Frontend State Management

Architecture and practical ideas

Alexey Volkov, March 2026

What about you?

Alexey Volkov

- Founder @ Clickly
- Product engineer
- 20+ years of software engineering experience

What is the **frontend state**?

Bad practices

Bad practices

- State in the DOM

Bad practices

- State in the DOM
- Spaghetti state

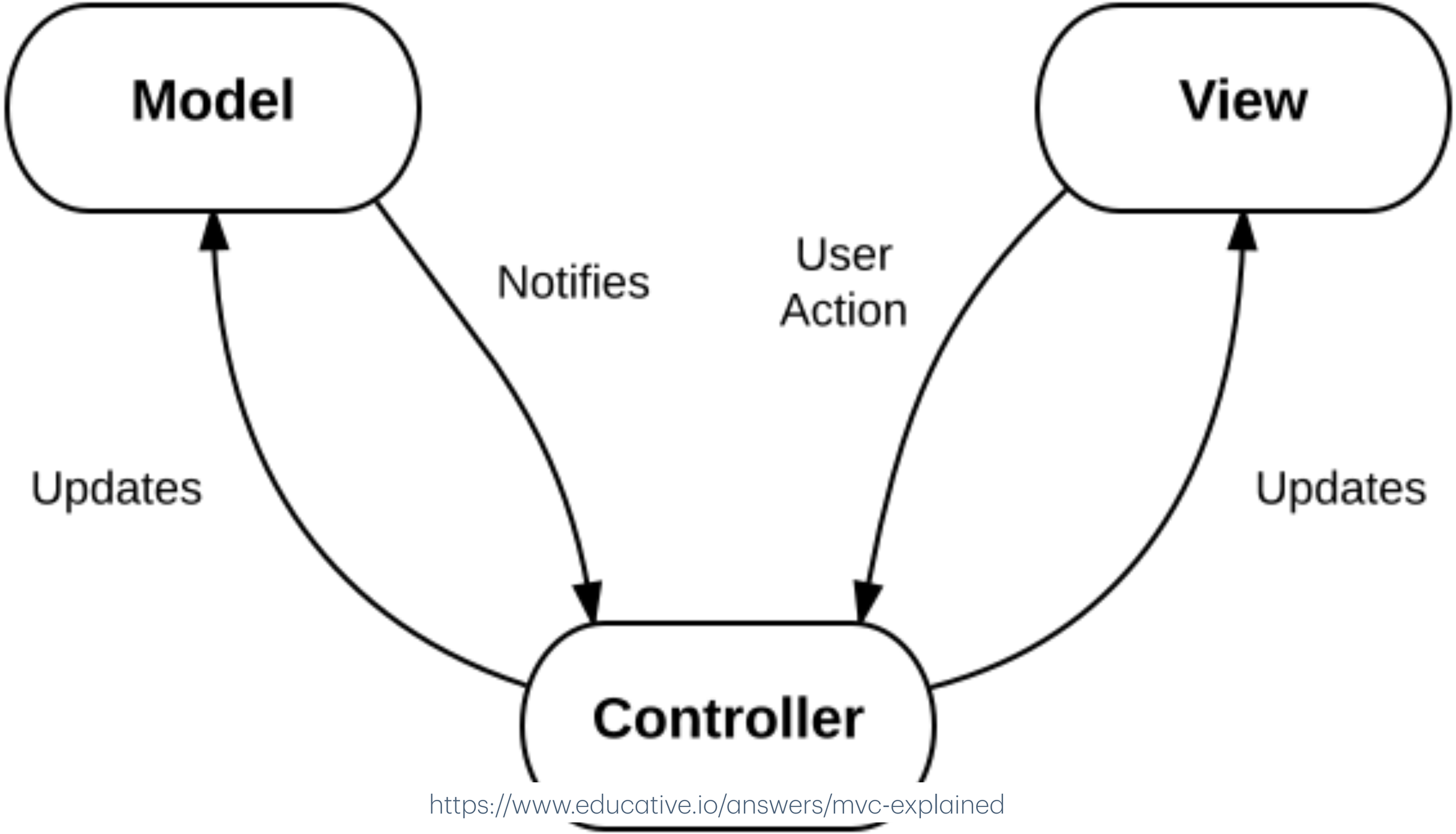
Bad practices

- State in the DOM
- Spaghetti state
- Use whatever your framework offers you

Bad practices

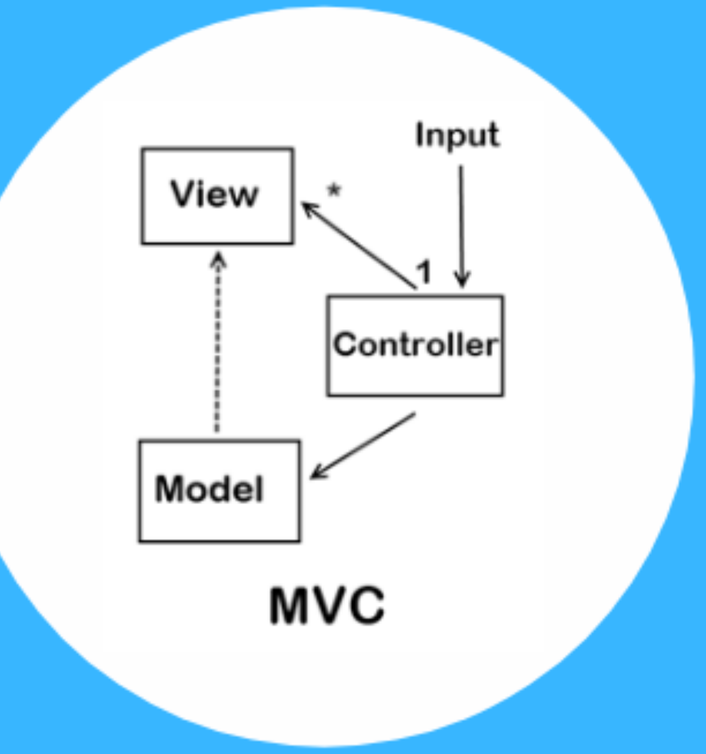
- State in the DOM
- Spaghetti state
- Use whatever your framework offers you
- One library for everything

