

WEB701 Evaluate: Comparing web frameworks for implementing a website

Mark Christison

Nelson Marlborough Institute of Technology

Table of Contents

Table of Contents	2
Table of Figures	3
Web Frameworks	5
Comparison of Web Frameworks	7
Svelte Stack.....	7
Login.....	9
Account Administration.....	13
Register Products	17
Acquire Tokens.....	19
Vue Stack	21
Login.....	21
Administer Account.....	24
Products	26
Recommendation of web framework to use for the website.....	29
Blog Review of Assignment	31
References.....	32

Table of Figures

Figure 1 Svelte syntax.....	8
Figure 2 Sign in form.....	9
Figure 3 Svelte file for the login form	10
Figure 4 Login Routes	11
Figure 5 Login Controller login method.....	12
Figure 6 Account Registration Form	13
Figure 7 Account Registration Svelte file.....	13
Figure 8 Store method in the Account Controller.....	15
Figure 9 Create User action of Laravel Fortify.....	16
Figure 10 Svelte File for Adding a new product.....	17
Figure 11 Add new product form.....	17
Figure 12 Product routes.....	18
Figure 13 Store method of the Product Controller.....	18
Figure 14 Svelte file for getting tokens.....	19
Figure 15 Get Token form	19
Figure 16 Token routes	20
Figure 17 Store method of the token controller.....	20
Figure 18 Vue File for Login Page 1 of 2.....	22
Figure 19 Vue File for Log in page 2 of 2	23
Figure 20 Account Registration Page	24
Figure 21 Profile Page	24
Figure 22 Jetstream features config.....	25

Figure 23 Product Page.....	26
Figure 24 Vue File for Product page.....	27
Figure 25 Vue File for Product Card 1 of 2	27
Figure 26 Vue File for Product Card 2 of 2	28
Figure 27 Google Trends of React, Vue and Svelte (Google Trends, n.d.).....	30

Web Frameworks

The purpose of web frameworks is to solve most of the trivial tasks that can be repeated across different web applications, and instead allow developers to solve unique and interesting problems.

A common pattern that web frameworks follow is the MVC or Model-View-Controller design pattern. This separates the site into logical parts, each with its concerns. The views handle the logic for how things display on the screen and provide user intractable elements. The controller takes in those interactions and applies some business logic to the events. The model stores state and the logic of the controller is applied to. Hence, the flow control normally proceeds from the view to the controller to the model, back to the controller, and then finally to the view. (*Simple Example of MVC (Model View Controller) Design Pattern for Abstraction*, 2008)

Another feature of web frameworks is that they allow developers to interact with the request and response objects. Most frameworks provide helper methods that create objects for the request and response object such that they can interact within a controller.

By creating an object of the request, the request can have additional parameters including the type of request (get, put the update, delete, patch, and post), parameters of the request (for example search terms of a get request), and any authentication or session tokens that the user may have already received from the server to provide authentication or authorization for protected resources. (*Server-Side Web Frameworks*, 2019)

Web frameworks also often utilize an Object Relational Mapper (ORM) that provides a connection between database objects and objects that can be used by the framework. ORM's encapsulate the queries that are made to the database in a more fluent style with the language that

the framework is written in. Often, helper methods are written to perform repeated tasks, such as reading or updating fields in the database. This allows the developer to focus on writing code that performs well and encapsulates the logic of a function, rather than perfect SQL to do the same thing.

Another feature that web frameworks provide is Application Programming Interfaces (APIs) which act as an abstraction layer over more complex tasks that a framework can perform. APIs provide a clear and well defined way to interact with a framework that handles requests and responds with either the data requested or an error. A single page of a website may call many different APIs to get different data to display to the user, or might show data differently depending on the data that a page receives from an API depending on the logic of the page.

For security, web frameworks also offer Authentication and Authorization to protect or secure parts of the website or data. Authentication is a process that is used to verify that the user is who they say they are and authorization is a process of verifying what a user has access to. Authentication is normally achieved through the use of a user name and password. After a user passes authentication, they normally receive some form of access token that is stored in memory and passed in the header of each external call to the framework. To access protected assets normally you would need authentication and authorization. (*Simple Example of MVC (Model View Controller) Design Pattern for Abstraction*, 2008)

Comparison of Web Frameworks

For this Assignment, I have chosen to work with 2 different but similar stacks. The first stack uses:

- Laravel for the back-end logic (Otwell, 2015)
- Inertia as the adapter between Vue and Laravel (*Inertia.js - the Modern Monolith*, n.d.)
- Laravel Jetstream for authorization and authentication (*Introduction | Laravel Jetstream*, n.d.)
- Vue for the front-end views and logic (You, 2000)
- Tailwind CSS for styling the Vue components (*Tailwind CSS - Rapidly Build Modern Websites without Ever Leaving Your HTML.*, n.d.)

