Modern web & React

Contents

- Modern Web
- React 소개
- JSX
- Props & State
- LifeCycle API
- Todo list 만들기

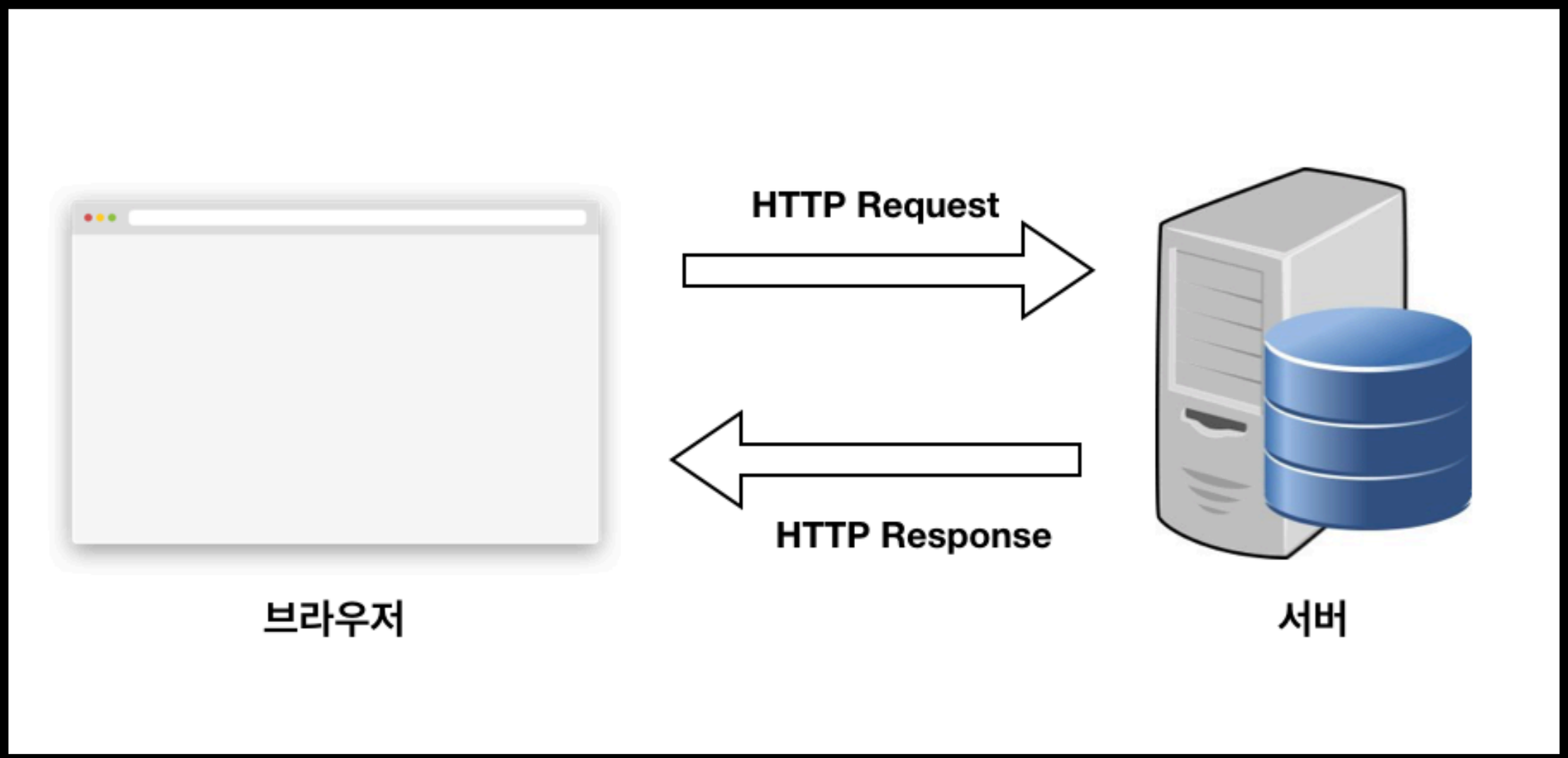
Modern Web

- *Web*
- *HTTP*
- *Browser & DOM*
- *Moderen web paradigm*

Modern Web

Web

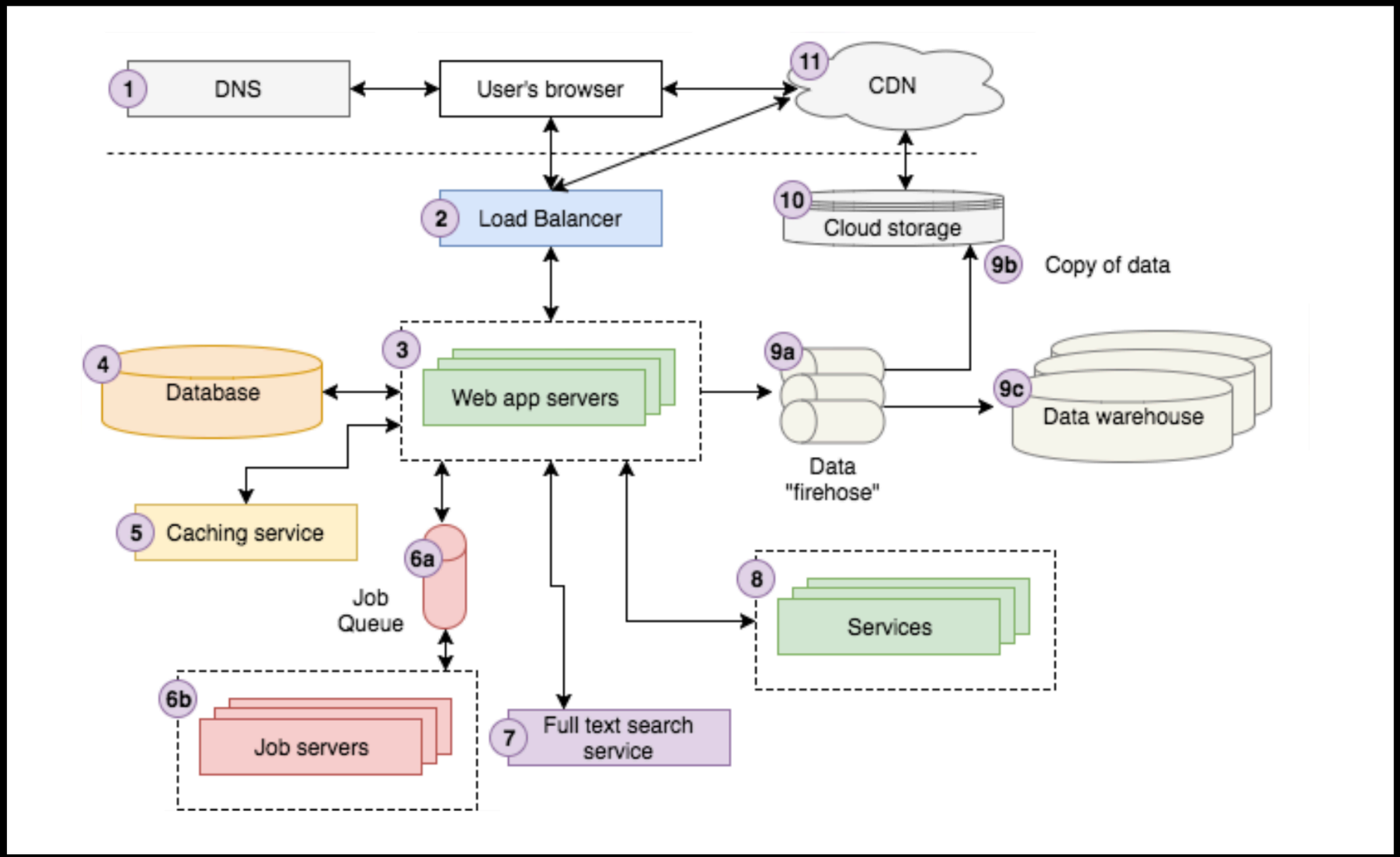
- 웹 아키텍처



Modern Web

Web

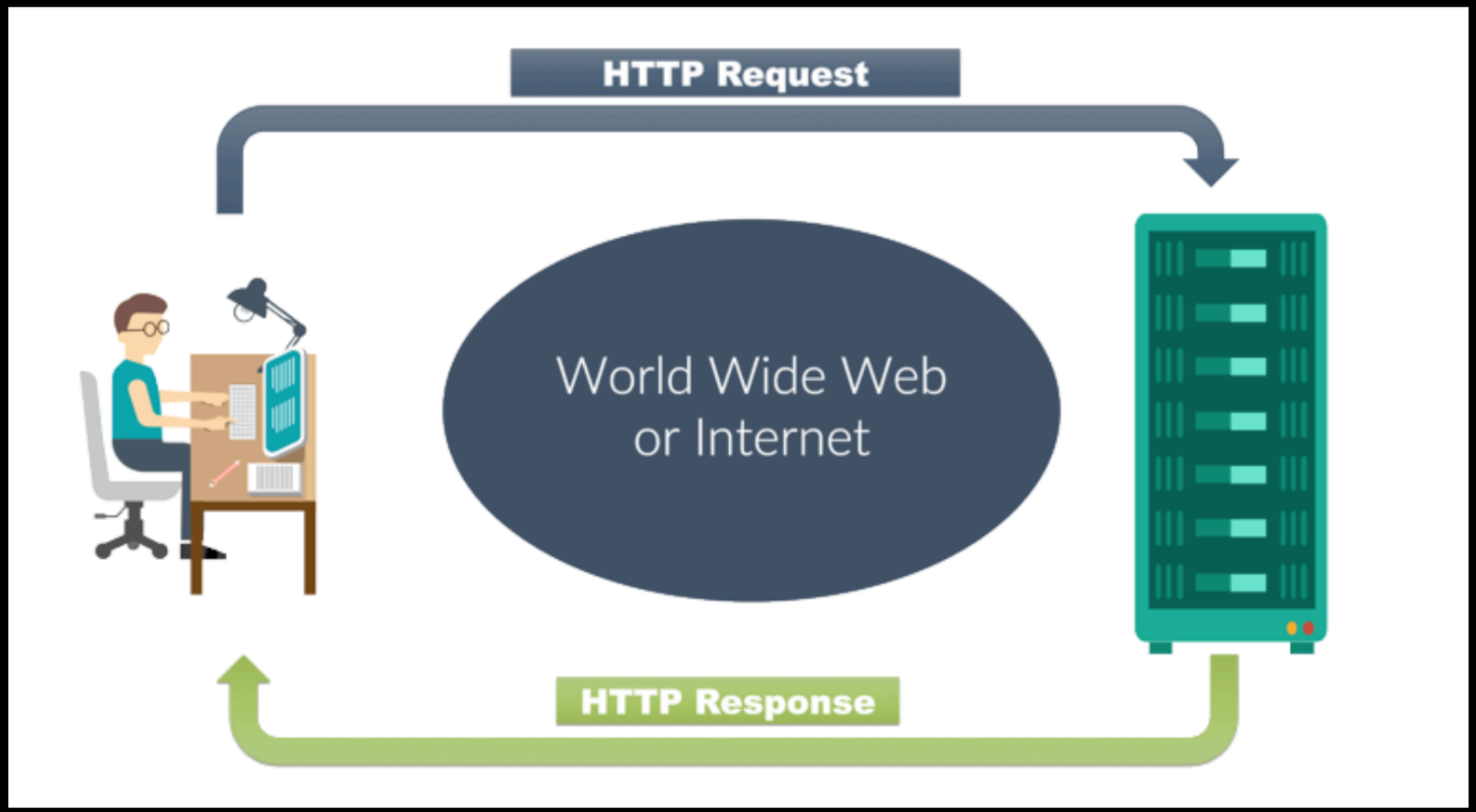
- 웹 아키텍처



Modern Web

HTTP

- HTTP통신이란?
  인터넷 통신을 위해 사용되는 프로톨로서 요청-응답이 하나의 통신구조로 이루어진 메시지기반 모델의 통신방법으로 웹 브라우저에서 HTML을 주고 받기 위해 설계 되었으나 최근 SPA, IoT 등에서도 통신 표준으로 사용하기도 한다.



React

Browser & DOM

- 브라우저 기본구조

  1. 사용자 인터페이스 - 요청한 페이지를 보여주는 창을 제외한 나머지 모든 부분

  2. 브라우저 엔진 - 사용자 인터페이스와 렌더링 엔진 사이의 동작을 제어

  3. 렌더링 엔진 - 요청한 콘텐츠를 표시 (요청 시 HTML과 CSS를 파싱하여 화면에 표시)

  4. 통신 - 네트워크호출(HTTP 등)에 사용 (플랫폼 독립적인 인터페이스이며 각 플랫폼 하부에서 실행)

