



Module

# Javascript 클라이언트 [담]을 넘다\_

클라이언트/서버 환경에서 JavaScript를 범용적으로 사용하기 위한 조건

## — 모듈화 (Modularization)

### JavaScript 모듈화 선두 주자

- > CommonJS ([commonjs.org](http://commonjs.org))
- > AMD ([github.com/amdjs/amdjs-api/blob/master/AMD.md](https://github.com/amdjs/amdjs-api/blob/master/AMD.md))

# CommonJS

# AMD

두 진영에서 정의하는 모듈 명세의 차이는 모듈 로드에 있다.

필요한 파일이 모두 로컬 디스크에 있어 바로 불러쓸 수 있는 상황, 즉 서버사이드에서는 CommonJS 명세가 AMD 방식보다 간결하다. 반면 필요한 파일을 네트워크를 통해 내려받아야 하는 브라우저와 같은 환경에서는 AMD가 CommonJS보다 더 유연한 방법을 제공한다.



# CommonJS

Common + Javascript



클라이언트, 서버사이드 애플리케이션이나 데스크톱 애플리케이션에서도 JavaScript를 사용하려고 조직한 자발적 워킹 그룹. (2009.01, Kevin Dangoor)  
JavaScript 사용 표준화를 위해 필요한 명세(Specification)를 만듬.

2005년 Ajax가 부상하면서 JavaScript의 중요성은 그전보다 더 부각되었다. Ajax의 활성화와 함께 JavaScript 연산이 증가했고, 자연스레 더 빠른 JavaScript 엔진이 필요하게 되었다.

이런 맥락에서 2008년 Google에서 공개한 V8 JavaScript 엔진은 많은 주목을 받았다. V8 엔진은 기존의 JavaScript 엔진보다 월등히 빨랐을 뿐 아니라, 브라우저 밖에서도 충분히 쓸만한 성능을 자랑했다.

JavaScript 표준화를 위해서는 Ruby나 Python과 같은 체계가 필요!

- 서로 호환되는 표준 라이브러리가 없다.
- 데이터베이스에 연결할 수 있는 표준 인터페이스가 없다.
- 다른 모듈을 삽입하는 표준적인 방법이 없다.
- 코드를 패키징해서 배포하고 설치하는 방법이 필요하다.
- 의존성 문제까지 해결하는 공통 패키지 모듈 저장소가 필요하다.

# CommonJS

---

앞에서 언급한 문제점들은 결국 모듈화로 귀결된다. 그리고 CommonJS의 주요 명세는 바로 이 모듈을 어떻게 정의하고, 어떻게 사용할 것인가에 대한 것이다.

## JavaScript 표준화를 위한 모듈화

- **스코프(Scope)** 모든 모듈은 자신만의 독립적인 실행 영역이 있어야 한다.
- **정의(Definition)** 모듈 정의는 exports 객체를 이용한다.
- **사용(Usage)** 모듈 사용은 require 함수를 이용한다.

# Scope

모듈은 자신만의 독립적인 실행 영역이 있어야 한다. 따라서 전역변수와 지역변수를 분리하는 것이 매우 중요하다. 서버사이드 JavaScript의 경우 파일마다 독립적인 파일 스코프가 있기 때문에 파일 하나에 모듈 하나를 작성하면 간단히 해결된다. 즉 서버사이드 JavaScript는 아래와 같이 작성하더라도 전역변수가 겹치지 않는다.

## Apple.js

```
var brand = 'Apple',  
    watch = 'Apple Watch';
```

## Samsung.js

```
var brand = 'Samsung',  
    watch = 'Galaxy Gear';
```

# Definition

두 모듈(파일) 사이에 정보 교환이 필요하다면, exports 전역객체를 통해 공유하게 된다.  
exports를 통해 전역으로 공개된 데이터는 다른 모듈에서 사용 가능해진다.

## Apple.js

```
var brand = 'Apple',
    watch = 'Apple Watch';

// 전역 객체를 통한 모듈 공유
exports.launchProduct = function(brand, product) {
    return brand + '사의 ' + product + ' 출시!';
};
```

# Usage

공개(공유)된 함수를 다른 모듈에서 사용하려면, 다음과 같이 require() 함수를 이용한다.

CommonJS의 모듈 명세는 모든 파일이 로컬 디스크에 있어 필요할 때 바로 불러올 수 있는 상황을 전제로 한다.  
다시 말해 서버사이드 JavaScript 환경을 전제로 한다.

## Samsung.js

```
var brand      = 'Samsung',
    watch       = 'Galaxy Gear',
    releaseProduct = require('./Apple.js');
```

```
// Apple.js 모듈(파일)의 launchProduct 함수(공유)를
// Samsung.js 모듈에서 불러들여 releaseProduct 함수로 사용.
releaseProduct(brand, watch);
```

# Problem of Client Side

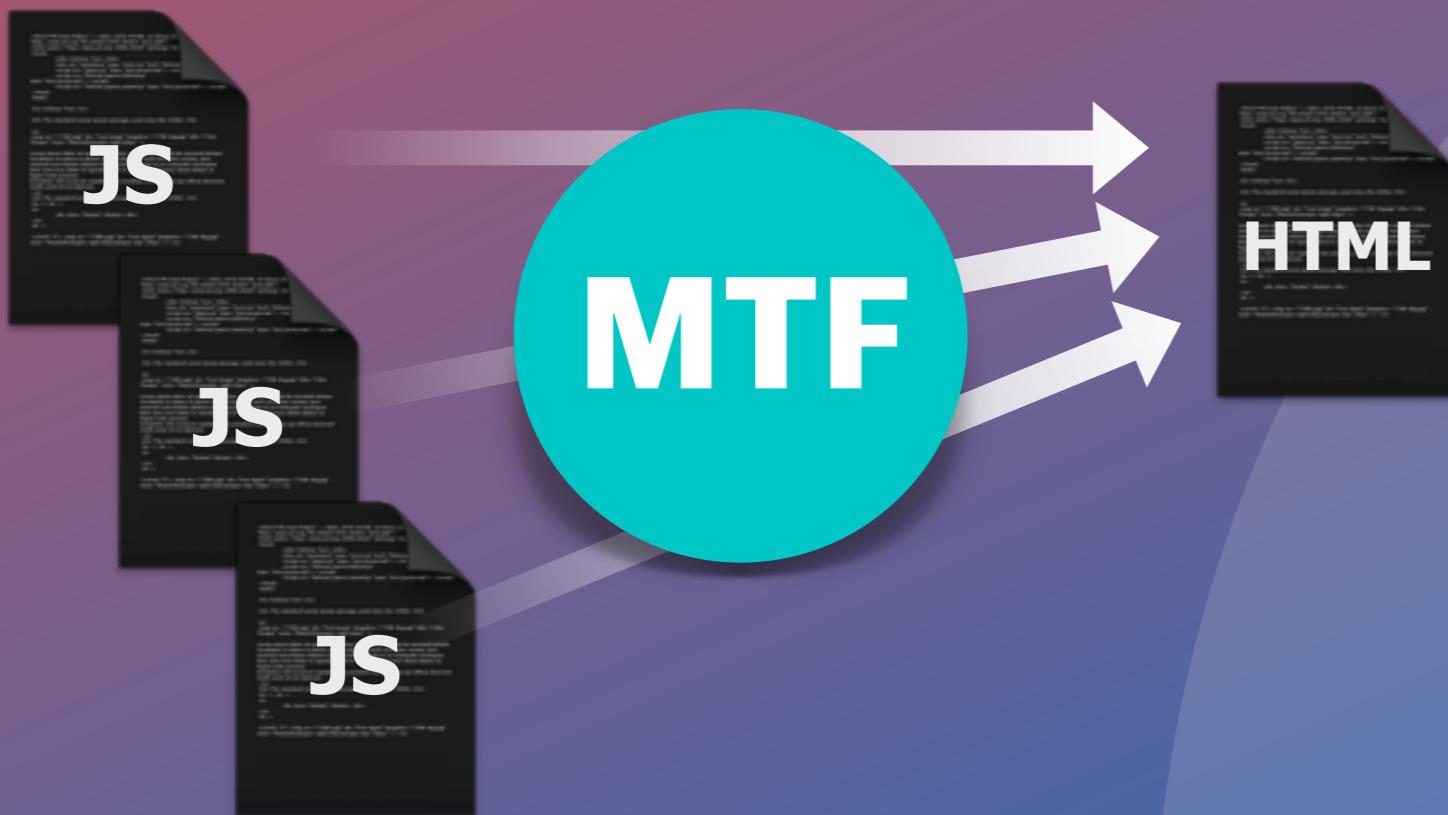
이런 방식은 브라우저에서는 결정적인 단점이 있다. 필요한 모듈을 모두 내려받을 때까지 아무것도 할 수 없게 되는 것이다. 이 단점을 극복하려는 여러 방법이 CommonJS에서 논의되었지만, 결국 동적으로 <script> 태그를 삽입하는 방법으로 가닥을 잡는다. <script> 태그를 동적으로 삽입하는 방법은 JavaScript로 더들이 사용하는 가장 일반적인 방법이기도 하다.

```
function loaderJS(path) {  
    var js = document.createElement('script');  
    js.setAttribute('src', path);  
    document.head.appendChild(js);  
}
```

# Problem of Asynchronous Module Load

---

JavaScript가 브라우저에서 동작할 때는 서버 사이드 JavaScript와 달리 파일 단위의 스코프가 없다. 즉, 클라이언트 환경에서는 전역 변수/함수를 덮어쓰는 문제가 발생한다. 이 문제를 해결하기 위해 CommonJS는 서버 모듈을 비동기적으로 클라이언트에 전송할 수 있는 모듈 전송 포맷(Module Transport Format)을 추가로 정의했다.



# Problem of Asynchronous Module Load

## Server-Side

```
// 서버-사이드에서 모듈 정의  
// -----  
  
// 의존 모듈 호출  
var organize = require("./company").organize;  
  
// globalization 함수 공개(공유)  
exports.globalization = function(brand){  
    return organize(brand);  
};
```

Node.js는 Javascript 서버사이드 환경으로  
CommonJS 방법을 사용하여 모듈을 공유한다.

# Problem of Asynchronous Module Load

## Client-Side

```
// 클라이언트-사이드에서 모듈 정의  
// 모듈 전송 포맷(module transport format)  
// -----  
  
// 의존 모듈 호출  
require.define({  
    // 콜백 함수를 이용해 모듈 정의  
    "./company/globalization": function(require, exports) {  
        var organize = require("./company").organize;  
        exports.globalization = function(brand) {  
            return organize(brand);  
        };  
    }  
    // 먼저 로드되어야 할 모듈을 기술한다.  
}, ["./company"]);
```

브라우저에서 사용하는 모듈 부분에서 특히 주목해야 할 것은 **require.define()** 함수를 통해(함수 클로저) 전역변수를 통제하고 있다는 사실이다.

# Asynchronous Module Definition

# AMD

JavaScript 표준 API 라이브러리 제작 그룹에는 CommonJS만 있는 것이 아니고, AMD라는 그룹도 있다. AMD 그룹은 비동기 상황에서도 JavaScript 모듈을 쓰기 위해 CommonJS에서 함께 논의하다 합의점을 이루지 못하고 독립한 그룹이다.

본래 CommonJS가 JavaScript를 브라우저 밖으로 꺼내기 위한 노력의 일환으로 탄생했기 때문에 브라우저 내에서의 실행에 중점을 두었던 AMD와는 합의를 이끌어 내지 못하고 결국 둘이 분리되었다.

AMD 진영의 대표적인  
모듈 로더 라이브러리

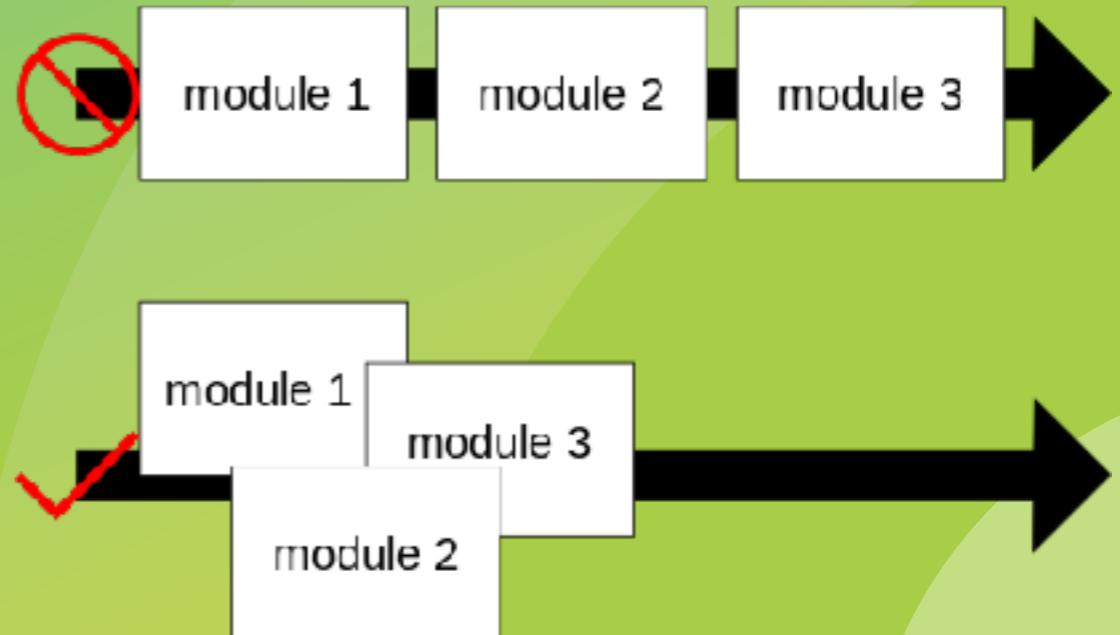


# AMD

'Asynchronous Module Definition'이라는 말에서 알 수 있듯이, AMD에서는 비동기 모듈(필요한 모듈을 네트워크를 통해 내려받을 수 있도록 하는 것)에 대한 표준안을 다루고 있다. 물론 CommonJS도 비동기 상황을 고려한 모듈 전송 포맷을 제공하지만, 순수 AMD를 지지하는 사람들과 합의를 도출해 내지는 못했다.

