Tree, will update accordingly

```
new Vue({ // create the ViewModel
  el: '#app', // mount the Vue app at #app
})
```

v-if, v-show, v-for, v-on,
v-bind, v-model, v-text, v-html,
v-cloak, ...

Template & Directives

- `{{ variableName }}` interpolate variable into your HTML code. (`v-text` is similar)
- `<div v-if="cond">Test</div>` only rendered when `cond` is a truthy value.
  - otherwise, the element is taken away from the DOM tree.
- `<div v-show="cond">Test</div>` set `display: none;` when cond is a falsy value.



```
new Vue({
  el: '#app',
  data: {
    message: 'Hello world'
  }
})
```

Hello world

Sometimes we want to declare methods in a Vue instance, which can be then used in event callbacks, etc.

Declare them in the `methods` property:

```
new Vue({
  el: '#app', // currentVal is then injected into `this`
  data: { currentVal: 1 },
  methods: {
    increaseBy (difference) { this.currentVal += difference }
  }
})
```

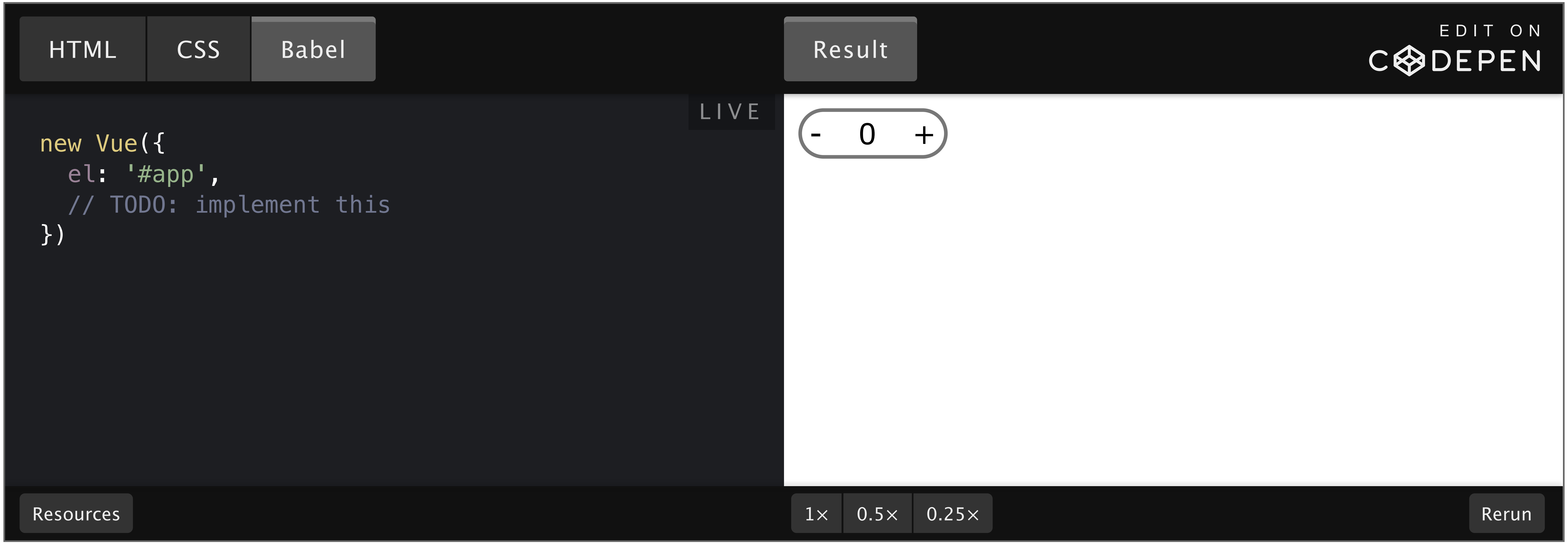
At times, we need to compute a property whose value is based on items in `data`, while maintaining reactivity, i.e. its value will update accordingly if one of its dependency changes. Use `computed` for this, and use `plusTwo` in the template like a normal data property.

```
new Vue({
  el: '#app', // currentVal is then injected into `this`
  data: { currentVal: 1 },
  computed: { // will be updated when `currentVal` changes. otherwise the cached value will be used.
    plusTwo () { return this.currentVal + 2 }
  }
})
```

