Building CQRS/ES web applications in Elixir using Phoenix

Thursday March 23rd, 2017

Presented by Ben Smith

In this talk you will discover how to build applications following domain-driven design, using the CQRS/ES pattern with Elixir and Phoenix.

I'll take you through a real-world case study to demonstrate how these principles can be applied.

What you'll learn

- Elixir, and some OTP (Open Telecom Platform).
- Command Query Responsibility Segregation and event sourcing (CQRS/ES).
- Implementing these concepts in a real-world case study.

How we'll get there

1. Concepts & building blocks:
   - Elixir and OTP.
   - CQRS/ES: aggregate roots, commands, events, event handlers, process managers, read-only projections.
2. Implementation:
   - Event store using PostgreSQL for persistance.
   - Phoenix web application.

With plenty of code samples and demos along the way.

Before we begin …

- Who has heard of Elixir?
- Who has used Elixir?
- Who has heard of Erlang?
- Who has used Erlang?
- Who has heard of Command Query Responsibility Segregation and event sourcing?
- Who has experience with CQRS/ES?

My experience with Elixir

August 2015	Phoenix web framework 1.0 released. Discovered Elixir, began reading about the language.
January 2016	Purchased Programming Elixir book. Started building first Elixir & Phoenix web app.
October 2016	Full-time Elixir development.

Erlang

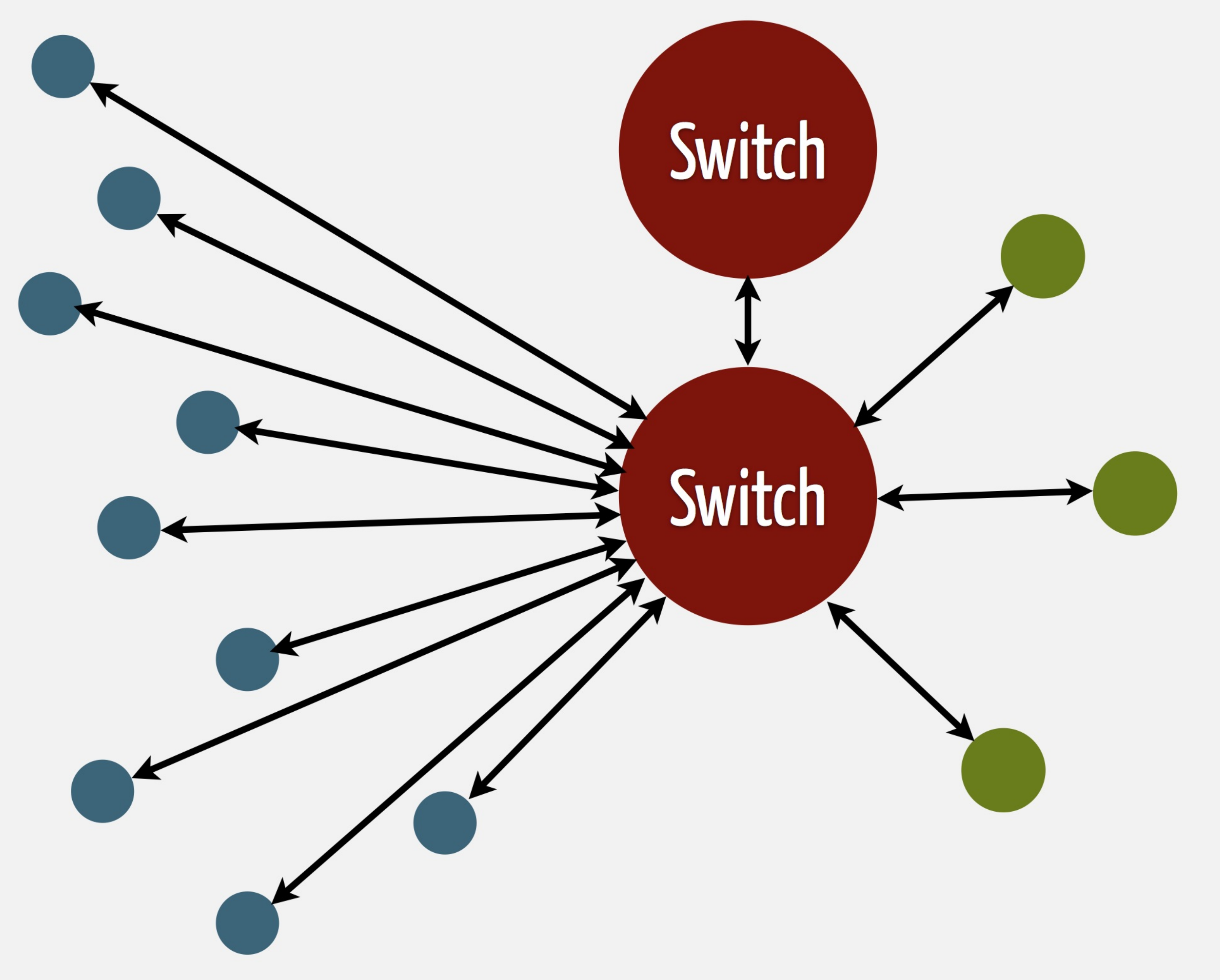
- The term Erlang is used interchangeably with Erlang/OTP, or OTP, which consists of:
  - Erlang runtime system.
  - A number of ready-to-use components, mainly written in Erlang.
  - A set of design principles for Erlang programs.
- It was originally a proprietary language within Ericsson, developed by Joe Armstrong, Robert Virding and Mike Williams in 1986.
- Released as open source in 1998.

