FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Improving LearnShell backend for analytics
Student:	Dan Pejchar
Supervisor:	Ing. Jakub Žitný
Study program:	Informatics
Branch / specialization:	Information Systems and Management
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

LearnShell is a modular system for managing and performing exams with programming assignments in scripting languages, especially Shell. LearnShell currently offers basic functionality for creating assignments and exams.

Improve the performance and functionality of analytics modules of the LearnShell backend.

1. Analyze the current architecture of the LearnShell backend and APIs.
- Identify the parts that need improvements
2. Propose improvements for analytics module backend functionality.
- Analyze the time complexity of this task
- Identify the rabbit-holes and no-gos
3. Optimize performance of all methods in analytics modules.
4. Compare the pricing and pros and cons of 3 major cloud providers where the LearnShell backend with analytics module could be migrated in the future.
5. Compile a report of improvements, future re-evaluation, and internal documentation.

Electronically approved by Ing. David Buchtela, Ph.D. on 7 March 2021 in Prague.

Bachelor’s thesis

Improving LearnShell backend for analytics

Dan Pejchar

Department of software engineering
Supervisor: Ing. Jakub Žitný

May 13, 2021

Acknowledgements

I would like to thank my thesis supervisor Ing. Jakub Žitný for helping with the whole thesis-creating process, giving me tips and mentoring. I would also like to thank the teachers Ing. Dana Vynikarová, Ph. D. and Ing. Ondřej Guth, Ph. D. for helping with the technical aspect of the thesis and leading me throughout the whole process. Finally I would like to thank all the teachers that have taught me throughout the whole bachelor studies and my family and friends for support.

Declaration

In Prague on May 13, 2021 …………………

Citation of this thesis

Pejchar, Dan. *Improving LearnShell backend for analytics.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

V této bakalářské práci pracuji na vylepšování aplikace LearnShell 2.0. Mým hlavním cílem je přidat do aplikace analytický modul. Tento modul vypočítává statistická data, která mohou následně býti vyobrazena ve webové aplikaci. Další částí této práce je finanční analýza poskytovatelů cloudových služeb vhodných pro migraci LearnShellu v budoucnu. Hlavním přínosem je zlepšení LearnShellu pro studenty resp. učitele, kteří LearnShell používají, díky kterému pro ně bude jednodušší studovat resp. učit.

Klíčová slova LearnShell, Analytický modul, Finanční analýza, Django, Python, PostgreSQL, GraphQL, Celery, Docker, výuka na ČVUT, backend

Abstract

In this bachelor's thesis my main focus is on improving the application: LearnShell 2.0. My main goal is to add an analytics module to the application. This module calculates statistical data which can then be visualised in the web application. Another part of my thesis is a financial analysis of cloud service providers ideal for migrating LearnShell to, in the future. My contribution is making LearnShell better for students resp. teachers who use this application – making it more convenient to teach resp. study.

Keywords LearnShell, Analytics Module, Financial analysis, Django, Python, PostgreSQL, GraphQL, Celery, Docker, teaching at CTU, backend

Contents

List of Figures

List of Tables

List of Listings

Introduction

In this bachelor's thesis I will be improving the web application: LearnShell 2.0. Learnshell is an application developed by several students at the Czech Technical University – Faculty of Information Technology.

First I would like to mention the subject BI-PS1 (Programming in Shell 1). BI-PS1 is a mandatory subject for all students in the first semester of FIT CTU. In this subject they learn the basics of programming in Shell (hence the name LearnShell). Throughout the semester, students take a very simple (approx. 5 minutes) test at the beginning of every lesson regarding the topics of the previous lesson.

That is where LearnShell comes in. Every year this subject has about 700 to 800 students. Correcting up to 800 exams every week by humans would be very time-consuming and probably impossible. LearnShell makes this possible. It can automatically generate assignments for students and correct their solutions within seconds.

In the assignments, students are asked to design a basic Shell script (often just one command). Then they enter it into a text-box and submit. The system corrects it immediately and gives the student a result. The student typically has a time limit within which the solution can be altered and submitted to the system repeatedly.

I have picked this topic because I think LearnShell is a very useful application for FIT students and teachers and could be even more useful with some changes made, perhaps even to students and teachers of other faculties or universities. LearnShell is a work in progress, different students take part in the development and there are always more things to add/improve. Thanks to LearnShell, students of BI-PS1 are forced to at least open the studying materials before every lesson. They do not need to study it intensively, just to keep up with the subject matter.

My main goal is to add an analytics module to the LearnShell backend, which will gather statistical data, so that students can view their personal data and the data of their course. The module will be prepared for other additions in the future. One of my colleagues is working on the frontend – the web page itself and I will prepare the data that will be visualised on the web interface.

This improvement will be very useful for both the students and the teachers. The students will be able to see clearly their progress in the subject. They will be able to decide how well they are doing and whether they need to work harder.

Apart from the analytics module, I will complete a financial analysis of 3 major cloud service providers, comparing their pricing and other pros and cons. I will choose an ideal cloud service provider to which LearnShell, including the new analytics module, could be migrated to and probably will be in the future.

In the first chapter I will be defining the goals of my thesis. The next chapter will be the theoretical part, where I will explain various technologies used in LearnShell on a general level citing their official documentations. This will lead to my practical part, in which I will analyse the current architecture of the LearnShell backend, then I will propose improvements and design them on a conceptual level. After that, I will describe the process of the implementation and finally I will reach the financial analysis. In the last chapter I will conclude a report of the carried-out improvements, select a way to re-evaluate the new implementation in the future and create a documentation for the newly implemented improvements.

Chapter 1

Goals

My goals are divided into five. Some are more theoretical and some more practical. However, they all have one common goal: to improve LearnShell.

In the first part of my thesis, I will be analysing the current architecture of the backend and APIs. In my analysis I will identify which parts need improving. After the analysis is finished, I will propose precise improvements/optimisations, that I will carry out. In the analysis, I will also analyse the time complexity of the proposed improvements. Then, I am going to identify the rabbit-holes and no-goes – mainly point out some ways of improving, which would not be effective and lead to a failure / make the program worse. After that, I will actually carry out the improvements and describe the whole process.

LearnShell is currently running on FIT local servers so as a next goal, I will chose 3 major cloud service providers, compare them – their pricing, pros and cons – and select the most suitable one, to which LearnShell could be migrated in the future. Finally I will summarise all executed improvements, prepare them for future re-evaluation and create a documentation about them where I will describe exactly how they work.

Chapter 2

Theoretical part

This chapter is the theoretical part of my thesis. I will not explain any of my practical work, I will only describe the technology and literature I will be using in my practical part on a theoretical and general level.

2.1 Literature

Before I start describing the technology I would like to write a few words about the used sources. The largest part of my thesis is the actual implementing of the new code. It is not a purely theoretical thesis so there was no need to study any computer science paradigms on a theoretical level. I will mostly be using official documentation of the technology in place, which will be cited wherever I will be using it. I will also be using some internet articles and studies/researches.

In my pricing analysis I will use some internet articles about, which cloud service providers are the most popular and then use their official websites price lists, documentations of their services and online calculators to calculate the long term pricing.

2.2 Technology in LearnShell

In this section, I will be describing the technology (used in LearnShell backend at the moment and which I will be adding as a part of my work) generally. I will not go into detail on how it functions in LearnShell or in which way I have decided to use it. In the practical part, I will not be explaining how the technology functions anymore. I will assume that all this information was already acquired in this section.

2.2.1 Docker

What is Docker? According to [1] "*Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and deploy it as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customised settings that machine might have that could differ from the machine used for writing and testing the code.*" In other words, Docker makes it possible for anyone with a Linux computer to download the repositories of a dockerised application, install and run it by one command which may look like this: `sudo docker-compose up --build`. Docker installs all the necessary libraries, technologies and application packages and then all the vital services run in the Docker container.

2.2.2 Celery

Now, let's explain, what Celery is. According to [2] "*Celery is an asynchronous task queue/job queue based on distributed message passing. It is focused on real-time operation but supports scheduling as well.*"

With web applications of larger scales such as LearnShell we absolutely need asynchronous tasks. Without asynchronous tasks/jobs it would basically be impossible for more than one user to connect to the application at once. We need the server to work asynchronously for each user at the same time with the same computing capacity.

Celery allows that. It creates separate threads for all users and their demanding tasks. Other than that, Celery can prove useful for scheduling tasks too. Such as sending emails periodically or on exact dates. Celery can even schedule tasks according to the sun, for example, it is possible to schedule a task on "every sundown".

2.2.3 Redis

What is Redis? Again, according to [3], its official website: "*Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams.*"

It is mainly used as a message broker between Django and Celery. Celery does not communicate straight with Django. There is no need to explain Redis more, as I will not be using it directly.

2.2.4 PostgreSQL and Django

First I would like to mention what Django and PostgreSQL are, and then explain how they work together. According to the official Django documentation website [4]: "*Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. [. . .] It's free and open source.*" LearnShell backend is implemented in Django.

Django projects consist of so-called *Django applications*. Those applications contain so-called *models*. The whole project is designed object oriented. These Django models are classes which handle jobs which the specific class is purposed for but also represent a table in the PostgreSQL database which stores data belonging to the class persistently.

And what is PostgreSQL? Once again, I will start with their own definition: "*PostgreSQL: The World's Most Advanced Open Source Relational Database*" [5].

PostgreSQL is the relation database Django generates. Django models represent an object oriented class and a database table at the same time.

These models in a *Django application* have something in common. For example we could have an application *home* which would contain the models: *House* and *Address*. The *House* would represent a building and contain data about it such as *number of rooms* and a one-to-one relation with the *address* which would be a table containing the address of a house in separate fields but they clearly belong together in one module.

