## **Session**

Eine **Session** (Sitzung) dient dazu, **Daten benutzerspezifisch zwischen HTTP-Anfragen zu speichern** – z. B. Benutzername, Warenkorb, Spracheinstellung usw.

HTTP ist **zustandslos**, aber mit Session können wir z. B. speichern:

Wer ist der Benutzer?

Was hat er bisher ausgewählt?

Welche Einstellungen hat er getroffen?

**Session in ASP.NET Core aktivieren**

**1. In Program.cs aktivieren**

var builder = WebApplication.CreateBuilder(args);

// Sessiondienste konfigurieren

builder.Services.AddDistributedMemoryCache(); // Speicher im RAM

builder.Services.AddSession(options =>

{

options.IdleTimeout = TimeSpan.FromMinutes(30); // Inaktivitätszeit

options.Cookie.HttpOnly = true; // Cookie nicht per JavaScript zugreifbar

options.Cookie.IsEssential = true; // Wichtig für DSGVO

});

var app = builder.Build();

// Middleware aktivieren (Reihenfolge ist wichtig!)

app.UseSession();

app.UseRouting();

app.UseAuthorization();

app.MapControllers();

app.Run();

**2. Session-Daten speichern und abrufen**

public class HomeController : Controller

{

public IActionResult Speichern()

{

HttpContext.Session.SetString("Name", "Max Mustermann");

return Content("Name gespeichert!");

}

public IActionResult Anzeigen()

{

string name = HttpContext.Session.GetString("Name") ?? "unbekannt";

return Content($"Hallo {name}!");

}

}

**3. Komplexe Objekte speichern (z. B. JSON)**

public class Benutzer

{

public int Id { get; set; }

public string Name { get; set; }

}

// Objekt speichern

var benutzer = new Benutzer { Id = 1, Name = "Anna" };

string json = JsonSerializer.Serialize(benutzer);

HttpContext.Session.SetString("Benutzer", json);

// Objekt abrufen

string jsonString = HttpContext.Session.GetString("Benutzer");

var benutzerObjekt = JsonSerializer.Deserialize<Benutzer>(jsonString);

**Was passiert im Hintergrund?**

Es wird ein Cookie (.AspNetCore.Session) an den Browser gesendet.

Der eigentliche Inhalt der Session liegt **serverseitig** im RAM (oder Redis/SQL, wenn gewünscht).

Die Session ist **benutzerspezifisch**.

**Standardverhalten von ASP.NET Core Session**

* **Session Timeout** (Idle Timeout) ist die Zeitspanne, die die Session *inaktiv* sein darf, bevor sie verfällt.
* **Idle Timeout** wird standardmäßig auf 20 Minuten gesetzt (kann aber konfiguriert werden).
* Bei jedem Request, bei dem die Session verwendet wird (also z.B. wenn HttpContext.Session gelesen oder geschrieben wird), **wird der Timeout zurückgesetzt**.
* Das heißt: Ja, die Session wird bei jeder Aktion verlängert, solange die Session im Request aktiv genutzt wird.

**Details**

* Wenn der Client also z.B. alle 10 Minuten eine Aktion macht, die Session benutzt, dann läuft die Session nie ab.
* Wenn der Client hingegen z.B. 30 Minuten inaktiv bleibt, ohne die Session zu verwenden, wird die Session beim nächsten Zugriff abgelaufen sein.

**Wie hole ich aber aus der Session ein Objekt?**

Um in ASP.NET Core ein Objekt aus der Session zu holen, müssen Sie Folgendes beachten:

1. In ASP.NET Core **können nur Strings oder Byte-Arrays** direkt in die Session geschrieben werden.
2. Um ein komplexes Objekt zu speichern oder zu lesen, verwenden Sie normalerweise **JSON-Serialisierung**.

Beispiel:

Angenommen, Sie haben vorher ein Objekt als JSON gespeichert:

var person = new Person { Name = "Max", Alter = 30 };

HttpContext.Session.SetString("meinePerson", JsonSerializer.Serialize(person));

Dann können Sie das Objekt so wieder aus der Session holen:

var json = HttpContext.Session.GetString("meinePerson");

if (json != null)

{

var person = JsonSerializer.Deserialize<Person>(json);

// person ist jetzt wieder ein Person-Objekt

}

**Übungsaufgabe**

**Aufgabe**:  
Erstellen Sie einen Controller mit zwei Methoden:

1. Speichern(string name) – speichert den Namen des Benutzers in der Session.
2. Anzeigen() – gibt „Willkommen, [Name]“ zurück.

**Zusatz**: Speichern Sie ein ganzes Objekt vom Typ Benutzer mit Vor- und Nachname.

**Unterschiede ASP.NET MVC & ASP.NET Core MVC**

Der Umgang mit **Session** in **ASP.NET MVC (Framework)** und **ASP.NET Core MVC** ist ähnlich in der Funktion, aber **technisch unterschiedlich in der Konfiguration und Architektur**.

Hier die wichtigsten **Unterschiede zwischen ASP.NET MVC (klassisch, .NET Framework)** und **ASP.NET Core MVC** im Bereich **Session**:

**1. Framework vs. Middleware**

|  |  |  |
| --- | --- | --- |
| Merkmal | ASP.NET MVC (.NET Framework) | ASP.NET Core MVC |
| Architektur | basiert auf System.Web und HTTP-Module | basiert auf Middleware-Pipeline |
| Session-Verwaltung | automatisch aktiviert | muss explizit in Program.cs aktiviert werden |
| Lebenszyklus | via Global.asax, Session\_Start etc. | kein Session\_Start – alles Middleware-basiert |

**3. Session-Storage-Typen**

|  |  |  |
| --- | --- | --- |
| Option | ASP.NET MVC | ASP.NET Core |
| In-Memory | Ja (InProc) | Ja (AddDistributedMemoryCache()) |
| SQL Server | Ja | Ja (AddDistributedSqlServerCache()) |
| Redis | Drittanbieter | Ja (AddStackExchangeRedisCache()) |
| StateServer | Ja | ❌ Nicht verfügbar |

**5. Lebensdauer / Verhalten**

|  |  |  |
| --- | --- | --- |
| Aspekt | ASP.NET MVC | ASP.NET Core |
| Timeout | In Web.config konfigurierbar | In Middleware konfigurierbar |
| Session\_Start | Gibt es (Global.asax) | Gibt es **nicht** |
| Events | Session\_Start, End möglich | Keine Session-Events |

|  |  |  |
| --- | --- | --- |
| Bereich | ASP.NET MVC (klassisch) | ASP.NET Core MVC |
| Einfachheit | Session sofort verfügbar | Session muss manuell aktiviert werden |
| Konfiguration | Web.config | Program.cs / Middleware |
| Zugriff | Session["Key"] | HttpContext.Session.SetString(...) |
| Flexibilität | Eher starr | Sehr flexibel (Redis, SQL, etc.) |
| StateServer möglich? | Ja | Nein |
| Cross-Plattform? | Nein (.NET Framework) | Ja (ASP.NET Core) |

<https://learn.microsoft.com/de-de/aspnet/core/fundamentals/app-state?view=aspnetcore-9.0>