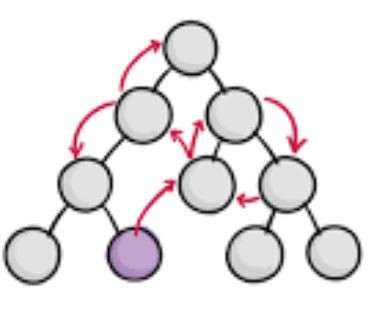
**1. mobx의 사용법을 예제소스를 통해 설명하시오.**

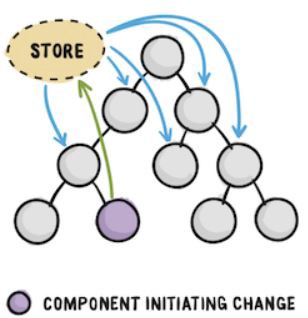
리액트로 프로젝트를 진행하게 되면, state와 props 만으로는 데이터의 관리가 매우 어렵다는 것을 느끼게 된다. 대표적인 어려움으로는 밑의 그림과 같은 상황을 둘 수 있다.



최상단의 component에서 state를 두고, 하위 component에 props를 뿌리는 상황이다. component수 자체가 적다면 큰 문제가 없을 수 있다. 하지만 그 수가 많아지고 관리해야하는 state와 props의 수가 많아진다면, 나중에 가서는 매우 혼란스러워 질 것이다. 이런 상황에서 다른 사람과 같이 프로젝트를 진행하게 된다면? 더 큰 어려움이 있을 수 있다.

물론, 이러한 상태 관리는 리액트 만으로도 할 수 있다. 리액트는 Cotext API 라는 것을 통해 다른 라이브러리 없이도 전역 상태를 관리할 수 있다.

리액트의 Context API, redux, 혹은 mobx을 사용하게 되면 위에 있는 복잡한 state의 관리는 다음과 같은 그림처럼 간편하게 변경될 수 있다.



하나 혹은 다수의 store(저장소)를 두고 그곳에서 state를 저장해 둔다. 이제 props를 전달할 때 등 데이터의 이동의 절차가 매우 줄어들게 된다.

**Mobx**

기본적으로 객체지향 느낌이 강하며 Component와 State를 연결하는(Redux와 달리) 번잡한 보일러플레이트 코드들을 데코레이터 제공으로 깔끔하게 해결합니다.

**Mobx의 장점**

**객체지향적**

보다 객체지향적입니다 ES6에서 추가된 Class를 이름뿐인 Class가 아니라 객체지향적으로 사용하고 개발하는 것을 권장하고 있습니다.

**서버개발자들에게 친숙한 아키텍쳐**

Java Spring Framework와 유사한 아키텍쳐구조를 지향하고 있어 서버개발자들에게 보다 친숙하고 낮은 러닝커브를 제공, 장점을 그대로 적용할 수 있습니다.

**Decorator**

데코레이터를 제공하기 때문에 Redux를 사용할 때 React Component와 state를 연결 하기위한 mapStateToProps, Redux action을 연결을 위한 mapDispatchToProps 그리고 bindActionCreators…. 등등의 보일러플레이트 코드가 사라지고 데코레이터가 처리하기 때문에 너무나도 깔끔한 코드가 생성됩니다.

Redux로 개발 해보신 분이라면 느끼시겠지만 보일러플레이트 코드들의 양 만만하지 않고 또 그런 코드들을 작성하기 위해서는 어느정도 학습이 동반되어야 합니다. Redux가 어렵다가 아니라 React가 Vue보다 어렵다고 하는 이유에도 이부분도 한 몫하는 것 같습니다.

**캡슐화**

Mobx Configuration 설정으로 State를 오직 메소드를 통하여 변경할 수 있도록 Private하게 관리 할 수 있습니다.

**불변성 유지를 위한 노력이 불필요**

State의 불변성을 유지하기 위해서 번잡스러운 코드나 ImmutableJs같은 라이브러리를 따로 사용할 필요가 없습니다. 이것이 왜 장점이 되냐 하면 불변성을 유지하면 서 State를 변경하는 코드는 Object가 Depth가 깊어지게 되면 코드의 가독성이 매우 떨어집니다. 그래서 ImmutableJs 라이브러리를 사용하게 되는데 Redux와 같이 사용하게 될 경우 여러가지 설정이 필요하고 추가적인 라이브러리도 필요할 뿐 만 아니라 추가적인 학습도 동반 되어야 합니다.

