Strategies for Documentation of Microservice Systems

— Bachelor Thesis —

Jennifer Nissley

University of Hamburg
MIN Faculty
Department of Informatics
B.Sc. Informatics
Matr. Nr.: 6232349

First Reviewer: Lars Braubach
Second Reviewer: Kai Jander
Research Group Distributed Systems

Abstract

This thesis presents a novel approach for microservice system documentation combining API documentation and a service dependency graph to provide an overview of the whole system and documentation of single services in one interface that is easy to navigate and provides multiple levels of detail for developers working on expanding and maintaining the system. The merits and shortcomings of existing solutions are highlighted and the novel approach evaluated based on requirements collected from literature on software system and microservice documentation. The resulting software tool sccp and the website are able to combine OpenAPI documentation with dependency graphs generated by Kiali or written by hand in a novel JSON format, but could be further developed to increase usability and integrate more documentation solutions established in the industry.

Contents

1 Introduction

Microservices are a growing practice in the development of large software projects, especially in the growing market of web-based services. Large monolithic systems are replaced by many independently developed microservices, which can very quickly be rearranged or exchanged by newer versions. Developers of one system are divided into groups, each working on one service and making their own design choices. These teams are often working for different companies, providing solutions for specific problems to other companies, who integrate their service into their system. Ideally, the only thing a developer from another team needs to know to use a microservice is the provided API and since the workings hours of a developer are very expensive, a microservice that can not be quickly understood and integrated, will be disregarded. Therefore, a comprehensible API and a good documentation are the most important prerequisites for the success of a microservice.[1]
In this rapidly changing environment, an overview over the system of microservices and how they interact with each other is also needed. The purpose of this thesis is to combine an overview of the whole system with the API documentation of each service to provide both an understanding of the whole system and an easy way to find out how to use a single service. To reach this goal, multiple existing solutions will be combined to create a system documentation for developers.

Chapter 2 will define microservices and point out their benefits compared to monolithic software. It will give an example of how to implement services and how they are able to communicate with each other.

In chapter 3, the type of documentation created in this thesis will be classified and a list of requirements will be established to evaluate different documentation methods. Existing solutions will be presented and their shortcomings regarding system documentation in the context of microservices will be highlighted.

A novel approach based on a combination of existing solutions will be presented in chapters 4 and 5. In a first step, API documentation for single services will be combined and displayed as a single document while keeping the ownership relation between a service and its endpoints visible.

Chapter 5 will present a novel website providing a single interface for API documentation and a system overview in the form of a dependency graph. This novel approach will be evaluated based on the requirements established in chapter 2.

2 Microservices

Microservices are small programs with a single task that communicate with users and each other over a network, each running independently with its private resources. A system of microservices, each with only one small task, can replace a more complex monolithic program. Every service can be developed by a different team of programmers in their preferred programming language. They are easy to replace and can be reused in different projects. Distributing them over multiple servers is an easy task since they use network communication by standard and rarely share resources.[2]
To make it more clear, what a microservice is and how a system of microservices can work together to solve complex tasks while staying flexible and easy to manage, here is a very basic example. It also gives an idea on how they can be programmed and gives a basis for later experiments with different approaches of documentation.

2.1 Example

Figure 2.1 shows the structure of the system. In an imagined environment with multiple servers, there could be multiple instances of a service that reads temperatures of CPUs. This information could be used to observe the effectiveness of a cooling system or to detect overheating of one of the servers. To interpret the data, it has to be collected on a central server. On this server, another microservice, called the client, receives all the data and writes it into a file. While the service that reads temperatures is programmed in Go, the service that writes into a file is programmed in Python. They communicate over a language that can be interpreted by both. In this case the temperature service provides an HTTP GET operation over which the collecting service can request the current temperature. The request and the data are encoded in JSON messages, which can be decoded into Python and Go variables, like strings and arrays.

Listing 2.1 shows the code of the service that reads the temperatures. The main function of the service initializes the endpoints. An endpoint is a URL, a network address, to which requests can be sent to invoke a function of a service. One microservice can offer multiple endpoints, for example if one service manages a database of customers, there could be one endpoint for reading customer information and one for adding a new customer. In this case there is only one endpoint, `/temp`, that offers the function of reading temperatures on the server that the service is running on. To establish this

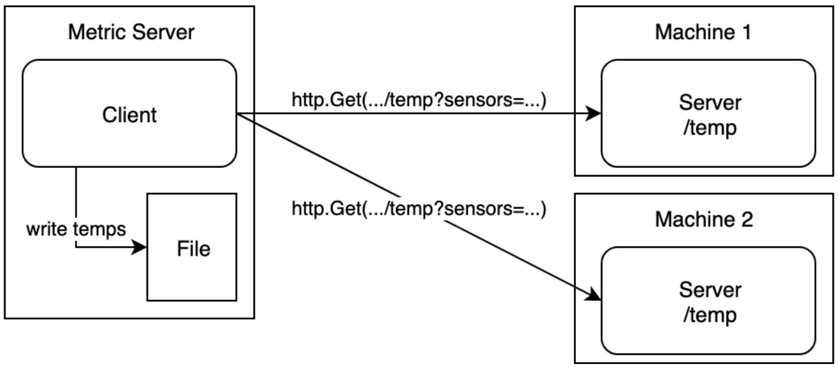


Figure 2.1: A small system with two microservices communicating over the HTTP protocol. The two servers only read temperatures and the client only stores the results. Each machine has its own instance of the temperature reader.

endpoint, the main function of the service uses two functions from the HTTP library. `HandleFunc()` assigns a function called `temp` to the URL `/temp`. This function will be called anytime the service gets a request to `/temp`, either from a human typing it into the address bar of a browser or another service using an HTTP library to send the request. The function `ListenAndServe()` starts the process of infinitely running a loop, in which a request is taken in, the associated function is called and an answer is sent. The IP and port number parameter specifies the address to which requests have to be sent to invoke the service.

This design makes the function available for many use cases, since there are HTTP libraries for most programming languages and every computer can be connected to a network over which requests can be sent to another computer, that runs the service. The function is not hidden in the code of a huge program, but available to every developer that knows any programming language and has a basic knowledge of network protocols. It also runs on its own, since the service has only one function and its own main function, so it is independent of the system around it and can easily be inserted into any other system. If another part of the system crashes, the service will still be running and answer its requests.

The use of parameters is a bit different from traditional programming. They are encapsulated in an HTTP message, so they can't be identified by simply reading the function header. That's one reason why good documentation of the API will highly increase the number of developers willing to use it and decrease the time spent to understand how to use it. The function `temp` has to first unpack the HTTP message found at the memory address `req`, use the function `URL.Query()`, provided by the HTTP library, to convert the parameters inside it to data types that are known to the programming language, do the work, compose an answer in JSON format and send it to the client with the help of the `ResponseWriter rw`, provided to `temp` as a parameter.

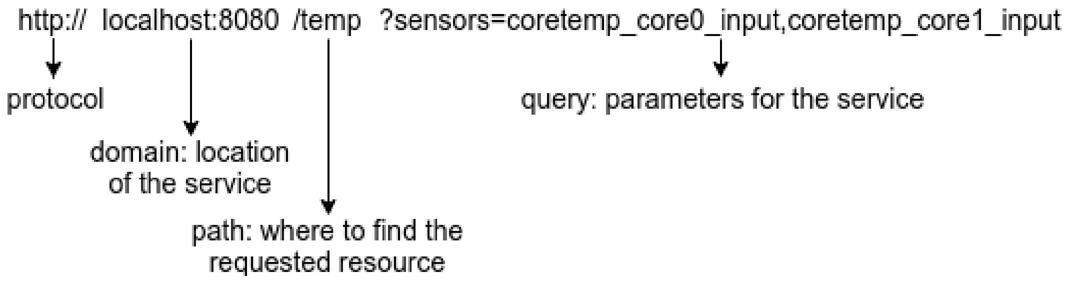
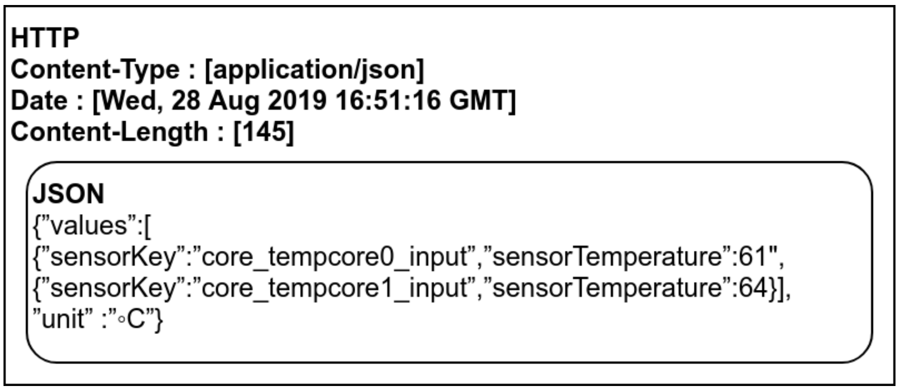

http:// localhost:8080 /temp ?sensors=coretemp_core0_input,coretemp_core1_input
protocol
domain: location of the service
path: where to find the requested resource
query: parameters for the service

Figure 2.2: URL for a request to the temperature service

The HTTP request send to the service contains a simple URL, shown in figure 2.2. The service needs to know which temperatures to read, so it unpacks the query part of the URL with `URL.Query()` into a map, that contains key-value pairs. In this case, the only key is `"sensors"` and the values are the names of temperature sensors. All values for the key `"sensors"` are stored in a string, separated by commas, with the function `Get(key)`. If there are no values for the key `"sensor"`, no sensor key or no query string, all sensors are requested.

If an error occurs while reading the temperatures, the server sends an HTTP error message to the `ResponseWriter`. `StatusInternalServerError` tells the client that the server was not able to complete the requested task because an unexpected error occurred, namely the sensors did not return temperatures. The `ResponseWriter` directs the message to the client. After a successful read, the string of temperatures is packaged into a self-defined struct. The struct definition defines names, that it's variables will take on when the struct is converted to JSON. The string `"Values"` is an array of `TemperatureStats`. Every `TemperatureStat` is a struct, including the name of the sensor and the temperature. The Unit is a simple String that defines whether the temperatures are in Celsius or Fahrenheit. With `json.Marshal`, the go struct is converted to the JSON format, resulting in listing 2.2. This answer is given to the `ResponseWriter`, which packages it into an HTTP message (see Figure 2.3) and sends it over a network to the client. The easiest way to test the service is to start the service and type in the URL from Figure 2.2 into the address bar of a browser. The answer from listing 2.2 should be displayed in the window.


HTTP
Content-Type : [application/json]
Date : [Wed, 28 Aug 2019 16:51:16 GMT]
Content-Length : [145]

JSON
```
{"values":[
{"sensorKey":"core_tempcore0_input","sensorTemperature":61",
{"sensorKey":"core_tempcore1_input","sensorTemperature":64}],
"unit" :"◦C"}
```

Figure 2.3: Response send from the server to the client in the form of JSON inside an HTTP message which can be sent over a network from one machine to another

```go
// data structure of the http response to the client
type tempresp struct {
  // temperatures of all requested sensors
  Values []host.TemperatureStat `json:"values"`
  // unit of the temperatures
  Unit string `json:"unit"`
}

func main() {
  // when the URL <this server>/temp is called, run the function temp
  http.HandleFunc("/temp", temp)
  // respond to requests for this server on the port 8080
  http.ListenAndServe("0.0.0.0:8080", nil)
}

func temp(rw http.ResponseWriter, req *http.Request) {
  rw.Header().Set("Content-Type", "application/json")
  // get the request parameters
  params := req.URL.Query()
  // a string containing all sensors from which to read temperatures
  sensorsStr := params.Get("sensors")
  var sensors []string
  if len(sensorsStr) > 0 {
    sensors = strings.Split(sensorsStr, ",")
  }

  // get the temperatures of all sensors
```

```go
	values, err := host.SensorsTemperatures()
	if err != nil {
		fmt.Printf("error: not able to get temperature: %s", err)
		// tell the client that an error occurred and end the function
		http.Error(rw, err.Error(), http.StatusInternalServerError)
		return
	}

	// filter the temperatures for the ones requested by the parameter "sensors"
	if len(sensors) > 0 {
		bufValues := make([]host.TemperatureStat, 0, len(sensors))
		for _, sensor := range sensors {
			for _, tempStat := range values {
				if tempStat.SensorKey == sensor {
					bufValues = append(bufValues, tempStat)
				}
			}
		}
		values = bufValues
	}

	// built the JSON response and give it to the response writer rw
	resp := &tempresp{
		Values: values,
		Unit: "°C",
	}
	byteResp, _ := json.Marshal(resp)
	rw.Write(byteResp)
}
```

Listing 2.1: A simple microservice reading requested temperatures and sending the results to its client, written in Go

```json
{
	"values": [
		{"sensorKey":"coretemp_core0_input", "sensorTemperature":61},
		{"sensorKey":"coretemp_core1_input", "sensorTemperature":64}
	],
	"unit":"°C"
}
```

Listing 2.2: Answer from the temperature reader service in JSON format

To integrate the service into a bigger system, a caller function has to be written. In this example, the client is a microservice that stores all the temperatures on a special metrics server. Listing 2.3 shows the code. The temperatures are requested in line 12 with the Get function provided by the HTTP library. The string parameter of the `http.Get()` function is the same URL that would be typed into a browser address bar and is explained in figure 2.2. It includes the names of the requested temperature sensors `coretemp_core0_input` and `coretemp_core1_input`. The answer is stored in a struct, defined beforehand as `tempresp`. It has the same structure as the response struct used by the server. If the client had been written in another programming language, the struct definition would look a little different, but it would have the same JSON names. The JSON format is the common language used by these microservices. Other formats are possible, but services need to agree on one language. Such communication rules are defined in the API of a service.

```
 1  type tempresp struct {
 2    Values []host.TemperatureStat ‘json:”values”‘
 3    Unit string ‘json:”unit”‘
 4  }
 5
 6  // request the temperatures of core 0 and 1
 7  // and store them into a file
 8  func main() {
 9    // request the temperatures from the microservice
10    // send parameters by adding ? to the url
11    resp, err := http.Get(”http://localhost:8080/temp” +
12      ”?sensors=coretemp_core0_input,coretemp_core1_input”)
13    if err != nil {
14      // this writes an error message and exits the program
15      log.Fatalf(”http request failed: %s”, err)
16    }
17
18    // translate the json response into a go struct
19    dec := json.NewDecoder(resp.Body)
20    temps := &tempresp{}
21    err = dec.Decode(temps)
22    if err != nil {
23      fmt.Println(”dec.Decode”, err)
24      log.Fatalf(”decoding of response from http://localhost:8080/temp failed”)
25    }
26
27    // write the temperatures into a file
28    f, err := os.OpenFile(”serverTemps.txt”,
          ↪ os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
```

```
    if err != nil {
        log.Fatalf("failed to open log file: %s", err)
    }
    defer f.Close()
    fmt.Fprintf(f, "%s:\n", time.Now().Format(time.RFC3339))
    for i := 0; i < len(temps.Values); i++ {
        fmt.Fprintf(f, "%s: %f\n", temps.Values[i].SensorKey,
            ↪ temps.Values[i].Temperature)
    }
    fmt.Fprintf(f, "\n")
}
```

Listing 2.3: A simple microservice using another microservice to collect data, written in Go

Writing such clients just to try out services that might be useful, is a lot of time-consuming work, so examples and the possibility to try out endpoints without writing code are useful features in documentation.

In this example, it actually matters which service runs on which computer, but in practice it mostly does not matter, so the services can be distributed evenly over a cluster of computers. If a new microservice is needed, it can be started on any computer that has enough resources left. Doing this by hand would be time-consuming. Therefore, and for many other reasons, microservices in practice are used in combination with tools that make the most of their benefits. Here are some of the most common ones:

2.2 Tools for Microservices

Microservice systems are often run on clusters, multiple machines connected with each other. To spread the services across the machines, monitor their behavior and make the system resilient against failures of single services, support from software tools is useful.[3]
They are briefly introduced here to give a better understanding of how microservices are used in practice and because some of their functionality will later be used to collect information for documentation.

2.2.1 Container Virtualization

Container virtualization is often used to isolate services and to send them to machines inside the cluster over network connections. Docker is a widely used tool that implements this functionality.[4] It allows processes to run in an isolated environment by wrapping them inside containers with all their dependencies and only giving them access to a part of the system resources. It has two advantages for microservices: Firstly, it prevents the services from interfering with each other. If each service has their own container, one service can not access the resources of another service.[5] So if, for example, one service has a bug that is using up all its memory space, the other services on the machine can continue their work as if nothing is happening. Without this protection, the machine could run out of memory space, crash and bring the whole system to a halt. The requests to the faulty service might even be rerouted to another service with the same functionality or to another instance of the same service that has not yet run into that bug, so that users won't even notice a difference. Secondly, docker containers can bundle services with all their dependencies and provide a single uniform interface for them, which makes it easy for developers or for orchestrators like Kubernetes to deploy them on any machine. So if there is a cluster of machines running an application that is composed of a system of microservices, each service can be packaged in a container and deployed on any machine without the need to install all its dependencies first and without installing and starting the service by hand.[4] The container can also contain a shell script that checks if all needed resources, like network connections or access to databases, are available and prepares the environment inside the container, before the service is started, like an installer would do. If the resources of the cluster are not sufficient anymore, a new machine can be added to the cluster and one of the services can be deleted on its old machine and installed on the new machine just by moving the container.

2.2.2 Orchestrators

The containers can be managed by hand, meaning an administrator logs into a machine of the cluster, downloads the container of the service that is supposed to run there and types in a docker command to run it. This would be done for every service of the application, for some services a second or third version would also need to be run simultaneously. The administrator would need to devise a plan on which services to run on which machines. If a machine runs out of memory space, the administrator needs to contemplate which service to stop and on which machine it should be restarted. Every time a service runs into a fatal error or has too many requests pending, a new instance of the service or another version of it must be started somewhere on the cluster to keep the application running. While doing this, the destinations of all the messages to the relocated services need to be adjusted, since the services change their IP address and URL every time they are started on a new machine. Multiple full-time employees would be needed to ensure that the application is running continuously to serve users 24 hours a day and 7 days per week. Instead of paying four employees, an orchestrator can be installed on the cluster to do the same job. A popular example is Kubernetes.