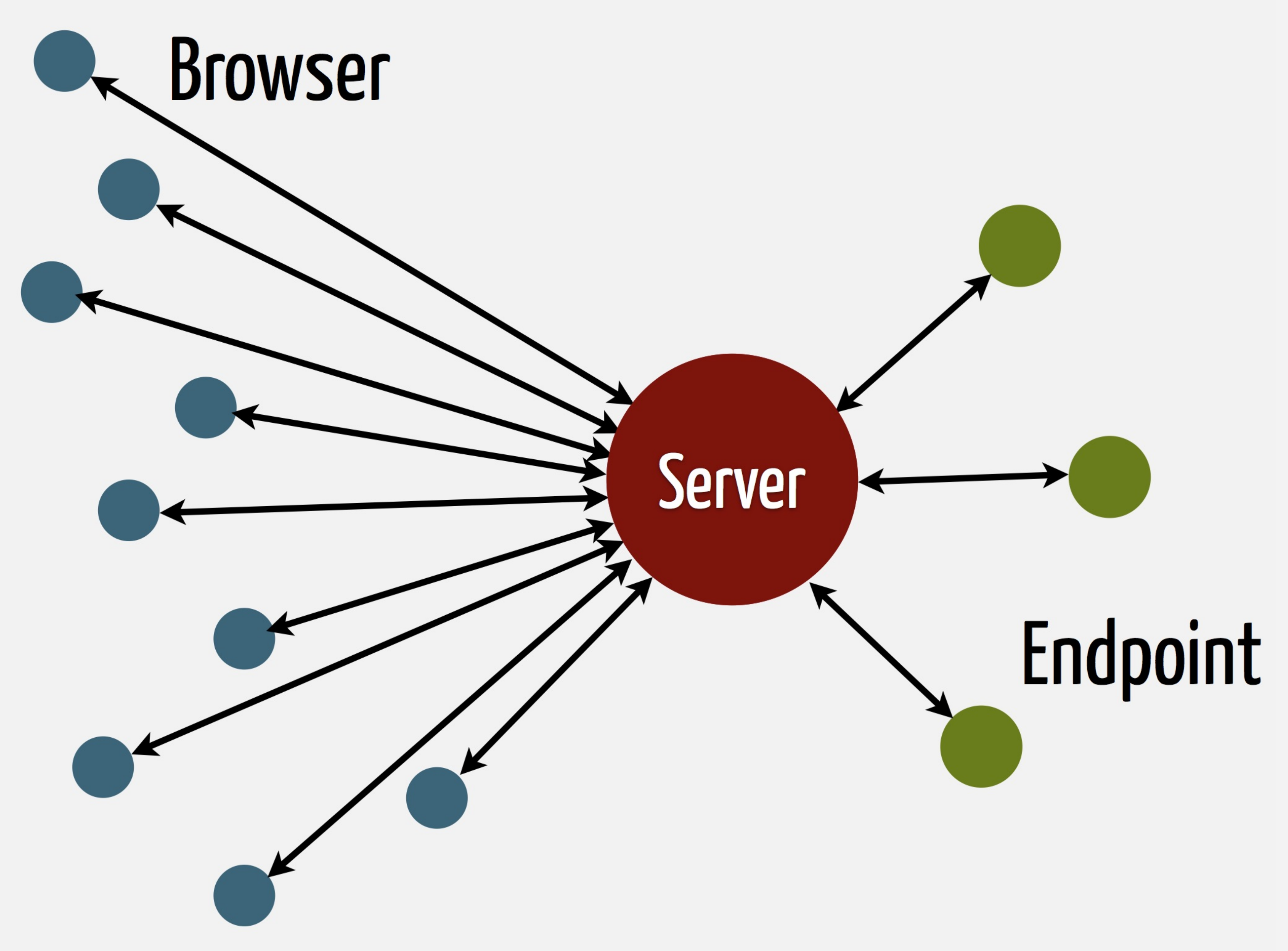
Why Erlang?

Erlang is well suited to systems that are:

- Distributed.
- Fault-tolerant.
- Soft real-time.
- Highly available, non-stop applications:
    - *nine nines* – 99.9999999% – availability.
- Hot swapping, where code can be changed without stopping a system.

Joe Armstrong — "Erlang The Movie"


Switch
Switch


Browser
Server
Endpoint

1 million is so 2011

Happy 2012 everyone!

A few months ago we published a blog post that talked about our servers doing 1 million tcp connections on a single box: http://blog.whatsapp.com/?p=170

Today we have an update for those keeping score at home: we are now able to easily push our systems to over 2 million tcp connections!

```
jkb@c123$ sysctl kern.ipc.numopensockets kern.ipc.numopensockets: 2277845
```

Best part is that we are able to do it with plenty of CPU and memory to spare and do it sustainably:

```
CPU: 37.9% user,  0.0% nice, 13.6% system,  6.6% interrupt, 41.9% idle Mem: 35G Active,
14G Inact, 18G Wired, 4K Cache, 9838M Buf, 27G Free
```

This time we also wanted to share some more technical details with you about hardware, OS and software:

```
hw.machine: amd64 hw.model: Intel(R) Xeon(R) CPU X5675 @ 3.07GHz hw.ncpu: 24
hw.physmem: 103062118400 hw.usermem: 100556451840
```

```
jkb@c123$ uname -rps FreeBSD 8.2-STABLE amd64 jkb@c123$ cat /boot/loader.conf.local
boot_verbose="" kern.hwpmc.nbuffers=32 kern.hwpmc.nsamples=64
kern.ipc.maxsockets=2400000 kern.maxfiles=3000000 kern.maxfilesperproc=2700000
kern.maxproc=16384 kern.timecounter.smp_tsc=1 net.inet.tcp.tcbhashsize=524288
net.inet.tcp.hostcache.hashsize=4096 net.inet.tcp.hostcache.cachelimit=131072
net.inet.tcp.hostcache.bucketlimit=120
```

and the last important piece of our infrastructure is Erlang:

```
8> erlang:system_info(system_version). "Erlang R14B03 (erts-5.8.4) [source] [64-bit]
[smp:24:24] [rq:24] [async-threads:0] [kernel-poll:false]\n"
```

P.S. - we are hiring in both client and server teams, so send your resume to jobs at whatsapp dot com if you are interested (.. and we are also looking for summer interns)

January 06, 2012

WhatsApp ♥ Erlang

WhatsApp architecture in 2014 when acquired by Facebook for $19 billion.

- Peaked at 2.8 million connections per server (24 CPU, 100GB RAM).
- 32 engineers — one developer supporting 14 million active users.
- 19 billion inbound & 40 billion outbound messages per day.
- More than 70 million Erlang messages per second.
- Erlang has awesome SMP scaling: nearly linear scalability.

We do not have one webserver handling 2 millions sessions.

We have 2 million webservers handling one session each.

Because we have one webserver per user we can easily make the system fault tolerant or scalable.

-- Joe Armstrong

> *Elixir is a dynamic, functional language designed for building scalable and maintainable applications.*
>
> *It leverages the Erlang VM, known for running low-latency, distributed and fault-tolerant systems.*

- First appeared in 2011 — six years old.
- Latest major release – 1.4.0 – in January 2017.

Elixir's key features

- Shared nothing concurrent programming via message passing.
- Immutable state.
- Emphasis on recursion and higher-order functions instead of side-effect-based looping.
- Pattern matching.
- Macros and meta-programming.
- Polymorphism via a mechanism called protocols.
- *"Let it crash"*

Elixir takes full advantage of OTP

- Elixir can call Erlang code, and vice versa, without any conversion cost at all.
- Elixir can use Erlang libraries.
- OTP behaviours:
  - `GenServer`
  - `Supervisor`
  - `Application`

Behaviours provide a module of common code — the wiring and plumbing — and a list of callbacks you must implement for customisation.

Elixir's promise

- Elixir **does not** promise:
  - That your code will scale horizontally.
  - Fault tolerant systems with no effort.
- But promises to give you:
  - Tools that you can employ to make it happen.
  - Patterns and building blocks of reliability instead.

Elixir's concurrency model

Elixir code runs inside lightweight threads of execution, called **processes**.

Processes are also used to hold state.

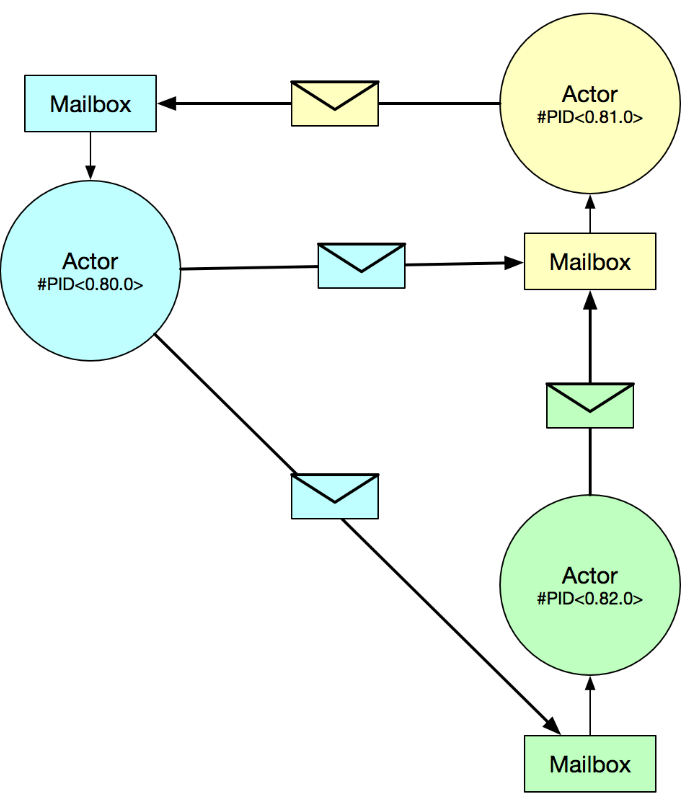
A process in Elixir is *not the same as an operating system process*. Instead, it is extremely lightweight in terms of memory and CPU usage.

