Frontend programozás és tesztelés

Jegyzet a tananyaghoz

v4.0.0

Írta:
Ignácz Dominik Bence

April 4, 2023

Chapter 1

CSS

1.1 CSS alapok

A következő fejezetben arról lesz szó, hogy hogyan tudnuk stíluslapokat készíteni a HTML oldalunkhoz. A fejezet végére képesek leszünk bonyolultabb elrendezéseket is megvalósítani.

1.1.1 Felhasználása HTML-ben

Jelenleg 3 lehetőségunk van arra, hogy megadjuk a megfelelő stílust az elemeinknek

1.1.1.1 Inline megadás

Ebbben az esetben a HTML tagünk kapni fog egy `style` attribútumot és ezen keresztül állítjuk a saját tulajdonságait.

```
1 <p style="color:blue;">...</p>
```

1.1.1.2 Internal

Ez a megoldás már jóval nagyobb szabadságot ad, ugyanis nem csak egy adott elemet tudunk állítani, hanem az egész oldalt. Viszont hátránya, hogy a HTML kódunk jelentősen meg tud nőni tőle és könnyen átláthatatlanná válik.

Itt amit alkalmazni fogunk az egy `<style>` tag lesz és ezek között helyezzük el a kódot.

```
1 <style>
2   p {
3     color: blue;
4   }
5 </style>
```

1.1.1.3 Különálló fájl

A legszebb és leggyakoribb megoldás, hogy a teljes stíluskészletünket egy külön fájlba mozgatjuk, így a HTML kódunk is áttekinthetőbb, valamint biztosak lehetünk benne, hogy az új fájlban csak a stlius található meg.

Ezeket a fájlokat `.css` kiterjesztéssel mentve fogjuk használni. És ahhoz, hogy a HTML-ben használni tudjuk, a `<link>` tagre lesz szükségünk, amelyet a `<head>`-ben fogunk mindig elhelyezni.

```
1 <link href="style.css" rel="stylesheet" />
```

::: warning Figyelem! A `rel="stylesheet"` mindig szerepeljen a `<link>`-ben, ezzel fogja tudni a böngésző, hogy stíluslapként használja a megadott fájlt. :::

1.1.2 Box model

1.1.3 Szelektorok

1.1.4 Specificity

1.1.5 Ellenőrző kérdések

1.2 Színek

1.2.1 Beépített színek

1.2.2 Kód szerinti megadás

1.2.3 Színkeverés

1.2.4 Ellenőrző kérdések

1.3 Keretek

1.3.1 Ellenőrző kérdések

1.4 Árnyékok

1.4.1 Ellenőrző kérdések

1.5 Betűtípusok és szövegek

1.5.1 Ellenőrző kérdések

1.6 Listák

1.6.1 Ellenőrző kérdések

1.7 Túlcsordulás

1.7.1 Ellenőrző kérdések

1.8 Pseudo osztályok

1.8.1 Ellenőrző kérdések

1.9 Z-index

1.9.1 Ellenőrző kérdések

1.10 Pozicionálás

1.10.1 Ellenőrző kérdések

1.11 Szűrők és Blend mód

1.11.1 Ellenőrző kérdések

1.12 Táblázatok kezelése

1.12.1 Táblázat tulajdonságai

1.12.2 Cellák tulajdonságai

1.12.3 Ellenőrző kérdések

1.13 Flexbox

1.13.1 Mi az a flexbox?

Egy speciális szerkezet, amellyel különböző egydimenziós elrendezéseket valósíthatunk meg. Alapvetően sor vagy oszlop elrendezésbe tudjuk a benne található elemeket rendezni.

1.13.2 Mire érdemes használni?

Az egyik legalapvetőbb felhasználása a menük, menüsávok kialakítása. Ugyanakkor bármilyen azonos magasságú (és/vagy szélességű) sorok vagy oszlopok megvalósítására is alkalmas, ilyen például egy banner az oldalon amely egyik oldalt egy térképet, másik oldalán az leérhetőségeket mutatja.

1.13.3 Felépítése

Alapvetően két elemből áll.

- **Flex container:** Alap táruló elemünk lesz, ő fogja megkapni a `display: flex;` tulajdonságot.
- **Flex item:** Az összes olyan elem amit szeretnénk elrendezni a flexbox segítségével.

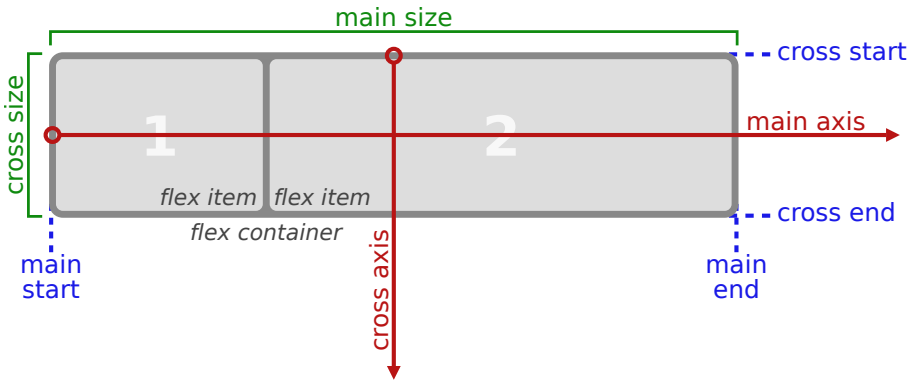

Figure 1.1: flex-direction-terms

1.13.3.1 Sor elrendezés

Alapvetően, ha külön nem adjuk meg, akkor egy sor elrendezést kapunk. Amennyiben explicit meg szeretnénk mondani, azt a `flex-direction: row;` segítségével tudjuk megtenni. A sorunknak a tengelyei a kövezkezők:

- **Main axis**/ **Főtengely**: Ez mindig a vízszintes tengely.
- **Cross axis**/ **Kereszttengely**: Ez mindig a függőleges tengely.

```
1 .row{
2     display: flex;
3     flex-direction: row;
4 }
```

1.13.3.2 Oszlop elrendezés

Ebben az esetben már direktben meg kell mondanunk a `flex-direction: column;` segítségével, hogy változzon az elrendezésünk. Viszont ezzel együtt figyelni kell rá, hogy a tengelyeink is fordulnak.

- **Main axis/ Főtengely:** Ez a függőleges tengelyünk lesz.
- **Cross axis / Kereszttengely:** Ez pedig a vízszintes tengellyé válik

```
1 .row{
2     display: flex;
3     flex-direction: column;
4 }
```

1.13.4 A tároló további tulajdonságai

1.13.4.1 Tördelés

Amennyiben, az elemeink nem férnek el egy sorban beállíthatjuk, hogy az oldalunkon több sorban jelenjenek meg. Ezt a `flex-wrap` tulajdonsággal lehet állítani. Amely a következő értékekkel rendelkezhet:

Érték	Leírás
`nowrap`	Tördelés nélkül egysorban vannak az elemek
`wrap`	Több sorba tördelés
`wrap-reverse`	Több sorba tördelés, de fordított irányban

1.13.4.2 Összevonás (Elrendezés és Tördelés)

A tároló elemünk tulajdnoságait a `flex-flow` tulajdonság segítségével tudjuk összevonni, mégpedig úgy, hogy első értéke az elrendzés lesz, a második értéke pedig a tördelés

```
1 div { flex-flow: row; }
```

Figure 1.2: flex-flow1

```
1 div { flex-flow: column wrap; }
```

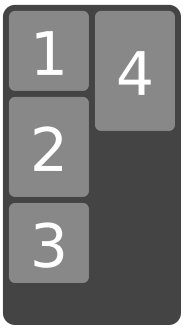

Figure 1.3: flex-flow2

```
1 div { flex-flow: row-reverse wrap-reverse; }
```

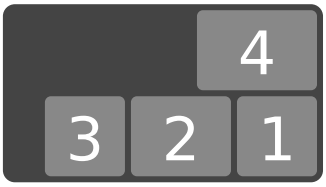

Figure 1.4: flex-flow3

1.13.5 Az elemek tulajdonságai

1.13.5.1 Sorbarendezés

Amennyiben szeretnénk, hogy a HTML struktúra helyett a flexboxunkban más sorrendben legyenek elemeink (pl: egy reszponzív weboldalon lehezt, hogy más sorrendben kell összecsúszniuk), abban az esetben az `order` tulajdonság segítségével tudjuk megtenni. Az `order` számára bármilyen egész szám megadható és a legkisebbtől a legnagyobbig fogja egymás után pakolni az elemeket a böngésző.

1.13.5.2 Növekedés

Előfordulhat, hogy szeretnénk, hogy egy elemünk nagyobb területet foglalhasson el a teljes boxból. Ekkor a `flex-grow` segítségével megadható az a tényezhő, hogy a többi elemhez képest mekkora helyet foglalhat el.

::: warning Figyelem! A `flex-grow` negatív számot **nem** kaphat és alapértéke a 0. :::

1.13.5.3 Zsugorodás

Az előző alapján a `flex-shrink` azt állítja nekünk, hogy egy elem mennyire zsugorodhat össze a többi elemhez viszonyítva.

:::warning Figyelem! Ő **sem** kaphat negatív számot, azonban alapértéke neki 1. :::

1.13.5.4 Alapméret

Amennyiben szeretnénk egy elem számára konkrét méretet adni (pl: 200px) azt a `flex-basis` segítségével tehető meg.

1.13.5.5 Összevonás (Növekedés, zsugorodás, Alapméret)

Lehetőségünk van a `flex-grow`, a `flex-shrink` és a `flex-basis` összevonására (**ebben a sorrendben**) a `flex` tulajdonság alatt.

1.13.6 Igazítások

1.13.7 `align-content`

A kereszttengely mentén igazítjuk a boxunk tartalmát. (sor -> függőleges | oszlop -> vízszintes) Értékei a következők lehetnek:

Érték	Leírás
`flex-start`	A tároló elejére helyezi az elemeket.
`center`	A tároló közepére helyezi az elemeket.
`flex-end`	A tároló végére helyezi az elemeket.
`space-between`	Minél meszebb próbálja helyezni az elemeket egymástól.
`space-around`	Az elemek körül arányos távolságot tart.
`stretch`	Az elemek kitöltik a teljes tárolót.

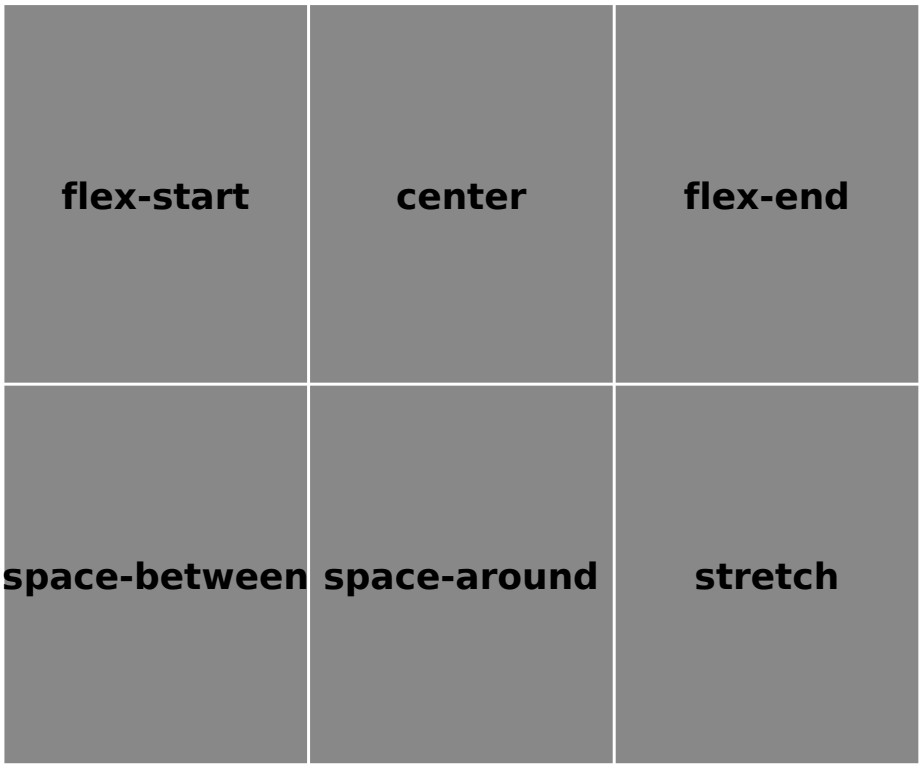

Figure 1.5: Align content example

1.13.8 `align-self`

Azt állíthatjuk be, hogy az adott elem hol jelenjen meg kereszttengely mentén a tárolóban. **Az előző kettővel szemben, ezt nem a tarolóra, hanem az adott elemre kell beállítani!!!** Értékei:

Érték	Leírás
`flex-start`	A tároló elejére helyezi az elemeket.
`center`	A tároló közepére helyezi az elemeket.
`flex-end`	A tároló végére helyezi az elemeket.
`stretch`	Az elemek kitöltik a teljes tárolót.
`baseline`	Az alapvonahoz igazít a legnagyobb távolság alapján a kereszt irányú kezdővonaltól.

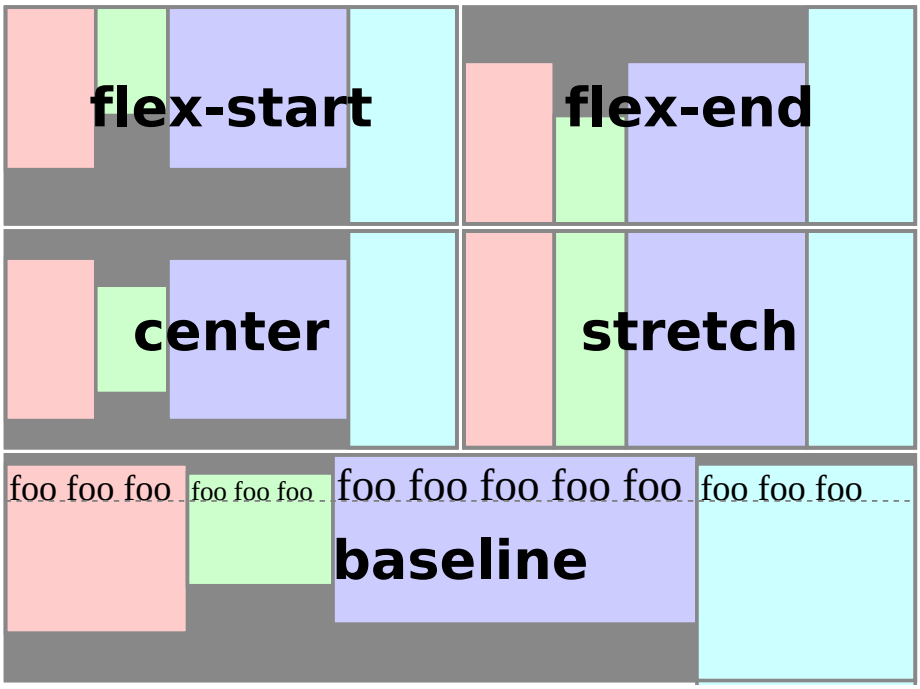

Figure 1.6: Flex align

1.13.9 `justify-content`

Az `align`-okkal szemben itt a főtengelyen állítjuk a boxunk tartalmának elhelyezését. Értékei a következők lehetnek:

Érték	Leírás
`flex-start`	A tároló elejére helyezi az elemeket.
`center`	A tároló közepére helyezi az elemeket.
`flex-end`	A tároló végére helyezi az elemeket.
`space-between`	Minél meszebb próbálja helyezni az elemeket egymástól.
`space-around`	Az elemek körül arányos távolságot tart.
`stretch`	Az elemek kitöltik a teljes tárolót.

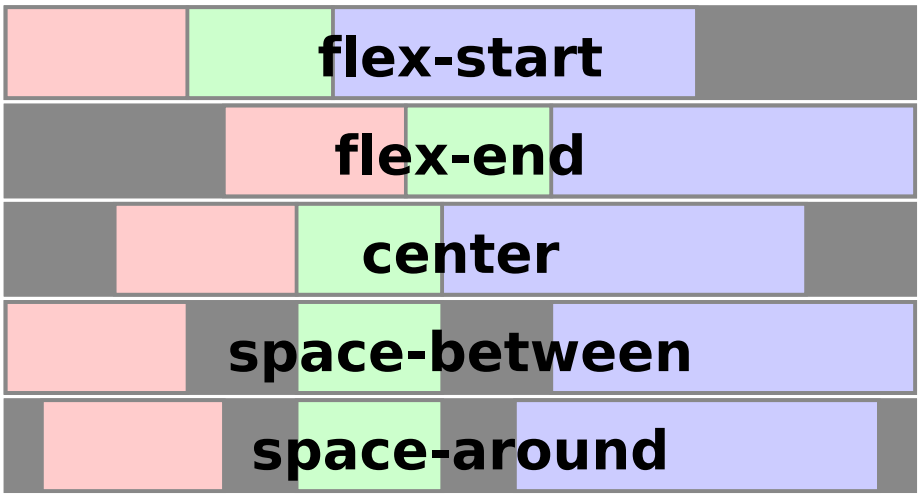

Figure 1.7: Flex pack

1.13.10 `justify-self`

Beállíthatjuk a főtengely mentén, hogy az adtott eleme hol jelenjen meg. **Az előző kettővel szemben, ezt nem a tarolóra, hanem az adott elemre kell beállítani!!!** Értékei:

Érték	Leírás
`flex-start`	A tároló elejére helyezi az elemeket.
`center`	A tároló közepére helyezi az elemeket.
`flex-end`	A tároló végére helyezi az elemeket.
`stretch`	Az elemek kitöltik a teljes tárolót.
`baseline`	Az alapvonahoz igazít a legnagyobb távolság alapján a kereszt irányú kezdővonaltól.

1.13.11 Összevonás: `place-content`

Az `align-content` és a `justify-content` összevonása

1.13.12 Összevonás: `place-self`

Az `align-self` és a `justify-self` összevonása

1.13.13 Ellenőrző kérdések

1.14 Grid

```
1 .container{
2   display:grid;
3 }
```

1.14.1 Szerkezet kialakítása

```
1 .container {
2   grid-template-columns: 40px 50px auto 50px 40px ;
3   grid-template-rows: 25% 100px auto ;
4 }
```

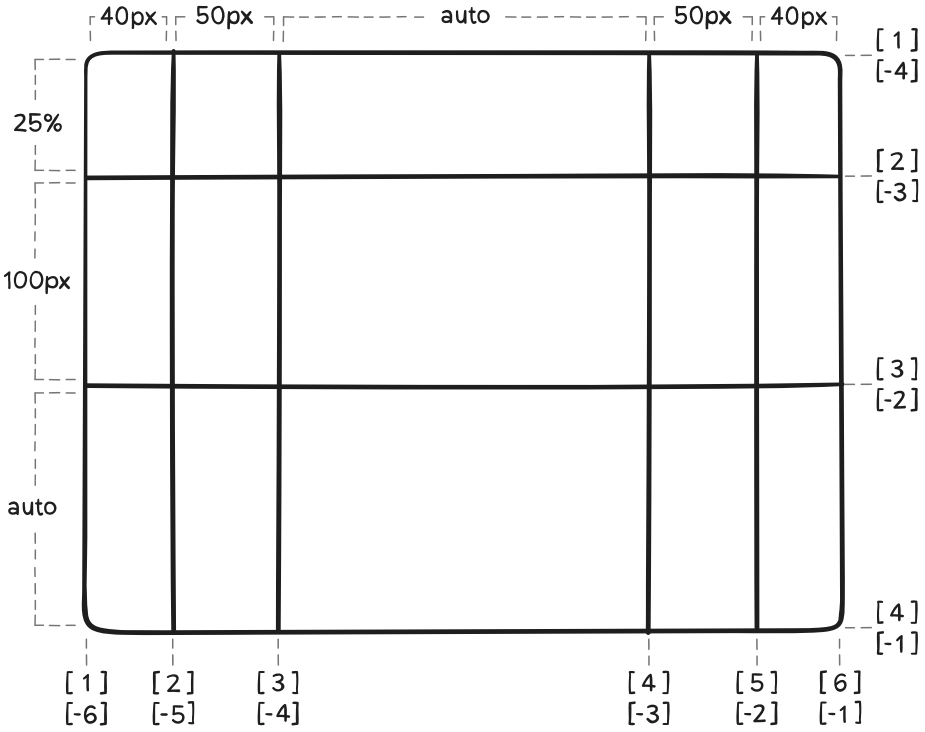

Figure 1.8: Grid sablon

1.14.1.1 Elnevezett sorok/oszlopok

```
1 .container {
2   grid-template-columns: [first] 40px [line2] 50px
       [line3] auto [col4-start] 50px [five] 40px
       [end];
3   grid-template-rows: [row1-start] 25% [row1-end]
       100px [third-line] auto [last-line];
4 }
```

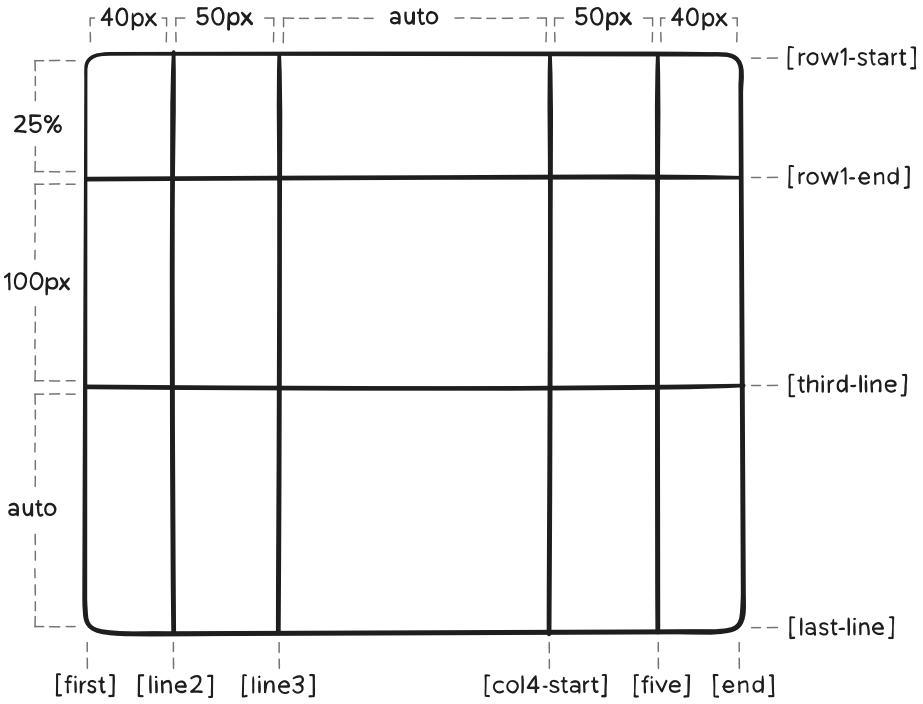

Figure 1.9: Elnevezett sorok/oszlopok

1.14.1.2 Területek

```
.container {
  display: grid;
  grid-template-columns: 50px 50px 50px 50px;
  grid-template-rows: auto;
  grid-template-areas:
    "header header header header"
    "main main . sidebar"
    "footer footer footer footer";
}

.item-a {
  grid-area: header;
}
.item-b {
  grid-area: main;
}
.item-c {
  grid-area: sidebar;
}
.item-d {
  grid-area: footer;
}
```

