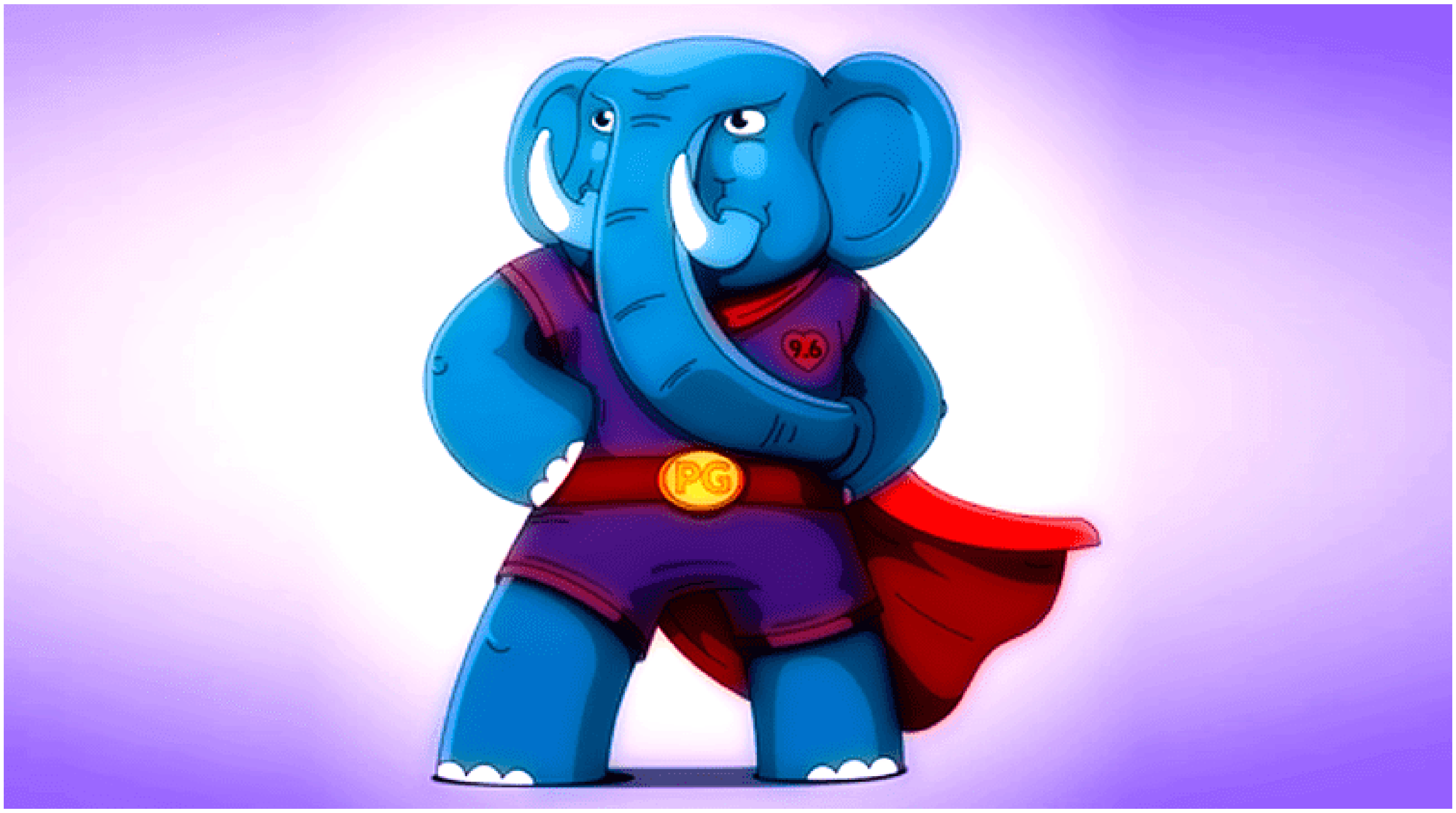
Postgres Superpowers in Practice

2023-04-15 OSKAR DUDYCZ POSTGRES



Look! Up in the sky! It's a bird! It's a plane! It's Superman! I have such a thought quite often while working with Postgres. Why?

