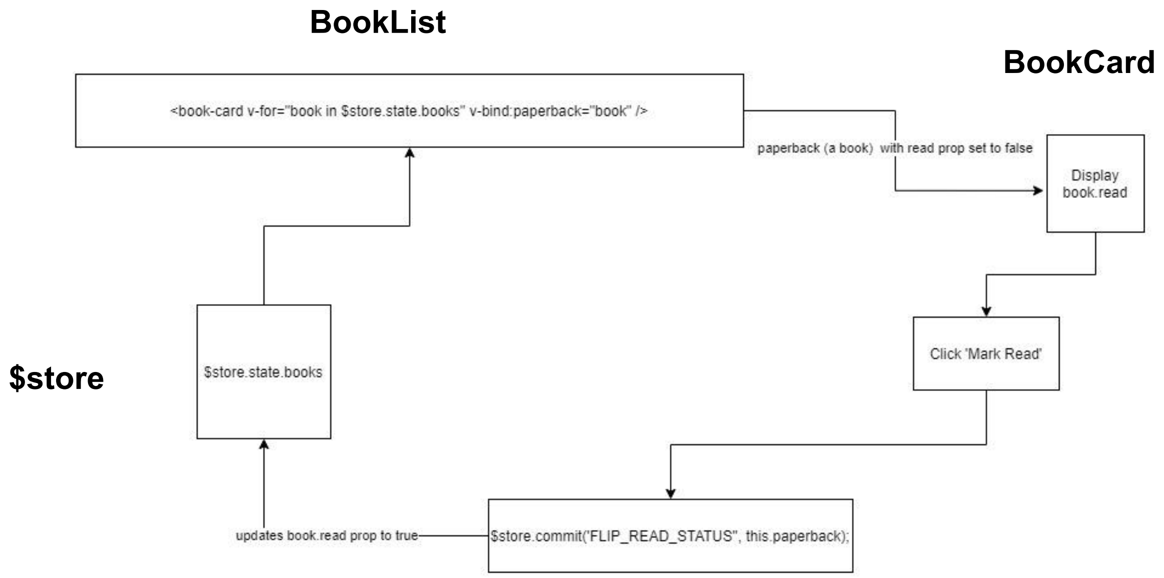
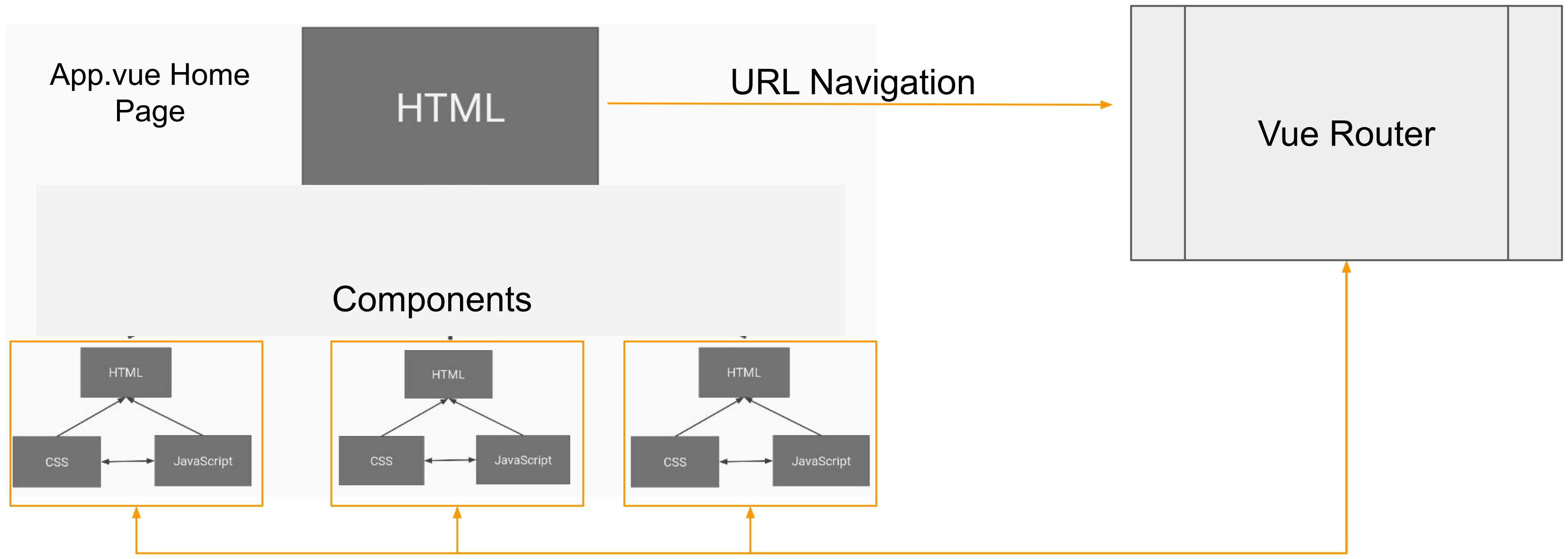
# Module 3
# Week 4 Review

