Konténer alapú virtualizáció Docker segítségével

Kiss Áron

Miskolc, 2023.

Tartalom

Cloud Computing 1

Konténerek 1

Docker 3

- Architektúra 4
- Telepítés 5
- Példa alkalmazás létrehozása 5
- Konténerek létrehozása 10
  - Kliens konténer létrehozása 11
  - Szerver konténer létrehozása 13
  - Adatbázis konténer létrehozása 15
- Konténerek összehangolása 17
  - Futtatás kézzel 17
  - Futtatás Docker Compose segítségével 20

Kód elérhetősége 22

Cloud Computing

A felhő alapú számítástechnika (Cloud Computing) létrejötte és rohamos fejlődése forradalmasította az informatikai szolgáltatások üzemeltetését. A Cloud Computing a 2010-es évektől kiemelkedő területe a számítástechnikának. Fontos előnye a számítási kapacitás igény szerinti elérhetősége, valamint a felhőszolgáltató által biztosított adattárolási lehetőségek és számítási teljesítmény igénybevétele a hardverek közvetlen, felhasználó általi karbantartása nélkül. Infrastrukturális szinten a felhőszolgáltatók virtuális gépeket bocsátanak rendelkezésre, és támogatják ezek későbbi, dinamikus skálázását (pl. memória és tárhely mennyiségének növelése).

A felhő alapú számítástechnika lehetővé teszi a vállalatok számára, hogy minimalizálják az IT-infrastruktúra felépítésének kezdeti költségeit, valamint alkalmazásaikat gyorsabban üzembe helyezzék, jobb kezelhetőség és kevesebb karbantartás mellett.

Lehetővé teszi továbbá, hogy a szolgáltatók gyorsan adaptálják saját erőforrásaikat az ingadozó, vagy csak időszakosan kiugró igényekhez (pl. Black Friday egy webshopnál, sorozat premier egy streaming szolgáltatónál). Ezt „cloud bursting”-nek is nevezzük.

Konténerek

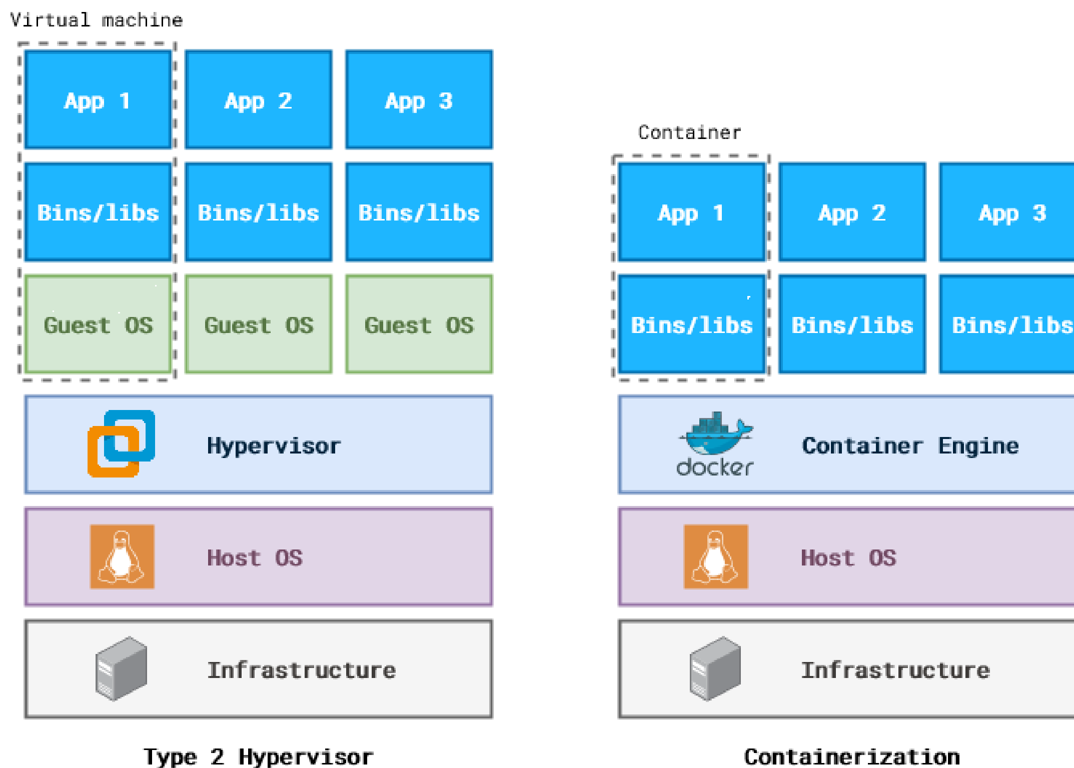
A natív felhő alapú számítástechnika (Cloud Native Computing) egy olyan modern szoftverfejlesztési megközelítés, amely a felhőt használja fel jól skálázható alkalmazások létrehozására és futtatására, dinamikus üzemeltetési környezetekben. Ennek a megközelítésnek gyakori elemei az olyan technológiák, mint a konténerek, mikroszolgáltatások, „serverless” függvények és a deklaratív kódon keresztül telepített infrastruktúra (Infrastructure as Code).

A natív felhő alapú alkalmazások gyakran Docker-tárolókban futó konténerek által nyújtott mikroszolgáltatásokból épülnek fel, amelyeket Kubernetes-ben irányítanak, és CI/CD, valamint DevOps munkafolyamatok segítségével telepítenek és üzemeltetnek.

A konténerizáció (containerization) jelentős hatást gyakorol a natív felhő alapú számítástechnikára, mert önálló telepítési egységek szabványosított formában történő létrehozásának lehetőségét biztosítja, továbbá energia-, költség-, erőforrás- és tárhely-hatékony, emellett a hagyományos virtuális gépekétől jelentősen gyorsabb rendszerindítást tesz lehetővé. Ezen tulajdonságai megkönnyítik a terheléselosztást, a rendszerkarbantartást,

valamint a konténerek földrajzi régiók közötti replikációját a jobb hibatűrés és az alkalmazások megbízhatóságának növelése érdekében.

A konténer technológia, mint OS-szintű virtualizáció jelen van, és egyre inkább teret nyer a modern szoftverek üzemeltetésében. A számítási felhő platformok következő generációja nem a hardverek, hanem az alkalmazások virtualizációján alapul.



1. ábra. Virtuális gép és konténer alapú virtualizáció.

A klasszikus virtuális gépek általában úgy működnek, hogy egy hypervisor felügyel több virtuális gépet (VM), melyek egyenként külön operációs rendszert futtatnak. A konténerek esetében minden alkalmazás ugyanazon az operációs rendszeren fut, viszont konténerenként külön-külön, elszeparált környezetekben. A konténerek között tehát processz szintű izoláció valósul meg, melyet a hoszt gép kernele biztosít.

Mivel egy-egy alkalmazás elindításakor nem kell a kernel elindulására várni, a rendszerindítási folyamat sokkal gyorsabb, mint a hagyományos VM-ek esetében. Emellett az erőforrás kihasználás jelentősen jobb (közel natív teljesítmény elérésére van lehetőség), mint a virtuális gépeknél.