Kubernetes uses containers like the ones provided by Docker, wraps one or multiple of them into a pod and runs these pods on the machines of a cluster (in Kubernetes, the machines are called nodes). A Scheduler automatically deploys pods to nodes, depending on the resources they need.[6] Kubernetes also provides a DNS Service Discovery in which all IP addresses of the services are stored. It runs as a service inside the Kubernetes cluster under the name kube-dns. Requests from one service to another are not addressed directly to the URL or IP address of the receiver, but to a URL provided by the DNS service that includes the name of the service. The client service asks the DNS service for the current IP address of the destination service and sends the request to it. Multiple pods can run under the same service name to ensure that the application keeps running if one of the pods or nodes runs into errors or is overwhelmed by too many requests at one time. Kubernetes can distribute the requests between all the pods running under the same name.[7] This means that developers do not need to know where the service they are using is located or how many instances of it are running, they only need to know the name of the service and the name of the function that it offers in its API (the endpoint). Also, the code does not need to be changed when a service changes locations inside the cluster or if one instance of it crashes and an alternative instance needs to be requested.

If a node goes down or a service is not responding, Kubernetes can automatically replace it by starting a new instance of it on another node. Developers can define tests for Kubernetes to run in a certain interval to check if the service is still responding and giving the correct output. These are called health checks. If the health check fails, Kubernetes automatically executes the defined action for this case, for example start another instance of the same service or start another version of the service. [6]

2.2.3 Traffic Management and Monitoring

Orchestrators help services to find and communicate with each other, but they mostly do not control or keep record of the network traffic between them. For a finer control, tools like Istio can be installed. Istio runs as a service in Kubernetes inside a pod. It intercepts all messages send over the network of the cluster, gathers information about them and redirects them based on rules set by the administrator.[8]

If there is an error in the system, the stored information can be used to understand what is happening. Is there a service that sends huge amounts of requests to other services, overburdening them with unnecessary work? Are all services responding to the requests they get and is there a service that takes an unusual amount of time to respond? Does one of the services request data that it is not allowed to have? The information can also be used in combination with other tools like Kiali to create a visual overview over the system and the interactions between its services. Later in this thesis, it will be used to create a graph, depicting the interactions of microservices inside a system managed by Kubernetes. This graph will be used to arrange the API documentations of the single services into an overview of the whole system in chapter 5.

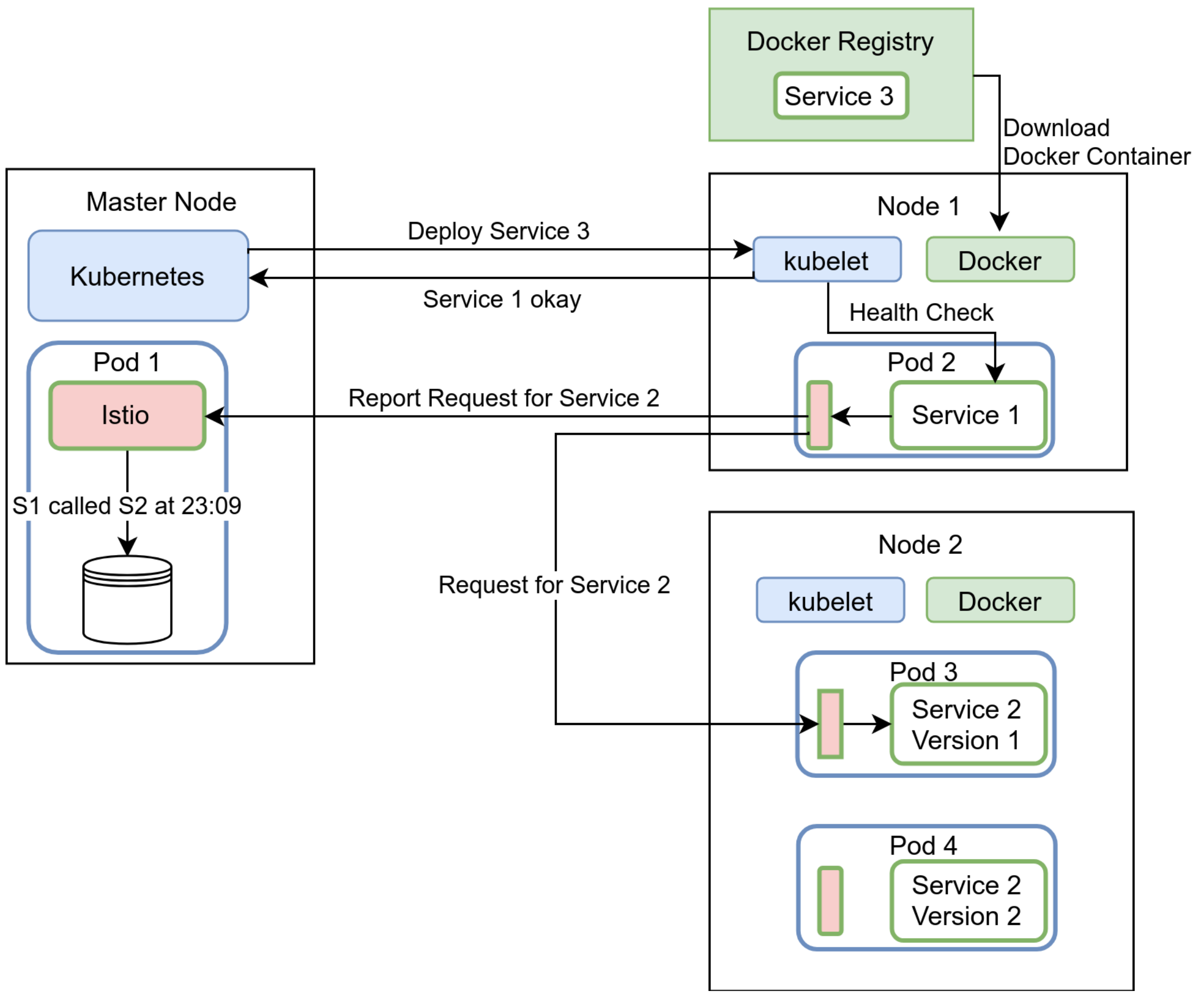


Figure 2.4: The interactions of an orchestrator(Kubernetes), container virtualization(Docker) and a network management tool(Istio) on a cluster.

Redirection of messages is used to gradually introduce changes in the application. Instead of redirecting all requests to a new version of a service, only a certain percent of them will be directed to it and the rest will go to the old version. This ensures that the users can still use the old version if they are not comfortable with the new one or if the new one runs into unexpected errors. Istio's interception of messages can also be used to enforce security rules, by stopping requests from services to resources that they are not allowed to use or data that they are not allowed to access and reporting these requests to the persons in charge of security. [8]

Figure 2.4 shows the interactions of all three tools on a cluster that runs an application composed of three services. The cluster has 3 physical machines, which are called nodes in Kubernetes terminology. The Kubernetes master runs on the master node and deploys pods to all the nodes of the cluster. In this example, Kubernetes decides to deploy service 3 to node 1, because node 1 currently has only one pod running on it and enough resources left to run service 3. It instructs the kubelet running on node 1 to deploy service 3. The kubelet is a Kubernetes program that needs to be installed on all nodes of the cluster. It manages and monitors the pods on its node and communicates with the Kubernetes master. After the kubelet gets the instruction to deploy service 3, it uses Docker to download the service with all its dependencies packaged inside a Docker container from the Docker Registry, a server that stores Docker containers. The service is then packaged inside a Kubernetes pod with other dependencies or resources based on the instructions given by the Kubernetes administrator for this specific pod, for example all services on this cluster have an Istio sidecar inside their pods (small red boxes), which intercepts their network traffic. One of the tasks of a kubelet is to perform health checks on the services to make sure, they are still able to perform their task. In this example, the kubelet of node 1 sends a request to service 1. Service 1 completes the requested task and sends a response to the kubelet. The kubelet checks the response for correctness based on rules given by the administrator. In this case the response is correct and the kubelet informs the Kubernetes master that service 1 is still running and completing its tasks on node 1, so it does not need to be restarted or deployed to another node. Like all other services, Istio runs inside a docker container inside a Kubernetes pod. Its pod also includes a database where it stores all the information it gathers. In this example, service 1 sends a request to service 2. The request is intercepted by the Istio sidecar installed in service 1's pod. It checks the request for any violations of the security rules, informs the Istio master about it and decides to send it to version 1 of service 2. The request arrives at pod 3 and is intercepted by another Istio sidecar, which redirects it to the service.

2.3 Conclusion

Using Microservices is more work at first, because the developers need to know how to use network protocols, some tools are needed to make good use of the benefits and keeping track of the whole system while giving each team the most possible freedom is not an easy task. But in the end, designing a project as a system of microservices makes it easier to quickly adapt to changes, to make the application resilient and to spread the work load among multiple self-reliant teams who can make the design choices best suited to their individual task.

3 Software System Documentation: Goals and State of the Art

3.1 Classification of Software Documentation

Depending on the audience and the stage of development there are three types of software documentation(according to [9]):

- process documentation
- product documentation
  - system documentation
  - user documentation

Process documentation documents the development process of the software. Its purpose is to help development, management and marketing departments to communicate the current status and the desired outcome of an unfinished project. Examples are to-do-lists, schedules, specifications, division of tasks, etc.

Product documentation describes the end product or a certain version of it, without explaining how it was produced. There are two types of audiences for product documentation with differing needs and expertise. On the one hand, users need to know how to use the software to complete their tasks. Their only interaction with it is over a graphical user interface(GUI) or sometimes a command-line interface(CLI). These are the things described by user documentation. Users do not need to know the inner workings of the software, like programming languages, the distribution of tasks among classes or services or the interactions between them. They do not need to know those details, because they do not need to add features to the software or correct errors in it.

Developers on the other hand need to know exactly, which lines of code they need to adjust to correct an error, which parts of the system they have to change to add a new feature and which existing functions they can reuse for it. Administrators also need more details than a user, but different details than a developer. They do not need to change the software itself, they only need a rough overview over the components of the system, how they can be installed on one or multiple machines, which networking and storage requirements they have and what kind of configuration is possible to adjust the software to a certain environment. System documentation provides the detailed view these groups need by showing the single components of the software and their interactions.

This thesis focuses on system documentation for a software system in use consisting of collaborating microservices for developers and administrators working on expanding and maintaining the system. It does not address the documentation needs of project managers or users of the software.
The emphasis is on documentation needed to understand the system as a whole, the information concerning all developers working on the system, not the internal details of single microservices that are only interesting to the developers inside the team working on one specific service.

3.2 System Documentation Requirements

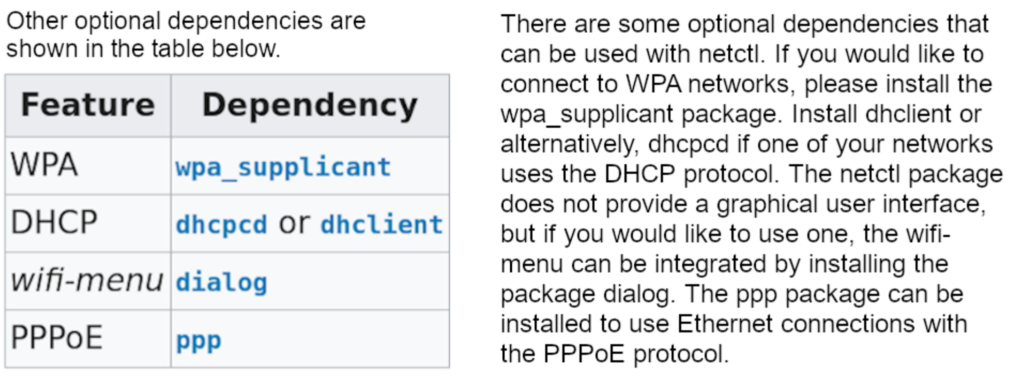
This section describes guidelines for good software system documentation. Most of them are not specific to systems that follow a microservice architecture, but applicable to them. Any well-designed software system consists of multiple components with specific tasks cooperating to solve a bigger problem. In a monolithic system these components could be separate files of code, each containing functions relevant to a certain task, that call functions inside another file to gain information needed to fulfil their task or to solve a subtask. Instead of files, the system could be divided into objects, each containing its own namespace, data and functions. The objects offer some of their functions to be called by other objects. In the same way, microservices have specific tasks and offer endpoints, so their service can be requested by other microservices.

Therefore, we can regard a system of cooperating microservices as a software system and apply the same documentation requirements.

3.2.1 Usability

Readers should find the information they are looking for as quickly as possible.[10] Therefore, documentation should not be one long text users have to read through to get to the one percent they need, but divided into sections and searchable for terms, either by the use of search functions in digital or an index in printed documentation. A digital version does not only automatically offer a search function through browsers and PDF viewers, it is also easier to distribute. Quickly finding relevant information is more important than getting complete knowledge. [10] For most readers, it is sufficient to show the most common use cases instead of including all possibilities. To produce documentation that is both complete and easy to read, it should be divided into multiple levels of detail from a short ”How to get started” with the most common use case and a rough explanation of the capabilities to the more complicated special cases, that require a deeper understanding and a finer control of parameters.[9] This way, most readers will find what they are looking for instantly without being deterred by details they do not need to understand. Since developers working hours are costly, a documentation that does not force them to spend time on unnecessary details is important.

Whenever possible, text should be replaced or enriched by pictures, lists or tables.[10] Pictures are proven to transfer information quicker and keep the information stored inside the recipients brain longer than text.[11][12] Lists and tables are easier to scan through than text. Filtering important information, comparing different options and ticking off a list of steps is much easier if the relevant terms are not buried in text. See figure 3.1 for an illustration of this point.



Other optional dependencies are shown in the table below.

Feature	Dependency
WPA	wpa_supplicant
DHCP	dhcpcd or dhclient
wifi-menu	dialog
PPPoE	ppp

There are some optional dependencies that can be used with netctl. If you would like to connect to WPA networks, please install the wpa_supplicant package. Install dhclient or alternatively, dhcpcd if one of your networks uses the DHCP protocol. The netctl package does not provide a graphical user interface, but if you would like to use one, the wifi-menu can be integrated by installing the package dialog. The ppp package can be installed to use Ethernet connections with the PPPoE protocol.

Figure 3.1: A list is much more scannable compared to a text containing the same information.

A digital version does not only automatically offer a search function through browsers and PDF viewers, it is also easier to distribute. Instead of printing and shipping the new version, it can be sent in an e-mail or distributed over links on a website. With PDFs, there is still the possibility that outdated versions get used accidentally. Websites are a better solution, since when they are updated, the older version disappears for the readers. Only the newest version is reachable and the old URL that the readers have bookmarked or can find somewhere on the product still leads them to it. Most people are used to searching answers to their questions online and are used to navigating websites. Web designers have many opportunities that a designer of a text document does not have. Through menus, clickable pictures and links, readers can follow cross-references much quicker than in a text. But there is also the danger of readers getting lost in a chain of links and never finding the information needed, whereas in a text, a reader could just go through the whole text from beginning to end and ensure to get all the available information.[9] So websites have to be carefully designed to provide the reader with an overview of all available information that can be used when he/she is not sure what exactly to look for.

Giving developers and users access to an online platform, where they can contribute their own examples on how to use the software can enhance the official documentation.[10] Researchers Parnin and Treude found that developers voluntarily published tutorials covering 87.9% of the jQuery API methods, mainly by creating their own blogs.[13] This shows that the willingness to help others by sharing knowledge exists and an open platform like a wiki greatly decreases the inhibition threshold for those who would like to contribute but do not have the time to create and host their own platform.

3.2.2 Quick Identification and Utilization of Relevant Components

Developers often only need to change a small part of the system to add a new feature or fix a problem. This part should be quickly identified and it should be possible to read the documentation of this part without having to go through the documentation of the whole system. In the context of microservices, each service should have its own separate documentation and documentations of single endpoints should be delimited from each other, for example by headlines, sections or colored boxes. The same microservice could be used in multiple projects, so it is sensible to keep its documentation self-contained. This way, it only has to be created once and changes in the documentation can be easily inserted into the system documentation of all involved projects.

To figure out whether a part of the system is relevant to a certain task, a developer needs to know its capabilities and how to use them. In the case of a microservice, these would be a description of its overall task in the system and a list of its endpoints with their URL, parameters and return values (mostly HTTP codes indicating either success or failure to complete the requested task) and a short description of each endpoints purpose.
After the relevant part of the system has been identified, there are two possible scenarios for a developer: If the behavior of this part needs to be changed, then the developer needs to know more about its inner details, meaning the source code. This kind of documentation can be found in the form of comments inside the code. To understand these details, seeing the code next to the comment is helpful and since most of the irrelevant parts of code have already been sorted out using the system documentation, the remaining code should not be too much for the developer to keep track of. So it is not necessary to include all these details into the system documentation and as mentioned above, this thesis will not further address them. It assumes that the teams working on the components of the system take care of documenting

their code and their design decisions and choose the tools for doing so themselves, since they are the only people who should be editing the source code of their services.

The second scenario is that the developer needs to use a function, that is offered by this part of the system. In this case, the source code of the function is irrelevant. The developer only needs to know how to use it from the outside. In the context of microservices, the developer sends a request to one of the endpoints of the service, containing parameters with which he can send data and tune the behavior of the service. This is the scenario this thesis concerns itself with.

Since there are often many possible combinations of parameters that can be confusing and off-putting when all of them are listed, some code examples with the most common use cases and some tips on how to avoid common mistakes can speed up the process of understanding[9] and in a lot of cases, a full understanding is not even necessary when the developer can use one of the code examples to solve his problem or only needs to alter one parameter.

Developers like to try out functions to get a better understanding of how they work. This can be faster than reading a text explaining it, since they can use their own examples, showcasing the behavior they are interested in and see the result, which is less ambiguous than a text, that can often be interpreted in multiple ways. Such experiments require working code, that needs to be compiled and integrated into the system. If the documentation offers a possibility to do these experiments without writing code, by just typing in the parameters, a lot of time can be saved in the development process.[14]

A list of functions offered to other components of the system with collapsible items to hide irrelevant details, enriched with examples and a possibility to try out functions is a good way to document single components of the system. Documenting the offered functions and how to use them is called API documentation.

3.2.3 Clear Overview of Complex Systems

Developers need to know how the part of the system they are working on integrates into the whole system. The documentation of the single components is not sufficient, because it does not show their interactions. If an error occurs in one part of the system, there could be a mistake in its code, but there could just as well be wrong information at fault, that it got from another part. Developers need to be able to reconstruct the line of interactions(called service invocation chain) between the different components of the system to find all possible error sources.[3] Also, the team working on one specific part of the system needs to know exactly what the function of their part is and what functions other components of the system already cover, so that there

are not two conflicting components offering the same function.
The documentation should include the names of responsible developers, testers and project managers for each part of the system to give an overview of the division of tasks and to have a contact person for questions or suggestions.

