

Routing



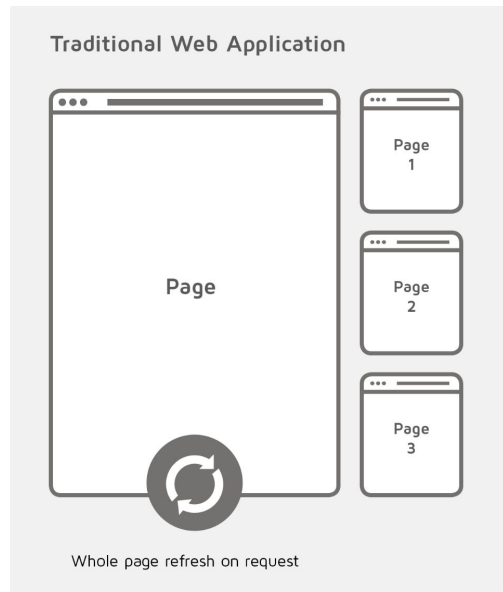
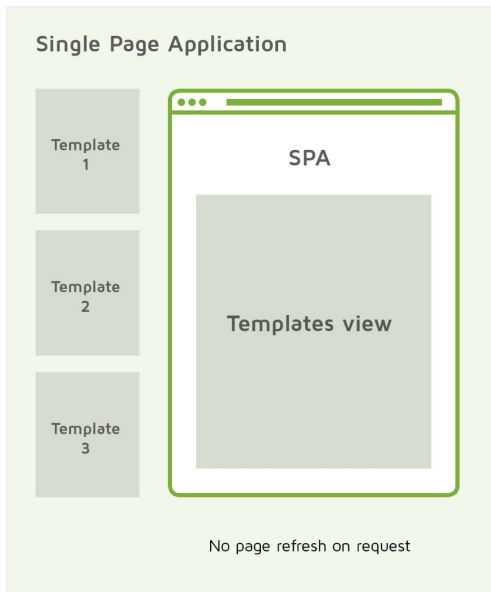
Vue Router

Vue Router - официальный router (маршрутизатор) для Vue.js.

Именно **Router** обеспечивает работу SPA (Single Page Application).

Vue Router с помощью HTML5 history API делает так, что адрес в адресной строке браузера меняется без фактического обновления страницы.

Также **Vue Router** сопоставляет url в адресной строке браузера и компонент страницы приложения, который сейчас должен отображаться на экране.



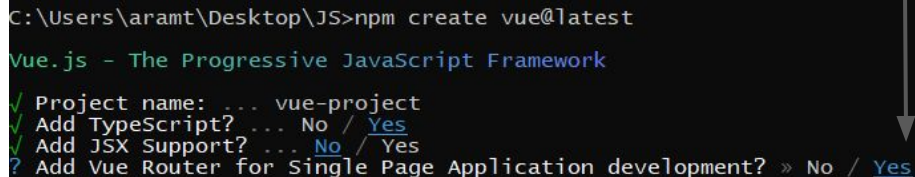
Installing

Установить Vue Router можно, как обычный пакет npm:

```
npm install vue-router@4
```

Также при создании нового проекта с помощью *create-vue*, когда утилита будет задавать вопросы о создании нового проекта, можно сразу выбрать опцию роутера:

```
npm create vue@latest
```



A terminal window showing the command `npm create vue@latest` being executed. The output shows the `create-vue` utility prompts for project name, TypeScript, JSX support, and Vue Router. The user selects 'vue-project' for the name, 'No' for TypeScript, 'No' for JSX, and 'Yes' for Vue Router. A grey arrow points from the `npm create vue@latest` command in the block above to the 'Yes' selection in the terminal.

```
C:\Users\aramt\Desktop\JS>npm create vue@latest
Vue.js - The Progressive JavaScript Framework
✓ Project name: ... vue-project
✓ Add TypeScript? ... No / Yes
✓ Add JSX Support? ... No / Yes
? Add Vue Router for Single Page Application development? » No / Yes
```

Setup

Для инициализации Vue Router используется функция `createRouter`. В нее мы передаем объект `history` и массив `routes`, где указываем связь между url-ами и нужными компонентами.

