



[스파르타코딩클럽] Node.js 심화반 - 5주차



매 주차 강의자료 시작에 PDF파일을 올려두었어요!

▼ PDF 파일

[수업 목표]

1. 실무에 도움이 되는 기술들을 익히고, 이것들이 왜 필요한지 알게 된다.

[목차]

- 01. 5주차 오늘 배울 것
- 02. 코드 서식 정리해보기(1)
- 03. 코드 서식 정리해보기(2)
- 04. 코드 분리하기
- 05. 소켓 연결 코드 분리하기
- 06. 테스트 코드에 대해 알아보기
- 07. Jest로 간단한 단위 테스트 코드 작성해보기 (1)
- 08. Jest로 간단한 단위 테스트 코드 작성해보기 (2)
- 09. 5주차 끝 & 숙제 설명



모든 토글을 열고 닫는 단축키

Windows : **Ctrl** + **alt** + **t**

Mac : **⌘** + **⌥** + **t**

01. 5주차 오늘 배울 것

- ▼ 1) 코드 서식을 일관적으로 관리하는 방법을 배운다.



여기서 말하는 코드 서식이라 함은, 코드의 형식(Format)이라고 보아도 좋습니다!

- ▼ 2) 코드를 나눠서 정리하는 방법을 배운다.



그동안 우리는 대부분의 코드를 app.js 파일에 작성했는데요, 파일을 어떤 기준으로 만들어서 코드를 나누는지 알아보는 시간을 가질 예정입니다!

- ▼ 3) 테스트코드가 무엇인지 알고, 단위 테스트코드를 작성하는 방법을 배운다.



테스트코드라는것이 조금은 생소하죠?

이건 여러분이 개발한 코드가 여러분이 의도한대로 동작하는지 작성하는 코드입니다!

내가 스스로 잘했는지 체크하기 위해 만들어두는 체크리스트와 비슷해요!

02. 코드 서식 정리해보기(1)

▼ 1) 코드 서식이 무엇일까요?

코드 서식은 여러분이 코드를 작성하는 방식을 말해요!

코딩 스타일(Coding Style)이라고 생각해도 되는데요, 생각보다 별것 아닌것같은 이 코드 서식은 오랜 시간동안 전세계의 개발자들의 화두에 오르내리는 주제입니다!

코드 서식이 무엇이 문제길래 이렇게 말이 많은걸까요?

1. 개발자들은 'Space, Tab'으로도 싸운다.
 - 개발자들, 왜 코드 빈 칸 때문에 싸우지?
 - 들여쓰기 100년 전쟁, Tab vs Space, 2글자 vs 4글자
2. 줄바꿈을 '특정 위치에서 하나 마나'로도 싸운다.
3. 코드 한 줄의 길이가 '몇자 이상을 넘어가면 된다 안된다'로도 싸운다.

이 외에도 특정 상황에서 중괄호를 사용하냐 마냐, 세미콜론을 쓰냐 마냐... 등 여러가지 사소한 보이는 부분으로 많이들 싸웁니다 😓 (이상해보이죠...? 여러분도 곧...)

여러분도 코드를 많이 작성하다 보면 본인의 코드 작성 스타일이 생기기 시작할텐데요, 자신만의 코드 스타일이 생기는건 좋지만, 언제나 "좋은 코드"를 작성하기 위해 노력하는것이 개발자의 미덕이죠.

전 "좋은 코드"중 일부가 가독성이 좋은 간결한 코드라고 봅니다.

우리는 이 미덕에 대해 알아보는 시간을 가져보겠습니다!! 🙌

▼ 2) 코드 서식을 일관적으로 관리하면 무엇이 좋을까요?

사람마다 똑같은 기능, 똑같은 내용의 코드를 작성하더라도 다른 모양으로 코드를 작성하게 되는데요, 그 예시가 아래와 같습니다.

▼ Case 1

```
sock.on("BUY", (data) => {
  const emitData = {
    ...data,
    date: new Date().toISOString(),
  };

  io.emit("BUY_GOODS", emitData);
});
```

▼ Case 2

```
sock.on("BUY", (data) => {
  io.emit("BUY_GOODS", { ...data, "date": new Date().toISOString() });
});
```

▼ Case 3

```
sock.on("BUY", (data) => {
  io.emit("BUY_GOODS", {
    ...data,
    date: new Date().toISOString(),
  });
});
```

▼ Case 4

```
sock.on('BUY', data => {
  io.emit('BUY_GOODS', { ...data, date: new Date().toISOString() });
});
```

위 코드는 모두 똑같은 동작을 하지만, 다른 형태로 작성되어 있습니다.

그냥 봤을때도 생각보다 많은 차이가 있지 않나요?

제가 보기엔 규칙성이 없는 코드도 있고, 보기 힘든 코드도 있는것 같아요!

여러분이 간단하게나마 보았듯 일관적인 코드 서식은 여러분의 오타를 줄여주고, 가독성을 크게 높여주며, 때로는 여러분이 더 전문적으로 보이게 돕기도 해요 😊

사람마다 취향이 갈리기는 하지만 많은 회사와 조직에서는 조금의 성능보단 가독성을 중요하게 생각합니다!

그렇기 때문에 조직에서는 서식을 관리해주는 도구를 사용하여 조금도 규칙에 어긋나지 않게 하기도 하죠 😊

우리는 다음 시간에 이 도구에 대해서 알아볼게요!

03. 코드 서식 정리해보기(2)

▼ 1) 코드 서식 관리 도구 Prettier(이하 프리티어)가 무엇인지 알아보기

▼ 프리티어가 어떤 기능을 하는지 실제로 체험해보기

프리티어 홈페이지에서 TRY IT ONLINE 버튼을 누르면, 실제로 프리티어를 사용해볼 수 있는 플레이그라운드 페이지가 뜹니다!

```

1 function HelloWorld({greeting = "hello", greeted = "World", silent = false, onMouseOver,}) {
2
3   if(!greeting){return null};
4
5   // TODO: Don't use random in render
6   let num = Math.floor(Math.random() * 1E+7).toString().replace(/\.\d+/, "")
7
8   return <div className="HelloWorld" title={ `You are visitor number ${ num }` } onMouseOver={onMouseOver}>
9
10    <strong>{ greeting.slice( 0, 1 ).toUpperCase() + greeting.slice(1).toLowerCase() }</strong>
11    {greeting.endsWith(",") ? " " : <span style={{color: 'grey'}}>"", "</span> }
12
13    <em>
14      { greeted }
15    </em>
16    { (silent)
17      ? " "
18      : "!"}
19
20    </div>;
21  }

```

```

1 function HelloWorld({
2   greeting = "hello",
3   greeted = "World",
4   silent = false,
5   onMouseOver,
6 }) {
7   if (!greeting) {
8     return null;
9   }
10
11   // TODO: Don't use random in render
12   let num = Math.floor(Math.random() * 1e7)
13     .toString()
14     .replace(/\.\d+/, "");
15
16   return (
17     <div
18       className="HelloWorld"
19       title={`You are visitor number ${num}`}
20       onMouseOver={onMouseOver}
21     >
22       <strong>
23         {greeting.slice(0, 1).toUpperCase() + greeting.slice(1).toLowerCase()}
24       </strong>
25       {greeting.endsWith(",") ? (
26         " "
27       ) : (
28         <span style={{ color: "grey" }}>"", "</span>
29       )}
30       <em>{greeted}</em>
31       {silent ? " " : "!"}
32     </div>
33   );
34 }
35

```

위 사진은 프리티어를 이용해 좌측의 코드가 우측의 형태로 자동으로 변환된 모습입니다.

어떤가요? 여러분이 보기에 읽기 쉬운 코드가 아닐 수도 있는데요, 그렇다면 여러분이 편한 규칙을 차근차근 적용해보면 좋습니
다!

물론 규칙을 적용하는 방법은 저와 같이 살펴볼거예요 🙏

▼ 어떤 언어에 사용할 수 있는지 살펴보기

최근(2021년 3월 기준) 공식 홈페이지에 나온 지원 언어는 아래와 같습니다!

Works with the Tools You Use

JS

- JavaScript
- JSX
- Flow
- TypeScript
- JSON

HTML

- HTML
- Vue
- Angular

CSS

- Less
- SCSS
- styled-components
- styled-jsx

GraphQL

- GraphQL
- GraphQL Schemas

Markdown

- CommonMark
- GitHub-Flavored Markdown
- MDX
- YAML

Community Plugins

- Apex
- Elm (via elm-format)
- Java
- PHP
- PostgreSQL
- Ruby
- Swift
- TOML
- XML

우리는 JavaScript, HTML, CSS를 중점적으로 사용하고 있는데 프리티어가 전부 기본적으로 지원해주므로 프리티어만 설치하
면 바로 사용할 수 있습니다!



