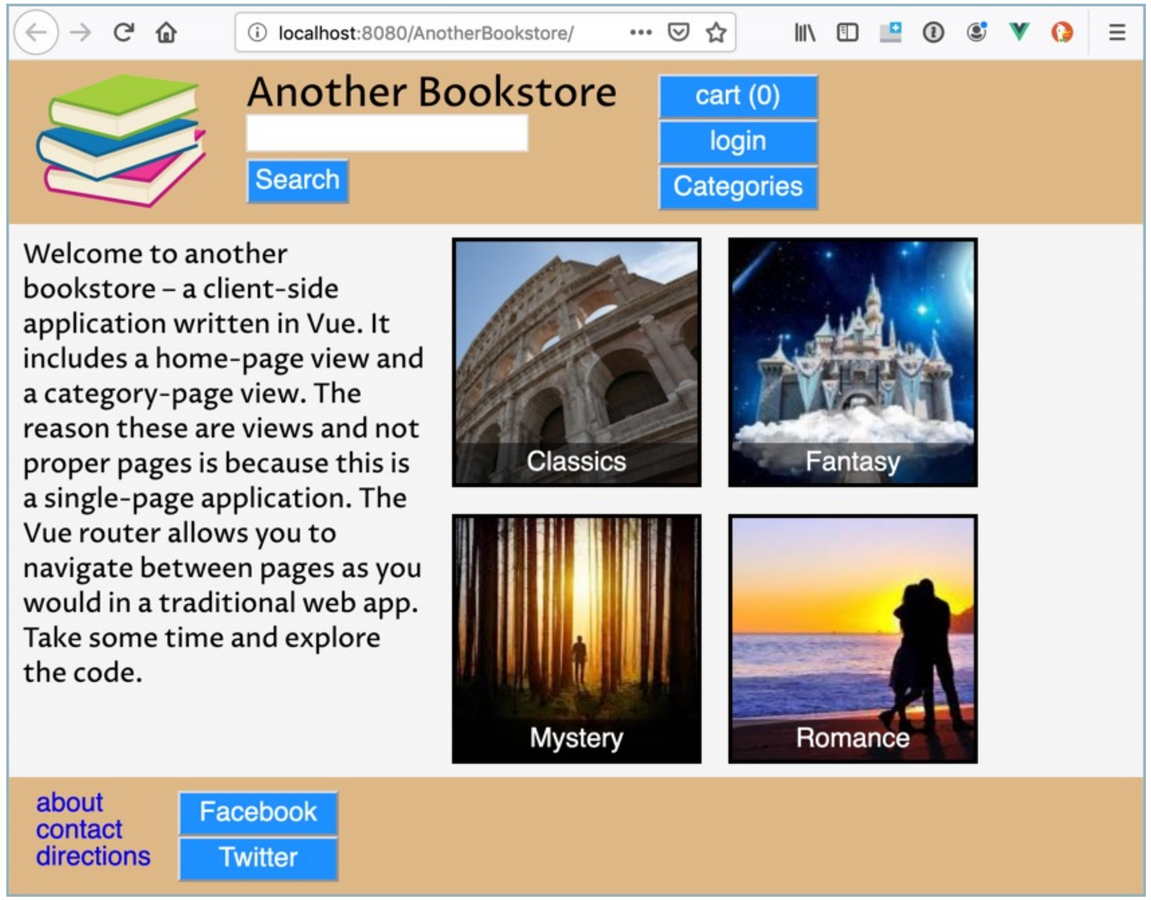
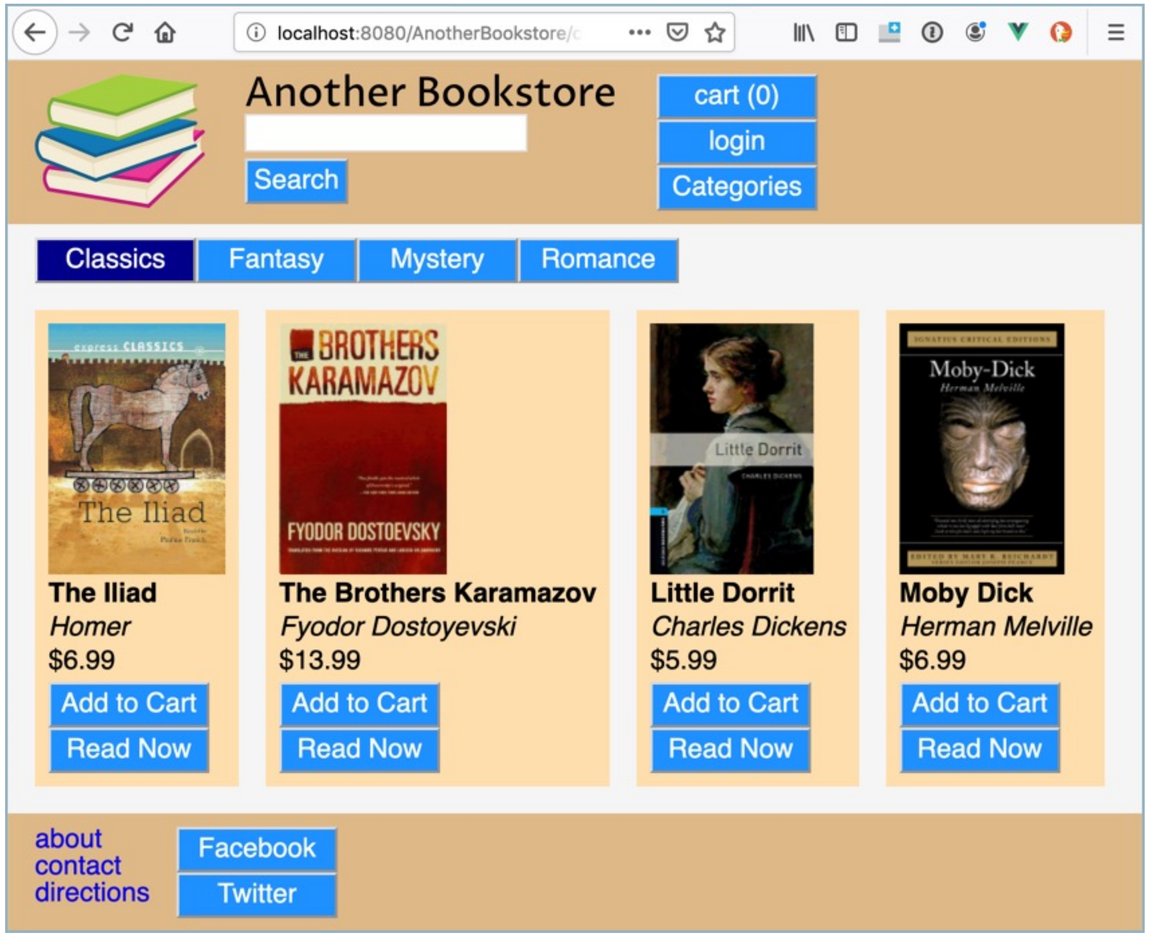
