## Contents

## 1 Foreword

I would like to thank Hannes Leutloff, who has spent countless hours on this project voluntarily and without whose tireless work on the user interface, the build infrastructure, mentoring and code reviews, this project would not have been possible.

## 2 Introduction

### 2.1 Terminology

**data subject**

**data client**

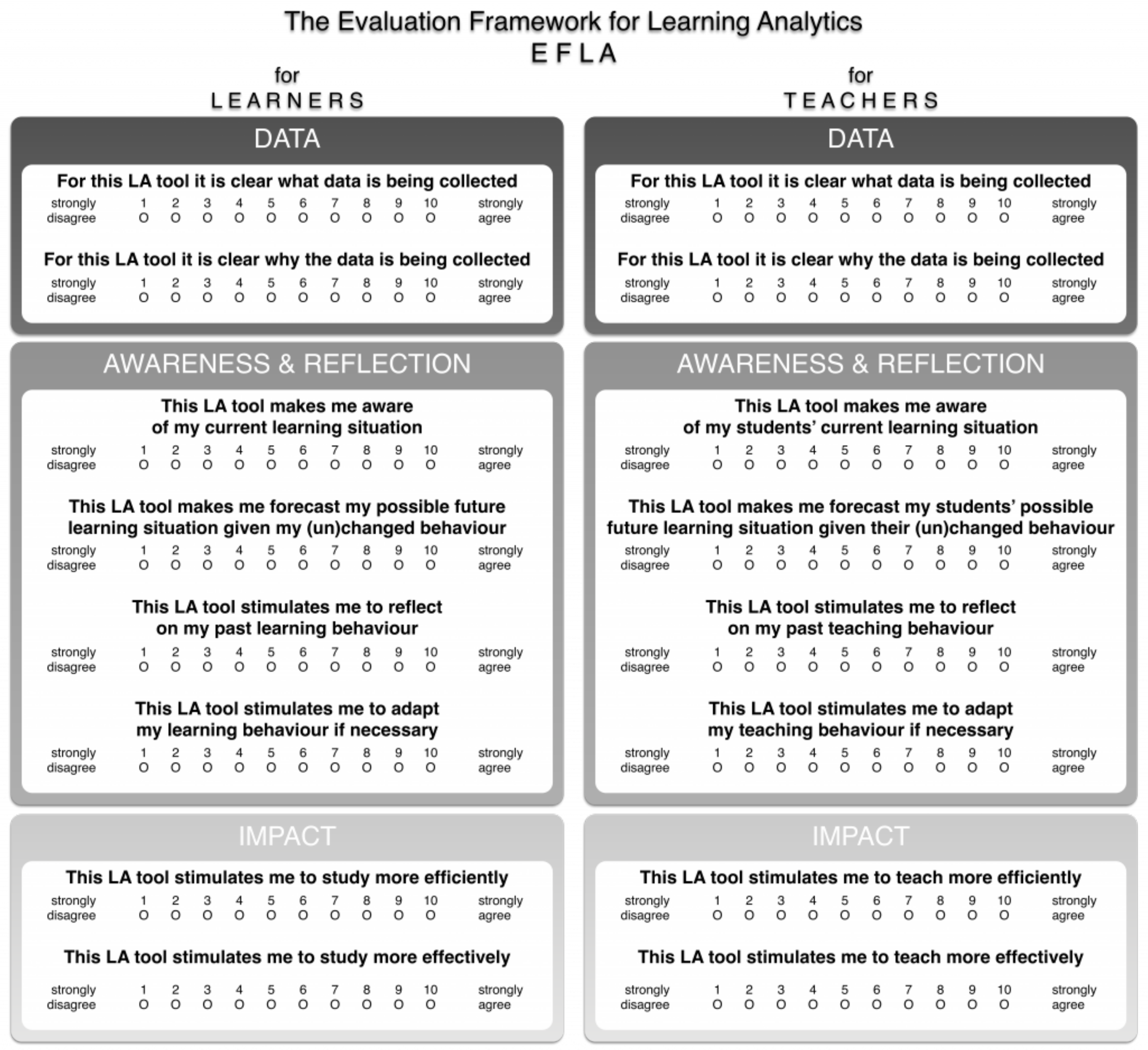
**survey item**

### 2.2 Previous Work

As part of the computational humanities seminar, which was held by Prof. H. Drachsler in winter 2017/18, Hannes Leutloff and I were tasked to digitize the evaluation framework for learning analytics (EFLA) (3) and to develop an online platform where the survey could be taken.

| Feature | Description |
|---|---|
| | |

Figure 2: Features of version 1.0.0 of the survey tool



The Evaluation Framework for Learning Analytics
EFLA

for LEARNERS

DATA

For this LA tool it is clear what data is being collected
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

For this LA tool it is clear why the data is being collected
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

AWARENESS & REFLECTION

This LA tool makes me aware of my current learning situation
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

This LA tool makes me forecast my possible future learning situation given my (un)changed behaviour
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

This LA tool stimulates me to reflect on my past learning behaviour
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

This LA tool stimulates me to adapt my learning behaviour if necessary
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

IMPACT

This LA tool stimulates me to study more efficiently
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

This LA tool stimulates me to study more effectively
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

for TEACHERS

DATA

For this LA tool it is clear what data is being collected
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

For this LA tool it is clear why the data is being collected
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

AWARENESS & REFLECTION

This LA tool makes me aware of my students' current learning situation
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

This LA tool makes me forecast my students' possible future learning situation given their (un)changed behaviour
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

This LA tool stimulates me to reflect on my past teaching behaviour
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

This LA tool stimulates me to adapt my teaching behaviour if necessary
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

IMPACT

This LA tool stimulates me to teach more efficiently
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

This LA tool stimulates me to teach more effectively
strongly disagree 1 2 3 4 5 6 7 8 9 10 strongly agree

Figure 1: Template for the EFLA survey (4)

The result was an online survey platform, where surveys similar to the EFLA can be created and hosted. While it was possible to create arbitrary survey items, some restrictions specific to the EFLA use-case applied (for a full list of features see table 2).

The original version of the survey tool was written in Python 3 and JavaScript for the server and user interface respectively. To be able to re-use code, the choice of language did not change with the new version.

### 2.2.1 Data Model

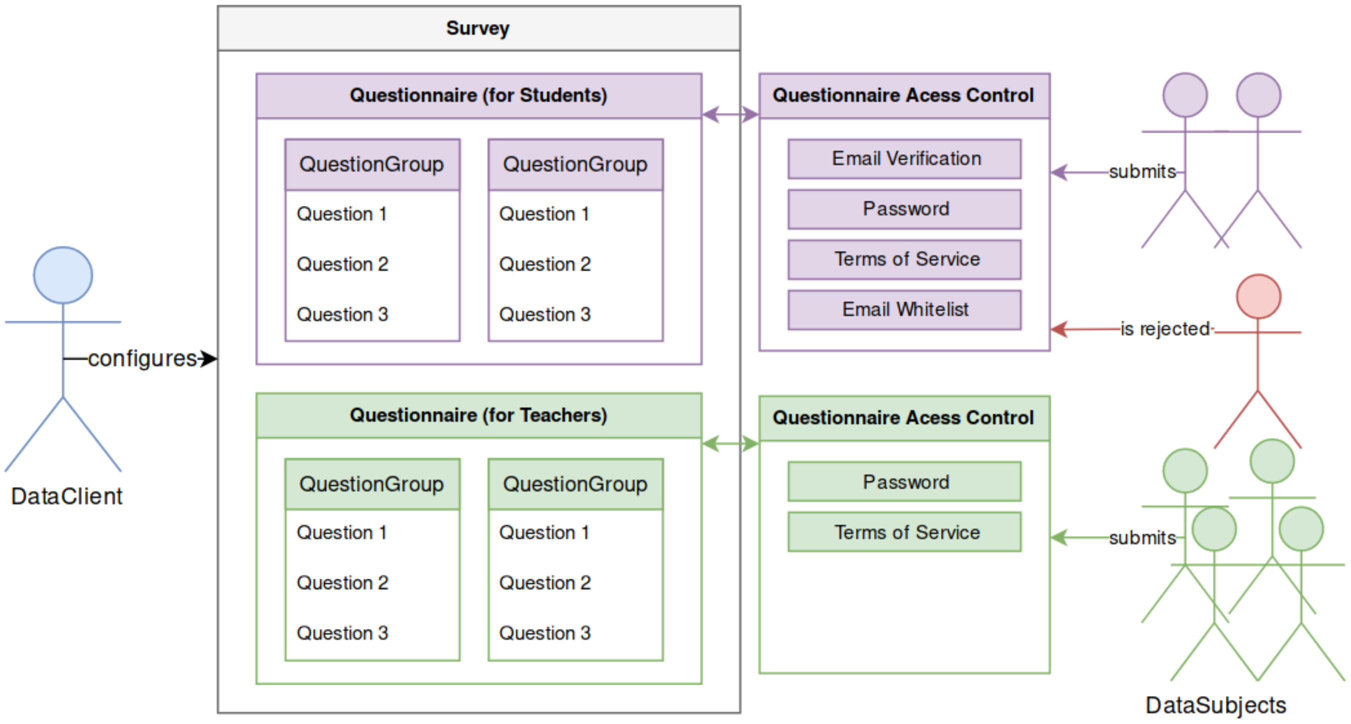
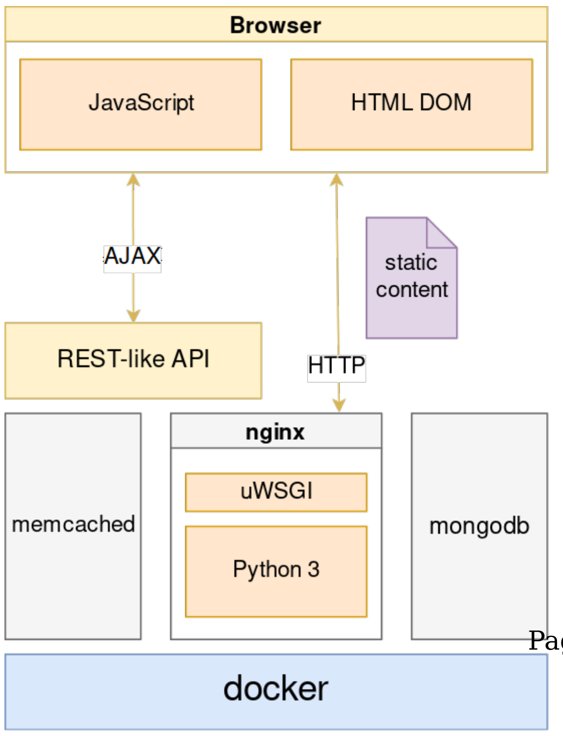


Figure 3: Data model of version 1.0.0

The data model for version 1.0.0 is closely coupled to the specific needs of the EFLA survey, where a single survey consists of one or more questionnaires, targeting multiple audience groups. Each questionnaire controlls DataSubjects' rights to submit by it's own set of access control modules, so that audience groups can be discriminated. Each questionnaire contains one or more question groups, which contains of one or more questions. All questionnaires within a survey would be published and retracted at once.

### 2.2.2 Architecture

The architecture for version 1.0.0 follows a simple client-server paradigm, where a web browser takes the role of the client. The server software consists of an application server, responding to API requests, a database and an in-memory key-value store for storing session data. There are two different user interfaces, one for data clients and one for data subjects. The data subjects' user interface is the page, where the survey is filled out and submitted. This page is rendered on the server using a templating engine and sent to the browser as a static HTML document. The user interface for data clients presents an editor, where survey content and settings can be created and modified. This user interface is realised as a one-page javascript application and was developed by



Hannes Leutloff. Deployment is handled through Docker, with the help of the docker-compose tool. The different parts of the server bundled up as separate docker images, which may be deployed on a docker host using docker-compose.

## 2.3 Goals

Describe goal of the thesis here -> extend software -> integrate with existing platforms

## 2.4 Scope

In this thesis, the overall structure of the survey platform and the server-side software stack, which was designed and developed by Noah Hummel is discussed. Implementational details are discussed, if they are central to the workings of the software, or solve a particular conceptual challenge.

The user interface was designed and mainly developed by Hannes Leutloff, and only portions contributed by Noah Hummel will be discussed.