위 사진에서 우측에 보이는 "Community Plugins"는 Prettier 개발사에서 직접 제공하지 않고, 해당 언어 커뮤니티의 개발자들이 따로 만들어놓은 third-party 플러그인으로 지원되는 언어이니 사용시 문제가 많을 수 있습니다! ★ 주의하세요!

▼ 어떤 규칙들을 제공하는지 살펴보기

문서에 따르면 프리티어가 기본적으로 제공하는 규칙은 아래와 같습니다.

```
module.exports = {
  trailingComma: "es5",
  tabWidth: 4,
  semi: false,
  singleQuote: true,
};
```

당연하지만, 이 규칙은 여러분의 입맛에 맞게 바꿀 수 있습니다!

그리고 위 규칙들이 무슨 역할을 하는지 프리티어를 직접 사용해보며 조금 알아보고 넘어갈게요!

▼ 2) 프리티어 사용 준비

▼ 프로젝트에 prettier 설치

```
npm i prettier -D
```

프리티어는 실제로 여러분의 서비스를 구동시킬때 필요한 모듈이 아니고, 여러분의 코드 작성을 조금 더 편리하게 해주는 도구이므로 devDependency로 설치하겠습니다!

▼ 설정파일 추가

app.js 파일이 있는 위치에 `.prettierrc.js` 라는 파일을 생성하고, 아래의 내용을 넣어주세요!

▼ [코드스니펫] .prettierrc.js 기본 설정

```
module.exports = {
  trailingComma: "es5",
  tabWidth: 4,
  semi: false,
  singleQuote: true,
};
```

▼ 스크립트 추가

이제 프리티어를 편리하게 사용할 수 있도록 `package.json` 파일에 스크립트를 추가해줄거예요!



`package.json` 파일에 있는 "scripts" 항목에 원하는 스크립트를 따로 추가해서 편리하게 단축키처럼 사용할 수 있습니다!

우선 아래의 내용에 있는것처럼 스크립트를 추가해주세요!

아래 코드스니펫을 복사해서 "scripts" 항목 안에 추가해주면 됩니다!

헷갈린다면 예시를 참고해보세요!

▼ [코드스니펫] prettify 스크립트 추가

```
"prettify": "prettier --write *.js **/*.js"
```

▼ 예시

```
{
  "name": "여러분의 프로젝트 이름일거예요",
  "version": "1.0.0",
  "scripts": {
    "start": "node app.js",
    "prettify": "prettier --write *.js **/*.js"
  }
}
```

```
},  
... 생략  
}
```



주의! scripts 내용 사이에는 예시처럼 항상 콤마(,)가 있어야 합니다!

만약 콤마를 빠트렸거나, 맨 마지막 항목에도 콤마를 넣었다면 정상적으로 동작하지 않을수도 있어요!

▼ 스크립트 실행해보기

```
npm run prettify
```

위 명령어를 터미널에서 입력해보세요!

만약 스크립트가 정상적으로 추가됐다면 아래와 비슷한 내용들이 주루룩 나올거예요!

```
tim@Timui-iMac ~/Projects/github/nodejs-shopping-api-demo week5 ➤ npm run prettify  
  
> nodejs-shopping-api-demo@1.0.0 prettify /Users/tim/Projects/github/nodejs-shopping-api-demo  
> prettier --write **/*.js  
  
events/index.js 54ms  
middlewares/auth-middleware.js 12ms  
middlewares/validation-middleware.js 7ms  
migrations/20210120151224-create-goods.js 7ms  
migrations/20210120160620-create-cart.js 10ms  
migrations/20210214073234-create-user.js 10ms  
models/cart.js 7ms  
models/goods.js 7ms  
models/index.js 14ms  
models/user.js 8ms  
routes/goods.js 26ms  
routes/users.js 16ms  
seeders/20210120153608-demo-goods.js 10ms  
static/api.js 34ms
```

▼ ☒ 혹시 이렇게 에러가 나나요?

만약 아래처럼 에러가 난다면 위 "스크립트 추가" 부분에서 `package.json` 파일을 수정하는 단계에서 오타가 났을 가능성이 매우 높아요!

```
tim@Timui-iMac ~/Projects/github/nodejs-shopping-api-demo week5 ➤ npm run prettify  
npm ERR! code EJSONPARSE  
npm ERR! file /Users/tim/Projects/github/nodejs-shopping-api-demo/package.json  
npm ERR! JSON.parse Failed to parse json  
npm ERR! JSON.parse Unexpected string in JSON at position 276 while parsing '{  
npm ERR! JSON.parse   "name": "nodejs-shopping-api-demo"  
npm ERR! JSON.parse Failed to parse package.json data.  
npm ERR! JSON.parse package.json must be actual JSON, not just JavaScript.  
  
npm ERR! A complete log of this run can be found in:  
npm ERR!   /Users/tim/.npm/_logs/2021-03-02T08_59_02_482Z-debug.log
```

이런 경우 package.json 파일에 콤마 또는 쌍따옴표가 정상적으로 열리고 닫혀 있는지 확인해보시길 바랍니다!

스크립트가 잘 실행됐다면 app.js 파일을 열어보세요!

코드의 서식이 자동으로 바뀌어 있을거예요!

▼ VS Code에서 사용하기

매번 스크립트를 직접 실행하는것도 귀찮을 수 있겠죠?

다행히 프리티어가 VS Code에서 사용할 수 있는 확장 플러그인을 제공하고 있어요!

▼ VS Code - Prettier 플러그인 설치하기

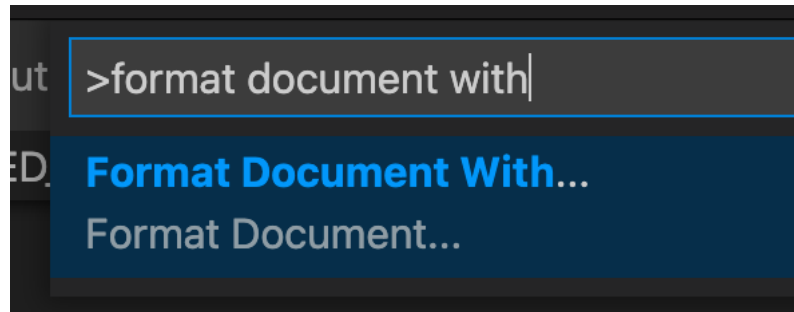
<https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode>

위 링크로 들어가 Install을 눌러서 설치를 완료해주세요!

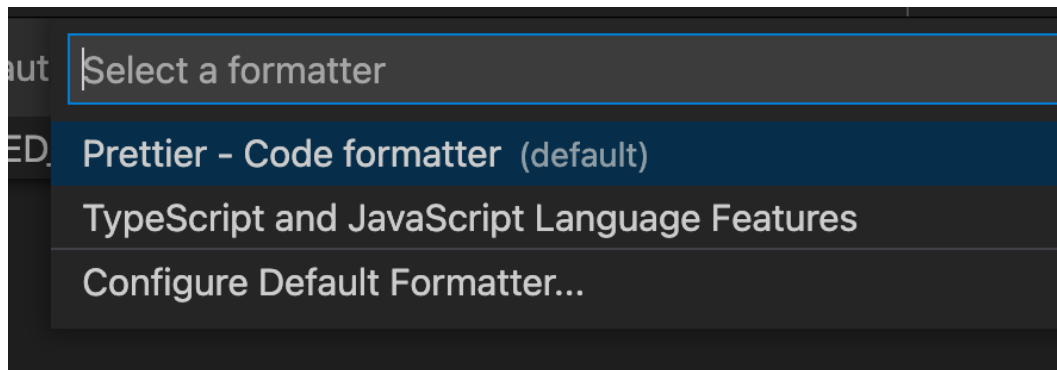
설치를 완료한 뒤에는 반드시 **VS Code**를 껐다가 켜 뒤, 프로젝트를 다시 열어주세요!

▼ Prettier 플러그인을 사용하도록 설정하기

1. 우선 Command Palette를 열어주세요! (단축키는 수업자료 맨 위에 적어두었어요)
2. "format document with" 라고 적은 뒤, 아래처럼 보이는 항목을 선택해주세요!



3. "Select a formatter" 라고 뜰텐데, 아래에 보이는 "Prettier - Code formatter" 를 선택해주세요!



▼ app.js 파일로 테스트해보기

우선 테스트를 위해서는 app.js 파일을 규칙에 맞지 않게 바꿔야만 해요!

