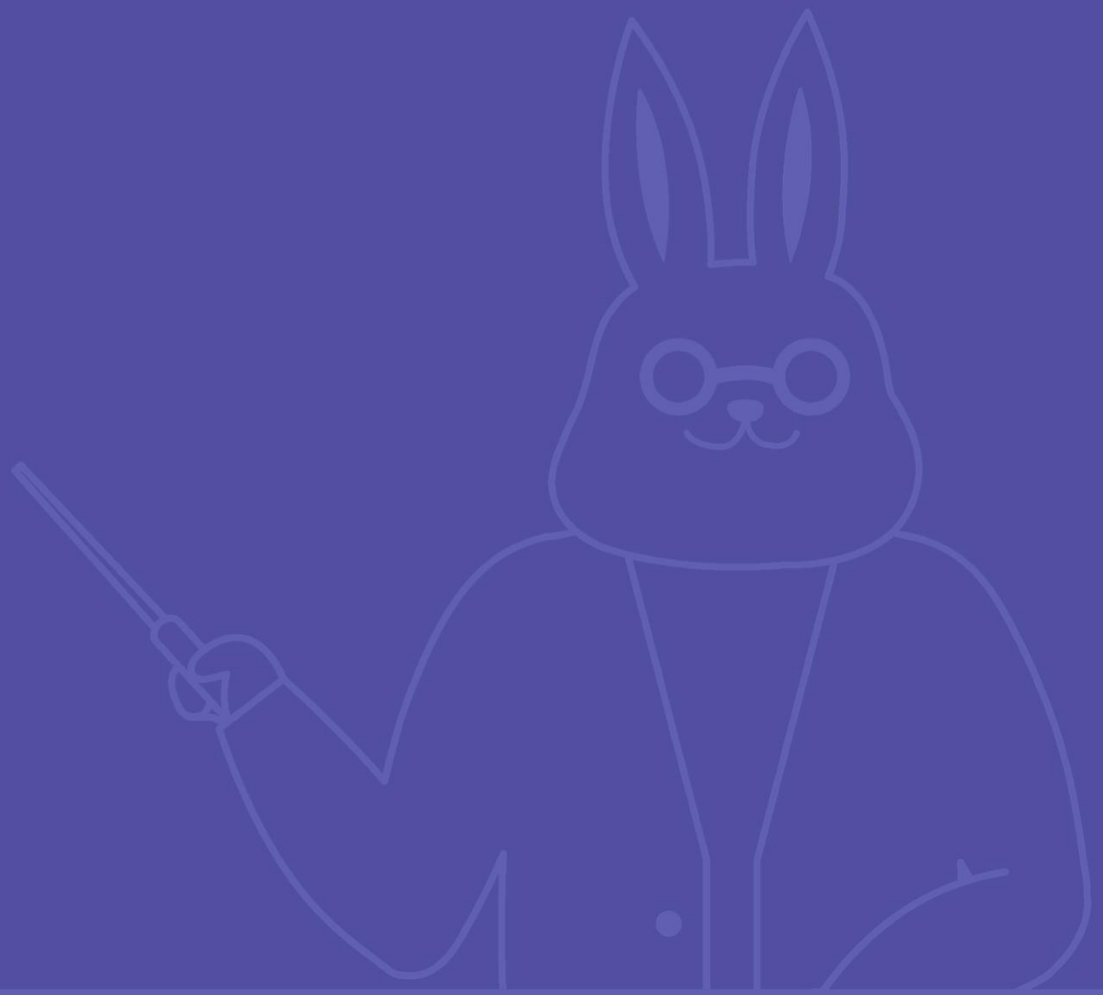
자바스크립트 심화

03 비동기



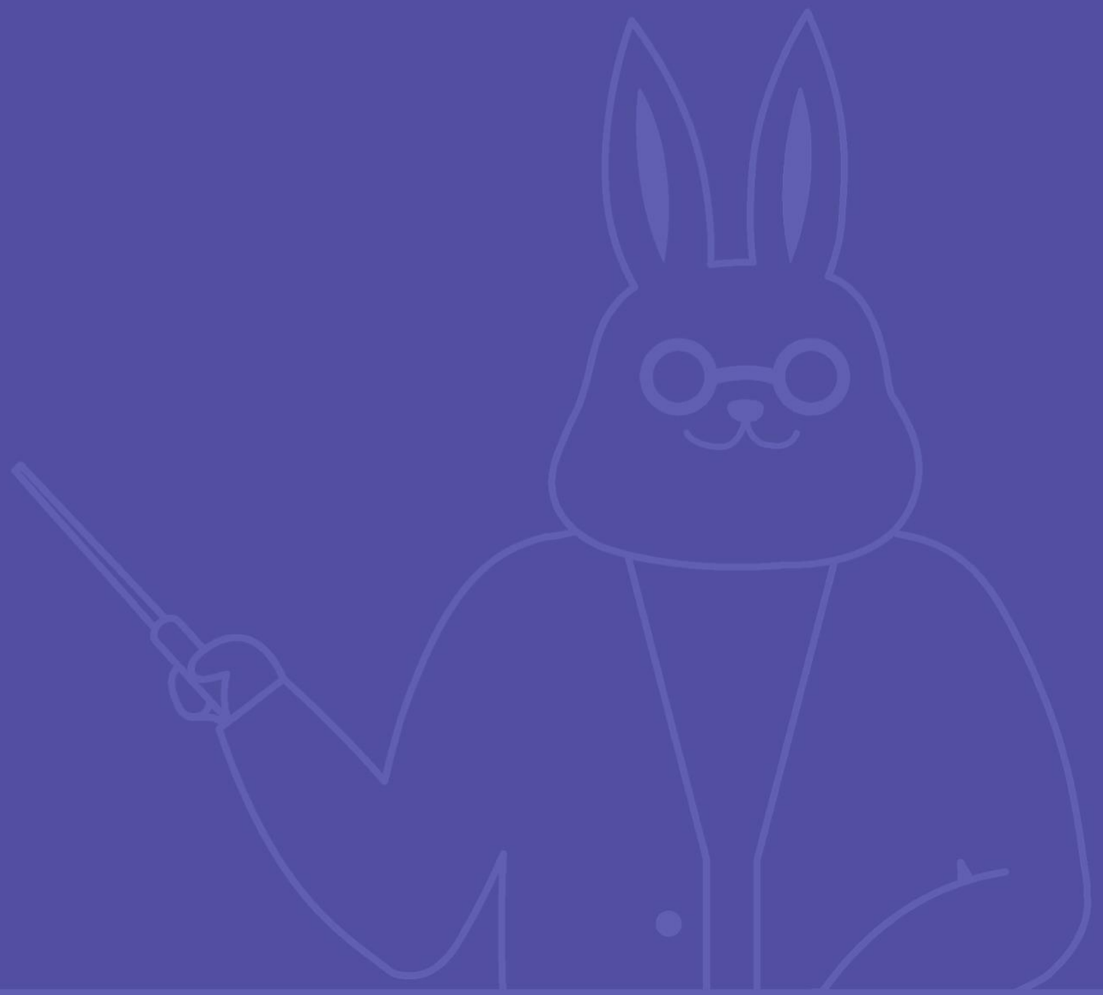
Copyright Elice. All Rights Reserved

/* elice */

목차

01

자바스크립트 제어 흐름



Copyright Elice. All Rights Reserved

자바스크립트 비동기 이해하기

- 자바스크립트는 다른 멀티스레드 프로그래밍 언어와 다른 방식으로 비동기 동작을 처리한다.
- 처음 자바스크립트를 접하는 경우, 동작에 대한 정확한 이해가 없으면 코드의 흐름을 따라가기 어렵다.
- 자바스크립트 내부의 비동기 동작을 이해하기 위해서는 이벤트 루프 등의 개념을 알아야만 한다.

자바스크립트 엔진

- 자바스크립트 엔진은 하나의 메인 스레드로 구성된다.
- 메인 스레드는 코드를 읽어 한 줄씩 실행한다.
- 브라우저 환경에서는 유저 이벤트를 처리하고 화면을 그린다.

동기적 제어 흐름

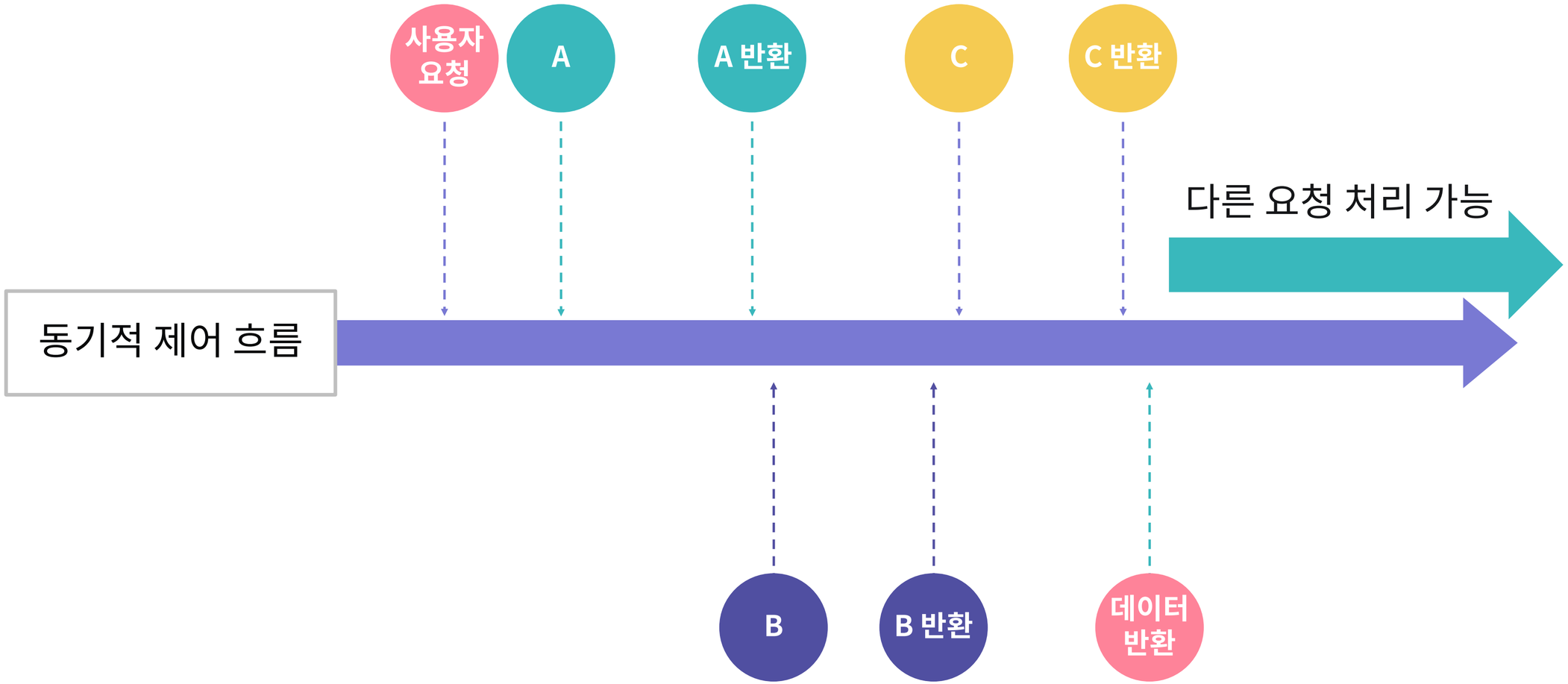
- 동기적 제어 흐름은 현재 실행 중인 코드가 종료되기 전까지 다음 줄의 코드를 실행하지 않는 것을 의미한다.
- 분기문, 반복문, 함수 호출 등이 동기적으로 실행된다.
- 코드의 흐름과 실제 제어 흐름이 동일하다.
- 싱글 스레드 환경에서 메인 스레드를 긴 시간 점유하면, 프로그램을 멈추게 한다.