## 2.5 Methods used

Although this text follows a fairly linear structure, development focused mainly on rapid prototyping and did not follow the waterfall model of software engineering. Requirements were prioritized based on risk, meaning that more complex requirements and requirements for which the concept was not initially clear were implemented first. For these more risky requirements, development largely concentrated on creating a proof of concept first and then incrementally improving the prototype based on shortcomings of the previous iteration. Although the requirements did not change much during development, concepts changed constantly, which makes it hard to clearly separate concept from implementation. At some points, implementational details have to be discussed in the concepts section, as a previous iteration of the implementation informed the final concept.

# 3 Requirements Analysis

Requirements analysis was performed on the basis of previously collected use cases. Required features and parts of the existing software that had to be refactored were identified.

## 3.1 Use Cases

### 3.1.1 Motivated Strategies for Learning Questionnaire and Others

The Motivated Strategies for Learning Questionnaire is a psychological survey for accessing learning strategies. -> Why is it interesting? -> Write something about the self-regulation dashboard -> Anything I can cite on this? It is not possible to create the MSLQ with version 1.0.0 of the survey tool, as it only supports answers on a scale from one to ten, whereas the MSLQ uses a scale from one to five. -> Really? Also, for some other surveys which are of interest -> find examples <-, the order in which questions are presented to the data subject has to be randomized each time the survey is taken.

### 3.1.2 Embedding in Moodle

Moodle is an LMS used at Goethe University. The platform supports embedding of external tools inside of a course context. Because previous and simultaneous efforts by Prof. Drachsler's work group and collaborators already target Moodle, the new survey tool should also integrate with it. Integrating with Moodle also allows the survey tool to leverage Moodle's integration with the universities central authentication service to perform single sign-on.

#### 3.1.3 Stand-Alone Survey Platform

In addition to directly embedding the survey tool into an LMS, it should be possible for a data subject to participate in a survey, if no LMS is used. This use case was carried over from version 1.0.0.

#### 3.1.4 Data Provider as Part of the TLA Infrastructure

Prof. Drachsler's work groups' current efforts are targeted towards the implementation of a trusted learning analytics (TLA) infrastructure. This infrastructure will provide big data storage and analysis while complying with the european general data protection regulation (GDPR). The new survey platform should publish collected data to the TLA facts engine. Data from the survey tool may then be corellated with data collected from other sources to form a more complete picture on the data subject's learning behaviour.

### 3.2 Refactoring & Restructuring

#### 3.2.1 Database Abstraction

The previous version of the survey tool uses a self-authored database abstraction library, the object-document-mapper (ODM). This library is capable of persisting python objects and relationships between them. It does this by serializing objects to JSON and storing using mongodb. Runtime interactions with these objects are intercepted and converted into database queries using the descriptor pattern (https://docs.python.org/3.7/howto/descriptor.html). This approach worked well for the limited use case, but it proved difficult to implement a transaction model that follows ACID properties.

To achieve transaction safety without compromising API transparency, an existing object-relational-mapper (ORM) was chosen to replace the ODM. Key factors for this choice were:

**Maturity of the project.** Data integrity is critical to the application. There is an increasing chance of bugs being found and resolved with increasing project age.

**Support for inheritance mapping.** Not being able to effectively use class hierarchies had proven itself to be a major burden on development speed during development of version 1.0.0, as attributes of related classes had to be explicitely duplicated. This was not only confusing to read but also difficult to maintain.

**Support for JSON or hash map columns.** JSON columns provide a convenient way to store multiple translations of a text column, without producing additional joins between the main table and supplemental translation tables. Storing multiple representations of a string inside a relational database usually involves a 1 to n relationship between the internationalised entity and it's translations. With the use of JSON columns, multiple translations of a string can be stored as a dictionary inside a single column of the entity itself.

In table 5 you can see a comparison between the some of the most popular ORM libraries available for Python 3. Because of the project age, previous experience with the library and it's reliability in production environments and the large feature set, SQLAlchemy was chosen to replace the ODM.

| Feature \ ORM | SQLAlchemy | PeeWee | PonyORM | SQLObject |
|---|---|---|---|---|
| Declarative API | Yes | Yes | Yes | Yes |
| Eager loading of relationships | Configurable | No | Configurable | No |
| Lazy loading of relationships | Configurable | Yes | Configurable | Yes |
| Query caching | Yes | Yes | Yes | Yes |
| Cascades | update, delete | update, delete | delete | delete |
| Inheritance mapping | Yes | No | Yes | No |
| JSON columns | Using PostgreSQL | No | Yes | Using PostgreSQL |
| First released | 2005 | 2010 | 2014 | 2003 |

Figure 5: Comparison of python ORM libraries

### 3.2.2 User Interface

As a parallel but somewhat separate effort, a major rewrite of the user interface was done by Hannes Leutloff. The user interface was re-written using VueJS. This was not explicitely required, but made integrating new features into the user interface easy.

## 3.3 Survey Content

To allow other surveys apart from the EFLA survey to work well with the platform, the following requirements were established:

**Response ranges should be adjustable.** Responses will still be on a discrete scale with a step size of 1. Start and end of the scale should be adjustable.

**Labels for the response ranges should be adjustable.** The EFLA survey used the phrases “Strongly disagree” and “Strongly agree” to describe the lower and upper ends of the response scale respectively. Other surveys may use different descriptions, hence these descriptions should be adjustable.

**Question order should optionally be randomizable.**

## 3.4 Template Management

It became clear during the initial meeting, that the set of questionnaires which are of interest is changing over time. As new research is published, and questionnaires become better statistically validated, the number of survey items needed to access a certain reasearch question ususally decreases. For these reasons, more current questionnaires can be preferred over older ones and it should be easy to integrate new survey content in the future. At the same time, once a questionnaire is reasonably well verified, multiple users may be interested in using the same questionnaire for different groups of participants. Version 1.0.0 had rudimentary template support, but templates are supplied as static YAML files on build and can not be changed or updated from the user interface. To overcome this limitation, the following requirements were established:

**Templates should be user-contributed.** Data clients with special privileges, called contributors, should be able to publish new templates using just the user interface.

**Individual survey items may be templates.** To create slightly modified versions of a questionnaire, it is useful to re-use existing survey items and not just entire questionnaires.

**Templates should be modifiable after creation.** Otherwise, editing errors would cause the template to become unusable and the entire template has to be created again.

**Changes should be traceable.** When the maintainer of a template makes modifies the template, other users who own copies of it should be able to review these changes.

## 3.5 Support for xAPI

Version 1.0.0 only supports limited data analysis capabilities, such as export of all response data for a given questionnaire as CSV and a rudimentary box plot visualisation. The new version should interface directly with the TLA infrastructure to allow for further data processing by TLA's analysis engine. TLA uses the xAPI statement format as it's common data representation. To provide data using xAPI to TLA, the following requirements were extracted:

**Survey results should be published to an LRS via xAPI.** A suiting xAPI representation of survey results has to be defined and should be transmitted to the LRS vie HTTP as defined by the xAPI specification (2).

**The destination LRS should be configurable.** This allows the location of TLA to change in the future. It also decouples the survey tool from TLA and allows interoperability with other LRSs.

**xAPI statements must not be lost due to failure.** When publishing data to an external storage, data integrity is no longer just determined by database integrity. Appropriate measures must be taken to ensure re-transmission in case of network failure. If transmission is not possible due to misconfiguration, an offline fallback method has to be provided.

## 3.6 Embedding & LTI Launch

To achieve compatibility between Moodle and the survey tool, the following requirements were established:

**The survey tool has to implement LTI.** LTI is used to launch external tools by Moodle. The survey tool has to recognize LTI launch requests and respond with an embeddable version of the survey.

**Data subjects should not have to authorize.** When using the survey tool from within Moodle, data subjects have already authorized with CAS in order to log in the LMS. Requiring another form of user authentication after this point would break seamless user experience.

LTI uses OAuth 1 to sign requests. This provides a way for the tool provider to verify the identity of the LMS and the authenticity of the LTI request. It was explicitely not required to implement OAuth as part of this work, but the design should take into account that future integration of OAuth should be easy.

## 3.7 Privacy Considerations

Since 25 May 2018, the general data protection regulation applies to members of the EU (1). The author is not in any way qualified to provide legal interpretation on the regulation. However, in consultation with Prof. H. Drachsler the following requirements were identified to conform with the regulation:

**Data deletion for data subjects.** Some personal data has to be stored on the server in order to identify data subjects between requests. Because of the right to be forgotten, there has to be a way to delete personal data stored for a specific data subject. It is sufficient if this is not automated and can be performed by the site's administrator.

**No user accounts are available to external data clients.** If third parties are allowed to use the service, we can not guarantee that they will follow GDPR guidelines. To avoid responsibility for third parties' actions on the site, there will be no way to sign up as a data client that does not involve contacting the administrator.

Other rights covered by the GDPR include the right to view and export personal data. Since all personal data collected is published to the TLA infrastructure, this will be implemented as part of the TLA trust engine and is not part of the survey tool.

# 4 Concept

## 4.1 Architecture

The overall architecture is the same as for version 1.0.0. The survey tool uses a classical client-server approach, where the client is a javascript application running inside a web browser. This approach enables everyone with a modern web browser to use the platform without having to install any additional software locally. The server-side software stack is deployed by using docker and exposed through a containerized web server. Communication between different containers on the server takes place on a virtual network which is not exposed to the internet. Communication between client and server is handled by a RESTful API which is provided by the server.

## 4.2 Survey Content

During refactoring, the top-level organisational unit "survey" was removed and replaced by "questionnaire", as grouping multiple questionnaires inside a single survey only makes sense in a small set of use cases. For other use cases, this has proven to be confusing to users. Most of the time, a survey contains only a single questionnaire. Use cases, where grouping of multiple questionnaires into a single survey is appropriate can still be achieved by creating multiple questionnaires without any explicit grouping. The overall survey structure did not change significantly between versions 1.0.0 and 2.0.0 or above.

### 4.2.1 Questionnaire

A questionnaire is a collection of dimensions, along with administrative information. A questionnaire's life cycle is modeled using three distinct states. The questionnaire starts out as not published, which means that it is only accessible to it's owner via the API (with the exception of templates, which will be accessible to other authenticated data clients) and will not display when accessed by data subjects via it's public URL. This state is meant for questionnaires which are incomplete and worked on. Once a questionnaire reaches it's published state, it will be visible to the public via it's public URL. It will not accept submissions via the API at this point. The publicly accessible representation of the questionnaire will not display form controls to submit and display an information page before accessing the survey content, informing the data subject that submissions are not accepted at this point. Once data subjects should be allowed to submit answers, the questionnaire enters the "accepts submissions" state. The public page will now display form controls to submit and new submissions are accepted by the API. Once the survey has concluded, the questionnaire returns to the "published" state and can be viewed for future reference. If this is not wanted, the data client has the option to retract the survey and return it to it's "NOT published" state. This life-cycle model is illustrated in figure 6.

Along with information about it's life cycle, the questionnaire also stores information about any access control challenges presented to data subject. As in the previous version, challenges are presented as additional form inputs when submitting. In contrast to the previous version, email addresses are now always validated when submitting by sending a one-tim-use token to the entered email address. Data clients may choose from these challenges:

**Password:** A password has to be entered in order to submit. The password is chosen by the data client.

**Email whitelist:** Only emails that are present in a list of email addresses are allowed to submit. The email whitelist also supports wildcard expressions to allow all email addresses following a certain schema.

**Email blacklist:** The same as email whitelist, but instead of allowing certain email addresses, this blocks certain addresses from participating.

### 4.2.2 Dimension

A dimension is a collection of questions usually pertaining to a specific topic. The only additional piece of information handled by the dimension is whether the order of it's questions should be randomized when the survey is taken.

### 4.2.3 Question

A question is a single statement to which the data subject may respond on a numerical scale. Lower and upper bounds of this scale may be adjusted by the data client, as well as the descriptions for these bounds. Individual questions may be used as templates, which is why the information on lower and upper bounds and scale labels is stored on a per-question basis. Convenience workflows for updating this information for an entire dimension allow easy editing despite the great granularity of this approach.

