Vue Router

Install Vue Router

Open your terminal or command prompt, navigate to your Vue.js project's root directory, and run one of the following commands based on your package manager

```
npm install vue-router
```

This will install Vue Router as a dependency in your project.

Configure Vue Router

In your project's `src` folder, locate the `main.js` file. This is the main entry point of your Vue application. Import Vue and Vue Router at the top of the file:

```
import Vue from 'vue';
import VueRouter from 'vue-router';
```

Register Vue Router with Vue by using the `Vue.use()` method: `Vue.use(VueRouter);`

Basic Routing

Define Routes

After registering Vue Router, you can proceed to define your routes. In the same `main.js` file, create an array of route objects representing each route in your application. Each route object should have a `path` (URL) and a `component` (Vue component) property.

```
 main.js
import Vue from 'vue';
import VueRouter from 'vue-router';
import App from './App.vue';
import Home from './components/Home.vue';
import About from './components/About.vue';

Vue.use(VueRouter);
```

```
const routes = [ { path: '/', component: Home }, { path: '/about', component: About }
];
```

Create Vue Router Instance

Create a new instance of `VueRouter` with the defined routes

```
// main.js
const router = new VueRouter({ routes
});
```

Create and Mount the Vue App

Now, create a new Vue instance, passing the `router` instance to the `router` option. Then, mount the Vue app to the DOM:

```
// main.js
new Vue({ router, render: h => h(App)
}).$mount('#app');
```

Navigating between Routes

Use `<router-link>` and `<router-view>`

With Vue Router integrated into your project, you can use the `<router-link>` component to navigate between routes and the `<router-view>` component to render the matched component for the current route.

```
<!-- App.vue -->
<template>
  <div>
    <router-link to="/">Home</router-link>
    <router-link to="/about">About</router-link>
    <router-view></router-view>
  </div>
</template>
```

In the above example, we have two routes: one for the path `'/'` (Home) and another for the path `'/about'` (About). When you click on the "Home" or "About" link, the corresponding component (Home or About) will be rendered inside the `<router-view>`.

Define Components

Make sure to define the `Home` and `About` components in separate files under the `components` folder.

```
<!-- Home.vue -->
<template>
  <div>
    <h1>Welcome to the Home page!</h1>
  </div>
</template>

<!-- About.vue -->
<template>
  <div>
    <h1>About Us</h1>
      <p>We are a passionate team of developers building awesome applications!</p>
  </div>
</template>`
```

Now, when you run your Vue.js application, you'll be able to navigate between the Home and About pages using the `<router-link>` components. The corresponding components will be displayed inside the `<router-view>` as you navigate to each route.

Route Parameters

Route parameters allow you to create dynamic routes that can match different values. For example, in a blog application, you might have a route for individual blog posts, where the URL contains the ID of the post. Route parameters are defined in the route path using a colon (`:`) followed by the parameter name.

Defining Route Parameters

Let's add a dynamic route that can match different blog post IDs.

```
// main.js
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  { path: '/blog/:id', component: BlogPost }
];
```

In this example, the `:id` in `/blog/:id` is a route parameter. It can match any value in the actual URL.

Accessing Route Parameters

To access the route parameters inside your component, you can use `this.$route.params`.

```
<!-- BlogPost.vue -->
<template>
  <div>
    <h1>Blog Post</h1>
    <p>Post ID: {{ postId }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      postId: null
    };
  },
  created() {
    this.postId = this.$route.params.id;
  }
};
</script>
```

Now, if you visit `/blog/123`, the `BlogPost` component will render, and it will show "Post ID: 123".

Reacting to Route Parameter Changes

If you navigate from one blog post to another within the same component, the route parameter will change, but the component won't be recreated. To react to the route parameter changes, you can use the `beforeRouteUpdate` hook.

```
<!-- BlogPost.vue -->
<script>
export default {
  data() {
    return {
      postId: null
    };
  },
  created() {
    this.postId = this.$route.params.id;
  },
  beforeRouteUpdate(to, from, next) {
    this.postId = to.params.id;
    next();
  }
```

```
};
</script>
```

The `beforeRouteUpdate` hook will be called whenever the route parameters change. In this example, we update the `postId` data property whenever the route parameter changes, allowing the component to react to the new value.

