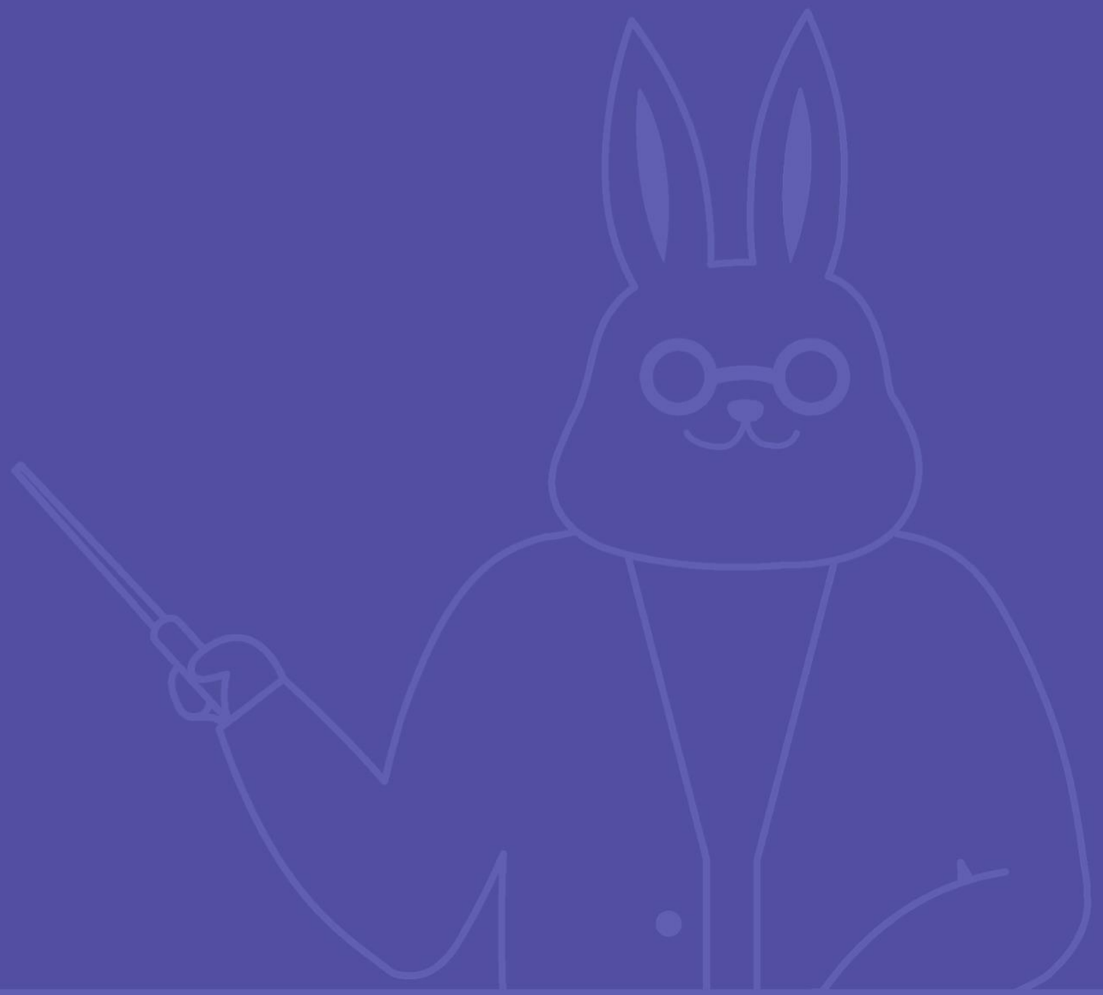
React 심화 I

03 비동기 통신과 Promise

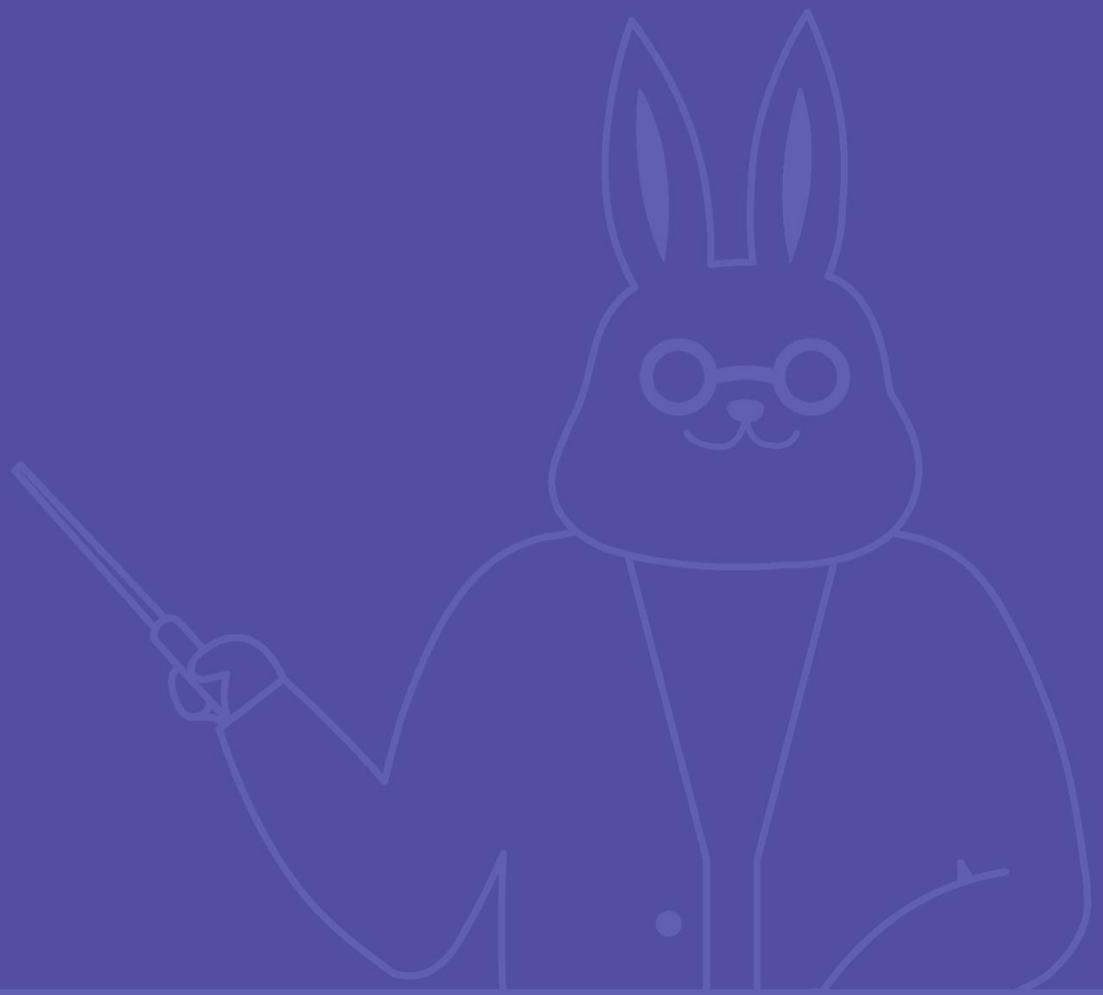


/* elice */

목차

01

자바스크립트 비동기 1



자바스크립트 비동기의 등장

- 초기 웹 환경에서는, 서버에서 모든 데이터를 로드하여 페이지를 빌드했으므로 자바스크립트에는 별도의 비동기 처리가 필요하지 않음.
- Ajax(Asynchronous JavaScript and XML) 기술의 등장으로 페이지 로드 없이 client-side에서 서버로 요청을 보내 데이터를 처리할 수 있게 됨.
- XMLHttpRequest라는 객체를 이용해 서버로 요청을 보낼 수 있게 됨.

자바스크립트와 비동기

- 자바스크립트는 single-threaded language이므로, 만일 서버 요청을 기다려야 한다면 유저는 멈춰있는 브라우저를 보게 될 것.
- 따라서 동기가 아닌 비동기 처리를 이용해 서버로 통신할 필요가 있음.
- 비동기 요청 후, main thread는 유저의 입력을 받거나, 페이지를 그리는 등의 작업을 처리.
- 비동기 응답을 받으면, 응답을 처리하는 callback 함수를 task queue에 넣음.
- event queue는 main thread에 여유가 있을 때 task queue에서 함수를 꺼내 실행.

동기 vs 비동기

- **동기(synchronous)** 코드는, 해당 코드 블록을 실행할 때 thread의 제어권을 넘기지 않고 순서대로 실행하는 것을 의미함.

동기 vs 비동기

Code

```
console.log("This is synchronous...")

for (let i = 0; i < 1000000000; ++i) {
  console.log("I am blocking the main thread...")
}

console.log("This is synchronous...DONE!")
```

동기 vs 비동기

- **비동기(asynchronous)** 코드는, 코드의 순서와 다르게 실행됨.
- 비동기 처리 코드를 감싼 블록은 task queue에 넣어짐.
- main thread가 동기 코드를 실행한 후에 제어권이 돌아왔을 때 event loop가 task queue에 넣어진 비동기 코드를 실행함.

