Data synchronization service Proof of Concept

# Overview

The data sync ensures that two records in two databases are kept in sync using simple rules:

1. Traverse through a source db patient list to see if each patient is in the destination db. If yes, update it if its last modified date is earlier than this record (destination record is stale). If not, update the source record.
2. If this record is not in the destination db, insert it.
3. You can use a UHID or Name or similar ID (not driven by database) to check whether the given entity is there in the destination db. Names can be repeated. So, using UHID or similar unique ID is recommended.
4. The same steps can be used for each and every entity.
5. The IDs automatically set by the database are unique within the same database but not across two databases. So these need to be avoided. UHID is the best bet.

# Create a patient table in source and destination databases.

These databases could be on the same PostgreSQL instance or two different instances.

-- Table: public.Patient

-- DROP TABLE public."Patient";

CREATE TABLE IF NOT EXISTS public."Patient"

(

"Name" character varying(100)[] COLLATE pg\_catalog."default" NOT NULL,

"ID" bigint NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1 START 1 MINVALUE 1 MAXVALUE 1000000000 CACHE 1 ),

"Address" character varying(500)[] COLLATE pg\_catalog."default" NOT NULL,

"LastModified" time without time zone,

CONSTRAINT "Patient\_pkey" PRIMARY KEY ("ID")

)

TABLESPACE pg\_default;

ALTER TABLE public."Patient"

OWNER to postgres;