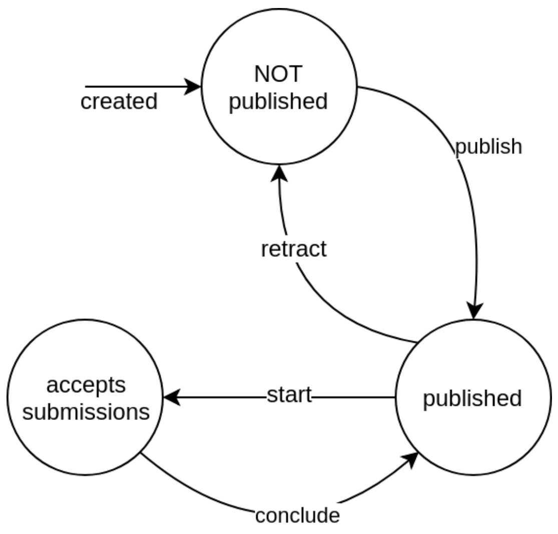


Figure 6: Questionnaire life-cycle

## 4.3 Internationalisation

Every time an API request is made, a request language is determined by the server and the request is handled in the request language. The client may explicitly request that a certain language should be used. For human readable attributes of survey items, multiple translations may exist. When a survey item is created, the language it is created in becomes the item's original language. Translations in different languages may be added later by updating the survey item using the desired language as the request language. When no translations for the request language exist, survey items are served in their original language instead. To communicate information about existing translations and what translation was served, the API includes the item's current language, original language and a list of available languages in the JSON representation of the item.

## 4.4 Ownership, Parties, Roles

To control API access to survey items, survey items had a single owner in versions 1.0.0. Owner refers to a data client who has access to the resource. In version superceding 1.0.0, the ownership model was expanded to allow n to n relationships between owning parties and owned resources. Ownership is also no longer restricted to data clients. These changes became necessary for two reasons. Survey item are no longer the only resource with access restrictions, as will become clear in section **??** "Mutation tracking". While survey items ususally only have a single owner - the author - tracking information has to be accessible to all parties sharing read access to the tracked resource. Because of the templating system, data clients may have read access to survey items of which tey're not the author (templates). This requires a single resource to have multiple owners. Of course, a party may own more than a single resource, hence the n to n cardinality of this relationship. The reason for expanding ownership to data subjects is that the mechanism makes it easy to identify personal data. Data subjects by default own all resources which contain personal data. The "right to be forgotten" may then be implemented by simpy retrieving all owned objects for a given data subject and deleting them from the database.

For some priviliged users, access control should not be enforced. Also, the right to publish templates is reserved for a select group of users. To accomplish different levels of privilege, roles were introduced to all parties. A party may have a number of different roles, depending on the actions they may take and the data they may access. There are currently five different roles:

| Role | ID | Description |
|---|---|---|
| Root | 0 | All available access control methods grant access to users with this role by default. |
| Admin | 10 | An administrative user, who may view and modify other user's data in order to ensure proper operation of the platform. |
| Contributor | 20 | A user who may publish survey items as templates for other users. |
| User | 30 | A regular user who may create and modify their own survey items. |
| Unprivileged | 40 | A user who may view and participate in published surveys, but not create or modify survey items. This role is used for all data subjects. |

Access control may then be enforced by checking if the role required for a certain action is held by the user in question.

## 4.5 Modification Tracking

To track modifications of survey items, every time a modification is made to an item, a record of this modification is stored in the database. To keep storage space needed for this feature to a minimum, only the most recent modification for each attribute is stored. During development it was discovered that different kinds of modifications require different information to be stored in order to create a meaningful record of the change, that can also be presented to the data client. Such information includes: The modified item, the modifying data client, previous and new values, as well as the point in time when the modification happened. For the workflows discussed previously, five different types of records were identified:

| # | Description | Stored information |
|---|---|---|
| 1 | An attribute was updated | Modified item, modifying data client, attribute name, previous value, new value, timestamp |
| 2 | A language map was updated | Modified item, modifying data client, attribute name, modified language name, previous value, new value, timestamp |
| 3 | A child item was added | Parent item, modifying data client, child item, timestamp |
| 4 | A child item was removed | Parent item, modifying data client, child item name, timestamp |
| 5 | A questionnaire was removed | Name of the questionnaire, modifying data client, timestamp |

The modified item is stored as a reference to the actual record of the item in the database. This makes it possible to quickly find the modified item based upon a tracking record. In the user interface, this is used to display the modified item as a clickable link which will show the modified item instantly. For types of modifications where an item was deleted, this kind of data model is not applicable, as the referenced record won't exist in the database anymore. In these instances, only the name of the item is stored. A special case exists for the deletion of questionnaires, as they don't have parent item needed to construct the tracking record. To deliver a personalized stream of modifications to data clients, including only those modifications which are of interest to them, each tracking records also use the ownership model.

## 4.6 Template Management

To allow user-contributed templates which may also be modified after creation, the same data structures are used for templates as for regular survey content. Every survey item may then optionally also be a template. Creating copies of these templates is done by reference instead of by physically copying the template's content. The reasons for this is, that template content may be modified at any time, and these modifications have to be propagated to all existing copies. If copies were created by physically copying the template's content, updating a template would also cause all copies to be updated in the database. The asymptotical time complexity of this approach scales linearly with the number of copies present and potentially produces large join operations in the database. It also stores large amounts of redundant data. Instead, copies are proxy objects, which delegate read operation to the referenced template. Survey items containg actual data are called concrete instances, wheras copies which do not contain the actual data are called shadow instances. Any concrete may be declared as a template by a data client who has at least the `Contributor` role. Templates are visible to all data clients.

### 4.6.1 Creation of Shadow Instances

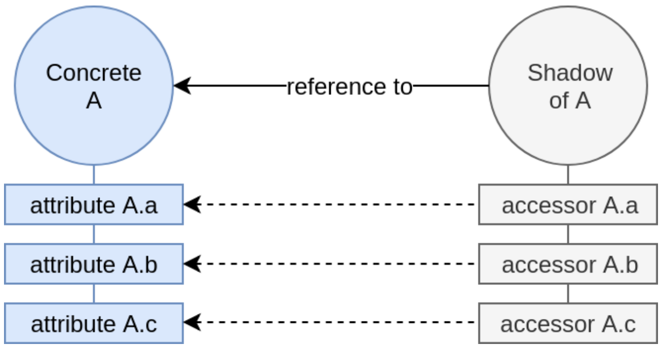
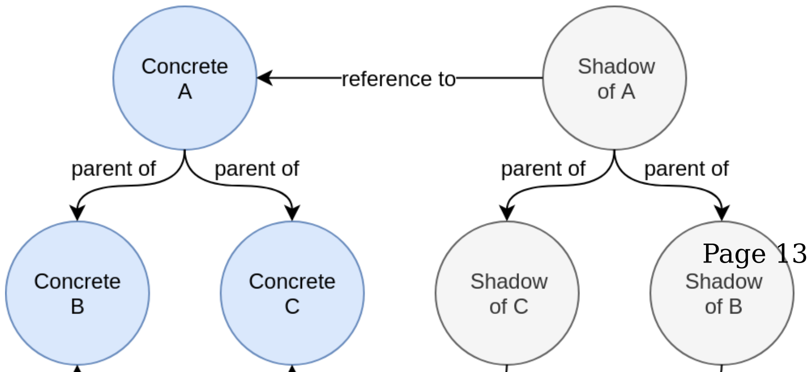


Figure 7: Schematic depiction of the relationship between concret & shadow instances

Creating a shadow instance from a concrete instance which does not have any relationships to other items is trivial. When creating a shadow instance from a concrete instance which has children, the children are considered to be a part of the content. This is true, since all parent-child relationships are aggregations; a dimension has no purpose without any questions and a questionnaire has no purpose without any dimensions. This means that the resulting shadow instance should also duplicate the relationship structure of the reference concrete instance. In figure 8 an example for this is given. In the example, shadow A was created from concrete A. We model parent-child relationships as a directed graph, with edges from parent to child, for the purpose of this example. The creation of shadow A will cause the entire subtree starting from concrete A to be duplicated with shadow instances, pointing to their corresponding concrete counterparts. The result is a copy of the concrete instance and it's relationship, which is always in synchronism with it's source, but can not be modified.

A special case occurs, when a shadow is created from a concrete, whose subtree also contains shadows. While it would be possible for a shadow insrance to reference another shadow instance, ashadow in-



stance should always reference a concrete instance directly. The reason for this is the time complexity of read accesses to the shadow instance. If a shadow references another shadow, the `reference_to` relationship would have to be traversed multiple times until the concrete instance is found. Access times would then scale linearly with the degree of separation between the accessed shadow instance and the referenced concrete. To avoid this, if a shadow instance is encountered while traversing a concrete's subtree on shadow creation, the shadow's `reference_to` relationship is traversed a copy of the referenced concrete instance will be created. The example in figure 9 depicts this case. In the example, shadow A was created from concrete A. Concrete A's subtree contains a shadow instance, shadow #1 of B, which should be duplicated in shadow A's subtree. Instead of pointing shadow A's second child to concrete A's child, shadow #1 of B's reference relationship is followed and becomes the referenced template for shadow #2 of B. This approach keeps the degree of separation between shadows and concretes always at one.

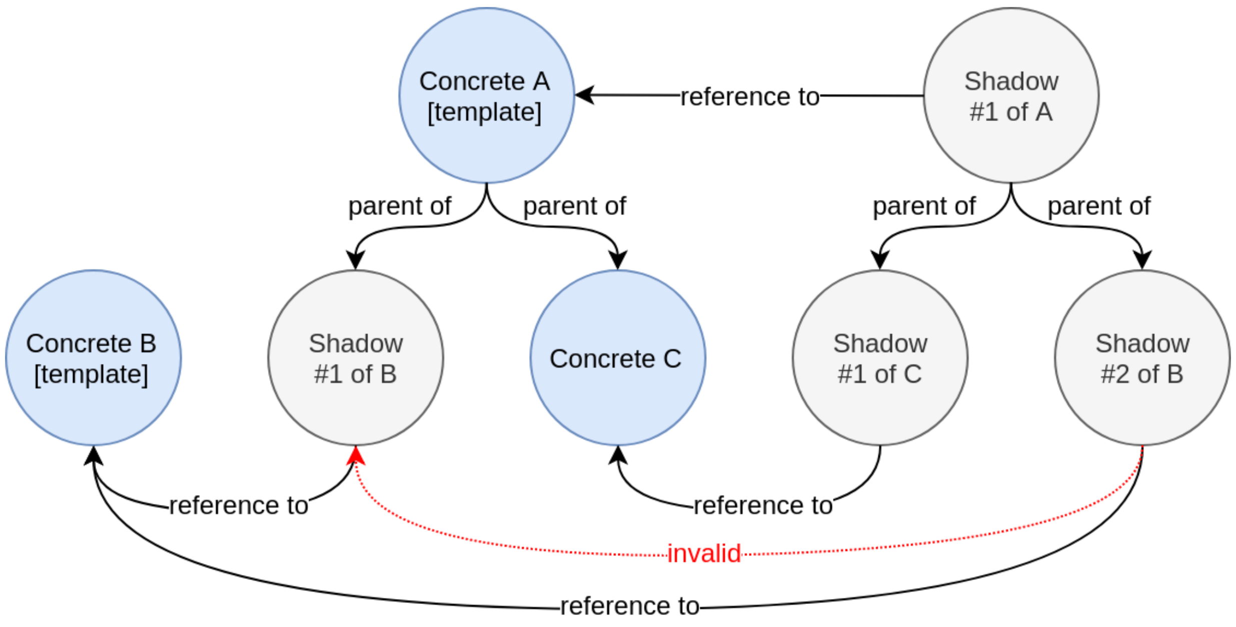


Figure 9: Duplication of a concrete structure which contains shadow instances

### 4.6.2 Modification of Templates

Modifications of template content is trivial, as shadow instances will always read from the referenced template. When modifying a template's relationships to it's children, the changes have to be duplicated in all copies of the template. To achieve this, the `reference_to` relationship is traversed backwards to find all copies of the template. The copies are then notified of the modification and apply it to their children as well.

### 4.6.3 Deletion of Templates

When a template is deleted, copies of it become invalid, as they don't have any instance to point to anymore. Two possible solutions for this are either deleting all associated shadows, or converting all associated shadows into concrete instances. The former is less complex implementation wise, while the latter is more user-friendly, as it doesn't result in unexpected deletion of content. There are also use cases, where deleting all associated shadows might be intended, for example if the survey item is retracted and should not be used anymore. There is no default behaviour which satisfies all use cases. For this reason, a compromise between the two approaches was made. By default, when a template is deleted, all associated shadows are also deleted. Contributors are therefore encouraged to edit existing templates instead of deleting them. Templates will also show a counter showing the number of associated shadows in the user interface, making the contributor aware of possible repercussions. If the contributor chooses to delete a template anyway, the modification tracking feature will provide accountability and transparency to the affected data clients. When deleting a template questionnaire however, the associated shadows are converted into conrete instances, as the questionnaire might already be published and available for participating. In this case, deleting the questionnaire would interfere with another data client's survey and would delete already collected results, which is not acceptable.

