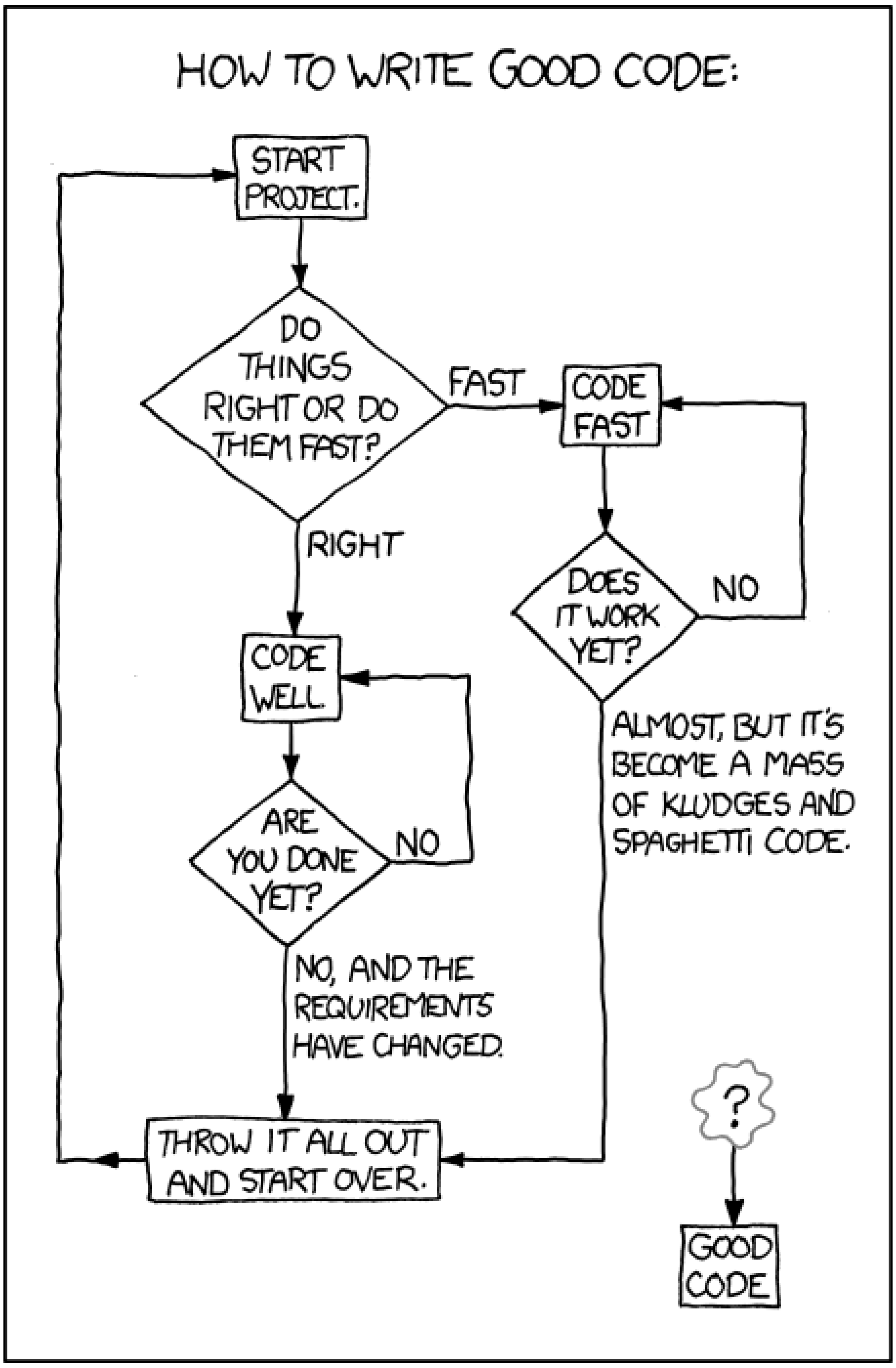
Assignment 1

Group 13B
17-12-2021


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL.
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
?
THROW IT ALL OUT AND START OVER.
GOOD CODE