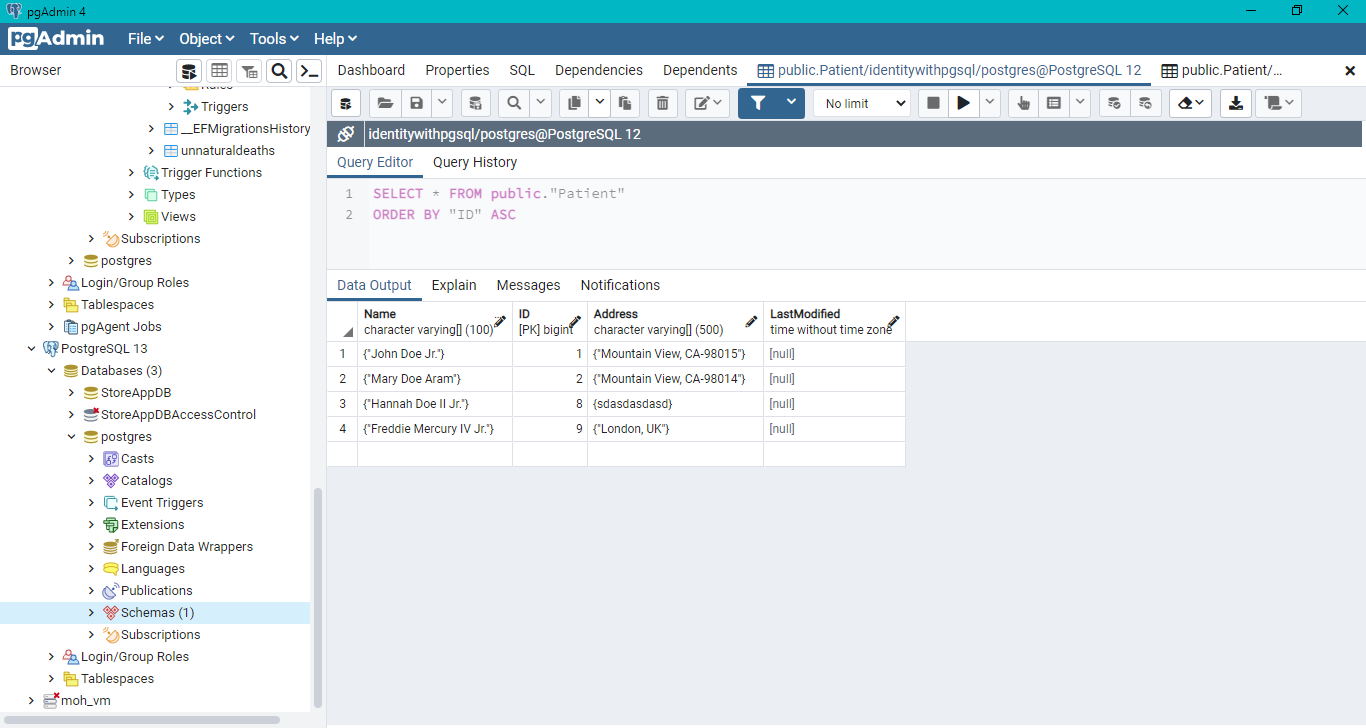


Figure Source patient list

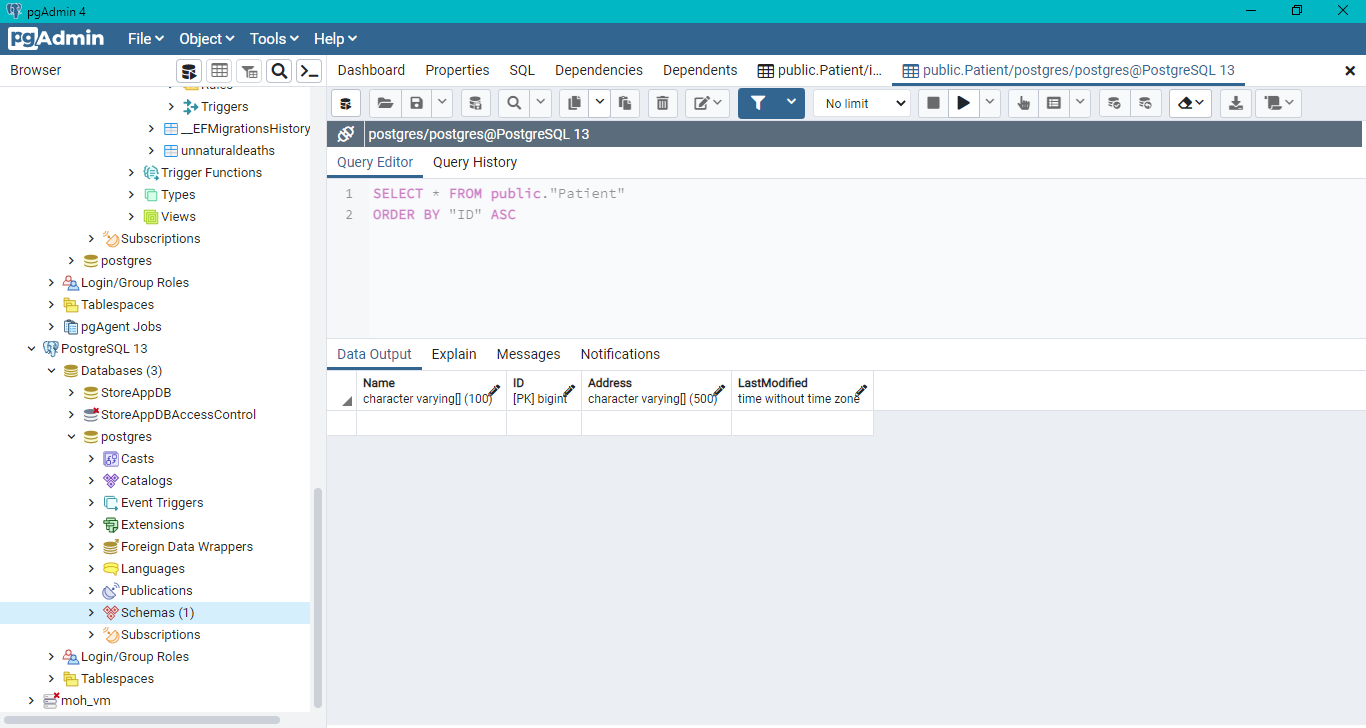


Figure Destination patient list (empty initially)

