Routing in a Vue.js Applicatoin

The most important part for building SPA. Technically we have only one page (index.html) and we use this page to simulate navigation for user. There is no reload pages. Now you are able to switch out different components in your application through the router dynamically.

Setting up the Vue.js Router (vue-router)

For this we need to install **vue-router**.

```
# npm install --save vue-router
```

Setting up and Loading Routes

In **main.js** file, we have to import **vue-router** and register it using **Vue.use()**. We do need to register it globally before the *new Vue(...)* root instance.

```
// main.js
...
import VueRouter from 'vue-router';

Vue.use(VueRouter);
new Vue({
    el: '#app',
    render: h => h(App)
})
...
```

To add routes, we create **routes.js** file with the following two routes for

start.

```
// route.js
import User from './components/user/User.vue';
import Home from './components/Home.vue';

export const routes = [
  { path: '', component: Home},
  { path: '/user', component: User},  //
www.example.com/user
]
```

and in **main.js**, we need to import the routes from **routes.js** file as following:

```
// main.js
...
import VueRouter from 'vue-router';
import { routes } from './routes';  // routes.js

Vue.use(VueRouter);

const router = new VueRouter({
    routes: routes          // or just 'routes' in
ES6
});

new Vue({
    el: '#app',
    router: router,         // or just 'router' in
ES6
    render: h => h(App)
})
```

Next thing to do is to add a selector (or a placeholder) for the route in **App.vue** using the tag <router-view></router-view>.

```
// App.vue
<template>
...
    <h1>Routing</h1>
    <hr>
    <router-view></router-view>
...
</template>
```

Now we can use the URL: *http://localhost/#/* to access **Home** component and URL: *http://localhost/#/user* to access **User** component.

Understanding Routing Modes

- The '#' is add automatically by default. This is quite common for SPA.
- The URL path that is before the '#' gets send to the server.
- The path behind the '#' is handle by the SPA.
- It would be nicer to have URL without '#'.
    - We just need to configure the web server to return the **index.html** in all cases (even in the case of errors).
    - In **main.js**, we also need to configure **history** mode.
    - In **npm run dev**, it is configured to always return **index.html** so the 'history' mode is working by default.

```
// main.js
...
import VueRouter from 'vue-router';
import { routes } from './routes';  // routes.js
```

```
Vue.use(VueRouter);

const router = new VueRouter({
    routes: routes,
    mode: 'history'                 // setting
history mode
});
...
```

Find more detail about **history** model and how to configure web servers at *https://router.vuejs.org/en/essentials/history-mode.html*

Navigating with Router Links

Let's create a new component, the **Header** component. We will use some **Bootstrap** style in Header's template. Go to the URL *https://getbootstrap.com/docs/3.3/components/* and look for **Pills** in the *Navs* section. Grap the Pills navigation style and put them in the **Header's template**.

We need to modify the template and use <router-link></router-link> as following:

```
// Header.vue
<template>
  <ul class="nav nav-pills">
    <li role="presentation"><router-link
to="/">Home</router-link></li>
    <li role="presentation"><router-link
to="/user">User</router-link></li>
  </ul>
</template>
```

The <router-link></router-link> tag gets replace by the when rendered. The important thing is there is no **reload** behavior

anymore.

Styling Active Links

There is a problem of the Bootstrap's **active link** style. We can use **active-class="active"** style on the <route-link></router-link> and modify the template as following:

```
// Header.vue
...
    <router-link to="/" tag="li" active-
class="active" exact><a>Home</a></router-link>
    <router-link to="/user" tag="li" active-
class="active"><a>User</a></router-link>
...
```

The keyword **exact** tells Vue.js that it will use "active" class only when the *path* is exact match.

Navigating from Code

We can add navigation from within the code by using the **this.$router.push()** method. For example, we can modify **User.vue** as following:

```
// User.vue
<template>
    <div>
        <h1>The User Page</h1>
        <hr>
        <button @click="navigateToHome" class="btn
btn-primary">Go to Home</button>
    </div>
</template>
```

```
<script>
    export default {
        methods: {
            navigateToHome() {
                this.$router.push('/');     // add
navigation
            }
        }
    }
</script>
```

Setting up Route Parameters

So far there is no dynamic route. We might want to pass a paramenter with the URL, for example, *http://localhost/user/10* where 10 is the *user_id*. We can do that by modifying the **routes.js** as following:

```
// route.js
import User from './components/user/User.vue';
import Home from './components/Home.vue';

export const routes = [
  { path: '', component: Home},
  { path: '/user/:id', component: User},  //
http://localhost/user/10
]
```

We can even user the path like */user/:id/detail*.

