

Node.JS ile Web Programlama

Rapor - 4

Mahire Zühal Özdemir

İÇİNDEKİLER

ŞEKİLLER DİZİN	3
Arrow Function	4
Callback Fonksiyonu	4
This Komutu	4
Arrow Function ve Callback Fonksiyonları	4
Not Uygulaması	12
app.js Dosyası	12
Notes.js Dosyası	12
List Komutu / Fonksiyonu	13
Filter / Find Komutu	13
Read Komutu / Fonksiyonu	13
Debug İşlemleri	14

ŞEKİLLER DİZİNİ

Şekil 1: this, setTimeout uygulamaları	5
Şekil 2: İlk Çıktı	5
Şekil 3: 1 sn sonra çıktı	5
Şekil 4: Arrow Function ve this komutu	5
Şekil 5: Uygulama Çıktısı	6
Şekil 6: Obje tanımlayarak this kullanımı	6
Şekil 7: Uygulama Çıktısı	6
Şekil 8: Arrow function vs Klasik fonksiyon	7
Şekil 9: Uygulama Çıktısı	7
Şekil 10: Uygulamalar	7
Şekil 11: Uygulama Çıktısı	7
Şekil 12: Arrow function	8
Şekil 13: Arrow function çıktı	8
Şekil 14: await ve setTimeout fonksiyonları	9
Şekil 15: await ve setTimeout fonksiyonları çıktı	9
Şekil 16: Fonksiyon	10
Şekil 17: Uygulama Çıktısı	10
Şekil 18: Arrow Function	11
Şekil 19: Komut Çıktısı	11
Şekil 20: Arrow function	11
Şekil 21: app.js dosyası	14
Şekil 22: Note_command.js dosyası	14

Arrow Function

Node.js'de arrow functionlar, ES6 ile tanıtılan bir JavaScript özelliğidir. Arrow fonksiyonlar, geleneksel JavaScript fonksiyonlarından farklı bir sözdizimine sahiptir ve genellikle daha kısa ve okunması daha kolay kod yazmanıza olanak tanır.

Arrow fonksiyonları tanımlamanın temel sözdizimi şu şekildedir:

```
(params) => {statements}
```

Callback Fonksiyonu

'**setTimeout()**' JavaScript'in bir fonksiyonudur ve belirli bir süre sonra belirtilen bir işlevi çalıştırmak için kullanılır. Bu komut, belirli bir zaman aralığı (milisaniye cinsinden) geçtikten sonra belirtilen işlevi bir kez çalıştırır.

setTimeout() fonksiyonu asenkron bir şekilde çalışır ve bir geri çağırım fonksiyonunu (callback function) belirli bir süre sonra çalıştırır. Asenkron bir işlem, işlem tamamlanana kadar beklemek yerine, işlem tamamlanmadan önce diğer işlemlere devam etmeyi sağlar.

Arrow function ve asenkron çalışmayı daha iyi anlamak için çeşitli kodlar oluşturarak uygulamalarını görelim.

This Komutu

'**this**' anahtar kelimesi, JavaScript'te mevcut olan özel bir kelime veya referanstır. Bu kelime, bir fonksiyonun hangi nesne tarafından çağrıldığını veya bir nesne içinde hangi kapsamda olduğunu belirlemek için kullanılır. Kullanımı, bağlamına bağlı olarak değişiklik gösterebilir.

Arrow Function ve Callback Fonksiyonları

- Klasik fonksiyon tanımlama yöntemi ile fonksiyon tanımlayıp içerisinde **setTimeout** ve **this** komutlarını aşağıdaki gibi kullanalım.

```
function regularFunction() {
  console.log("Regular function 'this':", this);
  //this; regularFunction fonksiyonunun çağrıldığı bağlamı (context) gösterir
  setTimeout(function() {
    console.log("Regular function callback 'this':", this);
    //this; setTimeout fonksiyonunun çağrıldığı bağlamı (context) gösterir.
  }, 1000);
}
regularFunction();
```