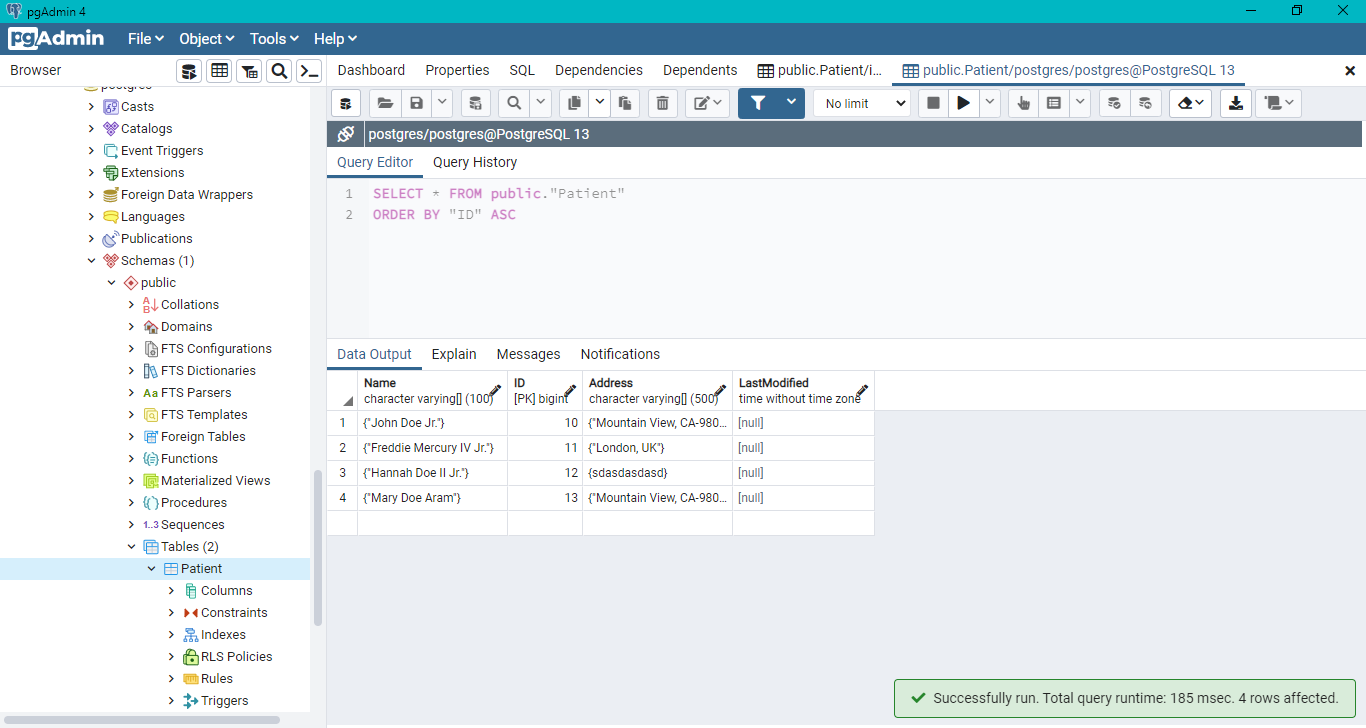


Figure Initial sync

# Data Sync Service Example

using Common;

using Common.Models;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Logging;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace datasyncservice.Services

{

public class PatientSyncService : BackgroundService

{

private readonly ILogger<PatientSyncService> \_logger;

public PatientSyncService(ILogger<PatientSyncService> logger)

{

\_logger = logger;

}

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

\_logger.LogDebug($"PatientSyncService is starting.");

stoppingToken.Register(() =>

\_logger.LogDebug($" PatientSyncService background task is stopping."));

var sourceConnection = @"Server=127.0.0.1;Port=5432;Database=identitywithpgsql;User Id=postgres;Password=Tetya1:2;";

var destinationConnection = @"Server=127.0.0.1;Port=5433;Database=postgres;User Id=postgres;Password=Tetya1:2;";

var \_sourceContext = new identitywithpgsqlContext();

\_sourceContext.Database.GetDbConnection().ConnectionString = sourceConnection;

var \_destinationContext = new identitywithpgsqlContext();

\_destinationContext.Database.GetDbConnection().ConnectionString = destinationConnection;

var listFromEF = \_sourceContext.Patient.ToList();

var list = new List<PatientDto>();

foreach (var pat in listFromEF)

{

// See if this patient is in the destination db.

