imc
part of Scheer

Technical Whitepaper

imc Learning Suite
New system architecture and technologies

im-c.com

Technical Whitepaper

imc Learning Suite
New system architecture and technologies

Author(s): Christoph Gast, Martin Mehlmann
Date: 2021-01-25

Document	Description
Version	ILS 14.8.0 / 14.8.1
Status (Draft / Review / Finalisation)	Finalisation
Contact Person(s)	Christoph Gast

History	Status	Who
2017-11-20	Draft	Christoph Gast
2020-10-02	Review	Christoph Gast
2021-01-25	Review	Christoph Gast, Martin Mehlmann
2021-01-25	Finalisation	Dr. Peter Zönnchen

imc
information multimedia communication AG
Headquarter Saarbrücken
Scheer Tower, Uni-Campus Nord
D-66123 Saarbrücken
T. +49 681 9476-0 | Fax -530
info@im-c.com
im-c.com

Content

1 Introduction

In recent years, Agile development, Cloud hosting, DevOps, Continuous integration and Continuous deployment have become increasingly important. As a result, a particular software architecture called "Microservices" has emerged as particularly suitable for complex web applications. The use of a Microservices architecture makes it possible to fulfill all the important requirements of a modern system such as performance, reliability, security, scalability, and speed of innovation at the same time. For this reason, imc AG decided to develop its Learning Management System (LMS) based on such an architecture starting with the release of version 14.8.0.0.

This document describes the various aspects of this architecture and provides an overview of the technologies used to implement it. The target audience of this document are decision-makers and IT professionals responsible for evaluating and establishing an LMS in their organization. The focus is on demonstrating the advantages of such an architecture from the customer's point of view, without going into too much technical detail.

