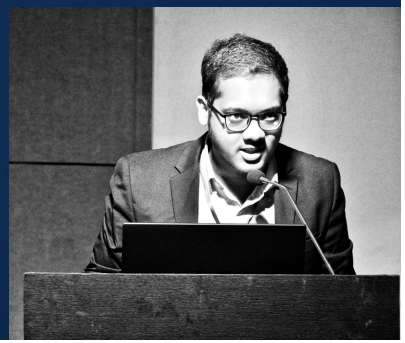
Building Minimal APIs - .NET

Presented by: Aditya Seth
Connect : adityaseth.in/linkedin



Agenda

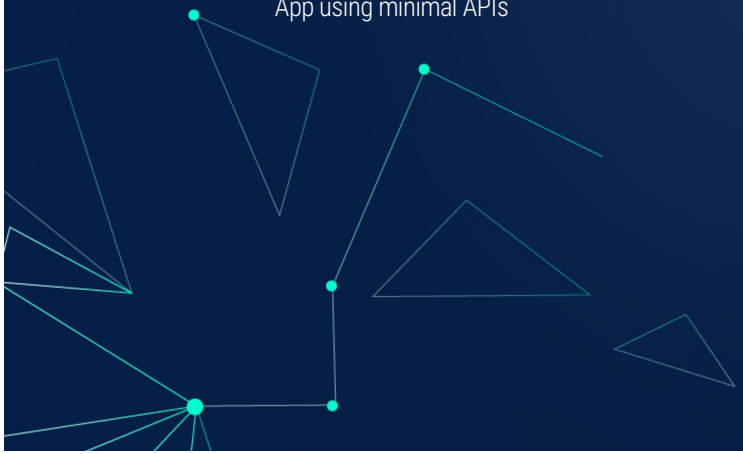
01 **Getting Started**
What to install, prerequisites, and learning objectives

02 **What are minimal APIs**
Low ceremony APIs in .NET

03 **Build a Todo API: Get started**
Let's build the backend to a Todo App using minimal APIs

04 **Build a Todo API: Build**
Build a CRUD application

05 **Build a Todo API: Databases**
Store Todo items in a database



01

What you need

VS Code

.NET Installed

Patience

:)

Getting Started

Here's what you'll need for this workshop:

- .NET 6 SDK (dot.net/download)
- An editor of your choice, such as VS Code or Visual Studio

If you do not have an editor, you can download both VS Code and the .NET 6 SDK...

For macOS
For Windows

Learning Objectives

Minimal APIs

Learn what a minimal API is and when you should use one.

Learn what a basic CRUD application is. Create your own minimal API for a Todo list . You will learn how to dynamically create, read, update, and delete Todo list items.

CRUD applications

Databases

Learn how to create a persistent database with SQLite.


02
What are minimal APIs?

Let's take a step back... What is an API?

💡 **API** stands for Application Programming Interface

These are collections of operations that you can invoke on a machine either locally or remotely.

💻 **Web APIs** are operations you can invoke over HTTP

This allows you to leverage the functionality that those operations provide.

❗ **Building an API can be complex**

They need to support many features like routing, reading and writing to data storage, and authentication.
This is where **minimal APIs** come in!

Minimal APIs

Minimal APIs are a low ceremony way to build HTTP APIs in ASP.NET Core. Minimal APIs hook into ASP.NET Core's hosting and routing capabilities and allow you to create fully functioning APIs with just a few lines of code.

Minimal First

Build an API in C# with just 3 lines of code.

Grows With You

The C# ecosystem powers the most productive applications on the web and will support your project, no matter the size.

Incredibly Fast

Proven to be one of the fastest web servers in the world, serving more than 4 million requests per second.


03
Build a Todo API
Getting started

Create and run your Todo API

01 Create your API using the .NET CLI

```
dotnet new web -o TodoApi
```

02 Run your minimal API

Now, that you have created your minimal API you can run it.
Navigate to the `TodoApi` folder and type the command below

```
TodoApi> dotnet watch
```

03 See your API in action

Navigate to http://localhost:[port] in a browser and see the text Hello World!

