OPAL
OPEN DATA PORTAL

Deliverable D8.1
Portalinfrastruktur

Autoren: Matthias Wauer, Afshin Amini, Adrian Wilke

Veröffentlichung	Vertraulich
Fälligkeitsdatum	31.12.2018
Fertigstellung	07.02.2019
Arbeitspaket	AP8
Typ	Bericht
Status	Final
Version	1.0

Kurzfassung:

Dieses Deliverable berichtet über die Auswahl der Plattformwerkzeuge, die für die Umsetzung des OPAL-Portals genutzt werden. Ausgehend von infrastrukturspezifischen Anforderungen werden die Vor- und Nachteile von möglichen Lösungsansätzen betrachtet. Daraus werden Entscheidungen für eine geeignete Plattforminfrastruktur abgeleitet.

Schlagworte:

Datenportal, Infrastruktur, Server

mFUND
Das Startkapital für die Mobilität 4.0

Bundesministerium für Verkehr und digitale Infrastruktur

Inhalt

1 Introduction

In the scope of the OPAL project we are developing a prototypical Open Data portal. In this task, we discuss different options for providing the necessary infrastructure for providing the portal. Based on the architecture developed in work package 1, we define and develop platform tools for creating and managing the necessary portal services. This includes aspects such as deployment, monitoring, configuration management and platform security.

In this first deliverable, we primarily discuss the requirements and potential solutions to the portal infrastructure management.

2 Infrastructure Requirements

Looking at the consolidated requirements in D1.1, there is only one item addressing the infrastructure:

- AK5 “Entwicklung eines komponentenbasierten Metadatenportals”: Since the metadata portal development should be component-based, the infrastructure must be suitable for managing and deploying such a component-based solution.

This is to be expected, since these requirements are primarily concerning the functional requirements of the applications and the portal’s data processing. Hence, further requirements regarding the portal infrastructure will be extracted from the project proposal and during the process of defining the infrastructure.

2.1 Requirements from the project proposal

The following requirements have been extracted from the work package 8 description in the DoW.

1. Web portal based on CKAN and its plugins
2. Additional components developed independently (like in the European Data Portal)
3. High degree of reliability and resiliency against peak load and attacks
4. Microservice based approach with adaptive deployment of services (on local servers or using cloud services)
5. Flexible automated deployment and management

2.2 Requirements identified during the prototypical infrastructure design

In addition to these previously defined requirements, the following list summarizes those requirements that have been identified during the process of setting up the prototypical infrastructure:

6. Interfaces between the components should be based on Semantic Web standards as far as possible.
7. Infrastructure technology can be used during development and runtime of the portal.

3 Requirements analysis

In this section, we will discuss the different requirements, potential solutions to them, and suggest an appropriate approach.

3.1 Web portal

There are very many options for implementing a Web application. Classic approaches include static pages hosted on a Web server, dynamic Web pages generated by the server (e.g., by processing HTML form input), and dynamic Web pages generated by client-side code. Static web pages have largely been replaced due to their limited functionality.

There are many different frameworks and technologies for implementing dynamic Web applications. Examples include Spring MVC, Django, Rails, and express.js. Most of them support different options for either server-side or client-side rendering. The primary difference between them is that server-side rendering generates the view (typically HTML) that is displayed at the client, while for client-side rendering it provides an API (typically REST), as well as mostly static view (HTML), styling (CSS) and client-side application code (JavaScript). For the latter, again there are many different options to choose from, including Angular, Vue.js and ReactJS. Finally, there are hybrids between these approaches, i.e., server-side rendered pages with elements generated on the client side.

There are different tradeoffs between these options. While client-side rendering can improve scalability (less load on the server), it can be an issue for mobile applications because some JavaScript frameworks pose higher system requirements on the rendering device. Additionally, search engine optimization (SEO) is an issue with completely client-side rendered Web pages, as currently only Google executes this code in order to index such Web pages (and it does only in a second processing attempt).