동기적 제어 흐름

code

```
let a = 10
console.log("a : ", a)

function foo(num) {
  for (let i = 0; i < 10; ++i) {
    console.log(num)
  }
}

foo(num)
```

동기 vs 비동기



비동기적 제어 흐름

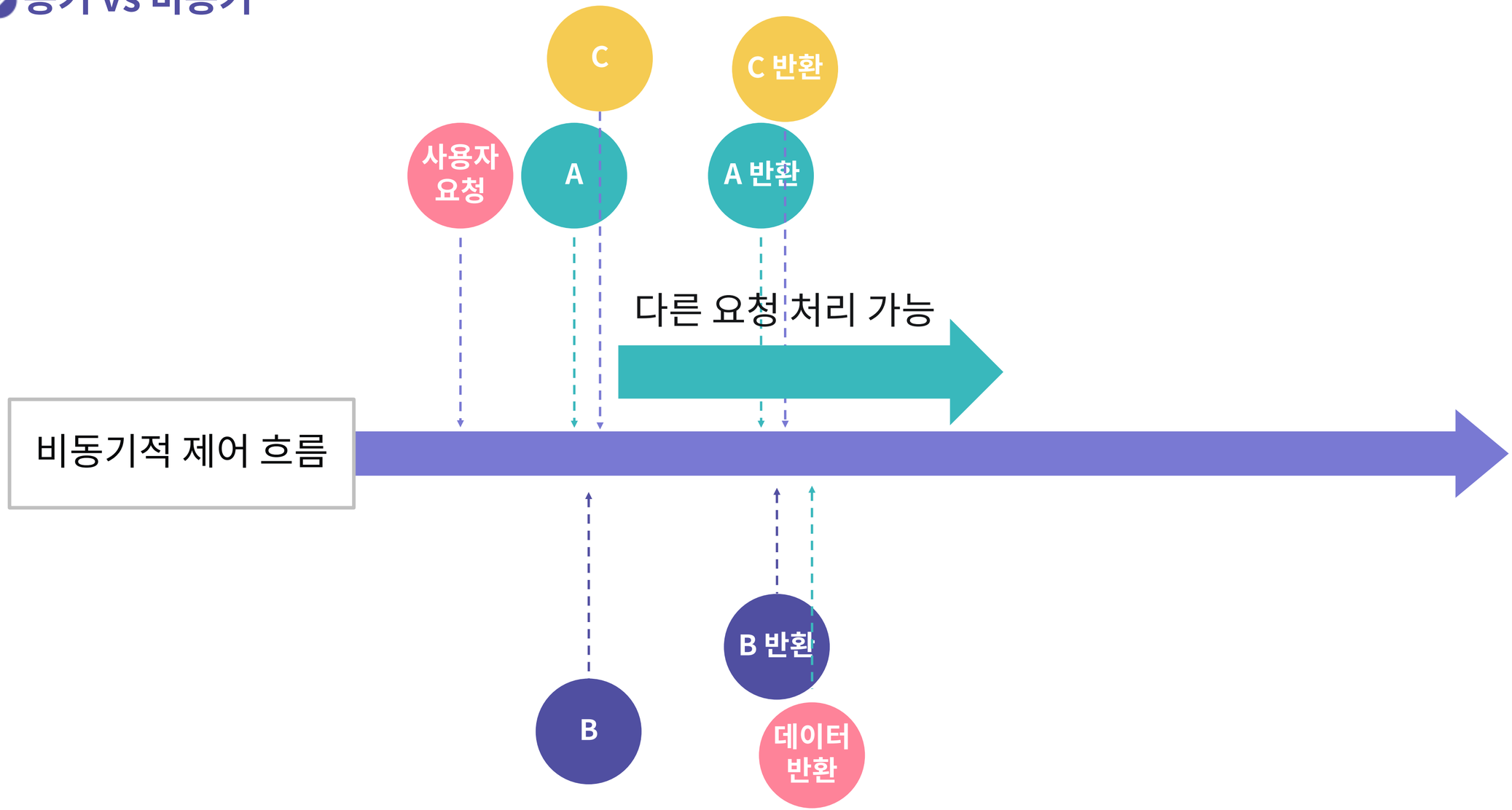
- 비동기적 제어 흐름은 현재 실행 중인 코드가 종료되기 전에 다음 라인의 코드를 실행하는 것을 의미한다.
- 프로미스, 콜백 함수를 호출하는 함수 등은 비동기적으로 실행된다.
- 코드 흐름과 실제 제어 흐름이 다르다.
- 비동기 작업을 기다리는 동안 메인 스레드는 다른 작업을 처리한다.

비동기적 제어 흐름

code

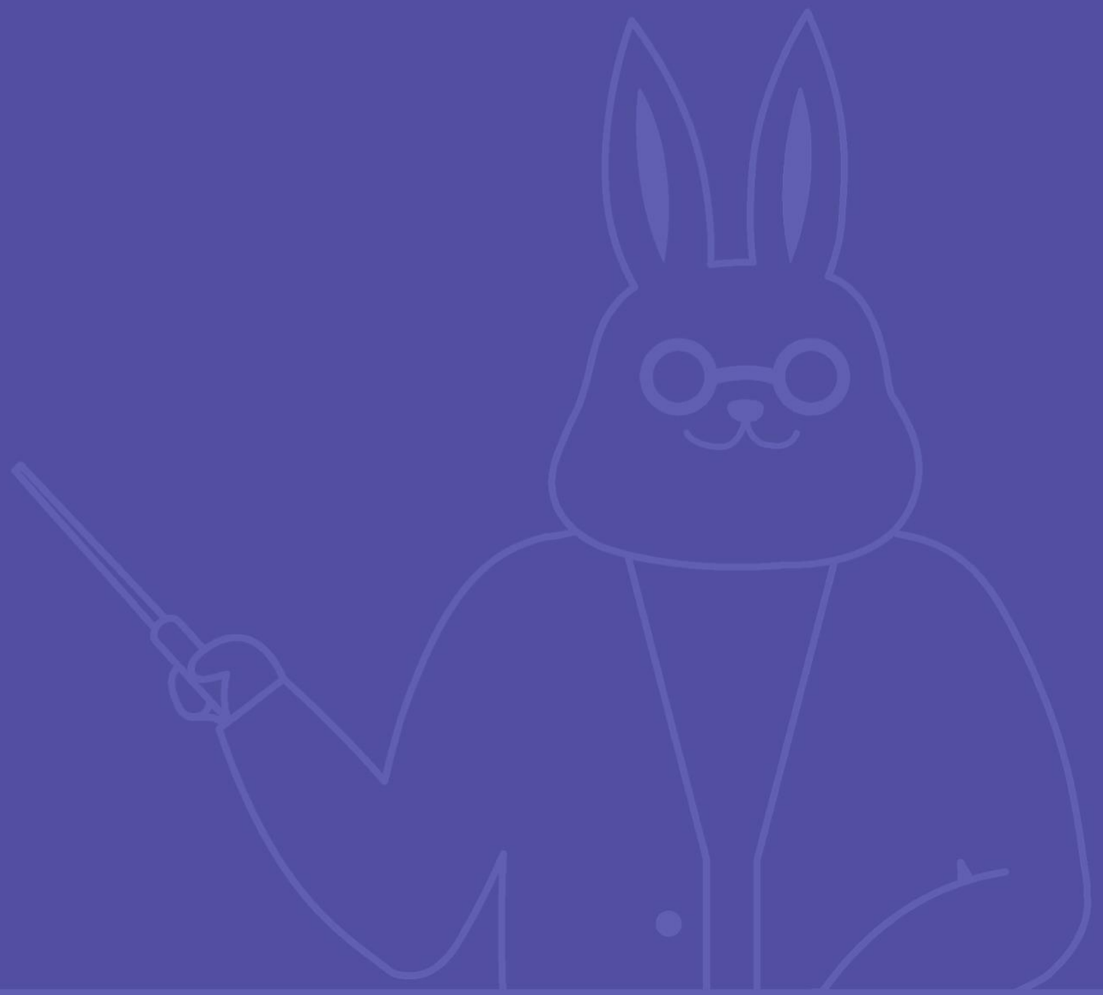
```
let a = 10

setTimeout(function callback() {
  console.log('a : ', a)
}, 3000)

console.log('Finished')
```

동기 vs 비동기



02

이벤트 루프



Copyright Elice. All Rights Reserved

자바스크립트 비동기

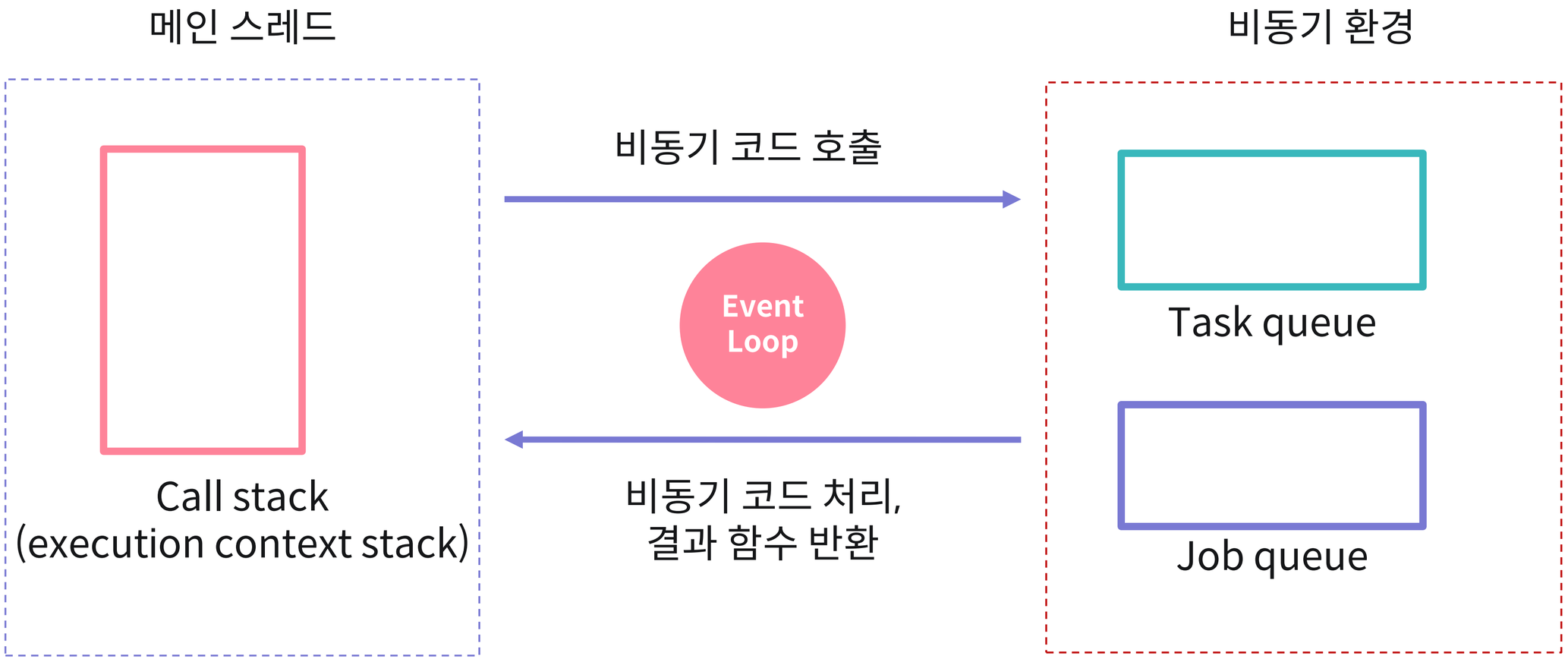
- 자바스크립트 엔진은 비동기 처리를 제공하지 않는다.
- 대신, 비동기 코드는 정해진 함수를 제공하여 활용할 수 있다.
- 이 함수들을 API(Application Programming Interface)라 한다.
- 비동기 API의 예시로, setTimeout, XMLHttpRequest, fetch 등의 Web API가 있다.
- node.js의 경우 파일 처리 API, 암호화 API 등을 제공한다.

비동기 코드 예시

code

```
// 타이머 비동기 처리
setTimeout(() => console.log('타이머 끝'), 1000)
setInterval(() => console.log('인터벌 타이머'), 1000)

// 네트워크 처리
fetch('https://google.com')
  .then(() => console.log('네트워크 요청 성공.'))
  .catch(() => console.log('네트워크 요청 실패.'))
```

비동기 처리 모델


메인 스레드
비동기 환경
비동기 코드 호출
Event Loop
Task queue
Job queue
Call stack
(execution context stack)
비동기 코드 처리,
결과 함수 반환

