Express.js

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

08.01.2026

Úvod

Webový framework

Co si představíte pod pojmem **webový framework**?

Znáte nějaké webové frameworky pro jiné jazyky?

Express.js

Express.js je minimalistický a flexibilní webový framework pro Node.js

- Umožňuje vytvářet **webové aplikace** a **API**
- Je **nejpopulárnějším** Node.js frameworkem
- Poskytuje **jednoduché rozhraní** pro práci s HTTP
- Je základem pro mnoho dalších frameworků (NestJS, …)

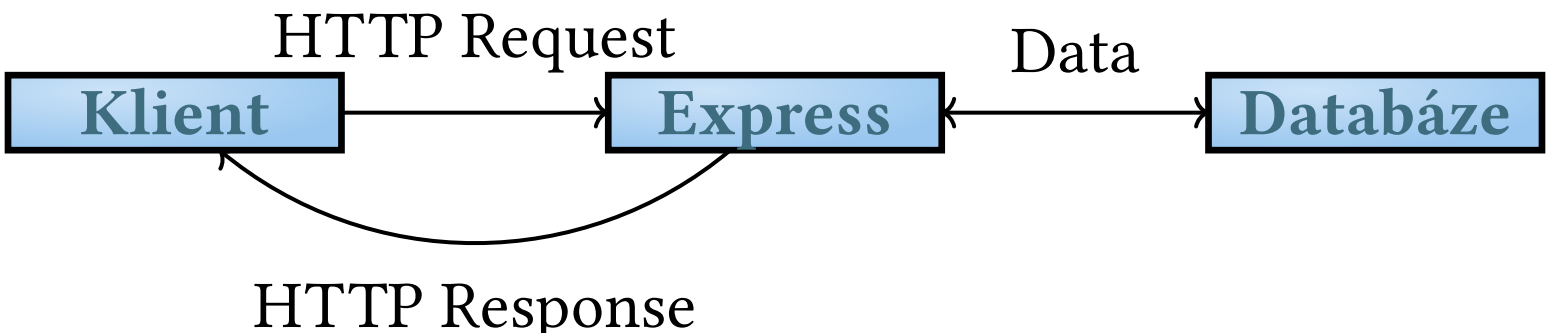
Express.js

Proč používat Express.js?

- **Minimalistický** - obsahuje pouze základní funkce
- **Flexibilní** - můžete si přidat co potřebujete
- **Rychlý** - nízká režie oproti větším frameworkům
- **Obrovská komunita** - spousta middleware a tutoriálů
- **Dobře zdokumentovaný** - kvalitní dokumentace

Express.js

Express.js architektura:



Instalace a první aplikace

Vytvoření nového Node.js projektu:

```
mkdir my-express-app
cd my-express-app
npm init -y
```

Instalace Express.js:

```
npm install express
```

Instalace a první aplikace

Hello World v Express.js:

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

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Instalace a první aplikace

Spuštění aplikace:

```
node index.js
```

Výstup:

```
Server running on http://localhost:3000
```

Nyní můžete otevřít prohlížeč na adrese `http://localhost:3000`

Instalace a první aplikace

Co se děje v kódu?

1. `require('express')` - importujeme Express
2. `express()` - vytvoříme aplikaci
3. `app.get('/', ...)` - definujeme route pro GET požadavky na `/`
4. `res.send(...)` - odešleme odpověď klientovi
5. `app.listen(3000, ...)` - spustíme server na portu 3000

Základy Express.js

Routing

Routing určuje, jak aplikace odpovídá na požadavky klienta na konkrétní endpoint

Každá route může mít jednu nebo více handler funkcí

Routing

Základní HTTP metody:

```
// GET - retrieve data
app.get('/users', (req, res) => {
  res.send('Get all users');
});

// POST - create data
app.post('/users', (req, res) => {
  console.log(req.body);
  res.send('Create user');
});
```

Routing

```
// PUT - update data
app.put('/users/:id', (req, res) => {
  console.log(req.body);
  res.send('Update user ' + req.params.id);
});

// DELETE - delete data
app.delete('/users/:id', (req, res) => {
  res.send('Delete user ' + req.params.id);
});
```