While the other stack uses:

- Laravel for the back-end logic (Otwell, 2015)
- Inertia as the adapter between Svelte and Laravel (*Inertia.js - the Modern Monolith*, n.d.)
- Svelte for the front-end views and logic (*Svelte • Cybernetically Enhanced Web Apps*, n.d.)
- Laravel Fortify for authentication and authorization (*Laravel - the PHP Framework for Web Artisans*, n.d.)
- Tailwind CSS for styling the Svelte components (*Tailwind CSS - Rapidly Build Modern Websites without Ever Leaving Your HTML.*, n.d.)

Svelte Stack

Components written in .svelte files are separated into 3 sections, Script, Styles and Markup. Since I am using tailwind css for my styles, I do not need to use the style section as the css classes defined in tailwind are used in the markup section.

The Script section contains JavaScript that runs when the component is created. This contains all the logic for the component and defines all the imported or defined variables used in the component.

Component props can be defined by adding the keyword 'export' before a variable definition e.g. 'export let propName'.

In the markup section, regular HTML elements can be added e.g. <div>, <p>, <a>. Capitalized tags denote that the component is a svelte component and requires that it is imported in the script section.

```
<script>
  import NavBar from "../Components/NavBar.svelte";
</script>

<NavBar />
```

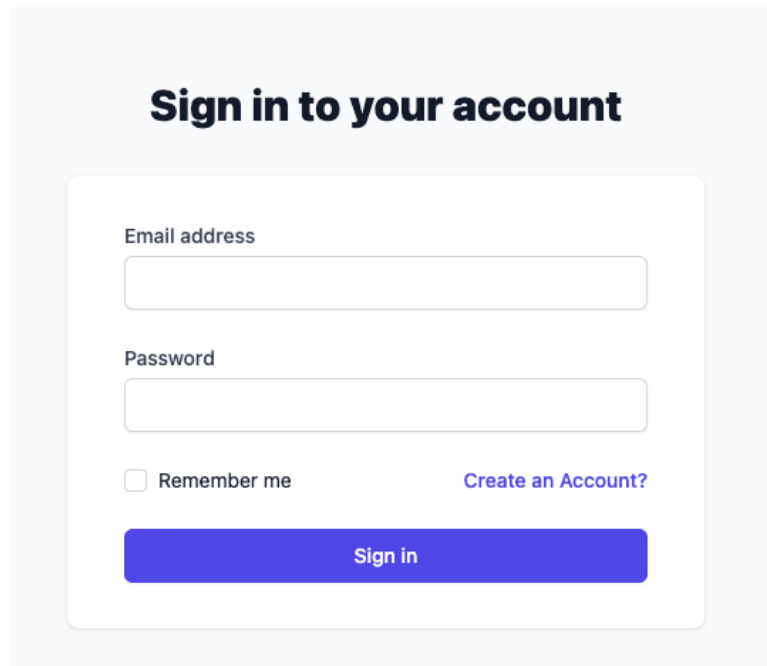
Figure 1 Svelte syntax

Svelte is a compiler so instead of just being vanilla JavaScript which is interpreted, svelte files are compiled down into vanilla JavaScript. This benefits the language when a page is loaded, due to the compiled files being smaller than other languages and load quicker as there is no virtual dom. As such, the speed of compilation which is relatively quick, reduces

The features that I have implemented in this stack include:

- Login
- Account Administration
- Register Products
- Acquire Tokens
- Accept tokens in transactions

Login



Sign in to your account

Email address

Password

☐ Remember me [Create an Account?](#)

Sign in

Figure 2 Sign in form

The login in form is relatively simple as seen in figure 2. It accepts a email address and password, and has a submit button to post the request to the server.

```

<script>
import NavBar from "../Components/NavBar.svelte";
import {useForm} from "@inertias/inertia-svelte";

export let errors = {}

let form = useForm({
  email: null,
  password: null,
});

function handleLoginSubmit() {
  $form.post('/login');
}
</script>

<NavBar/>
<div class="h-full bg-gray-50">
  <div class="min-h-full flex flex-col justify-center py-12 sm:px-6 lg:px-8">
    <div...>

    <div class="mt-8 sm:mx-auto sm:w-full sm:max-w-md">
      <div class="bg-white py-8 px-4 shadow sm:rounded-lg sm:px-10">
        <form class="space-y-6" on:submit|preventDefault={handleLoginSubmit}>
          <div...>

          <div...>

          <div...>

          <div>
            <button type="submit"
              class="w-full flex justify-center py-2 px-4 border border-transparent rounded-md
              shadow-sm text-sm font-medium text-white bg-indigo-600 hover:bg-indigo-700
              focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-indigo-500"
              disabled={$form.processing}>

```

Figure 3 Svelte file for the login form

The svelte component uses a svelte method useForm which allows for the binding of the values in the form object defined in the script section to the values on the form in the markdown section. The useForm method also allows the form to be submitted by calling the post method on the object and passing the endpoint of '/login'. Errors from the controller will be collected in the error prop and will be conditionally rendered to the page to provide feedback to the user on incorrect input to the form.

```
Route::controller(LoginController::class)
->middleware('web')->group(function () {
    Route::get('/login', 'index');
    Route::post('/login', 'login');
    Route::post('/logout', 'logout');
});
```

