МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Ижевский государственный технический университет имени
М.Т.Калашникова»
(ФГБОУ ВО «ИжГТУ имени М.Т.Калашникова»)
Кафедра «Программное обеспечение»

УДК xxx(xx)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к выпускной квалификационной работе бакалавра на тему:

«Разработка облачной платформы для создания и удалённого запуска учебных сервисов»

Дипломник
студент гр. Б08-191-1 М.С. Ивченко

Руководитель В.Г. Тарасов

Нормоконтролер В.П. Соболева

Зав. кафедрой ПО
к.т.н., доцент А.В. Коробейников

Ижевск 2020

РЕФЕРАТ

Пояснительная записка к выпускной квалификационной работе на тему «Разработка облачной платформы для создания и удалённого запуска учебных сервисов» представлена на 145 страницах и включает в себя 1 таблицу, 5 использованных источников и 45 иллюстраций.

Целью данной выпускной квалификационной работы является повышение эффективности образовательного процесса при изучении дисциплин IT-направления высшего образования путём оптимизации процесса создания учебных сервисов и предоставления доступа к ним из любого места с доступом к сети Интернет.

В итоге выполнения работы была разработана облачная платформа для создания и удалённого запуска учебных сервисов. Новизна облачной платформы заключается в использовании современных технологий изоляции, обеспечивающих большую эффективность использования ресурсов, а также ориентированности платформы на потребности образовательного процесса.

В будущем облачная платформа может использоваться как основа для разработки системы автоматической проверки лабораторных работ по дисциплине «Базы данных».

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Исследования показывают, что образование является важным фактором экономического развития государства [1]. Качество обучения новых кадров влияет на темпы развития новых технологий, рост ВВП, положение государства на мировом экономическом рынке. Особую роль занимают специальности IT-направления, что подчеркивается распоряжением Правительства РФ «Об утверждении перечня специальностей и направлений подготовки высшего образования, соответствующих приоритетным направлениям модернизации и технологического развития российской экономики» от 6 января 2015г.

Обучение специальностям, связанным с IT, зачастую связано с использованием различных учебных сервисов – баз данных (MS SQL, Redis, MongoDB, и др.), сетевых очередей сообщений (RabbitMQ), веб-оболочек для интерактивных вычислений (Jupyter Notebook). Все эти сервисы требуют установки и, зачастую, первоначальной настройки.

Эта процедура занимает значительное время у каждой обучающейся группы студентов, что снижает эффективность обучения. К тому же, студенты вынуждены либо использовать свои ноутбуки, что могут позволить себе не все, либо использовать компьютеры в лабораторных классах и каждое занятие сталкиваться с трудностями переноса данных между домашним компьютером и компьютером в университете.

Целью данной работы является повышение эффективности образовательного процесса при изучении дисциплин IT-направления высшего образования путём оптимизации процесса создания учебных сервисов и предоставления доступа к ним из любого места с доступом к сети Интернет.

В ходе работы разработана облачная платформа для создания и удалённого запуска учебных сервисов. С её помощью возможно быстро создавать на основе шаблонов учебные сервисы и осуществлять к ним доступ как в университете, так и за его пределами. На данный момент в публичном доступе существуют аналогичные платформы, однако они ориентированы на

коммерческие предприятия, а не на процесс обучения, поэтому их использование для решения поставленной цели затруднительно.

1. РАЗРАБОТКА ОБЛАЧНОЙ ПЛАТФОРМЫ ДЛЯ СОЗДАНИЯ И УДАЛЁННОГО ЗАПУСКА УЧЕБНЫХ СЕРВИСОВ

1.1. Обоснование целесообразности разработки платформы

1.1.1. Назначение платформы

Данная платформа предназначена для создания и удалённого запуска учебных сервисов. Она должна позволять преподавателям создавать учебные курсы, в рамках которых студентам будет предоставляться квота серверных ресурсов (ядра процессора, оперативная память, место в сетевом хранилище данных) и управлять членством студентов в этих курсах. Платформа должна позволять студентам подавать заявки на участие в учебных курсах, создавать учебные сервисы из шаблонов, предоставленных администраторами, в рамках учебных курсов, на которых они зачислены, просматривать и управлять состоянием сервисов и подключаться к ним. Квота, которой может распоряжаться каждый преподаватель, а также список преподавателей должны управляться администраторами платформы.

В совокупности это все позволит студентам создавать и запускать учебные сервисы, используя серверные ресурсы университета, находясь в любом месте.

Управление квотами позволит контролировать распределение серверных ресурсов, что обеспечит доступность возможностей платформы для всех её пользователей.

1.1.2. Обоснование цели создания платформы

Студенты IT-специальностей в процессе обучения тратят значительное время на установку и настройку различных учебных сервисов – баз данных (MS SQL, Redis, MongoDB, и др.), сетевых очередей сообщений (RabbitMQ), веб-оболочек для интерактивных вычислений (Jupyter Notebook). При этом студенты вынуждены либо использовать свои ноутбуки, либо повторять процедуру дома и каждое занятие переносить данные между компьютерами в университете и дома. Все это негативно сказывается на эффективности образовательного процесса. Цель создания платформы – повысить

эффективность образовательного процесса при изучении дисциплин IT-направления высшего образования путём оптимизации процесса создания учебных сервисов и предоставления доступа к ним из любого места с доступом к сети Интернет.

1.1.3. Обоснование состава автоматизируемых задач

На основании выше поставленной цели можно выделить следующие задачи:

1) разработка подсистемы управления пользователями. Эта подсистема отвечает за авторизацию пользователей и управления их ролями, определяющими, студент этот пользователь, преподаватель, или администратор;

2) разработка подсистемы управления учебными курсами. В эту подсистему входит создание и изменение учебных курсов, а также управление членством в них;

3) разработка подсистемы управления сервисами. Эта подсистема позволяет пользователям создавать учебные сервисы, узнавать и изменять их состояние;

4) разработка подсистемы управления квотами ресурсов. Эта подсистема отвечает за выделение квот ресурсов при вступлении студента в учебный курс и создании учебных сервисов, а также за своевременное их освобождение;

5) разработка клиентского приложения. Клиентское приложение позволяет пользователю взаимодействовать с платформой с помощью графического интерфейса.

1.2. Аналитический обзор

Успешный опыт реализации подобной платформы для использования при обучении студентов был представлен в статье «Facilitating education using cloud computing infrastructure» под авторством Lei Huang, Yonggao Yand [2]. Однако, авторы этой статьи использовали (вероятно, в связи с недостаточным развитием альтернатив в 2013 году) для обеспечения изоляции учебных

сервисов виртуализацию, что требует большего количества серверных ресурсов из-за накладных расходов на эмуляцию оборудования и работу гостевых ОС для каждого запущенного сервиса (см. рис. 1.1).

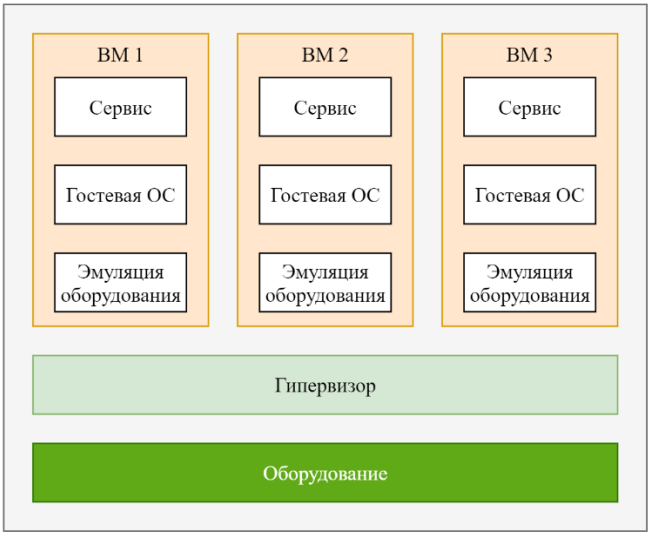


Рис. 1.1. Структурная схема изоляции сервисов при использовании виртуализации

В настоящее время широкое распространение получил способ изоляции, основанный на контейнеризации, благодаря снижению накладных расходов из-за изоляции не операционных систем, а отдельных приложений. За счёт того, что изоляция происходит на уровне ядра ОС (см. рис. 1.2) отсутствует необходимость эмуляции оборудования и отсутствуют накладные расходы на запуск гостевой ОС.

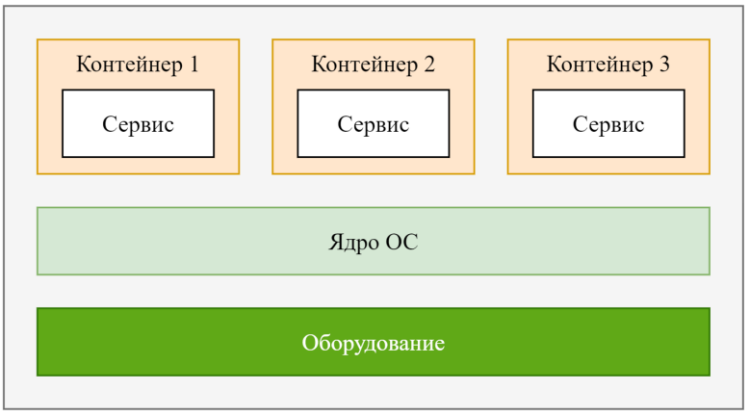


Рис. 1.2. Структурная схема изоляции сервисов при использовании контейнеризации

Поскольку использование контейнеризации является более эффективным, платформа, разработанная в ходе данной выпускной квалификационной работы, использует её.

Для запуска сервисов в контейнерах используется Kubernetes – инфраструктурный компонент, предоставляющий API для управления контейнерами на нескольких серверах одновременно. Kubernetes является лидирующим решением для построения облачной инфраструктуры и поддерживается большинством крупных поставщиков облачных услуг, таких как Google (GKE), Amazon (AWS EKS), Microsoft (AKS), Яндекс (Yandex Managed Service for Kubernetes), Mail.ru (Cloud Containers от Mail.ru) и др. Использование Kubernetes в качестве инфраструктуры облачной платформы позволяет использовать облачную платформу как на собственных серверах (путём установки Kubernetes), так и на серверах большинства крупных поставщиков облачных услуг, что обеспечит возможность выбора наиболее экономически привлекательного поставщика вычислительных ресурсов.

Так как обучающихся студентов значительное количество, предоставлять всем им ресурсы на протяжении всего обучения является слишком накладным, поэтому облачная платформа должна иметь возможность выделять вычислительные ресурсы студентам только на время прохождения определенных курсов.

Рассмотрим существующие публичные аналоги данной платформы:

1) OpenShift Container Platform. Облачная платформа, основанная на Kubernetes, позволяет пользователям создавать и запускать сервисы с помощью графического интерфейса, в том числе, используя шаблоны. Проектно-ориентированный интерфейс (см. рис. 1.3), для распределения ресурсов надо выделять квоты на отдельные проекты, нет простой возможности выделять ресурсы на время прохождения курса. Возможна установка на свои сервера и использование вычислительных ресурсов в облаке Red Hat.

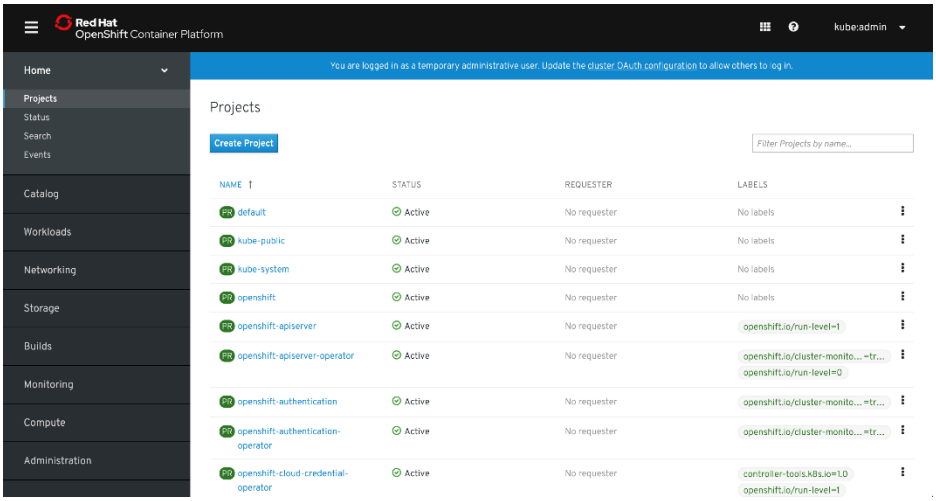


Рис. 1.3. Интерфейс OpenShift Container Platform

2) Sloppy.io. Облачная платформа, позволяет пользователям создавать и запускать сервисы с помощью графического интерфейса (см. рис. 1.4). Отсутствуют шаблоны сервисов, нет управления ресурсами, выделенным пользователям. Возможно только использование вычислительных ресурсов в облаке компании Sloppy.io.

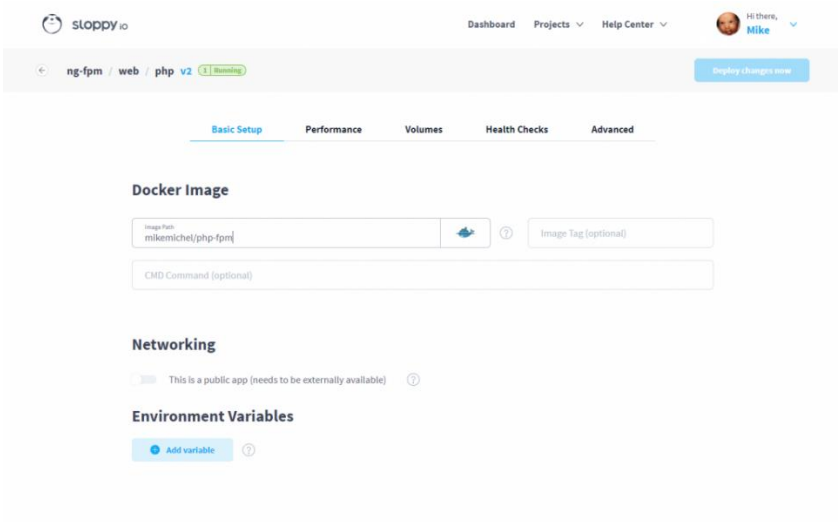


Рис. 1.4. Интерфейс sloppy.io

3) Digital Ocean. Облачное решение, предоставляющее возможность создавать и запускать сервисы. Возможно использовать только вычислительные ресурсы компании Digital Ocean. Есть графический интерфейс (см. рис. 1.5), возможно использовать шаблоны. Нет способа управлять лимитами на использование вычислительных ресурсов.

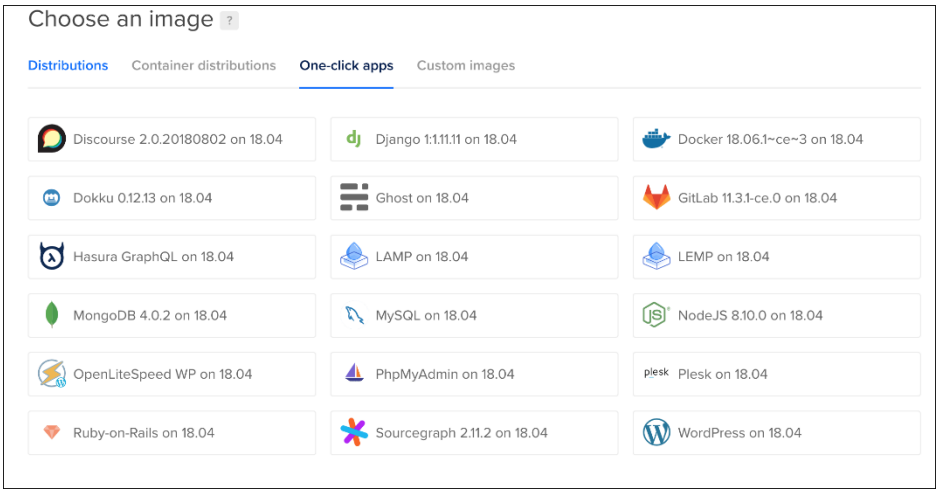


Рис. 1.5. Интерфейс Digital Ocean

В табл. 1 приведена сравнительная характеристика перечисленных выше аналогов и разработанной облачной платформы.

Таблица 1

Сравнительная характеристика аналогов

Название продукта	OpenShift Container Platform	Sloppy.io	Digital Ocean	Разработанная облачная платформа
Широкий выбор поставщиков вычислительных ресурсов	-	-	-	+
Возможность создания шаблонов сервисов	+	-	+	+
Предоставление ресурсов только на время прохождения курса	-	-	-	+

Анализ этой таблицы показывает, что на рынке нет продуктов, которые в полной мере удовлетворяют всем потребностям. Существующие продукты

проектно-ориентированы и рассчитаны на командное использование с выделением ресурсов в постоянное использование, а также они привязаны к одному или небольшому количеству поставщиков вычислительных ресурсов.

Разработка собственного решения также является более перспективной, поскольку позволит в дальнейшем автоматизировать проверку заданий по работе с базами данных. Так как облачная платформа обеспечивает удалённый доступ к учебным сервисам, в число которых входят базы данных, имеются все предпосылки для запуска программ, взаимодействующих с БД, запущенными конкретными студентами.

Исходя из вышеперечисленного, использование существующих решений затруднительно и необходимо проектирование оригинальной облачной платформы, не имеющих вышеперечисленных недостатков.

1.3. Основные требования к системе

1.3.1. Основные цели создания системы и критерии эффективности её функционирования

Основными целями создания системы являются:

1) предоставление пользователям простого и быстрого (в пределах нескольких минут) способа создавать и удалённо запускать учебные сервисы на вычислительных мощностях университета;

2) предоставление пользователям возможности осуществлять доступ к запущенным учебным сервисам как из университета, так и из любого места с подключением к сети Интернет;

3) предоставление преподавателям возможности выдавать студентам доступ к выделенным ресурсам на время прохождения учебного курса и отзывать этот доступ.

Критерии, определяющие эффективность функционирования системы:

1) соответствие требованиям. Система должна соответствовать требованиям и обеспечивать достижение поставленных целей;

2) надёжность. Система должна быть устойчива к ошибкам и успешно восстанавливаться после сбоев;

3) мобильность. Система должна быть кроссплатформенной и не должна быть привязана к одному поставщику вычислительных ресурсов, используемых для запуска учебных сервисов;

4) локализация. Система должна поддерживать выбор пользователем языка интерфейса;

5) расширяемость. Система должна быть пригодна для дальнейшего расширения при появлении новых функциональных требований;

6) масштабируемость. Система должны быть пригодна для горизонтального масштабирования путём одновременного запуска системы на нескольких серверах.

1.3.2. Функциональное назначение системы

Функции системы, предоставляемые пользователям в зависимости от их ролей показаны на рис. 1.6.

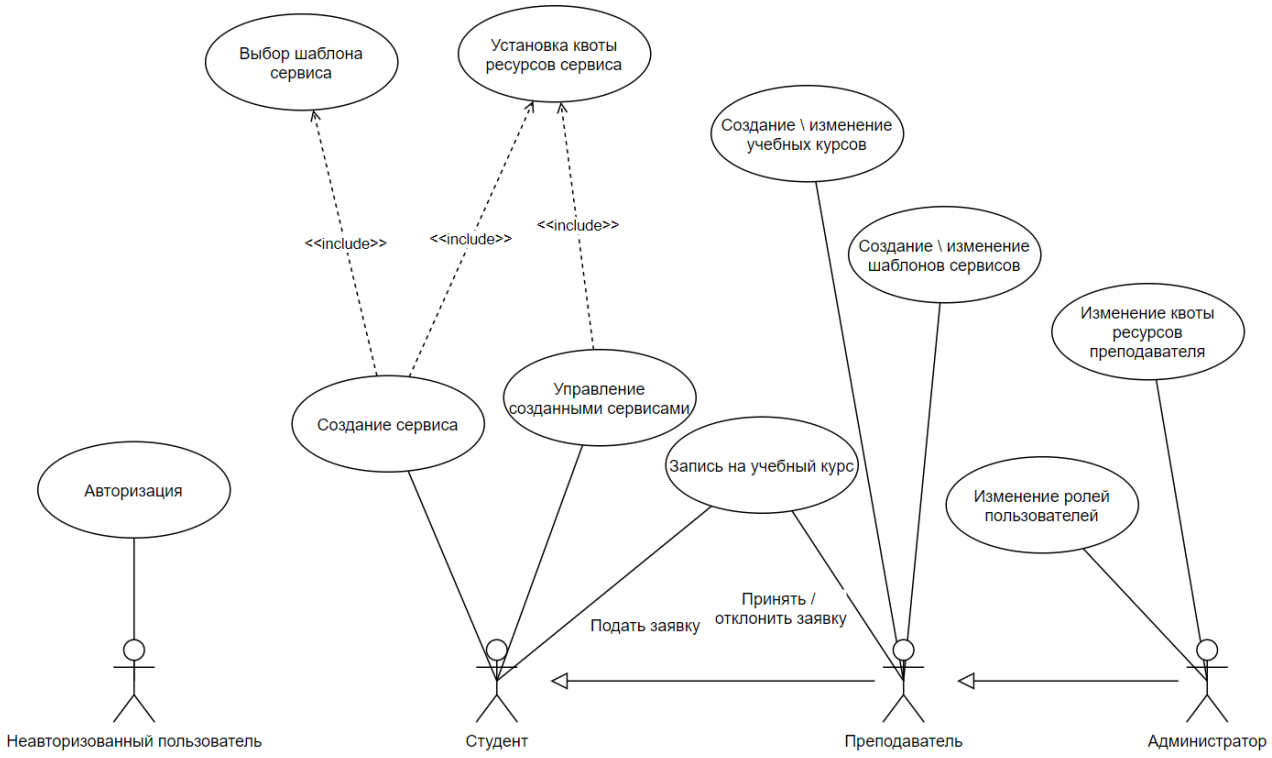


Рис. 1.6. Диаграмма вариантов использования в зависимости от роли пользователя

1.3.3. Основные особенности платформы и условия эксплуатации, определяющие основные требования к платформе

Пользователь платформы может использовать различные устройства для доступа к платформе, поэтому платформа должна быть выполнена в виде веб-приложения и не требовать установки на компьютер пользователя.

Операции создания и изменения учебных сервисов являются не столь частыми (ориентировочно, несколько таких операций в час во время проведения лабораторной работы для каждого студента). Неправильная проверка выделенной студенту квоты может привести к исчерпанию вычислительных ресурсов, которые используются для запуска учебных сервисов, что помешает нормальной работе всех студентов. Исходя из этих двух фактов, требуется отдать больший приоритет корректности работы операций, связанных с расчётом квот, а не скорости их выполнения.

1.3.4. Требования к функциональной структуре системы

Функциональная структура системы представлена на рис. 1.7.

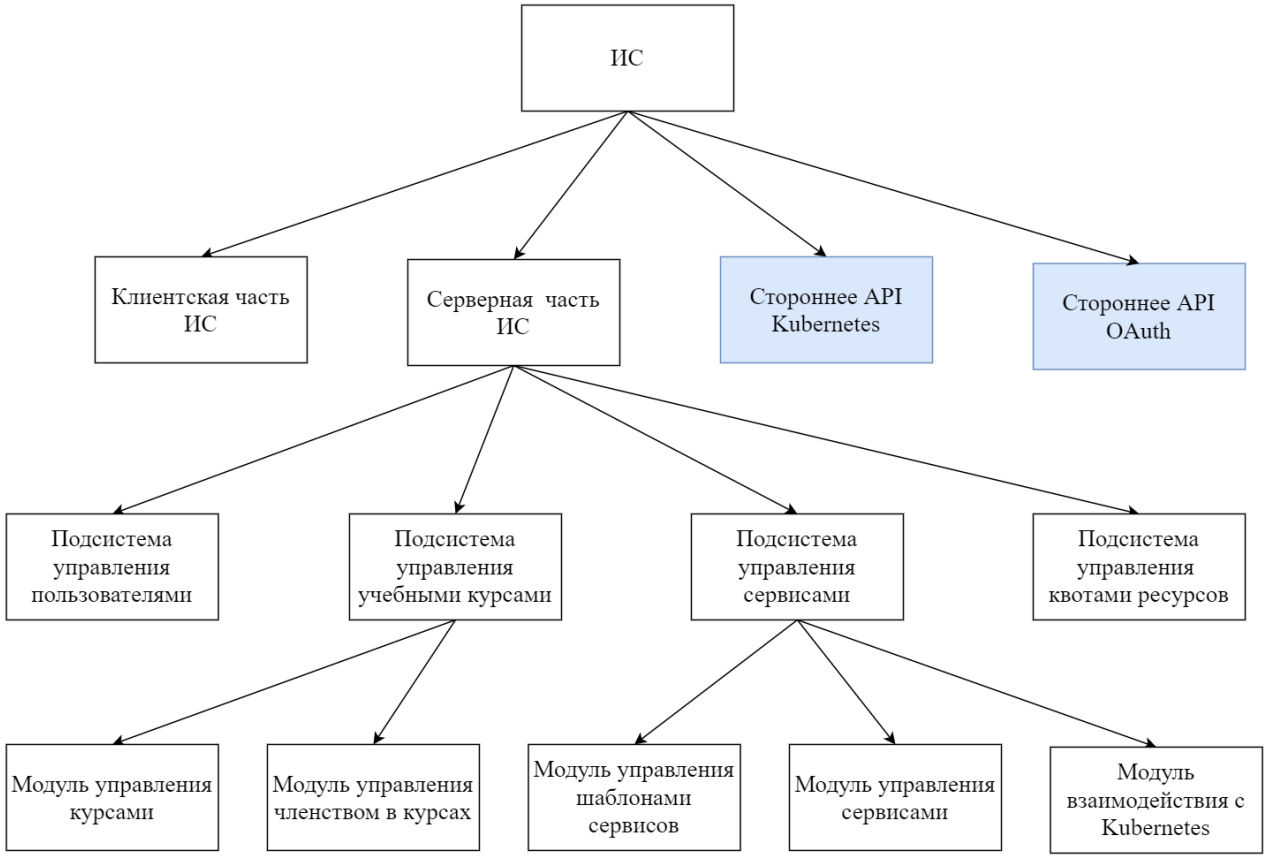


Рис. 1.7. Структурная схема системы

Система разделена на 4 основные части:

1) клиентская часть ИС. Это клиентское веб-приложение, которое позволяет пользователю интерактивно взаимодействовать с серверной частью с помощью графического интерфейса;

2) серверная часть ИС. Это ПО, запущенное на сервере, обрабатывающее запросы от клиентской части ИС.

3) стороннее API (программный интерфейс приложения) Kubernetes. Это стандартное API инфраструктурного компонента Kubernetes, и оно не является объектом разработки данной выпускной квалификационной работы. Этот интерфейс используется для создания и управления контейнерами, в которых запускаются сервисы, как было сказано выше в аналитическом обзоре;

4) стороннее API (программный интерфейс приложения) OAuth. Это стандартное API, описанное в RFC 6749 [3], RFC 6750 [4]. Оно используется для авторизации пользователей и получения основной информации о пользователе, такой как: ФИО, адрес электронной почты. Использование стандартного решения исключает необходимость повторной регистрации пользователей и обеспечивает возможность интеграции разрабатываемой системы в существующие системы управления пользователями, используемые в университете.

Серверная часть ИС в свою очередь состоит из следующих подсистем:

1) подсистема управления пользователями. Эта подсистема обеспечивает авторизацию пользователей и управление их ролями;

2) подсистема управления учебными курсами. Данная подсистема состоит из двух модулей:

2.1) модуль управления курсами, ответственный за создание и изменение учебных курсов;

2.2) модуль управления членством в курсах, ответственный за подачу, подтверждение, отклонение заявок на членство в учебных курсах, за проверку членства в курсе и за формирование списка студентов, зачисленных на курс;

3) подсистема управления сервисами. Эта подсистема обеспечивает выполнение операций над сервисами и состоит из следующих модулей:

3.1) модуль управления шаблонами сервисов, отвечающий за создание, изменение шаблонов сервисов и формированием заполненных шаблонов данных нужных для создания сервисов;

3.2) модуль управления сервисами, отвечающий за создание, изменение сервисов и сохранение заданной пользователем конфигурации в базу данных;

3.3) модуль взаимодействия с Kubernetes, который выполняет работу по преобразование заданной пользователем конфигурации в модуле управления сервисами в формат, понятный Kubernetes. Этот модуль отслеживает изменившиеся сервисы и применяет новую конфигурацию, используя стороннее API (программный интерфейс приложения) Kubernetes;

4) подсистема управления квотами ресурсов, отвечающая за выделение и освобождение квот ресурсов для преподавателей, студентов и сервисов;

1.3.5. Типовые проектные решения и пакеты прикладных программ, применяемые в системе

В качестве компонента для управления контейнерами используется Kubernetes. Для хранения данных выбрана NoSQL-база данных MongoDB из-за масштабируемости и удобства хранения иерархических структур, необходимых для расчёта квот ресурсов. Для серверной части используется язык Go, как язык, обеспечивающий кроссплатформенность и имеющий библиотеки для работы с API Kubernetes. Для клиентской части используется язык JavaScript с веб-фреймворком Svelte.

1.3.6. Требования к техническому обеспечению

Требования к техническому обеспечению не включают требования к вычислительным ресурсам, необходимые для запуска контейнеров с учебными сервисами, так как эта низкоуровневая функциональность предоставляется Kubernetes и не является частью данной выпускной квалификационной работы.

Минимальные требования к серверу:

1) двухъядерный процессор с частотой не менее 2.5 ГГц;

2) не менее 2 Гб ОЗУ;

3) не менее 2 Гб свободного места на жестком диске.

Минимальные требования к устройству пользователя:

1) процессор уровня Intel Pentium 4 или лучше;

2) не менее 1 Гб ОЗУ;

3) не менее 500 Мб свободного места на жестком диске.

1.3.7. Требования к информационному обеспечению

В состав информационного обеспечения входит NoSQL база данных MongoDB, которая должна быть сконфигурирована с поддержкой транзакций.

1.3.8. Требования к программному обеспечению

Программное обеспечение для сервера:

1) операционная система Windows 8.1 или выше, Linux с ядром версии 3.10 или выше, или FreeBSD 11 или выше;

2) MongoDB версии 4.0 или выше;

3) Node.js версии 13 или выше для запуска сервера клиентского приложения;

4) для работы серверного приложения на Go не требуется дополнительного ПО.

Программное обеспечение для компьютера пользователя:

1) современный браузер (Google Chrome версии 80 или выше, Safari версии 13 или выше, Firefox версии 74 или выше, или аналоги).

1.4. Основные технические решения проекта системы

1.4.1. Описание организации информационной базы

Информационная база представляет собой документы в NoSQL базе данных MongoDB. Схема организации документов представлена на рис. 1.8. Цветным заголовком выделены корневые документы, а белым – документы, которые включаются корневыми документами. Линиями показаны отношения ассоциации, т.е. ссылки из одного документа на другой. Стрелками показаны отношения композиции, т.е. включение одного документа в другой.

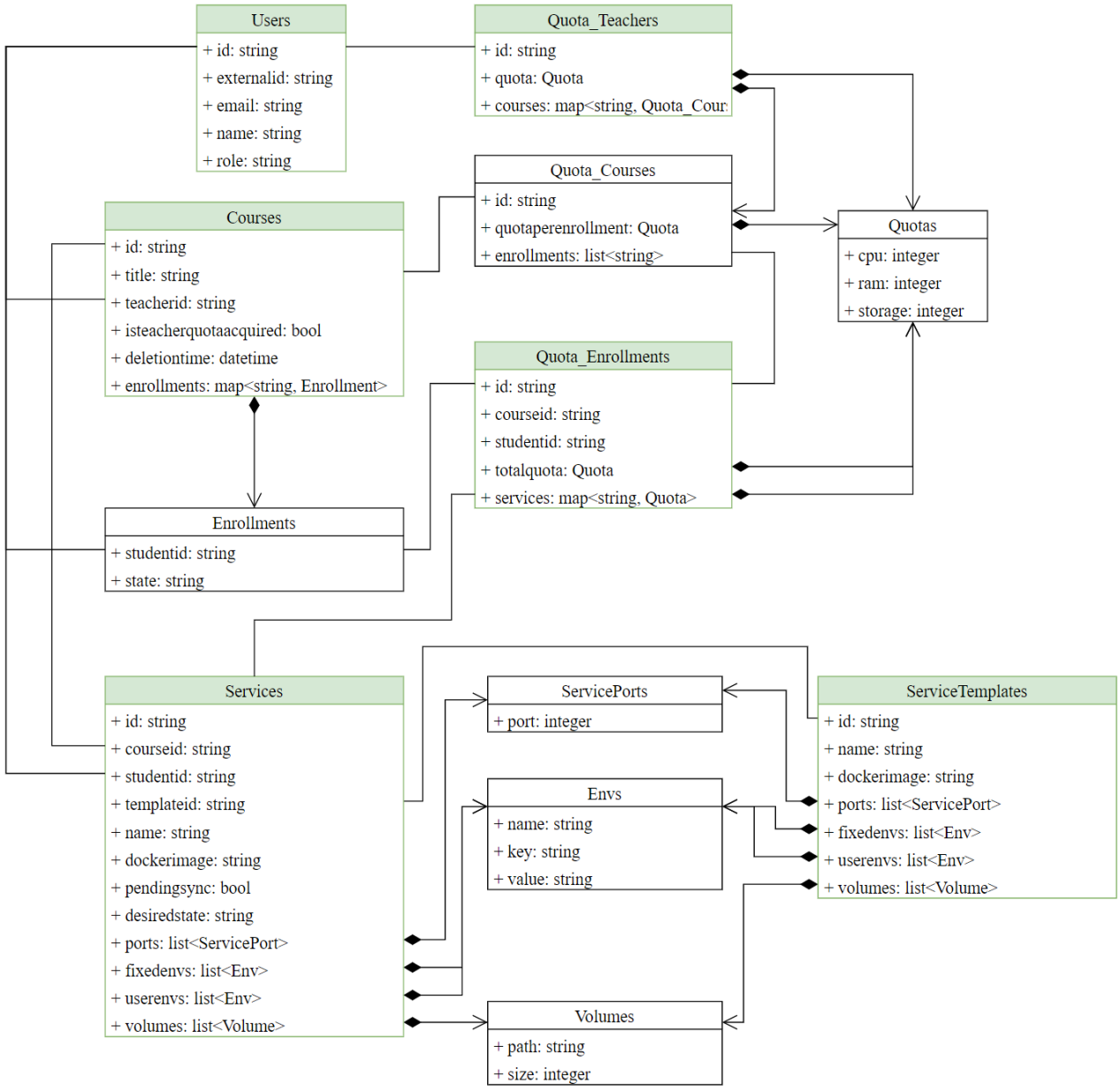


Рис. 1.8. Схема организации документов в базе данных

Корневой документ Users содержит информацию о пользователе:

1) id – внутренний идентификатор пользователя;

2) externalid – внешний идентификатор пользователя, получаемый из системы авторизации;

3) email – адрес электронной почты пользователя;

4) name – полное имя пользователя;

5) role – роль пользователя: «Admin» (администратор), «Teacher» (преподаватель), «Student» (студент).

Корневой документ Courses содержит информацию об учебных курсах:

1) id – внутренний идентификатор курса;

2) title – заголовок курса;

3) teacherid – идентификатор пользователя с ролью «преподаватель», который создал ответственен за этот курс;

4) isteacherquotaacquired – флаг, обозначающий, выделена ли квота ресурсов для самого преподавателя курса. Преподаватель может выделять и освобождать свою квоту для создания сервисов в рамках курса по необходимости;

5) deletiontime – дата, при наступлении которой курс будет автоматически завершен и данные будут удалены;

6) isactive – флаг, обозначающий, является ли курс активным или архивным;

7) enrollments – словарь вида «идентификатор студента – документ Enrollment», который содержит информацию о членстве студентов в курсе.

Документ Enrollment содержит информацию о членстве конкретного студента в курсе:

1) state – состояние членства в курсе: «нет», «ожидает подтверждения», «подтверждено».

Корневой документ Services содержит информацию о сервисах пользователей:

1) id – внутренний идентификатор сервиса;

2) courseid – идентификатор курса, в рамках которого создан сервис;

3) studentid – идентификатор студента, который создал сервис;

4) templateid – идентификатор шаблона сервиса, из которого был создан сервис;

5) name – название сервиса;

6) dockerimage – идентификатор Docker-образа сервиса;

7) pendingsync – флаг, обозначающий, если в конфигурации сервиса изменения, ещё не применённые в Kubernetes;

8) desiredstate – строка, обозначающее желаемое состояние сервиса: «started» (запущен), «stopped» (остановлен), «deleted» (удалён).

9) ports – список, содержащий документы ServicePort, определяющие сетевые порты сервиса, которые должны быть открыты извне;

10) fixedenvs – список, содержащий документы Envs, определяющие значения переменных окружения сервиса, фиксировано заданных в шаблоне сервиса;

11) userenvs – список, содержащий документы Envs, определяющие значения переменных окружения сервиса, значения которых заполняет пользователь;

12) volume – список документов Volume, определяющих пути монтирования сетевых дисков и их объёмы.

Корневой документ ServiceTemplates содержит информацию о сервисах пользователей:

1) id – идентификатор шаблона сервиса;

2) name – название шаблона сервиса;

3) dockerimage – идентификатор Docker-образа сервиса;

4) ports – список, содержащий документы ServicePorts, определяющие сетевые порты (поля port) сервиса, которые должны быть открыты извне;

5) fixedenvs – список, содержащий документы Envs, описывающие переменных окружения сервиса, фиксировано заданных в шаблоне сервиса;

6) userenvs – список, содержащий документы Envs, описывающие переменных окружения сервиса, значения которых заполняет пользователь;

7) volume – список документов Volumes, определяющих пути монтирования сетевых дисков и их объёмы.

Документ Envs содержит описание переменной окружения сервиса:

1) name – строка, содержащая понятное пользователю название переменной окружения;

2) key – строка, содержащая ключ переменной окружения;

3) value – строка, содержащая значение переменной окружения.

Документ Volumes описывает пути монтирования сетевых дисков и их объёмы:

1) path – строка, содержащая путь, по которому будет монтироваться сетевой диск;

2) size – число, задающее объём сетевого диска, в мегабайтах.

Корневой документ Quota_Teachers содержит информацию о квоте, отведенной преподавателя и о курсах, которые её используют:

1) id – внутренний идентификатор пользователя с ролью «преподаватель»;

2) quota – документ Quotas, содержащий информацию о выделенной квоте;

3) courses – словарь вида «идентификатор курса – документ Quota_Courses», содержащий информацию об курсах, использующих квоту преподавателя.

Документ Quotas содержит информацию о квоте ресурсов:

1) cpu - кол-во ядер ЦПУ, исчисляется в 1/1000 ядра;

2) ram - объем оперативной памяти, исчисляется в мегабайтах;

3) storage – объём сетевого хранилища, исчисляется в мегабайтах.

Документ Quota_Courses содержит информацию о квоте, отводимой на студентов, участвующих в курсе:

1) id – внутренний идентификатор курса;

2) quotaperenrollment – документ Quotas, содержащий информацию о том, какой объём квоты ресурсов выделяется каждому участнику курса;

3) enrollments – список идентификаторов документов Quota_Enrollment, содержащий информацию о потреблении ресурсов каждого участника курса.

Корневой документ Quota_Enrollments содержит информацию о потреблении ресурсов участника курса:

1) id – внутренний идентификатор квоты участника курса;

2) courseid – внутренний идентификатор курса;

3) studentid – внутренний идентификатор пользователя;

4) totalquota – документ Quotas, содержащий информацию о выделенной пользователю квоте;

5) services – словарь вида «идентификатор сервиса – документ Quotas», содержащий информацию о квотах, выделенных для каждого сервиса пользователя в этом в курсе.

1.4.2. Описание системы программного обеспечения

Схема компонентов системы представлена на рис. 1.9.

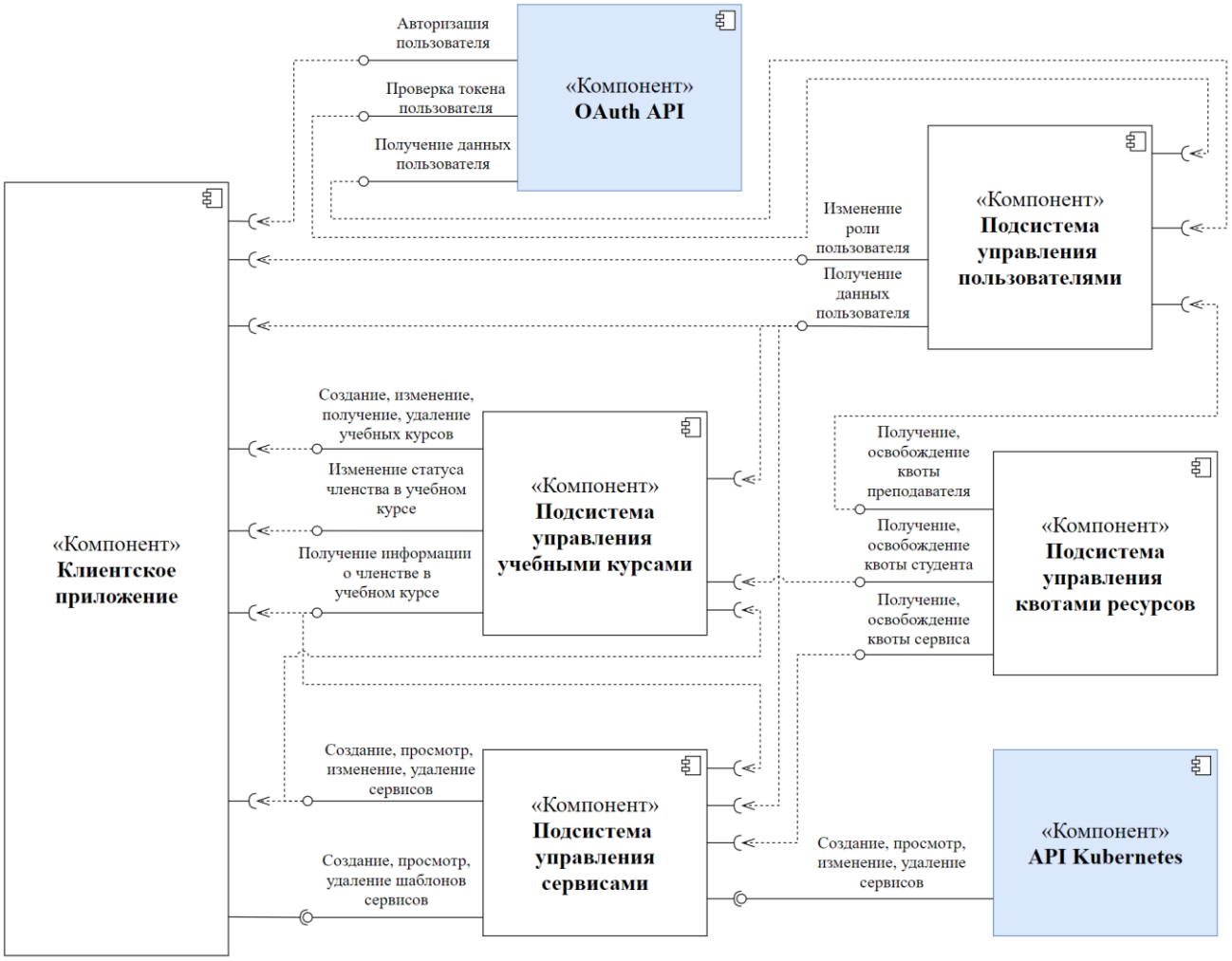


Рис. 1.9. Схема компонентов системы

Клиентское приложение – это та часть системы, с которой напрямую взаимодействует пользователь. Оно реализовано в виде Web-приложения на языке JavaScript с использованием фреймворка Svelte. В ответ на запросы пользователя клиентское приложение совершает запросы к серверным

подсистемам, получает от них данные и вставляет их в шаблоны HTML, тем самым формируя графический пользовательский интерфейс.