Authors

A.F. Yücel J.D.M. Savelkoul M. Mladenov

M.C.A. de Wit M.A.A. Kienhuis W. Smit

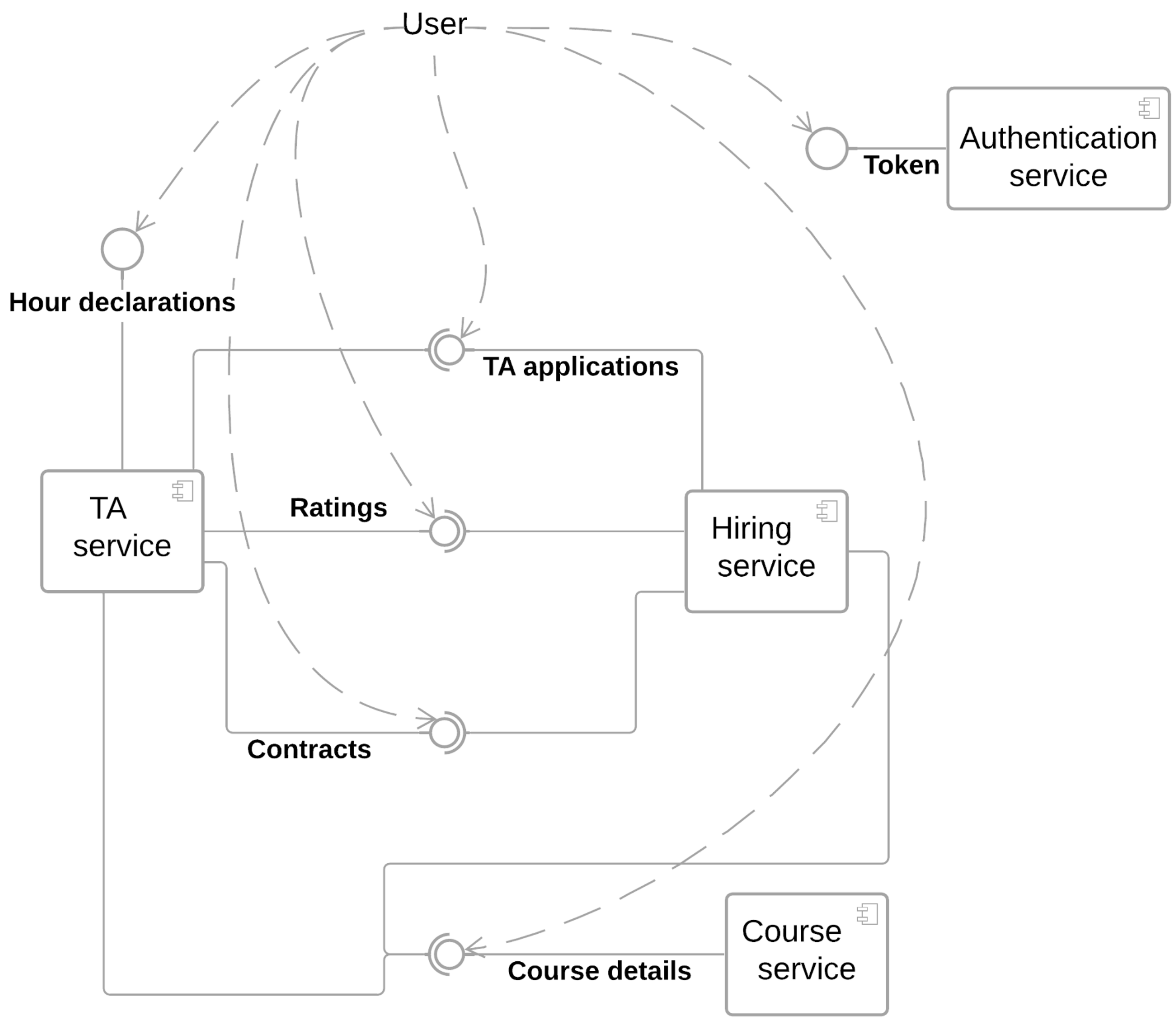
Supervisor

George Hellouin de Ménibus

Part 1

Bounded contexts / DDD

In our application there are four bounded contexts to be identified. These bounded contexts consist of authentication, hiring, TA (contracts + submitting worked hours), and course information. The authentication service is required to make sure that no user can impersonate another user without either their verified token or their username-password combination. With authentication out of the way our application will mainly consist of 2 workflows: either TAs are being hired, or TAs are being paid. These 2 workflows can be reasonably separated since paying out one TA is barely related to the hiring process of another. However there is still one main way the two workflows are connected and it is the metadata of the course. Both processes need to know who is responsible for approving contracts and worked hours. For this purpose the final service is created, the course information service. The course information service will contain all the metadata that a course will have and supply this to our other services.



Authentication service

This service will handle the authentication of users of the application. It will have a database containing all usernames and (securely hashed) passwords of users. This service can be asked to create a JWT token which the user can then use to prove their identity to the other microservices.

The token given, if the user is authenticated, is a signed string containing the user's netid and an expiration date. This token can then be verified by the other microservices to get the netid and check if a user with that netid has permission to do a certain task or operation. This way there is no need for any communication to the authentication service while a request is propagating through the microservice architecture. As a result the load on the authentication microservice will be significantly reduced which in turn improves performance, while not compromising on security in any way.

Note that it is not the task of the Authentication microservice to determine someone their role in the system. Its only purpose is to issue a token which can then be provided to other microservices as proof that the user is who they claim to be. One should see this as a form of identification you can carry around.This identification is then sent along with any request to the application. Each microservice will then individually determine the user's role based on information it already has or can collect from other services. For example, the role of a responsible lecturer is not in any way stored in the JWT token, but is validated by each microservice separately by checking the requester their netid against the list of responsible lecturers for a particular course.

