State machine pattern in FE development

by Eugenia Zigisova

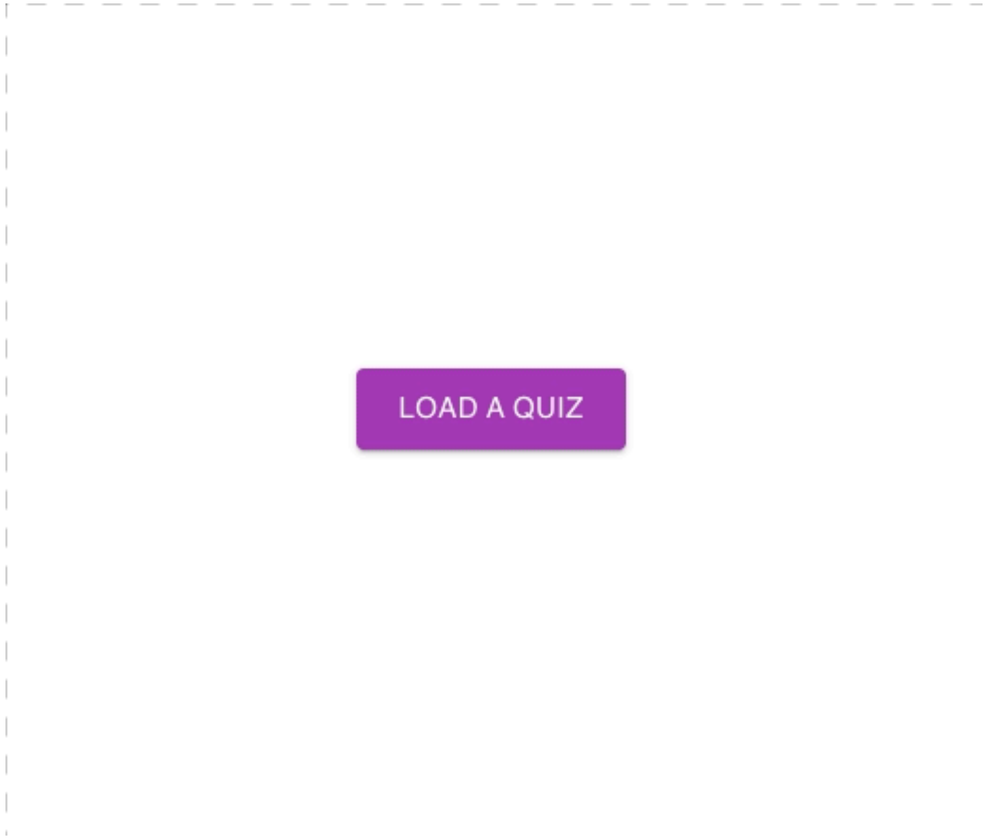
@jevgeniazi

Eugenia Zigisova

Senior Frontend Engineer
at Berlin-based startup Gorillas

Former Frontend Engineer at N26
mobile bank

@jevgeniazi


LOAD A QUIZ

How to implement it?

```
let state = {
    isLoading: false,




};
```

```
let state = {
    isLoading: false,
    question: null,



};
```

```
let state = {
    isLoading: false,
    question: null,
    isError: false,



};
```

```
let state = {
    isLoading: false,
    question: null,
    isError: false,
    isLoading: false,


};
```

```
let state = {
    isLoading: false,
    question: null,
    isError: false,
    isLoading: false,


};
```

```
let state = {
    ~~isLoading: false,~~
    question: null,
    isError: false,
    isQuestionLoading: false,
    isResultLoading: false,

};
```

```
let state = {
    ~~isLoading: false,~~
    question: null,
    isError: false,
    isQuestionLoading: false,
    isResultLoading: false,
    result: null,
};
```

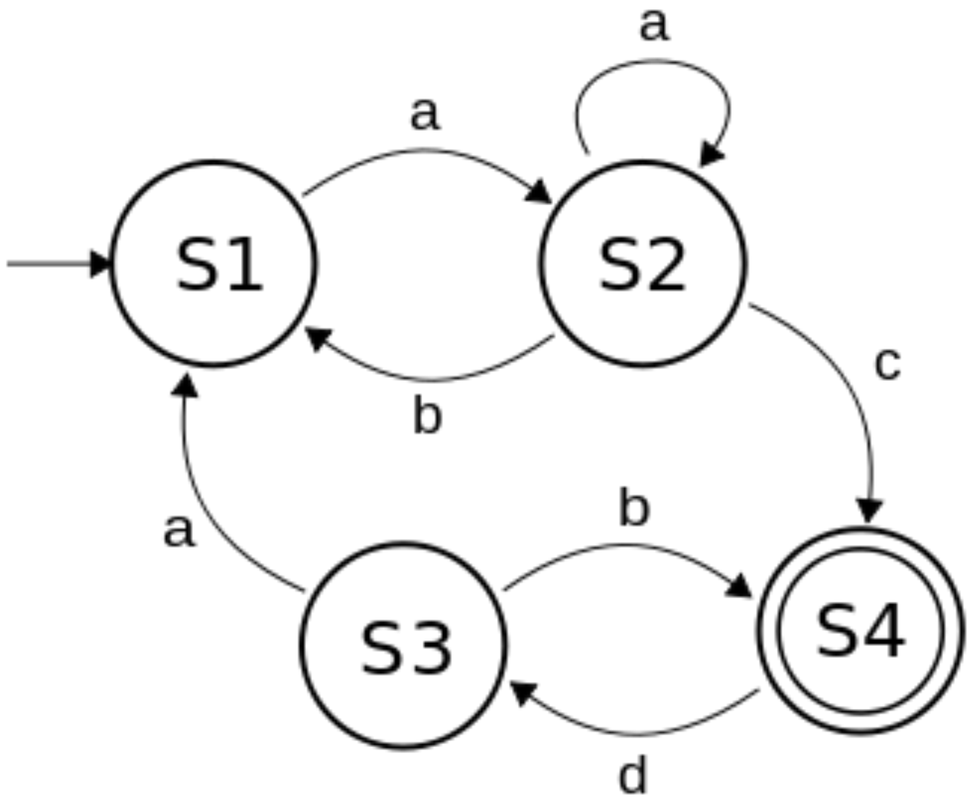
```
const handleClick = async () => {
    state.isQuestionLoading = true;

    try {
      state.question = await fetchQuestion();
    } catch {
      state.isError = true;
    } finally {
      state.isQuestionLoading = false;
    }
};
```

```
const handleSubmit = async (answer) => {
    state.isResultLoading = true
    state.question = null

    try {
      state.result = await checkResult(answer);
    } catch {
      state.isError = true;
    } finally {
      state.isResultLoading = false;
    }
};
```

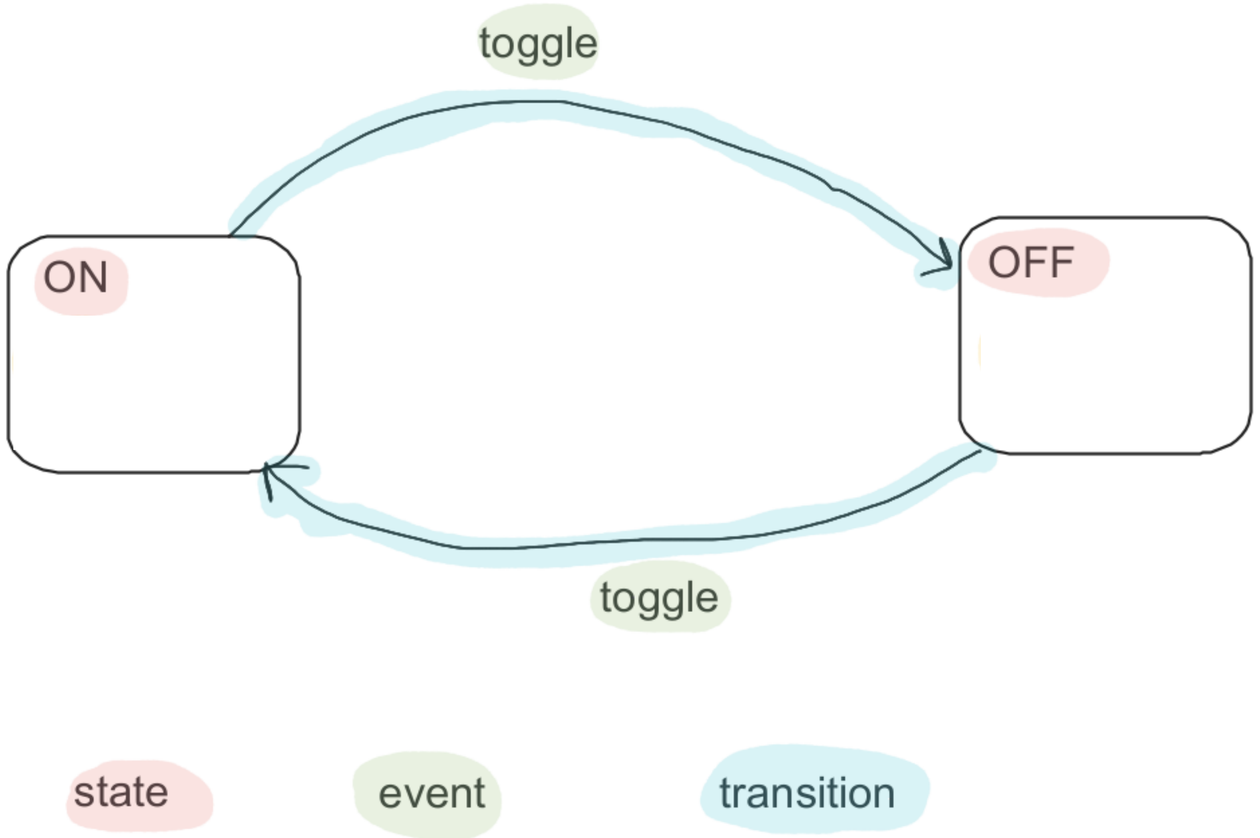
```
{(isQuestionLoading || !question ||
!isResultLoading || result) && (
    <Button loading={isQuestionLoading}>Load a quiz</Button>
)}

{isError && <Error />}

{(question || isResultLoading) && (
    <Form>
    <Button loading={isResultLoading}>Get results</Button>
    </Form>
)}

{result && <Result />}
```

