Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Модели данных и системы управления базами данных

*К защите допустить*:

И.О. Заведующий кафедрой информатики

\_\_\_\_\_\_\_\_\_\_\_\_ С.И. Сиротко

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

**МОБИЛЬНОЕ ПРИЛОЖЕНИЕ РАСПИСАНИЯ ЗАНЯТИЙ БГУИР**

БГУИР КП 1-40 04 01 010

Студент А.П. Кирзнер

Руководитель А.В. Давыдчик

Минск 2024

**СОДЕРЖАНИЕ**

[Введение 4](#_Toc183895377)

[1 Архитектура программного обеспечения 5](#_Toc183895378)

   [1.1 Архитектура вычислительного средства 5](#_Toc183895379)

[1.2 Архитектура *ARM* 7](#_Toc183895380)

[1.3 История, версии и достоинства 10](#_Toc183895381)

[2 Платформа программного обеспечения 15](#_Toc183895382)

[2.1 Основные понятия 15](#_Toc183895383)

[2.2 Обоснование выбора платформы 16](#_Toc183895384)

[2.3 Анализ операционной системы для написания программы 17](#_Toc183895385)

[3 Теоретическое обоснование разработки программного продукта 20](#_Toc183895386)

[3.1 Обоснование необходимости разработки 20](#_Toc183895387)

[3.2 Технологии программирования, используемые для решения   
         поставленных задач 21](#_Toc183895388)

[4 Проектирование функциональных возможностей программы 28](#_Toc183895389)

[4.1 Функциональные требования 28](#_Toc183895390)

[4.2 Интерфейс программы 29](#_Toc183895391)

[5 Проектирование разрабатываемой базы данных              программного обеспечения 39](#_Toc183895392)

[Заключение 40](#_Toc183895393)

[Список литературных источников 41](#_Toc183895394)

[Приложение А (обязательное) Листинг программного кода 42](#_Toc183895395)

[Приложение Б (обязательное) Конечная схема базы данных 43](#_Toc183895396)

[Приложение В (обязательное) Ведомость документов 44](#_Toc183895397)

# Введение

# 1 Архитектура программного обеспечения

## 1.1 Архитектура вычислительного средства

Структуру вычислительного средства необходимо отличать от архитектуры вычислительного средства. Структура вычислительного средства определяет его текущий состав на определенном уровне детализации и описывает связи внутри средства. Архитектура же определяет основные правила взаимодействия составных элементов вычислительного средства, описание которых выполняется в той мере, в какой необходимо для формирования правил взаимодействия. Она устанавливает не все связи, а только наиболее необходимые, которые должны быть известны для более грамотного использования применяемого средства [1].

Архитектура – это базовая организация системы, воплощённая в её компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы.

Понятие архитектуры охватывает общие понятия организации системы, включающие такие высокоуровневые аспекты разработки компьютера, как система памяти, структура системной шины, организация ввода/вывода и т.п. Архитектура определяет принципы действия, информационные связи и взаимное соединение основных логических узлов компьютера: процессора, оперативного запоминающего устройства (ЗУ), внешних ЗУ и периферийных устройств. Общность архитектуры разных компьютеров обеспечивает их совместимость с точки зрения пользователя.

Архитектура вычислительной системы – совокупность характеристик и параметров, определяющих функционально-логичную и структурно-организованную систему и затрагивающих в основном уровень параллельно работающих вычислителей [2].

Следует отметить, что многие из существующих классификационных схем, во-первых, довольно условны и не являются общепринятыми, а во-вторых, с течением времени претерпевают определенные изменения, связанные с бурным развитием информационных технологий и невозможностью точно предсказать направления будущего развития компьютерной техники. Наиболее распространенные в настоящее время классификационные схемы:

– по принципу действия;

– по поколениям;

– по областям применения;

– по архитектуре набора команд;

– по разрядности машинного слова;

– по типу управления;

– по степени и способу организации параллелизма.

На рисунке 1.1 изображена схематично классификация архитектур по архитектуре набора команд.

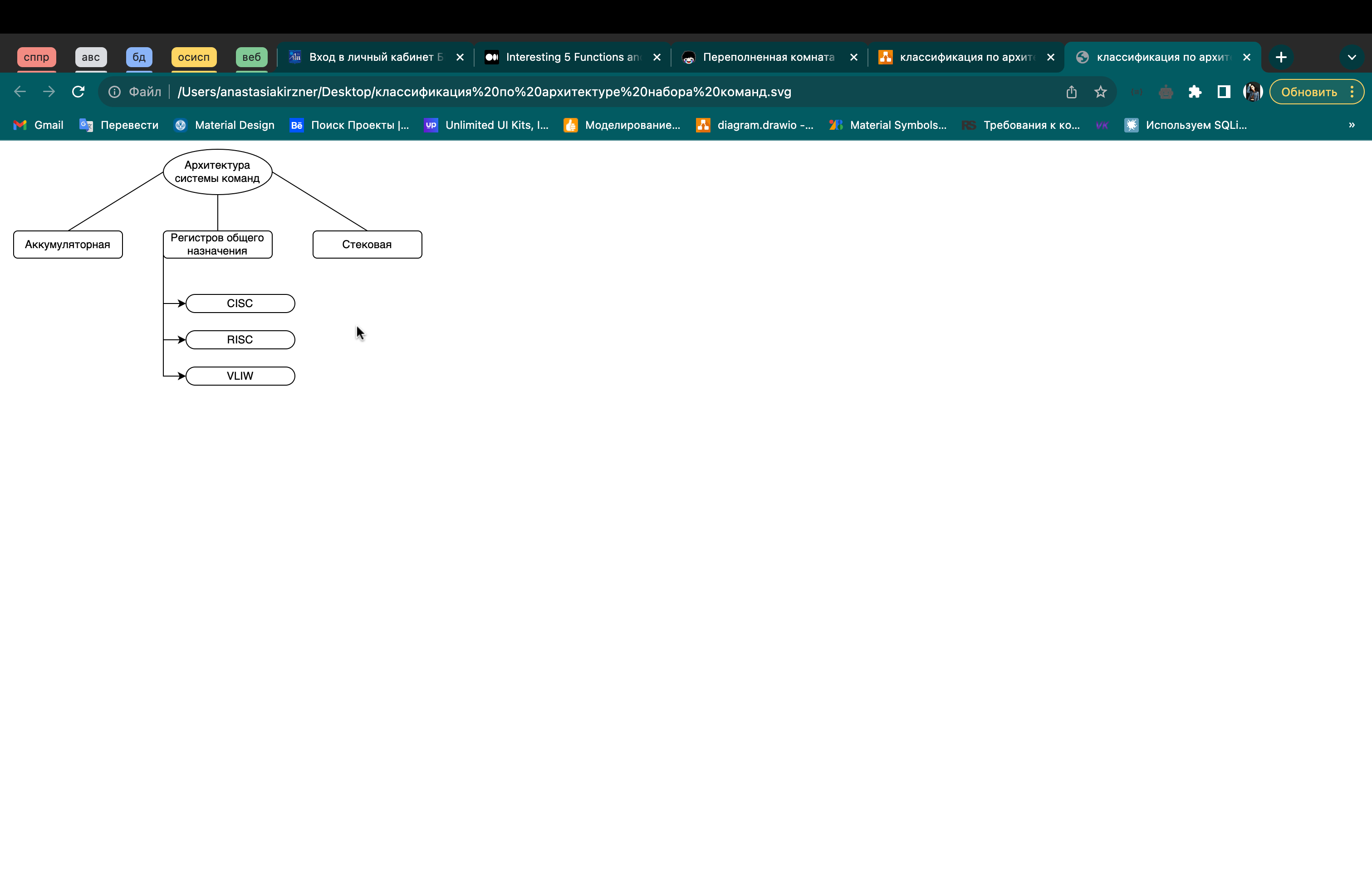


Рисунок 1.1 – Классификация по архитектуре набора команд

В рамках данного курсового проекта необходимо рассмотреть архитектуру регистров общего назначения, а именно архитектуру *RISC*.

*Restricted* (*Reduced*) *Instruction Set Computer* (*RISC*) – компьютер с сокращённым набором команд – архитектура процессора, в которой быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения – короче. В процессорах с *RISC*-архитектурой используется ограниченный набор быстрых команд. Каждая команда *RISC*-процессора должна выполняться за один машинный такт. Это облегчает повышение тактовой частоты и делает более эффективной суперскалярность (распараллеливание инструкций между несколькими исполнительными блоками). В таких микропроцессорах содержится меньшее количество транзисторов, что снижает их стоимость и энергопотребление. При этом, как правило, повышается их производительность. Архитектура *RISC* является основой современных высокопроизводительных электронных вычислительных машин. Она характеризуется наличием команд фиксированной длины, большого количества регистров, операций типа регистр-регистр, а также отсутствием косвенной адресации.

Для технологии *RISC* характерна сравнительно простая структура устройства управления. Площадь, выделяемая на кристалле микросхемы для его реализации, существенно меньше. Как следствие, появляется возможность разместить на кристалле большое число регистров центрального процессор (ЦП). Кроме того, остается больше места для других узлов ЦП и для дополнительных устройств: кэш-памяти, блока арифметики с плавающей запятой, части основной памяти, блока управления памятью, портов ввода/вывода.

Недостатки *RISC* прямо связаны с некоторыми преимуществами этой архитектуры. Принципиальный недостаток – сокращённое число команд: на выполнение ряда функций приходится тратить несколько команд вместо одной в *CISC*. Это удлиняет код программы, увеличивает загрузку памяти и трафик команд между памятью и ЦП. Исследования показали, что *RISC*-программа в среднем на 30% длиннее *CISC*-программы, реализующей те же функции [3].

## 1.2 Архитектура *ARM*

*ARM* архитектура объединяет в себе семейство как 32, так и 64-разрядных микропроцессорных ядер, разработанных и лицензируемых компанией *ARM* *Limited*.

Компания *ARM* *Limited* занимается сугубо разработкой ядер и инструментария для них (средства отладки, компиляторы и т.д.), но никак не производством самих процессоров. Компания *ARM* *Limited* продает лицензии на производство *ARM* процессоров сторонним фирмам. Многие известные компании уже получили лицензию на производство *ARM* процессоров, например, *AMD*, *Atmel*, *Altera*, *Cirrus* *Logic*, *Intel*, *Marvell*, *NXP*, *Samsung*, *LG*, *MediaTek*, *Qualcomm*, *Sony* *Ericsson*, *Texas* *Instruments*, *nVidia*, *Freescale* и многие другие.

*ARM* процессор – мобильный процессор для смартфонов и планшетов. Производительность *CPU* и *GPU* в различных *SoC* (*System*-*оn*-*Chip*) может значительно отличаться. *ARM* – это название архитектуры и одновременно название компании, ведущей ее разработку.

Аббревиатура *ARM* расшифровывается как *Advanced* *RISC* *Machine*, что можно перевести как: усовершенствованная *RISC*-машина.

Некоторые компании, получившие лицензию на выпуск *ARM* процессоров, создают собственные варианты ядер на базе *ARM* архитектуры. Например, *DEC* *StrongARM*, *Freescale i*.*MX*, *Intel XScale*, *NVIDIA Tegra*, *ST-Ericsson Nomadik*, *Qualcomm Snapdragon*, *Texas Instruments* *OMAP*, *Samsung Hummingbird*, *LG H*13, *Apple A*4/*A*5/*A*6 и *HiSilicon K*3.

На базе *ARM* процессоров работает фактически любая электроника: КПК, мобильные телефоны и смартфоны, цифровые плееры, портативные игровые консоли, калькуляторы, внешние жесткие диски и маршрутизаторы. Все они содержат в себе *ARM*-ядро, поэтому можно повторить, что *ARM* – это мобильные процессоры для смартфонов и планшетов.

*ARM* процессор представляет из себя *SoC*, или "систему на чипе". *SoC* система, или "система на чипе", может содержать в одном кристалле, помимо самого *CPU*, и остальные части полноценного компьютера. Это и контроллер памяти, и контроллер портов ввода-вывода, и графическое ядро, и система геопозиционирования (*GPS*). В нем может находится и 3*G* модуль, а также многое другое.

Если рассматривать отдельное семейство *ARM* процессоров, например, *Cortex*-*A*9, нельзя сказать, что все процессоры одного семейства имеют одинаковую производительность или все снабжены *GPS* модулем. Все эти параметры сильно зависят от производителя чипа и того, что и как он решил реализовать в своем продукте.

Сама по себе *RISC* (*Reduced Instruction Set Computer*) архитектура подразумевает под собой уменьшенный набор команд. Что соответственно ведет к очень умеренному энергопотреблению. Ведь внутри любого *ARM* чипа находится гораздо меньше транзисторов, чем у его аналога из *х*86 линейки. Кроме того, в *SoC*-системе все периферийные устройства находится внутри одной микросхемы, что позволяет *ARM* процессору быть еще более экономным в плане энергопотребления. *ARM* архитектура изначально была предназначена для вычисления только целочисленных операций, в отличии от *х*86, которые умеют работать с вычислениями с плавающей запятой или *FPU*. Нельзя однозначно сравнивать эти две архитектуры. В чем-то преимущество будет за *ARM*, а в чем-то и наоборот.

*ARM* процессор не знает того количества команд, которые знает *х*86 процессор. А те, что знает, выглядят гораздо короче. В этом его как плюсы, так и минусы. Как бы там ни было, *ARM* процессоры начинают медленно, но уверенно догонять, а кое в чем и перегонять обычные *х*86. Многие открыто заявляют о том, что в скором времени *ARM* процессоры заменят *х*86 платформу в сегменте домашних ПК.

Несмотря на то, что аббревиатура *ARM* расшифровывается как *Advanced RISC Machine*, с самого начала это была не совсем типичная *RISC*-архитектура. С одной стороны, она имела не очень много регистров общего назначения (сейчас с формальной точки зрения у большинства разновидностей архитектуры их 31 штука, однако программисту доступны лишь 16 из них, причём три регистра имеют специальные функции; таким образом, «настоящих» регистров общего назначения с точки зрения программиста всего 13, в то время как целый ряд «настоящих» *RISC*-процессоров имеют их больше сотни). С другой стороны, уникальной особенностью этой архитектуры являлась возможнось исполнения любой команды при соблюдении заданного условия.

Кроме того, в командах обработки данных в ряде случаев возможно совмещение выполнения основной операции (например, сложения) со сдвигом. Наконец, команды чтения и записи памяти у *ARM* располагают развитым набором видов адресации, который превосходит по своим возможностям не только *RISC*, но и основную массу *CISC*-процессоров.

Однако время показало ошибочность идеи, на которой базировались «настоящие» *RISC*-архитектуры: достижение высокой производительности через быстрое выполнение простых команд в противовес медленному выполнению сложных команд у *CISC*-процессоров (предполагалось, что, пока *CISC* выполнит одну свою сложную команду, *RISC* успеет выполнить несколько простых и в итоге обгонит *CISC* по производительности, хотя каждая команда индивидуально выполнит меньший объём действий, чем одна команда *CISC*). На практике оказалось, что даже очень неудачные с архитектурной точки зрения системы команд можно выбирать из памяти, декодировать и исполнять чрезвычайно. Неудивительно, что в своём развитии архитектура *ARM* постепенно эволюционировала в сторону *CISC*-процессоров.

На сегодняшний день у *ARM* от *RISC* остались только две вещи: невозможность прямой работы с операндами в памяти и название. Некоторые имеющиеся у современных процессоров *ARM* команды (в частности, деление) недопустимы в «настоящих» *RISC* в связи с их сложностью и переменным временем выполнения.

Современные 32-разрядные процессоры архитектуры *ARM* способны обрабатывать 8-, 16- и 32-разрядные целые числа, а процессоры архитектуры A*RMv*8-*A* – и 64-разрядные. Возможность обработки вещественных чисел является необязательной и достигается с помощью сопроцессоров (которые, если присутствуют, выполняются на одном кристалле с собственно центральным процессором и с точки зрения прикладного программиста неотличимы от него). Старшие модели способны работать с виртуальной памятью, поскольку имеют необходимое для этого устройство управления памятью (*MMU*). Более простые изделия не располагают *MMU*, но в некоторых случаях имеют устройство защиты памяти (*MPU*), не позволяющее реализовать виртуальную память, но обеспечивающее необходимую поддержку аппаратной защиты одних выполняющихся процессором задач от других [4].

## 1.3 История, версии и достоинства

С момента появления архитектуры *ARM* и до 2011 года появилось семь «больших» версий архитектуры – до *ARMv*7 включительно. Кроме того, если не у всех, то у многих версий имеются разновидности. Несмотря на все различия, все эти версии и разновидности являются 32-разрядными, используют одинаковый с точки зрения прикладного программиста набор регистров общего назначения и функционально одинаковый базовый набор команд (хотя каждая последующая версия добавляет новые команды и иногда расширяет возможности уже имевшихся).

Ранние процессоры архитектуры *ARM* использовали 26-разрядные адреса памяти. Первой версией с 32-разрядной адресацией и одновременно последней с 26-разрядной стала *ARMv*3. Все эти версии к 2010 году практически полностью исчезли, поэтому в дальнейшем они обсуждаться не будут.

Первой по времени появления из пока ещё достаточно широко используемых архитектур стала *ARMv*4. В ней впервые отказались от поддержки 26-разрядных адресов, но зато в дополнение к командам обработки слов (32 бита) и байтов ввели команды обработки полуслов (16 бит).

