2021

TAHS – Assignment 1

GROUP 13A

AUTHORS :

- ANDY LI
- TANESHWAR PARANKUSAM
- MEHMET SÖZÜDÜZ
- MIHNEA TOADER
- JEGOR ZELENJAK
- JEROEN BASTENHOF

Table of Contents

1 Bounded Contexts

- **Users** – represents the users of the system: *students* (who can become candidate TAs), *candidate TAs* (who can become TAs), *TAs* (who can become students again), *lecturers* (who are responsible for their courses and for hiring TAs), *admin* (a lecturer who has the full access to the system).
- **Authentication** – allows users to be authenticated (identified) and given permissions (access rights) based on their role in the system. The permission management is regulated by passing session tokens to users at authentication. Session tokens that were sent with requests will be checked here for authorization.
- **Courses** – represents the courses that are present within the system. Contains *information* about the course, e.g., name, recruited TAs, number of students.
- **Hiring procedure** – facilitates the *hiring* of students for TA positions by allowing users to *register* for a TA position (taking into account the timing constrains for the application) and allowing lectures to hire new TA's.
- Hour management – allows TAs to declare their hours for a specific course. Also, allows lecturers to approve / reject the declared hours.

Each bounded context is mapped to a single microservice design with identical name. A functional description of the interactions per bounded context can be found in the subsections below.

1.1 Users

The Users microservice:

- Deals with the functionality for each specific user (students, candidate TAs, TA, Lecturers).
- Interacts with the Authentication microservice to give access to authorized users based on their permissions.
- Interacts with other microservices depending on their roles.

1.2 Authentication

The Authentication microservice:

- Deals with authenticating users based on their NetID and password. Also responsible for passing session tokens to the clients who have been successfully identified.
- Stores notifications that are directed to the user (sent while logging in).

1.3 Courses

The Courses microservice:

- Lecturers can create / edit courses.
- Grade and pass information will be stored in the database belonging to course.

1.4 Hour Management

The Hour Management microservice:

- Tracks the TA's hours worked, and hours declared for every specific course.
- Lecturers can approve the hours worked for every TA.

1.5 Hiring Procedure

The Hiring Procedure microservice:

- Provides functionality to both candidate TAs and Lecturers to **submit** and **review TA applications**.
- Lecturers can **choose TAs** from these applications.
- TAs are **notified** of the decision.

1.6 Interactions

- Lecturers can **create courses**, **approve declared hours** by interacting with the Courses microservice. (*Users* → *Courses*)
- Lecturers can **hire TAs**, **see information** about candidates, **ask for recommendations** and **filter applications** by interacting with the Hiring Procedure microservice. (Users → Hiring Procedure)
- Students can **apply to become TAs** and **withdraw their candidacy** (unless the lecturer has started the selection procedure) by interacting with the Hiring Procedure microservice. (Users → Hiring Procedure)
- TAs can **declare hours worked and hours declared** by interacting with the Courses microservice. (Users → Courses)
- TAs can receive contracts to sign when the Hiring Procedure microservice interacts with the Users microservice. (Hiring Procedure → Users)
- Authentication interacts with Users to approve the requested actions. (Authentication → Users).

1.7 Explanations

We chose to build a microservice architecture based around a central microservice (**Users**) that interacts with the other microservices to provide the functionality. Users authenticate themselves and, based on their roles, requests get sent to the other microservices. Managing different user types is aggregated into a single microservice (Users) that acts as a central hub.

Authentication is specific enough to warrant building a separate microservice for the provided functionality. Every request is passed to this microservice to either assign a token or check for authorization.

Since **Courses** are a core concept in the university learning structure, we chose to integrate the functionality in a separate microservice. This microservice receives requests from Users to modify and add course specific information, such as hours worked or approving the hours worked.

The Hiring Procedure microservice provides most of the functionality that is specific to our scenario, i.e., submitting and withdrawing applications, reviewing them, and offering TA recommendations. Since this is a lot of functionality, we chose to put it in a separate microservice.

Our architecture is **vertically scaled enough** to provide modularity and scalability, but **not too large** as to not hamper our understanding of it and to not have interaction overhead. **Maintainability** is also an important factor of our system architecture. Inherently, systems designed around microservices are more easily maintained and this is also the case for us. Since mocking microservices is readily done, good **testability** is also a feature of our system.