I will explain how Django and PostgreSQL work together on an example. Let's say we create a Django model *Course* with a string *name*, an *id* and a *foreign key* of another model *CourseStat* and we want it to be in a relation 1:1 with this other model. The code would look roughly as shown in example (listing 2.1). This is not supposed to be a working code or a part of the actual code from LearnShell, it is just and example I made up, to show, how it could look.

After we would create a code like this, we would have to generate Django migrations. These migrations are files with commands that edit the PostgreSQL database. For this specific example, the migrations would create a new table (figure 2.1) in the database, with 3 columns:

id — an automatically generated primary key

name — a string with the name of the course

course_stat — a reference to another model of this type

One may be wondering where did the id *AutoField* come from. Every Django model inherits from a `models.Model` class which automatically creates an `id` field which is the primary key of the database table and also has some sort of automatic increment mechanism, unless we explicitly make the primary key something else, like for example a *ForeignKey* in a 1:1 relation.

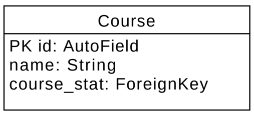

Figure 2.1: Course example table

```python
from django.db import models
from django.utils.translation import ugettext_lazy as _


class Course(models.Model):
    name = models.CharField(
        max_length=50,
        verbose_name=_("Course name"),
    )
    course_stat = models.OneToOneField(
        CourseStat,
        on_delete=models.CASCADE,
    )
```

Listing 2.1: Example of Django model

2.2.5 Python

Since Django is implemented in Python, I would also like to write a few words about it. If you know something about programming, you most definitely know Python, or have at least heard about it, because at the moment (5. 5. 2021) it is the most popular programming language and has been for a very long time [6].

In my opinion it is due to its readability and overall, it is easy-to-use. You do not even have to consider data types like in almost every other major programming language. Anyone who has programmed before in a different language can relatively easily switch to Python.

Although the Django models architecture is object-oriented and Python supports classes, it is not truly an object oriented programming language. There is no such thing as *private*, *public* or *protected*. All class methods and attributes are *public* and you can't change that. It is not even possible to define constants with a keyword `const` like e. g. in C++. When you need a constant, you have to make it a regular variable and never change it.

It is an interpreted language and quite slow compared to e. g. C++ and most compiled languages. (Based on an independent programming language benchmark website [7] and on my overall experience.) Fortunately, there are packages like NumPy or SciPy which are implemented in C and compiled with a C compiler. Their functions are basically wrappers for C implemented functions. As a result, these packages work a lot faster than pure Python would [8] which can prove useful under specific circumstances.

2.2.6 GraphQL and Django Describer

Finally let's explain GraphQL. GraphQL is a query language for any API. It is not tied to any specific database or storage engine but instead is backed by some existing data and code.

The user has to only define types and fields of those types and functions [9]. For generating the GraphQL API we can use some generators like Django Describer. Django Describer is a tool developed by another FIT student. In its official documentation is stated: [10]: "*An easy-to-use tool to auto-generate GraphQL API from Django models.*".

When we have a running GraphQL API we can query and mutate data from the database for which the GraphQL was generated. This can be done manually in a web service, or we can use for example some frontend to query through the GraphQL.

Chapter 3

Practical part

This is the part of the thesis, where all the practical work is done. First, I will describe the current architecture and the main technologies it uses. While describing the current architecture, I will identify some parts that need improvements. Then I will propose some improvements in the analytics module, analyse the time-complexity of that task and identify the rabbit holes and no-goes.

After proposing the improvements I will implement them into the current LearnShell backend and comment the whole implementation process in section 3.3.

Then I will select a cloud service provider to which LearnShell could migrate in the future, based on a pricing analysis.

Finally I will report all the implemented improvements, create a documentation of the newly implemented parts and select a tool way of future re-evaluation, which could analyse how much the users actually use the new functionalities.

3.1 Current architecture analysis

Now let's explain, how LearnShell works at the moment. LearnShell is divided into 4 main basic parts described on figure 3.1. These parts work as separate services that are capable of working on their own. For instance, you can do everything with the backend that you could do through the frontend web interface, without running it at all. The whole architecture is viewable on figure 3.2.

In my thesis I will be working mainly on the LS core, so here, I will focus mainly on the backend. I may mention the frontend a little bit, but I will not explain how the generator and evaluator works. It is not a part of my assignment.

Figure 3.1: LearnShell basic parts

LearnShell core (backend): connects all the other services together. It manages a database of all the necessary entities.

Generator (backend): is specific for PS1. It generates exams for specific students by the given template written in a special LearnShell language.

Evaluator (backend): is also specific for PS1. It corrects the solution given by the student automatically.

LearnShell web (frontend): is a web interface, which communicates with the LearnShell core and visualises the data in a user friendly manner.

Let's explain how the LearShell core works. How different technologies work on a generall level has been explained in section 2.2. Now I will describe how they are used specificaly in Learnshell.

LearnShell runs in a Docker container which can be built and ran by this command: `docker-compose up --build`. The LearnShell Docker runs the following services:

- ls-redis
- ls-postgres
- ls-celery
- ls-backend
- ls-generator-ps1

which all run on the *ls-bridge* network.

With a web application such as LearnShell, we absolutely need asynchronous tasks. To accomplish this, we use Celery. Celery does not communicate with Django directly. For that we have Redis, which works as a message broker between Celery and Django. Celery is used for example when a class (parallel) is writing an exam. Every time a student submits an assignment, LearnShell has to correct it immediately and give him/her a result. The LS core sends the students' solution to the *Evaluator* service which corrects it and returns some correction data in JSON format which contain for instance the score obtained from the given assignment.

Imagine if these tasks were running synchronously. The student who submitted the submission last would have to wait until the system corrects all the submissions before him. Also, if someone else would be working on the server,

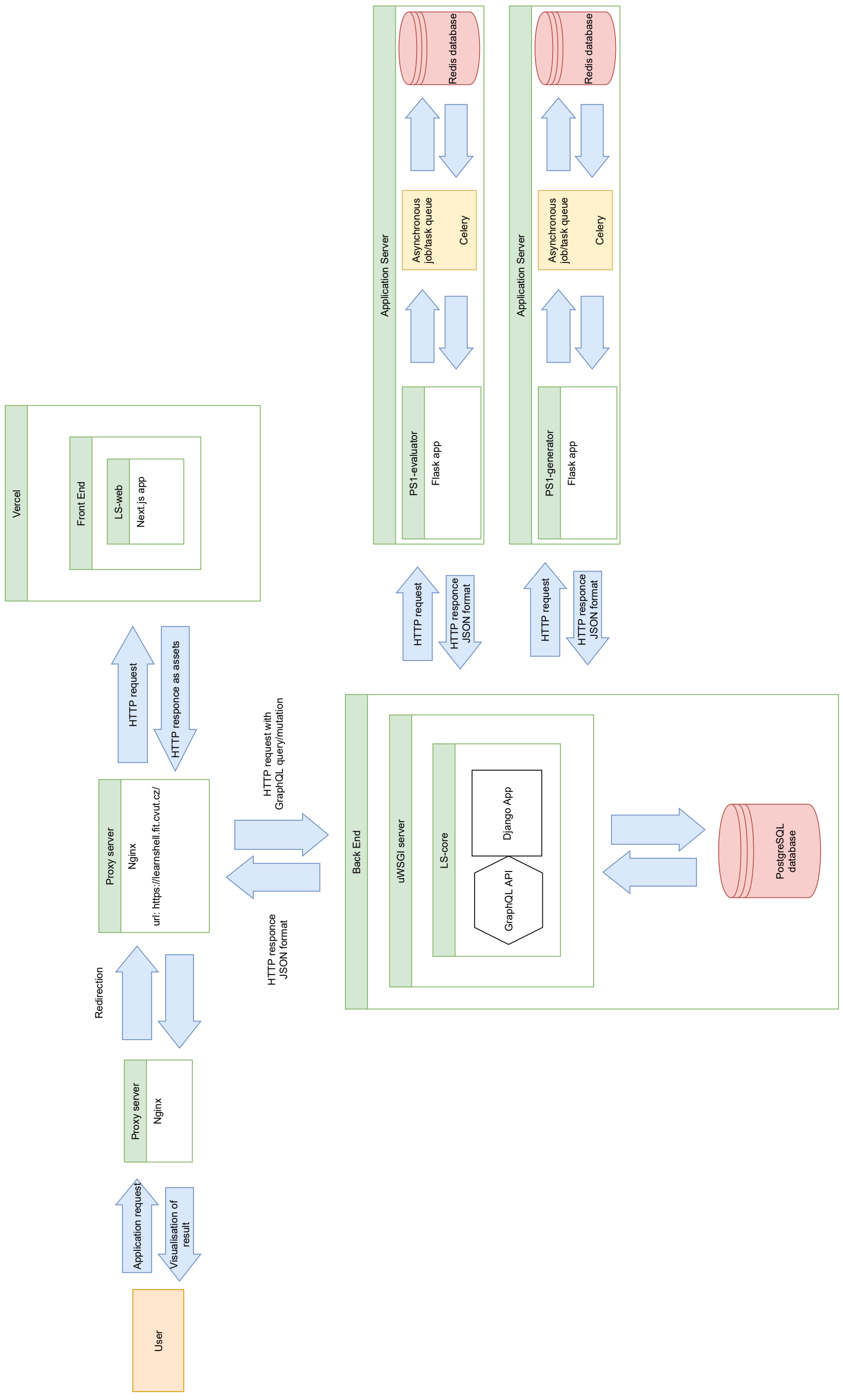

Figure 3.2: LearnShell architecture

```
class UserLevel:
    STUDENT = 1
    TEACHER = 2
```

Listing 3.1: Enumeration of *UserLevel*

running some demanding task, the students could spend the rest of the lesson waiting for their solution.

As written in the theoretical chapter 2.2, Celery also supports scheduling, which could prove very useful. This means that even though LearnShell does not have any scheduled tasks at the moment, it is very well prepared for them and it would be very easy to add a task like that.