Use `v-on` to listen on events. `v-on:click` is equivalent to `@click`.

```
<button v-on:click="handleClick">Click Me</button> <!--name of callback function-->
<button v-on:click="counter += 1">Click Me</button> <!--a single statement-->
```

HAVE A TRY!



```
new Vue({
  el: '#app',
  // TODO: implement this
})
```

Use `v-for` to implement a loop in templates. In the following code, `list` is an array declared in data.

```
<div id="app">
  <ul>
    <li v-for="x in list">{{ x }}</li>
  </ul>
</div>
```

```
new Vue({
  el: '#app',
  data: { list: [9, 8, 7] }
})
```

...will be rendered as:

```
<div id="app">
  <ul>
    <li>9</li><li>8</li><li>7</li>
  </ul>
</div>
```

When `list` is modified, the corresponding parts in HTML are also re-rendered.

Use `` `v-bind` `` to bind an attribute to the view model. When the data in view model is modified, the attribute with `` `v-bind` `` will also be updated. (shorthand: `` `v-bind:prop="var"` `` ⇔ `` `:prop="var"` ``)

Use `` `v-model` `` on form elements (e.g. `` `<input>` `` and `` `<select>` ``) to bind the form element with a variable in `` `data` ``.

How `` `v-model` `` works

- Edit the form element → `` `input` `` event triggered → data in view model changes
- Edit variable in view model → `` `value` `` attribute binded onto the variable → content in the element changes

```html
<input v-model="text">
```

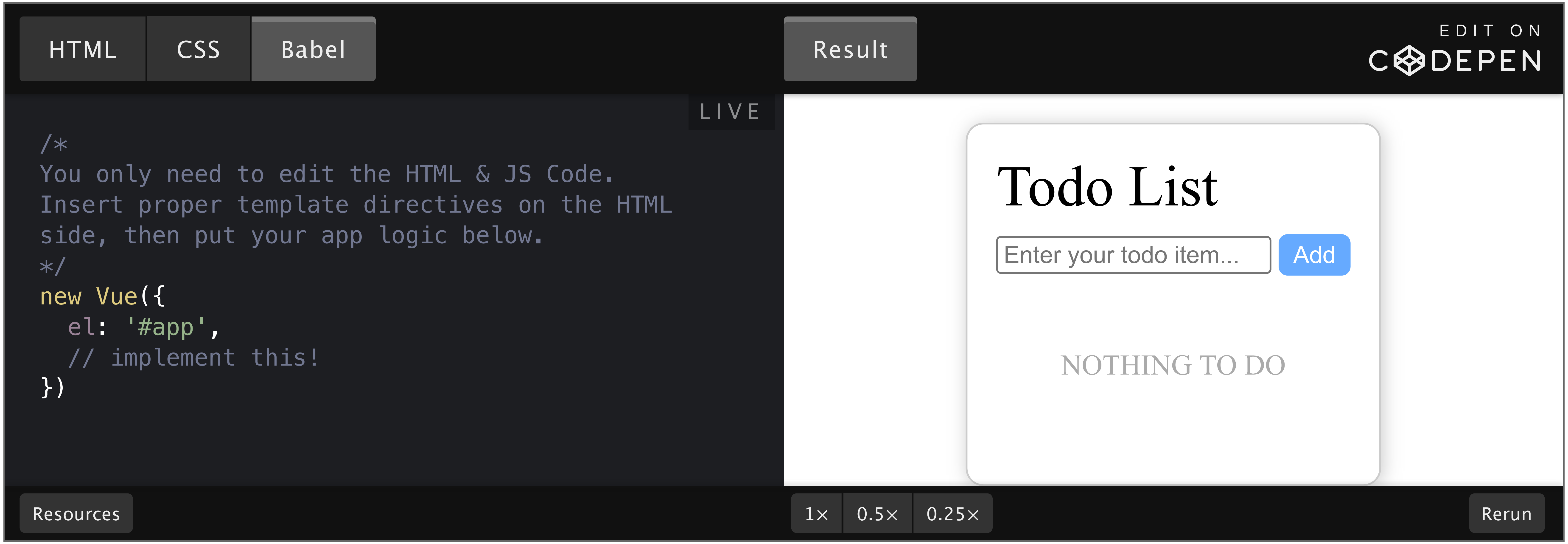
...is roughly equivalent to:

```html
<input :value="text" @change="handleChange">
```

