플렉스팀 프론트엔드 기술 스택의 이해: `lint`, `build`, `run`

2020-11-17

안희종 (플렉스팀 프론트엔드 챕터)

목표

- 플렉스팀 프론트엔드 프로젝트에서 사용 중인 기술 관련 아래 내용 소개:
    1. 어떤 문제를 풀기 위해 나왔는가?
    2. 어떤 기본 개념을 알아야 하나?
- **수정이 필요해졌을 때 모든 팀원이 어딜 봐야할 지 알 수 있도록!**
- React, Redux, styled-components 등 다 아실법한 주제는 다루지 않습니다
- 구체적인 동작 원리나 설정법 등은 추가 자료(👀)를 참고하세요

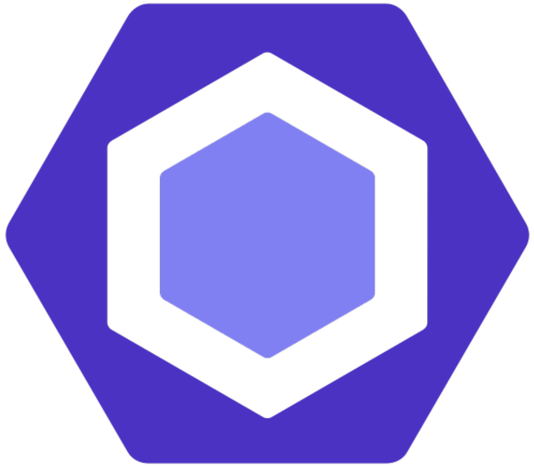
목차

`lint`

- eslint
- prettier
- lint-staged + husky

“코드베이스 일관성이 떨어진다!”

`lint`: eslint



`lint`: eslint – 설정

```
module.exports = {
  parser: '@typescript-eslint/parser',

  settings: {
    react: { version: 'detect' },
    'import/internal-regex': /^@flex/,
   },

  extends: [
    'eslint:recommended',
    'plugin:react/recommended',
    'plugin:prettier/recommended',
  ],

  plugins: ['@typescript-eslint', 'import', 'prettier'],

  rules: {
    'no-console': ['error', { allow: ['info', 'warn', 'error'] }],
    'react/jsx-filename-extension': ['error', { extensions: ['.js', 'tsx'] }],
    'import/order': [
      'error',
      {
        'newlines-between': 'always',
        groups: ['builtin', 'external', 'internal', 'parent', 'sibling', 'index'],
        pathGroups: [
          { pattern: 'react*', group: 'external', position: 'before', },
        ],
      },
    ],
  },

  overrides: [
    {
      files: ['*.ts', '*.tsx'],
      settings: {
        'import/resolver': { typescript: {} },
      },
    },
  ],
};
```

- **rule**
  - 실제 적용할 규칙 및 규칙 옵션
- **plugins**
  - 린트 규칙, 설정 등을 제공하는 묶음
- **extends**
  - 사전에 정의된 설정(`.eslintrc`)을 확장
- **overrides**
  - 특정 파일에 다른 룰을 적용
- 👀 공식 문서

“타입스크립트 파일도
린트 돌리고 싶어!”

`lint`: tslint

“eslint랑 tslint랑 비슷한 일을
하는데 왜 두 번씩 돌려야 되냐!”

`lint`: eslint – typescript-eslint

“린트 및 포매팅을 매번
손으로 맞추려니 귀찮다!”

`lint`: prettier

- “포매팅은 사람이 아니라 기계가 할 일!”
- 자동화된 코드 포매터
- 누가 작성해도 동일한 포맷의 코드가 나옴

`lint`: prettier – 설정

```
{
  "trailingComma": "es5",
  "tabWidth": 2,
  "semi": true,
  "singleQuote": true,
  "arrowParens": "avoid"
}
```

- “Prettier has a few options because of history. **But we don’t want more of them.** (…) By far the biggest reason for adopting Prettier is to stop all the on-going debates over styles.”
- 설정할 수 있는 값이 많지 않다
- 어차피 사람이 신경쓰지 않는 것이 목적이라 한 번 정하고 나면 별로 건드릴 일이 없다

“eslint랑 prettier랑 비슷한 일을 하는데 왜 두 번씩 돌려야 되냐!”

`lint`: eslint + prettier

“CI에서만 돌렸더니 매번 빌드가 깨지고
그 이후에 린트 고치는 커밋 추가로 하게 된다!”

“자주 돌리자니, 파일 한 개만 바꿨는데
전체 프로젝트 린트 돌길 기다리려니 너무 느리다!”

