Chapter 8

Managing application state

In the past, jQuery was a must-have library for any web project. It solved many problems by ensuring cross-browser compatibility and providing various useful methods for dealing with DOM, animations, AJAX, and more. It sped up development tremendously, but the code we were writing was imperative. When creating new elements and updating the DOM, we had to write code that was explicit about *how* things should be done. Since the release of ES2015, also known as ES6, JavaScript language evolved at a rapid pace, and new players appeared in front-end development, such as Angularjs, React, and Vue. These tools revolutionised how we write the code today, as we have switched from an imperative style to declarative. Instead of saying *how* something should be done, we define *what* should be done.

A lot of modern applications that use the tools mentioned above or any similar frameworks are *state-driven*. It means that when *state* changes, we don't have to deal with the DOM directly ourselves; instead, frameworks do it for us and update it accordingly. This paradigm shift, however, introduced another kind of a problem. How should state and business logic be managed and shared between different parts of the application? In this chapter, you will learn how to:

1. Share state between components by lifting it up to the lowest common

ancestor

2. Use an external Stateful Service and allow components to consume state directly

3. Use Composition API to create reusable composables and external reactive state to allow components to consume state directly

4. Use State Provider pattern to provide state to the whole application or only specific component tree

Vuex is not discussed in this chapter, as it has its own dedicated chapter, and is discussed in chapter 11.

If you would like to follow the code examples, you can scaffold a new Vue 3 app using Vue CLI.

8.1 Lifting up the state

Imagine you are working on a business card editor. You need to create a form that will allow users to upload their picture, and information such as name, description, phone number, and address. Let's start by creating two files in a newly scaffolded project.

views/businessCardEditor/BusinessCardEditor.vue

For now, the *BusinessCardEditor* component will import and render the *BusinessCardForm* component.

```
<template>
  <div class="p-8 container mx-auto grid grid-cols-2 gap-8">
    <BusinessCardForm />
  </div>
</template>

<script>
import BusinessCardForm from "./components/BusinessCardForm";
```

```
export default {
  components: {
    BusinessCardForm,
  },
};
</script>
```

views/businessCardEditor/components/BusinessCardForm.vue

It's a good practice to keep the state as close to where it belongs, so it does make sense to put it in the form component.

```
<template>
  <div class="shadow-md p-8">
    <h2 class="text-2xl font-semibold mb-4">Business Card Form</h2>
    <form>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Avatar</label>
        <input type="file" @change="onFileUpload" />
      </div>

      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Name</label>
        <input :class="$style.formInput" type="text" v-model="name" />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Description</label>
        <input :class="$style.formInput" type="text" v-model="description" />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Phone number</label>
        <input :class="$style.formInput" type="text" v-model="phoneNumber" />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Address</label>
        <input :class="$style.formInput" type="text" v-model="address" />
      </div>
    </form>
  </div>
</template>

<script>
export default {
  data() {
    return {
      avatarFile: null,
      name: "",
      phoneNumber: "",
      description: "",
      address: "",
    };
  },
  methods: {
    onFileUpload(e) {
      this.avatarFile = e.target.files?.[0];
    },
  },
};
</script>

<style module>
.formBlock {
  @apply flex flex-col mb-6;
}

.formLabel {
  @apply mb-3 font-semibold;
}

.formInput {
  @apply border border-gray-50 shadow p-4;
}
</style>
```

Let’s also update the routes config so we can access it.

