A Developer Portal for DevOps Environment

Niklas Sänger, Stefan Throner,
Simon Hanselmann, Michael Schneider, Sebastian Abeck
Research Group Cooperation & Management
Karlsruhe Institute of Technology (KIT)
Zirkel 2, 76131 Karlsruhe, Germany
email: (niklas.saenger | stefan.throner | michael.schneider | sebastian.abeck)@kit.edu

***Abstract*—A good microservice architecture divides a complex system into separate microservices, which can then be reused in several applications. To achieve efficient reuse of components, it is necessary to provide the developers with the required information on how they can use the running microservices and their application programming interfaces (APIs). A tool for this kind of information is the developer portal, which enables the administration of the interfaces and documentation of individual microservices and makes them available to all developers on a central platform. The diversity that arises from multiple teams and different environments makes it difficult to manage the information in the developer portal manually and in a central location. In this paper, we describe the development of a developer portal and focus on the following aspects: (i) designing a domain that represents a service environment, (ii) requirements for the developer portal to support the developer workflow, (iii) environment-agnostic approach for automated data gathering for the developer portal, (iv) microservice monitoring during runtime.**

Keywords*—*DevOps; microservices; api; development; developer portal.

I. Introduction

Agile methods and microservices are the main concepts to deal with today's complexity of modern software systems [1]. This is achieved by splitting a monolithic system into microservices [2], allowing to reduce the overall complexity of single components. Additionally, a good microservice architecture, designed according to the principles of Domain-Driven Design (DDD), results in reusable microservices which can be consumed by other microservices and applications [3]. Due to the use of APIs and the separation of functionality, microservices can be developed and maintained by small individual development teams as it is common in agile development. To enable efficient reuse of microservices and their APIs, the microservice information and API specifications have to be accessible to all developers. One solution for APIs is a developer portal [4].

A developer portal allows the provisioning of API specifications and documentation of the corresponding services and provides access to the data in a central spot. However, this requires that developers provide the data and publish them to the developer portal manually. While this approach is suitable for the classical use of the developer portal, the offering and marketing of APIs, it can lead to problems when the data (e.g., API specification) should be provided continuously in larger projects, with multiple distributed teams during the development process. Since agile methods aim to deliver new features and bug fixes on a daily basis, manually updating data can lead to an overhead for the developer, potential data inconsistencies (e.g., different API specifications), and distributed knowledge (i.e., teams have a different understanding of an API specification).

To overcome these problems, we propose an automated solution for a developer portal, which uses a development and operations (DevOps) approach to automate the process of service registration and lifecycle management of API specifications and service documentation, thus providing a continuous source of truth. The automated provision of data is supported by a template-based pipeline approach [5], which allows central adaptation and extension of the Continuous Integration / Continuous Deployment (CI/CD) pipelines.

The requirements on the developer portal are formally specified and implemented by applying a microservice engineering process. This results in a microservice-based application, called MicroserviceDeveloperPortal (MDP) and a business domain called ServiceEnvironment. In the context of this application, microservices play two different roles, which must be distinguished: on the one hand, the MDP application provides support to develop microservices, and on the other hand, the MDP itself is built as a microservice-based application.

This present article is structured as follows: Section II presents the state-of-the-art in microservices and DevOps. Section III provides our approach to the domain and application engineering. Section IV describes the microservice registration process. Section V introduces our health monitoring approach for microservices. Section VI describes the use of the MDP in our environment. Lastly, Section VII summarizes the main results of our approach and future work.

II. Related Work

The term microservice goes back to the year 2011. A group of software architects chose it as an appropriate name in 2012. Among them, Lewis was the first researcher using the term in a presentation [6]. Three years later, Lewis and Fowler published the first description of microservices and the microservice architecture style [1]. Based on this work, Newman published a well accepted and often cited book on microservices [7].

A strong motivation for the microservice architecture style is provided by the disadvantages of a monolithic software architecture in regard of change cycles and complex deployments [2]. This is because even small changes require a complete rebuild and redeploy of the monolith. This aspect is especially important for companies wanting to reduce the time-to-market or improve robustness of the provided services [2].