Fetching and Using Route Parameter

To get the dynamic parameter (corresponding to what defined in the routes). We use **this.$route.[param]** the get access to the passed parameter. For example, in the User component, we can get access to the **id** as following:

```
// User.vue
<template>
    ...
    <p>Load ID: {{ id }}</p>
    ...
</template>

<script>
    export default {
        data() {
            return {
                // corresponding to the '/usr/:id'
route
                // $route NOT $router
                id: this.$route.params.id
            }
        },
        ...
    }
</script>
```

Reacting to changes in Route Parameter

Let's add another **User** navigation in *Header.vue*.

```
// Header.vue
...
    <router-link to="/" tag="li" active-
class="active" exact><a>Home</a></router-link>
    <router-link to="/user" tag="li" active-
class="active"><a>User</a></router-link>
    <router-link to="/user/10" tag="li" active-
class="active"><a>User10</a></router-link>
    <router-link to="/user/20" tag="li" active-
class="active"><a>User20</a></router-link>
```

```
...
```

When we click *User20* the **id** is still '10'. Vue.js is keeping the existing instance, hence the old data. Therefore, we need to **watch** for the *route changes* then update the parameter using **watch** property.

```
// User.vue
<script>
    export default {
        ...
        watch: {
            // a function passing (new-route, old-
route)
            '$route'(to, from) {
                this.id = to.params.id;
            }
        }
        ...
    }
</script>
```

Setting up Child Routes (Nested Routes)

We can add subroutes within the **User.vue** component. This is done by adding **children: [... array of subroutes...]** in the **routes.js**.

```
// routes.js
import User from './components/user/User.vue';
import UserStart from
'./components/user/UserStart.vue';
import UserDetail from
'./components/user/UserDetail.vue';
import UserEdit from
'./components/user/UserEdit.vue';
```

```
import Home from './components/Home.vue';

export const routes = [
  { path: '', component: Home},
  // { path: '/user/:id', component: User}
  { path: '/user', component: User, children: [  //
add subroutes
    { path: '', component: UserStart },          //
'/user/'
    { path: ':id', component: UserDetail },      //
'/user/id'
    { path: ':id/edit', component: UserEdit }    //
'/user/id/edit
  ]}
]
```

Since all new child routes are the children of /**user** therefore we need to add a seclector (a placeholder) for these routes in the **User.vue** component.

```
// User.vue
<template>
    <div>
        ...
        <hr>
        <!-- add subroutes' component -->
        <router-view></router-view>
        ...
    </div>
</template>
```

Navigating to Nested Routes

Let's make the user list in the **UserStart.vue** component clickable to display clicked user detail in **UserDetail.vue**.

First, we need to modify the UserStart's template to use <router-link></router-link> as following:

```
// UserStart.vue
<template>
...
    <ul class="list-group">
        <router-link
            tag="li"
            to="/user/1"
            class="list-group-item"
            style="cursor: pointer">User 1</router-
link>
        <router-link
            tag="li"
            to="/user/2"
            class="list-group-item"
            style="cursor: pointer">User 2</router-
link>
        <router-link
            tag="li"
            to="/user/3"
            class="list-group-item"
            style="cursor: pointer">User 3</router-
link>
    </ul>
...
</template>
```

Of course, we can use **:to** to bind **to** attribute with data property that contains dynamic route.

Next, we interpolate the **$route.params.id** in the UserDetail's template to show the *id* parameter.

```
// UserDetial.vue
<template>
    <div>
        <h3>Some User Details</h3>
        <p>User loaded has ID: {{ $route.params.id }}
</p>
    </div>
</template>
```

Dynamic Router Links

To add **UserEdit.vue** component in **UserDetail.vue**, we add the <route-link></route-link> as following:

```
// UserDetail.vue
<template>
    <div>
        <h3>Some User Details</h3>
        <p>User loaded has ID: {{ $route.params.id }}
</p>
        <router-link
            tag="button"
            :to="'/user/' + $route.params.id +
'/edit'"
            class="btn btn-primary">Edit</router-
link>
    </div>
</template>
```

Here, we construct the path manually. However there is a better way to do this.