Вскоре появилась разновидность *ARMv*4*T*, в которой впервые был реализован набор команд *Thumb*. Команды этого набора функционально являлись подмножеством основного набора команд (с этого момента обозначаемого как набор команд *ARM*), причём в них отсутствовали команды системного назначения. По этой причине ни для этой, ни для основной массы последующих архитектур вплоть до версии 7 невозможно создать ОС или системно-независимую программу (т. е. программу, работающую на «голом железе» без использования ОС), использующую только систему команд *Thumb*: начальные участки обработчиков прерываний, а также некоторые действия по управлению процессором можно выполнить только с помощью команд набора *ARM*. Поскольку каждая команда *Thumb* кодируется полусловом (16 бит), а команда *ARM* всегда занимает целое слово (32 бита), программа, написанная с использованием системы команд *Thumb*, почти всегда занимает существенно меньше памяти, чем программа на *ARM* (конечно, при условии ручного её кодирования на языке ассемблера или при использовании компилятора, обеспечивающего нормальную поддержку *Thumb*, чего нет, например, в *GCC*).

Помимо *ARMv*4 и *ARMv*4*T*, четвёртая версия архитектуры *ARM* имела ещё две разновидности: *ARMv*4*x*M и *ARMv*4*TxM*, отличавшиеся от первых двух отсутствием команды длинного умножения (сомножители по 32 разряда, результат – 64 разряда). К началу 2010 года все эти разновидности, кроме *ARMv*4*T*, уже утратили актуальность и исчезли с рынка.

Пятая версия архитектуры в «чистом» виде – *ARMv*5, а также её разновидность без длинного умножения *ARMv*5*xM* тоже уже не употребляются. От предыдущей архитектуры они отличаются несколько расширенной системой команд. В частности, именно в *ARMv*5 появились первые команды набора *ARM*, не имеющие возможности условного выполнения.

*Thumb*-разновидность *ARMv*5*T* отличается от A*RMv*4*T* наличием нескольких дополнительных команд, улучшенными возможностями взаимодействия *ARM*- и *Thumb*-кода и некоторыми другими усовершенствованиями; она ещё иногда встречается. Её разновидность с «обрезанными» возможностями умножения, *ARMv*5*TxM*, из употребления вышла.

Вскоре после появления процессоров семейства *ARMv*5*T* архитектура была вновь весьма существенно расширена: появилась целая группа команд, намного упростившая реализацию алгоритмов цифровой обработки сигналов (так называемые *DSP*-команды). Новое семейство получило обозначение *ARMv*5*TE* и весьма широко используется до сих пор. Некоторые ранние реализации этого семейства имели лишь часть новых команд; эта разновидность получила обозначение *ARMv*5*TExP* и уже почти не применяется.

Венцом развития пятой версии архитектуры стало включение специальных средств, предназначенных для упрощения реализации виртуальной машины *Java* (технология *Jazelle*; в отличие от большинства других аспектов архитектуры *ARM*, она не документирована). Такие процессоры получили обозначение *ARMv*5*TEJ* и, вероятно, стали самыми распространёнными для пятой версии архитектуры.

Очередная версия архитектуры, *ARMv*6, изначально не имела разновидностей и включала все возможности *ARMv*5*TEJ*, но с рядом улучшений. Так, в ней были расширены обе системы команд, пересмотрена архитектура подсистемы памяти (ставшая намного более унифицированной; до появления *ARMv*6 очень многие важные решения отдавались «на откуп» производителям, из-за чего при переносе ОС с одного процессора на другой зачастую требовалось выполнить очень много доработок), обеспечен нормальный доступ к не выровненным данным в памяти и др.

Через некоторое время появилась разновидность *ARMv*6*T*2, в которой система команд *Thumb* была существенно расширена и получила наименование *Thumb*-2. Помимо нескольких новых 16-разрядных команд, в *Thumb*-2 появилось большое число 32-разрядных команд, покрывающих почти все возможности системы команд *ARM*. Особого распространения эта разновидность, однако, не получила.

Ещё одной разновидностью шестой версии архитектуры является *ARMv*6*K*, включающая некоторые расширения для упрощения реализации ядер операционных систем и тому подобного ПО. Некоторые процессоры этой разновидности поддерживают расширения безопасности, призванные упростить реализацию высоконадёжных систем, а также создание систем виртуальных машин. Эта подверсия иногда обозначается *ARMv*6*Z* или *ARMv*6*KZ*. Следует отметить, что версии *ARMv*6*K*/*KZ* не поддерживают систему команд *Thumb*-2, а версия *ARMv*6*T*2 не включает улучшения, реализованные в *ARMv*6*K*, т. е. эти две подверсии совместимы между собой только на уровне базовой версии *ARMv*6.

Повсеместную распространённость, особенно в потребительских мобильных устройствах (телефонах и планшетах) получила следующая версия архитектуры, *ARMv*7, более известная под именем *Cortex* (по общему имени реализующих её процессорных ядер). В ней фирма *ARM* впервые явным образом разделила свои разработки по сферам применения, или по «профилям», как это именуется в документации.

Процессоры *A*-профиля (версия A*RMv*7-*A*, ядра семейства *Cortex*-*A*) являются прямыми потомками предыдущих версий архитектуры, сохраняя с ними практически полную программную совместимость «снизу вверх». Они реализуют три системы команд: *ARM*, *Thumb*/*Thumb*-2 и *ThumbEE* (специфическая разновидность *Thumb*, предназначенная для использования компиляторами времени выполнения), могут поддерживать технологию *Jazelle*, а также иметь расширения безопасности и поддерживать виртуализацию. Система команд *Thumb*/*Thumb*-2 по сравнению с версией *ARMv*6*T*2 была расширена, благодаря чему, сохраняя намного более высокую плотность кода (а значит, потребляя меньше памяти), чем система команд *ARM*, *Thumb*/*Thumb*-2 обеспечивает практически равную с ней производительность. Все процессоры *Cortex*-*A* в обязательном порядке оснащены устройством управления памятью (*MMU*); часто на одном кристалле размещаются несколько процессорных ядер, дополненных графическим процессором. Такие микросхемы предназначены для использования в мощных мобильных устройствах. Нередко производители расширяют их возможности, добавляя специализированные узлы. Например, фирма *nVidia* выпускает семейство процессоров *Tegra*, у которых, помимо процессорных ядер *ARM*, имеется графическое ядро самой *nVidia*, архитектурно близкое к графическим процессорам, используемым на ПК, и обеспечивающее намного более высокую производительность, чем графическое ядро самой *ARM* (*Mali*).

Кристаллы *R*-профиля (версия *ARMv*7-*R*, ядра семейства *Cortex*-*R*) в целом аналогичны процессорам *A*-профиля (хотя поддержка *ThumbEE* и ряда других «продвинутых» возможностей не является обязательной), но оснащены не устройством управления памятью, а значительно более простым устройством защиты памяти (*MPU*). Они предназначены в первую очередь для использования в высоконадёжных промышленных контроллерах и тому подобных устройствах, работающих в реальном времени (отсюда буква *R* в обозначении), не нуждающихся в виртуальной памяти (её использование прямо противоречит целям работы в реальном времени), однако имеющих достаточно высокие требования к производительности.

Наконец, *M*-профиль (версии *ARMv*6-*M* и *ARMv*7-*M*, ядра семейства *Cortex*-*M*), строго говоря, не относится к «настоящим» *ARM*-процессорам. Во-первых, он кардинальным образом отличается по системной архитектуре от всех прочих разработок фирмы *ARM*, а соответственно, на системном уровне не совместим ни с более ранними процессорами, ни с другими профилями 7-й версии архитектуры. Во-вторых, в этих кристаллах реализована только система команд *Thumb* (*ARMv*6-*M*, ядра *Cortex*-*M*0 и -*M*1) или *Thumb*-2 (*ARMv*7-*M*, все прочие ядра *Cortex*-*M*), а команды набора *ARM* не поддерживаются. Предназначена эта серия для использования в качестве микроконтроллеров малой и средней производительности. Благодаря низкой стоимости и энергопотреблению они могут успешно конкурировать с намного более слабыми по вычислительными возможностям 8- и 16-разрядными микроконтроллерами. Заметим, что принадлежность ядер *Cortex*-*M*0 и -*M*1 к 6-й версии архитектуры является чисто формальной и показывает, что по своим возможностям (системе команд) они находятся ниже «настоящей» архитектуры *ARMv*7-*M*.

Первые 64-разрядные процессоры (версия архитектуры *ARMv*8) появились в 2012 году. Здесь сохранилось разделение по профилям, причём, если *A*-профиль стал действительно новым, то *R*- и *M*-профили остались 32-разрядными и являются, по существу, несколько расширенными вариантами этих же профилей 7-й версии архитектуры.

У процессорных ядер, разработанных *ARM*, существует своя система нумерации, не связанная с номером версии архитектуры. Так, ядра *ARM*7*TDM*I и *ARM*920*T* реализуют версию *ARMv*4*T*, большинство других ядер семейства *ARM*9 – различные разновидности версии *ARMv*5 (например, ядро *ARM*926*EJ*-*S* реализует архитектуру *ARMv*5*TEJ*). Все ядра семейства *Cortex*, за исключением *Cortex*-*M*0 и -*М*1, относятся к разновидностям 7-й или 8-й версии архитектуры (ядрам *Cortex*-*M*0 и -*М*1, как уже говорилось, соответствует версия *ARMv*6-*M*). Между собой ядра одной и той же версии и разновидности архитектуры могут отличаться конструкцией и объёмом кэш-памяти, наличием *MMU*, *MPU*, *FPU* и т. д.

Некоторые расширения, предпринятые фирмой *ARM* в своих разработках, оказались невостребованными или недостаточно эффективными. К ним в первую очередь относятся технология *Jazelle* и система команд *ThumbEE*, которые уже объявлены устаревшими и могут отсутствовать в новейших ядрах [5].

# 2 Платформа программного обеспечения

## 2.1 Основные понятия

В традиционном понимании платформа – это комплекс аппаратных и программных средств, на котором функционирует программное обеспечение пользователя ЭВМ. Основа аппаратной платформы (*hardware*-платформы) – процессор. Тип процессора определяет архитектуру аппаратных средств –аппаратную платформу, т. е. тип и характеристики компьютера.

Понятия «программная платформа» (*software*-платформа), или «программное обеспечение» вошли в жизнь с развитием компьютерной индустрии. Без программного обеспечения компьютер – всего лишь электронное устройство, которое не управляется, и потому не может приносить пользы. В зависимости от функций, выполняемых программным обеспечением, его можно разделить на две большие группы: системное и прикладное программное обеспечение.

Системное программное обеспечение – это «программная оболочка» аппаратных средств, предназначенная для отделения остальных программ от непосредственного взаимодействия с оборудованием и организации процесса обработки информации в компьютере. К системному программному обеспечению относятся такие типы программ, как операционные системы, различные сервисные средства, функционально дополняющие возможности операционных систем, инструментальные средства (системы управления базами данных, программирования, оболочки экспертных систем).

Прикладное программное обеспечение предназначено для решения определенных задач пользователя.

Операционная система (ОС) – это комплекс взаимосвязанных системных программ, которые загружаются при включении компьютера и постоянно находятся в памяти компьютера. Они производят диалог с пользователем, осуществляют управление компьютером, его ресурсами (оперативной памятью, местом на дисках и т.д.), запускают другие (прикладные) программы на выполнение. Операционная система обеспечивает пользователю и прикладным программам удобный способ общения (интерфейс) с устройствами компьютера.

Основная причина необходимости операционной системы состоит в том, что элементарные операции для работы с устройствами компьютера и управления ресурсами компьютера – это операции очень низкого уровня, поэтому действия, которые необходимы пользователю и прикладным программам, состоят из нескольких сотен или тысяч таких элементарных операций.

Операционная система скрывает от пользователя все эти сложные и ненужные подробности и предоставляет ему удобный интерфейс для работы. Она выполняет также различные вспомогательные действия, например, копирование или печать файлов. Операционная система осуществляет загрузку в оперативную память всех программ, передает им управление в начале их работы, выполняет различные действия по запросу выполняемых программ и освобождает занимаемую программами оперативную память при их завершении.

В функции операционной системы входит:

– осуществление диалога с пользователем;

– ввод-вывод и управление данными;

– планирование и организация процесса обработки программ;

– распределение ресурсов (оперативной памяти и кэша, процессора, внешних устройств);

– запуск программ на выполнение;

– всевозможные вспомогательные операции обслуживания;

– передача информации между различными внутренними устройствами;

– программная поддержка работы периферийных устройств (дисплея, клавиатуры, дисковых накопителей, принтера и др.) [6].

## 2.2 Обоснование выбора платформы

Выбор *ARM* для написания программы калибровки монитора обеспечивает множество преимуществ, а именно:

1 Совместимость с аппаратной платформой. *ARM*-архитектура становится все более распространенной в устройствах *Apple*. Использование *ARM*-версии *macOS* обеспечивает максимальную совместимость с современными компьютерами *Mac*, такими как *MacBook Air*, *MacBook Pro* и *Mac mini*, работающими на базе процессоров *Apple Silicon*.

2 Производительность и энергоэффективность. Процессоры *Apple* *Silicon* обеспечивают высокую производительность при одновременном снижении энергопотребления. Это позволяет создавать эффективные программы калибровки монитора, которые могут выполняться быстро и эффективно, не загружая систему.

3 Нативная поддержка инструментов разработки. Для *ARM*-версии доступны нативные версии многих популярных инструментов разработки, таких как *Xcode*. Это упрощает процесс разработки программы калибровки монитора, так как разработчики могут использовать современные средства разработки и получать доступ к последним функциям и возможностям.

4 Более простая миграция. Поскольку *ARM* становится стандартом для новых устройств *Apple*, написание программы для калибровки монитора с учетом этой платформы обеспечивает более простую миграцию и совместимость с будущими моделями компьютеров *Mac*.

5 Интеграция с экосистемой *Apple*. Использование *ARM* обеспечивает более гармоничную интеграцию с другими устройствами и сервисами *Apple*, такими как *iPhone*, *iPad* и *iCloud*. Это может быть особенно полезно для разработчиков, чьи программы калибровки монитора взаимодействуют с другими устройствами и сервисами *Apple*.

## 2.3 Анализ операционной системы для написания программы

В рамках данного курсового проекта была выбрана операционная система *MacOS*.

*MacOS* – это операционная система, разработанная компанией *Apple* для своих персональных компьютеров *Mac*. Она была первоначально выпущена в 1984 году и с тех пор прошла множество изменений и усовершенствований.

*MacOS* представляет собой *Unix*-подобную операционную систему, которая работает на базе ядра *Darwin*. В отличие от других операционных систем, таких как *Windows*, *MacOS* разработана специально для использования на компьютерах, произведенных компанией *Apple*, и не может быть установлена на других устройствах.

*MacOS* предлагает множество удобных функций и приложений, которые облегчают работу на компьютере. Некоторые из них включают в себя:

– *Siri* – голосовой помощник, который помогает пользователю искать информацию и выполнять различные задачи;

– iMessage – мессенджер для обмена сообщениями;

– iTunes – мультимедийный плеер.

*MacOS* также имеет множество удобных функций, таких как *Spotlight* – поисковую систему, которая помогает быстро находить файлы и приложения на компьютере, и *Mission* *Control* – управление вкладками и окнами приложений. Кроме того, *MacOS* обладает высоким уровнем безопасности и имеет встроенные средства защиты от вирусов и других угроз.

Общее устройство *MacOS* основано на концепции оконного интерфейса, где приложения запускаются в окнах, которые можно перемещать, изменять размер и закрывать. *MacOS* также использует меню и панели инструментов для управления приложениями и доступа к различным функциям.

Одним из ключевых элементов *MacOS* является *Dock* – панель быстрого запуска приложений и доступа к часто используемым файлам и папкам. Док расположен в нижней части экрана и позволяет легко переключаться между открытыми приложениями и файлами. Для запуска приложения достаточно просто щелкнуть на его значке в *Dock*.

*MacOS* имеет интегрированную систему управления файлами и папками, которая называется *Finder*. *Finder* позволяет пользователю быстро находить, просматривать и управлять файлами и папками на компьютере, а также осуществлять простые операции, такие как копирование, перемещение и удаление файлов.

В *MacOS* есть возможность создавать и использовать виртуальные рабочие столы, которые позволяют организовать открытые приложения и файлы по группам для удобства работы. Это особенно полезно при работе с несколькими приложениями одновременно.

Также *MacOS* поддерживает множество стандартов и протоколов, что делает его совместимым с большинством технологий и устройств. Например, она поддерживает *Bluetooth*, *Wi*-*Fi*, *USB*, *Ethernet* и другие технологии, что позволяет подключать к компьютеру различные устройства и обмениваться данными с другими устройствами.

Одним из ключевых преимуществ *MacOS* является его экосистема. Она интегрируется с другими устройствами и сервисами *Apple*, обеспечивая беспрерывную работу между различными устройствами и удобство использования.

Например, с помощью функции *Continuity* можно легко переносить работу между *Mac* и другими устройствами *Apple*, такими как *iPhone* и *iPad*. Можно начать работать над документом на своем *Mac*, а затем продолжить работу над ним на своем *iPhone* или *iPad* без необходимости сохранять и синхронизировать файлы вручную.

Благодаря функции *Handoff* можно начать работу над приложением на одном устройстве, а затем продолжить ее на другом устройстве без прерывания работы. Например, вы можно начать работу над письмом на своем *iPhone*, а затем продолжить его на своем *Mac*, не теряя прогресса.

В *MacOS* также встроен сервис *iCloud*, который позволяет хранить файлы в облаке и синхронизировать их между различными устройствами *Apple*. Например, если сохранить документ на *Mac*, он автоматически появится на всех устройствах, подключенных к этому *iCloud*.