A konténer technológia további előnye, hogy az alkalmazáskörnyezetet kisméretű önálló telepítési egységekbe, „képfájlokba” szervezik. Ezek a képfájlok kis méretükből adódóan

előállhatnak akár automatizáltan, egy CI/CD folyamat eredményeképpen is. Az elkészült képfájl módosítások nélkül azonnal futtatható megfelelő konténer motor segítségével.

Ennek jelentősége van a szoftverfejlesztés során is, a Docker Hub-on közel 1 millió konténer-képfájl található, melyek egy jelentős része a fejlesztéshez is használt, sokszor komplex módon üzemeltethető eszközöket (pl. adatbázisokat, gyorsítótárakat, webszervereket) tartalmaz, melyek így egyszerűen, bonyolultabb telepítési lépések nélkül indíthatók és távolíthatók el a fejlesztő gépéről.

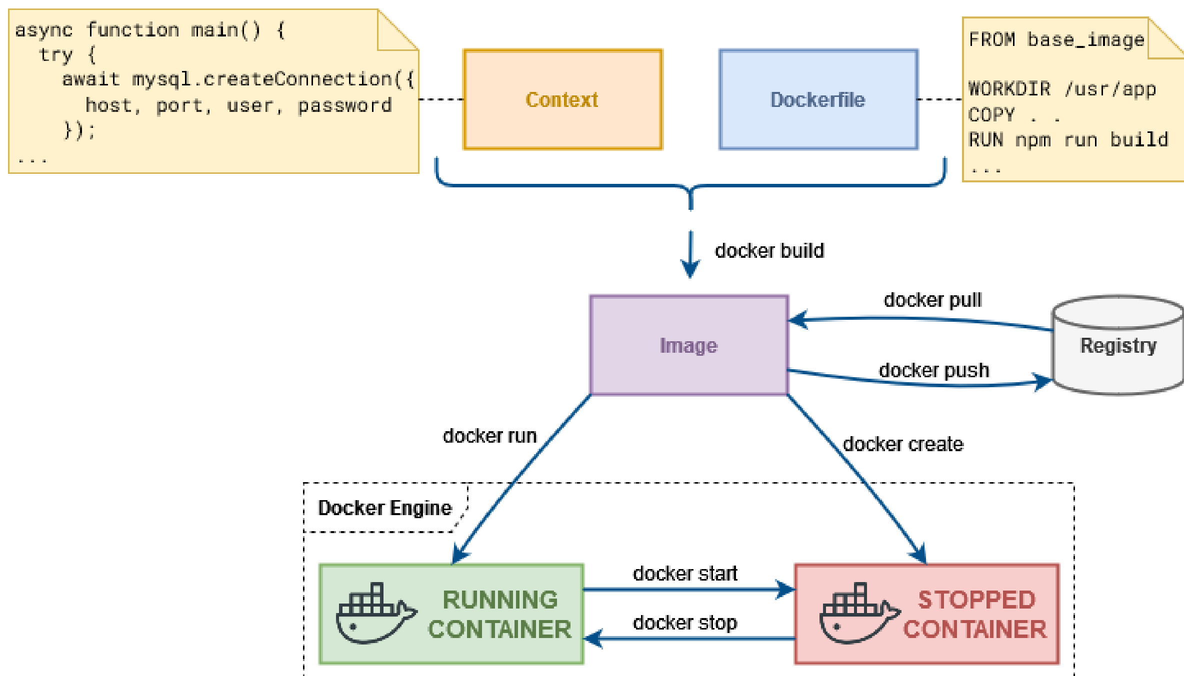
A konténerek emellett kisebb tár- és memóriaigénnyel rendelkeznek, mint a VM-ek, emiatt könnyebben lehet őket alkalmazni multi-cloud környezetben, ahol a szolgáltatás egyszerre több felhőben (pl. céges privátfelhő, Google Cloud Platform, Microsoft Azure, Amazon Web Services) van üzemeltetve. Jól alkalmazhatók továbbá cloud bursting során, amikor a megnövekvő igények miatt a saját infrastruktúra mellett igénybe kell venni egy publikus felhőszolgáltatást is az üzemeltetett alkalmazás számára.

A konténerek számos előnnyel rendelkeznek, de architektúrájukból adódóan sebezhetőbb megoldásnak számítanak, mint a klasszikus virtuális gépek, ugyanis minden konténer egyetlen kernelen fut, és csupán ennek a kernelnek a hibáiból adódóan (Single Point of Failure) előfordulhat nem kívánt adatszivárgás a konténerek között.

Docker

Az elméleti bevezető után tekintsük át egy példa alkalmazás konténer alapú üzemeltetését! Ehhez a napjainkban de-facto standardnek számító Docker platformot fogjuk felhasználni. Docker segítségével az alkalmazásokat egyszerűen csomagolhatjuk konténerekbe, majd telepíthetjük őket szinte bármilyen számítógépre vagy szerverre.

Architektúra



2. ábra. A Docker alapvető architektúrális felépítése.

A Docker architektúrális felépítését, valamint legfontosabb parancsait a 2. ábra mutatja be. Leendő példa alkalmazásunk beüzemelése során érdemes többször visszatekintenünk erre az ábrára, hogy jobban megértsük, hol tartunk éppen a folyamatban.

A fenti architektúra építőelemei a következők:

- **Image (képfájl)**: Állapotmentes, futtatható egység, amely tartalmazza az alkalmazást és annak összes szükséges függőségét, konfigurációját, akár forráskódját. A képfájl tulajdonképpen az alkalmazás-környezet statikus kezdőállapota. Docker képfájlok kontextus és Dockerfile alapján építhetőek.
  - **Kontextus**: Alkalmazásunk forráskódja és függőségei. Minden olyan fájl, információ, és egyéb erőforrás, melyre szükség van a szoftverünk működéséhez.
  - **Dockerfile**: A Docker-képfájl létrehozásához szükséges parancsokat tartalmazó fájl.
- **Container (konténer)**: A Docker-képfájl egy éppen futó példánya. Minden konténer rendelkezik képfájllal. Egy képfájl több példányban (azaz több konténerként) is elindítható.
- **Registry (tároló)**: Előre elkészített Docker-képfájlokat tároló szerver. Lehet publikus (pl. Docker Hub) vagy privát (pl. céges) elérésű. A registry-be feltölthetők (push), illetve onnan letölthetők (pull) a képfájlok.

Telepítés

Első lépésként telepítenünk kell a Docker-t, melynek legegyszerűbb módja a Docker Desktop elnevezésű eszköz telepítése, amely minden szükséges eszközt tartalmaz képfájlok és konténerek építéséhez, futtatásához, menedzseléséhez.

A telepítőt innen tudjuk letölteni, a telepítési folyamat egyszerű grafikus felületen keresztül történik: https://www.docker.com/products/docker-desktop/

A telepítést követően indítsuk el a Docker Desktop-ot, majd ha a megjelenő felületen azt látjuk, hogy minden sikeresen elindult, nyissunk egy terminál ablakot!

Innen futtassuk a `docker version` parancsot, így ellenőrizve a telepítés sikerességét!

