comp-sci-final-project

Note: If I don't explain how a certain segment of the code handles errors, it is because an error cannot occur there, NOT because I forgot to include it. Also the reason I used a combination of snake case and camel case is because rust typically uses snake case, while a lot of the Components and structs in the `yew` library I used, use camel case.

This project is seperated into 4 main sections.

1. Backend
2. Components
3. Pages
4. stores

Backend

This section organizes all of the data such as routing, menu items, and enums in a way that is easily accessable by any components that may need them.

Cart and category

The cart struct and category enum are kept very simple because they need to be used many times in the code and if they were large, they would be more difficult to debug and use.

```
// cart.rs
...
pub struct CartItem {
    pub person: String,
    pub item: MenuItem,
}

// category.rs
...
pub enum Category {
    Appetizer,
    Entree,
    Drink,
    Dessert,
}
```

Menu

The `menuItem` struct is used to neatly organize every item on the menu into its own object.

```
// menu.rs
...
pub struct MenuItem {
    pub id: u16,
    pub category: Category,
    pub name: String,
    pub price: f32,
    pub image: String,
    pub description: String,
```

```
}
```

The make menu fucntion can be called to return a vector/list of all the items on the menu.

```
// menu.rs
...
fn make_menu() -> Vec<MenuItem> {
    vec![
        MenuItem::new(Category::Appetizer, 1, "Guacamole & Chips".to_string(),
"/assets/img/Guacamole.jpg".to_string(), 5.99, "Homemade with fresh avacados and chips".to_string()),
        MenuItem::new(Category::Appetizer, 11, "Broccoli Cheddar Soup".to_string(), "/assets/img/Broccoli-
Cheddar.jpg".to_string(), 5.99, "Caramelized onion and garlic-based, thickened with a butter and flour mixture then
simmered with veggie stock, broccoli, and carrots".to_string()),

        ...
    ]
}
```

Page Routing

This file neatly maps the possible URLs to an enum so it can later be used to determine what page should be diplayed to the user.

```
    // route.rs
    ...
    pub enum Route {
    #[at ("/")]
    Root,
    #[at("/description/:item_id")]
    Description{item_id: u16},
    #[at("/:category")]
    Menu{category: String},
    #[at("/person-select/:item_id")]
    Select{item_id: u16},
    #[at("/bill")]
    Bill,
    #[not_found]
    #[at("/404")]
    NotFound,
}
```

Components

These components are dynamic reusable HTML elements that are displayed to the user's browser. I will not be explaining all of the components in this section because some of them are pretty stright forward(such as a `clear_cart_button`).

Bills

The bills are seperated into 2 different components. One is `CombinedBill` and the other is `SeperateBills`

The caclulation for the total and subtoal is simmilar for both bills, only the seperate bill makes sure it only adds the correct persons order to the total. The calculation cycles through the given item list and adds its price to the subtotal. Later when it is displayed, a 13% sales tax is added on to the total tag.

```
// combined_bills.rs/seperate_bills.rs
...
let listed_items: Html = items.iter().map(|item| {
        total_f += item.item.price as f64;

        html!{
            <p class={"billedItem"}>{format!("{} □ ${}", item.item.name, item.item.price)}</p>
        }
    }).collect();

...

let subtotal = html! {
        <p>{format!("Subtotal: ${:.2}", total_f)}</p>
    };
let total = html! {
    <p>{format!("Total: ${:.2}", total_f + (total_f * 0.13))}</p>
};
```

Item List

This component loops through all the items in a given category(from `menu.rs`) and turns them into html elements that can easily be styled.

```
// item_list.rs
...
#[function_component(ItemList)]
pub fn itemlist(Props { items }: &Props) -> Html {

    items.iter().map(|item: &MenuItem| {
        html!{
        <div class={"item"}>
            <p>{format!("{} □ ${}", item.name, item.price)}</p>
            <a href={format!("/description/{}", item.id)}><img class={"itemImage"} src={item.image.clone()}/></a>
            <br/>
            <OrderButton item={item.clone()}/>
        </div>
        }
    }).collect()
}
```

Navbar

This is the navigation bar at the top of the screen that allows you to pick between the different menu categories and see the bill screen. The `Bill` button is simply a link to the bill page, but the category selector is a tiny bit more complicated. The code checks for changes in the `select` tag's state and uses a `navigator` to reroute the page to display the chosen category.

```
// navbar.rs
...
let navigator = use_navigator().unwrap();

let on_change = Callback::from(move |event: Event| {
    let route = event.target()
        .unwrap()
        .unchecked_into::<HtmlInputElement>()
```

```
        .value();

    navigator.push(&Route::Menu { category: route });
});

...
```