  5. UI 백엔드 - 콤보 박스와 창 같은 기본적인 장치를 그림 (플랫폼에서 명시하지 않은 일반적인 인터페이스로서, OS 사용자 인터페이스 체계를 사용)

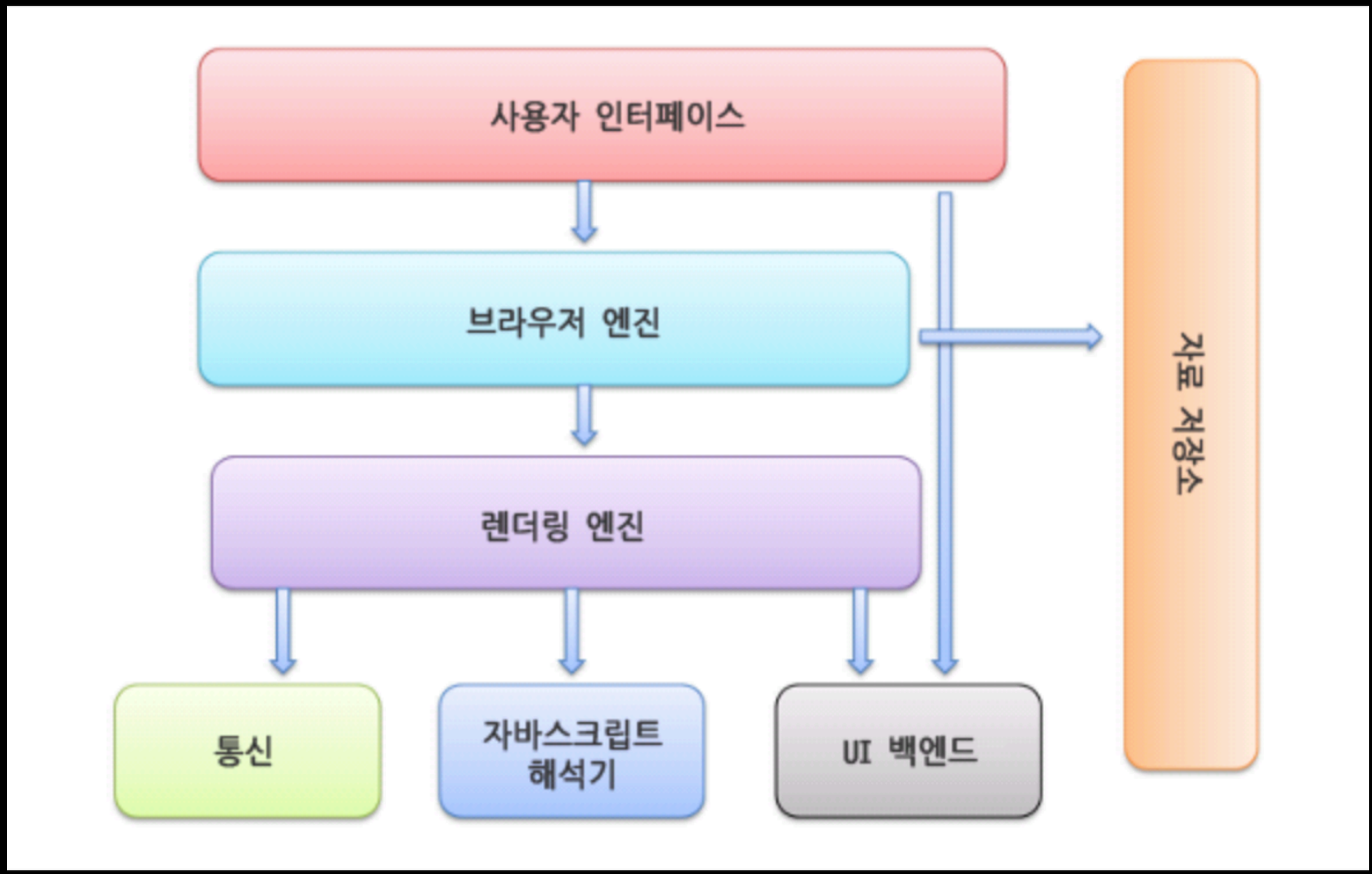
  6. 자바스크립트 해석기 - 자바스크립트 코드를 해석하고 실행

  7. 자료 저장소 - 이 부분은 자료를 저장하는 계층으로 쿠키를 저장하는 것과 같이 모든 종류의 자원을 하드 디스크에 저장할 필요가 있음 (HTML5 명세에는 브라우저가 지원하는 '웹 데이터 베이스'가 정의되어 있다)

React

Browser & DOM

- 브라우저 기본구조



Modern Web

Moderen web paradigm

- SPA
  Single Page Application의 약자로서 SPA는 현대 웹개발의 주요 기법으로 트랜드가 되는 차세대 패러다임이다.

  SPA 이전의 웹은 요청 시 마다 새로고침이 일어나며 서버로부터 요청결과를 전달받아 해석한뒤 화면에 전체를 렌더링 하게된다. 그러나 이러한 방식의 웹은 다이나믹하고 애플리케이션으로서의 웹환경을 표현하기에 한계와 많은 리스크를 가지고 있었다 이를 해소하기위해 캐싱과 압축이라는 방식과 함께 브라우저에서 동작하는 소프트웨어를 만들기 위한 유일한 수단이 자바스크립트(또는 자바스크립트 프레임웍)가 각광받으며 엄청난 발전을 했다.

Modern Web

Moderen web paradigm

- Javascript Framework

React

- *VanillaJS & jQuery & AngularJS*
- *Virtual DOM*

VanillaJS & jQuery & AngularJS

- VanillaJS & jQuery

```
document.getElementById('myId').appendChild(myNewNode) // javascript
$('#myId').append(myNewNode) // jquery
```