# Diagram of Vue Flow

When the data in the **`$store`** changes it is **automatically** reflected in entire flow

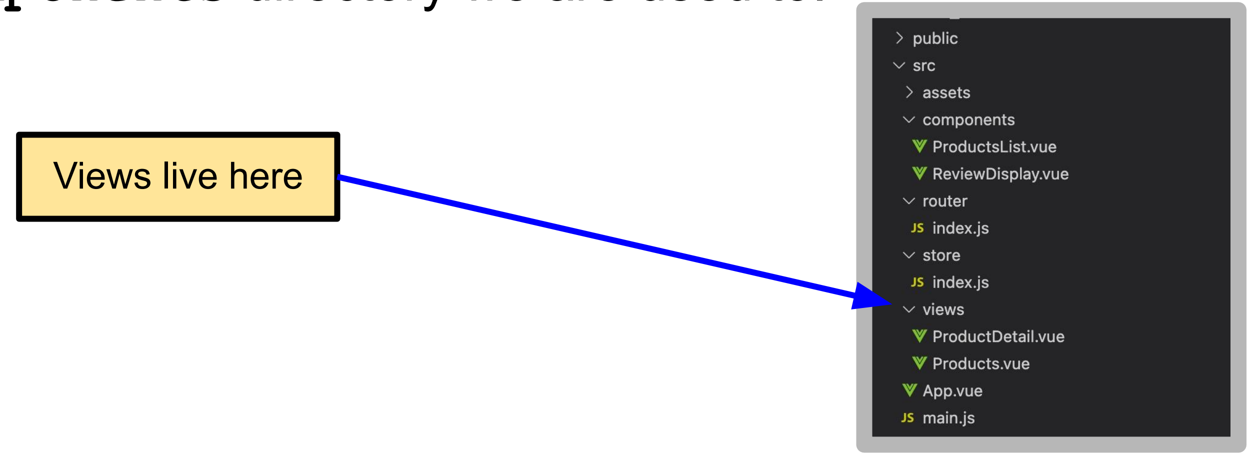

BookList
BookCard
$store
<book-card v-for="book in $store.state.books" v-bind:paperback="book" />
paperback (a book) with read prop set to false
Display book.read
Click 'Mark Read'
$store.commit('FLIP_READ_STATUS", this.paperback);
updates book.read prop to true
$store.state.books

# Vue: Evolution of Concepts - Routing to Site Pages



## Components vs. Views

- **Views** are just components that serve a special purpose: acting as virtual pages.
- Difference between a View and Component is conceptual, aside from the fact that Views live in a **`views`** directory rather than in the **`components`** directory we are used to.