router/index.js

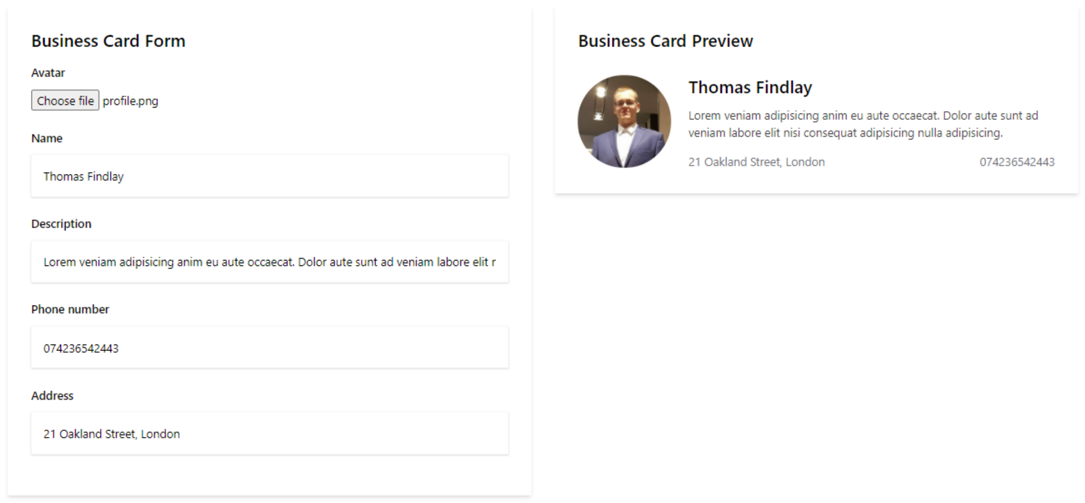
```
import { createRouter, createWebHistory } from "vue-router";
import Home from "@/views/Home";
import BusinessCardEditor from
        "../views/businessCardEditor/BusinessCardEditor.vue";

const routes = [
  {
    path: "/",
    name: "Home",
    component: Home,
  },
  {
    path: "/business-card-editor",
    name: "BusinessCardEditor",
    component: BusinessCardEditor,
  },
];
```

```
const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
});

export default router;
```

Great, we have a working form that is synchronised with the component state. You're happy that the functionality is done and are about to celebrate, but now your client is asking you to add a nice preview of this form that looks like a business card (Figure 8.1). The easiest solution would be to just add code for the preview in the *BusinessCardForm* component and be done with it. Unfortunately, the easiest solution is not always the best. This would not be a *BusinessCardForm* component anymore, but rather *BusinessCardFormWith-Preview* component. What if we would like to reuse this form somewhere else in the application, but without a preview? We could add a prop to indicate if we want to display the preview or not. The problem with this approach is that the more functionality we pack into components, the less reusable and maintainable they become. Therefore, instead of creating big, configurable components, it's better to create smaller components and compose them.



profile.png

Name

Thomas Findlay

Description

Lorem veniam adipisicing anim eu aute occaecat. Dolor aute sunt ad veniam labore elit r

Phone number

074236542443

Address

21 Oakland Street, London

Thomas Findlay

Lorem veniam adipisicing anim eu aute occaecat. Dolor aute sunt ad veniam labore elit nisi consequat adipisicing nulla adipisicing.

21 Oakland Street, London 074236542443

Figure 8.1: Business Card Editor

Let's create a new file for the preview component. It will receive data via props and display it.

views/product/businessCardEditor/components/BusinessCardPreview.vue

```
<template>
  <div>
    <div class="shadow-md p-8">
      <h2 class="text-2xl font-semibold mb-8">Business Card Preview</h2>
      <div class="flex">
        <div
          class="w-32 h-32 rounded-full bg-gray-100 mr-6
                 flex-shrink-0 overflow-hidden"
        >
          <img v-if="avatar" :src="avatar" alt="Avatar" />
        </div>
        <div class="flex flex-col flex-grow">
          <p class="font-semibold text-2xl mb-3">{{ name }}</p>
          <p class="text-gray-700 mb-4">{{ description }}</p>
          <div class="flex justify-between mt-auto">
            <p class="text-gray-500">{{ address }}</p>
            <p class="text-gray-500">{{ phoneNumber }}</p>
          </div>
        </div>
      </div>
    </div>
  </div>
</template>

<script>
export default {
  props: {
    avatar: String,
    name: String,
    phoneNumber: String,
    description: String,
    address: String,
  },
};
</script>
```

We also have to update the *BusinessCardForm* and *BusinessCardEditor* components.

views/businessCardEditor/components/BusinessCardForm.vue

Let's update the inputs. Instead of using the *v-model* directive, we need to add the *value* prop and *input* event listener, as we should not mutate props directly. To avoid unnecessary repetition, we will use a method that will $emit the appropriate event. It accepts *$event* and *field* arguments.