`lint`: lint-staged + husky

```
{
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },
  "lint-staged": {
    "*.{js,jsx,ts,tsx}": [
      "eslint --ext .js,.jsx,.ts,.tsx --fix",
    ]
  }
}
```

- **husky**
  - “Git hooks made easy”
- **lint-staged**
  - “Run linters against staged git files and don't let 💩 slip into your code base!”

`build`

- babel
- webpack
- Sentry

“옛날 브라우저에서도
편리한 새 ECMAScript 스펙을
쓰고 싶어!”

`build`: babel

- 시작은 ES6 ➡ ES5 컴파일러(6to5)
- 추후 babel 로 이름 변경했지만 하는 일은 비슷함
- 최신 스펙 및 프로포절을 포함하는 자바스크립트 코드를 ES5 환경에서 잘 동작하도록 컴파일

`build`: babel – 설정

```
{
  "presets": ["next/babel", "@zeit/next-typescript/babel"],
  "plugins": [
    [
      "@babel/plugin-proposal-object-rest-spread",
      {
        "loose": true,
        "useBuiltIns": true
      }
    ],
    ["styled-components", { "ssr": true }]
  ]
}
```

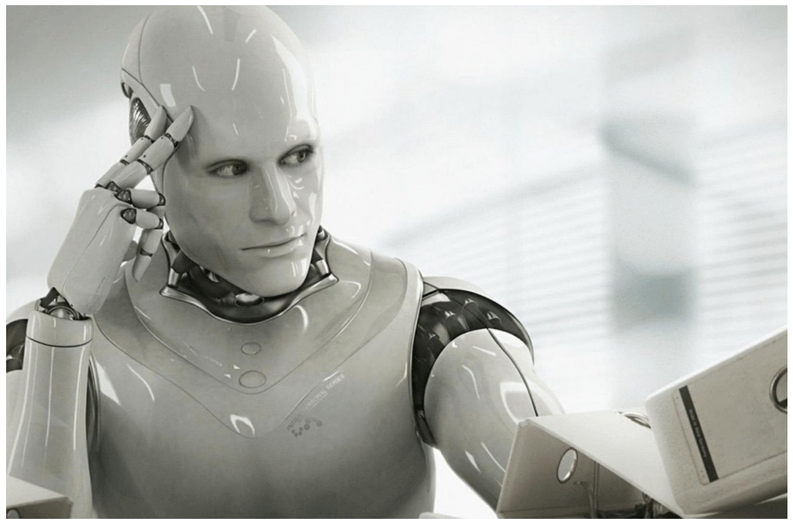
```
import "@babel/polyfill";
```

```
import "core-js/stable";
import "regenerator-runtime/runtime";
```

- 빌드 타임
  - `plugin`: 어떤 코드를 받아서 다른 코드를 내놓는다
  - `preset`: 자주 함께 쓰이는 플러그인의 묶음
- 런타임
  - `polyfill`: 코드 변경만으로는 처리 불가능한 빌트인/프로토타입 확장
- 👀 프론트엔드 기술 조감도 : Babel

“모든 변환/폴리필을 쓰긴 무겁고,
내가 지원하는 브라우저에 필요한 걸
세심하게 관리하긴 귀찮아!”