이런 역사적 배경 때문에 AMD는 CommonJS와 많은 부분이 닮아 있거나 호환할 수 있는 기능을 제공한다. `require()` 함수를 사용할 수 있으며, `exports` 형태로 모듈을 정의할 수도 있다. 물론 AMD만의 특징도 있다. 대표적으로 꼽을 수 있는 것이 바로 `define()` 함수다.

브라우저 환경의 JavaScript는 파일 스코프가 따로 존재하지 않기 때문에 이 `define()` 함수로 파일 스코프의 역할을 대신한다. 즉, 일종의 네임스페이스 역할을 하여 모듈에서 사용하는 변수와 전역변수를 분리한다. 물론 `define()` 함수는 전역함수로 AMD 명세를 구현하는 서드파티 벤더가 모듈 로더에 구현해야 한다.



# Define

define()함수는 전역함수로 다음과 같이 정의한다.

첫번째 인자는 모듈 식별을 위한 ID 값으로 특별한 경우가 아닐 경우, 생략 가능하다. 만약 id를 명시한다면 파일의 절대 경로를 식별자로 지정해야 한다.

```
define(id, [dependencies], factory);
```

세번째 인자는 모듈이나 객체를 인스턴스화하는 실제 구현을 담당. 인자가 함수일 경우 싱글톤으로 한번만 실행되고, 값을 반환하려면 exports 객체의 속성으로 할당한다. 반면 인자가 객체라면 exports 객체의 속성으로 할당된다.

두번째 인자는 모듈 정의에 사용되는 의존성 모듈 배열을 명시한다. 즉, 이 곳에 명시된 모듈이 모두 호출되기 전에는 모듈 정의가 수행되지 않는다.

생략할 경우 ['require', 'exports', 'module'] 기본 값이 설정된다. 이 모듈들은 CommonJS의 전역 객체와 동일한 역할을 수행한다.

# AMD Module Define 1

3가지 인자를 모두 사용하는 기본 AMD 모듈로 Tabs 모듈을 정의하는데  
jQuery 모듈이 필요하다는 것을 나타낸다.

```
define(  
    "Tabs",  
    [  
        "require",  
        "exports",  
        "jquery"  
    ],  
    function (require, exports, jQuery) {  
        exports.Tabs = function() {  
            // 전달받은 인자를 사용할 경우  
            var $ = jQuery;  
            // require()를 사용할 경우  
            var $ = require('jquery');  
            // jQuery를 사용하여 작성할 코드  
            // ...  
        };  
    });
```



# AMD Module Define 2

첫 번째 인자를 생략할 경우, jQuery 모듈을 필요로 하는 이름 없는 모듈을 만든다.

이때 require() 함수로 이 모듈을 사용하고 싶다면, 이 모듈이 정의된 파일의 경로를 지정해야 한다.

```
define( ["jquery"], function (jQuery) {
    exports.Tabs = function() {
        // 전달받은 인자를 사용할 경우
        var $ = jQuery;
        // jQuery를 사용하여 작성할 코드
        // ...
    };
});
```

```
require( "/js/Tabs.js" ), function (Tabs) {
    // Tabs 모듈을 사용하여 작성할 코드
    // ...
};
```

# AMD Module Define 3

---

의존성이 없는 모듈을 정의한다. 이 모듈 역시 첫 번째 id 식별자가 없으므로,  
모듈이 정의된 파일의 경로가 자동으로 식별자로 지정된다.

```
define({
  Tabs : function() {
    // Tabs를 정의하는 모듈 코드
    // ...
  },
});
```

# AMD Module Define 4

CommonJS 형태의 모듈을 래핑할 수도 있다.

```
define(function (require, exports, module) {  
    var $ = require('jquery'),  
        Modernizr = require('modernizr'),  
        exports.Tabs = function() {};  
});
```

# Asynchronous Module Definition

# AMD

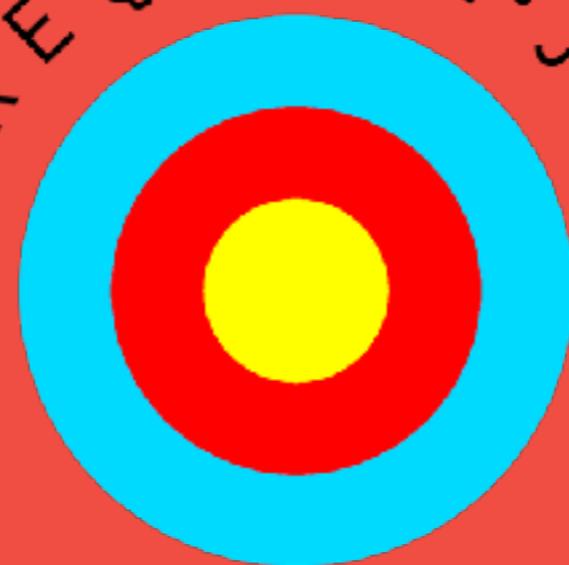
AMD 모듈 명세의 장점은 단연 비동기 환경에서도 매우 잘 동작할 뿐만 아니라, 서버사이드에서도 동일한 코드로 동작한다는 점이다. 그리고 CommonJS의 모듈 전송 포맷보다는 확실히 간단하고 명확하다.

AMD 명세는 `define()` 함수(클로저를 이용한 모듈 패턴)를 이용해 모듈을 구현하므로 전역변수 문제가 없다. 또한 해당 모듈을 필요한 시점에 로드하는 Lazy-Load 기법을 응용할 수도 있다.

성능 측면에서 보면, 확실히 구 버전의 Internet Explorer에서는 많은 이득을 볼 수 있지만, 그 외의 최신 브라우저에서는 성능이 비슷하다.



REQUIREMENTS





# RequireJS

RequireJS는 AMD 표준 명세를 따르는 모듈 로더 라이브러리.  
AMD는 <동적 로딩>, <의존성 관리>, <모듈화>가 톱니바퀴처럼  
아름답게 맞물린 API 디자인을 제시.

**Dynamic  
Loading**

**Manage  
Dependency**

**Modular-  
ization**



## Dynamic Loading

# 동적 로딩

<script> 태그는 페이지 렌더링을 방해한다. <script> 태그의 HTTP 요청과 다운로드, 파싱(Parsing), 실행이 일어나는 동안 브라우저는 다른 동작을 하지 않는다. 브라우저 입장에서는 당연하고 안전한 동작 방식이지만 사용자 입장에서는 빨리 화면이 보이고 버튼이 동작하기를 바랄 뿐이다. 그래서 최적화 기법 중의 하나로 <script> 태그를 가능한 한 <body> 태그의 마지막에 배치하는 방법이 있다.

동적 로딩(Dynamic Loading, Lazy Loading이라고도 부른다)은 페이지 렌더링을 방해하지 않으면서 필요한 파일만 로딩할 수 있다. 이를 구현하는 방법 중 하나로 <script> 태그의 동적 삽입이 있다. 이는 JavaScript로 <script> 태그를 생성하여 추가하는 방법이다. 이 외에도 다른 방법이 존재하지만, <script> 태그의 동적 삽입이 제일 안전하고 합리적이다.

```
function loaderJS(path, callback) {
  var js = document.createElement('script');
  js.setAttribute('src', path);
  // IE는 onreadystatechange 활용
  js[js.onreadystatechange ? 'onreadystatechange': 'onload'] = callback;
  document.head.appendChild(js);
}

loaderJS('example.js', function(){
  // example.js 로딩 완료 시점에 실행
});
```



Manage  
Dependency

# 의존성 관리

JavaScript는 스크립트 간의 의존성을 파악하기 힘들다. 언어 차원에서 `#include`나 `package/import`같은 명시적이고 강제적인 키워드, 패키징 정책을 지원하지 않기 때문이다. 파일 상단에 JSDoc `@requires`라도 적혀 있으면 다행이지만 이는 결국 주석일 뿐이다.

앞서 다룬 `loadJS()` 함수는 C/C++의 `#include` 기능을 흉내 낼 수 있다. 필요한 파일 목록과 로딩 순서를 파악 하는 데 도움이 되긴 하지만 파일 이름으로 사용하려는 함수나 객체의 이름을 유추할 수 있다는 보장은 없다. 코딩 규칙만으로는 부족하고 불안전하다. 결국 Java의 `package/import`같은 기능이 필요하고 이것이 특정 기능을 불러와서 사용할 수 있는 유일한 방법이어야 한다.

이를 위한 기본 조건은 특정 기능의 스크립트가 이름을 붙일 수 있는 하나의 단위로 묶여야 한다. 그래야 다른 스크립트에서 그 이름으로 스크립트를 불러올 수 있는 방법이 생기기 때문이다.



Manage  
Dependency

# 의존성 관리

유틸리티 함수를 모아놓은 객체를 모듈화하고자 할 경우, 일반적으로 그 객체를 전역변수 util로 할당하고 사용 하겠지만, 이 객체를 불러오는 강제적이고 유일한 방법을 구현해야 하므로 먼저 객체 이름과 정의를 비공개 공간에 넣을 수 있는 함수가 필요하다. 그 함수를 사용해서 모듈을 정의하는 방법은 다음과 같다.

```
defineModule('util', {
  proxy : function () {},
  extend : function () {}
});
```

정의한 모듈을 사용하기 위해서는 반대로 호출 함수가 있어야 한다. 정의한 객체가 비공개 공간에 있기 때문이다. 이 호출 함수가 다음과 같이 모듈을 불러오는 강제적이고 유일한 방법이 된다.

```
// 모듈 호출
var util = loadModule('util');

util.proxy();
```



## Modular- ization

# 모듈화

스크립트 내부에서만 사용하는 변수, 함수들은 전역 공간에 둘 필요가 없고 두어서도 안 된다. 전역 공간 오염으로인한 충돌은 유지 보수에 막대한 영향을 끼쳐서 심신을 괴롭히기 때문이다. 스크립트 모듈화는 이런 문제를 사전에 방지한다.

모듈 패턴을 작성하는 기본 방법은 `return`으로 외부에서 접근할 변수와 함수만 골라서 노출할 수 있으며, 외부에 노출할 필요 없는 변수와 함수는 클로저 (Closure)를 이용하여, 전역 공간에 위치시키지 않고 접근할 수 있도록 제한한다.

```
var counter = (function () {
    var i = 0;
    function init() {
        reset();
    }
    function reset() {
        i = 0;
    }
    function increase() {
        i++;
    }
    function decrease() {
        i--;
    }
    function get() {
        return i;
    }
    return {
        init : init,
        increase : increase,
        decrease : decrease,
        get : get
    };
}());
```



## Modular- ization

# 모듈화

앞서 정의한 counter 모듈은 결과적으로 단순 객체를 반환하므로 싱글턴(Singleton)으로 볼 수 있는데, 이를 조금 응용하면 모듈을 일종의 클래스(Class)처럼 사용할 수도 있다.

```
var DOM = (function (global, document) {
    'use strict';
    // 생성자 함수
    var _dom = function(selector) {
        this.el = document.querySelectorAll(selector);
    };
    // 생성자 프로토타입
    _dom.prototype = {
        'find': function() {},
        'attr': function() {}
    };
    return _dom;
}(this, this.document));
```



# RequireJS 로드

RequireJS 모듈 로더 라이브러리를 웹 페이지에서 호출한다.



```
<!DOCTYPE html>
<html lang="ko-KR">
<head>
    <meta http-equiv="X-UA-Compatible" content="IE=Edge">
    <meta charset="UTF-8">
    <title>RequireJS 호출/모듈정의/사용</title>
    <!-- RequireJS 호출 -->
    <script src="js/vender/require.min.js"></script>
</head>
<body>

</body>
</html>
```



# RequireJS 모듈 로드 후, 콜백

RequireJS 모듈 로더 라이브러리는 require() 함수를 통해 의존하는 모듈을 먼저 로드한 후, 콜백 함수를 실행시키는 구조이다.

```
yamoo9_example.html
yamoo9_example.html + 1

<script src="js/vender/require.min.js"></script>
<script>
    // jQuery를 가져온 후 실행한다.
    require(['http://code.jquery.com/jquery.min.js'], function() {
        // jQuery(CDN) 로드가 완료되면 코드 실행
        console.log( $().jquery ); // jQuery 버전 출력
    });
</script>
```



# RequireJS 모듈 정의

```
yamoo9_example.html
```

```
/* js/dom.js */
// 모듈 정의 기본 형태
define(
// 의존 모듈들. 모듈이 한 개라도 배열로 넘겨야 함.
[
    'js/dom/manipulation',
    'js/dom/events',
    'js/dom/utils',
],
// 의존 모듈들 순서대로 매개변수에 담김.
function (manipulation, events, utils) {
    // 의존 모듈들이 모두 로딩되면 콜백 함수 실행
    var _ver = '1.0.4',
        _on = events.addEvent;
    // 외부에 노출할 멤버만 반환한다.
    return {
        'version' : _ver,
        'on'       : _on
    };
});
```

RequireJS 모듈 패턴(`define`)은 전역 공간에 덮어쓰거나 오염시키는 일 없이 **별도의 컨텍스트에서 자바스크립트를 실행한 뒤에 결과를 다시 호출한 곳으로 리턴**해줄 수 있다.