비동기 처리 모델

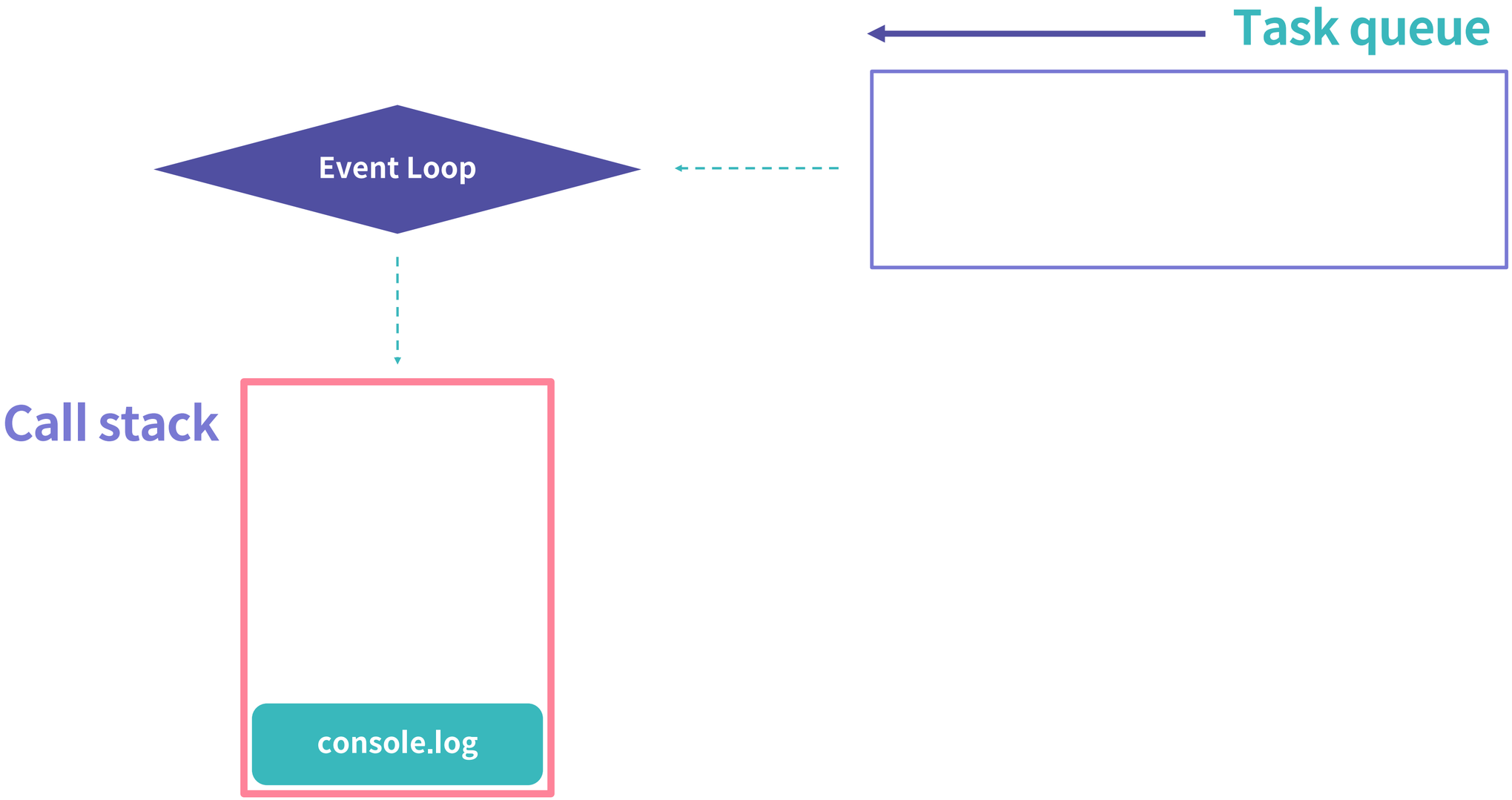
- 비동기 코드를 처리하는 모듈은 자바스크립트 엔진 외부에 있다.
- 이벤트 루프(event loop), 태스크 큐(task queue), 잡 큐(job queue) 등으로 구성된다.
- API 모듈은 비동기 요청을 처리 후 태스크 큐에 콜백 함수를 넣는다.
- 자바스크립트 엔진은 콜 스택이 비워지면, 태스크 큐의 콜백 함수를 실행한다.