Although the general functionality of Django was explained in the theoretical chapter 2.2, I would like to describe it again on a specific part of the LearnShell Django application.

In the following text I will sometimes be using the term *student* with which I will be refering to a model in the database *User*. The *Course* has a 1:N relation with the *Parallel*. The *Parallel* contains data about a weekly lesson taught on its *Course*. The data contain e. g. the time of week when it is taught or the room. For example there could exist a *Parallel* of one specific *Course* taught on the third day of the week (Wednesday), on the third hour of the week (from 11:00am to 12:30pm). The *ParallelMembership* is a class connecting together a specific student with a specific parallel. A *Parallel* has many students but a student typically has only one *ParallelMembership* although it is not restricted in the database. The membership could also belong to a teacher and a teacher could very easily have many *Parallels*. This information is stored as a property *level* which contains an enumeration for *STUDENT* or *TEACHER* created in the *ls/utils.py* file shown in listing 3.1. In the database it is stored purely as an integer. The *ParallelMembership* is a decomposition of an M:N relation between a *User* and a *Parallel*.

Information like this about the current architecture of the Django application can be found in figure 3.3. It is a full database scheme of the current architecture. By current I mean January 2021 when my analysis began. After this semester, when my work (and the work of my colleagues) is done, the database scheme will have changed.

Although we have the figure 3.3 I will go more into depth about how the parts, which I will be mostly needing and editing, work. The applications I will write about are: *assignment, generatedAssignment, submission* and *user*. You can see how they are connected together and which data they store in figure 3.4. It is basically a part of figure 3.3 with the previously mentioned applications.

Let's describe the process of students writing assignments. First someone (a teacher) needs to create the assignment. The teacher will most likely be

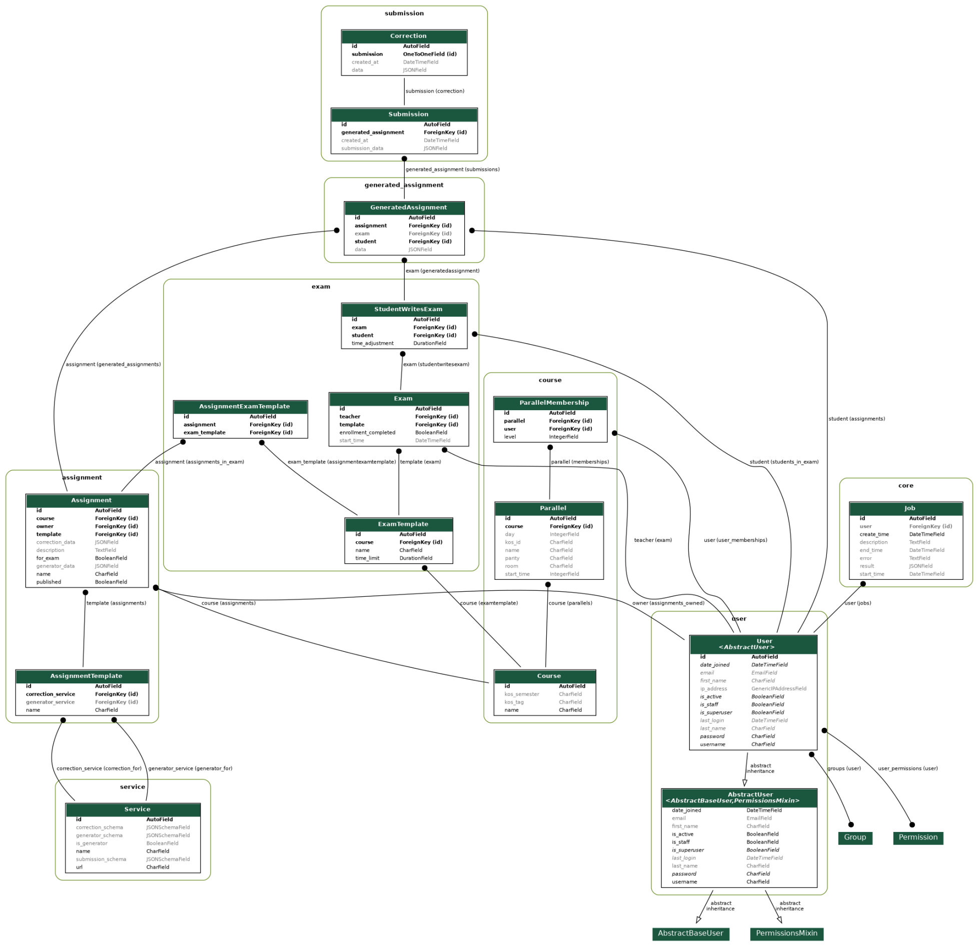

Figure 3.3: Database schema of the current architecture

creating the Assignment in the web interface. The web application will request a mutation via GraphQL which will add a new row in the *Assignment* table. The *Assignment* will contain some generator data in the form of a JSON. Then, the generator service (from figure 3.1) generates a *GeneratedAssignment* specifically for each student in the given *Course*. The *GeneratedAssignment* is in a 1:N relation with the *Submission* table. The student can create many submissions. When the student submits a new solution, the submission data – also stored in the *Submission* table as a JSON – are sent to the Evaluator service 3.1. The Evaluator returns some correction data which are stored in the *Correction* table. The *Correction* table is related to a *Submission* in a 1:1 relation and contains a property *score* which is retrieved form the JSON correction data. The *GeneratedAssignment* then contains the property *score* which is always on call calculated as the max score out all the *Corrections* related with the *Submission* which is related with the *GeneratedAssignment*.

Now I will explain how the Django project is then used to generate a GraphQL API which can be used by the frontend. GraphQL can be used seperately to query the backend API through a GraphQL interface (currently running on `localhost:8000/graphql`), but mainly, it can be used by the fronted to query data and view them on the webpage.

The GraphQL API is generated with *Django describer*. Let me remind the definition of Django Describer from the theoretical chapter: "*An easy-to-use tool to auto-generate GraphQL API from Django models.*"[10]. However I do not entirely agree with this statement. The tool does not have almost any documentation (apart from a brief readme file) which makes it quite complicated to use, for the users have to figure out by themselves how the tool actually works. In my opinion, this is the part of LearnShell that definitely needs improving. Django describer was created by another FIT student and is now obsolete. There are much better tools for this job. For instance Django Describer can't generate the API if the Django project contains a model without an auto-generated id as a primary key. That is possible in Django itself (e. g. using a foreign key as a primary key in a one-to-one relationship) but the describer can't generate the API like that.

Each model has a complementary *describer* class which contains information for the tool by which it creates the GraphQL API. Then you can access the GQL API through a web interface and query or mutate the data in the database. For instance you can list all the users and choose which data about them you want to list (e. g. username, first name, last name, etc.).

Now about the analytics module. The simple description of the analytics module is that there is no analytics module. The functionality of the analytics module is in the other modules. For example the overall score of each user is recalculated on call every time in the *User* model.

The data needed for analysing performance of students, parallels, courses, etc. are there, they just are not packed into one organised package in one place and there are not any methods to calculate statistical data like summations, medians, percentiles... One would have to calculate them manually by listing the raw data in the GQL web interface.

3.2 Proposing improvements

The improvement I will be working on is creating an analytics module which LearnShell needs. This improvement is in my bachelor's thesis assignment, therefore I will carry it out. This section will be about the conceptual design, time complexity of the implementation and identifying the rabbit holes and no-goes. The next section 3.3 will be about the actual implementation process.

