# 4 AJAX Patterns For Vue.js Apps

If you ask two Vue.js developers "what's the best way to use AJAX in an app?", you'll get three different opinions.

Vue doesn't provide an official way of implementing AJAX, and there are a number of different design patterns that may be used effectively. Each comes with its own pros and cons, and should be judged based on the requirements. You may even use several, simultaneously!

In this article, I'll show you four places you can implement AJAX in a Vue app:

1. Root instance
2. Components
3. Vuex actions
4. Route navigation guards

I'll explain each approach, give an example, and cover the pros and cons as well.

## 1. Root instance

With this architecture, you issue all your AJAX requests from the root instance, and store all state there too. If any sub components need data, it will come down as props. If sub components need refreshed data, a custom event will be used to prompt the root instance to request it.

Example:

```
new Vue({
  data: {
    message: ''
  },
  methods: {
    refreshMessage(resource) {
      this.$http.get('/message').then((response) {
        this.message = response.data.message;
      });
    }
  }
})

Vue.component('sub-component', {
  template: '<div>{{ message }}</div>',
  props: [ 'message' ]
  methods: {
    refreshMessage() {
      this.$emit('refreshMessage');
    }
  }
});
```

**Pros**

- Keeps all your AJAX logic and data in one place.
- Keeps your components "dumb" so they can focus on presentation.

**Cons**

A lot of props and custom events needed as your app expands.

## 2. Components

With this architecture, components are responsible for managing their own AJAX requests and state independently. In practice, you'll probably want to create several "container" components that manage data for their own local group of "presentational" components.

For example, `filter-list` might be a container component wrapping `filter-input` and `filter-reset`, which serve as presentational components. `filter-list` would contain the AJAX logic, and would manage data for all the components in this group, communicating via props and events.

> See <u>Presentational and Container Components</u> by Dan Abramov for a better description of this pattern.

To make implementation of this architecture easier, you can abstract any AJAX logic into a mixin, then use the mixin in a component to make it AJAX-enabled.

```
let mixin = {
  methods: {
    callAJAX(resource) {
      ...
    }
  }
}

Vue.component('container-comp', {

  template: '<div><presentation-comp :mydata="mydata"></presentation-comp>
</div>',
  mixins: [ myMixin ],
  data() {
    return { ... }
  },

})

Vue.component('presentation-comp', {
  template: <div>I just show stuff like {{ mydata }}</div>,
  props: [ 'mydata' ]
})
```

### Pros

- Keeps components decoupled and reusable.
- Gets the data when and where it's needed.

### Cons

- Not easy to communicate data with other components or groups of components.
- Components can end up with too many responsibilities and duplicate functionality.

## 3. Vuex actions

With this architecture, you manage both state and AJAX logic in your Vuex store. Components can request new data by dispatching an action.

If you implement this pattern, it's a good idea to return a promise from your action so you can react to the resolution of the AJAX request e.g. hide the loading spinner, re-enable a button etc.

```
store = new Vuex.Store({
  state: {
    message: ''
  },
  mutations: {
    updateMessage(state, payload) {
      state.message = payload
    }
  },
  actions: {
    refreshMessage(context) {
      return new Promise((resolve) => {
        this.$http.get('...').then((response) => {
          context.commit('updateMessage', response.data.message);
          resolve();
        });
      });
    }
  }
});

Vue.component('my-component', {
  template: '<div>{{ message }}</div>',
  methods: {
    refreshMessage() {
      this.$store.dispatch('refeshMessage').then(() => {

      });
    }
  },
  computed: {
    message: { return this.$store.state.message; }
  }
});
```

I like this architecture because it decouples your state and presentation logic nicely. If you're using Vuex, this is the way to go. If you're not using Vuex, this might be a good enough reason to.

**Pros**

> All the pros of the root component architecture, without needing props and custom events.

**Cons**

> Adds the overhead of Vuex.

## 4. Route navigation guards

With this architecture, your app is split into pages, and all data required for a page and its sub components is fetched when the route is changed.

The main advantage to this approach is that it really simplifies your UI. If components are independently getting their own data, the page will re-render unpredictably as component data gets populated in an arbitrary order.

A neat way of implementing this is to create endpoints on your server for each page e.g. `/about` , `/contact` etc, which match the route names in your app. Then you can implement a generic `beforeRouteEnter` hook that will merge all the data properties into the page component's data:

```
import axios from 'axios';

router.beforeRouteEnter((to, from, next) => {
  axios.get(`/api${to.path}`).then(({ data }) => {
    next(vm => Object.assign(vm.$data, data))
  });
})
```

**Pros**

> Makes the UI more predictable.

**Cons**

- Slower overall, as the page can't render until all the data is ready.

- Not much help if you don't use routes.

## Bonus pattern: server-render the first AJAX call into the page

It's not advisable to use AJAX to retrieve application state on the initial page load, as it requires an extra round-trip to the server that will delay your app from rendering.

Instead, inject initial application state into an inline script in the head of the HTML page so it's available to the app as a global variable as soon as it's needed.

```
<html>
...
<head>
  ...
  <script type="text/javascript">
  window.__INITIAL_STATE__ = '{ "data": [ ... ] }';
  </script>
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

AJAX can then be used more appropriately for subsequent data fetches.

If you're interested in learning more about this architecture, check out my article Avoid This Common Anti-Pattern In Full-Stack Vue/Laravel Apps.

> Thanks to React AJAX Best Practices by Andrew H Farmer for inspiration.