A microservice architecture reduces the complexity for each service but increases the overall amount of microservices [8]. When compared to a monolithic application, the urgent need for a service orchestration arises. A common solution for this problem is to use Kubernetes which is a widely used container orchestration system developed by Google [9] [10]. While Kubernetes can solve the orchestration of microservices, Kubernetes is inherently complex and introduces its own tools and paradigms [11].

The complexity of microservice architectures requires additional communication and contracts among development and operations teams. To overcome these obstacles, DevOps principles can be used. While there is no general definition of DevOps, various authors tried to define the term. Lwakatare et al. [12] perform a study to define the term DevOps and come up with five dimensions: collaboration, automation, culture, monitoring, and measurement. Erich et al. [13] find that companies have different definitions on DevOps but share a common understanding that team, culture, and automation are key aspects.

Moreover, Wilsenach [14] stated that the DevOps culture should have no silos between Dev and Ops. This is where agile methods, such as a CI/CD pipeline can be efficiently used [8]. It allows for cross-functional teams which are responsible for the development, deployment, and operation of their microservice. One aspect of this work is to use DevOps for the automation of data provisioning which in turn ensures up to date data for a developer.

In general, a developer portal must provide the developer with all necessary information required for the development of an application. De [4] refers to a developer portal in the context of API management which is a common use-case for a developer portal. It supports a developer by providing API documentation, API access, or API analytics information. The counterpart of a developer portal is the provider portal in which a service developer provides information (e.g., API specifications, API version) of a service. Generally, the API specifications Examples can be found in the developer portals by Amazon Web Services (AWS) [15] or Mercedes-Benz [16]. Existing solutions, such as the developer portal offered by AWS, work best with products of the same vendor, creating a vendor lock-in effect. They also neglect information about the backing services (i.e., microservices) and require manual data upload. We propose a microservice-based solution, which can be deployed in an arbitrary Kubernetes environment able of running Docker containers. Moreover, a CI/CD pipeline is used to automate the provisioning of data including the state of microservices running in a cluster.

The microservice engineering process we have applied to systematically implement the requirements on a developer portal into a microservice-based application is based on the Domain-Driven Design (DDD) from Eric Evans [3] and has been described by us in several publications. In [17], the process is introduced and applied to the domain of connected cars to provide a solution for the charging of electric cars (i); [18] describes the microservice architecture based on the domain-driven design concept of a context map in detail (ii); and [19] shows how we use the profiles of the Unified Modeling Language (UML) to formally specify the domain model by different types of UML diagrams (iii).

III. Domain and Application Engineering

The development of the MDP follows a previously mentioned microservice engineering process. Therefore, the first step is to develop a suitable domain in which the application is located. The difficulty lies in placing the components needed for a microservice environment in the domain.

A. ServiceEnvironment Domain

Since the MDP touches a variety of aspects related to microservices, DevOps, and container orchestration, we decided to name the domain ServiceEnvironment. The ServiceEnvironment domain should further include everything that is required for the deployment and operation (e.g., orchestration and monitoring) of microservices.

After setting the scope of the domain ServiceEnvironment, the engineering process requires a definition of a ubiquitous language and a context map. The ubiquitous language defines terms that are relevant (i.e., because they are frequently used) for the domain and its applications. This ensures that developers have a common understanding of the terms throughout the engineering process. Hence, the ubiquitous language must follow a previously fixed set of guidelines in order to be consistent. In case of the ServiceEnvironment domain, an extract of the ubiquitous languages defining terms such as Service, HealthState or Observation can be found in Table I.

TABLE I
EXCERPT OF THE UBIQUITOUS LANGUAGE OF THE DOMAIN SERVICEENVIRONMENT.

Term	**Definition**
HealthState	Data which describes the availability state of a service (e.g., latency, liveness).
Observation	Part of the domain that observes services after the deployment
Service	Part of the domain that represents a deployed application that can be accessed by the user

The next artifact for the ServiceEnvironment domain is a context map that defines the domain, its subdomains, and bounded contexts. The development of the artifact is motivated by DDD. Figure 1 illustrates the context map of the ServiceEnvironment domain, which includes the subdomains Observation, MicroserviceManagement, and Deployment. Each subdomain provides a bounded context that the MDP can interact with. To reduce the overall complexity of the ServiceEnvironment, the domain currently only includes subdomains that the MDP requires. If the functionalities of the MDP would be extended or another application would require additional data from the domain, the ServiceEnviornment should be extended accordingly.

