Introduction to Microservices

Michael Hackstein

ArangoDB GmBH
@mchacki
michael@arangodb.com

Michael Hackstein

- ArangoDB Core Team
  - Web Frontend
  - Graph visualisation
  - Graph features
- Host of cologne.js

Introduction

Monolith

One large Application
Designed to run on a single machine
Loose coupling of objects due to object orientation

Pros & Cons

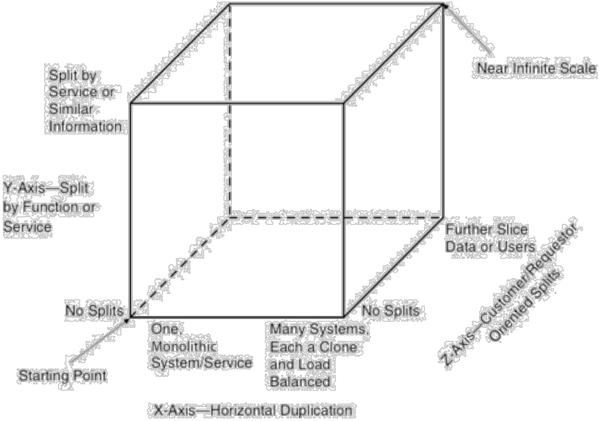
Pros

- Proven architecture
- Lot's of experience
- No network delay

Cons

- Grows large over time
    - Hard to maintain
    - refactoring expensive
- Typically written in one language
- Hard to scale-out
    - Probably contains local state
    - Can only be scaled as a whole
- "Hot-spots" require multi-threading
- High basic hardware requirements
    - Parts of the app are CPU intensive
    - Parts of the app are RAM intensive
    - Both have to be large

The scale cube


Near Infinite Scale
Split by Service or Similar Information
Y-Axis—Split by Function or Service
Further Slice Data or Users
Z-Axis—Customer/Requestor Oriented Splits
No Splits
No Splits
Starting Point
One, Monolithic System/Service
Many Systems, Each a Clone and Load Balanced
X-Axis—Horizontal Duplication

Taken from The art of Scalability

Microservices

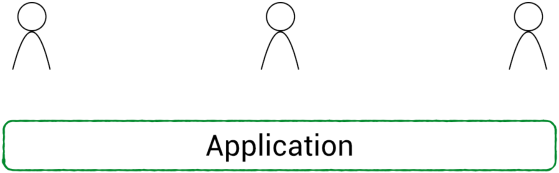
Microservice requirements (Fowler & Lewis)

Few lines of Code
Automated Deployment process
Independent Scalable
Design for failure
Different Languages
Different Databases
Asynchronous Calls
Self-handled Persistence
Source: http://martinfowler.com/articles/microservices.html

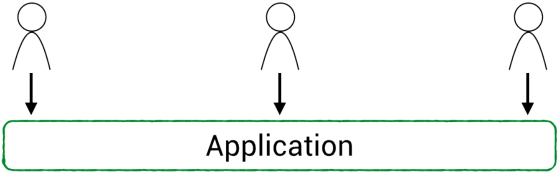
Microservices philosophy

Designed to run in a cluster of servers
Define many services, each for one purpose only
Each microservice offers a documented public API
A microservice can make use of other microservices
Each service should not have a large code base