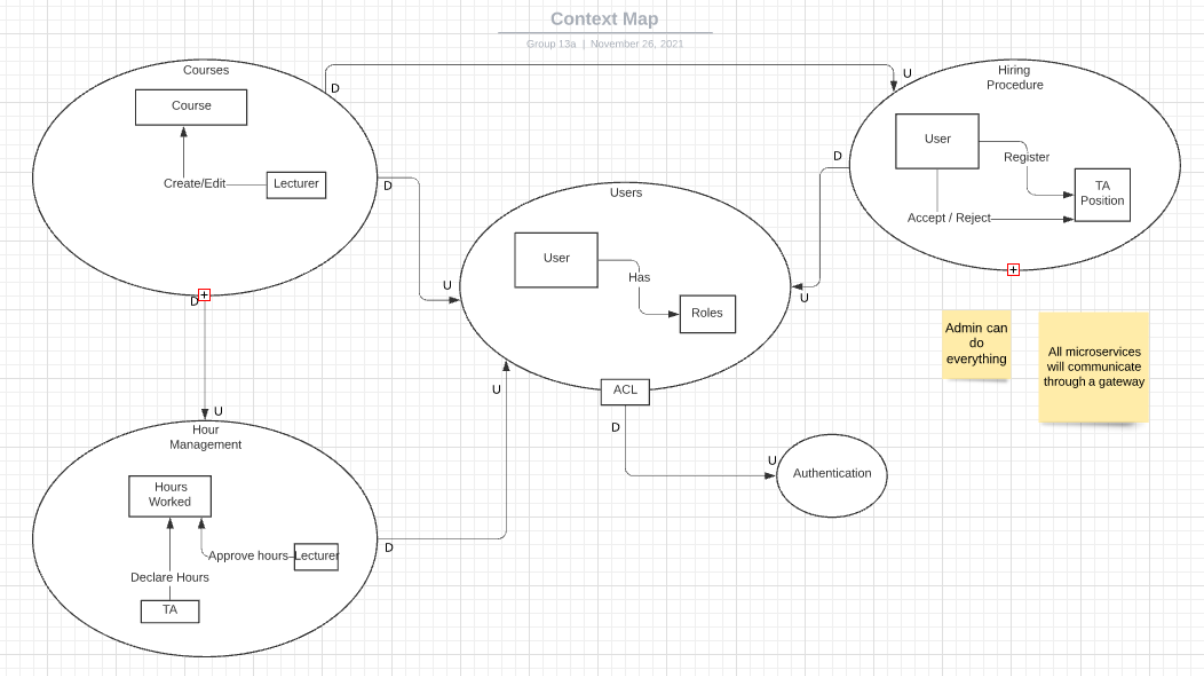

Figure 1: Context map of the derived contexts.

1.8 Component diagram

A high-level component diagram of the previously described components can be found in Figure 5 (Appendix A). The diagram shows a minimized version of the high-level interaction between microservices. The *REST API* and *DataAccess* interfaces act as groups to represent the overall API and data paths between microservices respectively. Important interfaces have been listed separately to give a visual indication of what the microservice can be used for.

2 Design patterns

In this chapter, you will find all the design patterns that are implemented in the project source code. Each pattern is explained in *natural* language and is accompanied by a class diagram that reflects its design.

2.1 AsyncValidator – Chain of Responsibility

2.1.1 Description and rationale

The chain of responsibility is a useful design pattern for delegating certain *responsibilities* to specialized classes, hence its name. Since most of the microservices contain several restrictions for interacting with endpoints, as well as additional constraints, the chain of responsibility pattern fits in quite nicely.

The following three reasons are the main reasons why the chain of responsibility pattern is crucial for being able to properly maintain, and extend, the current codebase.

1. All components within the chain perform their own action and are not influenced by other chain components. Therefore, they can live in separate classes;
2. Components can easily be re-used by other endpoints operating within the same microservice;
3. Since all components are implemented in separate classes, they are easily maintainable.

As the hour management and hiring procedure microservices heavily rely on other microservices and local restrictions on the input data, the chain of responsibility pattern is a perfect fit.

Both implementations make use of a top-level interface (*AsyncValidator*) that defines the general structure of all validators, which operate in an **asynchronous** context. The *AsyncBaseValidator* class is used as a base class for all validator components. This base class prevents from having a lot of code duplication in the chain components caused by the chain forwarding part of the pattern.

All validator components, which derive from the *AsyncBaseValidator*, override the *validate* method that is called upon validation and populate it with their own checks. Using the base class's *evaluateNext* function, it is possible for a chain component to advance to the next node, if possible.