Şekil 1: this, setTimeout uygulamaları

Uygulama çıktısı olarak;

```
Regular function 'this': <ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  performance: Performance {
    nodeTiming: PerformanceNodeTiming {
      name: 'node',
      entryType: 'node',
      startTime: 0,
      duration: 59.97719997167587,
      nodeStart: 0.9240999817848206,
      v8Start: 4.7692999839782715,
      bootstrapComplete: 40.5654000043869,
      environment: 22.02319997549057,
      loopStart: -1,
      loopExit: -1,
      idleTime: 0
    },
    timeOrigin: 1710320974979.425
  },
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  }
}
```

Şekil 2: İlk Çıktı

```
Regular function callback 'this': Timeout {
  _idleTimeout: 1000,
  _idlePrev: null,
  _idleNext: null,
  _idleStart: 60,
  _onTimeout: [Function (anonymous)],
  _timerArgs: undefined,
  _repeat: null,
  _destroyed: false,
  [Symbol(refed)]: true,
  [Symbol(kHasPrimitive)]: false,
  [Symbol(asyncId)]: 5,
  [Symbol(triggerId)]: 1
}
```

Şekil 3:1 sn sonra çıktı

- Arrow function yöntemi ile fonksiyon tanımlayarak this komutunu aşağıdaki gibi kullanalım.

```
const arrowFunction = () => {
  console.log("Arrow function 'this':", this);
  setTimeout(() => {
    console.log("Arrow function callback 'this':", this);
  }, 1000);
};
arrowFunction();
```

Şekil 4: Arrow Function ve this komutu

Bu kodun çıktısı aşağıdaki gibi olur.

```
C:\Users\zuhal\Desktop\Node.Js\Hafta-4\Lab>node arrow.js
Arrow function 'this': {}
Arrow function callback 'this': {}
```

Şekil 5: Uygulama Çıktısı

- Obje tanımladıktan sonra içerisinde tanımlanan arrow fonksiyonlar ile this komutu ekrana yazdırılır.

```
const obj = {
  value: "Object value",
  methodWithRegular: function() {
    console.log("Regular method 'this':", this);
  },
  methodWithArrow: () => {
    console.log("Arrow method 'this':", this);
  }
}
obj.methodWithRegular();
obj.methodWithArrow();
```

Şekil 6: Obje tanımlayarak this kullanımı

Bu komutun çıktısı aşağıdaki gibi olur.

```
Regular method 'this': {
  value: 'Object value',
  methodWithRegular: [Function: methodWithRegular],
  methodWithArrow: [Function: methodWithArrow]
}
Arrow method 'this': {}
```

Şekil 7: Uygulama Çıktısı

Hem arrow fonksiyonunu hem de klasik fonksiyonu kullanarak bir sayı dizisinin karesini alan fonksiyonları yazalım. Örnekte de görüldüğü gibi, tek satırda bu işlemi yapan arrow fonksiyonunun yazılması çok kolaydır.

```
const numbers = [1, 2, 3, 4];

// Regular functions
const squaredRegular = numbers.map(function(num) {
  return num * num;
});

// Arrow functions
const squaredArrow = numbers.map(num => num * num);

console.log(squaredRegular);
console.log(squaredArrow);
```

Şekil 8: Arrow function vs Klasik fonksiyon

```
[ 1, 4, 9, 16 ]
[ 1, 4, 9, 16 ]
```

Şekil 9: Uygulama Çıktısı

Tek/çift sayı kontrolü ve kare alma işlemi yapan fonksiyonları arrow fonksiyonu ile yazalım.

```
const double = x => x * 2;
const isEven = num => num % 2 === 0;

console.log(double(5));
console.log(isEven(4));
```

Şekil 10:Uygulamalar

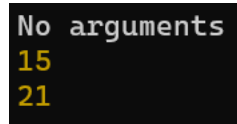
```
10
true
```

Şekil 11: Uygulama Çıktısı

Arrow fonksiyonu tanımlarken tek parametre alınan işlemlerde parantez kullanılmadığına dikkat ediniz. Eğer birden çok parametre tanımlarsak parantez kullanmamız gerekir.

```
const noArgs = () => "No arguments";  
const oneArg = x => x + 10; // Parentheses not required for a single argument  
const multipleArgs = (x, y) => x * y;  
  
console.log(noArgs());  
console.log(oneArg(5));  
console.log(multipleArgs(3, 7));
```

