

Aszinkron kérések & AJAX

Webprogramozás – 10. előadás

Sulyok Csaba

csaba.sulyok@gmail.com



1. rész

XMLHttpRequest

Szinkron (klasszikus) modell

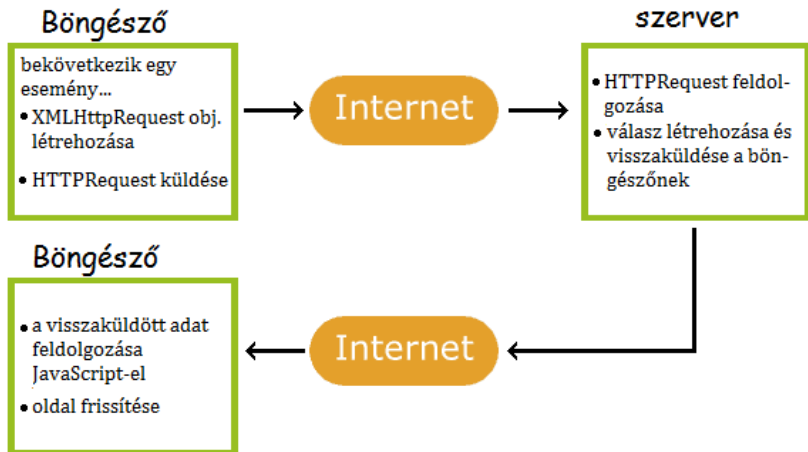
- ▶ a böngésző HTTP kérést küld a szervernek (tipikusan **GET** vagy **POST**)
- ▶ a szerver feldolgozza a kérést, és előkészíti a választ
- ▶ a szerver visszaküldi a választ (tipikusan (X)HTML)
- ▶ kliens oldalon a **teljes** oldal frissül – akkor is, ha annak egy részén egyáltalán nem történt változás

Aszinkron modell

- ▶ JavaScript kódból aszinkron HTTP kérést küldünk a szerverre
- ▶ a szerver feldolgozza a kérést, és előkészíti a választ (mint korábban)
- ▶ a visszaküldött (szöveges vagy XML formátumú) választ egy *callback* függvény értelmezi
- ▶ ennek alapján aktualizálja az oldal megfelelő részeit (de **nem tölti újra**)

- ▶ **AJAX:** Asynchronous JavaScript and XML
- ▶ az alábbi technológiákon alapul:
 - ▶ egy, a böngészőbe beépített **XMLHttpRequest** objektum
 - ▶ JavaScript + hozzáférés a DOM hierarchiához
 - ▶ adatcsere: (nem csak a névben szereplő) XML, szöveg, JSON
- ▶ 2005-ben vált népszerűvé a Google révén (Google Suggest)
- ▶ motiváció:
 - ▶ igény az interaktív webalkalmazásokra
 - ▶ a klasszikus webalkalmazás sok szempontból nem felel meg ennek az igénynek – a teljes oldal frissítése minden kérés/válasz esetén
 - ▶ a kliensek előnyben részesítenek egy böngészőben futó webalkalmazást egy specializált desktop-alkalmazással szemben
 - ▶ nem kell kliens oldali alkalmazást telepíteni
 - ▶ könnyebb karbantartás

- ▶ gyakran rövidítve **XHR**-re
- ▶ W3C Working Group Note (2016 okt.) – a legtöbb böngésző támogatja (böngészőfüggő eltérések)
- ▶ aktuális specifikáció: WHATWG Living Standard
- ▶ az XMLHttpRequest API részletes leírása
- ▶ használható JavaScript, Jscript, VBScript-ből
- ▶ segítségével aszinkron kapcsolat hozható létre a kliens és szerver között
- ▶ a kérés feldolgozását követően a szerver válasza lehet:
 - ▶ egyszerű szöveg
 - ▶ XML
 - ▶ objektum (JSON jelöléssel megadva)



