# Implementing User Signup Functionality: Step-by-Step Guide

This guide walks you through implementing a complete user signup feature that registers new users and automatically logs them in. You'll learn how to connect frontend Angular components with backend .NET Core APIs.

## Overview

By the end of this tutorial, you will have:

- ☑ A backend API endpoint that accepts signup data
- ☑ User validation and password hashing
- ☑ Database integration for storing new users
- ☑ JWT token generation for automatic login
- ☑ Frontend form that calls the signup API
- ☑ Automatic user authentication after successful signup

---

## Part 1: Backend Implementation

### Step 1: Create the Signup ViewModel

First, create a model to represent the signup data from the frontend.

**File:** `ListKeeper.ApiService/Models/ViewModels/SignupViewModel.cs`

```
using System.ComponentModel.DataAnnotations;

namespace ListKeeperWebApi.WebApi.Models.ViewModels
{
    public class SignupViewModel
    {
        [Required]
        [StringLength(50, MinimumLength = 2)]
        public string FirstName { get; set; } = string.Empty;

        [Required]
        [StringLength(50, MinimumLength = 2)]
        public string LastName { get; set; } = string.Empty;

        [Required]
        [EmailAddress]
        public string Email { get; set; } = string.Empty;

        [Required]
        [StringLength(50, MinimumLength = 3)]
        public string Username { get; set; } = string.Empty;
```

```
        [Required]
        [StringLength(100, MinimumLength = 6)]
        public string Password { get; set; } = string.Empty;

        [Required]
        [Compare("Password")]
        public string ConfirmPassword { get; set; } = string.Empty;

        [Required]
        public bool AgreeToTerms { get; set; }

        // Optional fields
        public bool WantsUpdates { get; set; } = true;
        public string? FavoriteTimHortonsItem { get; set; }
    }
}
```

**Key Learning Points:**

- Data annotations provide both validation and API documentation
- `[Compare("Password")]` ensures password confirmation matches
- Using `string.Empty` instead of null for better null safety

### Step 2: Add Repository Method for Email Lookup

We need to check if a user already exists with the same email.

**File:** `ListKeeper.ApiService/Data/IUserRepository.cs`

Add this method to the interface:

```
Task<User?> GetByEmailAsync(string email);
```

**File:** `ListKeeper.ApiService/Data/UserRepository.cs`

Implement the method:

```
/// <summary>
/// Finds a user by their email address.
/// </summary>
public async Task<User?> GetByEmailAsync(string email)
{
    _logger.LogInformation("Attempting to find user by email: {Email}", email);
    try
    {
        return await _context.Users
            .Where(u => u.Email == email && u.DeletedAt == null)
            .FirstOrDefaultAsync();
```

```
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error retrieving user by email {Email}", email);
            throw;
        }
    }
```

**Key Learning Points:**

- Always check for soft-deleted users (`DeletedAt == null`)
- Use parameterized logging to prevent injection attacks
- Entity Framework automatically parameterizes the SQL query

### Step 3: Add Signup Method to Service Layer

**File:** `ListKeeper.ApiService/Services/IUserService.cs`

Add the interface method:

```
Task<UserViewModel?> SignupAsync(SignupViewModel signupViewModel);
```

**File:** `ListKeeper.ApiService/Services/UserService.cs`

Add the implementation:

```
/// <summary>
/// Registers a new user and returns their information if successful.
/// </summary>
public async Task<UserViewModel?> SignupAsync(SignupViewModel signupViewModel)
{
    if (signupViewModel == null) return null;

    // Check if user already exists
    var existingUser = await _repo.GetByEmailAsync(signupViewModel.Email);
    if (existingUser != null)
    {
        _logger.LogWarning("Signup attempt with existing email: {Email}",
signupViewModel.Email);
        return null; // User already exists
    }

    // Create new user from signup data
    var userViewModel = new UserViewModel
    {
        Email = signupViewModel.Email,
        Username = signupViewModel.Username,
        Firstname = signupViewModel.FirstName,
        Lastname = signupViewModel.LastName,
```

```
            Password = signupViewModel.Password,
            Role = "User", // Default role for new signups
            Phone = string.Empty
        };

        // Use existing CreateUserAsync method which handles password hashing
        var createdUser = await CreateUserAsync(userViewModel);

        if (createdUser != null)
        {
            _logger.LogInformation("New user successfully created: {Email}",
    signupViewModel.Email);
        }

        return createdUser;
    }
```

