GeekBrains

Навигация в приложении. Vue-Router

Vue.js

Оглавление

Архитектурные паттерны навигации.

В рамках сегодняшнего урока мы с вами улучшим наше приложение для учета финансов. Настало время роста, добавим ему несколько новых, отдельных страниц. Однако, прежде чем мы перейдем к написанию кода, стоит разобраться, что за архитектура лежит в основе нашего приложения.

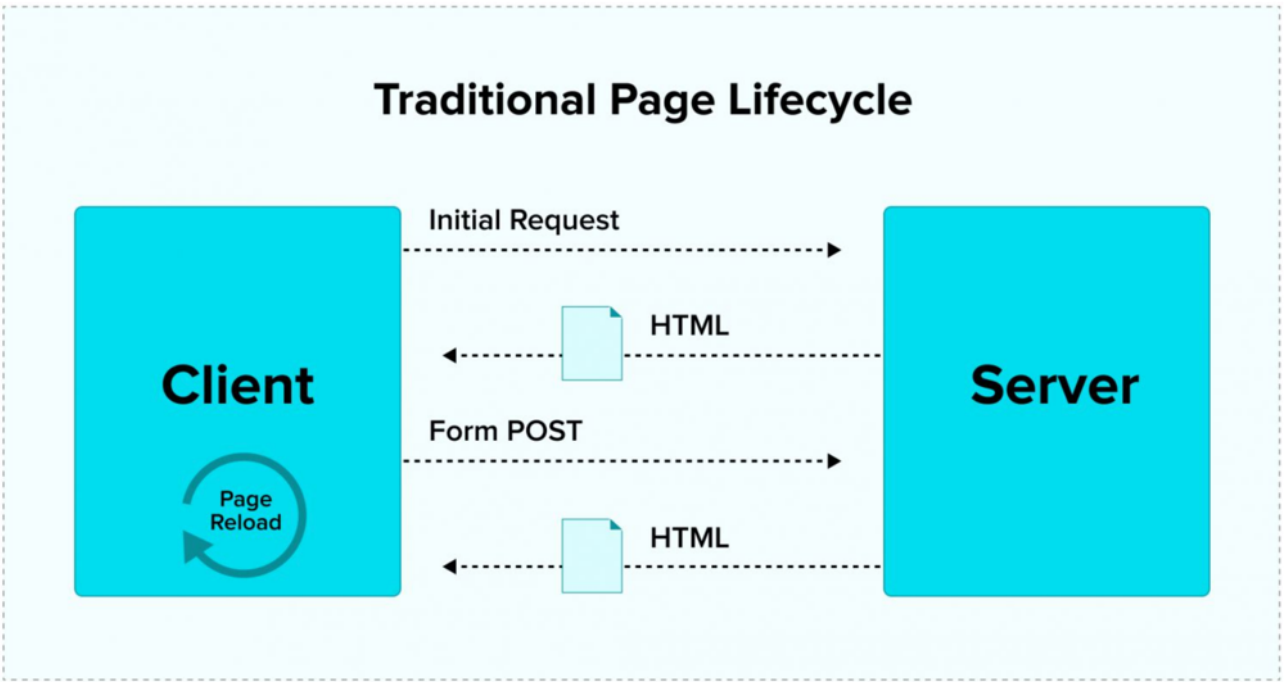
В современной веб-разработке существует несколько подходов проектирования приложений. Мы рассмотрим 2 наиболее популярных из них: MPA (Multiple Page Application) и SPA (Single Page Application). Разберем эти подходы более детально.

Multiple Page Application.

Multiple Page Application, или по-русски – многостраничное приложение, - это традиционное веб-приложение, в рамках которого при обращении к серверу каждый раз запрашивается новая готовая страница. В рамках этой страницы уже сформирован весь контент, построен интерфейс, браузеру остается лишь взять документ и отрендерить его.

При взаимодействии пользователя с формами (например, для отправки сообщения), сервер также отдает готовый документ с результатом обработки.