Şekil 12: Arrow function



```
No arguments  
15  
21
```

Şekil 13: Arrow function çıktı

Promise: JavaScript'te bir asenkron işlemin sonucunu temsil eden bir nesnedir. Promise, başarılı bir sonuç veya hata ile sonuçlanabilecek bir işlemi temsil eder. Bu işlemler genellikle ağ istekleri, dosya okuma/yazma gibi uzun sürecek işlemlerdir. Bir Promise, "pending" (beklemede), "fulfilled" (gerçekleşmiş) veya "rejected" (reddedilmiş) olabilir. İşlem tamamlandığında, sonuç veya hata promise nesnesi üzerinden erişilebilir.

await: await anahtar kelimesi, sadece async fonksiyonlar içinde kullanılabilir. Bir await ifadesi, bir Promise'in tamamlanmasını bekler ve ardından Promise'in çözüm değerini döndürür. await, asenkron bir işlemi senkron bir şekilde beklememizi sağlar. Bu sayede, bir işlem tamamlanana kadar diğer işlemlerin gerçekleşmesini bekleyebiliriz.

setTimeout: JavaScript'te zamanlayıcı işlevini temsil eden bir fonksiyondur. Bu fonksiyon, belirli bir süre geçtikten sonra belirtilen bir işlevi (veya kod bloğunu) çalıştırmak için kullanılır. setTimeout fonksiyonu iki argüman alır: birincisi çalıştırılacak işlev veya kod bloğu, ikincisi ise çalıştırılma süresidir (milisaniye cinsinden).

Aşağıdaki kod parçasından bu anahtar kelimelerin kullanımını görebiliriz.


```
async function doubleAfterDelay(x) {  
  // Simulate a delay using a Promise  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve(x * 2); // Calculate and resolve the promise with the doubled value  
    }, x * 1000); // Delay in milliseconds (x seconds)  
  });  
}  
  
// How to use the function:  
async function demo() {  
  console.log("request submitted, calculating...");  
  let result = await doubleAfterDelay(2); // Wait for 5 seconds  
  console.log(result); // Output: 10  
}  
  
demo();
```

Şekil 14: await ve setTimeout fonksiyonları

Scripting çıktısı aşağıdaki gibidir.

```
request submitted, calculating...  
10
```

Şekil 15: await ve setTimeout fonksiyonları çıktı

Arrow fonksiyonlar ile ilgili uygulamalara devam edelim. Klasik fonksiyon tanımlama yöntemi ile tasks nesnesi oluşturularak içerisinde çeşitli görevler tanımlıyoruz. Her bir görev için tamamlanma durumunu belirtiyoruz. **getTaskToDo** fonksiyonu ile tamamlanmamış görevleri ekrana yazdırıyoruz.

getTaskToDo fonksiyonu nesne içerisinde tanımlanan tasks dizisi üzerinde filtreleme işlemi yaparak tamamlanmamış görevleri ekrana yazdırır.

```
const tasks = {
  tasks :[
    {text: 'Shopping',
      completed: true},
    {text: 'Cleaning',
      completed:false},
    {text: 'Project',
      completed:false}
  ],

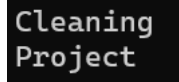
  //klasik fonksiyon tanımlama ile
  getTasksToDo: function() {
    // Tamamlanmamış görevlerin listesini almak için bir filtreleme yapacağız
    const tasksToDo = this.tasks.filter(task => !task.completed);

    // Ekrana tamamlanmamış görevleri yazdıralım
    tasksToDo.forEach(task => {
      console.log(task.text);
    });
  }
};

tasks.getTasksToDo()
```

Şekil 16: Fonksiyon

Yukarı verilen kodun çıktısı aşağıdaki gibidir. Görevler dizisinde yapılmamış görevleri ekrana yazdırır.