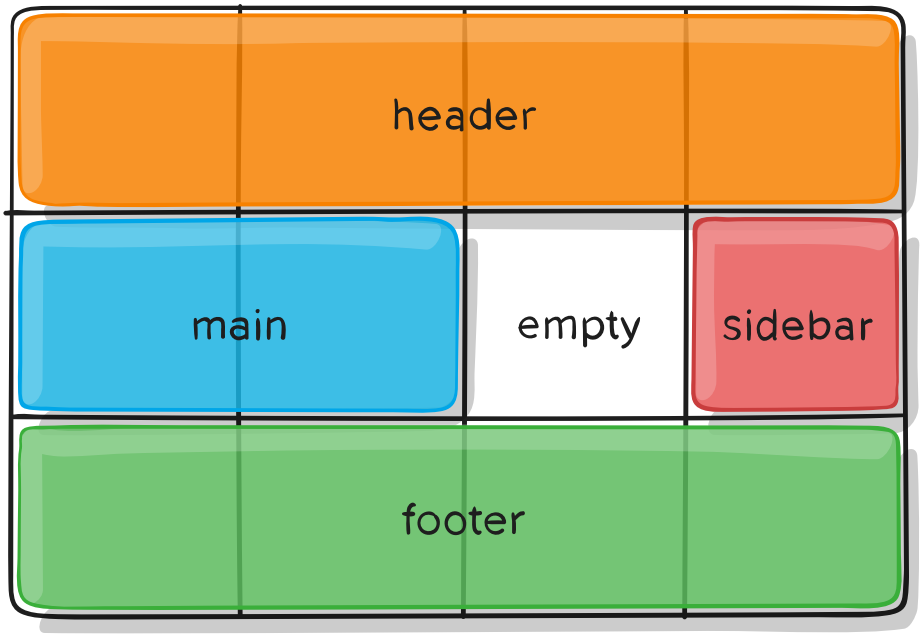

Figure 1.10: Gird területek

1.14.1.3 Sor/Oszlop közök

A közök megadásánál először a sorok, majd pedig az oszlopok közötti távolságot adjuk meg.

```
1 .container{
2   gap: 15px 10px;
3
4   row-gap: 15px;
5   column-gap: 10px;
6
7 }
```