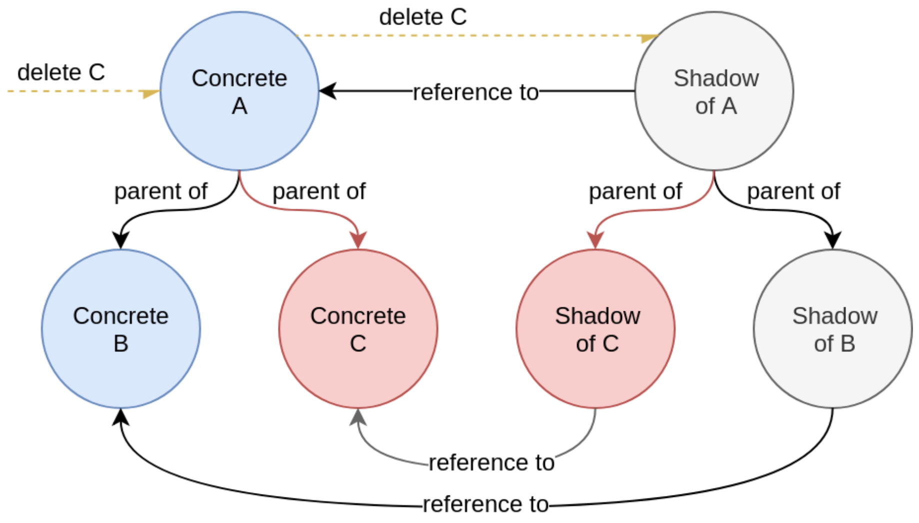


Figure 10: Modifications of templates are relayed to copies, items marked in red are deleted.

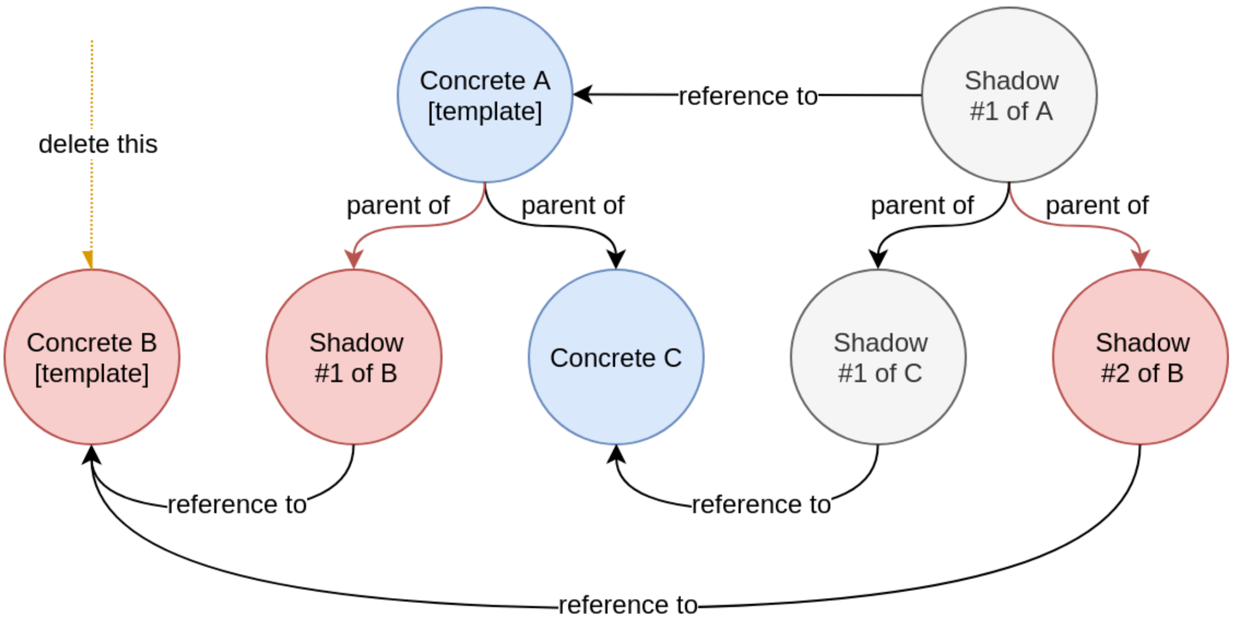


Figure 11: Deletion of a template is propagated to all associated shadow instances

### 4.6.4 Modification Tracking for Templates

The modification tracker for a given data client should not just show changes to concrete instances which are owned by the data client. Rather, it should also include changes made to templates, of which the data client owns copies of. This provides accountability and transparency to data clients who choose to use templates. As changes are immediately applied to copies, this is an important feature for consistent user experience. To achieve this, every time a modification is made to a template, ownership of the resulting tracking record is not only given to the template owner, but to all owners of associated shadow templates.

## 4.7 Integration with xAPI

### 4.7.1 Introduction to xAPI

xAPI, formerly known as TinCanAPI, is a data exchange standard closely resembling activity streams (https://www.w3.org/TR/activitystreams-core/). It was developed by the Advanced Distributed Learning (ADL) Intiative as a successor to SCORM and allows the exchange of experiential information in the form of statements. These statements use JSON as their data format and include a minimum of three semantical objects, “actor”, “verb” and “object”. The data is transmitted using HTTP, following REST principles.

```json
{
    "actor": {...},
    "context": {
        "contextActivities": {
            "grouping": [
                {...}
            ],
            "parent": [
                {...}
            ]
        },
        "extensions": {
            "http://activitystrea.ms/schema/1.0/place": {...}
        },
        "language": "en",
        "platform": "st3k101 via localhost"
    },
    "id": "896ef7f1-8d5c-4729-b028-e9d72df47fe8",
    "object": {...},
    "result": [
        {...}
    ],
    "timestamp": "2018-09-27T14:32:15.009513",
    "verb": {...}
}
```

Figure 12: Anatomy of an xAPI statement

Every xAPI statement may also include a “timestamp” stating the issue date and time of the statement, a “context”, providing additional information about the event or a “result”, detailing the outcome of the event. The format is also extensible, additional information can be provided in the “extension” object.

### 4.7.2 xAPI Statement Design

There are several actions which will trigger sending of an xAPI statement:

1) A data client logs in into the survey platform.

2) data subject launches an embedded survey.

3) data subject answers a single question.

4) data subject answers a known survey item (whole questionnaire or dimension)

5) data client updated the xAPI activity ID of a survey item.

Figure 13: List of cases where xAPI statements are emitted

For each of these cases, an xAPI statement had to be designed. All of these statements share at least some information. The `context` object of all xAPI statements emitted by the survey tool includes the `platform` and `language` and `extensions` attributes. The `language` attribute always contains an RFC 5646 (https://tools.ietf.org/html/rfc5646) compliant representation of the language the action was performed in. The `platform` attribute always contains the string `st3k101`, identifying the origin of the statement. The `platform` attribute may also include the URL of the source LMS in the case LTI was used. In this case, the URL is prepended as `st3k101 via SOURCE_LMS_URL`. The `extensions` obejct of the `context` also includes a geolocation for the client in RFC 7946 compliant GeoJSON format (https://tools.ietf.org/html/rfc7946). Below is a concept for what additional data should be included in the statements listed in 13.

| # | Actor | Verb | Object | Result | Context |
|---|---|---|---|---|---|
| 1 | data client | logged in | login page | - | - |
| 2 | data subject | accessed | questionnaire | - | - |
| 3 | data subject | answered | question | response value | parent dimension AND questionnaire |
| 4 | data subject | answered | qestionnare OR dimension | response value | parent item, if any |
| 5 | data client | updated | questionnaire OR dimension OR question | new acitivity ID | - |

Table 1: Concept for information included in emitted xAPI statements

Objects are identified by their `objectType`, `type` and `id` in xAPI, whereas verbs are just identified by their `id`. For the purpose of the survey tool, all objects share the same `objectType`, the `Activity`. The verb’s `id` is semantically equivalent to an object’s `type`. It is common practice to use a URL as identifier or type, which will return a human readable description of the item via HTTP GET. In theory, there’s no correct identifier to use when designing xAPI statements, as the standard does not prescribe the use of any specific verbs or objects. In practice, several registries with commonly used verbs and objects exist and should be consulted when choosing which identifier or type to use. This avoids re-definitions of already existing items and increases homogeneity among statements by different adopters of the

| Verb | Identifier |
|---|---|
| logged in | `https://brindlewaye.com/xAPITerms/verbs/loggedin` |
| accessed | `https://w3id.org/xapi/dod-isd/verbs/accessed` |
| answered | `http://adlnet.gov/expapi/verbs/answered` |
| updated | `http://activitystrea.ms/schema/1.0/update` |

| Object | Type |
|---|---|
| login page | `http://activitystrea.ms/schema/1.0/page` |
| questionnaire | `http://id.tincanapi.com/activitytype/survey` |
| dimension | `http://fantasy.land/dimension` |
| question | `http://adlnet.gov/expapi/activities/question` |

Table 2: Used xAPI verb identifiers and object types

```
"object": {
    "definition": {
        "description": {
            "en-US": "This is a particular scale of a survey, it usually contains
                multiple questions."
        },
        "name": {
            "de": "5. Anstrengung"
        },
        "type": "http://fantasy.land/dimension"
    },
    "id": "<bla@blubl.net>:lernstrategien_wild_schiefele--5_anstrengung",
    "objectType": "Activity"
}
```

Figure 14: Example of how a dimension is represented as an xAPI acitivity object

standard. For this reason, the verbs and objects used in table 1 had to be translated into already existing verbs and object. The results are detailed in table 2. For some of the objects, no suitable definitions existed. For those objects, dummy identifiers or types were used, which follow the URL format, but use `http://fantasy.land/` as a prefix.

In order to corellate objects in emitted xAPI statements with survey items in the survey tool, all survey items have a user-modifiable xAPI acitivity ID associated with them. This identifier is used as the object's `id` in xAPI statements. These identifiers are not modifiable in copies of templates, which means that all instances of a copy will use the same xAPI activity ID. This is useful for conducting meta-analyses, where all results for a certain template could be included in the analysis, regardless of who conducted the survey. During testing, this presented itself as an issue, because results for the same template, which were collected by different data clients would not be distinguishable, as they all used the same identifier. For this reason, the email adress of the data client who conducts the survey is added as a prefix to all xAPI activity IDs before sending. -> Drawback: not normalized, potential fix: use Instructor <- In figure 14, an example of how survey items will be represented in xAPI is given.

Actors may be represented in three different ways using xAPI. The identifying feature is either the person's email adress in the case of the `mbox` and `mbox-sha1sum` actor types, an account id in combination with a URL where the account is located in the case of the `account` actor type or OpenID credentials in the case of the `openid` actor type. Data clients are represented as `mbox` actor types, while data subjects may be represented by as `mbox-sha1sum` actor types or, in the embedded use-case, as an `account` actor type using their LTI user identifier. The latter is necessary, because the email address might not be available though LTI. The LTI user ID is, contrary to prior suppositions, not useful for data analysis, as Moodle will use the user entities database ID. Moodle does however communicate a username via LTI, which for the specific Moodle instance at Goethe University is the same as the data subject's CAS ID. To retain compatibility with other LMSs while taking this finding into account, the username will take precedence over the LTI ID, if it's present in the LTI request. The LTI ID is also not globally unqiue, as there may be separate LMSs which use the same IDs for different users. For this reason, the LTI user ID is prefixed with the identifier of the source LMS, which is always present in LTI requests.

The recipient of the statements is determined by the object which is acted upon. This makes intuitive sense, as the person owning a certain survey item is the one interested in collecting data on it. All survey items are on the highest level organized into some questionnaire and there's no use-case for different data consumers for individual parts of the questionnaire. For this reason, recipients are configured on a per-questionnaire basis. For objects which don't have an owner, for example the survey tool's login page, a default recipient is configured system-wide.

