Microservices
and
Modularity

or
the difference between

treatment
and
cure

Milen Dyankov
@milendyankov

JPoint
07.04.2017 Moscow

```
~$ pidof java
```

```
~$ pidof java
9927 2151
~$
```

```
~$ pidof java
9927 2151
~$ ps aux | grep java | grep -v grep | awk '{print $2}'
```

```
~$ pidof java
9927 2151
~$ ps aux | grep java | grep -v grep | awk '{print $2}'
2151
9927
~$
```


```
~$ pidof java
9927 2151
~$ ps aux | grep java | grep -v grep | awk '{print $2}'
2151
9927
~$
```

Microservices

What are Microservices ?

martinfowler.com/articles/microservices.html

MARTIN FOWLER

Intro Design Agile Refactoring NoSQL DSL Delivery About Me ThoughtWorks

Microservices

The term "Microservice Architecture" has sprung up over the last few years to descri particular way of designing software applications as suites of independently deployab services. While there is no precise definition of this architectural style, there are cert common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

25 March 2014

James Lewis

James Lewis is a Principal Consultant at ThoughtWorks and member of the Technology Advisory Board. James' interest in building applications out of small collaborating services stems from a background in integrating enterprise systems at scale. He's built a number of systems using microservices and has been an active participant in the growing community for a couple of years.

Martin Fowler

Martin Fowler is an author, speaker, and general loud-mouth on software development. He's long been puzzled by the problem of how to componentize software systems, having heard more vague claims than he's happy with. He hopes that microservices will live up to the early promise its advocates have found.

Translations: Japanese · Russian ·

Find **similar articles** to this by looking at these tags: popular · application architecture · web services · microservices

Contents

Microservices characteristics!

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

martinfowler.com/articles/microservices.html

MARTIN FOWLER

Intro Design Agile Refactoring NoSQL DSL Delivery About Me ThoughtWorks

Microservices

The term "Microservice Architecture" has sprung up over the last few years to descri particular way of designing software applications as suites of independently deployab services. While there is no precise definition of this architectural style, there are cert common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

25 March 2014

James Lewis

James Lewis is a Principal Consultant at ThoughtWorks and member of the Technology Advisory Board. James' interest in building applications out of small collaborating services stems from a background in integrating enterprise systems at scale. He's built a number of systems using microservices and has been an active participant in the growing community for a couple of years.

Martin Fowler

Martin Fowler is an author, speaker, and general loud-mouth on software development. He's long been puzzled by the problem of how to componentize software systems, having heard more vague claims than he's happy with. He hopes that microservices will live up to the early promise its advocates have found.

Translations: Japanese · Russian ·

Find **similar articles** to this by looking at these tags: popular · application architecture · web services · microservices

Contents

50% not strictly software but rather operations related!

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

Why consider

Microservices

Reducing the complexity of Monoliths

wrong but common answer

www.business.com/it-consulting/why-microservices-are-good-for-businesses/

The main benefit of using microservices is that, unlike a monolithic architecture style, a change made to a small part of the application does not require the entire structure to be rebuilt and redeployed (Tweet This!). This results in much less, if not zero downtime.

www.pwc.com/us/en/technology-forecast/2014/cloud-computing/features/microservices.jhtml

Why microservices?

In the software development community, it is an article of faith that apps should be written with standard application programming interfaces (APIs), using common services when possible, and managed through one or more orchestration technologies. Often, there's a superstructure of middleware, integration methods, and management tools. That's great for software designed to handle complex tasks for long-term, core enterprise functions—it's how transaction systems and other systems of record need to be designed.

Greater modularity, loose coupling, and reduced dependencies all hold promise in simplifying the integration task.

But these methods hinder what Silicon Valley companies call web-scale development: software that must evolve quickly, whose functionality is subject to change or obsolescence in a couple of years—even months—and where the level of effort must fit a compressed and reactive schedule. It's more like web page design than developing traditional enterprise software.

opensource.com/business/14/12/containers-microservices-and-orchestrating-whole-symphony

So, what are microservices really and how does this architecture improve delivery cycles?

Microservices were developed as a way to divide and conquer.

Basically, the microservices approach in a nutshell dictates that instead of having one giant code base that all developers touch, that often times becomes perilous to manage, that there are numerous smaller code bases managed by small and agile teams. The only dependency these code bases have on one another is their APIs. This means that as long as you maintain backwards and forward compatibility (which albeit is not that trivial), each team can work in release cycles that are decoupled from other teams. There are some scenarios where these release cycles are coupled, where one service depends on another or depends on a new feature in another service, but this is not the usual case.

www.infoq.com/news/2014/05/microservices

Some of the benefits of microservices are pretty obvious:

- Each microservice is quite simple being focused on one business capability
- Microservices can be developed independently by different teams
- Microservices are loosely coupled
- Microservices can be developed using different programming languages and tools