A graphical presentation is best suited to give an overview of the systems interactions, because it provides a two-dimensional view, allowing all interactions to be depicted in one picture. It can also be easily stored in long-term-memory(compared to a list of interactions), giving developers an orientation guide for the project they can keep in mind while working on it.[11]

Web development offers tools like links, block elements and collapsible elements to partition an all-encompassing system documentation into documentation of single components or, in reverse, combine already existing documentation of single components into a system documentation. The suggested solution in this thesis will use the latter approach to fulfill both of the above requirements to isolate documentation of single components and to give a complete overview of the system.

3.2.4 Machine Readability

Documentation should be stored in a machine-readable and widely used format to make it accessible to the wide range of existing tools helping technical writers and developers to work more efficient.
These tools can create styled documents from simple text, releasing the writers from the duty of designing the documentation's appearance, a time-consuming task.[15] If the format is widely used, switching to a different style can be easily done by switching the tool or its configuration. Integrating multiple documents into a documentation of the whole system is easier when they are stored in formats that can be translated into a uniform format and displayed in a uniform style.

Since software is always changed to fit new standards, regulations and user expectations, version control is an important tool in modern software development. Tools like git allow developers to keep track of changes between old and new versions. With a new version of code comes a new version of documentation and even if the code does not change, the documentation might be revised and updated. Things can get lost in the process and if the old version is overwritten, they cannot be retrieved. This problem can easily be prevented with version control, where all committed versions are stored. Another benefit is the acceleration of the reviewing process, since version control software can be used to highlight changes in a document. A machine-readable text format integrates well into version control systems, while styled documents like DOC or PDF are more complicated[10], because their format is a mixture of content and style that can not easily be scanned for differences between the versions.

For developers, there are tools to automatically generate code that is able to invoke functions (or endpoints) described by the documentation.[15] The functions can also be tested with automatically generated tests if the documentation describes what is considered to be a suitable response to the correct input data. These tests are mostly not able to determine if the function is correct in every intended way, but they can check if a function responds at all and whether the response fits the described format of a response that indicates success.[16] These tests can be useful, for example, to regularly test the components of a system in production to see if there is something wrong, for example a service that does not respond to requests because it is overburdened by too many other requests or because the machine it is running on has lost connection to the network or a service responding with errors because it is working on a corrupted data base.

3.2.5 Integration into Development Process

Developers write comments for their code, explaining its purpose. This is also a form of documentation and components of the information inside it is relevant to API or system documentation, so it can be used to generate at least a stub of it that can be revised and extended by a technical writer.[10] If a machine-readable format is followed in the part of the comments that is relevant for API documentation and filled out completely, a full API documentation can be generated from comments.[15] Programming, code commenting and creating documentation for functions available from the outside can be done step by step in the same file. Every time developers change the behavior of a function, the API documentation is right beside the code, reminding them to update this information along with the code and comments and, through the strict format, ensuring the completeness of information.

To ensure that the documentation is always up to date with the newest version of the software, it should be integrated into a deployment pipeline. A deployment pipeline is an ordered list of steps to follow, before a new version of software is deployed to the production server. Generating a new version of documentation can be a part of the pipeline, especially if it can be generated from code comments, styled by a software tool and uploaded to a server using an easy-to-use script. Such a script could be part of a deployment pipeline that is automatically executed every time a new version is committed to a version control server, which is possible with tools like git.[17]

3.2.6 Flexibility

A system documentation that is pieced together from the documentation of its components should be open to different formats. Even though a uniform look is desirable for an easier orientation, different styles might be better suited for different parts of the

system. Also, if the system was built out of components developed by independent teams, who each made documentation choices on their own before, there might not always be time to convert to a uniform format and a simple PDF uploaded to a server is better than no documentation at all, especially if the project is still in its early phase and developers from each team need to know how they can access the functions of the other components.
One possibility to fulfill this requirement is to let each team host their own documentation or upload it to a central server and built the system documentation as a website holding links to each team's individual document, which could be another website, a wiki or any other document displayable by a browser.

3.2.7 Summary

- Seperate Documentation for each Service
  Documentation of each service can be generated and used separately
- Overview of Whole System
  Whole system in one view, showing connections between all services
- Searchable
  Search function available
- Scannable
  Easy to scan for relevant items
- Multiple Levels of Detail
  Choice between completeness and most important information only
- Quick Identification of Relevant Components
  Relevant part of documentation for specific task easy to identify
- Easy to Update
  New versions can easily be delivered to all customers
- Time-Saving for Developers
  Includes tools to speed up programming and testing
- Integration into Development Process
  The creation of documentation is not separated from the rest of the development process
- Flexibility
  Different documentation solutions are combinable
- Accessible Platform
  Platform that allows contributions from outsiders
- Machine Readability
  Stored in a machine-readable format to be processed by software tools

3.3 State of the Art

This thesis focuses on approaches, that fulfil most of the requirements mentioned above. There are many other approaches, like writing documentation in markup language, converting it to HTML and hosting the HTML as a simple website, using wikis or simply printing PDF documents. Some more sophisticated approaches use code annotations and software to convert them into documentation. Comparing all of these methods would deviate too much from the goal. The suggested approach of combining documentation of single microservices by setting links on the visualization of a system of microservices would work with any approach that produces some kind of text document or website that can be hosted on a server, so there are many more possibilities than the ones listed here.

3.3.1 API Documentation for single Microservices

The most sensible way to document a single microservice for a developer that wants to use its functions from the outside, is API documentation. API stands for Application Programming Interface. It describes all functions of one part of a system, that are available to other components of the system and defines how these functions are to be called, what parameters they offer and which answers they give.[2] In the case of microservices, an API lists all endpoints of a service and for each endpoint, it describes the parameters that must or can be given and all possible HTTP messages send as answers. For developers working on the service itself, a more detailed design document is useful, but those internal details are not necessary for an overview of the whole system and should be managed by the team working on this specific service. Therefore, only API documentation methods will be discussed here.

APIs do not describe the interactions between services, they only list endpoints that could be called, not by whom they are called and mostly focus on one service at a time. Therefore, they can't give a complete view of a system of services that communicate with each other in a specific way.

Swagger and OpenAPI

Swagger offers multiple tools to generate, display and use APIs of microservices. They use the OpenAPI format, developed by swagger and currently used as an industry standard, to store and exchange information about APIs.[18]

Machine Readability and Integration into Development Process

OpenAPI specifications(OAS) can be generated by swagger tools from code annotations or by calling its endpoints with user defined requests and evaluating their

responses with Swagger Inspector.[19] If test cases are already defined, Swagger Inspector can be time-saving, but the defined requests need to include all possible parameters and edge-cases or the API specification will be incomplete. It can be used as a basis that can be completed with more information or to check an existing specification for missing information. Code annotations are easier to check for completeness and they can easily be updated with every change of code, since they can be seen right beside it, but the initial annotation takes time and knowledge of the OpenAPI format. See figure 3.2 for an example of what the OpenAPI format looks like as a code annotation for an endpoint.
If code annotations are present, the generation of a new version of documentation with every new version of code, can be part of a deployment pipeline. For example, a script can be added to a github pipeline, so that every time a new version of code is uploaded, a swagger tool generates an OAS file, which is then uploaded to a documentation server that runs SwaggerUI. This way, documenting the new implementation of a service will never be forgotten.

Easy to Update, Scannable, Searchable, Multiple Levels of Detail

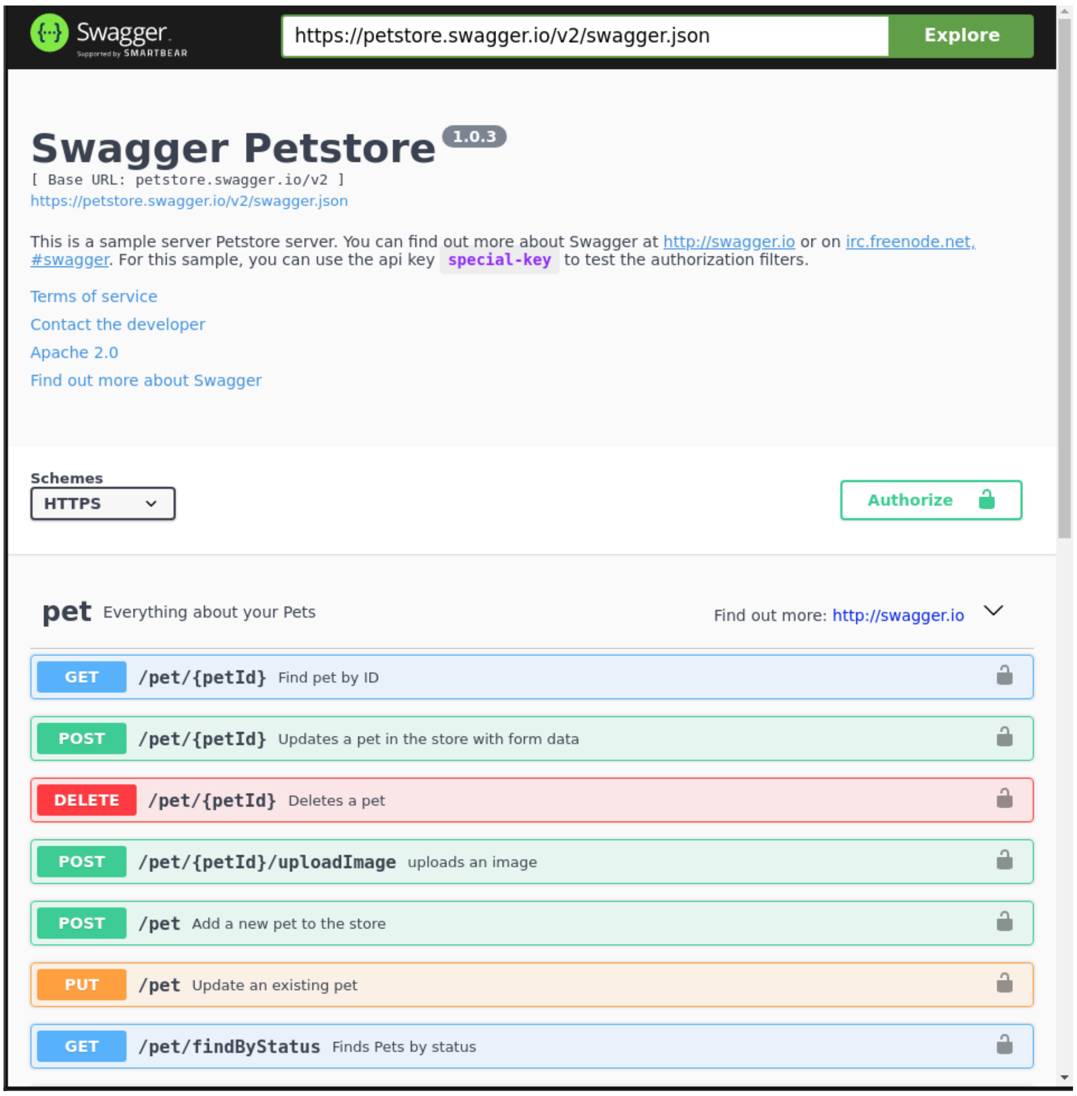
With SwaggerUI, the OAS can be displayed in the form of an interactive website, which can be hosted on a company server or, if the OAS is uploaded to SwaggerHub, can be displayed there. SwaggerUI is basically a fully functional and well documented website that can be downloaded, modified and hosted anywhere. To display a particular OAS, it has to be hosted on a server, which could be SwaggerHub or a private server, and the path to it has to be entered in a search field as a URL or added to the URL of the SwaggerUI website as a query(`?url=`).

Since the documentation is in the form of a website, new versions can be instantly delivered to all users as long as they have a link to it and the URL stays the same.
The interface is easily scannable since it is designed as a number of lists. The metadata, like `Terms of Services` or the type of server that the service is running on are styled in a distinctively different way then the list of endpoints. Endpoints are styled with a colored background and frame, so they are the first thing that catches the users eye. Since the endpoints are mostly what developers are looking for, this is a good way to guide them to the components that are most interesting to them. They are also grouped together by tags, making it easier to narrow the search for relevant endpoints down to a group of endpoints working with the relevant data. If a developer is interested in a function that adds a new customer, it will probably not be found in the group of endpoints with the header `pet`.

All endpoints and the tags grouping the endpoints are collapsible, so users can get a quick overview of the offered endpoints and get a more detailed view on the ones that are relevant to their task. Groups of endpoints that are not relevant, can be collapsed by their tag to get a better overview of the other endpoints that might be relevant to a

```go
// swagger:operation GET /temp getTemp
//
// Returns the current temperatures of the requested sensors of the server
//
// ---
// produces:
// - application/json
// - text/plain
// parameters:
// - name: sensors
//   in: query
//   description: temperature sensors to read
//   required: false
//   type: array
//   items:
//     type: string
//   collectionFormat: csv
// responses:
//   '200':
//     description: An array of TemperatureStat objects and the unit
//     schema:
//       type: object
//       properties:
//         values:
//           type: array
//           items:
//             type: object
//             properties:
//               sensorKey:
//                 type: string
//                 description: sensor ID
//               sensorTemperature:
//                 type: number
//                 description: temperature of sensor
//           description: An array of sensor key and temperature pairs
//         unit:
//           type: string
//   default:
//     description: error
//     schema:
//       type: string
func temp(rw http.ResponseWriter, req *http.Request) {
    rw.Header().Set("Content-Type", "application/json")
    // get the request parameters
    params := req.URL.Query()
    // this is a string
    sensorsStr := params.Get("sensors")
```

Figure 3.2: With code annotations in the OpenAPI format, the documentation is in a machine-readable format and is not separated from the implementation


Swagger
Supported by SMARTBEAR
https://petstore.swagger.io/v2/swagger.json
Explore

Swagger Petstore 1.0.3
[Base URL: petstore.swagger.io/v2]
https://petstore.swagger.io/v2/swagger.json

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

Terms of service
Contact the developer
Apache 2.0
Find out more about Swagger

Schemes
HTTPS
Authorize

pet Everything about your Pets
Find out more: http://swagger.io

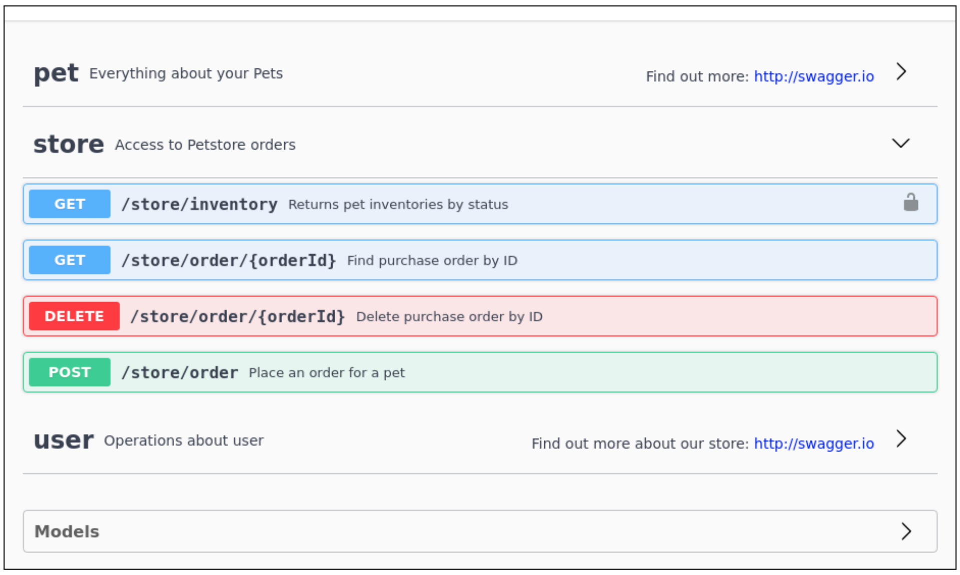
GET /pet/{petId} Find pet by ID
POST /pet/{petId} Updates a pet in the store with form data
DELETE /pet/{petId} Deletes a pet
POST /pet/{petId}/uploadImage uploads an image
POST /pet Add a new pet to the store
PUT /pet Update an existing pet
GET /pet/findByStatus Finds Pets by status

Figure 3.3: The SwaggerUI visualization of an API is easy to scan for relevant information

certain task. For an example of how SwaggerUI looks, see Figure 3.3 and 3.4.