Whilst there is a possibility to have the authentication service also save user information, such as what courses they lecture, it was decided not to do this. The main reasoning was that the system should be scalable and flexible. Having the authentication system also incorporate a lot of user information does not make the application scalable. Since all instances of the authentication service would need the exact same information, otherwise a user could end up with a token that might already be out of sync with their permissions. And the token would need to constantly be refreshed in the case that somebody else changes the users permissions during the session. It also hurts flexibility since all data is assumed to be present in the token, and a small change in its structure now needs to be incorporated in every service. Furthermore, it would also affect security, since a user who has had one of their roles taken away (for instance, due to account compromise) would still in practice retain those roles until all of their tokens expire. Hence it has been decided that only storing the netid in the JWT token is the optimal route, not only in terms of security but also scalability and performance.

Course service

This service is responsible for keeping track of courses and their responsible lecturers. This information should be available to both the hiring service and TA service. We are aware that this microservice is getting close to just a database talking to other databases. But since two of the main bounded contexts, hiring and submitting worked hours, are dependent on this data it was deemed necessary to split off this functionality into its own simple and highly-scalable service.

Courses are needed in the hiring service for applications and acceptance of applications. The hiring service needs it to check if the course has started yet and whether or not a certain user is a responsible lecturer and therefore has permission to accept an application. Furthermore, courses are needed in the TA service to check if a user can approve declared hours of a TA for the same reason. What sets apart the course microservice from an authorization microservice responsible for distributing roles is the fact that the information related to authorization is merely a small part of the information the course microservice provides about courses - name, start date, description, list of responsible lecturers, etc. This makes the course microservice an integral part of the internal network and all requests made to this service still require authentication via a JWT token for security reasons.

We expect this service to be the main bottleneck of the application since the other internal services can communicate with this service quite often. However, the course service is mostly a read only database. Changes to its data can be made but are neither critical nor require instant response. This makes it so that this service is exceptionally scalable in the event of a server overload occurring. If scaling is necessary, the different instances of this microservice can very effectively utilize caching in order to preserve its high performance even when the number of requests to this microservice is exceptionally high.

An alternative option was to incorporate this service into the other services. However, this introduces a problem. Either one of the services runs the database, or both of them do. If one runs the database this could result in choking behaviour, where a system hogs up all the resources to first resolve its own needs, taking out one of the services if another one is busy. If both of them run the database then there will be a lot of overhead when keeping them in sync and it is not clear which of the two services should receive the original update request. This made it trivial to split this area of concern into another microservice. Although splitting this functionality into its own microservice increases the general complexity of the entire network, this makes the entire system more scalable and easier to maintain in the long term.

As the application utilises a very flexible custom context-specific role framework, anyone is allowed to create a course. When a course is created, the user who creates it is assigned as a responsible lecturer, and they have lecturer permissions only on that particular course. If necessary, this behaviour can easily be modified using the details provided from the single sign on system, which could be integrated with the application.

Hiring service

This service is in charge of the hiring of new TAs. It will store and process all TA applications submitted by students and will be responsible for “transferring” an accepted TA to the TA service. Clustering this functionality into one service makes sense, the process of hiring someone has very little to do with an hour declaration service. Splitting this functionality into its own microservice allows it to easily be scaled up when the application period for many courses is open, and scaling it down when very few students want to apply.

The hiring service will not be running in isolation as it is the start of every TA's journey and therefore close to the heart of the application. First and foremost, the service which the hiring service needs to communicate with the most is the course microservice. It requires this communication to check the start date of the course or to verify the start date of the course a student is applying to be a TA for. Checking whether the user is a responsible lecturer through their netid is necessary for functionality such as approving or rejecting applications, retrieving a list of all applicants, and accessing the auto recommendation system.

Then at the final stage there will also be some communication with the TA service. This communication exists for two main reasons: to retrieve past TA performance and to transfer an accepted application. The past TA performance is needed to aid lecturers when making a selection between candidates. When an application is eventually approved, a contract is created on the TA service for the newly accepted TA. From now on, this new TA will only mainly communicate with the TA service.

Alternatively, the hiring service could have merged the TA service. However, we would argue that splitting the application into two sides: “applying for a TA” and “working as a TA”, is very logical and significantly improves scalability by allowing scaling up the two microservices independently from each other when necessary. This is possible thanks to the fact that there is a clear split in functionality between the two. As a result, it is possible to split the features into two microservices, with their own distinct features, which can retain their high performance even when the system is under heavy load thanks to their highly-scalable nature.

TA service

This service will handle everything that has to do with TAs. The TA contact, hour declarations, and ratings are all stored and processed here. We chose to separate this from the hiring service since there is a natural split in functionality between the features a student who is applying as a TA and someone who is an active TA would use.

When a student has applied and is accepted by a lecturer their information is “transferred” to the TA service and a contract is generated. This contract will be the main entity that makes a student a TA. This contract can also later store the rating of the TA given by lecturers.