Созданный `router` нужно подключить к приложению `app` с помощью метода `app.use`

router/index.ts

```
import { createRouter, createWebHistory } from 'vue-router'
import HomeView from '../views/HomeView.vue'
import AboutView from '../views/AboutView.vue'

const router = createRouter({
  history: createWebHistory('/'),
  routes: [
    {
      path: '/',
      component: HomeView
    },
    {
      path: '/about',
      component: AboutView
    }
  ]
})

export default router
```

views/HomeView.vue

```
<template>
  <div class="home">
    <h1>Home</h1>
    <p>Home page content</p>
  </div>
</template>
```

views/AboutView.vue

```
<template>
  <div class="about">
    <h1>About</h1>
    <p>About page content</p>
  </div>
</template>
```

main.ts:

```
import { createApp } from 'vue'

import App from './App.vue'
import router from './router'

const app = createApp(App)

app.use(router)

app.mount('#app')
```

RouterLink and RouterView

После инициализации и подключения роутера мы можем вставить компонент, соответствующий текущему url с помощью элемента `<RouterView />`.

Для создания же ссылок для роутера используется элемент `RouterLink`. В его проп `to` мы передаем адрес ссылки.

App.vue:

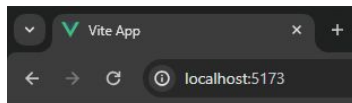
```
<script setup lang="ts">
import { RouterLink, RouterView } from 'vue-router'
</script>

<template>
  <header>
    <RouterLink to="/">Home</RouterLink> <br>
    <RouterLink to="/about">About</RouterLink>
  </header>

  <RouterView />
</template>
```

В отличие от обычных ссылок `<a>`, ссылки `RouterLink` созданы специально для роутера и они будут выполнять переадресацию и менять url без обновления страницы.

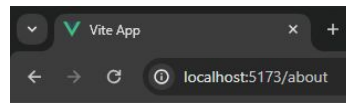
При изменении url роутер будет автоматически обновлять содержимое `RouterView` и подставлять в него нужный нам компонент:



[Home](#)
[About](#)

Home

Home page content



[Home](#)
[About](#)

About

About page content

Dynamic Route Matching with Params

Очень часто нам нужно будет сопоставлять пути определенного вида с одним и тем же компонентом. Например, у нас может быть `User` компонент, который должен отображаться для всех пользователей, но с разными id пользователей. В Vue Router мы можем использовать динамический сегмент в пути для достижения этой цели.

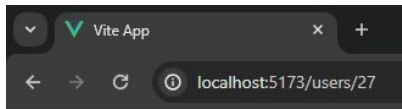
router/index.ts

```
// these are passed to `createRouter`  
const routes = [  
  // dynamic segments start with a colon  
  { path: '/users/:id', component: User },  
]
```

views/User.vue

```
<script setup lang="ts">  
import { useRoute } from 'vue-router';  
  
const route = useRoute()  
const userId = route.params.id  
console.log(`User id: ${userId}`)  
</script>  
  
<template>  
  <div>  
    User id: {{ $route.params.id }}  
  </div>  
</template>
```

Внутри целевого компонента мы можем получить нужный нам текущий url-параметр с помощью `useRoute` в коде или `$router` в шаблоне.



[Home](#)
[About](#)
User id: 27

Regex in params and Optional parameters

Стандартный плейсхолдер параметра вида “:userId” эквивалентен следующему регулярному выражению:

([^/]+) - т.е. по крайней мере, один символ, который не является “/”

Однако мы можем явно задать нужное нам регулярное выражение после имени параметра:

```
const routes = [  
  // /:userId -> matches only numbers  
  { path: '/:userId(\\d+)' },  
  // /:userName -> matches anything else  
  { path: '/:userName' },  
]
```

Удовлетворяет числам

Удовлетворяет всему остальному

(для упрощения здесь для роутов приведено только свойство path)

