MongoDB: Setup and Insert

SENG 4640
Software Engineering for Web Apps
Winter 2023

Sina Keshvadi
Thompson Rivers University

Using Node in Express, we can start to build web applications on the server side.

Using Node in Express, we can start to build web applications on the server side.

Then we can use a tool like EJS to separate the form from the functionality, separate the HTML from the JavaScript.

Using Node in Express, we can start to build web applications on the server side.

Then we can use a tool like EJS to separate the form from the functionality, separate the HTML from the JavaScript.

But when it comes to building a big web application that's going to serve dynamic content to the users, there's one big piece that's missing.

Using Node in Express, we can start to build web applications on the server side.

Then we can use a tool like EJS to separate the form from the functionality, separate the HTML from the JavaScript.

But when it comes to building a big web application that's going to serve dynamic content to the users, there's one big piece that's missing.

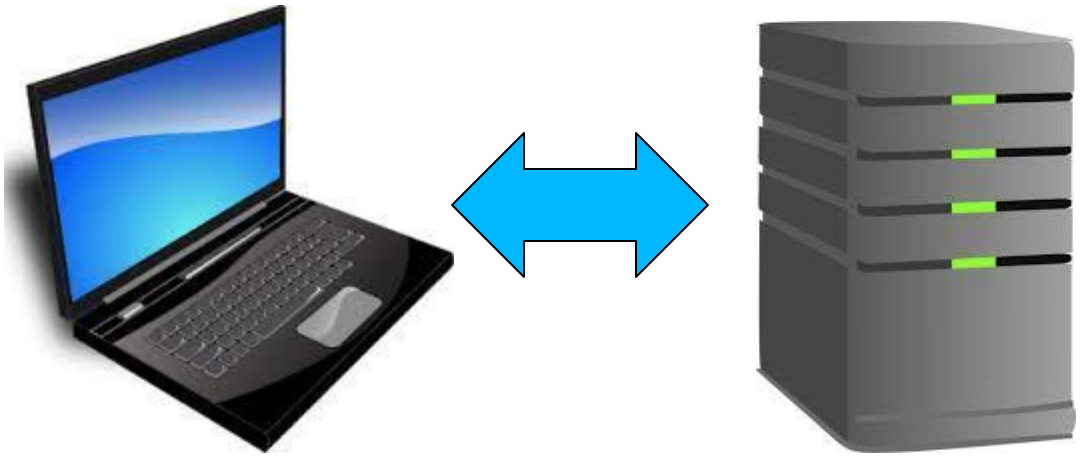
That's the **data**!

Review

- Node.js and Express allow us to build server-side web apps in JavaScript

- We can get data from the user via the URL query parameters or from form data

Server



Client

```
client and server
The server can generate the content, and the client can render web
content in the browser.
```

Server

Client

Database

But often, in a web application, behind the server, or communicating with the server, is some sort of database where we can persistently store data that can be accessed on future HTTP requests.

What is a NoSQL Database?

- A **NoSQL Database** is a database that does not use SQL, the traditional method of storing data

- In SQL, data is stored in tables and rows. This is also known as a **relational database**

- NoSQL Databases attempt to address some of the shortcomings of SQL and other relational databases by organizing and storing data differently

What is MongoDB?

- **MongoDB** is one NoSQL Database that **specifically** is designed for use with JavaScript apps

- MongoDB stores **collections** of **documents** (JS objects) rather than tables of rows

SQL Database

- A SQL table of Users might look like this:

Name	Age	Country	Occupation
Jane Doe	30	United States	Programmer
John Smith	25	Canada	Doctor
Kim Jones	27	France	Painter

```
downsides: if we want to add another column, every row or every record
would have to have a value in that column; if we wanted to do things
like searching for a name, that can be very expensive, depending on how
the data are organized; etc.
```

MongoDB Documents

- A MongoDB **collection** of User documents might look like this:

```
{
    name: 'Jane
    Doe', age: 30,
    country: 'United States',
    occupation: 'Programmer'
}
```

```
{
    name: 'John Smith',
    age: 25,
    country: 'Canada',
    occupation:
    'Doctor'
}
```

```
{
    name: 'Kim
    Jones', age: 27,
    country: 'France',
    occupation: 'Painter'
}
```

Mongo uses hashing functions that will allow us to easily find documents, rather than having to look through the entire table.

Using MongoDB with a Node.js app

1. Install MongoDB locally or create an account on a remote service

2. Install packages locally to allow your JavaScript programs to access your MongoDB instance

3. Write JavaScript to describe the Schema (blueprint for Documents) that you will use in the Collection

4. Use the Schema to access MongoDB in your app

Installing MongoDB

- You can find download/installation instructions for MongoDB at https://mongodb.com/download

- Follow these instructions to create a new empty database and run the MongoDB server

  https://www.mongodb.com/docs/manual/installation

- When you start it, it will tell you which port it is using

```
sina@sina:~$ mongosh
Current Mongosh Log ID: 64065d20831214ca8f3e06a0
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.8.0
Using MongoDB:          6.0.4
Using Mongosh:          1.8.0

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

------
   The server generated these startup warnings when booting
   2023-03-06T12:27:36.965-08:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesyste
m
   2023-03-06T12:27:37.514-08:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
   2023-03-06T12:27:37.514-08:00: vm.max_map_count is too low
------

------
   Enable MongoDB's free cloud-based monitoring service, which will then receive and display
   metrics about your deployment (disk utilization, CPU, operation statistics, etc).

   The monitoring data will be available on a MongoDB website with a unique URL accessible to you
   and anyone you share the URL with. MongoDB may use this information to make product
   improvements and to suggest MongoDB products and deployment options to you.

   To enable free monitoring, run the following command: db.enableFreeMonitoring()
   To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
------

test>
```

Here's a screenshot of what it looks like when I started Mongo on my Linux PC.