Figure 4 Login Routes

The request is handled by Laravel and endpoint is defined in the routes/web.php file. This is the block of logic that handles the request for the login. It passes the incoming request to the Login Controller class (which is imported at the top of the file). The routes defined in figure 3 are wrapped in a controller group, meaning that all the routes in the group will be handled by the actions defined in the controller passed, the login controller. The group also has web middleware which is defined in the Route Service Provider and bound in config/app.php. For the '/login' route if the method is 'GET' the 'index' method is called, and the same logic applies for the other routes.

```

/**
 * Login the user.
 *
 * @param Request $request
 * @return Application\RedirectResponse|Redirector|Response
 */
public function login(Request $request): Response|Redirector|Application\RedirectResponse
{
    $credentials = $request->validate([
        'email' => ['required', 'email'],
        'password' => ['required'],
    ]);

    if (Auth::attempt($credentials)) {
        $request->session()->regenerate();

        return redirect()->intended(RouteServiceProvider::HOME);
    }

    return back()->withErrors([
        'email' => 'These credentials do not match our records.',
    ])->onlyInput('email');
}

```

Figure 5 Login Controller login method

When a POST request is made to the /login route, the logic is handled by the 'login' method in the Login Controller. First the request is validated, if the email is in the request and is of an email format, and if the password is in the request then an attempt will be made to generate a session for the user, and redirect to the the 'Home' route defined in the Route Service Provider. If the request does not have a valid email, password or the username and password do not match a current user, then the error bag in the response will contain validation error messages.

Account Administration

Register for an Account

[Already have an account, Login?](#)

Name

Email address

Password

Account type

Figure 6 Account Registration Form

```
<script>
import NavBar from "../Components/NavBar.svelte";
import {useForm} from "@inertias/inertia-svelte";

let userRegistrationForm = useForm({
  name: null,
  email: null,
  password: null,
  accountType: null,
});

function handleSubmit() {
  $userRegistrationForm.post("/register");
}

</script>

<NavBar/>

<div class="h-full bg-gray-50">
  <div class="min-h-full flex flex-col justify-center py-4 sm:px-6 lg:px-8">
    <div class="mt-8 sm:mx-auto sm:w-full sm:max-w-md">
      <h2 class="my-6 text-center text-3xl font-extrabold text-gray-900">Register for an Account</h2>
      <div class="bg-white py-8 px-4 shadow sm:rounded-lg sm:px-10">
        <div class="flex justify-end text-sm">
          <a href="/login" class="font-medium justify-end text-indigo-600 hover:text-indigo-500">Already have an account, Login? </a>
        </div>
        <form class="space-y-6" on:submit|preventDefault={handleSubmit}>
          <div>
            <label for="name" class="block text-sm font-medium text-gray-700">Name </label>
            <div class="mt-1">
```

Figure 7 Account Registration Svelte file

The account registration form uses the same useForm method to create an object that is bound to the inputs in the form. The submit action of the form is prevented and the form is submitted to the '/register' route.

```
Route::controller(AccountController::class)
->middleware('web')->group(function () {
    Route::get('/register', 'index');
    Route::post('/register', 'store');
    Route::get('/account/', 'show');
    Route::post('/account/{user}/edit', 'update');
});
```

The account routes defined in web.php define the methods in the Account Controller for each of the routes and actions on the routes. To get the form, the user calls a GET request on the '/register' route. Once the form is completed and gets submitted it is submitted via a POST request to the '/register' route which is passed to the Account Controller class calling the 'store' method.

```
/**
 * Register a user.
 *
 * @param Request $request
 * @return Redirector|Application\RedirectResponse
 * @throws Exception
 */
public function store(Request $request): Redirector|Application\RedirectResponse
{
    $newUser = new CreateNewUser();
    $user = $newUser->create($request->all());

    Auth::login($user);
    $request->session()->flash('success', 'You have successfully registered!');

    return redirect(RouteServiceProvider::HOME);
}
```

Figure 8 Store method in the Account Controller

When the store method is called the CreateNewUser action of Laravel fortify as seen below. Once the user is validated and created, the Auth interface is called passing the user object. Finally a redirect to the 'Home' route defined in the route service provider is passed.

```
class CreateUser implements CreatesNewUsers
{
    use PasswordValidationRules;

    /**
     * Validate and create a newly registered user.
     *
     * @param array $input
     * @return \App\Models\User
     */
    public function create(array $input)
    {
        Validator::make($input, [
            'name' => ['required', 'string', 'max:255'],
            'email' => [
                'required',
                'string',
                'email',
                'max:255',
                Rule::unique(User::class),
            ],
            'password' => $this->passwordRules(),
            'accountType' => 'required',
        ])->validate();

        return User::create([
            'name' => $input['name'],
            'email' => $input['email'],
            'password' => Hash::make($input['password']),
            'accountType' => $input['accountType'],
        ]);
    }
}
```