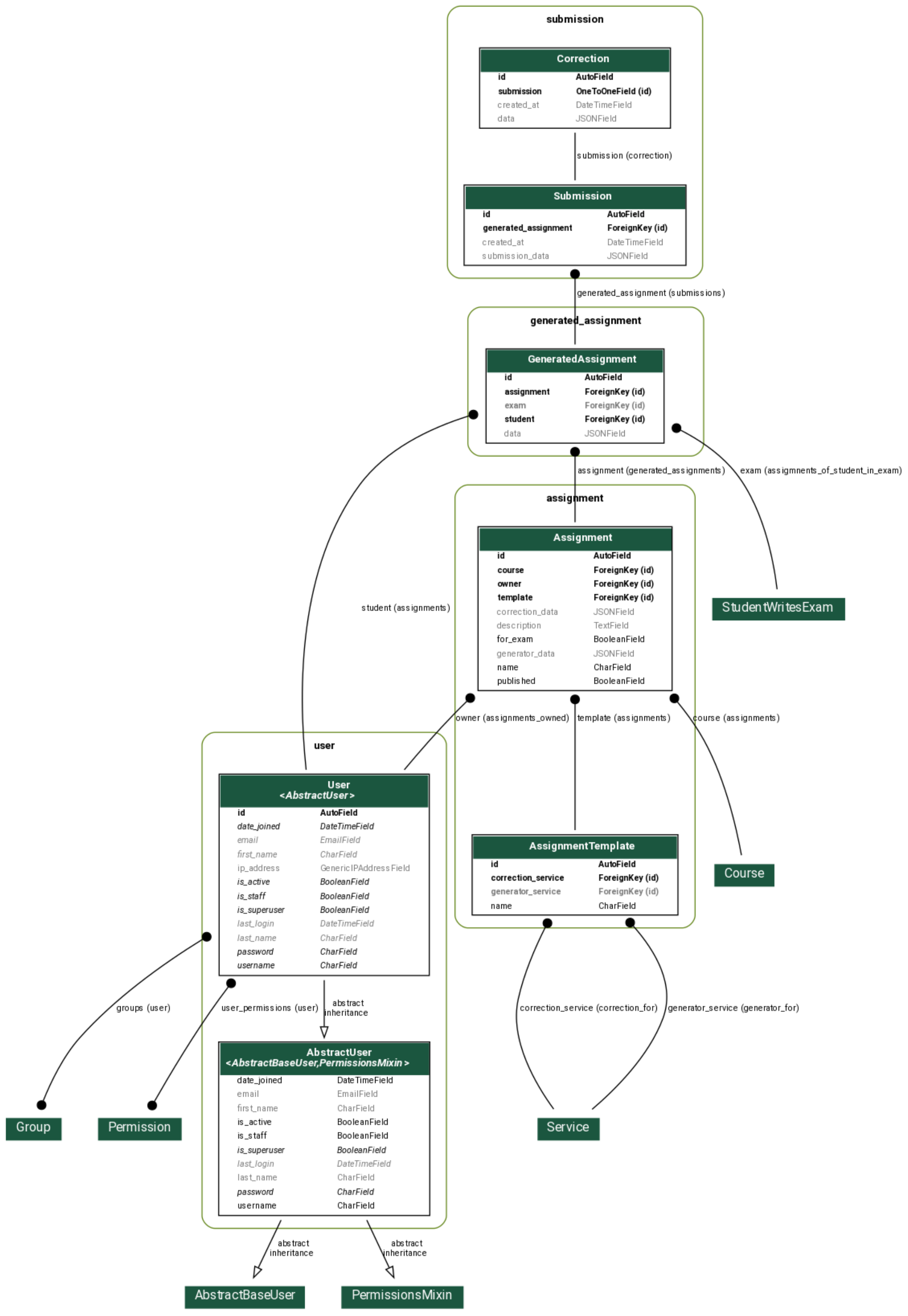

Figure 3.4: Part of database schema of the current architecture

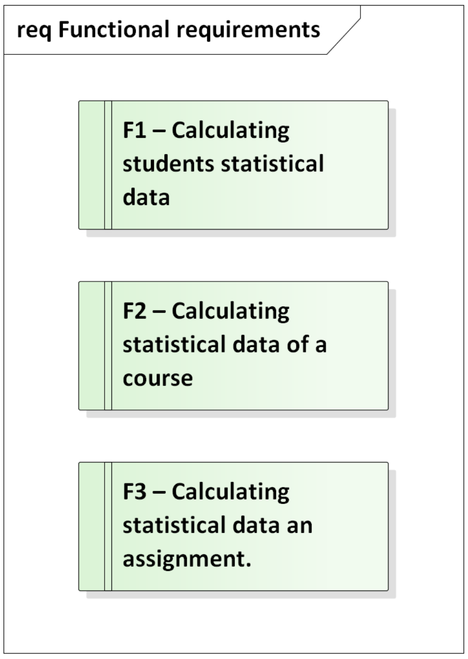
req Functional requirements
F1 – Calculating students statistical data
F2 – Calculating statistical data of a course
F3 – Calculating statistical data an assignment.

Figure 3.5: Functional requirements

3.2.1 Conceptual design

The goal of the analytics module is to calculate and store statistical data of all sorts. As a part of my thesis I will be creating the analytics module which will be able to respond to GraphQL queries. At the same time, there is a colleague working on the LearnShell frontend who is preparing a web page for visualising the data from the analytics module. Together, we will have to settle on some functional requirements. Then we will create some sample GraphQL queries which will fulfil the functional requirements. There are three basic functional requirements that can be seen on figure 3.5 and are described on figure 3.6.

Now, we will figure out how some sample GraphQL queries could look.

- Query 3.2 will return basic user stats.
- Query 3.3 will return stats of the course with id = 1
- Query 3.4 will return stats of assignment with id = 1

To support queries like this, it would be suitable to create classes for different kinds of statistical data which will all belong to one module (Django app) called *analytics*. The module is going calculate statistical data of students and courses. For this, I will create new classes *UserStat* and *CourseStat*, which will calculate statistical data of a specific student resp. course.

Figure 3.6: Functional requirements

Functional requirements

1. Calculating **students** statistical data

   Score The summation of scores from all submitted assignments.

   Score history An array of the history of these scores from each week.

   Percentile history An array of the percentile, of the students score at the end of the week compared to the other students scores.

   Score of assignment The students score of any given assignment

   Percentile from assignment The percentile obtained by a student from any given assignment

2. Calculating statistical data of a **course**

   Median The median of all students current score.

   Median history The history of the students median at the end of each week.

   Score histogram All the scores and their frequencies.

3. Calculating statistical data of an **assignment**

   Median The median of all scores obtained from an assignment

   Scores An array of all scores obtained from an assignment

   Max score The highest possible score from the given assignment.

```
query {
    UserMyself {
    id
    username
        userStats {
      results {
        score
        scoreHistory
        percentileHistory
      }
    }
  }
}
```

Listing 3.2: Basic user stats

```
query {
        CourseStatList(courseId: 1) {
    results {
      course {
        id
        kosTag
        kosSemester
      }
      median
      medianHistory
      scoreHistogram {
        results {
          score
          frequency
        }
      }
    }
  }
}
```

Listing 3.3: Basic course stats

```
query {
        AssignmentList(id: 1) {
    results {
      scores
      median
      maxScore
    }
  }
}
```

Listing 3.4: Basic assignment stats

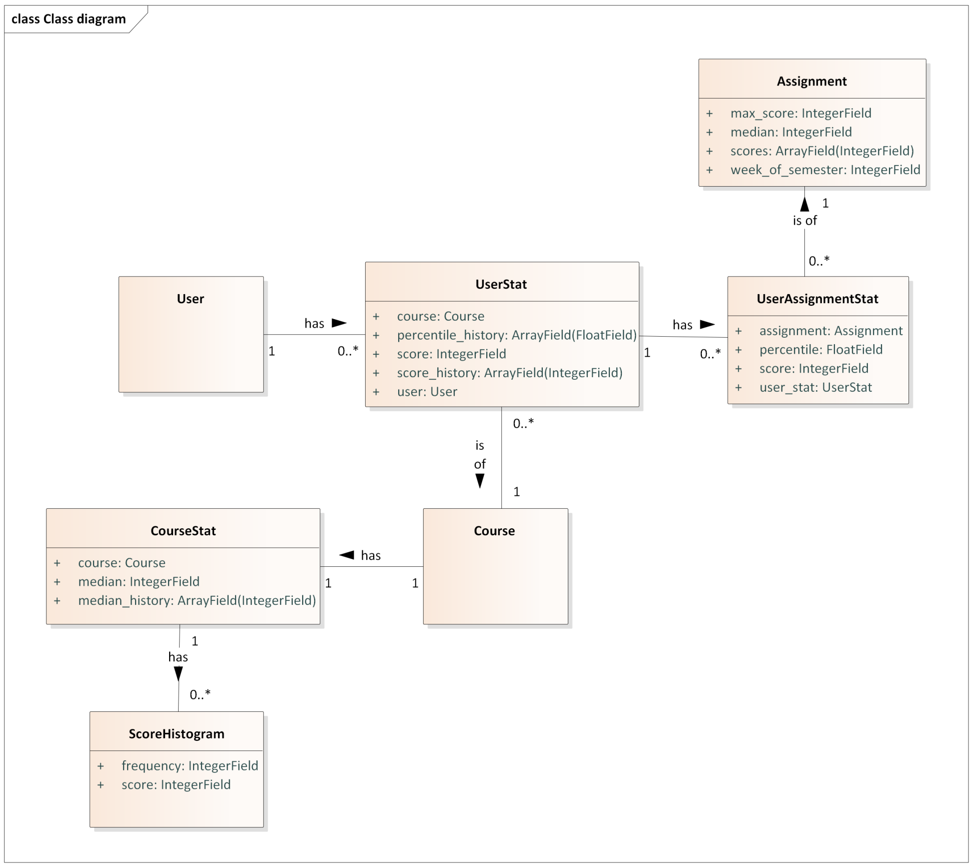

Figure 3.7: Class diagram

The main reason for this improvement is so that students will have the ability to track their progress throughout the semester. They will have the opportunity to compare their results with other students without actually revealing their identities, through analysing their percentile in different weeks. They will have a well-organised dashboard where they will have various graphs, charts and values to analyse.

It will also be possible to analyse the results of various courses. The students will be able to view the score histogram in their course. The whole conceptual design is on figure 3.7

The data of these tables will consist of data already described in the functional requirements 3.6. There will be need for methods for calculating the data, from data LearnShell already contains.

Before I continue, I will try to define some rabbit holes and no-goes to make sure I do not make a bad decision. One wrong way to create the analytics module would be to put the properties into the classes that are already there.

Let me elaborate. I mean adding properties like: score, median, percentile of a user directly into the *User* class. The *User* class would become too large and confusing. It makes much more sense to contain these data together in a separate class.

So I have decided to create new classes for the stats, but another mistake would be not to create a new module (Django app) but to put the newly proposed classes into other modules. These models should belong together into one module because they will have references to each other and also, it would not be clear, how to extend the analytics module. On the other hand, creating a true module will make LS backend prepared for analytics module extensions.

Another bad decision would be to create the *UserStat* table in a one-to-one relation with the *User.* The *UserStat* must be in an M:1 relation with the *User* because the user can have various sets of stats in various courses.

3.2.2 Defining property calculations

In this part, I will define how the proposed properties will be calculated. From the *UserStat* table I will need to define the calculation of: *percentileHistory, score, scoreHistory* and *AssignmentStat.* In the *CourseStat* table I will need to define the calculation of: *median, medianHistory* and *scoreHistogram.* In the *Assignment* I will define the calculation of: *maxScore, scores* and *median.*

All of these methods will be defined as properties of their stat table (class). That means, it will be recalculated on every query. These properties will not have any parameters and they will return the calculated stat.

Calculating the `UserStat.score` will be done by iterating through all the users assignments and adding the score from each assignment to the overall score.

The `UserStat.score_history` will also be calculated by iterating through all the users' assignments. This time, we will first need to sort the assignments by the week of semester and then add the score from the assignment to the position in the array (`score_history`) to the same position as is the week number of the assignment.

