Big data processing the lean way – a case study

by Siegfried Steiner (steiner@comcodes.com)

19/08/14 – 17/01/21

Content

1. Preface

In this case study, I use the present tense although everything took place in the past – within the years 2012 and 2013. I decided to place footnotes even for terms which might be considered generally known. Often terms are part of a specific domain with which not everybody of the audience necessarily is familiar with (a message selector[1] in Smalltalk[2] and a method signature[3] in Java can mean the same thing in their specific domain – a Java and a Smalltalk geek talking though might misunderstand each other). My footnotes usually denote secondary sources such as Wikipedia, which is not bullet proof from a scientific point of view. Every term once put into a footnote is written in italic style. As assuming something to be "secure" is already an error in reasoning, I put the word "secure" consequently into quotes. Regarding the diagrams I use a "no notation", borrowing from different notations at random to get different aspects such as structure and dynamics into a single self explanatory visualization. In case you find new findings in this paper, then this is by coincidence. This paper demonstrates on how to recombine "old wine in new containers" effectively ...

2. Abstract

Nowadays *big data*[4] seems to be everywhere: Monstrous amounts of data to be piped through *software systems*[5], to be digested by *services*[6], to be fed back into the circulation; for you to finally consume an individual view of the world. In times of *cloud computing*[7], putting your *big data* ideas into reality is no rocket science any more, even for small development teams with no *data center*[8] at hands.

In this session I will introduce you a promising approach on how to cope with

1 *Messages (Smalltalk)*, see http://en.wikipedia.org/wiki/Smalltalk#Messages (*Wikipedia*)
2 *Smalltalk*, see http://en.wikipedia.org/wiki/Smalltalk (*Wikipedia*)
3 *Method signature*, see http://en.wikipedia.org/wiki/Method_signature (*Wikipedia*)
4 *Big data*, see http://en.wikipedia.org/wiki/Big_data (*Wikipedia*)
5 *Software system*, see http://en.wikipedia.org/wiki/Software_system (*Wikipedia*)
6 *Service*, see http://en.wikipedia.org/wiki/Service_%28systems_architecture%29 (*Wikipedia*)
7 *Cloud computing*, see http://en.wikipedia.org/wiki/Cloud_computing (*Wikipedia*)
8 *Data center*, see http://en.wikipedia.org/wiki/Data_centre (*Wikipedia*)

big data projects the lean way.

Managing *big data* in the *cloud* bears a highly dynamic runtime behaviour. This is especially true, when costs matter: How to *scale*[9] up and *scale* down your *cloud* resource consumption? Which technology stack works promisingly well? Which *design patterns*[10] and *software architectures*[11] are suitable? How to populate and organize your *software systems* in this dynamic environment without loosing track? How to retain a clear view of your *services*' and *software systems*' health condition?

By dissecting a *case study* in this session, I'll introduce you a promising approach on how to tackle those challenges using the *Java* ecosystem, *Amazon's AWS*[12] and *NoSQL*[13] *databases*; in an *agile* working environment; breaking down your *software architecture* into subtle bits and pieces to just put the back together again.

3. The mission

Speaking in business terms, the mission taken for this *case study* is to improve (boost) a *webshop's* customer acquisition numbers as well as tighten the customer's loyalty with according services to be provided:

1. "*Learn from the customers' journeys*[14] *and their buying patterns*"
2. "*Optimize customer- and product proposal communication*"
3. "*Increase in sales and reduce in customer related costs*"

Technically speaking the mission is on "*put sensors into webshops' pages of special interest and record the users' journeys (clicks) through those web pages. Digest the recorded data to produce individual consumer profiles and provide the webshops with individual feedback on those user profiles – transaction based or batch based. "Provide" means to do it for many*

9 *Scalability*, see http://en.wikipedia.org/wiki/Scalability (Wikipedia)
10 *Design pattern*, see http://en.wikipedia.org/wiki/Software_design_pattern (*Wikipedia*)
11 *Software architecture,* see http://en.wikipedia.org/wiki/Software_architecture (*Wikipedia*)
12 *Amazon Web Services*, see http://en.wikipedia.org/wiki/Amazon_Web_Services (*Wikipedia*)
13 *NoSQL*, see http://en.wikipedia.org/wiki/NoSQL (*Wikipedia*)
14 *Customer Journey*, see http://de.wikipedia.org/wiki/Customer_Journey (*Wikipedia*)

webshops, smaller and bigger ones".

We are to provide the *software system* as a *SaaS*[15] platform, supporting multi-tenant capabilities, e.g. serving many different *webshops*. *1* Illustrates a simplified view of the mission's set-up:

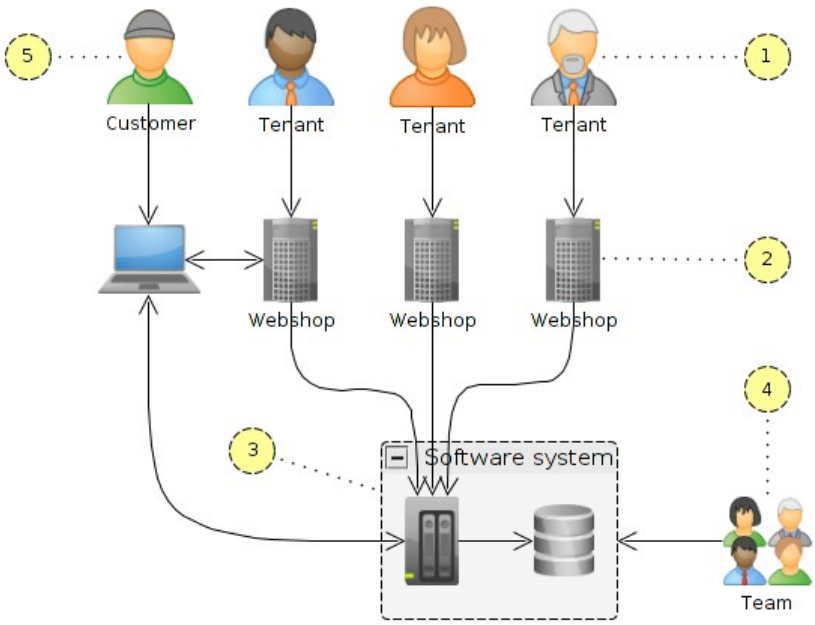

Figure 1: A rough impression on the mission's set-up.

The *tenants* (1) attach their *webshops* (2) to the *software system* (3) being implemented by the *team* (4). The *customers*' (5) *journeys* are analysed, individual feedback is returned.

4. TERMINOLOGY

For a better understanding of the *case study*, find below a set of terms and their meaning in this lecture's context:

Customer	A user visiting the *webshop* with the intention to buy one or many *products*.
Product	A commodity offered for sale by the *tenant*'s *webshop*.
Software system	The result of the mission; to be designed and implemented.
Tenant	A company or an individual operating a *webshop*.
Webshop	The *tenant*'s *webshop* being attached to our *software*

15 *Software as a Service*, see http://en.wikipedia.org/wiki/Software_as_a_service (*Wikipedia*)

	system.
Team	The bunch of developers and executives being on the mission.

5. CONSTRAINTS

There are many possible ways getting a solution done for a an undertaking. A promising approach chosen depends on the overall circumstances and constraints. In this *case study*, the actual business model is not finally settled yet, therefore we have to cope with moving targets. Depending on how you and your company is positioned, you will be confronted with a given set of specific constrains which you might be able to influence within very narrow limits only. The constraints below are considered to have influenced some of my architectural decisions:

Mission	We are confronted with moving targets in the given context of *webshop* optimization. We are entering uncharted territory as of *big data* processing by a small company.
Market	As the market is not well known yet, we have no valid metrics on the volume of data to expect and the required *scaling* of the underlying *software systems*.
Organization	The company to build the *software system* for is a *startup* alike company with restricted budget and short time-to-market requirements.
Team	The development *team's* size varies between five to eight developers including a *nearshoring* team in *Novi Sad* (*Serbia*).
Skills	The *team* has excellent *Java* knowledge and architectural skills, though no *cloud computing* and *big data* experience
Methodology	The development methodology is an *agile* orientated one, inspired by *Scrum*[16], though no real *Scrum* master is on board.
Burden	The *software system* is to inherited a *REST*[17] *(Representational state transfer)* alike *HTTP*[18] (*Hypertext Transfer Protocol*) interface from a third-party *software system*.

16 *Scrum*, see http://en.wikipedia.org/wiki/Scrum_%28software_development%29 (*Wikipedia*)
17 *REST* (*representational state transfer*), see http://en.wikipedia.org/wiki/REST (*Wikipedia*)
18 *HTTP*, see http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol (*Wikipedia*)

6. Approach

As the context of the mission is more or less clear, though still keeping room for surprises, some systematic and still flexible *software architecture* is to be set up for the *software system*. As of the uncertainty regarding the data loads we will be confronted with, we need to be *scalable* throughout all aspects of our *software system*. As of time-to-market constraints, each target of the mission moving must not endanger the overall effort. The *software architecture* must be flexible enough to cope with short term changes without tearing down the whole *software system*.

A clever software architecture expects changes and provides room for them within its given domain without breaking the overall software system

As of restricted development resources, parts of the *software system* should do their job in various contexts.

... this led to the decision to set up a *software architecture* based on the *IPO Model*[19] (“*Input – Process – Output*”) using the *composite pattern*[20] in combination with the *interceptor pattern*[21] when embedded into a *front controller pattern*[22]. All of them together applied with the *interface based programming*[23] approach, organized using the *repository pattern*[24] ...

In case you got confused, those paradigms, amongst others, will be intensified in below sections.

To keep everything lean, we make extensive use of *Software as a Service* (*SaaS*)[15] *services* supporting the development process and *Platform as a Service* (*PaaS*)[25] *services* hosting and operating our *software system* in terms of

19 *IPO Model*, see http://en.wikipedia.org/wiki/IPO_Model (*Wikipedia*)
20 *Composite pattern*, see http://en.wikipedia.org/wiki/Composite_pattern (*Wikipedia*) and http://01853.cosmonode.de/index.php/Composite_Pattern
21 *Interceptor pattern*, see http://en.wikipedia.org/wiki/Interceptor_pattern (*Wikipedia*)
22 *Front Controller*, see http://en.wikipedia.org/wiki/Front_Controller_pattern (*Wikipedia*)
23 *Interface based programming*, see http://en.wikipedia.org/wiki/Interface-based_programming (*Wikipedia*) and http://01853.cosmonode.de/index.php/Interfacebasierte_Programmierung
24 *Repository pattern*, see http://01798.cosmonode.de/index.php/Repository_pattern
25 *Platform as a Service*, see http://en.wikipedia.org/wiki/Platform_as_a_service (*Wikipedia*)

cloud computing.