An Elixir application may have tens, or even hundreds of thousands of processes running concurrently on the same machine.

A single Elixir process is analogous to the event loop in JavaScript.

Actor model

- An Actor has a mailbox.
- Actors communicate with other Actors by sending them immutable messages.
- Messages are put into the Actor’s mailbox.
- When an Actor’s mailbox has a message, code is run with that message as an argument. This code is called serially.
- When an Actor encounters an error, it dies.



Message passing

Processes are isolated and exchange information via messages:

```
current_process = self()

# spawn a process to send a message
spawn_link(fn ->
  send(current_process, {:message, "hello world"})
end)

# block current process until the message is received
receive do
  {:message, message} -> IO.puts(message)
end

# ... or flush all messages
flush()
```

Processes encapsulate state

State is held by a process while it runs an infinite receive loop:

```
defmodule Counter do
  def start(initial_count) do
    loop(initial_count)
  end

  defp loop(count) do
    new_count = receive do
      :increment -> count + 1
      :decrement -> count - 1
    end

    IO.puts(new_count)

    loop(new_count)
  end
end
```

```
iex> counter = spawn(Counter, :start, [0])
#PID<0.184.0>
iex> send(counter, :increment)
1
iex> send(counter, :increment)
2
iex> send(counter, :increment)
3
iex> send(counter, :decrement)
2
```

OTP behaviours

1. **GenServer**
2. Supervisor
3. Application

GenServer

- Generic client-server OTP behaviour.
- A GenServer is implemented in two parts:
    1. Client API
    2. Server callbacks
- The client and server run in separate processes.
- The client passes messages back and forth to the server as its functions are called.
- GenServer is a loop that handles one request per iteration passing along its updated state.

Example GenServer

```
defmodule ExampleServer do
  use GenServer

  def start_link do
    GenServer.start_link(ExampleServer, :ok, [])
  end

  # client API
  def put(server, key, value), do: GenServer.cast(server, {:put, key, value})
  def get(server, key), do: GenServer.call(server, {:get, key})

  # OTP GenServer behaviour callbacks
  def init(:ok) do
    {:ok, %{}}
  end

  def handle_cast({:put, key, value}, state) do
    {:noreply, Map.put(state, key, value)}
  end

  def handle_call({:get, key}, _from, state) do
    {:reply, Map.get(state, key), state}
  end
end
```

GenServer usage

```
iex> {:ok, pid} = ExampleServer.start_link()
{:ok, #PID<0.118.0>}

iex> ExampleServer.put(pid, :foo, 1)
:ok

iex> ExampleServer.get(pid, :foo)
1

iex> ExampleServer.get(pid, :foo2)
nil

iex> Process.unlink(pid)
true

iex> Process.exit(pid, :kill)
true

iex> Process.alive?(pid)
false

iex> ExampleServer.get(pid, :foo)
** (exit) exited in: GenServer.call(#PID<0.118.0>, {:get, :foo}, 5000)
    ** (EXIT) no process: the process is not alive ...
```

OTP behaviours

1. GenServer
2. **Supervisor**
3. Application

Supervision

A supervisor is responsible for starting, stopping, and monitoring its child processes. The basic idea of a supervisor is that it is to keep its child processes alive by restarting them when necessary.

- Processes can supervise other processes:
    - If a supervised process dies, the supervisor is sent a message.
    - If this message isn’t handled, the supervisor dies.

Supervision is key to Erlang – and Elixir's – "*let it crash*" philosophy.

Example Supervisor

Define child processes to monitor and restart:

```
import Supervisor.Spec

children = [
  worker(ExampleServer, [], [name: ExampleServer])
]
```

A supervisor's children can also include other supervisors.

Starting the supervisor will start its children:

```
{:ok, pid} = Supervisor.start_link(children, strategy: :one_for_one)
```

Refactor our GenServer to use its module name, not a process identifier.

```
defmodule ExampleServer do
  use GenServer

  def start_link do
    GenServer.start_link(ExampleServer, :ok, [name: ExampleServer])
  end

  # client API
  def put(key, value), do: GenServer.cast(ExampleServer, {:put, key, value})
  def get(key), do: GenServer.call(ExampleServer, {:get, key})

  # OTP GenServer behaviour callbacks
  def init(:ok) do
    {:ok, %{}}
  end

  def handle_cast({:put, key, value}, state) do
    {:noreply, Map.put(state, key, value)}
  end

  def handle_call({:get, key}, _from, state) do
    {:reply, Map.get(state, key), state}
  end
end
```

Supervisor in action

We can observe the Erlang VM, including running processes and their state:

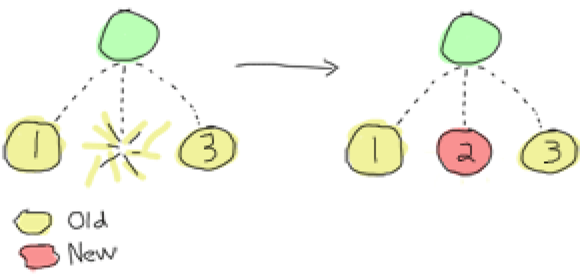
```
iex> :observer.start()
:ok
iex> {:ok, pid} = Supervisor.start_link(children, strategy: :one_for_one)
{:ok, #PID<0.116.0>}
iex> ExampleServer.put(:foo, 1)
:ok
iex> ExampleServer.get(:foo)
1
iex> pid = Process.whereis(ExampleServer)
#PID<0.117.0>
iex> Process.exit(pid, :kill)
true
iex> Process.whereis(ExampleServer)
#PID<0.123.0>
iex> ExampleServer.get(:foo)
nil
```

Supervisor restart strategies

1. One for one.
2. One for all.
3. Rest for one.

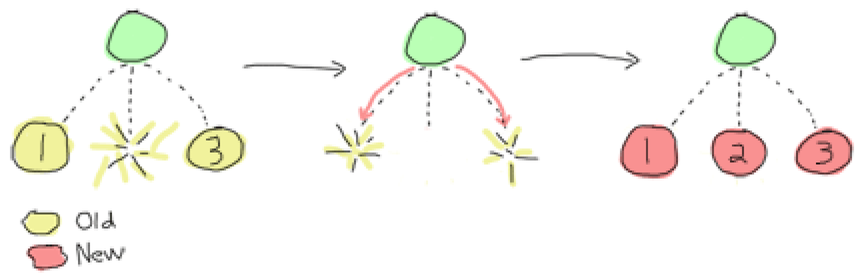
Restart strategies: one for one

When a process dies, restart only the process that failed.



