# SqlFun

F#, SQL and lot of fun

Lambda Days 2022

# Rationale

# No fp-friendly data access tools for .NET

- Entity Framework
  - Object oriented mindset
  - Inefficiency
- Dapper
  - Stringly typed
- FSharp.Data.SqlClient
  - Low modularity
  - Low configurability
  - MS SQL only
- Neither of them offers remedy for side-effects

# Data access mechanisms in overall application architecture

- Every serious architectural pattern describes a data access as a separate component/layer/service/etc...
- Maybe we should assume, that isolation of data access mechanisms is not an advice, but rather it is some kind of requirement, and utilize it some way?

# E.g. like Insight.Database

```
public interface IBlogging
{
    [Sql("GetBlog")]
    Blog GetBlog(int id);

    [Sql(@"select id, title, content from Post
           where blogId = @blogId")]
    List<Post> GetPosts(int blogId);

    [Sql(@"select * from post
           where id = @id;
           select * from comment
           where postId = @id")]
    [Recordset(1, typeof(Comment), IsChild = true)]
    PostDetails GetPostDetails(int id);
}
```

# Foundations

# How to represent an SQL query in functional language?

```
select id,
       name,
       title,
       description,
       owner,
       createdAt,
       modifiedAt,
       modifiedBy
from Blog
where id = @id
```

# Of course, as a function :-)

```
val getBlog: IDbConnection -> int -> Blog
```

Or even:

```
val getBlog: IDbConnection ->
             IDbTransaction option ->
             int ->
             Blog
```

And when we glue them together, because they doesn't make sense separately:

```
val getBlog: DataContext -> int -> Blog
```

# Parameters

Simple

```
val getBlog: DataContext -> int -> Blog
```

Complex

```
val insertPost: DataContext -> Post -> int
```

Tupled

```
val updatePost: DataContext -> Post * int -> unit
```

Curried

```
val updatePost: DataContext -> Post -> int -> unit
```

# Result

Nothing

```
val updateBlog: DataContext -> Blog -> unit
```

Simple value

```
val insertPost: DataContext -> Post -> int
```

Complex value

```
val getBlog: DataContext -> int -> Blog
```

List of simple values

```
val getTags: DataContext -> int -> string list
```

List of complex values

```
val getPosts: DataContext -> int -> Post list
```

# It can be implemented this way

```
let getBlog (ctx: DataContext) (id: int): Blog =
    ctx.Query "select id,
                      name,
                      title,
                      description,
                      owner,
                      createdAt,
                      modifiedAt,
                      modifiedBy
               from Blog
               where id = @id" id
```

# Or even better this way

```
let getBlog =
    sql<DataContext -> int -> Blog>
        "select id,
                name,
                title,
                description,
                owner,
                createdAt,
                modifiedAt,
                modifiedBy
         from Blog
         where id = @id"
```

# More conveniently

```
let getBlog: DataContext -> int -> Blog =
    sql "select id,
                name,
                title,
                description,
                owner,
                createdAt,
                modifiedAt,
                modifiedBy
           from Blog
           where id = @id"
```

# And sql is...

```
val sql<'q>: string -> 'q
```

A function generating function executing query by:

- Calling the query in the SchemaOnly mode
- Using System.Linq.Expressions to generate code
- Generates all required conversions
- Validates parameters and results
- Doesn’t cache generated code – the target variable is perfectly enough

# Query functions are grouped in modules

```
module Blogging =

    let getBlog: DataContext -> int -> Blog =
        sql "..."

    let getPosts: DataContext -> int -> Post list =
        sql "..."

    let getPostDetails: DataContext -> int ->
                        PostDetails =
        sql "..."
```

