Due Apr 15, 1159PM

Contents

[Lazy Loaded Modules 1](#_Toc499616978)

[Modules 3](#_Toc499616979)

[Events 8](#_Toc499616980)

[Mouse Enter and Leave 8](#_Toc499616981)

[Key Down and Key Up 9](#_Toc499616982)

[Focus and Blur 10](#_Toc499616983)

[Copy/Cut and Paste 11](#_Toc499616984)

[Mouse Down and Mouse Up 13](#_Toc499616985)

[Attribute Directives 15](#_Toc499616986)

[Vue 17](#_Toc499616987)

[Vue Routing 17](#_Toc499616988)

[Vue Router with Params 20](#_Toc499616989)

[Asynchronous Calls 22](#_Toc499616990)

## Lazy Loaded Modules

If you recall our routing example earlier in the course, if you check the sources tab of the debugger when running it, you will notice that loading one module loads all components within it. This is not always a bad option and is probably just fine for all applications that you might build in SSD. However, for really large applications users may have to wait for an unpleasant amount of time for the application to start in their web browser. While this situation is not so regular, components can be grouped within different modules and these modules can be loaded on demand. Loading modules when they are needed is also called Lazy Loading.

Example : Lazy Loaded Modules

This example shows how to implement lazy loading so a module which groups different color components is not actually fetched by the browser until the user requests it.

**Part A: Review Code**

In this first section, we will build the basic routing application that was discussed in example 8 of day 2. The code is then transformed to implement hyperlinks that retrieve and display different child views when clicked. The enable these views, add these three components which each have inline views.

**app/app.page-a.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is page A.`  })  export class PageAComponent { } |

**app/app.page-b.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is another page.`  })  export class PageBComponent { } |

**app/app.pagedefault.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This page does not exist.`  })  export class PageDefault { } |

Next, replace the code inside app.component.ts file with the following code that contains a template which includes a menu with hyperlinks. The hyperlinks are created using the <routerLink> tag with a value of the reference from the router. The <router-outlet> element serves as a placeholder for the child view which appears when the appropriate link is selected.

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  template:  `<h1>This is the header</h1>  <nav>  <a routerLink="/page-a" routerLinkActive="active">A</a> |  <a routerLink="/page-b" routerLinkActive="active">B</a>  </nav>  <!-- Where router should display a view -->  <router-outlet></router-outlet>`  })  export class AppComponent { } |

With a larger project you may want to store the routing logic in a separate file such as in app.routing.ts. The appRoutes array below defines a series of routing objects which include a hyperlink reference and corresponding component displayed. Since the components are referenced by the router they must first be imported. These components have been loaded through the import statements.

**app/app.routing.ts**

|  |
| --- |
| import { ModuleWithProviders } from '@angular/core';  import { Routes, RouterModule } from '@angular/router';  import { AppComponent } from './app.component';  import { PageAComponent } from './app.page-a';  import { PageBComponent } from './app.page-b';  import { PageDefault } from './app.pagedefault';  const appRoutes: Routes = [  { path: 'page-a', component: PageAComponent },  { path: 'page-b', component: PageBComponent },  { path: '', redirectTo: '/page-a', pathMatch: 'full' },  { path: '\*\*', component: PageDefault }  ];  export const routing: ModuleWithProviders = RouterModule.forRoot(appRoutes); |

### Modules

Modules group components together. This not only helps to organize your code, the modularization also allows us to split the application up so it does not need to be loaded all at once.

In app.module.ts, we are declaring AppModule which includes four different components and we are also importing the router defined in the file previously discussed.

**app/app.module.ts**

|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { AppComponent } from './app.component';  import { PageDefault } from './app.pagedefault';  import { PageAComponent } from './app.page-a';  import { PageBComponent } from './app.page-b';  import { routing } from './app.routing';  @NgModule({  imports: [ BrowserModule, routing ],  declarations: [ AppComponent, PageDefault,  PageAComponent, PageBComponent ],  bootstrap: [ AppComponent ],  })  export class AppModule { } |

**Part B: New Code**

This example starts with part A to enable lazy loaded modules. The following components are added inside a new ‘colors’ folder.