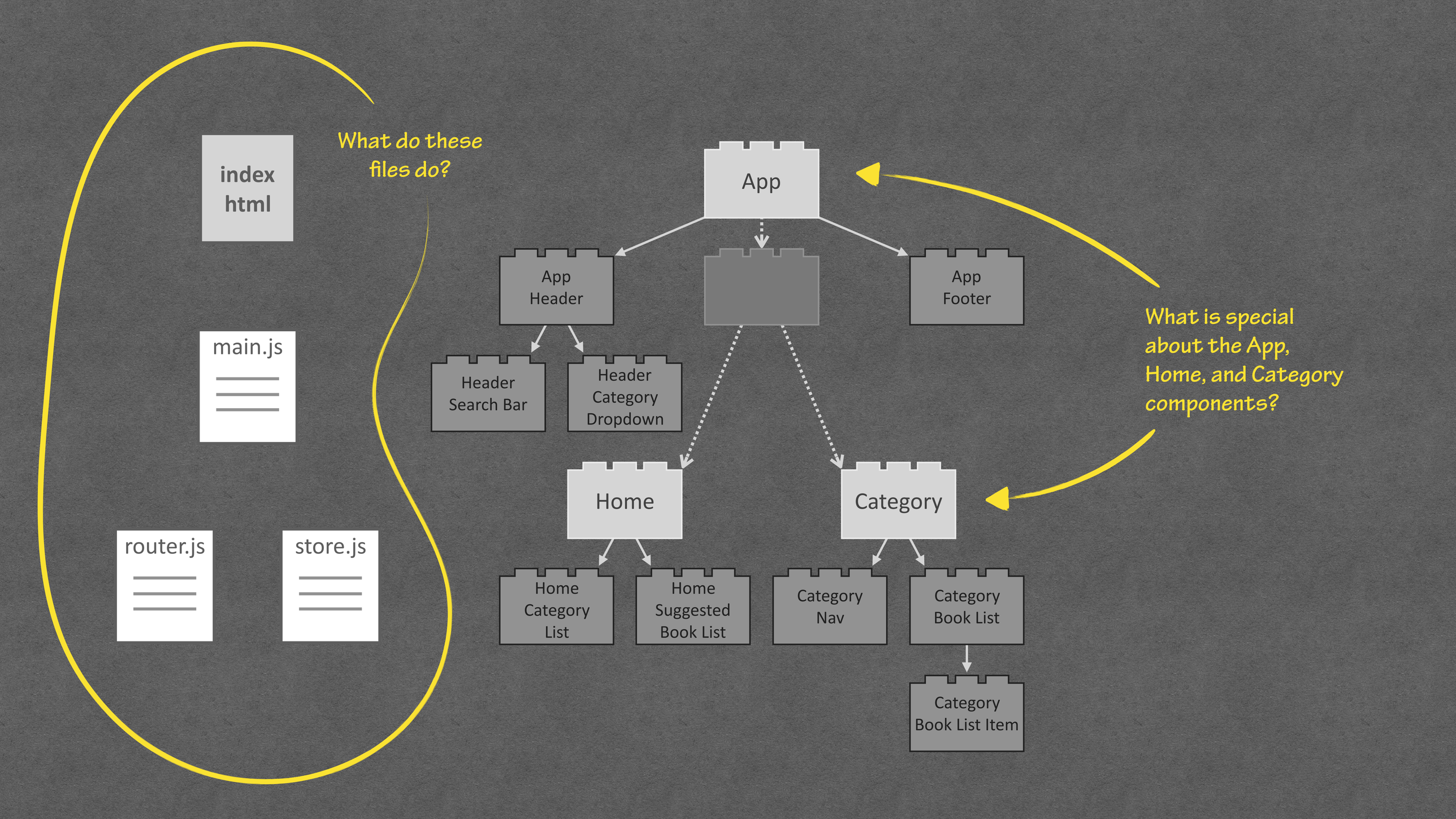
Another Bookstore with Vuex!