Installing MongoDB

- You can find download/installation instructions for MongoDB at https://mongodb.com/download

- Follow these instructions to create a new empty database and run the MongoDB server

  https://www.mongodb.com/docs/manual/installation

Installing MongoDB

- You can find download/installation instructions for MongoDB at https://mongodb.com/download

- Follow these instructions to create a new empty database and run the MongoDB server

  https://www.mongodb.com/docs/manual/installation

- Alternatively, you may use an online service, e.g. **MongoDB Atlas**

Installing Drivers for MongoDB

- You can access MongoDB directly from your Node app using the **MongoClient**

- Alternatively, you can install helper packages such as **Mongoose** to simplify some tasks:

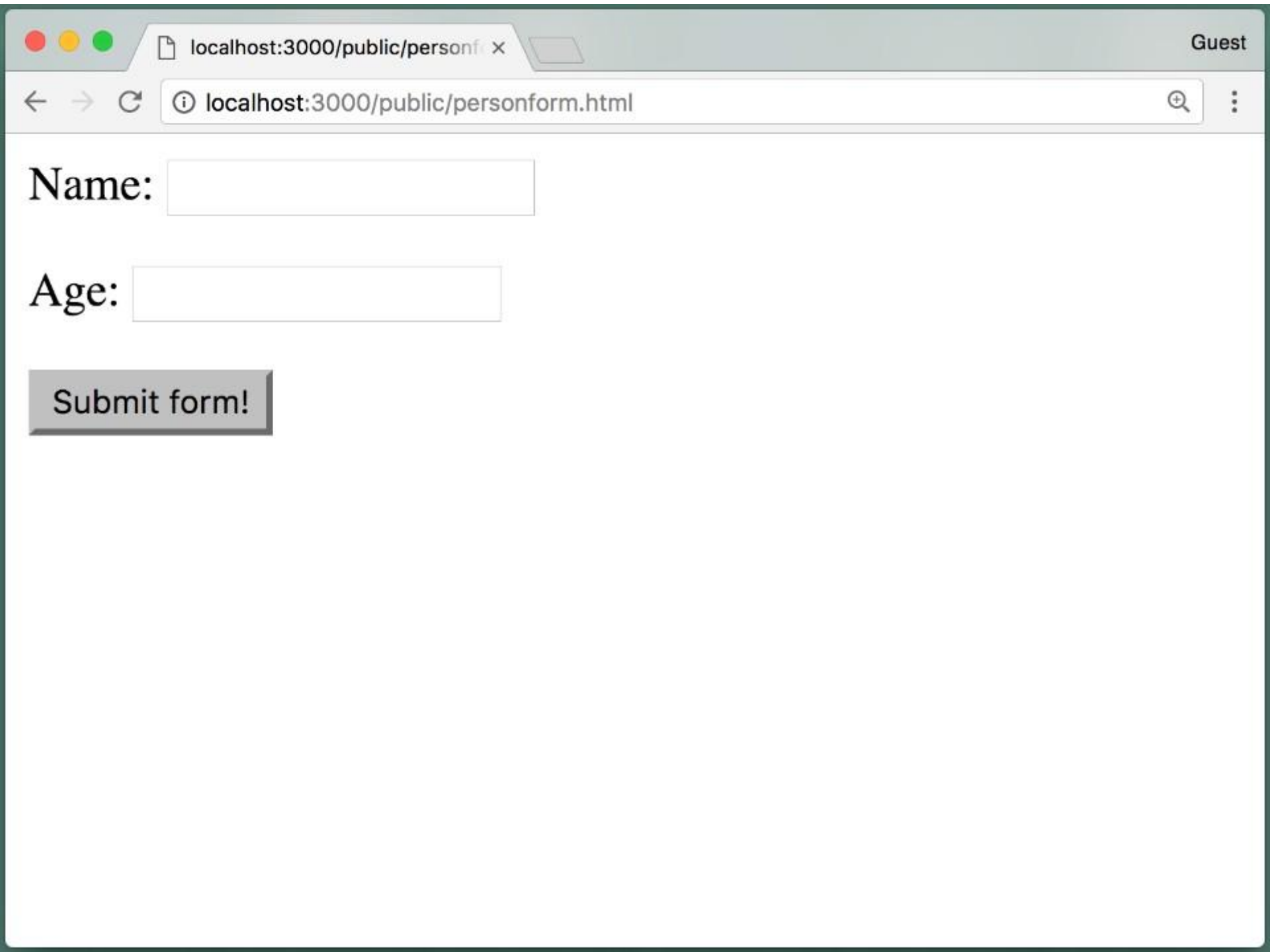
```
npm install mongoose --save
```

Official Learning Document

https://learn.mongodb.com/learning-paths/introduction-to-mongodb

Example - User DB

In the rest of this lecture, and in the next lecture, we're going to use this running example of where we want to have a form that will add some user data to our database.