**app/colors/colors.red.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is red.`  })  export class RedComponent {  } |

**app/colors/colors.green.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is green.<br/>`  })  export class GreenComponent { } |

**app/colors/colors.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `  <h4>Colors</h4>  <a routerLink="/colors/red" routerLinkActive="active">red</a> |  <a routerLink="/colors/green" routerLinkActive="active">green</a>`  })  export class ColorsComponent { } |

To navigate from one color component to the next we can have a separate router. These new routes are stored inside the *colorsRouting* object.

**app/colors/colors.routing.ts**

|  |
| --- |
| import { ModuleWithProviders } from '@angular/core';  import { Routes, RouterModule } from '@angular/router';  import { GreenComponent } from './colors.green';  import { RedComponent } from './colors.red';  import { ColorsComponent } from './colors.component';  const ROUTES = [  { path: '', component: ColorsComponent, pathMatch: 'full' },  { path: 'green', component: GreenComponent, pathMatch: 'full' },  { path: 'red', component: RedComponent, pathMatch: 'full' }  ];  export const colorsRouting: ModuleWithProviders = RouterModule.forChild(ROUTES); |

When we declare our module we import our colorsRouting object and declare all color components that are grouped by the module.

**colors/colors.module.ts**

|  |
| --- |
| import { NgModule, Component } from '@angular/core';  import { CommonModule } from '@angular/common';  import { GreenComponent } from './colors.green';  import { RedComponent } from './colors.red';  import { ColorsComponent } from './colors.component';  import { colorsRouting } from './colors.routing';  @NgModule({  // Import routing for this module and make available for all components.  imports: [  CommonModule,  colorsRouting  ],  // Declare components grouped by this module.  declarations: [  ColorsComponent,  GreenComponent,  RedComponent  ]  })  export class ColorsModule { } |

Now we can modify our parent route by referencing our color module as shown by the code highlighted in yellow. This is telling the application not to load the module until the user clicks on the colors link.

**app/app.routing.ts**

|  |
| --- |
| import { ModuleWithProviders } from '@angular/core';  import { Routes, RouterModule } from '@angular/router';  import { AppComponent } from './app.component';  import { PageAComponent } from './app.page-a';  import { PageBComponent } from './app.page-b';  import { PageDefault } from './app.pagedefault';  import { ColorsModule } from './colors/colors.module';  const subroutes: Routes = [  { path: 'page-a', component: PageAComponent },  { path: 'page-b, component: PageBComponent },  { path: '', redirectTo: '/page-a', pathMatch: 'full' },  // This says load the ColorsModule which includes color components.  { path: 'colors', loadChildren: 'app/colors/colors.module#ColorsModule' },  { path: '\*\*', component: PageDefault },  ];  export const routing: ModuleWithProviders = RouterModule.forRoot(subroutes); |

To give users access to the colors link it can be added to the AppComponent which is loaded when the application starts up.

**app/app.component.ts**

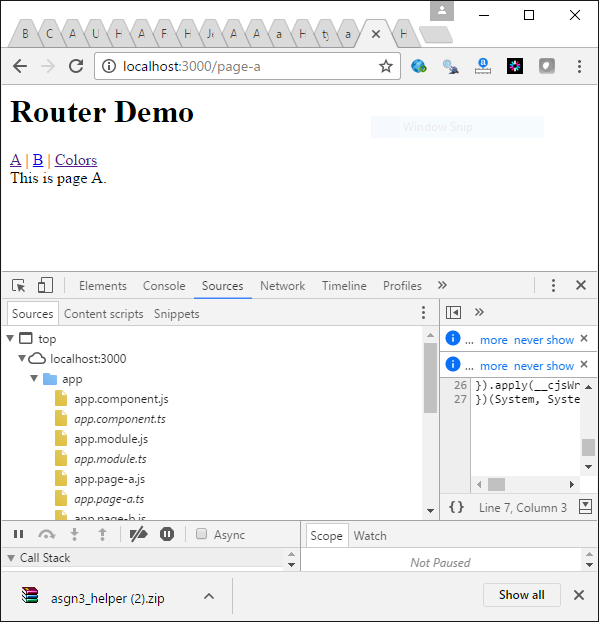
|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template:  `<h1>Router Demo</h1>  <nav>  <a routerLink="/page-a" routerLinkActive="active">A</a> |  <a routerLink="/page-b " routerLinkActive="active">B</a> |  <!-- Link to ColorsModule area -->  <a routerLink="/colors" routerLinkActive="active">Colors</a>  </nav>  <!-- Where router should display a view -->  <router-outlet></router-outlet>`  })  export class AppComponent { } |

Finally, check to ensure the index.html page already contains a base href tag define the application root for the router.