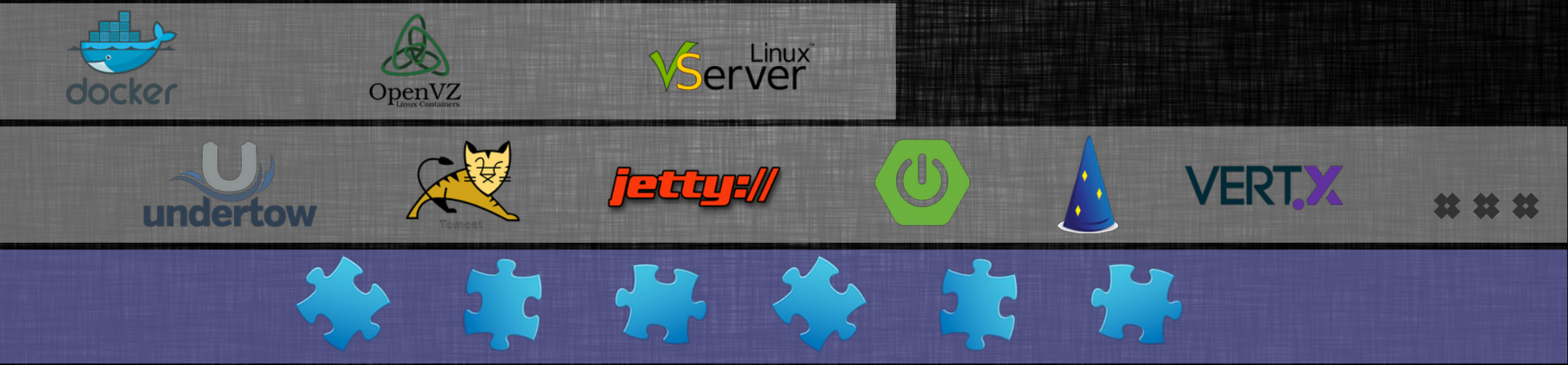

undertow

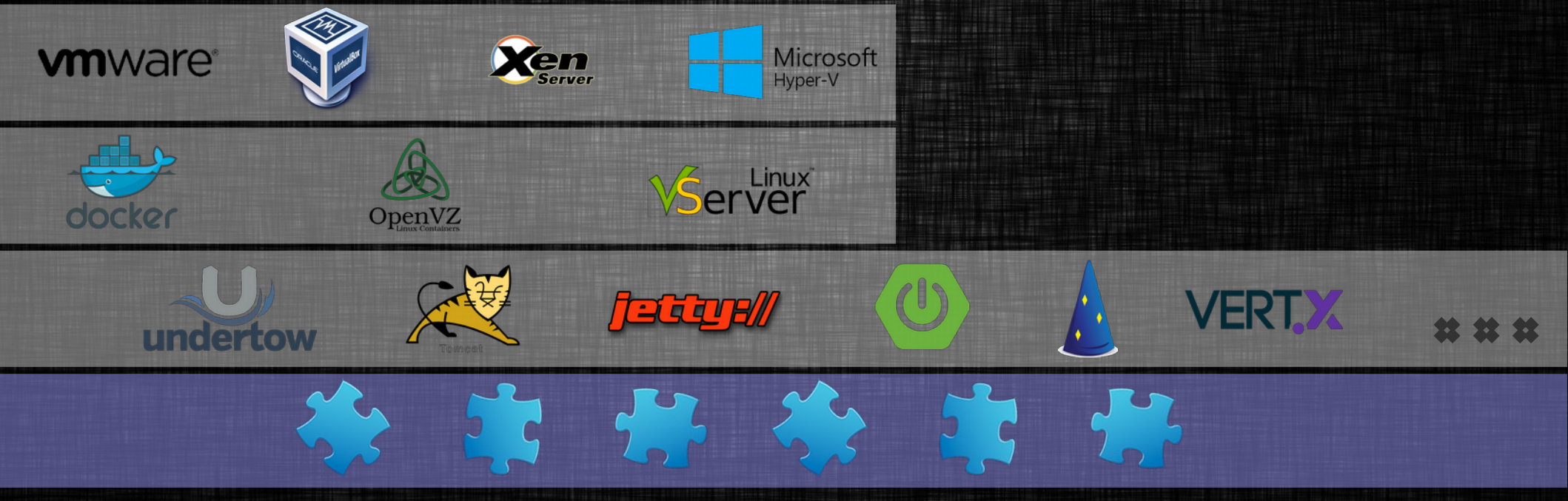
Tomcat

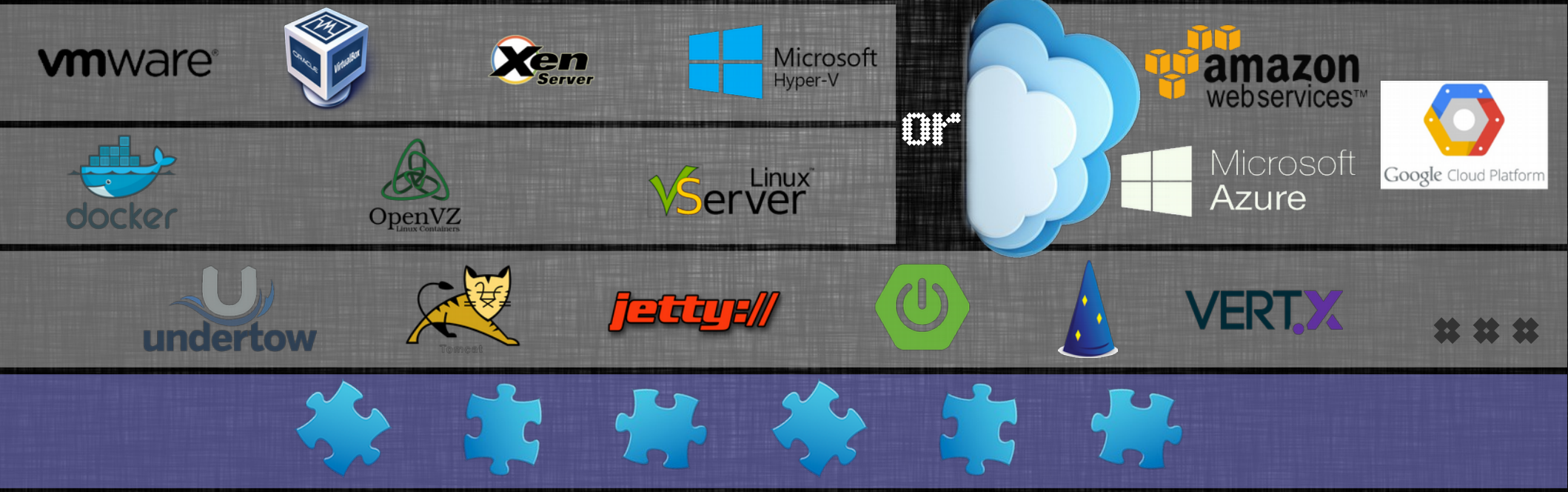
jetty://



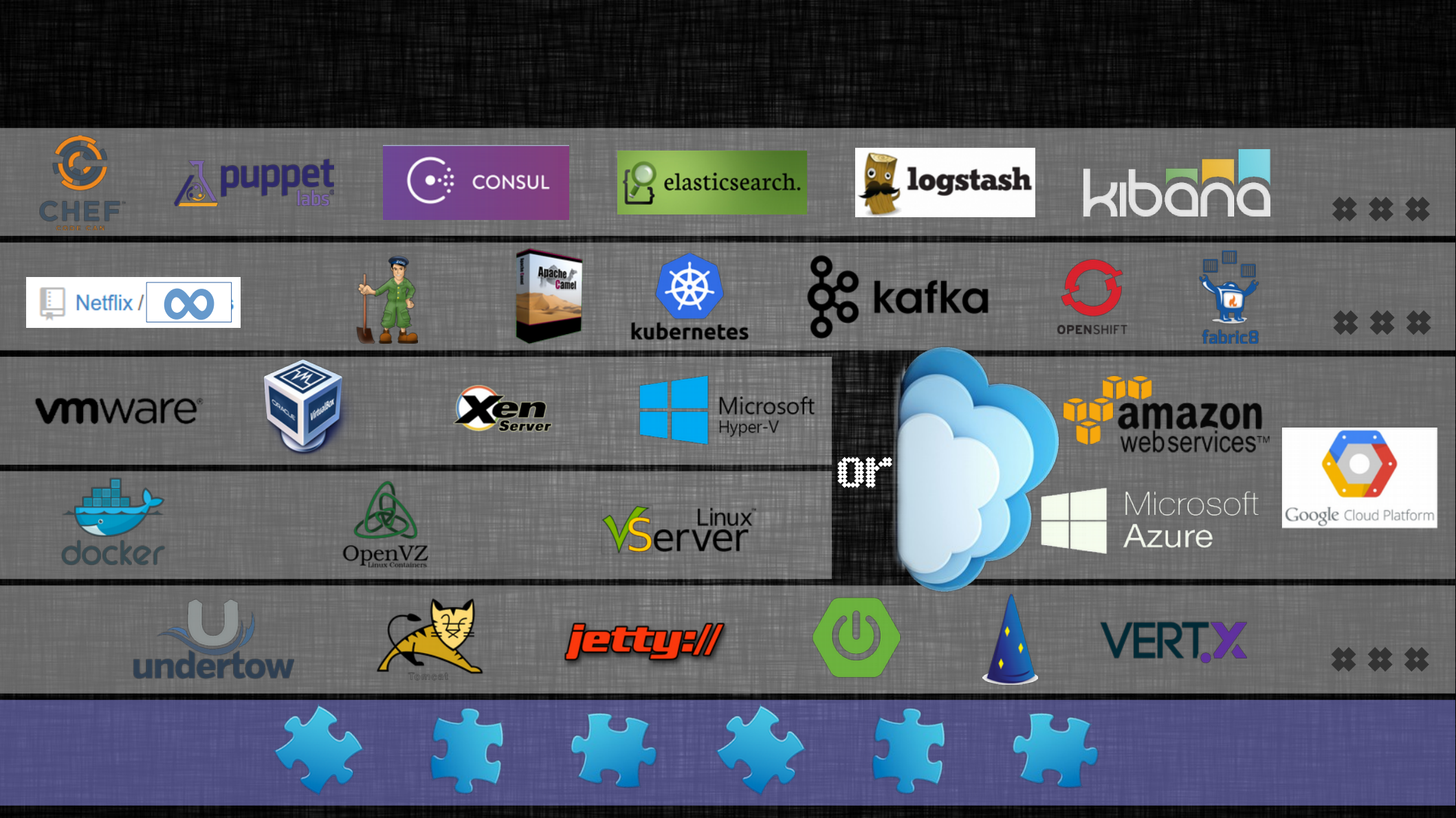
VERT.X


docker
OpenVZ
Linux Containers
Linux
VServer
undertow
Tomcat
jetty://
VERT.X


vmware
Xen Server
Microsoft Hyper-V
docker
OpenVZ
Linux VServer
undertow
Tomcat
jetty://
VERT.X


vmware
Xen Server
Microsoft Hyper-V
docker
OpenVZ
Linux Containers
Linux VServer
or
amazon web services
Microsoft Azure
Google Cloud Platform
undertow
Tomcat
jetty://
VERT.X


Netflix /
Apache Camel
kubernetes
kafka
OPENSHIFT
fabric8
vmware
Xen Server
Microsoft Hyper-V
or
amazon web services™
Google Cloud Platform
Microsoft Azure
docker
OpenVZ Linux Containers
Linux VServer
undertow
Tomcat
jetty://
VERT.X


CHEF
puppet labs
CONSUL
elasticsearch.
logstash
kibana
Netflix /
Apache Camel
kubernetes
kafka
OPENSHIFT
fabric8
vmware
Xen Server
Microsoft Hyper-V
or
amazon web services
Google Cloud Platform
Microsoft Azure
docker
OpenVZ
Linux VServer
undertow
Tomcat
jetty://
VERT.X

REDUCED COMPLEXITY

martinfowler.com/articles/microservices.html

Often the true consequences of your architectural decisions are only evident several years after you made them. We have seen projects where a good team, with a strong desire for modularity, has built a monolithic architecture that has decayed over the years. Many people believe that such decay is less likely with microservices, since the service boundaries are explicit and hard to patch around. Yet until we see enough systems with enough age, we can't truly assess how microservice architectures mature.

