**Website New**

<!-- Website-new allows users to create new websites. It's currenly showing the default websites list and other default route paths. We have to make this dynamic the way we did website-list.

We will need the same variables, imports, components and ngOnInit function from the website-list component ts for the website-new component ts, so we can copy and paste them in their respective places.

Then, we go to the html, delete all <li> items, except for the first one, which we will be editing.

Change the "fas-fa-check" <a> tag to a <button> with class="sw-font-white float-right sw-button-to-a", type="submit", form="websiteForm" and [disabled]="f.invalid".

Then we update the list of websites by adding \*ngFor = "let website of websites" to <li>, which lets the component ts doc know what we're using this list for. It will be a list of websites.

Then we change:

<a routerLink="/user/:uid/website/:wid/page" class="text-primary sw-no-underline">Address Book App</a>

to

<a class="text-primary sw-no-underline" routerLink="/user/{{uid}}/website/{{website.id}}/page">{{website.name}}</a>

We took this from what we did in the website-list html.

Now, we look at the form on the right side of the page. In the <form> tag, we add id="websiteForm", to link it with the form="websiteForm" that we put into the check mark <button> that we just edited a little while ago. id="websiteForm" identifies the form to be submitted, and form="websiteForm tells the button which form it will be working for by using.

To the <form> tag, we also add (ngSubmit)="create()" and #f="ngForm"

(ngSubmit)="create()" will link the form on the html page to the create() function in the component ts doc. #f="ngForm" will link the form to the ViewChild function in the component ts doc, where it says @ViewChild('f') websiteForm: NgForm.

Now, we want to update the inputs with their ngModel.

required ngModel #name="ngModel"

and

ngModel #description="ngModel"

Notice that we didn't make the description required, because all we really need the user to enter is the name. The description will be optional.

Update the Cancel button from

<a routerLink="/user/:uid/website" class="btn btn-success float-right">Submit</a>

to

<button routerLink="/user/:uid/website" class="btn btn-success float-right" [disable]="f.invalid">Submit</button>

It disables it if the form isn't filled in.

Note, because this button is inside of the form, we don't have to tell it which form it's assigned to submit.

Now, we link the form in the html to the component ts doc.

We import forms: import { NgForm } from '@angular/forms'.

Then we add the ViewChild: import { ... ViewChild } from '@angular/core', which makes the @ViewChild possible to run.

Then, we add @ViewChild('f') websiteForm: NgForm;

Now, let's add an alert so that it will fire if the Name field is left empty.

<div class="alert alert-warning" \*ngIf="name.invalid & name.touched">Name cannot be empty.</div>

Question: What's the difference between <span> and <div> in this case? We used <span> initially, but it looked/behaved differently.

This means that this alert will be invisible until the field is touched AND left blank, at which point a yellow warning field will be triggered.

So, now we have to link the service to the New Website form to be able to actually create a new website (and add it to our website array) when we use it.

First, we set up a way to retrieve the data. To do this, we set up/connect the data variables in the component ts doc, some of which were referened in the html doc.

name: string;

description: string;

Hint/Note: The #name has to match the variable in the component ts doc. So it needs name="name" and #name="ngModel" in the html form and has to also be called name: string; in the component ts doc.

Now, we establish the create() function. In Website Service, there's a createWebsite function that we can use. This function takes the userId and website, and will create a website with the userId=developerId.

When the user hits submit, the information retrieved is the name and the description, represented as such:

this.name = this.websiteForm.value.name;

this.description = this.websiteForm.value.description;

We also create a const variable to represent the newWebsite that will be created.

const newWebsite: Website = {

\_id: "",

name: this.name,

developerId: "",

description: this.description,

};

Looking at the Websites array in Website Service, we need 4 pieces of data: \_id, name, developerId and description.

The \_id: and developerId can be empty (""), because a random number will be generated for it, according to the create() function via the lines:

website.\_id = Math.floor(Math.random() \* 10000).toString();

and

website.developerId = userId;

The name and description will come from the name entered into (this) websiteForm.

Then, we incorporate the createWebsite function:

this.websiteService.createWebsite(this.uid, newWebsite);

A website will be created using the uid (showing in the url) and the details entered into newWebsite.

After the user creates a website, they should be rerouted back to their website list page.

Remember, ActivatedRoute only allows us to parse and find the url, it doesn't allow us to navigate to another page. The Router is the item that allows us to navigate to another page. So, we have to add Router to the ActivatedRoute import.

import { ActivatedRoute, Router } from '@angular/router'

Then, we create the code that will actually reroute to the website list page:

this.router.navigate(['user', this.uid, 'website']);

We look at that app.rouing.ts doc and find the WebsiteListComponent file path, which is

'user/:userId/website'.