I had to add the *week of semester* attribute to the assignment table. Until now, the assignment did not contain any information about when it will be / was published to the students. Some assignments were named according to a convention [<week of semester> – name of assignment] but not all teachers respected it. So I decided to create a new column where the teachers have to define for which week of the semester the assignment is intended.

For calculating the `UserStat.percentile_history` the code will need to (for each week of semester) iterate through all the other `UserStats` with the same course, obtain their score of the given week from the `score_history` and add it to a temporary array. The percentile of the given week of the given user will then be calculated from his score in relation to all the other scores.

To create all the `UserAssignmentStats` for a `UserStat` the code will iterate through all the assignments and check, if the `UserAssignmentStat` of the given assignment already exists. If not, it will add a new `UserAssignmentStat` to the table.

The `Assignment.max_score` will be determined from the correction data of the assignment by summing up all the testcase scores.

Calculating the `Assignment.scores` will be done by saving all scores obtained from all generated assignments of the given assignment.

The `Assignment.median` will then be calculated from the `scores` by sorting them and picking the middle value, or calculating the mean of the two middle values (if there is an even amount of scores).

Calculating the `Course.median` will be done by iterating through all the `UserStats` of the course, saving them in a temporary array and then calculating the median of these values.

The `Course.median_history` will be calculated for each week and then saved into an array. In each week the program will iterate through all the `UserStats` of the given course, save the values of their `score_history` of the given week, calculate the median of these values and then save them into the `median_history` array of the given week.

For the `CourseStat.score_histogram` there is a new table. The table contains a score and its frequency. To calculate the `score_histogram` the code will have to iterate through all the students of the given course and in every iteration check whether a `ScoreHistogram` row with the students score already exists. If not, it will create a new row and set the frequency to 1. If the row already exists, it will just increment the frequency of the students score.

3.2.3 Time complexity

Now I will analyse the time complexity of my task. The most time was already taken by analysing the backend architecture and understanding it without almost any documentation. Some time was also taken by the conceptual design but this analysis will consider only the implementation process.

I will separate the task *implementing the analytics module* into smaller tasks and estimate their time complexity. It will be more convenient to estimate a time complexity of a small clearly defined task, than estimating the time complexity of everything in one task.

The first partial task will be setting up the analytics module. Then I will create the Django models I have decided to in the conceptual design. Then I will declare their properties (the statistical data). Next I will decide how the data will be calculated and implement the calculating methods. After that I will test the newly implemented methods overall. Then I will do some optimisations and finally fix errors or add other functionalities which were not proposed but I will perhaps come up with them along the process. After each

Table 3.1: Estimated time complexity

Task	**estimated time complexity [MD]**
Setting up the analytics module	2
Creating the Django models	5
Declaring properties	3
Calculating properties	10
Testing	3
Optimisations	5
Extra time fixing & improvements	4
Total:	**32**

sub-task will follow a short testing period to determine whether it works how I expect it to. These tasks and their estimated man-days can be inspected on table 3.1. The overall time complexity of the whole implementation process was estimated to 32 man-days.

3.3 Implementing the improvements

This section is going to be about explaining how exactly I created the codes and why. I will include some shorter code samples but not all of the code I have programmed. The whole and final code can be examined on my branch of the LearnShell repository on the faculty GitLab (11) (if there is access) or on the enclosed CD (C). Because LearnShell is a real deployed application uploaded on a version control system, I may add a commit after submitting this thesis. I will not be changing any major parts in the analytics module architecture, but there could be some minor bug fixes. That does not mean that the code on the enclosed CD contains any bugs, that I know of. I completely stand by the submitted code. The source code on the enclosed CD will be identical with the code of the latest commit before 13. 05. 2021 on my branch *bap-pejchdan*, but I recommend using the faculty GitLab (if there is access) because there might be a newer and better version of the code and there it is also possible to see all the contributions I made.

3.3.1 Creating the analytics module

First of all I have created the Django application *analytics* by running the Django command (listing 3.5) in the `apps` folder. This created a new directory `ls/apps/analytics` which contained the basic components of a Django application. The directory of a fresh Django app is in figure 3.8. All the

```
./../manage.py startapp analytics
```

Listing 3.5: Creating new Django application

Figure 3.8: Empty analytics module

```
 migrations ............................ directory with Django migratoins
 └── __init__.py .....................................initial migrations
├── __init__.py .................................... init of Python package
├── admin.py ........................... file for registering models to admin
├── apps.py ........................................... configuration of app
├── models.py ........................................... models definitions
├── tests.py ..................................................file for tests
└── views.py .................................................file for views
```

files are empty apart from the `apps.py` file which contains 3 lines of code to register the application into the admin site.

Next I declared the models I have specified in the conceptual design. For now I have only declared the models with their foreign keys and empty properties. Each model inherits from the `django.db.models.Model` class, which makes it a Django model. The model declarations of `UserStat` can be examined on listing 3.6, `UserAssignmentStat` on listing 3.7 and the `CourseStat` on listing 3.8.

The calculations of the properties were carried out as stated in the conceptual design part. I will not include these codes, because it would be complicated to fit it into the page and there is not much point in it, because the final code can be accessed through the faculty GitLab (11) or on the enclosed CD (C). For calculating the statistical values like median or percentile, I will use NumPy and SciPy. The functions are much faster than regular Python functions.

To support this, I carried out a small test comparing the median and average functions. I generated a list of random values from -10^6 to 10^6 of length $5 * 10^6$. The test is available in figure 3.9 and the results on figure 3.10.

Because NumPy and SciPy is compiled in the `gcc` compiler, I not only had to install the packages, but also I also had to add the `gcc` installation commands to the `Dockerfile`. I used a code from stack-overflow to install all the necessities on building the Docker container [11].

To add these models to the Django admin interface, I had to register them to the admin site. This was carried out by adding each of these models to the `admin.py` file. I used the `register_model()` method which was already implemented in LearnShell in the `utils.py` file. This method accepts a Django model as a parameter and registers it to the admin site. Adding models to the admin site is visualised on listing 3.11.

```python
from django.db import models


class UserStat(models.Model):
    course = models.ForeignKey(
        'course.Course',
        on_delete=models.CASCADE,
    )
    user = models.ForeignKey(
        'user.User',
        on_delete=models.CASCADE
    )
    @property
    def score(self):
        pass

    @property
    def score_history(self):
        pass

    @property
    def percentile_history(self):
        pass

    def create_user_assignment_stats(self):
        pass

    @property
    def user_assignment_stat(self):
        return self.userassignmentstat_set
```

Listing 3.6: Declaration the `UserStat` model

```
from django.db import models


class UserAssignmentStat(models.Model):
    user_stat = models.\
        ForeignKey(UserStat, on_delete=models.CASCADE)
    assignment = models.\
        ForeignKey(
            'assignment.Assignment',
            on_delete=models.CASCADE,
        )

    @property
    def score(self):
        pass

    @property
    def percentile(self):
        pass

    @property
    def user(self):
        return self.user_stat.user
```

Listing 3.7: Declaration the `UserAssignmentStat` model

3.3.2 Generating the GraphQL API

To make the proposed (and other) GraphQL queries function I had to program Django describers. In the file structure I added the file `describers.py`. This file contains describers for distinct Django models. In the describers I defined how the GraphQL API for the new analytics module models should work. Each model has a describer class which inherits from the `Describer` class of the package `django_describer`.

The parent class automatically creates some actions, like for instance a `list_action` which allows you to query a list of all nodes of the given model. Just a basic code like in figure 3.12 will allow you to list all the `UserStats` along with the id of the `UserStat` as shown in figure 3.13. The properties are not detected automatically. They need to be added as `extra_fields` as shown for the properties of the `CourseStat` on figure 3.14. The full code of the describers for the analytics module can again be examined on my branch of the faculty GitLab (11) in file `ls/apps/analytics/describers.py`. This file contains describers for all the models in the analytics module. As may be noticed, in the `describers.py` there is an import from an `analytics.permissions` file.

```python
from django.db import models


class ScoreHistogram(models.Model):
    course_stat = models.ForeignKey(
        'CourseStat',
        on_delete=models.CASCADE,
    )
    score = models.\
        IntegerField(
            help_text='a score at least some user obtained',
        )
    frequency = models.\
        IntegerField(
            help_text='the frequency of this score',
        )

    def increment_frequency(self):
        self.frequency += 1
        self.save()
        return self.frequency


class CourseStat(models.Model):
    course = models.OneToOneField(
        'course.Course',
        on_delete=models.CASCADE,
        default=0,
    )

    @property
    def median(self):
        pass

    @property
    def median_history(self):
        pass

    @property
    def score_histogram(self):
        return self.scorehistogram_set
```

Listing 3.8: Declaration the `CourseStat` model

```python
import random
import statistics
from time import process_time

import numpy as np

rng = 1000000
length = 5000000

random_list = [random.randint(-rng, rng) for i in range(length)]

print('np median vs py median.')
start = process_time()
np.median(random_list)
end = process_time()
print('np median took {}s'.format(round(end-start, 5)))

start = process_time()
statistics.median(random_list)
end = process_time()
print('py median took {}s'.format(round(end-start, 5)))

print('np average vs py average. ')
start = process_time()
np.average(random_list)
end = process_time()
print('np average took {}s'.format(round(end-start, 5)))

start = process_time()
statistics.mean(random_list)
end = process_time()
print('py average took {}s'.format(round(end-start, 5)))
```

Listing 3.9: Testing NumPy vs Pyhon

```
np median vs py median.
np median took 0.38795s
py median took 1.51091s
np average vs py average.
np average took 0.31119s
py average took 1.91349s
```

Listing 3.10: NumPy vs Python results

```python
from utils import register_model
from .models import UserStat, CourseStat, UserAssignmentStat


register_model(UserStat)
register_model(CourseStat)
register_model(UserAssignmentStat)
```

Listing 3.11: Adding all the models from the analytics module to the admin site

```python
from django_describer.describers import Describer
from django_describer.permissions import Or
from analytics.models import UserStat
from analytics.permissions import IsAdmin, IsSelf, IsTeacherOf

class UserStatDescriber(Describer):
    model = UserStat

    default_field_permissions = Or(IsAdmin, IsSelf, IsTeacherOf)
    default_action_permissions = IsAdmin
```

Listing 3.12: Basic describer of UserStat

```
query {
    UserStatList {
    results {
      id
    }
  }
}
```

Listing 3.13: UserStat list action

```
from django_describer.datatypes import Integer, QuerySet
from django_describer.describers import Describer
from django_describer.permissions import Or

from analytics.models import CourseStat, ScoreHistogram
from analytics.permissions import IsAdmin, IsStudent

class CourseStatDescriber(Describer):
    model = CourseStat

    default_field_permissions = Or(IsAdmin, IsStudent)
    default_action_permissions = IsAdmin

    extra_fields = {
        'score_histogram': QuerySet(ScoreHistogram),
        'median': Integer
        'median_history': QuerySet(Integer)
    }
```

Listing 3.14: Basic describer of CourseStat

This file contains classes, which authorise actions by various groups of users. For instance, the `IsAdmin` permission checks whether the logged in user has an admin status and then allows the user to perform the action. If the user was not admin, the GraphQL would return an error message also defined in the `permissions.py` file.