Time-Saving for Developers

The information in the OAS can be used to generate code with Swagger Codegen. For clients, all information is included in the API to generate fully functional code that is able to call an endpoint using the HTTP protocol and sending a syntactically correct request to the correct URL. The code is generated in the form of a function, so instead of researching how to use the HTTP protocol in the programming language used for the client, looking up the address of the endpoint and writing the function, developers only need to call the generated function and look up the meaning of the parameters. Stubs of servers for the described API endpoints can also be generated, but since



pet Everything about your Pets — Find out more: http://swagger.io

store Access to Petstore orders

GET `/store/inventory` Returns pet inventories by status

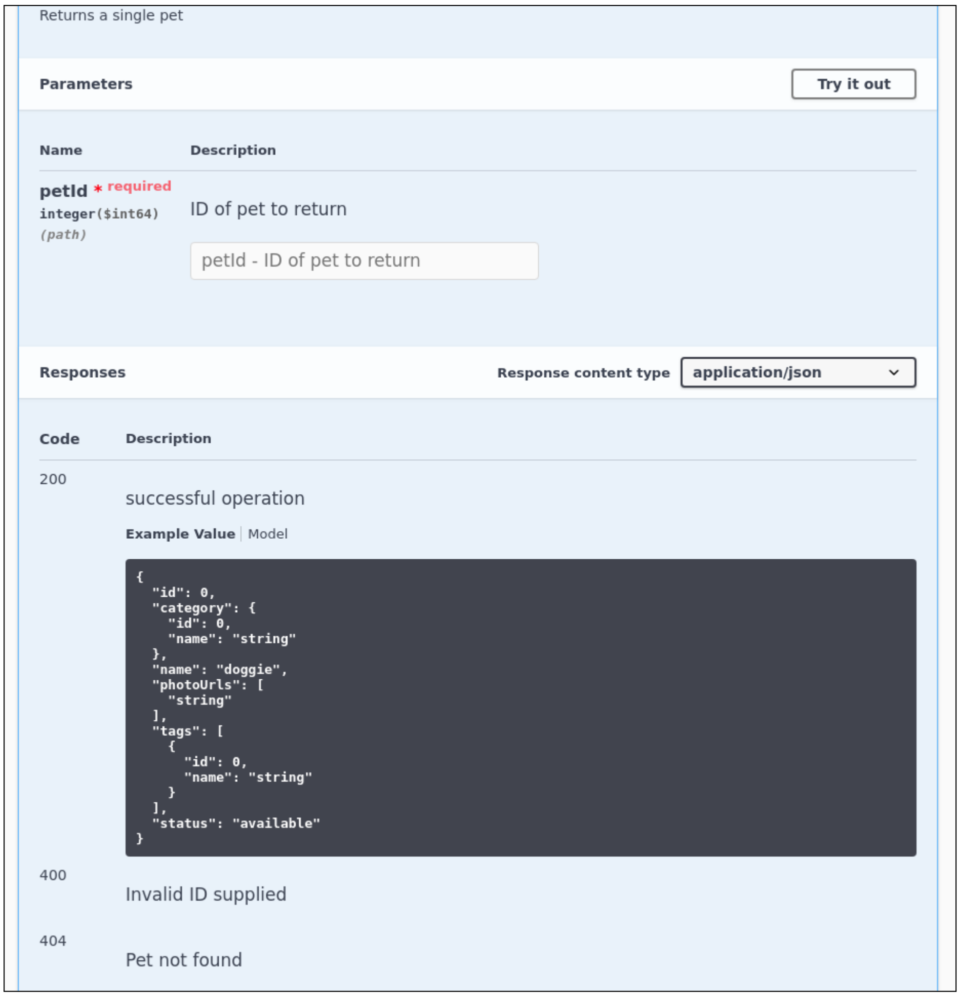
GET `/store/order/{orderId}` Find purchase order by ID

DELETE `/store/order/{orderId}` Delete purchase order by ID

POST `/store/order` Place an order for a pet

user Operations about user — Find out more about our store: http://swagger.io

Models



Returns a single pet

Parameters

Name	Description
petId * required integer($int64) *(path)*	ID of pet to return petId - ID of pet to return

Responses — Response content type: application/json

Code	Description
200	successful operation
400	Invalid ID supplied
404	Pet not found

Example Value | Model

```
{
  "id": 0,
  "category": {
    "id": 0,
    "name": "string"
  },
  "name": "doggie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 0,
      "name": "string"
    }
  ],
  "status": "available"
}
```

Figure 3.4: The elements in SwaggerUI are collapsible to enable readers to narrow the information down to what is relevant to their task

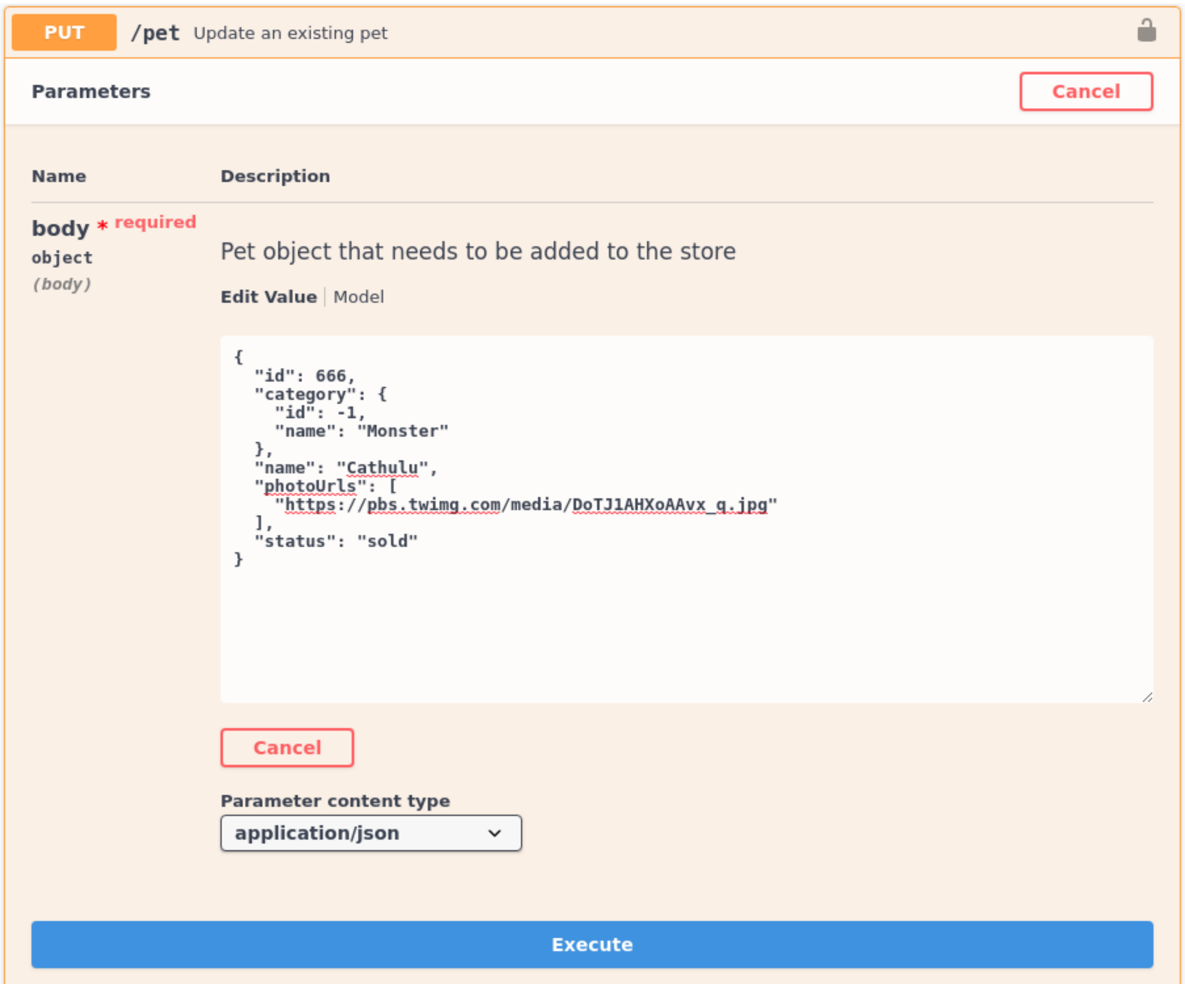
the API does not describe how exactly the function behind an endpoint should be performed, the stub still needs to be filled in by a developer. But having a stub with the right function header, parameters and return values that can receive requests and send responses in the right protocol is time saving, since the developer does not need to start from scratch and does not need to check the API multiple times, for example to avoid a wrong order of parameters.

Endpoints can be tested on the SwaggerUI website without writing any code by clicking on the endpoint, then clicking on the `Try it out` button and filling in the parameters. Example values can be listed in the OpenAPI specification to give developers a starting point that can be edited. These example values will be shown as a default value in the input field, so they don't need to be searched for and copied from somewhere else. See figure 3.5 for an example of how a test of an endpoint looks like. For examples that are not self-explanatory, the model can be viewed by choosing `Model` on top of the field. The elements of the model are collapsible for the sake of clarity. Multiple levels of detail are available on almost all elements of the SwaggerUI interface. After filling in the parameters, the developer can send a real request to the service by clicking on the highlighted `Execute` button. The response of the requested service is shown under the `Execute` button, along with a curl command, which can be copied and used in a console for other testing purposes or script writing. The only prerequisite for this function is that the service is running on a machine that is reachable from the machine running SwaggerUI.

Developers of the documented service are also part of the OpenAPI format, they can be found under `Contact the developer` on the interface and can be added under `contact` in the `info` part of the OAS. This makes it easier for developers to find and contact the responsible developers and use the appropriate e-Mail address to do so.

Accessible Platform

SwaggerHub is an online platform that allows companies and private persons to upload and download their OpenAPI specifications, group them into projects and display them with the build in SwaggerUI tool.[20] Other users of the platform can be added as collaborators with different levels of access to the API specification. They can be viewers, editors or commentators. Editors have the right to make changes to the specification, viewers are only able to view it and commentators are not allowed to edit the specification but leave comments when they find issues.[21] The commentator role makes SwaggerHub an accessible platform, since text based comments can be written without the knowledge of any special format or protocol and adding team members as commentators is no risk to the correctness of the API specification, because they are not able to change it directly. The comments can be regularly checked for rel-



Pet object that needs to be added to the store

```
{
  "id": 666,
  "category": {
    "id": -1,
    "name": "Monster"
  },
  "name": "Cathulu",
  "photoUrls": [
    "https://pbs.twimg.com/media/DoTJ1AHXoAAvx_q.jpg"
  ],
  "status": "sold"
}
```

application/json

Figure 3.5: Endpoints can be tested without writing code by filling in parameter values in the SwaggerUI interface

evant information by the team responsible for maintaining the API and can be used to correct errors, rewrite explanations to clear up misunderstandings and add examples.

Separate Documentation for each Service and Overview of Whole System

Since each service can have its own OAS, there is a possibility to document each part of the system separately. But there is no direct support to combine them into a system documentation. It is possible to list all endpoints inside the system in a single OAS, but since there can only be one host in each OAS, all the service would need to run on the same host. This is contradictory to the concept of microservices, since each service should be able to run independently under any address, as long as the services are able to reach each other. OpenAPI 3 offers the possibility to enter multiple host names, but this is only for redundant hosts, all the endpoints still need to run on both hosts. It is not possible to assign a host to a group of endpoints[22]

There are openly available tools like swagger-combine that are able to combine multiple OAS into one, making it possible to have a separate documentation for each service and combine them together.[23] But it is still a list of all endpoints with no headings to determine which endpoint belongs to which service. However, swagger-combine provide the function to alter the headings of single endpoints, so the name of the service to which the endpoint belongs can be added in front of the endpoints name. Since the endpoints are called by their URL paths, changing the name does not render the OAS useless. How exactly this approach can be implemented and how it can be automated to be conveniently used for any combination of APIs will be discussed in chapter 3.

Still, a combined list of endpoints is not a complete system documentation, since it does not show the interactions between the services, it only shows the services each of them offers, not how they are used. Also, there is no visualization of the system and the list of endpoints would be impossible to fully view or remember in a short glance for most complex systems, contrary to other representations like a graph. OAS files can be grouped into projects on SwaggerHub, but they can only be viewed one at a time and there is still no visualization of the whole system. The same result can be achieved by storing all OAS files in the same folder and naming the folder after the project.

Flexibility

As mentioned above, the OpenAPI format is supported by a foundation including many influential supporters. Tutorials are openly available and many tools outside of Swagger are able to work with the format. The format is language independent, it has the same structure for all programming languages, so every service could be programmed in a different language and the combined documentation would still look uniform. Many Swagger tools are open source, so their code can be viewed and edited by everyone. SwaggerUI is also customizable, a JavaScript layout file can be edited to change the appearance of the website and tutorials on how to do it are openly available.[24] Therefore, Swagger tools can be used in a very flexible way and other forms of documentation can be converted or added in with the use of different tools and some programming skills.

Quick Identification of Relevant Components

Within a single service, relevant endpoints can be quickly identified from the API, but since communication patterns between services are not described in API documentation, there is no possibility to identify dependencies, which are also relevant for developers planning a change in the system. If a service is updated, not only the updated service needs to be tested, but also all the services that use its endpoints and depend on the service behaving in a certain way. Relying solely on API documentation, there is no way to find out which services depend on a certain endpoint, so the components of the system that are relevant to the testing task cannot be identified by it.

✓ Separate Documentation for each Service	✓ Easy to Update
✗ Overview of Whole System	✓ Time-Saving for Developers
✓ Searchable	✓ Integration into Development Process
✓ Scannable	✓ Flexibility
✓ Multiple Levels of Detail	✓ Accessible Platform
✗ Quick Identification of Relevant Services	✓ Machine Readability

Table 3.1: The fulfillment of the system documentation requirements specified in the previous section by Swagger and OpenAPI

Alternative: RAML

RAML is an alternative API format. Just as for OpenAPI, there are tools to visualize the specification as a website and to generate code from it. There are fewer tutorials available, since the community behind it is not as big. Nevertheless, it has supporters, especially in design first projects, because it is open to more details that are necessary in the design phase of a project to have a more clear definition of how the endpoints should be implemented. RAML allows the insertion of any file format into the documentation, like code fragments or examples. It also allows requests and responses to be in other formats like XSD, while OpenAPI only supports the JSON format for them.[25]

Being open to more formats saves time and allows the usage of more different modeling and documentation tools, but it can also be abused to avoid the usage of elements in the documentation format that should be used for clarity or consistency in the graphical representation. It can also lead to the exclusion of software tools that are only able to work with a certain limited format and it makes the development of such tools harder. Since Swagger and OpenAPI are better documented and the inclusion of other data formats is not necessary to show the feasibility of the approach presented in this thesis, it will focus on this combination.

3.3.2 Software System Visualization

UML

The Unified Modeling Language(UML) defines multiple graphical components to represent software system components and their interactions and also formats for how to exchange them between software tools. It was originally designed to visualize object oriented monolithic software, but has since been expanded to offer graphical representations for all kinds of systems and workflows.[26] Since microservices are not always implemented in object oriented languages, a class or object diagram would not be a correct representation.

A deployment diagram, like in figure 3.6 shows the distribution of system components on machines. This could be useful for system administrators, but, as explained in chapter 2, microservices are mostly deployed by orchestrators like Kubernetes, so the deployment of services happens automatically and changes often. Orchestrators should also be able to correct errors in the system by themselves, so there should be no need for administrators or developers to know exactly where a microservice is running. For an overview of the system, the deployment diagram has too many components like machines, the network connections between them, libraries that are not directly a part of the system but are used by its services, multiple instances of the same service and other details that should be hidden behind an orchestrator. Only the people who install and manage the orchestrator should have to know these details and since this thesis focuses on the needs of developers working on the abstraction of the system that orchestrators provide, this sort of diagram would be too overbearing to provide an orientation for developers searching for certain endpoints and trying to understand the division of responsibilities among the services of the system.

Off all the UML models, component diagrams come closest to what could be used in the context of microservices. Components are depicted as boxes with interfaces. Each interface can have multiple circles attached to it, so one interface can offer multiple operations. In the context of microservices, the interfaces could represent a part of the API and the circles could be the single endpoints. Interactions between the components are depicted with a line going out from the client component to the re-

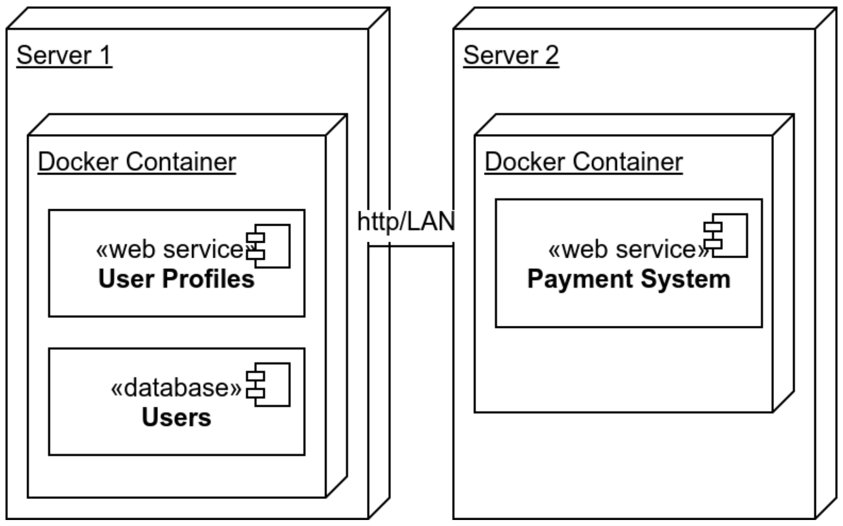


Figure 3.6: A UML deployment diagram can get complicated with only two services

quested operation of the server component, forming a half circle around it. These half circles are not as intuitive as simple arrows and depicting every single endpoint would make the system overview too complex. Single endpoints should only be visible, after the developer has decided, that the corresponding component is relevant to his task and should be collapsible. A component diagram of a complex system can get confusing, since its elements are not intuitive and can show many unnecessary details, like shown in figure 3.6.

Overall, the UML notation offers too many possibilities for the microservice architecture, which aims to design complex systems in a way that makes them as simple as possible. Complex relationships like inheritance, interfaces and so on are not necessary for microservices, because all services are independent of each other and each service is represented by its API, so there is no need for a separate interface. States of the system should not exist, because the endpoints should always behave in the same way, independent of their inner state. Inside the single services, these diagrams can be helpful for the team, but for an overview of the whole system, they are overwhelming and too complex to be processed by simple software tools.

There are many documentation methods for specific programming languages, like class diagrams, but using those in a microservice system will lead to disconnected and inconsistent documentation that will not be combinable into a system documentation. Especially documentation methods that are integrated into a programming environment designed for specific languages, like BlueJ, can not be used in a system that is open to other languages. These approaches can only be used to document the inner details of single microservices for the developers inside the team working on this specific service. But since a service could always be redeveloped in a new language, a language independent method should always be preferred.

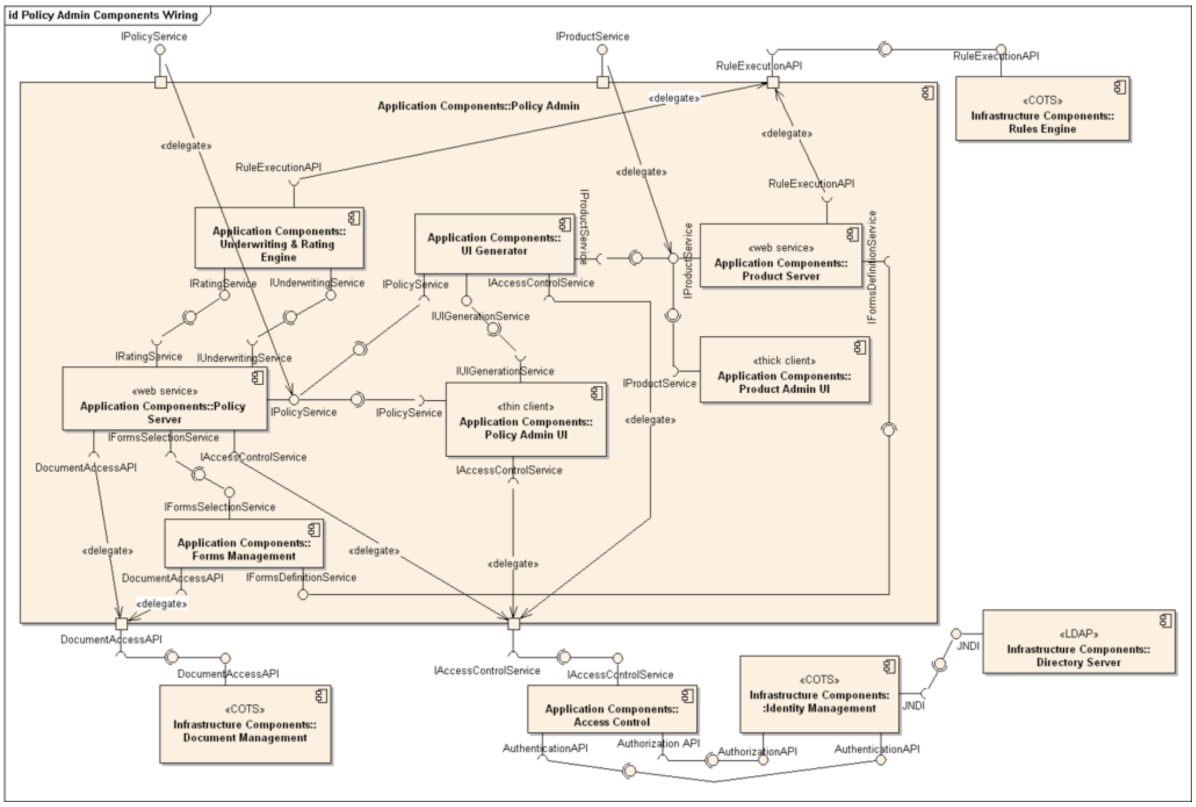


Figure 3.7: UML component diagrams can have multiple interfaces for each component and do not give a clear overview of the system, By Kishorekumar 62 - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5766927

Dependency Graph

A dependency graph is a set of nodes connected by directed edges. An edge from node A to node B indicates that B has to be evaluated before A.[27] In software systems, this means the result of A can only be found after B has produced a result or in other words, A needs B to perform a subtask to fulfill its own task. To do so, A either imports the code inside B and uses its functions directly or, in the context of microservices, service A sends a request to service B, B stores the request in a queue, fulfills the requested task when it is ready and sends a response to service A containing the requested data or information. A can then use this information to finish its own task.

