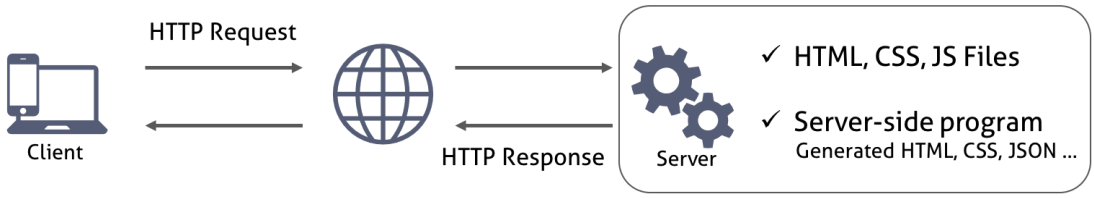
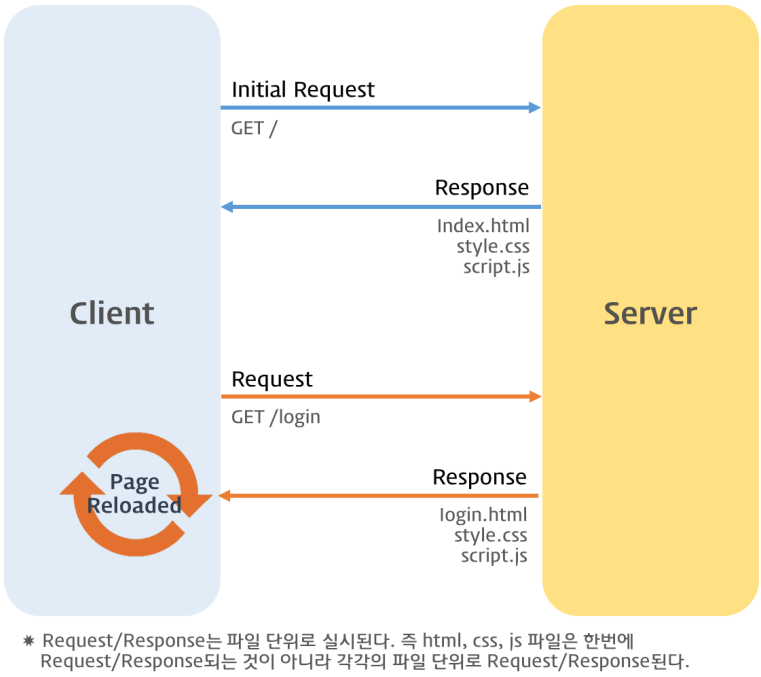
Javascript 3

Ajax(Asynchronous JavaScript and XML)

브라우저에서 웹페이지를 요청하거나 링크를 클릭하면 화면 갱신이 발생한다. 이것은 브라우저와 서버와의 통신에 의한 것이다.



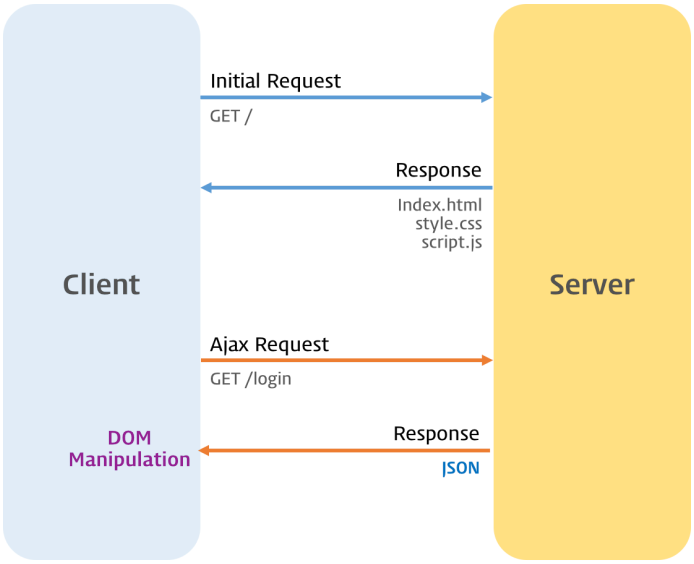
서버는 요청받은 페이지(HTML)를 반환하는데 이때 HTML에서 로드하는 CSS나 JavaScript 파일들도 같이 반환된다. 클라이언트의 요청에 따라 서버는 정적인 파일을 반환할 수도 있고 서버 사이드 프로그램이 만들어낸 파일이나 데이터를 반환할 수도 있다. 서버로부터 웹페이지가 반환되면 클라이언트(브라우저)는 이를 렌더링하여 화면에 표시한다.



Traditional Web Page Lifecycle

Ajax(Asynchronous JavaScript and XML)는 자바스크립트를 이용해서 **비동기적(Asynchronous)**으로 서버와 브라우저가 데이터를 교환할 수 있는 통신 방식을 의미한다.

서버로부터 웹페이지가 반환되면 화면 전체를 갱신해야 하는데 페이지 일부만을 갱신하고도 동일한 효과를 볼 수 있도록 하는 것이 Ajax이다. 페이지 전체를 로드하여 렌더링할 필요가 없고 갱신이 필요한 일부만 로드하여 갱신하면 되므로 빠른 퍼포먼스와 부드러운 화면 표시 효과를 기대할 수 있다.



✷ Request/Response는 파일 단위로 실시된다. 즉 html, css, js 파일은 한번에 Request/Response되는 것이 아니라 각각의 파일 단위로 Request/Response된다.

Ajax Lifecycle

JSON(JavaScript Object Notation)

클라이언트와 서버 간에는 데이터 교환이 필요하다. JSON(JavaScript Object Notation)은 클라이언트와 서버 간 데이터 교환을 위한 규칙 즉 데이터 포맷을 말한다.

JSON은 일반 텍스트 포맷보다 효과적인 데이터 구조화가 가능하며 XML 포맷보다 가볍고 사용하기 간편하며 가독성도 좋다.

자바스크립트의 객체 리터럴과 매우 흡사하다. 하지만 **JSON은 순수한 텍스트로 구성된 규칙이 있는 데이터 구조이다.**

```
{
  "name": "Lee",
  "gender": "male",
  "age": 20,
  "alive": true
}
```

키는 반드시 큰따옴표(작은따옴표 사용불가)로 둘러싸야 한다.

JSON.stringify

JSON.stringify 메소드는 객체를 JSON 형식의 문자열로 변환한다.

```
const o = { name: 'Lee', gender: 'male', age: 20 };

// 객체 => JSON 형식의 문자열
const strObject = JSON.stringify(o);
console.log(typeof strObject, strObject);
// string {"name":"Lee","gender":"male","age":20}

// 객체 => JSON 형식의 문자열 + prettify
const strPrettyObject = JSON.stringify(o, null, 2);
console.log(typeof strPrettyObject, strPrettyObject);
/*
string {
  "name": "Lee",
  "gender": "male",
  "age": 20
}
*/

// replacer
// 값의 타입이 Number이면 필터링되어 반환되지 않는다.
function filter(key, value) {
  // undefined: 반환하지 않음
```

```
  return typeof value === 'number' ? undefined : value;
}

// 객체 => JSON 형식의 문자열 + replacer + prettify
const strFilteredObject = JSON.stringify(o, filter, 2);
console.log(typeof strFilteredObject, strFilteredObject);
/*
string {
  "name": "Lee",
  "gender": "male"
}
*/

const arr = [1, 5, 'false'];

// 배열 객체 => 문자열
const strArray = JSON.stringify(arr);
console.log(typeof strArray, strArray); // string [1,5,"false"]

// replacer
// 모든 값을 대문자로 변환된 문자열을 반환한다
function replaceToUpper(key, value) {
  return value.toString().toUpperCase();
}

// 배열 객체 => 문자열 + replacer
const strFilteredArray = JSON.stringify(arr, replaceToUpper);
console.log(typeof strFilteredArray, strFilteredArray); // string "1,5,FALSE"
```

JSON.parse

JSON 데이터를 가진 문자열을 객체로 변환한다.

서버로부터 브라우저로 전송된 JSON 데이터는 문자열이다. 이 문자열을 객체로서 사용하려면 객체화하여야 하는데 이를 역직렬화(Deserializing)이라 한다. 역직렬화를 위해서 내장 객체 JSON의 static 메소드인 JSON.parse를 사용한다.

```
const o = { name: 'Lee', gender: 'male', age: 20 };

// 객체 => JSON 형식의 문자열
const strObject = JSON.stringify(o);
console.log(typeof strObject, strObject);
// string {"name":"Lee","gender":"male","age":20}

const arr = [1, 5, 'false'];

// 배열 객체 => 문자열
const strArray = JSON.stringify(arr);
console.log(typeof strArray, strArray); // string [1,5,"false"]

// JSON 형식의 문자열 => 객체
const obj = JSON.parse(strObject);
console.log(typeof obj, obj); // object { name: 'Lee', gender: 'male' }

// 문자열 => 배열 객체
const objArray = JSON.parse(strArray);
console.log(typeof objArray, objArray); // object [1, 5, "false"]
```

배열이 JSON 형식의 문자열로 변환되어 있는 경우 JSON.parse는 문자열을 배열 객체로 변환한다. 배열의 요소가 객체인 경우 배열의 요소까지 객체로 변환한다.

```
const todos = [
  { id: 1, content: 'HTML', completed: true },
  { id: 2, content: 'CSS', completed: true },
  { id: 3, content: 'JavaScript', completed: false }
];

// 배열 => JSON 형식의 문자열
const str = JSON.stringify(todos);
console.log(typeof str, str);

// JSON 형식의 문자열 => 배열
const parsed = JSON.parse(str);
console.log(typeof parsed, parsed);
```

XMLHttpRequest

브라우저는 **XMLHttpRequest 객체**를 이용하여 Ajax 요청을 생성하고 전송한다. 서버가 브라우저의 요청에 대해 응답을 반환하면 같은 XMLHttpRequest 객체가 그 결과를 처리한다.