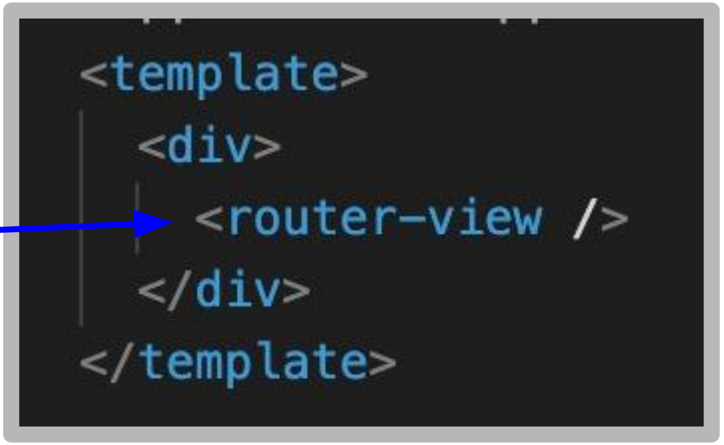
```
> public
v src
  > assets
  v components
    ProductsList.vue
    ReviewDisplay.vue
  v router
    index.js
  v store
    index.js
  v views
    ProductDetail.vue
    Products.vue
  App.vue
  main.js
```

## Using RouterView

- **`RouterView`** is a functional component that renders components (views) for a given path.
- **`RouterView`** does not require view components to be in the views directory, but they should be.

The **`App`** component includes the **`RouterView`** component in its **`<template>`** section.

**`RouterView`** handles all other rendering of components which are views.



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

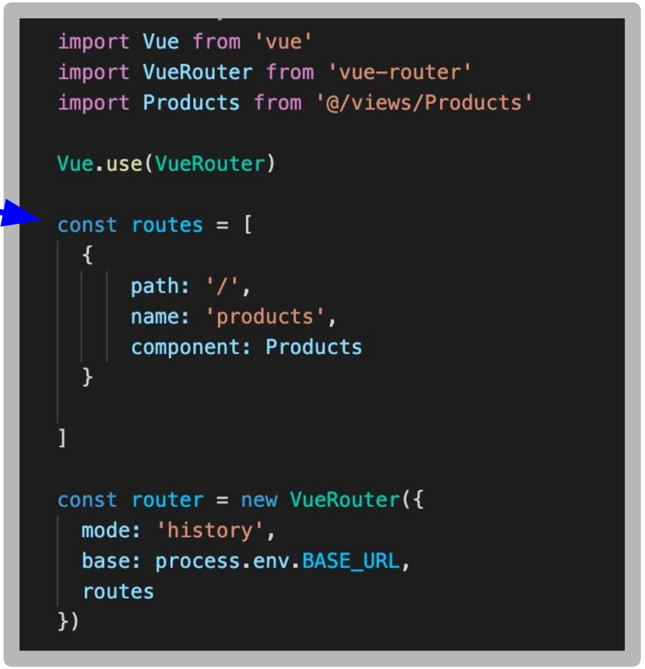
# Defining Routes

- Routes used by **`RouterView`** are defined in src/router/index.js

The **`routes`** array holds objects representing routes.

**`RouterView`** route objects require the route **`path`** and the **`component`** being routed to.

Optionally, a route can be given a **`name`**. *This is best practice.*



```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Products from '@/views/Products'

Vue.use(VueRouter)

const routes = [
  {
    path: '/',
    name: 'products',
    component: Products
  }

]

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})
```

## Dynamic Routing

- `path="/product/:productId"`
- `const product = this.$route.params.productId;`
- `<router-link v-bind:to="{ name: 'home', params: { id: 1234} }" tag="div">Home</router-link>`

## Synchronous vs. Asynchronous Programming

- **Synchronous Programming:**
  - When calling a function or method, the code expects to get the result before the flow of execution moves on to the next line of code.
- **Asynchronous Programming:**
  - When calling a function or method, the call returns right away but the called code continues to run until it completes. If the calling code expected a result, the result data will be resolved once the called code completes and the result is available. Using this approach for web service calls, which can be very slow, helps make code more efficient and responsive.