URL, который находится в адресной строке браузера (а также все URL в ссылках на странице), зачастую соответствуют конкретному готовому документу на сервере.



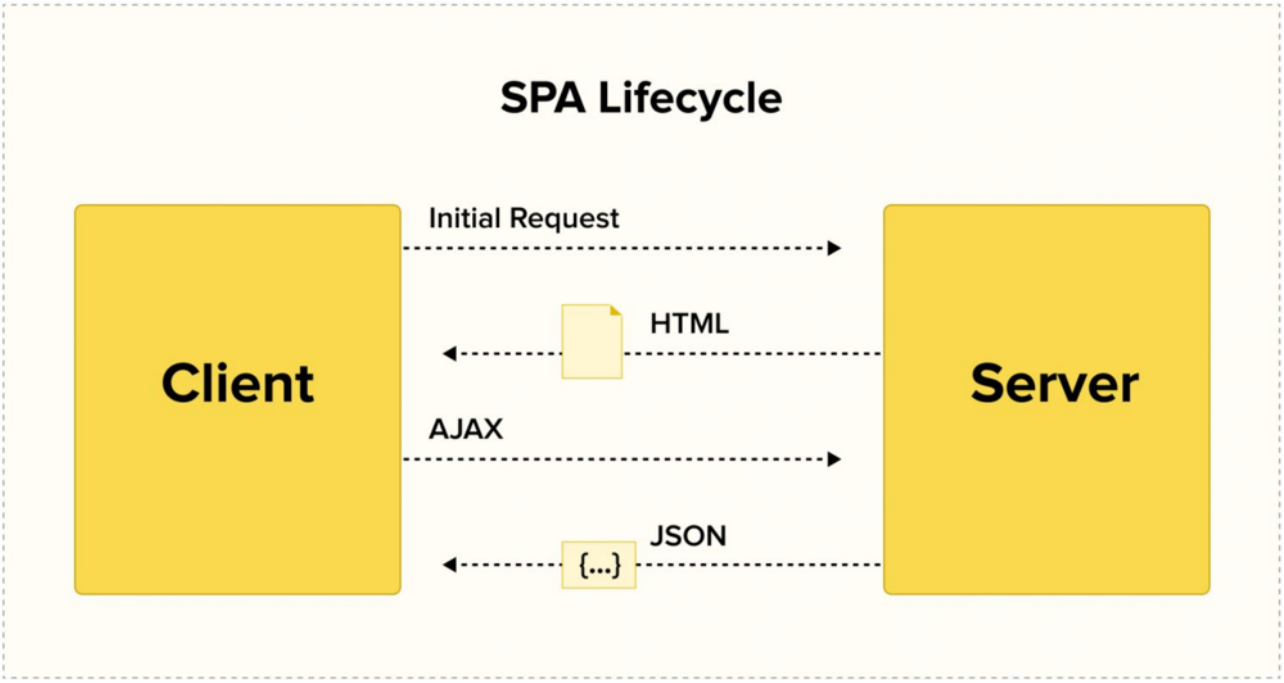
Этот подход весьма прост в реализации (если говорить про клиентскую часть) и зачастую не требует сложных архитектурных решений. Но, компенсируя простоту разработки фронтенда, есть очевидные минусы:

- В то время, как упрощается архитектура фронтенда, усложняется логика серверной части.
- Фронтенд и бэкенд тесно взаимосвязаны. Сложно вести разработку параллельно нескольким участникам команды.
- На каждый запрос клиента идет полный запрос готовой страницы, неэкономный расход трафика и менее отзывчивый интерфейс.

Рассмотрим второй подход.

Single Page Application.

