NodeJS tutorial

Dr. Balázs Simon (sbalazs@iit.bme.hu), BME IIT, 2023

1 Introduction

This document describes how to create a Hello World application built in NodeJS using MongoDB. Make sure you have installed Visual Studio Code, NPM and MongoDB according to the installation guide.

2 Preparing the NodeJS environment

At first we need some global NodeJS packages so that we can generate and run our project.

Issue the following commands from a command prompt:

```
npm install --global typescript
```

This will install TypeScript. TypeScript is an open-source, strongly typed superset language of JavaScript. TypeScript is created by Microsoft to make JavaScript development scalable for large production applications. TypeScript code is translated to JavaScript by the TypeScript compiler.

3 Creating the NodeJS project

Open a command prompt and go into a folder in your file system that should contain the project folder of your node project (e.g. **c:\Users\[user]\Documents**). **Important:** the complete path of the folder **must not** contain spaces or any special characters. In case your user name contains spaces or special characters, create the project folder outside the **c:\Users** folder.

Create a new folder called node_hello, and enter this folder:

```
c:\Users\[user]\Documents>md node_hello
c:\Users\[user]\Documents>cd node_hello
```

Initialize the NodeJS project:

```
c:\Users\[user]\Documents\node_hello>npm init --yes
```

This command will create default a **package.json** file which contains the dependencies of our project.

Now we can install all our required dependencies:

```
c:\Users\[user]\Documents\node_hello>npm install --save express mongoose
```

The modules should install without errors. The following modules have been installed:

- **express**: allows implementing REST operations

- **mongoose**: the driver to connect to MongoDB

Also, we can install our development-only dependencies:

```
c:\Users\[user]\Documents\node_hello>npm install --save-dev typescript @types/express ⤵
    @types/node
```

The modules should install without errors. The following modules have been installed:

- **typescript**: TypeScript language support
- **@types/express**: TypeScript type descriptors for express
- **@types/node**: TypeScript type descriptors for NodeJS

Now we can initialize TypeScript development:

```
c:\Users\[user]\Documents\node_hello>tsc --init
```

This command will create default a **tsconfig.json** file which contains the configuration of the TypeScript compiler.

Create two folders, one for the TypeScript files (**src**) and one for the generated JavaScript files (**app**):

```
c:\Users\[user]\Documents\node_hello>md src
c:\Users\[user]\Documents\node_hello>md app
```

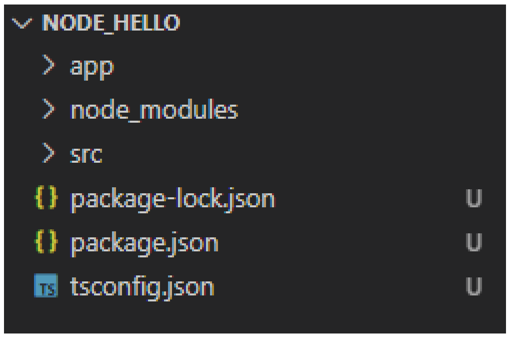
Open the **tsconfig.json** file in a text editor and modify the following highlighted lines to configure these directories:

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig to read more about this file */

    /* Projects */

    /* Language and Environment */
    "target": "es2016", /* Set the JavaScript language version for emitted JavaScript
                           and include compatible library declarations. */

    /* Modules */
    "module": "commonjs", /* Specify what module code is generated. */
    "rootDir": "./src", /* Specify the root folder within your source files. */

    /* JavaScript Support */

    /* Emit */
    "outDir": "./app", /* Specify an output folder for all emitted files. */
    "sourceMap": true, /* Create source map files for emitted JavaScript files. */
    /* Interop Constraints */
    "esModuleInterop": true, /* Emit additional JavaScript to ease support for
                                importing CommonJS modules. This enables
                                'allowSyntheticDefaultImports' for type
                                compatibility. */
    "forceConsistentCasingInFileNames": true, /* Ensure that casing is correct in
                                                 imports. */

    /* Type Checking */
    "strict": true, /* Enable all strict type-checking options. */

