ToDo List With Laravel 8 + Vue

- composer v 2.0.12
- npm v 6.14.12
- php v 8.0.1

1. Create a new Laravel project

```
composer create-project --prefer-dist laravel/laravel todolist
cd todolist
php artisan serve
```

2. Edit .env file and setup database settings

Make database settings. Database name, user and password. Change the name of the app (APP_NAME property). Don't forget to create database.

3. Create a model and migration file

The following command creates both the model and the migration file for that.

```
php artisan make:model Item -m
```

Model is created inside app/Models/Item.php and migration file is in database/migrations

4. Edit the migration file created and add some columns to our model database.

There is a public up function and a down function. We will add some columns to the up function.

```
$table->string('name');
$table->boolean('completed')->default(false);
$table->timestamp('completed_at')->nullable();
```

- name -> This is the name field for the items in the list.
- completed -> This is a boolean field for tracking if an item is completed or not. Its default value is false.
- completed_at -> This field is a timestamp for tracking the completion time of items. We set this as nullable since an item does not have a completion time when it is created.

5. Run migration

- Before running the migration make sure you have already created the database with the given name in .env file.

- Use the following command to run the migrations.

```
php artisan migrate
```

6. Create an item controller

Now we have to create a controller that will communicate with our model. There is a default controller in the controllers folder (app/http/Controllers).

- Now we create an ItemController using the following command.

```
php artisan make:controller ItemController --resource
```

- --resource option creates all the necessary resource creation functions inside the controller. This just saves some time for us.
- Before writing the controller functions let's create the necessary route definitions.

7. Create necessary routes

- We can create routes both in *web.php* and *api.php*

The routes created in the api.php are ***stateless***. They are also prefixed with "/api" URI. So, we access these routes by adding the "/api" prefix to the urls.

- Here we will be creting the routes in the *api.php* file. We first create the "/items" route.

```
Route::get('/items', [ItemController::class, 'index']);
```

- Import the ItemController

```
use App\Http\Controllers\ItemController
```

- We will create some routes with a common prefix. These routes will use post, put, and delete methods.

```
Route::prefix('item')->group( function() {
    Route::post('/store', [ItemController::class, 'store']);
    Route::put('/{id}', [ItemController::class, 'update']);
    Route::delete('/{id}', [ItemController::class, 'destroy']);
});
```

Some sample urls for these routes:

- http://localhost:8000/api/items
- http://localhost:8000/api/item/store
- http://localhost:8000/api/item/1

8. Write controller functions

- Now we edit the ItemController.php file to edit the necessary controller functions.
- First import the Model file

```
use App\Models\Item
```

- Write the index function that will simply return all the items in the todolist. We will order the list by the creation timestamp.

```
// index function
public function index() {
    return Item::orderBy('created_at', 'DESC')->get();
}
```

- Write the store function. When we use the put method to send data to our app it will hit the store function.

```
// store function
public function store(Request $req) {
    $newItem = new Item;
    $newItem->name = $req->item["name"];
    $newItem->save();
    return $newItem;
}
```

- Let's go ahead and test these routes and controller functions.
- We can use the Postman tool for this. This tool creates client side requests with necessary data.
  - Use post method
  - URL: http://localhost:8000/api/item/store
  - Set a header "Content-Type" in the Headers tab to "application/json"
  - Select the body tab and raw, then write the following content in json format:

  ```
  {
      "item":{
      "name":"Kitap oku"
      }
  }
  ```

- Now hit "send" and check the response. We should see the necessary record with necessary timestamp values created.

- Now, we can create a get request to see the inserted item.
    - Use get method
    - URL: http://localhost:8000/api/items
    - You should see the newly stored item returned with assigned creation timestamps

- Now let's create the update function to update exiting items.