Route parameters are a powerful feature that enables you to create flexible and dynamic routes in your Vue.js application. They are commonly used when you need to show different content or data based on the URL, such as displaying specific blog posts, products, or user profiles. Understanding and using route parameters will expand your ability to build more interactive and data-driven applications with Vue Router.

Query Parameters

Query parameters allow you to pass additional information in the URL. Unlike route parameters, query parameters are optional and are added after the route's path using the question mark (`?`). Multiple query parameters are separated by ampersands (`&`).

Defining Routes with Query Parameters

Let's add a route that accepts a query parameter for a search feature.

```
// main.js
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  { path: '/blog/:id', component: BlogPost },
  { path: '/search', component: SearchResults }
];
```

In this example, we have added a new route `/search`, which will display search results based on the provided query parameters.

Using Query Parameters

To use query parameters, you can access them in your component using `this.$route.query`.

```
<!-- SearchResults.vue -->
<template>
```

```
  <div>
    <h1>Search Results</h1>
    <p>Searching for: {{ searchTerm }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      searchTerm: null
    };
  },
  created() {
    this.searchTerm = this.$route.query.q;
  }
};
</script>
```

Now, if you visit `/search?q=vue`, the `SearchResults` component will render, and it will show "Searching for: vue".

Reacting to Query Parameter Changes

Similar to route parameters, you can also react to changes in query parameters using the `beforeRouteUpdate` hook.

```
<!-- SearchResults.vue -->
<script>
export default {
  data() {
    return {
      searchTerm: null
    };
  },
  created() {
    this.searchTerm = this.$route.query.q;
  },
  beforeRouteUpdate(to, from, next) {
    this.searchTerm = to.query.q;
    next();
  }
};
</script>
```

The `beforeRouteUpdate` hook will be called whenever the query parameters change. In this example, we update the `searchTerm` data property whenever the query parameter changes, allowing the component to react to the new value.

Query parameters are useful when you want to provide additional information to the component without affecting the URL's structure. They are commonly used for search filters, pagination, sorting, and other optional features in your application. By understanding and using query parameters, you can make your Vue.js application more dynamic and user-friendly, as it allows users to interact with the application by simply modifying the URL.

Nested Routes

Nested routes allow you to create complex page layouts with multiple components nested inside one another. This is useful when you have a parent component that serves as a layout or container, and you want to display different child components based on the URL.

Defining Nested Routes

Let's create a parent component called `Blog` that will have nested routes for individual blog posts.

```
// main.js
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  {
    path: '/blog',
    component: Blog,
    children: [
      { path: '', component: BlogList },
      { path: ':id', component: BlogPost }
    ]
  }
];
```

In this example, we added a parent route `/blog`, which will render the `Blog` component. Inside the `Blog` component, there are two nested routes: one for the blog list and another for individual blog posts.

Displaying Nested Routes

To display nested routes, you need to use the `<router-view>` component in the parent component's template.

```
<!-- Blog.vue -->
<template>
  <div>
```

```
    <h1>Blog</h1>
    <router-view></router-view>
  </div>
</template>
```

The `<router-view>` here will render the matched child component based on the current URL.

Creating Child Components:

Create the child components that will be rendered inside the `Blog` component.

```
<!-- BlogList.vue -->
<template>
  <div>
    <h2>Blog List</h2>
    <!-- Display a list of blog posts -->
  </div>
</template>
```

```
<!-- BlogPost.vue -->
<template>
  <div>
    <h2>Blog Post</h2>
    <p>Post ID: {{ $route.params.id }}</p>
    <!-- Display the content of the individual blog post -->
  </div>
</template>
```

In the `BlogList.vue` component, you can display a list of blog posts, and in the `BlogPost.vue` component, you can display the content of the individual blog post.

Navigating to Nested Routes

To navigate to nested routes, you need to use `<router-link>` components with appropriate `to` props.

```
<!-- BlogList.vue -->
<template>
  <div>
    <h2>Blog List</h2>
    <ul>
      <li>
        <router-link to="/blog/1">Blog Post 1</router-link>
      </li>
      <li>
        <router-link to="/blog/2">Blog Post 2</router-link>
```

```
      </li>
      <!-- Add more blog posts -->
    </ul>
  </div>
</template>
```

In this example, clicking on a blog post link will navigate to the corresponding `BlogPost` component and display the content of that blog post.

Route Guards

Route guards are functions that allow you to control the navigation behavior of your Vue.js application. They provide ways to guard or protect routes, allowing you to implement features like authentication, authorization, and data validation before a route is accessed.

