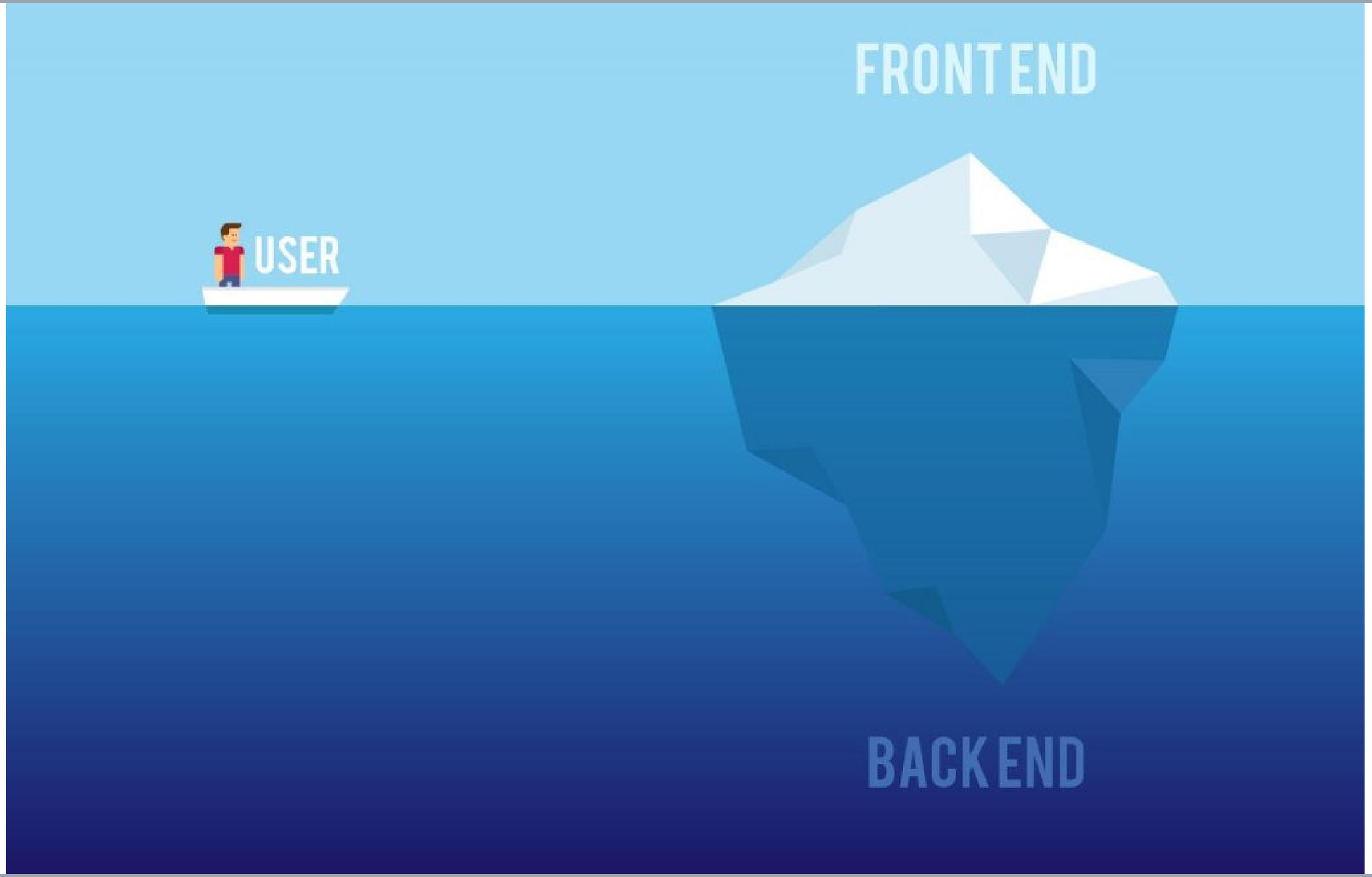
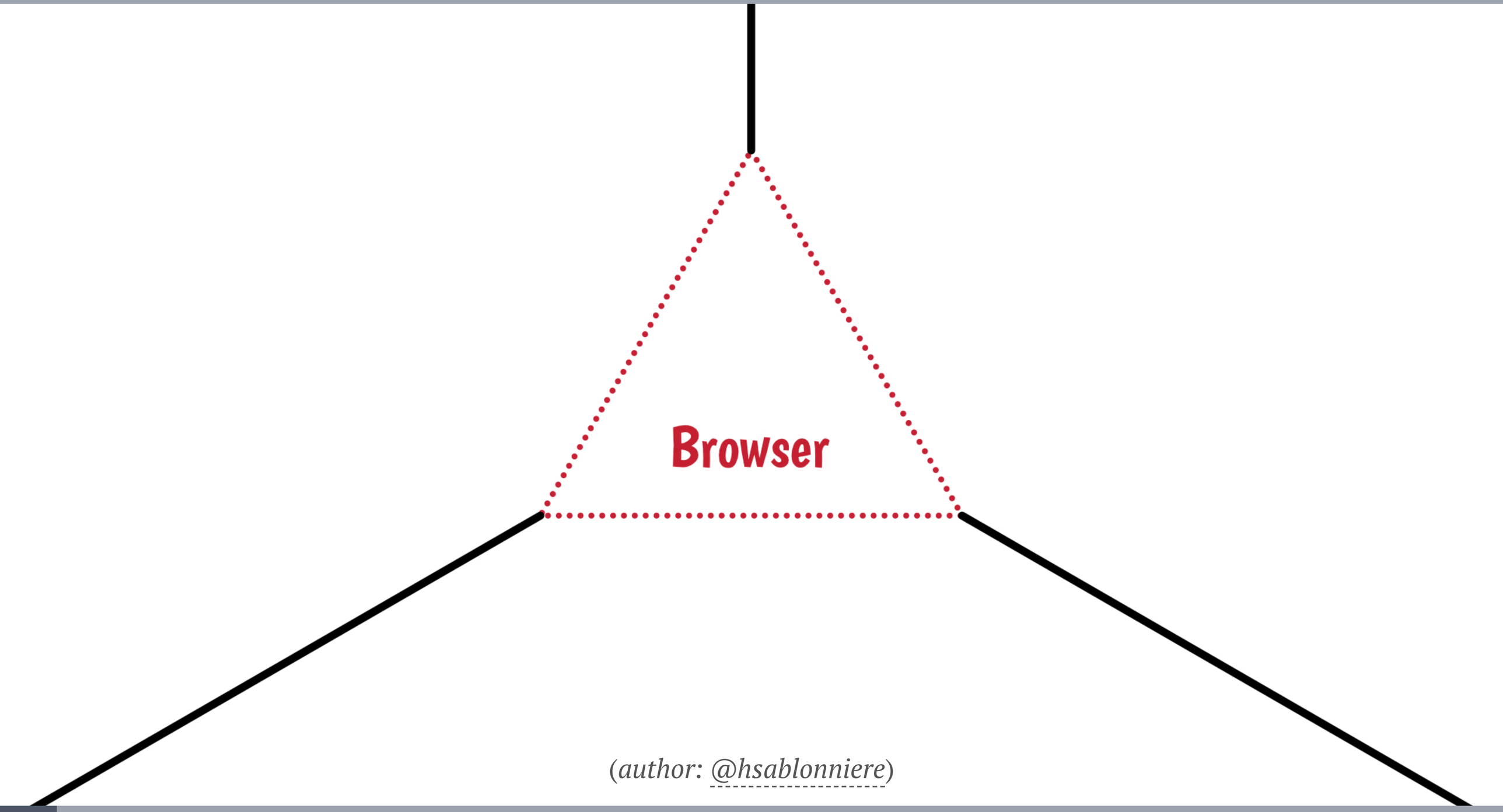
React

Introduction to React.js and Redux

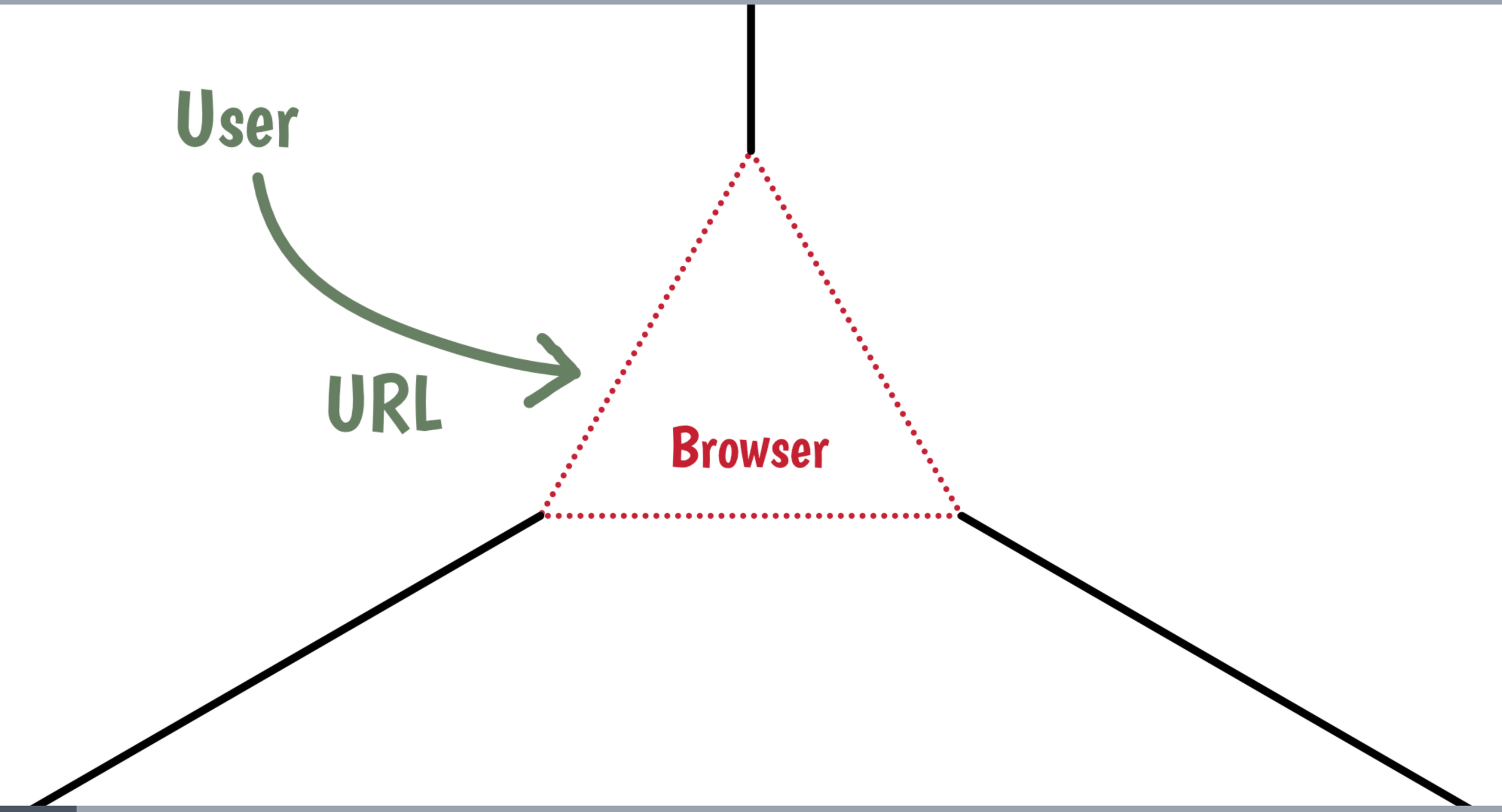
Press Space for next page →

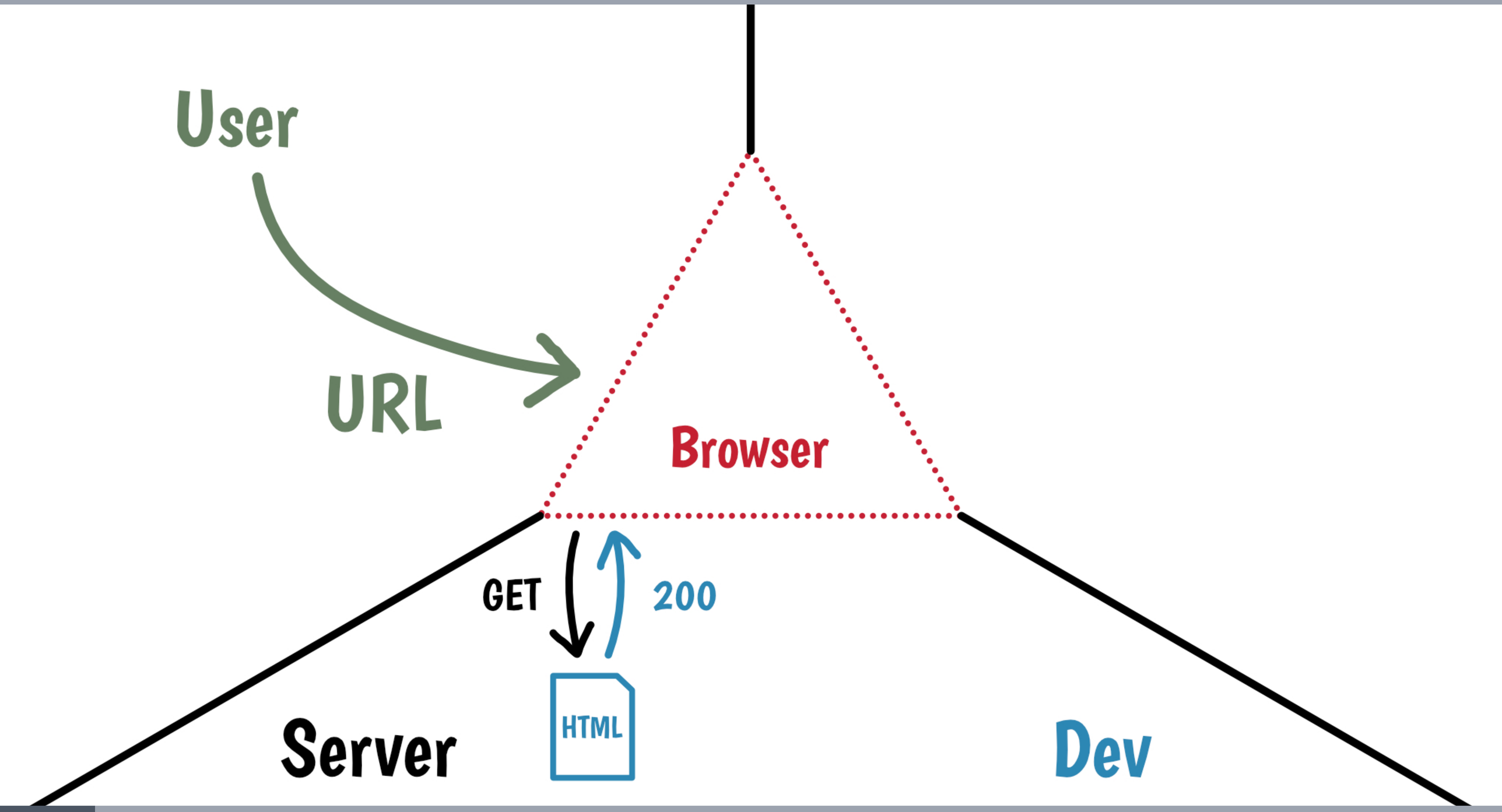

FRONTEND
USER
BACKEND

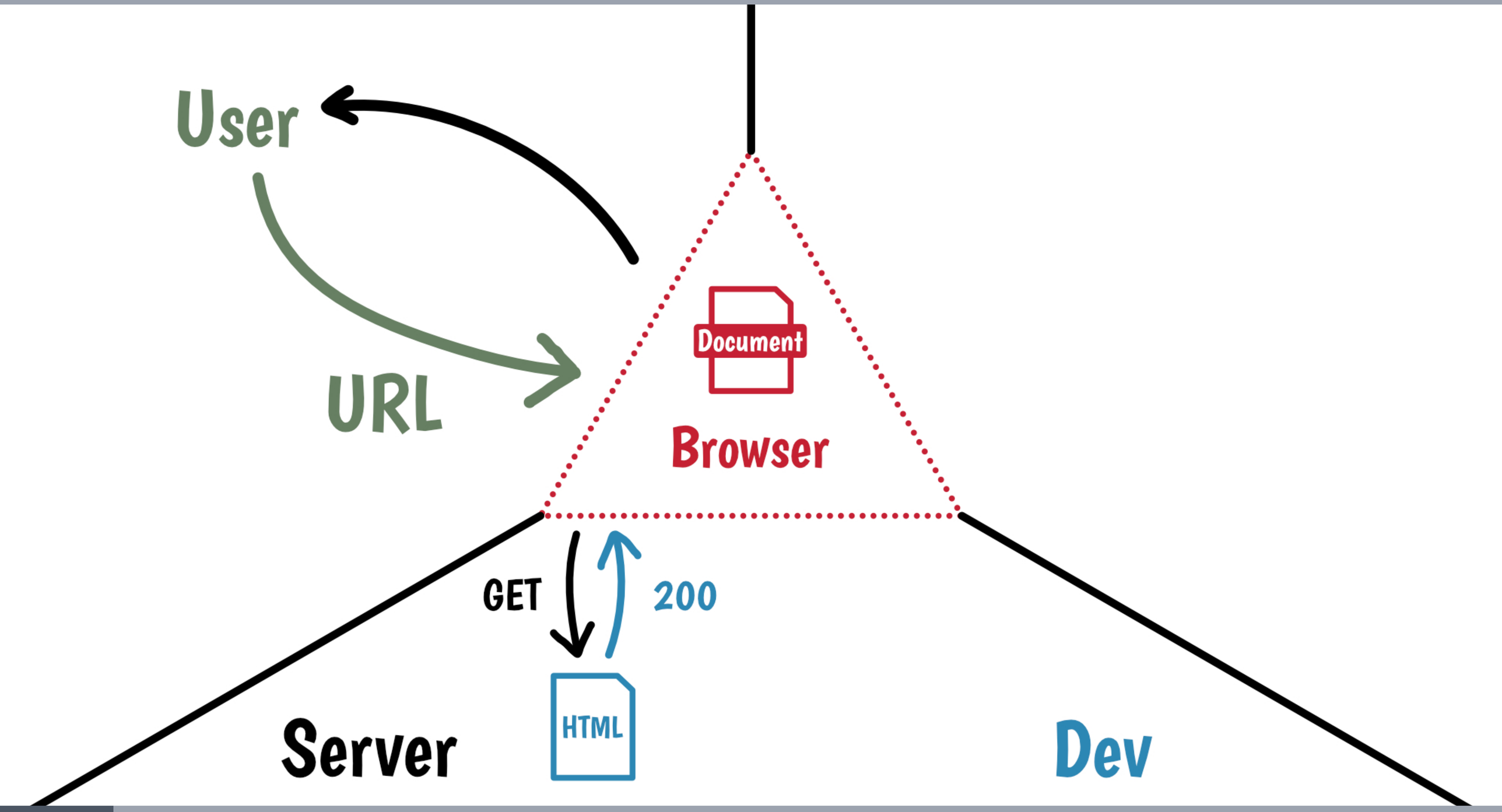
Why we speak about Front-End ?

