https://github.com/rohan-paul/Awesome-JavaScript-Interviews

Javascript

Hogyan működik az események delegálása (event delegation)?

Az Event delegation a JavaScript események két funkcióját használja: az event bubbling és target element. Amikor egy esemény elindul egy elemen, például egérgombbal kattint egy gombra, ugyanaz az esemény az elem összes őse esetében is elindul. Ez a folyamat event bubbling néven ismert. Tehát más szavakkal: Ha egy eseményt egy elemről lőnek ki, akkor az esemény a szülő csomópontjaig buborékolódik fel.

Az esemény az eredeti elemtől a DOM fa tetejéig buborékol fel. Bármely esemény céleleme (target element) a kiinduló elem, a példánkban található gomb, és az eseményobjektum egyik tulajdonságában van tárolva. A „cél" ugyanaz marad az eseményobjektumban. Az eseménydelegáció (event delegation) segítségével lehetőség van eseménykezelő hozzáadására egy elemre, megvárni, amíg egy esemény kibuggyan egy gyermekelemből, és könnyen meghatározhatja, hogy az esemény melyik elemből származik.

A céltulajdonság (target property) használatával mindig nyomon követhetjük, hogy melyik elem okozza valójában a szülője által rögzített eseményt, és ez segíthet az eseménykezelők számának csökkentésében, mivel időnként nem kell minden elemhez hozzáadni az event listener-eket.

Mi az Event Propagation?

. Ez az általános kifejezés mind az eseménybuborékolás (event bubbling), mind az eseményelkapás (event capturing) szempontjából. Példa:

```
<ul>
    <li><a href="..."><img src="..." alt=""></a>
    <li><a href="..."><img src="..." alt=""></a>
    ...
    <li><a href="..."><img src="..." alt=""></a>
</ul>
```

A képre történő kattintás nem csak a megfelelő IMG elemhez, hanem az A szülőhöz, a LI nagyapához és így tovább is generál kattintási eseményt, végig megy az elem minden elődjén, mielőtt abbahagyná az ablakobjektumot .

A DOM terminológiájában a kép az esemény célpontja (event target), a legbelső elem, amely felett a kattintás származott. Az event target és az ősei a szülőtől az ablakobjektumig elágazást képeznek a DOM-fában. Például a képgalériában ez az ág a csomópontokból áll: IMG, A, LI, UL, BODY, HTML, document, window.

Vegye figyelembe, hogy az ablak valójában nem DOM-csomópont, hanem megvalósítja az EventTarget interface-t, ezért az egyszerűség kedvéért úgy kezeljük, mintha a dokumentumobjektum szülőcsomópontja lenne.

A branch azért fontos, mert ez az az út, amelyen az események tovább haladnak (vagy áramlanak). Ez a terjedés az összes esemény meghívásának folyamata, amely az adott eseménytípushoz kapcsolódik, az ág elágazásaihoz csatlakozva. Minden listener-t egy eseményobjektummal hívunk meg, amely az esemény szempontjából releváns információkat gyűjt

Ne feledje, hogy egy csomóponton több listener is regisztrálható ugyanarra az eseménytípusra. Amikor a terjedés eléri az egyik ilyen csomópontot, a listener-eket regisztrálásuk sorrendjében hívják meg.

Azt is meg kell jegyezni, hogy a branch meghatározás statikus, vagyis az esemény kezdeti elküldésénél jön létre. Az esemény feldolgozása során bekövetkező famódosításokat a rendszer figyelmen kívül hagyja.

A terjedés kétirányú, az ablaktól az esemény célpontjáig (event target) és vissza. Ez a terjedés három szakaszra osztható:

- Az ablakból az event target parent: ez az elkapási (capture) szakasz
- Az eseménycél (event target) maga: ez a célfázis
- Az event target parent-től az ablakig: a buborékfázis (bubble phase)

Hogyan segít nekem?

Képzeljen el egy HTML oszlopot 10 oszloppal és 100 sorral, amelyben azt szeretné, ha valami történne, amikor a felhasználó egy táblázat cellájára kattint. Például egyszer egy ilyen méretű tábla minden celláját szerkeszthetővé kellett tennem, amikor rákattintottak. Az eseménykezelők hozzáadása az 1000 cellához jelentős teljesítményproblémát jelentene, és potenciálisan a böngésző által összeomló memóriaszivárgások forrása. Ehelyett az eseménydelegáció használatával csak egy eseménykezelőt adhat hozzá a táblázat eleméhez, elfogja a kattintási eseményt és meghatározza, hogy melyik cellára kattintottak.

Hogyan működik a this a javascript-ben?

„this” a hívó objektumra vonatkozik (szülő objektum)

Amikor egy objektum metódusát hívják meg, ez arra az objektumra vonatkozik, amely tartalmazza a meghívott metódust.

```
function foo() {
  console.log("Simple function call")
  console.log(this === global)
}

let user = {
  count: 10,
  foo: foo,
  foo1: function() {
    console.log(this === global)
```

```
  },
}

user.foo() // Prints false because now “this” refers to user object instead of global object.
let fun1 = user.foo1
fun1() // Prints true as this method is invoked as a simple function.
user.foo1() // Prints false on console as foo1 is invoked as a object’s method, and the 'this' refers
to the containing object NOT 'window' or 'global'
```

De ha a következőt teszem a user.foo () helyett

foo ()

Aztán kiírja az „igaz” szót - Mivel az egyszerű foo függvény a globális végrehajtási kontextusban van, ezért az „this” globális

De akkor gondold meg, ha a foo () funkcióm strict módban van

```
function foo() {
  "use strict"
  console.log("Simple function call")
  console.log(this === global)
}
```

Ekkor a foo () hamis értéket ad nekem, mivel alapértelmezettként ez a strict módban 'undefined'

Most ismét nézze meg a kód alábbi részét

```
fun1() // Prints true as this method is invoked as a simple function.
user.foo1() // Prints false on console as foo1 is invoked as a object’s method,
```

A foo1 függvénydefiníciója megegyezik, de amikor egyszerű függvényhívásként hívják meg, akkor ez globális objektumra vonatkozik. És amikor ugyanazt a meghatározást hívják meg az objektum módszereként, akkor ez a szülő objektumra vonatkozik. Tehát ennek értéke attól függ, hogy egy módszert hogyan hívnak meg.

this (más néven "a kontextus") egy speciális kulcsszó az egyes függvényekben, és értéke csak attól függ, hogy a függvényt hogyan hívták meg, és nem attól, hogy hogyan / mikor / hol definiálták. A lexikális hatókörök nem befolyásolják, mint más változók (a nyílfüggvények kivételével lásd alább). Íme néhány példa:

```
function foo() {
  console.log(this)
}

// normal function call
foo() // `this` will refer to `window` or global in node environment

// as object method
var obj = { bar: foo }
obj.bar() // `this` will refer to `obj`

// as constructor function
new foo() // `this` will refer to an object that inherits from `foo.prototype`
```

```
// Case-1 - WITHOUT STRICT MODE - Since the following code is not in strict mode, and because the
value of this is not set by the call, this will default to the global object, which is window in a
browser, or global in node environment.

function f1() {
  return this === global
}

console.log(f1()) // => true

// In a browser:
f1() === window // true

// In Node:
f1() === global // true

// Case-2 - In strict mode, however, the value of this remains at whatever it was set to when entering
the execution context, so, in the following case, this will default to undefined. So, in strict mode,
if this was not defined by the execution context, it remains undefined. And also, f2 was called
directly and not as a method or property of an object (e.g. window.f2())

function f2() {
  "use strict"
  return this === undefined
}

console.log(f2()) // true


Now the case when “this” Refers to a New Instance - HERE, `this` will refer to an object that inherits
from `instance.prototype` . Meaning with exact JS syntax
When a function is invoked with the new keyword, then the function is known as a constructor function
and returns a new instance. In such cases, the value of this refers to a newly created instance.
For example: */

function Person(first, last) {
  this.first = first
  this.last = last

  this.displayName = function() {
    console.log(`Full name : ${this.first} ${this.last}`)
  }
}

// Note in above, I can not use arrow function as I can not create a constructor function with arrow
syntax

let person1 = new Person("John", "Reed")
let person2 = new Person("Rohan", "Paul")

person1.displayName() // Full name : John Reed
person2.displayName() // Full name : Rohan Paul

In the case of person1.displayName, this refers to a new instance of person1, and in case of
person2.displayName(), this refers to person2 (which is a different instance than Person).

// NOW AS A SIDE POINT - To pass the value of this from one context to another, use call(), or
apply():

// An object can be passed as the first argument to call or apply and this will be bound to it.
var obj = { a: "Custom" }

// This property is set on the global object
var a = "Global"

function whatsThis() {
  return this.a // The value of this is dependent on how the function is called
}

whatsThis() // 'Global'
whatsThis.call(obj) // 'Custom'
whatsThis.apply(obj) // 'Custom'
```

```
/* Where a function uses the 'this' keyword in its body, its value can be bound to a particular object
in the call using the call() or apply() methods which all functions inherit from Function.prototype.
*/

function add(c, d) {
  return this.a + this.b + c + d
}

var o = { a: 1, b: 3 }

// The first parameter is the object to use as
// 'this', subsequent parameters are passed as
// arguments in the function call
add.call(o, 5, 7) // 16

// The first parameter is the object to use as
// 'this', the second is an array whose
// members are used as the arguments in the function call
add.apply(o, [10, 20]) // 34

// ********************** EXPLANATION - 2 ************************************//
global.a = 2

function foo() {
  return this.a
}

console.log(foo()) // => 2

// SECOND TEST CASE

var b = 2

function foo1() {
  console.log(this.b)
}

foo1() // => 'undefined'

/* THIRD TEST CASE - “this” refers to invoker object (parent object).
When an Object’s method is invoked then “this” refers to the object which contains the method being
invoked. */

var obj = {
  c: 3,
  foo3: foo3,
}

function foo3() {
  console.log(this.c)
}
obj.foo3() // => 3

// FOURTH TEST CASE
function foo4() {
  console.log(this === global)
}
foo4() // => true

// FIFTH TEST CASE - If strict mode is enabled for any function then the value of “this” will be
“undefined” as in strict mode, global object refers to undefined in place of windows object.

function foo5() {
  "use strict"
  console.log(this === global)
}
foo5() // => false
```

Magyarázza el, hogyan működik a prototípusos öröklés

A PROTOTÍPUS FONTOSABB PONTJA, mint a memória szivárgás elleni előnye - A metódus delegálás megőrzi a memória erőforrásait, mert az egyes metódusokból csak egy példányra van szükség az összes példány megosztására. Tehát a rendszeres függvénykonstruktor módszer szerint egy Person függvény minden új példánya megkapja a Person függvénnyel definiált metódus másolatát. De prototípus esetén ezt a módszert megosztják a Person összes példánya között.

Minden JavaScript objektumnak van egy [[Prototype]] nevű tulajdonsága. Ha egy objektumot keres az obj.propName vagy az obj ['propName'] útján, és az objektumnak nincs ilyen tulajdonsága - amelyet az obj.hasOwnProperty ('propName')-vel ellenőrizhetünk -, akkor a futás közben megkeresi az objektumban lévő tulajdonságot helyette a [[Prototype]] hivatkozik. Ha a prototípus-objektumnak sincs ilyen tulajdonsága, akkor annak prototípusát sorra ellenőrzik, és így járják az eredeti objektum prototípus-láncát, amíg egyezést nem találnak, vagy el nem érik a végét. Ha új objektumot hoz létre new Func () segítségével, akkor az objektum [[Prototype]] tulajdonságát a Func.prototype által hivatkozott objektumra állítja be.
Amikor megpróbál elérni egy tulajdonságot az új objektumon, akkor először az objektum saját tulajdonságait ellenőrzi. Ha nem találja ott, akkor ellenőrzi a [[Prototype]] -t, és így tovább a prototípus láncolatán, amíg vissza nem tér az Object.prototype-hoz, amely a legtöbb objektum gyökér delegáltja.

Minden objektumnak lehet egy másik objektuma mint prototípusa. Ezután az előbbi objektum örökli prototípusának összes tulajdonságát. Az objektum a belső tulajdonságon keresztül [[Prototype]] adja meg a prototípusát. A [[Prototype]] tulajdonsággal összekapcsolt objektumok láncát prototípus láncnak nevezzük:

A proto az Object.prototype objektum hozzáférési tulajdonsága. Kiteszi egy objektum belső prototípus-összekapcsolását ([[Prototype]]), amelyen keresztül elérhető.

```
function Foo(name) {
  this.name = name;
}
var b = new Foo('b');
var a = new Foo('a');
b.say = function() {
  console.log('Hi from ' + this.whoAmI());
}
console.log(a.__proto__ === Foo.prototype); // true
console.log(a.__proto__ === b.__proto__); // true
```

A Prototype alapú függvénydefiníció egyszerű megvalósítása

```
var Person = function(name)  {
    this.name = name;
```

```
}

Person.prototype.getName = function() {
    return this.name;
}

var john = new Person("John")

john.getName() // => "John"

Person.prototype.sayName = function() {
    return (`Hello, my name is ${this.getName()}`);
}
```

A klasszikus öröklés és a prototípusos öröklés ugyanaz, csak stíluspreferencia?
NEM.

A klasszikus és a prototípusos öröklés alapvetően és szemantikailag megkülönböztethető.

Az osztály öröklődésében a példányok egy tervrajzból (az osztály) örökölnek, és alosztálybeli kapcsolatokat hoznak létre. Más szóval, nem használhatja az osztályt, mint egy példányt. Magában az osztálydefinícióban nem hivatkozhat a példány metódusokra. Először létre kell hoznia egy példányt, majd metódusokat kell meghívnia az adott példányon. a létrehozandó objektum leírása. Az osztályok öröklődnek az osztályoktól, és alosztálykapcsolatokat hoznak létre: hierarchikus osztály taxonómiák.

