Министерство науки и высшего образования Российской Федерации
Байкальский государственный университет

А.В. Родионов

КРОССПЛАТФОРМЕННЫЕ ИНСТРУМЕНТАЛЬНЫЕ СИСТЕМЫ: РАЗРАБОТКА ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ XAMARIN.FORMS

Учебное пособие

В трех частях

Часть 1

Иркутск
Издательский дом БГУ
2023

УДК 004.4
ББК 32.973
Р60

Издается по решению редакционно-издательского совета
Байкальского государственного университета

Рецензенты
канд. техн. наук, доц. Т.И. Ведерникова (БГУ)
канд. техн. наук, доц. З.А. Бахвалова (ИРНИТУ)

Родионов, А.В.
Р60 Кроссплатформенные инструментальные системы: разработка приложений с использованием Xamarin.Forms : учеб. пособие. В 3 ч. / А.В. Родионов. – Иркутск : Изд. дом БГУ, 2023.

ISBN 978-5-7253-3135-6.
Ч. 1. – 142 с.
ISBN 978-5-7253-3138-7.

В учебном пособии рассматриваются вопросы разработки кроссплатформенных программных решений, дается обзор инструментов кроссплатформенной разработки, описываются область их применения, достоинства и недостатки. Детально исследуются фреймворк Xamarin.Forms, язык разметки XAML, принципы построения графического интерфейса кроссплатформенного приложения с использованием Xamarin.Forms, взаимодействие с кодом.

Пособие предназначено для студентов специальностей 09.04.03 Прикладная информатика (магистратура), 38.03.05 Бизнес-информатика (бакалавриат), 09.03.03 Прикладная информатика (бакалавриат) и аспирантов. Оно также будет полезно для специалистов, связанных с современными информационными технологиями, и для широкого круга читателей, интересующихся разработкой программного обеспечения.

УДК 004.4
ББК 32.973

ISBN 978-5-7253-3138-7 (ч. 1)
ISBN 978-5-7253-3135-6

© Родионов А.В., 2023
© ФГБОУ ВО «БГУ», 2023

ОГЛАВЛЕНИЕ

Введение 5
1. Введение в кроссплатформенную разработку 7
1.1. Аппаратная и программная платформы 7
1.1.1. Аппаратные платформы 7
1.1.2. Программные платформы 8
1.2. Мобильные устройства и проблема фрагментации 11
1.3. Нативная и кроссплатформенная разработка 12
Контрольные вопросы 16
2. Фреймворк для кроссплатформенной разработки мобильных приложений Xamarin.Forms 18
2.1. Основы Xamarin 18
2.1.1. Обзор и основные компоненты платформы 18
2.1.2. Методы создания общего кода 20
2.1.3. Поддерживаемые платформы и платформозависимый код 23
2.1.4. Класс Application и жизненный цикл приложения 25
2.2. Проект «Мобильное приложение (Xamarin.Forms)» 26
2.2.1. Создание проекта и интерфейс среды разработки 26
2.2.2. Структура проекта Xamarin 29
2.2.3. Отладка проекта 30
2.3. Язык eXtensible Application Markup Language 33
2.3.1. Основы eXtensible Application Markup Language 33
2.3.2. Структура файла eXtensible Application Markup Language 33
2.3.3. Элементы свойств 36
2.3.4. Присоединенные свойства 37
2.3.5. Позиционирование элементов 38
2.3.6. Выравнивание элементов 39
2.3.7. Взаимодействие с кодом приложения 40
Контрольные вопросы 43
3. Графический интерфейс в Xamarin.Forms 45
3.1. Основные элементы управления 45
3.1.1. Элементы для работы с текстом 45
3.1.2. Элементы для работы с числовыми значениями 51
3.1.3. Элементы оформления 53
3.1.4. Элементы выбора значений 54
3.1.5. Элементы для работы с датой и временем 60
3.1.6. Мультимедиаэлементы 62
3.1.7. Элементы – индикаторы процесса выполнения 67
3.1.8. Прокрутка элементов 69
3.1.9. Элементы для отображения коллекций 70
3.2. Макеты страниц и элементы компоновки 79
3.2.1. Введение в компоновку 79
3.2.2. Элемент Grid 81

3.2.3. Элемент StackLayout 84
3.2.4. Элемент AbsoluteLayout 85
3.2.5. Элемент RelativeLayout 88
3.2.6. Элемент FlexLayout 91
3.3. Многостраничные приложения в Xamarin.Forms 100
3.3.1. Навигация между страницами 100
3.3.2. Типы страниц в Xamarin.Forms 116
3.3.3. Xamarin.Forms Shell 123
Контрольные вопросы 139
Список рекомендуемой литературы 141

ВВЕДЕНИЕ

В процессе разработки приложений разработчики часто сталкиваются с необходимостью обеспечить работоспособность приложения на разных аппаратно-программных платформах. Данная задача приобрела особую актуальность в последнее десятилетие в связи с насыщением рынка мобильными устройствами, в первую очередь смартфонами.