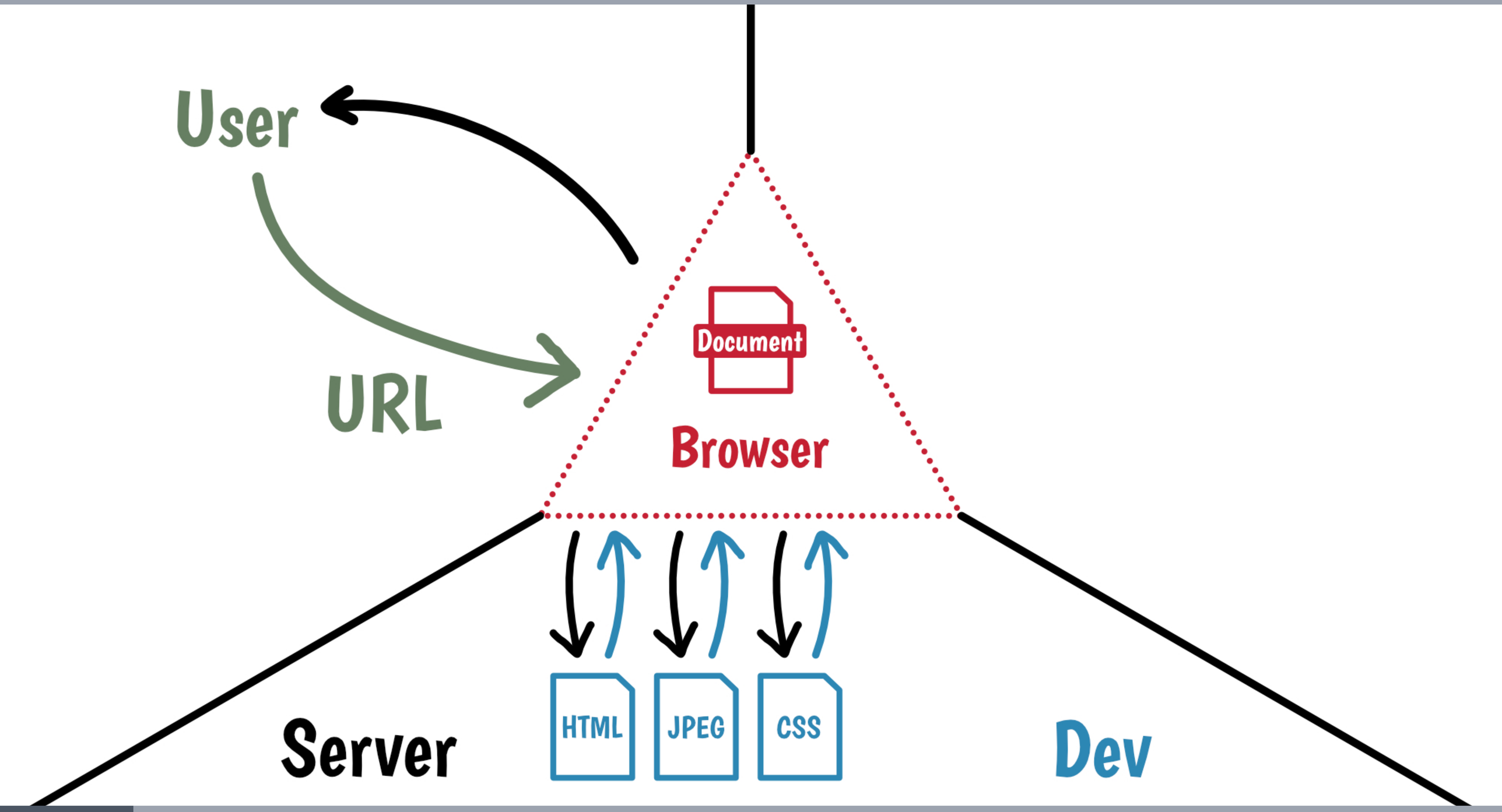


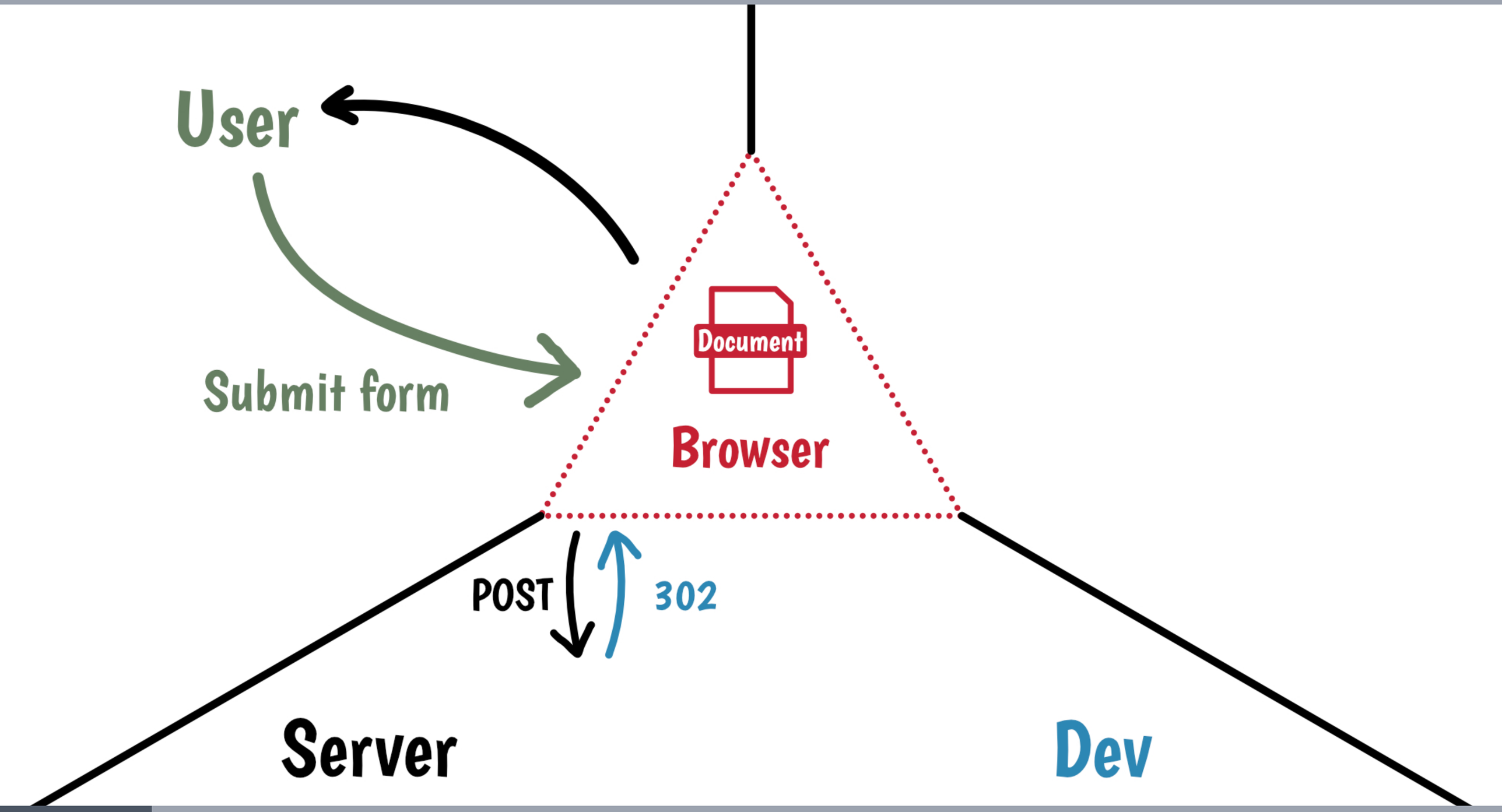
(*author: @hsablonniere*)