- Angular.js

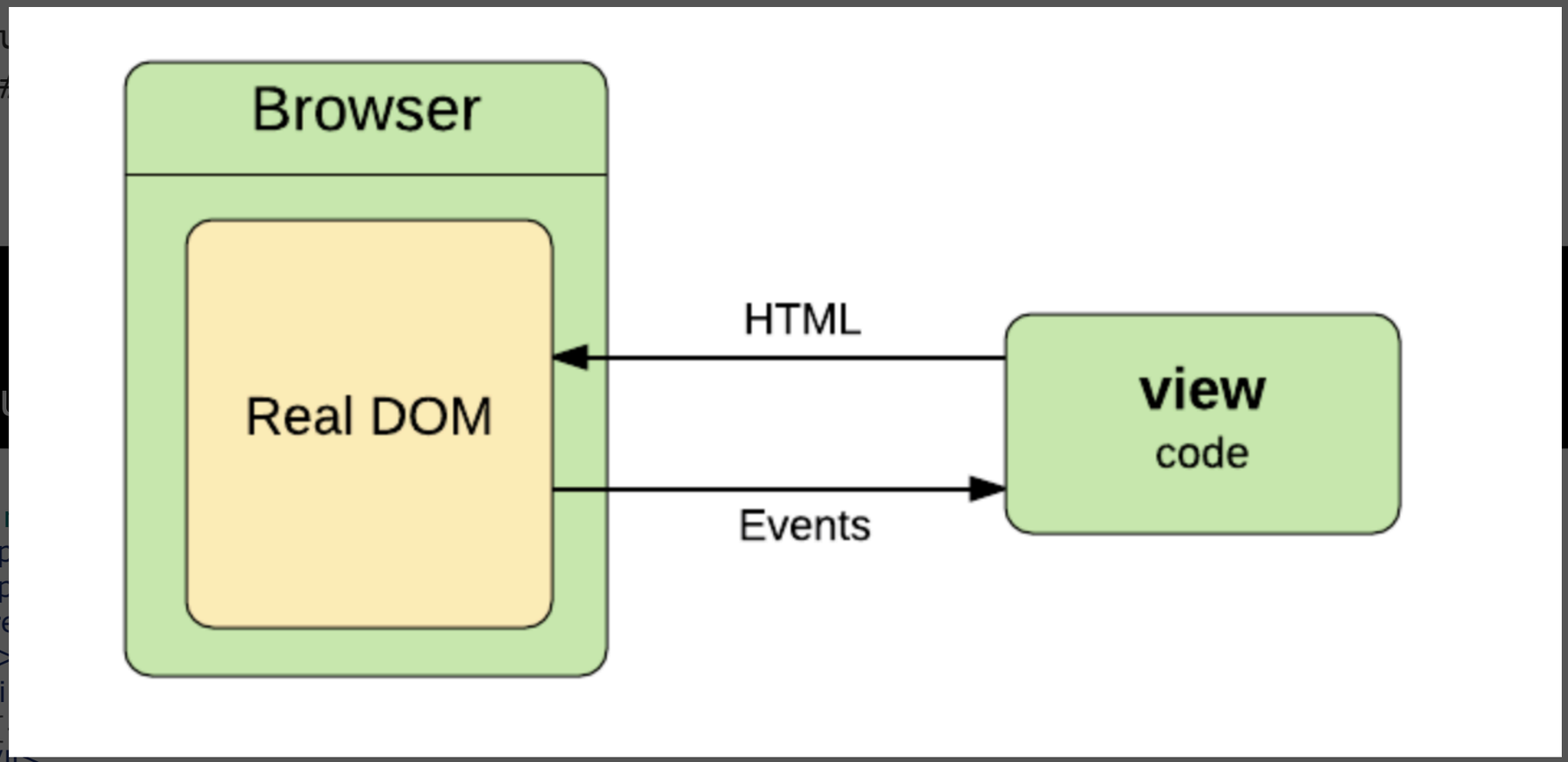
```
<div ng-init="list = ['Chrome', 'Safari', 'Firefox', 'IE'] ">
  <input ng-model="list" ng-list> <br>
  <input ng-model="list" ng-list> <br>
  <pre>list={{list}}</pre> <br>
  <ol>
    <li ng-repeat="item in list">