Кроме того, *MacOS* интегрируется с другими сервисами *Apple*, такими как *iTunes*, *Apple* *Music*, *Apple* *TV* и другие, что позволяет получить более полный и удобный опыт использования устройств *Apple*.

Преимущества *MacOS*:

1 Надежность и безопасность. *MacOS* известна своей высокой надежностью и безопасностью, так как она основана на *UNIX*, который имеет более надежную систему безопасности, чем *Windows*.

2 Удобство использования. *MacOS* имеет простой и интуитивно понятный интерфейс, что делает его легким в использовании для новых пользователей. Также *MacOS* часто использует сенсорные технологии, такие как трекпад, что делает работу с устройством более удобной и эффективной.

3 Экосистема. *MacOS* интегрируется с другими устройствами и сервисами *Apple*, что обеспечивает бесперебойную работу между различными устройствами и сервисами, такими как *iPhone*, *iPad*, *iCloud* и т. д.

4 Качество программного обеспечения. ПО для *MacOS* обычно имеет более высокое качество, чем ПО для *Windows* благодаря строгому контролю качества их создания со стороны *Apple*.

Недостатки *MacOS*:

1 Цена. *MacOS* устройства обычно стоят дороже, чем устройства с *Windows*.

2 Ограниченность выбора оборудования. *MacOS* не поддерживает такое многообразие аппаратных устройств, как *Windows*, что может ограничить выбор пользователей.

3 Ограниченность выбора программного обеспечения. Некоторые программы, особенно для специализированных профессиональных задач, могут быть недоступны на *MacOS*.

4 Необходимость переобучения. При переходе на *MacOS* может потребоваться некоторое время для переобучения и адаптации к новой системе.

В целом, *MacOS* – это мощная, удобная и интуитивно понятная операционная система, которая обеспечивает надежную и безопасную работу на компьютерах *Mac*, а экосистема операционной системы представляет собой мощный и удобный инструмент, который обеспечивает бесперебойную работу между различными устройствами и сервисами *Apple*, облегчая жизнь пользователям и упрощая их работу с устройствами [7].

# 3 Теоретическое обоснование разработки              программного продукта

## 3.1 Обоснование необходимости разработки

Современные образовательные процессы требуют эффективного использования цифровых технологий для повышения удобства и производительности как студентов, так и преподавателей. Разработка мобильного приложения расписания БГУИР является актуальной и необходимой задачей по следующим причинам:

1 Мобильность. Студенты и преподаватели всегда имеют доступ к расписанию через свои смартфоны, что особенно важно в условиях динамичного учебного процесса.

2 Интуитивный интерфейс. Приложение позволяет легко находить и просматривать актуальную информацию без необходимости поиска в браузере или бумажных носителях.

3 Обновление расписания в реальном времени. Изменения в расписании (например, перенос занятий, замены преподавателей) будут мгновенно доступны пользователям.

4 Снижение риска ошибок. Автоматизация исключает человеческий фактор при передаче информации.

5 Персонализация. Возможность настроить расписание под конкретную группу, факультет или преподавателя.

6 Фильтрация и поиск. Удобный поиск по группам и преподавателям, а также фильтрация для расписания конкретной подгруппы.

7 Скорость доступа. Приложение исключает необходимость многократного посещения веб-сайта университета или сверки бумажных версий расписания.

8 Синхронизация. Возможность интеграции с календарями Google или iOS для создания персональных учебных графиков.

9 Цифровизация учебного процесса. Соответствие трендам цифровой трансформации образовательных учреждений.

10 Повышение имиджа университета. Наличие современного мобильного приложения демонстрирует заботу о студентах и преподавателях, улучшая репутацию БГУИР.

11 Возможность интеграции с текущими системами. Приложение может быть связано с уже существующей базой данных расписания, минимизируя затраты на внедрение.

Таким образом, создание мобильного приложения расписания БГУИР будет способствовать оптимизации учебного процесса, повышению уровня комфорта и информированности студентов и преподавателей, а также укреплению позиций университета как инновационного образовательного учреждения.

## 3.2 Технологии программирования, используемые для решения                 поставленных задач

В рамках данного курсового проекта было выбрано реализовать мобильное приложение расписания БГУИР с помощью фреймворка *Flutter* на языке *Dart*.

*Flutter* – это платформа с открытым исходным кодом, который разработан и поддерживается *Google*. Фронтенд-разработчики и фуллстек-разработчики используют *Flutter* для создания пользовательского интерфейса приложений (*UI*) для различных платформ с применением единой базы кода.

После выпуска *Flutter* в 2018 году эта платформа в основном поддерживала разработку мобильных приложений. Теперь *Flutter* поддерживает разработку приложений на шести платформах: *iOS*, *Android*, веб-интерфейс, *Windows*, *MacOS* и *Linux*.

*Flutter* упрощает процесс создания единообразных привлекательных пользовательских интерфейсов для приложения на шести поддерживаемых платформах.

Разработка приложения для отдельной платформы, например *iOS*, называется разработкой платформозависимых приложений. Разработка кроссплатформенных приложений, напротив, является созданием приложений для нескольких платформ с единой базой кода.

Поскольку при разработке платформозависимых приложений разработчики создают программное обеспечение для конкретной платформы, они имеют полный доступ ко встроенным функциям устройства. Этим обычно обусловлена более высокая производительность и скорость платформозависимых приложений по сравнению с кроссплатформенными.

С другой стороны, чтобы приложение работало на нескольких платформах, то для разработки платформозависимого приложения потребуется больше кода и разработчиков. Помимо этого, платформозависимые приложения сложнее запускать на различных платформах с единообразным пользовательским интерфейсом. В этом случае пригодится платформа для разработки кроссплатформенных приложений, такая как *Flutter*.

Разработка кроссплатформенных приложений дает разработчикам возможность использовать один язык программирования и одну базу кода, чтобы создать приложение для нескольких платформ. Если вы выпускаете приложение для нескольких платформ, кроссплатформенная разработка потребует меньших затрат и меньшего времени, чем разработка платформозависимого приложения.

Также этот процесс позволяет разработчиков создавать более единообразные интерфейсы для различных платформ.

Этот подход может иметь недостатки по сравнению с разработкой платформозависимых приложений, в том числе ограниченный доступ к функциям устройства. Однако во *Flutter* реализованы возможности, которые делают разработку кроссплатформенных приложений более простой и высокоэффективной.

*Flutter* выделяется среди платформ для кроссплатформенной разработки благодаря следующим преимуществам:

1 Производительность близка к производительности платформозависимых приложений. *Flutter* использует язык программирования *Dart* и компилируется в машинный код. Устройства понимают этот код, что обеспечивает быструю работу и высокую производительность.

2 Быстрое, единообразное и настраиваемое отображение. *Flutter* не полагается на платформозависимые инструменты отображения, а использует для отображения пользовательского интерфейса графическую библиотеку *Google Skia* с открытым исходным кодом. Это предоставляет пользователям единообразные визуальные элементы, независимо от платформы, используемой для доступа к приложению.

3 Удобные инструменты для разработчиков Компания *Google* создала *Flutter* с акцентом на простоте использования. Благодаря таким функциям, как горячая перезагрузка, разработчики могут предварительно просматривать, как будут выглядеть изменения в коде, без потери состояния. Другие инструменты, такие как инспектор виджетов, упрощают визуализацию и решение проблем в макетах пользовательского интерфейса.

*Flutter* использует язык программирования с открытым исходным кодом *Dart*, который также разработан в *Google*. *Dart* оптимизирован для создания пользовательского интерфейса, и многие его преимущества используются во *Flutter*.

Например, одна из возможностей *Dart*, которая используется во *Flutter*, – защита от нулевых ссылок. Защита от нулевых указателей упрощает обнаружение распространенных ошибок, называемых ошибками нулевых ссылок. Эта возможность сокращает время, затрачиваемое разработчиками на обслуживание кода, освобождая время на создание приложений.

Разработчики создают макеты пользовательского интерфейса на *Flutter* с помощью виджетов. Это значит, что все, что пользователь видит на экране, от окон и панелей до кнопок и текста, состоит из виджетов.

Виджеты *Flutter* разработаны так, чтобы разработчикам было просто настраивать их. Во *Flutter* это реализовано с использованием композиционного подхода. Это значит, что большинство виджетов состоит из меньших виджетов и большинство базовых виджетов имеют специфическое предназначение. Это позволяет разработчикам комбинировать или редактировать виджеты для создания новых виджетов.

*Flutter* отображает виджеты с помощью собственного графического движка, не полагаясь на встроенные виджеты платформы. Благодаря этому пользователям предоставляется один и тот же внешний вид приложения *Flutter* на разных платформах. Кроме того, данный подход предоставляет разработчикам гибкость, потому что некоторые виджеты *Flutter* могут выполнять функции, недоступные для виджетов, специфических для платформы.

Кроме того, *Flutter* упрощает использование виджетов, разработанных сообществом. Архитектура *Flutter* поддерживает множество библиотек виджетов, и *Flutter* поддерживает сообщество в разработке и поддержке новых.

*Flutter* поставляется с обширным каталогом виджетов при загрузке. В каталоге 14 категорий, в том числе стили, Купертино (виджеты в стиле *iOS*) и Материальные виджеты (виджеты, соответствующие рекомендациям *Google* по материальному дизайну).

Кроме того, *Flutter* поставляется с макетами и темами, благодаря чему разработчики могут сразу приступать к компоновке.

*Flutter* поддерживается компанией *Google* и активным сообществом разработчиков программного обеспечения с открытым исходным кодом в *Reddit*, *Discord*, *Slack*, *Stack* *Overflow* и *Gitter*. Компания *Google* постоянно обновляла *Flutter* с момента выпуска в 2018 году, включая обновление *Flutter* 3 в 2022 году, которое распространило стабильную поддержку на *macOS* и *Linux*.

Чтобы было проще осваивать *Flutter*, компания *Google* снабдила эту платформу обширной документацией и учебными пособиями, приведенными на сайте *Flutter*. Для привлечения пользователей *Flutter* компания *Google* проводит глобальные мероприятия, поддерживает проекты сообщества и спонсирует состязания разработчиков. О предстоящих событиях можно узнать на сайте *Flutter*.

Сообщество *Flutter* создало тысячи сторонних пакетов и отличных инструментов, которые упрощают разработку. Эти библиотеки доступны на *pub*.*dev* [8].

*Dart* – язык программирования, который создали для использования в веб-разработке. Идея его создания – постепенно заменить популярный язык *JavaScript*: *Dart* позиционируется как решение, лишенное ключевых недостатков *JS*.

*Dart* считается языком общего назначения, но его создатели ориентировали его в первую очередь на фронтенд: создание интерфейсов и взаимодействие с браузером. А собственная платформа *Dart* *VM* призвана заменить *Node*.*js* – инструмент, с помощью которого пишут код на *JavaScript* для серверной части.

Это высокоуровневый язык: он оперирует абстрактными понятиями, далекими от «железа», но близкими к человеку. Поэтому код на *Dart* лаконичный и понятный, а синтаксис легко освоить. Особенно удобным *Dart* считается для людей, которые до этого уже изучали другие языки программирования.

Создатели *Dart* – компания *Google*, которая также придумала язык *Go*. У *Dart* открытый исходный код, язык полностью бесплатный и свободно распространяется.

Так как основная идея создания *Dart* – заменить *JavaScript*, сам по себе язык ориентирован на веб-разработку. Во встроенных инструментах языка много средств для работы с браузером и интерфейсами, веб-страницами – словом, фронтенд-частью разработки. В то же время в нем есть и инструменты для бэкенда и серверного программирования.

Но пока в веб-программировании, по крайней мере в России, *Dart* не прижился. В основном разработчиков на этом языке ищут в проектах по мобильной разработке. Дело в том, что на основе *Dart* написан *Flutter* – популярный фреймворк для создания мобильных приложений. Поэтому большинство вакансий по *Dart* – это работа с *Flutter* и с приложениями под *iOS* или *Android*.

В этом нет ничего удивительного. Возможности *Dart* для создания интерфейсов легко применить в мобильной разработке: они позволяют удобно и быстро создавать приложения.

Кроме того, на *Dart* написана надстройка *CSS* под названием *SASS*. Ею пользуются фронтенд-разработчики для более удобной верстки.

Особенности языка *Dart*:

1 Ориентированность на создание интерфейсов. *Dart* создавался как язык для интерфейсов, поэтому в нем есть много фишек для их разработки. Сейчас эта особенность нашла применение в программировании мобильных приложений: создавать визуальный интерфейс с помощью *Dart* довольно удобно. Идея создателей языка была в том, чтобы уменьшить количество переключений между разметкой и кодом – чтобы разработчик не терял времени и не отвлекался. На это и направлены некоторые фишки языка. Например, при компилировании в *JavaScript* автоматически создается шаблонная *HTML*-страница с разметкой. Ее сразу можно открыть в браузере и посмотреть, что получилось.

2 Наследие *C* и *C*++. *Dart* испытал сильное влияние системных языков *C* и *C*++, и это нашло отражение в его синтаксисе. Он *C*-подобный: это значит, что правила написания кода и его структура похожи на *C*. Хотя *Dart* проще в освоении и ориентирован на другие цели, он унаследовал некоторые важные элементы: например, основной блок программы заключается в функцию *main*().

3 Факультативная типизация. Типизация в языках программирования бывает статической и динамической. В первом случае разработчик должен сам указывать, к какому типу будет относиться переменная: число, строка, массив или что-то еще. Во втором случае язык решает это сам на основании того, какие данные записал в переменную разработчик. В *Dart* объединили эти подходы – типизация там факультативная. Это значит, что разработчик может указать тип при создании переменной, а может не указывать – допустимы оба подхода. Какой из них лучше использовать, зависит от задачи.

4 Компилируемость. В отличие от интерпретируемого *JavaScript*, *Dart* – компилируемый язык. Это значит, что код на нем проходит через специальную программу-переводчик – компилятор. Она переводит написанное в машинные коды, которые понимает операционная система, поэтому язык работает быстро и может активнее взаимодействовать с ОС.

5 Два вида компиляции. *AOT* (*ahead* *of* *time*, «прежде времени») – статическая, или опережающая компиляция. Написанная программа целиком переводится в машинные коды, и только после этого ее можно запустить. Это классический вариант компиляции, именно он используется в *C*++ и большинстве компилируемых языков. *JIT* (*just* *in* *time*, «точно вовремя») – динамическая компиляция. Программа компилируется по частям, и скомпилированные части тут же выполняются. Это роднит идею с интерпретацией – другим подходом к исполнению, когда специальная программа-интерпретатор исполняет код построчно. Обычно *JIT* используется в языках с байт-кодом, таких как *Java*. Программа, скомпилированная *AOT*, быстрее запускается и работает, но такая компиляция замедляет разработку. После каждого изменения разработчик должен перекомпилировать код полностью – а это лишнее время. *JIT*, наоборот, ускоряет цикл разработки, но сама по себе программа запускается медленнее и требует больше ресурсов. Поэтому в *Dart* и ввели оба вида компиляции: *JIT* для процесса разработки и *AOT* для финальной реализации уже готового продукта. Это нужно, чтобы экономить время и совмещать плюсы обоих типов.

6 Классическое ООП. В *Dart* реализован классический подход к объектно-ориентированному программированию, которое сейчас считается одной из наиболее популярных парадигм. В нем есть классы, в том числе абстрактные, реализовано наследование – все это без нововведений и экспериментов, так, как предписывает парадигма. Поэтому писать ООП-код в *Dart* довольно удобно: не приходится придумывать «костыли» и бороться с особенностями языка.

7 Изоляторы. *Dart* – однопоточный язык. Это значит, что все программы на нем выполняются в едином потоке действий, а не в нескольких. Однопоточность тоже роднит его с *JavaScript*. Но иногда процессы нужно распараллеливать, чтобы несколько действий могли выполняться одновременно. Для этого в *Dart* реализован механизм изоляторов (*isolate*). Иногда их еще переводят как изоляты. Изолятор – это сущность, которая выполняет какие-то действия. У каждого изолятора своя область памяти, он может выполняться одновременно с другими и обмениваться с ними данными. Это своеобразные виртуальные потоки, созданные на более высоком уровне абстракции, чем настоящие. Управляет ими *Event Loop* – процесс, который работает, пока открыта программа, и поочередно выполняет код внутри разных изоляторов. Он работает с очередью, действующей по принципу *FIFO*: «первый вошел, первый вышел». То есть тот процесс, который «пришел» в очередь раньше, и выполнится раньше других. Очередей две: для простых коротких действий и для блоков кода, которые должны выполниться в ответ на какое-то внешнее событие.

8 Горячая перезагрузка. Удобная особенность *Dart* – горячая перезагрузка, или *hot* *reload*. Термин означает, что код программы можно откорректировать, пока она запущена – и изменения внесутся «на ходу». Для этого не понадобится открывать и снова закрывать программу. В *JavaScript* и похожих языках тоже есть инструменты для *hot* *reload*, но там приходится использовать внешние решения, а в *Dart* эта возможность встроена изначально. Поэтому горячая перезагрузка в этом языке работает быстрее и эффективнее.

Современная версия языка называется *Dart* 2. Она вышла в 2018 году и отличается от более старой версии: в ней улучшили систему типов, сделали синтаксис лаконичнее, добавили новые возможности для создания клиентской части приложений. Инструменты новой версии позволяют реже переключаться между разметкой и кодом, а сущности можно описывать короче.

С выходом *Dart* 2 язык снова испытал взлет популярности. Сейчас новая версия считается более актуальной, поэтому если разработчики говорят про *Dart*, чаще всего они имеют в виду *Dart* 2.

В отличие от других языков, созданных как альтернатива *JS*, *Dart* не разрабатывался на его основе. Это полностью самостоятельный язык программирования, независимый от *JavaScript*. Несмотря на схожесть синтаксиса и назначения, языки не связаны.