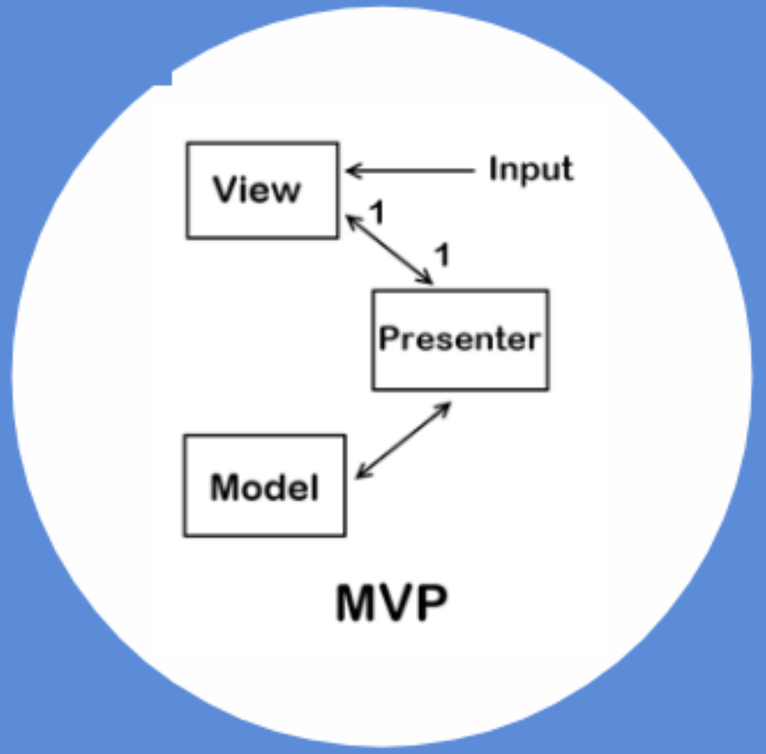
High-level intro to UI app architecture

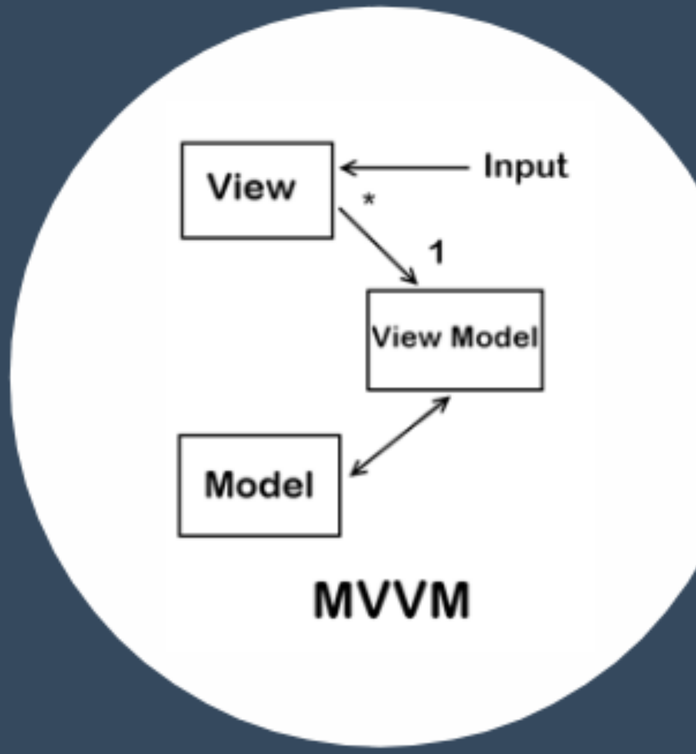


https://www.educative.io/answers/mvc-explained

MVC VS MVP VS MVVM