Let's say that you're building Car Fleet Management System. You must manage all data about the company's cars, drivers, trips, fuel management, etc. In a nutshell, that's more around accounting and compliance than driving. If your company is a big one in the logistics space, you have a lot of data to manage and process.

Let's start with a simple case: trip management. We could come up with the following table:

```sql
CREATE TABLE trips (
    trip_time TIMESTAMP NOT NULL,
    vehicle_id INT NOT NULL,
    driver_name VARCHAR(255) NOT NULL,
    start_location TEXT NOT NULL,
    end_location TEXT NOT NULL,
    distance_kilometers NUMERIC(10,2) NOT NULL,
    fuel_used_liters NUMERIC(10,2) NOT NULL,
    PRIMARY KEY (trip_time, vehicle_id)
);
```

It's a simplified version, but it covers basic needs. We have trip per vehicle, driver information, how long the trip was, and how much fuel was used. That doesn't look scary, but if we'd like to make it fast and scalable for reporting and alerting needs, then we should do better than that. Of course, we could add some indexes, but we might still need more adjustments. The data size could grow, and querying and processing might need to be faster. Especially keeping in mind that this may be just the centrepiece of the normalised table schema, add into that table that tracks online the GPS location, and the size of data is skyrocketing like Superman in the sky.

Postgres provides built-in partitioning capabilities. In a nutshell, we can define what data from the table we're using to partition our data. Data will be physically stored in different disk locations grouped by partition criteria. Postgres will handle the routing of inserts, queries, etc. We can still use the table as the regular one.

In our case, we could use the partitioning-by-date strategy because we'll be primarily interested in trips in a selected time range. We can do that by adding *PARTITION BY RANGE (trip_time)* in our table definition:

```sql
CREATE TABLE trips (
    trip_time TIMESTAMP NOT NULL,
    vehicle_id INT NOT NULL,
    driver_name VARCHAR(255) NOT NULL,
    start_location TEXT NOT NULL,
    end_location TEXT NOT NULL,
    distance_kilometers NUMERIC(10,2) NOT NULL,
    fuel_used_liters NUMERIC(10,2) NOT NULL,
    PRIMARY KEY (trip_time, vehicle_id)
```

We don't need to change our queries or inserts. We can also detach and attach those partitions by a single command, which makes ops work much easier.

That's neat, and the topic is on the blog by itself. Still, it has some tedious parts. For instance, you need to define partitions explicitly upfront; Postgres won't create them automatically while inserting data.

Typically, there's some CRON job setting up partitions, e.g.

```
DO $$DECLARE
    month_start_date DATE := '2023-01-01';
    month_end_date DATE := '2023-12-01';
BEGIN
    WHILE month_start_date < month_end_date LOOP
        EXECUTE format('
            CREATE TABLE trips_%s PARTITION OF trips
            FOR VALUES FROM (%L) TO (%L);',
            TO_CHAR(month_start_date, 'YYYY_MM'),
            month_start_date,
            month_start_date + INTERVAL '1 month'
        );
        month_start_date := month_start_date + INTERVAL '1 month';
    END LOOP;
END$$;
```

Sometimes they're used in triggers before inserting new rows.

Not that terrible, but with a bigger scale, managing that can get complicated.

Introducing TimescaleDB

Postgres is not only Superman but also Transformer. From the ground basis, it is built to support extensions without changing the core database system. PostgreSQL extensions can enhance the core features with new data types, functions, operators, indexing methods, and more. You can write them in languages like C, Rust, etc. Extensions are a way to package and distribute these additional features, making it easy for users to install and manage them.