(Роуты с регулярными выражениями имеют более высокий приоритет, поэтому даже если в данном примере мы поменяем роуты местами, то численные пути всё равно будут активировать нужный роут.)

```
const routes = [  
  // will match /users and /users/posva  
  { path: '/users/:userId?' },  
  // will match /users and /users/42  
  { path: '/users/:userId(\\d+)?' },  
]
```

Мы также можем пометить параметр как необязательный, используя модификатор “?” (0 или 1)

Repeatable params

Если нужно сопоставить роуты с несколькими разделами, такими как `/first/second/third`, вы должны пометить параметр как повторяемый с помощью “*” (0 или более) и “+” (1 или более):

```
const routes = [  
  // /:chapters -> matches /one, /one/two, /one/two/three, etc  
  { path: '/:chapters+' },  
  // /:chapters -> matches /, /one, /one/two, /one/two/three, etc  
  { path: '/:chapters*' },  
]
```

“*” и “+” также можно объединить с пользовательским регулярным выражением, добавив их после закрывающих круглых скобок:

```
const routes = [  
  // only match numbers  
  // matches /1, /1/2, etc  
  { path: '/:chapters(\\d+)+' },  
  // matches /, /1, /1/2, etc  
  { path: '/:chapters(\\d+)*' },  
]
```


Sensitive and strict route options

По умолчанию роуты не учитывают регистр и соответствуют путям с косой чертой в конце или без нее. Например, путь `/users` соответствует `/users`, `/users/` и даже `/Users/`. Это поведение можно настроить с помощью параметров `strict` и `sensitive`, они могут быть установлены как на уровне конкретного роута, так и на уровне всего роутера:

```
const router = createRouter({
  history: createWebHistory(),
  routes: [
    { path: '/users/:id', sensitive: true },
    { path: '/users/:id?' }
  ],
  strict: true
})
```

будет соответствовать `/users/posva`, но не:
`/users/posva/` из-за `strict: true`
`/Users/posva` из-за `sensitive: true`

будет соответствовать `/users`, `/Users` и
`/users/42`, но не `/users/` или `/users/42/`

применяется ко всем роутам

Nested Routes

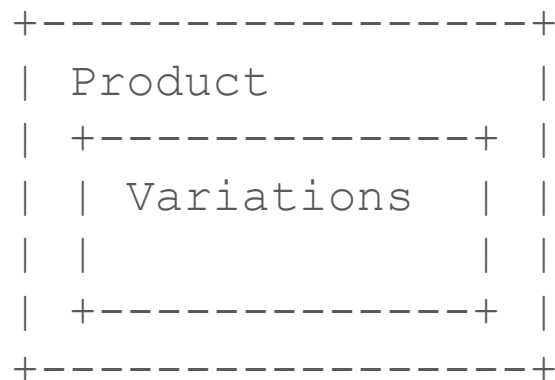
Пользовательские интерфейсы некоторых приложений состоят из компонентов, вложенных на несколько уровней вглубь. В этом случае очень часто сегменты URL соответствуют определенной структуре вложенных компонентов, например:

/product/1/reviews



+----->

/product/1/variations



Nested Routes

Для этого используется свойство `children` у роута:

router/index.ts:

```
const router = createRouter({
  //...
  routes: [
    {
      path: '/',
      //...
    },
    {
      path: '/contact',
      //...
    },
    {
      path: '/product/:id',
      name: 'product',
      component: ProductView,
      children: [
        {
          path: 'reviews',
          component: ProductReviews
        },
        {
          path: 'variations',
          component: ProductVariations
        }
      ]
    }
  ]
})
```

product/1

product/1/reviews

product/1/variations

views/ProductView.vue:

```
<template>
  <h1>{{ product.name }}</h1>
  ...
  <RouterView></RouterView>
</template>
```

components/ProductReviews.vue:

```
<template>
  <h2>Reviews</h2>
  ...
</template>
```

components/ProductVariations.vue:

```
<template>
  <h2>Variations</h2>
  ...
</template>
```

В компоненте родительского роута нужно поместить дополнительный **RouterView**. На его месте будут рендериться компоненты дочерних роутов.