3.3.3 Optimisations

After trying the new functionalities, I realised that this design is very slow. The methods for calculating the properties take time and listing all `UserStats` with all their properties took too long for it to be used in real deployment.

One option was to try and optimise the methods, make them more effective, by using NumPy for instance, but I knew that still would not make such a drastic change. The data had to be stored persistently and not recalculated on every call. The calculations still had to take place, but not on every call. So I had to decide when to recalculate them. Another alternative would be not to recalculate the data from scratch every time but update the data each week of the semester. However, I did not like that idea very much. I think it is better to recalculate the data from scratch because in that way it is not dependent on any changes in the semester, restarts of the server, different dates every year and so on.

Another option was to recalculate the data after each submission of an assignment by any user, but that still was not good enough, because there are times, when tens of students are creating multiple submissions at once and that would take up a lot of computing capacity at times, when the capacity is actually needed for other work.

Finally, for the computations, I decided to use a time, when the server is not taking large payloads – at night. I had to implement a function to recalculate all the statistical data and execute it periodically. Me and my colleague form the frontend agreed, that the recalculations should take place at the end of every week, because that is when the data are most relevant. It would not make much sense to analyse the weekly percentile on Monday, when most of the students have not even taken the weekly exam yet. The scores of other students would be very different at the end of the week, than at the beginning. The Monday students would have an unfair advantage of having taken one extra test in contrast with the other students.

In the end I decided to carry out the calculations every Saturday at 12:00am. For completing this task, it was quite convenient that Celery is already running in the LearnShell backend. I just had to schedule a task using Celery Beat and link it with the recalculating function. In order to use the Celery Beat I had to alter the `docker-compose.yml` file. In the command which activates the Celery worker in Docker I had to add the flag `-B`.

In the analytics module I created a new file `tasks.py` and in it defined a new function called: `friday_night_recalculation`. The content of the `tasks.py` file containing the code of the function can be examined on figure 3.15. This function basically calls all the recalculating methods from the other models in the right order. After this I had to schedule this task with Celery Beat.

To the `settings/celery_setup.py` I added a command to select the correct time zone and called `autodiscover_tasks()` which detects all functions with the decorator `@shared_task` in the whole project and allows me to schedule them. After updating the Celery setup file, I added the scheduling code (on listing 3.16) to the end of the `settings/settings.py` file. Here I added a new task to the Celery Beat schedule, selected the time and linked it with the `@shared_task`. To support these ideas and their execution I used [12] and [13].

I tested the scheduling by changing it to a time that was close at the moment of testing, the task was executed at the time, so the test was succesful. I also tried running Docker on a Friday night and at midnight the calculations did take place.

Thanks to Celery, the task runs asynchronously, so this does not mean that on Friday night nobody can use LearnShell for a few minutes but the server will definitely not crash from it as it could at a more busy time.

After implementing the scheduling at a time when the server is not used very much, there was not much need to optimise the recalculating methods anymore, because reducing the recalculating time by a few minutes/seconds would not make much of a difference. I decided to optimise the methods anyway because there was still room for it. Calculating some combinations of stats

```
from celery import shared_task
from assignment.models import Assignment
from .models import CourseStat, UserStat


@shared_task
def friday_night_recalculation():
    print('Initiating Friday night recalculation of all stats')
    # recalculate all users score and score history
    # and create assignment stats
    for user in UserStat.objects.all():
        user.recalculate_score()
        user.create_user_assignment_stats()

    # recalculate all users percentile from scores,
    # must be after first loop
    for user in UserStat.objects.all():
        user.recalculate_percentile()

    # recalculate median of all assignments,
    # max scores and score arrays
    for assignment in Assignment.objects.all():
        assignment.update()

    # recalculate median and median history of all courses
    for course_stat in CourseStat.objects.all():
        course_stat.calc_median()
        course_stat.calc_median_history()
        course_stat.calc_histogram()

    print('Stats recalculated')
```

Listing 3.15: Friday night recalculation

```
CELERY_BEAT_SCHEDULE = {
    # schedules the task to every saturday at 00:00
    'friday_night_recalculation': {
        'task': 'analytics.tasks.'
                'friday_night_recalculation',
        'schedule': crontab(
            hour=0,
            minute=0,
            day_of_week='saturday'
        ),
        'name': 'Statistics every friday night'
                ' at the end of the week.',
    },
}
```

Listing 3.16: Scheduling a task with celery

could be done in one method instead of having a method for recalculating each stat. For example calculating the `UserStat score` and `score_history` was done by iterating through the same data twice. The calculation of these stats was possible to merge into one method. It made the code less understandable, but saved one for loop in each users recalculation.

3.3.4 Testing, Fixes and other Improvements

To test the newly implemented code I used an anonymised dump of the PostgreSQL database from the previous semester (B201 – winter semester of 2020). The data contained students profiles with data of all the assignments they submitted. I had to create new tables for all the `UserStats` and `CourseStats`. I overrode the `save()` methods in the `User` and `Course` model to add a new stat table, if it does not exist yet on save of the model. After that I tested all the possible queries and added some sample queries to the the analytics module to the `samle_queries_for_analytics.graphql` file in the `sample_queries` directory. Then I created a new dump of the database with all the new tables which can be loaded by running the `load_dump.sh` script in the `scripts` directory and used for testing.

I called the queries in various combinations and analysed the data manually checking whether they make sense. Another big part of the testing was carried out when my colleague connected the new queries to the frontend and visualised the data. Occasionally, there occurred some minor errors which were not hard to solve. There were some issues with permissions in the Django describers, for example, the student was not able to list his own data. I missed this error before because I tested it only logged in as admin. Then there were some errors in the data calculation but nothing serious.

One thing we realised with my colleague from frontend was, that the `percentile_history` in `UserStat` and `median_history` in `CourseStat` was not very relevant, because the course is taught in the first semester of FIT and unfortunately a vast amount of students leave the faculty before the end of the first semester. As a result, in the data, there are a lot of students which are idle. We defined these students as students with the score at the end of the semester from 0 to 5. My colleague from the frontend decided to add a toggle option to display the statistics with either counting these students in or not. For this I decided to add new array-fields to the database. In `CourseStats` I added `median_history_idle` which is the history of median of all the students and `median_history` which excludes students with the score lower than or equal to 5. Analogically I added `percentile_history_idle` to the `UserStat` model.

Then my colleague from the frontend decided, that he wants to add a final score prediction to the students dashboard. He had some way of calculating it in mind, but since he proposed this addition a week before the deadline for the thesis, I decided to define the score prediction myself considering the lack of time. I added an integer field `final_score_prediction` to the `UserStat` table. I created a recalculation method for it and added it to the Friday night recalculation task. The score prediction is calculated by iterating through the students of the previous semester of the same course. I had to add an extra static method to retrieve the KOS tag of the previous semester from the current semester (for 'B201' the method returns 'B191'). In every iteration I store the final score of the students whose end-of-week score differs from the users end-of-week score by up to 3 and then I calculate the average of these scores (using NumPy). The average is the final score prediction.

3.4 Cloud service provider selection

In this section I will be analysing 3 major cloud service providers that can run Docker containers. Why use a third party cloud service provider in the first place? At the moment, LearnShell is running on faculty servers. These servers are not prepared to take such payloads LearnShell demands. Respectively, they are but the faculty servers are used for many other services and we can't take up all the computing capacity with LearnShell. Last year (in the semester B201 – winter semester of 2020) we experienced various problems with the server crashing whilst a whole class was writing an exam on LearnShell. And we are talking about only one course in one semester being taught with LearnShell. With LearnShell there are ambitions to teach more courses, perhaps even on different faculties or schools.

For the time being, LearnShell will stay on the faculty servers, but this financial analysis could prove useful in the future, when the management decides to migrate it. There are also possibilities of obtaining a grant from

Google which could be used to finance the cloud service provision by Google. When applying for these grants analysis like this are decisive.

I will compare the pricing of the three cloud service providers and pick the best one for LearnShell. There are of course many cloud service providers and I have to pick 3 to analyse. I have decided to choose:

- AWS – Amazon Web Services
- Microsoft Azure
- Google Cloud

because they are the leading cloud service providers according to [14] and other websites. Another important thing is, that they all support Docker containers. The Google Cloud is also picked due to the grant mentioned previously.

First, I will need to create an estimate of how much processing power and which services will be needed to satisfy the needs of LearnShell. I will consider two scenarios, one ideal and one with a little reserve (I will call it “excessive”) and then I will apply these parameters to the pricing calculator of each CSP. These calculators will calculate the estimate of the monthly fee for using their services. This will help me select the best one.