With the process of declaring hours also being handled through this microservice we have introduced quite a lot of functionality, but not without reason. In some situations it was deemed beneficial to partly sacrifice potential scalability for a significantly lower overhead. With this service containing the contracts of every TA we do not need to communicate with any other service when declaring worked hours. The alternative was to create a contract service, but in that case declaring worked hours would always require a call to the contract service to determine whether or not a student is filling in more hours than they are allowed. This would result in two microservices that are heavily coupled together, which would not improve scalability in a meaningful way, but would only introduce unnecessary overhead. Seeing as how there is no other service that requires this information, we can easily merge this functionality into one service in order to achieve better performance in high-demand situations.

While most of the requests to this service can be run in isolation, there is still a need for effective communication with other services. As mentioned before, there will still be necessary communication with the course service, which stores the lecturer data, especially when it comes to approving the hour declarations for a specific course. Not only that, there will also be requests originating from inside the network that will call this service, namely the hiring service. The hiring service is the bread and butter of the application in combination with the TA service. When the hiring service has approved a new TA it sends over the contract to the TA service effectively transforming the student into a TA. Finally, the TA service also exposes an endpoint for internal use, the ratings of a TA, this allows a recommendation system to quickly look up the score of every TA for automatic recommendation.

The big speaking point was moving the rating attribute into this service and not the hiring service, as both options are valid for their own reasons. The current solution of moving the rating into the TA service introduces extra overhead since it is an attribute that is only used by the hiring service. However, we like to look at future flexibility. Storing the rating in the TA service makes more sense logically - it is related to the work of an active TA, not to their application. Furthermore, In the future other data could also be used, such as hours worked, to create a more accurate score. This data cannot be exposed directly as it would contain financial or otherwise sensitive information. Therefore it was decided that with the rating being implemented in the TA service there would be more future potential.

Gateway

The gateway service only routes requests to the other services. It does not have a database, and does not process any information. This decouples it from the other services entirely, improving the scalability and maintainability of the entire application.

Summary

Course service

Stores all information of a course (dates, lecturer, etc.).

Users can create courses, and other services request course information (e.g. the netid of the responsible lecturer in order to check permissions, etc.).

TA service

Handles active and past contracts, hour declarations, and ratings.

TAs can submit hour declarations, and lecturers can approve them. Requests the netid of the lecturer for authorization before allowing users to approve hours or create contracts. Automatically approves a TA application when a contract is created.

Authentication service

Handles authentication.

Allows users to register. Allows registered users to login, and provides them with a JWT token that other services can then verify independently.

Hiring service

Handles applications.

Users can apply to be TAs, and lecturers can manage applications.

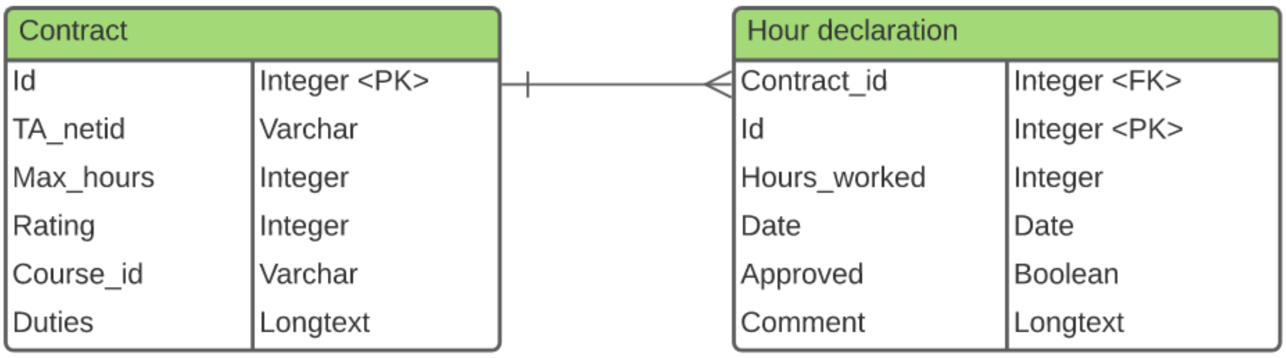
Requests course details to check if the user can apply at that time. Requests the netid of the lecturer for authorization before allowing users to modify applications (e.g. approve, reject, etc.).

Requests ratings and past contracts when showing applications to lecturers.

When an application is approved, it makes a request to the TA service to create a contract.

Database structure

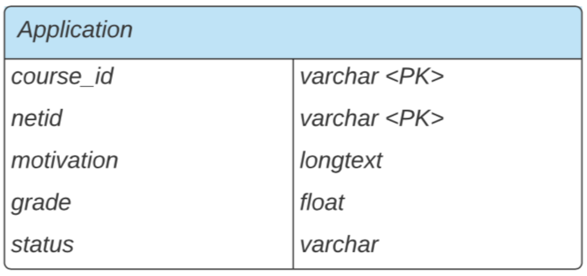
TA service



Contract	
Id	Integer <PK>
TA_netid	Varchar
Max_hours	Integer
Rating	Integer
Course_id	Varchar
Duties	Longtext