Single Page Application, оно же одностраничное приложение, - веб-приложение, которое работает в рамках одной страницы. При взаимодействии пользователя со страницей, например, при отправке форм, нажатии на ссылки и так далее, мы не уходим с текущей страницы и не перезагружаем ее. При необходимости, мы получаем от сервера дополнительную, запрошенную информацию в виде небольшого JSON-объекта (либо любого другого представления), и видоизменяем ранее загруженную страницу. Добавляем новые блоки, скрываем более ненужные. Пользуясь таким подходом, мы сильно снижаем трафик, при общении с сервером, а также повышаем отзывчивость интерфейса. Например, при переходе по ссылкам внутри нашего приложения, мы можем добавить какую-нибудь интересную анимацию, или информативный прелоадер.



Раз мы работаем всегда в рамках одной страницы, то можно подумать, что мы всегда работаем в рамках одного URL, а ссылки, которые обрабатывают переходы, никак не влияют на адресную строку. Спешим вас заверить – это не так! В отличие от Multiple Page Application, в SPA подходе у нас действительно любой адрес будет соотноситься к одной точке входа нашего приложения (чаще всего – index.html). Однако, нам до сих пор хочется, чтобы пользователи могли делиться ссылками на отдельные страницы нашего приложения, и при использовании этих ссылок приложение не открывало «главную» страницу. Значит, адресная строка должна задавать начальное состояние нашего SPA приложения, а также должна изменяться во время работы пользователя.

Отсюда возникает главная проблема. Как можно организовать навигацию внутри SPA приложения? Давайте разбираться.

Навигация внутри SPA приложения.

Какими свойствами должна обладать навигация внутри SPA приложения?

- При загрузке страницы мы должны отобразить только ту информацию, которую запросил пользователь, основываясь на адресе в URL.
- При переходе по ссылкам внутри приложения мы должны изменять состояние страницы – скрывать более неактуальную информацию (текущей страницы) и показывать новую информацию (для той ссылки, на которую нажал пользователь)
- Нажатие на ссылку не должно провоцировать переход и перезагрузку страницы.
- Нажатие на ссылку должно провоцировать изменение URL в адресной строке.

Условия мы поставили, давайте попробуем подумать, как их можно реализовать.

Традиционно существует два способа поддержать навигацию в SPA-приложении.

1. Использовать hash в URL, который будет отождествлять страницу.
2. Использовать привычные пользователю URL.

Давайте разберем каждый из них.

HASHED URL.

Пути с использованием hash в адресе имеют следующий формат:

https://mysite.com#path
https://mysite.com#dashboard
https://mysite.com#about

Адрес разбивается на 2 части, с помощью специального символа – решетки. В правой части располагается указание на текущую страницу. Эта часть называется хэшем, фрагментом, или якорем.

Если смотреть на то, как работали SPA приложения несколько лет назад, то такой подход встречался повсеместно. Сейчас же, когда SPA подход стал стандартом для веб-разработки, технологии шагнули далеко вперед, и поэтому сегодня можно не часто встречать хэши в адресе. Какие преимущества есть у этого похода?

- Страницы, адреса которых отличаются только хэшем являются одной и той же страницей. Для браузера это поведение очень удобно, так как при перезагрузке страницы мы останемся именно там, где и были, на том же html файле.
- Ссылки, которые меняют лишь хэш составляющую не провоцируют переход и перезагрузку страницы (более того, при первоначальной загрузке страницы, содержащей хэш составляющую, последняя не будет передаваться с запросом на сервер).
- Нам не требуется дополнительно настраивать сервер, чтобы все адреса обслуживались лишь файлом index.html

Давайте попробуем реализовать навигацию в нашем приложении учета расходов, с помощью этого способа.

