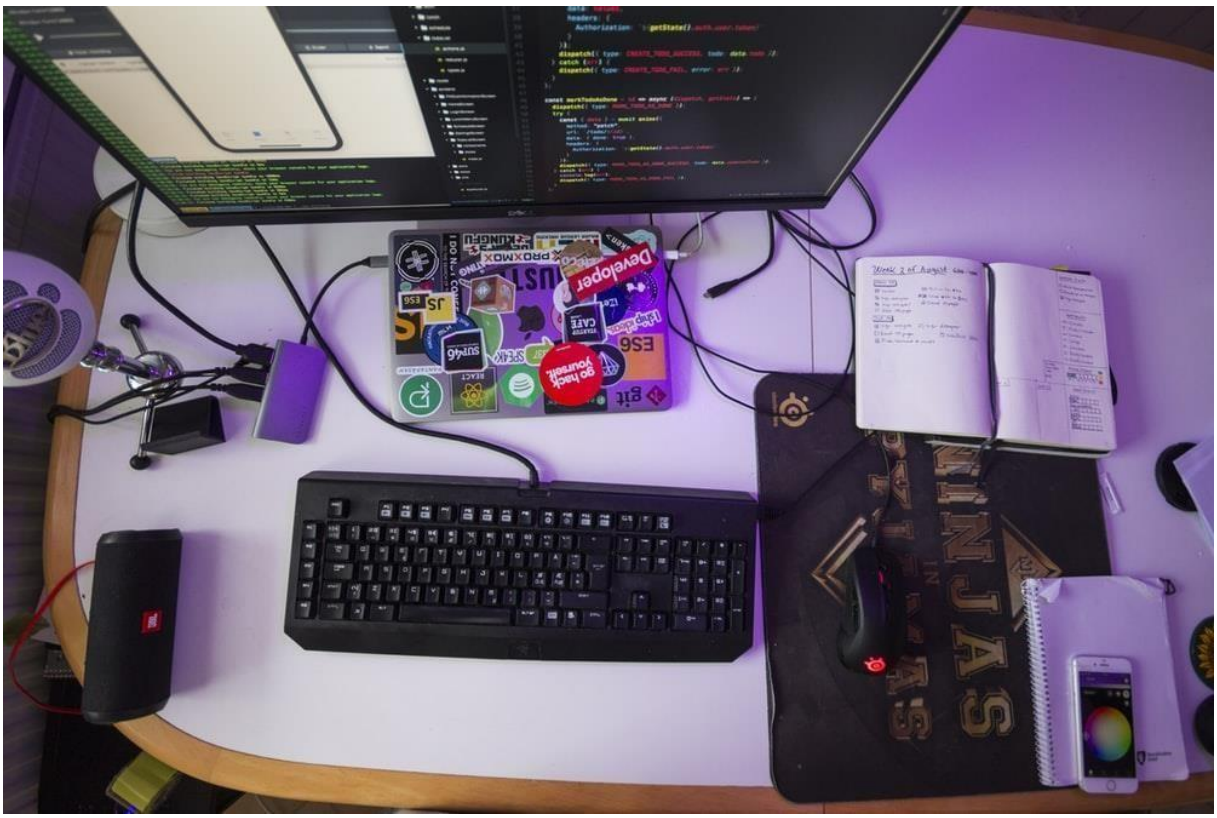
Programowanie w chmurze. Laboratorium 1. Konfiguracja środowiska developerskiego Visual Studio Code. Wirtualizacja, dockeryzacja oraz NodeJS.



Wprowadzenie

W pierwszej części tego przewodnika utworzymy prostą aplikację internetową w Node.js, następnie zbudujemy obraz Docker dla tej aplikacji, a na koniec utworzymy instancję kontenera z tego obrazu. Oczywiście można też prostą aplikację uruchamiać w środowisku Node.JS również lokalnie. Jednak chcemy nauczyć się tworzyć proste obrazy Docker.