and

```javascript
new Vue({
  el: '#app',
  data: { text: '' },
  methods: {
    handleChange(evt) {
      this.text = evt.target.value
    }
  }
})
```

Have a try!

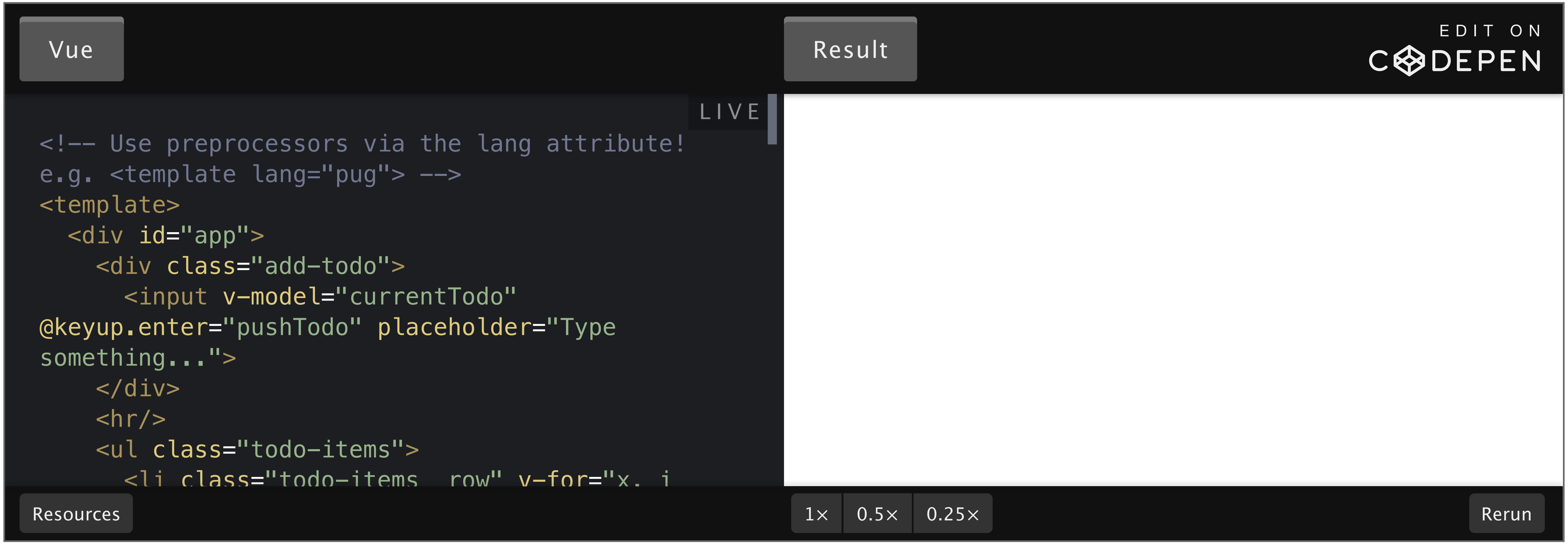


```
/*
You only need to edit the HTML & JS Code.
Insert proper template directives on the HTML
side, then put your app logic below.
*/
new Vue({
  el: '#app',
  // implement this!
})
```