There are certainly reasons why one might expect microservices to mature poorly. In any effort at componentization, success depends on how well the software fits into components. It's hard to figure out exactly where the component boundaries should lie. Evolutionary design recognizes the difficulties of getting boundaries right and thus the importance of it being easy to refactor them. But when your components are services with remote communications, then refactoring is much harder than with in-process libraries. Moving code is difficult across service boundaries, any interface changes need to be coordinated between participants, layers of backwards compatibility need to be added, and testing is made more complicated.

Another issue is If the components do not compose cleanly, then all you are doing is shifting complexity from inside a component to the connections between components. Not just does this just move complexity around, it moves it to a place that's less explicit and harder to control. It's easy to think things are better when you are looking at the inside of a small, simple component, while missing messy connections between services.

Finally, there is the factor of team skill. New techniques tend to be adopted by more skillful teams. But a technique that is more effective for a more skillful team isn't necessarily going to work for less skillful teams. We've seen plenty of cases of less skillful teams building messy monolithic architectures, but it takes time to see what happens when this kind of mess occurs with microservices. A poor team will always create a poor system - it's very hard to tell if microservices reduce the mess in this case or make it worse.

One reasonable argument we've heard is that you shouldn't start with a microservices architecture. Instead begin with a monolith, keep it modular, and split it into microservices once the monolith becomes a problem. (Although this advice isn't ideal, since a good in-process interface is usually not a good service interface.)

So we write this with cautious optimism. So far, we've seen enough about the microservice style to feel that it can be a worthwhile road to tread. We can't say for sure where we'll end up, but one of the challenges of software development is that you can only make decisions based on the imperfect information that you currently have to hand.

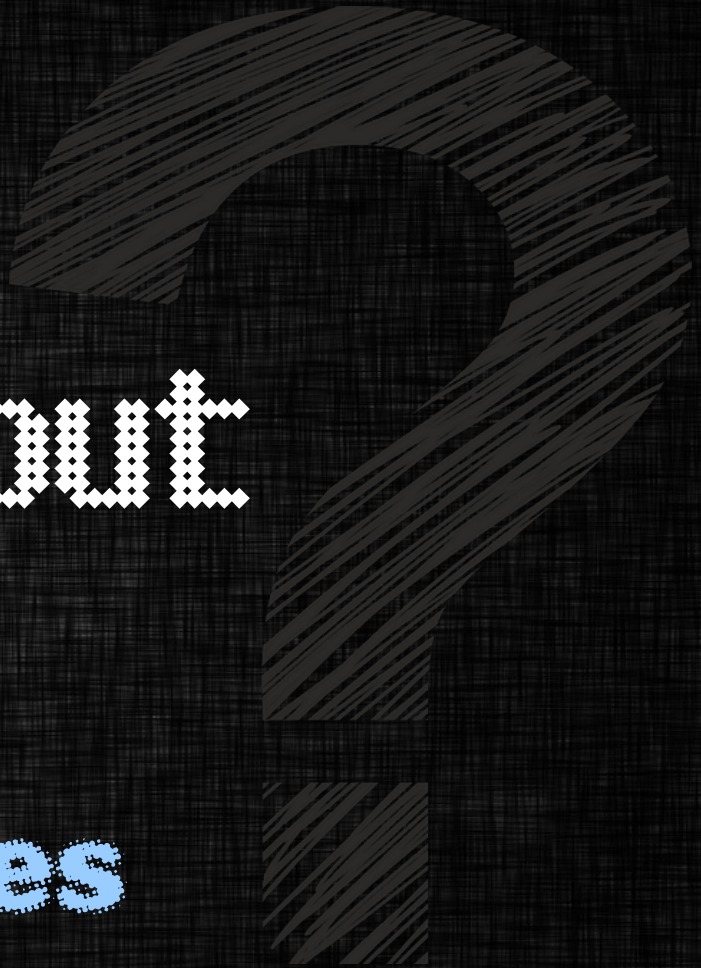
. . .

then all you are doing is shifting complexity from inside a component to the connections between components.

. . .

it moves it to a place that's less explicit and harder to control.

What's cool about



Microservices

“The real power … is the ability for a developer to develop a single entity and then deploy that component multiple times”

“Highly Scalable, Robust, Architecture”

“In very straightforward terms … is a component model for building portable, reusable and scalable business components … for distributed environment.”

Quotes from articles about EJB

(1999 - 2002)

"The real power ... is the ability for a developer to develop a single entity and then deploy that component multiple times"

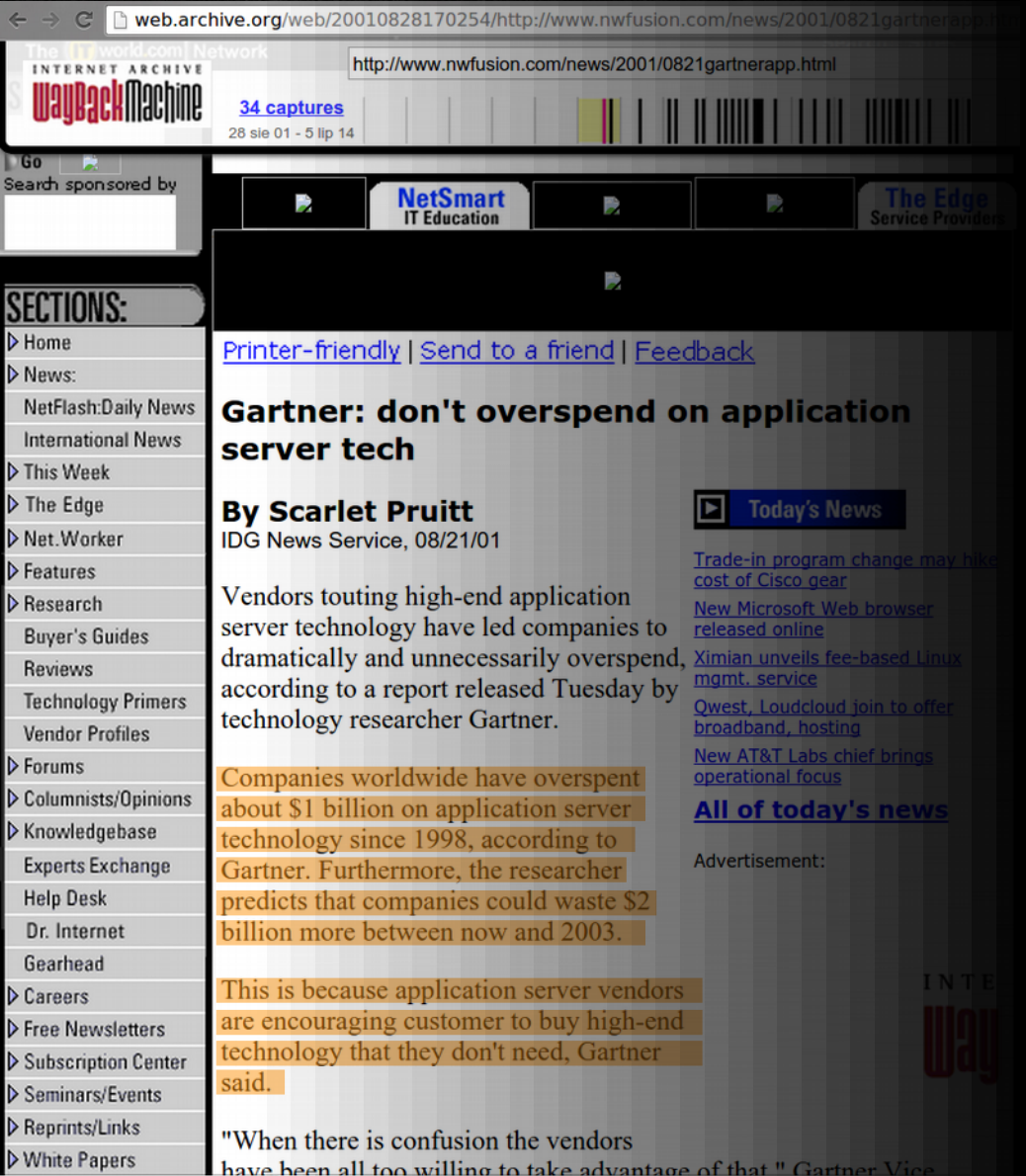
www.onjava.com/pub/a/onjava/2001/12/19/eejbs.html

"Highly Scalable, Robust, Architecture"

www.dhlee.info/computing/ejb/reference/seybold_ejb.pdf

"In very straightforward terms ... is a component model for building portable, reusable and scalable business components ... for distributed environment."

www.idt.mdh.se/kurser/ct3340/archives/ht08/papersRM08/37.pdf



...

Companies worldwide have overspent about $1 billion

...

vendors are encouraging customer to buy high-end technology that they don't need.