Exceptions/errors are handled by returning a *Mono.error(…)* instance which will be thrown in the future and handled by Spring's REST advisors. When a component specifies that there was an exception/error, the chain will not evaluate the components that are further up the chain.

2.1.2 Class diagram

The class diagram of the chain of responsibility implementations can be found in the figures below. Figure 2 shows the class diagram of the asynchronous validation chain of the hour management microservice, and Figure 3 shows that of the hiring procedure microservice.

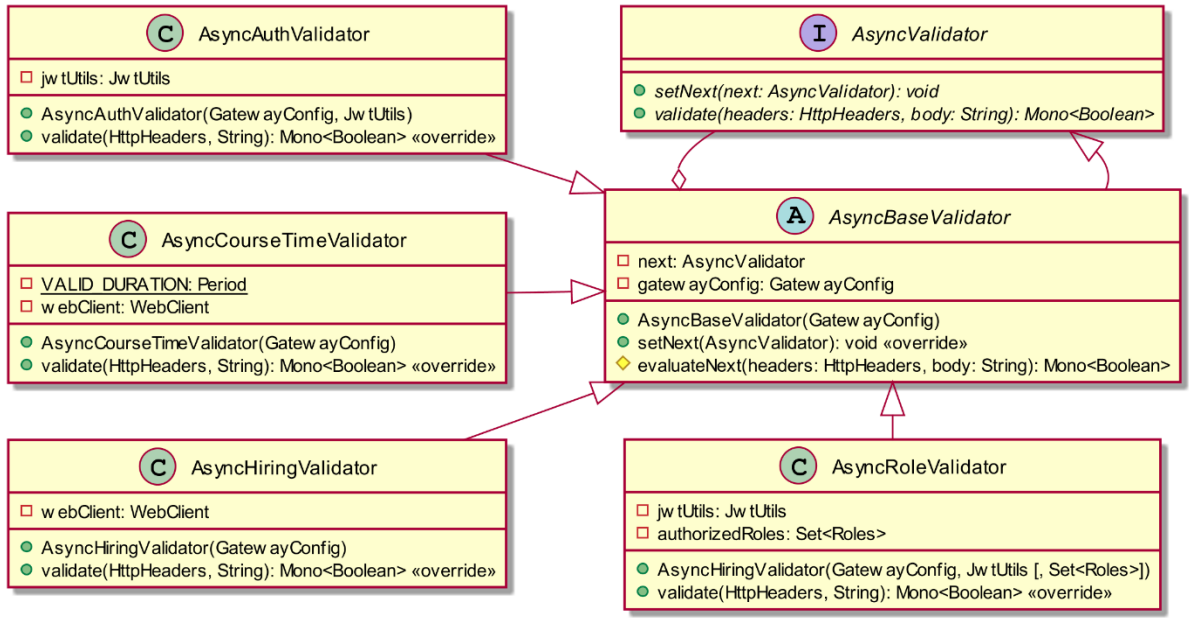

Figure 2: (Detailed) class diagram of the async validation components used by the hour-management microservice.

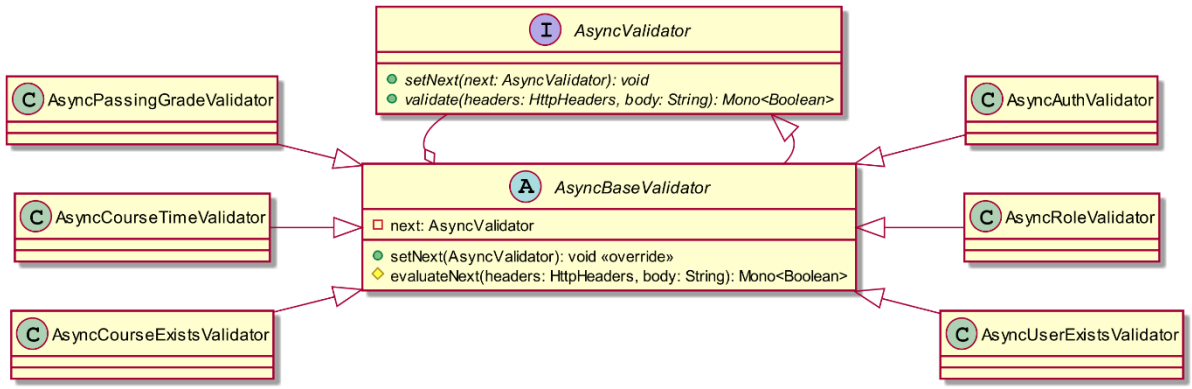

Figure 3: Class diagram of the async validation components used by the hiring-procedure microservice.