Programmatic Navigation

Vue Router позволяет совершить перенаправление программно. Для этого используется метод `push` объекта `router`.

```
<script setup>
import { useRouter } from 'vue-router'

const router = useRouter()

const goToProductList = () => {
  router.push('/product-list')
}
</script>
```

получаем объект роутера с помощью `useRouter`


совершаем программное перенаправление с помощью `router.push`

Named Routes

Именованные маршруты в Vue Router - это маршруты, которым присвоено уникальное имя. Они предоставляют альтернативный способ сослаться на роуты при генерации URL-адресов или перенаправлении, вместо хардкода путей в приложении.

```
const routes = [  
  { path: '/dashboard/user/:id', component: User, name: 'userProfile' }  
]
```

даём роуту уникальное имя
через свойство name



Теперь в RouterLink и router.push можем сослаться на роут по имени, а path параметры в таком случае передаются в объекте params:

```
<router-link :to="{ name: 'userProfile', params: { id: 42 } }">User Profile</router-link>
```

```
import { useRouter } from 'vue-router'  
  
const router = useRouter()  
  
const navigateUserProfile = (id) => {  
  router.push({ name: 'userProfile', params: { id } })  
}
```

Если потом решите изменить путь роута, например, с “/dashboard/user/:id” на “/profile/:id”, то не придется менять его во всех ссылках RouterLink и переходах по router.push

Named Views

Иногда нужно отобразить несколько компонентов одновременно, вместо того чтобы вкладывать их друг в друга, например, создать макет с видом боковой панели и основным видом. Для этого пригодятся **Named Views**. Вместо того, чтобы иметь в своем представлении один RouterView, мы можем иметь несколько и присвоить каждому из них имя. RouterView без имени будет присвоено имя default.

```
<router-view class="view left-sidebar" name="LeftSidebar"></router-view>
<router-view class="view main-content"></router-view>
<router-view class="view right-sidebar" name="RightSidebar"></router-view>
```

```
const router = createRouter({
  history: createWebHashHistory(),
  routes: [
    {
      path: '/',
      components: {
        default: Home,
        // short for LeftSidebar: LeftSidebar
        LeftSidebar,
        // they match the `name` attribute on `<router-view>`
        RightSidebar,
      },
    },
  ],
})
```

для задания нескольких компонентов в роуте используется свойство components (вместо component)

в components помещается объект, где перечисляются компоненты для разных RouterView

Redirect

В конфигурации роутов также можно задать перенаправление.

```
const routes = [{ path: '/home', redirect: '/' }]
```

← перенаправление по фиксированному урлу

```
const routes = [{ path: '/home', redirect: { name: 'homepage' } }]
```

← перенаправление по именному роуту

```
const routes = [{ path: '/home', redirect: to => ({ path: '/product-list' }) }]
```

← перенаправление с помощью функции

```
const routes = [{ path: '/home', redirect: 'posts' }]
```

```
const routes = [{ path: '/home', redirect: { path: 'posts' } }]
```

← перенаправление по относительному пути
(начинается без "/")

Alias

Перенаправление означает, что когда пользователь посещает `/home`, URL-адрес будет заменен на `/`, а затем сопоставлен как `/`. Но что такое псевдоним (**alias**)?

Псевдоним `/` в `/home` означает, что когда пользователь посещает `/home`, URL-адрес остается `/home`, но он будет сопоставлен так, как если бы пользователь посещал `/`.

```
const routes = [{ path: '/', component: Homepage, alias: '/home' }]
```

← Homepage будет отображен на:
“/” и “/home”

В данном случае `UserList` будет отображен на:
“/users”, “/users/list”, “/people”

```
const routes = [
  {
    path: '/users',
    component: UsersLayout,
    children: [
      { path: '', component: UserList, alias: ['/people', 'list'] },
    ],
  },
]
```