`build`: babel – `preset-env`



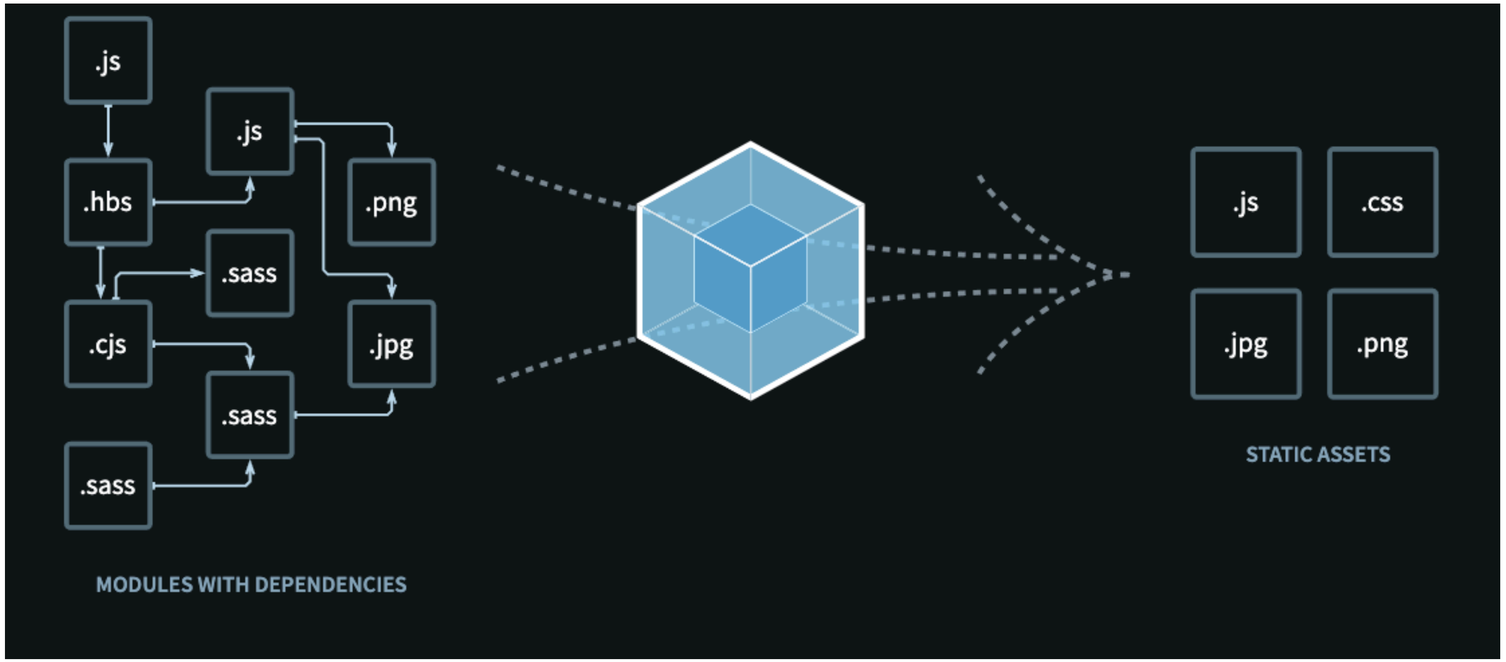
- “@babel/preset-env is a smart preset that allows you to use the latest JavaScript **without needing to micromanage which syntax transforms (and optionally, browser polyfills) are needed by your target environment(s)**.

- 지원하는 환경만 명시하면 (`browserslist`) 알아서 필요한 플러그인과 프리셋을 알려줌

“자바스크립트 프로젝트가 점점
복잡해져서 정리할 수단이 필요해!”

“파일을 나눠서 개발하고, 서빙할 땐
다양한 최적화를 적용하고 싶어!”

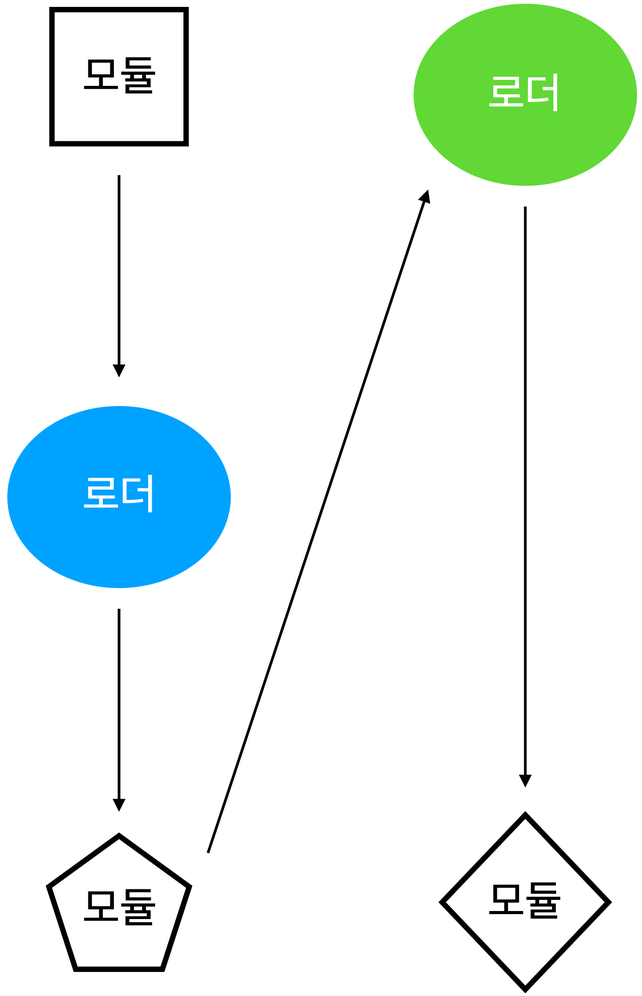
`build`: webpack



`build`: webpack – 로더

- 로더(모듈) = 모듈
- 로더는 한 **모듈**을 받아서 다른 **모듈**을 뱉는다
- `babel-loader`:
  최신 스펙의 JS 코드를 받아서
  예전 브라우저가 이해 가능한 JS 코드를 뱉는다
- `ts-loader`:
  TS 코드를 받아서
  JS 코드를 뱉는다