# To validate all functions in module

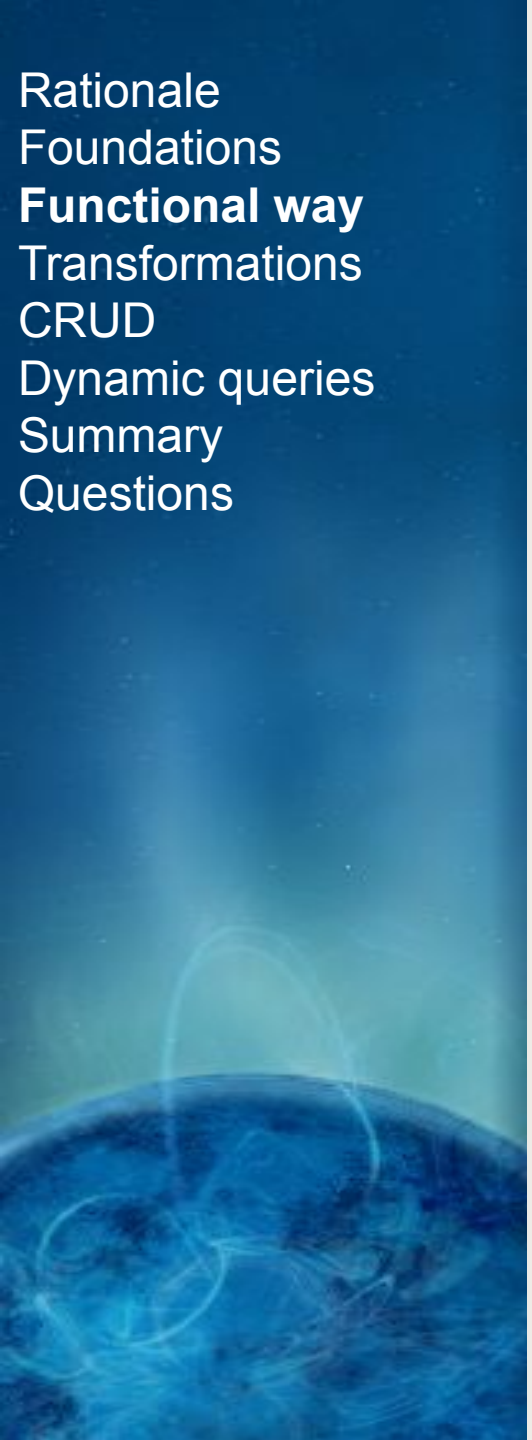
```
[<TestFixture>]
type BloggingTests =

    [<Test>]
    member this.QueriesAreValid() =
        Blogging.getBlog |> ignore
```

It’s enough to access one function in module. All functions will be instantiated.

# Strenghts

- Whole sql is available
  - Since we don’t use any DSL…
- Semi-automatic validation
  - One test per module, without executing queries, without test data, etc...
- Performance
  - No *runtime* reflection

# Functional way



# F-R impedance mismatch

- Every database access is side-effect
- Some very usual elements of data access API are stateful, e.g. connection

# Async is almost Haskell IO

```
let getBlog: DataContext -> int -> Blog Async =
    sql "select id,
                name,
                title,
                description,
                owner,
                createdAt,
                modifiedAt,
                modifiedBy
           from Blog
           where id = @id"
```

# Better, but...

```
async {
    use ctx = getDataContext()
    let! blog = getBlog ctx id
    return blog
}
```

Connection as a stateful component, breaks composability.

# Database connection is kind of dependency

Something between

*Dependency injection is fancy name for passing argument*

and

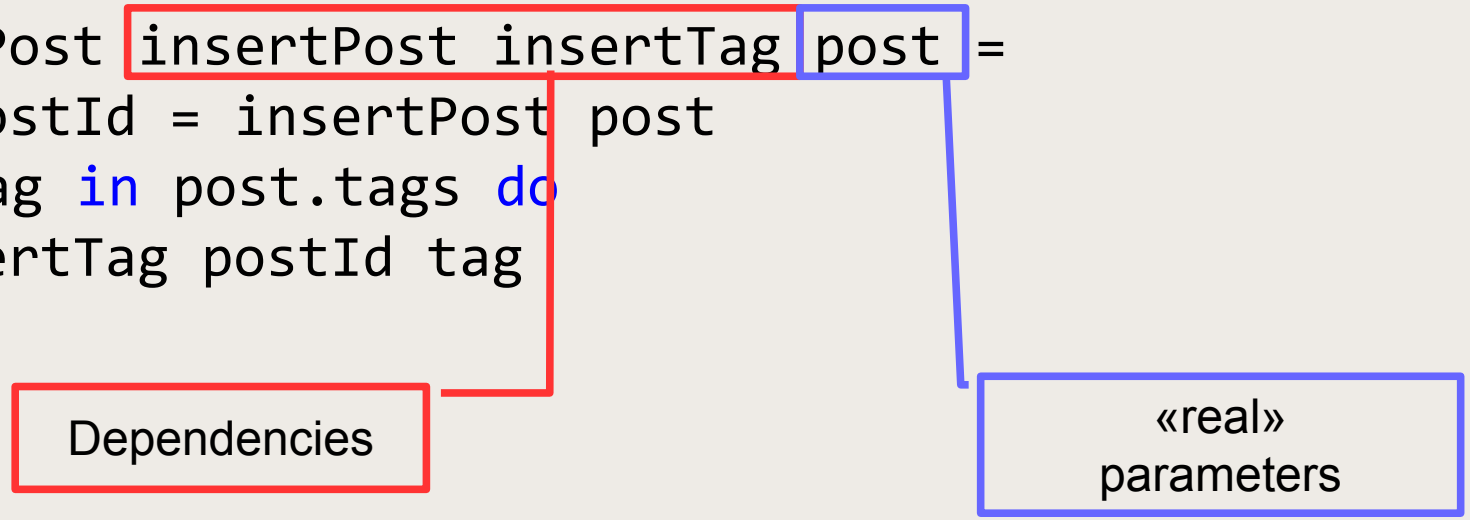
*Use Free monad which allows you to build a monad from any functor*