▼ 1) app.js 파일 아무거나 수정

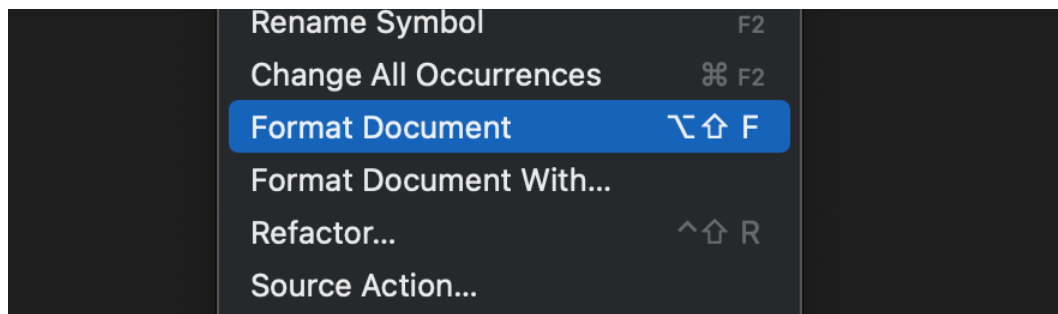
아래처럼 들여쓰기만 바꾼다던가, 세미콜론을 붙인다던가 하는 방식으로 코드에 에러는 없지만 이상하게 보이도록 바꿔주세요!

```

0
1 | 1 | < io.on('connection', (sock) => {
2 |   ... console.log('새로운 소켓이 연결됐어
3 |   ⚡
4 | 2 | < sock.on('BUY', (data) => {
5 | 3 | < const emitData = {
6 |   ... data,
7 |   date: new Date().toISOString(),
8 | }
9 |
0 |   io.emit('BUY_GOODS', emitData)
1 | }
2 |

```

- ▼ 2) app.js 파일 편집 창에서 오른쪽 마우스를 클릭해 "Format Document" 항목을 누릅니다.
아래처럼 생긴 버튼을 누르고 나면 app.js를 이상하게 수정했던 부분이 원래대로 돌아오게 됐을거예요!



- ▼ 3) 프리티어 규칙에 대해 조금 알아보기
우리는 아까 `.prettierrc.js` 파일을 만들었죠?
그 파일에 있는 "규칙"에 대해서 하나씩 알아보는 시간을 가지겠습니다!

💡 여기서는 지금 여러분의 환경에서 사용할 수 있는 규칙만 일부 소개하고 넘어가도록 하겠습니다!

▼ **trailingComma: "es5"**

- 코드 맨 뒤에 콤마(,)를 붙일수 있는 경우 콤마를 붙일지 말지 결정하는 규칙입니다!

```

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
1 | 2 | 3 | 4 | 5 | 6 | 7 |
2 | app.use("/api", express.urlencoded({
3 |   ... usersRouter,
4 |   ... goodsRouter
5 | });
6 | app.use(express.static("static"));
7 |
30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
31 | 32 | 33 | 34 | 35 | 36 | 37 |
32 | app.use("/api", e
33 |   ... usersRouter,
34 |   ... goodsRouter,
35 | });
36 | app.use(express.s
37 |

```

이렇게 붙여줍니다!

- **es5** 라는 값은 JavaScript 버전중 하나인 ES5에서 허용되는 부분까지만 코마를 뒤에 붙이는 규칙입니다.
- **none** 이라는 값은 가능한 모든 코드에서 코드 뒤에 붙은 코마를 제거합니다. (에러가 나도록 다 제거하는것은 아닙니다!)
- **all** 이라는 값은 ES8 이전의 버전까지 허용되는 모든 부분에 코마를 붙이도록 합니다.

아직 **all** 규칙에 해당하는 문법을 지원하지 않는 브라우저가 있기 때문에 현재 여러분에게 권장드리는 규칙은 "es5" 혹은 "none" 입니다.

▼ **tabWidth: 4**

- 들여쓰기 한번에 얼마만큼의 스페이스(여백)로 구성할지에 대한 설정입니다.



```

1  app.use("/api", express.urlencoded({ extend
2  usersRouter,
3  goodsRouter,
4  });
5  app.use(express.static("static"));
6
31 app.use("/api", express.urlencoded({ extend
32 usersRouter,
33 goodsRouter,
34 });
35 app.use(express.static("static"));
36

```

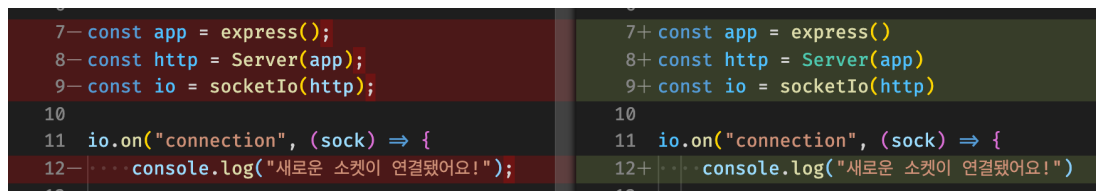
- 위처럼 2개의 여백으로 들여쓰기를 했던 파일이 4개의 여백으로 바뀌었습니다.

최근 스크립트 언어들의 들여쓰기는 2개~4개로 하는것이 일반적인 관례입니다.

만약 코드의 들여쓰기가 잘 구분이 안된다 하시면 4개, 코드가 길어지는게 싫다 하시면 2개를 권장합니다.

▼ **semi: false**

- 코드 뒤에 세미콜론을 붙일지 말지 결정하는 규칙입니다.
true인 경우 세미콜론을 항상 붙이도록 해줍니다.



```

7- const app = express();
8- const http = Server(app);
9- const io = socketIo(http);
10
11 io.on("connection", (sock) => {
12- console.log("새로운 소켓이 연결됐어요!");
13
7+ const app = express()
8+ const http = Server(app)
9+ const io = socketIo(http)
10
11 io.on("connection", (sock) => {
12+ console.log("새로운 소켓이 연결됐어요!")
13

```

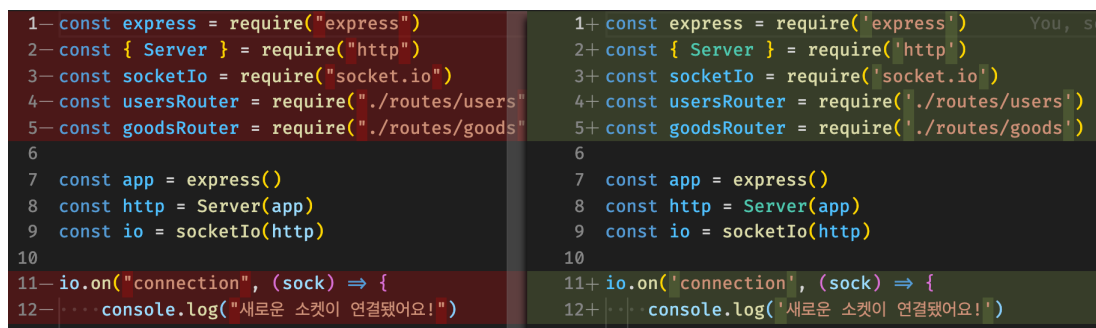
- 만약 이 설정을 false로 하는 경우 위 사진처럼 세미콜론이 모두 제거됩니다.
자바스크립트는 세미콜론이 없어도 정상적으로 동작할 수 있는 언어이기 때문에 존재할 수 있는 옵션입니다!

이 규칙은 항상 true로 하시길 권장드립니다.

코드가 어디서 끝났음을 하는지 코드의 맥락을 파악하는데 도움을 주는 중요한 요소이기 때문이죠 😊

▼ **singleQuote: true**

- 문자열을 표기하는 문법에서 외따옴표를 사용할지, 쌍따옴표를 사용할지의 여부를 설정하는 규칙입니다.



```

1- const express = require("express")
2- const { Server } = require("http")
3- const socketIo = require("socket.io")
4- const usersRouter = require("./routes/users")
5- const goodsRouter = require("./routes/goods")
6
7 const app = express()
8 const http = Server(app)
9 const io = socketIo(http)
10
11 io.on("connection", (sock) => {
12- console.log("새로운 소켓이 연결됐어요!")
13
1+ const express = require('express')
2+ const { Server } = require('http')
3+ const socketIo = require('socket.io')
4+ const usersRouter = require('./routes/users')
5+ const goodsRouter = require('./routes/goods')
6
7 const app = express()
8 const http = Server(app)
9 const io = socketIo(http)
10
11 io.on('connection', (sock) => {
12+ console.log('새로운 소켓이 연결됐어요!')
13

```

이 규칙은 true로 설정해두어 외따옴표를 사용하는것을 권장드립니다.