- Edit the update function as follows:

```
 // update function

use Illuminate\Support\Carbon;
public function update(Request $request, $id) {
    $existingItem = Item::find($id);
    if( $existingItem ) {
        $existingItem->completed = $request->item['completed'];
        $existingItem->completed_at = $request->item['completed'] ? Carbon::now()
: null;
        $existingItem->save();
        return $existingItem;
    }
    return "Item not found.";
}
```

- Let's check if it works:
    - Create a new request in postman
    - The request method will be "put" this time
    - URL: http://localhost:8000/api/item/store/1
    - Set Content-type to "application/json"
    - In the body tab type raw data:

```
{
    "item": {
        "completed": true
    }
}
```

-
    - Now if we try an item with a non-existing id we get an "item not found" Error.

- Now let's create out delete function.

- We will edit the destroy function. We will be doing very similiar to the update function.

```
// destroy function
public function destroy($id) {
    $existingItem = Item::find($id);
    if( $existingItem ) {
        $existingItem->delete();
        return "Item successfully deleted.";
    }
    return "Item not found.";
}
```

- Let's try the destroy function in the Postman tool.
    - Create a new request. This time we will be using delete method. Set the method to "delete".
    - URL: http://localhost:8000/api/item/1
    - Hit send.
    - We can use the get method to list the existing items to confirm that item actually is deleted.
- Now that we know our backend api is working we can continue to the front-end.

9. Front-end with Vue

- First of all we have to install all the dependencies for our project. These dependencies are coded in package.json file.
- For this we use the npm (node package manager) tool.

```
npm install
```

- Then we install Vue.js to our project. Again we use the npm tool.

```
npm install vue@next vue-loader@next
```

- Now we will be creating our view files.
- In resources/js:
    - Create a new folder "vue". We will store all our vue files in this folder.
    - Create a file named app.vue in the vue folder.

```
<template>
    <div>
        Hello
    </div>
</template>

<script>
```

```
export default {

}
</script>
```

```
- Now we have a basic skeleton for our app.vue file.
```

- Now we will edit our app.js file.

```
//  - First import the vue package.

import Vue from 'vue'
//  - Now we import our vue.app file
import App from './vue/app.vue'

// Here we create the Vue front-end application object.
const app = new Vue({
    // create an element with an id of #app
    el: '#app',
    // pass in our app component that we just created
    components: { App }
});
```

- Now we have to tell our Wellcome view to use this Vue application file.
    - For that we edit the wellcome.blade.php file
    - Delete all the style and body contents.
    - Create a div with id app.

    ```
    <div id="app">
        <app>
        </app>
    </div>
    ```

    - Now that we have created our vue front-end app holder div, we have to import our app to fill in this holder.

    ```
    <script src="{{ mix('js/app.js') }}"></script>
    ```

    - Final *wellcome.blade.php* file be as follows:

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Laravel</title>

    <!-- Fonts -->
    <link href="https://fonts.googleapis.com/css2?
family=Nunito:wght@400;600;700&display=swap" rel="stylesheet">

</head>

<body>
    <div id="app">
        <app>
        </app>
    </div>
</body>
<script src="{{ mix('js/app.js') }}"></script>

</html>
```

  - We will be using Laravel mix for this. Laravel mix recompiles our app.js file in the resources folder whenever a change occurs and put it into public/js folder. This is coded in webpack.mix.js file. Laravel mix points it at the compiled version of our application.
  - We need to change the *webpack.mix.js* file as follows:

```
mix.js('resources/js/app.js', 'public/js')
    .vue()
    .postCss('resources/css/app.css', 'public/css', [
        //
    ]);
```

- We have to recompile our application whenever changes happen. To have it done automatically we do this:

```
npm run hot
```

- Now we can refresh our wellcome page on the browser. We will see "Hello"

- Now that we have our Vue instance ready we will add our vue componets.

- We will first create our Vue form components.

- We need a addItem form to be able to add new items to our Todo List.
    - Create addItemForm.vue file under resources/js/vue folder.

    ```
    <template>
    <div>
        Form
    </div>
    </template>

    <script>
    export default {

    }
    </script>
    ```

- We also need to create listView.vue to show all the items in our list. Create the *listView.vue* file under resources/js/vue folder.

    ```
    <template>
    <div>
        List view
    </div>
    </template>

    <script>
    export default {

    }
    </script>
    ```

- Let's create one more component listItem.vue to show one list item.

