Real-Time Chat App — Step-by-Step Guide (from zero)

> This guide assumes **Windows + Visual Studio 2022**. Follow the checklist in order. Each task has copy-paste commands or exact UI clicks.

✅Phase 0 — Install & Prepare (once)

1) **Visual Studio 2022** - Launch the **Visual Studio Installer** → *Modify* your VS installation. - Workloads to select: - **ASP.NET and web development** - **Data storage and processing** (for EF tooling) - Optional: **.NET desktop development** - After install, open **Developer PowerShell for VS** and run: `dotnet --info` (just to check .NET SDK is ready).

2) **Node.js LTS** - Install Node.js LTS (v18 or v20). After install: `node -v` and `npm -v`.

3) **Git** - Install Git. After install: `git --version`.

4) **Database** - Easiest: **SQL Server Express** + **SSMS** (SQL Server Management Studio). Alternative (simpler for dev): **SQLite** (no separate service required).

5) **Postman** - Install Postman for API testing.

Create a root folder where you'll work: `C:\Projects\RealTimeChatApp`

✅Phase 1 — Create the Solution (backend + frontend + docs)

Folder layout (target):

```
RealTimeChatApp/
  backend/
  frontend/
  deployment/
  README.md
```

1A) Backend project (ASP.NET Core Web API, .NET 8)

- Open **Visual Studio 2022** → *Create a new project* → **ASP.NET Core Web API** → Next.
- Project name: `RealTimeChatApp.Backend`
  Location: `C:\Projects\RealTimeChatApp\backend`

Framework: **.NET 8.0**
Authentication: **None** (we'll add JWT manually with ASP.NET Identity).
Check **Use controllers**.

Install NuGet packages (Right-click project → *Manage NuGet Packages* → *Browse*): - `Microsoft.EntityFrameworkCore.SqlServer` (or `Microsoft.EntityFrameworkCore.Sqlite` if you choose SQLite) - `Microsoft.EntityFrameworkCore.Tools` - `Microsoft.AspNetCore.Identity.EntityFrameworkCore` - `Microsoft.AspNetCore.Authentication.JwtBearer` - `Microsoft.AspNetCore.SignalR` - `Swashbuckle.AspNetCore` (Swagger)

1B) Frontend project (React + Vite)

- Open **Terminal** in `C:\Projects\RealTimeChatApp` and run:

```
npm create vite@latest frontend -- --template react
cd frontend
npm install
npm install axios @microsoft/signalr react-router-dom jwt-decode
```

Create a file `frontend/.env` with:

```
VITE_API_BASE_URL=http://localhost:5000
VITE_SIGNALR_URL=http://localhost:5000/hubs/chat
```

(We'll match ports in Phase 2.)

✅Phase 2 — Configure Backend (Identity + EF Core + JWT + SignalR)

2A) appsettings.json

In `backend/appsettings.json` add a **ConnectionStrings** and **JWT** section:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=localhost;Database=ChatAppDb;Trusted_Connection=True;TrustServerCertificate=True;"
  },
  "Jwt": {
    "Issuer": "ChatApp",
    "Audience": "ChatAppClient",
    "Key": "REPLACE_WITH_A_LONG_RANDOM_SECRET_KEY"
  },
  "Logging": {
    "LogLevel": { "Default": "Information", "Microsoft.AspNetCore": "Warning" }
  },
  "AllowedHosts": "*"
}
```

(If using SQLite instead of SQL Server, use `"Data Source=chatapp.db"`*.)*

2B) Create Models

Create folder **Models** and add:

`ApplicationUser.cs`

```
using Microsoft.AspNetCore.Identity;

namespace RealTimeChatApp.Backend.Models
{
    public class ApplicationUser : IdentityUser
    {
        public string? DisplayName { get; set; }
        public string Status { get; set; } = "Available"; // Available | Busy |
Offline
        public DateTime LastSeenUtc { get; set; } = DateTime.UtcNow;
    }
}
```

`Message.cs`