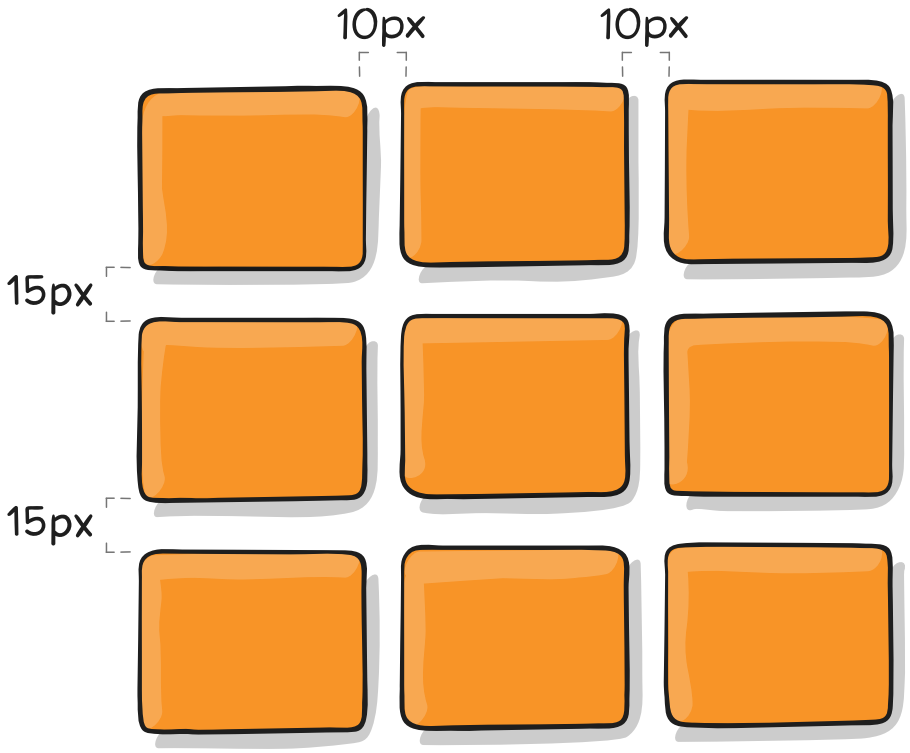

Figure 1.11: Grid gap

1.14.2 Elrendezések

Hasonló a sor elrendezésű flexbox-hoz

1.14.2.1 Align-items

A cellában levő elemek függőleges igazítása

1.14.2.1.1 start

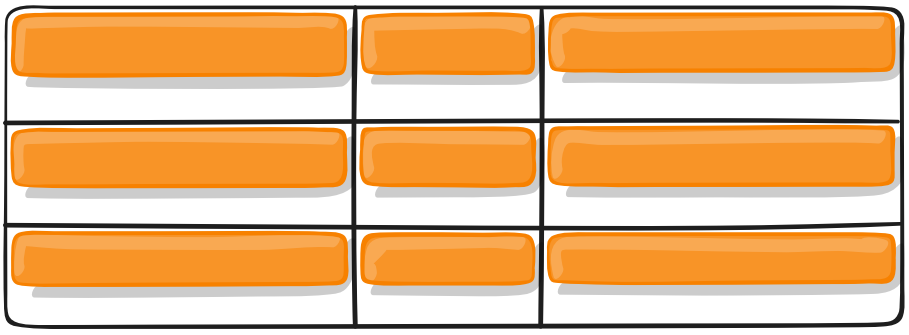

Figure 1.12: `align-items:start`

1.14.2.1.2 center

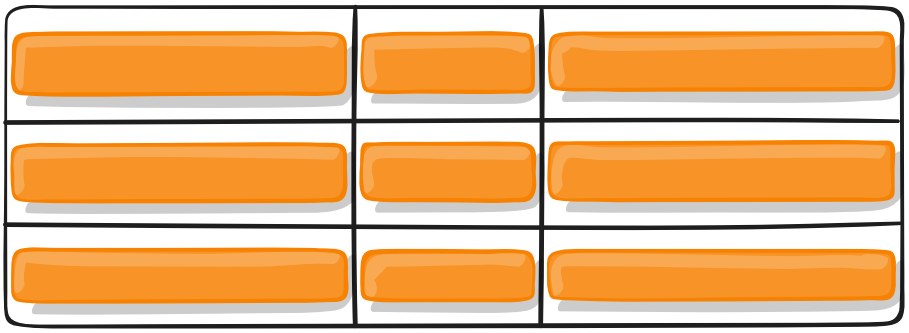

Figure 1.13: `align-items: center`

1.14.2.1.3 end

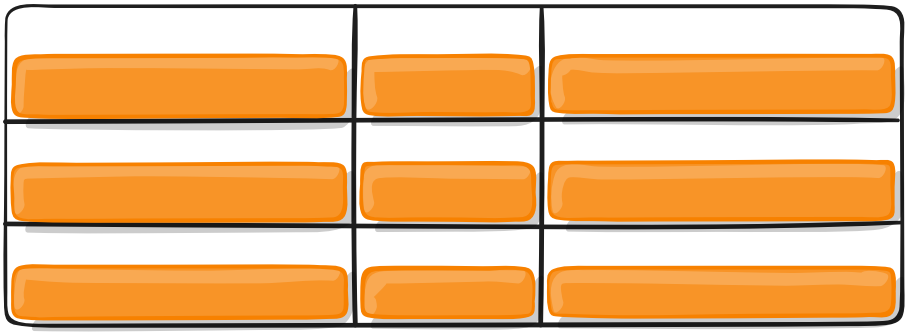

Figure 1.14: `align-items: end`

1.14.2.1.4 stretch

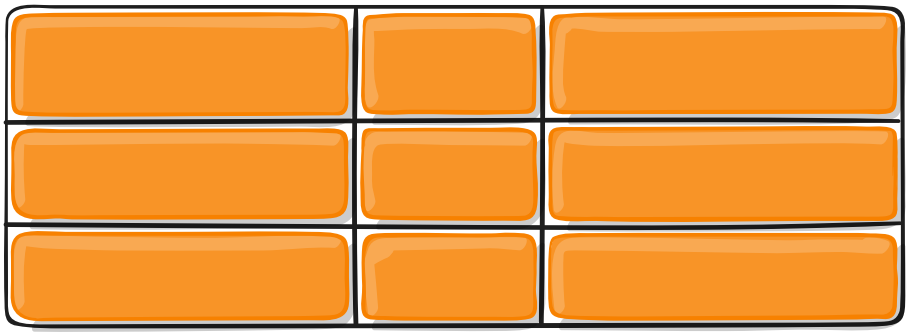

Figure 1.15: `align-items: stretch`

1.14.2.2 Align-content

A belső rácsszerkezet függőleges igazítása a container-hez képest

1.14.2.2.1 start

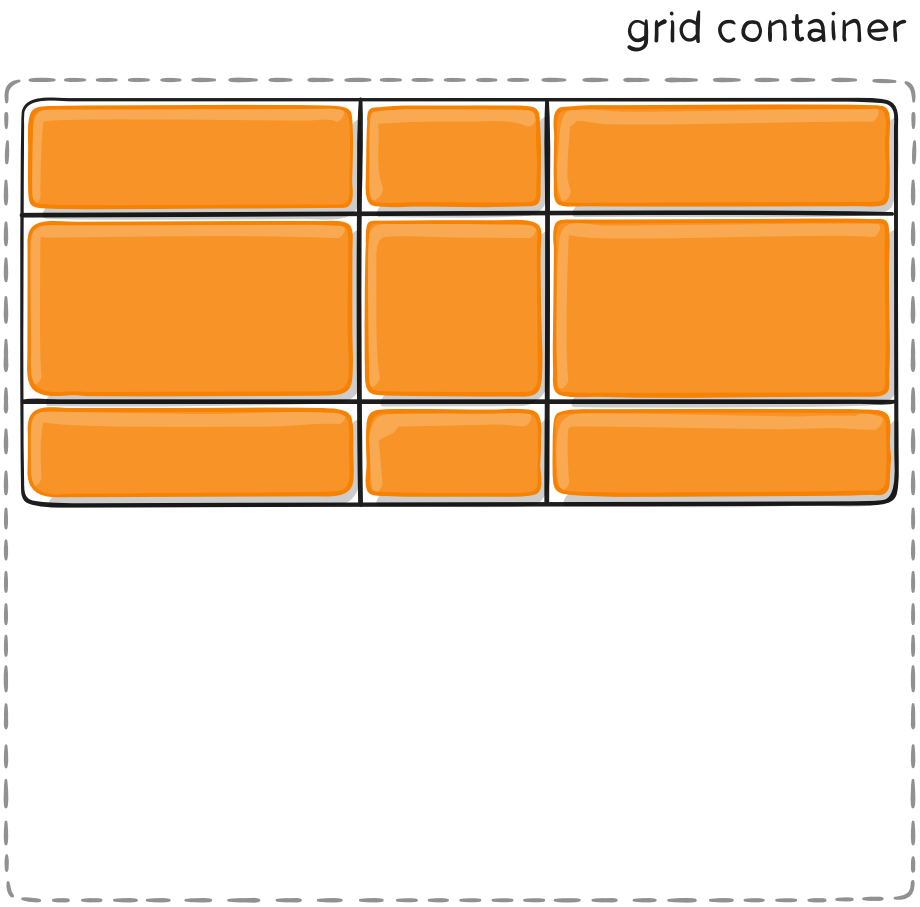

Figure 1.16: `align-content: start`

1.14.2.2.2 center

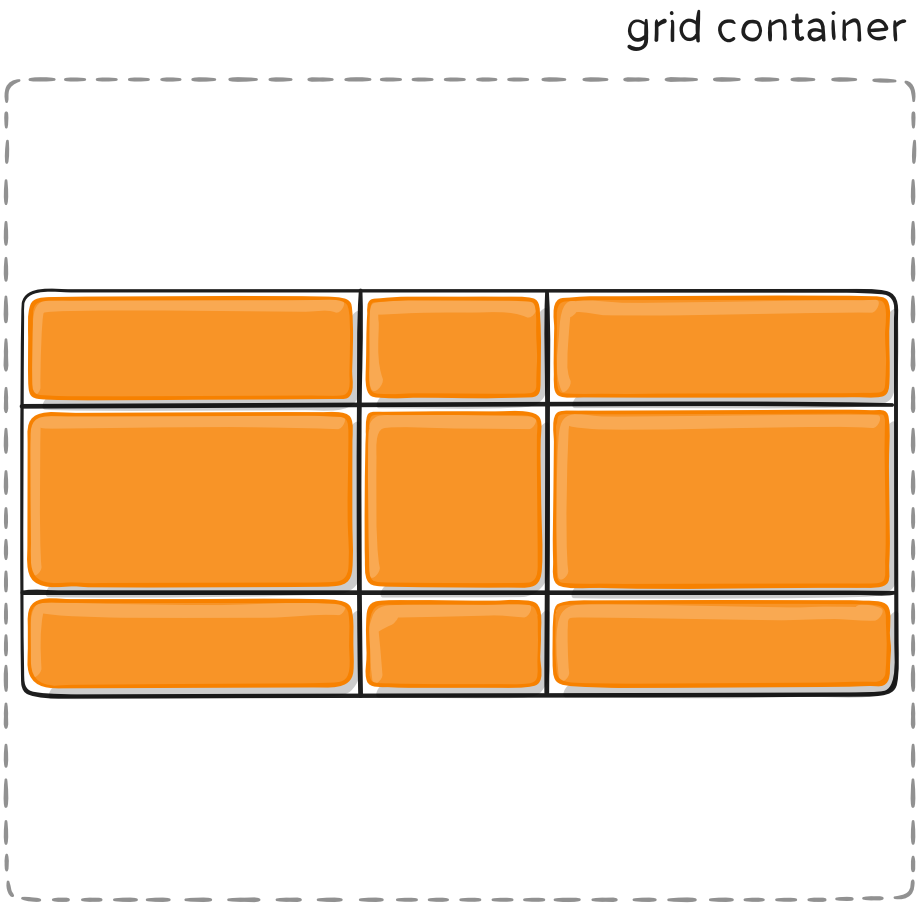

Figure 1.17: `align-content: center`

1.14.2.2.3 end

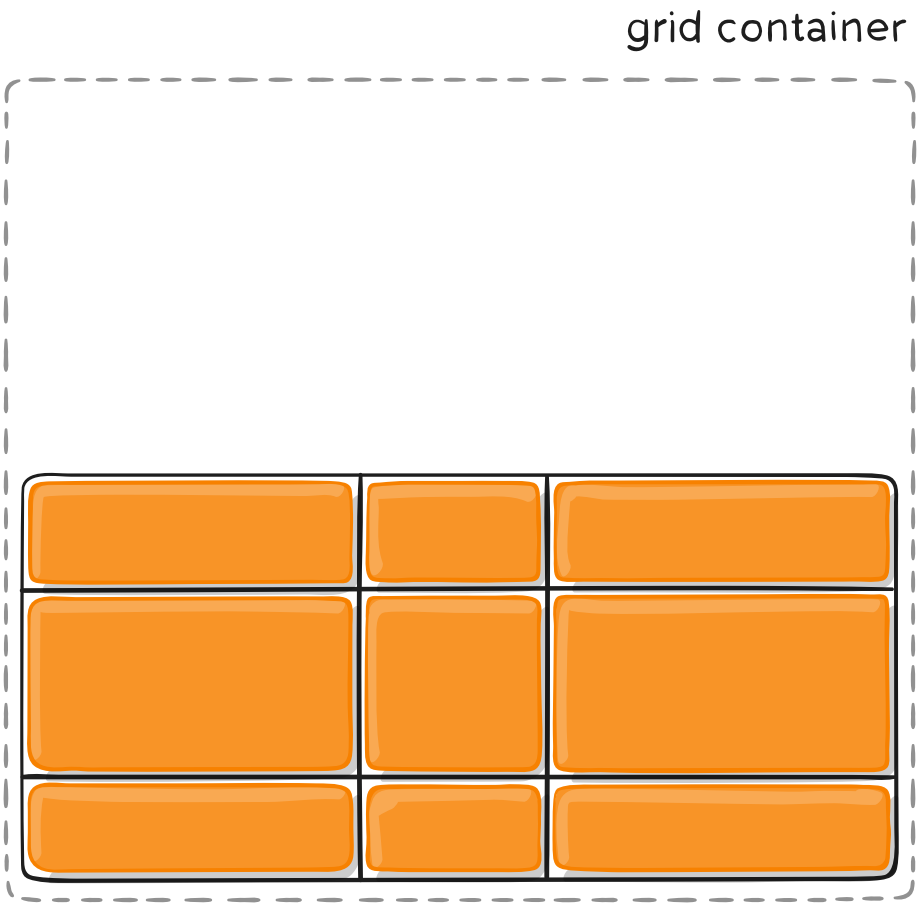

Figure 1.18: `align-content: end`

1.14.2.2.4 space-between

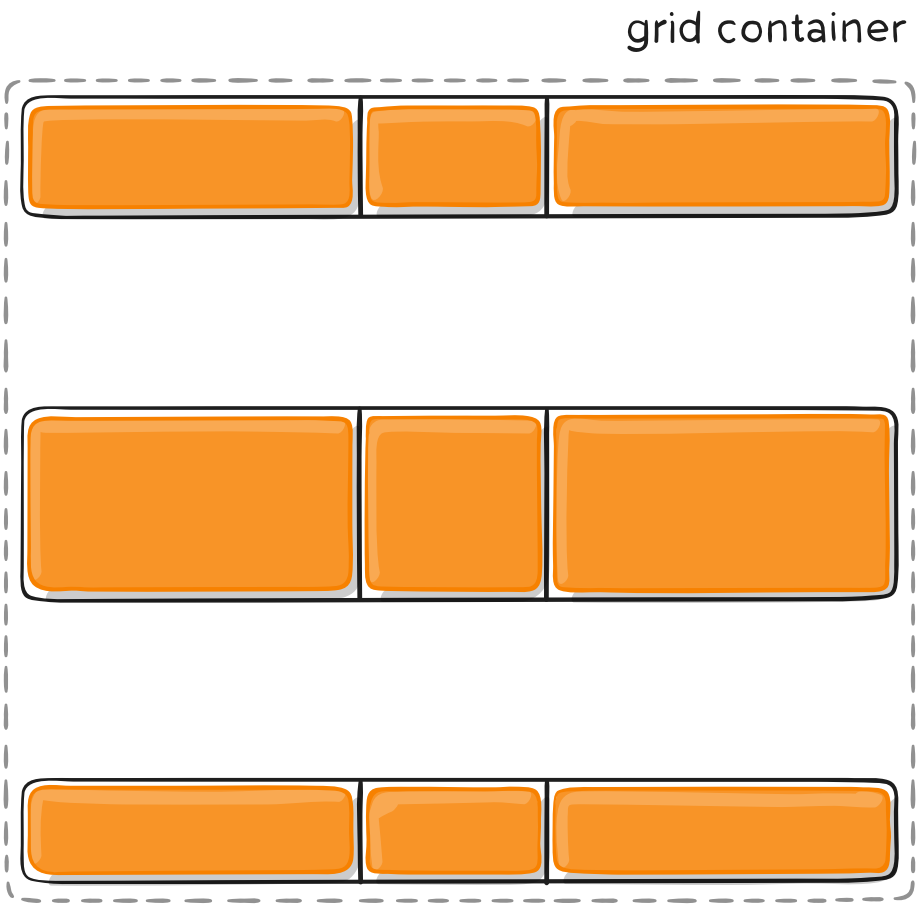

Figure 1.19: `align-content: space-between`

1.14.2.2.5 space-around

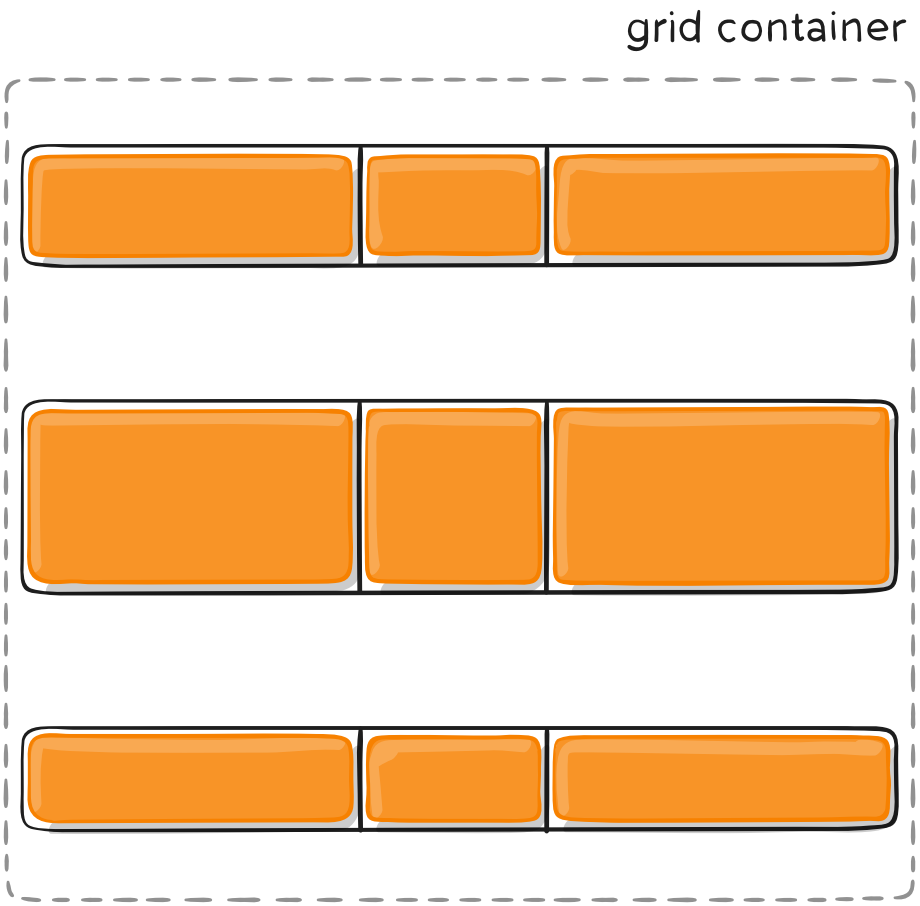

Figure 1.20: `align-content: space-around`

1.14.2.2.6 stretch

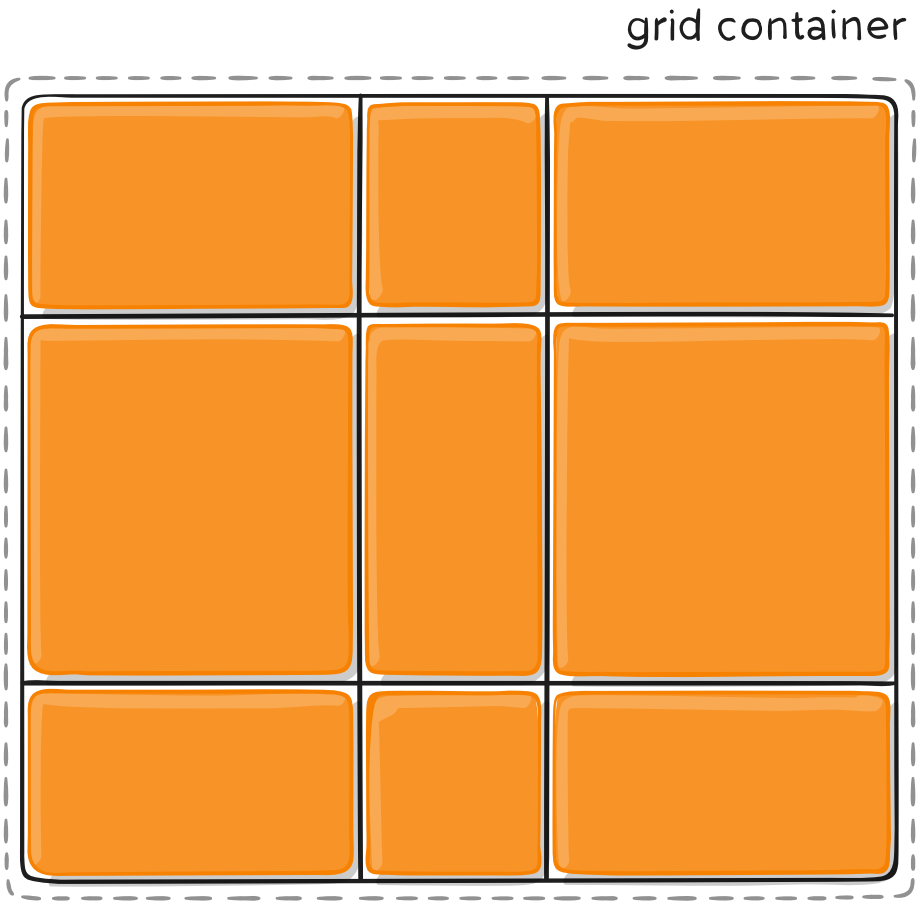

Figure 1.21: `align-content: stretch`

1.14.2.3 Justify-items

A cellában levő elemek vízszintes igazítása

1.14.2.3.1 start

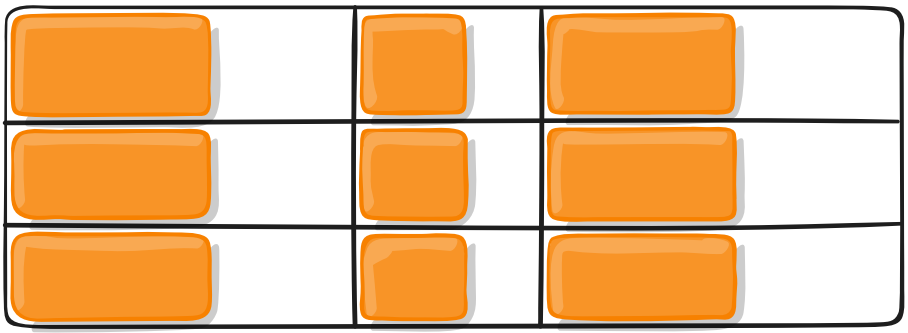

Figure 1.22: `justify-items: start`

1.14.2.3.2 center

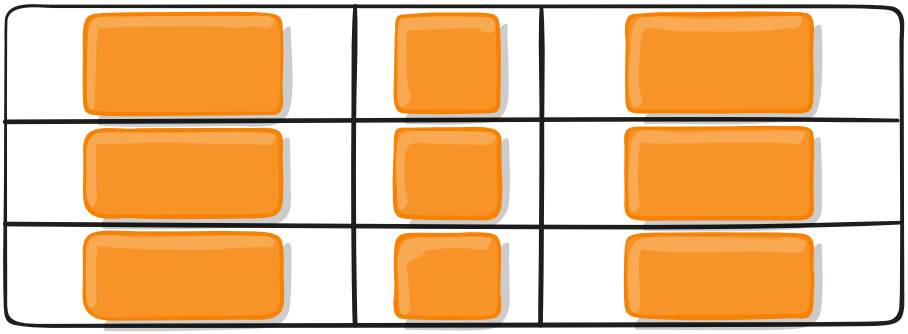

Figure 1.23: `justify-items: center`

1.14.2.3.3 end

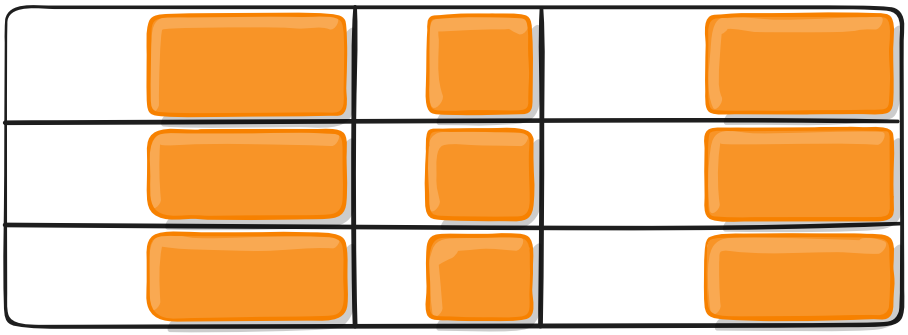

Figure 1.24: `justify-items: end`

1.14.2.3.4 stretch

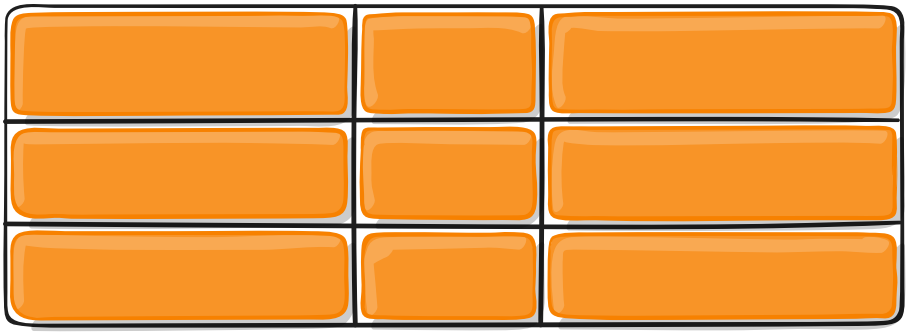

Figure 1.25: `justify-items: stretch`

1.14.2.4 Justify-content

A belső rácsszerkezet vízszintes igazítása a container-hez képest

1.14.2.4.1 start

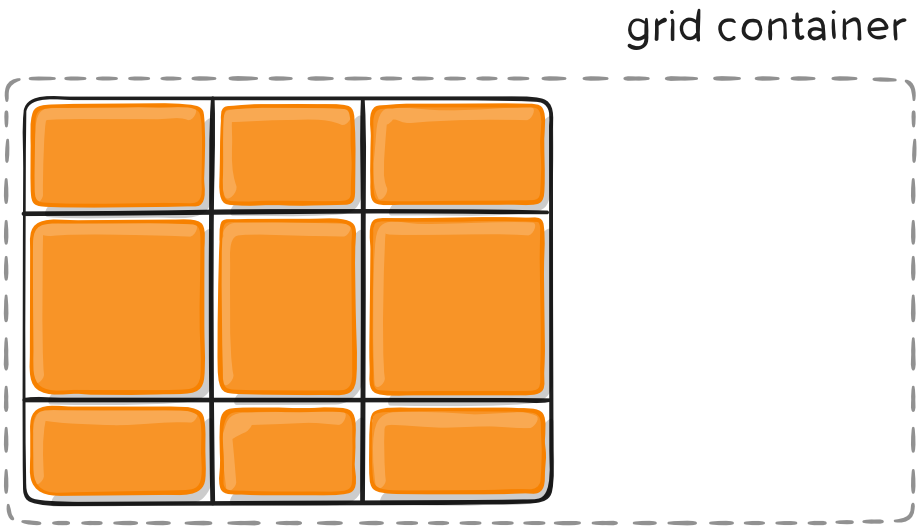

Figure 1.26: justify-content: start‘

1.14.2.4.2 center

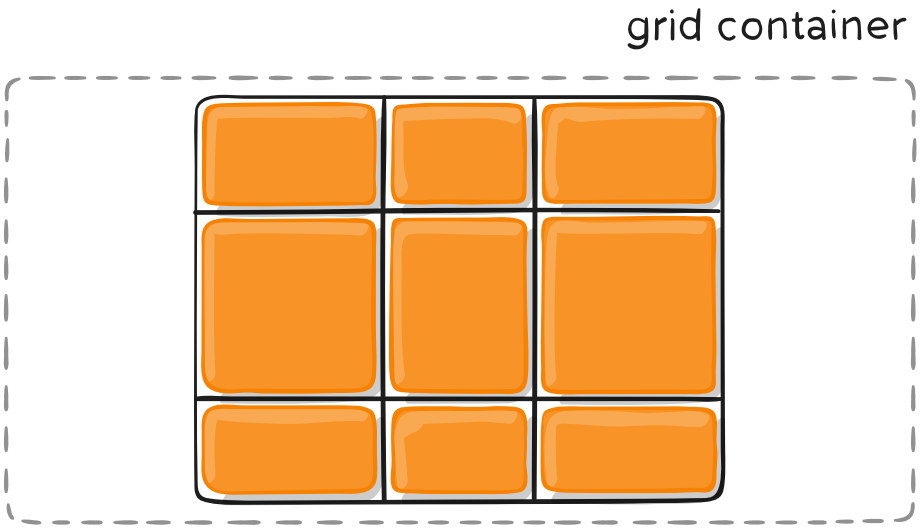

Figure 1.27: `justify-content: center`

1.14.2.4.3 end

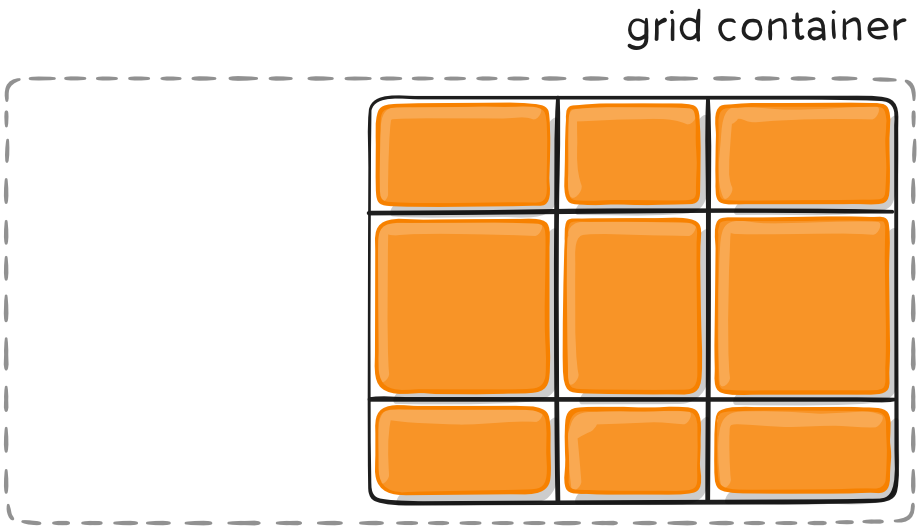

Figure 1.28: `justify-content: end`

1.14.2.4.4 space-between

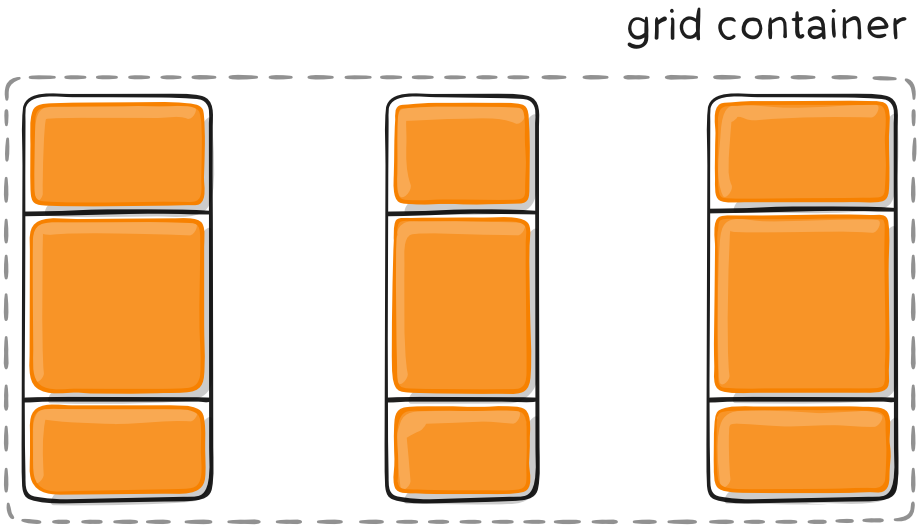

Figure 1.29: ‘justify-content: space-between

1.14.2.4.5 space-around

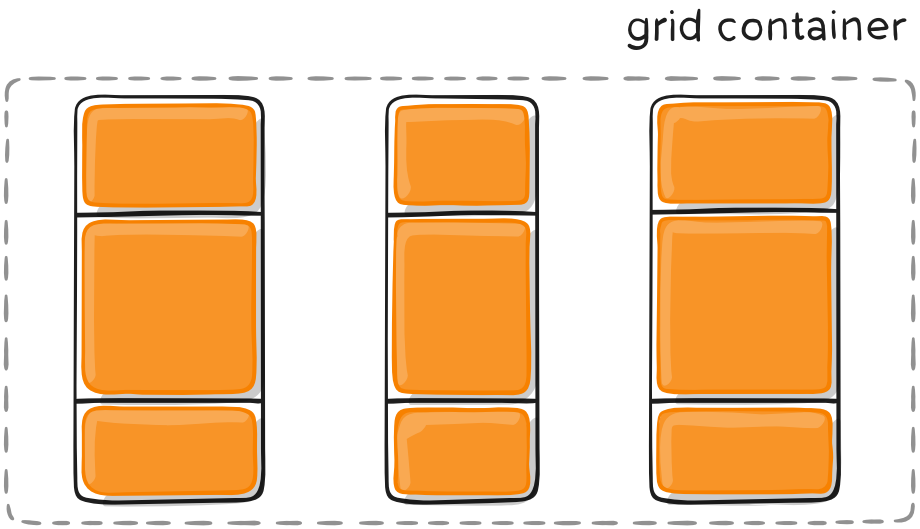

Figure 1.30: ‘justify-content: space-around

1.14.2.4.6 stretch

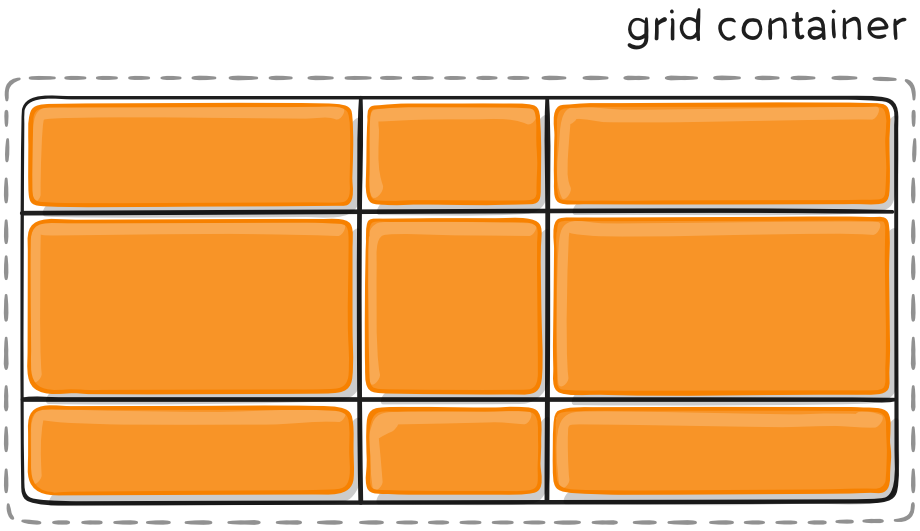

Figure 1.31: `justify-content: stretch`

1.14.3 Grid elemek

1.14.3.1 Elhelyezés

1.14.3.1.1 Sorban

```
.item-1{
  grid-row-start: 1;
  grid-row-end: 2;

  grid-row: 1/2; /* start/end */

  grid-row: elso-sor/masodik-sor;

  grid-row: 1/ span 3; /* span azt jelzi, hogy hany
     darab soron at tartson */

}
```

1.14.3.1.2 Oszlopban

```
.item-1{
  grid-column-start: 1;
  grid-column-end: 2;

  grid-column: 1/2; /* start/end */

  grid-column: elso-sor/masodik-sor;

  grid-column: 1/ span 3; /* span azt jelzi, hogy
     hany darab oszlopon at tartson */

}
```

1.14.3.1.3 Példa a sor/oszlop alapú elhelyezésre

```
.item-a {
  grid-column-start: 2;
  grid-column-end: five;
  grid-row-start: row1-start;
  grid-row-end: 3;
}
```

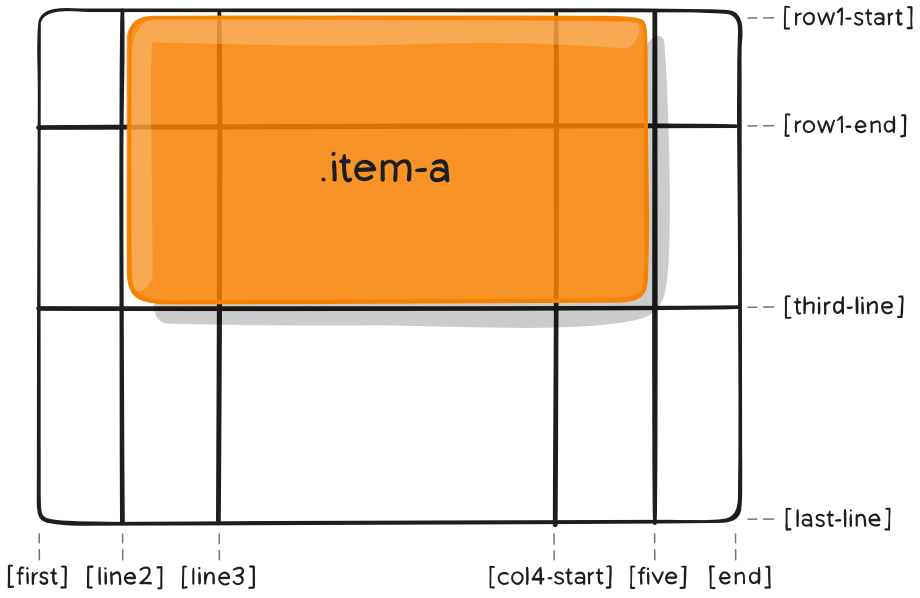

Figure 1.32: grid column row start end 1

```
1 .item-b {
2   grid-column-start: 1;
3   grid-column-end: span col4-start;
4   grid-row-start: 2;
5   grid-row-end: span 2;
6 }
```

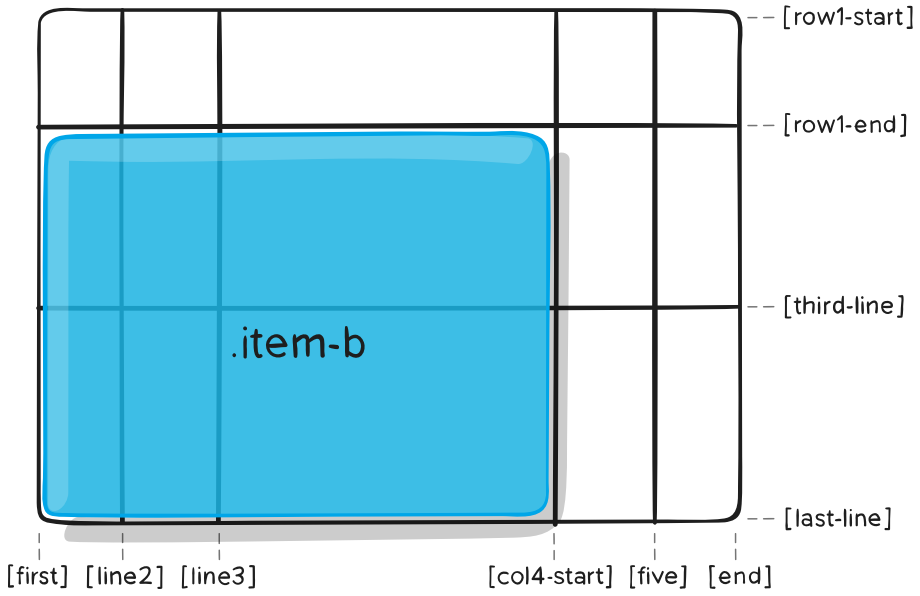

Figure 1.33: Grid column row start end 2

```
1 .item-c {
2   grid-column: 3 / span 2;
3   grid-row: third-line / 4;
4 }
```

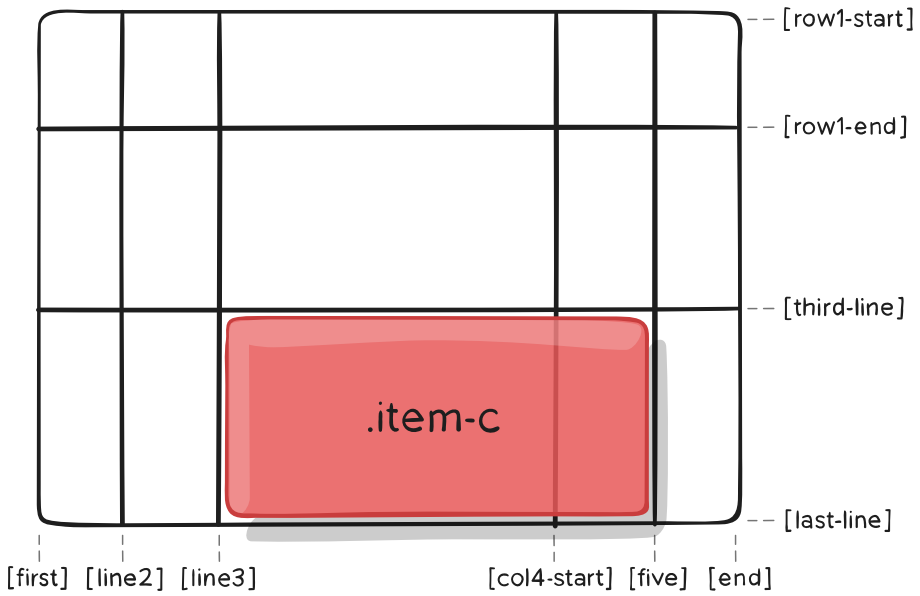

Figure 1.34: Grid column row

1.14.3.1.4 Területen

```
1 .item-d {
2
3   grid-area: header;
4
5   grid-area: 1 / col4-start / last-line / 6; /*
       row-start / column-start / row-end /
       column-end*/
6 }
```