**MobX 의 주요 개념들**

**1. Observable State (관찰 받고 있는 상태)**

MobX 를 사용하고 있는 앱의 상태는 Observable 합니다. Mobx에서 Rerendering 대상이 되는 state(상태, 값)를 관찰 대상(observable value)라고 칭하며 @observable 데코레이터로 지정한 State는 관찰대상으로 지정되고 그 State는 값이 변경될 때 마다 Rerendering됩니다.

**2. Reactions (반응) -addEventListener(“값변경”,function(){}) – 느낌**

Reactions 는 Computed Value 와 비슷한데요, Computed Value 의 경우는 우리가 특정 값을 연산해야 될 때 에만 처리가 되는 반면에, Reactions 은, 값이 바뀜에 따라 해야 할 일을 정하는 것을 의미합니다. 예를 들어서 Observable State 의 내부의 값이 바뀔 때, 우리가 console.log('ㅇㅇㅇ가 바뀌었어!') 라고 호출해 줄 수 있습니다.

**3. Computed Value (연산된 값)**

연산된 값은, 기존의 상태값과 다른 연산된 값에 기반하여 만들어질 수 있는 값입니다. 이는 주로 성능 최적화를 위하여 많이 사용됩니다. 어떤 값을 연산해야 할 때, 연산에 기반되는 값이 바뀔때만 새로 연산하게 하고, 바뀌지 않았다면 그냥 기존의 값을 사용 할 수 있게 해줍니다.

이를 이해하기 위해 간단한 예시를 들어보겠습니다. 우리가 편의점에서 800원짜리 물을 네병 샀는데 이게 얼마나오지? 하고 체크하는 함수 total() 이라는 함수가 있다고 가정하겠습니다. 우리가 처음 머릿속으로 계산할때 암산으로 4 \* 8 에 32! 라면서 3,200 원이군, 하고 간단히 계산을 하겠죠. 잠시 후에 친구가 그거 다 얼마냐고 또 물어봅니다. 이 때 우리는 머릿속에서 별 생각안하고 응 3,200원이야 라고 말합니다. 친구가, 나도 한병 사줘! 하면 이때 다시 우리는 무의식중에 800원을 더해서 우리가 내야 할 돈이 4,000원인걸 연산합니다.

**4. Actions (액션; 행동)**

액션은, 상태에 변화를 일으키는것을 말합니다. 만약에 Observable State 에 변화를 일으키는 코드를 호출한다? 이것은 하나의 액션입니다. - 리덕스에서의 액션과 달리 따로 객체형태로 만들지는 않습니다.

import { observable, reaction, computed, autorun } from 'mobx';

**observable**

observable 함수는 Observable State 를 만들어줍니다.

import { observable, reaction, computed, autorun } from 'mobx';

// \*\*\*\* Observable State 만들기

const calculator = observable({

a: 1,

b: 2

});

이렇게 Observable State 를 만들고나면 MobX 가 이 객체를 "관찰 할 수" 있어서 변화가 일어나면 바로 탐지해낼수있습니다.

**reaction**

특정 값이 바뀔 때 어떤 작업을 하고 싶다면 reaction 함수를 사용합니다.

한번 a 나 b 가 바뀔 때 console.log 로 바뀌었다고 알려주도록 코드를 작성해보겠습니다.

// \*\*\*\* 특정 값이 바뀔 때 특정 작업 하기!

reaction(

() => calculator.a,

(value, reaction) => {

console.log(`a 값이 ${value} 로 바뀌었네요!`);

}

);

reaction(

() => calculator.b,

value => {

console.log(`b 값이 ${value} 로 바뀌었네요!`);

}

);

calculator.a = 10;

calculator.b = 20;

콘솔쪽을 보시면 다음과 같이 나타날 것입니다.

a 값이 10 로 바뀌었네요!

b 값이 20 로 바뀌었네요!

**computed**

computed 함수는 연산된 값을 사용해야 할 때 사용됩니다. 특징은, 이 값을 조회할 때 마다 특정 작업을 처리하는것이 아니라, 이 값에서 의존하는 값이 바뀔 때 미리 값을 계산해놓고 조회 할 때는 캐싱된 데이터를 사용한다는 점 입니다.

