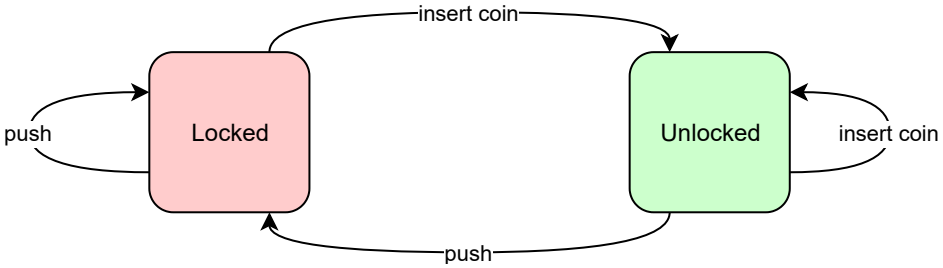
State Machines

This lesson talks about state machines, a concept closely related to discriminated unions.

WE'LL COVER THE FOLLOWING

Introduction to state machines

Managing the state is one of the hardest parts of frontend development. Many frameworks and libraries have been created to help with this task. One of the proposed approaches is to utilize **state machines**. A state machine is a directed graph where nodes represent all possible states of a view or of the whole app, and where edges represent possible transitions between the states.



Classic example of a state machine representing a turnstile. Turnstile has two states: locked and unlocked. You can unlock it by inserting a coin. If you push it, it's locked again. Pushing a locked turnstile doesn't affect the state. So doesn't inserting a coin into an unlocked turnstile.

An arrow between two nodes means that it's possible to go through one state to another via some action. All non-listed transitions are not possible. There can be a meaningful transition from a state to the same state that is marked by a circular arrow.

Using state machines is an explicit and safe way of dealing with the state. They're particularly useful when the state is very complex and involves

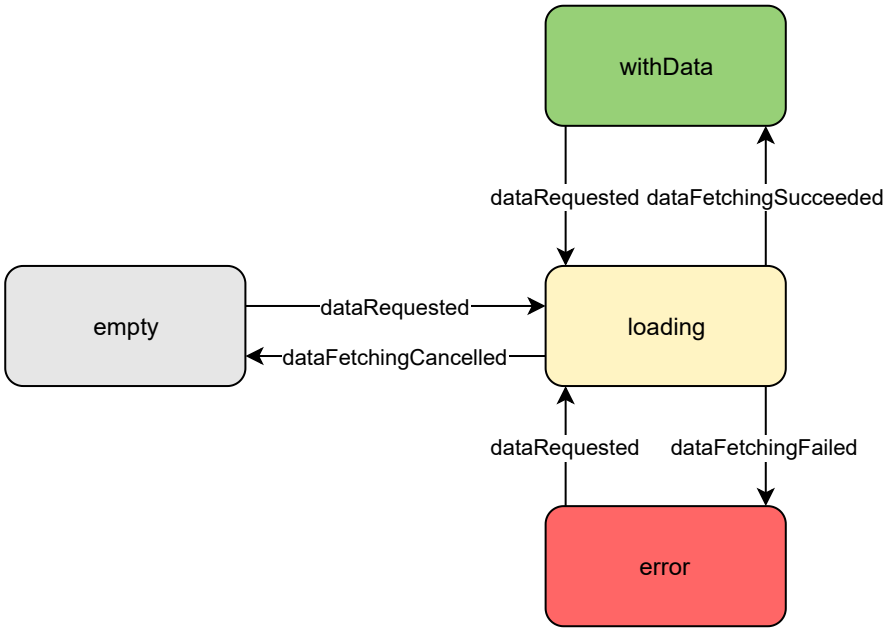
multiple transitions.

Example: Data fetching workflow

One very common task in web applications is loading data from the backend. Let's build a state machine that represents this workflow. The solution will be based on Redux architecture.

Let's say that the user can request data to be fetched by clicking on a button. It should also be possible to cancel the request. When data is being fetched, a loading indicator should be shown. Data fetching can be successful, in which case it will show the fetched data. It can also fail, in which case it will show an error.

Based on the above description, we can build the following state machine.



There are four states:

- *empty* - fetching has not started yet or has been canceled
- *loading* - fetching is in progress
- *withData* - fetching was successful
- *error* - fetching failed

Let's create a discriminated union that represents all of the possible states. Next, let's look at all possible actions that can result in state transitions. Finally, we'll build a reducer which describes all of the possible transitions.

```
type State =
    | { type: 'empty' }
    | { type: 'loading' }
    | { type: 'withData', data: string[] }
    | { type: 'error', errorMessage: string };

type Action =
    | { type: 'dataRequested' }
    | { type: 'dataFetchingSucceeded', data: string[] }
    | { type: 'dataFetchingFailed', errorMessage: string }
    | { type: 'dataFetchingCancelled' };

function reducer(prevState: State, action: Action): State {
    switch (prevState.type) {
        case 'loading':
            switch (action.type) {
                case 'dataFetchingSucceeded':
                    return { type: 'withData', data: action.data };
                case 'dataFetchingCancelled':
                    return { type: 'empty' };
                case 'dataFetchingFailed':
                    return { type: 'error', errorMessage: action.errorMessage };
                case 'dataRequested':
                    return prevState;
            }
        case 'empty':
        case 'error':
        case 'withData':
            switch (action.type) {
                case 'dataRequested':
                    return { type: 'loading' };
                case 'dataFetchingCancelled':
                case 'dataFetchingFailed':
                case 'dataFetchingSucceeded':
                    return prevState;
            }
    }
}
```

As you can see, the reducer closely matches the state machine. First, we switch over possible states. For each state, we defined what happens when a certain action is triggered. For some actions, we return `prevState` which means that this transition is not possible.

Note that we never defined the `default` case. Because of this, if we forgot to handle some state or transition, TypeScript would throw a compilation error. This pattern only works if you explicitly defined the return type of the containing function. It's extremely useful, even outside of the context of state machines.

As you can see, data fetching workflow is very clear when represented using a state machine. It's a type-safe approach that helps you break down the complexity in a very structured way.

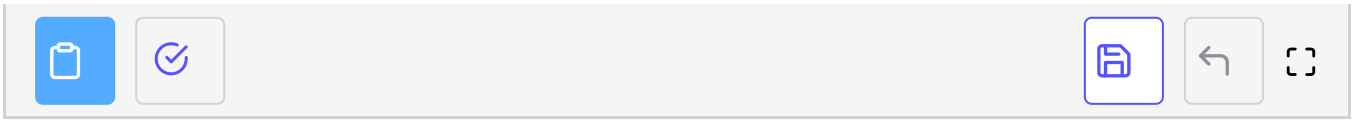
Exercise

Implement the turnstile state machine.

```
type State =
    | { type: 'empty' }
    | { type: 'loading' }
    | { type: 'dataPartiallyLoaded', data: string[] }
    | { type: 'withData', data: string[] }
    | { type: 'error', errorMessage: string };

type Action =
    | { type: 'dataRequested' }
    | { type: 'dataFetchingInProgress', data: string[] }
    | { type: 'dataFetchingSucceeded', data: string[] }
    | { type: 'dataFetchingFailed', errorMessage: string }
    | { type: 'dataFetchingCancelled' };

function reducer(prevState: State, action: Action): State {
  return prevState;
}
```



The final lesson in this chapter compares discriminated unions with an equivalent OOP technique called subtyping.