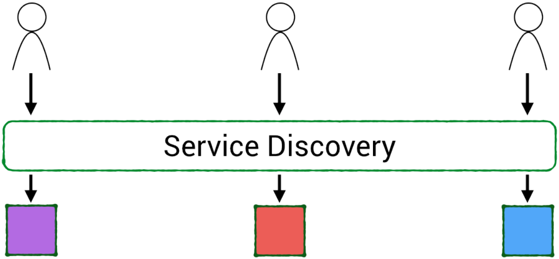
Architecture



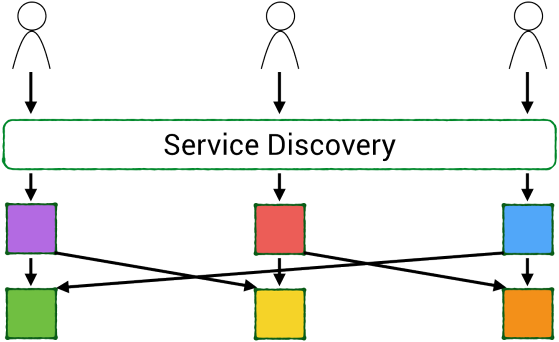
Architecture



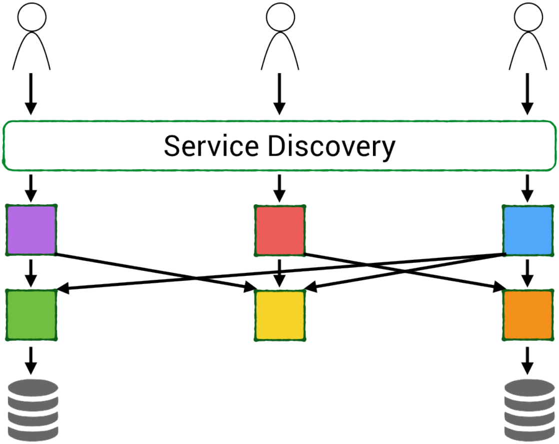
Architecture



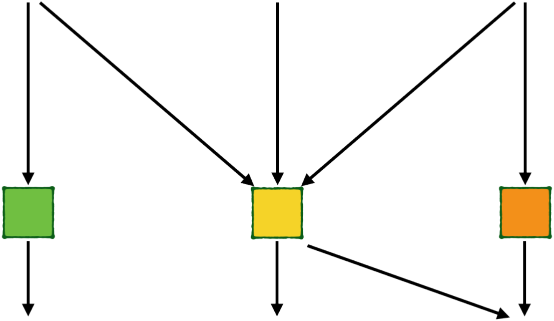
Architecture



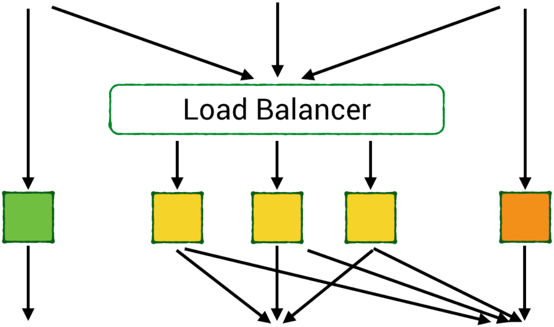
Architecture



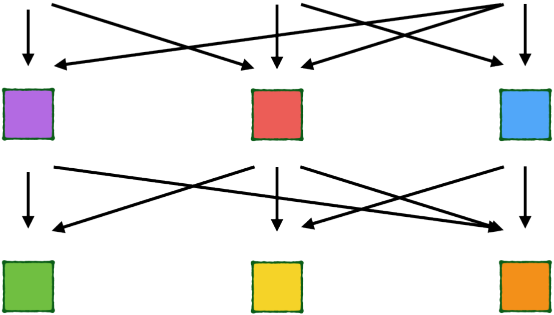
Scaling



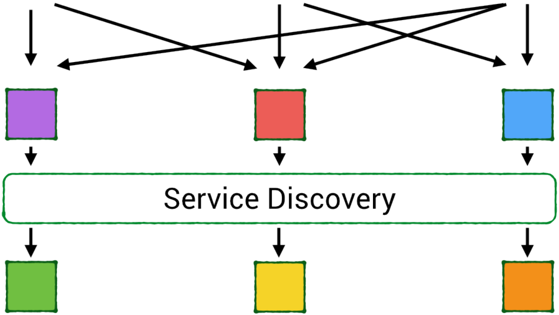
Scaling



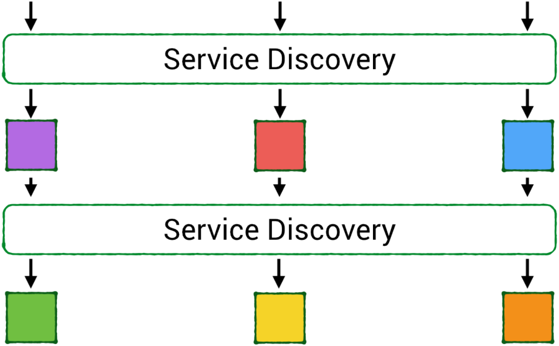
Communication Layers



Communication Layers



Communication Layers



General Advises

- You pay with network traffic
  - Keep the tree as flat as possible
- Request async. and in parallel wherever possible
- Try to build a tree
  - Avoid circles
  - Avoid communication within the same layer

Pros & Cons

Pros

Easy to scale
Loose coupling
Single Threaded
Language Independed
Seamless updates

Cons

Communication Overhead
Load balancing
Service Discovery
Different Architecture
Assume Failures
- Avoid single-point of failure by all means

Monolith ⇒ Microservices

Where to cut?

Cut by logical unit's
Do not rigorously cut by datamodels
Try to maintain a tree-like structure
Do not overdo it

Hands-on

Which logical components do we have?
Where do these components overlap?
Let's define some APIs.

