Vue Essentials & the Fritter Frontend Starter Code

Intent of this recitation:
1) review Vue basics from lecture
2) introduce new Vue concepts that weren't covered in lecture
3) help familiarize with the starter code

If you want to follow along:

- You can look through the files here: https://github.com/61040-fa22/fritter-frontend
- Or git clone the repo
- If you want to actually run the code, you need to copy in your MONGO_SRV, run npm i, npm run dev, npm run serve

Application structure

- File structure
  - The client/ and server/ directories reflect the client-server model we discussed in class
  - For today's recitation, we're only looking at client/
  - server/ is where you want to copy in your A5 code.
- Components
  - Vue components group by *functionality* instead of the traditional way of separating your template, logic, and styling code (HTML/CSS/JS) into separate files. All of the template, logic, and styling for each component is bundled together in that component's file.
- Components are organized as a **tree**
  - App.vue is the root component (if you imagine the components as a tree, App.vue is the root node of the tree)
  - You can see this organization using Vue DevTools
  - The **"top-level" components** displayed when you navigate to certain URLs (like /login or /account) are shown in client/router.ts.
  - The **"lower level" components** are the general form components we have provided.
- **Mounting** the application
  - public/index.html has a div with id="app" – this is the container. (Note you'll probably never need to touch this file)
  - main.ts "mounts" the Vue application inside of that container
  - The <router-view/> tag is what makes Vue Router work. This tag renders the right component when navigation links are triggered
- Lifecycle Hooks
  - There's a lifecycle diagram but you don't need to understand all the details.

    - "Each Vue instance goes through a series of initialization steps when it's created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes. Along the way, it also runs functions called lifecycle hooks, giving users the opportunity to add their own code at specific stages."
    - A good reference for when to use which hook: https://fjolt.com/article/vue-lifecycle-hooks
    - Example: FreetsPage.vue mounted() - this is used to automatically load Freets into the component when the user visits the page. We can't use beforeCreate() here because we are interacting with the DOM in this function.
    - Example: App.vue beforeCreate() - this is used to load in the username from the store before the App gets mounted onto the page.
        - This is a good example of getting data from the backend (server) to the frontend

Components

- **Data and methods** (example: FreetComponent.vue)
    - **data()** stores data associated with this Vue instance
        - This is where you can manage local state in a component
        - This is a *function*. This is to keep the data values unique for each instance of a component at runtime — the function is invoked separately for each component instance. If you declared data as just an object, all instances of that component would share the same values. This is a side-effect of the way Vue registers components and something you do not want.
        - You should Initialize with default values
        - **Reactivity:** When a Vue instance is created, it adds all the properties found in its data object to Vue's reactivity system. When the values of those properties change, the view will "react", updating to match the new values. Vue is also aware of when the dependencies of computed properties change.
    - **Computed properties**
        - You can data-bind to computed properties in templates just like a normal property. Vue is aware of which computed properties depend on which data, so it will update any bindings that depend on the computed property when data changes.
    - **Methods:** methods associated with the current component that can be used in it
- Example: BlockForm.vue
    - **Attribute binding** (v-bind with shorthand :)
        - Bind built-in HTML attributes like src, class, style, href,
        - Class and style binding
    - **Event Handling** (v-on with shorthand @)
        - Used to attach an event listener to the element

- Example: InlineForm.vue
    - **Form Input Bindings** (v-model and no shorthand)
        - This can be used on inputs of different types (e.g. checkboxes, textareas, dropdown menus, etc.)
        - Two-way synchronizes the state of form input elements with the corresponding state
- Summary of commonly-used directives:
    - **v-on** - Attach an event listener to the element.
    - **v-bind** - overloaded
        - bind one or more HTML attributes OR a component prop to an expression
        - Will talk about props later
        - using JavaScript so that it's dynamic
    - **v-model** - Create a two-way binding on a form input element or a component.
        - You can think of it as combining v-bind, which brings a js value into the markup and v-on:input to update the js value. (The js value must be present in your data)
        - You can achieve v-model with v-bind and v-on, but this v-model combines the two
    - Note that there are more directives but these are some of the most common ones
- FreetPage.vue
    - **List rendering**: v-for
        - The **key** attribute is to help Vue optimize rendering the elements in the list, it tries to patch list elements so it's not recreating them every time the list changes and you need a key on everything to do that
    - **Conditional rending**: v-if, v-else-if, v-else directives control what's rendering based on boolean conditions
- Back to FreetComponent.vue
    - The method submitEdit() is responsible for sending the request to the server. Here is a good example of sending data from the frontend to the backend (server). Note the **error handling on the frontend** (don't waste network and server resources and fail fast!)

State management

- **Parent to child: Props** (example: FreetComponent.vue)
    - props are data that are passed from a parent component to child components
    - FreetComponent has one prop - a freet
    - FreetsPage is the parent component that passes in the freet object as a prop. (Recall that :freet="freet" is shorthand for the directive v-bind:freet)
    - Props are similar to params/inputs in a function and that the value of a prop gives components an initial state that affects their display.

- **Child to Parent**: **custom events with this.$emit** (No examples in starter code)
    - A component can “emit” custom events with data, and the parent can listen for the event
    - Only the direct parent can listen for the event; no other components will receive the event
- **Store** (example: App.vue) (The package we are using for Store is called Vuex.)
    - “Props give us one-way data flow - from parent to child components. What happens when we need multiple components that share a common state?
        - Multiple views may depend on the same piece of state.
        - Actions from different views may need to mutate the same piece of state.
    - So why don't we extract the shared state out of the components, and manage it in a global singleton? With this, our component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!
    - By defining and separating the concepts involved in state management and enforcing rules that maintain independence between views and states, we give our code more structure and maintainability.
    - This is the basic idea behind Vuex”
    - Example in beforeCreate(): when the website is first loaded, we get the currently-logged-in user and save the username to the store
    - In store.ts: here is where the username is stored and the mutation (setUsername) is defined
    - Example in AccountPage.vue: here is how to retrieve the username from the store
- **When to use props and this.$emit vs. when to use store?**
    - Is there a direct hierarchical relationship between my parent and child components that store and use this prop? If yes then use props/events
    - how long must the data persist? If you’re only using this data for one form and then never needing it again, try to use props/events
    - do you need the same data across many “pages”? Then use Vuex
    - will the data be changed frequently and/or quickly? use Vuex because it can manage transactional updating
    - are many components sharing and updating the same data object? If yes, use Vuex
    - Source: When to use vuex store vs events? : r/vuejs

Etc.

- **Mixins**: mixins are used to have reusable logic between components. In this case, mixins have components in client/components/common/
    - Example: BlockForm is a mixin for ChangePassword

    - Mixins are a flexible way to distribute reusable functionalities for Vue components. A mixin object can contain any component options. When a component uses a mixin, all options in the mixin will be “mixed” into the component’s own options.
    - It’s kind of like inheriting classes - e.g. “Dog” inherits “Animal”
- *Concepts covered in lecture that we aren’t reviewing here:*
    - Template syntax (e.g. {{message}})
    - Slots
    - Scoped CSS
    - Router
- Reminder: we are using Vue 2 not Vue 3
    - You might see it as Options API vs Composition API - use Options API