      {{item}}
    </li>
  </ol>
</div>
```

React

VanillaJS & jQuery & AngularJS

- VanillaJS & jQuery

```
docu
$('#
```



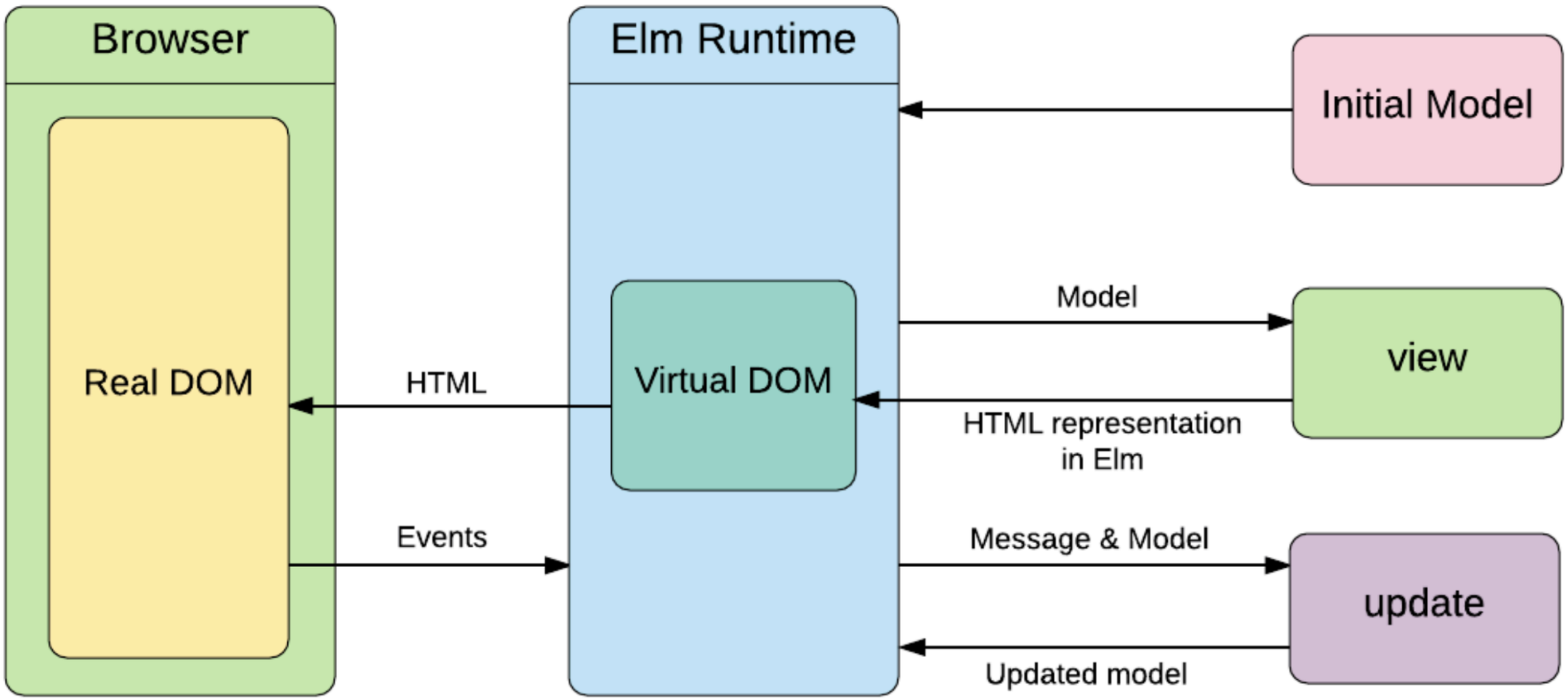
- Angu

```
<div
  <inp
  <inp
  <pre
  <ol>
    <li
      {
    </li>
  </ol>
</div>
```

