Render.com Migration Guide: Bladder Event Tracker

From Azure Container Apps ($35/month) to Render.com ($13/month)

Time Required: 2-4 hours
Difficulty: Intermediate
Cost Savings: $22.83/month ($274/year)

Overview

This guide migrates your .NET 9 Minimal API + Angular 17 application from Azure Container Apps with SQL Server to Render.com with PostgreSQL.

What You'll Accomplish:

- ✅ Convert database from SQL Server → PostgreSQL
- ✅ Update .NET 9 API to use Npgsql (PostgreSQL driver)
- ✅ Deploy backend API to Render Web Service ($7/month)
- ✅ Deploy Angular 17 frontend to Render Static Site (Free)
- ✅ Set up PostgreSQL database on Render ($6/month)
- ✅ Migrate existing data
- ✅ Configure CI/CD with GitHub Actions

Phase 1: Preparation (30 minutes)

1.1 Create Render Account

Done!

1. Go to https://render.com
2. Sign up with GitHub (recommended for easy deployment)
3. Verify your email
4. Add a payment method (required for paid services, but won't be charged until you create paid resources)

1.2 Backup Your Current Database

In Azure Portal:

bash

```
# Connect to your Azure SQL Database and export data
# We'll use this backup if anything goes wrong
```

1. Go to Azure Portal → SQL Databases → Your database
2. Click "Export"

3. Save the .bacpac file locally as backup

1.3 Install PostgreSQL Locally (for testing)

Windows (using Chocolatey):


cmd

```
choco install postgresql
```

Or download installer: https://www.postgresql.org/download/windows/

Verify installation:


cmd

```
psql --version
```

1.4 Update Your Project Dependencies

Navigate to your backend project folder:


cmd

```
cd C:\dev\apps\BladderTracker\appProject\backend
```

Phase 2: Convert Backend to PostgreSQL (1 hour)

2.1 Update NuGet Packages

Remove SQL Server packages and add PostgreSQL:


cmd

```
dotnet remove package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL --version 9.0.0
```

Your .csproj should now include:


xml

```
<PackageReference Include="Npgsql.EntityFrameworkCore.PostgreSQL" Version="9.0.0" />
<PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="9.0.4" />
```

2.2 Update DbContext Configuration

Find your `Program.cs` (or wherever you configure services):

BEFORE (SQL Server):


csharp

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

AFTER (PostgreSQL):


csharp

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseNpgsql(builder.Configuration.GetConnectionString("DefaultConnection")));
```

2.3 Update appsettings.json

BEFORE:


json

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=tcp:yourserver.database.windows.net,1433;Database=BETrackingDb;..."
  }
}
```

AFTER (for local PostgreSQL testing):


json

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Host=localhost;Database=bladdertracker;Username=postgres;Password=yourpassword"
  }
}
```

Note: For Render deployment, we'll use environment variables instead.

2.4 Handle PostgreSQL-Specific Differences

PostgreSQL handles some things differently than SQL Server. Here are common adjustments:

A. Case Sensitivity

PostgreSQL is case-sensitive for table/column names. Update your entity configurations:

In your DbContext or entity configurations:


csharp

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // PostgreSQL naming convention: use lowercase with underscores
    modelBuilder.Entity<User>()
        .ToTable("users");  // lowercase table names

    modelBuilder.Entity<TrackingLog>()
        .ToTable("tracking_logs");

    // Or apply to all entities:
    foreach (var entity in modelBuilder.Model.GetEntityTypes())
    {
        entity.SetTableName(entity.GetTableName().ToLower());
    }
}
```

B. GUID/UUID Handling

If you use GUIDs as primary keys:


csharp

```csharp
public class TrackingLog
{
    public Guid Id { get; set; }  // This works fine in PostgreSQL as UUID
    // ... other properties
}
```

C. DateTime Handling

PostgreSQL uses UTC timestamps. Ensure your code explicitly uses UTC:


csharp

```csharp
// In your entities or when saving:
public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
```

D. Check Constraints (If You Use Them)

The syntax is slightly different:

SQL Server:


csharp

```csharp
entity.ToTable("TableName", ck => ck.HasCheckConstraint("CK_Name", "[Column] > 0"));
```

PostgreSQL (same syntax works in EF Core 9):


csharp

```csharp
entity.ToTable("tablename", ck => ck.HasCheckConstraint("ck_name", "\"column\" > 0"));
```

2.5 Create New PostgreSQL Migration

Delete old SQL Server migrations:


cmd

```
rd /s /q Migrations
```

Create fresh PostgreSQL migration:


cmd

```
dotnet ef migrations add InitialPostgreSQL
```

Review the generated migration to ensure it looks correct.