Egy XMLHttpRequest objektum (főbb) metódusai



- ▶ `open(method, URL[, async, userName, password])`
 - ▶ kérés inicializálása
 - ▶ mindig használjunk relatív URL útvonalat, másképp az alkalmazásunk sebezhető lehet és a böngésző nem engedélyezi a kérést
 - ▶ az `async` alapértelmezett értéke `true`
- ▶ `send([content])`
 - ▶ kérés küldése a megadott body-val (a tartalom opcionális)
 - ▶ a fenti `async` paraméter befolyásolja ennek a metódusnak a szinkronicitását
- ▶ `getResponseHeader(key)`
- ▶ `setRequestHeader(key, value)`
- ▶ `getAllResponseHeaders()`

Egy XMLHttpRequest objektum adatai



- ▶ **readyState:**
 - ▶ **0** – a kérés még nincs inicializálva
 - ▶ **1** – a kérés inicializálva van
 - ▶ **2** – a kérés el lett küldve
 - ▶ **3** – a kérés feldolgozás alatt áll
 - ▶ **4** – megérkezett a válasz
- ▶ **onreadystatechange** – ennek értékeként kell megadni a callback függvényt, mely meg fog hívódni a **readyState** minden egyes változásakor
- ▶ **status**, **statusText** – a HTTP válasz státuszkódja, s ennek szöveges változata
- ▶ **responseType** – a válasz **Content-Type**-ja
- ▶ **response** – a HTTP válasz body tartalma – típusa változik a **responseType** szerint, pl. **ArrayBuffer**, **Blob**, **Document**, JavaScript objektum vagy **DOMString**
- ▶ **responseText**, **responseXML** – a válasz tartalma különböző formátumokban

XMLHttpRequest



The server will respond to your request several times while it's working on it. The browser uses the `readyState` property to tell you where your request is in its processing lifecycle.

If the server is sending back data as XML, `responseXML` will contain the XML tree that contains the server's response.

← We'll look more at XML responses in Chapter 9.

The server's response will be stored in `responseText`. This is usually text, but it might also be XML data.



`status` and `statusText` are used by the browser to tell your code the HTTP status that was returned by the server, such as 200 for "OK," when the server thinks everything worked as it should, or 404 for "Not Found," when the server couldn't find the requested URL.

`onreadystatechange` is the property we use to tell the browser what function to call when the server responds to a request.

▶ Példa: 5-ajax/getmessage_xhr

▶ Kliensoldali HTML:

```
<input type="button" onclick="getMessage()" value="Get message!"/>
<div id="message">No message yet...</div>
```

▶ Kliensoldali JavaScript:

```
function getMessage() {
  const xhr = new XMLHttpRequest();
  xhr.onreadystatechange = () => {
    if (xhr.readyState === 4 && xhr.status === 200) {
      document.getElementById('message').innerHTML = xhr.responseText;
    }
  };
  xhr.open('GET', '/message');
  xhr.send();
}
```

▶ Szerveroldali JavaScript:

```
app.use(express.static(path.join(process.cwd(), 'static')));
app.get('/message', (req, res) => {
  res.send('Hello from the server');
});
```

- ▶ Annak elkerülése, hogy a böngésző a cache-ből töltsse be a kért URL tartalmát:
 - ▶ válasz fejlécének beállítása szerveroldalon:
 - ▶ HTTP 1.1: `response.setHeader('Cache-Control', 'no-cache, no-store, must-revalidate');`
 - ▶ HTTP 1.0: `response.setHeader('Pragma', 'no-cache');`
 - ▶ Proxy-k: `response.setHeader('Expires', '0');`
 - ▶ változó érték (pl. véletlen szám vagy az aktuális dátum) küldése az URL-ben
- ▶ XML-alapú válasz esetén a kérésobjektum `responseXML` mezője XML-ként tartalmazza a választ
 - ▶ ennek feldolgozása `XML DOM` segítségével történik – az XML dokumentumok feldolgozásához biztosít egy standard API-t
 - ▶ a DOM az XML dokumentumot egy faszervezet formájában ábrázolja, melynek csomópontjai az elemek, attribútumok, illetve szövegrészek.
 - ▶ egyenértékű a böngésző HTML DOM-feldolgozójával, ugyanazon függvények érvényesek