7. Technology stack

The technologies chosen to develop and build up the *software system* are quite straight forward:

Java[26]	A *software platform*[27] and *programming language*[28] coming with a wide variety of free and *open source*[29] development tools and *frameworks*[30]; providing us a big ecosystem.
Tomcat[31]	A *Java* based *open source web container*[32] managed by the *Apache Software Foundation*[33], hosting our *software system's REST* alike *interface* and attaching it to the web.
Spring[34]	A *framework* in the *Java* ecosystem providing us with *transaction processing*[35] and *dependency injection*[36] without having to carry around a heavy weight *application server*[37].
MySQL[38]	A *relational database management system*[39] (*RDBMS*) used for critical data requiring consistency such as *machine*, *service* and *tenant* management.
Maven[40]	Our *build automation*[41] tool with dependency and version management capabilities for *software components*[42]; *Maven* is managed by the *Apache Software Foundation*.

As of our lean approach, the technology stack includes external *cloud services* provided as *PaaS,* mainly *Amazon Web Services*:

26 Java, see http://en.wikipedia.org/wiki/Java_%28software_platform%29 (*Wikipedia*)
27 *Computing platform*, see http://en.wikipedia.org/wiki/Computing_platform (*Wikipedia*)
28 *Programming* language, see http://en.wikipedia.org/wiki/Programming_language (*Wikipedia*)
29 *Open source*, see http://en.wikipedia.org/wiki/Open_source (*Wikipedia*)
30 *Framework*, see http://en.wikipedia.org/wiki/Software_framework (*Wikipedia*)
31 *Apache Tomcat*, see http://en.wikipedia.org/wiki/Apache_Tomcat (*Wikipedia*)
32 *Web container*, see http://en.wikipedia.org/wiki/Web_container (*Wikipedia*)
33 *Apache Foundation*, see http://en.wikipedia.org/wiki/Apache_Foundation (*Wikipedia*)
34 *Spring Framework*, see http://en.wikipedia.org/wiki/Spring_Framework (*Wikipedia*)
35 *Transaction processing*, see http://en.wikipedia.org/wiki/Transaction_processing(*Wikipedia*)
36 *Dependency injection*, see http://en.wikipedia.org/wiki/Dependency_injection (*Wikipedia*)
37 *Application server*, see http://en.wikipedia.org/wiki/Application_server (*Wikipedia*)
38 *MySQL*, see http://en.wikipedia.org/wiki/MySQL (*Wikipedia*)
39 *RDBMS*, see http://en.wikipedia.org/wiki/Relational_DBMS (*Wikipedia*)
40 *Apache Maven*, see http://en.wikipedia.org/wiki/Apache_Maven (*Wikipedia*)
41 *Build automation*, see http://en.wikipedia.org/wiki/Build_automation (*Wikipedia*)
42 *Component-based software engineering*, see http://en.wikipedia.org/wiki/Component-based_software_engineering (*Wikipedia*)

S3	The *Simple Storage Service* (*S3*) *from Amazon's AWS provides* us with file storage for flat files.
SimpleDB	*An AWS NoSQL database* for feeding the vast amounts of traffic data into the persistence layer.
RDS	The *AWS Relational Database Service* (*RDS*) *provides* us our *MySQL instances*[84] for managing the *inventory*[43].
EC2	The *AWS Elastic Compute Cloud* (*EC2*) provides *virtual machines*[44], in our case running *Linux*[45] *operating system*[46].
Elastic Load Balancing	The *AWS load balancer*[47] is used for *load balancing* our incoming vast amounts of traffic data.

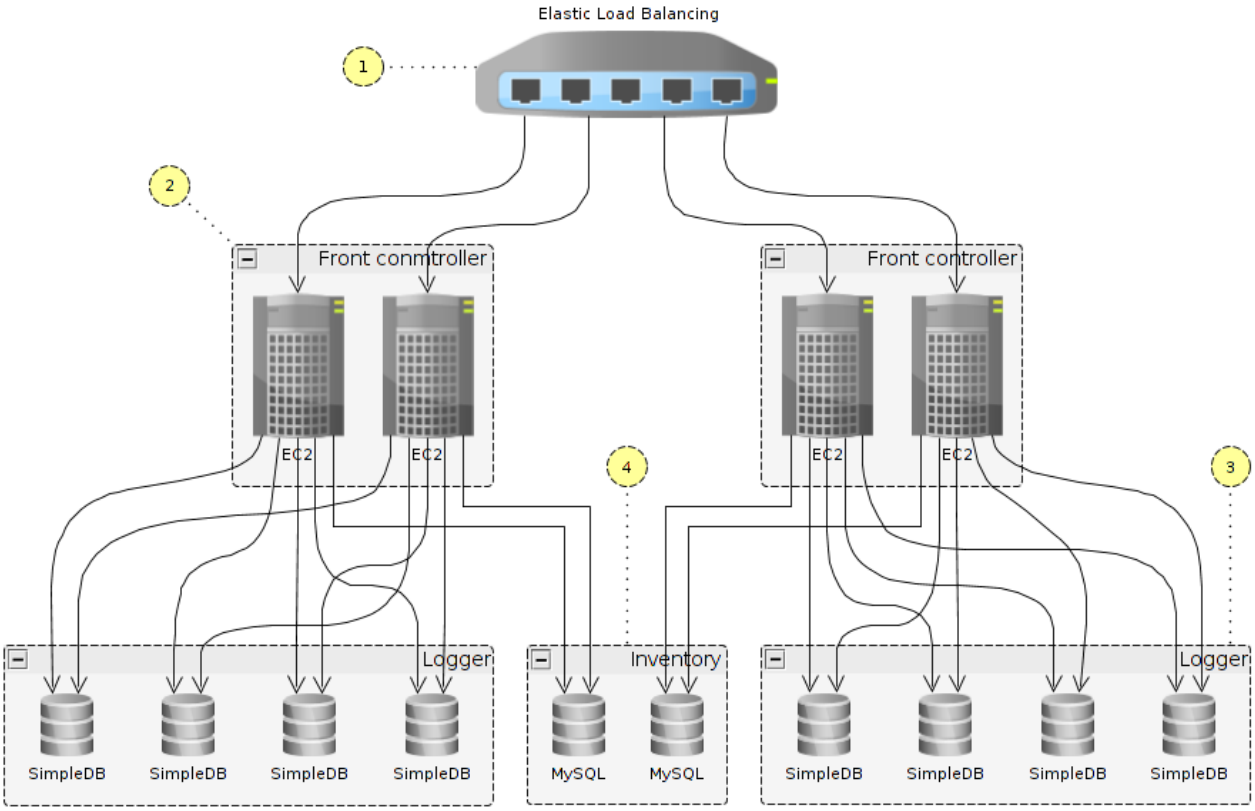

Figure 2: A high level overview of the system landscape

2 Illustrates the technology stack: The *software system* consists of a *load balancer*, *virtual machines* and *NoSQL databases* as well as *relational databases*:

43 *Inventory*, see chapter *11 Herding cats: Resource management* on page 27
44 *Virtual machine*, see http://en.wikipedia.org/wiki/Virtual_machine (*Wikipedia*)
45 *Linux*, see http://en.wikipedia.org/wiki/Linux (*Wikipedia*)
46 *Operating System* (*OS*), see http://en.wikipedia.org/wiki/Operating_system (*Wikipedia*)
47 *Load balancing*, see http://en.wikipedia.org/wiki/Load_Balancer (*Wikipedia*)

The *Elastic Load Balancer* attaches the *software system* to the *internet*[48] and receives the attached *webshops*' *HTTP* requests (*1*). The *EC2 virtual machines* run *Linux* with a *Tomcat* on top, which processes the *HTTP* requests, digests the data and feeds the *SimpleDB domains* (*2*). The *SimpleDB domains*[49] hold the digested data from the *webshops* for processing by subsequent steps (*3*). The *RDS*' *MySQL instances* provide the *inventory*'*s* management and configuration information – on the *machines* and *services* related to the *tenants* managed by the *software system* (4).

8. Development process

The development *team* is split up into two parts, an on-site *team* in *Munich* and a *nearshoring*[50] *team* in *Novi Sad*, we make a conference call every day. As of our lean approach, we decide to use a *cloud* based *SaaS* tool-set, namely *Assembla*[51]*,* providing us with task and code management facilities. Working with distributed *teams*, *SaaS* tools are most useful by eliminating the hassle of setting up a distributed development environment manually.

Excursus: Scrum

Regarding *Scrum, Wikipedia* says: "*... A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need ..., and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner ...*"[16].

Agile does not mean "chaotic" or "unplanned", it actually enforces a well defined process with a set of roles, rules and tools to be applied to successfully challenge moving targets in the *life cycle* of your project.

Our *development process*[52] is supported by a *SaaS* tool-set consisting of:

- A task management tool

48 *Internet*, see http://en.wikipedia.org/wiki/Internet (*Wikipedia*)
49 *SimpleDB domain*, see http://aws.amazon.com/simpledb/faqs (*Amazon SimpleDB FAQs*)
50 *Nearshoring*, see http://en.wikipedia.org/wiki/Nearshoring (*Wikipedia*)
51 *Assembla*, see https://www.assembla.com (*Assembla*)
52 *Software development process*, see http://en.wikipedia.org/wiki/Software_development_process (Wikipedia)

- *Git*[53] code repositories (a *version control system*[54])
- *Agile* project management including an *agile* planner
- Work-flows for code *life cycle*[55] management (such as *in progress*, *ready-for-test* or *tested* process lanes)
- User management and time tracking

The *development process* itself is backed by an *agile methodology* and *test-driven development*[56] (*TDD*): A *Scrum* alike *agile development process* for *team* organization and *gray-box testing*[57] for automated *unit testing*[58] using *JUnit*[59].

Excursus: Test-driven development

Gray-box testing is the *test driven development*[56] approach we have chosen. There are different understandings on *gray-box testing*[57], let's start off with *black-box testing*[60] and *white-box testing*[61]:

With *white-box testing*, the internal structure of the code to be tested is known, as if you were to write *unit tests* after you programmed the code. With *black-box testing*, just the *interface*[62] is known by the tester, tests are written before the code behind the *interface* is being programmed; the tester and the *programmer*[63] are different people.

With *gray-box testing*, as we understand it in this *case study*, you write the *unit tests* for an *interface* before you program the code (tester and *programmer* are the same individuals). We assume this to be an acceptable compromise between the *black-box testing* and the *white-box testing* paradigms in a *startup* alike context.