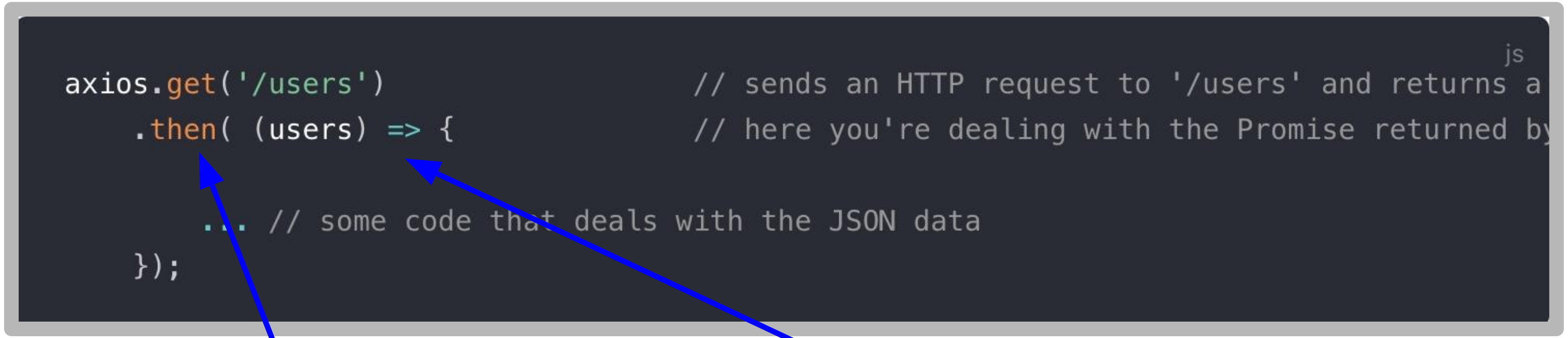
## Using Axios to Make Web API Calls

**Axios** is a library that is used to make calls to Web API services from a JavaScript front-end application.

- `axios.get('/users');`
- `axios.post('/users', newUser);`
- `axios.put('/users', updateUser);`
- `axios.delete('/users/1284');`

Axios make Web API asynchronously. The calls above return a Promise.

# Using Axios to Make Asynchronous Web API Calls



```js
axios.get('/users')                 // sends an HTTP request to '/users' and returns a
    .then( (users) => {             // here you're dealing with the Promise returned by

        ... // some code that deals with the JSON data
    });
```

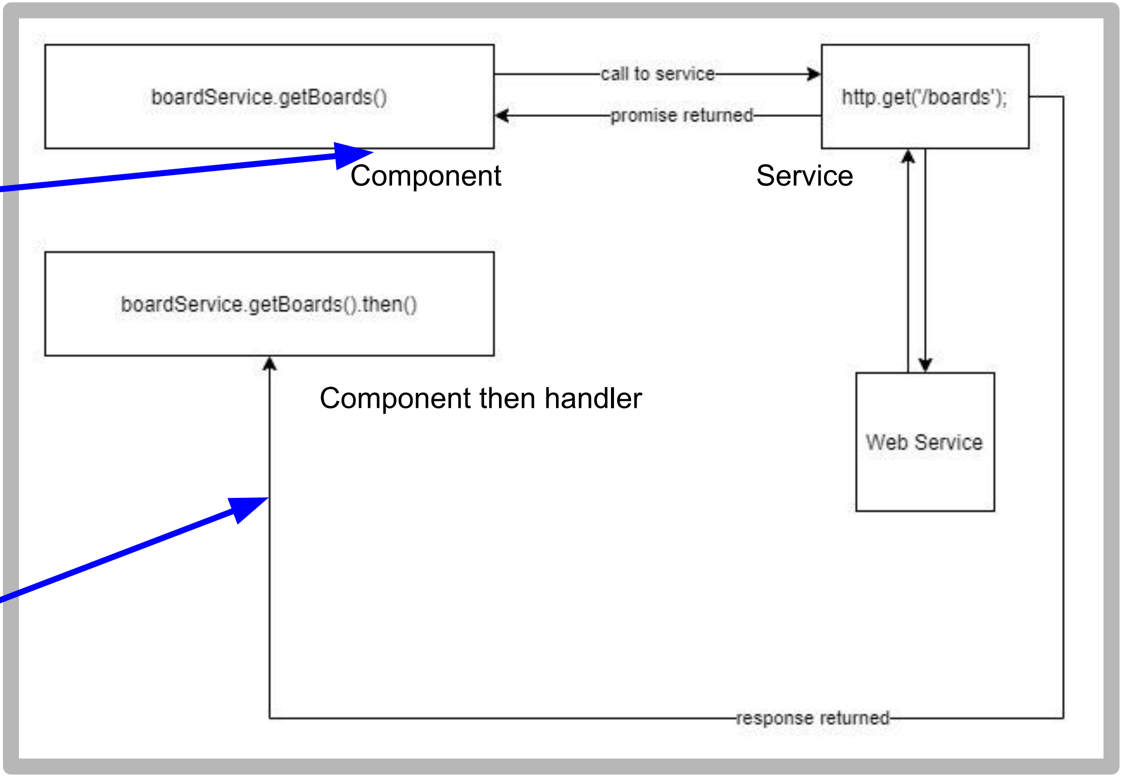
When the HTTP request completes, the **`then`** portion of the call executes..