Template

```
<template>
  <div class="shadow-md p-8">
    <h2 class="text-2xl font-semibold mb-4">Business Card Form</h2>
    <form>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Avatar</label>
        <input
          type="file"
          @change="$emit('avatar-upload', $event)"
          aria-label="User Avatar"
        />
      </div>

      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Name</label>
        <input
          :class="$style.formInput"
          type="text"
          :value="name"
          @input="update($event, 'name')"
        />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Description</label>
        <input
          :class="$style.formInput"
          type="text"
          :value="description"
          @input="update($event, 'description')"
        />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Phone number</label>
        <input
          :class="$style.formInput"
          type="text"
          :value="phoneNumber"
          @input="update($event, 'phoneNumber')"
        />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Address</label>
        <input
          :class="$style.formInput"
          type="text"
          :value="address"
          @input="update($event, 'address')"
        />
      </div>
    </form>
  </div>
</template>
```

Script

```
<script>
export default {
  props: {
    name: String,
    description: String,
    phoneNumber: String,
    address: String,
  },
  emits: [
    "update:name",
    "update:description",
    "update:phoneNumber",
    "update:address",
    "avatar-upload",
  ],
  methods: {
    update(e, field) {
      this.$emit(`update:${field}`, e.target.value);
    },
  },
};
</script>
```

This example is for Vue 3, so we also have the *emits* array with a list of emitted events from the component. In Vue 2 you can skip that part.

views/businessCardEditor/BusinessCardEditor.vue

There are quite a few changes that we have to make. First, the business card form state needs to be placed here and then passed down to the *BusinessCard-Form* component. However, passing the values down is not enough, as we need to make sure that the values are also updated when a user enters something in the inputs. In Vue 3, we can use multiple *v-models* for that, as shown below. If you are using Vue 2, consider using the sync modifier.

Second, we have to import and register the *BusinessCardPreview* component and pass the required props. Considering that this component should display a preview of an image uploaded by a user, we need to render it to the screen. Thus, *onFileUpload* method, besides updating the state, will also use the *FileReader* API to create a base64 of the image uploaded.

Template

```
<template>
  <div class="p-8 container mx-auto grid grid-cols-2 gap-8">
    <BusinessCardForm
      v-model:name="name"
      v-model:phoneNumber="phoneNumber"
      v-model:description="description"
      v-model:address="address"
      @avatar-upload="onFileUpload"
    />
    <BusinessCardPreview
      :name="name"
      :phoneNumber="phoneNumber"
      :description="description"
      :address="address"
      :avatar="avatarPreview"
    />
  </div>
</template>
```

Script

```
<script>
import BusinessCardForm from "./components/BusinessCardForm";
import BusinessCardPreview from "./components/BusinessCardPreview";

export default {
  components: {
    BusinessCardForm,
    BusinessCardPreview,
  },
  data() {
    return {
      avatarFile: null,
      name: "",
      phoneNumber: "",
      description: "",
      address: "",
      avatarPreview: "",
    };
  },
  methods: {
    onFileUpload(e) {
      const file = e.target.files?.[0];
      this.avatarFile = file;

      // Reset preview if there is no file and bail out
      if (!file) {
        this.avatarPreview = "";
        return;
      }

      // Get Base64 for the avatar preview
      const reader = new FileReader();
      reader.addEventListener(
        "load",
        () => {
          this.avatarPreview = reader.result;
        },
        false
      );

      reader.readAsDataURL(file);
    },
  },
};
</script>
```

These are all the updates we needed. We have successfully lifted the state to the lowest common ancestor component. This is a great pattern and is very useful in many cases, but it is not a silver bullet. For instance, after a user logs in, you might want to store information related to the user and then access it in multiple places in your application. You could potentially store this data in the root component - *App.vue*, but you would have to pass this data via many layers of components that do not even need it. This approach, also known as *prop drilling*, would quickly clutter many components and make the application much harder to maintain. Fortunately, there are solutions to this problem, and we will cover them in the next sections.

Chapter 9

Managing application layouts

Consistent design is essential for any application, as it provides a better experience and is more predictable for users. Thus, pages and features might share the same layout pattern. For example, consider a dashboard application. If a user is logged in, they should see a dashboard layout (Figure 9.1). However, not logged in users should be redirected to a login or register page. Both of these would share an auth layout (Figure 9.2).