Who is doing

Microservices


MICROSERVICES
NETFLIX
amazon.com
GILT
SOUNDCLOUD
LinkedIn
allegro
karma
the guardian

What do they have in common ?

They build
microservices for
their own needs!

They build microservices for their own needs!

This makes it easer for them to

- grow the DevOps culture
- hire the right people
- accept "Decentralized" approach
- automate infrastructure

martinfowler.com/articles/microservices.html

Often the true consequences of your architectural decisions are only evident several years after you made them. We have seen projects where a good team, with a strong desire for modularity, has built a monolithic architecture that has decayed over the years. Many people believe that such decay is less likely with microservices, since the service boundaries are explicit and hard to patch around. Yet until we see enough systems with enough age, we can't truly assess how microservice architectures mature.

There are certainly reasons why one might expect microservices to mature poorly. In any effort at componentization, success depends on how well the software fits into components. It's hard to figure out exactly where the component boundaries should lie. Evolutionary design recognizes the difficulties of getting boundaries right and thus the importance of it being easy to refactor them. But when your components are services with remote communications, then refactoring is much harder than with in-process libraries. Moving code is difficult across service boundaries, any interface changes need to be coordinated between participants, layers of backwards compatibility need to be added, and testing is made more complicated.

Another issue is If the components do not compose cleanly, then all you are doing is shifting complexity from inside a component to the connections between components. Not just does this just move complexity around, it moves it to a place that's less explicit and harder to control. It's easy to think things are better when you are looking at the inside of a small, simple component, while missing messy connections between services.

Finally, there is the factor of team skill. New techniques tend to be adopted by more skillful teams. But a technique that is more effective for a more skillful team isn't necessarily going to work for less skillful teams. We've seen plenty of cases of less skillful teams building messy monolithic architectures, but it takes time to see what happens when this kind of mess occurs with microservices. A poor team will always create a poor system - it's very hard to tell if microservices reduce the mess in this case or make it worse.

One reasonable argument we've heard is that you shouldn't start with a microservices architecture. Instead begin with a monolith, keep it modular, and split it into microservices once the monolith becomes a problem. (Although this advice isn't ideal, since a good in-process interface is usually not a good service interface.)

So we write this with cautious optimism. So far, we've seen enough about the microservice style to feel that it can be a worthwhile road to tread. We can't say for sure where we'll end up, but one of the challenges of software development is that you can only make decisions based on the imperfect information that you currently have to hand.

. . .

a technique that is more effective
for a more skillful team
isn't necessarily going to work
for less skillful teams

. . .

A poor team will always create
a poor system

. . .

Should I do

Microservices

Does your organization
fit into that space?

martinfowler.com/articles/microservices.html

Often the true consequences of your architectural decisions are only evident several years after you made them. We have seen projects where a good team, with a strong desire for modularity, has built a monolithic architecture that has decayed over the years. Many people believe that such decay is less likely with microservices, since the service boundaries are explicit and hard to patch around. Yet until we see enough systems with enough age, we can't truly assess how microservice architectures mature.

There are certainly reasons why one might expect microservices to mature poorly. In any effort at componentization, success depends on how well the software fits into components. It's hard to figure out exactly where the component boundaries should lie. Evolutionary design recognizes the difficulties of getting boundaries right and thus the importance of it being easy to refactor them. But when your components are services with remote communications, then refactoring is much harder than with in-process libraries. Moving code is difficult across service boundaries, any interface changes need to be coordinated between participants, layers of backwards compatibility need to be added, and testing is made more complicated.

Another issue is If the components do not compose cleanly, then all you are doing is shifting complexity from inside a component to the connections between components. Not just does this just move complexity around, it moves it to a place that's less explicit and harder to control. It's easy to think things are better when you are looking at the inside of a small, simple component, while missing messy connections between services.

Finally, there is the factor of team skill. New techniques tend to be adopted by more skillful teams. But a technique that is more effective for a more skillful team isn't necessarily going to work for less skillful teams. We've seen plenty of cases of less skillful teams building messy monolithic architectures, but it takes time to see what happens when this kind of mess occurs with microservices. A poor team will always create a poor system - it's very hard to tell if microservices reduce the mess in this case or make it worse.

One reasonable argument we've heard is that you shouldn't start with a microservices architecture. Instead begin with a monolith, keep it modular, and split it into microservices once the monolith becomes a problem. (Although this advice isn't ideal, since a good in-process interface is usually not a good service interface.)

So we write this with cautious optimism. So far, we've seen enough about the microservice style to feel that it can be a worthwhile road to tread. We can't say for sure where we'll end up, but one of the challenges of software development is that you can only make decisions based on the imperfect information that you currently have to hand.

...

you shouldn't start with a microservices architecture.
Instead begin with a monolith,
keep it modular,
and split it into microservices
once the monolith
becomes a problem.

...

What else, if not

Microservices

Microservices do not

cure complexity!

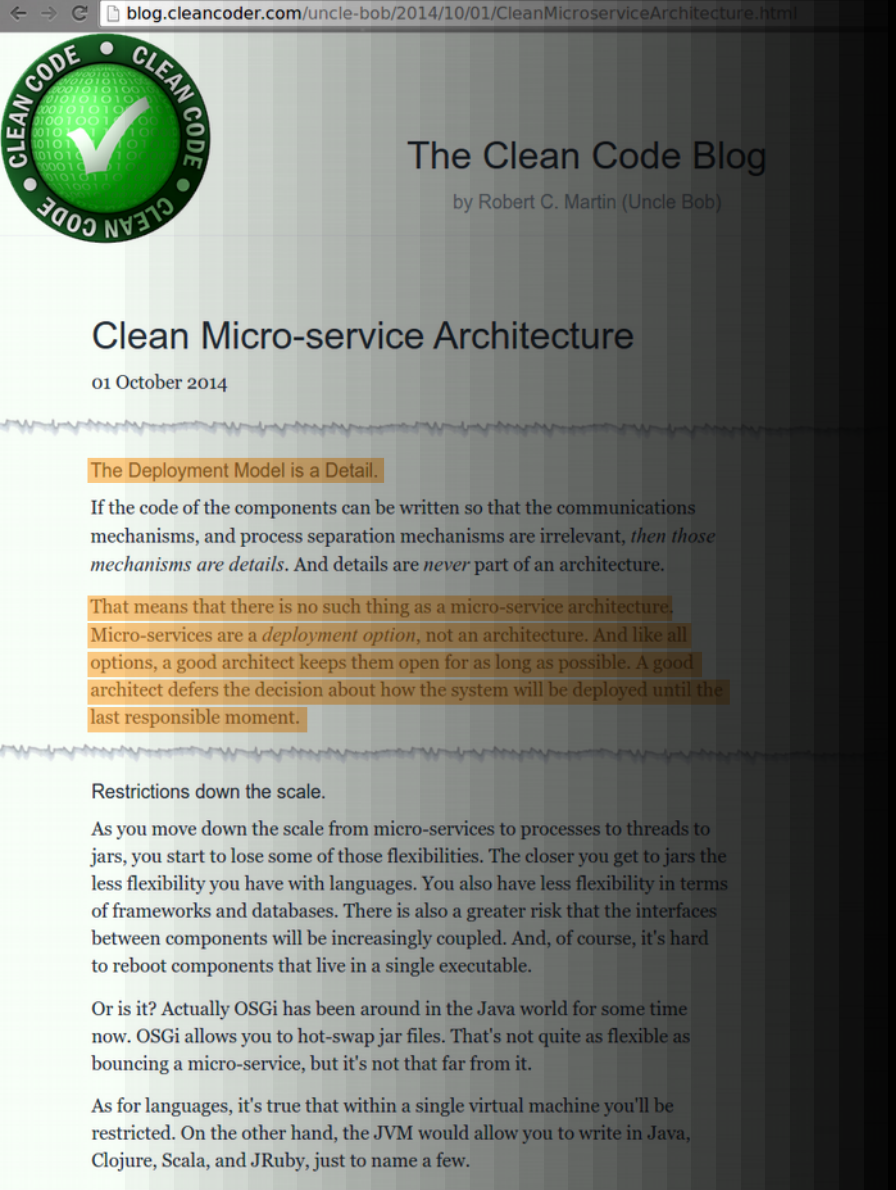
Actually nothing does !

The term "cure" means that, after medical treatment, the patient no longer has that particular condition anymore.

Some diseases have no cure. The patient will always have the condition, but treatment can help to manage it.

Good treatment for complexity is enforcing

clean modular architecture



. . .

The Deployment Model is a Detail.

. . .

there is no such thing as a micro-service architecture.

Micro-services are a deployment option

. . .