There are a lot of mature plugins provided by external companies and communities. Most of them are open-sourced and free. One of them is TimescaleDB. Their slogan is *"Postgres for time-series"*. And it's like that. It provides various tooling to make your time-based analytics and data processing easier and faster. You can install it on your Postgres installation for free, but most hostings (also big cloud providers) provide you with an option to enable it out of the box.

Once you have it, toggle it with the following SQL:

```sql
CREATE EXTENSION IF NOT EXISTS timescaledb;
```

It's a Postgres built-in syntax and a general pattern for enabling extensions.

Ok, getting back to TimescaleDB and partitioning. Postgres has a feature that maybe is not as super as Superman, but at least it's hyper.

```sql
SELECT create_hypertable('trips', 'trip_time');
```

create_hypertable is a custom function that enables partitioning on TimescaleDB steroids. We don't need to create partitions; TimescaleDB will handle that for us and do other performance optimisation.

That's also why TimescaleDB is an excellent solution for IoT. When we have a huge data coming in a short period, the built-in internal capabilities can optimise the ingress for us.

That's sweet, but let's not stop here and just do simple queries you can imagine (like average distance in a date range, etc.).

Let's build a report calculating the average fuel efficiency in the last 30 days. This is quite useful in Fleet Management to detect fraud on people forgetting to log all trips, using the car for their needs, etc. We can do that by getting the average usage per kilometre. It's, of course, a simplified scenario, but you get the idea.

```sql
CREATE MATERIALIZED VIEW vehicle_fuel_efficiency_avg
WITH (timescaledb.continuous) AS
SELECT time_bucket('1 day', trip_time) AS bucket,
       AVG(distance_kilometers)/AVG(fuel_used_liters) AS fuel_efficiency_avg
FROM trips
WHERE trip_time >= now() - INTERVAL '30 days'
GROUP BY bucket, vehicle_id;
```

Materialised views are a feature that enables you to build a read-only aggregation of your table data; what's more, they're not calculated on the fly while doing queries but stored on disk, thus getting better performance. They're automatically available in Postgres and cool enough, but again a bit tedious, as you need to trigger their recalculation manually. That's necessary, as rebuilding them may take time and be resource-demanding. You probably know where I'm going; yes, TimescaleDB can help you with the continuous aggregates feature.

We need to mark our materialised view using *WITH (timescaledb.continuous)* and add policy:

```
SELECT add_continuous_aggregate_policy(
       continuous_aggregate => 'vehicle_fuel_efficiency_avg',
       start_offset => INTERVAL '30 days',
       end_offset => INTERVAL '1 second',
       schedule_interval => INTERVAL '1 day'
);
```

We're defining the initial date range of recalculation (from 30 days till 1 second ago) and the interval in which it should be updated (1 day). From now on, TimescaleDB will refresh materialised view for us automatically.

TimescaleDB extends Postgres also with a cron-like scheduler. It uses it internally to update materialised views. We'll use it later.

By the way, *Interval* is also a decent example of custom types feature Postgres provides. TimescaleDB defines this one, but you can define your own.

Generating alerts

Let's use our materialised view to generate alerts based on detected fuel usage anomalies. This table could look as:

```
CREATE TABLE fuel_efficiency_alerts (
    vehicle_id INT NOT NULL,
    start_time TIMESTAMP NOT NULL,
    end_time TIMESTAMP NOT NULL,
    fuel_efficiency NUMERIC(10,2) NOT NULL,
    PRIMARY KEY (vehicle_id, start_time)
);
```

Let’s not define it as *hypertable*; why? I’ll explain that later, for now, trust me, that’s better. We won’t be querying it; just getting information about the new records.