localhost:3000/public/personf
Guest
localhost:3000/public/personform.html
Name:
Age:
Submit form!

```
// All actions at a glance
> mkdir app_people // create a app's directory
> cd app_people
> npm install mongoose --save
> npm install ejs
> npm install express --save
> npm init
> mkdir public // create public directory
> cd public
> touch personform.html
// write the form HTML page
> cd ..
> mkdir views
> cd views
> touch created.ejs
// write created.ejs file
> cd ..
> touch Person.js
// create database
> touch index.js
// write backend handler
> node index.js
```

```
<!-- public/personform.html -->
<html>

<body>
    <form action="/create" method="post">
        <p>
            Name: <input name="name">
        </p>
        <p>
            Age: <input name="age">
        </p>
        <p>
            <input type=submit value="Submit form!">
        </p>
    </form>
</body>

</html>
```

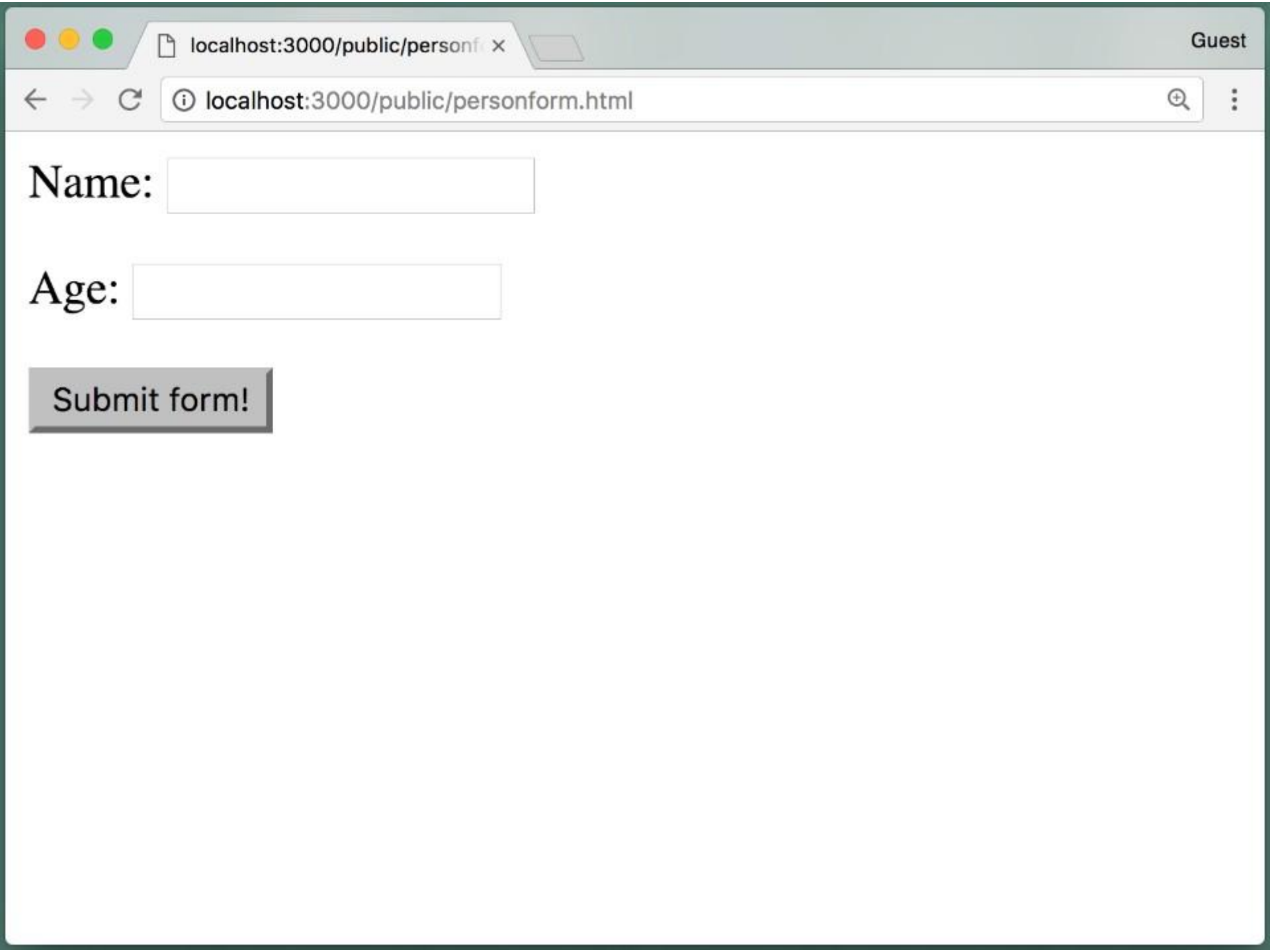
This is just static HTML that would be served from our Express app.

