How to Synchronize Microservices

Sebastian Copei[1] and Albert Zündorf[1]

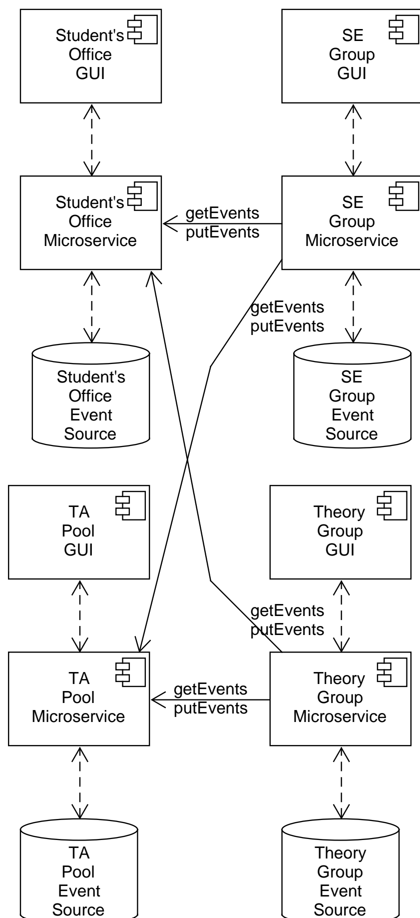
Kassel University, Germany
[sco|zuendorf]@uni-kassel.de

Abstract

We start by splitting an University wide data model for student data into four example microservices. Then we address the problem of data synchronisation for these microservices, systematically.

1 Introduction

Figure 1: Student Affairs Microservices



Our example employs four microservices that all deal with student data at Kassel University, cf. Fig. 1. The Students' Office deals with course programs and all the examinations of the students. The SE Group deals e.g. with assignments in the modelling course. The Theory Group provides a specific grading scheme for seminar presentations. And the two research groups exchange data on Teaching Assistance students via the TA Pool. Each microservice is developed independently and uses its own bounded context data model [2]. As shown in Fig. 1 the microservices use Event Sourcing [2] to store data persistently. Each microservice also provides an API that is used by the corresponding GUIs as well as for loading and logging events as well as for the synchronisation of the microservices. For example, each time a student enrolls for a certain examination within the Students' Office, a corresponding event is raised and added to the Students' Office's Event Source. At any time, e.g. the SE Group may issue a getEvents request causing the Students' Office to respond with all *studentEnrolled* events referring to courses run by the SE Group. The SE Group may now do the grading of these students with the help of the students' performance data gathered locally. Each grading operation will raise and record a *studentGraded* event within the SE Group microservice. After the grading, the SE Group may submit the *studentGraded* events (that of course include the achieved grades) to the Students' Office via a *putEvents* request.

Similarly, the SE Group may hire some of its (excellent) students as teaching assistance. The corresponding *studentHired* event may then be send to the TA Pool. Then the Theory Group may retrieve all *studentHired* events. Thus the research groups may avoid to hire the same student twice.

2 General Event Sourcing Requirements

First of all we propose that there shall be no difference between API calls via a GUI or issued by loading an event from the Event Source or by applying an Event received via a REST request. Of course each microservice has full control which external events it accepts for application to its internal model. However, once an external event is accepted, it shall be handled like a GUI request or an event loaded from the persistent Event Source.

As messages may be delivered or loaded multiple times, we require that the application of events is *idem potent*, i.e. if you apply an event two times, the second application shall have no effect.

As [2] states, events give witness of model changes that have already happened (e.g. in another microservice). Thus, if e.g the SE Group receives a *studentEnrolled* event, in order to achieve synchronization, the SE Group must incorporate this information within its own model. Thus (external) events are *not veto-able*. If e.g. the handling of a *studentEnrolled* event within the SE Group requires the existence of an appropriate *Student* object, the SE Group shall create such a *Student* object on the fly, if necessary. This requires that the *studentEnrolled* event contains sufficient information that allows the creation of a (placeholder for a) *Student* object, in this case e.g. a *studentId*.

In [1] we derived some formal specification of the synchronization of Event Sourcing based microservices via shared Events. With the help of this formalization we can prove that the synchronization of microservices is valid, if the implementation sticks to the requirements stated above. In our reference implementation for our four Students' Affairs microservices we achieved coherence to the Event Sourcing requirements using API methods that implement a *getOrCreate* behavioral pattern. For example the *getOrCreateStudent(studentId, name)* API method uses the *studentId* to look up the corresponding student. If such a student does not yet exist, the corresponding object is created. Then the student's *name* is updated. If you call this API method twice with the same parameter values, the second call will not affect the internal data, thus method *getOrCreateStudent* is idem potent as required. API method *grade(student, examination)* requires a valid student object and a valid examination object as parameters. Thus, the corresponding *studentGraded* event must provide a *studentId* and an *examinationId* in order to be able to call *getOrCreateStudent* and *getOrCreateExamination*. These operations will reliably retrieve the required *Student* and *Examination* objects. Note, if the corresponding *Student* or *Examination* objects do not yet exist, they will be created. This may e.g. result in a *Student* with a *studentId* but without a proper *name*. To provide a proper *name*, we may either include the *name* in the *studentGraded* event or someone has to send an appropriate *studentCreated* event at some time.

Overall we have collected a set of requirements for the synchronization of microservices that are based on Event Sourcing. These requirements allow to guarantee certain consistency conditions between microservices. In addition, we derived clear implementation guidelines from these conditions that make it easy to meet the requirements and to achieve consistency and synchronization schemes for multiple microservices that share some common data.

References

[1] S. Copei, Marco Sälzer, and A. Zündorf. Mx for microservices. In Perdita Stevens Vadim Zaytsev Anthony Cleve, Ekkart Kindler, editor, *Proc. Dagstuhl Seminar 18491 Multidirectional Transformations and Synchronisations*. Schloss Dagstuhl, 2018.

[2] Eric Evans and Rafał Szpoton. *Domain-driven design*. Helion, 2015.