Add a new route

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Open your **TodoApi** app in an editor of your choice and open the **Program.cs** file. Your **Program.cs** looks like the code above.

```
app.MapGet("/todo", () => new { Item = "Water plants", Complete = "false" });
```

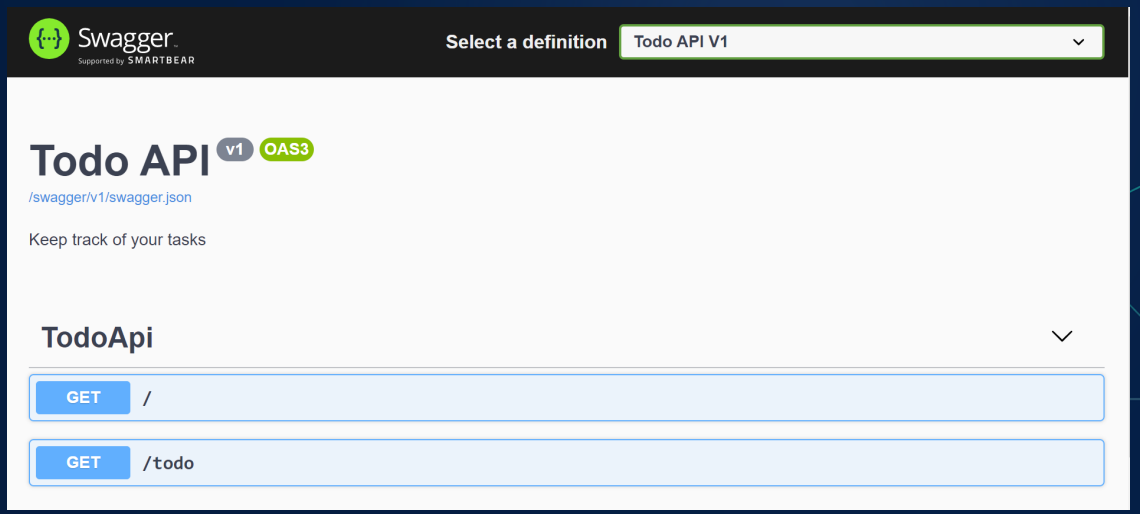
In **Program.cs**, create new todo route to our api that returns a list items. Add a new **MapGet** after **app.MapGet("/", () => "Hello World!");**

Navigate to https://localhost:[port]/todo to see this newly created list item

Bonus: Configure OpenAPI and Swagger UI

Install the Swashbuckle.AspNetCore package to add Swagger support.

```
TodoApi> dotnet add package Swashbuckle.AspNetCore
```



Configure OpenAPI and Swagger UI

Snippet 1: Under `var builder = WebApplication.CreateBuilder(args);`, add the following lines of code.

```
builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "Todo API", Description = "Keep track of your tasks",
Version = "v1" });
});
```

The **AddSwaggerGen** method adds information such as title, description, and version to your API.

Snippet 2: Above `app.MapGet("/", () => "Hello World!");`, add following lines of code:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
   c.SwaggerEndpoint("/swagger/v1/swagger.json", "Todo API V1");
});
```

Go back to your browser where your app is and navigate to this

04

Build a Todo API!

Before, you built simple API where you hard coded the results to HTTP method.
Now, we are going to step it up a notch and create something **dynamic**. Instead of returning a static item that is hardcoded to our route, we are going to be creating to-do list we can update, create new items, mark an item as complete and delete an item.

Create Read Update Delete

Our todo API is going to...

Create a new item	Return a specific item on a list	Update an existing item	Delete an item

Create a data model

Create a class that models the data we want to collect aka Data model. The code for your `TodoItem` will go after `app.Run();`

```
class TodoItem
{
    public int Id { get; set; }
    public string? Item { get; set; }
    public bool IsComplete { get; set; }

}
```

This is how you define the data that we want to collect.

Store an item

Install the Entity Framework Core InMemory package

Entity Framework is a code library that enables the transfer of data stored in relational database tables into objects that are more commonly used in application code.