React

Virtual DOM

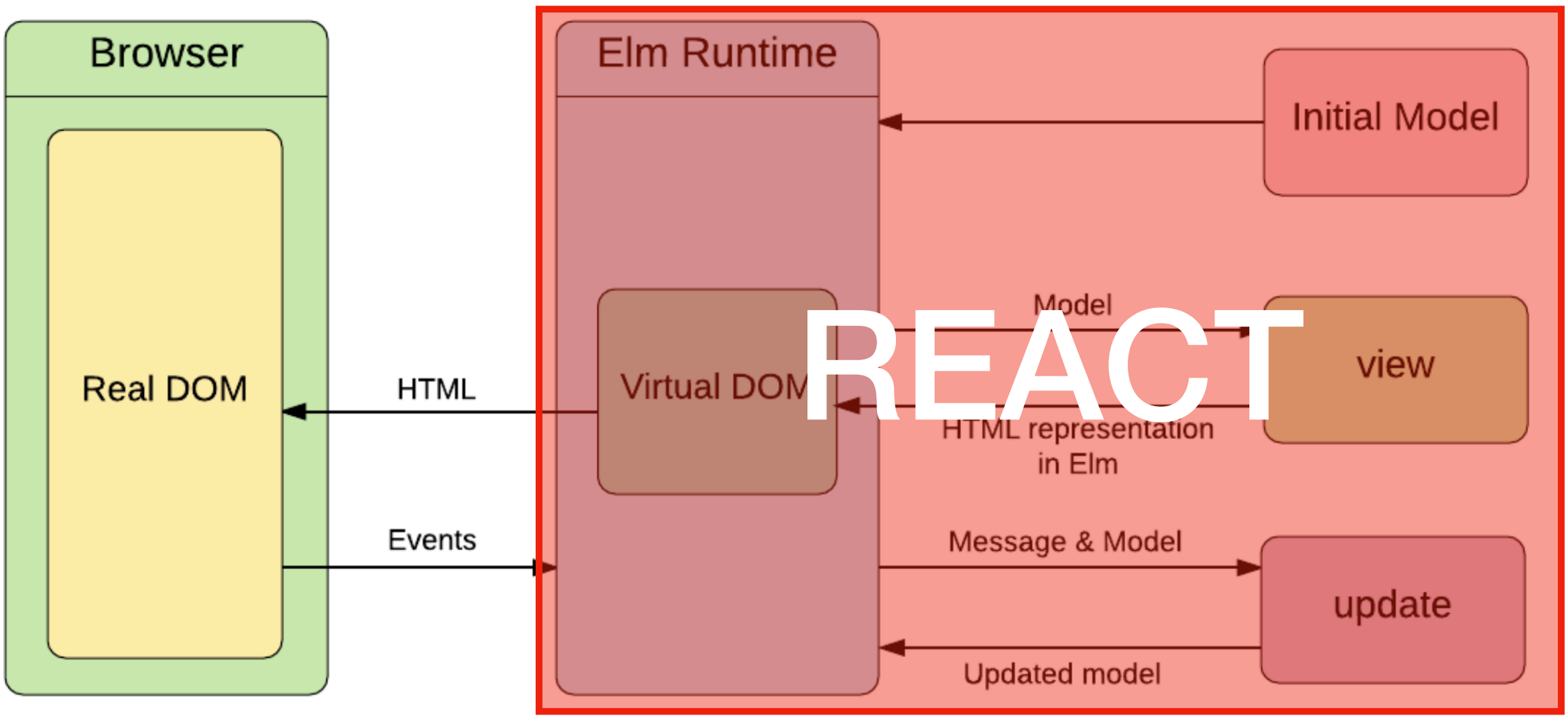
- Virtual DOM이란?



React

Virtual DOM

- Virtual DOM이란?