2 System Architecture

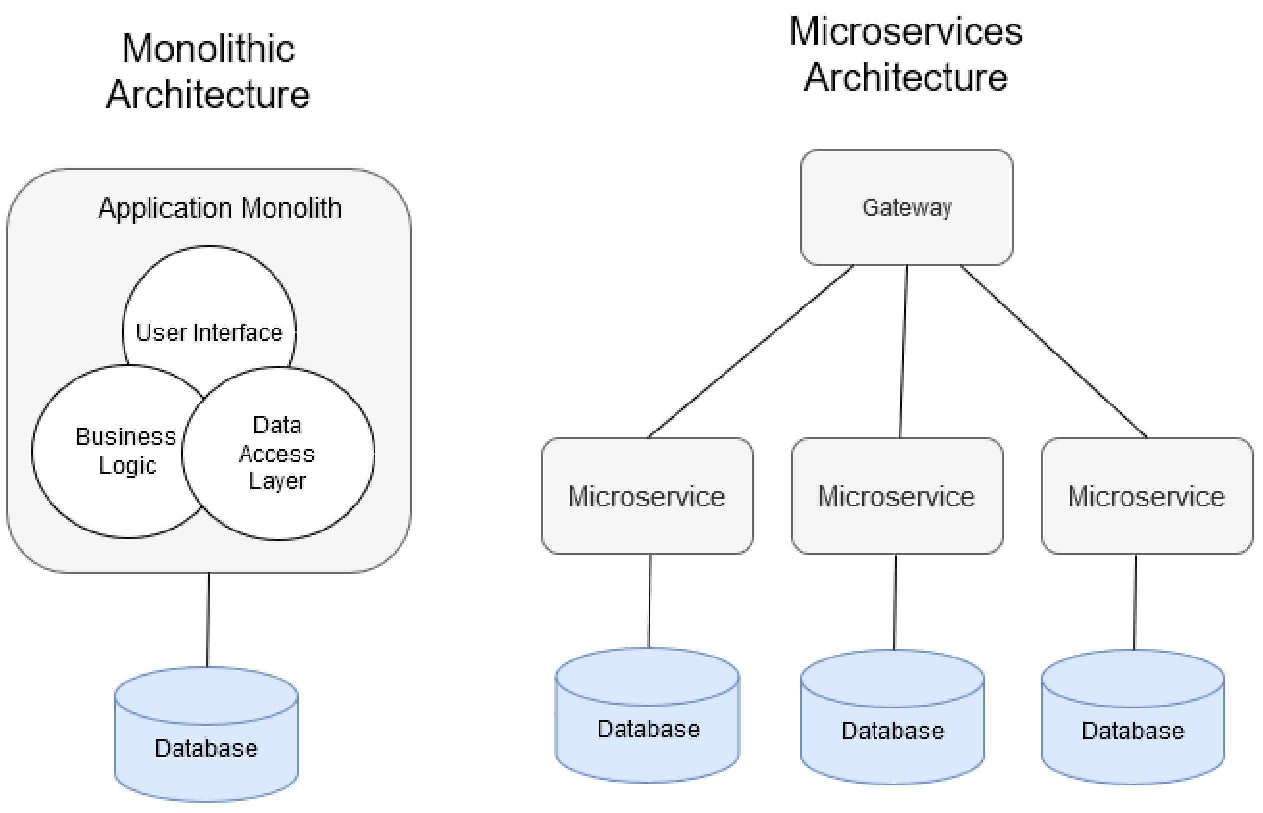

Fig. 2.1: System architecture

2.1 Microservices

Until version 14.8.0.0, the LMS was implemented as a monolithic application. A monolithic application is self-contained and independent of other applications. It provides the entire functionality of the system and performs all required tasks by itself. The advantage of a monolithic application is its independence, meaning it can be deployed as a single unit.

As the size and complexity of the application grows, however, this independence turns out to be a hindrance rather than a help. The program code, that may consist of hundreds of thousands of lines is often located in a single repository and becomes hard to manage. A great discipline is necessary to ensure a modular structure. Even the smallest change to some part of the application requires the whole application to be rebuilt and redeployed. This is error prone, tedious and time consuming. In addition, a monolithic architecture can cause a release to be postponed because implementation on a particular part of the application is not yet complete, whereas all other parts are.

These facts slow down the speed of innovation and lead to a state where the system is becoming more error-prone over time. For this reason, imc AG has decided to change its architecture to a modern microservices architecture. Even if this means a large investment, we are convinced that this will provide our customers with the best possible software solution for their needs.

Microservices are an architecture pattern in which the overall system is composed of a set of independent services that communicate with each other using language-independent programming interfaces. The services are largely decoupled and each service performs a well defined small task. The fundamental idea of a Microservice is about doing one thing and doing that one thing well. In this way, Microservices enable a modular structure of application software. A Microservices architecture yields a number of distinct benefits:

- The complexity of a single service is small and easily manageable. This reduces the probability of implementation errors and ensures high quality software.
- Each service is developed independently of all other services. Since the dependencies to other services are kept small, a service can usually be released as soon as it is completed. This clearly improves speed of innovation.
- A service can be deployed independently. As soon as a new version of a service is available, it can replace an older version of the same service, even in a running system. This leads to faster releases which can be deployed with less or even zero downtime.
- The interfaces of the services are based on proven technologies such as REST and asynchronous message passing. At imc AG, these interfaces are defined by a dedicated team of experienced experts to ensure reliable, high-performance communication and to prevent unwanted dependencies which often arise in monolithic applications.
- Microservices can be scaled automatically, dynamically and independently of each other. By setting up replicas, that is, multiple instances of the same service, it is possible to balance the load and to keep the system robust even if one instance of a service fails. This leads to better performance and fault tolerance.
- By using containers and systems for container orchestration, such as Kubernetes, the usage of resources is automatically adapted to the respective circumstances at all times. This leads to optimal resource utilization and cost savings.
- imc AG mainly serves large customers who have tens of thousands of users. By using microservices, it is now possible to compose a system individually from the available set of services, each of them implementing a specific feature domain. This enables new pricing models, as services can be sold independently and the basic system can be offered at a lower price, making it attractive and affordable for smaller customers as well.