동기 vs 비동기

Code

```
setTimeout(() => console.log("This is asynchronous..."), 5000)

console.log("This is synchronous...")

for (let i = 0; i < 1000000000; ++i) {
  console.log("I am blocking the main thread...")
}
```

비동기 처리 예시

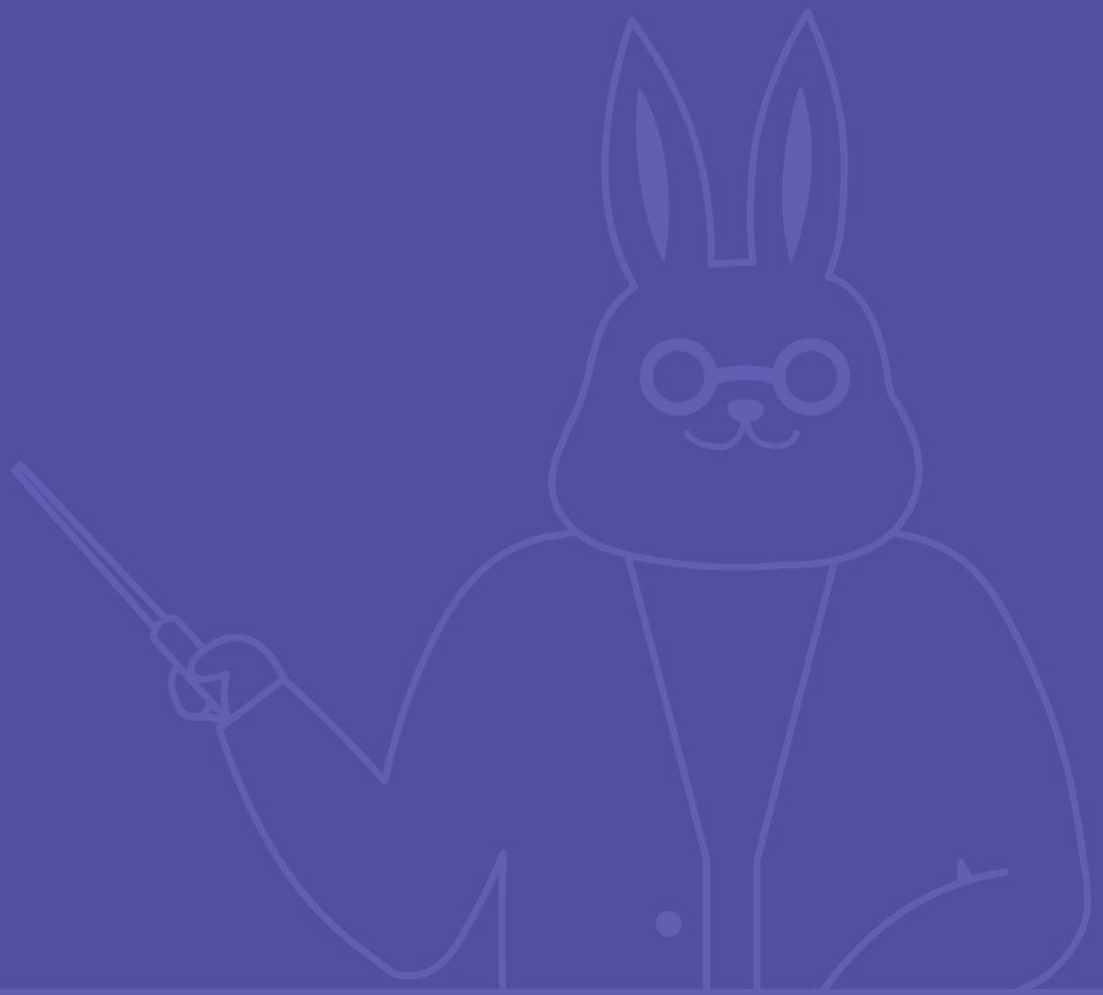
Code

```
request("user-data", (userData) => {
  console.log("userData 로드")
  saveUsers(userData)
});

console.log("DOM 변경")
console.log("유저 입력")
```

02

자바스크립트 비동기 2



비동기 처리를 위한 내부 구조

- 브라우저에서 실행되는 자바스크립트 코드는 event driven 시스템으로 작동.
- 웹앱을 로드하면 브라우저는 HTML document를 읽어 문서에 있는 CSS code, JS code를 불러옴.
- 자바스크립트 엔진은 코드를 읽어 실행.

비동기 처리를 위한 내부 구조

- 브라우저의 main thread는 자바스크립트 코드에서 동기적으로 처리되어야 할 코드 실행 외에도, 웹 페이지를 실시간으로 렌더링하고, 유저의 입력을 감지하고, 네트워크 통신을 처리하는 등 수많은 일을 처리.
- 비동기 작업을 할당하면, 비동기 처리가 끝나고 브라우저는 task queue에 실행 코드를 넣음.
- main thread는 event loop를 돌려, task queue에 작업이 있는지 체크.
- 작업이 있으면 task를 실행.