Hour declaration	
Contract_id	Integer <FK>
Id	Integer <PK>
Hours_worked	Integer
Date	Date
Approved	Boolean
Comment	Longtext

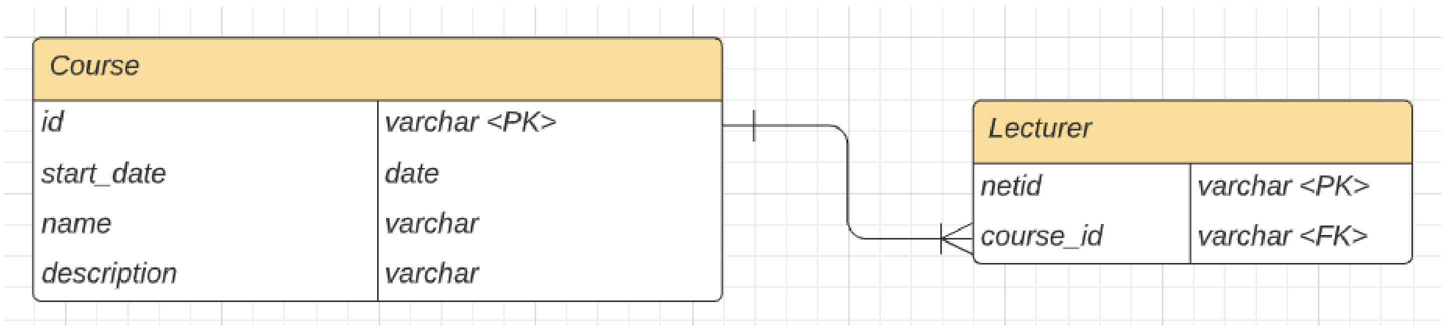
Hiring service



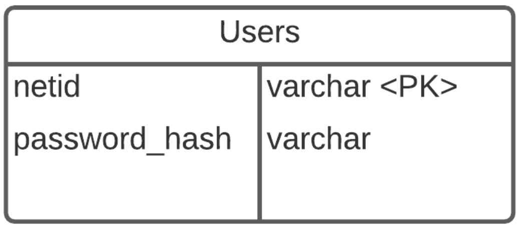
Application	
course_id	*varchar <PK>*
netid	*varchar <PK>*
motivation	*longtext*
grade	*float*
status	*varchar*

Remarks
The hiring microservice will handle everything that has to do with the hiring proccess. It will only need to store applications of students with their motivation and grade. This all can be stored in one simple table.

Course service



Authentication service



Users	
netid	varchar <PK>
password_hash	varchar

Remarks:
The authentication service is not responsible for roles. Roles in our application are context-dependent and the authentication service only deals with authentication and not authorization.

Part 2

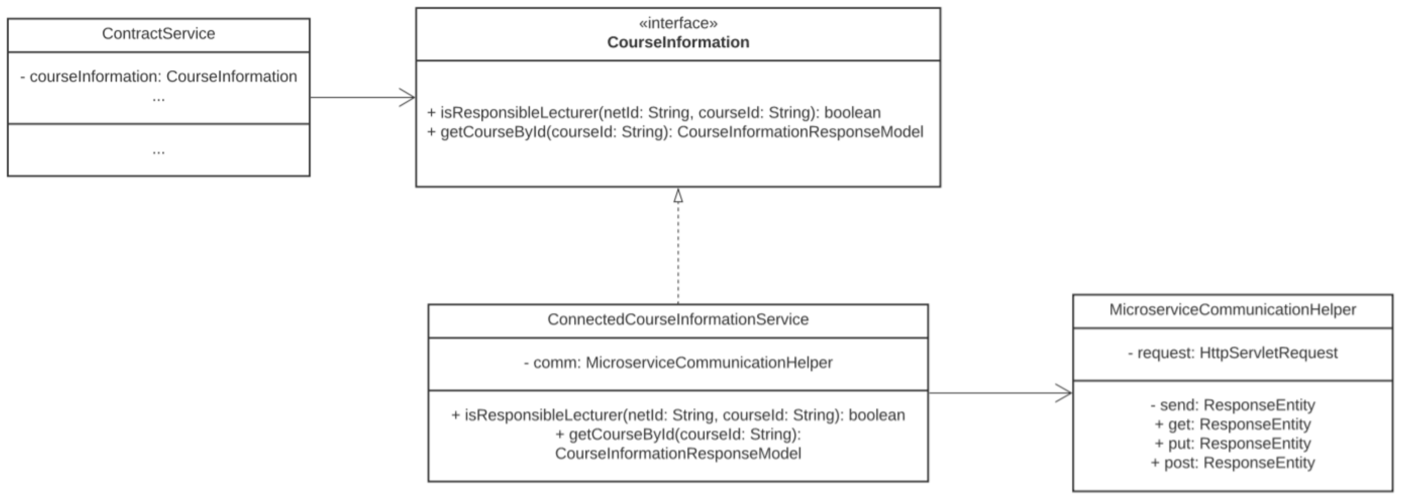
Adapter pattern

The adapter pattern is probably the pattern that can emerge the most. It focuses on being able to switch between different implementations depending on what is needed and is very closely related to Java's concept of an interface. The pattern is most useful when dealing with code that cannot or will not be changed to meet a new output format but still needs to communicate with newer machines. In those cases one might want to create an adapter between the old and the new machine and this is where the adapter pattern was born. On the new machine we will implement this adapter and the adapter itself will implement some common interface that all adapters share. This way the adapter can be swapped out depending on the format that is being presented.

