System architecture manual
–in practice–

by Slobodan Erakovi Ć

System architecture manual -in practice-

Book type: Ebook
Writing and editing: Slobodan Eraković

2018, October, Serbia

about author

Husband and father of two kids at first place. Software enthusiast, playing around with different technologies and languages for almost two decades. Certified by Sun Microsystems and Oracle, because I am not interesting in formalities and titles, but in concrete things within education.
Problem solving, and getting things done, at first place, is my paradigm I am led with, instead of being slave of formality.
I like to do knowledge transfer to juniors, and whoever is interesting in, usually applying any kind of lecturing adjustment, which would help them easily to comprehend topic of interest.
I like hiking, basketball and billiard, when I have available moments.

Table of contents

about this book

Audience of this book is anybody interesting in software system architecture, in action, starting from junior up to senior developers, project managers as well. I am going to cover with real use case architecture of the system, and its parts. I am intending to give you an explanation of each architectural layer, his purpose and components.
Also, an explanation of this concrete solution will be provided, with a words of potentially available technologies to do the same work, or different strategies which might be used as an alternative to proposed solution.
This is book is not written in usual Manning or O'Reilly publication form of book content, but more as an intuitive approach describing usual steps one would take when start building the system architecture.

acknowledgement

All links within a manual, are pointing not to most descriptive and best resource, but just as a hint, which might be a starting point for further technology/concept investigation and learning.
Usually, links provided here, will also be most generics, and directed to sites like officials technology/language link, wikipedia, or others.
All technologies mentioned are simply one choice between multiple technologies which suits the same needs, but here I selected them based on some preference of mine, or because it fit best in particular situation.

preface

“Big Picture” of software system, is necessary when it comes up to creating enterprises. Just as any architectural creation, “bird point of view”, is actually a way of making you sure that everything is on its place, nothing is missed out and give you even a reminder, which one should consult during the design and implementation phase, occasionally, just to make sure that everybody are on the same page, in implementation process.
System architect job is not only monitoring sequential phases of software implementation, its progress. Indeed, he or she is in charge of that, but not solely.
Very often system/solution architect is the same role, where you must pay attention on requirements business value, precisely, does it fits implementation cost. It happens as well that, implementation cost is more than expected business will produce value, mostly because of non-technical product managers/owners nature. Simply, it can be easily over sighted that for the informational pop-up, with some data, requires much more implementation effort than it looks like at first glance.

Business requirements are the key influencer of building a specific architecture, which is suitable for that particular need, with very emphasized key point of business value. You do not want to create monster of enterprise, where requirements are simple web application presentation, with mere of business logic. Also, quick-n-dirty, system architecture is

something which should be never practiced, because simply, it is failure at the beginning.
However, there is no “golden bullet” for enterprise architecture solution, because the system architecture has a dynamic nature, created and based on business logic. Business logic changes has impact on the architecture creation and modification, (but not vice verse).
Your specific business requirements are hardly to find completely and sufficient system architecture which suits your needs, without modification, but on the other hand, you are never gonna create and adjust your business requirement, based on the “fast-food” architectural solution, you have in your hand.
Hence, the level of system, architect is handling with is high, usually not going into lower levels of design and implementation details.
One of the best definitions of “*what does architect do*”, and difference between an architect and senior developer, is expressed as:

“ *The designer (senior developer) is concerned with what happens when a user presses a button, and the* ***architect*** *is concerned with what happens when* ***ten thousand*** *users press a button*”[1].

In other words, architect take care and emphasize non-functional service level requirements (so called: **QoS - Quality of Service**), based on long experience and vast knowledge obtained through the career with different languages, technologies, and situations it was into.
On the other hand, designer/senior developers takes care of functional part of the application, and concrete implementation of business requirements, by coding concrete behavior of that.
In order to make such an architecture of the system, which support all service level requirements (**performance, scalability, reliability, availability, extensibility, maintainability, manageability, and security**) http://www.informit.com/articles/article.aspx?p=29030&seqNum=5 , business requirement understanding and translation is the first line of defense.

---

1 Sun Certified Enterprise Architect for J2EE™ Technology Study Guide”, Mark Cade Simon Roberts, Publisher: Prentice Hall PTR, First Edition March 11, 2002, ISBN: 0-13-044916-4, page 13

Even lack of less seginificant information, can have huge impact on the system to support all those *-ilities, so architect must be sure that all eligible information are included in the business requirement, prior to their translation and creation of system architecture, which will simple be a business reflection in mirror.
Hence, idea of this book is not to cover every single aspect of the enterprise architecture, but to provide an insight, how potentially architecture might look like, what is the purpose of particular architecture parts, just as an elements which those parts might consists of, and covering different parts of real world architecture starting from scratch.
Also, the intention of writing this book was a fact that I could not found, similar publication, which describe the entire concrete process of building enterprise application, from the very beginning (developing) and covering all further processes which are happening during the enterprise software production. I just wanted to paint a big picture, as clear as possible, and to bring it closer to other people with fewer or no knowledge/experience at all.

We will talk about concrete things which are included during the enterprise system implementation, and not only to talk about abstraction regarding architecture. Words like Eclipse IDE, Git, VPN, SSH are the one I am gonna use in order to present a concrete solution. The lecturing is gonna be divided by topics, concerning different aspects and layers of the system. Every each of them I am going to explain in general, as particular implementation of single solution I choose, explaining why is that chosen and why not some another one, which could also be eligible solution for that requirements.
I will address to everynon-functional requirements, at those places where one of them is affected, like, when we talk about certain protocols, like https/ssh, I am gonna explain, why those protocols are applied, which non-functional requirement is covered by that, etc/etc.

There are several architectural layers which are going to be comprehended within this lecture.

- ***Development/Deployment***
- ***Presentation Layer***

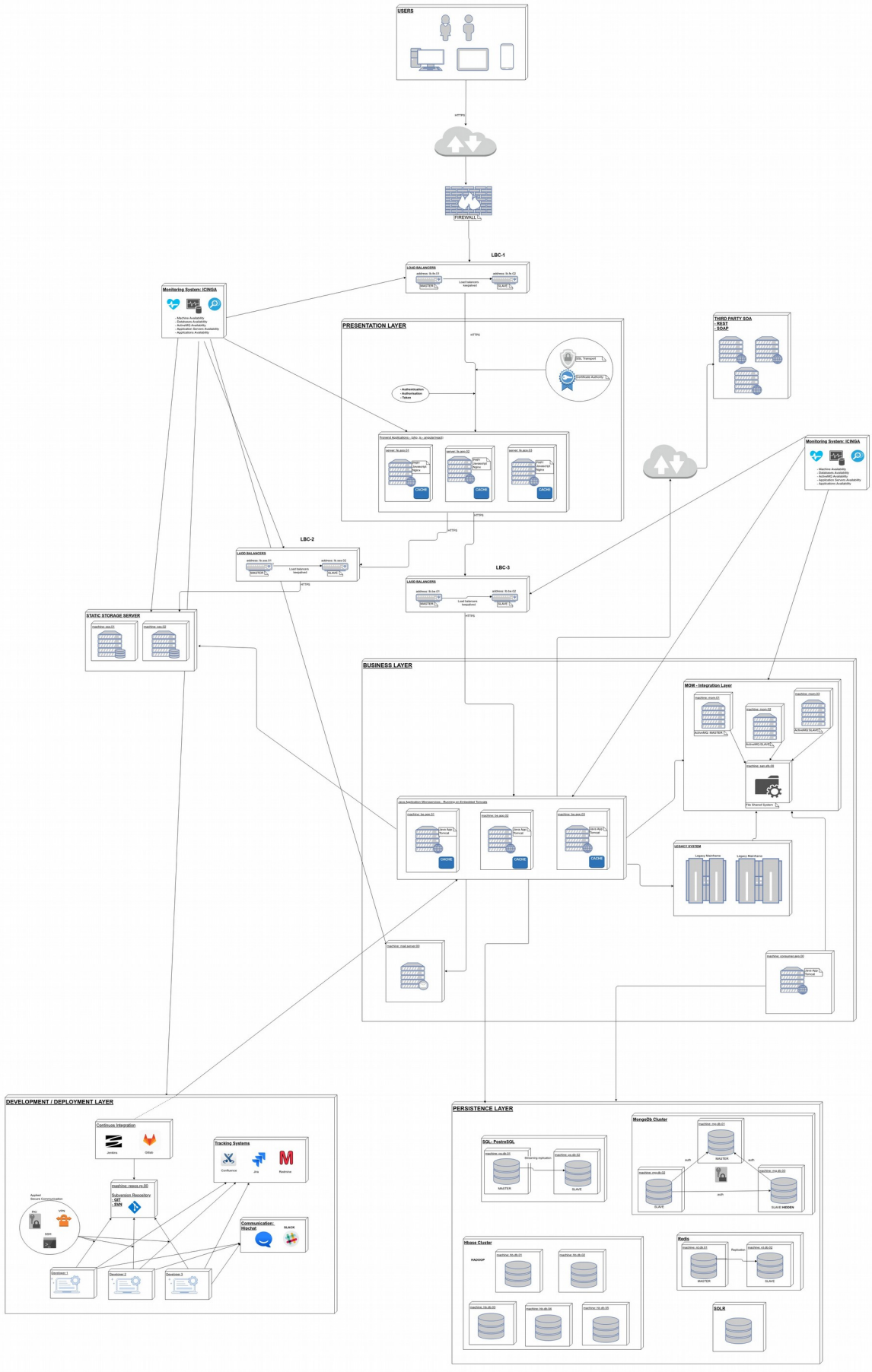
- ***Firewalls & Load balancers***
- ***Business Layer (also as this layer will be considered as a superset of integration layer)***
- ***Persistence Layer***

I am going to break down each of these layers, and dissect them with explanation of:

- ✔ ***which technologies are available to be used within particular layer***
- ✔ ***which set of technology is suitable for which business requirements***
- ✔ ***choosing one and concrete appliance of the chosen technology, to address particular requirement,***
- ✔ ***list of potential issues and mitigation strategy***

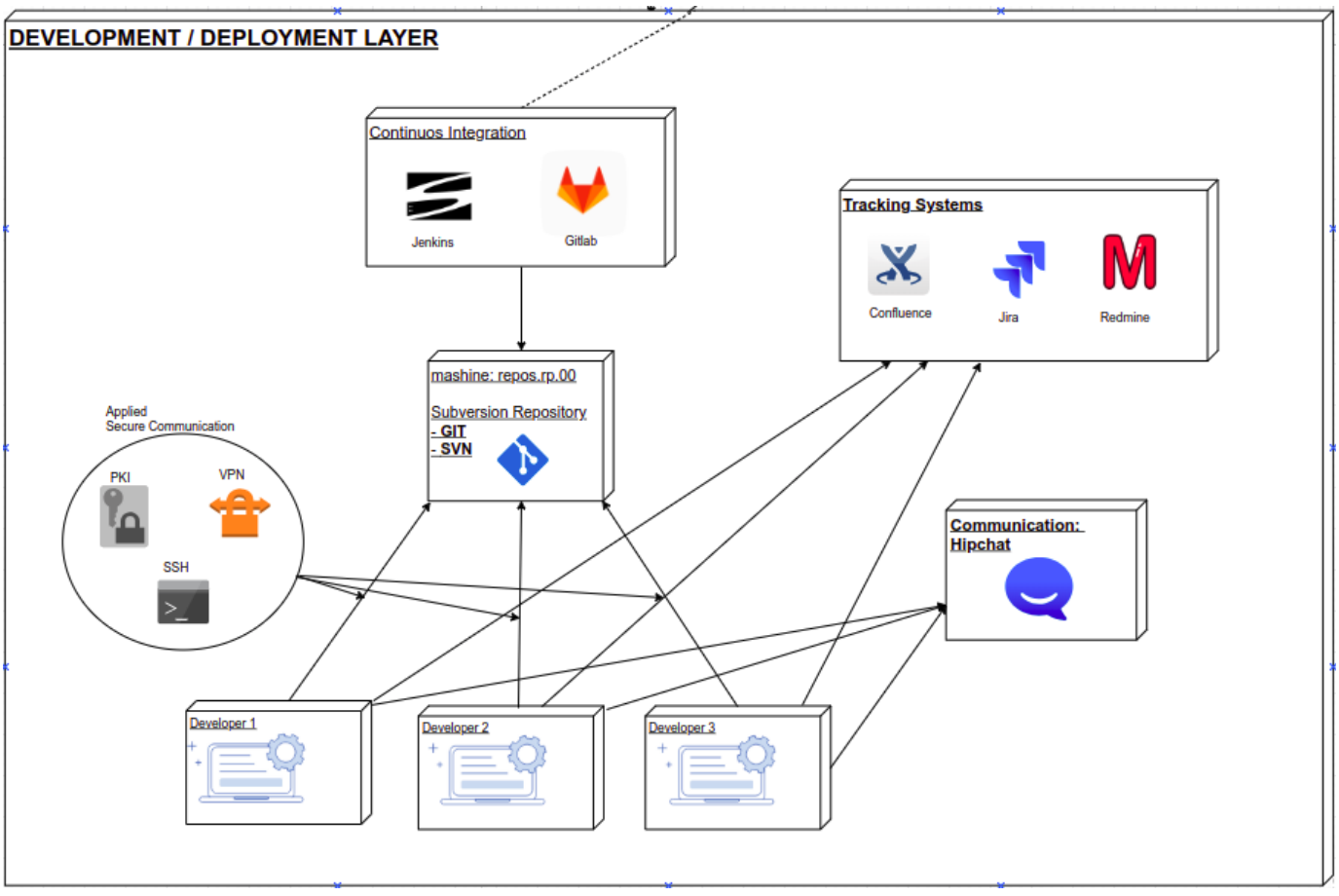
Also, I will explain how certain non-functional requirements are supported for particular use case, outside of layered architectural parts, like high availability, reliability, etc.
Please, keep in mind that, ideas was ***not*** to cover every single technology, potential another architectural solution, but to give you an insight how it works in general when it comes to building system architecture from scratch.
And here it comes – **A BIG PICTURE**

ARCHITECTURE



Development/Deployment

for me, it sounds perfectly logical to start with this layer, since, after all, this is the starting point of any software creation (at least its development part). Development and deployment processes are quite tightly coupled (maybe the only place of software development where these two words together are welcome) processes. Ensuring correct and smooth work of these processes, facilitate you accomplishment of higher quality level of software production, which later has a positive impact for service level requirements realization. Let check a diagram

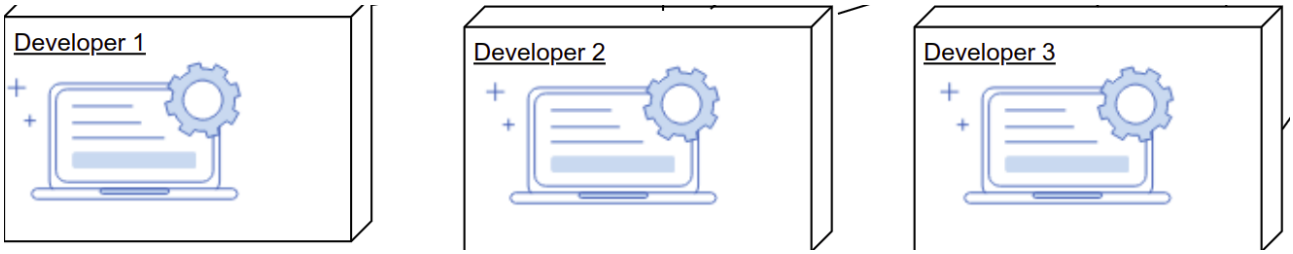


As expected, everything starts with developers work stations. Having correctly configured we are making sure that we are on the same page in development phase.

IDE (Integrated Development Environment)

Coordinated installations of IDE's (Integrated Development Environment, https://www.eclipse.org/, https://www.jetbrains.com), with company's code formatters makes sure that, there will be no collisions in development process itself. If developers uses different IDE, it will introduce non-uniformity. There is no guaranty that formatter used in one IDE will be acceptable for use by another one.
Second thing is code formatting itself, which should by uniform of company level, since without common formatter, code itself, will be different (at least from synchronization perspective), even no business code is changed, IDE it self might make phantom changes by inserting blank lines. For example one developer use IDE with default formatter and another one company's one. On save action, code will be reformatted (usually configured in IDE) in accordance to formatter (insert new lines, break long code lines, etc, etc). Even that there was no code change, or one just delete superfluous blank lines, another developer will see these changes and it would appear to him like entire file lines are changed (not just for example erased blank one), which might introduce confusion and irritation.