Cleaning
Project

Şekil 17: Uygulama Çıktısı

Aynı nesneyi bir de arrow function kullanarak kısa bir şekilde aşağıdaki gibi yazabiliriz. Nesne içerisindeki fonksiyonun kısaltıldığını görebiliriz.

```

const tasks = {
  tasks :[
    {text: 'Shopping',
      completed: true},
    {text: 'Cleaning',
      completed:false},
    {text:'Project',
      completed:false}
  ],

  //arrow fonksiyon ile
  getTasksToDo() {
    // Tamamlanmamış görevlerin listesini almak için bir filtreleme yapacağız
    const tasksToDo = this.tasks.filter((task) => {
      return !task.completed});

    // Ekrana tamamlanmamış görevleri yazdıralım
    tasksToDo.forEach(task => {
      console.log(task.text);
    });
    return tasksToDo;
  }
};

tasks.getTasksToDo()

```

Şekil 18: Arrow Function

Cleaning
Project

Şekil 19: Komut Çıktısı

Arrow function kullanarak kısalttığımız fonksiyonu tek satırda aşağıdaki gibi yazabiliriz. Atama işlemi yapmadan doğrudan return işlemi yapabiliriz.

```

const tasks = {
  tasks :[
    {text: 'Shopping',
      completed: true},
    {text: 'Cleaning',
      completed:false},
    {text:'Project',
      completed:false}
  ],

  //arrow fonksiyon ile
  getTasksToDo() {
    return this.tasks.filter((task) => !task.completed);
  }
};

console.log(tasks.getTasksToDo())

```

Şekil 20: Arrow function

Not Uygulaması

Daha önce oluşturmuş olduğumuz not uygulamasını geliştirerek, daha önce klasik yöntemle tanımladığımız fonksiyonları tek tek arrow function'lara dönüştürelim.

app.js Dosyası

App.js dosyası içerisinde '**function**' anahtar kelimesi ile tanımladığımız handler fonksiyonlarını arrow function'a dönüştürelim.

```
handler: function (argv) {  
  command.addNote(argv.title, argv.body)  
}
```

```
handler (argv){  
  command.addNote(argv.title, argv.body)  
},
```

Bu işlemi tüm handler fonksiyonlarına uygulayalım.

Notes.js Dosyası

Notes_command.js dosyası içerisindeki fonksiyonları aşağıdaki gibi düzenleyelim.

```
const duplicateNotes = notes.filter(function (note){  
  return note.title === title})
```

→

```
const duplicateNotes = notes.filter(  
  (note)=> {note.title === title })
```

```
const notesToKeep = notes.filter(function(note)  
{ return note.title !== title})
```

→

```
const notesToKeep = notes.filter((note) =>  
{note.title !== title})
```

```
const loadNotes = function(){  
  try{  
    const data_buffer = fs.readFileSync('notes.json')  
    const data_json = data_buffer.toString()  
    return JSON.parse(data_json)  
  }  
  
  catch(e){  
    return []  
  }  
}
```

→

```
const loadNotes = () => {  
  try{  
    const data_buffer = fs.readFileSync('notes.json')  
    const data_json = data_buffer.toString()  
    return JSON.parse(data_json)  
  }  
  
  catch(e){  
    return []  
  }  
}
```

```
const saveNotes = function (notes){  
  const dataJson = JSON.stringify(notes)  
  fs.writeFileSync('notes.json',dataJson)  
}
```

→

```
const saveNotes = (notes) => {  
  const dataJson = JSON.stringify(notes)  
  fs.writeFileSync('notes.json',dataJson)  
}
```

```
const remove_note = function(title){...}
```

→

```
const remove_note = (title) => {...}
```

```
const addNote = function(title,body){...}
```

→

```
const addNote = (title,body) => {...}
```

List Komutu / Fonksiyonu

Uygulamamızda henüz içeriği boş bıraktığımız list komutunu dolduralım. Bu komut bize eklemiş olduğumuz notları yazdıracak. Sırasıyla app.js ve notes_command.js dosyalarında aşağıdaki kod parçalarını ekliyoruz.

```
yargs.command({
  command: 'list',
  describe: 'list note',
  handler: function () {
    command.listNotes()
  }
})
```

app.js dosyası içeriği

```
const listNotes = () => {
  const notes = loadNotes();
  console.log(chalk.inverse('NOTES'))
  notes.forEach(element => {
    console.log(element.title)
  });
}
```