What do these files do?
index html
main.js
router.js
store.js
App
App Header
App Footer
Header Search Bar
Header Category Dropdown
Home
Category
Home Category List
Home Suggested Book List
Category Nav
Category Book List
Category Book List Item
What is special about the App, Home, and Category components?

Sole HTML page

index
html

Main JS page

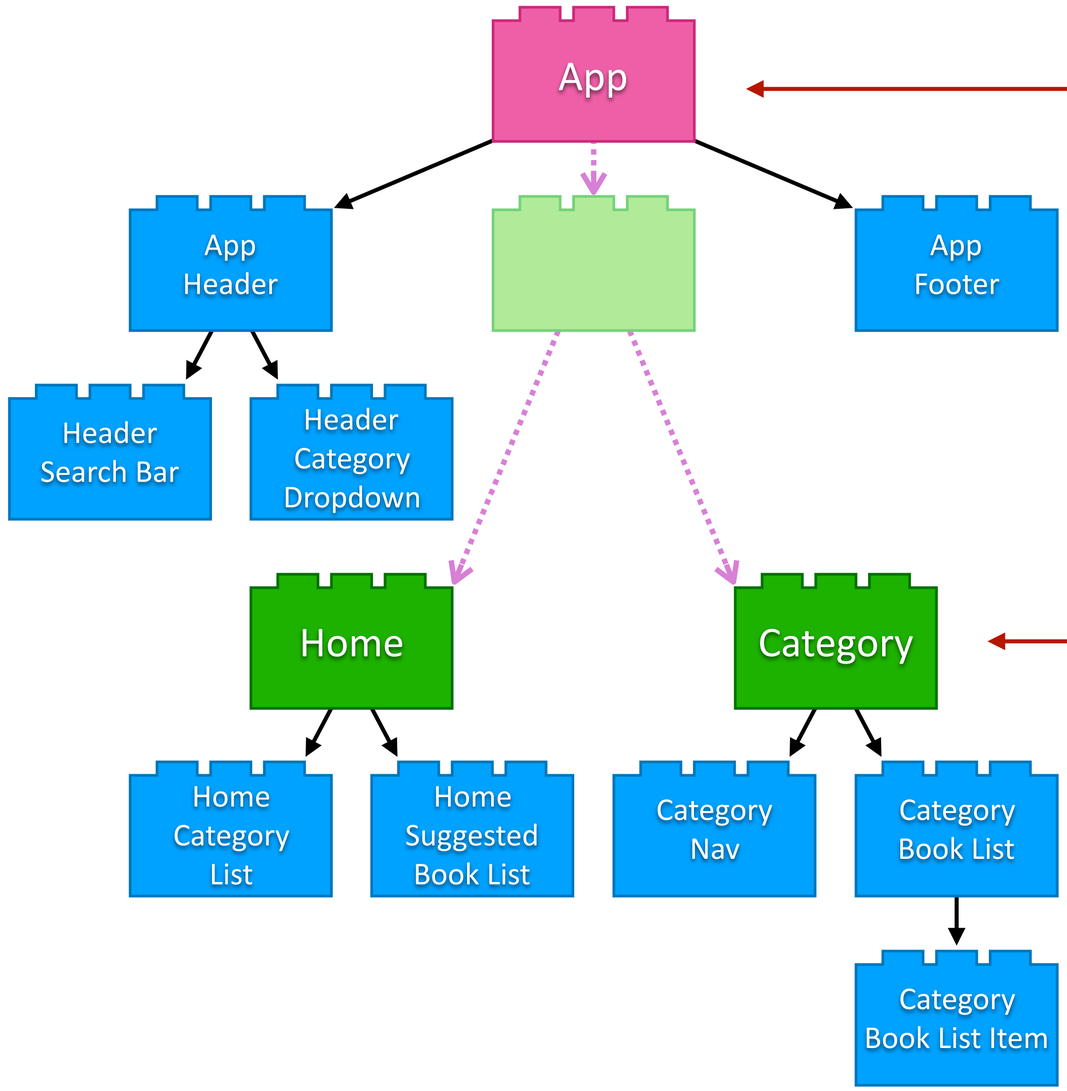
main.js

Vue-router info

router.js

Vuex info

store.js



App.vue is the root Vue component

The Home and Category components represent different pages (or views)

```
<!DOCTYPE html>
<html lang="en">
  <head>
   ...
  </head>
  <body>
    <noscript>
      <strong>JavaScript is required ...
    </noscript>
    <div id="app"></div>
  </body>
</html>
```

index.html

```
import Vue from "vue";
import App from "./App.vue";
import router from "./router";
import store from "./store";
import "@/assets/css/normalize-and-reset.css";
import "@/assets/css/global.css";

Vue.config.productionTip = false;

new Vue({
  router,
  store,
  render: function(h) {
    return h(App);
  }
}).$mount("#app");
```

main.js

```
<template>
  <div id="app">
    <app-header></app-header>
    <router-view
        :key="this.$route.fullPath">
    </router-view>
    <app-footer></app-footer>
  </div>
</template>

<script>
import AppHeader from ...
import AppFooter from ...

export default {
  name: "App",
  components: {
    AppHeader,
    AppFooter
  },
  mounted: function() {
    this.$store.dispatch("fetchCategories");
  }
};
</script>
```

App.vue

What is the relationship between these three files?

What ties them together?

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  </head>
  <body>
    <noscript>
      <strong>JavaScript is required ...
    </noscript>
    <div id="app"></div>
  </body>
</html>
```

index.html

gets replaced with

```vue
<template>
  <div id="app">
    <app-header></app-header>
    <router-view
        :key="this.$route.fullPath">
    </router-view>
    <app-footer></app-footer>
  </div>
</template>

<script>
import AppHeader from ...
import AppFooter from ...

export default {
  name: "App",
  components: {
    AppHeader,
    AppFooter
  },
  mounted: function() {
    this.$store.dispatch("fetchCategories");
  }
};
</script>
```

App.vue

render the hyperscript from the App component

mount this on the element with ID app

```js
import Vue from "vue";
import App from "./App.vue";
import router from "./router";
import store from "./store";
import "@/assets/css/normalize-and-reset.css";
import "@/assets/css/global.css";

Vue.config.productionTip = false;

new Vue({
  router,
  store,
  render: function(h) {
    return h(App);
  }
}).$mount("#app");
```

main.js

```
<!DOCTYPE html>
<html lang="en">
  <head>
   ...
  </head>
  <body>
    <noscript>
      <strong>JavaScript is required ...
    </noscript>
    <div id="app"></div>
  </body>
</html>
```

index.html

```
import Vue from "vue";
import App from "./App.vue";
import router from "./router";
import store from "./store";
import "@/assets/css/normalize-and-reset.css";
import "@/assets/css/global.css";

Vue.config.productionTip = false;

new Vue({
  router,
  store,
  render: function(h) {
    return h(App);
  }
}).$mount("#app");
```

main.js

What does this setting do?

What are router and store?

```
<template>
  <div id="app">
    <app-header></app-header>
    <router-view
        :key="this.$route.fullPath">
    </router-view>
    <app-footer></app-footer>
  </div>
</template>

<script>
import AppHeader from ...
import AppFooter from ...

export default {
  name: "App",
  components: {
    AppHeader,
    AppFooter
  },
  mounted: function() {
    this.$store.dispatch("fetchCategories");
  }
};
</script>
```

App.vue

What does fullPath do?

What is this line doing?

```html
<!DOCTYPE html>
<html lang="en">
  <head>
   ...
  </head>
  <body>
    <noscript>
      <strong>JavaScript is required ...
    </noscript>
    <div id="app"></div>
  </body>
</html>
```

index.html

don't keep displaying messages warning that your are in production mode

use router and store objects from router.js and store.js

```js
import Vue from "vue";
import App from "./App.vue";
import router from "./router";
import store from "./store";
import "@/assets/css/normalize-and-reset.css";
import "@/assets/css/global.css";

Vue.config.productionTip = false;

new Vue({
  router,
  store,
  render: function(h) {
    return h(App);
  }
}).$mount("#app");
```

main.js

```vue
<template>
  <div id="app">
    <app-header></app-header>
    <router-view
          :key="this.$route.fullPath">
    </router-view>
    <app-footer></app-footer>
  </div>
</template>

<script>
import AppHeader from ...
import AppFooter from ...

export default {
  name: "App",
  components: {
    AppHeader,
    AppFooter
  },
  mounted: function() {
    this.$store.dispatch("fetchCategories");
  }
};
</script>
```

App.vue

force the router view to be replaced anytime the router path changes

call action fetchCategories from the Vuex store

```
import Vue from "vue";
import Router from "vue-router";
import Home from "@/views/Home.vue";
import Category from "@/views/Category";

Vue.use(Router);

export default new Router({
  mode: "history",
  base: process.env.BASE_URL,
  routes: [
    {
      path: "/",
      name: "home",
      component: Home
    },
    {
      path: "/category/:name",
      name: "category",
      component: Category,
      props: true
    }
  ]
});
```

router.js

How are the structures of these files similar?