```
namespace RealTimeChatApp.Backend.Models
{
    public class Message
    {
        public int Id { get; set; }
        public string SenderId { get; set; } = default!;
        public string? ReceiverId { get; set; } // for private chat
        public int? GroupId { get; set; }        // for group chat
        public string Content { get; set; } = string.Empty;
        public string? AttachmentUrl { get; set; }
        public DateTime SentAtUtc { get; set; } = DateTime.UtcNow;
    }
}
```

`Group.cs`

```
namespace RealTimeChatApp.Backend.Models
{
    public class Group
    {
        public int Id { get; set; }
        public string Name { get; set; } = string.Empty;
        public ICollection<GroupMember> Members { get; set; } = new
List<GroupMember>();
    }

    public class GroupMember
    {
        public int Id { get; set; }
        public int GroupId { get; set; }
        public string UserId { get; set; } = default!;
        public string Role { get; set; } = "member"; // member | admin
    }
}
```

2C) DbContext

Create **Data/AppDbContext.cs**

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using RealTimeChatApp.Backend.Models;

namespace RealTimeChatApp.Backend.Data
{
    public class AppDbContext : IdentityDbContext<ApplicationUser>
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options) { }

        public DbSet<Message> Messages => Set<Message>();
        public DbSet<Group> Groups => Set<Group>();
        public DbSet<GroupMember> GroupMembers => Set<GroupMember>();

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
        }

    }
}
```

2D) Program.cs configuration

Open `Program.cs` and replace contents with:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using RealTimeChatApp.Backend.Data;
using RealTimeChatApp.Backend.Models;
using System.Text;

var builder = WebApplication.CreateBuilder(args);

// DB
builder.Services.AddDbContext<AppDbContext>(opt =>

opt.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"))
    // For SQLite
use: .UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection"))
);

// Identity
builder.Services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<AppDbContext>()
    .AddDefaultTokenProviders();

// JWT
var jwt = builder.Configuration.GetSection("Jwt");
var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwt["Key"]!));

builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = jwt["Issuer"],
        ValidAudience = jwt["Audience"],
        IssuerSigningKey = key
    };
    options.Events = new JwtBearerEvents
    {
        OnMessageReceived = context =>
        {
            // Allow SignalR access token via query string
            var accessToken = context.Request.Query["access_token"];
            var path = context.HttpContext.Request.Path;
            if (!string.IsNullOrEmpty(accessToken) && path.StartsWithSegments("/
hubs/chat"))
            {
                context.Token = accessToken;
            }
            return Task.CompletedTask;
        }
    };
});

builder.Services.AddAuthorization();

builder.Services.AddSignalR();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddCors(opt =>
{
    opt.AddPolicy("client", policy =>
        policy.AllowAnyHeader().AllowAnyMethod()
              .AllowCredentials()
              .WithOrigins("http://localhost:5173") // Vite dev server
    );
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseCors("client");
app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();
app.MapHub<ChatHub>("/hubs/chat");

app.Run();

// ChatHub minimal implementation
public class ChatHub : Microsoft.AspNetCore.SignalR.Hub
{
    public async Task SendPrivate(string receiverUserId, string message)
        => await Clients.User(receiverUserId).SendAsync("ReceiveMessage",
Context.UserIdentifier, message, DateTime.UtcNow);

    public async Task SendGroup(string groupName, string message)
        => await Clients.Group(groupName).SendAsync("ReceiveGroupMessage",
Context.UserIdentifier, message, DateTime.UtcNow);

    public async Task Typing(string toUserId)
        => await Clients.User(toUserId).SendAsync("Typing",
Context.UserIdentifier);
}
```

2E) Migrations

- **Tools → NuGet Package Manager → Package Manager Console**