// \*\*\*\* 특정 값이 바뀔 때 특정 작업 하기!

reaction(

() => calculator.a,

(value, reaction) => {

console.log(`a 값이 ${value} 로 바뀌었네요!`);

}

);

reaction(

() => calculator.b,

value => {

console.log(`b 값이 ${value} 로 바뀌었네요!`);

}

);

calculator.a = 10;

calculator.b = 20;

// \*\*\*\* computed 로 특정 값 캐싱

const sum = computed(() => {

console.log('계산중이예요!');

return calculator.a + calculator.b;

});

sum.observe(() => calculator.a); // a 값을 주시

sum.observe(() => calculator.b); // b 값을 주시

calculator.a = 10;

calculator.b = 20;

//\*\*\*\* 여러번 조회해도 computed 안의 함수를 다시 호출하지 않지만..

console.log(sum.value);

console.log(sum.value);

// 내부의 값이 바뀌면 다시 호출 함

calculator.a = 20;

console.log(sum.value);

결과를 확인해볼까요?

계산중이예요!   
a 값이 10 로 바뀌었네요!   
계산중이예요!   
b 값이 20 로 바뀌었네요!   
계산중이예요!   
30  
30  
a 값이 20 로 바뀌었네요!   
계산중이예요!   
40

**autorun**

autorun 은 reaction 이나 computed 의 observe 대신에 사용 될 수 있는데, autorun 으로 전달해주는 함수에서 사용되는 값이 있으면 자동으로 그 값을 주시하여 그 값이 바뀔 때 마다 함수가 주시되도록 해줍니다. 여기서 만약에 computed 로 만든 값의 .get() 함수를 호출해주면, 하나하나 observe 해주지 않아도 됩니다.

import { observable, reaction, computed, autorun } from 'mobx';

// Observable State 만들기

const calculator = observable({

a: 1,

b: 2

});

// computed 로 특정 값 캐싱

const sum = computed(() => {

console.log('계산중이예요!');

return calculator.a + calculator.b;

});

// \*\*\*\* autorun 은 함수 내에서 조회하는 값을 자동으로 주시함

autorun(() => console.log(`a 값이 ${calculator.a} 로 바뀌었네요!`));

autorun(() => console.log(`b 값이 ${calculator.b} 로 바뀌었네요!`));

autorun(() => sum.get()); // su

calculator.a = 10;

calculator.b = 20;

// 여러번 조회해도 computed 안의 함수를 다시 호출하지 않지만..

console.log(sum.value);

console.log(sum.value);

calculator.a = 20;

// 내부의 값이 바뀌면 다시 호출 함

console.log(sum.value);

결과를 확인해볼까요?

a 값이 1 로 바뀌었네요!   
b 값이 2 로 바뀌었네요!   
계산중이예요!   
a 값이 10 로 바뀌었네요!   
계산중이예요!   
b 값이 20 로 바뀌었네요!   
계산중이예요!   
30  
30  
a 값이 20 로 바뀌었네요!   
계산중이예요!   
40

**class 문법을 사용해서 조금 더 깔끔하게**

ES6 의 class 문법을 사용하면 조금 더 깔끔하게 코드를 작성 할 수 있습니다. 기존의 코드를 날리고, 이번엔 편의점 장바구니를 만들어보겠습니다. class 로 장바구니를 구현 후, decorate 함수를 통하여 MobX 를 적용해주겠습니다.

import { decorate, observable, computed, autorun } from 'mobx';

class GS25 {

basket = [];

get total() {

console.log('계산중입니다..!');

// Reduce 함수로 배열 내부의 객체의 price 총합 계산

// https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global\_Objects/Array/Reduce

return this.basket.reduce((prev, curr) => prev + curr.price, 0);

}

select(name, price) {

this.basket.push({ name, price });

}

}

// decorate 를 통해서 각 값에 MobX 함수 적용

decorate(GS25, {

basket: observable,

total: computed,

});

const gs25 = new GS25();

autorun(() => gs25.total);

gs25.select('물', 800);

console.log(gs25.total);

gs25.select('물', 800);

console.log(gs25.total);

gs25.select('포카칩', 1500);

console.log(gs25.total);

결과는 다음과 같습니다.