Согласно similarweb[1], доля трафика интернета с мобильных устройств в сравнении с настольными компьютерами и планшетами за 2022 г. составила более 65 % для всего мира, и более 58 % для России. Это косвенно подчеркивает факт того, что пользователи чаще используют не персональные компьютеры, а смартфоны и иные переносные устройства.

Для разработчиков программного обеспечения это означает, что нельзя делать ставку только на разработку приложений для настольных компьютеров – приложения должны работать на любых, в том числе переносных устройствах, которые можно брать с собой. И в этом кроется проблема, так как на IT-рынке всегда была конкуренция, и каждый производитель предоставлял свой набор инструментов для разработки приложений под свои операционные системы и устройства.

Использование штатных, «родных» инструментов для разработки программного обеспечения обозначается термином «нативная разработка». С одной стороны, нативная разработка обычно позволяет создать оптимальное с точки зрения быстродействия, качества и функциональности приложение, но в отсутствие одной, «тотально доминирующей» платформы, применение только нативной разработки приводит к необходимости создания отдельных команд разработчиков на каждую платформу, потому что нативные инструменты обычно не несовместимы друг с другом на уровне языков разработки, принятых соглашений и архитектур, механизмов работы с операционной системой и т.п. Все это приводит к большим расходам как на саму разработку, так и на дальнейшее сопровождение программного обеспечения.

Решение этой проблемы заключается в использовании кроссплатформенных инструментов разработки. Кроссплатформенные приложения разрабатываются на каком-либо одном языке программирования, который не является «родным» для всех платформ. Сегодня существует огромное множество инструментов, которые позволяют разработчикам использовать один и тот же повторяющийся код для создания приложений, предназначенных для различных платформ.

В учебном пособии рассматриваются подходы к кроссплатформенной разработке, их достоинства и недостатки, инструмент Xamarin.Forms, приводятся примеры исходного кода для решения типовых задач при разработке кроссплат-

[1] Сервис анализа трафика Similarweb. URL: https://www.similarweb.com/ru/platforms.

форменных приложений. Освоение и закрепление изучаемого материала студентами происходит с помощью разбора исходного кода примеров и контрольных вопросов.

1. ВВЕДЕНИЕ В КРОССПЛАТФОРМЕННУЮ РАЗРАБОТКУ

1.1. Аппаратная и программная платформы

Термин платформа может применяться к разным уровням абстракции, включая определенную аппаратную архитектуру, операционную систему (ОС) или библиотеку времени выполнения.

1.1.1. Аппаратные платформы

Под аппаратной платформой подразумевается конкретный набор комплектующих, лежащих в основе конечного устройства[2]. Поскольку в современных устройствах остро стоит проблема миниатюризации компонент и сокращения их энергопотребления, в настоящий момент в подавляющем числе случаев индивидуальные аппаратные модули – процессор, контроллеры периферийных устройств и сами устройства – собираются на одном физическом чипе, образуя так называемую System on Chip (SoC). В пределах одного поколения такого чипа его структура и возможности фиксированы, но могут отличаться некоторые параметры, например рабочая частота, количество вычислительных ядер и т.п. Среди основных процессорных архитектур, которые используются мобильными устройствами, можно выделить следующие:

1. *Архитектура ARM.* Разработана ARM Limited в 1983 г. как основа для простого и эффективного процессора[3]. Набор команд базируется на 32-битной и 64-битной RISC-архитектуре. Возможно использование дополнительных аппаратных расширений для работы с плавающей точкой и SIMD-операциями. Интересной особенностью является то, что ARM Limited не производят готовые чипы самостоятельно, а лишь проектируют процессорные ядра и лицензируют их сторонним производителям. Это выгодно сказывается на цене и возможностях по интеграции процессорного ядра с другими устройствами. Последние десять лет архитектура ARM занимает доминирующее положение на рынке мобильных устройств.

2. *Архитектура MIPS.* MIPS была представлена MIPS Technologies в 1981 г. MIPS базируется на RISC-наборе команд и позволяет оперировать 31 регистром[4]. В последних ревизиях добавлена поддержка 64-битных инструкций. Также возможна опциональная поддержка FPU и SIMD-операций. Как и ARM, архитектура лицензируется сторонним производителям чипов. В настоящее время архитектура MIPS применяется в основном во встраиваемых устройствах, смартфонах, маршрутизаторах, шлюзах.