Restart strategies: one for all

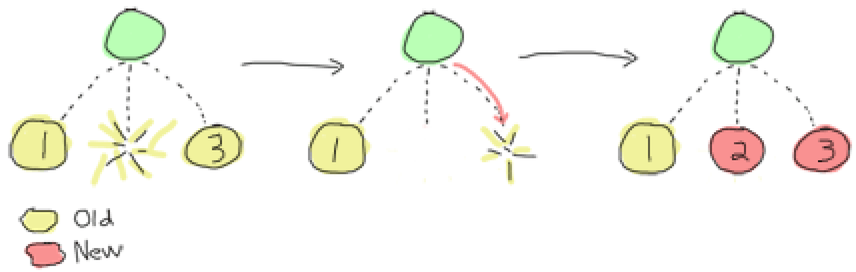
When a process dies, kill any remaining supervised processes and restart them all.



Used whenever processes under a single supervisor heavily depend on each other to be able to work normally.

Restart strategies: rest for one

When a process dies, kill any processes defined after it and restart them.



Used whenever you have to start processes that depend on each other in a chain

"Let it crash"

- Elixir – and Erlang – encourage you to code the *happy path*.
- Runtime errors should be allowed to crash the process.
- Supervisors handle restarting a process:
  - *"Have you tried turning it off, and back on again?"*
- Restarting allows a process to initialise to a known good state.
- Too many process restarts are propagated up the supervision tree.

This leads to a clean separation of issues. We write code that solves problems and code that fixes problems, but the two are not intertwined.

OTP behaviours

1. GenServer
2. Supervisor
3. **Application**

Application

- An independent module or group of modules.
- That implements a specific piece of functionality.
- Can be started and stopped independently.
- Is supervised.
- Can be reused in other OTP Applications.

```
def application do
  [
    mod: {PhoenixExample, []},
    applications: [:phoenix, :phoenix_html, :cowboy, :logger, :gettext, :phoenix_ecto, :postgrex]
  ]
end
```

Elixir's multicore advantage

- Elixir has immutable data structures:
  - No locks, easy to parallelise.
- BEAM (VM) will run on one OS process:
  - One thread – as a scheduler – per CPU core.
  - Elixir processes are distributed amongst available CPU cores.
  - Can redistribute processes to even load, or reduce CPU core usage.
- Processes running across multiple nodes are no different than when running on a single node.
- Once started, Elixir (Erlang/OTP) applications are expected to run forever.

Let's dive into some Elixir usage

- Pattern matching.
- Pipe operator.
- Macros.
- Testing.
- Using `mix` to:
  - Scaffold a new application.
  - Manage third party dependencies.

Pattern matching using the match operator

- In Elixir, the `=` operator is actually called the match operator.
- It can be used to match against simple values, but is more useful for destructuring complex data types.

```
iex> a = 1
1
iex> [a, b, c] = [1, 2, 3]
[1, 2, 3]
iex> a
1
iex> b
2
iex> c
3
```

Case statements

Using pattern matching within a case statement:

```
def buy_ticket?(age) do
  case age do
    age when age >= 18 -> true
    _ -> false
  end
end
```

This example could also be written by pattern matching on function arguments:

```
def buy_ticket?(age) when age >= 18, do: true
def buy_ticket?(_age), do: false
```

Pipe operator

The pipe operator `|>` passes the result of an expression as the first parameter of another expression.

```
foo(bar(baz(new_function("initial value"))))
```

Becomes:

```
"initial value"
|> new_function()
|> baz()
|> bar()
|> foo()
```

Macros and testing

Elixir's built-in unit testing framework, ExUnit, takes advantage of macros to provide great error messages when test assertions fail.

```
defmodule ListTest do
  use ExUnit.Case, async: true

  test "can compare two lists" do
    assert [1, 2, 3] == [1, 3]
  end
end
```

The `async: true` option allows tests to run in parallel, using as many CPU cores as possible.

Unit test execution

Running the failing test produces a descriptive error:

```
$ mix test

1) test can compare two lists (ListTest)
     test/list_test.exs:13
     Assertion with == failed
     code:  [1, 2, 3] == [1, 3]
     left:  [1, 2, 3]
     right: [1, 3]
```

The equality comparison failed due to differing left and right hand side values.

Using `mix` to create an Elixir app

```
$ mix new example --sup --module Example --app example
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/example.ex
* creating lib/example/application.ex
* creating test
* creating test/test_helper.exs
* creating test/example_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd example
    mix test

Run "mix help" for more commands.
```

Let's have a peek inside an Elixir app ...

- Environment specific configuration: `config.exs`.
- Application supervisor defined, by using the `--sup` flag to mix.
- Barebones initial app module: `Example`.
- Testing included by default.
- Application related mix settings: `mix.exs`.

Managing dependencies

- Packages can be sourced from:
  - Erlang's package manager: Hex.
  - Git repository, or GitHub.
  - Local path.
  - An umbrella application.
- Third party dependencies defined in `mix.exs`:

```
defp deps do
  [
    {:ecto, "~> 2.1"},
    {:local_dependency, path: "~/src/local_dependency"},
  ]
end
```

Fetching dependencies

Once the dependencies have been listed in `mix.exs`, you run `mix deps.get`:

```
$ mix deps.get
Running dependency resolution...
Dependency resolution completed:
  decimal 1.3.1
  ecto 2.1.3
  poolboy 1.5.1
* Getting ecto (Hex package)
  Checking package (https://repo.hex.pm/tarballs/ecto-2.1.3.tar)
  Fetched package
```

Then compile them using `mix deps.compile`.

Elixir development

- Install using Homebrew (on Mac).
- Elixir's interactive shell: `iex`.
- Hex package manager.
- Atom text editor:
  - Elixir language support for Atom.
  - Atom Elixir package.

my_module.ex

```
1  defmodule MyModule do
2    @joiner "|"
3
4    def my_func(param1, param2) do
5      alias Enum, as: MyEnum
6
7      list = ["a", "b", "b"]
8
9      list |> MyEnum.join(
10
11     my_priv_func(1)
12   end
13
14   defp my_priv_func(
15     unless some_para
16       IO.puts "OUT"
17     else
18       IO.puts "IN"
19     end
20   end
21 end
22
```

v	list	variable
v	param1	variable
v	param2	variable
@	joiner	attribute
f	my_priv_func/1	private
f	abs(number)	Kernel
m	alias(module, opts)	Kernel.SpecialForms
m	alias!(alias)	Kernel
m	and(left, right)	Kernel
f	apply(fun, args)	Kernel

```
@spec abs(number) :: number
```

Returns an integer or float which is the arithmetical absolute value of `number`.

sup.ex

```elixir
defmodule Sup do
  use Application

  # See http://elixir-lang.org/docs/stable/elixir/Application.
  # for more information on OTP Applications
  def start(_type, _args) do
    import Supervisor.Spec, warn: false

    children = [
      # Define workers and child supervisors to be supervised
      # worker(Sup.Worker, [arg1, arg2, arg3]),
      worker(MyServer, [])
    ]

    # See http://elixir-lang.org/docs/stable/elixir/Supervisor
    # for other strategies and supported options
    opts = [strategy: :one_for_one, name: Sup.Supervisor]
    Supervisor.start_link(children, opts)
  end

end
```

Elixir Docs - Supervisor.Spec

Docs | Types | Callbacks

See Online Docs

> Supervisor.Spec

Convenience functions for defining a supervision specification.