Of course, there are also drawbacks to using a microservices architecture:

- Since microservices are a distributed architecture, they exhibit all the problems that come with distributed architectures in general such as increased complexity, data consistency and the need for sophisticated error handling. Since our software developers are trained accordingly, we are convinced that we can handle the increased complexity safely.
- A microservices architecture generally requires more resources than a monolithic application, especially if containers are used. However, nowadays, hardware is usually no longer the limiting factor. In addition, systems for container orchestration ensure that only the resources that are actually needed are used.
- The deployment of a microservices architecture is more challenging than that of a monolithic application. For this reason, imc AG provides its on-premise customers with deployment packages for different target platforms, which are largely self-contained. The hosting of cloud systems at imc is done by an experienced team of engineers.

2.2 Containerisation

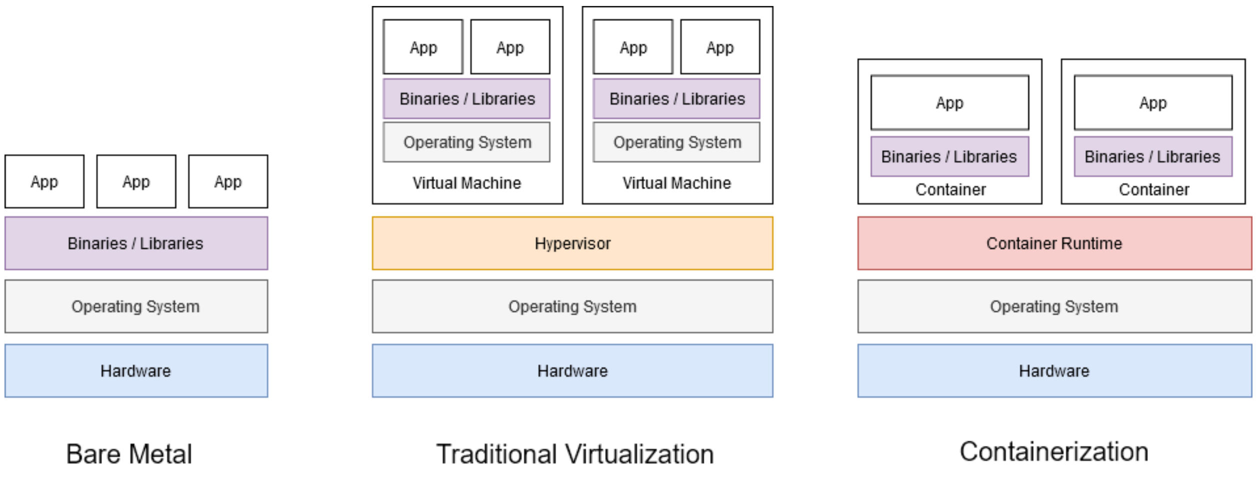

Fig. 2.2: Containerisation

Containerization is a form of operating system virtualization where applications are run in isolated software environments called "containers". A container is essentially a fully packaged computing environment that bundles the application, its dependencies and its configuration in a single "container image". Multiple containers can be run an the same shared operating system using a Containerization software such as Docker.

The container itself is abstracted from the operation system with only limited access to the underlaying resources. As a result, the containerized application can be run an various types of infrastructure, on bare metal, within virtual machines, and in the Cloud - whithout the need to adapt it for each environment.

Since the 14.8.0.0 release, the LMS can be deployed as a set of container images. These container images are started together to form an isolated network of containers where a defined set of ports can be made accessible to the host system. Containers work especially well for a microservices architecture where each service and all it's depencencies are bundled within a single container image.
Containerization provides a lot of benefits:

– Portability between different platform. Docker containers can run nearly anywhere, on virtualized infrastructures as well as on bare metal servers. They can be deployed in the Cloud or on any self hosted machine running Linux or Microsoft Windows.
– Improved security by isolating applications from the host system and from each other.
– Fast and easy install, upgrade, and rollback processes using a container orchestration software like Kubernetes.
– Scalability and replication on container/microservices level. This enables performant and highly available system deployments.
– Flexible routing between services that are natively supported by containerisation platforms.