- ▶ a `send` metódus paramétereként adhatunk meg kérestestben küldött adatokat
- ▶ küldés előtt a tartalom típusának fejlécinformációját be kell állítsuk (**Content-Type** header)

- ▶ példa urlencoded form információ esetén:

```
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');  
xhr.send('kulcs1=ertekek1&kulcs2=42');
```

- ▶ példa JSON esetén (**preferált**):

```
xhr.setRequestHeader('Content-Type', 'application/json');  
xhr.send(JSON.stringify({ kulcs1: 'ertekek1', kulcs2: 42 }));
```

- ▶ JSON előnyei ilyen esetben:

- ▶ egymásba ágyazott komplex objektumok küldhetőek
- ▶ számszerű és boolean típusok automatikusan deszerializáltak
- ▶ egyszerűbben használhatóak a lefoglalt szimbólumok (pl. `&`, `?`, `=`)

Példa: 5-ajax/savemessages_xhr

- ▶ egyszerű üzenetek karbantartása
- ▶ a statikus oldal aszinkron GET hívással tölti be az üzeneteket
- ▶ új üzenetet aszinkron POST hívással küldünk

Szerveroldali index.js

```
// karbantartunk egy lista üzenetet
const messages = [];

// ha kérés-body-ban JSON van, dekódoljuk
app.use(bodyParser.json());

// GET = visszaküldjük az összes eddigi üzenetet JSON-ban
app.get('/messages', (req, res) => {
  res.json(messages);
});

// POST = új üzenet beszúrása
app.post('/messages', (req, res) => {
  messages.push(req.body);
  res.send('Message inserted successfully');
});
```

Kliensoldali index.html

```
<!DOCTYPE html>
<html lang="en" title="">

<head>
  <meta charset="utf-8">
  <title>AJAX example</title>
</head>

<body onload="getMessages()">
  <input id="newMessage" type="text" placeholder="Write message here"/>
  <input type="button" onclick="sendMessage()" value="Send message!"/>
  <div>Messages:</div>
  <textarea id="messages" rows="5" cols="100" disabled></textarea>

  <script src="script.js"></script>
</body>

</html>
```

Kliensoldali `script.js` 1/2:

```
function getMessages() {  
  const xhr = new XMLHttpRequest();  
  
  xhr.onreadystatechange = () => {  
    if (xhr.readyState === 4 && xhr.status === 200) {  
      const messages = JSON.parse(xhr.responseText);  
      const formattedMessages = messages.  
        map(message => `message: ${message.text}, date: ${message.date}`).join('\n');  
      document.getElementById('messages').value = formattedMessages;  
    }  
  };  
  
  xhr.open('GET', '/messages');  
  xhr.send();  
}
```

Kliensoldali `script.js` 2/2:

```
function sendMessage() {  
    const xhr = new XMLHttpRequest();  
  
    xhr.onreadystatechange = () => {  
        if (xhr.readyState === 4 && xhr.status === 200) {  
            alert(xhr.responseText);  
            getMessages();  
        }  
    };  
  
    xhr.open('POST', '/messages');  
    xhr.setRequestHeader('Content-Type', 'application/json');  
    xhr.send(JSON.stringify({  
        text: document.getElementById('newMessage').value,  
        date: new Date(),  
    }));  
}
```


Messages:

message: My first message, date: 2019-05-14T05:35:25.950Z

message: My second message, date: 2019-05-14T05:35:31.909Z

Hívások monitorizálása a böngészőben (F12)



The screenshot displays the Chrome DevTools interface with the Network and Headers panels open. The Network panel shows two requests:

Status	Method	Domain	File	Cause	Type	Transferred	Size
200	POST	localhost:8080	me... xhr		html	234 B	29 B
200	GET	localhost:8080	me... xhr		json	401 B	188 B

The GET request is selected, and the Headers panel shows the following details:

- Request URL: `http://localhost:8080/messages`
- Request method: `GET`
- Remote address: `127.0.0.1:8080`
- Status code: `200 OK`
- Version: `HTTP/1.1`
- Referrer Policy: `no-referrer-when-downgrade`