```
Add-Migration InitialCreate
Update-Database
```

✅Phase 3 — Auth Controller (Register/Login/Logout + JWT)

Create `Controllers/AuthController.cs`:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using RealTimeChatApp.Backend.Models;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
```

```
[ApiController]
[Route("api/[controller]")]
public class AuthController : ControllerBase
{
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly SignInManager<ApplicationUser> _signInManager;
    private readonly IConfiguration _config;

    public AuthController(UserManager<ApplicationUser> um,
SignInManager<ApplicationUser> sm, IConfiguration cfg)
    { _userManager = um; _signInManager = sm; _config = cfg; }

    [HttpPost("register")]
    public async Task<IActionResult> Register(RegisterDto dto)
    {
        var user = new ApplicationUser { UserName = dto.Email, Email =
dto.Email, DisplayName = dto.DisplayName };
        var result = await _userManager.CreateAsync(user, dto.Password);
        if (!result.Succeeded) return BadRequest(result.Errors);
        return Ok(new { message = "Registered" });
    }

    [HttpPost("login")]
    public async Task<IActionResult> Login(LoginDto dto)
    {
        var user = await _userManager.FindByEmailAsync(dto.Email);
        if (user == null) return Unauthorized();

        var ok = await _signInManager.CheckPasswordSignInAsync(user,
dto.Password, false);
        if (!ok.Succeeded) return Unauthorized();

        return Ok(new { token = CreateToken(user), user = new { user.Id,
user.Email, user.DisplayName } });
    }

    [Authorize]
    [HttpPost("logout")]
    public IActionResult Logout() => Ok(new { message = "Client should discard
JWT" });

    private string CreateToken(ApplicationUser user)
    {
        var claims = new List<Claim>
        {
            new Claim(JwtRegisteredClaimNames.Sub, user.Id),
            new Claim(JwtRegisteredClaimNames.Email, user.Email ?? ""),
            new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
        };

        var jwt = _config.GetSection("Jwt");
        var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwt["Key"]!));
        var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

        var token = new JwtSecurityToken(
            issuer: jwt["Issuer"], audience: jwt["Audience"],
            claims: claims, expires: DateTime.UtcNow.AddHours(12),
signingCredentials: creds);

        return new JwtSecurityTokenHandler().WriteToken(token);
    }
}

public record RegisterDto(string Email, string Password, string DisplayName);
public record LoginDto(string Email, string Password);
```

✅Phase 4 — Chat + Users + Groups Controllers

Create `Controllers/ChatController.cs`:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using RealTimeChatApp.Backend.Data;
using RealTimeChatApp.Backend.Models;

[ApiController]
[Route("api/[controller]")]
[Authorize]
public class ChatController : ControllerBase
{
    private readonly AppDbContext _db;
    public ChatController(AppDbContext db) { _db = db; }

    [HttpGet("private/{userId}")]
    public async Task<IActionResult> GetPrivate(string userId)
    {
        var me = User.FindFirst("sub")?.Value;
        var msgs = await _db.Messages
            .Where(m => (m.SenderId == me && m.ReceiverId == userId) ||
(m.SenderId == userId && m.ReceiverId == me))
```

```
            .OrderBy(m => m.SentAtUtc)
            .ToListAsync();
        return Ok(msgs);
    }

    public record SendDto(string? ReceiverId, int? GroupId, string Content);

    [HttpPost("send")]
    public async Task<IActionResult> Send([FromBody] SendDto dto)
    {
        var me = User.FindFirst("sub")?.Value!;
        var msg = new Message { SenderId = me, ReceiverId = dto.ReceiverId,
GroupId = dto.GroupId, Content = dto.Content };
        _db.Messages.Add(msg);
        await _db.SaveChangesAsync();
        return Ok(msg);
    }