User
URL
Browser

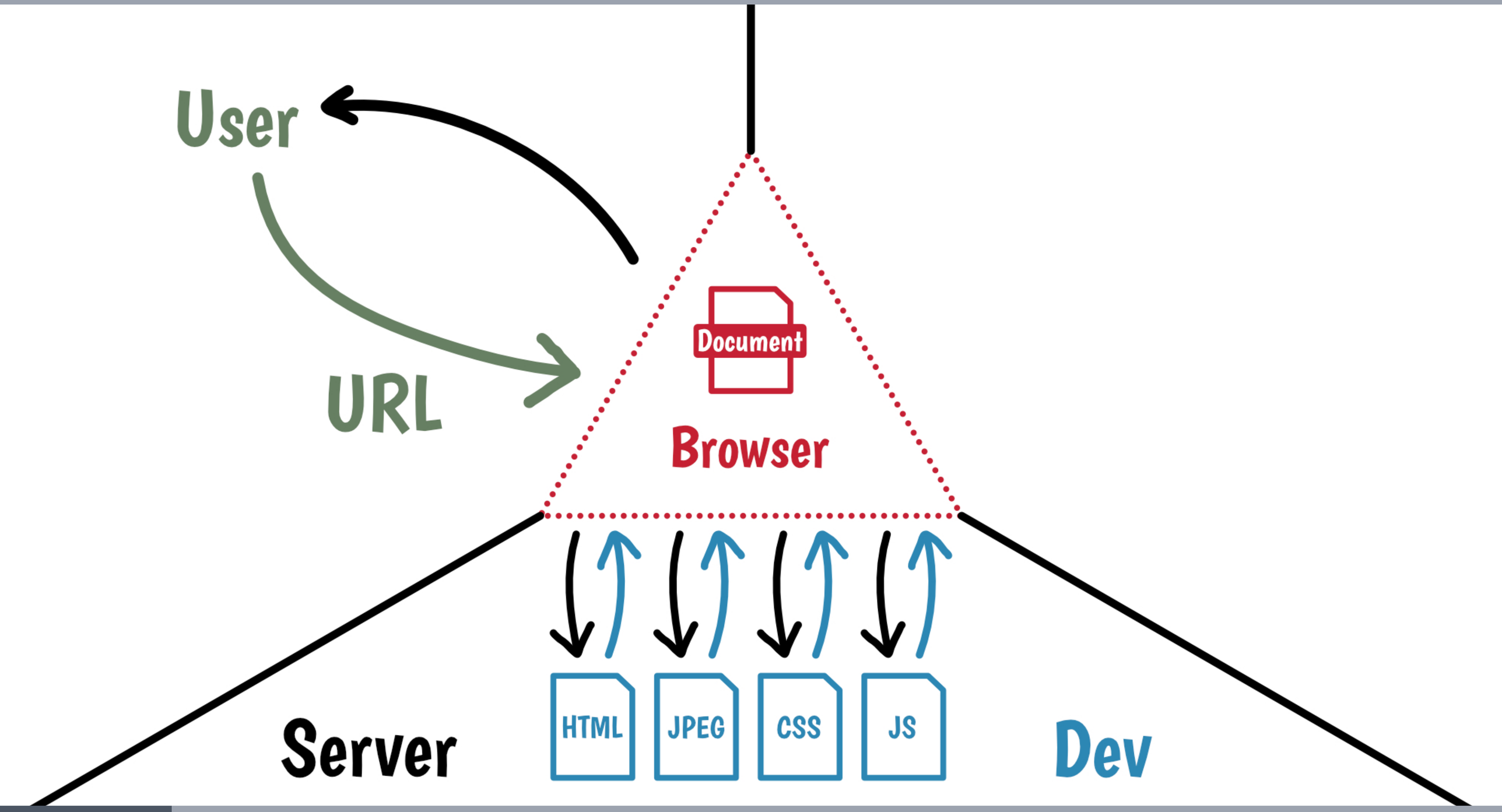

User
URL
Browser
GET
200
HTML
Server
Dev


User
URL
Document
Browser
GET
200
HTML
Server
Dev

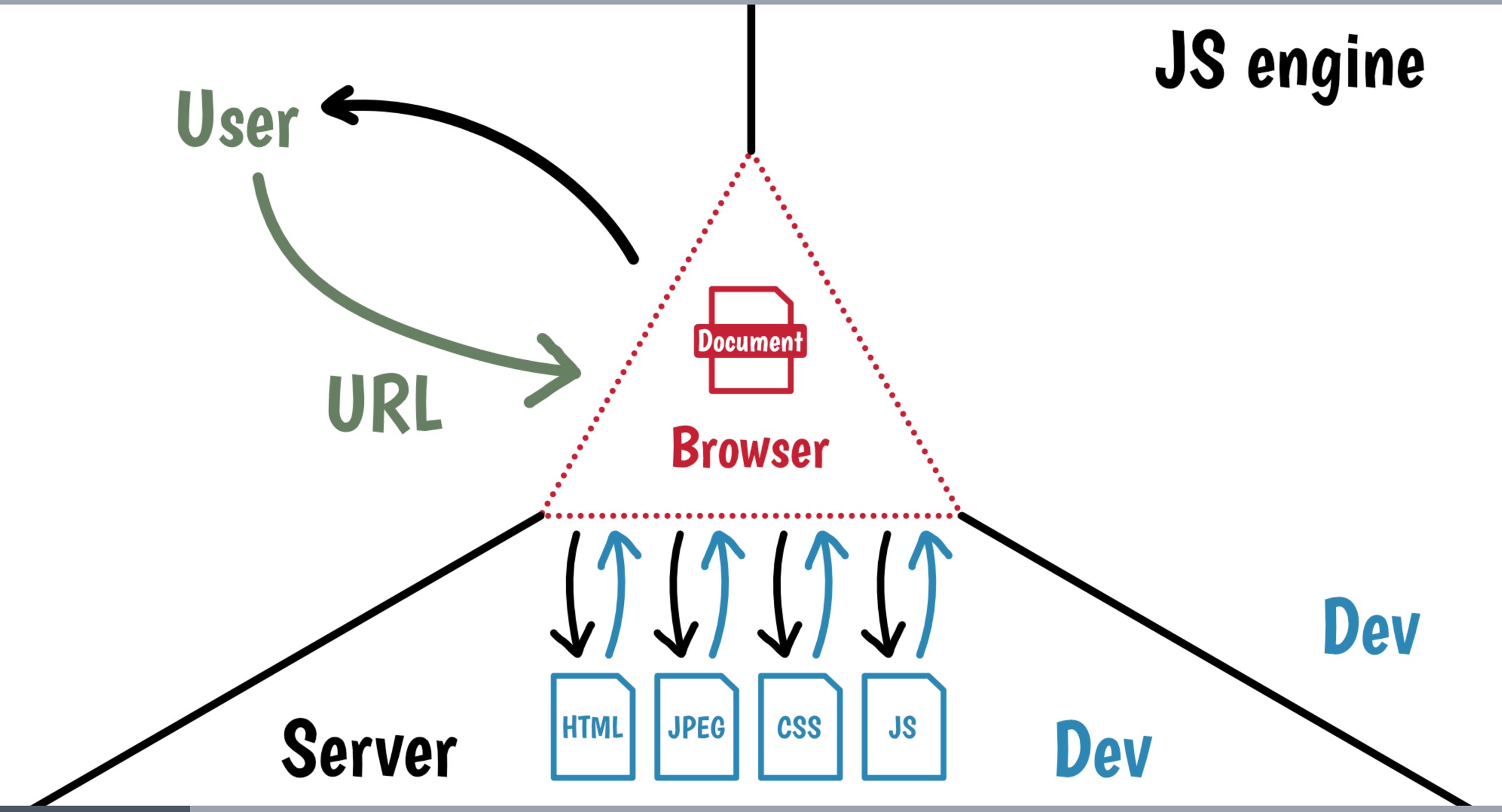

User
URL
Document
Browser
HTML
JPEG
CSS
Server
Dev

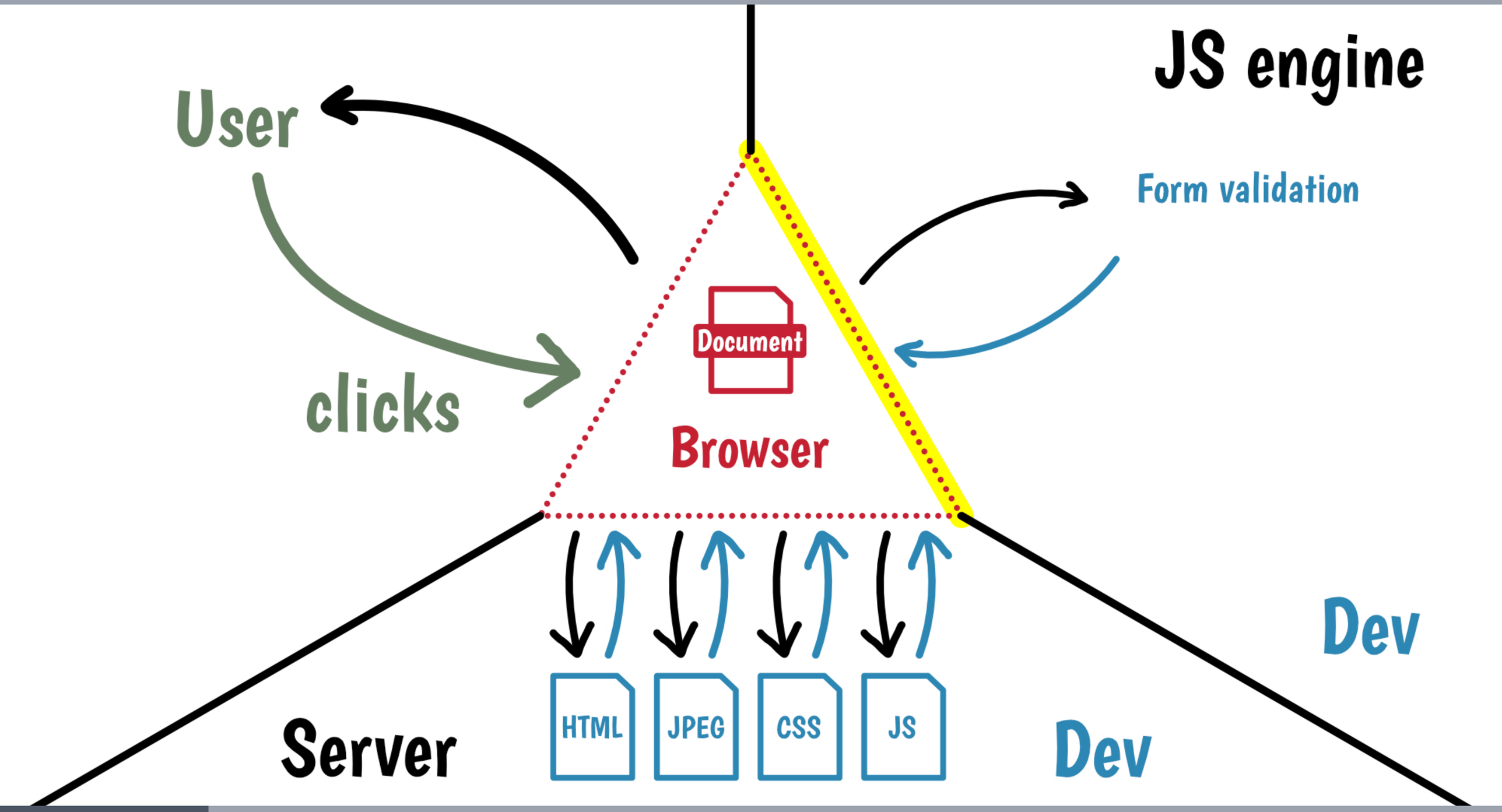

User
Submit form
Document
Browser
POST
302
Server
Dev

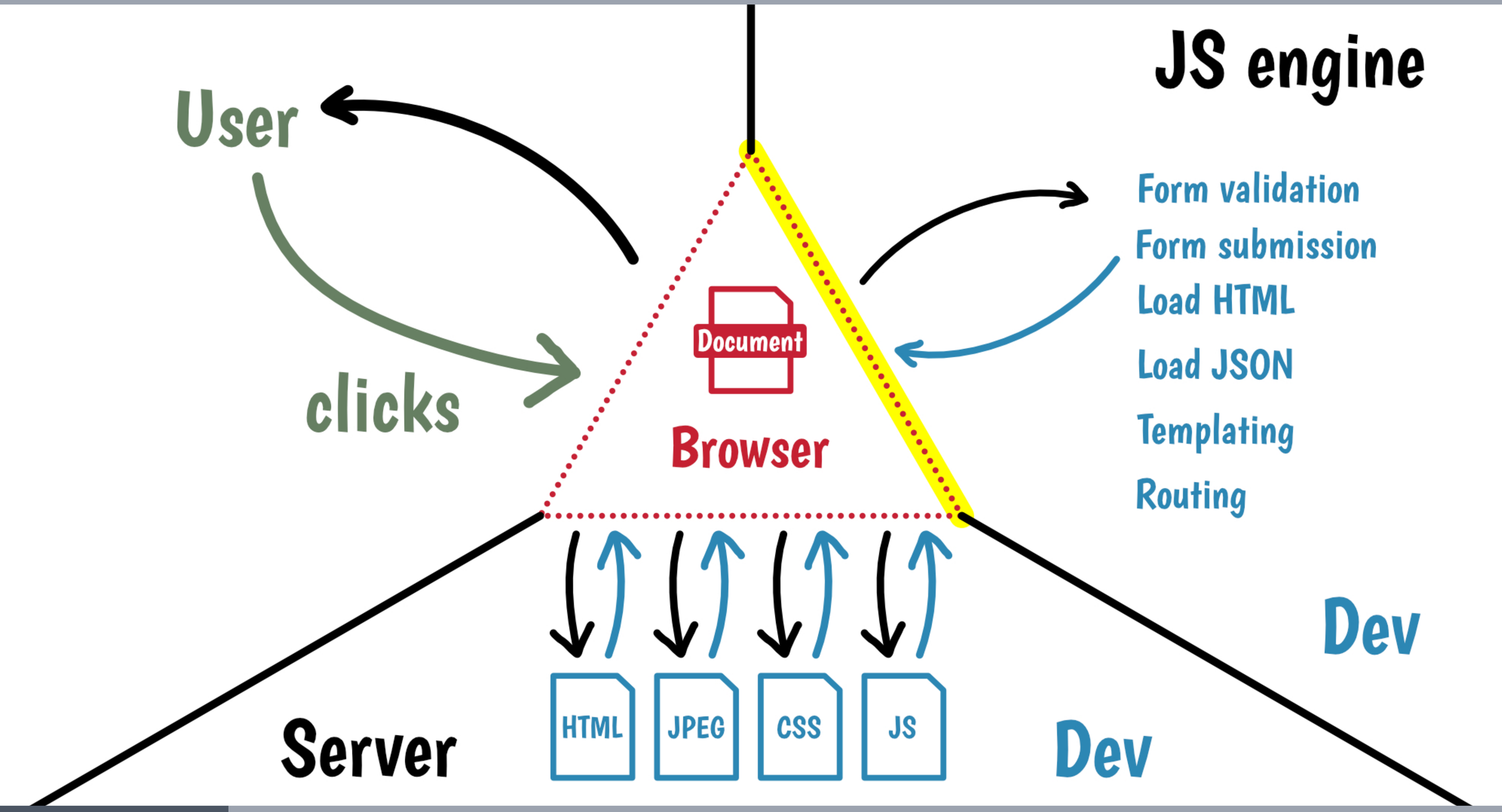
loiclegoff @llegoff loiclegoff.com



loiclegoff @llegoff loiclegoff.com




JS engine
User
clicks
Form validation
Document
Browser
Dev
Server
HTML
JPEG
CSS
JS
Dev


JS engine
User
clicks
Document
Browser
Form validation
Form submission
Load HTML
Load JSON
Templating
Routing
Dev
Server
HTML
JPEG
CSS
JS
Dev

What problem does a framework solve?

Base requirements

- ♻️ **Component system** - re-use component in different files
- 🖼️ **Templating / ViewModel** - inject values in HTML
- 🏭 **DOM manipulation** - facilitate the DOM update
- 🗺️ **Router** - allow to orient
- ⚙️ **State Management** - share a state between components

What problem does a framework solve?

Others thinks for the developer...

Improve the developer experience

- 🗃️ **Organize** - follows guidelines to produce clean code
- 🔃 **Hot-reload** - show directly your changes
- 👨‍💻 **Community** - interact with the community to find answer
- 🛠️ **Server Side Rendering** - improve SEO

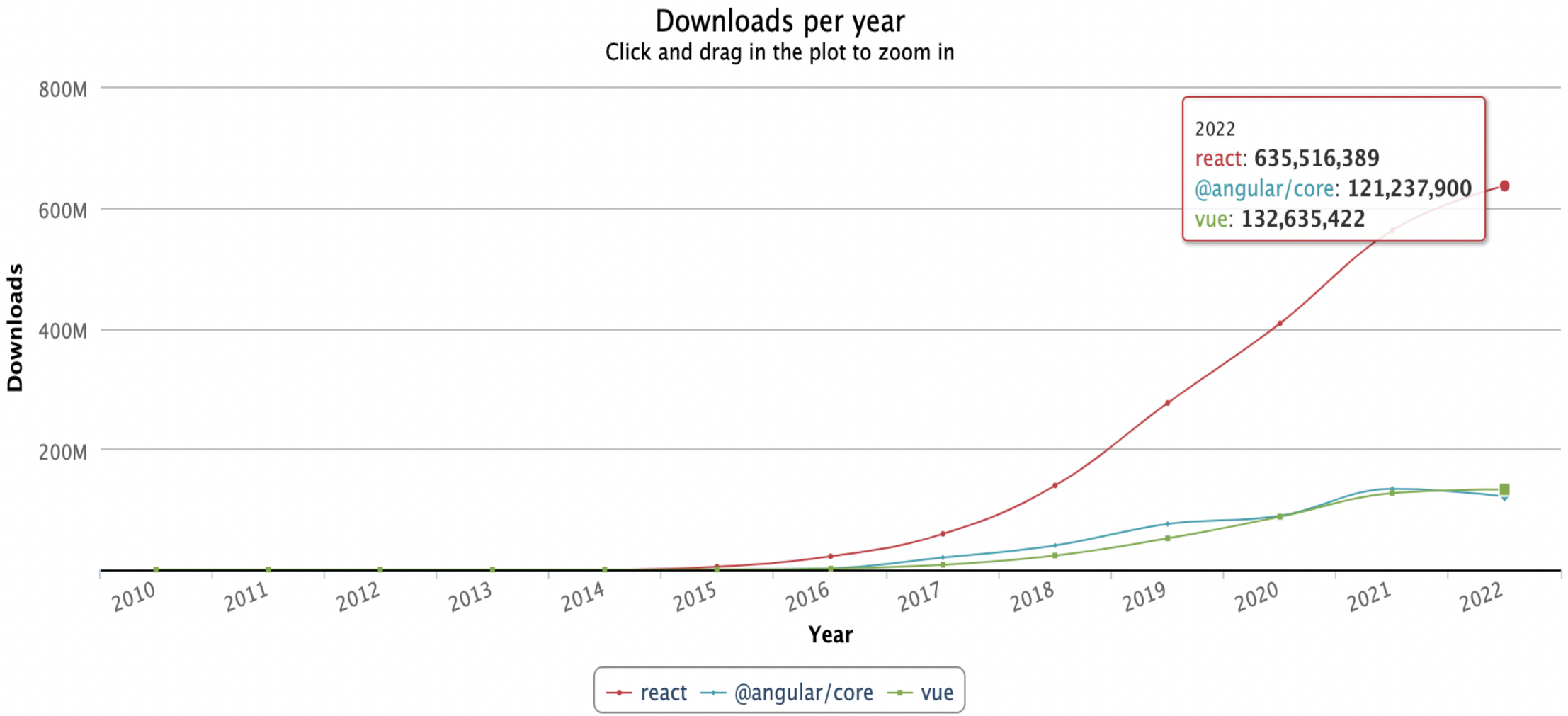
or the user...

Improve the user experience

- 🚄 **Quick** - load quickly your first page
- 🤹 **Interactive** - allow lazy-loading for others resources
- 🛠️ **Server Side Rendering** - prepare render with the backend

loiclegoff @llegoff loiclegoff.com

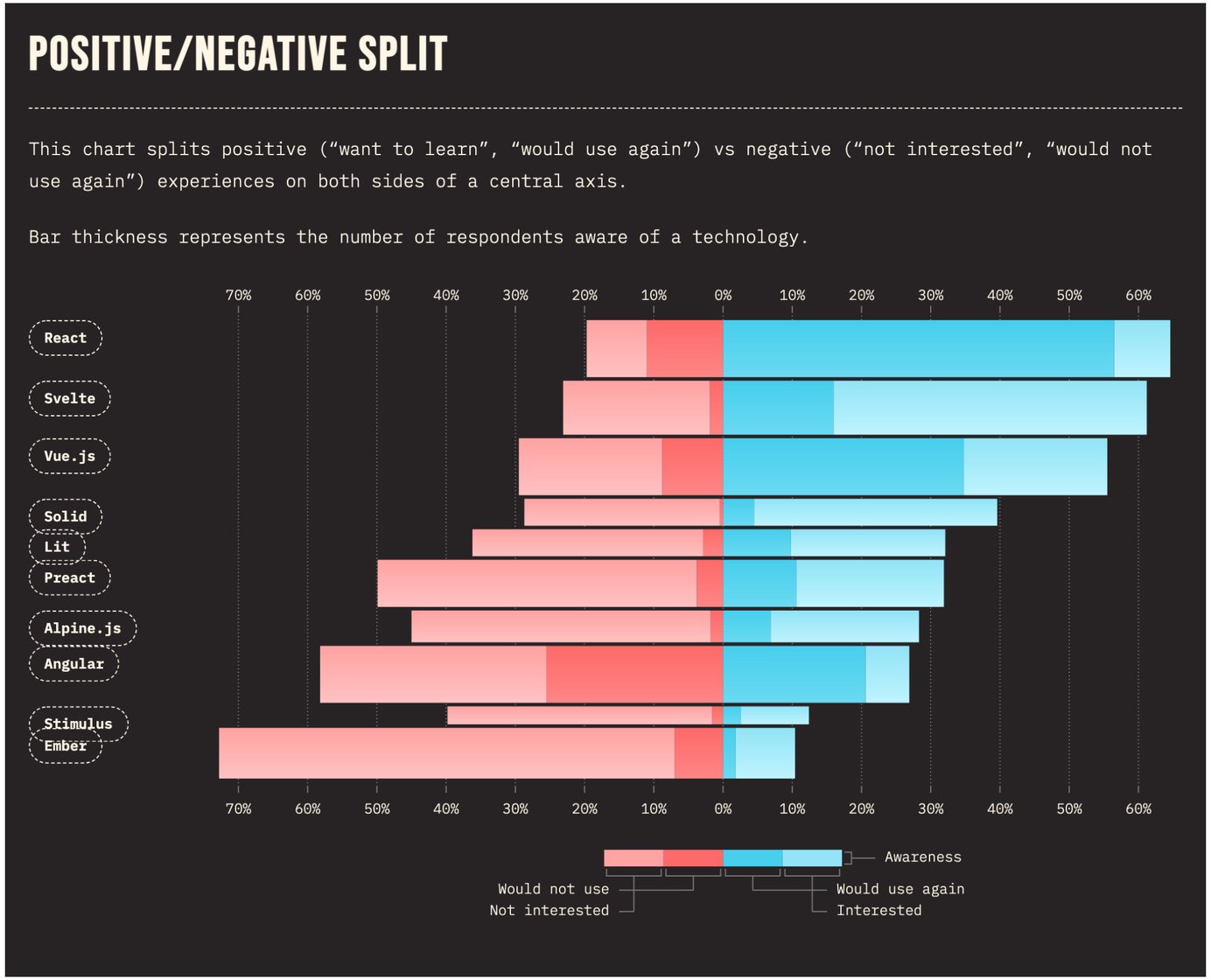
Why React ?



source: npm-stat

loiclegoff @llegoff loiclegoff.com

Why React ?



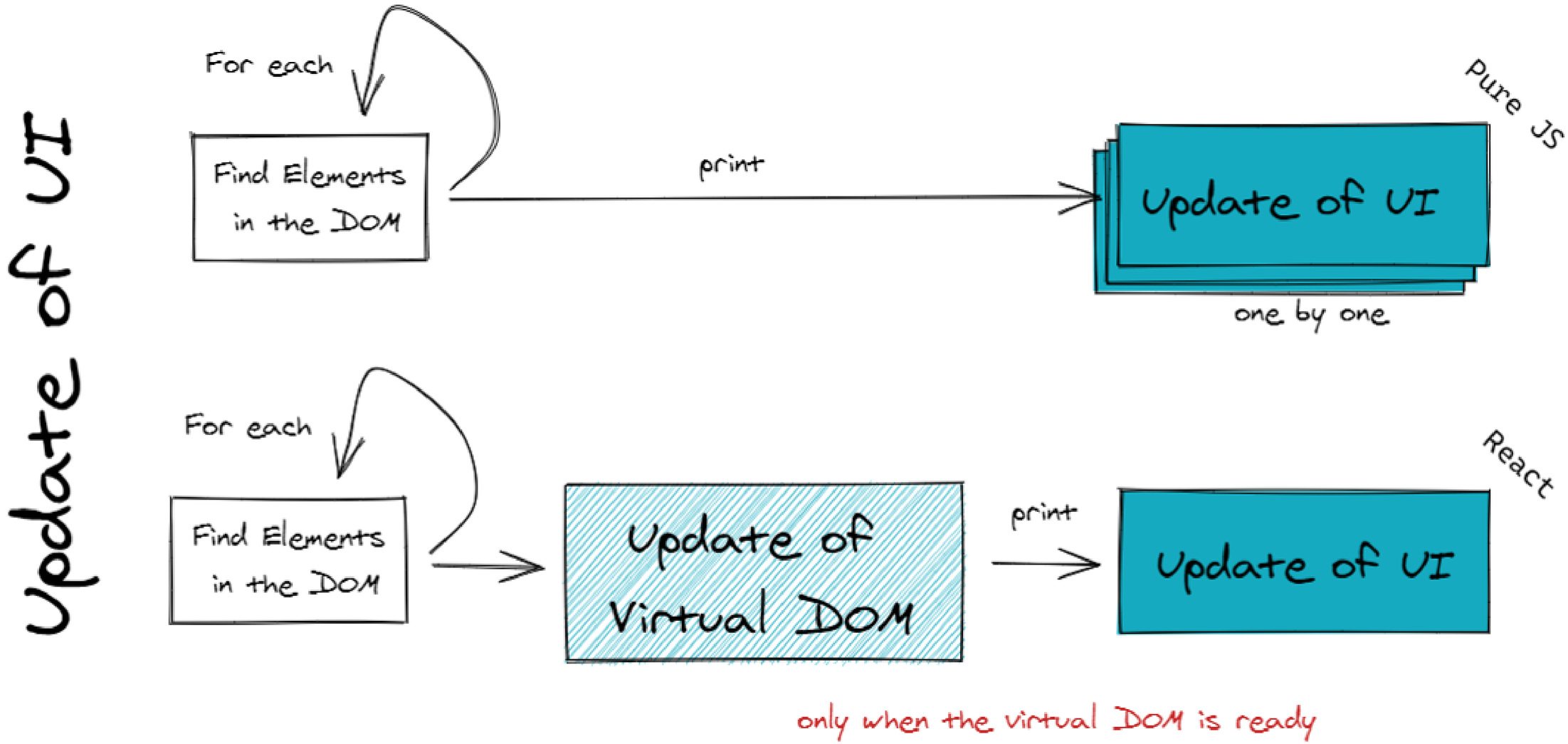
source: stateofjs

Pure JS vs React

loiclegoff @llegoff loiclegoff.com

Pure JS vs React : Concrete case

I need to update my UI ...trigger by an event (from user or server)



Pure JS vs React : Concrete case

I need to update my UI

With Pure JS : Concrete case

1. Find the elements to update
2. Set directly the value in the DOM

```
var elem = document.querySelector('#some-elem');
// Set HTML content
elem.innerHTML = 'We can dynamically change the HTML. We
```

If many, you need to do it for each element... With an update of the DOM

Read more about Virtual DOM

With React

1. Update the state
2. Triggers a new render of each associated component
3. Updates the Virtual DOM and print in one time in the DOM

State and DOM are synchronized

Let's go: My first Hello World with React

Create a static HTML file

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />

    <title>Hello React!</title>

    <!-- import deps -->
    <script src="https://unpkg.com/react@18.3.1/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@18.3.1/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>

  <body>
    <div id="root"></div>

    <script type="text/babel">
      // React code will go here
    </script>
  </body>
</html>
```

Let’s go: My first Hello World with React

- Create a React function component called `App` with a `return` which is used to render DOM nodes

```
<script type="text/babel">
    const App = () => {
          return <h1>Hello world!</h1>
    }
</script>
```

- Use the React DOM `render()` method to render the `App` function we created into the `root` div in our HTML

```
<script type="text/babel">
    const App = () => {
          return <h1>Hello world!</h1>
    }
    const rootElement = document.getElementById("root");
    const root = ReactDOM.createRoot(rootElement);
    root.render(
       <App />
    );
</script>
```

React Concepts

React concepts

- Templating with **JSX**
- DOM manipulation with **Virtual DOM**
- Develop with `Component` approach
- Implements **one-way reactive data flow** (Parent => Children)

 Templating : JSX

JSX is a **syntax extension for JavaScript** that lets you write **HTML-like markup inside a JavaScript file**. Although there are other ways to write components, most React developers prefer the conciseness of JSX, and most codebases use it.

To resume:

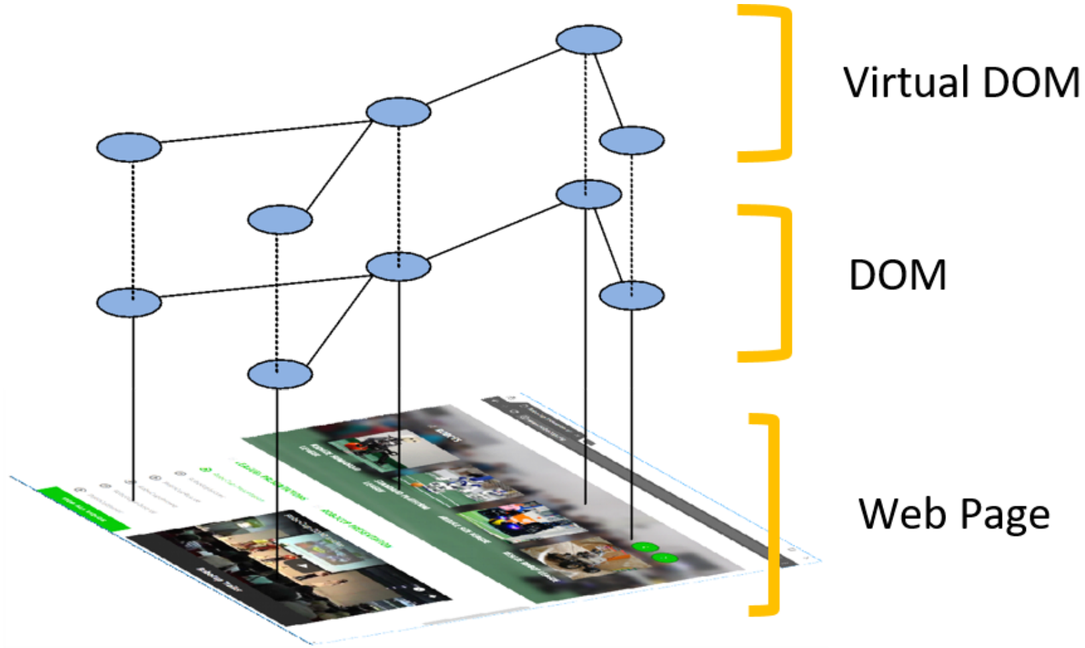
- Syntax near the Html syntax helping *describing visual result*
- *Associate* Javacript elts/results and UI elements
- Helping building React components (base of Virutal Dom)

How works?