Typically, bounded contexts are implemented as domain microservices. This means that they must have no dependencies on other services. We decided that a domain microservice should only provide Create, Read, Update, and Delete (CRUD) operations to handle data requests (e.g., monitoring data) and not contain any application logic. This makes the microservices, and thus the data, reusable for multiple applications. Moreover, it allows the application microservices to be stateless since the data is stored within the domain microservices following the 12 Factors App paradigm [20].

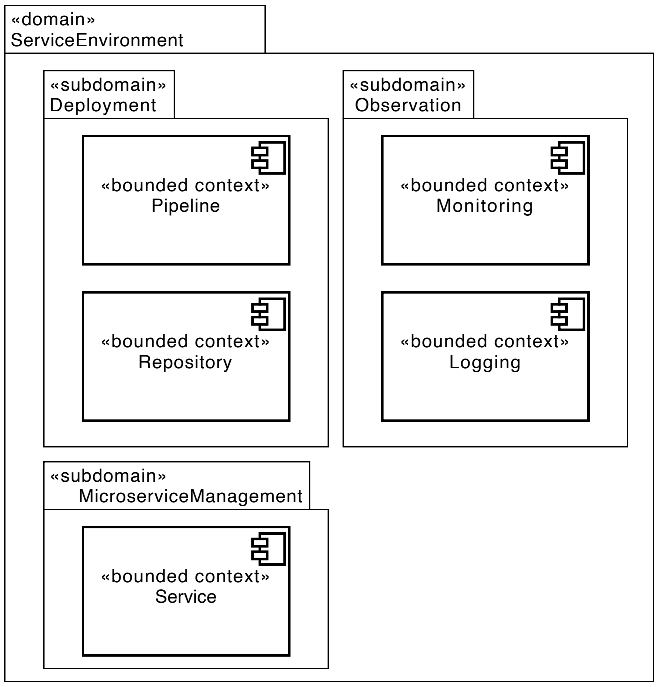


Figure 1. ServiceEnvironment Context Map.

The subdomain Observation includes everything regarding the observation of a microservice. The bounded context Monitoring is implemented as a domain microservice that stores monitoring events (e.g., if a container has been created). Logging data can be stored within the bounded context Logging. If tools such as Prometheus were to be used, they would be located in the subdomain Observation and replace or complement the domain microservice Monitoring.

The subdomain Deployment includes a bounded context Pipeline, which represents a CI/CD pipeline including its steps that are used across the ServiceEnvironment domain (e.g., build or test). Furthermore, the bounded context Repository stores the microservice source code, documentation, API specification dependencies, or design artifacts. Such bounded contexts are later not implemented as domain microservices. Instead, the functionalities are provided by GitLab.

Finally, the subdomain MicroserviceManagement includes a bounded context called Service which should store information about a deployed microservice (e.g., API, deployment name, version, or dependencies). This bounded context is also implemented as a domain microservice.

The context map and the ubiquitous language are not final. Another application, requiring additional subdomains, bounded contexts or terms can add them to the domain and reuse existing subdomains.

B. *MicroserviceDeveloperPortal*

The MDP is the first application in the ServiceEnvironment domain. The basic idea of the MDP is to have a single source of truth for developers and operators alike. For example, whenever a developer wants to find out which microservices are currently running in a Kubernetes cluster or find the API specification of a specific microservice, the portal should be the first place to look for this information. Figure 2 presents the application sketch, which defines how a developer interacts with the ServiceEnvironment to get the information they are looking for. These interactions lead to two distinct capabilities which must be provided by the MDP.

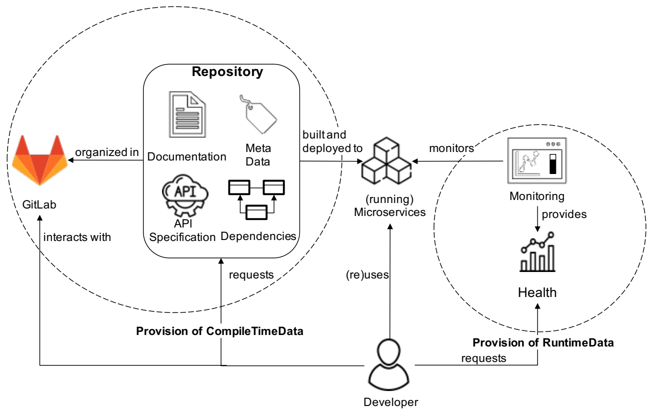


Figure 2. Application Sketch of the MDP.

