[Hopp til innhold](https://chatgpt.com/c/68c33b27-3aec-8327-8df9-7975f5e7e63a" \l "main)

https://chatgpt.com/c/68c33b27-3aec-8327-8df9-7975f5e7e63a

* **Chattehistorikk**
* **Du sa:**

I am a total newbie in Windows programming so I wonder how do I VIBE web develop with .net, C#, GitHub Copilot code helper and with commit in the IDE Visual Studio Code to program an ToDo webapp. I like to store content and tables in a live free database and publish the code on a free webserver so I can reach it from an weburl. Can you make me an 8 chapters tutorial (step by step)? Show me first the table of contents and show med one chapter at a time.

* **ChatGPT sa:**

Tenkte i 32s

Nice — let’s do this! I’ll give you an 8-chapter beginner-friendly tutorial that walks you from zero → live ToDo webapp (C#, .NET, VS Code + GitHub Copilot, free live DB, and a free URL).

First I’ll show the **Table of contents**, then I’ll deliver **Chapter 1** (Environment & tools). After you read Chapter 1, say **“Next”** and I’ll show Chapter 2, etc — one chapter at a time as you requested.

* **Table of contents (8 chapters)**

1. **Setup: install .NET, VS Code, Git, GitHub, and GitHub Copilot** — get your dev environment ready on Windows.
2. **Scaffold the project** — create a simple Blazor WebAssembly app (C# front-end) so you can write UI in C#.
3. **Make a local ToDo prototype (model + mock service)** — build the Todo model and a simple local data service so you can iterate fast.
4. **Create a free live database (Supabase)** — sign up, create a todos table, get your API URL & anon key (free Postgres).
5. **Wire the app to Supabase** — implement an HTTP service in C# to do Create/Read/Update/Delete against Supabase REST endpoints.
6. **UI: list, add, edit, delete, UX polish** — build the Razor/Blazor pages and use Copilot prompts to speed coding.
7. **Git + GitHub + Copilot workflow in VS Code** — init repo, commit, push, use Copilot effectively, and add CI (GitHub Action).
8. **Deploy to a free host & secure keys** — publish the app to a free static host (GitHub Pages or Netlify) and set secrets; final checks and next features.

* **Chapter 1 — Setup: install .NET, VS Code, Git, GitHub, and GitHub Copilot**

Goal: get a working Windows dev environment so you can create, build, run, and commit .NET/C# web code from VS Code.

* **1.1 What you'll install (quick list)**
* **.NET SDK** (the .NET SDK that lets you dotnet new, build and run apps).
* **Visual Studio Code** (lightweight editor with extensions).
* **Git** (version control CLI).
* **GitHub account** (host your repo and later publish).
* **GitHub Copilot extension** in VS Code (AI code helper).
* Optional but recommended: **Windows Terminal** (better terminal), **REST Client** extension (test APIs), **C# extension** (for C# intellisense).
* **1.2 Step-by-step install (Windows, beginner-friendly)**
* **.NET SDK**

1. Open a browser and go to the official .NET download page (search “dotnet download”).
2. Download and install the **.NET SDK** for Windows (the installer). Accept defaults.
3. Verify: open PowerShell (or Windows Terminal) and run:
4. dotnet --version

You should see a version number (e.g. 8.0.x or similar). If you see a number, .NET is installed.

* **Visual Studio Code**

1. Download VS Code from code.visualstudio.com and run the installer.
2. Open VS Code after install. (Pin it to the taskbar if you like.)

* **Git**

1. Download Git from git-scm.com and install using the default options.
2. Verify in a terminal:
3. git --version

* **GitHub account & Personal setup**

1. Create a free account at github.com (if you don’t have one).
2. In GitHub, you’ll later create a repository to push your code to. For now just create the account.

* **GitHub Copilot (in VS Code)**

1. In VS Code open the **Extensions** panel (left icon or Ctrl+Shift+X).
2. Search for **GitHub Copilot** and install the official extension.
3. Click **Sign in to GitHub** when prompted and authorize Copilot.
   * Copilot may require a subscription or trial (GitHub offers trials / student licenses sometimes). If you don’t have access, you can still code; Copilot is optional but helpful.