Заведем 3 страницы:

- dashboard – страница, на которой будет вывод формы добавления расходов, а также их вывод
- about – страница с описанием функционала нашего приложения
- 404 – техническая страница, которая будет открываться при неизвестном адресе

Под каждую страницу создадим свой компонент, которые разместим в главном компоненте

```
<template>
 <div>
```

```
  <header>
        <a href="#dasboard">Dashboard</a>
        <a href="#about">Dashboard</a>
        <a href="#unknown">dev/null</a>
  </header>
  <PageDashboard v-if="page === 'dasboard'" />
  <PageAbout v-else-if="page === 'about'" />
  <Page404 v-else />
 </div>
</template>
```

Свойство page, на основании которого отображаем наши страницы будем передавать через props, из точки входа нашего приложения. Таким образом, если извне, поменять нашу страницу, то будет скрываться или показываться тот или иной компонент.

```
props: {
  page: String,
},
```

Осталось доработать нашу точку входа, чтобы она передавала параметр page, а также отслеживала изменение фрагмента в адресе страницы. Для второй задачи у нас есть специальное событие hashchange, которое срабатывает в контексте window при изменении хэша.

App.vue

```
new Vue({
 el: 'main',
 template: '<App :page="page" />',
 components: {
  App,
 },

 methods: {
  setPage () {
   this.page = location.hash.slice(1)
  }
 },
 mounted () {
  this.setPage()
  window.addEventListener('hashchange', () => {
   this.setPage()
  })
 }
})
```

Теперь, если мы будем кликать на ссылки, которые размещены в компоненте <App />, в адресе будет меняться хэш, что будет провоцировать срабатывание события hashchange. В обработчике события мы меняем свойство page, которое будет управлять отображением компонентов страниц.

Давайте рассмотрим второй подход – работу с привычными URL.

ORDINARY URL

Исходя из «названия», адреса страниц будут выглядеть в привычном нам виде:

https://mysite.com/path
https://mysite.com/dashboard
https://mysite.com/about

Путь до страниц отделяется от доменного имени привычным символом слэша. Какие преимущества есть у данного подхода?

- Адреса выглядят чисто, и привычно, нет никаких специальных символов
- Страницы с такими URL лучше индексируются поисковиками, что важно, если мы хотим привлекать к нашему продукту новых пользователей

Но, как часто бывает, красота требует жертв. В нашем случае, чтобы использовать привычные адреса страниц, нам необходимо будет дополнительно настраивать сервер, чтобы он отдавал на все запросы только точку входа нашего приложения.

Давайте переделаем текущую реализацию навигации на новый подход. Какие шаги необходимо предпринять, чтобы реализовать данный подход?

1. В отличие от HASHED URL, ссылки, с обычным адресом провоцируют переход на другую страницу, или перезагрузку текущей (если адрес ссылки совпадает с текущем URL). Отсюда следует, что нам необходимо повесить обработчик на все ссылки, который будет предотвращать стандартное поведение.
2. При клике на ссылку необходимо изменить URL в адресной строке. Чтобы сделать это в нашем распоряжении есть специальный объект History API.
3. При клике также необходимо изменить свойство page в точке входа нашего приложения. Чтобы это сделать, а обработчике из пункта 1 следует послать событие, которое будет сообщать об изменениях в навигации. В точке входа необходимо подписаться на это событие и непосредственно менять page.
4. В точке входа необходимо подписаться на изменение URL при переходах «вперед»-«назад» через специальные кнопки в браузере. Для этих целей мы можем воспользоваться специальным событием popstate.

План составили, начинаем программировать.

Первым делом необходимо в компоненте <App> повесить обработчик на все ссылки, чтобы предотвратить переходы. Сделаем это в хуке жизненного цикла mounted, чтобы наши ссылки были уже отрендерены в DOM.

App.vue mounted()

```
const links = document.querySelectorAll('a')
links.forEach(link => {
  link.addEventListener('click', event => {
    event.preventDefault()
  })
})
```

Далее, по плану, необходимо изменить URL в адресной строке. Воспользуемся методом pushState объекта History API.

```
const links = document.querySelectorAll('a')
links.forEach(link => {
  link.addEventListener('click', event => {
    event.preventDefault()
    history.pushState({}, '', link.href)
  })
})
```

Метод pushState принимает в себе 3 параметра:

1. state – объект, в который мы можем положить состояние приложения, или каких-то отдельных свойств, которыми можно будет воспользоваться позднее, при возврате на текущую страницу.
2. titile – в настоящее время большинство браузеров игнорируют этот параметр, однако для будущей обратной совместимости он необходим. Можно передавать пустую строку, или любой заголовок, который ассоциируется с текущей страницей
3. url – адрес страницы, на которую изменится URL в адресной строке

Следующим шагом необходимо оповестить наш инстанс Vue приложения, о готовящемся переходе по ссылке. Необходимо послать событие, которое будем обрабатывать в index.js.