This pattern greatly improves code maintainability as it makes developers avoid writing a single god class for parsing but will instead allow them simply to create a new class that can be swapped out. Even when only one format is presented the pattern should still be applied. Especially because you cannot know in advance if there will be any new format in the future that you will need to adapt your code to. This pattern also helps alleviate complex modules by hiding implementation specifics behind an interface which can then also easily be mocked. This overall makes the adapter pattern one of the easiest to work with patterns, and one that in general can even be created on accident.

At the moment, there is one implementation of each of the interfaces. However, using the adapter pattern here makes it easy to swap the adapter with another implementation later if that is necessary. For example, we could easily make the means of communication between microservices work differently without having to change much code - we would only have to create a new adapter. Furthermore, this opens the door to improving the application without the need to modify existing code. For instance, caching could be implemented without having to change the way internal services interact with the CourseInformation interface, and there could simultaneously exist cached and non-cached implementations of CourseInformation which can easily be swapped as necessary. The use of the adapter pattern in this way improves the code maintainability of the application.

UML diagram of our implemented adapter pattern:


ContractService
- courseInformation: CourseInformation
...
...
«interface»
CourseInformation
+ isResponsibleLecturer(netId: String, courseId: String): boolean
+ getCourseById(courseId: String): CourseInformationResponseModel
ConnectedCourseInformationService
- comm: MicroserviceCommunicationHelper
+ isResponsibleLecturer(netId: String, courseId: String): boolean
+ getCourseById(courseId: String): CourseInformationResponseModel
MicroserviceCommunicationHelper
- request: HttpServletRequest
- send: ResponseEntity
+ get: ResponseEntity
+ put: ResponseEntity
+ post: ResponseEntity

An example implementation of the adapter pattern:

- The contract service: https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/55afc6264f963a2c9da18ac836706934f9307de9/ta-service/src/main/java/nl/tudelft/sem/template/ta/services/ContractService.java
- The adapter: https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/55afc6264f963a2c9da18ac836706934f9307de9/ta-service/src/main/java/nl/tudelft/sem/template/ta/services/communication/ConnectedCourseInformationService.java
- The interface: https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/55afc6264f963a2c9da18ac836706934f9307de9/ta-service/src/main/java/nl/tudelft/sem/template/ta/interfaces/CourseInformation.java
- The adaptée: https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/55afc6264f963a2c9da18ac836706934f9307de9/ta-service/src/main/java/nl/tudelft/sem/template/ta/services/communication/MicroserviceCommunicationHelper.java

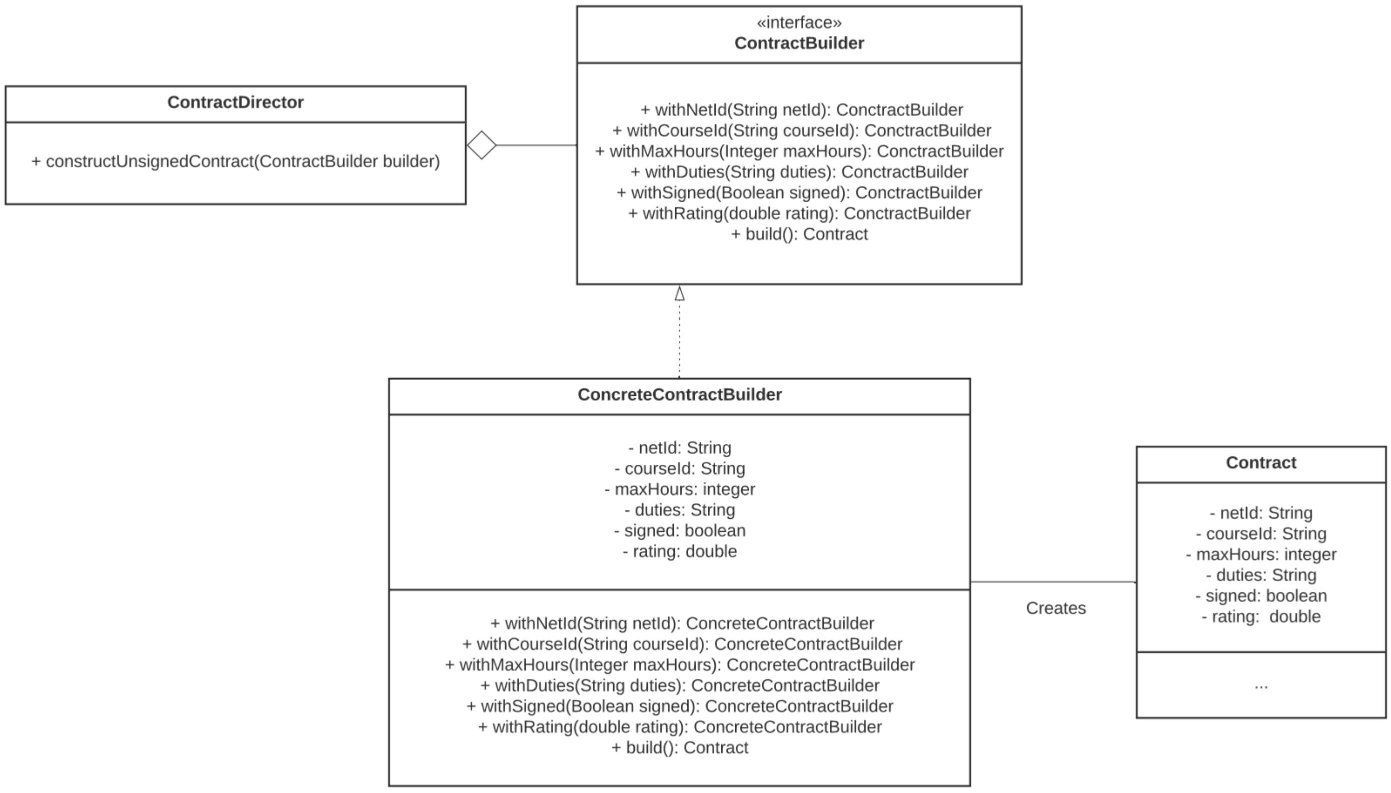
We have implemented the adapter pattern in multiple places throughout the application, but this is omitted here for briefness due to the fact that they are implemented in a very similar way as the one displayed above.