1) **Provision of CompileTimeData:** Data already present during the compilation of a microservice should be collected and provided to the user. This data includes information of the repository provided by Git (e.g., commit reference or name of the branch), API data in form of OpenAPI specifications or dependencies on other microservices (e.g., databases). This information is of particular interest to developers who plan on reusing existing microservices and therefore are looking for their documentation.
2) **Provision of RuntimeData:** This data can only be collected while the microservice is running. For now, the MDP is only capable of monitoring a simple health state of microservices. This information of the health state is interesting to developers who are troubleshooting problems that occur during runtime.

While the introduced application sketch captures interactions between various actors and objects, a relation view captures the data entities, called shared entities, that are used for these interactions. Figure 3 shows the entity relation view of the MDP, which contains the entities shared by the bounded contexts of the domain ServiceEnvironment (i.e., shared entities) relevant for the MDP. Furthermore, the shared entities are set into a relationship with one another.

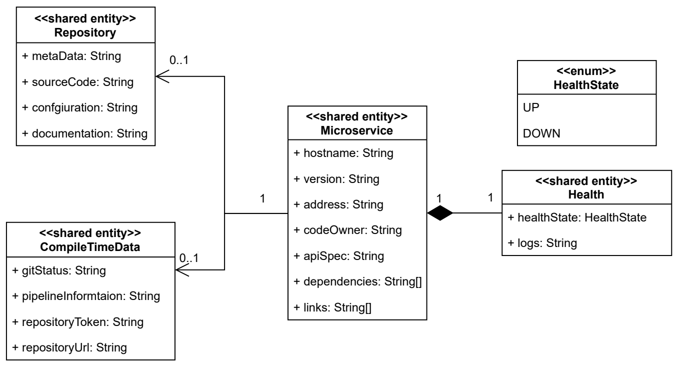


Figure 3. The Entity Relation View of the MDP.

The shared entity Repository represents a Git repository provided by GitLab. Thus, it contains source code, configuration artifacts, and documentation. The CompileTimeData entity represents all data that is available during the compilation of the microservice source code. Furthermore, the CompileTimeData contains a reference to a Repository and data about the event (e.g., Git commit) that triggered the pipeline that executed the compilation. This data is then combined into the shared entity Microservice. Each shared entity Microservice has an associated shared entity Health. This entity contains the most recent HealthState of a microservice, which can be *UP* or *DOWN*, together with logs that were written when the health state was observed.

Different services are responsible for creating and storing these data entities. The application sharing view in Figure 4 visualizes relations between shared entities and bounded or application contexts. Bounded contexts represent reusable services that are part of the ServiceEnvironment domain. Applications contexts provide the application specific logic for the MDP and are not reusable by other applications in the ServiceEnvironment.

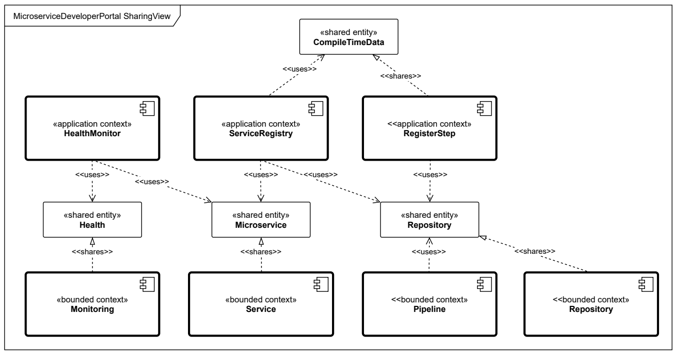


Figure 4. The Application Sharing View of the MDP.