How to generate alerts? Let’s define a function for that:

```
CREATE OR REPLACE FUNCTION check_fuel_efficiency_and_insert_alerts(p_job_id INT
RETURNS VOID AS $$
BEGIN
  INSERT INTO
    fuel_efficiency_alerts (
      vehicle_id,
      start_time,
      end_time,
      fuel_efficiency
    )
  SELECT
    vehicle_id,
    bucket AS start_time,
    bucket + INTERVAL '1 day' AS end_time,
    fuel_efficiency_avg AS fuel_efficiency
  FROM
    vehicle_fuel_efficiency_avg
  WHERE
    fuel_efficiency_avg < 5
    AND bucket >= now() - INTERVAL '30 days'
  ON CONFLICT (vehicle_id, start_time) DO UPDATE
    SET
      fuel_efficiency = EXCLUDED.fuel_efficiency,
      end_time = EXCLUDED.end_time;

  DELETE FROM
    fuel_efficiency_alerts AS a
    NOT EXISTS (
      SELECT 1
      FROM
        vehicle_fuel_efficiency_avg AS f
      WHERE
        a.vehicle_id = f.vehicle_id
        AND f.bucket >= now() - INTERVAL '30 days'
        AND a.start_time = f.bucket
        AND f.fuel_efficiency_avg < 5
    );
END;
$$ LANGUAGE plpgsql;
```

Nothing fancy here; we're generating alerts based on the *magic factor* of average fuel usage (equal to 5), inserting or updating the current alert information and cleaning obsolete alerts.

I told you before that I'll use the TimescaleDB scheduler. Now it's the right time.

We can do it by calling the following function:

```
SELECT add_job('check_fuel_efficiency_and_insert_alerts', '5 seconds');
```

It tells which function should be called in which interval. Simple as that!

Having the data, we can generate a report that shows the fuel efficiency for each vehicle over time, as well as any alerts that have been generated.

```
SELECT trips.vehicle_id,
       trips.trip_time,
       trips.distance_kilometers/trips.fuel_used_liters AS fuel_efficiency,
       fuel_efficiency_alerts.start_time,
       fuel_efficiency_alerts.end_time
FROM trips
LEFT JOIN vehicle_fuel_efficiency_avg
ON trips.vehicle_id = vehicle_fuel_efficiency_avg.vehicle_id
    AND time_bucket('1 day', trips.trip_time) = vehicle_fuel_efficiency_avg.buc
LEFT JOIN fuel_efficiency_alerts ON trips.vehicle_id = fuel_efficiency_alerts.v
WHERE trips.trip_time >= now() - INTERVAL '30 days'
ORDER BY trips.vehicle_id, trips.trip_time;
```

Adding PostGIS

To not make this a TimescaleDB love poem, let's introduce another extension to give you broader coverage of the Postgres capabilities.

PostGIS is a plugin that enables advanced storage and transformation of spatial data, so geographic locations etc. It's compatible with most of the standards for storing and transforming positions. Sounds like we could use for our trips, aye?

Let's enable PostGIS:

```
CREATE EXTENSION IF NOT EXISTS postgis;
```

And extend our table with route information.

```
ALTER TABLE trips
ADD COLUMN route GEOMETRY(LINESTRING, 4326) NULL;
```

GEOMETRY is one of the types that PostGIS is adding. It allows you to store multiple points inside a single column. Of course, in the real world, we'd keep the GPS recordings in a dedicated table, but we could also keep a summary with the main route points in the trips table. We could use it to display it on the map in the UI.

If we store our start and end position in a format *{Latitude}, {Longitude}*, e.g. *52.292064, 21.036320*, we can also provide a default value based on the start and end locations.

```
UPDATE trips
SET route = ST_Transform(
    ST_MakeLine(
        ST_GeomFromText(
          'POINT(' ||
          split_part(start_location, ',', 1) || ' ' ||
          split_part(start_location, ',', 2) || ')',
          4326
        ),
        ST_GeomFromText(
          'POINT(' ||
          split_part(end_location, ',', 2) || ')',
          4326
        )
    ), 4326)
WHERE route IS NULL;

ALTER TABLE trips
ALTER COLUMN route SET NOT NULL;
```

You already see examples of the multiple functionalities that PostGis provides, e.g. parsing points and creating lines from them.