2.6 Test Locally with PostgreSQL

Start local PostgreSQL:


cmd

```
# Create local database
psql -U postgres
CREATE DATABASE bladdertracker;
\q
```

Update migrations:


cmd

```
dotnet ef database update
```

Run the API:


cmd

```
dotnet run
```

Test endpoints using Postman or your browser to ensure everything works with PostgreSQL.

Phase 3: Update Angular Frontend (15 minutes)

3.1 Update API URLs for Render

In `frontend/src/environments/environment.ts`:

typescript

```typescript
export const environment = {
  production: false,
  apiUrl: 'http://localhost:8080/api'  // For local development
};
```

In `frontend/src/environments/environment.prod.ts`:

typescript

```typescript
export const environment = {
  production: true,
  apiUrl: 'https://your-app-name.onrender.com/api'  // Will update after Render deployment
};
```

3.2 Ensure CORS is Configured

In your backend `Program.cs`:

csharp

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowRenderFrontend", policy =>
    {
        policy.WithOrigins(
            "http://localhost:4200",  // Local development
            "https://your-frontend-app.onrender.com"  // Render static site (update after deployment)
        )
        .AllowAnyMethod()
        .AllowAnyHeader()
        .AllowCredentials();
    });
});

// After app = builder.Build():
app.UseCors("AllowRenderFrontend");
```

Phase 4: Deploy to Render.com (45 minutes)

4.1 Push Code to GitHub

Ensure your code is in a GitHub repository:

cmd

```
cd C:\dev\apps\BladderTracker\appProject

# If not already a git repo:
git init
git add .
git commit -m "Prepare for Render deployment - PostgreSQL migration"

# Create GitHub repo and push
git remote add origin https://github.com/yourusername/bladdertracker.git
git branch -M main
git push -u origin main
```

4.2 Create PostgreSQL Database on Render

1. Log into https://dashboard.render.com
2. Click **"New +"** → **"PostgreSQL"**
3. Configure:
   - **Name:** `bladdertracker-db`
   - **Database:** `bladdertracker` (auto-filled)
   - **User:** `bladdertracker` (auto-filled)
   - **Region:** Oregon (US West) - cheapest for West Coast
   - **PostgreSQL Version:** 16 (latest)
   - **Instance Type: Starter** ($7/month) - wait, we want **Basic 256MB** ($6/month)

   IMPORTANT: Select **"Basic 256MB"** for $6/month, not "Starter" which is $7/month!
4. Click **"Create Database"**

Wait 2-3 minutes for database provisioning.

5. Once created, you'll see:
   - **Internal Database URL** (use this from backend)
   - **External Database URL** (use for psql access)

Copy the Internal Database URL - it looks like:

```
postgresql://bladdertracker:RANDOM_PASSWORD@dpg-xxxxx-a/bladdertracker
```

4.3 Deploy Backend Web Service

1. In Render Dashboard, click **"New +"** → **"Web Service"**
2. Connect your GitHub repository
3. Configure:
   - **Name:** `bladdertracker-api`
   - **Region:** Same as database (Oregon US West)
   - **Branch:** `main`
   - **Root Directory:** `backend` (or path to your .NET project)
   - **Runtime:** `Docker` (Render auto-detects .NET)

   Build Command: Leave blank (Render auto-detects .NET) **Start Command:** Leave blank (uses default)
   - **Instance Type: Starter** ($7/month)
4. **Environment Variables -** Click "Add Environment Variable":

Key	Value
ASPNETCORE_ENVIRONMENT	Production
ConnectionStrings__DefaultConnection	(Paste Internal Database URL from step 4.2)
JWT_SECRET	(Generate: openssl rand -base64 32)
JWT_ISSUER	BladderTrackerAPI
JWT_AUDIENCE	BladderTrackerClients
JWT_EXPIRY_MINUTES	60

5. Click **"Create Web Service"**

Wait 5-10 minutes for initial deployment. Render will:

- Clone your repo
- Detect .NET 9
- Build your application
- Start the service

Important: If build fails, you may need to add a `Dockerfile`:

Create `backend/Dockerfile`:


dockerfile

```
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base
WORKDIR /app
EXPOSE 8080