```
import Vue from "vue";
import Vuex from "vuex";
import ApiService from "@/services/ApiService.js";


Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    categories: []
  },
  mutations: {
    SET_CATEGORIES(state, newCategories) {
      state.categories = newCategories;
    }
  },
  actions: {
    fetchCategories(context) {
      ApiService.fetchCategories()
        .then(categories =>
          context.commit("SET_CATEGORIES", categories))
        .catch(reason => {
          console.log("Error loading categories ", reason);
        });
    }
  },
  getters: {}
});
```

store.js

```
import Vue from "vue";
import Router from "vue-router";
import Home from "@/views/Home.vue";
import Category from "@/views/Category";

Vue.use(Router);

export default new Router({
  mode: "history",
  base: process.env.BASE_URL,
  routes: [
    {
      path: "/",
      name: "home",
      component: Home
    },
    {
      path: "/category/:name",
      name: "category",
      component: Category,
      props: true
    }
  ]
});
```

router.js

import Vue

import object

import views

import services

use object

export router

export store

```
import Vue from "vue";
import Vuex from "vuex";
import ApiService from "@/services/ApiService.js";


Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    categories: []
  },
  mutations: {
    SET_CATEGORIES(state, newCategories) {
      state.categories = newCategories;
    }
  },
  actions: {
    fetchCategories(context) {
      ApiService.fetchCategories()
        .then(categories =>
          context.commit("SET_CATEGORIES", categories))
        .catch(reason => {
          console.log("Error loading categories ", reason);
        });
    }
  },
  getters: {}
});
```

store.js

```
import Vue from "vue";
import Router from "vue-router";
import Home from "@/views/Home.vue";
import Category from "@/views/Category";

Vue.use(Router);

export default new Router({
  mode: "history",
  base: process.env.BASE_URL,
  routes: [
    {
      path: "/",
      name: "home",
      component: Home
    },
    {
      path: "/category/:name",
      name: "category",
      component: Category,
      props: true
    }
  ]
});
```

router.js

What does history mode do?

What is the base URL?

What properties should all routes have?

What is :name?

What does props: true do?

```
import Vue from "vue";
import Router from "vue-router";
import Home from "@/views/Home.vue";
import Category from "@/views/Category";

Vue.use(Router);

export default new Router({
  mode: "history",
  base: process.env.BASE_URL,
  routes: [
    {
      path: "/",
      name: "home",
      component: Home
    },
    {
      path: "/category/:name",
      name: "category",
      component: Category,
      props: true
    }
  ]
});
```

router.js

get rid of hash on the end of addresses

same as publicPath in vue.config.js (ex: /AnotherBookstore/)

all routes should have:
- path
- name
- component

:name is a dynamic segment

In Category, you can use
{{ name }} instead of {{ $route.params.name }}

How are Vuex.Store properties
similar to Vue instance properties?

What do mutations do?
What do we know about them?
How are they called?

What do actions do?
What do we know about them?
How are they called?

What do getters do?
Can you give an example of a getter?

```
import Vue from "vue";
import Vuex from "vuex";
import ApiService from "@/services/ApiService.js";


Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    categories: []
  },
  mutations: {
    SET_CATEGORIES(state, newCategories) {
      state.categories = newCategories;
    }
  },
  actions: {
    fetchCategories(context) {
      ApiService.fetchCategories()
        .then(categories =>
          context.commit("SET_CATEGORIES", categories))
        .catch(reason => {
          console.log("Error loading categories ", reason);
        });
    }
  },
  getters: { }
});
```

store.js

mutations are unlike any properties in the Vue instance →

Vuex stores have:
- state (similar to data)
- mutations
- actions (similar to methods)
- getters (similar to computed)

mutations
- change Vuex state
- are synchronous
- allow time-travel debugging
- called using commit

actions
- take a context
- commit mutations
- are asynchronous
- called by Vue components using dispatch

getters are used for non-trivial state access

```
import Vue from "vue";
import Vuex from "vuex";
import ApiService from "@/services/ApiService.js";


Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    categories: []
  },
  mutations: {
    SET_CATEGORIES(state, newCategories) {
      state.categories = newCategories;
    }
  },
  actions: {
    fetchCategories(context) {
      ApiService.fetchCategories()
        .then(categories =>
          context.commit("SET_CATEGORIES", categories))
        .catch(reason => {
          console.log("Error loading categories ", reason);
        });
    }
  },
  getters: {}
});
```

```
firstCategories(state, num) {
  return state.categories.filter(
    category => category.categoryId <= 1000 + num
)}
```

store.js

Vuex

- **state management pattern** + library for Vue.js apps
- **centralized store** for all components in an app
- rules ensure state can only be **mutated predictably**
- integrates with **devtools extension** to provide
  - zero-config **time-travel debugging**
  - **state snapshot** export / import

– https://vuex.vuejs.org/

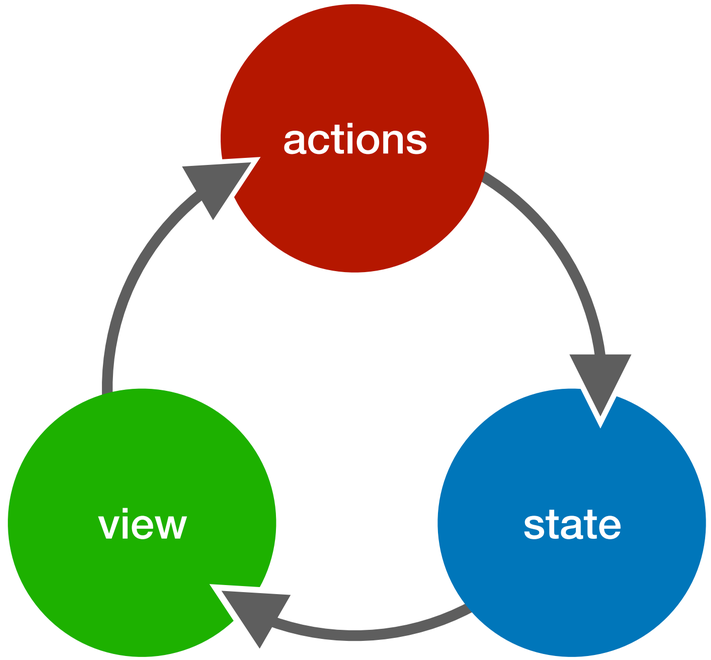
Simple Vue counter app

