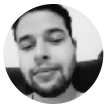
REST API with MongoDB and F# on .NET Core



Leo Cavalcante Follow

Sep 29, 2018 · 5 min read

I have just a couple years in functional programming, just a couple weeks in F# and no background or experience with .NET, so don't take this as a reference from an expert, it's just me documenting what I've learned so far.

You can start by building a very raw-vanilla-bare-bones HTTP server using the HttpListener from .NET, **but is very low-level** in terms of having few abstractions, like: you have to write on stream buffers to output something.

There is ASP.NET Core, **but is is very object-oriented**, it's constructions aren't functional and hey, we're using F# for a reason, right?

Freya seams to be an awesome purely functional high-level way to go, **but few maintainers and months without updates**.

Introducing Giraffe

Then there is Giraffe, the survival of my biased criteria above, it's **high-level, functional-first and actively maintained!**

You can start by using it's dotnet-new template, but I like a greenfield so let's get started from scratch:

```
λ mkdir Todos
λ cd Todos
λ dotnet new console -lang F#
```

We're starting using the *console* template, it's the simplest I've found, it will put just a minimal **.fsproj** file. You can run it to make sure everything is fine:

```
λ dotnet run
Hello World from F#!
```

Let's add our dependencies. Giraffe runs on top of ASP.NET Core and we'll be using the C#'s MongoDB driver, but don't worry, it's awesome for F# too.

```
λ dotnet add package Microsoft.AspNetCore.App
λ dotnet add package Giraffe
λ dotnet add package MongoDB.Driver
```

Their are all coming from NuGet.

Let's change our Hello World to print on a Request:

```
open System
open Microsoft.AspNetCore.Builder
open Microsoft.AspNetCore.Hosting
open Microsoft.Extensions.DependencyInjection
open Giraffe

let routes =
    choose [
        route "/" >=> text "Hello World from F#!" ]

let configureApp (app : IApplicationBuilder) =
    app.UseGiraffe routes

let configureServices (services : IServiceCollection) =
    services.AddGiraffe() |> ignore
```

To start our development server:

```
λ dotnet run
```

You'll see a message telling you that it's running on http://localhost:5000 head over to this URL on your favorite web browser and you should be seeing: Hello World from F#!

We have a running server! Now what?

We're going to make this simple

Todos have a text telling what to-do and a boolean flag telling if it's already done, despite a unique identifier, of course:

```
namespace Todos

type Todo =
  { Id: string
    Text: string
```

Framing our CRUD endpoints

Despite serving "Hello World" texts, of couse, Giraffe can delegate HTTP endpoints to function handlers defined as `HttpFunc -> HttpContext -> HttpFuncResult`:

```
1  namespace Todos.Http
2
3  open Giraffe
4  open Microsoft.AspNetCore.Http
5
6  module TodoHttp =
7    let handlers : HttpFunc -> HttpContext -> HttpFuncResult
8      choose [
9        POST >=> route "/todos" >=>
10         fun next context ->
11           text "Create" next context
12
13       GET >=> route "/todos" >=>
14         fun next context ->
15           text "Read" next context
16
```

We also need to update our `Program.fs` to include this handlers to our routes:

```
open System
open Microsoft.AspNetCore.Builder
open Microsoft.AspNetCore.Hosting
open Microsoft.Extensions.DependencyInjection
open Giraffe
open Todos.Http

let routes =
  choose [
    TodoHttp.handlers ]

let configureApp (app : IApplicationBuilder) =
  app.UseGiraffe routes

let configureServices (services : IServiceCollection) =
  services.AddGiraffe() |> ignore

```

Start the server with `dotnet run` and try it!

InMemory Todos

Before we put our hands on MongoDB, we could make a proof-of-concept using and very simple in-memory representation of what would be a Todo repository and how to operate our CRUD functions on it.

Also, it's a way to make sure our domain isn't too tied to any database vendor, we could replace MongoDB by any other type of storage just by re-implementing the function signatures to the new database.