// If yes, update it if its last modified date is earlier than this

// record (todo). If not, update this record (todo).

// If this record is not in the destination db, insert it (see below).

// You can use a UHID or Name or similar to check whether the given

// patient is there in the dest db.

// Names can be repeated. So please use UHID.

// The below is just a crude example.

var isThere = \_destinationContext.Patient.ToList().FirstOrDefault(p => p.Name.FirstOrDefault().ToLower() == pat.Name.FirstOrDefault().ToLower()) != null;

if (!isThere)

{

// Insert it.

// Reset Unique IDs because these will be inserted by the system.

// Use UHIDs as stated above to check for uniqueness.

pat.Id = 0;

await \_destinationContext.Patient.AddAsync(pat);

}

}

try

{

await \_destinationContext.SaveChangesAsync();

}

catch (Exception ex)

{

}

\_logger.LogDebug($"PatientSyncService task doing background work.");

\_logger.LogDebug($"PatientSyncService background task is stopping.");

}

}

}

The data sync service compares two patient records by their UHID/Name/any other suitable attribute and also the last modified time.

## Algorithm

1. See if this patient is in the destination db. If yes, update it if its last modified date is earlier than this record (destination record is stale). If not, update the source record.
2. If this record is not in the destination db, insert it.
3. You can use a UHID or Name or similar to check whether the given patient is there in the destination db. Names can be repeated. So using UHID is recommended.
4. The same steps can be used for each and every entity.