```
new Vue({
  // state
  data() {
    return {
      count: 0
    };
  },
  // view
  template: `
    <div>{{ count }}</div>
  `,
  // actions
  methods: {
    increment() {
      this.count++;
    }
  }
});
```

Simple Vue counter app

- **state** – source of truth that drives our app
- **view** – a declarative mapping of the **state**
- **actions** – ways the state can change in reaction to user inputs from the **view**.

Problem: Doesn't work when multiple components share a common state



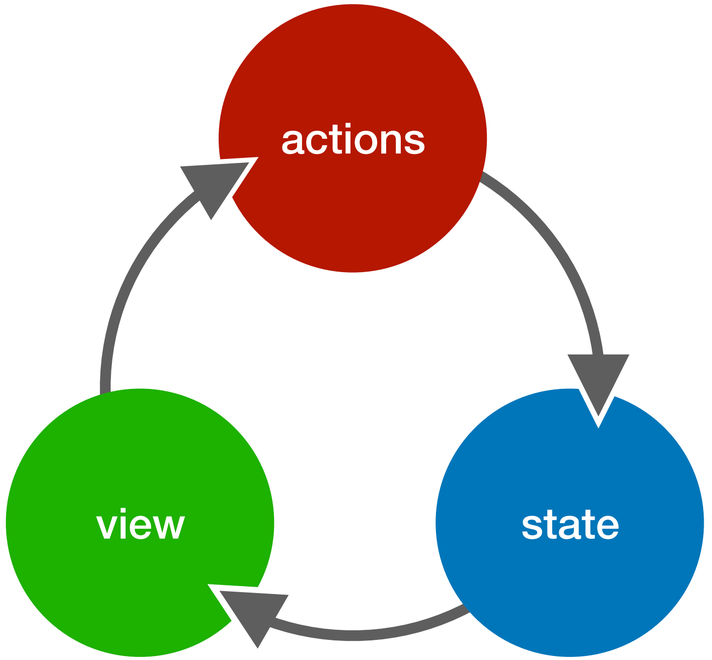
Representation of one-way data flow

– https://vuex.vuejs.org/

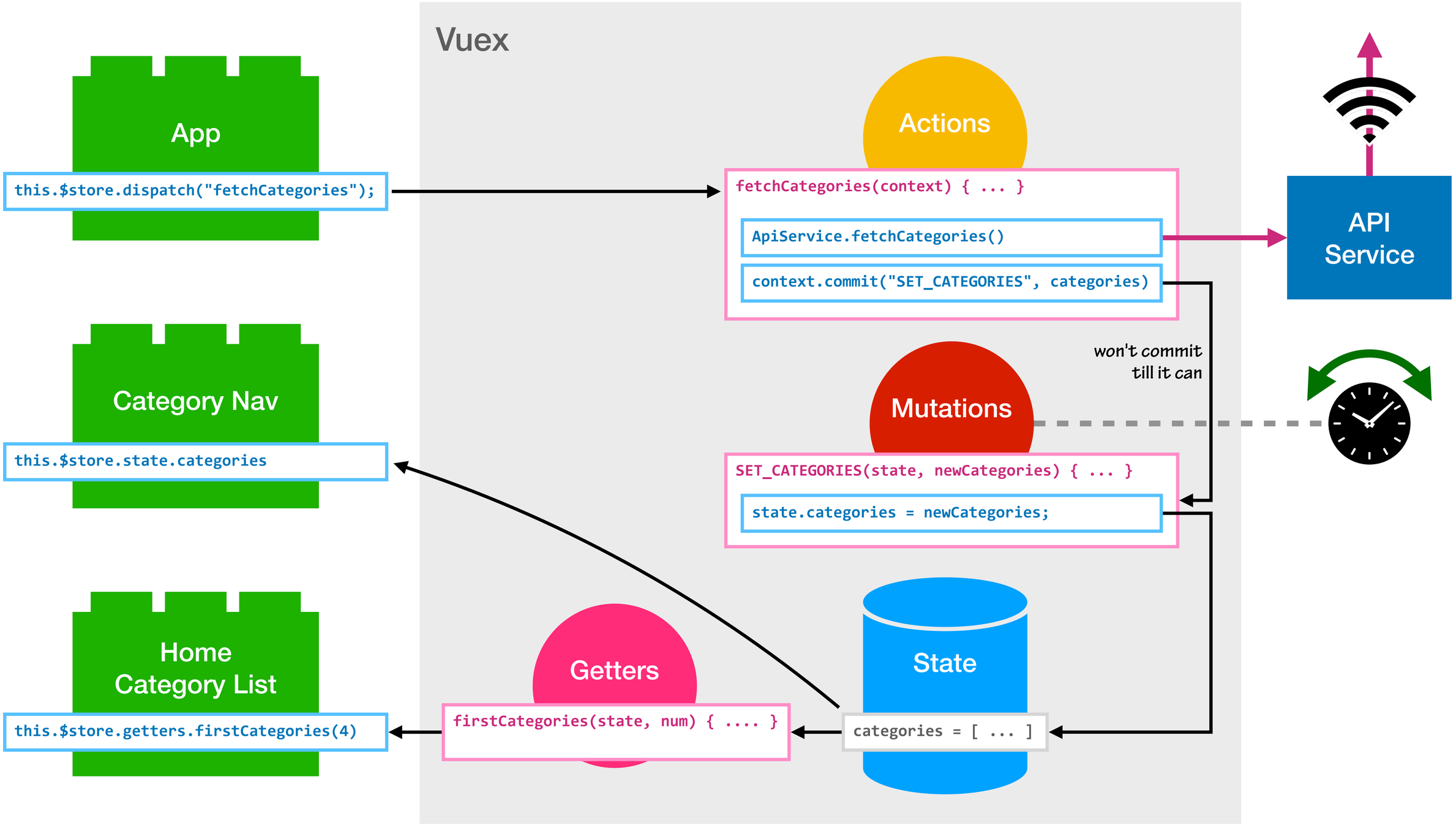
State Management Pattern

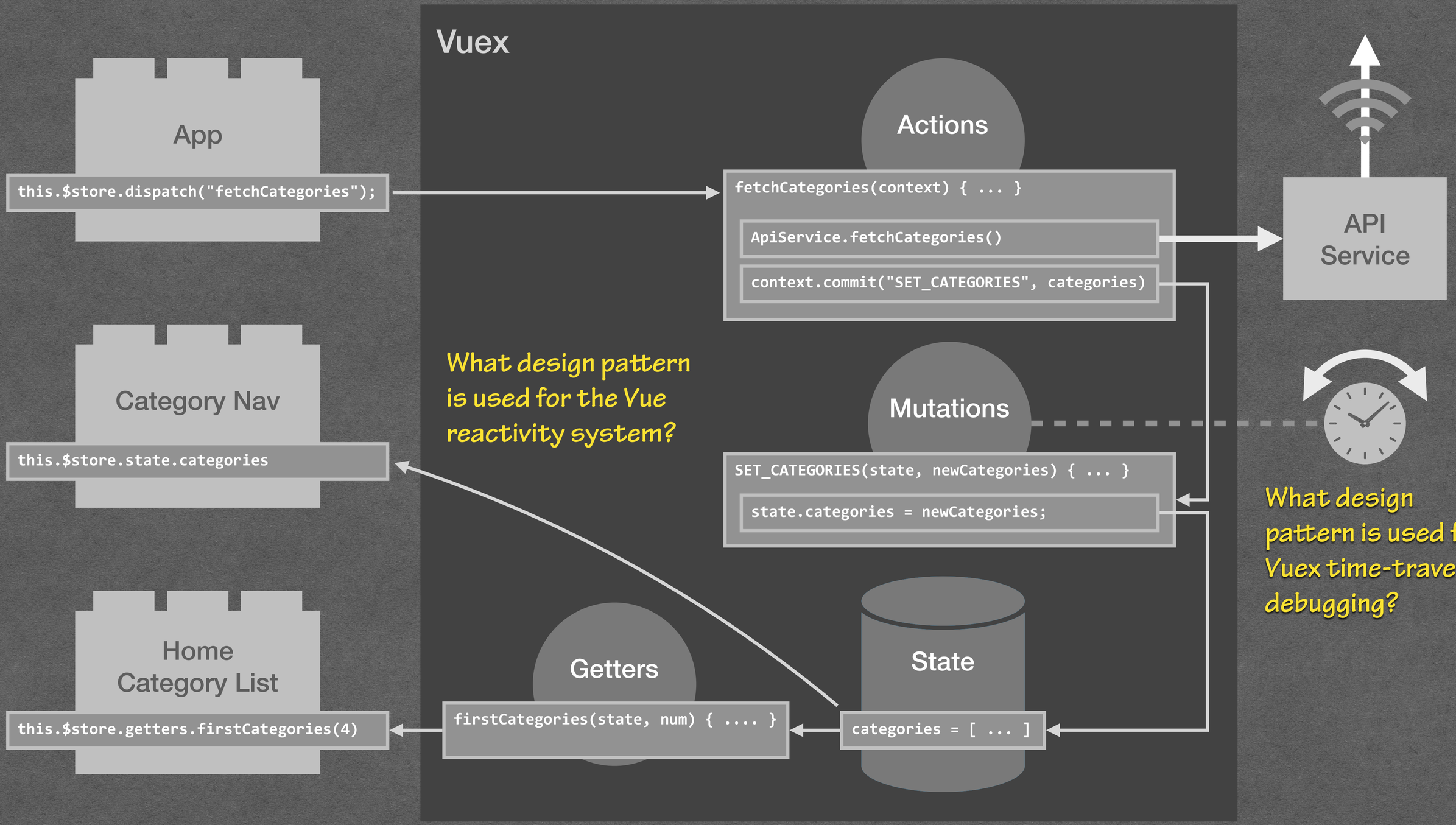
So why don’t we extract the shared state out of the components, and manage it in a global singleton? With this, our component tree becomes a big “view”, and any component can access the state or trigger actions, no matter where they are in the tree!

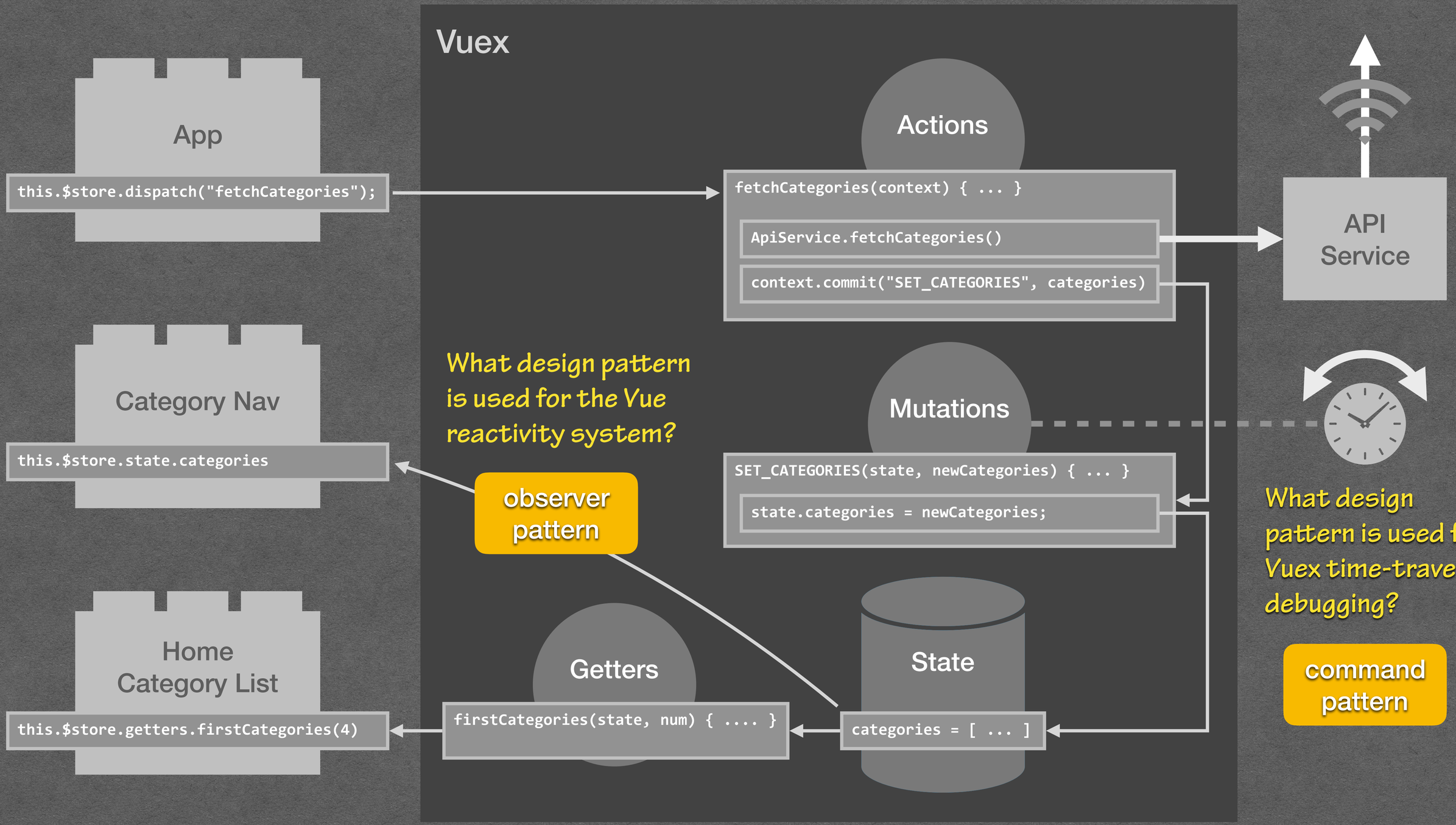
By defining and separating the concepts involved in state management and enforcing rules that maintain independence between views and states, we give our code more structure and maintainability.