Foundation

Requirements

Network communication
Automatic Failover strategies

Incremental Update strategies

Centralized logging

Communication Tools

Load Balancer

- HAProxy http://www.haproxy.org
- NGINX http://www.nginx.org

Queues

- RabbitMQ http://www.rabbitmq.com
- ActiveMQ http://activemq.apache.org

Communication Tools

```
global
  daemon
  maxconn 4096
  pidfile haproxy.pid

backend microserviceName
  balance roundrobin
  server server1Name 127.127.127.127:8000
  server server2Name 42.42.42.42:8086

frontend http-farm
  bind *:9000
  use_backend microserviceName if { path_beg /prefix }
```

Service Discovery

Mesosphere http://mesosphere.com
kubernetes http://kubernetes.io
Consul http://consul.io

Cluster Management Tools

Apache Mesos http://mesos.apache.org
Giant Swarm http://giantswarm.io
ClusterHQ http://clusterhq.com

Logging Tools

Logstash http://www.elastic.co/products/logstash
Fluentd http://www.fluentd.org
Splunk http://www.splunk.com

Hands-on

Connect to Virtual Machine

SSH:

```
ssh -p 56789 training@127.0.0.1
```

Password: training

Virtual Box Login

Username: training
Password: training

Example Project

In the secure message board it should be possible to register new users.

> This registration should generate a token.

A registered user should be able to login with a password.
A logged-in user should be able to store text messages ([a-z] and ' ').

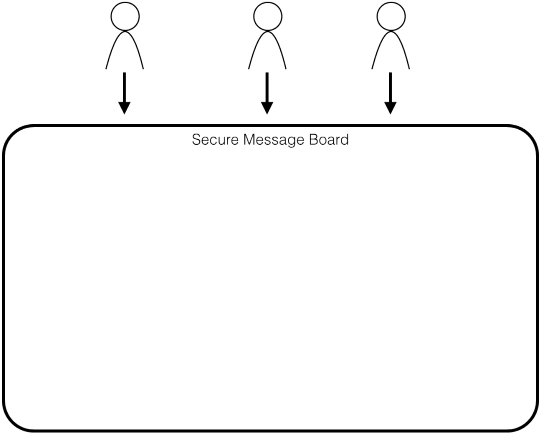
> The message should be encrypted with the users token.

A logged-in user should be able to receive her token.
Every user should be able to use insert another users token.

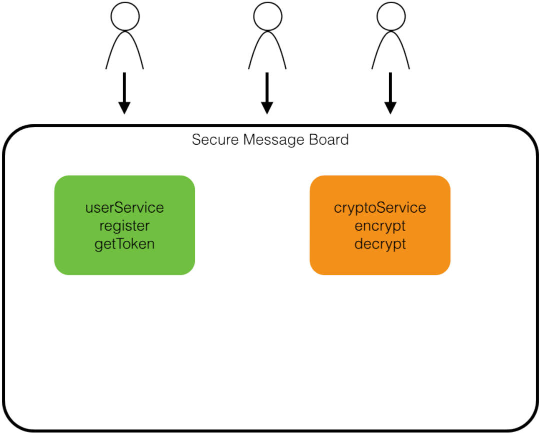
> Only if a user has another users token, she should be able to read any message stored by the other user.

For simplicity we omit sessions and passwords during this training

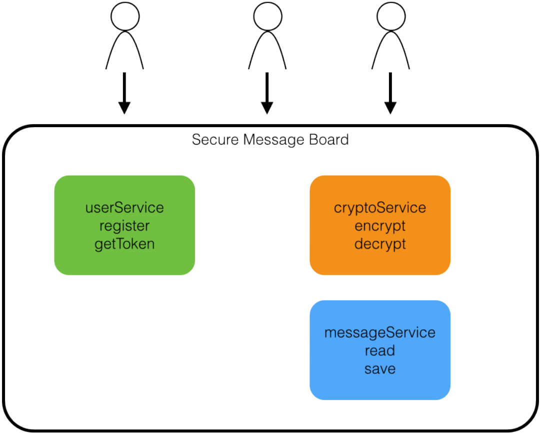
Where to cut?



