

Database Administration Bundle

Design Report

Pearls of Computer Science

Pearls of Computer Science

You have credentials for this course

To use the database, connect to utwente.nl using these credentials:

Username and DB name:	dab_Pearls_of_Computer_Science_20
Password:	YDwDhw4LJAHxQNqD
Example command for connecting:	<pre>psql -h utwente.nl -U dab_Pearls_of_Computer_Science_20 dab_Pearls_of_Computer_Science_20</pre>

Delete database and release credentials

Reset database

Get dump of database

By:

Floris Breggeman	s2001942
Freek Nijweide	s1857746
Raoul Linssen	s1578197
Denys Flederus	s1855034

Client/Supervisor:

Maurice van Keulen (Client/Supervisor)
Jan Flokstra (Client)

Table of contents

Introduction	4
Stakeholder analysis	4
Requirements	4
Client requirements	4
System requirements	5
Must	5
Should	5
Could	5
Won't	6
Product description	6
Design decisions	7
Back-end	7
Front-end	8
Components and technologies	10
Database	10
Django	10
Production environment	10
Apache2	10
uWSGI[19]	10
Git[28]	11
Shell scripts[29]	11
Postfix	11
Architecture	11
The back-end	12
API calls	12
Permissions	12
Rest_urls.py	12
Models.py	12
Serializers.py	12
Views.py	12
Hash.py	13
Studentdb_functions.py	13
Schemas.py	13
Schemacheck.py	13
Mail.py	13
The front-end	14
NPM (Node Package Manager)	14
Webpack	14
Bootstrap	14
TypeScript	15
Sass	15
Django template engine	16
List of .ts and .html files	16
base.html	16
head.html	16

main.ts	16
interfaces.ts	16
alert.ts	16
navbar.html / .ts	17
admin.html / .ts	17
change_password.html / .ts	17
edit_courses.html / .ts	17
edit_users.html / .ts	17
forgot_password_page.html / .ts	17
login.html / .ts	17
register.html / .ts	17
reset_password.html / .ts	17
student_view.html / .ts	18
Security Analysis	18
Internal Security	18
UT exclusivity	18
Authentication	18
Authorization	19
External security	19
Database	19
Shell	20
Frontend	20
Information security	21
Potential for abuse	21
Conclusion of security analysis	21
Testing	22
Unit tests	22
Field test	22
Team member test	22
“Break it” test	23
Acceptance test	23
Integration tests (front-end)	23
Performance	24
Conclusion	24
Discussion	24
Possible future improvements	26
References	27
Appendix	30
Use case model	30
Glossary:	30
Requirements list:	30
Actor list:	32
Use case diagram:	33
Use case description:	33
Planning & task division (past vs. present)	34
Initial planning:	34
Realized planning:	36

Initial task division	38
Realized task division	38
Webapp overview	39
Database design	41
Roles	41
Syllabus	42
Manual	43
Documentation REST calls	43

Introduction

The University of Twente has several courses that give their students databases to practice on. Currently, these databases are created by several Perl^[1] and SQL^[2] scripts, which result in a large document of database credentials. This document is then printed, and scissors are used to cut out strips with credentials for a single database, which are then handed to students.

With the student sets of previous years, this was manageable, although somewhat cumbersome solution. However, the amount of enrollments in the computer science program has increased explosively in the last few years (from less than 100 students to over 300), and this increase shows no signs of stopping. As such, a solution that involves less manual labor had to be developed.

The general idea is to create an application that can be used by the University of Twente to manage the databases students use during courses. The databases are used for educational purposes during the course, usually a project that requires the use of a database. The current solution involves either Dr. Van Keulen or Ing. Flokstra since the database used at the university can not be managed by every teacher. Besides a couple of scripts, all the steps need to be done by hand. Having the possibility to let students use a database at a later point in time (e.g. a resit) is also cumbersome practice. The goal is to have a web application handle this for the teachers in question and thus also opening this option to other potential teachers without the technical knowledge.

Stakeholder analysis

- The customer: Maurice van Keulen a professor from the UT who specializes in Data Science will be 'buying' our product.

Our product will have to meet his requirements

- Operators: Maurice van Keulen en Jan Flokstra will be operating the product as administrators. But the product should be intuitive and easy to use also for other people. Like teachers and future operators who did not engage with us, the developers, during development like Maurice and Jan did.
- Maintenance and further developers: Our software should be maintainable, meaning well documented and easy to read and understand. It should also be able to have its functionality extended and/or changed by other developers in the future.
- TCS program coordinator and board: The development of the product should pose challenges that require intelligence of the expected university level to ensure educational quality. Our product should also require no or very little funding.
- Government: We will have to respect relevant laws like GDPR.
- UT network (LISA) and server administrators: Our system will host data on the UT network so we will have to make sure no excessive amounts of internet traffic and/or data storage, due to people misusing the database, occurs.
- TCS students: Our system should be safe and reliable. Easy to use and access, and data does not get lost unexpectedly.

Requirements

From the needs of the supplier and customer, a set of requirements can be formed which defines the complete system.

Client requirements

The following are the client requirements after the first and second meeting:

1. The system must be able to automate the handling of databases during the module

2. A student should be able to create a database without help
3. The system should be easy to use for teachers who have no technical background
4. The system should be stateless outside of the main database, meaning that no variable information is stored outside this database
5. System migration, e.g. moving the database to a different host server, should be a relatively easy task to accomplish

The following are the client requirements based on the text provided by the client:

1. Teachers and admins can create, update and delete users
 2. Create a database based on a preset template given by a teacher
 3. Provide a template for students to use
 4. TA's, teachers and admins can access a list of students
 5. Delete a course along with all its databases
 6. Archive all databases of a course
 7. Reset databases to a template
- PostgreSQL^[4] will be used for the database because the existing environment is built upon PostgreSQL
 - Apache2^[5] will be used to host any web server because this is what is used to host existing content on our client's server
 - An admin can create users of any role
 - An admin is able to delete users and databases
 - A teacher is able to specify a database schema for a course. To provide this schema the teacher should not need technical knowledge of the database itself or SQL commands.
 - A student must be able to independently request a database for a course providing a group number
 - The DBMS and the system itself must be able to handle a migration
 - A student can only view their own databases.
 - A TA can view the databases of all students for the courses they are assigned TA to
 - An admin and teacher can view and alter databases of students

System requirements

To further deepen the understanding of the requirements the MoSCoW^[3] system is used. This system basically prioritizes the features needed in the application. There are 4 categories: must, should, could and won't. Traditionally the W in MoSCoW stands for 'would' but the authors think this category is often covered by 'should'. Therefore, we use the category 'won't' instead.

Must

These are the requirements that are absolutely needed. Without it, the application would not suffice to be used by the client in any way.

- Every role (you can find an overview of the roles in the [Actor List](#) p.33) must be able to interact through a web application

Should

These features are non-essential but are highly desired.

- A student can reset the database to the schema provided by the teacher
- Logging events (e.g. database wipes) such that you can see what happened if e.g. a student loses their data

Could

These features could be implemented if time allows it. They will not have the main focus during the core part of development. Design decisions will be made with these features in mind to allow their development in the future.

- Docker^[6] integration: users can create docker instances just like databases
- Scalability: being able to host databases on multiple servers if the server demand is high

Won't

These requirements are essential to state the boundaries of the project. With these, the client know will know what the project is not used for. This means we will not implement these features. But the given features might be able to be added to the system in the future.

- The product will not be used for automating the creation of virtual environments and machines, such as the use of Docker. While this was looked into, we expected it would create too many uncertainties and problems during development. Without technologies like Docker, we only have to focus on delivering working front-end files (HTML, CSS, JS) and editing databases through SQL scripts. Adding virtual machines would increase the scope of the project by too much for us to deliver a focused complete product.
- There will be no capabilities to access, alter or change anything inside the databases (aside from a reset button, and the possibility to transfer a schema from another database). The purpose of the application is to generate credentials for databases. Building functionality that allows the user to edit the tables in the database would make the scope of the application much larger and does not improve the user experience by much. The user probably has a favorite application for editing databases already.
- The product is designed for at most a large class (around 100 people) creating databases at the same time. Large groups (e.g. >1000) are not taken into account as this could require a different architecture.

Product description

Here we want to familiarize you with our product by asking and answering some questions. Known as the Five W's^[7].

Who, What, Where, When and Why.

Who is the product for?

Teachers and administrators of database-related courses of UT and the students that follow these courses.

What are the product's basic details?

The product consists of a WebApp built from:

- Django^[8] (a web framework for Python^[python])
- TypeScript^[10] (Javascript^[11]-like language with types)
- Sass^[12] (a language that extends CSS^[13])
- Bootstrap^[14] (a CSS framework)
- HTML^[15] pages using Django template engine^[16]
- PostgreSQL^[4]
- NPM^[17] and Webpack^[18]
- uWSGI^[19] and Apache2^[5]

The front-end communicates with the data through API calls on the back-end which gives it controlled access to the main database. Outside of this database, the system is completely stateless. Meaning that all non-static data resides inside the database making the system very portable.

Where would someone use this product?

This product can be run on any device that has access to a browser. Phones, PCs, laptops, tablets, etc. The user will need access to a UTwente email to create an account.

When would someone use this product?

When students in a course need databases for their assignments and projects. Their teacher can use this product to provide them with a database. Students can also be assigned TA to a course, giving them insight on all other students' databases in that course.

Why is this product useful or better than its alternatives?

As described in the [introduction](#) this product was designed to replace the mostly manual

task of executing scripts to create databases and having to personally hand over the credentials to these databases to the students. After this the teacher would have little control over the databases and who had access to them. Often leading to time-consuming conversations over email when something went wrong. This system makes it easy to create and manage databases for both teachers and students.

Design decisions

During the project numerous decisions needed to be made to get to a final product, ranging from core critical to less important decisions. Below is a list of the most important design decisions made during the project. These design decisions are only related to the requirements that we had already decided to implement. For an explanation of why certain requirements were given priority over others, or some requirements were discarded, see the Requirements section)

- The system uses PostgreSQL. Not using this would cause incompatibility with previous software used by the customer. Thus, it was not an option for us to use other SQL dialects.
- We chose to write a web app, not a native application. This is because the application needs to be used by teachers, admins, and students who might be using all sorts of devices. To ensure users can use DAB from whatever platform they might be using, developing a web app was the most logical choice. Furthermore, because we would be dealing with secure information and lots of SQL statements, having a back-end + front-end architecture seemed like the most logical choice. Back-end frameworks/languages such as PHP have good support for writing and executing SQL. This separation between back-end and front-end also keeps the end-user from

accessing secure data, such as the master database credentials.

Back-end

- The client requested an application written in Python. By this time, we already knew that we would develop a web application, which still left the choice between Django and Flask^[20]. Django was chosen because at the time it was thought the greater functionality would lead to faster development. In retrospect, it has turned out quite a lot of the Django built-in functionality was unsuitable for our purposes and we had to develop our own solutions anyway; it might, therefore, have been better to have chosen Flask instead, although the difference would probably not have been significant.
- A student can only have one database per course. This fact has changed quite often during development, with students being allowed multiple databases per course at several points; there was even the idea of only one database being allowed per group of students for a while. However, one database per course per student was eventually settled upon, because it was a good way to prevent abuse, much simpler to implement, and easier for administrators to manage.
- DAB is a RESTful^[21] application (that is, the frontend uses Javascript to load in content, instead of the content being present when the page is loaded). This results in a better separation of front- and back-end and means that the user does not have to reload the entire page upon performing an action. There are two places where the principle is breached: login and logout. This decision was made because these actions edit the session cookie, and doing so via javascript has been known to cause issues; also, the entire page has to be reloaded upon login and logout anyways, so doing them RESTfully has no advantage apart from ideological consistency.

- The application is mostly stateless, as requested by the customer. As mentioned previously, results of API calls are independent of any previous API calls, (session notwithstanding). Furthermore, the back-end only contains settings for connecting to the database server, a boolean for debug mode, the URL prefix (whether the application will be hosted under /dab/ or under another path), and some strings to display to the user. All these settings are saved in the file `secret.py`, which the user has to create themselves. We provide the user with a `secret.py.example`, with some example settings. This was quite easy to implement because it aligned with our own development strategies.
- Shell scripts are used to set up the application, and to update to the latest version. This runs on any Unix-like operating system (e.g. macOS, Linux). This is by far the easiest way to ensure a few system commands are run in sequence. In our case, it is used to download the latest versions of dependencies and compile the front-end files. The only operating system not covered by this is Windows. Our customer will not be using a Windows environment to host this server, thus using shell scripts for installation and updates seemed like a reasonable option. Using these shell scripts allowed us to develop a git-based rolling release model, where we can ship changes to the client when they are already using the software.
- The splitting of roles was decided to ensure maximum security. TAs do not have a separate role; they are students that have edit permissions on a course and its associated databases, as a TA in one course could be a student in another. Teachers should not have access to the entire system and be able to edit all of its users; this privilege is only for the administrator(s). This division also makes the build-up of the application quite simple: the "main" roles are admins and students; teachers can access all things that an admin can access, except for the editing of users, and two other

small API calls. TAs can do everything a student can, and a tiny bit more.

Front-end

- NPM^[17], the Node Package Manager, is used to manage non-Python dependencies in the project. This allows us to install new packages and features into the `node_modules` folder, in the same way, one would install an application on Linux. The alternatives to this are downloading the files manually (which prevents us from receiving updates) or using CDNs^[22] (which is the equivalent of letting the user download Bootstrap from another server). While setting up NPM is quite easy, properly using the acquired packages in a front-end is very challenging and requires more advanced techniques (see: Webpack). However, the performance benefits and customization options (see: Sass) this brings compared to using CDNs are worth the effort.
- Webpack^[18] is software that bundles modules together. We use this to make sure all the TypeScript, Sass, and other languages are written, are compiled, and that any imports they use are included. Setting this up is a lot of work, but the smooth update experience for the customer (running one single shell script that fetches the newest dependencies and compiles them into the correct folders) is worth it. The alternatives to Webpack are Gulp and Browserify. We looked into these, but they did not offer the extensive compiling, linking and chunk-splitting functionality that we require.
- Bootstrap^[14] is used as the basis for styling on most elements. Using Bootstrap allows us to easily put content on intended parts of the page, and achieve Responsive Design^[23] without putting large amounts of time into developing CSS for styling. One of the largest reasons this approach was chosen is that the front-end developers did not have much experience with styling using CSS. Their previous experiences with web development made them quite sure that an approach that would not use a framework like Bootstrap

would cause a lot of time to be spent on styling. They were, however, quite confident in their ability to quickly achieve a basic design using Bootstrap due to previous experiences with it. The reason for choosing Bootstrap instead of another framework is due to its popularity, and the authors already being experienced with it.

- We decided to use Sass instead of pure CSS for styling. Sass is a language that adds variables and various other features to CSS. It was chosen because it allows us to import variables from other packages (such as Bootstrap), and override those variables. This allows us to be more consistent in our styling choices.
- TypeScript^[10] is a language that adds static typing to JavaScript. Using JavaScript (JS) is unavoidable in front-end development, as it is the only language that browsers truly support, next to HTML and CSS. However, its weak typing often leads to problematic situations where a bug is caused by an implicit type conversion (from a string to an integer, for example) which causes unexpected behavior (an empty string becomes zero when cast to an integer, which is probably not what the developer intended). TypeScript fixes this by adding types to all variables and functions. The authors hoped that their biggest problems encountered previously during front-end development would be solved by using TypeScript. In the end, this decision was worth it, as it led to very modular, clean code. Furthermore, static typing is quite educational, as it forces one to learn the small differences between types such as an `HTMLButtonElement` and an `HTMLInputElement`, preventing mistakes (even when writing regular JavaScript).
- Email verification is used for registering new users, instead of using the existing University of Twente OAuth^[24] login system. We would have preferred using the UT OAuth because it leads to quite a smooth user experience. However, we were quite certain that implementing this in our own application would cost a lot of time. This is

because none of us had any experience with this system, and we were already short on time because of a lack of front-end development experience overall. We chose to use email verification (only allowing `utwente.nl` addresses), as one of the authors was already experienced with how to implement this on a back-end. This saved us quite a lot of time.