Such a graph can be constructed by analyzing all function calls in the code.[28] But this approach would also collect all function calls inside a single service, making the graph of a big system impossible to display. Also, the requests to endpoints of other services would be recognized as a call to the HTTP library, not to the endpoint or the service that owns the endpoint. All services would be dependent on their HTTP libraries instead of each other.

The service dependency graph proposed for microservice systems in [3] uses the same definition but differentiates service nodes from endpoint nodes and adds undirected edges to represent an ownership relation between a service and its endpoints. See figure 3.8 for an example. The paper proposes to construct this graph from information gathered by the Spring framework, which only works with Java code. All requests from a service to endpoints of other services need to be done using the Spring Feign library and Spring Boot Actuator needs to be active while the system is running to gather information on the requests. The gathered dependencies are not guaranteed to be complete, because the actuator only collects information on the requests that are made during the observed runtime. There could be requests that are only done in rare cases that didn't occur during the observed time period. This method limits the available programming languages to just one and also limits the libraries used to call endpoints and the framework in which the code is developed to one. Existing services in other languages would have to be rewritten and developers who don't know Java would have to be replaced or retrained. Designing a whole project around a documentation method is not feasible, so this method can only be used for projects already designed to use Java and the Spring framework. Switching to a new programming language or framework would mean the documentation method also has to be redesigned.

Instead of observing the running application, code annotations could be added to each request containing information on which endpoint is called. To include all programming languages, the comments would need to follow a defined format, so that graph data from each team that works on a part of the system can be combined into a graph of the whole system. Software tools for this format need to exist for all programming languages used in the project, because every language has its own comment

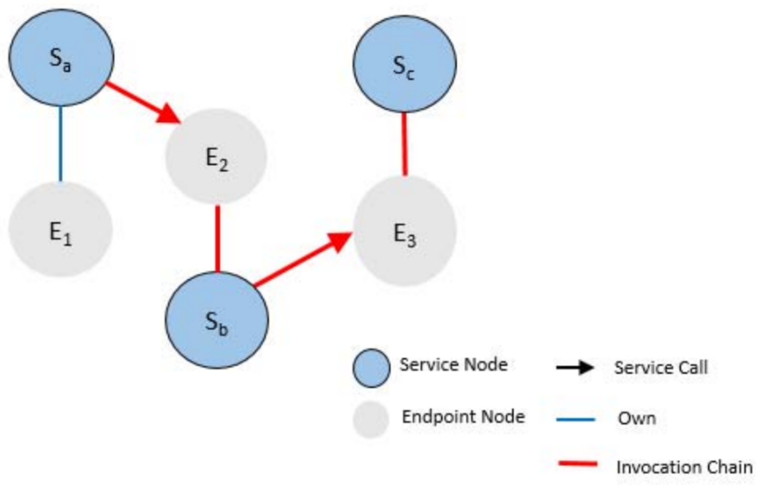


Figure 3.8: Service Dependency Graph developed by [3], picture taken from the paper [3]

syntax. If a request is added without the corresponding comment, it will not show up in the graph, so another tool to monitor the system and detect missing connections should be used to make the graph as complete as possible.

Istio is a tool that can collect the information needed for a service dependency graph by monitoring the network traffic inside a microservice system at runtime.(See chapter 2, Section 2.2.3 for more details on Istio) The collected data can be used by Kiali to draw a graph and display it on an interactive website. As mentioned before, dependency data collected during runtime is not guaranteed to be complete, so relying solely on Istio and Kiali can lead to incomplete documentation. The graph should be stored in a format that can easily be completed by humans with dependency data gathered by other software tools or the knowledge of developers. The Kiali graph can be downloaded from the website in a machine and human readable format that allows it to be altered and displayed by different tools, while the graph generated in [3] is stored in a specific graph database that is not as easy to read and distribute. Istio does not require the services to be written in a certain programming language or use special libraries, which gives developer teams the freedom to implement their service in any way that suits the task. It can be run as a service inside Kubernetes or installed on its own, so orchestrators, discovery services and other tools can also be chosen freely as long as it is possible to inject an Istio sidecar into each service.

Kiali only displays services, not specific endpoints. This makes it harder for developers to find the specific endpoints that a service depends on, but it highly increases the clarity of a system containing more than 10 or even 100 microservices. The Kiali approach is more suited for systems with many small microservices where each service owns few endpoints, while the [3] approach is more suited for systems with fewer microservices where each service owns many endpoints.

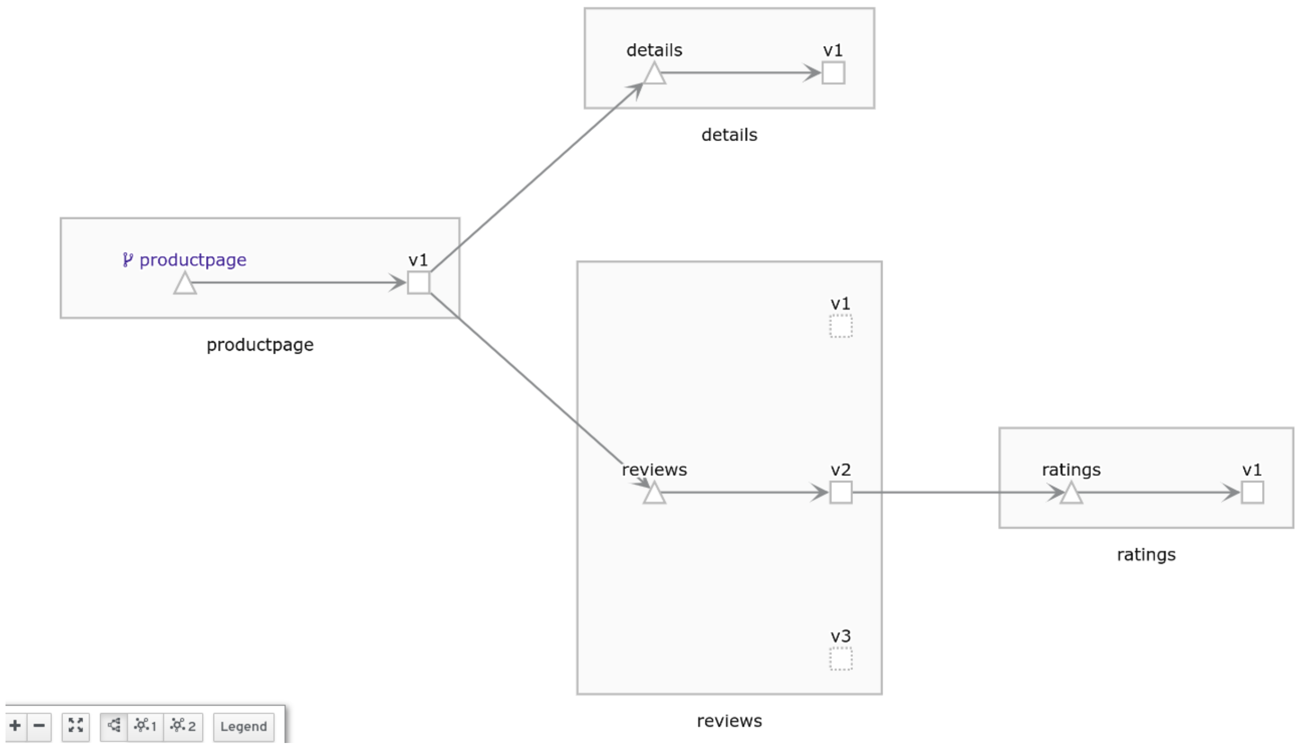


Figure 3.9: A Service Dependency Graph depicted by Kiali

(A dependency graph can also be used to display data dependencies between services and databases and how they are distributed on multiple machines. These kinds of graphs will not be discussed in this thesis)

3.4 Conclusion

API Documentation is a good way to document singly microservices, but it lacks an overview of the whole system and it does not document the communication patterns in a system of cooperating microservices. A dependency graph depicts all the aspects of a microservice system that can not be represented in API documentation. The next two chapters will show how these two methods can be combined to get a full system documentation fulfilling all the requirements listed above. Chapter 4 will focus on combining multiple OpenAPI specifications into one with the goal of constructing a complete and searchable list of all endpoints in the system while still differentiating between the single services. Chapter 5 shows how a dependency graph can be constructed, displayed on a website and combined with API documentation by connecting each node of the graph to a SwaggerUI visualization of the corresponding microservice.

4 Combined API Documentation

4.1 Purpose of this Approach

The approach introduced here allows the developer teams of a microservice system to produce separate API documentation for each service and combine them into one searchable website containing all endpoints in the system. Endpoints belonging to the same service are grouped by headings (called tags in the OpenAPI specification) containing the name of the service. Already existing headings in the single APIs are kept intact and are prefixed with the name of the service. Since all headings are collapsible, an overview of all services in the system in one view is possible, as long as the single service APIs do not contain too many tags (each tag results in a new heading) and as long as there are not more than 20 services. Otherwise, some scrolling and usage of the search function of the browser is necessary.

The display of all endpoints in one website allows developers to get an idea of the division of labor among the services, provided that they and their endpoints are named after their tasks. It can help them find a starting point if they are not sure which service to use for a specific task. For system designers, it can also give an idea whether there are faults in the design or in the communication between teams, for example if one service combines too many tasks or two services fulfill the same function because the teams misunderstood their role.

This approach requires each team to produce documentation in the OpenAPI format and host it on a machine that is reachable from the machine running the combination script. A URL to the OAS(OpenAPI specification) needs to be entered in the config file for the combination tool. To combine all OAS files into one and host this file on a documentation server, only one script needs to be called with the config file as a parameter. This script could be integrated into a deployment pipeline to update the combined documentation every time a new version of a service is deployed, as long as the generation of a new OAS from code annotations is part of the pipeline and is done first and assuming that the config file for the combination step contains all services inside the system.

This approach solves the first shortcoming of API documentation by providing an overview of the whole system while keeping the single services visible. It does not solve the second problem that there is no information on communication patterns between the services. A solution for this will be introduced in chapter 5.

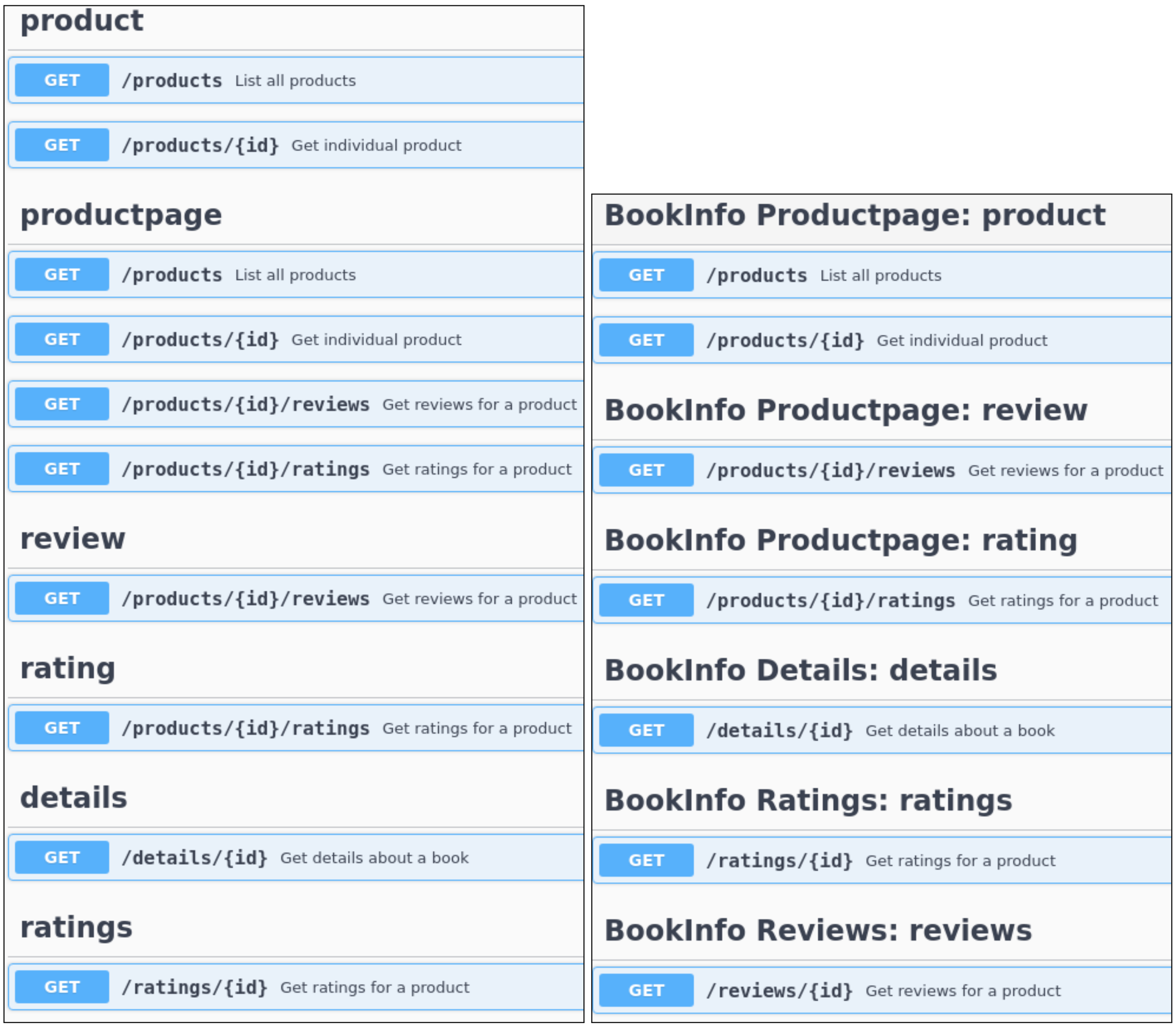
4.2 Integration into Existing Tools

As mentioned in chapter 3, APIs can be described in the OpenAPI format and displayed in the form of a website by SwaggerUI. The OpenAPI specification (OAS) can be generated from code annotations with the help of software tools, in this case go-swagger is used. (See 4.3 for detailed instructions) Each developer team can create the OAS for their own services using any tools they like, display the API with SwaggerUI or another tool that is able to work with the OpenAPI format and host the OAS on their own servers or a central documentation server for the whole project.

Swagger-Combine is a software tool developed by maxdome and openly available on github. (`https://github.com/maxdome/swagger-combine`) It is able to combine multiple OpenAPI specifications into one and is called with a config file as parameter that contains URLs to all specifications to be combined. Tags, which are displayed as headings in SwaggerUI, are kept as they are by default, so it is not possible to distinguish which endpoint belongs to which service. Endpoints, paths and tags can be renamed. To distinguish services, a tag containing the name of the service can be added to the api inside the config file. Every endpoint of the API will get an additional tag, but since other tags will remain, the same endpoint could be displayed in two different groups in SwaggerUI and since the tags are displayed in the order they are defined in[29], all tags of the first service would be displayed in front of the next service, so the result would be confusing. There would be no clear distinction between tags that group part of the endpoints in one service and tags that are named after a service and group all its endpoints, as shown in figure 4.1. There is no possibility in the OpenAPI format to group multiple tags under another tag,[30][31] so if the original tags are to be preserved, the new tag has to be combined with the old ones.

The novel sccp tool developed in this thesis modifies a swagger-combine config file that contains only the URLs to the API specifications of the single services. It uses the URLs to read the specifications and find all tags inside them. For each tag, a rename-command is added to the config file, replacing it with a prefixed tag that starts with the name of the service (or the API title) and connects the old tag with a colon. For example, the tag `review` in the API specification with the title `BookInfo Productpage` is replaced by `BookInfo Productpage: review`. This could also be done by hand, but it would require someone to add a rename-command for each tag in the whole system and if one of the tags is overlooked, it would seem as though a whole new service appeared in the system. sccp is able to find all tags inside the system automatically.

