WEB3, Session 4

Components

Routing

- Typically, an app is in different states - corresponding to different screens/tabs/etc.
- Changing between the different states is sometimes called *routing* in web apps.
- Poor Man's Routing:

```
<lobby-view v-if="gameState.mode=='no game'" .../>
<waiting-view v-if="gameState.mode=='waiting'" .../>
<game-view v-if="gameState.mode=='playing'" .../>
```

Problems with Poor Man's Routing

- It gets big and difficult to maintain in large apps
- It doesn't leave a trace in the history
- It breaks the back button
  - Note: sometimes the back button just have to be broken

vue-router

- Same basic idea as React routing (WEB 2)
- Each route is a path
- A path is bound to a component
- You can transfer path and query parameters
- Advice: try to make sure the URLs are usable directly in the browser

Example

```
const routes = [
  { path: '/', component: LobbyView },
  { path: '/waiting/:gameNumber', component: WaitingView },
  { path: '/playing/:gameNumber', component: GameView}
]

const router = VueRouter.createRouter({
  history: VueRouter.createWebHistory(),
  routes
})
```

~~Poor Man's~~ Routing

```
<template>
  <h1 class="header">Yahtzee!</h1>
  <h2 v-if="playerStore.player" class="subheader">
    Welcome player {{playerStore.player}}
  </h2>
  <nav v-if="playerStore.player">…
  </nav>

  <RouterView class='main'/>
</template>
```

Navigating

```
async function join(gameNumber: number) {
  const game = await api.joinGame(gameNumber)
  model.startGame('O', game)
  router.push(`/playing/${game.gameNumber}?player=O`)
}

// Or a direct link:
<RouterLink to="/playing/5?player=O">Go to Game</RouterLink>
```

Path and query parameters

```
const route = useRoute()
const gameNumber = Number.parseInt(route.params.gameNumber as string)
const player = route.query.player as Player
```

A Component

- A visual unit that we can use as normal HTML tags
- A component has
  - A view model
  - Underlying data
  - Properties
  - Events (“emits”)
  - template (Vue HTML template language)
  - style (CSS)
  - declaration of sub-components

State management

- Local State Management
  - The state is mostly defined in the components
  - State is transferred to child components using properties (props)
  - State is transferred to parent components using listeners (emits)
- Global State Management
  - The state is defined in a single place (single source of truth)
  - This can be a global variable (or singleton pattern, if you prefer)
  - The variable need to be reactive
  - Alternatively: Use a library

Props

- A property that we can use like
  - <component :prop=“user”/>
- Defined in the viewmodel
- Typed or untyped
- Optional or required

Emits

- Events that you can use like
  - <component @event=“method”/>
- Declared in the view model
- Either
  - Parameterless
  - or with a parameter and a check for validity

Defining props and emits in <setup script>

```
defineProps<{
  personData: Data[]
}>()
```

```
let emit = defineEmits({
  hire(_: number) {
    return true
  }
})
```

Using emits in <setup script>

```
function hire(id: number) {
  emit('hire', id)
}
```

Using props in the component template

```
<tr v-for='p in personData'>
…
</tr>
```

Using props and emits in the parent component

```
/* In <setup script>:
  function hire(id: number) {...}
*/

<person-view
    :person-data="model.personData()"
    @hire="hire">
</person-view>
```

Component Hierarchy

Avoiding prop drilling

- Prop drilling
  - (From React)
  - When props are forwarded down through the hierarchy to end up at a very low level.
- Provide
  - A component high in the hierarchy can provide one or more objects.
- Inject
  - A component lower in the hierarchy can declare that it wants the property injected.
  - (renaming, default value)

provide and inject

```
// In a parent component:
provide('game', ref(props.game))


// In a child component:
const game = inject('game')
```

Slots

- Usually, HTML tags *surround* other content
- In vue.js components, there is a special tag, `<slot>`, that stands for the surrounded content.
- When rendered, the `<slot>` tag will be replaced with the surrounded content.
- You may know this from HTML Web Components
  - Note: `<template>` and `<slot>` tags get replaced in the vue.js rendering, so they will not appear in the DOM. You can't mix vue.js with HTML Web Components.

A Bit of a Hack
(old yahtzee-graphql/src/App.vue)

```
<template>
  <h1 class="header">Yahtzee!</h1>
  <h2 v-if="playerStore.player" class="subheader">
    Welcome player {{playerStore.player}}
  </h2>
  <nav v-if="playerStore.player">···
  </nav>

  <RouterView class='main'/>
</template>
```

A wrapper for the relevant pages (src/components/Page.vue)

```
1  <script setup lang="ts">…
17 </script>
18
19 <template>
20   <h2 class="subheader">Welcome player {{playerStore.player}}</h2>
21   <nav>…
33   </nav>
34
35   <slot class='main'></slot>
36 </template>
```

Wrapping the page

```
<template>
  <Page v-if="playerStore.player">
    <main>
      Number of players:
        <input min="1" type="number" v-model="number_of_players"/>
      <button @click="new_game(playerStore.player)">New Game</button>
    </main>
  </Page>
</template>
```

Advanced Slots

- Named slots – allows more slots in the component
  - Naming a slot outlet: `<slot name='header'></slot>`
  - Content for that outlet: `<template v-slot:'header'>…</template>`
- Scoped slots
  - Passing properties to slot content: `<slot :count='7'></slot>`
  - Receiving properties in the content: `<MyComponent v-slot='{count}'>`
- Named scoped slots
  - `<slot name='header' :count='7'></slot>`
  - `<template #header='{count}'>…</template>`

Example: Grid

```
<template>
    <table id = 'board'>
        <tr v-for='row in tiles()'>
          <td name='tile' v-for='tile in row' class="cell">
            <slot name='tile' v-bind='tile'></slot>
          </td>
        </tr>
    </table>
</template>
```

Using the Grid component

- 04 Components/tic-tac-toe - routing/components
  - Grid.vue
  - ActiveGame.vue
  - FinishedGame.vue

Component tree

Global State Management

- Global state:
  ```
  const gameState = ref({mode: 'no game'} as GameState)
  ```
- You can either
  - export the ref and import it in all components
  - create an object that uses the variable and export that
  - use the singleton pattern and have the variable as private in a class
- Alternatively, use a library
  - We'll be looking at Pinia

Pinia

- Sometimes touted as the Vue.js equivalent to Redux
- A tool to create a global, reactive state
- Two API versions
    1. Option stores: Like the Vue.js options API
    2. Setup stores: Like the Vue.js composition API
- Allows for better tooling than the simple reactive object
- Safer, if you want to use Vue.js for server-side rendering

Pinia option store

```
export const store = defineStore('model', {
  state() {return {games: [], gameState: {mode: 'no game'}},
  getters: {
    player: () => this.gameState.value.player
  },
  actions: {
    endGame() {
      this.gameState.value = {mode: 'no game'}
    }
  }
})
```

Pinia Composition

```
export const store = defineStore('model', () => {
  const games = ref([] as Game[])
  const gameState = ref({mode: 'no game'})

  const player = computed(() => gameState.value.player)

  function endGame() {
    gameState.value = {mode: 'no game'}
  }

  return {games, gameState, player, endGame}
})
```

Example

- `tic-tac-toe - global state` - the global variable solution
- `tic-tac-toe - pinia` - the Pinia solution

Testing

- `tic-tac-toe – routing/__test__/lobby.test.ts`