비동기 처리 예시

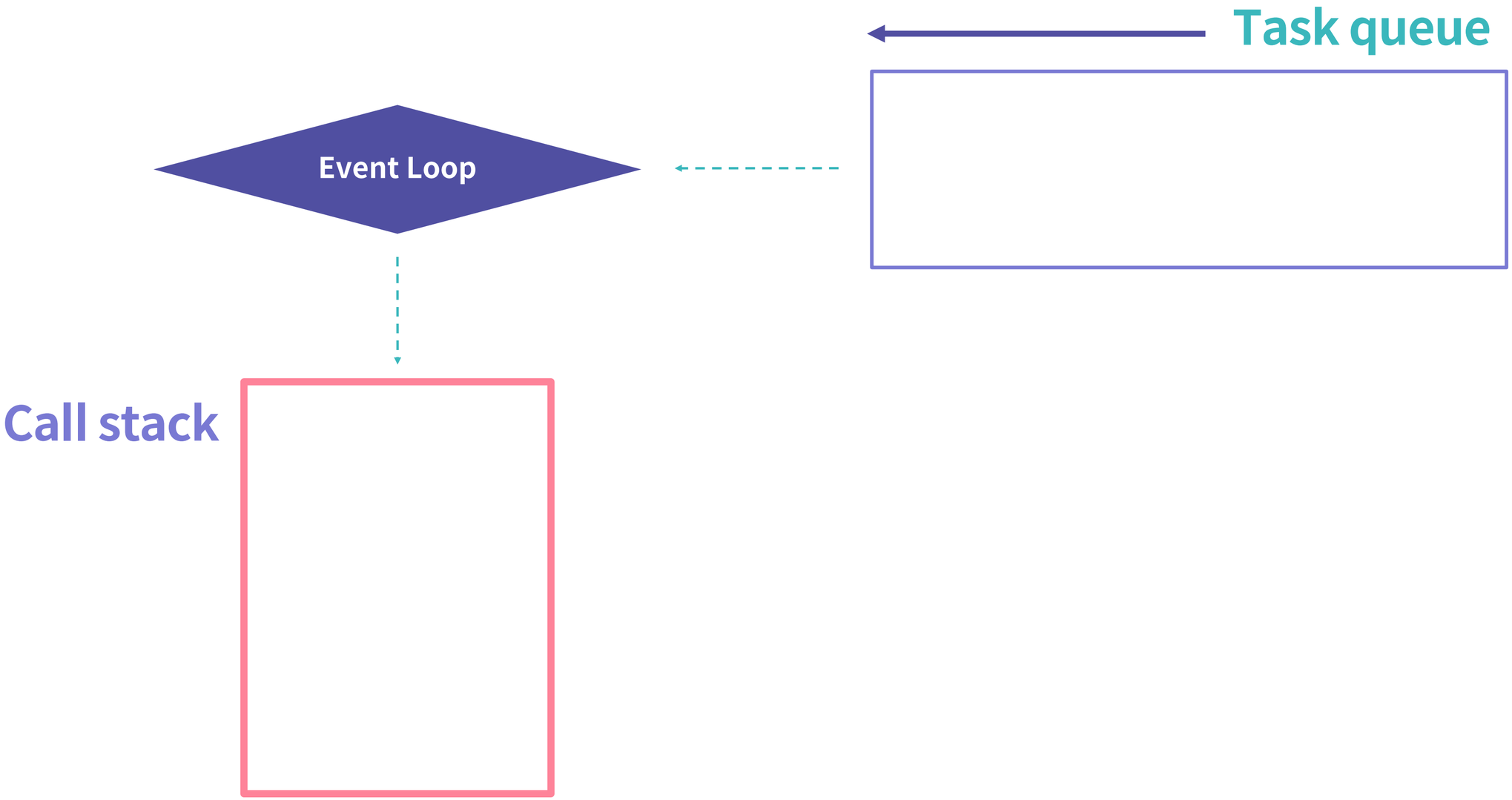
code

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

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

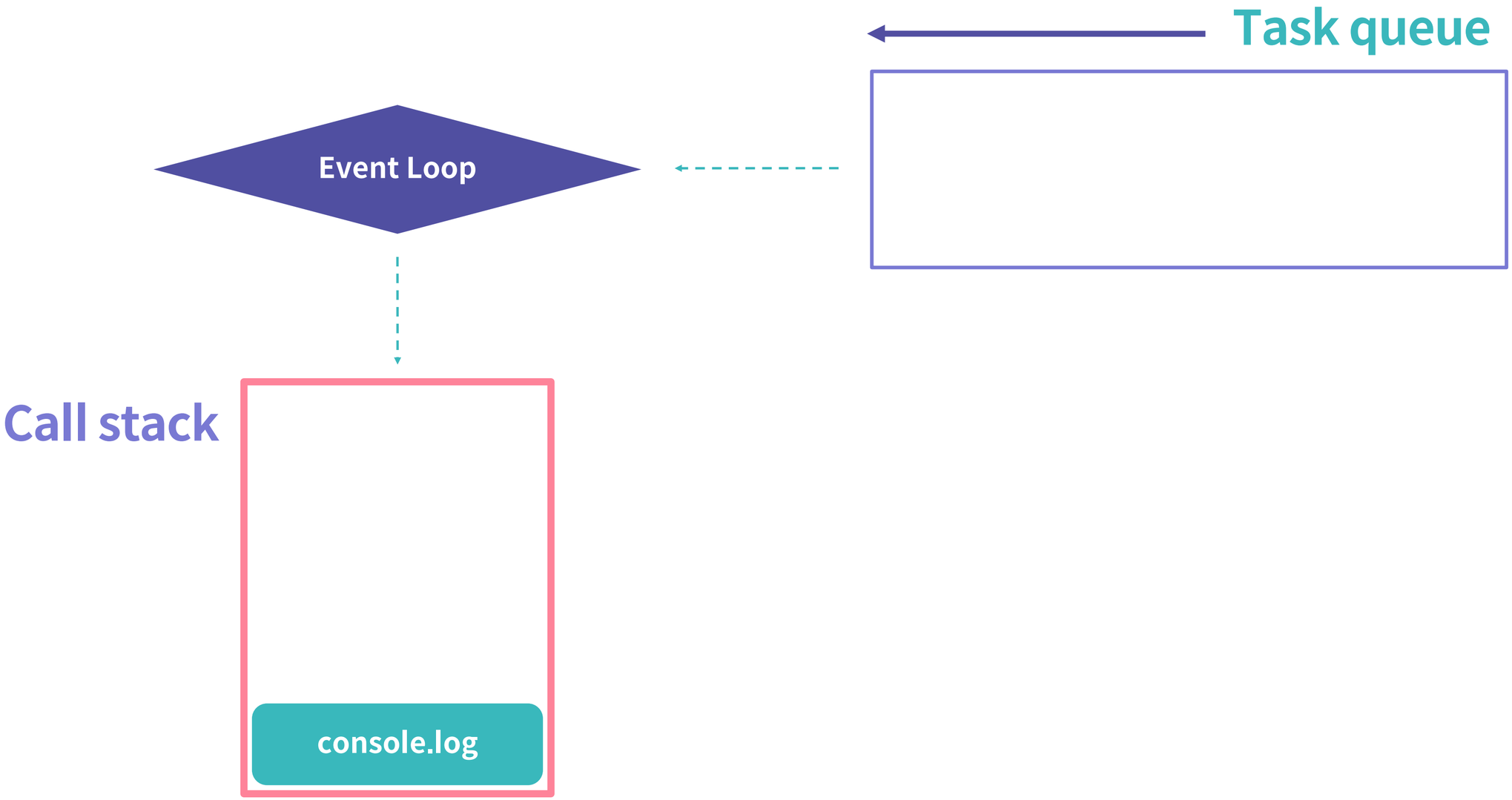
비동기 처리를 위한 내부 구조 - 다이어그램



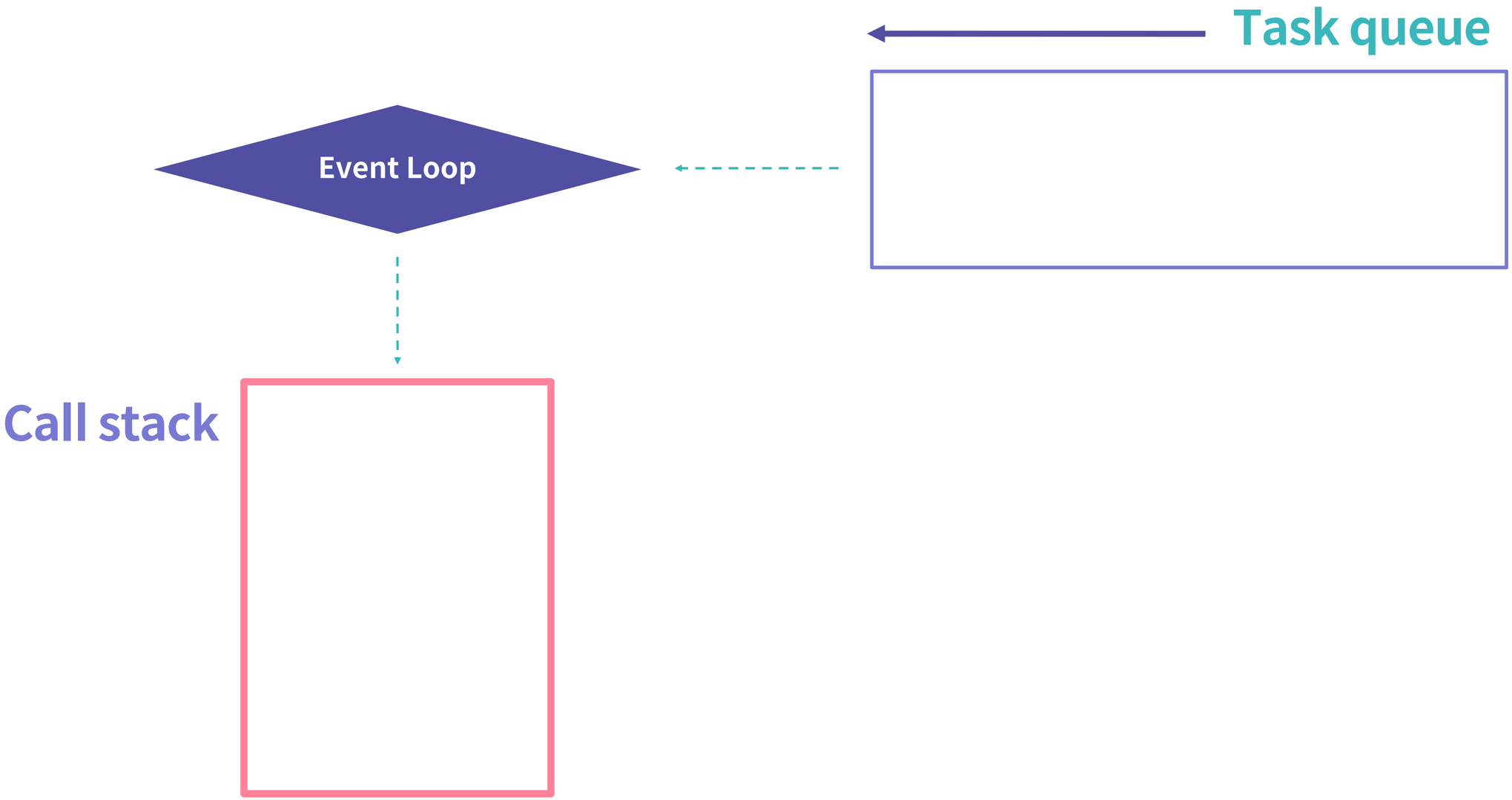
비동기 처리를 위한 내부 구조 - 다이어그램



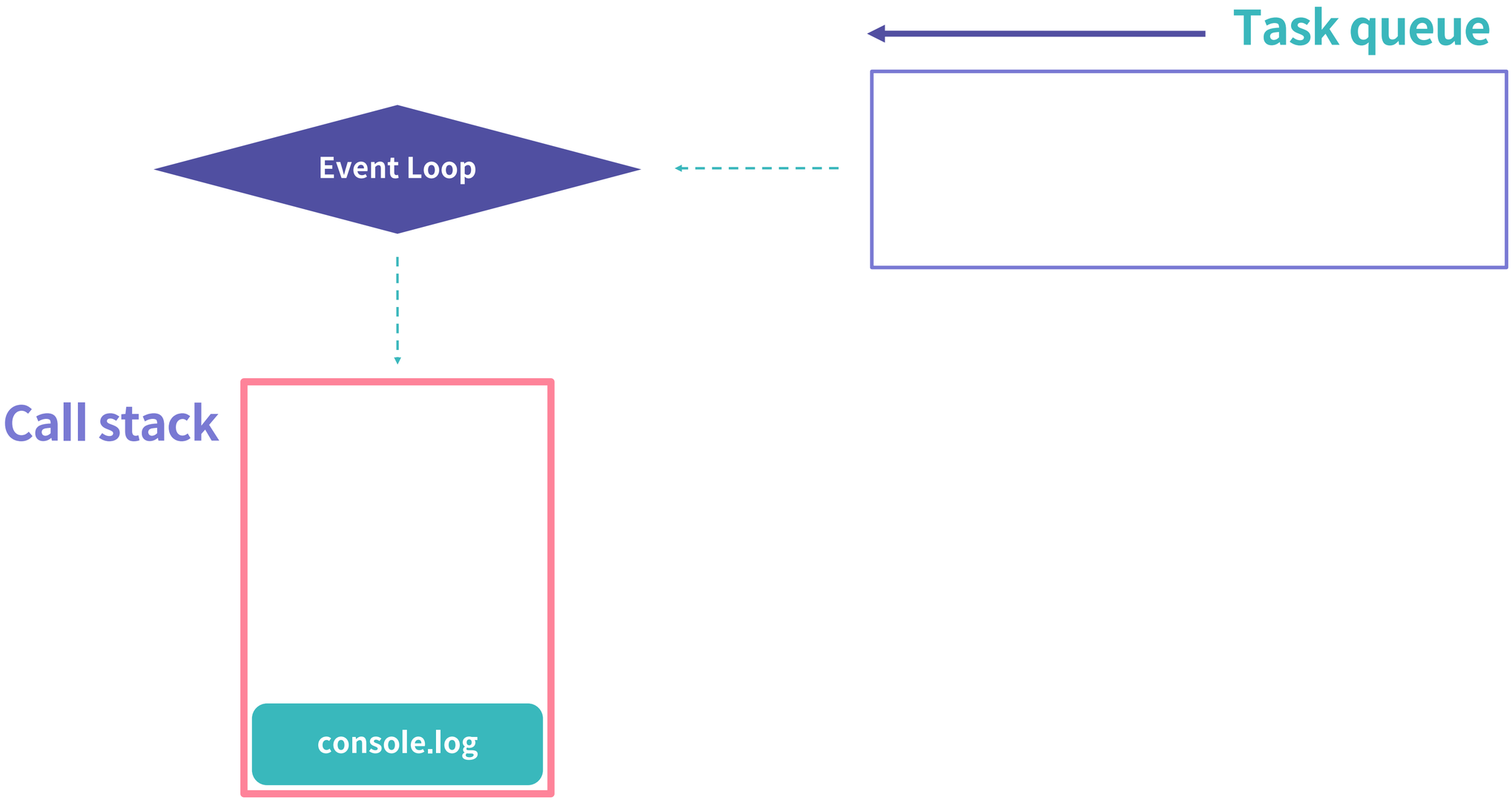
비동기 처리를 위한 내부 구조 - 다이어그램



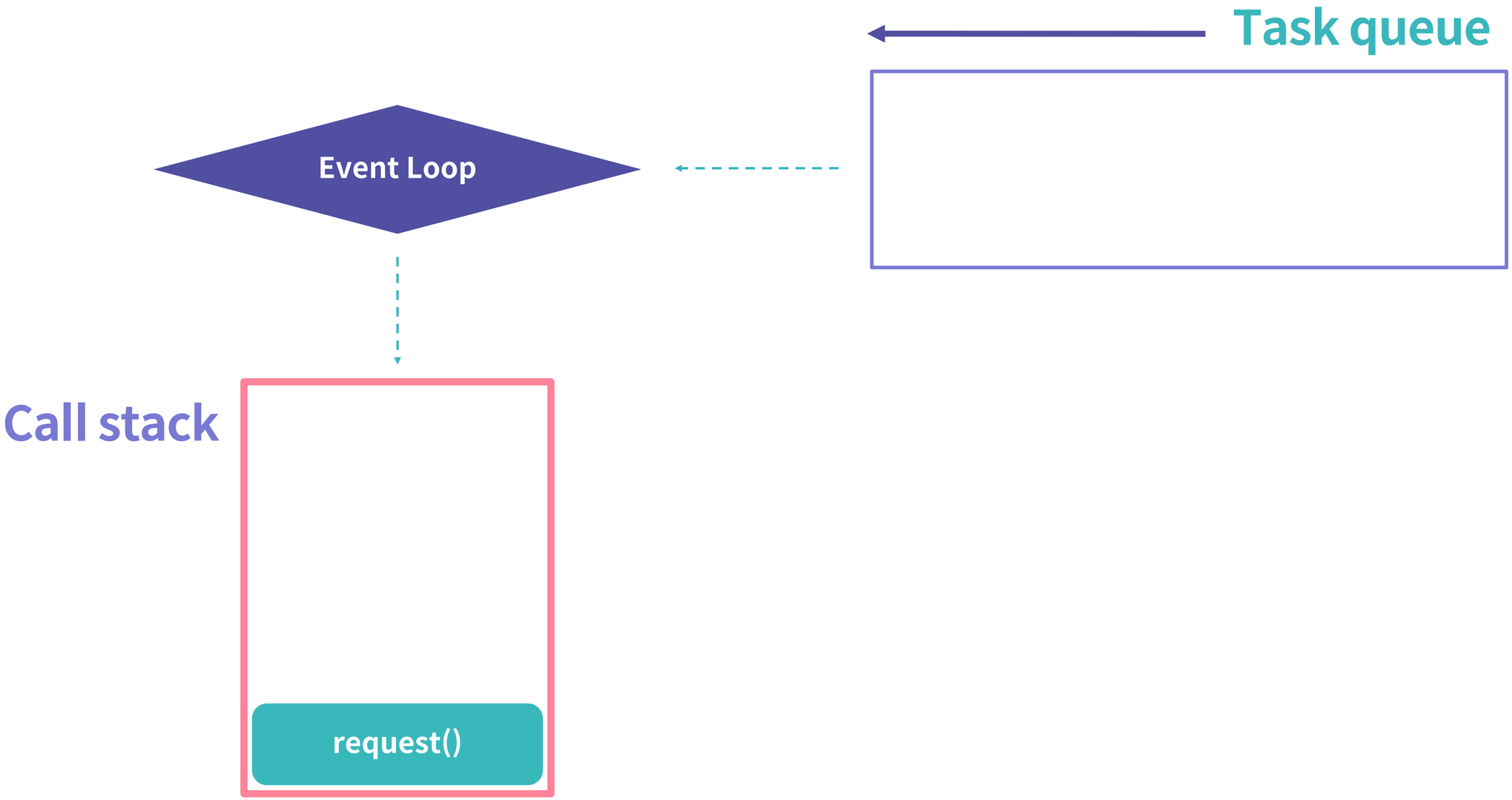