`build`: webpack – 로더



- 로더는 체이닝 될 수 있다
- `ts-loader` + `babel-loader`:
  TS 코드를 받아서
  내뱉은 최신 스펙의 JS 코드를 받아서
  예전 브라우저가 이해 가능한 JS 코드를 뱉는다
- 최종적으로 JS / JSON 형태로 떨어지면 됨

`build`: webpack – 로더

```
{
  test: /\\.css$/,
  use: [
    { loader: 'style-loader' },
    {
      loader: 'css-loader',
      options: {
        modules: true
      }
    },
    { loader: 'sass-loader' }
  ]
}
```

- `test`: 어떤 파일들을
- `use`: 어떻게 변환해라
- 예시:
  - `.css` 확장자를 갖는 파일을 찾아서 `sass-loader`, `css-loader`, `style-loader` 순으로 변환해라

`build`: webpack – 플러그인

- 플러그인은 **로더로 할 수 없는** 모든 동작을 수행한다
  - 모든 컴파일 단계 라이프사이클을 제공
- fork-ts-checker-plugin: 번들에 대해 타입 체크를 수행
- terser-webpack-plugin: 번들에 대해 Minification을 수행
- sentry-webpack-plugin: 번들에 대해 Sentry CLI의 역할을 수행
- antd-dayjs-webpack-plugin: 번들에 포함된 antd 속 moment.js 를 day.js로 대체

`build`: webpack – 그 외 설정

- `entry` / `output`
  - 어떤 파일로부터 의존성을 따라가기 시작하며, 결과 번들을 어디에 저장할지
- `resolve`
  - require 또는 import 문에 명시된 모듈 명을 어떻게 해석할지
- `mode`
  - 개발 환경인지 프로덕션 환경인지 (최적화, 소스맵 등 다양한 옵션에 영향을 줌)
- 👀 프론트엔드 개발환경의 이해 (웹팩) 기본 심화

“사용자들이 겪는 오류를 트래킹하고
잘 해결하고 싶어!”

`build`: Sentry

- 오류 발생시 관련 정보 및 사용자가 추가로 입력한 맥락을 로깅
- flex 프론트엔드 프로젝트 내에는 3개의 트래킹 포인트가 있음
- **서버**
  - express 미들웨어 (`requestHandler` / `errorHandler`)
- **클라이언트**
  - axios error interceptor (네트워크 요청 중 오류 발생)
  - React ErrorBoundary (렌더링 로직 내 오류 발생)

“사용자들이 겪는 오류를 트래킹하고
잘 해결하고 싶지만
우리 소스 정보는 최대한 감추고 싶어!”

`build`: Sentry (Release)

- Sentry Release API
  - 소스 코드 및 소스맵에 **특정 버전을 부여**한 뒤 Sentry 서버에 업로드
  - 로깅하는 쪽에서 SDK 초기화시 릴리즈를 명시하면 **에러가 해당 파일과 매칭됨**
- with-sourcemap, sentry-webpack-plugin
  - Release API를 이용해 웹팩 빌드시 소스 코드 및 소스맵에 딱지를 붙이고 업로드
- CI/CD (플렉스팀의 경우 AWS CodeBuild)
  - 실제 브라우저에서는 소스맵 접근이 불가하도록 적절한 처리

`run`

- Next.js
- swr
- openapi-generator

“프로덕션 React 앱 만들려니 챙겨야 할 게 너무 많아!”

서버 사이드 렌더링, 빌드 및 코드 스플리팅, 라우팅, …

`run`: Next.js

- “The React Framework for Production”
- SSR, 라우팅, 메타 태그 설정, 코드 스플리팅 등의 필수적인 기능 제공
- 점점 out-of-box로 지원하는 기능의 수를 늘리고 있음
  - 예시: Next.js 10의 next/image 컴포넌트

`run`: Next.js – SSR

- SSR을 하는 경우, 렌더링은 두 번 일어남
  - 서버에서 한 번, 클라이언트에서 한 번
  - 최초 클라이언트 렌더 시,
    서버에서 그려진 DOM과 React 상태를 연결시키는 작업 (hydration) 이 필요
- 빌드도 두 번 일어남
  - 서버용 번들 한 뭉치, 클라이언트용 번들 한 뭉치
- 빌드 타임과 런타임 모두에 서버/클라이언트를 구분하기 위한 수단이 존재