Тем не менее код на *Dart* можно скомпилировать в код на *JavaScript* – перевести с помощью специальной утилиты, встроенной в язык. Это сделано потому, что браузеры пока понимают только *JS*, а код на *Dart* выполнить не смогут. Так что разработчик может написать код для фронтенда на *Dart*, а потом автоматически преобразовать его в *JavaScript* и запустить в браузере.

Преимущества *Dart*:

1 Удобство. На языке просто писать, у него лаконичный синтаксис, понятный разработчикам.

2 Простота освоения. Разработчики говорят, что освоить *Dart* легко, в особенности если до этого у человека был опыт программирования.

3 Широкие возможности. В языке много особенностей и фишек, расширяющих возможности и облегчающих разработку.

4 Высокая производительность. *Dart* компилируемый, поэтому быстрый и мощный. Он нормально работает и на клиенте, и на сервере.

5 Поддержка *Google*. Корпорация поддерживает и рекламирует язык, стремится повысить его популярность. Поэтому у *Dart* отличная документация и туториалы [9].

# 4 Проектирование функциональных              возможностей программы

Функциональные требования можно рассматривать как особенность продукта, которые обнаруживает пользователь. Это может быть очевидная функция, например, большая кнопка «Добавить в корзину». Но это также может быть и менее очевидная функция, например, правильный расчет налога с продаж для онлайн-покупки пользователя.

Функциональное требование – это заявление о том, как должна вести себя система. Он определяет, что система должна делать, чтобы удовлетворить потребности или ожидания пользователя. Функциональные требования можно рассматривать как функции, которые обнаруживает пользователь. Они отличаются от нефункциональных требований, которые определяют, как система должна работать внутри (например, производительность, безопасность и т. д.).

Функциональные требования состоят из двух частей: функции и поведения. Функция – это то, что делает система (например, «рассчитать налог с продаж»). Поведение определяется тем, как это делает система (например, «Система должна рассчитать налог с продаж путем умножения покупной цены на налоговую ставку»).

Таким образом, функциональные требования являются ключом к успеху любого проекта по разработке программного обеспечения. Создавая функциональные требования, гарантируется, что каждый в команде разработки понимает, что нужно создать, и может соответствующим образом расставить приоритеты в своей работе [10].

## 4.1 Функциональные требования

Функциональные требования для мобильного приложения расписания БГУИР:

1 Просмотр расписания. Просмотр основного расписания для студентов и преподавателей.

2 Поддержка различных форматов. Просмотр расписания в различных форматах: ежедневное расписание с учётом номера текущей недели, общее расписание недели, расписание экзаменов и консультаций.

3 Поддержка фильтрации. Фильтрация расписании группы по номеру подгруппы.

4 Поиск. Возможность поиска расписания конкретной группы или преподавателя.

5 Детальное отображение. Возможность посмотреть все подробности конкретного предмета в расписании.

6 Совместимость. Поддержка операционных систем *Android* и *iOS*.

## 4.2 Интерфейс программы

Создание понятного и удобного интерфейса (*UI*) является ключевым аспектом разработки любого программного продукта, особенно мобильных приложений. Это влияет на удовлетворенность пользователей, их желание продолжать использовать приложение и общую эффективность выполнения задач.

Основные причины создания понятного и удобного интерфейса:

1 Интуитивность. Пользователи ожидают, что интерфейс будет понятным без необходимости изучать инструкции. Это снижает порог входа для новых пользователей.

2 Скорость выполнения задач. Удобный интерфейс позволяет быстро находить и использовать нужные функции, что делает приложение полезным и функциональным.

3 Логичная структура. Понятный интерфейс снижает вероятность ошибок, таких как неправильный выбор функции или ввод некорректных данных.

4 Эстетика и простота. Удобное приложение вызывает положительные эмоции, что способствует длительному использованию.

5 Меньше необходимости в обучении. Простота использования снижает потребность в обучающих материалах и инструкциях.

6 Выделение среди аналогов. Приложения с удобным интерфейсом чаще выбирают пользователи, даже если их функциональность схожа с конкурентами.

Таким образом, создание понятного и удобного интерфейса является не просто техническим требованием, но стратегическим инструментом, который напрямую влияет на успех приложения. Оно помогает удовлетворить потребности пользователей, минимизировать их усилия и укрепить связь с продуктом, что в конечном итоге приводит к повышению эффективности и популярности приложения.

При запуске приложения открывается главный экран (рисунок 4.1), на котором располагается нижняя панель навигации. Она позволяет легко перемещаться между основными экранами: список групп, список преподавателей и настройки. Изначально открыта вкладка списка групп.

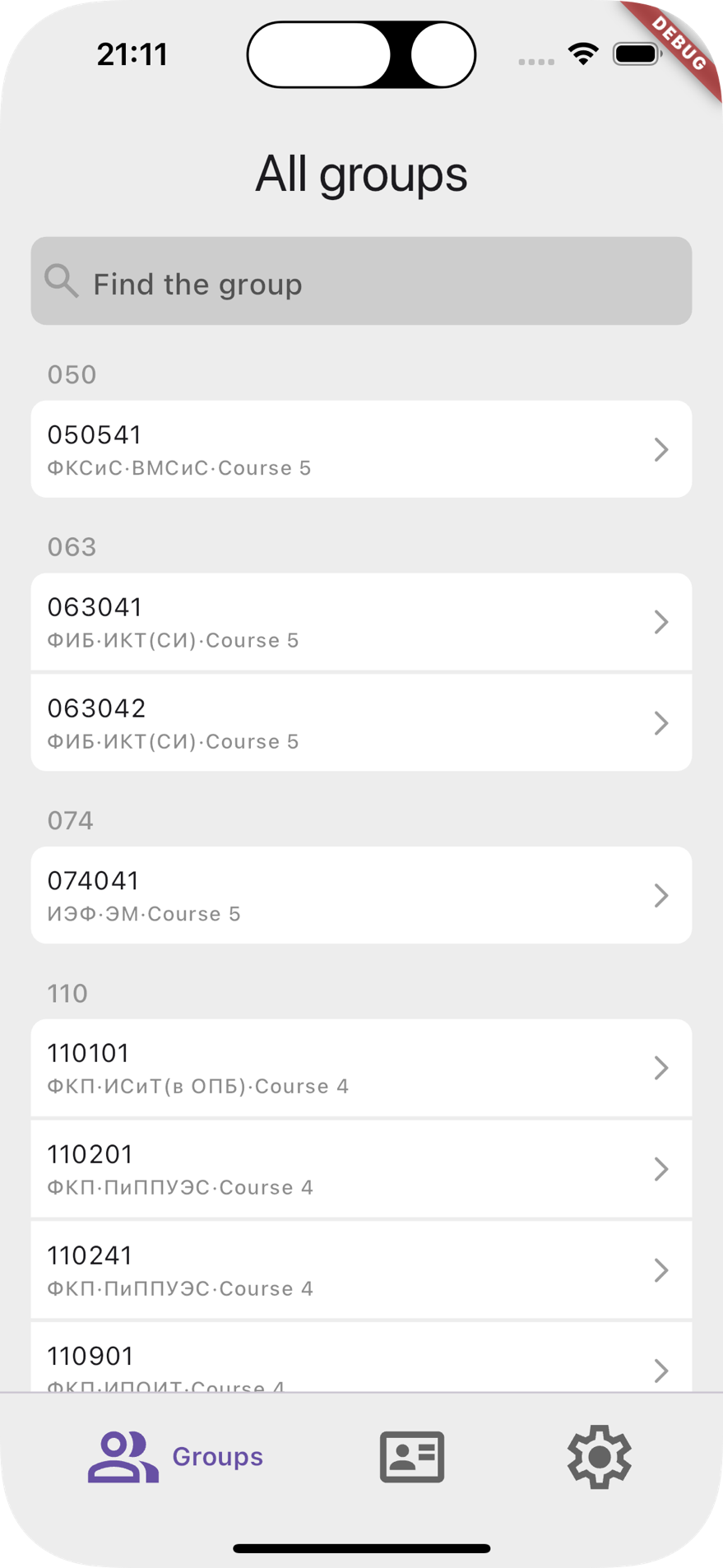


Рисунок 4.1 – Главный экран приложения

Список групп разбит на группы по первым трём цифрам в номере. Также сразу же видна информация о названии факультета, специальности, а также о номере курса, на котором находятся обучающиеся.

Помимо этого вверху экрана находится поле для поиска необходимой группы. При несовпадении введённой информации выводится соответствующее сообщение (рисунки 4.2 и 4.3).

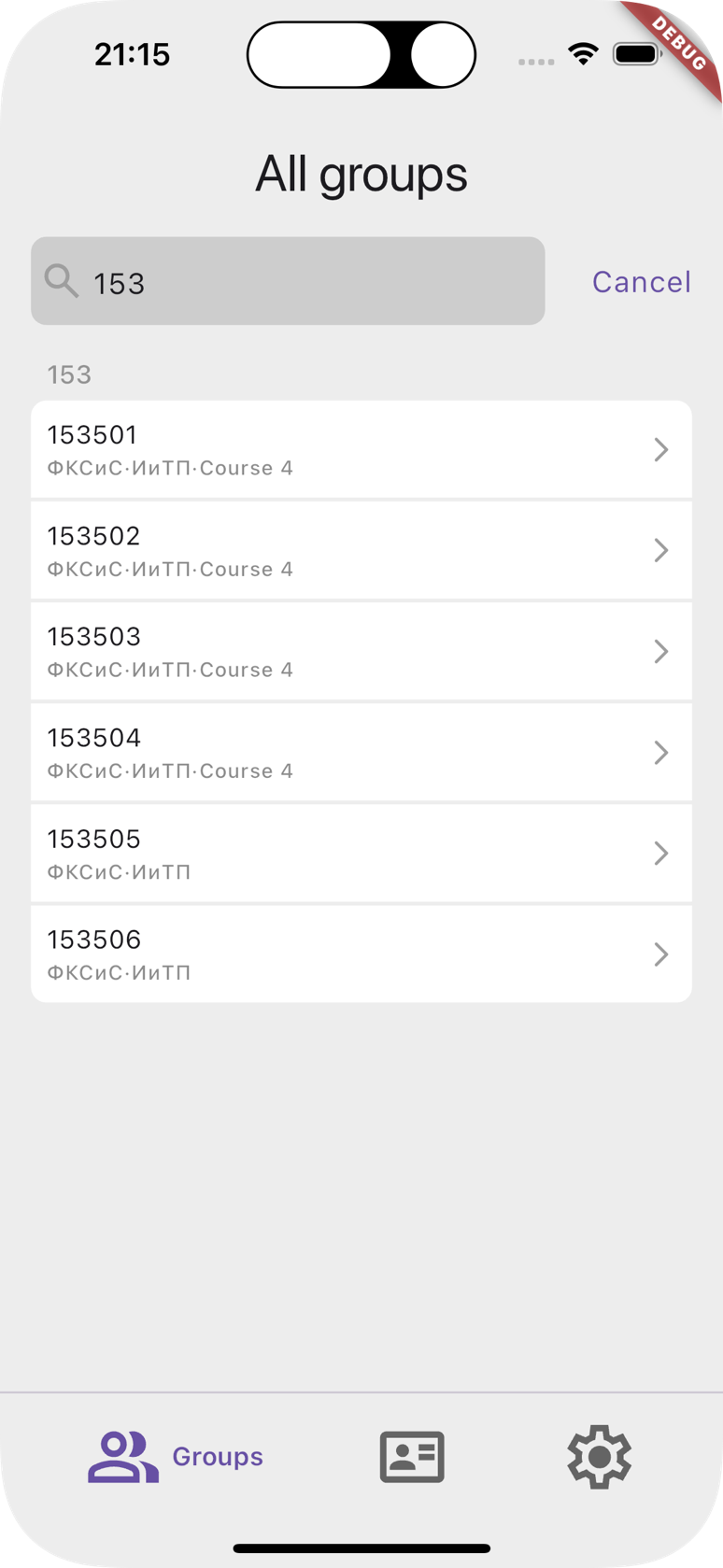


Рисунок 4.2 – Удачный поиск группы

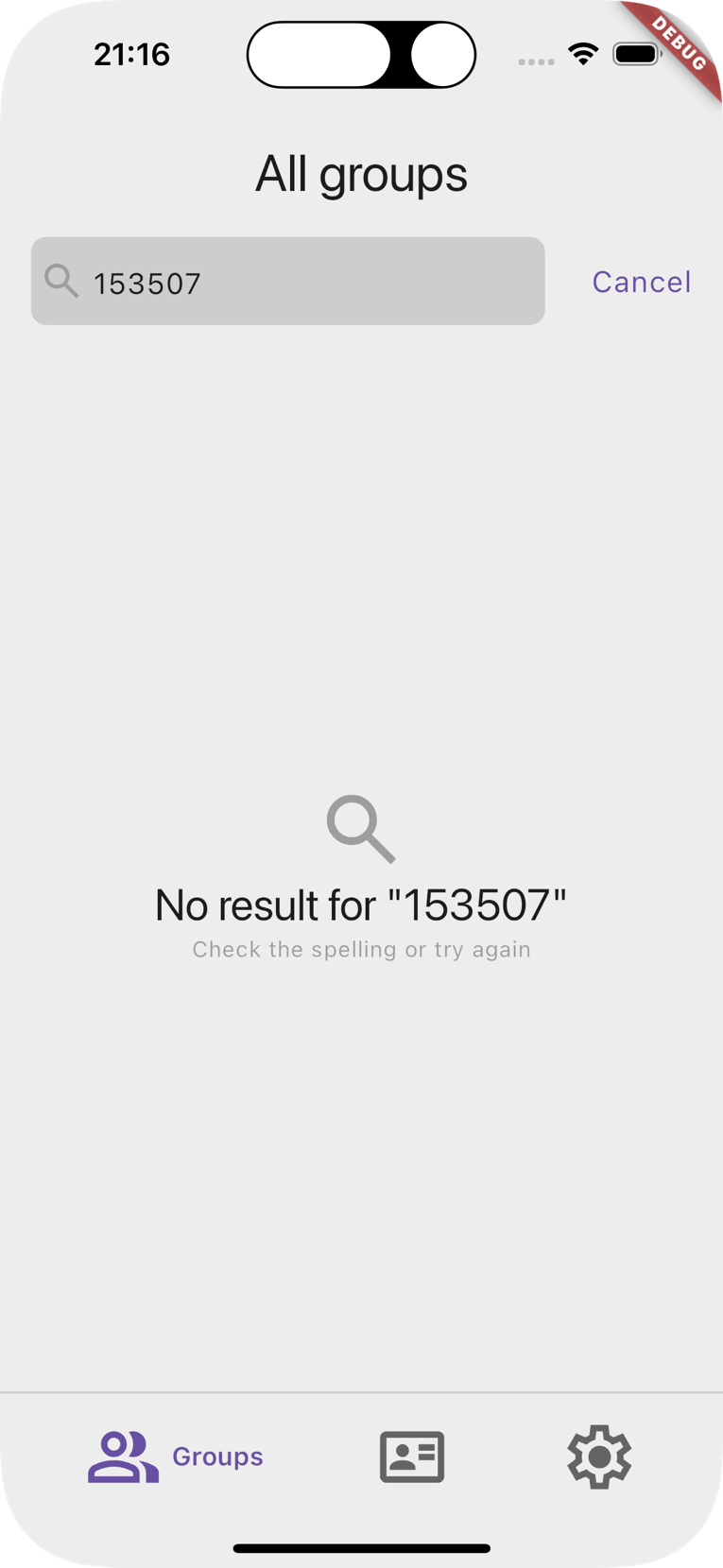


Рисунок 4.3 – Неудачный поиск группы

Экран списка преподавателей аналогичный экрану списка групп. На нём отображены ФИО преподавателя и фотография, если таковая имеется.

Для удобства просмотра преподаватели отсортированы по фамилии.

Также имеется поиск по содержанию в ФИО преподавателей введённого текста. При несоответствии введённого текста также выводится информационное сообщение об этом.

Интерфейс данного экрана представлен на рисунке 4.4.