The Promise returned by **`get()`** resolves, **`users`** contains the information from the response, and the code in the arrow function now has access to do what it needs with it.

# Axios Asynchronous Flow

**`boardService.getBoards()`** code continues to run once the promise is returned.

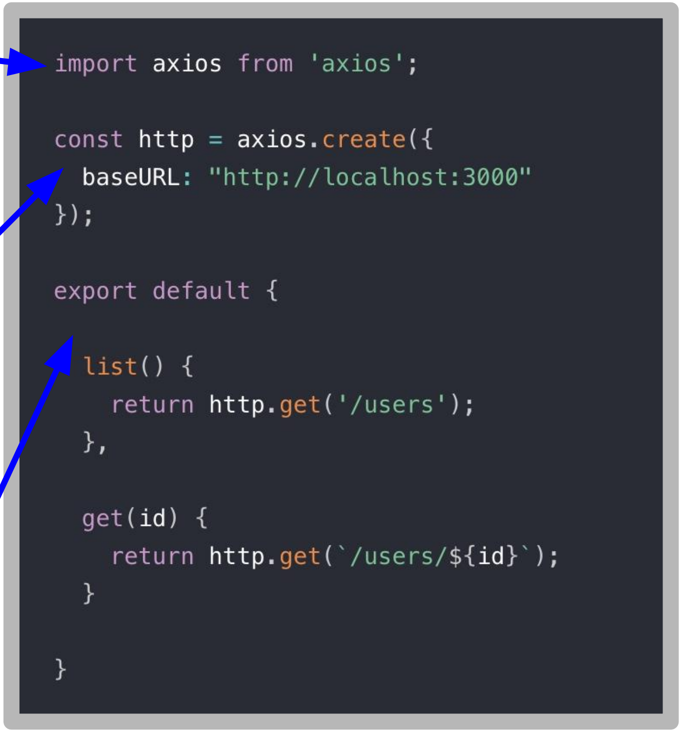
Once web call completes, the response is sent to the **`then`** handler in **`boardService.getBoards()`** and the promised data will be in the **`data`** property of the response if call was successful.



Imports the **axios** library.

Creates the a configured version of the Axios object and assigns it to an **http** variable. Here we set the base URL for calls made using the Axios object.

