# 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.

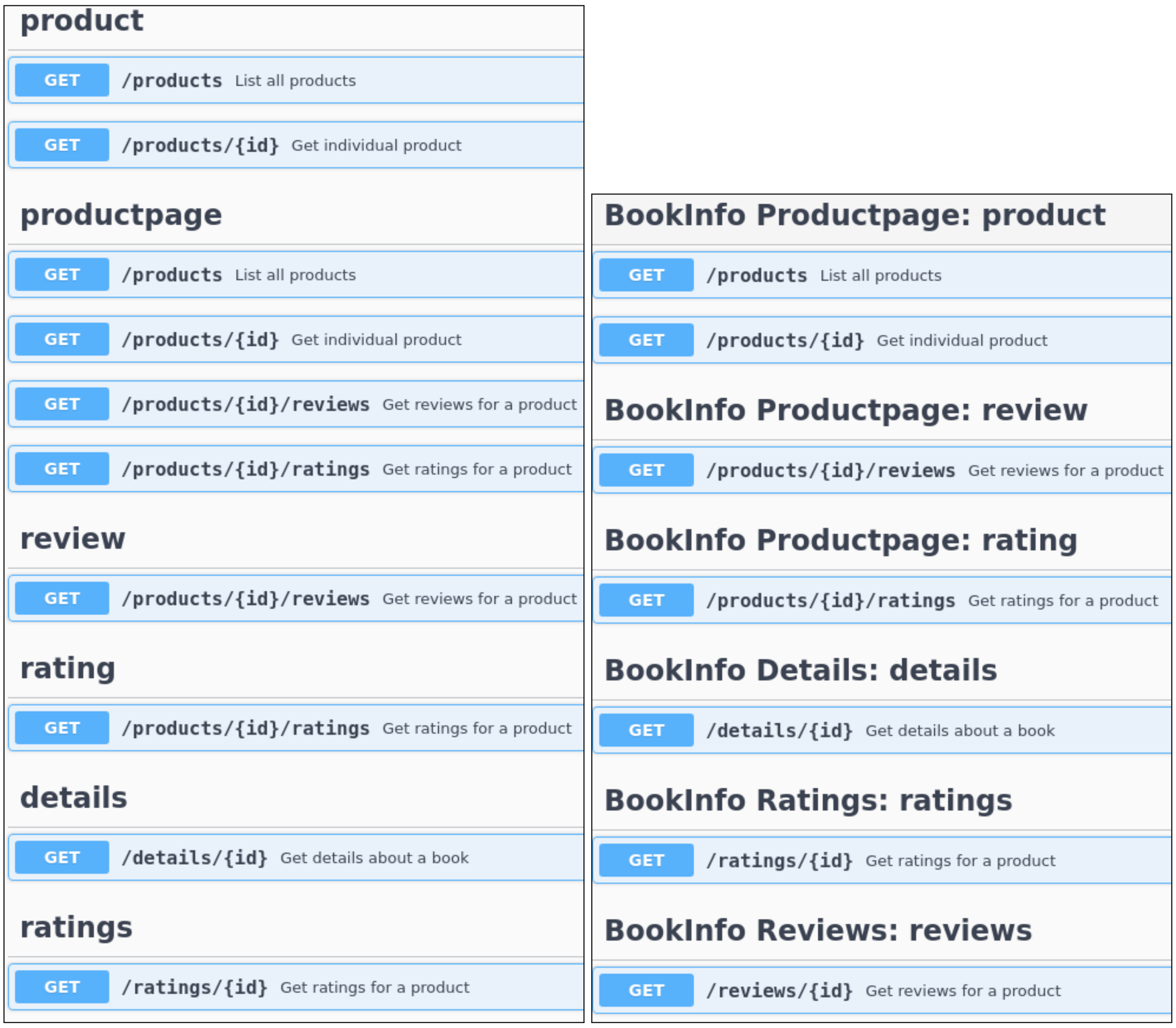
## 4.2 Integration into Existing Tools

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

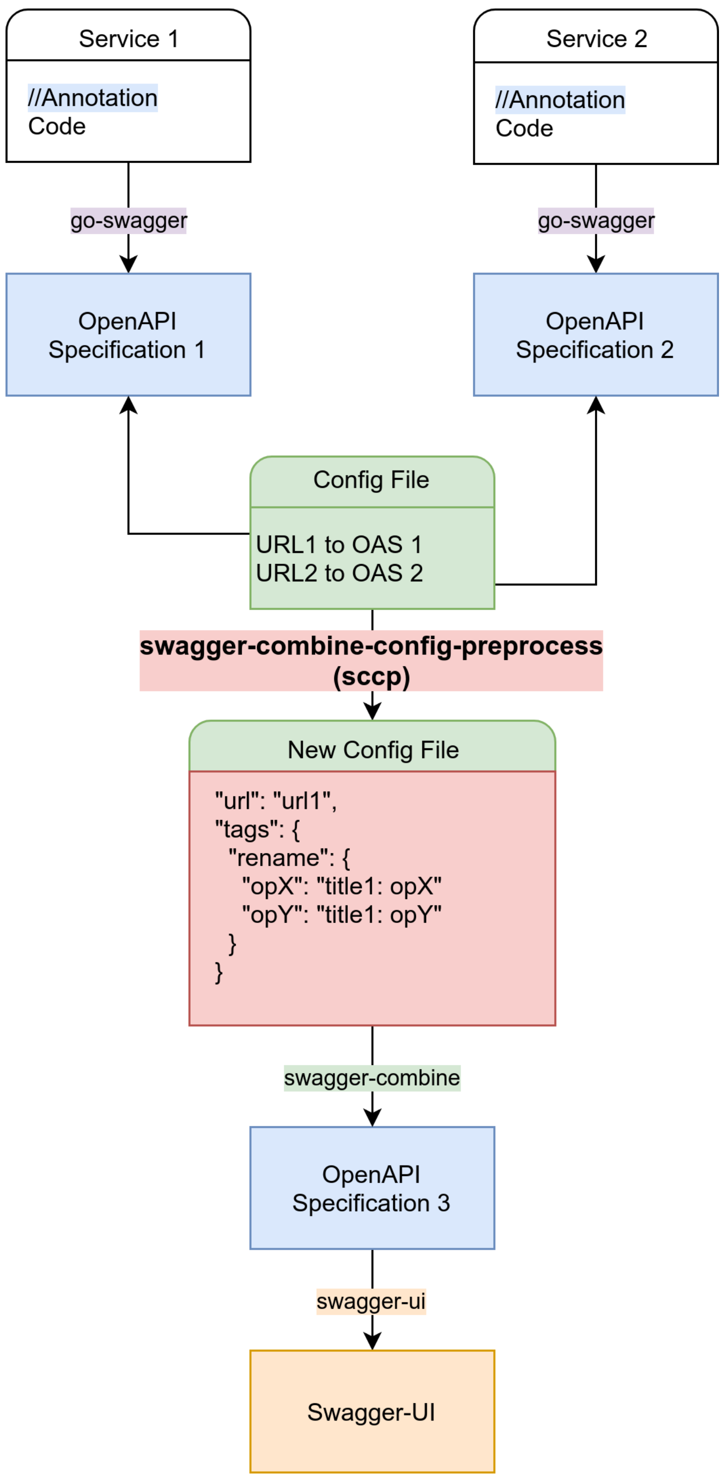
**BookInfo Productpage: rating**
- GET `/products/{id}/ratings` Get ratings for a product

**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)



```
"url": "url1",
"tags": {
  "rename": {
    "opX": "title1: opX"
    "opY": "title1: opY"
  }
}
```

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:

```go
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:

```javascript
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:

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

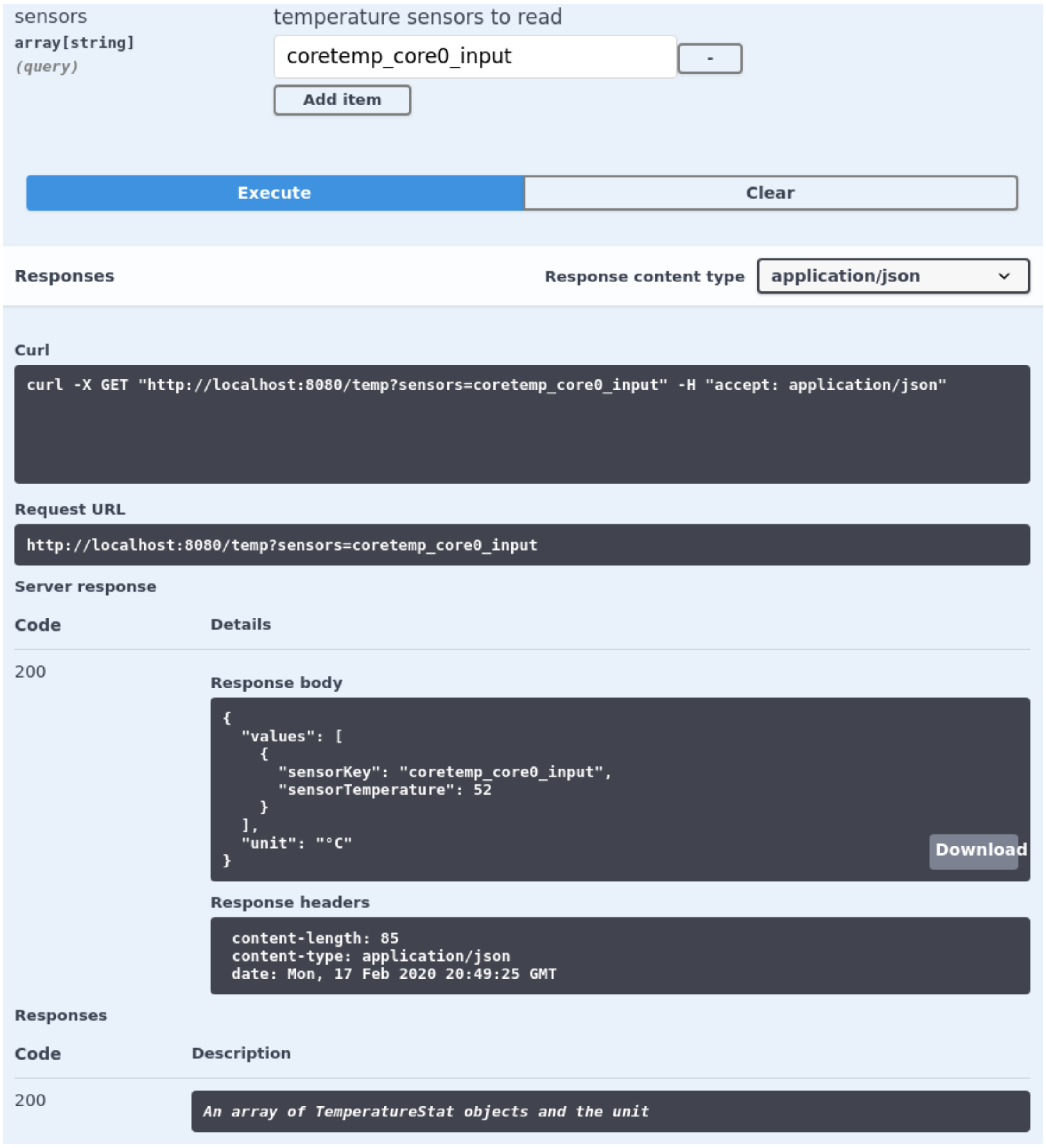
The two handler functions for the documentation endpoints are bound to them in the main function:

```go
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)
```