So, you are commit some code, and you see that it is not in synch with remote file. You want to see a changes, and potentially merge your file with that one, before commit, and suddenly you notice all file lines are different. WTF ?!
That is not something you want to see on each of your synchronization (only real code changes are accepted), but uniform development process. That's is why, at the very beginning, standard for development IDE, and formatting must be established.
However, I feel this as a natural place to address another convention which should be obey during the development process. And it is concern about code revision, mostly emphasizing a following coding conventions like, in semantic terms, meaningful names for methods and classes, and another things which will bring to clean and understandable code (remember, tomorrow someone else will sit on your place, and take your code to work with). It is not usually part of architecture job description, but it is good to mention as well.



Subversion systems

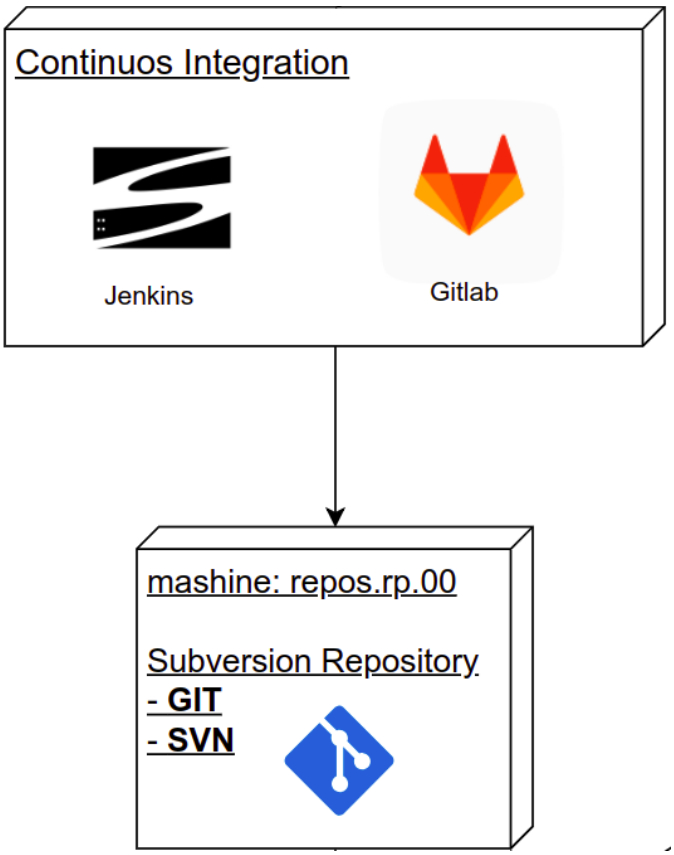
No matter even if you are a single developer on the code, you **must** be using one of the subversion systems. For several reasons:

- ✔ You do not want that, blocking and failure of your development machine, lead to code lose
- ✔ Company code is no your property, so having access of that code to another one, is mandatory for many reasons
- ✔ Usually you will work in team, with same code, so exchanging your modification and merging with existing one, without any issue, is mandatory as well, thats why subversion system must be used (you do not want to share your classes via skype and merge using diff)
- ✔ CI (continue integration) is tightly related to subversion systems as well
- ✔ You want to track code development progress and be on the *right page* at any moment

Subversion system, along with uniform work station set up, is another part of the story, which lead to quality insurance of software development, just as support for one of the service level requirement - **maintainability**.
Which subversion system you gonna use (**SVN** https://subversion.apache.org/ or **Git** https://en.wikipedia.org/wiki/Git for example, as most popular one), is question which is as I mentioned, related to automation servers as well (at least, for me, that is main concern of relation).
If you do not have any automation servers and continues integration, from my perspective, it is quite up to personal preference which one is going to

be used. Otherwise, if that is something concerning you, then you have next solutions.
If you are using **Git** as a subversion system, it is quite natural that you are going to use **Gitlab** (https://about.gitlab.com/) for code compilation, running tests, automated deployment, and continuous delivery, just as for issue tracking (which is going to be covered later).
Gitlab is a great tool, where its usage will just improve more quality level and **maintainability** itself.
If you choose SVN, then **Jenkins** (https://jenkins.io/) is perfect counterpart as automated tool, to perform more/less everything crucial as you can perform with Gitlab.
Once you have your commits and deployments automated, you are gonna raise quality of code, reduce errors, and improve project continious integrations (https://en.wikipedia.org/wiki/Continuous_integration).

Email notifications for **compilations/tests run/deployment** status (success or failed), is a great way creating a global consciousness of what is going on, for all members of interest, within a company.



Secure communication

Security is definitely one of the most frequent buzzwords on the internet, and lately, especially when we are talking about software. Even more, software development security is included as one of most important level to achieve of all service level requirements.
Code repository should be accessible only to chosen one, and in this case, to developers and another company members, which are only related to particular project. Let say that we are using Git/Gitlab combination of subversion system and continuous development.
Project ABC, should be available only to members of that project (developers, product owners, project managers, etc), and also, with different permissions as well (in regard to role).
On Gitlab, you can easily do setting which person has what permission related to specific project, and that is only high level of overview, at which is, as you see, can be established secureity as well.

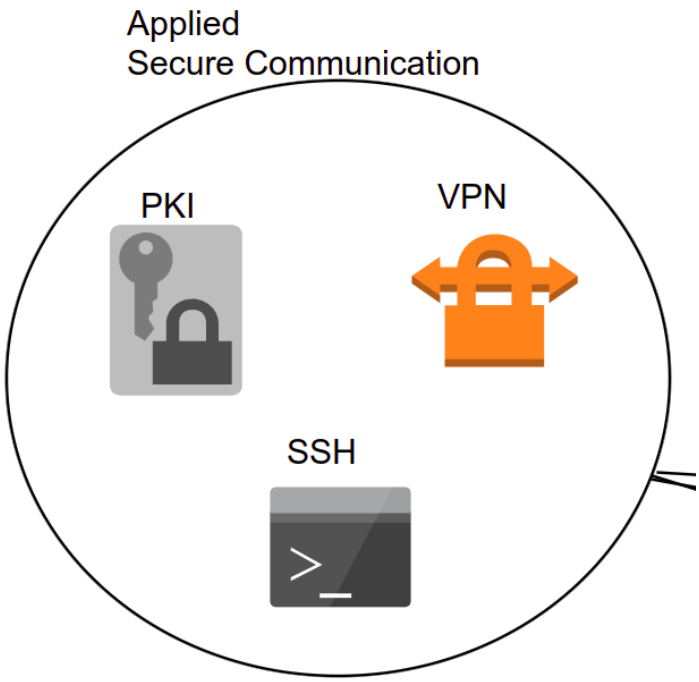
Developer needs to download project from subversion system, but only project related to that persons. That means, we must make sure who is accessing our code. Applying that security constraint, that only authenticated persons can download our code, can be easily and securely done by **PKI** (https://en.wikipedia.org/wiki/Public_key_infrastructure).
Developer creates public/private encrypted key (using openssl for example), and provide repository server with public key. Once authentication occurred on code download attempt, developer is authenticated (or not), using this strategy (matching private/public key with configured pass-phrase), and gain access to repository.

Another security aspect is **VPN** (https://en.wikipedia.org/wiki/Virtual_private_network) usage, if we are accessing from external location.
Usually, code repository is accessible only from specific location (company ip range addresses), but not external (ip address of code repository is simply not accessible to anyone from external ip address).
In that case, we can do such a server setup, that using VPN client, with authentication credentials, allows developer to access code repository (actually machine itself where repository is held) from any location.

Last, but not least, aspect of communication is covered by using **SSH** (https://en.wikipedia.org/wiki/Secure_Shell), for communication with company server machines.
It works with PKI (as described above), and any connection to server machines (occasionally by developer but frequent by **devops** and **sys admins**) from employee machines, should be done this was (as only possible and preferable way).
All the time one would have needs to access server machines for different purpose (check logs, manage some cpu process, give some permission, etc.), and using ssh for that purpose brings up security level.

Using **VPN**, **PKI** along with **SSH**, is actually approach which raise up security service level requirement on top.
You always need to configure system in such a way to cover several aspects:

- From which location your system is accessible (**location** aspect)
- Who can access your system (**authentication** aspect)
- Who is authorized to access resources (**authorization** aspect)
- Is communication secured from interception (**eyedropper, main in the middle** aspect)



Issue tracker

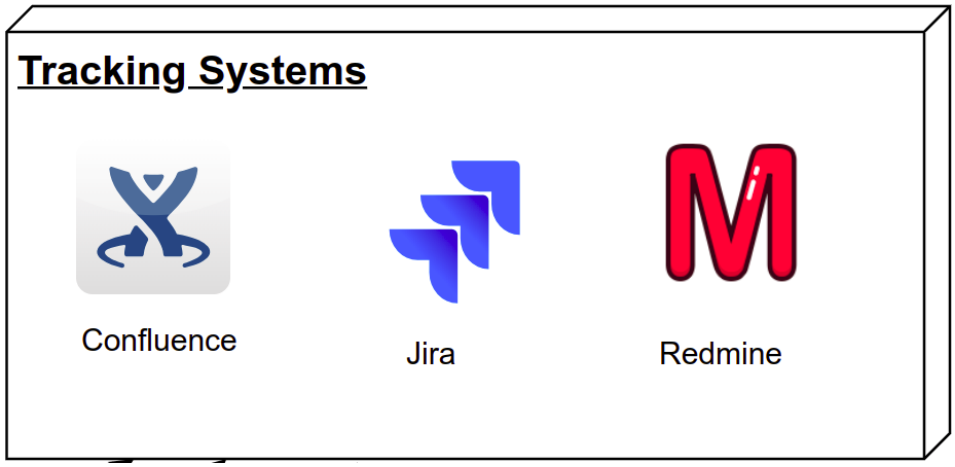
Task descriptions, definition of epics, stories and tasks (if you are practicing SCRUM), bug reporting, documentation related to issues and system as well, needs to be handled as well in, I would said – any, company, of any size and structure.
Here we are talking about issue tracking tools like **Jira** (https://www.atlassian.com/) and **Redmine** (https://www.redmine.org/), and documentation tool like **Confluence** (https://www.atlassian.com/software/confluence) pages.
Issue tracking tools helps us regarding issue organization, software feature definitions, and their tracking within implementation process, with comments and attachments.
It is simply centralized and aggregated in one place, and represent a software itself in lexical form. Once issue is defined, and assigned to developer, one can easily track which developer is in charge, then, check potential issue or doubt by reading a comments related to that task, as well as occurred bug which are related.

We can easily link another tasks one to another, just as to create hierarchy and inheritance of the tasks, by following some business logic.
Issue tracker can be easily integrated with Subversion and event hooks (for example if commit contains jira task number, upon commit, automatically comment of commit will be append to that task in jira), just as integration with Confluence, for documentation purpose (eg. you are creating a document describing some business logic of feature, and you want to make a connection with specific jira tasks), etc.

Here I would also take time refer to usage of **comments** in general, and their appreciation (importance) for improving code quality, just as great effects on **manageability**, **maintainability** and system **extensibility** starting from:

- ✔ Comments in code (describing a business logic of methods/class wherever it needs, and it mandatory existence at places with complex business logic - if someone new check your code, and is not sure what does it do from glance)

- ✔ Comments in commits, referring to issue tracker task, and brief (but essential) description of it
- ✔ Concise and precise task definition in issue tracking tools, just as comments following progress of task implementation

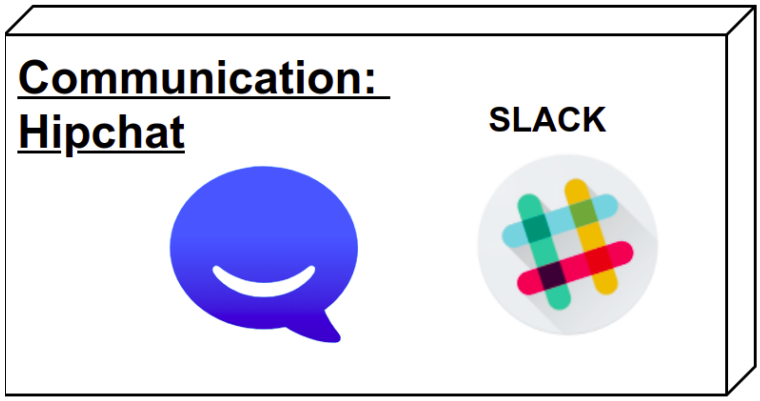


Project member communication

Communication between members is **crucial** part of, not just within their department and software development, but company entirely.
Having reliable communication, is best achieved if you have at least, at some level management of that communication. I am talking about security and conversation persistence, and not about communication access for bosses to check what their employees talk about ;).
Skype can serve purpose as well, to exchange information, but having own server, for communication purpose, is simply reliable for communication security and persistence.

Slack (https://slack.com) is one of the choice, but **hipchat** (https://en.wikipedia.org/wiki/HipChat) as well. I have been using hipchat mainly (along with skype, of course), and main advantage is that communication is kept on server in persistent state, just as secured as well. It can be used as a fall-back resource for forgotten things (once pc crash down, no more skype data), but in this way, you can always access your

chat history (which is from my perspective, sometimes more valuable than real time chat info I can get).
Research and pick one for your solution, since it has more valuable advantages .



Conclusion

As you can see, **development** and **deployment** are tightly coupled and quite related one to each other, and I could not make clear separation between them.
It consists of several sub-layers, each one comprised of several technologies cooperating together, or with several choices to pick up as a solution. I stressed which service level requirements are affected, by appliance of certain solutions for particular sub-layer, which probably always can be done another way, with substitute of proposed solution. But anyway, my goal is to give you Big picture of this layer in application development process, bringing you full fledged and workable solution.
As I mentioned at the beginning, you should always adjust system architecture for your business requirements, add/removing parts, depending on your specific needs.

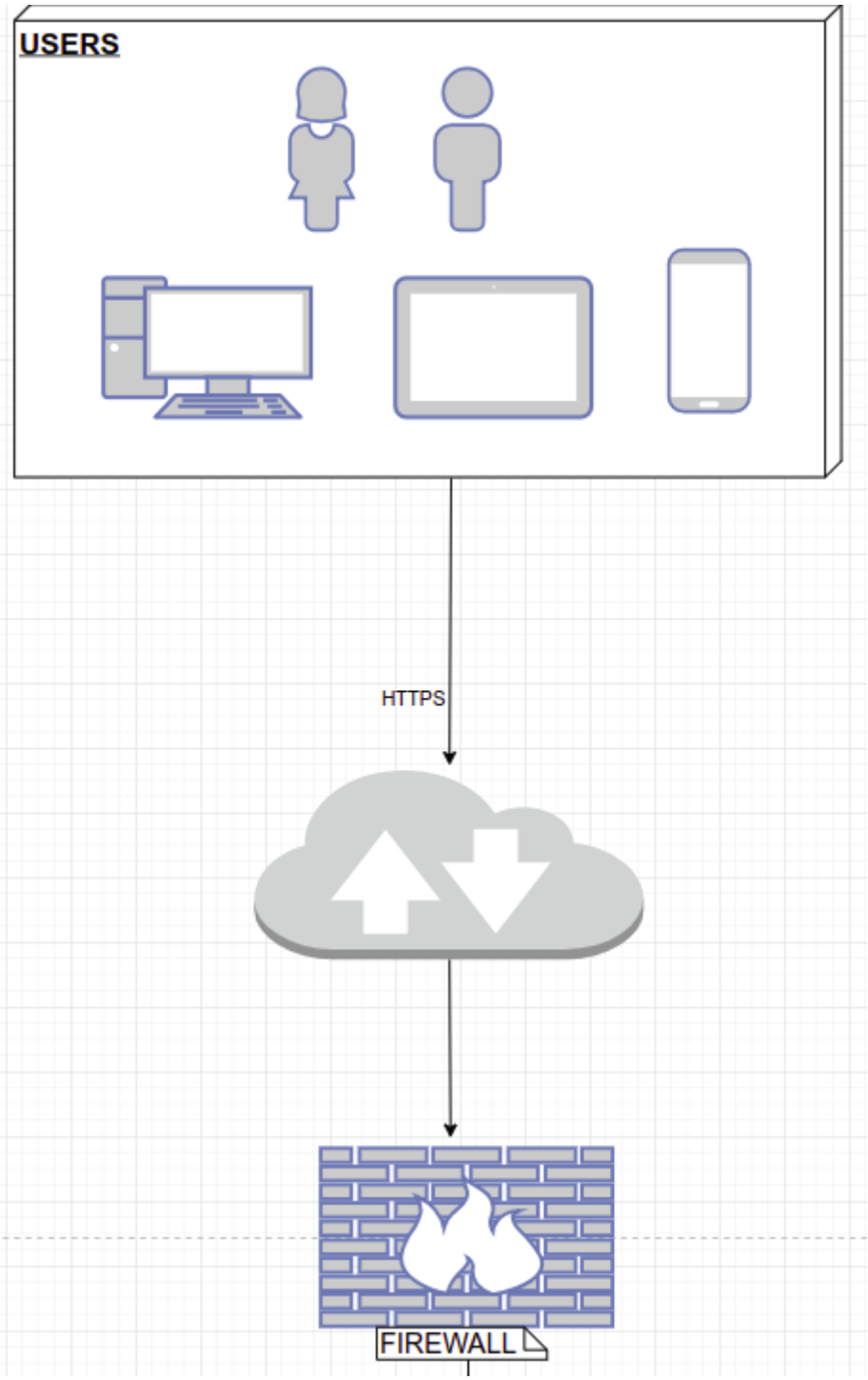
Firewall and Load balancers

Since I want to be in synch with Big picture graphics, before I dive into presentation layer, I must explain two ***bounding*** parts of the architecture. Firewall as an entry point of your system, and load balancers in general, as glue of your application layers, maintaining some of crucial service level requirements. I will give you an overview of both parts, not going into depth, because, they are huge areas if you want to go into details, and most of the setup and configuration would requires specific knowledge, usually provided by system administrators.
Hence, here I will refer to main reasons of including them in the architecture, and how they perform.