Для авторизации пользователя клиентское приложение совершает запросы во внешнее OAuth API. Это API показывает пользователю форму входа, общую для всех сервисов, и выдаёт пользователю токен доступа в случае успешной авторизации.

Полученный токен используется подсистемой управления пользователями для получения общих данных о пользователе и проверки подлинности. Кроме того, данная подсистема хранит роли пользователей и обрабатывает запросы на их изменение. При запросах на изменение доступной квоты учителя, эта подсистема совершает запрос в подсистему управления квотами ресурсов.

Запросы, связанные с просмотром и управлением учебными курсами и членством в учебных курсах, обрабатываются подсистемой управления учебными курсами. Она использует подсистему управления пользователями для получения ролей пользователей, подсистему управления сервисами для определения, имеются ли у пользователя сервисы, запущенные в рамках учебного курса, а также подсистему управления квотами ресурсов для выделения квот студентов.

Подсистема управления сервисами обрабатывает запросы, связанные с учебными сервисами. Она использует данные из подсистемы управления пользователями и подсистемы управления сервисами для определения того, имеет ли пользователь право на выполнение действия (является ли он учителем, состоит ли он в курсе, в рамках которого пытается создать сервис). После успешной проверки прав данная подсистема при необходимости производит выделение нужной квоты в подсистеме управления квотами ресурсов. Актуальное состояние сервисов хранится в Kubernetes, так как именно этот компонент отвечает за фактическую работу сервисов, поэтому данная подсистема совершает запросы на изменение и чтение данных в API Kubernetes.

Подсистема управления квотами ресурсов отвечает за выделение и освобождение квот ресурсов, предназначенных для преподавателей, студентов, и сервисов. Задача этой подсистемы – инкапсулировать логику учета квот ресурсов,

Все четыре серверных подсистемы представляют собой приложение, написанное на языке Go, разбитое на модули. В качестве хранилища данных в этих подсистемах используется NoSQL база данных MongoDB.

2. РАЗРАБОТКА ЗАДАЧ ОБЛАЧНОЙ ПЛАТФОРМЫ ДЛЯ СОЗДАНИЯ И УДАЛЁННОГО ЗАПУСКА УЧЕБНЫХ СЕРВИСОВ

2.1. Разработка задачи «подсистема управления пользователями»

2.1.1. Описание постановки задачи

2.1.1.1. Характеристика задачи

Данная подсистема предназначена для аутентификации пользователей, хранения их данных (уникальный идентификатор, полное имя, электронная почта) и управления их ролями, такими как: студент, преподаватель, администратор. Подсистема должна предоставлять следующие функции:

1) аутентификация пользователя. Эта функция будет широко использоваться в других подсистемах для ограничения доступа к функциям;

2) получение данных о пользователе по его уникальному идентификатору (полное имя, роль). Эта функция будет использоваться клиентским приложением для отображения данных о пользователе, в частности, при просмотре информации об участниках курса;

3) получение списка пользователей. Эта функция должна быть доступна только администраторам и будет использоваться клиентским приложением для отображения списка пользователей для последующего управления ими;

4) изменение роли пользователя. Эта функция должна быть доступна только администраторам и будет использоваться клиентским приложением при изменении роли пользователя администратором.

Поскольку функции получения данных по идентификатору и получения списка пользователей являются типовыми функциями получения данных из БД, они опущены в последующем описании алгоритмов.

2.1.1.2. Входная информация

В процессе работы подсистема использует данные, хранящиеся в БД.

Эти данные состоят из следующих полей:

1) id – строка, представляющая собой уникальный внутренний идентификатор пользователя. После чтения из БД эта строка преобразовывается в объект UUID – универсальный уникальный

идентификатор, описанный в RFC 4122 [5] с использованием библиотечных функций;

2) externalid – строка, представляющая собой внешний идентификатор пользователя, получаемый из внешней системы авторизации OAuth 2. Используется для сопоставления информации о пользователе, хранимой в БД, с информацией из внешней системы;

3) email – строка, хранящая адрес электронной почты пользователя;

4) name – строка, хранящая полное имя пользователя;

5) role – строка, хранящая роль пользователя: «Admin» (администратор), «Teacher» (преподаватель), «Student» (студент).

Запросы к подсистеме могут содержать уникальный идентификатор пользователя (поле id), поле role, задающую роль пользователя, а также OAuth2-токен, предназначенный для аутентификации (описание представлено в подпункте 2.1.2.1).

Функции, которые предназначены для вызова изнутри системы (из описываемой и других подсистем), получают данные с помощью аргументов функций стандартными средствами языка Go. Функции, которые предназначены для вызова извне системы, получают данные с помощью GET-параметров и тела HTTP запроса. В случае передачи данных в теле HTTP запроса, данные представляют собой объект в формате JSON.

2.1.1.3. Выходная информация

Функции, которые предназначены для вызова изнутри системы (из описываемой и других подсистем), возвращают данные с помощью стандартного способа возврата значений языка Go. Функции, которые предназначены для вызова извне системы, возвращают данные в теле HTTP ответа в формате JSON.

Выходная информация включает в себя один или несколько объектов, содержащих следующие поля:

1) id – строка, представляющая собой уникальный идентификатор пользователя;

2) name – строка, содержащая полное имя пользователя,

3) role – строка, хранящая роль пользователя: «Admin» (администратор), «Teacher» (преподаватель), «Student» (студент).

Выходная информация также может включать в себя объект ошибки, содержащий следующие поля:

1) code – строка, обозначающая тип ошибки;

2) message – строка с подробным описанием возникшей ошибки.

2.1.2. Описание алгоритма аутентификации пользователя

2.1.2.1. Назначение и характеристика алгоритма

Данный алгоритм предназначен для аутентификации пользователя и получения данных о нём. Алгоритм должен использовать протокол OAuth 2.0 для взаимодействия с сервером, отвечающим за аутентификацию пользователя. OAuth 2.0 – это широко применяемый стандарт аутентификации, описанный в RFC 6749 [3], RFC 6750 [4]. Процесс аутентификации состоит из 2 этапов, которые показаны на рис. 2.1.

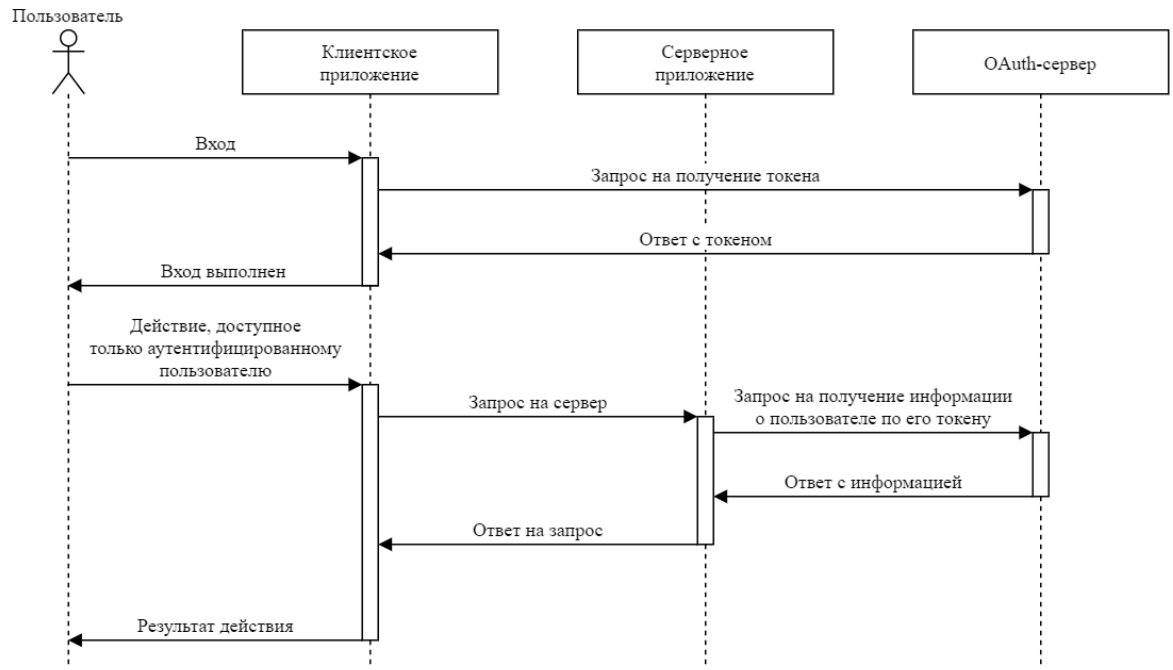


Рис. 2.1. Этапы аутентификации при использовании протокола OAuth 2.0

Вначале, при нажатии кнопки «Вход» пользователем, клиентское приложение отправляет запрос OAuth-серверу на получение специального

ключа – токена. При этом OAuth-сервер отвечает за проверку пароля пользователя, для чего клиентскому приложению может понадобиться показать пользователю страницу OAuth-сервера (для наглядности этот процесс пропущен на рис. 2.1 и OAuth-сервер сразу возвращает токен).

Токен представляет собой строку, которая каким-либо образом описывает идентификатор пользователя, время действия токена. Способ формирования токена не описан в стандарте и оставлен на усмотрение разработчиков OAuth-серверов: это может быть уникальный идентификатор записи в БД со всей нужной информацией, сами данные вместе с цифровой подписью, или что-либо иное.

Когда пользователь хочет выполнить действие, доступное только аутентифицированным пользователям, клиентское приложение включает в запрос (записывает в HTTP-заголовок Authorization) к серверному приложению токен, полученный ранее. Серверное приложение совершает запрос к OAuth-серверу с этим токеном и получает в ответ информацию о пользователе – его идентификатор, полное имя, адрес электронной почты или ошибку, если токен недействителен. Полученная информация может использоваться для дальнейшей проверки прав доступа.

В данном подразделе рассматривается только алгоритм аутентификации в серверном приложении, происходящий на втором этапе; первый этап реализован в клиентском приложении.

Поскольку в разрабатываемой системе присутствуют функции, требующие отображения имён других пользователей, а OAuth-сервер предоставляет доступ только к данным текущего пользователя, необходимо сохранять нужные данные каждого пользователя в собственной БД. Поэтому, при аутентификации пользователя в первый раз, подсистема должна сохранять в БД данные пользователя, полученные от OAuth-сервера.

2.1.2.2. Используемая информация

Алгоритм аутентификации использует специальный ключ – токен, которое клиентское приложение получает от OAuth-сервера и передаёт в запросах к серверному приложению в HTTP заголовке Authorization.

2.1.2.3. Результаты решения

Алгоритм всегда возвращает стандартными средствами языка программирования Go два значения – объект с данными о пользователе и объект ошибки. Объект с данными о пользователе представлен структурой Go, содержащей следующие поля:

1) id, уникальный идентификатор пользователя, представляющий собой UUID – универсальный уникальный идентификатор, описанный в RFC 4122 [5] и создаваемый библиотечными функциями;

2) email, строка, содержащая адрес электронной почты;

3) name, строка, содержащая полное имя пользователя;

4) role, строка, содержащее роль пользователя – одно из 3 значений: «Student» (студент), «Teacher» (преподаватель), «Admin» (администратор).

Объект ошибки представляет собой стандартный тип ошибки языка Go и содержит описание ошибки.

В случае успешной аутентификации объект с данными содержит данные пользователя: уникальный идентификатор, полное имя, адрес электронной почты, роль пользователя (администратор, преподаватель, студент), а объект ошибки содержит пустое значение (nil).

В случае, когда пользователь не аутентифицирован, объект с данными пользователя содержит пустое значение (nil), а объект ошибки содержит описание ошибки.

2.1.2.4. Алгоритм решения

Алгоритм состоит из следующих шагов:

1) Получение токена из HTTP заголовка Authorization.

2) Если токен получен, переход к пункту 3, иначе – к пункту 9.

3) Запрос к OAuth-серверу на получение информации о пользователе по его токену.

4) В случае, если запрос успешен, пользователь считается аутентифицированным, и осуществляется переход к пункту 5, иначе – переход к пункту 9.

5) Получение данных о пользователе в БД.

6) Если пользователь найден в БД, то переход к пункту 8, иначе необходимо создать в БД запись о нём, переход к пункту 7.

7) Создание в БД записи с информацией о пользователе. Пользователю присваивается роль «студент» и сохраняются его полное имя и адрес электронной почты, полученные от OAuth-сервера.

8) Пользователь аутентифицирован, возвращается объект с информацией о пользователе.

9) Пользователь не аутентифицирован, возвращается ошибка.

Схема данного алгоритма представлена на рис. 2.2.

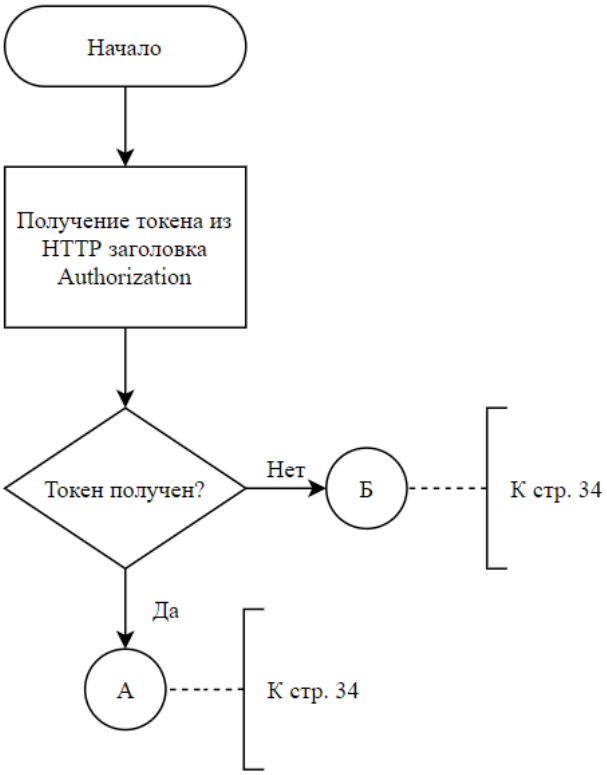
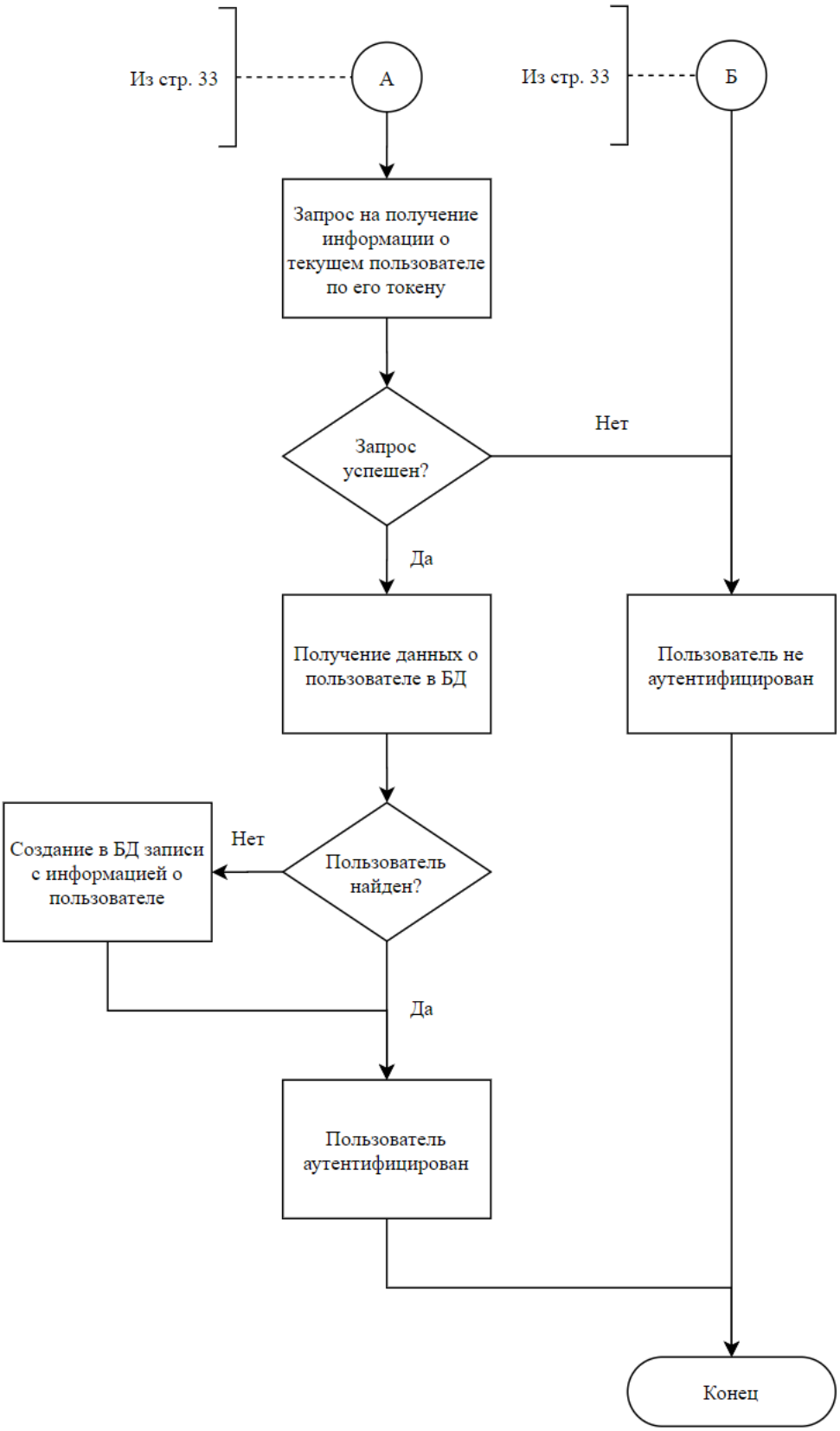


Рис. 2.2. Схема алгоритма аутентификации



Продолжение рис. 2.2

2.1.3. Описание алгоритма изменения роли пользователя

2.1.3.1. Назначение и характеристика алгоритма

Данный алгоритм предназначен для изменения роли пользователя. Необходимо учесть, что изменять роль может только пользователь с ролью администратора. Также, при изменении роли с роли студент на роль преподаватель или администратор необходимо выделять квоту ресурсов, а при изменении роли с роли преподаватели или администратора на роль студента – освобождать, что отображено на рис. 2.3.

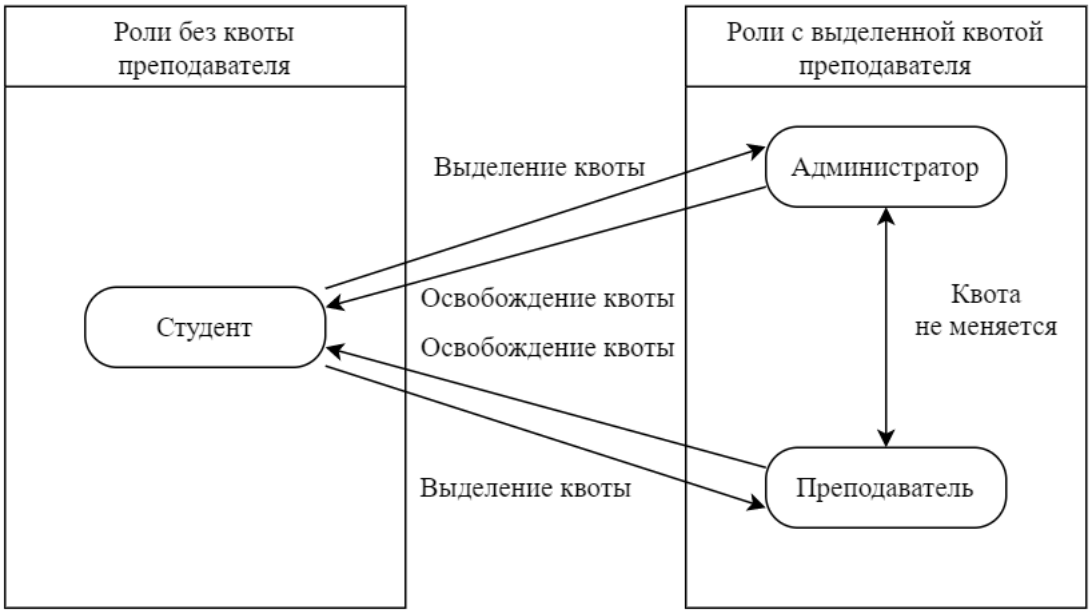


Рис. 2.3. Выделение и освобождение квоты при смене роли

2.1.3.2. Используемая информация

На вход алгоритм получает следующие данные:

1) токен текущего пользователя, используемый для аутентификации (передаётся в HTTP заголовке Authorization);

2) уникальный идентификатор пользователя, чью роль необходимо изменить (передаётся, как часть URL HTTP запроса);

3) новая роль пользователя (передаётся в виде объекта с единственным строковым полем role в формате JSON в теле HTTP запроса). Разрешённые значения: «Student» (студент), «Teacher» (преподаватель), «Admin» (администратор).

2.1.3.3. Результаты решения

В случае успешной смены роли, в БД сохраняется информация о новой роли пользователя и при необходимости происходит выделение или освобождение квоты ресурсов.

При неуспешной смене роли возвращается ошибка в теле HTTP-ответа в виде объекта в формате JSON с полями, описанными выше.

2.1.3.4. Алгоритм решения

Алгоритм состоит из следующих шагов:

1) Аутентификация пользователя. Для этого вызывается описанная ранее функция аутентификации подсистемы управления пользователями.

2) Проверка, имеет ли пользователь роль администратора. Если имеет, то переход к пункту 3, иначе к пункту 14.

3) Получение данных в БД о пользователе, роль которого нужно поменять, по его уникальному идентификатору, переданному на вход алгоритма.

4) Если пользователь есть в БД, то переход к пункту 5, иначе к пункту 15.

5) Если новая роль корректна, то переход к пункту 6, иначе к пункту 16.

6) Запись новой роли в объект пользователя.

7) Проверка, необходимо ли освободить квоту в результате смены роли с роли преподавателя или администратора на роль студента. Если необходимо, то переход к пункту 8, иначе к пункту 10.

8) Освобождение квоты. Для этого вызывается функция освобождения квоты преподавателя подсистемы управления квотами ресурсов.

9) Если освобождение квоты завершилось успешно, то переход к пункту 13, иначе к пункту 17.

10) Проверка, необходимо ли выделить квоту в результате смены роли с роли студента на роль преподавателя или администратора. Если необходимо, то переход к пункту 11, иначе к пункту 13.

11) Выделение квоты. Для этого вызывается функция выделения квоты преподавателя подсистемы управления квотами ресурсов.

12) Если выделение квоты завершилось успешно, то переход к пункту 13, иначе к пункту 17.

13) Сохранение данных в БД, завершение работы алгоритма.

14) У пользователя недостаточно прав для изменения роли пользователя, возвращается ошибка.

15) Указанный пользователь не существует в БД, возвращается ошибка.

16) Новая роль не является корректной, возвращается ошибка.

17) Изменение квоты завершилось неуспешно, возвращается ошибка.

Схема данного алгоритма представлена на рис. 2.4.

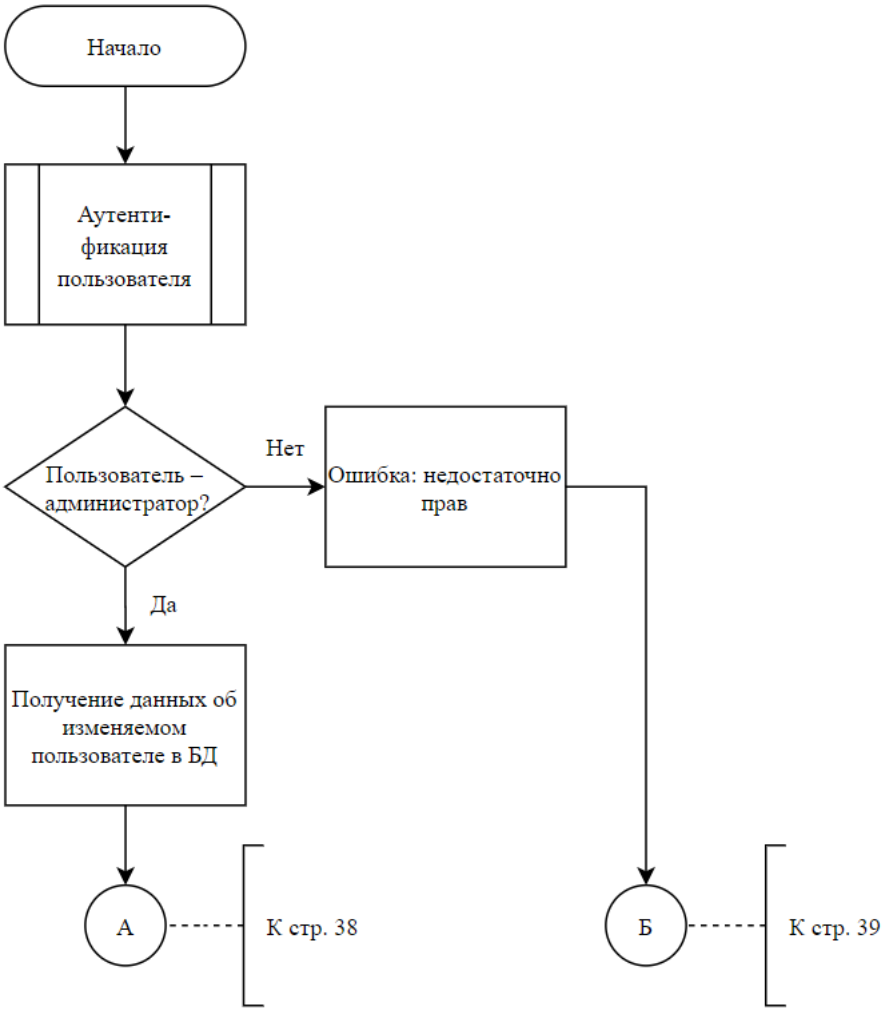
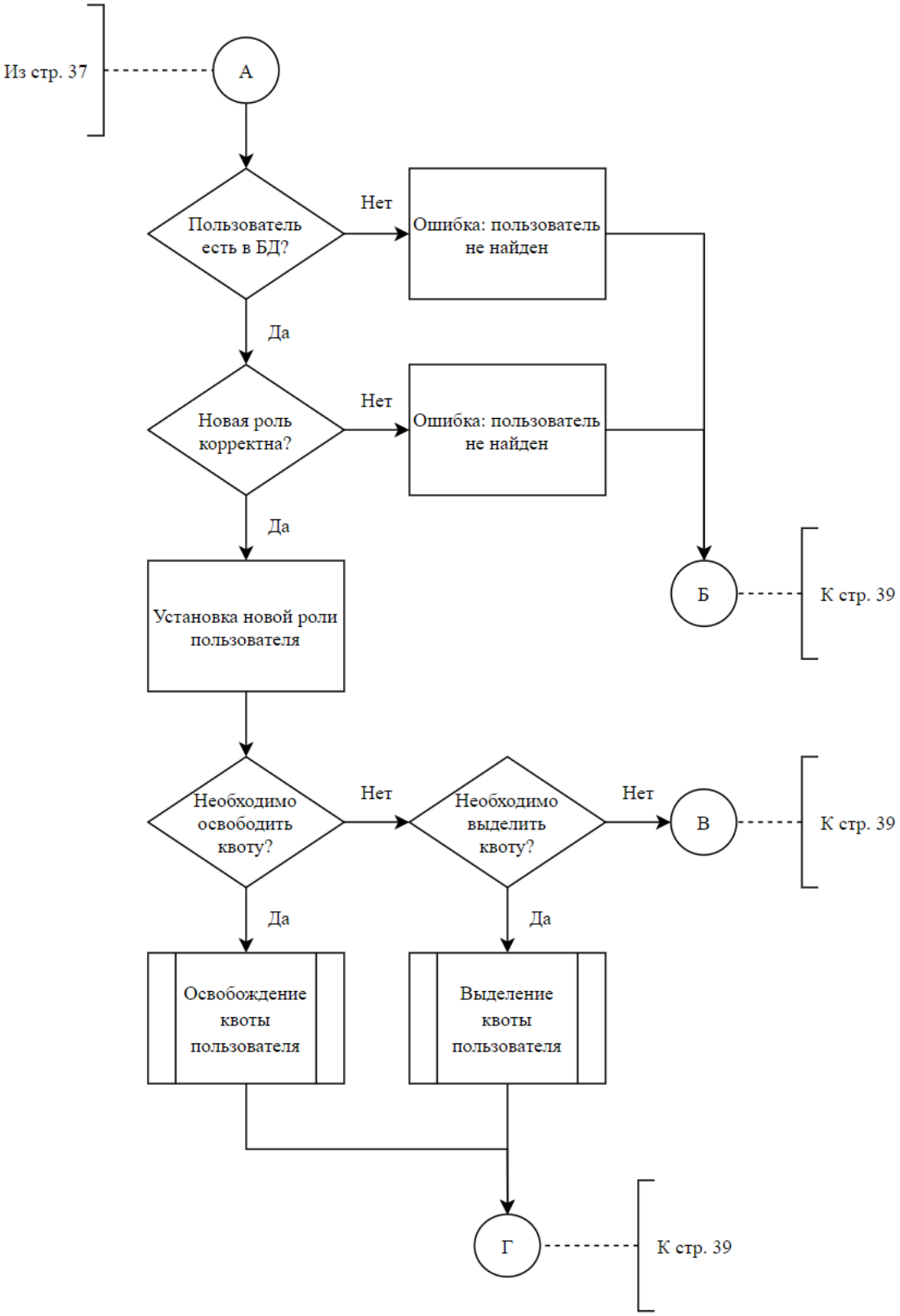
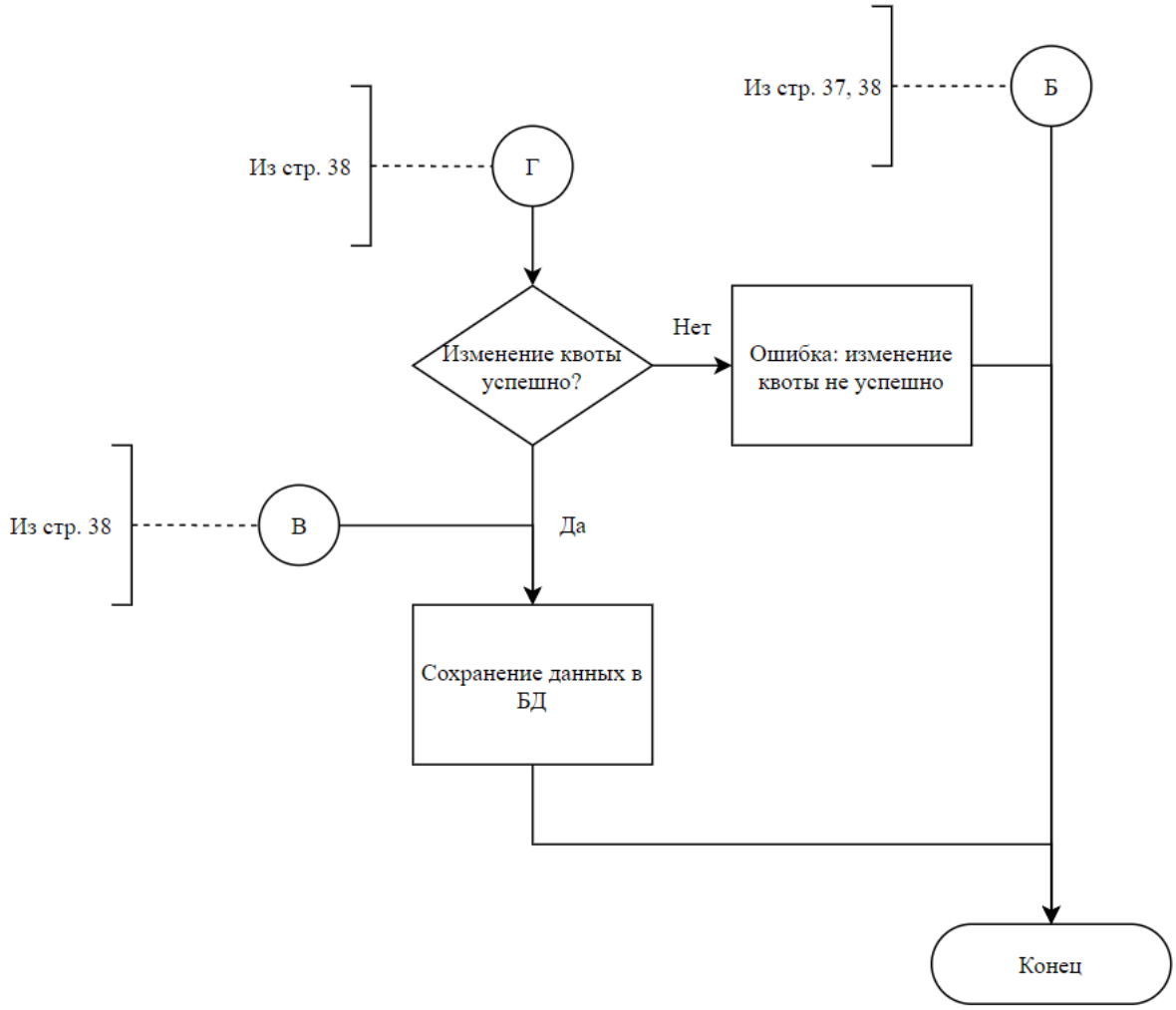


Рис. 2.4. Схема алгоритма изменения роли пользователя



Продолжение рис. 2.4



Продолжение рис. 2.4

2.2. Разработка задачи «подсистема управления учебными курсами»

2.2.1. Описание постановки задачи

2.2.1.1. Характеристика задачи

Данная подсистема предназначена для управления учебными курсами и членством в них. Учебный курс – это временное объединение студентов, которым преподавателем предоставлена квота серверных ресурсов. В начале семестра преподаватель создаёт учебный курс, связанный с дисциплиной и устанавливает дату окончания курса. По наступлению этой даты учебный курс автоматически удаляется и все занятые в рамках этого курса ресурсы освобождаются. Подсистема должна предоставлять следующие функции:

1) получение списка курсов. Эта функция будет использоваться клиентским приложением для отображения пользователю списка курсов для взаимодействия;

2) добавление нового курса. Эта функция должна быть доступна только преподавателям и администраторам и будет использоваться клиентским приложением при создании нового учебного курса;

3) получение информации о курсе. Эта функция будет использоваться клиентским приложением при отображении информации об отдельном курсе;

4) изменение курса. Эта функция должна быть доступна только преподавателю, который отвечает за курс, и администраторам и будет использоваться клиентским приложением при изменении параметров курса (названия, даты завершения);

5) удаление курса. Эта функция должна быть доступна только преподавателю, который отвечает за курс, и администраторам и будет использоваться клиентским приложением при удалении учебного курса;

6) получение списка участников курса. Эта функция должна быть доступна только преподавателю, который отвечает за курс, и администраторам и будет использоваться клиентским приложением при отображении списка участников курса и заявок на участие для дальнейшего взаимодействия;

7) добавление заявки на участие в курсе. Эта функция будет использоваться клиентским приложением при подаче пользователем заявки на участие в курсе;

8) подтверждение участия в курсе. Эта функция должна быть доступна только преподавателю, который отвечает за курс, и администраторам и будет использоваться клиентским приложением при подтверждении заявки на участие в курсе;

9) удаление участия в курсе. Эта функция будет использоваться клиентским приложением при отмене заявки пользователем и при отказе преподавателем в участии в курсе.

В дальнейшем будут рассмотрены только функции подтверждения участия в курсе и удаления курса, как наиболее нетривиальные.

2.2.1.2. Входная информация

В процессе работы подсистема использует данные, хранящиеся в БД. Схема организации этих данных представлена на рис. 2.5.

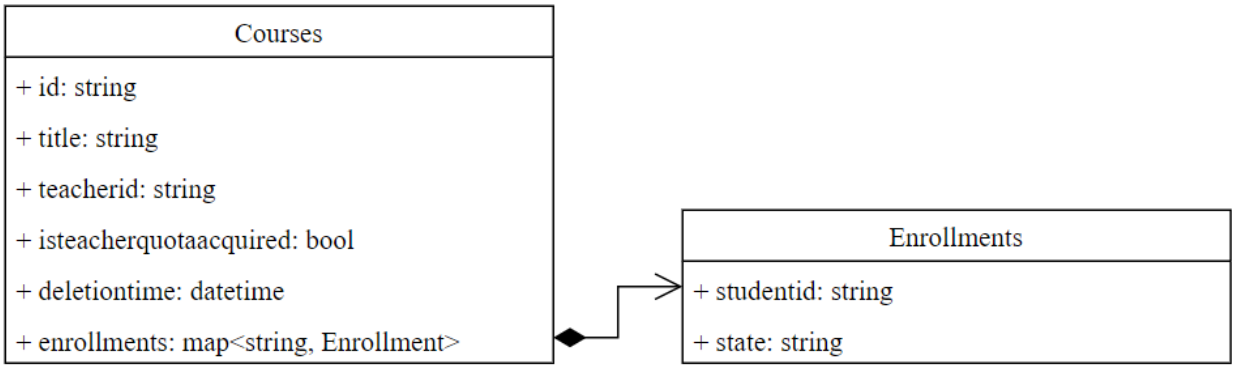


Рис. 2.5. Схема организации данных в БД

Courses является корневым документом и может включать в себя один или несколько документов Enrollments.

Документ Courses содержит информацию об учебных курсах:

1) id – строка, представляющая собой уникальный внутренний идентификатор курса. После чтения из БД эта строка преобразовывается в объект UUID – универсальный уникальный идентификатор, описанный в RFC 4122 [5] с использованием библиотечных функций;

2) title – строка, содержащая заголовок курса;

3) teacherid – строка, содержащая уникальный идентификатор пользователя с ролью «преподаватель», который ответственен за этот курс;

4) isteacherquotaacquired – флаг, обозначающий, выделена ли квота ресурсов для самого преподавателя курса. Преподаватель может выделять и освобождать свою квоту для создания сервисов в рамках курса по необходимости;

5) deletiontime – дата, при наступлении которой курс будет автоматически завершен и данные будут удалены;

6) enrollments – словарь вида «идентификатор студента – документ Enrollment», который содержит информацию о членстве студентов в курсе.

Документ Enrollment содержит информацию о членстве конкретного студента в курсе:

1) state – строка, описывающая состояние членства в курсе: «Pending» (ожидает подтверждения), «Approved» (подтверждено).

Запросы к подсистеме могут содержать уникальный идентификатор курса, уникальный идентификатор студента, а также OAuth2-токен, предназначенный для аутентификации (описание представлено в подпункте 2.1.2.1).

Запрос на создание курса должен включать в теле HTTP запроса объект в формате JSON со следующими полями:

1) title – строка, содержащая заголовок курса;

2) deletion_time – строка, содержащая дату в формате, определенном стандартом ISO 8601, например «2020-05-18T07:00:03Z»;

3) per_enrollment_quota – объект, содержащий информацию о квоте ресурсов, выделяемой каждому студенту курса:

3.1) cpu – целое число, обозначающее количество ядер ЦПУ, исчисляется в 1/1000 ядра;

3.2) ram – целое число, обозначающее объём оперативной памяти, исчисляется в мегабайтах;

3.3) storage – целое число, обозначающее объём сетевого хранилища, исчисляется в мегабайтах.

Запрос на изменение курса должен включать в теле HTTP запроса объект в формате JSON со следующими полями:

1) is_teacher_quota_acquired – флаг, который при наличии определяет, должна быть выделена квота для преподавателя в рамках курса или освобождена;

2) course_info – объект, который при наличии определяет, какие поля курса нужно изменить:

2.1) title - строка, содержащая заголовок курса;

2.2) deletion_time – строка, содержащая дату в формате, определенном стандартом ISO 8601, например «2020-05-18T07:00:03Z». При наступлении этой даты курс будет автоматически завершен и данные будут удалены.

2.2.1.3. Выходная информация

Функции возвращают данные в теле HTTP ответа в формате JSON.

Выходная информация может содержать один или несколько из следующих типов объектов CourseInfo, EnrollmentInfo.

Объект CourseInfo содержит информацию о курсе и включает в себя следующие поля:

1) id – строка, представляющая собой уникальный внутренний идентификатор курса;

2) title – строка, содержащая заголовок курса;

3) teacher_id – строка, содержащая уникальный идентификатор пользователя с ролью «преподаватель», который ответственен за этот курс;

4) is_teacher_quota_acquired – флаг, обозначающий, выделена ли квота ресурсов для самого преподавателя курса;

5) deletion_time – строка, содержащая дату в формате, определенном стандартом ISO 8601, например «2020-05-18T07:00:03Z». При наступлении этой даты курс будет автоматически завершен и данные будут удалены;

6) enrollment_info – объект типа EnrollmentInfo, включающий в себя информацию о статусе участия в курсе текущего пользователя.

Объект EnrollmentInfo содержит следующие поля:

1) user_id – строка, содержащая уникальный идентификатор пользователя;

2) state – строка, содержащая статус участия этого пользователя в курсе, принимает значения «None» (пользователь не участвует в курсе), «Pending» (пользователь подал заявку на участие в курсе, но её ещё не одобрили), «Approved» (пользователь подал заявку на участие в курсе и она была одобрена).

2.2.2. Описание алгоритма подтверждения участия в курсе

2.2.2.1. Назначение и характеристика алгоритма

Данный алгоритм предназначен для подтверждения участия студента в курсе. Необходимо учесть, что данное действие может совершить только преподаватель, ответственный за курс, или пользователь с ролью администратора.

При изменении статуса участия в курсе происходит выделение и освобождение квоты студента, как показано на рис. 2.6 (описываемый алгоритм выделен жирным).

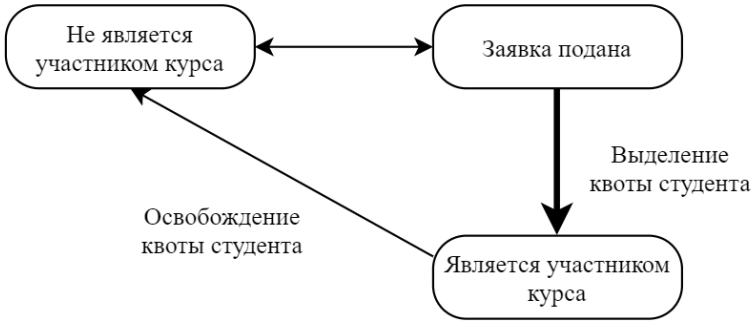


Рис. 2.6. Выделение и освобождение квоты при смене статуса участия

2.2.2.2. Используемая информация

На вход алгоритма получает следующие данные:

1) токен текущего пользователя, используемый для аутентификации (передаётся в HTTP заголовке Authorization);

2) уникальный идентификатор курса (передаётся как часть URL);

3) уникальный идентификатор студента (передаётся как часть URL);

4) объект в теле HTTP запроса в формате JSON, обозначающий тип действия и содержащий единственное строковое поле state с значением «Approved».

2.2.2.3. Результаты решения

В случае успеха, алгоритм возвращает пустой HTTP ответ с кодом ответа «200 OK». В случае возникновения ошибки алгоритм возвращает HTTP ответ

с объектом в формате JSON, содержащим поле message – строку, описывающую ошибку.

2.2.2.4. Алгоритм решения

Алгоритм состоит из следующих шагов:

1) Аутентификация пользователя. Для этого вызывается описанная ранее функция аутентификации подсистемы управления пользователями.