Example

By using the functions in this module one can define a supervisor and start it with `Supervisor.start_link/2`:

```elixir
import Supervisor.Spec

children = [
  worker(MyWorker, [arg1, arg2, arg3]),
  supervisor(MySupervisor, [arg1])
]

Supervisor.start_link(children, strategy: :one_for_one)
```

In many situations, it may be handy to define supervisors backed by a module:

```elixir
defmodule MySupervisor do
  use Supervisor

  def start_link(arg) do
    Supervisor.start_link(__MODULE__, arg)
  end
```

Phoenix Framework

> *A productive web framework that does not compromise speed and maintainability.*

- Written in Elixir.
- Implements the server-side model-view-controller (MVC) pattern.
- *Channels* for soft-realtime features.
- Pre-compiled view templates for blazing speed.
- Uses the Erlang HTTP server, Cowboy.
- Response times measured in microseconds (μs).

Chris McCord
@chris_mccord

Final results from Phoenix channel benchmarks on 40core/128gb box. 2 million clients, limited by ulimit
#elixirlang

Phoenix vs another web framework

Technical requirement	Server A	Server B
HTTP server	Nginx and Phusion Passenger	Erlang
Request processing	Ruby on Rails	Erlang
Long-running requests	Go	Erlang
Server-wide state	Redis	Erlang
Persistable data	Redis and MongoDB	Erlang
Background jobs	Cron, Bash scripts, and Ruby	Erlang
Service crash recovery	Upstart	Erlang

Comparison of technologies used in two real-life web servers.

Command Query Responsibility Segregation and event sourcing

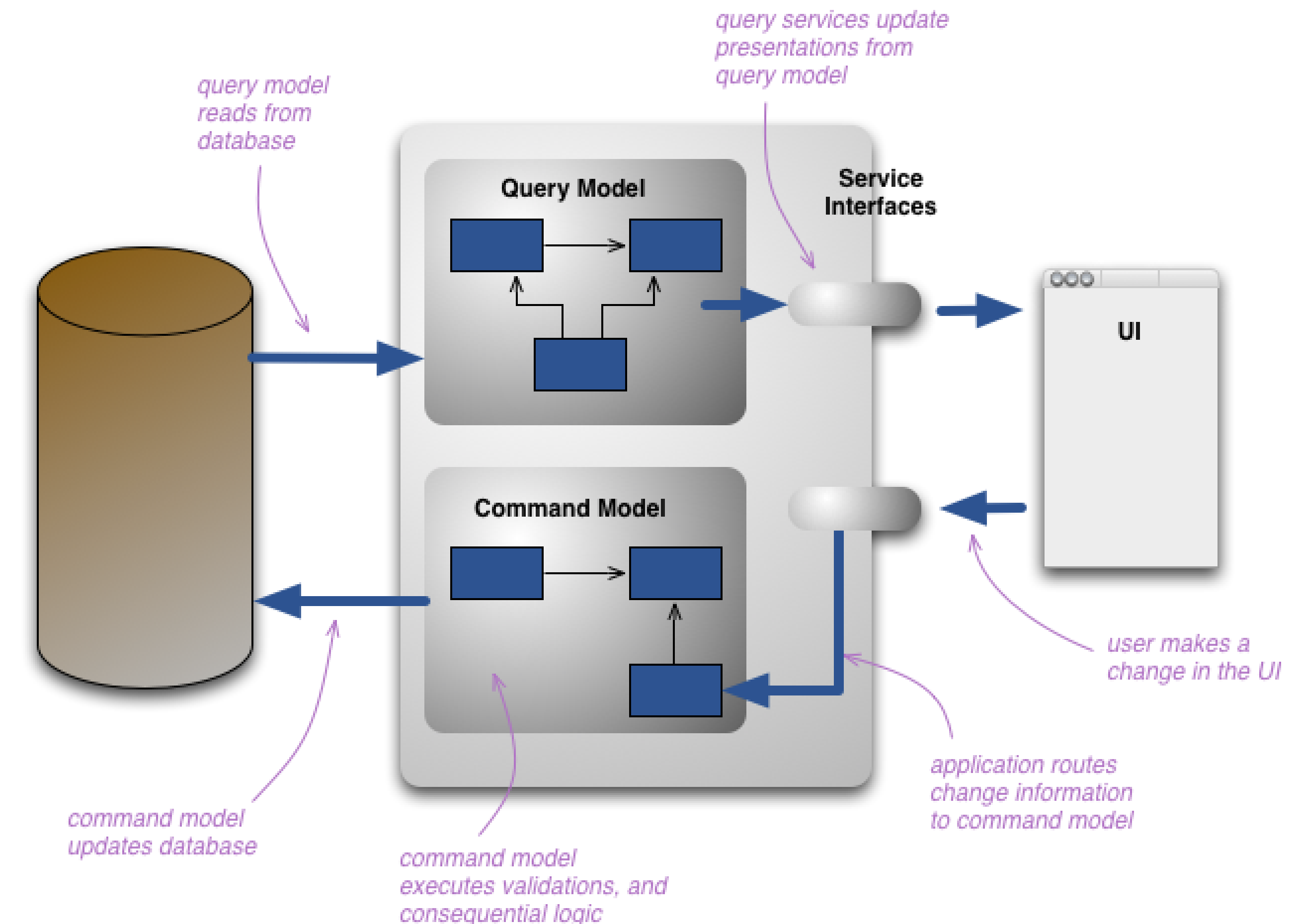
CQRS/ES

At it’s simplest CQRS is the separation of commands from queries.

- Commands are used to mutate state in a write model.
- Queries are used to retrieve a value from a read model.

The read and write models are different logical models.

They may also be separated physically by using a different database or storage mechanism.


query model
reads from
database
query services update
presentations from
query model
Query Model
Service
Interfaces
UI
Command Model
user makes a
change in the UI
application routes
change information
to command model
command model
updates database
command model
executes validations, and
consequential logic

Commands

- Commands are used to instruct an application to do something.
- They are named in the imperative:
  - Register account
  - Transfer funds
  - Mark fraudulent activity

Domain events

- Domain events indicate something of importance has occurred within a domain model.
- They are named in the past tense:
    - Account registered
    - Funds transferred
    - Fraudulent activity detected

Queries

- Domain events from the write model are used to build the read model.
- The read model is optimised for querying, using whatever technology is most appropriate:
  - Relational database
  - In-memory store
  - NoSQL database
  - Full-text search index

Event sourcing

- Application state changes are modelled as a stream of domain events:

BankAccountCreated	DepositPerformed	OwnerChanged	WithdrawalPerformed
id: 123	accountId: 123	accountId: 123	accountId: 123
owner: John Doe	amount: 20€	newOwner: Jane Doe	amount: 10€

- An aggregate's current state is built by replaying its domain events:

```
f(state, event) => state
```

Event streams

- Domain events are persisted in order – as a logical stream – for each aggregate.
- The event stream is the canonical source of truth.
- It is a perfect audit log.
- All other state in the system may be rebuilt from these events:
    - Read models are projections of the event stream.
    - You can rebuild the read model by replaying every event from the beginning of time.

Why choose CQRS/ES?