비동기 처리를 위한 내부 구조 - 다이어그램



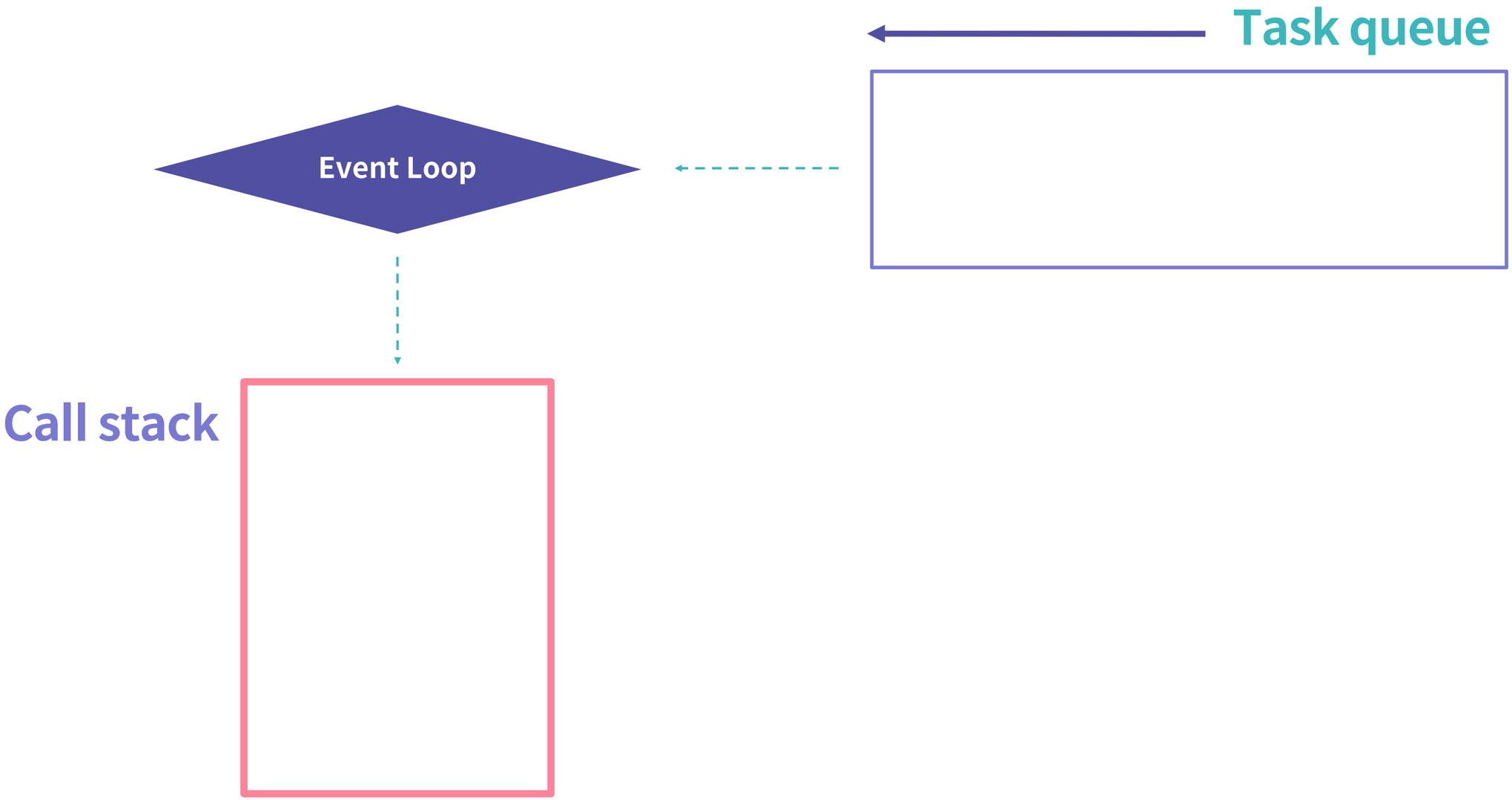
비동기 처리를 위한 내부 구조 - 다이어그램



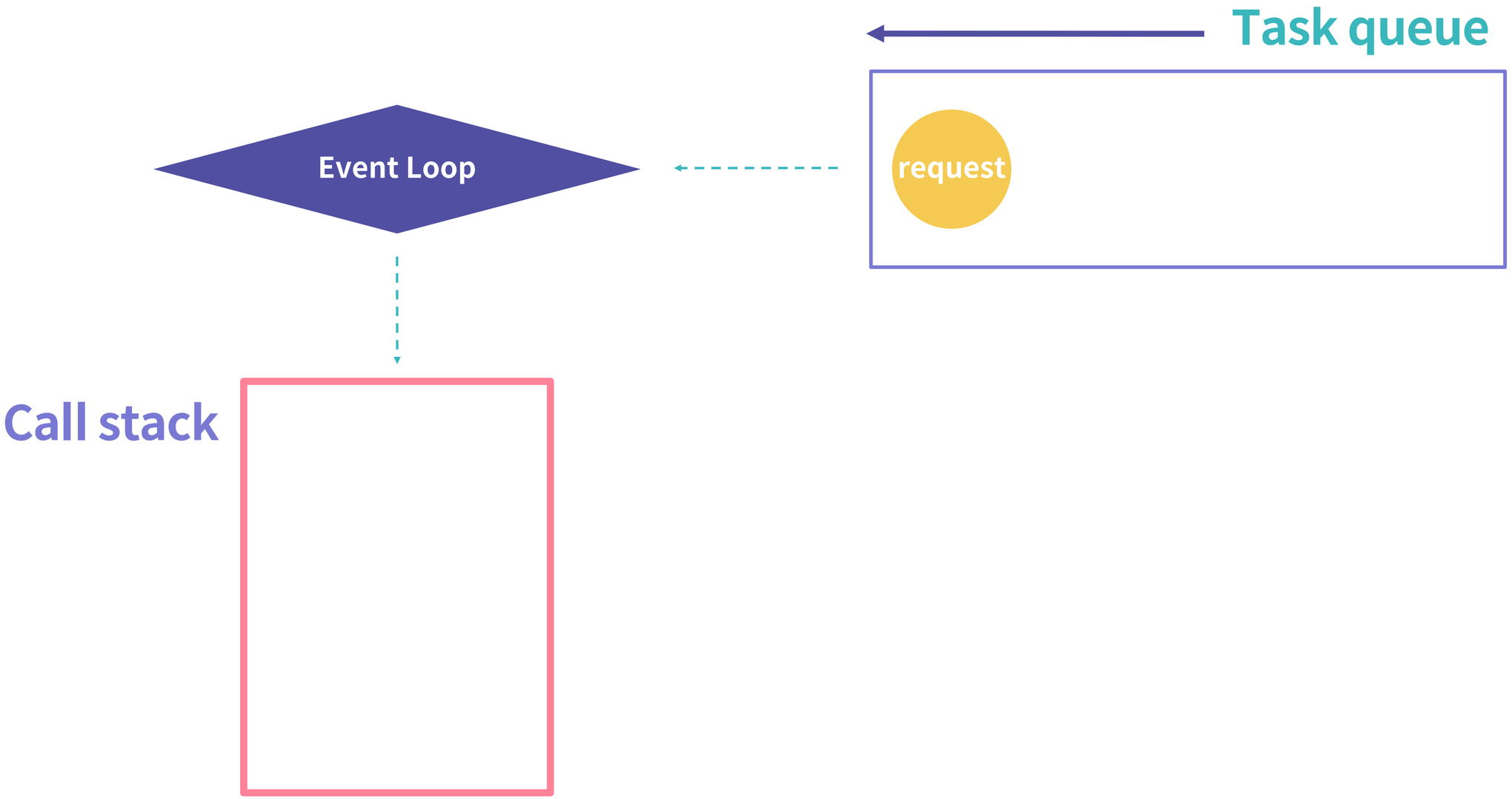
비동기 처리를 위한 내부 구조 - 다이어그램



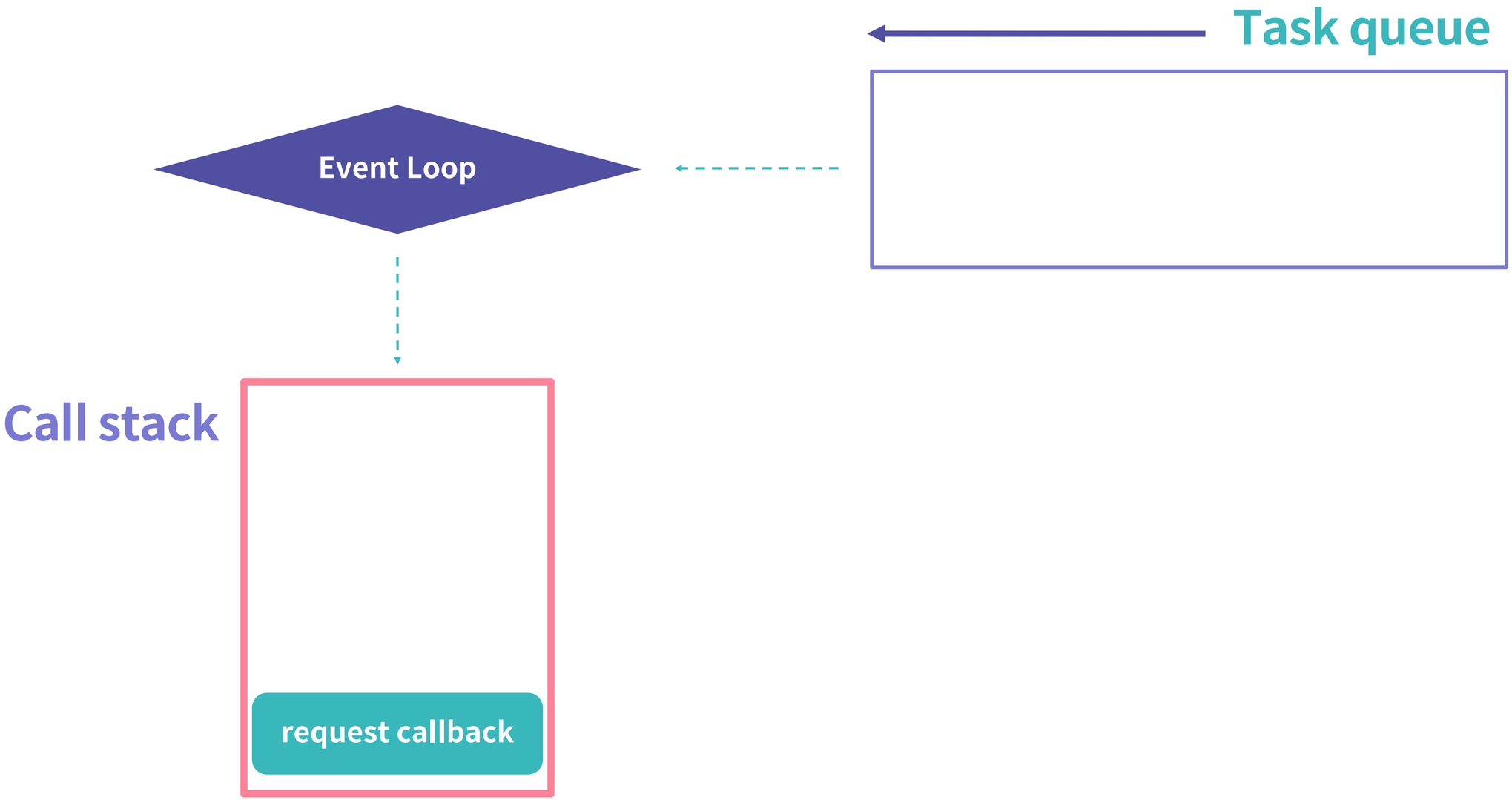
비동기 처리를 위한 내부 구조 - 다이어그램



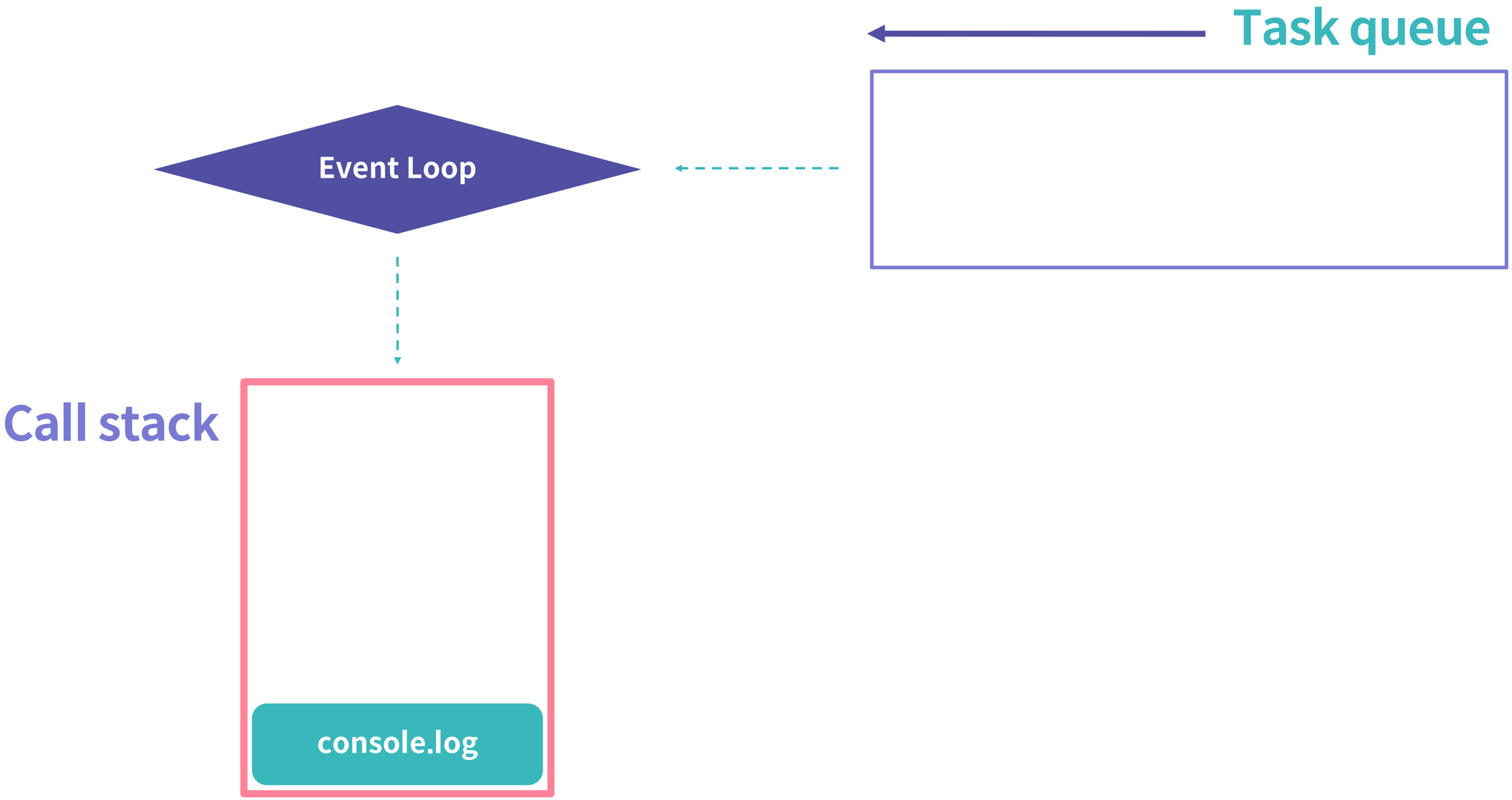
비동기 처리를 위한 내부 구조 - 다이어그램



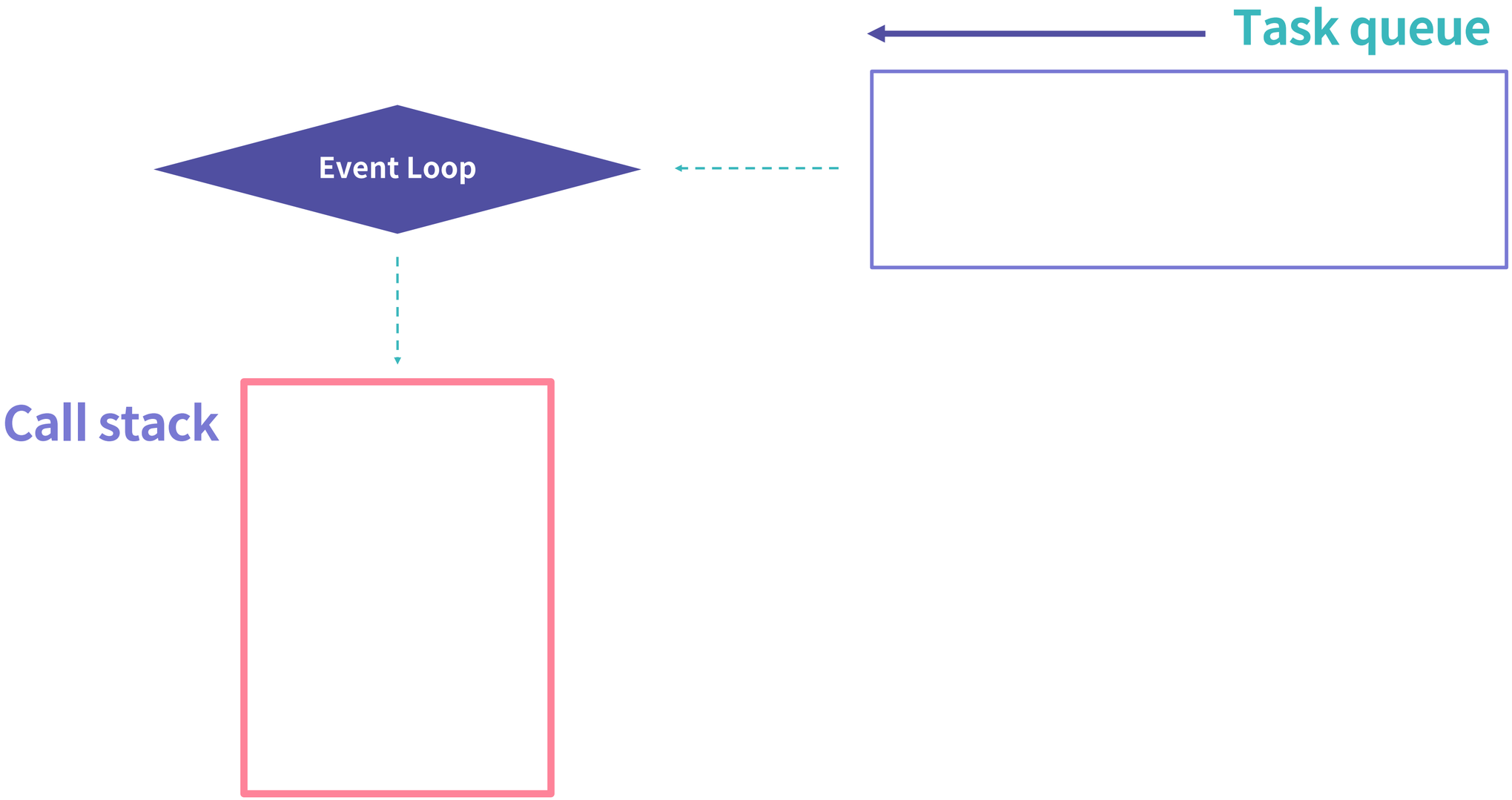
비동기 처리를 위한 내부 구조 - 다이어그램



