**Tutorial**

**Web API with ASP.NET Core and MongoDB**

In this tutorial, you learn how to:

* Configure MongoDB
* Create a MongoDB database
* Define a MongoDB collection and schema
* Perform MongoDB CRUD operations from a web API

**Prerequisites**

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-mac)
* [.NET Core SDK 2.2 or later](https://www.microsoft.com/net/download/all)
* [Visual Studio 2017 version 15.9 or later](https://www.visualstudio.com/downloads/?utm_medium=microsoft&utm_source=docs.microsoft.com&utm_campaign=button+cta&utm_content=download+vs2017) with the **ASP.NET and web development** workload
* [MongoDB](https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/)

**Configure MongoDB**

If using Windows, MongoDB is installed at *C:\Program Files\MongoDB* by default. Add *C:\Program Files\MongoDB\Server\<version\_number>\bin* to the Path environment variable. This change enables MongoDB access from anywhere on your development machine.

Use the mongo Shell in the following steps to create a database, make collections, and store documents. For more information on mongo Shell commands, see [Working with the mongo Shell](https://docs.mongodb.com/manual/mongo/#working-with-the-mongo-shell).

1. Choose a directory on your development machine for storing the data. For example, *C:\BooksData* on Windows. Create the directory if it doesn't exist. The mongo Shell doesn't create new directories.
2. Open a command shell. Run the following command to connect to MongoDB on default port 27017. Remember to replace <data\_directory\_path> with the directory you chose in the previous step.

console

 mongod --dbpath <data\_directory\_path>

 Open another command shell instance. Connect to the default test database by running the following command:

console

 mongo

 Run the following in a command shell:

console

 use BookstoreDb

If it doesn't already exist, a database named *BookstoreDb* is created. If the database does exist, its connection is opened for transactions.

 Create a Books collection using following command:

console

db.createCollection('Books')

The following result is displayed:

console

 { "ok" : 1 }

 Define a schema for the Books collection and insert two documents using the following command:

console

db.Books.insertMany([{'Name':'Design Patterns','Price':54.93,'Category':'Computers','Author':'Ralph Johnson'}, {'Name':'Clean Code','Price':43.15,'Category':'Computers','Author':'Robert C. Martin'}])

The following result is displayed:

console

 {

"acknowledged" : true,

"insertedIds" : [

ObjectId("5bfd996f7b8e48dc15ff215d"),

ObjectId("5bfd996f7b8e48dc15ff215e")

]

}

 View the documents in the database using the following command:

console

db.Books.find({}).pretty()

The following result is displayed:

console

1. {
2. "\_id" : ObjectId("5bfd996f7b8e48dc15ff215d"),
3. "Name" : "Design Patterns",
4. "Price" : 54.93,
5. "Category" : "Computers",
6. "Author" : "Ralph Johnson"
7. }
8. {
9. "\_id" : ObjectId("5bfd996f7b8e48dc15ff215e"),
10. "Name" : "Clean Code",
11. "Price" : 43.15,
12. "Category" : "Computers",
13. "Author" : "Robert C. Martin"
14. }
15. The schema adds an autogenerated \_id property of type ObjectId for each document.

The database is ready. You can start creating the ASP.NET Core web API.

**Create the ASP.NET Core web API project**

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio-mac)

1. Go to **File** > **New** > **Project**.
2. Select **ASP.NET Core Web Application**, name the project *BooksApi*, and click **OK**.
3. Select the **.NET Core** target framework and **ASP.NET Core 2.2**. Select the **API** project template, and click **OK**:
4. Visit the [NuGet Gallery: MongoDB.Driver](https://www.nuget.org/packages/MongoDB.Driver/) to determine the latest stable version of the .NET driver for MongoDB. In the **Package Manager Console** window, navigate to the project root. Run the following command to install the .NET driver for MongoDB:

PowerShell

1. Install-Package MongoDB.Driver -Version {VERSION}

**Add a model**

1. Add a *Models* directory to the project root.
2. Add a Book class to the *Models* directory with the following code:

C#

using MongoDB.Bson;

using MongoDB.Bson.Serialization.Attributes;

namespace BooksApi.Models

{

public class Book

{

[BsonId]

[BsonRepresentation(BsonType.ObjectId)]

public string Id { get; set; }

[BsonElement("Name")]

public string BookName { get; set; }

[BsonElement("Price")]

public decimal Price { get; set; }

[BsonElement("Category")]

public string Category { get; set; }

[BsonElement("Author")]

public string Author { get; set; }

}

}

In the preceding class, the Id property:

* Is required for mapping the Common Language Runtime (CLR) object to the MongoDB collection.
* Is annotated with [BsonId] to designate this property as the document's primary key.
* Is annotated with [BsonRepresentation(BsonType.ObjectId)] to allow passing the parameter as type string instead of ObjectId. Mongo handles the conversion from string to ObjectId.

Other properties in the class are annotated with the [BsonElement] attribute. The attribute's value represents the property name in the MongoDB collection.