And hey, why not go further? We could notice that distance kilometres and start and end locations could be derived from route information. Why do we always need to calculate them when inserting? Actually, there's no need, as vanilla Postgres can help with that!

Generated columns

Postgres provides an option to automatically compute the column's value based on the data from others and store it for you inside the other column. This feature is called Generated columns. Let's showcase them by calculating data derived from the route in our trip table.

```
  ALTER TABLE trips
  DROP COLUMN distance_kilometers,
  ADD COLUMN distance_kilometers NUMERIC(10, 2) GENERATED ALWAYS
  AS ( ST_Length(route::geography) / 1000 ) STORED;

  DROP COLUMN start_location,
  ADD COLUMN start_location GEOMETRY(POINT, 4326) GENERATED ALWAYS
  AS ( ST_StartPoint(route) ) STORED;

  ALTER TABLE trips
  DROP COLUMN end_location,
  ADD COLUMN end_location GEOMETRY(POINT, 4326) GENERATED ALWAYS
  AS ( ST_EndPoint(route) ) STORED;
```

As you see, we can also use custom functions from plugins in it!

Logical Replication

If you read everything in the previous paragraphs, you deserve the Grand Finale!

I explained the superpowers of Postgres logical replication in detail in Push-based Outbox Pattern with Postgres Logical Replication.

> *Postgres have a concept called "Write-Ahead Log" (WAL). It is an append-only structure that records all the operations during transaction processing (Inserts, Updates, Deletes). When we commit a transaction, the data is firstly appended to the Write-Ahead Log. Then all operations are applied to tables, indexes, etc. Hence the name "Write-Ahead": from this writing data to the log in advance of other changes. So from that perspective, tables and indexes are just read models for Write-Ahead Log.*
>
> *Postgres is a rock-solid database with many superb features. One of them is JSON support we're using in Marten, and the other is logical replication that we'll look closer at now.*
>
> *Logical replication takes the traditional approach to the next level. Instead of sending the raw binary stream of backed-up database files, we're sending a stream of changes that were recorded in the Write-Ahead Log. It's named logical, as it understands the operations' semantics, plus the information about the tables it's replicating. It's highly flexible; it can be defined for one or multiple tables, filter records and copy a subset of data. It can inform you about changes to specific records. Thus it requires the replicated table to have primary keys.*

We'll use it to publish our notifications from the alerts table into the web UI! I'll use C#, .NET and SignalR, but you can apply this pattern to other technologies.

We'll subscribe to the changes from the alerts table (*vehicle_fuel_efficiency_avg*). That's also why we didn't make it a *hypertable*. Reminder: it uses partitioning underneath. Postgres, behind the scenes, is creating the table for each partition. Technically, it's possible to tell Postgres to replicate data from those tables, but if we add to that dynamic creation etc., things are getting harder. That's also why TimescaleDB discourages using logical replication for *hypertables*. That may change, as Postgres is investing a lot of effort to make logical replication and partitioning more aligned and easier to use together. However, for now, let's focus on our scenario. For our alerting case, it makes perfect sense, as we're interested in new records, and they're mostly ephemeral notifications.

In C#, using the example code from mentioned article we can do it as follows:

```
public record FuelEfficiencyAlert(
    int VehicleId,
    DateTime StartTime,
    DateTime EndTime,
    decimal FuelEfficiency
);

public class FuelEfficiencyAlertsPostgresSubscription
{
    public static async Task SubscribeAsync(
        string connectionString,
        IHubContext<FleetManagementHub> hubContext,
        CancellationToken ct
    )
    {
        const string slotName = "fuel_efficiency_alerts_slot";

        var dataMapper = new FlatObjectMapper<FuelEfficiencyAlert>(NameTransfor

        var subscriptionOptions = new SubscriptionOptions(
            connectionString,
            slotName,
            "fuel_efficiency_alerts_pub",
            "fuel_efficiency_alerts",
            dataMapper,
            CreateStyle.WhenNotExists
        );

        var subscription = new Subscription();

        await foreach (var alert in subscription.Subscribe(subscriptionOptions,
        {
           await FleetManagementHub.SendFuelEfficiencyAlert(hubContext, (FuelEf
        }
    }
}
```

Behind the scenes, it'll set up the publication for our table, replication slot, and subscribe for the changes.