Routing

Route parametry umožňují zachytit hodnoty z URL:

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send('User ID: ' + userId);
});
```

Požadavek na `/users/42` vrátí `User ID: 42`

Routing

Query parametry se předávají za `?` v URL:

```
app.get('/search', (req, res) => {
  const query = req.query.q;
  const limit = req.query.limit || 10;
  res.send('Searching for: ' + query);
});
```

Požadavek na `/search?q=express&limit=5` vrátí

```
Searching for: express
```

Request a Response

Request objekt (`req`) obsahuje informace o příchozím požadavku:

- `req.params` - route parametry (`/users/:id`)
- `req.query` - query string (`?name=John`)
- `req.body` - tělo požadavku (pro POST/PUT)
- `req.headers` - HTTP hlavičky
- `req.method` - HTTP metoda (GET, POST, ...)
- `req.path` - cesta URL

Request a Response

Response objekt (`res`) umožňuje odeslat odpověď:

- `res.send()` - odešle text nebo HTML
- `res.json()` - odešle JSON data
- `res.status()` - nastaví HTTP status kód
- `res.redirect()` - přesměruje na jinou URL
- `res.sendFile()` - odešle soubor

Request a Response

Příklady odpovědí:

```
// Text response
res.send('Hello World');

// JSON response
res.json({ message: 'Hello', status: 'ok' });

// Status code with response
res.status(201).json({ id: 1, name: 'John' });
```

Request a Response

```
// Redirect
res.redirect('/login');

// Redirect with different status code
res.redirect(301, '/new-url');

// Send file
res.sendFile('/path/to/file.pdf');
```

Statické soubory

Express může servírovat **statické soubory** (HTML, CSS, JS, obrázky):

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

// Serve static files from 'public' folder
app.use(express.static('public'));

app.listen(3000);
```

Soubor `public/style.css` bude dostupný na `/style.css`

Statické soubory

Můžete přidat prefix pro statické soubory:

```
app.use('/static', express.static('public'));
```

Soubor `public/style.css` bude dostupný na `/static/style.css`

Middleware

Co je middleware?

Middleware jsou funkce, které mají přístup k request a response objektům

- Mohou **modifikovat** request a response
- Mohou **ukončit** request-response cyklus
- Mohou **předat řízení** další middleware funkci

Co je middleware?

Middleware funkce se volají postupně pomocí `next()`

Co je middleware?

Základní struktura middleware:

```
function myMiddleware(req, res, next) {
  // Do something with req or res
  console.log('Middleware executed');

  // Pass control to next middleware
  next();
}
```

Vestavěné middleware

Express poskytuje několik vestavěných middleware:

```
// Parse JSON bodies
app.use(express.json());

// Parse URL-encoded bodies (forms)
app.use(express.urlencoded({ extended: true }));

// Serve static files
app.use(express.static('public'));
```

Vestavěné middleware

Příklad s JSON body:

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

app.use(express.json());
app.post('/users', (req, res) => {
  console.log(req.body); // { name: 'John', age: 30 }
  res.json({ received: req.body });
});

app.listen(3000);
```

Vlastní middleware

Logging middleware - zaznamenává všechny požadavky:

```
function logger(req, res, next) {
  const timestamp = new Date().toISOString();
  console.log(`[${timestamp}] ${req.method} ${req.path}`);
  next();
}

app.use(logger);
```

Vlastní middleware

Autentizační middleware - kontroluje přihlášení:

```
function authenticate(req, res, next) {
  const token = req.headers.authorization;

  if (!token) {
    return res.status(401).json({ error: 'No token' });
  }

  // Verify token...
  next();
}
```

Vlastní middleware

Použití middleware pro konkrétní routes:

```
// Apply to all routes
app.use(logger);

// Apply to specific route
app.get('/admin', authenticate, (req, res) => {
  res.send('Admin panel');
});

// Apply to route group
app.use('/api', authenticate);
```

Error handling middleware

Error handling middleware má **4 parametry**:

```
function errorHandler(err, req, res, next) {
  console.error(err.stack);
  res.status(500).json({ error: 'Something went wrong!' });
}