- The splitting up of functionality across 3 pages (*edit courses*, *edit users*, *student view*) was decided on after much deliberation. It keeps all functionality (the list of objects to access in the left pane, and the information on the object in the right pane) similar across pages. This makes it easy to reuse content and keeps the functionality clear for the user. Furthermore, it makes it easy to separate functionality for different roles. Admins can see all three pages, teachers do not get to see the *edit users* page (and only see their own courses on the *edit courses* page), and students only get to see the *student view* page. Students that are TA for one or more courses do get to access the *edit courses* page, but they can only access the courses they are TA for, they cannot delete or get dumps of courses, nor can they add new courses.
- No framework such as React^[25] or Angular^[26] was used for the front-end. These technologies are highly popular these days and improve the user experience with features such as dynamically changing page content instead of reloading the page. However, because of the authors' lack of front-end development experience overall, it was decided that no such framework would be used. This way, no time would have to be spent learning to work with these frameworks, next to the time needed to learn proper front-end development.

Components and technologies

This section will outline the different components of the system and the technologies that underlie them.

Database

Central to DAB is, of course, the Database server. This is an instance of the open-source PostgreSQL^[4] database system, and actually runs independently from DAB; however, DAB both uses a database and manages databases.

The data that DAB needs to operate, such as the data on users and courses, is stored in the so-called master database. This database is owned by the superuser that DAB operates with. DAB interacts with this database through the Django ORM^[27] (discussed below). Apart from being accessed by a superuser, this database is no different than how one would normally set up a database for use by a program.

The databases for the students are different; while DAB can create them, delete them, and even get and set their contents, they are in principle completely independent entities, which are managed by DAB, but not necessarily part of DAB.

Django

To function as a web application, DAB uses Django. Django is a Python web framework, which provides many useful tools and libraries allowing the programmer to quickly set up a website.

DAB uses two of these: the standard web part and the ORM.

The web part is responsible for handling the HTTP exchange. We have written functions that perform certain actions, specified that they should be bound to certain URI's and Django will make sure that they are.

The Object-Relational Manager, or ORM^[27], is an abstraction layer between the program and the database, which represents database entries as if they were Python objects. While the more advanced database tasks, such as the creation of student databases, still require raw SQL, all interaction between DAB and the master database is done via this ORM. This makes the database more intuitive to work with for the programmers, and allows certain checks to be performed at a lower level; e.g. the verification of email addresses is done in the ORM, which means that the program simply can't store invalid emails in the database. This could also be done at a higher level, but overriding the ORM lowers the change for programmer error as it is only done in one location, which is good for both reliability and security.

Production environment

There are several technologies that are not necessarily part of the product, but that are required to run it in a production environment.

Apache2

While Django does have a web server, this is only intended for development purposes. In production, a proper web server should be used, which can serve static files itself, and pass the information that Django needs to process to a socket. On the Bronto production server that DAB will eventually run on, this web server is Apache2; in the production setup used for testing, it was Nginx.

To properly use DAB, the production server must be set up to serve static files in the projects static directory if the request is under /static, and pass it to Django if the request is under /dab.

uWSGI^[19]

As mentioned previously, the Django webserver is intended for development purposes only. In production, Django will

not be run under Python directly, but instead, use the uWSGI runtime. To do so, one configures a file telling uWSGI how to run DAB, and uWSGI will form a much more stable and efficient runtime than the standard Django webserver. Furthermore, uWSGI will create a UNIX socket at a specified location, which the web server can pass requests to as described in the previous section.

Git^[28]

As any development process should, DAB was developed with version control, specifically git. Git allows programmers to work on the program simultaneously on different devices without conflict, allows one developer's unfinished features to be separated from the other versions, and allows for the reversal of incorrect changes. Furthermore, the development repository is mirrored to GitHub, where there is a special branch for features that are ready for the actual production environment. This means that the system administrator of the said environment (i.e. Ing. Flokstra) can update the program using two simple shell commands.

Shell scripts^[29]

A shell script is a simple file in UNIX-based operating systems often used to automatically install the software. It is a file that contains shell commands, and executing a shell script is practically equivalent to pasting all of these commands in the user's shell. There are several of these shell scripts in the DAB project, all of which do slightly different things:

- `Install_development.sh`: installs the python and typescript dependencies for use in a development environment.
- `Install_production.sh`: installs the python and typescript dependencies for use in a production environment, compiles the typescript to javascript for production use, and runs `migrate.sh`

- `Migrate.sh`: puts the SQL schema of the master database into the database specified in the settings.
- `Promote.sh`: asks for a user email and promotes this user to administrator. Useful for the first-time setup, or in case of credential loss.
- `Start_development.sh`: starts the Django development server, and watches the TypeScript files for changes in order to always have a compiled version up to date.
- `Update_production.sh`: pulls the latest files from the git repository, makes sure the dependencies are still up to date, and recompiles the TypeScript. After this is done, one simply needs to restart uWSGI to have an up-to-date version of the software.

Postfix

There are several features of DAB that require the sending of an email, most notably email verification. Python can send email natively by connecting to the appropriate SMTP^[30] server; however, because email is based on trust, this is a bad idea. A better idea would be to hand the email to `smtp.utwente.nl`, which will then forward it to the appropriate address, but this also has downsides: if `smtp.utwente.nl` is experiencing issues the mail might not be delivered, or certain parts of the program might become very slow. As such, the appropriate solution is to have the host server run a simple SMTP server, i.e. Postfix^[31], which will take in email messages from DAB and forward them to `smtp.utwente.nl`. This will ensure that the email is sent securely, asynchronously, and buffered. Furthermore, it will ensure that the appropriate headers are set, decreasing the emails spam score, and it will create a proper log with all sent mail.

Architecture

DAB uses a RESTful architecture: there is a back-end, which provides an API to the

front-end, which is presented to the user in a web browser.

The back-end

The back-end consists of several Python modules and using the Django infrastructure they are responsible for providing the API to the front-end.

API calls

During the project, a very strict and clear REST^[21] API call system is created. This section is meant as a short introduction to the philosophy behind the system. A more extensive version of every call can be found in the appendix *REST call documentation*.

The DAB application uses 4 tables to function properly. A REST call is either directly interfacing with one of these tables or performs another action that will have a distinct REST call name.

In the case of the 4 tables, the call will start with the domain name followed by /rest/ to indicate a REST call followed by the name of the table. So in the case of the course table, this would result in: <domain>/rest/courses/. The CRUD^[32] (Create, Read, Update, Delete) functions are used to indicate which action will be taken. Because HTTP has functions for each of these, this is highly efficient.

Permissions

These API calls are protected by a permission system for every role as mentioned in Roles (the different roles are listed in the appendix under "roles"). Whether a user is allowed to use a certain API call is based on 2 questions:

Do I have the correct role level or above?
Am I the owner of this item?

For example, a teacher can change the info about one of his own courses, but cannot change the name of a course which does not belong to him.

The permissions that go along with every call is included in the appendix under *REST call documentation*.

The back-end consists of different files with a specific task. Below is a list of these files and the reason for this file. A significantly larger and more detailed documentation, generated from the comments in the source code, can be found in the file docs/backend-html/index.html, in the source code^[39]

Rest_urls.py

This file binds the URLs under /rest to the corresponding functions in views.py, as well as extracting parameters out of the URL and passing them to the front-end where necessary.

Models.py

This file is responsible for telling Django what our database schema looks like in order to ensure the functionality of the ORM^[27]. It's also where ORM level verification happens; e.g. the verification of emails happens in this file, which ensures the program can't save a user with an invalid email. Finally, the bitmask conversion of database passwords (see security analysis) also happens in models.py.

Serializers.py

This file constructs Django serializers, which can automatically convert between Django ORM objects, which the database understands, and JSON^[33] objects, which the front-end understands.

Views.py

Most of the functionality comes from views.py, and it is by far the largest code file

in the project. It contains all of the functions that are responsible for actually serving the API calls, as well as a few helper functions. The functions in `views.py` can be categorized into four categories, roughly in order of appearance in the file.

The first are helper functions for the Authorisation system. The most trivial is a function that checks if the user has at least a certain role, but there are also functions which check if the user is allowed to access an object, because they own it or because they are TA over a certain database. Also notable in this category are the authorization decorators, which make it much easier to build authorization into the other functions, as the programmer only has to put the decorator above their function to set basic permissions.

The second category is the serializer functions, which provide the bulk of the API. Each CRUD operation is handled by a single function for all models, which uses the appropriate serializer to perform the operation. What the relevant model and serializer are handled in `rest_urls.py`. This category also includes slightly more specific calls, such as the call that gets all objects owned by the user, which is essentially the same as the regular GET, but with one extra search parameter.

The third category is the functions for more specific rest calls. A call that has to do with a schema, for example, does not fit within the traditional REST model and will have a separate function. There are also calls to get information that is already available through regular means, but would require many API calls to assemble; for example there API call that returns all databases that are part of a course that a certain teacher owns. Furthermore, this category also includes calls that are responsible for rare administrative functions, such as generating an archive for a course.

Finally, there are the functions that handle authentication: this includes the login and logout functions, which parse the POST

forms, but also the functions to verify a user email and reset a password.

Hash.py

A helper module to `views.py` responsible for handling encryption-related tasks, such as the creation and verification of password hashes, as well as the creation of random values, such as verification tokens and database passwords.

Studentdb_functions.py

A helper module that provides functions to create, modify, and delete student databases. The module does not interact with the master database, but only with the student database in question

Schemas.py

A helper module that is responsible for importing and exporting the contents of student databases. Like `studentdb_functions.py`, it does not interact with the master database, but only with the student databases. It contains functions to dump a database, write a schema to a database, and generate a dump of a course.

Schemacheck.py

It contains a single function, which is responsible for verifying that a schema can be imported into a student database without errors. It does this by generating a temporary student database, attempting to import the schema into this new database, and checking if this results in errors. Originally, this function was in `schemas.py`, but this generated a circular dependency with `studentdb_functions.py`

Mail.py

A helper module is responsible for sending emails. It contains three different email messages and will send a message to the given email address of the user. If the application is in development mode, it will not send an email, but instead print the

message to the standard output, as most development machines do not have a mail server.

The front-end

The front-end is the part of the application that the user sees. It communicates with the backend via API calls. It consists of various components interacting in many ways.

NPM (Node Package Manager)

NPM^[17] is used to manage dependencies, as mentioned before. The use of NPM indirectly makes our project a Node.js project. Node.js is a JavaScript runtime environment that is used to run JavaScript outside of a browser (for back-end or native applications). All Node applications come with a *package.json* file that contains information such as the project name, description, version number, and most importantly: its dependencies. These dependencies are specified with version numbers included so that a project will never stop working because the newest version of a dependency breaks or deprecates an important feature.

The *package.json* file also allows us to define npm commands, which we use often throughout the project. "npm run production", for example, generates documentation, and runs webpack in production mode (see: the Webpack subsection after this). By default, "npm install" downloads all dependencies specified in *package.json* and extracts them into the *node_modules* folder. Because NPM does not always download all dependencies recursively, we have had to include some dependencies which are only used by some small parts of other dependencies.

Webpack

Webpack^[18] is used to bundle parts of front-end projects together. It is quite a large application with many options and

even more plugins. These plugins are downloaded via NPM, while the options always have to be specified in *webpack.config.js*. This file can become quite large, and setting it up correctly can take quite long for inexperienced users (which was the case for us). We use JavaScript within this options file to look at the arguments that were used to run webpack, and the contents of the *src/frontend* folder, and adjusts the settings according to those parameters.

In our case, we use Webpack to call the TypeScript compiler (which changes the TypeScript into JavaScript by removing the types and converts the code into an older version of JavaScript to be compatible with browsers that do not support ES6 yet). This is done recursively on all dependencies until the output files have self-contained JavaScript. Then, if webpack was run with the "production" argument, the JavaScript is minimized (making variable and function names shorter, removing comments, etc.), dramatically reducing page load times. Furthermore, in production mode, common dependencies are split off into "chunk" JavaScript files. Because most modern browsers use caching, these files (which contain code for the navbar, Bootstrap dropdowns, and other things used on many pages) only have to be downloaded the first time the user visits the application. This increases the performance of the application, changing load times from half a second or more to less than 100ms when hosting the server locally.

The same principle is applied to the Sass files: they are compiled into CSS files, minimized (although this cannot be done as extensively as with JavaScript files because class names and such cannot be changed without breaking the code), and split into chunks.

Bootstrap

Bootstrap^[14] is used to achieve basic styling on webpages without writing much CSS ourselves. It is actually quite a simple module, consisting of lots of styling for predefined HTML classes, and some JavaScript for basic animations. After adding classes such as "btn btn-danger" to an HTML button element with Bootstrap, the button suddenly looks quite modern, with white text in a red background that shows the button should clearly only be used with caution.

We use some other open-source packages for the same purpose, such as "sweetalert2"^[34] (which shows Confirm / Cancel alerts to the user on attempting an important action), "bootstrap-cookie-alert"^[35] (for showing that we use cookies), and "autosize"^[36] (which resizes HTML Text Areas to fit the content).

The use of Bootstrap has many benefits. Firstly, we wrote very little CSS or Sass during the project, as the Bootstrap classes were specific enough for us most of the time. This means that the styling of the page is already quite clear from the HTML. Secondly, if the customer would want it, we could write custom CSS to bring the current styling and color scheme more in line with other Data Science projects, while keeping the overall look and feel of the app mostly unchanged.

Lastly, the application is completely mobile friendly because Bootstrap allowed us to easily implement responsive design^[23]. Any views that have two panes or columns will have them placed underneath each other, with the text made slightly larger, and other small changes made to improve the user experience. The navbar is also changed to behave as expected on a mobile website. The experience on smaller screens such as tablets is similar to the desktop experience,

but would not have been as smooth without the user of Bootstrap.

TypeScript

TypeScript^[10] is a language developed by Microsoft which adds types to JavaScript. As explained in the *design decisions* section, this has many benefits that improve code quality. There are no more unexpected type coercions, functions must have well-defined parameter types and return types, etc.

Explicitly defining all types (such as *number*, *string*, or *HTMLCollectionOf<HTMLDivElement>*)

prevents any type of errors from going undetected. If any typing mistakes are detected in the code, the compiler will notify the developer and refuse to compile the code. The compiler can be instructed on how strict it should be. This is, among other settings, defined in the file *tsconfig.json*. We use the strictest settings available (adapted from *gts*^[37], the TypeScript code style used at Google) to ensure the reliability of our code.

These strict settings include small changes, such as not allowing unreachable code (such as after a return statement) to exist at any point, and not allowing functions without explicit return types. We also used TSLint^[tslint], a tool to highlight possible code style improvements. Again, the settings from *gts*^[37] were used, which do not allow the user to use `==` for equality checks, because this might implicitly change `"2" == 3` into `2 == 3`. Instead, using the triple equals operator (`"2" === 3`) is encouraged, which will never change the types of variables used. Also, the use of *const* to declare variables is always used whenever the variable is never changed. TSLint highlights all instances where this is possible.