- Domain events describe your system activity over time using a rich, domain-specific language.
- They are an immutable source of truth for the system.
- Auditing.
- They support temporal queries, and *after the fact* data analysis of events.
- A separate logical read model allows optimised and highly specialised query models to be built.
- Bypass the object-relational (ORM) impedance mismatch.

Benefits of using CQRS

- The processing of commands and queries is asymmetrical, you can:
  - Scale the read and write models independently.
  - Dedicate the appropriate number of servers to each side.
- Events and their schema provide the ideal integration point for other systems.
- They allow migration of read-only data between persistence technologies by replaying and projecting all events.

Costs of using CQRS

- Events also provide a history of your poor design decisions.
- Events are immutable.
- It's an alternative – and less common – approach to building applications than basic CRUD.
- It demands a richer understanding of the domain being modelled.
- CQRS adds risky complexity.
- Eventual consistency.

Recipe for building a CQRS/ES application in Elixir

- A domain model containing our aggregates, commands, and events.
- Hosting of an aggregate root and a way to send it commands.
- An event store to persist the domain events.
- Read model store for querying.
- Event handlers to build and update the read model.
- A web front-end UI to display read model query data, and to dispatch commands to the write model.

High-level application lifecycle

1. Read model queried to display data: UI, API.
2. Command built from a user request: Form POST data, API.
3. Command is validated and authorised.
4. Aggregate is located: its state rebuilt from previous events, or created from scratch.
5. Command passed to aggregate, business logic is validated.
6. Domain events are appended to the aggregate's event stream.
7. Its internal state is updated by applying these events.
8. Event handlers are notified of the events: read-model updated in response.

An aggregate in domain-driven design

- Defines a consistency boundary for transactions and concurrency.
- Aggregates should also be viewed from the perspective of being a "conceptual whole".
- Are used to enforce invariants in a domain model.
- Naturally fit within Elixir's actor concurrency model:
  - `GenServer` enforces serialised access.
  - Communicate by sending messages: commands and events.

An event-sourced aggregate

Must adhere to these rules:

1. Each public function must accept a command and return any resultant domain events, or raise an error.
2. Its internal state may only be modified by applying a domain event to its current state.
3. Its internal state can be rebuilt from an initial empty state by replaying all domain events in the order they were raised.

Let's build an aggregate in Elixir

```
defmodule ExampleAggregate do
  # aggregate's state
  defstruct [
    uuid: nil,
    name: nil,
  ]

  # public command API
  def create(%ExampleAggregate{}, uuid, name) do
    %CreatedEvent{
      uuid: uuid,
      name: name,
    }
  end

  # state mutator
  def apply(%ExampleAggregate{} = aggregate, %CreatedEvent{uuid: uuid, name: name}) do
    %ExampleAggregate{aggregate |
      uuid: uuid,
      name: name,
    }
  end
end
```

A bank account example

This example provides three public API functions:

1. To open an account: `open_account/2`.
2. To deposit money: `deposit/2`.
3. To withdraw money: `withdraw/2`.

A guard clause is used to prevent the account from being opened with an invalid initial balance.
This protects the aggregate from violating the business rule that an account must be opened with a positive balance.

Using the aggregate root

Initial empty account state:

```
account = %BankAccount{}
```

Opening the account returns an account opened event:

```
account_opened = BankAccount.open_account(account, %BankAccount.Commands.OpenAccount{
  account_number: "ACC123",
  initial_balance: 100
})
```

Mutate the bank account state by applying the opened event:

```
account = BankAccount.apply(account, account_opened)
```

Aggregates in Elixir

I've shown examples of aggregates implemented using *pure* functions.

The function always evaluates the same result value given the same argument value.

- A *pure* function is highly testable.
- You will focus on behaviour rather than state.
- Decouple your domain from the framework's domain.
- Build your application separately first, and layer the external interface on top.

Unit testing an aggregate

```
defmodule BankAccountTest do
  use ExUnit.Case, async: true

  alias BankAccount.Commands.OpenAccount
  alias BankAccount.Events.BankAccountOpened

  describe "opening an account with a valid initial balance"
    test "should be opened" do
      account = %BankAccount{}
      open_account = %OpenAccount{
        account_number: "ACC123",
        initial_balance: 100,
      }

      account_opened = BankAccount.open_account(account, open_account)

      assert account_opened == %BankAccountOpened{
        account_number: "ACC123",
        initial_balance: 100,
      }
    end
  end
end
```

Unit testing a CQRS/ES application

- Create factory functions for commands and events.
- Use these to direct and verify aggregate behaviour.
- Events can also be used to test: event handlers; read-model projections; inter-aggregate communication using process managers.

ExMachina example:

```
defmodule Factory do
  use ExMachina

  def open_account_factory do
    %BankAccount.Commands.OpenAccount{
      account_number: "ACC123",
      initial_balance: 100,
    }
  end
end
```

Hosting an aggregate in a GenServer

- One Elixir `GenServer` process per aggregate instance:
  - Serialises access to an aggregate instance.
  - Built-in support for terminating idle processes.
- Registry used to track active aggregates:
  - Maps an aggregate identity to a process id (PID).
- Handles side effects from *pure* aggregate functions.
- Commands are routed to an instance.

Executing a command

1. Rebuild an aggregate's state from its events.
2. Execute the aggregate function, providing the state and command.
3. Update the aggregate state by applying the returned event(s).
4. Append the events to storage.
5. An error will terminate the process:
    - Caller will receive an `{:error, reason}` tagged tuple.
    - Aggregate state rebuilt from events in storage on next command.

Command dispatch

- Synchronous command dispatch:
  - Receiving an `:ok` response indicates the command succeeded.
- Middleware can be used to validate, audit commands.
- Commands routed to a process hosting an aggregate instance:

```
defmodule BankRouter do
  use Commanded.Commands.Router

  dispatch OpenAccount,
    to: OpenAccountHandler,
    aggregate: BankAccount,
    identity: :account_number
end

:ok = BankRouter.dispatch(%OpenAccount{account_number: "ACC123", initial_balance: 1_000})
```

Process managers

- A process manager is responsible for coordinating one or more aggregates.
- It handles events and may dispatch commands in response.
- Each process manager has state used to track which aggregate roots are being orchestrated.
- They are vital for inter-aggregate communication, coordination, and long-running business processes.
- Typically, you would use a process manager to route messages between aggregates within a bounded context.

Before I forget ...

It is worth remembering that domain events are the **contracts** of our domain model.

They are recorded within the *immutable* event stream of the aggregate.

A recorded domain event cannot be changed; history cannot be altered.

I'll show you how to migrate, modify, and retire domain events — in effect rewriting history — later.