// Must be last middleware
app.use(errorHandler);
```

Důležité: Error handler musí být **poslední** middleware!

Kvízová otázka

Jak express ví, že to, co mu předáváme do `app.use(...)` je middleware nebo error handler?

Kvízová otázka

Podle **arity** (počtu parametrů)

```
function middleware(req, res, next) { }

function errorHandler(err, req, res, next) { }

console.log(middleware.length);   // 3
console.log(errorHandler.length); // 4
```

Error Handling

Jak předat chybu do error handleru:

```
app.get('/users/:id', (req, res, next) => {
  const user = findUser(req.params.id);
  if (!user) {
    const error = new Error('User not found');
    error.status = 404;
    return next(error); // Pass error to error handler
  }
  res.json(user);
});
```

Error Handling

Vylepšený error handler:

```
function errorHandler(err, req, res, next) {
  const status = err.status || 500;
  const message = err.message || 'Internal Server Error';

  console.error(`[ERROR] ${status}: ${message}`);

  res.status(status).json({
    error: message, status: status
  });
}
```

Integrace TypeScriptu

Proč TypeScript s Express.js?

Dosud jsme používali **čistý JavaScript**

Proč bychom měli přejít na **TypeScript**?

- **Typová bezpečnost** - odhalení chyb při kompilaci
- **Lepší IntelliSense** - automatické doplňování
- **Dokumentace v kódu** - typy jako dokumentace
- **Refaktoring** - bezpečnější změny kódu
- **Škálovatelnost** - lepší pro větší projekty

Proč TypeScript s Express.js?

Od této chvíle budeme psát **všechny ukázky v TypeScriptu**!

Nastavení TypeScript projektu

Instalace potřebných balíčků:

```
npm install -D typescript @types/express @types/node
npm install -D ts-node nodemon
```

- `typescript` - TypeScript kompilátor
- `@types/express` - typové definice pro Express
- `@types/node` - typové definice pro Node.js
- `ts-node` - spouštění TS bez kompilace
- `nodemon` - automatický restart při změnách

Nastavení TypeScript projektu

Inicializace TypeScript:

```
npx tsc --init
```

Nastavení TypeScript projektu

Upravte `tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true
  }
}
```

Nastavení TypeScript projektu

Přidejte skripty do `package.json`:

```
{
  "scripts": {
    "dev": "nodemon --exec ts-node src/index.ts",
    "build": "tsc",
    "start": "node dist/index.js"
  }
}
```

Nastavení TypeScript projektu

Hello World v TypeScriptu:

```
import express, { Request, Response } from 'express';

const app = express();

app.get('/', (req: Request, res: Response) => {
  res.json({ message: 'Hello World!' });
});
app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Nastavení TypeScript projektu

Rozdíly oproti JavaScriptu:

- `import` místo `require`
- Explicitní typy `Request` a `Response`
- Soubory mají příponu `.ts`

Routing v TypeScriptu

Router moduly

Pro větší aplikace je vhodné rozdělit routes do **samostatných modulů**:

```
// src/routes/users.ts
import { Router, Request, Response } from 'express';

const router = Router();

router.get('/', (req: Request, res: Response) => {
  res.json({ users: [] });
});

export default router;
```

Router moduly

Použití routeru v hlavním souboru:

```
// src/index.ts
import express from 'express';
import userRoutes from './routes/users';

const app = express();

app.use('/api/users', userRoutes);
app.listen(3000);
```

Všechny routes z `users.ts` budou mít prefix `/api/users`

Typované parametry

TypeScript umožňuje typovat **route parametry**:

```
interface UserParams {
  id: string;
}
router.get('/:id', (
  req: Request<UserParams>, res: Response
) => {
  const userId = req.params.id; // string !!!
  res.json({ id: userId });
});
```

Typované parametry

Typování **query parametrů**:

```
interface SearchQuery {
  q?: string;
  limit?: string;
}
router.get('/search', (
  req: Request<{}, {}, {}, SearchQuery>, res: Response
) => {
  const { q, limit = '10' } = req.query;
  res.json({ query: q, limit });
});
```

Typované parametry

Generické typy pro Request:

```
Request<Params, ResBody, ReqBody, Query>
```

- `Params` - route parametry (`:id`)
- `ResBody` - typ response body
- `ReqBody` - typ request body
- `Query` - query parametry (`?key=value`)

Runtime validace typů

Důležité: TypeScript typy existují pouze při kompilaci!

Za běhu jsou **všechny** query a route parametry typu `string`

```
interface SearchQuery {
  limit?: number;  // TypeScript says number
}

// But at runtime it's always a string!
// And what if user sends ?limit=abc ?
```

Runtime validace typů

Problém s přetypováním:

```
router.get('/users', (req: Request, res: Response) => {
  const limit = req.query.limit; // string | undefined

  // Dangerous! parseInt('abc') === NaN
  const limitNum = parseInt(limit as string);

  // NaN will cause problems later in code
  const users = await db.users.limit(limitNum);
});
```

Runtime validace typů

Bezpečná konverze s validací:

```
function parseIntSafe(
  value: string | undefined,
  defaultValue: number
): number {
  if (!value) return defaultValue;
  const parsed = parseInt(value, 10);
  return isNaN(parsed) ? defaultValue : parsed;
}
```

Runtime validace typů

```
router.get('/users', (req: Request, res: Response) => {
  const limit = parseIntSafe(req.query.limit as string, 10);
  const offset = parseIntSafe(req.query.offset as string, 0);

  // limit and offset are always valid numbers
  const users = await db.users.limit(limit).offset(offset);
  res.json(users);
});
```

Runtime validace typů

Validace route parametrů:

```
router.get('/:id', (req: Request, res: Response) => {
  const id = parseInt(req.params.id, 10);
  if (isNaN(id) || id <= 0) {
    return res.status(400).json({
      error: 'Invalid ID format'
    });
  }
  const user = await userService.findById(id);
  res.json(user);
});
```

Runtime validace typů

Validace boolean query parametrů:

```
function parseBoolSafe(
  value: string | undefined,
  defaultValue: boolean
): boolean {
  if (!value) return defaultValue;
  return value.toLowerCase() === 'true';
}
```

Runtime validace typů

```
// ?active=true or ?active=false
const active = parseBoolSafe(req.query.active as string,
true);
```

Runtime validace typů

Komplexní validace pomocí middleware:

```
interface QuerySchema {
  limit?: { type: 'number'; default: number; min?: number };
  offset?: { type: 'number'; default: number };
  sort?: { type: 'string'; allowed: string[] };
}
```

Runtime validace typů

```
function validateQuery(schema: QuerySchema) {
  return (
    req: Request, res: Response, next: NextFunction
  ) => {
    const errors: string[] = [];

    for (const [key, config] of Object.entries(schema)) {
      const value = req.query[key] as string | undefined;
      // ... validate according to schema
    }
```

Runtime validace typů

```
    if (errors.length > 0) {
      return res.status(400).json({ errors });
    }
    next();
  };
}
```

Runtime validace typů

Pro komplexnější validaci doporučuji knihovnu **Zod**:

```
npm install zod
```

- Runtime validace a parsování
- TypeScript inference z runtime schémat
- Detailní chybové zprávy
- Podpora pro komplexní datové struktury

Runtime validace typů

Příklad s Zod:

```
import { z } from 'zod';

const searchQuerySchema = z.object({
  q: z.string().min(1),
  limit: z.coerce.number().int().min(1).max(100).default(10),
  offset: z.coerce.number().int().min(0).default(0),
});
```

Runtime validace typů

```
router.get('/search', (req: Request, res: Response) => {
  const result = searchQuerySchema.safeParse(req.query);

  if (!result.success) {
    return res.status(400).json({
      errors: result.error.errors
    });
  }
```

Runtime validace typů

```
  // result.data has correct types!
  const { q, limit, offset } = result.data;
  // q: string, limit: number, offset: number
});
```

Runtime validace typů

Zod middleware pro validaci:

```
function validateQueryZod<T extends z.ZodTypeAny>(schema: T)
{
  return (
    req: Request, res: Response, next: NextFunction
  ) => {
    const result = schema.safeParse(req.query);
```

Runtime validace typů