Single File Component

- Write HTML, CSS & JavaScript in a single `.vue` file.
- Used in `vue-cli` and Nuxt.js.
- Separated into 3 tags: `<template>` `<script>` & `<style>`

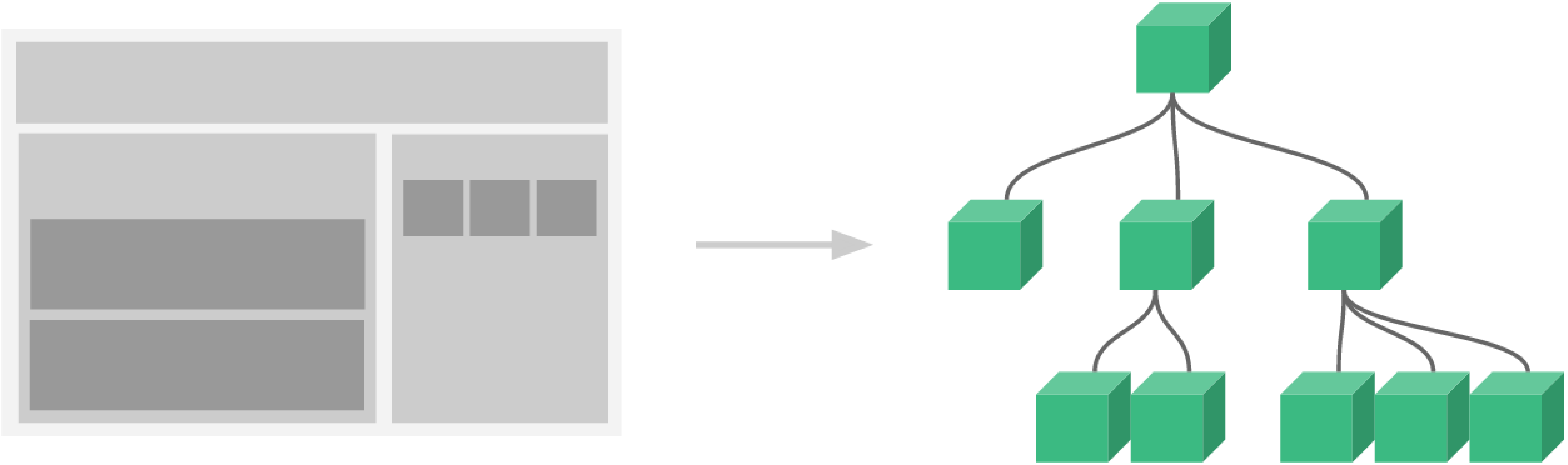
Example



```
<!-- Use preprocessors via the lang attribute! e.g. <template lang="pug"> -->
<template>
  <div id="app">
    <div class="add-todo">
      <input v-model="currentTodo" @keyup.enter="pushTodo" placeholder="Type something...">
    </div>
    <hr/>
    <ul class="todo-items">
      <li class="todo-items  row" v-for="x, i
```

Composing with Components

- abstraction: application to tree of components
- components are small, self-contained and often reusable



ENCAPSULATION

```
<div> <!--From http://slides.com/sdrasner/intro-to-vue-3?token=LwIVIblm#/4/0/2-->
  <p></p>
  <div></div>
  <p></p>
  <small></small>
</div>
```

↓

```
<call-out />
```

DECLARATION OF COMPONENTS

- Components receive data from their parents via `props`, which is similar to attributes of HTML tags.
- They may also receive fragments of tags from parents, using `<slot></slot>`