Using lean task and code management tools “from the *cloud*” keeps costs low in the beginning (similar to outsourcing the *data center* in terms of *SaaS* and

53 *Git*, see http://en.wikipedia.org/wiki/Git_%28software%29 (*Wikipedia*)
54 *Revision control*, see http://en.wikipedia.org/wiki/Revision_control (*Wikipedia*)
55 *Software release life cycle*, see http://en.wikipedia.org/wiki/Software_release_life_cycle (*Wikipedia*)
56 *TDD*, see http://en.wikipedia.org/wiki/Test_driven_development (*Wikipedia*)
57 *Gray-box testing*, see http://en.wikipedia.org/wiki/Gray_box_testing (*Wikipedia*)
58 *Unit testing*, see http://en.wikipedia.org/wiki/Unit_testing (*Wikipedia*)
59 *JUnit*, see http://en.wikipedia.org/wiki/JUnit (*Wikipedia*)
60 *Black-box testing*, see http://en.wikipedia.org/wiki/Black_box_testing (*Wikipedia*)
61 *White-box testing*, see http://en.wikipedia.org/wiki/White-box_testing (*Wikipedia*)
62 *Interface*, see http://en.wikipedia.org/wiki/Interface_%28object-oriented_programming%29 (Wikipedia)
63 *Programmer*, see http://en.wikipedia.org/wiki/Programmer (*Wikipedia*)

PaaS services).

9. Into the cloud: Scalability

As of the mission, many requests and data chunks are to be received via *HTTP* within *acceptable response times*[64] by the *software system*. Selected operations of a *customer* (who is identified e.g. by a *cookie*[65]) navigating an attached *webshop* each produces a request to our *software system*, answered by the *software system* with an according individual response.

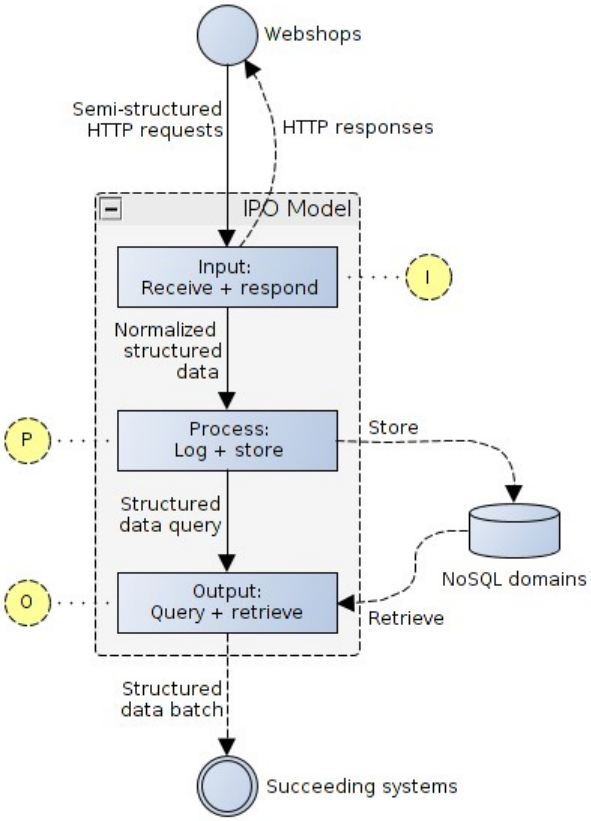

Figure 3: The IPO Model in general, being applied to the software system

A request may identify a *product* being placed in a *customer*'s shopping basket or a *customer* viewing *product* details. All of those operations of a single user are called *customer journey*. An individual response by the *software system* means producing feedback to the *customer*, based on the individual

64 *Acceptable Response Times*, see http://www.webperformancematters.com/journal/2007/7/10/acceptable-response-times.html (*Web Performance Matters*)
65 *HTTP cookie*, see http://en.wikipedia.org/wiki/HTTP_cookie (*Wikipedia*)

customer's journey.

Receiving a single request and producing an individual response is to take place within ¼th of a second[66]. The *software system* actually receives several thousand requests per second. This in turn means that the data chunks are to be digested by the *software system* accordingly quick.

Simplifying matters, the basic principle processing the incoming data is represented by the *IPO Model* (as of *Input – Process – Output*):

1. *Input*: Receive the requests (and process output)
2. *Process*: Log and store the structured data
3. *Output*: Query and retrieve logs (for processing by succeeding systems)

In the real world things are more complicated, we actually apply the *IPO Model* to the *Input* step as well (in this context, replacing "*Input*" with "*IPO*", we could call it "*IPOPO Model*"), explaining why we process *output* in the *Input* step.

Excursus: IPO Model

Actually, the *IPO Model*[19] is one of the most fundamental recurring *fractals*[85] in *information technology*[67] and *computer science*[68]. Each *method*[69] receives *input* and *processes output*. The *IPO Model* denotes nothing more than that operations can be split into three steps: *Input*, *Process* and *Output*. In *data warehousing*[70], the *IPO Model* is called *ETL*[71] (for *extract*, *transform* and *load*).

In their atomic structure, most *software systems* are build up of *IPO Model* artefacts – such as *methods*, *Java Servlets or pipes and filters*[72] – some of them reflect the *IPO Model* also in their overall *software architecture*.

As of the three *Input – Process – Output* steps, the *software system* is separated into three sub-systems (*3*), each of which being modularized. The sub-systems each are to *scale* and provide flexibility as well as room for

66 Some marketing guy knows someone from an adverser company who says so ...
67 *Information technology*, see http://en.wikipedia.org/wiki/Information_technology (Wikipedia)
68 *Computer science*, see http://en.wikipedia.org/wiki/Computer_science (*Wikipedia*)
69 *Method*, see http://en.wikipedia.org/wiki/Method_%28computer_science%29 (*Wikipedia*)
70 *Data warehouse*, see http://en.wikipedia.org/wiki/Data_warehouse (*Wikipedia*)
71 *ETL*, see http://en.wikipedia.org/wiki/Extract,_transform,_load (Wikipedia)
72 *Pipeline*, see http://en.wikipedia.org/wiki/Pipeline_%28software%29 (Wikipedia); *Pipe and filter pattern*, see http://01798.cosmonode.de/index.php/Pipe_and_filter_pattern

extensions.

Without an own *data center*, we operate the *software system* using *web services*[73] from the *AWS cloud* – all of which is backed by the *software architecture*.

1. Input - Receive requests (and process output)

As said before, the overall *Input* step itself is divided into the three *IPO Model's sub*-steps *Input*, *Process* and *Output*. The overall *Input* step is to process loads of incoming *HTTP* requests, do pre-processing in its *Processing* sub-step and produce *HTTP* responses in its *Output* sub-step. The overall *Input* step then passes the now structured data elements to the overall *Process* step. Let us take a closer look at the overall *Input* step:

1. ***Input**: Receive + respond*
    1. *(Sub-)* ***Input***: Receive loads of *HTTP* requests
    2. *(Sub-)* ***Process***: Pre-process the *HTTP* requests
    3. *(Sub-)* ***Output***: Produce *HTTP* responses
2. *Process*: Log + store
3. *Output*: Query + retrieve

Technically speaking, the *HTTP* requests and the *HTTP* responses are handled by a *front controller* (represented by a *Java Servlet*[74], a basic *web container*) dissected into many *interceptors*[21] assembled by means of the *composite pattern*[20].

As seen in *4,* we have a *front controller* constructed of multiple nested *interceptors: S*tep *1* (“*begin*”) is the connection receiving *HTTP* requests and step *5* (“*end*”) is the connection passing back *HTTP* responses – in between refining the *HTTP* request, done in stages *2*, *3* and *4*. The outgoing connection “*Process*“ is directed to incoming connection “*Input*” of the next overall

73 *Web service*, see http://en.wikipedia.org/wiki/Web_service (*Wikipedia*)
74 *Java Servlet*, see http://en.wikipedia.org/wiki/Java_Servlet (*Wikipedia*)

Process step (*5*).

The data being passed from the overall *Input* step to the overall *Process* step is normalized in terms of heterogeneous input data being unified and consolidated and assigned a unique *tenant's* (*webshop's*) identifier.

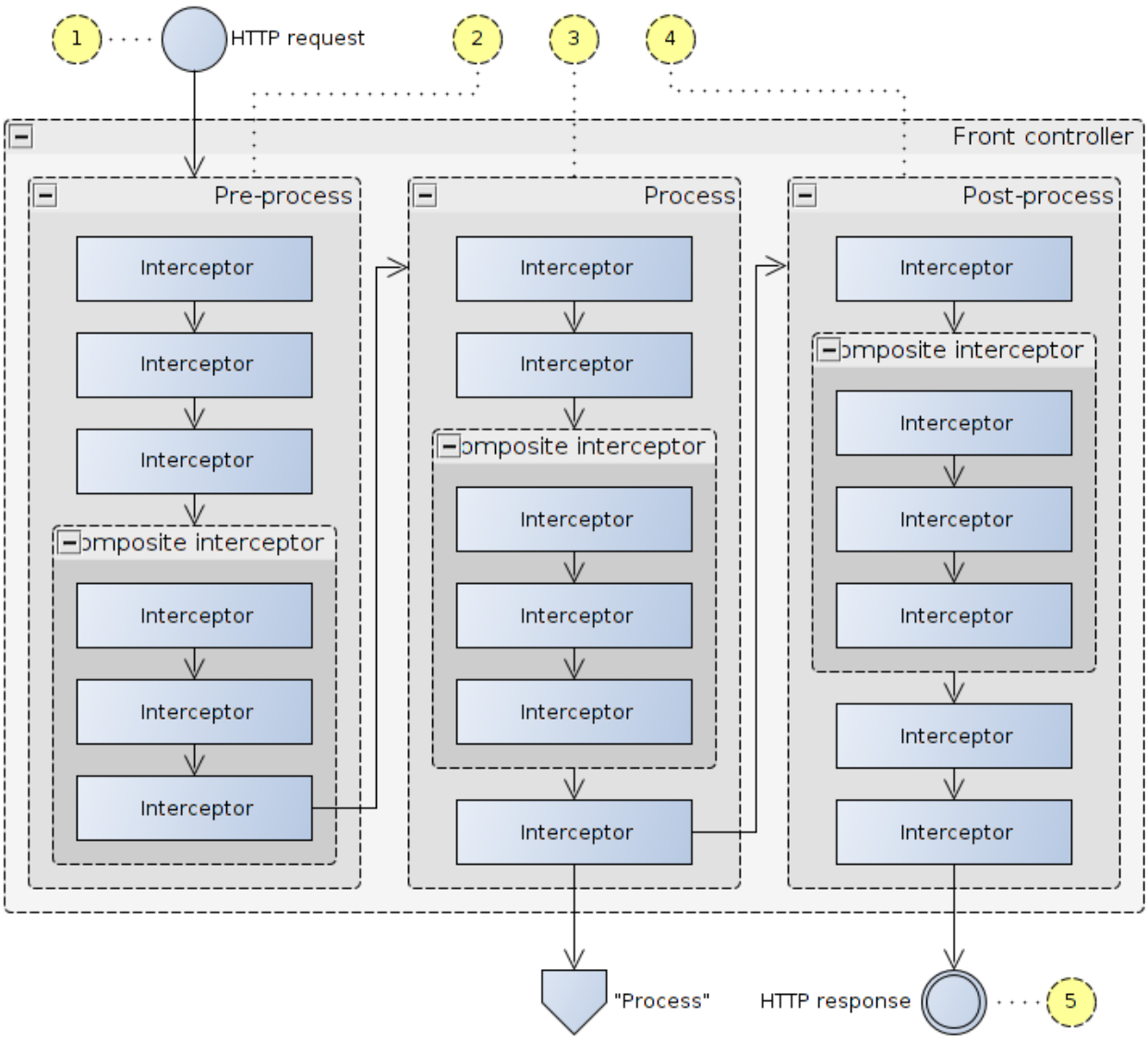

Figure 4: The Input step of the IPO Model in detail

In this *case study*, the *front controller* is nothing more than a *composite interceptor*. A *composite interceptor* is composed of *interceptor components* which may be *atomic* (a real *interceptor* with an own implementation, for example parsing the *HTTP* request and producing normalized data or *logging*

statistical information on the incoming load) or nested *composite interceptors*.