Confusing. Buggy.
But it works!

Finite State Machine




ON, OFF. ON, OFF.

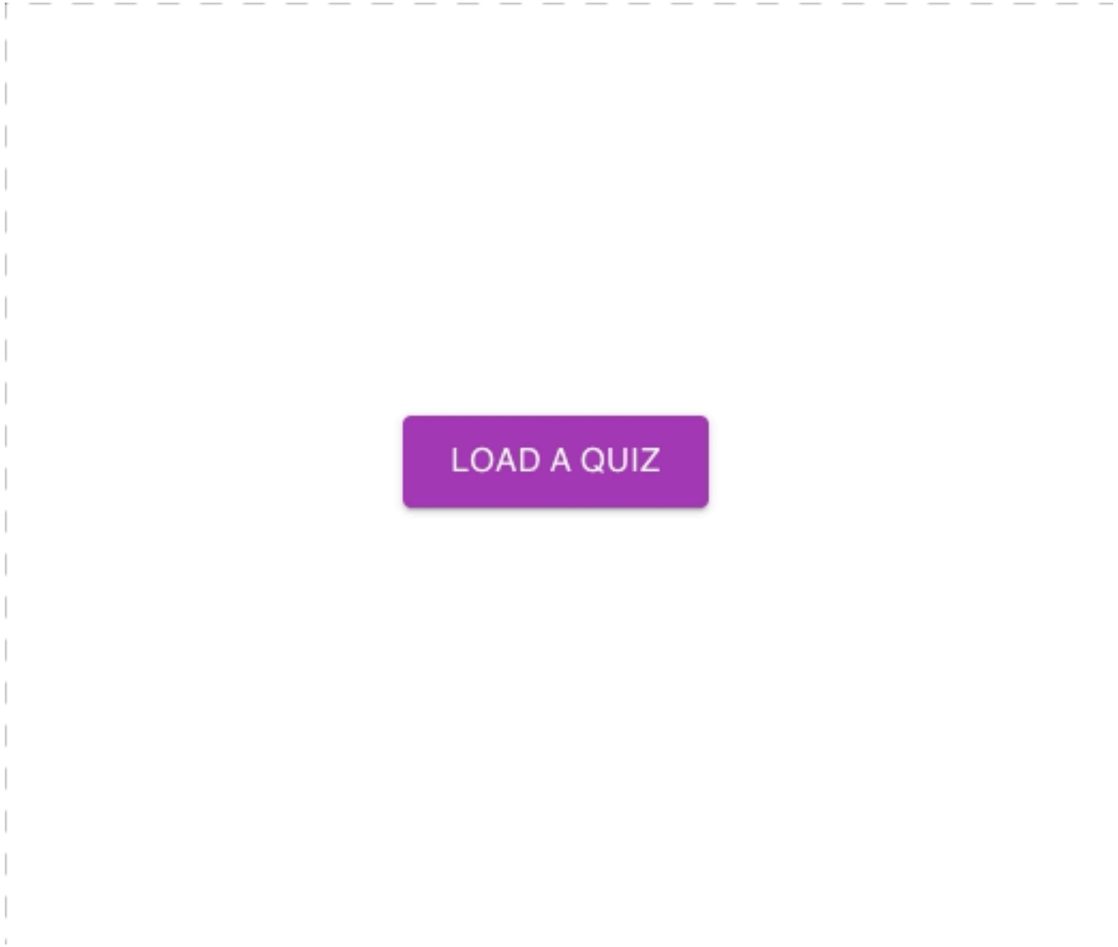

toggle
ON
OFF
toggle
state
event
transition

How many states?