```
TodoApi>dotnet add package Microsoft.EntityFrameworkCore.InMemory
```

Add `using Microsoft.EntityFrameworkCore;` to the top of your `Program.cs` file:

EntityFramework let's us wire up our code to the data we want save and query it. To do this, let's create a `TodoDb` class!

Snippet 1:

Below the `TodoItem` create a `TodoDb` class

```
class TodoDb : DbContext
{
    0 references
    public TodoDb(DbContextOptions options) : base(options) { }
    6 references
    public DbSet<TodoItem> Todos { get; set; }
}
```

Snippet 2:

Before `AddEndpointsApiExplorer` services we configured in the first tutorial add the code snippet below

```
builder.Services.AddDbContext<TodoDb>(options => options.UseInMemoryDatabase("items"));
```

To read from a list of items in the todo list replace the "/todo" route with the "/todos" route below.

```
app.MapGet("/todos", async (TodoDb db) => await db.Todos.ToListAsync());
```

Create new items

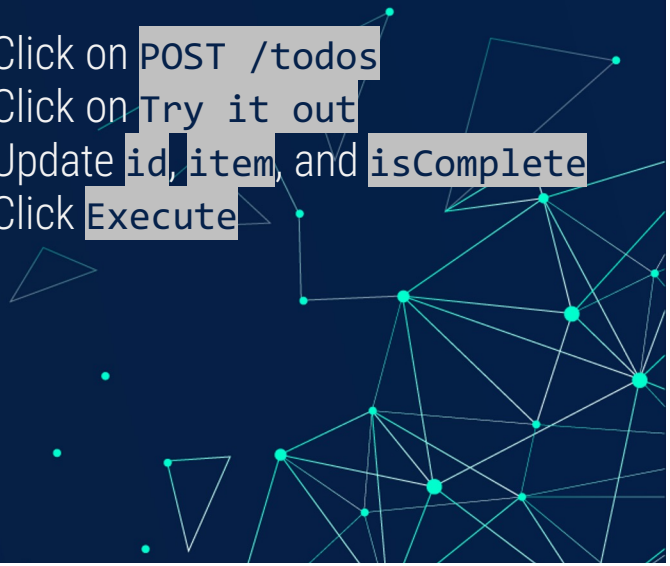
Let's `POST` new tasks to the todos list.
Below `app.MapGet` you create earlier.

```
app.MapPost("/todos", async (TodoDb db, TodoItem todo) =>
{
    await db.Todos.AddAsync(todo);
    await db.SaveChangesAsync();
    return Results.Created($"/todo/{todo.Id}", todo);
});
```

Try it out!

Go back to `Swagger` and now you should see `POST /todos`. To add new items to the todo list

1. Click on `POST /todos`
2. Click on `Try it out`
3. Update `id`, `item`, and `isComplete`
4. Click `Execute`



Read the items

```
[
  {
    "id": 1,
    "item": "Buy groceries",
    "isComplete": true
  },
  {
    "id": 2,
    "item": "Water plants",
    "isComplete": false
  }
]
```

1. Click on `GET /todos`
2. Click on `Try it out`
3. Click `Execute`

The `Response body` will include the items just added

To `GET` an item by `id` add the code below `app.MapPost` route created earlier

```
app.MapGet("/todos/{id}", async (TodoDb db, int id) => await db.Todos.FindAsync(id));
```

To check this out you can either go to `https://localhost:5001/todos/1` or use the Swagger UI.

Update an item

To update an existing item add the code below `GET /todos/{id}` route we created above.

```
app.MapPut("/todos/{id}", async ( TodoDb db, TodoItem updateTodo ,int id) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Item = updateTodo.Item;
    todo.IsComplete = updateTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});
```

1. Click on `PUT/todos/{id}`
2. Click on `Try it out`
3. In the `id` text box enter 2
4. Update `Request body` paste the JSON below and update `isComplete` to `true`.

```
{
    "id": 2,
    "item": "Water plants",
    "isComplete": false
  }
```

5. Click `Execute`

To test, scroll to `GET/todos/{id}` and now Water Plants is completed

Delete an item

To delete an existing item add the code below `PUT/todos/{id}` we created above.