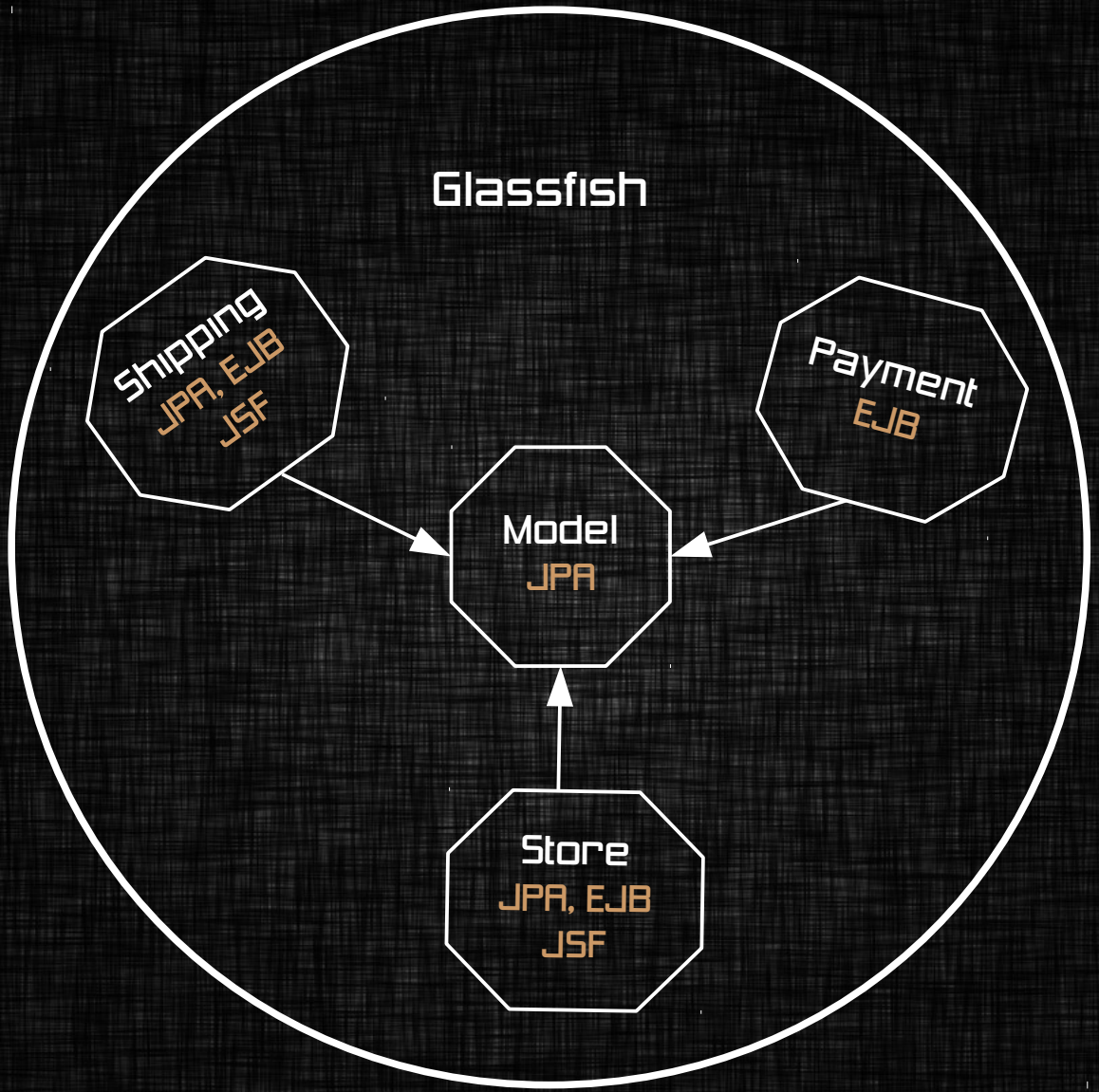
Interesting !?!

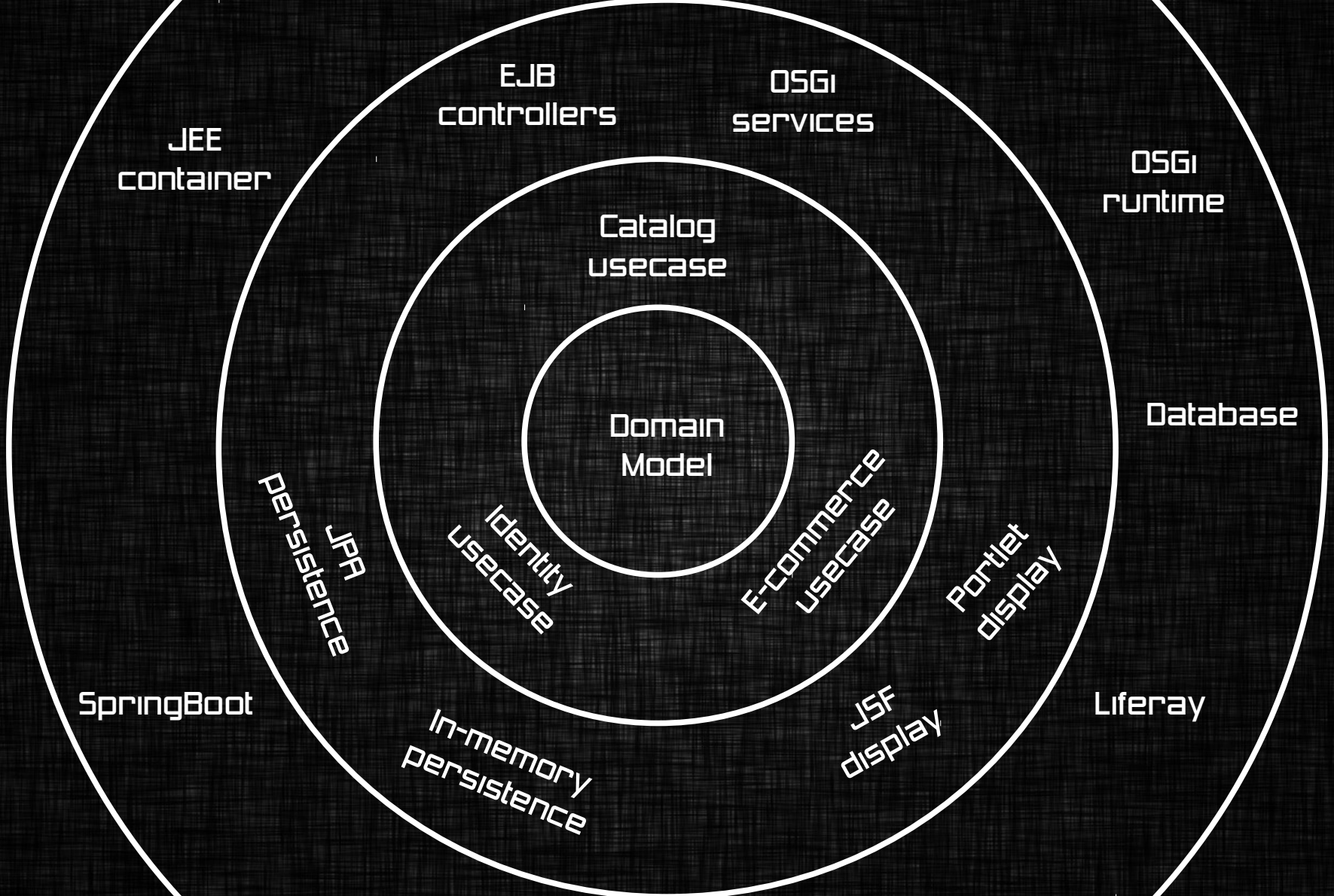
But in my project it's not possible because of . . .