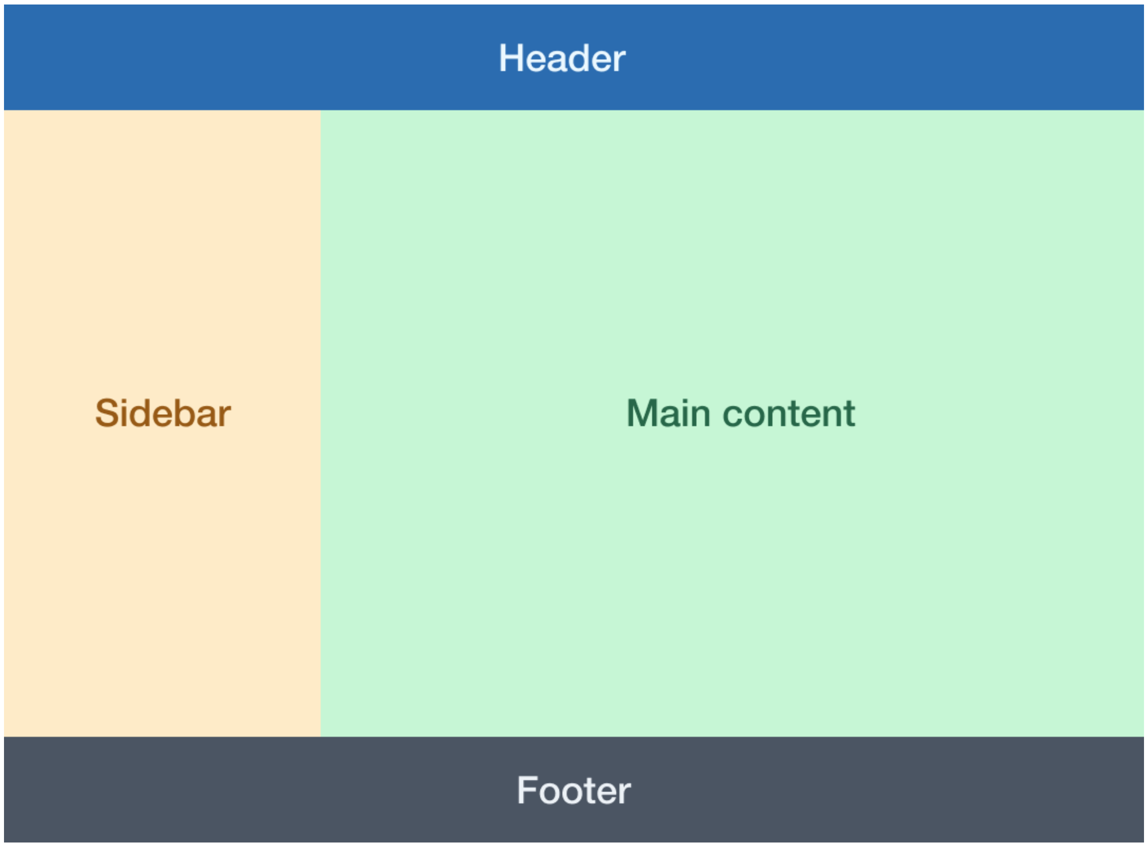


Figure 9.1: Standard layout

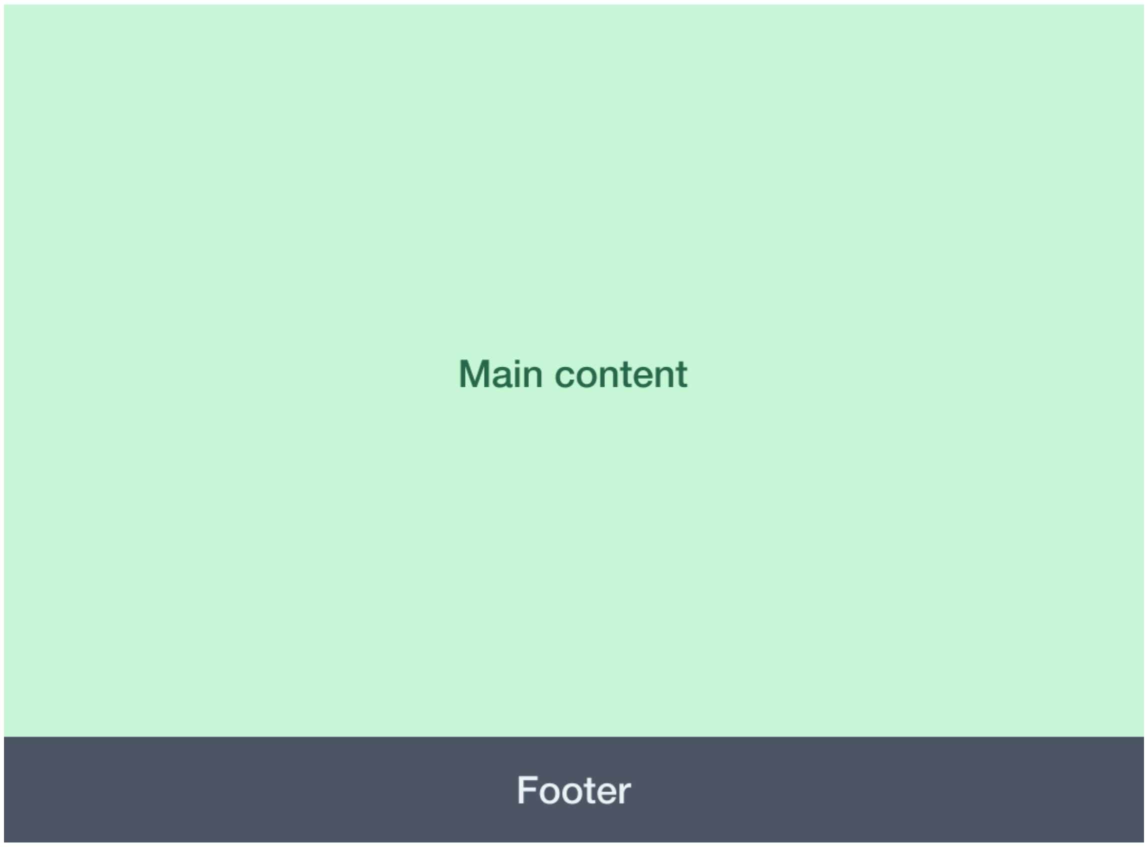


Figure 9.2: Auth layout

In this chapter, you will learn how to :

1. Manage layout via a route based meta config
2. Dynamically change the layout using a *LayoutService*
3. Dynamically change the layout using a *useLayout* composable
4. Create grid and list layouts for product cards
5. Create a *useLayoutFactory* composable to decouple layout state while using more than one layout

6. Create a *layoutFactory* to decouple layout template from layout components and dynamically inject *useLayout* composable

If you would like to follow the code examples, you can scaffold a new Vue 3 app using Vue-CLI.

9.1 Route-meta based page layout

There are different ways to specify which layout should be rendered. Let's start with layouts based on the *meta* property. These can be configured for each route defined for the vue-router. For this example, we will need the files listed below. All of them should be in the 'src' directory.

- router/index.js
- layout/Layout.vue
- layout/components/StandardLayout.vue
- layout/components/AuthLayout.vue
- views/LayoutExample.vue

For the purpose of demonstration, all the routes will be handled by the *LayoutExample.vue* component. Usually, you would have different paths and components that would render the same *Layout.vue* component inside. For instance, for */login* and */register* pages, the *Layout* component would render an *auth* layout, whilst */home* would have a standard layout.

router/index.js

For this example, we need three routes. The first default route will have no meta layout option specified, whilst the other two, */layout/standard* and */layout/auth* will have `layout: standard` and `layout: auth` respectively, as shown below.

```
import { createRouter, createWebHistory } from "vue-router";
import LayoutExample from "@/views/LayoutExample.vue";

const routes = [
  {
    path: "/",
    name: "LayoutExample",