Global Before Guards

Global before guards are applied to all routes in your application. They are useful for implementing authentication checks before accessing any route.

```
// main.js
const router = new VueRouter({
  routes
});

router.beforeEach((to, from, next) => {
  // Check if the user is authenticated
  if (to.meta.requiresAuth && !isLoggedIn()) {
    // Redirect to login page if not authenticated
    next('/login');
  } else {
    // Proceed to the next route
    next();
  }
});
```

In this example, we're using a global before guard to check if the route requires authentication (specified by the `requiresAuth` meta field) and if the user is logged in. If the user is not authenticated, we redirect them to the login page; otherwise, we proceed to the next route.

Per-Route Before Guards

Per-route before guards are applied to specific routes. They allow you to implement route-specific checks and validations.

```
// main.js
const routes = [
  // ...
  {
    path: '/admin',
    component: AdminDashboard,
    beforeEnter: (to, from, next) => {
      // Check if the user is an admin
      if (isAdmin()) {
        // Proceed to the admin dashboard
        next();
      } else {
        // Redirect to the home page if not an admin
        next('/');
      }
    }
  }
];
```

In this example, we're using a per-route before guard for the `/admin` route to check if the user is an admin. If the user is an admin, they can access the admin dashboard; otherwise, they will be redirected to the home page.

After Navigation Hooks

After navigation hooks are used to perform actions after a route has been accessed.

```
// main.js
const routes = [
  // ...
  {
    path: '/profile',
    component: UserProfile,
    meta: { requiresAuth: true }
  }
];

router.afterEach((to, from) => {
  // Log the navigation
  console.log(`Navigated from ${from.path} to ${to.path}`);
});
```

In this example, we're using the `afterEach` hook to log the navigation from the previous route (`from`) to the current route (`to`). This can be useful for analytics or debugging purposes.

Route guards are essential for managing the flow of your application and ensuring that certain conditions are met before users can access specific routes. By implementing route guards, you can create secure and user-friendly Vue.js applications that provide a seamless navigation experience.

Lazy Loading Routes

Lazy loading routes is a technique to optimize the initial loading time of your Vue.js application by loading components only when they are needed. This can significantly improve the performance of your application, especially if it has a large number of components.

Lazy Loading with Dynamic Imports

To lazy load a route, you can use dynamic imports in your route configuration.

```
// main.js
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  {
    path: '/blog/:id',
    component: () => import('./components/BlogPost')
  },
  {
    path: '/contact',
    component: () => import('./components/ContactForm')
  }
];
```

In this example, we used dynamic imports (using `import()`) to load the `BlogPost` and `ContactForm` components only when the corresponding routes are accessed. The components will be fetched asynchronously, reducing the initial bundle size.

Grouping Components with Lazy Loading

You can also lazy load multiple components by grouping them in the same chunk.

```
// main.js
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  {
    path: '/admin',
    component: () => import(/* webpackChunkName: "admin" */ './components/Admin'),
    children: [
      {
```

```
        path: 'dashboard',
        component: () => import(/* webpackChunkName: "admin" */ './components/AdminDas
hboard')
      },
      {
        path: 'settings',
        component: () => import(/* webpackChunkName: "admin" */ './components/AdminSet
tings')
      }
    ]
  }
];
```

In this example, we are lazy loading the `Admin`, `AdminDashboard`, and `AdminSettings` components as a group under the `admin` chunk. When the `/admin` route is accessed, all these components will be fetched together in a single chunk.

Lazy Loading with Named Chunks

You can give a specific name to the chunk, making it easier to identify in the build output.

```
// main.js
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  {
    path: '/dashboard',
    component: () => import(/* webpackChunkName: "dashboard" */ './components/Dashboar
d')
  },
  {
    path: '/analytics',
    component: () => import(/* webpackChunkName: "dashboard" */ './components/Analytic
s')
  }
];
```

In this example, we used the `webpackChunkName` comment to give a named chunk called `dashboard` to both the `Dashboard` and `Analytics` components. This helps in debugging and understanding the bundle output during development.

Lazy loading routes allows your application to load only the required components, making the initial loading faster and improving the user experience. As users navigate through the application, the necessary components are loaded on-demand, reducing the overall bundle size and making your Vue.js application more efficient.