Augment the `Todo.fs` with the following types:

```
namespace Todos

type Todo =
  { Id: string
    Text: string
    Done: bool
  }

type TodoSave = Todo -> Todo

```

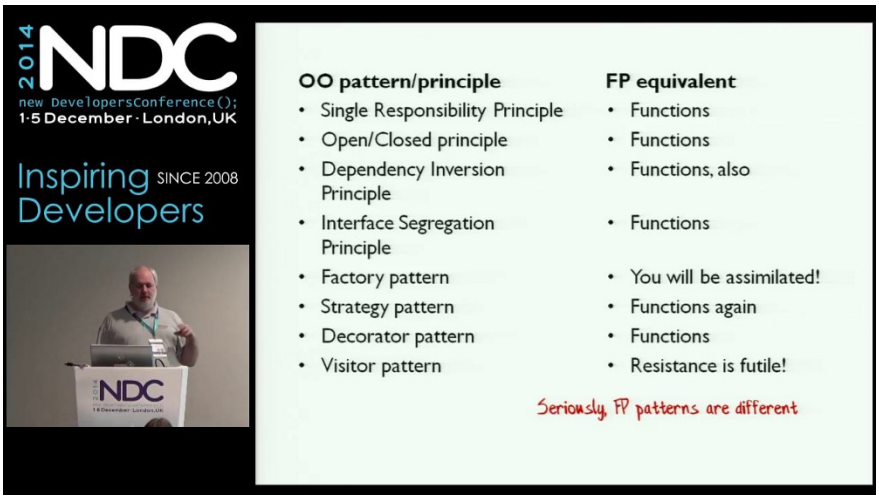
It isn't doing so much, but think on this like OO interfaces. The big deal here is that we're not going to reinvent the wheel and instead use the awesome Dependency Injection constructs that ASP.NET Core already provides.

For example, our HTTP GET endpoint will look like:

```
GET >=> route "/todos" >=>
  fun next context ->
    let find = context.GetService<TodoFind>()
    let todos = find TodoCriteria.All
```

We're relying on the `TodoFind` "interface" and for the endpoint, the implementation doesn't matter, it could be in-memory, MongoDB or any other, it only knows that it receives a `TodoCriteria` and returns a `Todo[]` (which then Giraffe auto-magically parses to JSON through the `json` function).

Yes, we're doing Dependency Inversion with Functions!



Functional Programming Design Patterns by Scott Wlaschin

Let's implement the C and R parts of the CRUD in-memory with the help of a `Hashtable`:

```
1  module Todos.TodoInMemory
2
3  open Todos
4  open Microsoft.Extensions.DependencyInjection
5  open System.Collections
6
7  let find (inMemory : Hashtable) (criteria : TodoCriteria) :
```

You probably noticed that `find` here is slight different from `TodoFind` it is `Hashtable -> TodoCriteria -> Todo[]`. That is because we need to share the same `Hashtable` along with other functions, but no worry and thanks to Currying we can give a `Hashtable` to `find` and it returns a perfect fine `TodoFind`. We can inject it like a Singleton through `configureServices`:

```
1  let configureServices (services : IServiceCollection) =
2    services.AddGiraffe() |> ignore
3    services.AddSingleton<TodoFind>(TodoInMemory.find(Hashtabl
```

You got it, right? We are defining our `TodoFind` with the **"hashtabled"** `find` function from `TodoInMemory`. Let's implement the `TodoSave`:

```
module Todos.TodoInMemory

open Todos
open Microsoft.Extensions.DependencyInjection
open System.Collections

let find (inMemory : Hashtable) (criteria : TodoCriteria) :
  match criteria with
  | All -> inMemory.Values |> Seq.cast |> Array.ofSeq
```

Then let's inject it to our services, but don't forget we have to share the `Hashtable` so we'll be finding and saving on the save place in-memory:

```
let configureServices (services : IServiceCollection) =
  let inMemory = Hashtable()

  services.AddGiraffe() |> ignore
  services.AddSingleton<TodoFind>(TodoInMemory.find inMemory
```

We can make things better here. Note that `IServiceCollection` got an `AddGiraffe` method? No, Giraffe isn't built-in on .NET, it used the power of Type Extensions and we can do this to for `TodoInMemory`:

```
module Todos.TodoInMemory

open Todos
open Microsoft.Extensions.DependencyInjection
open System.Collections

let find (inMemory : Hashtable) (criteria : TodoCriteria) :
  match criteria with
  | All -> inMemory.Values |> Seq.cast |> Array.ofSeq

let save (inMemory : Hashtable) (todo : Todo) : Todo =
  inMemory.Add(todo.Id, todo) |> ignore
```