Docker pozwala spakować aplikację z jej środowiskiem i wszystkimi jej zależnościami do "pudełka", zwanego kontenerem. Zazwyczaj kontener składa się z aplikacji działającej w wersji systemu operacyjnego Linux z dodatkiem stripped to-basics. Obraz jest planem kontenera, kontener jest uruchomioną instancją obrazu.

INSTALACJA NIEZBĘDNEGO OPROGRAMOWANIA

W systemie Windows (na maszynie wirtualnej lub lokalnie) zainstaluj następujące oprogramowanie:

1. **Docker Desktop:**
    1. Pobierz instalator ze strony Docker Desktop.
    2. Uruchom instalator i postępuj zgodnie z instrukcjami na ekranie.
    3. Po zakończeniu instalacji uruchom Docker Desktop i sprawdź, czy aplikacja działa poprawnie. Wymaga to zainstalowanego systemu WSL 2 (Windows Subsystem for Linux 2), więc upewnij się, że masz go włączonego.

2. **Node.js:**

    1. Odwiedź stronę Node.js i pobierz wersję LTS (Long Term Support).
    2. Uruchom instalator i zaakceptuj domyślne ustawienia instalacji.
    3. Aby sprawdzić, czy Node.js został poprawnie zainstalowany, otwórz terminal (np. PowerShell lub Command Prompt) i wpisz:

        node --version

        Powinna wyświetlić się wersja Node.js.

3. **Visual Studio Code:**

    1. Pobierz instalator ze strony Visual Studio Code.
    2. Uruchom instalator i postępuj zgodnie z instrukcjami. Upewnij się, że zaznaczasz opcję "Add to PATH", aby móc uruchamiać Visual Studio Code z terminala.
    3. Po zakończeniu instalacji uruchom Visual Studio Code i upewnij się, że środowisko jest gotowe do pracy.
    4. Możesz doinstalować wiele wtyczek (extensions), w tym te, które wykorzystują sztuczną inteligencję do wspomagania programowania np.: GitHub Copilot, CodeWhisperer (AWS)

Pierwsza aplikacja chmurowa

Tworzymy **serwer Node.js** z użyciem **Express.js**, który obsługuje stronę internetową, wykorzystując silnik szablonów **EJS** oraz pliki statyczne. Express.js służy developerom do usprawnienia pracy i rozszerzenia funkcjonalności samego Node.js. Ułatwia i automatyzuje skomplikowane operacje związane z systemami API, zarządza żądaniami HTTP i HTTPS, sesjami oraz routingiem, a także obsługą błędów. Jest chętnie wykorzystywany ze względu na dużą elastyczność, minimalizm i skalowalność w pracy nad projektem. Express opisuje się jako minimalistyczną i elastyczną platformę do tworzenia aplikacji WWW w technologii Node.js, która oferuje solidny zestaw funkcji dla aplikacji WWW i mobilnych.

PRZYGOTOWANIE PROJEKTU W VISUAL STUDIO CODE

2. Otwórz Visual Studio Code.

3. Utwórz nowy folder node-web-app i otwórz go w VS Code.

4. Otwórz terminal w VS Code (Terminal > New Terminal).

5. Zainicjalizuj projekt Node.js, wpisując w terminalu:

npm init -y

6. Zainstaluj Express:

npm install express

7.Utwórz plik server.js i wklej poniższy kod:

```
const express = require('express');
```

```
const path = require('path');
const morgan = require('morgan');

const app = express();
const port = process.env.PORT || 8080;

// Middleware
app.use(morgan('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, 'public')));

// Ustawienie silnika szablonów EJS
app.set('view engine', 'ejs');

// Główna trasa
app.get('/', (req, res) => {
  res.render('index', { title: 'Strona główna', message: 'Witaj świecie!' });
});

// Start serwera
app.listen(port, () => {
  console.log(`Aplikacja nasłuchuje na porcie ${port}`);
});
```