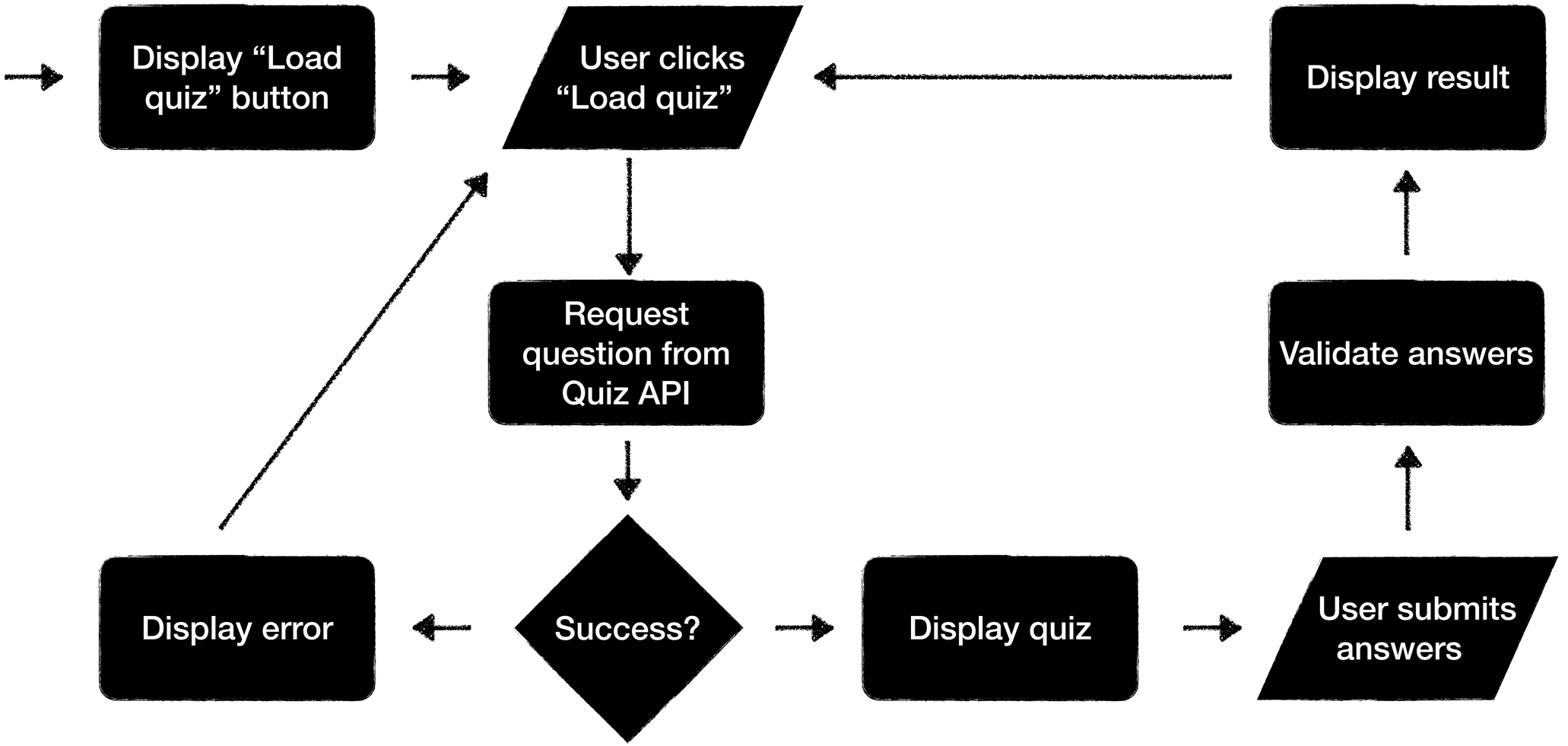
A. 3

B. 5

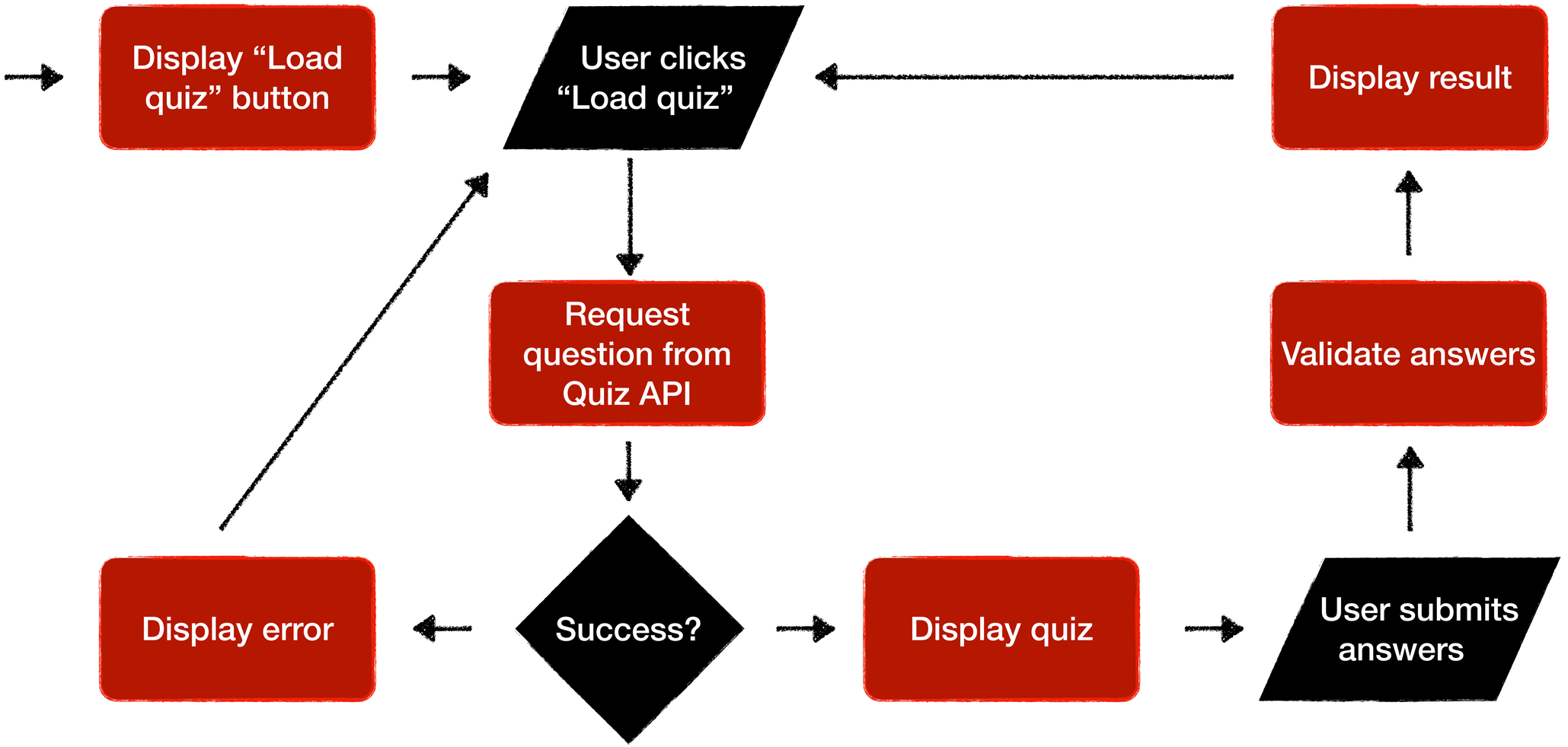
C. 6

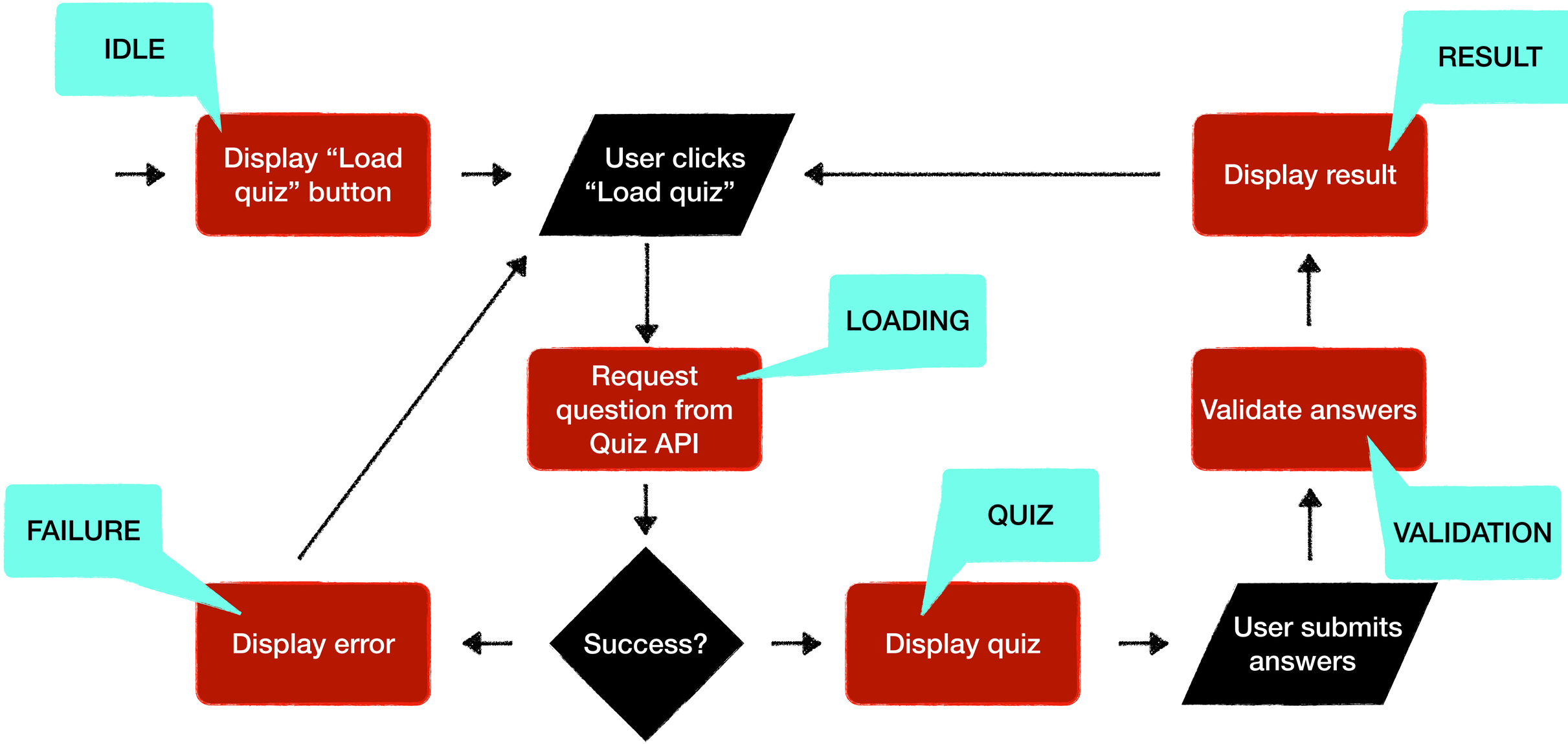


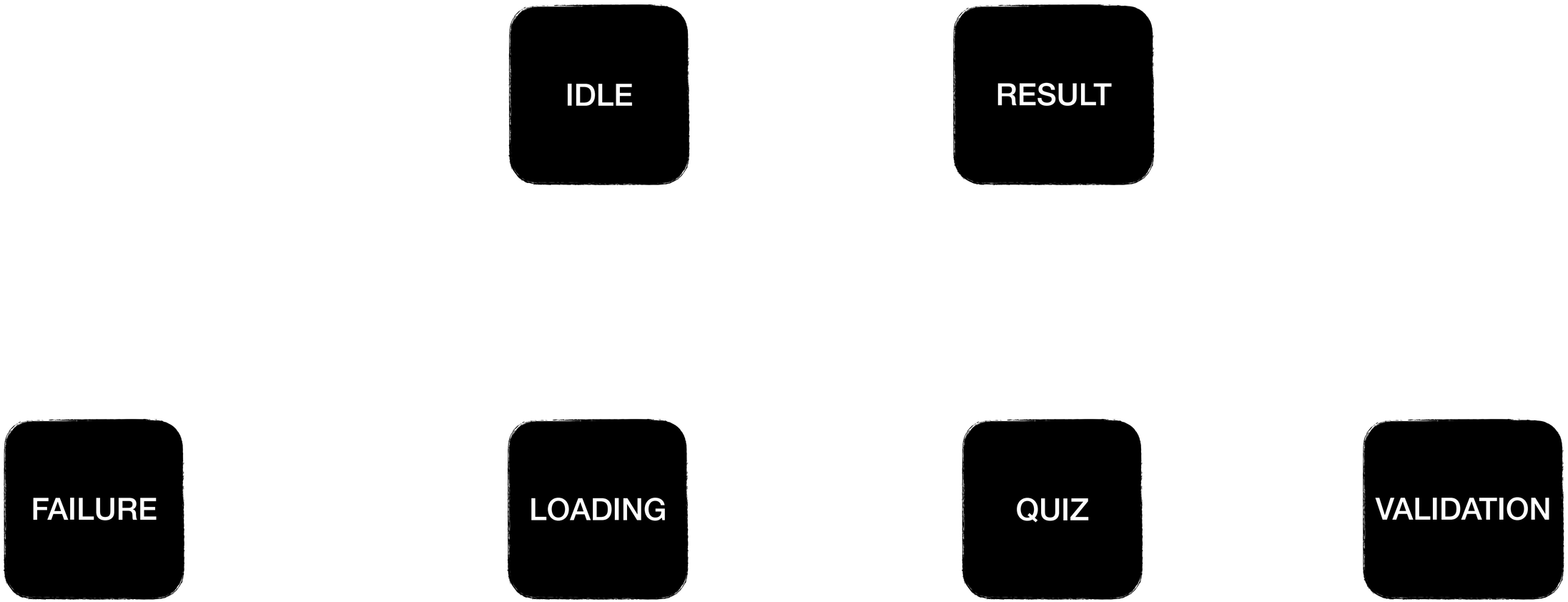
Defining states for a FE app

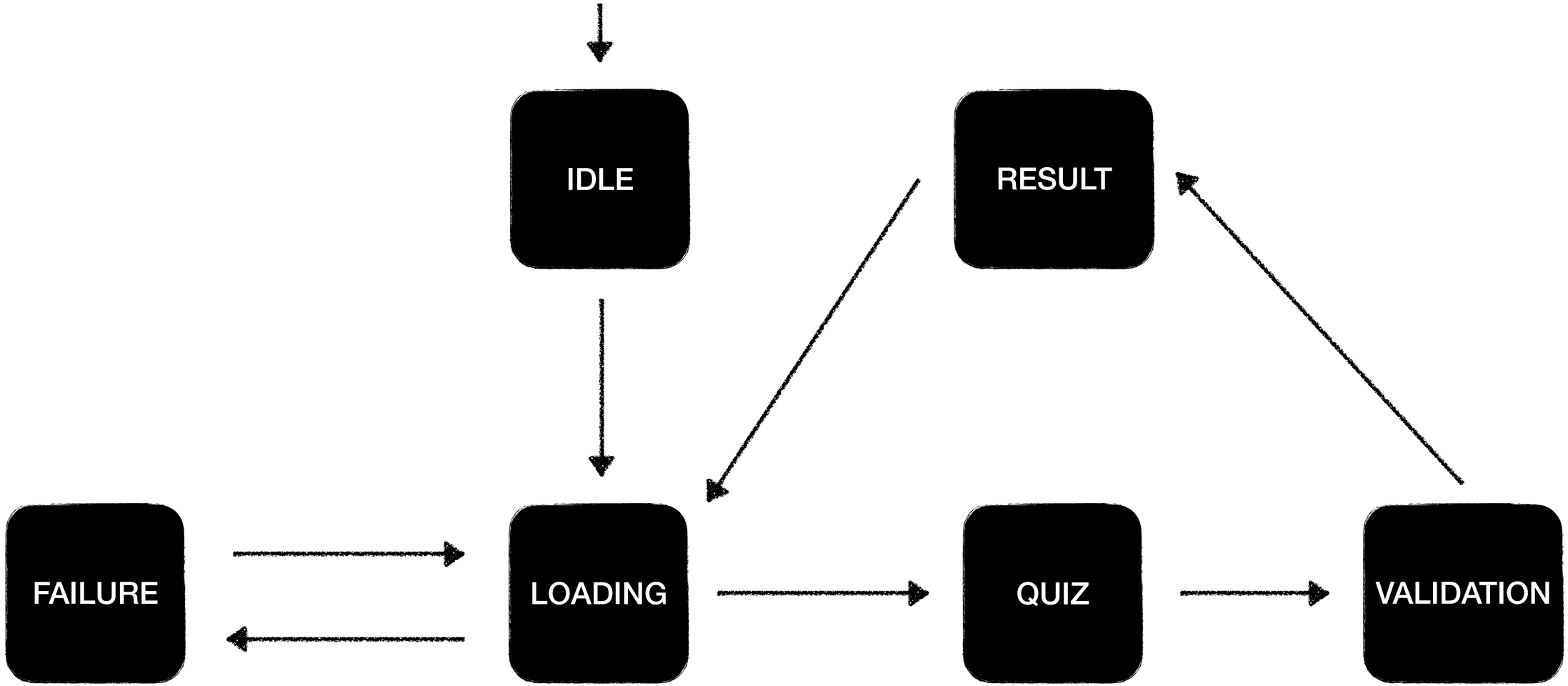

