A Class Concept for Microservices to Manage Dynamic Complexity and Code Reuse

Lukas Reinhardt and Marcus Hilbrich

Technische Universität Chemnitz, Chemnitz, Saxony, Germany
lukas.reinhardt@s2015.tu-chemnitz.de

To manage the complexity of software projects is an ongoing and rising challenge for decades. Complex software is hard to realize, hard to understand, hard to verify, hard to maintain and leads to failing projects [8]. To manage the rising complexity, numerous different strategies are developed. Using microservices is such a strategy. The overall software is devised vertically into services. Idealistically, the services are independent and can be developed independently of each other. Based on our observation, the remaining communication of services is still a challenge and focusing on service development, the complexity of the communication (dynamic complexity) is not managed well. In addition, code sharing [2], abstraction of services, and cross-cutting concerns are also challenges for microservices. In this paper, we present an extension of the microservice concept with object orientation. Based on object orientation, communication of services (dynamic complexity) can be managed explicitly, abstraction can be realized, and code sharing becomes possible. A first evaluation of our concept and a prototype realization indicates that the overall complexity of the software can be reduced significantly, while not harming the microservice development and deployment process.

1 Introduction to our Approach

The idea of our approach is to represent the communication dependencies of services by using the class concept of Object Oriented (OO) programming [2]. This means to explicitly express the already existing communication dependencies of the microservices in the class-description and -interaction. During design and realization, a class contains a set of methods, each of them representing a single microservices, as well as attributes containing the state which will later be processed by the services. In this paper, those special classes will be called *Microservice Classes (MCs* and similar to that their methods will be called *Microservice Methods (MMs.* Those MCs are used to generate the microservices either during compile time or at development time after writing them. This generation process consists of two steps as shown in Figure 1.

Step 1: For each existing MM in the MC, a microservice is generated. In order to accomplish that, Microservice-specific code is added to the existing logic of the MM. This includes code for request handling, authentication, error handling and sending back the response. Depending on the implementation of the generation process this step could also allow the developers to add their own custom code for other cross-cutting concerns as well. Due to the generation of code for cross-cutting concerns in this step MMs only have to contain the actual business logic of the services.

Step 2: The business logic of the MMs is replaced by the calling logic for the corresponding services. Since the altered methods/classes are the interface through which the services are called, we named them *Interface Methods/Classes (IMs/ICs).* The logic for calling the microservices includes the creation of a transfer object containing the specific instance of the IC (the state), the transmission of the transfer object to the service, and the response handling.

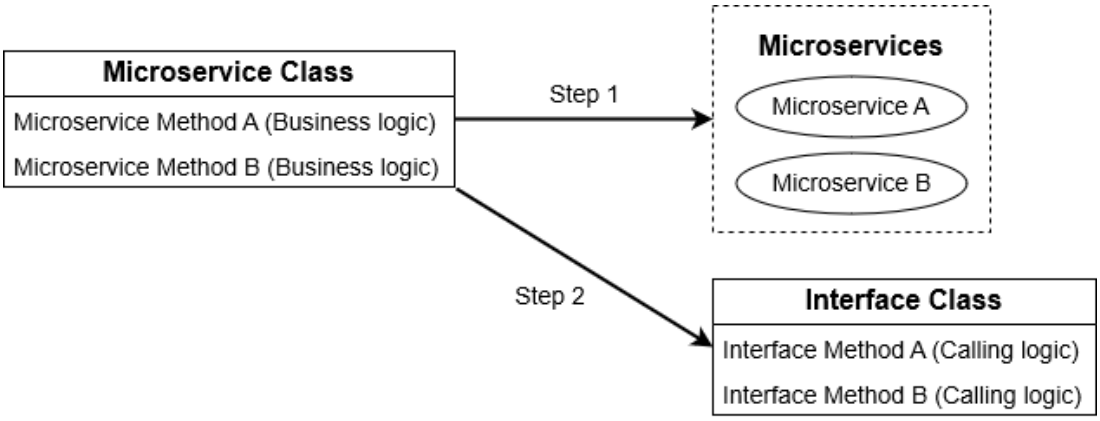

Figure 1: Overview of the generation process of microservices from MCs

Aside from the possibility to inject custom code for cross-cutting concerns in the first generation step, there is also another way in which this approach can address the question on how to handle code reuse for microservices. If MCs allow defining Non-MMs, those methods can contain code which is used by multiple different MMs within the same class. This means the code of those Non-MMs can be reused by multiple different microservices as long as the code of every referenced Non-MMs is also made available inside the microservice. Since an instance of the IC is sent to the microservice, the services already have access to the methods of this object. In practice this process could look like this:

1. The transfer object containing the instance of the IC is serialized to *JSON* or *XML* and sent to the service
2. The service receives the transfer object and deserializes it including the instance of the IC
3. The service now has access to the Non-MMs of the instance of the IC

Another main advantage of our approach (as introduced by Hilbrich at. al [2]) is that a class (a set of microservices) can represent real-world objects (e.g. transactions, or other shared communication constructs). In this case, a class is used to describe the shared information of a set of services as well as their behavior and communication structure. By that, the already existing dependencies between services at development time can be expressed explicitly. This allows to handle and manage dynamic complexity present in the code by using already known methods of abstraction. In our case, this is realized by using classification and explicit description.