Builder pattern

The builder pattern enables us to build complex objects in a simple and less error-prone way. It serves as a verbose interface, making the creation of objects with many attributes very straightforward.

We have utilised the builder pattern in many places throughout the entire application. We chose to use this pattern for instantiating most of our models due to its simplicity - instead of invoking a lot of setters every time we create an instance of an object, or using constructor with many arguments, where making a mistake in the order of arguments is easy, we chose to create builders in order to keep the code maintainable and easily readable even by people not familiar with the codebase.

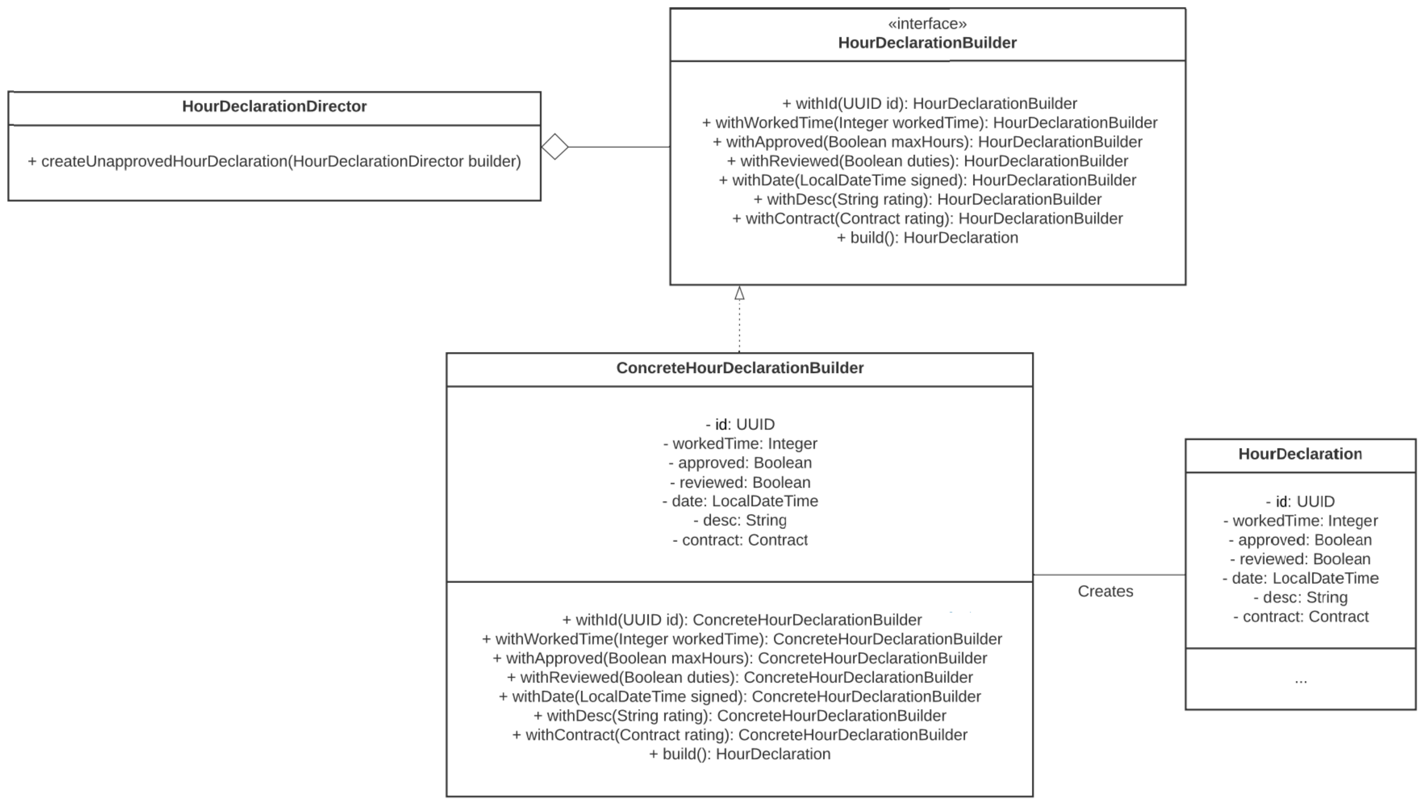
UML diagram of our implemented builder pattern, applied to the Contract entity.