**export** the functions that make up the service object.

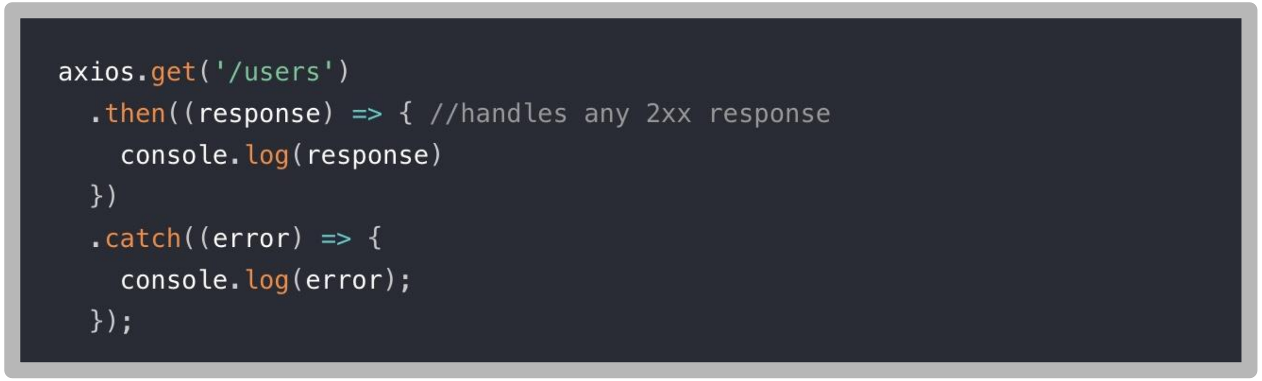


```
import axios from 'axios';

const http = axios.create({
  baseURL: "http://localhost:3000"
});

export default {

  list() {
    return http.get('/users');
  },

  get(id) {
    return http.get(`/users/${id}`);
  }

}
```

## Handling errors with Axios

You can chain a `.catch()` method after your `.then()` method. The `.catch()` method runs if

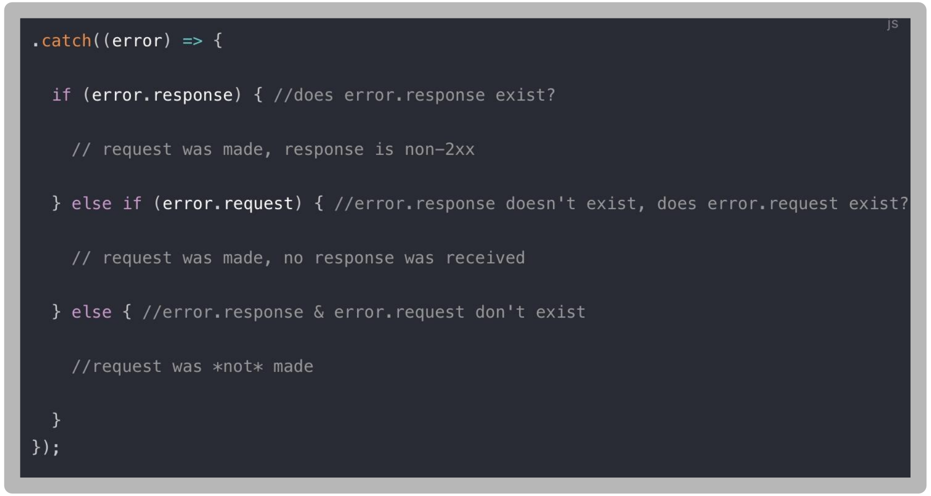
- The server responds with a non-**`2xx`** response code—remember that **`2xx`** codes are "success" messages.
- The server fails to respond due to an incorrect domain/port/protocol or network error.
- Something happened while setting up the request that triggered an error.



```
axios.get('/users')
  .then((response) => { //handles any 2xx response
    console.log(response)
  })
  .catch((error) => {
    console.log(error);
  });
```

## Handling errors with Axios

You can distinguish between these situations with an **`if-else`** block that tests for the **`.response`** and **`.request`** properties of the error object:



```js
.catch((error) => {

  if (error.response) { //does error.response exist?

    // request was made, response is non-2xx

  } else if (error.request) { //error.response doesn't exist, does error.request exist?

    // request was made, no response was received

  } else { //error.response & error.request don't exist

    //request was *not* made

  }
});
```

## Handling errors with Axios

With this process, you can handle all three situations that the .catch() method fires on. The .response object contains some more properties that can help you determine what happened:

- **`error.response.status`** is the response status code, like **`401`** or **`503`**. The description of the status code can be found in **`error.response.statusText`**.
- **`error.response.data`** contains information sent back from the server that might help you diagnose the error. This isn't a guarantee, but it's worth looking at if you do run into an issue.