```
    if (!result.success) {
      return res.status(400).json({
        error: 'Validation failed',
        details: result.error.errors
      });
    }

    req.query = result.data;
    next();
  };
}
```

Runtime validace typů

Shrnutí runtime validace:

- TypeScript typy **neexistují za běhu**
- Query a route parametry jsou **vždy stringy**
- Vždy **validujte a konvertujte** vstupní data
- Používejte **helper funkce** nebo **knihovny** (Zod, Joi)
- Vracejte **srozumitelné chybové zprávy**

RESTful API design

RESTful API používá HTTP metody pro CRUD operace:

Metoda	Route	Akce
GET	`/users`	Získat všechny
GET	`/users/:id`	Získat jednoho
POST	`/users`	Vytvořit
PUT	`/users/:id`	Aktualizovat
DELETE	`/users/:id`	Smazat

RESTful API design

Kompletní CRUD router:

```
import { Router, Request, Response } from 'express';
const router = Router();
interface User {
  id: number;
  name: string;
  email: string;
}
let users: User[] = [];
let nextId = 1;
```

RESTful API design

```
// GET all users
router.get('/', (req: Request, res: Response) => {
  res.json(users);
});

// GET user by ID
router.get('/:id', (req: Request, res: Response) => {
  const user = users.find(
    u => u.id === parseInt(req.params.id)
  );
```

RESTful API design

```
  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }
  res.json(user);
});
```

RESTful API design

```
// POST create user
router.post('/', (req: Request, res: Response) => {
  const { name, email } = req.body;
  const user: User = { id: nextId++, name, email };
  users.push(user);
  res.status(201).json(user);
});
```

RESTful API design

```
// PUT update user
router.put('/:id', (req: Request, res: Response) => {
  const id = parseInt(req.params.id);
  const index = users.findIndex(u => u.id === id);
  if (index === -1) {
    return res.status(404).json({ error: 'User not found' });
  }
  users[index] = { ...users[index], ...req.body };
  res.json(users[index]);
});
```

RESTful API design

```
// DELETE user
router.delete('/:id', (req: Request, res: Response) => {
  const id = parseInt(req.params.id);
  const index = users.findIndex(u => u.id === id);
  if (index === -1) {
    return res.status(404).json({ error: 'User not found' });
  }
  const deleted = users.splice(index, 1)[0];
  res.json(deleted);
});
export default router;
```

Request a Response Handling

Typované Request Body

Definování typu pro request body:

```
interface CreateUserBody {
  name: string;
  email: string;
  age?: number;
}
router.post('/', (
  req: Request<{}, {}, CreateUserBody>, res: Response
) => {
  const { name, email, age } = req.body;
});
```

Typované Request Body

Vlastní typy pro Request a Response:

```
interface UserParams {
  id: string;
}
interface UserResponse {
  id: number;
  name: string; email: string;
}
interface CreateUserBody {
  name: string; email: string;
}
```

Typované Request Body

```
router.post('/', (
  req: Request<{}, UserResponse, CreateUserBody>,
  res: Response<UserResponse>
) => {
  const { name, email } = req.body;
  const user: UserResponse = {
    id: nextId++,
    name, email
  };
  res.status(201).json(user);
});
```

Validace dat

Jednoduchá validace v TypeScriptu:

```
interface CreateUserBody {
  name: string; email: string;
}
function validateUser(body: any): body is CreateUserBody {
  return (
    typeof body.name === 'string' &&
    typeof body.email === 'string' &&
    body.email.includes('@')
  );
}
```

Validace dat

Použití validace:

```
router.post('/', (req: Request, res: Response) => {
  if (!validateUser(req.body)) {
    return res.status(400).json({
      error: 'Invalid user data'
    });
  }
  // TypeScript now knows req.body is CreateUserBody
  const { name, email } = req.body;
  // ...
});
```

Validace dat

```
function validateBody<T>(validator: (body: any) => body is T)
{
  return (req: Request, res: Response, next: Function) => {
    if (!validator(req.body)) {
      return res.status(400).json({
        error: 'Validation failed'
      });
    }
    next();
  };
}
```

Validace dat

