fs.readdir

- 디렉토리에 있는 아이템의 이름들을 리턴
- `fs.readdir(path[, optinos], callback)`
  - path <string> | <Buffer> | <URL>
  - options <string> | <Object>
    - encoding <string> Default : 'utf8'
    - withFileTypes <boolean> Default : false
  - callback <Function>
  - err <Error>
  - files <string[]> | <Buffer[]> | <fs.Dirent[]>

querystring

- 기존 노드 방식에서는 url querystring을 querystring 모듈로 처리
  - `querystring.parse(쿼리)` : url의 query 부분을 자바스크립트 객체로 분해
  - `querystring.stringify(객체)` : 분해된 query 객체를 문자열로 다시 조립

이벤트 이해하기

- Node에서 이벤트 처리 하는 EventEmitter 라는 것이 만들어져 있음
- 이벤트 보내고 받기
  - 노드의 객체는 EventEmitter를 상속 받을 수 있으며, 상속 후에는 EventEmitter 객체의 on()과 emit() 메소드 사용 가능
  - on() 메소드 = 이벤트가 전달될 객체에 이벤트 리스너를 설정하는 역할. 보통은 노드 내부에서 미리 만들어 제공하는 이벤트를 받아 처리하지만, 필요할 때는 직접 이벤트를 만들어 전달 가능
  - once() 메소드 = 이벤트 리스너 함수가 한번이라도 실행하고 나면 자동으로 제거되므로 이벤트를 딱 한번만 받아서 처리할 수 있음
  - emit() 메소드 = 이벤트를 다른쪽으로 전달하고 싶을 경우
- 이벤트를 처리하는 EventEmitter의 주요 메소드
  - `on(event, callback)`
    - 지정한 이벤트의 리스너를 추가
    - 지정한 Event 생성하고 on 메소드를 이용하여 이벤트를 등록하면 이 메소드를 호출하면서 파라미터로 전달한 이벤트가 발생했을 때 그 다음에 나오는 콜백함수 실행

```
process.on('exit', function() {
    console.log('exit 이벤트 발생');
});
// process 객체는 내부적으로 EventEmitter 상속

setTimeout(function() {
    console.log('2초 후에 시스템 종료 시도함');
    process.exit();
}, 2000);
```

버퍼와 스트림 이해하기

- 버퍼 : 일정한 크기로 모아두는 데이터
  - 일정한 크기가 되면 **한 번에 처리**
  - 버퍼링 : 버퍼에 데이터가 찰 때까지 모으는 작업
- 스트림 : 데이터의 흐름
  - 일정한 크기로 나눠서 **여러 번에 걸쳐서 처리**
  - 버퍼(또는 청크)의 크기를 작게 만들어서 주기적으로 데이터를 전달
  - 스트리밍 : 일정한 크기의 데이터를 지속적으로 전달하는 작업

Buffer의 메서드

- 노드에서는 Buffer 객체 사용
  - `from(문자열)`
    - 문자열을 버퍼로 바꿈
    - length 속성은 버퍼의 크기를 알려줌(바이트 단위)
  - `toString(버퍼)`
    - 버퍼를 다시 문자열로 바꿀 수 있음
    - 이 때, base64나 hex를 인자로 넣으면 해당 인코딩등으로 변환 가능
  - `concat(배열)`
    - 배열 안에 든 버퍼들을 하나로 합침
  - `alloc(바이트)`
    - 빈 버퍼를 생성
    - 바이트를 인자로 지정해주면 해당 크기의 버퍼 생성

```
const buffer = Buffer.from('저를 버퍼로 바꿔보세요');
console.log('from():', buffer);
console.log('length:',buffer.length);
console.log('toString():',buffer.toString());

const array = [Buffer.from('띄엄 '), Buffer.from('띄엄'), Buffer.from('띄어쓰
기')];
const buffer2 = Buffer.concat(array);
console.log('concat():', buffer2.toString());

const buffer3 = Buffer.alloc(5);
console.log('alloc():', buffer3);
```

파일 읽는 스트림 사용하기

- `fs.createReadStream`
  - createReadStream에 인자로 파일 경로와 옵션 객체 전달
  - highWaterMark 옵션은 버퍼의 크기(바이트 단위, 기본값 64KB)
  - **data**(chunk 전달), **end**(전달 완료), error(에러 발생) 이벤트 리스너와 같이 사용
    - data 이벤트 : 파일의 일부 리턴
    - end 이벤트 : 읽기가 완료되었을 때 호출
    - error 이벤트 : 에러가 발생했을 때 호출

```
const fs = require('fs');

const readStream = fs.createReadStream('./readme3.txt', {highWaterMark:16});
const data = [];
```

```
readStream.on('data', (chunk) => {
    data.push(chunk);
    console.log('data :', chunk, chunk.length);
});

readStream.on('end', () => {
    console.log('end :', Buffer.concat(data).toString());
});

readStream.on('error', (err) => {
    console.log('error :', err);
})
```

폼(form)

- 사용자의 데이터를 서버에 저장할 때 사용하는 방법
- 보통 로그인, 회원가입, 게시판에 글을 작성하거나 파일을 전송할 떄 사용

```
<form action = "URL" method = "데이터전송법">
    <!-- 텍스트, 라디오, 체크박스와 같은 컨트롤 생성 -->
</form>
```

  - action : 사용자가 입력한 데이터를 전송할 서버의 URL
  - method : 사용자가 입력한 데이터를 전송할 방법, get 방식과 post 방식이 대표적
    - get : action에 입력한 URL에 파라미터 형태로 전송(생략 가능)
    - post : header의 body에 포함해서 전송

get방식

- 클라이언트가 입력한 query의 이름과 값이 결합되어 스트링 형태로 서버에 전달
- 한번 요청시 전송 데이터 양은 주소값 + 파라미터로 **255자로 제한**
- DB에 추가로 정보를 처리하지 않고, 저장된 Data를 단순 요청하는 정도로 사용
- URL에 그대로 query의 이름과 값이 연결되어 표현

post방식

- 클라이언트와 서버간에 인코딩하여 서버로 전송
- 헤더를 통해 요청이 전송되는 방식
- 한번 요청시 데이터 양 제한 없음
- DB에 추가로 서버에서 갱신작업을 할 때, 서버에서 정보가 가공되어 응답하는 경우에 사용
- 클라이언트에서 데이터를 인코딩 ▶ 서버측에서 디코딩 해서 사용
- query는 body안에 들어가 있어서 보안에 조금 유리