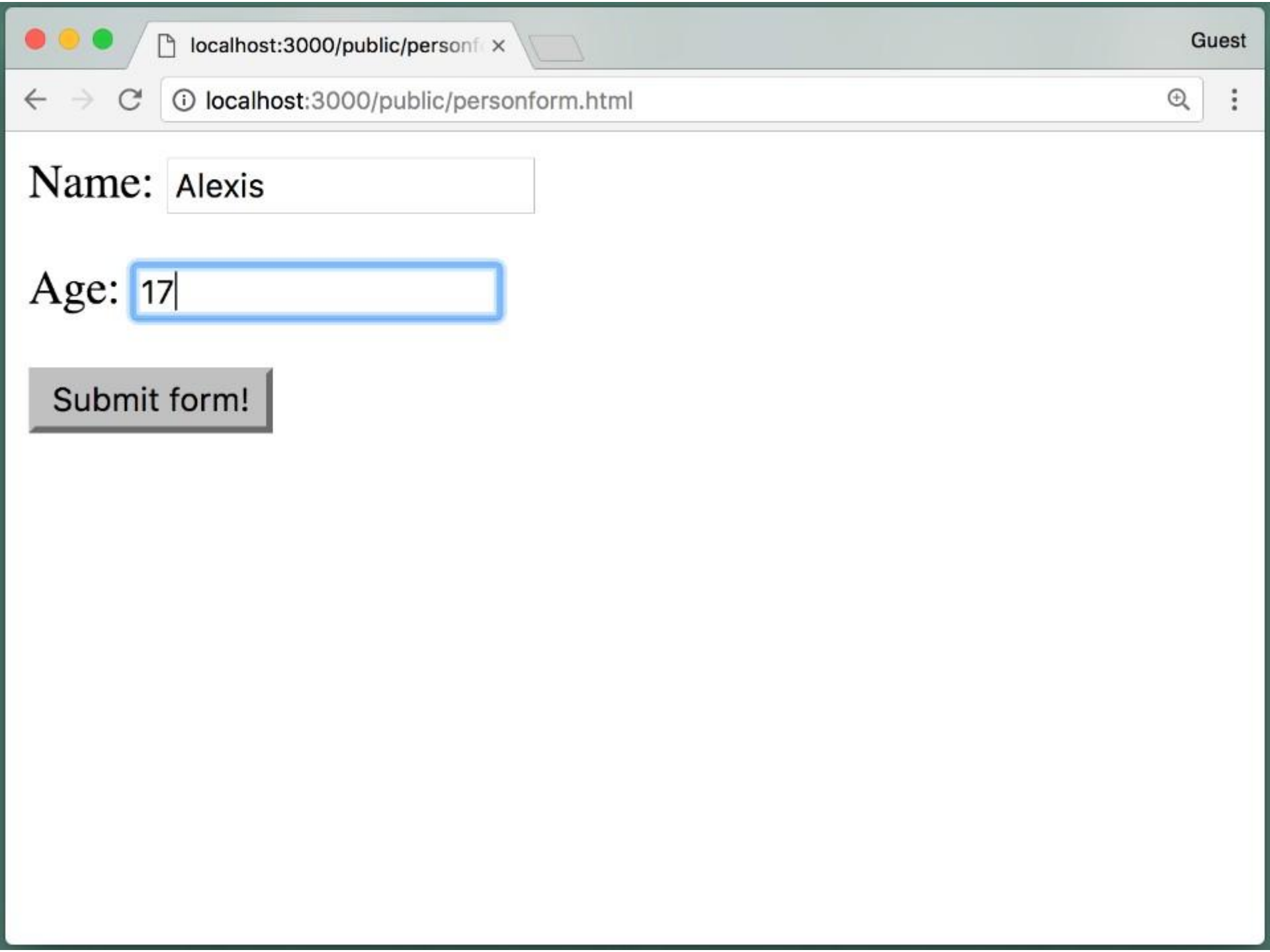
```
<!-- public/personform.html -->
<html>

<body>
   <form action="/create" method="post">
       <p>
           Name: <input name="name">
       </p>
       <p>
           Age: <input name="age">
       </p>
       <p>
           <input type=submit value="Submit form!">
       </p>
   </form>
</body>

</html>
```