With the new config file, swagger-combine can generate a new OpenAPI specification containing all operations inside the system and tags that group endpoints by their service and also keep the original grouping intact. It can be displayed by SwaggerUI as a website with the tags as collapsible headings.


product

GET /products List all products

GET /products/{id} Get individual product

productpage

GET /products List all products

GET /products/{id} Get individual product

GET /products/{id}/reviews Get reviews for a product

GET /products/{id}/ratings Get ratings for a product

review

GET /products/{id}/reviews Get reviews for a product

rating

GET /products/{id}/ratings Get ratings for a product

details

GET /details/{id} Get details about a book

ratings

GET /ratings/{id} Get ratings for a product

BookInfo Productpage: product

GET /products List all products

GET /products/{id} Get individual product

BookInfo Productpage: review

GET /products/{id}/reviews Get reviews for a product

BookInfo Productpage: rating

GET /products/{id}/ratings Get ratings for a product

BookInfo Details: details

GET /details/{id} Get details about a book

BookInfo Ratings: ratings

GET /ratings/{id} Get ratings for a product

BookInfo Reviews: reviews

GET /reviews/{id} Get reviews for a product

Figure 4.1: Result of swagger-combine with only one ”add tag” command(left) and with renaming commands generated by sccp(right)

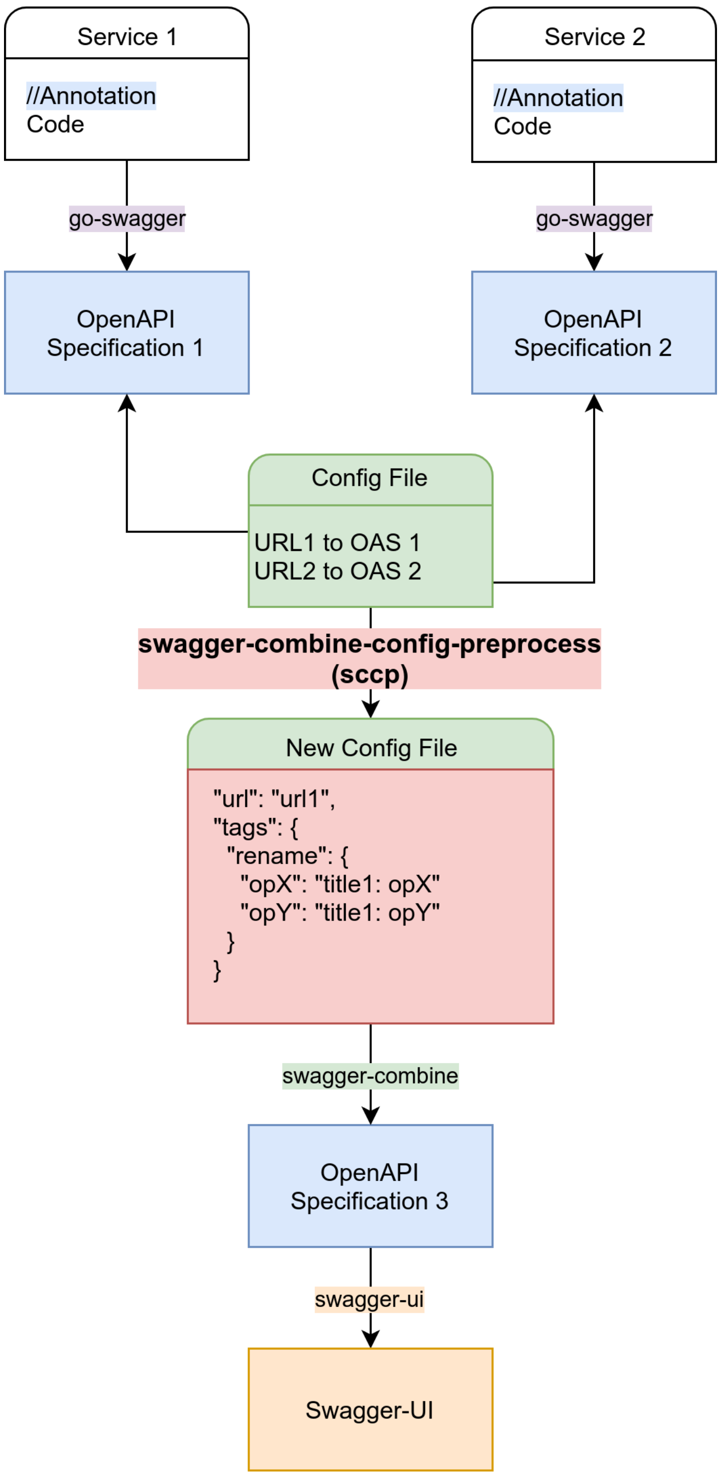


Figure 4.2: Integration of the novel tool sccp into existing API documentation tools

4.3 Usage

4.3.1 Generating Single Service API Specification

To generate an OpenAPI specification, the code can be annotated with the necessary information and processed by a software that is able to generate the OAS from them. Alternatively, tools like the Swagger Inspector can be used, but like discussed in 3.3.1, there is a risk of overlooking endpoints or use-cases and the information is separate from the code, so updating the documentation can easily be forgotten after modifications of the service. Depending on the programming language used, there are different kinds of annotations and different tools to generate the OAS from them. Tutorials on the format are available here:
`https://app.swaggerhub.com/help/tutorials/openapi-3-tutorial#creating-an-api`

As an example, the microservice from chapter 2 will be annotated and the corresponding OAS generated by the go-swagger tool found here:
`https://github.com/go-swagger/go-swagger`
It can be installed with this command:

```
$ go get github.com/go-swagger/go-swagger/cmd/swagger
```

(OpenAPI 2.0 is used here, the newest version is OpenAPI 3.0)
Somewhere in the main file, the following comment has to be added:

```
//go:generate swagger generate spec -o ./swagger.json
```

This adds the command `swagger generate` to the list of commands executed when `go generate` is called in a command line or script. Other files in the same application do not need to include this comment, the go-swagger tool finds them through code paths that lead to them. Code blocks meant for the go-swagger tool are marked by keywords at the end or the beginning of the block. Information about the API is marked by `swagger:meta`:

```
// Package classification Hardware Monitoring API
//
// monitors temperatures of the machine it is running on
//
// Schemes: http
// Host: localhost:8080
// BasePath:
// Version: 0.0
// Contact: Jennifer Nissley<developer@mail.de>
//
// swagger:meta
```

Endpoints can be added with `swagger:operation`, see figure 3.2 for a whole endpoint annotation. The endpoint annotations should be in front of the function that handles a request to that endpoint, since this is where changes to it are made, but it could also be in front of the function, that binds a handler function to the endpoint, like `http.HandleFunc()`:

```
func main() {
  // when the URL <this server>/temp is called, run the function temp
  http.HandleFunc("/temp", temp)
  http.ListenAndServe("0.0.0.0:8080", nil)
}
```

A list of all keywords and examples can be found here:
`https://github.com/go-swagger/go-swagger/blob/master/docs/use/spec.md`
To generate the OAS, go to the project folder with the main file in it and use this command:

```
$ go generate
```

4.3.2 Hosting and Displaying API Specification

The OAS files needs to be accessible by a URL so it can be found by the swagger-combine tool and by developers that want to see the documentation for a specific service. A good practice to achieve this is to let each service deliver their own documentation by adding an endpoint called `/docs`. This ensures, that there is a documentation available for each service and that it can be found by every person that knows where to find the service, because it will have the same host and base path. Since `/docs` should be reserved for a human friendly interface of the documentation, like the SwaggerUI website, the OAS should be offered by an endpoint like `/docs/swagger.json`. In this example, the handler function for the endpoint `/docs/swagger.json` looks like this:

```
func OAS(rw http.ResponseWriter, req *http.Request) {
  content, err := ioutil.ReadFile("swagger.json")
  rw.Header().Set("Content-Type", "application/json")
  fmt.Fprintf(rw, "%s", content)
}
```

In this case, the `swagger.json` file is located in the same directory as the service. It is read in, stored in a string, packed in an HTTP message and send to the client. The client could be a service, a browser or a software tool like swagger-combine.

To provide a visualization of the OAS, the SwaggerUI website also needs to be delivered by the service. It can be downloaded here:

`https://github.com/swagger-api/swagger-ui`
If it is located in the same directory as the service, the handler function looks like this:

```
func docs(rw http.ResponseWriter, req *http.Request) {
  handler := http.FileServer(http.Dir("./swagger-ui"))
  handler.ServeHTTP(rw, req)
}
```

To get SwaggerUI to display the services API specification, the `url` parameter in its `index.html` file needs to be altered to the URL of the OAS. It should look like this:

```
window.onload = function() {
      // Begin Swagger UI call region
      const ui = SwaggerUIBundle({
        url: "/docs/swagger.json",
        dom_id: '#swagger-ui',
        deepLinking: true,
```

This can also be done by redirecting any request to `/docs` to the SwaggerUI website with the `url` query parameter set to the URL of the OAS, like this:

```
func redocs(rw http.ResponseWriter, req *http.Request) {
  fmt.Fprint(rw, "localhost:8080/docs?url=localhost:8080/docs/swagger.json")
  rw.WriteHeader(http.StatusTemporaryRedirect)
}
```

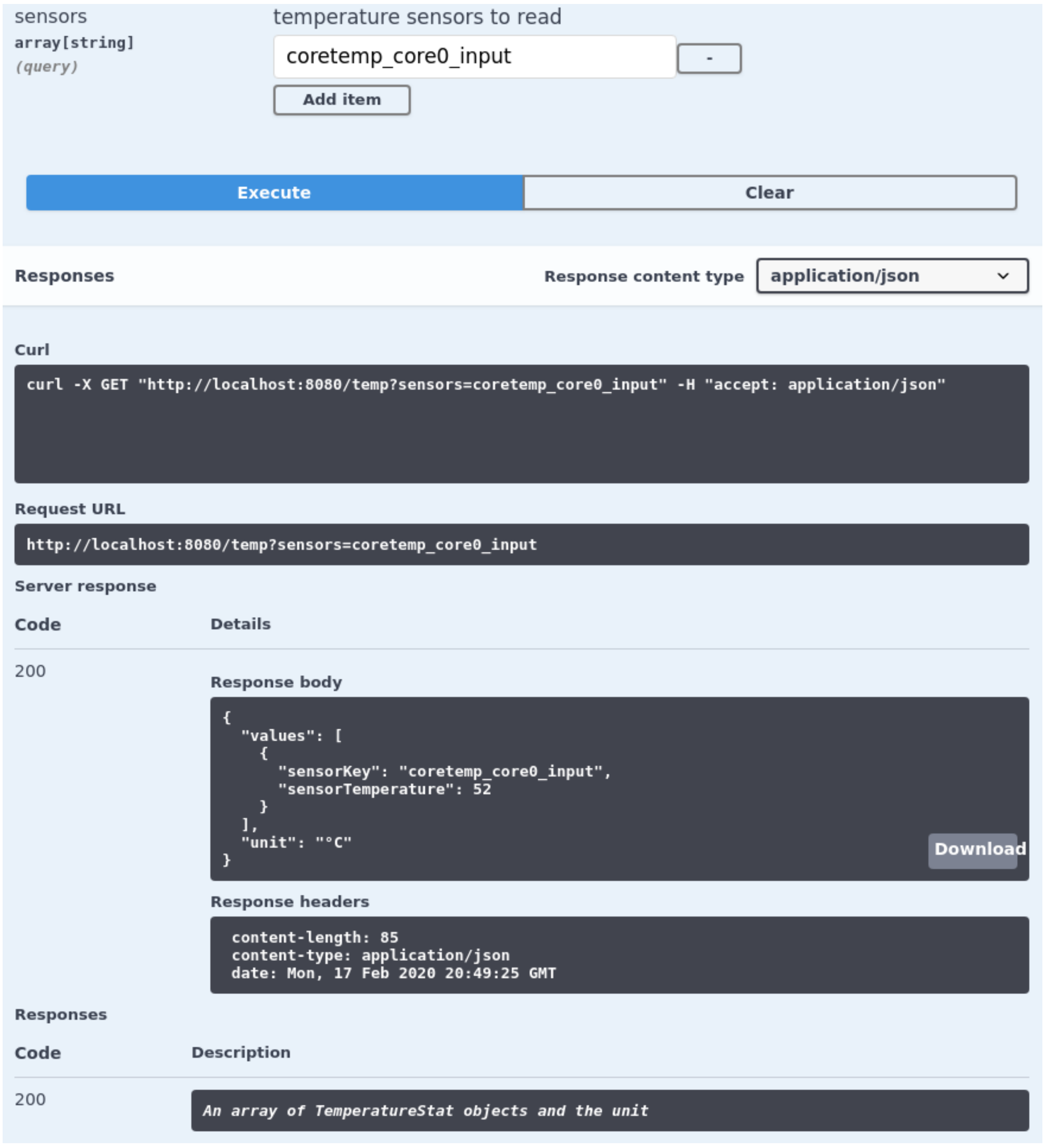
The two handler functions for the documentation endpoints are bound to them in the main function:

```
func main() {
  // when the URL <this server>/temp is called, run the function temp
  http.HandleFunc("/temp", temp)
  // when SwaggerUI website is requested, run the function docs
  http.HandleFunc("/docs/", docs)
  // when the OpenAPI specification is requested, run the function OAS
  http.HandleFunc("/docs/swagger.json", OAS)
  // respond to requests for this server on the port 8080
  http.ListenAndServe("0.0.0.0:8080", nil)
}
```

The documentation can be viewed in a browser under the address:

```
http://localhost:8080/docs/
```

Figure 4.3 shows the resulting SwaggerUI website and the `Try it out` function.



Curl

```
curl -X GET "http://localhost:8080/temp?sensors=coretemp_core0_input" -H "accept: application/json"
```

Request URL

```
http://localhost:8080/temp?sensors=coretemp_core0_input
```

Server response

Code	Details
200	Response body

```
{
  "values": [
    {
      "sensorKey": "coretemp_core0_input",
      "sensorTemperature": 52
    }
  ],
  "unit": "°C"
}
```

Response headers

```
content-length: 85
content-type: application/json
date: Mon, 17 Feb 2020 20:49:25 GMT
```

Responses

Code	Description
200	*An array of TemperatureStat objects and the unit*

Figure 4.3: The Hardware Monitoring service can deliver its own documentation and as long the service is running under the specified Base URL, its endpoints can be executed from the website

4.3.3 Generating and Displaying Combined API Specification

In this step, BookInfo will be used as an example of a microservice system. It was developed by Istio to give developers new to Kubernetes and Istio a small system to experiment with and can be found here:
`https://github.com/istio/istio/tree/master/samples/bookinfo`
Swagger-combine can be downloaded here and needs to be installed for this step:
`https://github.com/maxdome/swagger-combine`
It needs a config file that contains URLs to all OpenAPI specifications in the system, following this format:

```
{
  ”swagger”: ”2.0”,
  ”info”: {
    ”title”: ”Bookinfo Combined Swagger Documentation”,
    ”version”: ”1.0.0”
  },
  ”apis”: [
    {
      ”url”: ”http://localhost/bookinfo_swagger/productpage.yaml”
    },
    {
      ”url”: ”http://localhost/bookinfo_swagger/details.yaml”
    },
    {
      ”url”: ”http://localhost/bookinfo_swagger/ratings.yaml”
    },
    {
      ”url”: ”http://localhost/bookinfo_swagger/reviews.yaml”
    }
  ]
}
```

The original BookInfo project only has one API specification for the service `productpage`. All other services are reachable through the `productpage`. To get a complete documentation of all available endpoints in the system, specifications for the services `details`, `ratings` and `reviews` have been added. Having a separate documentation for all services ensures that the system is expandable and can be redesigned. For example, `productpage` can easily be exchanged without redeveloping the subtasks that are handled by details, reviews and ratings if their operations are accessible to developers directly without the need to use `productpage`.

The sccp tool can be found on the CD under:

```
go/src/github.com/jennissey/thesis-code/swagger_combine_config_preprocess
```

Or on github:
`https://github.com/jennissey/thesis-code/tree/master/swagger_combine_config_preprocess`
To compile the program use this command in the directory containing the main function:

```
$ go build -mod=vendor
```

Located in the folder `swagger_host` is another go program needed to display the resulting combined documentation. This program also needs to be compiled by running the go build command in the `swagger_host` folder.
To combine the API specifications given in the config file, call the script in the sccp project folder with this command:

```
$ ./OAScombine.sh <config file>
```

The script will call sccp to preprocess the config file and add rename-commands for all tags in the OAS. It will then invoke swagger-combine with the new config file and start a program, that hosts the SwaggerUI website, to make the result visible. Since the host has to keep running, the script will not stop automatically. Stopping it will not destroy the new files created. The script creates two new files:

- combined-config.json Result of sccp, contains rename commands
- combinedOAS.json Result of swagger-combine, an OAS containing all endpoints in the system

The new OAS can be viewed directly or in the form of a SwaggerUI website under the URL:
`localhost:8080/docs/`
(If the same URL was used before to display another SwaggerUI website with a different index.html, it might be necessary to reload it without the cache of the browser with a special key combination)

4.4 Implementation of sccp

To read in the old config file, its structure needs to be defined in the programming language of the tool, in this case Go. In doing so, new types are defined for the elements of the config file, enabling developers to store them in variables with the correct type, which makes it clearer what these variables are meant for and enables testing for correct formats. The JSON library uses the type definition to check for errors in the format. This is the type definition used for the config file format:

```
type Config struct {
  Swagger string ‘json:”swagger” yaml:”swagger”‘
  Info Info ‘json:”info” yaml:”info”‘
  APIs []*API ‘json:”apis” yaml:”apis”‘
}

type Info struct {
  Title string ‘json:”title” yaml:”title”‘
  Version string ‘json:”version” yaml:”version”‘
}

type API struct {
  URL string ‘json:”url” yaml:”url”‘
  Tags *Tag ‘json:”tags,omitempty” yaml:”tags,omitempty”‘
  Paths map[string]string ‘json:”paths,omitempty” yaml:”tags,omitempty”‘
}

type Tag struct {
  Rename map[string]string ‘json:”rename,omitempty” yaml:”tags,omitempty”‘
  Add []string ‘json:”add,omitempty” yaml:”tags,omitempty”‘
}
```

Some elements, like the rename-tags, are optional. To communicate this information to the JSON library, the `omitempty` keyword is added to these elements.
With the type defined, the config file can then be used to find the API specifications for the systems services:

```
var config Config
json.Unmarshal(configFileByte, &config)
for _, api := range config.APIs {
    // get the API specification from the URL
    url := api.URL
    oasHTTP, err := http.Get(url)
    // unpack the specfication from the http response
    oasByte, err := ioutil.ReadAll(oasHTTP.Body)
    var oas OAS
    json.Unmarshal(oasByte, &oas)
}
```

(some error handling and the parsing of yaml-files was left out for clarity)

The API specification format also needs to be defined:

```
type OAS struct {
  Info Info ‘json:”info” yaml:”info”‘
  Paths map[string]Path ‘json:”paths” yaml:”paths”‘
}

type Path map[string]Method

type Method struct {
  Tags []string ‘json:”tags” yaml:”tags”‘
}
```

For each URL, the API specification is requested and searched for any tags. The tags that are in use, are listed on the endpoints. They are called paths in the OpenAPI format. Each path can have multiple HTTP methods, like GET, PUT and so on. They hold a list of tags, which are displayed as headings in SwaggerUI. These are the elements that need to be prefixed with the name of the service, which can be found in the info element of the specification. The prefixed tags and their corresponding original tag, are stored in a map, so they can later be used to add rename commands to the config file. The type map will automatically prevent any element from being added twice, which is necessary because multiple endpoints have the same tag, since the tags are used to group multiple endpoints:

```
var oas OAS
json.Unmarshal(oasByte, &oas)

// get the name of the API
prefix := oas.Info.Title + ”: ”
fmt.Printf(” Found API name: %s\n”, prefix

// prefix tags with API name
preTags := map[string]string{}
for _, p := range oas.Paths {
  for _, method := range p {
    for _, tag := range method.Tags {
      preTags[tag] = prefix + tag
    }
  }
}
```

After collecting all tags and storing them with their new prefixed version, Rename commands can be added to the new config file:

```
if api.Tags == nil {
  api.Tags = &Tag{}
}
// if there were no tags found in the API,
// add an Add−command in the config file of swagger−combine
// that adds a tag into the API with the name "default"
// Otherwise, add a Rename−command for each tag
if len(preTags) == 0 {
  if api.Tags.Add == nil {
    api.Tags.Add = make([]string, 0)
  }
  api.Tags.Add = append(api.Tags.Add, prefix+"default")
} else{
  if api.Tags.Rename == nil {
    api.Tags.Rename = make(map[string]string)
  }
  for tag, preTag := range preTags {
    api.Tags.Rename[tag] = preTag
  }
}
```

These two steps are done for all service APIs in the system. A new config file is stored under the name `combined-config.json`:

```
newConfigByte, err := json.MarshalIndent(config, "", " ")
err = ioutil.WriteFile("combined−config"+fileExt, newConfigByte, 0644)
```

5 Combining API Documentation and Dependency Graph

5.1 Purpose of this Approach

In this approach, a website is built to combine API documentation of single services with a dependency graph, giving an overview of the whole system while still being able to focus on single services. The graph provides an overview of the system that is easy to understand and to remember and gives developers an idea of the communication patterns and the division of labor among the services. Through links on each node of the graph, the API documentation of the single service can be viewed on its own.