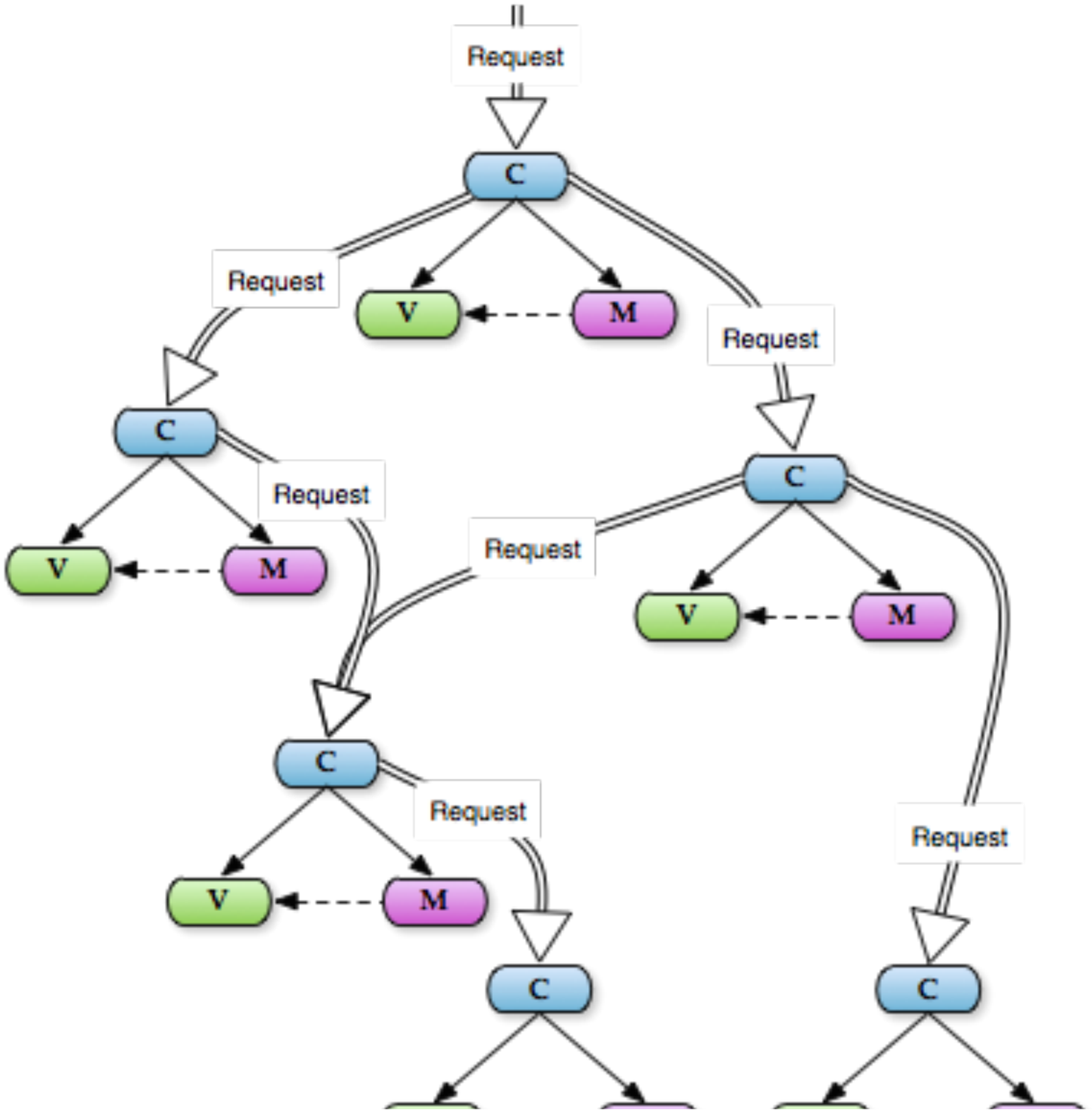




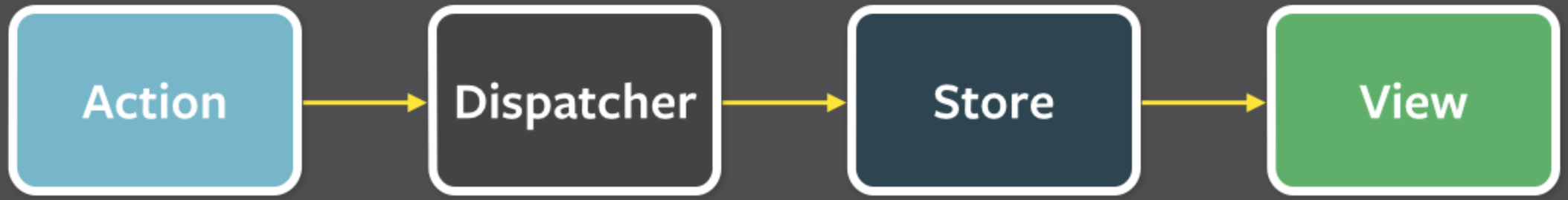


https://blog.stackademic.com/understanding-mvc-mvvm-and-mvp-a-comprehensive-comparison-324fd6e3c730