```
import React from 'react';

function App() {
  return <h1>Hello World</h1>;
}
```

```
import React from 'react';

function App() {
  return React.createElement('h1', null, 'Hello world');
}
```

🏭 DOM manipulation : Virtual Dom



Virtual DOM

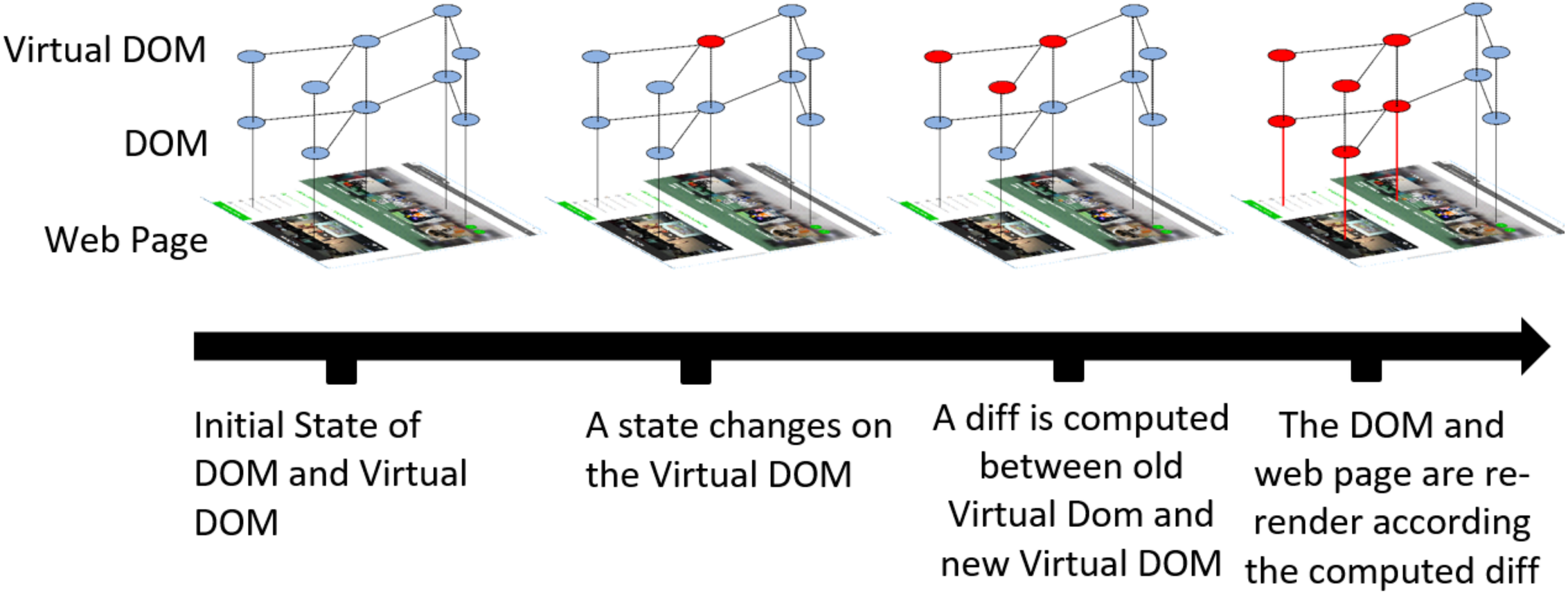
- **Light copy** of the DOM
- Quick navigation and update
- Detached from Browser specific

DOM

- Hierarchical component view
- Could update and navigate slowly
- Browser specific

🏭 DOM manipulation : Virtual Dom

Update exemple



React objects

ReactElement

- Lowest type of virtual dom

ReactNode

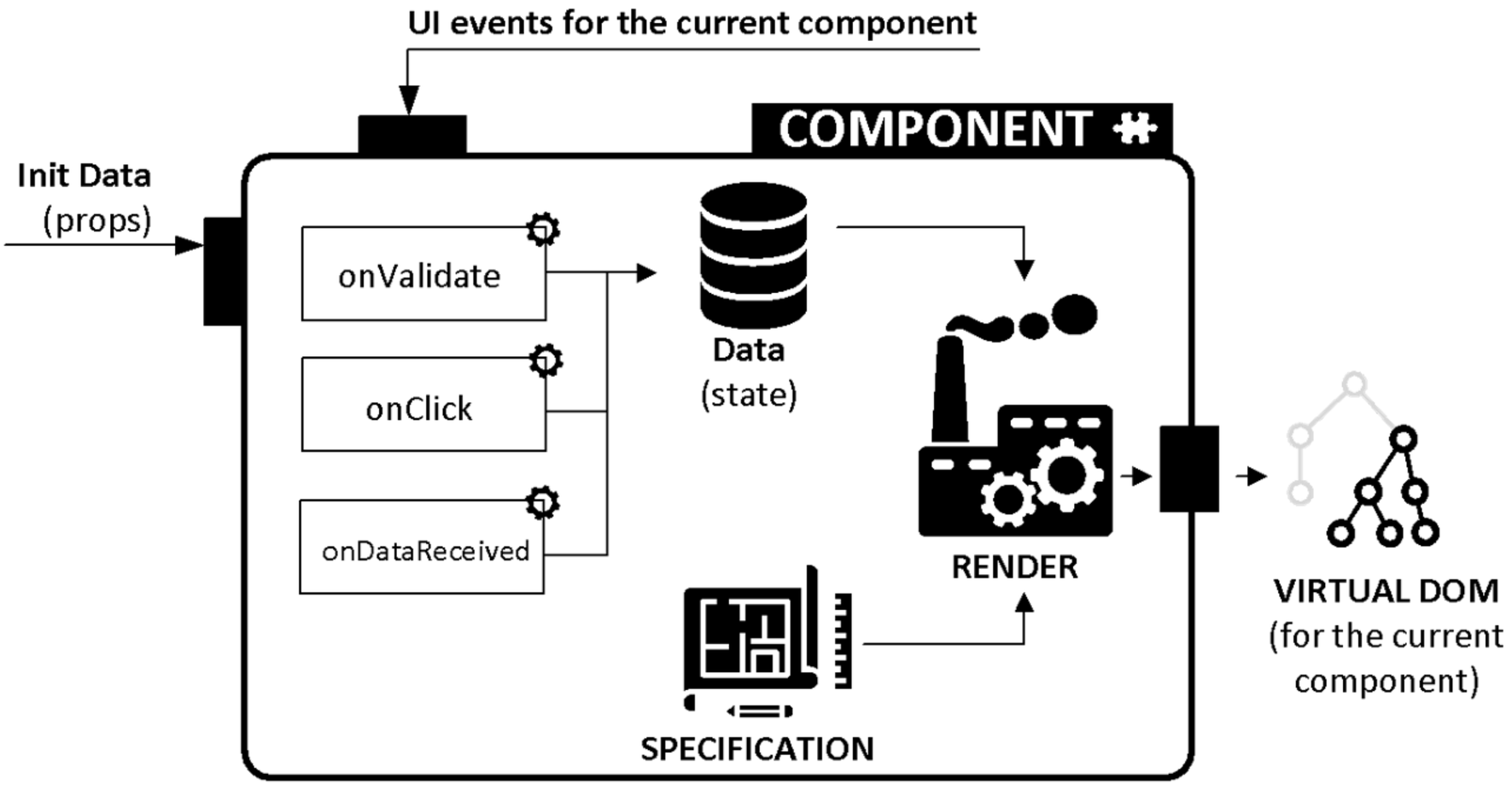
- Hierachical Element of the virtual dom
- ReactElement, string, number
- Array of virtual nodes

ReactComponent

- Specification of how to build react elements

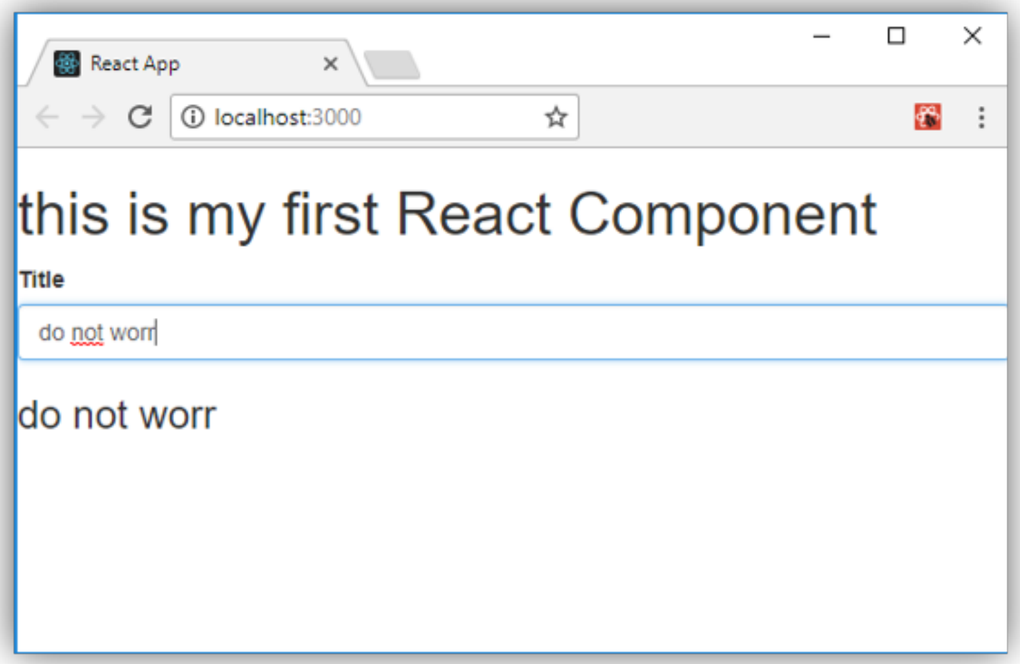
React Component

In react.js everything (mostly) is a component



React Component

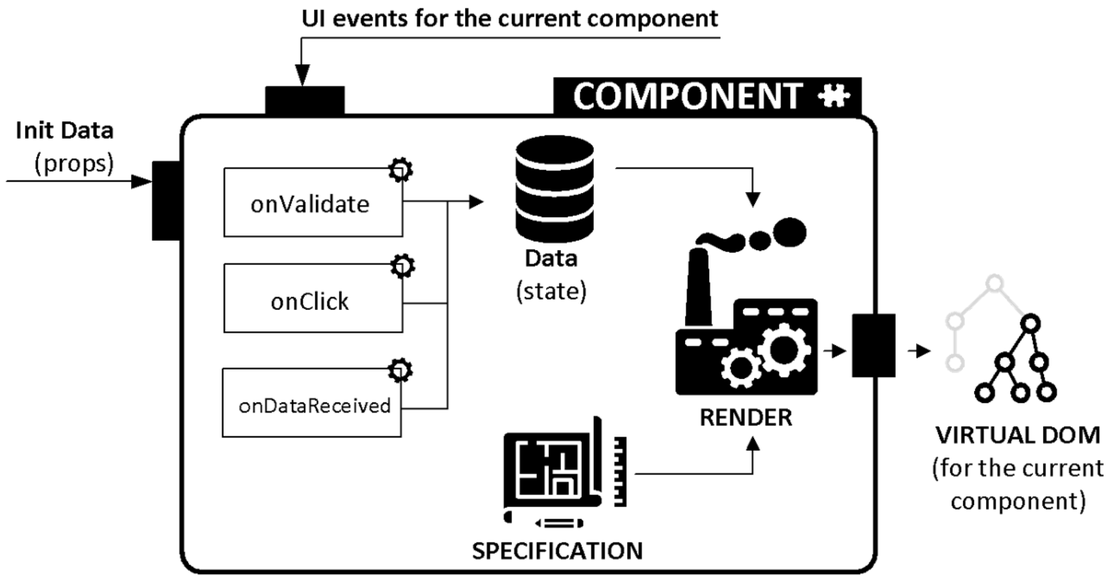
```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
```




```
import React, { useState } from 'react'
export const App = (props) => {
  const [title, setTitle] = useState(props.title)

  const handleChangeTitle = (e) => {
    setTitle(e.target.value)
  }

  return (
    <div className='App'>
      <h1> this is my first React Component</h1>
      <label htmlFor='titleInput'>Title</label>
      <input
        type='text'
        id='titleInput'
        onChange={handleChangeTitle}
        value={title}
      />
      <h3>{title}</h3>
    </div>
  )
}
```

React Component

```
import React, { useState } from 'react'
export const App = (props) => {
  const [title, setTitle] = useState(props.title)

  const handleChangeTitle = (e) => {
    setTitle(e.target.value)
  }

  return (
    <div className='App'>
      <h1> this is my first React Component</h1>
      <label htmlFor='titleInput'>Title</label>
      <input
        type='text'
        id='titleInput'
        onChange={handleChangeTitle}
        value={title}
      />
      <h3>{title}</h3>
    </div>
  )
}
```



ES6 Syntax

Javascript based on the ES6 standard

- let and **const** in addition to var
- Arrow Functions (=>)
- Object/Array Destructuring (spread operator)
- Set of libraries (promises, new collections, types arrays)

Thinking in React

Imagine that you already have a JSON API and a mockup from a designer.

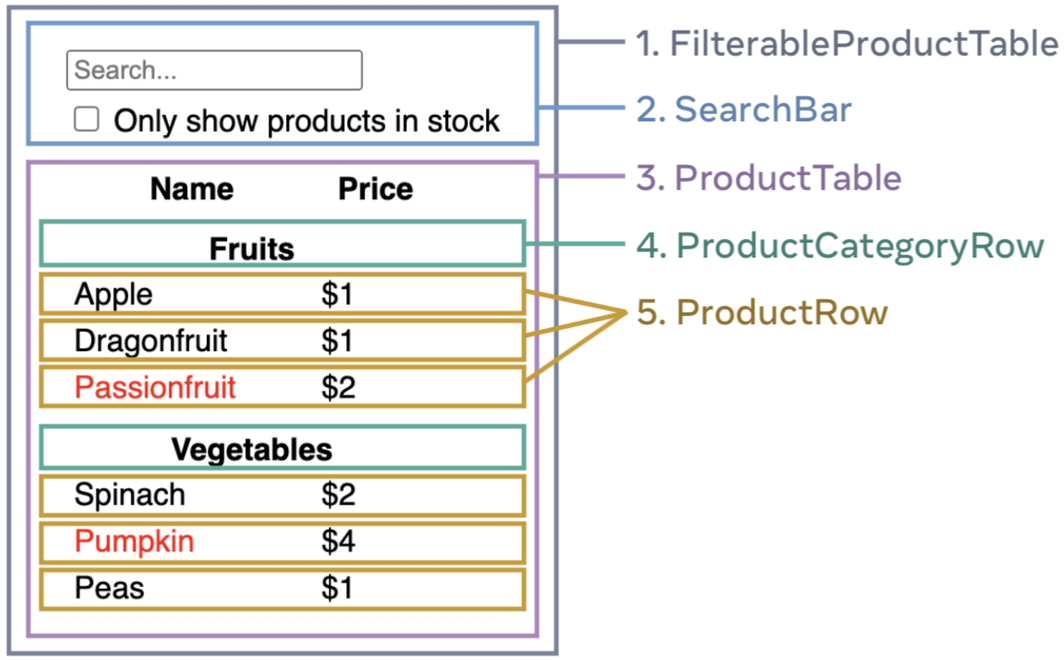
The JSON API returns some data that looks like this:

```
[
  { category: "Fruits", price: "$1", stocked: true, name: "Apple" },
  { category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit" },
  { category: "Fruits", price: "$2", stocked: false, name: "Passionfruit" },
  { category: "Vegetables", price: "$2", stocked: true, name: "Spinach" },
  { category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin" },
  { category: "Vegetables", price: "$1", stocked: true, name: "Peas" }
]
```

The mockup looks like this:

Extract from React Documentation, read more about Thinking in React

Thinking in React



1. `FilterableProductTable` (grey) contains the entire app.
2. `SearchBar` (blue) receives the user input.
3. `ProductTable` (lavender) displays and filters the list according to the user input.
4. `ProductCategoryRow` (green) displays a heading for each category.
5. `ProductRow` (yellow) displays a row for each product.

Extract from React Documentation, read more about Thinking in React

Describe the UI

Importing and Exporting Components

```
function ProductRow() {
  return (
    <tr>
      <td>Name</td>
      <td>Price</td>
    </tr>
  );
}
```

```
export default function ProductTable() {
  return (
    <table>
      <ProductCategoryRow />
      <ProductRow />
      <ProductRow />
      <ProductRow />
    </table>
  );
}
```

Extract from React Documentation, read more about Importing and Exporting Components

Describe the UI

Passing Props to a Component

Step 1: Pass props to the child component

```
export default function OneProducTable() {
  return (
    <ProductRow
      info={{ name: 'Orange', picture: 'https://urlforpicture' }}
      price={100}
    />
  );
}
```

Step 2: Read props inside the child component

```
function ProductRow({ info, price }) {
  // info and price are available here
}
```

Extract from React Documentation, read more about Passing Props to a Component

Describe the UI

Conditional Rendering

- In React, you control branching logic with JavaScript.
- You can return a JSX expression conditionally with an `if` statement.
- You can conditionally save some JSX to a variable and then include it inside other JSX by using the curly braces. ex: `<div>{variable}</div>`
- In JSX, `{cond ? <A /> : <B />}` means “if cond, render `<A />`, otherwise `<B />`”.
- In JSX, `{cond && <A />}` means “if cond, render `<A />`, otherwise nothing”.
- The shortcuts are common, but you don’t have to use them if you prefer plain `if`.

Extract from React Documentation, read more about Conditional Rendering

Describe the UI

Rendering Lists

You can store that data in JavaScript objects and arrays and use methods like `map()` and `filter()` to render lists of components from them

You need to give each array item a key

```
<li key={product.id}>...</li>
```

```
export default function List() {
  const listItems = products.map(product =>
    <li key={product.id}>
      <img
        src={getImageUrl(product)}
        alt={product.name}
      />
      <p>
        <b>{product.name}</b>
          {' ' + product.category + ' '}
          known for {product.benefit}
      </p>
    </li>
  );
  return <ul>{listItems}</ul>;
}>
```

Extract from React Documentation, read more about Rendering Lists

Adding Interactivity

- You can add **event handlers** (ex: `onClick`) to your JSX.
- You can manage a component memory with the hook : `useState`

```
const [index, setIndex] = useState(0);
```

The `useState` Hook returns a pair of values: the current state and the function to update it. Prefer atomic state when values are not related.

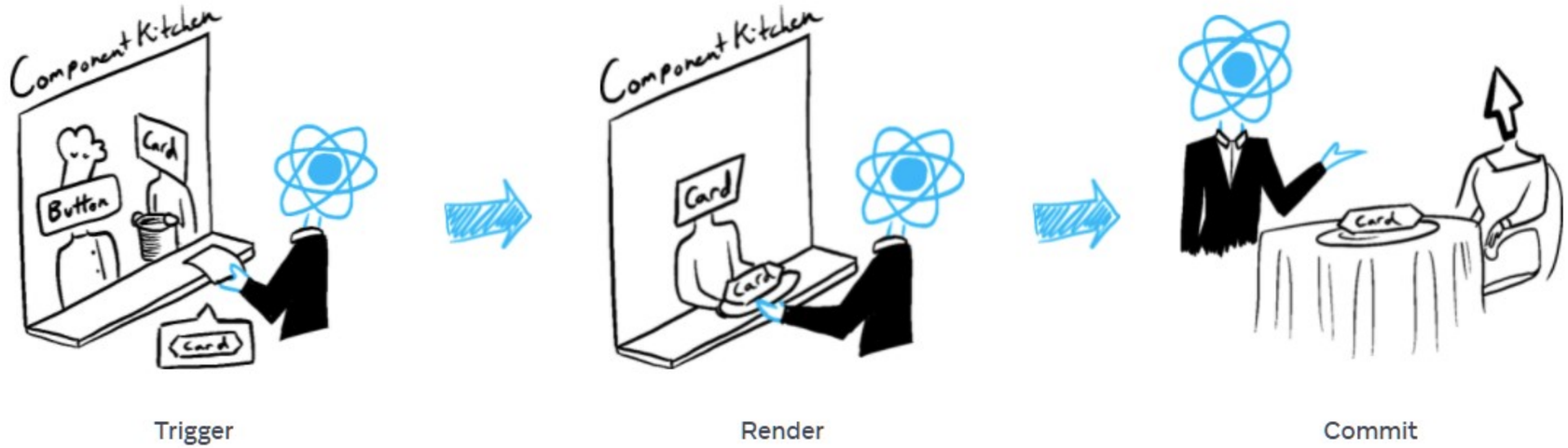
Extract from React Documentation, read more about Responding to Events or State

```
export default function Gallery() {
  const [index, setIndex] = useState(0);

  function handleClick() {
    setIndex(index + 1);
  }

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
      </h2>
      <p>
        {sculpture.description}
      </p>
    </>
  );
}
```

React Component Lifecycle

Imagine that your components are *cooks in the kitchen*. In this scenario, React is the waiter who puts in requests from customers and brings them their orders.

1. **Triggering** a render (delivering the diner's order to the kitchen)
2. **Rendering** the component (getting the order from the kitchen)
3. **Committing** to the DOM (placing the order on the table)



Trigger Render Commit

Extract from React Documentation, read more about Render and Commit

Updating : object are immutable

React needs to see another reference to trigger a re-render. Without update of reference, your change is not visible.

Updating and object without mutation

Use **spread operator**. Don't do : `person.firstName = newValue`. This is a mutation.