Due to these advantages, we recommend deploying the LMS in a containerized environment if possible. Of course, we still support the option to deploy the new microservices architecture in a non-containerized environment. Starting with 14.8.0.0, we provide a Microsoft Windows deployment package with each release that is fully self-contained. Besides the WAR files that are required to run the services, it contains a Tomcat servlet container, a Java runtime environment and maintenance scripts to install and maintain the deployment as a set of Microsoft Windows Services.

2.3 Communication

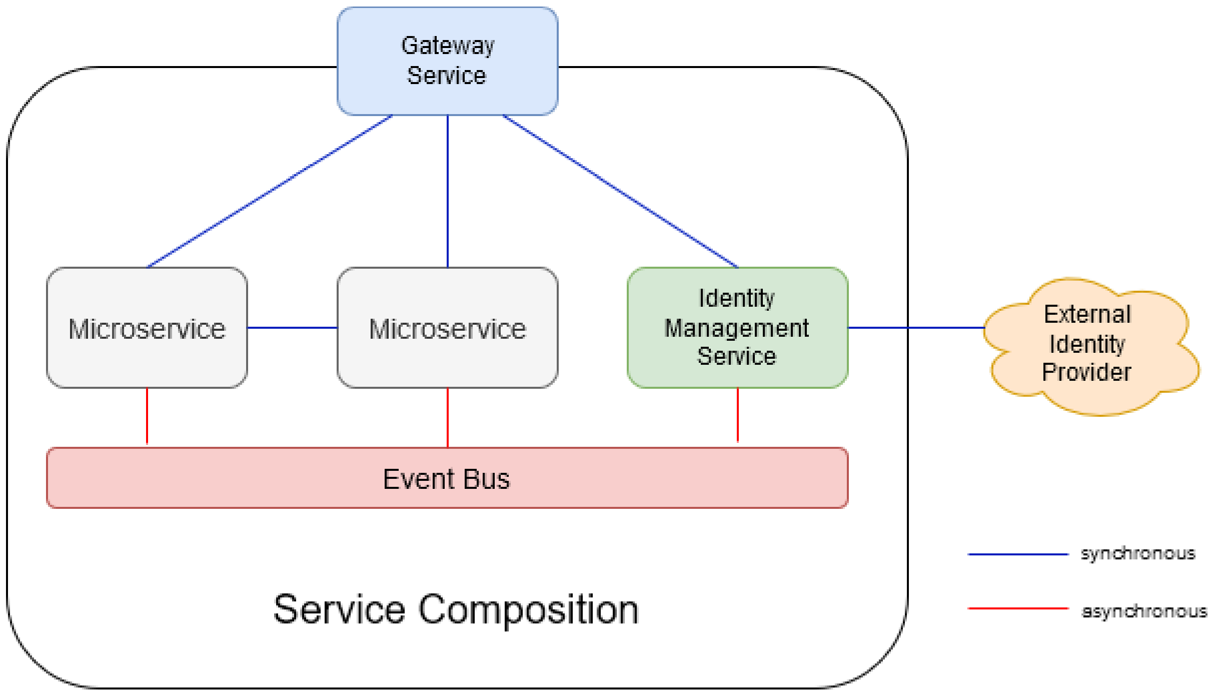

Fig. 2.3: Communication

With regard to communication, we distinguish between external and internal communication. Within the system, microservices communicate either synchronously via HTTP or through asynchronous messages using a message bus. The Event Bus is based on Active MQ Artemis. Whether synchronous or asynchronous communication is used depends on the respective use case.

From the outside, the system can by default only be reached via a single HTTPS port. All incomming request are handled by a dedicated microservice, the Gateway service. For each request, the Gateway will check and verify a JWT that serves to authenticate the user. It will then pass on incomming request to the correspondig service using a set of routing rules thereby acting as a reverse proxy. The Gateway is based on Netflix Zuul.