Figure 9 Create User action of Laravel Fortify

Register Products

```
<script>
import NavBar from "../Components/NavBar.svelte";
import {useForm} from "@inertiajs/inertia-svelte";

let form = useForm({
  name: "",
  description: "",
  price: "",
  imageSource: "",
});

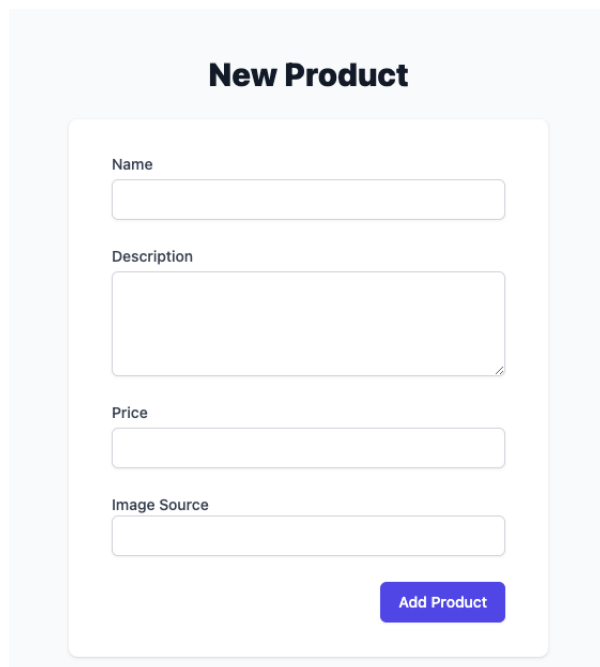
function handleSubmit() {
  $form.post("/products");
}

</script>

<NavBar/>

<div class="h-full bg-gray-50">
  <div class="min-h-full flex flex-col justify-center py-4 sm:px-6 lg:px-8">
    <div class="mt-8 sm:mx-auto sm:w-full sm:max-w-md">
      <h2 class="my-6 text-center text-3xl font-extrabold text-gray-900">New Product</h2>
      <div class="bg-white py-8 px-4 shadow sm:rounded-lg sm:px-10">
        <form class="space-y-6" on:submit|preventDefault={handleSubmit}>
          <div>
            <label for="name" class="block text-sm font-medium text-gray-700"> Name </label>
            <div class="mt-1">
```

Figure 10 Svelte File for Adding a new product



New Product

Name

Description

Price

Image Source

Add Product

Figure 11 Add new product form

The add new product svelte file works the same as the other forms. It uses the useForm method as the other forms previously described. The form is submitted to the '/products' route with the POST method.

```
Route::controller(ProductController::class)
->middleware('web')->group(function () {
    Route::get('/products', 'index');
    Route::get('/products/{product}', 'show');
    Route::get('/product/create', 'create');
    Route::post('/products', 'store');
});
```

Figure 12 Product routes

The POST method for storing a new product is passed to the 'store' method in the Product Controller.

```
/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Contracts\Foundation\Application|\Illuminate\Http\RedirectResponse
 */
public function store(Request $request)
{
    $product = $request->validate([
        'name' => 'required',
        'price' => 'required|numeric',
        'description' => 'required',
        'imageSource' => 'required',
    ]);

    Product::create($product);

    return redirect('/products');
}
```

Figure 13 Store method of the Product Controller

When the store method is called the request is validated, after it is validated the product is created and then the user is redirected to the ‘/products’ route.

Acquire Tokens

```
<script>
import NavBar from "../Components/NavBar.svelte";
import {page, useForm} from "@inertiajs/inertia-svelte";

let tokenForm = useForm({
  value: null,
  user_id: $page.props.user.user_id,
});

function handleSubmit() {
  $tokenForm.post("/tokens");
}
</script>

<NavBar/>

<div class="h-full bg-gray-50">
  <div class="min-h-full flex flex-col justify-center py-4 sm:px-6 lg:px-8">
    <div class="mt-8 sm:mx-auto sm:w-full sm:max-w-md">
      <h2 class="my-6 text-center text-3xl font-extrabold text-gray-900">Get a Token</h2>
      <div class="bg-white py-8 px-4 shadow sm:rounded-lg sm:px-10">
        <form class="space-y-6" on:submit|preventDefault={handleSubmit}>
          <div>
            <label for="value" class="block text-sm font-medium text-gray-700">Value </label>
            <div class="mt-1">
              <input id="value"
                name="value"
                type="number"
                required
                class="appearance-none block w-full px-3 py-2 border border-gray-300 rounded-md
                shadow-sm placeholder-gray-400 focus:outline-none focus:ring-indigo-500
                focus:border-indigo-500 sm:text-sm"
                bind:value={$tokenForm.value}>