# UI

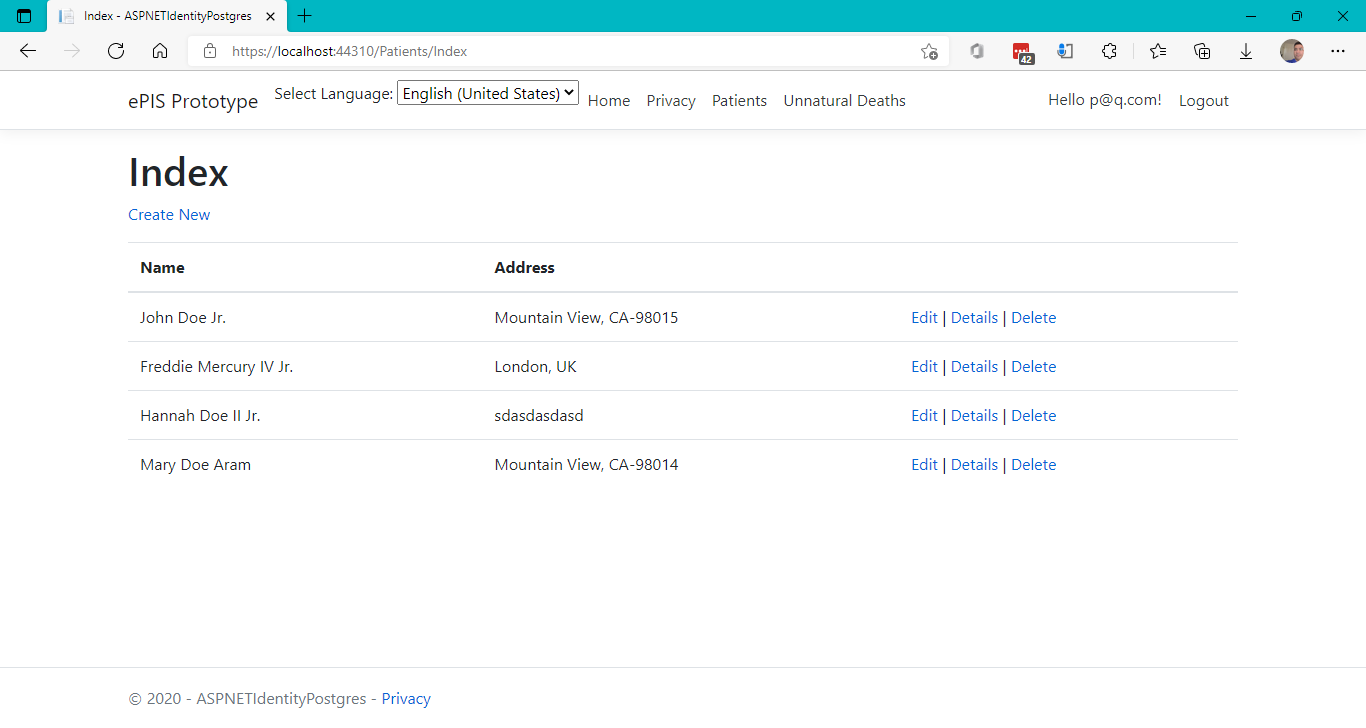


Figure Patient List. Click the Create new button.

## Create Patient

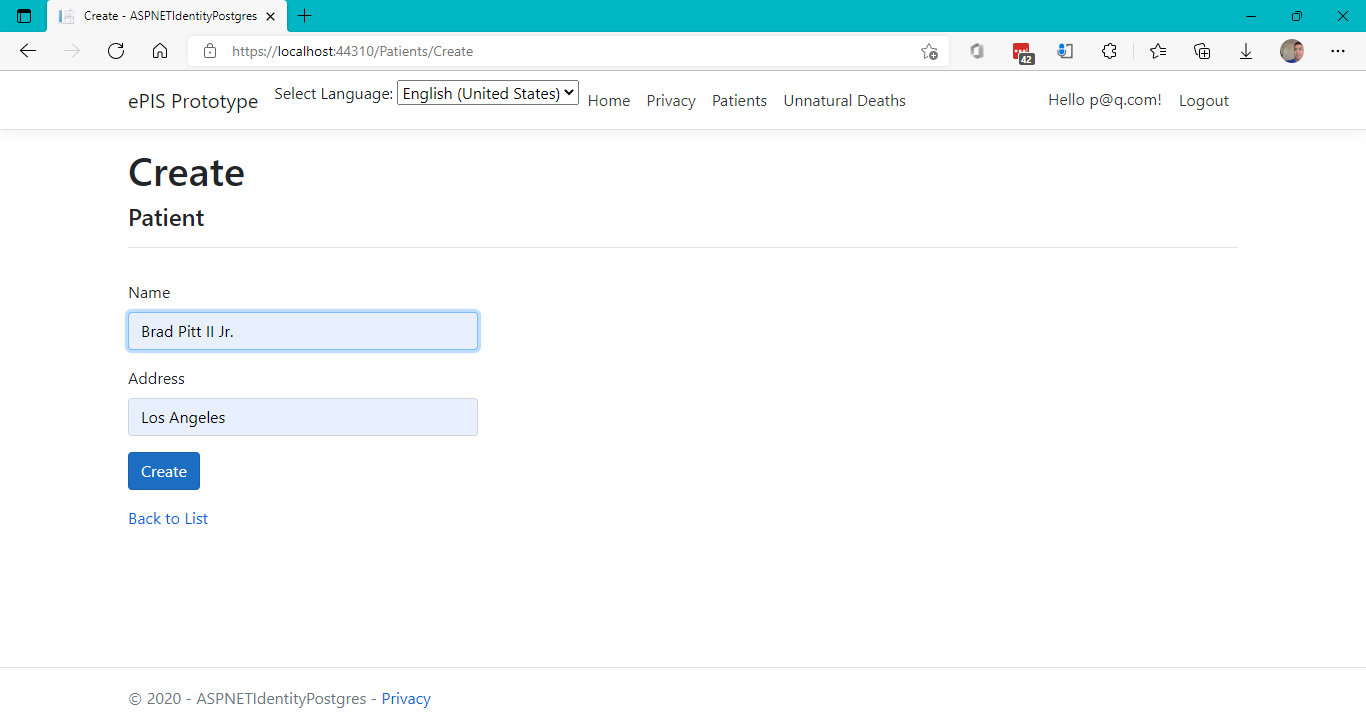


Figure Add patient details. Click “Create”.