# Dependency injection in functional languages

```
module Posts =

  let addPost post =
    let postId = Data.insertPost post
    for tag in post.tags do
      Data.insertTag postId tag
```

# Dependencies as function parameters and partial application

```
module Posts =

  let addPost insertPost insertTag post =
    let postId = insertPost post
    for tag in post.tags do
      insertTag postId tag
```



# Resolving dependencies by partial application

```
module CompositionRoot =

  let addPost = Posts.addPost Data.insertPost
                              Data.insertTag
```

# Currying as a dependency management mechanism

- Good for pure functions
- For dependencies making side effects,  async can be used
- Stateful dependencies can not be resloved statically (in additional module)
- (*unless we accept singletons*)

# Stateful dependency problem

```
module CompositionRoot =

  let ctx = createDataContext()

  let insertPost = Data.insertPost ctx

  let insertTag = Data.insertTag ctx

  let addPost = Posts.addPost insertPost insertTag
```

# What about placing stateful parameters at the end?

```
let insertPost: Post -> DataContext -> int Async =
  sql "insert into post
         (blogId, name, title, content,
          author, createdAt, status)
       values
         (@blogId, @name, @title, @content,
          @author, @createdAt, @status);
       select scope_identity()"
```

# Connection management can be encapsulated in one function

```
let run (f : DataContext -> 't Async) =
    async {
        use ctx = createDataContext()
        return! f ctx
    }


async {
   let! id = insertPost post |> run
   return id
}
```

# But how to call many functions on one connection?

```
fun ctx -> async {
    let! postId = insertPost post ctx
    do! insertTags postId tags ctx
}
|> run
```

# Reader monad

Having the type `M<'t> = DataContext -> 't` we can define operations:

```
val bind M<'t1> -> 't1 -> M<'t2> -> M<'t2>
val return 't -> M<'t>
```

Furthermore, M fulfills some rights.
Shortly – M is a monad.

# Async + Reader and computation expression

```
type AsyncDb<'t> = DataContext -> 't Async

asyncdb {
    let! postId = insertPost post
    do! insertTags postId tags
}
|> run
```

Executing transaction is also easy:

```
asyncdb {
    let! postId = insertPost post
    do! insertTags postId tags
}
|> inTransaction
|> run
```

# With partial application

```
module Posts =

  let addPost insertPost insertTag post =
    asyncdb {
      let! postId = insertPost post
   for tag in post.tags do
        do! insertTag postId tag
    }

module CompositionRoot =

  let addPost = Posts.addPost Data.insertPost
                              Data.insertTag

module Facade =

  let addPost post = CompositionRoot.addPost post |> run
```

# Result transformations

# What if result is hierarchical?

```
let getPost: int * DataContext -> Post =
    sql "select * from Post
         where id = @id;
         select * from Comment
         where postId = @id"
    >> (fun (p, c) -> { p with comments = c })
```

# With AsyncDb

```
let getPost: int -> AsyncDb<Post> =
    sql "select * from Post
         where id = @id;
         select * from Comment
         where postId = @id"
    >> AsyncDb.map (fun (p, c) -> { p with comments = c })
```

# More complex case

```
let getPosts: int -> AsyncDb<Post list> =
    sql "select * from Post
         where blogId = @blogid;
         select * from Comment c
             join Post p on c.postId = p.id
         where p.blogId = @blogid"
    >> AsyncDb.map
         (join (fun p -> p.id)
               (fun c -> c.postId)
               (fun (p, c) -> { p with comments = c }))
```

# Having some conventions

```
let getPosts: int -> AsyncDb<Post list> =
    sql "select * from Post
         where blogId = @blogid;
         select * from Comment c
             join Post p on c.postId = p.id
         where p.blogId = @blogid"
    >> AsyncDb.map join<_,_,Comment>
```

# Transformations can be composed

```
let getPosts: int -> AsyncDb<Post list> =
    sql "select * from Post
         where blogId = @blogid;
         select * from Comment c
             join Post p on c.postId = p.id
         where p.blogId = @blogid;
       select * from Tag t
             join Post p on t.postId = p.id
         where p.blogId = @blogid"
    >> AsyncDb.map(join<_, Comment> >-> join<_, Tag>)
```