```
const newPerson = {
  ...person, // Copy the old fields
  firstName: newValue // But override this one
};
```

If we decompose the processus :

1. `const newPerson = {}` create a new reference of empty object
2. `{...person}` attach each value of attribute in the empty object with the same key
3. `firstName: newValue` replace the value of the key *firstName*, the value has a new reference too.

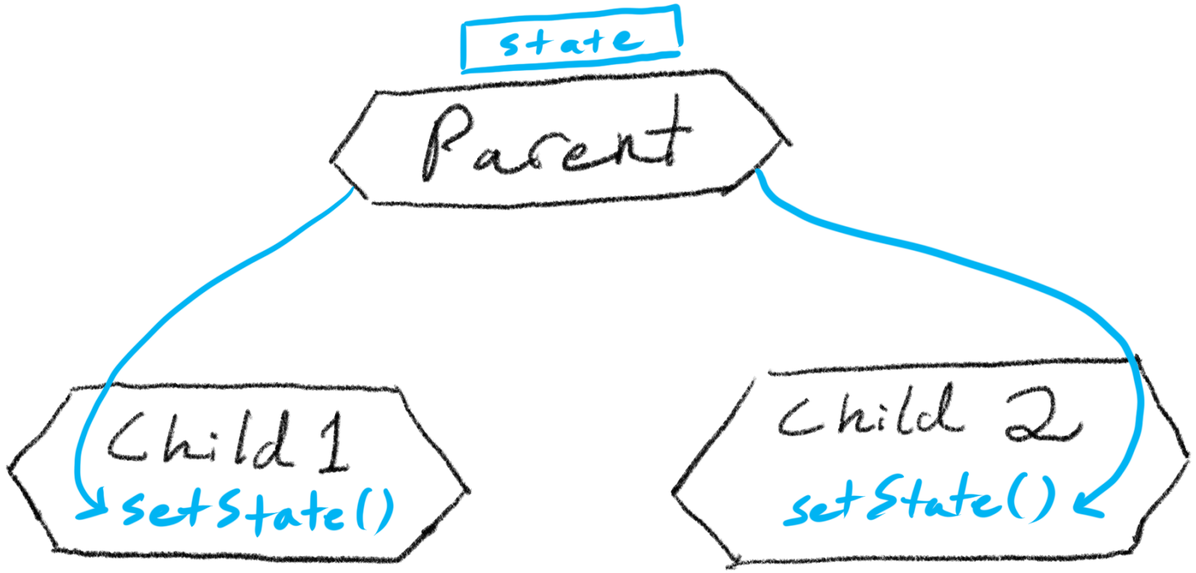
Updating arrays without mutation

	avoid (mutates the array)	**prefer (returns a new array)**
adding	push, unshift	concat, [...arr] spread syntax (example)
removing	pop, shift, splice	filter, slice (example)
replacing	splice, arr[i] = ... assignment	map (example)
sorting	reverse, sort	copy the array first (example)

Extract from React Documentation, read more about Updating arrays in state

Sharing State Between Components

- When you want to **coordinate two components**, move their state to their **common parent**.
- Then pass the information down through **props from their common parent**.
- Finally, **pass the event handlers** down so that the children can change the parent's state.



Extract from React Documentation, read more about Sharing State Between Components

Others Hooks

- `useReducer` like *useState*, this hook allow to save a state. This hook helps to separate the **update logic**.
- `useEffect` allow to trigger callback when deps has been updated.