We only use one import within TypeScript not related to Bootstrap: *Axios*^[38], a framework for reliable execution of XMLHttpRequests using JavaScript promises. This makes the code more easily readable; *axios.get("URL")* is a lot shorter

than creating new XMLHttpRequest objects and setting multiple fields of these objects to do the same thing each time. Furthermore, the use of Axios saved us days of refactoring code when we discovered that the customer wants to host the project under /dab/ instead of on the root path of the server. We only required one change in the axios settings to have all links on the front-end point to the correct destination again.

Sass

Sass^[12] is a language that extends the functionality of and compiles to, CSS. We only use one instance of Sass: the main.sass file. This file imports the bootstrap styling, and styling for the cookie alert and Sweetalert2. We then set the color of links and buttons to be a consistent cyan color throughout the website. Some colors of nav-link elements (on the left of the page in all sections of the website) to be blue when clicked, green when owned by the user, grey when inactive, etc.

The file also fixes styling on tables to be consistent and defines styling so that checkboxes look more like modern, toggleable buttons.

Django template engine

We use the Django template engine^[16] to render our webpages. This allows us to reuse large parts of webpages without having to write them more than once, by importing the contents of other .html files or extending other files. The template engine also has the capability to pass variables (via a *context*) to the template. These variables can then be displayed directly on the page. They can also be used in if-statements or loops, to only display certain content if a boolean is set, or to display the contents of an array in a table.

List of .ts and .html files

We will first list 2 HTML files and 4 TypeScript files that do not map directly to a page in the application, after which we will

describe the pages of the application. For more detailed explanations on the code of these files, please look at the extensive HTML files generated from our front-end documentation, in the docs/frontend folder of the source code^[39]

base.html

All webpages in the DAB extend a file called base.html, which defines which files to import. It gets passed some variables from the backend, such as a list of JavaScript chunks and CSS chunks to import (see: the subsection on Webpack), and has a default page title.

head.html

This file is where various things that should be included in the <head> tags of the base.html are defined. This file is now only a few lines long, but it used to be quite large when we still used CDNs^[22] during early development.

main.ts

main.ts includes some imports and changes the default Axios settings to include the URL prefix (in the case of the customer, /dab/). It is used to make sure that the various imports, such as Bootstrap, are actually imported and compiled.

interfaces.ts

This file contains various interfaces and enums that represent objects in the data model such as courses, or users. These interfaces directly match with the format sent in the body of API calls and received in responses.

alert.ts

This file contains all the code for error handling seen on the front-end. It contains methods for generating the HTML for alerts, functions that place said alerts on the page and function that convert JavaScript errors into alerts.

navbar.html / .ts

These files contain the navbar. Every page that has a navbar imports navbar.html, and executes the `initNavbar` function at some point. The navbar only displays links to a page that the user is allowed to access, according to their role. It also includes a dropdown with links to the change password and logout page. The link to the page that the user is currently on, is disabled.

We will now list all the files that map directly to one page in the application.

admin.html / .ts

Contains the welcome page for administrators. Displays the user email and role

change_password.html / .ts

Contains functionality to change the user's password when they are already logged in (not the same as forgot password). Contains some input validation. If the input is valid, and the user has successfully entered their previous password, changes their password to the new one that was just entered.

edit_courses.html / .ts

The largest files in the project. Contains functionality to add new courses, and edit, delete or get dumps of existing courses. Also allows users to change the schemas of these courses by entering text, uploading a file, or by schema transfer from another database. Also displays all databases associated with the course, and allows the user to search this list, and edit or delete or download a dump for any of these databases. Furthermore, displays a searchable list of all users that can be given, or stripped off, TA status. Parts of this functionality are disabled or hidden depending on the user's role.

edit_users.html / .ts

(Admin-only)

Contains a filterable list of users, and gives the admin the ability to change other users' role, or delete them. Also, all the databases that a user owns are displayed, and the admin can delete, reset or get a dump of this database.

forgot_password_page.html / .ts

If the user forgot their password, they can click a link on the login page which brings them here. Checks if the user enters a valid email, and if so, sends the user an email with a link to the `reset_password` page (containing a secret temporary token for this user to reset their password).

login.html / .ts

Login page. Has some input verification, so that the user does not have to wait for a page refresh to be told that they did not enter a `utwente.nl` address. Contains links to register page, and forgot password page. If input is verified, sends a POST to `/login` and redirects the user to their home page if the login attempt was successful (`admin.html` for admins and teachers, `student_view.html` for students).

register.html / .ts

Register page. Contains some input validation to check if the user enters a valid `utwente.nl` email, and a valid password according to some security constraints. If the input is validated, sends an API call. If the API call succeeds (no duplicate email or other errors), sends the user an email to verify that they own that email address.

reset_password.html / .ts

Reset password page. The user only gets to see this page after clicking a link contained in an email sent from the 'forgot password' page. Contains some input validation on the password. If the input is valid, changes the user's password to the one that was just entered.

student_view.html / .ts

Page that students get to see. After they enter a valid integer as group name, allows them to request a database for active courses. If the user already has a database for a course, displays the credentials, and allows the user to delete, reset or get a dump of the database.

Security Analysis

As with any software, security is highly important. DAB specifically needs a superuser on the database server it is administrating, which could lead to remote code execution if not properly secured. This analysis will cover possible vulnerabilities, possible attack vectors, and the steps that have been taken to secure them.

Internal Security

This section will focus on the security-related behavior of the program itself, the possible vulnerabilities, and the steps that have been taken to secure them.

UT exclusivity

The system is only intended to be used by the University of Twente. Enforcing this lowers the security risks significantly, as any possible attacker would come from inside the campus, making them known to LISA (the UT's ICT service) as well as dramatically lowering the number of potential attackers. To enforce this, the user must sign up with a UT email address. As these email addresses are only given to people associated with the UT, the UT exclusivity is practically enforced. Furthermore, the domain of the address may have no more than one subdomain, e.g. `student.utwente.nl`, so domains generated via SNT's DAS^[40], which are handed out to students and are of the form `<name>.student.utwente.nl`, are also excluded; if one of these student servers

would run an open relay, this could not be used to circumvent our UT exclusivity.

After the user has created an account, said account can't be used without the email being verified. To verify the email, the system sends a message to the specified email address containing a link with a 128-bit token. While this token could theoretically be guessed, this is practically infeasible. If the link is accessed, the system has a pretty good guarantee that the user does indeed own the provided email address, is therefore affiliated with the UT, and will allow them to log in.

Authentication

To ensure a user is who they say they are, they must be authenticated. The authentication is done via a POST form, which is standard practice. Assuming the production system runs over HTTPS, which we have explicitly mentioned being a security requirement in the installation instructions, this data could not feasibly be intercepted.

Once the data enters the system, the password is compared to the one in the database (see also: information security), and if correct the user is logged in.

The user's authentication details are stored in the session. This means the user's device can only see a random token, which the backend matches to their user id. Therefore, if the user device were compromised, their credentials are not in danger. Furthermore, the session expires after a certain amount of time, so even if the session token were to be stolen, it is only temporarily useful.

Therefore, there is a slight authentication vulnerability if the user device were compromised: an attacker could impersonate that user for a short amount of time. To prevent the attacker from being able to compromise this user permanently, the password change form requires the previous form to be submitted.

As previously mentioned, the primary identifying characteristic of the user is their

email, and they use their password as private authentication. If they have forgotten their password and are verified, they can click a password forget link, which will send another email with a 128-bit token to their email address. Like with account creation, the user knowing the token is proof of their identity. Finally, to increase security, password reset tokens are only valid for 4 hours.

Authorization

Like in most management systems, not all users in DAB are equal. A teacher, for example, can create a course, whereas a student cannot. Authorization is what enforces this difference.

There are two types of vulnerabilities related to authorization; vertical escalation, and horizontal escalation.

DAB distinguishes between three types of users: administrators, teachers, and students. In the code, these are modeled by the integers 0, 1, and 2 respectively. Vertical escalation is when a user can do something above their user level, e.g. a student creating a course. To prevent this, every backend call checks the role of the user before making any changes to or providing the user with data. For some calls, this is particularly easy, as they may only be accessed by certain user levels; these calls have a decorator above their code blocking access to lower user levels. Other calls use more complex logic to figure out whether a user is allowed to do something; for this purpose, a function has been made that safely checks the session cookie for the user role.

Horizontal escalation is when a user does something that their user level may do but is outside the scope of the specific user; e.g. a student requesting a database for another student. To prevent this, backend calls also check whether the object is owned by the user before changing or providing the user with data. Ownership, in this case, means that the object has been created by the user. In principle, a user may only view and edit

objects that were created by them, although there are several exceptions to this, e.g. a teacher being allowed to view all databases within their own course and everyone being allowed to see a list of databases.

As long as the backend calls correctly check the role of the user and the ownership of the data, there is no possibility for vertical or horizontal escalation. While we can't guarantee this, the way that we have set up our code, to first check the permissions before making any changes, makes such programmer error unlikely. Any errors that nevertheless exist could be discovered by having a field tester try out different call parameters.

External security

This section will focus on points where a security breach of or through our system could be used to breach other systems, in particular, the host server. Note that a breach of the host server would also be a breach of DAB itself, as an attacker could manipulate DAB through the said breach.

Database

As DAB is responsible for managing the database, it needs a superuser account on the database server. Furthermore, in order to work with schemas, it quite often dumps entire databases and executes large amounts of raw SQL. As such, a breach of DAB could escalate to the entire database server.

To prevent this, the following precautions were taken:

- Any time DAB interacts with schemas, such as the creation of dumps or the reset of a student database, it will use the credentials of the student database. As these credentials only have permission on that particular database, any security flaws in calls that use them can't escalate to other databases.
- To reduce the risk of SQL injection^[41], the only string that a student can enter into the database is their email, which has to match a

strict regular expression. The group number is not a string, as it must parse to an integer before being inserted into the database. Teachers can enter course names and course information as a string, but these are escaped before being loaded into the database. Course names are used in the generation of student database names; to prevent injection, they therefore have to match a regular expression that severely limits the character set. To ensure that they are not circumventable due to programmer error, all of these regular expression checks are done at ORM^[27] level.

- Any schema that will be used in the database, whether it is uploaded or transferred from another database, will first be imported into a contained database, generated in the same manner as a student database. This ensures that no database corruption can occur through the importing of bad schemas into student databases.

Shell

For some purposes, most notably working with schemas, other terminal utilities from the host server are used. The only user input passed to these utilities as a parameter is the course name, which has to match a strict regular expression. To further decrease the risk of shell injection^[42], these utilities are explicitly called as direct subprocesses without the interference of a shell. As such, the risk of shell injection is negligible, despite the frequent use of the shell.

Frontend

There are only 8 instances of user input. All of these are secure.

1. The user entering their email on the register page (or the admin entering it on the edit user page). This email has to match quite a strict regular expression that cannot contain characters such as <, > or ;. Thus, there is no possibility for SQL injection^[41] or XSS^[43] here
2. The user enters their password on the register page, change password page, and

reset password page. The password is never processed or displayed as a string. It is processed as bytes, hashed, and stored as Base64.

3. The user enters a group name when requesting a database. This is only allowed to be a positive integer on the front-end and is stored as an integer on the database. Any form of injection or XSS is not possible here.
4. The course name is entered by a teacher or admin when creating a new course. This name has to match a regex (on both the front-end and back-end) which only allows alphanumeric characters, spaces, - and /. Any form of XSS or SQL is not possible here.
5. The edit user page allows the admin to change the user role. This is done using a dropdown, and on the back-end, the role is stored as an integer. Thus, there is no possibility for XSS or SQL injection here.
6. On the edit courses page, when the admin creates a new course, they can select the owner of the new course from a dropdown list. This is sent to the back-end as an integer (the ID of the owner) and stored as an integer. There is no way to send or store strings here, thus there are no vulnerabilities.
7. The course info field, which is specified by the admin or teacher when creating a new course, is converted into HTML special characters^[44] on the back-end so that we do not pass characters such as <, > and ; to the database, but pass < and &semi, etc. This prevents SQL injection, and XSS, as these characters are displayed as they normally would be displayed on the HTML page, without the possibility of interpreting them as <script> tags.
8. The search fields on the front-end do not send anything to the back-end. Their function is to look for elements in an array that contain the same text as the search box, and alter their "display" boolean. Even if the user would be able to run JavaScript code by doing this, it will only be run on their own machine. The same could be achieved by directly typing things into the browser console, thus this is not a security risk.

Information security

DAB contains two kinds of valuable information.

The first, and most obvious, is user information. As the users are all part of the UT, they are already aware of each other's email, meaning that a leak of email addresses would not be a disaster. Passwords, however, are still highly confidential information. Fortunately, user passwords only need to be compared, and not read; they can therefore be hashed. DAB stores passwords as an sha3_256^[45] hash, because this was available by default in Python. Passwords are also salted using a 48-bit salt^[46]. Furthermore, the passwords storage is designed to be upgradable in production; the passwords algorithm and salt length can be changed easily if this becomes necessary in the future, without breaking compatibility with older passwords.

The second type of valuable information is the contents of the databases themselves. While teachers and teaching assistants need access to them, an unauthorized student could use this access to copy answers from their peers. To prevent this, the authentication/authorization system described above plays a role, as do the measures taken to prevent breaching the database server. Finally, passwords for student databases as they are stored in the database are not the same as the passwords the students use to log in; they are secured using a bitmask of equal length stored in the project settings. This means that even if an attacker compromises the master database, this will not yield them any usable credentials.

Potential for abuse

Unfortunately, there is no way to check the contents of the student databases. This means that there is nothing stopping users from using their databases for purposes

outside of the teacher's intentions. To a certain extent, this is a calculated risk. Prof. Maurice van Keulen has indicated that there has been no significant abuse of practicum databases in the past, and he feels that any methods to prevent such abuse would hinder the ability of the students to experiment with their databases.

However, this does not mean there is no abuse prevention at all. Firstly, there is a limited number of databases that a student can request; as they can only request one database per active course, a student can't request more databases than there are active courses. Secondly, the contents of a database can always be inspected by the course teacher, teaching assistants, and system administrators, who can choose to delete the database if they notice unacceptable activities. Finally, the databases will be deleted with the course; this means that a user who decides to use the database for purposes outside of the practicum, they can only do so for the duration of the course.

Conclusion of security analysis

The external security of DAB is relatively good; both database and shell injection through DAB are extremely unlikely. The information security is of similar quality. The potential for abuse is present. However, past experiences, as well as the few highly practical anti-abuse measures, make the risk manageable. The authentication system is also quite solid, although a user leaving their device unattended could temporarily compromise them. The largest security risk is programmer error in the authorization code, but this can be avoided through thorough security checks. Finally. The fact that the system can only be used by people affiliated with the UT dramatically lowers the number of potential attackers, thereby reducing the security risks quite a bit.

Testing

The test plan for the project is discussed with the client as to which parts are important to have tested and how to test these parts. For testing, we tried to adhere to the V-model^[47].

The following parts are part of the test plan after discussion with the client.

Unit tests

A program is composed of smaller components, and errors in the individual components are the first thing that should be tested for.

Firstly, there are the backend unit tests. These are not in any way linked to Django; they simply use the requests module to do HTTP calls, because it should not matter what technologies the backend uses; it should simply do what it promises. Because of this, it could be argued that these tests are more integration than unit tests; however, because the program is RESTful^[21], a single REST call is the smallest unit that can be tested, and since we can assume Django to be reliable, this practically functions as a unit test.