Display “Load quiz” button
User clicks “Load quiz”
Display result
Request question from Quiz API
Validate answers
Display error
Success?
Display quiz
User submits answers

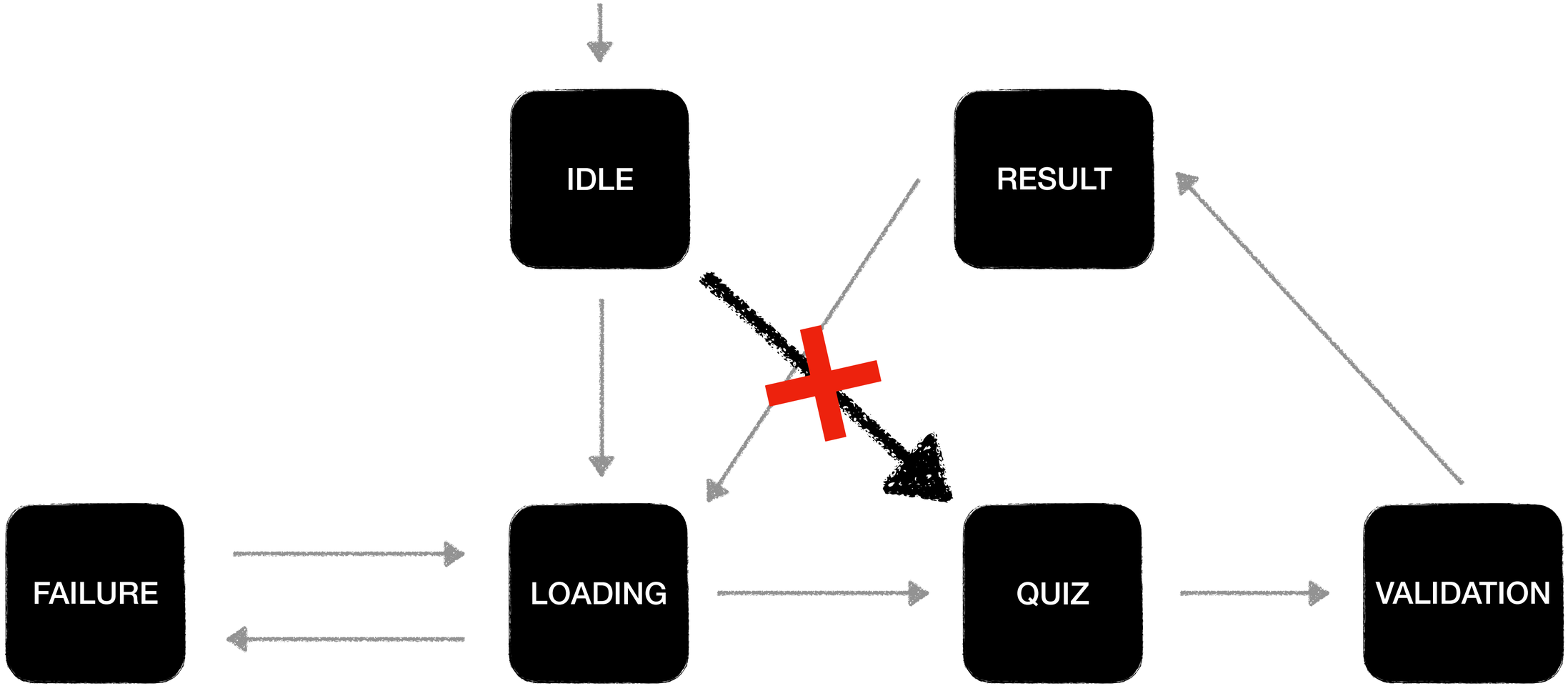
State describes a particular behaviour of a system