계산중입니다..!   
계산중입니다..!   
800  
계산중입니다..!   
1600  
계산중입니다..!   
3100

**action**

우리가 이전에 상태에 변화를 일으키는것을 action 이라고 부른다고 언급했습니다. 만약에, 이 변화를 일으키는 함수에 MobX 의 action 을 적용하면 무엇을 할 수 있는지 알아보겠습니다.

우선, 코드 상단에서 action 함수를 불러오고, decorate 쪽에 select 가 action 이라는 것을 명시해줄게요.

// \*\*\*\* 액션 불러옴

import { decorate, observable, computed, autorun, action } from 'mobx';

class GS25 {

basket = [];

get total() {

console.log('계산중입니다..!');

// Reduce 함수로 배열 내부의 객체의 price 총합 계산

// https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global\_Objects/Array/Reduce

return this.basket.reduce((prev, curr) => prev + curr.price, 0);

}

select(name, price) {

this.basket.push({ name, price });

}

}

decorate(GS25, {

basket: observable,

total: computed,

select: action // \*\*\*\* 액션 명시

});

const gs25 = new GS25();

autorun(() => gs25.total);

gs25.select('물', 800);

console.log(gs25.total);

gs25.select('물', 800);

console.log(gs25.total);

gs25.select('포카칩', 1500);

console.log(gs25.total);

이 action 을 사용함에 있어서의 이점은 나중에 개발자도구에서 변화의 세부 정보를 볼 수 있고, 변화를 한꺼번에 일으켜서 변화가 일어날 때 마다 reaction 들이 나타나는것이 아니라, 모든 액션이 끝나고 난 다음에서야 reaction 이 나타나게끔 해줄 수 있습니다.

액션을 한꺼번에 일으키는건, transaction 을 통해 할 수 있습니다.

우선 다음 예제의 콘솔 결과를 확인해보겠습니다.

import {

decorate,

observable,

computed,

autorun,

action,

transaction // \*\*\* transaction 불러옴

} from 'mobx';

class GS25 {

basket = [];

get total() {

console.log('계산중입니다..!');

// Reduce 함수로 배열 내부의 객체의 price 총합 계산

// https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global\_Objects/Array/Reduce

return this.basket.reduce((prev, curr) => prev + curr.price, 0);

}

select(name, price) {

this.basket.push({ name, price });

}

}

decorate(GS25, {

basket: observable,

total: computed,

select: action

});

const gs25 = new GS25();

autorun(() => gs25.total);

// \*\*\* 새 데이터 추가 될 때 알림

autorun(() => {

if (gs25.basket.length > 0) {

console.log(gs25.basket[gs25.basket.length - 1]);

}

});

gs25.select('물', 800);

gs25.select('물', 800);

gs25.select('포카칩', 1500);

console.log(gs25.total);

계산중입니다..!   
계산중입니다..!   
Object {name: "물", price: 800}  
계산중입니다..!   
Object {name: "물", price: 800}  
계산중입니다..!   
Object {name: "포카칩", price: 1500}  
3100

계산의 경우, 가장 처음 한번 호출이 되고, 데이터가 추가 될 때마다 계산이 되고 있습니다.

한번 이걸 transaction 으로 감싸보겠습니다.

import {

decorate,

observable,

computed,

autorun,

action,

transaction

} from 'mobx';

class GS25 {

basket = [];

get total() {

console.log('계산중입니다..!');

// Reduce 함수로 배열 내부의 객체의 price 총합 계산

// https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global\_Objects/Array/Reduce

return this.basket.reduce((prev, curr) => prev + curr.price, 0);

}

select(name, price) {

this.basket.push({ name, price });

}

}

decorate(GS25, {

basket: observable,

total: computed,

select: action

});

const gs25 = new GS25();

autorun(() => gs25.total);

// 새 데이터 추가 될 때 알림

autorun(() => {

if (gs25.basket.length > 0) {

console.log(gs25.basket[gs25.basket.length - 1]);

}

});

transaction(() => {

gs25.select('물', 800);

gs25.select('물', 800);

gs25.select('포카칩', 1500);

})

console.log(gs25.total);

계산중입니다..!   
계산중입니다..!   
Object {name: "포카칩", price: 1500}  
3100

transaction 을 통하여 계산 작업은 가장 처음 한번, 그리고 transaction 끝나고 한번 호출이 되었고, 새 데이터 추가 될 때마다 알리는 부분도 3개를 다 추가하고 나서야 딱 한번 콘솔에 마지막 객체가 나타났습니다.