We aren't avoiding a call to `AddSingleton` to every function signature, but at least it's well placed on the same module, is very clear and easy to reason about. We can use `AddTodoInMemory` just like `AddGiraffe`:

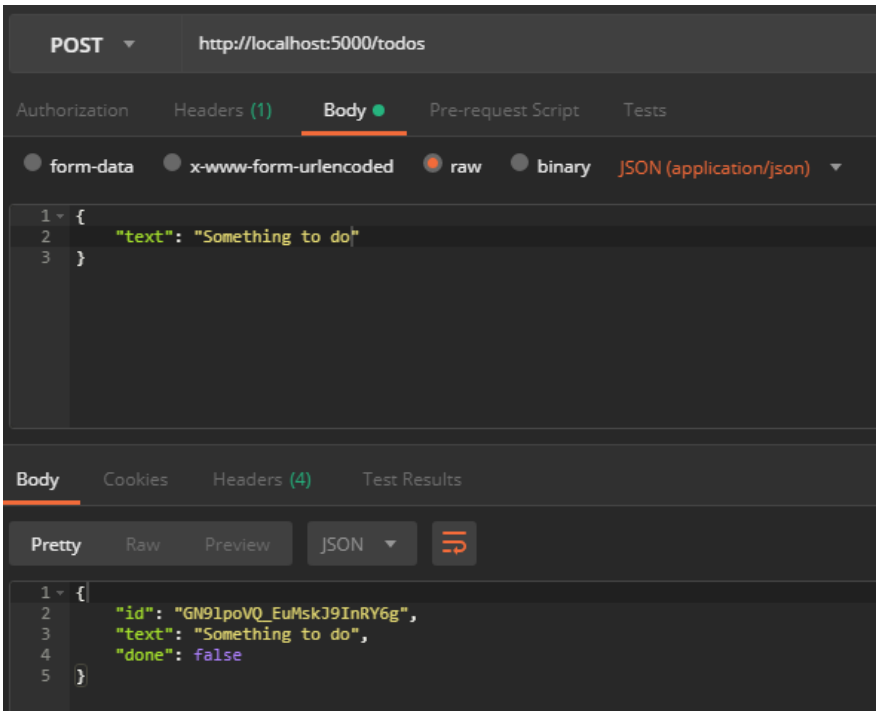
```
let configureServices (services : IServiceCollection) =
  services.AddGiraffe() |> ignore
  services.AddTodoInMemory(Hashtable()) |> ignore
```

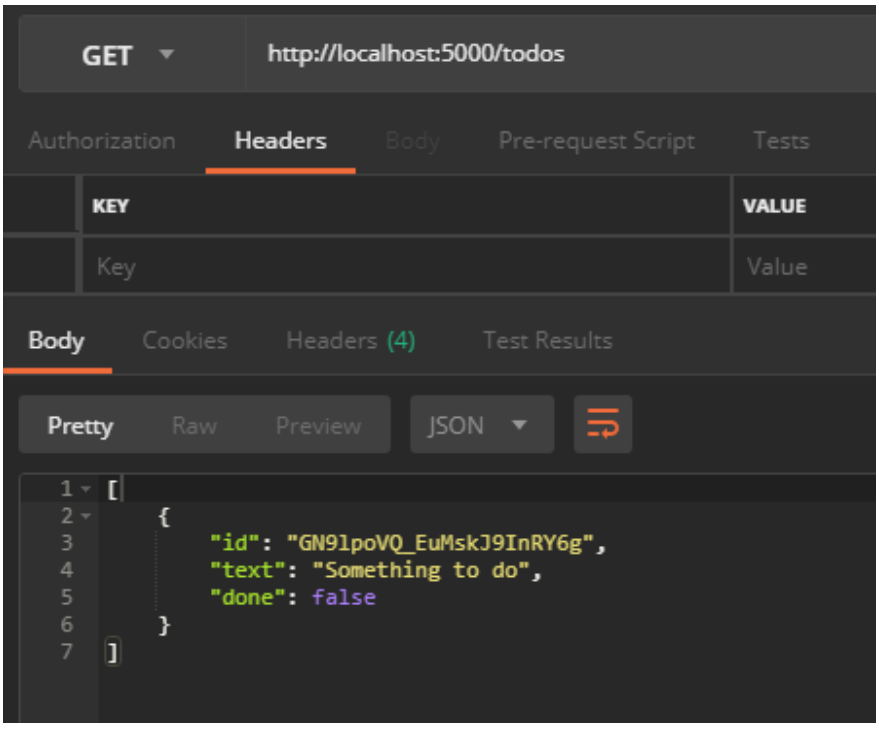
Later it could be AddTodoMongoDB, for example.