The test can be found in `/src/tests/test.py`. It first logs in using four different users (which have to be present in the system for the test to work): an admin, a teacher, a TA, and a student. The test then tries to perform the following actions, testing that the response is consistent with the documentation and the previous calls:

- Login of each of the four users
- Attempt to log in as a non-existent user
- Creating a course (teacher)
- Adding a TA to this course (teacher)
- Creation of a database in this course (student)

- Connect to said database using the provided credentials, and try the different statements a student should be able to perform (student)
- Dump the database (student)
- Retrieve information about the database (teacher, TA)
- Reset the database to the original schema (student)
- Delete the database (student)
- Set incorrect schemas for the course, which should return certain types of errors (teacher, TA)
- Remove the TA from the course (teacher)
- Delete the course (teacher)

All of these 17 tests pass successfully at the time of writing this report.

We also wrote tests for the front-end, but these are integration tests, not unit tests.

Field test

The client gave a clear indication that this part is essential. There should be numerous field tests to see how good the program “holds” during a real-life test. The real-life test consisted of the following 3 parts. The tests are in order of early development until the final product. This will give a gradual test plan during the development.

Team member test

This is the first field test. This simply means that as a group team members will test each others parts. This is to simply test early bugs in the program. The following parts were tested during the project.

- The front-end development team has tested the back-end code, as the front-end uses this code extensively
- Conversely, the back-end team occasionally tested the front-end, because that was the easiest way of calling the back-end
- During the final phases of the project, we had to set up the production environment on

the bronto.ewi.utwente.nl server, together with Jan Flokstra. While this was already a large system test in and of itself, we prepared for this process by having Freek set up the production environment on his own computer. One of the authors had only worked on the front-end so far and had no experience with production environments of any kind. Thus, if he could install the production environment, an experienced sysadmin like Jan Flokstra could surely do it. The entire process took about 3 hours, and we encountered many errors in the README file that were improved upon before the meeting with Jan.

“Break it” test

This is one step further in development. The goal is to see how “robust” the program is in edge cases. The basic question is: “can you break the system?”. This exact question will be giving to 4 individually persons. Prior will be given the goal and usage of the system (with credentials to use the whole application). After that, the goal of the user is to break the system in however way possible. Found bugs are listed in the results and their fixes are discussed too.

- Throughout the early stages of the module, the authors asked various friends and family members for feedback on the product. The feedback that was received during these tests was very valuable. People like them, who know almost nothing about computer science, should be able to use the product. One of the largest problems that we saw was the fact that people will try to click buttons more than once if they do not immediately see feedback. This completely broke the functionality of our product, because it sent large amounts of API calls requesting the same operation (such as deleting a database).
- About 4 weeks before the deadline, the student view was as good as done. To test this, the authors asked several first-year TCS students to request a database for the

course Pearls of Computer Science. None of the first-years had, at that point, had any prior experience with the product, although they had already done the database pearl of module one. They managed to make an account with their student.utwente.nl addresses without any kind of supervision within a reasonable timeframe (about five minutes). They also did not manage to break the website, despite playing around with it. One of them did find it sad that a turtle emoji was not accepted as a group number.

Acceptance test

In the ninth week of the module, we sent an almost finished version to the customer for the acceptance test, where the customer decides whether the product meets all the requirements (see: requirements). After a week, the client had successfully made two courses that will be used for educational purposes in real-life situations. The client was happy with the result. Therefore, we can say that we met all of the customer's requirements.

Integration tests (front-end)

Finally, when the front-end and backend are mostly done we can automate testing through the browser. We will use Selenium WebDriver^[48] in Python to test various use cases across common browsers.

The test will use an existing admin account to conduct five tests.

- Logging in and logging out
- Registering a new account and deleting the new account, as admin
- Changing the password for the admin account, logging in with the new password, And changing it back.
- Adding a new user, meaning an admin creates an account for someone else, Changing the role of the new user to admin and finally deleting the new user.
- Adding a course followed by requesting, resetting and deleting databases for this

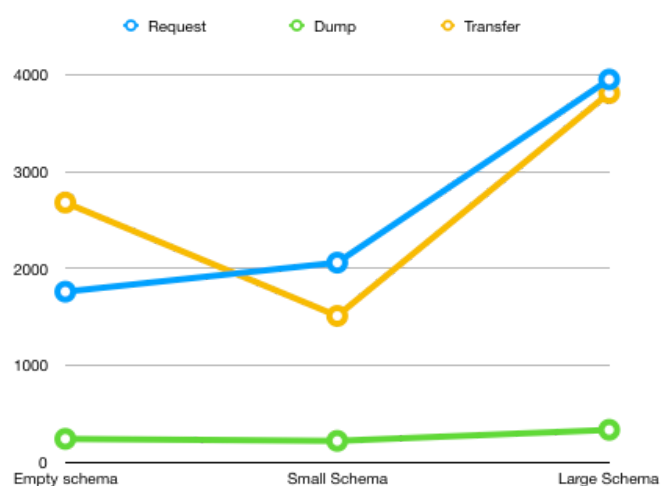
course via different views. And editing the course, adding a TA to the course and removing the TA from the course. And finally removing the course.

Performance

Testing the performance of the system was rather hard, as the database server used for development also ran a myriad of other tasks, which makes it hard to get meaningful results for most of our API calls; the noise would be much more significant than the signal. However, there are three calls that could be long enough to a performance test on requesting a database, getting a dump of that database, and transferring a schema. These calls were each tested on an empty schema, a small schema (the one used for data science) and a very large schema (the one used for Pearls of Computer Science). This resulted in the following data

Type	Empty schema	Small Schema	Large Schema
Request	1760	2060	3950
Dump	241	220	334
Transfer	2680	1510	3810
Lines in schema	44	65	21047

Performance test results. All times are in milliseconds



Performance test graphed

Note that these results were taken on a busy server and that they are accurate to an order of magnitude.

What can be concluded from these results is that the calls that require importing a schema into a database easily take more than a second. This seems logical, as PostgreSQL generally takes quite a long time to create databases. The authors suspect drive access to be the bottleneck in this. Furthermore, it seems that the duration of these calls is indeed related to the length of the schema, but as an increase in schema length of about 30.000% resulted in a call duration increase of about 100%, it does not seem like this is going to be an issue.

Conclusion

In short, DAB can be viewed as a management application used by students to get a digital item used for studying the material provided by teachers.

Going over our [early requirements](#) set using the MoSCoW^[3] system we accomplished all the ‘Must’ requirements, which were the driving force of our development; the ‘Should’ requirements were also met. We were not able to meet the two ‘Could’ requirements. The features that these requirements describe are too complex to develop in the allotted time, and out of scope for this project. It is possible that these requirements will be met in the future by expanding the current system. From the result of the [acceptance test](#) we can conclude that our end-product met the client’s expectations and requirements. The DAB system will be used in modules to come.

Discussion

The back-end has turned out to be a somewhat convoluted, but robust piece of software. While there are certainly some places where the code best practices were not followed, the end result has been

bug-free for the last several weeks of the development process. Said process was rocky, but successful. Initially, there were some conflicts between the two back-end developers. After a while, an equilibrium was established, but this led to a new problem: the developer responsible for the code closest to the front-end could only work at home, being the least in contact with the front-end developers. This problem was eventually resolved by switching these roles, after which the back-end development process went a lot smoother.

Although not as severe as the front-end, the back-end development process was also a learning process, which means that the program would be different if it was rewritten today. In particular, we would try to implement the creation and deletion of the student databases at the ORM level, making the back-end code a lot cleaner and potentially solving a few heisenbugs^[49].

Finally, we would like to highlight a certain bug that we experienced with the transfer of schemas, known as the ‘must be owner of plpgsql’ bug^[50]. This was the result of a bug in PostgreSQL that had been intermittently plaguing us during the development process, that could never be reproduced for debugging. Eventually, it was well documented during a demonstration for the project owners, where it turned out there was a bug in PostgreSQL itself: when dumping a database, it would try to copy the comment on any extensions available in the database. However, in order to modify the comment on default extensions one needs to be a superuser. So if a dump is created by a regular user, one would still need to be superuser to import it. This was eventually fixed by using regular expressions to remove this comment from the dump, but this is really a workaround for a bug in PostgreSQL.

One of the potential features, that was identified to be a wishlist priority at a very early stage, was the ability to use multiple database servers in case there would be too many databases for one server to manage.

Given the capacity and other workloads of Bronto, the current production server, this does not seem to be necessary at the moment; however, it might still be a good idea to think about it.

The feature should not be too hard to implement; one would simply have to add an extra database table with servers, containing the hostname of each database server that could be used, and make sure this functions as a foreign key to all of the student databases. The only remaining challenge would be how to divide the student databases fairly across these servers, but that should be surmountable.

We are quite proud of the front-end. The TypeScript code is very robust and maintainable. However, because the authors had practically no experience with front-end development, the quality of the code written towards the end of the project is a lot better than the code written at the start.

We do, however, have a few improvements that we would like to see ourselves.

Because we did not know that TypeScript fully supports classes, inheritance, and objects, we wrote our code from the perspective of pages that have self-contained functions. We do use some interfaces, which contain the fields for database rows (such as *courseid* for a Course object).

However, instead of having a *deleteCourse* function, which takes a *courseid* as a parameter, we should have turned the interfaces into classes, and have a *Course.delete()* method. This realization only came during the last weeks of the project. While making this change would have improved code quality, it would not have improved the functionality at all. This is why we decided not to implement this change, as it would take quite some time while not improving anything for the customer.

The front-end code uses promises, await and async function calls. This keeps the code very readable, but it is not as parallelized and asynchronous as it could be. Instead of

having global variables such as a list of courses, the global variables could be wrapped in a promise. Then, each function would have to await the promise before doing anything with it. We quickly tested this, and while it did greatly improve the parallelization of API calls, the performance improvement was negligible, and properly implementing this would require days of refactoring.

There are some other front-end code quality issues that were hard to get rid of. Quite some functions written early on take an integer, `i`, as a parameter. This represents the index of a course, within the global "courses" variable. This does not seem like a very elegant solution, but getting rid of this problem would require a lot of effort, while not improving the functionality at all. Another problem is that we have some duplicate code in functions written early on. These same functions also paste a lot of raw HTML into div elements, which is not a very flexible solution. It prevents us from properly adding event listeners to elements, and putting large sections of HTML in our TypeScript files makes it harder to change the styling of the page.

While we are quite happy with the styling overall, there are two problems that we did not solve. Everything on the pages is flexible, and the elements grow with the content and screen size. However, anything on the page that scrolls has been hardcoded to have a certain size because we could not get the flexibility to work properly on elements that overflow. Furthermore, we are not completely happy with how long the admin pages have become (they require a lot of scrolling). A possible solution to this would be similar to the way the user can choose between ways to add schemas: multiple buttons that hide and show parts of the page.

We would have liked to include more pictures in this report such as class

diagrams, but we do not use classes in this project. As mentioned before, we do not use any on the front-end (but we would have liked to), and the back-end does not use classes either (all database abstraction objects are handled by Django and not created by us).

Our planning for this project (as explained in the planning section of the appendix) was quite optimistic. We thought that we would be able to create a minimum viable product in only a few weeks, but properly setting up the front-end while also learning how to write JavaScript, TypeScript, and do proper styling, took quite long for the authors. Getting a properly working production environment also took much longer than expected. However, we did finish our work on time, and we are happy with the result.

One reason for this is that the composition of this project group was not ideal. The authors did not know each other, while some other project groups had years of experience doing projects together. Furthermore, none of the authors had proper front-end development experience. If one of the authors had had this experience at the start of the project, the development would have been much faster. This could have lead to us developing many more of the nice-to-have features such as Docker integration.

Possible future improvements

Right now the manager is equipped to manage databases. In the future, this product could be expanded to manage other digital items. For example, Docker could be a good item to add to this list. This way a student could get access to a small container to experiment with various topics. By adding this functionality, DAB could be expanded for education on any computer science topic, as the virtual environments managed by Docker can contain files and programs on any topic.

Another example would be to add additional Relational Database Management Systems (RDBMS).

Another improvement could be to integrate the OAuth system used by the university. This way a user of DAB only has to login once and is not required to have separate login credentials for DAB.

The final improvement that we considered during development, and could still be implemented, is Canvas^[51] / Horus^[52] integration. With this integration, the student would not have to select a group, and the admin or teacher would have much more information. Roles could be automatically assigned to users based on their role in Canvas, saving the admin time on assigning roles.

All of these features were considered during development (as explained in the design decisions section), but were not included because we were confident the end product would be better if we focused our efforts on the databases. However, because our code is quite modular and maintainable, it should be easy to implement these functions. Now that we now that we have more experience, we could help out and do this ourselves if it turns out to be needed.

References

[1] Wall, L. (1994). The Perl programming language.

[2] Date, C. J., & Darwen, H. (1997). A guide to the SQL standard : a user's guide to the standard database language SQL. Addison-Wesley.

[3] Miranda, E. (2011). Time boxing planning: Buffered Moscow rules. ACM SIGSOFT Software Engineering Notes, 36(6), 1-5

[4] PostgreSQL: The world's most advanced open source database
<https://www.postgresql.org/>
Accessed: 2019-11-08

[5] The Apache HTTP Server Project
<https://httpd.apache.org/>
Accessed: 2019-11-08

[6] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239), 2.

[7] Hart, G. (1996). The five W's: An old tool for the new task of task analysis. Technical communication, 43(2), 139-145.

[8] The Web framework for perfectionists with deadlines | Django
<https://www.djangoproject.com/>
Accessed: 2019-11-08

[9] Rossum, G. (1995). Python reference manual.

[10] TypeScript - JavaScript that scales.
<https://www.typescriptlang.org/>
Accessed: 2019-11-08

[11] JavaScript | MDN
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
Accessed: 2019-11-08

[12] Sass: Syntactically Awesome Style Sheets
<https://sass-lang.com/>
Accessed: 2019-11-08

[13] CSS: Cascading Style Sheets | MDN
<https://developer.mozilla.org/en-US/docs/Web/CSS>
Accessed: 2019-11-08

[14] Bootstrap · The most popular HTML, CSS, and JS library in the world.
<https://getbootstrap.com/>
Accessed: 2019-11-06

[15] HTML: Hypertext Markup Language | MDN
<https://developer.mozilla.org/en-US/docs/Web/HTML>
Accessed: 2019-11-08

[16] Templates | Django documentation | Django
<https://docs.djangoproject.com/en/2.2/topics/templates/>
Accessed: 2019-11-08

[17] npm | build amazing things

<https://www.npmjs.com/>

Accessed: 2019-11-08

[18] webpack

<https://webpack.js.org/>

Accessed: 2019-11-08

[19] The uWSGI project — uWSGI 2.0 documentation

<https://uwsgi-docs.readthedocs.io/en/latest/>

Accessed: 2019-11-08

[20] Ronacher, A. (2018). Flask (a Python microframework).

<http://flask.pocoo.org>

Accessed: 2019-11-08

[21] Roy, T. F. (2000). The representational state transfer (rest). Department of Information and Computer Science, UCI.

[22] Buyya, R., Pathan, M., & Vakali, A. (Eds.). (2008). Content delivery networks (Vol. 9). Springer Science & Business Media.

[23] Responsive Web Design Basics

<https://developers.google.com/web/fundamentals/design-and-ux/responsive>

Accessed: 2019-11-06

[24] Leiba, B. (2012). OAuth web authorization protocol. IEEE Internet Computing, 16(1), 74-77.

[25] Gackenheim, C. (2015). What is react?. In Introduction to React (pp. 1-20). Apress, Berkeley, CA.