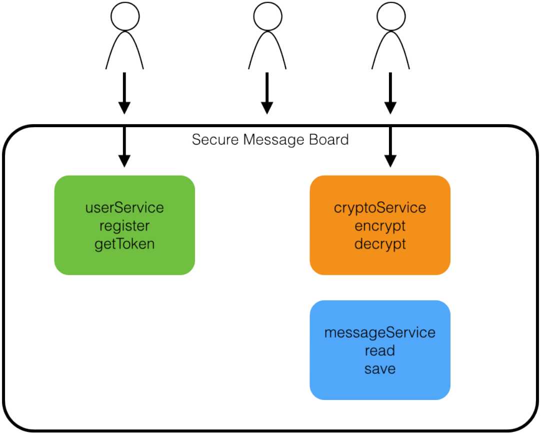
Where to cut?


Secure Message Board
userService
register
getToken
cryptoService
encrypt
decrypt

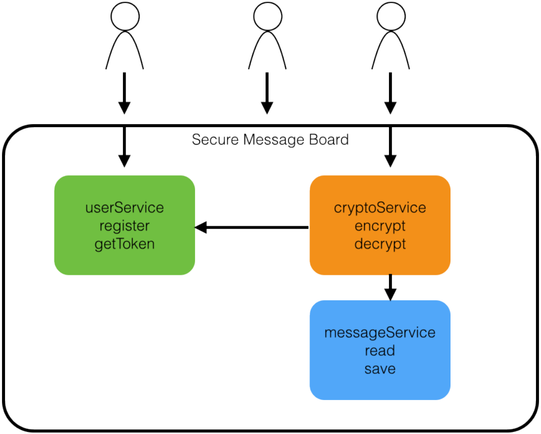
Where to cut?


Secure Message Board
userService
register
getToken
cryptoService
encrypt
decrypt
messageService
read
save

Where to cut?

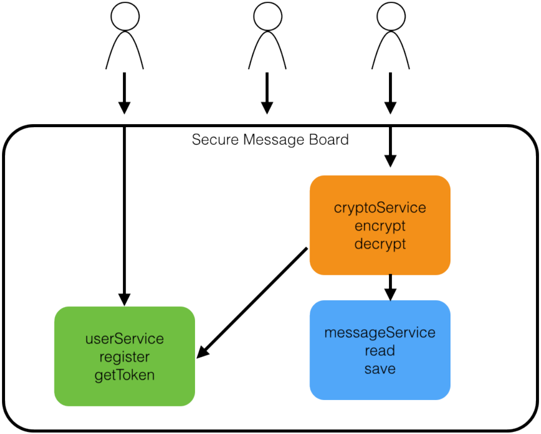

Secure Message Board
userService
register
getToken
cryptoService
encrypt
decrypt
messageService
read
save

Where to cut?



Secure Message Board
userService
register
getToken
cryptoService
encrypt
decrypt
messageService
read
save

Where to cut?


Secure Message Board
cryptoService
encrypt
decrypt
userService
register
getToken
messageService
read
save

API definition

```
GET /token/{name}
```

name is a string
Result is a string with the token
If user does not exist result state is 404

API definition

`GET /message/{id}`

id is a string
Result is a string with the message
If text does not exist result state is 404

`PUT /message/{id}`

id is a string
body is a string
Stores the **body** for this **id**
If **id** is used result state is 400

API definition

`GET /decrypt/{id}/{token}`

id is a string
token is a string
Result is a string.

Decrypts the message stored with **id** using the given **token**

`PUT /encrypt/{name}`

name is a string
body is a clear-text message
Result is the id of the encrypted message

Get's the token of **name**
Uses this token to encrypt the message

Let's do it

```
GET /token/{name}
```

User "alice" should get "abcde".
User "bob" should get "vwxyz".
User "charly" should get "fghij".
We will do the correct implementation later

Let's do it

`GET /message/{id}`

There is a basic implementation in the codebase for this training.
It contains one hard-coded message for each user (encrypted).
We will do the correct implementation later

hapi

```
let Hapi = require('hapi');
let server = new Hapi.Server();
server.connection({ host: "localhost", port: 8000 });
server.route({
  method: "GET",
  path: "/prefix/{var}",
  handler: function (request, reply) {
    // Allows to access:  request.params.var;
    // And the body as:   request.payload;
    reply("This will be a responded text");
  }
});
```

request

```
let request = require('request');
request("http://example.com", function (error, response,
  // If there was an error the error var will contain all
  // response contains headers of the response, including
  // body contains the payload of the response
}
```

Encryption

Input: a **key** and a **message** (both [a-z], message also ' ')
Output: An encrypted message.

> Transform each char (key & message) to a number a = 0, z = 25, ' ' = 26.
> **Add** the nth-char of key to nth-char of message, modulo 27.
> Transform the result back to characters.