액션을 사용하면, 이렇게 성능 개선도 이뤄낼 수 있고 나중에 MobX 개발자 도구를 사용하게 될 때 변화에 대한 더 자세한 정보를 알 수 있게 해줍니다.

출처 : <https://velog.io/@velopert/begin-mobx>

**2. observer 디자인 패턴에 대해 설명하시오**

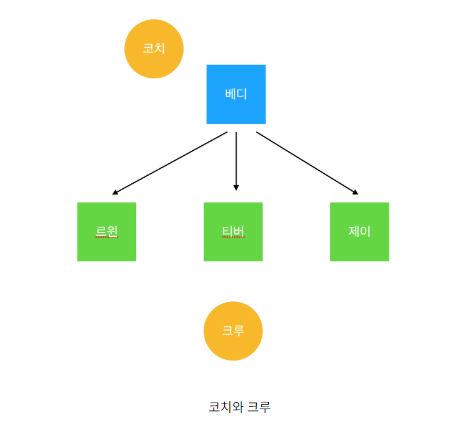
옵저버 패턴(observer pattern)은 객체의 상태 변화를 관찰하는 관찰자들, 즉 옵저버들의 목록을 객체에 등록하여 상태 변화가 있을 때마다 메서드 등을 통해 객체가 직접 목록의 각 옵저버에게 통지하도록 하는 디자인 패턴이다. 주로 분산 이벤트 핸들링 시스템을 구현하는 데 사용된다. 발행/구독 모델로 알려져 있기도 하다.

간단히 얘기하자면 **어떤 객체의 상태가 변할 때 그와 연관된 객체 들에게 알림을 보내는 디자인 패턴**이 옵저버 패턴이라고 할 수 있다.

**크루(학생)들은 코치가 하는 일들을 모두 notify(알림) 받아야 한다**. 즉, 코치가 "밥을 먹는다"면 모든 크루들은 코치가 밥을 먹었다는 것을 알아야 한다. 코치가 "농땡이를 친다"면 모든 크루들은 코치가 농땡이를 치는 것을 알아야 한다.

베디라는 객체는 코치이다. (코치 인터페이스를 구현해야 한다)

코치(코치 인터페이스)의 기능은 크루들을 등록한다, 크루들을 등록 해제한다, 크루들에게 행동을 알린다. 아주 간단하게 이 세가지의 기능을 가지고 있다.

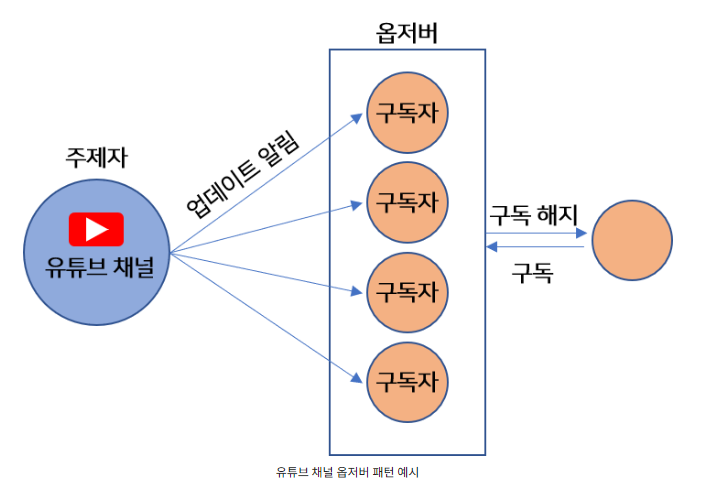


이렇게 코치인 베디는 모든 크루들에게 정보를 알려야 한다.

크루(크루 인터페이스)의 기능은 자신의 상태를 업데이트하는 기능을 가진다.