# RequireJS 모듈 사용

```
/* js/main.js */
require([
    'js/dom.js'
], function (dom) {
    dom.on(); // js/dom.js 모듈 객체 멤버 사용
});
```

RequireJS 모듈 로더 라이브러리는 **require()** 함수를 통해  
의존하는 모듈을 먼저 로드한 후, 콜백 함수를 실행시키는  
구조이다.



# RequireJS 모듈 정의/사용

모듈의 이름을 명시적으로 설정할 수도 있지만 이름 없는 모듈로 정의하는 것을 권장한다.

이름 없는 모듈은 호출될 때 모듈의 위치에 따라 이름을 결정한다. 개발할 때 파일의 이름이나 위치는 자주 변경되므로 유연한 상태로 둘 필요가 있다.

의존 모듈은 배열로 나열하긴 했지만 로딩 순서를 보장한다는 뜻은 아니다. 순서에 상관없이 병렬로 네트워크를 통해 다운로드되거나 브라우저의 캐시에서 꺼내진다. 어떤 모듈이 먼저 로딩되어 실행될지 모른다. 따라서 로딩 순서가 중요하다면 아래와 같이 require를 중첩해서 사용하는 방법이 있다.

The screenshot shows a browser window with the title "yamoo9\_example.html". The address bar also displays "yamoo9\_example.html". The main content area contains the following JavaScript code:

```
require(['js/moduleA'], function (moduleA) {
    require(['js/moduleB'], function (moduleB) {
        //
    });
});
```



# RequireJS & jQuery

jQuery를 AMD 방식으로 사용하고 싶다면 jQuery 코드 뒷 부분에 아래와 같은 코드를 넣은 후 require() 함수로 의존성 관리하여 사용하면 된다.

```
yamoo9_example.html
+ yamoo9_example.html +≡
1
//jquery 파일 모듈화
define(function() {

    // 원본 jQuery 코드
    // ...
    //
    // jQuery 객체를 리턴한다.
    // true 값을 전달하면 window.jQuery 변수도 제거된다.
    return jQuery.noConflict(true);

});
```



# RequireJS & jQuery

jQuery 코드 하단에 보면 AMD 환경에서 jQuery를 사용할 수 있도록 이미 모듈 배포 코드가 구현되어 있다.

```
// Register as a named AMD module, since jQuery can be concatenated with other
// files that may use define, but not via a proper concatenation script that
// understands anonymous AMD modules. A named AMD is safest and most robust
// way to register. Lowercase jquery is used because AMD module names are
// derived from file names, and jQuery is normally delivered in a lowercase
// file name. Do this after creating the global so that if an AMD module wants
// to call noConflict to hide this version of jQuery, it will work.

// Note that for maximum portability, libraries that are not jQuery should
// declare themselves as anonymous modules, and avoid setting a global if an
// AMD loader is present. jQuery is a special case. For more information, see
// https://github.com/jrburke/requirejs/wiki/Updating-existing-libraries#wiki-anon

if ( typeof define === "function" && define.amd ) {
    define( "jquery", [], function() {
        return jQuery;
    });
}
```



# jQuery 버전별 충돌 없이 사용

프로젝트에 다수의 jQuery 버전을 사용하여야 한다면 아래 처럼 각 영역에서 해당 jQuery 버전을 사용하여 프로그래밍 할 수 있다.

```
yamoo9_example.html
+ yamoo9_example.html +≡
1
<script>
  //jQuery 1.11.2 버전 사용
  require(['http://code.jquery.com/jquery-1.11.2.js'], function($) {
    // 이 공간 $에는 jQuery 1.11.2 객체가 담기게 된다.
    var jQuery = $;
    console.log(( $().jquery) ); // 1.11.2
  });

  //jQuery 2.1.4 버전을 사용
  require(['http://code.jquery.com/jquery-2.1.4.js'], function($) {
    // 이 공간 $에는 jQuery 2.1.4 객체가 담기게 된다.
    var jQuery = $;
    console.log(( $().jquery) ); // 2.1.4
  });
</script>
```



# RequireJS 설정 및 비동기 로드

RequireJS는 다양한 설정 옵션을 제공하며, 설정 값에 따라 비동기 로드할 수 있다.

비동기 로드 방법은 `data-main` 속성 값으로 호출하고자 하는 JS 파일 경로를 제공한다.

```
yamoo9_example.html
+ × yamoo9_example.html +≡
1
<script>
    // Require.js 환경설정(Configuration)
    // http://requirejs.org/docs/api.html#config
    // require.js가 로딩 되면 이 객체를 자동으로 읽어 들여 반영한다.
    var require = {
        // index.html 기준, 모듈의 기본 위치 설정
        baseUrl: 'js/'
    };
</script>

<!-- data-main 속성에 설정된 JS 파일은 비동기 호출됩니다.
(* 파일 경로는 Require.js에 설정된 baseUrl 기준:
http://requirejs.org/docs/api.html#jsfiles) -->
<script src="js/libs/require.js" data-main="js/main"></script>
```





# RequireJS 설정 옵션

```
yamoo9_example.html
+ yahoo9_example.html +≡
1 // Require.js 환경설정(Configuration)
// http://requirejs.org/docs/api.html#config
// require.js가 로딩 되면 이 객체를 자동으로 읽어 들여 반영한다.
var require = {
    // 모듈의 기본 위치 설정 (index.html 기준)
    baseUrl: 'js/app'
    // 모듈 단축 경로 또는 별칭(Alias) 지정
    paths: {
        vender: '../vender' // "js/vender"와 동일 (baseUrl 기준)
    },
    // AMD 미지원 외부 라이브러리 모듈 사용을 위한 shim 등록
    shim: {
        // Modernizr 라이브러리
        'modernizr': {
            exports: 'Modernizr'
        }
    },
    // 모듈 위치 URL 쿼리 접미사(Time Stamp) 설정
    urlArgs : 'ts=' + (new Date()).getTime()
};
```



# RequireJS 설정 옵션

`var require = {};` 설정은 `require.js` 파일의 로딩 전에 사용하는 방법이다.  
`require.js` 파일을 로딩한 후에는 `require.config()` 함수를 사용하여 설정할 수 있다.

```
yamoo9_example.html
+ × yamoo9_example.html +≡
1
// js/app/main.js
require.config({
  'baseUrl' : '',
  'paths'   : {},
  'shim'    : {},
  'urlArgs' : 'ts=' + (new Date()).getTime()
});

define([
  // 의존 모듈 관리
], function() {
  // 의존 모듈 로드 후, 콜백 함수
});
```



# RequireJS 모듈 위치

RequireJS는 호출하는 모듈의 위치를 찾을 때 baseUrl과 이름을 결합하여 찾는다. baseUrl이 유동적이라면 결국 모듈 위치도 유동적이 된다는 이야기이므로 특별한 경우가 아니라면 다음과 같이 baseUrl을 설정하는 편이 좋다.

The screenshot shows a browser window titled "yamoo9\_example.html". The code in the editor is as follows:

```
// js/app/main.js
require.config({
  'baseUrl' : 'js/',
});

// ...
require([
  'vender/detectizr.min',
  'app.js',
  '/js/vender/modernizr.min.js',
  'http://code.jquery.com/jquery.min.js'
], function (Detectizr, app, Modernizr, $) {
  //
});

```

A red box highlights the module names in the require() call: 'vender/detectizr.min', 'app.js', and 'http://code.jquery.com/jquery.min.js'. A callout bubble points to these highlighted modules with the following text:

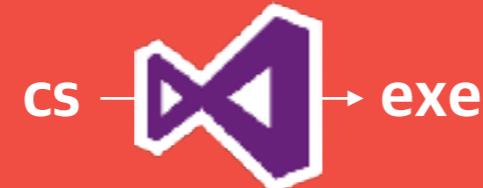
- (1) 위치: "/js/vender/detectizr.min.js"
- (2) 위치: "/app.js"
- (3) 위치: "/js/vender/modernizr.min.js"
- (4) 위치: "http://code.jquery.com/jquery.min.js"



# Build 프로세스

Javascript는 C#, JAVA와 달리 빌드 프로세스 기능을 기본적으로 지원하지 않습니다만,  
RequireJS를 사용하면 라이브러리에서 빌드 프로세싱 처리를 할 수 있다.

Develop



Production



min.js



# RequireJS 최적화

RequireJS는 `r.js`를 이용해 Node.js 서버 환경에서 코드를 최적화하는 기능을 지원한다. `node` 명령으로 `r.js`를 실행하면 모듈을 병합/최적화할 수 있다.

```
$ node build/r.js -o
  name=main                      # 모듈 이름
  baseUrl=src/js                  # 모듈 기본 주소( root 기준)
  mainConfigFile=src/js/main.js    # 메인 설정 파일 경로( root 기준)
  out=src/js/main-optimized.min.js # 아웃풋 파일 경로( root 기준)
  generateSourceMaps=true         # 소스맵 생성 설정
  preserveLicenseComments=false    # 라이선스 주석 보존 설정
  optimize=uglify2                # 최적화 엔진 설정
```





# Build Profile

앞서 다룬 명령어는 입력이 불편함으로 빌드 프로필 파일을 만들면 손쉽게 빌드를 처리할 수 있다. 방법은 아래처럼 빌드 옵션을 객체로 설정하면 된다. (빌드 파일에 상대 경로 사용)

```
/* build.config.js */
({
  name : "main",
  baseUrl : "../www/_/js/develop",
  paths : { requireLib: 'libs/require' },
  include : ["requireLib"],
  mainConfigFile : "../www/_/js/develop/main.js",
  out : "../www/_/js/app.min.js",
  optimize : "uglify2", // "none"
  generateSourceMaps : true,
  preserveLicenseComments : false
})
```

```
$ node r.js -o build.config.js # build.config.js 파일 옵션을 사용
```





## Build Profile

```
● ● ● 1. yamoo9@yamoo9-3: ~/Desktop/TASMA/step07-Optimize (zsh)
step07-Optimize > node build/r.js -o build/build.config.js

Tracing dependencies for: main
uglify2 file: /Users/yamoo9/Desktop/TASMA/step07-Optimize/js/main.bundle.min.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/main.bundle.min.js
-----
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/vender/jquery-2.1.4.min.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/app/model/tasmaModel.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/vender/require-handlebars-plugin/hbs/handlebars.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/vender/require-handlebars-plugin/hbs/underscore.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/vender/require-handlebars-plugin/hbs/i18nprecompile.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/vender/require-handlebars-plugin/hbs/json2.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/vender/require-handlebars-plugin/hbs.js
hbs!templates/tasmaTemplate
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/app/view/tasmaView.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/app/controller/tasmaController.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/app/tasma.js
/Users/yamoo9/Desktop/TASMA/step07-Optimize/js/main.js

    preserveLicenseComments : false
})
```

node r.js -o build.config.js # build.config.js 파일 옵션을 사용



# Build Profile

CSS 파일 또한 Javascripts 파일 병합/최적화 과정과 동일하게 처리 가능하다.  
아래처럼 명령어를 입력한 후, 동일한 방법을 사용한다.

```
/* build.css.config.js */
({
  cssIn : "../css/style.css",
  out   : "../css/style.min.css",
  optimizeCss: ""
})
```

```
$ node r.js -o build.css.config.js
```





# Build Profile

CSS 파일 또한 Javascripts 파일 병합/최적화 과정과 동일하게 처리 가능하다.

아래는 RequireJS를 사용하여 CSS 파일을 최적화하는 예제입니다.

```
1. yamoo9@yamoo9-3: ~/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO (zsh)
RequireJS-DEMO git:master > node build/r.js -o build/build.css.config.js
-----  
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/css/style.min.css  
-----  
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/css/base/reset.css  
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/css/base/common.css  
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/css/style.css
```

```
$ node r.js -o build.css.config.js
```

> -



# Shell Script

매번 명령창에 명령어를 입력하는 과정을 반복하지 않고 줄이려면 쉘스크립트를 사용한다.