В данном случае `UserDetails` будет отображен на:
“/users/24”, “/users/24/profile”, “/24”

```
const routes = [
  {
    path: '/users/:id',
    component: UsersByIdLayout,
    children: [
      { path: 'profile', component: UserDetails, alias: [':id', ''] },
    ],
  },
]
```

(Если у вашего роута есть параметры, обязательно укажите их в любом абсолютном псевдониме)

Passing Props

Как мы видели ранее, path параметры можно получить внутри компонента с помощью `useRoute` и обращению к `route.params`. Однако Vue позволяет получать их с помощью пропсов. Вот пример, как изменится код в этом случае:

```
<script setup lang="ts">
import { useRoute } from 'vue-router';

const route = useRoute()
console.log(`User: ${route.params.id}`)
</script>

<template>
  <div class="user">User: {{ $route.params.id }}</div>
</template>
```

```
<script setup lang="ts">
const props = defineProps(['id'])

console.log(`User: ${props.id}`)
</script>

<template>
  <div class="user">User: {{ id }}</div>
</template>
```

```
const routes = [
  {
    path: '/user/:id',
    component: User
  }
]
```

```
const routes = [
  {
    path: '/user/:id',
    component: User,
    props: true
  }
]
```

Если props имеет значение true, то `route.params` будет установлен в качестве props компонента.

History modes

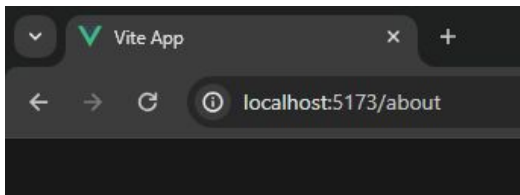
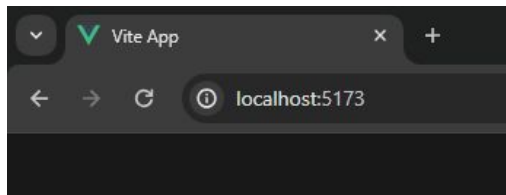
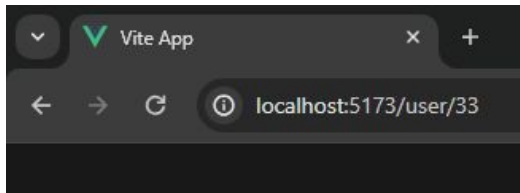
Как можно было заменить ранее, при создании роутера требуется указать объект `history`.

Существует два основных метода для этого: `createWebHistory` и `createWebHashHistory` (есть ещё также `createMemoryHistory`, но он не предназначен для браузерной среды).

`createWebHistory` (он же **HTML5 mode**) - является рекомендуемым и обеспечивает нормальный вид URL в адресной строке.

```
const router = createRouter({
  history: createWebHistory(),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView
    },
    {
      path: '/about',
      name: 'about',
      component: AboutView
    },
    {
      path: '/user/:id',
      component: User
    }
  ]
})
```

Использование `createWebHistory` рекомендовано и даёт стандартный вид URL в адресной строке:



History modes

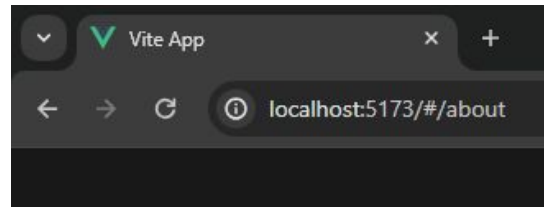
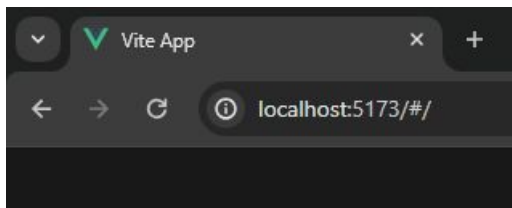
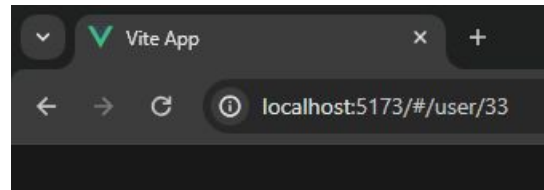
Но стандартный вид URL, обеспечиваемый `createWebHistory` имеет один недостаток: он требует [настройки сервера](#) на перенаправление ошибки **404** на `index.html`, где URL уже подхватит клиентский роутинг (Vue Router).