#### 예시 유튜브 구독 알람

**철수(Observer)**가 코딩 공부를 위해 유튜브에서 프로그래밍 관련 영상을 찾고 있습니다. 그런데 **Luckygg(Subject)**라는 사람의 채널에 좋은 영상들이 있는 것을 알게 됐습니다. 하나 둘 개제 된 영상들을 보게 되다가 어느새 모든 영상들을 보게 됐습니다. 이제 **새로운 영상이 업데이트되는 것**을 기다리고 있습니다. 그런데 이 영상이 언제 올라올 줄 알고 마냥 기다릴까요? 이때, 채널 **구독(Register)**을 하여 **업데이트 알림(Notify)을 받게 하는 방법**이 있습니다. 유튜브 채널 관리자가 영상을 업데이트하면, 구독하고 있는 모든 사용자들에게 알림을 전달하는 것입니다.



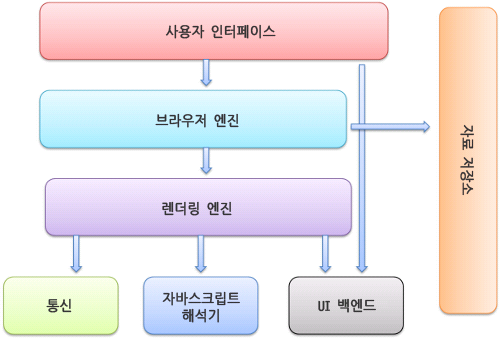
한 객체의 상태가 바뀌면 그 객체에 의존하는 다른 객체들에게 정보가 자동으로 갱신되는 방식으로, 일대다(one-to-many) 의존성을 정의합니다.

옵저버 패턴에서 주제자는 옵저버에 대한 정보가 없습니다. 오직 옵저버가 **특정 인터페이스(Interface)를 구현**한다는 것 만 알고 있습니다. 즉, **옵저버가 무슨 동작을 하는지 모른다는 것**입니다. 게다가 옵저버는 **언제든지 새로 추가**되거나 **제거**될 수 있으며 새로운 형식의 옵저버를 추가할 때에도 **주제에 전혀 영향을 주지 않습니다**.

3. 브라우저 렌더링 과정에 대해 상세하게 설명하시오.

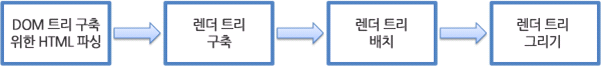
1. 브라우저의 구조

* 사용자 인터페이스: 주소 표시줄, 이전/다음 버튼, 북마크 메뉴 등. 요청한 페이지를 보여주는 창을 제외한 나머지 모든 부분이다.
* 브라우저 엔진: 사용자 인터페이스와 렌더링 엔진 사이의 동작을 제어.
* 렌더링 엔진: 요청한 콘텐츠를 표시. 예를 들어 HTML을 요청하면 HTML과 CSS를 파싱하여 화면에 표시함.
* 통신: HTTP 요청과 같은 네트워크 호출에 사용됨. 이것은 플랫폼 독립적인 인터페이스이고 각 플랫폼 하부에서 실행됨.
* UI 백엔드: 콤보 박스와 창 같은 기본적인 장치를 그림. 플랫폼에서 명시하지 않은 일반적인 인터페이스로서, OS 사용자 인터페이스 체계를 사용.
* 자바스크립트 해석기: 자바스크립트 코드를 해석하고 실행.
* 자료 저장소: 이 부분은 자료를 저장하는 계층이다. 쿠키를 저장하는 것과 같이 모든 종류의 자원을 하드 디스크에 저장할 필요가 있다. HTML5 명세에는 브라우저가 지원하는 '웹 데이터 베이스'가 정의되어 있다.



1. 렌더링 엔진

* 파이어폭스는 모질라에서 직접 만든 게코(Gecko) 엔진을 사용하고 사파리와 크롬은 웹킷(Webkit) 엔진을 사용
* 동작과정



1. DOM(Document Object Model), CSSOM(CSS Object Model) 생성

가장 첫번째 단계는 서버로부터 받은 HTML, CSS를 다운로드 받습니다. 그리고 HTML, CSS파일은 단순한 텍스트이므로 연산과 관리가 유리하도록 Object Model로 만들게 됩니다. HTML CSS 파일은 각각 DOM Tree와 CSSOM으로 만들어집니다.