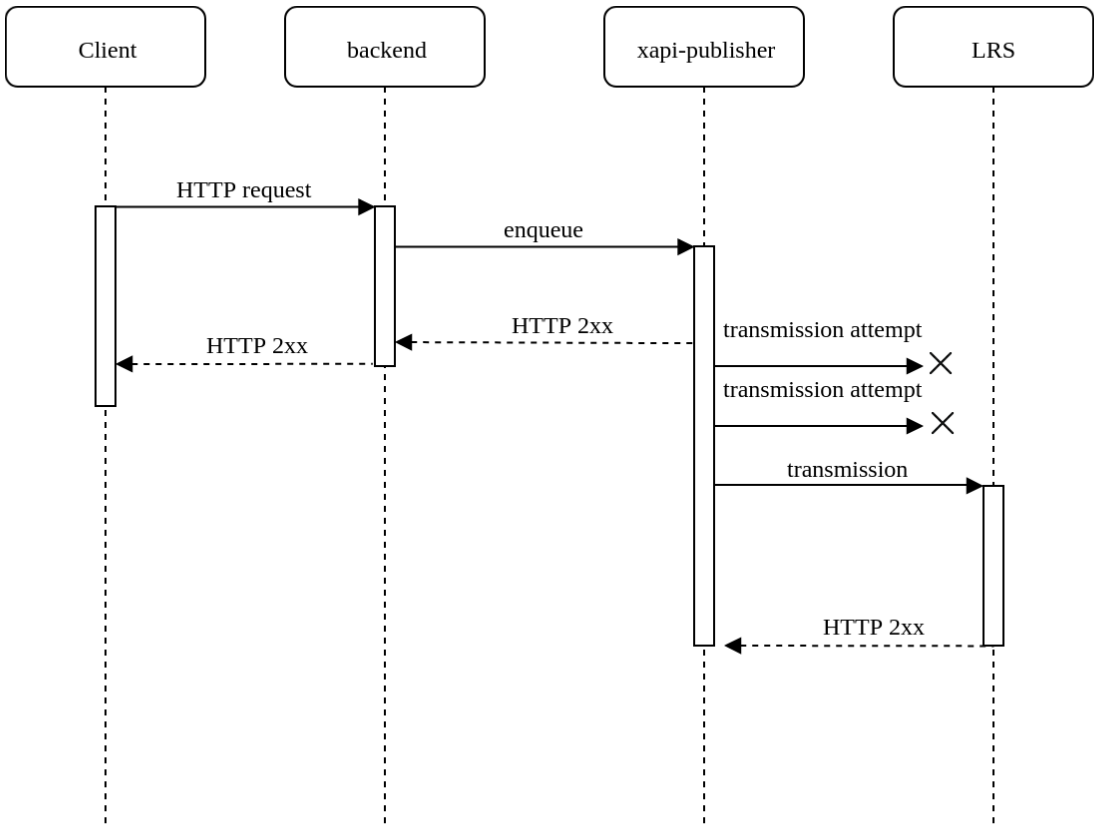
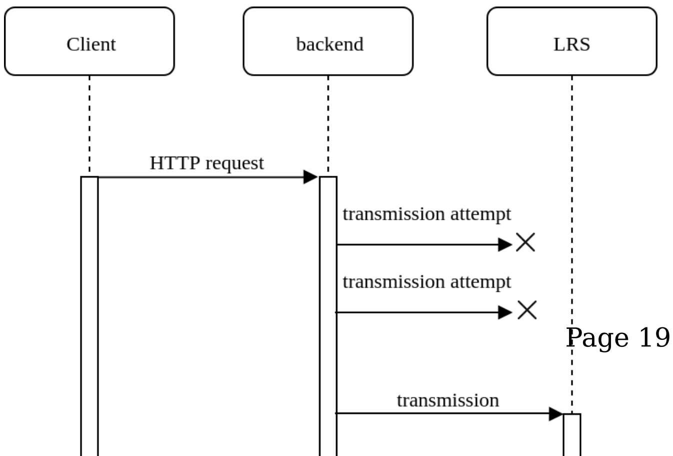


Figure 16: Asynchronous (non-blocking) transmission of xAPI statements

Sending of xAPI statements is separated from the API, as sending and possible re-transmissions should not block the API from responding to requests. This was discovered during testing, when a misconfigured xAPI recipient was used. HTTP timeouts are usually in order of seconds and unsuccessful connection attempts are retried. This resulted in the site becoming unresponsive when



submitting the survey, as the submission of a survey would cause a large number of xAPI statements to be sent. Figures 15 and 16 illustrate this issue. To recover from failure in the case that an xAPI statement can not be transmitted over a prolonged period of time, the statement as well as the recipient and timestamp of the failure are logged to file and may then periodically be recovered.

### 4.8 Support for LTI

The embedded user interface is launched by the LMS using the LTI protocol. To identify the source LMS, a random token, the consumer key, is generated in the back-end for every questionnaire. To embed the survey within the course context, an LTI request with the correct combination of request URL and consumer key has to be made by the LMS. Information about the user is already present in the request body and is used to identify the data subject. Since there's some time between the LTI launch request and the survey submission, the request information needed to identify the user when submitting has to be stored on the server for this period of time. To achieve this, the required data is stored in the data subject's account. If a data subject accesses the service through LTI for the first time, a new account is provisioned for them. When launching the survey tool via LTI, a session is created for the data subject and a session token is embedded into the user interface. Actions by the data client in the embedded user interface will use the embedded session token for authentication with the API. This mechanism allows the API to identify the data subject on every subsequent request. A valid LTI request is treated as sufficient authentication for the data subject, as the source LMS already authenticated the user prior to them accessing the survey.

### 4.9 Email validation

When data subjects participate in a standalone survey, it is difficult to recognize repeated submission by the same person. Most features used for identification, for example the client's IP address or cookies present in the browser can easily be modified by the data subject. Even more advanced measures, like browser fingerprinting, are ultimately controlled by the client, as communication with the server is handled by a publicly available API. For this reason, a third party has to be involved in validating the user's identity. The survey tool achieves this validation through email. When submitting responses, the data subject has to enter their email address. At this point, the responses are stored on the server, but no xAPI statements have been emitted and the responses do not yet count towards the generated statistics. A randomly generated token is embedded into a URL pointing back to the survey tool. Ths URL is then sent to the entered email address. Once the data subject follows the URL, the server will associate the token with the user's responses in order to validate them.

### 4.10 Privacy Considerations

As mentioned in the section above, when a data subject participates in a survey, a user account is provisioned for them. Removal of this account and all associated personal data is performed via the API when authenticated as an admin user. In order for the admin to know which account to communicate to the API, the API allows admin users to query for existing user accounts. Data removal is limited to admin users, as removal of accounts by data subjects themselves would require some sort of authentication mechanism for data subjects. When an email address is present for the data subject, authorization via email is a possible solution for this. In this scenario, the data subject would receive an email with a one-time token, which can be used to remove their personal data. If there's no email present for the data subject, for example when the account was provisioned using LTI, or if the data subject hasn't got access to their email account, this mechanism fails. Hence, email as sole mechanism for data removal is not a viable option at the moment.

# 5 Implementation

## 5.1 Architecture

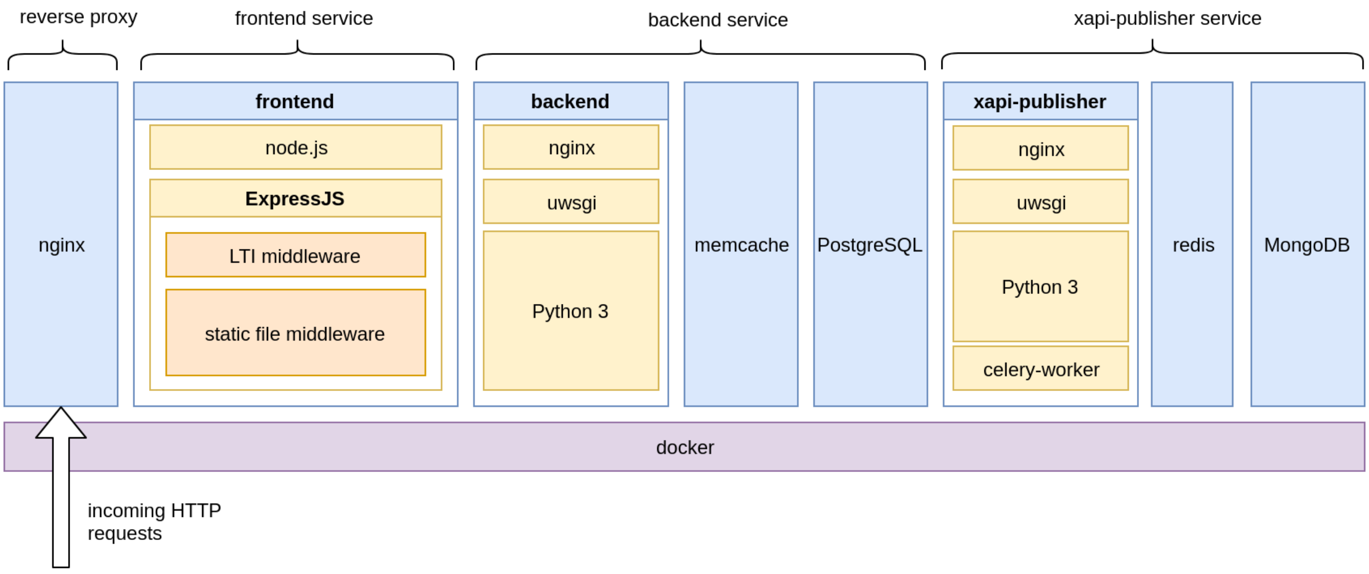


Figure 17: Server-side software stack

The server-side architecture is composed of multiple services. The "frontend" service is responsible for serving the user interface statically and intercepting LTI requests. The "backend" service is responsible for serving the API, user and session management and communication with the database. The "xapi-publisher" service is responsible for transmission of xAPI statements generated by the backend service. Each service uses slightly different technologies, depending on the requirements of the service. The API and "frontend" service are exposed through an nginx web server, which acts as a reverse proxy, forwarding incoming HTTP traffic to the appropriate docker containers.

### 5.1.1 The backend Service

The "backend" service is implemented using the "flask" library, which allows Python to interface with a web server using the web server gateway interface. Similar to PHP or CGI extensions, the nginx web server will accept incoming requests. It will then, using uwsgi, fork a python interpreter which will respond to the request. This means that no data can be persisted in the Python application itself, as the Python context will be discarded for every request. To persist data, a PostgreSQL database and a memcached in-memory key-value store are used. True persistent data, for example survey items, are stored in the database, whereas semi-persitent data like user session are stored in memcached. This design allows for horizontal scalability, as multiple instances of the "backend" containers can be used with the same database, as no data dependencies between the "backend" containers exists. The only bottleneck in this scenario is the shared database, which could be solved long-term by replacing the single PostgreSQL by a load balancing database cluster.

### 5.1.2 The `frontend` service

The “frontend” service consists of a simple node.js application running the ExpressJS web server. For the web server, two middlewares are provided, one for serving static files and another for intercepting LTI launches. This is necessary, since the user interface has to be configured for each LTI launch before serving it to the data client.

### 5.1.3 The `xapi-publisher` Service

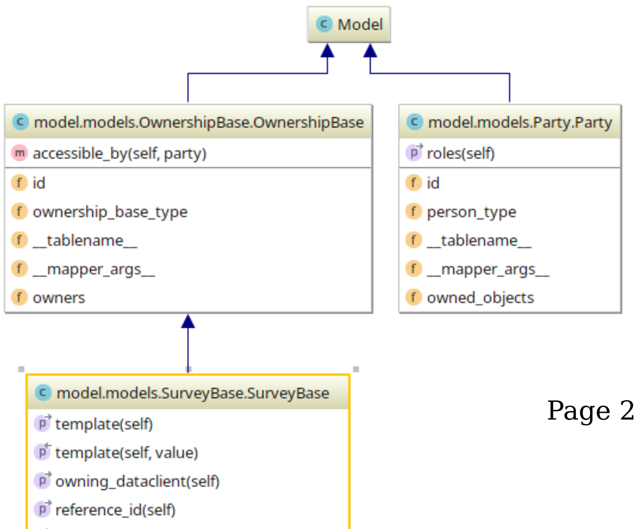
The “xapi-publisher” container mimics the design of the “backend” container closely. In addition to flask and uwsgi, it also runs several worker threads, which are responsible for asynchronous sending of xAPI statements. Since xAPI statements are JSON documents, a document based database is used instead of a relational database to persist xAPI statements between requests. For inter-process communcation between web server and worker threads, a task queue consisting of the celery library and the redis in-memory database is used.

