SumoDB

A clean persistence layer for Erlang/OTP

Scenario

Erlang + Databases

Scenario

In the beginning, everything was void, and without form...

The God Module

```erlang
handle_call({id, Id}, _From, State) ->

    Query = <<"select name from ", ?TABLE," where id = ?">>,
    emysql:prepare(get_name, Query),

    Result = case emysql:execute(mysql_pool, get_name, [Id]) of
        #result_packet{rows = []} -> {error, notfound};
        #result_packet{rows = Rows} -> process_results(Rows);
        Error -> lager:error("error: ~p with id ~p",[Error, Id]),{error,Error}
    end,

    {reply, Result, State};
```

The God Module

- God Module
- No clear definition of concerns and responsibilities
- Db implementation details mixed up with business logic
- Might be hard to test without setting up the whole system
- Not easily scalable in terms of business evolution and new features
- Duplicated code
- A higher potential for programming errors
- Difficulty in centralizing data-related policies such as caching
- An inability to easily test the business logic in isolation from external dependencies

But we want...

- Faster coding! :)
- Fast application prototyping
- Reuse code in different projects
- Get away as much as possible from writing and debugging SQL
- Reduce impedance mismatches between "our state representation" and the different databases
- Possibility of mixing different databases (SQL/NoSQL) with minimum effort

Let's ~~*steal*~~ *borrow* some ideas the OOP world

Patterns!

ActiveRecord

"*Active record is an approach to accessing data in a database. A database table or view is wrapped into a class. Thus, an object instance is tied to a single row in the table.*"

http://en.wikipedia.org/wiki/Active_record_pattern

ActiveRecord

"*An object carries both data and behavior. Active Record uses the most obvious approach, putting data access logic in the domain object.* "

"*The wrapper class implements accessor methods or properties for each column in the table or view.*"

http://www.martinfowler.com/eaaCatalog/activeRecord.html

ActiveRecord

```
part = new Part()

part.name = "Sample part"
part.price = 123.45

// Domain Entity "polluted" with db access
part.save()

b = part.find_first("name", "gearbox")
```

ActiveRecord

Benefits

- Easy
- Better than the god module, simple architecture, some code decoupling

Caveats

- Violates the "Single Responsibility Principle"
- Why should my domain know how to use emysql?
- Usually ties the entity name to the table name, which couples your domain to your db storage representation.
- FAT models.
- Models knows how to talk to the db (they need to find a db connection, etc)
- Most suitable for simple CRUD applications
- Hard to test without a database
- Business logic and database implementation details reside in the same code
- Can't use the same entity with a different database

Domain Entity

"An object model of the domain that incorporates both behavior and data."

"*At its worst business logic can be very complex. Rules and logic describe many different cases and slants of behavior, and it's this complexity that objects were designed to work with.* "

http://martinfowler.com/eaaCatalog/domainModel.html

DAO Pattern

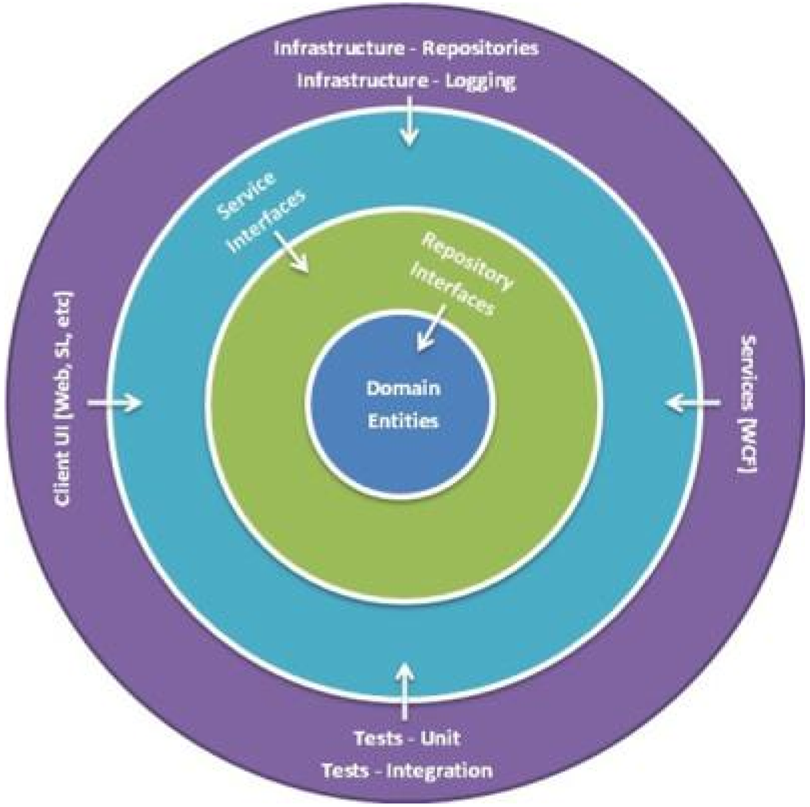
" an object that provides an abstract interface to some type of database or other persistence mechanism"