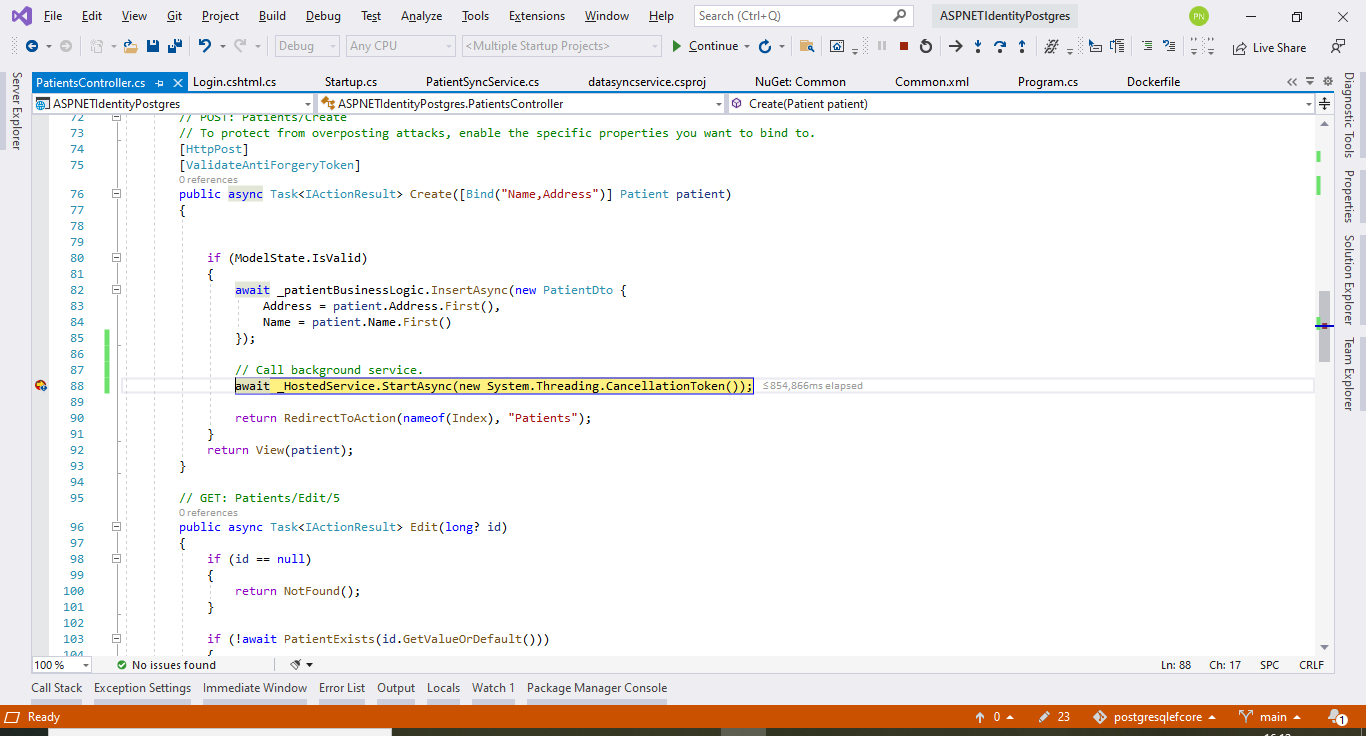


Figure The MVC controller calls the background service to check if this newly added patient is in the destination db.