Named Routes

Named routes are helpful when you want to give a name to a specific route. Instead of hardcoding URLs in your components, you can use the route names to navigate to different routes programmatically.

Defining Named Routes

To create a named route, you simply assign a `name` property to the route object.

```
// main.js
const routes = [
  { path: '/', component: Home, name: 'home' },
  { path: '/about', component: About, name: 'about' },
  {
    path: '/blog/:id',
    component: BlogPost,
    name: 'blog'
  }
];
```

In this example, we gave names to the routes for the home, about, and blog pages.

Using Named Routes

You can use the `name` property to navigate to a specific route using `<router-link>` or programmatically with `this.$router.push`.

```
<!-- Example.vue -->
<template>
  <div>
    <router-link :to="{ name: 'home' }">Home</router-link>
    <router-link :to="{ name: 'about' }">About</router-link>
    <router-link :to="{ name: 'blog', params: { id: 123 } }">Blog Post</router-link>
  </div>
</template>
```

In this example, we used `<router-link>` to navigate to the named routes. The `:to` prop is bound to an object containing the `name` property for each route. For the blog post link, we also provided a route parameter `id` in the `params` object.

Programmatic Navigation with Named Routes

You can use the `name` property with `this.$router.push` to navigate programmatically.

```
<!-- Example.vue -->
<script>
export default {
  methods: {
    goToAboutPage() {
      this.$router.push({ name: 'about' });
    },
    goToBlogPost(id) {
      this.$router.push({ name: 'blog', params: { id } });
    }
  }
};
</script>
```

In this example, we created two methods, `goToAboutPage` and `goToBlogPost`, that use the named routes to navigate to the about page and a specific blog post, respectively.

Named routes make your code more maintainable and readable. They help avoid hardcoding URLs throughout your application, making it easier to change routes in the future without affecting the entire codebase. Additionally, using named routes promotes a more semantic and intuitive way of navigating in your Vue.js application.

Route Transitions

Route transitions allow you to add animations to the transition between different routes in your Vue.js application. This makes the user experience more engaging and visually appealing.

Adding Transitions with `<transition>`

To create route transitions, you can use Vue's built-in `<transition>` component.

```
<!-- App.vue -->
<template>
  <div>
    <transition name="fade" mode="out-in">
      <router-view></router-view>
    </transition>
  </div>
</template>
```

In this example, we wrap the `<router-view>` with the `<transition>` component. The `name` attribute is set to "fade," which will be used to apply CSS transitions between

route changes. The `mode="out-in"` attribute ensures that the leaving component transitions out before the entering component transitions in.

Adding CSS Transitions

To apply the actual CSS transitions, you need to define them in your styles.

```
<!-- App.vue -->
<style>
.fade-enter-active,
.fade-leave-active {
  transition: opacity 0.5s;
}

.fade-enter,
.fade-leave-to {
  opacity: 0;
}
</style>
```

In this example, we use the `fade-enter-active` and `fade-leave-active` classes to set the transition duration and property. The `fade-enter` and `fade-leave-to` classes are used to set the initial and final states of the transition.

Now, when navigating between routes, the old component will fade out, and the new component will fade in with a smooth transition.

Adding Different Transitions

You can apply different transitions to different routes by adding unique transition names and defining their corresponding CSS rules.

```
<!-- App.vue -->
<template>
  <div>
    <transition :name="currentTransition" mode="out-in">
      <router-view></router-view>
    </transition>
  </div>
</template>

<script>
export default {
  data() {
    return {
      currentTransition: 'fade'
    };
  },
  watch: {
```

```
    '$route'(to, from) {
      // Detect the route change and set the appropriate transition
      this.currentTransition = to.meta.transition || 'fade';
    }
  }
};
</script>
```

In this example, we added a `currentTransition` data property and a watcher on `$route`. The watcher detects the route change and checks if the route has a `meta.transition` property. If it does, the custom transition name specified in `meta.transition` will be used. Otherwise, it falls back to the default "fade" transition.

```
<!-- About.vue -->
<script>
export default {
  metaInfo: {
    // Add a meta tag for custom transition
    transition: 'slide'
  }
};
</script>
```

In this example, we set a custom transition named "slide" for the `About` component by adding the `metaInfo` property with the `transition` attribute.

Route transitions enhance the user experience by adding visually appealing animations during route changes. They help to create a more polished and professional feel for your Vue.js application. By applying different transitions to different routes, you can customize the animations to match the content and context of each page, making the transitions more meaningful and impactful.

