Express

Hata İşleme

Error Handling, senkron ve asenkron olarak meydana gelen hataların Express tarafından nasıl yakalandığına ve işlendiğine değinir. Express varsayılan olarak bir hata işleyiciyle gelir, bu nedenle hata işlemeye başlamak için kendinizin yazmanıza gerek yoktur.

Hataları Yakalamak

Express tarafından, rota işleyicileri ve ara yazılımları koşarken oluşan hataların yakalanmasının sağlanması önemlidir. Rota işleyicilerinde ve ara yazılımlarda senkron kodda oluşan hataları yakalamak için ek birşey yapmaka gerek yoktur. Eğer senkron kod bir hata fırlatırsa, Express onu yakalayıp işleyecektir. Örneğin:

```
app.get('/', (req, res) => {
  throw new Error('BROKEN') // Express bunu kendi kendine yakalayacak
})
```

Rota işleyicileri ve ara yazılım tarafından çağrılan asenkron fonksiyonlardan dönen hataları Express'in yakalayıp işleyeceği `next()` fonksiyonuna vermelisiniz. Örnek olarak:

```
app.get('/', (req, res, next) => {
  fs.readFile('/file-does-not-exist', (err, data) => {
    if (err) {
      next(err) // Hataları Express'e ver
    } else {
      res.send(data)
    }
  })
})
```

Express 5 ile başlayarak, Promise döndüren rota işleyicileri ve ara yazılımlar ret verdiklerinde (reject) veya hata fırlattıklarında otomatik olarak `next(value)` fonksiyonunu çağıracaklar. Örneğin:

```
app.get('/user/:id', async (req, res, next) => {
  const user = await getUserById(req.params.id)
  res.send(user)
})
```

`getUserById` bir hata fırlattığında veya ret verdiğinde, `next` fonksiyonu ya fırlatılan hatayla ya da ret verilen değer ile çağrılacaktır. Eğer herhangi bir ret değeri verilmediyse, `next` fonksiyonu Express yönlendiricisi tarafından sağlanan varsayılan Error objesiyle çağrılacak.

Eğer `next()` fonksiyonuna herhangi birşey verdiğinizde (`'route'` karakter dizisi hariç), Express şimdiki isteği bir hata olarak sayıp hata işlemeyen yönlendirici ve ara yazılım fonksiyonlarını es geçecektir.

Eğer bir dizideki geri çağırma fonksiyonu veri sağlamıyorsa, sadece hataları veriyorsa, kodu bu şekilde basitleştirebilirsiniz:

```
app.get('/', [
  function (req, res, next) {
    fs.writeFile('/erişilemez-yol', 'data', next)
  },
  function (req, res) {
    res.send('OK')
  }
])
```

Yukarıdaki örnekte `next`, hata veya hatasız olarak çağrılan `fs.writeFile` için geri çağırma fonksiyonu olarak verlidi. Eğer hata yok ise ikinci işleyici çalışacak, aksi takdirde Express hatayı yakalayıp işler.

Rota işleyicileri ve ara yazılımlar tarafından çağrılan asenkron kodda oluşan hataları yakalayıp işlemesi için Express'e geçmelisiniz. Örnek olarak:

```
app.get('/', (req, res, next) => {
  setTimeout(() => {
    try {
      throw new Error('BROKEN')
    } catch (err) {
      next(err)
    }
  }, 100)
})
```

Yukarıdaki örnek asenkron kodda hataları yakalamak için bir `try...catch` bloku kullanıyor. Eğer `try...catch` bloku olmaz ise, işleyici senkron kodun bir parçası olmadığı için Express hatayı yakalamayacak.

`try..catch` blokunun yükünden kaçınmak için promise veya promise döndüren fonksiyonlar kullanın. Örnek olarak:

```
app.get('/', (req, res, next) => {
  Promise.resolve().then(() => {
    throw new Error('BROKEN')
  }).catch(next) // Hatalar Express'e geçer
})
```

Promise'lar otomatik olarak senkron hatalarını ve ret edilen promise'ları yakaladığından, sonuncu yakalama işleyicisi olarak `next` fonksiyonunu verebilirsiniz ve Express hataları yakalar, çünkü yakalama işleyicisine birinci argüman olarak hata verimiştir.

Senkron hata yakalamaya güvenmek için asenkron kodu basite indirgeyerek bir işleyiciler zincirini de kullanabilirsiniz. Örnek olarak:

```
app.get('/', [
  function (req, res, next) {
    fs.readFile('/maybe-valid-file', 'utf-8', (err, data) => {
      res.locals.data = data
      next(err)
    })
  },
  function (req, res) {
```

```
    res.locals.data = res.locals.data.split(',')[1]
    res.send(res.locals.data)
  }
])
```

Yukarıdaki örnek `readFile` çağrısından birkaç basit ifadeye sahip. `readFile` bir hata alırsa, bu hatayı Express'e verir, aksi takdirde hızlı bir şekilde zincirdeki bir sonraki işleyicide asenkron hata işleme dünyasına dönersiniz. Ve, yukarıdaki örnek veriyi işlemeyi gösterir. Bu işlem hata verirse, onu senkron hata işleyicisi yakalayacak. Eğer bu işlemleri `readFile` geri çağırma fonksiyonunun içinde yaptıysanız uygulama kapanabilir ve Express hata işleyicileri çalışmaz.

Hangi yöntemi kullanırsanız kullanın, Express hata işleyicilerinin çağrılmasını ve uygulamanın hayatta kalmasını istiyorsanız, Express'in hatayı aldığından emin olmalısınız.

Varsayılan hata işleyicisi

Express, uygulamada oluşabilecek herhangi bir hatayla ilgilenecek gömülü bir hata işleyicisiyle gelir. Bu varsayıla hata işleyici ara yazılım fonksiyonu, ara yazlım fonksiyon yığınının en sonuna eklenir.