Each time a new record appears, we'll get a notification from AsyncEnumerable and forward it to SignalR. SignalR is a .NETopen-source library that enables sending server-side notifications to client applications (e.g. web clients).

The hub is implemented and configured simply as:

```
public class FleetManagementHub : Hub
{
    public static Task SendFuelEfficiencyAlert(IHubContext<FleetManagementHub>
        hubContext.Clients.All.SendAsync("FuelEfficiencyAlertRaised", alert);
}
```

We're also using:

- Npgsql, an Open Source Postgres provider for .NET,
- NetTopologySuite a GIS solution for .NET
- their plugins *Npgsql.NetTopologySuite* and *NetTopologySuite.IO.GeoJSON4STJ* packages to handle geometry types and their (de)serialisation.

If you think that's a lot of plumbing, then look from a different angle that all of those tools are integrating easily with each other and building a great ecosystem of Open Source tooling. You can build a complex solution based on that without the struggle. It also shows the power of Postgres and proves its maturity.

Now the simple Web App subscribing to Postgres notifications and pushing it forward through SignalR can look as:

```
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.SignalR;
using NetTopologySuite.IO.Converters;
using Npgsql;
using PostgresForDotnetDev.Api;
using JsonOptions = Microsoft.AspNetCore.Http.Json.JsonOptions;

// tell Npgsql that we're using GIS coordinates
NpgsqlConnection.GlobalTypeMapper.UseNetTopologySuite(geographyAsDefault: true)

var builder = WebApplication.CreateBuilder(args);

// Enable CORS for local web app
builder.Services.AddCors(options =>
{
    options.AddPolicy("ClientPermission", policy =>
    {
        policy
            .WithOrigins("http://localhost:3000")
            .AllowAnyMethod()
            .AllowAnyHeader()
            .AllowCredentials();
    });
});

// configure serialisation of GeoJSON
void Configure(JsonSerializerOptions serializerOptions)
{
    serializerOptions.Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping;
    serializerOptions.NumberHandling = JsonNumberHandling.AllowNamedFloatingPoi

    serializerOptions.Converters.Add(new GeoJsonConverterFactory());
    serializerOptions.Converters.Add(new JsonStringEnumConverter());
}

builder.Services.Configure<JsonOptions>(o => Configure(o.SerializerOptions));
builder.Services.Configure<Microsoft.AspNetCore.Mvc.JsonOptions>(o => Configure

// Add Postgres Subscription
builder.Services.AddHostedService(serviceProvider =>
    {
        var logger =
            serviceProvider.GetRequiredService<ILogger<BackgroundWorker>>();
        var hubContext =
            serviceProvider.GetRequiredService<IHubContext<FleetManagementHub>>

        return new BackgroundWorker(
                builder.Configuration.GetConnectionString("Postgres")!,
                hubContext,
                ct
            )
        );
    }
);

// Add SignalR
builder.Services.AddSignalR();

var app = builder.Build();

app.UseCors("ClientPermission");
app.UseAuthorization();

app.UseRouting();

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

// map SignalR
app.MapHub<FleetManagementHub>("/hubs/fleet-management");

app.Run();
```

Getting notifications on React web app

And why not complete that with a quadruple axel? Let’s create a React app:

```
npx create-react-app fleet-management --template typescript
```