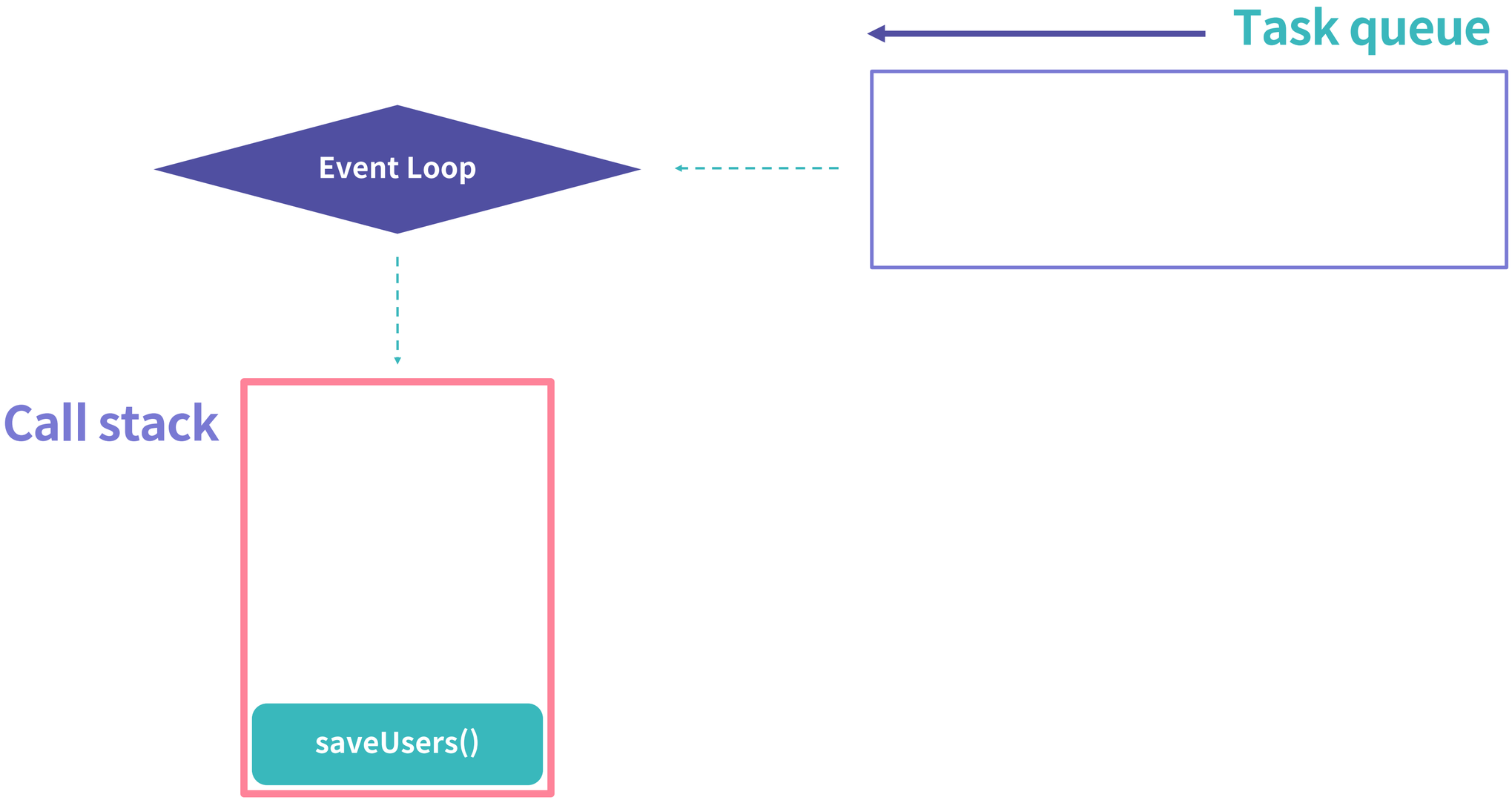
비동기 처리를 위한 내부 구조 - 다이어그램



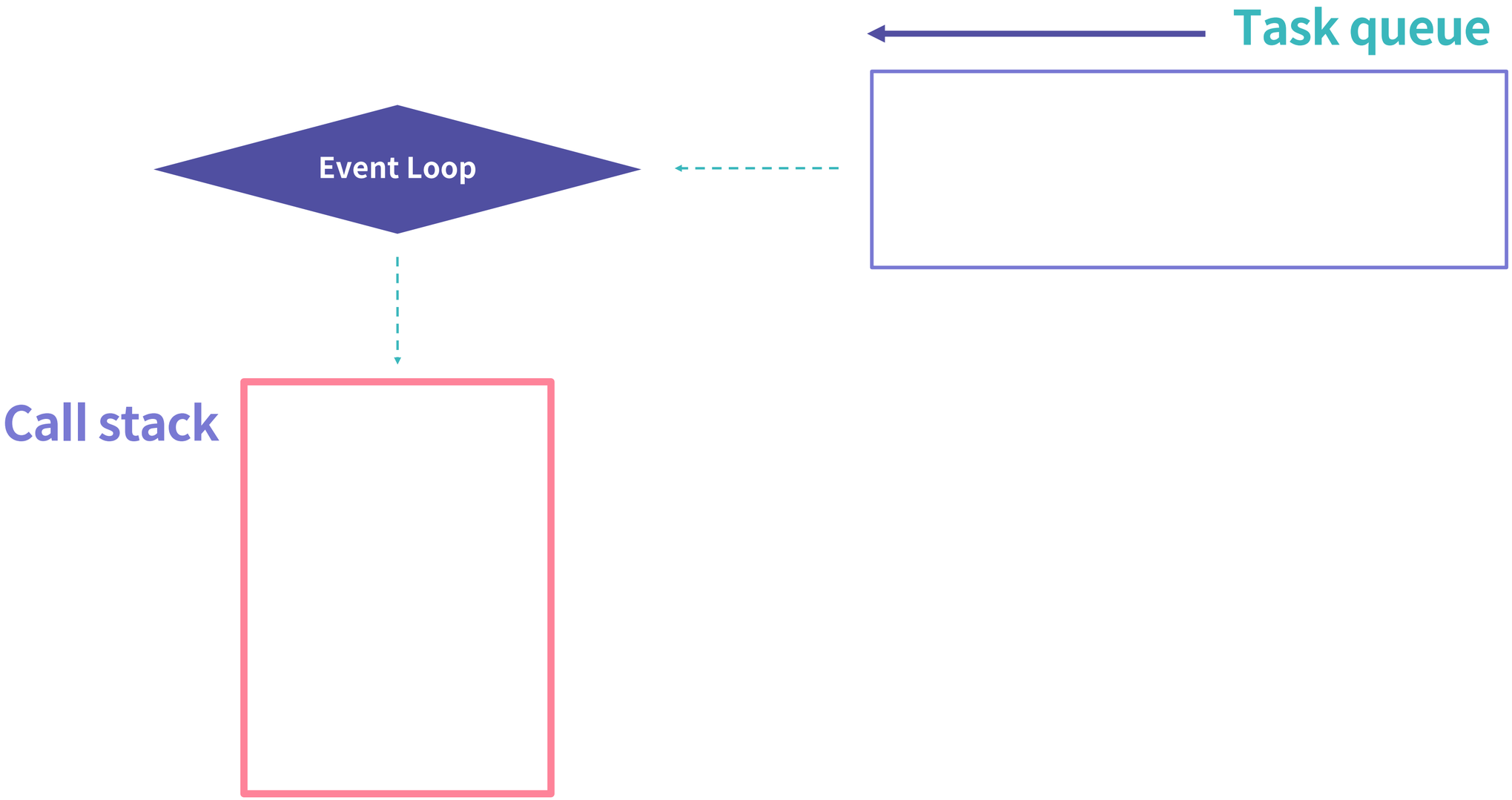
비동기 처리를 위한 내부 구조 - 다이어그램



비동기 처리를 위한 내부 구조 - 다이어그램

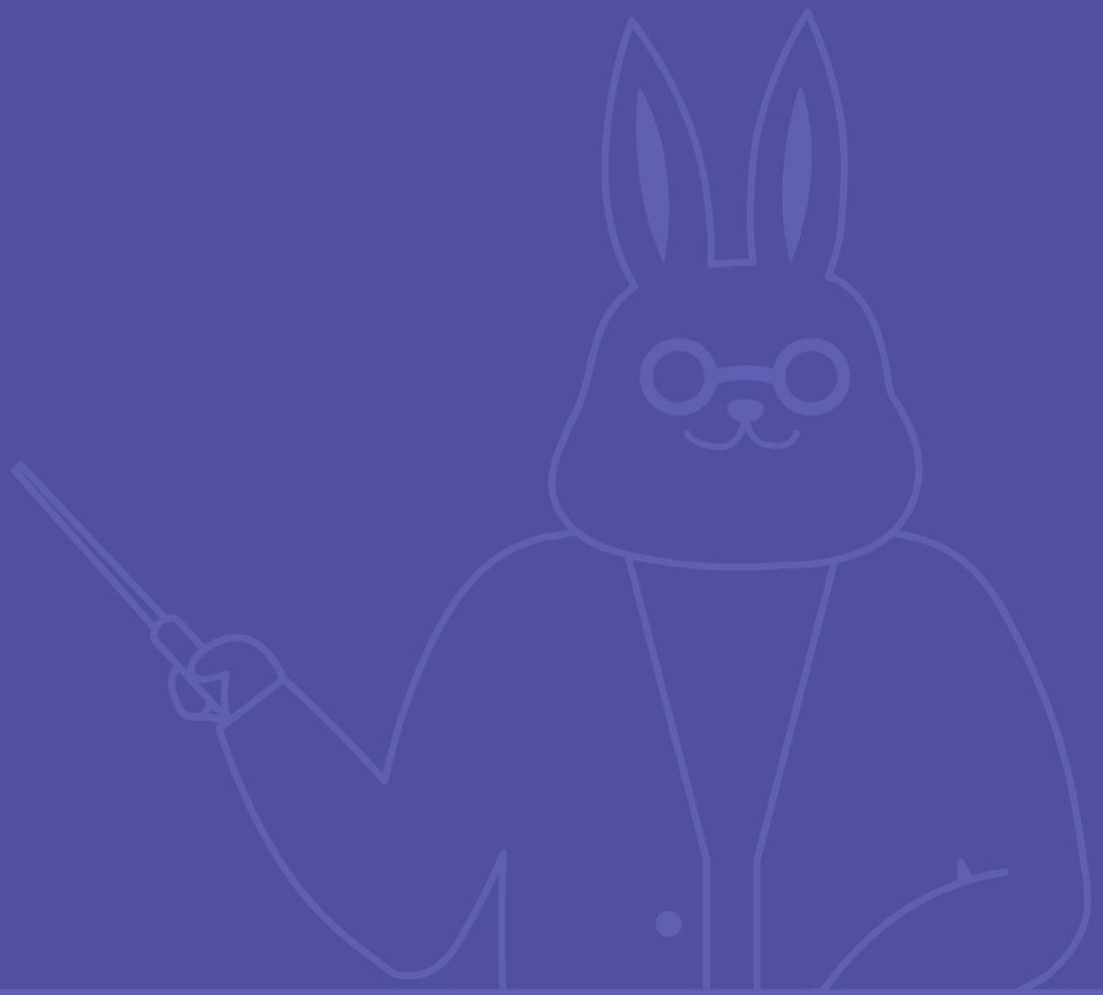


비동기 처리를 위한 내부 구조 - 다이어그램



03

Promise



Copyright Elice. All Rights Reserved

Promise API

- Promise API는 비동기 API 중 하나이다.
- 태스크 큐가 아닌 잡 큐(Job queue, 혹은 microtask queue)를 사용한다.
- 잡 큐는 태스크 큐보다 우선순위가 높다.