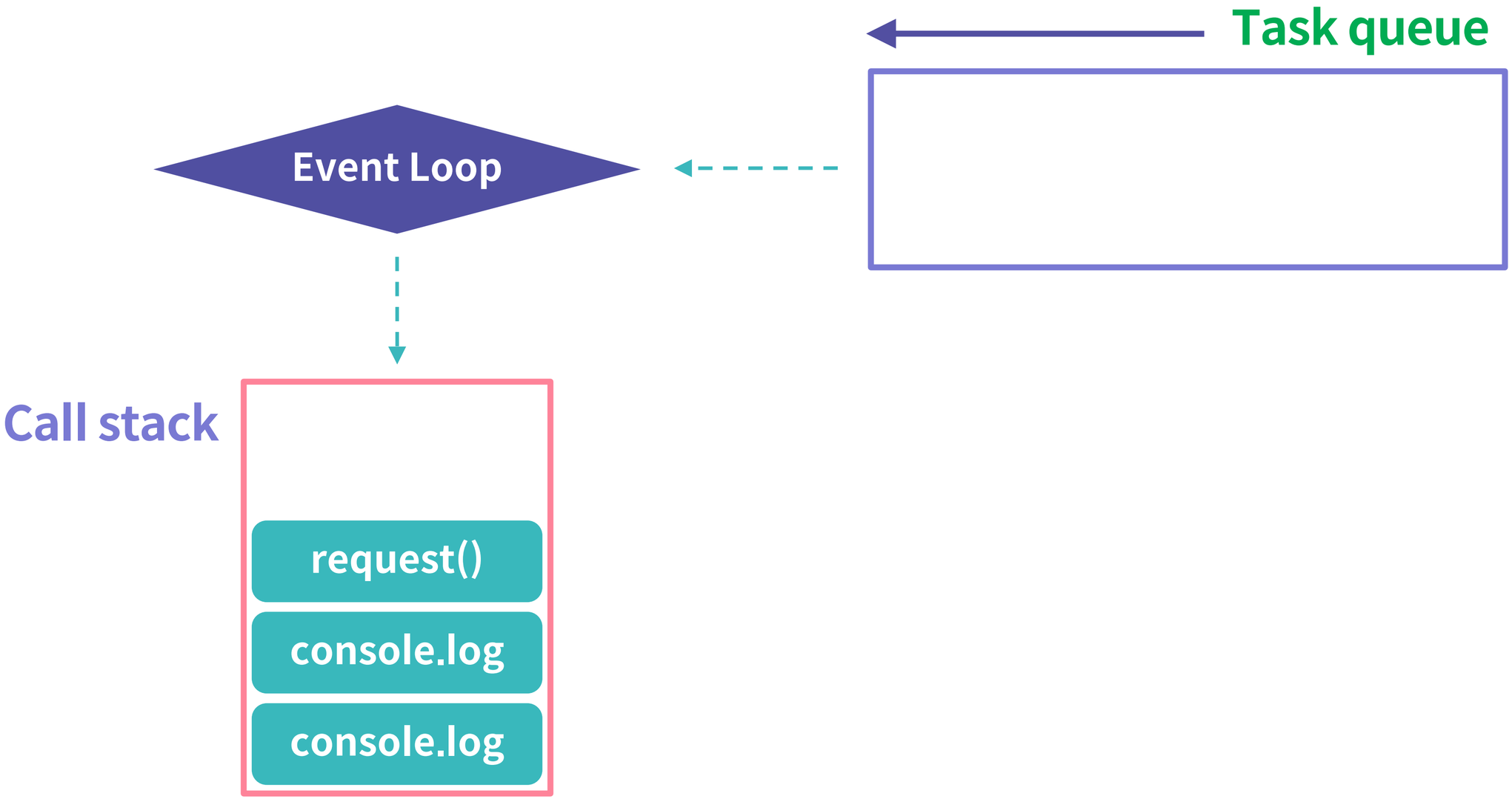
비동기 처리 예시

Code

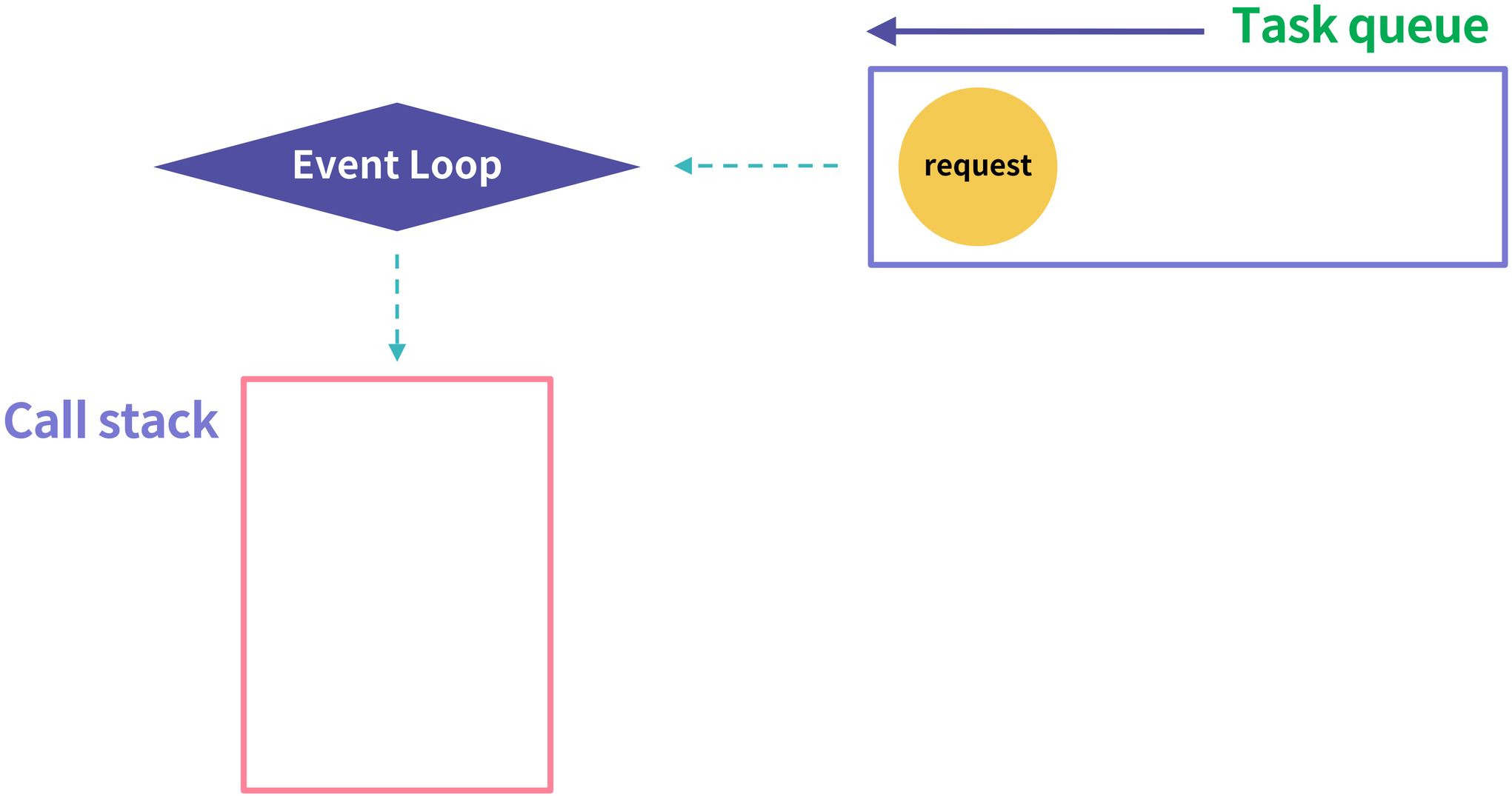
```
request("user-data", (userData) => {
  console.log("userData 로드")
  saveUsers(userData)
});

console.log("DOM 변경")
console.log("유저 입력")
```