Для решения этой задачи мы воспользуемся уже знакомым нам методом $emit, однако будем вызывать его для корневого элемента нашего приложения: $root.

```
const links = document.querySelectorAll('a')
links.forEach(link => {
  link.addEventListener('click', event => {
    event.preventDefault()
    history.pushState({}, '', link.href)

  })
})
```

Сгенерировав событие внутри нашей точки входа, мы должны подписаться на это событие и поменять наше свойство page. Как будет выглядеть наш файл index.js после обновления логики?

```
new Vue({
  el: 'main',
  template: '<App :page="page" />',
  components: {
    App,
  },
  data () {
    return {
      page: ''
    }
  },
  methods: {
    setPage () {
      this.page = location.pathname.slice(1)
    }
  },
  mounted () {
    this.setPage()
    this.$on('router-go', this.setPage)
  }
})
```

Последним, завершающим шагом, необходимо поддержать работу браузерную навигацию – работу стрелочек «вперед» и «назад». Воспользуемся событием popstate. Подпишемся на него в хуке mounted, сразу после подписки на событие router-go:

```
mounted () {
  this.setPage()
  this.$on('router-go', this.setPage)
  window.addEventListener('popstate', this.setPage)
}
```

Если мы сейчас попробуем перейти на новые страницы по ссылкам, то увидим, что наша навигация отлично работает. Однако, функционал нашей реализации слишком примитивен. Мы не можем передавать параметры через url, мы не можем отслеживать начало и конец перехода по ссылкам, мы не можем программно переходить на нужную нам страницу, компоненты не знают, какая страница сейчас отображается. Все это и многое другое уже реализовано в библиотеке Vue-Router. Давайте с ней познакомимся.

Vue-Router. Установка и настройка.

Чтобы установить Vue-Router к нам в приложении, достаточно воспользоваться уже хорошо знакомой нам командой npm install:

```
npm install -S vue-router@2
```

Дальше, по аналогии с тем, как мы подключали библиотку Vuex, нам надо подключить роутер к нам в приложение. Создадим файл router/index.js со следующим содержимым:

router/index.js

```
import Vue from 'vue'
import Router from 'vue-router'

Vue.use(Router)

export default new Router({

})
```

Сообщаем Vue, что собираемся использовать роутер внутри нашего приложения, а также создаем экземпляр самого роутера. Теперь его надо подключить в точке входа нашего приложения:

main.js

```
import Vue from 'vue'
import App from './App.vue'

import store from './store'

// импортируем экземпляр роутера из листинга выше
import router from './router'

new Vue({
  el: 'main',
  template: '<App />',
  components: {
    App,
  },
```

```
  store,
  router, // подключаем роутер к нашему приложению
})
```

Создавать одностраничное (SPA) приложение используя Vue в связке с Vue Router очень просто. У нас уже есть компоненты, которые содержат в себе отображение будущих страниц. Это компоненты, которые мы создали чуть ранее, когда проектировали свой собственный способ навигации в приложении. Эти компоненты: <PageDashboard />, <PageAbout />, <Page404 />. Добавляя Vue Router мы просто сопоставляем наши компоненты с маршрутами и говорим роутеру где именно их отображать.