```
<template>
<div>
    List item
</div>
</template>

<script>
export default {

}
</script>
```

- Now that we have all our components let's edit the contents of these components.
- First we edit the app.vue file.

- Let's set how our ToDo list to look.
- Set main div class to "todoListContainer"

```
<div class="todoListContainer">
    <div class="heading">
        <h2 id="title"> ToDo List</h2>
    </div>
</div>
```

- We want to include our AddItemForm component under the title.
- To be able to use this component here we first include our form component. For that we add under script tag:

```
    <script>
    import addItemForm from "./addItemForm"
    export default {
        components: {
            addItemForm
        }

    }
    </script>
```

- Now we can use the form component. Now add the form component under title as follows:

```
<div class="todoListContainer">
    <div class="heading">
        <h2 id="title"> ToDo List</h2>
        <add-item-form />
    </div>
</div>
```

- Now we want to have all our list items after the heading div. We created our listView component for this purpose. So, we first load the listView component and use it here.

```
    <script>
    import addItemForm from "./addItemForm"
    import listView from "./listView"
    export default {
        components: {
            addItemForm,
            listView
        }
    }
    </script>
```

- Now we can put it under the heading div.

```
<div class="todoListContainer">
    <div class="heading">
        <h2 id="title"> ToDo List</h2>
        <add-item-form />
    </div>
    <list-view />
</div>
```

- Now let's add some styling to our app.vue file in order to have a better appearance.

```
<style scoped>
.todoListContainer {
    width: 350px;
    margin: auto;
}

.heading {
    background: #e6e6e6;
    padding: 10px;
}

#title {
    text-align: center;
}
</style>
```

- Let's have a look at our app and see how it looks.
- Now that we have our basic layout. We can create our form component.
- Open the addItemForm.vue file
    - We will give a class name to our form div.

```
    <template>
    <div class="addItem">
        <input type="text" />
    </div>
    </template>
```

- We have to put an add button to our form.

- We can use some icons for the buttons in our forms.
- For that we can import some icon pack. We can use font awesome icons.
    - Search web for "vue font awesome icons".
    - From the gitHub project page:
    - Install using npm

    ```
    npm i --save @fortawesome/fontawesome-svg-core
    npm i --save @fortawesome/free-solid-svg-icons
    npm i --save @fortawesome/vue-fontawesome@latest
    ```

    - To use the icons add following to the app.js file:

    ```
    import { library } from '@fortawesome/fontawesome-svg-core'
    import { faPlusSquare, faTrash } from '@fortawesome/free-solid-svg-
    icons'
    import { FontAwesomeIcon } from '@fortawesome/vue-fontawesome'

    library.add(faPlusSquare, faTrash)

    Vue.component('font-awesome-icon', FontAwesomeIcon)
    ```

- Let's see if we can use this icon. Open the *addItemForm.vue* file and edit as follows:

```
    <template>
    <div class="addItem">
        <input type="text" />
        <font-awesome-icon
        icon="plus-square"
        />
    </div>
    </template>
```

- Now let's add some styling to our form. Edit the *addItemForm.vue* file as follows:

```
<style scoped>
.addItem {
    display: flex;
    justify-content: center;
    align-items: center;
}

input {
    background: #f7f7f7;
    border: 0px;
```

```
    outline: none;
    padding: 5px;
    margin-right: 10px;
    width: 100%;
}
</style>
```

- Let's link the input to some data. Edit the script tag as follows inside *addItemForm.vue* file.