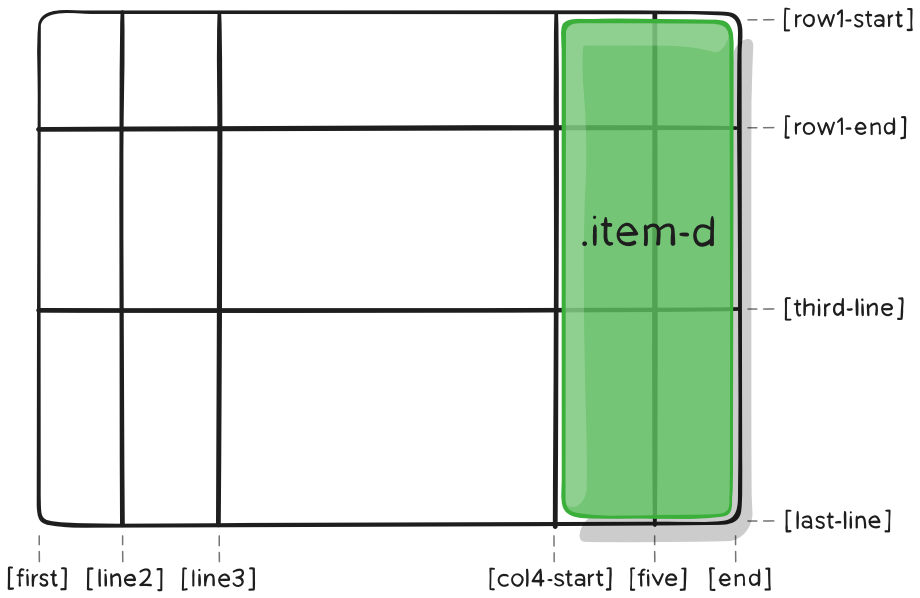

Figure 1.35: grid-area

1.14.3.1.5 Align-self

Függőlegesen helyezi el az adott elemet

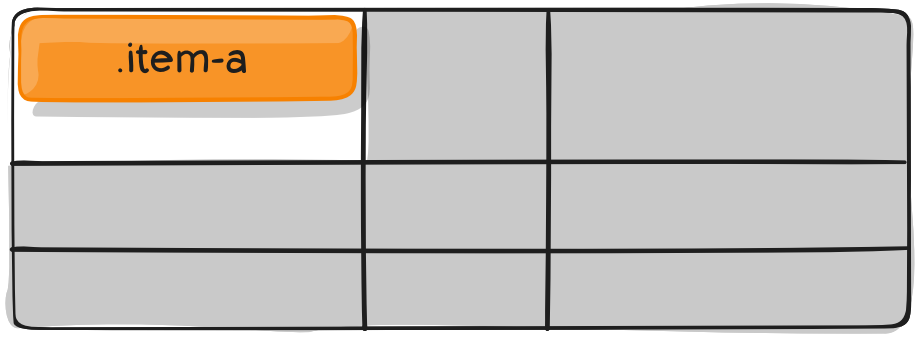

Figure 1.36: Align self start

start

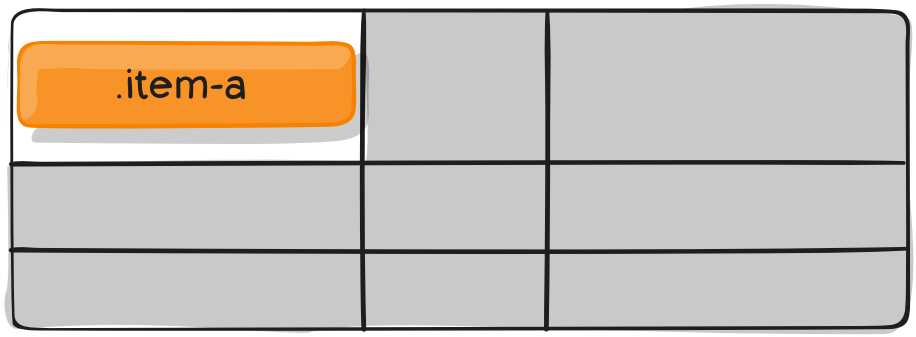

Figure 1.37: Align self center

center

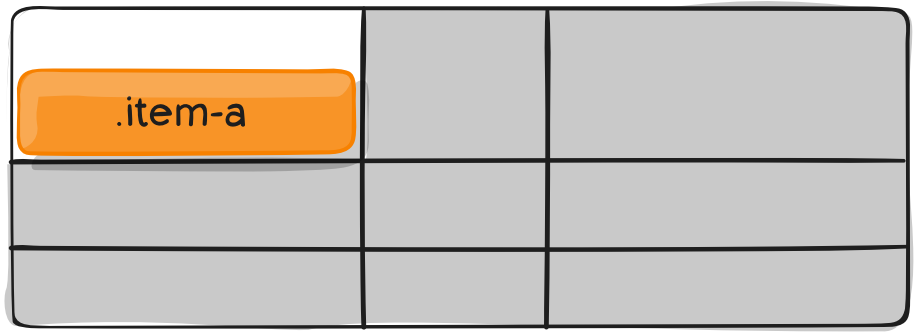

Figure 1.38: Align self end

end

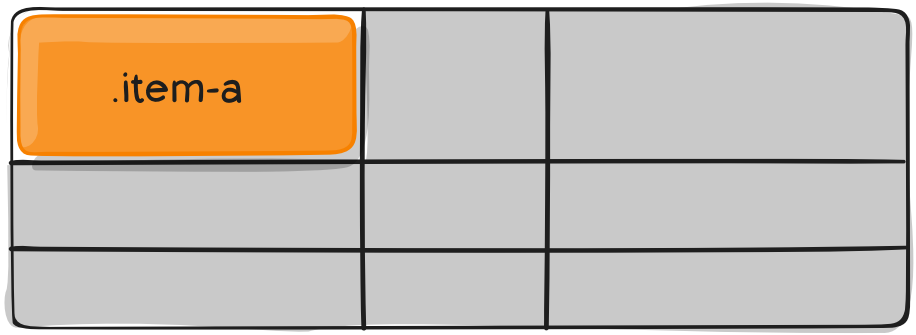

Figure 1.39: Align self stretch

stretch

1.14.3.1.6 Justify-self

Vízszintesen helyezi el az adott elemet

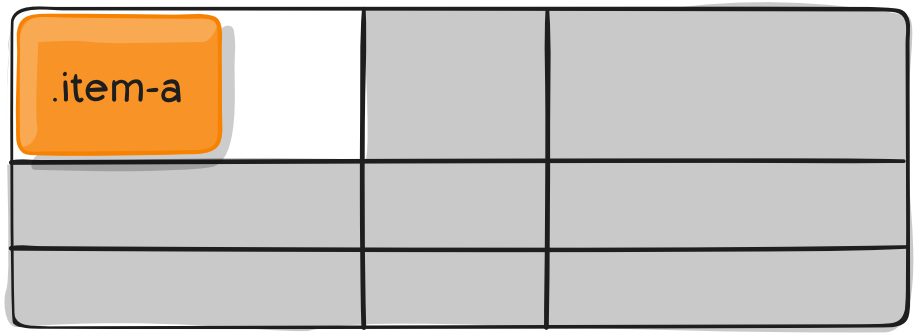

Figure 1.40: Justify self start

start

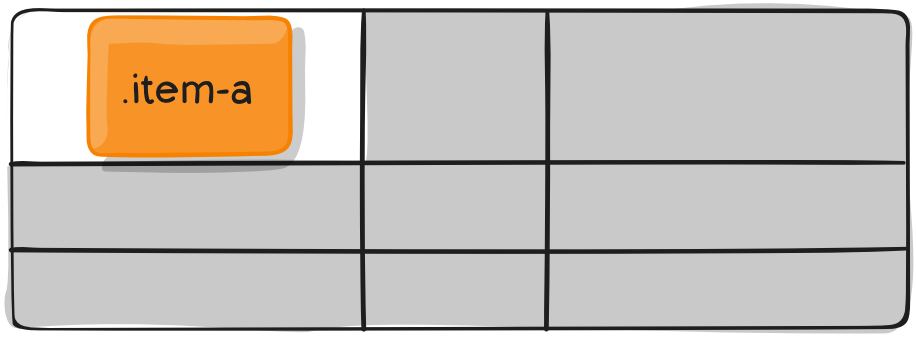

Figure 1.41: Justify self center

center

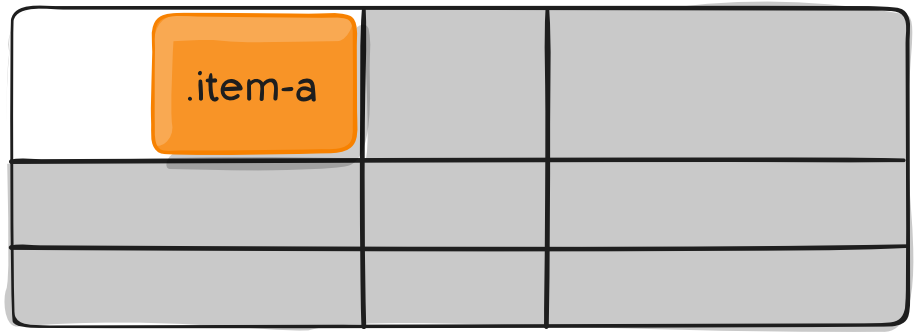

Figure 1.42: Justify self end

end

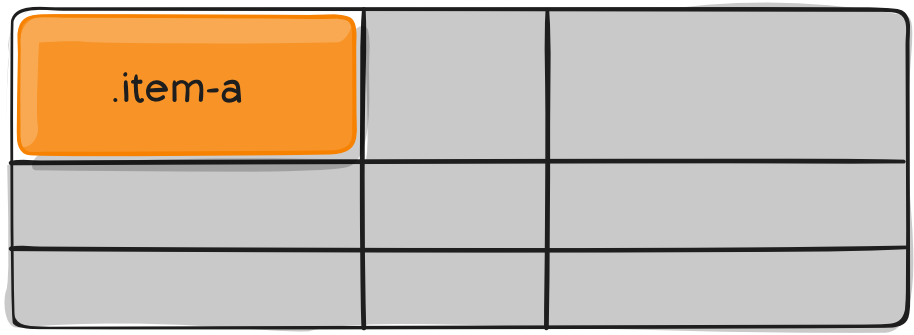

Figure 1.43: Justify self stretch

stretch

1.14.4 Függvények

1.14.4.1 minmax()

Megadhatjuk vele, hogy az adott sor/oszlop minimum mekkora és maximum mekkora lehet

```
1 grid-template-columns: minmax(100px, 1fr) 3fr;
```

1.14.4.2 repeat()

Ha több azonos méretű sort/oszlopot akarunk definiálni, célszerű ezt a függvényt, használni.

Először megadjuk neki, hogy hány darabot, majd, hogy mekkora legyen az adott sor/oszlop.

```
1 grid-template-columns:
2   1fr 1fr 1fr 1fr 1fr 1fr 1fr 1fr;
3
4 /* egyszerubb megoldas: */
5 grid-template-columns:
6   repeat(8, 1fr);
```

1.14.5 Ellenőrző kérdések

1.15 Media query

A CSS lehetőséget ad számunkra, hogy a böngésző mérete, beállítása vagy környezete alapján az oldalunk másképp jelenjen meg azaz reszponzív legyen.

Az `@media` lesz az aminek a segítségével meg tudjuk határozni, azt ami alapján változtatni szeretnénk. Ezt követi a környezet megadása, ami lehet:

- `screen`: Ha kijelzőn tekintjük meg az oldalt
- `print`: Ha nyomtatni szeretnénk az oldalt
- `all`: Minden típus egyben

Ezeket elláthatjuk olyan jellzőkkel amelyek szűkítik a körv, vagy negálják. Az alábbiakat a megfelelő típus elé írva tudjuk felhasználni.

- `not`: Minden kivéve a jelzett típus
- `only`: Csak és kizárólag a jelzett típus

Mindezek után különböző logikai műveletek segítségével tudjuk tovább pontosítani.

Például, ha szeretnénk mobil eszközre optimailálni az oldalunkat, akkor azt így tehetjük meg:

```
1 @media screen and (width <= 480px){
2   ...
3 }
```

1.15.1 Desktop First

1.15.2 Mobile First

1.15.3 Ellenőrző kérdések

1.16 Színátmenetek

1.16.1 Lineáris

1.16.2 Radiális

1.16.3 Conic

1.16.4 Ellenőrző kérdések

1.17 Átmenetek

1.17.1 Ellenőrző kérdések

1.18 Animációk

1.18.1 Ellenőrző kérdések

1.19 Transzformációk

1.19.1 Ellenőrző kérdések

Chapter 2

Bootstrap

2.1 Bootstrap alapok

2.1.1 Ellenőrző kérdések

Chapter 3

SASS

3.1 Sass

3.1.1 Változók

3.1.2 Egymásba ágyazás

3.1.3 Modulok

3.1.4 Mixinek

3.1.5 Kiterjesztés

3.1.6 Operátorok

3.1.7 Ellenőrző kérdések

Chapter 4

ECMAScript

4.1 ECMAScript alapok

4.1.1 Mi az az ECMAScript?

Az ECMAScript tulajdonképpen nem más, mint a JavaScript hivatalos megnevezése, miután az ECMA szabványosította ’97-ben. 2015-ig az egyes verziók verziószámmal voltak ellátva. Az utolsó nagy verziózott frissítés 2015-ben érkezett JavaScript néven. Ez volt az első olyan frissítés ami egy új irányba tolta el a nyelv fejlődését és már képes a korábbi könyvtárak funkcióit önállóan is ellátni.

2016-tól viszont az évszámot kapta meg, mint verziószám. Jelenleg az ES2022 a legfrisszebb verzió, amely június óta a hivatalos standard. A tananyag igyekszikezt a verziót követni.

::: info Információ A tananyagban a továbbiakban **ES** néven fogok a nyelvre hivatkozni. :::

4.1.2 Hogyan használjuk a HTML oldalunkon?

Az oldalunkra két lehetőségünk van a kód használatára.

Az egyik lehetőségünk, hogy `<script>` tagben inline megadjuk azt a kódot, amelyet szeretnénk futtatni. Ezt általában akkor érdemes, ha nem végzünk túl sok és/vagy bonyolult műveletet az oldalon.

```
1 <script>
2     alert("Hello World!");
3 </script>
```

A másik lehetőségünk, hogy behivatkozzuk a megfelelő fájlt. Ekkor célszerű a `<head>`-ben elhelyezni a `<script>` taget és a `defer` attribútumot használni,

ugyanis ilyenkor a scriptet csak az oldal betöltése után kezdi el értelmezni nekünk a böngésző.

```
1 <script src="main.js" defer></script>
```

::: danger FONTOS! A tag **nem** rövidíthető és **nem** hagyható el a záró párja sem. :::

4.1.3 Szintaktika

Mivel a ES egy gyengén típusos nyelv, így a változóink felvételekor csak arra kell figyelni, hogy megadjuk, hogy később a tárolt értéket megtudjuk-e változtatni, vagy sem.

- `let`: bármikor szabadon változtatható
- `var`: bár szabadon változtatható, viselkedése eltér az előbbitől, ugyanis nem veszi figyelembe a különböző blokkokat.
- `const`: csak kezdeti érték adható neki

::: danger Vigyázat! A `var`-t változó létrehozására **ne** használjuk már! :::

Változónévnek bármilyen UTF-8-as karaktersor megadható, így a következők teljesen érvényes változók lesznek. ::: info Információ Valószínűleg a PDF formátumú jegyzetben ezek hibásan fognak megjelenni :::

```
1 let åbokml = "Norwegian";
2 const ászmok = [1,2,3,4,5];
```

::: danger FONTOS! Azonban oda kell figyelni, hogy `-`-et **NEM** tartalmazhat az elnevezés. :::

Mivel a nyelv nem követeli meg tűlünk a sor végén a ;-t, így mindig figyelni kell, hogy egy sorba csak akkor kerüljön több utasítás, ha minden utasítás ;-vel le van zárva

4.1.4 Típusok

4.1.4.1 Number

A legtöbb nyelvhez képest a ES nem kezeli külön a az `int` és a `double` típusokat.

```
1 let egesz = 10;
2 let tizedesTort = 3.14;
```

Két speciális értéke van a számoknak:

- `infinity` vagy `-infinity` => Matematikailag hibás képlet eredménye, vagy kellően nagy szám amit nem tud már értelmezni. Pl: `10/0 = infinity`
- `NaN` => Not a Number. Akkor keletkezik, ha egy számmá nem konvertálható értékkel próbálunk meg matematikai műveletet végezni. Pl: `"Hello"/"there"= NaN` (az osztás csak a számokra értelmezett)

4.1.4.2 BigInt [ES2020]

A Number kibővítésére készült egy új típus, amely nagyobb számokat is képes tárolni.Ezeket két féleképpen tudunk felvenni, amelyek egyenértékűek:

```
1 let bigInt1 = 10n;
2 let bigInt2 = BigInt(10);
```

::: danger FONTOS! **Bizonyos műveletek (pl osztás) csak és kizárólag azonos típussal működnek.** Vagyis nem lehet BigInt-et Numberrel osztani. :::

4.1.4.3 String

Szöveg tárolására alkalmas típus. A szöveg megadható " vagy ' között. Speciális jelölése, az úgynevezett `template literal`: ‘ (backtick) A template literal alkalmas arra, hogy könnyen és gyorsan belefüzzük a szövegbe a változóinkat.

```
1 let nev = "Papp Zsombor";
2 let udvozles = `Hello, ${nev}!`
3 //Hello, Papp Zsombor!
```

Több szöveg összefűzése a + operátor segítségével tehető meg

```
1 let elso = "Ez az elso tagmondat,";
2 let masodik = "ez pedig a masodik.";
3 let mondat = elso+masodik;
```

Ha számot szeretnénk összefűzni szöveggel akkor nagyon oda kell figyelni, hogy mit milyen sorrendben teszünk, ugyanis mindig stringé próbál meg alakítani.

```
1 let szamString = 30 + '10' //3010
2 let stringSzam = '10' + 20 //1020
```

Legegyszerűbb megoldás erre, hogy az átalakítandó string elé (feltéve, hogy tényleg csak számot tartalmaz) egy + jelet helyezünk el.

```
1 let szam = 30 + +"10"; //40
```

Ha nem csak szám van az adott stringben akkor keletkezhetnek elég furcsa dolgok is, pl:

```
1 let banan = 'b' + 'a' + +'a' + 'a'; // baNaNa
```

Ez azért fordul elő, mert a `+'a'` az nem konvertálható számmá, így `NaN`-t ad vissza.

4.1.4.3.1 String kezelés

4.1.4.4 Boolean

Logikai változó két értéket tud felvenni: `true` vagy `false`.

4.1.4.5 undefined

Legtöbbször akkor találkozunk vele, ha az adott változónak nem adunk értéket, ugyanis ilyenkor nem tudja a ES eldönteni, hogy minek kéne ott lennie.

4.1.4.6 null

A null esetében, az adott változónk valamilyen objektumra szeretne mutatni, azonban mivel az adott objektum nem létezik vagy nem tudott létrejönni, így a semmibe mutat a változónk.

4.1.5 Konzol kezelése

A konzolon keresztül tudunk a fejlesztőknek számt üzeneteket közvetíteni.

4.1.5.1 Egyszerű kiírások

```
1 console.log("Egyszeru bejegyzes");
2
3 console.info("Informacio!");
4 console.warn("Figylemeztetes!");
5 console.error("Hiba tortent!");
```

4.1.5.2 Táblázatos

Összetettebb adatokat, például tömböt a legegyszerűbben táblázatként tudunk kíírni a következőképp:

```
1 console.table(tomb1);
```

4.1.5.3 Számlálás

Amennyiben azt szeretnénk ellenőrizni, hogy egy adott kódrészlet hányszor fut le akkor a következő módszerrel tudjuk ezt megtenni:

```
1 console.count("cimke");
```

4.1.5.4 Eltelt idő vizsgálata

A szeretnénk például egy függvény lefutási idejét megvizsgálni akkor a kódrészletet az alábbi módon körbevéve, annak lefutása után látni fogjuk, hogy pontosan mennyi időt vett igénybe.

```
1 function fgv(){
2     console.time("cimke");
3
4     ...
5
6     console.timeEnd("cimke");
7 }
```

4.1.5.5 Konzol ürítése

```
1 console.clear();
```

4.1.6 Ellenőrző kérdések

1. Hogyan tilos változókat elnevezni?
2. A `const` kulcsszóval megjelölt változónak milyen különlegessége van a többivel szemben?
3. Milyen módon lehet egyszerű adatot, figyelmeztetés vagy hibát kiírni a fejlesztői konzolra?
4. Mit csinál a `console` `time()` és `timeEnd()` függvénye?
5. Mi a különség a `Number` és a `BigInt` típusok között?

4.2 Vezérlési szerkezetek

4.2.1 Elágazások

4.2.1.1 if-else

```
1 if (vizHomerseklet <= 0)
2 {
3      console.log("Fagyott");
4 }
5 else if(vizHomerseklet > 100)
6 {
7      console.log("Forr");
8 }
9 else
10 {
11      console.log("éFolykony");
12 }
```

4.2.1.2 switch

```
1 switch (jegy){
2     case 1: console.log("éElgtelen");break;
3     case 2: console.log("ééElgsges");break;
4     case 3: console.log("öKzepes");break;
5     case 4: console.log("óJ");break;
6     case 5: console.log("Jeles");break;
7     default: console.error("Nincs ilyen jegy")
8 }
```

Ne felejtsük el a különböző eseteinket `break;` utasítással lezárni, különben tovább fog csordulni a következő esetre.

4.2.1.3 Hármas operátor (Ternary)

```
1 x < 0 ? alert("Negativ") : alert("Pozitiv")
```

4.2.2 Ciklusok

4.2.2.1 while

```
1 let i = 0;
2 while (i<10)
3 {
4     console.log(i++);
5 }
```

4.2.2.2 do-while

```
1 let szam;
2 do
3 {
4     szam = +prompt("Adjon meg egy 0-tol kulonbozo
          szamot!")
5 }while(szam != 0);
```

4.2.2.3 for

```
1 for (let i = 0; i < 10; ++i)
2 {
3     /* code */
4 }
```

4.2.2.4 for..in

Az `in` kulcsszóval ellátott for ciklus mindig az adott szerkezet indexét vagy kulcsát adja vissza nekünk.

::: info Információ A foreach típusú függvénykben nyugodtan használjunk `const` típusú változót, ugyanis ezek a változók minden iteráció alatt újra és újra létrejöbbek, így nem probléma. :::

```
1 for (const index in tomb ){
2     /* code */
3 }
```

4.2.2.5 for..of

Az `of` kulcsszóval ellátott ciklus azonban mindig az adott elemet adja vissza.

::: info Információ A foreach típusú függvénykben nyugodtan használjunk `const` típusú változót, ugyanis ezek a változók minden iteráció alatt újra és újra létrejöbbek, így nem probléma. :::

```
1 for (const elem of tomb ){
2     /* code */
3 }
```

4.2.3 Kivételkezelés

```
1 try {
2     functionWithError();
3 }
4 catch (e)
5 {
6     console.error(`An error occured:
          ${e.toString()}`)
7 }
```

4.2.4 Ellenőrző kérdések

1. Mi a különbség a `for..in` és a `for..of` között?
2. Miért nem probléma a konstans változó a foreach típusú `for`ban?
3. Hova kell helyezni az alapértelmezett paramétereket?
4. Hogyan adhatunk meg függvény referenciát?
5. Elmenthetünk-e névtelen függvényt konstans változóba, ha igen, hogyan?