2.1.3 Occurrence(s) in source code

The asynchronous validator components and interface can be found in the following locations.

- */microservices/hiring-procedure/src/<package>/validation/*
- */microservices/hour-management/src/<package>/validation/*

Notice from the images in 2.1.2 that the interface class (*AsyncValidator*) is identical for both microservices. Therefore, one might wonder why this interface class is not positioned in a library or something similar. The reason for this is that the microservices themselves might, at a later stage, decide to change the constraints of the data that is being validated, or the default information that is stored by, e.g., the base class. Propagating these changes to all microservices would be cumbersome, and might not even be possible due to, e.g., microservice-specific object types. Hence the reason why the currently used *AsyncValidator* interface can be found in both microservices.

2.2 AsyncValidator – Builder

2.2.1 Description and rationale

The use of the builder pattern is, to some degree, tied to the chain of responsibility that we touched upon in the previous section. The main purpose of the builder is to provide an easy-to-use interface for chaining all validators used by the chain of responsibility without having to store intermediate variables. Variables that are solely used to construct the chain. A comparison can be found in Appendix B.

Now, one might wonder why this chain is not made upfront. This is mainly due to the following reasons.

1. The composition of the chain differs per endpoint, i.e., one endpoint might have to verify the duration of a course, whereas another endpoint might not;
2. The construction of some chain components require information that is request-specific and is not passed alongside the 'chain'.

Because of this, a builder was brought to life that allows construction of a chain of responsibility without having to perform the linking manually. To facilitate this process, the builder offers two distinct methods/functions.

- *addValidator* – allows the caller to add a single validator to the chain;
- *addValidators* – allows the caller to add multiple validators to the chain at the same time (order is preserved).

The builder stores a pointer to the head and tail of the chain internally. The head is used to represent the starting point of the validator chain, whereas the tail node is used to append new validators upon calling one of the methods offered by the builder.

2.2.2 Class diagram

The class diagram of the builder implementation can be found in Figure 4. Notice that the implementation resides *within* the interface itself. This makes it more explicit that the builder belongs to the *AsyncValidator* only.

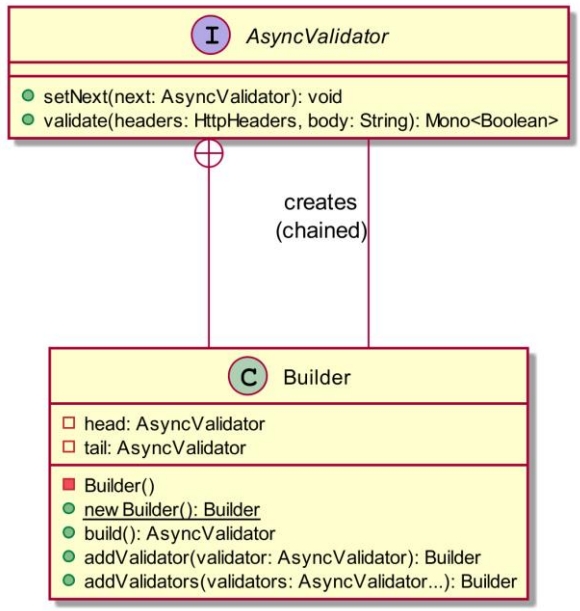

Figure 4: Class diagram of the used builder that follows the builder design pattern.

2.2.3 Occurrence(s) in source code

The implementations of the described builder pattern can be found in the following locations.

- */microservices/hiring-procedure/src/<package>/validation/AsyncValidator.java*
- */microservices/hour-management/src/<package>/validation/AsyncValidator.java*

Note that both occurrences are similar. The reasoning behind this is explained in 2.1.3.