2) Проверка, может ли пользователь управлять курсом (имеет роль администратора или является преподавателем, отвечающим за курс). Если имеет, то переход к пункту 3, иначе к пункту 10.

3) Получение объекта курса из БД.

4) Если курс есть в БД, то переход к пункту 5, иначе к пункту 11.

5) Выделение квоты ресурсов студента. Для этого вызывается функция выделения квоты студента подсистемы управления квотами ресурсов.

6) Если выделение прошло успешно, то переход к пункту 7, иначе к пункту 12.

7) Подтверждение участия в курсе.

8) Если подтверждение успешно, то переход к пункту 9, иначе к пункту 13.

9) Сохранение данных в БД, алгоритм завершается и возвращает пустой HTTP ответ с кодом «200 OK».

10) У пользователя недостаточно прав для создания курса, возвращается ошибка.

11) Запрошенный курс не найден в БД, возвращается ошибка.

12) Не удалось выделить квоту, возвращается ошибка.

13) Не удалось подтвердить участие в курсе, возвращается ошибка.

Схема данного алгоритма представлена на рис. 2.7.

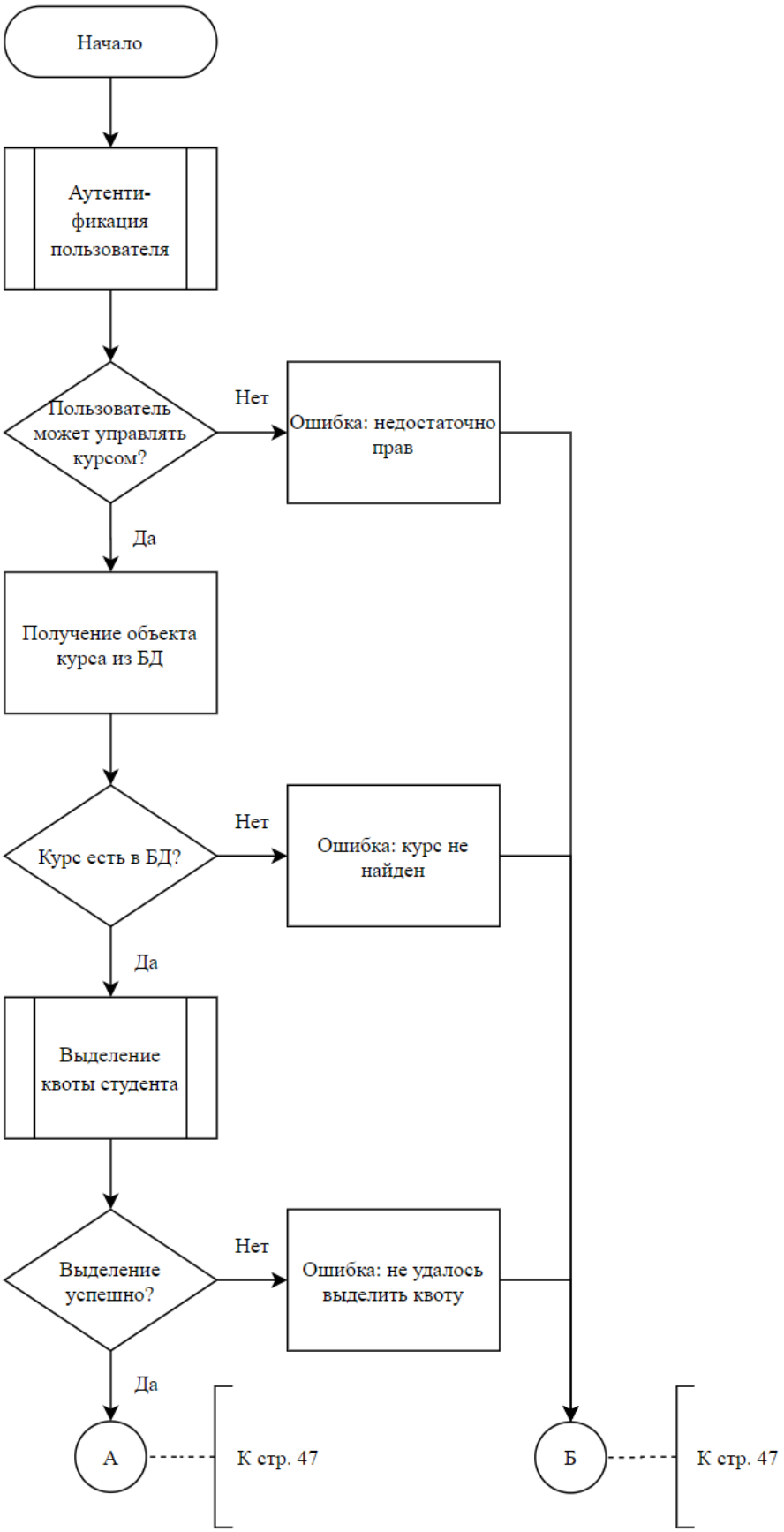
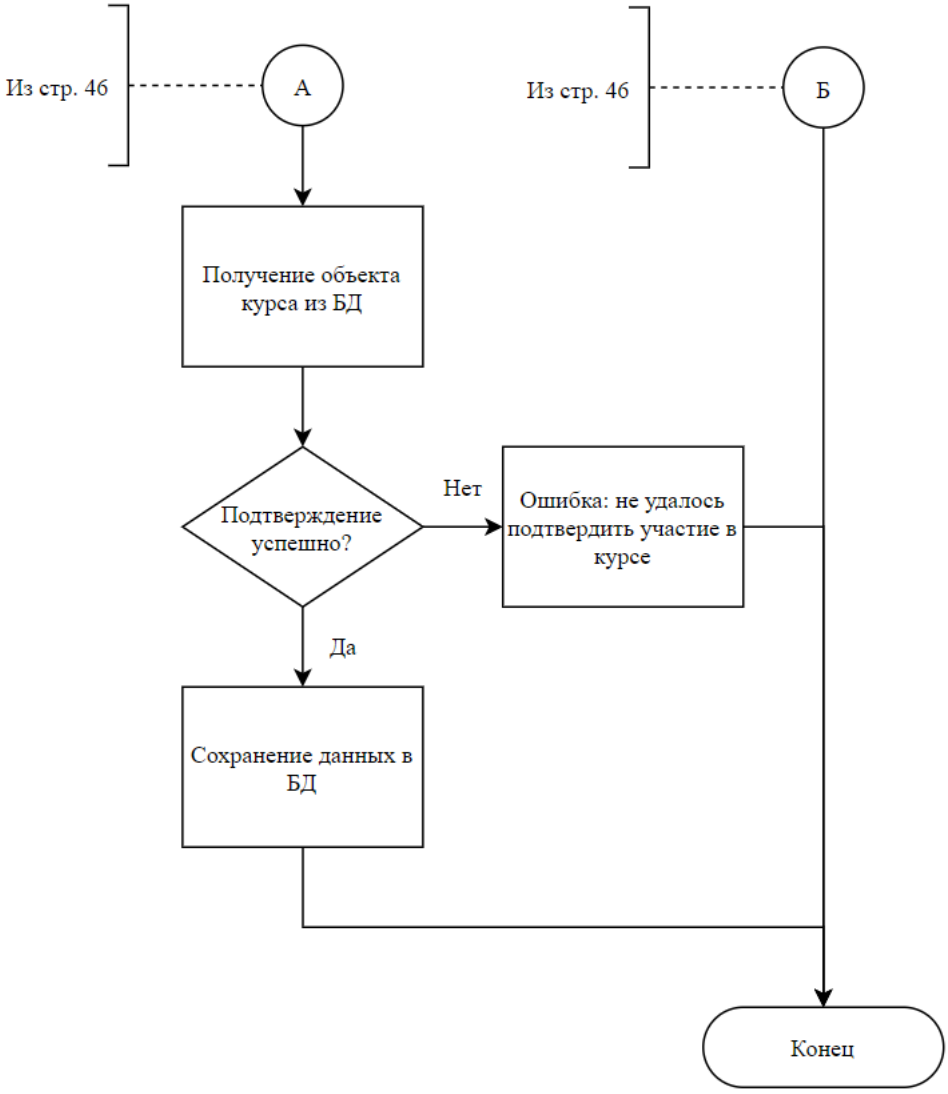


Рис. 2.7. Схема алгоритма подтверждения участия в курсе



Продолжение рис. 2.7

2.2.3. Описание алгоритма удаления курса

2.2.3.1. Назначение и характеристика алгоритма

Данный алгоритм предназначен для удаления учебного курса. Необходимо учесть, что удалить курс может только пользователь с ролью администратора или преподаватель, который отвечает за этот курс.

Учебный курс является корневым элементом иерархии, включающей в себя участников курса, их учебных сервисов, и квоты учебного курса, студентов, сервисов. Схема этой иерархии показана на рис. 2.8, где сплошными линиями показаны связи между объектами в рамках одной подсистемы, а пунктирными – между подсистемами. Учебный курс содержит некоторое количество участников курса и связан с квотой учебного курса в

подсистеме управления квотами ресурсов. Каждый участник курса являет создателем некоторого количества учебных сервисов и связан с квотой студента, выделенной в подсистеме управления квотами ресурсов. Каждый учебный сервис связан с квотой сервиса выделенной в подсистеме управления квотами ресурсов. И, наконец, квота сервиса зависит от квоты студента, и квота студента зависит от квоты учебного курса.

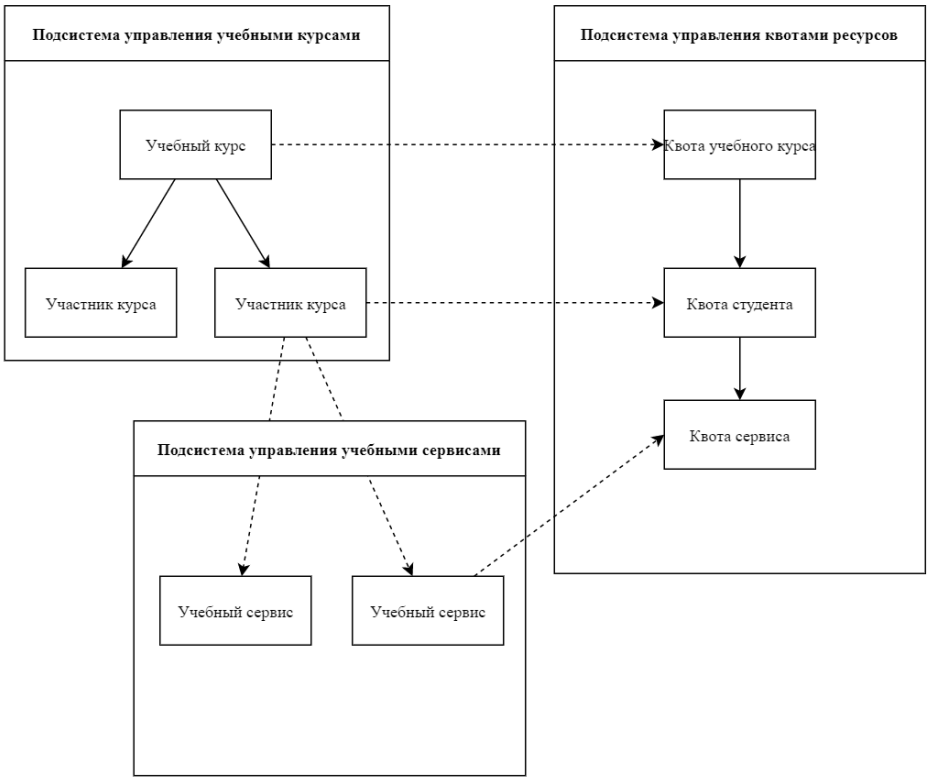


Рис. 2.8. Иерархия объектов, связанных с учебным курсом

При удалении курса необходимо произвести удаление всех связанных с ним объектов, для чего производятся запросы в другие подсистемы.

2.2.3.2. Используемая информация

На вход алгоритм получает следующие данные:

1) токен текущего пользователя, используемый для аутентификации (передаётся в HTTP заголовке Authorization);

2) уникальный идентификатор курса, который должен быть удалён (передаётся как часть URL HTTP запроса);

2.2.3.3. Результаты решения

В случае успеха, в БД удаляется информация об этом курсе и всех зависимых от него объектов, а алгоритм возвращает пустой HTTP ответ с кодом ответа «200 OK».

В случае возникновения ошибки алгоритм возвращает HTTP ответ с объектом в формате JSON, содержащим поле message – строку, описывающую ошибку.

2.2.3.4. Алгоритм решения

Алгоритм состоит из следующих шагов:

1) Аутентификация пользователя. Для этого вызывается описанная ранее функция аутентификации подсистемы управления пользователями.

2) Проверка, может ли пользователь управлять курсом (имеет роль администратора или является преподавателем, отвечающим за курс). Если может, то переход к пункту 3, иначе к пункту 13.

3) Получение объекта курса из БД.

4) Если курс есть в БД, то переход к пункту 5, иначе к пункту 14.

5) Начало цикла, если i меньше кол-ва участников курса, переход к пункту 6, иначе к пункту 9.

6) Удаление всех сервисов i-го участника курса. Для этого вызывается функция удаления сервисов подсистемы управления сервисами.

7) Освобождение квоты i-го участника курса. Для этого вызывается функция освобождения квоты участника курса подсистемы управления квотами ресурсов.

8) Конец цикла. Увеличение i на 1, переход к пункту 5.

9) Проверка, выделена ли квота преподавателя в данном курсе. Если да, то переход к пункту 10, иначе к пункту 12.

10) Удаление всех сервисов преподавателя курса. Для этого вызывается функция удаления сервисов подсистемы управления сервисами.

11) Освобождение квоты преподавателя. Для этого вызывается функция освобождения квоты участника курса подсистемы управления квотами ресурсов.

12) Освобождение квоты курса. Для этого вызывается функция освобождения квоты курса подсистемы управления квотами ресурсов.

13) Удаление курса из БД, алгоритм завершается и возвращает пустой HTTP ответ с кодом «200 OK».

14) Недостаточно прав для удаления курса, возвращается ошибка.

15) Курс не существует в БД, возвращается ошибка.

Схема данного алгоритма представлена на рис. 2.9.

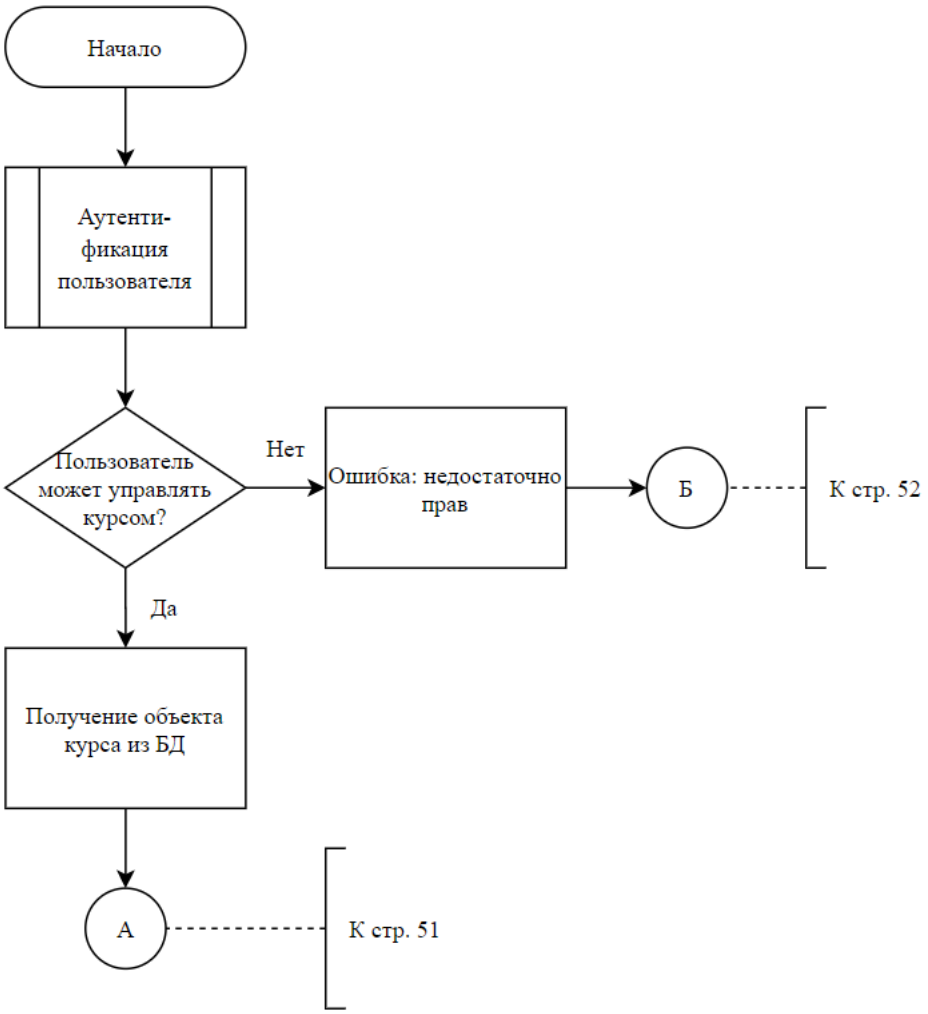
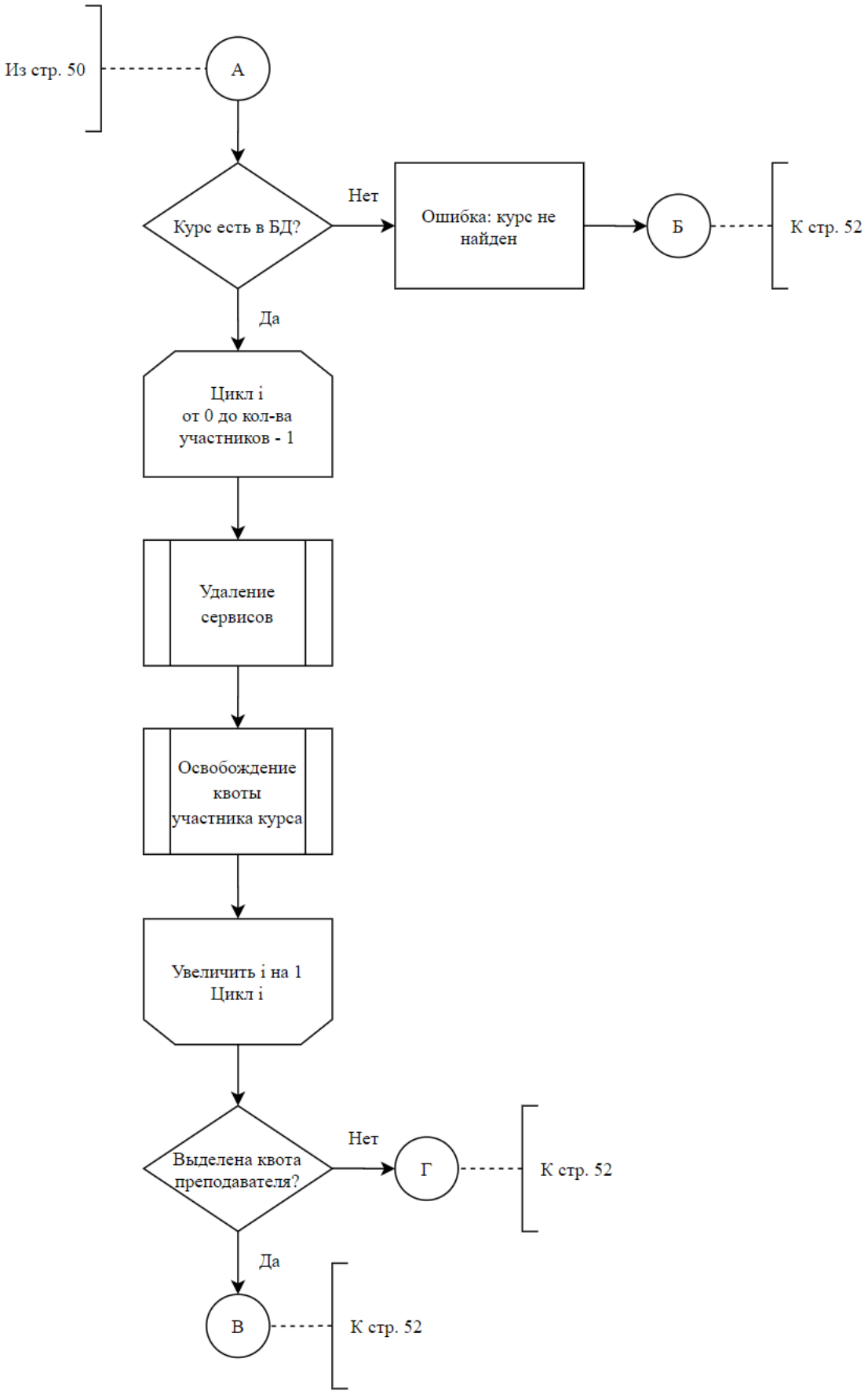
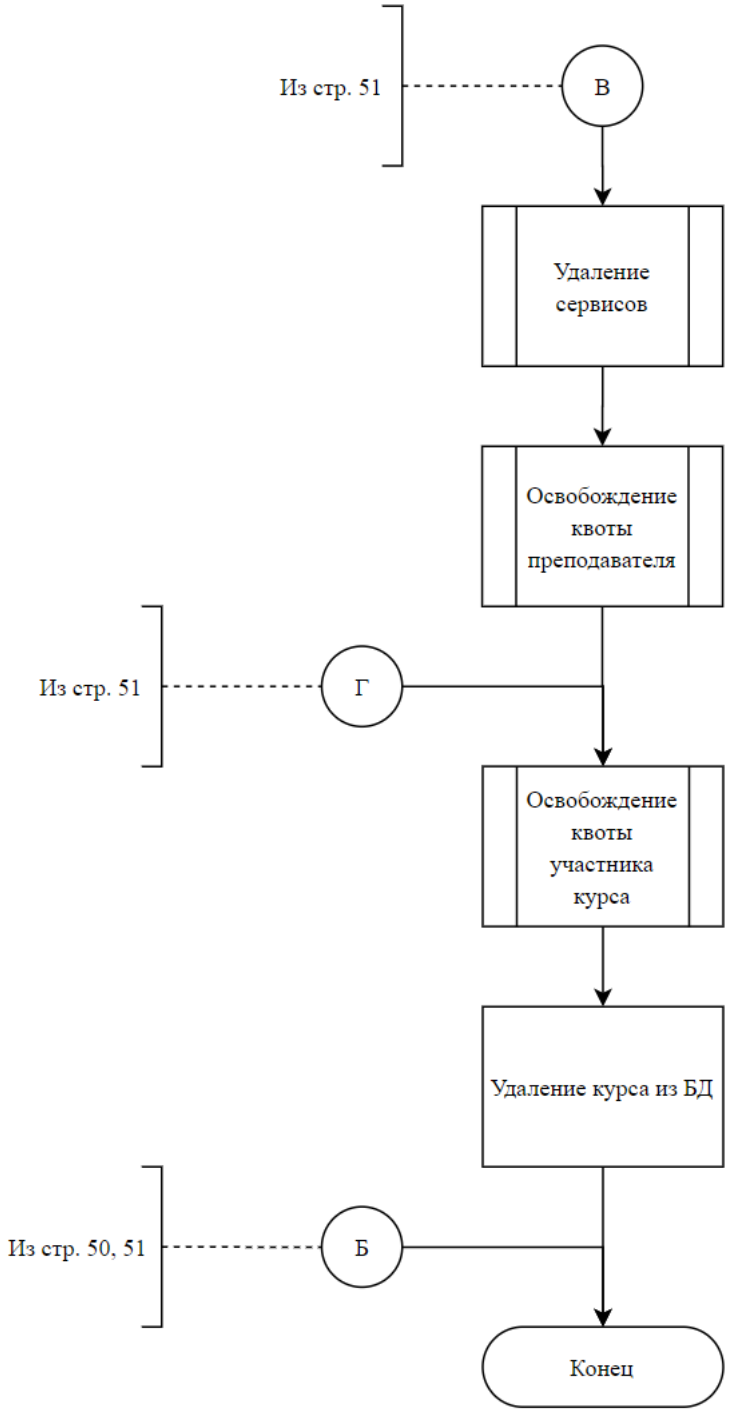


Рис. 2.9. Схема алгоритма удаления курса



Продолжение рис. 2.9



Продолжение рис. 2.9

2.3. Разработка задачи «подсистема управления квотами ресурсов»

2.3.1. Описание постановки задачи

2.3.1.1. Характеристика задачи

Данная подсистема предназначена для выделения и освобождения квот ресурсов. Квота ресурсов представляет собой часть серверных ресурсов, выделенную преподавателю, курсу, студенту или сервису. Серверные ресурсы

включают в себя: ядра процессора, оперативную память, место в сетевом хранилище данных). Подсистема должна предоставлять следующие функции:

1) получение текущей квоты преподавателя. Эта функция будет использоваться клиентским приложением для отображения доступной квоты преподавателя при создании нового курса и подтверждении заявки на участии в курсе;

2) выделение квоты преподавателя. Эта функция будет использоваться подсистемой управления пользователями при изменении роли пользователя;

3) освобождение квоты преподавателя. Эта функция будет использоваться подсистемой управления пользователями при изменении роли пользователя;

4) изменение квоты преподавателя. Эта функция должна быть доступна только администраторам и будет использоваться клиентским приложением при изменении количества доступной квоты преподавателя;

5) выделение квоты курса. Эта функция будет использоваться подсистемой управления курсами при создании нового курса;

6) освобождение квоты курса. Эта функция будет использоваться подсистемой управления курсами при удалении курса;

7) получение текущей квоты у студента в курсе. Эта функция будет использоваться клиентским приложением для отображения доступной квоты при создании нового сервиса;

8) выделение квоты студента в курсе. Эта функция будет использоваться подсистемой управления курсами при подтверждении заявки студента на участие в курсе;

9) освобождение квоты студента в курсе. Эта функция будет использоваться подсистемой управления курсами при исключении студента из курса;

10) выделение квоты сервиса. Эта функция будет использоваться подсистемой управления сервисами при создании нового сервиса;

11) освобождение квоты сервиса. Эта функция будет использоваться подсистемой управления сервисами при удалении сервиса;

12) изменение квоты сервиса. Эта функция будет использоваться подсистемой управления сервисами при изменении сервиса.

В дальнейшем будут рассмотрены только функции выделения квоты студента и выделения квоты сервиса, как наиболее нетривиальные.

2.3.1.2. Входная информация

В процессе работы подсистема использует данные, хранящиеся в БД. Схема организации этих данных представлена на рис. 2.10. В схеме есть два типа корневых документов, которые включают остальные – Quota_Teachers, описывающий квоту, выделенную преподавателю и Quota_Enrollments, описывающий квоту, выделенную студенту в рамках курса.

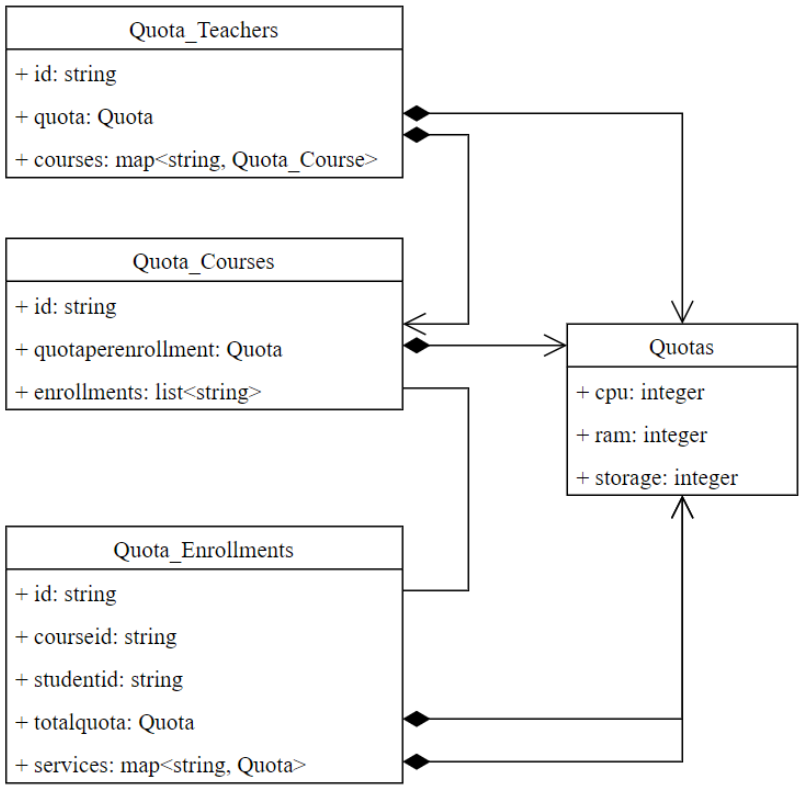


Рис. 2.10. Схема организации данных в БД

Корневой документ Quota_Teachers содержит информацию о квоте, отведенной преподавателя и о курсах, которые её используют:

1) id – строка, представляющая собой внутренний идентификатор пользователя с ролью «преподаватель»;

2) quota – документ Quotas, содержащий информацию о выделенной квоте;

3) courses – словарь вида «идентификатор курса – документ Quota_Courses», содержащий информацию об курсах, использующих квоту преподавателя.

Документ Quotas содержит информацию о квоте ресурсов:

1) cpu – целое число, обозначающее количество ядер ЦПУ, исчисляется в 1/1000 ядра;

2) ram – целое число, обозначающее объем оперативной памяти, исчисляется в мегабайтах;

3) storage – целое число, обозначающее объём сетевого хранилища, исчисляется в мегабайтах.

Документ Quota_Courses содержит информацию о квоте, отводимой на студентов, участвующих в курсе:

1) id – строка, внутренний идентификатор курса;

2) quotaperenrollment – документ Quotas, содержащий информацию о том, какой объём квоты ресурсов выделяется каждому участнику курса;

3) enrollments – список идентификаторов документов Quota_Enrollment, содержащий информацию о потреблении ресурсов каждого участника курса.

Корневой документ Quota_Enrollments содержит информацию о потреблении ресурсов участника курса:

1) id – строка, внутренний идентификатор квоты участника курса;

2) courseid – строка, внутренний идентификатор курса;

3) studentid – строка, внутренний идентификатор пользователя;

4) totalquota – документ Quotas, содержащий информацию о выделенной пользователю квоте;

5) services – словарь вида «идентификатор сервиса – документ Quotas», содержащий информацию о квотах, выделенных для каждого сервиса пользователя в этом в курсе.

Запросы к подсистеме могут содержать уникальный идентификатор курса, уникальный идентификатор студента, уникальный идентификатор сервиса, а также OAuth2-токен, предназначенный для аутентификации (описание представлено в подпункте 2.1.2.1).

Запросы на выделение и изменение квот получают данные с помощью аргументов функций стандартными средствами языка Go, и ожидают на вход объекты QuotaRequest со следующими полями:

1) cpu – целое число, обозначающее количество ядер ЦПУ, исчисляется в 1/1000 ядра;

2) ram – целое число, обозначающее объём оперативной памяти, исчисляется в мегабайтах;

3) storage – целое число, обозначающее объём сетевого хранилища, исчисляется в мегабайтах.

2.3.1.3. Выходная информация

Функции возвращают данные в теле HTTP ответа в формате JSON.

Выходная информация содержит объект, описывающий выделенную и используемую квоту и включающий в себя следующие поля:

1) cpu – целое число, обозначающее выделенное количество ядер ЦПУ, исчисляется в 1/1000 ядра;

2) ram – целое число, обозначающее выделенный объём оперативной памяти, исчисляется в мегабайтах;

3) storage – целое число, обозначающее выделенный объём сетевого хранилища, исчисляется в мегабайтах. Описание алгоритма подтверждения участия в курсе;

4) used_cpu – целое число, обозначающее использованное количество ядер ЦПУ, исчисляется в 1/1000 ядра;

5) used_ram – целое число, обозначающее использованный объём оперативной памяти, исчисляется в мегабайтах;

6) used_storage – целое число, обозначающее использованный объём сетевого хранилища, исчисляется в мегабайтах.

2.3.2. Описание алгоритма выделения квоты студента

2.3.2.1. Назначение и характеристика алгоритма

Данный алгоритм предназначен для выделения квоты студента на основе квоты учебного курса. Квота студента выделяется для каждого зачисленного участника курса и используется для последующего выделения квот сервисов, созданных участниками учебных курсов. Иерархия квот представлена на рис. 2.11. При выделении квоты должно проверяться наличие достаточного количества свободных ресурсов в вышестоящей квоте и при их отсутствии должна возвращаться ошибка.

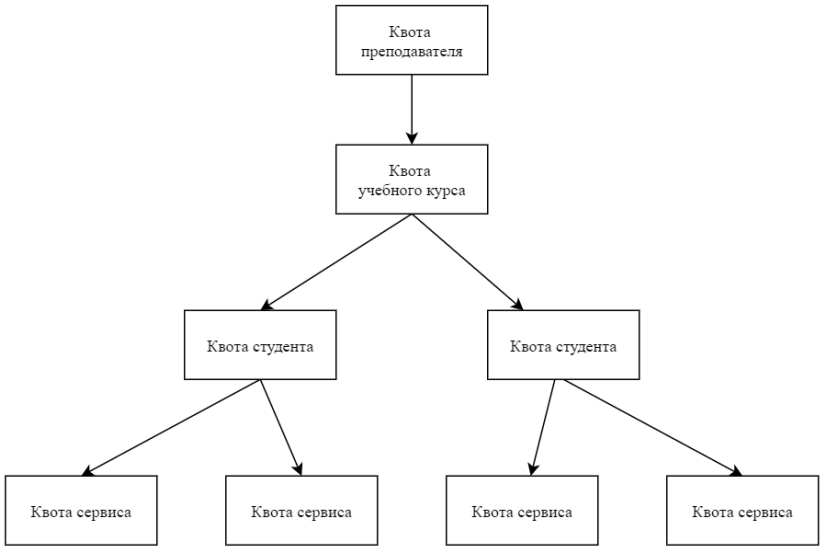


Рис. 2.11. Иерархия квот ресурсов

2.3.2.2. Используемая информация

На вход алгоритм получает следующие данные:

1) уникальный идентификатор курса, в рамках которого выделяется квота;

2) уникальный идентификатор пользователя, для которого выделяется квота.

2.3.2.3. Результаты решения

Алгоритм всегда возвращает объект ошибки стандартными средствами языка Go.

В случае успеха, объект ошибки принимает пустое значение (nil).

В случае возникновения ошибки, объект ошибки содержит описание ошибки.

2.3.2.4. Алгоритм решения

Алгоритм состоит из следующих шагов:

1) Получение объекта квоты преподавателя из БД по идентификатору курса.

2) Если квота преподавателя есть в БД, то переход к пункту 3, иначе к пункту 9.

3) Получение объекта квоты курса из объекта квоты преподавателя.

4) Если квота курса есть в квоте преподавателя, то переход к пункту 5, иначе к пункту 10.

5) Если квоты преподавателя достаточно для выделения квоты студента, то переход к пункту 6, иначе к пункту 11.

6) Если квота студента ещё не была выделена в этой квоте курса, то переход к пункту 7, иначе к пункту 12.

7) Создание объекта квоты студента.

8) Сохранение данных в БД, алгоритм завершается и возвращает пустой объект ошибки (nil).

9) Квота преподавателя не существует в БД, возвращается ошибка.

10) Квота курса не существует в квоте преподавателя, возвращается ошибка.

11) Квоты преподавателя недостаточно для выделения квоты студента, возвращается ошибка.

12) Квота студента уже была выделена в этом курсе, возвращается ошибка.

Схема данного алгоритма представлена на рис. 2.12.

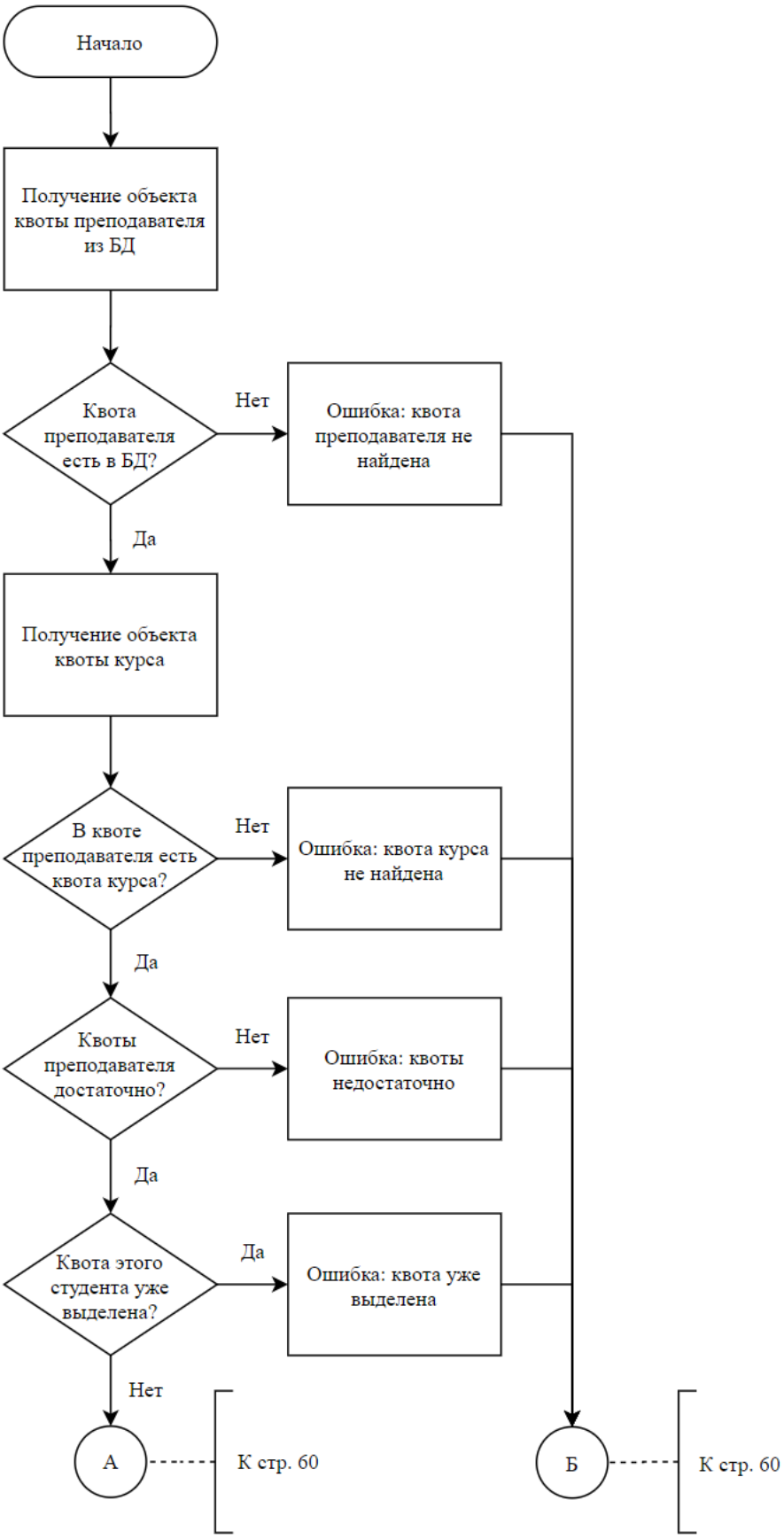
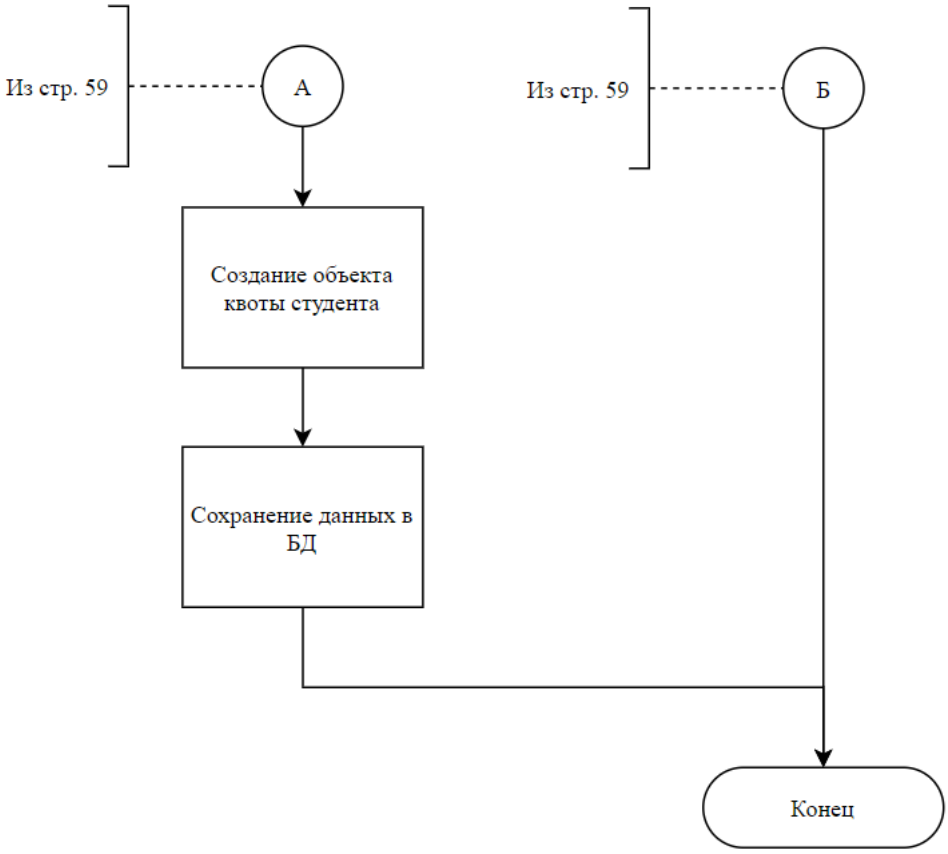


Рис. 2.12. Схема алгоритма выделения квоты студента



Продолжение рис. 2.12

2.3.3. Описание алгоритма выделения квоты сервиса

2.3.3.1. Назначение и характеристика алгоритма

Данный алгоритм предназначен для выделения квоты сервиса на основе квоты студента. Квота сервиса выделяется для каждого сервиса, созданного участниками учебных курсов. Иерархия квот представлена на рис. 2.13. При выделении квоты должно проверяться наличие достаточного количества свободных ресурсов в вышестоящей квоте и при их отсутствии должна возвращаться ошибка.

Рис. 2.13. Иерархия квот ресурсов

2.3.3.2. Используемая информация

На вход алгоритм получает следующие данные в качестве аргументов функции стандартными средствами языка Go:

1) уникальный идентификатор курса, в рамках которого выделяется квота;

2) уникальный идентификатор пользователя, от имени которого выделяется квота;

3) уникальный идентификатор сервиса, для которого выделяется квота;

4) объект типа QuotaRequest, описанный выше, содержащий информацию о размере квоты ресурсов, которую необходимо выделить.

2.3.3.3. Результаты решения

Алгоритм всегда возвращает объект ошибки стандартными средствами языка Go.

В случае успеха, объект ошибки принимает пустое значение (nil).

В случае возникновения ошибки, объект ошибки содержит описание ошибки.

2.3.3.4. Алгоритм решения

Алгоритм состоит из следующих шагов:

1) Получение объекта квоты студента из БД по идентификатору курса и идентификатору студента.

2) Если квота студента есть в БД, то переход к пункту 3, иначе к пункту 7.

3) Если квота сервиса ещё не была выделена в этой квоте студента, то переход к пункту 4, иначе к пункту 8.

4) Если квоты студента достаточно для выделения квоты сервиса, то переход к пункту 5, иначе к пункту 9.