4.3 Operátorok

4.4 Függvények

4.4.1 Hagyományos függvények

```
1 function fgv(name) {
2     alert(`Hello, ${name}!`)
3 }
```

Függvényeket azonban nem csak a fenti módszerrel tudunk megadni, de képesek vagyunk egy változó számára is értéknek adni, melyet később változatlanul fel tudunk használni.

```
1 const ujFgv = function (){}
```

4.4.1.1 Függvény referencia

Az ES-ben a függvényeket egy mutató segítségével, vagyis referenciaként tároljuk, ezért képesek vagyunk ezt a referenciát tovább adni. Ilyenkor annyi a teendőnk, hogy az átadni kívánt függvénynek csak a nevét írjuk le és a zárójeleket elhagyjuk.

```
1 const otherPointer = fgv;
```

4.4.1.2 Alapértelmezett paraméter

Lehetőségünk van arra, hogy olyan paramétereket adjunk meg a függvényinknek, amelyek egy bizonyos értéket kapnak abban az esetben, ha a meghíváskor nem kapnak értéket vagy `undefined` az átadott érték. Ahhoz, hogy ezt a beállítsuk a paraméterlistában adjuk meg a megfelelő értéket.

::: warning Figyelem! Csak akkor tudja értelmezni az ES, ha az alapértelmezett paraméterek a lista végén szerepelnek. :::

```
1 function fgvWithDefaultParams(a, b = "Default
      parameter"){
2     .....
3 }
```

4.4.2 Generátor függvények

4.4.3 Nyíl függvények

```
1 (a,b) => { return a + b; }
2
3 (a,b) => a+b;
```

4.5 Dátumok

4.6 Hibakezelés

4.7 Tárolók

:::warning Figyelem! A tárolókban **csak és kizárólag** string értéket tárolhatunk, így bármilyen objektumot `JSON.stringify()` segítségével konvertálni kell! :::

:::warning Figyelem! A tárolókban, ha objectet tároltunk lekérdezéskor **ne felejtsük el** visszaalakítani az információt a `JSON.parse()` segítségével! :::

4.7.1 Helyi Tároló

Az böngésző a `localStorage`-en keresztül lehetőséget biztosít, hogy egy domainen belül adatokat tudjunk tárolni olyan módon, hogy azok később is hozzáférhetővé válhassanak.

A következő függvények segítségével tudunk a tárolóval dolgozni.

`.setItem()`	Két paramétert vár, egy kulcsot és egy értéket
`.getItem()`	A kulcs alapján visszaadja az értéket
`.removeItem()`	A kulcs alapján törli az értéket
`.clear()`	Kiüríti a tárolót
`length`	A tárolóban tárolt értékek száma

4.7.1.1 Helyi tár előnyei

- Az adat addig tárolva marad amíg kézzel nem töröljük.
- A tárolási limit ~10MB.
- Sosem küljük el ezt az adatot a szervernek.

4.7.1.2 Helyi tár hátrányai

- Mindent sima szövegként tárol, így nem biztonságos.
- Csak és kizárólag stringet tárol, így mint át kell alakítani
- Csak kliens oldali.

4.7.2 Session tár

A Session tár előbbi társától annyiban küldönbözik, hogy az adatot addig tartja meg, amíg az adott böngészőlap nyitva van.

`.setItem()`	Két paramétert vár, egy kulcsot és egy értéket
`.getItem()`	A kulcs alapján visszaadja az értéket
`.removeItem()`	A kulcs alapján törli az értéket
`.clear()`	Kiüríti a tárolót
`length`	A tárolóban tárolt értékek száma

4.7.3 Sütik

4.7.4 Ellenőrző kérdések

1. Hol tároljuk a tárakba írt értékek?

2. Mi a legnagyobb előnye a táraknak?
3. Miért kell konvertálni mentéskor/kiolvasáskor az összetett típusú elemeket?
4. Melyik adatszekezet(ek)hez hasonlóan tárolunk?
5. Mi a legnagyobb különbség a session és a helyi tároló között?

4.8 Dokumentáció JSDOC-al

4.8.1 Ellenőrző kérdések

4.9 Tömbök

::: danger Vigyázat! ES-ben minden adatszerkezetet referencia szerint kezelünk. Vagyis a változónk tartalma az nem a tényleges adat, hanem egy mutató, ami megmondja, hogy hol van a megfelelő adatunk! Ezért vagyunk képesek konstansként felvenni és mégis módosítani. :::

Az ES lehetőséget biztosít számunkra, hogy egy rugalmas szerkezetben tárolassunk több elemet is. Azonban ez a hagyományosan (más nyelvekben ismert) tömbtől jelentősen eltér. Ugyanis amit kapunk valójában egy generikus lista lesz, tehát tulajdonképpen egy tömbön belül bármilyen különböző típusú elemet is tárolhatunk, valamint tetszőlegesen tudjuk bővíteni, vagy az elemeit törölni.

Tömböt az alábbi két módszer bármelyikével létre tudjuk hozni:

```
1 const tomb1 = Array();
2 const tomb2 = []
```

4.9.1 Elemek hozzáadása

```
1 tomb1.push("elem"); //Vegehez
2 tomb1.unshift(10); //Elejehez
```

4.9.2 Elemek törlése

Az elemek törlése esetén a függvény visszadja nekünk a megfelelő elemet, így azt el tudjuk menteni egy változóba.

```
1 let elem1 = tomb2.pop(); //Utolsot
2 let elem2 = tomb2.shift(); //Elsot
```

4.9.3 Elemek kiválasztása tetszőleges helyről

A függvény csak egy másolatot csinál a kiválasztott elemekről és az eredetit tömbben érintetlenül hagyja.

Amire oda kell figyelni itt, hogy a második paraméterünk mindig az utolsó kiválasztanó elem **utánit** kell megadni.

```
1 const FIRST = 1;
2 const AFTERLAST = 4;
3
4 let elemek2 = tomb2.slice(FIRST, AFTERLAST);
```

4.9.4 Elem törlése tetszőleges helyről

Adott mennyiségű elem törlése.

```
1 const FIRST = 1;
2 const QTY = 3;
3
4 let elemek1 = tomb1.splice(FIRST, QTY);
```

4.9.5 Elemszám lekérdezése

```
1 let arraySize = tomb.length;
```

4.9.6 Elemek rendezése

```
1 tomb1.sort();
```

4.9.7 Elemek sorrendjének megfordítása

```
1 tomb2.reverse();
```

4.9.7.1 Saját rendező függvény írása

```
1 function compareEgyszeru(a, b){
2     if(a > b){
3         return 1;
4     }
5     else if( a < b)
6     {
7         return -1;
8     }
9     else{
10        return 0;
11    }
12 }
13
14 function compareOsszetett(a, b){
15    if(a.tulajdonsag > b.tulajdonsag){
16        return 1;
17    }
18    else if( a.tulajdonsag < b.tulajdonsag)
19    {
20        return -1;
21    }
22    else{
23        return 0;
24    }
25 }
```

4.9.8 Spread operátor

ES6 óta lehetőségünk van arra, hogy a tömbünk elemeit "kibonttassuk" a böngészővel. Ez pedig úgy, fog nekünk kinézni, hogy ahol több elemet vár az függvényünk, ott a `...` segítségével szépen egyesével fogja a tömb elemeit kipakolni az ES.

```
1 const eredeti = [1,2,3];
2
3 const operatorNelkul =
      [eredeti[0],eredeti[1],eredeti[2]];
4 const operatorral = [...eredeti];
```

A második esetben látjuk, hogy egy nagyobb tömbb esetén sokkal egyszerűbb és kényelmesebb megoldás.

4.9.9 Tárolás

Az előző szekcióban láttuk, hogy egy tömb másolása nem a legszebb, de miért így másoljuk le a tömböt?

```
1 const masolat = eredeti;
```

Nem elég ez ↑?

Az előző kóddal az lesz a problémánk, hogy amikor létrehozzuk a tömböt, akkor valójában az `eredeti` nevű változónk nem a tömböt tárolja el, hanem egy mutatót a tömbre. Ezért, ha lemásoljuk ami a változóban van, akkor tulajdonképpen ezt a mutatót másoljuk le, így mindekettő változó ugyanazt a tömböt fogja nézni és változtatni.

:::danger Vigyázat! Ez nem csak a tömbre de a többi adatszerkezetre és objektumra is igaz!!! :::

4.9.10 Destrukció tömbök esetén

Lehetőségünk van, arra, hogy index szerint külön változókra, vagy tömbre bontsuk szét az eredeti tömbünket.

Ezt a következő módon tudjuk megtenni:

```
1 const [lastName, firstName] = "Papp
     Zsombor".split(' ');
```

A fenti esetben a `lastName` értékel `"Papp"` és a `firstName` értéke `"Zsombor"` lett. Innentől kezdve a `firstName` és a `lastName` teljesen önálló változóként használhatóak.

A helyzet akkor érdekes, ha több adatunk is van, de nem szeretnénk, hogy azok elvesszenek, hanem szeretnénk őket összegyűjteni.

```
1 const [lastName,firstName, ...rest] = "Papp Zsombor
     Frontend Developer @ Mozilla".split(' ');
```

Ekkor az két elemünk esetén változatlan a dolog. Az érdekes a `...rest` lesz, ugyanis a `...` azt mondja meg, hogy szeretnénk minden maradékot a `rest` nevű változóban összegyűjteni.

Tehát a `rest` értéke ez lesz: `["Frontend","Developer","@","Mozilla"]`.

4.10 Map

::: danger Vigyázat! ES-ben minden adatszerkezetet referencia szerint kezelünk. Vagyis a változónk tartalma az nem a tényleges adat, hanem egy mutató, ami megmondja, hogy hol van a megfelelő adatunk! Ezért vagyunk képesek konstansként felvenni és mégis módosítani. :::

A map-ek egy olyan szerkezetet biztosítanak számunkra, amivel egy kulcs-érték páros alapján tudjuk az adatainkat tárolni.

A kulcsaink bármilyen típusúak lehetnek, azonban **fontos**, hogy két ugyanolyan nem szerepelhet a map-ben. Ha megpróbálunk egy ugyanolyan kulccsal elemet felvenni, akkor a már létezőt fogjuk felülírni vele!.

A kulccsal szemben viszont az értékeink már ténylegesn bármilyenek lehetnek, akár lehet az összes kulcshoz tartozó érték is ugyanaz.

```
1 const map = new Map();
```

4.10.1 Elemek hozzáadásaa maphez

```
1 const KEY = "First";
2 const VALUE = "Papp Zsombor";
3
4 map.set(KEY, VALUE);
```

4.10.2 Kulcs vizsgálata (létezik-e a map-ben)

```
1 const KEY = "Second";
2
3 let contains = map.has(KEY); //false
```

Kulcshoz tartozó érték lekérdezése:

```
1 const KEY = "First";
2
3 let first = map.get(KEY); //Papp Zsombor
```

4.10.3 Kulcs - érték páros törlése

Ebben az esetben az ES egy `bool` értéket ad vissza, hogy sikerült-e törölni az elemet a mapből.

```
1 const KEY = "First";
2
3 let success = map.delete(KEY);
```

4.10.4 A map mérete

```
1 let mapSize = map.size;
```

4.10.5 A map kiürítése

```
1 map.clear();
```

4.10.6 A map iterálása

4.10.6.1 Kulcsok

Egy tömböt ad vissza a map kulcsaival.

```
1 map.keys();
```

4.10.6.2 Értékek

Egy tömböt ad vissza a map értékeivel.

```
1 map.values();
```

4.10.6.3 Bejegyzések

Egy olyan tömböt ad vissza, amely minden eleme egy kételemű tömb, a megfelelő kulcs-érték párokkal

```
1 map.entries();
```

4.11 Set

::: danger Vigyázat! ES-ben minden adatszerkezetet referencia szerint kezelünk. Vagyis a változónk tartalma az nem a tényleges adat, hanem egy mutató, ami megmondja, hogy hol van a megfelelő adatunk! Ezért vagyunk képesek konstansként felvenni és mégis módosítani. :::

A Set (*vagy másnéven Halmaz*) egy olyan szerkezetet biztosít ahova az elemeinket "bedobálva" biztosak lehetunk benne, hogy minden érték csak és kizárólag egyetlen egyszer jelenik meg.

:::info Információ Alkalmas lehet például egyedi elemek kiszűrűsére. :::

```
1 const set = new Set();
```

4.11.1 Elemek hozzáadása a sethez

```
1 const VALUE = 42;
2
3 set.add(VALUE);
```

4.11.2 Elem vizsgálata (létezik-e a set-ben)

```
1 const VALUE = 39;
2
3 let contains = set.has(VALUE)
```

4.11.3 Érték törlése

Visszakapjuk, hogy `bool`ként, hogy sikerült-e a törlés.

```
1 const VALUE = 12;
2
3 let success = set.delete(VALUE);
```

4.11.4 A set mérete

```
1 let setSize = set.size;
```

4.11.5 A set kiürítése

```
1 set.clear()
```

4.12 Object

::: danger Vigyázat! ES-ben minden adatszerkezetet referencia szerint kezelünk. Vagyis a változónk tartalma az nem a tényleges adat, hanem egy mutató, ami megmondja, hogy hol van a megfelelő adatunk! Ezért vagyunk képesek konstansként felvenni és mégis módosítani. :::

Az object, a Maphez hasonlóan kulcs-érték páros alapján fog nekünk értékeket tárolni, azonban jóval rugalmasabban kezelhező, mint társa.

```
1 const object1 = Object();
2 const object2 = {};
```

```
1 const person = {
2     name: "Papp Zsombor",
3     birthYear: 2003,
4     hello(){
5         alert(`Hello, ${this.name}`);
6     }
7 }
```

4.12.1 Elemek felvétele

Az elemek felvételénél a legegyszerűbb módszer, hogy a tömbhöz hasonlóan kezelve adjuk meg a hozzádni kívánt elemet.

::: warning Figyelem! A kulcsot mindig stringként adjuk meg! :::

```
1 person["dogs"] = ["Bodri", "Frakk"];
```

4.12.2 Elemek elérése

Az elemeket két módon is elérhetjük. Lekérdezhetjük a kulcs alapján, mintha tömb lenne, azonban lekérdezhetjük, úgyis mintha egy osztály egy tulajdonsága lenne.

```
1 alert(person["name"]);
2 alert(2022-person.birthYear);
```

4.12.3 Elemek törlése az objektumból

Amennyiben megszabadulnánk egy kulcs-érték párostól, akkor már egy kicsit érdekesebb a helyzet, ugyanis egy speciális függvényt kell meghívni hozzá, amely a mögötte álló kiválasztást törli.

```
1 delete person.dogs;
```

4.12.4 Objektum védelme

Az objektumunkat többféleképpen is meg tudjuk védeni küldő beavatkozásoktól.

:::danger Vigyázat! Bármelyiket is alkalmazzuk, azzal végelesen lezárjuk az objektumunkat, így később nem lehet feloldani! :::

4.12.4.0.1 Seal

Amennyiben ezt alkalmazzük, az objektunkhoz nem lehet úgy tulajdonságot adni majd, valamit törölöni bármely tulajdonságát. De a meglévőket tudjuk még szerkeszteni.

```
1 const person = {
2     name: "Papp Zsombor"
3 }
4 Object.seal(person);
```

4.12.4.0.2 Freeze

Amennyiben lefagyasztunk egy objektumot, úgy többet semmilyen módon nem tudjuk szerkeszteni azt.

```
1 const person = {
2     name: "Papp Zsombor"
3 }
4 Object.freeze(person);
```

4.12.5 Az objektum végig iterálása

Az Object osztály statikus függvényei segítségével az előzőekhez haszonlóan végig tudunk menni az adott object kulcsain, értékein, vagy bejegyzésein.

```
1 for (const key of Object.keys(person)){
2     console.log(key);
3 }
4
5 for(const value of Object.values(person)){
6     console.log(value);
7 }
8
9 for(const entries of Object.entries(person)){
10     console.table(entries);
11 }
```

Viszont az object esetén nem csak `for..of`-al, hanem `for..in`-el is végig tudunk menni, így a kulcsokon megyünk végig.

```
1 for(const key in person){
2     console.log(person[key]);
3 }
```

4.12.6 Destrukció

Annak érdekében, hogy bizonyos adatokat az objektumból ki tudjuk emelni változóként, lehetőségünk van szétbontani azt.

```
1 const {name,birthYear} = person;
```

Ekkor a `name` és a `birthYear` onállő változóként jönnek létre és az objectben adott helyen lévő értékekkel töltődnek fel. Abban az esetben, ha a maradékot is szeretnénk elmenteni egy külön objectbe, akkor azt a `...` (rest) operátorral tehetjük meg, akkor a fenti művelet így módosul:

```
1 const {name, birthYear, ...rest} = person;
```

Mikor lehet ez hasznos nekünk?

A leghasznosabb, amikor egy függvénynek sok paramétert kell átadni, úgy, hogy egy objectből kell őket kiválogatni, akkor a következő módon meg tudjuk oldani energiatakarékosan:

```
1 function clog({name, birthYear}){
2     console.log(name);
3     console.log(birthYear);
4 }
5
6 clog(person);
```

4.13 Prototípusok

A `prototype`-on keresztül egy darab függvényt vagy változót tudunk hozzáragasztani az osztályhoz. **Fontos, hogy mindig legyen kezdőértéke.** A függvényen belül `this` segítségével tudunk hivatkozni az adott példányra amin meg fogjuk hívni. **Fontos, hogy csak publikusan látható tagokat érünk el!**

```
1 Map.prototype.sum = function (){
2     let sum = 0;
3     for(const item of this.values()){
4       sum += item;
5     }
6     return sum;
7 }
8
9 const map1 = new Map();
10
11 ...
12
13 console.log(map1.sum());
```

::: danger VIGYÁZAT! Amire nagyon oda kell figyelni,hogy a `prototype`-on keresztül hozzáadott változó automatikusan statikus lesz, de nem érhetjük el, csak a `prototype`-on keresztül! :::

Ha példány szinten szeretnénk megváltozhatni, akkor már a létrehozott néven tudjuk a pédányon hívni.

```
1 class Card{
2   ...
3 }
4
5 Card.prototype.image = "https://place.tld/best.png";
6
7 const card1 = new Card();
8
9 card1.image = "https://other.tld/best.png"
10
11 console.log(card1.image);
12 //https://other.tld/best.png
13 console.log(Card.prototype.image);
14 //https://place.tld/best.png
```

4.13.1 Mixinek

Mixineknek azokat az objektumokat hívjuk, amelyeket hozzá ragasztunk egy osztály prototípusához. Ennek az az előnye, hogy egyszerre több tulajdonságot tudunk hozzáragasztani. A hozzáragasztás az `Object` osztály statikus `assign` függvényén keresztül fog történni, ahol először megadjuk az osztályunk prototípusát, majd ezt követően a mixint, amit hozzá szeretnkénk tenni.

```
1 const userMixin = {
2   image: "https://server.tld/profile-picture.png",
3   resetPassword(newPass){
4     this.Password = newPass;
5   }
6 }
7
8 Object.assign(User.prototype,userMixin);
```

4.14 Osztályok

4.14.1 Osztályok létrehozása

ES-ben az osztályokat a többi nyelvhez hasonlóan a `class` kulcsszó segítségével definiálhatjuk.

```
1 class Triangle{
2
3 }
```

Tulajdonképpen egy ilyen módon létrehozott osztály, egy speciális függvényként fog működni a háttérben, ami egy `Object`et ad vissza nekünk. Így amikor a létrejött példányunak vizsgáljuk ne lepődjünk meg rajta, hogy egy `Object`ként fogjuk látni.

4.14.1.1 Adattagok és tagfüggvények

Az adattagjaink (azaz az oszályon belüli változóink), valamint a tagfüggvényink (azaz az osztályon belüli függvényeink) felvétele egyszerűsödik, ugyani nem kell a változót (`let`,`var`,`const`) vagy a függvényt (`function`) jelző kulcsszót kiírnunk.

Amikor felhasználjuk az osztályon belüli adattagokat (vagy tagfüggvényeket), akkor minden esetben ki **kell** írni a `this` kulcsszót elé.

```
1 class Triangle{
2   a_side;
3   b_side;
4   c_side;
5
6   //kerulet
7   perimeter(){
8     return this.a_side + this.b_side + this.c_side
9   }
10 }
```

4.14.1.2 Konstruktor

A példányunk inicializását ES-ben a `constructor` nevű speciális függvény segítségével tudjuk megtenni.

```
1 class Triangle{
2   a_side;
3   b_side;
4   c_side;
5
6   constructor(a,b,c){
7     this.a_side = a;
8     this.b_side = b;
9     this.c_side = c;
10  }
11
12  //kerulet
13  perimeter(){
14    return this.a_side + this.b_side + this.c_side
15  }
16 }
```

4.14.1.3 Példányosítás

Egy új példányt úgy tudunk létrehozni az osztályunkból, hogy `new` kulcsszó segítségével és az Osztály nevét használjuk fel. Ekkor a paraméterezés a konstruktornak megfelelően történik.

```
1 const tri1 = new Triangle(6,8,10);
```

4.14.1.4 Privát tagok és Getter-setter tagjaik

ES2022 óta képe a nyelv arra, hogy privát adattogat és tagfüggvényeket kezeljen. Ez úgy fog történni, hogy a megfelelő változót lellátjuk egy hashmark (#) karakterrel. Innentől kezdve konstruktoron és a tagfüggvényeken keresztuk tudjuk állítani az adattagunkat, azonban, az osztályon kívül nem fogjuk tudni felhasználni azt. Erre lesz megoldás a `get` valamint a `set` jelzésű függvény.

Vigyázat, habár függvényként definiáljuk, a felasználásakor valójában változóként bánunk vele!

```
1 class Triangle{
2   #a_side;
3   #b_side;
4   #c_side;
5
6   constructor(a,b,c){
7     this.#a_side = a;
8     this.#b_side = b;
9     this.#c_side = c;
10  }
11
12  get A(){
13    return this.#a_side;
14  }
15
16  set A(newValue){
17    this.#a_side = newValue;
18  }
19
20  //kerulet
21  perimeter(){
22    return this.#a_side + this.#b_side +
          this.#c_side
23  }
24 }
25
26 const tri1 = new Triangle(3,4,5);
27
28 tri1.A = 6;
29 console.log(tri1.A);
```

4.14.1.5 Osztály színű tagok

Osztály szintű tagokat a `static` kulcsszóval ellátva tudunk létrehozni. Ekkor az adattagunk vagy tagfüggvényünk az osztály bármely példányától **függetlenül** működik

```
1 class User{
2   id;
3   username;
4   #password;
5
6   static idGen = 0;
7
8   constructor(un,pw){
9     this.username = un;
10    this.#password = pw;
11
12    id = User.idGen;
13    User.idGen++;
14  }
15
16  set Password(newPass){
17    this.#password = newPass;
18  }
19
20  static compare(userA, userB){
21    return userA.id - userB.id;
22  }
23 }
24
25 ...
26
27 const users = [....];
28 users.sort(User.compare)
```

A fenti példában az `idGen`-t arra tudtuk felhasználni, hogy minden egyes példány esetében a következő számot fogja `id`-ként s flehaszánlóhoz rendelni. Ugyanúgy mint a MySQL esetében az `Auto increment`el megjelölt kulcs.

A `compare` függvény esetében pedig az volt a cél, hogy két felhasználót össze tudjuk haszonlítani, azonban a felhasználókat nem egy konkrét példánynak adjuk oda.

4.14.2 Öröklés

Az OOP-s nyelvekhez hasonlóan itt is le tudunk származni egy ős osztályból, az összes különbség a szintaktikában fog megjelenni.