키를 매번 입력하다보면 쉬프트 키를 매번 누르는것이 상당히 번거롭고, 외따옴표를 썼을때 단점이 특별히 없기 때문입니다 😊

▼ **arrowParens: "always"**


- Arrow function에서 파라미터 부분에 괄호를 항상 붙여주는 규칙입니다.


<pre> 10 11- io.on('connection', sock => { 12 console.log('새로운 소켓이 연결됐어요!'); 13 14- sock.on('BUY', data => { 15 const emitData = { </pre>	<pre> 10 11+ io.on('connection', (sock) => { 12 console.log('새로운 소켓이 연결됐어요!'); 13 14+ sock.on('BUY', (data) => { 15 const emitData = { </pre>
---	---

괄호가 없었다가 생겨났죠??

이 규칙은 항상 "always"로 해두시는걸 권장합니다.

Arrow function이 아무리 편리하고 코드를 짧게 쓰도록 도와준다고 해도, 괄호가 있는것이 일관적인 코드를 작성하는것에 도움이 됩니다.

일관적인 코드는 무엇이라고 했죠? 가독성 항상  입니다!!

여기까지 잘 따라왔다면 여러분은 이제 아주 쉽게 예쁘고 일관적인 코드를 작성할 준비가 된 조금 더 프로페셔널한 개발자가 되었습니다! 

(단축키를 익혀두면 훨씬 더 좋겠죠?)

▼ 4) 정리하자면

최종적으로 제가 권장하는 기본 규칙은 아래와 같습니다!

```

module.exports = {
  trailingComma: 'es5',
  tabWidth: 2,
  semi: true,
  singleQuote: true,
  arrowParens: 'always',
};

```

제가 이런 규칙을 권장하는 이유는 각 규칙을 설명하면서 같이 말씀드렸는데요, 만약 취향에 맞지 않는다면 조금씩 바꿔보면서 본인의 코드 스타일을 정립해보면 되겠습니다!

04. 코드 분리하기



이제 여러분이 app.js에 작성했던 코드를 여러개의 파일로 나눠서 분리시킬 예정인데요, 먼저 이것을 하면 무엇이 좋은지 알아보고 가겠습니다!

▼ 1) 코드를 분리하는게 왜 필요할까요?

1. 가독성 향상: 코드를 분리하면서 추상화가 자연스럽게 되기 때문에 코드를 읽기가 훨씬 수월해집니다.
2. 관리(유지보수)의 용이: 어떤 함수가 어떤 역할을 갖는지 비교적 쉽게 파악할 수 있음

▼ 추상화가 무엇일까요?!

추상화란, 여러분이 작성하는 함수랑 비슷하다고 생각하시면 쉽습니다.

여러분이 "지금 서버와 소켓으로 연결된 모든 클라이언트에 메시지를 보내는 기능"을 구현하려면 어떻게 할까요?

```

// 1. 함수 생성
function sendMessageAll(message) {
  // 2. 메시지를 보내기 위한 로직 작성
  ~~~~
  ~~~~
}

```

위의 코드와 같은 모양으로 기능을 구현하려고 하지 않을까요?

저렇게 단순한 함수로 만든 기능도 "추상화"라고 부를수 있습니다!

내가 만든 `sendMessageAll` 라는 기능을 사용하는 다른 개발자는 `sendMessageAll` 라는 기능에서 코드가 어떻게 작성됐는지 굳이 알 필요가 없습니다.

내가 `socket.io` 라이브러리를 이용해서 기능을 구현했든, 직접 웹소켓을 만들어서 기능을 구현했든 신경쓰지 않아도 되고 동작만 하면 되는것이죠.

다른 사람들은 이제 단순히 `sendMessageAll` 라는 함수를 호출하면 "지금 서버와 소켓으로 연결된 모든 클라이언트에 메시지를 보내는 기능"을 수행하는것만 알고 있을뿐이죠.

여러분은 지금 "지금 서버와 소켓으로 연결된 모든 클라이언트에 메시지를 보내는 기능" 라는 내용을 `sendMessageAll` 라는 함수 하나로 추상화한 것입니다!

▼ 2) 어떤 기준으로 코드를 분리하면 좋을까요?

위에서 말했던것중 중요한 키워드가 몇개 있었는데요, 무엇일까요?

- 추상화
- 기능(함수)의 역할

위에 적어놓은 두개의 키워드를 꼽을수 있겠습니까!

그럼, 두개의 키워드를 기준으로 코드를 분리하려면 어떻게 해야할지 고민하면서 바로 코드를 정리해볼게요! (백날 고민하는것보다 실제로 정리된 코드를 보는게 이해가 빠르겠죠!)

05. 소켓 연결 코드 분리하기



앞서 언급한 키워드를 중심으로 코드를 분리해볼게요!

▼ 1) 추상화: 코드를 이해하기 쉽게 함수로 분리하기

이제 다시 쇼핑몰 프로젝트의 `app.js` 코드로 돌아와서, 아래의 코드를 정리해보도록 하겠습니다!

▼ 원래 코드

```
io.on('connection', (sock) => {
  console.log('새로운 소켓이 연결됐어요!');

  sock.on('BUY', (data) => {
    const emitData = {
      ...data,
      date: new Date().toISOString(),
    };

    io.emit('BUY_GOODS', emitData);
  });

  ... // 생략

  sock.on('disconnect', () => {
    console.log(sock.id, '연결이 끊어졌어요!');
  });
});
```

▼ 정리한 코드

짬! 저는 이렇게 정리해볼게요!

`on`, `emit` 이런 내용이 있는것보다 이 코드가 무슨 기능을 구현하려는지 파악하기 쉽지 않나요?

```
io.on('connection', (sock) => {
  const { watchBuying, watchByebye } = initSocket(sock);

  watchBuying();

  watchByebye();
});
```

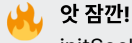
1. 서버에 소켓이 새로 연결되면,
(`io.on('connection', ~~~~)`)
2. 서버에서 소켓 연결할때 필요한 준비를 하고,
(`~~~ = initSocket(sock);`)

3. 구매를 하는지 감시하면서

```
(watchBuying());
```

4. 나가는지 감시한다.

```
(watchByebye());
```



앗 잠깐!

initSocket 이라는 함수가 없죠? 추상화 된 결과물만 보여드리기 위해 잠깐 숨겨두었어요!

▼ [코드스니펫] initSocket 코드

```
function initSocket(sock) {
  console.log('새로운 소켓이 연결됐어요!');

  // 특정 이벤트가 전달됐는지 감지할 때 사용될 함수
  function watchEvent(event, func) {
    sock.on(event, func);
  }

  // 연결된 모든 클라이언트에 데이터를 보낼때 사용될 함수
  function notifyEveryone(event, data) {
    io.emit(event, data);
  }

  return {
    watchBuying: () => {
      watchEvent('BUY', (data) => {
        const emitData = {
          ...data,
          date: new Date().toISOString(),
        };
        notifyEveryone('BUY_GOODS', emitData);
      });
    },

    watchByebye: () => {
      watchEvent('disconnect', () => {
        console.log(sock.id, '연결이 끊어졌어요!');
      });
    },
  };
}
```

이렇게 코드를 읽기 쉽도록 원래 있던 코드를 분리하면서, 새로운 함수를 만들어 나가는 과정을 "추상화"한다고 표현합니다!

이렇게 추상화 된 코드는 보통 역할에 맞는 파일에 묶여서 관리되는데요, 바로 해보겠습니다!

▼ 2) 기능의 역할: 특정 역할을 하는 코드들은 묶어서 파일로 분리할게요!

일단, initSocket 함수와 `io.on('connection', ~~~)` 처럼 생긴 소켓 연결과 관련된 코드를 따로 분리해볼까요?

차근차근 따라해주세요!

1. 소켓 관련된 코드만 따로 정리해둘 socket.js 파일 만들기
2. initSocket 함수와 `io.on('connection', ~~~)` 처럼 생긴 소켓 연결과 관련된 코드를 app.js 파일에서 socket.js 파일로 옮기기
3. initSocket 함수가 `const io = socketIo(http);` 객체를 필요로 하니 이 코드도 그대로 옮깁니다!
4. `const io = socketIo(http);` 에서 http 객체도 필요하니 app.js에서 http 객체를 모듈로 내보내줍니다!
5. socket.js 파일에서 app.js 파일이 내보내준 http 객체를 가져와서 사용하도록 해줍니다!