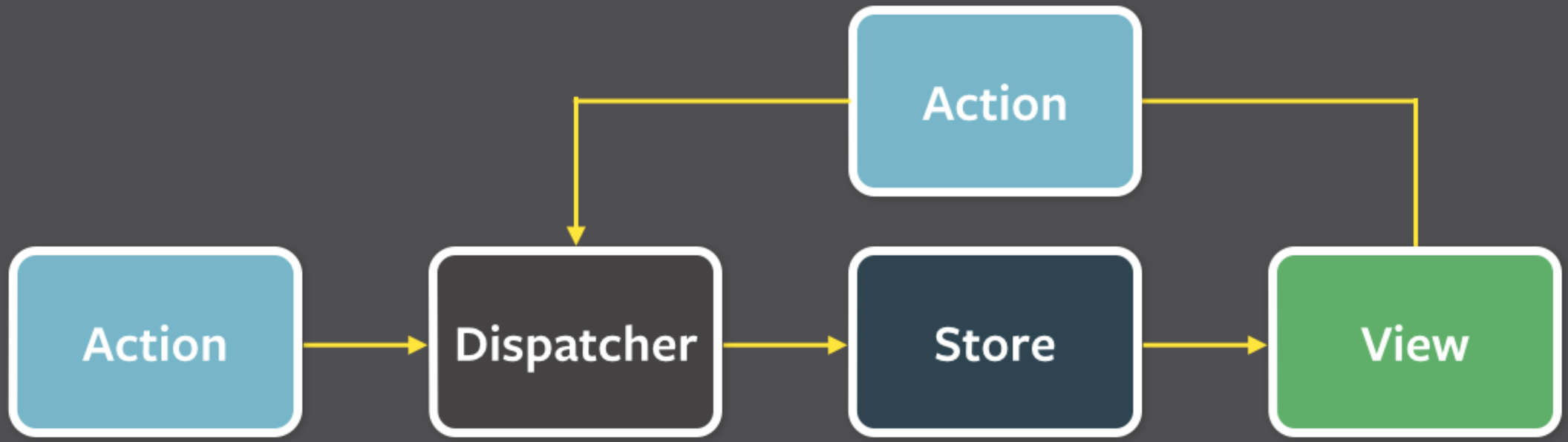
Hierarchical-Model-View-Controller



https://www.oligriffiths.com/2015/01/06/HMVC-and-why-you-should-use-it/



https://facebookarchive.github.io/flux/



https://facebookarchive.github.io/flux/

States are different

States are different

- Forms (capture the data, validate it, process it)

States are different

- Forms (capture the data, validate it, process it)
- Routing (path, params)

States are different

- Forms (capture the data, validate it, process it)
- Routing (path, params)
- Data tables/grids (sorting, filtering, pagination)

States are different

- Forms (capture the data, validate it, process it)
- Routing (path, params)
- Data tables/grids (sorting, filtering, pagination)
- Server state (fetch and update, aka queries and mutations)

States are different

- Forms (capture the data, validate it, process it)
- Routing (path, params)
- Data tables/grids (sorting, filtering, pagination)
- Server state (fetch and update, aka queries and mutations)
- UI state (menu, modals, selected theme, etc.)

Some advice

Some advice

- Keep different types of states separately

Some advice

- Keep different types of states separately
- Integrate different types of states wisely (e.g., data grid sorting and routing; capture form data -> validate -> submit)

Some advice

- Keep different types of states separately
- Integrate different types of states wisely (e.g., data grid sorting and routing; capture form data -> validate -> submit)
- Consider different solutions for different types of states (instead of relying on one universal state management library, e.g., redux or rxjs)

Different libraries for different types of states

Server state

- **React:** react-query (aka TanStack Query) / swr / apollo client / RTK Query
- **Angular:** NgRx effects / @ngneat/query / Apollo
- **Vue:** Vue Query / Apollo Client / swrv

```
import useSWR from 'swr'

function Profile ({ userId }) {
  const { data, error, isLoading } = useSWR(`/api/user/${userId}`, fetcher)

  if (error) return <div>failed to load</div>
  if (isLoading) return <div>loading...</div>

  // render data
  return <div>hello {data.name}!</div>
}
```

https://swr.vercel.app/docs/getting-started

Forms

- **React:** react-hook-form/formik + zod/yup + react-query/swr
- **Angular:** Reactive Forms / ngx-formly
- **Vue:** vee-validate, vue use form