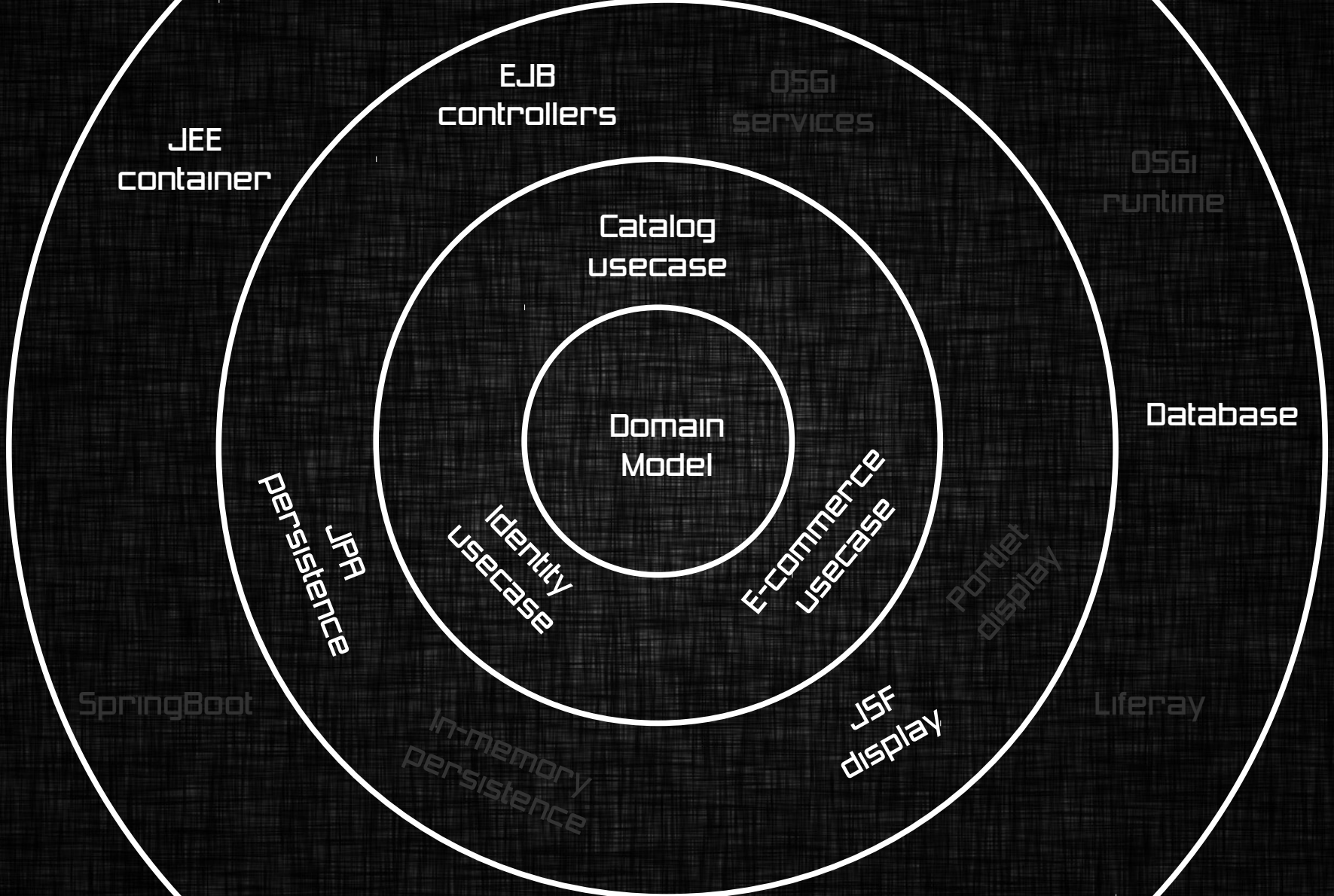
Really ?!?

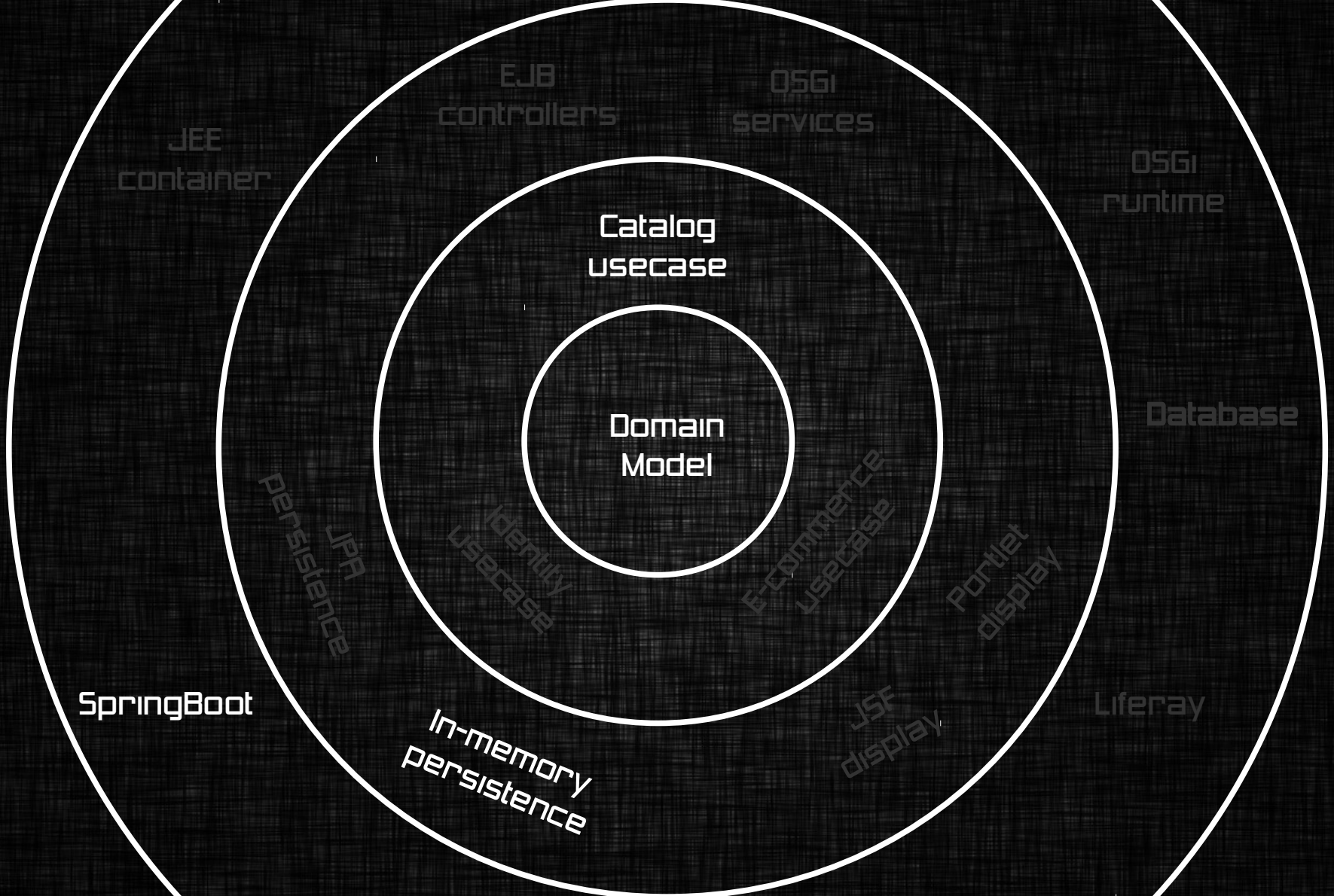
Modularizing "Duke's forest" JEE tutorial demo!

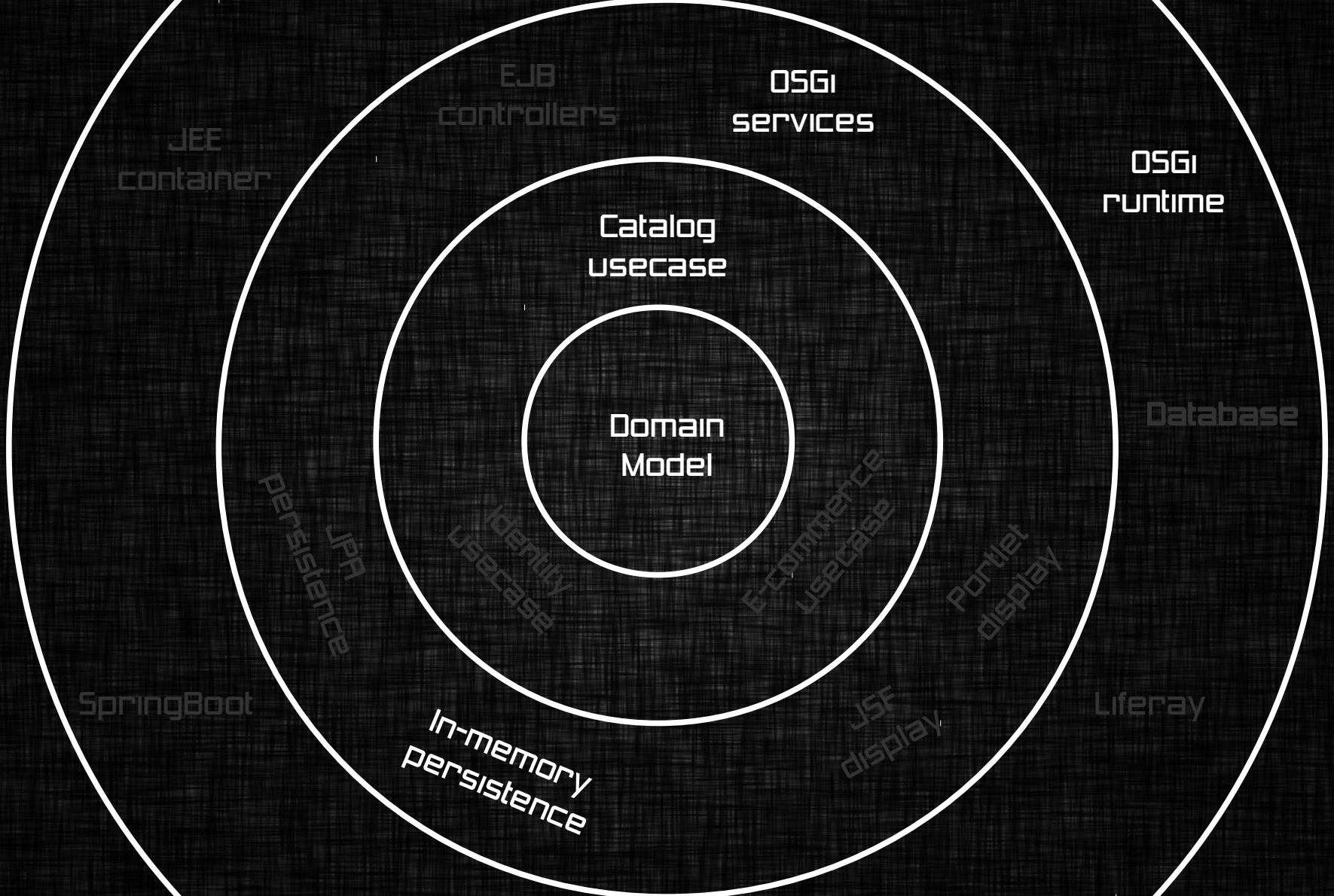
https://github.com/azzazzel/modular-dukes-forest