Firewalls

As a main entrance point of your system, in the architecture where it exists, **firewall** (https://en.wikipedia.org/wiki/Firewall_(computing)) can be configured as a software just as hardware.
Difference is, as you might guess, in costs and complexity, at first place.
Software firewalls, can also be distinct, starting from built in software like **iptables** in Linux OS, which can be configured to grant/deny access to different services and ip address ranges.
You also have full control of what coming in and goes out, having insight in the payload as well (if encrypted not much interesting things to look into).
Along with iptables, you can find a lot of freeware just as payable solutions for managing network traffic, with more-less costs (but usually much less cost than hardware firewalls) and documentation as well.
Software firewalls are much easier configure and manage, and simpler to use either via command console or web interfaces.

Hardware firewalls are used usually in big enterprises, by large companies and corporations. They perform better in comparison to first one (they are payed more as well), and they are loosely coupled between system and internet connection. However, along the cost difference, it is much more difficult to maintain and configure them, requiring specific knowledge related to particular option, in order to get the best of its features.



No matter which one you choose (probably selection will be made based on the real needs just as money you dispose of), choosing a firewall in front of you system has both pros and cons. Considering them thoroughly, you can decide what is your solution.
One of the main pros for having firewall is monitoring of incoming/outgoing network traffic with potential examination of the payload. Also, configuration of accessibility for certain resources in general, or from specific ip ranges.

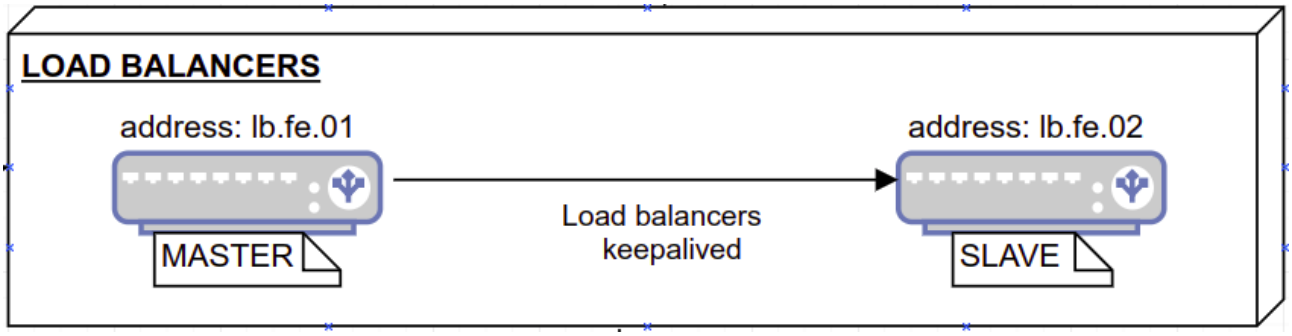
Anti-spamming and **anti-virusing** modules, are usually easy to configure and provide you quite high level of security. Another important setup is concerning **DDoS** attack (https://en.wikipedia.org/wiki/Denial-of-service_attack, means having huge volume of requests coming from thousands or millions ip addresses, making site unavailable since system cannot respond to them in some acceptable time, due to this attack), since you can actually filter and limit incoming connection to alleviate DDoS attack (Firewall is not best solution as strategy mitigation, but can serve the purpose).

Having all this in mind, it is quite conceivable that **security**, as QoS feature, is covered by firewall configuration, but also a **managability of system.**

Biggest cons is, from my point of view, single point of failure, and potential bottleneck of the system. However, these days, firewalls are quite upgraded technology, which handle successfully all potential issue (huge bandwidth, automatized operations, failure recovery, etc.).
Firewall, is not something you are mandatory to put immediately in front of your system (talking about system building from scratch because, just as premature code optimization is bad stepmother, plugin-in bunch of features in architectural design in advance, without precise needs for that is the same bad habit and very wrong approach of architect), but definitely option which should be consider during the time, whether it fits system needs.

Load balancing

As you can notice, **load balancing** (https://en.wikipedia.org/wiki/Load_balancing_(computing)) clusters are scattered all over system architecture. It covers and address to one of the most valuable service level requirements, which **reliability** and **high availability** are.
Topic regarding load balancing, exceed these pages and I would actually refer to some of the books talking about it in depth.
Again, we are handling two types of load balancer, **software** and **hardware**, with quite similar pros and cons as we had for firewalls.
However, here in example I am going to talk about software used for load balancing (**HAProxy** http://www.haproxy.org/ in this case), since I found is suitable for most use cases.
Hardware load balancers are much more expensive solution, but it has their own features, which makes them more eligible for large corporations.
The bottom line is that, purpose of load balancing, as word said, to properly balance incoming requests within system, providing high availability of the software service. It is actually tightly related to **redundancy**, since request load is distributed over multiple servers, according to selected **balancing strategy**.



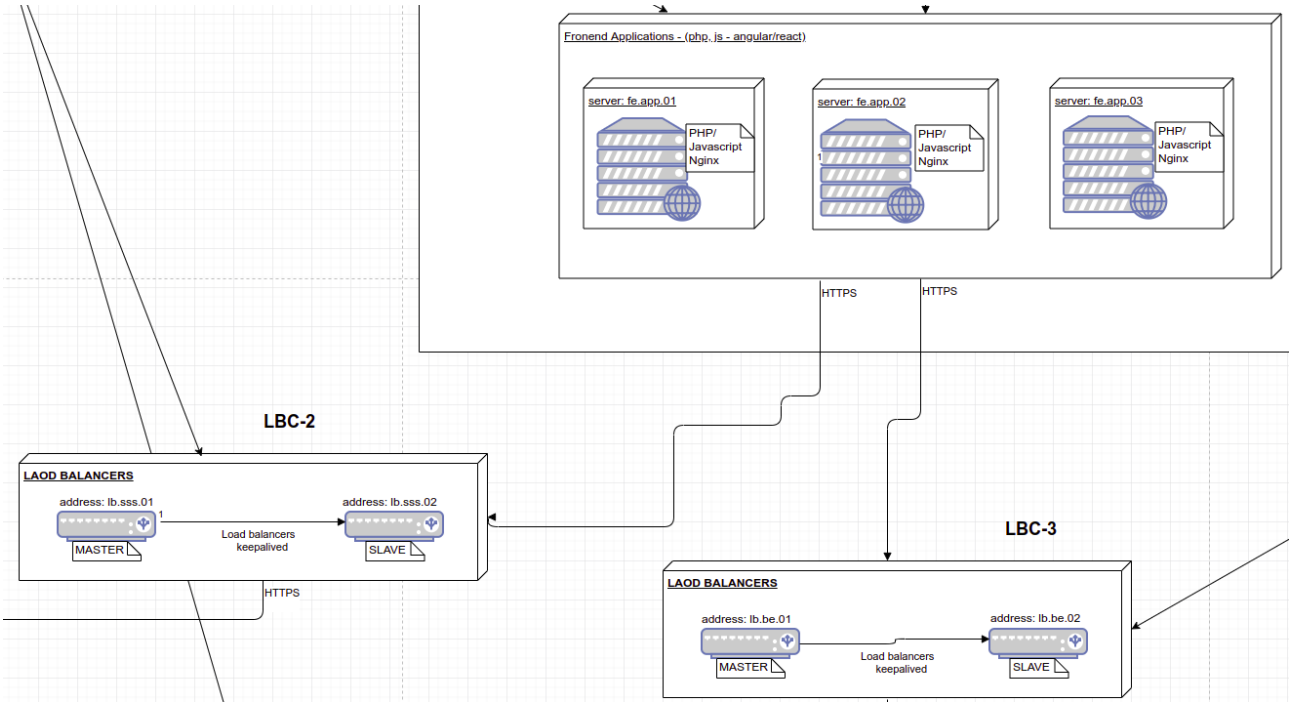
Depending on the location of the load balancing cluster in your system, it can serve for same purposes but address to different service level requirements (we would address at each position of LB cluster within diagram).

Load balancing not clear? Let considered most frequent usage of load balancers.
Check this out following situations, where you have single server, where application is running on. Application handle incoming/outgoing requests, and thats fine. However, for simplest situation, we do not want to have a downtime of the service, when we do new release of the application, or that server for any reason goes down. That would jeopardize high availability as service level requirement, and makes your application inaccessible. To overcome this issue, we would introduce more than one servers and load balancing.

Now we can have a situation where load balancer receive initial request for the application service, and redirect it to particular server (chosen from cluster). If for any reason server or application goes down, load balancer can be easily configured to redirect user requests to another servers which are up.
It is simple for load balancer to figure out when server goes down, since they are checking the health of the all servers in the cluster all the time (listening for so called *heart beat* of servers), and had information of which servers are up and which down, before they route incoming request to specific one.

Load balance cluster

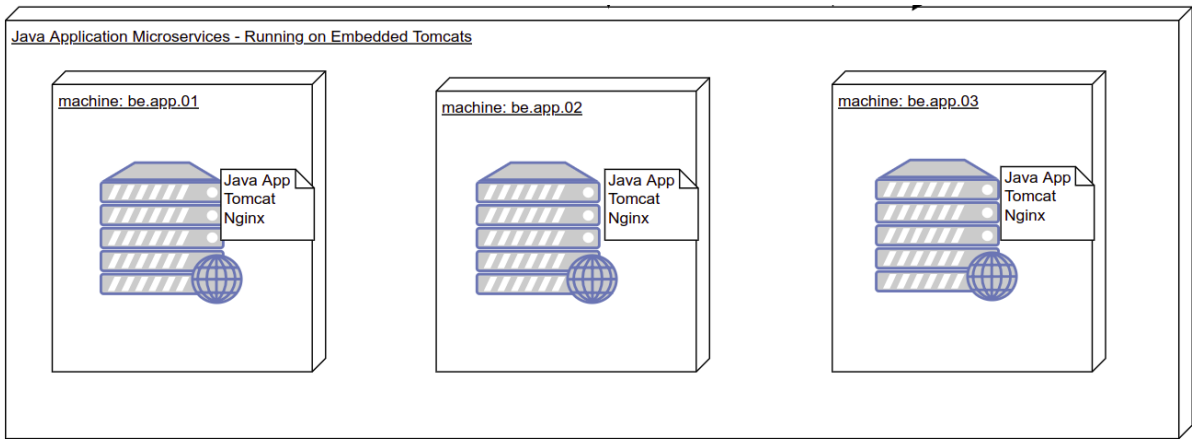
Even more, having single load balancer in front of the application cluster, can also lead to single point of failure (which is what we deeply wants to avoid).
If that server machine, running load balancer goes down, or load balancing software itself get stucked, no traffics to your application will be redirected, and it will be unavailable.
Thats is the reason, why I put load balancing cluster (two load balancers in this case) in front of application tier, because I want to maintain **high availability** of load balancers as well.
Avoiding single point of failure is actually ground motive to have clustering of any type of platforms (application, server machines, load balancers, database servers, etc.)
Let have an example in diagram, what situation we have and how it works.



In this architecture, we have a **master load balancer**, to which, all requests are routed from router (whether its firewall, or internal application itself who initiate requests to load balancers or else), and **slave load balancer** (so called master-slave topology).
Every request is targeting master load balancer, and slave is **redundant** one, which serves only as **hot stand by**, meaning, it will be engaged at very moment when master goes down.
In this case, we have sacrifice investment in slave load balance, serving only for that purpose (and hopefully never engaged).
Using **keepalived** configuration, load balancers are in constant communication one with another, checking health status, and if master goes down, slave will immediately start receiving incoming requests, and from client perspective, no downtime happens.
Example of simple load balancing configuration can be checked on this resource https://www.howtoforge.com/haproxy_loadbalancer_debian_etch.

Now, let return to our application clustering.
In the use case of application failure, described above, introduction of redundancy is done only for serving as fail-over fallback, in case when master server goes down, and it is actually quite ok when we are talking about load balancing cluster.

However, when we talk about application cluster, no utilization of resources, is not actually architecture usually used, because of waste of resources, if we have a server serving only for this purpose.
More likely that our redundant application servers, will actively be included for serving requests. In our case, as you can see on diagram, we have a three servers in our application tier, in front of which we have a load balancing cluster.
Scaling of application servers, we also gain support for **scalability** as a QoS feature, getting opportunity to handle more requests, and increased traffic itself.



Number of application servers, containing same application version, is something which is decided based on needs (one of them is maintaining requested performance, as requests per seconds).

Architecture of load balancers, just as balancing strategy, deeply depends of the system and application nature, in front which load balancing cluster is setup, for example, whether we talk about **stateless** or **statefull** system.
Choosing **load balancing strategy**, according to which, requests are going to balanced among the cluster of application servers, is something which should be configured based on your application logic. More about them, you would need to check detailed implementation of strategy, but here is overview of some of them.
For example you can apply following **load balance strategies**:
Random, Round and Robin, Least frequent used, Sticky session, and others.

Choosing a strategy, must be coordinated with your application purpose and business needs, in order to pick up one best fit your needs.

Random strategy, sends requests to random clients. **Round and robin** is also called next-in-loop, since it sending requests sequentially to the list of servers (list of ip addresses), so user A will be sent to one ip address of server, and user B to another one, selected from the ip addresses pool.
Least frequently used, means that, incoming request will be handled by least used server (so lowest loaded server).
Al of these strategies sounds fine when we are talking about stateless services (REST services which does not maintain user state), so we can apply most suitable of them to **LBC2** and **LBC3** clusters on diagram, but when it comes up to user state (session) management, then sticky session is best choice.
Sticky session strategy ensures that, each incoming request of particular user (so with the same IP address), is handled by the same application servers (hence session can be managed easily by application server, since every request of user A will be handled with server AA).

Quite handy solution regardless your technology solution (Servlets, PHP, or else) depicted on diagram with **LBC1** label.
Techniques for using sticky session strategy, would include some additional tunning and configuration like, whether we want **session replication** to achieve reliability of the QoS level, with several options how to perform that (**In-memory session replication**, or **persistent session management** with different persistent store, etc.).