`next()` fonksiyonuna bir hata verip özel bir hata işleyicisinde işlemezseniz, bu hata gömülü hata işleyicisi tarafından işlenir; hata istemcide stack-trace ile beraber yazdırılır. Stack-trace üretim (production) ortamında dahil değildir.

> Uygulamayı üretim modunda koşmak için `NODE_ENV` ortam değişkeninin değerini `production` olarak ayarlayın.

Bir hata yazdırıldığında, aşağıdaki bilgiler yanıta eklenir:

- `res.statusCode` alanının değeri `err.status` alanından gelir (veya `err.statusCode`). Eğer bu değer 4xx veya 5xx'in aralığında değilse, 500 olarak ayarlanacak.
- `res.statusMessage` alanı statü koduna göre ayarlanır.
- Üretim modunda ise, gövde (body) statü kodu mesajının HTML'i olur, aksi takdirde ise `err.stack`.
- `err.headers` objesinde belirtilen herhangi bir başlık (header).

`next()` fonksiyonunu yanıtı yazmaya başladıktan sonra bir hata ile çağırırsanız (örneğin, istemciye yanıtı aktarma esnasında bir hata ile karşılaşırsanız) varsayılan Express hata işleyicisi bağlantıyı kapatıp isteği başarısız kılar.

Özel bir hata işleyicisi eklediğiniz zaman başlıklar (header) halihazırda istemciye gönderilmiş ise varsayılan Express hata işleyicisine yetki vermelisiniz:

```
function errorHandler (err, req, res, next) {
  if (res.headersSent) {
    return next(err)
  }
  res.status(500)
  res.render('error', { error: err })
}
```

Kodunuzda `next()` fonksiyonunu bir hata ile birden fazla kez çağırdığınızda varsayılan hata işleyicisi tetiklenebilir, özel hata işleyici ara yazılımı yerinde olsa bile.

Hata işleyicileri yazmak

Hata işleyici ara yazılım fonksiyonlarını diğer ara yazılım fonksiyonları gibi tanımlayınız, bundan farklı olarak hata işleyici fonksiyonlar üç yerine dört argümana sahipler: `(err, req, res, next)`. Örneğin:

```
app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Birşeyler bozuldu')
})
```

Hata işleyici ara yazılımları diğer `app.use()` ve rotaların çağrılarından sonra, en son tanımlanır; örnek olarak:

```
const bodyParser = require('body-parser')
const methodOverride = require('method-override')

app.use(bodyParser.urlencoded({
  extended: true
}))
app.use(bodyParser.json())
app.use(methodOverride())
app.use((err, req, res, next) => {
  // iş mantığı
})
```

Ara yazılımdaki yanıtlar HTML hata sayfası, basit bir mesaj veya bir JSON karakter dizisi gibi herhangi bir biçimde olabilir.

Organizasyonel (ve daha üst-düzey çatı) amaçlar için, normal ara yazılım fonksiyonları gibi, birden fazla hata-işleyici ara yazılım fonksiyonu tanımlayabilirsiniz. Örnek olarak, XHR kullanılan ve XHR kullanılmayan istekler için bir hata işleyici tanımlamak gibi:

```
const bodyParser = require('body-parser')
const methodOverride = require('method-override')

app.use(bodyParser.urlencoded({
  extended: true
}))
app.use(bodyParser.json())
app.use(methodOverride())
app.use(logErrors)
app.use(clientErrorHandler)
app.use(errorHandler)
```

Bu örnekte, jenerik `logErrors`, `stderr`'a istek ve hata bilgileri yazabilir, örneğin:

```
function logErrors (err, req, res, next) {
  console.error(err.stack)
  next(err)
}
```

Ayrıca bu örnekte, `clientErrorHandler`aşağıdaki gibi tanımlanır; bu durumda, hata açıkça bir sonrakine aktarılır.

Bir hata işleme fonksiyonunda "next" ***çağrılmadığında***, yanıtın yazılmasından (ve sonlandırılmasından) siz sorumlusunuz. Aksi takdirde o istekler "havada" kalır ve çöp toplama (garbage collection) için geçerli olmayacaktır.

```
function clientErrorHandler (err, req, res, next) {
  if (req.xhr) {
```

```
    res.status(500).send({ error: 'Birşeyler ters gitti' })
  } else {
    next(err)
  }
}
```

"Hepsini-yakala" `errorHandler` fonksiyonunu aşağıdaki gibi tanımlayın:

```
function errorHandler (err, req, res, next) {
  res.status(500)
  res.render('error', { error: err })
}
```

Birden fazla geri çağırma fonksiyonu olan bir rota işleyiciniz var ise bir sonraki rota işleyicisine geçmek için `route` parametresini kullanabilirsiniz. Örnek:

```
app.get('/a_route_behind_paywall',
  (req, res, next) => {
    if (!req.user.hasPaid) {
      // bu isteği işlemeye devam et
      next('route')
    } else {
      next()
    }
  }, (req, res, next) => {
    PaidContent.find((err, doc) => {
      if (err) return next(err)
      res.json(doc)
    })
  })
```

Bu örnekte, `getPaidContent` işleyicisi es geçilecek ama `app` uygulaması `/a_route_behind_paywall` yolu için geriye kalan herhangi bir işleyici çalışmaya devam edecek.

> `next()` ve `next(err)` fonksiyonlarına yapılacak çağrılar şimdiki işleyicinin tamamlandığını ve hangi durumda tamamlandıklarını belirtir. `next(err)` çağrısı, yukarıda gösterildiği gibi hata işlemek için kurulanlar hariç, zincirde geriye kalan bütün işleyicileri es geçer.