```
router.post(
  '/',
  validateBody(validateUser),
  (req: Request, res: Response) => {
    // req.body is validated
    const user = createUser(req.body);
    res.status(201).json(user);
  }
);
```

Response metody

Různé způsoby odpovědi:

```
// JSON response
res.json({ message: 'Success' });
// Status + JSON
res.status(201).json({ id: 1 });
// Error response
res.status(404).json({ error: 'Not found' });
// Empty response
res.status(204).send();
```

Response metody

Typovaná response:

```
interface ApiResponse<T> {
  success: boolean;
  data?: T;
  error?: string;
}
```

Response metody

```
function sendSuccess<T>(res: Response, data: T, status = 200)
{
  const response: ApiResponse<T> = {
    success: true,
    data
  };
  res.status(status).json(response);
}
```

Response metody

```
function sendError(
  res: Response, error: string, status = 400
) {
  const response: ApiResponse<null> = {
    success: false,
    error
  };
  res.status(status).json(response);
}
```

Response metody

Použití helper funkcí:

```
router.get('/:id', (req: Request, res: Response) => {
  const user = users.find(
    u => u.id === parseInt(req.params.id)
  );
  if (!user) {
    return sendError(res, 'User not found', 404);
  }
  sendSuccess(res, user);
});
```

Error Handling v TypeScriptu

Custom Error třídy

Vytvoření vlastní třídy pro HTTP chyby:

```
class HttpError extends Error {
  constructor(
    public statusCode: number,
    message: string
  ) {
    super(message);
    this.name = 'HttpError';
  }
}
```

Custom Error třídy

Specializované error třídy:

```
class NotFoundError extends HttpError {
  constructor(resource: string) {
    super(404, `${resource} not found`);
    this.name = 'NotFoundError';
  }
}
```

Custom Error třídy

```
class ValidationError extends HttpError {
  constructor(message: string) {
    super(400, message);
    this.name = 'ValidationError';
  }
}
```

Custom Error třídy

```
class UnauthorizedError extends HttpError {
  constructor(message = 'Unauthorized') {
    super(401, message);
    this.name = 'UnauthorizedError';
  }
}
```

Custom Error třídy

```
class ForbiddenError extends HttpError {
  constructor(message = 'Forbidden') {
    super(403, message);
    this.name = 'ForbiddenError';
  }
}
```

Async Error Handling v Express 4

Problém s async funkcemi:

```
// This won't catch errors properly!
router.get('/:id', async (req: Request, res: Response) => {
  const user = await findUserById(req.params.id); // throw?
  res.json(user);
});
```

Express nezachytí chyby z async funkcí automaticky!

Async Error Handling v Express 4

Řešení - wrapper funkce:

```
type AsyncHandler = (
  req: Request,
  res: Response,
  next: NextFunction
) => Promise<any>;
```

Async Error Handling v Express 4

```
function asyncHandler(fn: AsyncHandler) {
  return (
    req: Request, res: Response, next: NextFunction
  ) => {
    Promise.resolve(fn(req, res, next)).catch(next);
  };
}
```

Async Error Handling v Express 4

Použití async handleru:

```
router.get('/:id', asyncHandler(async (req, res) => {
  const user = await findUserById(req.params.id);
  if (!user) {
    throw new NotFoundError('User');
  }
  res.json(user);
}));
```

Nyní se chyby správně předají error handleru!

Async Error Handling v Express 5

Express 5 automaticky zachytává chyby z async funkcí:

```
// Express 5 - works without wrapper!
router.get('/:id', async (req: Request, res: Response) => {
  const user = await getUserById(req.params.id);
  res.json(user);
});
```

Pokud `getUserById` vyhodí chybu nebo vrátí rejected Promise, Express 5 automaticky zavolá `next(error)`

Poznámka: `asyncHandler` je stále užitečný pro Express 4 projekty

Centralizovaný Error Handler

```
import { Request, Response, NextFunction } from 'express';

function errorHandler(
  err: Error,
  req: Request,
  res: Response,
  next: NextFunction
) {
```

Centralizovaný Error Handler