While the bounded contexts Pipeline and Repository are already present in the system, the bounded contexts Service and Monitoring are implemented as domain microservices. The domain microservice Service stores the currently registered microservices and provides the data through CRUD operations. The domain microservice Monitoring stores the monitoring events for the currently registered microservice and provides the data through CRUD operations. The application contexts HealthMonitor, ServiceRegistry and RegisterStep perform specific tasks for the MDP. Hence, they are placed in the application layer instead of the domain layer.

The general flow of data starts with the bounded context Repository, which triggers the Pipeline. The bounded context Repository shares the shared entity Repository, which is used by the bounded context Pipeline and each of its steps (e.g., build, test, deploy). An application context RegisterStep is added to the CI/CD pipeline, which creates and shares the shared entity CompileTimeData with the application context ServiceRegistry. The application context ServiceRegistry uses the shared entity CompileTimeData and the shared entity Repository to create a shared entity Microservice which is persisted by the bounded context Service. The application context HealthMonitor uses the shared entity Microservice to determine which microservices in the cluster have to be monitored. The shared entity Health is created from the results of the application context HealthMonitor and persisted and shared in the domain by the bounded context Monitoring.

The application contexts Pipeline, RegisterStep, ServiceRegistry and bounded context Service fulfill the capability to provision CompileTimeData, while the remaining two contexts HealthMonitor and Monitoring are needed to fulfill the capability provision of RuntimeData.

IV. Microservice Registration

Each deployed microservice in a Kubernetes cluster should be represented by a shared entity Microservice. To create these entities, either the current state of the ServiceEnvironment has to be observed at all time or the deployments and undeployments have to be monitored. Because the pipeline can be used to perform a job after each successful deployment, the latter approach was chosen. Thus, an extension for the pipeline was developed with the goal of automatically registering a microservice after a successful deployment. The pipeline extension represents the application context RegisterStep.

To use the RegisterStep, the pipeline receives two new stages. First, the register stage is triggered after a successful deployment of a microservice. Second, the unregister stage is executed parallel to the undeployment of a microservice. The RegisterStep is a Docker image that contains Python scripts for each stage. Depending on the stage, either the register or the unregister script is executed. The scripts send a HTTP request to the representational state transfer (REST) endpoint of the ServiceRegistry to either register or unregister the microservice.

Figure 5 shows the process for registering a new microservice. The Repository triggers the execution of the Pipeline (1)

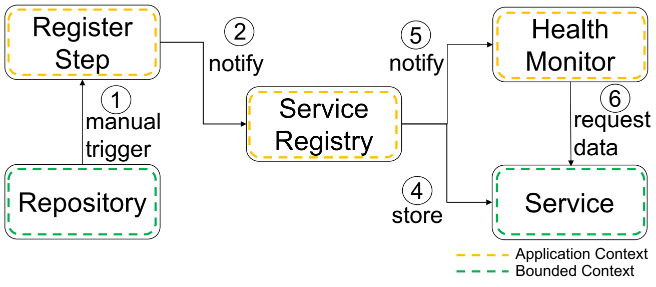


Figure 5. The Service Registration Process.

which notifies the ServiceRegistry that a new microservice has been successfully deployed (2). This notification includes data about the name of the deployment and the URL to the Git repository. The ServiceRegistry then requests relevant information from the repository such as the API specification and the dependencies to other microservices (3). After gathering all relevant information, the data is passed to the domain microservice Service which persists it in a database (4). In the last step, the HealthMonitor is notified about the new deployment (5). This will cause the HealhMonitor to request an updated list of the currently running microservices from the domain microservice Service (6).

The process to unregister a microservice is similar to the registration process with three essential differences. First, the pipeline is not necessarily triggered by the repository but can also be triggered manually by a developer. Second, after the ServiceRegistry is notified, it will not request data from the repository. Finally, the ServiceRegistry will delete the microservice entity from the domain microservice Service instead of saving it.

Figure 6 presents an excerpt of the GitLab pipeline. The columns specify the different stages of the pipeline and the boxes specify the tasks which should be executed in each stage. The lines between the boxes visualize a dependency relationship between the tasks. A microservice is only registered after a successful deployment. A microservice is unregistered when the undeploy stage is executed. In this step, the connected entry will be deleted from the domain microservice Service and the HealthMonitor is notified about the update. The undeploy stage should be executed whenever the base branch for the deployment is deleted or the microservice is not needed anymore.