3. *Архитектура x86/x64*. x86 разрабатывается компанией Intel с 1978 г.[5] За время своего существования она развилась от 16-битного процессора, работающего на частоте 5 МГц, до 64-битных многоядерных систем с частотами до

[2] Обзор мобильных платформ. URL: https://www.securitylab.ru/analytics/430488.php.

[3] ARM architecture family. URL: https://en.wikipedia.org/wiki/ARM_architecture_family.

[4] MIPS architecture. URL: https://en.wikipedia.org/wiki/MIPS_architecture.

[5] x86. URL: https://en.wikipedia.org/wiki/X86.

4 ГГц, многомегабайтным встроенным кэшем и всеми передовыми расширениями для обработки данных. Архитектура использует CISC-набор команд, что ведет к возникновению дополнительных сложностей при написании кода и разработке оптимизирующих компиляторов под такую архитектуру. Процессоры на основе архитектуры x86/x64 производятся довольно ограниченным числом компаний – Intel, AMD, VIA (Zhaoxin). Архитектура отдельно не лицензируется, а поставляется только в виде готовых изделий. Это негативно сказывается на цене и интеграции процессорных ядер в другие системы. Архитектура x86/x64 доминирует на рынке персональных компьютеров, однако она может встречаться на некоторых моделях высокопроизводительных планшетов.

1.1.2. Программные платформы

Под программной платформой подразумевается основная ОС, выполняемая на данной аппаратной платформе или процессорной архитектуре. Некоторые программные платформы поддерживают различные архитектуры на уровне отдельных сборок ядра. Программное обеспечение (ПО) под одну из платформ обычно невозможно напрямую запустить ни на другой программной платформе, ни на другой процессорной архитектуре этой же программной платформы.

Среди основных ОС, которые используются в мобильных устройствах, можно выделить две – Android и iOS.

Система Android разрабатывается компанией Google с 2005 г. (хотя, можно сказать, с 2003 г., так как Android изначально создавался как ОС для цифровых фотокамер, но вскоре акцент сместился на мобильные телефоны из-за их большой распространенности на рынке). Платформа Android была представлена в 2007 г. и вышла на рынок на следующий год. Поначалу ей мешал ограниченный набор функций и небольшая база пользователей по сравнению с конкурентами (Symbian и Windows). Однако возможность обновления стала самым большим преимуществом этой ОС, поскольку каждое обновление давало новые функции и улучшенную производительность[6]. Из-за «сладости, которую они приносят в нашу жизнь», первые версии были названы в честь десертов в алфавитном порядке, например Cupcake, Jellybean и KitKat. Начиная с 2019 г. новые версии ОС получают номера, первой «пронумерованной» версией стал Android 10. Лицензия с открытым исходным кодом также помогла увеличить популярность этой ОС среди производителей мобильных устройств, поскольку они могут теперь модифицировать ОС под свои требования, не влияя при этом на разработку приложений.

В основе ОС лежит ядро Linux, хотя на текущий момент Android имеет мало общего с обычной ОС Linux. Пользовательские приложения исполняются на модифицированной версии виртуальной машины Java. Для критичных к скорости выполнения участков допускается использование внешних модулей под

[6] Чистяков В. Операционная система Android. URL: https://medium.com/nuances-of-programming/%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F-%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0-android-826fb74c5af9.

конкретную процессорную архитектуру. Таким образом, прикладные приложения не привязаны к конкретной процессорной архитектуре. Исходный код системы полностью открыт. Поддерживаемые архитектуры: ARM, MIPS, x86/x64[7].

iOS (ранее iPhone OS) разработана компанией Apple в 2007 г. для использования в их смартфоне iPhone и плеере iPod Touch. Ядро системы основано на ОС Darwin (строго говоря, взято ядро XNU, появившееся на свет в результате слияния микроядра Mach и компонентов ядра FreeBSD). Данная ОС широко популяризовала идеологию touch-интерфейса и использование сразу нескольких одновременных касаний экрана. Все приложения изолированы друг от друга и системы, а запустить их на немодифицированном устройстве можно только скачав из официального интернет-магазина Apple iTunes. Для размещения приложения в магазине необходимо приобрести лицензию разработчика и для каждого приложения пройти обязательную процедуру сертификации. Сторонним производителям устройств iOS недоступна. Поддерживаемые архитектуры: ARM.