Now we should update `TodoHttp` to retrieve and call our functions, we already saw how GET looks like, here is the full file with the POST modification:

```
namespace Todos.Http

open Giraffe
open Microsoft.AspNetCore.Http
open Todos
open FSharp.Control.Tasks.V2
open System

module TodoHttp =
  let handlers : HttpFunc -> HttpContext -> HttpFuncResult
    choose [
      POST >=> route "/todos" >=>
        fun next context ->
          task {
            let save = context.GetService<TodoSave>()
            let! todo = context.BindJsonAsync<Todo>()
            let todo = { todo with Id = ShortGuid.fromGuid(
            return! json (save todo) next context
          }

      GET >=> route "/todos" >=>
        fun next context ->
```

We're already able to save and find some Todos!





Remember, it's all in-memory on that mutable Hashtable, if we restart the server everything is gone.

Implementing MongoDB!

First our final type `TodoDelete` for Delete, we're going to use `TodoSave` for Create and Update:

```
1  namespace Todos
2
3  type Todo =
4    { Id: string
5      Text: string
6      Done: bool
7    }
8
9  type TodoCriteria =
10   | All
11
```

Not much difference from in-memory, we going to replace the `Hashtable` by a `IMongoCollection` and refactor our save to know when to insert or update, also implement `TodoDelete` :

```
module Todos.TodoMongoDB

open Todos
open MongoDB.Driver
open Microsoft.Extensions.DependencyInjection

let find (collection : IMongoCollection<Todo>) (criteria :
  match criteria with
  | TodoCriteria.All -> collection.Find(Builders.Filter.Emp

let save (collection : IMongoCollection<Todo>) (todo : Todo
  let todos = collection.Find(fun x -> x.Id = todo.Id).ToEn

  match Seq.isEmpty todos with
  | true -> collection.InsertOne todo
  | false ->
    let filter = Builders<Todo>.Filter.Eq((fun x -> x.Id),
    let update =
      Builders<Todo>.Update
        .Set((fun x -> x.Text), todo.Text)
        .Set((fun x -> x.Done), todo.Done)

```

And, as expected, `TodoHttp` has no changes to Create and Read, let's just implement Update and Delete:

```
namespace Todos.Http

open Giraffe
open Microsoft.AspNetCore.Http
open Todos
open FSharp.Control.Tasks.V2
open System

module TodoHttp =
  let handlers : HttpFunc -> HttpContext -> HttpFuncResult
    choose [
      POST >=> route "/todos" >=>
        fun next context ->
          task {
            let save = context.GetService<TodoSave>()
            let! todo = context.BindJsonAsync<Todo>()
            let todo = { todo with Id = ShortGuid.fromGuid(
            return! json (save todo) next context
          }

      GET >=> route "/todos" >=>
        fun next context ->
          let find = context.GetService<TodoFind>()
          let todos = find TodoCriteria.All
          json todos next context
```

And we replace services, from `AddTodoInMemory` to `AddTodoMongoDb`. Of course, we'll need a MongoDB client connection and a MongoDB database:

```
let configureServices (services : IServiceCollection) =
  let mongo = MongoClient (Environment.GetEnvironmentVariabl
  let db = mongo.GetDatabase "todos"

  services.AddGiraffe() |> ignore
```

I'm assuming you have a running MongoDB server somewhere and we're getting the connection string from the environment variables, so before you `dotnet run` make sure o have this variables or set/export

one. In my case I have MongoDB on localhost, so: `export MONGO_URL=mongodb://localhost:27017/` .

That is it! We have a functional-first, decoupled HTTP Rest API with F# and MongoDB. As said before, I'm new to all of this, so I'd love to hear some feedback from experienced folks and if this is new to you too, don't mind to comment your questions, let's figure it out together.

Thanks!