notes_command.js dosyası

Filter / Find Komutu

"Filter" ve "find" fonksiyonları JavaScript'te oldukça faydalı ve sık kullanılan fonksiyonlardır. Bu fonksiyonlar, bir dizi üzerinde belirli bir koşulu kontrol etmek ve dizi içindeki belirli öğeleri bulmak için kullanılırlar. "Filter" fonksiyonu, bir dizi içinde belirli bir koşulu sağlayan tüm öğeleri içeren yeni bir dizi döndürür. "Find" fonksiyonu, bir dizi içinde belirli bir koşulu sağlayan ilk öğeyi bulur ve bu öğeyi döndürür. Daha önce title değerini *unique* olarak ayarlamıştık. Bu yüzden title değerini ilk bulunduğu yerden sonra arama işlemini durdurması işlem performansı açısından daha faydalı olur. Find ile filter anahtar kelimelerini yer değiştiriyoruz.

```
const duplicateNotes = notes.filter(
  (note) => {note.title === title })
```

→

```
const duplicateNotes = notes.find((note) =>
  {note.title === title })
```

Read Komutu / Fonksiyonu

Uygulamamızda içeriğini boş bıraktığımız read komutunu dolduralım. Bu komut bize, girdi olarak verdiğimiz title değerine ait notun body kısmını verecek.

İlk olarak note_command.js içerisinde readNotes fonksiyonunu oluşturuyoruz. Daha sonra app.js dosyası içerisinde read komutun handler kısmını dolduruyoruz.

```

yargs.command({
  command: 'read',
  describe: 'read note',
  //builder alınan argümanları içerir.
  builder: {
    title: {
      describe: 'Note Title',
      demandOption: true,
      type: 'string',
    },
  },
  handler(argv) {
    command.readNotes(argv.title)
  }
})

```



```

const readNotes = (title) => {
  //önce dosyadan okuma işlemi
  const notes = loadNotes();
  //bu notlar arasında title uyan notu döndür
  const note = notes.find((note) => {note.title === title })
  console.log(note)
  if (note == undefined){
    console.log(chalk.red.inverse('Note not found'))
  }
  else{
    console.log(chalk.inverse(note.title));
    console.log(chalk.green(note.body));
  }
}

```

Şekil 21:app.js dosyası

Şekil 22: Note_command.js dosyası

Debug İşlemleri

Debug işlemleri için genellikle ilk olarak ekrana yazdırma işlemi tercih edilir. Bu, hataların varlığını belirlemek için kodun belirli noktalarında veri veya durumları kontrol etmeyi sağlar.

Alternatif olarak, breakpoint kullanılabilir. Breakpoint, kodun çalışmasını durdurarak belirli bir noktada inceleme yapmayı sağlar. JavaScript'te breakpoint eklemek için "**debugger**" ifadesi kullanılır. Kod içinde bu ifadeyi bulunduran bir yerde çalışma durur ve geliştirme aracı (tarayıcı konsolu veya başka bir IDE) üzerinden değişkenleri ve kod durumunu incelemek mümkün olur.