AN EXAMPLE OF SFC

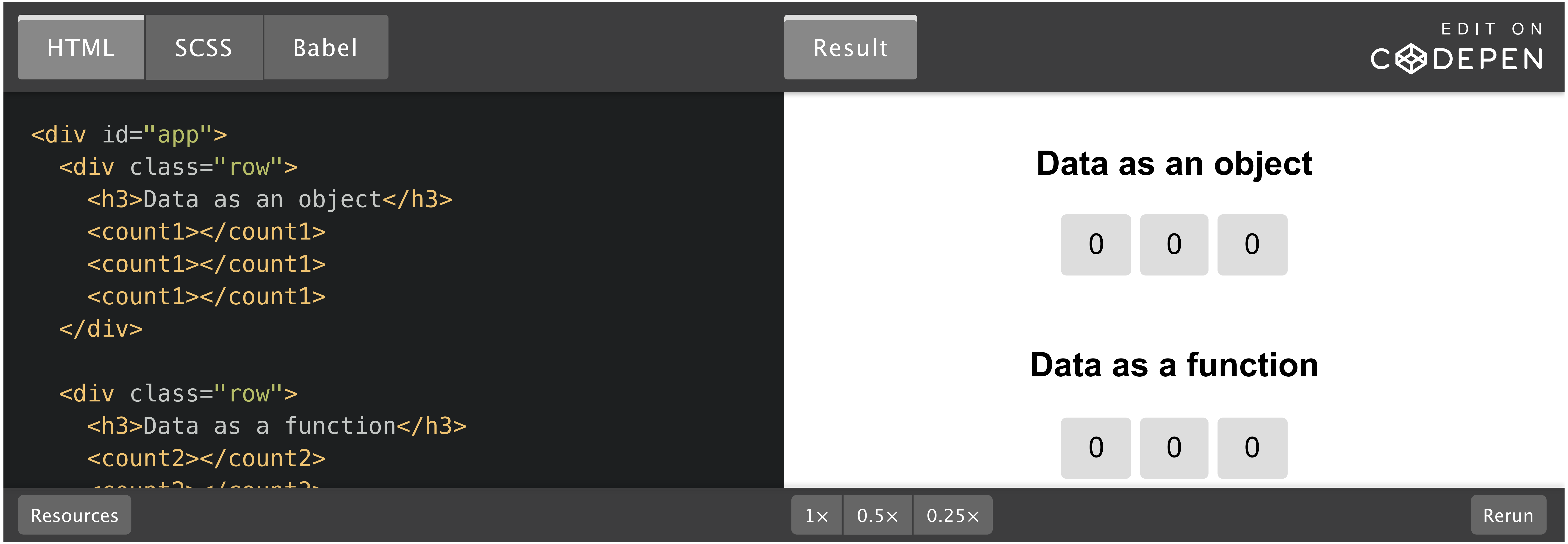
```
// hello-user.vue
<template>
  <span>Hello, {{ username }}</span>
</template>

<script>
export default {
  props: { username: String }
}
</script>
```

```
<template>
  <div><hello-user username="admin" /><!--you can also use v-bind:username--></div>
</template>
<script>
import HelloUser from './hello-user.vue'
export default {
  components: { HelloUser } // import and register the component
}
</script>
```

`data` should be a function in components

Different from using `new Vue` directly, because each component has its own isolated scope.

```
data: { a: 1 } // ❌ does not work as intended
data () { return { a: 1 } } // ✅ works
```



```
<div id="app">
  <div class="row">
    <h3>Data as an object</h3>
    <count1></count1>
    <count1></count1>
    <count1></count1>
  </div>

  <div class="row">
    <h3>Data as a function</h3>
    <count2></count2>
    <count2></count2>
```

Data as an object

0 0 0

Data as a function

0 0 0

Thanks @sdras for her example!