Если подобное не настроить, то http сервер, получая путь типа `"/user/33"` будет пытаться найти у себя папку `"users"`, в которой есть папка `"33"`, где лежит `index.html` (т.е. `/user/33/index.html`), и не найдя этого файла, сервер вернет ошибку 404.

Поэтому Vue обеспечивает альтернативный режим работы роутинга - **Hash Mode**, при котором `history` создается с помощью метода `createWebHashHistory`. Этот метод добавляет знак `"#"` перед настоящим путем, делая его хэшем, который не оказывает влияния на сервер и не требует его отдельной настройки.

```
const router = createRouter({
  history: createWebHashHistory(),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView
    },
    {
      path: '/about',
      name: 'about',
      component: AboutView
    },
    {
      path: '/user/:id',
      component: User
    }
  ]
})
```

Использование `createWebHashHistory` ставит `"#"` перед адресом, делая его просто хэшем, не влияющим на серверный роутинг



Route Meta Fields

Иногда может быть полезно прикрепить произвольную информацию к роутам, например: роли, чтобы контролировать, кто может получить доступ к маршруту, и т.д. Этого можно достичь с помощью свойства **meta**, которое принимает объект свойств и может быть доступно позднее через `route.meta`. Можно определить метасвойства следующим образом:

```
const routes = [
  {
    path: '/posts',
    component: PostsLayout,
    children: [
      {
        path: 'new',
        component: PostsNew,
        // only authenticated users can create posts
        meta: { requiresAuth: true },
      },
      {
        path: ':id',
        component: PostsDetail,
        // anybody can read a post
        meta: { requiresAuth: false },
      },
    ],
  },
]
```

добавляем мета-информацию о доступности роута

Если задать meta для родительского и для дочернего роута одновременно, то **\$route.meta** будет объединением родительских и дочерних meta-полей

Navigation Guards

Далее, чтобы ограничить доступ к роуту на основе указанной мета-информации применяются перехватчики **Navigation Guards**. Они могут отменить либо перенаправить переход на защищенный роут.

Один из самых популярных перехватчиков `beforeEach`. Он принимает callback обработки редиректа. Данный callback вызывается перед любым редиректом и получает объекты конечного (`to`) и начального роутов (`from`) со всей их мета-информацией. Данный метод может вернуть `false` (чтобы отменить переход), либо вернуть объект `Route Location` (чтобы совершить редирект).

```
const router = createRouter({ ... })

router.beforeEach((to, from) => {
  if (to.meta.requiresAuth && !auth.isLoggedIn()) {
    // this route requires auth, check if logged in
    // if not, redirect to login page.
    return {
      path: '/login',
      // save the location we were at to come back later
      query: { redirect: to.fullPath },
    }
  }
})
```

Вот как выглядит Navigation Guard для переадресации неавторизованных пользователей на страницу входа.

(Если страница требует авторизации, а пользователь не авторизован, то выполняем переадресацию на login страницу и сохраняем query параметр redirect, чтобы после входа, знать, куда изначально хотел попасть пользователь.)

Существуют также и другие перехватчики, напр. `afterEach` и `beforeResolve`. Более подробно с ними можно ознакомиться [в документации](#).