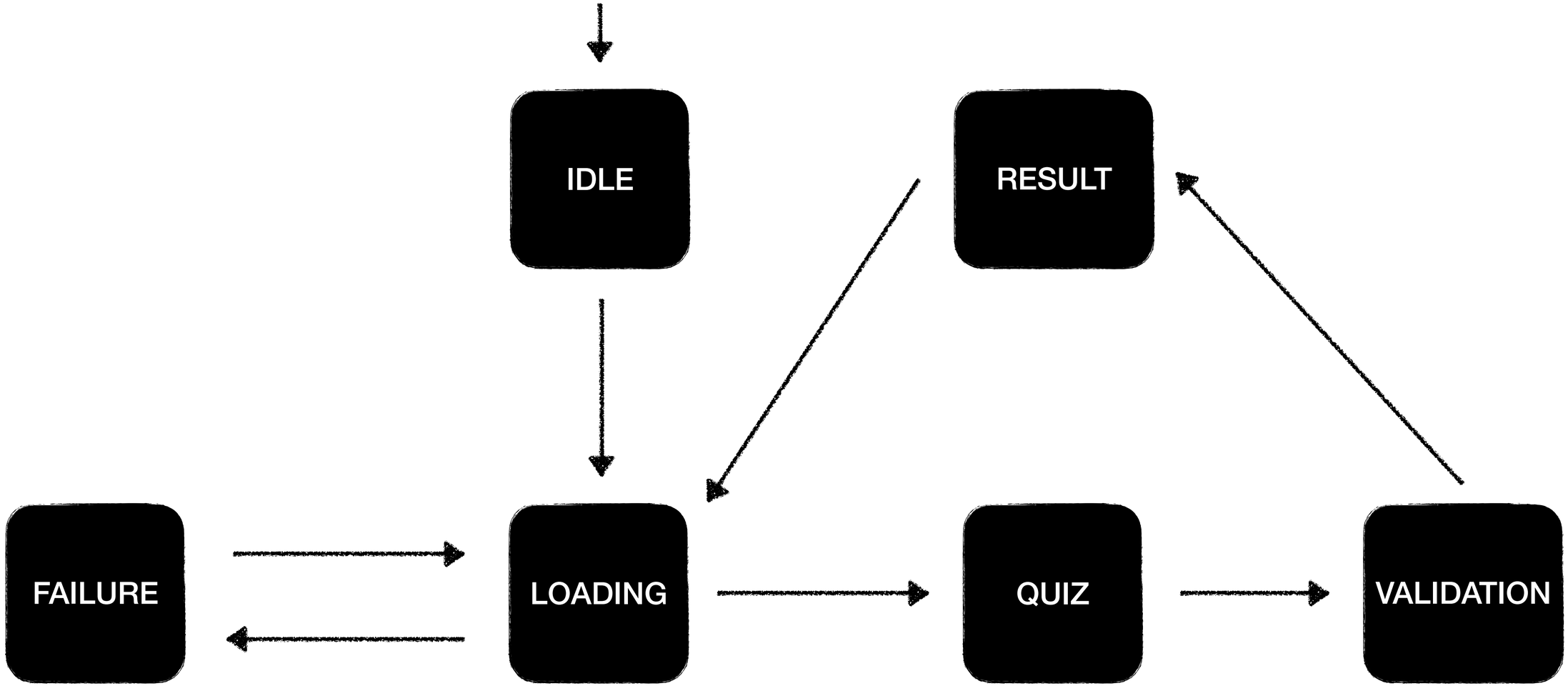

Display “Load quiz” button
User clicks “Load quiz”
Display result
Request question from Quiz API
Validate answers
Display error
Success?
Display quiz
User submits answers

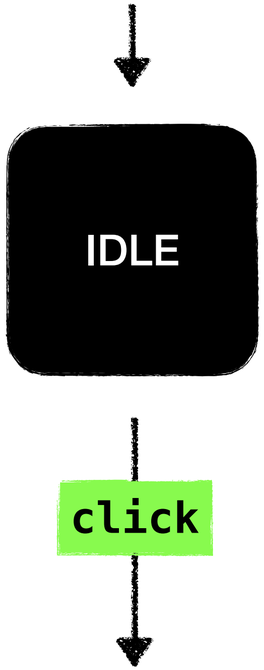

IDLE
Display “Load quiz” button
User clicks “Load quiz”
Display result
RESULT
LOADING
Request question from Quiz API
Validate answers
FAILURE
QUIZ
VALIDATION
Display error
Success?
Display quiz
User submits answers