::: warning Figyelem! Más programozási nyelvekkel szemben, itt ténylegesen semmilyen lehetőség és trükk nem áll rendelkezésre, hogy a privát tagokat a leszármatott ösztályban is lássuk! :::

Ahhoz, hogy az osztályunk leszármazzon egy másikból az `extends` kulcssóra lesz szükségünk amikor a gyereket definiáljuk, ezt követi az ős osztály megnevezése. Ami fontos elem, hogy a gyermek konstruktorában a `super()` függvényen

keresztül átadjuk az ősnek a hozzá tartozó értékeket.

```
1 class Vehicle{
2   max_speed;
3
4   constructor(max){
5     this.max_speed = speed;
6   }
7 }
8
9 class Car extends Vehicle{
10  engine_ccm;
11
12  constructor(max,ccm){
13    super(max);
14    engine_ccm = ccm;
15  }
16
17 }
```

4.14.2.1 Védett tagok

Jelenleg a nyelv nem kezeli a `protected` tagokat viszont konvencióként kialakult, hogy a `_` karakterrel jelzett változókat úgy kezelik, mintha ténylegesen védett lenne. Várhatóan 2024 után fogja a nyelv ténylegesen támogatni.

4.14.3 Osztályok bővítése

Az ES-ben minden osztály rendlekezik egy `prototype` nevű adattagja. Ezen keresztül tetszőleges példány szintű adattaggal vagy függvénnyel tudjuk a már meglévő osztályokat bővíteni. Ezért vagyunk képesek saját függvényt definiálni például az adatszerkezetekhez is.

4.14.4 Ellenőrző kérdések

1. Hogyan tudunk konstruktort megadni?
2. Hogyan veszünk fel privát tagokat?
3. Lehetnek-e védett tagjai egy osztálynak?
4. Láthatóak-e az leszármazottban a privát tagok?
5. Mi a különbség a prototípus és a mixin között?

4.15 Proxy

4.16 ES Modulok

4.17 Időzített futattás

4.17.1 Egyszeri futtatás x idő után

Lehetőségünk van arra, hogy egy bizonyos idő eltelte után szeretnénk valamilyen függvényt futtatni. Példaul megjelnítük a felhasználó számára egy figyelmeztetés amikor a bejelentkezés során hibás jelszót írt be és 5s múlva szeretnénk eltüntetni az alertet.

Ezt a `setTimeout()` segítségével tudjuk megtenni, melynek első paramétere a függvény (vagy annak referenciája) lesz, és a második paramétere az idő msben megadva. Azt ezt követően megadott paramétereket a függvénynek tudjuk átadni.

```
1 const timeout = setTimeout(hideAlert,5000);
```

4.17.2 Ismétlődő futtatás x időnként

Ha szeretnénk folyamatosan frissíteni valamit az oldalon pl. az időt, vagy az üzeneteket lekérni távoli kiszolgálóról, akkor a `setInterval()` segítségével tudjuk megtenni. Paraméterezése teljes mértékben megegyezik az előzővel.

```
1 const interval = setInterval(getMessages,1000);
```

4.17.3 Ismétlődés megszüntetése

Miután egy változóba elmentettük a fenti példában a függvényünket, így kaptunk egy azonosítot amellyel meg tudjuk szüntetni az idmétlődést amennyiben arra már nincs szükségünk. Például ha stoppert csinálunk, akkor nem szeretnénk, ha a stopper megállítása után számolna tovább.

```
1 clearInterval(interval);
```

4.18 Promise-ok

4.18.1 Bevezetés

A Promise egy különleges osztály a ES-ben. A célja, hogy tudjunk olyan adatokkal, dolgozni, amelyek nem feltétlenül állnak rendelkezésre azonnal. Ebből kifolyólag egy Promise-nak 3 állapota lehet:

1. `pending`: Az adatra még várni kell
2. `fulfilled`: Sikerült megszerezni a megfelelő adatot
3. `rejected`: Valami hiba történt

A következőképp fog ez nekünk kódban kinézni:

```
1 let promise = new Promise((resolve,reject) => {})
```

A Promise mindig egy függvényt kér paraméternek amit megadhatünk így nyílfüggvényként, vagy a következőképpen referenciaként is:

```
1 let promise = new Promise(func)
2
3 function func(resolve,reject)
4 {
5
6 }
```

A lényeg, hogy a függvényünk ezzel a két paraméterrel rendelkezzen.

Innentől a függvényünk törzsében megvalósíthatjük a tetszőleges logikát, ami a resolve és a reject paramétereket fogja felhasználni.

Például kérjünk egy random számot és ha páros, akkor sikerült teljesíteni a Promise-t, ha páratlan akkor nem.

```
let promise = new Promise(func);

function func(resolve,reject)
{
    let random = Math.random();
    if(random % 2 == 0)
    {
        resolve("Paros");
    }
    else{
        reject("Paratlan");
    }
}
```

Most, hogy megvan a Promise-unk, már csak fel kellene tudnunk használni. Az elkészült változónkon fogunk tudni két függvényt h1vni, a `then()`-t és a `catch()`-t. Az előbbi akkor fog lefutni, ha sikeresen teljesült a Promise, az utóbbi pedig amikor meghiusult. Nézzük meg a gyakorlatban:

```
promise.then((uzenet) => "Sikerult a Promise: " +
    uzenet).catch((uzenet)=>"Meghiusult a Promise:
    " + uzenet);
```

::: info Információ A fenti példában a nyílfüggvények ugyanúgy sima függvényre cserélhetőek, mint a `Promise()` esetén. :::

4.18.2 Ellenőrző kérdések

1. Milyen állapotai lehetnek egy `Promise`-nak?
2. Mikor fog a `.then()` lefutni?
3. Hogyan érhető el, hogy a `.catch()` fusson le?
4. A `resolve()` és a `reject()` milyen paramétereket kaphat?
5. Hogyan adjuk meg a két függvényt `Promise` számára?

4.19 Aszinkron programozás

4.19.1 `async` - `await`

Tulajdonképpen az aszinkron függvényeknek köszönhetően a Promise-oknál tanult mechanizmust tudjuk sokkal szebben és letisztultabban megoldani.

Ahhoz, hogy egy aszinkron függvényünk legyen, meg kell jelölnünk az `async` külcsszóval.

```
1 async function aszinkronFgv(){
2
3 }
```

Mit tudunk még kezdeni az aszinkron függvényekkel?

Ha be szeretnénk várni egy bizonyos adatot, akkor lehetőségünk van rá, hogy a függvényunk futása szüneteljen addig, amig meg nem érkezik a másik aszinkron függvényből vagy Promise-ból (lásd Fetch API). Ehhez arra van szükségünk, hogy `await` kulcsszót tegyünk a mgefelelő hívás elé.

```
1 async function aszinkron(){
2   const result = await MyPromise();
3   return result["counter"];
4 }
```

::: danger VIGYÁZAT! Az aszinkron függvény a visszaadott értéket egy Promise-ba fogja becsomagolni, ugyanis nem tudjuk, hogy mikor fog végigfutni a függvényünk. :::

4.19.2 Ellenőrző kérdések

1. Mi történik amikor a függvény hívás előtt `await` kulcsszó van?
2. Mi lesz az aszinkron függvény visszatérési értéke?
3. Érdemes-e elmenteni változóba az ismétlődő függvényhívást?
4. Hogyan törölhetjük a folyamatosan ismétlődő hívást, ha már nincs rá szükségünk?
5. Mennyi idő után fut le a `setTimeout()`, ha a második paramétere 15.000 (tizenötezer)?

Chapter 5

RESTful API

5.1 RESTful API

5.1.1 Methodok

Method	Feladata
GET	Lekéri az információt egy adott erőforrásról
POST	Létrehoz egy új erőforrást
PUT	Frissíti a teljes erőforrást
PATCH	Frissíti az adott erőforrás egyrészét
DELETE	Törli a megadott erőforrást

5.1.2 Fejlécek

A teljesség igénye nélkül a feladatokban a leggyakrabban használtak.

Neve	Értéke általában	Leírás
`Accept`	`application/json`	Az adat típusa amit mi **feltudunk dolgozni**.
`Content-Type`	`application/json`	Az adat típusa amit mi **küldünk**.
`Authorization`	`Bearer ${token}`	Hitelesítő token, ami azonosít minket a szerveren

5.1.3 HTTP kódok

5.1.3.1 Információ

Statusz kód	Név	Leírás
100	Continue	A szerver megkapta a kérések fejlécét, folytatódhat az adatküldés.
101	Switching Protocols	A kliens és a szerver megállapodnak a protokoll váltásról.
103	Early Hints	Lekérjük előre a fejléceket, mielőtt a tényleges kérést elvégeznénk.

5.1.3.2 Siker

Statusz kód	Név	Leírás
200	OK	A kérés sikeresen lefutott, minden rendben van.
201	Created	A kért erőforrás a megadott adatokat sikerült létrehozni.
202	Accepted	Elfogadta a szerver a kérést, de nem fejezte be annak a feldolgozását.
204	No content	Sikeresen feldolgozta a kérést, de nincs visszatérő tartalom.

5.1.3.3 Átirányítás

Statusz kód	Név	Leírás
301	Moved Permanently	Véglegesen áthelyezve
302	Found (“Moved temporarily”)	Ideiglenesen áthelyezve másik címre

5.1.3.4 Kliens oldali hiba

Statusz kód	Név	Leírás
400	Bad Request	Hibás a kérés amit küldtünk, valószínűleg helytelenül adtuk át az adatokat.

Statusz kód	Név	Leírás
401	Unauthorized	A kérés mellé nem küldtünk semmilyen fejlécet, amely hitelesítést tesz lehetővé.
403	Forbidden	Az adott felhasználónak nincs jogosultsága végrehajtani ezt a kérést.
404	Not Found	Az URL, amelyre a kérést küldtük nem létezik.
405	Method Not Allowed	Ezzel a method-al az adott URL nem tudja kezelni a kérést.
406	Not Acceptable	A szerver nem tud olyan formában adatot előállítani és visszaküldeni, amely megfelelne az `Accept` fejlécben küldött formátumnak.
415	Unsupported Media Type	A szerver nem tud olyan formátumu adatot feldolgozni ami a kérést `Content-Type` fejlécnek megfelel.

5.1.3.5 Szerver oldali hibák

Statusz kód	Név	Leírás
500	Internal Server Error	Valami hiba történt a szerveren, nem feltétlenül a kérés feldolgozása közben.
501	Not Implemented	A szerveren még nincs implementálva a megfelelő funkció.
502	Bad Gateway	A szerver átjáróként próbált meg működni.
503	Service Unavailable	A szolgáltatás jelenleg nem elérhető.
504	Gateway Timeout	Az átjáró kifutott a megadott időablakból.

5.1.4 Ellenőrző kérdések

5.2 JSON

5.2.1 Ellenőrző kérdések

5.3 Fetch API

5.3.1 `fetch(url,config)`

Mint a címben látszik, a fetch függvényünk két paramétert is kaphat ahhoz, hogy a megfelelő címmel a megfelelő módon tudjon kommunikálni.

Az első paraméter az `url`, amely stringként várja az a végpontot amelyre csatlakozni szeretnénk.

A fetch érdekesebb rásza a második helyen látható `config`, ami egy objectként fogja nekünk a kérés adatait tárolni.

5.3.2 Config object

Akövetkező kulcs érték párokat tudjuk megadni az objectünknek:

- `method`: Ez bármelyik Method beírható ide stringként
- `headers`: Ide fognak kerülni Map/objectként azok az információk amiket külön kér a szerver, például azonosításhoz.
- `body`: A ténylegesen elküldeni szánt adataink fognak objectként ide kerülni, érdemes fogyelni, hogy stringesítsük (`JSON.stringify()`). `GET` típusú kérés esetén TILOS!
- `mode`: Erre csak akkor van szükség, ha a szerveren be van állítva, hogy csak ugyanarról a domainről érkező kéréseket fogadja. Ennek az értéke lehet `cors`, `no-cors`, `same-origin` vagy `navigate`

5.3.3 A fetch, mint Promise

Az első `.then()` -ben mindig azt határozzuk meg, hogy milyen típusú adatokkal dolgozunk tovább. JSON/Blob/Text.

```
1 fetch(url, config).then(response => response.json())
```

A következőekben már foglalkozhatunk az adatok kezelésével.

```
1 fetch(url, config)
2 .then(response => response.json())
3 .then(data => myArray = data)
4 .then(()=>ShowData())
```

5.3.3.1 Példa adatok lekérdezésére

```
const headers = {
    Authorization: `Bearer ${token}`,
    "Content-Type": "application/json",
    "Accept": "application/json"
}

const config = {
    method: "GET"
    headers,
}

fetch(`kedvenchelyem.tld/api/lista`,config)
.then(response => response.json())
.then(ShowList)

function ShowList(dataArray){
    ....
}
```

5.3.3.2 Példa adatfeltöltése

```
const data = {....};

const config = {
    method: "POST",
    headers,
    body: JSON.stringify(data)
}

fetch(`kedvenchelyem.tld/api/lista`,config)
.then(response => response.json())
.then(console.table)
```

5.3.3.3 Példa módosításra

```
const editedData = {....};

const config = {
    method: "PUT",
    headers,
    body: JSON.stringify(editedData)
}

fetch(`kedvenchelyem.tld/api/lista/${dataId}`,config)
.then(response => response.json())
.then(console.table)
```

5.3.3.4 Példa törlésre

```
const config = {
    method: "DELETE",
    headers,
}

fetch(`kedvenchelyem.tld/api/lista/${dataId}`,config)
.then(response => response.json())
.then(console.table)
```

5.3.3.5 Példa OOP megvalósítás

```
1 class FetchService {
2   #base_url;
3   #sub_url;
4
5   #headers = { "Accept": "application/json",
        "Content-Type": "application/json" };
6
7   constructor(base = "kedvenchelyem.tld/api", sub =
        "/posts") {
8     this.#base_url = base;
9     this.#sub_url = sub;
10  }
11
12  list_all() {
13    return
          fetch(`${this.#base_url}${this.#sub_url}`,{method:"GET",headers:
          this.#headers}).then(r => r.json());
14  }
15
16  show_one(id) {
17    return
          fetch(`${this.#base_url}${this.#sub_url}/${id}`,
          { method: "GET", headers: this.#headers
          }).then(r => r.json());
18  }
19
20  create(created_item) {
21    return
          fetch(`${this.#base_url}${this.#sub_url}`,
          { method: "POST", headers: this.#headers,
          body: JSON.stringify(created_item)
          }).then(r => r.json());
22  }
23
24  edit(id, edited_item){
25    return
          fetch(`${this.#base_url}${this.#sub_url}/${id}`,
          { method: "PUT", headers: this.#headers,
          body: JSON.stringify(edited_item) }).then(r
          => r.json());
26  }
27
28  delete(id){
29    return
          fetch(`${this.#base_url}${this.#sub_url}/${id}`,
          { method: "DELETE", headers: this.#headers
          }).then(r => r.json());
30  }
31 }
32
33 const fs = new FetchService();
34
35 fs.list_all().then(console.table);
```

5.3.4 A fetch, mint aszinkron művelet

Ahhoz, hogy aszinkron módon tudjuk kezelni a fetchet, először is szükségünk lesz egy `async` függvényre.

Első körben az érkező adatot lementjük egy tetszőleges változóban.

::: danger Vigyázat! Figyelni kell rá, hogy a fetch előtt legyen await különben anélkül fut tovább a kód, hogy véget ér a fetch! :::

Második körben, hasonlóan a promise-os megoldáshoz, megadjuk az adatunk típusát. Az így kapott változóban már a ténylegesen feldolgozható adat van már.

```
1 async getData(){
2     let rawData = await
          fetch("kedvenchelyem.tld/api/lista");
3     let data = await rawData.json();
4     return data;
5 }
```

5.3.5 Ellenőrző kérdések

5.4 XHR Web API

5.4.1 Ellenőrző kérdések

Chapter 6

Node.js

6.1 Bevezetés a Node-ba

6.1.1 Mi a Node

Először is nézzük meg, hogy tulajdonképpen mi az a Node és mivel kapunk többe, mintha simán böngészőben futtatnánk kódot.

A Node.js egy szerver oldali ES értelmező motor. Különlegessége, hogy ennek segítségével el tudjuk hagyni a böngésző adta kereteket, így például teljes hozzáférésünk van a rendszer egyes komponenseihez, mint például a fájlrndszerhez.

::: info Információ Egész konkrétan a Google Chrome-ban és MS Edge-ben is megtalálható V8 nevű motor fut alatta :::

Az így készült alkalmazásokat azonabn nem fogjuk tudni közvetlenül futtatni böngészőben, végig kell egy `build` folyamaton esniük, ahhoz, hogy egy olyan csomagot kapjunk, amely már mehet ki az éles helyére.

6.1.2 Ellenőrző kérdések

1. Mire jó a Node?
2. Mik azok a csomagok?
3. Mire jó a `-D` kapcsoló a csomag telepítésénél?
4. Hogyan hozunk létre új csomagot?
5. Mi a különbség a CommonJs és az ESM között?

6.2 NPM Csomagok

Innentől kezdve a hagyományos weblapoktól eltérően fogjuk kezeleni az alkalmazásainkat, ugyanis innentől kezdve egységes szerkezetben fogjuk őket fejleszteni.

Ez az egységes szerkezet lesz számunkra egy csomag. Ami azért lesz jó nekünk, mert könnyebben karbantartható és bővíthető lesz mint a hagyományos megoldás.

6.2.1 Hogyan néz ki egy ilyen csomag

Tualjdonképpen két megkötése vagy ezeknek a csomagoknak, ami nélkül nem tudnak működni. Ezeken kívül minden mást szabadon határozhatunk meg, ha a használt keretrendszer azt nem szabja meg külön.

6.2.1.1 `package.json` és `package-lock.json`

Ez a két fájl írja le nekünk, hogy milyen adatai és függőségei vannak a csomagunknak, valamint milyen lehetőségeink vannak a futtatásra.

```
1 {
2   "name": "d-frontend",
3   "description": "Jegyzet a tananyaghoz",
4   "version": "1.0.0",
5   "scripts": {
6     "dev": "vite",
7     "test": "vitest"
8   },
9   "repository": {
10    "type": "git",
11    "url":
          "https://github.com/NJIT-frontend-2021/d-frontend.git"
12  },
13  "dependencies": {
14    "vite": "^4.0.0."
15  }
16  "keywords": [],
17  "author": "áIgncz Dominik Bence",
18  "bugs": {
19    "url":
          "https://github.com/NJIT-frontend-2021/d-frontend/issues"
20  },
21  "homepage": "https://frontend-idb.web.app"
22 }
```

6.2.1.2 `node_modules` könyvtár

Ez a könyvtár minden alkalommal létrejön, amikor a csomagunk függőségeit telepítjük, ugyanis ebbe a könyvtárba kerülnek letöltésre. Így verziókezelésnél,

vagy egyéb mozgatásnál gondoskodjunk róla, hogy ezt a könyvtárat töröljük vagy ignoráljuk.

6.2.2 Csomagok kezelése

A csomagok kezeléséhez a Node.js-el érkező NPM (azaz Node Package Manager) programoz fogjuk használni.

6.2.2.1 Létrehozás

Terminál (Bash, PowerShell, stb.) navigáljunk el abba a könyvtárba ahova létre szeretnénk hozni a csomagunkat és adjuk ki a következő parancsot.

```
1 npm init
```

Ekkor az NPM meg fogja kérdezni tőlünk a csomagunk alap adatait és létrehozza a könyvtárat a megfelelő fájlokkal.

6.2.2.2 Függőségek telepítése

Abban az esetben, ha egy konkrét csomagot szeretnénk telepíteni azt úgy tudjuk megtenni, hogy a csomag könyvtárába navigálva a következő parancsot adjuk ki:

```
1 npm install bootstrap
```

A fenti példa a Bootstrap nevű csomagot adja hozzá a `package.json` fájlunkhoz és letölti a szükséges fájlokat.

A parancs rövidíthető is a következő módon:

```
1 npm i bootstrap
```

Ha már fel vannak véve a megadott csomagjaink de csak a `mode_modules` könyvtár hiányzik, akkor elegendő a következő parancs futtatása:

```
1 npm i
```

Előfordulhat, hogy olyan csomagokat kell telepítsünk, amire csak addig van szükség, amíg fejlesztjük azt, tehát a felhasználónak nincs szüksége rá, ahhoz hogy működjön normálisan a webalkalmazás. Ilyen csomagok lehetnek például a teszteléshez szükséges csomagok.

```
1 npm i -D vitest
```

A `-D` kapcsoló jelzi az NPM számára, hogy csak fejlesztői csomagról van szó.

6.2.2.3 Scriptek

A `package.json`-be tesztőleges scripteket tudunk felvenni amiket tudunk az NPM segítségével futtatni

```
1 "scripts": {
2     "dev": "vite",
3     "test": "vitest"
4   },
```

A scriptek tetszőleges nevűek lehetnek és tetszőleges érvényes utasítást végrehajthatnak vagy programot futtathatnak a következőképp:

```
1 npm run dev
```

Így az NPM a “dev” nevű scriptet fogja futtatni.

Jelen példában szeretné a vite progaramot futtatni a gépen az NPM, azonban nagy valószínűséggel ilyen telepített programunk nem lesz. Megpróbálhatjuk a Terminálban futtatni, de csak egy hiba üzenetet fogunk kapni, hogy nem található.

Amit tudunk a csoagunkról, hogy van egy vite nevű függősége, amit telepítettünk is. A feladatunk pedig, hogy ezt futtassuk. Miután a gépen nincs ilyen futtatható program, itt az ideje, hogy megvizsgáljuk kicsit jobban a `node_modules` könyvtárat, ugyanis a megoldás ott lesz. A könyvtáron belül található egy `.bin` könyvtár és ez fogja tarlamazni a szükséges futtatható állományokat. Terminál segítségével pedig az NPM-en keresztül tuduk elrni őket a következőképp:

```
1 npx vite
```

6.2.3 Modulok

A Node alapvetően egy új módszert vezet be a fájlok kezelésére, annak érdekében, hogy a különböző külső csomagokat fel tudjuk használni. Ez azt fogja számunkra jelenteni, hogy a minden fájlunk szolgáltatni fog változókat, függvényeket, osztályunkat és igénybe vehet más fájl által szolgáltatott erőforrásokat is.

Ehhez a művelethez az alábbi két módszer áll a rendelkezésünkre.

6.2.3.1 ES6 Modules (rövidítve ESM)

Mint a neve is mutatja, ez a módszer a modern ECMAScript-el együtt érkezett meg a Node-ba. Ahhoz, hogy jelezük, hogy ESM-ről van szó a fájlt érdemes `.mjs` kiterjesztéssel elmentenünk. Ha pedig HTML-ben adjuk meg a `<script>` tag segítségével, akkor a `type="module"` attribútumot kell hozzáírjuk.