«interface»
ContractBuilder
+ withNetId(String netId): ConctractBuilder
+ withCourseId(String courseId): ConctractBuilder
+ withMaxHours(Integer maxHours): ConctractBuilder
+ withDuties(String duties): ConctractBuilder
+ withSigned(Boolean signed): ConctractBuilder
+ withRating(double rating): ConctractBuilder
+ build(): Contract
ContractDirector
+ constructUnsignedContract(ContractBuilder builder)
ConcreteContractBuilder
- netId: String
- courseId: String
- maxHours: integer
- duties: String
- signed: boolean
- rating: double
+ withNetId(String netId): ConcreteContractBuilder
+ withCourseId(String courseId): ConcreteContractBuilder
+ withMaxHours(Integer maxHours): ConcreteContractBuilder
+ withDuties(String duties): ConcreteContractBuilder
+ withSigned(Boolean signed): ConcreteContractBuilder
+ withRating(double rating): ConcreteContractBuilder
+ build(): Contract
Creates
Contract
- netId: String
- courseId: String
- maxHours: integer
- duties: String
- signed: boolean
- rating: double
...

An example implementation of the builder pattern for the Contract entity.

- Contract.java
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/entities/Contract.java
- ContractBuilder.java
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/entities/builders/ConcreteContractBuilder.java
- ConcreteContractBuilder.java
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/entities/builders/interfaces/ContractBuilder.java
- ContractDirector.java
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/entities/builders/directors/ContractDirector.java
- A use case of the director on line 69
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/services/ContractService.java

UML diagram of our implemented builder pattern, applied to the HourDeclaration entity.


«interface»
HourDeclarationBuilder
+ withId(UUID id): HourDeclarationBuilder
+ withWorkedTime(Integer workedTime): HourDeclarationBuilder
+ withApproved(Boolean maxHours): HourDeclarationBuilder
+ withReviewed(Boolean duties): HourDeclarationBuilder
+ withDate(LocalDateTime signed): HourDeclarationBuilder
+ withDesc(String rating): HourDeclarationBuilder
+ withContract(Contract rating): HourDeclarationBuilder
+ build(): HourDeclaration
HourDeclarationDirector
+ createUnapprovedHourDeclaration(HourDeclarationDirector builder)
ConcreteHourDeclarationBuilder
- id: UUID
- workedTime: Integer
- approved: Boolean
- reviewed: Boolean
- date: LocalDateTime
- desc: String
- contract: Contract
+ withId(UUID id): ConcreteHourDeclarationBuilder
+ withWorkedTime(Integer workedTime): ConcreteHourDeclarationBuilder
+ withApproved(Boolean maxHours): ConcreteHourDeclarationBuilder
+ withReviewed(Boolean duties): ConcreteHourDeclarationBuilder
+ withDate(LocalDateTime signed): ConcreteHourDeclarationBuilder
+ withDesc(String rating): ConcreteHourDeclarationBuilder
+ withContract(Contract rating): ConcreteHourDeclarationBuilder
+ build(): HourDeclaration
Creates
HourDeclaration
- id: UUID
- workedTime: Integer
- approved: Boolean
- reviewed: Boolean
- date: LocalDateTime
- desc: String
- contract: Contract
...

An example implementation of the builder pattern for the HourDeclaration entity.

- HourDeclaration.java
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/entities/HourDeclaration.java
- HourDeclarationBuilder.java
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/entities/builders/ConcreteHourDeclarationBuilder.java
- ConcreteHourDeclarationBuilder.java
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/entities/builders/interfaces/HourDeclarationBuilder.java
- HourDeclarationDirector.java
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/entities/builders/directors/HourDeclarationDirector.java
- A use case of the director on line 47
  https://gitlab.ewi.tudelft.nl/cse2115/2021-2022/sem-group-13b/sem-repo-13b/-/blob/7ef6f63e26deb369d2a890a6c8b27f563d6a03f0/ta-service/src/main/java/nl/tudelft/sem/template/ta/services/HourService.java