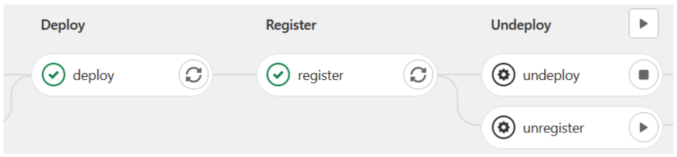


Figure 6. Extract of the GitLab Pipeline.

This workflow ensures that the domain microservice Service always holds an up-to-date list of the shared entities Microservice which represent the microservices that are running in the cluster. The name of the deployment in the Kubernetes cluster is used as a unique identifier for the running instances of the microservice and its dependencies (e.g., database container). Other services can now request a list of deployments and collect further information with a reference to the deployment. The dashboard can request the list of all registered deployments and get more detailed information by querying data from other data sources using a specific deployment name. This process is described in the following section with the example of a service that monitors the health states of deployed and registered microservices.

V. Health Monitoring

One of the MDP's capabilities is the provisioning of runtime data. In the initial version of the MPD, the focus is set on the health data of running microservices that are registered with the MDP. In the ServiceEnvironment, microservices are running in a Kubernetes cluster which offers a pod lifecycle that is used to determine the health state of containers running inside a pod. The health state is stored in a health data entity, which includes the current state of the pod (e.g., running or terminated) and a log message.

Two microservices are responsible to track the current state of a deployed microservice and storing the health data. Those are the application microservice HealthMonitor and the domain microservice Monitoring, which can also be found in Figure 4. Since the microservice Monitoring is part of the ServiceEnvironment domain, it offers CRUD functionalities and is responsible for storing health data. The stored health data can later be displayed in a dashboard. HealthMonitor is responsible for extracting the health data from the microservices and is implemented as a Kubernetes operator in Go. An operator is a software extension for Kubernetes and has access to Kubernetes cluster resources such as pods and events. If the status of a pod changes (e.g., container running or container stopped), an event is created. The operator works in a control loop and will receive the new event. Afterward, the operator can process the event accordingly.

An overview of the operator process can be found in Figure 7. HealthMonitor fetches a list of registered services from the domain microservice Service. This list determines which deployed microservices must be monitored. The list is fetched whenever the ServiceRegistry registers a new microservice and notifies HealthMonitor through an API endpoint (see Figure 5). If the state of a microservice in the cluster changes, HealthMonitor is notified and will check if the microservice is registered with ServiceRegistry (i.e., the microservice is in the fetched list of registered microservices). If this is the case, HealthMonitor will process the event and store the data in the domain microservice Monitoring. The dashboard can then access the health state for a microservice through REST calls to the domain microservice Monitoring.

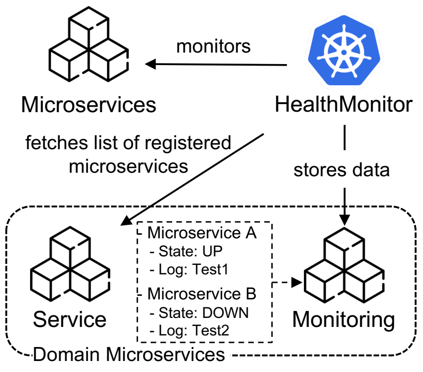
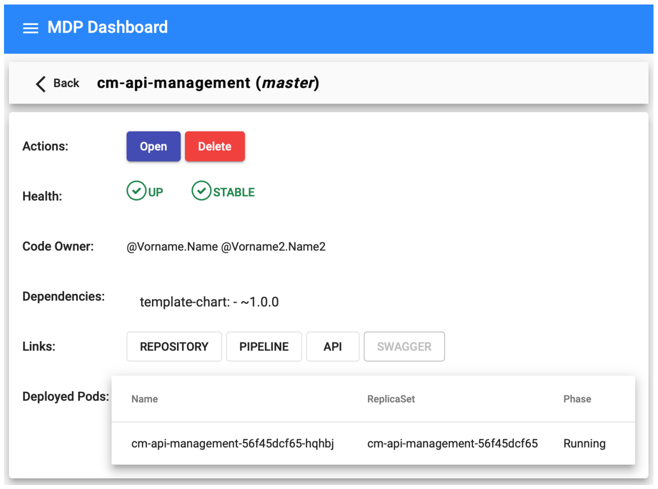


Figure 7. Overview of HealthMonitor.