5) Создание объекта квоты сервиса.

6) Сохранение данных в БД, алгоритм завершается и возвращает пустой объект ошибки (nil).

7) Квота студента не существует в БД, возвращается ошибка.

8) Квота сервиса уже была выделена в этой квоте студента, возвращается ошибка.

9) Квоты студента недостаточно для выделения квоты сервиса, возвращается ошибка.

Схема данного алгоритма представлена на рис. 2.14.

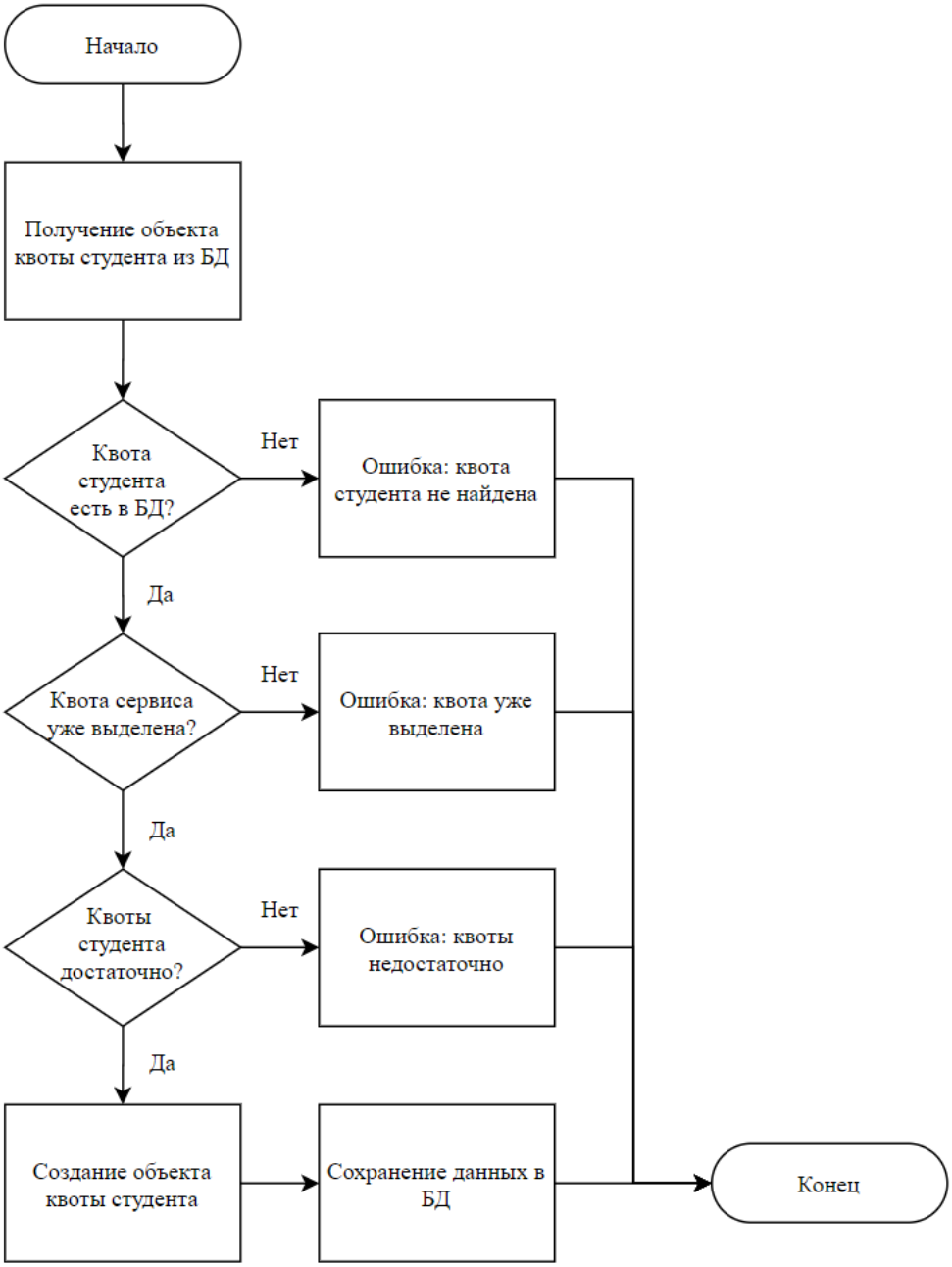


Рис. 2.14. Схема алгоритма выделения квоты сервиса

2.4. Разработка задачи «подсистема управления сервисами»

2.4.1. Описание постановки задачи

2.4.1.1. Характеристика задачи

Данная подсистема предназначена для создания учебных сервисов и управления ими. Учебные сервисы создаются из шаблонов с подстановкой пользовательских параметров, таких как: количество ресурсов, выделенных этому сервису, имя пользователя для подключения, пароль и другие (конкретный список параметров зависит от шаблона). Процессы сервисов запускаются в изолированных контейнерах с использованием Kubernetes – инфраструктурного компонента для управления контейнерами на нескольких серверах одновременно. Для этого изменившиеся в БД объекты сервисов проходят процесс синхронизации – обновления параметров сервиса в Kubernetes в соответствии с изменившимся объектом сервиса в БД. И, наоборот, для получения актуального, а не желаемого состояния сервиса подсистема совершает запросы к Kubernetes. Эти процессы отображены на рис. 2.15.

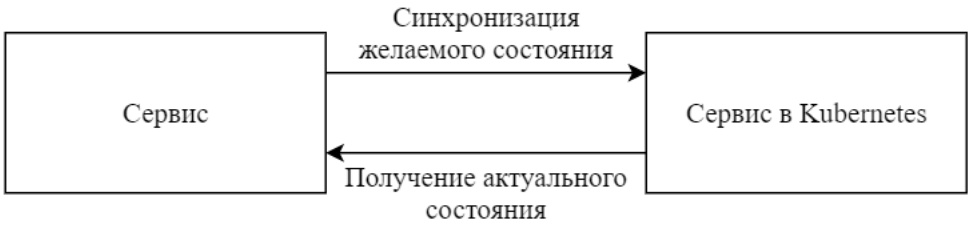


Рис. 2.15. Процессы взаимодействия с Kubernetes

Подсистема должна предоставлять следующие функции:

1) получение списка сервисов. Эта функция будет использоваться клиентским приложением для отображения списка сервисов, созданных пользователем в рамках курса;

2) создание нового сервиса. Эта функция будет использоваться клиентским приложением при создании нового сервиса пользователем;

3) получение информации о сервисе. Эта функция будет использоваться клиентским приложением для отображения информации о выбранном пользователем сервисе;

4) изменение сервиса. Эта функция будет использоваться клиентским приложением при внесении пользователем изменений в параметры сервиса;

5) удаление сервиса. Эта функция будет использоваться клиентским приложением при запросе пользователя на удаление сервиса, а также подсистемой управления учебными курсами при исключении студента из курса или удалении курса;

6) получение списка шаблонов сервисов. Эта функция будет использоваться клиентским приложением для отображения списка шаблонов, доступных для заполнения пользователю, при создании сервиса;

7) добавление шаблона сервиса. Эта функция должна быть доступна только преподавателям и администраторам. Она будет использоваться клиентским приложением при добавлении нового шаблона сервиса;

8) синхронизация параметров сервиса из БД в Kubernetes. Эта функция автоматически запускается серверным приложением для каждого сервиса с ещё не синхронизированными параметрами. Это необходимо для настройки сервисов в Kubernetes в соответствии с пожеланиями пользователя.

В дальнейшем будут рассмотрены только функции создания нового сервиса и синхронизации параметров сервиса из БД в Kubernetes, как наиболее нетривиальные.

2.4.1.2. Входная информация

В процессе работы подсистема использует данные, хранящиеся в БД. Схема организации этих данных представлена на рис. 2.16.

Корневой документ Services содержит информацию о сервисах пользователей:

1) id – строка, представляющая собой уникальный внутренний идентификатор сервиса;

2) courseid – строка, представляющая собой уникальный внутренний идентификатор курса, в рамках которого создан сервис;

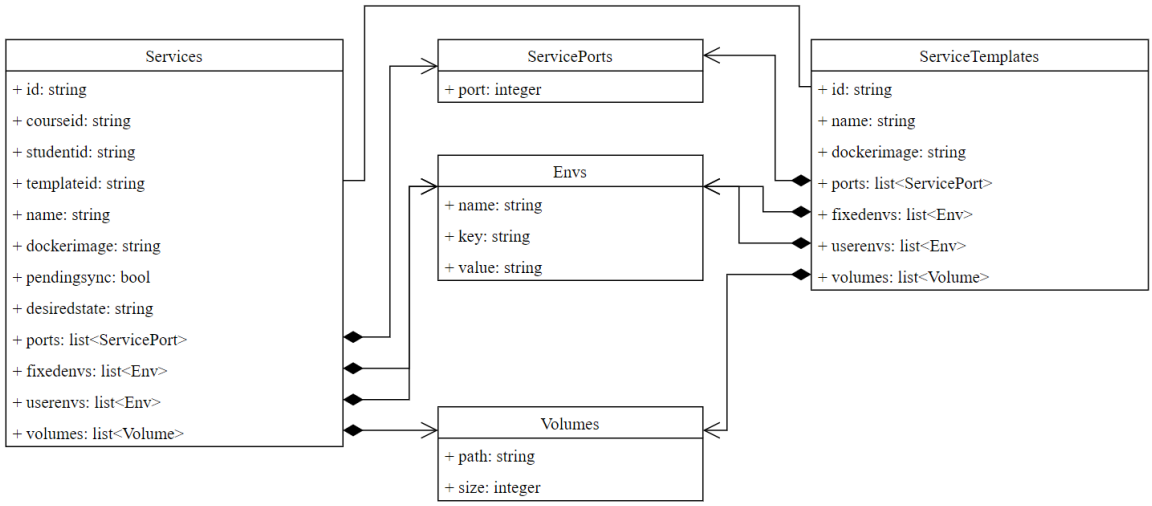


Рис. 2.16. Схема организации данных в БД

3) studentid – строка, представляющая собой уникальный внутренний идентификатор студента, который создал сервис;

4) templateid – строка, представляющая собой уникальный внутренний идентификатор шаблона сервиса;

5) name – строка, содержащая название сервиса;

6) dockerimage – строка, содержащая идентификатор Docker-образа сервиса;

7) pendingsync – флаг, обозначающий, если в конфигурации сервиса изменения, ещё не применённые в Kubernetes;

8) desiredstate – строка, обозначающее желаемое состояние сервиса: «started» (запущен), «stopped» (остановлен), «deleted» (удалён).

9) ports – список, содержащий документы ServicePorts, определяющие сетевые порты (поля port) сервиса, которые должны быть открыты извне;

10) fixedenvs – список, содержащий документы Envs, определяющие значения переменных окружения сервиса, фиксировано заданных в шаблоне сервиса;

11) userenvs – список, содержащий документы Envs, определяющие значения переменных окружения сервиса, значения которых заполняет пользователь;

12) volume – список документов Volumes, определяющих пути монтирования сетевых дисков и их объёмы.

Корневой документ ServiceTemplates содержит информацию о сервисах пользователей:

1) id – строка, представляющая собой уникальный внутренний идентификатор шаблона сервиса;

2) name – строка, содержащая название шаблона сервиса;

3) dockerimage – строка, содержащая идентификатор Docker-образа сервиса;

4) ports – список, содержащий документы ServicePorts, определяющие сетевые порты (поля port) сервиса, которые должны быть открыты извне;

5) fixedenvs – список, содержащий документы Envs, описывающие переменных окружения сервиса, фиксировано заданных в шаблоне сервиса;

6) userenvs – список, содержащий документы Envs, описывающие переменных окружения сервиса, значения которых заполняет пользователь;

7) volume – список документов Volumes, определяющих пути монтирования сетевых дисков и их объёмы.

Документ Envs содержит описание переменной окружения сервиса:

1) name – строка, содержащая понятное пользователю название переменной окружения;

2) key – строка, содержащая ключ переменной окружения;

3) value – строка, содержащая значение переменной окружения.

Документ Volumes описывает пути монтирования сетевых дисков и их объёмы:

1) path – строка, содержащая путь, по которому будет монтироваться сетевой диск;

2) size – число, задающее объём сетевого диска, в мегабайтах.

Запросы к подсистеме могут содержать уникальные идентификаторы сервиса, шаблона сервиса, курса, студента, а также OAuth2-токен, предназначенный для аутентификации (описание представлено в подпункте 2.1.2.1).

Запрос на создание шаблона сервиса должен включать в теле HTTP запроса объект типа ServiceTemplate в формате JSON со следующими полями:

1) name – строка, содержащая название шаблона сервиса;

2) docker_image – строка, содержащая идентификатор Docker-образа сервиса;

3) ports – массив целых чисел, содержащий номера портов, используемых сервисом;

4) fixed_envs – массив объектов типа Env, содержащий описание переменных окружения сервиса, фиксировано заданных в шаблоне сервиса;

5) user_envs – массив объектов типа Env, содержащий описание переменных окружения сервиса, значения которых должен будет заполнить пользователь при создании сервиса;

6) volumes – массив объектов типа Volume, содержащий описание путей монтирования сетевых дисков и их объёмы.

Поля объектов Env и Volume в точности соответствуют полям документов Envs и Volumes соответственно.

Запрос на создание сервиса должен включать в теле HTTP запроса объект в формате JSON со следующими полями:

1) name – строка, содержащая название сервиса;

2) template_id – строка, содержащая уникальный внутренний идентификатор шаблона сервиса, который будет использоваться при создании сервиса;

3) user_envs – массив объектов типа Env, содержащий описание переменных окружения сервиса, значения которых были заполнены пользователем при создании сервиса;

4) volumes – массив объектов типа Volume, содержащий описания путей монтирования сетевых дисков и объёмы дисков, выбранные пользователем при создании сервиса;

5) quota – объект типа QuotaRequest содержащий описание желаемой квоты ресурсов для создаваемого сервиса:

5.1) cpu – целое число, обозначающее количество ядер ЦПУ, исчисляется в 1/1000 ядра;

5.2) ram – целое число, обозначающее объём оперативной памяти, исчисляется в мегабайтах.

Запрос на изменение состояния сервиса должен включать в теле HTTP запроса объект в формате JSON со следующими полями:

1) user_envs - массив объектов типа Env, содержащий описание переменных окружения сервиса, значения которых заполнены пользователем;

2) state – строка, содержащее желаемое состояние сервиса: «started» (запущен), «stopped» (остановлен), «deleted» (удалён);

3) quota – объект типа QuotaRequest содержащий описание желаемой квоты ресурсов для сервиса.

2.4.1.3. Выходная информация

Функции возвращают данные в теле HTTP ответа в формате JSON. Выходная информация может содержать объект типа ServiceState, описывающий текущее состояние сервиса или объект типа ServiceTemplate, описанный выше.

Поля объекта ServiceState:

1) id – строка, представляющая собой уникальный внутренний идентификатор сервиса;

2) course_id – строка, представляющая собой уникальный внутренний идентификатор курса, в рамках которого создан сервис;

3) student_id – строка, представляющая собой уникальный внутренний идентификатор студента, который создал сервис;

4) template_id – строка, представляющая собой уникальный внутренний идентификатор шаблона сервиса;

5) name – строка, содержащая название сервиса;

6) docker_image – строка, содержащая идентификатор Docker-образа сервиса;

7) state – строка, обозначающее актуальное состояние сервиса: «started» (запущен), «stopped» (остановлен), «unkown» (неизвестно), «starting» (запускается), «stopped» (останавливается).

8) ports – массив целых чисел, содержащий номера портов, используемых сервисом;

9) fixed_envs – массив объектов типа Env, содержащий описание переменных окружения сервиса, фиксировано заданных в шаблоне сервиса;

10) user_envs – массив объектов типа Env, содержащий описание переменных окружения сервиса, значения которых были заполнены пользователем;

11) volumes – массив объектов типа Volume, содержащий описание путей монтирования сетевых дисков и их объёмы.

12) quota – объект типа Quota содержащий описание квоты ресурсов, выделенной для сервиса:

12.1) cpu – целое число, обозначающее количество ядер ЦПУ, исчисляется в 1/1000 ядра;

12.2) ram – целое число, обозначающее объём оперативной памяти, исчисляется в мегабайтах.

12.3) storage – целое число, обозначающее объём сетевого хранилища, исчисляется в мегабайтах.

13) port_mappings – массив объектов, описывающий соответствия между публичными сетевыми портами, доступными пользователю и внутренними портами сервиса:

13.1) service_port – целое число, содержащее номер сетевого порта сервиса;

13.2) external_port – целое число, содержащее номер публичного сетевого порта, по которому пользователем может быть произведен доступ к сетевому порту сервиса.

2.4.2. Описание алгоритма создания нового сервиса

2.4.2.1. Назначение и характеристика алгоритма

Данный алгоритм предназначен для создания нового учебного сервиса на основе шаблона, выбранного пользователем, и заполненных данных, таких как: название сервиса, квота ресурсов, пользовательские значения переменных окружения.При создании нового сервиса происходит выделение квоты сервиса с помощью запроса в подсистему управления квотами ресурсов и в случае успешного выделения объект сохраняется в БД для последующего сохранения в БД.

2.4.2.2. Используемая информация

На вход алгоритм получает следующие данные:

1) токен текущего пользователя, используемый для аутентификации (передаётся в HTTP заголовке Authorization);

2) уникальный идентификатор курса, в рамках которого создаётся сервис (передаётся как часть URL HTTP запроса);

3) уникальный идентификатор пользователя, от чьего имени создаётся сервис (передаётся как часть URL HTTP запроса). Преподаватель или администратор могут создавать сервисы от имени студентов при необходимости;

4) объект с описанием создаваемого сервиса с полями, описанными в подпункте 2.4.1.2 (передаётся в теле HTTP запроса в формате JSON).

2.4.2.3. Результаты решения

В случае успеха, в БД сохраняется информация о новом сервисе, выделяется квота ресурсов для этого сервиса и возвращается HTTP ответ с кодом «201 Created», содержащий в теле объект с единственным строковым полем id – уникальным идентификатором нового сервиса.

В случае возникновения ошибки алгоритм возвращает HTTP ответ с объектом в формате JSON, содержащим поле message – строку, описывающую ошибку.

2.4.2.4. Алгоритм решения

Алгоритм состоит из следующих шагов:

1) Аутентификация пользователя. Для этого вызывается описанная ранее функция аутентификации подсистемы управления пользователями.

2) Проверка, может ли пользователь создавать сервисы в данном курсе от имени данного пользователя. Если может, то переход к пункту 3, иначе к пункту 10.

3) Получение объекта шаблона сервиса из БД.

4) Если шаблон сервиса есть в БД, то переход к пункту 5, иначе к пункту 11.

5) Проверка, заполнены ли корректно пользователем значения шаблона. Если все корректно, то переход к пункту 6, иначе к 12.

6) Создание объекта сервиса на основе шаблона и значений, заполненных пользователем.

7) Выделение квоты сервиса. Для этого вызывается описанная ранее функция выделения квоты сервиса подсистемы управления квотами ресурсов.

8) Если выделение прошло успешно, то переход к пункту 9, иначе к 13.

9) Сохранение данных в БД, алгоритм завершается и возвращает HTTP ответ с идентификатором созданного сервиса.

10) Недостаточно прав, возвращается ошибка.

11) Указанный шаблон сервиса не существует, возвращается ошибка.

12) Шаблон заполнен некорректно, возвращается ошибка.

13) Не удалось выделить квоту ресурсов, возвращается ошибка.

Схема данного алгоритма представлена на рис. 2.17.

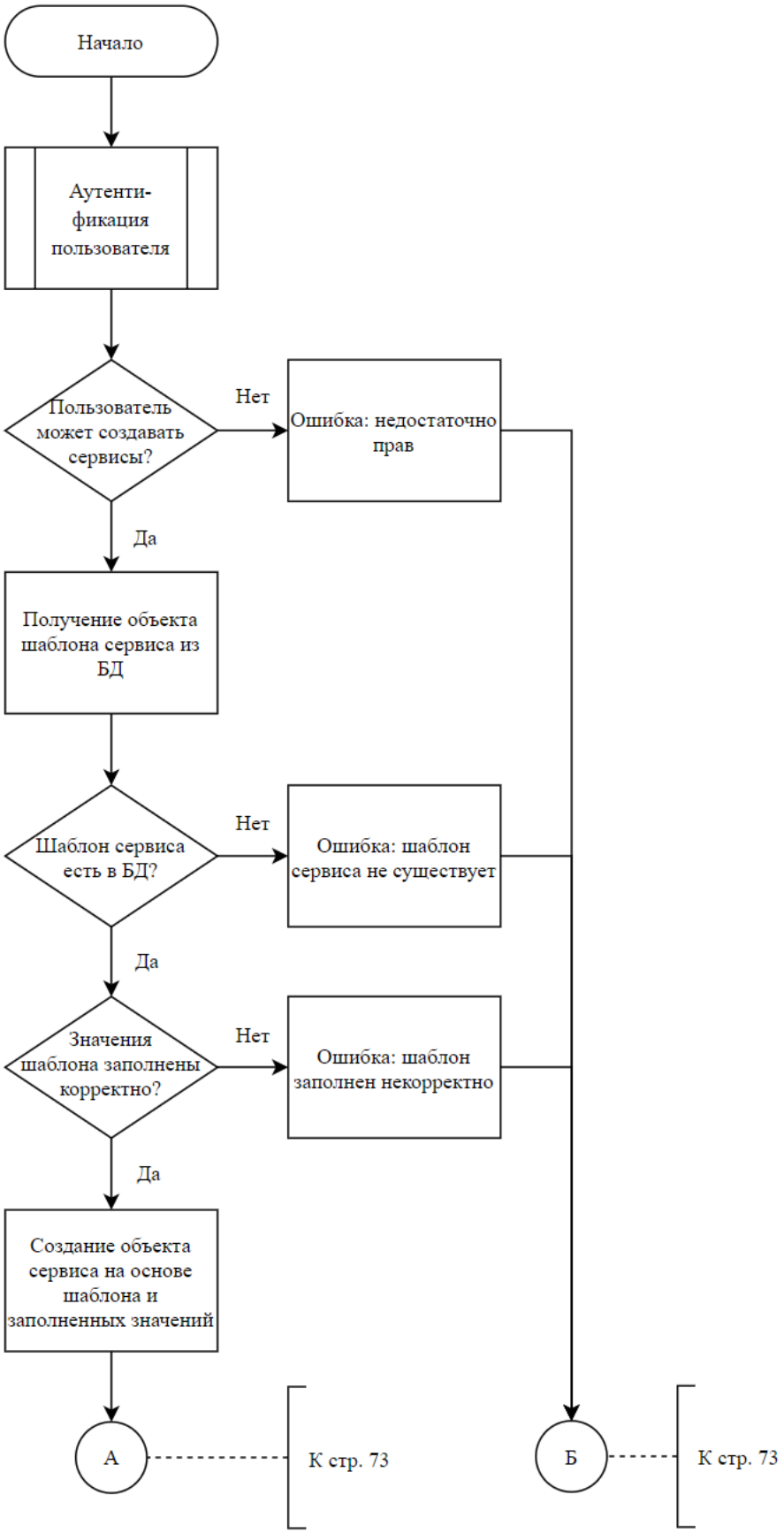
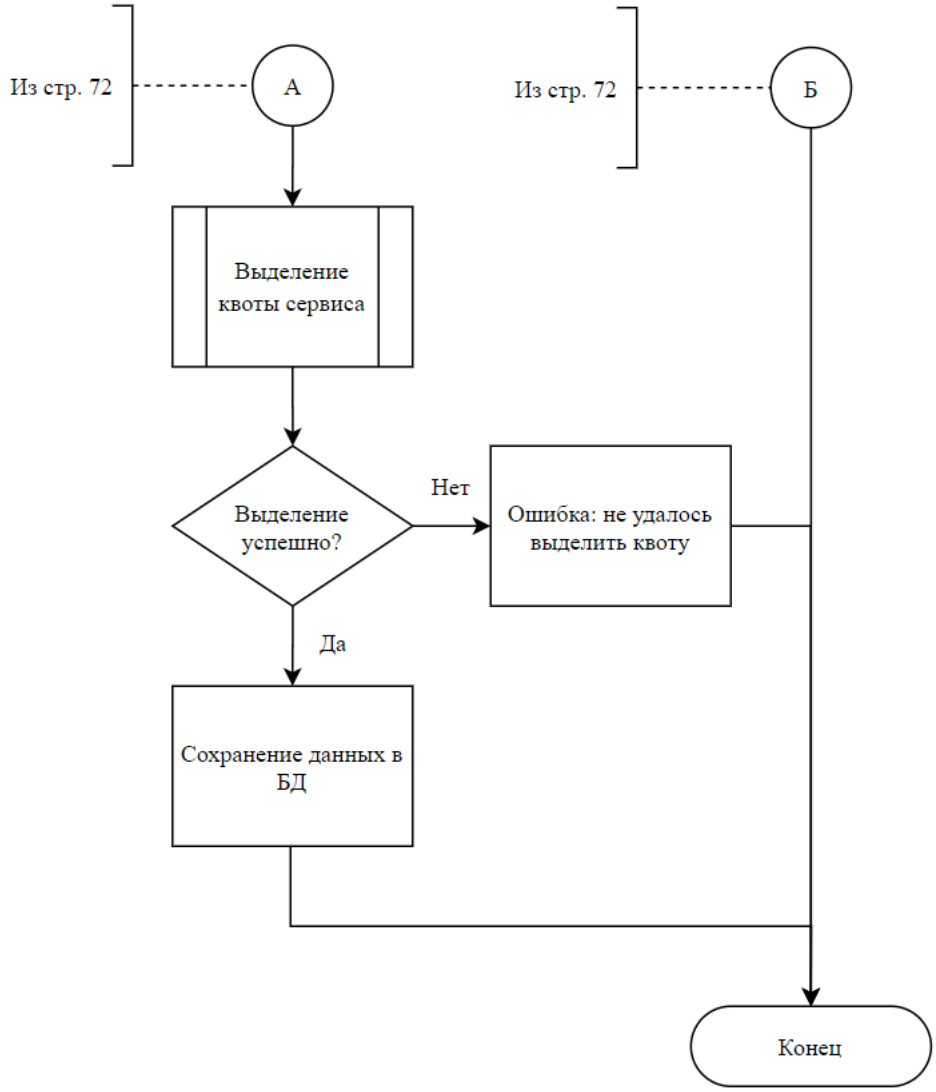


Рис. 2.17. Схема алгоритма создания сервиса



Продолжение рис. 2.17

2.4.3. Описание алгоритма синхронизации параметров сервиса в Kubernetes

2.4.3.1. Назначение и характеристика алгоритма

Данный алгоритм предназначен для синхронизации (обновления параметров) сервиса в Kubernetes в соответствии с изменившимся объектом сервиса в БД. Алгоритм вызывается для каждого объекта сервиса, у которого поле pendingsync (ожидает обновления) именное истинное значение. После успешного выполнения алгоритма это поле принимает ложное значение, чтобы исключить излишнюю повторную синхронизацию. При внесении изменений в объект сервиса (при создании или обновлении) поле pendingsync принимает истинное значение. Эти переходы показаны на рис. 2.18.

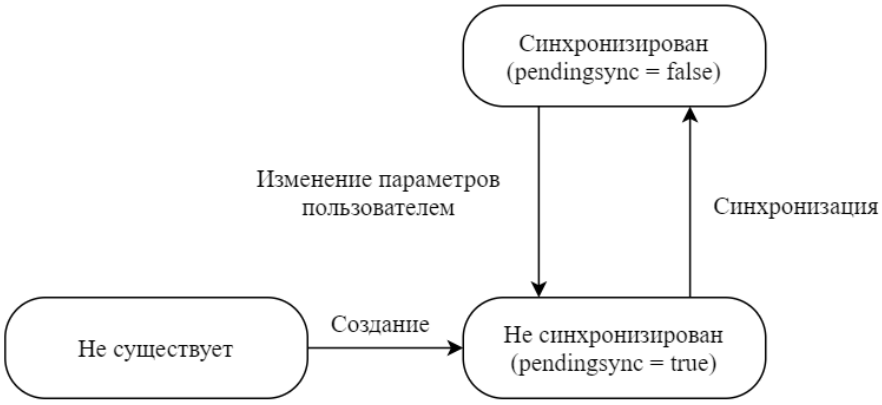


Рис. 2.18. Переходы между состояниями синхронизации сервиса

2.4.3.2. Используемая информация

Алгоритм использует следующие данные:

1) объект сервиса, описанный выше;

2) объект квоты ресурсов сервиса, описанный выше;

2.4.3.3. Результаты решения

В случае успеха, в Kubernetes сохраняется информация о желаемом состоянии сервиса, в БД сервис помечается, как синхронизированный (pendingsync принимает ложное значение).

В случае возникновения ошибки алгоритм производит запись в журнал ошибок и завершает работу. Так как сервис не был помечен, как синхронизированный, он будет обработан позднее.

2.4.3.4. Алгоритм решения

Алгоритм состоит из следующих шагов:

1) Получение объекта сервиса, содержащего поле pendingsync с истинным значением, из БД.

2) Если подходящий сервис есть в БД, переход к пункту 3, иначе алгоритм завершает работу.

3) Если сервис помечен, как удалённый, то переход к пункту 4, иначе к пункту 5.

4) Подготовка запроса на удаление сервиса в Kubernetes. Переход к пункту 8.

5) Получение объекта квоты этого сервиса. Для этого вызывается функция получения квоты сервиса подсистемы управления квотами ресурсов.

6) Если квота получена успешно, то переход к пункту 7, иначе к пункту 12.

7) Преобразование данных в формат Kuberenetes.

8) Отправка запроса в Kubernetes.

9) Если запрос успешен, то переход к пункту 10, иначе к пункту 13.

10) Выставление флага синхронизации pendingsync в false.

11) Запись данных в БД, алгоритм завершает работу.

12) Не удалось получить квоту сервиса. Запись ошибки в журнал, завершение работы алгоритма.

13) Не удалось поменять данные в Kubernetes. Запись ошибки в журнал, завершение работы алгоритма.

Схема данного алгоритма представлена на рис. 2.19.

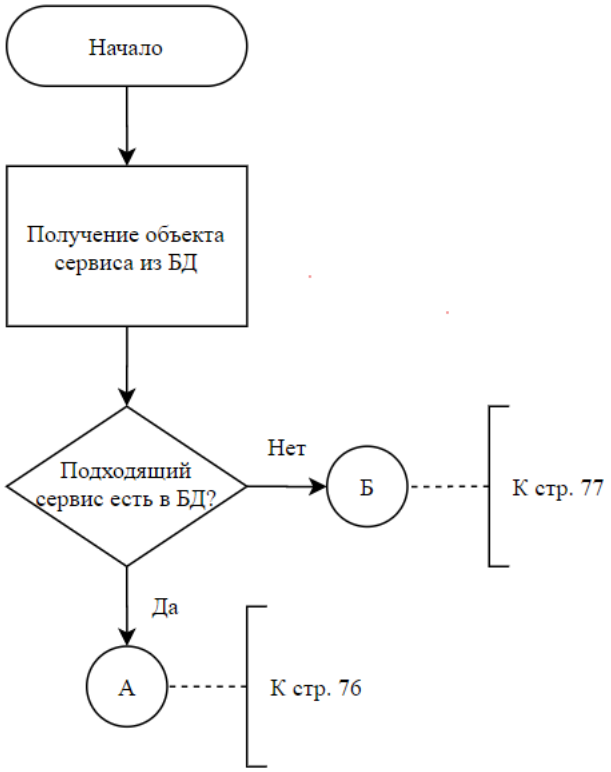
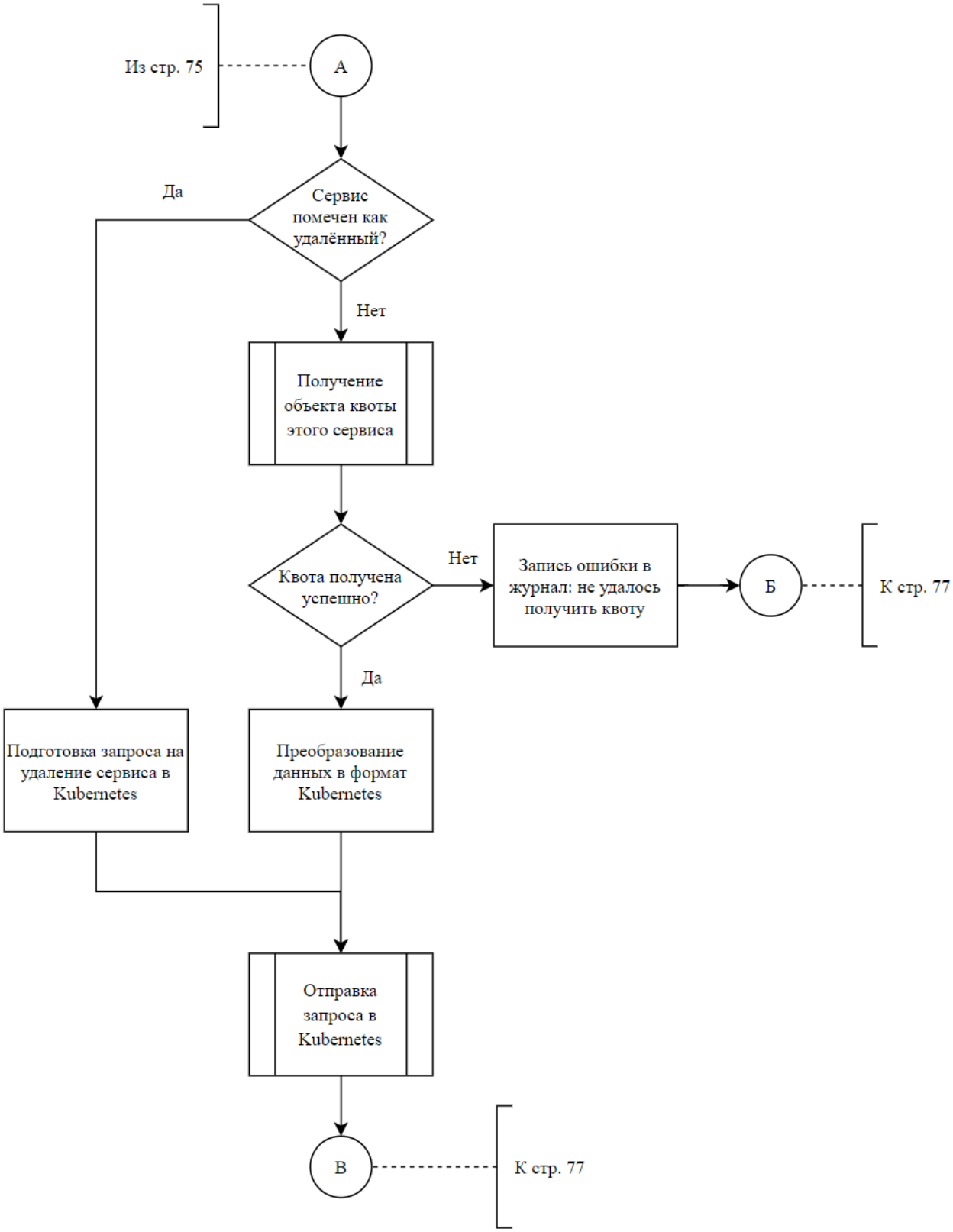
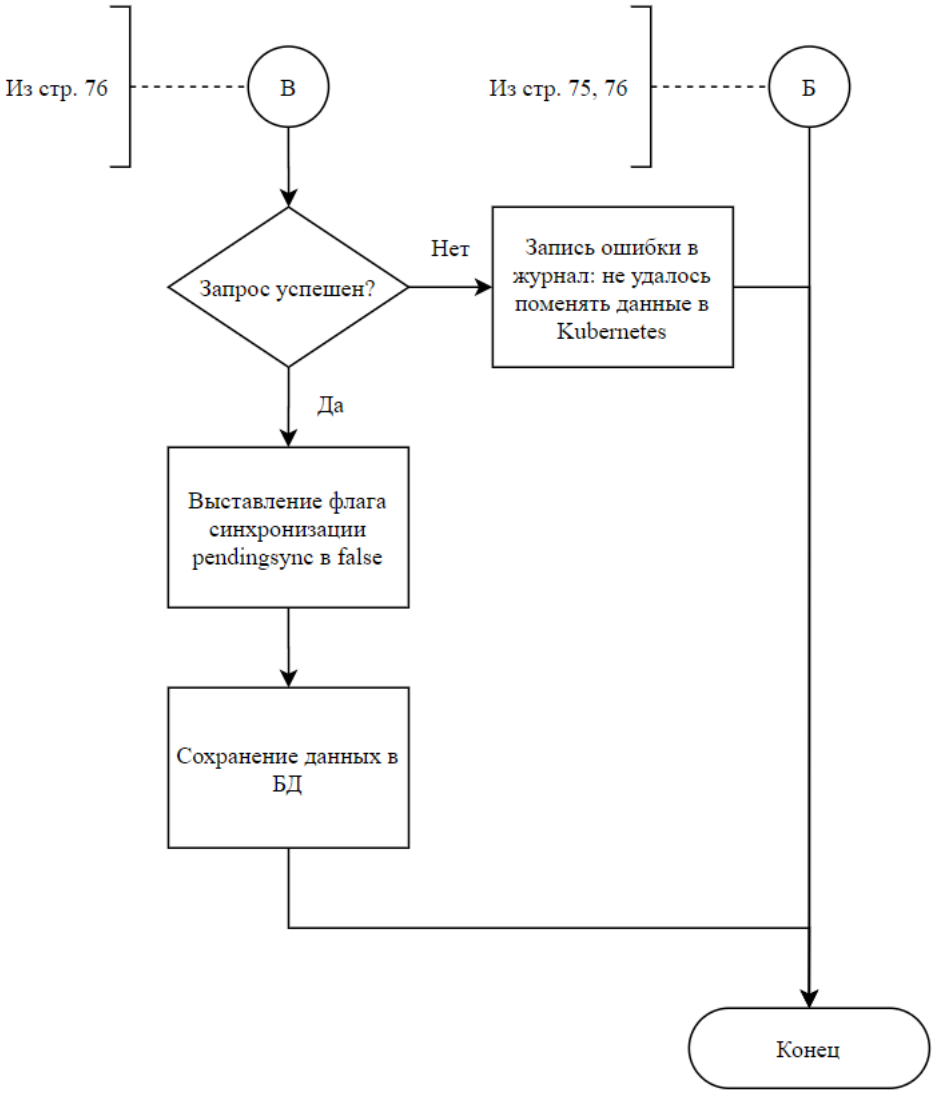


Рис. 2.19. Схема алгоритма синхронизации параметров сервиса в Kubernetes



Продолжение рис. 2.19



Продолжение рис. 2.19

2.5. Описание контрольного примера

2.5.1. Назначение

Контрольный пример служит для проверки работоспособности облачной платформы. В контрольном примере предусмотрена проверка следующих возможностей системы:

1) аутентификация пользователя;

2) подача заявок на участие в курсе;

3) подтверждение заявок на участие в курсе;

4) выделение квоты пользователя при зачислении на курс;

5) создание учебного сервиса;

6) получение информации о статусе учебного сервиса.

2.5.2. Исходные данные

Исходными данными являются созданный в платформе курс, шаблон сервиса, название нового сервиса и его параметры.

2.5.3. Результаты

После подачи заявки на зачисление на курс с аккаунта студента статус зачисления меняется на «ожидает подтверждения» (рис. 2.20). После подтверждения этой заявки с помощью аккаунта преподавателя статус меняется на «зачислен» (рис. 2.21, показан интерфейс преподавателя). После этого студенту становится доступна форма создания нового сервиса (рис. 2.22), при заполнении которой происходит создание сервиса, после чего пользователь перенаправляется на страницу с информацией о состоянии сервиса (рис. 2.23).

2.5.4. Результаты испытания

В результате испытания можно сделать вывод, что разработанная платформа выполняет все поставленные задачи.