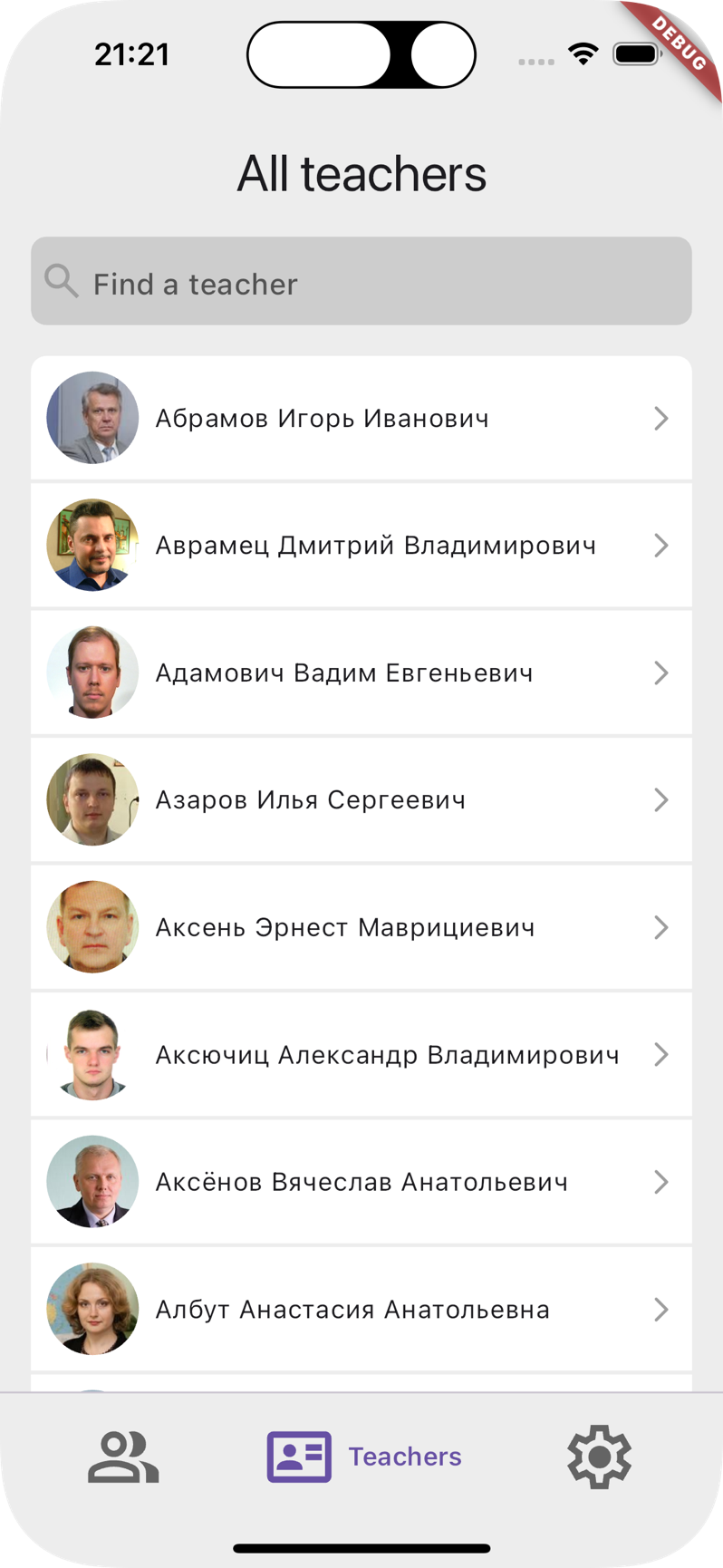


Рисунок 4.4 – Экран списка преподавателей

При нажатии на конкретную группу или преподавателя открывается экран соответствующего расписания (рисунок 4.5).

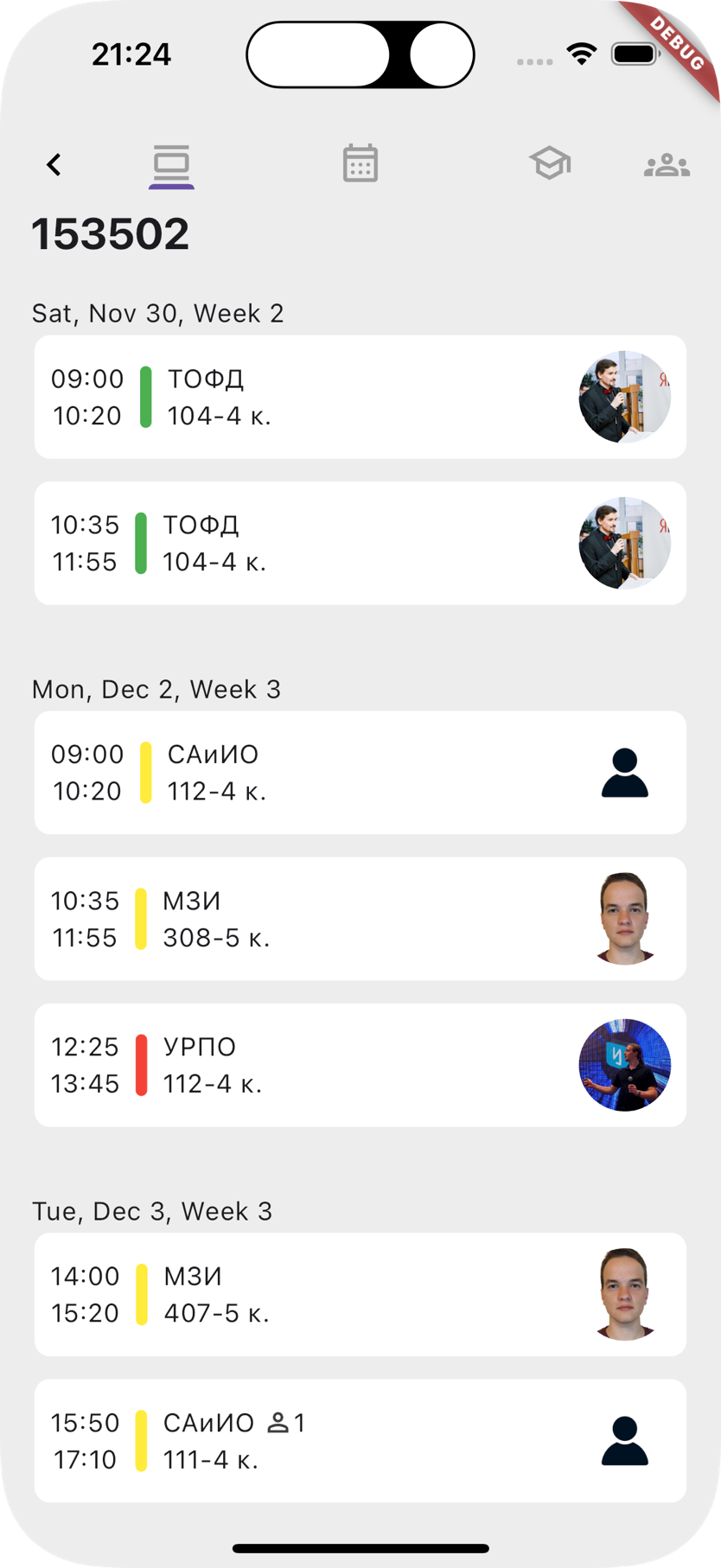


Рисунок 4.5 – Экран расписания группы

В самом верху экрана располагаются различные кнопки. При нажатии на иконки стрелочки пользователя возвращает на предыдущий экран. Следующие три иконки позволяют поменять вид отображения расписания: ежедневное расписание, всё расписание или же расписание экзаменов и консультаций. Кнопка с иконкой человека позволяет фильтровать расписание по подгруппам: только для первой подгруппы, только для второй или же расписание для всей группы (рисунок 4.6).

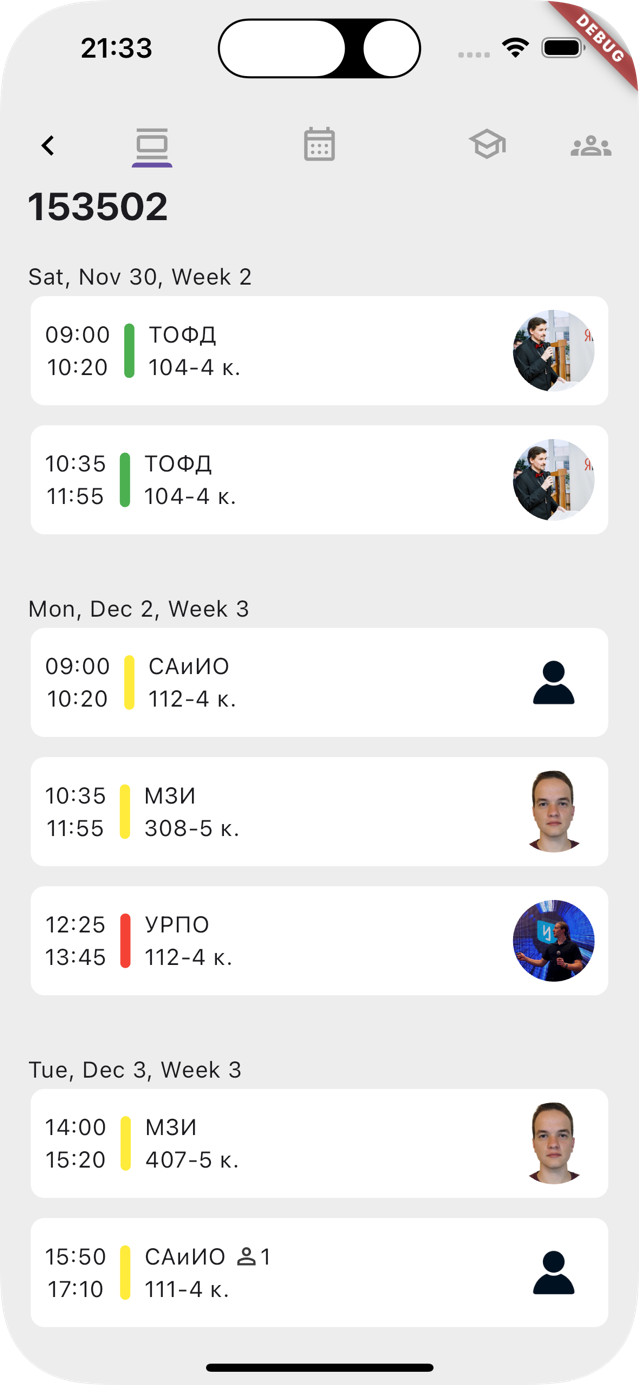


Рисунок 4.6 – Экран ежедневного расписания группы

Изначально открыта вкладка ежедневного расписания.

Для каждого дня до даты завершения семестра отображаются занятия с учётом номера недели, что позволяет посмотреть расписание на конкретный день. Занятия отсортированы по их времени начала.

Сразу же можно увидеть время начала и время окончания занятия, сокращённое название, номер аудитории, цветной маркер, обозначающий тип данного занятия, а также изображение преподавателя. Цвет маркера зависит от типа занятия: зелёный – лекция, жёлтый – лабораторное занятие, красное – практическое занятие, фиолетовый – экзамен, коричневый – консультация.

При выборе конкретной подгруппы те занятия, которые к ней не относятся, отображаются так, чтобы они были менее заметны глазу (рисунок 4.7).

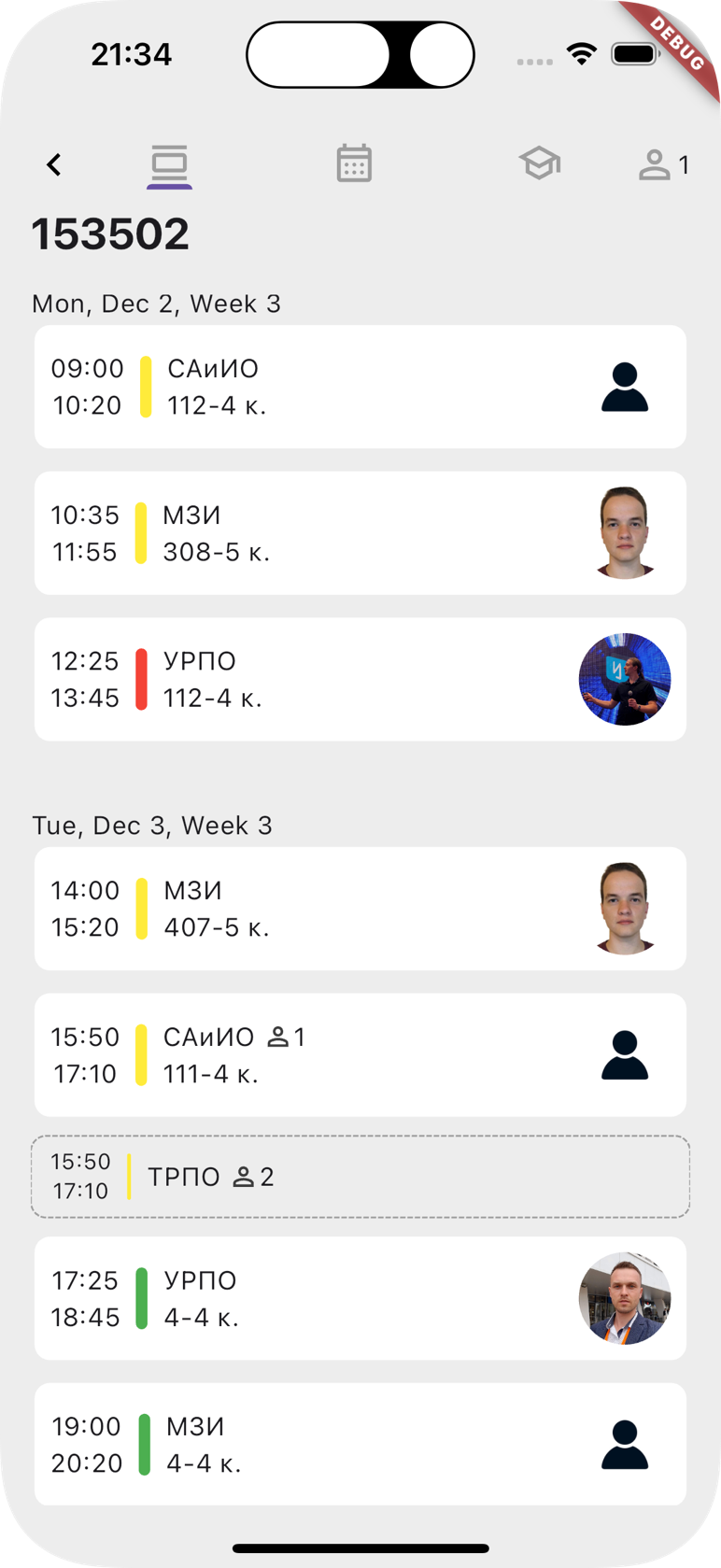


Рисунок 4.7 – Экран ежедневного расписания группы первой подгруппы

При переключении на вкладку с общим расписанием занятия отображаются все с обозначением номера недели, в которой они проводятся.

Также видны даты начала и окончания занятий в семестре.

Интерфейс данного экрана представлен на рисунке 4.8.

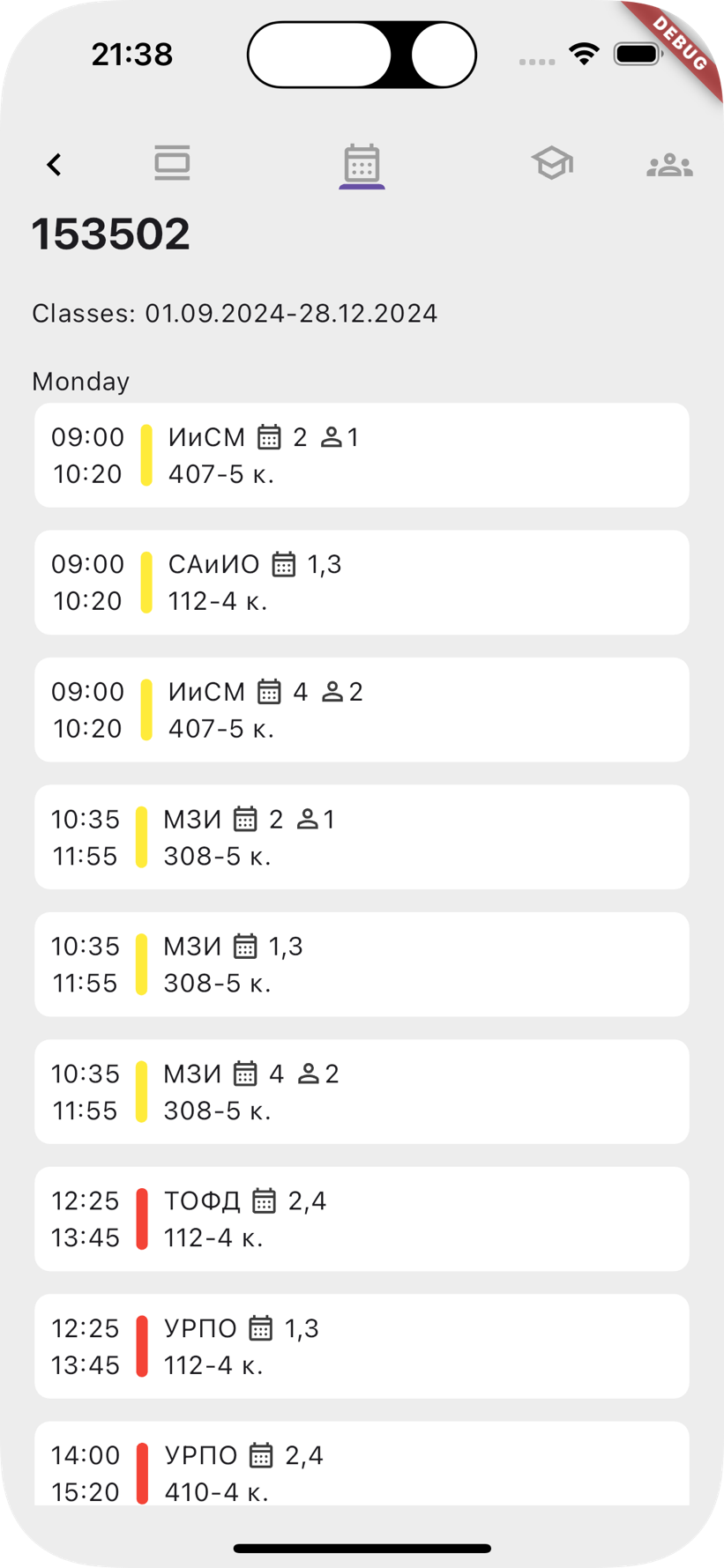


Рисунок 4.8 – Экран всего расписания группы

Экран расписания экзаменов и консультаций представлен на рисунке 4.9.