```

Figure 14 Svelte file for getting tokens

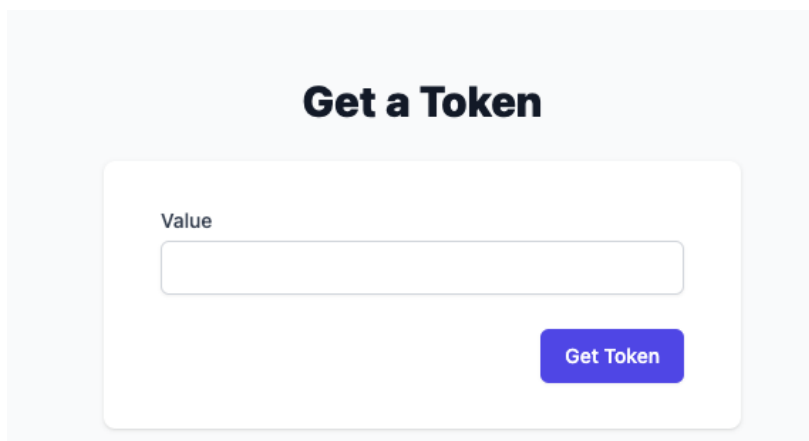


Figure 15 Get Token form

The token form is quite simple and just accepts a value as input. I would have added extra information to it but the brief was quite short on what the token needed to be.

```
Route::controller(TokenController::class)
->middleware('web')->group(function () {
    Route::get('/tokens', 'index');
    Route::post('/tokens', 'store');
});
```

Figure 16 Token routes

When the token form is submitted it is passed to the store method of the Token Controller.

```
/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\RedirectResponse|\Illuminate\Routing\Redirector
 */
public function store(Request $request)
{
    $request->validate([
        'value' => 'required|numeric|min:1'
    ]);

    $user_id = Auth::user()->id;

    Token::create([
        'user_id' => $user_id,
        'value' => $request->value
    ]);


    return redirect('/')->with('success', 'Token created successfully');
}
```

Figure 17 Store method of the token controller

Similar with the user and products, the request is validated, the user id is set to a value from the currently logged in user, the token is created and stored in the database and then the user is redirected to the home route.

Vue Stack

Login



Email

Password

☐ Remember me

[Register for an Account?](#)

```
<template>
  <Head title="Log in" />

  <jet-authentication-card>
    <template #Logo>
      <jet-authentication-card-logo />
    </template>

    <jet-validation-errors class="mb-4" />

    <div v-if="status" class="mb-4 font-medium text-sm text-green-600">
      {{ status }}
    </div>

    <form @submit.prevent="submit">
      <div>
        <jet-label for="email" value="Email" />
        <jet-input id="email" type="email" class="mt-1 block w-full" v-model="form.email" required autofocus />
      </div>

      <div class="mt-4">
        <jet-label for="password" value="Password" />
        <jet-input id="password" type="password" class="mt-1 block w-full" v-model="form.password" required autocomplete="current-password" />
      </div>

      <div class="block mt-4">
        <label class="flex items-center">
          <jet-checkbox name="remember" v-model:checked="form.remember" />
          <span class="ml-2 text-sm text-gray-600">Remember me</span>
        </label>
      </div>

      <div class="flex items-center justify-end mt-4">
        <Link href="/register" class="underline text-sm text-gray-600 hover:text-gray-900">
          Register for an Account?
        </Link>

        <jet-button class="ml-4" :class="{ 'opacity-25': form.processing }" :disabled="form.processing">
          Log in
        </jet-button>
      </div>
    </form>
  </jet-authentication-card>
</template>
```

Figure 18 Vue File for Login Page 1 of 2

```
</template>

<script>
import ...

export default defineComponent({ options: {
  components: {
    Head,
    JetAuthenticationCard,
    JetAuthenticationCardLogo,
    JetButton,
    JetInput,
    JetCheckbox,
    JetLabel,
    JetValidationErrors,
    Link,
  },

  props: {
    canResetPassword: Boolean,
    status: String
  },

  data() {
    return {
      form: this.$inertia.form({
        email: '',
        password: '',
        remember: false
      })
    }
  },

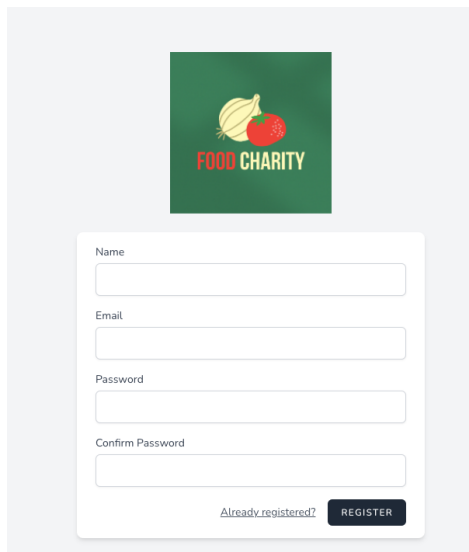
  methods: {
    submit() {
      this.form
        .transform(data => ({
          ... data,
          remember: this.form.remember ? 'on' : ''
        }))
        .post(this.route('login'), {
          onFinish: () => this.form.reset('password'),
        })
    }
  }
})
</script>
```

Figure 19 Vue File for Log in page 2 of 2

This is the log in page that is provided by Jetstream. In comparison to the Svelte file I find it much harder to understand what is going on in the page. Additionally, I tried to add the Nav bar to the top of the page so that I could navigate around the website, but I was unable to import the Nav Bar component. I'm not really sure how or why I couldn't do it when all other components are imported via an import command.