아래와 같이 작성한 후, 명령창에서 쉘스크립트 파일을 실행한다.

```
# Shell Script
# -----
#!/usr/bin/env sh
# 퍼미션 권한 오류일 경우, 아래 명령어 수행
# chmod +x build/build.sh

node build/r.js -o build/build.config.js
node build/r.js -o build/build.css.config.js
```

```
$ build/build.sh
```





## Shell Script

```
1. yamoo9@yamoo9-3: ~/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO (zsh)
RequireJS-DEMO git:master > build/build.sh

Tracing dependencies for: main
Uglify2 file: /Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/js/main-bundle.min.js
-----  

/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/js/main-bundle.min.js
-----  

/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/js/vender/jquery-2.1.4.min.js
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/js/vender/modernizr-custom.min.js
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/js/vender/detectizr.js
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/js/main.js

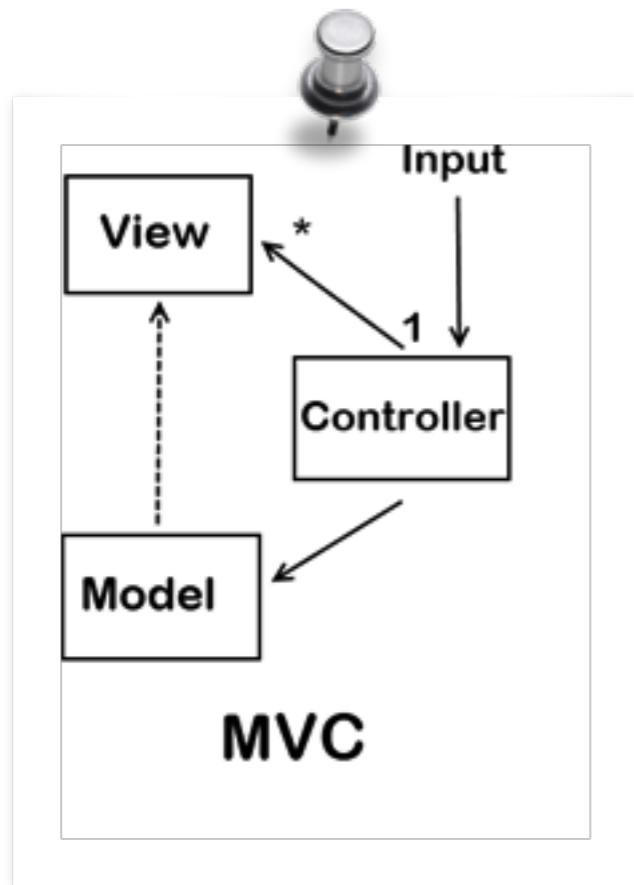
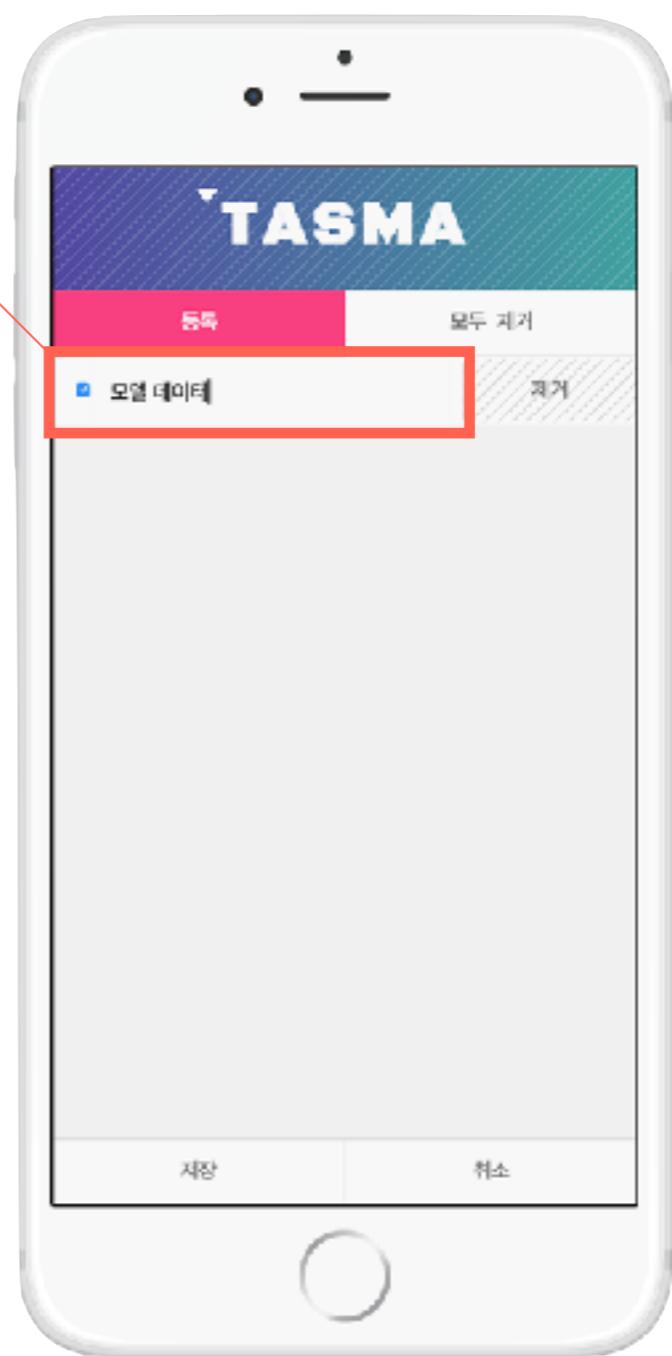
-----  

/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/css/style.min.css
-----  

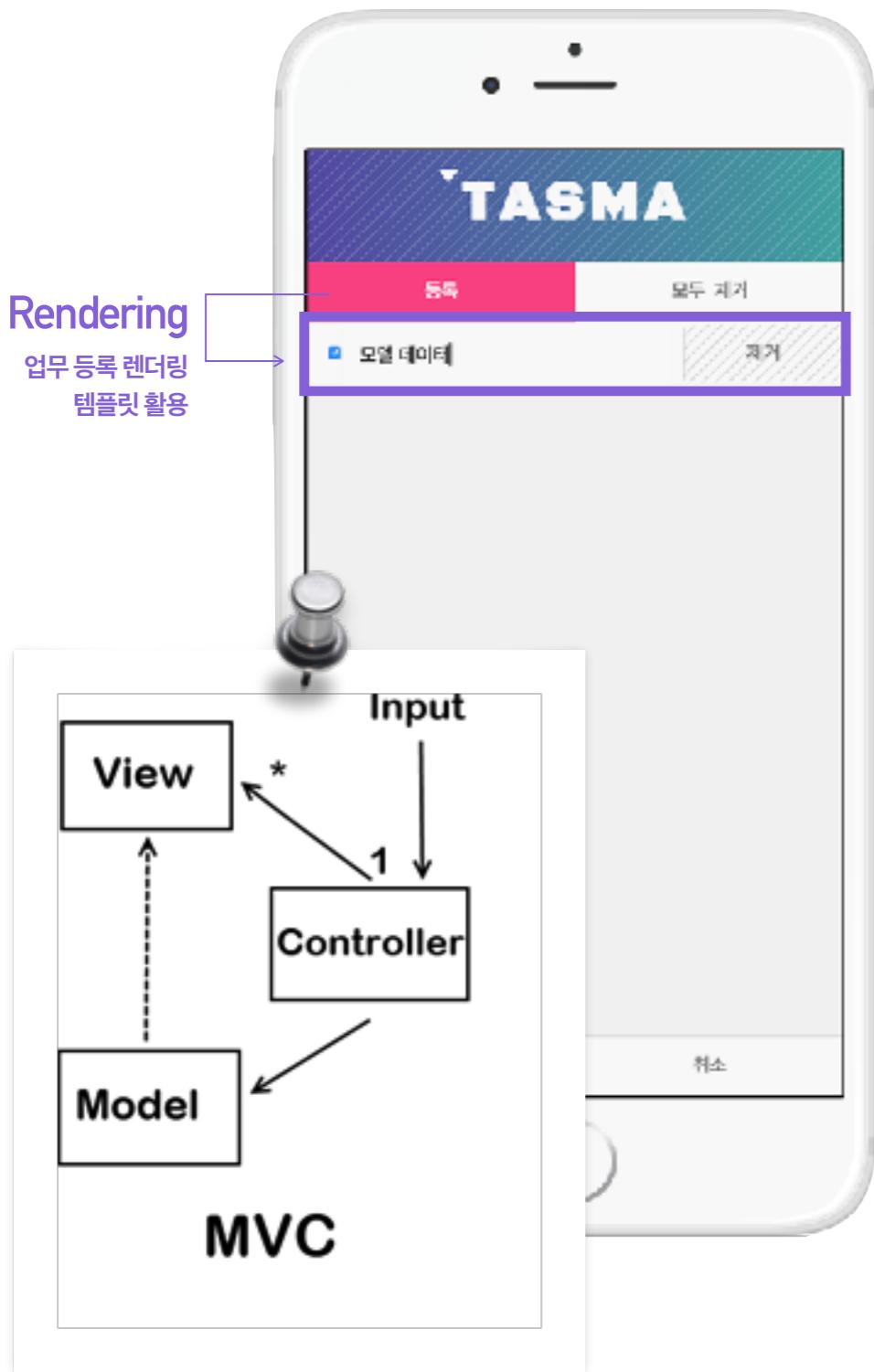
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/css/base/reset.css
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/css/base/common.css
/Users/yamoo9/Documents/GIT/fc-camp/DAY08/RequireJS-DEMO/css/style.css
```

```
$ build/build.sh
```