A prototípusos öröklésben a példányok más példányokból származnak. A delegált prototípus egy olyan objektum, amely egy másik objektum alapjául szolgál. Amikor egy megbízott prototípusától örököl, az új objektum hivatkozást kap a prototípusra. A prototípus egy működő objektumpéldány. Az objektumok közvetlenül más objektumoktól öröklődnek. ”

A delegált prototípusok használatával (egy példány prototípusának beállítása egy példamutató objektumra való hivatkozásként) szó szerint az Objects Linking to Other Objects, vagy az OLOO, ahogy Kyle Simpson nevezi.

Delegált prototípusokat csatolhat az Object.create () (egy ES5 szolgáltatás) segítségével:

```
let animal = {
  animalType: 'animal',

  describe () {
    return `An ${this.animalType}, with ${this.furColor} fur,
      ${this.legs} legs, and a ${this.tail} tail.`;
  }
};

let mouse = Object.assign(Object.create(animal), {
  animalType: 'mouse',
  furColor: 'brown',
  legs: 4,
  tail: 'long, skinny'
```

```
});
```

Bontjuk le ezt egy kicsit. az állat egy delegált prototípus. egér egy példány. Amikor megpróbál elérni egy egéren lévő tulajdonságot, amely nincs ott, a JavaScript futásideje megkeresi az állatot (a delegáltat). Az Object.assign () egy új ES6 szolgáltatás, amelyet átad egy célobjektumnak, és annyi forrásobjektumot, amennyit csak akar, vesszővel elválasztva. Másolja az összes megszámlálható saját tulajdonságot hozzárendeléssel a forrásobjektumoktól a célobjektumokhoz, amelyek elsőbbséget élveznek. Ha bármilyen tulajdonságnév ütközik, akkor az utolsó objektumban megadott verzió nyer.

Az Object.create () egy olyan ES5 funkció, amellyel a delegált prototípusokat csatolhassuk konstruktorok és a new kulcsszó használata nélkül.

A new azt jelenti, hogy a kód a klasszikus öröklést használja?
Nem.
Az new kulcsszó egy konstruktor meghívására szolgál. Ami valójában az:

- Hozzon létre egy új példányt
- Kösse (bind) ezt az új példányba

Az alábbiakban egy példa arra is, hogy mire utal a this: amikor egy konstruktor függvényt definiálok new kulcsszóval. ITT ez egy olyan objektumra fog vonatkozni, amely örököl az instance.prototype tól

```
function Person(first, last) {
  this.first = first
  this.last = last

  this.displayName = function() {
    console.log(`Full name : ${this.first} ${this.last}`)
  }
}

// Note in above, I can not use arrow function as I can not create a constructor function with arrow
syntax

let person1 = new Person("John", "Reed")
let person2 = new Person("Rohan", "Paul")

person1.displayName() // Full name : John Reed
person2.displayName() // Full name : Rohan Paul



console.log(person1.prototype) // undefined

Object.getPrototypeOf(person1) // Person {}

person1.__proto__ // Person {}
```

Másik példa:

```
function Student() {
  this.name = "John"
  this.gender = "M"
}

var studObj = new Student()

console.log(Student.prototype) // Student {}
console.log(studObj.prototype) // undefined
console.log(studObj.__proto__) // Student {}
console.log(Object.getPrototypeOf(studObj)) // Student {}
```

```
console.log(typeof Student.prototype) // object
console.log(typeof studObj.__proto__) // object

console.log(Student.prototype === studObj.__proto__) // true
```

Öröklés-OOP-osztály-vs-prototípusok-példa

ES5 way

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}

Animal.prototype.eat = function (amount) {
  console.log(`${this.name} is eating.`)
  this.energy += amount
}

Animal.prototype.sleep = function (length) {
  console.log(`${this.name} is sleeping.`)
  this.energy += length
}

Animal.prototype.play = function (length) {
  console.log(`${this.name} is playing.`)
  this.energy -= length
}

const leo = new Animal('Leo', 7)
```

ES6 way

```
class Animal {
  constructor(name, energy) {
    this.name = name
    this.energy = energy
  }
  eat(amount) {
    console.log(`${this.name} is eating.`)
    this.energy += amount
  }
  sleep() {
    console.log(`${this.name} is sleeping.`)
    this.energy += length
  }
  play() {
    console.log(`${this.name} is playing.`)
    this.energy -= length
  }
}

const leo = new Animal('Leo', 7)
```

Now for the sub-class Dog, in ES5, here's what we will have.

```
function Dog (name, energy, breed) {
  Animal.call(this, name, energy)

  this.breed = breed
}

Dog.prototype = Object.create(Animal.prototype)
// In the above I could have also used the below format
// Dog.prototype = new Animal()

Dog.prototype.bark = function () {
  console.log('Woof Woof!')
```

```
  this.energy -= .1
}

Dog.prototype.constructor = Dog
```

Let's refactor Dog to use an ES6 class

```
class Dog extends Animal {
  constructor(name, energy, breed) {
    super(name, energy) // calls Animal's constructor

    this.breed = breed
  }
  bark() {
    console.log('Woof Woof!')
    this.energy -= .1
  }
}
```

Fontos különbség a függvénydeklarációk és az osztálydeklarációk között az, hogy a függvénydeklarációk hoised-ek, az osztálydeklarációk pedig nem. Először definiálni kell osztályát, majd hozzá kell férnie, különben az alábbiakhoz hasonló kód dob egy ReferenceError-t:

```
const p = new Rectangle(); // ReferenceError

class Rectangle {}
```

Öröklés-OOP-osztály-vs-prototípus-elmélet

Osztály öröklődés:
A példányokat általában konstruktorfüggvényeken keresztül a new kulcsszóval hozzák létre. Az osztályöröklés használhatja az ES6 class kulcsszavát.

Prototípusos öröklés:
. A példányokat általában az Object.create () segítségével állítják elő.
Most az ES6-ban, ha class és extends kulcsszavakat használunk, a JavaScript továbbra is prototípus-alapú öröklődést fog használni. Csak leegyszerűsíti a szintaxist. Talán ezért fontos megérteni, hogyan működik a prototípus-alapú öröklés. Ez továbbra is a nyelvtervezés középpontjában áll.

Öröklés-osztályokkal-super-kulcsszó-kimerítő-magyarázat

```
/* Parent Class */
class ParentClass
{
   constructor(properties){
     this.properties = properties;
     }
   getMothod(){return this.properties};
   methods(){}
}

/*Inheriting class*/
class ChildClass extends ParentClass {
    constructor(properties, classSpecificProperties) {
         super(properties);
         this.classSpecificProperties = classSpecificProperties
     }
    super.getMethod();
    classSpecificeMethods(){}
}

let childClassObject = new ChildClass(properties, classSpecificProperties);
```

```
childClassObject.methods();

childClassObject.classSpecificeMethods();
```

Másik példa:

```
/*Parent Class*/
class Vehicle {
  constructor(name, model) {
    this.name = name;
    this.model = model;
  }
  showModel() {
    console.log(this.model);
  }
  getModel() {
    return this.model;
  }
  getName() {
    return this.name;
  }
}
/*Child Class inheriting parent class*/
class FourWheelers extends Vehicle {
  constructor(name, model, noOfSeats) {
    super(name, model);
    this.noOfSeats = noOfSeats;
  }
  showNoOfSeats() {
    console.log(
      super.getName() + super.getModel() + "has " + this.noOfSeats + "seats"
    );
  }
}

/*Creating Child class object*/
let myCar = new FourWheelers("Audi", "R8", 5);
myCar.showNoOfSeats(); // Child Class method
myCar.showModel(); // Parent class method
```

OOP alapok

Two principles with OOP in JS are:

Object Creation Pattern (Encapsulation)

Object Reuse Pattern (Inheritance)

There are many object creation ways:

1- Ubiquitous Object literal:

```
var myOoj = {
          name: "Nikki",
          city: "New Delhi",
          loves: "so many"
};
```

2- Prototype Pattern

```
function myFun() {}
myFun.prototype.name = "Nikki";
```

```
myFun.prototype.city = "New Delhi";
var myFun1 = new myFun();
console.log(myFun1.name); //Nikki
```

3- Constructor Pattern:

```
function myFun(name, city, loves) {
          this.name = name;
          this.city = city;
          this.loves = loves;
}
var myFun1 = new myFun("Nikki", "New Delhi", "so much things");
console.log(myFun1.name); //Nikkia
```

Constructor Functions

These functions are the most conventional way to create objects that use functionality from each other using prototypal inheritance.

```
function Person(name) {
          // 1
          this.name = name;
          this.greet = function() {
                    console.log("Hello, I'm " + this.name);
          };
}
//2
var person = new Person("Jack Johnson");

//3
console.log(Object.getPrototypeOf(person)); // Person {}
```

OOP alapok 2

```
function Person() {
          //properties/fields
          this.name = "Rob Gravelle";
          this.height = 68;
          this.weight = 170;
          this.socialInsuranceNumber = "555 555 555";

          //methods/actions
          this.setHeight = function(height) {
                    this.height = height;
          };
          this.getHeight = function() {
                    return this.height;
          };
          this.setWeight = function(weight) {
                    this.weight = weight;
          };
          this.getWeight = function() {
                    return this.weight;
          };
          this.setName = function(name) {
                    this.name = name;
          };
          this.getName = function() {
                    return this.name;
          };
          this.setSocialInsuranceNumber = function(socialInsuranceNumber) {
                    this.socialInsuranceNumber = socialInsuranceNumber;
          };

          return this;
}
//instanciate the Person class
var aPerson = new Person();
```

```
var myName = aPerson.getName(); //myName now contains "Rob Gravelle"
aPerson.setName("mud"); //change the name
var myName = aPerson.getName(); //aPerson's name is now "mud"
var sinNo = aPerson.getSocialInsuranceNumber(); //will also throw an exception.  No getter implemented
for that field!



//instanciate the Person class
var aPerson = new Person();
var myName = aPerson.name; //this works
//as does this:
aPerson.name = "whatever.";
```

OOP encapsulation

Az adatok elrejtésére vagy az absztrakcióra utal, ami azt jelenti, hogy a lényeges jellemzőket a háttér részleteinek elrejtésével kell ábrázolni. Az OOP nyelvek többsége hozzáférés-módosítókat biztosít a változó hatókörének korlátozására, de ezek nem ilyen hozzáférés-módosítók a JavaScript-ben, de bizonyos módon korlátozhatják a változó hatókörét az Osztályon / Objektumon belül.

```
function person(fname, lname) {
  let firstname = fname;
  let lastname = lname;

  let getDetails_noaccess = function() {
    return `First name is: ${firstname} Last
    name is: ${lastname}`;
  };

  this.getDetails_access = function() {
    return `First name is: ${firstname}, Last
    name is: ${lastname}`;
  };
}
let person1 = new person("Mukul", "Latiyan");

console.log(person1.firstname); // => undefined

console.log(person1.getDetails_noaccess); // => undefined

console.log(person1.getDetails_access()); // => First name is: Mukul, Last    name is: Latiyan
```

OOP encapsulation 2

This is an example of "class-free" object oriented programming

```
const MyObj3 = initVal => {
          let myVal = initVal;
          return {
                    get: function() {
                              return myVal;
                    },
                    set: function(val) {
                              myVal = val;
                    }
          };
};
```

A closures lehetővé teszik az erős szerződést
A változó fogalmilag „mentve" van a függvényből visszaküldött objektumon belül a Javascript closures-nek köszönhetően. Ez azt jelenti, hogy az értéket beállíthatjuk a setter módszerrel (azaz

x.set (2)), de a mező közvetlen megváltoztatásának megkísérlése (azaz x.myVal = 2) nem tesz semmit.
Az állapot teljesen be van burkolva, és az objektumba csak a settereken és a gettereken keresztül lehet be- és kilépni.
A closures használatának egyik hátránya a teljesítmény kérdése. Bár úgy tűnik, hogy nincs különbség egy objektum létrehozásakor, a metódusok closure-okkal történő hívása körülbelül 80% -kal volt lassabb.

print-All-Prototypes-of-Objects

```
printPrototypeProperties = (obj) => {
  var result = []

  let objProp = Object.getOwnPropertyNames(obj)
  console.log(objProp)

  // console.log(Object.keys(obj))

  for (let i of objProp) {
    if (result.indexOf(i) === -1) {
      result.push(i)
    }
  }
  return result
}

let myObj = {
  a: 1,
  b: 2,
  c: 3,
}

console.log(printPrototypeProperties(myObj))
```

Prototype-Example-2

```
function ParentClass (foo) {
    this.foo = foo;
}

ParentClass.prototype = {
    bar: function () {
        console.log("Hello from ParentClass")
    }
}

function ChildClass () {
}


ChildClass.prototype = new ParentClass()

// console.log(ChildClass.prototype)

ChildClass.prototype.constructor = ChildClass;

// Shortcut to parent class
ChildClass.prototype.super = ParentClass.prototype;

ChildClass.prototype.bar = function () {
    this.super.bar.call(this)
    console.log("Hello from ChildClass")
}

var p = new ParentClass()
var c = new ChildClass()
```

```
console.log(p.bar())
console.log(c.bar())
```

prototype-func-print-array-elements

```
// SOL-1 - Not a good solution as I dont need the extra step of pushing to the resultArr
Array.prototype.print = function() {
  let resultArr = []
  this.forEach(i => resultArr.push(i))
  return console.log(resultArr.join())
}

myArr = [1, 2, 3]
myArr.print() // => 1,2,3

// SOL-2
Array.prototype.print_1 = function() {
  return this.forEach(i => console.log(i))
}

myArr.print_1()
```

Prototype-func-String-dasherize

```
// Write a function to replace a single space with a "-"

String.prototype.dasherize = function() {
    return this.replace(/\s/g, '-')
}

console.log('Hello world'.dasherize());
```

Prototypes-Prevents-Memory-Leaks

A PROTOTYPE HASZNÁLATÁNAK FONTOSABB PONTJA - megőrzi a memória erőforrásait, mert az egyes metódusokból csak egy példányra van szükség.

call-function-basics-1

A függvény definíciójában a “this” a függvény "tulajdonosára" utal.
A call () metódus segítségével meghívhatunk egy metódust, amelynek argumentuma (paraméter) egy tulajdonos objektum.

A call () segítségével az objektum egy másik objektumhoz tartozó módszert használhat.