```
console.error(`[ERROR] ${err.name}: ${err.message}`);
if (err instanceof HttpError) {
  return res.status(err.statusCode).json({
    success: false,
    error: err.message
  });
}
```

Centralizovaný Error Handler

```
  // Unknown error
  res.status(500).json({
    success: false,
    error: 'Internal Server Error'
  });
}

export default errorHandler;
```

Centralizovaný Error Handler

Použití v aplikaci:

```
import express from 'express';
import userRoutes from './routes/users';
import errorHandler from './middleware/errorHandler';

const app = express();

app.use(express.json());
app.use('/api/users', userRoutes);
```

Centralizovaný Error Handler

```
// Error handler must be last!
app.use(errorHandler);

app.listen(3000);
```

Organizace projektu

Struktura složek

Doporučená struktura pro Express.js + TypeScript projekt:

```
src/
├── index.ts          # Entry point
├── app.ts            # Express app setup
├── controllers/      # Request handlers
├── routes/           # Route definitions
├── middleware/       # Custom middleware
├── models/           # Data models/types
├── services/         # Business logic
└── utils/            # Helper functions
```

Controllers

Controller obsahuje logiku pro zpracování požadavků:

```
// src/controllers/userController.ts
import { Request, Response } from 'express';
import { UserService } from '../services/userService';

export class UserController {
  constructor(private readonly userService: UserService) {}
```

Controllers

```
async getAll(req: Request, res: Response) {
  const users = await this.userService.findAll();
  res.json(users);
}
```

Controllers

```
  async getById(req: Request, res: Response) {
    const userId = req.params.id;
    const user = await this.userService.findById(userId);
    if (!user) {
      return res.status(404).json({ error: 'Not found' });
    }
    res.json(user);
  }
```

Controllers

```
  async create(req: Request, res: Response) {
    const user = await this.userService.create(req.body);
    res.status(201).json(user);
  }
```

Controllers

```
async update(req: Request, res: Response) {
  const userId = req.params.id;
  const user = await this.userService.update(
    userId,
    req.body
  );
  res.json(user);
}
```

Controllers

```
  async delete(req: Request, res: Response) {
    const userId = req.params.id;
    await this.userService.delete(userId);
    res.status(204).send();
  }
}
```

Services

Service obsahuje business logiku:

```
// src/services/userService.ts
import { User, CreateUserDto } from '../models/user';

export class UserService {
  private users: User[] = [];
  private nextId = 1;
```

Services

```
  async findAll(): Promise<User[]> {
    return this.users;
  }
  async findById(id: string): Promise<User | undefined> {
    return this.users.find(u => u.id === parseInt(id));
  }
  async create(data: CreateUserDto): Promise<User> {
    const user: User = { id: this.nextId++, ...data };
    this.users.push(user);
    return user;
  }
```

Services

```
  async update(
    id: string, data: Partial<User>
  ): Promise<User> {
    const index = this.users.findIndex(
      u => u.id === parseInt(id)
    );
    if (index === -1) { throw new Error('User not found'); }
    this.users[index] = { ...this.users[index], ...data };
    return this.users[index];
  }
```

Services

```
  async delete(id: string): Promise<void> {
    const index = this.users.findIndex(
      u => u.id === parseInt(id)
    );
    if (index === -1) { throw new Error('User not found'); }
    this.users.splice(index, 1);
  }
}
```

Routes

Routes propojují URL s controllery:

```
// src/routes/userRoutes.ts
import { Router } from 'express';
import { UserController } from '../controllers/
userController';
import { UserService } from '../services/userService';
import { asyncHandler } from '../utils/asyncHandler';
```

Routes

```
const router = Router();
const userService = new UserService();
const userController = new UserController(userService);
```

Routes

```
router.get('/', asyncHandler(
  (req, res) => userController.getAll(req, res)
));

router.get('/:id', asyncHandler(
  (req, res) => userController.getById(req, res)
));

router.post('/', asyncHandler(
  (req, res) => userController.create(req, res)
));
```

Routes

```
router.put('/:id', asyncHandler(
  (req, res) => userController.update(req, res)
));

router.delete('/:id', asyncHandler(
  (req, res) => userController.delete(req, res)
));

export default router;
```

App Setup

Oddělení Express aplikace od serveru:

```
// src/app.ts
import express from 'express';
import userRoutes from './routes/userRoutes';
import errorHandler from './middleware/errorHandler';

const app = express();
```

App Setup

```
// Middleware
app.use(express.json());

// Routes
app.use('/api/users', userRoutes);

// Error handler (must be last)
app.use(errorHandler);

export default app;
```

App Setup

Entry point:

```
// src/index.ts
import app from './app';

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Oddělení `app` od `index` usnadňuje testování!