Ajax request

```
// XMLHttpRequest 객체의 생성
const xhr = new XMLHttpRequest();
// 비동기 방식으로 Request를 오픈한다
xhr.open('GET', '/users');
// Request를 전송한다
xhr.send();
```

XMLHttpRequest.open

XMLHttpRequest 객체의 인스턴스를 생성하고 XMLHttpRequest.open 메소드를 사용하여 서버로의 요청을 준비한다. XMLHttpRequest.open의 사용법은 아래와 같다.

```
XMLHttpRequest.open(method, url[, async])
```

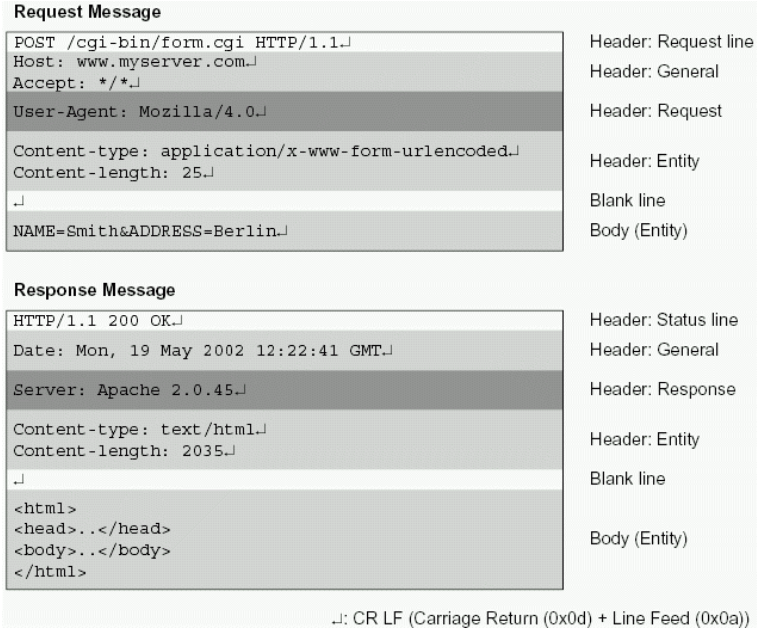
매개변수	설명
method	HTTP method('GET","POST","PUT","DELETE"등)
url	요청을 보낼 URL
async	비동기 조작 여부, 옵션으로 default는 true이며 비동기 방식으로 동작한다.

XMLHttpRequest.send

XMLHttpRequest.send 메소드로 준비된 요청을 서버에 전달한다.

기본적으로 서버로 전송하는 데이터는 GET, POST 메소드에 따라 그 전송 방식에 차이가 있다.

- GET 메소드의 경우, URL의 일부분인 쿼리문자열(query string)로 데이터를 서버로 전송한다.
- POST 메소드의 경우, 데이터(페이로드)를 Request Body에 담아 전송한다.



Example of a HTTP Request/Response message pair

XMLHttpRequest.send 메소드에는 **request body**에 담아 전송할 인수를 전달할 수 있다.

```
xhr.send(null);
// xhr.send('string');
// xhr.send(new Blob()); // 파일 업로드와 같이 바이너리 컨텐트를 보내는 방법
// xhr.send({ form: 'data' });
// xhr.send(document);
```

만약 **요청 메소드가 GET인 경우, send 메소드의 인수는 무시되고 request body은 null로 설정된다.**

- HTTP GET with request body

XMLHttpRequest.setRequestHeader

XMLHttpRequest.setRequestHeader 메소드는 HTTP Request Header의 값을 설정한다. setRequestHeader 메소드는 반드시 XMLHttpRequest.open 메소드 호출 이후에 호출한다.

자주 사용하는 Request Header인 Content-type, Accept에 대해 살펴보자.

Content-type

Content-type은 request body에 담아 전송할 데이터의 MIME-type의 정보를 표현한다. 자주 사용되는 MIME-type은 아래와 같다.

타입	서브타입
text 타입	text/plain, text/html, text/css, text/javascript
Application 타입	application/json, application/x-www-form-urlencode
File을 업로드하기 위한 타입	multipart/formed-data

다음은 request body에 담아 서버로 전송할 데이터의 MIME-type을 지정하는 예이다.

```
// json으로 전송하는 경우
xhr.open('POST', '/users');

// 클라이언트가 서버로 전송할 데이터의 MIME-type 지정: json
xhr.setRequestHeader('Content-type', 'application/json');

const data = { id: 3, title: 'JavaScript', author: 'Park', price: 5000};

xhr.send(JSON.stringify(data));
```

```
// x-www-form-urlencoded으로 전송하는 경우
xhr.open('POST', '/users');

// 클라이언트가 서버로 전송할 데이터의 MIME-type 지정: x-www-form-urlencoded
// application/x-www-form-urlencoded는 key=value&key=value...의 형태로 전송
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

const data = { title: 'JavaScript', author: 'Park', price: 5000};

xhr.send(Object.keys(data).map(key => `${key}=${data[key]}`).join('&'));
// escaping untrusted data
// xhr.send(Object.keys(data).map(key => `${key}=${encodeURIComponent(data[key])}`).join('&'));
```

Accept

HTTP 클라이언트가 서버에 요청할 때 서버가 센드백할 데이터의 MIME-type을 Accept로 지정할 수 있다.

다음은 서버가 센드백할 데이터의 MIME-type을 지정하는 예이다.

```
// 서버가 센드백할 데이터의 MIME-type 지정: json
xhr.setRequestHeader('Accept', 'application/json');
```

만약 Accept 헤더를 설정하지 않으면, send 메소드가 호출될 때 Accept 헤더가 `*/*`으로 전송된다.

Ajax response

```
// XMLHttpRequest 객체의 생성
const xhr = new XMLHttpRequest();

// XMLHttpRequest.readyState 프로퍼티가 변경(이벤트 발생)될 때마다 onreadystatechange 이벤트 핸들러가 호출된다.
xhr.onreadystatechange = function (e) {
  // readyStates는 XMLHttpRequest의 상태(state)를 반환
  // readyState: 4 => DONE(서버 응답 완료)
  if (xhr.readyState !== XMLHttpRequest.DONE) return;

  // status는 response 상태 코드를 반환 : 200 => 정상 응답
  if(xhr.status === 200) {
    console.log(xhr.responseText);
  } else {
    console.log('Error!');
  }
};
```

XMLHttpRequest.send 메소드를 통해 서버에 Request를 전송하면 서버는 Response를 반환한다. 하지만 언제 Response가 클라이언트에 도달할 지는 알 수 없다. XMLHttpRequest.onreadystatechange는 Response가 클라이언트에 도달하여 발생된 이벤트를 감지하고 콜백 함수를 실행하여 준다. 이때 이벤트는 Request에 어떠한 변화가 발생한 경우
즉 XMLHttpRequest.readyState 프로퍼티가 변경된 경우 발생한다.

```
// XMLHttpRequest 객체의 생성
var xhr = new XMLHttpRequest();
// 비동기 방식으로 Request를 오픈한다
xhr.open('GET', 'data/test.json');
// Request를 전송한다
xhr.send();

// XMLHttpRequest.readyState 프로퍼티가 변경(이벤트 발생)될 때마다 콜백함수(이벤트 핸들러)를 호출한다.
xhr.onreadystatechange = function (e) {
  // 이 함수는 Response가 클라이언트에 도달하면 호출된다.
};
```

XMLHttpRequest 객체는 response가 클라이언트에 도달했는지를 추적할 수 있는 프로퍼티를 제공한다. 이 프로퍼티가 바로 XMLHttpRequest.readyState이다. 만일 XMLHttpRequest.readyState의 값이 4인 경우, 정상적으로 Response가 돌아온 경우이다.

readXMLHttpRequest.readyState의 값

Value	State	Description
0	UNSENT	XMLHttpRequest.open()메소드 호출 이전
1	OPENED	XMLHttpRequest.open()메소드 호출 완료
2	HEADERS_RECEIVED	XMLHttpRequest.send()메소드 호출 완료
3	LOADING	서버 응답 중(XMLHttpRequest.responseText 미완성 상태)
4	DONE	서버 응답 완료

```
// XMLHttpRequest 객체의 생성
var xhr = new XMLHttpRequest();
// 비동기 방식으로 Request를 오픈한다
xhr.open('GET', 'data/test.json');
// Request를 전송한다
xhr.send();

// XMLHttpRequest.readyState 프로퍼티가 변경(이벤트 발생)될 때마다 콜백함수(이벤트 핸들러)를 호출한다.
xhr.onreadystatechange = function (e) {
  // 이 함수는 Response가 클라이언트에 도달하면 호출된다.

  // readyStates는 XMLHttpRequest의 상태(state)를 반환
  // readyState: 4 => DONE(서버 응답 완료)
  if (xhr.readyState !== XMLHttpRequest.DONE) return;

  // status는 response 상태 코드를 반환 : 200 => 정상 응답
  if(xhr.status === 200) {
```

```
    console.log(xhr.responseText);
  } else {
    console.log('Error!');
  }
};
```

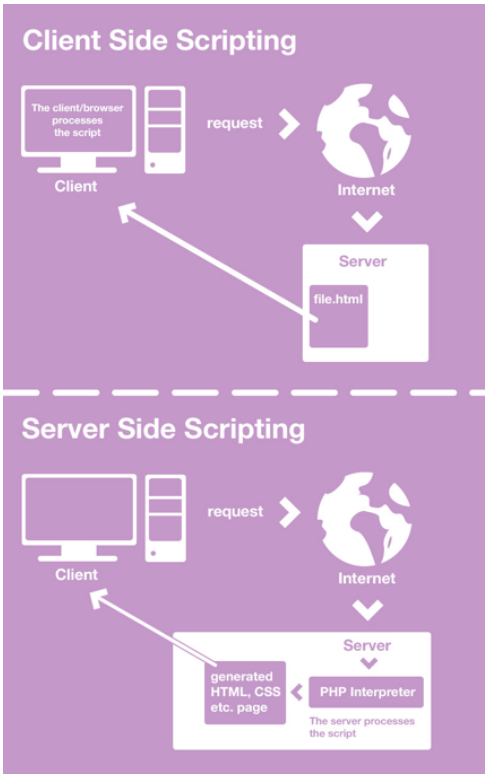
XMLHttpRequest의.readyState가 4인 경우, 서버 응답이 완료된 상태이므로 이후 XMLHttpRequest.status가 200(정상 응답)임을 확인하고 정상인 경우, XMLHttpRequest.responseText를 취득한다. XMLHttpRequest.responseText에는 서버가 전송한 데이터가 담겨 있다.