COPY CODESANDBOX JS

```jsx
import { useForm } from "react-hook-form"
import { yupResolver } from "@hookform/resolvers/yup"
import * as yup from "yup"

const schema = yup
  .object({
    firstName: yup.string().required(),
    age: yup.number().positive().integer().required(),
  })
  .required()

export default function App() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm({
    resolver: yupResolver(schema),
  })
  const onSubmit = (data) => console.log(data)

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName")} />
      <p>{errors.firstName?.message}</p>

      <input {...register("age")} />
      <p>{errors.age?.message}</p>

      <input type="submit" />
    </form>
  )
}
```

https://react-hook-form.com/get-started

Routing

- **React:** frameworks (Nextjs, React Router, Tanstack Start, Expo), Tanstack Router, wouter
  - File-based vs. Code-based
- **Angular:** built-in
- **Vue:** Vue Router

Data tables/grids

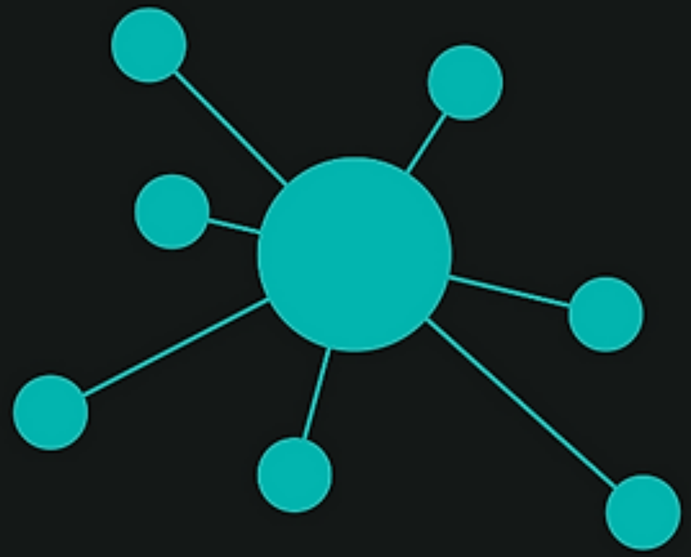
- **React:** Tanstack Table (react-table), ag-grid
- **Angular:** Tanstack Table, ag-grid
- **Vue:** Tanstack Table, ag-grid

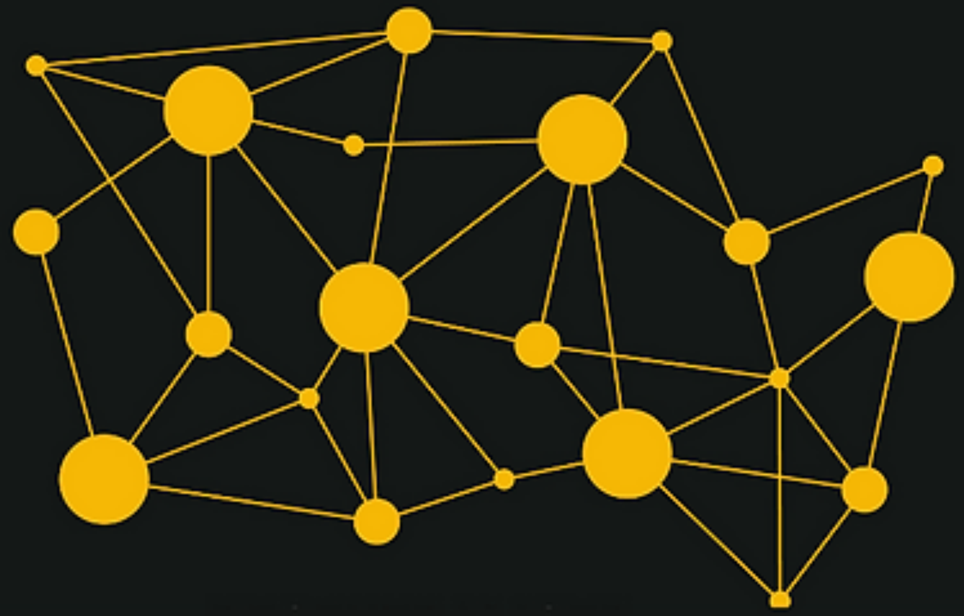
Miscellaneous UI state

- **React:**
  - global: redux, zustand
  - atomic: recoil, jotai
- **Angular:** NgRx
- **Vue:** Pinia