    /* Completeness */
    "skipLibCheck": true /* Skip type checking all .d.ts files. */
  }
}
```

Now the project is ready for development.

4 Opening the project in VSCode

Start **Visual Studio Code** and select **File > Open Folder...** from the menu, and open the project folder you have just created. VSCode will open the node project. The project has the following structure:



Create a new file called **app.ts** under the **src** folder (right click on the src folder and select New File…). The content of this file should be:

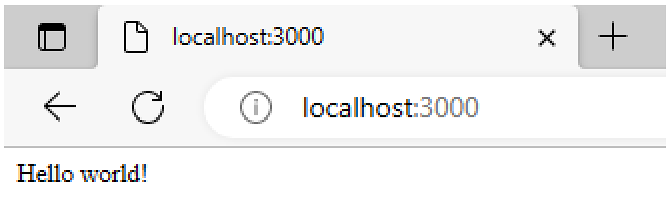
```
import express, { Application, Request, Response } from 'express';

const app: Application = express();
const PORT: number = 3000;

app.use('/', (req: Request, res: Response): void => {
    res.send('Hello world!');
});

app.listen(PORT, (): void => {
    console.log('Listening on:', PORT);
});
```

We need the TypeScript compiler to continuously run in the background and automatically compile TypeScript files to JavaScript. To start the TypeScript compiler, press **Ctrl+Shift+B**. Select the **tsc: watch** task. This will immediately compile the existing **.ts** files in the project and will continue to do so whenever you change a **.ts** file. The generated JavaScript code will appear in the **app** folder. VSCode will also report any errors found by the TypeScript compiler.

Modify the **package.json** to specify the generated **app/app.js** JavaScript file as our main file:

```
{
  "name": "node_hello",
  "version": "1.0.0",
  "description": "",
  "main": "app/app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node app/app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2",
    "mongoose": "^7.1.1"
  },
  "devDependencies": {
    "@types/express": "^4.17.17",
    "@types/node": "^20.1.5",
    "typescript": "^5.0.4"
  }
}
```

Now we can start our application. Go back to the **app.ts** file and select **Debug / Start Debugging** from the menu or pressing **F5**. Select **Node.js** as the debugger. The server will start on port 3000. You can test it in the browser:



Hello world!

You can stop the server in VSCode by pressing **Shift+F5**.

You can set breakpoints in the TypeScript file by selecting **Debug / Toggle breakpoint** from the menu or pressing **F9**. The execution will stop at that line when the server is started in debug mode (by pressing **F5**). The keyboard shortcuts for debugging are similar to the ones in Visual Studio.

5 Publishing a new route

Create a new folder called **services** under the **src** folder. Create a file called **hello_service.ts** inside the **src/services** folder with the following content:

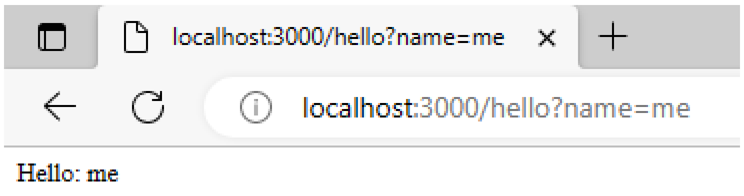
```
// Import express:
import express, {Request, Response} from 'express';

// Create a new express router:
const router = express.Router();

interface RequestParams {}

interface ResponseBody {}

interface RequestBody {}

// Specify that 'name' is a query parameter of type 'string':
interface RequestQuery {
  name: string;
}

/* GET request: */
router.get('/', (req: Request<RequestParams, ResponseBody, RequestBody,
RequestQuery>, res: Response) => {
    // Get the 'name' query param:
    let name: String = req.query.name;
    // Send response:
    res.send("Hello: "+name);
});

// Export the router:
export default router;
```

Next, we need to wire this router into the main application's router. In order to do this, open the **src/app.ts** and add the following highlighted lines:

```
import express, { Application, Request, Response } from 'express';
import bodyParser from 'body-parser';
import helloService from './services/hello_service';
```

```
const app: Application = express();
const PORT: number = 3000;

app.use(bodyParser.json());

app.use("/hello", helloService);

app.use('/', (req: Request, res: Response): void => {
    res.send('Hello world!');
});

app.listen(PORT, (): void => {
    console.log('Listening on:', PORT);
});
```

Now every request to a URL starting with **/hello** will go to our **hello_service**. Restart the application and test it in a browser:



```
Hello: me
```

6 Accessing MongoDB

We need an ORM mapping from TypeScript to MongoDB to be able to access objects stored in the database conveniently.

At first, we have to define the interfaces of the objects. Create a new folder called **interfaces** under the **src** folder. Create a file called **hello.ts** inside the **src/interfaces** folder with the following content:

```
export interface IHello
{
    name: string;
    message: string;
}
```

Next, we have to define the Mongoose mapping of this interface to MongoDB. Create a new folder called **schemas** under the **src** folder. Create a file called **hello.ts** inside the **src/schemas** folder with the following content (make sure to use the String type with capital ‘S’ here):

```
import mongoose from 'mongoose';
import { IHello } from '../interfaces/hello';

export interface HelloEntity extends IHello, mongoose.Document { }

let HelloSchema = new mongoose.Schema({
    name: String,
    message: String
});

export var Hello = mongoose.model<HelloEntity>('Hello', HelloSchema, 'Hello');
```

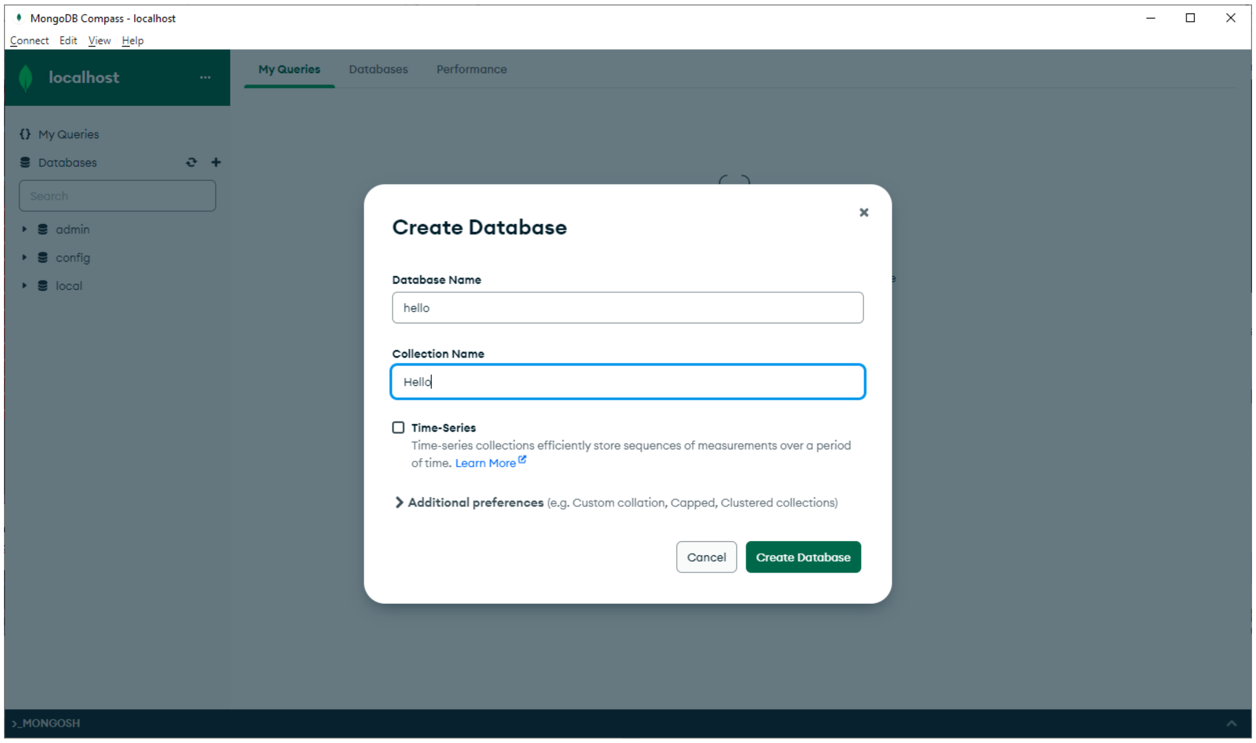
Now we need to call this mapping from the service. Update the **src/services/hello_service.ts** with the highlighted lines:

```
// Import express:
import express, {NextFunction, Request, Response} from 'express';
import { Hello } from '../schemas/hello';
var createError = require('http-errors');

// Create a new express router:
const router = express.Router();

interface RequestParams {}

interface ResponseBody {}

interface RequestBody {}

// Specify that 'name' is a query parameter of type 'string':
interface RequestQuery {
  name: string;
}