IDLE
RESULT
FAILURE
LOADING
QUIZ
VALIDATION

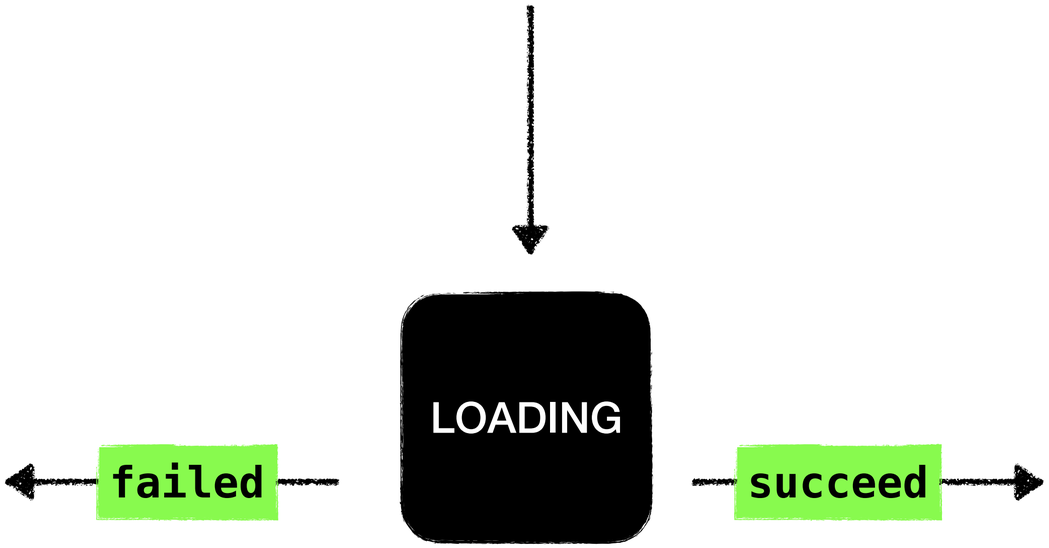

IDLE
RESULT
FAILURE
LOADING
QUIZ
VALIDATION

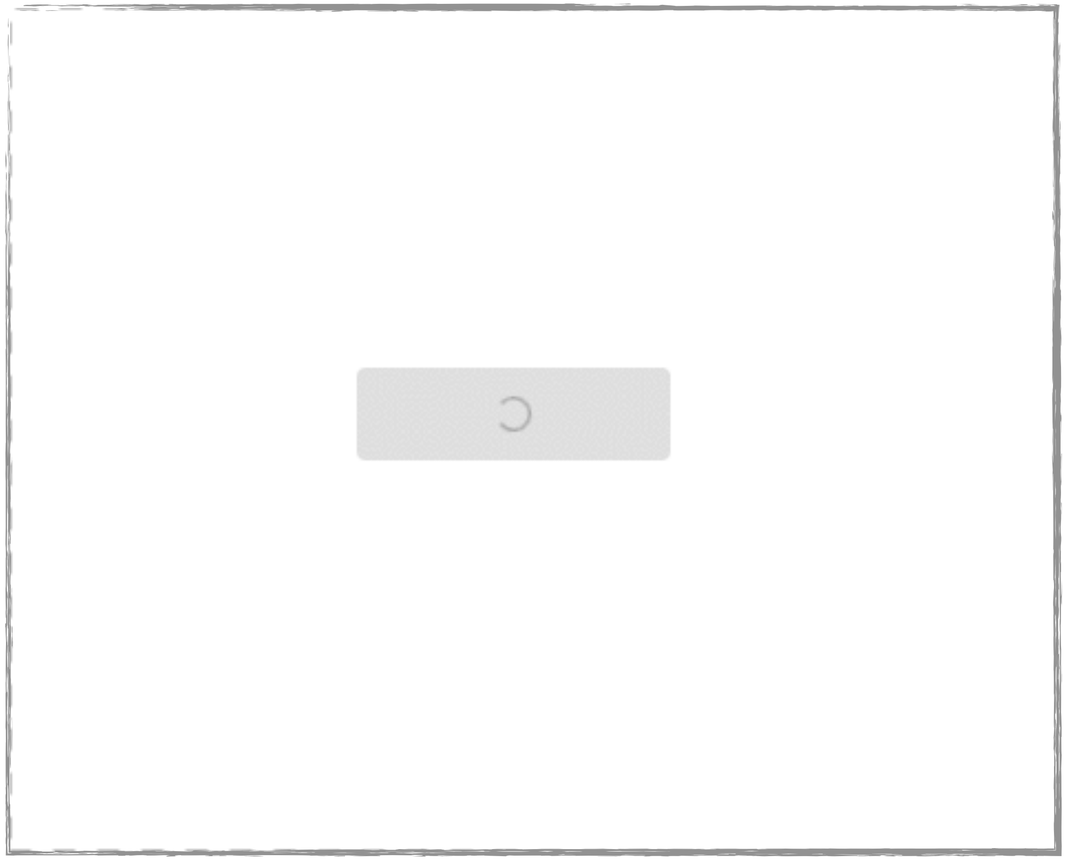

IDLE
RESULT
FAILURE
LOADING
QUIZ
VALIDATION

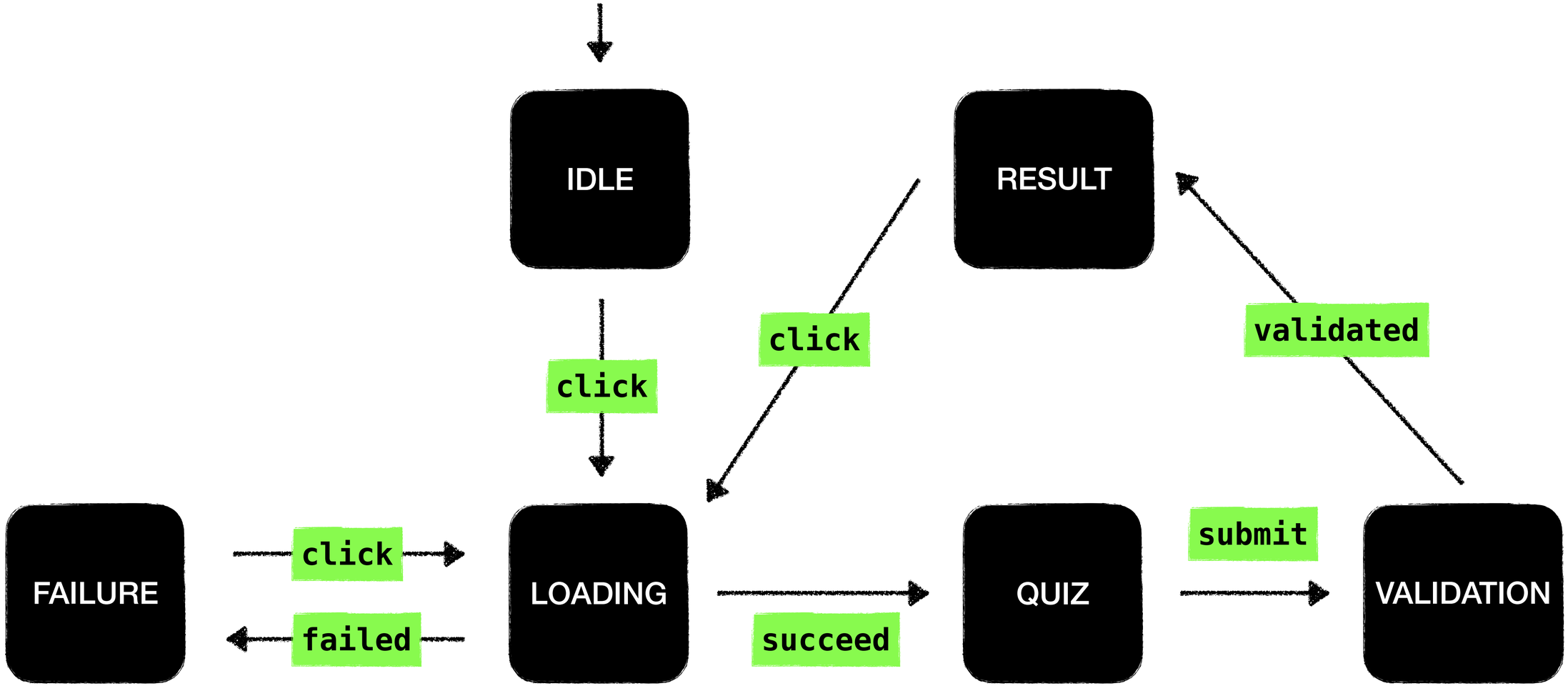

IDLE
RESULT
FAILURE
LOADING
QUIZ
VALIDATION


IDLE
click


LOAD A QUIZ


LOADING
failed
succeed




IDLE
RESULT
click
click
validated
FAILURE
click
failed
LOADING
succeed
QUIZ
submit
VALIDATION

```
const STATES = {
  IDLE: "IDLE",
  LOADING: "LOADING",
  FAILURE: "FAILURE",
  QUIZ: "QUIZ",
  VALIDATION: "VALIDATION",
  RESULT: "RESULT",
};
```

```
const EVENTS = {
  click: "click",
  succeed: "succeed",
  failed: "failed",
  submit: "submit",
  validated: "validated",
};
```

```
let state = STATES.IDLE;

// ...

<Button
  onClick={() => {
    // Inform machine about “click” event
    // Get new state value from machine
  }}
>
  Load a quiz
</Button>
```

```
let state = STATES.IDLE;

// ...

<Button
  onClick={() => {
    state = transition(state, EVENTS.click);
  }}
>
  Load a quiz
</Button>
```