Creating Links with Named Routes

We can give each route a name by adding **name:** in the **route.js** file.

```
// routes.js
...
    { path: '', component: Home, name: 'home'},
    ...
    { path: ':id/edit', component: UserEdit, name:
'userEdit' }
...
```

Later on we can use this name to identify the a route on any page. For example, we can use the name *userEdit* as following:

```
// UserDetail.vue
<template>
...
    <router-link
            tag="button"
            :to="{ name: 'userEdit', params: { id:
$route.params.id } }"
            class="btn btn-primary">Edit</router-
link>
...
</template>
```

This time we use **params** object to pass all parameters. We can also use the name **home** in the code to route as well. All of the following **push()** give the same result.

```
// User.vue
<script>
...
    methods: {
```

```
        navigateToHome() {
            this.$router.push('/');
            // this.$router.push( { path: '/' })
            // this.$router.push( { name: 'home' })
        }
    }
...
</script>
```

Using Query Parameters

Query parameters are the parameters you have at the end of URL. They are separated by **?**, for example, *http://localhost/xxx?a=100&b=hello*. There are many ways to add query parameters.

```
<router-link to="/?a=100" ... > xxx </router-link>

or
// UserDetail.vue
<router-link
    tag="button"
    :to="{ name: 'userEdit',
           params: { id: $route.params.id },
           query: { locale: 'en', q: 100 } }"   //
query params
    class="btn btn-primary">Edit</router-link>
```

To access query parameters, we use **$route.query.[param]**. For example:

```
// UserEdit.vue
<template>
    <div>
        <h3>Edit the User</h3>
        <p>Locale: {{ $route.query.locale }}</p>
```

```
        <p>Data: {{ $route.query.q }}</p>
</div>
</template>
```

Multiple Router Views (Named Router Views)

We can assign a name to <router-view></router-view>. For example:

```
// App.vue
    <router-view name="header-top"></router-view>
    <router-view></router-view>
    <router-view name="header-bottom"></router-view>
```

The <router-view> without the *name* attribute will be the **default** one. So how do we assign name in **routes.js**.

```
// routes.js
export const routes = [
  { path: '', name: 'home', components: {
    default: Home,
    'header-top': Header
  } },
  { path: '/user', components: {
    default: User,
    'header-bottom': Header
  }, children: [
    { path: '', component: UserStart },         //
'/user/'
    { path: ':id', component: UserDetail },      //
'/user/id'
    { path: ':id/edit', component: UserEdit, name:
'userEdit' }  // '/user/id/edit
  ]}
]
```

The named router view is normally used when we want to have dynamic location for components.

Redirecting

What if user enter something that is not cover by the router. We want to redirect to a specific path. To do this, we need to add **redirect** path in the **routes.js**:

```
// routes.js
export const routes = [
  { path: '', component: Home, name: 'home'},
  // { path: '/user/:id', component: User}
  { path: '/user', component: User, children: [
    { path: '', component: UserStart },
    { path: ':id', component: UserDetail },
    { path: ':id/edit', component: UserEdit, name:
'userEdit' }
  ]},
  { path: '/redirect-me', redirect: '/user'} ,
  { path: '/redirect-you', redirect: { name: 'home'}}
]
```

Setting up Catch All Routes / Wildcards

To handle any routes that do not exist in our application, we can use wildcard '*' in the **routes.js**:

```
// routes.js - any other routes are redirect to '/'
...
 { path: '*', redirect: '/' }
...
```

Animating Route Transitions

How to animate switching from one route to other routes. This requires some CSS setup and also the <transition></transition> tag.

```
// App.vue - wrap the transition tag around the
<router-view>
...
    <app-header></app-header>
    <transition name="slide" mode="out-in">
        <router-view></router-view>
    </transition>
...
```

Passing the Hash Fragment

Sometimes we want to navigate to a specific part on the page by adding **#[id]** at the end of the URL. Let's add some *id* to **UserEdit.vue** template.

```
// UserEdit.vue
<template>
    ...
    <!-- adding a section with the height of 700px --
>
    <div style="height: 700px"></div>
    <hr>
    <p id="data">Some extra paragraph with id="data"
</p>
    <hr>
</template>
```

The default behavior when navigate using *id* is to *jump (or scroll down)* to the DOM with that *id*. We can pass the *#[id]* by adding *hash* object to

the **:to** binding. Let's modify the **UserDetail** as following:

```
// UserDetail.vue
<template>
        ...
        <router-link
            tag="button"
            :to="link"      // binding to the link
data property
            class="btn btn-primary">Edit</router-
link>
        ...
</template>

<script>
export default {
    data() {
        return {
            link: {
                name: 'userEdit',
                params: {
                    id: this.$route.params.id
                },
                query: {
                    locale: 'en',
                    q: 100
                },
                hash: '#data'   // adding hashtag
            }
        }
    }
}
</script>
```

However, in Vue.js application by default, it does not have the *jump (or scroll down)* behavior. We can use Vue.js to control this.