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src
COPY ["trackerApi.csproj", "./"]
RUN dotnet restore "trackerApi.csproj"
COPY . .
RUN dotnet build "trackerApi.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "trackerApi.csproj" -c Release -o /app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "trackerApi.dll"]
```

Commit and push, Render will auto-redeploy.

4.4 Run Database Migrations

Once your API is deployed:

1. In Render Dashboard → Your Web Service → **Shell** tab
2. Click **"Connect"** to open SSH shell
3. Run migrations:


bash

```
dotnet ef database update
```

Alternative: Add migration command to startup:

In `Program.cs` (before `app.Run()`):

csharp

```
// Auto-run migrations on startup (convenient for Render)
using (var scope = app.Services.CreateScope())
{
    var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
    dbContext.Database.Migrate();
}

app.Run();
```

Redeploy and migrations will run automatically.

4.5 Deploy Frontend to Render Static Site

1. In Render Dashboard, click **"New +"** → **"Static Site"**
2. Connect your GitHub repository
3. Configure:
    - **Name:** `bladdertracker-frontend`
    - **Branch:** `main`
    - **Root Directory:** `frontend`
    - **Build Command:** `npm install && npm run build`
    - **Publish Directory:** `dist/frontend/browser` (Angular 17 output path)
4. **Environment Variables:**

Key	Value
API_URL	https://bladdertracker-api.onrender.com

5. Click **"Create Static Site"**

Important: Update Angular environment file to use this environment variable:

`frontend/src/environments/environment.prod.ts`:

typescript

```
export const environment = {
  production: true,
  apiUrl: 'https://bladdertracker-api.onrender.com/api'
};
```

Commit and push the change.

4.6 Update CORS Configuration

Now that you have your frontend URL, update backend CORS:

In `backend/Program.cs`:

csharp

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowRenderFrontend", policy =>
    {
        policy.WithOrigins(
            "http://localhost:4200",
            "https://bladdertracker-frontend.onrender.com"  // Update with your actual URL
        )
        .AllowAnyMethod()
        .AllowAnyHeader()
        .AllowCredentials();
    });
});
```

Commit, push, and Render will auto-redeploy.

Phase 5: Migrate Data from Azure SQL (30 minutes)

5.1 Export Data from Azure SQL

Option A: Using Azure Data Studio or SSMS:

1. Connect to your Azure SQL Database
2. Right-click database → Tasks → Export Data
3. Export to CSV files for each table

Option B: Using SQL Script:

sql

```sql
-- Export Users table
SELECT * FROM Users;
-- Copy results to CSV

-- Export TrackingLogs table
SELECT * FROM TrackingLogs;
-- Copy results to CSV
```

5.2 Import Data to Render PostgreSQL

Connect to Render PostgreSQL:

1. Get the **External Connection String** from Render Dashboard → PostgreSQL Database
2. Use psql or a PostgreSQL client:

bash

```bash
psql "postgresql://bladdertracker:PASSWORD@dpg-xxxxx-a.oregon-postgres.render.com/bladdertracker"
```

Import data using COPY command:

sql

```sql
-- First, verify your tables exist:
\dt

-- Import Users (adjust column names to match your schema)
COPY users(id, username, password_hash, is_admin, created_at)
FROM '/path/to/users.csv'
DELIMITER ','
CSV HEADER;

-- Import TrackingLogs
COPY tracking_logs(id, user_id, event_type, timestamp, notes)
FROM '/path/to/tracking_logs.csv'
DELIMITER ','
CSV HEADER;
```

Alternative: Use DBeaver or pgAdmin for GUI-based import if you prefer.

5.3 Verify Data Migration


sql

```sql
-- Check row counts
SELECT COUNT(*) FROM users;
SELECT COUNT(*) FROM tracking_logs;

-- Sample data check
SELECT * FROM users LIMIT 5;
SELECT * FROM tracking_logs LIMIT 5;
```

Phase 6: Testing & Validation (15 minutes)

6.1 Test Backend API

Your API URL: `https://bladdertracker-api.onrender.com`

Test endpoints:


bash

```
# Health check (if you have one)
curl https://bladdertracker-api.onrender.com/health

# Login
curl -X POST https://bladdertracker-api.onrender.com/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"yourpassword"}'
```

6.2 Test Frontend

Your frontend URL: `https://bladdertracker-frontend.onrender.com`

1. Open in browser
2. Try logging in
3. Create a test tracking log
4. Verify data appears

6.3 Monitor First Deployment

Important Render Free Tier Behaviors:

- **Free web services spin down after 15 minutes of inactivity**
- Your backend is on **Starter ($7/month)** so it stays running
- First request after idle may take 30-60 seconds (cold start)
- Static site (frontend) has no cold starts