* **Useful VS Code extensions to install now**
* **C#** (adds language support, debugging).
* **GitLens** (supercharges Git UI).
* **REST Client** (send HTTP requests from an editor tab — great for testing Supabase).
* **EditorConfig for VS Code** (optional).

Install them from the Extensions view in VS Code.

* **1.3 Configure Git (one-time)**

From a terminal inside VS Code or PowerShell:

git config --global user.name "Your Name"

git config --global user.email "you@example.com"

Optional: set up SSH keys for GitHub (search “GitHub SSH key Windows”). If you prefer HTTPS, you can push with username/password or GitHub token.

* **1.4 Bonus: quick check commands**

Open a new terminal (inside VS Code: `Ctrl+``) and run:

dotnet --version

git --version

code --version

All should print versions. If yes — you're set.

* **1.5 Quick notes (pitfalls & tips)**
* If dotnet --version fails, reboot or ensure the PATH was updated.
* Copilot suggestions appear inline — press Tab to accept, Esc to ignore. Learn to prompt Copilot with short comments like // Create method to fetch todos from Supabase and then accept the suggestion.
* Keep your GitHub credentials handy. VS Code will help with sign-in flows when you push for the first time.

That’s Chapter 1 — a working environment. When you’re ready I’ll show **Chapter 2 — Scaffolding the Blazor WebAssembly project** with exact commands, the basic project layout, and a tiny “Hello world” ToDo page so you can run the app locally.

Say **Next** when you want Chapter 2 (or tell me if you prefer a different stack — e.g., ASP.NET MVC or a React front-end — I’ll adapt the tutorial).

* **Du sa:**

Blazor is good. :) Next

* **ChatGPT sa:**

Perfect 😎 Blazor it is! We’ll use **Blazor WebAssembly** (C# in the browser, no backend server required). Later we’ll call the Supabase API from the client directly.

* **Chapter 2 — Scaffold the Blazor WebAssembly Project**

Goal: create the ToDo app skeleton, run it locally, and understand the folder structure.

* **2.1 Create the project**

Open VS Code terminal (`Ctrl+``) and run:

dotnet new blazorwasm -o TodoApp cd TodoApp

* dotnet new blazorwasm → creates a Blazor WebAssembly project.
* -o TodoApp → puts it in a new folder called **TodoApp**.
* **2.2 Run the project**

Still in the terminal:

dotnet run

You’ll see something like:

Now listening on: https://localhost:7173

Now listening on: http://localhost:5173

Open the https://localhost:7173 link in your browser. You’ll see the default Blazor “Hello, world!” app with a counter page.

✅ Congrats — you’ve got a working C# web app running in your browser!

* **2.3 Explore the project structure**

Inside **TodoApp**, notice these key folders:

* **wwwroot/** → static files (CSS, JS, images).
* **Pages/** → Blazor pages (.razor files). Default ones are Index.razor, Counter.razor, and FetchData.razor.
* **Shared/** → reusable components (e.g., NavMenu.razor).
* **Program.cs** → app startup file (configures the Blazor app).
* **TodoApp.csproj** → project definition.

Blazor pages (.razor) are like HTML + C#. Example:

@page "/hello" <h3>Hello Page</h3> <p>Hello from Blazor! The time is: @DateTime.Now</p>

Save this as Hello.razor in the **Pages/** folder. Add @page "/hello" at the top.  
Now run again (dotnet run) and navigate to /hello. Boom 💥 — a new page.

* **2.4 Add a placeholder ToDo page**

Let’s create the future home of our ToDo list.

In **Pages/Todo.razor**:

@page "/todo" <h3>My ToDo List</h3> <ul> <li>✅ Learn Blazor basics</li> <li>🔲 Connect to a database</li> <li>🔲 Deploy live</li> </ul>

Restart (dotnet run) → go to /todo in the browser. You should see your starter ToDo page.

* **2.5 Add to the navigation menu**

Open **Shared/NavMenu.razor**, and under the existing links add:

<NavLink class="nav-link" href="todo"> <span class="oi oi-list-rich" aria-hidden="true"></span> ToDo </NavLink>

Now you’ll see **ToDo** in the left-hand nav.

* **2.6 Quick recap**
* You scaffolded a Blazor WebAssembly app.
* You ran it locally (dotnet run).
* You created your own page (/todo).
* You added a menu link.