For the generic portal infrastructure in OPAL, these issues are less of a concern as long as the development, deployment and monitoring of the respective Web applications can be provided. However, based on the platform architecture, the Web portal should be built in a two-layer approach, with separated components for the different APIs and user interfaces. Therefore, a **server-side API** in addition to mostly **client-side rendering in the application layer** is suggested. We also suggest to use a framework building on **existing well-known standards** (HTML, CSS, JavaScript, Web Components) with **high performance / low memory footprint** (considering the mobile development) and is **community-driven** for less vendor lock-in, such as *Vue.js* or, with some limitations, *ReactJS*.

Note that there is also the approach of serverless Web applications, which separate application functionalities from actually hosting the application. However, this approach is limited, as the hosting is only provided by a certain Cloud service provide. This does conflict with the requirement discussed in Section 3.4.

3.2 Additional components developed independently

In the OPAL Description of Work, we have stated OPAL should extend existing solutions, such as CKAN, with additional components for the required functionality. Most of the components defined in the OPAL architecture (D1.3) are already defined in separate layers and, as such, will be implemented separately from CKAN and its extensions for OPAL. This is particularly necessary for reused components like LIMES, which are implemented using different tooling (e.g., Java instead of Python).

3.3 High degree of reliability and resiliency against peak load and attacks

Achieving high reliability of a Web portal requires a set of measures to be taken into account. For all critical system components (i.e., networking, data storage, and all user-interfacing components like Web user interfaces and APIs), the infrastructure must include load balancing and scaling of the respective components. This might include clustering of the triple store, using caches, or even hosting the services in two separate computing centers.
However, all of these measures have drawbacks:

- The system architecture is complicated by additional components, such as a load balancer.
- Higher requirements and cost of required computing hardware.
- Additional network traffic can occur, e.g., because of inter-node traffic in clusters of message queues or RDF stores, which might also degrade performance.
- Non-trivial testing, monitoring and bug-fixing of, e.g., auto-scaling mechanisms.

While some of these aspects have to be considered separately for each specific components, we suggest the following general guidelines for OPAL:

- Selection and use of storage and messaging components that support at least master-slave **clustering**, preferably with **declarative discovery**-based cluster formation.
- Processing components have to be provided as **containers** which can be used as services in, e.g., Docker Swarm for **horizontal scalability** (see Section 3.4).
- Clear specification of interfaces which conform to sound **architectural styles** (such as REST principles for HTTP, or similar architectural styles for, e.g., messaging), in order to enable effective **caching etc**.
- Implementation of **resiliency** mechanisms, such as circuit breaker or rate limiter, using libraries like Hystrix or Resilience4j.

3.4 Microservice based approach with adaptive deployment of services

The use of (micro-)services has become a trend in recent years. Compared to earlier methods developing monolithic software, there are several benefits to composing software using components separated by their individual concerns, including:

- Improved maintainability of the different components
- Loose coupling, which makes it easier to replace individual parts of the software
- Independent deployability, which enables individual development and bugfixing of certain services
- Organization around business capabilities, so the software architecture is driven by use cases rather than technical concerns

Drawbacks of this approach include increased complexity in testing and deployment, as well as higher memory consumption. Furthermore, it can be difficult to decide how to decompose an application into services. This particularly applies to research projects like OPAL. However, the improved flexibility should help with rapid development of the OPAL portal. Therefore, we suggest to **first develop services according to the architecture** (see Deliverable D1.3), which can then be further **decomposed as needed**. At first, these services might be relatively large, such as the interlinking framework, but premature decomposition would distract from the actual goal.

3.5 Flexible automated deployment and management

Related to the resiliency discussed in Section 3.3, the portal infrastructure is concerned with deploying all components required for the OPAL portal, and managing the deployed environments. There are many benefits to automating this deployment task, including faster deployment of new portal environments and less risk of error in the process. With the increasing adoption of microservices, there are also many tools available for the implementation of continuous integration / continuous delivery (CI/CD) workflows. In such a workflow, component updates are released often, including the executing of unit and integration tests, packaging the software artifacts, and publishing them in repositories. They can then be used in application environments (pipelines) for development, staging, and production.