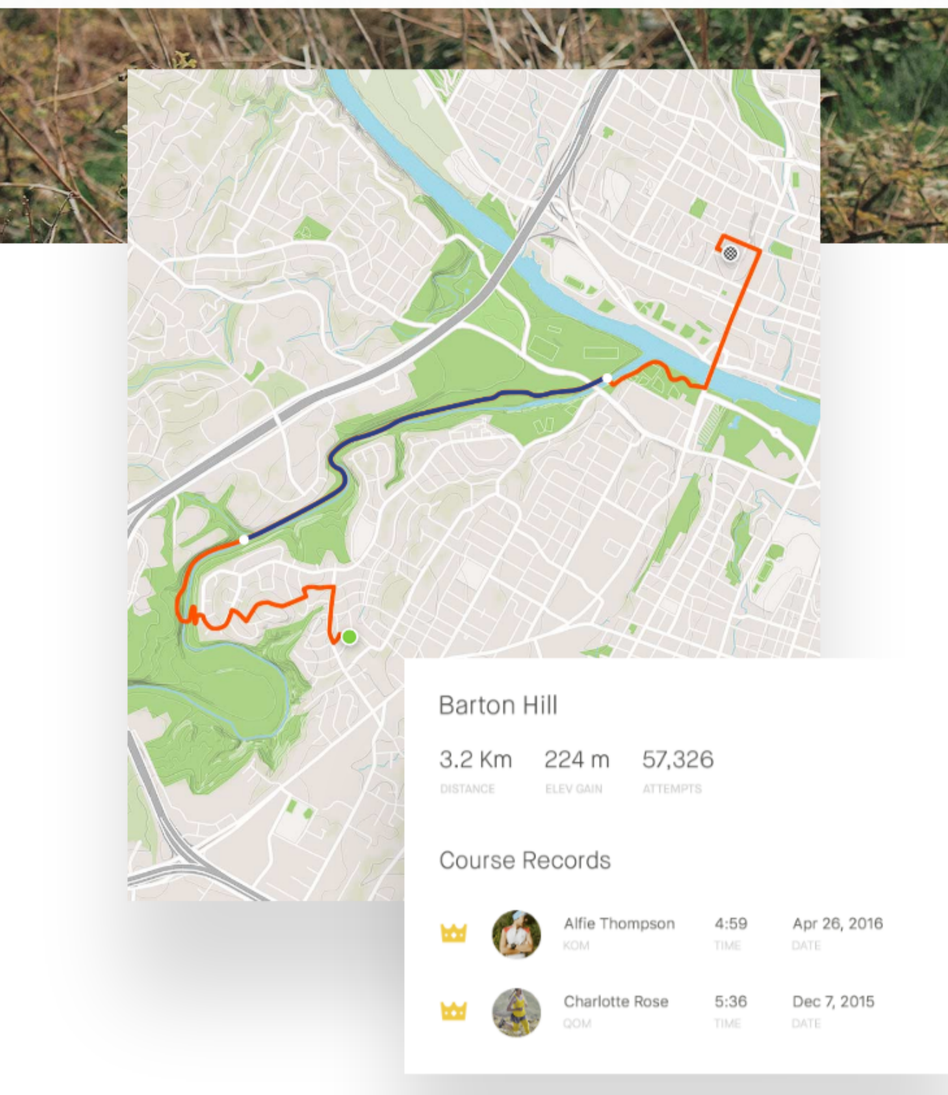
An Elixir CQRS/ES case study

Let's explore a real-world example of implementing these concepts in a Phoenix-based web app.

Strava

- Strava is a very popular social network for athletes (cyclists and runners).
- Who record their rides and runs, and upload them to Strava.
- Strava users create segments from sections of their routes.
- Athletes can compare themselves against other Strava users who cycle or run along the same routes.

STRAVA



Segments changed the game.

There are two eras of endurance sport history: before segments, and after segments. Created by the millions of Strava athletes, segments mark popular stretches of road or trail (like your favorite local climb) and create a leaderboard of times set by every Strava athlete who has been there before.

Segment Challenge

- Segment Challenge allows an athlete to create a competition for a cycling club and its members.
- A different Strava segment is selected each month to compete on.
- Club members' attempts at each segment are fetched from Strava's API.
- Their efforts are ranked by time on a leaderboard.
- Replaces manual tracking of each athlete's segment efforts in a spreadsheet.
- The site is entirely self-service: any Strava member can host a challenge for their own cycling club.

SEGMENT
CHALLENGE
Compete against your friends, teammates and fellow club riders — ride a different segment each month
Challenges
Host your own
Compete

Journey to CQRS/ES in Elixir

- Segment Challenge began life as a *vanilla* Phoenix web application.
- Fell into trap of database schema == domain model.
- I wanted to add an activity feed …
- Seems like a good fit for an event sourced system.
- So I stopped building the site and:
    - Built an event store in Elixir using PostgreSQL: eventstore
    - Wrote a CQRS/ES Elixir library: commanded
    - Rewrote the entire Segment Challenge application.
- 9 months later, added activity feed using a read-model projection.

Building an event store

- Uses PostgreSQL for persistence.
- Only requires four tables:
    1. `events`
    2. `snapshots`
    3. `streams`
    4. `subscriptions`
- Events are serialized to JSON, but stored as binary data.
- Subscriptions use a hybrid push/pull notification model.
- In-memory pub/sub; read from storage on catch-up or new subscriber.

Event store API

```elixir
defmodule EventStore do
  @doc """
  Append one or more events to a stream atomically.
  """
  def append_to_stream(stream_uuid, expected_version, events)

  @doc """
  Reads the requested number of events from the given stream,
  in the order in which they were originally written.
  """
  def read_stream_forward(stream_uuid, start_version \\ 0, count \\ 1_000)

  @doc """
  Subscriber will be notified of each batch of events persisted to a single stream.
  """
  def subscribe_to_stream(stream_uuid, subscription_name, subscriber, start_from \\ :origin)

  @doc """
  Subscriber will be notified of every event persisted to any stream.
  """
  def subscribe_to_all_streams(subscription_name, subscriber, start_from \\ :origin)
end
```

Event store single writer

- A single Elixir process is used to append events to the database.
- Assigns incrementing identifier to each persisted event.
- Guarantee consistent ordering of events within the event store.

Architecting an Elixir application

- An **OTP application** is one or more modules that implement a specific piece of functionality.
- Analogous to a *microservice*.
- Elixir provides support for internal dependencies specific to a project by creating an umbrella project.
- Umbrella projects allow you to create one project that hosts many OTP applications while keeping all of them in a single source code repository.

- segment_challenge
  - _build
  - .deliver
  - apps
    - authorisation
    - challenges
    - commands
    - events
    - infrastructure
    - migrations
    - projections
    - web
  - config
    - config.exs
  - deployment
  - deps
  - rel
    - config.exs
  - support
  - .gitignore
  - mix.exs
  - mix.lock
  - README.md
  - RELEASES.md

Elixir umbrella application

- `authorisation` - Policies to authorise command dispatch.
- `challenges` - Core domain model, command router, process managers, read model projections, queries, and periodic tasks.
- `commands` - Modules for each command.
- `events` - Modules for each domain event.
- `infrastructure` - Serialization and command middleware.
- `projections` - Ecto repository and database migrations to build the read model database schema.
- `web` - Phoenix web front-end.

Let's take a look at the implementation

- Aggregate root: `Challenge`
- Commands and events.
- Unit and integration testing.
- Routing commands: `Router`
- Event handling: `StageEventHandler`.
- Process manager: `ChallengeCompetitorProcessManager`
- Read model:
  - Projections.
  - Querying.

Challenge aggregate root

- Public command functions:
  - Accept challenge state and a command.
  - Return zero, one, or many domain events in response.
- Aggregate protects itself against commands that would cause an invariant to be broken.
- Pattern matching is used to validate the state of the aggregate.
- Every domain event returned by the aggregate has a corresponding `apply/2` function to mutate its state.