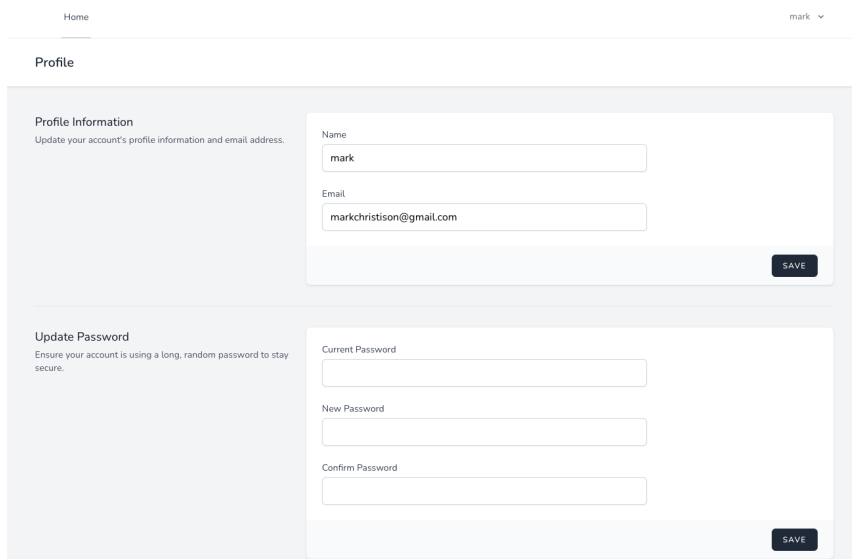
As with the Svelte components, this file uses the inertia form helper methods for handling the submission of the form. This posts the request to the /login route, passing the username and password as data.

Administer Account



The image shows a registration form for 'FOOD CHARITY'. At the top is a logo featuring a green square with a yellow corn cob and a red tomato, with the text 'FOOD CHARITY' below it. The form itself is a white box with the following fields: 'Name', 'Email', 'Password', and 'Confirm Password'. Below the 'Confirm Password' field is a link that says 'Already registered?' and a dark button labeled 'REGISTER'.

Figure 20 Account Registration Page



The image shows a 'Profile' page. At the top, there is a navigation bar with 'Home' and 'mark' (with a dropdown arrow). Below the navigation bar, the page is titled 'Profile'. There are two main sections: 'Profile Information' and 'Update Password'. The 'Profile Information' section has a sub-header 'Update your account's profile information and email address.' and contains two input fields: 'Name' (with the value 'mark') and 'Email' (with the value 'markchristison@gmail.com'). There is a 'SAVE' button at the bottom right of this section. The 'Update Password' section has a sub-header 'Ensure your account is using a long, random password to stay secure.' and contains three input fields: 'Current Password', 'New Password', and 'Confirm Password'. There is a 'SAVE' button at the bottom right of this section.

Figure 21 Profile Page

Jetstream out of the box offers forms prebuilt for account registration, and management. Features associated with the user account can be turned on and off such as if the user profile has pictures, 2 factor authentication, or if the user is a part of a ‘team’. These features can be turned on and off inside of the config/jetstream.php file.

```
'features' => [  
    // Features::termsAndPrivacyPolicy(),  
    // Features::profilePhotos(),  
    // Features::api(),  
    // Features::teams(['invitations' => true]),  
    Features::accountDeletion(),  
],
```