6. 끝일까요? 아쉽게도 아닙니다 😞

지금처럼 코드를 작성하면 아래와 같은 상황이 벌어집니다!

- `node app.js` 명령어로 실행해도 서버가 켜지고, `node socket.js` 명령어로 실행해도 서버가 켜집니다!!
- 단, app.js 파일으로 실행한 경우 소켓이 전혀 동작하지 않는 상황을 볼 수 있습니다.

🤔 뭔가 이상하죠? 이런 상황이 발생한 이유는 socket.js 파일이 app.js 파일을 가져다 사용하고 있는 상황에서 app.js 파일이 `http.listen` 함수를 실행하여 서버를 켜기 때문인데요, 어떻게 해결해야 할까요? 다시 차근차근 저를 따라하시면 됩니다!

7. 우선 app.js 파일에서는 서버를 켜지 않도록 `http.listen` 코드를 제거합니다.

8. server.js 파일을 생성합니다.

9. server.js 파일에서 app.js와 socket.js 파일을 참조합니다. 아래처럼요!

```
const http = require('./app');  
require('./socket'); // 이렇게 불러오기만 해도 소켓에 연결이 됩니다.
```

10. 참조만 하면 안되겠죠? 참조한 http 객체를 가지고 서버를 켜주세요!

```
http.listen(8080, () => {  
  console.log('서버가 요청을 받을 준비가 됐어요!');  
});
```

이제 server.js 파일을 실행해야만 서버가 켜지며, 서버가 켜질때는 항상 소켓까지 연결 준비가 완료된 상태가 되었습니다!!

또한 추상화를 통해 소켓 관련 코드의 가독성을 높이고, app.js 파일에 있던 소켓과 관련된 모든 코드를 socket.js 파일로 분리해서 app.js 파일의 가독성까지 높이게 되었습니다! 😊🎉

06. 테스트 코드에 대해 알아보기

▼ 1) 테스트 코드가 무엇일까요?

테스트 코드란 말 그대로 우리가 작성한 코드에 문제가 없는지 테스트하기 위해 작성하는 코드입니다.

흔히 테스트 기본 원칙이라고 불리는 "일곱 테스트 원칙"에서는 첫번째 규칙이 아래와 같습니다.

Testing shows the presence of defects, not their absence

테스팅은 결함이 없는것이 아니라, 결함의 존재를 보여주는것이다.

테스트 코드를 작성하는 방법은 앞으로 같이 알아볼것이지만, 여러분이 테스트 코드를 작성하기 전에 반드시 머릿속에 박아두어야 할 말입니다!

테스트 코드를 작성하는 목적은 "내 코드가 멀쩡하다!" 라고 증명하기 위함게 아니고, "내 코드가 멀쩡하다면 이렇게 결과가 나와야 한다!" 라고 생각하셔야 합니다.

하나의 기능을 여러번 수정하다 보면 예상치 못한 결과가 나오기 마련인데요, 테스트 코드를 이용하여 여러분의 코드를 믿고 사용할 수 있도록 해보시길 바랍니다.

아래와 같은 글들도 있으니 꼭 읽어보시길 바랍니다.

- <https://ssowonny.medium.com/설마-아직도-테스트-코드를-작성-안-하시나요-b54ec61ef91a>

▼ 2) 테스트 코드의 종류에 대해 알아보기

테스트 코드란, 어떠한 언어에 얽매이지 않고 개념적으로 표현하기도 합니다.

많은 개발자들이 오랜 기간동안 테스트 코드를 작성하며 아래와 같은 테스트 코드의 종류를 정립했으며, 우리는 이 중 가장 작은 기능을 테스트하는 "단위 테스트" 코드를 실제로 작성해볼 예정입니다 😊

- 단위 테스트 (Unit Test): 가장 작은 규모의 기능을 테스트합니다.
- 통합 테스트 (Integration Test): 여러가지 기능을 합쳤을때 생기는 문제를 방지하기 위한 테스트입니다.
- E2E 테스트 (End-to-end Test): 끝에서 끝(종단 간)을 의미하는 End to end 테스트입니다.
쉽게 말하면 백엔드부터 시작해서 최종적으로 웹 페이지가 원하는대로 동작하며 원하는 데이터를 잘 보여주는지 확인합니다.

이 외에도 수 많은 개발자가 돌연변이 테스트와 같은 다양한 기법들로 자신들의 서비스가 더욱 치밀하고 탄탄한 서비스로 거듭날 수 있도록 노력하고 있습니다 😊



번외) 넷플릭스의 자신들의 서버가 아무때나 무작위로 다운되도록 해서 자신들의 인프라에 약점이 있는지 찾아내는것을 도와주는 "카오스 몽키"라는 솔루션을 개발해서 사용했습니다. (카오스 엔지니어링)

참고로 제가 재직중인 이큐브랩에서도 이것과 비슷한 기능을 하는 환경을 구성해두고, 언제든지 서버가 죽더라도 최대한 빠르게 자동으로 서버를 다시 재구성하는 환경으로 운영중입니다.

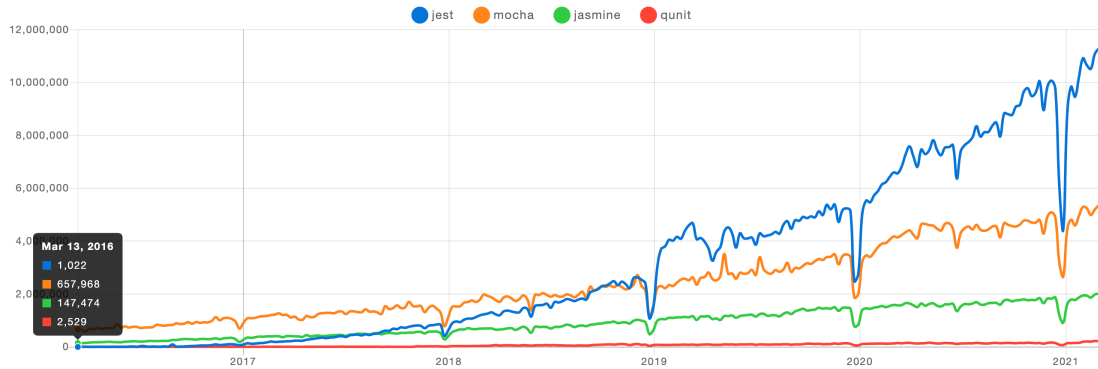
▼ 3) 테스트 프레임워크 Jest에 대해 알아보기

우리가 지금 Node.js를 통해 사용하고 있는 언어인 JavaScript에서 사용이 가능한 테스트 프레임워크의 종류는 매우 다양한데요, 이 중 페이스북에서 개발한 테스트 프레임워크인 Jest를 사용해보도록 하겠습니다!

▼ 왜 굳이 Jest를 써야할까요?

Jest는 출시한지 얼마 지나지 않아 주목받았는데, 페이스북에서 개발한 프론트엔드 라이브러리인 React.js와도 궁합이 아주 좋기 때문에 엄청난 성장세를 보이며 2020년 기준 JavaScript 개발자들 사이에서 가장 많이 사용되는 테스트 프레임워크로 뽑혔습니다.

Downloads in past 5 Years ▾



[테스팅 프레임워크 다운로드 수 지표] 파란색이 Jest, 그 외엔 원래 존재하던 테스트 프레임워크

▼ 왜 인기가 많은건데요?

Jest는 다른 테스트 프레임워크와 비교하면 여러가지 장점이 있지만, 그 중 가장 제일이라고 생각되는것은 테스트 코드의 표현이 다른 프레임워크보다 훨씬 간결합니다.

그렇기 때문에 여전히 많은 자바스크립트 개발자들에게 사랑받고 있습니다 😊

▼ 4) Jest를 실제로 사용해 볼 준비!

▼ 1) jest 모듈 설치

이번에도 Jest를 간단하게 사용해보기 위해 새로운 프로젝트를 구성해보겠습니다!

프로젝트로 사용할 폴더를 만들고, 그 폴더를 VS Code로 열어주세요!

그리고 언제나처럼 우리들의 친구, npm을 이용하여 모듈을 설치합니다! ([공식 문서 참고](#))

```
npm init
npm i jest -D
```

▼ 2) package.json 파일 수정하기

이 파일을 열어보시면, 아래와 같은 구문이 있을겁니다!

```
{
  ...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  ...
}
```

"test": 글자 뒤에 있는 내용을 지우고, 아래처럼 바꿔주세요!

```
{
  ...
  "scripts": {
    "test": "jest"
  },
  ...
}
```

이렇게 하면, `jest`를 `npm test` 와 같은 명령어로 실행시킬 수 있습니다.