```
app.MapDelete("/todos/{id}", async (TodoDb db, int id) =>
{
    var todo = await db.Todos.FindAsync(id);
    if (todo is null)
    {
        return Results.NotFound();
    }
    db.Todos.Remove(todo);
    await db.SaveChangesAsync();

    return Results.Ok(todo);
});
```

Now, try deleting an item.


05
Databases
Let's work with a persistent database so your data will be saved even after you shut down your application.

Working with Databases

We just learned how to build a basic CRUD application with an in-memory database.

Let's step it up notch and work with a persistent database. Meaning your data will be saved even after you shut down your application.

For this tutorial we will be using SQLite database.

Setup SQLite database

Install the following tools and packages

SQLite EF Core Database Provider: can access many different databases through plug-in libraries called database providers.

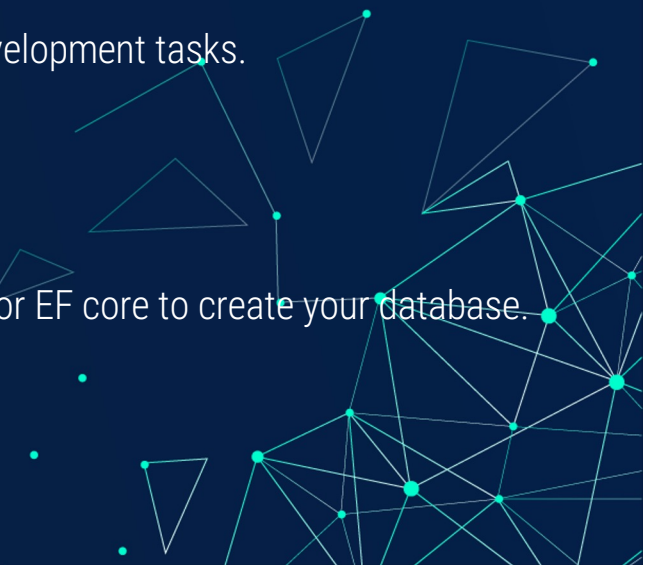
```
TodoApi>dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

Entity Framework Core tools: tools for Entity Framework Core perform design-time development tasks.

```
TodoApi>dotnet tool install --global dotnet-ef
```

`Microsoft.EntityFrameworkCore.Design`: contains all the design-time logic for EF core to create your database.

```
TodoApi>dotnet add package Microsoft.EntityFrameworkCore.Design
```



Enable database creation

01 Set the database connection string

02 Migrate your data model to a SQLite database.
Create a data model

```
class TodoItem
{
    public int Id { get; set; }
    public string? Item { get; set; }
    public bool IsComplete { get; set; }

}
```

03 Create your database and schema

Set connection string

In `Program.cs` below your app builder `var builder = WebApplication.CreateBuilder(args);` add a connection string.

```
var connectionString = builder.Configuration.GetConnectionString("todos") ?? "Data Source=todos.db";
```

Add context to your services

Now we are going to replace the in-memory database with a persistent database.

Replace your current in-memory database implementation `builder.Services.AddDbContext<TodoDb>(options => options.UseInMemoryDatabase("items"));` in your build services with the SQLite one below:

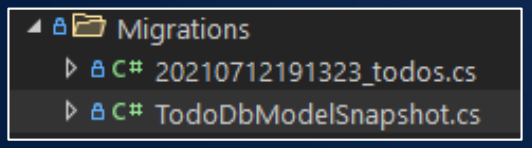
```
builder.Services.AddSqlite<TodoDb>(connectionString);
```

Migrate data model

With EF Core migration tool, you can now start your first migration `InitialCreate`. In a terminal window, run the `migrations` command below:

```
TodoApi> dotnet ef migrations add InitialCreate
```

EF Core will create a folder called Migrations in your project directory containing two files