    component: LayoutExample,
  },
  {
    path: "/layout/standard",
    name: "LayoutStandard",
    component: LayoutExample,
    meta: {
      layout: "standard",
    },
  },
  {
    path: "/layout/auth",
    name: "Home",
    component: LayoutExample,
    meta: {
      layout: "auth",
    },
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

layouts/Layout.vue

Next, let's take care of the *Layout* component. We know that it has to be able to render different components based on the route *meta* property. For that, we can use the `<component />`, and specify the component we want to render by providing `:is="componentToRender"` prop. In this case, it will be the *StandardLayout* or *AuthLayout* component. Besides that, we also need to forward all the slots so that the content can be rendered in correct places.

Template

```
<template>
  <!-- Render appropriate layout component -->
  <component :is="currentLayoutComponent">
    <!-- Pass through all the slots -->
    <template
      v-for="slotName in Object.keys($slots)"
      :key="slotName"
      v-slot:[slotName]="slotProps"
    >
      <slot :name="slotName" v-bind="slotProps" />
    </template>
  </component>
</template>
```

In this example, we have two layout components - *StandardLayout* and *AuthLayout*. They are both quite small, so they are not lazy-loaded, but if you have many complex layout components, you might want to consider lazy loading them. Lazy loading is covered in chapter 10. Importing layout components is not enough, as we also need to tell the `<component />`, which layout component should be rendered. Therefore, we need an object that will map the value specified in the *meta* object to the appropriate component, and a computed property to return the correct component. Any time the *meta.layout* value changes, the computed property will be re-evaluated. If there is no *layout* property specified on the *meta* object, then the standard layout is used as a fallback.

Script

