A Transmuting Journey:
From a Ruby on Rails Monolith
to Elixir and Elm Microservices

Volker Rabe - Technical Lead Architect - MyMeds&Me
The Mission
Scale the business without linear scaling the team
What's wrong with Monoliths?
Adding more and more functionality to the system leads to entropy
A new level of quantity leads to a new level of quality
Entropy leads to complexity in the code
It gets harder and harder to add new functionality and maintain the system.
The Case

Growing platform with diverging functionality per product, version and customer
New Quantity

Instead of 1 product, 2 customers and 2 versions
up to 4 products, 20 customers and 20 versions
New Quality

Up to 80 different versions of the same code base hosted on up to 320+x hosts
Result

Platform complexity grows exponential per product, version and customer
Consequence

Decreasing Productivity, Extensibility and Maintainability
Outcome

Imploding Time to Market and Stability plus Exploding Costs
How should we deal with it?
Turtle tactic
...or Raven tactic
What are the challenges we have to face in order to scale the business?
Configuring Tailored Products for Customers in SaaS

- Specific questionnaires and reports per customer
- Map questionnaires to report structure
- Increase customers without linear growing the team
Problem: Hard-coded Configuration of Questionnaires

- Manual effort for testing changes is huge
- Setup new questionnaires takes long
- Maintaining the existing ones doesn't scale
Solution: Configure Tailored Products in a Graph

- Changes can be tested in automation
- Setup new questionnaire takes hours not days
- Existing ones can be maintained via graph analytics
Storing Sparse Data of Customer Tailored Products

Data structure can vary from product to product
Data structure can vary from customer to customer
Map form data to specific report structure

{  
  "firstName": "John",  
  "lastName": "Smith",  
  "isAlive": true,  
  "age": 25,  
  "height_cm": 167.64,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": "10021-3100"  
  },  
  "phoneNumbers": [  
    { "type": "home", "number": "212 555-1234" },  
    { "type": "fax", "number": "646 555-4567" }  
  ]
}
Problem: Ad hoc Format of Sparse Data

> Changes on the data structure are not transparent
> Diverging data structures are hard to provide
> Querying data for analysis is non trivial
Solution: Store Sparse Data as Connected Documents in Graph

- Changes on the data structure are controlled and transparent
- Complex queries can be done with ease in a graph
- Realtime analytics can run on mass data
Support Multiple Frontends

- Backend APIs for web and mobile frontends
- Same scope of functionality for all frontends
- Switch web technology without backend changes
Problem: Tight Coupling of Frontend and Backend

- Changes on both ends are fragile
- Extending functionality is pretty costly
- Providing multiple frontends is very hard
Solution: Decouple Frontend and Backend via WebSockets

- Frontend/Backend can be developed and tested separately
- Extending functionality of API and UI is straightforward
- UI can be generated for multiple frontends
Sharing Modules for Multiple Products

- Products have the same modules in common
- Modules have various dependencies with each other
- Modules evolve with different speed per product
Problem: Tight Coupling of Modules

- Diverging code between products and versions
- Refactorings have side effects on other modules
- Need to deploy, test and ship the entire product
Solution: Decompose Application as Decoupled Microservices

> Microservices can be developed and replaced individually
> Microservices can be reused for products and legacy systems
> Microservices can be tested, deployed and scaled individually
Agile Operational Approach

- Increase agility with DevOp culture
- Minimise the operational efforts
- Traceability over state and changes
Problem: Entropy of Mutable Infrastructure

- Accumulated changes feed system entropy
- Changes are hard, error prone and a time sink
- Operations become bottleneck
Solution: Containerise Application and Infrastructure with Docker

- System provides traceability of processes, state and changes
- Devs can manage system setup via containers
- Operations can become a shared responsibility
Support for Multi-Customer, -Product, -Version, -Environment

- Single-tenant hosting for customers
- One or more products per customer
- Support 2 major versions per customer
Problem: Managing Complexity

- Up to 20 clients with up to 4 products
- result in up to 80 versions in 4+x environments
- spread over up 320+x hosts
Solution: Orchestrate the Containerised System with Kubernetes

- Versions can be handled transparently
- Clusters can be controlled centrally for all versions
- Releases and patches can be promoted in the clusters
Conclusion
The solutions sound promising by themselves, but how do they work together?
The Vision
Perceptrix

- Data Transformation
- Propagation
- Parallelisation
- Realtime
Perceptron

![Perceptron Diagram](image_url)
Perceptrix
The Architecture
The Approach
Approach

- Why Event-Driven Architecture?
- Why Elixir Backend Services?
- Why Phoenix WebSocket Broker?
- Why RabbitMQ Message Broker?
- Why Elm Frontend Services?
Why Event-Driven Architecture?
Why Event-Driven Architecture?