One-way data flow

> All props form a one-way-down binding between the child property and the parent one: when the parent property updates, it will flow down to the child, but not the other way around. This prevents child components from accidentally mutating the parent's state, which can make your app's data flow harder to understand. (from Vue.js documentation)

→ **Never, ever** write to your `` `props` `` like this:

```
<script>
export default {
  props: { username: String },
  methods: {
    reverseUsername () { // ❌❌❌ Vue will give a warning in console
      this.username = this.username.split('').reverse().join('')
    }
  }
}
</script>
```

instead, copy `` `username` `` before use in `` `data` ``, and use the copied value instead:

```
data () { // ✅
  return { usernameVal: this.username }
}
```

USING SLOTS

```
<navigation-link url="/profile">
  <font-awesome-icon name="user"></font-awesome-icon>
  Your Profile
</navigation-link>
```

and in `NavigationLink.vue`:

```
<a
  v-bind:href="url"
  class="nav-link"
>
  <slot></slot>
</a>
```

Upon rendering, the `<slot />` will be replaced with the icon and `Your Profile`.

COMPILATION SCOPE

> **Everything in the parent template is compiled in parent scope; everything in the child template is compiled in the child scope.**

```
<navigation-link url="/profile">
  Logged in as {{ user.name }} <!-- ✅ since user is defined in parent component -->
</navigation-link>
<navigation-link url="/profile">
  Clicking here will send you to: {{ url }} <!-- ❌ url is undefined -->
</navigation-link>
```

FALLBACK CONTENT

```
<a>
  <slot>Nothing provided. This is the fallback content!</slot>
</a>
```

NAMED SLOTS

A `` `<slot>` `` outlet without name implicitly has the name “default”.

```
<div class="container">
  <header> <slot name="header"></slot> </header>
  <main> <slot></slot> </main>
  <footer> <slot name="footer"></slot> </footer>
</div>
```

then use the `v-slot` directive on a `<template>` to provide some content in parent components:

```
<base-layout>
  <template v-slot:header>
    <h1>Here might be a page title</h1>
  </template>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template v-slot:footer>
    <p>Here's some contact info</p>
  </template>
</base-layout>
```

> You may also write:
>
> ```
> <template v-slot:default>
>   <p>A paragraph for the main content.</p>
>   <p>And another one.</p>
> </template>
> ```

A brief review on shorthands

Original Form	Shorthand form
`v-bind:value="var"`	`:value="var"`
`v-on:input="callback"`	`@input="callback"`
`v-slot:header`	`#header`

Custom events

In methods of Vue instance, you may use `` `this.$emit` `` to fire up custom events; this can be useful if you want to pass something from the component back to its parent.

```
this.$emit('change', this.val)
```

Nuxt.js

- Bundler is needed for Single File Components, because `` `.vue` `` files aren’t natively understood by browsers.
- Webpack + `` `vue-loader` `` is usually used.
  - Recall: loaders are used for source transformation, importing assets, etc.
- The official solution is `` `@vue/cli` ``, however you need to deal with `` `vue-router` `` and `` `vuex` `` all yourself.
- Nuxt.js handles routing and state management for you, with server-side rendering enabled and more

WHAT IS SERVER-SIDE RENDERING

- Some content is rendered on the server side, and the rendered version, along with page logic code, is sent to clients.
- Crucial for SEO because some crawlers cannot run JavaScript. Without SSR, they’ll crawl blank pages.

WHAT NUXT.JS CAN DO

- automatically generates the `` `vue-router` `` configuration
- server-side rendering & static sites
  - difference: for static sites, all pages are rendered at **build time**.
- better data fetching, other than the traditional approach using `` `mounted()` `` hook
  - fetch data with `` `asyncData(ctx)` `` or `` `fetch()` `` hook to get correct SSR results
- builtin loading progress bar support (also used by axios module)

CREATE A BASIC NUXT.JS PROJECT

Reference: https://nuxtjs.org/docs/2.x/get-started/installation

```
$ yarn create nuxt-app nuxt-example
$ cd nuxt-example
$ yarn dev
```

DIRECTORY STRUCTURE

If any of these folders is missing, create them.