```
var person = {
  fullName: function() {
    return this.firstName + " " + this.lastName
  },
}
var person1 = {
  firstName: "John",
  lastName: "Doe",
}
var person2 = {
  firstName: "Mary",
  lastName: "Doe",
}
person.fullName.call(person1) // Will return "John Doe"
```

call-function-basics-2

CALL (): Egyenként megadott argumentumú függvény. Ha ismeri az átadandó argumentumok számát, vagy nincs átadandó argumentum, használhatja.

APPLY (): Meghív egy tömbként megadott argumentumú függvényt. Használhatja, ha nem tudja, hány argumentum kerül átadásra a függvény számára.

Előnye van az Apply használatának a call-al szemben, nincs szükségünk az argumentumok számának megváltoztatására, csak egy átadott tömböt módosíthatunk.

A teljesítményben nincs nagy különbség. De azt mondhatjuk, hogy a call valamivel gyorsabb.

call-vs-apply-vs-bind

A call és az apply azonnal végrehajtja a függvény hívást. A bind egy függvénnyel tér vissza.
A bind egy később hívható függvény visszaadására szolgál.

```
const obj  = {
    x: 42,
    getX: function() {
        return this.x;
    }
}

const unBoundX = obj.getX
console.log(unBoundX()); // => undefined

// But to get it to work
const boundX = unBoundX.bind(obj)
console.log(boundX()); // => 42
```

when-not-to-use-arrow-function

Event Handlers

Így jó:

```
var myLink = document.getElementById("myLink")
myLink.addEventListener("mouseenter", function() {
  this.classList.toggle("highlight")
  console.log(this.classList)
})
```

Így nem jó, mert az arrow function-nál nem kötődik az elemhez, és a this az a szülő less (ami a window most itt).

```
const myLink = document.getElementById("myLink")
myLink.addEventListener("mouseenter", () => {
  this.classList.toggle("hightlight")
  console.log(this.classList)
})
```

2: Object Methods

```
const person = {
  points: 23,
  score: () => {
    return this.points++
  },
}

person.score()

console.log(person.points) // it outputs 23 irrespective of how many times i run the above block of
code instead of getting incremented by earlier call of person.score().
```

Mert pontokat próbál hozzáadni az ablakhoz! Ne feledje, hogy egy arrow function használatakor ez nem kötődik semmihez, és csak örökli a szülő hatóköréből, amely ebben az esetben az ablak.

Így viszont már jó:

```
const person = {
    points: 23,
    score: function()  {
        this.points++;
    }
}

person.score();

console.log(person.points)
```

Destructuring_Geneal

General Object Destructuring Example

```
let myObj = {
    name: "Luke",
    age: 25,
    hobbies: "music"
};

let { name, age, hobbies } = myObj;

console.log(name, age, hobbies);    // => Luke 25 music
```

General Array Destructuring Example

```
let arr = [‘Jim’, ‘Bob’, ‘Sarah’, ‘Cassie’];

let [ jim, bob, sarah, cassie ] = arr;

console.log(jim, bob, sarah, cassie); //outputs: Jim Bob Sarah Cassie
```

Az objektumokkal ellentétben az a név, amelyet a változóknak adunk, nem számít. Változtassuk meg a fenti példát: Tehát minden változó neve CSAK az általam beolvasott indexpozíciókra számít.

```
let arr = [‘Jim’, ‘Bob’, ‘Sarah’, ‘Cassie’];
```

```
let [ var1, var2, var3, var4] = arr;

console.log(var1, var2, var3, var4); //outputs: Jim Bob Sarah Cassie
```

Using Spread operator

```
let myObj = {

    name: "Luke",
    age: 25,
    hobbies: "music"

};

let { hobbies, ...rest } = myObj;  // => Luke 25 music

console.log(hobbies, rest)  // => music { name: 'Luke', age: 25 }

console.log(hobbies, rest.age)  // => music 25
```

filter-implement

```
// Problem-1 Filter even numbers

let numberArray = [1,2,3,4,5,6,7,8,9,10];

let evenNumbers = [];

for (let i = 0; i < numberArray.length; i++) {
          if (numberArray[i] % 2 === 0) {
                    evenNumbers.push(numberArray[i]);
          }
}
// console.log(evenNumbers);

let evenNumbersWithFilter = numberArray.filter((item) => (item % 2 === 0));



// Problem 2:- Filter objects with tags javascript

var persons = [
  {id : 1, name : "John", tags : "javascript"},
  {id : 2, name : "Alice", tags : "javascript"},
  {id : 3, name : "Roger", tags : "java"},
  {id : 4, name : "Adam", tags : "javascript"},
  {id : 5, name : "Alex", tags : "java"}
];

let jsTags = persons.filter((item) => (item.tags === "javascript"))



function findNonDuplicatesFilter (array) {
          return array.filter((elem, index, arr) => {
                    return array.indexOf(elem) === index;
          })
}
```

forEach-vs-map

forEach () - minden megadott tömbelemhez egyszer végrehajt egy megadott függvényt.

map () - új tömböt hoz létre a kapott függvény meghívásának eredményeivel a hívó tömb minden elemén.

Nos, a forEach () metódus valójában nem ad semmit (undefined). Egyszerűen meghív egy megadott függvényt a tömb minden elemén.
Eközben a map () metódus is egy függvényt is meghív a tömb minden elemére. A különbség az, hogy a map () ad visszatérési értékeket, és valójában egy új, azonos méretű tömböt ad vissza.

Visszatérési érték
- forEach undefined-t ad vissza
- map: új tömböt ad vissza

```
const array1 = [1, 2, 3]

const func = arr => {
  const result = arr.forEach(i => i * 2)
  return result
}
func() // Undefined
```

Mutáció
A forEach () befolyásolja és megváltoztatja az eredeti tömböt, míg a map () egy teljesen új tömböt ad vissza - így az eredeti tömb változatlan marad.

closure

A closure függvények kombinációja a bezáró hivatkozásával. Más szavakkal, a closure hozzáférést biztosít egy külső függvény hatóköréhez egy belső függvényből. A JavaScript-ben a függvény létrehozásakor minden alkalommal létrejönnek closure-ok.

```
function init() {
  var name = "Mozilla" // name is a local variable created by init
  function displayName() {
    // displayName() is the inner function, a closure
    alert(name) // use variable declared in the parent function
  }
  displayName()
}
init()
```

Mivel azonban a belső függvények hozzáférnek a külső függvények változóihoz, a displayName () hozzáférhet a szülőfüggvényben deklarált változó nevéhez.

A JavaScript-ben a closure-ok lehetővé teszik az adatvédelmet. Amikor closure-okat használ az adatvédelem érdekében, a mellékelt változók csak a tartalmazó (külső) függvényben találhatók. Az adatok kívülről nem érhetők el, kivéve az objektum privilegizált módszereit. A JavaScript-ben minden, a closure körében meghatározott függvény kiváltságos.

```
const outerFunc = () => {
  let name = "Rohan"

  const closureFunc = () => {
    console.log(name)
  }
  return closureFunc()
}

var name = "Paul"
```

```
outerFunc() // => Will Print 'Rohan'
```

custom_Callback

```
//Example -1 super simple example of custom callback. done() is a callback which I am defininig
seperately.
done = () => {
  console.log("Done from Callback");
};

printNum = (num, callback) => {
  for (let i = 0; i <= num; i++) {
    console.log(i);
  }
  if (callback && typeof callback === "function") {
    callback();
  }
};

printNum(5, done);
```

custom_Callback-2

```
/* Example - 1 Define our function with the callback argument that generates a random number between
arg1 and arg2 */

printRandom = (max, min, callback) => {

          let randNum = Math.floor(Math.random() * (max - min) + min)    ;

          if (callback && typeof(callback) === 'function' ) {
                    callback(randNum);
          }
}

printRandom(15, 5, (num) => {
          console.log(num);
})
```

what-is-type-coercion

A típuskényszer az, amikor bizonyos adatokat vagy objektumokat más típusú objektumokká kényszerítenek.

```
// returns true
1 == true;
```

Példa:

```
var a = "hello";
var b = 99;

// logs the string 'hello99'
// in this case the number 99 is coerced into a string
console.log(a + b);

// another examples
var object = "apple";

// object is coerced into true, so we log a statement
if (object) {
  console.log("I have an " + object);
}
```

Ezzel bajba is kerülhet. Tekintsük a következőt.

```
// I have an array, I may not know how long it is
var a = [0, 1, 2, 3];

// I want to say if there is something at index 0 of the array a then do something
// But a[0] is actually the number 0 when coerced is false
// This if statement does nothing
if (a[0]) {
  // do something
}
```

Ha a[0] az 0, akkor nem fog belemenni az if-be.

coercion

```
// Implicit
console.log("20" + 18);                  // Logs: 2018
// The number 18 is implicitly coerced into a string so that it can be concatenated onto the string
“20”.

console.log("20" * 18);                  // Logs: 360
//  The string “20” is implicitly coerced into a number so that it can be multiplied by the number 18.

console.log(20 + true);                  // Logs: 21
// The boolean true is implicitly coerced into a number (1) so that it can be added to the number 20.

console.log("20" == 20);                 // Logs: true
// The string “20” is implicitly coerced into a number so that it can be tested by loose equality
against the number 20.

console.log("20" === 20);                // Logs: false
// No coercion takes place when the strict equality operator is used.

// Explicit
console.log(Number("20") + 18);          // Logs: 38
// The string “20” is explicitly coerced into a number and added to the number 18 (giving us a
different result than the implicit coercion version of this expression on line 2).

console.log(String(20) + String(true));   // Logs: "20true"
// The number 20 and the boolean true are explicitly coerced into strings and concatenated (giving us
a different result than the implicit coercion version of this expression on line 4.)
```

spread-operator-vs-rest-parameters

Rest parameter:

```
function add(...args) {
  let result = 0

  for (let arg of args) result += arg

  return result
}

add(1) // returns 1
add(1, 2) // returns 3
add(1, 2, 3, 4, 5) // returns 15



function xyz(x, y, ...z) {
  console.log(x, " ", y) // hey hello

  console.log(z) // ["wassup", "goodmorning", "hi", "howdy"]
```

```
  console.log(z[0]) // wassup
  console.log(z.length) // 4
}

xyz("hey", "hello", "wassup", "goodmorning", "hi", "howdy")
```

Spread operator:

```
const arr = ["Joy", "Wangari", "Warugu"]
const newArr = ["joykare", ...arr]

const arr = [1, 2, 3]
const arr2 = [...arr]
```

```
const obj1 = { a: 10 }
const obj2 = { b: 20 }
const obj3 = { c: 30 }

// ES2018
console.log({ ...obj1, ...obj2, ...obj3 }) // → {a: 10, b: 20, c: 30}

// ES2015
console.log(Object.assign({}, obj1, obj2, obj3)) // → {a: 10, b: 20, c: 30}
```

call-stack-good-example

Mi a call stack?

A legalapvetőbb szinten a hívásverem (call stack) olyan adatstruktúra, amely a Last In, First Out (LIFO) elvet használja a függvényhívás (hívás) ideiglenes tárolására és kezelésére.

A végrehajtási kontextus (execution context) egy olyan fogalom a nyelvi specifikációban, amely - laikus kifejezéssel élve - nagyjából megegyezik azzal a „környezettel”, amelyben a függvény végrehajtja; azaz változó hatókör (és a hatókör lánc, változók a külső hatókörektől bezárva), függvény argumentumok és az objektum értéke.

Amikor a kódot JavaScript-ben futtatják, a végrehajtás környezete nagyon fontos, és az alábbiak egyikeként értékelik:

Globális kód - Az az alapértelmezett környezet, ahol a kódot először hajtják végre. Függvénykód - Valahányszor a végrehajtás folyamata belép egy függvénytestbe. Eval kód - A belső eval függvényen belül végrehajtandó szöveg.

LIFO: Amikor azt mondjuk, hogy a hívásverem (call stack) a Last In, First Out adatstruktúra elvével működik, ez azt jelenti, hogy az utolsó függvény, amely a verembe kerül, elsőként jelenik meg, amikor a függvény visszatér.

Vessen egy pillantást egy kódmintára a LIFO bemutatásához, verem nyomkövetési hibát nyomtatva a konzolra.

```
function firstFunction(){
    throw new Error('Stack Trace Error');
}
function secondFunction(){
    firstFunction();
}
function thirdFunction(){
    secondFunction();
}

thirdFunction();
```

Szándékosan helyeztem el az firstFunction () hibát, így láthatjuk a függvény végrehajtásának teljes híváskötegét. Egyébként, ha csak visszaküldök egy console.log-ot abból a függvényből, akkor rendesen kijelentkezik.

Vegye figyelembe a függvények elrendezését, amikor a verem az firstFunction () paranccsal kezdődik (ez az utolsó függvény, amely bekerült a verembe, de az első végrehajtásra került. És kiugrik, hogy eldobja a hibát. Ezután következik a secondFunction (), majd a thirdFunction () (amely az első olyan függvény, amely a kód végrehajtásakor a verembe kerül).

Ideiglenes tárolás: Ha egy függvényt meghívnak (meghívják), akkor a függvényt, annak paramétereit és változóit a hívásverembe tolják, hogy veremkeretet képezzenek. Ez a veremkeret (stack frame) egy memóriahely a veremben. A memória törlődik, amikor a függvény visszatér, amikor kiugrik a veremből.

Funkcióhívás (hívás) kezelése: A hívásverem (call stack) nyilvántartást vezet az egyes veremkeretek helyzetéről. Tudja a következő végrehajtandó funkciót (és a végrehajtás után eltávolítja). Ez teszi a kód futtatását a JavaScript-ben szinkronossá. Most nézze meg az alábbi kódot

```
function firstFunction() {
    console.log("hello from firstFunction")
}

function secondFunction() {
    firstFunction();
    console.log("The end from secondFunction");
}

function thirdFunction() {
    secondFunction();
    console.log("The end from thirdFunction");
}

thirdFunction()



OUPUT
firstFunction
The end from secondFunction
The end from thirdFunction
```

A kód futtatásakor ez történik:

Amikor a thirdFunction () végrehajtásra kerül, egy üres veremkeret jön létre. Ez a program fő (névtelen) belépési pontja.
A thirdFunction () ekkor meghívja a secondFunction () függvényt, amelyet a verembe tolnak.
A secondFunction () ekkor meghívja az firstFunction () függvényt, amelyet a verembe tolnak.
A firstFunction () visszatér és kinyomtatja a „Hello from firstFunction” szót a konzolra.
Az firstFunction () kiugrik a veremből.
Ezután a végrehajtási sorrend a secondFunction () elemre lép.
A secondFunction () visszatér és kinyomtatja a „ End from secondFunction” parancsot a konzolra.
Ezután a végrehajtási sorrend lépjen a thirdFunction () elemre.
A thirdFunction () visszaadja és kinyomtatja a konzolra az „End from thirdFunction” végét.

Mi okozza a verem túlcsordulását (stack overflow)?

Verem túlcsordulás akkor fordul elő, ha van rekurzív függvény (egy önmagát hívó függvény) kilépési pont nélkül. A böngésző (tárhelykörnyezet) rendelkezik egy maximális veremhívással, amelyet képes elhelyezni, mielőtt veremhibát dobna.