[26] Jain, N., Bhansali, A., & Mehta, D. (2015). AngularJS: A modern MVC framework in JavaScript. Journal of Global Research in Computer Science, 5(12), 17-23.

[27] Django ORM - Full Stack Python

<https://www.fullstackpython.com/django-orm.html>

Accessed: 2019-11-08

[28] Torvalds, L., & Hamano, J. (2010). Git: Fast version control system.

<http://git-scm.com>

Accessed: 2019-11-08

[29] The Open Group Base Specifications Issue 7, 2018 edition

<https://pubs.opengroup.org/onlinepubs/9699919799>

Accessed: 2019-11-08

[30] Postel, J. (1982). Simple mail transfer protocol. Information Sciences.

[31] Postfix Documentation

<http://www.postfix.org/documentation.html>

Accessed: 2019-11-08

[32] Martin, J. (1983). Managing the data base environment (p. 381). Prentice Hall PTR.

[33] Bray, T. (2014). The javascript object notation (json) data interchange format.

[34] GitHub - sweetalert2/sweetalert2: A beautiful, responsive, highly customizable and accessible (WAI-ARIA) replacement for JavaScript's popup boxes. Zero dependencies.

<https://github.com/sweetalert2/sweetalert2>

Accessed: 2019-11-08

[35] GitHub - Wruczek/Bootstrap-Cookie-Alert: A simple, good looking cookie alert built for Bootstrap 3/4. No dependencies required.

<https://github.com/Wruczek/Bootstrap-Cookie-Alert>

Accessed: 2019-11-08

[36] GitHub - jackmoore/autosize: Autosize is a small, stand-alone script to automatically adjust textarea height to fit text.

<https://github.com/jackmoore/autosize>

Accessed: 2019-11-08

[37] GitHub - google/gts: TypeScript style guide, formatter, and linter.

<https://github.com/google/gts>

Accessed: 2019-11-07

[38] GitHub - axios/axios: Promise based HTTP client for the browser and node.js

<https://github.com/axios/axios>

Accessed: 2019-11-08

[39] For the source code of the project, click this link, or contact the authors.

<http://floris.thebias.nl/dab.tar.gz>

Last updated: 2019-11-08

[40] Studenten Net Twente | Service | DAS

<https://www.snt.utwente.nl/en/services/das>

Accessed: 2019-11-08

[41] SQL Injection - OWASP

https://www.owasp.org/index.php/SQL_Injection

Accessed: 2019-11-08

[42] Command Injection - OWASP

https://www.owasp.org/index.php/Command_Injection

Accessed: 2019-11-08

[43] Cross-site Scripting (XSS) - OWASP

[https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))

Accessed: 2019-11-08

[44] Entity - MDN Web Docs Glossary: Definitions of Web-related terms | MDN

<https://developer.mozilla.org/en-US/docs/Glossary/Entity>

Accessed: 2019-11-07

[45] Dworkin, M. J. (2015). SHA-3 standard: Permutation-based hash and extendable-output functions (No. Federal Inf. Process. Stds.(NIST FIPS)-202).

[46] Garfinkel, S., Spafford, G., & Schwartz, A. (2003). Practical Unix & Internet Security, 3rd. Edition (pp. 85-88). USA O'Reilly.

[47] Balaji, S., & Murugaiyan, M. S. (2012). Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. International Journal of Information Technology and Business Management, 2(1), 26-30.

[48] Selenium WebDriver — Selenium Documentation

https://www.seleniumhq.org/docs/03_webdriver.jsp

Accessed: 2019-11-08

[49] Grottke, M., & Trivedi, K. S. (2005). A classification of software faults (p. 2). Journal of Reliability Engineering Association of Japan, 27(7), 425-438.

[50] PostgreSQL: BUG #14311: ERROR: must be owner of extension plpgsql

<https://www.postgresql.org/message-id/20160905121142.1404.83483%40wrigleys.postgresql.org>

Accessed: 2019-11-08

[51] Canvas the Learning Management Platform | Instructure

<https://www.instructure.com/canvas/en-gb>

Accessed: 2019-11-08

[52] Horus User Manual

<https://horusapp.nl/manual.pdf>

Accessed: 2019-11-08

Appendix

Use case model

The roles of actors, as described below, work in a hierarchical model where a student has limited access to and control over the system. A TA is also a student, but has more rights over certain courses. A teacher has all the rights of a TA and by extension also a student, and more. An admin has full rights.

Glossary:

<i>Term</i>	<i>Description</i>
Course	Representing a module students can sign up for, provides databases with optional schema's
SQL dump	A SQL file generated from a database which can be used to create a database that is the same as the one the dump was generated from
Database	A remote storage unit where data is stored in tables
Schema	A preset state for a database
TA	Teaching assistant, an experienced student assigned by a teacher to help other students
Role	A student, TA, teacher or admin
Migrations	Containing a system into a single file so that the system can be transferred between servers easily
WebApp	An application built for web browsers
Active/Inactive	Describing the state of a course, active courses are visible to all users. While inactive courses are only visible to admins, TA's and their owners.

Requirements list:

#	<i>Requirement</i>	<i>Use case(s)</i>
1	As a student, I want to login using my login credentials	Log In
2	As a student, I want to logout	Log out
3	As a student, I want to request a database for a course	Request Database
4	As a student, I want to delete my database for a course	Delete Database Self

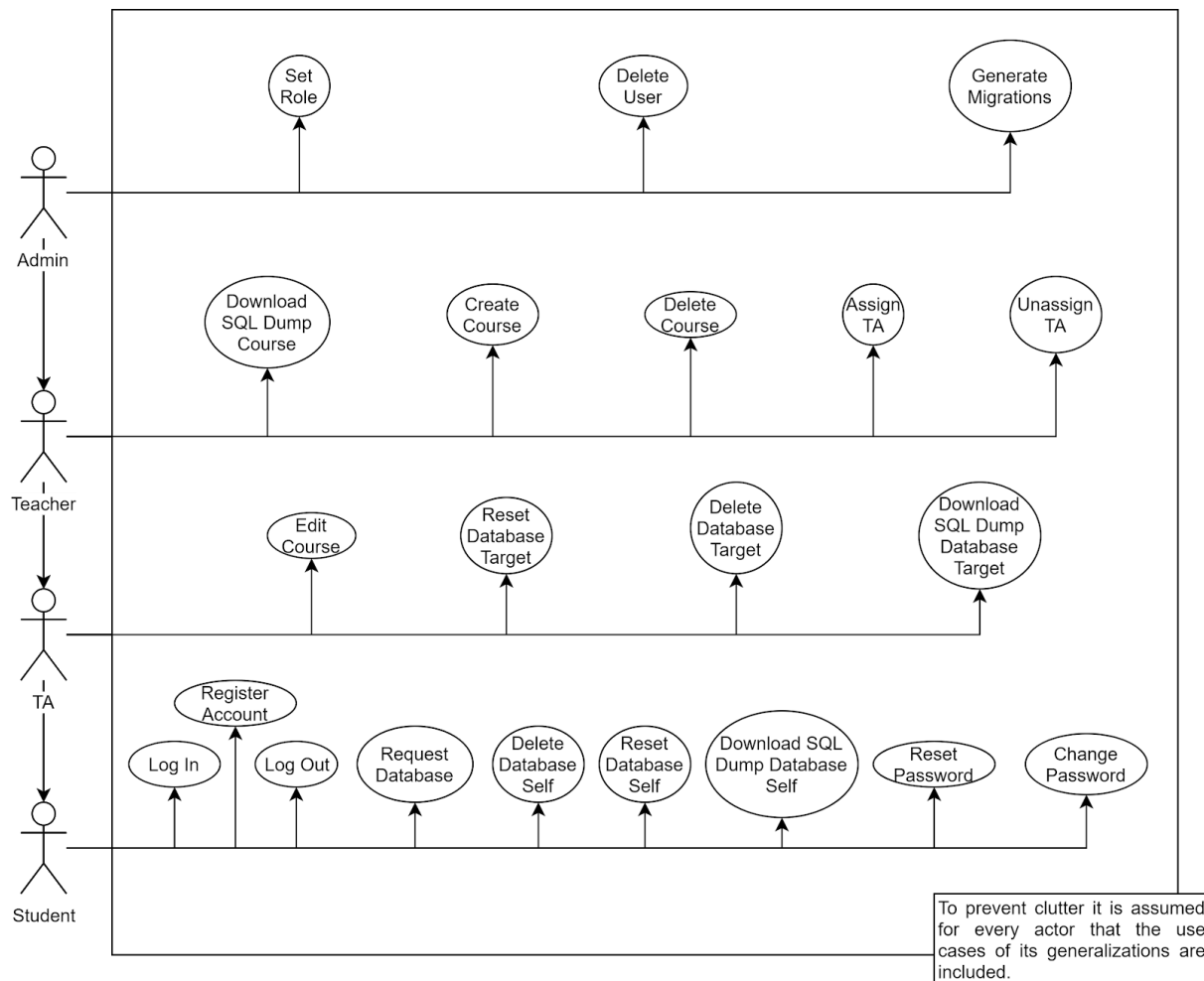
5	As a student, I want to reset my database for a course to the course schema	Reset Database Self
6	As a student, I want to download a SQL file of a database I own	Download SQL Dump Database Self
7	As a student, I want to be able to change my password when I forget it	Reset Password
8	As a student, I want to change my password	Change Password
9	As a student, I want to register an account using my UTwente email	Register Account
10	As a TA, I want to edit the course I am assigned to	Edit Course
11	As a TA, I want to reset/delete/dump databases of other users in the course I am assigned to	Reset Database Target, Delete Database Target, Download SQL Dump Database Target
12	As a teacher, I want to reset/delete/dump databases of any user in courses I own	Reset Database Target, Delete Database Target, Download SQL Dump Database Target
13	As a teacher, I want to download a zip-file with dumps of all databases in a course I own	Download SQL Dump Course
14	As a teacher, I want to create a new course	Create Course
15	As a teacher, I want to edit a course I own	Edit Course
16	As a teacher, I want to delete a course I own	Delete Course
17	As a teacher, I want to assign and unassign TA's to and from courses I own	Assign TA, Unassign TA
18	As an admin, I want to edit any course	Edit Course
19	As an admin, I want to delete any course	Delete Course
20	As an admin, I want to download a zip-file with dumps of all databases of any course	Download SQL Dump Course
21	As an admin, I want to reset/delete/dump any database	Reset Database Target, Delete Database Target, Download SQL Dump Database Target
22	As an admin, I want to assign and unassign TA's to and from any course	Assign TA, Unassign TA
23	As an admin, I want to edit the role of any user	Set Role
24	As an admin, I want to delete any user	Delete User
25	As an admin, I want to directly add another user to the	Add User

	system, register and verify their account for them	
26	As an admin, I want to generate a file that contains the entire system	Generate Migrations

Actor list:

<i>Actor</i>	<i>Description</i>
Student	Requests and manages up to one database per course
TA	A student who is assigned a teaching assistant of a specific course by a teacher or admin and can manage all databases in that course
Teacher	Manages the courses they are the owner of and database that belong to those courses
Admin	Manages all users, courses, and databases in the system

Use case diagram:



Use case description:

Use case	Description
Log In	Using an email and password to gain access to the web app
Log Out	Ending the session on the web app
Register Account	Creating a new account by giving an UTwente mail and choosing a password. An email will be sent to the UTwente mail address where a link is given to verify the email. When the email is verified the user can log in with the new account.
Change Password	While logged in, changing the password by giving the current password and giving a new password.
Reset Password	Request a new password by giving an UTwente mail. An email will be sent to the address with a link to set a new password.
Create a Course	Create a new course by giving a course name, course info, selecting the course owner, whether the course will be active

Delete Course	Deleting a course removes the course for every user and deletes all databases associated with the course
Edit Course	Changing any of the following: course name, course info, active status, schema
Download SQL Dump Course	Download a zip-file of all databases associated with the course
Download SQL Dump Database Self	Download a SQL file of the database you own
Download SQL Dump Database Target	Download a SQL file of a database any user owns
Reset Database Self	Reset a database you own to the course schema
Reset Database Target	Reset a database of any user the the course schema
Request Database	For a course, request the creation of a database and receive the login credentials
Delete Database Self	Delete a database you own
Delete Database Target	Delete the database of any user
Assign TA	Make any user TA for a course
Unassign TA	Remove TA rights from a user for a course
Set Role	Change the role of a user
Add User	Add a user directly to the system by giving their email, role and temporary password, the new user be automatically verified
Delete User	Delete a user and from the system
Generate migrations	A script puts the entire system including courses, users and databases into a zip-file (such that the system can be moved to a different server for example)

Planning & task division (past vs. present)

Initial planning:

Within the context of a 10-week module, starting on September 2nd, 2019.

Week 0

- Find group, choose a project

Week 1

- Tuesday: first meeting with Maurice, first meeting/lecture with the module coordinator

- Asking questions, getting an idea of what needs to be done
 - Formulating first requirements
- Write out the draft project proposal
- Write out draft project planning

Week 2

- Start of the week: second meeting with Maurice
 - Sign agreement on intellectual property
 - Get feedback on the project proposal, project planning
 - Try and schedule the final presentation already
 - Other small things such as important points on grading schema(discuss final form of design report), and long term planning (absence?)
- Start writing out a draft test plan
- Finish project planning and project proposal
 - Show to Maurice, hopefully agree on it
 - Hand in: send to module coordinator
- Start of the first sprint (try and get to a minimum viable product)

Week 3

- **First peer review**
 - According to the project manual, we should have a project proposal and planning by now
- End of the first sprint (hopefully present minimum viable product)

Week 4

- Start of the second sprint
 - Finish anything that was not finished in the first sprint
 - Try and implement all must-have requirements that were excluded from the first sprint
- Test plan should be finished around this time

Week 5

- **Second peer review**
 - According to project manual, we should have a requirements specification and test plan
- End of the second sprint

Week 6

- Start of the third sprint
 - Hopefully, all must-have requirements are done by now
 - Focus on polishing, writing for deliverables, and nice-to-have requirements

Week 7

- **Third peer review**
 - According to project manual, we should have a design, and first prototype
- End of the third sprint

Week 8

- Product should be functional by now, and we should head into the testing / report writing phase

- Start of the fourth sprint
 - Focus on deliverables, testing, polishing

Week 9

- **Fourth peer review/presentation**
- End of fourth sprint
- Final presentation to Maurice should be around this time
- Product and testing should be completely done
- Hand in: send presentation slides to BOZ
- Hand in: manual to Maurice

Week 10

- Final week. Work on deliverables.
- **Poster presentation**
- Hand in: poster at BOZ
- Hand in: design report at BOZ
 - *"including a requirement specification, a global design, a detailed design with a justification of the design choices, a test plan, and test results, and (pointers to) source code and a manual"*

Realized planning:

During the project translating the

Week 1

- Requirement analysis
- Setting up a git repo, development environment
- Initial design decisions (Python, PostgreSQL)

Week 2

- Setting up of testing production environment on db.thebias.nl
- A draft version of the project proposal ready
- First API calls are being realized
- Mock-ups for front-end

Week 3

- Login page ready

Week 4

- Front-end setup ready (Webpack, TypeScript, etc.)
- Student can successfully request credentials

Week 5

- Student can enter a group number
- Register page ready
- Decided on final styling of front-end (two columns)
- Back-end structure is already a rough draft of the final product
- Back-end unit testing

Week 6

- Student view page is done
- Forgot password page ready
- Implemented proper navbar