Routing Hooks

Иногда при переходе на новый маршрут требуется получить данные с сервера. Например, перед отображением профиля пользователя необходимо получить данные о нем с сервера. Мы можем добиться этого двумя разными способами:

Первый способ - **Fetching после навигации**: сначала выполнить навигацию, а затем получить данные в хуке жизненного цикла входящего компонента (напр. в `onMounted` или просто в `script setup`). Во время получения данных отображается состояние загрузки. Пример:

components/Post.vue (template):

```
<template>
  <div class="post">
    <div v-if="loading" class="loading">Loading...</div>

    <div v-if="error" class="error">{{ error }}</div>

    <div v-if="post" class="content">
      <h2>{{ post.title }}</h2>
      <p>{{ post.body }}</p>
    </div>
  </div>
</template>
```

components/Post.vue (script setup):

```
<script setup>
import { ref, watch, onMounted } from 'vue'
import { useRoute } from 'vue-router'

const loading = ref(false)
const post = ref(null)
const error = ref(null)

const route = useRoute()

onMounted(() => {
  // Watch the params of the route to fetch the data again
  watch(
    () => route.params,
    () => {
      fetchData()
    },
    // Fetch the data when the view is created and the data is
    // already being observed
    { immediate: true }
  )
})

function fetchData() {
  error.value = post.value = null
  loading.value = true

  // Replace `getPost` with your data fetching util / API wrapper
  getPost(route.params.id, (err, fetchedPost) => {
    loading.value = false
    if (err) {
      error.value = err.toString()
    } else {
      post.value = fetchedPost
    }
  })
}
</script>
```

Routing Hooks

Второй способ - **Fetching перед навигацией**:
получаем данные до перехода к новому маршруту.

Мы можем выполнить fetch данных в компоненте в хуке `beforeRouteEnter` и вызвать `next` только после завершения загрузки данных. Callback, переданный в `next`, будет вызван после mounting компонента.

Хук `beforeRouteEnter` не реализован в Composition API script setup, поэтому код приведен для Options API.

Пользователь будет оставаться на предыдущем view, пока происходит fetch ресурсов для нового view. Поэтому рекомендуется отображать прогресс-бар или какой-либо индикатор во время получения данных.

components/Post.vue (script)

```
<script>
export default {
  data() {
    return {
      post: null,
      error: null,
    }
  },
  beforeRouteEnter(to, from, next) {
    getPost(to.params.id, (err, post) => {
      // `setData` is a method defined below
      next(vm => vm.setData(err, post))
    })
  },
  // when route changes and this component is already rendered,
  // the logic will be slightly different.
  async beforeRouteUpdate(to, from) {
    this.post = null
    try {
      this.post = await getPost(to.params.id)
    } catch (error) {
      this.error = error.toString()
    }
  },
  methods: {
    setData(error, post) {
      if (error) {
        this.error = error
      } else {
        this.post = post
      }
    }
  }
}
</script>
```

Transitions

Как вы помните, `<Transition>` используется для применения анимации при добавлении или удалении элемента или компонента из DOM. Для того чтобы использовать `<Transition>` в компонентах роута и анимировать навигацию, необходимо использовать слот `<RouterView>`:

```
<router-view v-slot="{ Component, route }">
  <!-- Use a custom transition or fallback to `fade` -->
  <transition :name="route.meta.transition || 'fade'">
    <component :is="Component" />
  </transition>
</router-view>
```

Тип transition храним в мета-информации, которая может задаваться, как статически, так и динамически

```
router.afterEach((to, from) => {
  const toDepth = to.path.split('/').length
  const fromDepth = from.path.split('/').length
  to.meta.transition = toDepth < fromDepth ? 'slide-right' : 'slide-left'
})
```

```
const routes = [
  {
    path: '/custom-transition',
    component: PanelLeft,
    meta: { transition: 'slide-left' },
  },
  {
    path: '/other-transition',
    component: PanelRight,
    meta: { transition: 'slide-right' },
  },
]
```


Scroll Behavior

При использовании маршрутизации на стороне клиента мы можем захотеть прокручивать страницу к верху при переходе к новому маршруту или сохранять позицию прокрутки записей history так же, как это делает перезагрузка страницы. Vue Router позволяет настроить поведение прокрутки при навигации по маршруту.

Настройка осуществляется с помощью задания метода `scrollBehavior`, который получает объекты маршрута `to` и `from`, как в Navigation Guard. Третий аргумент, `savedPosition`, доступен только в том случае, если это popstate-навигация (запускается кнопками браузера "назад/вперед"). `scrollBehavior` может возвращать объект позиции `ScrollToOptions`.