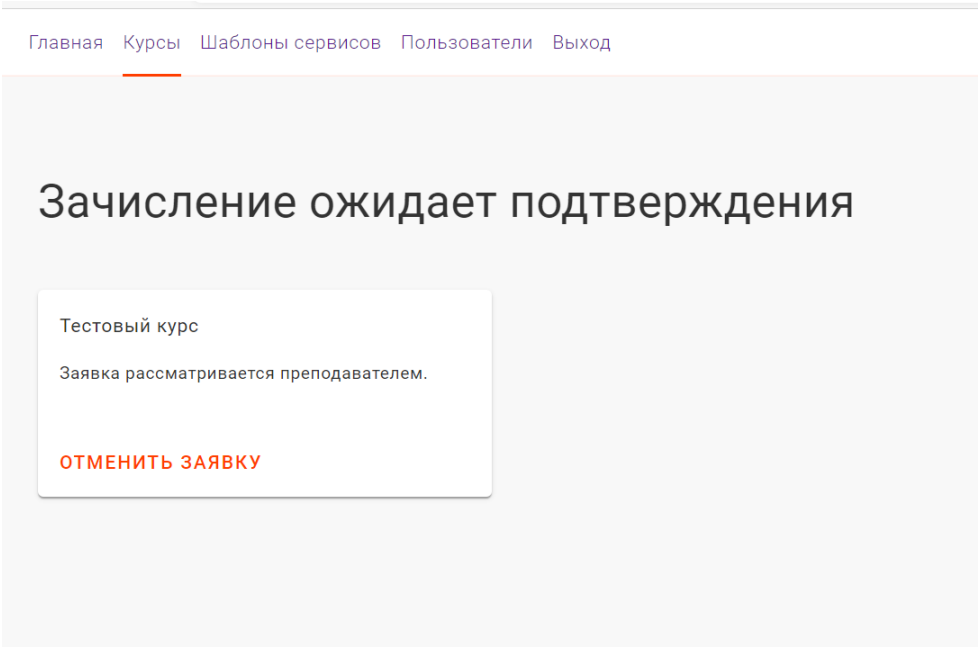


Рис. 2.20. Результат подачи заявки на курс

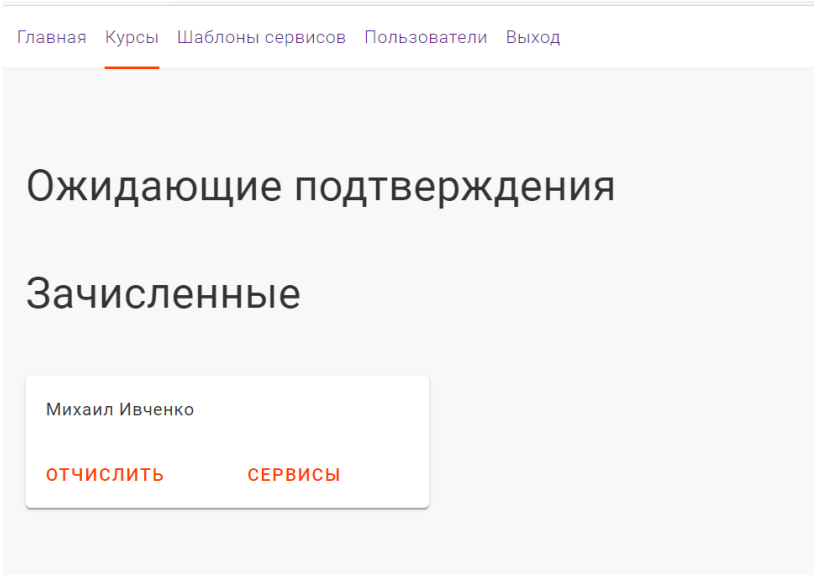


Рис. 2.21. Результат одобрения заявки на участие в курсе

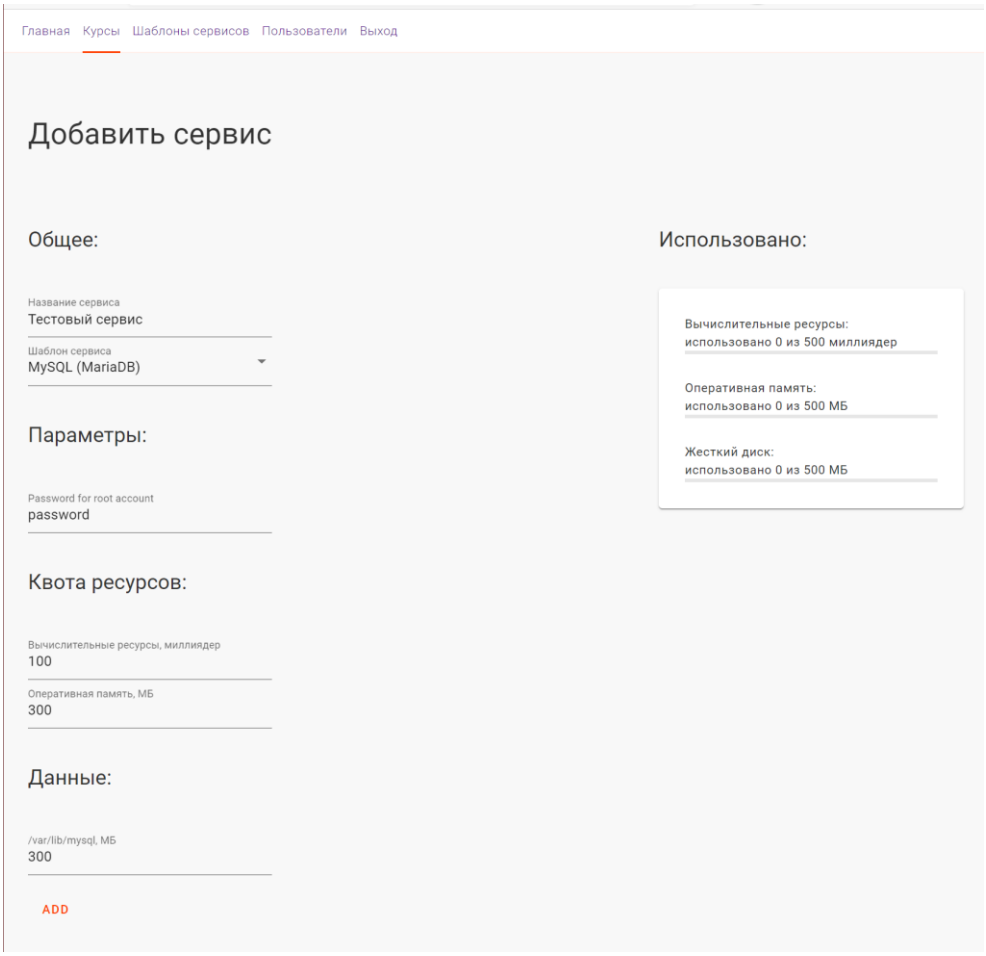


Рис. 2.22. Форма создания нового сервиса

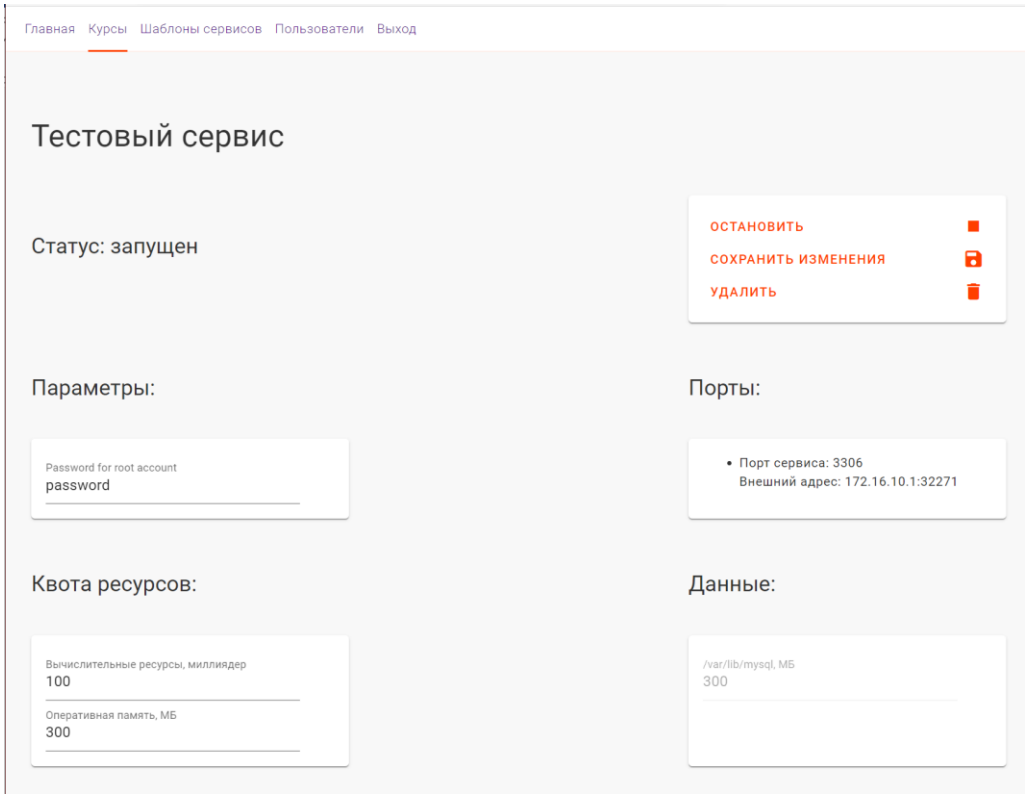


Рис. 2.23. Страница с информацией о сервисе

ЗАКЛЮЧЕНИЕ

Все поставленные задачи были выполнены в полном объёме:

1) разработана подсистема управления пользователями;

2) разработана подсистема управления учебными курсами;

3) разработана подсистема управления сервисами;

4) разработана подсистема управления квотами ресурсов;

5) разработано клиентское приложение.

Разработанная облачная платформа может применяться в ходе лабораторных работ по различным дисциплинам IT-направления высшего образования с целью сокращения времени, затрачиваемого студентами на создание и запуск учебных сервисов. Она может быть внедрена как с использованием собственных серверных ресурсов университета, так и с использованием арендованных мощностей поставщиков облачных услуг. Также данная облачная платформа имеет перспективы развития и может служить основой для разработки системы автоматической проверки лабораторных работ по дисциплине «Базы данных».

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Lingaraj Mallick, Pradeep Das, Kalandi Pradhan. Impact of educational expenditure on economic growth in major Asian countries: Evidence from econometric analysis // Theoretical and Applied Economics. – 2016. – №2 (№607). – С. 173-186.

2. Lei Huang, Yonggao Yang. Facilitating education using cloud computing infrastructure // Journal of Computing Sciences in Colleges 28. – 2013. – №4. – С. 19-25.

3. Dick Hardt. The OAuth 2.0 Authorization Framework // Internet Requests for Comments. – 2012. – №6749. – С. 1-76.

4. Mike Jones, Dick Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage // Internet Requests for Comments. – 2012. – №6750. – С. 1-18.

5. Paul Leach, Michael Mealling, Rich Salz. A Universally Unique IDentifier (UUID) URN Namespace // Internet Requests for Comments. – 2005. – №4122. – С. 1-32.

ПРИЛОЖЕНИЕ 1

ТЕКСТ ПРОГРАММЫ

cmd\scloud-api\main.go:

```
package main
import (...)
type configuration struct {
	MongoDB struct {
		URI      string
		Database string
	}
	HTTP struct {
		Address string
	}
	Services struct {
		Services struct {
			SyncEnabled       bool
			KubernetesService kubernetesservice.Config
		}
		Users usersservice.Config
	}
}
func newConfig() (*configuration, error) {
	viper.SetConfigName("config")
	viper.AddConfigPath(".")
	viper.AutomaticEnv()
	viper.SetConfigType("yml")
	if err := viper.ReadInConfig(); err != nil {
		return nil, fmt.Errorf("error reading config file: %w", err)
	}
	var config configuration
	err := viper.Unmarshal(&config)
	if err != nil {
		return nil, fmt.Errorf("unable to decode config: %w", err)
	}
	return &config, nil
}
func NewDatabase(config *configuration) (*mongo.Database, error) {
	ctx, cancel := context.WithTimeout(context.Background(), time.Second)
	defer cancel()
	client, err := mongo.Connect(ctx, options.Client().ApplyURI(config.MongoDB.URI))
	if err != nil {
		return nil, fmt.Errorf("failed to connecto to mongodb: %w", err)
	}
	return client.Database(config.MongoDB.Database), nil
}
func NewHTTPServer(
	lc fx.Lifecycle,
	config *configuration,
) *echo.Echo {
	e := echo.New()
	e.HideBanner = true
	e.HidePort = true
	lc.Append(fx.Hook{
		OnStart: func(ctx context.Context) error {
			go func() {
				err := e.Start(config.HTTP.Address)
				if errors.Is(err, http.ErrServerClosed) {
					e.Logger.Fatal(err)
				}
			}()
			return nil
		},
		OnStop: e.Shutdown,
	})
	return e
}
func Register(
	e *echo.Echo,
```

```
		usersService *usersservice.UsersService,
		usersAPI *usershttpapi.API,
		quotasAPI *quotashttpapi.API,
		servicesAPI *serviceshttpapi.API,
		coursesAPI *courseshttpapi.API,
) {
		basehttpapi.ConfigureMiddleware(e, func(ctx context.Context, token string) (user interface{}, err error) {
			return usersService.Auth(ctx, token)
		})
		usersAPI.RegisterRoutes(e)
		quotasAPI.RegisterRoutes(e)
		servicesAPI.RegisterRoutes(e)
		coursesAPI.RegisterRoutes(e)
}
func StartKubernetesSync(
		lc fx.Lifecycle,
		config *configuration,
		synchronizer *servicesservice.KubernetesSynchronizer,
) {
		if config.Services.Services.SyncEnabled {
			cancelCtx, cancel := context.WithCancel(context.Background())
			var wg sync.WaitGroup
			lc.Append(fx.Hook{
				OnStart: func(ctx context.Context) error {
					wg.Add(1)
					go func() {
						for cancelCtx.Err() == nil {
							err := synchronizer.SyncOne(cancelCtx)
							if err != nil {
								log.Printf("failed to sync: %v", err)
							}
						}
						wg.Done()
					}()
					return nil
				},
				OnStop: func(ctx context.Context) error {
					cancel()
					wg.Wait()
					return nil
				},
			})
		}
}
func main() {
		app := fx.New(
			fx.Provide(
				newConfig,
				NewDatabase,
				kubernetesservice.NewKubernetesService,
				quotasservice.NewQuotaService,
				servicesservice.NewServicesService,
				servicesservice.NewKubernetesSynchronizer,
				servicesservice.NewServiceTemplatesService,
				servicesservice.NewServicePermissionsService,
				usersservice.NewUsersService,
				coursesservice.NewCourseService,
				usershttpapi.NewAPI,
				quotashttpapi.NewAPI,
				serviceshttpapi.NewAPI,
				courseshttpapi.NewAPI,
				NewHTTPServer,
				func(c *configuration) usersservice.Config { return c.Services.Users },
				func(c *configuration) kubernetesservice.Config {
					return c.Services.Services.KubernetesService
				},
				func(s *quotasservice.QuotaService) usersservice.QuotaService { return s },
				func(s *quotasservice.QuotaService) servicesservice.QuotaService { return s },
				func(s *quotasservice.QuotaService) servicesservice.QuotaView { return s },
				func(s *quotasservice.QuotaService) coursesservice.QuotaService { return s },
				func(s *kubernetesservice.KubernetesService) servicesservice.KubernetesService { return s },
				func(s *coursesservice.CourseService) servicesservice.CoursesService { return s },
```

```
                        func(s *servicesservice.ServicesService) coursesservice.ServicesService { return s },
                ),
                fx.Invoke(coursesrepository.CreateIndexes),
                fx.Invoke(usersrepository.CreateIndexes),
                fx.Invoke(servicesrepository.CreateIndexes),
                fx.Invoke(quotasrepository.CreateIndexes),
                fx.Invoke(StartKubernetesSync),
                fx.Invoke(Register),
        )
        app.Run()
}
```

courses\coursesdomain\course.go:

```
package coursesdomain
import (...)
type Course struct {
        ID                  uuid.UUID
        Title               string
        TeacherID           uuid.UUID
        IsTeacherQuotaAcquired bool
        DeletionTime        time.Time
        Enrollments map[uuid.UUID]Enrollment
}
func NewCourse(
        id uuid.UUID,
        title string,
        teacherID uuid.UUID,
        isTeacherQuotaAcquired bool,
        deletionTime time.Time,
        enrollments map[uuid.UUID]Enrollment,
) (*Course, error) {
        if enrollments == nil {
                enrollments = make(map[uuid.UUID]Enrollment)
        }
        return &Course{
                ID:                  id,
                Title:               title,
                TeacherID:           teacherID,
                IsTeacherQuotaAcquired: isTeacherQuotaAcquired,
                DeletionTime:        deletionTime,
                Enrollments:         enrollments,
        }, nil
}
func CreateCourse(
        title string,
        teacherID uuid.UUID,
        deletionTime time.Time,
) (*Course, error) {
        enrollments := make(map[uuid.UUID]Enrollment)
        return &Course{
                ID:                  uuid.Must(uuid.NewV4()),
                Title:               title,
                TeacherID:           teacherID,
                IsTeacherQuotaAcquired: false,
                DeletionTime:        deletionTime,
                Enrollments:         enrollments,
        }, nil
}
func (c *Course) AddEnrollment(userID uuid.UUID) error {
        if _, ok := c.Enrollments[userID]; ok {
                message := fmt.Sprintf("enrollment for student %q in course %q already exists", userID, c.ID)
                return baseerrors.NewBusinessRule("enrollmentAlreadyExists", message)
        }
        c.Enrollments[userID] = EnrollmentPending
        return nil
}
func (c *Course) AcquireTeacherQuota() error {
        if c.IsTeacherQuotaAcquired {
                return baseerrors.NewBusinessRule("enrollmentAlreadyExsits", "teacher quota already acquired")
        }
```

```
	c.IsTeacherQuotaAcquired = true
	return nil
}
func (c *Course) ReleaseTeacherQuota() error {
	if !c.IsTeacherQuotaAcquired {
		return baseerrors.NewBusinessRule("enrollmentDoesNotExist", "teacher quota already released")
	}
	c.IsTeacherQuotaAcquired = false
	return nil
}
func (c *Course) GetEnrollmentByUserID(userID uuid.UUID) (Enrollment, error) {
	if enrollment, ok := c.Enrollments[userID]; ok {
		return enrollment, nil
	}
	return EnrollmentNone, nil
}
func (c *Course) ApproveEnrollment(studentID uuid.UUID) error {
	enrollment, exists := c.Enrollments[studentID]
	if !exists {
		message := fmt.Sprintf("enrollment for student %s does not exist", studentID)
		return baseerrors.NewBusinessRule("enrollmentDoesNotExist", message)
	}
	if enrollment != EnrollmentPending {
		message := fmt.Sprintf("enrollment for student %s is not in pending state", studentID)
		return baseerrors.NewBusinessRule("enrollmentIsNotPending", message)
	}
	c.Enrollments[studentID] = EnrollmentApproved
	return nil
}
func (c *Course) RemoveEnrollment(studentID uuid.UUID) error {
	_, exists := c.Enrollments[studentID]
	if !exists {
		message := fmt.Sprintf("enrollment for student %s does not exist", studentID)
		return baseerrors.NewBusinessRule("enrollmentDoesNotExist", message)
	}
	delete(c.Enrollments, studentID)
	return nil
}
func (c *Course) SetTitle(title string) error {
	if title == "" {
		return baseerrors.NewBusinessRule("emptyTitle", "title should not be empty")
	}
	c.Title = title
	return nil
}
func (c *Course) SetDeletionTime(deletionTime time.Time) error {
	if c.DeletionTime.Before(time.Now().UTC()) {
		return baseerrors.NewBusinessRule("deletionTimeInPast", "deletion time should not be in past")
	}
	c.DeletionTime = deletionTime
	return nil
}
```

courses\coursesdomain\enrollment.go:

```
package coursesdomain
import (...)
type Enrollment struct {
	State string
}
var (
	EnrollmentStateNone     = "None"
	EnrollmentStatePending  = "Pending"
	EnrollmentStateApproved = "Approved"
)
var possibleStates = make(map[string]bool, 5)
func init() {
	possibleStates[EnrollmentStateNone] = true
	possibleStates[EnrollmentStatePending] = true
	possibleStates[EnrollmentStateApproved] = true
}
```

```
func NewEnrollment(state string) (*Enrollment, error) {
	if _, ok := possibleStates[state]; !ok {
		message := fmt.Sprintf("invalid enrollment state: %q", state)
		return nil, baseerrors.NewBusinessRule("invalidEnrollmentState", message)
	}
	return &Enrollment{
		State: state,
	}, nil
}
var EnrollmentNone = Enrollment{State: EnrollmentStateNone}
var EnrollmentPending = Enrollment{State: EnrollmentStatePending}
var EnrollmentApproved = Enrollment{State: EnrollmentStateApproved}
```

courses\courseshttpapi\addcourse.go:

```
package courseshttpapi
import (...)
type IDResponse struct {
	ID string `json:"id" form:"id" query:"id"`
}
func (api *API) addCourseRoute(c echo.Context) error {
	course := new(coursesservice.Course)
	if err := c.Bind(course); err != nil {
		return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("could not bind request: %s", err))
	}
	user := basehttpapi.GetUser(c).(*usersservice.User)
	id, err := api.coursesService.CreateCourse(context.Background(), user, course)
	if err != nil {
		var businessRuleErr *baseerrors.BusinessRule
		if errors.As(err, &businessRuleErr) {
			if businessRuleErr.Code == "forbidden" {
				return echo.NewHTTPError(http.StatusForbidden, businessRuleErr.Code)
			}
			return echo.NewHTTPError(http.StatusBadRequest, businessRuleErr.Code)
		}
		log.Errorf("failed to create course: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to create course")
	}
	return c.JSON(http.StatusCreated, IDResponse{ID: id})
}
```

courses\courseshttpapi\addenrollment.go:

```
package courseshttpapi
import (...)
func (api *API) addEnrollmentRoute(c echo.Context) error {
	courseID, err := uuid.FromString(c.Param("courseID"))
	if err != nil {
		log.Warnf("invalid course id %q: %v", c.Param("courseID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid course id")
	}
	user := basehttpapi.GetUser(c).(*usersservice.User)
	err = api.coursesService.CreateEnrollment(context.Background(), courseID, user.ID)
	if err != nil {
		log.Errorf("failed to create enrollment: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to create enrollment")
	}
	return c.NoContent(http.StatusCreated)
}
```

courses\courseshttpapi\courseshttpapi.go:

```
package courseshttpapi
import (...)
type API struct {
	coursesService *coursesservice.CourseService
	usersService   *usersservice.UsersService
}
func NewAPI(courseService *coursesservice.CourseService, usersService *usersservice.UsersService) (*API, error) {
	return &API{
		coursesService: courseService,
```

```
                usersService:   usersService,
        }, nil
}
```

courses\courseshttpapi\deletecourse.go:

```
package courseshttpapi
import (...)
func (api *API) deleteCourseRoute(c echo.Context) error {
        user := basehttpapi.GetUser(c).(*usersservice.User)
        courseID, err := uuid.FromString(c.Param("courseID"))
        if err != nil {
                log.Warnf("invalid course id %q: %v", c.Param("courseID"), err)
                return echo.NewHTTPError(http.StatusBadRequest, "invalid course id")
        }
        if err := basehttpapi.CheckAccess(func() (bool, error) {
                return api.coursesService.CanManage(context.Background(), user, courseID)
        }); err != nil {
                return err
        }
        err = api.coursesService.DeleteCourse(context.Background(), courseID)
        if err != nil {
                if errors.Is(err, baserepository.ErrNoEntities) {
                        return echo.NewHTTPError(http.StatusNotFound, "course not found")
                }
                log.Errorf("failed to delete course: %v", err)
                return echo.NewHTTPError(http.StatusInternalServerError, "failed to delete course")
        }
        return c.NoContent(http.StatusOK)
}
```

courses\courseshttpapi\deleteenrollment.go:

```
package courseshttpapi
import (...)
func (api *API) deleteEnrollmentRoute(c echo.Context) error {
        courseID, err := uuid.FromString(c.Param("courseID"))
        if err != nil {
                log.Warnf("invalid course id %q: %v", c.Param("courseID"), err)
                return echo.NewHTTPError(http.StatusBadRequest, "invalid course id")
        }
        user := basehttpapi.GetUser(c).(*usersservice.User)
        enrollmentID, err := basehttpapi.HandleMeUserID(user.ID, c.Param("enrollmentID"), func() (bool, error) {
                return api.coursesService.CanManage(context.Background(), user, courseID)
        })
        if err != nil {
                return err
        }
        ep := new(enrollmentPatch)
        if err := c.Bind(ep); err != nil {
                return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("could not bind request: %s", err))
        }
        err = api.coursesService.DeleteEnrollment(context.Background(), courseID, enrollmentID)
        if err != nil {
                var businessError *baseerrors.BusinessRule
                if errors.As(err, &businessError) {
                        return echo.NewHTTPError(http.StatusBadRequest, businessError.Code)
                }
                log.Errorf("failed to delete enrollment: %v", err)
                return echo.NewHTTPError(http.StatusInternalServerError, "failed to delete enrollment")
        }
        return c.NoContent(http.StatusOK)
}
```

courses\courseshttpapi\getcourse.go:

```
package courseshttpapi
import (...)
func (api *API) getCourseRoute(c echo.Context) error {
        user := basehttpapi.GetUser(c).(*usersservice.User)
        courseID, err := uuid.FromString(c.Param("courseID"))
```

```
	if err != nil {
		log.Warnf("invalid course id %q: %v", c.Param("courseID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid course id")
	}
	courses, err := api.coursesService.GetCourse(context.Background(), user.ID, courseID)
	if err != nil {
		if errors.Is(err, baserepository.ErrNoEntities) {
			return echo.NewHTTPError(http.StatusNotFound, "course not found")
		}
		log.Errorf("failed to get course: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get course")
	}
	return c.JSON(http.StatusOK, courses)
}
```

courses\courseshttpapi\getcourses.go:

```
package courseshttpapi
import (...)
func (api *API) getCoursesRoute(c echo.Context) error {
	user := basehttpapi.GetUser(c).(*usersservice.User)
	courses, err := api.coursesService.GetCourses(context.Background(), user.ID)
	if err != nil {
		log.Errorf("failed to get courses: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get courses")
	}
	return c.JSON(http.StatusOK, courses)
}
```

courses\courseshttpapi\getenrollments.go:

```
package courseshttpapi
import (...)
func (api *API) getEnrollmentsRoute(c echo.Context) error {
	user := basehttpapi.GetUser(c).(*usersservice.User)
	courseID, err := uuid.FromString(c.Param("courseID"))
	if err != nil {
		log.Warnf("invalid course id %q: %v", c.Param("courseID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid course id")
	}
	if err := basehttpapi.CheckAccess(func() (bool, error) {
		return api.coursesService.CanManage(context.Background(), user, courseID)
	}); err != nil {
		return err
	}
	enrollments, err := api.coursesService.GetEnrollments(context.Background(), courseID)
	if err != nil {
		log.Errorf("failed to get course enrollments: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get course enrollments")
	}
	userIDs := make([]uuid.UUID, len(enrollments))
	for i := range enrollments {
		userIDs[i] = enrollments[i].UserID
	}
	users, err := api.usersService.GetUsersByID(context.Background(), userIDs)
	if err != nil {
		log.Errorf("failed to get enrollment users: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get enrollment users")
	}
	enrollmentsWithUsers := make([]EnrollmentInfo, len(enrollments))
	for i := range enrollments {
		enrollmentsWithUsers[i] = EnrollmentInfo{
			User:  users[enrollments[i].UserID],
			State: enrollments[i].State,
		}
	}
	return c.JSON(http.StatusOK, enrollmentsWithUsers)
}
type EnrollmentInfo struct {
	User  *usersservice.User `json:"user"`
	State string             `json:"state"`
```

```
}
```

courses\courseshttpapi\patchcourse.go:

```
package courseshttpapi
import (...)
type patchCourseRequest struct {
	IsTeacherQuotaAcquired *bool `json:"is_teacher_quota_acquired,omitempty"`
	CourseInfo	*struct {
		Title	string	`json:"title"`
		DeletionTime time.Time `json:"deletion_time"`
	} `json:"course_info,omitempty"`
}
func (api *API) patchCourseRoute(c echo.Context) error {
	user := basehttpapi.GetUser(c).(*usersservice.User)
	courseID, err := uuid.FromString(c.Param("courseID"))
	if err != nil {
		log.Warnf("invalid course id %q: %v", c.Param("courseID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid course id")
	}
	if err := basehttpapi.CheckAccess(func() (bool, error) {
		return api.coursesService.CanManage(context.Background(), user, courseID)
	}); err != nil {
		return err
	}
	var request patchCourseRequest
	err = c.Bind(&request)
	if err != nil {
		return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("could not bind request: %s", err))
	}
	if request.IsTeacherQuotaAcquired != nil {
		if *request.IsTeacherQuotaAcquired {
			err = api.coursesService.AcquireTeacherQuota(context.Background(), courseID)
		} else {
			err = api.coursesService.ReleaseTeacherQuota(context.Background(), courseID)
		}
		if err != nil {
			if errors.Is(err, baserepository.ErrNoEntities) {
				return echo.NewHTTPError(http.StatusNotFound, "course not found")
			}
			log.Errorf("failed to change teacher quota allocation status: %v", err)
			return echo.NewHTTPError(http.StatusInternalServerError, "failed to change teacher quota allocation status")
		}
	}
	if request.CourseInfo != nil {
		err = api.coursesService.ChangeCourse(
			context.Background(),
			courseID,
			request.CourseInfo.Title,
			request.CourseInfo.DeletionTime,
		)
		if err != nil {
			if errors.Is(err, baserepository.ErrNoEntities) {
				return echo.NewHTTPError(http.StatusNotFound, "course not found")
			}
			log.Errorf("failed to change course: %v", err)
			return echo.NewHTTPError(http.StatusInternalServerError, "failed to change course")
		}
	}
	return c.NoContent(http.StatusOK)
}
```

courses\courseshttpapi\patchenrollment.go:

```
package courseshttpapi
import (...)
type enrollmentPatch struct {
	State string `json:"state" form:"state" query:"state"`
}
func (api *API) patchEnrollmentRoute(c echo.Context) error {
	user := basehttpapi.GetUser(c).(*usersservice.User)
```

```
	courseID, err := uuid.FromString(c.Param("courseID"))
	if err != nil {
		log.Warnf("invalid course id %q: %v", c.Param("courseID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid course id")
	}
	enrollmentID, err := uuid.FromString(c.Param("enrollmentID"))
	if err != nil {
		log.Warnf("invalid enrollment id %q: %v", c.Param("enrollmentID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid enrollment id")
	}
	ep := new(enrollmentPatch)
	if err := c.Bind(ep); err != nil {
		return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("could not bind request: %s", err))
	}
	if err := basehttpapi.CheckAccess(func() (bool, error) {
		return api.coursesService.CanManage(context.Background(), user, courseID)
	}); err != nil {
		return err
	}
	if ep.State == "Approved" {
		err = api.coursesService.ApproveEnrollment(context.Background(), courseID, enrollmentID)
		if err != nil {
			var businessError *baseerrors.BusinessRule
			if errors.As(err, &businessError) {
				return echo.NewHTTPError(http.StatusBadRequest, businessError.Code)
			}
			log.Errorf("failed to approve enrollment: %v", err)
			return echo.NewHTTPError(http.StatusInternalServerError, "failed to approve enrollment")
		}
		return c.NoContent(http.StatusOK)
	}
	return echo.NewHTTPError(http.StatusBadRequest, "invalid state")
}
```

courses\courseshttpapi\router.go:

```
package courseshttpapi
import (...)
func (api *API) RegisterRoutes(echo *echo.Echo) {
	echo.GET("/courses", api.getCoursesRoute)
	echo.POST("/courses", api.addCourseRoute)
	echo.GET("/courses/:courseID", api.getCourseRoute)
	echo.PATCH("/courses/:courseID", api.patchCourseRoute)
	echo.DELETE("/courses/:courseID", api.deleteCourseRoute)
	echo.GET("/courses/:courseID/enrollments", api.getEnrollmentsRoute)
	echo.POST("/courses/:courseID/enrollments", api.addEnrollmentRoute)
	echo.PATCH("/courses/:courseID/enrollments/:enrollmentID", api.patchEnrollmentRoute)
	echo.DELETE("/courses/:courseID/enrollments/:enrollmentID", api.deleteEnrollmentRoute)
}
```

courses\coursesservice\courseservice.go:

```
package coursesservice
import (...)
type CourseService struct {
	db             *mongo.Database
	quotaService   QuotaService
	servicesService ServicesService
}
func NewCourseService(
	db *mongo.Database,
	quotaService QuotaService,
	servicesService ServicesService,
) (*CourseService, error) {
	return &CourseService{
		db:             db,
		quotaService:   quotaService,
		servicesService: servicesService,
	}, nil
}
type Course struct {
```

```
	Title		string			`json:"title"`
	DeletionTime	time.Time		`json:"deletion_time"`
	PerEnrollmentQuota quotasmodels.QuotaRequest `json:"per_enrollment_quota"`
}
type QuotaService interface {
	AcquireEnrollmentQuota(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID) error
	ReleaseEnrollmentQuota(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID) error
	AcquireCourseQuota(ctx context.Context, teacherID uuid.UUID,
		courseID uuid.UUID, perEnrollmentQuota quotasmodels.QuotaRequest) error
	ReleaseCourseQuota(ctx context.Context, courseID uuid.UUID) error
}
type ServicesService interface {
	DeleteServicesByEnrollment(ctx context.Context, courseID uuid.UUID, userID uuid.UUID) error
}
func (cs *CourseService) CreateCourse(
	ctx context.Context,
	teacher *usersservice.User,
	courseCreationForm *Course,
) (id string, err error) {
	coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
	if err != nil {
		return "", fmt.Errorf("failed to create courses repository: %w", err)
	}
	if !teacher.IsTeacher() && !teacher.IsAdmin() {
		return "", baseerrors.NewBusinessRule("forbidden", "user is not allowed to create courses")
	}
	course, err := coursesdomain.CreateCourse(courseCreationForm.Title, teacher.ID, courseCreationForm.DeletionTime)
	if err != nil {
		return "", fmt.Errorf("failed to create new course domain object: %w", err)
	}
	err = baseservice.WithTransaction(ctx, cs.db,
		baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
			err = cs.quotaService.AcquireCourseQuota(ctx, teacher.ID, course.ID,
courseCreationForm.PerEnrollmentQuota)
			if err != nil {
				return "", fmt.Errorf("failed to acquire course quota: %w", err)
			}
			err = coursesRepository.Insert(ctx, course)
			if err != nil {
				return "", fmt.Errorf("failed to save new course: %w", err)
			}
			return nil, nil
		}))
	if err != nil {
		return "", fmt.Errorf("failed to commit transaction: %w", err)
	}
	return course.ID.String(), nil
}
func (cs *CourseService) CreateEnrollment(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID) error {
	coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
	if err != nil {
		return fmt.Errorf("failed to create courses repository: %w", err)
	}
	course, err := coursesRepository.Get(ctx, courseID)
	if err != nil {
		return fmt.Errorf("failed to find course %q: %w", courseID, err)
	}
	err = course.AddEnrollment(studentID)
	if err != nil {
		return fmt.Errorf("failed to add enrollment to course %q: %w", courseID, err)
	}
	err = coursesRepository.Save(ctx, course)
	if err != nil {
		return fmt.Errorf("failed to save course %q: %w", courseID, err)
	}
	return nil
}
func (cs *CourseService) ApproveEnrollment(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID) error {
	coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
	if err != nil {
		return fmt.Errorf("failed to create courses repository: %w", err)
	}
```

```
		err = baseservice.WithTransaction(ctx, cs.db,
			baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
				course, err := coursesRepository.Get(ctx, courseID)
				if err != nil {
					return nil, fmt.Errorf("failed to find course %q: %w", courseID, err)
				}
				err = cs.quotaService.AcquireEnrollmentQuota(ctx, courseID, studentID)
				if err != nil {
					return nil, fmt.Errorf("failed to acquire enrollment quota: %w", err)
				}
				err = course.ApproveEnrollment(studentID)
				if err != nil {
					return nil, fmt.Errorf("failed to change enrollment state: %w", err)
				}
				err = coursesRepository.Save(ctx, course)
				if err != nil {
					return nil, fmt.Errorf("failed to save course %q: %w", courseID, err)
				}
				return nil, nil
			}))
		if err != nil {
			return fmt.Errorf("failed to approve enrollment: %w", err)
		}
		return nil
}
func (cs *CourseService) AcquireTeacherQuota(ctx context.Context, courseID uuid.UUID) error {
		coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
		if err != nil {
			return fmt.Errorf("failed to create courses repository: %w", err)
		}
		err = baseservice.WithTransaction(ctx, cs.db,
			baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
				course, err := coursesRepository.Get(ctx, courseID)
				if err != nil {
					return nil, fmt.Errorf("failed to find course %q: %w", courseID, err)
				}
				err = cs.quotaService.AcquireEnrollmentQuota(ctx, courseID, course.TeacherID)
				if err != nil {
					return nil, fmt.Errorf("failed to acquire enrollment quota: %w", err)
				}
				err = course.AcquireTeacherQuota()
				if err != nil {
					return nil, fmt.Errorf("failed to acquire teacher quota in course: %w", err)
				}
				err = coursesRepository.Save(ctx, course)
				if err != nil {
					return nil, fmt.Errorf("failed to save course %q: %w", courseID, err)
				}
				return nil, nil
			}))
		if err != nil {
			return fmt.Errorf("failed to acquire teacher quota: %w", err)
		}
		return nil
}
func (cs *CourseService) ReleaseTeacherQuota(ctx context.Context, courseID uuid.UUID) error {
		coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
		if err != nil {
			return fmt.Errorf("failed to create courses repository: %w", err)
		}
		err = baseservice.WithTransaction(ctx, cs.db,
			baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
				course, err := coursesRepository.Get(ctx, courseID)
				if err != nil {
					return nil, fmt.Errorf("failed to find course %q: %w", courseID, err)
				}
				err = cs.cleanupEnrollment(ctx, courseID, course.TeacherID)
				if err != nil {
					return nil, fmt.Errorf("failed to cleanup teacher enrollment: %w", err)
				}
				err = course.ReleaseTeacherQuota()
				if err != nil {
```

```
                                        return nil, fmt.Errorf("failed to release teacher quota in course: %w", err)
                                }
                                err = coursesRepository.Save(ctx, course)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to save course %q: %w", courseID, err)
                                }
                                return nil, nil
                        }))
                if err != nil {
                        return fmt.Errorf("failed to release teacher quota: %w", err)
                }
                return nil
}
func (cs *CourseService) cleanupEnrollment(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID) error {
                err := cs.servicesService.DeleteServicesByEnrollment(ctx, courseID, studentID)
                if err != nil {
                        return fmt.Errorf("failed to delete services from enrollment: %w", err)
                }
                err = cs.quotaService.ReleaseEnrollmentQuota(ctx, courseID, studentID)
                if err != nil {
                        return fmt.Errorf("failed to release enrollment quota: %w", err)
                }
                return nil
}
func (cs *CourseService) DeleteCourse(ctx context.Context, courseID uuid.UUID) error {
                coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
                if err != nil {
                        return fmt.Errorf("failed to create courses repository: %w", err)
                }
                err = baseservice.WithTransaction(ctx, cs.db,
                        baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
                                course, err := coursesRepository.Get(ctx, courseID)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to find course %q: %w", courseID, err)
                                }
                                for userID, enrollment := range course.Enrollments {
                                        if enrollment == coursesdomain.EnrollmentApproved {
                                                err = cs.cleanupEnrollment(ctx, courseID, userID)
                                                if err != nil {
                                                        return nil, fmt.Errorf("failed to cleanup enrollment: %w", err)
                                                }
                                        }
                                        err = course.RemoveEnrollment(userID)
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to remove enrollment: %w", err)
                                        }
                                }
                                if course.IsTeacherQuotaAcquired {
                                        err = cs.cleanupEnrollment(ctx, courseID, course.TeacherID)
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to cleanup enrollment: %w", err)
                                        }
                                        err = course.ReleaseTeacherQuota()
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to release teacher quota: %w", err)
                                        }
                                }
                                err = cs.quotaService.ReleaseCourseQuota(ctx, courseID)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to release course quota: %w", err)
                                }
                                err = coursesRepository.Delete(ctx, course)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to delete course %q: %w", courseID, err)
                                }
                                return nil, nil
                        }))
                if err != nil {
                        return fmt.Errorf("failed to delete course: %w", err)
                }
                return nil
}
```

```
func (cs *CourseService) DeleteEnrollment(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID) error {
	coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
	if err != nil {
		return fmt.Errorf("failed to create courses repository: %w", err)
	}
	err = baseservice.WithTransaction(ctx, cs.db,
		baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
			course, err := coursesRepository.Get(ctx, courseID)
			if err != nil {
				return nil, fmt.Errorf("failed to find course %q: %w", courseID, err)
			}
			enrollment, err := course.GetEnrollmentByUserID(studentID)
			if err != nil {
				return nil, fmt.Errorf("failed to find enrollment %q - %q: %w", courseID, studentID, err)
			}
			if enrollment == coursesdomain.EnrollmentApproved {
				err = cs.cleanupEnrollment(ctx, courseID, studentID)
				if err != nil {
					return nil, fmt.Errorf("failed to cleanup enrollment: %w", err)
				}
			}
			err = course.RemoveEnrollment(studentID)
			if err != nil {
				return nil, fmt.Errorf("failed to remove enrollment: %w", err)
			}
			err = coursesRepository.Save(ctx, course)
			if err != nil {
				return nil, fmt.Errorf("failed to save course %q: %w", courseID, err)
			}
			return nil, nil
		}))
	if err != nil {
		return fmt.Errorf("failed to delete enrollment: %w", err)
	}
	return nil
}
type CourseInfo struct {
	ID                    string        `json:"id"`
	TeacherID             string        `json:"teacher_id"`
	Title                 string        `json:"title"`
	IsTeacherQuotaAcquired bool         `json:"is_teacher_quota_acquired"`
	DeletionTime          time.Time     `json:"deletion_time"`
	EnrollmentInfo        EnrollmentInfo `json:"enrollment_info"`
}
type EnrollmentInfo struct {
	UserID uuid.UUID `json:"user_id" form:"user_id" query:"user_id"`
	State  string    `json:"state" form:"state" query:"state"`
}
func (cs *CourseService) GetCourses(ctx context.Context, studentID uuid.UUID) ([]CourseInfo, error) {
	coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
	if err != nil {
		return nil, fmt.Errorf("failed to create courses repository: %w", err)
	}
	courses, err := coursesRepository.GetCourses(ctx)
	if err != nil {
		return nil, fmt.Errorf("failed to get courses: %w", err)
	}
	courseInfos := make([]CourseInfo, len(courses))
	for i := range courseInfos {
		courseInfo, err := cs.convertCourse(courses[i], studentID)
		if err != nil {
			return nil, fmt.Errorf("failed to convert course: %w", err)
		}
		courseInfos[i] = *courseInfo
	}
	return courseInfos, nil
}
func (cs *CourseService) GetCourse(ctx context.Context, userID uuid.UUID, courseID uuid.UUID) (*CourseInfo, error) {
	coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
	if err != nil {
		return nil, fmt.Errorf("failed to create courses repository: %w", err)
	}
```

```
		course, err := coursesRepository.Get(ctx, courseID)
		if err != nil {
			return nil, fmt.Errorf("failed to get course: %w", err)
		}
		return cs.convertCourse(course, userID)
}
func (cs *CourseService) CanManage(ctx context.Context, user *usersservice.User, courseID uuid.UUID) (bool, error) {
		if user.IsAdmin() {
			return true, nil
		}
		if user.IsTeacher() {
			coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
			if err != nil {
				return false, fmt.Errorf("failed to create courses repository: %w", err)
			}
			versionedCourse, err := coursesRepository.Get(ctx, courseID)
			if err != nil {
				return false, fmt.Errorf("failed to get course: %w", err)
			}
			return versionedCourse.TeacherID == user.ID, nil
		}
		return false, nil
}
func (cs *CourseService) convertCourse(course *coursesdomain.Course, studentID uuid.UUID) (*CourseInfo, error) {
		enrollment, err := course.GetEnrollmentByUserID(studentID)
		if err != nil {
			return nil, fmt.Errorf("failed to get enrollment: %w", err)
		}
		courseInfo := &CourseInfo{
			ID:                 course.ID.String(),
			TeacherID:          course.TeacherID.String(),
			Title:              course.Title,
			IsTeacherQuotaAcquired: course.IsTeacherQuotaAcquired,
			DeletionTime:       course.DeletionTime,
			EnrollmentInfo: EnrollmentInfo{
				UserID: studentID,
				State:  enrollment.State,
			},
		}
		return courseInfo, nil
}
func (cs *CourseService) GetEnrollments(ctx context.Context, courseID uuid.UUID) ([]EnrollmentInfo, error) {
		coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
		if err != nil {
			return nil, fmt.Errorf("failed to create courses repository: %w", err)
		}
		course, err := coursesRepository.Get(ctx, courseID)
		if err != nil {
			return nil, fmt.Errorf("failed to get course: %w", err)
		}
		enrollments := make([]EnrollmentInfo, 0, len(course.Enrollments))
		for k, v := range course.Enrollments {
			enrollments = append(enrollments, EnrollmentInfo{
				UserID: k,
				State:  v.State,
			})
		}
		return enrollments, nil
}
func (cs *CourseService) ChangeCourse(
		ctx context.Context,
		courseID uuid.UUID,
		title string,
		deletionTime time.Time,
) error {
		coursesRepository, err := coursesrepository.NewCourseRepository(cs.db)
		if err != nil {
			return fmt.Errorf("failed to create courses repository: %w", err)
		}
		err = baseservice.WithTransaction(ctx, cs.db,
			baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
				course, err := coursesRepository.Get(ctx, courseID)
```

```
                                if err != nil {
                                        return nil, fmt.Errorf("failed to find course %q: %w", courseID, err)
                                }
                                if title != course.Title {
                                        err = course.SetTitle(title)
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to set course title: %w", err)
                                        }
                                }
                                if deletionTime != course.DeletionTime {
                                        err = course.SetDeletionTime(deletionTime)
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to set course deletion time: %w", err)
                                        }
                                }
                                err = coursesRepository.Save(ctx, course)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to save course %q: %w", courseID, err)
                                }
                                return nil, nil
                        }))
        if err != nil {
                return fmt.Errorf("failed to change quota: %w", err)
        }
        return nil
}
```