만약 서버 응답 완료 상태에만 반응하도록 하려면 readystatechange 이벤트 대신 load 이벤트를 사용해도 된다. load 이벤트는 서버 응답이 완료된 경우에 발생한다.

```
// XMLHttpRequest 객체의 생성
var xhr = new XMLHttpRequest();
// 비동기 방식으로 Request를 오픈한다
xhr.open('GET', 'data/test.json');
// Request를 전송한다
xhr.send();

// load 이벤트는 서버 응답이 완료된 경우에 발생한다.
xhr.onload = function (e) {
  // status는 response 상태 코드를 반환 : 200 => 정상 응답
  if(xhr.status === 200) {
    console.log(xhr.responseText);
  } else {
    console.log('Error!');
  }
};
```

Web Server

브라우저와 같은 클라이언트로부터 HTTP 요청을 받아들이고 HTML 문서와 같은 웹 페이지를 반환하는 컴퓨터 프로그램이다.



Ajax는 웹서버와의 통신이 필요하므로 예제를 실행하기 위해서는 웹서버가 필요하다.

Node.js가 설치되어 있다면 Express로 간단한 웹서버를 생성한다.

```
## 데스크탑에 webserver-express 폴더가 생성된다.
$ cd ~/Desktop
$ git clone https://github.com/ungmo2/webserver-express.git
$ cd webserver-express
## install express
$ npm install
## create public folder
$ mkdir public
```

webserver-express 디렉터리 내에 있는 **public** 디렉터리가 루트 디렉터리이다.

웹서버를 실행한다.

```
## start webserver
$ npm start
```

http://localhost:3000에 접속하여 Hello World!가 표시되면 웹서버가 정상 동작하는 것이다.

Ajax 예제

Load HTML

Ajax를 이용하여 웹페이지에 추가하기 가장 손쉬운 데이터 형식은 HTML이다. 별도의 작업없이 전송받은 데이터를 DOM에 추가하면 된다.

```
<!-- 루트 폴더(webserver-express/public)/loadhtml.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="https://poiemaweb.com/assets/css/ajax.css">
  </head>
  <body>
    <div id="content"></div>

    <script>
      // XMLHttpRequest 객체의 생성
      const xhr = new XMLHttpRequest();
      // 비동기 방식으로 Request를 오픈한다
      xhr.open('GET', 'data/data.html');
      // Request를 전송한다
      xhr.send();

      // Event Handler
      xhr.onreadystatechange = function () {
        // 서버 응답 완료 && 정상 응답
        if (xhr.readyState !== XMLHttpRequest.DONE) return;

        if (xhr.status === 200) {
          console.log(xhr.responseText);

          document.getElementById('content').innerHTML = xhr.responseText;
        } else {
          console.log(`[${xhr.status}] : ${xhr.statusText}`);
        }
      };
    </script>
  </body>
</html>
```

```
<!-- 루트 폴더(webserver-express/public)/data/data.html -->
<div id="tours">
  <h1>Guided Tours</h1>
  <ul>
    <li class="usa tour">
      <h2>New York, USA</h2>
      <span class="details">$1,899 for 7 nights</span>
      <button class="book">Book Now</button>
```

```
    </li>
    <li class="europe tour">
      <h2>Paris, France</h2>
      <span class="details">$2,299 for 7 nights</span>
      <button class="book">Book Now</button>
    </li>
    <li class="asia tour">
      <h2>Tokyo, Japan</h2>
      <span class="details">$3,799 for 7 nights</span>
      <button class="book">Book Now</button>
    </li>
  </ul>
</div>
```

http://localhost:3000/loadhtml.html

Load JSON

서버로부터 브라우저로 전송된 JSON 데이터는 문자열이다. 이 문자열을 객체화하여야 하는데 이를 역직렬화(Deserializing)이라 한다. 역직렬화를 위해서 내장 객체 JSON의 static 메소드인 JSON.parse()를 사용한다.

```
<!-- 루트 폴더(webserver-express/public)/loadjson.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="https://poiemaweb.com/assets/css/ajax.css">
  </head>
  <body>
    <div id="content"></div>

    <script>
      // XMLHttpRequest 객체의 생성
      var xhr = new XMLHttpRequest();

      // 비동기 방식으로 Request를 오픈한다
      xhr.open('GET', 'data/data.json');
      // Request를 전송한다
      xhr.send();

      xhr.onreadystatechange = function () {
        // 서버 응답 완료 && 정상 응답
        if (xhr.readyState !== XMLHttpRequest.DONE) return;

        if (xhr.status === 200) {
          console.log(xhr.responseText);

          // Deserializing (String → Object)
          responseObject = JSON.parse(xhr.responseText);

          // JSON → HTML String
          let newContent = '<div id="tours"><h1>Guided Tours</h1><ul>';

          responseObject.tours.forEach(tour => {
            newContent += `<li class="${tour.region} tour">
                <h2>${tour.location}</h2>
                <span class="details">${tour.details}</span>
                <button class="book">Book Now</button>
              </li>`;
          });

          newContent += '</ul></div>';

          document.getElementById('content').innerHTML = newContent;
        } else {
          console.log(`[${xhr.status}] : ${xhr.statusText}`);
        }
      };
    </script>
  </body>
</html>
```

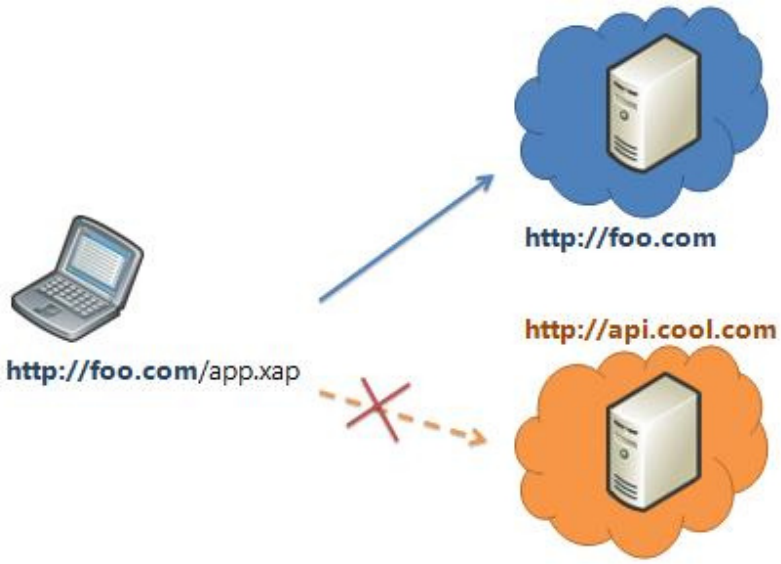
경로: 루트 폴더(webserver-express/public)/data/data.json

```
{
  "tours": [
    {
      "region": "usa",
      "location": "New York, USA",
      "details": "$1,899 for 7 nights"
    },
    {
      "region": "europe",
      "location": "Paris, France",
      "details": "$2,299 for 7 nights"
    },
    {
      "region": "asia",
      "location": "Tokyo, Japan",
      "details": "$3,799 for 7 nights"
    }
  ]
}
```

http://localhost:3000/loadjson.html

Load JSONP

요청에 의해 웹페이지가 전달된 서버와 동일한 도메인의 서버로 부터 전달된 데이터는 문제없이 처리할 수 있다. 하지만 보안상의 이유로 다른 도메인(http와 https, 포트가 다르면 다른 도메인으로 간주한다)으로의 요청(크로스 도메인 요청)은 제한된다. 이것을 동일출처원칙(Same-origin policy)이라고 한다.



동일출처원칙(Same-origin policy)

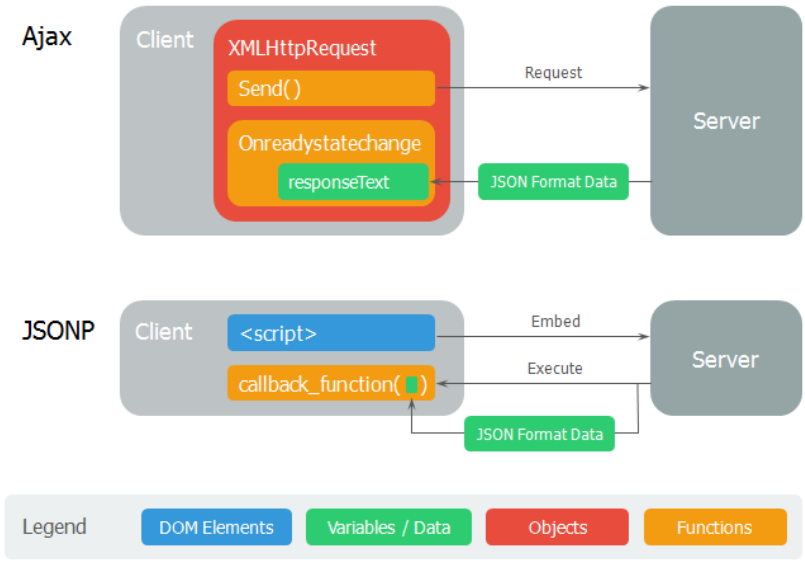
동일출처원칙을 우회하는 방법은 세가지가 있다.