Praktické příklady

Logging Middleware

```
// src/middleware/logger.ts
import { Request, Response, NextFunction } from 'express';

export function logger(
  req: Request,
  res: Response,
  next: NextFunction
) {
  const start = Date.now();
```

Logging Middleware

```
  res.on('finish', () => {
    const duration = Date.now() - start;
    console.log(
      `[${new Date().toISOString()}] ` +
      `${req.method} ${req.path} ` +
      `${res.statusCode} ${duration}ms`
    );
  });

  next();
}
```

Autentizační Middleware

```
// src/middleware/auth.ts
import { Request, Response, NextFunction } from 'express';

interface AuthRequest extends Request {
  userId?: string;
}
```

Autentizační Middleware

```
export function authenticate(
  req: AuthRequest, res: Response, next: NextFunction
) {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) {
    return res.status(401).json({
      error: 'No token provided'
    });
  }
```

Autentizační Middleware

```
  try {
    // Verify token (simplified example)
    const decoded = verifyToken(token);
    req.userId = decoded.userId;
    next();
  } catch (error) {
    res.status(401).json({
      error: 'Invalid token'
    });
  }
}
```

Autentizační Middleware

Použití autentizace:

```
// Protect all routes in this router
router.use(authenticate);

// Or protect specific routes
router.get('/profile', authenticate, (req: AuthRequest, res)
=> {
  res.json({ userId: req.userId });
});
```

Rate Limiting Middleware

Jednoduchý rate limiter:

```
// src/middleware/rateLimiter.ts
interface RateLimitStore {
  [ip: string]: {
    count: number;
    resetTime: number;
  };
}

const store: RateLimitStore = {};
```

Rate Limiting Middleware

```
export function rateLimiter(
  maxRequests: number,
  windowMs: number
) {
  return (
    req: Request, res: Response, next: NextFunction
  ) => {
    const ip = req.ip || 'unknown';
    const now = Date.now();
```

Rate Limiting Middleware

```
    if (!store[ip] || store[ip].resetTime < now) {
      store[ip] = {
        count: 1,
        resetTime: now + windowMs
      };
      return next();
    }

    store[ip].count++;
```

Rate Limiting Middleware

```
    if (store[ip].count > maxRequests) {
      return res.status(429).json({
        error: 'Too many requests'
      });
    }

    next();
  };
}
```

Rate Limiting Middleware

Použití:

```
// 100 requests per 15 minutes
app.use(rateLimiter(100, 15 * 60 * 1000));

// Or for specific routes
app.use('/api/login', rateLimiter(5, 60 * 1000));
```

Best Practices

Best Practices

Doporučené postupy:

- Používejte **TypeScript** pro větší projekty
- Oddělujte **business logiku** od **handlerů**
- Používejte **centralizovaný error handling**
- Validujte **vstupní data**
- Logujte **požadavky a chyby**

Best Practices

- Používejte **environment variables** pro konfiguraci
- Oddělte **app** od **serveru** pro testování
- Používejte **async/await** s proper error handling
- Dokumentujte API (např. pomocí Swagger/OpenAPI)
- Testujte routes a middleware

Užitečné zdroje

Oficiální dokumentace

- Express.js: https://expressjs.com/
- TypeScript: https://www.typescriptlang.org/

Další frameworky

- NestJS - full-featured TypeScript framework: https://nestjs.com/
- Fastify - rychlá alternativa k Express: https://www.fastify.io/

Užitečné zdroje

Middleware a nástroje

- `cors` - Cross-Origin Resource Sharing
- `helmet` - Security headers
- `morgan` - HTTP request logger
- `express-validator` - Validace dat
- `passport` - Autentizace