oauth2checker\oauth2checker.go:

```
package oauth2checker
import (...)
type OAuthUserInfo struct {
        Subject        string `json:"sub"`
        Name           string `json:"name"`
        Email          string `json:"email"`
        IsAuthenticated bool   `json:"active"`
}
var ErrBadResponse = errors.New("bad response")
func CheckToken(ctx context.Context, introspectURL string, token string) (*OAuthUserInfo, error) {
        form := url.Values{
                "token": {token},
        }
        req, _ := http.NewRequestWithContext(ctx, http.MethodPost, introspectURL, strings.NewReader(form.Encode()))
        req.Header.Add("Content-Type", "application/x-www-form-urlencoded")
        client := &http.Client{}
        resp, err := client.Do(req)
        if err != nil {
                return nil, fmt.Errorf("failed to send request: %w", err)
        }
        defer resp.Body.Close()
        body, err := ioutil.ReadAll(resp.Body)
        if err != nil {
                return nil, fmt.Errorf("failed to read response: %w", err)
        }
        if resp.StatusCode != http.StatusOK {
                return nil, fmt.Errorf("response status code is not 200 OK: %d, %w", resp.StatusCode, ErrBadResponse)
        }
        var result OAuthUserInfo
        err = json.Unmarshal(body, &result)
        if err != nil {
                return nil, fmt.Errorf("failed to parse response: %w", err)
        }
        return &result, nil
}
```

quotas\quotasdomain\course.go:

```
package quotasdomain
import (...)
type Course struct {
        ID uuid.UUID
```

```
	QuotaPerEnrollment Resources
	Enrollments map[uuid.UUID]struct{}
}
func NewCourse(courseID uuid.UUID, quotaPerEnrollment Resources, enrollments map[uuid.UUID]struct{}) (*Course, error) {
	return &Course{
		ID:		courseID,
		QuotaPerEnrollment: quotaPerEnrollment,
		Enrollments:	enrollments,
	}, nil
}
func (c *Course) UsedQuota() Resources {
	return c.QuotaPerEnrollment.Multiply(int32(len(c.Enrollments)))
}
```

quotas\quotasdomain\enrollment.go:

```
package quotasdomain
import (...)
type Enrollment struct {
	CourseID  uuid.UUID
	StudentID uuid.UUID
	totalQuota Resources
	services   map[uuid.UUID]*Service
}
func NewEnrollment(
	courseID uuid.UUID,
	studentID uuid.UUID,
	quota Resources,
	services map[uuid.UUID]*Service,
) (*Enrollment, error) {
	usedQuota := Resources{}
	for i := range services {
		usedQuota = usedQuota.Add(services[i].Quota)
	}
	if !quota.IsEnough(usedQuota) {
		return nil, baseerrors.NewBusinessRule("notEnoughQuota", "not enough quota")
	}
	return &Enrollment{
		CourseID:   courseID,
		StudentID:  studentID,
		totalQuota: quota,
		services:   services,
	}, nil
}
func CreateEnrollment(courseID uuid.UUID, studentID uuid.UUID, quota Resources) (*Enrollment, error) {
	return &Enrollment{
		CourseID:   courseID,
		StudentID:  studentID,
		totalQuota: quota,
		services:   make(map[uuid.UUID]*Service),
	}, nil
}
func (e *Enrollment) FreeQuota() Resources {
	return e.TotalQuota().Subtract(e.UsedQuota())
}
func (e *Enrollment) UsedQuota() Resources {
	usedQuota := Resources{}
	for i := range e.services {
		usedQuota = usedQuota.Add(e.services[i].Quota)
	}
	return usedQuota
}
func (e *Enrollment) TotalQuota() Resources {
	return e.totalQuota
}
func (e *Enrollment) Services() map[uuid.UUID]*Service {
	return e.services
}
func (e *Enrollment) AddService(serviceID uuid.UUID, quota Resources) (*Service, error) {
	if _, exists := e.services[serviceID]; exists {
		message := fmt.Sprintf("service %q already exists", serviceID.String())
```

```
                return nil, baseerrors.NewBusinessRule("serviceAlreadyExists", message)
        }
        if !e.FreeQuota().IsEnough(quota) {
                message := fmt.Sprintf("not enough quota: %#v %#v", e.FreeQuota(), quota)
                return nil, baseerrors.NewBusinessRule("notEnoughQuota", message)
        }
        service, err := NewService(serviceID, quota)
        if err != nil {
                return nil, fmt.Errorf("failed to create service: %w", err)
        }
        e.services[serviceID] = service
        return service, nil
}
func (e *Enrollment) GetServiceQuota(serviceID uuid.UUID) (Resources, error) {
        if service, exists := e.services[serviceID]; exists {
                return service.Quota, nil
        }
        message := fmt.Sprintf("service %q does not exist", serviceID.String())
        return Resources{}, baseerrors.NewBusinessRule("serviceDoesNotExist", message)
}
func (e *Enrollment) DeleteService(serviceID uuid.UUID) error {
        if _, exists := e.services[serviceID]; !exists {
                message := fmt.Sprintf("service %q does not exist", serviceID.String())
                return baseerrors.NewBusinessRule("serviceDoesNotExist", message)
        }
        delete(e.services, serviceID)
        return nil
}
func (e *Enrollment) ScaleService(serviceID uuid.UUID, newQuota Resources) error {
        service, exists := e.services[serviceID]
        if !exists {
                message := fmt.Sprintf("service %q does not exist", serviceID.String())
                return baseerrors.NewBusinessRule("serviceDoesNotExist", message)
        }
        freeQuota := e.FreeQuota().Add(service.Quota)
        if !freeQuota.IsEnough(newQuota) {
                message := fmt.Sprintf("not enough quota: %#v %#v", freeQuota, newQuota)
                return baseerrors.NewBusinessRule("notEnoughQuota", message)
        }
        service.Quota = newQuota
        return nil
}
```

quotas\quotasdomain\resources.go:

```
package quotasdomain
type Resources struct {
        CPU     int32
        RAM     int32
        Storage int32
}
func NewResources(cpu, ram, storage int32) (*Resources, error) {
        return &Resources{
                CPU:     cpu,
                RAM:     ram,
                Storage: storage,
        }, nil
}
func (r Resources) Add(other Resources) Resources {
        return Resources{
                CPU:     r.CPU + other.CPU,
                RAM:     r.RAM + other.RAM,
                Storage: r.Storage + other.Storage,
        }
}
func (r Resources) Multiply(coef int32) Resources {
        return Resources{
                CPU:     r.CPU * coef,
                RAM:     r.RAM * coef,
                Storage: r.Storage * coef,
        }
```

```
}
func (r Resources) IsEnough(request Resources) bool {
	return r.CPU >= request.CPU && r.RAM >= request.RAM && r.Storage >= request.Storage
}
func (r Resources) Subtract(other Resources) Resources {
	return Resources{
		CPU:     r.CPU - other.CPU,
		RAM:     r.RAM - other.RAM,
		Storage: r.Storage - other.Storage,
	}
}
func (r Resources) Scale(from Resources, to Resources) Resources {
	return Resources{
		CPU:     (to.CPU * r.CPU) / from.CPU,
		RAM:     (to.RAM * r.RAM) / from.RAM,
		Storage: (to.Storage * r.Storage) / from.Storage,
	}
}
```

quotas\quotasdomain\service.go:

```
package quotasdomain
import (...)
type Service struct {
	ID uuid.UUID
	Quota Resources
}
func NewService(serviceID uuid.UUID, quota Resources) (*Service, error) {
	return &Service{
		ID:    serviceID,
		Quota: quota,
	}, nil
}
```

quotas\quotasdomain\teacher.go:

```
package quotasdomain
import (...)
type Teacher struct {
	ID uuid.UUID
	Quota Resources
	Courses map[uuid.UUID]*Course
}
func NewTeacher(ID uuid.UUID, quota Resources, courses map[uuid.UUID]*Course) (*Teacher, error) {
	usedQuota := Resources{}
	for k := range courses {
		usedQuota = usedQuota.Add(courses[k].UsedQuota())
	}
	if !quota.IsEnough(usedQuota) {
		message := fmt.Sprintf("not enough quota: %#v %#v", quota, usedQuota)
		return nil, baseerrors.NewBusinessRule("notEnoughQuota", message)
	}
	return &Teacher{
		ID:      ID,
		Quota:   quota,
		Courses: courses,
	}, nil
}
func CreateTeacher(ID uuid.UUID, quota Resources) (*Teacher, error) {
	return &Teacher{
		ID:      ID,
		Quota:   quota,
		Courses: make(map[uuid.UUID]*Course),
	}, nil
}
func (t *Teacher) Scale(newQuota Resources) error {
	if !newQuota.IsEnough(t.UsedQuota()) {
		return baseerrors.NewBusinessRule("quotaLessThanUsed", "cannot make new quota less than currently used")
	}
	t.Quota = newQuota
	return nil
```

```
}
func (t *Teacher) UsedQuota() Resources {
	var quota Resources
	for i := range t.Courses {
		quota = quota.Add(t.Courses[i].UsedQuota())
	}
	return quota
}
func (t *Teacher) TotalQuota() Resources {
	return t.Quota
}
func (t *Teacher) FreeQuota() Resources {
	return t.Quota.Subtract(t.UsedQuota())
}
func (t *Teacher) AddCourse(courseID uuid.UUID, quota Resources) (*Course, error) {
	if _, exists := t.Courses[courseID]; exists {
		message := fmt.Sprintf("course %q already exists", courseID.String())
		return nil, baseerrors.NewBusinessRule("courseAlreadyExists", message)
	}
	course, err := NewCourse(courseID, quota, nil)
	if err != nil {
		return nil, fmt.Errorf("failed to create course: %w", err)
	}
	t.Courses[courseID] = course
	return course, nil
}
func (t *Teacher) DeleteCourse(courseID uuid.UUID) error {
	if _, exists := t.Courses[courseID]; !exists {
		message := fmt.Sprintf("course %q not found", courseID.String())
		return baseerrors.NewBusinessRule("courseDoesNotExist", message)
	}
	if len(t.Courses[courseID].Enrollments) > 0 {
		const message = "could not delete course quota with active enrollments"
		return baseerrors.NewBusinessRule("courseIsNotEmpty", message)
	}
	delete(t.Courses, courseID)
	return nil
}
func (t *Teacher) AcquireEnrollmentQuota(courseID uuid.UUID, studentID uuid.UUID) (*Enrollment, error) {
	course, exists := t.Courses[courseID]
	if !exists {
		message := fmt.Sprintf("course %q not found", courseID.String())
		return nil, baseerrors.NewBusinessRule("courseDoesNotExist", message)
	}
	if !t.FreeQuota().IsEnough(course.QuotaPerEnrollment) {
		message := fmt.Sprintf("not enough quota: %#v %#v", t.FreeQuota(), course.QuotaPerEnrollment)
		return nil, baseerrors.NewBusinessRule("notEnoughQuota", message)
	}
	if _, exists := course.Enrollments[studentID]; exists {
		message := fmt.Sprintf("enrollment %q already exists in course %q", studentID.String(), courseID.String())
		return nil, baseerrors.NewBusinessRule("enrollmentAlreadyExists", message)
	}
	course.Enrollments[studentID] = struct{}{}
	enrollment, err := CreateEnrollment(courseID, studentID, course.QuotaPerEnrollment)
	if err != nil {
		return nil, fmt.Errorf("failed to create enrollment: %w", err)
	}
	return enrollment, nil
}
func (t *Teacher) ReleaseEnrollmentQuota(courseID uuid.UUID, studentID uuid.UUID) error {
	course, exists := t.Courses[courseID]
	if !exists {
		message := fmt.Sprintf("course %q not found", courseID.String())
		return baseerrors.NewBusinessRule("courseDoesNotExist", message)
	}
	if _, exists := course.Enrollments[studentID]; !exists {
		message := fmt.Sprintf("enrollment %q don't exists in course %q", studentID.String(), courseID.String())
		return baseerrors.NewBusinessRule("enrollmentDoesNotExist", message)
	}
	delete(course.Enrollments, studentID)
	return nil
}
```

```
func (t *Teacher) CanBeReleased(e *error) bool {
	if len(t.Courses) > 0 {
		const message = "could not release teacher quota with active courses"
		*e = baseerrors.NewBusinessRule("notEmptyTeacherQuota", message)
		return false
	}
	return true
}
```

quotas\quotashttpapi\getenrollmentquota.go:

```
package quotashttpapi
import (...)
func (api *API) getEnrollmentQuotaRoute(c echo.Context) error {
	courseID, err := uuid.FromString(c.Param("courseID"))
	if err != nil {
		log.Warnf("invalid course id %q: %v", c.Param("courseID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid course id")
	}
	user := basehttpapi.GetUser(c).(*usersservice.User)
	studentID, err := basehttpapi.HandleMeUserID(user.ID, c.Param("studentID"), func() (bool, error) {
		return api.coursesService.CanManage(context.Background(), user, courseID)
	})
	if err != nil {
		return err
	}
	quota, err := api.quotasService.GetEnrollmentQuota(context.Background(), courseID, studentID)
	if err != nil {
		if errors.Is(err, baseerrors.EntityNotFound) {
			return echo.NewHTTPError(http.StatusNotFound, "quota not found")
		}
		log.Errorf("failed to get quota: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get quota")
	}
	return c.JSON(http.StatusOK, quota)
}
```

quotas\quotashttpapi\getteacherquota.go:

```
package quotashttpapi
import (...)
func (api *API) getTeacherQuotaRoute(c echo.Context) error {
	user := basehttpapi.GetUser(c).(*usersservice.User)
	teacherID, err := basehttpapi.HandleMeUserID(user.ID, c.Param("userID"), func() (bool, error) {
		return user.IsAdmin(), nil
	})
	if err != nil {
		return err
	}
	quota, err := api.quotasService.GetTeacherQuota(context.Background(), teacherID)
	if err != nil {
		if errors.Is(err, baseerrors.EntityNotFound) {
			return echo.NewHTTPError(http.StatusNotFound, "quota not found")
		}
		log.Errorf("failed to get quota: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get quota")
	}
	return c.JSON(http.StatusOK, quota)
}
```

quotas\quotashttpapi\patchteacherquota.go:

```
package quotashttpapi
import (...)
func (api *API) patchTeacherQuotaRoute(c echo.Context) error {
	var request quotasmodels.QuotaRequest
	err := c.Bind(&request)
	if err != nil {
		return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("could not bind request: %s", err))
	}
	user := basehttpapi.GetUser(c).(*usersservice.User)
```

```
	userID, err := basehttpapi.HandleMeUserID(user.ID, c.Param("userID"), func() (bool, error) {
		return user.IsAdmin(), nil
	})
	if err != nil {
		return err
	}
	err = api.quotasService.ScaleTeacherQuota(context.Background(), userID, request)
	if err != nil {
		log.Errorf("failed to patch quota: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to patch quota")
	}
	return c.NoContent(http.StatusOK)
}
```

quotas\quotashttpapi\quotashttpapi.go:

```
package quotashttpapi
import (...)
type API struct {
	quotasService  *quotasservice.QuotaService
	coursesService *coursesservice.CourseService
}
func NewAPI(quotasService *quotasservice.QuotaService, coursesService *coursesservice.CourseService) (*API, error) {
	return &API{
		quotasService:  quotasService,
		coursesService: coursesService,
	}, nil
}
```

quotas\quotashttpapi\router.go:

```
package quotashttpapi
import (...)
func (api *API) RegisterRoutes(echo *echo.Echo) {
	echo.GET("/courses/:courseID/enrollments/:studentID/quota", api.getEnrollmentQuotaRoute)
	echo.GET("/users/:userID/quota", api.getTeacherQuotaRoute)
	echo.PATCH("/users/:userID/quota", api.patchTeacherQuotaRoute)
}
```

quotas\quotasservice\quotaservice.go:

```
package quotasservice
import (...)
type QuotaService struct {
	db *mongo.Database
}
func NewQuotaService(db *mongo.Database) (*QuotaService, error) {
	return &QuotaService{
		db: db,
	}, nil
}
func (qs *QuotaService) AcquireTeacherQuota(
	ctx context.Context,
	teacherID uuid.UUID,
	quota quotasmodels.QuotaRequest,
) error {
	repo, err := quotasrepository.NewTeacherRepository(qs.db)
	if err != nil {
		return fmt.Errorf("failed to create teacher repository: %w", err)
	}
	teacher, err := quotasdomain.CreateTeacher(teacherID, quotasdomain.Resources{
		CPU:     quota.CPU,
		RAM:     quota.RAM,
		Storage: quota.Storage,
	})
	if err != nil {
		return fmt.Errorf("failed to create new teacher: %w", err)
	}
	err = repo.Insert(ctx, teacher)
	if err != nil {
		return fmt.Errorf("failed to save new teacher: %w", err)
```

```
		}
		return nil
}
func (qs *QuotaService) ReleaseTeacherQuota(ctx context.Context, teacherID uuid.UUID) error {
		repo, err := quotasrepository.NewTeacherRepository(qs.db)
		if err != nil {
				return fmt.Errorf("failed to create teacher repository: %w", err)
		}
		err = baseservice.WithTransaction(ctx, qs.db,
				baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
						teacher, err := repo.Get(ctx, teacherID)
						if err != nil {
								if errors.Is(err, baserepository.ErrNoEntities) {
										return nil, baseerrors.EntityNotFound
								}
								return nil, fmt.Errorf("failed to find teacher quota: %w", err)
						}
						if !teacher.CanBeReleased(&err) {
								return nil, fmt.Errorf("unable to release teacher quota: %w", err)
						}
						err = repo.Delete(ctx, teacher)
						if err != nil {
								return nil, fmt.Errorf("failed to delete teacher quota: %w", err)
						}
						return nil, nil
				}))
		if err != nil {
				return fmt.Errorf("failed to release teacher quota: %w", err)
		}
		return nil
}
func (qs *QuotaService) ScaleTeacherQuota(
		ctx context.Context,
		teacherID uuid.UUID,
		newQuota quotasmodels.QuotaRequest,
) error {
		repo, err := quotasrepository.NewTeacherRepository(qs.db)
		if err != nil {
				return fmt.Errorf("failed to create teacher repository: %w", err)
		}
		err = baseservice.WithTransaction(ctx, qs.db,
				baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
						teacher, err := repo.Get(ctx, teacherID)
						if err != nil {
								return nil, fmt.Errorf("failed to find teacher %q: %w", teacherID, err)
						}
						err = teacher.Scale(quotasdomain.Resources{
								CPU:     newQuota.CPU,
								RAM:     newQuota.RAM,
								Storage: newQuota.Storage,
						})
						if err != nil {
								return nil, fmt.Errorf("failed to scale teacher quota: %w", err)
						}
						err = repo.Save(ctx, teacher)
						if err != nil {
								return nil, fmt.Errorf("failed to save teacher: %w", err)
						}
						return nil, nil
				}))
		if err != nil {
				return fmt.Errorf("failed to scale teacher quota: %w", err)
		}
		return nil
}
func (qs *QuotaService) GetTeacherQuota(ctx context.Context, teacherID uuid.UUID) (*quotasmodels.QuotaStatus, error) {
		teacherRepository, err := quotasrepository.NewTeacherRepository(qs.db)
		if err != nil {
				return nil, fmt.Errorf("failed to create teacher repository: %w", err)
		}
		teacher, err := teacherRepository.Get(ctx, teacherID)
		if err != nil {
```

```
                if errors.Is(err, baserepository.ErrNoEntities) {
                        return nil, baseerrors.EntityNotFound
                }
                return nil, fmt.Errorf("failed to find teacher quota: %w", err)
        }
        totalQuota := teacher.TotalQuota()
        usedQuota := teacher.UsedQuota()
        return &quotasmodels.QuotaStatus{
                CPU:        totalQuota.CPU,
                RAM:        totalQuota.RAM,
                Storage:    totalQuota.Storage,
                UsedCPU:    usedQuota.CPU,
                UsedRAM:    usedQuota.RAM,
                UsedStorage: usedQuota.Storage,
        }, nil
}
func (qs *QuotaService) AcquireCourseQuota(
        ctx context.Context,
        teacherID uuid.UUID,
        courseID uuid.UUID,
        perEnrollmentQuota quotasmodels.QuotaRequest,
) error {
        repo, err := quotasrepository.NewTeacherRepository(qs.db)
        if err != nil {
                return fmt.Errorf("failed to create teacher repository: %w", err)
        }
        err = baseservice.WithTransaction(ctx, qs.db,
                baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
                        teacher, err := repo.Get(ctx, teacherID)
                        if err != nil {
                                return nil, fmt.Errorf("failed to find teacher by id %q: %w", teacherID, err)
                        }
                        _, err = teacher.AddCourse(courseID, quotasdomain.Resources{
                                CPU:    perEnrollmentQuota.CPU,
                                RAM:    perEnrollmentQuota.RAM,
                                Storage: perEnrollmentQuota.Storage,
                        })
                        if err != nil {
                                return nil, fmt.Errorf("failed to add course quota: %w", err)
                        }
                        err = repo.Save(ctx, teacher)
                        if err != nil {
                                return nil, fmt.Errorf("failed to save teacher: %w", err)
                        }
                        return nil, nil
                }))
        if err != nil {
                return fmt.Errorf("failed to acquire course quota: %w", err)
        }
        return nil
}
func (qs *QuotaService) ReleaseCourseQuota(ctx context.Context, courseID uuid.UUID) error {
        repo, err := quotasrepository.NewTeacherRepository(qs.db)
        if err != nil {
                return fmt.Errorf("failed to create teacher repository: %w", err)
        }
        err = baseservice.WithTransaction(ctx, qs.db,
                baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
                        teacher, err := repo.GetByCourseID(ctx, courseID)
                        if err != nil {
                                return nil, fmt.Errorf("failed to find teacher by course id %q: %w", courseID, err)
                        }
                        err = teacher.DeleteCourse(courseID)
                        if err != nil {
                                return nil, fmt.Errorf("failed to delete course: %w", err)
                        }
                        err = repo.Save(ctx, teacher)
                        if err != nil {
                                return nil, fmt.Errorf("failed to save teacher: %w", err)
                        }
                        return nil, nil
                }))
```

```
	if err != nil {
		return fmt.Errorf("failed to release course quota: %w", err)
	}
	return nil
}
func (qs *QuotaService) AcquireEnrollmentQuota(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID) error {
	teacherRepository, err := quotasrepository.NewTeacherRepository(qs.db)
	if err != nil {
		return fmt.Errorf("failed to create teacher repository: %w", err)
	}
	enrollmentRepository, err := quotasrepository.NewEnrollmentRepository(qs.db)
	if err != nil {
		return fmt.Errorf("failed to create enrollment repository: %w", err)
	}
	err = baseservice.WithTransaction(ctx, qs.db,
		baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
			teacher, err := teacherRepository.GetByCourseID(ctx, courseID)
			if err != nil {
				return nil, fmt.Errorf("failed to find teacher by course %q: %w", courseID, err)
			}
			enrollment, err := teacher.AcquireEnrollmentQuota(courseID, studentID)
			if err != nil {
				return nil, fmt.Errorf("failed to acquire enrollment quota: %w", err)
			}
			err = teacherRepository.Save(ctx, teacher)
			if err != nil {
				return nil, fmt.Errorf("failed to save teacher: %w", err)
			}
			err = enrollmentRepository.Insert(ctx, enrollment)
			if err != nil {
				return nil, fmt.Errorf("failed to insert enrollment: %w", err)
			}
			return nil, nil
		}))
	if err != nil {
		return fmt.Errorf("failed to acquire enrollment quota: %w", err)
	}
	return nil
}
func (qs *QuotaService) ReleaseEnrollmentQuota(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID) error {
	teacherRepository, err := quotasrepository.NewTeacherRepository(qs.db)
	if err != nil {
		return fmt.Errorf("failed to create teacher repository: %w", err)
	}
	enrollmentRepository, err := quotasrepository.NewEnrollmentRepository(qs.db)
	if err != nil {
		return fmt.Errorf("failed to create enrollment repository: %w", err)
	}
	err = baseservice.WithTransaction(ctx, qs.db,
		baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
			teacher, err := teacherRepository.GetByCourseID(ctx, courseID)
			if err != nil {
				return nil, fmt.Errorf("failed to find teacher by course %q: %w", courseID, err)
			}
			err = teacher.ReleaseEnrollmentQuota(courseID, studentID)
			if err != nil {
				return nil, fmt.Errorf("failed to release enrollment quota: %w", err)
			}
			err = teacherRepository.Save(ctx, teacher)
			if err != nil {
				return nil, fmt.Errorf("failed to save teacher: %w", err)
			}
			err = enrollmentRepository.Delete(ctx, courseID, studentID)
			if err != nil {
				return nil, fmt.Errorf("failed to delete enrollment: %w", err)
			}
			return nil, nil
		}))
	if err != nil {
		return fmt.Errorf("failed to release enrollment quota: %w", err)
	}
	return nil
```

```
}
func (qs *QuotaService) GetEnrollmentQuota(
	ctx context.Context,
	courseID uuid.UUID,
	studentID uuid.UUID,
) (*quotasmodels.QuotaStatus, error) {
	enrollmentRepository, err := quotasrepository.NewEnrollmentRepository(qs.db)
	if err != nil {
		return nil, fmt.Errorf("failed to create enrollment repository: %w", err)
	}
	enrollment, err := enrollmentRepository.Get(ctx, courseID, studentID)
	if err != nil {
		if errors.Is(err, baserepository.ErrNoEntities) {
			return nil, baseerrors.EntityNotFound
		}
		return nil, fmt.Errorf("failed to get enrollment quota: %w", err)
	}
	totalQuota := enrollment.TotalQuota()
	usedQuota := enrollment.UsedQuota()
	return &quotasmodels.QuotaStatus{
		CPU:        totalQuota.CPU,
		RAM:        totalQuota.RAM,
		Storage:    totalQuota.Storage,
		UsedCPU:    usedQuota.CPU,
		UsedRAM:    usedQuota.RAM,
		UsedStorage: usedQuota.Storage,
	}, nil
}
func (qs *QuotaService) AcquireServiceQuota(
	ctx context.Context,
	courseID uuid.UUID,
	studentID uuid.UUID,
	serviceID uuid.UUID,
	quota quotasmodels.QuotaRequest,
) error {
	repo, err := quotasrepository.NewEnrollmentRepository(qs.db)
	if err != nil {
		return fmt.Errorf("failed to create enrollment repository: %w", err)
	}
	err = baseservice.WithTransaction(ctx, qs.db,
		baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
			enrollment, err := repo.Get(ctx, courseID, studentID)
			if err != nil {
				return nil, fmt.Errorf("failed to find enrollment %q %q: %w", courseID, studentID, err)
			}
			_, err = enrollment.AddService(serviceID, quotasdomain.Resources{
				CPU:    quota.CPU,
				RAM:    quota.RAM,
				Storage: quota.Storage,
			})
			if err != nil {
				return nil, fmt.Errorf("failed to acquire service quota: %w", err)
			}
			err = repo.Save(ctx, enrollment)
			if err != nil {
				return nil, fmt.Errorf("failed to save enrollment: %w", err)
			}
			return nil, nil
		}))
	if err != nil {
		return fmt.Errorf("failed to acquire service quota: %w", err)
	}
	return nil
}
func (qs *QuotaService) ReleaseServiceQuota(
	ctx context.Context,
	courseID uuid.UUID,
	studentID uuid.UUID,
	serviceID uuid.UUID,
) error {
	repo, err := quotasrepository.NewEnrollmentRepository(qs.db)
	if err != nil {
```

```
                return fmt.Errorf("failed to create enrollment repository: %w", err)
        }
        err = baseservice.WithTransaction(ctx, qs.db,
                baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
                        enrollment, err := repo.Get(ctx, courseID, studentID)
                        if err != nil {
                                return nil, fmt.Errorf("failed to find enrollment %q %q: %w", courseID, studentID, err)
                        }
                        err = enrollment.DeleteService(serviceID)
                        if err != nil {
                                return nil, fmt.Errorf("failed to release service quota: %w", err)
                        }
                        err = repo.Save(ctx, enrollment)
                        if err != nil {
                                return nil, fmt.Errorf("failed to save enrollment: %w", err)
                        }
                        return nil, nil
                }))
        if err != nil {
                return fmt.Errorf("failed to release service quota: %w", err)
        }
        return nil
}
func (qs *QuotaService) ReleaseServiceQuotas(
        ctx context.Context,
        courseID uuid.UUID,
        userID uuid.UUID,
        serviceIDs []uuid.UUID,
) error {
        repo, err := quotasrepository.NewEnrollmentRepository(qs.db)
        if err != nil {
                return fmt.Errorf("failed to create enrollment repository: %w", err)
        }
        err = baseservice.WithTransaction(ctx, qs.db,
                baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
                        enrollment, err := repo.Get(ctx, courseID, userID)
                        if err != nil {
                                return nil, fmt.Errorf("failed to find enrollment %q %q: %w", courseID, userID, err)
                        }
                        for _, serviceID := range serviceIDs {
                                err = enrollment.DeleteService(serviceID)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to release service quota: %w", err)
                                }
                        }
                        err = repo.Save(ctx, enrollment)
                        if err != nil {
                                return nil, fmt.Errorf("failed to save enrollment: %w", err)
                        }
                        return nil, nil
                }))
        if err != nil {
                return fmt.Errorf("failed to release services quota: %w", err)
        }
        return nil
}
func (qs *QuotaService) ScaleServiceQuota(
        ctx context.Context,
        courseID uuid.UUID,
        studentID uuid.UUID,
        serviceID uuid.UUID,
        newQuota quotasmodels.QuotaRequest,
) error {
        repo, err := quotasrepository.NewEnrollmentRepository(qs.db)
        if err != nil {
                return fmt.Errorf("failed to create enrollment repository: %w", err)
        }
        err = baseservice.WithTransaction(ctx, qs.db,
                baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
                        enrollment, err := repo.Get(ctx, courseID, studentID)
                        if err != nil {
                                return nil, fmt.Errorf("failed to find enrollment %q %q: %w", courseID, studentID, err)
```

```
                                        }
                                        err = enrollment.ScaleService(serviceID, quotasdomain.Resources{
                                                CPU:     newQuota.CPU,
                                                RAM:     newQuota.RAM,
                                                Storage: newQuota.Storage,
                                        })
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to scale service quota: %w", err)
                                        }
                                        err = repo.Save(ctx, enrollment)
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to save enrollment: %w", err)
                                        }
                                        return nil, nil
                                }))
                if err != nil {
                        return fmt.Errorf("failed to scale service quota: %w", err)
                }
                return nil
}
func (qs *QuotaService) GetServiceQuota(ctx context.Context, serviceID uuid.UUID) (*quotasmodels.QuotaRequest, error) {
                repo, err := quotasrepository.NewEnrollmentRepository(qs.db)
                if err != nil {
                        return nil, fmt.Errorf("failed to create enrollment repository: %w", err)
                }
                enrollment, err := repo.GetByServiceID(ctx, serviceID)
                if err != nil {
                        return nil, fmt.Errorf("failed to find enrollment by service id %q: %w", serviceID, err)
                }
                quota, err := enrollment.GetServiceQuota(serviceID)
                if err != nil {
                        return nil, fmt.Errorf("failed to get service quota: %w", err)
                }
                return &quotasmodels.QuotaRequest{
                        CPU:     quota.CPU,
                        RAM:     quota.RAM,
                        Storage: quota.Storage,
                }, nil
}
func (qs *QuotaService) GetServiceQuotas(
                ctx context.Context,
                serviceIDs []uuid.UUID,
) (map[uuid.UUID]*quotasmodels.QuotaRequest, error) {
                result := make(map[uuid.UUID]*quotasmodels.QuotaRequest, len(serviceIDs))
                for _, serviceID := range serviceIDs {
                        quota, err := qs.GetServiceQuota(ctx, serviceID)
                        if err != nil {
                                return nil, fmt.Errorf("failed to get service quota: %w", err)
                        }
                        result[serviceID] = quota
                }
                return result, nil
}
```

quotas\quotasservice\quotasmodels\quota.go:

```
package quotasmodels
type QuotaRequest struct {
                CPU     int32 `json:"cpu" form:"cpu" query:"cpu"`
                RAM     int32 `json:"ram" form:"ram" query:"ram"`
                Storage int32 `json:"storage" form:"storage" query:"storage"`
}
type QuotaStatus struct {
                CPU        int32 `json:"cpu" form:"cpu" query:"cpu"`
                RAM        int32 `json:"ram" form:"ram" query:"ram"`
                Storage    int32 `json:"storage" form:"storage" query:"storage"`
                UsedCPU    int32 `json:"used_cpu" form:"used_cpu" query:"used_cpu"`
                UsedRAM    int32 `json:"used_ram" form:"used_ram" query:"used_ram"`
                UsedStorage int32 `json:"used_storage" form:"used_storage" query:"used_storage"`
}
```

services\kubernetesservice\kubernetesservice.go:

```
package kubernetesservice
import (...)
const mega int64 = 1024 * 1024
const fsGroup = 1000
type KubernetesService struct {
	client    *kubernetes.Clientset
	namespace string
}
type Config struct {
	KubeConfigPath string
	Namespace      string
}
func NewKubernetesService(serviceConfig Config) (*KubernetesService, error) {
	config, err := clientcmd.BuildConfigFromFlags("", serviceConfig.KubeConfigPath)
	if err != nil {
		return nil, fmt.Errorf("failed to parse config: %w", err)
	}
	clientset, err := kubernetes.NewForConfig(config)
	if err != nil {
		return nil, fmt.Errorf("failed to create kubernetes client: %w", err)
	}
	err = ensureNamespaceExists(clientset, serviceConfig.Namespace)
	if err != nil {
		return nil, fmt.Errorf("namespace does not exist: %w", err)
	}
	return &KubernetesService{
		client:    clientset,
		namespace: serviceConfig.Namespace,
	}, nil
}
func ensureNamespaceExists(client *kubernetes.Clientset, name string) error {
	_, err := client.CoreV1().Namespaces().Get(name, metav1.GetOptions{})
	if err != nil && errors.ReasonForError(err) != metav1.StatusReasonNotFound {
		return fmt.Errorf("failed to get namespace: %w", err)
	}
	if errors.ReasonForError(err) == metav1.StatusReasonNotFound {
		_, err := client.CoreV1().Namespaces().Create(&apiv1.Namespace{
			ObjectMeta: metav1.ObjectMeta{
				Name: name,
			},
		})
		if err != nil {
			return fmt.Errorf("failed to create namespace: %w", err)
		}
	}
	return nil
}
func getUniqueNameForID(id uuid.UUID) string {
	return "srv-" + id.String()
}
func getPVCName(serviceID uuid.UUID, pvcIndex int) string {
	return getUniqueNameForID(serviceID) + "-pvc-" + strconv.Itoa(pvcIndex)
}
func getVolumeName(volumeIndex int) string {
	return "volume-" + strconv.Itoa(volumeIndex)
}
func getLabels(domainService *servicesdomain.Service) map[string]string {
	return map[string]string{
		"app": getUniqueNameForID(domainService.ID),
	}
}
func (ks *KubernetesService) deployment(
	domainService *servicesdomain.Service,
	quota servicesdomain.Quota,
) *appsv1.Deployment {
	var replicas int32 = 0
	if domainService.DesiredState == servicesdomain.DesiredStateStarted {
		replicas = 1
	}
```

```
		uniqueName := getUniqueNameForID(domainService.ID)
		labels := getLabels(domainService)
		return &appsv1.Deployment{
			ObjectMeta: metav1.ObjectMeta{
				Name: uniqueName,
			},
			Spec: appsv1.DeploymentSpec{
				Replicas: &replicas,
				Selector: &metav1.LabelSelector{
					MatchLabels: labels,
				},
				Template: apiv1.PodTemplateSpec{
					ObjectMeta: metav1.ObjectMeta{
						Labels: labels,
					},
					Spec: apiv1.PodSpec{
						Containers: []apiv1.Container{
							{
								Name:         uniqueName,
								Image:        domainService.DockerImage,
								Ports:        ks.getContainerPorts(domainService),
								Resources:    ks.getResourceRequirements(quota),
								Env:          ks.getEnvs(domainService),
								VolumeMounts: ks.getVolumeMounts(domainService),
							},
						},
						Volumes: ks.getVolumes(domainService),
						SecurityContext: &apiv1.PodSecurityContext{
							FSGroup: int64Ptr(fsGroup),
						},
					},
				},
			},
		}
}
func (ks *KubernetesService) getResourceRequirements(quota servicesdomain.Quota) apiv1.ResourceRequirements {
		memoryLimit := mega * int64(quota.RAM)
		cpuLimit := int64(quota.CPU)
		return apiv1.ResourceRequirements{
			Limits: apiv1.ResourceList{
				apiv1.ResourceCPU:    *resource.NewMilliQuantity(cpuLimit, resource.DecimalSI),
				apiv1.ResourceMemory: *resource.NewQuantity(memoryLimit, resource.BinarySI),
			},
			Requests: apiv1.ResourceList{
				apiv1.ResourceCPU:    *resource.NewQuantity(0, resource.DecimalSI),
				apiv1.ResourceMemory: *resource.NewQuantity(0, resource.BinarySI),
			},
		}
}
func (ks *KubernetesService) getPersistentVolumeClaims(
		domainService *servicesdomain.Service,
) []apiv1.PersistentVolumeClaim {
		fsMode := apiv1.PersistentVolumeFilesystem
		accessModes := []apiv1.PersistentVolumeAccessMode{apiv1.ReadWriteOnce}
		persistentVolumeClaims := make([]apiv1.PersistentVolumeClaim, len(domainService.Volumes))
		for i := range persistentVolumeClaims {
			size := mega * int64(domainService.Volumes[i].Size)
			persistentVolumeClaims[i].ObjectMeta.Name = getPVCName(domainService.ID, i)
			persistentVolumeClaims[i].Spec.AccessModes = accessModes
			persistentVolumeClaims[i].Spec.VolumeMode = &fsMode
			persistentVolumeClaims[i].Spec.Resources.Requests = map[apiv1.ResourceName]resource.Quantity{
				apiv1.ResourceStorage: *resource.NewQuantity(size, resource.BinarySI),
			}
		}
		return persistentVolumeClaims
}
func (ks *KubernetesService) getServicePorts(domainService *servicesdomain.Service) []apiv1.ServicePort {
		ports := make([]apiv1.ServicePort, len(domainService.Ports))
		for i := range ports {
			ports[i].Port = domainService.Ports[i].Port
		}
		return ports
```

```
}
func (ks *KubernetesService) getContainerPorts(domainService *servicesdomain.Service) []apiv1.ContainerPort {
	ports := make([]apiv1.ContainerPort, len(domainService.Ports))
	for i := range ports {
		ports[i].ContainerPort = domainService.Ports[i].Port
	}
	return ports
}
func (ks *KubernetesService) getVolumes(domainService *servicesdomain.Service) []apiv1.Volume {
	volumes := make([]apiv1.Volume, len(domainService.Volumes))
	for i := range volumes {
		volumes[i].Name = getVolumeName(i)
		volumes[i].VolumeSource.PersistentVolumeClaim = &apiv1.PersistentVolumeClaimVolumeSource{
			ClaimName: getPVCName(domainService.ID, i),
			ReadOnly:  false,
		}
	}
	return volumes
}
func (ks *KubernetesService) getVolumeMounts(domainService *servicesdomain.Service) []apiv1.VolumeMount {
	volumeMounts := make([]apiv1.VolumeMount, len(domainService.Volumes))
	for i := range domainService.Volumes {
		volumeMounts[i].Name = getVolumeName(i)
		volumeMounts[i].MountPath = domainService.Volumes[i].Path
	}
	return volumeMounts
}
func (ks *KubernetesService) getEnvs(domainService *servicesdomain.Service) []apiv1.EnvVar {
	domainsServiceEnvs := domainService.AllEnvs()
	env := make([]apiv1.EnvVar, len(domainsServiceEnvs))
	for i := range domainsServiceEnvs {
		env[i].Name = domainsServiceEnvs[i].Key
		env[i].Value = domainsServiceEnvs[i].Value
	}
	return env
}
func (ks *KubernetesService) createService(domainService *servicesdomain.Service) *apiv1.Service {
	return &apiv1.Service{
		ObjectMeta: metav1.ObjectMeta{
			Name: getUniqueNameForID(domainService.ID),
		},
		Spec: apiv1.ServiceSpec{
			Ports:    ks.getServicePorts(domainService),
			Selector: getLabels(domainService),
			Type:     "NodePort",
		},
	}
}
func (ks *KubernetesService) DeleteService(domainService *servicesdomain.Service) error {
	deploymentsClient := ks.client.AppsV1().Deployments(ks.namespace)
	pvcClient := ks.client.CoreV1().PersistentVolumeClaims(ks.namespace)
	servicesClient := ks.client.CoreV1().Services(ks.namespace)
	err := deploymentsClient.Delete(getUniqueNameForID(domainService.ID), &metav1.DeleteOptions{})
	if err != nil {
		if errors.ReasonForError(err) != metav1.StatusReasonNotFound {
			return fmt.Errorf("failed to delete deployment: %w", err)
		}
	}
	pvcs := ks.getPersistentVolumeClaims(domainService)
	for i := range pvcs {
		err = pvcClient.Delete(pvcs[i].Name, &metav1.DeleteOptions{})
		if err != nil {
			if errors.ReasonForError(err) != metav1.StatusReasonNotFound {
				return fmt.Errorf("failed to delete pvc: %w", err)
			}
		}
	}
	err = servicesClient.Delete(getUniqueNameForID(domainService.ID), &metav1.DeleteOptions{})
	if err != nil {
		if errors.ReasonForError(err) != metav1.StatusReasonNotFound {
			return fmt.Errorf("failed to delete service: %w", err)
		}
```

```
		}
		return nil
}
func (ks *KubernetesService) applyPVCState(domainService *servicesdomain.Service) error {
		pvcClient := ks.client.CoreV1().PersistentVolumeClaims(ks.namespace)
		pvcs := ks.getPersistentVolumeClaims(domainService)
		for i := range pvcs {
				_, err := pvcClient.Get(pvcs[i].Name, metav1.GetOptions{})
				if err != nil {
						if errors.ReasonForError(err) != metav1.StatusReasonNotFound {
								return fmt.Errorf("failed to check existence of pvc: %w", err)
						}
						_, err := pvcClient.Create(&pvcs[i])
						if err != nil {
								return fmt.Errorf("failed to create persistent volume claim: %w", err)
						}
				}
		}
		return nil
}
func (ks *KubernetesService) applyDeploymentState(
		domainService *servicesdomain.Service,
		quota servicesdomain.Quota,
) error {
		deploymentsClient := ks.client.AppsV1().Deployments(ks.namespace)
		deployment := ks.deployment(domainService, quota)
		deploymentExists := true
		_, err := deploymentsClient.Get(deployment.Name, metav1.GetOptions{})
		if err != nil {
				if errors.ReasonForError(err) != metav1.StatusReasonNotFound {
						return fmt.Errorf("failed to get deployment: %w", err)
				}
				deploymentExists = false
		}
		if deploymentExists {
				_, err = deploymentsClient.Update(deployment)
				if err != nil {
						return fmt.Errorf("failed to update deployment: %w", err)
				}
		} else {
				_, err = deploymentsClient.Create(deployment)
				if err != nil {
						return fmt.Errorf("failed to create deployment: %w", err)
				}
		}
		return nil
}
func (ks *KubernetesService) applyServiceState(domainService *servicesdomain.Service) error {
		servicesClient := ks.client.CoreV1().Services(ks.namespace)
		serviceExists := true
		service, err := servicesClient.Get(getUniqueNameForID(domainService.ID), metav1.GetOptions{})
		if err != nil {
				if errors.ReasonForError(err) != metav1.StatusReasonNotFound {
						return fmt.Errorf("failed to get service: %w", err)
				}
				serviceExists = false
		}
		if serviceExists {
				service.Spec.Ports = ks.getServicePorts(domainService)
				_, err = servicesClient.Update(service)
				if err != nil {
						return fmt.Errorf("failed to update service: %w", err)
				}
		} else {
				service := ks.createService(domainService)
				_, err = servicesClient.Create(service)
				if err != nil {
						return fmt.Errorf("failed to create service: %w", err)
				}
		}
		return nil
}
```

```
func (ks *KubernetesService) ApplyDomainServiceState(
	domainService *servicesdomain.Service,
	quota servicesdomain.Quota,
) error {
	err := ks.applyPVCState(domainService)
	if err != nil {
		return fmt.Errorf("failed to apply pvc state: %w", err)
	}
	err = ks.applyDeploymentState(domainService, quota)
	if err != nil {
		return fmt.Errorf("failed to apply deployment state: %w", err)
	}
	err = ks.applyServiceState(domainService)
	if err != nil {
		return fmt.Errorf("failed to apply service state: %w", err)
	}
	return nil
}
func (ks *KubernetesService) GetServiceState(
	ctx context.Context,
	serviceID uuid.UUID,
) (*servicesdomain.ServiceState, error) {
	deploymentsClient := ks.client.AppsV1().Deployments(ks.namespace)
	deployment, err := deploymentsClient.Get(getUniqueNameForID(serviceID), metav1.GetOptions{})
	if err != nil {
		return nil, fmt.Errorf("failed to get deployment: %w", err)
	}
	servicesClient := ks.client.CoreV1().Services(ks.namespace)
	service, err := servicesClient.Get(getUniqueNameForID(serviceID), metav1.GetOptions{})
	if err != nil {
		return nil, fmt.Errorf("failed to get service: %w", err)
	}
	serviceState := &servicesdomain.ServiceState{}
	if deployment.Status.AvailableReplicas > 0 {
		serviceState.ExecutionState = servicesdomain.Started
	} else {
		serviceState.ExecutionState = servicesdomain.Stopped
	}
	for i := range service.Spec.Ports {
		serviceState.PortMappings = append(serviceState.PortMappings, servicesdomain.PortMapping{
			ServicePort:  servicesdomain.Port{Port: service.Spec.Ports[i].Port},
			ExternalPort: servicesdomain.Port{Port: service.Spec.Ports[i].NodePort},
		})
	}
	return serviceState, nil
}
func (ks *KubernetesService) GetServiceStates(
	ctx context.Context,
	serviceIDs []uuid.UUID,
) (map[uuid.UUID]*servicesdomain.ServiceState, error) {
	result := make(map[uuid.UUID]*servicesdomain.ServiceState, len(serviceIDs))
	for _, serviceID := range serviceIDs {
		serviceState, err := ks.GetServiceState(ctx, serviceID)
		if err != nil {
			return nil, fmt.Errorf("failed to get service state: %w", err)
		}
		result[serviceID] = serviceState
	}
	return result, nil
}
func int64Ptr(i int64) *int64 { return &i }
```