## Model

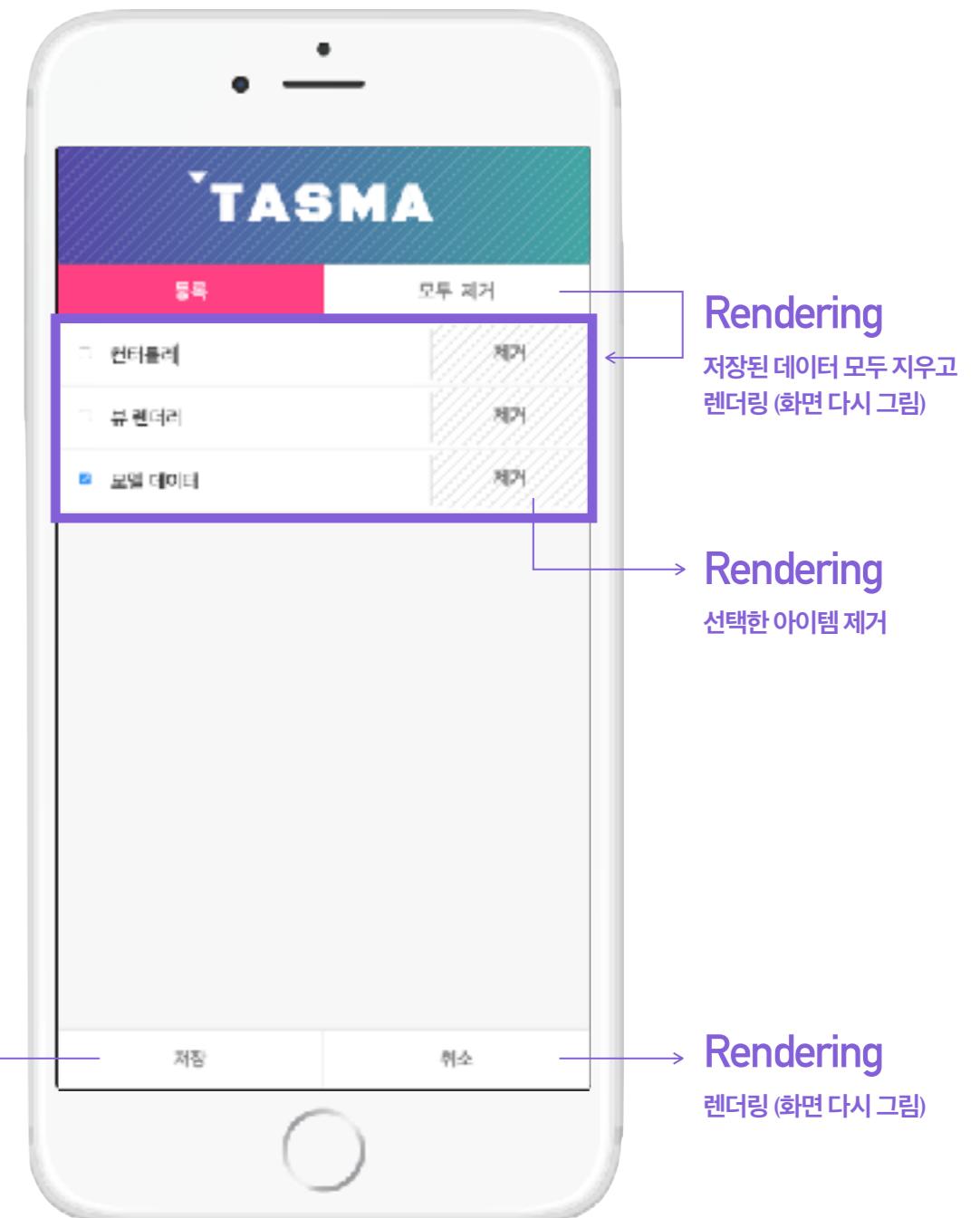


## View



Changing View

Rendering  
데이터 저장

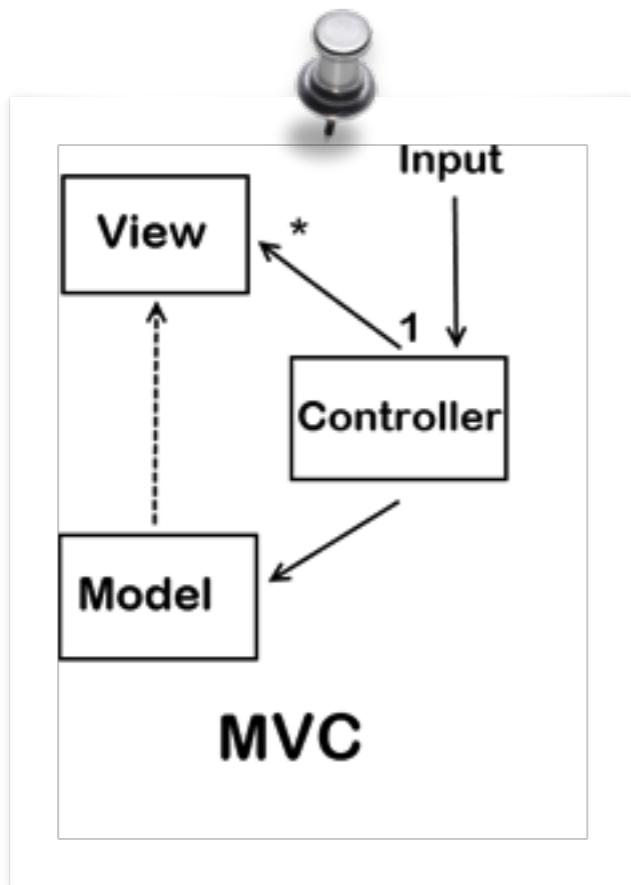
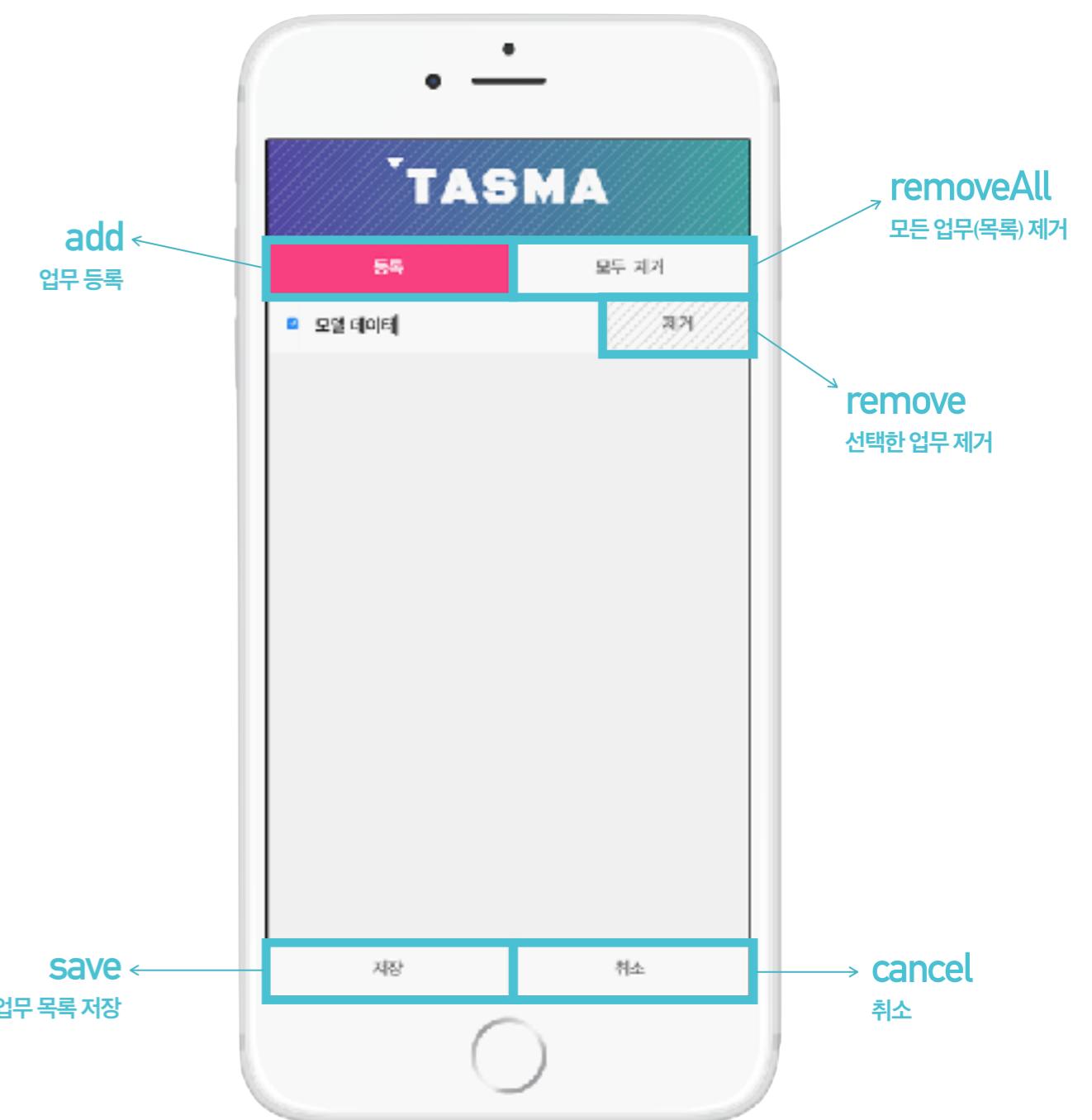


Rendering  
저장된 데이터 모두 지우고  
렌더링 (화면 다시 그림)

Rendering  
선택한 아이템 제거

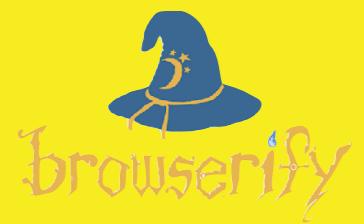
Rendering  
렌더링 (화면 다시 그림)

## Controller



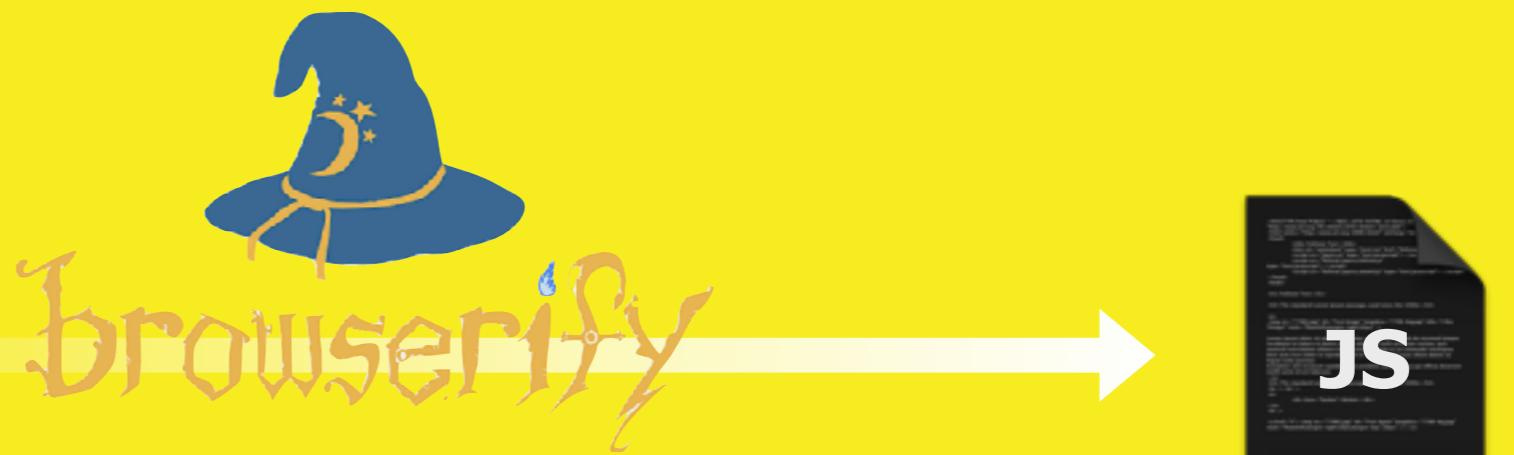


Browserify



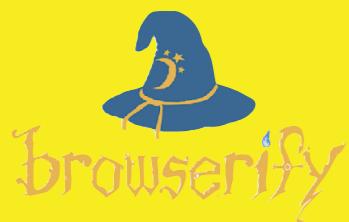
# browserify

browserify는 CommonJS 표준 명세를 따르는 모듈로더 라이브러리.  
Server-Side 기반의 문법을 사용하여 Client-Side 기반에 활용할 수 있도록 처리.  
결과물은 병합(Bundle, 뭉치)된 JS 파일을 만듬.



browserify input > output

bundle.js

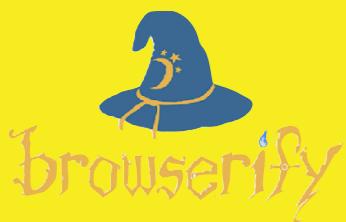


# Node.js

browserify는 CommonJS 표준 명세를 따르는 모듈 로더 라이브러리로 RequireJS와 달리 Node.js Server-Side 기반 환경을 필요로 한다. [nodejs.org](http://nodejs.org) 웹사이트에서 사용자 플랫폼에 적합한 설치 파일을 내려받아 설치한다.

The screenshot shows the Node.js Downloads page. At the top left, there's a sidebar with two main sections: 'LTS' (Mature and Dependable) and 'Stable' (Latest Features). The 'Stable' section is currently selected. Below the sidebar, there are four main download categories: 'Windows Installer', 'Macintosh Installer', and 'Source Code'. Under each category, there are two options: 32-bit and 64-bit. The 'Windows Installer' section also includes links for 'Windows Binary (.exe)' and 'Mac OS X Installer (.pkg)'. The 'Macintosh Installer' section includes links for 'Mac OS X Binaries (.tar.gz)'. The 'Source Code' section includes links for 'Linux Binaries (.tar.gz)', 'SunOS Binaries (.tar.gz)', 'ARM Binaries (.tar.gz)', 'Docker Image', and 'Source Code'. A large table at the bottom lists all these download links with their corresponding file names and bit architectures.

	32-bit	64-bit
Windows Installer (.msi)	<a href="#">Windows Installer (.msi)</a>	<a href="#">Windows Installer (.msi)</a>
Windows Binary (.exe)	<a href="#">Windows Binary (.exe)</a>	<a href="#">Windows Binary (.exe)</a>
Mac OS X Installer (.pkg)		<a href="#">Mac OS X Installer (.pkg)</a>
Mac OS X Binaries (.tar.gz)		<a href="#">Mac OS X Binaries (.tar.gz)</a>
Linux Binaries (.tar.gz)	<a href="#">Linux Binaries (.tar.gz)</a>	<a href="#">Linux Binaries (.tar.gz)</a>
SunOS Binaries (.tar.gz)	<a href="#">SunOS Binaries (.tar.gz)</a>	<a href="#">SunOS Binaries (.tar.gz)</a>
ARM Binaries (.tar.gz)	<a href="#">ARM Binaries (.tar.gz)</a>	<a href="#">ARM Binaries (.tar.gz)</a>
Docker Image		<a href="#">Official Node.js Docker Image</a>
Source Code		<a href="#">node-v4.2.2.tar.gz</a>



# browserify 설치

NPM(Node Package Manager) 설치(Install) 명령을 적용하여 전역(Global)에서 이용 가능하도록 browserify 모듈을 설치한다.

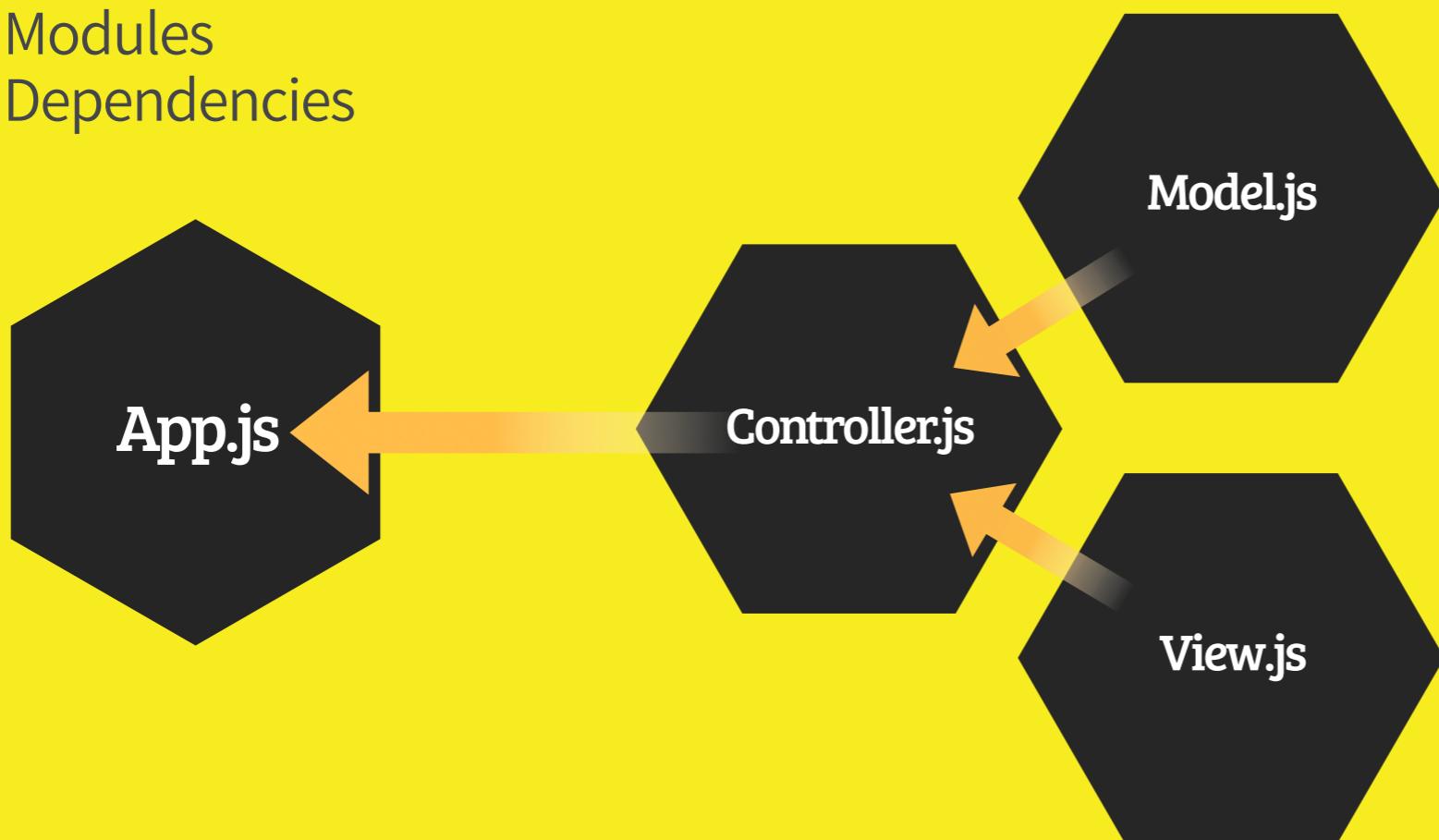
```
$ npm install --global browserify
```

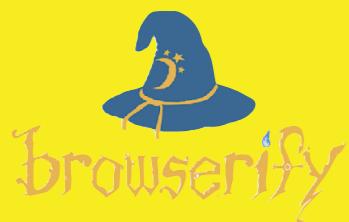
```
1. yamoo9@yamoo9-3: ~ (zsh)
~ > sudo npm install --global browserify
/usr/local/bin/browserify -> /usr/local/lib/node_modules/browserify/bin/cmd.js
browserify@12.0.1 /usr/local/lib/node_modules/browserify
|__ https-browserify@0.0.1
|__ tty-browserify@0.0.0
|__ htmlescape@1.1.0
|__ path-browserify@0.0.0
|__ os-browserify@0.1.2
|__ inherits@2.0.1
|__ string_decoder@0.10.31
|__ duplex@0.1.4
|__ through@2.0.0
|__ process@0.11.2
|__ constants-browserify@1.0.0
|__ isarray@0.0.1
|__ defined@1.0.0
|__ punycode@1.3.2
|__ domain-browser@1.1.4
|__ browser-resolve@1.10.1
```

# 모듈 의존성 관계도

애플리케이션의 개별 역할을 Model - View - Controller로 구분하여 모듈 간 의존성 관계를 설계(Design)한다.