```
function callMyself(){
  callMyself();
}
callMyself();
```

const-var-let

let vs var
a var függvény hatóköre, és ha megpróbál egy változót használni a tényleges deklaráció előtt, akkor csak undefined lesz. Az const és az let blokk hatókörűek, és ha a let vagy az const segítségével deklarált változót próbáljuk használni a deklaráció előtt, akkor a "ReferenceError variable is not defined " hibát kapunk.

```
function discountPrices(prices, discount) {
  var discounted = []

  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}
```

```
function discountPrices(prices, discount) {
  let discounted = []

  for (let i = 0; i < prices.length; i++) {
    let discountedPrice = prices[i] * (1 - discount)
    let finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // ReferenceError: i is not defined
  console.log(discountedPrice)
  console.log(finalPrice)

  return discounted
}

discountPrices([100, 200, 300], 0.5) // ReferenceError: i is not defined
```

const - Az konstansokban a mutáció megengedett, de az újra értékadás nem megengedett.

```
const person = { name: 'Kim Kardashian' }

person.name = 'Kim Kardashian West' // 

person = {} //  Assignment to constant variable
```

Különbség az Object.freeze () és a const között

a const és az Object.freeze két teljesen különböző dolog.

A const a kötésekre ("változók") vonatkozik. Változhatatlan kötést (bind) hoz létre, azaz nem rendelhet új értéket a kötéshez.

Az Object.freeze az értékeken, pontosabban az objektumértékeken működik. Az Object.freeze () egy olyan módszer, amely elfogad egy objektumot, és ugyanazt az objektumot adja vissza. Most az objektum egyetlen tulajdonságát sem távolíthatja el, sem új tulajdonságokat nem adhat hozzá. Ez egy objektumot megváltoztathatatlanná tesz, vagyis nem változtathatja meg annak tulajdonságait.

```
const obj = {
  name: "rohan",
}

obj.name = "paul"

console.log(obj)
```

curried-function

Van-e olyan technika, amellyel az N argumentumú függvényhívásokat N függvényhívások láncolatává konvertálják, egyetlen argumentummal az egyes függvényhívásokhoz?
A curry-zés mindig egy másik függvényt ad vissza, csak egy argumentummal, amíg az összes argumentumot nem alkalmazzák. Tehát csak addig hívjuk a visszatérő függvényt, amíg ki nem merítjük az összes argumentumot, és a végső érték vissza nem kerül.

Example

```
const add = (x, y) => x + y
add(2, 3) // => 5
```

And now, below is the same function in curried form.

```
const add = x => y => x + y
```

Same code without arrow syntax

```
const add = function(x) {
  return function(y) {
    return x + y
  }
}
```

add(2)(3) // returns 5

```
const three = a => b => c => a + b + c

const four = a => b => c => d => a + b + c + d

three(1)(2)(3) // 6

four(1)(2)(3)(4) // 10
```

hashing-vs-encrypting

Mi a különbség a titkosítás és a hash között?

A TLDR válasza az, hogy a titkosítás kétirányú, a hashelés pedig csak egyirányú.

Mindkettő módszer az adatok biztonságos továbbítására, de van némi különbség.

A titkosítás kétirányú
Amikor azt mondjuk, hogy a titkosítás kétirányú, az azt jelenti, hogy valamit visszafejteni lehet a titkosítás után. Ez a titkosítás teljes célja. Az ötlet az, hogy titkosított információkat továbbít, és a vevő ezt követően visszafejtheti és elolvashatja a tartalmát.

A hash az egyik módja, nem vonható vissza
A hash között az áll, hogy nem lehet visszavonni. Ez egyirányú művelet, és nincs mód a hash-tól az eredeti tartalom felé haladni. Bár ez biztosabbnak tűnhet, nyilvánvalóan költségekkel jár.

Ha a hash-elt adatokat átadja, a címzettnek nincs módja a hash-elt adatokról az eredeti adatokra való visszatérésre. Ez azt jelenti, hogy a használati eset egy kicsit korlátozottabb. Mivel a hash-elés folyamata egyedi (adott egy bemenet, mindig ugyanazt a kimenetet adja, és nem ugyanazt a kimenetet állítja elő két különböző bemenethez), ez azt jelenti, hogy összehasonlíthatjuk, hogy valóban ugyanazt a bemenetet adják-e be később egy későbbi időpontban (például jelszó benyújtása?), de nem tudjuk azonosítani a jelszót.

undefined-vs-not_defined

Mi a különbség a Javascriptben a "undefined" és a "not defined" között?
not defined (Ez egy hiba, amely jelzi a kódolónak / programozónak / felhasználónak, hogy a változó nem létezik a hatókörben. Nem deklarálható a hatókörben.)
Azok a változók, amelyek valójában „not defined”, vagyis nem léteznek, mivel egy adott név nincs kötve a jelenlegi lexikai környezetben. Egy ilyen változó elérése hibát okoz, de a typeof használata nem okoz hibát, és 'undefined' értéket ad vissza.

```
console.log(a) // => a is not defined
```

var undefined (Deklarálva, de az érték nincs hozzárendelve)
Meglévő változók, amelyekhez nem rendeltek értéket
Egy ilyen változó elérése undefined, a typeof 'undefined' pl.

```
console.log(a) // => a is undefined
var a
```

Why-eval-function-considered-dangerous

```
var greeting = "good morning"
function speak(str) {
  eval(str)
  console.log(greeting)
}
speak("var greeting = 'meow'")
```

Meow... valóban. Mivel nem definiáltak helyi üdvözlő változót. Arra számítottunk, hogy hozzáférünk a globális hatókörhöz és kiírhatjuk a „good morning” kifejezést. Ehelyett az eval új helyi változót adott be hatókörünkbe.

Tehát milyen rossz ez:

A kódot sebezhetővé teszi a rosszindulatú kódok beütése ellen
Lassítja a kód teljesítményét
Tehát veszélyes, és a legtöbb esetben el kell kerülni. Van néhány egyedi forgatókönyv, amikor eválra van szükség, de az eval 99% -ához nem szükséges.

use-strict-describe

a use stict meghatározza, hogy a Javascriptet szigorú módban kell végrehajtani. A szigorú mód egyik fő előnye, hogy megakadályozza a fejlesztőket a deklarálatlan változók használatában.
A legjobb gyakorlat, ha use strict-el dolgozunk

Megkönnyíti a hibakeresést: A kódhibák, amelyeket egyébként figyelmen kívül hagytak volna, most hibákat generálnak, vagy kivételeket vetnek fel, hamarabb figyelmeztetve a kód problémáira, és gyorsabban a forrásukra irányítva.
Megakadályozza a véletlen global változókat: szigorú mód nélkül, ha egy értéket hozzárendel egy deklarálatlan változóhoz, akkor automatikusan létrejön egy globális változó ezzel a névvel. Ez az egyik leggyakoribb hiba a JavaScript-ben. Szigorú módban ennek megkísérlése hibát eredményez.
Nem engedélyezi a duplikált tulajdonságneveket vagy paraméterértékeket: A szigorú mód hibát dob, ha egy objektumban megnevezett tulajdonság duplikátumot észlel (pl. Var object = {foo: "bar", foo: "baz"};) vagy megnevezett argumentumot egy függvény (pl. függvény foo (val1, val2, val1) {}), ezáltal elkapva azt, ami szinte biztos, hogy hibát jelent a kódban, amellyel egyébként rengeteg időt vesztegethetett.
Hibát dob a delete érvénytelen használatakor: A delete operátor (tulajdonságok objektumokról való eltávolítására szolgál) nem használható az objektum nem konfigurálható tulajdonságainál.

Async/Await - Understanding the fundamentals

Async - aszinkron függvényt deklarál (async function someName () {...}).

- A szokásos függvényt automatikusan átalakítja Promise-ra.
- Az aszinkron függvények lehetővé teszik az await használatát.

Await- szünetelteti az aszinkron függvények végrehajtását. (var result = await someAsyncCall();).

- Amikor a promise hívása elé tesszük, az await a kód többi részét kikényszeríti hogy megvárja, amíg a Promise befejeződik, és eredményt ad vissza.
- Ez szünetelteti az aszinkron függvényt, és a továbblépés előtt megvárja aamíg a Promise resolved nem lesz.
- Az Await csak az Promise-al működik, callback-el nem.
- Az await csak aszinkron funkciókban használható.

Szabályok
Az await blokkolja a kód végrehajtását az async függvényen belül, amelynek része (azaz az 'await' utasítás).
Egy aszinkron függvényen belül több await utasítás is lehet.
Az async várakozás használatakor győződjön meg arról, hogy a hibakezeléshez használja a try catch parancsot.
Ha a kódom blokkoló kódot tartalmaz, akkor jobb, ha aszinkron funkcióvá teszem. Ezzel biztosítom, hogy valaki más aszinkron módon tudja használni a funkcióját.
Azáltal, hogy blokkoló kódból aszinkron függvényeket hoz létre, lehetővé teszi, hogy a felhasználó, aki hívja a függvényét, döntsön a kívánt aszinkronitás szintjéről.

```
async function f(){
```

```
    // response will evaluate as the resolved value of the promise
    const response = await rp('http://example.com/');
    console.log(response);
}

async function doSomethingAsync(){
    try {
        // This async call may fail.
        let result = await someAsyncCall();
    }
    catch(error) {
        // If it does we will catch the error here.
    }
}
```

Async function without a try/catch block.

```
async function doSomethingAsync(){
    // This async call may fail.
    let result = await someAsyncCall();
    return result;
}

// We catch the error upon calling the function.
doSomethingAsync().
    .then(successHandler)
    .catch(errorHandler);
```

async-await-1

```
getJSON = () => {
  return new Promise(resolve => {
    fetch("https://tutorialzine.com/misc/files/example.json")
      // The data from the request is available in a .then block
      // We return the result using resolve.
      .then(json => {
        resolve(json)
        console.log("Successively fetched")
      })
  })
}

getJSON()
```

```
const getJSONAsync = async () => {
  // The await keyword saves us from having to write a .then() block.
  let json = await fetch("https://tutorialzine.com/misc/files/example.json")

  return json && console.log("Fetched with async")
}

getJSONAsync() // Will fetch and then print "Fetched with async"
```

async-await-3

```
doubleAfter1Second = x => {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve( x * 2)
        }, 1000)
    })
}
```

```
doubleAfter1Second(10).then((sum) => {
    console.log(sum);
  });


addAsync = async x => {
    const a = await doubleAfter1Second(10);
    const b = await doubleAfter1Second(20);
    const c = await doubleAfter1Second(30);
    return x + a + b + c
}


addAsync(10).then(sum => {
    console.log(sum)
});
```

async-await-example-when-Promise-is-preferred

```
getABC = async () => {

  let A = await getValueA();  // getValueA takes 2 second to finish
  let B = await getValueB();  // getValueB takes 4 second to finish
  let C = await getValueC();  // getValueC takes 3 second to finish

  return A * B * C
}


getABC = async () => {

  // Promise.all() allows us to send all requests at the same time. But of course, it will give me 3
independent results, from the 3 independent function invocations. From those 3 independent results,
getting the final return value by applying reduce on them.

  let results = Promise.all([ getValueA, getValueB, getValueC ])

  return results.reduce((total, value) => total * value);
}
```

converting-callback-to-Promise-and-async-await-1

Az await csak Promise-okkal működik, nem működik callback-ekkel. Tehát az async/await kódto szeretnénk használni, akkor a callback-et Promise-ra kell konvertálni.

Callback version

```
const callbackFn = (firstName, callback) => {
  setTimeout(() => {
    if (!firstName) return callback(new Error("no first name passed in!"));

    const fullName = `${firstName} Doe`;

    return callback(fullName);
  }, 2000);
};

callbackFn("John", console.log);
callbackFn(null, console.log);
```

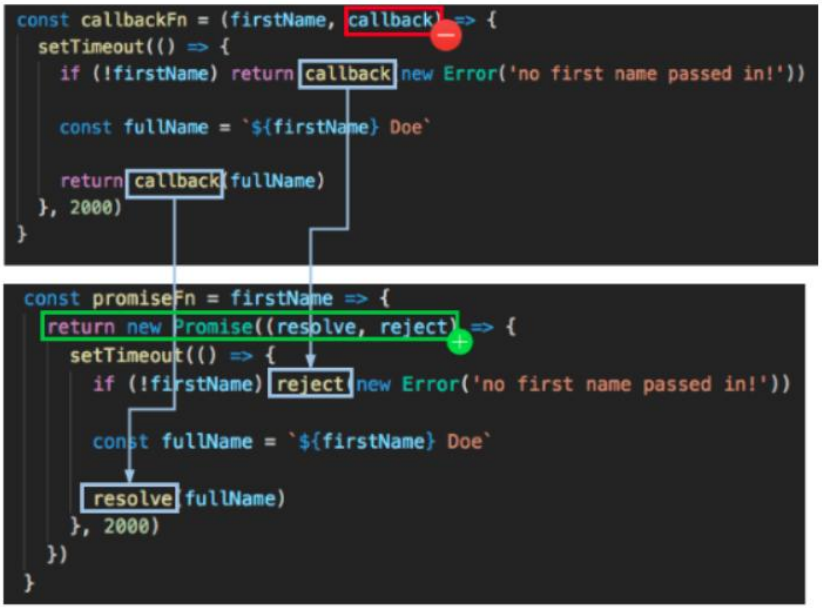
Promise version - And here's the Promise-based version of that function:

```
const promiseFn = firstName => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (!firstName) reject(new Error("no first name passed in!"));

      const fullName = `${firstName} Doe`;

      resolve(fullName);
    }, 2000);
  });
};

promiseFn("Jane").then(console.log);
promiseFn().catch(console.log);
```



converting-callback-to-Promise-and-async-await-2

Function without callback implementation

```
function printString(string) {
  setTimeout(() => {
    console.log(string);
  }, Math.floor(Math.random() * 100) + 1);
}
```

Let's try to print the letters A, B, C in that order:

```
function printAll() {
  printString("A");
  printString("B");
  printString("C");
}
printAll();
```

You will notice that A, B, and C print in a different and random order each time you call printAll!

Function with callback implementation

A callback is a function that is passed to another function. When the first function is done, it will run the second function.