Рассматривая данные ОС (iOS и Android), нельзя не отметить значительные отличия между архитектурными решениями на инфраструктурном уровне обеих ОС. Apple решила использовать Objective-C как язык программирования и среду выполнения приложения iOS. Objective-C выглядит более или менее естественным выбором для ОС, в основе которой лежит Free BSD. Можно рассматривать Objective-C как обычный C++ с кастомным препроцессором, добавляющим некоторые специфические лингвистические конструкции. В результате мы имеем ситуацию, что приложения iOS написаны более или менее на том же языке, что и стоящая за ними ОС[8].

Android-приложения сильно отличаются в этом смысле. Они написаны на Java, а это совсем другая технология, нежели C++ (хотя синтаксис и унаследован от C++). Кроме того, iOS работает только на оборудовании собственного производства, и Apple полностью контролирует весь процесс. Для Android же все наоборот: Google не контролирует производителей аппаратных средств. В результате приложения под Android должны работать на зачастую кардинально отличающихся платформах. В Java эта проблема решается с помощью виртуальной машины. Однако ввиду ограниченности ресурсов мобильных платформ использование стандартной JVM было затруднено. Решением стало создание своей виртуальной машины, которая получила название Dalvik. Dalvik – регистровая виртуальная машина для выполнения программ, написанных на языке программирования Java, созданная группой разработчиков Google во главе с Д. Борнштейном. Дальнейшие разработки привели к появлению среды выполнения Android-приложений (ART). ART впервые появился в Android 4.4 как тестовая функция, а в Android 5.0 полностью заменил Dalvik. В отличие от Dalvik, который использует JIT-компиляцию (во время выполнения приложения), ART компилирует приложение во время его установки.

---

[7] Основы Android. URL: http://aeterna-ufa.ru/sbornik/TN-08.pdf.

[8] Невзоров В. Архитектура Android-приложений. Ч. I – истоки. Блог. Хабр. URL: https://habr.com/ru/post/140459.

Windows – ОС компании Microsoft. Долгое время занимала доминирующее положение среди ОС вообще и по настоящее время занимает такое положение в плане использования в «настольных» персональных компьютерах и высокопроизводительных ноутбуках. Первые продукты с названием «Windows» от Microsoft не были ОС. Это были графические среды для ОС MS-DOS. Первая полноценная ОС была выпущена в 1995 г. – Windows 95. Она также стала первой системой целого семейства Windows 9x. Однако у систем данного семейства были проблемы с безопасностью и стабильностью работы приложений, вследствие чего последней версией стала Windows Me, после которой поддержка ветки Windows 9x была прекращена. Дальнейшее развитие получила другая ветка ОС, не основанная на MS-DOS: так называемая NT – от сокращения *New Technology*. Основной упор при разработке NT делался на безопасность и надежность системы, а также на совместимость с Windows на MS-DOS. Первая система называлась Windows NT 3.1 и была выпущена в 1993 г. ОС Windows 10/11 также формально относятся к этой ветке. Начиная с версии Windows 8 в ОС стало два основных API для работы приложений: Windows API (также известен как Win32) и Windows Runtime (WinRT)[9]. WinRT представляет принципиально новую технологию, направленную на разработку приложений, которые могут работать на разных типах устройств (например, ПК, смартфон, Xbox и пр.). Позже, начиная с версии Windows 10, данная технология стала называться *Универсальная платформа Windows* (UWP) (рис. 1).