1. 웹서버의 프록시 파일

서버에 원격 서버로부터 데이터를 수집하는 별도의 기능을 추가하는 것이다. 이를 프록시(Proxy)라 한다.

2. JSONP

script 태그의 원본 주소에 대한 제약은 존재하지 않는데 이것을 이용하여 다른 도메인의 서버에서 데이터를 수집하는 방법이다. 자신의 서버에 함수를 정의하고 다른 도메인의 서버에 얻고자 하는 데이터를 인수로 하는 함수 호출문을 로드하는 것이다.




```
<!-- 루트 폴더(webserver-express/public)/loadjsonp.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="https://poiemaweb.com/assets/css/ajax.css">
  </head>
  <body>
    <div id="content"></div>
  <script>
    function showTours(data) {
      console.log(data); // data: object

      // JSON → HTML String
      let newContent = `<div id="tours">
          <h1>Guided Tours</h1>
        <ul>`;

      data.tours.forEach(tour => {
        newContent += `<li class="${tour.region} tour">
            <h2>${tour.location}</h2>
            <span class="details">${tour.details}</span>
            <button class="book">Book Now</button>
          </li>`;
      });

      newContent += '</ul></div>';

      document.getElementById('content').innerHTML = newContent;
    }
    </script>
    <script src='https://poiemaweb.com/assets/data/data-jsonp.js'></script>
    <!-- <script>
      showTours({
        "tours": [
          {
            "region": "usa",
            "location": "New York, USA",
            "details": "$1,899 for 7 nights"
          },
          {
            "region": "europe",
            "location": "Paris, France",
            "details": "$2,299 for 7 nights"
          },
          {
            "region": "asia",
            "location": "Tokyo, Japan",
            "details": "$3,799 for 7 nights"
          }
        ]
      });
    </script> -->
```

```
  </body>
</html>
```

```
// https://poiemaweb.com/assets/data/data-jsonp.js
showTours({
  "tours": [
    {
      "region": "usa",
      "location": "New York, USA",
      "details": "$1,899 for 7 nights"
    },
    {
      "region": "europe",
      "location": "Paris, France",
      "details": "$2,299 for 7 nights"
    },
    {
      "region": "asia",
      "location": "Tokyo, Japan",
      "details": "$3,799 for 7 nights"
    }
  ]
});
```

http://localhost:3000/loadjsonp.html

3. Cross-Origin Resource Sharing

HTTP 헤더에 추가적으로 정보를 추가하여 브라우저와 서버가 서로 통신해야 한다는 사실을 알게하는 방법이다. W3C 명세에 포함되어 있지만 최신 브라우저에서만 동작하며 서버에 HTTP 헤더를 설정해 주어야 한다.

Node.js로 구현한 서버의 경우, CORS package를 사용하면 간단하게 Cross-Origin Resource Sharing을 구현할 수 있다.

```
const express = require('express')
const cors = require('cors')
const app = express()

app.use(cors())

app.get('/products/:id', function (req, res, next) {
  res.json({msg: 'This is CORS-enabled for all origins!'})
})

app.listen(80, function () {
  console.log('CORS-enabled web server listening on port 80')
})
```

REST(Representational State Transfer) API

웹이 HTTP의 설계 상 우수성을 제대로 사용하지 못하고 있는 상황을 보고 웹의 장점을 최대한 활용할 수 있는 아키텍쳐로서 REST를 소개하였고 이는 HTTP 프로토콜을 의도에 맞게 디자인하도록 유도하고 있다. REST의 기본 원칙을 성실히 지킨 서비스 디자인을 “RESTful”이라고 표현한다.

REST API 중심 규칙

REST에서 가장 중요한 기본적인 규칙은 두 가지이다. URI는 자원을 표현하는 데에 집중하고 행위에 대한 정의는 HTTP Method를 통해 하는 것이 REST한 API를 설계하는 중심 규칙이다.

1. URI는 정보의 자원을 표현해야 한다.

리소스명은 동사보다는 명사를 사용한다. URI는 자원을 표현하는데 중점을 두어야 한다. get 같은 행위에 대한 표현이 들어가서는 안된다.

```
# bad
GET /getTodos/1
GET /todos/show/1
```

```
# good
GET /todos/1
```

2. 자원에 대한 행위는 HTTP Method(GET, POST, PUT, DELETE 등)으로 표현한다.

```
# bad
GET /todos/delete/1

# good
DELETE /todos/1
```

HTTP Method

주로 5가지의 Method(GET,POST,PUT,PATCH,DELETE)를 사용하여 CRUD를 구현한다.

Method	Action	역할	페이로드
GET	index/retrieve	모든/특정 리소스를 조회	X
POST	create	리소스를 생성	O
PUT	replace	리소스의 전체를 교체	O
PATCH	modify	리소스의 일부를 수정	O
DELETE	delete	모든/ 특정 리소스를 삭제	X

REST API의 구성

자원(Resource), 행위(Verb), 표현(Representations)의 3가지 요소로 구성되다. REST는 자체 표현구조로 구성되어 REST API만으로 요청을 이해할 수 있다.

구성요소	내용	표현방법
Resource	자원	HTTP URI
Verb	자원에 대한 행위	HTTP Method
Representations	자원에 대한 행위의 내용	HTTP Message Pay Load

REST API의 Example

json-server

json-server를 사용하여 REST API를 사용하여 보자.

```
$ mkdir rest-api-exam && cd rest-api-exam
$ npm init -y
$ npm install json-server
```

db.json 파일을 아래와 같이 생성한다.

```
{
  "todos": [
    { "id": 1, "content": "HTML", "completed": false },
    { "id": 2, "content": "CSS", "completed": true },
    { "id": 3, "content": "Javascript", "completed": false }
  ]
}
```

npm script를 사용하여 json-server를 실행한다. 아래와 같이 package.json을 수정한다.

```
{
  "name": "rest-api-exam",
  "version": "1.0.0",
```

```
  "description": "",
  "scripts": {
    "start": "json-server --watch db.json --port 5000"
  },
  "dependencies": {
    "json-server": "^0.15.0"
  }
}
```

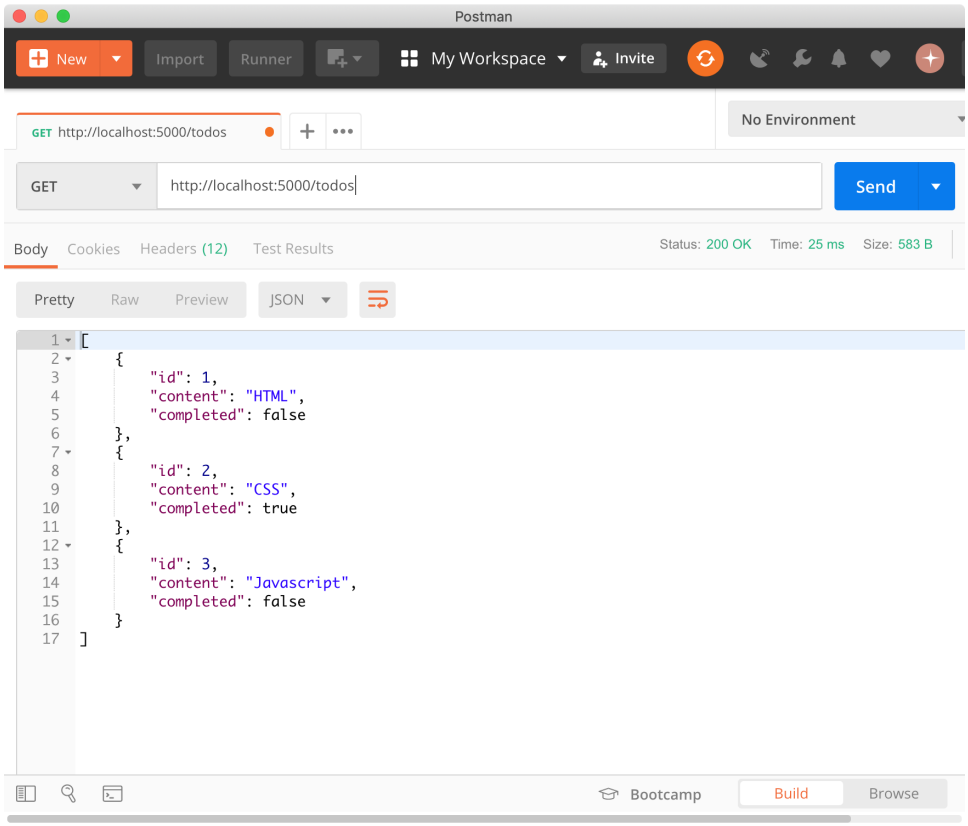
json-server를 실행한다. 포트는 5000을 사용한다.

```
$ npm start
```

GET

todos 리소스에서 모든 todo를 조회(index)한다.

```
$ curl -X GET http://localhost:5000/todos
[
  {
    "id": 1,
    "content": "HTML",
    "completed": false
  },
  {
    "id": 2,
    "content": "CSS",
    "completed": true
  },
  {
    "id": 3,
    "content": "Javascript",
    "completed": false
  }
]
```

