Store Todos in a database

# Use PostgresNIO to store your Todos in a Postgres database

Now we have a working API and a way to test it, lets look into storing our todos in a Postgres database with PostgresNIO.

15mins

**Estimated Time**

Section 1

## Setup your Postgres database

Setup a Postgres database to use with the Todos application.

Install Postgres

No Preview

```
> brew install postgresql
```

**Step 1**

You'll need to install postgres on your system if you don't already have it. Detailed instructions on installing Postgres can be found here.

---

Once you have installed Postgres follow the instructions on screen to start your Postgres database service.

**Step 2**

The Postgres install comes with `psql` the commandline interface to Postgres. We are going to use this to create a new database and a new role.

---

Note the SQL commands all end in a semi-colon. The `\c` command connects to a database and the `\q` command quits `pqsl`. You can find out more about `psql` here.

**Step 3**

We return to our project…

**Step 4**

And add PostgresNIO as a dependency

**Step 5**

In Sources/App/Application+build.swift…

**Step 6**

we add a new requirement `inMemoryTesting` to `AppArguments`. This will decide whether we store Todos in memory or a Postgres database.

Step 7

We then need to add implementations of this requirement in Sources/App/App.swift

Step 8

and Tests/AppTests/AppTests.swift

Step 9

We are going to use `PostgresClient` from PostgresNIO for our Postgres support. The `inMemoryTesting` flag is used to decide on whether we should set one up. Note the Postgres configuration details are the same as the Postgres role we set up earlier in psql.

Step 10

`PostgresClient` sets up background processes that requires lifecycle management. You can add a service to `Application` to have its lifecycle managed as long as it conforms to `Service`. This is done by adding it to an internally held `ServiceGroup`. More details on `Service` and `ServiceGroup` can be found in the documentation for Swift Service Lifecycle.

Section 2

## Setup a Postgres repository

Implement a version of `TodoRepository` that uses `PostgresClient`.

Sources/App/Repositories/TodoPostgresRepository.swift

No Preview ↙

```swift
import Foundation
import PostgresNIO
```

Step 1

We start our Postgres support by creating a type conforming to `TodoRepository` that uses `PostgresClient` from PostgresNIO. The functions are filled out with dummy code just now so the project will compile.

```

struct TodoPostgresRepository: TodoRepository {
    let client: PostgresClient
    let logger: Logger

    /// Create todo.
    func create(title: String, order: Int?, urlPrefix: String) async thr
        .init(id: UUID(), title: "", url: "")
    }
    /// Get todo.
    func get(id: UUID) async throws -> Todo? { nil }
    /// List all todos
    func list() async throws -> [Todo] { [] }
    /// Update todo. Returns updated todo if successful
    func update(id: UUID, title: String?, order: Int?, completed: Bool?)
    /// Delete todo. Returns true if successful
    func delete(id: UUID) async throws -> Bool { false }
    /// Delete all todos
    func deleteAll() async throws {}
}
```

Step 2

If we are going to be saving our todos to a database we are going to need a table to store them in.

---

I won't go into any great detail about the SQL calls. That is not the purpose of this tutorial. We will cover how you construct, send calls and parse their results with `PostgresClient` as we proceed through the tutorial.

Step 3

Return to `buildApplication(_:)` in Application+build.swift…

Step 4

Use the newly created `TodoPostgresRepository` and once the `PostgresClient` is running call `createTable`.

Step 5

Update `buildRouter(_:)` to take the repository as an argument and pass it to the controller.

Step 6

Back to TodoPostgresRepository.swift to start implementing our repository methods.

Step 7

To run a SQL query call `PostgresClient.query(_:logger:)` with the query string and a Logger`.

---

Wait a sec! If you look closer that query looks like it's got SQL injection. That's a classic security issue. Except this isn't the case here. The object being constructed is not a `String` but a `PostgresQuery` which uses `String Interpolation` to create parameter bindings for all the interpolated variables.

Step 8

The `get` method demonstrates how you get data returned from a query. The query returns a sequence of rows. You extract the data from the row by decoding it as a tuple. In this case there should only be one row so we return immediately as soon as we have it.

Step 9

`list` is very similar to `get`. Except there is no `WHERE` clause in the SQL and we return all of the rows returned from the query instead of just the first.

Step 10

`patch` has a complication where we only want to include the non optional values in the `UPDATE` query otherwise we'll be setting database columns to null. You could do this dynamically and build a `PostgresQuery.StringInterpolation` bit by bit but it is safer just to provide the full query strings for each situation.

**Step 11**

And finally the `delete` and `deleteAll` functions. This completes the implementation of the Postgres todos repository.

**Step 12**

If you go to Tests/AppTests/AppTests.swift…

**Step 13**

You can switch the `inMemoryTesting` boolean to false to test your Postgres solution.

**Step 14**

That’s us done, we have a working and tested Todos application.

---

The code for this tutorial can be found in the hummingbird-examples repository.