<html>

 <head>

<meta name="viewport" content="width=device-width,initial-scale=1">

   <link href="style.css" rel="stylesheet">

   <title>Critical Path</title>

 </head>

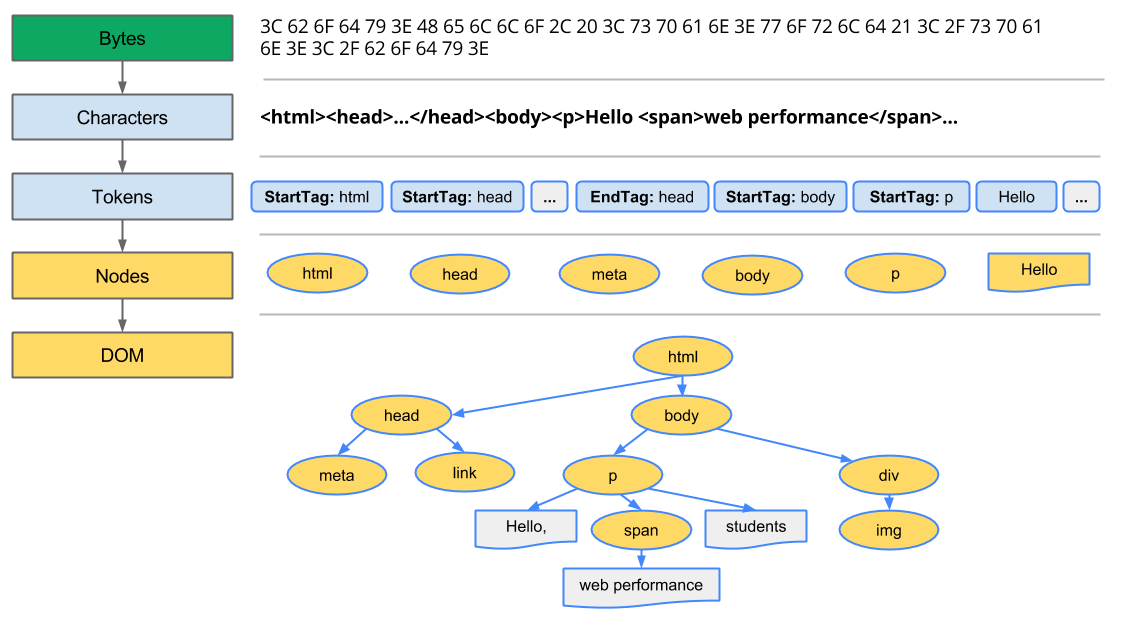
 <body>

   <p>Hello <span>web performance</span> students!</p>

   <div><img src="awesome-photo.jpg"></div>

 </body>

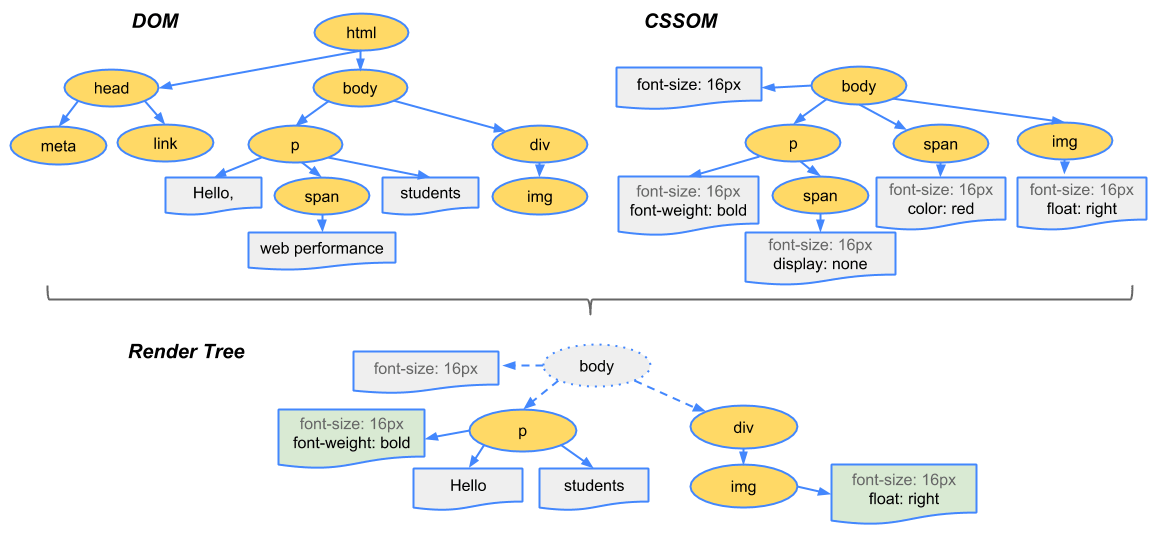
</html>



* 변환: 브라우저가 HTML의 원시 바이트를 디스크나 네트워크에서 읽어와서, 해당 파일에 대해 지정된 인코딩(예: UTF-8)에 따라 개별 문자로 변환합니다.
* 토큰화: 브라우저가 문자열을 W3C HTML5 표준에 지정된 고유 토큰으로 변환합니다(예: '', '' 및 꺽쇠 괄호로 묶인 기타 문자열). 각 토큰은 특별한 의미와 고유한 규칙을 가집니다.
* 렉싱: 방출된 토큰은 해당 속성 및 규칙을 정의하는 '객체'로 변환됩니다.
* DOM 생성: 마지막으로, HTML 마크업이 여러 태그(일부 태그는 다른 태그 안에 포함되어 있음) 간의 관계를 정의하기 때문에 생성된 객체는 트리 데이터 구조 내에 연결됩니다. 이 트리 데이터 구조에는 원래 마크업에 정의된 상위-하위 관계도 포함됩니다. 즉, HTML 객체는 body 객체의 상위이고, body 는 paragraph 객체의 상위인 식입니다.

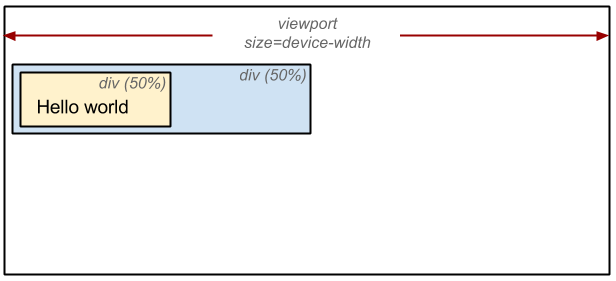
1. Render Tree 생성