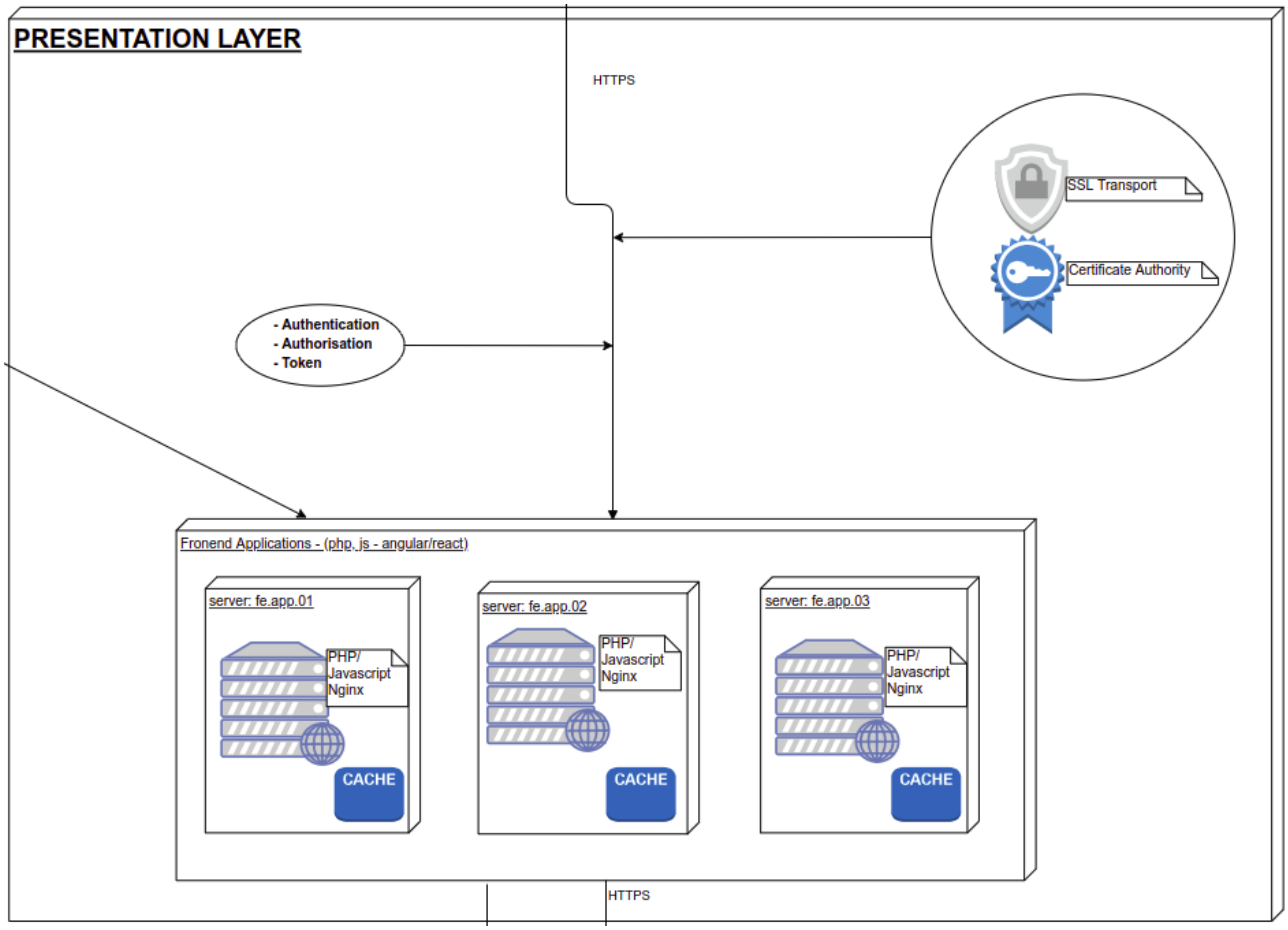
However, there is a tendency to manage user state in different ways, and to leave backend as stateless services (reasons for that is decoupling between layers, easier maintenance, separation of concern between layers, etc.).
Anyway, depending on the place or load balance clusters in our diagram, they could serves to satisfy different service level agreement as well, depending on the layer it is laying in front of.

Conclusion

Firewalls and load balancers (especially), are crucial part of any enterprise (even non-enterprise) application, which has to maintain and achieve certain service level requirements levels, like **high availability**, **reliability** just as **scalability**.
Even I didn’t graphically depicted these two terms as a layer, they are definitely layer on their own, with not only worth of mentioned, but unavoidable within architecture planning.
Making sure that your application is available for customer after single server crash, easily handling increased traffic, just as monitoring of this traffic, are matters of these two (firewalls, load balancing) architecture structures.
Again, decision what is going to be choosen and how, should be matter of application business requirements, not default or preferences.
I would also accent that for both parts, firewalls and load balancing, there is usually plenty of option and supports by hosting providers (if you have dedicated servers, or virtual private servers like Hetzner, DigitalCloud, or Linode) or by cloud provider as AWS.
There are automated solutions for covering failure parts, DDoS mitigations, and many other options, which should be checked by your hosting providers.

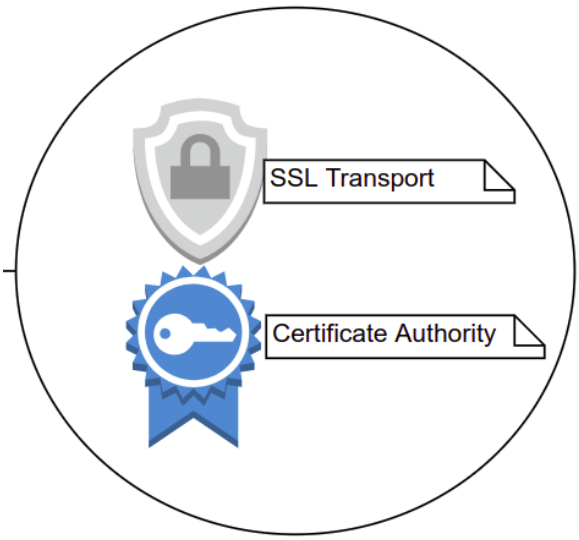
Presentation Layer

Once we have users incoming requests processed by firewalls and redirected to specific application server by load balancer, then we are entering presentation layer.
At first place, I would address to communication way itself, between user (client) and system, just as usual processes which occurred during this communication, and potential structure of frontend application part, including cluster of the application servers as a physical machines, and copy of the same application version running on each of them.
We will dive into details for each of them, at level this manual allows us.



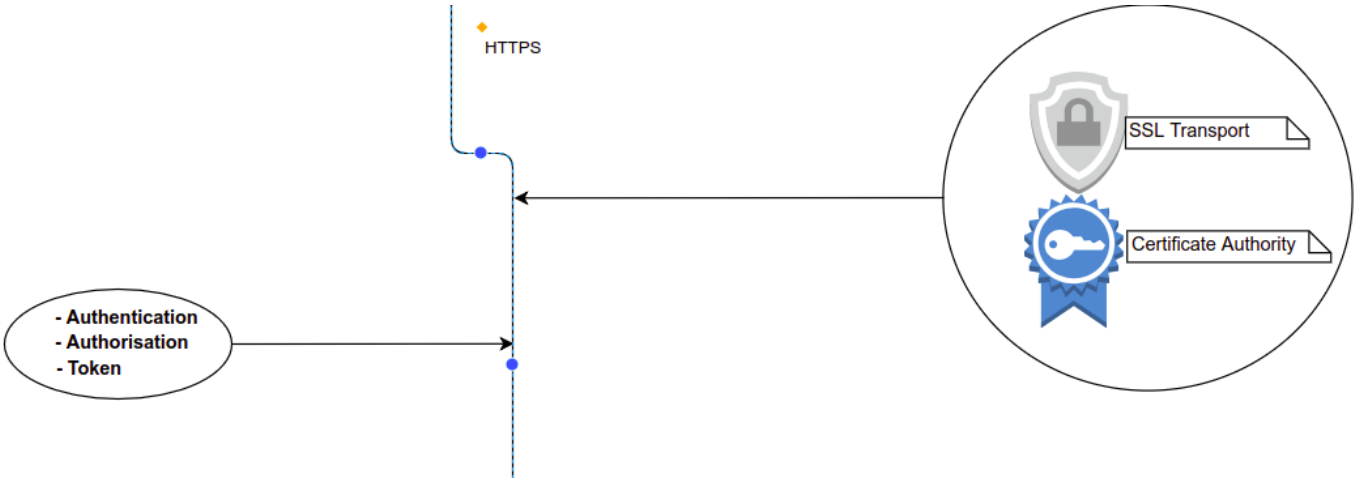
Communication way

Once client (web browser for example) direct request to server, it is done by default via **HTTP** protocol. It is protocol developed for communication, in this case browser and app server, and it is just fine. However, HTTP has one major vulnerability which is transparency of the data. In other words, anyone who has control over networks like ISP provider, or if we talk about internal networks communication with intranet then network administrators, or even wifi network interception, transferring data are exposed to **eavesdropper ,** or so called **man in the middle** (https://witestlab.poly.edu/blog/conduct-a-simple-man-in-the-middle-attack-on-a-wifi-hotspot).
This way, man-in-the-middle can not only have insight into communication between two parties, but would be even more able to modify content of the transfered data, and instead receiving “I love you”, Bob sent by Alice, he will get “I love you **NOT**”, because Alice’s jealous ex-boyfriend intercepted their communication and alter.
That is why we use **HTTPS** layer, which actually HTTP over **TLS**, or HTTP over **SSL** (https://en.wikipedia.org/wiki/HTTPS). Securing this way HTTP protocol, we are safe since entire communication is encrypted, by **PKI** principle (check below more about his).
Further, each request/response exchanged by client and server are encrypted and decrypted by arival on the destination (both client and server), with exchanged shared key.



One would think that is all regarding communication, but what if we have an imposter on the server side? Instead facebook, we could have a malicious site named facebok? Even this is obvious misspelling, it is actually quite common way of passive attack, since not everybody looks throughly in the name of domain.
Accessing such a site, can lead to providing them sensitive data, and be conned somehow. Implementing https connection for your server, is also providing **reliability** and **confidentiality** for your customers. In order to maintain and offer client https communication, your certificate, initially exchanged to client browser, must be signed by so called **Certification Authorities** (**CA**s). Having approvement of this certified authority, that everything is fine, and certificate holder is the one it present him self)so not imposter), client can be sure about identity and confidentiality of server with whom communication will be continued.

Keep in mind that server does not need to have signed certificate by CA, but created one on its own just for establishing encrypted communication, but in this case, client (browser), are going to be warned that it accessing unreliable site, requesting manual confirmation of user if he is going to proceed to thin unsafe location.
Certificate exchange, and secure socket layer is not only applied on direct client/server communication, but also within enterprise system in B2B communication (Business to business communication), in oder to ensure **reliability** and **security** as service level agreements (great article about HTTP / TLS communication: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm)



Communication processes

As I mentioned as a second part of presentation layer, we must address on several concerns in client/server communication, from security perspective, which should always be considered in system architecture design.

Authentication, is definitely the first one since you do not want everyone to have an access to every resource of your system, but more likely, selectively. Even if you are building most publicly wide software, you will have some part of them which should accessible by only preselected persona (like administration part for example).
Username/password is usual way to authenticate client, where in the case of B2B authentication, certification exchange can be choice of authentication, or some and the other ways depending on the technology used by these B2B communication.
Authenticating user means that, it is a guy who is presented to be, and resource should be accessible to him/her. In the case of username/password usually we are talking about permanent persistent stores (like databases), where these information are stored in **encrypted** form, and before checking existence in our database of such a user, data are encrypted as well, using secure way, and then compared against another authentication data from database.
You do not want such an important data as user credentials to store in database in plain form, even if you limit external access to database. Security, is never redundant!.

Authorisation, is second step in process, because, even if the user is authenticated (meaning, it really has been registered by our system and does exists within), we must check whether it is eligible-authorized to access requested resource. For example, you have some customer applications, which can be used only by registered users. Some parts of that application are intended to be used only by certain people, like administration part to be used by administrators only.
If registered user, non administrator, after authentication (so user exists in our system) tries to access this admin part of the application, we would

check is he authorized to access this application part, and if not (since it is not admin), it is going to be rejected.

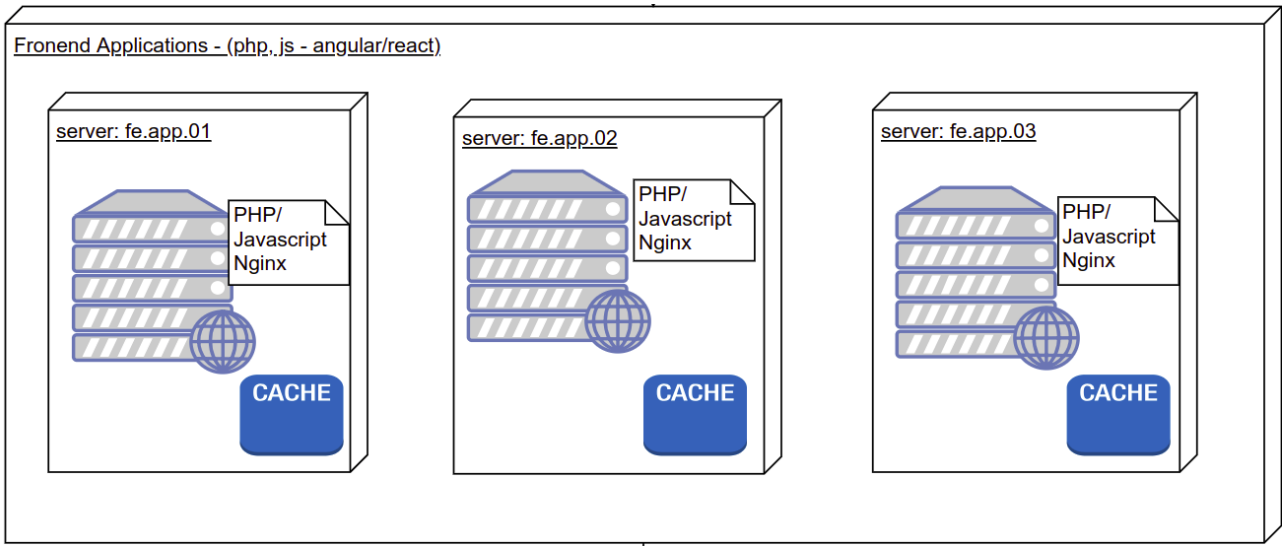
But, what about accessing these resources, intended to be accessed only by authenticated user, if we try to skip them? What stops some hacker to get list of web services, and access/use them without any kind of authentication, in other words, not coming to authentication page at all, but directly to invoke those web services?
We really want to track our users, and to log who is visiting what on our site, and not to have some phantom reads. One might cross mind, hey let enforce authentication upon each request, so even if one bypass our authentication page, still every further request must include username/password and we will perform authentication and authorization on each request.

This solution, basically, could serve the purpose, if we are looking from perspective of securing resources. However, this way, we would actually seriously damage **performance** as QoS feature, since we would need to check database for every single request, and probably several another databases accesses related to particular business logic which targeted web service consists of.
If we have quite load on our system, with thousands of simultaneous users, we would easily end up with database bottle neck, influencing application performance.
Also, from coding perspective, it is not quite clean situation, where upon each request you would need to perform authentication.
Exchanging **token**, for each request, generated on authentication process, is actually one of solutions.
Once authentication occurs, we generate a token, related to that authenticated user, and exchange it during the communication between client and server. Upon each request, we check for unique **token presence**, and reject request without one (and probably ban further requests from that Ip if it continue and we label it as malicious).
In Java, for example, exchange of **JSON Web Tokens (**https://jwt.io/**)** can be good solution, with **jjwt library.** JWT actually helps you to create encrypted token, using algorithm and level of encryption of your choice (very fine tunning is configurable).

Also, information embedded within token are customized, best suiting your requirements, where for example, authorization role is perfect place to embed in.
Another strategy for application access securing, which is common technique and worth of mentioning, is making that application (ip address of machine where app is running), visible only to users, which requests are made from specific ip range (usually, making resource accessible only from company network, and creating it unaccessible from outside).

I would not say that this is definite list of security strategy during client/server communication process, but at least most crucial one regarding architecture itself. I could mention XSS, SQL Injection, and another attacks and system messing, but those are part of the software implementation, and something which developer should take care of.
But from the architectural perspective, and as a preface of fronted tier, as part of presentation layer, those methodologies and approaches were mandatory to cover.