VI. Use of the MDP Application in a Real Service Environment

Currently, the MDP is deployed in a Kubernetes cluster and is used by up to 20 users including academic staff and students. The cluster runs five containers for the MDP including the domain microservices Service and Monitoring, as well as the application microservices ServiceRegistry, HealthMonitor, and a container for the front-end. The Git repositories as well as the pipeline and the pipeline runner are provided by GitLab. Each academic semester, students develop microservice-based applications using DevOps templates which configure and set up the CI/CD pipeline [5]. The DevOps templates automatically include the additional register and unregister step. Hence, the microservices are all automatically deployed to the Kubernetes cluster and registered to the MDP.



cm-api-management (*master*)

Actions:

Health: UP STABLE

Code Owner: @Vorname.Name @Vorname2.Name2

Dependencies: template-chart: - ~1.0.0

Links:

Deployed Pods:

Name	ReplicaSet	Phase
cm-api-management-56f45dcf65-hqhbj	cm-api-management-56f45dcf65	Running

Figure 8. Excerpt of the MDP Dashboard.

In the context of a practical lecture, students (i.e., developers) fluctuate throughout the semesters. This makes the continuous development of microservice-based applications difficult. The MDP provides an entry-point for new students to see which microservices are currently running in the cluster, who developed them, how they can be accessed (i.e., API specification), and other microservices they depend on. Therefore, they can get into the actual microservice development work faster. In Figure 8, an extract of the dashboard of the MDP shows a selected registered microservice (i.e., cm-api-management). The user can see which version of the microservice is deployed and can automatically open it if a Kubernetes Ingress resource is created (i.e., HTTP traffic from outside the cluster is allowed). Moreover, the user is presented with information about the code owner, dependencies (e.g., Helm dependencies) and links to the repository, the executed pipeline, the API specification, and a Swagger UI (if available). Finally, the user can see the health status of the deployed instances of the microservices.

While the MDP currently works in a real microservice environment, further validation of the results has to be done. Especially regarding the correctness of the stored data including the API specifications and information of a deployed microservices (i.e., CompileTimeData and RuntimeData). Moreover, analyzing if the MDP does support developers and improve their workflow should be performed through a questionnaire or case study.

VII. Conclusion and Future Work

Reusing existing software components should be a high priority in a sustainable software development environment. Microservice-architectures offer a good opportunity to reuse or exchange individual components in the form of microservices.

Before an existing microservice can be reused, a developer has to know where a microservice is located and how it can be accessed. Keeping track of the currently deployed microservices in a cluster can be difficult once the amount of microservices increases. Therefore, we developed an application called MicroserviceDeveloperPortal (MDP) to provide users with all necessary information.

A key contribution of our approach is a first solution for the architecture of an environment which contains everything that is necessary to build, deploy, run, and operate microservices and its dependencies. We call this environment ServiceEnvironment. The ServiceEnvironment provides domain microservices which offer operations to store and retrieve relevant data. The domain microservices are used by the MDP but can also be reused by other applications.

The data is differentiated in CompileTimeData and RuntimeData. CompileTimeData contains data generated during the build process and is sent by the pipeline step to the MDP. The RuntimeData contains health information and is collected by the MDP through a Kubernetes operator. Finally, the user can retrieve the data through a front-end. Since the MDP is a microservice-based application running in Docker containers, it can be deployed in any Kubernetes environment. Although the paper used GitLab as a solution for a Git repository and CI/CD solution, exchanging GitLab for another solution such as AzureDevops or Jenkins should be easily possible.

The future development of the MDP includes an improvement of the architecture and the additions of new capabilities. Currently, the RegisterStep is modeled as an application context. Logically, the RegisterStep is closer to the bounded context Pipeline and thus should be part of the ServiceEnvironment domain. Moreover, the monitoring aspects of the MDP uses a custom solution to monitor the Pods running in the cluster. The monitoring aspects of the MDP should rather be replaceable by an arbitrary existing monitoring solution (e.g., Prometheus) which has to be modeled accordingly. Then, the application microservice HealthMonitor has to perform a mapping to the existing monitoring solution rather than performing the monitoring itself.