또한 이렇게 하는것이 모범 사례로 꼽힙니다 😊

▼ 3) 테스트 할 간단한 함수 만들어보기

프로젝트 폴더 안에, `validation.js` 라는 이름으로 파일을 하나 생성해주세요!

그리고 아래의 코드 스니펫을 복사해 `validation.js` 파일에 붙여넣고 저장해주세요!

▼ [코드스니펫] 이메일 검증 함수 기반 코드

```
module.exports = {
  isEmail: (value) => {
    // value가 이메일 형식에 맞으면 true, 형식에 맞지 않으면 false를 return 하도록 구현해보세요
    return false;
  },
};
```

07. Jest로 간단한 단위 테스트 코드 작성해보기 (1)



이번 시간은 이전 시간에 `jest`를 사용해보기 위해 구성했던 프로젝트에 본격적으로 코드를 적어보는 시간입니다!

▼ 1) 단위 테스트 코드 파일 생성

자, 이제 테스트 코드를 작성할 파일을 생성할건데요!

일반적인 관례로 `테스트할파일이름.spec.js` 와 같은 형식으로 파일을 만듭니다.

`jest` 또한 위 형식의 이름을 가진 파일을 읽어들이 테스트 코드를 실행하는게 기본 설정이기도 합니다.

그러면 우리는 `validation.spec.js` 라는 이름으로 만들면 되겠죠? 만들어주세요!

▼ 2) 단위 테스트 코드 작성

테스트 파일은 만들었는데, 코드를 어떻게 작성할지 막막한가요? 걱정마세요 😊

`jest`가 테스트를 위해 기본적으로 제공하는 함수들이 있는데, 지금 우리가 자주 쓸 함수는 아래와 같습니다!

- `test()`: 단위 테스트를 묶어주는 함수입니다.
- `expect()`: 특정 값이 만족되는지(정상적인지) 확인하기 위한 표현식을 작성할수 있게 해주는 함수입니다.

이 설명만 보면 잘 모르겠죠? 아래 예시를 참고해주세요!

▼ [코드스니펫] 테스트코드 예시

```
const { isEmail } = require('./validation');

test('테스트가 성공하는 상황', () => {
  expect(isEmail('이메일이 아니에요')).toEqual(false);
});

test('테스트가 실패하는 상황', () => {
  expect(isEmail('my-email@domain.com')).toEqual(true);
});
```

위 예시를 `validation.spec.js` 에 붙여넣어 저장한 뒤, `npm test` 명령어를 실행했을때 아래와 같이 뜨면 정상적인 상황입니다!

> jest

FAIL ./validation.spec.js

✓ 테스트가 성공하는 상황 (2 ms)

✗ 테스트가 실패하는 상황 (2 ms)

● 테스트가 실패하는 상황

expect(received).toEqual(expected) // deep equality

Expected: true

Received: false

```
6 |  
7 | test('테스트가 실패하는 상황', () => {  
> 8 |     expect(isEmail('my-email@domain.com')).toEqual(true);  
9 | }  
10 | });
```

at Object.<anonymous> (validation.spec.js:8:44)

Test Suites: 1 failed, 1 total

Tests: 1 failed, 1 passed, 2 total

Snapshots: 0 total

Time: 0.817 s, estimated 1 s

Ran all test suites.

npm ERR! Test failed. See above for more details.

▼ 요구사항

- 입력한 이메일 주소에는 "@" 문자가 1개만 있어야 이메일 형식이다.
- 입력한 이메일 주소에 공백(스페이스)이 존재하면 이메일 형식이 아니다.
- 입력한 이메일 주소 맨 앞에 하이픈(-)이 있으면 이메일 형식이 아니다.

요구사항 1개마다 test 함수를 1개씩 만들고, 그 안에 테스트 코드를 작성해주세요!



위 요구사항에 해당하는 테스트 코드를 먼저 작성해주세요!

테스트가 실패해도 절대로 isEmail 함수를 먼저 수정하지 말아주세요!!

▼ 요구사항 작성 예시 (직접 작성해본 뒤에 열어보세요!!!)

```
const { isEmail } = require("./validation");  
  
test('입력한 이메일 주소에는 "@" 문자가 1개만 있어야 이메일 형식이다.', () => {  
    expect(isEmail("my-email@domain.com")).toEqual(true); // 1개만 있는 상황  
    expect(isEmail("my-email@@@domain.com")).toEqual(false); // 여러개 있는 상황  
    expect(isEmail("my-emaildomain.com")).toEqual(false); // 하나도 없는 상황  
});  
  
test("입력한 이메일 주소에 공백(스페이스)이 존재하면 이메일 형식이 아니다.", () => {  
    expect(isEmail("myemail@domain.com")).toEqual(true);  
    expect(isEmail("my email@domain.com")).toEqual(false);  
});  
  
test("입력한 이메일 주소 맨 앞에 하이픈(-)이 있으면 이메일 형식이 아니다.", () => {  
    expect(isEmail("e-m-a-i-l@domain.com")).toEqual(true);  
    expect(isEmail("-email@domain.com")).toEqual(false);  
});
```

▼ 3) 테스트 코드 실행

요구사항을 테스트 코드로 잘 표현해보셨나요?

이번에도 똑같이 `npm test` 명령어를 이용해 테스트 코드를 실행해보세요!

▼ 제가 작성한 코드는 이렇습니다!

```
const { isEmail } = require("../validation");

test('입력한 이메일 주소에는 "@" 문자가 1개만 있어야 이메일 형식이다.', () => {
  expect(isEmail("my-email@domain.com")).toEqual(true); // 1개만 있는 상황
  expect(isEmail("my-email@@@domain.com")).toEqual(false); // 여러개 있는 상황
  expect(isEmail("my-emaildomain.com")).toEqual(false); // 하나도 없는 상황
});

test("입력한 이메일 주소에 공백(스페이스)이 존재하면 이메일 형식이 아니다.", () => {
  expect(isEmail("myemail@domain.com")).toEqual(true);
  expect(isEmail("my email@domain.com")).toEqual(false);
});

test("입력한 이메일 주소 맨 앞에 하이픈(-)이 있으면 이메일 형식이 아니다.", () => {
  expect(isEmail("e-m-a-i-l@domain.com")).toEqual(true);
  expect(isEmail("-email@domain.com")).toEqual(false);
});
```

```
> jest

FAIL ./validation.spec.js
  ✕ 입력한 이메일 주소에는 "@" 문자가 1개만 있어야 이메일 형식이다. (4 ms)
  ✓ 입력한 이메일 주소에 공백(스페이스)이 존재하면 이메일 형식이 아니다.
  ✓ 입력한 이메일 주소 맨 앞에 하이픈(-)이 있으면 이메일 형식이 아니다. (1 ms)

  ● 입력한 이메일 주소에는 "@" 문자가 1개만 있어야 이메일 형식이다.

    expect(received).toEqual(expected) // deep equality

    Expected: true
    Received: false

      2 |
      3 | test('입력한 이메일 주소에는 "@" 문자가 1개만 있어야 이메일 형식이다.', () => {
    > 4 |     expect(isEmail('my-email@domain.com')).toEqual(true); // 1개만 있는 상황
        |                                             ^
      5 |     expect(isEmail('my-email@@@domain.com')).toEqual(false); // 여러개 있는 상황
      6 |     expect(isEmail('my-emaildomain.com')).toEqual(false); // 하나도 없는 상황
      7 |   });

      at Object.<anonymous> (validation.spec.js:4:44)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 2 passed, 3 total
Snapshots:   0 total
Time:        1.351 s
Ran all test suites.
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! hello-jest@1.0.0 test: `jest`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the hello-jest@1.0.0 test script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.
```

위 사진처럼 failed가 있도록. 즉, 실패하는 테스트가 단 하나라도 있도록 작성해주세요!!

우리가 항상 false를 return 하도록 구현해둔 함수이기 때문이긴 하지만 우리는 지금부터 테스트 코드를 통해 여러분의 코드에 결함이 있다는것을 스스로 증명할 수 있게 되었어요! 🎉

개발자는 항상 모든걸 확인하고 증명하는 역할을 잊지 마세요!



만약 항상 모든 코드가 성공한다면 지금 작성한 테스트 코드에 빈틈이 많으며, 자신의 코드의 결함을 찾지 못하는 테스트 코드를 작성하신거랍니다!

▼ 4) 테스트 코드가 통과하도록 isEmail 함수 코드 디버깅

실패하는 테스트가 있었나요? 만약 없었다면 다시 위로 올라가서 요구사항을 만족시키며 테스트가 통과하는 내용이 있도록 작성해보세요!