3 Appendix

Appendix A – Component Diagram

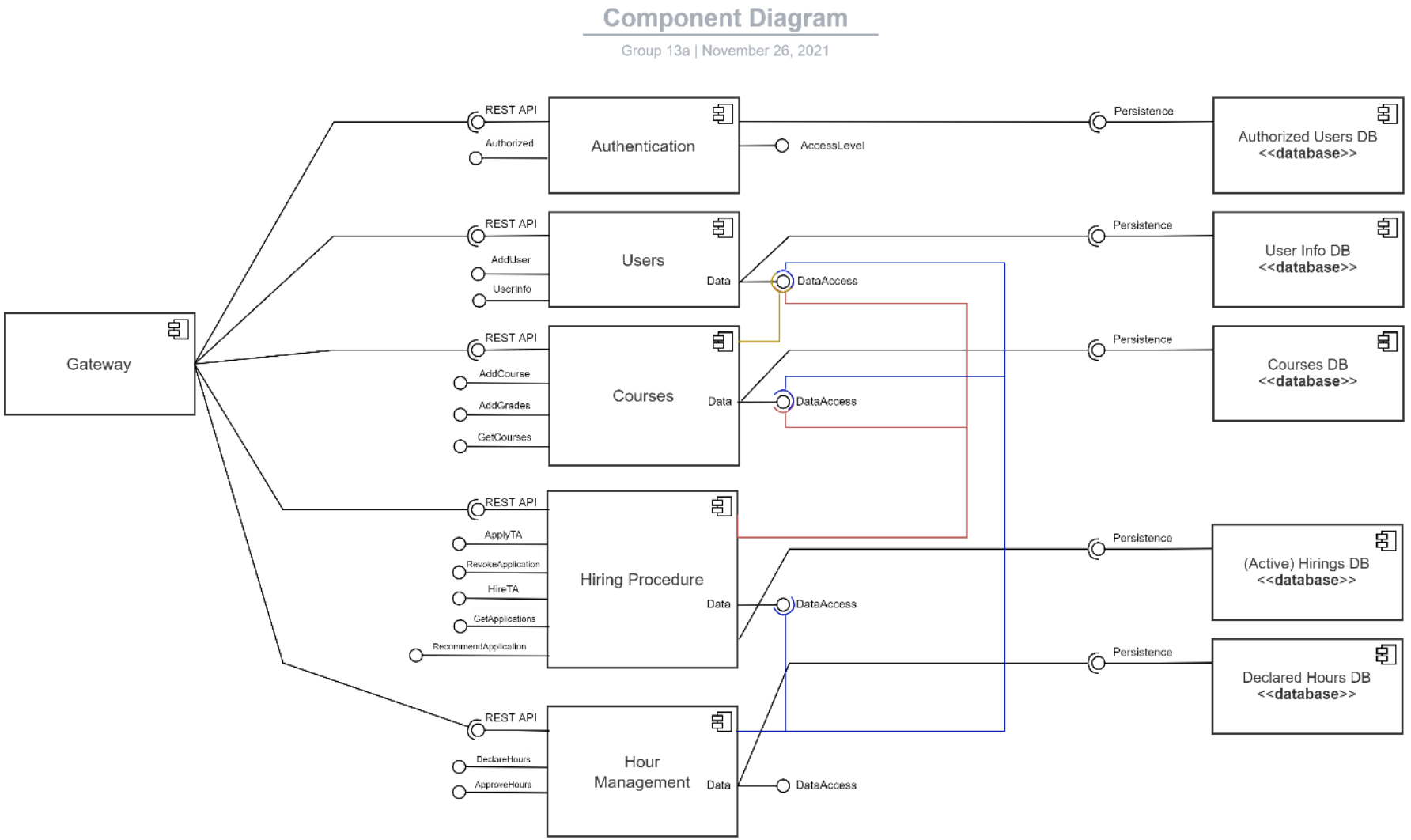

Figure 5: High-level component diagram of the derived microservices.

Appendix B – Asynchronous validator builder example

The code snippet below is an example of the construction of an asynchronous validator chain using the *AsyncValidator.Builder* class. Notice that the builder approach is more expressive and less error-prone compared to the manual approach. Since the order is preserved by the builder, the structure of the validator chain is directly visible.

```
// Asynchronous validator building using the AsyncValidator.Builder class
AsyncValidator head = AsyncValidator.Builder.newBuilder()
    .addValidators(
        new AsyncAuthValidator(...),
        new AsyncRoleValidator(...),
        new AsyncCourseTimeValidator(...),
        new AsyncHiringValidator(...),
        ...
    ).build();

// Manual asynchronous validator building
AsyncValidator head = new AsyncAuthValidator(...);
AsyncValidator roleValidator = new AsyncRoleValidator(...);
AsyncValidator courseTimeValidator = new AsyncCourseTimeValidator(...);
AsyncValidator hiringValidator = new AsyncHiringValidator(...);
...
head.setNext(roleValidator);
roleValidator.setNext(courseTimeValidator);
courseTimeValidator.setNext(hiringValidator);
...

// Accidents can happen during manual construction
head.setNext(roleValidator);
head.setNext(courseTimeValidator); // Error: roleValidator is skipped
...
```