**index.html**

|  |
| --- |
| <!doctype html>  <html lang="en">  <head>  <meta charset="utf-8">  <title>Myapp</title>  <base href="/">  <meta name="viewport" content="width=device-width, initial-scale=1">  <link rel="icon" type="image/x-icon" href="favicon.ico">  </head>  <body>  <app-root></app-root>  </body>  </html> |

If you run this app and refresh the browser you will notice that the colors folder does not show up in the debugger. Show a screenshot of the ‘Sources’ tab



Exercise

If you click on the colors link you the colors folder will appear underneath the app folder. Show a screenshot of this.

|  |
| --- |
|  |

Exercise

What does it mean to lazy load modules?

|  |
| --- |
| components can be grouped within different modules like **app.routing.ts** and **colors.routing.ts** these modules can be loaded on demand. Loading modules when they are needed is also called Lazy Loading. |

## Events

Most of the standard events that are available in JavaScript are also available through Angular.

<https://developer.mozilla.org/en-US/docs/Web/Events>

Angular typically exposes these JavaScript events within round braces (). Including the $event object as a parameter exposes properties that are specific to the current event.

### Mouse Enter and Leave

Handling mouse entry and leave events is simple.

Example : Mouse Entry

This example shows just how easy capturing a mouse enter event is. In a brand new Angular application replace the contents of app.component.ts with the following code:

**app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'app';  myMouseEnter() {  console.log("Mouse entered!")  }  } |

Here is the HTML needed which contains the mouse handler.

**app.component.html**

|  |
| --- |
| <h1>  Welcome to **{{**title**}}**!  </h1>  <input type="submit" value="my button" (mouseenter)="myMouseEnter()"> |

Exercise

Starting with Example 2, add a mouseleave handler and log the output to show when the event has taken place. Show your revised app.component.ts file here:

|  |
| --- |
| **myMouseEnter**() {  **console**.**log**("Mouse entered!")  }  **myMouseLeave**() {  **console**.**log**("Mouse left!")  } |

Show your revised HTML file here:

|  |
| --- |
| <input type="submit" value="my button" (mouseenter)="myMouseEnter()" (mouseleave)="myMouseLeave()"> |

### Key Down and Key Up

Key Down and Key Up handlers make good use of the optional $event object which contains properties that are specific to that event.

<input type='text' (keydown)="myKeyDown($event)">

Example : Key Down Handling

To handle the key down event, in a brand new Angular application replace the contents of app.component.ts with the following code:

**app.component.ts**

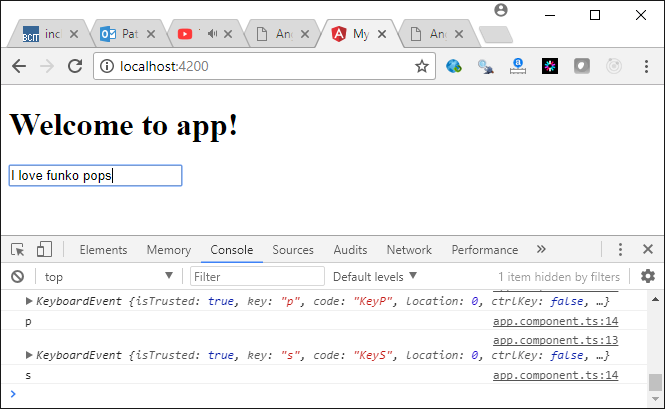
|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: [ './app.component.css']  })  export class AppComponent {  title = 'app';  myKeyDown(event) {  console.log(event)  console.log(event.key)  }  } |

Here is the HTML needed which contains (keydown) handler:

**app.component.html**

|  |
| --- |
| <h1>  Welcome to **{{**title**}}**!  </h1>  <input type='text' (keydown)="myKeyDown($event)"> |

When you run this example, notice how the keys pressed are captured in the output that is written to the console.