Alternate implementations exist for the *composite interceptor*, varying in their behaviour, mainly in the decision, which *interceptors* are to be invoked in any case and which *interceptors* are to be skipped in case of errors (such as a malformed request).

Excursus: Front controller pattern

As of *Wikipedia*, *"... Front controllers are often used in web applications to implement workflows. While not strictly required, it is much easier to control navigation across a set of related pages ... from a front controller than it is to make the individual pages responsible for navigation..."*[22]

The *front controller pattern* in this *case study* consists of exactly one *software component* being the entry point of our *web application*. All *HTTP* requests entering the *web application* are picked up by the *front controller*. The *front controller* then dispatches the *HTTP* request for further processing or enriches it with *aspects*[75] – similar to *AOP* (a*spect-oriented programming*).

In this *case study*, the *font controller* makes use of the *interceptor pattern* to add *aspects* to the *HTTP* requests as well as to dispatch processing of the *HTTP* requests accomplished by the according *HTTP* responses.

The *front controller* is assembled of the three *composite interceptors*

2. *pre-process* (*non-functional*[76] *aspects*)
3. *process (functional*[77] *aspects)*
4. *post-process* (*non-functional aspects*)

The *composite interceptor* representing the *front controller* is always to process the *pre-process interceptors* and the *post-process interceptors*, the *process* interceptors' execution is to be aborted in case of errors (no succeeding *process interceptors* are invoked).

Generally speaking, the *pre-process* and the *post-process interceptors* handle a request's *non-functional aspects* whereas the *process interceptors* address *functional* operations.

75 *AOP*, see http://en.wikipedia.org/wiki/Aspect-oriented_programming (*Wikipedia*)
76 *Non functional*, see http://en.wikipedia.org/wiki/Non-functional_requirement (*Wikipedia*)
77 *Functional*, see http://en.wikipedia.org/wiki/Functional_requirement (*Wikipedia*)

As seen above, the *front controller* actually combines the *front controller pattern* with the *interceptor pattern* and the *composite pattern*.

Excursus: Interceptor pattern

The *interceptor pattern*[21] is used when an operation's sequential execution may differ depending on the operation's incoming *messages*[78]. The *interceptor pattern* chains *software components* in a sequence – the *interceptors* – one after the other. An incoming *message* is passed to the first *interceptor* in the chain, in case it feels responsible for the *message*, it *executes* and terminates the chain, else the *message* is passed to the next *interceptor*; till an *interceptor* feels *responsible* or the *message* has passed the last *interceptor*.

This makes it particular suitable inside a *front controller* pattern. Here the messages are the *HTTP* requests. Depending on the kind of request being received, an *interceptor* takes over, does its processing and contributes to the *HTTP* response. In this special *case study*, we have different kinds of *interceptors*: The *composite interceptors* and the *atomic interceptors*.

An *atomic interceptor* is a plain interceptor doing its work. A *composite interceptor* contains other *interceptors*, being *composite* or *atomic interceptors*. A *composite interceptor* may delegate to its *contained interceptors* just one after the other or it may apply rules such as the first *interceptor* and the last *interceptor* being pre-process and post-process steps always to be executed. Also type conversion could take place inside a compound interceptor so that the therein contained *interceptors* process on transformed data types (as of *generic programming*[79]). A *composite interceptor* could also always invoke each therein contained *interceptor* – no matter whether an *interceptor* feels responsible or not – in this way establishing an assembly line.

In case you want to *scale,* make sure that your *interceptors* are stateless[80], it makes things easier – you might need *sticky sessions*[81] else and your *software system* in turn may tend to become a *monolithic system*[82] (bad).

2. Process – Log and store

The normalized outcome of the *Input* step's “*Process*” connection (*4*) is passed to the overall *Process step's “Input” connection* (*5*). The normalized data is a result of the refinement of a *customer's journey* on a tenant's *webshop*. Let us

78 *Message*, see http://en.wikipedia.org/wiki/Message (*Wikipedia*)
79 *Generic programming*, see http://en.wikipedia.org/wiki/Generic_programming (*Wikipedia*)
80 *Stateless protocol*, see http://en.wikipedia.org/wiki/Stateless_protocol (*Wikipedia*)
81 *Sticky sessions*, also known as *session affinity*, used in the context of *load balancing*[47]
82 *Monolithic system*, see http://en.wikipedia.org/wiki/Monolithic_system (*Wikipedia*)

take a closer look at the overall *Process* step:

1. *Input: Receive + respond*
2. ***Process***: Log + store
    1. *Partitioned logging*
    2. *Composite logging*
    3. *SimpleDB logging*
3. *Output*: Query + retrieve

As easily can be seen, the overall *Process* step of the *case study* addresses the *logger system* (*5*). Similar to the *Input* step's *front controller*, it is designed with the *composite pattern* in mind:

Excursus: Composite pattern

As of *Wikipedia, "... The composite pattern describes that a group of objects*[83] *is to be treated in the same way as a single instance*[84] *of an object ..."*[20].

Simply speaking, you define an *interface* for a *software component* for which you implement one ore more *atomic components* ("real" implementations of the business logic) as well as at least one *composite component*. The *composite component* contains *objects* implementing that *interface* and delegates *method* calls to the according *methods* of the therein contained *objects*; either your *atomic components* or other *composite components*. That way you can create arbitrary nested such structures.

As it is irrelevant from your business logic's point of view whether it "talks" to a *composite object* or its *atomic* counterpart – as they share the same *interface* – the *composite pattern* is especially interesting regarding *scalability*. This way you can parallelize operations defined in your *interface* – your *software system* stays slim, testable and maintainable*. Furthermore, depending on the load you are experiencing, you can increase or decrease the number of instances being encapsulated in your composite component.*

The overall *logger system* is represented by *parted loggers,* themselves being composed of *composite loggers,* which themselves delegate to *atomic loggers* (a real *logger* with an own implementation towards *SimpleDB domains*).

83 *Object*, see http://en.wikipedia.org/wiki/Object_%28computer_science%29 (*Wikipedia*)
84 *Instance*, see http://en.wikipedia.org/wiki/Instance_%28computer_science%29 (*Wikipedia*)

Looking closer at the *parted loggers* and the *composite loggers*, we can identify the *composite pattern* – the *fractal*[85] of this *software system*. Here again, we have a nested *composite* structure, possible configurations may be any combination of *parted loggers, composite loggers* and *atomic loggers*.

As many *webshops* are managed by the *software system*, many requests from many *customers* of many *tenants* are to be handled efficiently.

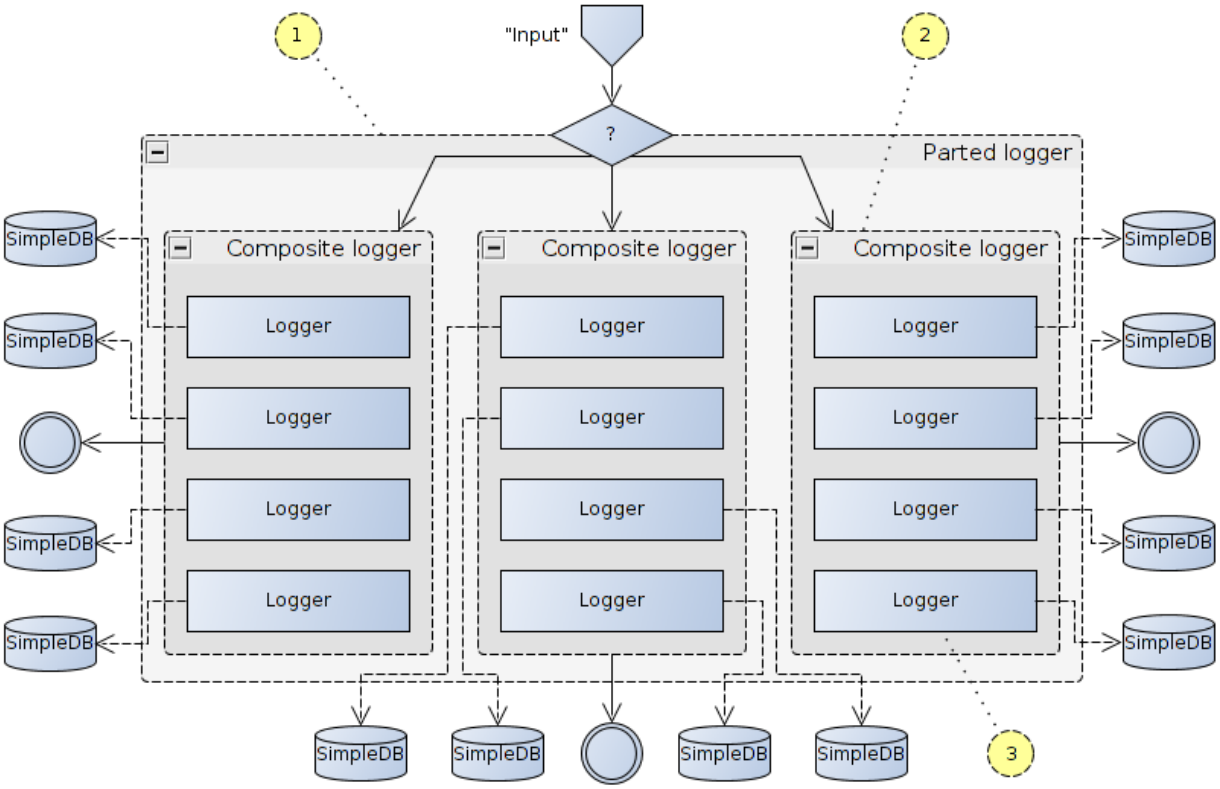

Figure 5: The Process step of the IPO Model in detail

Dissecting the *logger system* as of *5*, we notice a *parted logger (1)* being the top *logger* acting as the *logger's* entry point. At first, the normalized data from the overall *Input* step is inspected by the *parted* logger for a *partitioning*[86] criteria. In our case, the *partitioning* criteria is represented by a (unique) *tenant*'s identifier provided by the normalized data from the *front controller*. Depending on the *tenant*'s identifier (e.g. from which *tenant*'s *webshop* did a *customer* trigger a request), the normalized data from the *Input* connection is

85 *Fractal*, see http://en.wikipedia.org/wiki/Fractal (*Wikipedia*)
86 *Partition (database)*, see http://en.wikipedia.org/wiki/Partition_%28database%29 (*Wikipedia*)

delegated by the *parted logger* to one of its underlying *composite loggers (2)*. Each *composite logger* receives all the normalized data for exactly one *tenant*'s requests. To *scale* up the load of an individual *tenant*, the c*omposite logger* for a *tenant* delegates the normalized data loads round robin – by *asynchronous method dispatch* (*AMD*)[87] aka *asynchronous I/O*[88] – to one of its nested *atomic loggers (3)*, each of which attached to its own *Simple DB domain*. This way the the number of *tenants* cannot affect an individual *composite logger's* throughput.

