The next microservice we decided to create corresponds to the business logic concerning the activities, i.e., trainings and competitions. Hence, the main goal of this service is to allow the user to create activities which will later on be matched with the ‘User’ entities. The core ‘Activity’ microservice will be the sole provider of the functionality for interacting with the activity database. The ‘Activity’ entities will have such attributes as the owner (the ‘User’ who published it), it’s timeslot, the boat type, the certificate and the positions that need to be filled. It will be implemented as an abstract class from which the ‘Training’ and the ‘Competition’ subclasses will inherit.

Originally, the service is only connected to a single other microservice, namely, the ‘Matching’ service, however, it will implement the communication with the client and the ‘Activity Repository’ as well. [Here I can continue talking about the relation between ‘Matching’ and ‘Activity’ if it isn’t described in detail already]. The main interfaces it contains are for the client manipulating the activities, ‘Matching’ service querying the activities for matches and the ‘Activity Repository’ for persisting an activity. As for the interfaces it requires, there is the extraction of actions from the repository, the ability to update the activities in matchings, and finally an interface from the client to update the information.

The key reason that we chose to implement it this way was to keep the microservices lightweight. Instead of handling the activity entities’ manipulation on the same server as the matchings, we opted for decoupling them so the can separate the concerns of manipulating the activities and using them to pair with the users. Regarding the option of splitting it into smaller parts, we decided that it is as atomic as it could be and doing so would introduce more communication overhead.