Frontend applications

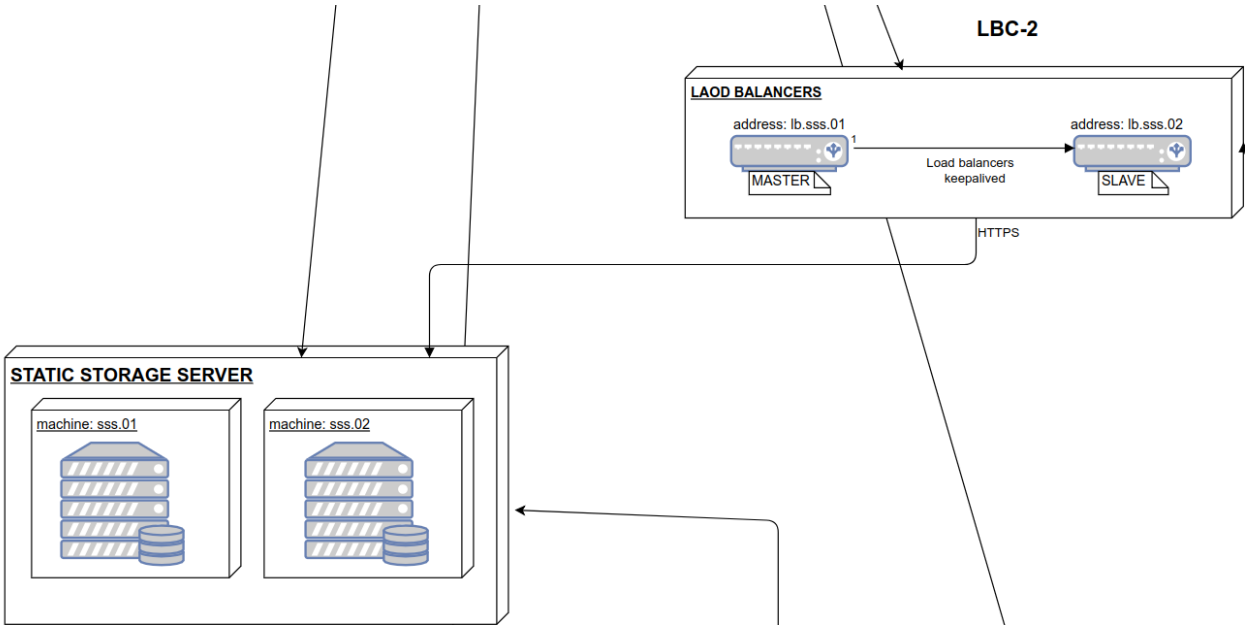


As you can see on the diagram, and in order to support service level requirements like **high availability** (as described on load balancing section), cluster of frontend servers is something default in this case. We

are talking about frontend because, in enterprises you will probably have needs for separation of concern between frontend and backend parts, where first one is in charge for display (views) and up to some level flow control aspect of the application, and backend where business logic relay on.
Frontend layer (applications hosted in this layer) will communicate with another part of enterprise, like backend layer (backend servers), storage systems (for static data), or some third parties as well.
In our case, load balancers direct request, based on the selected load balance strategy, to one of our machines. Each machine has running web server, hosting frontend application (let say Javascript – Angular/React or PHP).
Web server usually is **nginx** (https://www.nginx.com/) or **apache** (https://httpd.apache.org/) (sure, you have microsoft, google, Sun, and other, but I would stick with these two, as in my opinion, most frequent one). Even that both save the same purpose, *nginx* is preferred lately because of event driven methodology of handling requests, which is built on, and hence being able to process considerably larger request load than apache can. I am not saying that apache is deprecated or bad, but just that you should check which one is most suitable. If we are talking up to thousand requests per second, then it probably does not matter which one you gonna choose (very good article about nginx vs apache performance https://www.hostingadvice.com/how-to/nginx-vs-apache/).

The application deployed on frontend server, will communicate with **static storage system**, for retrieving a static data like images, binary files, or any another utility resource. Of course, it is not mandatory to have such a structure, but it is good practice to do separation of concern. If we had no centralized tier for storing such a data, then each machine hosting application, should have replica of the very same static data.
Imagine now bigger enterprise, running dozens of very complex microservices, on separate machines, and each one of the machine to hold exactly the same static data… You see the point that is does not makes sense for that, because of many reasons.
I have not separate storage system as separate layer, since it is kind a helper and utility layer, which existence within architecture mainly depends of the application type running on the servers. In diagram I stressed on that, wanting to provide insight how that would work from

architecture level, and why is as such described (we provide high availability with load balancing to those storage, and we extracted static files on one place, because of no repeating them).



Beside the communication with static data storage tier, in frontend application the user interface will be build and customized for every user, by custom volume. Even if the difference is based on the **login <username>** label, that username value must come from somewhere.
Any kind of customization, like background of the web app, messages exchanged with another users display, bringing up related ads to user, must be dragged out from somewhere, and usually these customizations are retrieved from the backend server, where business applications run, based on the required business logic.

Again, it is not mandatory structure, nor each every business data is solely stored on backend, can be in the **cache** as well, and probably will be in order to reduce communication to another systems.
Cache represent a **fast-access** mechanism, usually keeping data in memory for fast access, with or without disk synchronizations (depending on the type of cache).

In general, different technology stack would include different caching systems like **DOMStorage** (https://en.wikipedia.org/wiki/Web_storage) for javascript apps, or full fledged key value databases systems like **Memcached** (https://memcached.org/) for example used in PHP/Javascript applications or **Redis** (https://redis.io/), which can be used for this purpose as well, **Ehcache** (http://www.ehcache.org/) cache for java apps, etc.).

So, usually, you would need to flag which data are eligible to be saved in the cache, once frontend applications get data from backend.
Less frequent changing or non changing data are the best candidate for caching.
It will reside in the memory, without redundant traffic to business tier to retrieve them. For example, users personal data, or some very static non changeable data like country/city/date of birth, and stuff like that are perfectly eligible for caching.
However, if we have modification of some cached data like, user preferences, as something which might be changed, but in your application anticipated as rare, cache simply needs to be refreshed with this new modified data.
But having very frequent data changing in cache makes not much sense, neither from development nor architectural point of view, since at any change we would need to update, and refresh memory holding that information, which anyway, previously has been retrieved from business tier.
Eligibility of data for caching is something you need to think ahead. But sometimes, after application usage you might have additional idea, what data is great candidate for caching.

Conclusion

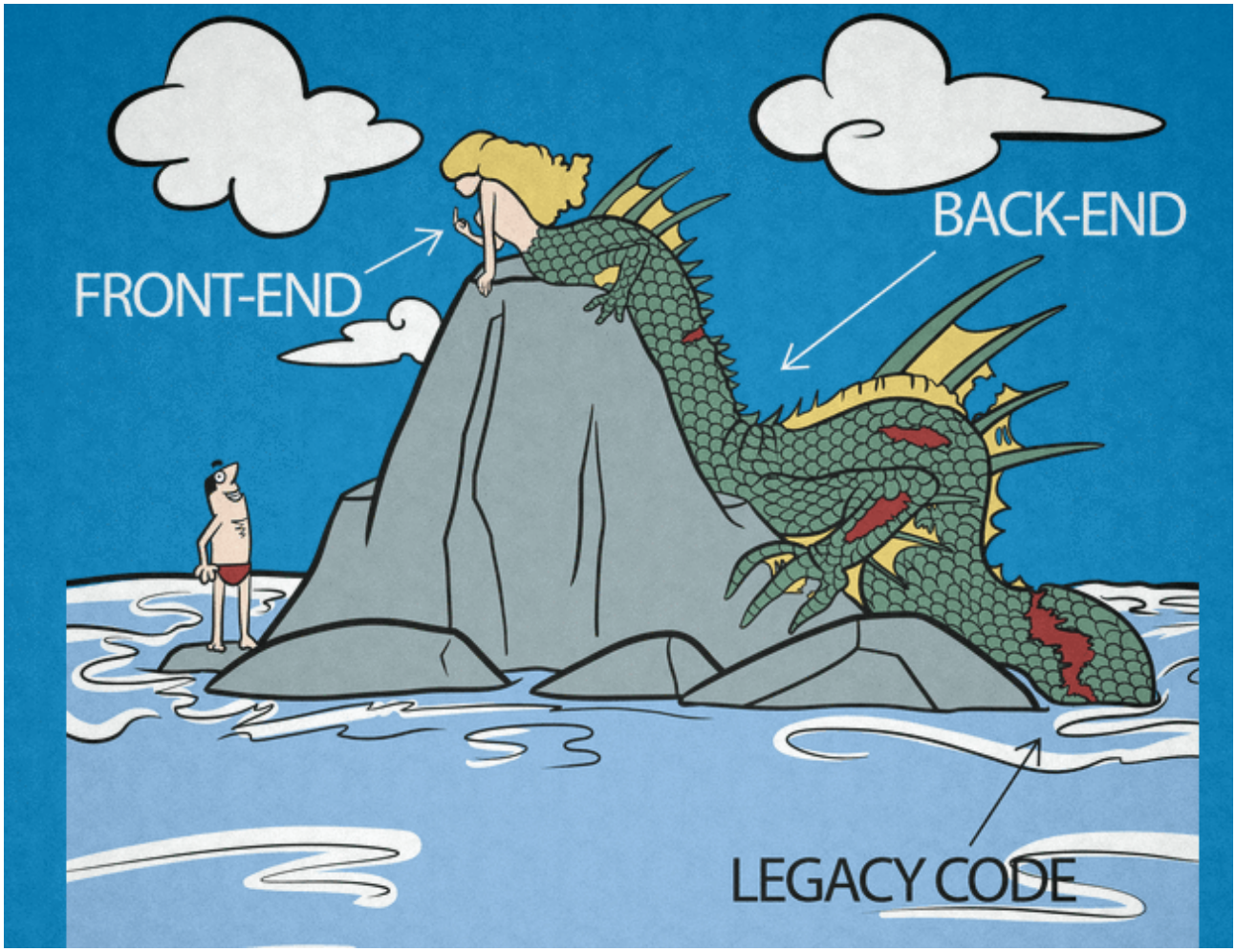
Regardless of the technology you choose to be ground of your frontend application, presentation layer, as described above, will be pretty much the same, with few tunings related to your very specific case like:

- in which language you are developing your frontend application
- which cache are you going to choose, will depend on your needs (application language, application business logic, resources you dispose of, etc.).

Communication way and usual **processes** which **occurred during communication** between user and frontend, will be probably the one I talked about, especially if your application maintain user states, and work with some user provided data.
Certain level of security will be always must (from my POW), at least using **https** for communication (where all data exchanged between user and your system will be guarded by encryption), and using **Certification Authorities** certificate, ensuring authority validation (in term that client can see the issuing authority of the certificate and the corporate name of the website owner.

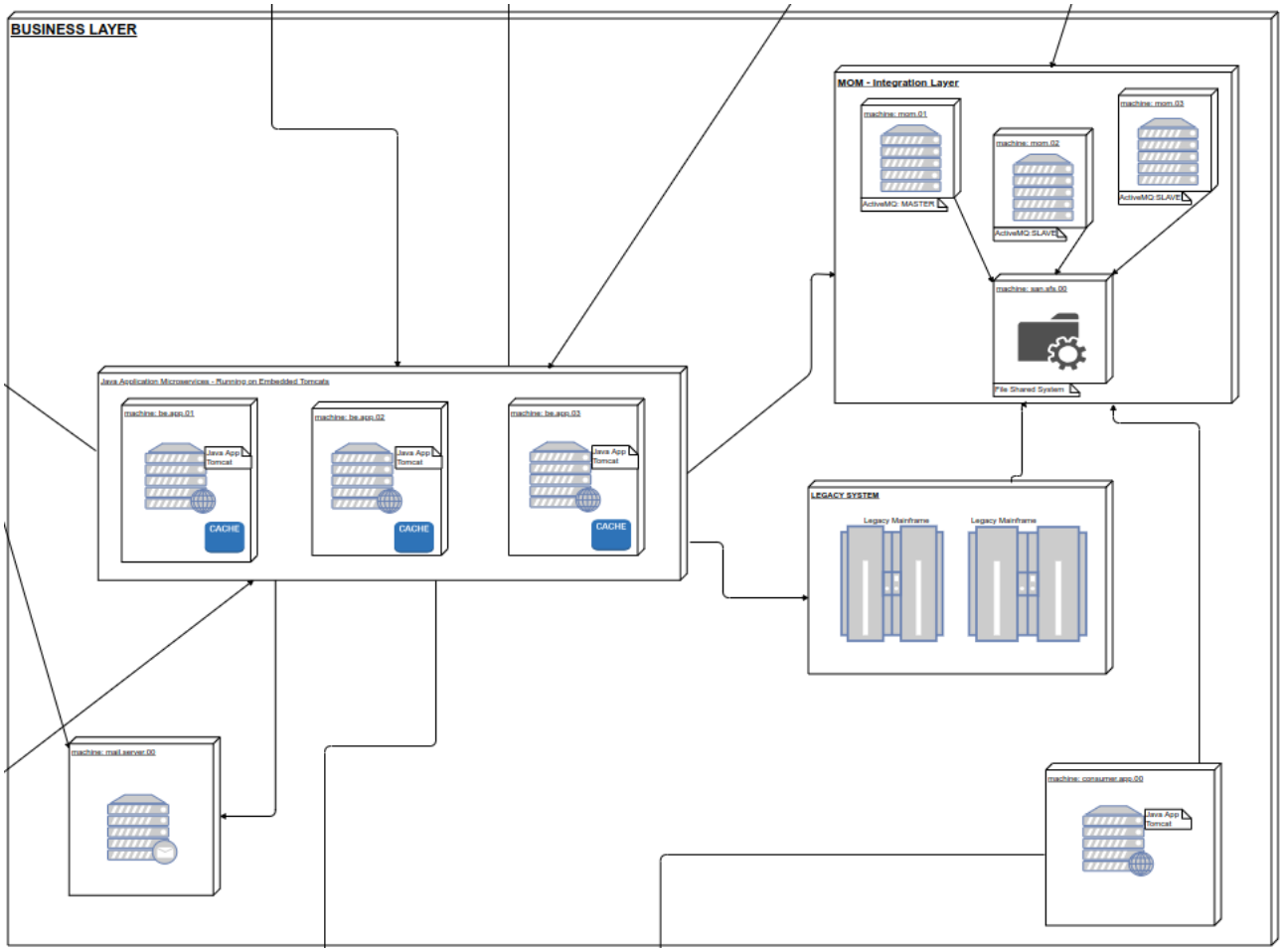
Business Layer

If presentation layer is face of the application, business layer is hart of it. Stereotype of the multi later (tier) applications and comparison between frontend and backend is usually depicted as:



The truth is not far away of this image.
Backend tier, or business layer, can be that much complex, to blow your mind. Here I will simply accent several components which most likely are going to be included in your business model like: **core business application**, **integration layers** with message oriented middlewares, integration with **legacy systems** in terms of code migration or building new applications based on legacy business logic, integration with **third**

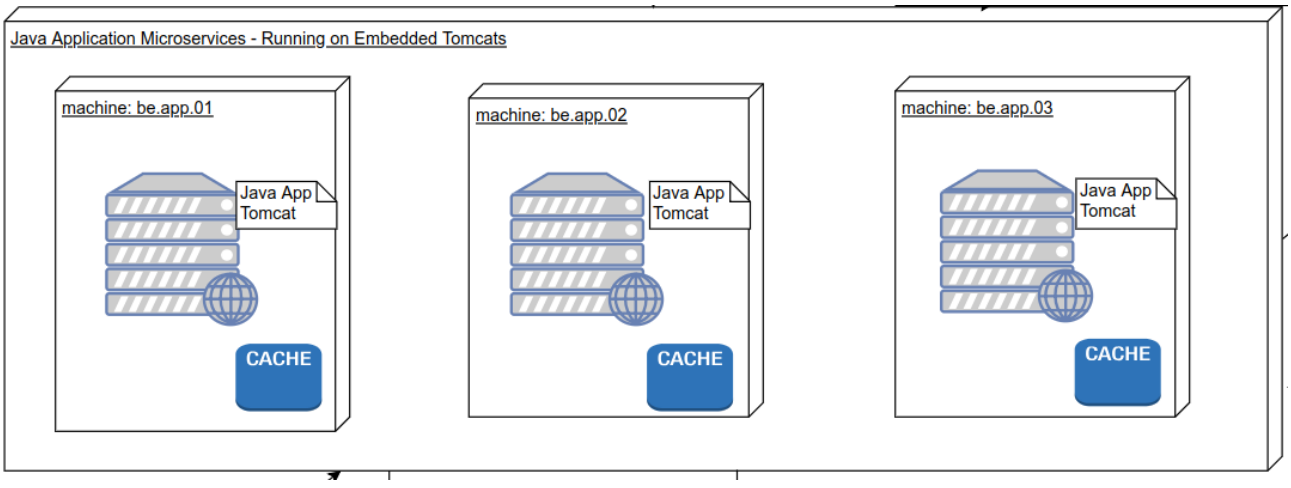
party systems, **notification system** with mailing server, just as **consumer application**.
It is going to be long and bumpy ride, so hold on.