When it comes to rolling out such pipelines, again, there are several technologies available to support this task. The most basic approach is the use of shell scripts, which only targets part of the issue and can get increasingly complicated with larger application environments. Tools like Chef and Puppet can help by providing a repository of such scripts, but these are still non-trivial to maintain successfully. Again, containers can help by providing a common concept to managing different components. Also, there are many mature tools that help with the deployment of container compositions, such as Docker Compose, and managing scaling, such as Docker Swarm.

In recent years, Kubernetes has become a preferred approach to managing more complex computing environments. This includes solutions to many issues, such as configuration management, load balancing, optimizing workloads on cluster nodes, and self-healing. While this can be very helpful for complex applications in production environments, the primary drawback is a steep learning curve of Kubernetes concepts and terms.

Therefore, we suggest the following approach: OPAL should use a **CI/CD approach** to developing the services of the OPAL portal. All components should be made available as **Docker images**, which can easily be integrated in a Maven **build process**. These images should be published on **Docker Hub**. The prototypes (entire portal compositions or subsets, such as a CKAN environment with the respective dependencies) should be defined declaratively with **Docker Compose**. With these artifacts available, more comprehensive solutions can then be **applied if necessary (Docker Swarm, Kubernetes)**.[1]

On this foundation, it would also be possible to manage complex deployments, spanning different cloud and on premises infrastructures. This enables a much higher flexibility regarding future potential deployment requirements for the OPAL portal and components. For example, Figure 1 shows which components are required to manage an application environment distributed over different infrastructures.

[1] https://kubernetes.io/docs/tasks/configure-pod-container/translate-compose-kubernetes/