The Response headers section shows:

- Connection: `keep-alive`
- Content-Length: `188`
- Content-Type: `application/json; charset=utf-8`
- Date: `Tue, 14 May 2019 05:36:38 GMT`
- ETag: `W/"bc-7WCz13d14n14KtDNtNid/Y5rYuI"`
- X-Powered-By: `Express`

The Request headers section shows:

- Accept: `*/*`

2. rész

A fetch API

- ▶ A **fetch** API egy egyszerűbb módszert nyújt aszinkron kérések elküldésére
- ▶ Legtöbb böngésző támogatja natívan: <https://caniuse.com/#feat=fetch>
- ▶ A klasszikus JavaScript **on...Event** mechanizmus helyett **promise**-okat használ
- ▶ Hivatalos standard: <https://fetch.spec.whatwg.org/>

► **Példa:** XMLHttpRequest GET kérés leegyszerűsítése promise-szal:

```
function get(url) {  
  return new Promise((resolve, reject) => {  
    const xhr = new XMLHttpRequest();  
  
    xhr.onreadystatechange = () => {  
      if (xhr.readyState === 4 && xhr.status === 200) {  
        resolve(xhr.responseText);  
      }  
    };  
    xhr.onerror = () => {  
      reject(Error('Network error'));  
    };  
  
    xhr.open('GET', url);  
    xhr.send();  
  });  
}  
  
function getMessage() {  
  get('/message').then((resp) => {  
    document.getElementById('message').innerHTML = resp;  
  });  
}
```

- ▶ A sikert kezelő callback egy **Response** típusú paramétert kap, melyben elérhetőek releváns információk a válaszról:

```
fetch('endpoint_url').then((response) => {  
  console.log(response.headers.get('Content-Type'));  
  console.log(response.headers.get('Date'));  
  
  console.log(response.status);  
  console.log(response.statusText);  
  console.log(response.type);  
  console.log(response.url);  
});
```

- ▶ A korábbi példa **fetchesítve** (ld. [5-ajax/getmessage_fetch](#)):

```
function getMessage() {  
  fetch('/message')  
    .then(response => response.text()) // 1. promise - válasz szövegének kinyerése  
    .then((message) => {                // 2. promise - szöveg használata  
      document.getElementById('message').innerHTML = message;  
    });  
}
```

A `fetch` metódus opciói

- ▶ a `fetch` metódusnak megadható egy második objektum típusú paraméter, amely konfigurálja a hívást

```
fetch('/endpoint_url', {  
  configKey: configValue,  
  ...  
}).then(...);
```

- ▶ alapbeállítással a `fetch` egy `GET` kérést küld a megadott URL-re, üres body-val, alapértelmezett fejlécekkel, stb.
- ▶ fontosabb beállítások (kulcsok a fenti objektumban):
 - ▶ `method` - HTTP kérés metódusa
 - ▶ `headers` - objektum a megadandó kérésfejlécekkel
 - ▶ `body` - a kérés teste szöveges formátumban
 - ▶ `redirect` - kövesse-e a hívás az átirányításokat automatikusan
 - ▶ lehetséges értékek: `follow` (alapért.), `manual`, `error`

A fetch metódus opciói

Példa: `5-ajax/savemessages_fetch` – a korábbi üzenetmentő példa `fetch` használatával

```
function getMessages() {
  fetch('/messages')
    .then(response => response.json())
    .then(messages => messages.map(message =>
      `message: ${message.text}, date: ${message.date}`)
      .join('\n'))
    .then((formattedMessages) => {
      document.getElementById('messages').value = formattedMessages;
    });
}

function sendMessage() {
  fetch('/messages', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      text: document.getElementById('newMessage').value,
      date: new Date(),
    }),
  })
    .then(response => response.text())
    .then(getMessages);
}
```