Core business application

Just as you can see on diagram, and in similar way as we did with presentation layer, backend related load balancer cluster, direct request to our core application clusters.
Depending on number of operations per specific time range, which should satisfy our performance SLA, and an expected traffic volume on backend tier, we should make decision of backend cluster organization, number of instances and internal structure.

Regarding internal structure, it usually consists of application and server which is used for that purpose. If we are talking about java EE, application will be deploy either on full application servers like **Jboss** (http://www.jboss.org/), **Weblogic**, **Websphere**, etc, or on web application servers like **Tomcat** (http://tomcat.apache.org), **Jetty**, etc.
Difference is based on service which full application servers like Jboss provide, and which are not available at web application server counterpart like tomcat is.
If you want to maintain **container management** of transaction, scheduling operations, jms providers and some additional automated operations provided by application server tools and configurations, then Jboss, Weblogic, etc is right choice for you.
Also, if your application is relaying on container management of lifecycles like Enterprise Java Beans do, then full application servers are must.
Usually used by large companies and enterprises, management of these quite expensive softwares, requires appropriate knowledge for that purpose.
Also, configuration is more complex than web servers like tomcat setting up.
All these things said, does not mean that tomcat is not worth of, since it does not (or very limited) support those enterprise features. However, in conjunction with **Spring** (https://spring.io/), every Java application running on web servers like tomcat,jetty, becomes full fledged operable enterprise application, supporting all of capabilities listed for application servers.



Similarly, as we reduced a business layer call from presentation layer, by introduction of cache, here we are going to do same way, for our communication with database.
You do not want to go into database for every (frontend) request, especially not for rare changing data. Hence, usage of cache will facilitate all things, reduce database traffic and avoid potential bottleneck. **Ehcache** is most frequently used, in memory caching system, for java applications. It is simply library, which allows you simplicity of use, and very good performance.
It is used for both entity caching (with Hibernate) and/or dynamic data caching with Spring.

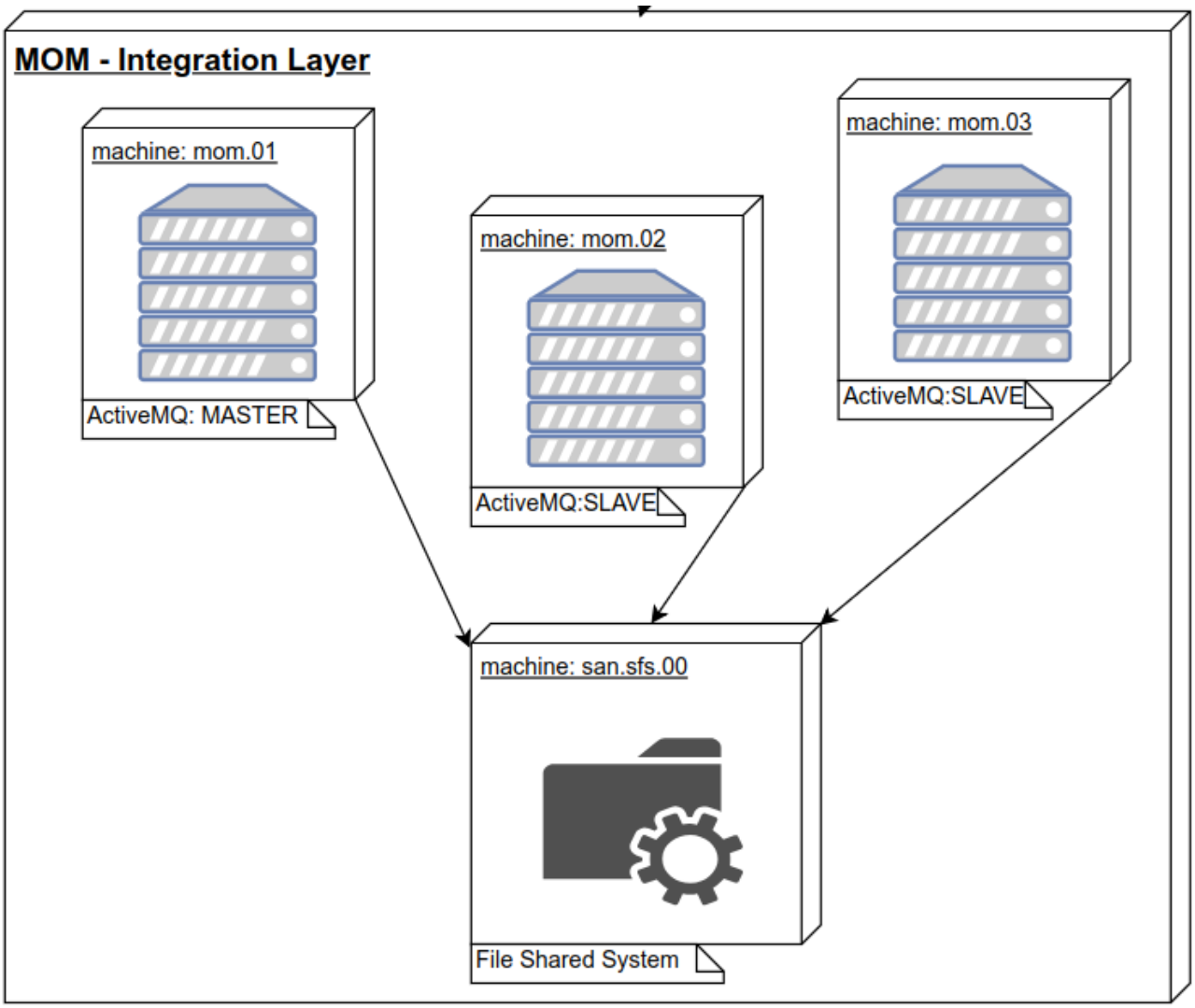
However, Redis become more and more attractive option to be used as a cache server. It is key-value database (more about it in persistence layer), which handle complex data structure to be cached, just as fine tunning how caching is going to occurs, remain in memory, flush to disk, etc.
Depending on your application business, it can be sufficient to deliver requirement, or communicate with another system. Usually the last will be option.

Integration layer

When we talk about integration layer, we are referring to leveraging our application or integration with another systems.
Leveraging is directed on separation of concern per application (microservice). For instance, after purchase is done by payment application, its concern is done. It is not that sending notification email, creation of invoice, storing data into internal statics and account system, will be part of payment concern. From payment perspective, payment is done, and has finish its job.
All following steps, are part of some another system parts (let say **email** app). Once payment is done, we will send asynchronously data to **JMS** (https://en.wikipedia.org/wiki/Java_Message_Service), to specific destination with required payload.

Email and/or accounting microservice is going to listening to that specific topic, and process data in regards to their own business logic (invoice creation, accounting management, email sending, etc.).

Having that separation of concern at application level, payment application will be much more responsive (hence **performance** will be better), and improve **scalability** and application **extensibility**, as one of three service level requirements, than if we would maintain all those side things within payment application.
Once payment did payment, its job is done and it should be used only for that purpose, not polluting business logic and purpose of the applications.



For JMS providers (talking about java apps as clients) we gonna most likely use **RabbitMQ** or **ActiveMQ** (http://activemq.apache.org/), and for the full list you need to do your own research.
My choice in this case was cluster of ActiveMQ instances, which is a message broker written in Java.
So let me check little bit about strategy and pattern used in this situation, where we have an enterprise system using message oriented middleware for internal microservices integrations.

There are a lot of patterns and strategies, how integration layer (**MOM**) could be designed, but as always, it depends on your application requirements and needs.
Here we have a cluster made of three activemq instances, where one is master, responsible to read/write, and additional two slaves, serving on purpose of hot stand by, in the case of master failure.
All three instances communicate between each other, checking the health and once master goes down, for any reason, first slave is promoted in master, continuing processing messages.
Third instance remain as a slave, and when first master comes up, it will simply jump in the place of slave.

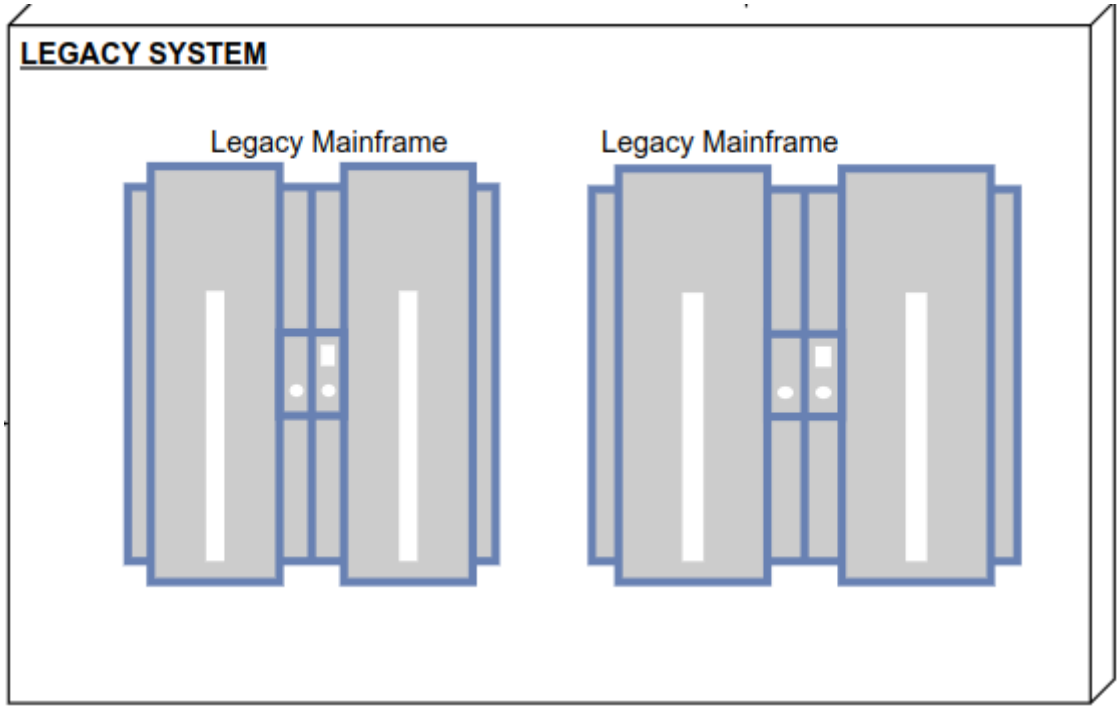
Brokers are stateless in this strategy, which means, none of them physically keeps data, but just **exclusively lock** of the system where messaging data reside. Data reside on separate machine, with is **shared file system**, and exclusive lock of that file system is held by the master until he goes down and release it. Once released, exclusive lock of shared files system is grabbed by elected slave, which become a new master.
This is typical master/slave scenario, with some peculiarities.
However, another strategies are available as well like, more complex network of brokers, or using another persistence system (if any) like databases (more about available strategies and features at official documentation http://activemq.apache.org/features.html) .

We want to maintain **high availability**, and this architecture helps us to achieve that, taking care that integration layer is always up, and ready to accept/deliver messages.
Client application (java application), communicate with this tier via **failover protocol**, in other words, application is connected to first up

instance of activemq, which is defined in the failover string (failover url consist of list of activemq instances, which should be attempted to connect in the case of first one failure, just as a bunch of additional key-value configurations).

Legacy system

Legacy integration can also use message oriented middleware (MOM). We could have legacy system, written in Perl for example, or any other language. The good thing is that any jms provider talk the same language, in other word, for jms provider, producer/consumer language are not important. As long as jms implementation api obey certain rules, which are proposed by broker. For example producing messages with JMS API, will be easily consumed by C++ Messaging Service which is a JMS-like API . In which language message is going to be sent/received, is more-less unimportant.



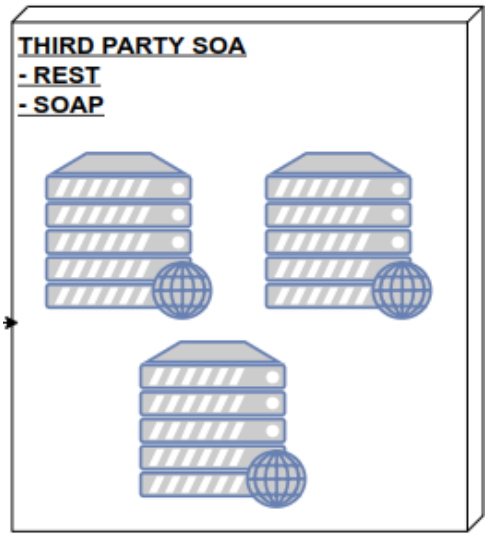
That is why MOM is actively used in integrations as par of ESB, new applications with legacy systems inetgration.

Point is that, legacy system (written in perl, c++, or else), simple provide data, obtained by anyway and applied any business logic upon, in form of message, where new system (written in Java), can easily consume it. ActiveMQ support multiple protocols, so you would have to investigate most suitable for you if default (**openwire**) is not best choice. This specifically means when consumer/producer of messages are written in different languages (depending on languages not every listed protocol is supported).

Another common approach for successful legacy system integration, or even entire migration, is usage of **shared database**. This mostly relay on Model Driven principle, where we have a database used by two different systems, but the new one usually has to coordinate its business logic, in regard to data from database.
Some kind of custom ESB has to be built, as additional layer, in oder to make database data usable for the app, since, we are anticipating that model stored within, is hardly completely adjustable by legacy system in order to be reused easily into new one.

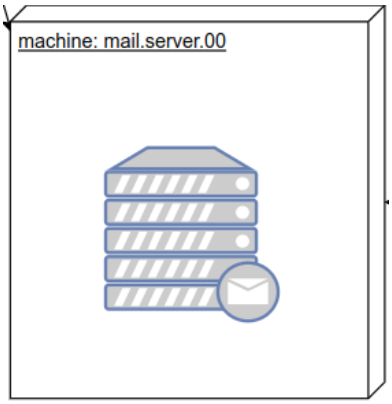
Third aspect of integration, and part of **ESB** solution, is implementation and exposure of web services architecture (REST or SOAP usually). Exposure of web service, and consumption of the same, for example if we talk about REST WS, can be also language and system independent. As long as we have agreed API of exchanged object, using JSON. XML or plain String, integration of our systems (whether we consuming or exposing data via web service), can be easy peasy job.

In modern applications, you are most likely that you are going to integrate your system with some third party, by one of these two techniques. SOAP, will be probably choosen if you want to bring some higher level of **abstraction**, **security**, and business to business communication. It brings with self, more complexity for implementation, and more knowledge in order to produce and maintain such a system.
On other hand, so far simplicity of REST light weight services, and exchanging data in proposed format of JSON or XML (pure String as well), is something which is embraced more, and more, as a web service default development approach, within development community.



Which techniques we are going to use, depends on the systems we want to integrate with each other, level of abstraction they expose, just as level of coupling we want to achieve between them.

There is also mailing server in this architecture, used for outgoing maling traffic (for example, once payment is done, this is used for sending an email to client informing him about success purchase).
However, mailing server can also be part of company's mailing system in general, including hosting of employees emails.