```
useEffect(() => {
  // myMethod is called when deps change
  myMethod()
}, [deps, myMethod])
```

You can have an empty deps array. In this case, the callback will be trigger after the first render of the component. We can use this for a fetching (React Documentation).

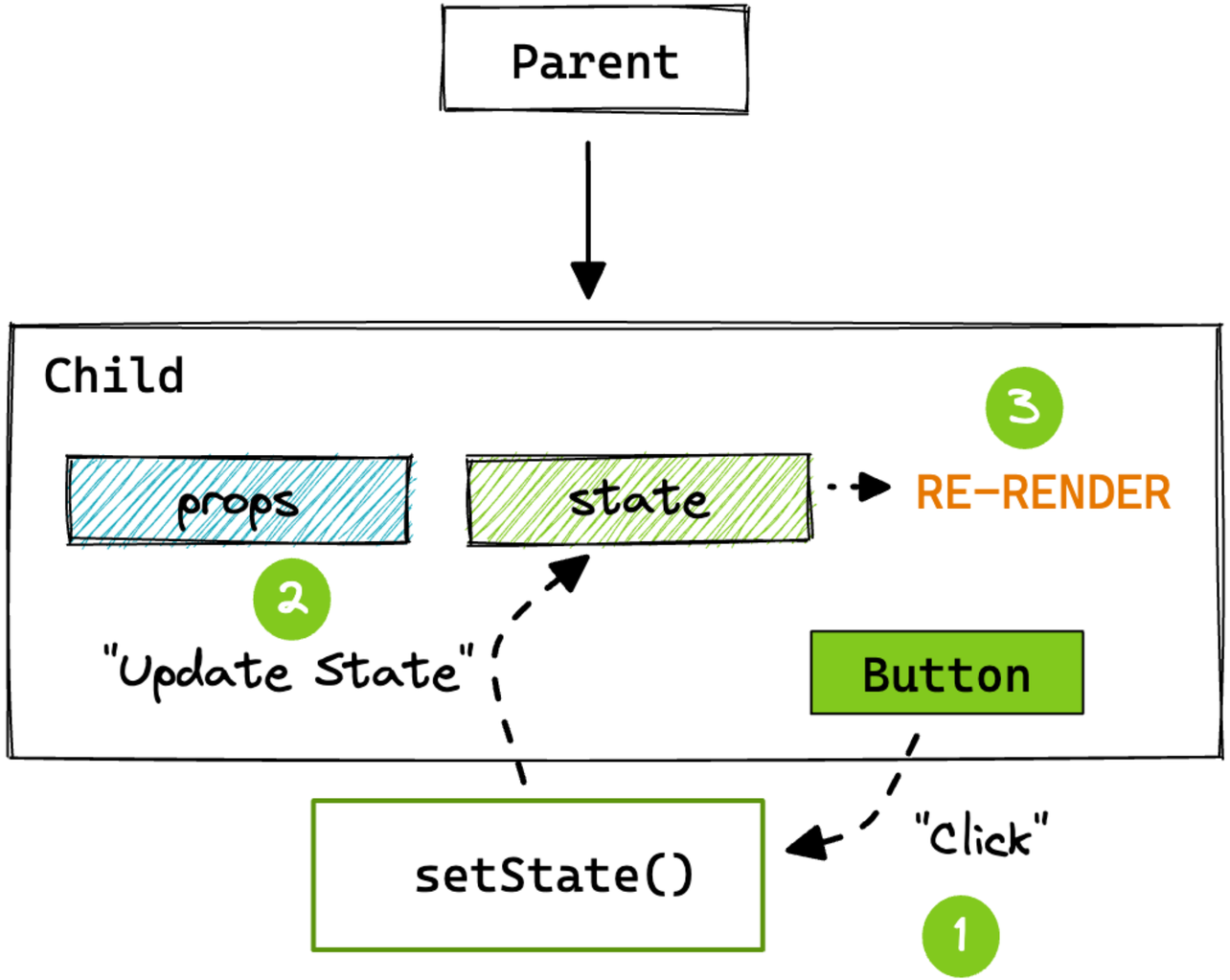
Advanced Hooks

- `useContext` allow to access of a context sharing between components.
- `useRef` allow to mutate directly a value. The reference is never updated.
- `useMemo` and `useCallback` allow to memorize value or callback. The usage is not required for the most use case (see Dan article)

Re-render

When does React re-render component?

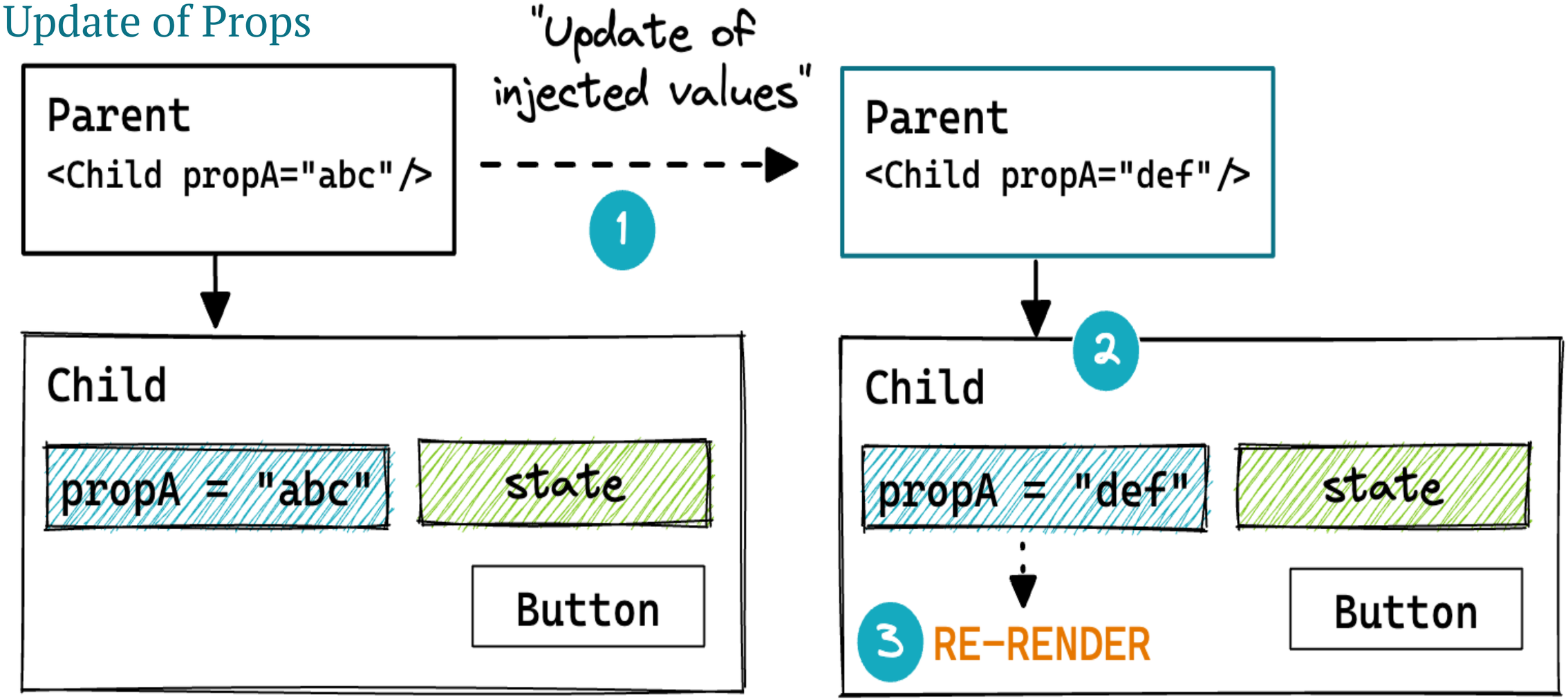
Update of State



Re-render

When does React re-render component?

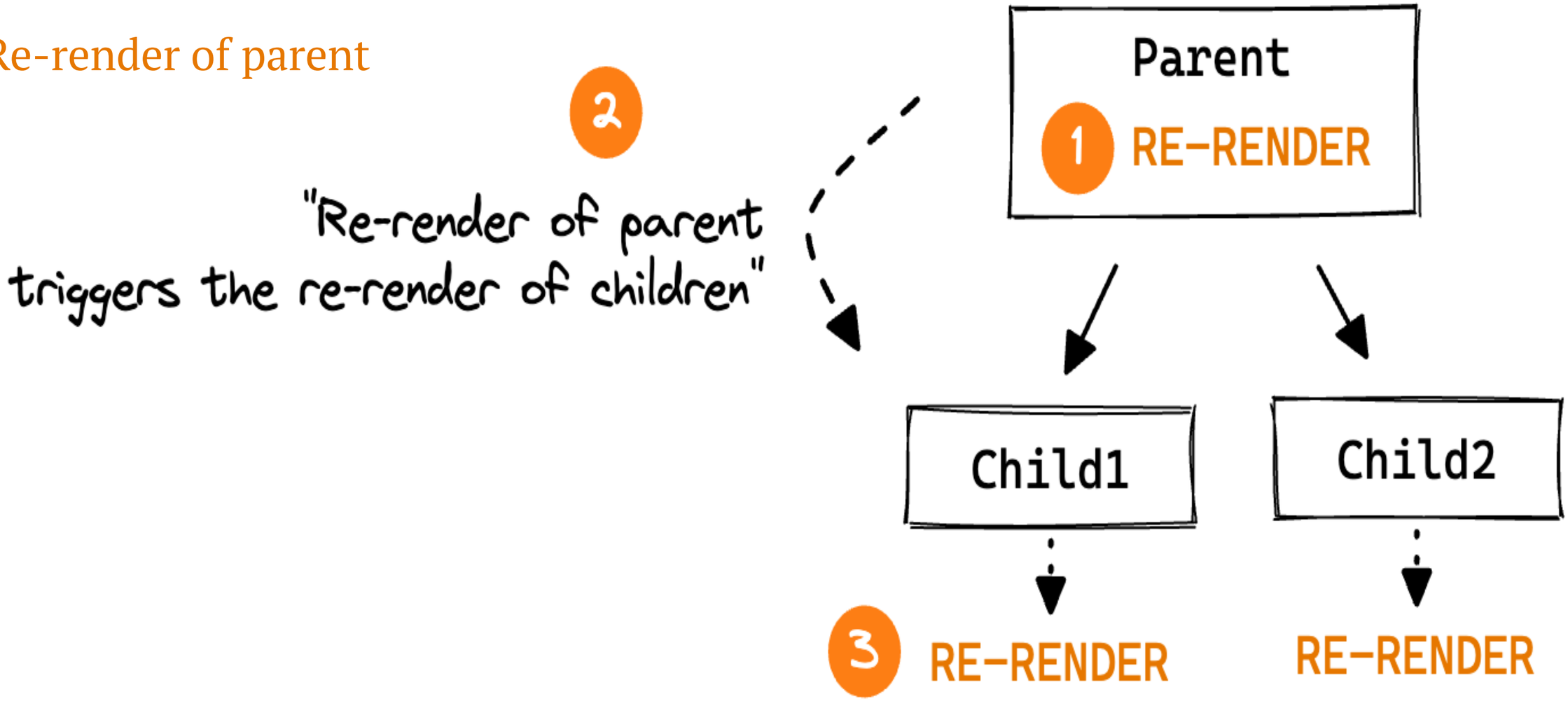
Update of Props



Re-render

When does React re-render component?

Re-render of parent



Re-render

When does React re-render component?

- Update of the props (reference change)
- Update of the state (setState change the reference of current state)
- The Parent is re-rendered (It can be avoid with `` `React.memo` ``)

Practice : Installation

Required:

- Node.js
- npm

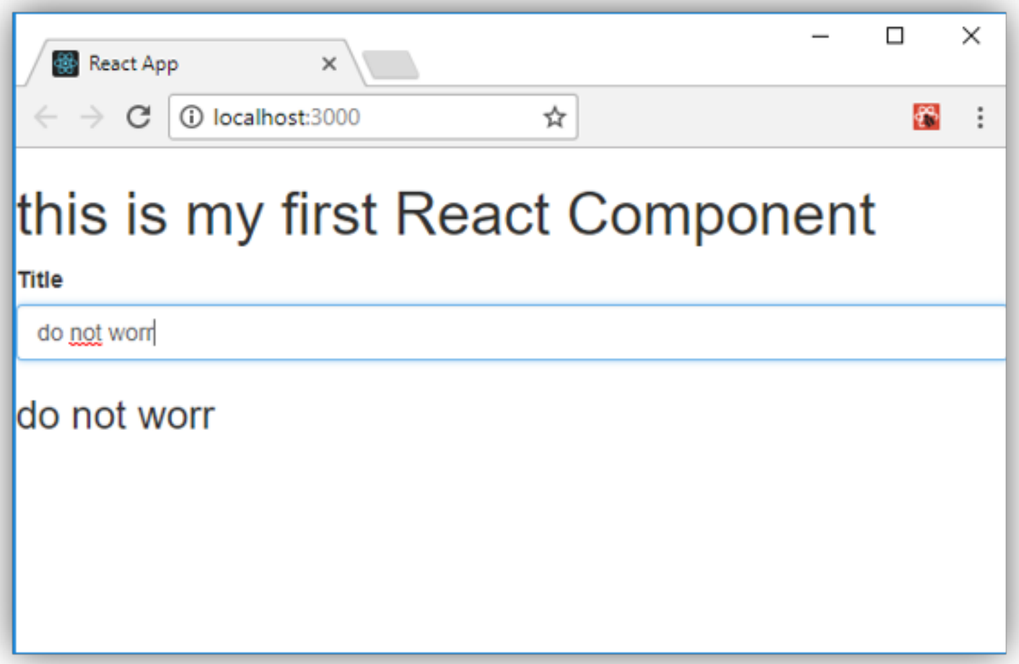
Go to https://github.com/loiclegoff/irc-react-2024?#create-your-app-folder

Practice

Step 0

See README

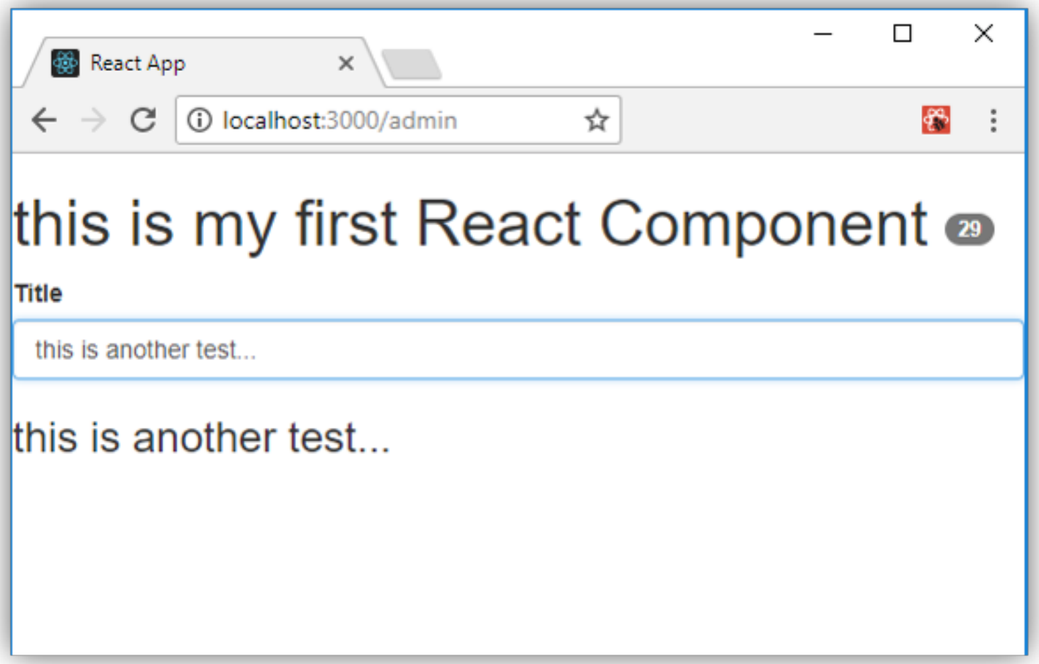
- Add bootstrap import in index.html
- Create App allowing to get as input a **title** and **print it below**
- Your App component must be initialized with the title property = `'default_title'`



Practice

Sept 0 (next part)

- Update the number of mouse over the printed title

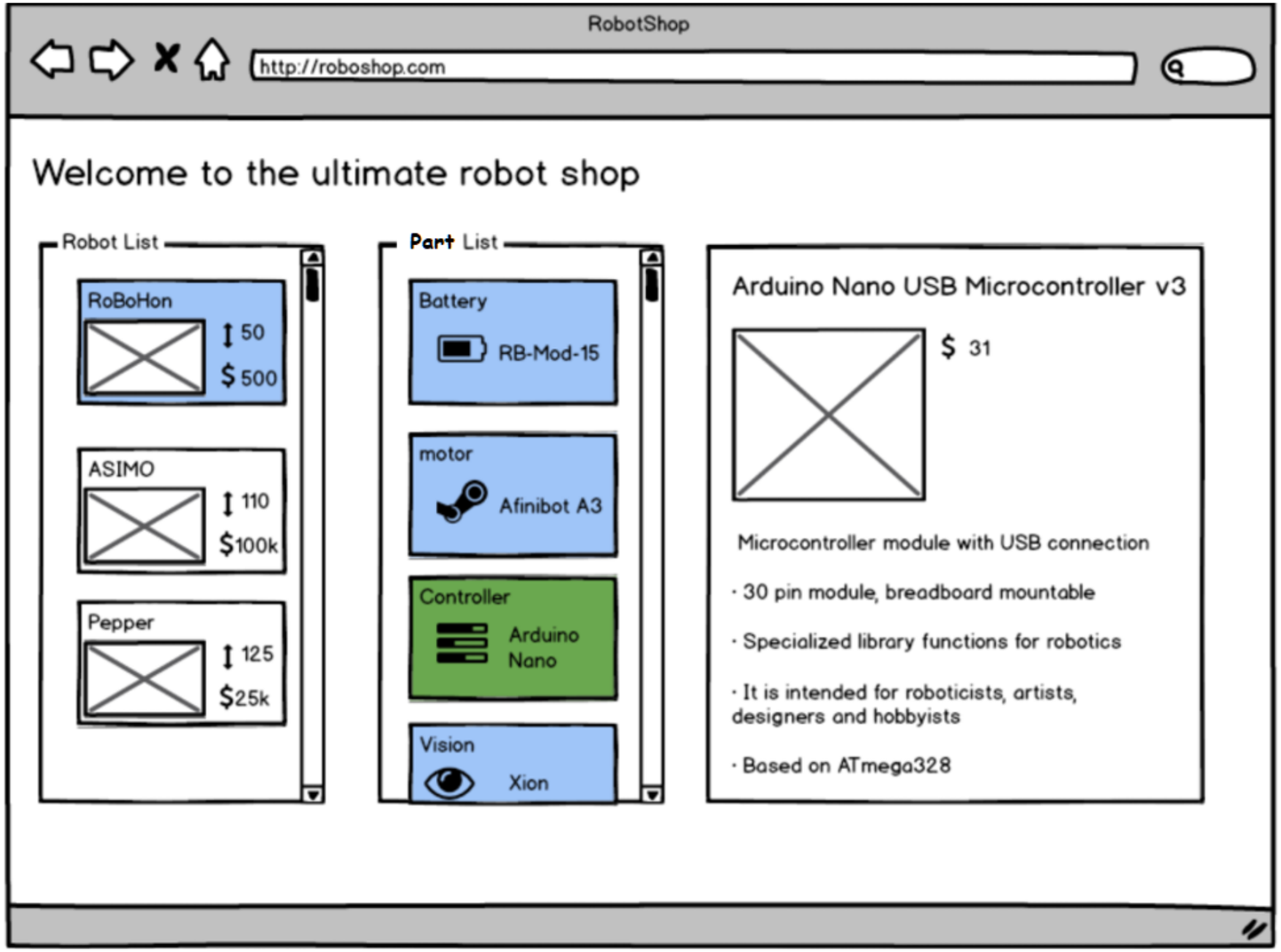


Step 1

See README

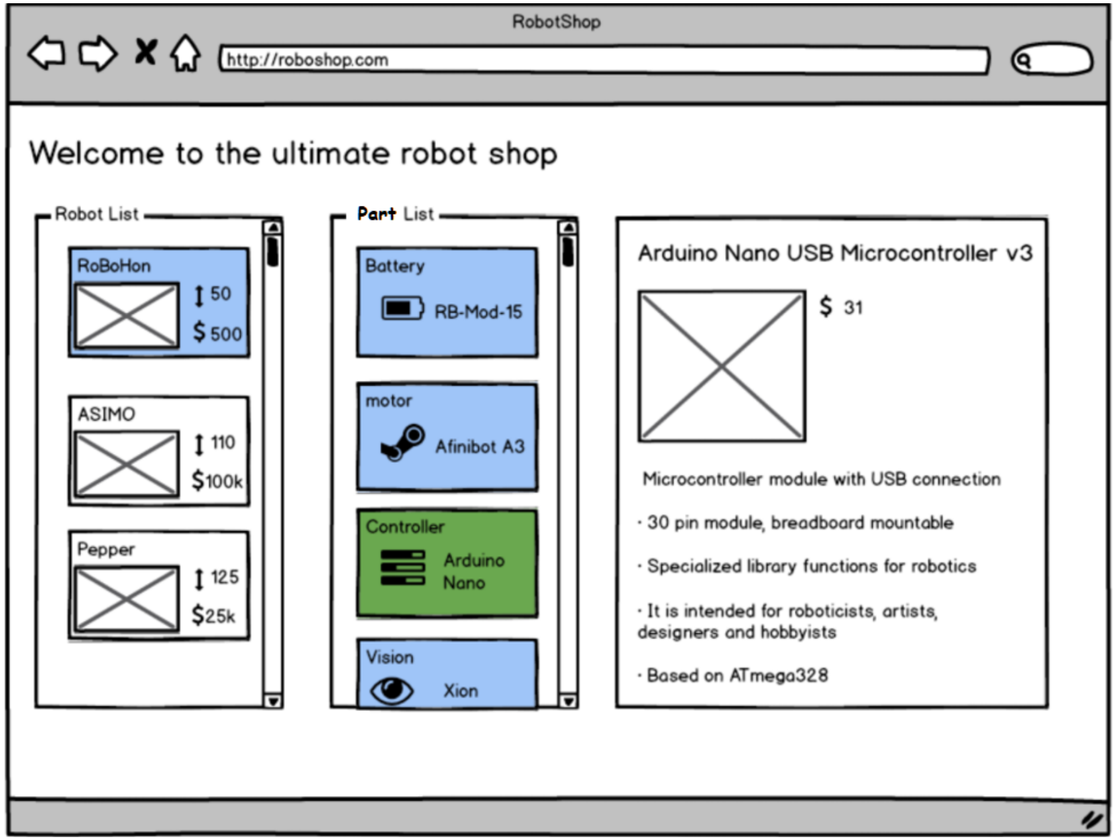
- Install `react-bootstrap`
- Use bootstrap components

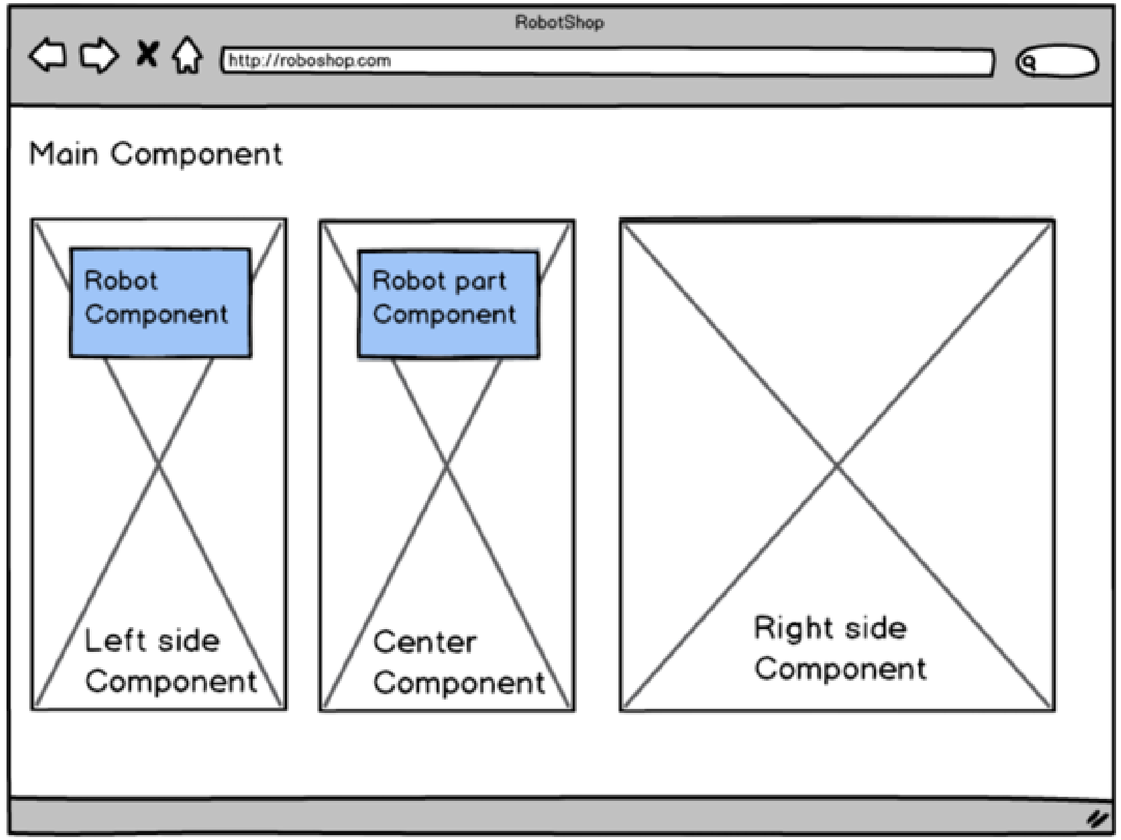
Practice



- As seen in "*Thinking in React*", split this mockup in components

Practice

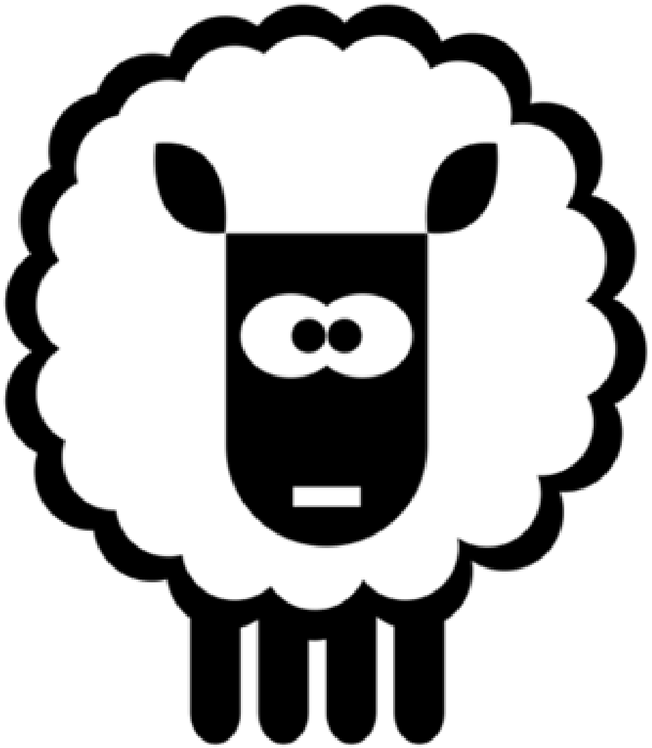




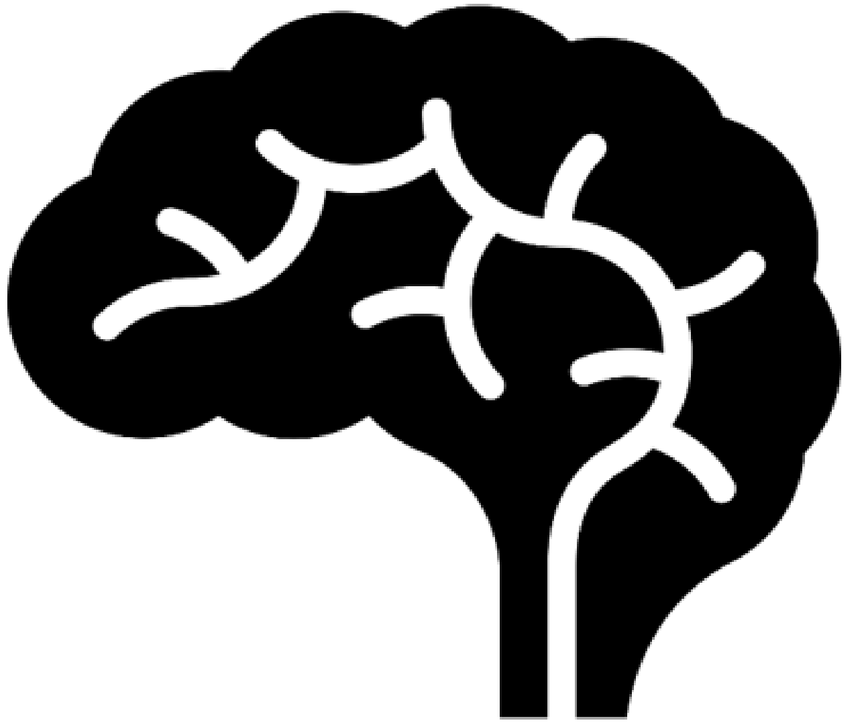
- Left side component component could be named `RobotList`

Best Practice

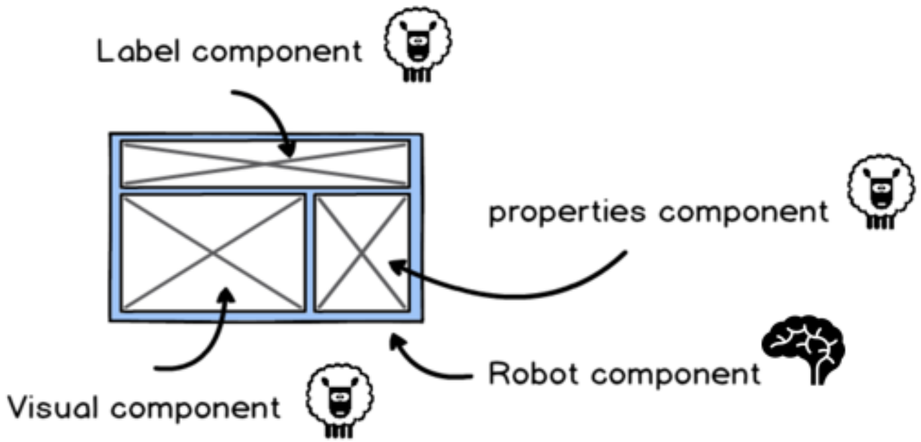
Displaying (presentational Component) Vs *Processing* (container Component)



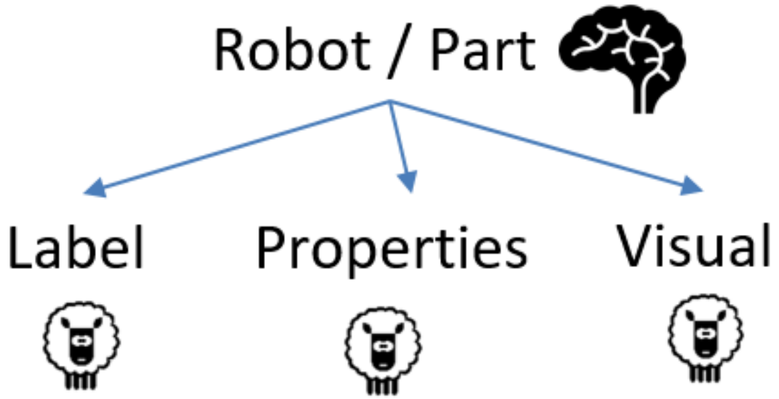
VS



Best Practice



Process Data



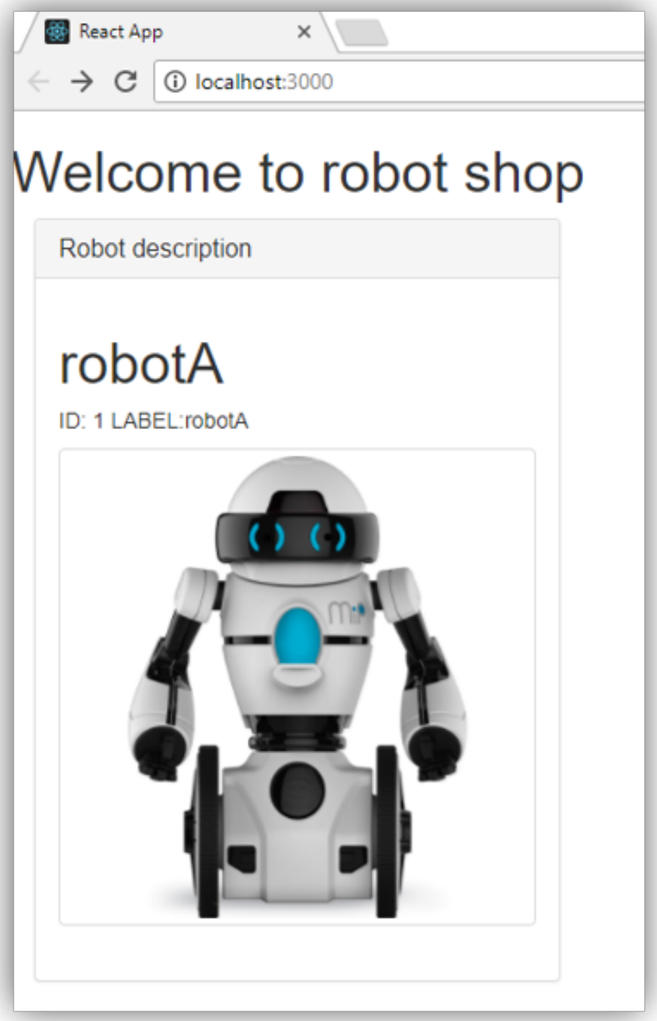
Display Data

Practice

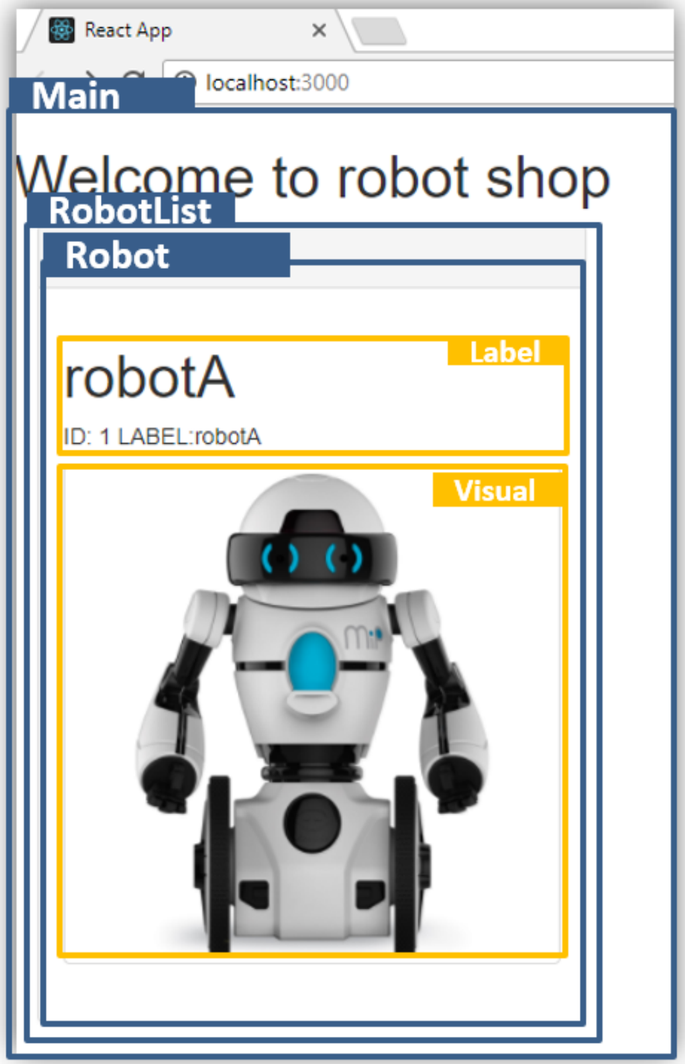
Step 2

See README

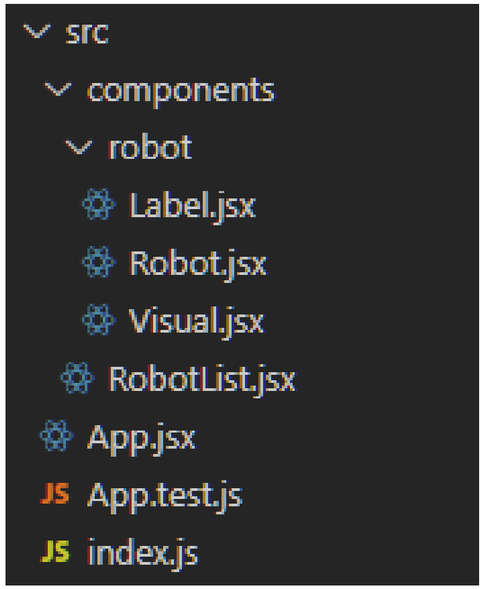
- A main component that fetches data from robots
- A Left side component : `RobotList`
- A `Robot` component
- A `Label` component for the Robot Component



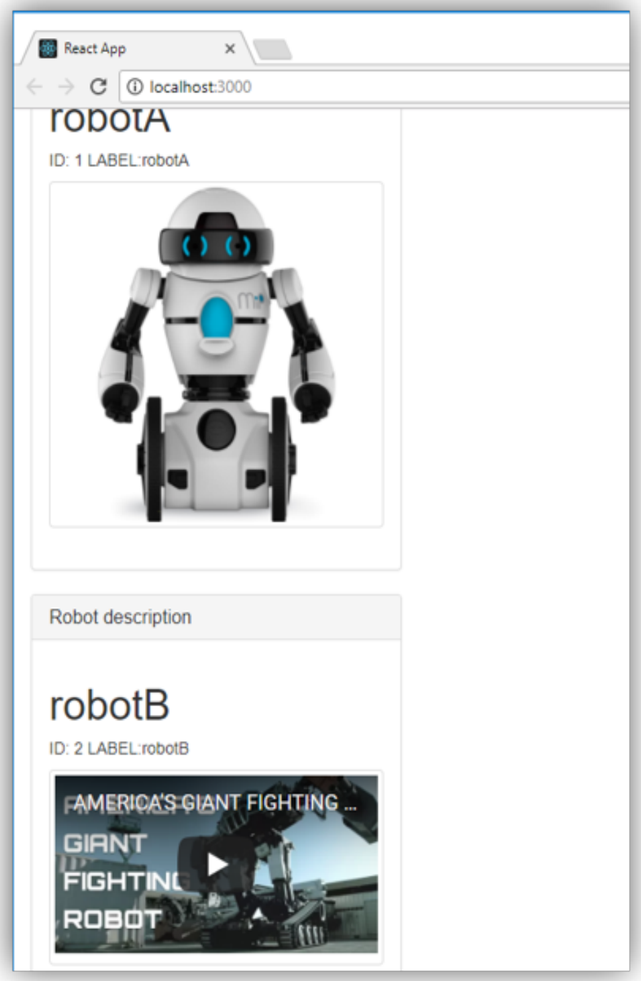
Practice : correction