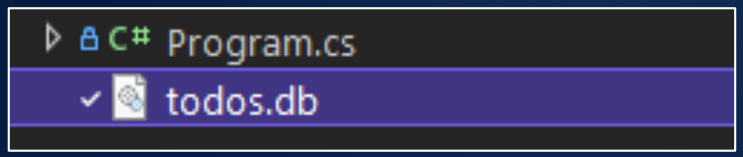


Create your database and schema

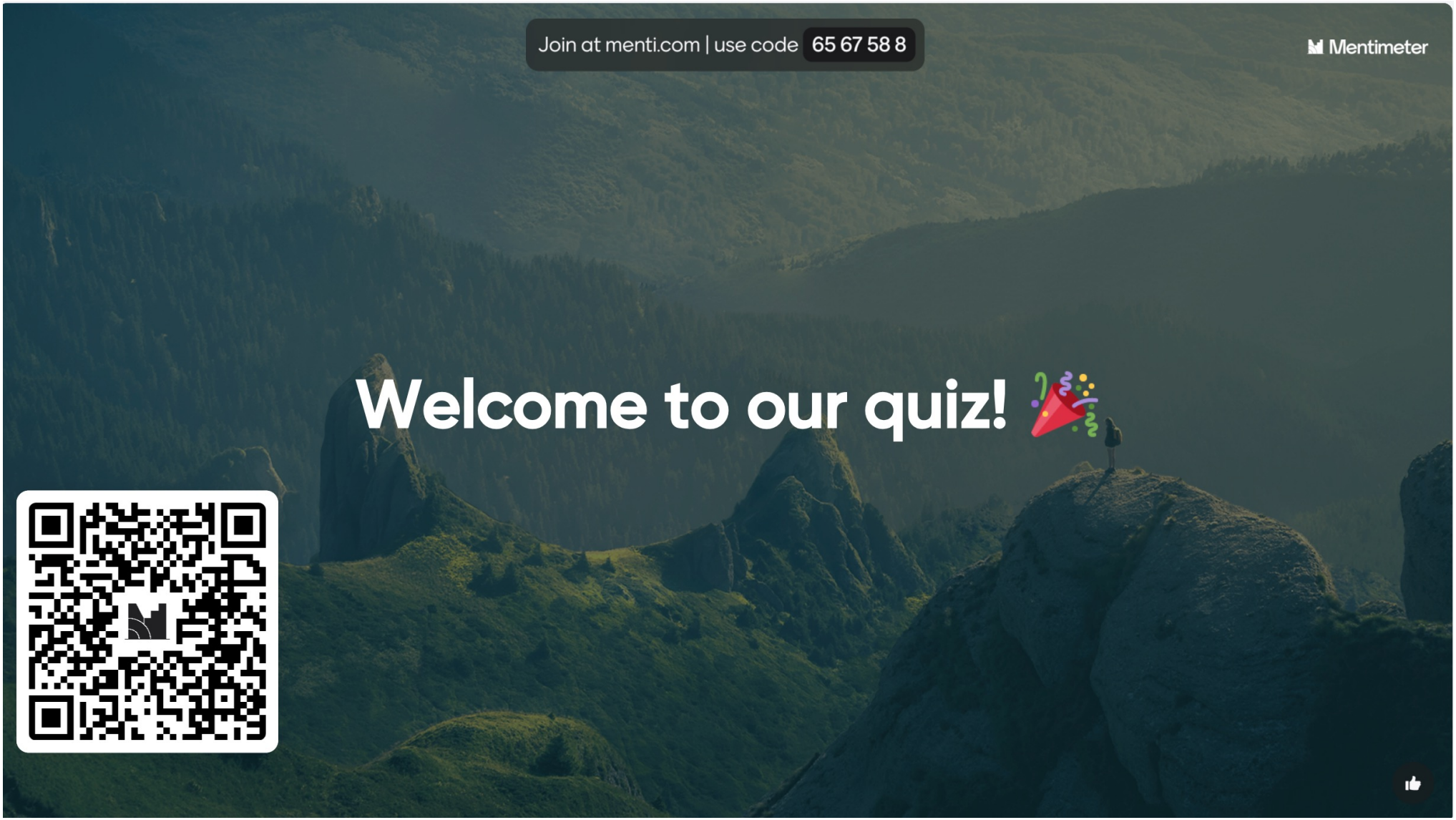
Now that you have completed the migration, you can use it to create your database and schema. In a terminal window, run the `database update` command below to apply migrations to a database:

```
TodoApi> dotnet ef database update
```

You should see a newly created `todos.db` file in your project directory:



Test the application again in Swagger – you will see that your Todo items are stored in the database, so they will remain if you stop and start the app.


Join at menti.com | use code 65 67 58 8
Mentimeter
Welcome to our quiz!


THANKS
Happy Coding!