Promise API 예시

code

```
setTimeout(() => {
  console.log("타임아웃 1");
}, 0);

Promise.resolve().then(() => console.log("프로미스 1"));

setTimeout(() => {
  console.log("타임아웃 2");
}, 0);

Promise.resolve().then(() => console.log("프로미스 2"));

// 프로미스 1 프로미스 2
// 타임아웃 1 타임아웃 2
```

Promise

- 비동기 작업을 표현하는 자바스크립트 객체.
- 비동기 작업의 진행, 성공, 실패 상태를 표현한다.
- 비동기 처리의 순서를 표현할 수 있다.

Promise 생성자

code

```
let promise = new
Promise((resolve, reject) =>
{
  if (Math.random() < 0.5) {
    return reject("실패")
  }
  resolve(10)
})
```

- new Promise(callback)
- callback 함수는 (resolve, reject) 두 인자를 받는다.
- Promise가 성공했을 때 resolve를 호출한다.
- Promise가 실패했을 때 reject를 호출한다.

Promise 메서드

code

```
promise
  .then(data => {
    console.log("성공:", data)
  })
  .catch(e => {
    console.log("실패:", e)
  })
  .finally(() => {
    console.log("promise 종료")
  })
```

- then() 메서드에 성공했을 때 실행할 콜백 함수를 인자로 넘긴다.
- catch() 메서드에 실패했을 때 실행할 콜백 함수를 인자로 넘긴다.
- finally() 메서드는 성공/실패 여부와 상관없이 모두 실행할 콜백 함수를 인자로 넘긴다.
- then(callback1, callback2)로 callback1의 자리에 성공, callback2의 자리에 실패 메서드를 인자로 넘길 수 있다.

Promise 메서드 체인

code

```
promise
  .then(data => {
    return fetchUser(data)
  })
  .then(user => {
    console.log('User : ', user)
  })
  .catch(e => {
    console.log("실패:", e)
  })
```

- then/catch 메서드가 또 다른 promise를 리턴하여, 비동기 코드에 순서를 부여한다.
- 이렇게 동일한 객체에 메서드를 연결할 수 있는 것을 체이닝(chaining)이라 한다.
- 함수를 호출한 주체가 함수를 끝낸 뒤 자기 자신을 리턴하도록 하여 구현한다.

Promise.resolve, Promise.reject

code

```
Promise
  .resolve(10)
  .then(console.log)

Promise
  .reject("Error")
  .catch(console.log)
```

- Promise.resolve 함수는 성공한 Promise를 바로 반환한다.
- Promise.reject 함수는 실패한 Promise를 바로 반환한다.
- 인위적으로 Promise 메서드 체인을 만들 수 있다.
- 비동기 코드로 진행해야 하는 상황 등에 유용하게 사용할 수 있다.

Promise.all

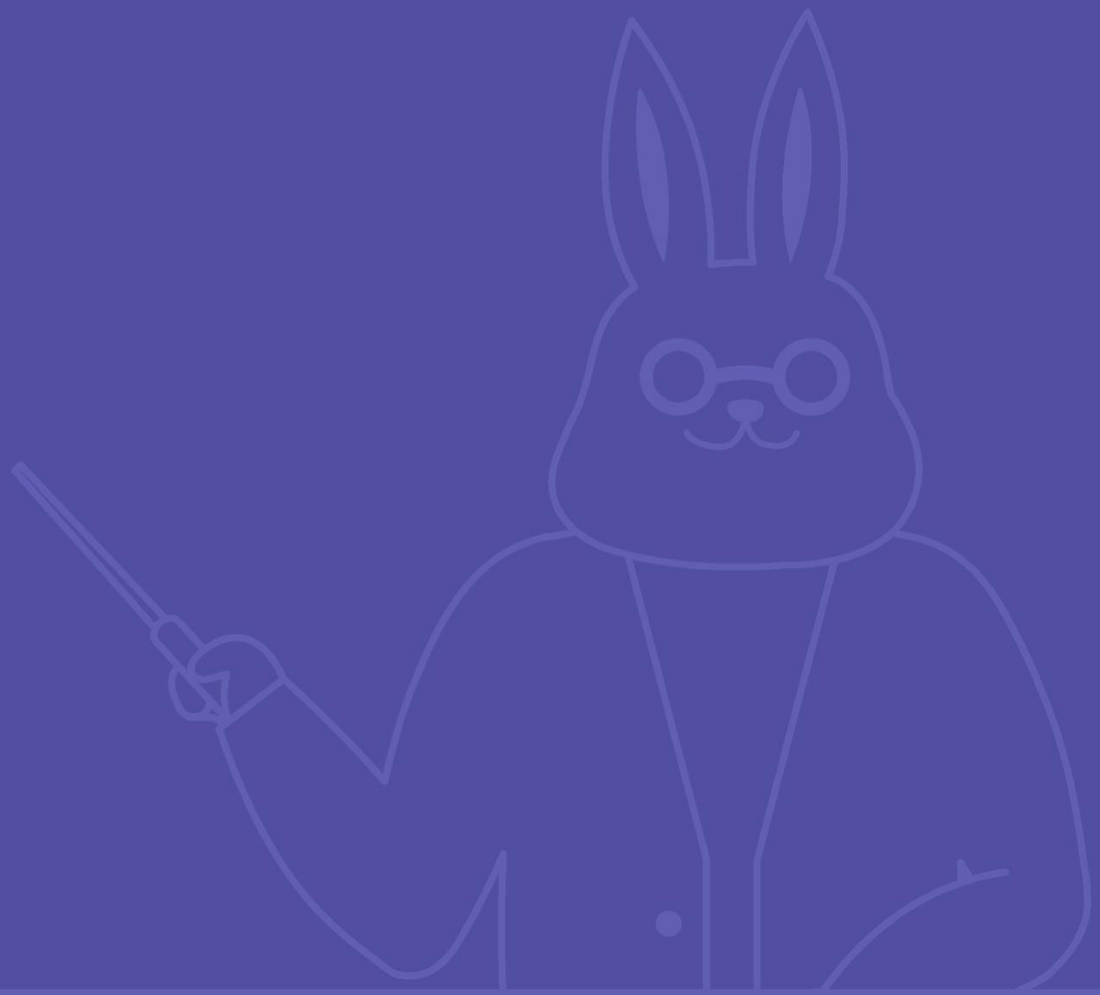
code

```
Promise.all([
  promise1,
  promise2,
  promise3
])
  .then(values => {
    console.log("모두 성공:", values)
  })
  .catch(e => {
    console.log("하나라도 실패:", e)
  })
```

- Promise.all은 Promise의 배열을 받아 모두 성공 시 각 Promise의 resolved 값을 배열로 반환한다.
- 하나의 Promise라도 실패할 시, 가정 먼저 실패한 Promise의 실패 이유를 반환한다.

04

async/await



Copyright Elice. All Rights Reserved

async/await

- Promise를 활용한 비동기 코드를 간결하게 작성하는 문법.
- async/await 문법으로 비동기 코드를 동기 코드처럼 간결하게 작성할 수 있다.
- async 함수과 await 키워드를 이용한다.
- await 키워드는 반드시 async 함수 안에서만 사용해야 한다.
- async로 선언된 함수는 반드시 Promise를 리턴한다.

async 함수

code

```
async function asyncFunc() {
  let data = await fetchData()
  let user = await
fetchUser(data)
  return user
}
```

- async 함수는 function 키워드 앞에 async를 붙여 만든다.
- async 함수 내부에서 await 키워드를 사용한다.
- fetchData, fetchUser는 Promise를 리턴하는 함수이다.

await 키워드 실행 순서

code

```
async function asyncFunc() {
  let data1 = await fetchData1()
  let data2 = await fetchData2(data1)
  let data3 = await fetchData3(data2)
  return data3
}
function promiseFunc() {
  return fetchData1()
    .then(fetchData2)
    .then(fetchData3)
}
```

- await 키워드는, then 메서드 체인을 연결한 것처럼 순서대로 동작한다.
- 비동기 코드에 쉽게 순서를 부여한다.

에러 처리

code

```
function fetchData1() {
  return request()
    .then((response) =>
response.requestData)
    .catch(error => {
      // error 발생
    })
}
```

- Promise를 리턴하는 함수의 경우, 에러가 발생하면 catch 메서드를 이용하여 에러를 처리한다.
- catch 메서드를 사용하지 않는다면 async 함수에서 try-catch 구문을 이용하여 에러를 처리한다.

에러 처리

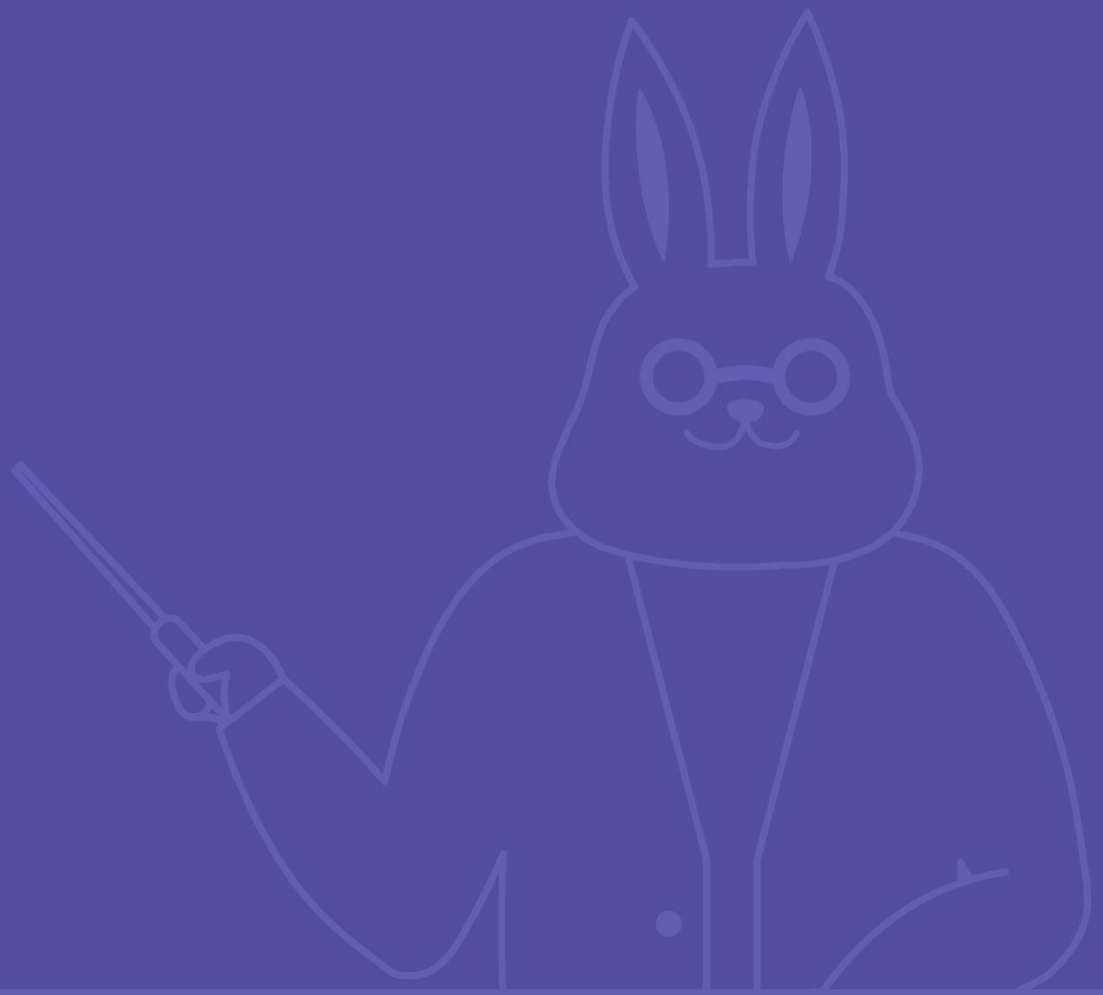
code

```
async function asyncFunc() {
  try {
    let data1 = await
fetchData1()
    return fetchData2(data1)
  } catch (e) {
    console.log("실패:", e)
  }
}
```

- try-catch 구문으로 async/await 형태 비동기 코드 에러 처리가 가능하다.
- catch 절의 e는, Promise의 catch 메서드가 받는 반환 값과 동일하다.