Excursus: Partitioning and composition strategies

The *parted logger* actually applies the principle of *database shards*[89], each *tenant's composite logger* is one dedicated *shard*, the *tenant's* identifier being the criteria for which *shard* to address. The *composite loggers* actually apply plain *horizontal partitioning*[86], a *shard* in this use case is actually *horizontally partitioned*.

Determining the most efficient *partitioning* criteria and therewith structure of *parted loggers, composite logger*s and the number of *atomic loggers* down the data sinks is a science in itself. You have to know various factors on your business domain, such as the expected load, the composition of your data, the locations and according volumes of your data's origin, costs such as transfer times, response times and transport fees regarding the location ... just to mention some of them, in order to identify promising *partitioning* criteria candidates or the number of *atomic loggers* per partition. Things get more complicated when you choose a different nested *loggers* structure. One may start by observation and adjustments regarding the *logger* set-up.

Using the *composite pattern* enables us to *scale* up or down regarding on the actual load we are experience. Customers may expect lots of traffic during Christmas time and decreasing load in summer time: The *composite logger* can take this into account and attach or detach *atomic loggers* as required.

To relieve the top *parted logger* when the number attached *tenants* increases, several *parted loggers* may be put in parallel, each of which receiving its *HTTP* requests from a *load balancer* or from dedicated (*parted logger* specific) *URL*s[90] (*uniform resource locator*).

87 *AMD*, see http://en.wikipedia.org/wiki/Asynchronous_method_dispatch (*Wikipedia*)
88 *Asynchronous I/O*, see http://en.wikipedia.org/wiki/Asynchronous_I/O (*Wikipedia*)
89 *Shard*, see http://en.wikipedia.org/wiki/Shard_%28database_architecture%29 (*Wikipedia*)
90 *URL*, see http://en.wikipedia.org/wiki/Uniform_resource_locator (*Wikipedia*)

The actual partitioning and composition structure chosen for this *case study* is straight forward: A *tenant's* identifier offers itself for charging *tenants* individually depending on their individual resource consumption. Therefore the top *logger* is a *parted logger* having a *tenant*'s identifier as partitioning criteria. Using *composite loggers* per *tenant* underneath rises the throughput per *tenant*: We assume that the expressions querying a *tenant's* data are heterogeneous so that all *atomic loggers* for a *tenant* are to be queried. Therefore partitioning in this layer would make things complicated without a benefit. As of the moving targets, the *partitioning* criteria in this layer would also be volatile soon.

Our first throw regarding the *loggers* structure is actually a strike, enabling to log 5.000 to 7.000 *HTTP* requests per second and *tenant*[91].

3. Output – Query and retrieve

The *Process* step of the *IPO Model* results in structured data stored by various *SimpleDB instances* in the *cloud*. Succeeding processing or analysis steps, such as feeding a *data warehouse*[92] or further processing and refinement, query and retrieve the data from the *SimpleDB instances* via the *logger system*. The querying mechanism is to be quite flexible, automatic or human analysts are to fire ever changing queries into the *logger system* (as of the moving targets, the targeted result-set is not yet settled upon).

Let us take a closer look at the overall *Output* step:

1. *Input: Receive + respond*
2. *Process*: Log + store
3. ***Output***: Query + retrieve

The overall *Output* step is realized by a tool-set being reflected by a *console application*[93]; embeddable in various succeeding automatic or manual processes. As of the requirements to provide flexible querying mechanisms as

91 *Me*: "*... my system administrator told me that that's about the throughput...*"
92 *Data warehouse*, see http://en.wikipedia.org/wiki/Data_warehouse (*Wikipedia*)
93 *Console application*, see http://en.wikipedia.org/wiki/Console_application (*Wikipedia*)

well as support of not yet thought of requirements, the following decisions are made:

a) Provide a simple and dynamic *sentential logic*[94] based querying "language" interpreted (translated) by our tool-set.

b) Provide a modular design of the tool-set's *console application;* as requirements may change during the development phase.

Regarding the *sentential logic* based querying "language", on purpose we do not use *SimpleDB*'*s* query language: We never know in which way *SimpleDB* will evolve, whether we will stay with *SimpleDB* or move on to another technology. Using *SimpleDB*'*s* query language whilst evolving to another technology would break our *software system*.

Regarding the modular design of the *console application*, a modification of the *command pattern*[95] is being applied (*6*): An additional *context*[96] is passed to the *execute* methods of the individual *commands.*

The *context*, constructed by the *console application*, includes the *command line arguments* as well as access to the *software system*'s functionality, being the *logger system* and the *inventory*.

The *console application* is divided into a set of *commands*. As long as a *command* obeys defined criteria, it can be added to the *console application* (for example expecting the *context* in its *execute* method). Each *command* is invoked with the *context* by the *console application*, one by one. Depending whether the *command* detects its responsibility for the given *command line arguments* (passed in the *context*), it executes and signals to exit the *console application* afterwards. If a *command* is not responsible for the given *command line arguments*, control is passed to the succeeding *command*.

As of *6*, *command line arguments* are passed to the *console application (1)*. The *console application* constructs a *context* including the *command line*

94 *Propositional calculus*, see http://en.wikipedia.org/wiki/Propositional_calculus (*Wikipedia*)
95 *Command pattern*, see http://en.wikipedia.org/wiki/Command_pattern (*Wikipedia*)
96 *Context*, see http://en.wikipedia.org/wiki/Context_%28computing%29 (*Wikipedia*)

arguments. The *context* is passed to the first *command (2)* which evaluates the *command line arguments: i*n case the *command* is responsible, it executes by applying operations on the passed *context* (e.g. on the *logger system* or the *inventory*) – finally execution of the *console application* terminates *(3)*. In case the *command* is not responsible for the passed *command line arguments*, control is passed to the next *command (4)*. This chain of responsibility is continued till a *command* takes over responsibility or till there are no more *commands* left in the chain (*10*).

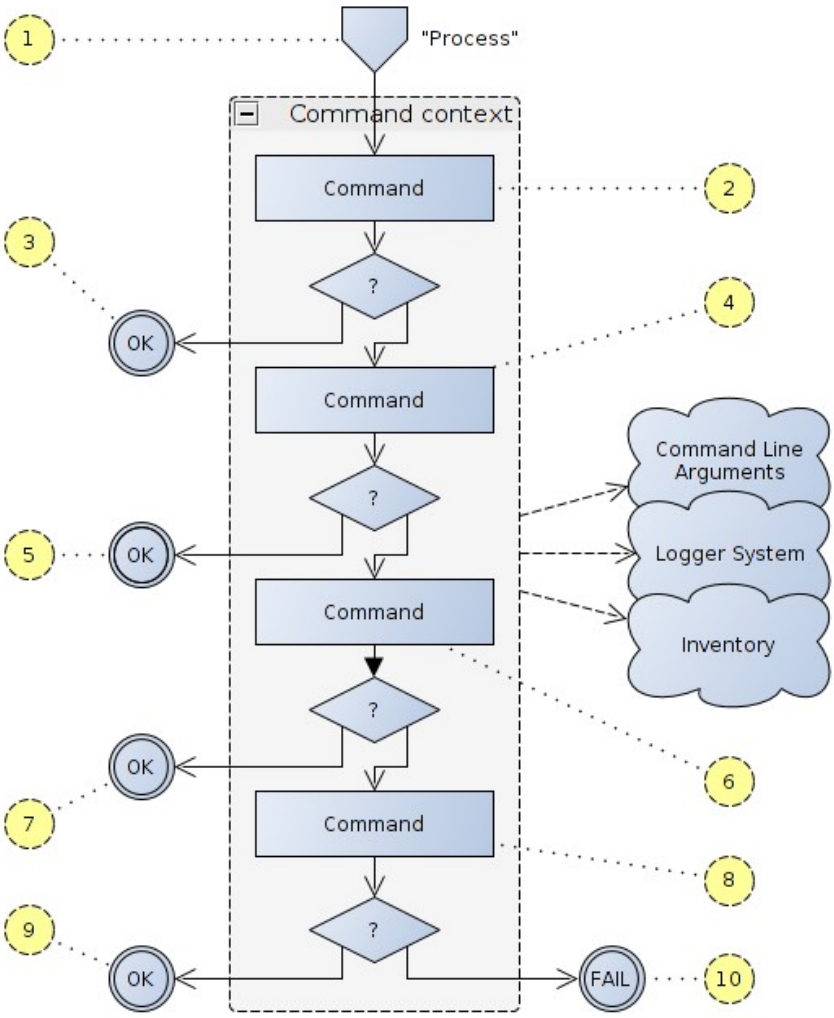

Figure 6: The Output step of the IPO Model in detail

As of *separation of concerns*[97], the *console application* takes care of *command line argument* parsing, *context* creation, help functionality or error handling

97 *Separation of concerns*, see http://en.wikipedia.org/wiki/Separation_of_concerns (*Wikipedia*)

while the *commands* implement the actual business functionality. This way we can react to moving targets without breaking things.