Figure 22 Jetstream features config

Products

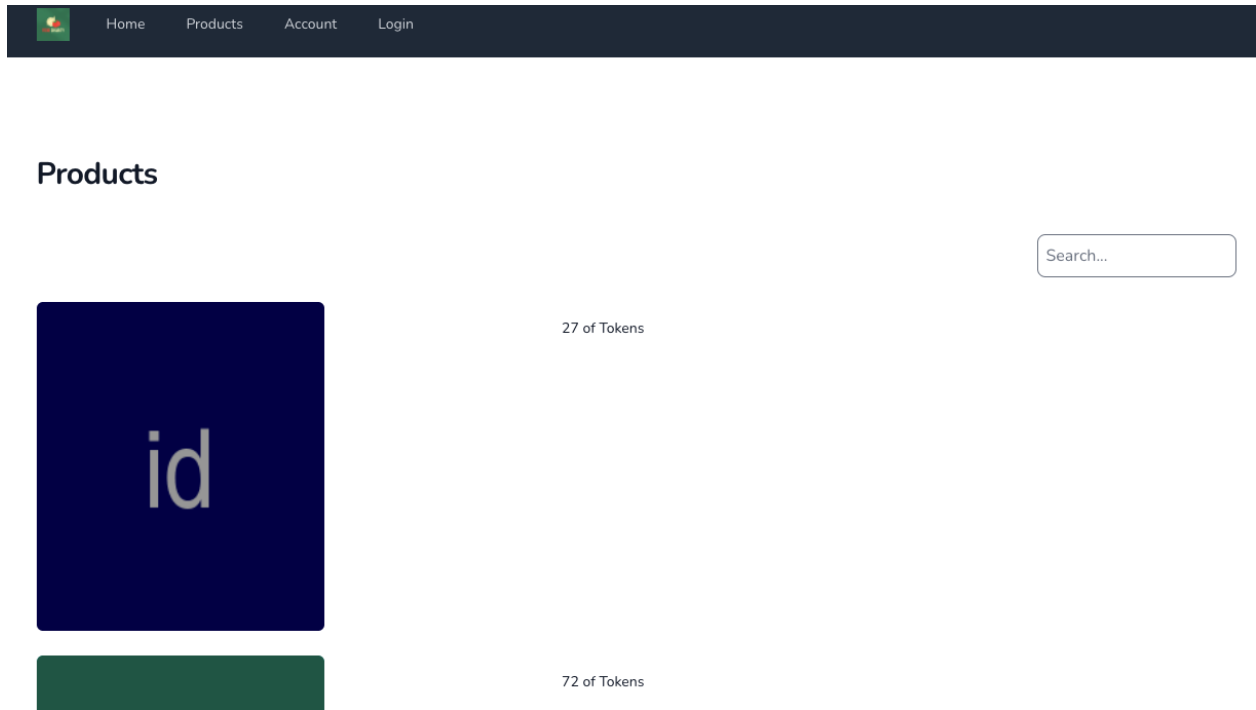


Figure 23 Product Page

```
<template>
  <NavBar></NavBar>

  <div class="max-w-2xl mx-auto py-16 px-4 sm:py-24 sm:px-6 lg:max-w-7xl lg:px-8">
    <div...>

    <div class="flex justify-end">

    <div class="mt-6 grid grid-cols-1 gap-y-10 gap-x-6 sm:grid-cols-2 lg:grid-cols-4 xl:gap-x-8"
      v-for="product in products">
      <ProductCard v-bind:product="product" />
    </div>
  </div>
</template>

<script setup>
import NavBar from './components/NavBar.vue';
import ProductCard from './components/ProductCard.vue';

let props = defineProps( props: {
  products: Object,
  filters: Object,
  auth: Object,
});

</script>
```

Figure 24 Vue File for Product page

```
<template>
  <div class="w-full min-h-80 bg-gray-200 aspect-w-1 aspect-h-1 rounded-md overflow-hidden group-hover:opacity-75 lg:h-80 lg:aspect-none">
    
  </div>
  <div class="mt-4 flex justify-between">
    <div>
      <h3 class="text-sm text-gray-700">
        <Link href="/products" method="get" :data="{product: product.id}" class="absolute inset-0">
          {{ product.name }}
        </Link>
      </h3>
    </div>
    <div>
      <p class="text-sm font-medium text-gray-900">{{ product.price }} of Tokens</p>
    </div>
  </div>
</template>
```

Figure 25 Vue File for Product Card 1 of 2

```
<script>
import { Link } from '@inertiajs/inertia-vue3';

export default {
  name: 'ProductCard',
  props: {
    product: {
      id: {
        type: String,
        required: true,
      },
      name: {
        type: String,
        required: true,
      },
      price: {
        type: Number,
        required: true,
      },
      imageSource: {
        type: String,
        required: true,
      },
    },
  },
  setup(props) {
    return {
      props,
      Link,
    };
  }
}
```

Figure 26 Vue File for Product Card 2 of 2

It was building this page with Vue, that I decided against the Vue stack. On this page, using the same markup, with the same styles as the Svelte file I could not replicate the functionality nor the styles. I don't even understand how the style would be different when the html elements have the same tailwind css classes. Moreover, the individual cards are not clickable and it isn't clear as to why. I tried several options of passing props to the child component on this page, yet nothing worked.

To me, the unnecessary complexity of such a simple and often used feature of a language shouldn't be this hard. I should be able to pass props to a component as I did with svelte by calling the spread operator (...) on the object that I want those props to come from. Instead, with Vue I have to vbind all the props, and it is not clear how or what I should be binding. I would like this to be simpler, but the documentation for Vue doesn't explain it clearly to me as a new developer working with Vue.

Recommendation of web framework to use for the website

For the continued development of the website, I would recommend and will be using Svelte.

The simplicity of Svelte is one of its major benefits. To define a prop, you create a variable and export it i.e. `export let prop`, whereas with react if a variable change would cause a component re-render, then the `setState` and/or `useState` would have to be used. This bulks out code with an extra import, a line for defining the state of the variable then actually using the variable.

Additionally, svelte props/variables have 2 way data binding, meaning that there is no need to define how the variable is used in the component markup, and what it is bound to.

The biggest benefit and one of the major ways in which Svelte is superior is that it does not use a virtual DOM. A virtual DOM is a set of rendered objects built by a framework such as Vue or React. These objects represent how the page should look in the real DOM. When props on these objects change, the framework re-renders the components and outputs them to the real DOM. These additional steps add computation complexity at the time when a user is expecting to see a change in the DOM as fast as possible. While the virtual DOM provides a simple way to write declarative, state-drive UI components, Rich Harris argues that it is a means to an end (Harris, 2018). Rich built Svelte without a virtual DOM to avoid the inherent problems of computational complexity and performance cost that it brings.

Svelte was the “most loved” framework in the stack overflow 2021 survey (Stack Overflow Developer Survey 2021, n.d.). However, the community using it is relatively small.

Looking at google trends, the search volume in comparison to that of React or Vue is rather small (granted the word React may be included in other search terms).