Exercise

In the debugger window, expand the KeyboardEvent output from Example 3 to see which other properties are available through the $event object. Show a screenshot of this view.

|  |
| --- |
|  |

Exercise

In addition to having a key down handler, add a key up event handler to Example 3 to display the key released during an up stroke. Show your revised app.component.ts file here:

|  |
| --- |
| **myKeyDown**(event) {  **console**.**log**(event)  **console**.**log**(event.key)  }  **myKeyUp**(event) {  **console**.**log**(event)  **console**.**log**(event.key)  } |

Show your revised app.component.html file here:

|  |
| --- |
| key down: <input type='text' (keydown)="myKeyDown($event)"> <br>  key up: <input type='text' (keyup)="myKeyUp($event)"> |

### Focus and Blur

A focus event is raised whenever a control is given focus. A blur event occurs when the control loses focus.

Example : Focus Event Handling

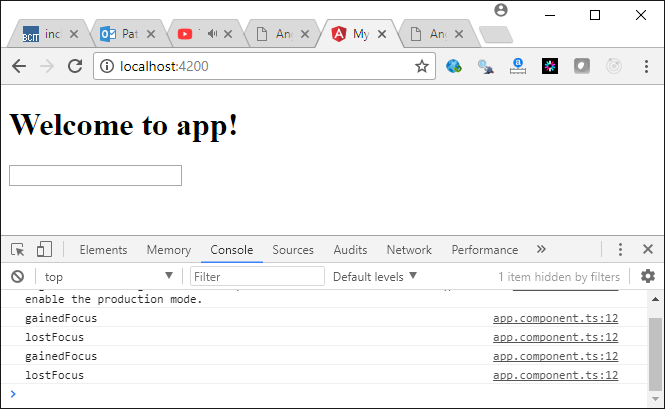
This sample shows how you could handle a focus event. This is the app.component.ts file:

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'app';  myFocusEvent(description) {  console.log(description)  }  } |

Here is the app.component.html file:

|  |
| --- |
| <h1>  Welcome to **{{**title**}}**!  </h1>  <input type='text' (focus)="myFocusEvent('gainedFocus')" (blur)="myFocusEvent('lostFocus')"> |

When running the application, you will notice the logged output shows when the input gains and loses focus.



### Copy/Cut and Paste

Cut or Copy and paste are also easy to implement. Here is a sample below.

Example : Cut / Copy and Paste

This example demonstrates how to implement Cut/Copy and Paste from a brand new Angular project. ngModel is being used so the FormsModule must be imported through app.module.ts:

**app.module.ts**

|  |
| --- |
| import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { AppComponent } from './app.component';  @NgModule({  declarations: [  AppComponent  ],  imports: [  BrowserModule, FormsModule  ],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule { } |

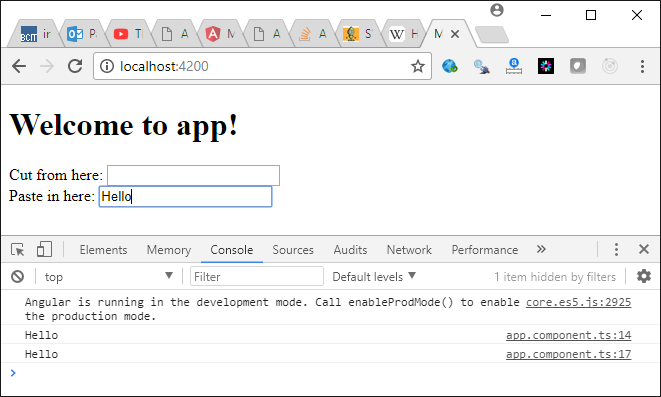
The (cut) and (paste) event handlers are placed in an input control inside app.component.ts:

|  |
| --- |
| <h1>  Welcome to **{{**title**}}**!  </h1>  Cut from here:  <input [(ngModel)]="myContent" type='text' (cut)="cutEvent(myContent)"><br />  Paste in here:  <input type='text' [(ngModel)]="myPastedContent" (paste)="pasteContent(myPastedContent)"> |