# Instead of joining, we can use grouping

```
let getPosts: int -> AsyncDb<Post list> =
    sql "select * from Post p
                join Comment c on c.postId = p.id
            where p.blogId = @blogid"
    >> AsyncDb.map group<_, Comment>
```

**Create**
**Read**
**Update**
**Delete**

# CRUD queries can be generated

```
let insertPost: Post -> AsyncDb<unit> =
    sql <| Crud.Insert<Post> ["id"]

let updatePost: Post -> AsyncDb<unit> =
    sql <| Crud.Update<Post> ["id"]

let deletePost: int -> AsyncDb<unit> =
    sql <| Crud.DeleteByKey<Post> ["id"]

let getPost: int -> AsyncDb<Post> =
    sql <| Crud.SelectByKey<Post> ["id"]
```

# Dynamic queries

# Dynamic query composition is unavoidable

```
select id,
       blogId,
       name,
       title,
       content,
       author,
       createdAt
from post
where
   (@name is null or name like '%' + @name + '%')
and
   (@startDate is null or createdAt >= @startDate)
and
   (@endDate is null or createdAt <= @endDate)
```

In this case query optimizer doesn’t use an index

# Criteria as record of options

```
type PostCriteria = {
    Title: string option
    Content: string option
    Author: string option
    CreatedAfter: DateTime option
    CreatedBefore: DateTime option
}
```

# Criteria as record of options

```
let filterPosts (criteria: PostSearchCriteria):
                AsyncDb<Post list> =
  let template = "select * from Post
                  where {{WHERE-CLAUSE}}"
  let template =
    if criteria.Title.IsSome then
      expandWhere "rs.Name like @Title + '%'" template
    else
      template
  ...
  let template = cleanUpTemplate template
  let query = buildAndMemoizeQuery template
  query criteria
```

# Little bit smarter

```
let filterPosts (criteria: PostSearchCriteria)
                : AsyncDb<Post list> =
  "select * from Post where {{WHERE-CLAUSE}}"
  |> applyWhen criteria.Title.IsSome
      (expandWhere "rs.Name like @Title + '%'")
  |> applyWhen criteria.Content.IsSome
      (expandWhere "rs.Name like '%' + @Content + '%'")
  ...
  |> cleanUpTemplate
  |> buildAndExecute criteria
```

# Continuation passing style

```
let titleStartsWith<'q>
        (phrase: string)
        (template: string)
        (next: string -> (string -> 'q)): 'q =
    let query =
      template
      |> expandWhere "title = like @title + '%'"
      |> next<string -> 'q>
    query phrase
```

Last function in chain would be responsible for generating query function and caching it.

# Continuation cannot be a function

```
let olderThan<'q>
        (creationDate: DateTime option)
        (template: string)
        (next: string -> 't): 'q =
    match creationDate with
    | Some date ->
      let query =
        template
        |> expandWhere "createdAt > @date"
        |> next<DateTime -> 'q>
      query date
    | None -> next<'q> template
```

# It can be implemented as an object

Instead of function

```
val next: string -> 'q
```

Use object with generic method

```
type IQueryPart =
  abstract member Combine: string -> 'q
```

You can’t use generic function as a parameter, but you can use object with a generic method.

# Define query part for each parameter

```
type OlderThanPart(creationDate: DateTime option,
                   next: IQueryPart) =
  interface IQueryPart with
    member Combine (template: string): 'q =
      match creationDate with
      | Some date ->
        let query =
            template
        |> expandWhere "createdAt > @date"
            |> next.Combine<DateTime -> 'q>
        query date
      | None -> next.Combine<'q> template


let olderThan date next = OlderThanPart(date, next)
```

# And we end up with nice query DSL

```
let posts =
    buildQuery
    |> olderThan startDate
    |> earlierThan endDate
    |> titleContains phrase
    |> authorIs author
    |> selectPosts
```

# Summary

# Not covered in presentation

- Configuration & customisation
- Managing different SQL dialects
- Managing unusual data types
- DB model vs application model

# Conclusions

- Building your own microORM is fun
  - Even if it’s like rediscovering a wheel
  - System.Linq.Expressions is powerful
  - Type providers are not so excellent
- Functional programming is fun
  - There are still undiscovered areas
  - There are unwritten tools as well

# Sources?

- Available on GitHub
  - https://github.com/jacentino/SqlFun

# Binaries?

- NuGet packages
  - https://www.nuget.org/packages/SqlFun
  - https://www.nuget.org/packages/SqlFun.MsSql
  - https://www.nuget.org/packages/SqlFun.MsDataSql
  - https://www.nuget.org/packages/SqlFun.NpgSql
  - https://www.nuget.org/packages/SqlFun.Oracle

# Questions