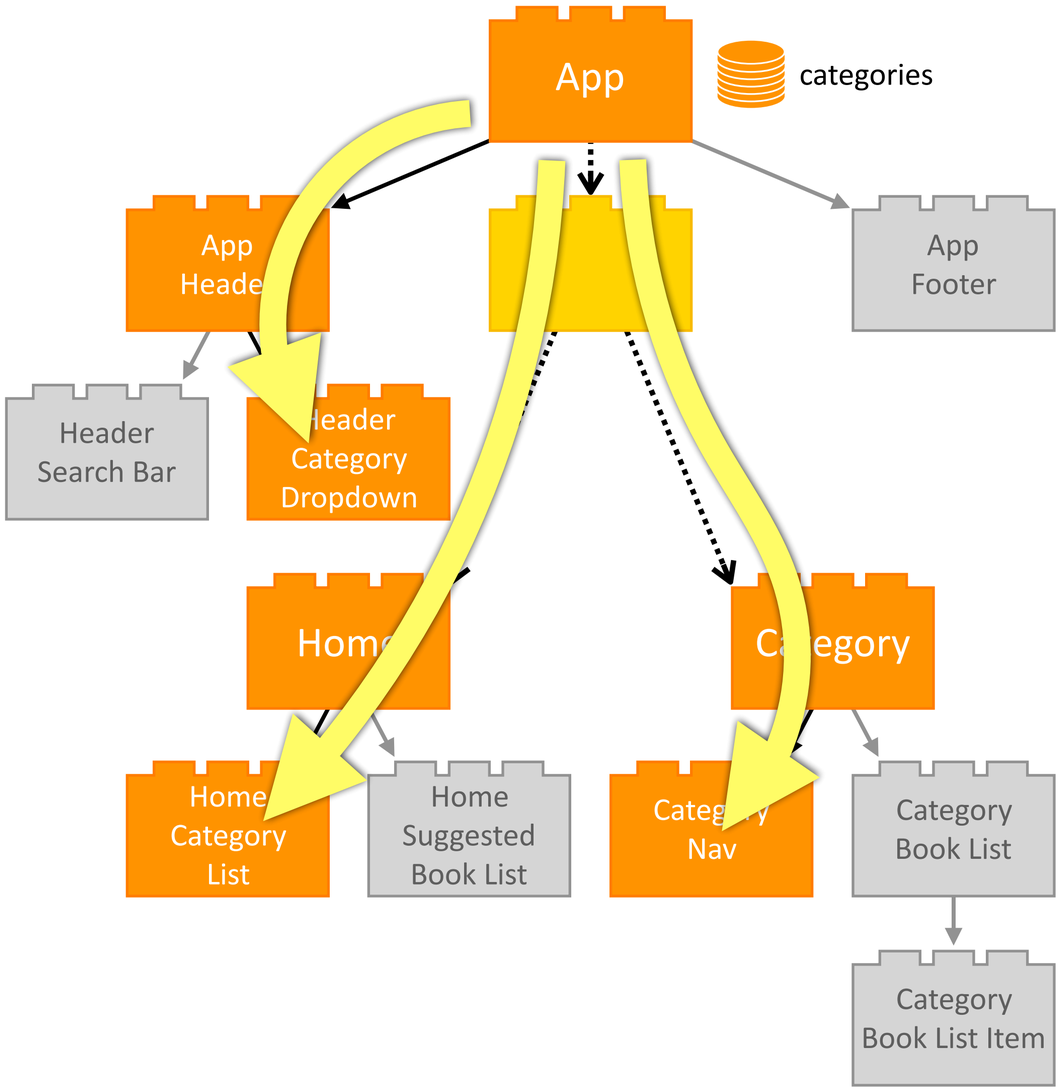


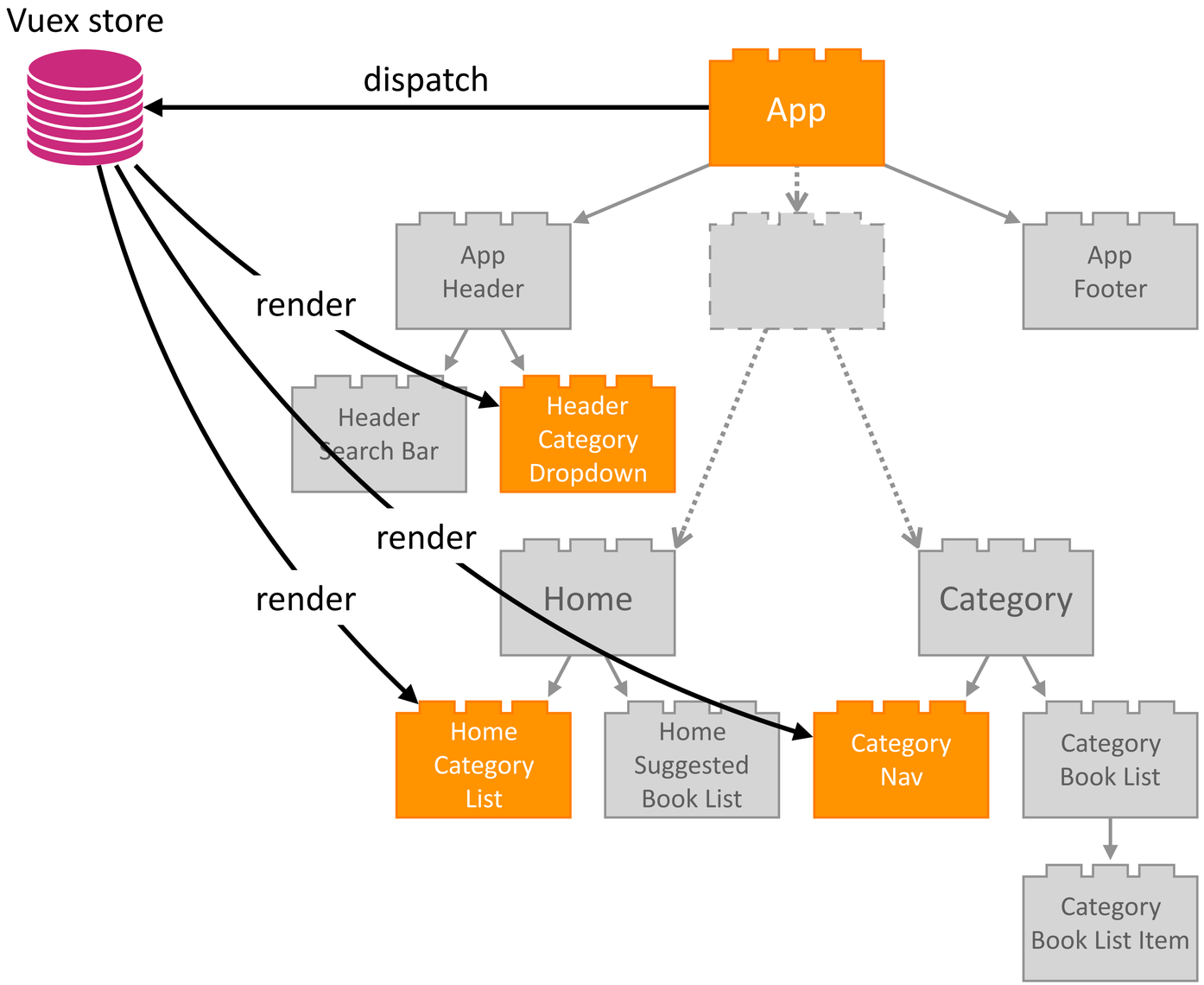
– https://vuex.vuejs.org/


Vuex
App
this.$store.dispatch("fetchCategories");
Actions
fetchCategories(context) { ... }
ApiService.fetchCategories()
context.commit("SET_CATEGORIES", categories)
API Service
won't commit till it can
Mutations
SET_CATEGORIES(state, newCategories) { ... }
state.categories = newCategories;
Category Nav
this.$store.state.categories
State
categories = [...]
Getters
firstCategories(state, num) { .... }
Home Category List
this.$store.getters.firstCategories(4)


Vuex
App
this.$store.dispatch("fetchCategories");
Actions
fetchCategories(context) { ... }
ApiService.fetchCategories()
context.commit("SET_CATEGORIES", categories)
API Service
Category Nav
this.$store.state.categories
What design pattern is used for the Vue reactivity system?
Mutations
SET_CATEGORIES(state, newCategories) { ... }
state.categories = newCategories;
What design pattern is used for Vuex time-travel debugging?
Home Category List
this.$store.getters.firstCategories(4)
Getters
firstCategories(state, num) { .... }
State
categories = [...]


Vuex
App
this.$store.dispatch("fetchCategories");
Actions
fetchCategories(context) { ... }
ApiService.fetchCategories()
context.commit("SET_CATEGORIES", categories)
API Service
Category Nav
this.$store.state.categories
What design pattern is used for the Vue reactivity system?
observer pattern
Mutations
SET_CATEGORIES(state, newCategories) { ... }
state.categories = newCategories;
What design pattern is used for Vuex time-travel debugging?
command pattern
Home Category List
this.$store.getters.firstCategories(4)
Getters
firstCategories(state, num) { .... }
State
categories = [...]


App
categories
App Header
App Footer
Header Search Bar
Header Category Dropdown
Home
Category
Home Category List
Home Suggested Book List
Category Nav
Category Book List
Category Book List Item


Vuex store
dispatch
App
App Header
App Footer
render
Header Search Bar
Header Category Dropdown
render
render
Home
Category
Home Category List
Home Suggested Book List
Category Nav
Category Book List
Category Book List Item

HTML templates from CategoryNav

These templates behave similarly "Another Bookstore"