그리고 실패하는 테스트가 있었다면 이제 테스트가 통과할 수 있도록 isEmail 함수를 수정해볼건데요, 이렇게 자신의 코드에 문제(Bug)가 있는것을 찾아서 증명하고, 문제(Bug)를 고치는 행위를 디버깅(Debugging)이라 부릅니다!

그럼 isEmail 함수가 제대로 동작할 수 있도록 디버깅해주세요!

함수를 제대로 고쳤다고 판단하면 테스트 코드를 실행해서 테스트가 모두 통과하는지 보는것입니다!

▼ isEmail 함수 디버깅 예시

```
module.exports = {
  isEmail: (value) => {
    const email = (value || '');

    if (email.split('@').length !== 2) {
      return false;
    } else if (email.includes(' ')) {
      return false;
    } else if (email[0] === '-') {
      return false;
    }

    return true;
  },
};
```

```
> jest
PASS ./validation.spec.js
  ✓ 입력한 이메일 주소에는 "@" 문자가 1개만 있어야 이메일 형식이다. (2 ms)
  ✓ 입력한 이메일 주소에 공백(스페이스)이 존재하면 이메일 형식이 아니다.
  ✓ 입력한 이메일 주소 맨 앞에 하이픈(-)이 있으면 이메일 형식이 아니다.

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.141 s
Ran all test suites.
```

08. Jest로 간단한 단위 테스트 코드 작성해보기 (2)

▼ 5) 테스트 코드가 실패할 수 있도록 단위 테스트 코드 추가 작성

여러분이 작성한 코드가 잘 동작하는것을 확인하셨나요?

그런데 갑자기 기획자가 이메일 주소 양식을 변경하고 싶다고 하는 상황이 발생해버렸어요! 📢

추가된 요구사항은 아래와 같습니다.

▼ ☒ 추가 요구사항

- 입력한 이메일 주소중, 로컬 파트(골뱅이 기준 앞부분)에는 영문 대소문자와 숫자, 특수문자는 덧셈기호(+), 하이픈(-), 언더바(_) 3개 외에 다른 값이 존재하면 이메일 형식이 아니다.
- 입력한 이메일 주소중, 도메인(골뱅이 기준 뒷부분)에는 영문 대소문자와 숫자, 점(.), 하이픈(-) 외에 다른 값이 존재하면 이메일 형식이 아니다.

여러분은 다행히 테스트 코드를 작성해두었기 때문에 지금 구현해둔 함수가 위 요구사항에 맞게 동작하는지 쉽게 확인할 수 있어요!! 그러면, 위 요구사항을 만족시키도록 테스트 코드를 추가로 작성해주세요!

▼ 추가 요구사항을 만족시키는 테스트 코드 예시

```

...
test("입력한 이메일 주소중, 로컬 파트(괄뱅이 기준 앞부분)에는 영문 대소문자와 숫자, 특수문자는 덧셈기호(+), 하이픈(-), 언더바(_) 3개 외에 다른 값이 존재하면", () => {
  expect(isEmail(' _good-Email+test99@domain.com')).toEqual(true);
  expect(isEmail('my$bad-Email9999@domain.com')).toEqual(false);
});

test("입력한 이메일 주소중, 도메인(괄뱅이 기준 뒷부분)에는 영문 대소문자와 숫자, 하이픈(-) 외에 다른 값이 존재하면 이메일 형식이 아니다.", () => {
  expect(isEmail('my-email@my-Domain99.com')).toEqual(true);
  expect(isEmail('my-email@my_Domain99.com')).toEqual(false);
  expect(isEmail('my-email@my$Domain99.com')).toEqual(false);
});

```

만약 아래처럼 테스트가 실패한다면 잘 작성했다고 볼 수 있습니다!

```

> jest
FAIL ./validation.spec.js
  ✕ 입력한 이메일 주소에는 "공" 문자가 1개만 있어야 이메일 형식이다. (2 ms)
  ✕ 입력한 이메일 주소에 공백(스페이스)이 존재하면 이메일 형식이 아니다. (1 ms)
  ✕ 입력한 이메일 주소 맨 앞에 하이픈(-)이 있으면 이메일 형식이 아니다.
  ✕ 입력한 이메일 주소중, 로컬 파트(괄뱅이 기준 앞부분)에는 영문 대소문자와 숫자, 특수문자는 덧셈기호(+), 하이픈(-), 언더바(_) 3개 외에 다른 값이 존재하면 이메일 형식이 아니다. (2 ms)
  ✕ 입력한 이메일 주소중, 도메인(괄뱅이 기준 뒷부분)에는 영문 대소문자와 숫자, 하이픈(-) 외에 다른 값이 존재하면 이메일 형식이 아니다. (1 ms)

  ● 입력한 이메일 주소중, 로컬 파트(괄뱅이 기준 앞부분)에는 영문 대소문자와 숫자, 특수문자는 덧셈기호(+), 하이픈(-), 언더바(_) 3개 외에 다른 값이 존재하면 이메일 형식이 아니다.
    expect(received).toEqual(expected) // deep equality

    Expected: false
    Received: true

    19 |   test("입력한 이메일 주소중, 로컬 파트(괄뱅이 기준 앞부분)에는 영문 대소문자와 숫자, 특수문자는 덧셈기호(+), 하이픈(-), 언더바(_) 3개 외에 다른 값이 존재하면 이메일 형식이 아니다.", () => {
    20 |     expect(isEmail(' _good-Email+test99@domain.com')).toEqual(true);
    21 |     expect(isEmail('my$bad-Email9999@domain.com')).toEqual(false);
    22 |   });
    23 |
    24 |   test("입력한 이메일 주소중, 도메인(괄뱅이 기준 뒷부분)에는 영문 대소문자와 숫자, 하이픈(-) 외에 다른 값이 존재하면 이메일 형식이 아니다.", () => {
      |                                     ^
    25 |     expect(isEmail('my-email@my-Domain99.com')).toEqual(true);
    26 |     expect(isEmail('my-email@my_Domain99.com')).toEqual(false);
    27 |     expect(isEmail('my-email@my$Domain99.com')).toEqual(false);
    28 |   });
    29 |
      |                                     ^
    at Object.<anonymous> (validation.spec.js:21:58)

  ● 입력한 이메일 주소중, 도메인(괄뱅이 기준 뒷부분)에는 영문 대소문자와 숫자, 하이픈(-) 외에 다른 값이 존재하면 이메일 형식이 아니다.
    expect(received).toEqual(expected) // deep equality

    Expected: false
    Received: true

    24 |   test("입력한 이메일 주소중, 도메인(괄뱅이 기준 뒷부분)에는 영문 대소문자와 숫자, 하이픈(-) 외에 다른 값이 존재하면 이메일 형식이 아니다.", () => {
    25 |     expect(isEmail('my-email@my-Domain99.com')).toEqual(true);
    26 |     expect(isEmail('my-email@my_Domain99.com')).toEqual(false);
    27 |     expect(isEmail('my-email@my$Domain99.com')).toEqual(false);
    28 |   });
    29 |
      |                                     ^
    at Object.<anonymous> (validation.spec.js:26:47)

Test Suites: 1 failed, 1 total
Tests:       2 failed, 3 passed, 5 total
Snapshots:   0 total
Time:        1.228 s
Ran all test suites.
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! hello-jest@1.0.0 test: `jest`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the hello-jest@1.0.0 test script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.

```

▼ 6) isEmail 함수 다시 디버깅

우리가 작성한 함수가 요구사항을 만족시키지 못하는것을 확인했으니, 테스트가 통과하도록 다시 디버깅을 해주세요!

우리는 테스트 코드를 작성해두었기 때문에 코드를 마음대로 바꿔도 버그가 생겼는지 안생겼는지 쉽게 체크할 수 있습니다!

그리고 항상 그렇지만, 직접 코드를 작성한 뒤 예시 코드를 참고해보시면 더 좋습니다!

▼ 특정 문자열 외에 다른 문자열이 있는지 어떻게 알아낼까요?

항상 정해진 방법은 없습니다!

두개의 방법을 놓고 볼 때 어떤게 더 낫냐를 판단하기는 하지만, 여러분의 능력과 지식을 기반으로 참신한 방법을 떠올려보세요!! (검색도 여러분의 능력입니다.)

▼ 10분 이상 고민하고 검색해봐도 진짜 모르겠다... 하시면 펼쳐서 보세요!