8. Uruchom aplikację, wpisując w terminalu:

```
node server.js
```

9. Otwórz przeglądarkę i przejdź pod adres http://localhost:8080, aby zobaczyć komunikat Witaj świecie!.

Omówienie kodu

Middleware

Middleware to funkcje pośredniczące, które przetwarzają żądania HTTP przed dotarciem do obsługi końcowej, takiej jak app.get('/'). Middleware może:

- Modyfikować obiekt req (żądanie) i res (odpowiedź).
- Kończyć obsługę żądania, np. poprzez res.send().
- Przekazywać żądanie do kolejnej funkcji za pomocą next().

Morgan – middleware do logowania

Morgan to narzędzie do logowania żądań HTTP. Użycie opcji dev powoduje wyświetlanie podstawowych informacji o żądaniach, takich jak metoda żądania, kod statusu i czas odpowiedzi.

Obsługa danych JSON i formularzy

- express.json() umożliwia odczyt danych JSON przesyłanych w ciele żądań metodami POST i PUT.
- express.urlencoded({ extended: true }) pozwala na obsługę bardziej złożonych struktur, takich jak tablice i obiekty zagnieżdżone, w formularzach przesyłanych metodą POST.

Obsługa plików statycznych

Funkcja express.static(path.join(__dirname, 'public')) ustawia katalog public/ jako źródło plików dostępnych dla użytkownika, takich jak pliki CSS, skrypty JavaScript oraz obrazy.

Silnik szablonów EJS

Express wykorzystuje silnik szablonów **EJS (Embedded JavaScript Templates)**, który pozwala dynamicznie generować strony HTML. Dzięki temu możliwe jest przekazywanie danych do szablonów i ich dynamiczne renderowanie na stronie.

ROZSZERZENIE APLIKACJI

---

1. Dodaj dodatkowe trasy:

```
app.post('/', (req, res) => {
  res.send('Got a POST request');
});

app.put('/user', (req, res) => {
  res.send('Got a PUT request at /user');
});

app.delete('/user', (req, res) => {
```

```
  res.send('Got a DELETE request at /user');
});
```

2. Utwórz katalog public i dodaj do niego pliki statyczne (np. obrazy, CSS, JavaScript). Strona internetowa powinna być typu „HOMEPAGE/PORTFOLIO" i może zawierać np. ofertę, galerię, formularze kontaktowe, księgę gości itp.

```
public/
├── css/
|   └── style.css
├── js/
|   └── script.js
└── images/
```

3. Dodaj proste menu z podstronami statycznymi i zaimplementuj ich routing.
4. Uruchom aplikację i sprawdź, czy działa poprawnie.
5. Zweryfikuj obraz Docker i działanie kontenera.
6. Pamiętaj, że przy każdej zmianie kodu aplikacji musisz ponownie zbudować obraz Docker i uruchomić nowy kontener, aby zmiany zostały uwzględnione. Możesz to zrobić, uruchamiając polecenie:
   docker build -t <twoja_nazwa_uzytkownika>/node-web-app .
7. Po zbudowaniu obrazu, należy uruchomić nowy kontener, aby aplikacja działała z najnowszą wersją kodu.

DODANIE SZABLONÓW EJS

EJS (Embedded JavaScript) to popularny system szablonów dla Node.js, który pozwala na dynamiczne generowanie HTML. Dodajmy go do naszej aplikacji:

1. Zainstaluj EJS: *npm install ejs*

2. Skonfiguruj Express do używania EJS:

```
app.set('view engine', 'ejs');
```

3. Utwórz katalog views i dodaj pliki szablonów, np. index.ejs:

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
</head>
<body>
  <h1><%= message %></h1>
</body>
```

</html>

4. Zmodyfikuj trasę w server.js:

```
app.get('/', (req, res) => {
  res.render('index', { title: 'Strona główna', message: 'Witaj w mojej aplikacji!' });
});
```

5. Dodajmy prosty formularz kontaktowy. Zainstaluj body-parser do obsługi danych formularza:
npm install body-parser
6. Dodaj konfigurację w server.js:

```
const express = require('express'); const
bodyParser = require('body-parser'); const
app = express();

app.use(bodyParser.urlencoded({ extended: true })); app.use(bodyParser.json());
```

7. Utwórz nowy szablon contact.ejs w katalogu views:

```
<form action="/submit-form" method="POST">
  <input type="text" name="name" placeholder="Imię">
  <input type="email" name="email" placeholder="Email">
  <textarea name="message" placeholder="Wiadomość"></textarea>
  <button type="submit">Wyślij</button>
</form>
```

8. Dodaj nowe trasy w server.js: `app.get('/contact', (req, res) => { res.render('contact', { title: 'Kontakt' });`

```
});

app.post('/submit-form', (req, res) => {
console.log(req.body);
  res.send('Formularz został wysłany!');
});
```

9. Dodajmy middleware do logowania żądań http. Zainstaluj morgan:
npm install morgan

10. Dodaj konfigurację w server.js:

```
const morgan = require('morgan'); app.use(morgan('dev'));
```

11. Dodajmy podstawową obsługę błędów:

```
// Obsługa 404 app.use((req,
res, next) => {
```

```
  res.status(404).render('404', { title: 'Nie znaleziono' });
});

// Obsługa innych błędów app.use((err,
req, res, next) => {
console.error(err.stack);
  res.status(500).render('error', { title: 'Błąd serwera' });
});
```

12. Sprawdź strukturę katalogów i plików w naszej aplikacji. Zorganizuj następnie naszą aplikację w bardziej uporządkowaną strukturę:

```
projekt/
  ├── node_modules/
  ├── public/
  |   ├── css/
  |   ├── js/
  |   └── images/
  ├── views/
  |   ├── partials/
  |   ├── index.ejs
  |   ├── contact.ejs
  |   ├── 404.ejs
  |   └── error.ejs
  ├── routes/
  |   └── index.js
  ├── server.js
  ├── package.json
  └── Dockerfile
```

DOCKERYZACJA APLIKACJI

---

1. Utwórz plik Dockerfile w katalogu projektu i wklej poniższą zawartość:

```
FROM node:16

WORKDIR /usr/src/app

# Kopiowanie plików package.json i package-lock.json do kontenera
COPY package*.json ./

# Instalowanie zależności aplikacji
RUN npm install

# Kopiowanie całego kodu aplikacji do kontenera
COPY . .
```

```
# Otwieranie portu 8080
EXPOSE 8080

# Komenda uruchamiająca aplikację
CMD ["node", "server.js"]
```

2. Utwórz plik .dockerignore i dodaj do niego:

```
node_modules
npm-debug.log
.DS_Store
*.log
```

3. Zbuduj obraz Dockera:

```
docker build -t <twoja_nazwa_uzytkownika>/node-web-app .
```

4. Uruchom kontener:

```
docker run -p 8080:8080 -d <twoja_nazwa_uzytkownika>/node-web-app
```

5. Otwórz przeglądarkę i przejdź pod adres http://localhost:8080 aby zobaczyć swoją stronę internetową. Możesz również przetestować endpoint /api, wpisując w przeglądarce: http://localhost:8080/api

6. Pamiętaj, że przy każdej zmianie kodu aplikacji musisz ponownie zbudować obraz Docker i uruchomić nowy kontener, aby zmiany zostały uwzględnione. Możesz to zrobić, uruchamiając polecenie:
   ```
   docker build -t <twoja_nazwa_uzytkownika>/node-web-app .
   ```
7. Po zbudowaniu obrazu, należy uruchomić nowy kontener, aby aplikacja działała z najnowszą wersją kodu.

Ten Dockerfile wykonuje następujące kroki:

- Używa oficjalnego obrazu Node.js w wersji 16 jako bazowego obrazu dla kontenera.
- *COPY package.json ./**- Kopiuje pliki package.json oraz package-lock.json do kontenera. Dzięki temu możliwa jest instalacja zależności przed skopiowaniem całego kodu aplikacji. To przyspiesza proces budowy obrazu, ponieważ zależności są instalowane tylko wtedy, gdy te pliki się zmienią.
- RUN npm install - Instaluje zależności aplikacji w kontenerze na podstawie plików package.json i package-lock.json.
- EXPOSE 8080 - Informuje Dockera, że aplikacja wewnątrz kontenera będzie nasłuchiwała na porcie 8080, co pozwala na przekierowanie portów pomiędzy kontenerem a hostem.

- CMD ["node", "server.js"] - Określa polecenie, które ma zostać uruchomione w kontenerze — w tym przypadku jest to aplikacja Node.js uruchomiona za pomocą server.js.

PODSTAWOWE KOMENDY DOCKER:

- docker ps - wyświetla uruchomione kontenery
- docker images - wyświetla dostępne obrazy
- docker stop <container_id> - zatrzymuje kontener
- docker rm <container_id> - usuwa kontener
- docker rmi <image_id> - usuwa obraz

KORZYSTANIE Z DOCKER DESKTOP:

---

1. Uruchom Docker Desktop.
2. Przejdź do zakładki "Containers" aby zarządzać kontenerami.
3. W zakładce "Images" znajdziesz swoje obrazy Docker.
4. Użyj interfejsu graficznego do uruchamiania, zatrzymywania i usuwania kontenerów oraz obrazów.

SERVERLESS

---

1. Aby przekształcić tę aplikację w wersję serverless, należy wykonać następujące kroki:
2. Zainstaluj niezbędne pakiety:

npm install serverless serverless-http express

3. W pliku server.js zmień kod, aby używał serverless-http, co umożliwi uruchomienie aplikacji w środowisku serverless:

```
const serverless = require('serverless-http');
const express = require('express');
const app = express();

// Konfiguracja EJS
app.set('view engine', 'ejs');
```

// Dodaj pozostałe konfiguracje i trasy (np. /, /contact itp.)

module.exports.handler = serverless(app);

4. Stwórz plik konfiguracyjny serverless.yml w głównym katalogu aplikacji, który określi usługę, dostawcę (np. AWS), oraz funkcje:

```
service: moja-aplikacja-serverless

provider:
  name: aws
  runtime: nodejs14.x
  stage: dev
  region: eu-central-1

functions:
  app:
    handler: server.handler
    events:
      - http:
          path: /
          method: ANY
      - http:
          path: /{proxy+}
          method: ANY
```

5. W pliku package.json dodaj odpowiednie skrypty do uruchamiania aplikacji lokalnie oraz do jej wdrażania na serwerze:

```
"scripts": {
  "start": "serverless offline start",
  "deploy": "serverless deploy"
}
```

5. Zainstaluj plugin serverless-offline do lokalnego testowania:

```
npm install --save-dev serverless-offline
```

6. Dodaj plugin do serverless.yml:

```
plugins:
  - serverless-offline
```

7. Aby przetestować aplikację lokalnie, użyj:

```
npm start
```

Aplikacja będzie działała lokalnie na porcie 3000 (domyślnie dla serverless-offline).

8. W środowisku serverless może być konieczne dostosowanie ścieżek do szablonów EJS. Dodaj poniższy kod do pliku server.js, aby wskazać lokalizację szablonów:

```
const path = require('path');
app.set('views', path.join(__dirname, 'views'));
```

9. Przy korzystaniu z EJS w środowisku serverless, może być konieczne dostosowanie ścieżek do szablonów:

```
const path = require('path');
app.set('views', path.join(__dirname, 'views'));
```

10. Rozważ dodanie sekcji custom w serverless.yml dla lepszej konfiguracji serverless-offline:

```
custom:
  serverless-offline:
    httpPort: 3000
    noPrependStageInUrl: true
```

Po wykonaniu tych kroków, Twoja aplikacja Express.js zostanie przekształcona w wersję serverless, którą można wdrożyć np. w usłudze AWS Lambda. Aplikacja będzie reagować na żądania HTTP za pośrednictwem API Gateway, a Ty nie będziesz musiał zarządzać serwerami. Pamiętaj, że musisz mieć skonfigurowane konto AWS i odpowiednie uprawnienia, aby móc wdrożyć aplikację. Możesz też rozważyć dodanie obsługi środowiska, zmiennych środowiskowych i innych konfiguracji specyficznych dla Twojej aplikacji.

11. Istnieje kilka darmowych opcji serverless, które mogą być odpowiednie dla Twojego projektu. Oto kilka popularnych platform oferujących darmowe plany:

Vercel:

- Oferuje darmowy plan z hojnymi limitami
- Doskonały dla aplikacji frontendowych i serverless funkcji
- Łatwa integracja z repozytoriami Git Netlify:
- Darmowy plan z 125k funkcji serverless miesięcznie
- Świetny dla statycznych stron i aplikacji JAMstack
- Automatyczne wdrożenia z GitHuba Cloudflare Workers:
- Darmowy plan z 100,000 żądań dziennie
- Bardzo szybkie dzięki globalnej sieci edge
- Wspiera JavaScript i WebAssembly Google Cloud Functions:
- Darmowe 2 miliony wywołań miesięcznie
- Integracja z innymi usługami Google Cloud
- Wspiera wiele języków programowania AWS Lambda:
- 1 milion darmowych żądań miesięcznie
- 400,000 GB-sekund obliczeniowych miesięcznie
- Część darmowego planu AWS Free Tier (przez 12 miesięcy) Firebase Functions:
- Darmowy plan z ograniczonymi wywołaniami
- Łatwa integracja z innymi usługami Firebase

- Dobry wybór dla aplikacji mobilnych

Pamiętaj, że darmowe plany często mają ograniczenia, które mogą nie być odpowiednie dla aplikacji produkcyjnych o dużym ruchu. Jednak są doskonałe do nauki, testowania i małych projektów

WDROŻENIE APLIKACJI SERVERLESS W VERCEL

Na wstępie utwórz konto w portalu https://vercel.com/

1. Po skonfigurowaniu swojego konta przez portal, zainstaluj Vercel CLI globalnie:
   npm install -g vercel
2. Uruchom polecenie, aby zalogować się do swojego konta Vercel:
   vercel login
3. W katalogu projektu uruchom polecenie *vercel*
4. Postępuj zgodnie z instrukcjami, aby skonfigurować projekt. Vercel automatycznie wykryje, że jest to aplikacja Node.js i skonfiguruje odpowiednie środowisko
5. Konfiguracja pliku vercel.json

Aby dostosować konfigurację, możesz utworzyć plik vercel.json w katalogu głównym projektu:

```
{
 "version": 2,
 "builds": [
  {
   "src": "index.js",
   "use": "@vercel/node"
  }
 ],
 "routes": [
  {
   "src": "/(.*)",
   "dest": "index.js"
  }
 ] }
```

Ten plik informuje Vercel, jak budować i routować Twoją aplikację.

6. Wdrożenie i testowanie

Po skonfigurowaniu wszystkiego, ponownie uruchom polecenie ***vercel***, aby wdrożyć zmiany. Po zakończeniu procesu otrzymasz adres URL, pod którym będzie dostępna Twoja aplikacja.

Dodatkowe uwagi

- **Zmienna środowiskowa**: Jeśli potrzebujesz zmiennych środowiskowych, możesz je dodać w panelu sterowania Vercel.
- **Darmowy plan**: Upewnij się, że znasz limity darmowego planu Vercel, aby uniknąć niespodzianek w przyszłości.