Two main approaches to manage state

CENTRALIZED





DECENTRALIZED

INNO https://www.innovx.org/post/centralized-vs-decentralized-systems-understanding-the-businesses-impacts

Zustand

```ts
import { create } from 'zustand'

type State = {
  count: number
}

type Actions = {
  increment: (qty: number) => void
  decrement: (qty: number) => void
}

const useCountStore = create<State & Actions>((set) => ({
  count: 0,
  increment: (qty: number) => set((state) => ({ count: state.count + qty })),
  decrement: (qty: number) => set((state) => ({ count: state.count - qty })),
}))
```

https://zustand.docs.pmnd.rs/getting-started/comparison

Redux

```
import { createStore } from 'redux'
import { useSelector, useDispatch } from 'react-redux'

type State = {
  count: number
}

type Action = {
  type: 'increment' | 'decrement'
  qty: number
}

const countReducer = (state: State, action: Action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + action.qty }
    case 'decrement':
      return { count: state.count - action.qty }
    default:
      return state
  }
}

const countStore https://zustand.docs.pmnd.rs/getting-started/comparison
```

Jotai

```
import { atom } from 'jotai'

const countAtom = atom<number>(0)
```

https://zustand.docs.pmnd.rs/getting-started/comparison

```
import { useAtom } from 'jotai'

import { animeAtom } from './atoms'

const AnimeApp = () => {
  const [anime, setAnime] = useAtom(animeAtom)

  return (
    <>
      <ul>
        {anime.map((item) => (
          <li key={item.title}>{item.title}</li>
        ))}
      </ul>
      <button onClick={() => {
        setAnime((anime) => [
          ...anime,
          {
            title: 'Cowboy Bebop',
            year: 1998,
            watched: false
          }
        ])
      }}>
        Add Cowboy Bebop
      </button>
    </>
  )
}
```

https://jotai.org/

```tsx
import { atom, useAtom } from 'jotai'

// Create your atoms and derivatives
const textAtom = atom('hello')
const uppercaseAtom = atom(
  (get) => get(textAtom).toUpperCase()
)

// Use them anywhere in your app
const Input = () => {
  const [text, setText] = useAtom(textAtom)
  const handleChange = (e) => setText(e.target.value)
  return (
    <input value={text} onChange={handleChange} />
  )
}

const Uppercase = () => {
  const [uppercase] = useAtom(uppercaseAtom)
  return (
    <div>Uppercase: {uppercase}</div>
  )
}

// Now you have the components
const App = () => {
  return (
    <>
      <Input />
      <Uppercase />
    </>
  )
}
```

https://jotai.org/

```tsx
import { useAtom } from 'jotai'
import { atomWithStorage } from 'jotai/utils'

// Set the string key and the initial value
const darkModeAtom = atomWithStorage('darkMode', false)

const Page = () => {
  // Consume persisted state like any other atom
  const [darkMode, setDarkMode] = useAtom(darkModeAtom)
  const toggleDarkMode = () => setDarkMode(!darkMode)
  return (
    <>
      <h1>Welcome to {darkMode ? 'dark' : 'light'} mode!</h1>
      <button onClick={toggleDarkMode}>toggle theme</button>
    </>
  )
}
```

https://jotai.org/

Let’s summarize

1. Besides separating state/model, logic/controller and view, keep different types of states also separately
    - Server state
    - Forms
    - Routing
    - Data tables/grids
    - General UI state
2. Different problems require different solutions (there is no silver bullet and no one-size-fits-all)

I love your questions