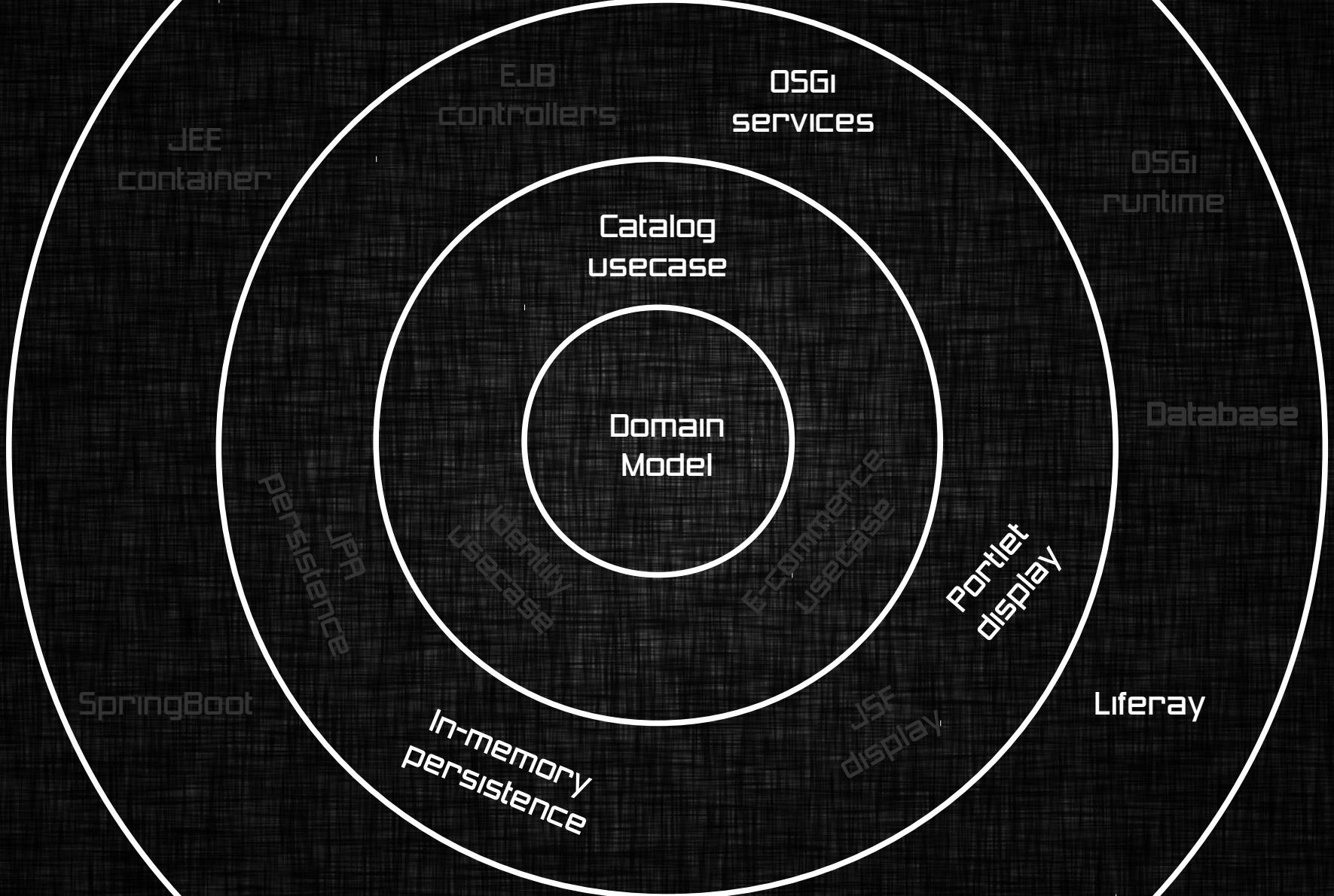

Glassfish
Shipping
JPA, EJB
JSF
Payment
EJB
Model
JPA
Store
JPA, EJB
JSF

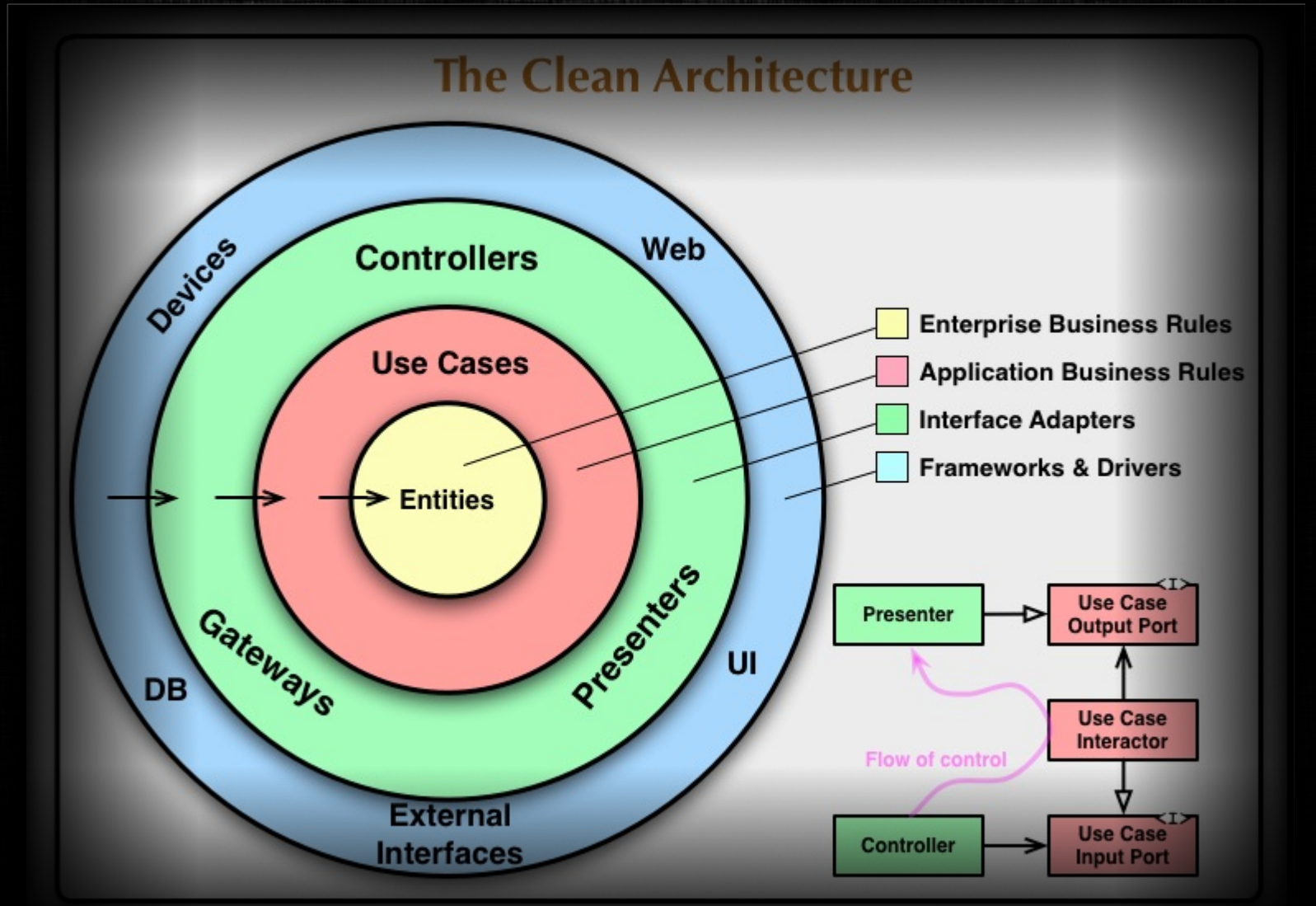

EJB
controllers
OSGi
services
JEE
container
OSGi
runtime
Catalog
usecase
Domain
Model
Database
JPA
persistence
Identity
usecase
E-commerce
usecase
Portlet
display
SpringBoot
In-memory
persistence
JSF
display
Liferay