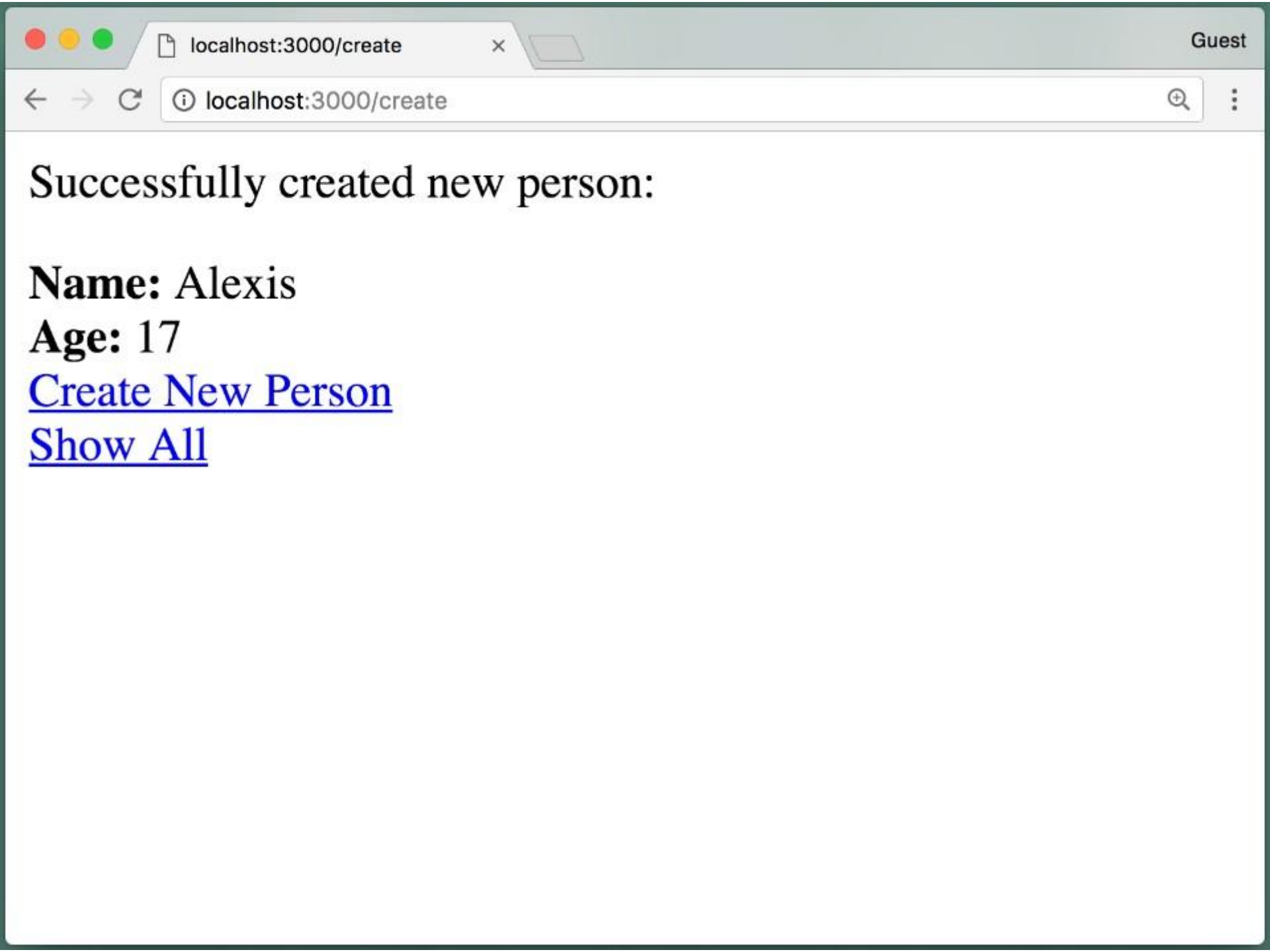
The action of this form will be /create.
That's the URL that will be requested when we submit the form.

```
<!-- public/personform.html -->
<html>

<body>
   <form action="/create"  method="post">
       <p>
           Name: <input name="name">
       </p>
       <p>
           Age: <input name="age">
       </p>
       <p>
           <input type=submit value="Submit form!">
       </p>
   </form>
</body>

</html>
```

Post method for sending the data from the form in the body of the http request.


localhost:3000/public/personf
Guest
localhost:3000/public/personform.html
Name:
Age:
Submit form!


localhost:3000/public/personf
Guest
localhost:3000/public/personform.html
Name: Alexis
Age: 17
Submit form!


localhost:3000/create
Guest
localhost:3000/create
Successfully created new person:
Name: Alexis
Age: 17
Create New Person
Show All

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase" );

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model( "Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

In Person.js, we're going to create our schema for MongoDB.
And then we'll export it so that other files can use it.

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase");

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

use the Mongoose package, which will simplify some tasks a little bit.

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase");

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

As mentioned earlier, we need to have Mongo running locally or have access to some cloud service, for instance, MongoDB Atlas.
The default port is 27017, but yours might be different.

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/ myDatabase");

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

I just chose the name of a database instance, myDatabase.
You can choose anything you want here.

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase");

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

So now we've connected to the Mongo server.
And now we're going to create the schema, or the blueprint, for the documents that will be stored in that database.

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase");

var Schema = mongoose.Schema;

var personSchema  = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

Now we create the variable called, personSchema, which is the blueprint for how we'll store person documents or objects in my collection in my database.

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase");

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

We use the Mongoose Schema, and specify that a person will have two properties: name and age

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase");

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true } ,
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