Your project now has a place for the ToDo app UI.

✅ Next step (Chapter 3): We’ll **make a real ToDo model + a local mock service**. This way you can add/remove items without a database yet — perfect for testing UI and Copilot.

Do you want me to also explain *how to commit this initial project into GitHub* now, or wait until Chapter 7 (the dedicated GitHub/Copilot workflow chapter)?

* **Du sa:**

Now run again (`dotnet run`) and navigate to `/hello`. Boom 💥 — a new page.

I get this error at http://localhost:5195/hello Sorry, there's nothing at this address.

* **ChatGPT sa:**

Ah, good catch 👍 That usually happens because Blazor doesn’t automatically know about your new page unless:

1. You saved the file in the right place, and
2. You restarted (or hot-reloaded) the app.

Let’s double-check together:

* **✅ Checklist to fix**/hello

1. **File location**
   * The file should be in:
   * TodoApp/Pages/Hello.razor
   * The very first line in that file must be:

@page "/hello"

1. **Restart the app**  
   After saving, go back to terminal and run:

dotnet run

Then open the new URL it shows (sometimes it changes ports, e.g. http://localhost:5195 or http://localhost:5173).  
Now try http://localhost:5195/hello.

1. **Case sensitivity**  
   Make sure you typed /hello (all lowercase). The route in @page is case-sensitive.
   * @page "/hello" → works at /hello
   * @page "/Hello" → works at /Hello
2. **Still not working?**  
   Check if the project actually rebuilt. Sometimes VS Code hot reload doesn’t pick up new files unless you restart.

* **🔍 Quick test**

Can you paste your full Hello.razor file here so I can verify the contents?

If it’s all good, /hello should definitely load after restarting.

👉 Do you want me to also show you how to make sure it shows up in the **navigation menu** so you don’t have to type /hello manually?

* **Du sa:**

Restart (`dotnet run`)

How do I Restart (dotnet run)?

* **ChatGPT sa:**

Good question — let’s make it super clear.

When you run dotnet run, the app keeps running in your terminal until you stop it. If you add a new page (like Hello.razor), Blazor won’t see it until you restart.

Here’s how to restart:

* **1. Stop the app**
* In the terminal where dotnet run is running, press:  
  **Ctrl + C**  
  (this tells .NET to stop the server).

You should see something like:

Application is shutting down...

* **2. Run it again**

In the same folder (TodoApp), type:

dotnet run

You’ll see something like:

Now listening on: http://localhost:5195

Now listening on: https://localhost:7195

* **3. Open your browser**
* Go to the port shown (e.g., http://localhost:5195/hello).
* If you added the @page "/hello" page correctly, it should now load.

👉 Quick check: do you see “My ToDo List” at /todo already? If /todo works but /hello doesn’t, then the file content or location might need fixing.

Do you want me to show you **exactly where to put**Hello.razor**in VS Code’s Explorer**, with the folder tree?

* **Du sa:**

Open \*\*Shared/NavMenu.razor\*\*, and under the existing links add:

This file "NavMenu.razor" is in the folder "Layout" and the folder Shared is missing.

* **ChatGPT sa:**

Ahh 👍 you’ve spotted a difference:  
That’s because the **Blazor WebAssembly template** got slightly reorganized in .NET 8.

* In older versions, NavMenu.razor lived under **Shared/**.
* In the newer template, it’s under **Layout/**.
* The **Shared/** folder might be missing completely — that’s normal in newer projects.

So don’t worry — your template is correct. 🎉

* **✅ What to do**
* Instead of Shared/NavMenu.razor, just edit:
* TodoApp/Components/Layout/NavMenu.razor

(you might see the full path as Components/Layout/NavMenu.razor depending on the exact SDK).

* Inside that file, you’ll see existing <NavLink>s for Home, Counter, Fetch data.
* Just add your new ToDo link **under them**. Example:

<NavLink class="nav-link" href="todo"> <span class="oi oi-list-rich" aria-hidden="true"></span> ToDo </NavLink>

* **🔍 Why the docs differ**

Microsoft adjusted the Blazor folder structure in newer SDKs to make components more modular. So if tutorials mention **Shared/**, just mentally swap it for **Components/** (and specifically Components/Layout for the nav).