Future research will focus on the management of APIs, which includes researching what is required for the management of APIs and how it can be done by creating a reusable microservice architecture. The management of APIs is mostly neglected in the current version of the MDP and should be added in a future version. Since APIs and microservices depend on each other, the information about APIs could also be stored within the ServiceEnvironment domain. Ideally, the RegisterStep is extended to store the API specification of a registered microservice. This would also allow further API management capabilities such as the versioning of APIs as well as the configuration of API gateways to directly allow developers to access a microservice.

REFERENCES

[1] M. Fowler and J. Lewis, "Microservices," Thouhtworks, Tech. Rep., 2014, [retrieved 21/08/2021]. [Online]. Available: http://martinfowler.com/articles/microservices.html

[2] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Inc., 2019.

[3] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[4] B. De, *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. Apress, 2017. [Online]. Available: http://link.springer.com/10.1007/978-1-4842-1305-6

[5] S. Throner *et al.*, "An advanced devops environment for microservice-based applications," in *2021 IEEE 16th International Conference of System of Systems Engineering (SoSE)*, 2021, in press.

[6] J. Lewis, "Micro services - the java way," Thouhtworks, Tech. Rep., 2012, [retrieved 21/08/2021]. [Online]. Available: http://2012.33degree.org/talk/show/67

[7] S. Newman, *Building Microservices: Designing Fine-grained Systems*. O'Reilly Media, Inc., 2015.

[8] L. Chen, "Microservices: Architecting for continuous delivery and devops," in *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*, 2018, pp. 39–46.

[9] Sysdig, "2019 container usage report," Sysdig, Tech. Rep., 2019, [retrieved 21/08/2021]. [Online]. Available: https://dig.sysdig.com/c/pf-2019-container-usage-report

[10] Portworx and Aqua Security, "2019 container adoption survey," Porworx and Aqua Security, Tech. Rep., 2019, [retrieved 21/08/2021]. [Online]. Available: https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf

[11] Cloud Native Foundation, "CNCF survey 2020," Cloud Native Foundation, Tech. Rep., 2020, [retrieved 21/08/2021]. [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf?utm_source=thenewstack&utm_medium=website&utm_campaign=KCCNC-NA-2020-Referral

[12] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "An exploratory study of devops extending the dimensions of devops with practices," *ICSEA 2016*, vol. 104, pp. 91–99, 2016.

[13] F. Erich, C. Amrit, and M. Daneva, "A qualitative study of devops usage in practice," *Journal of Software: Evolution and Process*, vol. 29, no. 6, p. e1885, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1885

[14] R. Wilsenach, "Devops culture," Tech. Rep., 2015, [retrieved 21/08/2021]. [Online]. Available: https://martinfowler.com/bliki/DevOpsCulture.html

[15] Amazon Web Services, "Aws for developers — programming languages, tools, community — aws developer center," AWSb, Tech. Rep., 2021, [retrieved 21/08/2021]. [Online]. Available: https://aws.amazon.com/de/developer/

[16] Mercedes-Benz, "Mercedes–benz /developers – the api platform by daimler," Mercedes-Benz, Tech. Rep., 2021, [retrieved 21/08/2021]. [Online]. Available: https://developer.mercedes-benz.com

[17] S. Abeck *et al.*, "A context map as the basis for a microservice architecture for the connected car domain," in *INFORMATIK 2019*, 2019, pp. 125–138.

[18] B. Hippchen, M. Schneider, I. Landerer, P. Giessler, and S. Abeck, "Methodology for splitting business capabilities into a microservice architecture: Design and maintenance using a domain-driven approach," in *Conference on Advances and Trends in Software Engineering (SOFTENG)*, 2019, pp. 51–61.

[19] M. Schneider, B. Hippchen, P. Giessler, C. Irrgang, and S. Abeck, "Microservice development based on tool-supported domain modeling," in *Conference on Advances and Trends in Software Engineering (SOFTENG)*, 2019, pp. 11–16.

[20] A. Wiggins, "The twelve-factor app," 12factor, Tech. Rep., 2012, [retrieved 21/08/2021]. [Online]. Available: https://12factor.net/