DOM Tree와 CSSOM Tree가 만들어졌으면 그 다음으로는 이 둘을 이용하여 Render Tree를 생성합니다. 순수한 요소들의 구조와 텍스트만 존재하는 DOM Tree와는 달리 Render Tree에는 스타일 정보가 설정되어 있으며 실제 화면에 표현되는 노드들로만 구성됩니다.



1. Layout

Layout 단계는 브라우저의 뷰포트(Viewport) 내에서 각 노드들의 정확한 위치와 크기를 계산합니다. 풀어서 얘기하자면 생성된 Render Tree 노드들이 가지고 있는 스타일과 속성에 따라서 브라우저 화면의 어느위치에 어느크기로 출력될지 계산하는 단계라고 할 수 있습니다. Layout 단계를 통해 %, vh, vw와 같이 상대적인 위치, 크기 속성은 실제 화면에 그려지는 pixel단위로 변환됩니다.



여기서 뷰포트(Viewport)란 그래픽이 표시되는 브라우저의 영역, 크기를 말합니다. 뷰포트는 모바일의 경우 디스플레이의 크기, PC의 경우 브라우저 창의 크기에 따라 달라집니다. 그리고 화면에 그려지는 각 요소들의 크기와 위치는 %, vh, vw와 같이 상대적으로 계산하여 그려지는 경우가 많기 때문에 viewport 크기가 달라질 경우 매번 계산을 다시해야 합니다.

1. Paint

Layout 계산이 완료되면 이제 요소들을 실제 화면을 그리게 됩니다. 이전 단계에서 이미 요소들의 위치와 크기, 스타일 계산이 완료된 Render Tree 를 이용해 실제 픽셀 값을 채워넣게 됩니다. 이 때 텍스트, 색, 이미지, 그림자 효과등이 모두 처리되어 그려집니다.

이 때 처리해야 하는 스타일이 복잡할수록 Paint 단계에 소요되는 시간이 늘어나게 됩니다. 간단한 예시로 단순한 단색 background-color의 경우 paint 속도가 빠르지만 그라데이션이나 그림자 효과등은 painting 소요시간이 비교적 더 오래 소요됩니다.