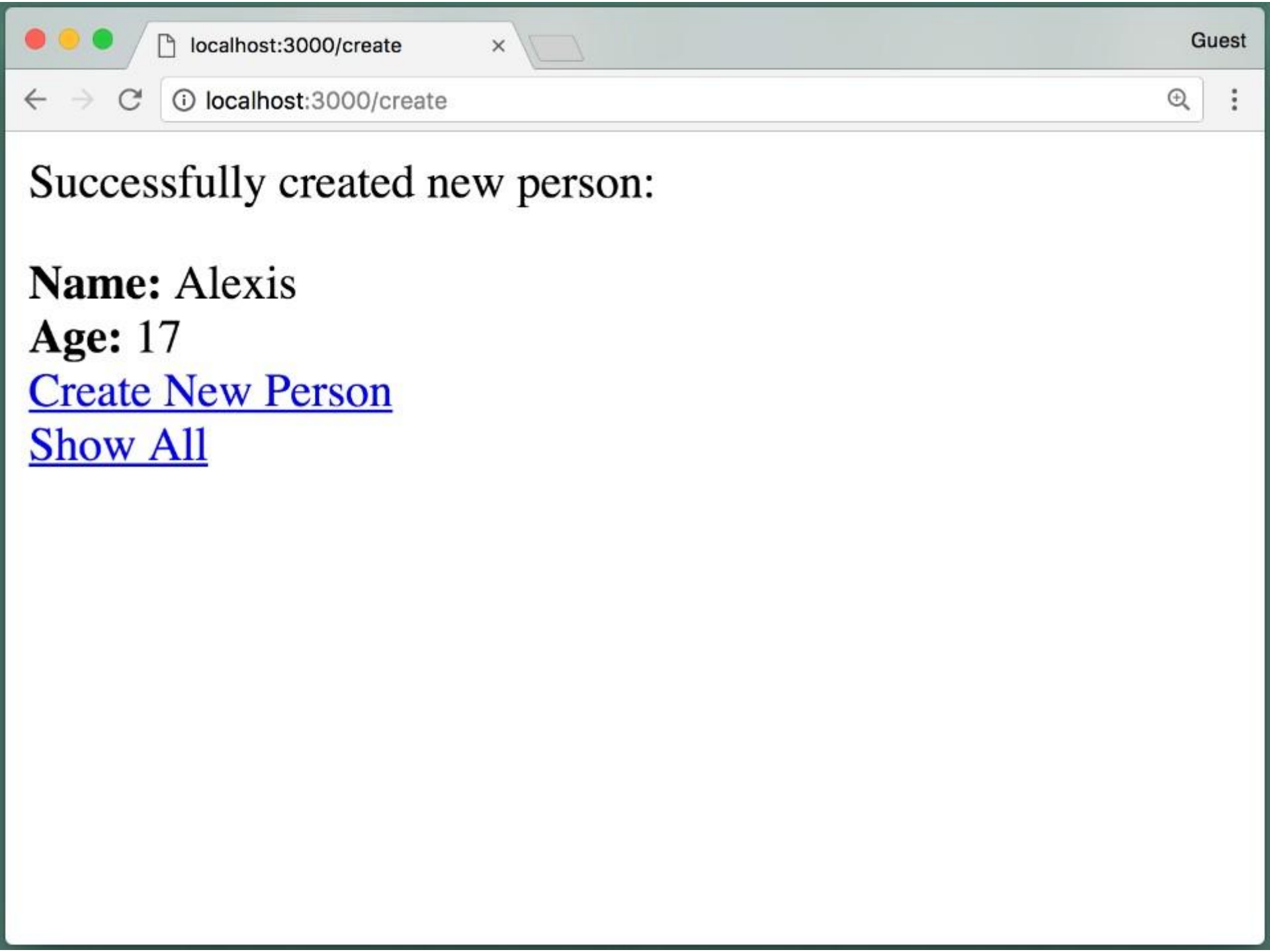
```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase");

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase");

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

we export this model using mongoose.model. And we'll say that we're going to export this model as the Person class.
That is, in other files, for instance index.js, we'll just create a new person, which is really an instance of personSchema.

```
/* This is Person.js */

var mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/myDatabase");

var Schema = mongoose.Schema;

var personSchema = new Schema({
 name: { type: String, required: true, unique: true },
 age: Number,
});

module.exports = mongoose.model("Person", personSchema);

personSchema.methods.standardizeName = function () {
 this.name = this.name.toLowerCase();
 return this.name;
};
```

The purpose of the standardizeName() method is to convert the name property of a person document to lowercase letters and then return the standardized name.


localhost:3000/create
Guest
localhost:3000/create
Successfully created new person:
Name: Alexis
Age: 17
Create New Person
Show All

We've set up our database connection, we've created the schema.
Now let's use that schema to insert a user into our database.

```
/* This is index.js */
var express = require("express");
var app = express();

app.set("view engine", "ejs");

var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({  extended: true }));

var Person = require("./Person.js");

... Next Slide Here

app.listen(3000, () => {
 console.log("Listening on port 3000");
});
```

```
/* This is index.js */
var express = require("express");
var app = express();

app.set("view engine", "ejs");

var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: true }));

var Person = require("./Person.js");

... Next Slide Here

app.listen(3000, () => {
 console.log("Listening on port 3000");
});
```

We're requiring Express and using Express to create the app.

```
/* This is index.js */
var express = require("express");
var app = express();

app.set("view engine", "ejs");

var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: true }));

var Person = require("./Person.js");

... Next Slide Here

app.listen(3000, () => {
 console.log("Listening on port 3000");
});
```

We're going to use EJS as our view engine.

```
/* This is index.js */
var express = require("express");
var app = express();

app.set("view engine", "ejs");

var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: true }));

var Person = require("./Person.js");

... Next Slide Here

app.listen(3000, () => {
 console.log("Listening on port 3000");
});
```

include the person class by requiring Person.js

```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

we'll use the express.static middleware to serve static content.

```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

if we get a URI request for /, we're going to use a redirect function in the response to redirect to this page here.

```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

/create URI

```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

The **async** keyword is used to define an asynchronous function. This means that the function will not block the execution of other code while it is waiting for an asynchronous operation to complete. The async keyword in JavaScript is used to indicate that a function is asynchronous, which means that it may take some time to complete its task.

```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

create a variable called newPerson.
Person is the class that we defined in Person.js, and required here in our index.js file.

```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

we've done so far is read the name and age out of the body, which were put there by the body parser that came from the form data.