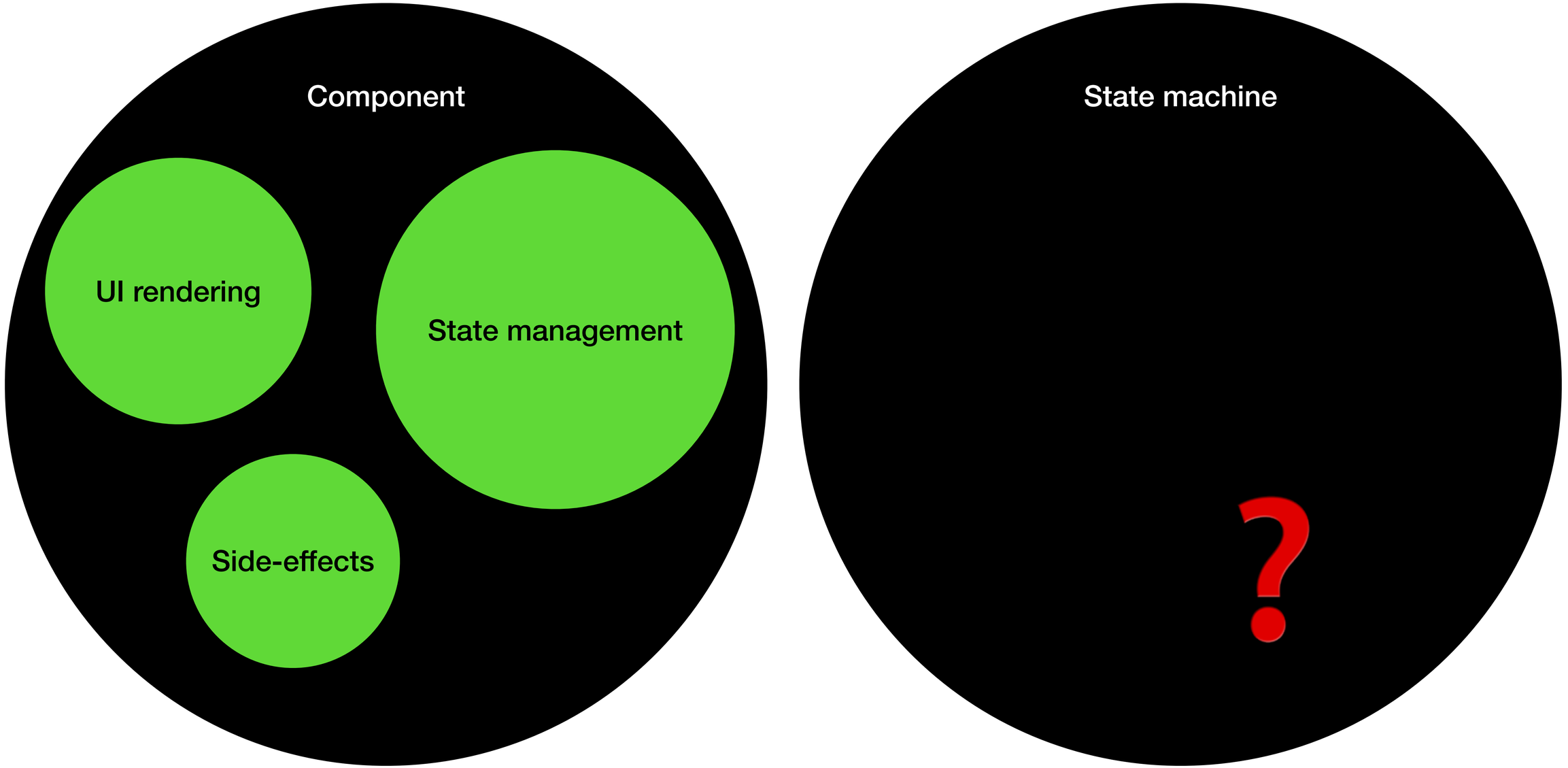
```
function transition(currentState, event) {
  switch (currentState) {
    case STATES.IDLE: { // … }
    case STATES.LOADING: { // … }
    case STATES.FAILURE: { // … }
    case STATES.QUIZ: { // … }
    case STATES.VALIDATION: { // … }
    case STATES.RESULT: { // … }

    default: {
      return currentState;
    }
  }
}
```

```
function transition(currentState, event) {
  switch (currentState) {
    case STATES.IDLE: {
      switch (event) {
        case EVENTS.click: {
          return STATES.LOADING;
        }
        default: {
          return currentState;
        }
      }
    }
  }
}
```

```
let state = {
  isError: false,
  isQuestionLoading: false,
  isResultLoading: false,
  isResultCorrect: false,
};
```

```
let state = "IDLE"
```

```
{(state === STATES.QUIZ || state === STATES.VALIDATION) && (
  <Form>
    <Button loading={state === STATES.VALIDATION} />
  </Form>
)}
```

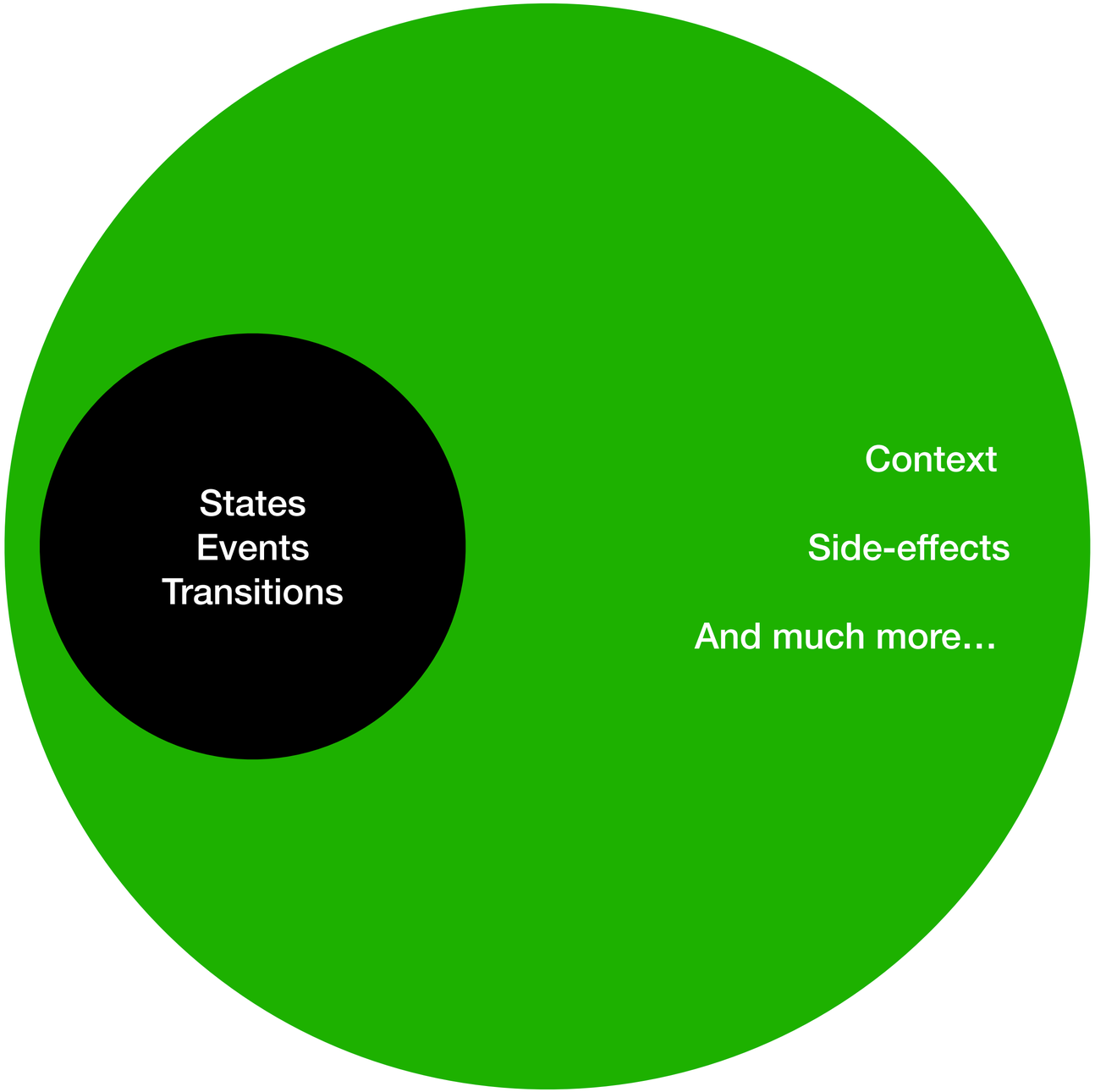

Component
UI rendering
State management
Side-effects
State machine
?