services\servicesdomain\env.go:

```
package servicesdomain
import (...)
type Env struct {
	Name  string
	Key   string
	Value string
}
```

```
func NewEnv(name string, key string, value string) (*Env, error) {
	if name == "" {
		return nil, baseerrors.NewBusinessRule("invalidRequest", "name must not be empty")
	}
	if key == "" {
		return nil, baseerrors.NewBusinessRule("invalidRequest", "key must not be empty")
	}
	return &Env{Name: name, Key: key, Value: value}, nil
}
func (e Env) WithValue(value string) Env {
	return Env{
		Name:  e.Name,
		Key:   e.Key,
		Value: value,
	}
}
```

services\servicesdomain\quota.go:

```
package servicesdomain
type Quota struct {
	CPU int32
	RAM int32
}
```

services\servicesdomain\service.go:

```
package servicesdomain
import (...)
type Service struct {
	ID         uuid.UUID
	CourseID    uuid.UUID
	StudentID   uuid.UUID
	TemplateID  uuid.UUID
	Name       string
	DockerImage string
	PendingSync bool
	Ports     []Port
	FixedEnvs []Env
	UserEnvs  []Env
	Volumes   []Volume
	DesiredState string
}
const (
	DesiredStateStarted = "started"
	DesiredStateStopped = "stopped"
	DesiredStateDeleted = "deleted"
)
func (s *Service) MarkAsSynced() error {
	if !s.PendingSync {
		return baseerrors.NewBusinessRule("alreadySynced", "already marked as synced")
	}
	s.PendingSync = false
	return nil
}
func (s *Service) MarkAsPendingSync() error {
	if s.PendingSync {
		return baseerrors.NewBusinessRule("alreadyPending", "already marked as pending sync")
	}
	s.markAsPendingSync()
	return nil
}
func (s *Service) markAsPendingSync() {
	s.PendingSync = true
}
func checkDesiredState(desiredState string) error {
	if desiredState != DesiredStateStarted && desiredState != DesiredStateDeleted && desiredState != DesiredStateStopped {
		message := fmt.Sprintf("wrong desired state: %q", desiredState)
		return baseerrors.NewBusinessRule("wrongDesiredState", message)
	}
	return nil
```

```
}
func (s *Service) SetDesiredState(desiredState string) error {
	if err := checkDesiredState(desiredState); err != nil {
		return err
	}
	s.DesiredState = desiredState
	s.markAsPendingSync()
	return nil
}
func (s *Service) getUserEnvMap() map[string]Env {
	result := make(map[string]Env, len(s.UserEnvs))
	for i := range s.UserEnvs {
		result[s.UserEnvs[i].Name] = s.UserEnvs[i]
	}
	return result
}
func (s *Service) SetUserEnvs(userEnvs []Env) error {
	filteredUserEnvs := make([]Env, 0, len(s.UserEnvs))
	userEnvMap := s.getUserEnvMap()
	for _, v := range userEnvs {
		if _, ok := userEnvMap[v.Name]; !ok {
			message := fmt.Sprintf("provided env %q is not allowed in this service", v.Name)
			return baseerrors.NewBusinessRule("envIsNotAllowed", message)
		}
		filteredUserEnvs = append(filteredUserEnvs, userEnvMap[v.Name].WithValue(v.Value))
		delete(userEnvMap, v.Name)
	}
	for _, v := range userEnvMap {
		filteredUserEnvs = append(filteredUserEnvs, v)
	}
	s.UserEnvs = filteredUserEnvs
	s.markAsPendingSync()
	return nil
}
func (s *Service) AllEnvs() []Env {
	allEnvs := make([]Env, 0, len(s.UserEnvs)+len(s.FixedEnvs))
	allEnvs = append(allEnvs, s.FixedEnvs...)
	allEnvs = append(allEnvs, s.UserEnvs...)
	return allEnvs
}
func (s *Service) RequestedStorageQuota() int32 {
	var quota int32 = 0
	for i := range s.Volumes {
		quota += s.Volumes[i].Size
	}
	return quota
}
func NewService(
	ID uuid.UUID,
	courseID uuid.UUID,
	studentID uuid.UUID,
	templateID uuid.UUID,
	name string,
	dockerImage string,
	pendingSync bool,
	ports []Port,
	fixedEnvs []Env,
	userEnvs []Env,
	volumes []Volume,
	desiredState string,
) (*Service, error) {
	if err := checkDesiredState(desiredState); err != nil {
		return nil, err
	}
	return &Service{
		ID:          ID,
		CourseID:    courseID,
		StudentID:   studentID,
		TemplateID:  templateID,
		Name:        name,
		DockerImage: dockerImage,
		PendingSync: pendingSync,
```

```
                Ports:       ports,
                FixedEnvs:    fixedEnvs,
                UserEnvs:     userEnvs,
                Volumes:      volumes,
                DesiredState: desiredState,
        }, nil
}
func CreateService(
        courseID uuid.UUID,
        studentID uuid.UUID,
        templateID uuid.UUID,
        name string,
        dockerImage string,
        ports []Port,
        fixedEnvs []Env,
        userEnvs []Env,
        volumes []Volume,
) (*Service, error) {
        return &Service{
                ID:          uuid.Must(uuid.NewV4()),
                CourseID:     courseID,
                StudentID:    studentID,
                TemplateID:   templateID,
                Name:         name,
                DockerImage:  dockerImage,
                PendingSync:  true,
                Ports:       ports,
                FixedEnvs:    fixedEnvs,
                UserEnvs:     userEnvs,
                Volumes:      volumes,
                DesiredState: DesiredStateStarted,
        }, nil
}
```

services\servicesdomain\serviceport.go:

```
package servicesdomain
type Port struct {
        Port int32
}
func NewPort(port int32) (*Port, error) {
        return &Port{Port: port}, nil
}
```

services\servicesdomain\servicestate.go:

```
package servicesdomain
import (...)
type ExecutionState int
const (
        Stopped ExecutionState = iota
        Starting
        Started
        Stopping
)
type ServiceState struct {
        ID             uuid.UUID
        ExecutionState ExecutionState
        PortMappings   []PortMapping
}
type PortMapping struct {
        ServicePort  Port
        ExternalPort Port
}
```

services\servicesdomain\template.go:

```
package servicesdomain
import (...)
type Template struct {
        ID         uuid.UUID
```

```
	Name        string
	DockerImage string
	Ports     []Port
	FixedEnvs []Env
	UserEnvs  []Env
	Volumes   []Volume
}
func CreateTemplate(
	name string,
	dockerImage string,
	ports []Port,
	fixedEnvs []Env,
	userEnvs []Env,
	volumes []Volume,
) (*Template, error) {
	return &Template{
		ID:         uuid.Must(uuid.NewV4()),
		Name:       name,
		DockerImage: dockerImage,
		Ports:      ports,
		FixedEnvs:   fixedEnvs,
		UserEnvs:    userEnvs,
		Volumes:     volumes,
	}, nil
}
func NewTemplate(
	id uuid.UUID,
	name string,
	dockerImage string,
	ports []Port,
	fixedEnvs []Env,
	userEnvs []Env,
	volumes []Volume,
) (*Template, error) {
	return &Template{
		ID:         id,
		Name:       name,
		DockerImage: dockerImage,
		Ports:      ports,
		FixedEnvs:   fixedEnvs,
		UserEnvs:    userEnvs,
		Volumes:     volumes,
	}, nil
}
func (t *Template) getUserEnvMap() map[string]Env {
	result := make(map[string]Env, len(t.UserEnvs))
	for i := range t.UserEnvs {
		result[t.UserEnvs[i].Name] = t.UserEnvs[i]
	}
	return result
}
func (t *Template) getVolumesMap() map[string]Volume {
	result := make(map[string]Volume, len(t.Volumes))
	for i := range t.Volumes {
		result[t.Volumes[i].Path] = t.Volumes[i]
	}
	return result
}
func (t *Template) ConstructService(
	courseID uuid.UUID,
	studentID uuid.UUID,
	name string,
	userEnvs []Env,
	volumes []Volume,
) (*Service, error) {
	filteredUserEnvs := make([]Env, 0, len(t.UserEnvs))
	userEnvMap := t.getUserEnvMap()
	for _, v := range userEnvs {
		if _, ok := userEnvMap[v.Name]; !ok {
			message := fmt.Sprintf("provided env %q is not allowed in this template", v.Name)
			return nil, baseerrors.NewBusinessRule("envIsNotAllowed", message)
		}
```

```
                filteredUserEnvs = append(filteredUserEnvs, userEnvMap[v.Name].WithValue(v.Value))
                delete(userEnvMap, v.Name)
        }
        for _, v := range userEnvMap {
                filteredUserEnvs = append(filteredUserEnvs, v)
        }
        volumesMap := t.getVolumesMap()
        for _, v := range volumes {
                if _, ok := volumesMap[v.Path]; !ok {
                        message := fmt.Sprintf("usage of volume path %q which was not declared in template", v.Path)
                        return nil, baseerrors.NewBusinessRule("pathIsNotAllowed", message)
                }
        }
        service, err := CreateService(
                courseID,
                studentID,
                t.ID,
                name,
                t.DockerImage,
                t.Ports,
                t.FixedEnvs,
                filteredUserEnvs,
                volumes,
        )
        if err != nil {
                return nil, fmt.Errorf("failed to create service: %w", err)
        }
        return service, nil
}
```

services\servicesdomain\volume.go:

```
package servicesdomain
import (...)
type Volume struct {
        Path string
        Size int32
}
func NewVolume(path string, size int32) (*Volume, error) {
        if size <= 0 {
                return nil, baseerrors.NewBusinessRule("invalidVolumeSize", "volume size should be greater than 0")
        }
        return &Volume{Path: path, Size: size}, nil
}
```

services\serviceshttpapi\addservice.go:

```
package serviceshttpapi
import (...)
type IDResponse struct {
        ID string `json:"id" form:"id" query:"id"`
}
func (api *API) addServiceRoute(c echo.Context) error {
        user := basehttpapi.GetUser(c).(*usersservice.User)
        serviceRequest := new(servicemodels.ServiceRequest)
        if err := c.Bind(serviceRequest); err != nil {
                return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("could not bind request: %s", err))
        }
        courseID, err := uuid.FromString(c.Param("courseID"))
        if err != nil {
                return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("illformatted course id: %q", c.Param("courseID")))
        }
        enrollmentID, err := basehttpapi.HandleMeUserID(user.ID, c.Param("enrollmentID"), func() (bool, error) {
                return api.coursesService.CanManage(context.Background(), user, courseID)
        })
        if err != nil {
                return err
        }
        id, err := api.servicesService.Create(context.Background(), courseID, enrollmentID, serviceRequest)
        if err != nil {
                var businessRuleErr *baseerrors.BusinessRule
```

```
                if errors.As(err, &businessRuleErr) {
                        return echo.NewHTTPError(http.StatusBadRequest, businessRuleErr.Code)
                }
                log.Errorf("failed to create service: %v", err)
                return echo.NewHTTPError(http.StatusInternalServerError, "failed to create service")
        }
        return c.JSON(http.StatusCreated, IDResponse{ID: id})
}
```

services\serviceshttpapi\addservicetemplate.go:

```
package serviceshttpapi
import (...)
func (api *API) addServiceTemplateRoute(c echo.Context) error {
        user := basehttpapi.GetUser(c).(*usersservice.User)
        serviceTemplate := new(servicemodels.ServiceTemplate)
        if err := c.Bind(serviceTemplate); err != nil {
                return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("could not bind request: %s", err))
        }
        if !user.IsTeacher() && !user.IsAdmin() {
                return echo.NewHTTPError(http.StatusForbidden, "only for teachers and admins")
        }
        id, err := api.serviceTemplatesService.Create(context.Background(), serviceTemplate)
        if err != nil {
                var businessRuleErr *baseerrors.BusinessRule
                if errors.As(err, &businessRuleErr) {
                        return echo.NewHTTPError(http.StatusBadRequest, businessRuleErr.Code)
                }
                log.Errorf("failed to create service template: %v", err)
                return echo.NewHTTPError(http.StatusInternalServerError, "failed to create service template")
        }
        return c.JSON(http.StatusCreated, IDResponse{ID: id})
}
```

services\serviceshttpapi\deleteservice.go:

```
package serviceshttpapi
import (...)
func (api *API) deleteServiceRoute(c echo.Context) error {
        user := basehttpapi.GetUser(c).(*usersservice.User)
        serviceID, err := uuid.FromString(c.Param("serviceID"))
        if err != nil {
                log.Warnf("invalid service id %q: %v", c.Param("serviceID"), err)
                return echo.NewHTTPError(http.StatusBadRequest, "invalid service id")
        }
        if err := basehttpapi.CheckAccess(func() (bool, error) {
                return api.servicePermissionsService.CanManage(context.Background(), user, serviceID)
        }); err != nil {
                return err
        }
        err = api.servicesService.DeleteService(context.Background(), serviceID)
        if err != nil {
                if errors.Is(err, baseerrors.EntityNotFound) {
                        return echo.NewHTTPError(http.StatusNotFound, "service not found")
                }
                log.Errorf("failed to delete service: %v", err)
                return echo.NewHTTPError(http.StatusInternalServerError, "failed to delete service")
        }
        return c.NoContent(http.StatusOK)
}
```

services\serviceshttpapi\getservice.go:

```
package serviceshttpapi
import (...)
func (api *API) getServiceRoute(c echo.Context) error {
        user := basehttpapi.GetUser(c).(*usersservice.User)
        serviceID, err := uuid.FromString(c.Param("serviceID"))
        if err != nil {
                log.Warnf("invalid service id %q: %v", c.Param("serviceID"), err)
                return echo.NewHTTPError(http.StatusBadRequest, "invalid service id")
```

```
	}
	if err := basehttpapi.CheckAccess(func() (bool, error) {
		return api.servicePermissionsService.CanManage(context.Background(), user, serviceID)
	}); err != nil {
		return err
	}
	service, err := api.servicesService.Get(context.Background(), serviceID)
	if err != nil {
		if errors.Is(err, baseerrors.EntityNotFound) {
			return echo.NewHTTPError(http.StatusNotFound, "service not found")
		}
		log.Errorf("failed to get service: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get service")
	}
	return c.JSON(http.StatusOK, service)
}
```

services\serviceshttpapi\getservices.go:

```
package serviceshttpapi
import (...)
func (api *API) getServicesRoute(c echo.Context) error {
	user := basehttpapi.GetUser(c).(*usersservice.User)
	courseID, err := uuid.FromString(c.Param("courseID"))
	if err != nil {
		log.Warnf("invalid course id %q: %v", c.Param("courseID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid course id")
	}
	enrollmentID, err := basehttpapi.HandleMeUserID(user.ID, c.Param("enrollmentID"), func() (bool, error) {
		return api.coursesService.CanManage(context.Background(), user, courseID)
	})
	if err != nil {
		return err
	}
	services, err := api.servicesService.GetByEnrollmentID(context.Background(), courseID, enrollmentID)
	if err != nil {
		if errors.Is(err, baseerrors.EntityNotFound) {
			return echo.NewHTTPError(http.StatusNotFound, "service not found")
		}
		log.Errorf("failed to get services: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get services")
	}
	return c.JSON(http.StatusOK, services)
}
```

services\serviceshttpapi\getservicetemplate.go:

```
package serviceshttpapi
import (...)
func (api *API) getServiceTemplateRoute(c echo.Context) error {
	serviceTemplateID, err := uuid.FromString(c.Param("serviceTemplateID"))
	if err != nil {
		log.Warnf("invalid service template id %q: %v", c.Param("serviceTemplateID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid service template id")
	}
	serviceTemplate, err := api.serviceTemplatesService.Get(context.Background(), serviceTemplateID)
	if err != nil {
		if errors.Is(err, baseerrors.EntityNotFound) {
			return echo.NewHTTPError(http.StatusNotFound, "service template not found")
		}
		log.Errorf("failed to get service template: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get service template")
	}
	return c.JSON(http.StatusOK, serviceTemplate)
}
```

services\serviceshttpapi\getservicetemplates.go:

```
package serviceshttpapi
import (...)
func (api *API) getServiceTemplatesRoute(c echo.Context) error {
```

```
	services, err := api.serviceTemplatesService.GetAll(context.Background())
	if err != nil {
		log.Errorf("failed to get service templates: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to get service templates")
	}
	return c.JSON(http.StatusOK, services)
}
```

services\serviceshttpapi\patchservice.go:

```
package serviceshttpapi
import (...)
func (api *API) patchServiceRoute(c echo.Context) error {
	user := basehttpapi.GetUser(c).(*usersservice.User)
	serviceRequest := new(servicemodels.ServicePatchRequest)
	if err := c.Bind(serviceRequest); err != nil {
		return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("could not bind request: %s", err))
	}
	serviceID, err := uuid.FromString(c.Param("serviceID"))
	if err != nil {
		log.Warnf("invalid service id %q: %v", c.Param("serviceID"), err)
		return echo.NewHTTPError(http.StatusBadRequest, "invalid service id")
	}
	if err := basehttpapi.CheckAccess(func() (bool, error) {
		return api.servicePermissionsService.CanManage(context.Background(), user, serviceID)
	}); err != nil {
		return err
	}
	service, err := api.servicesService.ChangeService(context.Background(), serviceID, serviceRequest)
	if err != nil {
		if errors.Is(err, baseerrors.EntityNotFound) {
			return echo.NewHTTPError(http.StatusNotFound, "service not found")
		}
		log.Errorf("failed to patch service: %v", err)
		return echo.NewHTTPError(http.StatusInternalServerError, "failed to patch service")
	}
	return c.JSON(http.StatusOK, service)
}
```

services\serviceshttpapi\router.go:

```
package serviceshttpapi
import (...)
func (api *API) RegisterRoutes(echo *echo.Echo) {
	echo.POST("/courses/:courseID/enrollments/:enrollmentID/services", api.addServiceRoute)
	echo.GET("/courses/:courseID/enrollments/:enrollmentID/services", api.getServicesRoute)
	echo.GET("/services/:serviceID", api.getServiceRoute)
	echo.PATCH("/services/:serviceID", api.patchServiceRoute)
	echo.DELETE("/services/:serviceID", api.deleteServiceRoute)
	echo.GET("/serviceTemplates", api.getServiceTemplatesRoute)
	echo.POST("/serviceTemplates", api.addServiceTemplateRoute)
	echo.GET("/serviceTemplates/:serviceTemplateID", api.getServiceTemplateRoute)
}
```

services\serviceshttpapi\serviceshttpapi.go:

```
package serviceshttpapi
import (...)
type API struct {
	servicesService         *servicesservice.ServicesService
	servicePermissionsService *servicesservice.ServicePermissionsService
	serviceTemplatesService   *servicesservice.ServiceTemplatesService
	coursesService          *coursesservice.CourseService
}
func NewAPI(
	servicesService *servicesservice.ServicesService,
	serviceTemplatesService *servicesservice.ServiceTemplatesService,
	servicePermissionsService *servicesservice.ServicePermissionsService,
	coursesService *coursesservice.CourseService,
) (*API, error) {
	return &API{
```

```
                servicesService:          servicesService,
                serviceTemplatesService:  serviceTemplatesService,
                servicePermissionsService: servicePermissionsService,
                coursesService:           coursesService,
        }, nil
}
```

services\servicesservice\converters.go:

```
package servicesservice
import (...)
func convertPortsModelToPorts(portsModel []int32) ([]servicesdomain.Port, error) {
        ports := make([]servicesdomain.Port, len(portsModel))
        for i := range ports {
                port, err := servicesdomain.NewPort(portsModel[i])
                if err != nil {
                        return nil, fmt.Errorf("failed to create new service port domain object: %w", err)
                }
                ports[i] = *port
        }
        return ports, nil
}
func convertPortsToPortsModel(ports []servicesdomain.Port) []int32 {
        portsModel := make([]int32, len(ports))
        for i := range ports {
                portsModel[i] = ports[i].Port
        }
        return portsModel
}
func convertEnvsModelToEnvs(envsModel []servicemodels.Env) ([]servicesdomain.Env, error) {
        envs := make([]servicesdomain.Env, len(envsModel))
        for i := range envs {
                env, err := servicesdomain.NewEnv(envsModel[i].Name, envsModel[i].Key, envsModel[i].Value)
                if err != nil {
                        return nil, fmt.Errorf("failed to create new service env domain object: %w", err)
                }
                envs[i] = *env
        }
        return envs, nil
}
func convertEnvsToEnvsModel(envs []servicesdomain.Env) []servicemodels.Env {
        envsModel := make([]servicemodels.Env, len(envs))
        for i := range envs {
                envsModel[i] = servicemodels.Env{
                        Name:  envs[i].Name,
                        Key:   envs[i].Key,
                        Value: envs[i].Value,
                }
        }
        return envsModel
}
func convertVolumesModelToVolumes(volumesModel []servicemodels.Volume) ([]servicesdomain.Volume, error) {
        volumes := make([]servicesdomain.Volume, len(volumesModel))
        for i := range volumes {
                volume, err := servicesdomain.NewVolume(volumesModel[i].Path, volumesModel[i].Size)
                if err != nil {
                        return nil, fmt.Errorf("failed to create new service volume domain object: %w", err)
                }
                volumes[i] = *volume
        }
        return volumes, nil
}
func convertVolumesToVolumesModel(volumes []servicesdomain.Volume) []servicemodels.Volume {
        volumesModel := make([]servicemodels.Volume, len(volumes))
        for i := range volumes {
                volumesModel[i] = servicemodels.Volume{
                        Path: volumes[i].Path,
                        Size: volumes[i].Size,
                }
        }
        return volumesModel
```

```
}
func convertPortMappingToPortMappingModel(portMappings []servicesdomain.PortMapping) []servicemodels.PortMapping {
	portMappingsModel := make([]servicemodels.PortMapping, len(portMappings))
	for i := range portMappings {
		portMappingsModel[i] = servicemodels.PortMapping{
			ServicePort:  portMappings[i].ServicePort.Port,
			ExternalPort: portMappings[i].ExternalPort.Port,
		}
	}
	return portMappingsModel
}
func convertToServiceState(
	serviceEntity *servicesdomain.Service,
	quota *quotasmodels.QuotaRequest,
	kubernetesState *servicesdomain.ServiceState,
) *servicemodels.ServiceState {
	serviceState := &servicemodels.ServiceState{
		ID:          serviceEntity.ID.String(),
		CourseID:    serviceEntity.CourseID.String(),
		StudentID:   serviceEntity.StudentID.String(),
		TemplateID:  serviceEntity.TemplateID.String(),
		Name:        serviceEntity.Name,
		DockerImage: serviceEntity.DockerImage,
		Ports:       convertPortsToPortsModel(serviceEntity.Ports),
		UserEnvs:    convertEnvsToEnvsModel(serviceEntity.UserEnvs),
		Volumes:     convertVolumesToVolumesModel(serviceEntity.Volumes),
		Quota: servicemodels.Quota{
			CPU:     quota.CPU,
			RAM:     quota.RAM,
			Storage: quota.Storage,
		},
	}
	if kubernetesState == nil {
		serviceState.State = "unknown"
		serviceState.PortMappings = make([]servicemodels.PortMapping, 0)
		return serviceState
	}
	if kubernetesState.ExecutionState == servicesdomain.Started {
		if serviceEntity.DesiredState == servicesdomain.DesiredStateStarted {
			serviceState.State = "started"
		} else {
			serviceState.State = "stopping"
		}
	}
	if kubernetesState.ExecutionState == servicesdomain.Stopped {
		if serviceEntity.DesiredState == servicesdomain.DesiredStateStarted {
			serviceState.State = "starting"
		} else {
			serviceState.State = "stopped"
		}
	}
	serviceState.PortMappings = convertPortMappingToPortMappingModel(kubernetesState.PortMappings)
	return serviceState
}
func convertServiceTemplateToServiceTemplateModel(template *servicesdomain.Template) *servicemodels.ServiceTemplate {
	return &servicemodels.ServiceTemplate{
		ID:          template.ID.String(),
		Name:        template.Name,
		DockerImage: template.DockerImage,
		Ports:       convertPortsToPortsModel(template.Ports),
		FixedEnvs:   convertEnvsToEnvsModel(template.FixedEnvs),
		UserEnvs:    convertEnvsToEnvsModel(template.UserEnvs),
		Volumes:     convertVolumesToVolumesModel(template.Volumes),
	}
}
```

services\servicesservice\kubernetessyncer.go:

```
package servicesservice
import (...)
type KubernetesService interface {
```

```
	ApplyDomainServiceState(service *servicesdomain.Service, quota servicesdomain.Quota) error
	DeleteService(service *servicesdomain.Service) error
	GetServiceState(ctx context.Context, serviceID uuid.UUID) (*servicesdomain.ServiceState, error)
	GetServiceStates(ctx context.Context, serviceIDs []uuid.UUID) (map[uuid.UUID]*servicesdomain.ServiceState, error)
}
type KubernetesSynchronizer struct {
	db		*mongo.Database
	kubernetesService KubernetesService
	quotasViewService QuotaView
}
type QuotaView interface {
	GetServiceQuota(ctx context.Context, serviceID uuid.UUID) (*quotasmodels.QuotaRequest, error)
}
func NewKubernetesSynchronizer(
	db *mongo.Database,
	kubernetesService KubernetesService,
	quotasViewService QuotaView,
) (*KubernetesSynchronizer, error) {
	return &KubernetesSynchronizer{
		db:		db,
		kubernetesService: kubernetesService,
		quotasViewService: quotasViewService,
	}, nil
}
func (ks *KubernetesSynchronizer) SyncOne(ctx context.Context) error {
	serviceRepository, err := servicesrepository.NewServiceRepository(ks.db)
	if err != nil {
		return fmt.Errorf("failed to create service repository: %w", err)
	}
	pendingService, err := serviceRepository.FindOnePendingSync(ctx)
	if err != nil {
		if errors.Is(err, baserepository.ErrNoEntities) {
			return nil
		}
		return fmt.Errorf("failed to fetch pending services: %w", err)
	}
	log.Printf("applying service state: %s", pendingService.ID)
	if pendingService.DesiredState == servicesdomain.DesiredStateDeleted {
		err = ks.kubernetesService.DeleteService(pendingService)
		if err != nil {
			return fmt.Errorf("failed to delete service: %w", err)
		}
	} else {
		quota, err := ks.quotasViewService.GetServiceQuota(ctx, pendingService.ID)
		if err != nil {
			return fmt.Errorf("failed to get service quota: %w", err)
		}
		err = ks.kubernetesService.ApplyDomainServiceState(pendingService, servicesdomain.Quota{
			CPU: quota.CPU,
			RAM: quota.RAM,
		})
		if err != nil {
			return fmt.Errorf("failed to apply service state: %w", err)
		}
	}
	err = pendingService.MarkAsSynced()
	if err != nil {
		return fmt.Errorf("failed to mark service as synced: %w", err)
	}
	err = serviceRepository.Save(ctx, pendingService)
	if err != nil {
		return fmt.Errorf("failed to save service: %w", err)
	}
	log.Printf("applied service state: %s", pendingService.ID)
	return nil
}
```

services\servicesservice\servicemodels\env.go:

```
package servicemodels
type Env struct {
```

```
	Name  string `json:"name" form:"name" query:"name"`
	Key   string `json:"key" form:"key" query:"key"`
	Value string `json:"value" form:"value" query:"value"`
}
```

services\servicesservice\servicemodels\portmapping.go:

```
package servicemodels
type PortMapping struct {
	ServicePort  int32 `json:"service_port" form:"service_port" query:"service_port"`
	ExternalPort int32 `json:"external_port" form:"external_port" query:"external_port"`
}
```

services\servicesservice\servicemodels\quota.go:

```
package servicemodels
type Quota struct {
	CPU     int32 `json:"cpu" form:"cpu" query:"cpu"`
	RAM     int32 `json:"ram" form:"ram" query:"ram"`
	Storage int32 `json:"storage" form:"storage" query:"storage"`
}
```

services\servicesservice\servicemodels\service.go:

```
package servicemodels
type Service struct {
	Name        string   `json:"name" form:"name" query:"name"`
	DockerImage string   `json:"docker_image" form:"docker_image" query:"docker_image"`
	Ports       []int32  `json:"ports" form:"ports" query:"ports"`
	Envs        []Env    `json:"envs" form:"envs" query:"envs"`
	Volumes     []Volume `json:"volumes" form:"volumes" query:"volumes"`
	Quota       Quota    `json:"quota" form:"quota" query:"quota"`
}
```

services\servicesservice\servicemodels\servicerequest.go:

```
package servicemodels
import (...)
type ServicePatchRequest struct {
	UserEnvs []Env        `json:"user_envs,omitempty" form:"user_envs" query:"user_envs"`
	State    string       `json:"state,omitempty" form:"state" query:"state"`
	Quota    *QuotaRequest `json:"quota,omitempty" form:"quota" query:"quota"`
}
type ServiceRequest struct {
	Name       string       `json:"name" form:"name" query:"name"`
	TemplateID uuid.UUID    `json:"template_id" form:"template_id" query:"template_id"`
	UserEnvs   []Env        `json:"user_envs" form:"user_envs" query:"user_envs"`
	Volumes    []Volume     `json:"volumes" form:"volumes" query:"volumes"`
	Quota      QuotaRequest `json:"quota" form:"quota" query:"quota"`
}
type QuotaRequest struct {
	CPU int32 `json:"cpu" form:"cpu" query:"cpu"`
	RAM int32 `json:"ram" form:"ram" query:"ram"`
}
```

services\servicesservice\servicemodels\servicestate.go:

```
package servicemodels
type ServiceState struct {
	ID          string       `json:"id" form:"id" query:"id"`
	CourseID    string       `json:"course_id" form:"course_id" query:"course_id"`
	StudentID   string       `json:"student_id" form:"student_id" query:"student_id"`
	TemplateID  string       `json:"template_id" form:"template_id" query:"template_id"`
	Name        string       `json:"name" form:"name" query:"name"`
	DockerImage string       `json:"docker_image" form:"docker_image" query:"docker_image"`
	Ports       []int32      `json:"ports" form:"ports" query:"ports"`
	UserEnvs    []Env        `json:"user_envs" form:"user_envs" query:"user_envs"`
	Volumes     []Volume     `json:"volumes" form:"volumes" query:"volumes"`
	Quota       Quota        `json:"quota" form:"quota" query:"quota"`
	State       string       `json:"state" form:"state" query:"state"`
```

```
	PortMappings []PortMapping `json:"port_mappings" form:"port_mappings" query:"port_mappings"`
}
```

services\servicesservice\servicemodels\servicetemplate.go:

```
package servicemodels
type ServiceTemplate struct {
	ID          string   `json:"id" form:"id" query:"id"`
	Name        string   `json:"name" form:"name" query:"name"`
	DockerImage string   `json:"docker_image" form:"docker_image" query:"docker_image"`
	Ports       []int32  `json:"ports" form:"ports" query:"ports"`
	FixedEnvs   []Env    `json:"fixed_envs,omitempty" form:"fixed_envs,omitempty" query:"fixed_envs,omitempty"`
	UserEnvs    []Env    `json:"user_envs" form:"user_envs" query:"user_envs"`
	Volumes     []Volume `json:"volumes" form:"volumes" query:"volumes"`
}
```

services\servicesservice\servicemodels\volume.go:

```
package servicemodels
type Volume struct {
	Path string `json:"path" form:"path" query:"path"`
	Size int32  `json:"size" form:"size" query:"size"`
}
```

services\servicesservice\servicepermissions.go:

```
package servicesservice
import (...)
type ServicePermissionsService struct {
	db             *mongo.Database
	coursesService CoursesService
}
type CoursesService interface {
	CanManage(ctx context.Context, user *usersservice.User, courseID uuid.UUID) (bool, error)
}
func NewServicePermissionsService(
	db *mongo.Database,
	coursesService CoursesService,
) (*ServicePermissionsService, error) {
	return &ServicePermissionsService{
		db:             db,
		coursesService: coursesService,
	}, nil
}
func (srv *ServicePermissionsService) CanManage(
	ctx context.Context,
	user *usersservice.User,
	serviceID uuid.UUID,
) (bool, error) {
	serviceRepository, err := servicesrepository.NewServiceRepository(srv.db)
	if err != nil {
		return false, fmt.Errorf("failed to create service repository: %w", err)
	}
	serviceEntity, err := serviceRepository.Get(ctx, serviceID)
	if err != nil {
		return false, fmt.Errorf("failed to find service: %w", err)
	}
	if serviceEntity.StudentID == user.ID {
		return true, nil
	}
	return srv.coursesService.CanManage(ctx, user, serviceEntity.CourseID)
}
```

services\servicesservice\servicesservice.go:

```
package servicesservice
import (...)
type ServicesService struct {
	db                 *mongo.Database
	kubernetesService KubernetesService
```

```
	quotaService     QuotaService
}
type QuotaService interface {
	AcquireServiceQuota(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID,
		serviceID uuid.UUID, quota quotasmodels.QuotaRequest) error
	ScaleServiceQuota(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID,
		serviceID uuid.UUID, newQuota quotasmodels.QuotaRequest) error
	GetServiceQuota(ctx context.Context, serviceID uuid.UUID) (*quotasmodels.QuotaRequest, error)
	GetServiceQuotas(ctx context.Context, serviceIDs []uuid.UUID) (map[uuid.UUID]*quotasmodels.QuotaRequest, error)
	ReleaseServiceQuota(ctx context.Context, courseID uuid.UUID, studentID uuid.UUID, serviceID uuid.UUID) error
	ReleaseServiceQuotas(ctx context.Context, courseID uuid.UUID, userID uuid.UUID, serviceIDs []uuid.UUID) error
}
func NewServicesService(
	db *mongo.Database,
	kubernetesService KubernetesService,
	quotaService QuotaService,
) (*ServicesService, error) {
	return &ServicesService{
		db:              db,
		kubernetesService: kubernetesService,
		quotaService:     quotaService,
	}, nil
}
func (srv *ServicesService) Get(ctx context.Context, serviceID uuid.UUID) (*servicemodels.ServiceState, error) {
	serviceRepository, err := servicesrepository.NewServiceRepository(srv.db)
	if err != nil {
		return nil, fmt.Errorf("failed to create service repository: %w", err)
	}
	serviceEntity, err := serviceRepository.Get(ctx, serviceID)
	if err != nil {
		if errors.Is(err, baserepository.ErrNoEntities) {
			return nil, baseerrors.EntityNotFound
		}
		return nil, fmt.Errorf("failed to find service: %w", err)
	}
	kubernetesState, err := srv.kubernetesService.GetServiceState(ctx, serviceID)
	if err != nil {
		log.Printf("failed to get current service state: %v", err)
	}
	quota, err := srv.quotaService.GetServiceQuota(ctx, serviceID)
	if err != nil {
		return nil, fmt.Errorf("failed to get service quota: %w", err)
	}
	serviceState := convertToServiceState(serviceEntity, quota, kubernetesState)
	return serviceState, nil
}
func (srv *ServicesService) Create(
	ctx context.Context,
	courseID uuid.UUID,
	studentID uuid.UUID,
	serviceRequest *servicemodels.ServiceRequest,
) (id string, err error) {
	serviceRepository, err := servicesrepository.NewServiceRepository(srv.db)
	if err != nil {
		return "", fmt.Errorf("failed to create service repository: %w", err)
	}
	serviceTemplateRepository, err := servicesrepository.NewServiceTemplateRepository(srv.db)
	if err != nil {
		return "", fmt.Errorf("failed to create service template repository: %w", err)
	}
	userEnvs, err := convertEnvsModelToEnvs(serviceRequest.UserEnvs)
	if err != nil {
		return "", fmt.Errorf("failed to create new service user env domain objects: %w", err)
	}
	volumes, err := convertVolumesModelToVolumes(serviceRequest.Volumes)
	if err != nil {
		return "", fmt.Errorf("failed to create new service volume domain objects: %w", err)
	}
	serviceTemplate, err := serviceTemplateRepository.Get(ctx, serviceRequest.TemplateID)
	if err != nil {
		return "", fmt.Errorf("failed to find service template %q: %w", serviceRequest.TemplateID.String(), err)
	}
```

```
		service, err := serviceTemplate.ConstructService(courseID, studentID, serviceRequest.Name, userEnvs, volumes)
		if err != nil {
			return "", fmt.Errorf("failed to construct new service domain object from template: %w", err)
		}
		quota := quotasmodels.QuotaRequest{
			CPU:     serviceRequest.Quota.CPU,
			RAM:     serviceRequest.Quota.RAM,
			Storage: service.RequestedStorageQuota(),
		}
		err = baseservice.WithTransaction(ctx, srv.db,
			baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
				err = srv.quotaService.AcquireServiceQuota(ctx, courseID, studentID, service.ID, quota)
				if err != nil {
					return nil, fmt.Errorf("failed to acquire service quota: %w", err)
				}
				err = serviceRepository.Insert(ctx, service)
				if err != nil {
					return nil, fmt.Errorf("failed to save service: %w", err)
				}
				return nil, nil
			}))
		if err != nil {
			return "", fmt.Errorf("failed to create service: %w", err)
		}
		return service.ID.String(), nil
}
func (srv *ServicesService) DeleteService(ctx context.Context, serviceID uuid.UUID) error {
		serviceRepository, err := servicesrepository.NewServiceRepository(srv.db)
		if err != nil {
			return fmt.Errorf("failed to create service repository: %w", err)
		}
		err = baseservice.WithTransaction(ctx, srv.db,
			baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
				serviceEntity, err := serviceRepository.Get(ctx, serviceID)
				if err != nil {
					if errors.Is(err, baserepository.ErrNoEntities) {
						return nil, baseerrors.EntityNotFound
					}
					return nil, fmt.Errorf("failed to find service: %w", err)
				}
				err = serviceEntity.SetDesiredState(servicesdomain.DesiredStateDeleted)
				if err != nil {
					return nil, fmt.Errorf("failed to mark service as deleted: %w", err)
				}
				err = srv.quotaService.ReleaseServiceQuota(ctx, serviceEntity.CourseID, serviceEntity.StudentID, serviceID)
				if err != nil {
					return nil, fmt.Errorf("failed to release service quota: %w", err)
				}
				err = serviceRepository.Save(ctx, serviceEntity)
				if err != nil {
					return nil, fmt.Errorf("failed to save service: %w", err)
				}
				return nil, nil
			}))
		if err != nil {
			return fmt.Errorf("failed to delete service: %w", err)
		}
		return nil
}
func (srv *ServicesService) DeleteServicesByEnrollment(
		ctx context.Context,
		courseID uuid.UUID,
		userID uuid.UUID,
) error {
		serviceRepository, err := servicesrepository.NewServiceRepository(srv.db)
		if err != nil {
			return fmt.Errorf("failed to create service repository: %w", err)
		}
		err = baseservice.WithTransaction(ctx, srv.db,
			baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
				serviceEntities, err := serviceRepository.GetByEnrollment(ctx, courseID, userID)
				if err != nil {
```

```
                                        if errors.Is(err, baserepository.ErrNoEntities) {
                                                return nil, baseerrors.EntityNotFound
                                        }
                                        return nil, fmt.Errorf("failed to find servicies: %w", err)
                                }
                                serviceIDs := make([]uuid.UUID, len(serviceEntities))
                                for i, serviceEntity := range serviceEntities {
                                        err = serviceEntity.SetDesiredState(servicesdomain.DesiredStateDeleted)
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to mark service as deleted: %w", err)
                                        }
                                        serviceIDs[i] = serviceEntity.ID
                                }
                                err = srv.quotaService.ReleaseServiceQuotas(ctx, courseID, userID, serviceIDs)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to release service quotas: %w", err)
                                }
                                for _, serviceEntity := range serviceEntities {
                                        err = serviceRepository.Save(ctx, serviceEntity)
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to save service: %w", err)
                                        }
                                }
                                return nil, nil
                        }))
                if err != nil {
                        return fmt.Errorf("failed to delete servicies: %w", err)
                }
                return nil
}
func (srv *ServicesService) ChangeService(
                ctx context.Context,
                serviceID uuid.UUID,
                request *servicemodels.ServicePatchRequest,
) (*servicemodels.ServiceState, error) {
                serviceRepository, err := servicesrepository.NewServiceRepository(srv.db)
                if err != nil {
                        return nil, fmt.Errorf("failed to create service repository: %w", err)
                }
                var serviceEntity *servicesdomain.Service
                err = baseservice.WithTransaction(ctx, srv.db,
                        baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
                                var err error
                                serviceEntity, err = serviceRepository.Get(ctx, serviceID)
                                if err != nil {
                                        if errors.Is(err, baserepository.ErrNoEntities) {
                                                return nil, baseerrors.EntityNotFound
                                        }
                                        return nil, fmt.Errorf("failed to find service: %w", err)
                                }
                                if request.Quota != nil {
                                        err = srv.quotaService.ScaleServiceQuota(ctx,
                                                serviceEntity.CourseID,
                                                serviceEntity.StudentID,
                                                serviceID,
                                                quotasmodels.QuotaRequest{
                                                        CPU:     request.Quota.CPU,
                                                        RAM:     request.Quota.RAM,
                                                        Storage: serviceEntity.RequestedStorageQuota(),
                                                })
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to scale service quota: %w", err)
                                        }
                                }
                                if request.State != "" {
                                        err = serviceEntity.SetDesiredState(request.State)
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to set service status: %w", err)
                                        }
                                }
                                if request.UserEnvs != nil {
                                        envs, err := convertEnvsModelToEnvs(request.UserEnvs)
```

```
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to convert envs: %w", err)
                                        }
                                        err = serviceEntity.SetUserEnvs(envs)
                                        if err != nil {
                                                return nil, fmt.Errorf("failed to set service user envs: %w", err)
                                        }
                                }
                                err = serviceRepository.Save(ctx, serviceEntity)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to save service: %w", err)
                                }
                                return nil, nil
                        }))
        if err != nil {
                return nil, fmt.Errorf("failed to update service: %w", err)
        }
        kubernetesState, err := srv.kubernetesService.GetServiceState(ctx, serviceID)
        if err != nil {
                log.Printf("failed to get current service state: %v", err)
        }
        quota, err := srv.quotaService.GetServiceQuota(ctx, serviceID)
        if err != nil {
                return nil, fmt.Errorf("failed to get service quota: %w", err)
        }
        serviceState := convertToServiceState(serviceEntity, quota, kubernetesState)
        return serviceState, nil
}
func (srv *ServicesService) GetByEnrollmentID(
        ctx context.Context,
        courseID uuid.UUID,
        studentID uuid.UUID,
) ([]*servicemodels.ServiceState, error) {
        serviceRepository, err := servicesrepository.NewServiceRepository(srv.db)
        if err != nil {
                return nil, fmt.Errorf("failed to create service repository: %w", err)
        }
        serviceEntities, err := serviceRepository.GetByEnrollment(ctx, courseID, studentID)
        if err != nil {
                if errors.Is(err, baserepository.ErrNoEntities) {
                        return nil, baseerrors.EntityNotFound
                }
                return nil, fmt.Errorf("failed to find services: %w", err)
        }
        serviceStates := make([]*servicemodels.ServiceState, len(serviceEntities))
        serviceIDs := make([]uuid.UUID, len(serviceEntities))
        for i := range serviceIDs {
                serviceIDs[i] = serviceEntities[i].ID
        }
        serviceQuotas, err := srv.quotaService.GetServiceQuotas(ctx, serviceIDs)
        if err != nil {
                return nil, fmt.Errorf("failed to get service quotas: %w", err)
        }
        kubernetesServiceStates, err := srv.kubernetesService.GetServiceStates(ctx, serviceIDs)
        if err != nil {
                return nil, fmt.Errorf("failed to get kubernetes service states: %w", err)
        }
        for i := range serviceEntities {
                serviceID := serviceEntities[i].ID
                serviceStates[i] = convertToServiceState(
                        serviceEntities[i],
                        serviceQuotas[serviceID],
                        kubernetesServiceStates[serviceID],
                )
        }
        return serviceStates, nil
}
```