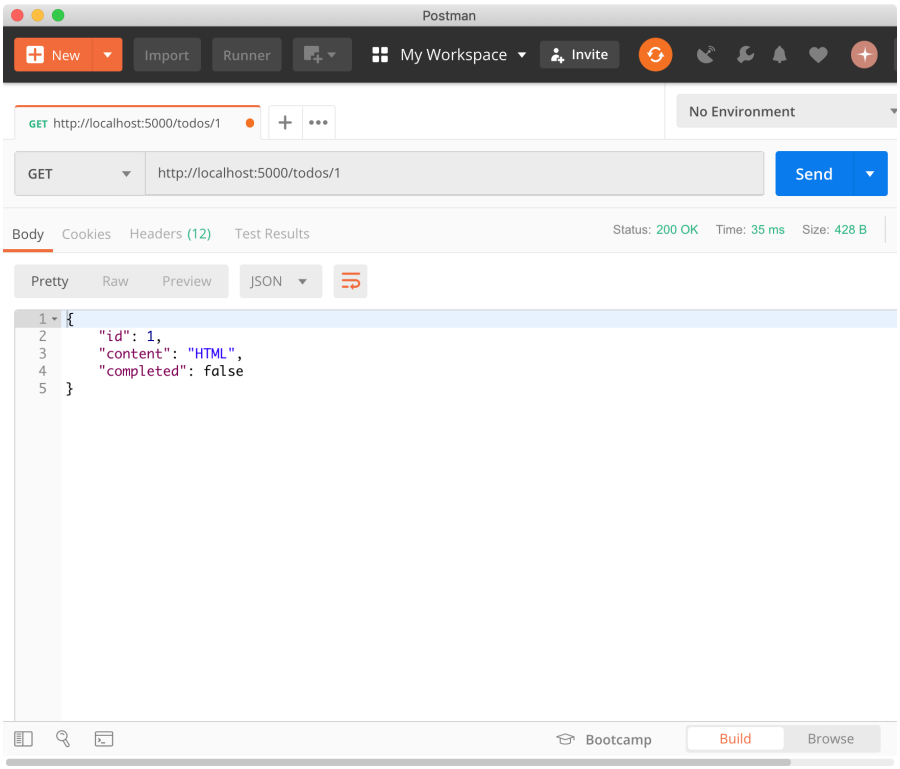


Postman

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'http://localhost:5000/todos');
xhr.send();

xhr.onreadystatechange = function (e) {
  if (xhr.readyState !== XMLHttpRequest.DONE) return;

  if(xhr.status === 200) { // 200: OK => https://httpstatuses.com
    console.log(xhr.responseText);
  } else {
    console.log("Error!");
  }

};
```

todos 리소스에서 id를 사용하여 특정 todo를 조회(retrieve)한다.

```
$ curl -X GET http://localhost:5000/todos/1
{
  "id": 1,
  "content": "HTML",
  "completed": false
}
```

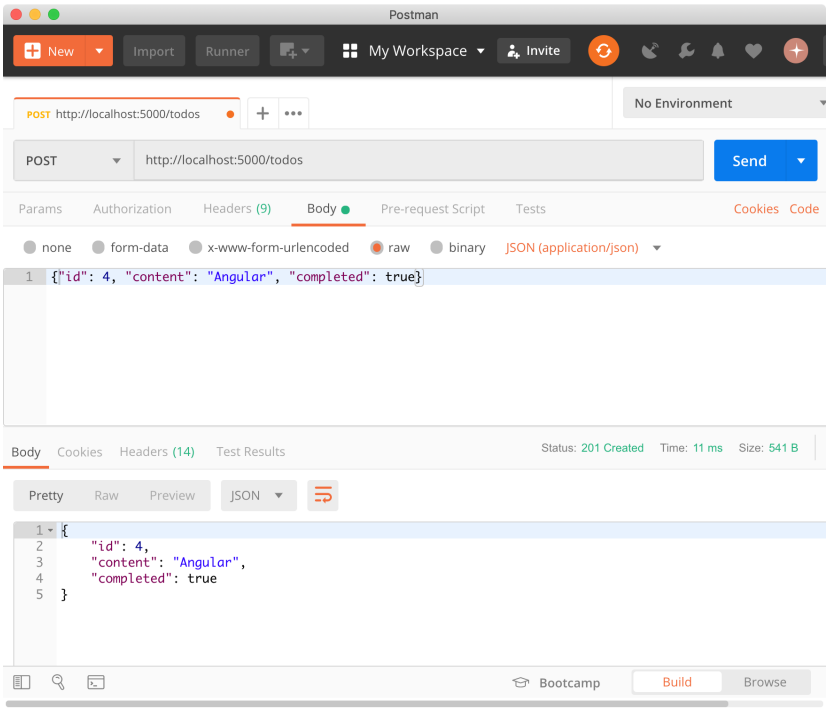



```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'http://localhost:5000/todos/1');
xhr.send();

xhr.onreadystatechange = function (e) {
  if (xhr.readyState !== XMLHttpRequest.DONE) return;

  if(xhr.status === 200) {
    console.log(xhr.responseText);
  } else {
    console.log("Error!");
  }
};
```

POST

todos 리소스에 새로운 todo를 생성한다.

```
$ curl -X POST http://localhost:5000/todos -H "Content-Type: application/json" -d '{"id": 4, "content": "Angular", "comple
ted": true}'
{
  "id": 4,
  "content": "Angular",
  "completed": true
}
```



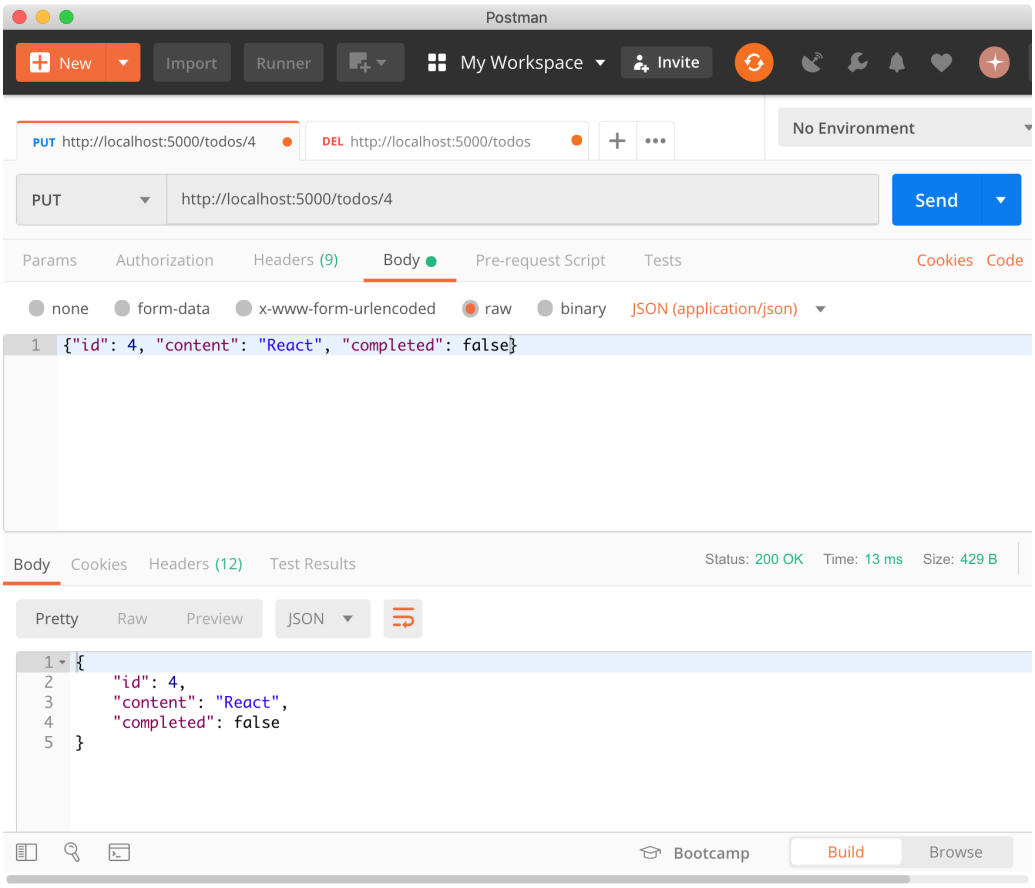

```
const xhr = new XMLHttpRequest();
xhr.open('POST', 'http://localhost:5000/todos');
xhr.setRequestHeader('Content-type', 'application/json');
xhr.send(JSON.stringify({ id: 4, content: 'Angular', completed: true }));

xhr.onreadystatechange = function (e) {
  if (xhr.readyState !== XMLHttpRequest.DONE) return;

  if(xhr.status === 201) { // 201: Created
    console.log(xhr.responseText);
  } else {
    console.log("Error!");
  }
};
```

PUT

PUT은 특정 리소스의 전체를 갱신할 때 사용한다. todos 리소스에서 id를 사용하여 todo를 특정하여 id를 제외한 리소스 전체를 갱신한다.

```
$ curl -X PUT http://localhost:5000/todos/4 -H "Content-Type: application/json" -d '{"id": 4, "content": "React", "complet
ed": false}'
{
  "content": "React",
  "completed": false,
```

```
  "id": 4
}
```

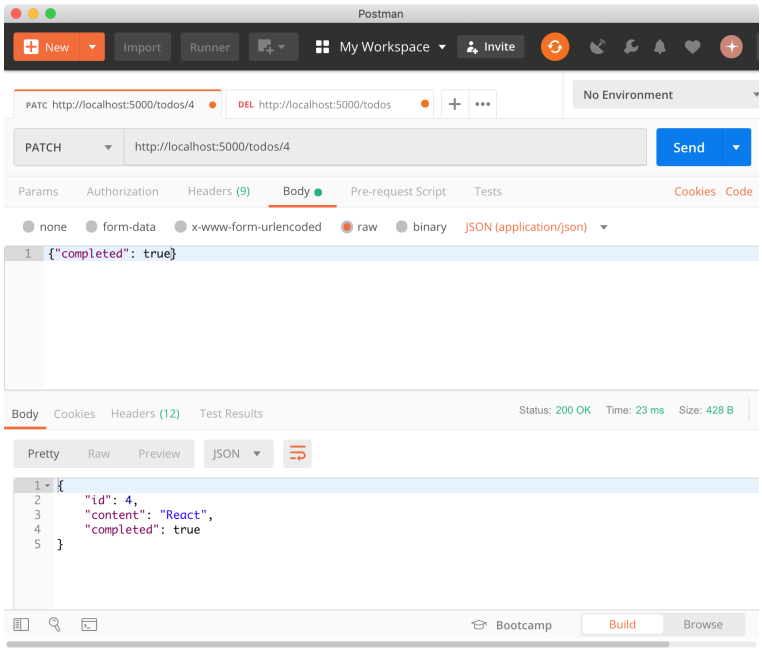



```
const xhr = new XMLHttpRequest();
xhr.open('PUT', 'http://localhost:5000/todos/4');
xhr.setRequestHeader('Content-type', 'application/json');
xhr.send(JSON.stringify({ id: 4, content: 'React', completed: false }));

xhr.onreadystatechange = function (e) {
  if (xhr.readyState !== XMLHttpRequest.DONE) return;

  if(xhr.status === 200) {
    console.log(xhr.responseText);
  } else {
    console.log("Error!");
  }
};
```

PATCH

PATCH는 특정 리소스의 일부를 갱신할 때 사용한다. todos 리소스의 id를 사용하여 todo를 특정하여 completed만을 true로 갱신한다.