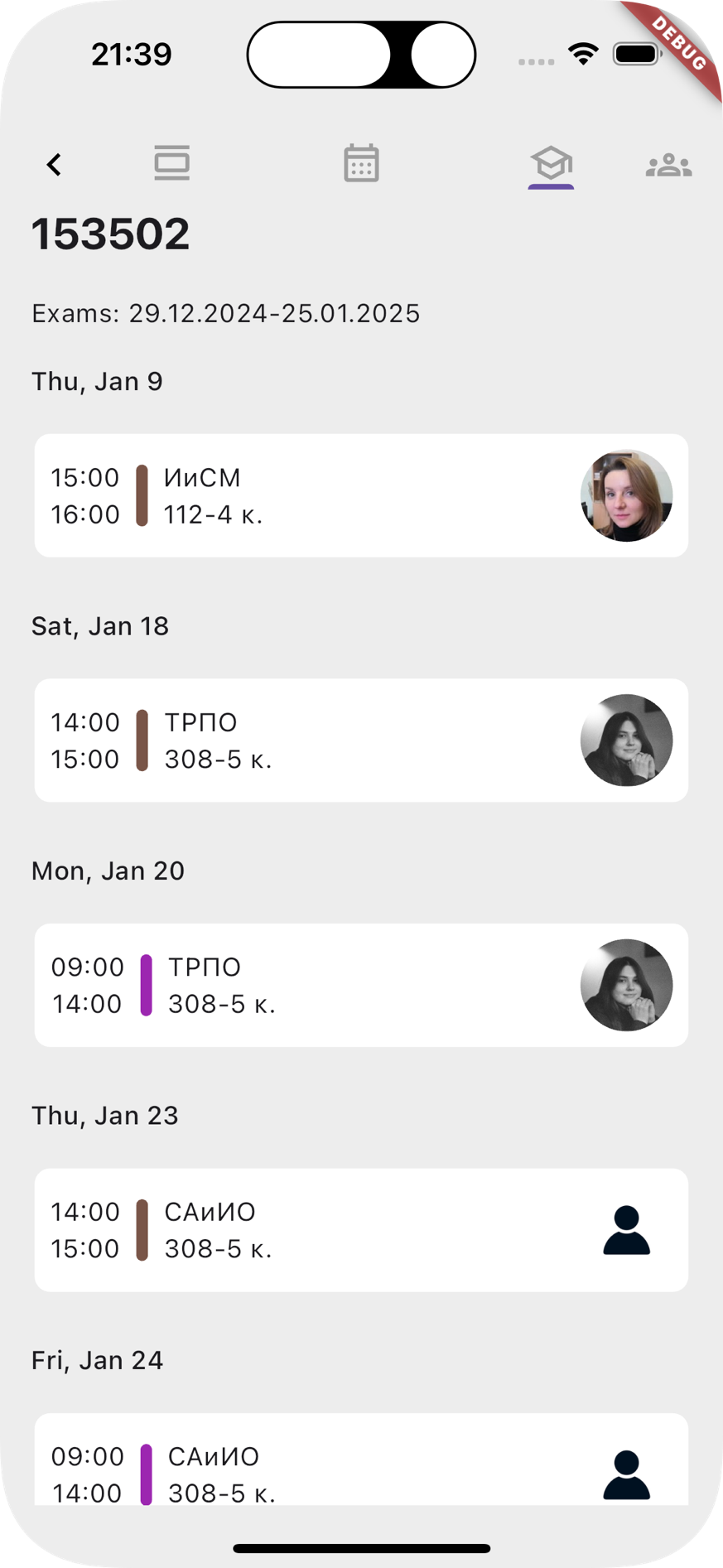


Рисунок 4.9 – Экран расписания экзаменов и консультаций группы

Экран расписания преподавателя аналогичен экрану расписания группы, только в верхней панели нет фильтрации по номеру подгруппы, а также вместо изображения преподавателя у занятия отображается группа, у которой данное занятие проводится (рисунок 4.10).

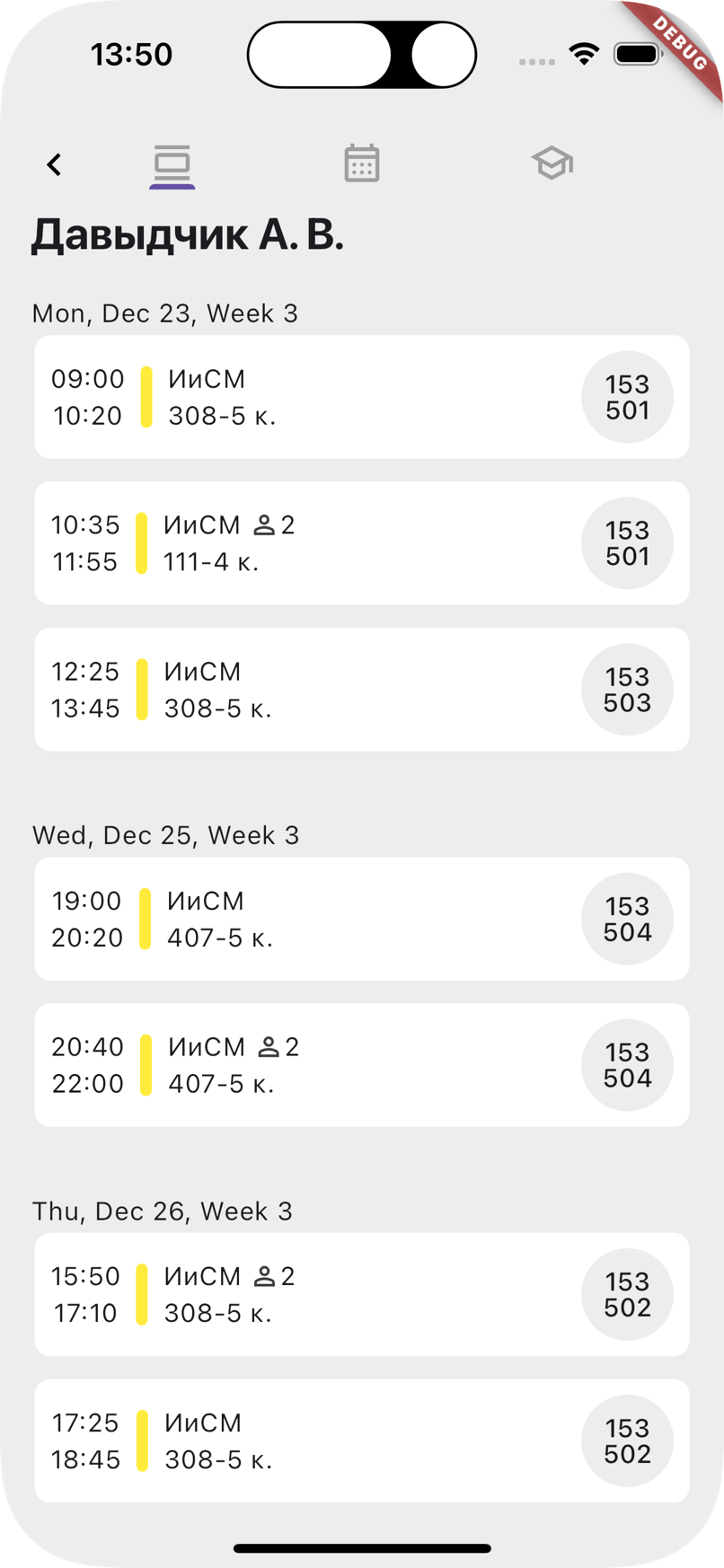


Рисунок 4.10 – Экран ежедневного расписания преподавателя

При нажатии на занятие в расписании открывается модальное окно с подробной информацией данного занятия. Если данное окно открывается из расписания группы, то на нем будет показан преподаватель, а если из расписания преподавателя – группы, у которых будет проводиться данное занятие. Также если окно открыто из вкладки расписания экзаменов, то будет отображена дата проведения экзамена или консультации. Примеры таких окон представлены на рисунках 4.11–4.13.

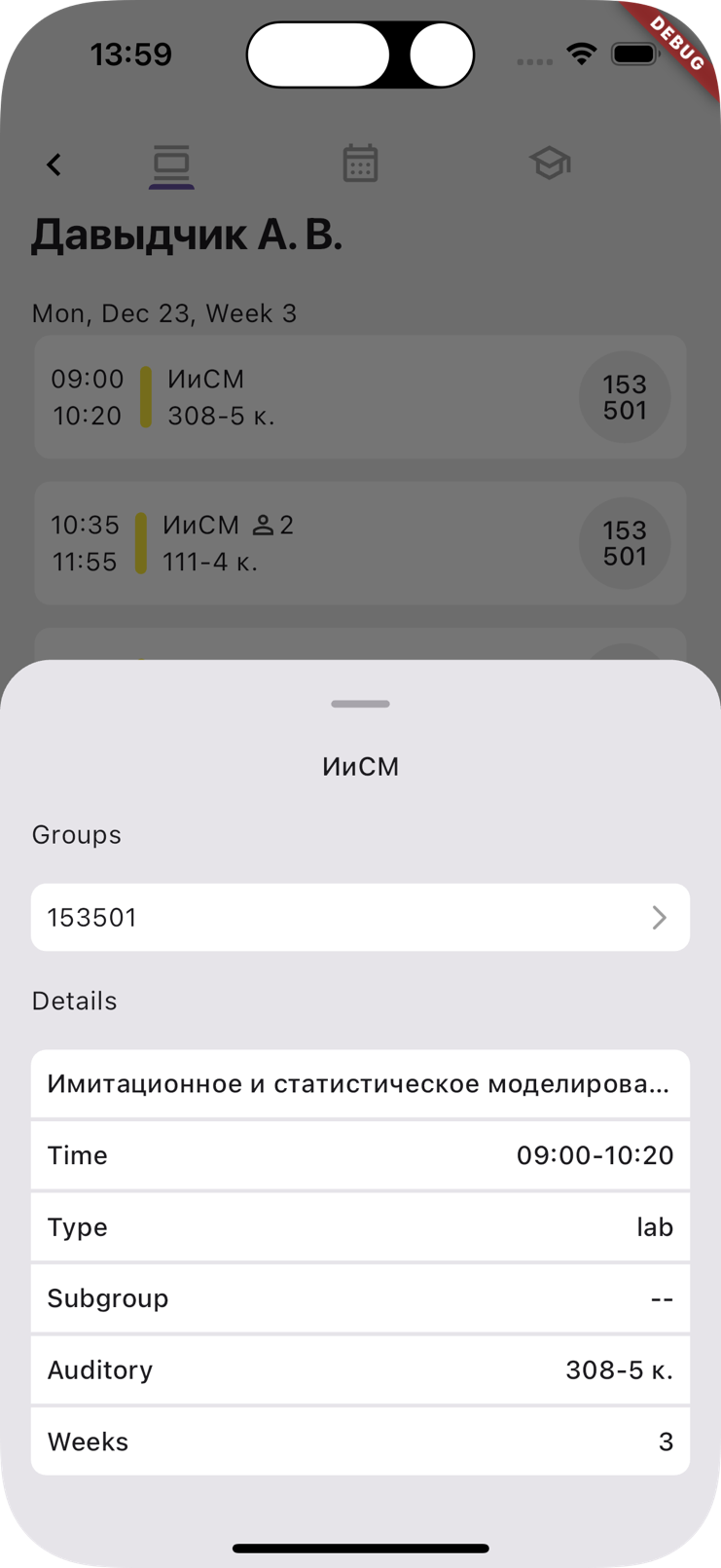


Рисунок 4.11 – Модальное окно занятия преподавателя

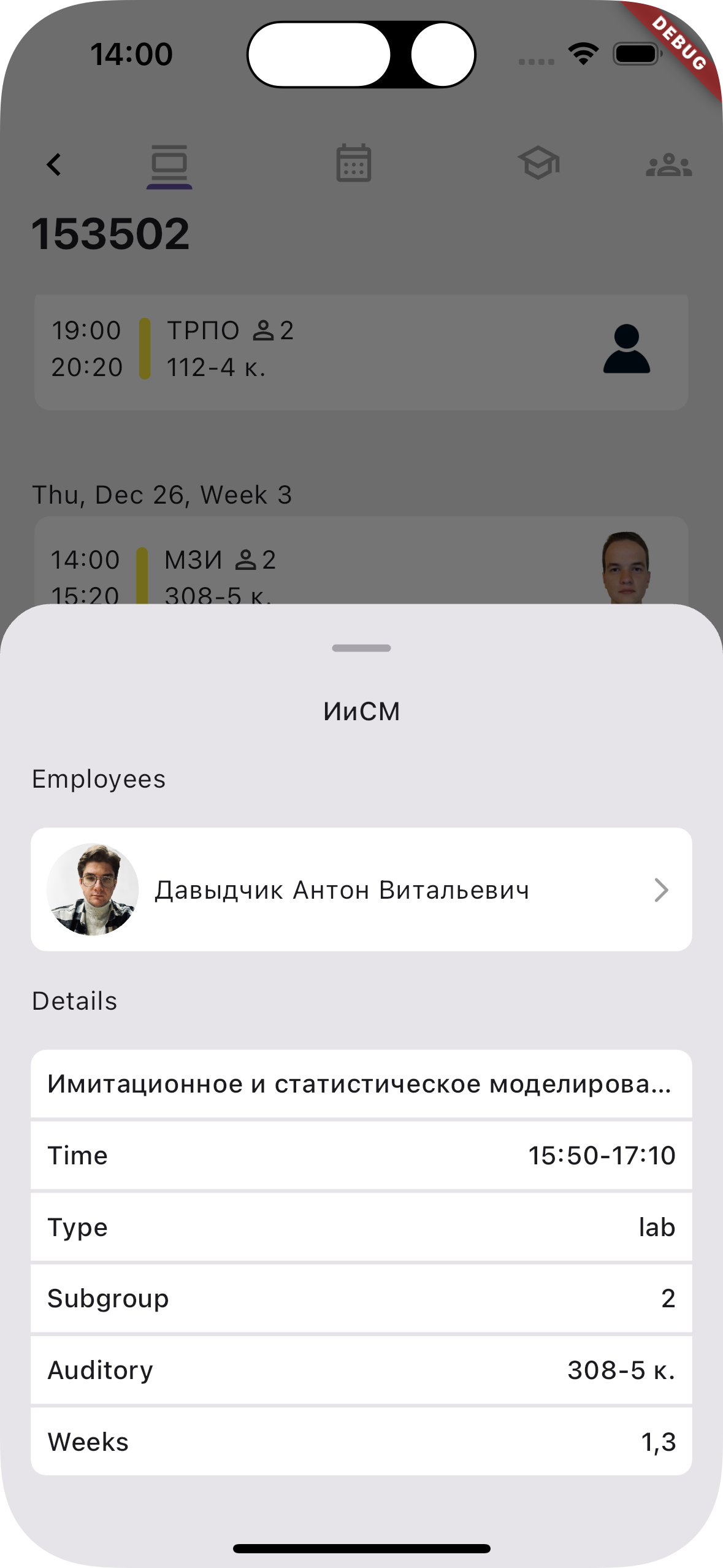


Рисунок 4.12 – Модальное окно занятия группы

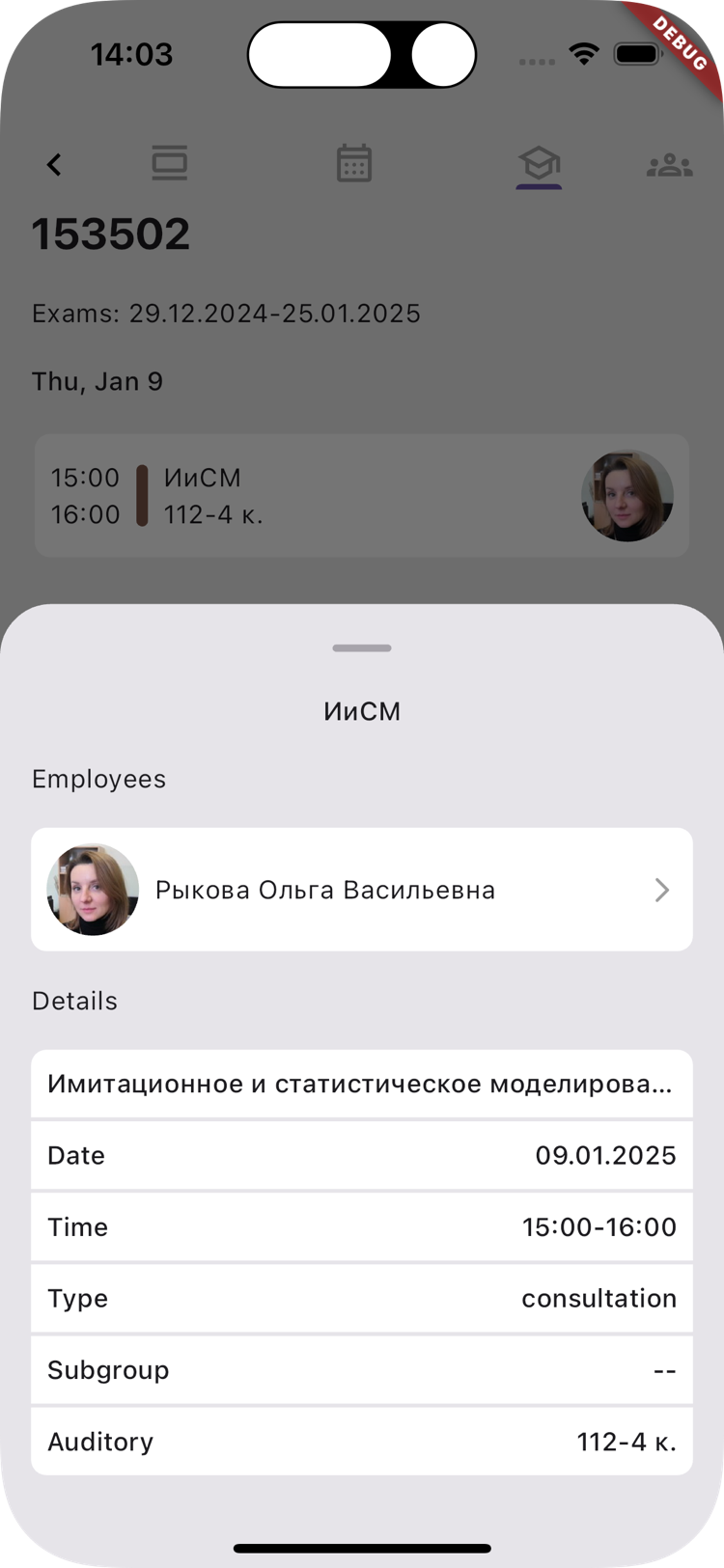


Рисунок 4.13 – Модальное окно консультации группы

При переходе на экран настроек отображается информация об аккаунте. Если пользователь еще не входил в аккаунт, то будет отображено соответствующее сообщение об этом и кнопка для того, чтобы войти в аккаунт или же зарегистрироваться (рисунок 4.14). Если же пользователь вошел в аккаунт, то отображается почта, которая привязана к аккаунту, а также кнопка для выхода из аккаунта (рисунок 4.15).