```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

We invoke the newPerson.save function to put newPerson in our Mongo database

```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

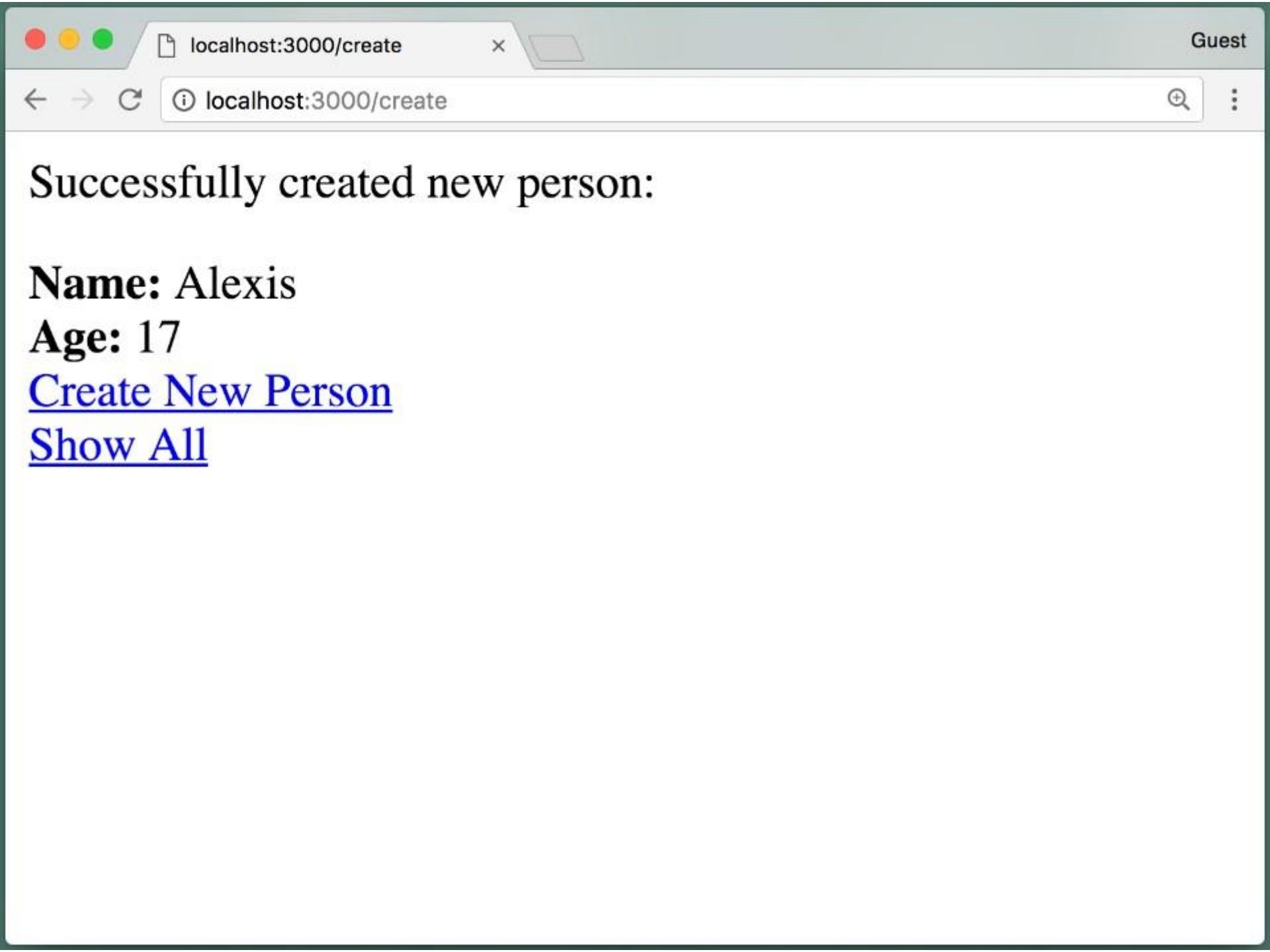
The await keyword is used here to wait for the newPerson.save() function to complete before moving on to the next line of code.

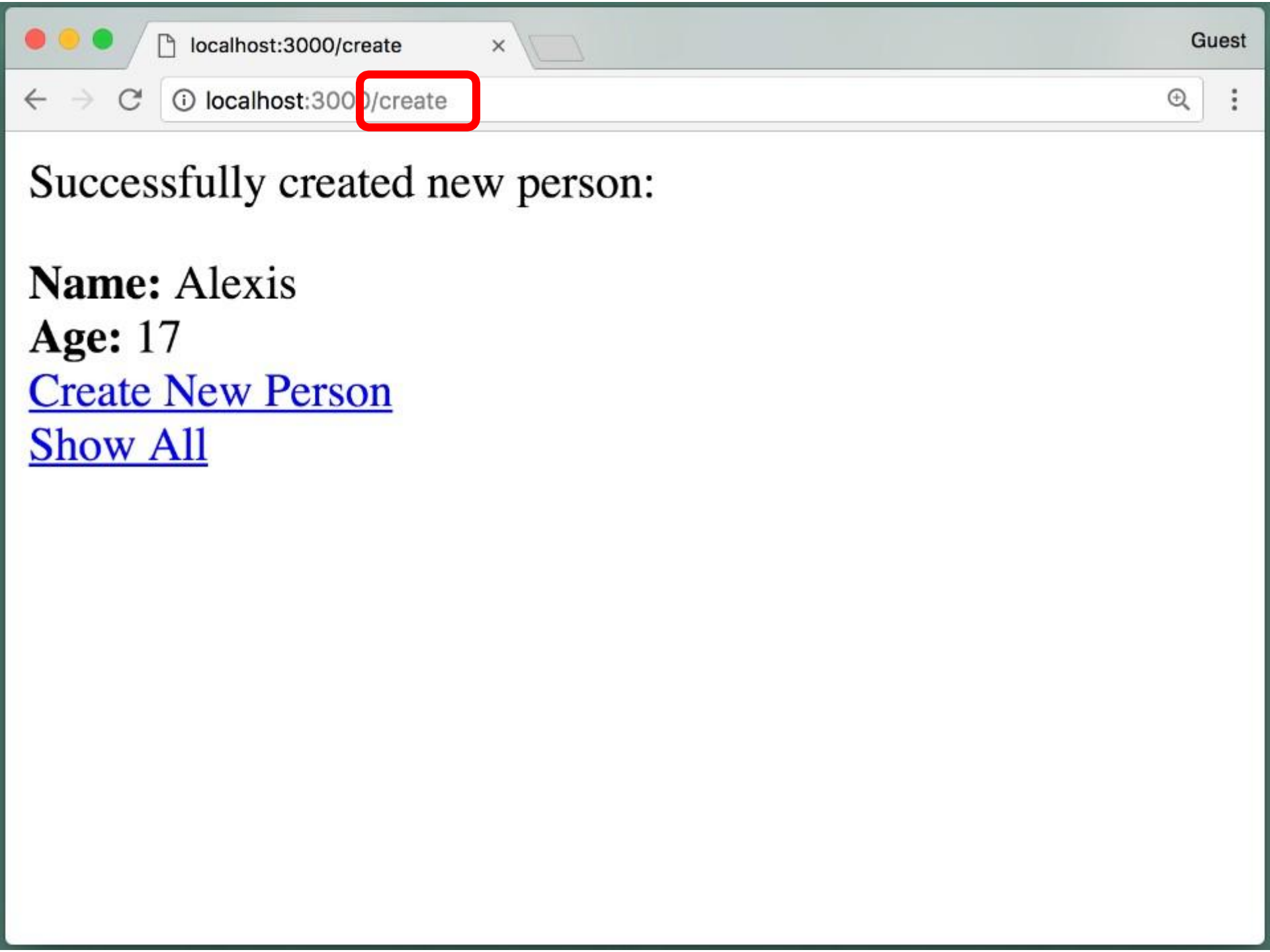
```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