👉 Question for you: Do you want me to rewrite all future steps using the **new folder structure** (so it matches exactly what you see in your project)? That way you won’t run into these “missing folder” confusions again.

* **Du sa:**

It is no folder named Components. The path is TodoApp/Layout/NavMenu.razor

* **ChatGPT sa:**

Got it ✅ Thanks for clarifying!  
You’re using the **simpler Blazor WebAssembly project template**, which puts things directly like this:

TodoApp/

├── wwwroot/

├── Pages/

├── Layout/

│ └── NavMenu.razor

├── Program.cs

└── TodoApp.csproj

So in **your** case:

* There is **no**Shared/ folder.
* The navigation is in Layout/NavMenu.razor.

That’s perfectly fine — you just need to edit that file instead of the Shared one.

* **✅ Add your ToDo link**

Open Layout/NavMenu.razor and add this snippet under the other <NavLink>s:

<NavLink class="nav-link" href="todo"> <span class="oi oi-list-rich" aria-hidden="true"></span> ToDo </NavLink>

Now when you restart (Ctrl+C → dotnet run), you should see **ToDo** in the sidebar menu. Clicking it should take you to /todo.

* **ℹ️ Why your project looks like this**
* Some tutorials use older templates (with Shared/).
* The .NET SDK you installed gave you a **newer Blazor template**, which has a flatter structure (Layout/ instead of Shared/).

So: everything is working as expected, just slightly different organization. 👍

👉 Do you want me to adapt all the upcoming chapters to **your exact folder layout** (so you never get these “missing folder” surprises again)?

* **Du sa:**

Yes

* **ChatGPT sa:**

Perfect ✅ I’ll keep everything aligned with **your structure** (Pages/ + Layout/), so no more surprises with missing folders.

So far, you’ve got:

* Todo.razor inside **Pages/**
* NavMenu.razor inside **Layout/** updated with a **ToDo link**
* Project runs at http://localhost:xxxx/todo 🎉
* **Coming up next**

In **Chapter 3** we’ll:

* Create a **ToDo model** (TodoItem.cs).
* Build a **mock data service** (stores ToDos in memory).
* Hook it up to the Todo.razor page so you can **add/remove items locally** (no database yet).

That way, you’ll see the app come alive before we connect Supabase.

👉 Ready for me to walk you through **Chapter 3** with the code?

* **Du sa:**

Next

* **ChatGPT sa:**

Awesome 😎 Let’s bring your ToDo page to life!

* **Chapter 3 — Make a Local ToDo Prototype (Model + Mock Service)**

Goal: create a TodoItem model, store items in memory, and display/add/remove them in your Todo.razor page **without a database yet**. This makes UI development fast and safe.

* **3.1 Create the ToDo model**

1. In your project root, create a new folder called **Models**:

TodoApp/Models

1. Inside **Models**, create a file TodoItem.cs:

namespace TodoApp.Models { public class TodoItem { public int Id { get; set; } *// unique identifier* public string Title { get; set; } *// task text* public bool IsDone { get; set; } *// completed flag* } }

✅ This is your simple ToDo class.

* **3.2 Create a mock service**

We’ll store items in memory while the app is running.

1. Create a new folder **Services**:

TodoApp/Services

1. Inside **Services**, create TodoService.cs:

using TodoApp.Models; namespace TodoApp.Services { public class TodoService { private readonly List<TodoItem> \_todos = new List<TodoItem>(); private int \_nextId = 1; public List<TodoItem> GetAll() => \_todos; public void Add(string title) { \_todos.Add(new TodoItem { Id = \_nextId++, Title = title, IsDone = false }); } public void ToggleDone(int id) { var todo = \_todos.FirstOrDefault(t => t.Id == id); if (todo != null) todo.IsDone = !todo.IsDone; } public void Remove(int id) { var todo = \_todos.FirstOrDefault(t => t.Id == id); if (todo != null) \_todos.Remove(todo); } } }

* **3.3 Register the service in Blazor**

Open Program.cs and add the service:

using TodoApp.Services; var builder = WebApplication.CreateBuilder(args); builder.Services.AddRazorPages(); builder.Services.AddServerSideBlazor(); *// required* builder.Services.AddSingleton<TodoService>(); *// register our mock service* var app = builder.Build(); *// ... existing code*

Using AddSingleton keeps your ToDo list in memory while the app is running.