- Support synchronous, blocking RCP where necessary
- Use asynchronous, non-blocking PubSub where possible
- Listen to messages for multiple concerns
- React on events in multiple microservices in parallel
- Loose coupling between microservices
- Facilitate changes and integration of new microservices

> Supports nicely reactive realtime application
> Easy integration of legacy and third party systems
> Fits distributed architecture and Internet of Things
Why Elixir Backend Services?
defmodule Math do
  def square(x) do
    x * x
  end
end

Enum.map [1, 2, 3], &Math.square/1
Immutable State

```elixir
iex> tuple = { {:ok, "hello"}, {:ok, "hello"} }
iex> put_elem(tuple, 1, "world")
{:ok, "world"}
iex> tuple
{:ok, "hello"}
```
Pattern Matching and Recursion

defmodule Calculator do
  def sum(list) when is_list(list) do
    add(list)
  end

  def sum(_) do
    nil
  end

  defp add([head | tail]) do
    head + add(tail)
  end

  defp add([]) do
    0
  end
end
Protocols and Meta Programming

defprotocol Blank do
  def blank?(data)
  end

defimpl Blank, for: Integer do
  def blank?(_, do: false)
end

defimpl Blank, for: List do
  def blank?([]), do: true
  def blank?(_), do: false
end

defmodule TestCase do
  @doc false
  defmacro __using__(_opts) do
    quote do
      import TestCase
    end
  end

  defmacro test(description, do: block) do
    function_name = String.to_atom("test " <> description)
    quote do
      def unquote(function_name)(), do: unquote(block)
    end
  end
end
Concurrent Programming

[Diagram showing the Actor model with Elixir]
Fault-tolerant
Why Elixir Backend Services?

- Functional Programming Language
- Immutable State
- Pattern Matching and Recursion
- Protocols and Meta Programming
- Concurrent Programming
- Fault-tolerant

> High productivity and low maintenance code
> Efficient Data Transformation made simple
> Efficient, Scalable, Fault-tolerant Applications made simple
Why Phoenix WebSocket Broker?
Why Phoenix WebSocket Broker?

- Beauty of Elixir
- Lightweight and Modular
- Power of Plug in Controllers and Routers
- Pattern Matching and Authorisation in Channels
- Option to use Phoenix.PubSub with Redis
- Scalable and Efficient WebSocket Communication

> Highly available, dump and secure Broker
> Central instance for crosscutting non-functional concerns
> Scalable, performant and efficient alternative to API Gateways
Why RabbitMQ Message Broker?
Why RabbitMQ Message Broker?

- Build in Erlang with OTP
- Real Queues: First in, First out
- Transient and Persistent Message Queues
- Many Patterns: RPC, PubSub, Routing, Topics, Worker Queues
- Transactional Handling and Replays Messages
- Management Interface

> Highly available, resilient and reliable Broker
> Enables Inter-process calls and data synchronisation
> Technology independent, smart backend services
Why Elm Frontend Services?

```haskell
import Html exposing (text)

-- Zip two lists together. In this case, we are pairing up
-- names and ages.
main =
  text (toString (zip ["Tom", "Sue", "Bob"] [45, 31, 26]))

zip : List a -> List b -> List (a,b)
zip xs ys =
  case (xs, ys) of
  ( x :: xs', y :: ys' ) ->
  (x,y) :: zip xs' ys'
  (_, _) ->
  []
```
Why Elm Frontend Services?

- Functional Programming Language
- Static Typed and Compiler Safe
- Stateless, Immutable Data
- Pattern Matching, Recursion, Map, Merge, FoldP, Filter
- Elm Architecture
- Time Traveling Debugger
- Semantic Versioning Package Manager

> High productivity and low maintenance code
> Efficient data transformation made simple
> Event-driven architecture made simple
The Learnings
Infrastructure

- Microservices imply more complexity in infrastructure
- Complexity can be managed by isolation and automation
- Docker and Kubernetes fit isolation and automation
- Immutable infrastructure makes the difference
- New infrastructure requires a lot of groundwork
- Good logging and monitoring is key
- Ecosystem is quite young and need some tweaks
- OPs need to adapt new approach to productionise it
- An agile approach can minimise the risks
OrientDB is very powerful and leverages new possibilities.
OrientDB is still under development.
Microservices lead to good separation and autonomy.
Container automatisation leads to streamlined setups.
Message-driven enables choreography of microservices.
Some ground work upfront necessary using Elixir.
Functional language fits in event-driven approach.
OTP makes the difference in distributed application.
Elixir was picked up with joy by every back-/frontend Dev.
Frontend

- Splitting off backend/frontend leads to exchangeable clients
- Fetching dependencies with NPM is slow and not reliable
- Elm compiler, package manager, debugger make it productive
- JS libraries can still be used for interoperation
- Using JS libraries won't leverage the type safety of Elm
- Elm is still young and has breaking changes
- Functional language fits in event-driven approach
- Elm architecture makes the difference for event-driven design
- Elm was picked up with joy by every front-/backend Dev