```
$ curl -X PATCH http://localhost:5000/todos/4 -H "Content-Type: application/json" -d '{"completed": true}'
{
  "id": 4,
  "content": "React",
  "completed": true
}
```

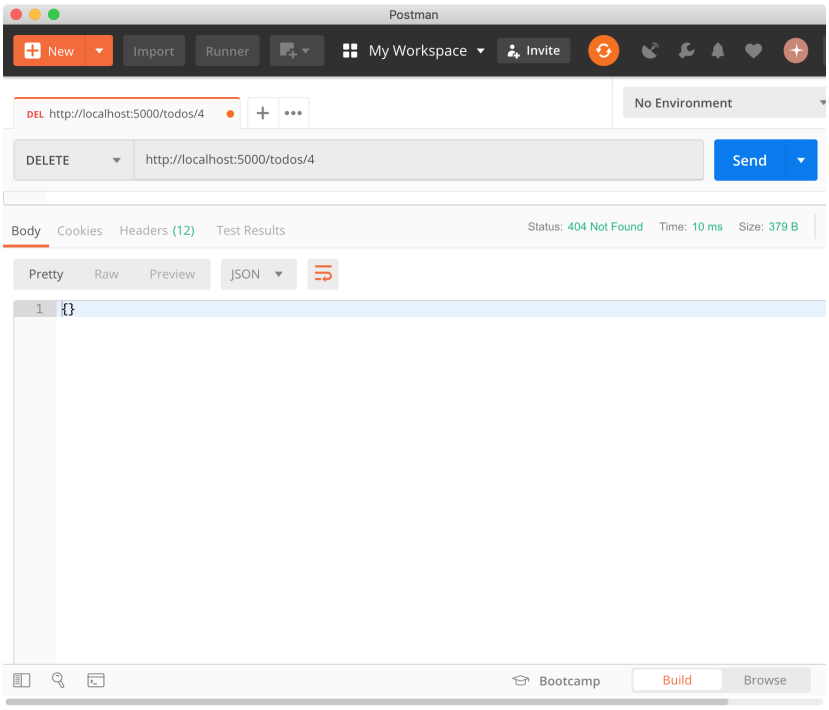



```
const xhr = new XMLHttpRequest();
xhr.open('PATCH', 'http://localhost:5000/todos/4');
xhr.setRequestHeader('Content-type', 'application/json');
xhr.send(JSON.stringify({ completed: true }));

xhr.onreadystatechange = function (e) {
  if (xhr.readyState !== XMLHttpRequest.DONE) return;

  if(xhr.status === 200) {
    console.log(xhr.responseText);
  } else {
    console.log("Error!");
  }
};
```

DELETE

todos 리소스에서 id를 사용하여 todo를 특정하고 삭제한다.

```
$ curl -X DELETE http://localhost:5000/todos/4
{}
```




```
const xhr = new XMLHttpRequest();
xhr.open('DELETE', 'http://localhost:5000/todos/4');
xhr.send();

xhr.onreadystatechange = function (e) {
  if (xhr.readyState !== XMLHttpRequest.DONE) return;

  if(xhr.status === 200) {
    console.log(xhr.responseText);
  } else {
    console.log("Error!");
  }
};
```

Single Page Application & Routing

SPA(Single Page Application)

단일 페이지 애플리케이션(Single Page Application, SPA)는 모던 웹의 패러다임이다. SPA는 기본적으로 단일 페이지로 구성되며 기존의 서버 사이드 렌더링과 비교할 때, 배포가 간단하며 네이티브 앱과 유사한 사용자 경험을 제공할 수 있다는 장점이 있다. SPA는 기본적으로 웹 애플리케이션에 필요한 모든 정적 리소스를 최초 접근시 단 한번만 다운로드한다. 이후 새로운 페이지 요청 시, 페이지 갱신에 필요한 데이터만을 JSON으로 전달받아 페이지를 갱신하므로 전체적인 트래픽을 감소시킬 수 있고, 전체 페이지를 다시 렌더링하지 않고 변경되는 부분만을 갱신하므로 새로고침이 발생하지 않아 네이티브 앱과 유사한 사용자 경험을 제공할 수 있다.

모든 소프트웨어 아키텍처에는 트레이드오프(trade-off)가 존재하며 모든 애플리케이션에 적합한 은탄환(Silver bullet)은 없듯이 SPA 또한 구조적인 단점을 가지고 있다. 대표적인 단점은 다음과 같다.

초기 구동 속도

SPA는 웹 애플리케이션에 필요한 모든 정적 리소스를 최초 접근시 단 한번 다운로드하기 때문에 초기 구동 속도가 상대적으로 느리다. 하지만 SPA는 웹페이지보다는 애플리케이션에 적합한 기술이므로 트래픽 감소와 속도, 사용성, 반응성의 향상 등의 장점을 생각한다면 결정적인 단점이라고 할 수는 없다.

SEO(검색엔진 최적화) 이슈

SPA는 일반적으로 서버 사이드 렌더링(SSR) 방식이 아닌 자바스크립트 기반 비동기 모델의 클라이언트 사이드 렌더링(CSR) 방식으로 동작한다. 클라이언트 사이드 렌더링(CSR)은 일반적으로 데이터 패치 요청을 통해 서버로부터 데이터를 응답받아 뷰

를 동적으로 생성하는데 이때 브라우저 주소창의 URL이 변경되지 않는다. 이는 사용자 방문 history를 관리할 수 없음을 의미하며 SEO 이슈의 발생 원인이기도 하다. SPA의 SEO 이슈는 언제나 단점으로 부각되어 왔다. SPA는 정보 제공을 위한 웹페이지보다는 애플리케이션에 적합한 기술이므로 SEO 이슈는 심각한 문제로 취급할 수 없다고 생각할 수도 있지만 블로그와 같이 애플리케이션의 경우 SEO는 무시할 수 없는 중요한 의미를 갖는다. Angular나 React 등의 SPA 프레임워크는 서버 사이드 렌더링(SSR)을 지원하는 기능이 이미 존재하고 있고 크롬 등의 모던 브라우저는 SPA의 SEO 문제를 해결하고 있는 것으로 알려져 있다.

Routing

라우팅이란 출발지에서 목적지까지의 경로를 결정하는 기능이다. 애플리케이션의 라우팅은 사용자가 태스크를 수행하기 위해 어떤 화면(view)에서 다른 화면으로 화면을 전환하는 내비게이션을 관리하기 위한 기능을 의미한다. 일반적으로 라우팅은 사용자가 요청한 URL 또는 이벤트를 해석하고 새로운 페이지로 전환하기 위해 필요한 데이터를 서버에 요청하고 페이지를 전환하는 위한 일련의 행위를 말한다.

브라우저가 화면을 전환하는 경우는 다음과 같다.

1. 브라우저의 주소창에 URL을 입력하면 해당 페이지로 이동한다.
2. 웹페이지의 링크(a 태그)를 클릭하면 해당 페이지로 이동한다.
3. 브라우저의 뒤로가기 또는 앞으로가기 버튼을 클릭하면 사용자 방문 기록(history)의 뒤 또는 앞으로 이동한다. **history 관리를 위해서는 각 페이지는 브라우저의 주소창에서 구별할 수 있는 유일한 URL을 소유해야 한다.**

SPA와 Routing

전통적인 링크 방식에서 SPA까지 발전하게 된 과정과 SPA의 라우팅(Routing)에 대해 살펴보도록 하자. 예제를 실행하기 위해 과정은 다음과 같다.

github에서 소스코드를 clone한다.

```
$ git clone https://github.com/ungmo2/spa-router.git
$ cd spa-router
$ npm install
```

예제의 실행 방법은 다음과 같다.