6.2.3.1.1 Importálás {esm-import}

Amennyiben olyan fájlból szeretnénk importálni bármit amely nem küön nem jelzi, hogy bármit is szolgáltatna azt a következő módon tudjuk megtenni:

```
1 import "bootstrap";
```

Ilyen módon azonban nem csak ES kódot importálhatunk, de például képeket vagy stíluslapokat is. Pl:

```
1 import "bootstrap/dist/css/bootstrap.min.css"
```

Ha nincs elnevezett változó, függvény, osztály azaz anonim módon szolgáltat a fájl, akkor a mi felelősségünk elnevezni a importálni kívánt dolgot:

```
1 import MyObject from "./my-object.mjs";
```

Amennyiben több névvel rendelkező dolgot is kapunk az adott fájltól, akkor azt így tudjuk megtenni:

```
1 import { szolgaltatottFuggveny } from
      "./my-functions.mjs"
```

6.2.3.1.2 Exportálás

Két lehetőségünk van arra, hogy exportáljunk a fájlunkból.

Anonim módon történő exportálás:

```
1 export default {
2     key: "value"
3 }
```

Konkrét elnevezett dolgok:

```
function szolgaltatottFuggveny(){
    //code
}

export { szolgaltatottFuggveny }
```

6.2.3.2 CommonJS (RÉGI)

Ez a módszert már egyáltalán nem használjuk új alkalmazások fejlesztéséhez. Ez még az ES6 előtt volt a Node saját megoldása.

6.2.3.2.1 Importálás {cjs-export}

Az ESM-től eltérően, ő csak anonim dolgokat tud importálni a következőképp:

```
const object = require("my-object.js")
```

6.2.3.2.2 Exportálás

Miután csak anonim importlálás lehetséges így az exportálásnál se lesz ez másképp számunkra:

```
module.exports = {
    key: "value"
}
```

6.3 Vite

6.3.1 Mire jó a Vite

A vite (ejtsd: `/vit/`) csomag olyan eszközöket csomagol össze számunkra, amellyel egyszerűen és gyorsan tudunk:

- Helyi fejlesztői szervert üzemeltetni
- Elkészíteni a végleges helyére kerülő appot
- Kezelni a statikus forrásainkat (Pl: képek, stílusok)
- Környezeti fájlból adatot olvasni

6.3.2 Hogyan használjuk

Telepítés után a legtöbbször a következő script-eket célszerű megadni (ha a csomag készítésekor már nem adottak):

```
1 "scripts": {
2     "dev": "vite --host",
3     "build": "vite build",
4     "preview": "vite preview"
5 }
```

Script	Mit csinál
`dev`	Fejlesztői szervert futtatja, folyamatosan frissül ha változást észlel.
`build`	Tárhelyre való feltöltésre alkalmas csomagot pakolja össze.
`preview`	Megnézhetjük vele, hogy az összepakolt csomagunk helyesen működik-e.

Ezek után a fejlesztői szervert elindítva a vite meg fogja keresni a gyökérben az `index.html` fájlunkat. Innentől kezdve a HTML fájl felelőssége, hogy amegfelelő scriptet/scripteket futtassa. Legtöbbször ez a következőképpen fog kinézni:

```
1 <script src="./main.mjs" type="module"
      defer></script>
```

Amire szükségünk lesz, az egy `public` nevű könyvtár, ugyanis a statikus elemeket (pl: képek) ide fogjuk elhelyezni

6.3.3 Laravel

Nem csak frontend oldalon, de backend oldalon is sokat tud nekünk segíteni a csomag, ugyanis képes a Blade-del is együttműködni. Ez azért lesz számunkra előnyös, mert a különböző források importálását teszi nagyon egyszerűvé számunkra.

A gyökérben találhat `vite.config.js` segítségével megadhatjuk, hogy milyen állományokat szeretnénk felhasználni a Blade-ben.

Pl.:

```
1  laravel({
2      input: ['resources/css/app.scss',
           'resources/js/app.js'],
3      refresh: true,
4 }),
```

Ebben az esetben van egy alap stílusunk és ES fájlunk amit be tudunk importálni egyszerűen, és a vite fogja felügyelni, hogy minidg helyes legyen a kapcsolat.

A vite használatához, mindig futtatnunk kell NPM-el is, erre külön figyelmeztetni fog a Laravel is.

6.3.3.1 Blade importálás

Két lehetőségünk van az importálásra Blade-ben.

Az egyik az egy adott fájl importálása:

```
1 <head>
2     @vite("resources/js/app.js")
3 </head>
```

A másik lehetőségünk, hogy importálunk minden fájlt:

```
1 <head>
2     @vite
3 </head>
```

6.3.4 Ellenőrző kérdések

1. Miket tud kezelni a vite?
2. Mi a különbség a `dev` és a `preview` között?
3. Mire való a public könyvtár?
4. Mire jó nekünk backenden a vite?
5. Hogyan lehet Blade-ben felhasználni?

6.4 Axios

6.4.1 Telepítése

NPM segítségével lehet a legegyszerűbben felhasználni:

```
1 npm i axios
```

Amennyiben hagyományos ES projekthez eszertnénk használni, abban az esetben a HTML-ben a `<script>`-ünk elé kell elhelyezni a következőt:

```
1 <script
     src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

6.4.2 Kérések felépítse

Alapvetően az axios egy XHR műveletet fog a háttérben végezni. Ennek során előre összeállított konfigurációból dologozik. A kérések során továbbá az adatok konvertálásával sem kell foglalkozni, ugyanis beépítetten végzi azt el. Sőt szemben a fetch-el a Laravel biztsonsági előírásait is tudja kezelni, mint például a CSRF token.

Az axios használatához először importálnunk kell azt.

```
1 import axios from "axios";
```

Ezt követően az `axios`-on a kérés típusának megfelelő függvényt fogjuk meghívni. Ezekre az alábbiakban látunk egy-egy példát:

6.4.2.1 GET

```
1 async getData(){
2   const response = await
       axios.get('https://kedvenchelyem.tld/api');
3   return response;
4 }
```

6.4.2.2 POST

```
1 async createData(data){
2   const response = await
       axios.post('https://kedvenchelyem.tld/api',data);
3   return response;
4 }
```

6.4.2.3 PUT

```
1 async modifyData(id,data){
2   const response = await
       axios.put(`https://kedvenchelyem.tld/api/${id}`,data);
3   return response;
4 }
```

6.4.2.4 DELETE

```
1 async deleteData(id){
2   const response = await
       axios.delete(`https://kedvenchelyem.tld/api/${id}`);
3   return response;
4 }
```

6.4.3 Válaszok feléíptése

A válaszokban egy objectet kapunk, amely a következő adatokat tárolja:

Kulcs	Leírás
`data`	Tényleges adat amit küld a server
`status`	Milyen kóddal tért vissza a kérés
`statusText`	Milyen üzenetet küldött vissza a szerver
`headers`	A szerver által küldött fejlécek
`config`	Adott példányunk beállításai
`request`	Ténylegesen lefuttatot kérés

6.4.4 Saját példány

Lehetőségünk van arra, hogy létrehozzunk egy saját példányt is, amelynek előre megadhatjuk a konfigurációt. Ezt az `axios.create()` segítségével tudjuk megtenni, és általában célszerű külön madulba szervezni annak érdekében, hogy több fájl számára is elérhetővé váljon.

```
1 //http.mjs
2 import axios from 'axios';
3
4 export default const http = axios.create({
5   baseURL: "https://kedvenchelyem.tld",
6   headers: {'Authorization': 'Bearer my_token'}
7 });
```

Bármilyen alap konfigot megadhatunk neki amit szeretnénk, ha alapértelmezetten minden kérés tudna kezelni, ilyen például a `baseURL`, aminek megadása után már csak az url megfelelő részét kell függvényhíváskor.

Pl:

```
1 import {http} from "./http.mjs"
2
3 async getData(){
4   const response = await http.get("api");
5   return response;
6 }
```

6.4.5 Interceptors

6.4.6 Ellenőrző kérdések

Chapter 7

Vue.js

7.1 Vue.js alapok

7.1.1 Mik azok a keretrendszerek

Keretrendszer olyan könyvtárakat, kódokat és eszközöket tömörít, amelynek segítségével sokkal gyorsabban és egyszerűbben tudunk egy adott alkalmazást elkészíteni, mivel egy alaprendszer már adott amit nekünk csak bővítenünk kell. Viszont nem csak szabadságot de kötöttségeket is adnak ezek a keretrendszerek, ugyanis megköthetik, hogy pontosan milyen könyvtárszerkezettel, vagy kódolási stílussal dolgozzunk.

7.1.2 Hogyan épül fel a Vue.js

Két nagyon fontos alapra épül maga a Vue keretrendszer.

```
1 import { createApp } from 'vue'
2
3 createApp({
4   data() {
5     return {
6       count: 0
7     }
8   }
9 }).mount('#app')
```

```
1 <div id="app">
2   <button @click="count++">
3     Count is: {{ count }}
4   </button>
5 </div>
```

Az egyik a **deklaratív renderelés** lesz, ami azt jelenti, hogy teljesen szét fogjuk szedni a HTML oldalunkat minél kisebb egységekre és az egyes egységekhet tartozó ES kód állapotán múlik majd, hogy hogyan fog a böngészében megjelenni az adott kódrészlet.

A másik fontos eleme pedig a **reaktivitás**, vagyis a rendszer azonnal reagálni fog, ha valamely elem állapota megváltozik és ez alapján a böngészőben is látni fogjuk a változást.

És most felmerülhet a kérdés, hogy nem-e lesz lassú, ha folymatosan minden változásnál firssítjük a DOM-ot?

Valójában a DOM-ot teljes egészében nem fogjuk már használni, ugyanis az egyes egységek amiket szétszedtünk, azok a háttérben kapni fognak egy render függvényt és ennek a függvények lesz a feladat az adott elem kezelése és frissítése. Ennek az egységnek lesz a neve a **virtuális node**. És ezekből fogja a Vue összeépíteni nekünk a **Virtuális DOM**-ot.

7.1.3 Telepítés

Az appunkat legegyszerűbben az NPM-en keresztül a következő parancs segítségével tudjuk:

```
1 npm init vue@latest
```

Ekkor kapunk néhány kérdést ami alapján generálni fog nekünk, nézzük meg miket is kapunk: (A csillagok között megjelölt válaszokat válasszük ki minden esetben)

```
1 Project name: ... <your-project-name>
2 Add TypeScript? ... **No** / Yes
3 Add JSX Support? ... **No** / Yes
4 Add Vue Router for Single Page Application
    development? ... No / **Yes**
5 Add Pinia for state management? ... No / **Yes**
6 Add Vitest for Unit testing? ... No / **Yes**
7 Add Cypress for both Unit and End-to-End testing?
    ... **No** / Yes
8 Add ESLint for code quality? ... No / **Yes**
9 Add Prettier for code formatting? ... No / **Yes**
```

Ezután már csak annyi feladatunk van, hogy telepítsük a függőségeket, megnyissuk a kódot a kedvenc kódszerkesztőnkben (VSCode és PHPStrom ajánlott), valamint futtassük a fejlesztői szervert

7.1.4 Ellenőrző kérdések

1. Milyen hátrányai lehetnek a keretrendszereknek?
2. Mit használ a Vue a DOM kezelésére?
3. Mi az a virtuális node?
4. Mire jó a reaktivitás?
5. Mi az a deklaratív renderelés?

7.2 Global API

A vue-ban a Global APi sgeítségével fogjuk tudni kezelni az alkalmazásunk példányát. Így többek között lehetőségünk van létrehozni azt, valamint csatolni a megfeelő helyre, de a későbbiekben a különböző pluginokat és globálsnak szánt elemeket is ezen keresztül fogjuk megadni.

7.2.1 Vue app létrehozása

7.2.1.1 HTML

HTML oldalról egyetlen egy dologra van szükségünk, ahhoz hogy létrehozhassuk a webalkalmazásunkat, ez pedig nem más, mint egy azonosítóval ellátott üres keret.

```
1 <div id="app">
2
3 </div>
```

7.2.1.2 ECMAScript

Erről az oldalról két fontos résszel fogunk találkozni. Az egyik az alkalmazást definiáliáló object.

```
1 const App = {
2
3 }
```

A másik pedig az ezt felhasználó Vue példány.

```
1 import { createApp } from "vue";
2
3 const vueApp = createApp(App);
4
5 vueApp.mount("#app");
```

Ami itt történik, hogy létrehozunk egy új Vue példányt, az `App` objectben meghatározott szempontok alapján, és ezt hozzá csatoljuk a `#app` azonosítóval ellátott kerethez.

7.2.2 Ellenőrző kérdések

1. Mit kezel a Global API?
2. Mire van szükségünk HTML-ben, az app működéséhez?
3. Miért kell egy objektum?

4. Mit csinál a `createApp()`?
5. Mit csinál a `mount` függvény?

7.3 Direktívák

A direktívák segítségével fogjuk kiegészíteni a HTML-t. Tulajdonképpen nem mások mint speciális attribútumok, amelyek az értéket nem a megszokott módon kezelik. Tulajdonképpen ES változókat, függvényeket és kódrészleteket fogunk értékül adni, amiket a v-node (virtuális node) renderelésekor fog cserélgetni a rendszer.

7.3.1 v-text

Arra szolgál, hogy egy szöveget elhelyezzük az elemünkön belül. Erre is két lehetőség ad a kezünkbe a Vue:

```
1 <p v-text="bekezdes"> </p>
```

Látjuk, hogy nem ez a legszebb és a legegyszerűbb módszer. Szerencsére mindenki tanult az esetből, így felhasználhatjük a Blade-ből ismert bajusz operátoros megoldást itt is:

```
1 <h1> {{ oldalCime }} </h1>
```

7.3.2 v-bind

Ez a dirketíva lesz az, ami önállóan nem fog semmi értelmes dolgot csinálni nekünk. Viszont, ha más attribútummal kombináljuk, akkor láthatjuk, hogy képes a többi dirketívához hasonló képességet adni a hagyományos attribútumoknak is, tehát ők is ki fogják értékelni az értékként megadott utasítást.

```
1 <div v-bind:id="elem.id"> ... </div>
```

De, hogy ne legyen hiányérzetünk, őt is lehet (sőt célszerű) rövidíteni, ami így fog kinézni:

```
1 <div :id="elem.id"> ... </div>
```

7.3.3 v-show

Enne a segítségével mondhatjuk meg, hogy megjelenjen-e a megjelölt elemünk az oldalon vagy sem. Ammenyiben igaz értéket kap megjelenik, ha hamis értéket kap, akkor elrejti a Vue.

```
1 <div v-show="isVisible"> ... </div>
```

::: danger Vigyázat!

Az elem ugyanúgy az oldal része marad, csak egy `display: none;`-t fog kapni

:::

7.3.4 v-if | v-else-if | v-else

Ez előzőhöz hasonlóan ő is az elemek megjelenésével fog foglalkozni. Azonban lényegi különbség, hogy nem csak CSS segítségével rejtjük el a nem kellő elemet, hanem még a DOM-ból is eltávolításra kerül.

```
1 <p v-if="viz_hom < 0"> Fagyott </p>
2 <p v-else-if="0 <= viz_hom && viz_hom <= 100">
     éFolykony </p>
3 <p v-else> Forr </p>
```

7.3.5 v-for

Lehetőségünk van arra is, hogy egy tömb segítségével több elemet hozzunk létre.

```
1 <ul>
2     <li v-for="elem in tomb" :key="elem.id">
         {{elem.name}} </li>
3 </ul>
```

::: danger Figyelem!

A `:key` számára mindig adjunk meg valamilyen azonosítót ami egyedi az elemeink között, mert ez alapján tudjuk később az elemeinket kezelni.

Például, ha törlünk a tömből, akkor törlődni fog a listából is.

:::

7.3.6 v-on

Ahhoz, hogy tudjuk eseményeket kezelni (például gombnyomást), a `v-on`-t fogjuk használni, ugyanis sokkal rugalmasabb, mint a hagyományos eseménykezelők. Többek között képes egymaga kezelni, hogy milyen egégombbal kattintottunk egy elemre, vagy, hogy csak egyszer legyen lehetőség lefuttatni azt.

```
1 <button v-on:click="klikk()" > áSajt gomb </button>
```

Ezt is tudjuk azonban rövidíteni, valamit bővíteni módosítókkal:

```
1 <button @click.right="csakJobbKlikk()"> Jobb
     klikkes gomb </button>
```

7.3.6.1 Módosítók

A teljesség igénye nélkül.

Módosító	Művelet
`.stop`	meghívja az `event.stopProprgation()`-t.
`.prevent`	meghívja az `event.preventDefault()`-ot.
`.{űbillenty}`	Csak a módosító billenytű lenyomása mellett fut le.
`.left`	Csak bal egérgombra fut le.
`.right`	Csak jobb egérgombra fut le.
`.middle`	Csak középső egérgombra fut le.
`.once`	Csak és kizárólag egyszer fut le.

7.3.6.1.1 Módosító billentyűk

- `.enter`
- `.tab`
- `.delete`
- `.esc`
- `.space`
- `.up`
- `.down`
- `.alt`
- `.ctrl`
- `.shift`
- `.meta`

7.3.7 v-model

Biztosítja a kétirányú adatáramlást. Ezt a direktívát mindig beviteli mezőn fogjuk használni és a megadott változó értékét folyamatosan frissíti, ha frissül az értéke.

```
1 <input type="text" v-model="username" />
```

7.3.8 Ellenőrző kérdések

1. Miben hasonli a vue és a blade?
2. Mi a legnagyobb különség a v-show és a v-if között?
3. Mit fontos a v-for-nál megadni? Miért?
4. Miért jó a v-bind-ot használni?
5. Mire használhatók a módosítók a v-on esetén? Mutass egy példát!

7.4 Options API

Ahhoz, hogy a komponensünket megadhassuk, az egyik lehetőségünk az Options API, amely object-ek segítségével definiálja az alkalmazásunkat vagy komponensünket.

7.4.1 Állapotok

Az alkalmazáshoz vagy a komponenshez tartozó állapotainkat a `data()` függvény segítségével tudjuk tárolni. A függvényünk mindig egy objectet fog visszaadni, annak érdekében, hogy folyamatosan tudjuk frissíteni őket.

Például:

```
1 const App = {
2     data(){
3         return {
4             counter: 0,
5             firstRun: true,
6             name: "Zsombor"
7         }
8     }
9 }
```

7.4.2 Számított állapotok

Tudunk több állaptból is egy teljesen új állapotot számítani, ezeket úgy tudjuk megtenni, hogy a `computed` object-ben felvesszük őket függvényként. Egy számított állapotot jelző függvénynek midnig van visszatérési értéke.

Példa:

```
1 const App = {
2     data(){
3         return{
4             elements: ["elem1","elem2","elem3"]
5         }
6     },
7     computed:{
8         numberOfElements(){
9             return this.elements.lenght;
10        }
11    }
12 }
```

7.4.3 A függvények felvétele

Tudunk függvényeket is definiálni, például amit felhasználunk az állapotaink manipulásához, például adatok lekérése API-ról vagy eseménykezelés. Ezeket a `methods` objectben fogjuk elhelyezni.

Példa:

```
const App = {
    methods: {
        sum(a,b){
            return a+b;
        },
        mul(a,b){
            return a*b;
        }
    }
}
```

7.4.4 Állapotváltozások figyelése

A `watch` segítségével, meg tudunk figyelni állapotokat és tudunk reagálni a változásukra. Figyelni kell rá, hogy a figyelni kívánt állapotot adjuk meg a függvény nevének. Valamint ezek a függvények 1 vagy két paraméterrel rendelkezhetnek, ha csak egy paraméterünk van, akkor mindig az új értéket kapjuk meg, kettő esetén, először az új, aztán pedig a régi értéket fogjuk elérni.

Példa:

```
const App = {
    data(){
        return{
            name: "Papp Zsombor"
        }
    },
    watch:{
        name(val,oldVal){
            console.log(`Regi ertek: ${oldVal}. Uj:
                ${val}`)
        }
    }
}
```

7.4.5 Ellenőrző kérdések

1. Mit csinál a `data()`?
2. Miért kell a `data()`-ban mindig objektumot visszaadni?

3. Mi a különség a függvények és a számított állapotok között?
4. Hogyan figyelhetünk egy állapotot?
5. Mire kell odafigyelni, a `watch`-ban létrehozott függvények esetén?

7.5 Komponensek

A következőkben azt fogjuk áttekinteni, hogy hogyan lehet komponensekre bontani és azokat egymásba építve felhasználni az alkalmazásunkban.

7.5.1 Komponens importálása és felhasználása

Elsőként importálni kell a megfelelő `.vue` fájlt, ezek után szükséges lesz az obejektumunkat bővíteni a `components` nevű objectel. Ebbe fogjuk felvenni az importált komponensünket és ezáltal fog elérhetővé válni a template számára.

::: info Információ Célszerű a komponenseket egységesen egy `components` könyvtárban elhelyezni. Ha szükséges, akkor további szintekkel bővítsük ezt úgy, hogy egységben kezeljük az összetartozó komponenseinket. :::

```
1 import SubComponent from
      "./components/SubComponent.vue"
2
3 export default{
4   components: {
5     SubComponent
6   }
7 }
```

Ennek felhasználása a template-ben a következőképpen fog történni:

```
1 <template>
2   <SubComponent />
3 </template>
```

Vagy...

```
1 <template>
2   <sub-component />
3 </template>
```

::: warning Figyelem! A két megoldás között semmi különbség nincs, szabadon felhasználható bármelyik verzió. Lehetőleg kezeljük őket egységesen egy projekten belül. :::

7.5.2 Átadott tulajdonságok

Az adott komponentünkben tudjuk definiálni, hogy milyen tulajdonságokat tudunk átvenni a szülőtől.

::: danger Vigyázat! Az átadott tulajdonságok, **csak és kizárólag** olvasható módban kapjuk meg, nem módosíthatunk rajta. Azonban ha a szülőnél modosul, azt mi is megkapjuk. :::

A `props:` obejct segítségével tudjuk megadni a komponensünk számára, úgy, hogy a kulcsunk a tuladonság neve lesz, míg az érték többféleképp is összeállhat. Nézzük a példát-

```
export default{
  props: {
    name: String,
    birthDate: [String,DateTime],
    email: {
      type: String,
      required: true
    },
    connections:{
      type: Number,
      default: 0
    }
  }
}
```

Az első esetben (`name: String`) csak a típust vagy a megfelelő osztályt adjuk meg a tulajdonságunk számára.

A második esetben (`birthDate: [String,DateTime]`) a tömb azért szükséges, hogy több féle típust is elfogadunk, így a születési dátum esetén, ha fentről DateTime-os vagy Stringet kapunk, akkor mindettő helyes lesz.

A harmadik és negyedik esetben egy objektummal adhatunk meg több információt a tulajdonságról. Például, hogy kötelező-e, vagy hogy milyen alapértelmezett értéket kapjon.

A template-ben úgy fogjuk tudni átadni ezeket mint bármely más HTML attribútumot.

