SignalR Real-time Key-Value Cache — Initial Setup

# 🎯 Goal

Create a real-time cache library that provides key-value access over SignalR and supports:  
- Authentication via BASIC → JWT  
- Client and server validation  
- Configuration with options like `strict`, `logLevel`  
- TTL and schema validation of values

# 🧱 Project Structure

project-root/  
├── server/ # ASP.NET Core SignalR server  
└── client-lib/ # TypeScript client library (SignalR client)

# 🧰 Step 1: Server Setup (`server/`)

## 1. Initialize the project

dotnet new webapi -n server  
cd server  
dotnet add package Microsoft.AspNetCore.SignalR  
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer  
dotnet add package FluentValidation.AspNetCore  
dotnet add package System.IdentityModel.Tokens.Jwt

## 2. Configure `appsettings.json`

{  
 "Jwt": {  
 "Key": "your\_secure\_long\_key\_here\_256bit",  
 "Issuer": "your-app-name",  
 "Audience": "your-app-users"  
 },  
 "Logging": {  
 "LogLevel": {  
 "Default": "Information",  
 "Microsoft.AspNetCore": "Warning"  
 }  
 },  
 "AllowedHosts": "\*"  
}

## 3. Modify `Program.cs`

builder.Services.AddAuthentication(...).AddJwtBearer(...); // see JWT\_Auth\_Guide\_Final.docx  
builder.Services.AddFluentValidationAutoValidation();  
builder.Services.AddValidatorsFromAssemblyContaining<LoginModelValidator>();  
builder.Services.AddSignalR();  
  
app.UseAuthentication();  
app.UseAuthorization();  
  
app.MapControllers();  
app.MapHub<CacheHub>("/hub/cache");

## 4. Add required classes

- `AuthController.cs` – handles `/api/auth/authenticate`, generates JWT  
- `CacheHub.cs` – main SignalR Hub  
- `LoginModel.cs` – login model  
- `LoginModelValidator.cs` – FluentValidation rules

# 🧪 Step 2: Client Setup (`client-lib/`)

## 1. Initialize the project

mkdir client-lib && cd client-lib  
npm init -y  
npm install typescript @microsoft/signalr joi  
npx tsc --init

In `tsconfig.json`, include:

"include": ["src"]

## 2. Add Joi validation schema (`authValidator.ts`)

import Joi from "joi";  
  
export const authSchema = Joi.object({  
 username: Joi.string().min(3).required(),  
 password: Joi.string().min(4).required()  
});

## 3. Create authentication client (`auth-client.ts`)

import { HubConnectionBuilder } from "@microsoft/signalr";  
import { authSchema } from "./authValidator";  
  
const credentials = { username: "user", password: "pass" };  
const { error } = authSchema.validate(credentials);  
if (error) throw new Error(error.details.map(d => d.message).join("; "));  
  
async function authenticate(): Promise<string> {  
 const response = await fetch("http://localhost:5000/api/auth/authenticate", {  
 method: "POST",  
 headers: { "Content-Type": "application/json" },  
 body: JSON.stringify(credentials),  
 });  
  
 const json = await response.json();  
 return json.token;  
}  
  
(async () => {  
 const token = await authenticate();  
 const connection = new HubConnectionBuilder()  
 .withUrl("http://localhost:5000/hub/cache", {  
 accessTokenFactory: () => token,  
 })  
 .build();  
  
 await connection.start();  
 console.log("Connected!");  
})();

# 🔐 Authentication

- Client sends login credentials → receives JWT token  
- JWT includes: `iss`, `exp`, `iat`, `cid` (UUIDv4)  
- Token is signed using SHA-256  
- Every SignalR action (set/get/upsert) must include the token  
- Server validates token on each request

# ✅ At This Point

- ✅ Server accepts SignalR connections and authenticates clients  
- ✅ Client connects, is authenticated, and can call `set`  
- ✅ Input is validated both on client (Joi) and server (FluentValidation)  
- ✅ Test data is cached via SignalR

# 🔧 Next Steps

After initial setup, proceed with implementing other methods (`get`, `upsert`, `config`) and strict mode validation using Joi schemas on the client and validation rules on the server.