Modules  
Dependencies





# CommonJS 모듈정의

CommonJS 표준 사양 모듈 정의에서는 `require, module, exports` 3개의 인자를 받는다.  
module.exports와 exports는 동일한 객체를 말하며 아래와 같이 모듈을 정의 한다.

The screenshot shows a browser window titled "yamoo9\_example.html". The code in the editor is:

```
1 function(require, module, exports) {
  'use strict';

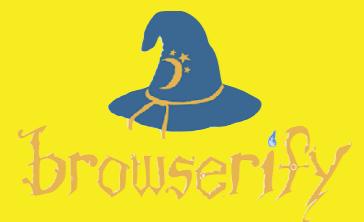
  // module.exports === exports

  module.exports = {
    'method1' : function(){}
    'method2' : function(){}
    'method3' : function(){}
  };
}

}
```

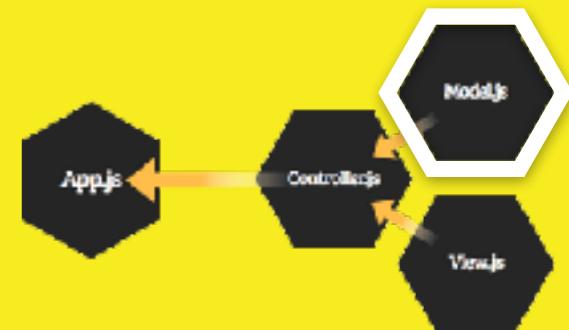
A black rectangular callout box with the Korean word "동일" (same) written on it points from the text "module.exports === exports" to the line "exports.method1 = function(){};" in the code.

exports.method1 = function(){};  
exports.method2 = function(){};  
exports.method3 = function(){};



# Model 모듈

localStorage 객체 메소드를 이용하여 데이터(Data)를 사용자의 브라우저에 반영구적으로 저장하는 모델 모듈(Model Module) - JSON 형식으로 데이터를 저장/읽기/지우기



```
1  'use strict';

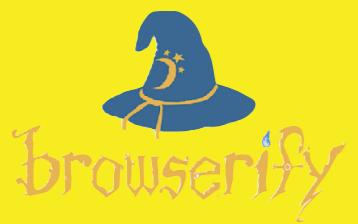
2  var DATA_NAME = 'TASMA';

3  exports.save = function(data) {
4      data = JSON.stringify( data );
5      localStorage.setItem( DATA_NAME, data );
6  };

7  exports.load = function() {
8      var data = localStorage.getItem( DATA_NAME );
9      if (data) {
10          return JSON.parse( data );
11      }
12      return [];
13  };

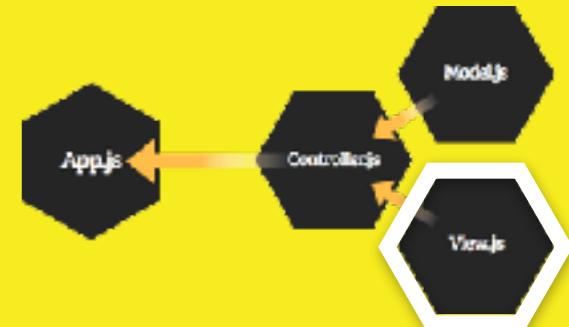
14  exports.clear = function() {
15      localStorage.removeItem( DATA_NAME );
16  };

```



# View 모듈

뷰 모듈에서는 모델 데이터를 전달받아, 템플릿 코드를 토대로 동적으로 코드를 생성.  
화면에 렌더링되도록 설정한다.



```
+ * yahoo9_example.html          $ npm install --save-dev jquery
1
use strict;

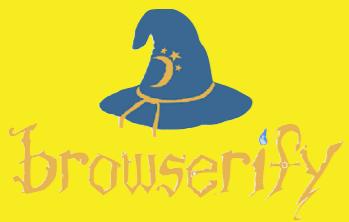
// jQuery 의존 모듈 호출
var $ = require('jquery'); ←

var render_template = [
  '<li class="task clearfix">',
  '  <div class="task-desc">',
  '    <input type="checkbox" class="complete">',
  '    <input type="text" class="description" placeholder="등록할 내용을 기입해주세요.">',
  '  </div>',
  '  <button type="button" class="anim button remove">제거</button>',
  '</li>'
].join('');

var $task_list = $$tasma.find('.tasks-list');

var _renderTask = function(task) {
  var $template = $( render_template );
  if ( task.complete ) {
    $template.find('.complete').attr('checked', 'checked');
  }
}
```

A screenshot of a terminal window showing the command \$ npm install --save-dev jquery being run. The output shows the package jquery@2.1.4 installed in the node\_modules directory. An orange arrow points from the line var \$ = require('jquery'); in the code editor to the terminal output, indicating the dependency.



```
1 'use strict';

// jQuery 의존 모듈 호출
var $ = require('jquery');

var render_template = [
  '<li class="task clearfix">',
    '<div class="task-desc">',
      '<input type="checkbox" class="complete">',
      '<input type="text" class="description" placeholder="등록할 내용을 기입해주세요.">',
    '</div>',
    '<button type="button" class="anim button remove">제거</button>',
  '</li>'
].join('');

var $task_list = $$tasma.find('.tasks-list');

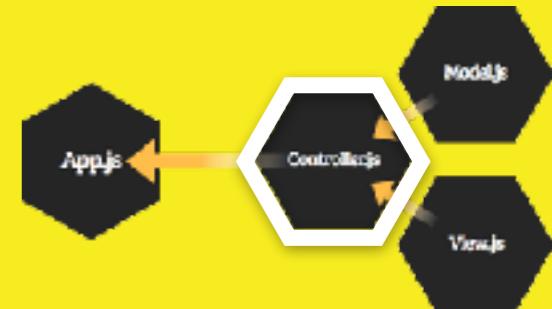
var _renderTask = function(task) {
  var $template = $( render_template );
  if ( task.complete ) {
    $template.find('.complete').attr('checked', 'checked');
  }
  $template.find('.description').val( task.description );
  return $template;
};

exports.renderTasks = function(tasks) {
  var $elsArray = $.map( tasks, _renderTask );
  $task_list.empty().append( $elsArray );
};

exports.renderNew = function() {
  console.log( $task_list[0] );
  $task_list.prepend( _renderTask( {} ) );
};
```

# Controller 모듈

컨트롤러 모듈에서는 모델 데이터/뷰 렌더러 모듈을 호출한 후,  
이벤트에 따라 콜백 될 함수를 정의한다.



```

yamoo9_example.html

1 'use strict';

var $           = require('jquery'),
    TaskData    = require('./taskModel'),
    TaskRenderer = require('./taskView');

exports.render = function() {
    TaskRenderer.renderTasks( TaskData.load() );
};

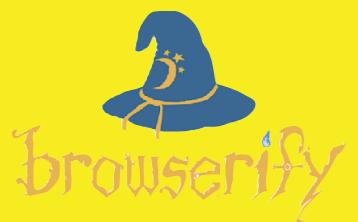
exports.add = function() {
    TaskRenderer.renderNew();
};

exports.clearAll = function() {
    TaskData.clear();
    exports.render();
};

exports.remove = function(ev) {
    var $target = $( ev.target );
    $target.closest('.task').remove();
};

```

require() 함수 인자 값 작성 시, `./` 를 제외하면  
`node\_modules/` 안에서 해당 모듈을 찾음.  
  
TaskData=require('data/taskModel')  
TaskRenderer=require('renderer/taskView')



```
1  'use strict';

var $          = require('jquery'),
    TaskData    = require('./taskModel'),
    TaskRenderer = require('./taskView');

exports.render = function() {
  TaskRenderer.renderTasks( TaskData.load() );
};

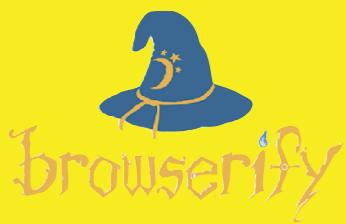
exports.add = function() {
  TaskRenderer.renderNew();
};

exports.clearAll = function() {
  TaskData.clear();
  exports.render();
};

exports.remove = function(ev) {
  var $target = $( ev.target );
  $target.closest('.task').remove();
};

exports.save = function() {
  var tasks = [], $tasks = $$tasma.find('.task');
  $.each($tasks, function(index, task) {
    var $task = $tasks.eq(index);
    tasks.push({
      'complete' : $task.find('.complete').prop('checked'),
      'description' : $task.find('.description').val()
    });
  });
  TaskData.save( tasks );
};

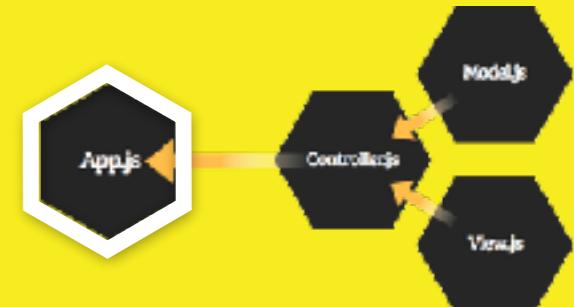
exports.cancel = function() {
  exports.render();
};
```



# App 모듈

앱 모듈에서는 컨트롤러 모듈을 호출한 후, 이벤트에 핸들러를 연결(Bind)한다.

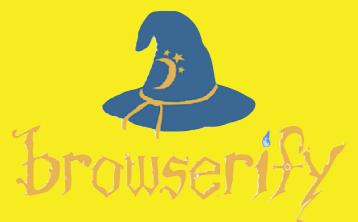
초기 실행에 필요한 동작을 init() 함수로 정의한 후, DOM이 완성되는 시점에 콜백(Callback)한다.



```
1  'use strict';

2  var $      = require('jquery'),
3      Tasks = require('./app/taskController');

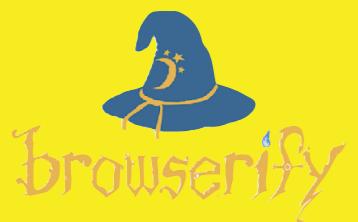
4  function _addTask() {
5      Tasks.add();
6  }
7  function _clearAllTasks() {
8      Tasks.clearAll();
9  }
10 function _saveChange() {
11     Tasks.save();
12 }
13 function _cancelChanges() {
14     Tasks.cancel();
15 }
16 function _deleteTask(ev) {
17     Tasks.remove(ev);
18 }
19 function _registerEventHandler() {
20     $('#taskList').on('click', '#newTask', addTask);
21 }
```



```
1  'use strict';

var $      = require('jquery'),
    Tasks = require('./app/taskController');

function _addTask() {
    Tasks.add();
}
function _clearAllTasks() {
    Tasks.clearAll();
}
function _saveChange() {
    Tasks.save();
}
function _cancelChanges() {
    Tasks.cancel();
}
function _deleteTask(ev) {
    Tasks.remove(ev);
}
function _registerEventHandler() {
    $('.tasma').find('.button.new').on('click', _addTask);
    $('.tasma').find('.button.new').on('click');
    $('.tasma').find('.button.remove-all').on('click', _clearAllTasks);
    $('.tasma').find('.button.save').on('click', _saveChange);
    $('.tasma').find('.button.cancel').on('click', _cancelChanges);
    $('.tasma').find('.tasks-list').on('click', '.remove', _deleteTask);
}
function init() {
    $('.tasma') = $('#TASMA');
    _registerEventHandler();
    Tasks.render();
}
// DOM Ready
$(init);
```



# browserify 빌드

전역(Global)에 설치된 browserify 명령을 이용하여 각 모듈을 하나의 번들(Bundle) 파일로 변경한다.  
이 과정에서 CommonJS의 MTF(Module Transport Format) 형식으로 감싸게(Wrapper) 된다.