Давайте заведем маршруты в нашем роутере, которые будут сопоставлены с вышеперечисленными компонентами. В объекте, который передается в конструктор роутера есть специальная директива routes, внутри которой мы и будем прописывать все существующие маршруты в нашем приложении.

index.js

```
import Vue from 'vue'
import Router from 'vue-router'

import PageDashboard from '@/pages/PageDashboard.vue'
import PageAbout from '@/pages/PageAbout.vue'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/dashboard',
      name: 'dashboard',
      component: PageDashboard
    },
    {
      path: '/about',
      name: 'about',
      component: PageAbout
    },
  ]
})
```

Каждый отдельный маршрут представляет из себя объект, реализующий интерфейс RouterConfig. Внутри этого объекта есть множество свойств, которые мы можем

использовать при настройке маршрута. В нашем примере мы воспользовались некоторыми из них:

- **path –** путь, которому будет соответствовать компонент страницы
- **component –** компонент, который будет отображаться, если URL страницы будет совпадать с указанной в path строкой
- **name –** имя маршрута (будем использовать далее)

Маршруты мы завели, осталось отобразить наши компоненты, когда потребуется. Перейдем в файл App.vue и отобразим работу нашего роутера:

App.vue

```
<template>
 <div>
  <header>
   <router-link to="/dashboard">Dashboard</router-link>
   <router-link to="/about">About</router-link>
  </header>
  <router-view />
 </div>
</template>
```

На что стоит здесь обратить внимание. Мы воспользовались сразу двумя новыми компонентами: <router-link /> и <router-view />. Названия компонентов четко передают их предназначение.

С помощью компонента <router-link> мы отображаем пользователю ссылку (по умолчанию идет рендер в тэг <a><a/>). Такая ссылка уже содержит в себе обработчик нажатия: вместо стандартного поведения будет запущена логика работы Vue Router. Свойство to определяет URL для перехода.

Компонент <router-view /> отображает компонент, для которого совпадает маршрут (определенный в роутере).

Vue-Router. Программная навигация.

Помимо уже рассмотренного нами метода навигации с помощью предоставленных компонентов-ссылок, в нашем распоряжении также имеется возможность программно вызывать переход на требуемый маршрут.

После добавления роутера в наше приложение, в экземпляре Vue у нас появился доступ к свойству $router, в котором находится инстанс нашего роутера.

С помощью специального метода push мы можем из любого компонента, в любое время сделать переход на нужную страницу.

Например, если мы в хуке created компонета <App /> добавим следующую строку, то при первом заходе на наше приложение у нас всегда будет открывать страница Dashboard:

```
created () {
  this.$router.push({ name: 'Dashboard' })
  this.setPaymentsListData(this.fetchData())
}
```

Vue-Router. 404.

Мы разобрались как добавлять маршруты в наше приложение, но что будет, если пользователь перейдет по ссылке, которую не определяет ни один компонент? Что если на адрес не задан никакой маршрут?

На самом деле ничего не произойдет. Компонент <router-view /> просто скроется и ничего не отобразит. Но для пользователя это не самое очевидное поведение, пользователь ожидает увидеть контент, который должен был быть по ссылки, или хотя бы какое-то сообщение, которое объяснит ему, что искать по данному адресу ничего не стоит. Для таких случаев хорошей практикой будет завести отдельную страницу 404 ошибки.

Чуть ранее мы уже завели компонент <Page404 />, и именно его мы будем показывать, если адрес текущей страницы не совпадет ни с одним из маршрутов. Как это можно сделать?

Следует понимать, что необходимо соблюдать четкий порядок маршрутов в роутере. Первый маршрут, в котором path совпадет с текущим URL будет использоваться в качестве выбранного маршрута. Для заведенных нами маршрутов /dashboard и /about это не актуально, однако помимо такой простой записи path, мы можем использовать маршруты, которые попадают под целую группу URL. Чтобы завести такой маршрут, можно воспользоваться символом «*».

Например, мы хотим, чтобы все адреса, которые начинаются с «about» вели на соответствующую страницу:

```
export default new Router({
  mode: 'history',
  routes: [
    {
      path: '/dashboard',
      name: 'Dashboard',
      component: PageDashboard
    },
    {
      path: '/about*',
      name: 'About',
      component: PageAbout
```

```
    }
  ]
})
```

Теперь, если пользователь перейдет по ссылкам:

- /about
- /aboutus
- /aboutapp
- /about-payments-app

То каждая из таких ссылок будет обработана маршрутом About.