05

HTTP, REST API

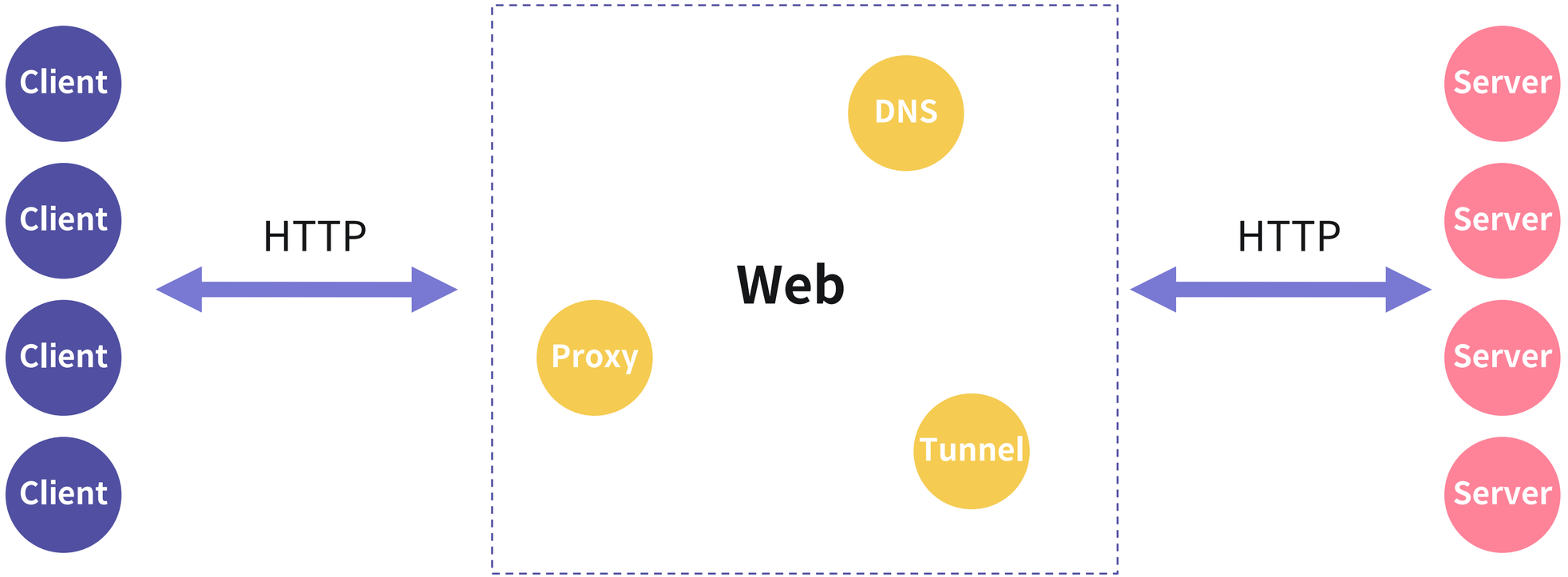


Copyright Elice. All Rights Reserved

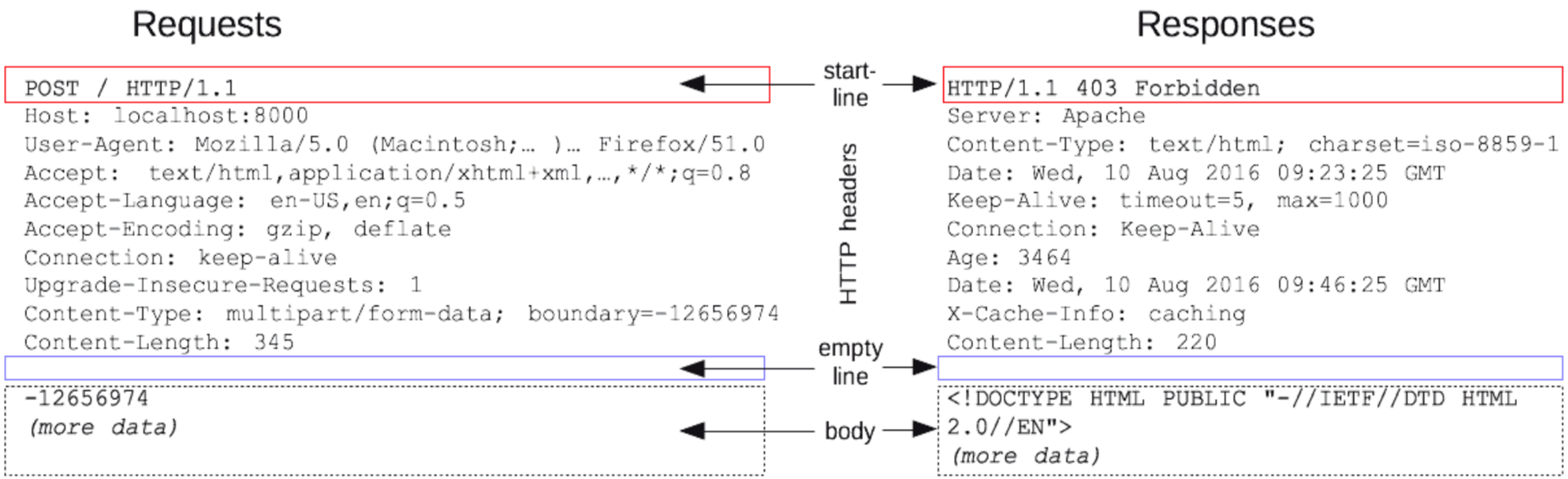
HTTP(Hypertext Transfer Protocol)

- Web에서 서버와 클라이언트 간의 통신하는 방법을 정한 것.
- 클라이언트는 웹 브라우저 등 서버로 요청을 보내는 대상.
- 서버는 클라이언트가 요청을 보내기 전까지 대응하지 않음.
- 서버와 클라이언트 사이에는 무수히 많은 요소가 존재.
- HTTP는 이런 존재들 사이의 통신 방법을 규정.

HTTP(Hypertext Transfer Protocol)



HTTP Message



HTTP Message

- 서버 주소, 요청 메서드, 상태 코드, target path, 헤더 정보, 바디 정보 등이 포함.
- 요청 메시지, 응답 메시지의 모양이 다름.
- HTTP/1.1 메시지는 사람이 읽을 수 있음.

HTTP Header

- HTTP 메시지의 헤더에는 콘텐츠 관련 정보, 인증 관련 정보, 쿠키 정보, 캐시 관련 정보 등 서버와 클라이언트 간 통신 시 필요한 정보를 담는다.
- 클라이언트 요청 시, 서버 응답 시 모두 헤더에 정보를 담을 수 있다.

HTTP Status

- HTTP 요청 시, 클라이언트는 요청의 결과에 대한 상태 정보를 얻는다.
- 200, 400, 500 등 숫자 코드와, OK, NOT FOUND 등의 텍스트로 이루어짐.
- 코드를 이용해 각 결과에 해당하는 행위를 할 수 있음.

요청 메서드

- HTTP에서, 클라이언트는 서버로 요청을 보낸다.
- 요청 시 요청 메서드로 특정 요청에 대한 동작을 정의한다.
- GET, POST, PUT, PATCH, DELETE, OPTIONS, CONNECT, TRACE 등이 규정됨.

REST API(Representational State Transfer API)

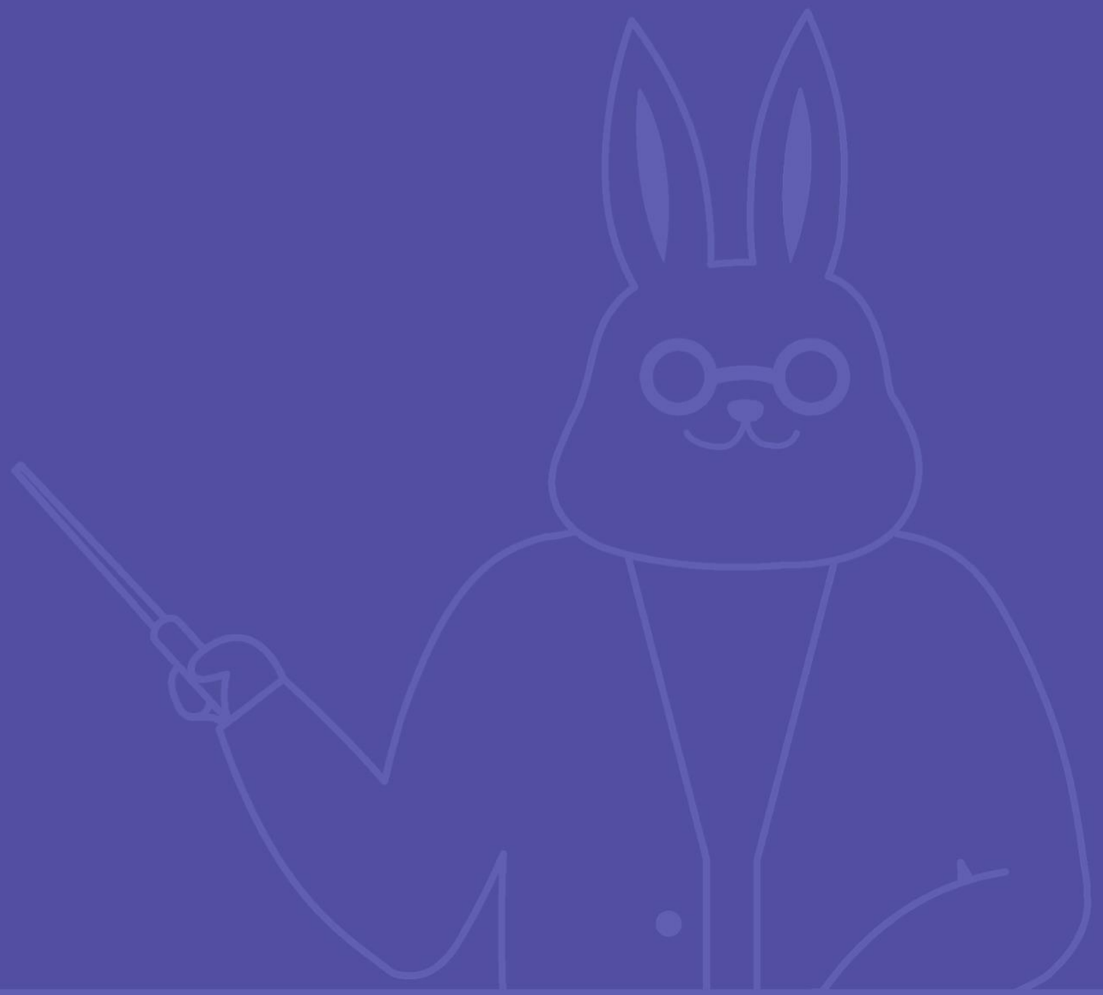
- API(Application Programming Interface)는 사용자가 특정 기능을 사용할 수 있도록 제공하는 함수를 의미한다.
- REST API는 HTTP의 요청 메서드에 응하는 서버 API와 클라이언트 간 통신의 구조가 지켜야 할 좋은 방법을 명시한 것이다.
- 구체적인 내용으로는 요청 메서드의 의미, URI 설계, 클라이언트의 상태에 대한 동작 등을 정의한다.

REST API 요청 메서드의 의미

- GET - 리소스 정보를 얻음.
- POST - 리소스를 생성.
- PUT - 리소스를 생성하거나 업데이트.
- DELETE - 리소스를 제거.

06

Fetch API



Copyright Elice. All Rights Reserved

Fetch API

code

```
let result = fetch(serverURL)

result
  .then(response => {
    if (response.ok) {
      // 요청 성공.
    }
  })
  .catch(error => {
    // 요청 실패.
  })
```

- 기존 XMLHTTPRequest를 대체하는 HTTP 요청 API.
- ES6에 추가된 Promise를 리턴하도록 정의됨.
- 네트워크 요청 성공 시, Promise는 Response 객체를 resolve 한다.
- 네트워크 요청 실패 시, Promise는 에러를 reject 한다.

Response

code

```
fetch(serverURL)
  .then(response => {
    response.ok
    response.status
    response.statusText
    response.url
    response.bodyUsed
  })
```

- Response 객체는 결과에 대한 다양한 정보를 담는다.
- response.ok는 HTTP Status code가 200-299 사이면 true, 그 외 false이다.
- response.status는 HTTP status code를 담는다.
- response.url은 요청한 URL 정보를 담는다.

Header

code

```
fetch(serverURL)
  .then(resp => {
    for (let [k, v] of
resp.headers) {
      console.log(k, v)
    }
  })
```

- response.headers로 Response 객체의 헤더 정보를 얻을 수 있다.

Body 메서드

code

```
fetch(serverURL)
  .then(response => {
    return response.json()
  })
  .then(json => {
    console.log('body : ', json)
  })
```

- response.json() 메서드는 얻어온 body 정보를 json으로 만드는 Promise를 반환한다.
- Promise가 resolve 되면 얻어온 body 정보를 읽는다.
- response.text(), response.blob(), response.formData() 등의 메서드로 다른 형태의 바디를 읽는다.

POST 요청

code

```
fetch(serverURL, {
  method: 'post',
  headers: {
    'Content-Type':
'application/json;charset=utf-8',
    Authentication: 'mysecret'
  },
  body: JSON.stringify(formData)
})
  .then(response => {
    return response.json()
  })
  .then(json => {
    console.log('POST 요청 결과:', json)
  })
```

- fetch(url, options) 로, fetch 메서드 옵션을 넣는다.
- method 필드로 여러 요청 메서드를 활용한다.
- headers, body 필드를 활용해 서버에 추가 정보를 보낸다.

크레딧

/* elice */

코스 매니저
이재성

콘텐츠 제작자
김일식

강사
김일식

감수자
김일식

디자이너
강혜정

연락처

TEL
070-4633-2015

WEB
https://elice.io

E-MAIL
contact@elice.io