- ▶ Létező **form**ok leadása: egy gomb nyomásával kiváltunk egy **submit** eseményt, amely a böngésző alapbeállítása szerint szinkron hívást küld.
- ▶ Ha ezt ki szeretnénk védeni, mert aszinkron hívást szeretnénk küldeni:
 - ▶ **form**okon kívül helyezzük a releváns bemeneteket
 - ▶ fölülírjuk a **submit** esemény lekezelését – ebben az esetben explicit meg kell kérjünk a böngészőt hogy ne hajtsa végre az alapértelmezett lekezelést a **preventDefault** metódus segítségével
 - ▶ HTML:

```
<form onsubmit="customSubmit(event)">...
```
 - ▶ JS:

```
function customSubmit(event) {  
    event.preventDefault();  
    // az esemény feldolgozása  
}
```

- ▶ Az aszinkron kérésekhez begyűjthetjük az adatokat a DOM `getElementById` információja szerint, de így nem használjuk ki a `formok` adatgyűjtési lehetőségeit (csak a `name` attribútummal ellátott mezők csoportosítása).
- ▶ Ennek segítségére adott a `FormData` specifikáció, mellyel felépíthetünk form stílusú kulcs-érték párokat.
- ▶ `FormData` objektumokat építhetünk manuálisan, vagy a DOM `form` elemeiből automatikusan. Mind az `XMLHttpRequest.send` metódus, mind a `fetch` metódus `body` opciója fogadja, mint paraméter.
 - ▶ Manuális felépítés:

```
const formData = new FormData();
formData.append('key1', 'value1');
formData.append('key2', 'value2');
```
 - ▶ Automatikus felépítés:

```
const formData = new FormData(document.getElementById('myForm'));
```
- ▶ Ezek használata automatikusan `multipart/form-data` típussal küldi az aszinkron hívásainkat.

Példa: 5-ajax/formhandling_fetch - a 3-nodejs/formhandling aszinkron változata

```
<!-- a form action és method-je helyett
      a submit eventre figyelünk -->
<form id="myForm" onsubmit="submitForm(event)">

  <!--
    form elemek
  -->

  <input type="submit" value="Leadom!" />
</form>

<!-- ide írjuk a választ -->
<textarea id="result" rows="8" cols="70">
  Még semmi...
</textarea>

<script src="script.js"></script>
```

```
function submitForm(event) {
  // az esemény eredeti lekezelését
  // (szinkron leadás) fölülírjuk
  event.preventDefault();

  // az esemény a formra van értelmezve
  const form = event.target;
  // kivesszük a form adatait
  const formData = new FormData(form);

  // aszinkron kérést küldünk
  fetch('/submit_form', {
    method: 'POST',
    body: formData,
  }).then(response => response.text())
    .then((responseText) => {
      document.getElementById('result').value
        = responseText;
    });
}
```

3. rész

API-k és REST

▶ **Representational State Transfer**

- ▶ Eredetileg Roy Fielding 2000-ben készített PhD tézisében jelent meg. Azóta széles körben alkalmazásra került.
- ▶ Általános design elveket ír körbe kliens-szerver szolgáltatásoknak.
- ▶ CRUD műveletek átírása HTTP metódusokra jelölőnyelv (pl. JSON) body-val
- ▶ Mivel JSON-t alkalmaz, több típusú klienst kiszolgálhat (böngésző, mobil, más szerver, stb.). Böngészőkben gyakran kliensoldali UI eszközökkel együtt használjuk (pl. React, Angular) **rich client** felépítéséhez.
- ▶ metódusok:
 - ▶ **GET** - erőforrás(ok) lekérése
 - ▶ **POST** - új erőforrás beszúrására használják
 - ▶ **PUT/PATCH** - erőforrás módosítása teljes vagy részleges információk szerint
 - ▶ **DELETE** - erőforrás törlése

Példa: 5-ajax/rest

- ▶ 2 entitással dolgozó REST API, kliensoldalt nem tartalmaz
- ▶ *entitások*: blog posztok és felhasználók – a 2 entitás között egy-a-többhöz kapcsolat áll fenn
- ▶ a teljes REST API az `/api` útvonalra bekötött router segítségével jön létre