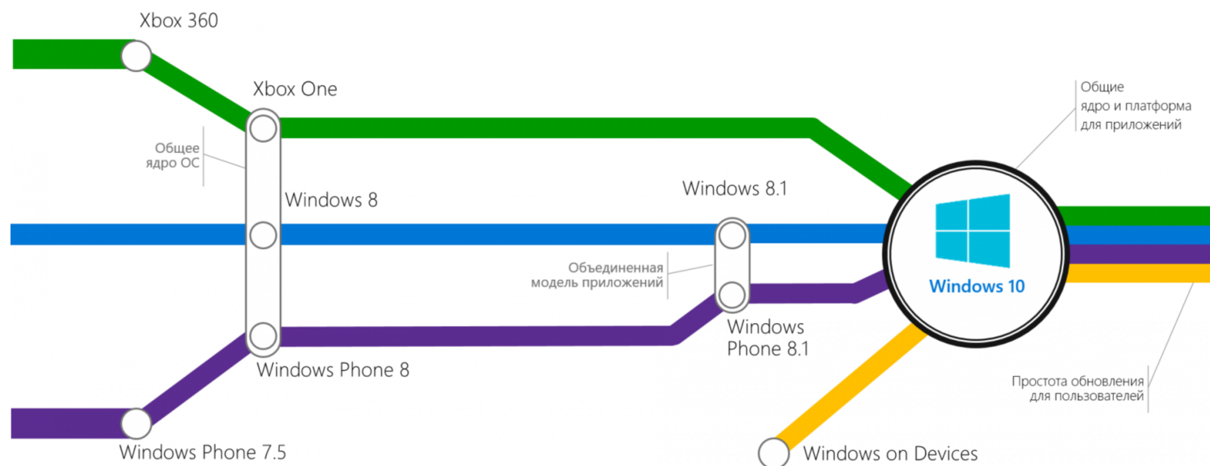

Рис. 1. История развития WinRT

UWP – это специальная платформа для создания приложений на Windows 10. С ее использованием можно разрабатывать приложения для UWP с помощью всего одного набора API, одного пакета приложений и одного магазина для доступа ко всем устройствам Windows 10 – ПК, планшета, телефона, Xbox, HoloLens, Surface Hub и др. Легче поддерживать несколько размеров экрана, а также различные модели взаимодействия, будь то сенсор, мышь и клавиатура, игровой контроллер или ручка. В основе приложений UWP лежит следующая идея: пользователи хотят, чтобы их работа и их задачи могли выполняться на

[9] Windows Runtime. URL: https://ru.wikipedia.org/wiki/Windows_Runtime.

всех устройствах, чтобы можно было использовать любое устройство, наиболее удобное или производительное для конкретной задачи[10].

1.2. Мобильные устройства и проблема фрагментации

Под мобильным устройством принято понимать любой портативный компьютер: планшет, электронный ридер, смартфон, карманный персональный компьютер (КПК), «умные» часы, портативный музыкальный плеер с интеллектуальными возможностями. Часто в качестве синонимов понятия «мобильное устройство» употребляются такие термины, как «карманное устройство», «карманный компьютер». Отличительными характеристиками таких устройств являются малые размеры и вес, возможность работать с сетями связи с помощью беспроводных технологий.

С начала 2000-х гг. ни одно устройство в мире не развивалось так быстро, как мобильное устройство. Сегодня практически у каждого жителя планеты есть один или несколько подобных устройств. Бурное развитие мобильных устройств началось в 2002 г., когда на свет появился первый в мире смартфон – мобильный телефон с ОС Nokia 9210. Вначале смартфоны рассматривались как «продвинутые» версии обычных телефонов, но примерно с 2007 г. ситуация стала меняться – люди начали активно использовать смартфоны не только для звонков, но и для фотографий, серфинга в Интернете, прослушивания музыки, игр. Делать все это на маленьком экране телефона стало неудобно, поэтому следующие пять лет смартфоны начали расти в размерах и производительности. Немалая заслуга в этом принадлежит компании Apple, которая в 2007 г. выпустила свой iPhone – первый в мире смартфон с сенсорным экраном. По сути, это устройство стало стандартом для всех современных смартфонов – увеличенный сенсорный экран и минимум кнопок. В этом отношении знаковым можно считать 2012 г. Именно в этом году количество смартфонов с большим экраном превысило число устройств с маленькими дисплеями.

В 2015 г. начались продажи «умных» часов Apple Watch, анонсированных в сентябре 2014 г. Первые модели выполняли простые задачи, например, выступали в роли калькулятора, переводчика или игрового устройства. В настоящее время компьютеризированные наручные часы обладают очень широкой функциональностью и по этому показателю зачастую сравнимы с коммуникаторами. Вполне допустимо сказать, что современные «умные» часы – это носимые компьютеры. Многие модели поддерживают сторонние приложения и управляются мобильными операционными системами, а также могут выступать в качестве мобильных медиаплееров[11].

Также нельзя обойти вниманием и планшеты (tablet computers) – как и ноутбуки, планшетные компьютеры предназначены для портативной работы. Наиболее заметным отличием планшета от ноутбука является то, что планшетные компьютеры не имеют клавиатуры и тачпада (в ряде случаев клавиатуру

[10] Что такое Universal Windows Platform (UWP)? Блог ITVDN. URL: https://itvdn.com/ru/blog/article/uwp.
[11] Умные часы (устройство). URL: https://en.wikipedia.org/wiki/Smartwatch.

можно подключить отдельно). Однако весь экран является сенсорным, виртуальная клавиатура появляется тогда, когда необходимо ввести текстовую информацию, а перемещение указателя на экране осуществляется пальцем. Планшетные компьютеры предназначены главным образом для потребления контента, т.е. они оптимизированы для таких задач, как просмотр веб-страниц, видео, работа с электронной почтой, чтение электронных книг, игры и пр.