Modern Javascript

- *JSX*
- *Babel*
- *Webpack*

Modern Javascript

Syntax

- 이 문법은 JSX라고 부르며, 자바스크립트의 문법 확장이다. JSX를 리액트와 함께 사용하여 UI가 실제로 어떻게 보일지 설명하는 걸 권장하며 JSX는 템플릿 언어처럼 보일 수 있지만, 자바스크립트를 기반으로 하고 있다.

```
const element = <h1>Hello, world!</h1>;
```

```
const element = <img src={user.avatarUrl} />;
```

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

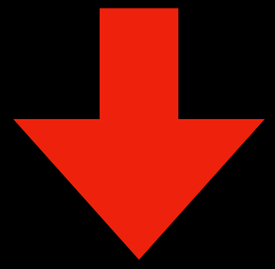
https://reactjs-kr.firebaseapp.com/docs/introducing-jsx.html

Modern Javascript

Babel

- ES6/ES7 코드를 ECMAScript5 코드로 transpiling 하기 위한 도구로서 Babel은 다양한 작은 모듈들로 구성되어 있다. Babel 다양한 모듈을 담는 일종의 상자 역할을 하며 코드를 컴파일 하기 위해 작은 모듈들(ex:presets)을 사용한다.

```
// ES6(Arrow Function) + ES7(Exponentiation operator)
[1, 2, 3].map(n => n ** n);
```



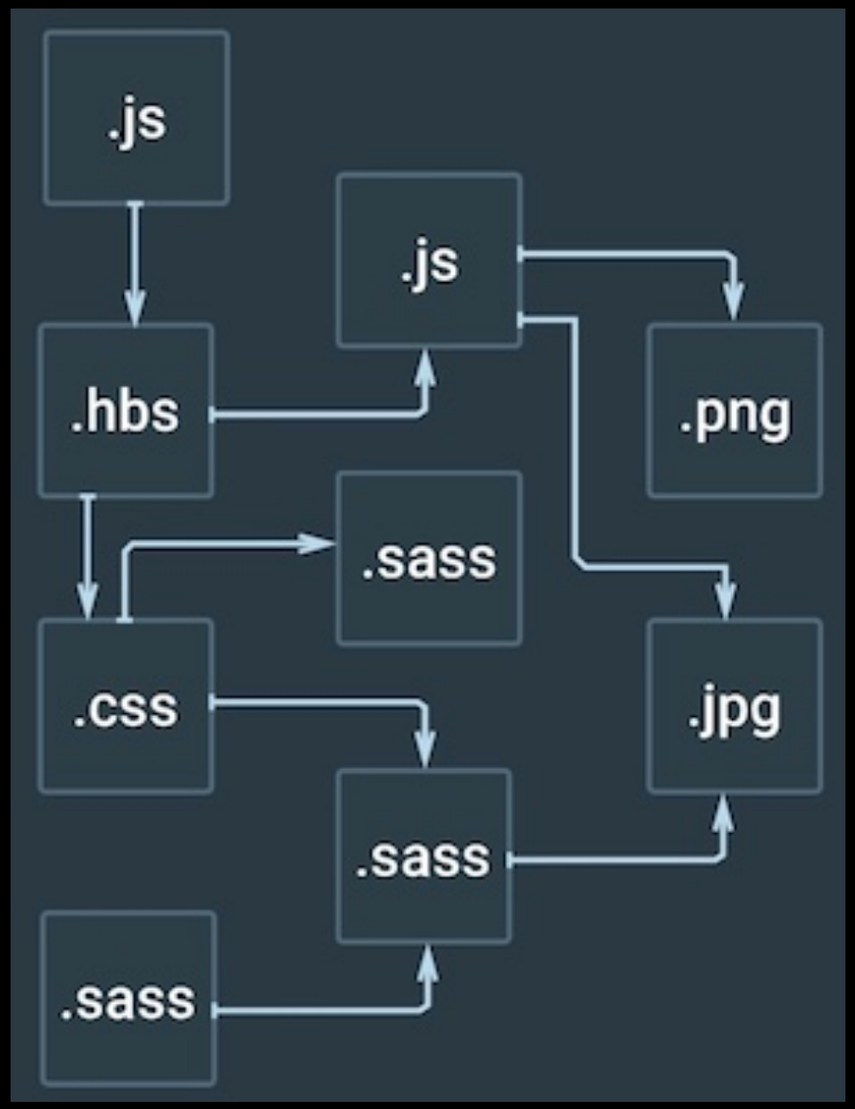
```
// ES5
"use strict";

[1, 2, 3].map(function (n) {
  return Math.pow(n, n);
});
```

Modern Javascript

Webpack

- 웹팩은 기본적으로 모듈 번들러다. 의존성 그래프에서 엔트리로 그래프의 시작점을 설정하면 웹팩은 모든 자원을 모듈로 로딩한 후 아웃풋으로 묶어준다. 로더로 각 모듈별로 바벨, 사스변환 등의 처리하고 이 결과를 플러그인이 받아 난독화, 텍스트 추출 등의 추가 작업을 한다.



Props & State

- 상태
- *Props*
- *State*

Props & State

상태

- 리액트에서 다루는 데이터는 크게 Props와 State 두 가지로 나뉜다. 이런 데이터들을 상태라고 표현하며 정해진 규칙에 의해서 단방향으로만 흐르도록 유도하여 좋은 퍼포먼스를 보여준다. 또한 상태는 리액트에서의 컴포넌트가 다시 렌더링되는 시점과 직접적인 연관을 가지고있다.