Excursus: Command pattern

As of *Wikipedia*, "*... the command pattern is a ... pattern in which an object is used to represent and encapsulate all the information needed to call a method at a later time. This information includes the ... method parameters...*"[95]. A *command* can be seen as a *method* and its *context* (*variables*[98]) all transformed to an *object. Having such an object, a command* can be executed at any time in the future, it can be placed on a *stack*[99] or it can be transferred throughout *software system* boundaries.

In this *case study*, we used the *command pattern* merely to keep the *console application* maintainable and expandable as the *console application* is set up of a set of dedicated *commands*. The *commands* share the same *interface* and are provided with a *context by the console application* – consisting of the *logger system* and the *inventory*. To keep the *boilerplate*[100] small when adding new functionality to the *console application*, the *console application* itself is the *boilerplate*. *It* parses the *command line arguments*[101] from the *command-line interface* (*CLI*), constructs the *context*, instantiates the *commands* and invokes the *command* as specified by the *command line arguments*.

Commands offer you funny possibilities: Given you define a *command's interface* providing an "*execute*" and an "*undo*" *method* and your *software system* strictly makes use of *commands*, then you easily can provide *undo*[102] functionality by putting your executed *commands* onto a *stack*, where they are just waiting for having their "*undo*" *method* called in reverse order.

10. ... AND OUT OF THE CLOUD: SECURITY

Privacy protection law in *Germany* is quite strict, privacy protection law in other countries is different, in some case it is less strict. Our *customers* originate from *Germany*, they require *German* privacy protection law to be applied. Privacy protection applied on *services* having their head office outside of *Germany* cannot be granted to be compatible with the *German* law. When using *cloud services* from outside *Germany*, we have to make sure that we can grant privacy protection as of *German* law.

98 *Variable*, see http://en.wikipedia.org/wiki/Variable_%28computer_science%29 (*Wikipedia*)
99 *Stack*, see http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29 (*Wikipedia*)
100 *Boilerplate*, see http://en.wikipedia.org/wiki/Boilerplate_code (*Wikipedia*)
101 *CLI*, see http://en.wikipedia.org/wiki/Command-line_interface (*Wikipedia*)
102 *Undo*, see http://en.wikipedia.org/wiki/Undo (*Wikipedia*)

We identify the following constraints: We have to store *big data* in the untrusted *cloud* in a “secure” though fast manner. We also have to retrieve *big data* in a “secure” though fast manner from the untrusted *cloud*. Decryption parts and encryption parts are to be strictly separated from each other to enable operation of physically separated encryption (untrusted *cloud*) and decryption (“secure” *data center*) systems.

We choose a p*ublic-key (PK) cryptography*[103] approach (*asymmetric encryption*) combined with a *symmetric-key algorithm*[104] (*symmetric cryptography*) to store data in the *cloud* and decrypt it in a “secure” *data center* out of the untrusted *cloud*. A mechanism somehow similar to the *forward secrecy*[105].

As of performance issues, the *asymmetric keys* generated by the “secure” *data center* are only used to negotiate *symmetric-key algorithm's ciphers* with the *cloud*, the *ciphers* are used to encrypt and decrypt the actual data (which is much faster than an *asymmetric encryption* approach is being capable of):

Encrypting parts are to generate their *ciphers* (for encryption and decryption) by themselves, in-memory, volatile and exclusively for their own (in-memory) use, encryption must not persist *ciphers*. Decryption parts are to use the (decrypted) *ciphers* only in-memory, volatile and exclusively for their own (in-memory) use. The *ciphers* are transferred from the *cloud* to the “secure” *data center* – being encrypted using the public part of the *asymmetric keys* (provided by the “secure” *data center*).

The *cloud* only knows of *public keys* and generates *ciphers* encrypted with the *public keys* from the “secure” *data centers*. The “secure” *data center* has access to the *private keys* for decrypting the *ciphers* which are used to decrypt data from the *cloud*. The *ciphers* generated in the *cloud* change continuously, the *ciphers* are volatile and only locally visible in the encryption and the decryption *context*.

To protect from disclosed *encryption ciphers*, the *encryption ciphers* change

103 *PK cryptography*, see http://en.wikipedia.org/wiki/Public-key_cryptography (*Wikipedia*)
104 *Symmetric encryption*, see http://en.wikipedia.org/wiki/Symmetric_encryption (*Wikipedia*)
105 *Forward secrecy*, see http://en.wikipedia.org/wiki/Forward_secrecy (*Wikipedia*)

continuously: In case of disclosed *ciphers*, only the portion of the data actually encrypted with the disclosed *cipher* is affected, minimizing the extend of affected data.

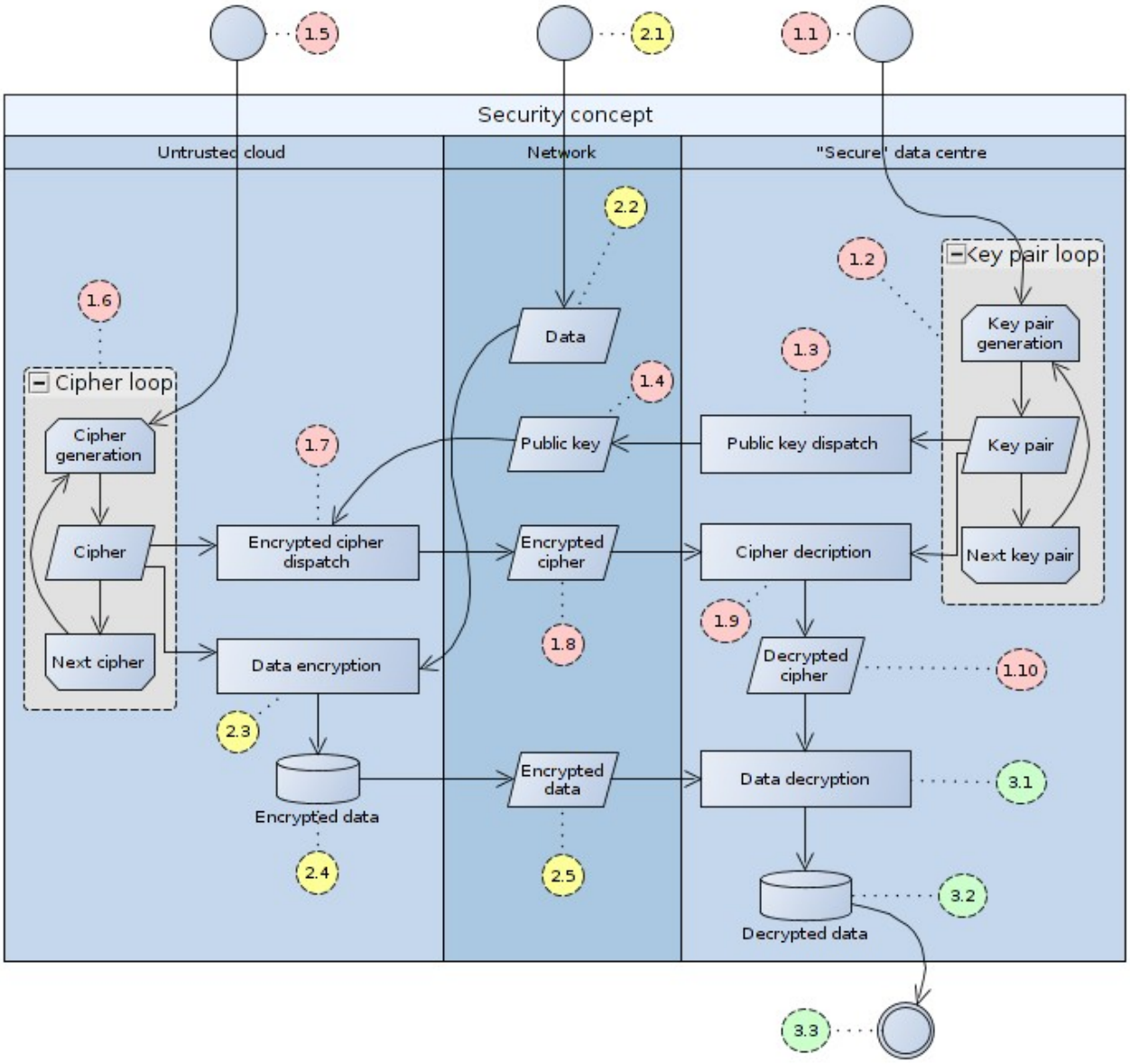

Figure 7: The security concept's cipher and data exchange in detail

7 scribbles the security architecture: Steps *1.1* through *1.10* illustrate the *public key* and *cipher* exchange. Steps *2.1* through *2.5* illustrate the data encryption process. Step *2.4* represents the *logger system* (as of *5*) and steps *3.1* through *3.3* illustrate data decryption. Note steps *1.4*, *1.8* and *2.5* which denote the only data possibly disclosed as of being exchanged between the the

participating *software systems*; being the *cloud* and the *data center*.

Decryption and storage of critical data must only take place in the *data centers* outside the *cloud*, preferably hosted in the country whose privacy protection law is to be applied and by a company having its head office in that country.

Excursus: Forward secrecy

"... In cryptography, forward secrecy ... is a property of key-agreement protocols ensuring that a session key derived from a set of long-term keys cannot be compromised if one of the long-term keys is compromised in the future. The key used to protect transmission of data must not be used to derive any additional keys, and if the key used to protect transmission of data is derived from some other keying material, then that material must not be used to derive any more keys. In this way, compromise of a single key permits access only to data protected by that single key...".[103] (Wikipedia)

Having said this, the key exchange used for this *case study* aims at the same goal: Disclosure of a single *cipher* does not compromise all data, it just compromises the data being encrypted with that given *cipher*. By changing *ciphers* in high frequency as well as changing *key pairs* in less high frequency (as it is quite "expensive" in terms of computing time), we assume that disclosure of a single *cipher* is strictly limited to the data being encrypted with the compromised *cipher*.

This concept has not been challenged seriously, consider it as "work in progress". "Security" is an ever evolving topic, you can never consider to be finished with handling "security" issues. Usually each concept can be challenged or enhanced. For example, critical data can by completely physically separated from harmless data, critical (even encrypted) data never finding its way into the cloud. Automation of public key handshake is another issue looking for enhancements.

11. Herding cats: Resource management

In the sections above, we discussed topics such as *scalability* and "security". As we operate the *software system* in the *cloud*, we are to manage three related entities to keep the *software system* up and running:

a) *Tenants*, operating the attached *webshops*.

b) *Machines*, being the hardware running the *services* for the *tenants.*

c) *Services*, being the software executed for the *tenants on* the *machines.*

One changing entity affects the other two entities. In case the number of *tenants* increases, additional *machines* might be required for hosting the *services*, all of which assigned to according *tenants*. In case the number of *tenants* decreases, *machines* might get consolidated and *services* got to be shut down to save costs. Depending on t*enants*' individual requirements, they might have varying kinds of *services* to be operated with individual configuration parameters. This kind of *herding cats* requires a management tool for all participating sub-systems to know ”*what*” (*services*) they “*where*” (*machines*) are to run for ”*whom*” (*tenants*).