А что произойдет, если мы в качестве всего path оставим лишь одну «*»? Это будет значить, что абсолютно любой URL будет подходить под данное описание пути. Если такой маршрут поставить первым в списке роутов, то все остальные маршруты будут игнорироваться.

Однако, если маршрут, который соответствует всем возможным адресам поставить самым последним, то мы сможем обработать любой URL, который не совпал с маршрутами, расположенными выше. Очень похоже на поведение для 404 страницы! Давайте добавим ее в наш роутер.

```
export default new Router({
  mode: 'history',
  routes: [
    {
      path: '/dashboard',
      name: 'Dashboard',
      component: PageDashboard
    },
    {
      path: '/about*',
      name: 'About',
      component: PageAbout
    },
    {
      path: '*',
      component: Page404
    }
  ]
})
```

Теперь, при неизвестном URL будет отображаться компонент Page404. Можно улучшить поведение, и при неизвестном URL принудительно переводить пользователя на адрес /404.

Давайте заведем соответствующий маршрут, а в текущим воспользуемся директивой redirect:

```
{
  path: '/404',
  name: 'NotFound',
  component: Page404
},
{
  path: '*',
  redirect: '/404'
}
```

При редиректе мы можем воспользоваться не адресом, куда редиректить, а именем маршрута (который указан в директиве name):

```
{
  path: '*',
  redirect: { name: 'NotFound' }
}
```

Vue-Router. Динамические пути.

Мы уже познакомились с одной особенностью директивы path – группировка URL по символу «*». Однако, возможности наши на этом не заканчиваются!

Очень часто нам требуется сопоставить маршруты с заданным шаблоном с одним и тем же компонентом. Например, в прошлый раз на практике была задача реализовать пагинацию в компоненте просмотра списка наших платежей. В нашем приложении этот компонент открывается по адресу /dashboard. Но что, если мы сразу хотим задать определенную страницу при открытии компонента? Во Vue-Router мы можем использовать динамический сегмент, чтобы достичь этого:

```
export default new Router({
  mode: 'history',
  routes: [
    {
      path: '/dashboard/:page',
      name: 'Dashboard',
      component: PageDashboard
    },
    …
  ]
})
```

Теперь все URL вида /dashboard/1 и /dashboard/999 будут соответствовать данному маршруту.

Динамические сегменты начинаются с двоеточия «:». Казалось бы, в чем отличие от записи через «*», зачем такие сложности? На самом деле, при сопоставлении маршрута, значение динамического сегмента можно получить через обращение к инстансу роутера в каждом компоненте:

```
const page = this.$route.params.page
```

Таким образом, в компоненте PaymentsDisplay.vue мы можем сразу получить необходимую нам страницу из URL.

Vue-Router. Навигационные хуки.

Мы рассмотрели уже весьма внушающий набор функционала, который нам предоставляет Vue-Router. Однако на этом наши возможности не заканчиваются;

Когда мы рассматривали работу с компонентами, мы познакомились с понятием жизненного цикла, а также увидели работу специальных функций – хуков, которые срабатывают в определенный момент жизненного цикла.

Такой функционал очень полезен. Мы можем отследить начало выполнения какого-то действия, а также его завершение. В рамках роутера Vue нам также предоставляет свой набор хуков. Более того, хуки роутера предоставляются как для каждого конкретного маршрута, так и для всех маршрутов разом!

Навигационные хуки используются для перенаправлений, отмены навигационных переходов, а также для выполнения какой-то логики, которая необходима в рамках перехода на новый URL.

Давайте рассмотрим, как работают глобальные хуки, то есть те методы, которые вызываются при любом переходе, на любой маршрут.