Week 7

- Edit courses page is done
- Back-end unit testing is done

Week 8

- Edit users page is done
- Added "generate migrations" functionality
- The product is finished (rough prototype) and ready to be deployed on the University.
- Started installing production setup on UT servers
- Ethics reflection report
- Field testing

Week 9

- Mostly focusing on polishing and deliverables by now
- Fixing problems after installing DAB on the production server of the University. For example, there were some difficulties with handling /dab/ in the URL.
- Implementing final functionality on front-end: search bars, add user functionality, deleting ghost databases
- Front-end unit testing

Week 10

- Final bugfixes and polishing
- Creating the poster
- Preparing both presentations
- Creating manuals
- Finishing the Design report

Initial task division

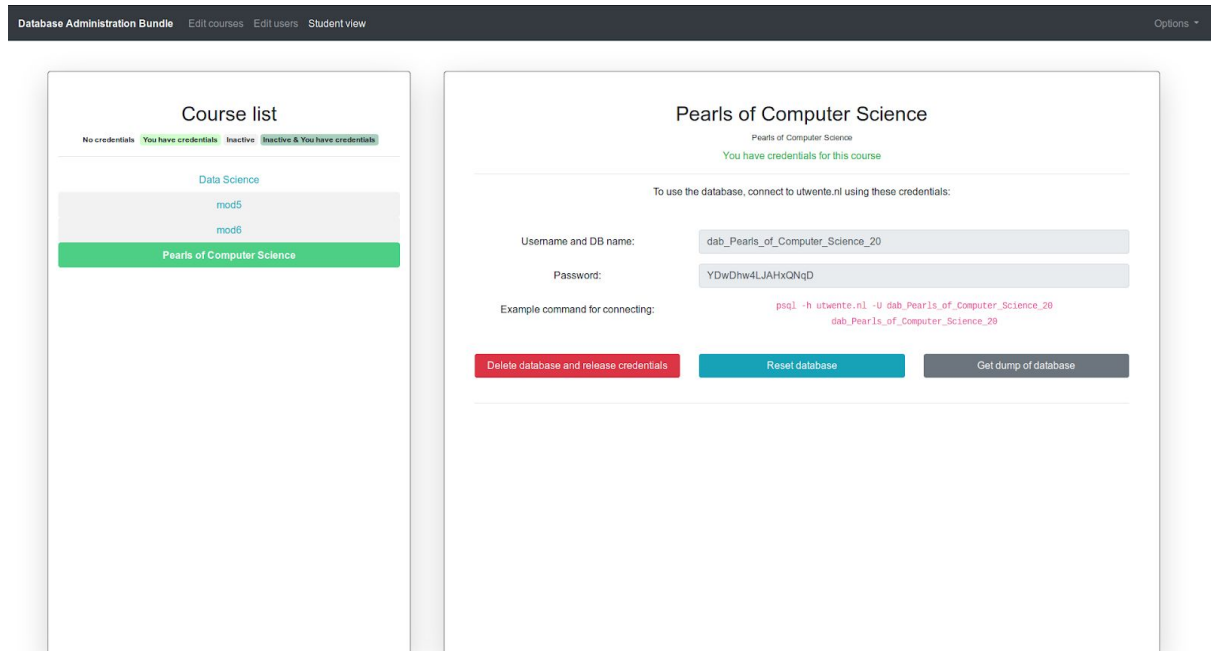
	Denys	Floris	Freek	Raoul
Frontend	0.5		0.5	
Backend		0.5		0.5
Report	0.25	0.25	0.25	0.25
Poster	0.25	0.25	0.25	0.25

Realized task division

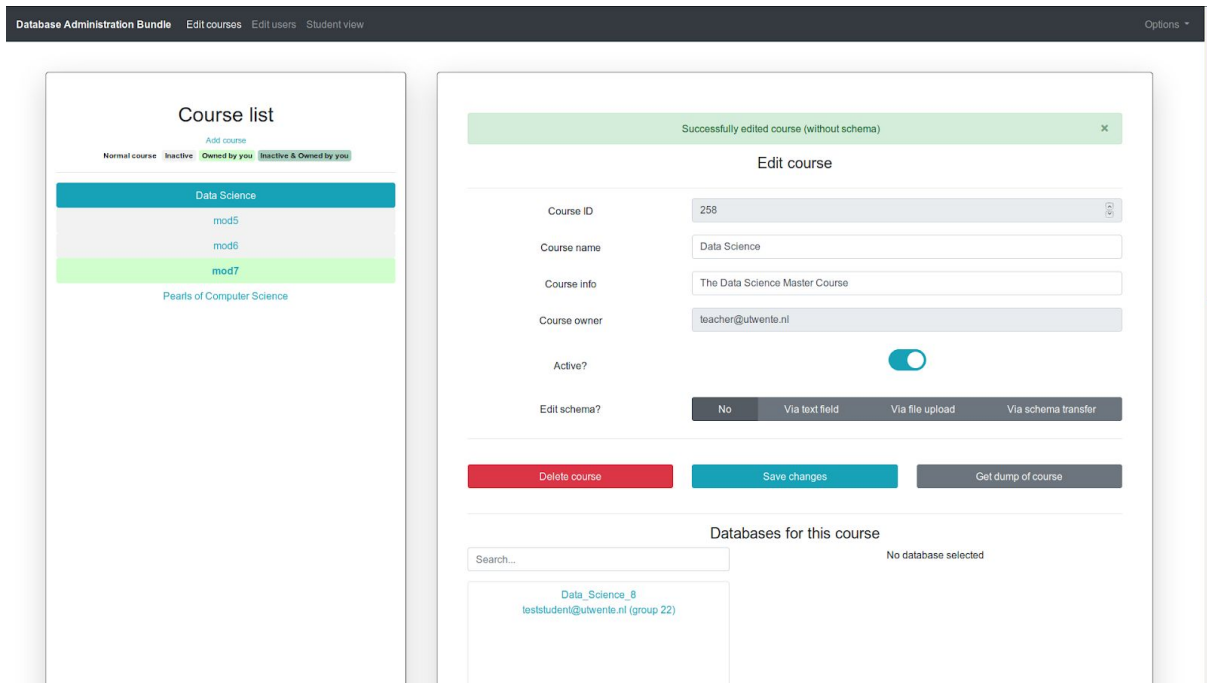
	Denys	Floris	Freek	Raoul
Front-end functionality and testing	0.4		0.6	
Front-end styling	0.5		0.5	
Back-end API calls		0.5		0.5
Proposal report	0.25	0.25	0.25	0.25
Final presentation and manual				1
Poster		0.5	0.5	
Production environment		0.8	0.2	
Design report	0.25	0.25	0.25	0.25
Peer review presentations	0.25	0.25	0.25	0.25

Webapp overview

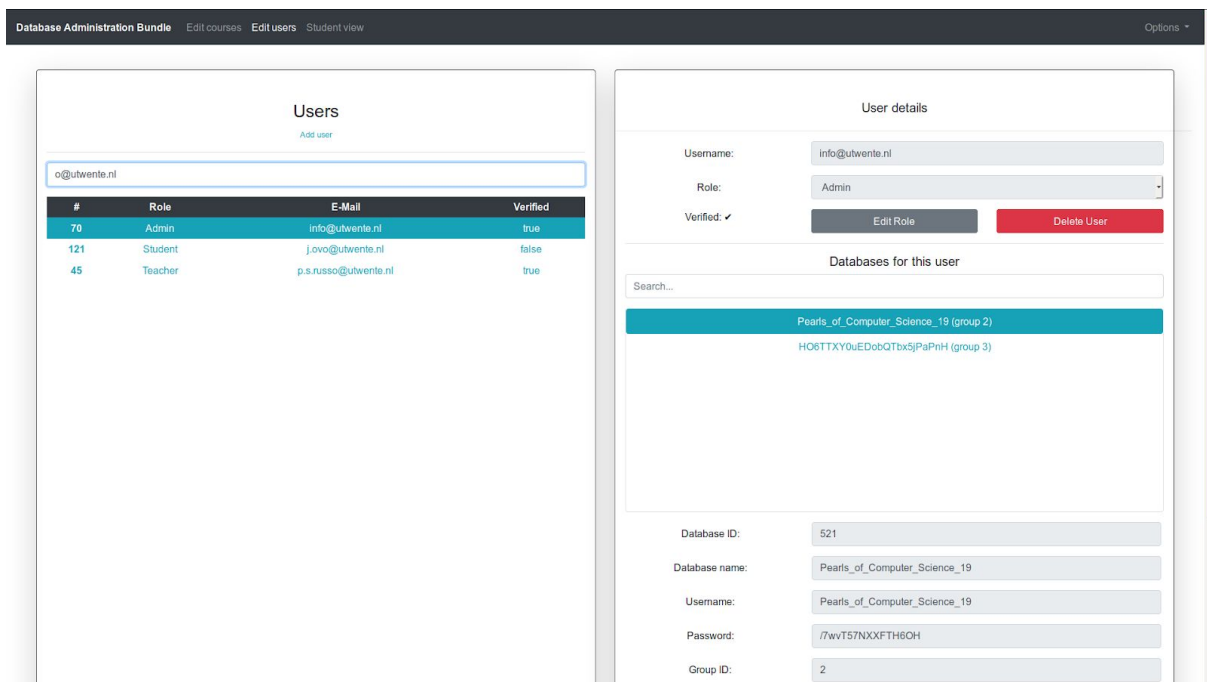
To give a general impression of the application below are some screenshots of the web app provided. More information about every single function can be found in the manual. The student view is accessible to every user. The second page is the edit courses view. This page is visible to a teacher, admin or a TA. For the TA only the specific course will show up. The last screenshot is of the page edit users. This page is only available to the administrator.



Overview of the page “student view”

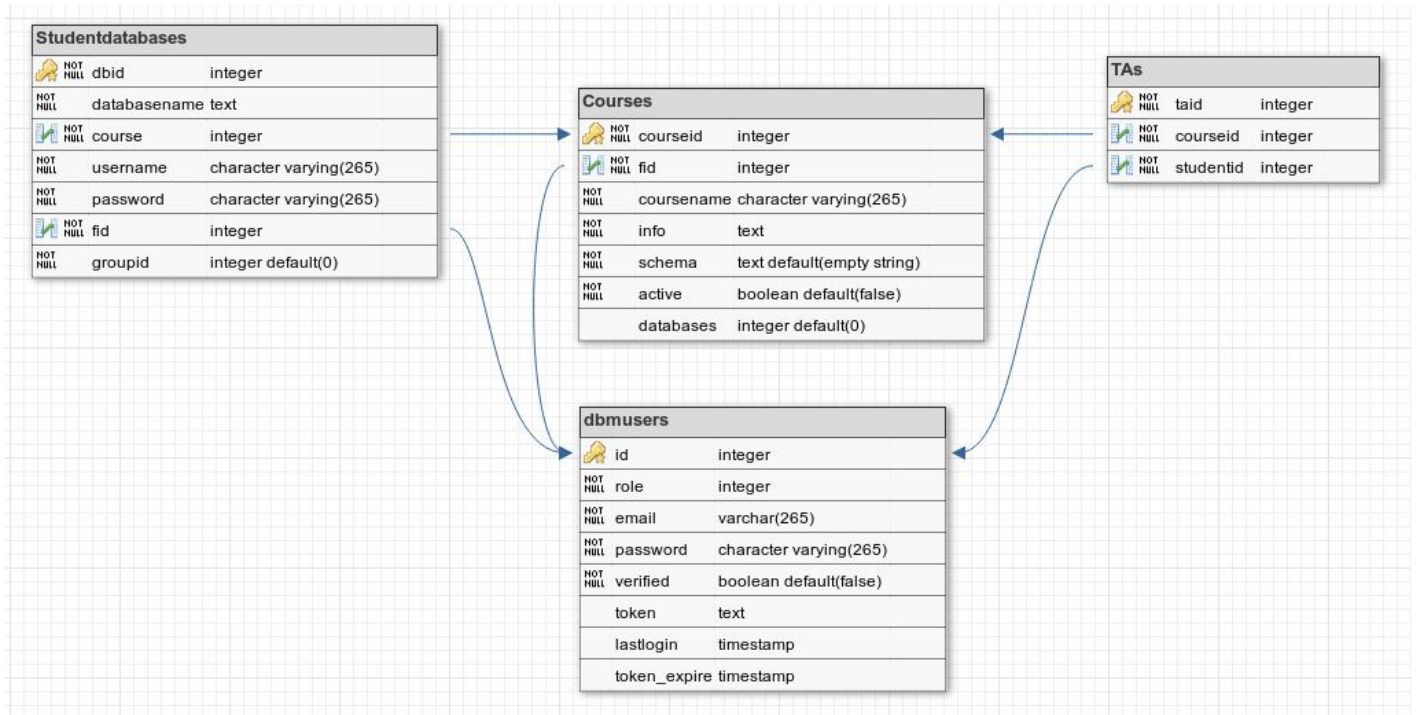


Overview of the page “edit courses”



Overview of the page “edit users”

Database design



Overview of the database design. A key with a star next to it is a primary key which will autoincrement when a new row is added. An arrow is a visual illustration of a foreign key connection between two tables.

Roles

The application consists of different types of users. These types are called roles in DAB. There are 3 roles, namely:

- **Administrator(admin in short):** this user can perform any action on any user or database. There are no restrictions. Of course, with these capabilities come great responsibility. Use this role with caution and do not grant every user this level of privileges.
- **Teacher:** this user is able to create, edit and remove courses (belong to the teacher). Any action a student can perform is also available to the teacher. If a student belongs to a course owned by the teacher, the teacher is able to make alterations to this database. The teacher is not able to edit users.
- **Student:** a student can only create or delete course owned by himself. A student can be granted extra privileges. These privileges fall under the term TA. A TA is able to make changes to a course where he or she belongs to. Though, a TA cannot delete a course or create a new one.

Syllabus

Template/schema: A template is a structure of how the database (including tables and schemes) will look like. A template does not require knowledge of any DB manager or command. This is not to be confused with the PostgreSQL definition of a “schema”.

Database practicum database manager

Manual for administrators

Pearls of Computer Science

Pearls of Computer Science

You have credentials for this course

To use the database, connect to utwente.nl using these credentials:

Username and DB name:

dab_Pearls_of_Computer_Science_20

Password:

YDwDhw4LJAHxQNqD

Example command for connecting:

```
psql -h utwente.nl -U dab_Pearls_of_Computer_Science_20  
dab_Pearls_of_Computer_Science_20
```

Delete database and release credentials

Reset database

Get dump of database

Table of content

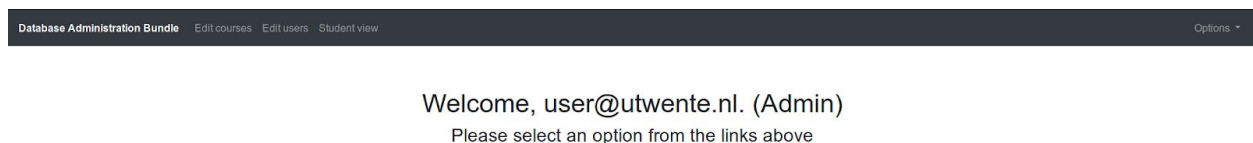
Introduction	3
Welcome Screen	3
Edit Courses	4
Edit users	5
Student view	7
Options	9
Change password	9
Log out	10
Generate migrations	10
View/edit ghost databases	11
FAQ	11

Introduction

This manual is meant for users of the Database Administration Bundle (DAB). This manual will cover all functions which can be used inside of DAB. This manual will handle the functions page for page. The first section will describe the general structure of the site. In the remaining sections the functions per page will be explained. At the end a FAQ is included. In order to use DAB to its full potential it's advisable to read this FAQ. This manual will provide screenshots of DAB used on a desktop. For a mobile phone, the site structure will look slightly different. Apart from some minor visual differences the functions will behave exactly the same.