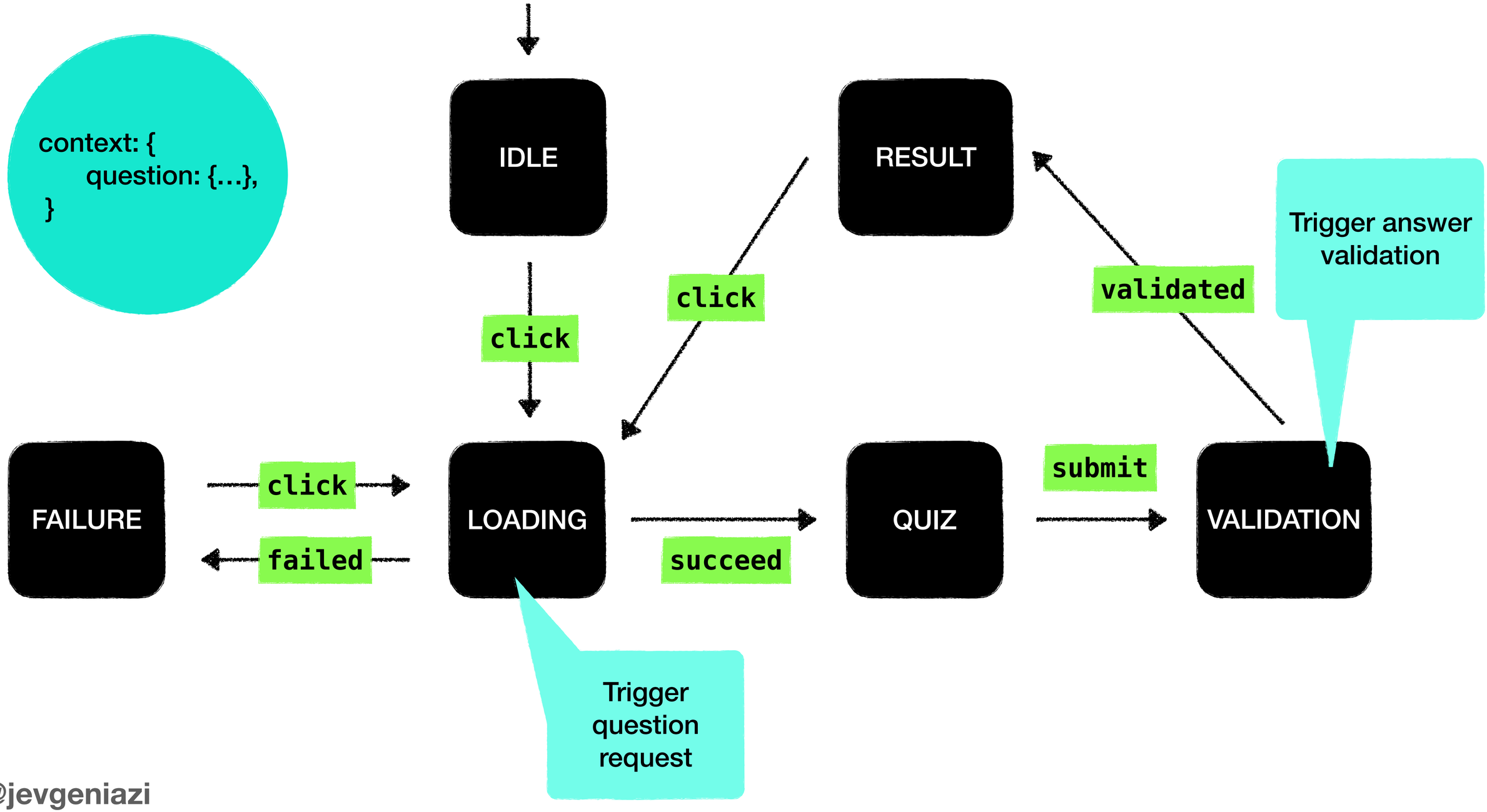
xstate.js.org

```
const definition = {
  initial: "IDLE",
  context: {
    question: null,
    result: undefined,
  },
  states: {
    IDLE: {
      on: {
        click: { target: "LOADING" },
      },
    },
    LOADING: {
      invoke: {
        id: "fetchQuestion",
        src: () => fetchQuestion(),
        onDone: {
          target: "SUCCESS",
          actions: assign({ question: (c, e) => e.data }),
        },
        onError: {
          target: "FAILURE",
        },
      },
    },
    SUCCESS: {
      on: {
        submit: "VALIDATION",
      },
    },
    FAILURE: {
      on: {
        click: "LOADING",
      },
```

```
const machine = {
  initial: "IDLE",
  states: {
    IDLE: {},
    LOADING: {},
    QUIZ: {},
    FAILURE: {},
    VALIDATION: {},
    RESULT: {},
  },
};
```

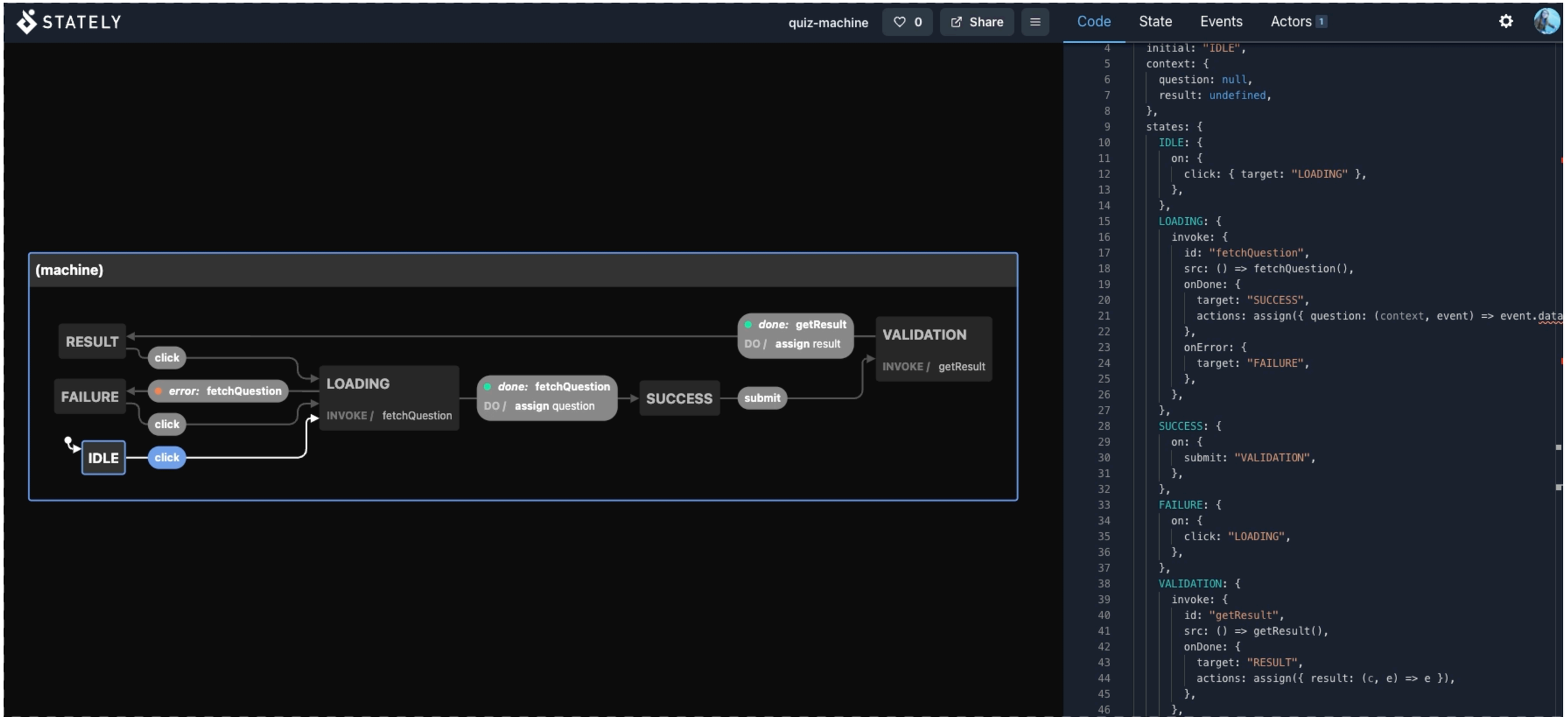
```
const machine = {
  initial: "IDLE",
  states: {
    IDLE: {
      on: {
        click: “LOADING",
      },
    },
    // ...
  },
};
```


States
Events
Transitions
Context
Side-effects
And much more…


context: {
question: {...},
}
IDLE
RESULT
click
click
validated
Trigger answer validation
FAILURE
click
failed
LOADING
succeed
QUIZ
submit
VALIDATION
Trigger question request

```
const machine = {
  context: {
    question: null,
  },
  initial: "IDLE",

  states: {
    // ...
  },
};
```

```
const machine = {
  // context: ...,
  states: {
    LOADING: {
      invoke: {
        src: () => fetchQuestion(),
        onDone: {
          target: "SUCCESS",
          actions: assign({ question: (c, e) => e.data }),
        },
        onError: {
          target: "FAILURE",
        },
    // ...
```



https://stately.ai/viz

clean and readable

scalable

diagram as code

good for event-driven UIs

learning curve

How to start?

Process of identifying states, transitions and events

1. Draw a user flow including

- user and system events
- UI views
- internal app processes

2. Group UI views and internal processes into states
3. For each state ask “What are possible events that can happen?”
4. Add transitions and events to the state machine

Links

Slides: vocal-unicorn-1c25ed.netlify.app

You don't need a library for state machines (https://dev.to/davidkpiano/you-don-t-need-a-library-for-state-machines-k7h)

State charts (https://statecharts.dev/)

xState library (https://xstate.js.org/)

Thank you!

Blog: imeugenia.medium.com

Slides: vocal-unicorn-1c25ed.netlify.app