```
src
|--- components
|    |--- robot
|    |    |    Label.jsx
|    |    |    Robot.jsx
|    |    |    Visual.jsx
|    |    RobotList.jsx
|    App.jsx
|    App.test.jsx
|    index.jsx
```



Practice



Step 3

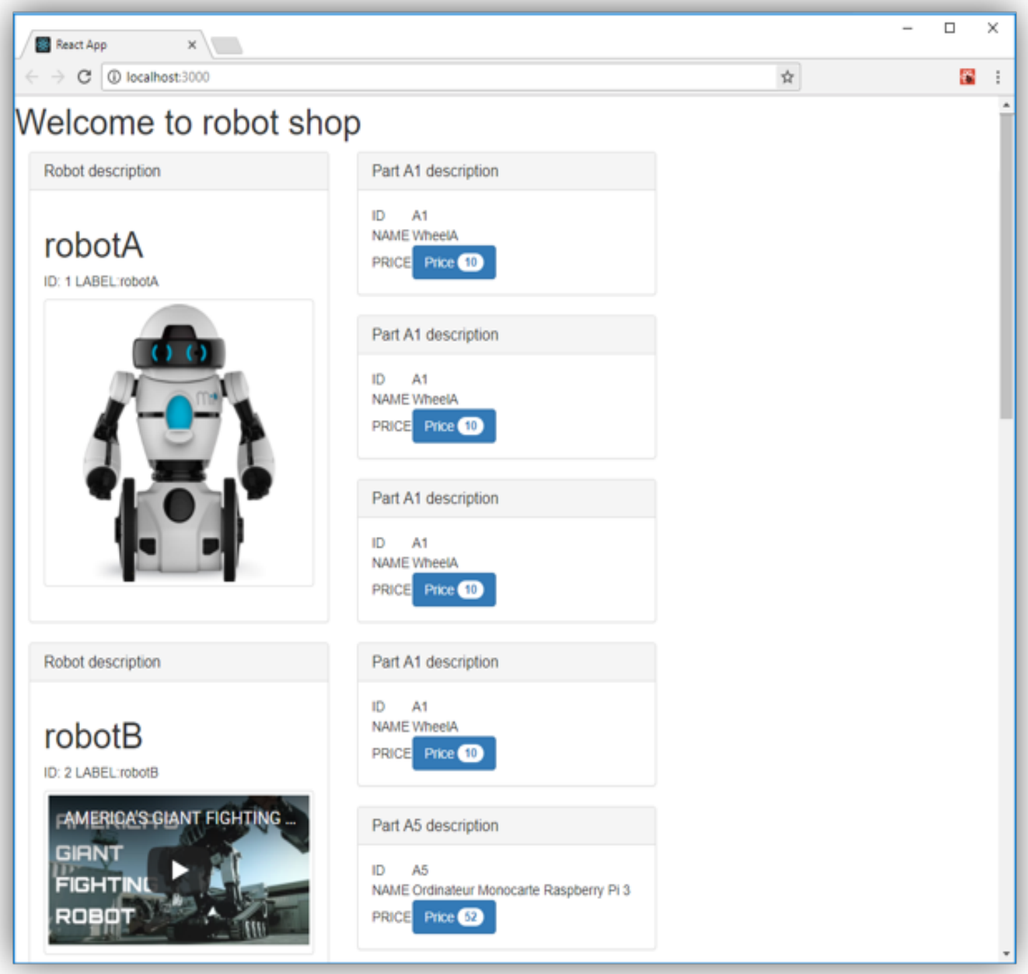
See README

- Same as previously but with a list of robots
- Some robot can have a visual as a video.

Tips

- Use `` `<array>.map((item) => <your code>);` `` to display list of objects
- Don't forget *key* that help react to identify which items have changed. More info here.

Practice

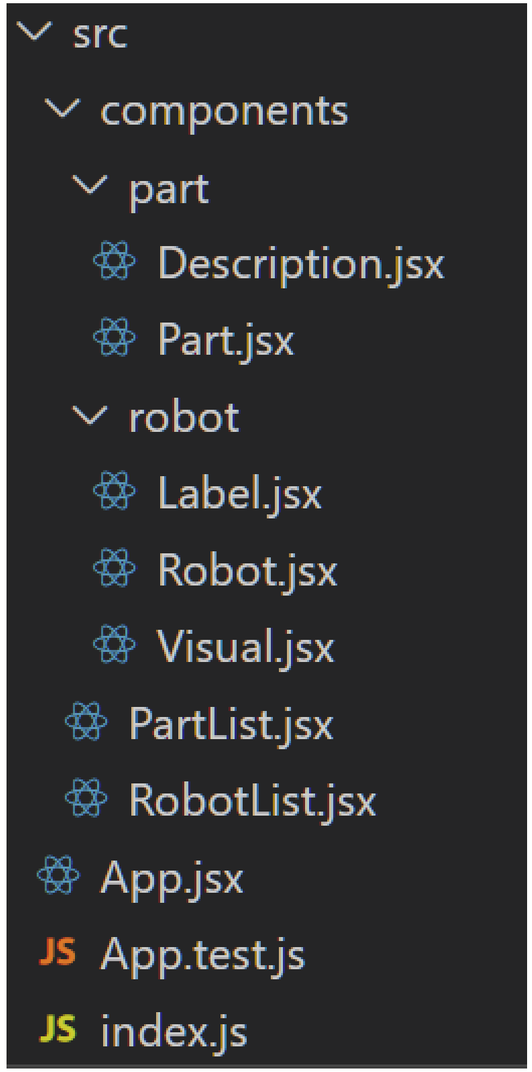


Step 4

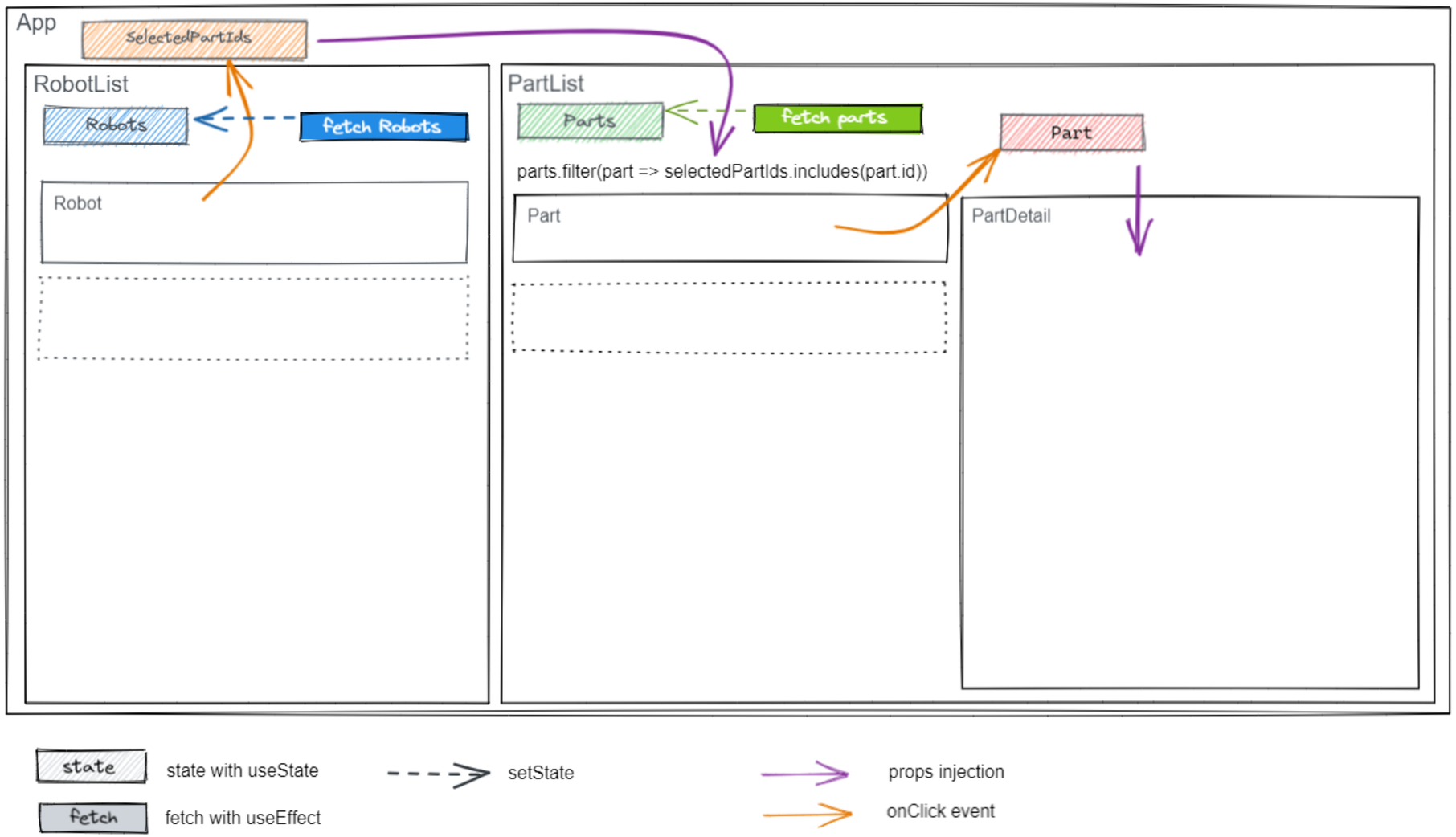
See README

- On paper, suggest a splitting in many components
- Add a `PartList` Component displaying the parts list
- Add a `Part` Component using
  - `Description` Component
  - `Price` Component
- Update the `PartList` Component to show only the part related to the selected robot

Practice : correction



Practice : correction

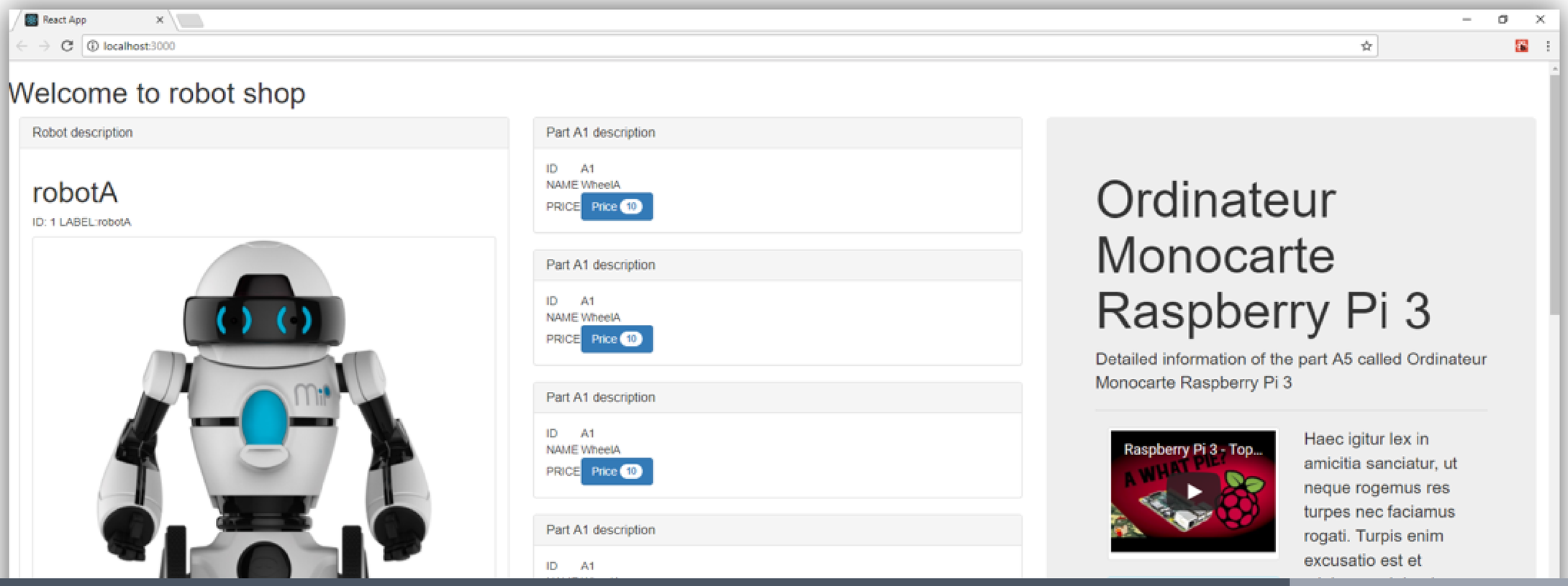


Practice

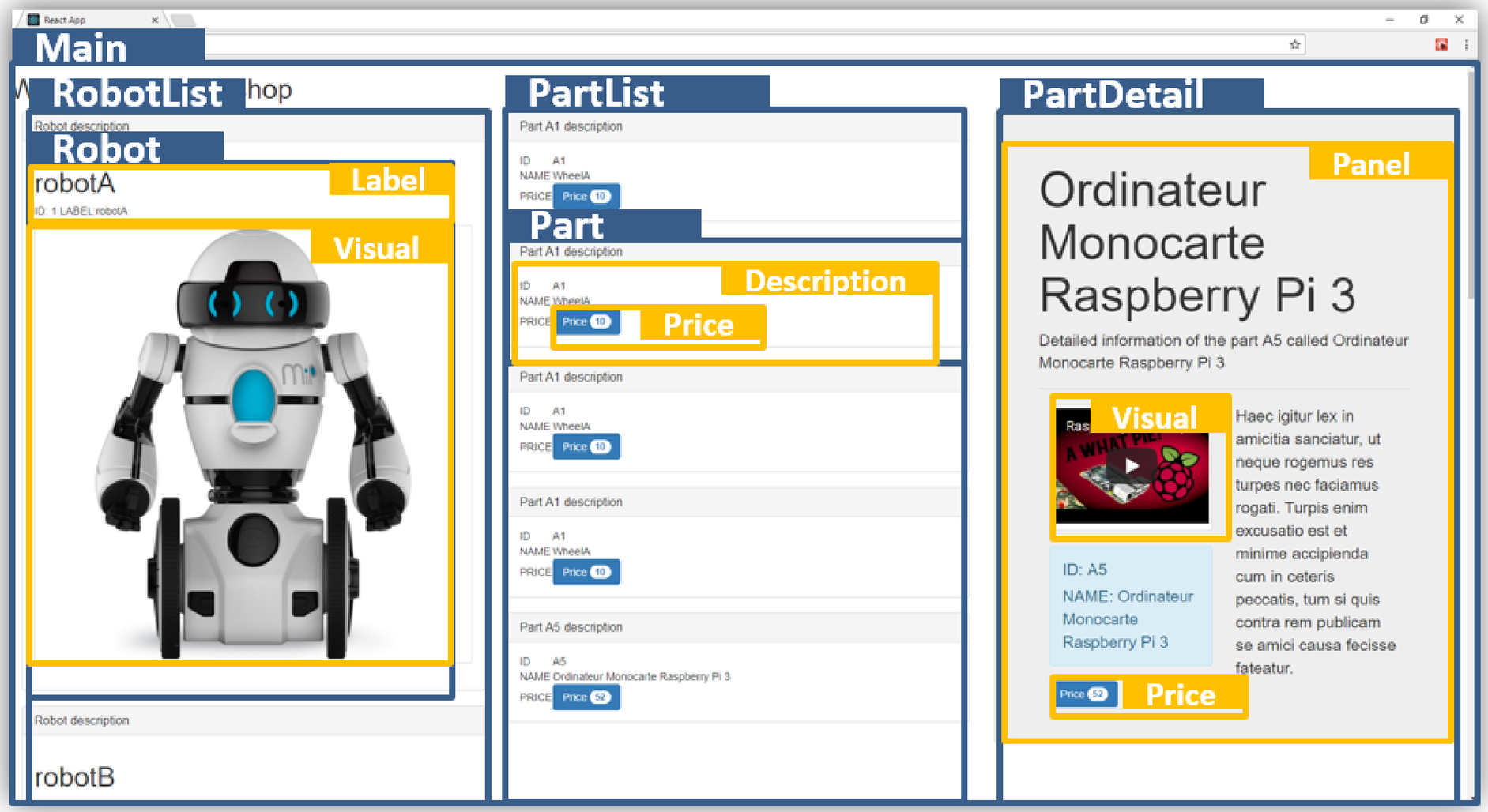
Step 4 bis

See README

- Add a `PartDetail` Component on the right using `Panel` Component displaying selected part



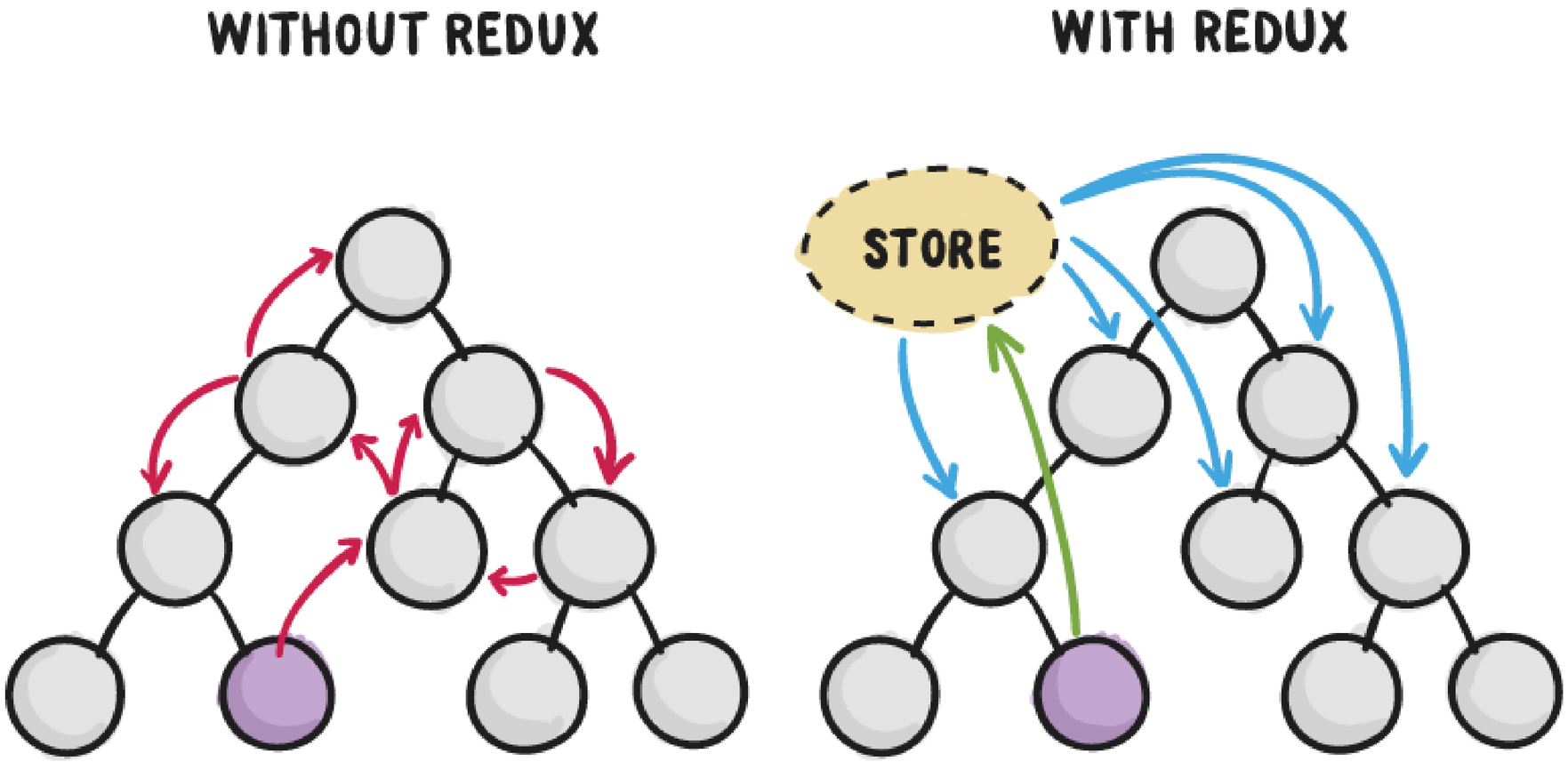
Practice : correction



Redux to manage application state

What problem does Redux solve?

- Extract Business logic in reducer
- **Hierarchical dependencies**



Redux Terms and Concepts : State Management

Let's start by looking at a small React counter component. It tracks a number in component state, and increments the number when a button is clicked:

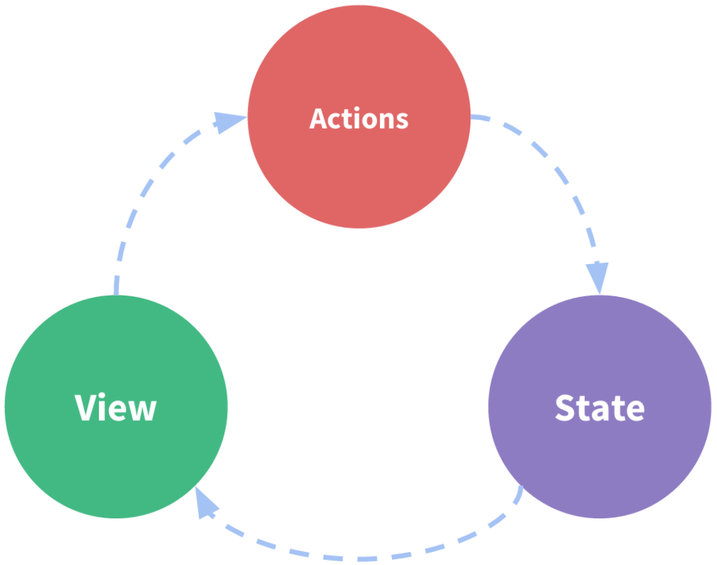
```
function Counter() {
  // State: a counter value
  const [counter, setCounter] = useState(0)