Következő lépésben egy kész Docker képfájlt fogunk letölteni, majd konténerként futtatni.

A `docker pull hello-world` parancs segítségével töltsük le a „hello-world" elnevezésű képfájlt a Docker Hub-ról! Ez egy egyszerű alkalmazás, mely kiír egy üdvözlő szöveget, majd leáll.

A `docker run hello-world` parancs segítségével a képfájlból konténert készítünk és azt el is indítjuk.

A terminál ablakban megjelent a konténer kimenete, melyben elolvashatók az eddig elvégzett lépéseink.

Most futtassuk a `docker images` parancsot!

```
PS C:\ > docker images
REPOSITORY   TAG                  IMAGE ID          CREATED           SIZE
hello-world  latest               9c7a54a9a43c      1 minutes ago     13.3kB
```

A fenti kimenet azt jelzi, hogy a `hello-world` képfájl tárolásra került saját számítógépünkön, így amennyiben a jövőben újból szeretnénk azt használni, nem szükséges letöltenünk egy távoli szerverről, hiszen rendelkezünk helyi másolattal.

Természetesen a képfájl későbbi változásai (pl. biztonsági frissítések) szükségessé tehetik azt, hogy az image egy új verzióját is letöltsük a registry-ből.

Példa alkalmazás létrehozása

Amennyiben az előbbi parancsokat sikeresen futtattuk, az azt jelenti, hogy Docker környezetünk megfelelően működik.

Most állítsunk össze egy egyszerű webalkalmazást, mely a napi teendőink nyilvántartását hivatott segíteni. Alkalmazásunkat három részre, és ennek megfelelően 3 konténerre fogjuk osztani:

- kliensre,
- szerverre,
- és adatbázisra.

Projektünk számára hozzunk létre egy könyvtárat `docker-tutorial` elnevezéssel! Ezen belül hozzuk létre a `frontend` és `backend` mappákat! A cél a következő fájlstruktúra kialakítása lesz:

```
docker-tutorial/
├── backend/
│   ├── package.json
│   ├── src/
│   │   └── index.ts
│   └── tsconfig.json
└── frontend/
    └── index.html
```

A szerveroldalunk egy Node.js projekt lesz, TypeScript nyelven írva. Ehhez először hozzuk létre `backend` mappánkban a `package.json` fájlt, a következő tartalommal:

```
1.  {
2.    "name": "docker-tutorial-backend",
3.    "version": "1.0.0",
4.    "description": "Basic backend application to introduce Docker.",
5.    "scripts": {
6.      "start": "ts-node ./src/index.ts",
7.      "build": "tsc --build"
8.    },
9.    "author": "Kiss, Áron",
10.   "license": "MIT",
11.   "dependencies": {
12.     "cors": "^2.8.5",
13.     "express": "^4.18.2",
14.     "mysql2": "^3.2.4"
15.   },
16.   "devDependencies": {
17.     "@types/cors": "^2.8.13",
18.     "@types/express": "^4.17.17",
19.     "@types/node": "^18.16.3",
20.     "ts-node": "^10.9.1",
21.     "typescript": "^5.0.4"
22.   }
23. }
```

A 11-15. sorokban láthatók azok a csomagok, melyekre az alkalmazás működéséhez feltétlenül szükség van. A 16-22. sorokban olyan csomagok vannak felsorolva, melyek csak a fejlesztés során segítik a programozó munkáját. Ezek a fejlesztői függőségek az elkészült konténerünkbe nem is fognak belekerülni, hiszen feleslegesen növelnék annak méretét.

A TypeScript nyelven írt projektünk közvetlenül nem futtatható Node.js segítségével. Futtatása előtt a projektet JavaScript nyelvre szükséges lefordítani. Ennek a fordítási folyamatnak a beállításait tartalmazza a `tsconfig.json` fájl, melyet szintén a `backend` mappában kell létrehozni a következő tartalommal:

```
1.  {
2.    "compilerOptions": {
3.      "target": "es2016",
4.      "module": "commonjs",
5.      "esModuleInterop": true,
6.      "forceConsistentCasingInFileNames": true,
7.      "strict": true,
8.      "skipLibCheck": true,
9.      "outDir": "build"
10.   }
11. }
```

A fenti kód 9. sora lényeges számunkra: a közvetlenül futtatható JavaScript-fájlok a `backend/build/` mappába fognak bekerülni a fordítást követően.

Most hozzuk létre szerveroldalunk érdemi implementációját, melyet a `backend/src/` mappán belüli `index.ts` fájlba kell bemásolni:

```
1.  import mysql from 'mysql2/promise';
2.  import express from 'express';
3.  import cors from 'cors';
4.
5.  const host = process.env.DB_HOST || 'localhost';
6.  const port = Number(process.env.DB_PORT || 3306);
7.  const user = process.env.DB_USER || 'root';
8.  const password = process.env.DB_PASSWORD || '';
9.  const database = process.env.DB_DATABASE;
10.
11. async function main() {
12.     try {
13.         const connection = await mysql.createConnection({
14.             host, port, user, password, database
15.         });
16.
17.         connection.query("CREATE TABLE IF NOT EXISTS todo (id INT NOT NULL
    AUTO_INCREMENT, description varchar(250) NOT NULL default '', done BOOLEAN NOT NULL
    default false, PRIMARY KEY (id))");
18.
19.         const app = express();
20.         app.use(cors());
21.
22.         app.get("/api/todo", async (req, res) => {
23.             try {
24.                 const result = await connection.query('SELECT * FROM todo');
25.                 const todos = result[0];
26.                 res.json(todos);
27.             } catch (err) {
28.                 res.status(500).send();
29.             }
30.         });
31.
32.         app.put('/api/todo/:id/done', async (req, res) => {
33.             try {
```

```
34.                 await connection.execute('UPDATE todo SET done=1 WHERE id=?',
    [req.params.id]);
35.                 res.status(200).send();
36.             } catch (err) {
37.                 res.status(500).send();
38.             }
39.         });
40.
41.         app.post('/api/todo', async (req, res) => {
42.             try {
43.                 await connection.execute('INSERT INTO todo (description) VALUES
    (?)', [req.query.description]);
44.                 res.status(200).send();
45.             } catch (err) {
46.                 res.status(500).send();
47.             }
48.         });
49.
50.         app.listen(3000, () => console.log("Listening on port 3000 ..."));
51.     } catch (err) {
52.         console.error('Can not start server, error:', err);
53.     }
54. }
55.
56. setTimeout(() => main(), 5000);
```

A fenti kód 5-9. sora az adatbázis kapcsolat felépítéséhez szükséges adatokat olvassa be, környezeti változókból. A 12-17. sorok segítségével kapcsolódik, majd hoz létre egy megfelelő táblát az adatbázisban alkalmazásunk.

A kód további része egy HTTP szervert indít el a 3000-es porton, mely az alábbi műveleteket támogatja:

- `GET /api/todo` Minden teendő lekérdezése az adatbázisból.
- `POST /api/todo` Új teendő hozzáadása az adatbázishoz.
- `PUT /api/todo/:id/done` Meglévő teendő készre állítása.