Commands & events

- Defining a command with validation: `CreateChallenge`
- An Event: `ChallengeCreated`
    - Decoding an event struct from JSON using a protocol implementation.
    - Used when custom deserialisation is required.

Unit & integration testing

- Unit testing an aggregate: `ChallengeTest`
- Integration testing a use case: `HostChallengeTest`

Tag individual tests to allow specific test runs:

```
mix test --only unit
mix test --only integration
mix test --only wip
```

Use mix test.watch to run tests each time you save a file:

```
mix test.watch --only " wip"
```

Stage event handler

- Used as a *background* worker.
- Fetches data from the Strava API.
- Dispatches command to an aggregate with retrieved data.
- Supervised to handle restarting on failure.

Challenge competitor process manager

- Used to track atheletes who are competiting in a challenge.
- Records which challenges are being hosted by a club.
- Adds, or removes, members when they join or leave a club.
- Events are routed to an instance of the process manager using the `interested?/1` function.

Read model

- Projection: `ChallengeProjection`
  - Uses Ecto, a database wrapper and query language for Elixir.
  - Projections are specialised event handlers.
- Supervision: `Projections.Supervisor`
- Query: `ChallengesByStatusQuery`

Phoenix web framework integration

- Commands:
  - Construction: `CreateChallengeBuilder`.
  - Dispatch: `CommandController`.
  - Validation middleware: `Validation.Middleware`.
  - Authorisation: `Authorisation`.
- Querying the read model:
  - Plug request pipeline: `LoadChallengeBySlug`.
  - Render query: `ChallengeController`.

Deployment

- Build a release using distillery.
- Deploy using edeliver.

```
mix edeliver build release --skip-mix-clean
mix edeliver deploy release to production
ssh edeliver@segmentchallenge.com 'sudo /bin/systemctl restart segment_challenge'
```

- Erlang releases *must be built* on a remote host that is a similar architecture to the production machines.
- Release contains the full Erlang runtime system, all dependencies, the Elixir runtime, and your Elixir application in a standalone embedded node.

Production monitoring

- Command auditing middleware.
  - Open source command dispatching middleware.
  - Records every dispatched command to the configured database storage.
  - Includes whether the command was successfully handled, or any error.
- Monitor events & subscriptions in the event store.
- Use Prometheus to record and Grafana to display key metrics: Erlang, Phoenix controller actions, Ecto queries.

RCRA Instrumentation via Prometheus
Zoom Out
Sep 29, 2016 07:52:59 to Sep 29, 2016 08:22:29
PHOENIX PIPELINE
HTTP request time, 30s average
Controller call durations, 30s average
HTTP request count, 10s average
Erlang VM memory
Ecto Query Duration, 30s average
CPU Time, 10s average
BEAM Resident RAM use
275 MiB
RAM
ok
utime
stime

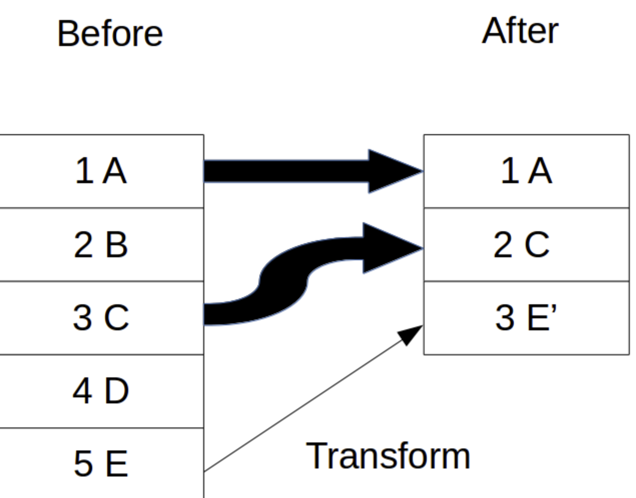
Domain event migration strategy

1. **Multiple versions**: `RegisterAccountV1`, `RegisterAccountV2`
2. **Upcasting**: single event version, upgrade old events on read.
3. **Lazy transformation**: upcast the events on read, and persist the modified event.
4. **In place transformation**: background job to edit events in the database.
5. **Copy & transfrom**: migrate an event store into a new database, altering events as required.

Greg Young has published: [Versioning in an Event Sourced System.]()

Copy & transform

> *Copy and transformation transforms every event to a new store. In this technique the old event store stays intact, and a new store is created instead.*



Event store migration

- Noticed large number of similar events in production, wanted to combine them into single event.
- Created an EventStore migrator:
  - Copies an event store PostgreSQL database from a source to a target.
  - You can transform, remove, aggregate, and alter the serialization format of the events.

Remove an event

Uses Elixir's standard `Stream` module to exclude a particular event:

```
EventStore.Migrator.migrate(fn stream ->
  Stream.reject(
    stream,
    fn (event_data) -> event_data.event_type == "UnwantedEvent" end
  )
end)
```

Upgrade an event

Using pattern matching to migrate a specific type of event:

```
defmodule OriginalEvent, do: defstruct [uuid: nil]
defmodule UpgradedEvent, do: defstruct [uuid: nil, additional: nil]

EventStore.Migrator.migrate(fn stream ->
  Stream.map(
    stream,
    fn (event) ->
      case event.data do
        %OriginalEvent{uuid: uuid} ->
          %EventStore.RecordedEvent{event |
            event_type: "UpgradedEvent",
            data: %UpgradedEvent{uuid: uuid, additional: "upgraded #{uuid}"},
          }
        _ -> event
      end
    end
  )
end)
```

Aggregate events

```
defmodule SingleEvent, do: defstruct [uuid: nil, group: nil]
defmodule AggregatedEvent, do: defstruct [uuids: [], group: nil]

# aggregate multiple single events for the same group into one aggregated event
defp aggregate([%{data: %SingleEvent{}}] = events), do: events
defp aggregate([%{data: %SingleEvent{group: group}} = source | _] = events) do
  [
    %EventStore.RecordedEvent{source |
      data: %AggregatedEvent{
        uuids: Enum.map(events, fn event -> event.data.uuid end),
        group: group,
      },
      event_type: "AggregatedEvent",
    },
  ]
end
defp aggregate(events), do: events

EventStore.Migrator.migrate(fn stream ->
  stream
  |> Stream.chunk_by(fn event -> {event.stream_id, event.event_type} end)
  |> Stream.map(fn events -> aggregate(events) end)
  |> Stream.flat_map(fn events -> events end)
end)
```

Live queries

- Use Phoenix channels — WebSockets, or long polling — to create a bidirectional client-server connection.
- Client subscribes to interested queries: a read model projection.
- Publish a notification when the read model projection is updated in response to an event.
- Subscribed clients are notified, and can refresh their query.

Lessons learnt

- Events are the *immutable contracts* of your application:
  - Hard – but not impossible – to change once deployed to production.
  - Keep event payloads succinct.
- Read model can be rebuilt easily:
  - Assuming the underlying events contain the data you need.
- Requires building a task-based UI.
- Inter-aggregate communication – using process managers or event handlers – adds complexity.

Questions?

Thank you

- Ben Smith

  ben@10consulting.com
- https://10consulting.com/
  - Browse these slides & read more about Elixir and CQRS/ES.
  - Subscribe to my CQRS/ES and Elixir mailing list.
- GitHub open source projects:
  - eventstore
  - commanded
- Start learning Elixir (startlearningelixir.com).