    [HttpGet("group/{groupId:int}")]
    public async Task<IActionResult> GetGroup(int groupId)
    {
        var msgs = await _db.Messages.Where(m => m.GroupId ==
groupId).OrderBy(m => m.SentAtUtc).ToListAsync();
        return Ok(msgs);
    }
}
```

Create `Controllers/UsersController.cs`:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using RealTimeChatApp.Backend.Models;

[ApiController]
[Route("api/[controller]")]
[Authorize]
public class UsersController : ControllerBase
{
    private readonly UserManager<ApplicationUser> _um;
    public UsersController(UserManager<ApplicationUser> um) { _um = um; }

    [HttpGet("status")]
    public IActionResult Status()
        => Ok(_um.Users.Select(u => new { u.Id, u.Email, u.DisplayName,
u.Status, u.LastSeenUtc }));
```

```
    public record StatusDto(string Status);

    [HttpPut("update-status")]
    public async Task<IActionResult> UpdateStatus([FromBody] StatusDto dto)
    {
        var meId = User.FindFirst("sub")?.Value!;
        var me = await _um.FindByIdAsync(meId);
        if (me == null) return NotFound();
        me.Status = dto.Status;
        me.LastSeenUtc = DateTime.UtcNow;
        await _um.UpdateAsync(me);
        return Ok(new { me.Id, me.Status });
    }
}
```

Create `Controllers/GroupController.cs`:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using RealTimeChatApp.Backend.Data;
using RealTimeChatApp.Backend.Models;

[ApiController]
[Route("api/[controller]")]
[Authorize]
public class GroupController : ControllerBase
{
    private readonly AppDbContext _db;
    public GroupController(AppDbContext db) { _db = db; }

    public record CreateGroupDto(string Name, List<string> MemberUserIds);

    [HttpPost("create")]
    public async Task<IActionResult> Create([FromBody] CreateGroupDto dto)
    {
        var g = new Group { Name = dto.Name };
        _db.Groups.Add(g);
        await _db.SaveChangesAsync();

        foreach (var uid in dto.MemberUserIds.Distinct())
            _db.GroupMembers.Add(new GroupMember { GroupId = g.Id, UserId =
uid, Role = "member" });

        await _db.SaveChangesAsync();
```

```
            return Ok(g);
        }
    }
```

✅Phase 5 — Enable Swagger & Run Backend

In `Program.cs` we already added Swagger. Run backend: - Set project to **RealTimeChatApp.Backend** → *F5*. - Swagger UI should open at `http://localhost:5000/swagger` (port may differ → match in `launchSettings.json` and in frontend `.env`).

Test endpoints in Swagger/Postman: - `POST /api/auth/register` - `POST /api/auth/login` → copy token - Authorize (Swagger lock icon) with `Bearer {token}` - Test chat endpoints.

✅Phase 6 — Frontend (React)

6A) Project wiring

In `frontend/src`, create a simple API helper:

`services/api.js`

```
import axios from "axios";
const api = axios.create({ baseURL: import.meta.env.VITE_API_BASE_URL });
api.interceptors.request.use(cfg => {
  const token = localStorage.getItem("token");
  if (token) cfg.headers.Authorization = `Bearer ${token}`;
  return cfg;
});
export default api;
```

`services/hub.js`

```
import * as signalR from "@microsoft/signalr";
export function createHubConnection() {
  const url = import.meta.env.VITE_SIGNALR_URL;
  const token = localStorage.getItem("token");
  return new signalR.HubConnectionBuilder()
    .withUrl(url, { accessTokenFactory: () => token })
    .withAutomaticReconnect()
    .build();
}
```

6B) Auth pages

`pages/Login.jsx`

```
import { useState } from "react";
import api from "../services/api";

export default function Login({ onLogin }) {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const submit = async e => {
    e.preventDefault();
    const res = await api.post("/api/auth/login", { email, password });
    localStorage.setItem("token", res.data.token);
    onLogin(res.data.user);
  };
  return (
    <form onSubmit={submit} className="p-6 space-y-4">
      <input placeholder="Email" value={email}
onChange={e=>setEmail(e.target.value)} />
      <input placeholder="Password" type="password" value={password}
onChange={e=>setPassword(e.target.value)} />
      <button>Login</button>
    </form>
  );
}
```

6C) Chat page (private + typing)

`pages/Chat.jsx`