```
function printString(string, callback) {
  setTimeout(() => {
    console.log(string);
    callback();
  }, Math.floor(Math.random() * 100) + 1);
}
```

Again, let's try to print the letters A, B, C in that order:

```
function printAll() {
  printString("A", () => {
    printString("B", () => {
      printString("C", () => {});
    });
  });
}
printAll();
```

Promise implementation

```
function printString(string) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(string);
      resolve();
    }, Math.floor(Math.random() * 100) + 1);
  });
}


function printAll() {
  printString("A")
    .then(() => {
      return printString("B");
    })
    .then(() => {
      return printString("C");
    });
}
printAll();


function printAll() {
  printString("A")
    .then(() => printString("B"))
    .then(() => printString("C"));
}
printAll();
```

Async-Await implementation

```
async function printAll() {
  await printString("A");
  await printString("B");
  await printString("C");
}
printAll();
```

Now pass output of one function to the next function

Callbacks Implementation

```
function addString(previous, current, callback) {
  setTimeout(() => {
    callback(previous + " " + current);
  }, Math.floor(Math.random() * 100) + 1);
}

// And in order to call it:

function addAll() {
  addString("", "A", result => {
    addString(result, "B", result => {
      addString(result, "C", result => {
        console.log(result); // Prints out " A B C"
      });
    });
  });
}
addAll();
```

Promise Implementation

```
function addString(previous, current) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(previous + " " + current);
    }, Math.floor(Math.random() * 100) + 1);
  });
}

// And in order to call it:

function addAll() {
  addString("", "A")
    .then(result => {
      return addString(result, "B");
    })
    .then(result => {
      return addString(result, "C");
    })
    .then(result => {
      console.log(result); // Prints out " A B C"
    });
}
addAll();

function addAll() {
  addString("", "A")
    .then(result => addString(result, "B"))
    .then(result => addString(result, "C"))
    .then(result => {
      console.log(result); // Prints out " A B C"
    });
}
addAll();
```

Async-Await Implementation

```
async function addAll() {
  let toPrint = "";
  toPrint = await addString(toPrint, "A");
  toPrint = await addString(toPrint, "B");
  toPrint = await addString(toPrint, "C");
  console.log(toPrint); // Prints out " A B C"
}
addAll();
```

calback-hell-resolved-with-promise

```
addOneToNum = (number, callback) => {
  let result = number + 1
  if (callback) {
    callback(result)
  }
}


addOneToNum(5, res => {
  console.log(res)
})


addOneToNum(5, res1 => {
  addOneToNum(res1, res2 => {
    addOneToNum(res2, res3 => {
      addOneToNum(res3, res4 => {
        addOneToNum(res4, res5 => {
          console.log(res5)
        })
      })
    })
  })
})


addOneToNumPromise = (number, callback) => {
  let result = number + 1
  return new Promise((resolve, reject) => {
    resolve(result)
  })
}


addOneToNumPromise(5).then(res => console.log(res))


addOneToNumPromise(5)
  .then(res1 => {
    return addOneToNumPromise(res1)
  })
  .then(res2 => {
    return addOneToNumPromise(res2)
  })
  .then(res3 => {
    return addOneToNumPromise(res3)
  })
  .then(res4 => {
    return addOneToNumPromise(res4)
```

```
  })
  .then(res5 => {
    console.log(res5)
  })


addOneToNumPromise(5)
  .then(res1 => addOneToNumPromise(res1))
  .then(res2 => addOneToNumPromise(res2))
  .then(res3 => addOneToNumPromise(res3))
  .then(res4 => addOneToNumPromise(res4))
  .then(res5 => console.log(res5))


addOneToNumPromise(5)
  .then(addOneToNumPromise)
  .then(addOneToNumPromise)
  .then(addOneToNumPromise)
  .then(addOneToNumPromise)
  .then(console.log)
```

multiple-API-fetch-before-executing-next-function-in-React-Promise-1

```
// Refactor getStudents and getScores to return  Promise for their response bodies
function getStudents() {
  return fetch(`api/students`, {
    headers: {
      "Content-Type": "application/json",
      Accept: "application/json"
    }
  }).then(response => response.json());
}

function getScores() {
  return fetch(`api/scores`, {
    headers: {
      "Content-Type": "application/json",
      Accept: "application/json"
    }
  }).then(response => response.json());
}

// Request both students and scores in parallel and return a Promise for both values.
// `Promise.all` returns a new Promise that resolves when all of its arguments resolve.
function getStudentsAndScores() {
  return Promise.all([getStudents(), getScores()]);
}

// When this Promise resolves, both values will be available. And because getStudentsAndScores()
returns a Promise, I can chain a .then() function to it.
getStudentsAndScores().then(([students, scores]) => {
  // both have loaded!
  console.log(students, scores);
});
```

How-Promise-makes-code-Asynchronous-non-blocking

```
fetch("http://api.exampledomain.com/api/search")
  .then(function(response) {
    return response.json();
  })
  .then(function(jsonData) {
```

```
    //handle json data processing here
  });
```

Use case of Promise.all()

```
getABC = async () => {
  let A = await getValueA(); // getValueA takes 2 second to finish
  let B = await getValueB(); // getValueB takes 4 second to finish
  let C = await getValueC(); // getValueC takes 3 second to finish

  return A * B * C;
};
```

```
getABC = async () => {
  // Promise.all() allows us to send all requests at the same time. But of course, it will give me 3
independent results, from the 3 independent function invocations. From those 3 independent results,
getting the final return value by applying reduce on them.

  let results = Promise.all([getValueA, getValueB, getValueC]);

  return results.reduce((total, value) => total * value);
};
```

Most common Node Interview Topics & Questions

why-nodejs-required-at-all-and-difference-vs-plain-js

A Node.js jó választás azoknak az alkalmazásoknak, amelyeknek muszáj

- nagy mennyiségű rövid késleltetést igénylő rövid üzenet feldolgozása. Az ilyen rendszereket valós idejű alkalmazásoknak (RTA) nevezzük - specifikációinak köszönhetően a Node.js jó választás lesz a valós idejű, együttműködő szerkesztő típusú alkalmazásokhoz, ahol megnézheti, hogy valaki más élőben módosítja a dokumentumot (például Trello, Dropbox Paper vagy Google Docs). Az RTA-k egyik legnépszerűbb felhasználása az élő csevegés és az azonnali üzenetküldés.
- Videokonferencia-alkalmazás, amely meghatározott hardverrel vagy VoIP-vel fog működni.
- online játékalkalmazások vagy e-kereskedelmi tranzakciós szoftverek, ahol az online adatok nagy jelentőséggel bírnak
- A Node természetes módon alkalmas az adatok tárolására objektum DB-kből (pl. MongoDB). A JSON által tárolt adatok lehetővé teszik a Node.js működését adatkonverzió nélkül.
- A Node.js nagyon hatékony a valós idejű alkalmazásokkal: megkönnyíti több kliens kérés kezelését, lehetővé teszi a könyvtár kódcsomagjainak megosztását és újrafelhasználását, és az ügyfél és a szerver közötti adatszinkron nagyon gyorsan történik.

Amikor a Node.js nem használható?

- Mikor van sok szinkron kód, amelyet futtatni kell. Azok az alkalmazások, amelyek nagy számolást vagy adatelemzést igényelnek, nem használhatják a Node.js fájlt. Például Mátrix szorzás, összegzés, nagy adatkészletek összesítése. Ilyen esetekben olyan alkalmazásokat kell használni, amelyek lehetővé teszik a Multi Threading használatát, mint például a Python, a Java.
- Nehéz CPU-feladatok - A Node.js eseményvezérelt, nem blokkoló I / O modellen alapul, és csak egyetlen CPU-magot használ. A processzorigényes műveletek csak blokkolják a beérkező kéréseket, így a Node.js legnagyobb előnye haszontalan lesz.

Authentication-vs-Authorization

Az authentication a felhasználó azonosságának ellenőrzése valamilyen hitelesítő adatok megszerzésével, és ezek felhasználásával a felhasználó személyazonosságának ellenőrzésére. Ha a hitelesítő adatok érvényesek, elindul az engedélyezési (authorization) folyamat. A hitelesítési (authentication) folyamat mindig az engedélyezési (authorization) folyamat felé halad.

Az engedélyezés (authorizaiton) az a folyamat, amely lehetővé teszi a hitelesített (authenticated) felhasználók számára az erőforrásokhoz való hozzáférést annak ellenőrzésével, hogy a felhasználó rendelkezik-e hozzáférési jogokkal a rendszerhez. A jogosultság segít a hozzáférési jogok

ellenőrzésében azáltal, hogy meghatározott engedélyeket ad meg vagy tagad meg egy hitelesített felhasználónak.

Például egy adott egyetem hallgatóinak hitelesíteniük kell magukat, mielőtt hozzáférnének az egyetem hivatalos honlapjának hallgatói linkjéhez. Ezt hívják hitelesítésnek.

Másrészt az Engedélyezés (authorization) pontosan meghatározza, hogy a hallgatók milyen információkhoz férhetnek hozzá az egyetem honlapján a sikeres hitelesítést követően.

bodyParser_what-does-it-do

Miért használja a body-parser csomagot. Felépítheti a projektet bodyParser nélkül

A> A HTTP POST kérés kezeléséhez az Express.js 4-es és újabb verzióiban telepítenie kell a body-parser nevű middleware-t.

- body-parser kivonja a bejövő kérésfolyam teljes testrészét, és kiteszi a req.body fájlra.
- A midlleware korábban az Express.js része volt, de most külön kell telepítenie.
- Ez a body elemző modul elemzi a HTTP POST kéréssel elküldött JSON-, puffer-, karakterlánc- és URL-kódolt adatokat.
- app.use (bodyParser) - a bodyParser hozzáadásával biztosítani tudja, hogy szervere az express köztes szoftveren keresztül kezelje a beérkező kéréseket. Tehát most a bejövő kérések törzsének elemzése annak az eljárásnak a része, amelyet a midlleware szoftver a bejövő kérések kezelésekor elvégez - mindez azért van, mert az app.use (bodyParser) nevet hívta.

Kicsit jobban elmélyülni; A body-parser egy olyan köztes programot ad, amely a nodejs / zlib segítségével kicsomagolja a bejövő kérelem adatait, ha azok tömörítve vannak, és a stream-utils / raw-body megvárja a kérelem törzsének teljes, nyers tartalmát, mielőtt "elemezné".

Miután megkapta a nyers tartalmat, a body-parser a négy stratégia egyikével elemzi azt, attól függően, hogy melyik midlleware szoftvert választotta:

- bodyParser.raw (): Valójában nem elemzi a body-t, hanem csak az előző pufferelt tartalmakat tárolja a req.body pufferben.
- bodyParser.text (): A puffert sima szövegként olvassa fel, és az így kapott karakterláncot kiteszi a req.body fájlra.
- bodyParser.urlencoded (): A szöveget URL kódolt adatokként elemzi (így a böngészők hajlamosak a POST-ra beállított szokásos űrlapok adatainak küldésére), és a kapott objektumot (amely tartalmazza a kulcsokat és értékeket) a req.body-on tárolja
- bodyParser.json (): A szöveget JSON-ként elemzi, és az eredményül kapott objektumot kiteszi a req.body-ra.

Csak a req.body kívánt értékre állítása után hívja meg a verem következő köztes szoftverét, amely ekkor hozzáférhet a kérelem adatokhoz anélkül, hogy azon kellene gondolkodnia, hogyan kell kibontani és elemezni azokat.

app.use(bodyParser.json()); // for parsing application/json

app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded

app.use(multer()); // for parsing multipart/form-data

error-handling-in-node

```
app.get("/", function(req, res) {
  throw new Error("BROKEN"); // Express will catch this on its own.
})
```

```
app.get("/", function(req, res, next) {
  fs.readFile("/file-does-not-exist", function(err, data) {
    if (err) {
      next(err); // Pass errors to Express.
    } else {
      res.send(data);
    }
  });
});
```

throw error vs next(error)

throw error is brute-force ugly way to do error handling in ExpressJS apps. Basically, just throw the exception after it bubbles back up to the route handler

```
app.get("/users", function(req, res) {
  User.find(function(err, users) {
    // an error?! let's crash the app!
    if (err) {
      throw err;
    }
    // no error? ok, fine. do normal stuff here
    // res.render... etc.
  });
});
```

```
app.get("/users", function(req, res, next) {
  User.find(function(err, users) {
    // an error? get it out of here!
    if (err) {
      return next(err);
    }

    // no error? good. I'll do normal stuff here
    // res.render... etc.
  });
});
```

Ez egy egyszerű változás, de a return next (err) használata a dobás hibája helyett lehetővé teszi az aszinkron kód számára, hogy kivételt tegyen, és továbbra is elkapja az alkalmazásod hibakezelésének folyamata. a next(err) azt mondja az Express és a Connect keretrendszernek, hogy vigye tovább a hibát, amíg a funkciók köztes szoftverének hibakezelése nem tudja megfelelően ellátni.

Mikor kell használni a next() és mikor a return next()-et?

Példa arra, hogy itt nem használjuk a return-t:

```
app.use((req, res, next) => {
  console.log("This is a middleware");
  next();
  console.log("This is first-half middleware");
});

app.use((req, res, next) => {
  console.log("This is second middleware");
  next();
});

app.use((req, res, next) => {
  console.log("This is third middleware");
  next();
});
```

You will find out that the output in console is:

```
This is a middleware
This is second middleware
This is third middleware
This is first-half middleware
```

De ha return next()-et használun akkor rögtön kilép a callback-ből.

REST-architectural-concepts

A REST achitektúrális stílus hat megszorítást ír le.

1. Egységes felület - az erőforrásokat állandó azonosítóval azonosítják: Az URI-k az azonosító mindenütt megválasztott lehetőségei

Az egységes interfész-kényszer meghatározza a kliensek és a szerverek közötti interfészt. Leegyszerűsíti és szétválasztja az architektúrát, ami lehetővé teszi az egyes részek önálló fejlődését. Az egységes felület négy vezérelve:

- Erőforrás-alapú: Az egyéni erőforrásokat az URI-kat erőforrás-azonosítóként használó kérelmekben azonosítják. Maguk az erőforrások fogalmilag elkülönülnek a klienshez visszaküldött ábrázolásoktól. Például a kiszolgáló nem küldi el az adatbázisát, hanem néhány HTML, XML vagy JSON, amely képvisel bizonyos adatbázis-rekordokat, például finn nyelven kifejezve és UTF-8-ban kódolva, a kérés és a szerver megvalósításának részleteitől függően. .
- Az erőforrások kezelése reprezentációkon keresztül: Amikor az ügyfél rendelkezik egy erőforrás reprezentációjával, beleértve a csatolt metaadatokat is, elegendő információval

rendelkezik az erőforrás módosításához vagy törléséhez a kiszolgálón, feltéve, hogy erre engedélye van.

- Önleíró üzenetek: Minden üzenet elegendő információt tartalmaz az üzenet feldolgozásának leírására. Például azt, hogy mely elemzőt hívja meg, megadhatja egy internetes médiatípus (korábban MIME típusként ismert). A válaszok kifejezetten jelzik gyorsítótár-képességüket is.
- A Hypermedia mint az alkalmazás állapotának motorja (Hypermedia as the Engine of Application State- HATEOAS): A kliensek állapotot adnak le a törzs tartalmán, a lekérdezési karakterlánc paraméterein, a kérelem fejlécén és a kért URI (az erőforrás neve) útján. A szolgáltatások állapotot juttatnak el a kliensekhez a testtartalom, a válaszkódok és a válaszfejlécek segítségével. Ezt technikailag hipermédiának (vagy hipertexten belüli hivatkozásoknak) nevezik.
  A fenti leíráson kívül a HATEOS azt is jelenti, hogy ahol szükséges, a visszaküldött törzsben (vagy fejlécekben) linkek találhatók, amelyek biztosítják az URI-t az objektum vagy a kapcsolódó objektumok visszakereséséhez. A későbbiekben erről részletesebben beszélünk.

2. Stateless (állapottalan) - A kérések között nem tárolnak kliens adatokat a szerveren, a munkamenetek állapotát a kliens tárolja.

Mivel a REST a REpresentational State Transfer rövidítése, az állapot mentesség (sateless) kulcsfontosságú. Lényegében ez azt jelenti, hogy a kérés kezeléséhez szükséges állapot magában a kérésben található, legyen az URI, lekérdezési karakterlánc-paraméterek, törzs vagy fejlécek részeként. Az URI egyedileg azonosítja az erőforrást, és a törzs (body) tartalmazza az erőforrás állapotát (vagy állapotváltozását). Ezután a szerver feldolgozása után a megfelelő állapotot vagy az adott állapot (ok) t a fejléceken, az állapoton és a válasz body-n keresztül közlik a klienssel.

A REST-ben a kliensnek minden információt tartalmaznia kell a kiszolgáló számára a kérés teljesítéséhez, szükség esetén újraküldi az állapotot, ha annak több kérelemre is kiterjednie kell. A stateless nagyobb méretezhetőséget tesz lehetővé, mivel a szervernek nem kell fenntartania, frissítenie vagy kommunikálnia a munkamenet állapotát.

3. Gyorsítótárazható – A kliensek gyorsítótárazhatják a választ (akárcsak a weblap statikus elemeit tároló böngészők) a teljesítmény javítása érdekében.

A világhálóhoz hasonlóan a kliensek is tárolhatják a válaszokat. A jól kezelt gyorsítótár részben vagy teljesen kiküszöböli a kliens-szerver interakciókat, tovább javítva a skálázhatóságot és a teljesítményt.

4. Client-Server: A kliens kezeli a front-end-et, a szerver a backend-et kezel, és mindkettő egymástól függetlenül cserélhető.

Az egységes felület elválasztja a klienseket a szerverektől. Az aggodalmak ilyen elkülönítése azt jelenti, hogy például a kliensek nem foglalkoznak az adattárolással, amely továbbra is az egyes szerverek belső része, így a kliens hordozhatósága javul. A szerverek nem foglalkoznak a felhasználói felülettel vagy a felhasználó állapotával, így a szerverek egyszerűbbek és skálázhatóbbak lehetnek. A szerverek és a kliensek önállóan is cserélhetők és fejleszthetők, amennyiben az interfész nem változik.

5. Réteges rendszer: A kliens általában nem tudja megmondani, hogy közvetlenül a végkiszolgálóhoz vagy egy közvetítőhöz csatlakozik-e. A közvetítő szerverek javíthatják a rendszer méretezhetőségét azáltal, hogy engedélyezik a terheléselosztást és megosztott gyorsítótárakat biztosítanak. A rétegek kikényszeríthetik a biztonsági irányelveket is.

6. Igény szerinti kód (opcionális): A szerverek ideiglenesen kiterjeszthetik vagy testreszabhatják a kliens funkcionalitását azáltal, hogy átadják neki a végrehajtandó logikát. Ilyenek lehetnek például compiled components, például Java kisalkalmazások, és kliensoldali szkriptek, például JavaScript.

Ezeknek a korlátozásoknak való megfelelés és így a REST stílusnak való megfelelés lehetővé teszi, hogy bármilyen elosztott hipermédia rendszer kívánatos megjelenő tulajdonságokkal rendelkezzen, mint például teljesítmény, méretezhetőség, egyszerűség, módosíthatóság, láthatóság, hordozhatóság és megbízhatóság.

cookie-parser-what-does-it-do

A cookie-parser elemzi a sütiket, és a req objektumra vonatkozó cookie-információkat a midlleware programba helyezi.

cors_Why_its_needed

Cross-origin resource sharing (CORS) lehetővé teszi az AJAX kérések számára, hogy kihagyják a Same-origin policy-t, hozzáférjenek az erőforrásokhoz a távoli gazdagépektől.

A böngésző biztonsági házirendjének biztonsági korlátozásai megakadályozzák, hogy a böngésző AJAX-kéréseket küldjön egy másik tartományban lévő szerverre. Ezt same-origin policy-nek is nevezzük Más szavakkal, a böngésző biztonsága megakadályozza, hogy egy tartomány egyik weboldala AJAX-hívásokat hajtson végre egy másik tartományban. Vegye figyelembe, hogy a kérelem eredete a következőket tartalmazza: séma, gazdagép és portszám. Tehát két kérés akkor tekinthető azonos eredetűnek, ha ugyanaz a séma, a gazdagép és a portszám (Scheme, Host and Port). Ha ezek bármelyike eltér, akkor a kéréseket cross origin-nek tekintjük, vagyis nem azonos eredetekhez (same origin) tartoznak.

Az same-origin policy korlátozza, hogy az egyik eredetből betöltött dokumentum vagy szkript miként léphet kapcsolatba egy másik eredetű erőforrással. Kritikus biztonsági mechanizmus a potenciálisan rosszindulatú dokumentumok elkülönítésére. Biztonsági okokból a böngészők korlátozzák a szkripteken belül kezdeményezett, kereszt eredetű HTTP-kérelmeket. Például az XMLHttpRequest és a Fetch API azonos eredetű házirendet követ. Ez azt jelenti, hogy az említett API-kat használó webalkalmazások csak ugyanabból a forrásból kérhetnek HTTP-erőforrásokat, ahonnan az alkalmazást betöltötték, kivéve, ha a másik eredetű válasz tartalmazza a megfelelő CORS fejléceket.

A same origin policy korlátozza, hogy az egyik eredetből (origin) betöltött dokumentum vagy szkript miként léphet kapcsolatba egy másik eredetű (origin) erőforrással. Kritikus biztonsági mechanizmus

a potenciálisan rosszindulatú dokumentumok elkülönítésére. Biztonsági okokból a böngészők korlátozzák a szkripteken belül kezdeményezett, kereszt eredetű (cross-origin) HTTP-kérelmeket. Például az XMLHttpRequest és a Fetch API azonos eredetű (same origin policy) házirendet követ. Ez azt jelenti, hogy az említett API-kat használó webalkalmazások csak ugyanabból a forrásból kérhetnek HTTP-erőforrásokat, ahonnan az alkalmazást betöltötték, kivéve, ha a másik eredetű válasz tartalmazza a megfelelő CORS fejléceket.

A Cross-Origin Research Sharing (CORS) egy olyan mechanizmus, amely további HTTP fejléceket használ arra, hogy a böngészőnek megadja, hogy az egy eredetnél (origin) (tartományban) futó webalkalmazásoknak engedélyezniük kell a kiválasztott erőforrásokhoz való hozzáférést egy másik eredetű szerverről. Egy webalkalmazás kereszt-eredetű (cross-origin) HTTP-kérelmet küld, ha olyan erőforrást kér, amelynek eredete (tartomány, protokoll és port) eltér a saját eredetétől.

Példa cross-orogin kérelemre: A http://domain-a.com webhelyről kiszolgált webalkalmazás frontend JavaScript kódja az XMLHttpRequest használatával kérelmet nyújt be a http://api.domain-b.com/data.json címre. .

Biztonsági okokból a böngészők korlátozzák a szkripteken belül kezdeményezett, kereszt eredetű (cross-origin) HTTP-kérelmeket. Például az XMLHttpRequest és a Fetch API azonos eredetű házirendet (same origin policy) követ. Ez azt jelenti, hogy az említett API-kat használó webalkalmazások csak ugyanabból a forrásból kérhetnek HTTP-erőforrásokat, ahonnan az alkalmazást betöltötték, kivéve, ha a másik eredetű válasz tartalmazza a megfelelő CORS fejléceket.

http-vs-https

https = http + cryptographic protocols.

HTTPS: Encrypted Connections

A kettő lényegében ugyanaz, mivel mindkettő ugyanarra a „hipertext átviteli protokollra” hivatkozik, amely lehetővé teszi a kért webes adatok megjelenítését a képernyőn. De a HTTPS még mindig kissé más, fejlettebb és sokkal biztonságosabb.

Egyszerűen fogalmazva, a HTTPS protokoll a HTTP kiterjesztése. A rövidítésben szereplő „S” a Secure szóból származik, és a Transport Layer Security (TLS) [a Secure Sockets Layer (SSL utódja)] hajtja, ez a szabványos biztonsági technológia, amely titkosított kapcsolatot hoz létre egy webkiszolgáló és egy webszerver között.

HTTPS nélkül a webhelyre beírt minden adatot (például felhasználónév / jelszó, hitelkártya vagy banki adatok, bármely más űrlap-benyújtási adat stb.) Szöveges üzenetet küldünk, és így hajlamosak lehallgatásra. Ezért minden információ megadása előtt mindig ellenőriznie kell, hogy egy webhely HTTPS-t használ-e.

A szerver és a böngésző között továbbított adatok titkosításán túl a TLS hitelesíti azt a szervert is, amelyhez csatlakozik, és megvédi az átadott adatokat a hamisítástól.

A https biztonságának elérése érdekében a nyilvános kulcsú infrastruktúrát (PKI) is használják, mivel a nyilvános kulcsokat több webböngésző, míg a magánkulcsot az adott webhely webszervere használhatja. Ezeknek a nyilvános kulcsoknak a terjesztése a Böngésző által fenntartott tanúsítványokon keresztül történik. Ezeket a tanúsítványokat a Böngésző beállításaiban ellenőrizheti.

Különbségek a HTTP és a HTTPS között - néhány kulcsfontosságú pont

- A HTTP-ben az URL kezdete „http: //”, míg az URL „https: //” kezdetű
- A HTTP a 80-as portot használja a kommunikációhoz, a HTTPS pedig a 443-at
- A HTTP működik az alkalmazásrétegen, a HTTPS pedig a szállítási rétegen
- A HTTP-ben a titkosítás hiányzik, és a titkosítás jelen van a HTTPS-ben, a fentiek szerint
- A HTTP nem igényel tanúsítványokat, a HTTPS pedig SSL tanúsítványokat igényel
- Ha a küldő és a címzett között bárki meg tudja nyitni az üzenetet, a HTTPS segítségével továbbra sem értheti meg. Csak a feladó és a címzett tudja megfejteni az üzenetet, akik ismerik a "kódot".
- Ehhez a számítógép mindkét végén egy "SSL-tanúsítvány" nevű dokumentumot használ, amely karakterláncokat tartalmaz, amelyek a titkos "kódjuk" kulcsai.

Az SSL tanúsítványok tartalmazzák tulajdonosának "nyilvános kulcsát".

A tulajdonos megosztja a nyilvános kulcsot bárkivel, akinek szüksége van rá. A többi felhasználónak a nyilvános kulcsra van szüksége a tulajdonosnak küldött üzenetek titkosításához. A tulajdonos elküldi a felhasználóknak az SSL tanúsítványt, amely tartalmazza a nyilvános kulcsot. A tulajdonos nem osztja meg a magánkulcsot senkivel.

A HTTPS előnyei

- A legtöbb esetben a HTTPS-en futó webhelyek átirányítással rendelkeznek. Ezért, még ha beírja is a HTTP: // fájlt, egy biztonságos kapcsolaton keresztül irányít át egy https-re
- Lehetővé teszi a felhasználók számára, hogy biztonságos e-kereskedelmi tranzakciókat hajtsanak végre, például online banki tevékenységet.
- Az SSL technológia védi a felhasználókat és növeli a bizalmat
- Egy független hatóság ellenőrzi a tanúsítvány tulajdonosának személyazonosságát. Tehát minden SSL-tanúsítvány egyedi, hitelesített információkat tartalmaz a tanúsítvány tulajdonosáról.

A HTTPS korlátai

- A HTTPS protokoll nem hagyja a bizalmas információk lopását a böngészőben tárolt oldalakról
- Az SSL-adatok csak a hálózaton történő továbbítás során titkosíthatók. Tehát nem tudja törölni a böngésző memóriájában található szöveget
- A HTTPS növelheti a számítási és a hálózati általános költségeket

Most common Angular Interview Topics & Questions

AsyncPipe-basic-Oberservable-use-case

First the implementation without asyn pipe, with a simple observable, like so:

```
   import { Observable } from 'rxjs/Rx';
.
.
.
@Component({
  selector: 'async-pipe',
  template: `
 <div class="card card-block">
  <h4 class="card-title">AsyncPipe</h4>
  <p class="card-text" ngNonBindable>{{ observableData }}
  <p class="card-text">{{ observableData }}</p> (1)
 </div>
`
})
class AsyncPipeComponent {

  // Declared these local states to hold the emitted value from observable
  // 'observableData' will actually be used as a data in the template
  observableData: number;
  // this 'subscription' state is only to store a reference to the subscription so we can unsubscribe
to it later.
  subscription: Object = null;

  constructor() {
    this.subscribeObservable();
  }

  getObservable() { (2)
    return Observable
        .interval(1000)
        .take(10)
        .map((v) => v * v);
  }

  subscribeObservable() { (3)
    this.subscription = this.getObservable()
        .subscribe( v => this.observableData = v);
  }

// We should also be destroying the subscription when the component is destroyed. Otherwise we will
start leaking data as the old observable, which isn't used any more, will still be producing results.
  ngOnDestroy() {
    if (this.subscription) {
      this.subscription.unsubscribe();
    }
  }

}
```

By using AsyncPipe we don't need to perform the subscribe and store any intermediate data on our component, like so:

```
@Component({
    selector: "async-pipe",
    template: `
        <div class="card card-block">
            <h4 class="card-title">AsyncPipe</h4>
            <p class="card-text" ngNonBindable>{{ observable | async }}</p>
            <p class="card-text">{{ observable | async }}</p>
            (1)
```

```
        </div>
    `
})
class AsyncPipeComponent {
    observable: Observable<number>;

    constructor() {
        this.observable = this.getObservable();
    }

    getObservable() {
        return Observable.interval(1000)
            .take(10)
            .map(v => v * v);
    }
}
```

Component-Communications-via-Input

3 common methods that we can use to share data between Angular components.

1. Sharing Data via Input
2. Sharing Data via Output and EventEmitter
3. Sharing Data with a Service

Parent to Child via @Input() decorator.

When you declare a variable in the child component with the @Input() decorator, it allows that variable to be “received” from the parent component template.

parent.component.ts

```
import { Component } from "@angular/core"

@Component({
  selector: "app-parent",
  template: "parent.component.html",
  styleUrls: ["./parent.component.css"],
})
export class ParentComponent {
  Message = "Parent to Child"
  constructor() {}
}
```

parent.component.html

See that the way a prop is passed from the parent.component.html is using the syntax

```
[prop]="prop"
```

child.component.ts

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-child',
  template: './child.component.html,
  styleUrls: ['./child.component.css']
```

```
})
export class ChildComponent {

  @Input() Message: string;

  constructor() { }

}
```

child.component.html

```
<h1>
  Message from Parent : {{Message}}
  <h1></h1>
</h1>
```

Component-Communications-via-Output-EventEmitter.

Az adatok megosztásának másik gyakran használt módja az, ha adatokat bocsát ki Gyermektől Szülőig. Ezzel a megközelítéssel könnyű átadni az adatokat olyan események által, mint például a gombok kattintása. A szülőkomponensben létre kell hoznunk egy módszert az üzenetek fogadására, a gyermekkomponensben pedig a @Output () dekorátorral deklarálunk egy messageEvent változót, és egyenlővé tesszük egy új eseménykibocsátóval. Ezt követően létrehozhatunk egy módszert az adatok kibocsátására, és egy gombnyomásra kiválthatjuk.

parent.component.ts

```
import { Component } from "@angular/core";

@Component({
  selector: "app-parent",
  template: "./parent.component.html",
  styleUrls: ["./parent.component.css"]
})
export class ParentComponent {
  constructor() {}

  message: string;

  receiveMessage($event) {
    this.message = $event;
  }
}
```

parent.component.html

```
<app-child (messageEvent)="receiveMessage($event)"></app-child>
<h1>
  Message from Child : {{message}}
  <h1></h1>
</h1>
```

child.component.ts

```
import { Component, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-child',
  template: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
```

```
  message: string = "Hello Angular!"

  @Output() messageEvent = new EventEmitter<string>();

  constructor() { }

  sendMessage() {
    this.messageEvent.emit(this.message)
  }
}
```

child.component.html

```
<button (click)="sendMessage()">Send Message</button>
```

What is the difference betwen @Input and @Output

@Input() is to pass data In to the component

```
class ChildComponent {
  @Input() data;
}

@Component({
  template: `<child [data]="parentData"></child>
})
class ParentComponent {
  parentData;
}
```

@Output is to emit data (events) Out from a component

```
class ChildComponent {
  @Output() dataChange = new EventEmitter();

  click() {
    dataChange.emit('new Value');
  }
}

@Component({
  template: `<child (dataChange)="onDataChange($event)"></child>
})
class ParentComponent {

  onDataChange(event) {
    console.log(event);
  }
}
```

Observable-basics

Observables in Angular

```
import { Observable } from "rxjs/Rx"
import { Injectable } from "@angular/core"
import { Http, Response } from "@angular/http"

@Injectable()
```

```
export class HttpClient {

    constructor(
        public http: Http
    ) {}

    public fetchUsers() {
        return this.http.get("/api/users").map((res: Response) => res.json())
    }
}
```

Komponensben két féle módon tudunk hozzáférni: az egyik az async pipe:

component.ts

```
import { Component } from "@angular/core"
import { Observable } from "rxjs/Rx"

// client
import { HttpClient } from "../services/client"

// interface
import { IUser } from "../services/interfaces"

@Component({
    selector: "user-list",
    templateUrl:  "./template.html",
})
export class UserList {

    public users$: Observable<IUser[]>

    constructor(
        public client: HttpClient,
    ) {}

    // do a call to fetch the users on init of component
    // the fetchUsers method returns an observable
    // which we assign to the users$ property of our class
    public ngOnInit() {
        this.users$ = this.client.fetchUsers()
    }
}
```

```
<!-- We use the async pipe to automatically subscribe/unsubscribe to our observable -->
<ul class="user__list" *ngIf="(users$ | async).length">
    <li class="user" *ngFor="let user of users$ | async">
        {{ user.name }} - {{ user.birth_date }}
    </li>
</ul>
```

Itt a komponensben a users$-nál a $ a végén egy bevett módszer arra hogy jelezzük hogy ez egy Observable.

A másik mód a scubscribe() meghívása az Observable-re.

```
import { Component } from "@angular/core"

// client
import { HttpClient } from "../services/client"

// interface
import { IUser } from "../services/interfaces"

@Component({
    selector: "user-list",
```

```
    templateUrl:  "./template.html",
})
export class UserList {

    public users: IUser[]

    constructor(
        public client: HttpClient,
    ) {}

    // do a call to fetch the users on init of component
    // we manually subscribe to this method and take the users
    // in our callback
    public ngOnInit() {
        this.client.fetchUsers().subscribe((users: IUser[]) => {

            // do stuff with our data here.
            // ....

            // asign data to our class property in the end
            // so it will be available to our template
            this.users = users
        })
    }
}
```

```
<ul class="user__list" *ngIf="users.length">
    <li class="user" *ngFor="let user of users">
        {{ user.name }} - {{ user.birth_date }}
    </li>
</ul>
```

cold-vs-hot-observable

Hot vs Cold Observables

A Cold Observables csak akkor kezdi el kibocsátani vagy létrehozni az értékeket, amikor a feliratkozás (subscription) megkezdődik, például egy tipikus YouTube-videó. Minden feliratkozó (subscriber) az elejétől a végéig ugyanazt az eseménysorozatot (vagy mintát) látja.

A Hot Observables mindig új értékekkel frissül, például egy élő közvetítés a YouTube-on. Ha feliratkozik, akkor a legfrissebb értékkel kezdi, és csak a jövőbeni változásokat látja.

A gondolkodás másik módja - Egy Observable-t „cold” nevezünk, ha nem kezd el bocsátani, amíg egy observer nem iratkozott fel rá; egy Observable-t „hot” nevezünk, ha bármikor megkezdheti a kibocsátását, és az subscriber megkezdheti a kibocsátott tételek sorrendjének megfigyelését valamikor annak megkezdése után, kihagyva a korábban az subscription időpontjáig kibocsátott tételeket .

Cold Observable Example

We know an Observable is cold if we subscribe at the same time, but get a different value.

```
const cold = Rx.Observable.create((observer) => {
  observer.next(Math.random())
})

cold.subscribe((a) => console.log(`Subscriber A: ${a}`))
cold.subscribe((b) => console.log(`Subscriber B: ${b}`))
```

```
// Subscriber A: 0.2298339030
// Subscriber B: 0.9720023832
```

Hot Observable Example

A hot observable gets its values from an outside source. We can make it hot by simply moving the random number outside of the observable creation function.

```
const x = Math.random()

const hot = Rx.Observable.create((observer) => {
  observer.next(x)
})

hot.subscribe((a) => console.log(`Subscriber A: ${a}`))
hot.subscribe((b) => console.log(`Subscriber B: ${b}`))
// Subscriber A: 0.312580103
// Subscriber B: 0.312580103
```

But how do we make an already cold observable hot? We can make a cold Observable hot by simply calling publish() on it. This will allow the subscribers to share the same values in realtime. To make it start emitting values, you call connect() after the subscription has started.

```
const cold = Rx.Observable.create((observer) => {
  observer.next(Math.random())
})

const hot = cold.publish()

hot.subscribe((a) => console.log(`Subscriber A: {a}`))
hot.subscribe((b) => console.log(`Subscriber B: {b}`))

hot.connect()

/// Subscriber A: 0.7122882102
/// Subscriber B: 0.7122882102
```

angular-httpclient-unsubscribe

Is there a need to unsubscribe from the Observable the Angular HttpClient's methods return?

For example:

```
this.http.get(url).subscribe(x => this.doSomething());
```

No, You don't need to unsubscribe it. It observes until getting a response from api. And once the Http request completes it unsubscribes automatically.

component-selectors-different-way

1. Selector can directly be used by typing element-name directly as a legacy selector:

```
@Component({
    selector: 'app-element',
    template:  './element.component.html',
    styleUrls: ['./element.component.css']
})
```

This type of selector can access directly by typing the selector name inside the <> brackets as:

```
<app-element></app-element>
```

2. The selector can be used as attribute selector by putting it inside a square brackets:

```
@Component({
    selector: '[app-element]',
    template:  './element.component.html',
    styleUrls: ['./element.component.css']
})
```

```
<div app-element></div>
```

3. The selector can also be selected by class just like in a CSS, by putting a dot in the beginning:

```
@Component({
    selector: '.app-element',
    template:  './element.component.html',
    styleUrls: ['./element.component.css']
})
```

In this, we can select by class as:

```
<div class="app-element"></div>
```

Most common CSS Interview Topics & Questions

flexbox

Mi is pontosan a Flexbox -

A Flexbox Layout (Flexible Box) modul célja, hogy hatékonyabban biztosítsa a hely elrendezését, igazítását és elosztását a tárolóban lévő elemek között, még akkor is, ha méretük ismeretlen és / vagy dinamikus (tehát a "flex" szó).

Konténer és tárgyak

A flexbox-ban van egy konténerünk és annak elemei. Ez a flexbox bármelyik komponensének két alkotóeleme. Bármely html elem, amely a flexbox-tároló közvetlen gyermeke, flexbox-elemmé válik.

A rugalmas elrendezés fő gondolata az, hogy a tárolónak lehetővé tegye az elemek szélességének / magasságának (és sorrendjének) megváltoztatását, hogy a lehető legjobban kitöltse a rendelkezésre álló helyet (főleg mindenféle megjelenítő eszköz és képernyőméret számára). A rugalmas tároló kibővíti az elemeket, hogy kitöltse a rendelkezésre álló szabad helyet, vagy összezsugorítja őket a túlcsordulás megakadályozása érdekében.

Ami a legfontosabb, hogy a flexbox elrendezése irány-agnosztikus, szemben a szokásos elrendezéssel (blokk, amely függőlegesen és inline, amely vízszintesen alapul). Bár ezek jól működnek az oldalaknál, nem rendelkeznek rugalmassággal a nagy vagy összetett alkalmazások támogatásához (különösen, haátméretezés, nyújtás, zsugorítás stb.).

A Flexible Layout must have a parent element with the display property set to flex. Like below

```
.flex-container {
  display: flex;
}
```

Ne feledje, hogy ez az egyetlen tulajdonság, amelyet be kell állítania a szülőtárolóba, és annak közvetlen gyermekei automatikusan flex elemek legyenek.

- hozzá kell adnom a display: flex-et a szülőhöz, majd flex:1-et a gyermekhez, hogy a gyermek kibővülhessen a szülő 100% -áig.

```
.fb-container {
  background-color:green;
  flex: 1;

}
.somedatadiv {
  width: 75%;
  max-width: 345px;
  background-color: grey;
  padding: 30px;
```

```
}
<body style="height:100vh;display:flex;">
  <div class="fb-container">
    <div class="somedatadiv">
      Some data
    </div>
    <div class="anotherdiv">
      data
    </div>
  </div>
</body>
```

Grid-Layout

Mi is pontosan a Grid Layout

A CSS Grid Layout a CSS-ben elérhető legerősebb elrendezési rendszer. Ez egy kétdimenziós rendszer, vagyis oszlopokat és sorokat is képes kezelni, ellentétben a flexbox-szal, amely nagyrészt egydimenziós rendszer. A Grid Layout-al úgy dolgozol, hogy CSS-szabályokat alkalmazol mind a szülőelemre (amelyből a Grid Container lesz), mind a gyermekelemekre (amelyek Grid Items válnak).