```
<template>
  <div>
    <template v-for="category in this.$store.state.categories">
      <button
        :key="category.categoryId"
        v-if="selectedCategoryName === category.name"
        id="selected"
      >
        {{ category.name }}
      </button>
      <button :key="category.categoryId" v-else>
        <router-link
          :to="{ name: 'category', params: { name: category.name } }"
        >
          {{ category.name }}
        </router-link>
      </button>
    </template>
  </div>
</template>
```

```
<template>
  <div>
    <button
      v-for="category in this.$store.state.categories"
      :key="category.categoryId"
      :class="{ selected: selectedCategoryName === category.name }"
    >
      <router-link :to="'../category/' + category.name">
        {{ category.name }}
      </router-link>
    </button>
  </div>
</template>
```

What are the differences between the templates?
Which template is more flexible?
What are the differences in the :to expressions?
Which :to expression is more flexible?
Can we delete the in the first template?
Can we delete the in the second template?

HTML templates from CategoryNav

uses conditional rendering (v-if and v-else directives)

```
<template>
  <div>
    <template v-for="category in this.$store.state.categories">
      <button
        :key="category.categoryId"
        v-if="selectedCategoryName === category.name"
        id="selected"
      >
        {{ category.name }}
      </button>
      <button :key="category.categoryId" v-else>
        <router-link
          :to="{ name: 'category', params: { name: category.name } }"
        >
          {{ category.name }}
        </router-link>
      </button>
    </template>
  </div>
</template>
```

- more flexible. allows you to add links (and extra classes if desired) only to unselected category buttons
- :to expression is a route object; name property is used instead of path property; more flexible because it allows you to work with multiple params if necessary

uses class binding (add "selected" class when condition is true)

```
<template>
  <div>
    <button
      v-for="category in this.$store.state.categories"
      :key="category.categoryId"
      :class="{ selected: selectedCategoryName === category.name }"
    >
      <router-link :to="'../category/' + category.name">
        {{ category.name }}
      </router-link>
    </button>
  </div>
</template>
```

- every button has a link, but the page does not change when selected button is pressed, so you can't tell something is happening
- to expression is a string object; Vue router assumes it's the value of the path property

Neither template can omit the root element; an element with a v-for directive is treated like multiple elements by Vue