В нашем распоряжении оказалось 3 хука: beforeEach и beforeResolve и afterEach.

beforeEach

Как следует из названия, данный хук вызывается перед любым переходом на новый URL. Когда это может быть полезно? Например, мы хотим, чтобы с нашим приложением могли работать только авторизованные пользователи. Тогда, перед тем как перейти на запрошенную страницу, мы можем сделать проверку на авторизацию, и при ее отсутствии перенаправить пользователя на страницу логина. Как это могло бы выглядеть в рамках нашего приложения?

```
const router = new Router({
  mode: 'history',
  routes: [
    {
      path: '/auth',
      name: 'Login',
      component: { template: '<form>*AUTH FORM*</form>' }
    },
    …
  ]
})


const userAuthExists = false // зададим текущее состояние, для примера

router.beforeEach((to, from, next) =>{
  if (to.name !== 'Login' && !userAuthExists) {
    next({ name: 'Login' })
  } else {
    next()
  }
})
```

С помощью метода beforeEach мы зарегистрировали глобальный навигационный хук – функцию-обработчик, которые передали в качестве параметра.

Внутри нашей функции-обработчика доступно 3 аргумента функции: to, from, next. Разберемся, за что они отвечают:

1. **to –** целевой объект Route, к которому осуществляется переход. Внутри этого объекта мы можем получить доступ к таким свойствам как path (путь маршрута), params (параметры из динамического пути, которые мы рассмотрели в предыдущей главе), name (имя маршрута, если оно задано в роутере) и многие другие.
2. **from –** объект Route маршрута, который пользователь собирается покинуть
3. **next –** функция, вызов которой сообщает роутеру, что хук завершил свое выполнение и можно переходить к следующему этапу работы. Мы обязаны вызывать данную функцию внутри хука, причем обязаны вызывать ее только один раз! С помощью этой функции мы можем управлять поведением нашего перехода, в зависимости от переданного параметра:
    - **false** – переход будет отменен
    - **‘/url’, { path: ‘url’ }, { name: ‘routeName’ } –** переход будет перенаправлен по новому маршруту, который указан в параметре
    - **Пустое значение –** завершение хука без изменения поведения навигации.

Следующим этапом работы роутера, после вызова функции next рассмотренного хука будет срабатывание хука, зарегистрированного с помощью beforeResolve.

beforeResolve

Хук, зарегистрированный с помощью метода роутера beforeResolve очень похож на beforeEach, с той разницей, что разрешающий хук будет вызван непосредственно перед подтверждением навигации, то есть после того, как будут разрешены все остальные хуки (включая хуки, которые мы можем зарегистрировать для каждого компонента в роутере отдельно).

Когда переход полностью завершился, мы можем воспользоваться еще одним хуком, который можем зарегистрировать с помощью метода роутера afterEach.

afterEach

Основное отличие хука, зарегистрированного с помощью метода роутера afterEach заключается в том, что в параметрах хука нет доступа к функции next. Это достаточно логично, так как хук является завершающим в цепочке навигационного перехода. Когда нам может быть полезен этот хук? В том случае, если мы хотим совершать какую-то операцию после каждого перехода на новую страницу. Например, мы хотим изменить title нашего документа:

```
const router = new Router({
 routes: [
  {
   path: '/dashboard/:page',
   name: 'Dashboard',
   component: PageDashboard
  },
  {
   path: '/about*',
   name: 'About',
   component: PageAbout
  },
  {
   path: '/404',
   name: 'NotFound',
   component: Page404
  },
  {
   path: '*',
   redirect: { name: 'NotFound' }
  }
 ]
})
```

```
const getTitleByRouteName = routeName => {
  return {
    'Dashboard': 'Take a look on your payments and add more!',
    'About': 'Anything about our awesome application!',
    'NotFound': 'Oops! Seems like we lost this page :('
  }[routeName]
}

router.afterEach((to, from) => {
  document.title = getTitleByRouteName(to.name)
})
```

Используемая литература

1. Официальный сайт Vue.js - ссылка
2. Документация Vue CLI - ссылка