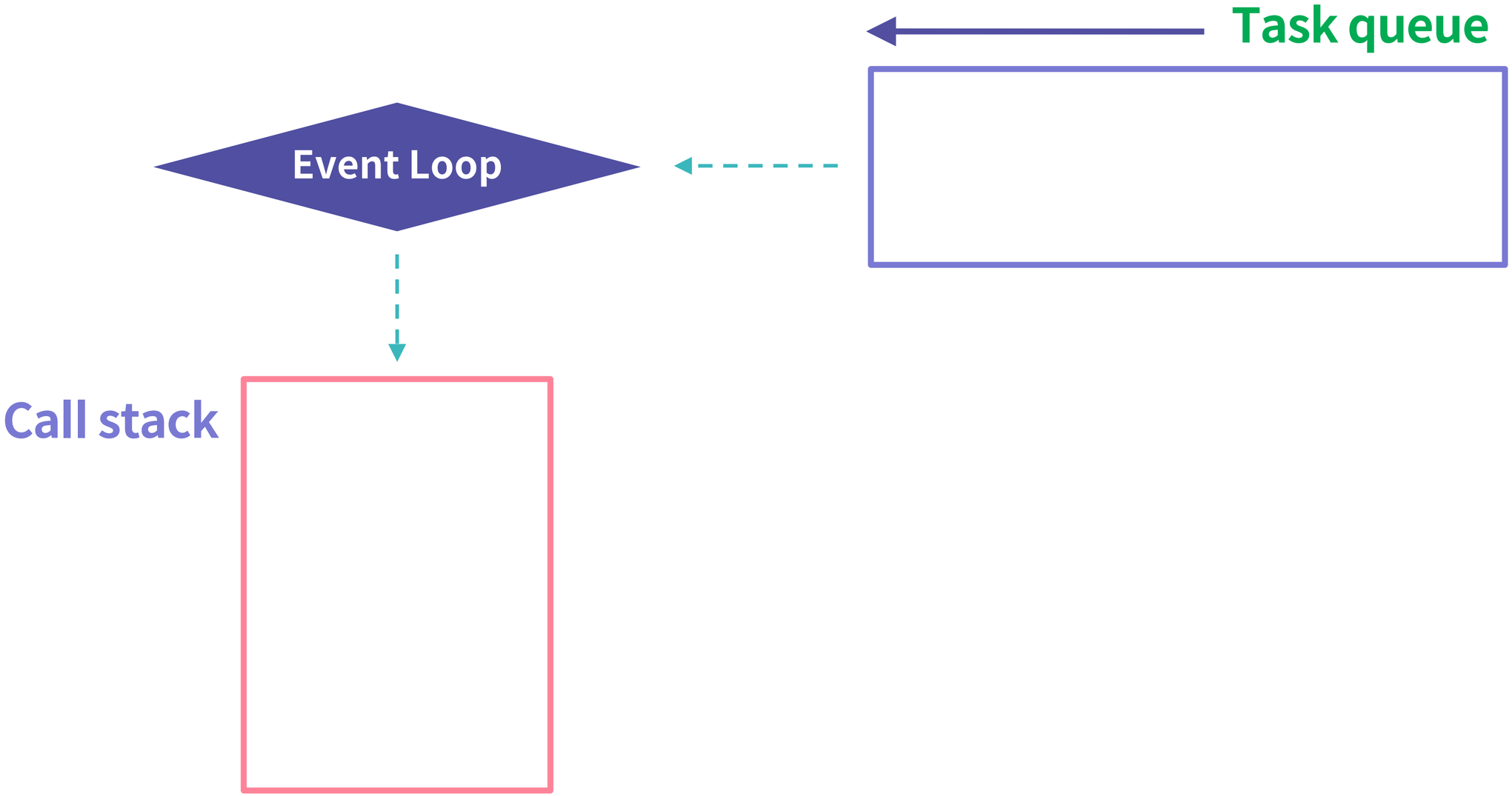
비동기 처리를 위한 내부 구조 - 다이어그램



비동기 처리를 위한 내부 구조 - 다이어그램

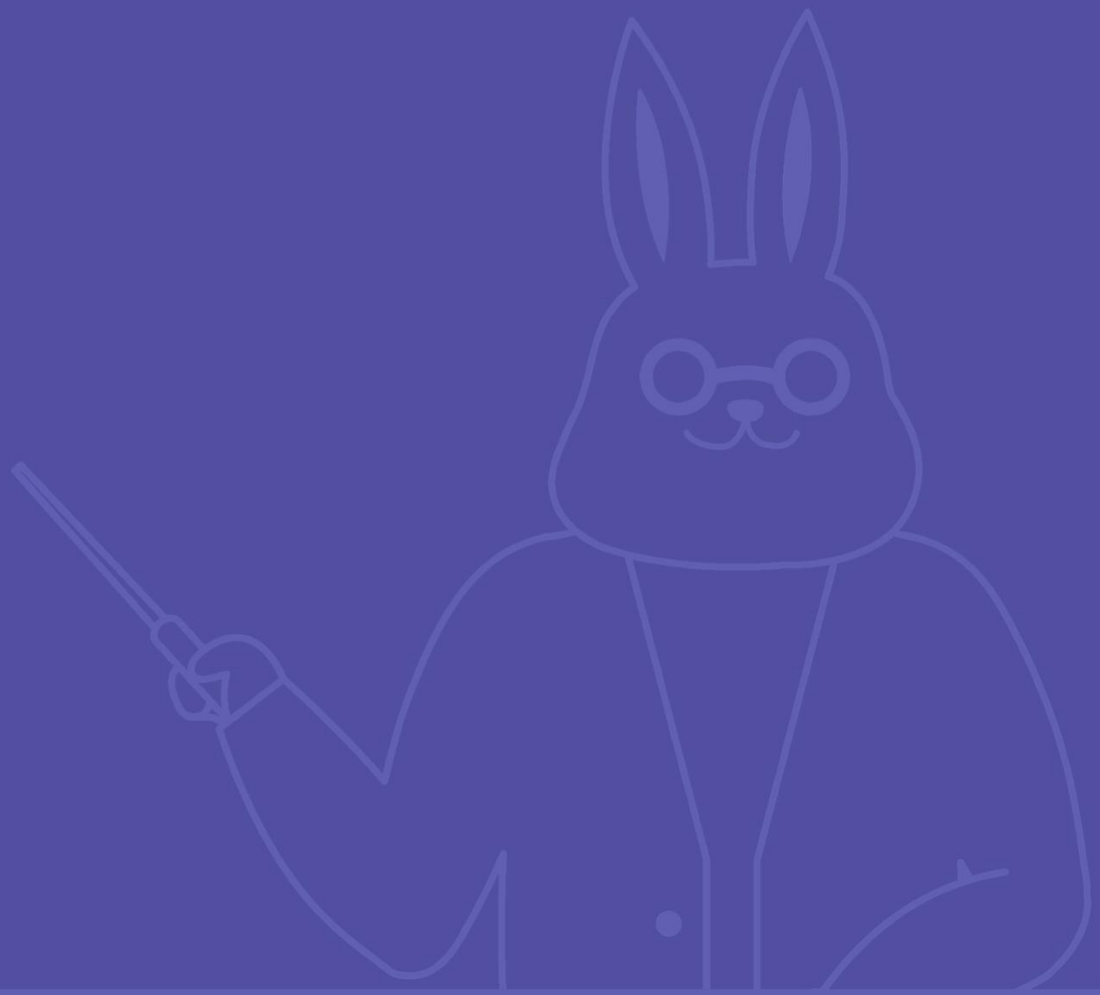


비동기 처리를 위한 내부 구조 - 다이어그램



03

Promise



Callback pattern vs Promise

- 비동기 처리 후 실행될 코드를 Callback function으로 보내는 것.
- 비동기 처리가 고도화되면서, Callback hell 등이 단점으로 부각됨.
- Promise를 활용하여 비동기 처리의 순서 조작, 에러 핸들링,
  여러 비동기 요청 처리 등을 쉽게 처리할 수 있게 됨.

Callback pattern - Single request

fetchUserAddress

```
function fetchUsers(onSuccess) {
  request('/users', onSuccess)
}
```

Promise - Single request

fetchUserAddress

```
function fetchUsers(onSuccess) {
  return request('/users').then(onSuccess)
}
```

Callback pattern - Error handling

fetchUserAddress

```
function fetchUsers(onSuccess, onError) {
  return request('/users')
          .then(onSuccess)
          .catch(onError)

  // or
  return request('/users').then(onSuccess, onError)
}
```

Promise - Single request

fetchUserAddress

```
function fetchUsers() {
  return request('/users')
}
```

Callback pattern - Multiple request

fetchUserAddress

```
function fetchUserAddress(onSuccess) {
  request('/users', (userData) => {
    const userDataWithAddress = []
    const userLength = userData.length

    userData.map(user => request(`/users/${user.userId}/address`, (address) => {
      const userDataWithAddress.push({ ...user, address })
      if (userDataWithAddress.length === userLength) {
        onSuccess(userDataWithAddress)
      }
    })
  })
}
```

Promise - Multiple request

fetchUserAddress

```
function fetchUserAddress() {
  return request("/users").then((userData) =>
    Promise.all(
      userData.map((user) =>
        request(`/users/{user.userId}/address`).then((address) => ({
          ...user,
          address,
        }))
      )
    )
  );
}
```

Promise

- Promise 객체는, 객체가 생성 당시에는 알려지지 않은 데이터에 대한 Proxy.
- 비동기 실행이 완료된 후에, `.then`, `.catch`, `.finally` 등의 핸들러를 붙여 각각 데이터 처리 로직, 에러 처리 로직, 클린업 로직을 실행.
- then 체인을 붙여, 비동기 실행을 마치 동기 실행처럼 동작하도록 함.
- 전체 스펙은 https://promisesaplus.com/ 참조

Promise

Code

```
function returnPromise() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const randomNumber = generateRandomNumber(100)
      if (randomNumber < 50) resolve(randomNumber)
      else reject(new Error("Random number is too small."))
    }, 1000)
  })
}
```

Promise

Code

```
returnPromise()
  .then(num => {
    console.log("First random number : ", num)
  })
  .catch(error => {
    console.error("Error occured : ", error)
  })
  .finally(() => {
    console.log("Promise returned.")
  })
```

Promise

- Promise 객체는 pending, fulfilled, rejected 3개의 상태를 가짐.
- fulfilled, rejected 두 상태를 settled 라고 지칭.
- pending은 비동기 실행이 끝나기를 기다리는 상태.
- fulfilled는 비동기 실행이 성공한 상태.
- rejected는 비동기 실행이 실패한 상태.

Promise

- then, catch는 비동기(Promise), 동기 실행 중 어떤 것이라도 리턴할 수 있음.

Promise

Code

```
function returnPromise() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const randomNumber = generateRandomNumber(100)
      if (randomNumber < 50) resolve(randomNumber)
      else reject(new Error("Random number is too small."))
    }, 1000)
  })
}
```

Promise

Code

```
returnPromise()
  .then(num => {
    console.log("First random number : ", num)
  })
  .catch(error => {
    console.error("Error occured : ", error)
  })
  .finally(() => {
    console.log("Promise returned.")
  })
```

Multiple Promise handling

- Promise.all()은 모든 프로미스가 fulfilled 되길 기다림.
  하나라도 에러 발생시, 모든 프로미스 요청이 중단됨.
- Promise.allSettled()는 모든 프로미스가 settled 되길 기다림.
- Promise.race()는, 넘겨진 프로미스들 중 하나라도 settled 되길 기다림.
- Promise.any() - 넘겨진 프로미스 중 하나라도 fulfilled 되길 기다림.

Promise.all

Code

```
Promise.all(
  users.map(user => request('/users/detail', user.name))
  // [Promise, Promise, ... ,Promise ]
)
  .then(console.log) // [UserNameData, UserNameData, ..., UserNameData]
  .catch(e => console.error("하나라도 실패했습니다.")
```

Promise.allSettled

Code

```
function saveLogRetry(logs, retryNum) {
  if (retryNum >= 3) return; // no more try.

  Promise.allSettled(logs.map(saveLog))
    .then((results) => {
      return results.filter((result) => result.status === "rejected");
    })
    .then((failedPromises) => {
      saveLogRetry(
        failedPromises.map((promise) => promise.reason.failedLog),
        retryNum + 1
      );
    });
}
```

Promise.race

Code

```
function requestWithTimeout(request, timeout = 1000) {
  return Promise.race([request, wait(timeout)]).then((data) => {
    console.log("요청 성공.");
    return data;
  });
}

requestWithTimeout(req)
  .then(data => console.log("data : ", data))
  .catch(() => console.log("타임아웃 에러!"))
```

Promise.any

Code

```
function getAnyData(dataList) {
  Promise.any(dataList.map((data) => request(data.url)))
    .then((data) => {
      console.log("가장 첫 번째로 가져온 데이터 : ", data);
    })
    .catch((e) => {
      console.log("아무것도 가져오지 못했습니다.");
    });
}
```

Promise chaining, nested promise

- Promise객체는, settled되더라도 계속 핸들러를 붙일 수 있음.
- 핸들러를 붙인 순서대로 호출됨.
- .catch 뒤에 계속 핸들러가 붙어있다면, 에러를 처리한 후에 계속 진행됨.
  이때는 catch에서 리턴한 값이 then으로 전달됨.