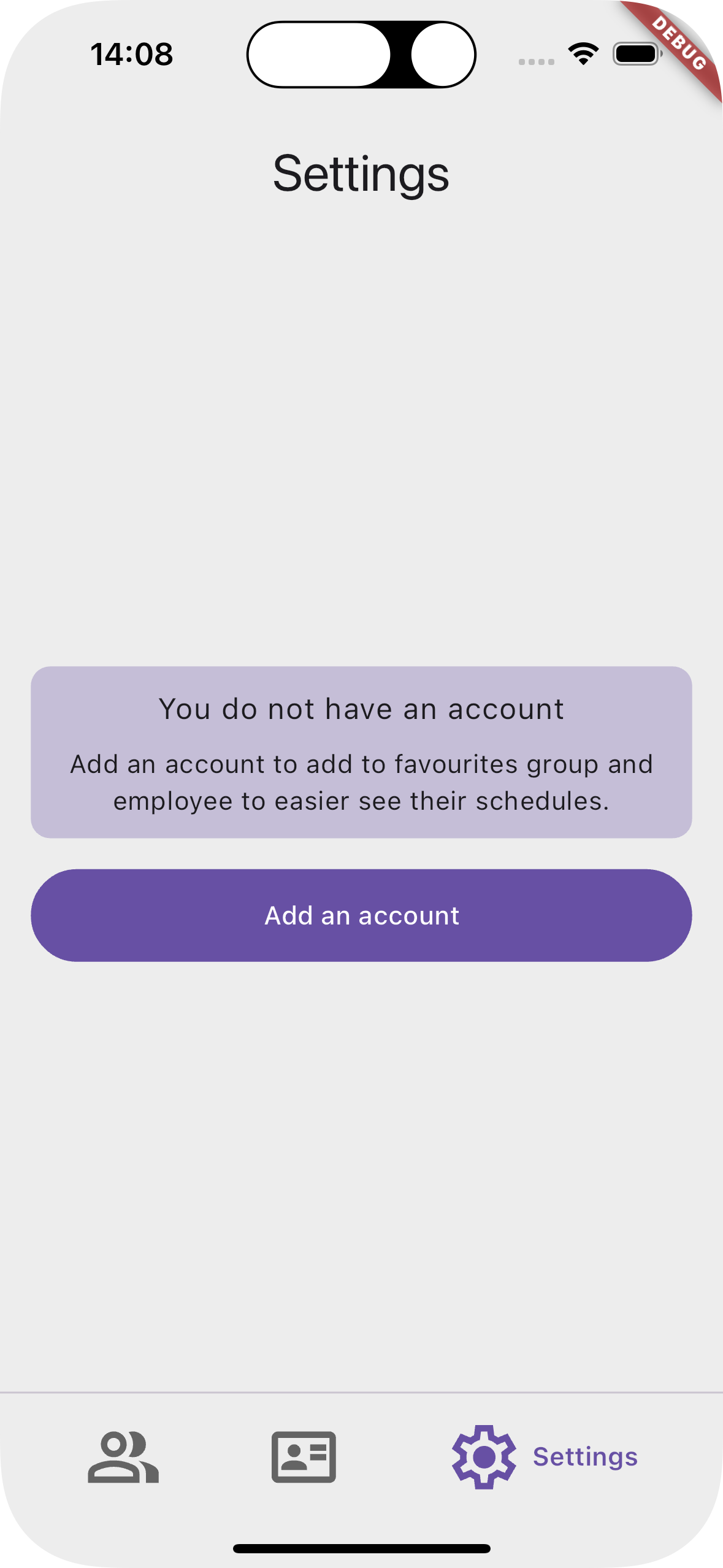


Рисунок 4.14 – Экран настроек без сохраненного аккаунта

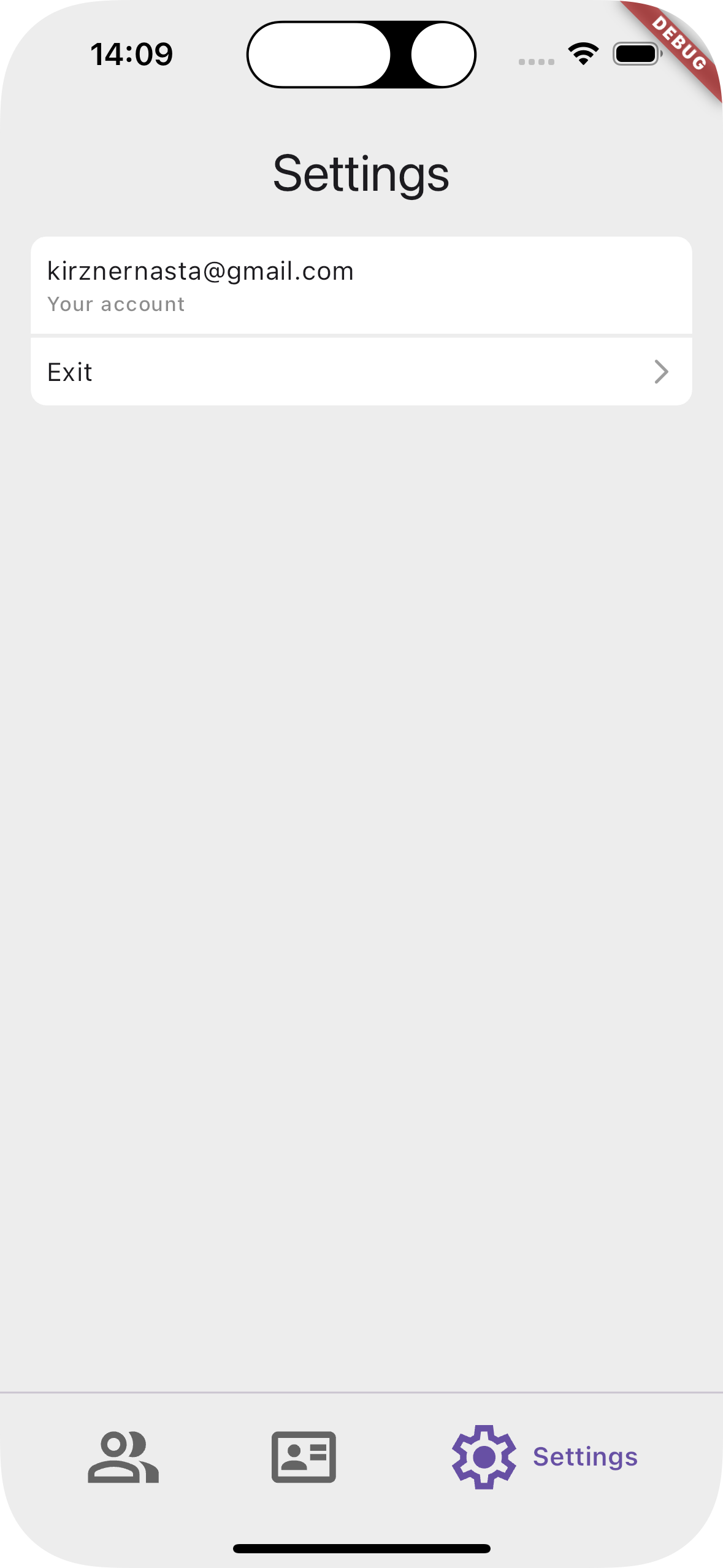


Рисунок 4.15 – Экран настроек c сохраненным аккаунтом

Если пользователь вошел в аккаунт, то главных экранах групп и преподавателей будут сначала отображаться те группы и преподаватели, добавленные в сохраненные (рисунок 4.16). Также на экране расписания в верхней панели появляется дополнительная кнопка в виде звезды для того, чтобы добавить расписание в сохраненное или же убрать его оттуда (рисунок 4.17).

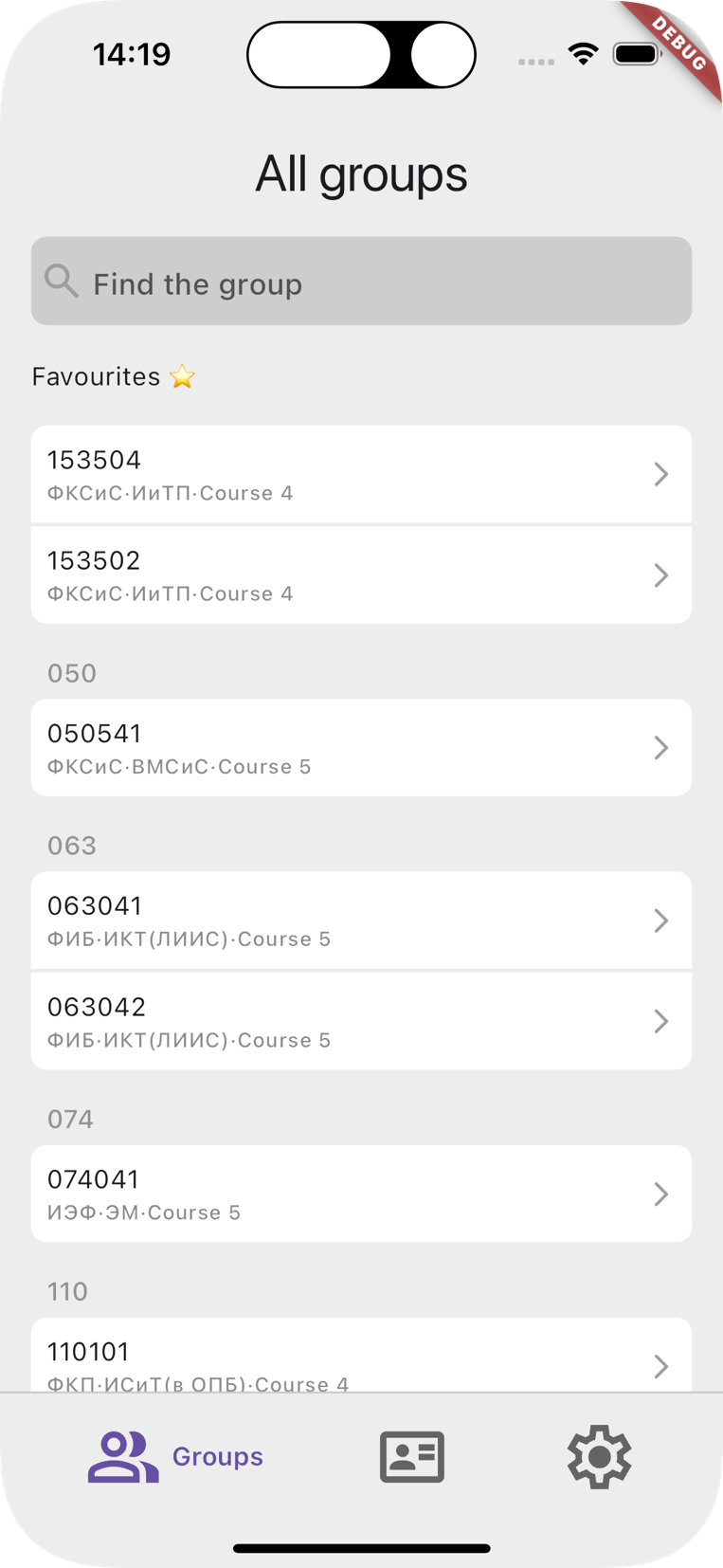


Рисунок 4.16 – Экран групп с сохраненными группами

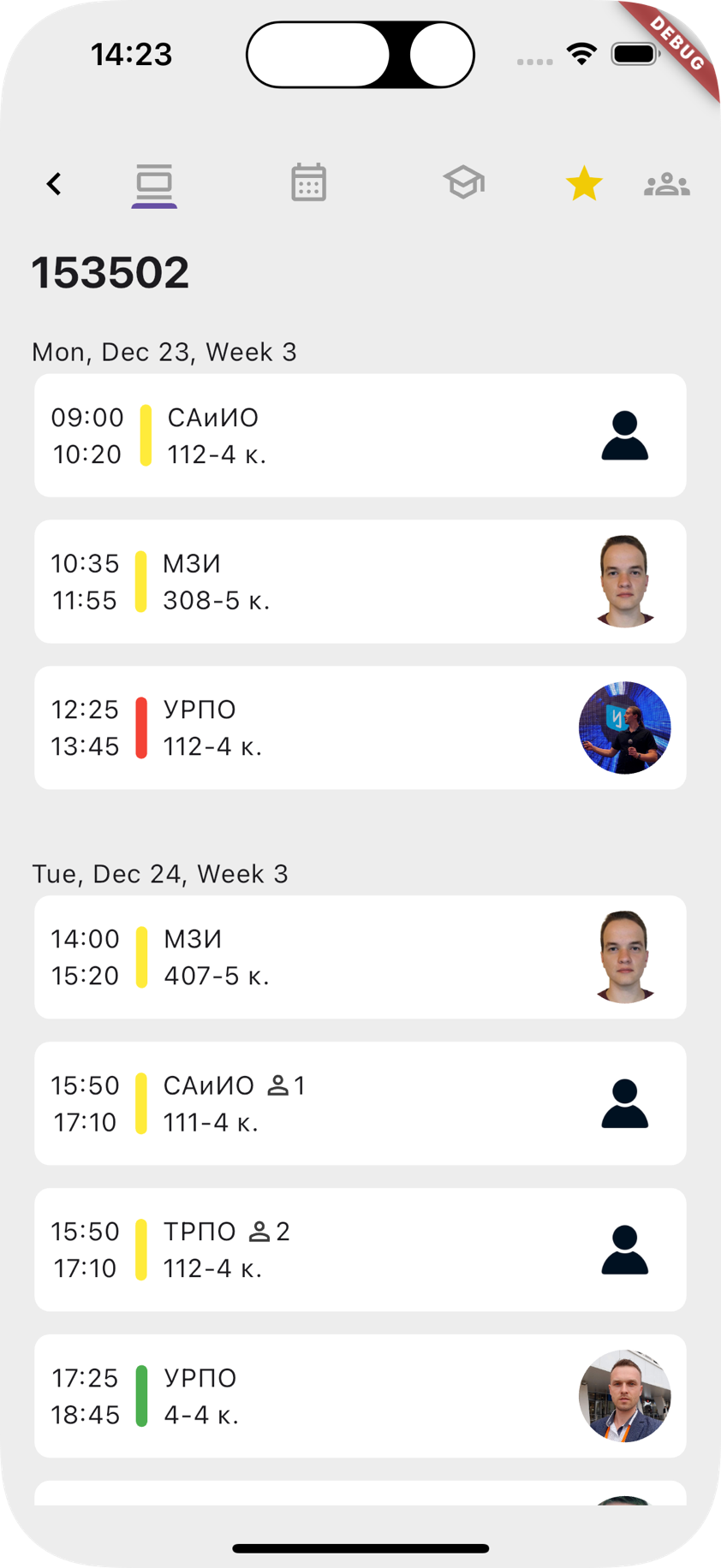


Рисунок 4.17 – Экран расписания сохраненной группы

Таким образом было реализовано мобильное приложение для *Android* и *iOS* расписания занятий БГУИР с учетом всех вышеизложенных функциональных требований.

# 5 Проектирование разрабатываемой базы данных              программного обеспечения

# Заключение

# Список литературных источников

[1] Архитектура вычислительных систем : учебное пособие / С. В. Грейбо [ и др.]. – М. : Профессиональная наука, 2019. – 77 с.

[2] Степина, В. В. Архитектура ЭВМ и вычислительные системы / В. В. Степина – М. : Курс, 2017. – 384 с.

[3] *RISC*-архитектура [Электронный ресурс]. – Режим доступа: *http*://*db*4.*sbras*.*ru*/*elbib*/*data*/*show*\_*page*.*phtml*?20+740. – Дата доступа: 30.11.2023.

[4] Архитектура *ARM* (*Advanced RISC Machine*) [Электронный ресурс]. – Режим доступа: *http*://*al*-*tm*.*ru*/*stati*/*stati*-*po*-*mat*.-*obespecheniyu*/*arm*-*advanced*-*risc*-*machine*. – Дата доступа: 30.11.2024.

[5] Общие сведения об архитектуре *ARM* [Электронный ресурс]. – Режим доступа: *https*://*osdev*.*fandom*.*com*/*ru*/. – Дата доступа: 30.11.2024.

[6] Понятие платформы программного обеспечения. Сравнительная характеристика используемых платформ (Windows, Linux и др.) [Электронный ресурс]. – Режим доступа: https://studfile.net/preview/4235646/. – Дата доступа: 30.11.2024.