Most hozzuk létre a `docker-tutorial/frontend/` könyvtárat, benne egy `index.html` fájllal:

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.
4.  <head>
5.      <meta charset="UTF-8">
6.      <meta http-equiv="X-UA-Compatible" content="IE=edge">
7.      <meta name="viewport" content="width=device-width, initial-scale=1.0">
8.      <link
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
    rel="stylesheet"
9.          integrity="sha384-
    EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC"
    crossorigin="anonymous">
10.     <title>TODO Application</title>
11. </head>
12.
13. <body>
14.     <div class="container mt-2" style="max-width: 640px;">
15.         <div class="row">
```

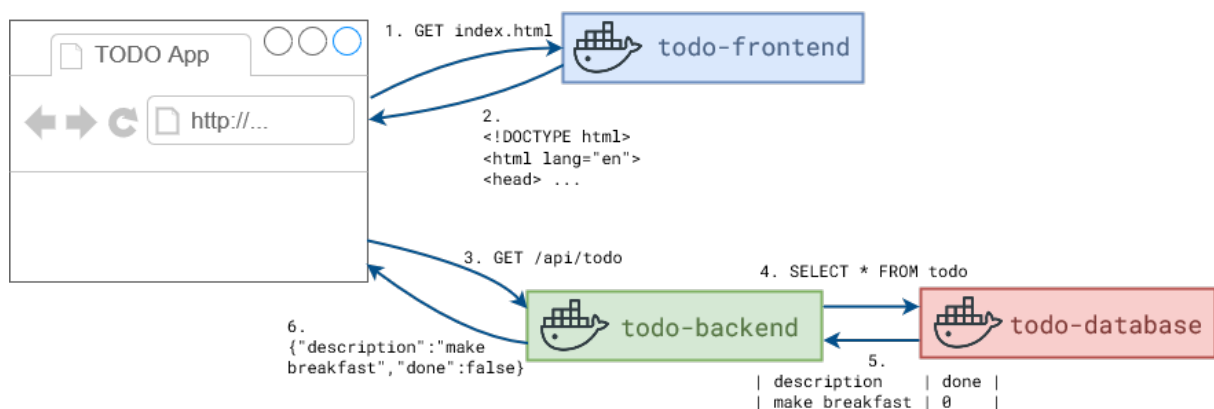
```
16.             <div class="col-md-12">
17.                 <h4>What To Do?</h4>
18.                 <hr>
19.                 <table id="todo-table" class="table table-striped">
20.                     <thead>
21.                         <tr>
22.                             <th>#</th>
23.                             <th>Description</th>
24.                             <th>Done</th>
25.                             <th></th>
26.                         </tr>
27.                     </thead>
28.                     <tbody></tbody>
29.                 </table>
30.             </div>
31.         </div>
32.
33.         <h5>New Thing To Do</h5>
34.         <div class="row">
35.             <div class="col-md-10">
36.                 <input type="text" id="description" class="form-control"
    placeholder="Description of your task ...">
37.             </div>
38.             <div class="col-md-2 text-end">
39.                 <button class="btn btn-outline-primary"
    onclick="saveTodo()">Save</button>
40.             </div>
41.         </div>
42.         <div class="mt-2">
43.             <small class="fst-italic">If the UI is unresponsive, press F12 for
    further information.</small>
44.         </div>
45.     </div>
46.     <script>
47.         const baseUrl = 'http://localhost:3000';
48.
49.         async function done(id) {
50.             const res = await fetch(`${baseUrl}/api/todo/${id}/done`, { method:
    'PUT' });
51.             if (res.ok) {
52.                 populateTable();
53.             }
54.         }
55.
56.         async function saveTodo() {
57.             const todoInput = document.querySelector('#description');
58.             const description = todoInput.value;
59.
60.             await fetch(`${baseUrl}/api/todo?` + new URLSearchParams({ description
    }), { method: 'POST' });
61.             await populateTable();
62.
63.             todoInput.value = '';
64.         }
65.
66.         async function populateTable() {
67.             const res = await fetch(`${baseUrl}/api/todo`);
68.             const json = await res.json();
69.
70.             const tbody = document.querySelector('#todo-table>tbody');
71.             tbody.innerHTML = '';
72.             json.forEach(todo => {
73.                 tbody.innerHTML += `<tr>
74.                     <td>${todo.id}</td>
75.                     <td>${todo.description}</td>
76.                     <td>${todo.done ? '✔' : '✕'}</td>
```

```
77.                     <td class="text-end">
78.                         ${todo.done ? '' : `<button class="btn btn-sm btn-outline-success" onclick="done(${todo.id})">Done it</button>`}
79.                     </td>
80.                 </tr>`
81.             });
82.         }
83.
84.         populateTable();
85.     </script>
86. </body>
87.
88. </html>
```

A fenti HTML fájl egy grafikus felületet ír le, melyen keresztül elérhetjük a szerverünk által biztosított műveleteket.

Konténerek létrehozása

A 3 konténer létrehozása előtt tekintsük át, hogy hogyan is kell működnie az alkalmazásunknak, mely konténerek fognak együttműködni egymással. Ezt a következő ábra mutatja be.



3. ábra. Példa alkalmazásunk felépítése.

- a `todo-frontend` konténer egy egyszerű fájlszerver lesz, mely jelen esetben egyetlen fájlt, az index.html-t szolgálja ki a kliensek számára.
- a `todo-backend` konténer a saját HTTP szerverünk, mely kéréseket fogad a klienstől, az adatbázissal kommunikál, és az onnan kapott értékeket visszaküldi a kliensnek.
- a `todo-database` konténer egy hagyományos relációs adatbázis (pl. MySQL) lesz.

Az alkalmazás működésének alapvető lépései a következők:

1-2. lépésben a kliens (egy tetszőleges böngésző) letölti az index.html fájl tartalmát, mely az alkalmazásunk grafikus felületét tartalmazza.

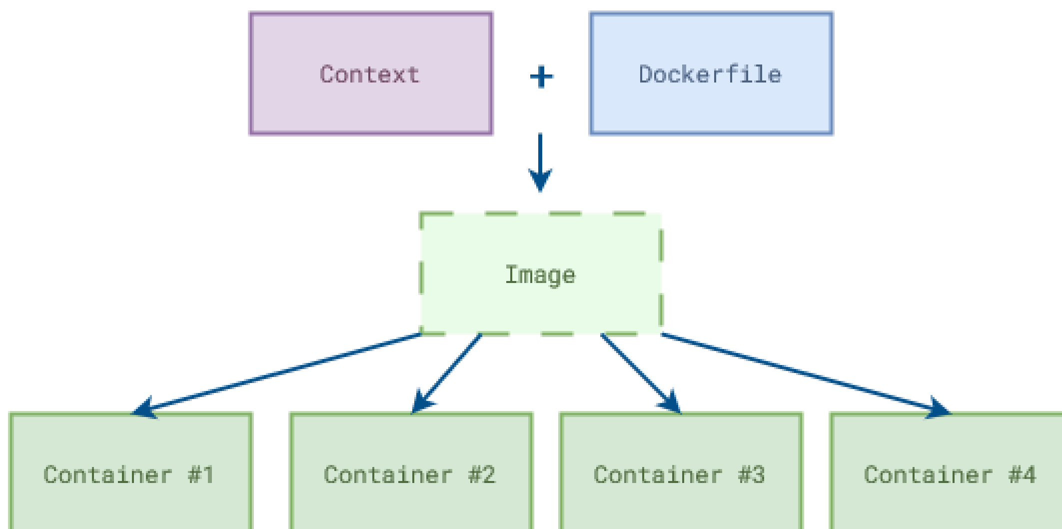
3. lépésben a kliens HTTP kérést indít a szerver felé (pl. lekérdezi az adatbázisban szereplő összes teendőt).

4. lépésben a szerver SQL lekérdezést hajt végre az adatbázisban, majd 5. lépésként megkapja az eredményeket, jellemzően valamilyen bináris formátumban.

6. lépésben a szerver valamilyen szabványos szöveges formátumba (pl. JSON) konvertálja az adatbázistól megkapott rekordokat, és visszaküldi ezt a kliensnek.

A kliens ezután megjelenítheti a szervertől érkezett adatokat.

Kezdésképp készítsük el a `todo-frontend` konténerhez tartozó képfájlt! Ehhez tekintsük át a Docker konténerek létrehozásának folyamatát a következő ábrán!



4. ábra. Docker konténerek létrehozásának folyamata.

A Docker konténerek alapjául szolgáló képfájl kontextus (jelen esetben ez az `index.html` fájlunkat jelenti) és `Dockerfile` segítségével állítható elő.

A képfájl a konténer statikus váza, egy recept arra vonatkozóan, hogy hogyan tudjuk a konténert létrehozni és elindítani.

Egy képfájlból akár több konténer is létrehozható (pl. abban az esetben lehet erre szükség, ha fájlszerverünket szeretnénk egy magyarországi és egy egyesült államokbeli adatközpontban is elindítani).

Kliens konténer létrehozása

A fenti képletből nálunk már csak egy `Dockerfile` elnevezésű fájl hiányzik, melyet hozzunk létre a `docker-tutorial/frontend` mappán belül, a következő tartalommal:

```
1.  FROM nginx:1.24.0
2.  COPY index.html /usr/share/nginx/html
```

Docker-fájlunk mindössze 2 sorból áll, tekintsük át, mire szolgálnak ezek!

Képfájlunk a hivatalos `nginx` image 1.24.0-s verzióját használja fel alapként. Az nginx egy webszerver, mely alapértelmezetten úgy van konfigurálva, hogy statikus fájlokat szolgáljon ki a 80-as porton. Ezen statikus fájlok a konténer saját fájlrendszerének `/usr/share/nginx/html` könyvtárában találhatók meg.

A 2. sor saját gépünkről a konténer fájlrendszerének `/usr/share/nginx/html` könyvtárába másolja át az `index.html` fájlt.

Alkalmazásunk kliensoldala készen áll arra, hogy konténert hozzunk létre belőle. Ehhez először szükséges a konténer „receptjéül" szolgáló képfájlt létrehoznunk. Indítsuk el a következő parancsot egy terminálablakban, a `docker-tutorial` mappából:

```
docker build -t todo-frontend ./frontend
```

Ezután a `docker images` parancsot futtatva ellenőrizhetjük, hogy képfájlunk `todo-frontend` néven létrejött, a `frontend` mappában található `Dockerfile` és kontextus alapján.

Ebből a képfájlból a `docker run todo-frontend` paranccsal készíthetünk és indíthatunk konténert.

(Amennyiben csak létrehozni szeretnénk a konténert, használjuk a `docker create todo-frontend` parancsot! Ezt követően a `docker start <container_id>` és `docker stop <container_id>` parancsokkal futtathatók, illetve állíthatók le a konténerek.)

Ha konténerünk elindult, nyissunk egy másik terminálablakot (a már meglévőt ne zárjuk be!), és futtassuk a `docker ps` parancsot. A következőhöz hasonló kimenetet kell kapnunk:

```
PS C:\ > docker ps
CONTAINER ID       IMAGE              …      PORTS        NAMES
cfe8d31fe046       todo-frontend      …      80/tcp       amazing_cohen
```

A parancs az éppen futó konténereket listázza.

Konténerünk kapott egy azonosítót (cfe8d31fe046) és egy véletlenszerűen generált elnevezést (amazing_cohen), valamint látható, hogy a 80-as porton várja a beérkező kéréseket.

Fontos megjegyezni, hogy alapértelmezetten a konténer által használt portok a gazdagép számára nem elérhetőek, csak az azonos virtuális hálózaton található konténerek érhetik azt el. Számunkra ez a beállítás nem megfelelő, ugyanis a böngészőnk segítségével szeretnénk letölteni webalkalmazásunk kezdőoldalát, így képesnek kell lennünk a kommunikációra a konténerrel. Ennek érdekében először állítsuk le konténerünket a `docker stop cfe8d31fe046`

paranccsal! A konténerre azonosítójával vagy a Docker által generált névvel (jelen esetben „amazing_cohen”) is hivatkozhatunk.

A `docker ps` parancs újbóli futtatásakor konténerünk már nem látható a listában, mert le lett állítva. A `docker ps -a` parancsot futtatva azonban kilistázható az összes rendszerben szereplő konténer:

```
PS C:\ > docker ps -a
CONTAINER ID        IMAGE               …       PORTS         NAMES
cfe8d31fe046        todo-frontend       …       80/tcp        amazing_cohen
```

Az itt látható leállított konténereket a `docker start <container_id>` paranccsal akár újra is lehetne indítani, a `docker container rm <container_id>` paranccsal pedig a törlésükre van lehetőségünk.

Ahhoz, hogy konténerünket kívülről elérhessük, hozzunk létre és futtassunk egy új konténert, a `docker run -p 8080:80 todo-frontend` paranccsal!

A `-p` argumentum két port számot tartalmaz `<host_port>:<container_port>` formátumban. A fenti parancs segítségével azt adtuk meg, hogy a konténer 80-as portja a gazdagép 8080-as portjához legyen hozzárendelve. Ez azt jelenti, hogy mostantól webalkalmazásunk felhasználói felületét a http://localhost:8080/-as webcímet meglátogatva elérhetjük.

Példánk azonban még nem teljes, alkalmazásunk backend-jét is be kell üzemelnünk!

Szerver konténer létrehozása

Ha még nem tettük meg, állítsuk le a kliensoldalhoz tartozó konténert a `docker ps`, majd a `docker stop <container_id>` parancsok futtatásával!

Ezután létre kell hoznunk a szerveroldalunkhoz tartozó konténert is. Ehhez fontos ismerni azt, hogy szerveroldali kódunk hogyan kerül futtatható állapotba. Ennek lépései a következők:

1. Projektünk fejlesztői- és éles függőségeit telepítenünk kell. Ez tulajdonképpen kész csomagok letöltését jelenti az NPM-ből, a `backend` mappában található `package.json` fájl alapján.
2. Ha a függőségek települtek, akkor a projektet le kell fordítani TypeScript-ről JavaScript nyelvre. Ekkor a `build` könyvtárban közvetlenül futtatható .js fájlok jönnek létre.
3. Az így megkapott fájlokból össze kell állítani egy közvetlenül futtatható projektet:
    - 3.1. Egy tetszőleges mappába át kell másolni a `package.json` fájlt.
    - 3.2. A `package.json` fájl alapján telepíteni kell az éles függőségeket (a fejlesztői függőségeket nem, azokra csak a fordítás során volt szükség!).

3.3. Ezután az alkalmazás kódját tartalmazó .js (JavaScript) fájlokat is át kell másolni ebbe a mappába.

4. Az így létrehozott mappából a `node index.js` paranccsal futtatható a szerverünk.

Hozzunk létre egy `Dockerfile`-t a `backend` mappában, mely épp a fenti folyamatot írja le:

```
1.  # create a container for building the project
2.  FROM node:lts-hydrogen AS builder
3.
4.  WORKDIR /usr/app
5.  COPY package*.json ./
6.  RUN npm install
7.  COPY . .
8.  RUN npm run build
9.
10. # create a container for running the project
11. FROM node:lts-hydrogen AS deployment
12.
13. WORKDIR /usr/app
14. COPY package*.json ./
15. RUN npm install --only=prod
16. COPY --from=builder /usr/app/build ./
17.
18. WORKDIR /usr/app
19. EXPOSE 3000
20. CMD [ "node", "index.js" ]
```

Elsőre meglepő lehet, hogy a Dockerfile-ban több FROM utasítást is használunk. Ennek oka, hogy ún. multi-stage build-et (többlépcsős építés) hajtunk végre.

Minden `FROM` utasítással más képfájlt használhatunk alapként, és mindegyik `FROM` az összeállítás egy új szakaszát jelenti. Egy szakaszban szabadon átmásolhatunk az image-ünkbe fájlokat a korábbi szakaszok valamelyikéből. A végső képfájlba csak az utolsó szakaszban (11. sortól kezdődően) előállított eredmények kerülnek be.

A `Dockerfile`-ban szereplő 1-8. sorok bemásolják a `package.json` fájlt az image-be, majd telepítik az összes függőséget. Ezt követően a projekt teljes kódja bemásolásra kerül az image-be, majd megtörténik a TypeScript → JavaScript fordítás.

A 11-20. sorok új szakaszt indítanak, elsőként a `package.json` fájlt bemásolva az image-be, majd csak az éles függőségeket telepítve. Ezt követően az előző szakaszból (mely `builder`-nek lett elnevezve) átmásoljuk a jelenlegibe a lefordított fájlokat.

Mindkét szakasz a hivatalos `node` képfájl `lts-hydrogen` verziójából indul ki, ez azt jelenti, hogy a konténeren belül a `node` és `npm` parancsok elérhetők.

Az `EXPOSE` utasítás dokumentálja, hogy az alkalmazás a 3000-s porton fog figyelni. A `CMD` utasítás a szerver indításához szükséges parancsot írja le, ez fog elindulni a konténer indításakor.

Készítsünk képfájlt a szerveroldalunk kódjából és Dockerfile-jából:

```
docker build -t todo-backend ./backend
```

A `docker images` paranccsal ellenőrizhetjük, hogy a képfájl sikeresen létrejött-e.

Ha szerverünket futtatjuk a `docker run todo-backend` paranccsal, egyelőre azt kell látnunk, hogy az adatbázis hiánya miatt hibára fut és megáll.

Adatbázis konténer létrehozása

Adatbázisként a készen elérhető `mysql` képfájlt fogjuk használni: https://hub.docker.com/_/mysql

Ehhez futtassuk a `docker pull mysql:8.0.33-debian` parancsot!

Mielőtt adatbázisunkat elindítanánk, fontos, hogy a Docker egy újabb funkciójával megismerkedjünk. A konténereket alapvetően „állapotmentes" működésre tervezték, ez azt is jelenti, hogy bármikor eldobhatónak és egy kezdeti, tiszta állapotból újraindíthatónak kell lenniük, anélkül, hogy ez a rendszer egészére komoly hatást gyakorolna. Nem célszerű tehát, ha olyan állapotot tartalmaznak, amit fontos lenne hosszabb ideig megőrizni.

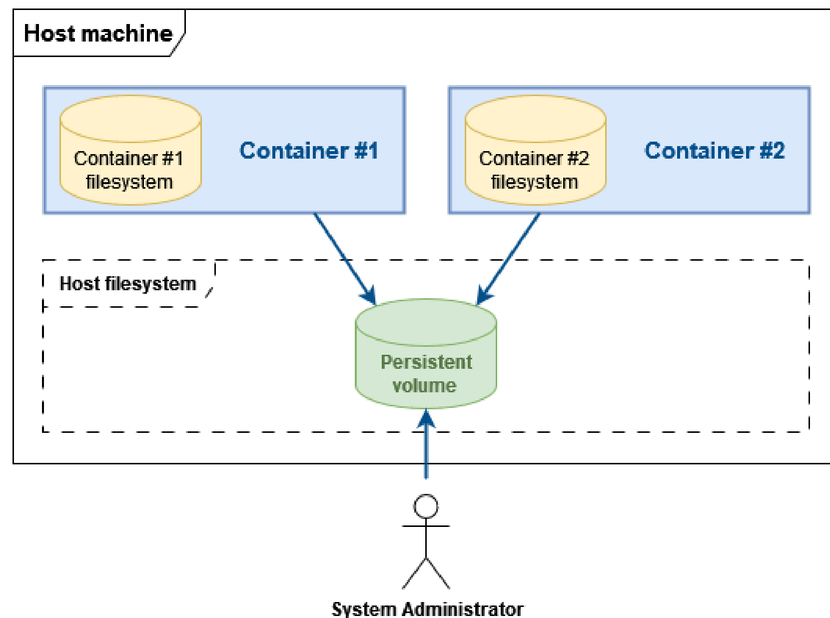
A fentiek alapján érezhetjük, hogy ez a fajta állapotmentesség az adatbázisok esetében nem állja meg a helyét, hiszen relációs adatbázisunk célja éppen az, hogy a tárolt adatokat hosszabb távon, perzisztens módon megőrizze és szükség esetén rendelkezésre bocsássa. Adatbázis konténerünk tehát nem lesz állapotmentes, így nem érvényesülnek rá a könnyű leállíthatóságból, eldobhatóságból, újra létrehozhatóságból adódó előnyök sem.

Ennek a problémának a megoldása miatt van lehetőségünk ún. perzisztens tárolók (volume) létrehozására. A perzisztens tárolók nem a konténer virtualizált – és kívülről elérhetetlen - fájlrendszerén, hanem közvetlenül a hoszt gépen kerülnek tárolásra, ahogy az az alábbi ábrán is látható.

Perzisztens tárolókat nevük és opcionálisan tulajdonságaik (pl. méretük) megadásával hozhatunk létre. A tároló nevének egyedinek kell lennie.

A konténer elindításakor meghatározhatjuk, hogy a konténer fájlrendszerének mely könyvtárát rendeljük hozzá a tárolóhoz. Így ezek a fájlok nem a konténer, hanem a hoszt gép fájlrendszerében lesznek tárolva, innen lesznek elérhetők és módosíthatók.

Ezt a folyamatot a Docker Engine kezeli, a konténer számára teljesen transzparens módon.



5. ábra. Perzisztens tároló Docker-ben.

A volume-ok használatának legfontosabb előnye, hogy az adatok – mivel a hoszt gépen vannak tárolva – a konténerek leállítása, törlése esetén is megőrződnek. Adatbázis konténerünk így „állapotmentesíthető", hiszen az adatbázis állapotát (pl. konfiguráció, táblák, rekordok) nem a konténer fájlrendszere tárolja.

A volume-ok használatának további előnye, hogy közvetlen fájlmegosztást biztosítanak a konténerek között. Ugyanaz a volume több konténerhez is hozzárendelhető, így azonos fájlokat érhetnek el a különböző alkalmazás-környezetek (adatbázisunknál ennek nincs jelentősége, azonban egyéb esetekben, pl. központi helyre történő naplózás vagy fájl alapú adatcsere esetén fontos előny lehet).

Teljesítmény szempontjából is előnyök mutatkoznak, hiszen a fájlrendszer virtualizációja a perzisztens tároló használatával szükségtelenné válik, legalábbis a tárolóhoz rendelt könyvtár esetében.

Fontos továbbá megjegyezni, hogy mivel az adatokat közvetlenül a hoszt gép tárolja, így könnyedén, a konténerbe történő belépés nélkül elérhetők a szükséges fájlok pl. biztonsági mentés készítése céljából.

Hozzunk is létre egy volume-ot `todo-data` néven, melyben az adatbázis konfigurációját és adatait fogjuk tárolni: `docker volume create todo-data`

Konténerek összehangolása

Futtatás kézzel

Próbáljuk meg a konténereinket kézzel futtatni! Első ránézésre ezek a parancsok bonyolultnak tűnhetnek, de később kipróbálunk egy sokkal hatékonyabb módszert is a rendszer üzembehelyezésére.

A kliens elindítása viszonylag egyszerű, a következő parancs szükséges hozzá:

```
docker run -p 8080:80 --detach --name todo-frontend-manual todo-frontend
```

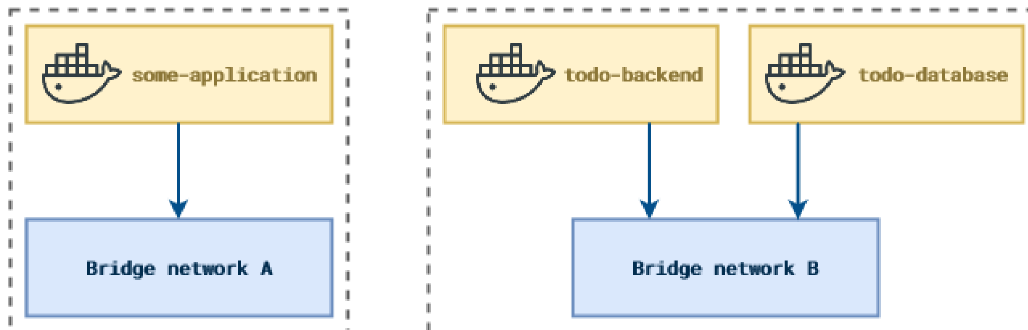
Az indítás során a következő beállításokat alkalmaztuk:

- beállítottuk, hogy a konténer 80-as portja kerüljön összerendelésre a hoszt gép 8080-as portjával (így utóbbi porton keresztül elérhetjük alkalmazásunk grafikus felületét),
- konténerünknek a `todo-frontend-manual` elnevezést adtuk, ezen hivatkozhatunk rá a későbbiekben,
- a `--detach` beállítás miatt konténerünk a háttérben indult el, nem látjuk a kimenetét a terminál ablakban:
    - az indítás sikerességéről a `docker ps` paranccsal győződhetünk meg; amennyiben a konténer nem fut, indítsuk újra `detach` beállítás nélkül, és ellenőrizzük a látott hibaüzenetet,
    - a `docker logs todo-frontend-manual` parancs segítségével kiírathatjuk a konténer által eddig generált kimenetet,
    - a `docker logs -f todo-frontend-manual` paranccsal valós időben követhetjük a kimenet változásait.

Ha a kliensünk sikeresen elindult, bírjuk működésre a szerveroldalt is!

A szerveroldal esetében 2 konténernek (`todo-database` és `todo-backend`) kell együttműködnie egymással. Ahhoz, hogy a köztük történő kommunikáció megvalósulhasson, elsőként fontos áttekintenünk a Docker hálózatkezelésének alapjait!

A Docker többféle hálózatkezelési módszert támogat, melyek közül ebben a példában az ún. bridge network lesz segítségünkre.



6. ábra. Hálózat-virtualizáció Dockerben.

Ahogy az a képen is látható, az azonos hálózathoz hozzákapcsolt konténerek (`todo-backend`, `todo-database`) egymással kommunikálni tudnak, azonban a különböző hálózaton lévő konténerek (pl. a `some-application` és a `todo-backend`) nem érhetik el egymást. Egy konténer egyébként akár több hálózathoz is kapcsolódhat.

Amennyiben az elindított konténert nem kapcsoljuk hálózathoz, az alapértelmezett `bridge` hálózathoz fog tartozni. Fontos megjegyezni, hogy ezen az alapértelmezett hálózaton a konténerek kizárólag közvetlenül, IP-cím alapján érhetik el egymást, a DNS szolgáltatás nem biztosított számukra (a DNS a névfeloldásért szerepel, pl. az `uni-miskolc.hu` domainből előállítja a `193.6.10.2` IP-címet, ezt ki is lehet próbálni, pl. itt: https://toolbox.googleapps.com/apps/dig/#A/).

A DNS szolgáltatás hiánya bonyolult feladattá teszi a konténereink kommunikációjának megvalósítását, hiszen például egy újraindított konténer esetében semmi nem garantálja azt, hogy újra a korábbi IP-címét kapja meg a hálózaton.

Amennyiben saját hálózatot („user-defined bridge”) hozunk létre, azon belül a névfeloldás automatikusan biztosított, az egyes konténerekre konkrét IP-címük helyett elegendő elnevezésükkel (pl. `todo-database`) hivatkoznunk, így az esetleges IP-cím váltás sem okoz gondot a kommunikációban.

A fentiek miatt hozzunk is létre egy saját bridge-hálózatot a következő parancs segítségével: `docker network create todo-webapp`

Ezt követően minden készen áll arra, hogy konténereinket is elindítsuk, kezdjünk az adatbázissal:

```
docker run -e "MYSQL_DATABASE=docker_tutorial" -e "MYSQL_ROOT_PASSWORD=Porcica1." --network todo-webapp -v "todo-data:/var/lib/mysql" -d --name "todo-database-manual" mysql:8.0.33-debian
```

Az indításkor a következő beállításokat határoztuk meg:

- az adatbázis néhány alapvető beállítását környezeti változók segítségével adhatjuk meg (`-e` kapcsoló):
    - a `MYSQL_DATABASE` környezeti változóban megadott adatbázis automatikusan létrejön a MySQL indításakor,
    - a `MYSQL_ROOT_PASSWORD` környezeti változóban az alapértelmezett `root` felhasználó jelszavát adhatjuk meg,
- a `--network` kapcsoló segítségével az előzőleg létrehozott `todo-webapp` hálózatra kapcsoljuk a konténert,
- a `-v` kapcsoló segítségével a korábban létrehozott `todo-data` volume-hoz kapcsoljuk a konténer fájlrendszerének `/var/lib/mysql` könyvtárát (ez tartalmazza az adatbázis állapotát),

Megfigyelhető, hogy a `-p` kapcsolót a fenti parancsban nem használtuk. Azért tettünk így, mert nincs szükségünk arra, hogy az adatbázist közvetlenül, a saját gépünkről elérjük. Csak a `todo-backend` konténerrel fogunk kommunikálni, ami a közös hálózatuk miatt el fogja érni az adatbázist.

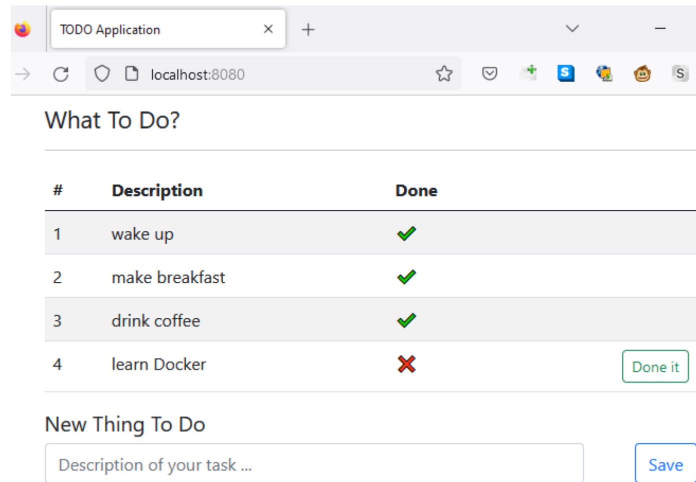
Az adatbázis sikeres indítását a `docker ps` paranccsal ellenőrizzük, majd indítsuk el a szervert is:

```
docker run -e "DB_HOST=todo-database-manual" -e "DB_PASSWORD=Porcica1." -e
"DB_DATABASE=docker_tutorial" --network todo-webapp -p 3000:3000 -d --name "todo-
backend-manual" todo-backend
```

A szerver számára a következő beállításokat határoztuk meg:

- környezeti változókban átadtunk minden szükséges információt az adatbázishoz történő kapcsolódáshoz,
- hozzákapcsoltuk a konténert a közös virtuális hálózatra,
- a hoszt gép 3000-es portjához rendeltük a konténer 3000-es portját (ez azt jelenti, hogy kívülről ezen a porton elérhetjük a szervert).

Most keressük fel a böngészőben a http://localhost:8080/ webcímet, ahol a következő képen látható felületet kell látnunk. Próbáljuk ki az alkalmazást néhány feladat hozzáadásával, készre állításával!



7. ábra. TODO alkalmazás felülete.

Jelenleg a konténerek futtatása lehetséges ugyan kézzel vagy egyedi scriptek segítségével, de az előbbi megoldás körülményes és sok odafigyelést igényel, utóbbi megoldás pedig könnyen vezethet átláthatatlan és karbantarthatatlan egyedi megoldásokhoz.

Ennek a problémának a megoldására találták ki a Docker Compose elnevezésű eszközt, mely segítségével egyetlen konfigurációs állományban leírhatjuk a több konténerből álló alkalmazásunk futtatásához szükséges összes beállítást, majd egyszerű parancsokkal indíthatjuk el és állíthatjuk le rendszerünket.

Mielőtt erre rátérnénk, állítsuk le, és töröljük futó konténereinket a következő parancs segítségével:

```
docker container rm todo-backend-manual todo-database-manual todo-frontend-manual --force
```

Futtatás Docker Compose segítségével

A Docker Compose-t a Docker Desktop alapértelmezetten tartalmazza, így már telepítettük a számítógépünkre. Egyetlen dolgunk tehát egy konfigurációs fájl létrehozása a `docker-tutorial/` mappában, `docker-compose.yml` néven, a következő tartalommal:

```
1.  volumes:
2.    todo-data:
3.      external: true
4.
5.  services:
6.    database:
7.      container_name: todo-database
8.      image: mysql:8.0.33-debian
9.      environment:
10.       - MYSQL_DATABASE=docker_tutorial
11.       - MYSQL_ROOT_PASSWORD=Porcica1.
12.     volumes:
```

```
13.         - "todo-data:/var/lib/mysql"
14.
15.   backend:
16.     container_name: todo-backend
17.     build:
18.       context: ./backend
19.       dockerfile: Dockerfile
20.     environment:
21.       - DB_HOST=todo-database
22.       - DB_PORT=3306
23.       - DB_USER=root
24.       - DB_PASSWORD=Porcica1.
25.       - DB_DATABASE=docker_tutorial
26.     ports:
27.       - 3000:3000
28.
29.   frontend:
30.     container_name: todo-frontend
31.     build:
32.       context: ./frontend
33.       dockerfile: Dockerfile
34.     ports:
35.       - 8080:80
```

Ebben a fájlban egy komplex, több szolgáltatásból álló rendszer beállításait, működését írhatjuk le. Megfigyelhetjük, hogy konfigurációs fájlunk közel ugyanazokat a beállításokat tartalmazza, mint a korábban kiadott `docker run` parancsaink.

A Docker Compose a rendszer elindításakor egy saját hálózatot hoz létre, melyre csatlakoztatja az összes meghatározott szolgáltatást (konténert).

Az 1-3. sorokban a rendszer által használt perzisztens tároló van megadva. Ezt a rendszer első indítása előtt létre kell hozni (mi ezt már korábban megtettük).

Az 5-35. sorokban a rendszer szolgáltatásai vannak felsorolva, melyek külön konténerekként indulnak el. Rendszerünk 3 konténerből áll, melyeket `database`, `backend` és `frontend` elnevezésekkel láttunk el.

Minden szolgáltatás esetében fel vannak sorolva a konténer létrehozásához szükséges beállítások (pl. a kész képfájl neve vagy a kontextus és a Dockerfile, melyekből a képfájl létrehozható; a konténerhez rendelt környezeti változók; volume-ok; port hozzárendelések).

Rendszerünket a `docker-compose up` paranccsal indíthatjuk el (a `docker-tutorial/` mappában állva), melynek hatására a terminálban megjelenik konténereink kimenete. Amennyiben minden sikeresen elindult, a http://localhost:8080/ címen elérhető a webalkalmazásunk. Ha a `--detach` kapcsolót (mely a háttérben indítja el a konténereket) nem használtuk, a Ctrl+C billentyűkombinációval állíthatjuk le a rendszert.

Amennyiben a konténereket a háttérben indítottuk el, használjuk a `docker-compose down` parancsot a leállításhoz!

Kód elérhetősége

A bemutatóban szereplő alkalmazás kódja elérhető GitHub-on: aron123/docker-tutorial