`index.js`:

```
import apiRoutes from './api.js';  
// az API-t bekötjük a /api... endpointra  
app.use('/api', apiRoutes);  
...
```

`api/index.js`

```
const router = express.Router();  
  
// minden testtel ellátott API hívás  
// JSON-t tartalmaz  
router.use(bodyParser.json());  
  
// API endpointok a megfelelő alrouterbe  
router.use('/blogPosts', blogPostRoutes);  
router.use('/users', userRoutes);
```

GET <http://myhost/api/users>

- ▶ összes felhasználó lekérése
- ▶ **Figyelem:** az URI az entitás neve többesszámban (nem [/user](#), nem [/getUsers](#), stb.)
- ▶ konvencionális tömböt térít vissza erőforrásleírásokkal
- ▶ query paraméterekben adhatunk meg szűrési információkat (adattag szerinti szűrés, pagination információ)

[api/users.js](#)

```
// findAll
router.get('/', (req, res) => {
  userDao.findAllUsers()
    .then(users => res.json(users))
    .catch(err => res.status(500).json({ message: `Could not find all users: ${err.message}` }));
});
```

Összes erőforrás elérése



GET http://localhost:8080/api/users

Params Authorization Headers Body Pre-request Script Tests

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies (1) Headers (6) Test Results Status: 200 OK Time: 29 ms Size: 287

Pretty Raw Preview JSON

```
[
  {
    "id": 1,
    "fullName": "Woody Woodpecker"
  },
  {
    "id": 2,
    "fullName": "Another user"
  }
]
```


Erőforrás elérése egyedi azonosító szerint

GET <http://myhost/api/users/2>

- ▶ egyedi azonosító esetén nem query paramétert használunk (**nem** [/users?id=2](http://myhost/api/users?id=2))
- ▶ nem talált erőforrás esetén 404-es státuszkód elvárt
- ▶ az erőforrás formátuma *nem* muszáj megegyezzen a korábbival (pl. lehet több információt küldeni, amikor csak egy entitást térítünk vissza)

[api/users.js](#)

```
// findById
```

```
router.get('/:userId', (req, res) => {  
  const { userId } = req.params;  
  userDao.findById(userId)  
    .then(user => (user ? res.json(user) : res.status(404).json({ message: `User with ID ${userId} not found` })))  
    .catch(err => res.status(500).json({ message: `Error while finding user with ID ${userId}: ${err.message}` })));  
});
```

Erőforrás elérése egyedi azonosító szerint



GET http://localhost:8080/api/users/2

Params Authorization Headers Body Pre-request Script Tests

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies (1) Headers (6) Test Results Status: 200 OK Time: 227 ms Size: 246

Pretty Raw Preview JSON

```
1 {  
2   "id": 2,  
3   "fullName": "Another user"  
4 }
```

GET <http://myhost/api/users/2/blogPosts>

- ▶ A 2-es azonosítójú felhasználóhoz tartozó összes blog poszt
- ▶ Alkalmazható egy-a-többhöz kapcsolatok esetén
- ▶ Nem intuitív, mivel a blog posztok kollekciónak kérünk le külső kulcs szerint. Így lehetne [/blogPosts?authorId=2](#) is, de ez REST szervereknél nem szokás, mert:
 - ▶ Az endpoint jogosultságok nem változnak query paraméterek alapján (pl. mert a 2-es ID-jú felhasználó posztjait láthatjuk, még nem jelenti azt, hogy mindegyiket láthatjuk)
 - ▶ További egymásbaágyazást akadályoz meg - pl.