Check logs in Render Dashboard:

- Web Service → Logs tab
- Look for errors or warnings

Phase 7: Setup CI/CD (15 minutes)

Render **automatically deploys** when you push to GitHub! But you can enhance this:

7.1 Create GitHub Actions Workflow

`.github/workflows/render-deploy.yml`:

yaml

```yaml
name: Deploy to Render

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test-backend:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Setup .NET 9
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: '9.0.x'

      - name: Restore dependencies
        run: dotnet restore
        working-directory: ./backend

      - name: Build
        run: dotnet build --no-restore
        working-directory: ./backend

      - name: Test
        run: dotnet test --no-build --verbosity normal
        working-directory: ./backend

  test-frontend:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
```

```
        node-version: '20'
        cache: 'npm'
        cache-dependency-path: ./frontend/package-lock.json

    - name: Install dependencies
      run: npm ci
      working-directory: ./frontend

    - name: Build
      run: npm run build -- --configuration production
      working-directory: ./frontend

    - name: Test
      run: npm test -- --watch=false --browsers=ChromeHeadless
      working-directory: ./frontend
```

7.2 Configure Auto-Deploy Notifications

In Render Dashboard → Settings → Notifications:

- Enable email notifications for deployments
- Add Slack webhook (optional)

Phase 8: Cleanup Azure Resources (Save Money!)

Once everything works on Render:

8.1 Stop Azure Services

In Azure Portal:

1. **Stop Container Apps:**
   - Navigate to your Container Apps
   - Click "Stop" (stops billing immediately)
2. **Pause Azure SQL Database:**
   - Navigate to SQL Database
   - Click "Pause" (for serverless, it auto-pauses)
   - Or delete if you've migrated all data
3. **Delete Container Registry:**
   - Navigate to Container Registry
   - Click "Delete"
   - Confirm deletion

Monthly savings: $23-35!

8.2 Keep Azure Account Active

Keep your Azure account with free tier services in case you need them later.

Cost Summary

Before Migration (Azure)

- Container Apps: $18.23/month
- Azure SQL Database: $12.32/month
- Container Registry: $4.99/month
- Files: $0.29/month
- **Total: $35.83/month**

After Migration (Render)

- Web Service (Starter): $7.00/month
- PostgreSQL (Basic): $6.00/month
- Static Site: $0.00/month
- **Total: $13.00/month**

Annual Savings: $274.00 🎉

Troubleshooting Common Issues

Issue 1: Database Connection Fails

Error: `Npgsql.NpgsqlException: Connection refused`

Solution:

- Verify connection string in Render environment variables
- Ensure you're using **Internal Database URL**, not External
- Check database is in same region as web service

Issue 2: Build Fails on Render

Error: `No executable found matching command "dotnet"`

Solution:

- Add `Dockerfile` to backend (see Phase 4.3)
- Ensure .NET 9 SDK is specified in Dockerfile
- Check build logs for specific errors

Issue 3: CORS Errors in Frontend

Error: `Access to fetch at '...' from origin '...' has been blocked by CORS`

Solution:

- Update CORS configuration in `Program.cs` with correct Render URLs
- Ensure `app.UseCors()` is called BEFORE `app.MapControllers()`
- Verify frontend is using correct API URL in environment.prod.ts

Issue 4: Cold Starts Are Slow

Behavior: First request after 15 minutes takes 30-60 seconds

Solution:

- This is normal for free tier web services
- Your **Starter ($7/month) backend won't have this issue**
- Consider upgrading to Standard ($25/month) if you need instant response

Issue 5: Migration Command Not Found

Error: `dotnet ef: command not found`

Solution: Install EF Core tools globally:


bash

```
dotnet tool install --global dotnet-ef
```

Next Steps

1. **Monitor costs** in Render Dashboard → Billing
2. **Set up backups** - Render Basic PostgreSQL includes daily backups
3. **Configure custom domain** (optional) - Render supports custom domains for free
4. **Enable HTTPS** - Automatic with Render (free SSL certificates)
5. **Set up monitoring** - Consider using Render's built-in monitoring or add Application Insights

---

Support Resources

- **Render Docs:** https://render.com/docs
- **Render Community:** https://community.render.com
- **PostgreSQL Docs:** https://www.postgresql.org/docs/
- **Npgsql Docs:** https://www.npgsql.org/doc/

Questions?

If you encounter any issues during migration, let me know and I can help troubleshoot specific errors with your .NET 9 or Angular 17 code!