/* GET request: */
router.get('/', async (req: Request<RequestParams, ResponseBody, RequestBody,
RequestQuery>, res: Response, next: NextFunction) => {
    // Get the 'name' query param:
    let name: String = req.query.name;
    let query = { name: name };
    let hello = await Hello.findOne(query);
    if (hello) {
        // Send response:
        res.send(hello.message);
    } else {
        return next(createError.NotFound())
    }
});

// Export the router:
export default router;
```

Finally, we have to connect to MongoDB in the application. Modify the following highlighted lines in the **src/app.ts** file:

```
import express, { Application, Request, Response } from 'express';
import bodyParser from 'body-parser';
import helloService from './services/hello_service';
import mongoose from 'mongoose';

const app: Application = express();
const PORT: number = 3000;

app.use(bodyParser.json());

app.use("/hello", helloService);

app.use('/', (req: Request, res: Response): void => {
    res.send('Hello world!');
});

mongoose.connect('mongodb://127.0.0.1:27017/hello')
    .then(res => {
        console.log("Connected to MongoDB");
        app.listen(PORT, (): void => {
            console.log('Listening on:', PORT);
        });
    })
    .catch(err => console.log(err));
```

Let’s add some data into the database. Open the MongoDB Compass client and create a new database called **hello** with a collection name **Hello**:



Database Name: hello

Collection Name: Hello

Time-Series
Time-series collections efficiently store sequences of measurements over a period of time. Learn More

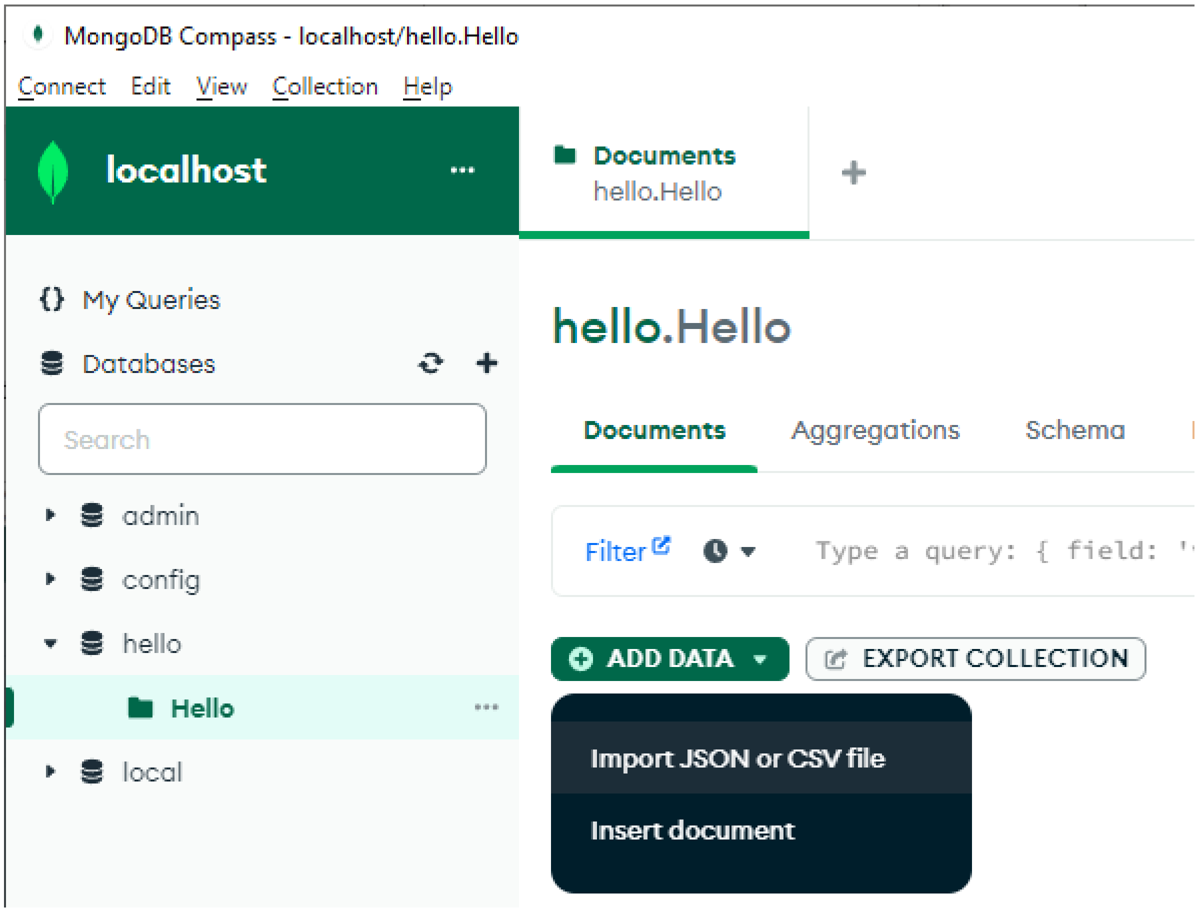
Additional preferences (e.g. Custom collation, Capped, Clustered collections)

Click **Create Database** to create the database and the collection.

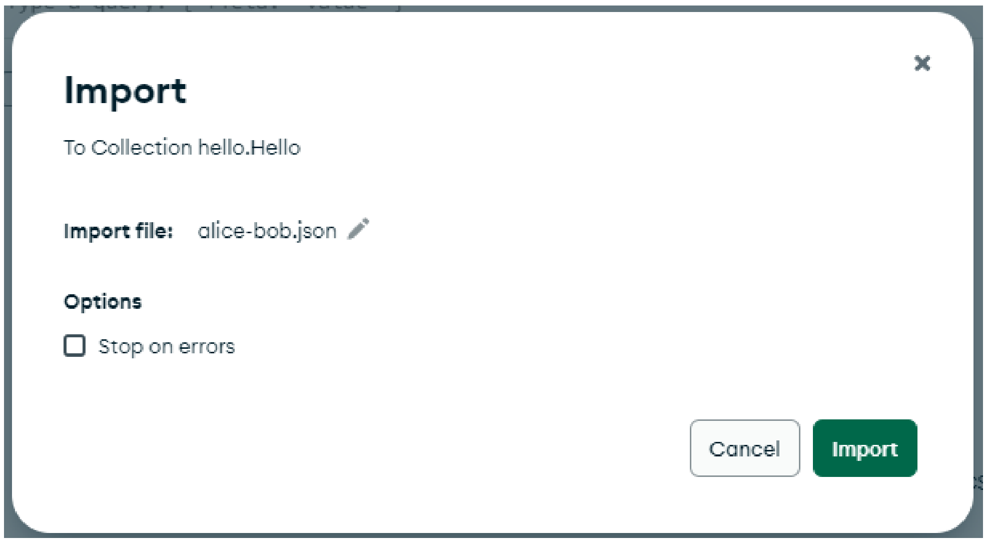
Create a JSON file somewhere in your file system with the following content:

```
{"name": "Alice", "message": "Good morning, Alice!"}
{"name": "Bob", "message": "Good evening, Bob!"}
```

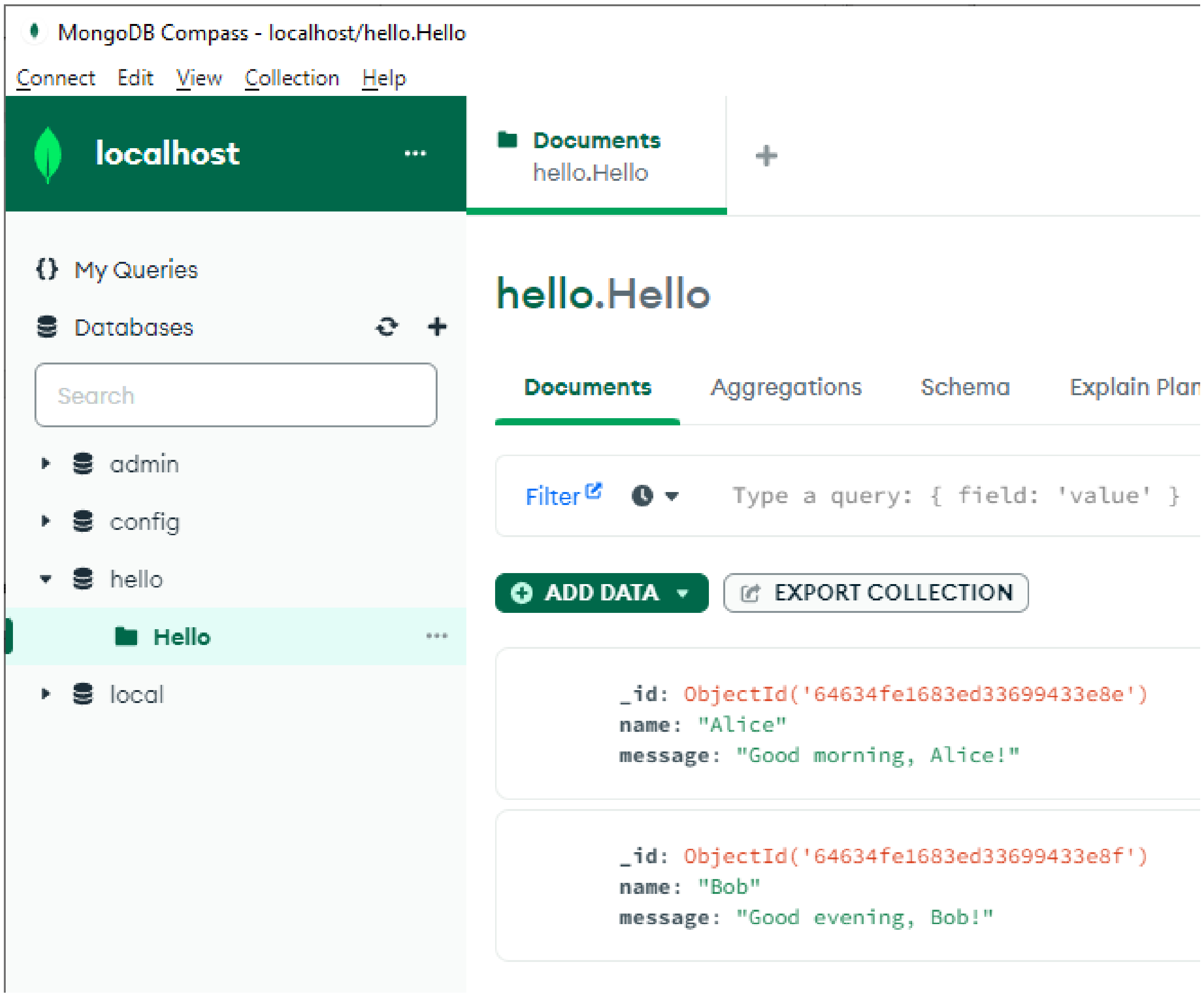
Select the **Hello** collection inside the **hello** database and click on **Add data > Import JSON or CSV file**:



Then select the previously created file to be imported:



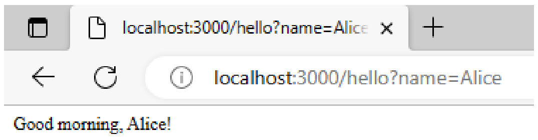
Click **Import**. This will insert two records into the database:


MongoDB Compass - localhost/hello.Hello
Connect Edit View Collection Help
localhost
Documents hello.Hello
My Queries
Databases
Search
admin
config
hello
Hello
local
hello.Hello
Documents Aggregations Schema Explain Plan
Filter
Type a query: { field: 'value' }
ADD DATA
EXPORT COLLECTION

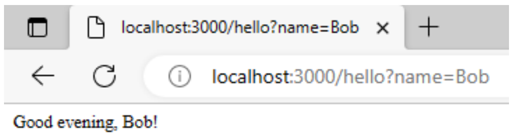
```
_id: ObjectId('64634fe1683ed33699433e8e')
name: "Alice"
message: "Good morning, Alice!"
```

```
_id: ObjectId('64634fe1683ed33699433e8f')
name: "Bob"
message: "Good evening, Bob!"
```

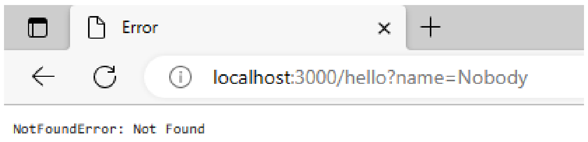
Now we can test our application in a browser:


localhost:3000/hello?name=Alice
localhost:3000/hello?name=Alice

Good morning, Alice!



Good evening, Bob!



```
NotFoundError: Not Found
```