This leads to the decision to design an *inventory* using the *repository pattern*[106] being applied. The *inventory* makes use of a classic *relational database management system*. In this case we use a *MySQL*[107] *database* – as of required *referential integrity*[108].

Excursus: Repository pattern

“... The repository regarding the repository pattern denotes a central memory component accessed by different clients. It is used to store common (shared) data to the clients and may act as an asynchronous communication means between the clients ...“[106].

In this *case study*, the *inventory* represents the *repository*, the *clients*[109] are represented by the *services* which share the *inventory's* data. The *services* actually share *their view* of the *inventory's* data. The *inventory* relates *tenants*, *machines* and *services* to each other: From these relationships a *service's* dedicated view can be retrieved from the *inventory* in terms of “configuration data of a *service* for a *tenant* on a *machine*”.

We are to keep track of the *tenants*, the *services* subscribed by the *tenants* and the allocation of the *tenants*' *services* to the the *machines*. Along the way, we store configuration information on various combinations of *tenants, services*

106 *Repository pattern*, see http://01798.cosmonode.de/index.php/Repository_pattern
107 *MySQL*, see http://en.wikipedia.org/wiki/MySQL (*Wikipedia*)
108 *Referential integrity*, see http://en.wikipedia.org/wiki/Referential_integrity (*Wikipedia*)
109 *Client*, see http://en.wikipedia.org/wiki/Client_%28computing%29 (*Wikipedia*)

and *machines*. E.g. the configuration of a *service* for *different* tenants differs or the configuration of a *service* depends on the *machine* it is deployed on.

The *inventory* actually stores the information on which *front controller* is responsible for which *tenants*. It also stores the information on which *composite logger* is claimed by that *tenant* including the *SimpleDB domains* being assigned to that *composite logger*.

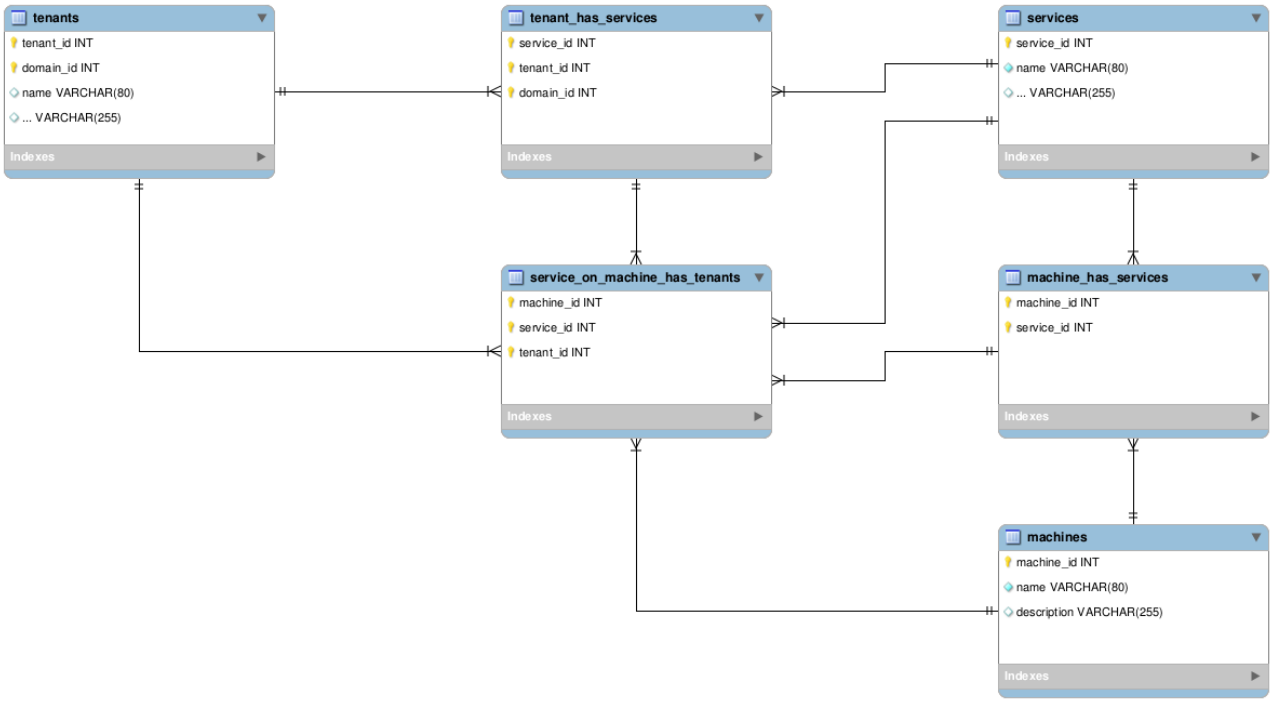

Figure 8: A simplified entity–relationship model of the inventory

See *8* for a simplified e*ntity–relationship model*[110] (*ER model*) of the *inventory*: The crow's feet of the relations denote a “one to many” relationship whereas the opposite end of the relation denotes an “exactly one” relationship.

12. SOFTWARE DESIGN AND IMPLEMENTATION

To finish off, some words are to be said on *software design,* implementation and *scalability;* affecting all of the above mentioned *software systems*. I will intensify on *asynchronous* calls, *interface based programming*, *big data*

110 *ER model*, see http://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model (*Wikipedia*)

processing, application assembly or *pattern* and *refactoring*[111].

1. Asynchronous calls

Implementing *asynchronous methods* is especially useful to grant quick response times; not blocking the caller's *thread*[112]. Usually them are applicable when your *method* being called has to return immediately and just has to trigger an operation while not caring for the operation's result – as it is not passed back to the caller.

To avoid blocking your *thread* when triggering that operation, you somehow have to decouple your *thread* from the actual operation's execution, so that your *thread* quickly can pass back control to the caller. In the context of (*web*) *application servers*, the *HTTP* request / response cycle should be very quick. *Node.js*[113] for example propagates the asynchronous paradigm in its server design. *Java* as well provides neat means meeting asynchronous requirements.

For the *logger system* to respond quickly when being invoked, the log process is taking log data and returns immediately while writing it asynchronously to the data sinks. A *Java Servlet*'s request *thread* passed to the *front controller* is not blocked while request data is being persisted by the *logger system*. The *Java Concurrency Utilities*[114] provide us with *blocking queues*[115] used for asynchronous *logging* and *executor services*[116] for managing *thread* creation in an *application server* conforming way.

2. Interface based programming

I regard *interfaces* between *components* a vital part in *software engineering*[117], I prefer a good *interface* to a good implementation. Why? A good *interface* hides bad code from depending *components.* The impact of bad code to be

111 *Code refactoring*, see http://en.wikipedia.org/wiki/Code_refactoring (*Wikipedia*)
112 *Thread*, see http://en.wikipedia.org/wiki/Thread_%28computing%29 (*Wikipedia*)
113 *Node.js*, see http://en.wikipedia.org/wiki/Node.js (*Wikipedia*)
114 *Java concurrency Utilities*, see http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency (*Oracle*)
115 *Interface BlockingQueue*, see http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html (*Oracle*)
116 *Interface ExecutorService*, see http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html (Oracle)
117 *Software engineering*, see http://en.wikipedia.org/wiki/Software_engineering (*Wikipedia*)

refactored behind a good *interface* is locally isolated whereas changing a bad *interface* affects depending *components* globally as well as their implementations. Therefore attention is to be paid regarding *interfaces* and keep in mind that *interfaces* are provided and used. Each interface implies *interface* partners with whom to negotiate a good *interface*. *Interface based programming* takes account of this.

A side effect is the decoupling of your depending *components* from specific implementations, making it easy to replace a specific *NoSQL database* with another technology.

Excursus: Interface based programming

"Modular Programming defines the application as a collection of intercoupled modules ... Interface Based Programming adds more to modular Programming in that it insists that Interfaces are to be added to these modules. The entire system is thus viewed as Components and the interfaces that helps them to coact.".[23] (*Wikipedia*)

Actually most *design patterns* require you to use *interfaces* in order to apply them *patterns*. The role an *interface* has in a *software system* depends on the point of view; it depends whether your *component* defines the *interfaces* for attached *components* by itself or whether it uses the *interfaces* of provided *components* (yes, that makes the difference). *Interfaces* are engineered differently when a *layered architecture*[118] (use given *interfaces*) is being applied than when a *ports and adapters*[119] architecture (*define your own interfaces*) is being applied.

As of moving targets in the *use case's* scope, i*nterface based programming* helps us to cope with changes and changing technologies without breaking neither the *components* nor the *software system.* Impacts are limited locally to dedicated *components* to be recoded or replaced.

3. Big data processing

Why did we not use a *MapReduce*[120] framework such as *Apache's Hadoop*[121]? *MapReduce* implies to put your algorithm where your data is. Frameworks such

118 *Mulitlayerd architecture*, see http://en.wikipedia.org/wiki/Multilayered_architecture (*Wikipedia*)
119 *Hexagonal architecture* , see http://alistair.cockburn.us/Hexagonal+architecture (Alistair Cockburn)
120 *MapReduce*, see http://en.wikipedia.org/wiki/MapReduce (*Wikipedia*)
121 *Apache Hadoop*, see http://en.wikipedia.org/wiki/Apache_Hadoop (*Wikipedia*)

as *Hadoop* expect the *big data* to be processed to reside in big flat files. In our *case study*, small chunks of data drop into the *front controller*: Here we do pre-processing and normalization of the *HTTP* requests. As we already have small chunks of structured data we do the processing where the data is dropped (that is also an aim of *MapReduce*). The data as we get it from the *front controller* is exactly what we need in our *logger system* and the succeeding steps, there is no need to generate a big flat file which then is being processed in chunks (we do processing in many *front controllers* beforehand or in parallel succeeding steps). Succeeding steps working with the *logger system* can be highly parallelized, having many tasks doing the succeeding processing. Probably the *logger system* is the conceptual counterpart of the *Hadoop distributed file system* (*HDFS*).

Excursus: Big data housekeeping

Big data means that you have to do *housekeeping*[122] for your collected data. Storing data costs money, querying data costs time. To work efficiently with your *big data* you must have a clear understanding of the *life cycle* of your data and a strategy to cope with it: You may refine and consolidate it through several steps of a processing *pipeline*, ending up in the end with consolidated data to keep; so being able to throw away the raw data of the beginning of your refinement *pipeline*. Not having a strategy will make you end up with problems; as refinement afterwards has to keep up with incoming data...