Переход к сохраненной позиции (если она есть), либо к верху страницы:

```
const router = createRouter({
  //...
  scrollBehavior(to, from, savedPosition) {
    if (savedPosition) {
      return savedPosition
    } else {
      return { top: 0 }
    }
  },
})
```

Переход всегда к позиции на 10px выше элемента #main

```
const router = createRouter({
  //...
  scrollBehavior(to, from, savedPosition) {
    // always scroll 10px above the element #main
    return {
      el: '#main',
      top: 10
    }
  },
})
```

Плавный переход к позиции, указанной в хэше

```
const router = createRouter({
  //...
  scrollBehavior(to, from, savedPosition) {
    if (to.hash) {
      return {
        el: to.hash,
        behavior: 'smooth',
      }
    }
  },
})
```

Lazy Loading

При создании приложений с помощью сборщика бандл JavaScript может стать довольно большим и, как следствие, повлиять на время загрузки страницы. Было бы эффективнее разделить компоненты каждого маршрута на отдельные куски и загружать их только при посещении маршрута.

Vue Router поддерживает динамический импорт из коробки, то есть можно заменить статический импорт на динамический:

```
// replace
// import UserDetails from './views/UserDetails'
// with
const UserDetails = () => import('./views/UserDetails.vue')

const router = createRouter({
  // ...
  routes: [
    { path: '/users/:id', component: UserDetails }
  ],
})
```

← динамический (Lazy) импорт

также можно импортировать напрямую в объекте роута:

```
{ path: '/users/:id', component: () => import('./views/UserDetails.vue') }
```

Navigation Failures

При использовании `router-link` Vue Router вызывает `router.push` для запуска навигации. Хотя ожидаемым поведением большинства ссылок является переход пользователя на новую страницу, есть несколько ситуаций, когда пользователи остаются на той же странице:

- Пользователь уже находится на странице, на которую он пытается перейти.
- Navigation Guard прерывает навигацию, выполняя `return false`.
- Новый Navigation Guard выполняется, пока предыдущий не завершен.
- Navigation Guard перенаправляет в другое место, возвращая новое местоположение (например, `return '/login'`).
- Navigation Guard выбрасывает ошибку.

Если мы хотим сделать что-то после завершения навигации, мы должны подождать после вызова `router.push`, используя `await`

```
const navigationResult = await router.push('/my-profile')  
  
if (navigationResult) {  
  // navigation prevented  
} else {  
  // navigation succeeded (this includes the case of a redirection)  
}
```

навигация - асинхронная операция, поэтому мы должны поставить `await` перед `router.push`, если мы хотим получить результат навигации

Navigation Failures


Результат навигации - это экземпляры `Error` с несколькими дополнительными свойствами, которые дают нам достаточно информации, чтобы понять, какая навигация была предотвращена и почему. Чтобы проверить характер результата навигации, используется функция `isNavigationFailure`:

```
import { NavigationFailureType, isNavigationFailure } from 'vue-router'

// trying to leave the editing page of an article without saving
const failure = await router.push('/articles/2')

if (isNavigationFailure(failure, NavigationFailureType.aborted)) {
  // show a small notification to the user
  showToast('You have unsaved changes, discard and leave anyway?')
}
```

Navigation Guard `afterEach` принимает третьим параметром объект ошибки (результата навигации)



```
router.afterEach((to, from, failure) => {
  if (failure) {
    sendToAnalytics(to, from, failure)
  }
})
```

Существует 3 типа возможных ошибок навигации, которые мы и можем проверить с помощью `isNavigationFailure`

- `aborted`: внутри Navigation Guard возвращено `false`.
- `cancelled`: Новая навигация произошла до того, как текущая навигация успела завершиться. Например, вызов `router.push` произошел во время ожидания внутри navigation guard.
- `duplicated`: Навигация была предотвращена, так как мы уже находимся в нужном месте.