```
import { useEffect, useRef, useState } from "react";
import api from "../services/api";
import { createHubConnection } from "../services/hub";

export default function Chat({ peerUserId }) {
  const [messages, setMessages] = useState([]);
  const [text, setText] = useState("");
  const [typing, setTyping] = useState(false);
  const hubRef = useRef(null);
  useEffect(() => {
    api.get(`/api/chat/private/${peerUserId}`).then(r => setMessages(r.data));

    const hub = createHubConnection();
    hub.on("ReceiveMessage", (from, body, at) => {
      setMessages(m => [...m, { senderId: from, content: body, sentAtUtc:
at }]);
    });
    hub.on("Typing", (from) => setTyping(true));
    hub.start();
    hubRef.current = hub;
    return () => { hub.stop(); };
  }, [peerUserId]);

  const send = async () => {
    if (!text.trim()) return;
    await api.post("/api/chat/send", { receiverId: peerUserId, content: text });
    setMessages(m => [...m, { senderId: "me", content: text, sentAtUtc: new
Date().toISOString() }]);
    setText("");
    hubRef.current?.invoke("SendPrivate", peerUserId, text);
  };

  const onType = () => {
    setText(t => t);
    hubRef.current?.invoke("Typing", peerUserId);
  };

  return (
    <div className="p-4">
      <div className="h-80 overflow-auto border">
        {messages.map((m,i) => (
          <div key={i}>{m.senderId}: {m.content}</div>
        ))}
      </div>
      {typing && <div>Typing...</div>}
      <input value={text} onChange={e=>{setText(e.target.value); onType();}} />
      <button onClick={send}>Send</button>
    </div>
  );
}
```

(Group chat is similar but uses `SendGroup` *and* `GET /api/chat/group/{groupId}` *.)*

✅Phase 7 — File Uploads (images/docs)

- Simplest approach: accept multipart to an endpoint `/api/files/upload` that stores to local `/wwwroot/uploads` (dev) then to cloud storage in prod (Azure Blob).
- Save returned `AttachmentUrl` on message.

✅Phase 8 — User Status

- On login: call `PUT /api/users/update-status` with `Available`.
- On window blur/close (frontend `beforeunload`), optionally set `Offline`.
- Server side: update `LastSeenUtc` in a background filter or in hub `OnConnected`/`OnDisconnected`.

✅Phase 9 — Swagger + Postman

- Swagger is already enabled → visit `/swagger` and try each API.
- Export a Postman collection with auth set to *Bearer Token* (paste JWT after login).

✅Phase 10 — Deployment (Azure easiest)

Option A (recommended for beginners): - Backend → **Azure App Service** (Linux) + **Azure SQL Database** + **Azure SignalR Service**. - Frontend → **Azure Static Web Apps** (build `npm run build`, output `dist`). - Configure **CORS** and **appsettings** (JWT Key, SQL connection, SignalR connection if you move to Azure SignalR Service).

Option B (advanced): - Backend on **Azure Functions** with **Azure SignalR Service** bindings.

CI/CD (GitHub Actions) - Add a workflow that builds backend with `dotnet build` and deploys to App Service; another that builds frontend and deploys to Static Web Apps.

✅Phase 11 — Submission Pack

- GitHub repo with structure:

```
backend/
frontend/
deployment/
README.md
```

- README sections: *Prerequisites*, *Local Setup*, *Environment Variables*, *Run Backend*, *Run Frontend*, *API Docs*, *Deployment*, *Troubleshooting*.
- Include Postman collection and screenshots of the running app.

🔕What to do right now

1) Finish **Phase 0** installs.
2) Create backend (Phase 1A) and install packages.
3) Add **appsettings.json**, Models, DbContext, Program.cs (Phase 2).
4) Run migrations.
5) Add Auth + Chat controllers (Phases 3–4).
6) Run backend and test via Swagger.
7) Create frontend (Phase 1B) and wire login + chat (Phase 6).

If anything fails, copy the exact error and we'll fix it.