Next I will explain which services I have selected in the individual CSP calculators, adding a link to the calculator configurations. The links to the calculators expire in some time e. g. 3 years, but all the calculated prices will be listed. At the end there will be an overview of the pricing analysis in table 3.2 with the data visualised in figure 3.9.

For the ideal configuration I will select a Kubernetes engine to run 10 virtual private servers with a 4 core processor and 16 GiB of RAM. These servers will run the LearnShell services. As explained in the *current architecture analysis* 3.1, there are only 4 services. Having 10 virtual private servers will allow to duplicate the services when needed, to double the computing capacity of the given service. Then for the excessive scenario, I will select 16 virtual private servers with the same configuration. We will need a 24/7 availability, because students could have homework assignments, which they could be working on, whenever.

I will be using the dollar currency in all the calculators and calculate with the monthly fee in order to keep the ratio.

With Amazon Web Services I have selected the service Amazon Elastic Computer Cloud (EC2). Into the calculator I entered the previously mentioned parameters of the two scenarios. Amazon also offers various financing plans. I will look into the on-demand pricing and the EC2 instance saving plans. The ideal scenario with on demand pricing totalled to $ 1,750.70 per month (link to calculator 1). The ideal scenario with the *EC2 Instance Savings Plans* reserving the service for 3 years totalled to $ 823.60 per month (link to calculator 2). The excessive scenario totalled to $ 2,801.12 per month

Table 3.2: Summary of CSP pricing

Cloud service provider	**monthly fee**	
	ideal	**excessive**
Amazon web services	$ 1,750.70 $ 823.60*	$ 2,801.12 $ 1,317.76*
Microsoft Azure	$ 1,708.20 $ 741.10*	$ 2,733.12 $ 1,185.75*
Google cloud	$ 1,675.94	$ 2,681.51

Note: * price with the savings plan.

(link to calculator 3) with on demand pricing and $ 1,317.76 per month (link to calculator 4) with the savings plan.

Next, let's take look at Microsoft Azure. In the Microsoft Azure calculator I have selected the Azure Kubernetes Service (AKS). Microsoft Azure also offers price saving plans, so there will be 4 estimates. For the ideal scenario with on demand pricing they charge $ 1,708.20 per month (link to calculator 5). Selecting the 3 year saving plan drops the price to $ 741.10 per month (link to calculator 6). The excessive scenarios are priced at $ 2,733.12 per month (link to calculator 7) for the on demand pricing plan and $ 1,185.75 per month (link to calculator 8) for the savings plan.

With Google Cloud I have selected the service Google Kubernetes engine (GKE) Autopilot. This server can automatically manage nodes based on their health and payload [15]. Google Cloud offers only on demand pricing, so we will look into only 2 estimates. These estimates reached $ 1,675.94 per month (link to calculator 9) for the ideal scenario and $ 2,681.51 per month (link to calculator 10) for the excessive scenario.

Finally I have reached the conclusion that the main parameter which will affect the monthly fee is whether the faculty would be willing to commit to the cloud service provider for 3 years or not. This option is a huge advantage of AWS and Microsoft Azure opposing to Google cloud. If the faculty would commit, then the cheapest would be Microsoft Azure for both the *ideal* and the *excessive* scenario. If the faculty decided to choose on demand pricing, Google Cloud would be the best bet for both scenarios. The commitment makes such a significant difference that with the savings plan, the monthly fee would be lower for the *excessive* scenario (with both AWS and Microsoft Azure) than it would for the *ideal* scenario with all cloud service providers. Overall the monthly fee (when selecting on demand pricing) is very similar with all cloud service providers and could probably be tweaked with some minor changes in the configuration. Therefore I would recommend the Google Cloud if the faculty went for the on demand pricing, due to its GKE Autopilot and the possibility of financing it with a Google grant.

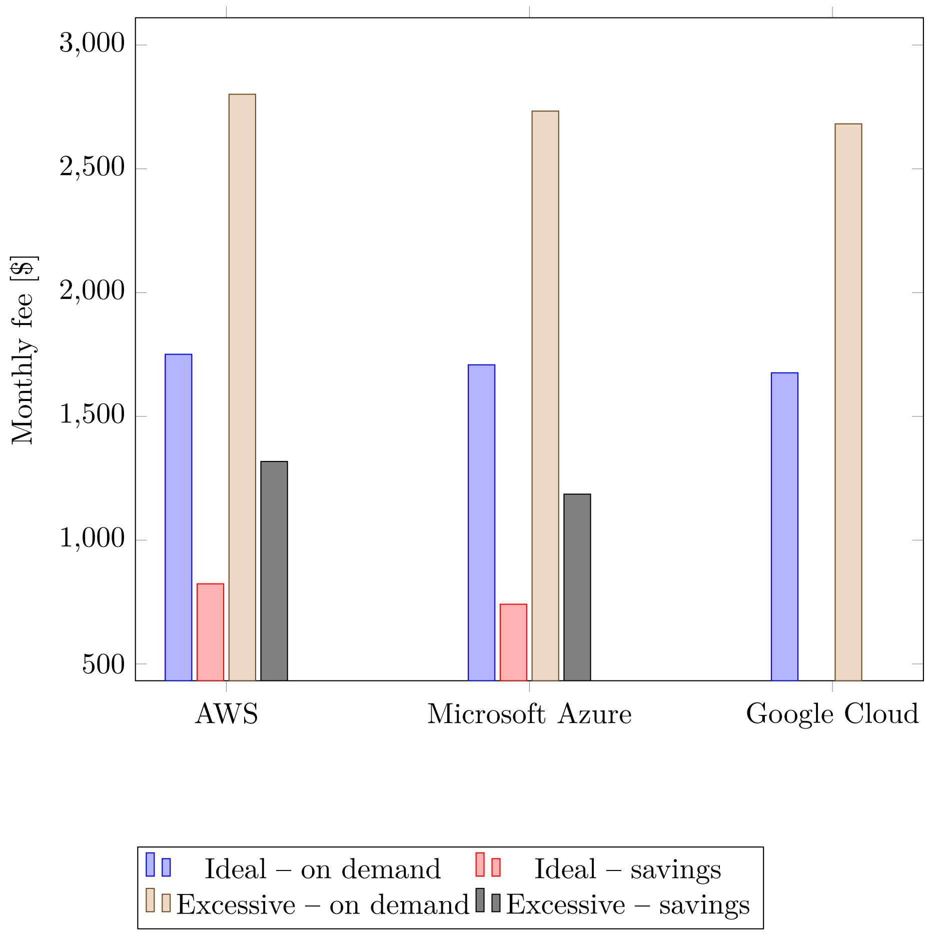

Figure 3.9: Chart of CSP pricing

Chapter 4

Accomplishments

4.1 Report of improvements, Future re-evaluation

In conclusion I would like to summarise, what improvements I have accomplished to carry out in the LearnShell backend. I have created a functioning analytics module, which calculates and stores statistical data of students and courses. The module is prepared for other additions in the future.

Students now have the possibility to analyse their performance throughout the semester. They can view their overall score (without having to calculate it manually from all their assignments), they can analyse how their score developed through time and analyse their percentile change through time. They also received a final score prediction based on the performance of students from previous years. Students can also view data about the whole course they are studying at the moment, like for instance the score histogram of all current scores or median in the given course. They can also view overall statistics of all assignments that were generated for them, like the median of all scores obtained from the given assignment.

My module calculates and returns the data with GraphQL queries and my colleague from the frontend bound them to user friendly charts and graphs on the web interface. Each student now has a *Dashboard* tab, where he/she can display the data the analytics module provides.

There are also data, which are not being used in the frontend yet, like summaries of all courses performances or medians of all assignments.

Another contribution is that LearnShell now possesses an analytics module which can be very simply extended. In the future, it would be possible to analyse other data. Ideas for future extension of the analytics module are:

- Calculating statistical data of various parallels
    - The module could have a table for storing statistical data of various parallels

    - Then, it would be possible to analyse performances of parallels and deduct what the differences of their performances are due to. For instance, whether evening parallels are less productive than morning parallels, determining the most productive day of week, etc.

- Analysing the success rates of assignments
    - There could be added an assignment stat table, which would calculate the success rate of the assignment.
    - That would make possible determining which assignments were perhaps too easy and which were too hard.

- Teachers dashboard
    - In the future there could be added a table for statistics of a teacher
    - The teacher could have the possibility to analyse the success rates of his/her parallels
    - Compare his/her parallel performances to other teachers parallels

These improvements (and more) can be added to LearnShell in the future. Perhaps it could even be a topic for another bachelor's thesis.

Another topic I would like to discuss is future re-evaluation. It would be appropriate to analyse, whether and how much students actually use the analytics module resp. how much they use the students dashboard on the frontend. Here comes the question whether it would be better / more convenient to collect the data about the usage in the backend or the frontend. Since, at the moment the, students do not really have an option to view specific statistical data discretely, there is not much point in adding tracking to the backend queries. The students open their dashboard tab and they see all the viewable data at once. In conclusion it would probably be best to add tracking to the dashboard tab button and analyse how often students actually access the data from the analytics module overall.

If however, the frontend changed in the future and made each statistic available discretely, it would be possible to measure which statistic interests them most and which they perhaps do not use at all.

To do this there are various options. One option would be to implement a self-made tracking module into LearnShell which would contain counters for individual queries and mutations. Another way is to use a professional charged tool. There are some popular tools for this like Mixpanel or Amplitude. In my opinion, these tools are too advanced for LearnShell at the moment. Since LearnShell is not openly on the internet, there is no need to maximise the effectiveness and reach as much users as possible. That is what these tools are usually used for. They are payed-for, so their users look for some money in return by reaching more users. LearnShell is a specific system for students who have no other option but to use it. However, if we decided to use one of these tools in the future, I would recommend Amplitude. According to [16], Mixpanel concentrates on mobile apps a lot, whetheras Amplitude is more product based and more appropriate for web apps, which LearnShell is.