A combined API documentation containing all service endpoints in the system is also included and can be easily found on a node not connected to the graph and displayed on the top left side with the name of the system as a label. It allows developers who do not know which service a certain function belongs to scroll through all endpoints and use the search function of the browser to find keywords that might lead to it. Other advantages and a more detailed description of the combined API documentation can be found in chapter 4.

The graph can be generated by tools like Kiali or written by hand in the format defined in section 5.3.5). Other methods are possible too, as long as the resulting graphs can be converted to either the Kiali format or the format defined in this thesis, called standard format. It has to be uploaded to the website using the upload buttons and only one graph can be hold at a time. The graph is not guaranteed to be complete and the website does not check whether the dependencies given are correct.

Each node of the graph can hold a link to the API documentation of the corresponding service. The URLs for these links can be specified in the standard graph format. If the graph is uploaded in the Kiali format, it has to be downloaded by the user to add them. Nodes that are equipped with documentation are displayed in green, to make it easy to discern where documentation is missing. The nodes links can lead to any document. The website does not check for a certain format, to give each team the possibility to include any form of documentation they have. New services under development can be documented with tools that are used in the design phase or a simple PDF listing endpoints or offering a short description, which is better than no documentation. This approach requires each team to provide an API documentation that is hosted on a server reachable through a URL.

5.2 Integration into Existing Tools

To generate the graph, the services are deployed in a Kubernetes cluster. Inside the cluster, Istio is intercepting all messages between the services and stores metadata about them. Kiali can use this data to generate a dependency graph. Since many microservice projects are already being managed by Kubernetes and Istio, this does not require many changes in the system. For projects that are managed differently, the graph can be generated by other tools or written by hand and converted to the standard graph format used by the website.

The JavaScript library dagre-d3 is used to render the graph in an SVG image, making it scalable and easy to attach to the DOM(Document Object Model) of the website. The library can be found here:
`https://github.com/dagrejs/dagre-d3`
The OpenAPI format is used to describe the APIs of the single services and the combined API of the system. Each API is displayed with SwaggerUI and hosted on a central documentation server or a server of the team working on the service. A link attached to each node leads to the API documentation of the corresponding service. Using the website with alternative API documentation methods is possible, as long as a link to any document is provided in the graph specification.

As discussed in chapter 4, swagger-combine and the novel tool sccp are used to combine all API specifications into a system API, which is connected to a special system node.

5.3 Usage

5.3.1 Quick Start

Use this command in the Website directory:

```
$ docker run −v $PWD/nginx/nginx.conf:/etc/nginx/nginx.conf −v $PWD:/var/www\
    −it −p 80:80 nginx
```

View the website in the browser on http://localhost and click the button `Upload Standard Graph`. Choose the graph specification located in
`Website/Graphs/graph_format.json`

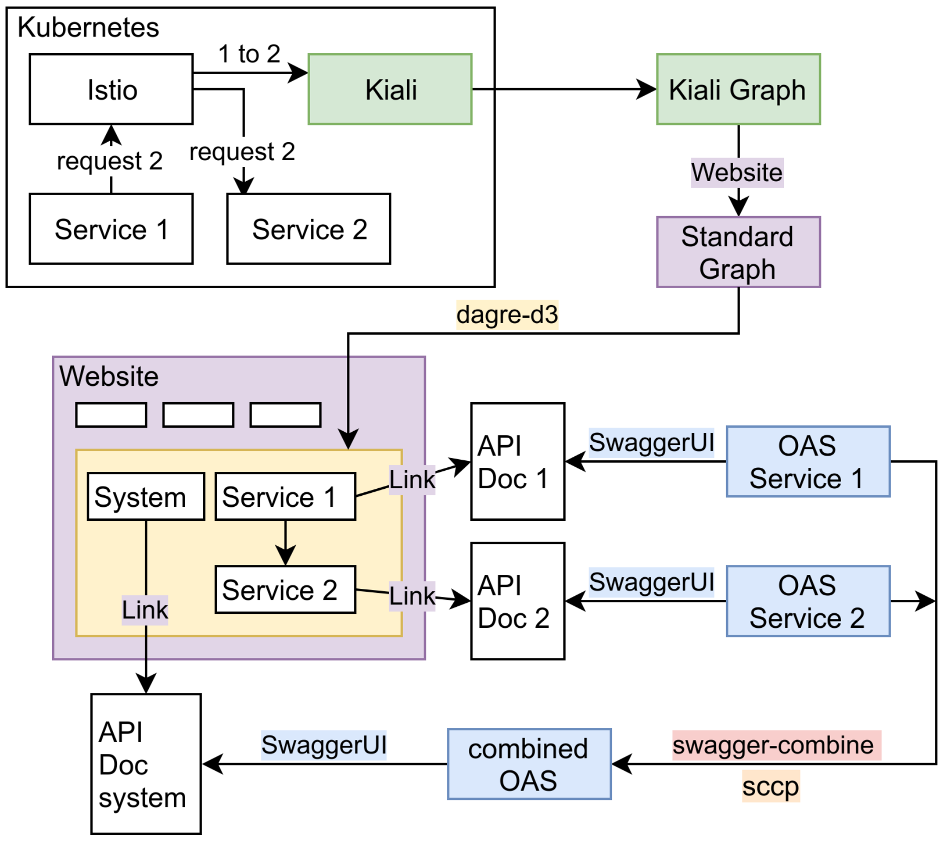


Figure 5.1: Integration of the novel Website into existing Documentation Tools

The website is now displaying a dependency graph of the BookInfo system with clickable nodes that lead to the API documentation of the corresponding services.

5.3.2 Generating Dependency Graph

To generate a dependency graph with Kiali, the system needs to be running in a Kubernetes cluster with Istio and Kiali installed and running as services inside the cluster. (Istio can also be used without Kubernetes, but this approach will not be discussed here). Examples of graph specifications can also be found on the CD in:

```
Website/Graphs
```

Instead of Kubernetes, minikube, a basic and simpler version of Kubernetes meant for small experiments, will be installed in this example. The commands are shown for the operating system Ubuntu.
Firstly, Docker and some other dependencies need to be installed with these commands:

```
$ sudo apt−get install apt−transport−https ca−certificates curl gnupg−agent\
    software−properties−common socat
$ curl −fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt−key add −
$ sudo add−apt−repository ”deb [arch=amd64]\
    https://download.docker.com/linux/ubuntu $(lsb_release −cs) stable”
$ sudo apt−get update
$ sudo apt−get install docker−ce docker−ce−cli containerd.io
```

Minikube can then be installed and started:

```
$ sudo −i
$ curl −L\
   https://storage.googleapis.com/minikube/releases/latest/minikube−linux−amd64\
   > /usr/bin/minikube && chmod +x /usr/bin/minikube
$ minikube start −−vm−driver=none
$ snap install kubectl −−classic
```

Istio and Kiali are installed with these commands:
(A demo is used here to omit the configuration steps needed to install Istio in a specific cluster, the demo also contains Kiali)

```
$ curl −L https://git.io/getLatestIstio | ISTIO_VERSION=1.2.5 sh −
$ cd istio−1.2.5
$ export PATH=$PWD/bin:$PATH
$ for i in install/kubernetes/helm/istio−init/files/crd*yaml; do kubectl apply −f $i;\
    done
$ kubectl apply −f install/kubernetes/istio−demo.yaml
```

BookInfo is a microservice system developed by Istio as an example. It is used here to demonstrate how to use the data collected by Istio to generate a graph in Kiali. To collect data on network communication in the BookInfo system, Istios sidecars need to be injected into all services and the system needs to be running to generate traffic:

```
# enable sidecar injection
$ kubectl label namespace default istio−injection=enabled
# start bookinfo application
$ kubectl apply −f samples/bookinfo/platform/kube/bookinfo.yaml
# enable traffic from outside the cluster to reach bookinfo
$ kubectl apply −f samples/bookinfo/networking/bookinfo−gateway.yaml
# apply traffic rules
$ kubectl apply −f samples/bookinfo/networking/destination−rule−all.yaml
```

The data collected by Istio is automatically forwarded to Kiali. To generate traffic, requests need to be send to the BookInfo application:

```
# store address of kubernetes cluster
$ export INGRESS_PORT=$(kubectl -n istio-system get service\
    istio-ingressgateway -o\
    jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
$ export INGRESS_HOST=$(minikube ip)
$ export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT
# send requests to bookinfo (request a list of all book titles)
$ curl -s http://${GATEWAY_URL}/productpage | grep -o "<title>.*</title>"
```

Kialis port needs to be forwarded to the standard port 80 so that the host machine knows that requests to its standard port should be forwarded to Kiali:

```
$ kubectl -n istio-system port-forward --address=116.203.10.109 $(kubectl -n\
    istio-system get pod -l app=kiali -o jsonpath='{.items[0].metadata.name}')\
    80:20001
```

The graph representation of the collected traffic metadata can be viewed in a browser (URL of the server that minikube is running on), as shown in figure 5.2. The graph

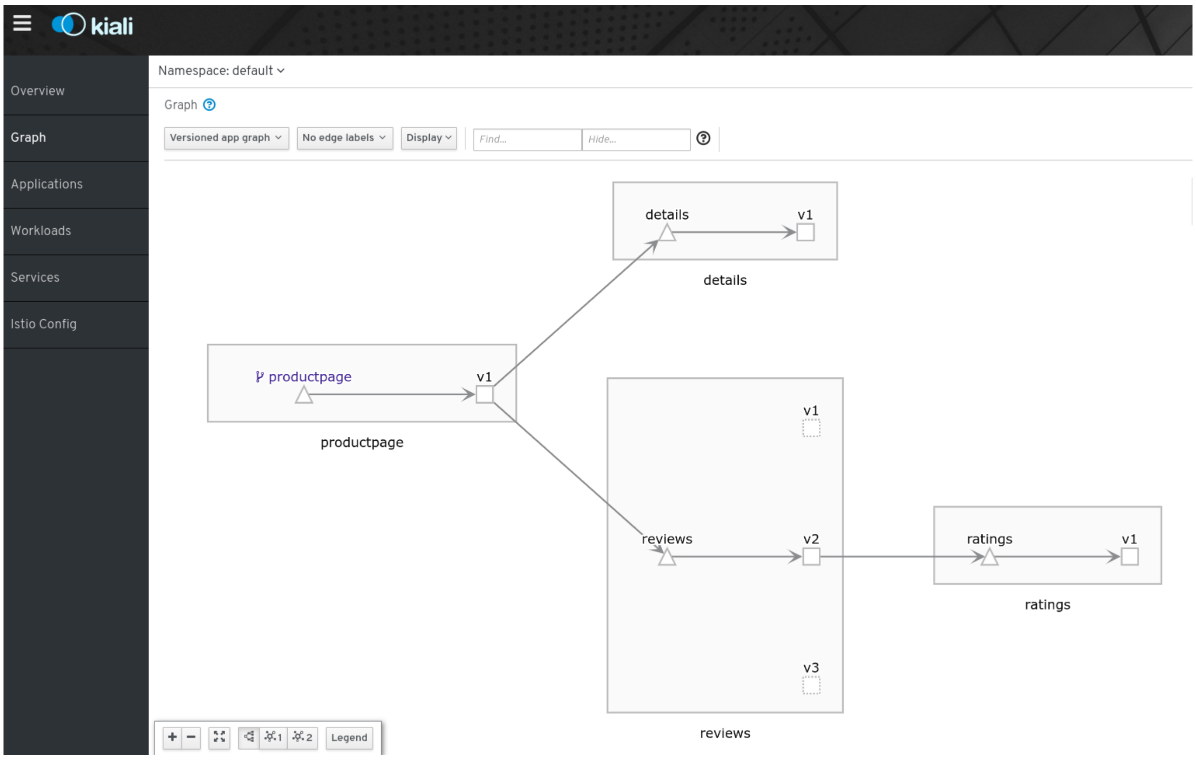
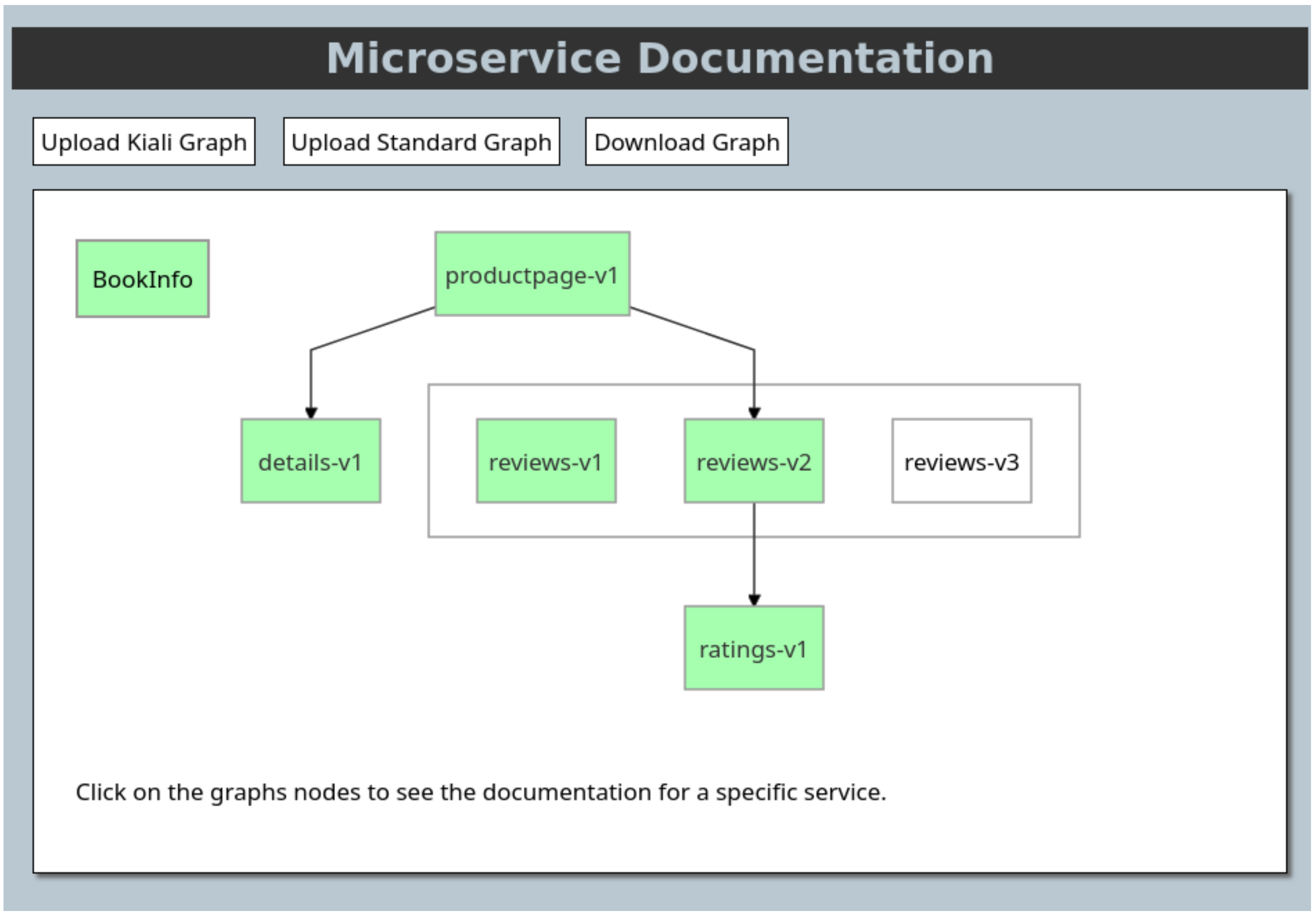


Figure 5.2: The kiali graph can be viewed and downloaded from a website provided by the kiali service

cannot be downloaded from the website, but it can be requested on Kialis API by typing this URL into a browser or curl command:

```
http://<server>/kiali/api/namespaces/graph?namespaces=default
```



Click on the graphs nodes to see the documentation for a specific service.

Figure 5.3: Graph as depicted by the novel Website. Nodes containing links to API Documentation are colored green

5.3.3 Hosting Website

The website can be found in the following directory on the CD:

```
Website/
```

Hosting the website can be done with a single command, given that Docker is installed on the host machine:

```
$ docker run −v $PWD/nginx/nginx.conf:/etc/nginx/nginx.conf −v $PWD:/var/www\
    −it −p 80:80 nginx
```

Docker automatically connects to Docker Hub, where it can download the container `nginx` and run it on the host machine. This container has all the necessary libraries

installed and is configured to run as a server. More details can be found here:
`https://hub.docker.com/_/nginx`
The command above copies all the files inside the current directory to the www-directory inside the nginx container, making them reachable on

```
http://localhost
```

The `index.html` file is automatically displayed first as the homepage.