[7] Что такое *Mac OS*: как она устроена и все ее версии по порядку [Электронный ресурс]. – Режим доступа: *https*://*digitalocean*.*ru*/*n*/*chto*-*takoe*-*mac*-*os*. – Дата доступа: 30.11.2024.

[8] Что такое Flutter? [Электронный ресурс]. – Режим доступа: *https://aws.amazon.com/ru/what-is/flutter/*. – Дата доступа: 30.11.2024.

[9] Dart [Электронный ресурс]. – Режим доступа: *https://blog.skillfactory.ru/glossary/dart/*. – Дата доступа: 30.11.2024.

[10] Что такое функциональные требования: примеры, определение, полное руководство [Электронный ресурс]. – Режим доступа: *https*://*visuresolutions*.*com*/*ru*/*blog*/*functional*-*requirements*/. – Дата доступа: 30.11.2024.

# Приложение а (обязательное) Листинг программного кода

**Файл *schedule*\_*api*\_*service*.*dart***

import 'package:dio/dio.dart';

import 'package:postgres/postgres.dart';

import '../data.dart';

import '../models/models.dart';

// ignore\_for\_file: avoid\_print

// ignore\_for\_file: avoid\_dynamic\_calls

class ScheduleApiService {

late final Connection \_connection;

ScheduleApiService(this.\_connection);

Future<List<StudentGroupModel>> fetchAllGroups() async {

final List<StudentGroupModel> groups = [];

try {

final result = await \_connection.execute('''

SELECT

g.id AS "groupId",

g.name AS "groupName",

g.course AS "course",

g.calendar\_id AS "calendarId",

g.education\_degree AS "educationDegree",

s.id AS "specialityId",

s.name AS "specialityName",

s.abbrev AS "specialityAbbrev",

f.id AS "facultyId",

f.name AS "facultyName",

f.abbrev AS "facultyAbbrev"

FROM

groups g

JOIN

specialities s ON g.speciality\_id = s.id

JOIN

faculties f ON s.faculty\_id = f.id

ORDER BY

g.name;

''');

for (final group in result) {

groups.add(

StudentGroupModel(

id: group[0] as int?,

name: (group[1] ?? '') as String,

course: group[2] as int?,

calendarId: group[3] as String?,

educationDegree: (group[4] ?? 1) as int,

specialityName: (group[6] ?? '') as String,

specialityAbbrev: group[7] as String?,

facultyAbbrev: group[10] as String?,

),

);

}

} on Exception catch (e) {

print('Get student groups error: $e');

}

return groups;

}

Future<void> \_insertItem(

Connection conn,

StudentGroupModel group,

ScheduleItemModel item,

int weekday,

) async {

final lessonTypeId = switch (item.lessonTypeAbbrev) {

'ЛК' => 1,

'ЛР' => 2,

'ПЗ' => 3,

'Экзамен' => 4,

'Консультация' => 5,

\_ => 6,

};

final values =

"(ARRAY[${item.auditories.map((e) => "'$e'").join(',')}], '${item.startLessonTime}', '${item.endLessonTime}', ${item.subgroupNumber}, $lessonTypeId, ARRAY[${item.weekNumbers.map((e) => e).join(',')}], ${item.dateLesson == null ? 'NULL' : "'${item.dateLesson}'"}, ${item.startLessonDate == null ? 'NULL' : "'${item.startLessonDate}'"}, ${item.endLessonDate == null ? 'NULL' : "'${item.endLessonDate}'"}, $weekday, '${item.subjectAbbreviationName}', '${item.subjectFullName}')";

final employees = item.employees;

var employeesSql = '';

if (employees != null) {

final employeeValues = employees.map((e) => '(s\_id, ${e.id})').join(',');

employeesSql = '''

INSERT INTO schedule\_employees (schedule\_id, employee\_id)

VALUES

$employeeValues

ON CONFLICT (schedule\_id, employee\_id)

DO NOTHING;

''';

}

await conn.execute('''

DO \$\$

DECLARE

s\_id INT;

BEGIN

INSERT INTO schedule (

auditories,

start\_lesson\_time,

end\_lesson\_time,

subgroup\_number,

lesson\_type\_id,

week\_number,

date\_lesson,

start\_lesson\_date,

end\_lesson\_date,

weekday,

subject\_name,

subject\_full\_name

) VALUES $values

ON CONFLICT (auditories, start\_lesson\_time, end\_lesson\_time, subgroup\_number, lesson\_type\_id, week\_number, date\_lesson, start\_lesson\_date, end\_lesson\_date, weekday, subject\_id, subject\_name, subject\_full\_name)

DO NOTHING

RETURNING id INTO s\_id;

INSERT INTO schedule\_student\_groups (schedule\_id, group\_id)

VALUES (s\_id, ${group.id})

ON CONFLICT (schedule\_id, group\_id)

DO NOTHING;

$employeesSql

END \$\$;

''');

}

Future<List<EmployeeModel>> fetchAllEmployees() async {

final List<EmployeeModel> employees = [];

try {

final result = await \_connection.execute('''

SELECT \* FROM employees;

''');

for (final employee in result) {

employees.add(

EmployeeModel(

id: (employee[0] ?? 0) as int,

firstName: (employee[1] ?? '') as String,

lastName: (employee[2] ?? '') as String,

middleName: (employee[3] ?? '') as String,

imageUrl: (employee[6] ?? '') as String,

calendarId: employee[7] as String?,

urlId: (employee[9] ?? '') as String,

fio: employee[10] as String?,

),

);

}

} on Exception catch (e) {

print('Get employees error: $e');

}

return employees;

}

Future<ScheduleModel?> fetchGroupSchedule(String groupNumber) async {

ScheduleModel? schedule;

try {

final informationResult = await \_connection.execute('''

SELECT

gs.start\_date AS "startDate",

gs.end\_date AS "endDate",

gs.start\_exams\_date AS "startExamsDate",

gs.end\_exams\_date AS "endExamsDate",

g.id AS "groupId",

g.name AS "groupName"

FROM

group\_schedule gs

JOIN

groups g ON gs.group\_id = g.id

WHERE

g.name = '$groupNumber';

''');

final exams = <ScheduleItemModel>[];

final mondayResult = await \_fetchWeekDaySchedule(groupNumber, 1);

final monday = mondayResult.$1;

exams.addAll(mondayResult.$2);

final tuesdayResult = await \_fetchWeekDaySchedule(groupNumber, 2);

final tuesday = tuesdayResult.$1;

exams.addAll(tuesdayResult.$2);

final wednesdayResult = await \_fetchWeekDaySchedule(groupNumber, 3);

final wednesday = wednesdayResult.$1;

exams.addAll(wednesdayResult.$2);

final thursdayResult = await \_fetchWeekDaySchedule(groupNumber, 4);

final thursday = thursdayResult.$1;

exams.addAll(thursdayResult.$2);

final fridayResult = await \_fetchWeekDaySchedule(groupNumber, 5);

final friday = fridayResult.$1;

exams.addAll(fridayResult.$2);

final saturdayResult = await \_fetchWeekDaySchedule(groupNumber, 6);

final saturday = saturdayResult.$1;

exams.addAll(saturdayResult.$2);

schedule = ScheduleModel(

startDate:

informationResult.elementAtOrNull(0)?.elementAt(0) as String?,

endDate: informationResult.elementAtOrNull(0)?.elementAt(1) as String?,

startExamsDate:

informationResult.elementAtOrNull(0)?.elementAt(2) as String?,

endExamsDate:

informationResult.elementAtOrNull(0)?.elementAt(3) as String?,

schedules: WeekScheduleModel(

monday: monday,

tuesday: tuesday,

wednesday: wednesday,

thursday: thursday,

friday: friday,

saturday: saturday,

),

exams: exams,

);

} on DioException catch (e) {

print('Get student group schedule error: $e');

}

return schedule;

}

Future<(List<ScheduleItemModel>, List<ScheduleItemModel>)>

\_fetchWeekDaySchedule(

String searchFilter,

int weekDay, {

bool isGroupSchedule = true,

}) async {

final weekdaySchedule = <ScheduleItemModel>[];

final exams = <ScheduleItemModel>[];

final result = await \_connection.execute('''

SELECT s.start\_lesson\_time AS "startTime",

s.end\_lesson\_time AS "endTime",

s.auditories AS "auditories",

s.week\_number AS "week\_number",

s.subgroup\_number AS "subgroupNumber",

s.date\_lesson AS "date\_lesson",

lt.abbrev AS "lessonType",

lt.full\_name AS "lessonTypeName",

sub.name AS "subjectName",

sub.full\_name AS "subjectFullName",

JSON\_AGG(DISTINCT JSONB\_BUILD\_OBJECT(

'id', e.id,

'firstName', e.first\_name,

'lastName', e.last\_name,

'middleName', e.middle\_name,

'degree', e.degree,

'rank', e.rank,

'photoLink', e.photo\_link,

'calendarId', e.calendar\_id,

'academicDepartment', e.academic\_department,

'urlId', e.url\_id,

'fio', e.fio

)) AS "employees",

JSON\_AGG(DISTINCT JSONB\_BUILD\_OBJECT(

'id', g.id,

'name', g.name,

'course', g.course,

'educationDegree', g.education\_degree

)) AS "groups"

FROM schedule s

JOIN

schedule\_student\_groups ssg ON s.id = ssg.schedule\_id

JOIN

groups g ON ssg.group\_id = g.id

JOIN

lesson\_types lt ON s.lesson\_type\_id = lt.id

JOIN

subjects sub ON s.subject\_id = sub.id

LEFT JOIN

schedule\_employees se ON s.id = se.schedule\_id

LEFT JOIN

employees e ON se.employee\_id = e.id

WHERE ${isGroupSchedule ? "g.name = " : "e.url\_id = "} '$searchFilter'

AND s.weekday = $weekDay

GROUP BY s.start\_lesson\_time,

s.end\_lesson\_time,

s.auditories,

s.subgroup\_number,

s.week\_number,

s.date\_lesson,

lt.abbrev,

lt.full\_name,

sub.name,

sub.full\_name

ORDER BY s.start\_lesson\_time;

''');

for (final item in result) {

final schedule = \_parseGroupScheduleItem(item);

if (schedule.lessonTypeAbbrev == 'Экзамен' ||

schedule.lessonTypeAbbrev == 'Консультация') {

exams.add(schedule);

} else {

weekdaySchedule.add(schedule);

}

}

return (weekdaySchedule, exams);

}

ScheduleItemModel \_parseGroupScheduleItem(ResultRow scheduleItem) {

return ScheduleItemModel(

isSplit: false,

isAnnouncement: false,

dateLesson: scheduleItem[5] as String?,

employees: \_tryParseEmployees(scheduleItem),

studentGroups: \_tryParseGroups(scheduleItem),

subgroupNumber: (scheduleItem[4] ?? 0) as int,

endLessonTime: (scheduleItem[1] ?? '') as String,

weekNumbers: (scheduleItem[3] ?? []) as List<int>,

subjectFullName: (scheduleItem[9] ?? '') as String,

startLessonTime: (scheduleItem[0] ?? '') as String,

auditories: (scheduleItem[2] ?? []) as List<String>,

lessonTypeAbbrev: (scheduleItem[6] ?? '') as String,

subjectAbbreviationName: (scheduleItem[8] ?? '') as String,

);

}

List<EmployeeModel> \_tryParseEmployees(ResultRow scheduleItem) {

final employees = <EmployeeModel>[];

final employeesInfo = scheduleItem[10] as List<dynamic>?;

if (employeesInfo == null) return employees;

for (final employee in employeesInfo) {

employees.add(

EmployeeModel(

id: employee['id'] as int,

fio: employee['fio'] as String?,

urlId: employee['urlId'] as String,

lastName: employee['lastName'] as String,

imageUrl: employee['photoLink'] as String,

firstName: employee['firstName'] as String,

middleName: employee['middleName'] as String?,

calendarId: employee['calendarId'] as String?,

),

);

}

return employees;

}

List<StudentGroupModel> \_tryParseGroups(ResultRow scheduleItem) {

final groups = <StudentGroupModel>[];

final groupsInfo = scheduleItem[11] as List<dynamic>?;

if (groupsInfo == null) return groups;

for (final group in groupsInfo) {

groups.add(

StudentGroupModel(

name: group['name'] as String,

specialityName: '',

educationDegree: 0,

),

);

}

return groups;

}

Future<ScheduleModel?> fetchEmployeeSchedule(String urlId) async {

ScheduleModel? schedule;

try {

final informationResult = await \_connection.execute('''

SELECT

es.start\_date AS "startDate",

es.end\_date AS "endDate",

es.start\_exams\_date AS "startExamsDate",

es.end\_exams\_date AS "endExamsDate",

e.id AS "employeeId",

e.url\_id AS "employeeUrlId"

FROM

employee\_schedule es

JOIN

employees e ON es.employee\_id = e.id

WHERE

e.url\_id = '$urlId';

''');

final exams = <ScheduleItemModel>[];

final mondayResult = await \_fetchWeekDaySchedule(

urlId,

1,

isGroupSchedule: false,

);

final monday = mondayResult.$1;

exams.addAll(mondayResult.$2);

final tuesdayResult = await \_fetchWeekDaySchedule(

urlId,

2,

isGroupSchedule: false,

);

final tuesday = tuesdayResult.$1;

exams.addAll(tuesdayResult.$2);

final wednesdayResult = await \_fetchWeekDaySchedule(

urlId,

3,

isGroupSchedule: false,

);

final wednesday = wednesdayResult.$1;

exams.addAll(wednesdayResult.$2);

final thursdayResult = await \_fetchWeekDaySchedule(

urlId,

4,

isGroupSchedule: false,

);

final thursday = thursdayResult.$1;

exams.addAll(thursdayResult.$2);

final fridayResult = await \_fetchWeekDaySchedule(

urlId,

5,

isGroupSchedule: false,

);

final friday = fridayResult.$1;

exams.addAll(fridayResult.$2);

final saturdayResult = await \_fetchWeekDaySchedule(

urlId,

6,

isGroupSchedule: false,

);

final saturday = saturdayResult.$1;

exams.addAll(saturdayResult.$2);

schedule = ScheduleModel(

startDate:

informationResult.elementAtOrNull(0)?.elementAt(0) as String?,

endDate: informationResult.elementAtOrNull(0)?.elementAt(1) as String?,

startExamsDate:

informationResult.elementAtOrNull(0)?.elementAt(2) as String?,

endExamsDate:

informationResult.elementAtOrNull(0)?.elementAt(3) as String?,

schedules: WeekScheduleModel(

monday: monday,

tuesday: tuesday,

wednesday: wednesday,

thursday: thursday,

friday: friday,

saturday: saturday,

),

exams: exams,

);

} on DioException catch (e) {

print('Get employee schedule error: $e');

}

return schedule;

}

Future<bool> createUser({

required String email,

required String passwordHash,

}) async {

try {

final result = await \_connection.execute('''

INSERT INTO users (email, password\_hash) VALUES ('$email', '$passwordHash');

''');

return true;

} catch (e) {

return false;

}

}

Future<bool> findUser({

required String email,

required String passwordHash,

}) async {

try {

final result = await \_connection.execute('''

SELECT id FROM users

WHERE email = '$email' and password\_hash = '$passwordHash';

''');

if (result.isEmpty) {

return false;

}

return true;

} catch (e) {

return false;

}

}

Future<List<int>> fetchFavouriteGroupsIds({

required String email,

required String passwordHash,

}) async {

final groupIds = <int>[];

try {

final result = await \_connection.execute('''

SELECT ug.group\_id

FROM users u

JOIN user\_groups ug ON u.id = ug.user\_id

WHERE u.email = '$email'

AND u.password\_hash = '$passwordHash';

''');

for (final id in result) {

groupIds.add(id[0] as int);

}

} catch (e) {

print('Fetch favourite group ids error: $e');

}

return groupIds;

}

Future<void> addGroupToFavourite({

required int id,

required String email,

required String passwordHash,

}) async {

try {

await \_connection.execute('''

SELECT add\_user\_group\_relation('$email', '$passwordHash', $id);

''');

} catch (e) {

print('Add favourite group error: $e');

}

}

Future<void> deleteGroupFromFavourites({

required int id,

required String email,

required String passwordHash,

}) async {

try {

await \_connection.execute('''

SELECT delete\_user\_group\_relation('$email', '$passwordHash', $id);

''');

} catch (e) {

print('Delete favourite group error: $e');

}

}

Future<List<int>> fetchFavouriteEmployeesIds({

required String email,

required String passwordHash,

}) async {

final employeesIds = <int>[];

try {

final result = await \_connection.execute('''

SELECT ue.employee\_id

FROM users u

JOIN user\_employees ue ON u.id = ue.user\_id

WHERE u.email = '$email'

AND u.password\_hash = '$passwordHash';

''');

for (final id in result) {

employeesIds.add(id[0] as int);

}

} catch (e) {

print('Fetch favourite employees ids error: $e');

}

return employeesIds;

}

Future<void> addEmployeeToFavourite({

required int id,

required String email,

required String passwordHash,

}) async {

try {

await \_connection.execute('''

SELECT add\_user\_employee\_relation('$email', '$passwordHash', $id);

''');

} catch (e) {

print('Add favourite employee error: $e');

}

}

Future<void> deleteEmployeeFromFavourites({

required int id,

required String email,

required String passwordHash,

}) async {

try {

await \_connection.execute('''

SELECT delete\_user\_employee\_relation('$email', '$passwordHash', $id);

''');

} catch (e) {

print('Delete favourite employee error: $e');

}

}

}

# Приложение Б (обязательное) Конечная схема базы данных

# Приложение В (обязательное) Ведомость документов