```
# 전통적 링크 방식
$ npm run link
# ajax 방식
$ npm run ajax
# hash 방식
$ npm run hash
# pjax(pushState + ajax) 방식
$ npm run pjax
```

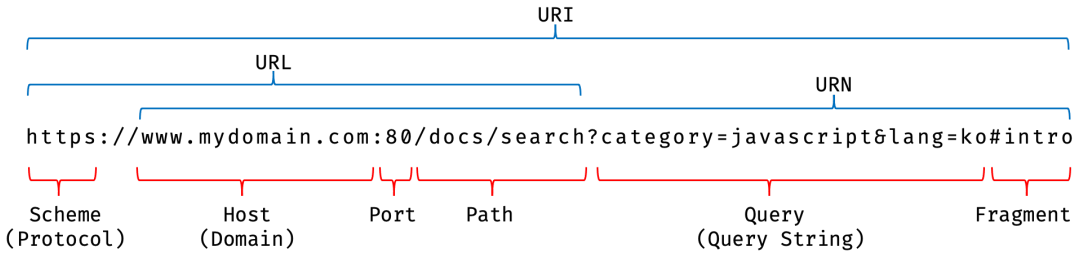
전통적 링크 방식

전통적 링크 방식은 link tag로 동작하는 기본적인 웹페이지의 동작 방식이다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>SPA-Router - Link</title>
    <link rel="stylesheet" href="css/style.css" />
  </head>
  <body>
    <nav>
      <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/service.html">Service</a></li>
        <li><a href="/about.html">About</a></li>
      </ul>
```

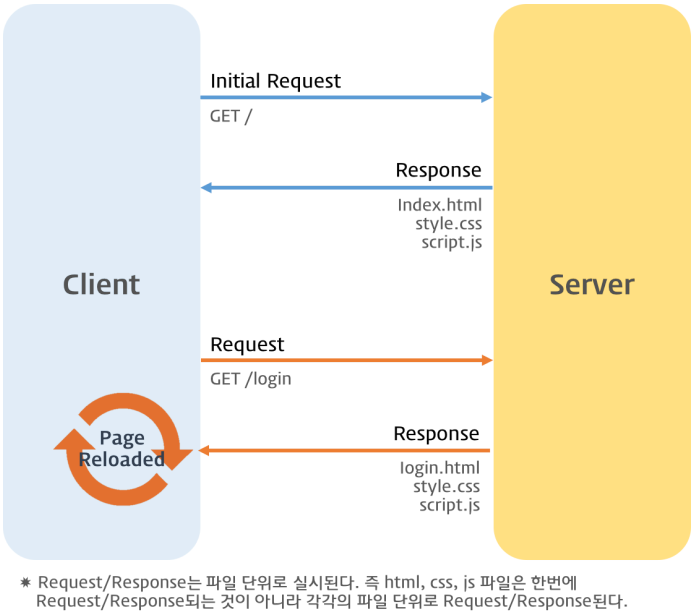
```
    </nav>
    <section>
      <h1>Home</h1>
      <p>This is main page</p>
    </section>
  </body>
</html>
```

link tag(`<a href="/service.html">Service</a>` 등)을 클릭하면 href 어트리뷰트 값인 리소스 경로가 URL의 path에 추가되어 주소창에 나타나고 해당 리소스를 서버에 요청한다.



URI의 구조

이때 서버는 html로 화면을 표시하는데 부족함이 없는 완전한 리소스를 클라이언트에 응답한다. 이를 **서버 사이드 렌더링(SSR)**이라 한다. 브라우저는 서버가 응답한 html을 응답받아 렌더링한다. 이때 응답받은 html로 전체 페이지를 다시 렌더링하게 되므로 새로고침이 발생한다.



traditional webpage lifecycle

이 방식은 자바스크립트의 도움 없이 응답받은 html만으로 렌더링이 가능하며 각 페이지마다 고유의 URL이 존재하므로 history 관리 및 SEO 대응에 아무런 문제가 없다. 하지만 요청마다 중복된 리소스를 응답받아야 하며 전체 페이지를 다시 렌더링하는 과정에서 새로고침이 발생하여 사용성이 좋지 않은 단점이 있다.

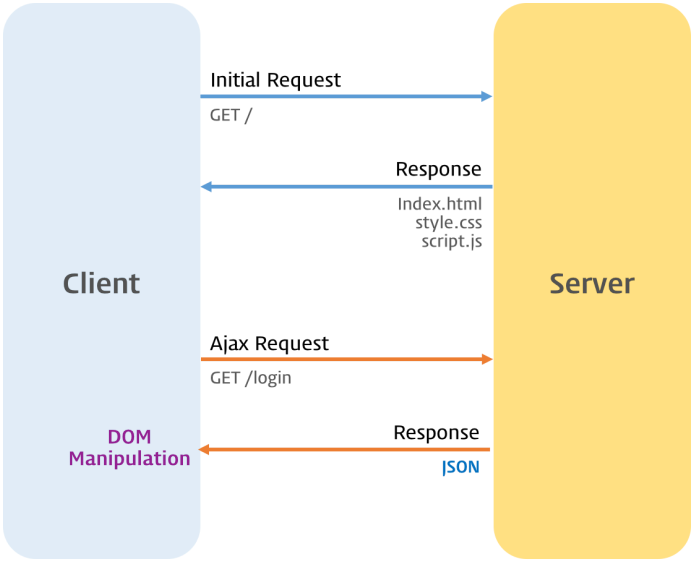
위 예제를 실행하려면 다음 명령을 실행한다.

```
$ npm run link
```

ajax 방식

전통적 링크 방식은 현재 페이지에서 수신된 html로 화면을 전환하는 과정에서 전체 페이지를 새롭게 렌더링하게 되므로 새로고침이 발생한다. 간단한 웹페이지라면 문제될 것이 없겠지만 복잡한 웹페이지의 경우, 요청마다 중복된 HTML과 CSS, JavaScript를 매번 다운로드해야하므로 속도 저하의 요인이 된다.

전통적 링크 방식의 단점을 보완하기 위해 등장한 것이 ajax(Asynchronous JavaScript and XML)이다. ajax는 자바스크립트를 이용해서 비동기적(asynchronous)으로 서버와 브라우저가 데이터를 교환할 수 있는 통신 방식을 의미한다.



✷ Request/Response는 파일 단위로 실시된다. 즉 html, css, js 파일은 한번에 Request/Response되는 것이 아니라 각각의 파일 단위로 Request/Response된다.

ajax lifecycle

서버로부터 웹페이지가 응답되면 화면 전체를 새롭게 렌더링해야 하는데 페이지 일부만 갱신하고도 동일한 효과를 볼 수 있도록 하는 것이 ajax이다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>SPA-Router - ajax</title>
    <link rel="stylesheet" href="css/style.css" />
    <script type="module" src="js/index.js"></script>
  </head>
  <body>
    <nav>
      <ul id="navigation">
        <li><a href="/">Home</a></li>
        <li><a href="/service">Service</a></li>
        <li><a href="/about">About</a></li>
      </ul>
    </nav>
    <div id="root">Loading...</div>
  </body>
</html>
```

위 예제를 살펴보면 link tag(`<a href="/service">Service</a>` 등)의 href 어트리뷰트에 path를 사용하고 있다. 따라서 내비게이션이 클릭되면 path가 추가된 URL이 서버로 요청된다. 하지만 ajax 방식은 내비게이션 클릭 이벤트를 캐치하고 preventDefault 메서드를 사용해 서버로의 요청을 방지한다. 이후, href 어트리뷰트에 path을 사용하여 ajax 요청을 하는 방식이다.

그리고 div.#root 요소에 웹페이지의 내용이 비어있는 것을 알 수 있다. 요청된 리소스(html, json 등)가 응답되면 클라이언트에서 div#root 요소에 응답받은 데이터를 기반으로 html을 생성해 추가한다.

이를 통해 불필요한 리소스의 중복 요청을 방지할 수 있다. 또한 페이지 전체를 리렌더링할 필요없이 갱신이 필요한 일부만 갱신하면 되므로 빠른 퍼포먼스와 부드러운 화면 표시 효과를 기대할 수 있어 새로고침이 없는 보다 향상된 사용자 경험을 구현할 수 있다는 장점이 있다.

JavaScript의 구현은 다음과 같다.

```
// index.js
import { Home, Service, About, NotFound } from './components.js';

const root = document.getElementById('root');
const navigation = document.getElementById('navigation');

const routes = [
  { path: '/', component: Home },
  { path: '/service', component: Service },
  { path: '/about', component: About },
];

const render = async path => {
  try {
    const component = routes.find(route => route.path === path)?.component || NotFound;
    root.replaceChildren(await component());
  } catch (err) {
    console.error(err);
  }
};

// TODO: ajax 요청은 주소창의 url을 변경시키지 않으므로 history 관리가 되지 않는다.
navigation.onclick = e => {
  if (!e.target.matches('#navigation > li > a')) return;
  e.preventDefault();
  const path = e.target.getAttribute('href');
  render(path);
};

// TODO: 주소창의 url이 변경되지 않기 때문에 새로고침 시 현재 렌더링된 페이지가 아닌 루트 페이지가 요청된다.
window.addEventListener('DOMContentLoaded', () => render('/'));
```

```
// components.js
const createElement = string => {
  const $temp = document.createElement('template');
  $temp.innerHTML = string;
  return $temp.content;
};

const fetchData = async url => {
  const res = await fetch(url);
  const json = await res.json();
  return json;
};

export const Home = async () => {
  const { title, content } = await fetchData('/data/home.json');
  return createElement(`<h1>${title}</h1><p>${content}</p>`);
};

export const Service = async () => {
  const { title, content } = await fetchData('/data/service.json');
  return createElement(`<h1>${title}</h1><p>${content}</p>`);
};

export const About = async () => {
  const { title, content } = await fetchData('/data/about.json');
  return createElement(`<h1>${title}</h1><p>${content}</p>`);
};

export const NotFound = () => createElement('<h1>404 NotFound</p>');
```

ajax 요청은 주소창의 URL을 변경시키지 않는다. 이는 브라우저의 뒤로가기, 앞으로가기 등의 **history 관리가 동작하지 않음을 의미한다.** 따라서 history.back(), history.go(n) 등도 동작하지 않는다. 주소창의 URL이 변경되지 않기 때문에 새로고침을 해도 언제나 첫페이지가 다시 로딩된다. 동일한 하나의 URL로 동작하는 ajax 방식은 **SEO 이슈**에서도 자유로울 수 없다.

위 예제를 실행하려면 다음 명령을 실행한다.