The functionality provided by the “xapi-provider” container is seperate from the “backend” container, duplicating most of the architecture already present. This might seem sub-optimal at first, as it violates the DRY (don’t repeat yourself) principle to the extent that configuration for two seperate containers has to be created and maintained. It does however allow for better seperation of concerns. The “backend” container already handles business logic and user management and there is a unique set of challenges that has to be solved for publishing xAPI statements, which would unnecessarily increase the complexity of the “backend” service. One such challenge is, that sometimes data needed to create an xAPI statement is present in a request before the data is actually valid an can be sent. For example, when a data subject answers a survey using the stand-alone survey interface, their answers are submitted to the server but only become valid, after they have verified their email address. The required data to build the xAPI statement, including the data subject’s IP adress, has to be stored on the server until this point. In order to avoid convoluting the “backend” service’s data model and business logic to account for this, a seperate service solely responsible for temporarily storing and safely transmitting the xAPI statement is used. This allows the “backend” service to only use minimal logic for communicating with “xapi-publisher” service. It also allows maintainers of the codebase to make changes to the publishing behaviour without having to know the internal workings of the “backend” service.

## 5.2 Data Model

### 5.2.1 Class Hierarchy

The class hierarchy is centered around three abstract base classes, `SurveyBase`, `OwnershipBase` and `Party`. `OwnershipBase` is the base class used for all classes which can be owned by a `Party`. `Party` is the base classes used for all user-type classes. `SurveyBase` is the base class for used for all survey items. Each of these classes provide common functionality and interfaces to their subclasses.



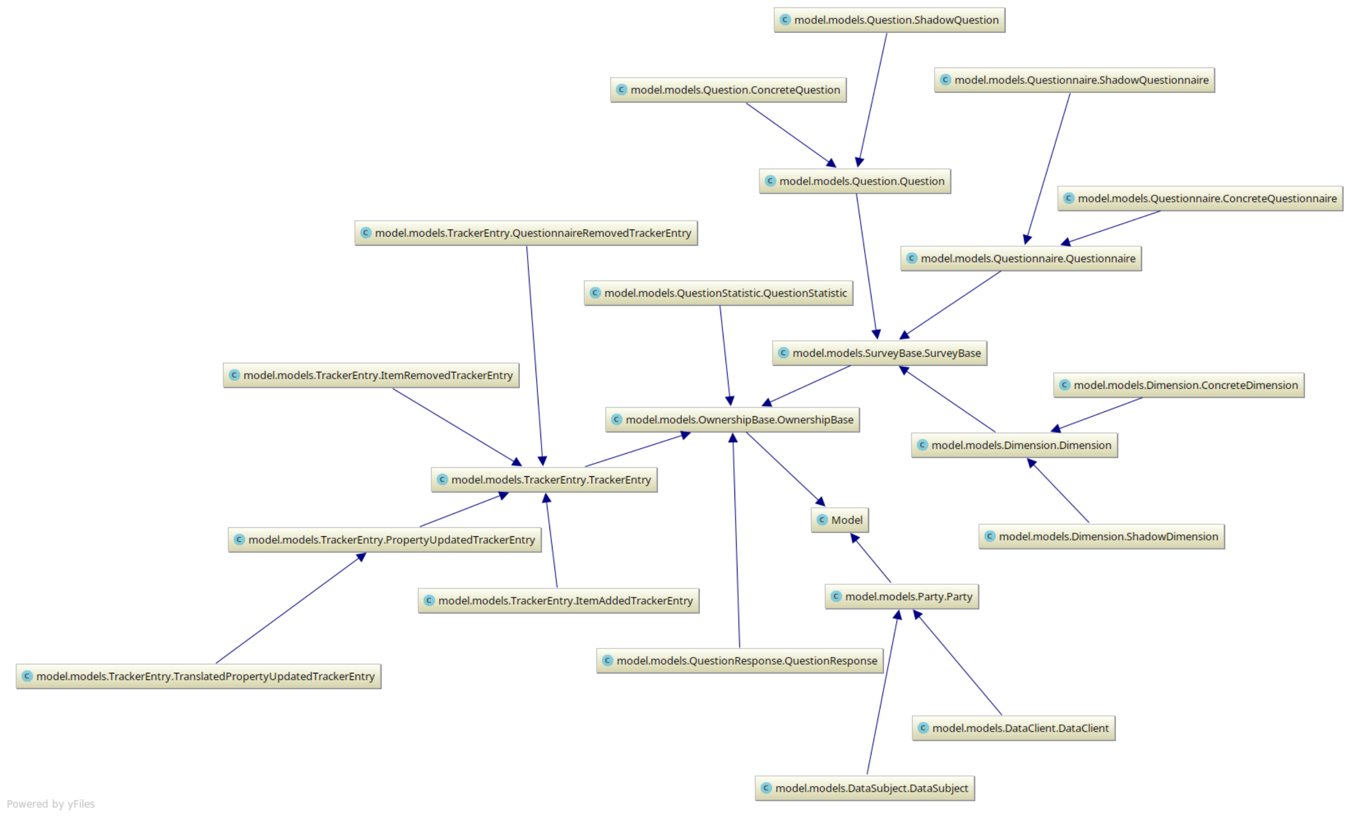

model.models.Question.ShadowQuestion
model.models.Question.ConcreteQuestion
model.models.Questionnaire.ShadowQuestionnaire
model.models.Question.Question
model.models.Questionnaire.ConcreteQuestionnaire
model.models.TrackerEntry.QuestionnaireRemovedTrackerEntry
model.models.Questionnaire.Questionnaire
model.models.QuestionStatistic.QuestionStatistic
model.models.SurveyBase.SurveyBase
model.models.TrackerEntry.ItemRemovedTrackerEntry
model.models.Dimension.ConcreteDimension
model.models.OwnershipBase.OwnershipBase
model.models.Dimension.Dimension
model.models.TrackerEntry.TrackerEntry
Model
model.models.TrackerEntry.PropertyUpdatedTrackerEntry
model.models.Dimension.ShadowDimension
model.models.TrackerEntry.ItemAddedTrackerEntry
model.models.Party.Party
model.models.QuestionResponse.QuestionResponse
model.models.TrackerEntry.TranslatedPropertyUpdatedTrackerEntry
model.models.DataClient.DataClient
model.models.DataSubject.DataSubject
Powered by yFiles

Figure 18: Model class hierarchy

### 5.2.2 Inheritance Mapping via SQLAlchemy

The AQLAlchemy library provides three distrinct mechanisms for mapping class hierarchies to relational database schemas, *joined table inheritance*, *single table inheritance* and *concrete table inheritance*. Joined table inheritance uses a separate table for every class along the hierarchy, with each table only containing data declared by the corresponding class. Attributes inherited from superclasses are associated with subclasses by one-to-one relationships between superclass table and subclass table. When loading an instance of a subclass, a join statement is generated which also loads the appropriate record from the superclass table. Single table inheritance uses a single table for all subclasses of a certain class. Fields not used by a certain subclass are populated with `NULL` values. Concrete table inheritance uses a separate table for every class which contains all data needed to load an instance of the class. This means that inherited attributes will be duplicated in subclass tables. From a perfromance perspective, single or concrete table inheritance outperform joined table

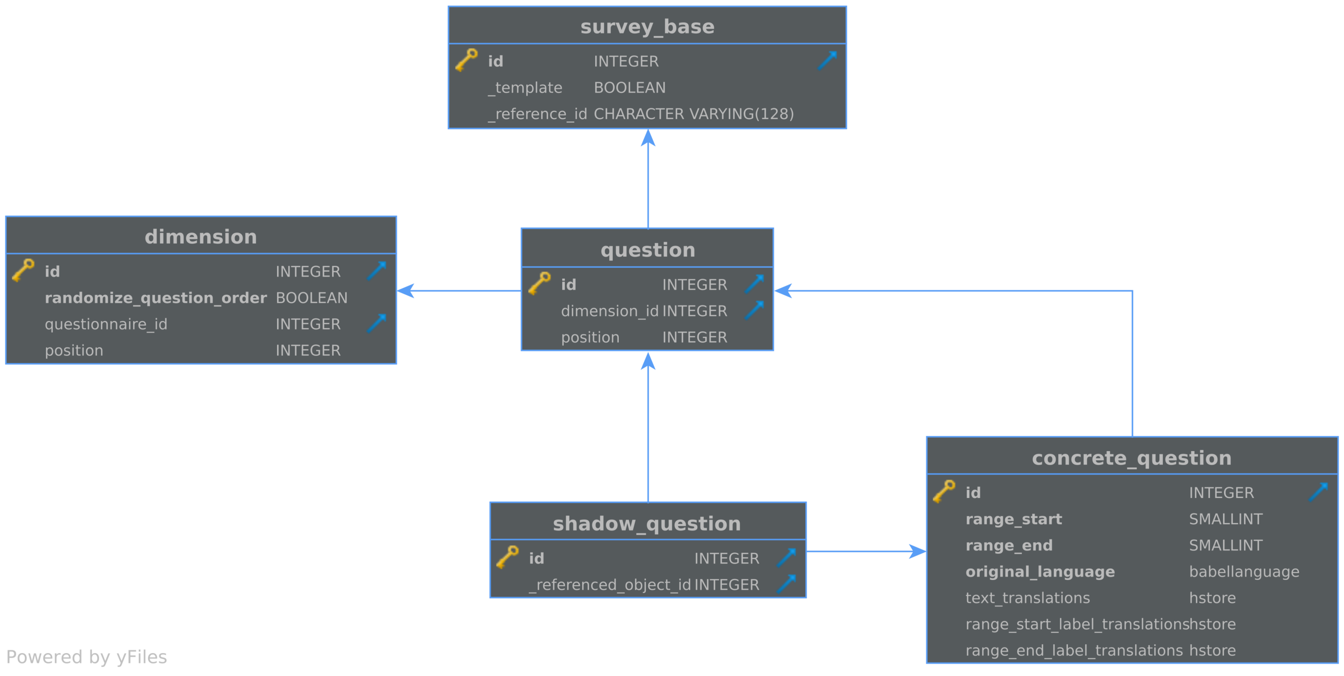


Figure 20: Database schema for a single question including foreign key constraints

inheritance, since no join operation is needed to load an instance. From a storage perspective, joined table inheritance is optimal. Since SQLAlchemy does not support as many operations on concrete table inheritance as for the other types of inheritance mapping, concrete inheritance mapping was not used. While single table inheritance works well for shallow class hierarchies, more complex hierarchies with multiple levels of inheritance produce only sparsely populated database records, as more attributes will be unqiue than shared for most classes. For the survey tool, this approach would only create two tables. For storage optimisation, the joined inheritance approach was used.

### 5.2.3 Template Management - The Template Triad

For each direct `SurveyBase` subclass, `Questionnaire`, `Dimension` `Question`, there are two more subclasses for the concrete and shadow representations of the class. In figure 21 and 20, this *template triad* is depicted. For the sake of brevity, the direct descendents of `SurveyBase` will be called the *super-classes*, it's descendents will be addressed by *concrete* and *shadow* respectively. For template management, each of these classes fulfills a certain role. All actual data is stored in the concrete- and super-classes, while shadow-classes only contain a reference to a concrete. As you can see in figure 21, some data is stored in the super-class instead of the concrete class. The reason behind this design is, that some attributes of shadow classes may still be modified and should not just reflect the state of the associated concrete. An example of this is the access control configuration of the `Questionnaire` class. Data stored in the concrete class can be categorized as *survey content*, while data stored in the super-class can be categorized as *administrative data*. The super-class also acts as an interface for the concrete and shadow classes, specifying all accessors that should be available for it's subclasses. The shadow classes act as proxies, which use Pythons `@property` decorator to provide transparent read access to the referenced concrete's attributes.