Welcome Screen

After a successful login an administrator will be presented with the following view:



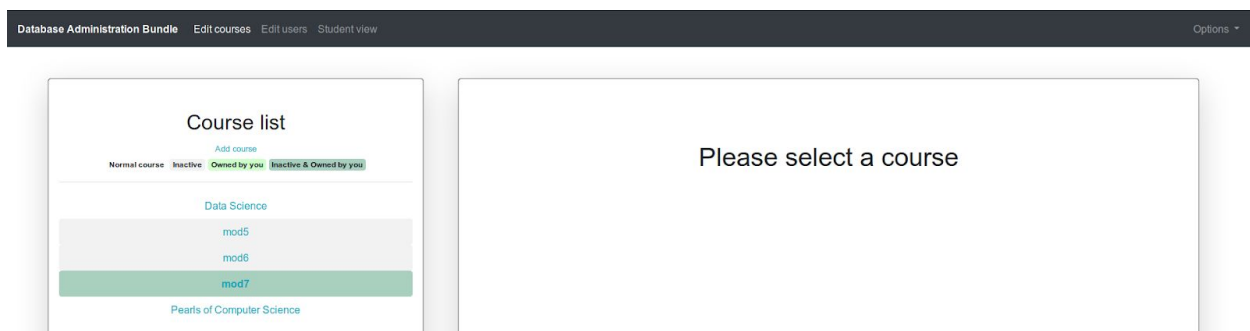
The different pages within DAB can be seen on top of the startpage, namely: Edit courses, Edit users and the Student view page. At the top right is a “options” button with some additional functions.

In short every page contains the following functionality:

- **Edit courses:** This page is also accessible for a teacher and can be used to manage courses. Tasks like creating courses or adding a TA to a course can be managed on this page. Further information can be found in the “edit courses” section.
- **Edit users:** This page can only be accessed by a user with administrative rights within DAB. On this page user can be managed. Like adding a new user to the system or removing a user of DAB. Further information can be found in the section “edit users”.
- **Student view:** This view is visible to every user with at least a student role. On this page databases can be created. Database creation is normally done by a student, but a teacher or administrator can also create a database this way. Further information can be found in the section “student view”.

Edit Courses

When the user clicks the “edit courses” button in the top menu an overview similar to the following will appear:



The view is divided into a left and a right part. On the left is a list with courses. Above the list with courses is extra information provided about the color coding in the list. In short the colors have the following meaning:

- **Normal course:** in this case the course will not have any color. The course is active, but is not owned by the currently logged in user.
- **Inactive:** The course is not active. This means that the course is known inside of DAB, but a student can not yet see this course. This can be convenient in case additional information is provided later in time or to make the course visible to the student at a specific moment in time. This will give the row a light gray color. **Important: although possible, it is not advised to activate the course during a course and let students create databases all at the same time. The best way is to request a database creation via a canvas notification. This will give the best experience for both the student and teacher.**
- **Owned by you:** this course belongs to the currently logged in user. For a teacher this means changes can be made. An administrator can make changes in any case. This will give the row a light green color.
- **Inactive & owned by you:** if the currently logged in user is owner of this course, but the course is not yet visible for students. This will give the row a dark green color.

On the top of this list is a button to add additional courses. Clicking this button will provide the user with a simple to follow instructions on how to add a course. These instructions will look similar to the following view:

The screenshot shows a web form titled "Add course". It includes the following elements:

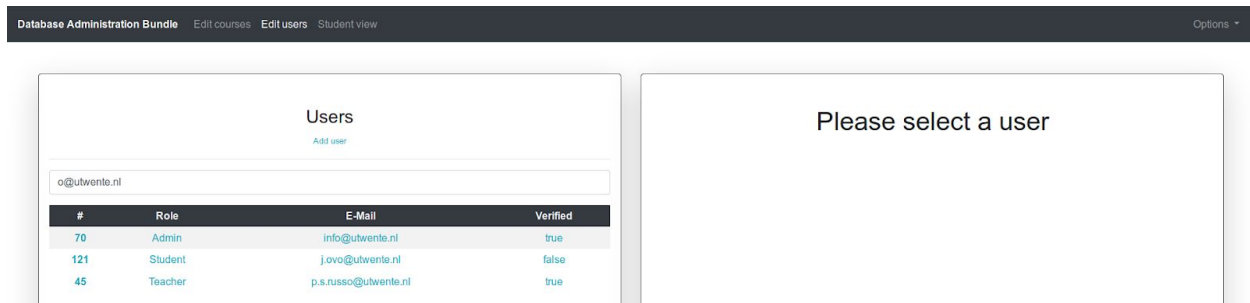
- Course name:** A text input field.
- Course info:** A text input field.
- Course owner:** A dropdown menu with the text "(if unchanged, you will be the owner)".
- Active?:** A toggle switch currently turned on.
- Add schema?:** A group of four radio buttons: "No", "Via text field", "Via file upload", and "Via schema transfer".
- Submit button:** A large blue button labeled "Add course" at the bottom.

An administrator will have an additional option to set the owner of the course. For a teacher this option will not be visible and will automatically be set to the currently logged in user. The active button is convenient in cases where the course should exist already, but not yet visible for students. When set to inactive the course will not be visible for a student, but a teacher or TA of this course will be able to change course settings. An important part of the course creation is the schema part at the bottom. This option can be used to set the structure of the database when the student creates a new database and possibly populate the databases with preset data. There are four options to choose from:

- **No:** this option will give the student an empty database. There will be no tables or any preset data present.
- **Via text field:** Using this option will give the user an extra field where the PostgreSQL code can be inserted directly. This is mostly convenient for a small setup. Larger and more complex database structures can be setup in this way, but is not recommended.
- **Via file upload:** an existing sql dump can be uploaded with this method. Every newly created database by the student will have the information as setup inside of the sql dump.
- **Via schema transfer:** this option can be used to transfer the schema from an existing database. This option is convenient in case the teacher creates a new iteration of the same course every year. In this case a new course can be created with an already existing schema structure.

Edit users

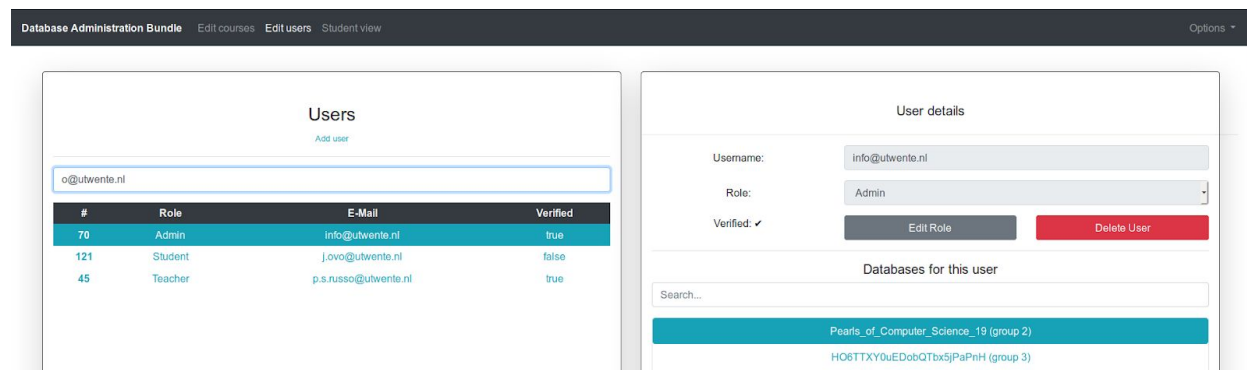
Editing users is an option only available to administrators. After clicking this option in the upper menu an overview similar to the following will appear:



The overview is divided into two parts. On the left is a list of users. On top of the list is an option to add additional users directly. This is most useful when adding new teachers or administrators. A student can create an account directly through DAB itself. After following the instructions an email will be sent to the new user. Keep in mind that the mail address should be a valid university email address and no other. An administrator can also filter on the mail address. In the picture above a search is made on “o@utwente.nl”. The right part of the screen will provide a complete overview of the user selected on the left.

The list on the left will give basic information about every user. The ID next to every user is the ID used inside of the database. The verified field will give an indication whether or not the user has verified its mail address via a link received from DAB.

When a user is selected the view on the right will look similar to the following image:



An administrator will be able to change the role of the user. This function is useful in the case when a teacher has created a new account through DAB itself and currently has a student role. Beneath these user details is a list of all the databases created by this user. After selecting one of these databases additional information will be loaded beneath the database list. This information will look similar to the following image:

Database ID:	188
Database name:	Lnqhohm1WT8PIC2OQNEV3Pr
Username:	Lnqhohm1WT8PIC2OQNEV3Pr
Password:	OQmHKbtVle1AaIXNSfKF0Zi6SplsV9Xo5lcFuevDqzJh
Group ID:	0
Owner ID:	73
Course ID:	12
Course name:	mod5

[Download dump](#)
[Reset database](#)
[Delete database](#)

The functions below the additional information are the same functions a student can perform on its own databases and are further explained in the *student view* part below.

Student view

A teacher or administrator is presented with the following view when clicking the student view button at the top menu:

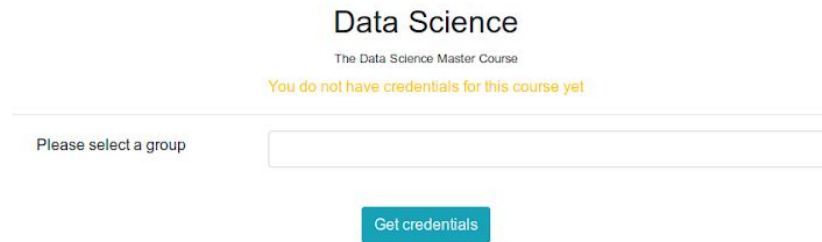
The screenshot shows a navigation bar at the top with 'Database Administration Bundle', 'Edit courses', 'Edit users', and 'Student view' (selected). On the right of the bar is an 'Options' dropdown. The main content area is split into two panels. The left panel, titled 'Course list', shows a legend with four categories: 'No credentials' (white), 'You have credentials' (light green), 'Inactive' (dark green), and 'Inactive & You have credentials' (light grey). Below the legend is a list of courses: 'Data Science' (white), 'mod5' (light green), 'mod6' (light green), 'mod7' (light green), and 'Pearls of Computer Science' (dark green). The right panel displays the text 'Please select a course'.

The view is split in two parts. On the left is an overview of all the courses a student can sign up for. On top of the list is an indication what different colors mean. The following list gives an explanation of the different colors:

- When the course has no color the user does not have a database yet for this course.
- When the course is light green the user already has a database for this course.
- When the course is dark green the user has a database for this course, but the course is not active. This can be convenient to test a database without actually presenting the course to the students yet.
- When the course is light grey the course is simply inactive and no database is active on the server under this username.

Any additional information about a database is presented in the right part of the screen.

When a user clicks a course without any color the following screen is presented.



Data Science

The Data Science Master Course

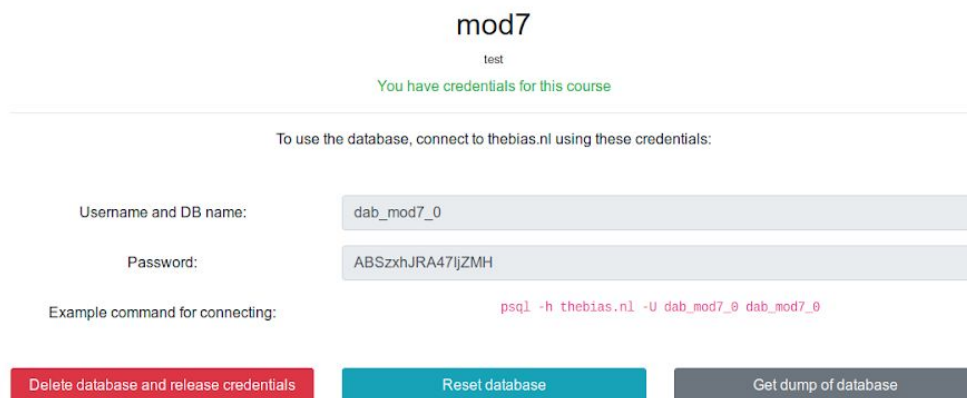
You do not have credentials for this course yet

Please select a group

Get credentials

A student first has to provide a valid group number in the course. This group number will be used by the teacher or TA to assist during the course. A teacher or administrator can use any number.

After the user clicks “get credentials” DAB will generate a database with credentials for the user. The course will become light green in the left plane and an overview of the database will show up on the right. This overview will look similar to the following image:



mod7

test

You have credentials for this course

Please select a group

Get credentials

Username and DB name:

dab_mod7_0

Password:

ABSzxxhJRA47ljZMH

Example command for connecting:

psql -h thebias.nl -U dab_mod7_0 dab_mod7_0

Delete database and release credentials

Reset database

Get dump of database

This overview will give the user a few options and essential information about the database itself. The username and name of the database are identical and are shown in the first row with the password of the database in the next row. A CLI command to connect to the database is provided in the next row. Make sure that psql is installed before using this command.

A user has 3 different options below the essential information. The buttons will perform the following functions:

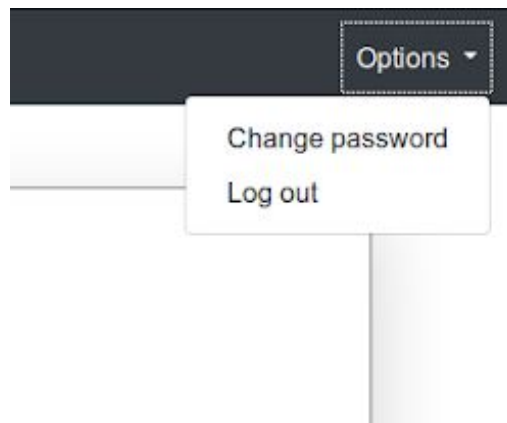
- **Delete database and release credentials:** Clicking this button will delete the database **including** all its content. The database will no longer exist on the server. Use this option with caution. A user will be asked for confirmation before proceeding.

- **Reset database:** The teacher will have setup a default structure of the database when creating the course. A user can revert back to the beginning state by clicking this button. All additional changes made by the user of the database will be gone, though the database will still exist with the default structure and content.
- **Get dump of database:** This button will create a sql dump file. This file will contain the complete structure and all the data of the database. This function can be used to locally recreate the database. Information stored on the local version will in no way synchronize any data with the database running on the server. All changes made locally will stay locally.

Options

Change password

In order the logout or change your password the option button in the top right corner can be used, as is seen in the following image:



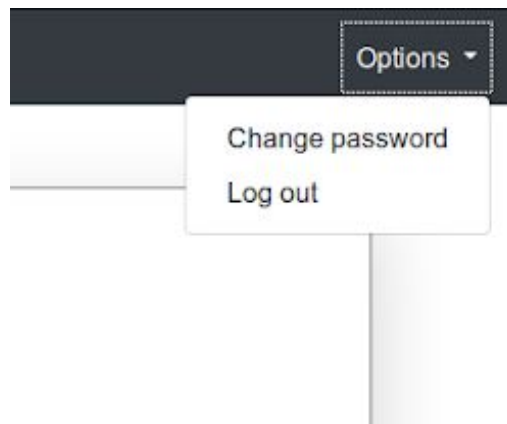
Depending on the type of user more options may be available here. Clicking the change password button will bring up a new screen similar to the following screen:

A screenshot of a 'Change password' form. The form has a title 'Change password' at the top. Below the title are three input fields: 'Current password', 'New password', and 'Confirm new password'. Below these fields is a small line of text: 'Please use a password that has 8 characters or more, and contains at least one lowercase letter, one uppercase letter, and one number'. At the bottom of the form is a blue button with the text 'Change password'.