Promise.any

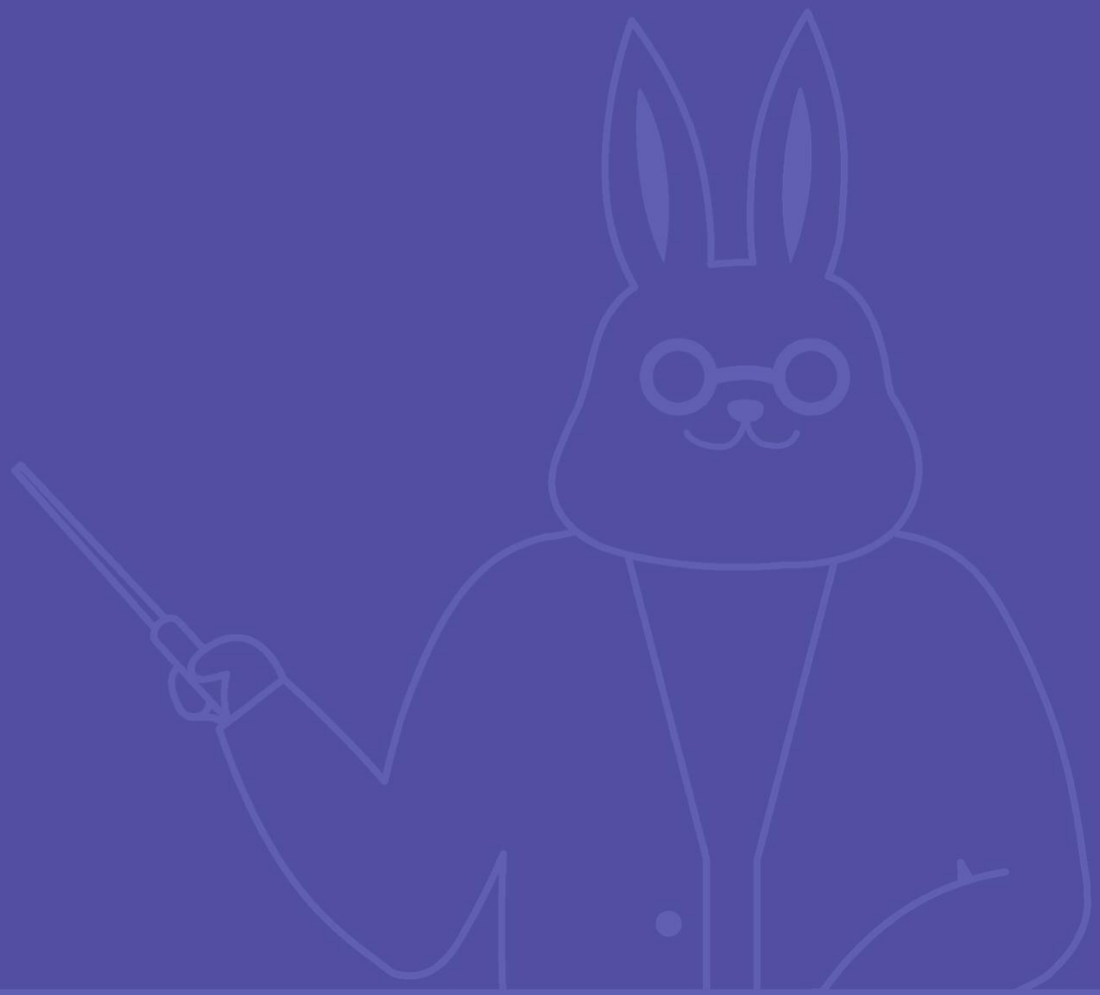
Code

```
Promise.resolve()
  .then(() => wait2(500).then(() => console.log("500 waited.")))
  .then(() => {
    console.log("After 500 wait.")
    return wait2(1000).then(() => console.log("1000 waited."))
  })
  .then(() => console.log("DONE"))

/*
500 waited.
After 500 wait.
1000 waited.
DONE
*/
```

04

async/await



async/await

- Promise 체인을 구축하지 않고도, Promise를 직관적으로 사용할 수 있는 문법.
- 많은 프로그래밍 언어에 있는 try ... catch 문으로 에러를 직관적으로 처리.
- async function을 만들고, Promise를 기다려야 하는 표현 앞에 await을 붙임.

Promise vs async/await

async/await

```
async function fetchUsers() {
  try {
    const users = await request('/users')
    console.log("users fetched.")
    return users
  } catch (e) {
    console.log("error : ", e)
  }
}
```

Promise

```
function fetchUsers() {
  return request('/users')
    .then(users => console.log("users fetched."))
    .catch(e => console.error("error : ", e))
}
```

async/await - 여러 개의 await

- 여러 개의 await을 순서대로 나열하여, then chain을 구현할 수 있음.
- try... catch 문을 자유롭게 활용하여 에러 처리를 적용.

async/await - 여러 개의 await

Code

```
async function fetchUserWithAddress(id) {
  try {
    const user = await request(`/user/${user.id}`)
    const address = await request(`/user/${user.id}/address`)
    return { ...user, address }
  } catch (e) {
    console.log("error : ", e)
  }
}
```

async/await - 여러 개의 await

Code

```
async function fetchUserWithAddress(id) {
  let user = null
  try {
    user = await request(`/user/${user.id}`)
  } catch (e) {
    console.log("User fetch error: ", e)
    return
  }
  try {
    const address = await
request(`/user/${user.id}/address`)
    return { ...user, address }
  } catch (e) {
    console.log("Address fetch error: ", e)
  }
}
```

async/await - 여러 개의 await

Code

```
async function fetchUserWithAddress(id) {
  try {
    const user = await request(`/user/${user.id}`)
    if (!user) throw new Error("No user found.")

    const address = await request(`/user/${user.id}/address`)
    if (!address.userId !== user.id) throw new Error("No address match with user.")

    return { ...user, address }
  } catch (e) {
    console.log("User fetch error: ", e)
  }
}
```

async/await - 여러 개의 await

Code

```
async function fetchUserWithAddress(id) {
  try {
    const user = await request(`/user/${user.id}`)
    const address = await request(`/user/${user.id}/address`)
    return { ...user, address }
  } catch (e) {
    try {
      sendErrorLog(e)
    } catch (e) {
      console.error("에러를 로깅하는데 실패하였습니다.")
    }
  }
}
```

async/await - Promise 와의 조합

- Promise.all은, 특정 비동기 작업이 상대적으로 빠르게 끝나도,
  느린 처리를 끝까지 기다려야만 함.
- 이와 달리, async/await을 활용할 경우 parallelism을 구현할 수 있음.
  즉, 끝난 대로 먼저 처리될 수 있음.

async/await - Promise 와의 조합

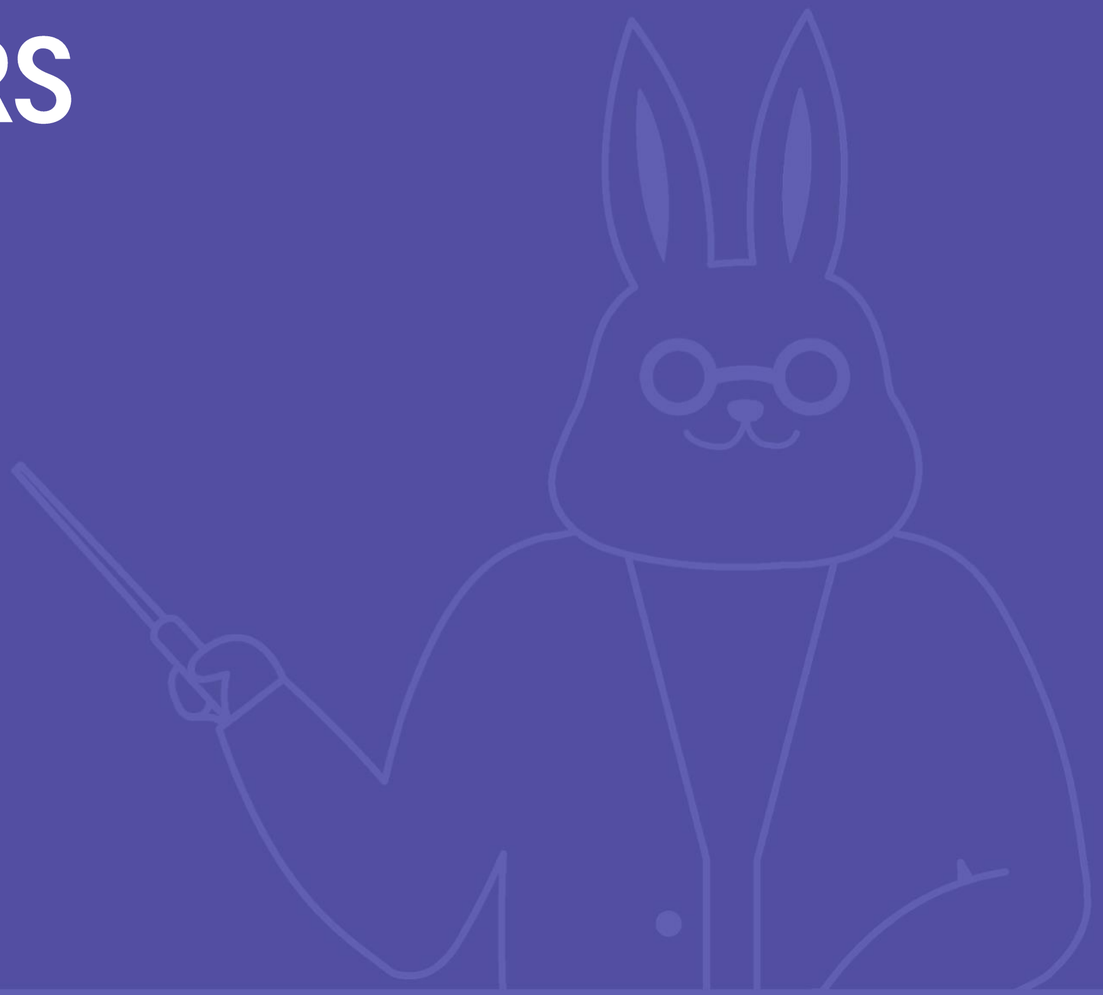
Code

```
async function fetchUserWithAddress(id) {
  return await Promise.all([
    (async () => await request(`/users/${id}`))(),
    (async () => await request(`/users/${id}/address`))(),
  ]);
}

fetchUserWithAddress('1234')
  .then(([user, address]) => ({ ...user, address }))
  .catch(e => console.log("Error : ", e))
```