A CSS Grid Layout (más néven "Grid") egy kétdimenziós rácsalapú elrendezési rendszer, amelynek célja nem kevesebb, mint a rácsalapú felhasználói felületek tervezésének teljes megváltoztatása. A CSS-t mindig is használták weblapjaink elrendezésére, de soha nem végzett túl jó munkát. Először tables, majd floats, positioning és inline-block használtunk, de ezek a módszerek lényegében hackek voltak, és sok fontos funkciót kihagytak (például függőleges központosítást). A Flexbox segített, de egyszerűbb, egydimenziós elrendezésre készült, nem pedig összetett kétdimenziósra (a Flexbox és a Grid valójában nagyon jól működnek együtt). A Grid a legelső CSS-modul, amelyet kifejezetten az elrendezési problémák megoldására hoztak létre.

A táblákhoz hasonlóan a rácsos elrendezés is lehetővé teszi a szerző számára az elemek oszlopokba és sorokba igazítását. A CSS Grid segítségével azonban sokkal több elrendezés lehetséges vagy könnyebb, mint a táblázatoknál. Például egy grid container gyermekei elhelyezhetik magukat, így valóban átfedik egymást és rétegeznek, hasonlóan a CSS-ben elhelyezett elemekhez.

grid-column-start grid-column-end grid-row-start grid-row-end: Meghatározza a rácselem helyét a rácson belül, konkrét rácsvonalakra hivatkozva. grid-column-start/grid-row-start az a sor, ahol az elem kezdődik, és a grid-column-end/grid-row-end az a sor, ahol az elem véget ér.

grid-column-gap and grid-row-gap: oszlopok/sorok közötti rés beállítása

grid-template-columns and grid-template-rows: A rács oszlopait és sorait szóközzel elválasztott értéklistával határozza meg. Az értékek a sáv méretét, a köztük lévő tér pedig a rácsvonalat jelöli.

left-vs-margin-left

Margin a Box model része.

```
.my_box {
  display:block;
  margin-left:10px;
}
```

Top, left, etc

szó szerint megmondják az elemeknek, hogy hol legyenek.

A fenti stílus azt mondja, hogy az elem 10 pixel legyen a tárolójától balra (if position:relative, or the page (if position:absolute).

not-pseudo-class-selector

Állítson be háttérszínt minden elemhez, amely nem a p elem:

```
:not(p) {
          background: #ff0000;
}
```

A: not (X) tulajdonság a CSS-ben negatív pseudo-class, és egy egyszerű selector-t fogad el argumentumként. Lényegében ez csak egy bármilyen típusú választó. :not felel meg egy olyan elemnek, amelyet nem képvisel az argumentum. Az átadott argumentum nem tartalmazhat további szelektorokat vagy pseudo-szelektorokat.

```
li:not(.different) {
          font-size: 3em;
}
```

rem-unit-basics-and-converting-px

Az em egyenlő az elem szülőjének számított betűméretével. Például, ha van egy div elem, amelynek font-size: 16px van megadva, akkor ennek a div-nek és gyermekeinek 1em = 16px. Ha a betűméretet nem határozzák meg kifejezetten, akkor ez az elem örökli a szülőelemtől. Az öröklés az ősök körében a gyökérelemig továbbra is így zajlik. A gyökérelem alapértelmezett betűméretét a böngésző biztosítja.

Ha em egységeket szeretne használni egységeihez, akkor körültekintően kell eljárnia az elrendezéssel kapcsolatban. Jó gyakorlat, ha nem definiáljuk a betűméretet kifejezetten, a gyökérelem kivételével, miközben az em-et használod a projektedben.

Mi a rem egység

rem jelentése gyökér em. A rem egység a root vagy a html elemhez viszonyított. Ez azt jelenti, hogy meghatározhatunk egyetlen betűméretet a html elemen, és az összes rem egységet meghatározhatjuk annak százalékában. ”.

Konvertálás px-ről rem-re

A legtöbb modern böngészőben az 1 rem egyenlő 16 pixellel. Tehát 1rem (más néven 16px) alapmérettel, most egyszerű felosztással tudjuk kideríteni az elemek megfelelő méretét.

Ha szóközről és betűméretről van szó, akkor inkább a rem-et használom. Mivel a rem a root elem betűméretét használja a szülő betűmérete helyett. Tegyük fel, hogy a font-size: 10px van beállítva a gyökérelemhez, amely 1rem = 10px-t jelent mindenhol a weboldalunkon. Mivel 1px = 0,1rem, a számítások egyszerűek.

Git

git stash - How to Save Your Changes Temporarily

Gyakran, amikor a projekt egy részén dolgoztál, a dolgok rendetlen állapotban vannak, és egy kicsit át akarsz váltani egy branch-et, hogy mással dolgozz. A probléma az, hogy nem akar félkész munkát commit-olni, csak később térhet vissza erre a pontra. A válasz erre a kérdésre a git stash parancs.

A Stashing megkapja a munkakönyvtár piszkos állapotát - vagyis a módosított nyomon követett fájlokat (tracked files) és a lépcsőzetes módosításokat (staged changes) -, és elmenti egy stack-be befejezetlen változtatáson, amelyeket bármikor újra alkalmazhat.

A munka elhalasztása A demonstrálás érdekében belemegy a projektbe, és elkezd dolgozni egy pár fájlon, és esetleg elvégzi az egyik változtatást. Ha futtatja a git status-t, láthatja piszkos állapotát:

$ git status

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

Most branch-et akar váltani, de nem akarja commit-olni azt, amin még dolgozott; így el fogja rejteni a változásokat. Ha új stash-t szeretne tolni a veremre, futtassa a git stash-t:

$ git stash

```
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Nézze meg, hogy a munkakönyvtár tiszta, ugyanúgy, mint akkor lett volna, ha commit-olom a változásokat.

git status

```
# On branch master
nothing to commit, working directory clean
```

Ezen a ponton könnyedén válthat branch-t és máshol végezhet munkát; a változtatások tárolódnak a veremben. A stash-ek megtekintéséhez használhatja a git stash-t:

$ git stash list

```
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

Ebben az esetben korábban két stash-t hajtottak végre, így három különböző elrejtett (stashed) munkához férhet hozzá. Az imént elrejtettet (stashed) újból alkalmazhatja az eredeti rejtés (stash) parancs súgó kimenetében látható paranccsal: git stash apply. Ha a régebbi stash egyikét szeretné alkalmazni, akkor megadhatja a nevével, így: git stash apply stash@{2}. Ha nem ad meg repository-t, a Git felveszi a legfrissebb repository-t, és megpróbálja alkalmazni:

Folytatás ott, ahol abbahagyta

A Git's Stash egy ideiglenes tároló. Ha készen áll a folytatásra ott, ahol abbahagyta, könnyen visszaállíthatja a mentett állapotot:

git stash apply

Ez a parancs a stash legfelső rejtettségét veszi át, és a repóra alkalmazza. Esetünkben @ @ {0}

Ha más repository-ra szeretne alkalmazni, megadhatja a repository azonosítóját.

Íme a példa:

git stash apply stash@{1}

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

Láthatja, hogy a Git módosítja azokat a fájlokat, amelyeket a stash mentésekor nem commit-olt. Ebben az esetben tiszta munkakönyvtárad volt, amikor megpróbáltad alkalmazni a stash-t, és ugyanarra az ágra próbáltad alkalmazni, ahonnan mentetted; de a tiszta munkakönyvtár megléte és ugyanazon az ágon történő alkalmazása nem szükséges a stash sikeres alkalmazásához. Menthet egy tárhelyet az egyik ágon, később átválthat egy másik ágra, és megpróbálhatja újra alkalmazni a módosításokat. A stash alkalmazásakor módosított és el nem kötött fájlokat is tartalmazhat a munkakönyvtárában - a Git merge konfliktusokat nyújt, ha bármi már nem érvényesül tisztán.

Here are some more useful trick about Git stash

git stash pop

Ez a parancs nagyon hasonlít a Stash Apply alkalmazáshoz, de a legfelső rejtettséget törli a veremből az alkalmazás után.

Tehát ez a pop olyan, mint a JS Array.pop () metódusa - Visszaadja a tömbből eltávolított elemet, és mutálja az eredeti tömböt. A "pop" zászló újra alkalmazza az utoljára mentett állapotot, és egyúttal törli a reprezentációját a Stash-en (más szóval: elvégzi az Ön számára a takarítást).

Hasonlóképpen, ha azt akarja, hogy egy adott stash megjelenjen, megadhatja a stash azonosítót.

git stash pop stash@{1}

git stash show

Ez a parancs a stash különbségek összefoglalását mutatja. A fenti parancs csak a legújabb stash-t veszi figyelembe.

Ha szeretné látni a teljes különbséget, használhatja

git stash show -p

Hasonlóan más parancsokkal, megadhatja a stash azonosítót is, hogy megkapja a diff összefoglalót.

git stash show stash@{1}

Git stash branch

Ez a parancs új ágat hoz létre a legújabb stash-el, majd törli a legutolsó stash-t (például a stash pop).

Ha szüksége van egy adott stash-re, megadhatja a stash azonosítóját.

git stash branch stash@{1}

Ez akkor lesz hasznos, ha ütközésekbe ütközik, miután a stash-t a branch legújabb verziójára alkalmazta.

git stash clear

Ez a parancs törli a repóban végrehajtott összes stash-t. Talán lehetetlen visszavonni.

git stash drop

Ez a parancs törli a legfrissebb stash-t a veremből. De óvatosan használja, talán nehéz visszaváltani (revert).

Megadhatja a stash azonosítót is.

git stash drop stash@{1}

Ha már nincs szüksége egy adott stash-re, akkor törölheti azt:

git stash drop <stash_id>

az összes stash-t törölheti a repóból a

$ git stash clear

Staging Area - Explanation-1

Legalapvetőbb magyarázata az, hogy csak azokat a fájlokat tudom a távoli repóba push-olni. És a fájlok staged-re állítása az, amikor git add fileName

Felhasználási eset - Tegyük fel, hogy csak a ../Git-and-Github/git-staging-area.md nevű fájl változtatásait akarom push-olni

1> Tehát a projekt gyökeréből ezt teszem

```
git add Git-and-Github/git-staging-area.md
```

2> Akkor csak azt a fájl lesz staged, és NEM MÁS FÁJL

3> Ezután futtatom a git push szokásos parancsait

```
git commit -m 'some message'
git push
```

És meg tudom nézni a távoli repót, a többi módosított fájlom nem került a helyi gépről a távoli repóba.

A legtöbb más verziókezelő rendszerrel 2 helyen lehet adatokat tárolni: a munkamásolatot (a jelenleg használt mappákat / fájlokat) és az adattárolót (ahol a verziókezelő dönti el, hogyan csomagolja és tárolja a módosításokat). A Git-ben van egy harmadik lehetőség: az átmeneti terület (vagy index). Ez alapvetően egy rakodóhely, ahol meg kell határoznia, milyen változásokat szállítanak el.

Ha megtesszük a git add. Fájlt, az hozzáad mindent, amit megváltoztattunk, vagy bármilyen új fájlt, amelyet még nem követtek nyomon. Az add parancs még nem tárolja az adatokat, egyszerűen csak a rakodótérre helyezi, és készen áll a következő git-elkötelezett teherautóra.

Staging Area - More Explanation

A fájl stage-be állítása egyszerűen előkészíti azt egy commit-ra. A Git az indexével lehetővé teszi, hogy az utolsó commit óta végrehajtott módosítások csak bizonyos részeit hajtsa végre. Tegyük fel, hogy két funkción dolgozik - az egyik elkészült, és az egyiknek még némi munkára van szüksége. Szeretne commit-ra és hazamenni (végül 5 óra!), De nem szeretné commit-olni a második funkció

egyes részeit, ami még nem történt meg. Beállítod azokat a részeket, amelyekről tudod, hogy az első funkcióhoz tartoznak, és commit-olod . Most commit-oltad a projektet az első funkcióval, míg a második még folyamatban van a munkakönyvtárban.

Tehát az átmeneti terület olyan, mint a fájlok gyorsítótára, amelyeket commit-olni akarsz:

Staging Area - Even More Explanation

A staging egy lépés a git-en a commit folyamat előtt. Vagyis a commit-ot gitben két lépésben hajtják végre: staging és tényleges commit.

Amíg egy változtatáskészlet az átmeneti területen (staging area) van, a git lehetővé teszi, hogy tetszés szerint szerkessze (helyettesítse az staging fájlokat az staging fájlok más verzióival, távolítsa el a változásokat stb.).

Vegyünk egy forgatókönyvet, amikor felhívja a költöztetőket, hogy a régi lakásokból az új lakásokba jussanak. Mielőtt ezt megtenné, átnézi a cuccait, eldönti, mit visz magával és mit dob el, zsákokba csomagolja és a fő folyosón hagyja. A költözők egyszerűen jönnek, a folyosóról előhozzák a (már becsomagolt) táskákat és elszállítják őket. Ebben a példában minden addig zajlik, amíg a költöztetők meg nem kapják a holmiját: ön dönti el, hogy mi merre tart, hogyan csomagolja és így tovább (pl. Dönthet úgy, hogy a holmijának felét eldobják, mielőtt a költöztetők odaérnek - ez része a színpadra állítás).

Technikai szempontból a staging a tranzakciós commit-ot is támogatja, azáltal, hogy az összes műveletet felosztja arra, hogy mi bukhat el (staginig) és mi nem bukhat (commit):

A gitben történő commit tranzakcióval valósul meg, miután a staging sikeres. A staging több lépése meghiúsulhat (például el kell végeznie, de a HDD-je 99,9999% -kal megtelt, és a git-nek nincs helye egy commit végrehajtására). Ez sikertelen lesz a staging-ben (a repository-t nem rontja meg egy részleges commit), és a staging folyamat nem befolyásolja a commit előzményeit (hiba esetén nem károsítja a repository-t).