EJB
controllers
OSGi
services
JEE
container
OSGi
runtime
Catalog
usecase
Domain
Model
Database
JPA
persistence
Identity
usecase
E-commerce
usecase
Portlet
display
SpringBoot
In-memory
persistence
JSF
display
Liferay


EJB
controllers
OSGi
services
JEE
container
OSGi
runtime
Catalog
usecase
Domain
Model
Database
JPA
persistence
Identity
usecase
E-commerce
usecase
Portlet
display
SpringBoot
In-memory
persistence
JSF
display
Liferay


EJB
controllers
OSGi
services
JEE
container
OSGi
runtime
Catalog
usecase
Domain
Model
Database
JPA
persistence
Identity
usecase
E-commerce
usecase
Portlet
display
SpringBoot
In-memory
persistence
JSF
display
Liferay


EJB
controllers
OSGi
services
JEE
container
OSGi
runtime
Catalog
usecase
Domain
Model
Database
JPA
persistence
Identity
usecase
E-commerce
usecase
Portlet
display
SpringBoot
In-memory
persistence
JSF
display
Liferay



http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

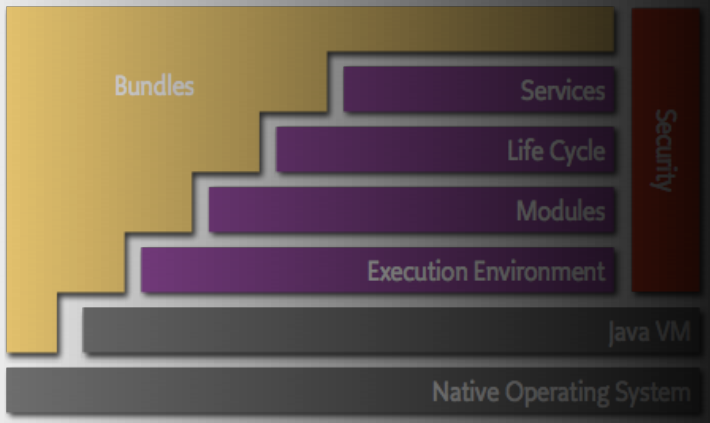
Modularity

Is a important software architecture concept!

One can design modular application without

Microservices

OSGi™ Alliance



The OSGi specification describes a modular system and a service platform for the Java programming language

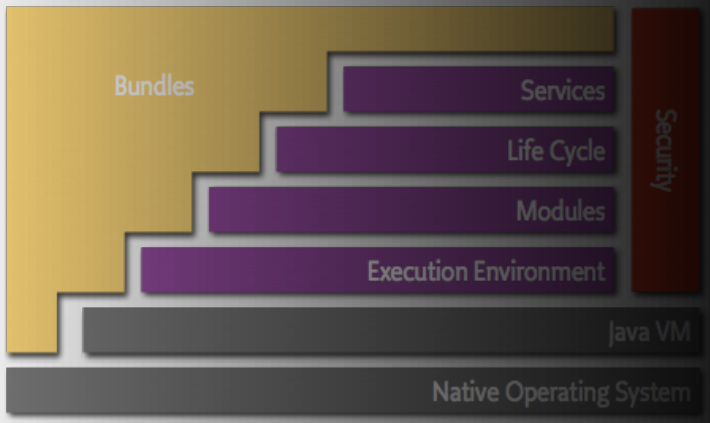
The architecture of choice for

LIFERAY®

and

- Confluence
- Eclipse
- Fuse ESB
- Glassfish
- Jboss
- JIRA
- JonAS
- Service Mix
- Weblogic
- Websphere
- . . .

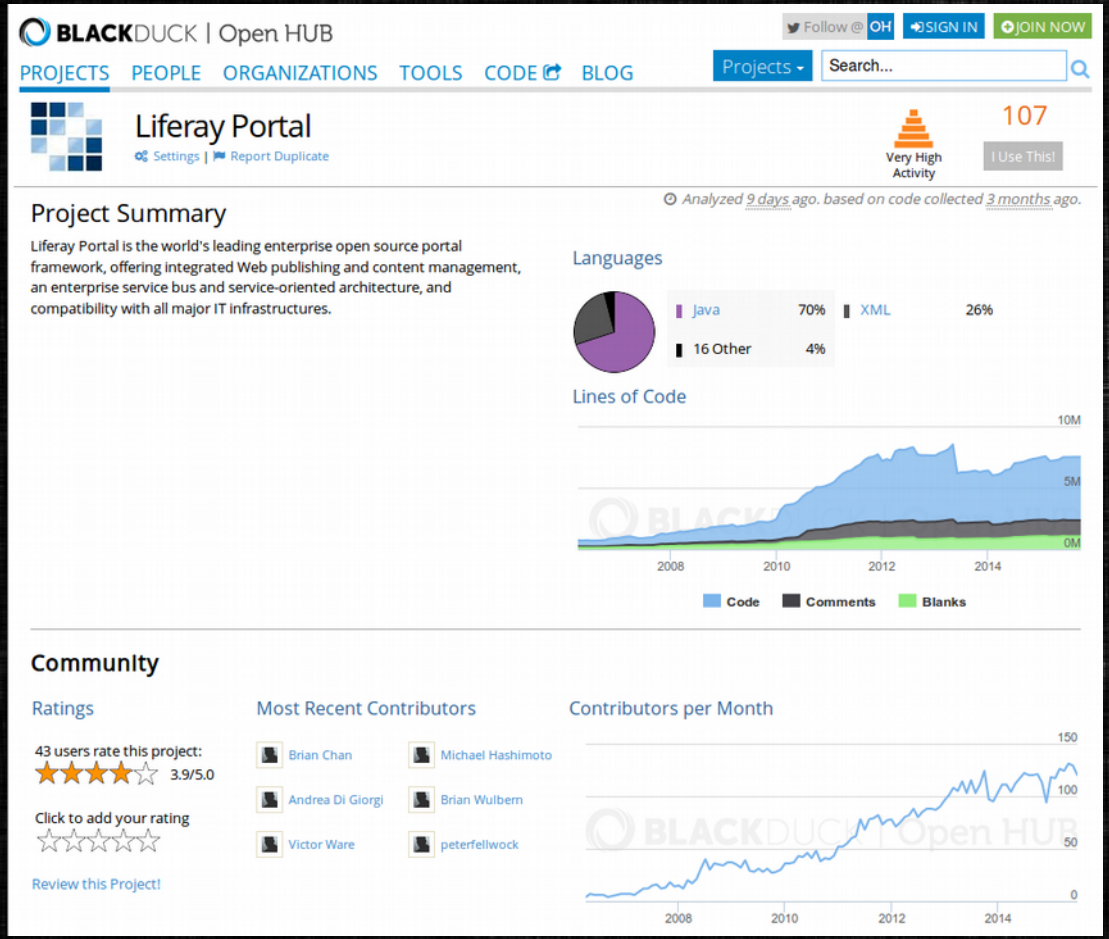
OSGi™ Alliance



Same characteristics but more flexible !

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

This is not theory! We do this at LIFERAY

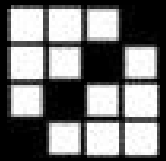


- 1 platform
- over 100 apps
- over 600 modules
- over 2500 µServices

milen.dyankov@liferay.com

http://MilenDyankov.com
http://www.liferay.com/web/milen.dyankov/

@milendyankov
@LiferayPL



http://www.liferay.com

@Liferay

http://www.facebook.com/Liferay