debugger

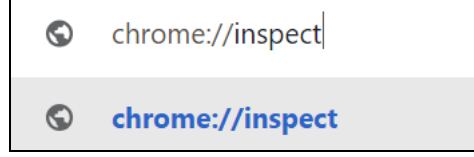
Komut satırı üzerinden breakpoint kullanılan dosyayı çalıştırmak için **node inspect app.js** komutu yazılır. "inspect" komutu, Node.js'in yerleşik hata ayıklama modunu etkinleştirir. Bu mod, uygulamanın çalışması sırasında kodunuzu duraklatmanıza, değişkenleri incelemenize ve adım adım ilerlemenize izin verir.

```

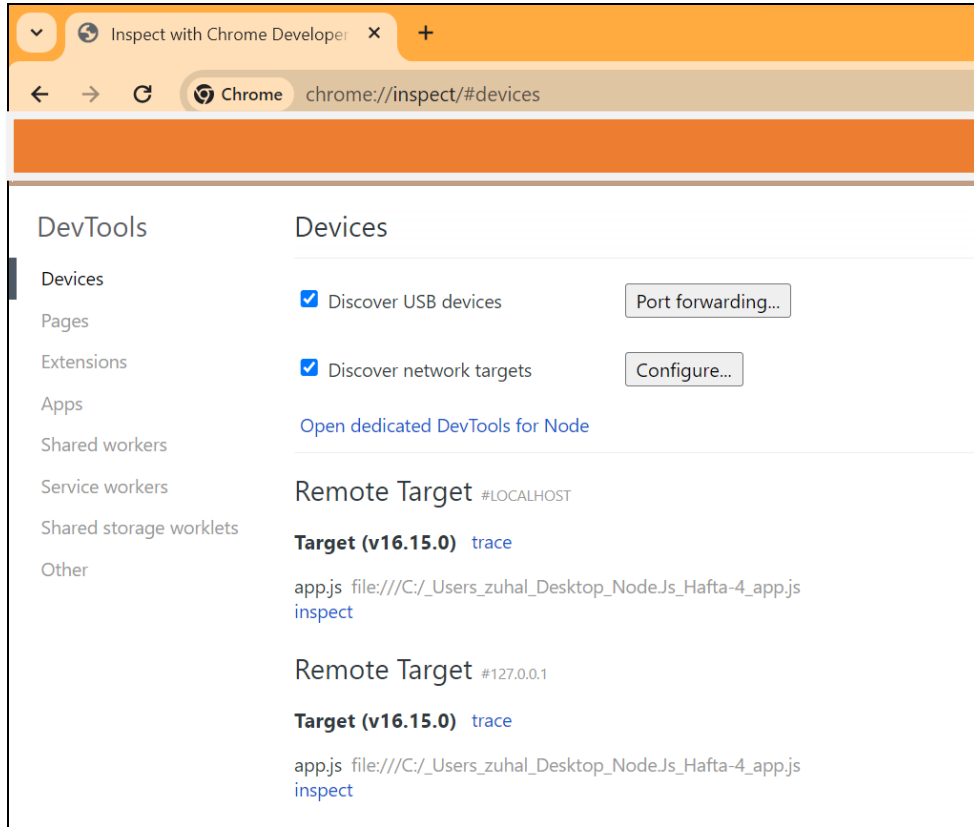
C:\Users\zuhal\Desktop\Node.Js\Hafta-4>node inspect app.js
< Debugger listening on ws://127.0.0.1:9229/f6ba0474-13ae-4c68-b29a-292c04449ff5
< For help, see: https://nodejs.org/en/docs/inspector
<
ok
< Debugger attached.
<
Break on start in app.js:2
1
> 2 const yargs = require('yargs') //import yargs
3 yargs.version('1.1.0') //adding code version
4 const command = require('./note_command.js')
debug>

```

chrome://inspect, Google Chrome'un geliştirici araçlarına erişmek için kullanılan özel bir URL'dir. Bu URL, Chrome'un yerel bilgisayarınızda çalışan ve hata ayıklama modunda olan web sayfalarını ve Node.js uygulamalarını izlemenize, incelemenize ve hata ayıklamanıza olanak tanır.



chrome://inspect adresine giderek, aşağıdaki gibi bir kullanıcı arayüzüne erişebilirsiniz:



- **Devices:** Bağlı olan Android cihazları ve emülatörleri gösterir. Bu, mobil web sitelerini ve uygulamalarını hata ayıklamanıza olanak tanır.
- **Remote Target:** Uzak cihazlarda çalışan Chrome oturumlarını gösterir. Örneğin, bir akıllı telefondaki Chrome tarayıcısını bilgisayarınızdaki Chrome geliştirici araçlarıyla hata ayıklamak istediğinizde bu bölümü kullanırsınız.

- **Inspect** : Inspect butonu, tarayıcınızın sayfa hata ayıklama moduna geçmesini sağlayan bir düğmedir. Tarayıcınızın geliştirici araçlarını açmanızı ve bir web sayfasının HTML, CSS ve JavaScript kodunu incelemenizi sağlar.

Inspect butonu seçildikten sonra açılan sayfa aşağıdaki gibidir. Bu sayfa üzerinde debug işlemi başlatılarak sonuçlar incelenebilir.