Pages

These are the seperate pages that are displayed depending on the current URL

App

The app page is technically the only page the site has and is constantly displayed at all times. It uses a switch funciton to change the html body depending on the URL.

```
// app.rs
...
fn switch(routes: Route) -> Html{
    match routes {
        Route::Root => html!{<Redirect<Route> to={Route::Menu{category: Category::Appetizer.to_string()}}/>},
        Route::Menu{category} => html! {<MenuPage category={category}/>},
        Route::Description{item_id} => html! {<DetailsPage item_id={item_id}/>},
        Route::Select{item_id} => html! {<PersonSelectPage item_id={item_id}/>},
        Route::Bill => html! {<BillPage/>},
        Route::NotFound => html!{<NotFound/>},
    }
}
...
```

Bill Page

This page displays either the `CombinedBill` component, or the `SeperateBills` component depending on the option selected. When one of the buttons is pressed, the `BillState` store is updated to save the state of the page.

This is the code that handles changing the state when the buttons are clicked:

```
// bill_page.rs
...
let c_dispatch = dispatch.clone();
let combined_click = Callback::from(move |_| {
    c_dispatch.reduce_mut(|bill| bill.bill_toggle = 0);
});

let seperate_click = Callback::from(move |_| {
    dispatch.reduce_mut(|bill| bill.bill_toggle = 1);
});
...
```

Details Page

This is the page that is shown if you click on the image of one of the items. It will display details about the item selected.

The code grabs the selected item from the menu and uses it to create an organized html element.

```
// details_page.rs
...
let item = get_item_from_id(item_id.to_owned()).unwrap();

html! {
   <>
      <Navbar/>

      <div class={"details"}>
         <h1>{format!("{} □ ${}", item.name, item.price)}</h1>
         <img src={item.image.clone()}/>
         <p>{item.description.clone()}</p>
         <OrderButton item={item}/>
      </div>
   </>
}
...
```

Menu Page

This page will display all the items in the selected category using the `ItemList` component and give you the option to order any of them. If the image of one of the items is clicked, it will take you to the description page of the item.

```
// menu_page.rs
...
let food_list = get_category_from_menu(c);

html! {
   <>
      <Navbar/>

      <div class={"menu"}>
         <h1 class={"categoryHeader"}>{category}</h1>
         <div class={"ItemCategory"}>
            <ItemList items={food_list} />
         </div>
      </div>
   </>
}
...
```

Not Found Page

This is a page that displays when the URL is not recognized

Person Select Page

After the order button is pressed, this page is brought up and it prompts the user to input the name of the person who ordered the item. Once a name is inputed, it creates a new `CartItem` and adds it to the `CartStore`.

```
// person_select_page.rs
...
let onchange = Callback::from(move |person: String| {
      let cart_item = CartItem {
         person: person.clone(),
```

```
            item: item.clone(),
        };

        dispatch.reduce_mut(|cart| cart.cart_items.push(cart_item));

        if !cart.people.contains(&person) {
            dispatch.reduce_mut(|cart| cart.people.push(person).clone());
        }

        navigator.push(&Route::Menu{category: Category::Appetizer.to_string()});
    });
...
```

Stores

Stores are the structs that save the state of the webpage and can be accessed globally. For all the stores used in this project, I chose to save all the data locally, instead of on the server. This was done because i wanted the state to save even if the user reloaded the browser accidentaly. This makes it easier to resume where you left off.

The stores are very small structs and are converted to JSON when they are stored locally. The following code block are the complete code for both stores.

```
// bill_state_store.rs
#[derive(Default, Clone, PartialEq, Serialize, Deserialize, Store)]
#[store(storage = "local")]
pub struct BillState {
    pub bill_toggle: u8,
}


// cart_store.rs
#[derive(Default, Clone, PartialEq, Serialize, Deserialize, Store)]
#[store(storage = "local")]
pub struct CartStore {
    pub people: Vec<String>,
    pub cart_items: Vec<CartItem>,
}
```

Links to resourses used

https://intendednull.github.io/yewdux/

https://yew.rs/docs/getting-started/introduction

https://doc.rust-lang.org/book/

Extras

I was unable to make this website public but if you want to run it yourself you will need to downlaod rust by following the instructions in the rust book, then following the first section of the "yew getting started" tutorial.