Props & State

Props

- **개념 :** 부모 컴포넌트로부터 속성형태로 받아오는 데이터를 말하며 원시값 뿐만 아니라 객체, 함수 JSX 등 모든 데이터를 Props를 통해 전달 할 수 있다. Props로 받아오는 데이터가 변경되면 해당 데이터를 받고있는 컴포넌트는 자동으로 업데이트 된다.

```
class App extends Component {
  render() {
    return (
      <MyName name="리액트" />
    );
  }
}
```

```
class MyName extends Component {
  render() {
    return (
      <div>
        안녕하세요! 제 이름은 <b>{this.props.name}</b>
      </div>
    );
  }
}
```

Props & State

Props

- **defaultProps:** 부모로부터 전달 받을 데이터가 없거나 초기 데이터를 설정하기 위한 문법으로 마치 변수를 미리 선언하듯 사용할 수 있다.

```
static defaultProps = {
  name: '기본이름'
}
```

```
MyName.defaultProps = {
  name: '기본이름'
};
```

Props & State

State

- state는 props와 state가 선언된 컴포넌트에서만 사용 가능한 상태이다. props와 차이점은 컴포넌트가 자체적으로 상태를 가질 수 있다는 점과 클래스형 컴포넌트에서만 사용가능하다. state는 setState라는 함수를 통해서만 변경이 가능하며 setState되는 시점에 컴포넌트가 업데이트를 시작한다. (16.8버전 이후 부터는 함수형 컴포넌트에서도 hook이라는 개념을 사용하여 상태를 직접적으로 관리할 수 있다.)

```
class Counter extends Component {
  state = {
    number: 0
  }
```

```
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      number: 0
    }
  }
```

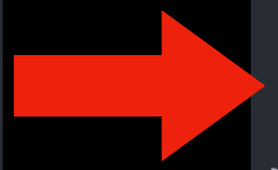
Props & State

State

- **setState:** state와 props 모든 상태를 변경하기 위해선 해당 함수를 사용해야만 한다.

```
this.setState({
  foo: {
    foobar: 2
  }
})
```

```
this.setState({
  number: this.state.number + 1
});
```



```
this.setState(
  ({ number }) => ({
    number: number + 1
  })
);
```

LifeCycle API

- 생성
- 업데이트
- 제거

LifeCycle API

생성

- **constructor:** 컴포넌트의 생성자 함수이며 컴포넌트가 새로 생성될 때 호출된다.

```
constructor(props) {
  super(props);
}
```

- **componentDidMount:** 컴포넌트가 화면에 나타난 후 실행되는 함수 즉 렌더링 이후 호출되는 함수로서 http통신, 차트데이터 반영 등을 해당 함수에서 사용한다.

```
componentWillMount() {

}
```

LifeCycle API

업데이트

- **static getDerivedStateFromProps:** Props로 받아온 데이터를 State로 동기화 해야하는 경우 사용한다.

```
static getDerivedStateFromProps(nextProps, prevState) {
  // setState가 아닌 props가 바뀔 때 설정하고 싶은 state값을 리턴하는 형태


  /*
  if (nextProps.value !== prevState.value) {
    return { value: nextProps.value };
  }
  return null; // null 을 리턴하면 따로 업데이트 할 것은 없다라는 의미
  */
}
```

- **shouldComponentUpdate:** 상태가 변경될 때 해당 함수의 리턴값(true || false)으로 컴포넌트의 업데이트 여부를 결정할 수 있다. 기본 return값은 true다.

```
shouldComponentUpdate(nextProps, nextState) {
  // return false
  // return this.props.num < nextProps.num
  return true;
}
```

LifeCycle API

업데이트

- **getSnapshotBeforeUpdate:** 이 API를 통해서, DOM 변화가 일어나기 직전의 DOM 상태를 가져오고, 여기서 리턴하는 값은 componentDidUpdate 에서 3번째 파라미터로 받아올 수 있게 된다.
  이 API 가 발생하는 시점은 다음과 같습니다.
  - render()
  - getSnapshotBeforeUpdate()
  - 실제 DOM 에 변화 발생
  - componentDidUpdate

```
getSnapshotBeforeUpdate(prevProps, prevState) {
  if (prevState.array !== this.state.array) {
    const {
      scrollTop, scrollHeight
    } = this.list;
    return {
      scrollTop, scrollHeight
    };
  }
}
```

```
componentDidUpdate(prevProps, prevState, snapshot) {
  if (snapshot) {
    const { scrollTop } = this.list;
    if (scrollTop !== snapshot.scrollTop) return;
    const diff = this.list.scrollHeight - snapshot.scrollHeight;
    this.list.scrollTop += diff;
  }
}
```

LifeCycle API

삭제

- **componentWillUnmount:** 컴포넌트가 사라지기 직전에 호출되며, 등록한 이벤트를 제거하거나 clearTimeout, 라이브러리 discope등을 해당 함수에서 사용한다.

```
componentWillUnmount() {
  // 이벤트, setTimeout, 외부 라이브러리 인스턴스 제거
}
```