Таким образом, за относительно недолгую, но «бурную» историю развития было разработано огромное количество устройств с очень различающимися характеристиками. Это приводит к тому, что приложения для мобильных платформ очень сильно отличаются от тех, что разработаны для «настольных» приложений. Немаловажным фактором тут является *проблема фрагментации*. Под фрагментацией платформы понимается ситуация, когда у какой-то вычислительной платформы становится настолько много моделей аппаратуры и версий ОС, что практически невозможно написать программу, хорошо работающую на всех устройствах, созданных на базе данной вычислительной платформы[12]. Действительно, в основе мобильных устройств лежат очень разные аппаратные и программные платформы.

1.3. Нативная и кроссплатформенная разработка

Нативная, или платформенно-ориентированная разработка означает, что используется оригинальный язык и инструменты конкретной ОС. Разработка приложений под iOS происходит в среде разработки XCode на языке Swift (раньше – на Objective-C). При использовании технологии разработки мобильных приложений на платформе Android используется среда Android Studio и язык Kotlin (до 2018 г. основным языком был Java).

Разработка нативного приложения имеет свои преимущества и недостатки. К преимуществам можно отнести[13]:

1. Гибкий функционал. Разработка приложения под определенную ОС позволяет реализовать возможности, поддерживаемые именно этой системой. К тому же такие функции будут работать более корректно. Кроме того, есть возможность разработать функционал с учетом уникальных функций устройства.

2. Более высокая скорость работы. При создании приложения используется понятный и привычный для платформы код, поэтому оно способно работать более быстро и качественно. При этом в кроссплатформенной разработке приложение может работать не так оперативно.

3. Более понятный интерфейс. Для дизайна приложения в нативной разработке используются гайдлайны[14]. Это рекомендации по адаптации дизайна приложения для конкретной платформы. Соответственно, дизайн нативного приложения будет более привычным и удобным для пользователя.

---

[12] Иванько А.Ф., Иванько М.А., Бурцева М.Б. Операционные системы мобильных мультимедиа устройств для журналиста // Молодой ученый. 2018. № 1 (187). С. 1–5. URL: https://moluch.ru/archive/187/47634.

[13] Что выбрать: кроссплатформенную или нативную разработку. URL: https://sibdev.pro/blog/articles/chto-vybrat-krossplatformennuyu-ili-nativnuyu-razrabotku.

[14] Гайдлайн – свод правил и/или рекомендаций для формирования внешнего вида продукт.

К недостаткам можно отнести:

1. Зависимость от платформы. Приложение, разработанное для одной системы, можно запустить только в рамках этой системы.

2. Более высокая стоимость разработки. Чтобы программой могли пользоваться и владельцы Android, и iOS, придется нанимать две команды разработчиков. Бюджет в этом случае может вырасти примерно на 20 % по сравнению с кроссплатформенной разработкой. Если же стоит задача учесть и потребности потребителей Windows, то увеличение стоимости будет трехкратное. Это может привести к сложностям в коммуникации, могут появиться труднораспознаваемые различия между платформами, отставания в обновлениях и рассинхронизация в функциях.

Кроссплатформенные приложения пишутся сразу для нескольких платформ на одном языке, отличном от нативного. Для того чтобы такой код мог работать, есть два подхода[15]:

1. На этапе подготовки приложения к публикации код превращается в нативный для определенной платформы с помощью транспилера[16]. Фактически один кроссплатформенный язык программирования «переводится» на другой.

2. К получившемуся коду добавляется определенная «обертка», которая, работая уже на устройстве, на лету транслирует вызовы из неродного кода к родным функциям системы.

Наибольшее распространение получил второй подход. Кроме этого, все кроссплатформенные фреймворки можно разделить на две группы по тому, задействован или нет в разработке Web: Hybrid-Native (гибридно-нативная разработка) и Hybrid-Web (гибридная веб-разработка)[17].

Гибридно-нативный подход объединяет фреймворки с нативным пользовательским интерфейсом и общим кодом, а также фреймворки с общей кодовой базой и нативным кодом. Среди них кроссплатформенные платформы разработки приложений React Native, Xamarin, Flutter.

Гибридная веб-разработка кроссплатформенных приложений осуществляется на базе фреймворков с веб-интерфейсом и общими компонентами и с единой кодовой базой, работающей где угодно. Примерами таких кроссплатформенных фреймворков являются PhoneGap, Ionic.

С точки зрения архитектуры, все ОС – iOS, Android и Windows UWP – предоставляют доступ к следующим подсистемам (наборы системных API)[18]:

– WebView (встроенный в приложение веб-браузер) используется в гибридных приложениях на базе PhoneGap и фактически выступает средой выполнения локального веб-сайта;

---

[15] Беспалов Д.А., Гушанский С.М., Коробейникова Н.М. Операционные системы реального времени и технологии разработки кроссплатформенного программного обеспечения : учеб. пособие. Ч. 2. Ростов-на-Дону; Таганрог : Изд-во ЮФУ, 2019. 168 с.