5.3.4 Uploading Graph

Kiali and standard graphs in JSON format can be uploaded over two buttons on the top of the website. For a definition of the standard graph, see section 5.4.1. The graph is automatically displayed in the center element, it should look like figure 5.3. Links to API documentation and the system node are only displayed for graphs uploaded in the standard format, but every Kiali graph is automatically translated into the standard format on upload and can be downloaded to add links and information on the system and reuploaded afterwards.

5.3.5 Adding missing Dependencies and API Documentation

There is no possibility yet to edit the graph directly on the website. Its specification can be downloaded with a button on top of the website. Whether it was uploaded in Kiali or standard format, the downloaded file will always be in standard format.
A standard graph specification of the bookinfo system is shown in the listing below.

Links to API documentation can be added in the node elements with the keyword link. Missing nodes can be added by adding a node element to the nodes array. They need an id, a name (called app) and a version. Documentation links are optional. Multiple nodes can have the same name, if they are a different version of the same service. Services who share an app name are grouped together in the graph's depiction and a box is drawn around them.

Missing edges can be added in the edges array. They need a source and a target, matching the ids of the nodes they are connecting.

System API documentation can be added by adding a system element to the graph specification and setting its link element to the URL of the documentation.

```
{
  ”nodes”: [
    {
      ”id”: ”47efcb6a38cec94b8f02e15c58ce44df”,
      ”app”: ”details”,
      ”version”: ”v1”,
      ”link” :
        ”http://localhost/swagger−ui/index.html” +
        ”?url=http://localhost/bookinfo_swagger/details.yaml”
    },
    {
      ”id”: ”c7c4074158d3c18e0c780126e50637db”,
      ”app”: ”productpage”,
      ”version”: ”v1”,
      ”link” :
        ”http://localhost/swagger−ui/index.html” +
        ”?url=http://localhost/bookinfo_swagger/productpage.yaml”
    }, ...
  ],
  ”edges”: [
    {
      ”source”: ”11b370489b738638fabddc4f4ce47ebd”,
      ”target”: ”b744c74ec7104950b36f9bf0b47a22fd”
    }, ...
  ],
  ”system” : {
    ”name” : ”BookInfo”,
    ”link” : ”http://localhost/swagger−ui/index.html” +
      ”?url=http://localhost/bookinfo_swagger/combinedOAS.json”
  }
}
```

In this example, there is only one instance of the SwaggerUI website hosted on the same machine as the novel website. The API specifications are also hosted there and included into the SwaggerUI URLs as query parameters to point SwaggerUI to the API it should display.

5.4 Implementation of Website

The website is divided into three files: `index.html`, `styles.css` and `graph.js`. `index.html` is the first file read by the browser and defines the elements of the website, like titles, boxes and buttons. `styles.css` is included into index.html and further defines the style of the elements, like colors, size and shadows. The elements style can change over time, for example they can be made invisible by giving them the same color as the background and can later be made visible again. Such manipulations can be done by JavaScript. It is a programming language that turns a website from a static document with links to an interface with functionality like buttons that change the appearance of the website, data forms that can be used to transfer information from the user to the website and so on. To create such functionality, JavaScript functions are bound to the elements defined in the HTML file and executed when the user interacts with the element. In this case, the JavaScript code can be found in `graph.js`.

Three buttons are defined in the HTML file to upload and download graphs:

```
<input type="file" id="graph_file_kiali" accept=".json" class="filechooser"/>
<label for="graph_file_kiali">Upload Kiali Graph</label>
<input type="file" id="graph_file" accept=".json" class="filechooser"/>
<label for="graph_file">Upload Standard Graph</label>
<a id="download" class="button">Download Graph</a>
```

The upload buttons are defined as input elements that only accept JSON files and let the user choose a file from his own file system. After the user has chosen a file, the input elements report a change-event. These events are bound to the JavaScript functions `upload_kiali_graph()` and `upload_standard_graph()`:

```
document.addEventListener("DOMContentLoaded", function() {
    document.getElementById('graph_file_kiali').addEventListener('change',
        ↪ upload_kiali_graph, false);
  });
```

The graph specification is read in, stored in the local storage of the user's browser to prevent it from disappearing when the website is reloaded, and, if it is uploaded in Kiali format, converted to the standard format.:

```
var file = evt.target.files[0];
 var reader = new FileReader();
reader.onload = function() {
    kiali_graph_string = reader.result;
    window.localStorage.setItem('graph', kiali_graph_string);
    var kiali_graph_obj = JSON.parse(kiali_graph_string);
    graph_obj = convert_kiali_to_standard(kiali_graph_obj);
    draw_graph(graph_obj);
}
reader.readAsText(file);
```

The event `evt` is the change event of the upload button, in other words the upload of the graph specification. A reader object is created to handle the file that is uploaded and is instructed to read the file as a string as soon as the upload process is finished.

Kiali graph specifications are converted to the standard format before rendering:

```
var nodes = kiali_graph_obj.elements.nodes;
for(var i = 0; i < nodes.length; i++){
    node = nodes[i];
    graph_obj.nodes.push({
        id : node.data.id,
        app : node.data.app,
        version : node.data.version
    });
}

var edges = kiali_graph_obj.elements.edges;
for(var i = 0; i < edges.length; i++){
    edge = edges[i];
    graph_obj.edges.push({
        source : edge.data.source,
        target : edge.data.target
    });
}
```

After storing the graph specification in a JavaScript object, it is rendered in the `draw_-graph` function. In the first step, a graph object `g` is created from the graph specification in a format that is readable by the graph rendering library dagre-d3:

```
varnodes = graph_obj.nodes;
var clusters = [];
var nodes_included = [];
for(var i = 0; i < nodes.length; i++){
    node = nodes[i];

    // set the API documentation link of the node
    g.setNode(node.id, { labelType: "html", label: "<a href=" + node.link + ">" +
        ↪ node.app + "-" + node.version + "</a>", width: node.app.length*10+10,
        ↪ height: 40, style: "fill: #afa"});

    // create a cluster if the node is another version of an already existing app
    if(nodes_included.includes(node.app)){
        g.setNode(node.app, {label: node.app, clusterLabelPos: ""});
        clusters.push(node.app);
    }
}

// add the nodes to their corresponding clusters
for(var i = 0; i < nodes.length; i++){
    node = nodes[i];
    if(clusters.includes(node.app)){
        g.setParent(node.id, node.app);
    }
}

// add edges
var edges = graph_obj.edges;
for(var i = 0; i < edges.length; i++){
    edge = edges[i];
    g.setEdge(edge.source, edge.target);
}

dagre.layout(g);
```

The size of nodes is adjusted to the length of their label (the name of the service). Nodes that include a link to API documentation are colored green. Multiple versions of the same service are drawn as separate nodes, but are grouped in a cluster. To realize this, nodes are stored in an array and for every new node, the array is searched for another node with the same app name. If another node with the same name is found, a cluster is created as a node in the graph object `g` with a different style and the app name as name. After all nodes have been added to the graph object, the cluster node is set as a parent of all nodes with the same app name. Dagre-d3 draws the child nodes inside the parent node.

In the second step, the created graph object `g` is rendered by the dagre-d3 library, attached to the `svg` element on the website and centered:

```
var render = new dagreD3.render();
// attach graph to website element
var svg = d3.select("svg");
var svgGroup = svg.append("g");
// draw graph
render(d3.select("svg g"), g);
// center graph
document.getElementById("image").style.marginLeft = "" + -(g.graph().width / 2) +
    ↪ "px";
```

After a graph specification is uploaded, it is bound to the download button:

```
var button = document.getElementById("download");
var file = new Blob([JSON.stringify(graph_obj, undefined, 2)], {type: 'text/plain'});
button.href = URL.createObjectURL(file);
button.download = 'kiali_standard_graph.json';
```

5.4.1 Graph Format Definition

```
Graph {
    nodes : [
        node{
            id : string
            app : string
            version : string
            link : string (URL)
        }
    ],
    edges : [
        edge{
            source : string (node id)
            target : string (node id)
        }
    ],
    system : {
        name : string
        link : string (URL)
    }
}
```

5.5 Fulfillment of Requirements

Requirements fulfilled by SwaggerUI for single services are also fulfilled in this approach, because SwaggerUI is used to document the single services. A detailed discussion on this topic can be found in section 3.3.1.
The system documentation is searchable, if the combined system API documentation is used to search for single endpoints, and it is scannable since the graph lists all services in a clear overview. An overview of the whole system is achieved by showing the whole systems interactions in the dependency graph and listing all endpoints in one document in the combined API documentation.

Focusing on one service at a time is possible by following the link to the single service API documentation. Multiple levels of detail are available in the SwaggerUI documentation and this approach adds a level with the dependency graph, where no details other than the names of the services and their dependence on each other is shown. Services relevant to a certain task can be easily identified by using the combined API documentation to find out which service is implementing a specific function, using the single service documentation to single out the relevant endpoints and use the dependency graph to find out which other services could be executing a subtask of the relevant function. Naming the services and endpoints after the function they fulfill in the system and including a description that includes all the relevant keywords is required

Each team has the freedom to use their favored documentation tools, since the links can lead to any document, making this approach flexible. However, if the teams use different documentation methods, the combined API documentation is not guaranteed to contain all services and the swagger-combine approach is only possible with documents in the OpenAPI format. If the combined API documentation is needed, all teams should agree on one documentation format, but it could be a different one than OpenAPI. Graphs can be generated by Kiali or any other tool, as long as it is possible to convert the specification to either the standard format defined here or the Kiali format. The website could also be altered to display other graphical system documentation, since the dagre-d3 library is able to draw all kinds of graphs.

Both the graph specification and the API specifications are machine-readable and stored in the JSON format, for which libraries exist for most programming languages. The solution can be integrated in the development process by including the OpenAPI specification in code annotations and using a script to automatically update the specification when a new version of code is uploaded to a repository. The combination of these specifications and the upload to their designated URLs could also be included in the script. If the system is running in a kuernetes cluster, a new graph could be generated every day and automatically replace the old graph. But a review of the graph by a human would still be necessary, since Kiali can not guarantee completeness and the API documentation links need to be set. The process of merging new graph data with the old graph while keeping the links intact could be the topic of future work.

✓ Separate Documentation for each Service	✓ Easy to Update
✓ Overview of Whole System	✓ Time-Saving for Developers
✓ Searchable	✓ Integration into Development Process
✓ Scannable	✓ Flexibility
✓ Multiple Levels of Detail	✓ Accessible Platform
✓ Quick Identification of Relevant Services	✓ Machine Readability

6 Future Work

The approach presented in this thesis could be further improved to increase usability, flexibility and completeness.

The graph on the website presented in chapter 5 does not show which endpoints of the services are used by others, it only shows that there is a request from some function of a service to some endpoint of another service. Displaying all endpoints would make the graph overwhelming, but the service nodes could be collapsible to view only one services endpoints at a time. The graph format would need to be expanded to include an array of endpoints for each node. In the graph object used by the rendering library, the service node could be specified as a parent of the endpoints nodes. The information on which endpoint is requested could be found in Istios data on network traffic.

The graph could display more than just dependencies, for example the size of a node could represent the size of the code or the amount of endpoints to give project managers an overview of whether the workload is spread evenly. Edges could have different sizes based on the number of requests send per hour. It should be noted that a high dimensionality can lead to confusion and distraction from more relevant information, so the graph should be kept simple, option to display different aspects if needed can help realize both goals.

The graph specification uploaded to the website could be stored outside the local storage, to allow its download from every machine, not only the machine it was uploaded from. A real backend with a file server would be needed to store the files correctly. This would also allow multiple graph specifications to be stored for a system.

Tools, that are able to combine multiple graph specifications to increase completeness should be added to the website. Also, editing the graph specifications without downloading them could be made possible with an integrated editor or a form where users can add edges or nodes. The swagger-combine approach could be integrated into the website to increase usability.

Adding examples to the API documentation from different sources could be possible with the approach presented in the paper [32]. Examples and descriptions from Stack Overflow are collected, evaluated based on the number of upvotes and included into the documentation of the described services.

Multiple services in the dependency graph could be grouped together under a collapsible node with a label describing their combined function to get a clearer overview of big systems.

A combination tool that is able to combine different API documentation formats could be developed to increase the flexibility and ensure completeness at the same time.

Nodes in the dependency graph could have a function for mouse-over-events showing a short description of the services function to help developers decide whether the service is relevant for them before viewing its API documentation.

7 Conclusion

A list of requirements for system documentation was compiled from literature and the merits and shortcomings of existing modern microservice and system documentation solutions were evaluated based on them. It was concluded that there was no solution providing a full system documentation for microservice systems displaying the APIs of single services and the relationships of the services to each other in one single interface.

A novel tool was created to automate the combination of single service API documentation into a document displaying all services and all their endpoints with the ownership relations between them. The resulting system documentation contains all operations and the communication protocols needed to use them, but it does not show the communication patterns between the services and it does not provide a graphical representation of the system.

The website created to display a dependency graph with links on the nodes that lead to the services API documentation provides a depiction of the whole system, shows the relationships between the services and helps users to navigate through the system documentation by providing the graph as a map of the system.

All the system documentation requirements established in the thesis are at least partially fulfilled by the combined use of the sccp tool and the website created in this thesis, but further development is needed to ensure completeness of the graph, combine both tools in one interface, increase the usability of the websites interface and integrate more formats and software tools established in the industry. With further development, the approach could be useful in the industry.

Bibliography

[1] A. DuVander, “Api consumers want reliability, documentation and community.” https://www.programmableweb.com/news/api-consumers-want-reliability-documentation-and-community/2013/01/07, 2013. [Online; accessed 2019-11-20].

[2] S. Luber, “Was sind Microservices?.” https://www.cloudcomputing-insider.de/was-sind-microservices-a-669758/, 2017. [Online; accessed 2019-12-17].

[3] S. Ma, C. Fan, Y. Chuang, W. Lee, S. Lee, and N. Hsueh, “Using service dependency graph to analyze and test microservices,” in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 02, pp. 81–86, July 2018.

[4] R. McKendrick and S. Gallagher, *Mastering Docker*, ch. 1, pp. 8–12. Packt Publishing Ltd., 2 ed., 2017.

[5] R. Leszko, *Continuous Delivery with Docker and Jenkins*, ch. 2, p. 36. Packt Publishing Ltd., 2 ed., 2017.

[6] T. K. Authors, “Kubernetes Basics.” https://kubernetes.io/docs/tutorials/kubernetes-basics/, 2019. [Online; accessed 2019-12-17].

[7] N. Bhide, “K8s Know-How: Service Discovery and Networking.” https://dzone.com/articles/k8s-knowhow-service-discovery-and-networking, 2019. [Online; accessed 2019-12-17].

[8] S. Luber, “Was ist Istio?.” https://www.cloudcomputing-insider.de/was-ist-istio-a-855674/, 2019. [Online; accessed 2019-12-17].

[9] I. Sommerville, *Software Engineering*, ch. 30. Addison Wesley, 4 ed., 2001. A revised paper made from chapter 30 was used.

[10] A. Etter, *Modern Technical Writing: An Introduction to Software Documentation*. Andrew Etter, 2016.

[11] D. M. McBride and B. Dosher, “A comparison of conscious and automatic memory processes for picture and word stimuli: a process dissociation analysis,” *Consciousness and Cognition*, vol. 11, no. 2, pp. 423–460, 2002.

[12] D. T. Herman, I. G. Broussard, and H. R. Todd, “Intertrial interval and the rate of learning serial order picture material,” *The Journal of General Psychology*, vol. 2, no. 45, pp. 245–254, 1951.

[13] C. Parnin and C. Treude, “Measuring api documentation on the web,” in *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, pp. 25–30, 2011.

[14] B. Jin, S. Sahni, and A. Shevat, *Designing Web APIs: Building APIs that Developers Love*, ch. 4. O’Reilly Media, 4 ed., 2018.

[15] Swagger, “Documenting your existing apis: Api documentation made easy with openapi and swagger.” https://swagger.io/resources/articles/documenting-apis-with-swagger/, 2019. [Online; accessed 2019-10-26].

[16] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, “Wsdl-based automatic test case generation for web services testing,” in *Int. Workshop on Service-Oriented System Engineering*, pp. 207–212, 2005.

[17] G. Inc., “Creating and using ci/cd pipelines.” https://docs.gitlab.com/ee/ci/pipelines.html, 2017. [Online; accessed 2020-02-13].

[18] T. Tam, “Introducing the open api initiative!.” https://swagger.io/blog/api-development/introducing-the-open-api-initiative/, 2015. [Online; accessed 2020-02-13].

[19] Swagger, “Swagger inspector.” https://swagger.io/tools/swagger-inspector/, 2020. [Online; accessed 2020-02-13].

[20] Swagger, “Swaggerhub.” https://swagger.io/tools/swaggerhub/, 2020. [Online; accessed 2020-02-23].

[21] Swagger, “Collaborators.” https://app.swaggerhub.com/help/collaboration/collaborators, 2020. [Online; accessed 2020-02-13].

[22] Swagger, “Api server and base url.” https://swagger.io/docs/specification/api-host-and-base-path/, 2020. [Online; accessed 2020-02-13].

[23] maxdome, “Swagger combine.” https://github.com/maxdome/swagger-combine, 2017. [Online; accessed 2020-02-13].

[24] Swagger, “Creating a custom layout.” https://swagger.io/docs/open-source-tools/swagger-ui/customization/custom-layout/, 2020. [Online; accessed 2020-02-13].

[25] P. Brinkhoff, “Openapi und raml – ein vergleich.” https://www.q-perior.com/blog/openapi-und-raml-ein-vergleich/, 2020. [Online; accessed 2020-02-13].

[26] OMG Unified Modeling Language, *OMG Unified Modeling Language*. version 2.5.1 ed., 2017.

[27] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proceedings of the 14th International Conference on Software Engineering*, 1992.

[28] D. Callahan, A. Carle, M. Hall, and K. Kennedy, "Constructing the procedure call multigraph," in *IEEE Transactions on Software Engineering*, vol. 16, 1990.

[29] T. Johnson, "Step 7: The tags object (openapi tutorial)." https://idratherbewriting.com/learnapidoc/pubapis_openapi_step7_tags_object.html, 2019. [Online; accessed 2020-02-16].

[30] "sub-tags grouping nr.1367." https://github.com/OAI/OpenAPI-Specification/issues/1367, 2020. [Online; accessed 2020-02-16].

[31] Swagger, "Grouping operations with tags." https://swagger.io/docs/specification/grouping-operations-with-tags/, 2020. [Online; accessed 2020-02-16].

[32] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016.

8 Affirmation in lieu of Oath

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht

(Ort, Datum, Unterschrift)

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

(Ort, Datum, Unterschrift)

Hamburg, 24.02.2020 ....