  // Action: code that causes an update to the state when something happens
  const increment = () => {
    setCounter(prevCounter => prevCounter + 1)
  }

  // View: the UI definition
  return (
    <div>
      Value: {counter} <button onClick={increment}>Increment</button>
    </div>
  )
}
```

Extract from Redux Documentation, read more about State Management

Redux Terms and Concepts : State Management

`Counter` is a self-contained app with the following parts:

- The **state**, the source of truth that drives our app;
- The **view**, a declarative description of the UI based on the current state
- The **actions**, the events that occur in the app based on user input, and trigger updates in the state



Extract from Redux Documentation, read more about State Management

Terminology

Action

An **action** is a plain JavaScript object that has a `type` field. **You can think of an action as an event that describes something that happened in the application.**

```
const addTodoAction = {
  type: 'todos/todoAdded',
  payload: 'Buy milk'
}
```

Action Creators

An **action creator** is a function that creates and returns an action object. We typically use these so we don't have to write the action object by hand every time:

```
const addTodo = text => {
  return {
    type: 'todos/todoAdded',
    payload: text
  }
}
```

Extract from Redux Documentation, read more about Actions and Action Creators

Terminology

Reducers

A reducer is a function that receives the current `state` and an `action` object. It returns the new state: `(state, action) => newState`.

```
const initialState = { value: 0 }

function counterReducer(state = initialState, action) {
  // Check to see if the reducer cares about this action
  if (action.type === 'counter/increment') {
    // If so, make a copy of `state`
    return {
      ...state,
      // and update the copy with the new value
      value: state.value + 1
    }
  }
  // otherwise return the existing state unchanged
  return state
}
```

Reducers must always follow some specific rules:

- Only calculate the **new state** value **based on the state and action arguments**
- Not allowed to modify the existing state (**immutability**).
- Not do any asynchronous logic, calculate random values, or cause other "side effects"

Extract from Redux Documentation, read more about Reducers

Terminology

Store

The current Redux application state lives in an object called the **store**. To get the current state use the method `store.getState()`

```
import { createStore } from 'redux'

const store = createStore(counterReducer)

console.log(store.getState())
// {value: 0}
```

Dispatch

The **only way to update the state** is to call `store.dispatch()` and pass in an **action object**:

```
store.dispatch({ type: 'counter/increment' })

console.log(store.getState())
// {value: 1}
```

Extract from Redux Documentation, read more about Store and Dispatch

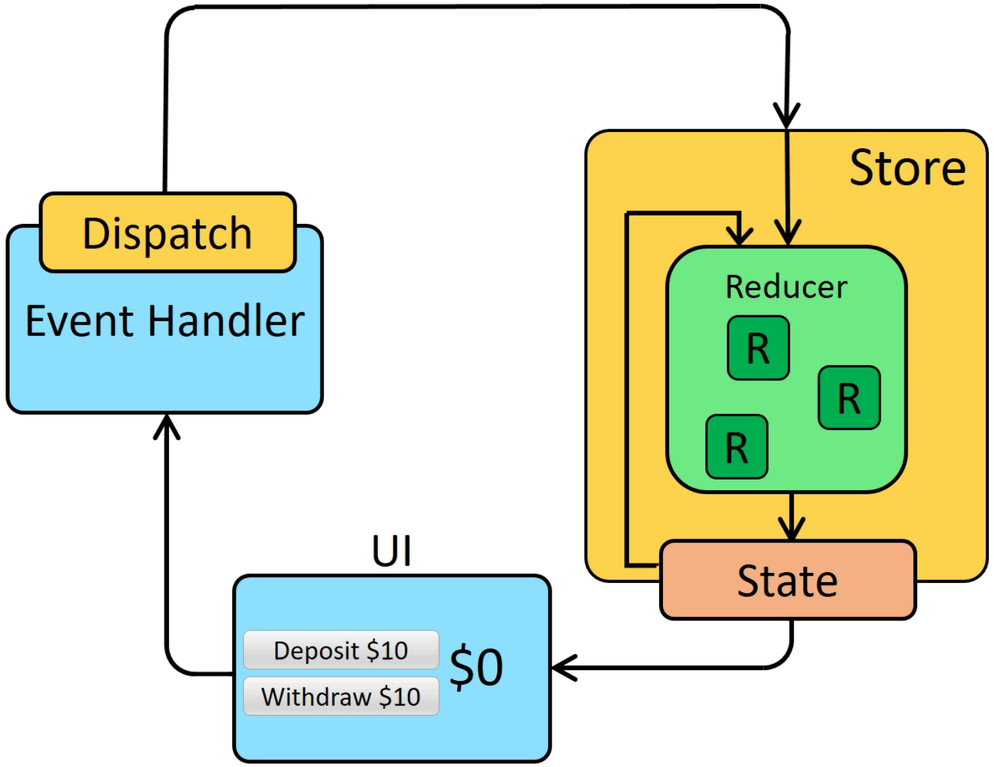
Terminology

Selectors

Selectors are functions that know how to extract specific pieces of information from a store state value.
Naming: *select* + *functionalName*

```
const selectCounterValue = state => state.value

const value = selectCounterValue(store.getState())
console.log(value)
// 2
```

Redux Application Data Flow ?



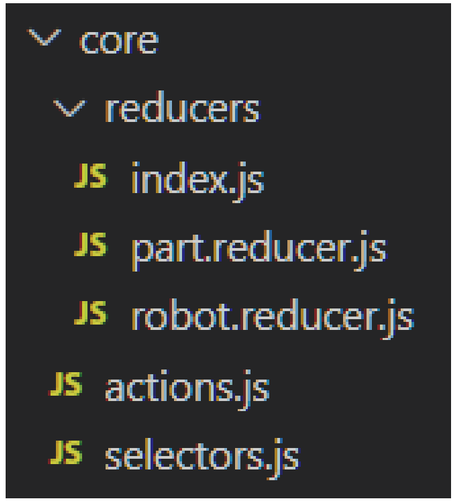
Extract from Redux Documentation, read more about Selector

Redux in practice

- Install reduce components

```
npm install redux
npm install react-redux
```

- Create actions and reducers into dedicated files and directories
- Use Global reducers merging several reducers



`index.js`

```
import { combineReducers } from 'redux';
import robotReducer from './robot.reducer';
import partReducer from './part.reducer';

const globalReducer = combineReducers({
  robotReducer: robotReducer,
  partReducer: partReducer
});

export default globalReducer;
```

`robot.reducer.js`

```
const robotReducer= (state, action) => {
    switch (action.type) {
        case 'UPDATE_SELECTED_ROBOT':
            return {
              ...state
              selectedRobotId: action.payload
            };
    default:
      return state;
    }
}
```

Redux in practice

Passing the Store with Provider

```
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'

import App from './App'

const store = createStore(globalReducer)

ReactDOM.render(
  // Render a `<Provider>` around the entire `<App>`,
  // and pass the Redux store to as a prop
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
)
```

Redux in practice

Hooks

- `useSelector` automatically **subscribes** to the Redux store for us! If the **value** returned by the selector **changes** from the last time it ran, useSelector will **force our component to re-render** with the new data.
- `useDispatch` hook gives us the store's dispatch method as its result.

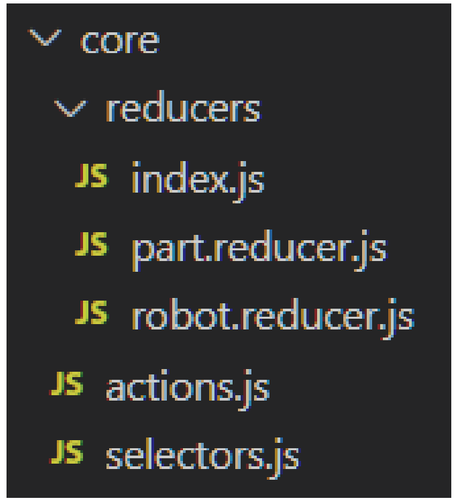
Extract from Redux Documentation, read more about UI and React

```
const selectRobots = state => state.robots

const RobotList = () => {
  const robots = useSelector(selectRobots)
  const dispatch = useDispatch()

  const onRobotSelected = (selectedRobotId) => dispatch({
    type: "UPDATE_SELECTED_ROBOT",
    payload: selectedRobotId
  })

  // since `robots` is an array, we can loop over it
  const renderedListItems = robots.map(robot => {
    return <Robot
      key={robot.id}
      robot={robot}
      onRobotSelected={onRobotSelected}
    />
  })

  return <ul className="robot-list">
    {renderedListItems}
  </ul>
}
```

Practice

Create Action Creators, Reducers and Selectors

- Create 4 action creators to save:
  - the selected robot id and part id
  - the list of robots and the list of parts
- Create 1 reducer *robotReducer* in charge of processing the selected robot id and the robots
- Create 1 reducer *partReducer* in charge of processing the selected part id and the parts
- Create selectors to get robots, parts, selectedParts, etc...



Practice

Adapt code to use Redux

- Dispatch fetching response (`/parts` and `/robots`) from `App` or `RobotList` or `PartList`
- Dispatch selected robot in the `Robot` component
- Subscribe to store and update list of parts in the `PartList` component
- Dispatch selected part in the `Part` component
- Subscribe to store and update part in the `PartDetail` component

Best Practice

- State structure : avoid redundant state, duplication, deeply nested state
- Separate responsibilities
- Re-use component
- Prefer simplicity than complexity

Industrialization

- Prevent bugs
  - Typing: TS
  - Tests: Jest
- Use popular library for utils : Community is better
- Readability
  - Comments
  - Naming
- Be attentive with UX experience
  - Empty page
  - Very long loading

Coding tips

- Ternary operator `` `predicat ? value1 : value2 ` ``
- Optional chaining operator `` `?.` ``
- Nullish coalescing `` `??` ``

References

Documentation:

- React : https://react.dev/learn & https://reactjs.org/
- Redux : https://redux.js.org/tutorials/essentials/part-1-overview-concepts

Credits

- Jacques Saraydaryan, courses and tutorials
- Hubert Sablonnière, Le Web, ses frameworks et ses standards : déconstruire pour mieux (re?)construie

Co-authors

- Jacques Saraydaryan
- Loïc Le Goff