**Add a CRUD operations class**

1. Add a *Services* directory to the project root.
2. Add a BookService class to the *Services* directory with the following code:

C#

using System.Collections.Generic;

using System.Linq;

using BooksApi.Models;

using Microsoft.Extensions.Configuration;

using MongoDB.Driver;

namespace BooksApi.Services

{

public class BookService

{

private readonly IMongoCollection<Book> \_books;

public BookService(IConfiguration config)

{

var client = new MongoClient(config.GetConnectionString("BookstoreDb"));

var database = client.GetDatabase("BookstoreDb");

\_books = database.GetCollection<Book>("Books");

}

public List<Book> Get()

{

return \_books.Find(book => true).ToList();

}

public Book Get(string id)

{

return \_books.Find<Book>(book => book.Id == id).FirstOrDefault();

}

public Book Create(Book book)

{

\_books.InsertOne(book);

return book;

}

public void Update(string id, Book bookIn)

{

\_books.ReplaceOne(book => book.Id == id, bookIn);

}

public void Remove(Book bookIn)

{

\_books.DeleteOne(book => book.Id == bookIn.Id);

}

public void Remove(string id)

{

\_books.DeleteOne(book => book.Id == id);

}

}

}

 Add the MongoDB connection string to *appsettings.json*:

C#

 {

"ConnectionStrings": {

"BookstoreDb": "mongodb://localhost:27017"

},

"Logging": {

"IncludeScopes": false,

"Debug": {

"LogLevel": {

"Default": "Warning"

}

},

"Console": {

"LogLevel": {

"Default": "Warning"

}

}

}

}

The preceding BookstoreDb property is accessed in the BookService class constructor.

 In Startup.ConfigureServices, register the BookService class with the Dependency Injection system:

C#

1. public void ConfigureServices(IServiceCollection services)
2. {
3. services.AddScoped<BookService>();
4. services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);
5. }
6. The preceding service registration is necessary to support constructor injection in consuming classes.

The BookService class uses the following MongoDB.Driver members to perform CRUD operations against the database:

* MongoClient – Reads the server instance for performing database operations. The constructor of this class is provided the MongoDB connection string:

C#

* public BookService(IConfiguration config)
* {
* var client = new MongoClient(config.GetConnectionString("BookstoreDb"));
* var database = client.GetDatabase("BookstoreDb");
* \_books = database.GetCollection<Book>("Books");
* }
* IMongoDatabase – Represents the Mongo database for performing operations. This tutorial uses the generic GetCollection<T>(collection) method on the interface to gain access to data in a specific collection. CRUD operations can be performed against the collection after this method is called. In the GetCollection<T>(collection) method call:
  + collection represents the collection name.
  + T represents the CLR object type stored in the collection.

GetCollection<T>(collection) returns a MongoCollection object representing the collection. In this tutorial, the following methods are invoked on the collection:

* Find<T> – Returns all documents in the collection matching the provided search criteria.
* InsertOne – Inserts the provided object as a new document in the collection.
* ReplaceOne – Replaces the single document matching the provided search criteria with the provided object.
* DeleteOne – Deletes a single document matching the provided search criteria.

**Add a controller**

1. Add a BooksController class to the *Controllers* directory with the following code:

C#

 using System.Collections.Generic;

using BooksApi.Models;

using BooksApi.Services;

using Microsoft.AspNetCore.Mvc;

namespace BooksApi.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class BooksController : ControllerBase

{

private readonly BookService \_bookService;

public BooksController(BookService bookService)

{

\_bookService = bookService;

}

[HttpGet]

public ActionResult<List<Book>> Get()

{

return \_bookService.Get();

}

[HttpGet("{id:length(24)}", Name = "GetBook")]

public ActionResult<Book> Get(string id)

{

var book = \_bookService.Get(id);

if (book == null)

{

return NotFound();

}

return book;

}

[HttpPost]

public ActionResult<Book> Create(Book book)

{

\_bookService.Create(book);

return CreatedAtRoute("GetBook", new { id = book.Id.ToString() }, book);

}

[HttpPut("{id:length(24)}")]

public IActionResult Update(string id, Book bookIn)

{

var book = \_bookService.Get(id);

if (book == null)

{

return NotFound();

}

\_bookService.Update(id, bookIn);

return NoContent();

}

[HttpDelete("{id:length(24)}")]

public IActionResult Delete(string id)

{

var book = \_bookService.Get(id);

if (book == null)

{

return NotFound();

}

\_bookService.Remove(book.Id);

return NoContent();

}

}

}

The preceding web API controller:

* Uses the BookService class to perform CRUD operations.
* Contains action methods to support GET, POST, PUT, and DELETE HTTP requests.
* The [CreatedAtRoute](https://docs.microsoft.com/dotnet/api/system.web.http.apicontroller.createdatroute) method returns a 201 response, which is the standard response for an HTTP POST method that creates a new resource on the server. CreatedAtRoute also adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See [10.2.2 201 Created](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html).

 Build and run the app.

 Navigate to http://localhost:<port>/api/books in your browser. The following JSON response is displayed:

JSON

1. [
2. {
3. "id":"5bfd996f7b8e48dc15ff215d",
4. "bookName":"Design Patterns",
5. "price":54.93,
6. "category":"Computers",
7. "author":"Ralph Johnson"
8. },
9. {
10. "id":"5bfd996f7b8e48dc15ff215e",
11. "bookName":"Clean Code",
12. "price":43.15,
13. "category":"Computers",
14. "author":"Robert C. Martin"
15. }
16. ]

**Next steps**