Example: **key**: “abc”, **message**: “hello world”

msg	**h**	**e**	**l**	**l**	**o**		**w**	**o**	**r**	**l**	**d**
key	a	b	c	a	b	c	a	b	c	a	b
res	h	f	n	l	p	b	w	p	t	l	e

Decryption

Reverse: Input a **key** and an **encrypted message**
Output: An encrypted message.

> Transform each char (key & message) to a number a = 0, z = 25, ’ ’ = 26.
> **Subtract** nth-char of key from nth-char of message, modulo 27.
> Transform the result back to characters.

Example: **key**: “abc”, **encrypted message**: “hfnlpbwptle”

msg	**h**	**f**	**n**	**l**	**p**	**b**	**w**	**p**	**t**	**l**	**e**
key	a	b	c	a	b	c	a	b	c	a	b
res	h	e	l	l	o		w	o	r	l	d

Let's do it

- We use Vigenère cipher
- The token is used as a key
- `GET /decrypt/{id}/{token}` **implemented now**
  - We get a token and a message id as input
  - The message service can deliver the message by id
- `PUT /encrypt/{name}` **implemented later**
  - We get a username and a message as input
  - The user service get's the user's token
  - The message service can store any message (encrypt it before)

Explore

See your code in action

- We will setup the load balancer
- We use ApacheBench to simulate heavy user interaction
- You start and stop your servers
    - We should see throughput changes
    - The system should never stop working

Connect to Database

Just another service

- Logic-Services are independed from Database
- No local access
- Database is another service in the cluster
  - Independently scalable
- Many databases expose an HTTP API
  - Could directly be used
  - Or write a wrapping service
    - Just one place to modify for an update

ArangoDB

open source (Apache 2)
AQL offering joins & traversals
ACID including Multi Collection Transactions
sharding & replication

Foxx

Customized REST API on top of ArangoDB

Streamlined for Microservices

Reuse your Node.js code and NPM modules

Built-in authentication using OAuth2.0 or HTTP-Basic Auth

Operations are encapsulated in the database

low network traffic

direct data access

increases data privacy

```
 /\
(~(
 ) )     /\_/\
(  _-----_(@ @)
  (       \ /
  /|/--\|\ V
 " "    " "
```

manifest.json

```
{
  "name": "userService",
  "version": "1.0.0",
  "description": "The user service for Introduction to I
  "author": "Michael Hackstein",
  "license": "Apache 2 License",
  "contributors": [
    {
      "name": "Michael Hackstein",
      "email": "michael@arangodb.org"
    }
  ],
  "engines": { "arangodb": "^2.6.0" },
  "controllers": { "/": "index.js" },
  "defaultDocument": "",
  "scripts": { "setup": "setup.js" }
}
```

Controller

```
var Foxx = require("org/arangodb/foxx");
var controller = new Foxx.Controller(applicationContext

/** Short description
 *
 * Long description text
 */
 controller.get("/prefix/:variable", function (request,
   // Reacts on HTTP GET
   // Can use request.params("variable");
   // reply.send(...) for text/html
   // reply.json(...) for JSON
 }).pathParam("variable", { type: joi.string().descript
 ).errorResponse(Error, 404, "User not found");
```

Working with Collections

Get a colletion name

```
var col = applicationContext.collectionName("myCollecti
```

Create a collection

```
var db = require("internal").db;
var colObj = db._create(col);
```

Get a collection object.

```
var db = require("internal").db;
var colObj = db._collection(col);
// colObj might be null if the collection does not exis
// Alternative:
var colObj = applicationContext.collection("myCollectio
```

Working with Collections

Save a document

```
colObj.save({foo: "bar"});
colObj.save({_key: "123", foo: "bar"});
```

Get a document by key

```
colObj.document("123");
```

Setup

Check if the collection is already there
if not create it

Fill it with sample data ‘

Hands-on

Start with the manifest.json

Continue with setup

Finish with the controller

Install

```
foxx-manager install <path to app folder> /token
```

Update

```
foxx-manager replace <path to app folder> /token
```

Hands-on

Start with the manifest.json

Continue with setup

Finish with the controller

Install

```
foxx-manager install <path to app folder> /message
```

Update

```
foxx-manager replace <path to app folder> /message
```

Hands-on

`PUT /encrypt/{name}`

We get a username and a message as input

The user service get's the user's token

The message service can store any message (encrypt it before)

Conclusion

Cons

No silver bullet
Multiple Machines
Design for failure
Requires infrastructure
Slower than a single machine with equal specs

Pro

- Easy scaling
  - Scale only hot-spots
- Easy work distribution
- Incremental updates
  - Fits well into agile development
- “Zero downtime”

Created with Madoko.net.