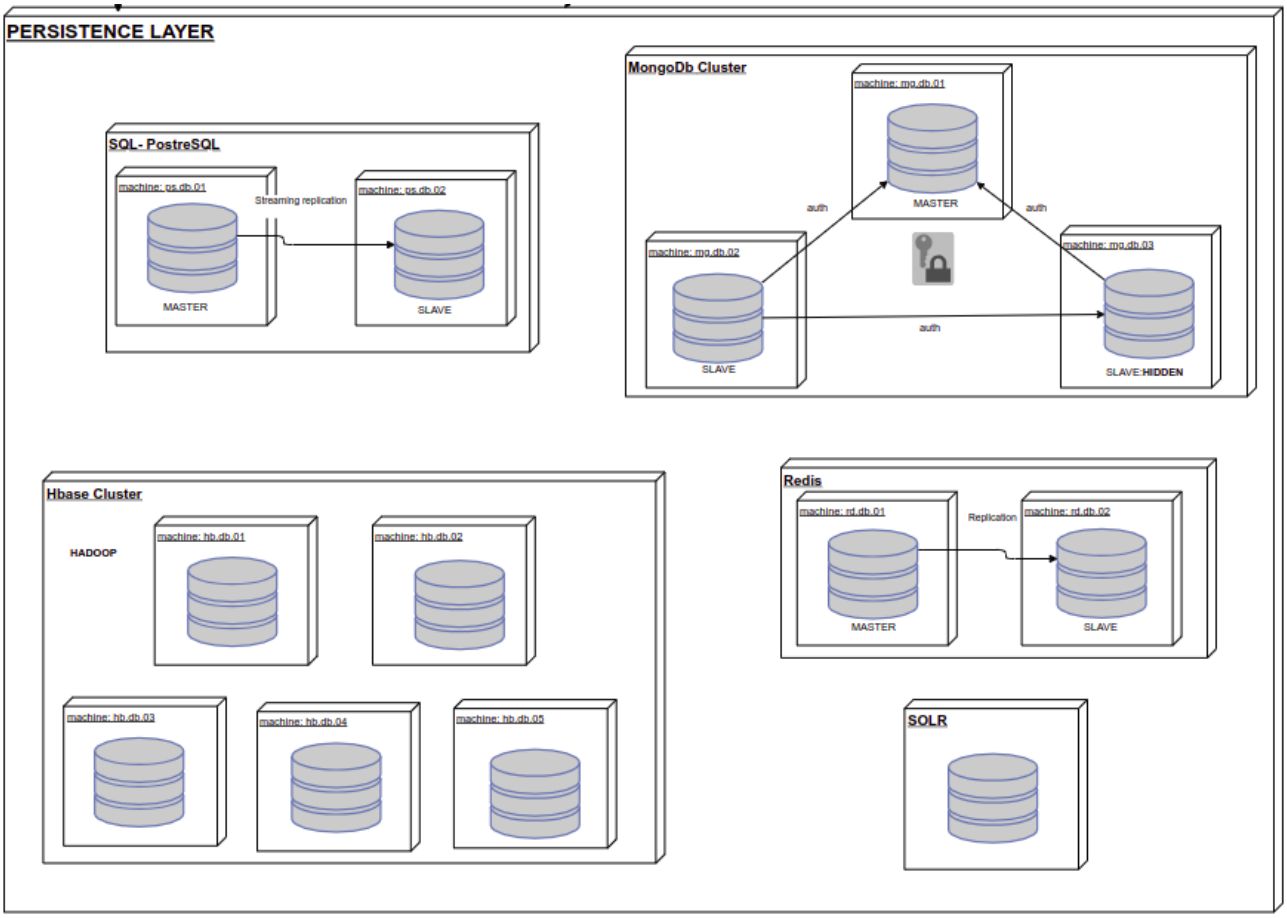


Conclusion

As I mentioned at the begin, this is just an example of business layer architecture, serving for system needs, and based on the application business requirement.
Not all technologies, and principles are cover, but the one which are cover will give insight to one and get cleaner big picture how business layer works, what is its architecture and how it fits with other system parts.
Also, I hope demystified which parts of layer serves for which purpose, and which technologies can be used in that parts and how.
I provided solutions, how some of them could be configured and used in practice.
Once business layer, and beckend, receive request from frontend, based on required business logic used for ground of application implementation, it is going to be processed by some of the proposed means.
At the end, as a result of that processing, it is probably going to be persisted, and persistence layer, our next station on user’s request journey.

Persistence Layer

In our request motion path, persistence layer is place where everything end up.
Persistence layer represent a modeled and static version of your business requirement, and your application in general. Selecting a technology and providing a solution for persistence layer, deeply is connected to application's features and behavior.
Different persisting technologies actually helps you to pick up best solution for your needs, and even more, you are not tight to only single persistence system, but you can use several them in enterprise applications, addressing to different aspects of your application as appropriate solution..
Persistence polyglots actually means usage of several database solution as a persistence unit for enterprise system, with mixture of **SQL** and **NoSQL** systems.



For example, SQL databases are going to be probably used for those parts of the system which requires **ACID** (https://en.wikipedia.org/wiki/ACID_(computer_science) implementation, like payment transaction, user registration, communications between users, etc.

Having SQL database for a long time period around us, would actually makes them very hard to replace, with NoSQL. I am not saying that NoSQL is not capable of supporting all mentioned uses cases above (payment transactions, etc.), with appropriate configuration and usage, but there are general rules why certain database type is better to be used for somethings, or the very purpose of the database.
You can write html using bash shell as well, but that is not how bash shell should be used for.

On the other side, SQL databases are not going to be used for some another purpose like caching, indexing, data analytics, graph representation of application data, but depending on needs, particular database system is going to be selected.

As you know, NoSQL database can be divided by three main data models:

- **key-value (Redis, Riak, etc)**
- **column family (Hbase, Cassandra, etc.)**
- **document (MongoDb, CouchDb)**
- **graph databases**

Schema-less nature of these database, is sometimes the biggest advantage they (NoSQL databases) have in comparing to classic SQL database model. Schema-less means that, no record is in obligation to follow mandatory data structure. In SQL, you cannot have a model, containing a field which you want to save in db, but does not reflect row structure within a table (not defined). It must be predefined.
In NoSQL systems (mostly) you have no such a constraint, so you are not forcing data model to consist of mandatory, databases imposed, internal structure.
Here business logic of the application will be in charge of doing so.

Aggregate Data Models

Providing detail description of each of these data models, just as existing technologies for each of them, is out of this book, so I would advice you first to learn about each data model, and figure out which one is most suitable for your specific business data, and after them to check concrete technology solutions you have on market, which are going to be used within system. Anyway, I will give you an overview for each of them

Key-Value data models you are most likely to use in caching part of your application, or to store some very simple business logic. There are usually fast lookup database solutions, based on the known key. Not just simple data structure like strings, value can be maintained as a different data structures (more complex) like **lists**, **sets**, **hashes** in **Redis** (thats why I said *very simple business logic*).
Having this ability to store complex data structure, there might be misuse of these system and involve them into architecture as a main persisting layer, for storing business model. From my pow, that is wrong, and should *not* be happen, since very purpose of these kind of databases was not that.

Fine tunning of persisting mechanism, memory management and flushing to disk, is easy manageable, which give simplicity of these systems usage.
I want to give one digression here, since one might note that I mentioned **Redis** (https://redis.io/) as a part of the previous layer, and how I will have it here in persistence layer as well.
We should actually use technology as it suits us, and not to bound it tightly to any concepts and terms. In this case, Redis can be used as a cache solution in business layer (as described in business layer section), but also can be used as an ultimate persistence solution for your architecture as well, if that is what is going to satisfied your business requirements.
For long time period, Javascript, was not even conceivable to major number of developers, as an backend technology, but today, here we are with Nodejs.

Graph database are going to covering different interconnections between model (nodes), by providing relations between them and easy traversal search of graph node.

Social media is best example of this database usage *where you have a person, which has a friend, who has another 3rd level friend, which is a second level connection for you* (something like linkedin connections relation, facebook friendship models, etc.)
Neo4J (https://neo4j.com/) is probably most famous database of this type.

Column family comes as a cannon, when we are talking about big data and clustering. **Hbase**, **Cassandra**, or any another column based solution will best suits you if you have giga or terabytes data size.
Node failure is even calculated as default within this structure, and large quantities of node, is something column family databases are targeted.
Even more, those databases (Hbase https://hbase.apache.org/ precisely), will have awful performance if you try to use in on small scale (5 nodes is minimum recommendation according to official documentation), but as larger your cluster is, it performs better (crazy, but it is as such).
Along performance and scalability of these systems, they are usually used for analytic purpose, logging and research (analyses) of these logged data.
Hbase runs on its own ecosystem, and is built on **Hadoop** (http://hadoop.apache.org), platform providing **distributed file systems**.
Column family persistence systems, are really a big gun, you are going to use only in big, big data, and for very specific needs.

Document related database, are probably most one related to SQL, from perspective of data presentation. Document, is a json representation of data, which can be compared to row of SQL databases, but schema-less of course.
NoSQL databases, especially column and document based, are most likely that are going to be used for analytic purpose, and especially for analytics of data values. Document in **MongoDB** (https://www.mongodb.com/) consists of json, with key values, where, comparing to SQL, key would be column names, and values row values.
Making analyze and examination in SQL, of row values is not possible or very limited available to be done. Usually, one would need to get row values, and then apply examination of values.
On the another side, we do examination/analyze of Mongo document on persisting servers itself, very easily. Means that, our analyzing query will be performed by database server, and retrieve data after that query is applied.

Usage of mixture database systems SQL/ NoSQL can significantly leverage our systems, and escape bottleneck situation, which is usually issue when is comes up to persisting layers.
In diagram we have a **Solr** (http://lucene.apache.org/solr/) which is fast search database platform (could be elastic search as well), with numerous built in features like (highlighting searched terms, providing distance parameters for required terms in the text files – eg. find occurrences of word president, having 50 characters before end line, etc.).
Here I put it in diagram, just as example of different databases systems, not going into details.

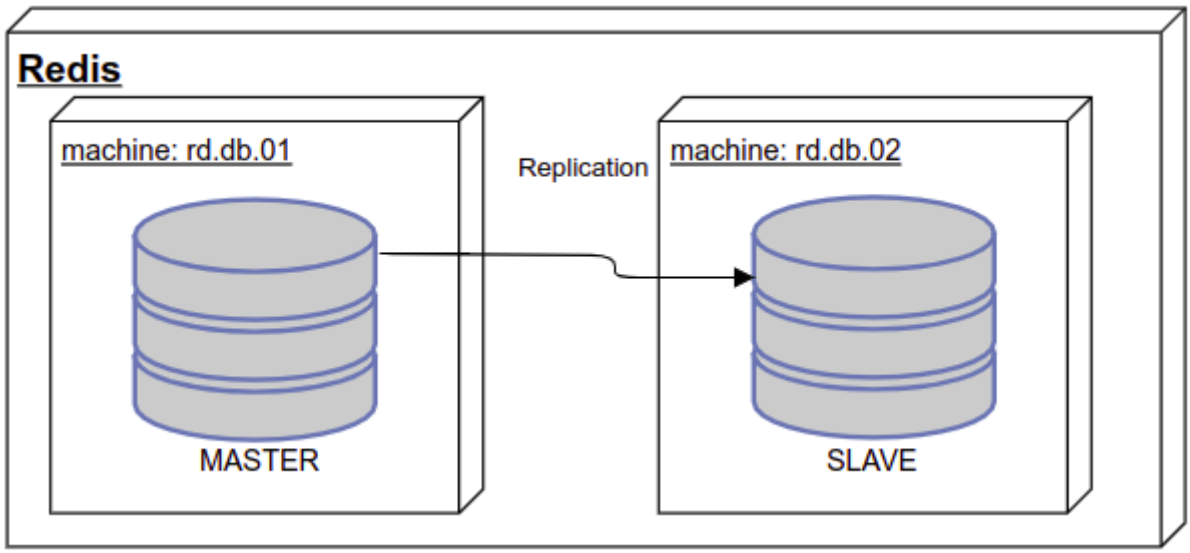
Anyway, I am going to explain usage/configuration of few database systems, whom application (business layer) would communicating with, solving real issues.

Caching tier

Lets say that we three different tiers within persistence layer (**redis** with master-slave cluster configuration), **mongo** cluster, and SQL (**postgres** in this case), addressing different purposes of the application data models.

We are going to have redis setup, with master slave replication. We want every tier in our persistence layer to provide us **high availability**, at first place, hence we have a replication.
I said at first place, since cluster address to **scalability** as well as **performance** as service level requirements, but with appropriate configuration and architecture.
Here, with redis cluster, master is in charge of receiving requests, and slave database instance serves as a hot standby server, which will keep copy of master Redis database data.
Since slave is '*monitoring master health*', in case of failure (both - master machine where Redis is host, or Redis itself) slave will be promoted in master, and going to keep providing application with read/write service.

Redis, with correct configuration, is fast as hell (you can do custom benchmark https://redis.io/topics/benchmarks), and with appropriate usage, hundreds of thousands requests per second can be processed, so having this Redis structure as depicted on diagram, is probably be appropriate addressing **scalability** and **performance** of that part of system.
You must always pay attention of configuration complexity or specific platforms, just as knowledge needed for that, in order to maintain **maintainability** of system as well. Remember that people are in charge of everything, so how easy you can get right person for specific job, is actually rate of your system maintainability.
Luckily, Redis provide you all of that (simple setup of master slave https://www.devops.zone/ha-performance-caching/installing-redis-2-8-masterslave-on-debian-wheezy/).

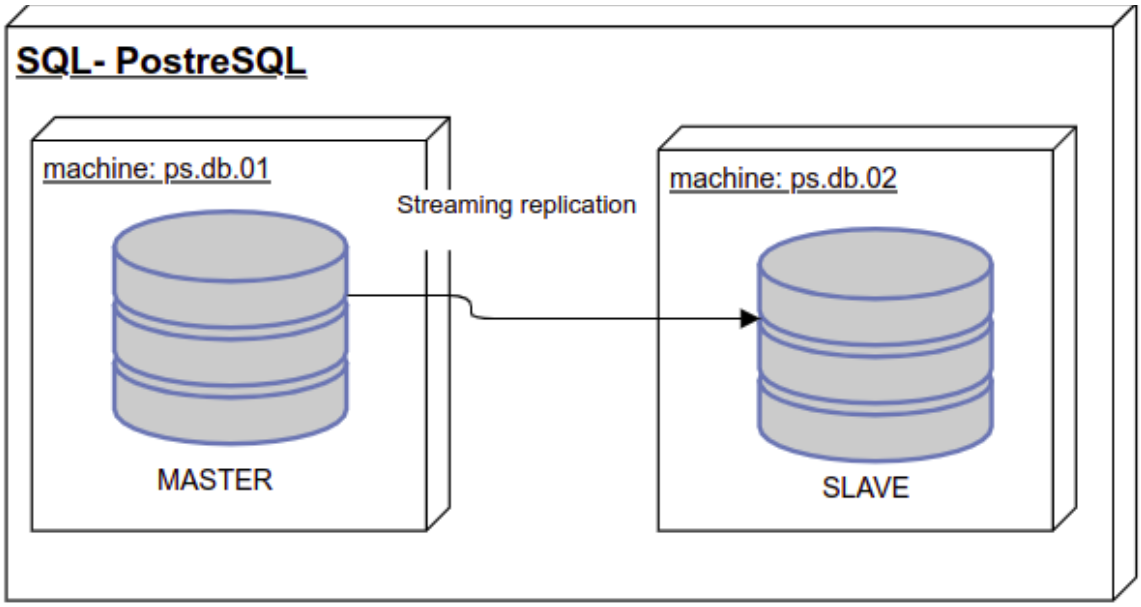


As I mentioned in caching section of business and presentation layer, benefits of fast look up will be for frequently queried data, especially if we are talking about static or rare changing data.
Disk read I/O is expensive operations, so memory lookup is a great choice for this purpose.
Once Redis reach threshold for certain objects in terms of timeout or amount, it will flush data to disk, so **reliability** is achieved as well.
What logic is gonna apply for this purpose (what would be timeout for objects residing in memory, number of them, their structure, etc), is

something which should be heavily considered before we start our Redis cluster, and based on application business logic and requirements.

ACID tier

Probably every application have needs for **ACID** achievement. For example, making a payment process on your site, when user do some purchase, you want that process, handling user data, to be **isolated** business process, which outcome should be accessible at any point in the future, hence **durable**.
You also want that payment process to be **atomic**, so either success or failed, not be in non-resolvable status (some kind of technical limbo) and to save eligible payment data into database, ensuring about their **consistency** with existing data related to that particular user and payment. For this purpose, SQL database system (in this case I selected PostgreSQL), would be perfectly choice.



Master slave replication provide us with high availability as well, where streaming method for replica is usual one of way of replication.