"The advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application that can and should know almost nothing of each other, and which can be expected to evolve frequently and independently. Changing business logic can rely on the same DAO interface, while changes to persistence logic do not affect DAO clients as long as the interface remains correctly implemented."

DAO is a technical abstraction for data access technology

http://en.wikipedia.org/wiki/Data_access_object

Layered Architecture



Repository Pattern

"A Repository mediates between the domain and data mapping layers..."

"Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes."

http://martinfowler.com/eaaCatalog/repository.html

Repository Pattern

"A system with a complex domain model often benefits from a layer, … that isolates domain objects from details of the database access code. In such systems it can be worthwhile to build another layer of abstraction over the mapping layer where query construction code is concentrated. In these cases particularly, adding this layer helps minimize duplicate query logic."

http://martinfowler.com/eaaCatalog/repository.html

Repository Pattern

"Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers."

http://martinfowler.com/eaaCatalog/repository.html

Repository Pattern

- Different team members can work on different layers (business logic vs data layer)
- Easy refactoring and testing for business logic and data layer.
- Easy to mock and inject different dependencies as needed.
- Layers can grow independently from each other
- Repositories can expose a well defined contract, more suitable to work with the domain.
- Multiple repositories could be combined to get the entities.
- Can easily replace a repository by another, just by respecting the interface (contract)

Sumo DB

- Persistence layer.
- By Inaka (yey!) and we use it, too
- Written in Erlang, for Erlang applications
- Nothing similar in the pure erlang world
- Kind of an ORM (if you know elixir, it's similar to ecto)
- Finally! Decouple your business logic from your state and your storage backend
- No more “god” modules

Sumo DB

- Supports MySQL
- A bit of support for MongoDB and Redis
- Planned to support sqlite, mnesia, and postgresql
- Allows to use multiple databases in your code
- Behaviors: Storage backends, repos, docs (entities). Go ahead, Implement your own :)

Sumo DB

- Repository Pattern
- Entities encapsulate state and behavior
- Repositories handle implementation details of the storage backend
- Domain events

Sumo DB

Behaviors

- sumo_doc: Entities
- sumo_backend: DB implementation details
- sumo_repo: Repositories, that contain some basic functionality and should be used and extended to add the domain specific queries

Sumo DB

NewUser = sumo_db:persist(mywebsite_user, User).

sumo_db:delete(mywebsite_user, Id).

User = sumo_db:find(mywebsite_user, Id).

Users = sumo_db:find_by(mywebsite_user, [{age, Age}]).

sumo_db:find_by(mywebsite_user, [{age, Age}], Limit, Offset).

sumo_doc

- sumo_doc:sleep/1

- sumo_doc:wake_up/1
-
- sumo_doc:schema/0

Sumo DB

```
{ sumo_db, [
  {storage_backends, [
    {my_mysql_backend, sumo_backend_mysql, [
      {username, "user"},
      {password, "pass"},
      {host, "127.0.0.1"},
      {port,3306},
      {database, "test"},
      {poolsize, 10}
    ]}
  ]},
  {repositories, [
    {users_repo, module_repo, [
      {storage_backend,my_mysql_backend},
      {workers, 10}
   ]}
 ]},
 {docs, [{users, user_repo}]}
```

Domain Events (or almost)

Example: A blog site

Thank you :)

https://github.com/inaka/sumo_db

Twitter: @inaka

We want YOU :) Come and learn Erlang with us :)
https://github.com/inaka/erlang_training