Simply follow the instructions on the screen and click “change password”. A new password will be set for this user.

Log out

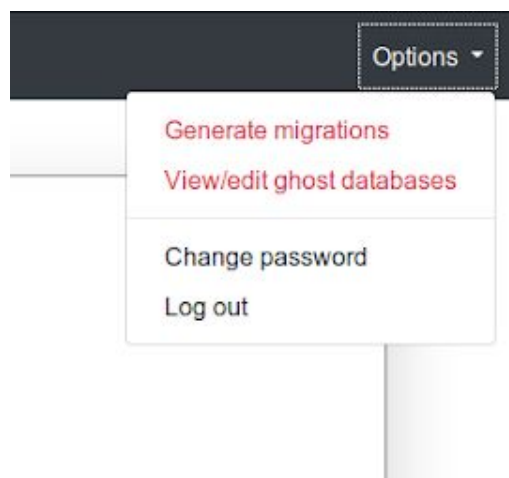
In order to log out the “log out” button can be used in the options drop down menu in the upper right corner, as is seen in the following image:



Generate migrations

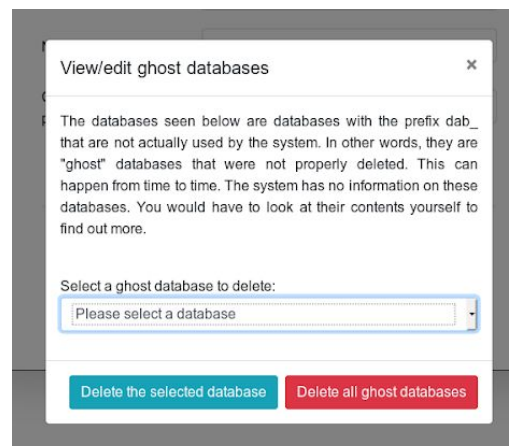
Use this function with caution, administrator credentials may be included in the generating file

This function is included in the options menu in the upper right corner (see the image below). After a confirmation popup a shell script is created. This script can migrate all databases in the system. The location of this shell script will be shown after it has been generated. The created script has to be run *manually*.



View/edit ghost databases

This function can be accessed in the top right corner under the options button. Every database created within DAB has a prefix. At the time of writing this prefix is “dab_”. In some cases a database is still active on the server, though the application DAB does not know anything about them. These databases can be deleted individually or all at once. The popup presented to the user after clicking the “View/edit ghost databases” in the options menu will look similar to the one below.



FAQ

What is DAB?

DAB is a database manager. Students can create a database for a course which is created by the teacher. The student can study the provided material in the course and can use the database as described by the teacher. By the use of the database the student will gain a deeper understanding of a database and how to use them.

Can any email address be used?

No, only an email provided by the university can be used in conjunction with DAB

Can I use the database for a production environment?

No, the database is only meant to be used inside of a course given at the University. A student is allowed to use a database for a personal project, however this is not advised. An administrator or teacher is able to delete any stored data at any given moment in time.

Who can use DAB?

DAB is meant for students and university personnel at the University of Twente.

I am no longer able to access my database and the database is no longer visible inside of DAB, what can I do?

Due to the lifecycle of a course a teacher can decide to delete the course including all the databases. Depending on the teacher the database may still be inside of a backup. If this is the case the teacher has the ability to put back the database (including all its content). Please, ask your teacher of the course in question for more information.

The database was not accessible during the night, but during the day it seems to work again. How is this possible?

*The server containing the database may be doing some administrative tasks during the night. Therefore it is advisable to **not depend on the database during the night**. Therefore, students should not make last minute alteration during the night.*

I am a student, how can I sign up?

*A student can access the site directly. On the site a “register here” button is provided to create a new account. The site will guide you through the process. **Only an email address provided by the university is allowed.***

I am a teacher at the University of Twente and would like to use DAB for my students, but don't have an account. how can I sign up?

There are 2 ways to obtain a teacher account within the DAB application. The first is to contact an administrator of DAB directly asking for an account (a valid University of Twente mail account is required). The second option is to create a student account on the URL where DAB is hosted and request teacher privileges from one of the DAB administrators (either by mail or in person).

Beside database management does DAB provide management over other pieces of software?

At this moment in time DAB only supports database management. Other pieces of software may be added in the future.

What type(s) of relational database management systems (RDMS) does DAB support?

*Right now only **PostgreSQL** is supported. Other types may be added in the future.*

What devices does DAB support?

DAB is a web application that can be used on both desktops and mobile phones. The website will automatically detect the type of device and scale to the screen size.

Documentation REST calls

This document gives a complete overview of all the REST calls implemented in the DAB (Database Administration Bundle) application. Every call will start with a /rest/ unless specified otherwise. Depending on the server that DAB is running there will be a hostname before /rest/. To keep this document as general as possible no hostname is included.

Some calls do require some input in the body. When this is not specified the server will ignore any input in the body and thus can be omitted. If however a body is required the body must be of type JSON.

DAB uses 3 different roles to identify privileges for a user. Each role has a specific value. An admin has value 0, a teacher has value 1 and a student has value 2. These values are used in some calls below.

Responses

Depending on the call there can be a number of different responses. When an error or successful action occurs the server will give back a HTTP response status code. These codes are listed below.

In general

```
RESPONSE:  Dont touch that:401 [you have no authorisation to do that action]
           Any other case: 405
```

Call specific

GET

```
RESPONSE:
  Succes:      data
  Otherwise:   404
```

PUT

```
RESPONSE:
  Success:      202
  bad request:  400 [is the JSON object correct?]
```

POST

```
RESPONSE:
  Succes:      201
  bad request:  400 [Did you provide the correct fields?]
  DB del err:   500 [ONLY with studentdatabases-> not good,report]
  duplicate key: 409 [This combination already exists]
  other db err: 406
```

DELETE

```
RESPONSE:
  Succes:      202
  Otherwise:    404 [Probably the pk is not in the db]
  DB del err:   500 [ONLY with studentdatabases-> not good,report]
```



```
protected db: 409 [deleting a user with existing db's]
other db err: 406
```

If in any situation a 500 error is returned the server has encountered an internal error. If this occurs, please report this instance to the administrator.

/set_role

CALL: POST

PERMISSIONS GRANTED: admin

Sets the role of a user. Unless one is admin, one can only set the role of someone with a higher role than oneself, and to a role higher to oneself.

Body

```
body:
{
  "user": 3, [USER ID]
  "role": 0 [ROLE TO BE SET TO]
}
```

/whoami

CALL: GET

PERMISSIONS GRANTED: every role

Returns a JSON object with the id, email, and role of the user. Useful for debugging and unit testing. If the user is not logged in, returns a 401.

/who

CALL: GET

PERMISSIONS GRANTED: every user

Same as whoami, but only return id and role. Because there is no (explicit) database query, this request is a bit faster

/dump/

CALL: GET

PERMISSIONS GRANTED: Admin (any case). student(only its own databases). Teacher (its own databases and student databases)

Returns a sql dump of the database corresponding to the specified id. Note that the content type is application/sql and that most browsers will see it as a file to be downloaded; a simple link with `target="_blank"` would suffice for the front-end.

/reset/

CALL: POST

PERMISSIONS GRANTED: Admin (any case). student(only its own databases). Teacher (its own databases and student databases)

Resets the table to the original schema from the course.

/missing_databases

CALL: GET

PERMISSIONS GRANTED: admin

GET -> A JSON array of databases that the system does not know about, starting with the prefix specified in settings.py

/missing_databases/all

CALL: GET,DELETE

PERMISSIONS GRANTED: admin

GET -> a JSON array of ALL databases not managed by DAB

DELETE -> gets a JSON array of database names. DAB will drop all databases in the array, provided they are not managed

TABLE: studentdatabases

/studentdatabases/

CALL: GET,POST

PERMISSIONS GRANTED: only admin

GET -> get info all users

POST -> add user

```
body:
{
  "fid":"5", [FOREIGN KEY, MUST EXIST. Optional, if not specified, your current
user id]
  "course":"3", [FOREIGN KEY, MUST EXIST]
  "groupid":"4" [FREE TO CHOOSE, responsibility is for the student as he is the
person who can change it]
}
```

Note: on success you will get the following object back:

```
body:
{
  "dbid":<GENERATED VALUE, used as primary key>
  "fid":<your value>
  "course":<your value>
  "databasename":<GENERATED VALUE>
  "username":<GENERATED VALUE (same as databasename)>
  "password":<GENERATED VALUE (long)>
}
```

Note that the student will want to know the generated values

/studentdatabases/pk

CALL: GET,DELETE

PERMISSIONS GRANTED: admin (any case).Teacher(if the database belongs to the teacher or from a student in his course). Student(only its own databases)

GET -> get database for that pk

DELETE -> delete database for that pk

/studentdatabases/name/value

CALL: GET

PERMISSIONS GRANTED: admin (any case).Teacher(if the database belongs to the teacher or from a student in his course). Student(only its own databases)

GET -> search on the name of the database. The value will be taken as the search field

/studentdatabases/own/

CALL: GET

PERMISSIONS GRANTED: every user

GET -> gives back all the studentdatabases owned by the user currently logged in

/studentdatabases/owner/value

CALL: GET

PERMISSIONS GRANTED: admin (every case). Teacher(if the student is in his course)

GET -> gives back all the studentdatabases owned by the user with the id of the value

/studentdatabases/course/value

CALL: GET

PERMISSIONS GRANTED: admin(any case). Teacher(if the course belongs to the teacher)

GET -> gives back all the studentdatabases belonging to this specific course

/studentdatabases/teacher/own

CALL: GET

PERMISSIONS GRANTED: admin and teacher

GET -> gives back all the studentdatabases owned by the teacher

TABLE: Courses

/courses/

CALL: GET,POST

PERMISSIONS GRANTED: every user

GET -> get all courses

NB: to save data, schemas are not mentioned in GET!

POST -> add a new course

body:

```
{
  "coursename": "test20", [FREE TO CHOOSE]
  "students": "2", [FREE TO CHOOSE]
  "info": "test200", [FREE TO CHOOSE]
  "fid": "7" [FOREIGN KEY, MUST EXIST; OPTIONAL, defaults to own]
  "schema": <sql> [OPTIONAL, DEFAULT=""]
  "active": [BOOLEAN, OPTIONAL, DEFAULT=false]
}
```

/courses/pk

CALL: GET,PUT,DELETE

GET -> get course for this specific course id

PERMISSIONS GRANTED: every user

PUT -> update information (updating the fid or courseid is not allowed)

PERMISSIONS GRANTED: admin (any case). Teacher(if the course belongs to the teacher). TA(if the TA belongs to the courses)

DELETE -> delete a course for a specific course id

PERMISSIONS GRANTED: admin (any case). Teacher (only if the course belongs to the teacher)

/courses/name/value

CALL: GET

PERMISSIONS GRANTED: every user

GET -> search for the value field, based on the course name

/courses/own/

CALL: GET

PERMISSIONS GRANTED: admin and teacher

GET -> gives back all the courses owned by the currently logged in user

/courses/pk/schema

CALL: GET,POST

GET -> returns the schema for that database as a sql file

PERMISSIONS GRANTED: every user

POST -> Takes the **plaintext** body, and makes it the schema of the database (if it passes verification).

PERMISSIONS GRANTED: admin(any case). Teacher(if the course belongs to the teacher). TA(if the TA belongs to the courses)

TABLE: dbmusers

/dbmusers/

CALL: GET,POST

GET -> get all users

PERMISSIONS GRANTED: admin only

POST -> add a new user

PERMISSIONS GRANTED: every user, though by default role is set to 2. only an admin can set the role.

body:

```
{
  "role": "0", [0=admin,1=teacher,2=student] [set to 2 if not included]
  "email": "asdfasdf2", [FREE TO CHOOSE, THOUGH NO DUPLICATE IN TABLE]
  "password": "test205", [FREE TO CHOOSE, IS HASHED]
}
```

/dbmusers/pk

CALL: GET,DELETE

GET -> get user for that user id

PERMISSIONS GRANTED: admin(any case). Teacher(of his course).TA(if the TA belongs to the courses)

DELETE -> delete user for that user id

PERMISSIONS GRANTED: admin(only)

/dbmusers/email/value

CALL: GET

PERMISSIONS GRANTED: admin and teacher

GET -> search for the value, based on emailaddress

/dbmusers/own/

CALL: GET

PERMISSIONS GRANTED: every user

GET -> gives back the info about the currently logged in user

/dbmusers/course/value

CALL: GET

PERMISSIONS GRANTED: admin(any case). Teacher(if the course belongs to the teacher).TA(if the TA belongs to the course).

GET -> gives back the users that have a database in that course

TABLE: TAs

/tas/

CALL: GET,POST

PERMISSIONS GRANTED: admin(any case). Teacher(for addition any case, for deletion only his own course)

GET -> get all tas

POST -> add a new ta

body:

```
{
  "courseid":"8",  [FOREIGN KEY, MUST EXIST]
  "studentid":"16" [FOREIGN KEY, MUST EXIST]
}
```

/tas/pk

CALL: GET,DELETE

PERMISSIONS GRANTED: admin(any case). Teacher(if the TA belongs to the course)

GET -> get ta for that ta id

DELETE -> delete ta for that ta id

/tas/teacher/own/

CALL: GET

PERMISSIONS GRANTED: admin and teacher

GET -> returns all the tas in the courses owned by this teacher

/tas/own/

CALL: GET

PERMISSIONS GRANTED: admin and teacher

GET -> gives back the ta information about the currently logged in ta

/tas/course/courseid

CALL: GET

PERMISSIONS GRANTED: admin(every case).Teacher(if the id belongs to a course owned by this teacher)

GET -> gives back the tas of that course

/schematransfer/[course]/[database]

CALL: POST

PERMISSIONS GRANTED: admin only

Transfers the schema from the database to the course.

Only works if the database belongs to the course or to your user personally, unless you are an admin.

Default schema (named after the user) is preserved from the target database

/generate_migration

CALL: POST

PERMISSIONS GRANTED: admin only

POST -> Generates the backup script. Returns the location of said script.

/course_dump/courseid

CALL: GET

PERMISSIONS GRANTED: admin(every case). Teacher(if this course belongs to the teacher)

GET -> returns a zip file with dumps of every database in the course. Dump filenames are named after the user email.

WARNING: May take a long time. Use with caution.

/request_reset_password/[email] (NOT /rest !!!)

CALL: POST

PERMISSIONS GRANTED: every user

POST -> Sends an password reset email

Email contains a link that is valid for 4 hours.

/reset_password/[user id]/[token] (NOT /rest !!!)

CALL: GET,POST

PERMISSIONS GRANTED: every user

GET -> Displays a "new password" prompt

POST -> Takes in a JSON object with one key, "password", and sets the password to that password.

Both of these will return errors if the token is invalid or expired.

The link sent will be valid for 4 hours.

/change_password/ (NOT /rest !!!)

PERMISSIONS GRANTED: every user

CALL: POST

POST -> Takes in the following JSON

```
{
  "current": "aoeu", [CURRENT USER PASSWORD]
  "new": "ueoa" [NEW PASSWORD]
}
```

and sets the password of the user to the new password, if the current one is correct.