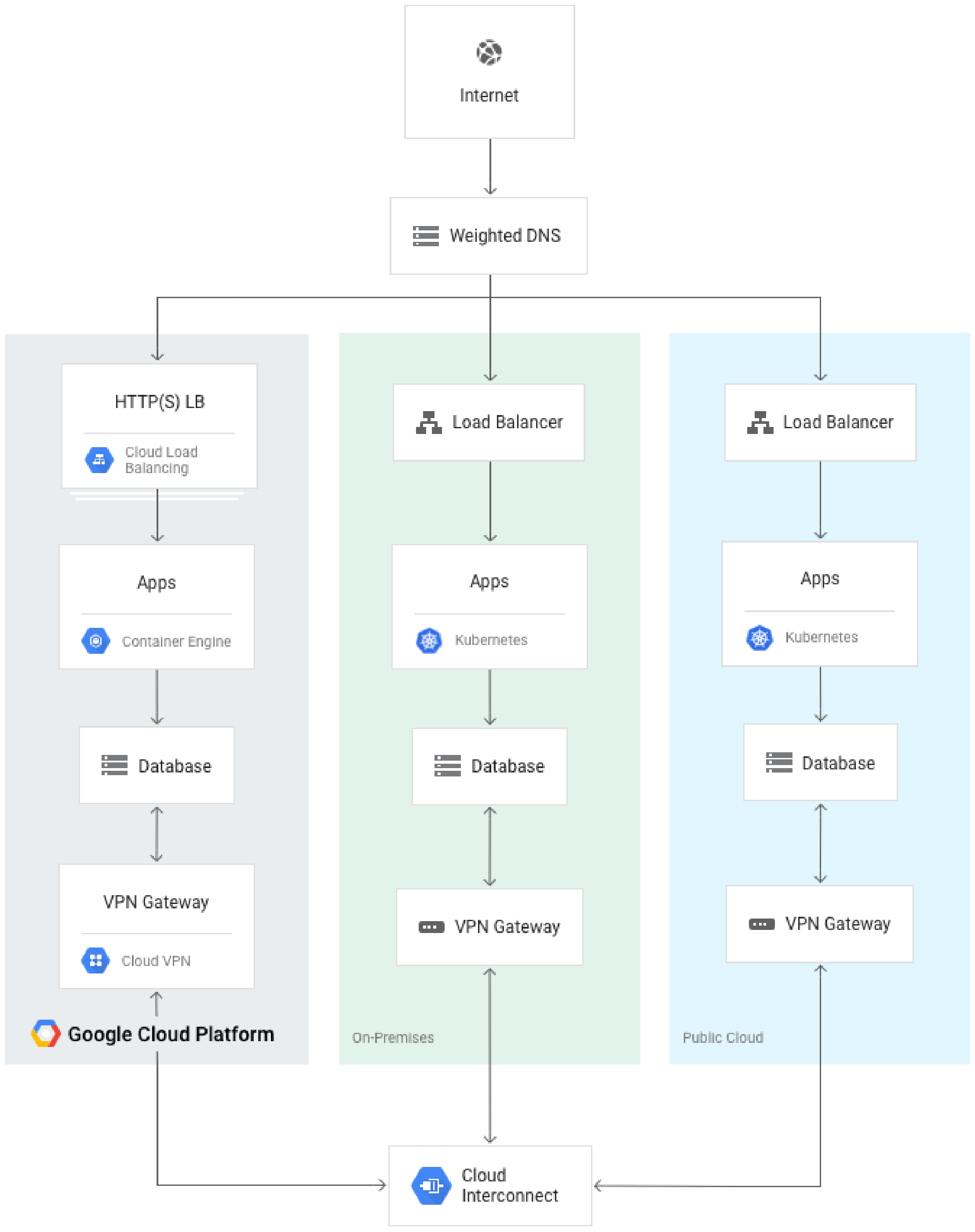

Figure 1: Example of a hybrid deployment approach, Source: Google Cloud

3.6 Interfaces between the components based on Semantic Web standards

Semantic Web standards have certain drawbacks, including a verbose representation of data (when comparing typical serialisations like Turtle with, e.g., ProtoBuf), certain limitations of query language and query optimisation, as well as somewhat intimidating logical foundations. However, clear semantics are required for communication between components. Explicit semantics can help provide a trustworthy interface between different components in the OPAL portal. Therefore, mature **Semantic Web standards** like RDF and SPARQL should be applied for the interfaces between different components in the OPAL portal, as long as the respective data is already available in a respective representation. For the implementation of these interfaces, existing mature **frameworks** like Jena and RDF4j should be used.

3.7 Infrastructure technology can be used during development and runtime

The tooling that has been discussed above, namely Docker containers and the respective infrastructure extensions, can be successfully used in both development and runtime environments. The primary advantage is that existing service images can be distributed via a repository, so they are readily available and can be executed in a very simple way. This helps with quickly setting up development and runtime environments in a reproducible way on different machines, as discussed in the Docker Reference Architecture.[2] In order to streamline the development process, we suggest to use existing mature libraries and tools, such as Spring Cloud.

4 Infrastructure Design

In this section, we summarize the findings of the previous requirements analysis section in order to define the portal infrastructure for OPAL.

1. Computing infrastructure
    a. One or more server nodes in an on-premises *or* cloud environment
    b. Docker engine available, node(s) registered in Docker Swarm

2. Service component development
    a. Agile development, with gradual application of CI/CD workflow
    b. Mandatory automated containerization for each component
    c. Service granularity as defined in D1.3, gradual microservice decomposition as needed[3]
    d. Interfaces based on Semantic Web standards (RDF, SPARQL, common serialisations)
    e. Frontend development using lightweight, high-performance client-side rendering; server-side API

3. Application deployment
    a. Docker Compose for building the complete prototype application from components
    b. Docker Swarm for horizontal scaling of Docker-based services
    c. Potential extensions using Kubernetes (not mandatory, as needed)

4. Non-functional requirements
    a. Resilience measures applied, using existing frameworks like Hystrix/Resilience4J
    b. Interfaces based on architectural styles like REST to enable/benefit from caching etc.

Figure 1 shows a high-level overview of the OPAL portal's infrastructure design. It primarily focuses on the composition of components and their deployment, while details (such as resilience and frontend details) have been omitted.