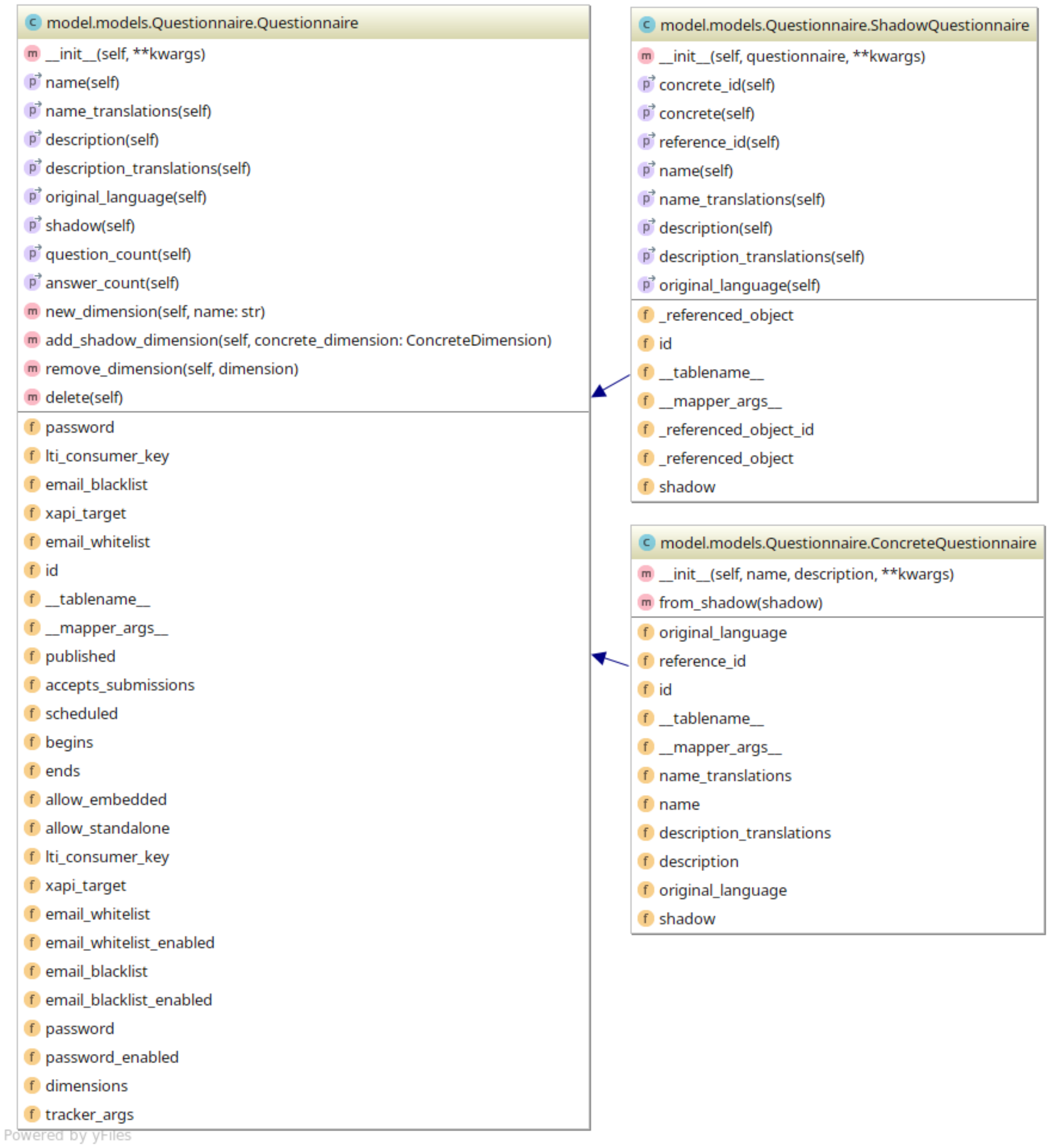


Figure 21: The template triad, exemplarily depicted by the `Questionnaire` data model

```
    @property
    def reference_id(self) -> str:
        return self._referenced_object.reference_id

    @property
    def name(self) -> str:
        return self._referenced_object.name
```

Figure 22: Example of transparent proxying of concrete attributes in shadow classes

## 5.3 API

The API is organized using the `Flask-RESTful` (https://pypi.org/project/Flask-RESTful/) library. For each type of survey item, a canonical endpoint and a set of contextual endpoints exist, which are all mapped to the same handler. Canonical endpoints follow the `http://HOST:PORT/api/TYPE/ID` format, where `TYPE` is the type of survey item. Each survey item is identified by it's unique ID, which may be used for accessing the item via it's endpoints. If a survey item has child elements, the child elements may be accessed by appending `/CHILD_TYPE/` or `/CHILD_TYPE/CHILD_ID` to any of the parent's endpoints. The former returns a list of all children, while the latter accesses a single child. These endpoints are called contextual endpoints.

Each survey item is also associated with at least one JSON schema, which is used for serialisation and deserialisation of the classes instances. The schemas are implemented using the `marshmallow` library. Each survey item's schema also includes an URL pointing to the canonical endpoint of the item, which may be used for fetching the item again in the future.

Before any other request handler is invoked in the backend, information about the request language and user session are parsed. The request locale is determined by three mechanisms, which take precedence over each other in the order they're mentioned here (the latter overrides the former). First, the HTTP headers are inspected for the `Accept-Languages` field and the best match is chosen from the list of available languages. I cookies were sent with the request, they are searched for a `locale` cookie. Lastly, the request parameters are inspected for the `locale` parameter. Session information is communicated using the `Authentication` field of the request headers, follwing the bearer authentication scheme. If a session token is found in the HTTP headers, the token is validated. On success, the associated `Party` object is loaded and injected into the request context.

### 5.3.1 Access Control

Access to API endpoints and actions is handled by two separate mechanisms. The first mechanism uses Python's function decorators to block or grant access to an entire endpoint by checking the current user's role. The second method involves checking the ownerhip status of the accessed resource. For both methods, different convenience methods exist. These methods are listed in table 3

## 5.4 Authentication

### 5.4.1 Data Client Authentication

Authentication for data clients can be performed by providing a valid combination of email and password. These parameters are sent as a request body to the backend. The request can be encrypted by the client using TLS, when a valid certificate is provided for the load balancer. The load balancer will

| Method | Arguments | Description |
|---|---|---|
| `@needs_role` | `[Either Role [Role]]` | Grants access to the wrapped endpoint, if the user holds all roles in the given list. List items may be tuples of roles. In this case, only one of the roles contained in the tuple is sufficient. The expression `[User, (Contributor, Admin)]` would match any user, who has the `User` role as well as either the `Contributor` or the `Admin` role. |
| `@needs_minimum_role` | `Role` | Grants access to the wrapped endpoint, if the role's ID value is lesser or equal to the ID of the given role. Roles can be sorted by their ID, with `Root` having the smallest and `Unprivileged` the largest ID. This method is useful for restricting access to an endpoint to all users with a a certain level of privilege while also allowing all users with a higher level of privilege. |
| `SurveyBase.accessible_by()` | `Party -> bool` | Checks whether a `SurveyBase` may be read by the given party. This methods defaults to true for `Admin` and above. Read access is also granted, if the `SurveyBase` is a template. Otherwise, this method only returns true, if the `Party` owns the `SurveyBase`. |
| `SurveyBase.modifiable_by()` | `Party -> bool` | Checks whether a `SurveyBase` may be modified by the given party. This methods defaults to true for `Admin` and above. Otherwise, this method only returns true, if the `Party` owns the `SurveyBase`. |

Table 3: Convenience methods for enforcing access control restrictions

then decrypt the request and pass it to the backend unencrypted on the virtual network. When creating a new data client, a random salt is generated, using the operating system's random device. The provided password and salt are then hashed using the argon2 libary, which provides cryptographically strong hashing algorithms. The password hash and salt are stored in the database and can be used to validate login attempts, by re-applying the hashing algorithm to the salt and the provided password, and comparing the resulting hash with the stored hash.

### 5.4.2 Session Management

Once a `Party` has sucessfully authenticated, a *session record* is created and published to the `memcached` instance. This session record includes information about the `Party`, a timestamp of the last performed action by the `Party` and a randomly generated *session token*. The session token is handed out to the client via the API and may be used to by the client to identify themselves in subsequent requests. Every time the session token is used in a request, the timestamp stored in the session record is updated to the current time. If the difference between the timestamp and the current time exceeds a certain limit in any request, the token is rejected and the session record is removed from `memcached`. This ensures the eventual removal of unused session records from the cache and protects the user against re-use of their session token if they forgot to log out. Another protection against token stealing is IP pinning. When the user succesfully authenticates, their IP address is included in the session record. If any subsequent request with the associated token is using a different IP address than was used for authenticating, the token will be rejected and the session will be removed from the cache.

## 5.5 Support for xAPI

### 5.5.1 xAPI in the `backend` Service

xAPI statements are created in the backend service using an object-oriented API. Statements are queued locally using the `XApiPublisher` class, which acts as a transaction manager for xAPI statements. When a request context is created by the WSGI middleware, a new transcation is started. Queued statements are only sent to the `xapi-publisher` service, when the transaction is committed at any point during handling of the request. By default, when the request was handled without raising an error, and a rollback was not explicitly executed during the request, the transaction is committed automatically at the end of the request.

### 5.5.2 The `xapi-publisher` Service

## 5.6 LTI Middleware

As mentioned before, recognition of data subjects uses the same token based mechanism that is also used for recognizing data client sessions. Since session management is performed by the backend service, but the frontend bundle is served by the frontend service, performing an LTI launch involves both services. The frontend provides an HTTP endpoint for requesting the LTI launch. The supplied information is then forwarded to the backend service, which validates the supplied combination of consumer key and requested questionnaire. If the LTI launch is valid, the backend service starts a new session for the data subject. A session token is then returned to the frontend service. The frontend service embeds the session token as a global variable into the JavaScript bundle which is the user interface and serves the parameterised bundle to the client. The user interface's bootstrapping process may then check for the presence of the session token and switch to it's embedded version. The embedded version of the user interface uses a different endpoint for submitting responses than the standalone version, as no information about the data subject has to be present in the respose. A sequence diagram for the LTI launch is provided in figure 23.

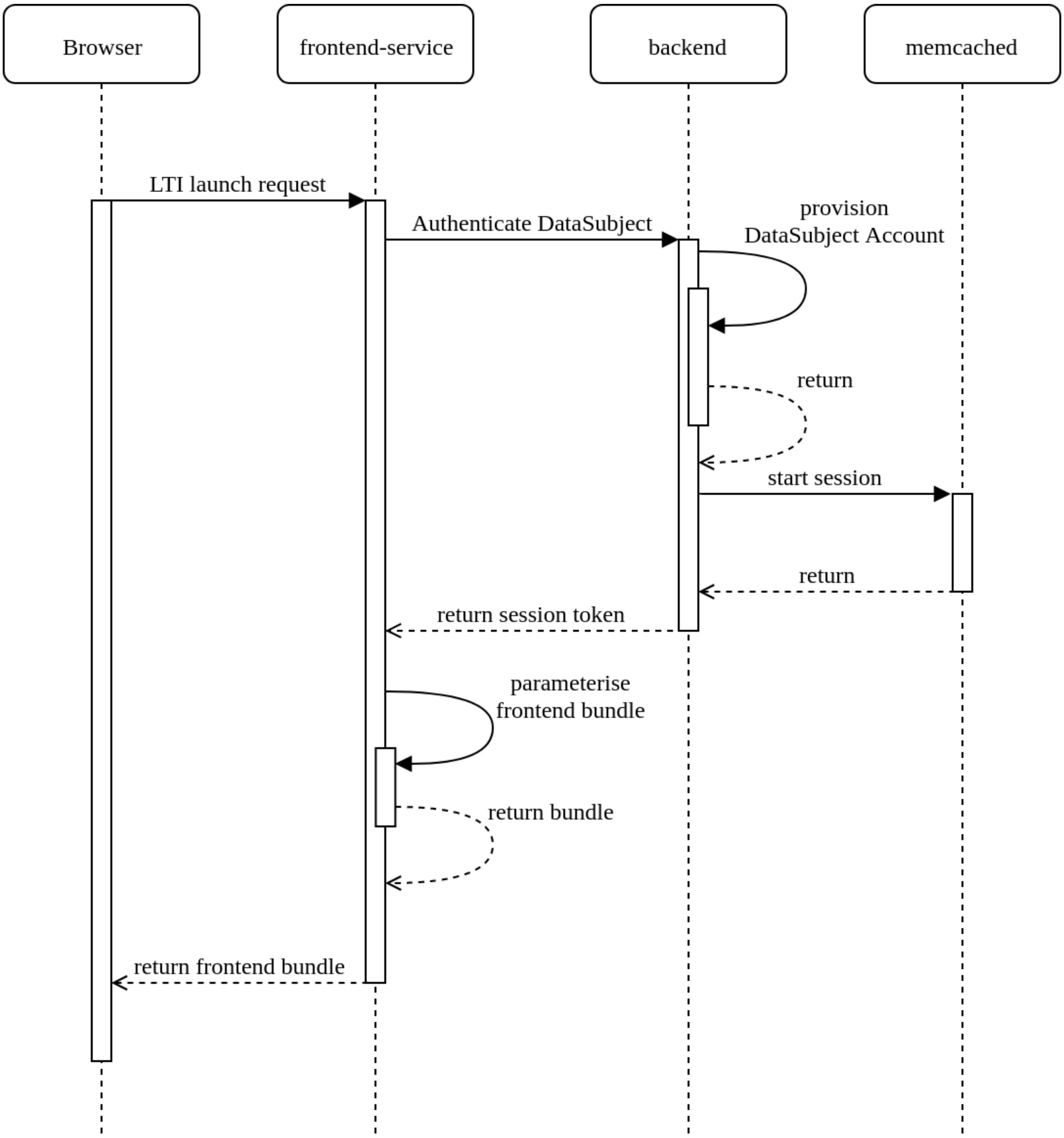


Figure 23: Sequence diagram depicting the LTI launch

# 6 Analysis & Outlook

## 6.1 Known Issues & Caveats

Joined table inheritance is inferior to single table inheritance for this application, even if it looks ugly. Why: performance, no foreign key constraints that mess with you