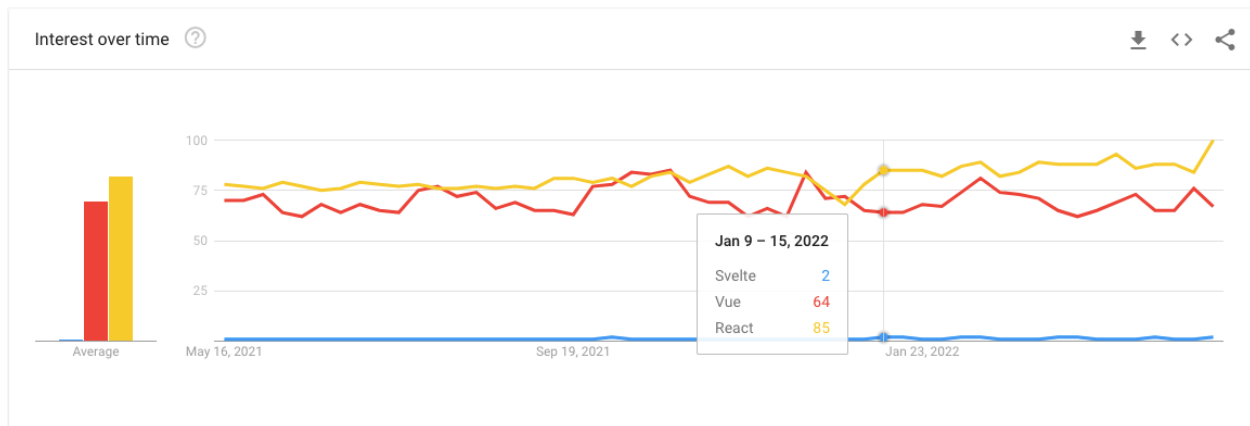


Figure 27 Google Trends of React, Vue and Svelte (Google Trends, n.d.)

It's also not clear to me why in Vue I have to both import and export a markup component that is being used inside the markup. Do other components then know about the component and which then takes precedent? It makes the component difficult to understand and adds extra code for no added benefit. Svelte is far more straight forward, import component to use it in the file. If a svelte component wants to use another svelte component it imports it. In Vue a component is both imported and exported, which seems unnecessarily complex.

Overall, I think the simplicity of Svelte has made it easier to learn, and write especially coming into it not really liking JavaScript frontend development. Writing Svelte was what I was hoping development was going to be like when I started working with it. I have found it pleasurable to write Svelte as it just makes sense at a glance what is going on. Even if Svelte is not going to gain mainstream adoption, I believe that it will change the development of future frameworks to reconsider reactivity in a different way. In fact, frameworks like Solid (*SolidJS*,

WEB701 Evaluate: Comparing web frameworks for implementing a website

n.d.) are already coming out that fundamentally change the way in which they interact with the virtual DOM and as a side effect have significant improvements in page load times and size

(*Interactive Results*, n.d.)

Blog Review of Assignment

<https://mckevmeister.github.io/web701/Review-of-Evaluate-Assignment>

References

- Auth0. (n.d.). *Authentication vs. Authorization*. Auth0 Docs. <https://auth0.com/docs/get-started/identity-fundamentals/authentication-and-authorization>
- Google Trends. (n.d.). Google Trends. Retrieved May 15, 2022, from <https://trends.google.com/trends/explore?q=Svelte>
- Harris, R. (2018, December 27). *Virtual DOM is pure overhead*. Svelte.dev. <https://svelte.dev/blog/virtual-dom-is-pure-overhead>
- Inertia.js - The Modern Monolith. (n.d.). Inertiajs.com. <https://inertiajs.com/>
- Interactive Results. (n.d.). Krausest.github.io. Retrieved May 15, 2022, from https://krausest.github.io/js-framework-benchmark/2022/table_chrome_101.0.4951.41.html
- Introduction | Laravel Jetstream. (n.d.). Jetstream.laravel.com. <https://jetstream.laravel.com/2.x/introduction.html>
- Laravel - The PHP Framework For Web Artisans. (n.d.). Laravel.com. <https://laravel.com/docs/9.x/fortify>
- Otwell, T. (2015). *Laravel - The PHP Framework For Web Artisans*. Laravel.com. <https://laravel.com/>
- Server-side web frameworks. (2019, May 7). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks
- Simple Example of MVC (Model View Controller) Design Pattern for Abstraction. (2008, April 8). CodeProject. <https://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design>
- SolidJS. (n.d.). Ww.solidjs.com. Retrieved May 15, 2022, from <https://www.solidjs.com/>

Stack Overflow Developer Survey 2021. (n.d.). Stack Overflow.

<https://insights.stackoverflow.com/survey/2021#section-most-loved-dreaded-and-wanted-web-frameworks>

Svelte • Cybernetically enhanced web apps. (n.d.). Svelte.dev. <https://svelte.dev/>

Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. (n.d.).

[Tailwindcss.com](https://tailwindcss.com/). <https://tailwindcss.com/>

You, E. (2000). *Vue.js*. [Vuejs.org](https://vuejs.org/). <https://vuejs.org/>