4.2 Documentation

The whole code of the new analytics module is thoroughly commented by docstrings. There is a generated documentation of the analytics module `models.py` file, in the `docs/analytics` directory on GitLab (11) and the enclosed CD (C). The documentation was generated using `pdoc`, which is an auto generator for Python APIs [17]. At first there were problems with generating the documentation due to Django applications not being started. I resolved this issue by temporarily removing the Django references from the code. Thanks to this, it was possible to generate the documentation. The code would not work and is not and identical copy of the real code, but the documentation generated from this "dummy" code is perfectly understandable and identical with how the documentation would look if the Django references did not cause exceptions and allowed the documentation to generate.

I have also added a readme file for programmers, who will want or need to change or update the analytics module. This file briefly describes the analytics module and gives a small hint on how to add new functionalities to the analytics module. The file also contains instructions on how to test the analytics module using a database dump.

In the directory, there is also a new database model of the analytics module, shown in figure 4.1.

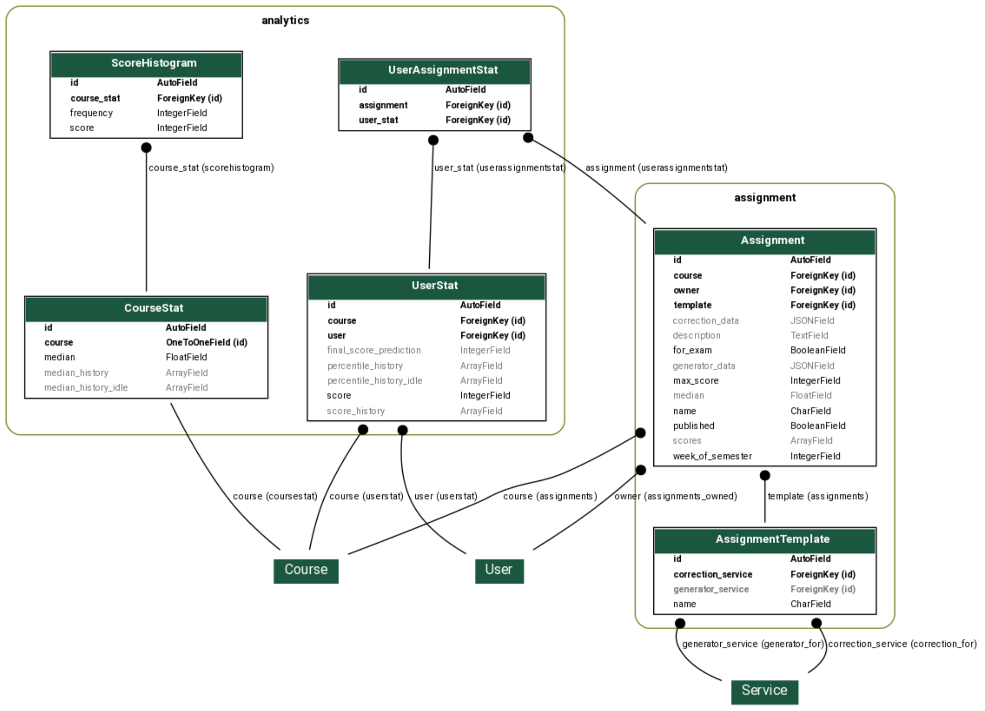

Figure 4.1: Schema of Django models in the analytics module

Conclusion

To conclude this thesis, I will evaluate whether I have accomplished to fulfil the goals defined in the beginning. The main goal – improving LearnShell backend for analytics – was accomplished.

I have defined the technology LearnShell uses, analysed the current LearnShell backend architecture and identified the parts which needed improvements.

Then, I have proposed the improvement: creating the new analytics module, designed it on a conceptual level and implemented the new functionalities. After that, I optimised the performance of all methods in the analytics module.

Inter alia, I have performed a financial analysis of three cloud service providers to which LearnShell could be migrated in the future and recommended Google Cloud.

Finally I have concluded a report of improvements in the LearnShell backend and proposed some possible improvements for the future. I also discussed the question of future re-evaluation and compiled internal documentation of the analytics module.

My main output is, that there is now a new functional analytics module, which students can use to analyse their performance throughout the semester. This improvement can be deployed immediately in the next semester and be a contribution to the students and teachers who use it.

Another contribution is my personal development in my field, as should of course be of any final thesis. I now have a deeper understanding of how software development works in practise. I have also discovered new technologies which I did not previously know of or how they work.

Bibliography

[1] Red Hat, Inc. What is Docker? *Opensource Resources [online]*, 2021, [cit. 2021-04-22]. Available from: `https://opensource.com/resources/what-docker`

[2] Juwel, A. I. Asynchronous Task with Django Celery Redis and Production using Supervisor. *Start it up [online]*, 2020, [cit. 2021-04-20]. Available from: `https://medium.com/swlh/asynchronous-task-with-django-celery-redis-and-production-using-supervisor-ef920725da03`

[3] Redis. Introduction to Redis. *Redis [online]*, 2021, [cit. 2021-04-21]. Available from: `https://redis.io/topics/introduction`

[4] Django Software Foundation. Django. *Meet Django [online]*, 2021, [cit. 2021-04-21]. Available from: `https://www.djangoproject.com/`

[5] The PostgreSQL Global Development Group. PostgreSQL: The World's Most Advanced Open Source Relational Database. *PostgreSQL official website [online]*, 2021, [cit. 2021-04-21]. Available from: `https://www.postgresql.org/`

[6] Carbonnelle, P. PYPL PopularitY of Programming Language. *PYPL index [online]*, 2021, [cit. 2021-05-07]. Available from: `https://pypl.github.io/PYPL.html`

[7] Gouy, I. Which programming language is fastest? *Meet Django [online]*, 2021, [cit. 2021-04-21]. Available from: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html`

[8] Bolton, D. 5 Reasons You Should Know NumPy. *Dice insights [online]*, 2016, [cit. 2021-05-09]. Available from: `https://insights.dice.com/2016/09/01/5-reasons-know-numpy/`

[9] The GraphQL Foundation. Introduction to GraphQL. *GraphQL Learn [online]*, 2021, [cit. 2021-04-28]. Available from: `https://graphql.org/learn/`

[10] Jílek, K. Project description. *Django Describer Documentation [online]*, 2020, [cit. 2021-04-25]. Available from: `https://pypi.org/project/django-describer/`

[11] ndclt. Unable to install numpy on docker python3.7-slim in a raspberry pi. *stackoverflow [online]*, 2020, [cit. 2021-04-27]. Available from: `https://stackoverflow.com/a/63979755`

[12] J-O Eriksson. Handling Periodic Tasks in Django with Celery and Docker. *TestDriven.io [online]*, 2020, [cit. 2021-04-29]. Available from: `https://testdriven.io/blog/django-celery-periodic-tasks/`

[13] Ask Solem. Periodic Tasks. *Celery 5.0.5 documentation [online]*, 2018, [cit. 2021-04-29]. Available from: `https://docs.celeryproject.org/en/stable/userguide/periodic-tasks.html`

[14] Django Software Foundation. Top Cloud Service Providers & Companies of 2021. *Meet Django [online]*, 2021, [cit. 2021-05-08]. Available from: `https://www.datamation.com/cloud/cloud-service-providers/`

[15] Google. Autopilot overview. *Google Cloud Documentation [online]*, 2021, [cit. 2021-05-08]. Available from: `https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview`

[16] Practico Analytics. Amplitude vs Mixpanel? Pros and Cons of Each. *Youtube [online]*, 2018, [cit. 2021-05-10]. Available from: `https://www.youtube.com/watch?v=SEnHJ6i2NOo&ab_channel=PracticoAnalytics`

[17] pdoc. What is pdoc? *pdoc documentation [online]*, 2020, [cit. 2021-05-12]. Available from: `https://pdoc.dev/docs/pdoc.html`

Appendix A

Acronyms

CTU	Czech Technical University
FIT	Faculty of Information Technology
LS	LearnShell
API	Application Programming Interface
PS1	Programming in shell 1
KOS	Study information system of CTU
GQL	GraphQL
UI	User interface
CSP	Cloud service providers
JSON	JavaScript Object Notation
OOP	Object oriented programming
RAM	Random Access Memory
VPS	Virtual Private Server

Appendix B

List of hyperlinks

1. Amazon calculator ideal, on demand, available from:
   https://bit.ly/3uARbzR
2. Amazon calculator ideal, savings, available from:
   https://bit.ly/2SH2NDp
3. Amazon calculator excessive on demand, available from:
   https://bit.ly/3f5GiiZ
4. Amazon calculator excessive savings, available from:
   https://bit.ly/2SIig67
5. Microsoft Azure calculator ideal, on demand, available from:
   https://azure.com/e/a4c4c94a0a154f339e6b7b86b6ae59b0
6. Microsoft Azure calculator ideal, savings, available from:
   https://azure.com/e/4a4ed8a194f1431e84e0fd4bd2aed339
7. Microsoft Azure calculator excessive, on demand, available from:
   https://azure.com/e/21388a95cb3c4e519745669acbce7fe4
8. Microsoft Azure calculator excessive, savings, available from:
   https://azure.com/e/67d9f020991042dea91280946c963884
9. Google calculator ideal, available from:
   https://bit.ly/33wgcjL
10. Google calculator ideal, available from:
    https://bit.ly/3y0KrNN
11. My branch on the faculty GitLab:
    https://gitlab.fit.cvut.cz/learnshell-2.0/ls/tree/bap-pejchdan

Appendix C

Contents of enclosed CD

```
README.md ........ the file with the CD contents description in md format
thesis.pdf ............................... the thesis text in PDF format
sources .................................... directory with source codes
  ls .............. the directory with a clone of my branch from GitLab
  ls-ps1-generator ...... the directory with LS generator from GitLab
thesis .................. the directory of LaTeX source codes of the thesis
```