05

POSTMAN, OpenAPI, CORS



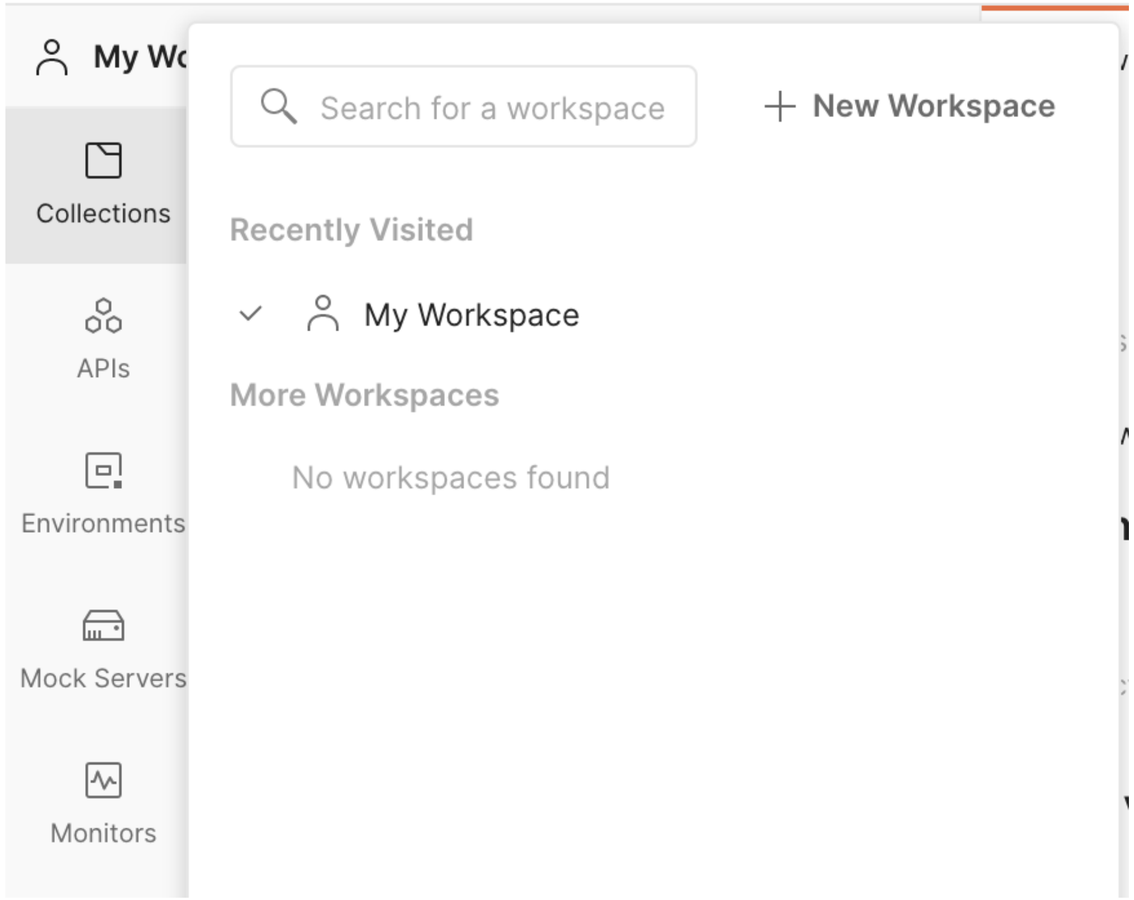
POSTMAN 소개

- 서버와의 통신을 위해 API를 활용하는 경우, React 앱으로만 요청하여 API가 잘 동작하는지 알아보는 건 비효율적.
- 수많은 API의 endpoint와 실행 조건 등을 관리하는 것도 힘듦.
- POSTMAN은 API를 테스트하기 위한 개발 도구.

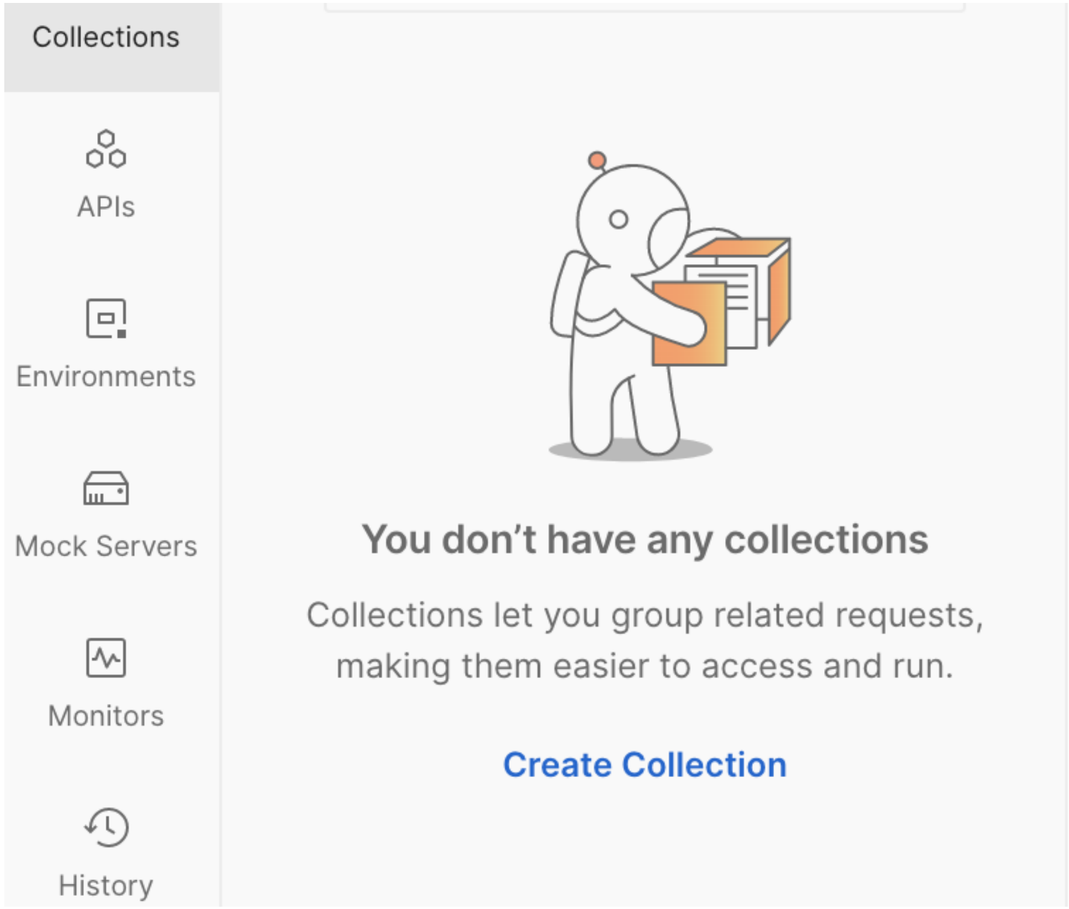
POSTMAN 소개

- Auth, header, payload, query 등 API 요청에 필요한 데이터를 쉽게 세팅.
- 다른 개발자가 쉽게 셋업해 테스트할 수 있도록 API 정보를 공유할 수 있음.
- Request를 모아 Collection으로 만들어, API를 종류별로 관리.
- 환경 변수를 정의하여, 환경별로 테스트 가능.
- https://www.postman.com/ 에서 다운로드.