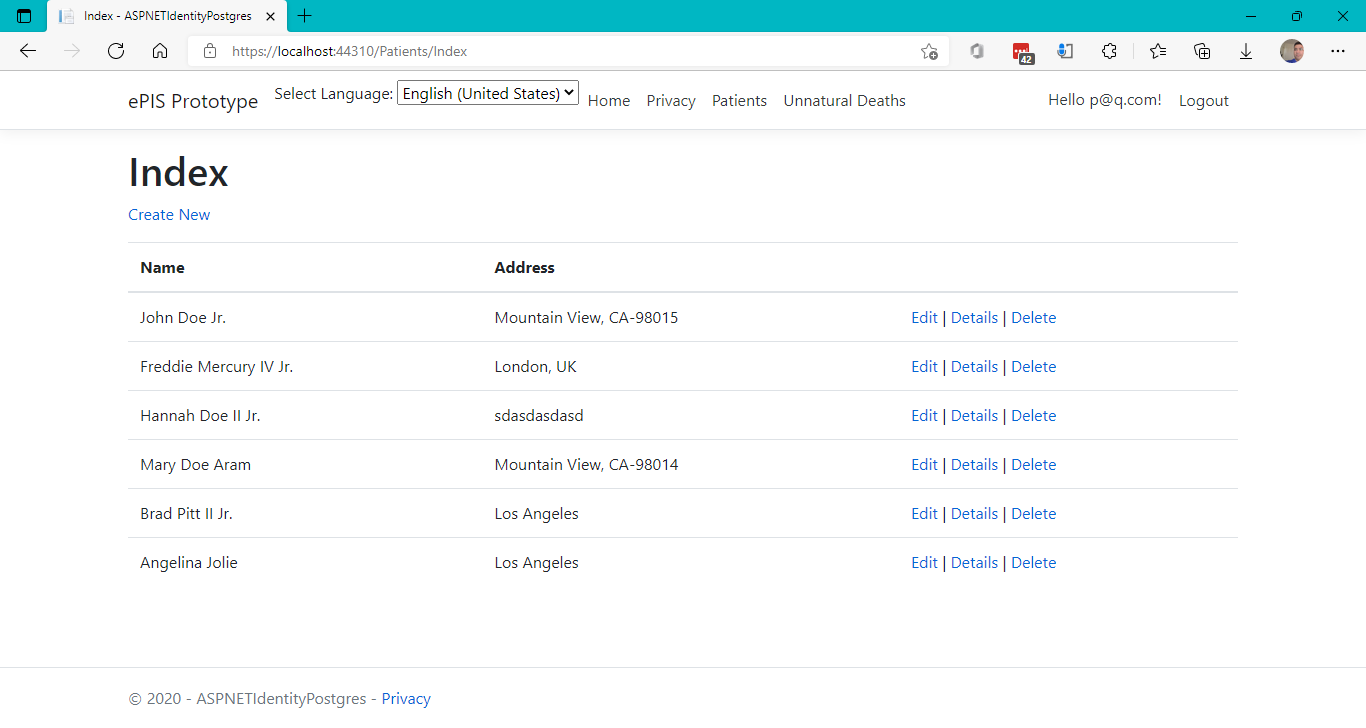


Figure Newly added patients

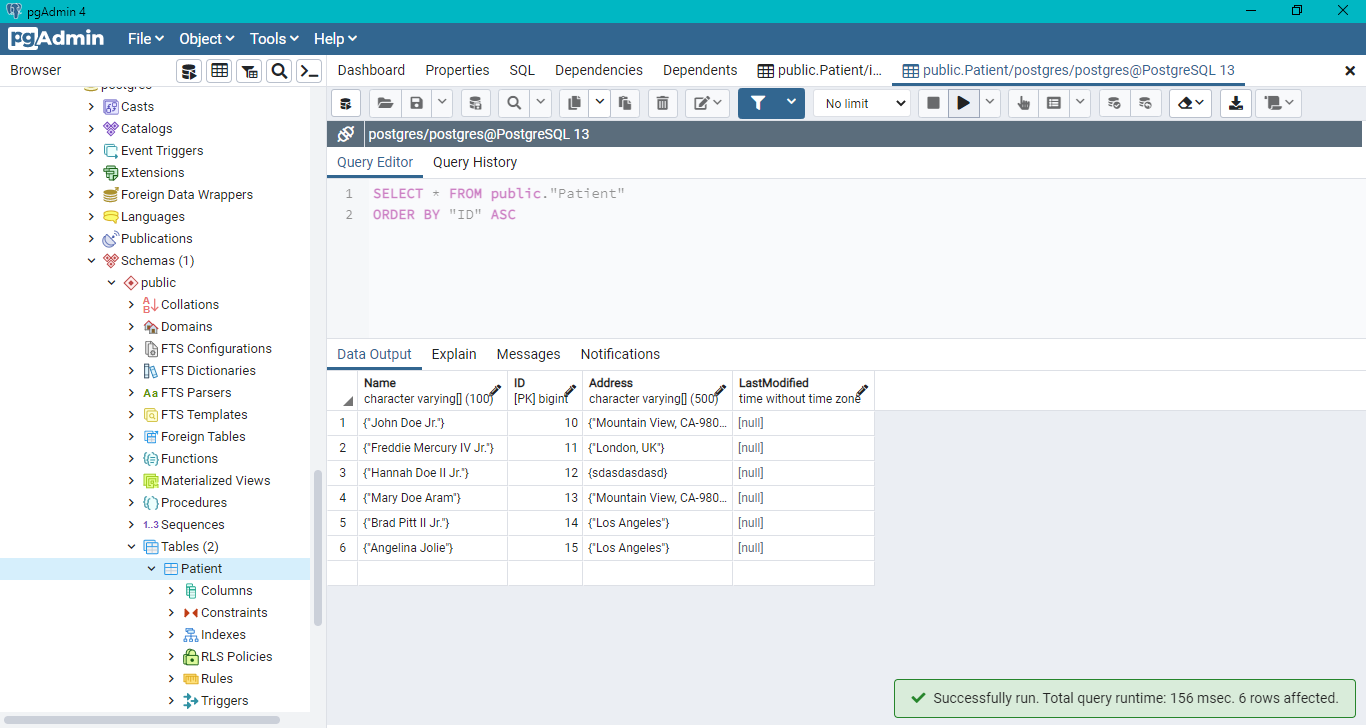


Figure Newly added records get added to destination db thereby proving a basic sync

# Appendix

## Background Service class

using Microsoft.Extensions.Hosting;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace datasyncservice.Services

{

// Copyright (c) .NET Foundation. Licensed under the Apache License, Version 2.0.

/// <summary>

/// Base class for implementing a long running <see cref="IHostedService"/>.

/// </summary>

public abstract class BackgroundService : IHostedService, IDisposable

{

private Task \_executingTask;

private readonly CancellationTokenSource \_stoppingCts =

new CancellationTokenSource();

protected abstract Task ExecuteAsync(CancellationToken stoppingToken);

public virtual Task StartAsync(CancellationToken cancellationToken)

{

// Store the task we're executing

\_executingTask = ExecuteAsync(\_stoppingCts.Token);

// If the task is completed then return it,

// this will bubble cancellation and failure to the caller

if (\_executingTask.IsCompleted)

{

return \_executingTask;

}

// Otherwise it's running

return Task.CompletedTask;

}

public virtual async Task StopAsync(CancellationToken cancellationToken)

{

// Stop called without start

if (\_executingTask == null)

{

return;

}

try

{

// Signal cancellation to the executing method

\_stoppingCts.Cancel();

}

finally

{

// Wait until the task completes or the stop token triggers

await Task.WhenAny(\_executingTask, Task.Delay(Timeout.Infinite,

cancellationToken));

}

}

public virtual void Dispose()

{

\_stoppingCts.Cancel();

}

}

}