In case the Gateway receives a request without a valid JWT, it will forward that request to the Identity Management Service (IDM). The IDM supports various authentication methods to authenticate the user. On success, it will issue a JWT containing some basic information about the user as payload. For browser based clients, the JWT will be stored as cookie so that every further request will pass the Gateway and reach the services within the composition. The LMS provides a comprehensive REST API, which is accessible from the outside via the gateway. However, access to most endpoints requires authentication.
Gateway service and IDM are both core services that are part of every deployment.

2.4 Configuration Management

Configuration Management can be split into startup configuration and runtime configuration.

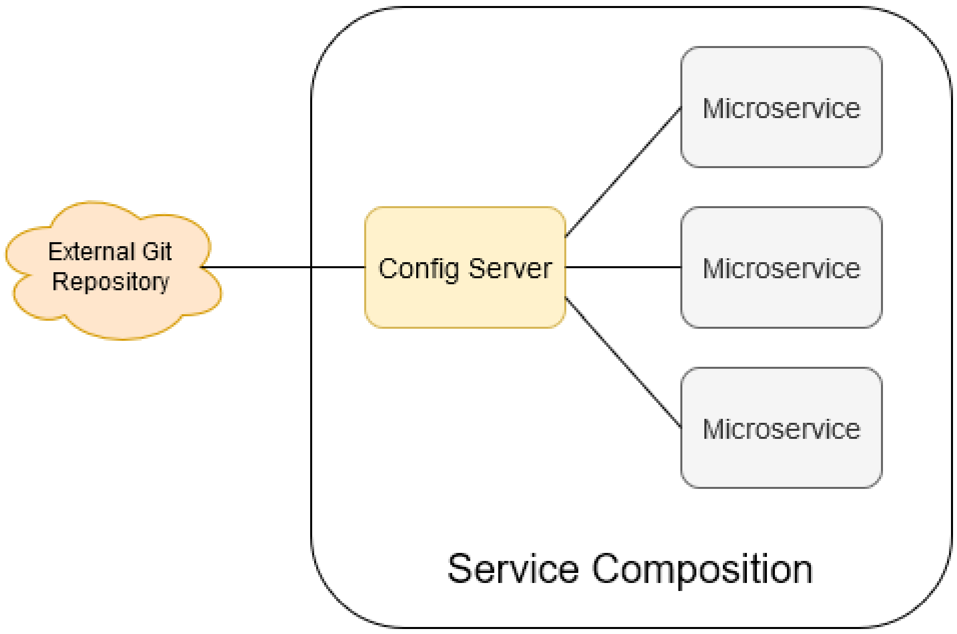

Fig. 2.4: Startup configuration

Startup configuration includes everything necessary for the system to boot up properly in the desired initial system state. This part of the configuration is processed at system startup and does not change at runtime. In case startup configuration must be changed, a system restart is required. Startup configuration is done via configuration files using an external, centralized approach.

External means that the configuration is not part of the build artifacts, but lives outside of them. This separation makes it possible to change the configuration of a service without having to rebuild its binaries.

Centralized means that there is a dedicated service, the Config Server that is part of every deployment and serves the configuration for all other services via a REST API. Please note that this results in a startup dependency. When starting the system, all services wait for the Config Server to be available in order to configure themselves before the startup. The Config Server is based on Spring Cloud Config Server. The actual configuration files may reside in various backends, e.g. in a Git repository, on a web server, or on a local or a mounted file system.
Runtime configuration in contrast includes anything else, that is, all configuration settings that must not be available at system startup. Runtime configuration is done using the UI of the Config Manager in the ILS service. The changes made there at runtime are stored in the ILS database. All other services use an internal REST API endpoint to poll their runtime configuration at regular short intervals in order to apply it.

Please note that the separation into startup and runtime configuration described above is largely but not yet fully implemented. This means that there are still values in configuration files that would be better kept in the database in order to be able to change them without a system restart required. Nevertheless, we will have this split fully implemented in the near future.