1. "정규식" 이라는것을 이용하면 아주 짧은 코드로 검사할 수 있습니다.
(단, 제대로 알고 사용해야 이상한 문제에 부딪히지 않습니다. 정규식을 이용해 검사해보고싶다면 공부해보세요!)
2. 허용하는 문자를 전부 제거했는데도 남은 문자열이 있으면? 허용하지 않은 문자를 포함했기겠죠?
ex) "Abcd_1234-abc+\$" → (영문 대소문자, 숫자, 언더바, 하이픈, 덧셈기호 모두 제거) → "\$"
3. 문자열을 하나씩 쪼개고, 각 문자가 허용하는 문자에 포함되는지 체크하면 어떨까요?

코드를 모두 작성했다면 테스트를 돌려서 잘 동작하는지 확인해보세요!

▼ 추가 요구사항에 대한 isEmail 함수 디버깅 예시

```

module.exports = {
  isEmail: (value) => {
    const email = value || "";
    const [localPart, domain, ...etc] = email.split("@");

```

```

    if (!localPart || !domain || etc.length) {
      return false;
    } else if (email.includes(" ")) {
      return false;
    } else if (email[0] === "-") {
      return false;
    }

    for (const word of localPart.toLowerCase().split("")) {
      if (!["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "-", "_", ".", "!", "#", "$", "%", "&*", ":", ";", "<", ">"].includes(word)) {
        return false;
      }
    }

    for (const word of domain.toLowerCase().split("")) {
      if (!["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "-", "_", ".", "!", "#", "$", "%", "&*", ":", ";", "<", ">"].includes(word)) {
        return false;
      }
    }

    return true;
  },
};

```

▼ 추가 요구사항에 대한 isEmail 함수 디버깅 예시 (정규식 버전)

여러분이 혹시라도 궁금해 할까봐 준비해왔습니다! 😊

코드가 엄청나게 간결해졌죠? 😎 정규식은 잘 공부해두면 쓸모가 굉장히 많기 때문에 학습하시길 권장드려요!

```

module.exports = {
  isEmail: (value) => {
    const email = value || "";
    const [localPart, domain, ...etc] = email.split("@");

    if (!localPart || !domain || etc.length) {
      return false;
    } else if (email.includes(" ")) {
      return false;
    } else if (email[0] === "-") {
      return false;
    } else if (!/^[a-z0-9+_-]+$/.test(localPart)) {
      return false;
    } else if (!/^[a-z0-9.-]+$/.test(domain)) {
      return false;
    }

    return true;
  },
};

```

▼ 7) 테스트 코드에 대한 조언

- 테스트 코드 작성은 빨리 시작할수록 좋습니다. 테스트 코드를 처음 작성하는데 시간이 오래 걸린다고 **절대로 귀찮아하지 마세요.** 테스트 코드가 있었으면 방지할 수 있는 상황이 끝도 없이 발생합니다 😊
- 테스트 코드를 작성했는데도 자신의 코드를 수정하는게 불안하다면, 지금 작성한 테스트 코드가 정말로 **"결함을 찾도록 도와주고 있는지"** 확인해보세요.
- 테스트 코드는 여러분이 **"질 떨어지는 코드"**를 작성하는것을 막아주는 **방파제**가 됩니다. 나중에 자신이 다니고 있는 회사에서 코드 리뷰나 좋은 선임 개발자가 없어서 내 코드의 질이 떨어지는것 같다고 생각하는 순간이 온다면 **테스트 코드를 잘 작성하고 있는지 한번쯤 돌아봐야 합니다.**
- 테스트 코드를 작성하기 어렵거나 애매한 기능이 있다면 **추상화가 해결해줄지도 모릅니다.**
- 테스트 코드를 작성할때는 비즈니스 로직을 중심으로 작성해보세요. 개발자의 입장에서 테스트 코드를 작성하는것보다, 내가 만들고 있는 서비스의 기능을 중심으로 테스트 코드를 작성하면 기획자가 신규 개발을 요청할때만 변경될것입니다.
- 테스트 코드의 작성이 조금 익숙해졌다면 TDD와 BDD가 무엇인지 찾아보세요.
- 단위 테스트 코드를 작성하는 경우, 테스트하려는 기능의 코드보다 해당 기능을 테스트하는 테스트 코드가 더 훨씬 많은것이 모범 사례라고 꼽힙니다.
- 테스트 코드는 몇번을 실행하든 같은 결과가 나와야 합니다.

09. 5주차 끝 & 숙제 설명

▼ 1) jest를 이용하여 스파르타 쇼핑물 테스트 코드 작성하기

- express 서버가 문제 없이 켜질 수 있는지 확인하는 테스트 코드를 작성해보세요!
- 사용자 인증 미들웨어가 토큰을 잘 검증하고, 사용자 정보를 제대로 불러올 수 있는지 확인하는 테스트 코드를 작성해보세요!
 - 실제로 DB를 연결하도록 테스트 하지 말고, "mock" 이라는것을 찾아보고, 실제 DB 대신 Mocking된 DB에 있는 사용자 정보를 불러올 수 있도록 해보세요!

아래는 숙제를 하면서 참고 할만한 글입니다.

- <https://github.com/goldbergonyi/javascript-testing-best-practices>
- <https://dev.wisedog.net/2018/04/04/how-to-use-jest-with-express-js/>
- <https://javascript.plainenglish.io/how-to-unit-test-express-middleware-typescript-jest-c6a7ad166e74>
- <https://lgphone.tistory.com/100>

▼ 2) 개발자로서 성장하고 싶은 분들을 위한 조언

아래에서 밑줄이 있는 키워드는 직접 찾아보면서 이게 무엇인지 제대로 기억하시는걸 추천합니다.

- 테스트 코드는 지금 당장 작성하기 시작해보세요.
- 항상 새로운것을 받아들이고, 핵심이 뭔지 파악하려고 노력하세요. 빠르게 성장할 수 있는 지름길입니다.
- 좋은 도구를 잘 쓰지만 해도 금방 현업 개발자들 만큼 잘 할 수 있습니다.
 - 장인은 도구탓을 하지 않는다는것은 옛말!
개발자는 도구를 직접 만들고 사용해서 더 효율적으로 더 좋은 도구를 만드는 사람입니다.
 - 대부분의 현업 개발자들은 이 "좋은 도구"들에 대한 경험이 풍부하며, 잘 사용하는 사람들이기도 합니다.
- 좋은 도구를 잘 쓰기 위해서는 도구를 잘 이해하고 사용해야 합니다.
 - 모두가 좋다고 떠들어대서 무턱대고 도구를 사용하면 큰 화를 입을 수 있습니다.
- 새로운 기술을 적용할 때는 **오버 엔지니어링**을 더욱 조심하세요.
- 대부분의 회사에서 **기술 부채**는 없을 수 없습니다.
다만, 기술 부채가 무작정 늘어만 간다면 잘못되고 있다는 신호일 수 있습니다.
"6개월동안 기술 부채를 없애려고 집중했을 때 전부 없앨수 있다" 라고 생각될 정도로만 유지하세요.
- **더 좋은 인터페이스**를 만들수 있도록 훈련하세요.
 - 인터페이스는 함수의 이름, 함수의 인자, 반환 값 모두 포함되며, API 조차 인터페이스라는 사실을 잊지 마세요.
- 실제로 서비스를 운영한다고 생각하고 사이드 프로젝트를 진행해보세요. 사용자의 입장에서 생각하고 개발해보세요.
당장은 코드를 더 많이 짜야하고 고려할게 많더라도, 이렇게 하는것이 장기적으로 유지보수에 더 도움됩니다.
 - 사용자의 입장에서 더 좋은 서비스라고 생각되게 도와주는 설계나 개발론 또는 기법들이 있으니 찾아보고 공부해보세요.
ex) **BDD(Behavior Driven Development)**, **DDD(Domain Driven Design)**
- 절대로 여러분만 **유지보수**가 가능한 코드를 짜지 마세요.
지금의 여러분은 유지보수가 가능한 코드일지 몰라도, 1년 뒤의 자신이 보았을 때는 정말 이상한 스파게티 코드로 보일겁니다.
즉, 여러분만 유지보수 가능한 코드는 그 당시에만 유지보수가 가능할 정도로 엉망이라는 뜻입니다.
- 보이카우트 원칙을 몸에 베도록 하세요.
캠프장은 처음 왔을 때보다 더 깨끗하게 해놓고 떠나라.
- 자신이 만들 서비스를 어떻게 구현할지 모르겠다면, 최대한 비슷한 유명한 서비스를 찾아서 그 서비스가 어떻게 만들어졌는지 찾아보세요.
만약 그것이 오픈소스라면 설계를 어떻게 했는지까지 파악할 수 있을겁니다.