```
<script>
import StandardLayout from './components/StandardLayout'
import AuthLayout from './components/AuthLayout'

const layoutComponents = {
  standard: StandardLayout,
  auth: AuthLayout,
};

export default {
  computed: {
    currentLayoutComponent() {
      const layout = this.$router.currentRoute.value?.meta?.layout || "standard";
      return layoutComponents[layout];
    },
  },
};
</script>
```

layout/components/StandardLayout.vue

The standard layout is supposed to have four sections - header, aside, content, and footer, as shown before in the figure 9.1. Therefore, we need to add four named slots, as well as some styling.

```
<template>
  <div class="container mx-auto" :class="$style.standardLayout">
    <header :class="$style.header">
      <slot name="header" />
    </header>
    <aside :class="$style.aside">
      <slot name="aside" />
    </aside>
    <main :class="$style.content">
      <slot name="content" />
    </main>
    <footer :class="$style.footer">
      <slot name="footer" />
    </footer>
  </div>
</template>

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

<style module>
.standardLayout {
  display: grid;
  grid-template-rows: 4rem 1fr 4rem;
  grid-template-columns: 1fr 4fr;
  height: 600px;
}

.header,
.aside,
.content,
.footer {
  display: grid;
  place-items: center;
}

.header {
  grid-column: span 2;
  @apply bg-green-200;
}

.aside {
  @apply bg-indigo-200;
}

.content {
  @apply bg-purple-300;
}

.footer {
  grid-column: span 2;
  @apply bg-gray-400;
}
</style>
```

layout/components/AuthLayout.vue

The *AuthLayout* component (Figure 9.2), similarly to the *StandardLayout* component, will use named slots, but only two of them - content and footer.

```
<template>
  <div class="container mx-auto" :class="$style.authLayout">
    <main :class="$style.content">
      <slot name="content" />
    </main>
    <footer :class="$style.footer">
      <slot name="footer" />
    </footer>
  </div>
</template>

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

<style module>
.authLayout {
  display: grid;
  grid-template-rows: 1fr 4rem;
  grid-template-columns: 1fr;
  height: 600px;
  @apply bg-purple-300;
}
```

```
.content,
.footer {
  display: grid;
  place-items: center;
}

.content {
  grid-area: 1 / 1 / 2 / 2;
  @apply bg-purple-300;
}

.footer {
  grid-area: 2 / 1 / 3 / 2;
  @apply bg-gray-400;
}
</style>
```

LayoutExample.vue

Finally, it's time to put all the pieces together. We are going to need two *router-links* to switch between standard and auth routes. Besides that, we will utilise the *Layout* component and provide custom content via named slots. Note that when the auth layout is activated, the header's content and aside slots are ignored.

```
<template>
  <div>
    <div class="space-x-4 mb-8 mx-auto flex justify-center items-center mt-4">
      <router-link to="/layout/standard">Layout standard</router-link>
      <router-link to="/layout/auth">Layout auth</router-link>
    </div>
    <Layout class="mx-auto max-w-7xl">
      <template #header>
        <p>Header</p>
      </template>
      <template #content>
        <p>Content</p>
      </template>
      <template #aside>
        <p>Aside</p>
      </template>
      <template #footer>
        <p>Footer</p>
      </template>
    </Layout>
  </div>
```

```
</template>
<script>
import Layout from "../layout/Layout";
export default {
  components: {
    Layout,
  },
};
</script>
```

A different layout component will now be displayed at any time the route changes. However, what if we want to have more control over the layout, and change it dynamically? Let's have a look at how we can accomplish that.

9.2 Dynamic layout with a pageLayoutService

In the previous example, we have derived information from the route meta object to display the appropriate layout component. This time we won't be relying anymore on the *layout/standard* and *layout/auth* paths defined in the *router.js* file. Instead, we are going to have a reactive state in a service file. As it is directly related to the layout functionality, we will not put it in the global *services* folder, but instead in the *layout* directory, so the file path will be *src/layout/services/pageLayoutService.js*.

For a usable layout service we need to be able to do 3 things:

1. Reactively store the currently selected layout, so all consumers can re-render and show appropriate layout component
2. Allow components to consume currently selected layout type
3. Provide a way to update the layout type

The first step we can accomplish by using the *reactive* method. If you are developing for Vue 2, then instead of *reactive* you can use the *observable* method,