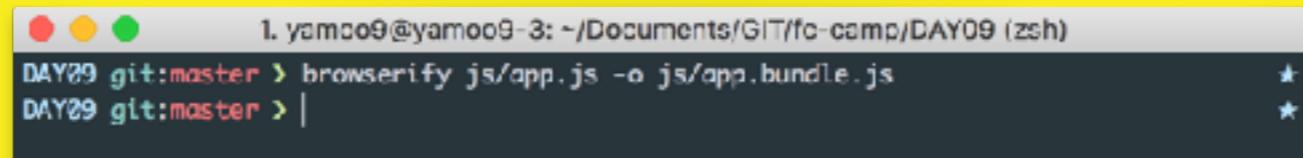
```
$ browserify js/app.js -o js/app.bundle.js
```



**browserify** input > output

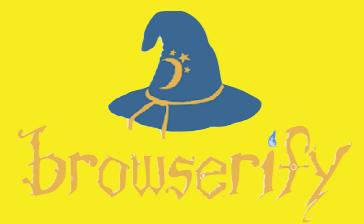


**bundle.js**

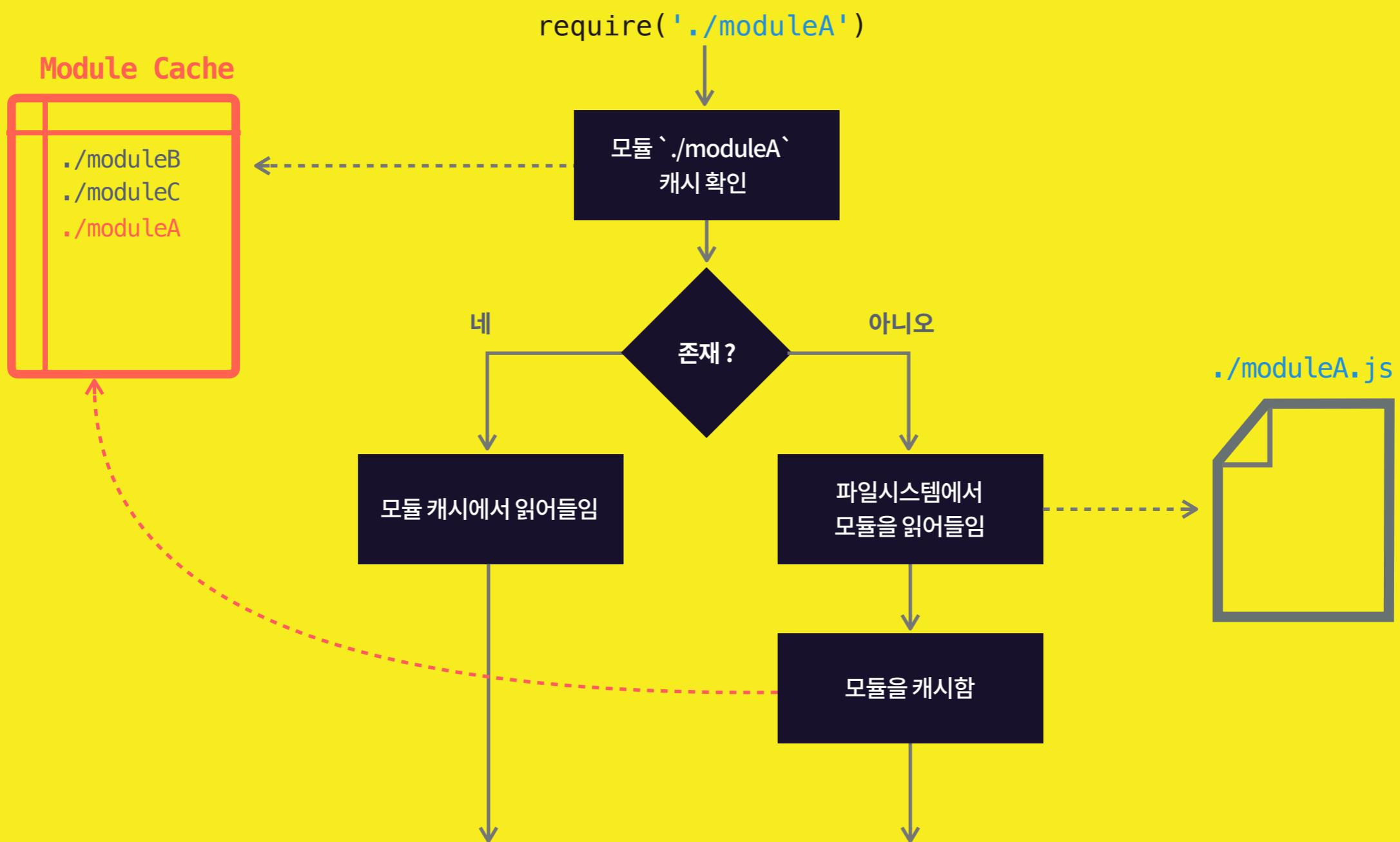


```
app.bundle.js x

1 (function e(t,n,r){function s(o,u){if(!n[o]){if(!t[o]){var a=typeof require=="function"&&require;if(!u&&a) return a(o,!0);if(i) return i(o,!0);var f=new Error("Cannot find module '"+o+"'");throw f.code="MODULE_NOT_FOUND",f}var l=n[o]={exports:{}},t[o][0].call(l.exports,function(e){var n=t[o][1][e];return s(n?n:e)},l,l.exports,e,t,n,r)}return n[o].exports}var i=typeof require=="function"&&require;for(var o=0;o<r.length;o++)s(r[o]);return s})({1:[function(require,module,exports){
2 'use strict';
3
```



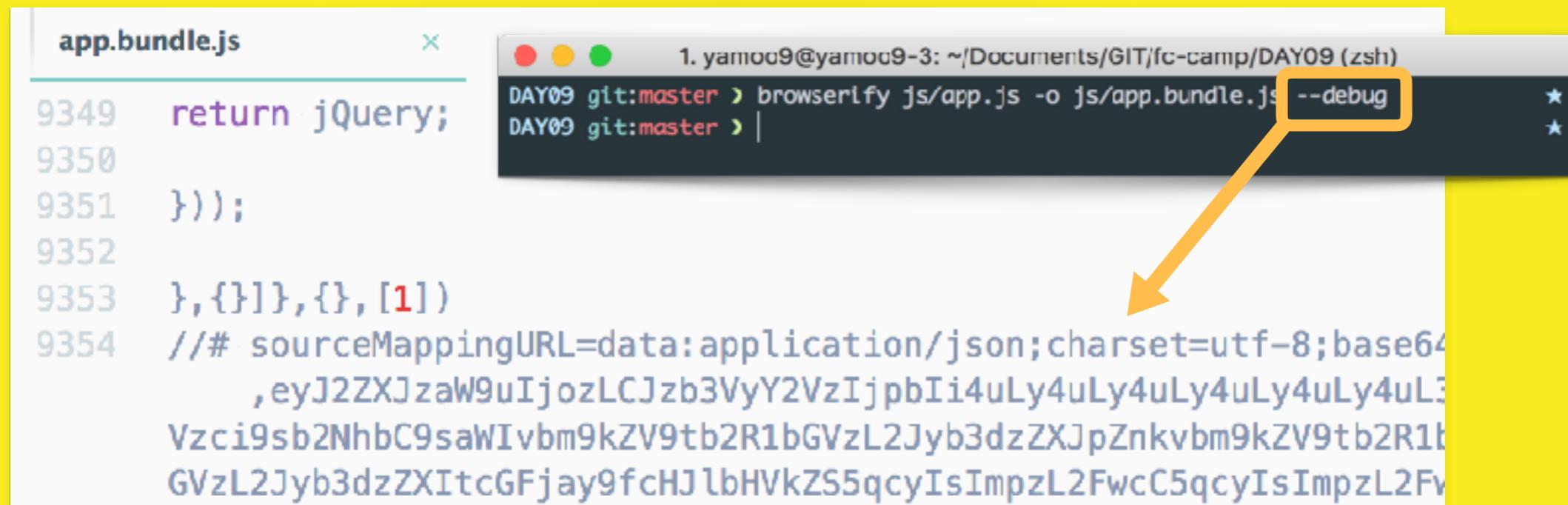
# 모듈 캐시



# Sourcemap 설정

Sourcemap 파일은 Bundle된 파일에서 찾기 힘든 잘못된 부분의 Original 파일 코드를 살펴볼 수 있도록 해주는 Map 파일을 말한다.

```
$ browserify js/app.js -o js/app.bundle.js --debug
```



The screenshot shows a terminal window with the command:

```
1. yamoo9@yamoo9-3: ~/Documents/GIT/fc-camp/DAY09 (zsh)
DAY09 git:master > browserify js/app.js -o js/app.bundle.js --debug
DAY09 git:master > |
```

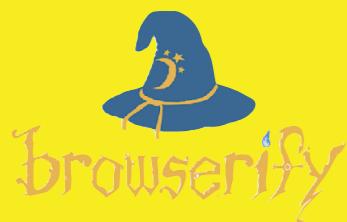
An orange arrow points from the terminal output to a code editor window titled "app.bundle.js". The code editor displays the following content:

```
9349 return jQuery;
9350
9351 });
9352
9353 }, {}, {}, [1])
9354 //# sourceMappingURL=data:application/json;charset=utf-8;base64,
         ,eyJ2ZXJzaW9uIjozLCJzb3VyY2VzIjpibIi4uLy4uLy4uLy4uLy4uL3
Vzci9sb2NhbC9saWIvbm9kZV9tb2R1bGVzL2Jyb3dzZXJpZnkvbm9kZV9tb2R1b
GVzL2Jyb3dzZXItcGFjay9fcHJlbHVkZS5qcyIsImpzL2FwcC5qcyIsImpzL2Fv
```

The screenshot shows the Chrome DevTools interface with the Sources tab selected. The left sidebar lists file origins: file://, Users, fonts.googleapis.com, and DAY04. The main pane displays the code for app.js:

```
5 function _addTask() {  
6   Tasks.add();  
7 }  
8  
9 function _clearAllTasks() {  
10  Tasks.removeAll(); ✖  
11 }  
12  
13 function _saveChanges() {  
14   Tasks.save();  
15 }
```

An orange arrow points from the error message in the bottom status bar to the line of code in the editor where `Tasks.removeAll()` is called. The status bar also shows the error message: `Uncaught TypeError: Tasks.removeAll is not a function` at `app.bundle.js:11`.



# watchify 설정

watchify 설정이 하는 일은 파일이 변경됨을 감지하여, 변경될 때마다 다시 browserify 명령을 실행한다.

-v는 --verbose 옵션의 약자로 ‘수다스럽게’ 명령창 콘솔 화면에 번들링 결과를 출력한다.

```
$ npm install --global watchify  
$ watchify js/app.js -o js/app.bundle.js --debug -v
```

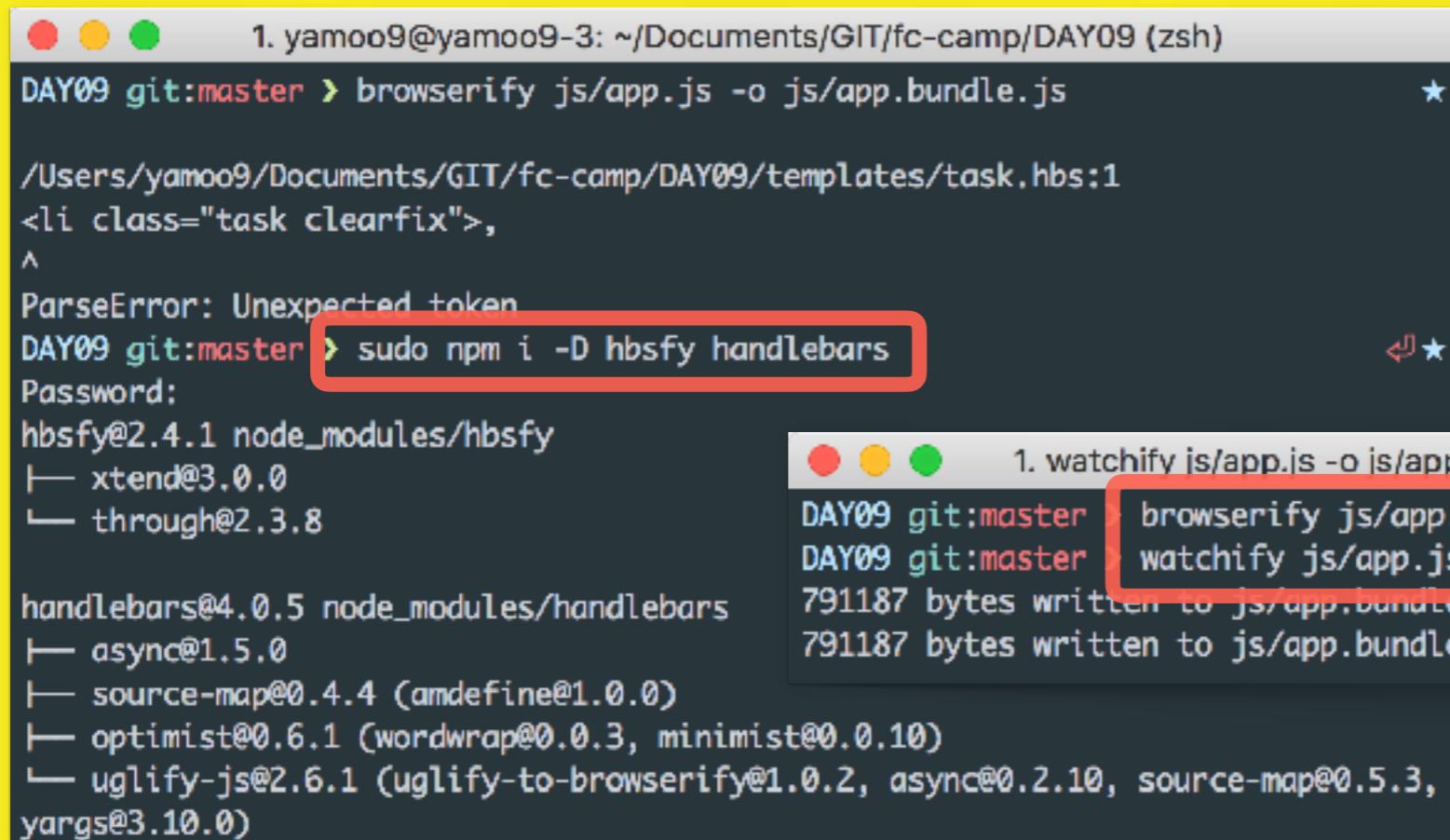
```
● ● ● 1. yamoo9@yamoo9-3: ~/Documents/GIT/fc-camp/DAY09 (zsh)  
DAY09 git:master > sudo npm i -g browserify ★  
Password:  
/usr/local/bin/browserify -> /usr/local/lib/node_modules/browserify/bin/cmd.js  
browserify@12.0.1 /usr/local/lib/node_modules/browserify  
|--- https-browserify@0.0.1  
|--- htmlEscape@1.1.0
```

```
● ● ● 1. watchify js/app.js -o js/app.bundle.js --debug -v (node)  
DAY09 git:master > watchify js/app.js -o js/app.bundle.js --debug -v ★  
691450 bytes written to js/app.bundle.js (0.64 seconds)  
691449 bytes written to js/app.bundle.js (0.07 seconds)
```