- components: all your Vue.js components (SFCs) which are then imported into your pages.
- pages: application’s views. routes are generated automatically.
- assets: uncompiled assets such as your styles, images, or fonts.
- static: directly mapped to the server root and contains files that have to keep their names (e.g. `` `robots.txt` ``) or likely won’t change (e.g. the favicon)
- plugins: usually used for Nuxt.js plugins
- store: Vuex store files. Vuex is enabled only if `` `store/index.js` `` is present.
- nuxt.config.js: configuration for Nuxt.js

`context` in Nuxt.js

- a `context` per page load / router push
- contains router params, Vuex store, Nuxt.js contents, etc.

- `context` is different from Vue.js instance object (`this`)
  - read the docs carefully. In some hooks (like `asyncData`) only `context` can be used.
- `ctx.app` is the **root** Vue instance
- `ctx.store` is Vuex store instance
  - `ctx.store.state`, `ctx.store.dispatch`, `ctx.store.commit`
- ctx.route is `vue-router` instance
- `ctx.params`: router params, like `id` in `pages/posts/_id.vue`
  - alias of `ctx.route.params`
- `ctx.query`: router query, i.e. parsed query string
  - query string: the `?a=1&b=2` part of URL (note: it should be encoded)

- some components are also injected into `` `this` ``, but `` `this` `` is not always available
- `` `this.$route` `` by Vue Router
- `` `this.$store` `` by Vuex (if enabled)
- `` `this.$content` `` by Nuxt Content

Using Life-Cycle Hooks and Nuxt.js Context

VUE.JS LIFE CYCLE HOOKS

Vue.js projects usually make calls to API in `` `mounted()` `` hook

```
export default {
  data () {
    return { currentWeather: null }
  },
  async mounted () {
    const res = await fetch('https://example.com/weather.json')
    this.currentWeather = await res.json()
  }
}
```

For Nuxt.js projects, `` `asyncData(ctx)` `` and `` `async fetch()` `` is preferred, since they're designed for SSR.

```
<template>
  <p v-if="$fetchState.pending">Fetching mountains...</p>
  <p v-else-if="$fetchState.error">An error occurred :(</p>
  <div v-else>
    <h1>Nuxt Mountains</h1>
    <ul>
      <li v-for="mountain of mountains">{{ mountain.title }}</li>
    </ul>
    <button @click="$fetch">Refresh</button>
  </div>
</template>
```

```
export default {
  async asyncData({ params }) {  // gettings `params` from nuxt context
    const { data } = await axios.get(`https://my-api/posts/${params.id}`)
    return { title: data.title } // replaced the good old `data` method
  }
}
```

NUXT.JS LIFE CYCLE HOOKS

https://s3-ap-southeast-2.amazonaws.com/kruties-diagrams/nuxtjs/NuxtJs_Lifecycle_Hooks.pdf