Programmatic Navigation

Programmatic navigation allows you to navigate between routes in your Vue.js application using JavaScript code. This is useful when you want to navigate based on user interactions, form submissions, or other events within your application.

Navigating with `this.$router.push`

To navigate programmatically, you can use the `this.$router.push` method.

```
<!-- Example.vue -->
<template>
  <div>
```

```
    <button @click="goToAboutPage">Go to About</button>
    <button @click="goToBlogPost(123)">Go to Blog Post</button>
  </div>
</template>

<script>
export default {
  methods: {
    goToAboutPage() {
      this.$router.push('/about');
    },
    goToBlogPost(id) {
      this.$router.push(`/blog/${id}`);
    }
  }
};
</script>
```

In this example, we use the `this.$router.push` method to navigate to the `/about` route when the "Go to About" button is clicked. We also use it to navigate to a specific blog post using the `id` parameter when the "Go to Blog Post" button is clicked.

Navigating with `this.$router.replace`

The `this.$router.replace` method is similar to `this.$router.push`, but it replaces the current history entry instead of adding a new one.

```
<!-- Example.vue -->
<script>
export default {
  methods: {
    goToLoginPage() {
      this.$router.replace('/login');
    }
  }
};
</script>
```

In this example, when the "Go to Login Page" button is clicked, we use `this.$router.replace` to navigate to the `/login` route and replace the current history entry.

Navigating with `this.$router.go`

The `this.$router.go` method allows you to navigate to a specific history entry. Positive integers go forward in history, and negative integers go back.

```
<!-- Example.vue -->
<template>
  <div>
    <button @click="goBack">Go Back</button>
    <button @click="goForward">Go Forward</button>
  </div>
</template>

<script>
export default {
  methods: {
    goBack() {
      this.$router.go(-1);
    },
    goForward() {
      this.$router.go(1);
    }
  }
};
</script>
```

In this example, when the "Go Back" button is clicked, we use `this.$router.go(-1)` to navigate back in history, and when the "Go Forward" button is clicked, we use `this.$router.go(1)` to navigate forward in history.

Programmatic navigation gives you full control over the navigation flow in your Vue.js application. You can use it to create dynamic and interactive user experiences, where the navigation is driven by the user's actions and interactions. By using these methods, you can navigate to different routes, replace history entries, and move back and forth through the navigation history, making your application more user-friendly and responsive.

Dynamic Route Matching

Dynamic route matching allows you to create routes that change based on the component's data or props. It's useful when you want to display different content or layouts depending on the data provided to the component.

Dynamic Routes with Props:

To create dynamic routes, you can use the `props` option in your route configuration.

```
// main.js
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  {
    path: '/user/:id',
```

```
    component: UserProfile,
    props: true
  }
];
```

In this example, we added a dynamic route `/user/:id`, where the `id` is a parameter that can change. We set `props: true` to automatically pass the route parameters as props to the `UserProfile` component.

Using Props in the Component

Now, the `UserProfile` component can access the `id` parameter as a prop.

```
<!-- UserProfile.vue -->
<template>
  <div>
    <h2>User Profile</h2>
    <p>User ID: {{ id }}</p>
    <!-- Display user information based on the ID -->
  </div>
</template>

<script>
export default {
  props: ['id']
};
</script>
```

In this example, the `UserProfile` component uses the `id` prop to display user information based on the provided `id`.

Dynamic Routes with Function-based Props

If you need more control over the props, you can define a function to map the route parameters to props.

```
// main.js
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
  {
    path: '/product/:id',
    component: ProductDetails,
    props: (route) => ({ id: Number(route.params.id) })
  }
];
```

In this example, we defined a function for the `props` option to explicitly cast the `id` parameter to a number before passing it to the `ProductDetails` component.

```
<!-- ProductDetails.vue -->
<template>
  <div>
    <h2>Product Details</h2>
    <p>Product ID: {{ id }}</p>
    <!-- Display product information based on the ID -->
  </div>
</template>

<script>
export default {
  props: ['id']
};
</script>
```

The `ProductDetails` component uses the `id` prop to display product information based on the provided `id`.

Dynamic route matching allows you to create more flexible and data-driven routes in your Vue.js application. By using dynamic routes, you can build components that adapt to different data and display relevant information based on the route parameters. This makes your application more versatile and capable of handling various scenarios with ease.