use EJS to generate the HTML that we'll send back to the browser.
This will look for the file called views/created.ejs, render the HTML, and send it back.

```
/* This is index.js */
... // previous slide
var Person = require("./Person.js");

app.use("/create", async (req, res) => {
 var newPerson = new Person({
   name: req.body.name,
   age: req.body.age,
 });

 try {
   const result = await newPerson.save();
   res.render("created", { person: newPerson });
 } catch (err) {
   res.type("html").status(500);
   res.send("Error: " + err);
 }
});

app.use("/public", express.static("public"));

app.use("/", (req, res) => {
 res.redirect("/public/personform.html");
});
... // previous slide
```

if any error occurred while attempting to save this information, send back an http response, setting the content type to html, but the status code to 500. 500 is the status code for when there's an error.


localhost:3000/create
Guest
localhost:3000/create
Successfully created new person:
Name: Alexis
Age: 17
Create New Person
Show All


localhost:3000/create
Guest
localhost:3000/create
Successfully created new person:
Name: Alexis
Age: 17
Create New Person
Show All

```
<!-- This is views/created.ejs -->

Successfully created new person:
<p>
   <b>Name:</b> <%= person.name %>
   <br>
   <b>Age:</b> <%= person.age %>
   <br> <a href='/public/personform.html'>Create New Person</a>
   <br> <a href='/all'>Show All</a>

</p>
```

How do I know that my data really got written to my Mongo database?

run the MongoDB shell using this command:

```
> use myDatabase
> db.people.find()
```

The shell allows you to interact with the database.

```
myDatabase> use myDatabase
already on db myDatabase
myDatabase> db.people.find()

myDatabase>
```

```
myDatabase> db.people.find()

myDatabase> db.people.find()
[
  {
    _id: ObjectId("64067e8567cbcf0287364116"
    name: 'Alex',
    age: 22,
    __v: 0
  },
  {
    _id: ObjectId("64067e8b67cbcf0287364118"
    name: 'Ben',
    age: 33,
    __v: 0
  }
]
myDatabase>
```

Reset a MongoDB database

Open a terminal and start the **mongo** shell

```
> mongo
```

Switch to the database

```
> use <database_name>
```

Drop the database

```
> db.dropDatabase()
```

Verify that the database has been dropped

```
> show dbs
```

Keep in mind that this will permanently delete all data in the database, so make sure to back up any important data before proceeding with the reset.

Source code:
app_people.zip

Summary

- **MongoDB** is a NoSQL Database that is designed for use with JavaScript apps
- MongoDB stores **collections** of **documents**
- We can access MongoDB from our Node/Express app using libraries such as Mongoose
- We define a **Schema** and then can create new documents using the **`save`** function