```
$ npm run ajax
```

Hash 방식

ajax 방식은 불필요한 리소스 중복 요청을 방지할 수 있고 새로고침이 없는 사용자 경험을 구현할 수 있다는 장점이 있지만 history 관리가 되지 않는 단점이 있다. 이를 보완한 방법이 Hash 방식이다.

Hash 방식은 URI의 fragment identifier(#service)의 고유 기능인 앵커(anchor)를 사용한다. fragment identifier는 hash mark 또는 hash라고 부르기도 한다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>SPA-Router - Hash</title>
    <link rel="stylesheet" href="css/style.css" />
    <script type="module" src="js/index.js"></script>
  </head>
  <body>
    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#service">Service</a></li>
        <li><a href="#about">About</a></li>
      </ul>
    </nav>
    <div id="root">Loading...</div>
  </body>
</html>
```

위 예제를 살펴보면 link tag(`<a href="#service">Service</a>` 등)의 href 어트리뷰트에 hash를 사용하고 있다. 즉, 내비게이션이 클릭되면 hash가 추가된 URI가 주소창에 표시된다. 단, **URL이 동일한 상태에서 hash만 변경되면 브라우저는 서버에 어떠한 요청도 하지 않는다. 즉, URL의 hash는 변경되어도 서버에 새로운 요청을 보내지 않으며 따라서 페이지가 갱신되지 않는다.** hash는 요청을 위한 것이 아니라 fragment identifier(#service)의 고유 기능인 앵커(anchor)로 웹페이지 내부에서 이동을 위한 것이기 때문이다.

또한 hash 방식은 서버에 새로운 요청을 보내지 않으며 따라서 페이지가 갱신되지 않지만 페이지마다 고유의 **논리적 URL이 존재하므로 history 관리에 아무런 문제가 없다.**

JavaScript의 구현은 다음과 같다.

```
// index.js
// components.js는 위와 동일하다.
import { Home, Service, About, NotFound } from './components.js';

const root = document.getElementById('root');

const routes = [
  { path: '', component: Home },
  { path: 'service', component: Service },
  { path: 'about', component: About },
];

const render = async () => {
  try {
    // url의 hash를 취득
    const hash = window.location.hash.replace('#', '');
    const component = routes.find(route => route.path === hash)?.component || NotFound;
    root.replaceChildren(await component());
  } catch (err) {
    console.error(err);
  }
};

// 네비게이션을 클릭하면 url의 hash가 변경되기 때문에 history 관리가 가능하다.
// 단, url의 hash만 변경되면 서버로 요청은 수행하지 않는다.
// url의 hash가 변경하면 발생하는 이벤트인 hashchange 이벤트를 사용하여 hash의 변경을 감지하여 필요한 ajax 요청을 수행한다.
// hash 방식의 단점은 url에 /#foo와 같은 해시뱅(HashBang)이 들어간다는 것이다.
window.addEventListener('hashchange', render);

// 새로고침을 하면 DOMContentLoaded 이벤트가 발생하고
// render 함수는 url의 hash를 취득해 새로고침 직전에 렌더링되었던 페이지를 다시 렌더링한다.
window.addEventListener('DOMContentLoaded', render);
```

hash 방식은 url의 hash가 변경하면 발생하는 이벤트인 hashchange 이벤트를 사용해 hash의 변경을 감지하고 url의 hash를 취득해 필요한 ajax 요청을 수행한다.

hash 방식의 단점은 url에 불필요한 #이 들어간다는 것이다. 일반적으로 hash 방식을 사용할 때 #!을 사용하기도 하는데 이를 해시뱅(Hash-bang)이라고 부른다.

hash 방식은 과도기적 기술이다. HTML5의 History API인 pushState가 모든 브라우저에서 지원이 된다면 해시뱅은 사용하지 않아도 되지만 현재 pushState는 일부의 브라우저(IE 10 이상)에서만 지원이 가능하다.

또 다른 문제는 **SEO 이슈**이다. 웹 크롤러는 검색 엔진이 웹사이트의 콘텐츠를 수집하기 위해 HTTP와 URL 스펙(RFC-2396같은)을 따른다. 이러한 크롤러는 JavaScript를 실행시키지 않기 때문에 hash 방식으로 만들어진 사이트의 콘텐츠를 수집할 수 없다. 구글은 해시뱅을 일반적인 URL로 변경시켜 이 문제를 해결한 것으로 알려져 있지만 다른 검색 엔진은 hash 방식으로 만들어진 사이트의 콘텐츠를 수집할 수 없을 수도 있다.

위 예제를 실행하려면 다음 명령을 실행한다.

```
$ npm run hash
```

pjax 방식

hash 방식의 가장 큰 단점은 SEO 이슈이다. 이를 보완한 방법이 HTML5의 History API인 pushState와 popstate 이벤트를 사용한 pjax(pushState + ajax) 방식이다. pushState와 popstate은 IE 10 이상에서 동작한다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>SPA-Router - pjax</title>
    <link rel="stylesheet" href="css/style.css" />
    <script defer src="js/index.js"></script>
  </head>
  <body>
    <nav>
      <ul id="navigation">
        <li><a href="/">Home</a></li>
        <li><a href="/service">Service</a></li>
        <li><a href="/about">About</a></li>
      </ul>
    </nav>
    <div id="root">Loading...</div>
  </body>
</html>
```

위 예제를 살펴보면 link tag(`<a href="/service">Service</a>` 등)의 href 어트리뷰트에 path를 사용하고 있다. 따라서 내비게이션이 클릭되면 path가 추가된 URL이 서버로 요청된다. 하지만 pjax 방식은 내비게이션 클릭 이벤트를 캐치하고 preventDefault 메서드를 사용해 서버로의 요청을 방지한다. 이후, href 어트리뷰트에 path을 사용하여 ajax 요청을 하는 방식이다.

이때 ajax 요청은 브라우저 주소창의 URL을 변경시키지 않아 history 관리가 불가능하다. 이때 사용하는 것이 pushState 메서드이다. **pushState 메서드는 주소창의 URL을 변경하고 URL을 history entry로 추가하지만 서버로 HTTP 요청을 하지는 않는다.**

JavaScript의 구현은 다음과 같다.

```
// index.js
// components.js는 위와 동일하다.
import { Home, Service, About, NotFound } from './components.js';

const root = document.getElementById('root');
const navigation = document.getElementById('navigation');

const routes = [
  { path: '/', component: Home },
  { path: '/service', component: Service },
  { path: '/about', component: About },
```

```
];

const render = async path => {
  try {
    const component = routes.find(route => route.path === path)?.component || NotFound;
    root.replaceChildren(await component());
  } catch (err) {
    console.error(err);
  }
};

// 네비게이션을 클릭하면 주소창의 url이 변경되므로 HTTP 요청이 서버로 전송된다.
// preventDefault를 사용하여 이를 방지하고 history 관리를 위한 처리를 실행한다.
navigation.addEventListener('click', e => {
  if (!e.target.matches('#navigation > li > a')) return;

  e.preventDefault();

  // 이동할 페이지의 path
  const path = e.target.getAttribute('href');

  // pushState는 주소창의 url을 변경하지만 HTTP 요청을 서버로 전송하지는 않는다.
  window.history.pushState({}, null, path);

  render(path);
});

// pjax 방식은 hash를 사용하지 않으므로 hashchange 이벤트를 사용할 수 없다.
// popstate 이벤트는 pushState에 의해 발생하지 않고 앞으로/뒤로 가기 버튼을 클릭하거나
// history.forward/back/go(n)에 의해 history entry가 변경되면 발생한다.
window.addEventListener('popstate', () => {
  console.log('[popstate]', window.location.pathname);

  // 앞으로/뒤로 가기 버튼을 클릭하면 window.location.pathname를 참조해 뷰를 전환한다.
  render(window.location.pathname);
});

// 웹페이지가 처음 로딩되면 window.location.pathname를 확인해 페이지를 이동시킨다.
// 새로고침을 클릭하면 현 페이지(예를 들어 localhost:5004/service)가 서버에 요청된다.
// 이에 응답하는 처리는 서버에서 구현해야 한다.
render(window.location.pathname);
```

pjax 방식에서 사용하는 history.pushState 메서드는 주소창의 url을 변경하지만 HTTP 요청을 서버로 전송하지는 않는다. 따라서 페이지가 갱신되지 않는다. 하지만 페이지마다 고유의 URL이 존재하므로 history 관리에 아무런 문제가 없다. 또한 hash를 사용하지 않으므로 SEO에도 문제가 없다.

단, 브라우저의 새로고침 버튼을 클릭하면 브라우저 주소창의 url이 변경되지 않는 ajax 방식과 해시(fragment identifier)만 추가되는 hash 방식은 서버에 별도의 요청을 보내지 않지만 pjax 방식은 브라우저 주소창의 url이 변경되기 때문에 요청(예를 들어 localhost:5004/service)이 서버로 전달된다. 즉, **pjax 방식은 서버 렌더링 방식과 ajax 방식이 혼재되어 있는 방식으로 서버의 지원이 필요하다.** 이에 대한 서버 구현은 다음과 같다.

```
const express = require('express');
const path = require('path');

const app = express();
const port = 5004;

app.use(express.static(path.join(__dirname, 'public')));

// 브라우저 새로고침을 위한 처리 (다른 route가 존재하는 경우 맨 아래에 위치해야 한다)
// 브라우저 새로고침 시 서버는 index.html을 전달하고 클라이언트는 window.location.pathname를 참조해 다시 라우팅한다.
app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'public/index.html'));
});

app.listen(port, () => {
  console.log(`Server listening on http:/localhost:${port}`);
});
```

pjax 방식의 예제를 실행하려면 다음 명령을 실행한다.

```
$ npm run pjax
```

Conclusion

구분	History 관리	SEO 대응	사용자 경험	서버 렌더링	구현 난이도	IE대응
전통적 링크 방식	O	O	X	O	간단	
ajax 방식	X	X	O	X	보통	7이상
hash 방식	O	X	O	X	보통	8이상
pjax 방식	O	O	O	△	복잡	10이상