[16] Транспиляция – преобразование программы, при котором используется исходный код программы, написанной на одном языке программирования в качестве исходных данных, и производится эквивалентный исходный код на другом языке программирования.

[17] Лучшие фреймворки для разработки кроссплатформенных мобильных приложений. URL: https://blog.sibirix.ru/crossplatform-frameworks.

[18] Нативно или нет? Четыре мифа о кроссплатформенной разработке. URL: https://xakep.ru/2017/12/25/crossplatform-myths.

– JavaScript-движки используются в React Native и аналогах для быстрого выполнения JS-кода и обмена данными между Native и JS;

– OpenGL ES (или DirectX) применяется в игровых движках и приложениях на Qt/QML или аналогах для отрисовки интерфейса;

– UI-подсистема отвечает за нативный пользовательский интерфейс приложения, что актуально для React Native и Xamarin (рис. 2).

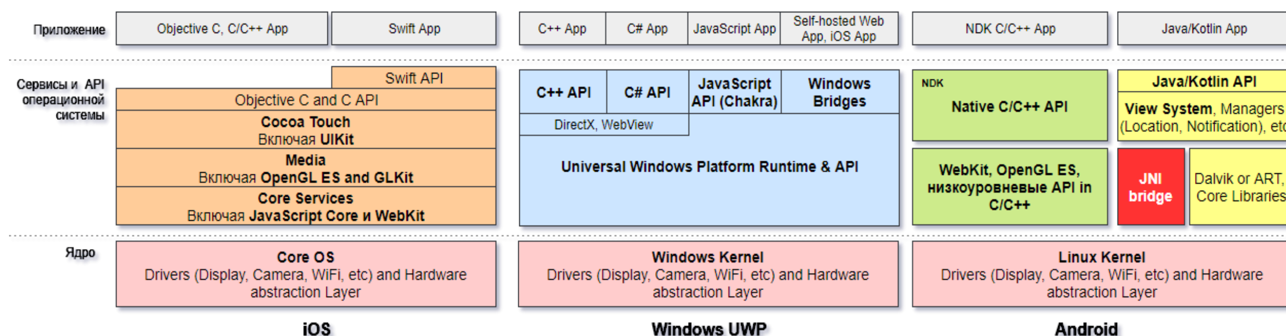

Рис. 2. Наборы системных API

Таким образом, кроссплатформенные приложения имеют нативную часть и такой же полный доступ к системным API, что и нативные приложения. Разница в том, что вызов системного метода идет через мост и инфраструктуру фреймворка.

Кроссплатформенная разработка крайне востребована среди крупных мировых компаний. Многие известные приложения являются кроссплатформенными. К недостаткам данного подхода можно отнести:

– более низкую производительность (зависит от выбранного инструмента);

– меньше возможностей интеграции с аппаратными функциями устройства;

– меньшая «гибкость» настройки и оптимизации, какая есть у каждой ОС со своим собственным стеком технологий.

Рассмотрим несколько популярных инструментов для разработки кроссплатформенных приложений[19].

PhoneGap – бесплатный open-source фреймворк для разработки мобильных приложений от компании Nitobi Software. Позволяет создавать приложения для мобильных устройств, используя JavaScript, HTML5 и CSS3, без необходимости знания «родных» языков программирования (например, Objective-C) под все мобильные ОС (iOS, Android и т.д.)[20].

Приложение на PhoneGap – это по факту нативное приложение, которое в качестве единственного UI-контрола отображает WebView (рис. 3). Именно через него и идет взаимодействие с нативной частью. Все стандартные WebView в iOS, Android и Windows UWP поддерживают возможность добавить свои нативные обработчики для JS-свойств и -методов. При этом JS-код живет в своей изолированной среде и ничего не знает о нативной части, а просто использует нужные

[19] Черников В.Н. Разработка мобильных приложений на C# для iOS и Android. М. : ДМК Пресс, 2020. 188 с.

[20] PhoneGap. URL: https://ru.wikipedia.org/wiki/PhoneGap.

JS-методы или меняет нужные JS-свойства. Все внутри стандартного DOM, в который просто добавляются новые элементы, связанные с нативной реализацией.