* **3.4 Update**Todo.razor**to use the service**

Replace your existing placeholder content with this:

@page "/todo" @using TodoApp.Models @inject TodoService TodoService <h3>My ToDo List</h3> <input @bind="newTodo" placeholder="Enter new task" /> <button @onclick="AddTodo">Add</button> <ul> @foreach (var todo in TodoService.GetAll()) { <li> <input type="checkbox" @bind="todo.IsDone" @onchange="() => ToggleTodo(todo.Id)" /> <span style="text-decoration:@(todo.IsDone ? "line-through" : "none")">@todo.Title</span> <button @onclick="() => RemoveTodo(todo.Id)">❌</button> </li> } </ul> @code { private string newTodo = ""; private void AddTodo() { if (!string.IsNullOrWhiteSpace(newTodo)) { TodoService.Add(newTodo); newTodo = ""; } } private void ToggleTodo(int id) => TodoService.ToggleDone(id); private void RemoveTodo(int id) => TodoService.Remove(id); }

* **3.5 Run and test**

1. Restart the app:

Ctrl + C dotnet run

1. Open http://localhost:5195/todo (or your actual port).
2. You should be able to:

* Add tasks
* Mark tasks done/undone
* Remove tasks

All in memory — no database yet.

* **✅ Recap**
* TodoItem.cs → defines your data.
* TodoService.cs → stores and manipulates ToDos.
* Todo.razor → UI binds to the service.

You now have a **fully functional local ToDo app**! This is perfect for testing UI and experimenting with Copilot before connecting a real database.

Next step (Chapter 4): we’ll create a **free live database with Supabase** and prepare to replace this mock service with real CRUD operations.

Do you want me to move on to Chapter 4?

* **Du sa:**

Open `Program.cs` and add the service:

The code is failing based on the code you are suggestion, so I need help. Here is the code in my existing Program.cs file: using Microsoft.AspNetCore.Components.Web; using Microsoft.AspNetCore.Components.WebAssembly.Hosting; using TodoApp; var builder = WebAssemblyHostBuilder.CreateDefault(args); builder.RootComponents.Add<App>("#app"); builder.RootComponents.Add<HeadOutlet>("head::after"); builder.Services.AddScoped(sp => new HttpClient { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) }); await builder.Build().RunAsync();

* **ChatGPT sa:**

Tenkte i 14s

Nice — thanks for pasting your Program.cs. That helped. The problem is just that my earlier instruction (meant for a server-side Blazor app) suggested AddServerSideBlazor() — which doesn't belong in a **Blazor WebAssembly** project. For your WASM app you should register the TodoService on the builder.Services used by WebAssemblyHostBuilder.

Below is a corrected, ready-to-use Program.cs for your Blazor WebAssembly project plus a few small fixes you should make to TodoService and Todo.razor.

* **1) Replace your**Program.cs**with this**

using Microsoft.AspNetCore.Components.Web; using Microsoft.AspNetCore.Components.WebAssembly.Hosting; using TodoApp; using TodoApp.Services; *// <--- add this* var builder = WebAssemblyHostBuilder.CreateDefault(args); builder.RootComponents.Add<App>("#app"); builder.RootComponents.Add<HeadOutlet>("head::after"); *// existing HttpClient registration* builder.Services.AddScoped(sp => new HttpClient { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) }); *// register the in-memory Todo service so components can inject it* builder.Services.AddSingleton<TodoService>(); await builder.Build().RunAsync();

Key points:

* Use builder.Services.AddSingleton<TodoService>() (or AddScoped — both work in WASM, but AddSingleton is common for app-wide in-memory state).
* **Do not** call AddServerSideBlazor() in a WebAssembly app.
* **2) Make sure**TodoService**compiles (example)**

Put this in TodoApp/Services/TodoService.cs:

using System.Collections.Generic; using System.Linq; using TodoApp.Models; namespace TodoApp.Services { public class TodoService { private readonly List<TodoItem> \_todos = new List<TodoItem>(); private int \_nextId = 1; public List<TodoItem> GetAll() => \_todos; public void Add(string title) { \_todos.Add(new TodoItem { Id = \_nextId++, Title = title, IsDone = false }); } public void ToggleDone(int id) { var todo = \_todos.FirstOrDefault(t => t.Id == id); if (todo != null) todo.IsDone = !todo.IsDone; } public void Remove(int id) { var todo = \_todos.FirstOrDefault(t => t.Id == id); if (todo != null) \_todos.Remove(todo); } } }