2 A Prototype Implementation using Azure Functions

A first prototype implementation for our approach was created by Reinhardt [7] using Microsoft Azure Functions [4] as microservice platform and C# as programming language [6]. This prototype indicates that the proposed approach is not only theoretically possible but can also be realized in practice. Since Azure Functions are used to realize microservices, the two terms microservice and (Azure) Function are used interchangeably in this section. Additionally, the

term attribute does not refer to the OO concept but to a special language construct of C# similar to annotations in Java (see [3] for more information).

The created prototype consists of two parts: a library with classes necessary for writing the MCs and a tool to generate the microservices from those classes. With the prototype, MMs are marked by a *Microservice-attribute* (included in the library). Additionally, to distinguishing between MMs and Non-MMs this attribute also has parameters providing Function-specific information necessary for the generation process. Those parameters are the name of the Function, the HTTP method used for calling the service, and optional authorization information for executing the Function. An example of how such a MM could look like is listed in Listing1.

```
[Microservice("CreateToDoService", FunctionHttpMethod.POST, "test")]
public void CreateToDo()
{
  //Logic for creating a new ToDo
}
```

Listing 1: Example on how a MM could look like using the prototype

All classes containing MMs are automatically considered MCs. After writing the MCs, the tool can be used to generate the microservice out of these classes following the above described two generation steps. In step 1 of the generation process the microservices are generated from the MMs of the MCs. In this step additional code for the following concerns is added by the tool to the microservices: request handling, logging, error handling, and creation and sending of the response. After this step, the microservice created from the method shown in Listing 1 now follows the schema presented in Listing 2.

```
[FunctionName("ToDo_CreateToDoService")]
public static HttpResponseMessage Run(
  [HttpTrigger(AuthorizationLevel.Function, "post")]HttpRequest req,
  TraceWritter log)
{
  string content = new StreamReader(req.Body).ReadToEnd();
  ObjectMethodRequest<ToDo> request = JsonConvert
    .DeserializeObject<ObjectMethodRequest<ToDo>>(content);

  if(request == null)
  {
    //Error handling/logging for wrong request type
    ...
  }
  ToDo requestObject = request.ObjectModel;
  if(requestObject == null)
  {
    //Error handling/logging for missing instance of IC
    ...
  }

  try
  {
    //Execute logic of original MM and send response
```

```
    ...
  }
  catch(Exception ex)
  {
    //Error handling/logging for exception in original logic
  }
}
```

Listing 2: Schema of the microservice generated from the method shown in Figure 1

For simplicity, the prototype does not allow the definition and injection of custom code for other cross-cutting concerns yet. After step 1 the MM is altered in step 2 to now contain the calling logic for the generated microservice. In the presented example the altered MM is transformed into the schema shown in Listing 3.

```
public void CreateToDo()
{
  //Create request for the Function (transerfer object)
  ObjectMethodRequest<ToDo> request = new ObjectMethodRequest<ToDo>()
  {
    ObjectModel = this,
    Uri = "...",
    Method = HttpMethod.Post
  };

  //Send request to the Function
  Task task = Task.Run(async () => await request.ExecuteAsync());

  //Wait for completion
  task.Wait();
}
```

Listing 3: Schema of the microservice generated from the method shown in Figure 1

After the microservices are generated, they only have to be deployed to an existing Azure Functions App.

3 A First Evaluation of the Approach

The first evaluation [7] was done by implementing a todo application as a microservice project. The application was implemented twice, once using the created prototype following the OO concept and once without the prototype using Azure Functions as microservices. While implementing these two test projects the necessary LoC (Lines of Code) and SLoC (LoC excluding comments) were counted with the following results:

- While using the created prototype, the required LoC were reduced by 72 %
- While using the created prototype, the required SLoC were reduced by 78 %

The vast reduction of the required lines of code strongly indicates a reduction of development expenses as well. This is especially true since the argument that sometimes fewer lines of code with a high-quality require more development expenses than more lines of code with a lesser

quality does not apply in this case. This is simply due to the fact that the required LoC/SloC with the prototype are a subset of the required LoC/SLoC without the prototype. The likely reduction of development expenses can be interpreted as a positive side effect of the approach allowing code reuse and the prototype automatically generating code for cross-cutting concerns.

4 Conclusions

It is clear, that our approach of extending microservices with a class concept sounds contradicting to the original microservice approach. Never the less, it also addresses open challenges such as management of dynamic complexity, code reuse or integration of cross-cutting concerns and provides quite optimistic evaluation results. Nevertheless, whether our approach can be helpful for microservices is still an open question and needs more discussions with experts of the community as well as with developers, and programing language experts. This paper is one step to keep this discussion going and to get to a more in-depth understanding of how to tackle the open challenges of the microservice concept in general.

References

[1] Manfred Sneps-Sneppe Dmitry Namiot. On micro-services architecture. *International Journal of Open Information Technologies*, 2014.

[2] Marcus Hilbrich and Markus Frank. Abstract fog in the bottle - trends of computing in history and future. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 519–522, Aug 2018.

[3] Microsoft. Attributes (c#). visited on 2019-01-20 15:20.

[4] Microsoft. Azure functions documentation. visited on 2018-11-28 10:20.

[5] Microsoft. Azure functions tools for visual studio. visited on 2019-01-20 17:00.

[6] Microsoft. C# guide. visited on 2018-11-28 10:20.

[7] Lukas Reinhardt. Ein objektorientierter Ansatz zum Generieren von Microservices : Erstellung eines Prototyps. Bachelorsthesis, Technische Universität Chemnitz, Germany, 2018.

[8] Chris Sauer, Andrew Gemino, and Blaize Horner Reich. The impact of size and volatility on it project performance. *Commun. ACM*, 50(11):79–84, November 2007.