GET <http://myhost/api/users/2/blogPosts/4>

api/users.js

```
// findBlogPostsForUser
router.get('/:userId/blogPosts', (req, res) => {
  const { userId } = req.params;
  userDao.userExists(userId)
    .then((exists) => {
      if (!exists) {
        return res.status(404).json({ message: `User with ID ${userId} not found.` });
      }
      return blogPostDao.findBlogPostsByAuthorId(userId)
        .then(blogPosts => res.json(blogPosts));
    })
    .catch(err => res.status(500).json({ message: `Error while finding blog posts for user with ID ${userId}` }));
});
```

POST `http://myhost/api/users`

- ▶ az URL megegyezik a lekérő URL-lel
- ▶ a felhasználó paraméterei a body-ban lesznek ugyanazon formátumban, mint ahogy a `GET` visszaadta
- ▶ egyedi azonosítót **nem** adunk, azt mindig a szerver generálja
- ▶ hibás bemenet esetén megfelelő státuszkód–pl. `400 Bad Request` (a példa validáláshoz middleware-t használ)

`api/users.js`

```
// insert
router.post('/', validate.hasProps(['fullName']), (req, res) => {
  userDao.insertUser(req.body)
    .then(user => res.status(201).location(`${req.fullUrl}/${user.id}`).json(user))
    .catch(err => res.status(500).json({ message: `Error while creating user: ${err.message}` }));
});
```

Erőforrás létrehozása



POST http://localhost:8080/api/users

Params Authorization Headers (1) **Body** Pre-request Script Tests

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary JSON (application/json)

```
1 {  
2   "fullName": "A new user"  
3 }
```

Body Cookies (1) Headers (7) Test Results Status: 201 Created Time: 85 ms Size: 294

Pretty Raw Preview JSON

```
1 {  
2   "id": 1,  
3   "fullName": "A new user"  
4 }
```

DELETE <http://myhost/api/users/2>

- ▶ Az egyedi azonosítót használó URI-ra küldjük a DELETE parancsot.

[api/users.js](#)


```
// delete
router.delete('/:userId', (req, res) => {
  const { userId } = req.params;
  userDao.deleteUser(userId)
    .then(rows => (rows ? res.sendStatus(204) : res.status(404).json({ message: `User with ID ${userId} not found` }))
    .catch(err => res.status(500).json({ message: `Error while deleting user with ID ${userId}: ${err.message}` }));
});
```

DELETE ▼ http://localhost:8080/api/users/2

Params Authorization Headers Body Pre-request Script Tests

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies (1) Headers (3) Test Results Status: 204 No Content Time:

Pretty Raw Preview Auto ▼ 

1

PATCH <http://myhost/api/blogPosts/2>

- ▶ A PATCH módszer kiegészíti az URI-n található erőforrást a body-ban megadott információval. Ezáltal a megadott URI-n kell létezzen már egy entitás, ellenkező esetben 404-et küldünk vissza.
- ▶ A kérés tartalmazhat részleges információt, mely esetben csak a megadott kulcsok lesznek frissítve.
- ▶ A válasz lehet 204 üres testtel, vagy 200-as az erőforrás teljes újdonsült teljes állapotával.

[api/blogPosts.js](#)

```
// update patch-csel
```

```
router.patch('/:blogPostId', (req, res) => {  
  const { blogPostId } = req.params;  
  blogPostDao.updateBlogPost(blogPostId, req.body)  
    .then(rows => (rows ? res.sendStatus(204) : res.status(404).json({ message: `BlogPost with ID ${blogPostId} not found` })))  
    .catch(err => res.status(500).json({ message: `Error while updating blog post with ID ${blogPostId}: ${err.message}` })))  
});
```

Erőforrás részleges módosítása



PATCH ▼ http://localhost:8080/api/blogPosts/1

Params Authorization Headers (1) **Body** Pre-request Script Tests

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary JSON (application/json) ▼

```
1 {  
2   "content": "My updated content"  
3 }
```

Body Cookies (1) Headers (4) Test Results Status: 204 No Content Time:

Pretty Raw Preview Auto ▼

```
1
```

Erőforrás részleges módosítása



GET

http://localhost:8080/api/blogPosts/1

Params

Authorization

Headers

Body

Pre-request Script

Tests

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body

Cookies (1)

Headers (6)

Test Results

Status: 200 OK Time: 11 r

Pretty

Raw

Preview

JSON

```
1 {  
2   "id": 1,  
3   "title": "My title",  
4   "content": "My updated content",  
5   "authorId": 1,  
6   "date": "2019-05-23T12:45:30.334Z"  
7 }
```