# 7 Appendix

## 7.1 Database Schemas

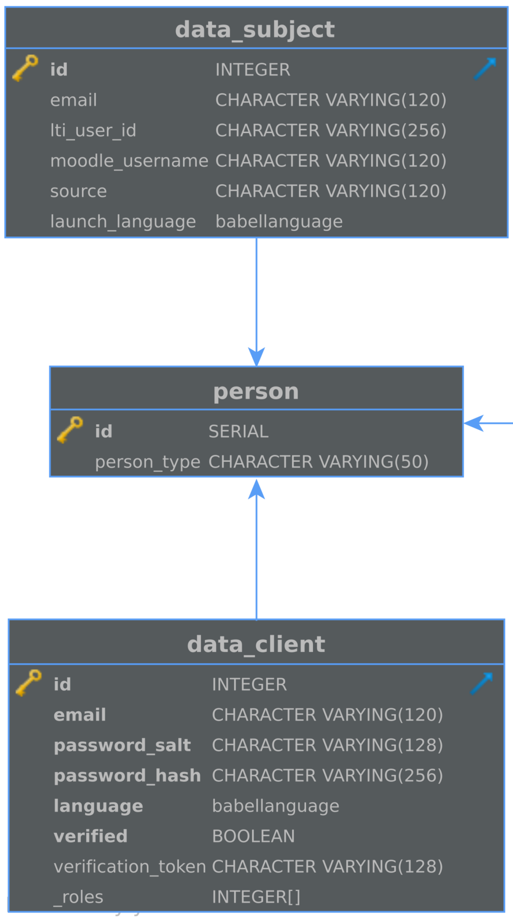

data_subject
id INTEGER
email CHARACTER VARYING(120)
lti_user_id CHARACTER VARYING(256)
moodle_username CHARACTER VARYING(120)
source CHARACTER VARYING(120)
launch_language babellanguage
person
id SERIAL
person_type CHARACTER VARYING(50)
ownership_assoc
person_id INTEGER
ownership_base_id INTEGER
data_client
id INTEGER
email CHARACTER VARYING(120)
password_salt CHARACTER VARYING(128)
password_hash CHARACTER VARYING(256)
language babellanguage
verified BOOLEAN
verification_token CHARACTER VARYING(128)
_roles INTEGER[]
ownership_base
id SERIAL
ownership_base_type CHARACTER VARYING(50)

Figure 24: Database schema for parties & ownership

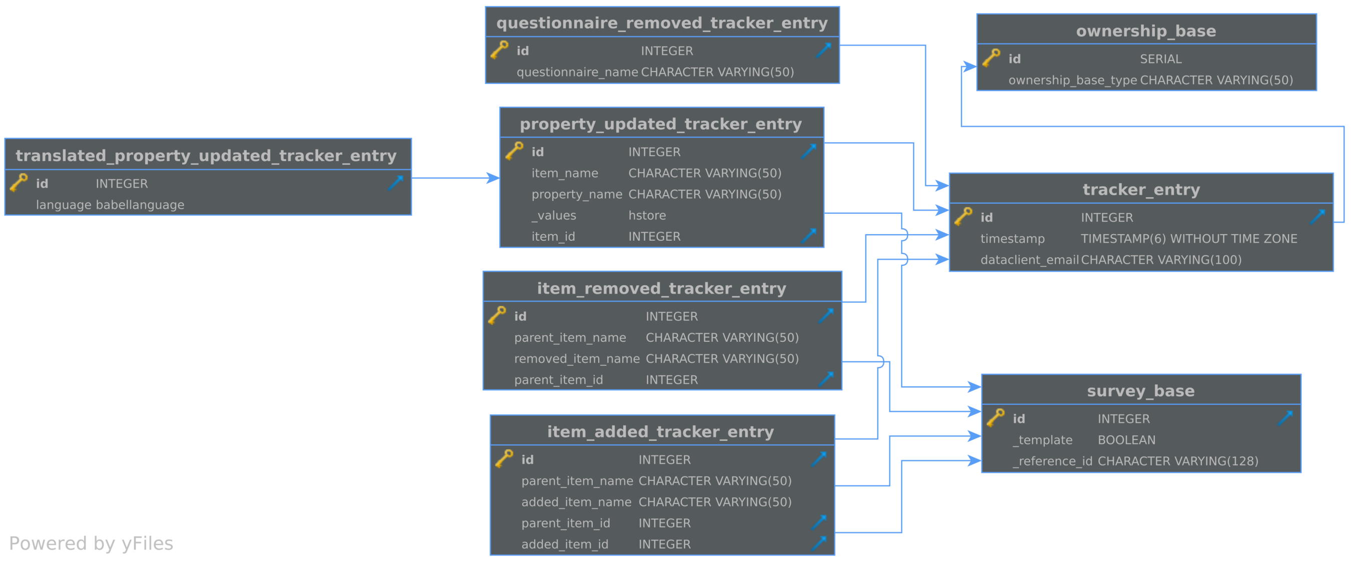

questionnaire_removed_tracker_entry
id INTEGER
questionnaire_name CHARACTER VARYING(50)
ownership_base
id SERIAL
ownership_base_type CHARACTER VARYING(50)
translated_property_updated_tracker_entry
id INTEGER
language babellanguage
property_updated_tracker_entry
id INTEGER
item_name CHARACTER VARYING(50)
property_name CHARACTER VARYING(50)
_values hstore
item_id INTEGER
tracker_entry
id INTEGER
timestamp TIMESTAMP(6) WITHOUT TIME ZONE
dataclient_email CHARACTER VARYING(100)
item_removed_tracker_entry
id INTEGER
parent_item_name CHARACTER VARYING(50)
removed_item_name CHARACTER VARYING(50)
parent_item_id INTEGER
survey_base
id INTEGER
_template BOOLEAN
_reference_id CHARACTER VARYING(128)
item_added_tracker_entry
id INTEGER
parent_item_name CHARACTER VARYING(50)
added_item_name CHARACTER VARYING(50)
parent_item_id INTEGER
added_item_id INTEGER
Powered by yFiles

Figure 25: Database schema for modification tracking

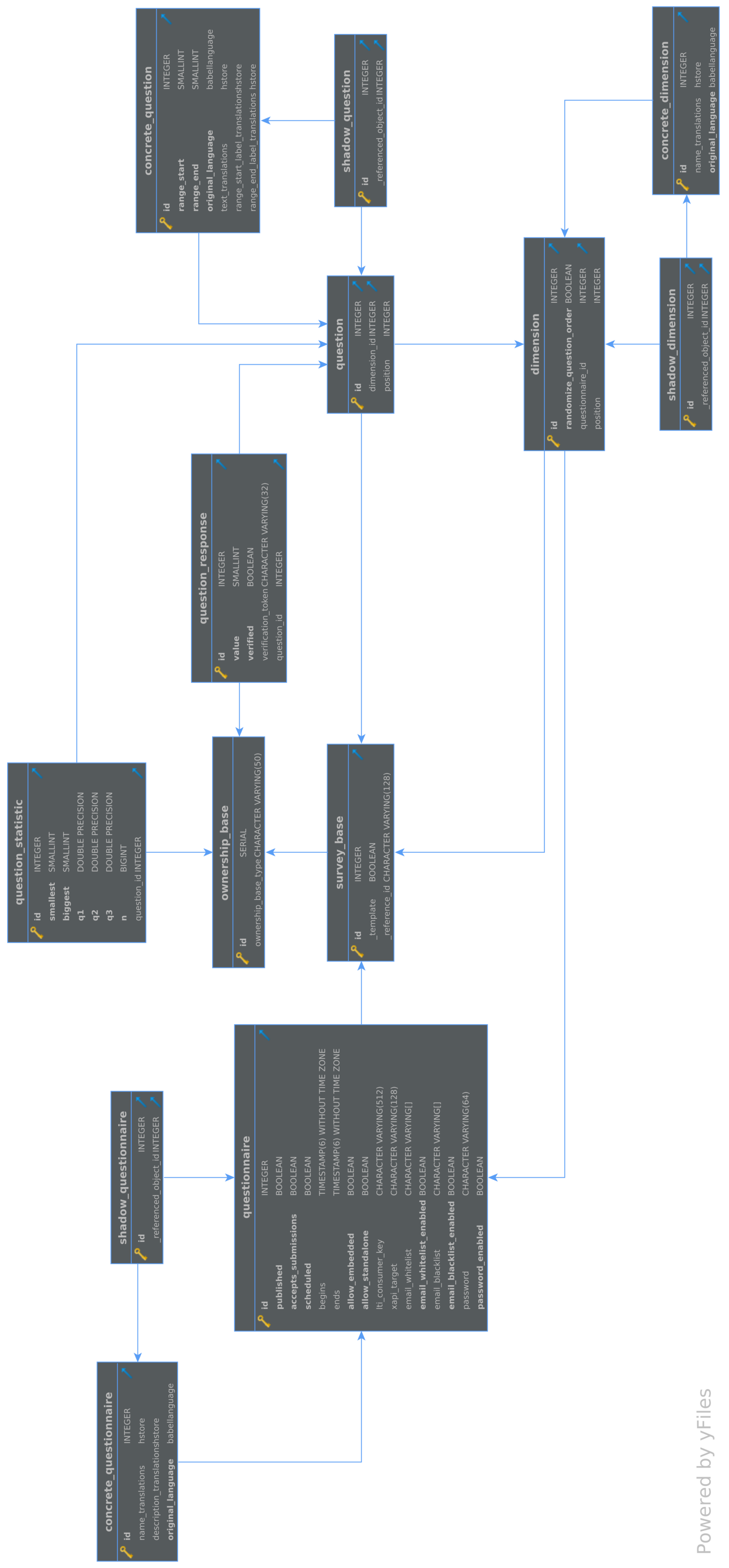


Figure 26: Database schema for survey items

## 7.2 JSON Schemas

TODO: List of all JSON schemas generated by API as JSON

## 7.3 API Endpoints

TODO: List of all API endpoints with description, access restrictions and possible status codes

## 7.4 Abbreviations

**API** Application programming interface; a software interface for interoperability between applications.

**CSV** Comma separated values; a plain-text data exchange format for homogenous data.

**ORM** Object-relational-mapper; a software library for persisting objects in object-oriented languages in a relational database.

**ODM** Object-document-mapper; a software library for persisting objects in object-oriented languages in a document based database.

**xAPI** Experience API, formerly TinCan API; an API specification for exchanging data about learning activities.

**REST** Representational state transfer; a paradigm for API design.

**JSON** Javascript object notation; a plain-text data exchange format.

**ACID** Atomicity, consistency, isolation, durability; a set of properties for database transaction, aiming to guarantee data validity in the event of failure.

**SQL** Structured query language; a language for interfacing with database systems.

**TLA** Trusted learning analytics; a big data storage and analysis infrastructure developed at Prof. H. Drachsler's work group.

**LRS** Learning record store; a big data store for xAPI statements.

**LTI** Learning technologies interoperability; a protocol for integrating third-party services with an LMS.

**LMS** Learning management system; a content management system specifically designed for learning environments.

**CAS** Central authentication service; the user authentication service provided by Goethe University's HRZ.

**HRZ** Hochschulrechenzentrum; Goethe University's data center and IT service provider.

## List of Figures

## List of Tables

## References

[1] The European Commission. Data protection in the EU. `https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en`, 2016. Accessed: October 19, 2018.

[2] The Advanced Distributed Learning Initiative. xAPI specification. `https://github.com/adlnet/xAPI-Spec/releases/tag/xAPI-1.0.3`, 2016. Accessed: October 19, 2018.

[3] Maren Scheffel. The Evaluation Framework for Learning Analytics. `http://hdl.handle.net/1820/8259`, 2017. Accessed: October 19, 2018.

[4] Maren Scheffel. The Evaluation Framework for Learning Analytics, greyscale template. `http://www.laceproject.eu/evaluation-framework-for-la/`, 2017. Accessed: October 19, 2018.