```
<template>
  <sub-component
    name="Papp Zsombor"
    :birthDate="new Date('2003-09-01')"
    email="papp.zsombor@example.com"
    :connections="item.connections"
  />
</template>
```

7.5.2.1 Attribútumok automatikus átadása

Abban az esetben, hogyha egy HTML attribútumot adunk meg a komponens számára, akkor az automatikusan át fog adódni, az első szinten található legelső elemnek.

Tehát a következő esetben így fog kinézni az átadódás:

```
1 <!-- App.vue -->
2 <template>
3   <sub-component
4     class="card text-white bg-dark"
5   />
6 </template>
```

```
1 <!-- SubComponent.vue áőfutsidben-->
2 <template>
3   <div class="card text-white bg-dark">
4     ...
5   </div>
6 </template>
```

Ezt felül tudjuk írni, amihez arra van szükségünk, hogy a komponensben az Options API obejktumához hozzáírjuk a következőt: `inheritAttrs: false`. A template-ben pedig a megfelelő elemre rá tudjuk tenni a `v-bind="$attrs"` direktívát. Ilyenkor az adott elemre fogja ezeket az attrivútumokat átadni a Vue.

7.5.3 Eseménykezelés

Lehetőségünk van arra, hogy a komponensünkben az eseményeket fel tudjuk adni a szülő számára. Ezt úgy tudjuk megtenni, hogy a megfelelő eseményt helyileg kezeljük, úgy hogy es speciális függvényt hívunk meg rajta. Ez pedig az `$emit()` lesz, amelynek az első paramétere az esemény neve, amelyiket felküldjük, és ezután tetszőleges számí paramétert megadhatunk, amit az eseménykezelő függvénynek adunk oda. Ezen kívül az objektumban fel kell vennünk az `emits` nevű tömböt, amelyben felvesszük a az összes felküldött eseményt.

`CounterButton.vue`:

```
1 <template>
2   <button @click="$emit('incrementClick',1)">A
       ááószmll: {{count}}</button>
3 <template>
```

```
export default{
  props:{
    count:{
      type: Number,
      default: 0
    }
  },
  emits: ['incrementClick']
}
```

App.vue:

```
<template>
  <counter-button @increment-click="inc"
     :count="count"/>
<template>
```

```
export default{
  data(){
    return{
      count: 0
    }
  },
  methods:{
    inc(n){
      this.count += n;
    }
  }
}
```

7.5.4 Életciklus

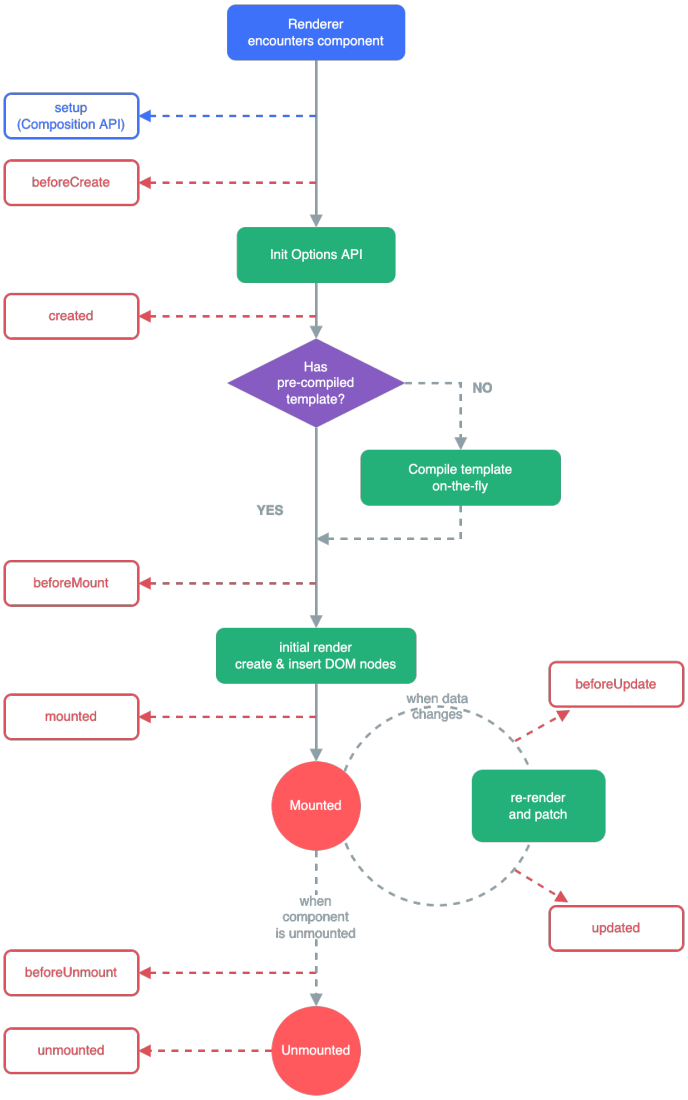

Figure 7.1: vue-lifecycle

A Vue lehetőséget ad arra, hogy a különböző komponenseink futtassanak függvényeket, akkor, ha az különböző életciklusbeli állapotot elérnek az alkalmazásban. Ezeket a fenti ábrán prossal jelzett néven tudjuk megtenni függvényként felvéve az objektumunkban. Példa egy csatolás után lefutó műveletre:

```
export default{
  methods:{
    async fetchData(){
      ...
    }
  },
  mounted(){
    this.fetchData();
  }
}
```

7.5.5 Globális komponensek

A `main.js`-ben fel tudjuk venni az alkalmazás plédányon keresztül a komponenseket úgy, hogy azt bárhol felhasználhassuk az alkalmazásunkban.

Ezt a Global API `.component` függvényével tudjuk megtenni, ahol az első paraméter a komponens felhasználandó neve lesz, míg a második a komponens objektuma importálva.

```
import MyComponent from './App.vue'

app.component('MyComponent', MyComponent)
```

7.5.6 Slotok

A keretrendszer biztosít nekünk egy olyan szerkezetet is amellyel fentől be tudunk ágyazni elemeket a komponensünkbe. Az ilyen bővítőhelyet a `<slot />` segítségével tudjuk elhelyezni.

Ekkor páros tag-ként kell mindenképp használni a komponensünket és a páros tag közé kerülő rész átadásra kerül, a Vue le fogja cserélni a `<slot />`-ot a megfelelő tartalomra. ## Ellenőrző kérdések

1. Milyen típusok adhatóak meg az átadott tulajdonságoknak?
2. Mire való az `emits: [...]`?
3. Válassz 3 életciklus függvényt, melyik mikor fut le?
4. Miért érdemes komonensekre bontani az alkalmazást?
5. Hogyan lehet globális komponenst regisztrálni?

7.6 Composition API

A Composition API egy komponens leíró API, hasonlóan az Options APi-hoz. Valójában a háttérben az előző API is ezt használja fel.

Options API	Composition API
Objektumot használ leírásra	Speciális függvényben valósul meg
Szigorúan meghatározott struktúra	Szabad kódrendezés
Nem bővíthető és nem lehet logikát kiemelni	Lehetőség van tetszőleges logikát kiemelni, újra felhasználhatóvá tenni
Kisebb projektekre és komponensekre jobb	Nagyobb projektekhez és összetettebb komponensekhez ajánlott

7.6.1 `setup()` függvény

A függvény az Options API-ban fog helyet foglalni, hasonlóan a `data()` függvényhez.

A függvényben tetszőleges ECMAScript logikát meg tudunk valósítani.

Ami fontos és nagyon figyelni kell rá, hogy csak az lesz elérhető a `<template>`-ben, amit visszadtaunk, egy objektumba ágyazva.

Tetszőleges dolgot át tudunk emelni az Options API-ból, ezeket paraméterként tudjuk odaadni a `setup()` függvénynek.

7.6.1.1 `<script setup>`

Ez előző egysezerűsítésére használhatjuk a `setup` attribútummal ellátott blokkot. Így ebben a blokkban már csak és kizárólag a `setup()` függvény törzse lesz benne.

Ami könnyebbség még, hogy nem kell visszaadnunk semmit, azt a háttérben kezeli a keretrendszer számunka.

7.6.2 Állapotok tárolása

Az állapotainkat mostmár egyszerű változóként fogjuk tárolni, így könnyebb lesz velükk dolgozni. Ennek a megoldásnak azonban az lesz a hátránya, hogy a reaktivitás képességét elveszítik ezek a változók. Ennek a megoldására két lehetőségünk van, az alábbi függvényebe csomagolva az értékeinket továbbra is reaktívak lesznek.

7.6.2.1 `ref()`

Csak és kizárólag egyszerű típusokat tudunk ezzel a megoldással használni, és figyelni kell rá, hogy a tényleges értéket a `.value`n keresztül fogjuk tudni elérni az scripten belül.

```
1 import { ref } from 'vue';
2 //...
3 const name = "Papp Zsombor";
4 //...
5 name.value = "Kocsis Zsolt";
```

7.6.2.2 `reactive()`

Az elözővel szembern, ezt a függvényt csak összetett típusokra, adatszerkezetekre tudjuk felhasználni. Viszont nagy különbség a társához képes, hogy nem kell semmilyen segítség ahhoz, hogy elérjük a benne levő értékeket. Mindemellett az objektumunk teljes mélységében képesek vagyunk módosítani.

```
1 import { reactive } from 'vue';
2 //...
3 const shoppingList = ['milk','bread','egg'];
4 //...
5 shoppingList.push('cheese');
```

7.6.3 `computed()`

A számított állapotunk úgy fog átalakulni, hogy a `computed()` függvény számára adjuk oda azt a függvényt ami előállítja. Ez segít nekünk abban is, hogy a tényleges felületet csak akkor frissítsük, ha érdemi változás történik. Ha minden állapotunk változik, azt kivéve, amiből számolunk, akkor egész egyszerűen a függvényünket nem fogja feleselgesen lefuttatni a vue.

```
1 import { computed, reactive } from 'vue';
2 //...
3 const books = reactive(["The Hitchhiker's Guide to
      the Galaxy", "Gulliver's Travels"]);
4 //...
5 const numberOfBooks = computed(()=>books.length);
```

7.6.4 `watch()`

Az Options API-ban használt változattól annyiban fogunk eltérni, hogy nem a név alapján fogunk hivatkozni a figyelt állapotra, hanem a konkrét figyelt állapotot adjuk meg neki, valamint a kezelő függvényt.

```
import {ref, watch} from 'vue';
//...
const question = ref('');
//...
watch(question,(updated){
  if(updated.includes('?')){
    getAnswer();
  }
});
```

7.6.5 `defineProps()` vagy `props:`

Az átadott tulajdonságok amik a legkevesebbet változnak, ugyanis ugyanazt az objektumot fogjuk létrehozni, csak kell csomagolnunk a `defineProps()` függvényben.

```
import { defineProps } from 'vue';
//...
const props = defineProps({id: Number, name:
    String});
```

Ha a `setup()` függvényes megoldást használjuk, akkor lehetőségünk van ezt az objektumot az Options API részeként létrehozni és csak átadjuk paraméterként.

```
import { defineProps } from 'vue';
export default{
  props: {
    id: Number,
    name: String
  },
  setup(props){
    //...
  }
}
```

7.6.6 `defineEmits()` vagy `emits:`

Az előzőhöz teljes mértékben hasonlóan működik az állapotok hirdetése.

```
1 import { defineEmits } from 'vue';
2 //...
3 const emits =
     defineEmits(['deleteItem','editItem']);
```

`setup()` függvény és Options API segítségével:

```
1 import { defineProps } from 'vue';
2 export default{
3   emits: ['deleteItem','editItem'],
4   setup(emits){
5     //...
6   }
7 }
```

7.6.7 Életciklus

7.6.7.1 `onMounted()`

Az életciklus eseményeinek kezelése úgy alakul át, hogy minden függvényünk egy `on` előtagot kap valamint, a kezdő betűnk nagyra vált. Ezen kívül a végrehajtani kívánt függvényt paraméterként tudjuk átadni neki.

```
1 import { onMounted } from 'vue';
2 //...
3 onMounted(()=>getData());
```

7.6.8 Példák

7.6.8.1 Számláló `setup()` függvénnyel

```
<script>
import {ref} from 'vue';
export default{
  setup(){
    const count = ref(0);

    const increment = () => count.value++;
    const decrement = () => count.value--;

    return {count, increment, decrement}
  }
}
</script>
<template>
  <h1>{{count}}</h1>
  <button @click="increment">+</button>
  <button @click="decrement">-</button>
</template>
```

7.6.9 Számláló `<script setup>`

```
<script setup>
import {ref} from 'vue';

const count = ref(0);

const increment = () => count.value++;
const decrement = () => count.value--;
</script>

<template>
  <h1>{{count}}</h1>
  <button @click="increment">+</button>
  <button @click="decrement">-</button>
</template>
```

7.6.10 Ellenőrző kérdések

7.7 Bejelentkezés és Hitelesítés

A következő leírásban azt fogjuk megtekinteni, hogy hogyan fog számunkra kinézni egy Vue.js alapú alkalmazásban a bejelentkezés Laravel alapú backenddel.

7.7.1 Bejelentkezési (Regisztrációs) űrlap

Elkészítjük az űrlapot ami bekéri a felhasználótól a bejelentkezési adatokat. Ez szinte mindig csak az email címet (vagy felhasználónevet) és a jelszót fogja tartalmazni számunkra.

Ezt követően az űrlap elküldését fogjuk figyelni a `@submit.prevent` segítségével, ami megakadályozza az oldal újratöltését, miközben lefuttatja a bejelentkeztető függvényünket.

A függvényben a modeleken keresztül begyűjtött adatokat a saját axios példányunkkon keresztül fogjuk elküldeni a szervernek. A visszaértkező tokent eltároljuk helyi tárolóban. Ezt követően a `useRouter()` segítségével átirányítjuk a felhasználót a következő oldalra.

```
<template>
 <div class="bg-info bg-opacity-50 m-auto mt-5 w-50
    p-3 rounded">
    <form @submit.prevent="login">
      <div class="form-group">
        <label for="email">Íéímlcm:</label>
        <input type="email" id="email"
           class="form-control"
           v-model="userData.email">
      </div>
      <div class="form-group mt-2">
        <label for="password">óJelsz:</label>
        <input type="password" class="form-control"
           v-model="userData.password">
      </div>
      <input type="submit" value="éBejelentkezs"
         class="btn btn-primary mt-3">
    </form>
  </div>
</template>
<script setup>
import {reactive,ref} from 'vue';
import {http} from '../../utils/http'
import {useRouter} from "vue-router";
const router = useRouter();
const userData = reactive({
  email: '',
  password: ''
});
const error = ref(null);
async function login(){
  const response = await http.post('login',
     userData);
  if(response.status !== 200){
    error.value = response.statusText
  }else{
    localStorage.setItem('token',response.data.data.token);
    router.push({name: 'index'});
  }
}
</script>
```

7.7.2 Axios beállítása

Szükségünk lesz egy saját Axios példányra annak érdekében, hogy egységesen tudjuk kezelni a kommunikációt. Ezt úgy fogjuk előállítani, hogy a szokásos `baseURL`-en kívül lesz eg plusz fejlécünk is. Ennek az értékét még a példányon

kívül állítjuk be, ugyanis, ha `null` értéket kap a fejléc, akkor nem kerül hozzáírásra. Ezt fogjuk kihasználni a hitelesítés meglétét vizsgáló token esetében.

Ha nem tároltuk el a token értékét, akkor a felhasználó nincs bejelnetkezve, így csak azokat az útvonalakat érheti majd el, amelyek nem kérnek hitelesítést.

Amenyiben pedig érvényes token van a helyi tárolóba, akkor hozzáfűzésre kerül a kéréshez.

```
1 import axios from 'axios';
2 const token = localStorage.getItem('token');
3 const bearer = token===null?null:`Bearer ${token}`;
4 export const http = axios.create({
5     baseURL: 'http://localhost:8881/api',
6     headers:{
7         Authorization: bearer
8     }
9 })
```

7.7.3 Router

7.7.3.1 Laravelben

Ha a két megoldást egybeépítve dolgozunk, akor figyeljünk rá, hogy a `createWebHistory()` helyett a `createWebHashHistory()`-t használjuk, annak érdekében, hogy a vue egy #-el ellássa az útvonalat azéltal kettéválasztva azt a laraveles routingtól.

```
1 export const router = createRouter({
2     history: createWebHashHistory(),
3     routes,
4 })
```

7.7.3.2 Az oldalak meta tulajdonságai

Az egyes útvonalak kialakításakor nem csak a `path`, `name` és a `component` adható meg, hanem az is, hogy milyen egyéb tulajdonságai lesznek amiket később fel lehet használni.

Például be kell-e jelentkezve lenni hozzá.

```
1 //...
2 // routes[]
3 {
4   name: 'register',
5   path: '/register',
6   component:
       ()=>import("../pages/auth/RegisterPage.vue"),
7   meta: {
8     requiresAuth: false,
9   }
10 },
11 {
12   name: 'user',
13   path: '/user',
14   component: () =>
       import('../pages/hidden/UserPage.vue'),
15   meta: {
16       requiresAuth: true,
17   }
18 },
19 //...
```

7.7.3.3 Guard

A következő feladat, hogy az előző metát, valamint a helyi tárolót felhasználva elkészítjünk egy olyan függvényt, amely eldönti, hogy továbbhaladhatunk-e a megadott útvonalon vagy sem.

Ezeket a függvényeket, amelyek az útvonalakat védik Guard-oknak hívjuk és a `router` könyvtárába fogjuk `guards` könyvetárban elhelyezni őket.

3 paraméter lesz amit tudunk kezelni:

1. `to`: hova megyünk, az útvonal minden adatával
2. `from`: honnan jöttünk, az útvonal minden adatával
3. `next`: az a függvény lesz ami elvégzi a routolást a megfelelő helyre

```
1 // /router/guards/AuthGuard.mjs
2 export function authGuard(to,from,next){
3     if (!to.meta.requiresAuth) next()
4     else {
5         if (localStorage.getItem('token') === null)
              next({name: 'login'})
6         else next()
7     }
8 }
```

Amit itt elvégzünk, hogy megnézzük, hogy az útvonal engedi e belépés néélkül a megtekintést, ha engedi, akkor a `next()`-el továbbkülsjük a cél felé.

Ha nem engedi, akkor megvizsgáljuk, hogy van-e tárolt tokenünk. Ha nincs, akkor nem vagyunk belépve, innentől a következő állomás számunkra a bejelentkező oldal lesz. Ha van tokenünk, akkor sikeresen továbbléphet a felhasználó.

7.7.3.3.1 Alkalmazása

Úgy fogjuk felhasználni ezt, hogy a router moduljába beimportálva odaadjuk a routernek, hogy ezt minden útvonal kezelése előtt futtassa le.

```
1 // /router/index.js
2 import { AuthGuard } from './guards/AuthGuard.js';
3 // ...
4 router.beforeEach(authGuard);
```

::: info Megjegyzés! Egy router több guardal is rendelkezhet, ilyenkor mindegyik le fog futni. :::

7.7.4 Ellenőrző kérdések

7.8 Vue Router

A Vue Router egy külső csomag, amely megoldja a különböző oldalak kezelését útvonalak alapján az webappunkban.

Ezt az `npm i vue-router` paranccsal tudjuk telepíteni a projektünkbe.

7.8.1 Útvonalak definiálása

Rendszerint az `/src/router/index.js`-ben fogjuk ezeket kezelni.

Elemezzük a következő mintát:

```
1 import { createRouter, createWebHistory } from
     'vue-router'
2 import HomeView from '../views/HomeView.vue'
3 import AboutView from '../views/AboutView.vue'
4
5 const router = createRouter({
6   history:
       createWebHistory(import.meta.env.BASE_URL),
7   routes: [
8     {
9       path: '/',
10      name: 'home',
11      component: HomeView
12    },
13    {
14      path: '/about',
15      name: 'about',
16      component: AboutView
17    }
18  ]
19 })
20
21 export default router
```

Ami fontos számonkra, hogy a `createRouter` és a `createWebHistory` függvényeket importáljuk a `vue-router` csomagból.

A `createRouter` felel azért, hogy elkészüljön a routoláshoz szükséges struktúra és függvények, míg a `createWebHistory` a History API-t felhasználva oldja meg, hogy leheseen az oldalak között navigálni.

::: warning Figyelem! Miután Single PAge Application-t készítünk, így ténylegesen nem lesz több oldalunk, ezért kell bővíteni a History API-t! :::

A router-ünket úgy fogjuk tudni előállítani, hogy a `createRouter` függvény egy olyan objektumot kap, amely kettő darab bejegyzéssel rendelkezik, az egyik

a `history`, amely az előbbi bővítést végrehajtja. A Másik pedig a `routes` lesz, amely egy többen tartalmazza az elérhető útvonalakat.

Minden útvonal bejegyzésünk egy-egy objektum lesz, amelyben megadjuk `path` néven az útvonalat, kap egy `name`-et amellyeltudunk rá hivatkozni később, valamint a `component` amelynek az importált nézetet átadjuk.

::: danger Vigyázat! Ne felejtsük, az egyes oldalak ugyanolyan komponensek, semmilyen különség nincs közöttük! :::

7.8.2 A router csatolása

Az alkalmazásunkban az elkészült routerünket a Global API segítségével tudjuk csatolni. Ehhez az alkalmazás példányán a `.use()` függvényt kell meghívni és az importált routert átadni neki paraméterben.

7.8.3 Routolás beállítása

Már csak két dolog hiányzik ahhoz, hogy használható legyen a rendszerünk.

Az egyik, hogy az `App.vue`-ban alakítsuk ki azt a megfelelő layoutot, amelyet szeretnénk az alkalmazásunkban és a tartalmat helyettesítsük a `<router-view />` komponenssel. Ez lesz az amibe bele fogja helyezni az adott oldal tartalmát a Vue.

A másik feladatunk, hogy az összes olyan linket, amely az alkalmazáson belülre mutat `<router-link></router-link>` segítségével oldjuk meg `<a></a>` helyett. Ennek a komponensnek a `to` attribútumon keresztül adhatjuk meg, hogy hova mutasson.

7.8.4 Paraméterek kezelése

A paraméterek megadására a lehető legegyszerűbb megoldás, hogy az útvonal megadásánál a megfelelő `/` után `:` segítségével megadjuk a paraméter nevét.

```
1 {
2   name: 'user',
3   path: "/users/:id"
4   component: UserDetails
5 }
```

Ezt a paramétert a `$route.params` segítségével érjük el a komponensünkön belül

```
1 <template>
2   <p>éParamterek: {{ $router.params }}</p>
3 </template>
```

7.8.5 Ellenőrző kérdések

1. A `routes` tömbbe, hogyan adhatjuk meg az adatokat?
2. Mire jó a `<router-view>`?
3. Hogyan haszánlható a `<router-link>`?
4. Mit jelent az `:id` az útvonalba?
5. Melyik API része a `use()` függvény?

7.9 Űrlapok kezelése

7.9.1 Ellenőrző kérdések

7.10 Állapotkezelés Pinia csomaggal

7.10.1 Ellenőrző kérdések

7.11 Vue i18n

Chapter 8

Tartalomkezelő rendszerek (CMS)

8.1 Wordpress 6

8.1.1 Ellenőrző kérdések

8.2 WordPress téma létrehozása

8.2.1 Ellenőrző kérdések

8.3 WordPress plugin létrehozása

8.3.1 Ellenőrző kérdések

8.4 Drupal 10

8.4.1 Ellenőrző kérdések

Chapter 9

Tesztelés Frontenden

9.1 Frontend tesztelés alapja

9.1.1 Ellenőrző kérdések

9.2 Vitest (Unit test)

9.2.1 Ellenőrző kérdések

Chapter 10

Hibakeresés

10.1 Hibakeresés alapok

10.1.1 Ellenőrző kérdések

Chapter 11

UX/UI tervezés (Kiegszítő)

11.1 UX/UI tervezés alapjai

11.1.1 Ellenőrző kérdések

Chapter 12

Közzététel (Kiegészítő)

12.1 Felkészülés a közzétételre

12.1.1 Ellenőrző kérdések