```
<script>
export default {
    data: function () {
        return {
            item: {
                name: ""
            }
        }
    }
}
</script>
```

- We link this data to the text input in our form. Edit the text input as follows:

```
        <input type="text" v-model="item.name"/>
```

- We can change the color of our button based on the text provided inside the text input. If there is no text inside the text field we want to gray out the button, so that it looks unclickable. And we want make it colored so that it looks clickable if there is text.
- For this to work we can use some class bindings. Edit the icon tag as follows:

```
    <font-awesome-icon
    icon="plus-square"
    :class="[ item.name ? 'active' : 'inactive', 'plus' ]"
    />
```

- We still have to create styling for these classes:

```
<style>
    .plus {
        font-size: 20px;
    }

    .active {
        color: #00CE25;
    }
    .inactive {
        color: #999999;
    }
</style>
```

- Now let's add the action to the button:

```
    <font-awesome-icon
    icon="plus-square"
    @click="addItem()"
    :class="[ item.name ? 'active' : 'inactive', 'plus' ]"
    />
```

- We have to add that addItem function. Whenever we click on the button this function will be executed.

```
<script>
export default {
    data: function () {
        return {
            item: {
                name: ""
            }
        }
    },
    methods: {
        addItem() {
            if( this.item.name == '' ) {
                return;
            }
            axios.post('api/item/store', {
                item: this.item
            })
            .then( response => {
                if( response.status == 201) {
                    this.item.neme = "";
                }
            })
            .catch( erro => {
                console.log(error);
            })
        }
    }
}
</script>
```

    - Let's try this. Enable developer tools in the browser and check for the network activity while submitting form. You can do this by opening the developer tools (F12 for Chrome browser). Then switch to "Network" tab and when you click the add button you will see network activity. If you select the "store" action from list you can see the contents of the data sent over.

- Now Let's create our list view to be able display our list items.
    - Open listView.vue
    - We want to receive our list of items as a prop. Here items will come from app.vue file. Edit the *listView.vue* file as follows:

```
<script>
export default {
    props: [ 'items' ]
}
```

    - Import the listItem and loop through the list items inside list view div.

```
<script>
import ListItem from "./listItem.vue"

export default {
    props: [ 'items' ],
    components: {
        listItem
    }
}
```

    - We are anticipating a list of items. So, let's create a vue for loop:

```
    <template>
    <div>
        <div v-for="(item, index) in items" :key="index">
            <list-item
                :item="item"
                class="item"
            />
        </div>
    </div>
    </template>
```

    - Let's give our items some style. Under the style tag:

```
<style scoped>
.item {
    background: #e6e6e6;
    padding: 5px;
    margin-top: 5px;
}
</style>
```

- We need to get all the items from the database and pass them to the prop: items. Open app.vue. Add some data after components.
- We also need to create method named getList() that will load the list of items from the database and put the items inside the items array.

```
    <script>
    import addItemForm from "./addItemForm"
    import listView from "./listView"
    export default {
        components: {
            addItemForm,
            listView
        },
        data: function () {
            return {
                items: []
            }
        },
        methods: {
            getList() {
                axios.get('/api/items')
                .then( response => {
                    this.items = response.data
                })
                .catch( error => {
                    console.log( error );
                })
            }
        },
        created() {
            this.getList();
        }

    }
    </script>
```

- We want to attach the items to a prop in our list of view

```
<div class="todoListContainer">
    <div class="heading">
        <h2 id="title"> ToDo List</h2>
        <add-item-form />
    </div>
    <list-view :items="items"/>
</div>
```

  - We are basically saying: Pass a property called "items" with the items that we are storing in the items into our list view. And in the listView we are saying that we are expecting a prop called items.

- Now lets create our list items. Open *listItem.vue* file.
  - First receive the item thorough props:

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

  - Include a class name to the item
  - Create a checkbox to check when an item is completed or uncheck if it is not completed.
  - When the state of the checkbox changes the updateCheck() function will be called.
  - We want to attach this checkbox to item.completed property. To do that we attach it by setting a v-model and setting it to "item.completed".
  - To display the item name with a look depending on its completed status we give it a class binding. If item is completed put a line through it.
  - Edit the *listItem.vue* file as follows:

```
<template>
<div class="item">
    <input
        type="checkbox"
        @change="updateCheck()"
        v-model="item.completed"
    />
    <span :class="[ item.completed ? 'completed' : '', 'itemText' ]">{{
item.name }}</span>
</div>
</template>

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

  - Create the styling

```
<style scoped>
.completed {
    text-decoration: line-through;
    color: #999999;
}
.itemText {
    width: 100%;
    margin-left: 20px;
}
.item {
    display: flex;
    justify-content: center;
    align-items: center;
}
</style>
```

- Create a button to delete the items.
- Make the button to look like a trashcan.
- When we click the button a function named *removeItem()* will be run. We will implement this function later.
- Edit the file as follows:

```
<template>
<div class="item">
    <input
        type="checkbox"
        @change="updateCheck()"
        v-model="item.completed"
    />
    <span :class="[ item.completed ? 'completed' : '', 'itemText' ]">{{
item.name }}</span>
    <button @click="removeItem()" class="trashcan">
        <font-awesome-icon icon="trash" />
    </button>
</div>
</template>

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

- Let's give the trashcan icon some style:

```
<style scoped>
.trashcan {
    background: #e6e6e6;
    border: none;
    color: #FF0000;
    outline: none;
}
</style>
```

- Let's create the updateCheck() function. In the listItem.vue file add following:

```
<script>
export default {
    props: [ 'item' ],
    methods: {
        updateCheck() {
            axios.put('api/item/' + this.item.id, {
                item: this.item
            })
            .then( response => {
                if( response.status == 200 ) {
                    this.$emit('itemchanged');
                }
            })
            .catch( error => {
                console.log( error );
            })
        }
    }
}
</script>
```

- Now let's create the "itemchanged" event. We go to listView.vue and add an event listener.

```
    <template>
    <div>
        <div v-for="(item, index) in items" :key="index">
            <list-item
                :item="item"
                class="item"
                v-on:itemchanged="$emit('reloadlist')"
            />
        </div>
    </div>
    </template>
```

- Now we have to create another event listener on app.vue that will listen for the reloadlist event.

```
<div class="todoListContainer">
    <div class="heading">
```

```
        <h2 id="title"> ToDo List</h2>
        <add-item-form />
    </div>
    <list-view
        :items="items"
        v-on:reloadlist="getList()"
    />
</div>
```

- Now let's create the removeItem() function in the listItem.vue

```
<script>
export default {
    props: ["item"],
    methods: {
        removeItem() {
        axios
            .delete("api/item/" + this.item.id)
            .then((response) => {
            if (response.status == 200) {
                this.$emit("itemchanged");
            }
            })
            .catch((error) => {
            console.log(error);
            });
        },
        updateCheck() {
        axios
            .put("api/item/" + this.item.id, {
            item: this.item,
            })
            .then((response) => {
            if (response.status == 200) {
                this.$emit("itemchanged");
            }
            })
            .catch((error) => {
            console.log(error);
            });
        },
    },
};
</script>
```

- To refresh the item list when we add a new item to the list edit the *addItemForm.vue* file as follows.

```
<script>
export default {
data: function () {
    return {
    item: {
        name: "",
    },
    };
},
methods: {
    addItem() {
    if (this.item.name == "") {
        return;
    }
    axios
        .post("api/item/store", {
        item: this.item,
        })
        .then((response) => {
        if (response.status == 201) {
            this.item.neme = "";
            this.$emit("reloadlist");
        }
        })
        .catch((erro) => {
        console.log(error);
        });
    },
},
};
</script>
```

- Finally edit the add-item-form component in the *app.vue* file so that when it receives a reloadlist event it calls the getList() function.

```
      <add-item-form v-on:reloadlist="getList()" />
```