We add it into this.router.navigate([]) to become

this.router.navigate(['user', this.uid, 'website']);

We essentially separated the file path into items.

user = 'user'

:userId = this.uid

website = 'website'

This is how the page address it will show up in the url once the website is clicked on.

**Profile**

<!-- We want to be able to view and update the user's profile information once they log in or register.

We start by adding (ngSubmit)="update()" to <form>.

ngSubmit - links the function from the compotent ts that allows us to submit the form. In this case, it's update(), since that will be the primary function while the user visits their profile page.

(ngSubmit)="update()" will connect to the update() function defined in the component ts.

Now, remember the check mark? That is also an event that will submit/update the profile information when we click it, so we have to make that functional as well. We do this by changing it from <a> into a <button> and adding type="submit". Because we've added this type, we delete the href, because we are now telling it to submit information, rather than to re-route to somewhere.

Changing it from <a> to <button> changes how it looks to a box button rather than the free-floating check mark that it was before. To update it, we create styling for it to remove the grey styling of the button. We have created the class sw-button-to-a, which we update in style.css as

.sw-button-to-a{

background-color: transparent;

The button is now "clear" and therefore showing the navbar color of blue, but there's still a button border around it. To get rid of it, we add btn (a Bootstrap class) to the <button>.

<button class=" btn sw-font-white sw-button-to-a" type="submit" form="profileForm" [disabled]="f.invalid"><i class="fas fa-check"></i></button>

Notice that the code for this check mark is outside of the form that updates the profile information, so this button doesn't yet know what it's submitting once it's pushed. Therefore, we need to bind that button with the form that does submit the profile info. To do this, we then add form="profileForm" to <button> and to <form>.

Question: Why form="profileForm"? Where is this equation housed? Why not use ngSubmit again in the check mark button?

We now make it so that if the form is invalid, the submit button is disabled, by adding [disabled]="f.invalid".

Now, we add the ngModel to the form.

ngModel - binds all of the fields of the form together under ngForm

We add #f="ngForm" to link it to the functionality in the profile component ts.

ngForm - gives the form a name

In the <label> for each respective field, we add:

for="username"

for="email"

for="firstName"

for="lastName"

The for= attribute specifies which form element a label is bound to. (See w3schools)

In the <input> tags, we add:

required ngModel #username="ngModel"

ngModel #email="ngModel"

ngModel #firstName="ngModel"

ngModel #lastName="ngModel"

For this form, we're making it so that the email and names aren't required, so we're leaving the required attribute off.

Now, we go to the component ts doc to add our functions.

ngOnInit() is a function that will automatically run when the page is loaded. So, within this function, we will indicate what we want to happen automatically, once the page is loaded.

The first thing we want to display is the user data in their proper fields in the profile.

Remember that we have made it so that the user ID is captured and displayed once a user either logs in or registers. We can use this same user ID to retrieve and display profile information.

We start by importing a class called ActivatedRoute, which gives us the ability to retrieve our current url from @angular/router.

import {ActivatedRoute} from '@angular/router'

We then create an instance for ActivatedRoute in the constructor with a type of ActivatedRoute.

constructor(private activatedRoute: ActivatedRoute)

In the ngOnInit(), we create

var profile = this;

this.activatedRoute.params.subscribe(

function info(params){

profile.uid = params['uid']

}

)

We created var profile = this, because if we used "this" within the info function, it would be so deeply nested, that it would not refer to the same "this" as before, and we would get an error.

params allows us to collect all of the parameters held inside the activatedRoute.

subscribe is a function tha will wait while the page is loading, and runs another function that will be ready by the time the page finishes loading.

profile.uid = params['uid'] represents the user ID in our url, so essentially, this function is first requiring that we identify and acknowledge the user ID in the url, which is found in the parameters (params) of the activatedRoute.

Now, we take that user ID that we found and grab the user data from userService, according to the user ID (uid) that we just found.