2.5 System Deployment

For on promise customers, the LMS can be installed as microservices on different platforms. The best option from our point of view is to deploy in a container environment due to the advantages mentioned above. We also provide deployment packages for installation on a bare metal machine running a Microsoft Windows operation system. We don't provide a deployment package for a bare metal Linux at the moment.

Our continuous pipeline automatically builds artifacts for every service that we offer. This are usually WAR files for Java backend services that are based on Spring boot and can either be run in a Tomcat Servlet Container or as standalone applications. For frontend services, the artifacts are minimized compressed archives containing JavaScript sources and other assets, that can be extracted on a simple webserver. Besides the artifacts, the pipeline also builds Docker images for every service. They are hosted in our internal Docker registry and will be pushed to AWS ECR or any other Container registry in case of a new release to make them available to customers.

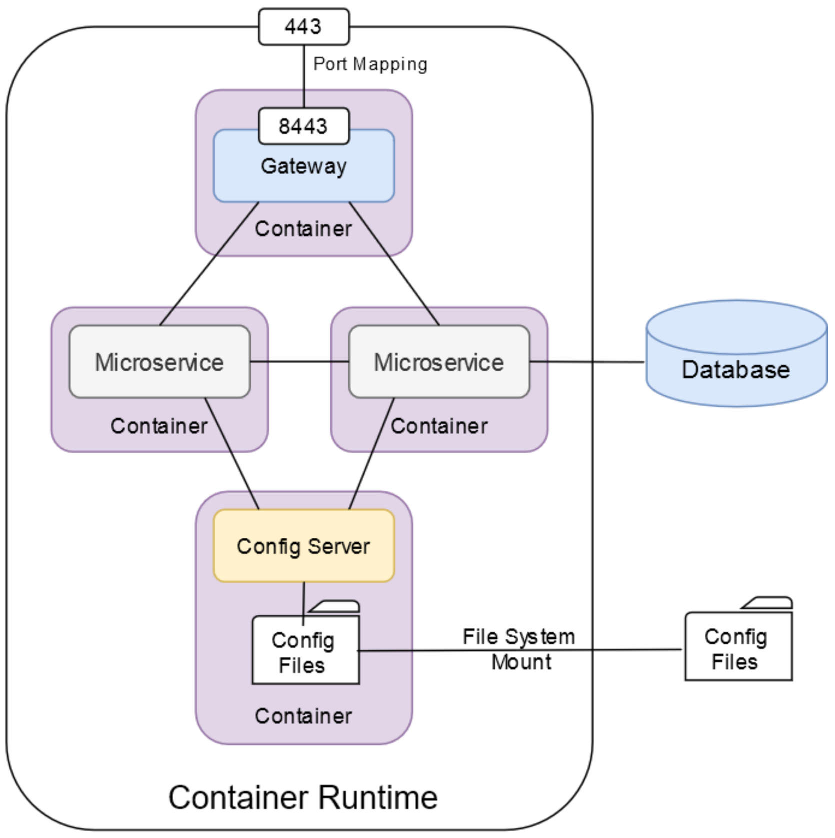

Fig. 2.5: Container Deployment

For a containerized deployment, we provide Docker images in AWS ECR customer specific namespaces. Please note that there may also be multiple images for a single service, e.g. for backend, frontend and even database. These images can be pulled by a customer after successfull login. In addition, we provide the startup configuration files as a compressed archive together with a docker-compose.yml file that shows how to the services are composed together. Of course, detailed documentation is also included in the package.