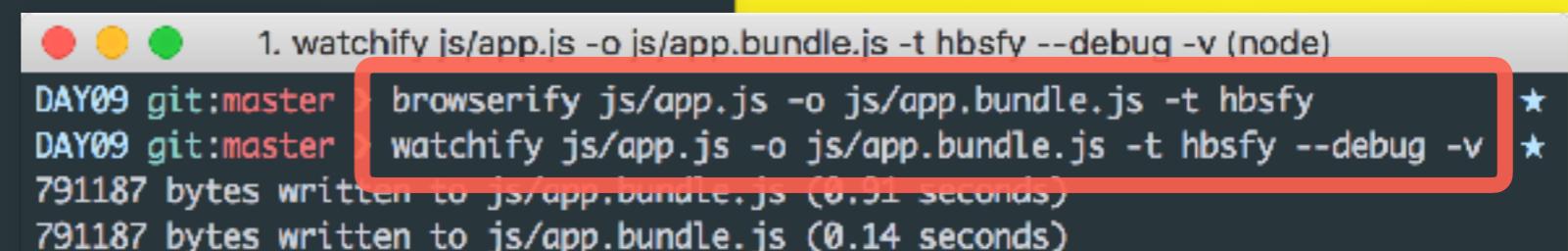
# Transform 설정

Transform 설정 중, Handlebars 플러그인을 사용해 템플릿을 분리 관리할 수 있다.

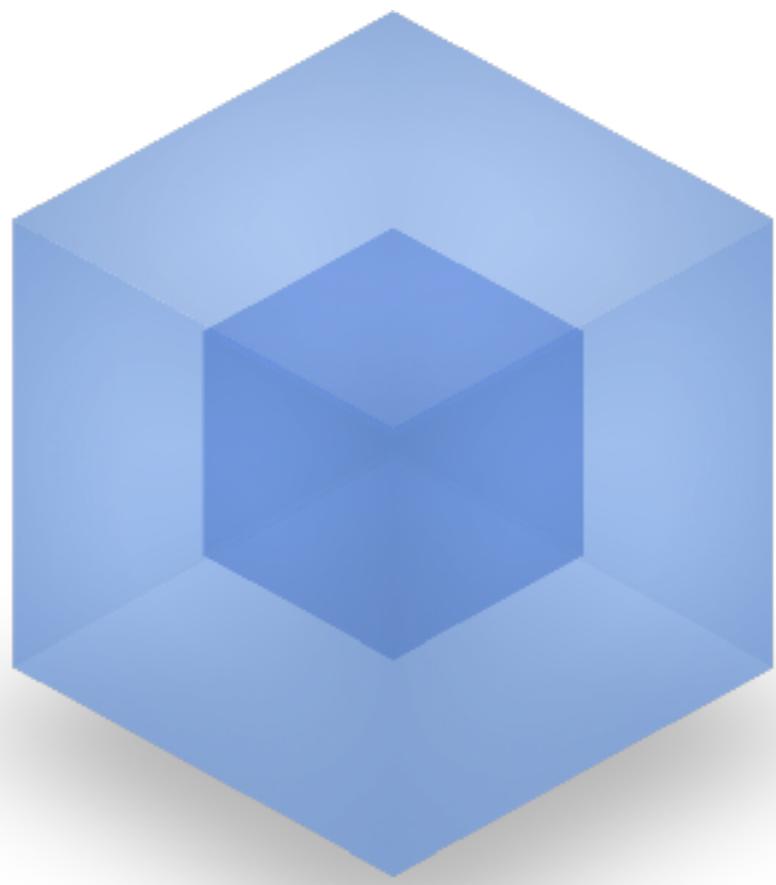
```
$ npm install --save-dev hbsfy handlebars  
$ watchify js/app.js -o js/app.bundle.js -t hbsfy --debug -v
```



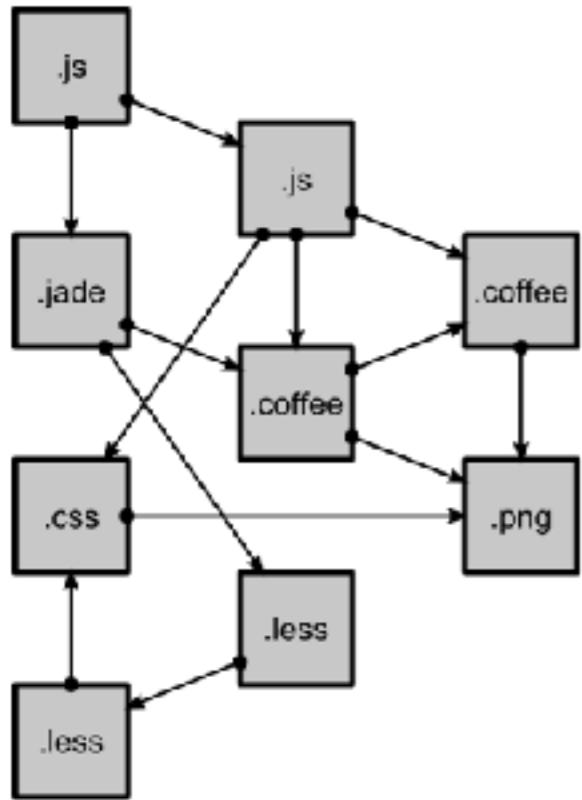
```
1. yamoo9@yamoo9-3: ~/Documents/GIT/fc-camp/DAY09 (zsh)  
DAY09 git:master > browserify js/app.js -o js/app.bundle.js ★  
  
/Users/yamoo9/Documents/GIT/fc-camp/DAY09/templates/task.hbs:1  
<li class="task clearfix">,  
^  
ParseError: Unexpected token  
DAY09 git:master > sudo npm i -D hbsfy handlebars  
Password:  
hbsfy@2.4.1 node_modules/hbsfy  
└── xtend@3.0.0  
  └── through@2.3.8  
  
handlebars@4.0.5 node_modules/handlebars  
└── async@1.5.0  
  └── source-map@0.4.4 (amdefine@1.0.0)  
    ├── optimist@0.6.1 (wordwrap@0.0.3, minimist@0.0.10)  
    └── uglify-js@2.6.1 (uglify-to-browserify@1.0.2, async@0.2.10, source-map@0.5.3,  
      args@3.10.0)
```



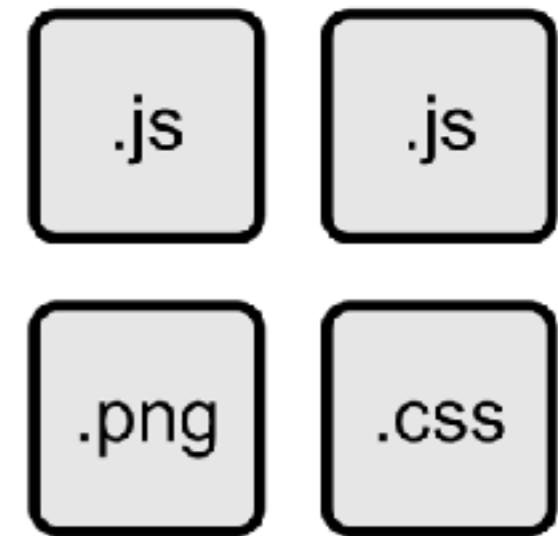
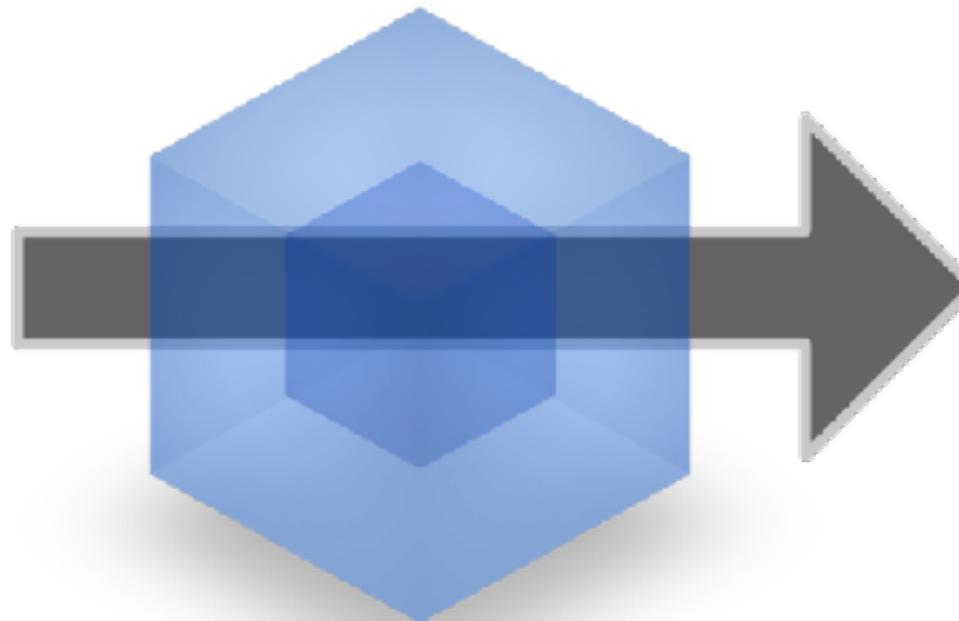
```
1. watchify js/app.js -o js/app.bundle.js -t hbsfy --debug -v (node)  
DAY09 git:master > browserify js/app.js -o js/app.bundle.js -t hbsfy ★  
DAY09 git:master > watchify js/app.js -o js/app.bundle.js -t hbsfy --debug -v ★  
791187 bytes written to js/app.bundle.js (0.91 seconds)  
791187 bytes written to js/app.bundle.js (0.14 seconds)
```



**webpack**  
MODULE BUNDLER

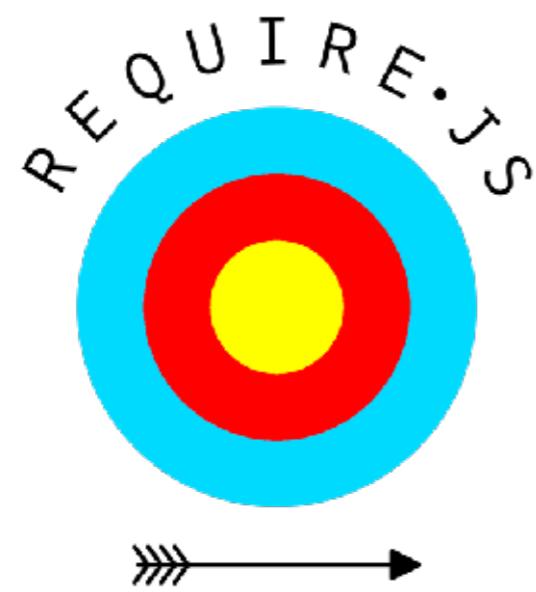


modules  
with dependencies



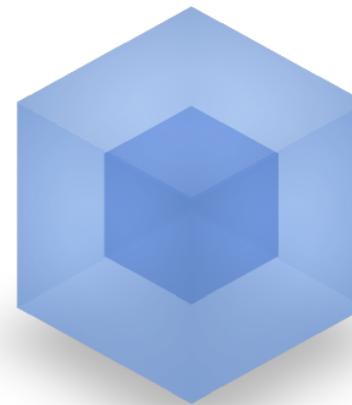
static  
assets

webpack은 모듈 번들러(Module Bundler)로 의존성을 가진 모듈들을 다루고,  
그 모듈로 부터 정적인 에셋(Asset)을 생성한다.



모듈 로더(Module Loader)

VS



**webpack**  
MODULE BUNDLER

모듈 번들러(Module Bundler)