Controlling the Scroll Behavior

Configuring the scroll behavior is simple. We can add the **scrollBehavior()** when creating the **VueRouter(...)** in the **main.js**.

```
// main.js
const router = new VueRouter({
  routes,
  mode: 'history',

  scrollBehavior(to, from, savedPosition) {
    if (savedPosition) {
      // jump to the saved position, if it is set.
      return savedPosition;
    }
    if (to.hash) {
      // jump to a selector, if there is one.
      return { selector: to.hash };
    }
    return {x: 0, y: 700};  // jump to the coordinate
  }

});
```

Protecting Routes with Guards

We may want to control if the user allowed to enter a certain route or leave it. Checking before enter the route and before leaving the route.

Using the "beforeEnter" Guard

We can call a method **router.beforeEach(...)** in **main.js** to define what to do before entering each route using *function*.

```
// main.js
```

```
...
router.beforeEach((to, from, next) => {
  console.log('global beforeEach');
  next(); // define what to do next
});

new Vue({
  el: '#app',
  router,
  render: h => h(App)
})
```

Now before entering each route, this function will be executed. The **next()** function define "what to do next". There are three options for this:

- next(); = continue as normal.
- next('false') = abort.
- next({ path: ... }) = redirect.

We do not use this all the time but only protecting a certain route. Let's say we want to protect the route to **UserDetail**. We can add the **beforeEnter** parameter in a specific route in **routes.js**.

```
// routes.js - protecting the route to UserDetail.vue
...
    { path: ':id', component: UserDetail,
beforeEnter: (to, from, next) => {
        console.log('inside route setup');
        next();
    } },
...
```

We can also do this in each component directly. Using the method **beforeRouteEnter(to, from, next)** defined in the component itself. It is similar to lifecycle hook implemented by vue-router. However, this is a

special case. In the

```
// UserDetail.vue
<script>
...
    data() {
        ...
    },
    beforeRouteEnter(to, from, next) {
        // here the component is not loaded yet, data
are not available
        // 'this.link' is not available at this
point.
        // inside the next() the component is
loaded!!!
        if (false) {    // may be checking for user
is authenticated
            next( vm => {
                vm.link
            });
        } else {
            next(false);
        }
    }
</script>
```

Using the "beforeLeave" Guard

Let check before leaving the **UserDetail** component. We add a button to its template and add **beforeRouteLeave(to, from, next)** to the component.

```
// UserDetail.vue
<template>
    <div>
```

```
        <h3>Edit the User</h3>
        <p>Locale: {{ $route.query.locale }}</p>
        <p>Data: {{ $route.query.q }}</p>

        <button class="btn btn-primary"
@click="confirmed = !confirmed">Confirm</button>

        <div style="height: 700px"></div>
        <hr>
        <p id="data">Some extra paragraph with
id="data"</p>
        <hr>
    </div>
</template>

<script>
    export default {
        data() {
            confirmed: false
        },
        beforeRouteLeave(to, from, next) {
            // before leaving the component
            // the component is created
            if (this.confirmed) {
                next();
            } else {
                if (confirm('Are you sure?')) {
                    next();
                } else {
                    next(false);
                }
            }
        }
    }
</script>
```

Now if user wants to leave the component without clicking *Confirm* button, user needs to comfirm through the javascript's *confirm dialog*.

Loadin Routes Lazily

Until now, if do not access the */user* route at all, the **User.vue** component will never be created. For big application, this is not good. We might want to load certains components at a certain time.

Using webpack to do this is not hard. Everything we **import** are include in the *bundle* no matter if we use them or not. We can lazily load components as following.

```
// routes.js

// this syntax is similar to the import syntax below
// require('./components/Home.vue');
import Home from './components/Home.vue';

const User = resolve => {
    require.ensure(['./components/user/User.vue'], ()
=> {
        // aync function that resolve the component
when required

resolve(require('./components/user/User.vue'));
    } );
}

const UserDetail = resolve => {

require.ensure(['./components/user/UserDetail.vue'],
() => {
        // aync function that resolve the component
when required

resolve(require('./components/user/UserDetail.vue'));
    } );
}
```

```
const UserEdit = resolve => {
    ...
}

const UserStart = resolve => {
    ...
}
```

This will divide the application into many small bundles. The bundle will be downloaded when required. In the *developer tools*, go to *Network* tab to see this. It is also possible to group bundles.

```
// routes.js

const User = resolve => {
    require.ensure(['./components/user/User.vue'], ()
=> {
        // aync function that resolve the component
when required

resolve(require('./components/user/User.vue'));
    }, 'bundle1' );
}

const UserDetail = resolve => {

require.ensure(['./components/user/UserDetail.vue'],
() => {
        // aync function that resolve the component
when required

resolve(require('./components/user/UserDetail.vue'));
    } , 'bundle1' );
}
```