That's essentially what this means:

profile.user = profile.userService.findUserById(profile.uid);

But first, we have to add user services by importing UserService to the component ts doc.

import {UserService} from '../../../services/user.service.client'

import {User} from '../../../models/user.model.client'

We also add userService to the constructor:

constructor(private activatedRoute: ActivatedRoute, private userService: UserService)

We then add the part that grabs the user data from userServices.

this.activatedRoute.params.subscribe(

function info(params){

profile.uid = params['uid']

profile.user = profile.userService.findUserById(profile.uid);

}

)

Now, we define the variables for the profile fields that will store the information entered.

uid: string;

user: User;

username: string;

email: string;

firstName: string;

lastName: string;

We will then add them to the info function.

ngOnInit() {

var profile = this;

this.activatedRoute.params.subscribe(

function info(params){

profile.uid = params['uid']

profile.user = profile.userService.findUserById(profile.uid);

profile.username = profile.user.username;

profile.email = profile.user.email;

profile.firstName = profile.user.firstName;

profile.lastName = profile.user.lastName;

});

These items represent the values of the profile form fields.

-----------------------------

Some alternative workarounds for this section:

ngOnInit() {

this.activatedRoute.params.subscribe(

function info(params){

this.uid = params['uid']

this.user = this.userService.findUserById(this.uid);

this.username = this.user.username;

this.email = this.user.email;

this.firstName = this.user.firstName;

this.lastName = this.user.lastName;

}.bind(this));

}

This binds "this" with the same "this" referred to in the class.

Or...

ngOnInit() {

this.activatedRoute.params.subscribe(

params => {

this.uid = params['uid']

this.user = this.userService.findUserById(this.uid);

this.username = this.user.username;

this.email = this.user.email;

this.firstName = this.user.firstName;

this.lastName = this.user.lastName;

});

This is an arrow function. It won't replace the this value.

This function takes the parameters and

-----------------------------

We now officially bind these items we created in the component ts with their values in the fields represented in the html. So, we then revisit the attributes that we created earlier:

required ngModel #username="ngModel"

ngModel #email="ngModel"

ngModel #firstName="ngModel"

ngModel #lastName="ngModel"

We update the ngModel to give them values:

required ngModel="{{user.username}}" #username="ngModel"

ngModel="{{user.email}}" #email="ngModel"

ngModel="{{user.firstName}}" #firstName="ngModel"

ngModel="{{user.lastName}}" #lastName="ngModel"

We use user. so that the values will be pulled from userServices.

We now create the function for updating user information.

update(){

this.username = this.profileForm.value.username;

this.email = this.profileForm.value.email;

this.firstName = this.profileForm.value.firstName;

this.lastName = this.profileForm.value.lastName;

}

We must now make sure that we check to see if the username already exists in the system.

-----------------------------------

Note: Get help with 4.2, starting @11:12. It doesn't make sense to me. What does aUser, oldUsername and this.username mean?!

// If we can find a user with this username, then it means that this user already exists, and cannot be used to register or update the account; or if the username has not been changed.

aUser will be the variable for the new user in this equation.

const aUser: User = this.userService.findUserByUsername(this.username);

This line looks into userService, calls upon the function findUserByUsername, according to the username entered into the form (this.username).

We then define what happens with this username (aUser) if it is or is not found to exist in the user array, using the following if statement:

if(aUser && this.oldUsername != this.username){

this.usernameTaken = true;

this.submitSuccess = false;

oldUsername: string; and username: string; are the same thing, but we have to make a distinction when they appear together at times when we're checking and updating information. oldUsername will appear in the if statement that checks for system matches if and only if information has been changed in the field.

aUser represents the user information as it exists once a profile page is loaded, and is sourced from userService.

-----------------------------------

Then, we create the alerts to express that the username is already taken or that the changes are submitted successfully.

In the component ts doc, we create two variables:

usernameTaken: boolean;

submitSuccess: boolean;

In the html, we add two <div> above the <form>:

<div \*ngIf="usernameTaken" class="alert alert-danger">Username taken, please try another one.</div>

<div \*ngIf="submitSuccess" class="alert alert-success">Updated successfully!</div>

When \*ngIf is true for the situation represented in the component ts doc, then these alerts will run. \*ngIf is the element that makes this happen. It's hidden until the respective boolean is true. So, if this.usernameTaken = true then the usernameTaken \*ngIf alert will appear. Same for \*ngIf="submitSuccess"

Once the username, etc. are verified, we have to officially add the updated data to our array. This is the info that will be captured:

else {

const updatedUser: User = {

\_id: this.user.\_id,

username: this.username,

password: this.user.password,

firstName: this.firstName,

lastName: this.lastName,

email: this.email

Note: At some point, add a change password feature here.

We then call the updateUser() function and instruct it to update the user according to the current uid and the information provided in updatedUser: User.

this.userService.updateUser(this.uid, updatedUser);

this.usernameTaken = false;

this.submitSuccess = true;

Question: Why do we need usernameTake and submitSuccess here again?

-->