Add SignalR npm package:

```
npm install @microsoft/signalr
```

Then if you replace your *App.tsx* code with:

```
import "./tailwind.css";

import './App.css';
import { useEffect, useState } from "react";
import { HttpTransportType, HubConnectionBuilder, LogLevel } from "@microsoft/s

type FuelEfficiencyAlert = {
  vehicleId: number;
  startTime: Date;
  endTime: Date;
  fuelEfficiency: number;
};

function FleetManagementApp() {
  const [alerts, setAlerts] = useState<FuelEfficiencyAlert[]>([]);

  useEffect(() => {
    // kids, don't do that on prod, be better and use https
    const connection = new HubConnectionBuilder()
      .configureLogging(LogLevel.Debug)
      .withUrl("http://localhost:5000/hubs/fleet-management", {
        skipNegotiation: true,
        transport: HttpTransportType.WebSockets
      })
      .withAutomaticReconnect()
      .build();

    connection.on("fuelefficiencyalertraised", (alert: FuelEfficiencyAlert) =>
      alert.startTime = new Date(alert.startTime);
      alert.endTime = new Date(alert.endTime);
      setAlerts((prevAlerts) => [...prevAlerts, alert]);
    });

    connection.start().catch((err) => console.error(err));

    return () => {
      connection.stop();
    };
  }, []);
```

```
      <div className="mx-auto max-w-5xl px-6 py-4">
        <h1 className="text-3xl font-bold mb-4">Fleet Management App</h1>
        {alerts.length === 0 ? (
          <div className="text-lg">There are no alerts at the moment.</div>
        ) : (
          <div className="grid grid-cols-3 gap-4">
            {alerts.map((alert) => (
              <div
                key={`${alert.vehicleId}-${alert.startTime.toISOString()}`}
                className="bg-white rounded-lg shadow p-4"
              >
                <div className="text-lg font-bold mb-2">
                  Alert for Vehicle {alert.vehicleId}
                </div>
                <div className="text-sm mb-2">
                  Start Time: {alert.startTime.toLocaleString()}
                </div>
                <div className="text-sm mb-2">
                  End Time: {alert.endTime.toLocaleString()}
                </div>
                <div className="text-sm">Fuel Efficiency: {alert.fuelEfficiency}<
              </div>
            ))}
          </div>
        )}
      </div>
    );
  }

  export default FleetManagementApp;
```

Et voilà! We just made our alerts from the database into the UI. All without resource-consuming polling and using fancy but practical Postgres features!

I hope this article shows you how extensible and powerful Postgres is and that it can give you a lot of fun and real help to deliver your business features in production-grade quality!

Want to see full code? Check my repositories:

- Postgres for .NET developer
- Postgres Outbox Pattern with CDC and .NET

Cheers!

Oskar

p.s. **Ukraine is still under brutal Russian invasion. A lot of Ukrainian people are hurt, without shelter and need help.** You can help in various ways, for instance, directly helping refugees, spreading awareness, putting pressure on your local government or companies. You can also support Ukraine by donating e.g. to Red Cross, Ukraine humanitarian organisation or donate Ambulances for Ukraine.

👋 **If you found this article helpful** and want to get notification about the next one, **subscribe to Architecture Weekly.**

✉️ **Join over 2600 subscribers**, get the best resources to boost your skills, and stay updated with Software Architecture trends!

Architecture Weekly

Weekly Software Architecture resources to boost your knowledge and developer skills.

Type your email… Subscribe

substack

SHARE

Oskar Dudycz

For over 15 years, I have been creating IT systems close to the business. I started my career when StackOverflow didn't exist yet. I am a programmer, technical leader, architect. I like to create well-thought-out systems, tools and frameworks that are used in production and make people's lives easier. I

believe Event Sourcing, CQRS, and in general, Event-Driven Architectures are a good foundation by which this can be achieved.

← Event stores are key-value databases, and why that matters
2023-04-07

© License Creative Commons BY-SA 4.0 to Oskar Dudycz 2020 - 2023 •