4. Application assembly

As of strictly applying *interface based programming*, we are free to change the implementation of parts of our *software system*. We might consider *SimpleDB* not fitting new requirements regarding *quality of service*[123] applied to *database management systems.* We might want to switch to *Amazon's DynamoDB* (“*... a service based on throughput, rather than storage ...*”)[124].

We use S*pring's dependency injection* mechanism throughout all parts of the *IPO Model* to glue together the *software system*. As of *interface based programming* we easily can provide an *atomic logger* using *DynamoDB* instead

122 *Housekeeping*, see http://en.wikipedia.org/wiki/Housekeeping_%28computing%29
123 *Quality of service*, see http://en.wikipedia.org/wiki/Quality_of_service (*Wikipedia*)
124 *DynamoDB*, see http://en.wikipedia.org/wiki/DynamoDB (*Wikipedia*)

of *SimpleDB*. To actually assemble the *software system* to use *DynamoDB*, we just have to reconfigure one spot in our *Spring* configuration.

5. Pattern and refactoring

As soon as a part of the *software system* begins to mutate in a bad way, having *god class*[125] delusions of grandeur or monster *method* ambitions, it's time to do some *refactoring*. There are quite a few *refactoring* techniques out there[126]. Please keep in mind when *refactoring* on how you best prevent the code smell in the future. Just resolving the symptoms won't fix the root cause:

When breaking down a *god class* or a monster *method*, think of a *design pattern* which will prevent that kind of mutation in the future. The *interceptor pattern* breaking down the *front controller* into maintainable bits and pieces helps us to effectively prevent mutation of the *front controller* code.

13. Conclusion

In the beginning of the session some questions were asked; let's take a look if the paper gave some answers:

- "*How to scale up and scale down your cloud resource consumption?*"

We used the *composite pattern* to do so, having stateless *atomic components* under the hood of our *composite components* makes it easy to increase or decrease the throughput by increasing or deceasing the the number of *atomic components;* provided that the *atomic components* do not share a *bottleneck*[127] and provided that our *composite components* can operate in parallel as well.

- "*Which technology stack works promisingly well? "*

We use *cloud services* and *NoSQL databases* for getting throughput. *Java* provides us with a rich ecosystem full of possibilities. *Relational databases* are used for data requiring *referential integrity* without the requirement of high throughput. This technology stack works very well for us. This does not mean

125 *God object*, see http://en.wikipedia.org/wiki/God_object (*Wikipedia*)
126 *Refactoring Auswahl*, see http://01853.cosmonode.de/index.php/Refactoring-Auswahl
127 *Bottleneck*, see http://en.wikipedia.org/wiki/Bottleneck (*Wikipedia*)

that other technologies wouldn't do the job.

- “*Which design patterns and software architectures are suitable? ”*

For us, using *the composite pa*ttern in combination with *asynchronous* operations works out to get the throughput we need. Think *stateless* to be able to easily parallelize your *components* and prevent *monolithic systems*. Make sure that in your architecture there is not one *bottleneck* which you cannot not *scale* up. In this *case study*, just using *composite loggers* would not have done the job; the *parted logger* utilizes the *scalability* we need.

- “*How to populate and organize your systems in this dynamic environment without loosing track?*”

We use the *inventory* with its *referential integrity* based on the *repository pattern* to keep track on the interdependencies between *machines*, *services* and *tenants*. The *inventory* actually reflects the *software systems*' state in the real world.

- “*How to retain a clear view of your services*' *and systems*' *health condition?*”

We have luck, our *logger system* is perfectly well suited to be used as a system logger, similar to *syslog*, though as of its distributed though centralized manner, we can use it for automatic system health and system load monitoring of distributed *cloud* services ...

14. Outlook

Currently I am working on a project evaluating promising architectural trends. An interesting concept is that of *microservices*[128]. The *inventory* is a candidate to be bundled as a *microservice*.

Amazon's Elastic Beanstalk[129] *service* is another interesting candidate to get rid of the *Elastic Cloud Computing instances* and deploy the *front controllers*

128 *Microservices*, see http://microservices.io/patterns/microservices.html (*Microservice architecture*)
129 *AWS Elastic Beanstalk*, see http://en.wikipedia.org/wiki/AWS_Elastic_Beanstalk (*Wikipedia*)

directly into the *Elastic Beanstalk service*.

Outlook: Cloud API (CAPI)

First there was *SaaS* (*Software as a Service*), then there came *PaaS* (*Platform as a Service*). *SaaS* represents full blown *web applications* whereas *PaaS* represents low level *web services* – *a*ll of which can be accessed via the *internet.* The former is operated by users, the latter is invoked by *software systems*. Between *SaaS* and *PaaS* there seems to be a gap. The gap should be filled by something more specific (less generic) than *PaaS* and less specific (more generic) than *SaaS ...*

It looks to me that there is a demand on *cloud services* providing ready to use *APIs*[130] (*Application Programming Interfaces*) as *REST services*. This could be called something like "C*loud API*" (*CAPI*) and fill that gap.

... CAPI would be a cloud hosted[131] *application domain*[132] *centric API ...*

CAPI is more specific (less generic) than a *PaaS file storage*[133] *service* (such as *Amazon S3*[134]) or a *PaaS database service*. *CAPS* is less specific (more generic) than a *SaaS webshop* or a *SaaS social networking service*[135] (such as *Facebook*[136]). *CAPI* would be part of a *software system* and not be the *software system* itself, whereas *SaaS* is the *software system*. *CAPI* would provide *business logic*[137] whereas *PaaS* is merely invoked by *business logic*.

An example for a *CAPI* could be some kind of *directory service*[138]: Users and groups, rights and roles, workspace and invitation management as well as notification and push functionality could be provided by a *CAPI's REST interface*. *Programmers* can concentrate on the development of their *mobile apps*[139] running on *smartphones*[140] by simply subscribing such a *CAPI* and connect their *mobile app* with that *CAPI* via *REST*. Such a *mobile app* could cover *working groups*[141] or *multiplayer*[142] business cases without the hassle for the *programmer* to take care of the *back end*[143] *systems* managing the *mobile app*'s community. Another *CAPI* candidate seems to be the *inventory*.

A *CAPI* would be fully set up by the *internet service provider*[144] (*ISP*) and be ready to use by any *programmer*. The technology stack required to get the *API* up and running would be hidden by the *CAPI* and managed by the *ISP.* The *ISP* would take care of *hosting*, *backup*[145] and *data recovery*[146].

In order to monitor all participating *software systems* in the *cloud* environment, a plan could be using the *logger system* not only to protocol a *customer's journey*. Furthermore a separate *instance* of the *logger system* may act as a

130 *API*, see http://en.wikipedia.org/wiki/Application_programming_interface (*Wikipedia*)
131 *Hosting service*, see http://en.wikipedia.org/wiki/Internet_hosting_service (*Wikipedia*)

system logger, similar to *syslog*[147], providing centralized means to inspect the system logs with cool querying features (similar but different to *Splunk*[148]).

15. EPILOGUE

Probably the main driver of getting a great challenge done is the lack of respect for that challenge. Creativity and a sound background of knowledge does no harm neither. Try to get the best compromise you can get under your given circumstances and constraints (unless you are working for one of the big and innovative IT companies around providing you with the best team, the best equipment, the best support and enough time and money). Trying to get the 100% best solution will most probably make you fail. A solution being 100% the best from one point of view often has shortcomings seen from another point of view. Getting those competitive goals into one solution is quite impossible (as the word “competitive” already suggests). In the past, some good work has only been finished because of time constraints – *Black Sabbath's*[149] debut album has been recorded within twelve hours[150] and is said to be one of the milestones in *Heavy Metal*[151]. Having infinite time you might search for the 100% solution and will never find it ...

---

132 *Domain*, see http://en.wikipedia.org/wiki/Domain_%28software_engineering%29 (*Wikipedia*)
133 *File system*, see http://en.wikipedia.org/wiki/File_system (*Wikipedia*)
134 *Amazon S3*, see http://en.wikipedia.org/wiki/Amazon_S3 (*Wikipedia*)
135 *Social networking service*, see http://en.wikipedia.org/wiki/Social_networking_service (*Wikipedia*)
136 *Facebook*, see http://en.wikipedia.org/wiki/Facebook (*Wikipedia*)
137 *Business logic*, see http://en.wikipedia.org/wiki/Business_logic (*Wikipedia*)
138 *Directory service*, see http://en.wikipedia.org/wiki/Directory_service (*Wikipedia*)
139 *Mobile app*, see http://en.wikipedia.org/wiki/Mobile_app (*Wikipedia*)
140 *Smartphone*, see http://en.wikipedia.org/wiki/Smartphone (*Wikipedia*)
141 *Working group*, see http://en.wikipedia.org/wiki/Working_group (*Wikipedia*)
142 *Multiplayer*, see http://en.wikipedia.org/wiki/Multiplayer_video_game (*Wikipedia*)
143 *Front and back ends*, see http://en.wikipedia.org/wiki/Front_and_back_ends (*Wikipedia*)
144 *ISP*, see http://en.wikipedia.org/wiki/Internet_service_provider (*Wikipedia*)
145 *Backup*, see http://en.wikipedia.org/wiki/Backup (*Wikipedia*)
146 *Data recovery*, see http://en.wikipedia.org/wiki/Data_recovery (*Wikipedia*)
147 *Syslog*, see http://en.wikipedia.org/wiki/Syslog (*Wikipedia*)
148 *Splunk*, see http://en.wikipedia.org/wiki/Splunk (*Wikipedia*)
149 *Black Sabbath*, see http://en.wikipedia.org/wiki/Black_sabbath (*Wikipedia*)
150 As of *MTV Masters Ozzy Osbourne / Black Sabbath, see http://de.wikipedia.org/wiki/Black_Sabbath#cite_ref-9* (*Wikipedia*)
151 *Heavy metal music*, see http://en.wikipedia.org/wiki/Heavy_metal_music (*Wikipedia*)

About the author

My name is *Siegfried Steiner*; born in *Hannover* (*Germany*) I spent some years as a teenager in *Zimbabwe*. I studied computer science in *Munich* where I now live. Currently I am working for an online bank as senior manager focussing on *software architecture* in an *agile team*. In the past I have been responsible for the product development dedicated to *big data* processing and *cloud computing*. Previous engagements also addressed issues such as *mobile computing*[152], *scalability* or *peer-to-peer*[153]. Areas of interest are to get some *open source* ideas up and running. For about 15 years I have been working as a computer scientist now and so was able to build up some know-how ...

152 *Mobile computing*, see http://en.wikipedia.org/wiki/Mobile_computing (*Wikipedia*)
153 *Peer-to-peer*, see http://en.wikipedia.org/wiki/Peer-to-peer (*Wikipedia*)

List of figures