Here is the code for app.component.html to handle the events. There could be a nicer way to do this but this version is not too complicated:

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'app';  cutContent = '';  cutEvent(content) {  this.cutContent = content;  console.log(content)  }  pasteContent() {  console.log(this.cutContent);  this.cutContent = '';  }  } |

The output shows changes:



With a one word change this example can be transformed to handle copy instead of cut.

|  |
| --- |
| <h1>  Welcome to **{{**title**}}**!  </h1>  Cut from here:  <input [(ngModel)]="myContent" type='text' (copy)="cutEvent(myContent)"><br />  Paste in here:  <input type='text' [(ngModel)]="myPastedContent" (paste)="pasteContent(myPastedContent)"> |

### Mouse Down and Mouse Up

Same as other events before, mouse down and up events are easy to handle with the $event object.

Example : Mouse Down and Up

For this case, the location of the mouse during down and up events is tracked.

**app.component.html**

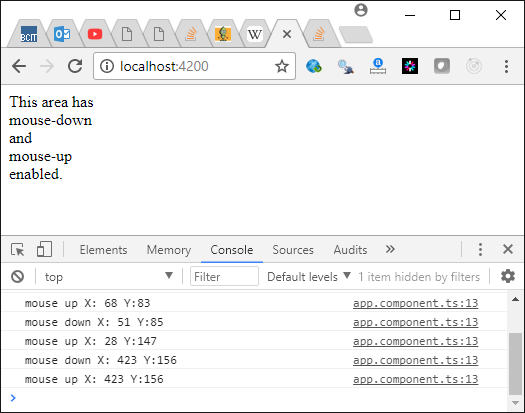
|  |
| --- |
| <div (mousedown)="myMouseHandler($event, 'mouse down')" (mouseup)="myMouseHandler($event, 'mouse up')">  This area has<br />  mouse-down<br />  and<br />  mouse-up<br />  enabled.  </div> |

The position of the mouse cursor is accessed through screenX and screenY properties that are associated with these events.

**app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'app';  myMouseHandler(event, description) {  console.log(description + " X: "  + event.screenX.toString()  + " Y:" + event.screenY.toString())  }  } |

Here is the output:



Exercise

Add a mouse move handler to Example 7. Show your revised HTML file here.

|  |
| --- |
| **myMouseMove**(event, description1, description2) {  **console**.**log**(description1 + " X: "  + event.pageX.**toString**() + description2 + " Y: "  + event.pageY.**toString**())  }  <div (mousemove)="myMouseMove($event, 'mouse down', 'mouse up')">  This area has<br />  mouse-down<br />  and<br />  mouse-up<br />  enabled.  </div> |

Show a screen shot of the output which displays the x and y position of the mouse pointer.

|  |
| --- |
|  |

## Attribute Directives

Until now we have created our own custom directives using component directives. Attribute directives are a little different. Attribute directives allow you to modify or change the behavior of a custom HTML element by adding a custom directive to it.

Example : Attribute Directive for Element Highlighting

This example shows how you can create an attribute directive which can be added to an element to produce highlighting during mouse over activity.

Angular provides a HostListener dependency to allow you to capture any document, window and body event that can be handled by JavaScript. For this example, we also need the ElementRef dependency so we can obtain a reference to the element which contains our custom directive.

**app.highlight.directive.ts**

|  |
| --- |
| // Directive allows use of the @Directive decorator.  // ElementRef allows access to element in HTML.  import { Directive, ElementRef, HostListener, Input } from '@angular/core';  @Directive({  // Name the attribute.  selector: '[myHighlight]'  })  export class HighlightDirective {  constructor(private el: ElementRef) { }  // Pass color from 'myHighlight' attribute to 'highlightColor' model.  @Input('myHighlight')  highlightColor: string;  // Set color on mouse entry. 'red' is the default.  @HostListener('mouseenter')  onMouseEnter() {  this.highlight( this.highlightColor || 'red');  }  // Set color to null on mouse exit.  @HostListener('mouseleave')  onMouseLeave() {  this.highlight(null);  }  private highlight(color: string) {  this.el.nativeElement.style.backgroundColor = color;  }  } |

After we need to tell the module about our new attribute directive.