**Key Learning Points:**

- Always validate input parameters
- Check for existing users to prevent duplicates
- Reuse existing methods (like `CreateUserAsync`) for consistency
- Default new users to "User" role for security

### Step 4: Create the Signup API Endpoint

**File:** `ListKeeper.ApiService/EndPoints/UserEndpoints.cs`

Add the endpoint mapping:

```
group.MapPost("/Signup", Signup)
     .WithName("Signup")
     .WithDescription("Registers a new user and returns a token")
     .AllowAnonymous();
```

Add the handler method:

```
private static async Task<IResult> Signup(
    [FromBody] SignupViewModel model,
    [FromServices] IUserService userService,
    [FromServices] IConfiguration config,
    [FromServices] ILoggerFactory loggerFactory)
{
    var logger = loggerFactory.CreateLogger("Signup");
    try
    {
        if (model == null)
        {
            return Results.BadRequest("Signup data is required");
```

```
        }

        logger.LogInformation("Signup attempt for email: {Email}", model.Email);

        var user = await userService.SignupAsync(model);
        if (user == null)
        {
            return Results.BadRequest("User already exists or signup failed");
        }

        // Generate JWT token for immediate login after signup
        var token = GenerateJwtToken(user, config);

        // Return user info with token (same format as authenticate endpoint)
        return Results.Ok(new
        {
            user = new
            {
                id = user.Id,
                email = user.Email,
                username = user.Username,
                firstname = user.Firstname,
                lastname = user.Lastname,
                role = user.Role
            },
            token = token
        });
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "Error during signup for email: {Email}",
model?.Email);
        return Results.Problem("An error occurred during signup", statusCode:
(int)HttpStatusCode.InternalServerError);
    }
}
```

**Key Learning Points:**

- Use `[FromBody]` to bind JSON request data to C# objects
- `[FromServices]` injects dependencies from the DI container
- Return consistent response format with existing endpoints
- Generate JWT token immediately for automatic login
- Always include proper error handling and logging

---

# Part 2: Frontend Implementation

## Step 5: Add Signup Method to User Service

**File:** `ListKeeper.Web/src/app/services/user.service.ts`

Add this method to the UserService class:

```
signup(signupData: any): Observable<any> {
  return this.http.post<any>(`${this.baseApiUrl}/users/signup`, signupData)
    .pipe(tap(response => {
      if (response.user && response.token) {
        // Store user and token for automatic login after signup
        const user: User = {
          ...response.user,
          token: response.token
        };
        localStorage.setItem('user', JSON.stringify(user));
        this.currentUserSubject.next(user);
      }
    }));
}
```

**Key Learning Points:**

- Use `tap` operator to perform side effects (like storing user data)
- Store both user info and JWT token for authentication
- Update the `currentUserSubject` to reflect the logged-in state
- Spread operator (`...`) copies all properties from response.user

## Step 6: Update the Signup Component

**File:** `ListKeeper.Web/src/app/components/users/signup/signup.component.ts`

Add the UserService import:

```
import { UserService } from '../../../services/user.service';
```

Inject the service in the constructor:

```
constructor(
  private fb: FormBuilder,
  private router: Router,
  private userService: UserService
) {
```