Example project using fetch

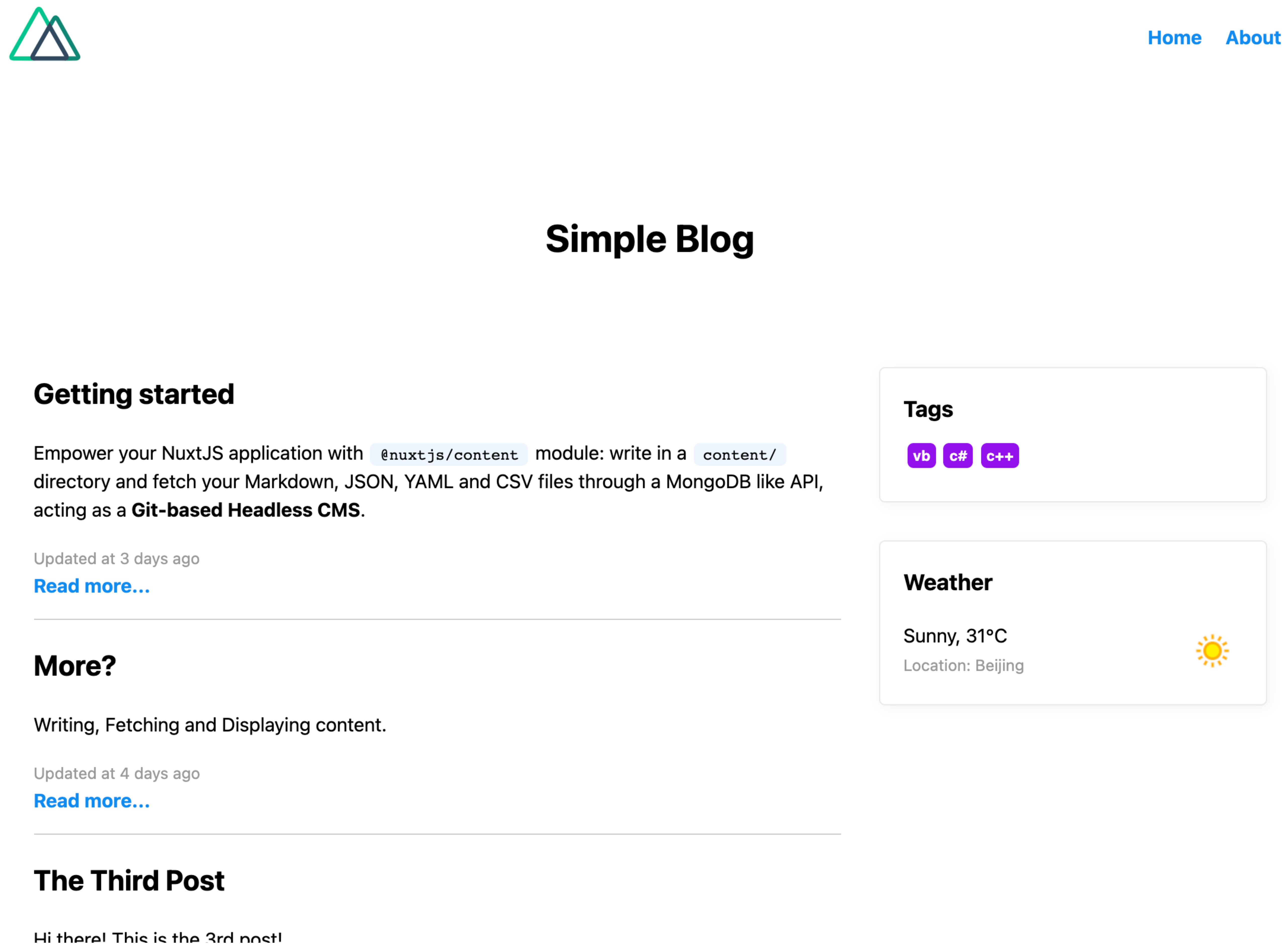
https://codesandbox.io/s/github/nuxtlabs/examples/tree/master/data-fetching/fetch-hook?from-embed

References

- Vue.js Official Document
- Nuxt.js Official Document, with in-depth explanation of internal structure
- Nuxt Axios Module
- Nuxt Content
  - Read the docs thoroughly before doing your homework

Homework

- Make your own static blog generator!
- Don’t worry, most of the code is written for you, you only need to fill in the blanks in the code.



Simple Blog

Getting started

Empower your NuxtJS application with `@nuxtjs/content` module: write in a `content/` directory and fetch your Markdown, JSON, YAML and CSV files through a MongoDB like API, acting as a **Git-based Headless CMS**.

Updated at 3 days ago

Read more...

More?

Writing, Fetching and Displaying content.

Updated at 4 days ago

Read more...

The Third Post

Hi there! This is the 3rd post!

Tags

vb c# c++

Weather

Sunny, 31°C ☀

Location: Beijing