POSTMAN 사용

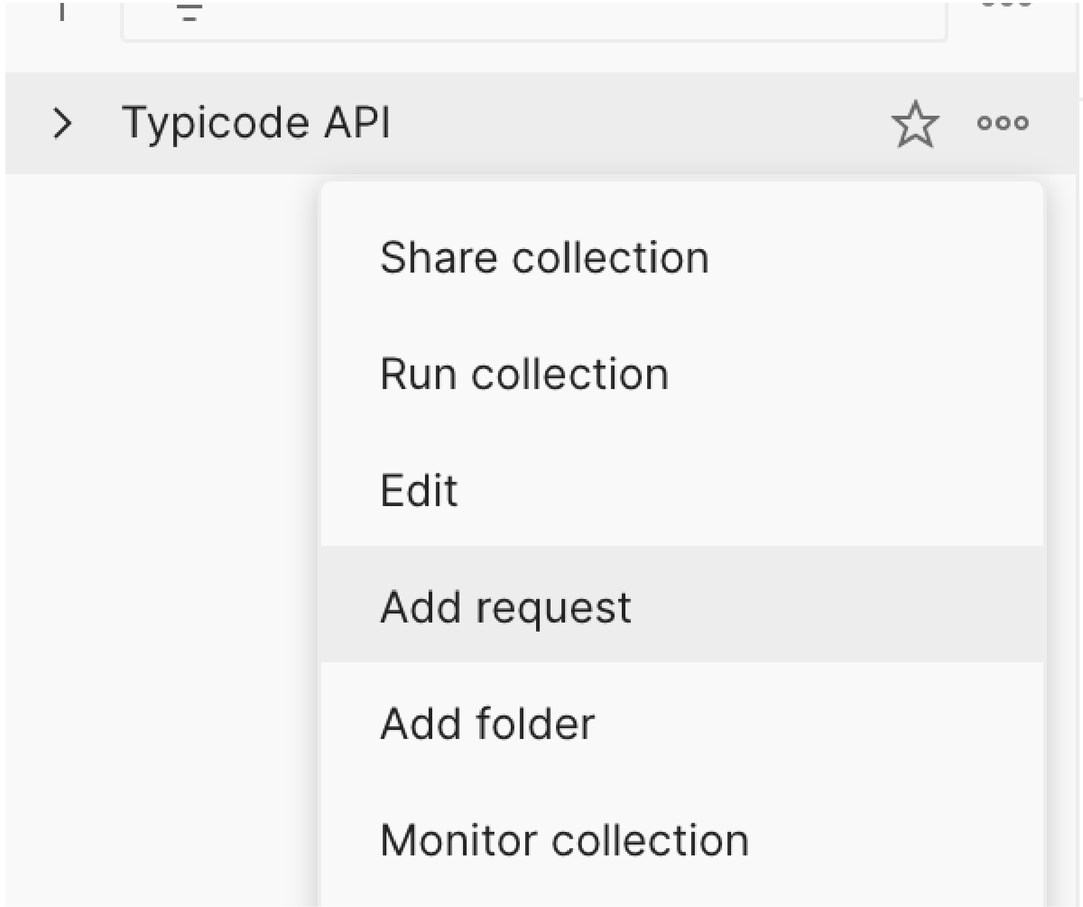


Workspace 설정

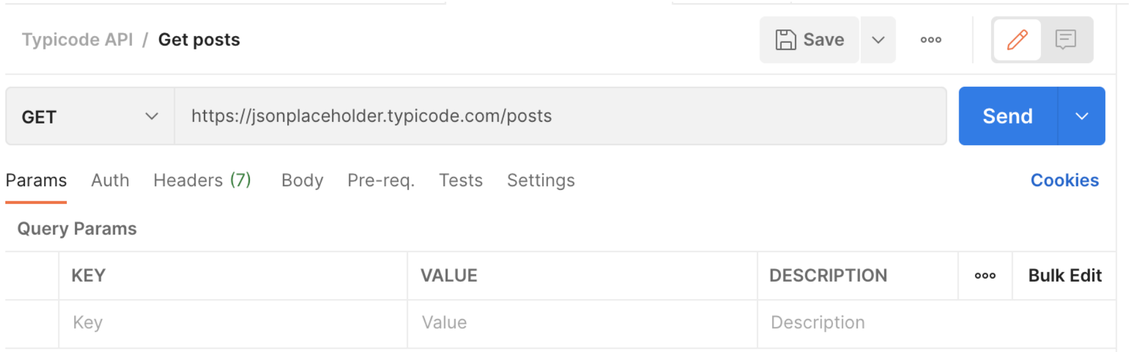


Collection 설정

POSTMAN 사용



Request 생성

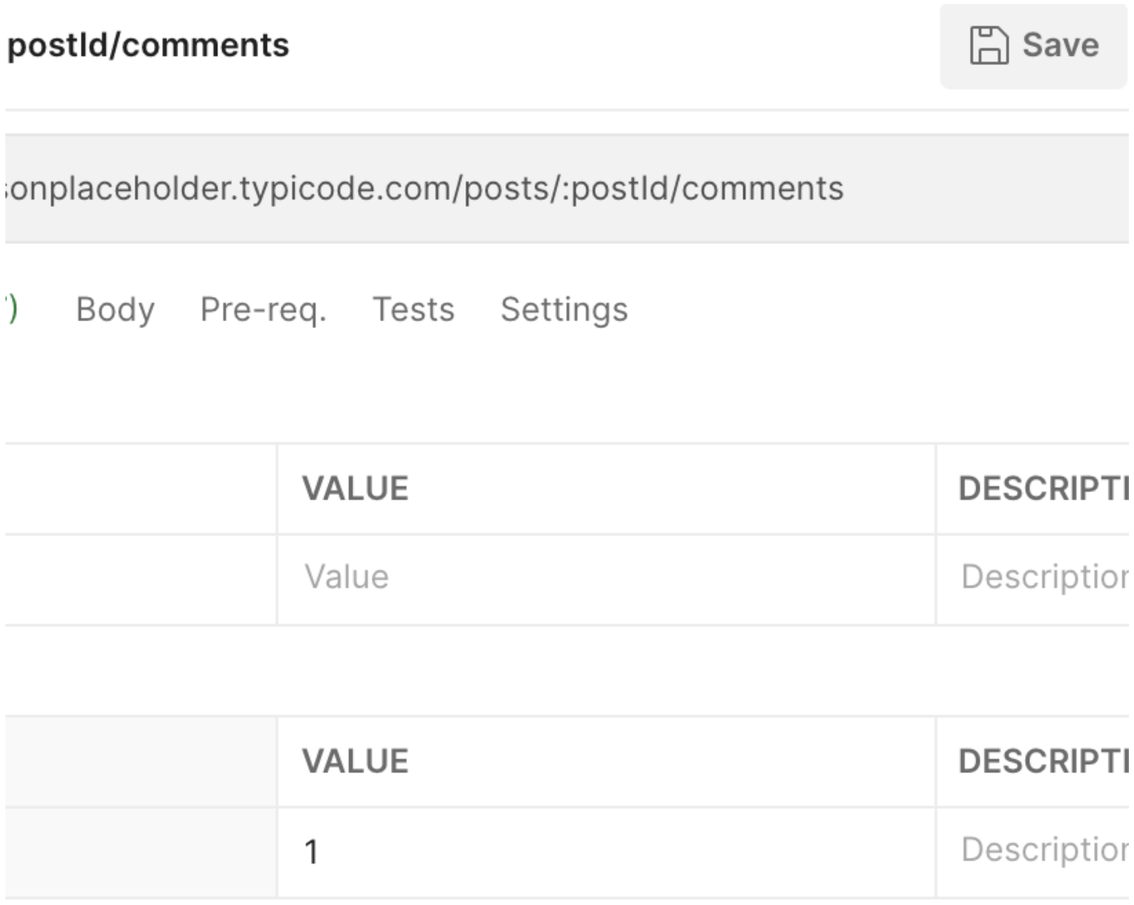


Collection 설정

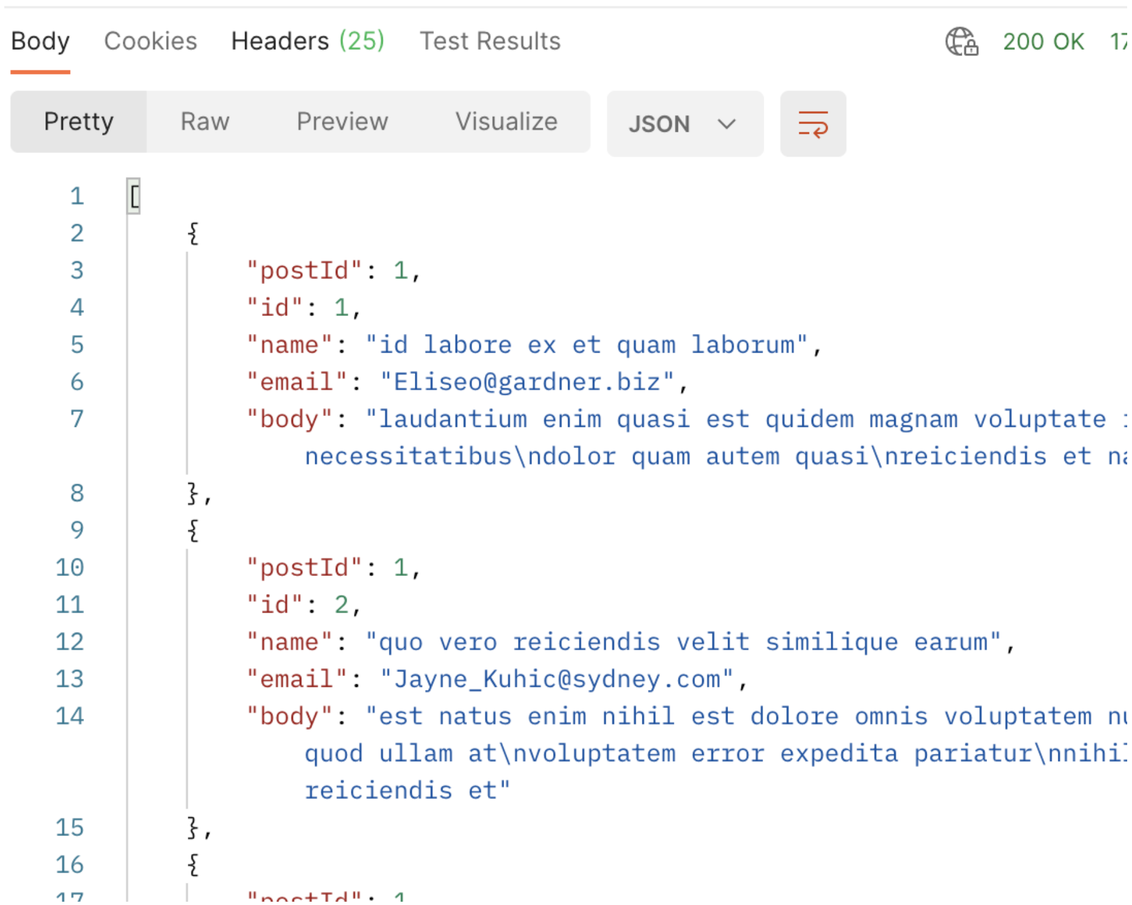
POSTMAN 사용

Payload 확인

POSTMAN 사용

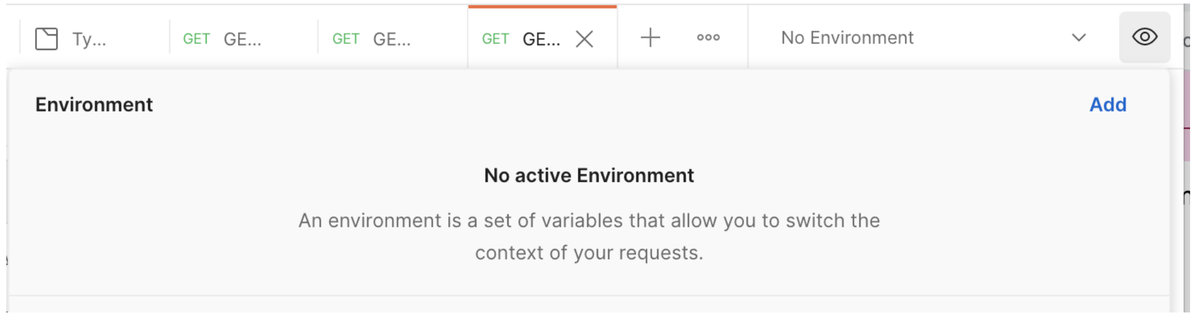


Path variable

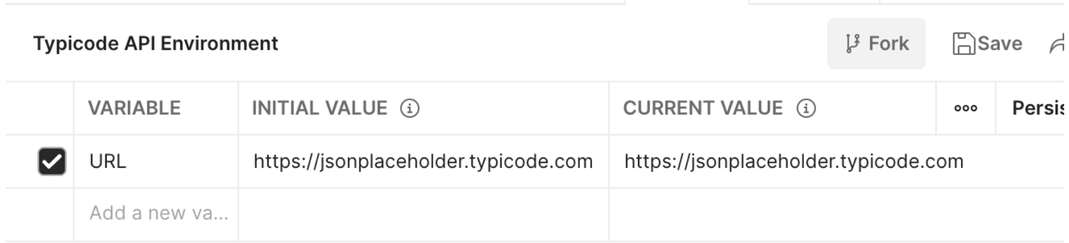


postId = 1인 데이터

POSTMAN 사용 - Environment 세팅

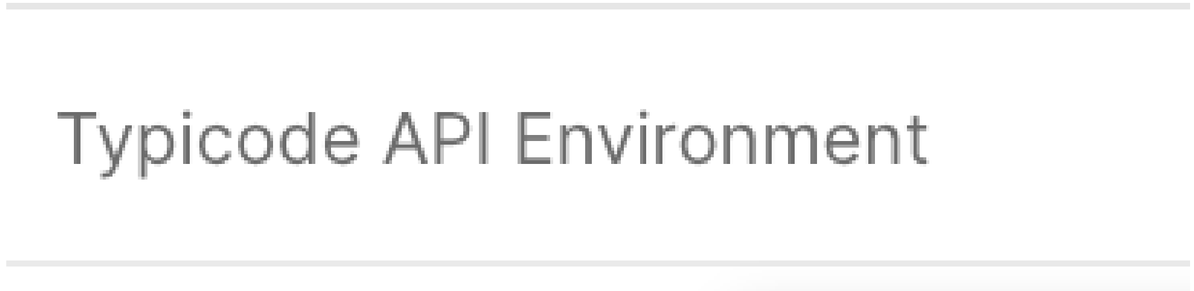


Environment 생성

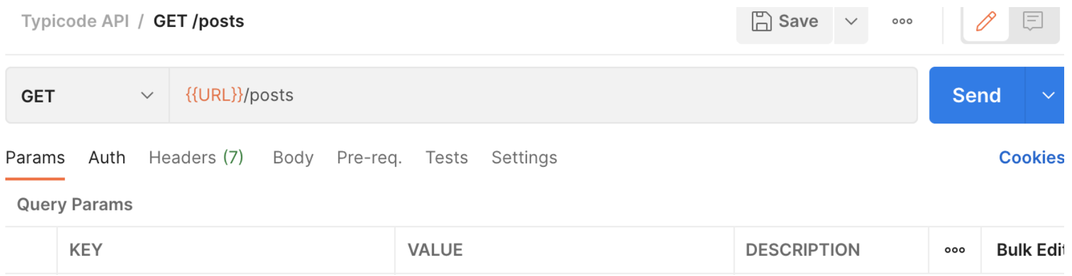


URL 변수 추가

POSTMAN 사용 - Environment 세팅

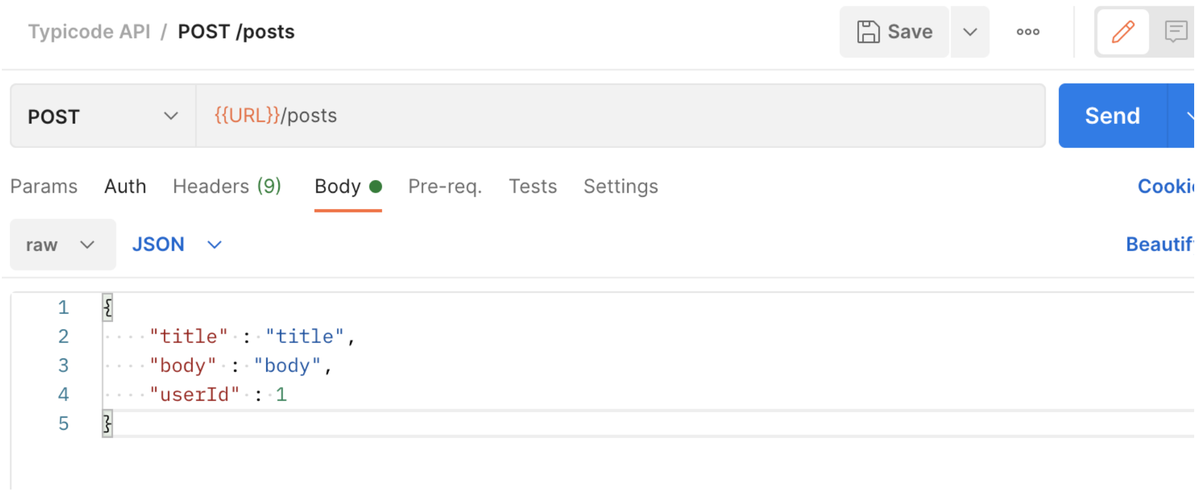


Environment 설정



URL 변수 적용

POSTMAN 사용



Post 요청 시 body 세팅

Open API

- RESTful API를 하나의 문서로 정의하기 위한 문서 표준.
- OpenAPI Specification(OAS)으로 정의됨.
- Swagger 등의 툴로, Open API로 작성된 문서를 파싱해 테스팅 도구로 만들 수 있음.