Add error handling property:

```
errorMessage = '';
```

Replace the `onSubmit()` method:

```
onSubmit() {
  if (this.signupForm.valid) {
    this.isSubmitting = true;
    this.errorMessage = '';

    // Prepare signup data
    const signupData = {
      firstName: this.signupForm.value.firstName,
      lastName: this.signupForm.value.lastName,
      email: this.signupForm.value.email,
      username: this.signupForm.value.username,
      password: this.signupForm.value.password,
      confirmPassword: this.signupForm.value.confirmPassword,
      agreeToTerms: this.signupForm.value.agreeToTerms,
      wantsUpdates: this.signupForm.value.wantsUpdates,
      favoriteTimHortonsItem: this.signupForm.value.favoriteTimHortonsItem
    };

    // Call the signup service
    this.userService.signup(signupData).subscribe({
      next: (response) => {
        this.isSubmitting = false;
        this.showSuccessMessage = true;

        // Auto-redirect after showing success message
        setTimeout(() => {
          this.router.navigate(['/notes']);
        }, 2000);
      },
      error: (error) => {
        this.isSubmitting = false;
        console.error('Signup error:', error);
        this.errorMessage = error.error?.message || 'Signup failed. Please try
again.';
      }
    });
  } else {
    // Mark all fields as touched to show validation errors
    Object.keys(this.signupForm.controls).forEach(key => {
      this.signupForm.get(key)?.markAsTouched();
    });
  }
}
```

**Key Learning Points:**

- Extract form values into a clean data object
- Use the new `subscribe()` syntax with `next` and `error` callbacks
- Reset error messages on each submission attempt
- Provide fallback error messages for better UX
- Mark form fields as touched to trigger validation display

### Step 7: Update the HTML Template (Optional)

**File:** `ListKeeper.Web/src/app/components/users/signup/signup.component.html`

Add error message display near the submit button:

```
<!-- Add this before the submit button -->
<div *ngIf="errorMessage" class="alert alert-danger">
  {{ errorMessage }}
</div>
```

---

## Part 3: Testing Your Implementation

### Step 8: Test the Complete Flow

1. **Start the Backend:**

   ```
   cd ListKeeper.ApiService
   dotnet run
   ```

2. **Start the Frontend:**

   ```
   cd ListKeeper.Web
   ng serve
   ```

3. **Test the Signup:**
   - Navigate to the signup page
   - Fill out the form with valid data
   - Submit and verify:
     - User is created in the database
     - JWT token is returned
     - User is automatically logged in
     - Redirect to notes page works

### Step 9: Common Issues and Solutions

**Issue:** "User already exists" error

- **Solution:** Check your email uniqueness validation
- **Debug:** Look at the `GetByEmailAsync` method logs

**Issue:** Password not hashing

- **Solution:** Verify the `CreateUserAsync` method is called

- **Debug:** Check if the `HashPassword` method is working

**Issue:** Token not generated

- **Solution:** Ensure JWT settings are configured in `appsettings.json`
- **Debug:** Check the `GenerateJwtToken` method

**Issue:** Frontend not redirecting

- **Solution:** Verify the `currentUserSubject` is updated
- **Debug:** Check browser localStorage for the user data

---

## Key Concepts Learned

### Backend Concepts:

1. **Layered Architecture:** Controller → Service → Repository pattern
2. **Data Validation:** Using DataAnnotations for input validation
3. **Password Security:** Hashing passwords before storage
4. **JWT Authentication:** Generating tokens for stateless auth
5. **Error Handling:** Proper logging and exception management
6. **Dependency Injection:** Using `[FromServices]` for clean code

### Frontend Concepts:

1. **Reactive Forms:** Validation and data binding
2. **HTTP Services:** Making API calls with error handling
3. **Observable Patterns:** Using RxJS operators like `tap`
4. **State Management:** Updating user authentication state
5. **Local Storage:** Persisting user session data
6. **Component Communication:** Service injection and usage

### Security Best Practices:

1. ☑ Passwords are hashed before storage
2. ☑ Input validation on both frontend and backend
3. ☑ JWT tokens for stateless authentication
4. ☑ No sensitive data in local storage (except encrypted tokens)
5. ☑ Proper error handling without exposing system details

---

## Next Steps

Consider implementing these additional features:

1. **Email Verification:** Send confirmation emails for new accounts
2. **Password Strength:** Add more sophisticated password requirements
3. **Rate Limiting:** Prevent signup spam with rate limiting
4. **Social Login:** Add Google/Facebook authentication options
5. **Profile Pictures:** Allow users to upload avatars during signup

6. **Terms & Privacy:** Link to actual terms of service and privacy policy

## Conclusion

You've successfully implemented a complete user signup system with automatic login! This foundation can be extended with additional features and security measures as your application grows.

Remember: Always test thoroughly, handle errors gracefully, and keep security as a top priority in authentication systems.