And TodoApp/Models/TodoItem.cs (if you don't already have it):

namespace TodoApp.Models { public class TodoItem { public int Id { get; set; } public string Title { get; set; } = ""; public bool IsDone { get; set; } } }

* **3) Update your**Todo.razor**so the component knows the service type**

At the top of Pages/Todo.razor add the services namespace (or add it to \_Imports.razor):

@page "/todo" @using TodoApp.Models @using TodoApp.Services @inject TodoService TodoService

(If you prefer not to add the @using, you can inject with the fully qualified name:  
@inject TodoApp.Services.TodoService TodoService)

* **4) Restart the app**

1. Stop the running app in the terminal with **Ctrl + C**.
2. Restart in the project folder:

dotnet run

1. Open the URL shown (e.g. http://localhost:5195/todo).

* **Troubleshooting quick checklist**
* If you get CS0246 or "type or namespace not found" for TodoService, check the namespace line in TodoService.cs matches TodoApp.Services.
* If the app doesn't pick up changes, make sure you restarted dotnet run.
* If injection fails at runtime, double-check you registered the service in Program.cs **before** Build().

If you run these steps and still see an error, paste the exact error text here and I’ll fix it right away.

* **Du sa:**

I am ready for the next chapter

* **ChatGPT sa:**

Perfect 🎉 That means your in-memory ToDo app is alive and well. Time to make it **real** with a live database.

* **Chapter 4 — Create a Free Live Database (Supabase)**

Goal: set up a free hosted database where your ToDos can be stored and retrieved via an API. We’ll use **Supabase** (free tier, PostgreSQL + REST + auth).

* **4.1 Create a Supabase account**

1. Go to https://supabase.com.
2. Click **Start your project** and sign up (you can use GitHub or email).
3. Once inside, click **New project**.

* **4.2 Create your project**
* **Organization**: Supabase will create one for you (you can keep default).
* **Name**: e.g. todoapp.
* **Database password**: set a secure one (save it somewhere — you’ll need it if you connect directly).
* **Region**: choose something near you.
* **Pricing plan**: select **Free**.

Wait ~1–2 minutes while it provisions. ✅

* **4.3 Create the**todos**table**

1. In the left menu, go to **Table Editor** → **New table**.
2. Table name: todos.
3. Add columns:
   * id → type: int8, set as **Primary key**, enable **Identity** (auto-increment).
   * title → type: text, not null.
   * is\_done → type: bool, default = false.
   * (optional) created\_at → type: timestamp, default = now().

Click **Save**.

Now you have a live table like this:

| **id** | **title** | **is\_done** | **created\_at** |
| --- | --- | --- | --- |
| 1 | Learn Blazor | false | 2025-09-12 12:00… |

* **4.4 Get your API URL and Key**

1. In the left menu, go to **Project settings → API**.
2. Copy the **Project URL** (looks like https://xxxx.supabase.co).
3. Copy the **anon public key**.

We’ll use these in your Blazor app to talk to Supabase.

⚠️ **Keep the anon key safe** — it’s public but still should not be shared carelessly.

* **4.5 Test the API (optional but recommended)**

Install the **REST Client** extension in VS Code.  
Create a new file test.http with this:

GET https://YOUR\_PROJECT\_URL/rest/v1/todos apikey: YOUR\_ANON\_KEY

Click “Send Request” above the GET line.

* If it returns [], your setup is correct.
* If it errors, double-check URL and key.
* **✅ Recap**
* You created a Supabase project (Postgres database).
* You added a todos table.
* You obtained the API URL + anon key.
* You tested the REST endpoint.
* **Coming up (Chapter 5)**

We’ll replace your mock TodoService with a **Supabase service** that uses HttpClient in Blazor to call the REST API for Create/Read/Update/Delete.

👉 Do you want me to show you a **safe way to store your Supabase URL + key** in your Blazor project before we connect it, or just keep it simple and paste them directly into the code first?

Top of Form

Bottom of Form

ChatGPT kan gjøre feil. Sjekk viktig informasjon.