`run`: Next.js – 설정

```
const release = getRelease();
const shouldUseAssetPrefix = !isLocal && release !== "invalid-release";
const assetPrefix = shouldUseAssetPrefix
  ? `https://${ASSET_PREFIX_CF_CNAME}/${release}`
  : "";

const plugins = isLocal
  ? []
  : [withSourceMaps(), [withTM, { transpileModules }]];

module.exports = withPlugins(plugins, {
  env: {
    isLocal,
  },
  assetPrefix,
  webpack: (config, options) => {
    config.plugins.push(new AntdDayjsWebpackPlugin());

    if (!options.isServer) {
      config.resolve.mainFields = ["browser", "main", "module"];
      config.resolve.alias["@sentry/node"] = "@sentry/browser";
    } else {
      config.plugins.push(
        new ForkTsCheckerWebpackPlugin({
          async: true,
        })
      );
    }

    config.module.rules.push({
      test: /\.(eot|woff|woff2|ttf|svg|png|jpg|gif)$/,
      use: {
        loader: "url-loader",
        options: {
          limit: 100000,
          name: "[name].[ext]",
        },
      },
    });

    return config;
  },
});
```

- env
  - 번들에서 접근 가능한 환경 변수
- assetPrefix
  - 에셋을 CDN 등 다른 장소에서 서빙하는 경우
- webpack
  - Next.js의 기본 웹팩 설정을 커스터마이즈할 때
- 👀 공식 문서

“서버에서 받아오는 온갖 데이터를 잘 관리하기가 너무 어려워!”

전역 캐시, 다시 받아올 때의 UX, 보일러플레이트…

`run`: swr

- 준비물 = 유니크 키 + 데이터 받아오는 함수
- 보일러 플레이트 없이
  리모트 데이터를 사용할 때 필요한 다양한 기능 제공
- stale-while-revalidate
  - 다시 받아오는 동안엔 로딩 상태가 아닌
    예전 데이터를 보여줘서 더 나은 UX를 가능하게 함
- 전역 데이터 캐싱 제공 (Redux 필요한 경우 일부 대체)

`run`: swr

```
import useSWR from 'swr';

function Profile () {
  const {
    data,
    error,
    revalidate
  } = useSWR('/api/user/123', fetcher);

  if (error) {
    return <div>failed to load</div>;
  }

  if (data == null) {
    return <div>loading...</div>;
  }

  return (
    <div>
      hello {data.name}!
      <button onClick={revalidate}>revalidate</button>
    </div>
  );
}
```

- **key**: 특정 데이터의 전역 식별자
  - 어디서 불려도 키가 같으면 같은 데이터를 공유
- **`fetcher`**: 실제로 데이터를 받아오는 함수
  - `data`: `fetcher`의 반환값 (또는 그를 `await`한 값)
  - `error`: 마지막으로 받아오던 중 발생한 오류
- 👀 Redux를 넘어 SWR로 1편 2편

“API 타입은 이미 Swagger에
다 있는데, 매번 손으로 짜야 한다니!”

`run`: openapi-generator

```tsx
const Today = ({ fetchTodaySchedule, daySchedule, extraTimeBlocks }: Props) => {
  const apiClients = useApiClients();

  const [editingDayWorkBlock, setEditingDayWorkBlock] = useState<FlexModel.UserDayWorkBlockDto>(
    daySchedule.dayBlock.activeRevision!,
  );

  const dryRun = useCallback(
    async (changedDayWorkBlock: FlexModel.UserDayWorkBlockDto) => {
      const dayWorkBlockRegisterDto = timeTracking.utils.getDayWorkBlockRegisterFromDayBlock({
        imputedDate: changedDayWorkBlock.date!,
        dayWorkBlock: changedDayWorkBlock,
      });

      const { data: result } = await apiClients.userDayWorkBlock.registerDayWorkBlockUsingPOST({
        ...dayWorkBlockRegisterDto,
        dryRun: true,
      });

      setEditingDayWorkBlock(result.data?.dayWorkBlock);
    },
    [apiClients.userDayWorkBlock, startFetching, finishFetching],
  );

  // ...
};
```

- OpenAPI 스펙 기반에서 코드 생성
- 서버 단에서 정의된 모델 타입, 엔드포인트 URL, 요청/응답 타입 및 호출 코드를 직접 정의할 필요가 사라짐
- 👀 OpenAPI Specification으로 타입-세이프하게 API 개발하기: 희망편 VS 절망편

What’s next?

- Rome Toolchain
- Webpack 5 (module federation)
- GraphQL, Relay

What’s next?

- Rome Toolchain
- Webpack 5 (module federation)
- GraphQL, Relay

분량 조절 실패로 다음 기회에...