services\servicesservice\servicetemplatesservice.go:

```
package servicesservice
```

```
import (...)
type ServiceTemplatesService struct {
	db *mongo.Database
}
func NewServiceTemplatesService(db *mongo.Database) (*ServiceTemplatesService, error) {
	return &ServiceTemplatesService{
		db: db,
	}, nil
}
func (srv *ServiceTemplatesService) GetAll(ctx context.Context) ([]*servicemodels.ServiceTemplate, error) {
	serviceTemplateRepository, err := servicesrepository.NewServiceTemplateRepository(srv.db)
	if err != nil {
		return nil, fmt.Errorf("failed to create service template repository: %w", err)
	}
	serviceTemplates, err := serviceTemplateRepository.GetAll(ctx)
	if err != nil {
		return nil, fmt.Errorf("failed to find service templates: %w", err)
	}
	serviceTemplateModels := make([]*servicemodels.ServiceTemplate, len(serviceTemplates))
	for i := range serviceTemplates {
		serviceTemplateModels[i] = convertServiceTemplateToServiceTemplateModel(serviceTemplates[i])
	}
	return serviceTemplateModels, nil
}
func (srv *ServiceTemplatesService) Get(
	ctx context.Context,
	serviceTemplateID uuid.UUID,
) (*servicemodels.ServiceTemplate, error) {
	serviceTemplateRepository, err := servicesrepository.NewServiceTemplateRepository(srv.db)
	if err != nil {
		return nil, fmt.Errorf("failed to create service template repository: %w", err)
	}
	serviceTemplate, err := serviceTemplateRepository.Get(ctx, serviceTemplateID)
	if err != nil {
		if errors.Is(err, baserepository.ErrNoEntities) {
			return nil, baseerrors.EntityNotFound
		}
		return nil, fmt.Errorf("failed to find service template: %w", err)
	}
	serviceTemplateModel := convertServiceTemplateToServiceTemplateModel(serviceTemplate)
	return serviceTemplateModel, nil
}
func (srv *ServiceTemplatesService) Create(
	ctx context.Context,
	serviceTemplateRequest *servicemodels.ServiceTemplate,
) (id string, err error) {
	serviceTemplateRepository, err := servicesrepository.NewServiceTemplateRepository(srv.db)
	if err != nil {
		return "", fmt.Errorf("failed to create service repository: %w", err)
	}
	ports, err := convertPortsModelToPorts(serviceTemplateRequest.Ports)
	if err != nil {
		return "", fmt.Errorf("failed to create new service template port domain objects: %w", err)
	}
	fixedEnvs, err := convertEnvsModelToEnvs(serviceTemplateRequest.FixedEnvs)
	if err != nil {
		return "", fmt.Errorf("failed to create new service template fixed env domain objects: %w", err)
	}
	userEnvs, err := convertEnvsModelToEnvs(serviceTemplateRequest.UserEnvs)
	if err != nil {
		return "", fmt.Errorf("failed to create new service template user env domain objects: %w", err)
	}
	volumes, err := convertVolumesModelToVolumes(serviceTemplateRequest.Volumes)
	if err != nil {
		return "", fmt.Errorf("failed to create new service template volumes domain objects: %w", err)
	}
	serviceTemplate, err := servicesdomain.CreateTemplate(
		serviceTemplateRequest.Name,
		serviceTemplateRequest.DockerImage,
		ports,
		fixedEnvs,
		userEnvs,
```

```
                volumes,
        )
        if err != nil {
                return "", fmt.Errorf("failed to create new service template domain object: %w", err)
        }
        err = baseservice.WithTransaction(ctx, srv.db,
                baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
                        err = serviceTemplateRepository.Insert(ctx, serviceTemplate)
                        if err != nil {
                                return nil, fmt.Errorf("failed to save service template: %w", err)
                        }
                        return nil, nil
                }))
        if err != nil {
                return "", fmt.Errorf("failed to create service template: %w", err)
        }
        return serviceTemplate.ID.String(), nil
}
```

users\usersdomain\user.go:

```
package usersdomain
import (...)
type User struct {
        ID        uuid.UUID
        ExternalID string
        Email      string
        Name       string
        Role       string
}
func (u *User) IsAdmin() bool {
        return u.Role == "Admin"
}
func (u *User) IsTeacher() bool {
        return u.Role == "Teacher"
}
func (u *User) SetName(name string) error {
        u.Name = name
        return nil
}
func (u *User) SetRole(role string) error {
        if role != "Admin" && role != "Teacher" && role != "Student" {
                return baseerrors.NewBusinessRule("invalidRole", fmt.Sprintf("invalid role %q", role))
        }
        u.Role = role
        return nil
}
func CreateUser(
        externalID string,
        email string,
        name string,
) (*User, error) {
        return &User{
                ID:        uuid.Must(uuid.NewV4()),
                ExternalID: externalID,
                Email:      email,
                Name:       name,
                Role:       "Student",
        }, nil
}
```

users\usershttpapi\getuser.go:

```
package usershttpapi
import (...)
func (api *API) getUserRoute(c echo.Context) error {
        user := basehttpapi.GetUser(c).(*usersservice.User)
        userID, err := basehttpapi.HandleMeUserID(user.ID, c.Param("userID"), func() (bool, error) {
                return true, nil
        })
        if err != nil {
```

```
                return err
        }
        fetchedUser, err := api.usersService.GetUser(context.Background(), userID)
        if err != nil {
                log.Errorf("failed to get user: %v", err)
                return echo.NewHTTPError(http.StatusInternalServerError, "failed to get user")
        }
        return c.JSON(http.StatusOK, fetchedUser)
}
```

users\usershttpapi\getusers.go:

```
package usershttpapi
import (...)
func (api *API) getUsersRoute(c echo.Context) error {
        user := basehttpapi.GetUser(c).(*usersservice.User)
        err := basehttpapi.CheckAccess(func() (bool, error) {
                return user.IsAdmin(), nil
        })
        if err != nil {
                return err
        }
        searchQuery := c.QueryParam("query")
        fetchedUsers, err := api.usersService.GetUsers(context.Background(), searchQuery)
        if err != nil {
                log.Errorf("failed to get users: %v", err)
                return echo.NewHTTPError(http.StatusInternalServerError, "failed to get users")
        }
        return c.JSON(http.StatusOK, fetchedUsers)
}
```

users\usershttpapi\patchuser.go:

```
package usershttpapi
import (...)
type patchUserRequest struct {
        Role string `json:"role"`
}
func (api *API) patchUserRoute(c echo.Context) error {
        user := basehttpapi.GetUser(c).(*usersservice.User)
        var request patchUserRequest
        err := c.Bind(&request)
        if err != nil {
                return echo.NewHTTPError(http.StatusBadRequest, fmt.Sprintf("failed to bind request: %v", err))
        }
        userID, err := basehttpapi.HandleMeUserID(user.ID, c.Param("userID"), func() (bool, error) {
                return user.IsAdmin(), nil
        })
        if err != nil {
                return err
        }
        err = api.usersService.ChangeRole(context.Background(), userID, request.Role)
        if err != nil {
                log.Errorf("failed to change user role: %v", err)
                return echo.NewHTTPError(http.StatusInternalServerError, "failed to change user role")
        }
        return c.NoContent(http.StatusOK)
}
```

users\usershttpapi\router.go:

```
package usershttpapi
import (...)
func (api *API) RegisterRoutes(echo *echo.Echo) {
        echo.GET("/users", api.getUsersRoute)
        echo.GET("/users/:userID", api.getUserRoute)
        echo.PATCH("/users/:userID", api.patchUserRoute)
}
```

users\usershttpapi\usershttpapi.go:

```
package usershttpapi
import (...)
type API struct {
	usersService *usersservice.UsersService
}
func NewAPI(
	usersService *usersservice.UsersService,
) (*API, error) {
	return &API{
		usersService: usersService,
	}, nil
}
```

users\usersservice\usersservice.go:

```
package usersservice
import (...)
type QuotaService interface {
	AcquireTeacherQuota(ctx context.Context, teacherID uuid.UUID, quota quotasmodels.QuotaRequest) error
	ReleaseTeacherQuota(ctx context.Context, teacherID uuid.UUID) error
}
type UsersService struct {
	db          *mongo.Database
	quotaService QuotaService
	config      Config
}
type Config struct {
	IntrospectURL string
}
func NewUsersService(db *mongo.Database, quotaService QuotaService, config Config) (*UsersService, error) {
	return &UsersService{
		db:          db,
		quotaService: quotaService,
		config:      config,
	}, nil
}
type User struct {
	ID    uuid.UUID `json:"id"`
	Email string    `json:"email"`
	Name  string    `json:"name"`
	Role  string    `json:"role"`
}
func (u *User) IsTeacher() bool {
	return u.Role == "Teacher"
}
func (u *User) IsAdmin() bool {
	return u.Role == "Admin"
}
func (s *UsersService) Auth(ctx context.Context, token string) (*User, error) {
	userInfo, err := oauth2checker.CheckToken(ctx, s.config.IntrospectURL, token)
	if err != nil {
		return nil, fmt.Errorf("failed to check token: %w", err)
	}
	if !userInfo.IsAuthenticated {
		return nil, baseerrors.NewBusinessRule("unauthorized", "user was not authenticated by external oauth provider")
	}
	var userEntity *usersdomain.User
	usersRepository, err := usersrepository.NewUserRepository(s.db)
	if err != nil {
		return nil, fmt.Errorf("failed to create users repository: %w", err)
	}
	err = baseservice.WithTransaction(ctx, s.db,
		baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
			needCreate := false
			userEntity, err = usersRepository.GetByExternalID(ctx, userInfo.Subject)
			if err != nil {
				if !errors.Is(err, baserepository.ErrNoEntities) {
					return nil, fmt.Errorf("failed to fetch user by user id: %w", err)
				}
				needCreate = true
			}
```

```
                                if !needCreate {
                                        return nil, nil
                                }
                                userEntity, err = usersdomain.CreateUser(userInfo.Subject, userInfo.Email, userInfo.Name)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to create user entity: %w", err)
                                }
                                err = usersRepository.Insert(ctx, userEntity)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to insert user entity: %w", err)
                                }
                                return nil, nil
                        }))
        if err != nil {
                return nil, fmt.Errorf("failed to run transaction: %w", err)
        }
        return &User{
                ID:    userEntity.ID,
                Email: userEntity.Email,
                Name:  userEntity.Name,
                Role:  userEntity.Role,
        }, nil
}
func (s *UsersService) GetUser(ctx context.Context, userID uuid.UUID) (*User, error) {
        usersRepository, err := usersrepository.NewUserRepository(s.db)
        if err != nil {
                return nil, fmt.Errorf("failed to create users repository: %w", err)
        }
        user, err := usersRepository.Get(ctx, userID)
        if err != nil {
                if errors.Is(err, baserepository.ErrNoEntities) {
                        return nil, baseerrors.EntityNotFound
                }
                return nil, fmt.Errorf("failed to get user: %w", err)
        }
        return &User{
                ID:    user.ID,
                Email: user.Email,
                Name:  user.Name,
                Role:  user.Role,
        }, nil
}
func (s *UsersService) GetUsersByID(ctx context.Context, userIDs []uuid.UUID) (map[uuid.UUID]*User, error) {
        usersRepository, err := usersrepository.NewUserRepository(s.db)
        if err != nil {
                return nil, fmt.Errorf("failed to create users repository: %w", err)
        }
        users, err := usersRepository.GetMany(ctx, userIDs)
        if err != nil {
                return nil, fmt.Errorf("failed to get users: %w", err)
        }
        userModels := make(map[uuid.UUID]*User, len(users))
        for i := range users {
                userModels[users[i].ID] = &User{
                        ID:    users[i].ID,
                        Email: users[i].Email,
                        Name:  users[i].Name,
                        Role:  users[i].Role,
                }
        }
        return userModels, nil
}
func (s *UsersService) ChangeRole(ctx context.Context, userID uuid.UUID, role string) error {
        usersRepository, err := usersrepository.NewUserRepository(s.db)
        if err != nil {
                return fmt.Errorf("failed to create users repository: %w", err)
        }
        err = baseservice.WithTransaction(ctx, s.db,
                baseservice.WithConcurrentRetry(func(ctx mongo.SessionContext) (interface{}, error) {
                        user, err := usersRepository.Get(ctx, userID)
                        if err != nil {
                                return nil, fmt.Errorf("failed to find user: %w", err)
```

```
                        }
                        hadQuota := user.IsTeacher() || user.IsAdmin()
                        err = user.SetRole(role)
                        if err != nil {
                                return nil, fmt.Errorf("failed to change user role: %w", err)
                        }
                        hasQuota := user.IsTeacher() || user.IsAdmin()
                        if !hadQuota && hasQuota {
                                err = s.quotaService.AcquireTeacherQuota(ctx, userID, quotasmodels.QuotaRequest{
                                        CPU:     0,
                                        RAM:     0,
                                        Storage: 0,
                                })
                                if err != nil {
                                        return nil, fmt.Errorf("failed to acquire empty quota for user: %w", err)
                                }
                        } else if hadQuota && !hasQuota {
                                err = s.quotaService.ReleaseTeacherQuota(ctx, userID)
                                if err != nil {
                                        return nil, fmt.Errorf("failed to release quota from user: %w", err)
                                }
                        }
                        err = usersRepository.Save(ctx, user)
                        if err != nil {
                                return nil, fmt.Errorf("failed to save user: %w", err)
                        }
                        return nil, nil
                }))
        if err != nil {
                return fmt.Errorf("failed to complete transaction: %w", err)
        }
        return nil
}
func (s *UsersService) GetUsers(ctx context.Context, nameQuery string) ([]*User, error) {
        usersRepository, err := usersrepository.NewUserRepository(s.db)
        if err != nil {
                return nil, fmt.Errorf("failed to create users repository: %w", err)
        }
        users, err := usersRepository.GetByName(ctx, nameQuery)
        if err != nil {
                return nil, fmt.Errorf("failed to get users: %w", err)
        }
        userModels := make([]*User, len(users))
        for i := range userModels {
                userModels[i] = &User{
                        ID:    users[i].ID,
                        Email: users[i].Email,
                        Name:  users[i].Name,
                        Role:  users[i].Role,
                }
        }
        return userModels, nil
}
```

ПРИЛОЖЕНИЕ 2

РУКОВОДСТВО ОПЕРАТОРА

1. Назначение программы

Данная программа предназначена для создания и удалённого запуска учебных сервисов преподавателями и студентами при изучении IT-направления высшего образования. Преподаватель создаёт учебный курс и подтверждает заявки студентов на этот курс. Студенты подают заявки на участие в учебных курсах и в случае одобрения заявки получают квоту серверных ресурсов, которую могут использовать для создания и удалённого запуска учебных сервисов при помощи шаблонов сервисов.

2. Условия выполнения программы

Для работы программы (сайта) пользователю необходимо иметь современный браузер (Google Chrome версии 80 или выше, Safari версии 13 или выше, Firefox версии 74 или выше, или аналоги).

3. Выполнение программы

3.1. Аутентификация пользователя

При открытии сайта необходимо нажать на кнопку «Вход» в меню сверху экрана (Рис. П2.1) для начала аутентификации.

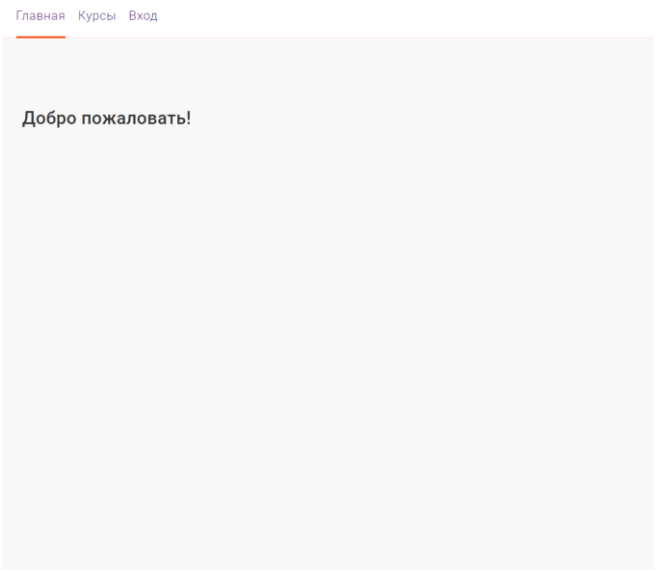


Рис. П2.1. Главная страница сайта

После этого будет отображена страница входа (Рис. П2.2), на которой нужно ввести логин и пароль пользователя в соответствующие поля и нажать кнопку «Вход».

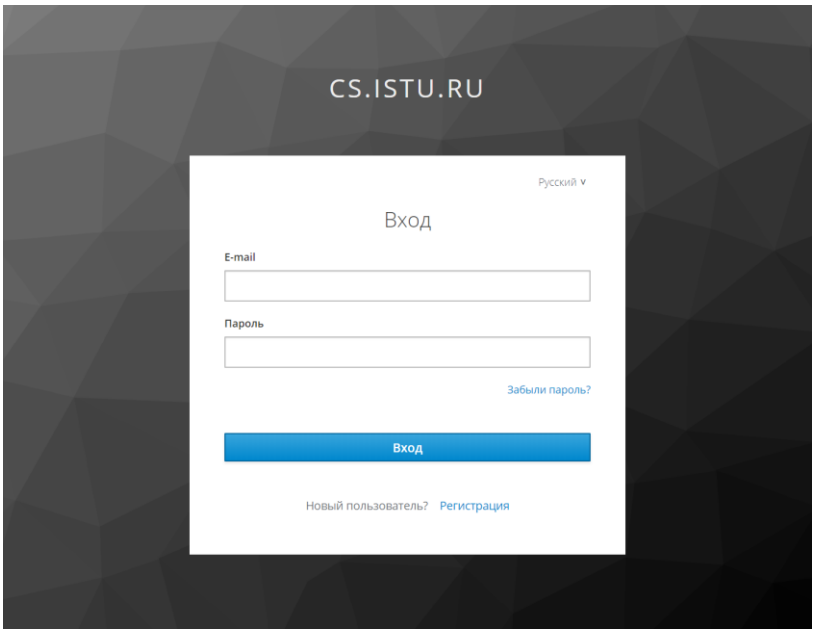


Рис. П2.2. Страница входа

Если у пользователя ещё нет учетной записи, нужно нажать кнопку «Регистрация», ввести имя, фамилию, email и пароль в соответствующие поля (Рис. П2.3) и нажать кнопку «Регистрация», после чего произвести аутентификацию.

Рис. П2.3. Страница регистрации

3.2. Подача заявки на курс

Для подачи заявки на участие в курсе необходимо нажать кнопку «Курсы» в меню сверху экрана. После этого будет отображена страница со списком курсов (Рис. П2.4).

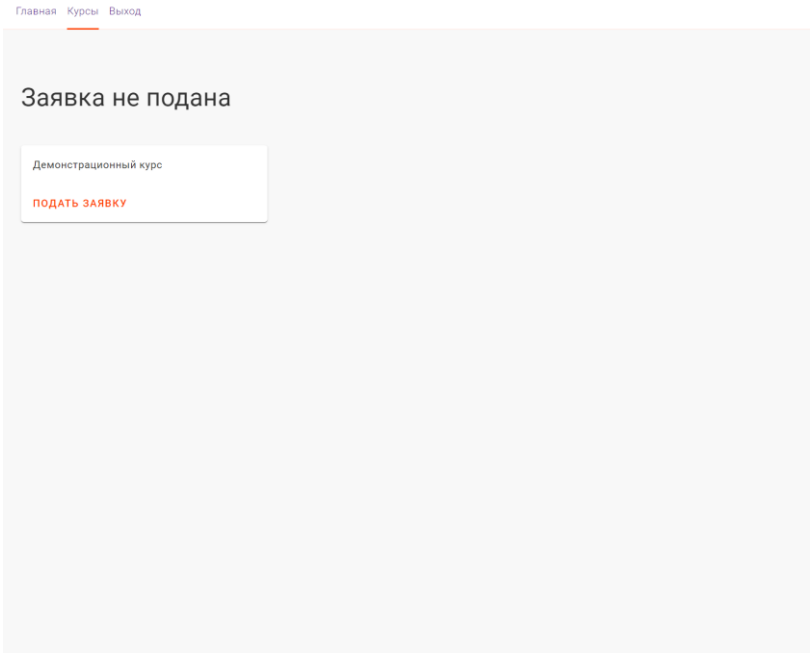


Рис. П2.4. Список курсов

Для подачи заявки необходимо нажать на кнопку «Подать заявку» рядом с нужным курсом. После этого статус участия в курсе сменится (Рис. П2.5) и появится кнопка для отмены заявки.

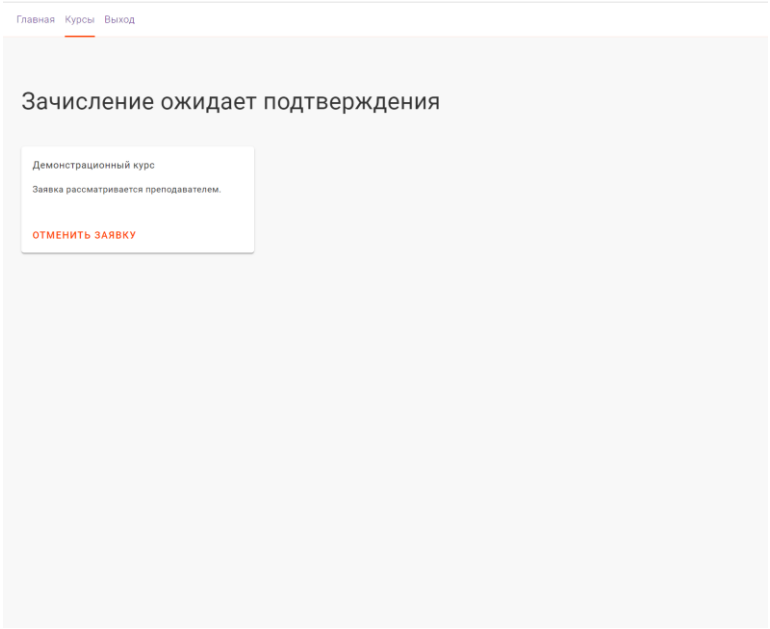


Рис. П2.5. Список курсов после подачи заявки

3.3. Создание учебного сервиса

Для создания учебного сервиса необходимо нажать на кнопку «ваши сервисы» рядом с нужным курсом на странице списка курсов (Рис. П2.6).

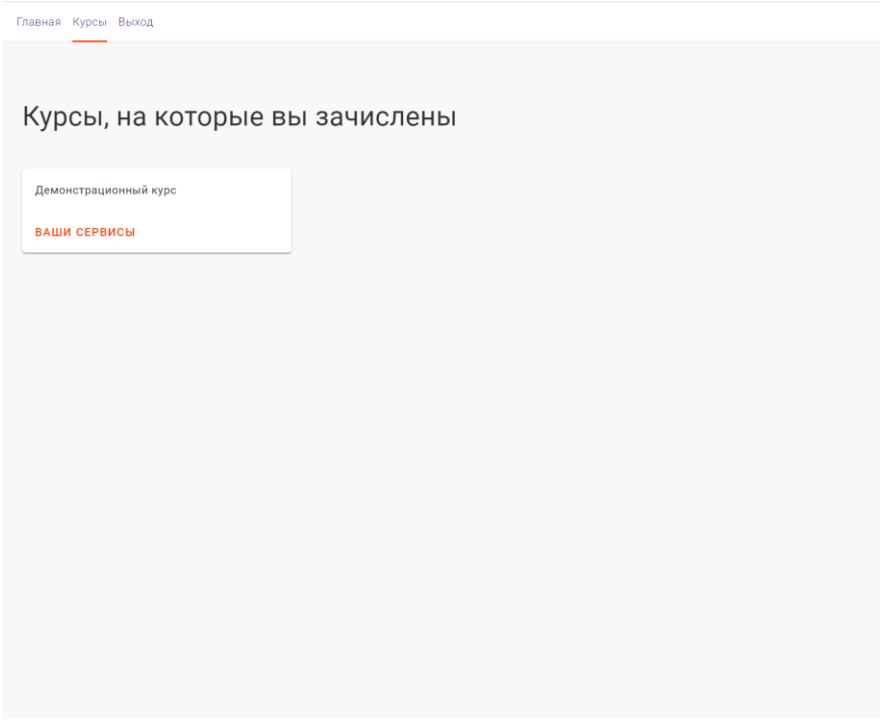


Рис. П2.6. Список курсов после подтверждения заявки

После этого будет отображена страница со списком сервисов. Для добавления нового сервиса необходимо нажать на кнопку «+» в правом нижнем углу (Рис. П2.7).

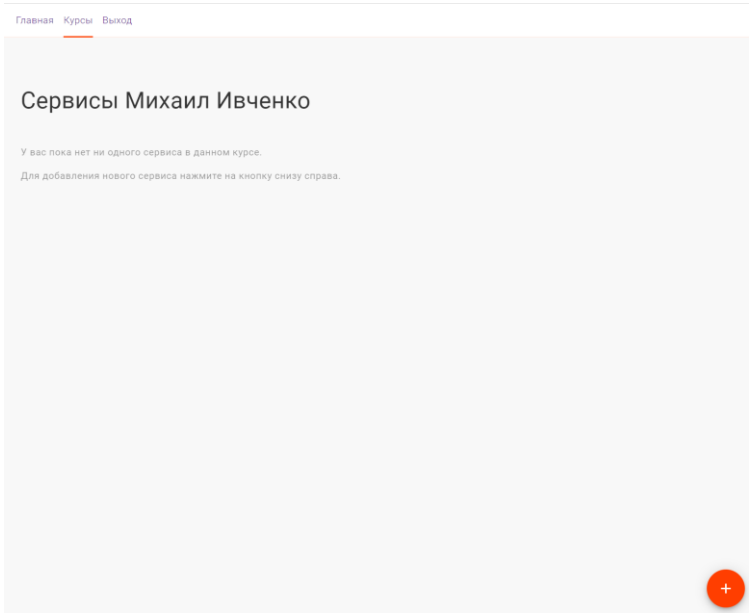


Рис. П2.7. Список сервисов

После этого будет отображена страница добавления нового сервиса (Рис. П2.8). Необходимо указать заполнить поля формы: указать название сервиса, шаблон, который используется для создания, параметры этого шаблона, квоту вычислительных ресурсов и объём хранилища для данных.

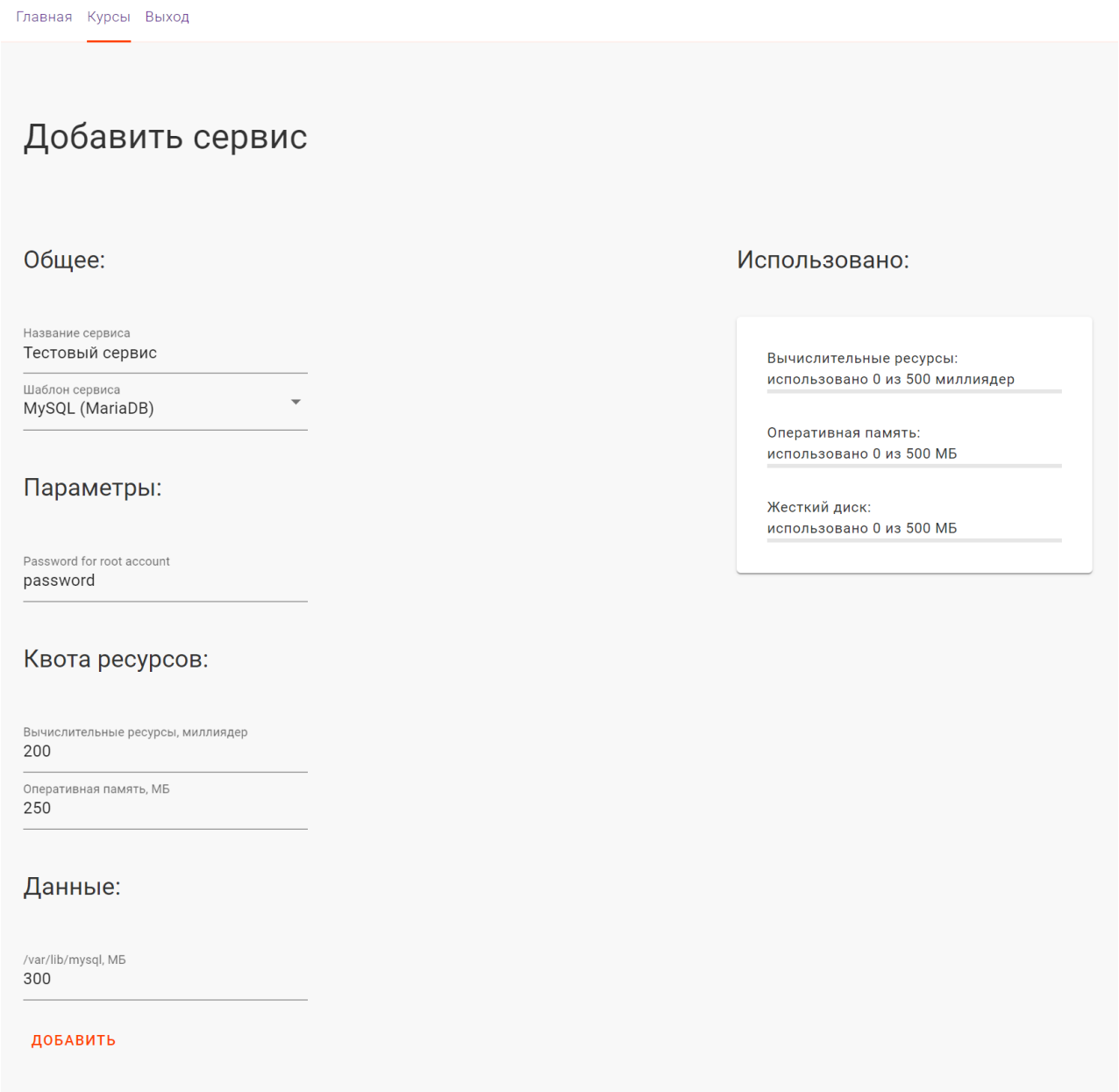


Рис. П2.8. Страница добавления нового сервиса

После этого новый сервис будет создан и будет отображена страница с информацией о нём (Рис. П2.9). Для изменения параметров сервиса необходимо ввести значения в соответствующие поля и нажать кнопку «сохранить изменения». Для остановки, запуска или удаления сервиса

необходимо нажать кнопки «остановить», «запустить» или «удалить» соответственно.

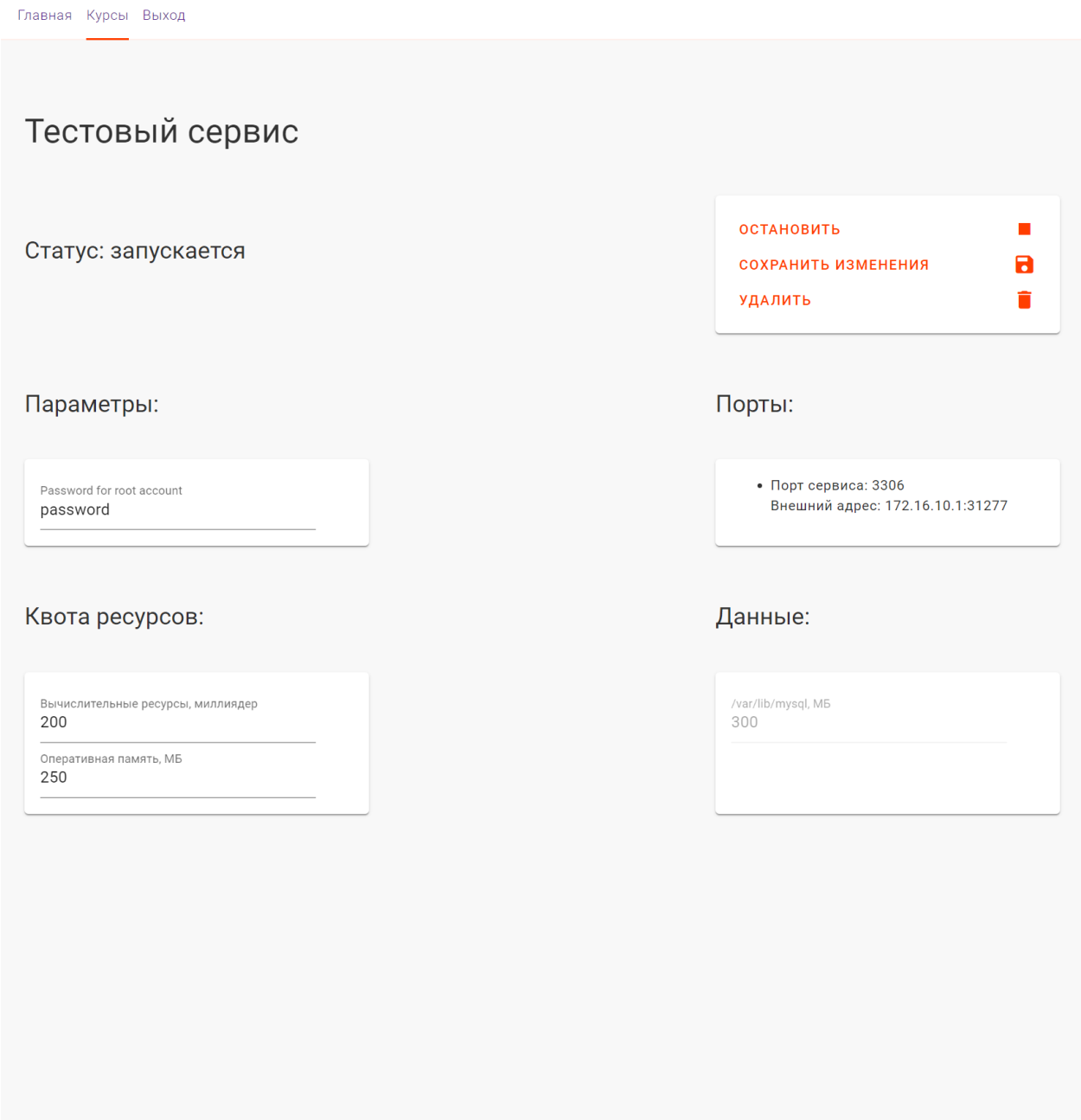


Рис. П2.9. Страница с информацией о созданном сервисе

3.4. Подтверждение заявки на участие в курсе

Для подтверждения заявки на участие в курсе преподавателю необходимо зайти на страницу со списком курсов и нажать на кнопку «студенты» (Рис. П2.10). После этого откроется страница со списком студентов, зачисленных на курс и подавших заявки (Рис. П2.11). Для подтверждения зачисления студента на курс необходимо нажать на кнопку

«подтвердить» около нужного студента. В случае, если квоты, имеющейся у преподавателя, недостаточно для зачисления студента, будет показано сообщение внизу экрана (Рис. П2.12).

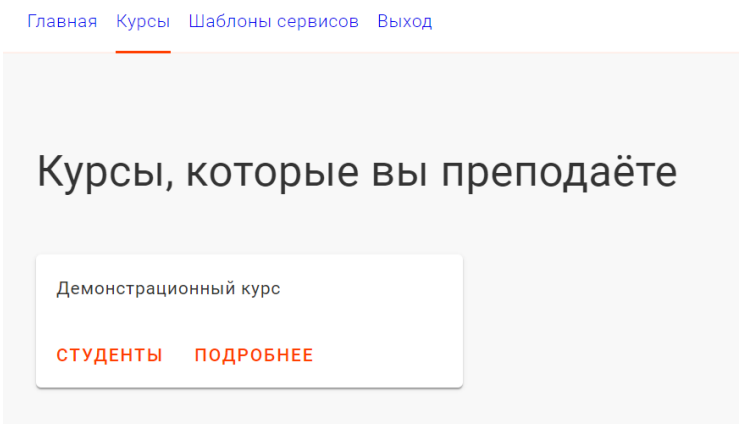


Рис. П2.10. Список курсов преподавателя

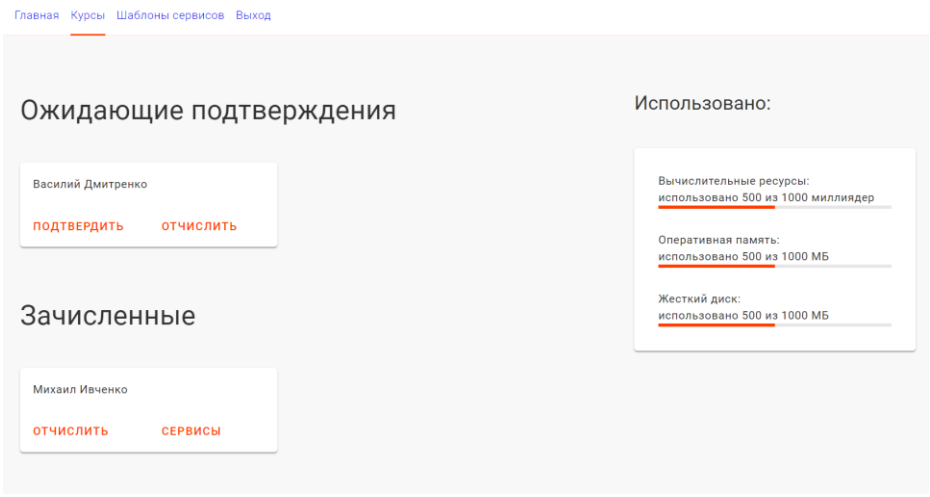


Рис. П2.11. Список студентов, зачисленных на курс и подавших заявки

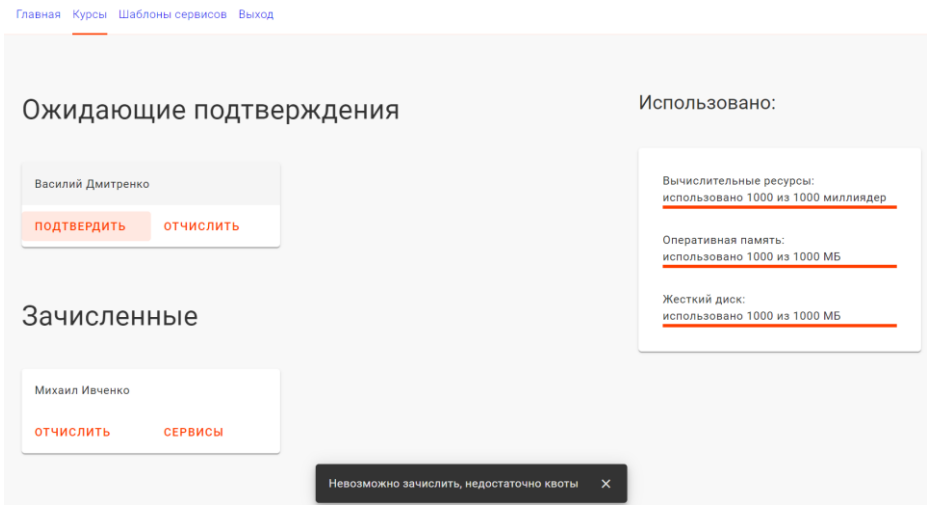


Рис. П2.12. Сообщение о том, что квоты преподавателя недостаточно

3.5. Создание курса

Для создания нового курса преподавателю необходимо нажать на кнопку «+» на странице со списком курсов. После этого будет отображена страница добавления курса (Рис. П2.13). Необходимо заполнить название курса, дату его окончания, квоту ресурсов для каждого студента в соответствующие поля и нажать на кнопку «добавить».

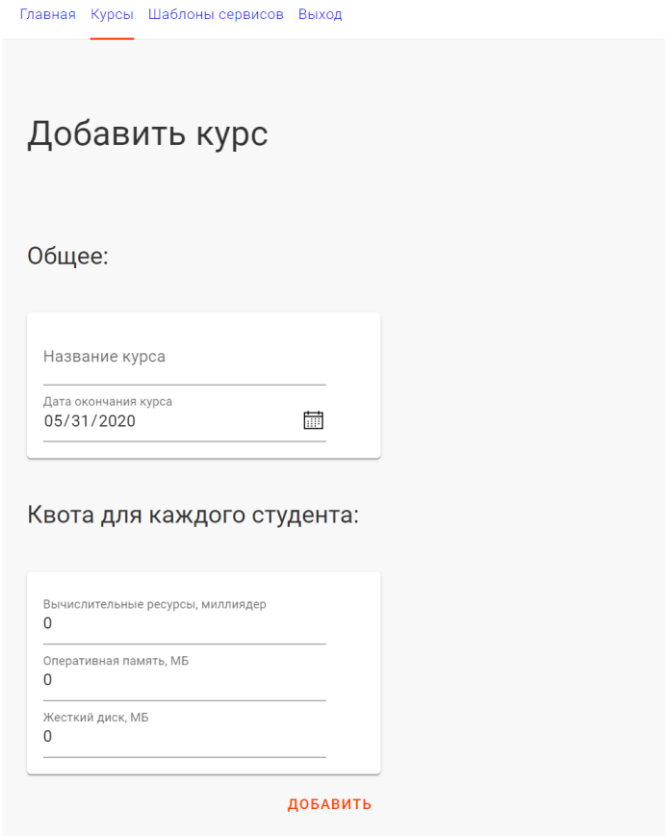


Рис. П2.13. Страница создания курса