- 프론트엔드 개발자, 백엔드 개발자와의 협업 시 주요한 도구로 사용.

Open API

- API의 method, endpoint 를 정의.
- endpoint마다 인증 방식, content type 등 정의.
- body data, query string, path variable 등 정의.
- 요청, 응답 데이터 형식과 타입 정의 - data model 활용(schema).

CORS

- Cross-Origin Resource Sharing.
- 브라우저는 모든 요청 시 Origin 헤더를 포함.
- 서버는 Origin 헤더를 보고, 해당 요청이 원하는 도메인에서부터 출발한 것인지를 판단.
- 다른 Origin에서 온 요청은 서버에서 기본적으로 거부함.

CORS

- 그러나, 보통 서버의 endpoint와 홈페이지 domain은 다른 경우가 많음.
- 따라서 서버에서는 홈페이지 domain을 허용하여, 다른 domain이라 하더라도 요청을 보낼 수 있게 함.
- 서버는 Access-Control-Allow-Origin 외에 Access-Control-* 을 포함하는 헤더에 CORS 관련 정보를 클라이언트로 보냄.

CORS

- 웹사이트에 악성 script가 로드되어, 수상한 요청을 하는 것을 막기 위함.
- 반대로, 익명 유저로부터의 DDos공격 등을 막기 위함.
- 서버에 직접 CORS 설정을 할 수 없다면, Proxy 서버 등을 만들어 해결.

크레딧

/* elice */

코스 매니저
이재성

콘텐츠 제작자
김일식

강사
김일식

감수자
-

디자이너
강혜정

연락처

TEL
070-4633-2015

WEB
https://elice.io

E-MAIL
contact@elice.io