**app.module.ts**

|  |
| --- |
| import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { HighlightDirective } from './app.highlight.directive';  import { AppComponent } from './app.component';  @NgModule({  declarations: [  AppComponent,HighlightDirective  ],  imports: [  BrowserModule  ],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule { } |

Next, we can add our custom attribute to a paragraph tag.

**app.component.html**

|  |
| --- |
| <p [myHighlight]>Highlight me!</p>  <p [myHighlight]="'violet'">  Highlight me too!  </p> |

Here is our app.component.ts file which references app.component.html

**app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'app';  } |

Exercise

Add a third element to app.component.html which references the attribute directive to color itself green. Show your revised app.component.html file here:

|  |
| --- |
|  |

Exercise

Create the same effect as the attribute directive in Example 8 but this time do it with a component directive which nests its content inside a div tag. Show your revised directive file here:

|  |
| --- |
|  |

Show your revised app.component.html file:

|  |
| --- |
|  |

Show your revised app.component.ts file:

|  |
| --- |
|  |

# Vue

Recalling from last week, in case you are working on a lab PC you can obtain a copy of Vue by running the command:

**>npm install -g vue-cli**

This is the command for creating a new Vue application.

**>vue init webpack starterapp**

Once you set up the Vue templates you will need to run the install command from the directory where the package.json file exists.

**>npm install**

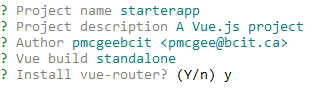
To build and run the vue application run the command:

**>npm run dev**

## Vue Routing

Example : Vue Routing

This example shows how to add in basic routing to a Vue application. One way to obtain code for the router is to answer ‘y’ when prompted to install the router. So, to build this example, create a new Vue application and respond with ‘y’ at the router install prompt.



Replace the existing code inside HelloWorld.vue with the following code. This new version is a little more simplified than the original version that is included with the auto-generated file.

**components/HelloWorld.vue**

|  |
| --- |
| <template>  <div class="hello">  <h1>{{ msg }}</h1>  <h2>Essential Links</h2>  </div>  </template>  <script>  export default {  name: 'HelloWorld',  data () {  return {  msg: 'Welcome to Your Vue.js App'  }  }  }  </script>  <!-- Add "scoped" attribute to limit CSS to this component only -->  <style scoped>  h1, h2 {  font-weight: normal;  }  a {  color: #42b983;  }  </style> |

Next, create an About component.

**components/About.vue**

|  |
| --- |
| <template>  <div class="about">  <h1>This is the about page.</h1>  </div>  </template>  <script>  export default {  name: 'About'  }  </script> |

This router code shows the changes needed to reference a brand new **About** component with the router. Notice how each component is referenced. Also notice how the path is referenced:

**router/index.js**

|  |
| --- |
| import Vue from 'vue'  import Router from 'vue-router'  import HelloWorld from '@/components/HelloWorld'  import About from '@/components/About'  Vue.use(Router)  export default new Router({  mode: 'history', // Removes the hashbang.  routes: [  {  path: '/',  component: HelloWorld  },  // NEW  {  path: '/about',  component: About  },  ]  }) |

The app.vue file contains the router-links and the router-view where the child view is shown.

**Src/App.vue**

|  |
| --- |
| <template>  <div id="app">  <router-link to="/">Home</router-link>  <router-link to="/about">About</router-link>  <router-view/>  </div>  </template>  <script>  export default {  name: 'app'  }  </script>  <style>  #app {  font-family: 'Avenir', Helvetica, Arial, sans-serif;  }  </style> |

Exercise

What is the difference between the router-link and router-view elements?

|  |
| --- |
|  |

Exercise

Create a contact component. Add a reference to it in the routing file. Adjust the App.vue page so a hyperlink references the new contact page in addition to the existing home and about pages. Show your new component here:

|  |
| --- |
|  |

Show your modified router/index.js file here:

|  |
| --- |
|  |

Show your modified app.vue file here:

|  |
| --- |
|  |

## Vue Router with Params

Example : Vue Routing with Params

This example shows how you could create a Vue application that manages parameters. The routing option that manages a component with parameters is highlighted in turquoise below.