[2] https://success.docker.com/article/dev-pipeline
[3] https://microservices.io/patterns/microservices.html

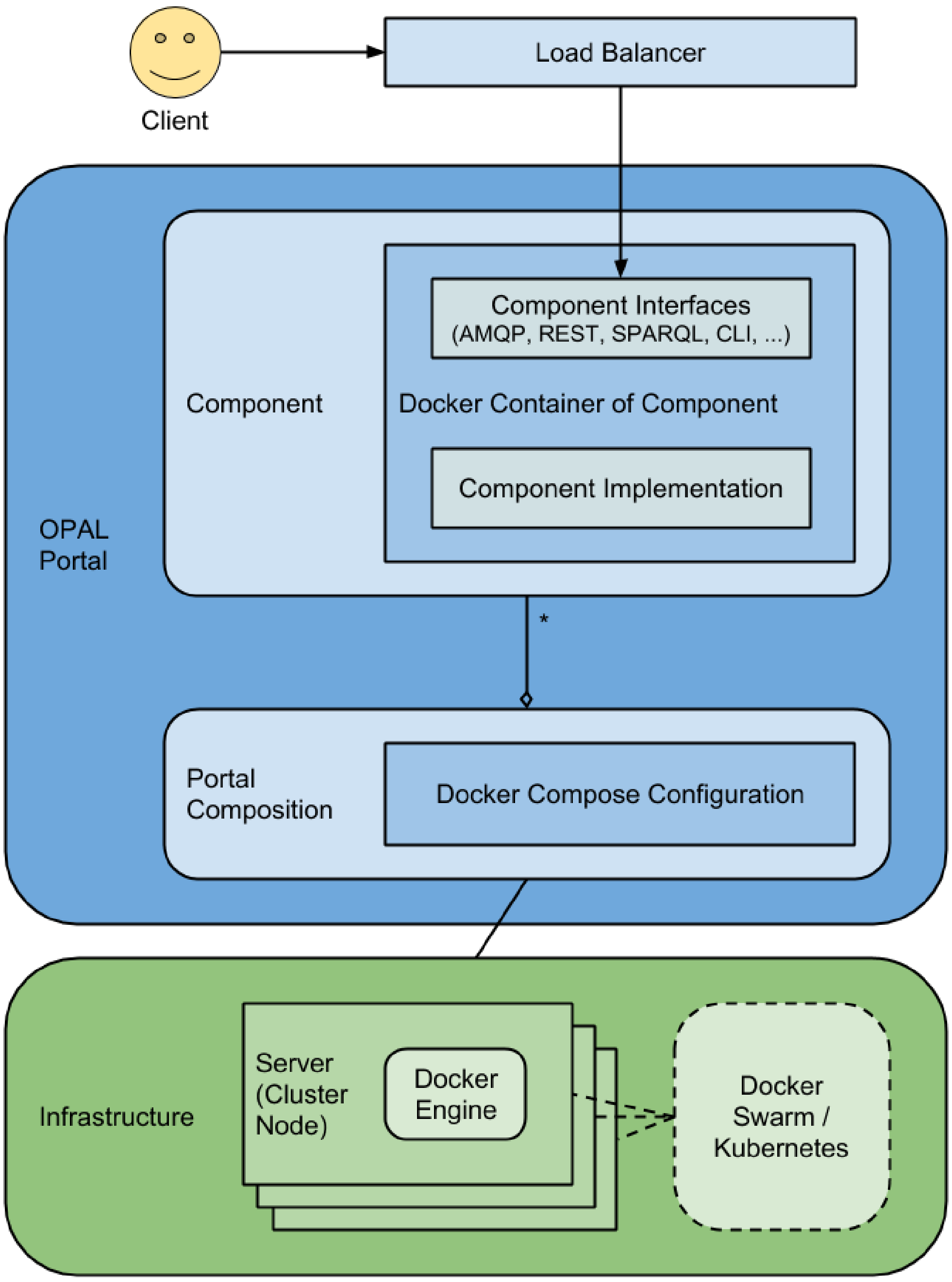

Figure 1: OPAL portal infrastructure design

5 Conclusions

In this deliverable we have specified the portal infrastructure for the development of the OPAL portal. Based on a analysis of requirements from work package 1, the description of work, and prior experiences, we have defined guidelines for the infrastructure design. This includes suggestions for the computing infrastructure, service component development, application deployment, and further aspects regarding non-functional requirements.