Master will play single point of write, but slave can participated as well for reading (however, this can bring consistency issue, if the same app is perform this).

Pre-NoSQL persistence solutions, usually brings databases as a main bottleneck of the applications, but here as you can see, we can leverage that and create different part of persisting layer to serve different purposes (as you saw, introduction of Redis, will reduce database traffic, for cachable data, but also Mongo cluster will came up as a another leveraging solution).

NoSQL tier

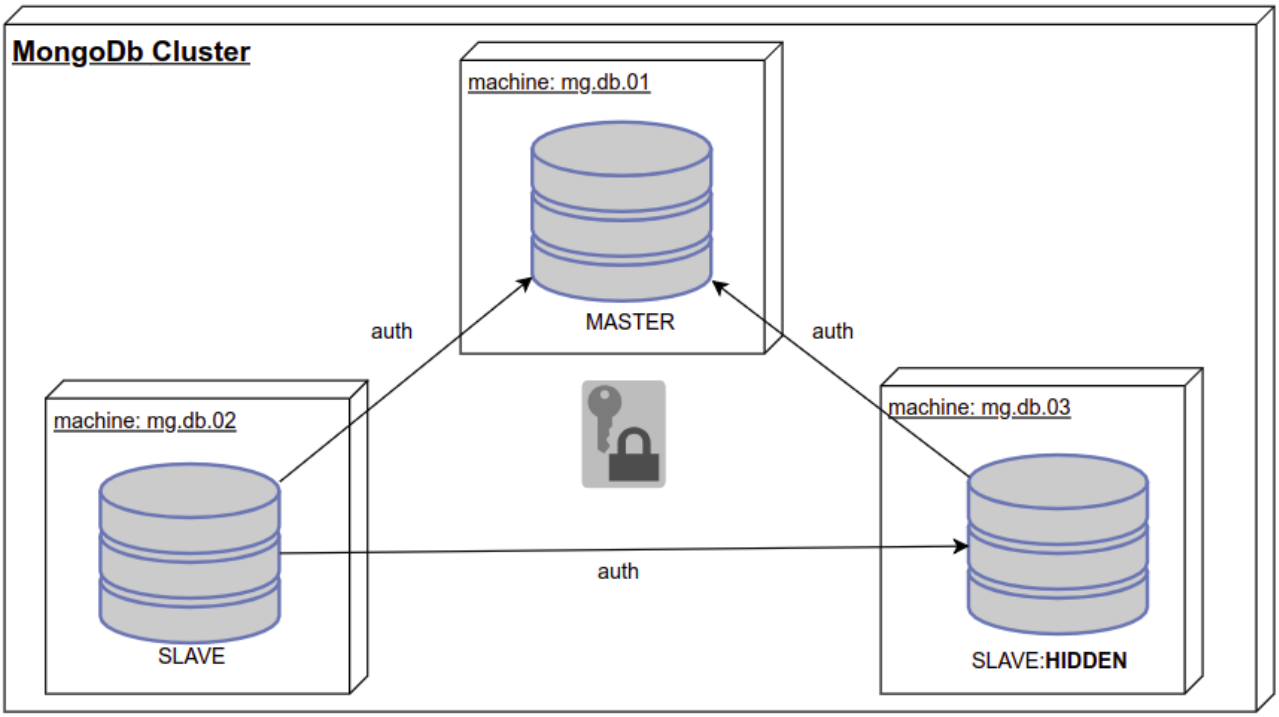
Cluster of mongo databases, is probably best solution when you have more-less uniform data structure in terms of document structure, you need to perform heavy analytics on them, but they are still very related to business layer and potentially another parts of persistent layer.
Here I am going to explain you one of the **mongo** cluster solution configuration, with reason why is so.
Imagine you have an application, which maintain higher load of some sequential data related to each registered users like, comments, posts, messages, etc.

If we use relational model to maintain this state, it would mean that user, would have a posts, messages, related to him, but hold in particular table (posts, messages).
Since user can have as much as he want of this related data, that particular database might grow very fast, making it slow for queries or updates.
Also, members of our BI (business intelligence or analytics) department are doing continuously research of the content of that table. As we mentioned, having examination and analysis of the data on server side, is not most achievable in SQL systems.

We could do some kind of table sharding to address first issue, and perform sharding per username first letter, or geolocation.

Anyway, still we would potentially have non-balanced distribution of data (maybe 60% of data is coming from users with starting letter **G**, or from specific **country**), since we do not have relevant parameters which would bring us balanced distribution of data.
So, we partially solve first issue, but not as we would like to, but it would complicated data analytics, even more then it is currently.

Approach is to introduce a mongodb cluster. First, we would avoid potential bottleneck with SQL database, since vast of communication with potgres (in this case) would be ceased, and at application level we would talk to mongo instead.
We would introduce data distribution balancing as well, in real time, thanking to making decision of persistence tier to communicate with at application level.



We could setup mongo collection threshold for users. For example, every collection containing post data, should have posts of 1000 users.
First we migrate data from postgres into new mongo system, in accordance to this threshold rule.
So having this, we ensure that every collection contains balanced data, and which collection is going to be used by which customer, will be decided at

application level (we would maintain connection between user id from postgres db, and mongo collection where document regarding its posts are held).
Whenever threshold is reached, collection reach 1000 users related documents, we are going to create a new one. Having a lot of collection, is actually not issue for mongo, but contrary (https://docs.mongodb.com/manual/reference/limits/#Number%20of%20Namespaces) .
Having this setup, we also resolve issue of data analysis, since with introduction of document database, we improve that as well.

Mongo cluster I decided is following: Three instances - master slave replication, where one slave is hidden.
Replicaset (which is how mongo clusters is called) members are communicating with heart beats, listening each others. Communication between them is secure, and authenticated with shared public key amongst them.
Master is receiving read/write requests from the main application, and replicate to slaves. One must be aware of read/write configuration, specially regarding **write concern** and **election process**, in the case of **failover of master**.
The reason is, if we have a majority write concern, in this case, if our cluster ends up with one instance, and our write concern at begin was 2, the app will block, since majority of 2 cannot be achieved.
The same issue is if we have an arbiter in architecture, since it holds no data and has some other peculiarities, which should we known at first place, before creating architecture.
Overcoming these obstacles, can be done with correct write concern (master only, nearest node, etc), or having enough of mongo instances, etc.

Reason why we have a hidden member is because, it will be solely used for read from analytic department, and not application itself. Having this, we also have separation of concern from different application at persistence level (application communicate with their db instance, analytic department with their).
As we said, in the case master goes down, non-hidden slave member become a master and continue working as it planned.

Keep in mind that hidden member cannot become a primary, but just to participate in election process and hold data.

It is important to stress that, atomicity in context we used to have in SQL databases is not achievable at database level. Atomicity exists but on document level, in case of document based model, and cannot be spread around multiple aggregated data models (neither in family column based databases as well), so this should be have in mind prior to architectural design.
In SQL databases, transaction can be spread amongst several tables/rows easily controlled by **RDMBS** systems itself, but when it comes up to NoSQL databases, then such a transaction logic, including management of several different document from same or different collections, must be maintained from the application code at application level.
Regarding security aspects, I think that is quite clear that, no database cluster (regardless of which db cluster we are talking about), should be accessible externally but from application only.
Exception of this rule might be administrators access, and that could be guarded for white listing specific range port, but anyway, database cluster should run in its own DMZ (https://en.wikipedia.org/wiki/DMZ_(computing)) zone.

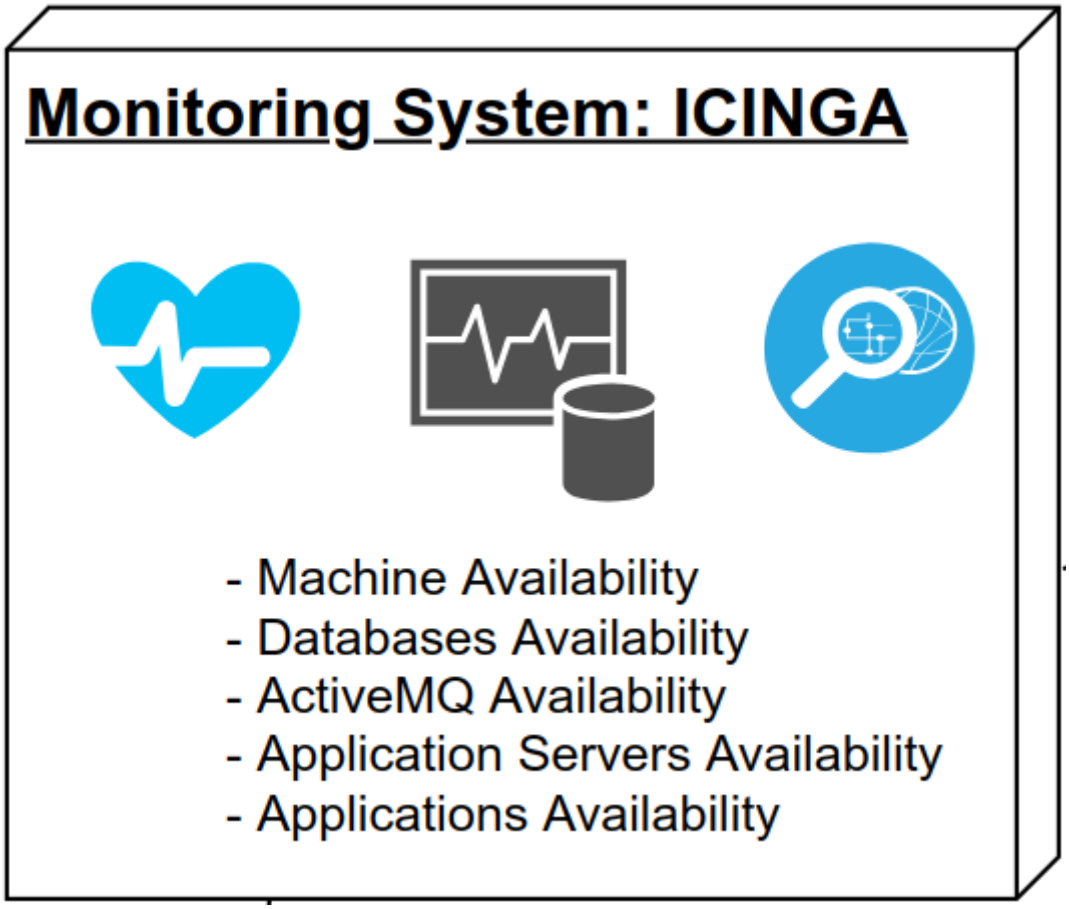
Conclusion

You have seen usage of different database systems in practice, to satisfied concrete needs.
I tried to explained you what players we have in town, when we are talking about solution for persistence layer, and show you how and why successfully different database systems operates together, within application business.
Different database systems, both **SQL** and **NoSQL**, should be used carefully, to address most related data model. There is distinctions between them, their nature, and way of using it.
Mixture of several database platform, in the same application is actually good solution when it comes to enterprise systems, mitigating several obstacles, as I described above.

Monitoring Layer

Last, but not the least, is system monitor implementation. Without this one, no matter what solution for this purpose you are going to bring to, the worse one is the better than the none.
You cannot say that is has direct impact on any service level requirements, but **high availability** and (potentially) **performance**, but implicitly, it can give you an answer how well you performed in other service level requirements.

You need to have some Big Brother of your system, otherwise, how would know when something goes wrong, or how long will it take (cost you) until you figure out, if there is not monitoring and alarming solution for entire system.
In our case, I will talk about monitoring of following application points and solution of choice, related to proposed architecture.



As a solution for this purpose, I choose **Icinga** (https://www.icinga.com/ precisely **icinga 2**) monitoring system software.
It is easily configured GNU licensed system monitoring server, with notification mechanism for reporting system parts outage or network performance issues.
List of monitoring services like network, resources, and the other one, just failure notification and the other icinga components, you can read shortly on https://en.wikipedia.org/wiki/Icinga#Features, or go into documentation of the software for details.
In use case I want to present it, its running on separate machine, and taking care of various aspects and health of the system, depending on the configuration you setup, but here I am going to talk about few.
We are going to talk about several health check of your systems:

- ➔ ***Physical machines health status***
- ➔ ***Application servers health status***
- ➔ ***Application health status***
- ➔ ***Another systems availability like databases and activemq***

Physical machine monitoring

High availability as one of the crucial feature of service level requirements, would not be possible without **redundancy** and **clustering**. Does not matter if **redundancy** is actively participating in business (like application server as member of cluster in any layer), or stand as a **hot stand by** for master **failover** cases (like slave servers in case of activemq clustering, or even load balancing, in our case), high availability must have some, but only.
If one of your server machines goes down for some reason (it goes broken, or shut down for some maintenance reason), we must know that its happened. We have an architecture that going down of single machine will not have impact on application availability, but we must know when and why that happens for several reasons.

First is the reason of happening, where we have to provide an answers for several questions:
Why is happen, was that planned or not. If not, what cause it, and how to overcome that behavior in the future?
Even if we planned it, we still want everybody of interest to be notified about current system status.
Finally, we must focus our attention on this event because, even that currently application availability is not jeopardized because of redundancy we introduce for this purpose (if something goes down), the original state of the system must be restored at certain point, in order to be 100% functional again.
If one load balancer goes down, slave will continue serving requests, but we need to boot our first (former master) load balancer asap, since we want all time provided high availability (what is slave goes down as well – no traffics!)

In icinga, you can easily do (I would say it is basic) setup configuration to monitor particular machine health status (is it up and down), and perform appropriate action when certain events occurs.
You can not only monitor if machine is up or down, which is basics for me as I said, but even entire health status of the machine including CPU load (which process takes most load), memory usage inspection (which application consume over custom setup memory consumption threshold), disk writing (most likely concern) and reading operations, and many other things.

Details of monitoring level is up to you to figure out, what makes sense to go into details and what not.
Monitoring in general is responsible for quality of service, even it is not part of the system business.
Also, notification mechanism and graphical presentation of system health status, is configurable per needs, so once you decide to implement monitoring system, you should sit down and put on the list all these things.

Application servers monitoring

As second point of measurement in your system architecture, comes application servers. You might or not have any another layers like persistence and integration (I am just saying that), but application server you must have in order to run your application itself, at least partially bringing some services if another layers are off.
If application server is off, then nothing is delivered.
Most feasible way of bringing this functionality I to listen for appropriate port, of ip address, where application sever is running (remember, application server is just a software), or to listen some of the server services it provides, to see if they are available and everything works fine.

Level of details you want to monitor, is always up to you (would it include monitoring used resources like CPU or memory, or just ping of the server),.
Notification mechanism could be setup similar to machine monitoring, with appropriate adjustment, related to application server content.

Application monitoring

Every application should have exposed some kind of health checking web service. Everything might looks like fine, the app is deployed at specific port, application server is running nice, but application itself might be blocked, not providing intending services.
We must occasionally ping application for a health, to see is it running or not, awaiting appropriate response back, denoting is everything is fine or not.

Along with solely checking is application running or not, we could (should) introduce even some kind of metrics monitoring, to see how our application is performing.
Health response is good, but is it only what we want? Maybe we should check if positive live response came back within acceptable time range?

Maybe our application should exposed some performance-check delivery web service, where app itself will give us overview how good application is performing?

Spring boot microservices, for example, provides you a lot of built in web services, giving you some details regarding application status, just as well statistics considering application.

Another system monitoring

When I said another, I think on all along application services like databases, jms provider, etc.
Similar to previous three approaches, you must implement some kind of monitoring to those systems as well.
More even, every instance of these platforms should be monitored as well, in order to have clear picture what is going on with your system.
Database failure, will lead , at least, to stateless behavior of your system, or even more, non-workable at all!
On the other side, integration system failure would probably partially has effects on your system, but nevertheless, it is unacceptable level of service requirements.
Overload of non processed messages in ActiveMq, is not situation where you want to stuck with, since it will definitely lead to dirty state of the application.

Conclusion

As you saw, monitoring of your system will lead to healthy state of your system, and maintaining such a state, will lead system to highest level of service level requirements.
Checking heart beat of different layers, different tiers and parts, checking detailed level of healthiness, just as implementation of appropriate notification mechanism and visual presentation of monitored system, is altogether receipt for stability and reliability.
And remember, it is always to have *any* sort of monitoring, but none!

final note

If one person has better and clean understanding of the system architecture in general, I achieved my mission.