**Router/index.js**

|  |
| --- |
| import Vue from 'vue'  import Router from 'vue-router'  import HelloWorld from '@/components/HelloWorld'  // NEW  import About from '@/components/About'  import Details from '@/components/Details'  Vue.use(Router)  export default new Router({  mode:'history',  routes: [  {  path: '/',  component: HelloWorld  },  {  path: '/about',  component: About  },  {  path: '/details/:id/:day',  component: Details  }  ]  }) |

The parameters are retreived after the created() function is called when the component is initialized. The parameter data can also be retrieved like a model as shown in yellow.

**components/Details.vue**

|  |
| --- |
| <template>  <div>  id: {{id}}<br>  <!-- Reference route parameters -->  day: {{$route.params.day}}<br>  day another way: {{dayofweek}}  </div>  </template>  <script>  export default {  name: 'Details',  data() {  return {  id:'',  dayofweek:''  }  },  // Like 'ngOnInit'. Gets called once object is created.  created() {  this.fetchData()  },  methods: {  fetchData() {  // Store parameters in class.  this.id = this.$route.params.id;  this.dayofweek = this.$route.params.day;  }  }  }  </script> |

Just like In Angular, the parameters are passed through the router-link.

**App.vue**

|  |
| --- |
| <template>  <div id="app">  <router-link to="/">Home</router-link>  <router-link to="/about">About</router-link>  <router-link to="/details/20/Friday">Details</router-link>  <router-view />  </div>  </template>  <script>  export default {  name: 'app'  }  </script>  <style>  #app {  font-family: 'Avenir', Helvetica, Arial, sans-serif;  }  </style> |

## Asynchronous Calls

There are several ways to perform asynchronous calls for data. A popular way to enable the asynchronous behavior with Vue.js is through the Axios promise library.

Axios makes the threaded call and returns valid results within a then() block. Errors are returned to the catch() block.

axios.get('https://api.coinmarketcap.com/v1/ticker/bitcoin/')

.then((resp) => {

this.coinInfo = resp.data[0]

this.currency.id = resp.data[0].id;

this.currency.price\_usd = resp.data[0].price\_usd;

})

.catch((err) => {

console.log(err)

})

Example : Asynchronous Calls

This example demonstrates how to create a project that is able to perform asynchronous calls for remote resources. To do this, you run the following command in the same directory where your package.json file exists.

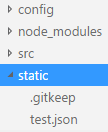
npm install --save axios

Next, replace the HelloWorld.vue component contents with the following code:

|  |
| --- |
| <template>  <div class="hello">  <h1>{{ msg }}</h1>  <b>Coin Type</b> {{currency.id}} <br/>  <b>Price USD</b> {{currency.price\_usd}}  </div>  </template>  <script>  import axios from 'axios'  export default {  name: 'HelloWorld',  data () {  return {  msg: 'Welcome to Your Vue.js App',  currency: {  id:'',  price\_usd:''  }  }  },  created() {  axios.get('https://api.coinmarketcap.com/v1/ticker/bitcoin/')  .then((resp) => {  this.coinInfo = resp.data[0]  this.currency.id = resp.data[0].id;  this.currency.price\_usd = resp.data[0].price\_usd;  })  .catch((err) => {  console.log(err)  })  }  }  </script>  <!-- Add "scoped" attribute to limit CSS to this component only -->  <style scoped>  h1, h2 {  font-weight: normal;  }  </style> |

Exercise

Starting with the code in Example 8, read the contents of test.json dynamically. The test.json file needs to be placed in the static folder. The static folder is already generated automatically when you build the Vue project.



**static/test.json**

|  |
| --- |
| [  { "firstName":"Mary", "lastName":"Jones"},  { "firstName":"Jay", "lastName":"Leno"}  ] |

Next inside **src/components/HelloWorld.vue** add some code to parse and display the response. Here is some hint code to get you started.

|  |
| --- |
| created() {  axios.get('static/test.json')  .then((resp) => {  this.firstName = resp.data[0].firstName;  })  .catch((err) => {  console.log(err)  })  } |

Show your resided complete HelloWorld.vue file when you are finished:

|  |
| --- |
|  |