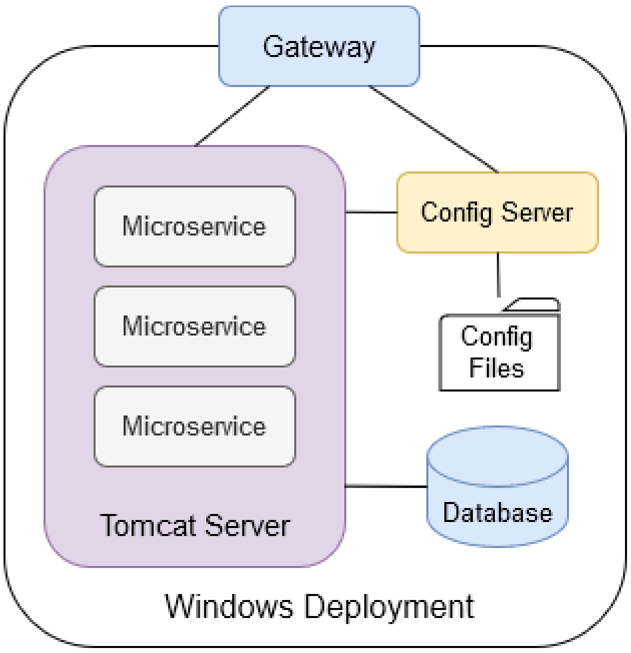

Fig. 2.6: Windows Deployment

For bare metal Microsoft Windows deployments, we provide a package that contains everything to install and maintain the system. This includes a hardened Tomcat servlet container, artifacts of all services, a Java runtime environment, configuration files, maintenance scripts and detailled documentation. Most of the service artifacts are deployed in Tomcat, however there are two exceptions, the Gateway and the Config Server are provided as standalone spring boot applications. Tomcat, Gateway and Config Server are registered as Microsoft Windows services during installation of the system to enable automated startup on Microsoft Windows startup and the option to stop and start these services via the Microsoft windows services manager.
For cloud customers we offer a variety of deployment options. Please contact our hosting experts so that they can assess your requirements and find the best possible solution for you. In particular, deployment on Kubenetes allows a variety of options for customizing and fine-tuning.

2.6 Scalability and Load balancing

When it comes to load balancing and fail safe operation of a containerized deployment, Kubernetes comes with multiple ways how to achieve that, for instance by setting up replicas of containers and by using the Kubenetes Ingress controller. In addition to Kubernetes built-in load ballancing, many cloud providers offer container load balancing services with provider-dependent capabilities. We refer the reader to the documentation of the container orchestration platform and the cloud provider for more information. In general, for containerized deployments, there is wide range of load balancing solutions available on the market. The main advantage of a containerized deployment with respect to load ballancing is the fact, that it is possible to scale individual containers and therefore services dynamically and automatically. That leads to higly performant systems that adapt themselfes according to ressource needs.

For a Windows deployment, scaling of individual services, that run as contexts in Tomcat servlet container is not supported. Instead, the system can only be scaled by setting up replicas of the whole system. That is, two deployments are run in parallel and a server that supports load ballancing, e.g. Microsoft IIS is put in front of them. This form of scaling is fixed and does not adapt itself as long as there is no additional scaling software involved.

There are some peculiarities that need to be considered if the LMS is to be operated in a cluster mode with multiple replicas:

- The LMS is not stateless and uses web sessions to identify the current user. Due to that, Session affinity needs to be enabled on the Load balancer.
- The LMS does not store files as binary blobs in the database but as regular files in a dedicated "data" directory in the file system instead. This directory must be shared among all replicas by using networks drives for bare metal Microsoft Windows deployments or shared cloud file storage services, like Azure Files or AWS EFS for containerized deployments.

2.7 Centralized logging

For bare metal Microsoft Windows deployments, all logfiles can be found in a subfolder of the installation folder.

For containerized deployments, logging happens within the individual containers. As soon as a container is undeployed e.g. by the orchestration software, logfiles for that container are no longer available. To enable persisitency for logfiles, our architecture offers the possibility to collect the console output and all log files of every service in a central location. This task is performed by a dedicated service, the Logging Service. The Logging service includes an Elasticsearch instance for log file aggregation and a Kibana instance for logfile visualisation and analysis. Together with Logstash, which is a small application that is part of every service image, centralized logging is realized.