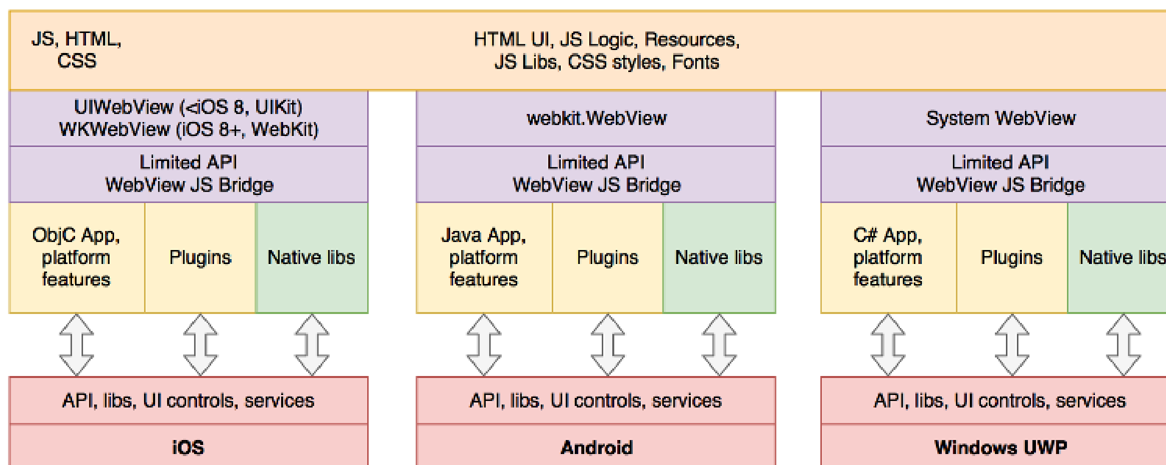

Рис. 3. Архитектура PhoneGap

React Native. Кроссплатформенный фреймворк с открытым исходным кодом для разработки нативных мобильных и настольных приложений на JavaScript и TypeScript, созданный Facebook, Inc. React Native поддерживает такие платформы, как Android, Android TV, iOS, macOS, tvOS, Web, Windows и UWP, позволяя разработчикам использовать возможности библиотеки React вне браузера для создания нативных приложений. Основные принципы работы React Native практически идентичны принципам работы React, за исключением того, что React Native управляет не браузерной DOM, а платформенными интерфейсными компонентами. JavaScript-код, написанный разработчиком, выполняется в фоновом потоке, и взаимодействует с платформенными API через асинхронную систему обмена данными, называемую Bridge (рис. 4)[21]. Особенностью является работа фреймворка на iOS: из-за ограничений iOS код JavaScript на лету интерпретируется, а не компилируется. В целом это не особо сказывается на производительности в реальных приложениях, но помнить об этом стоит.

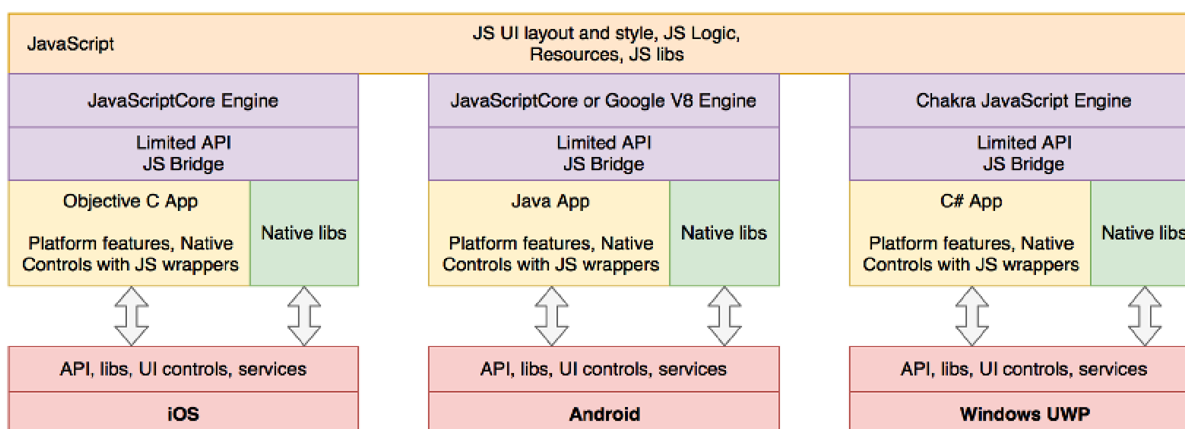

Рис. 4. Архитектура React Native

[21] React Native. URL: https://ru.wikipedia.org/wiki/React_Native.

Xamarin. Xamarin использует библиотеку Mono для взаимодействия с целевой ОС, которая позволяет вызывать нативный код с помощью механизма P/Invoke. Он же задействуется и для общения с нативными API в iOS/Android, т.е. для всех публичных нативных API-методов создаются обертки на C#, которые в свою очередь вызывают системные API (рис. 5). Таким образом, из Xamarin-приложения можно обращаться ко всем системным API.

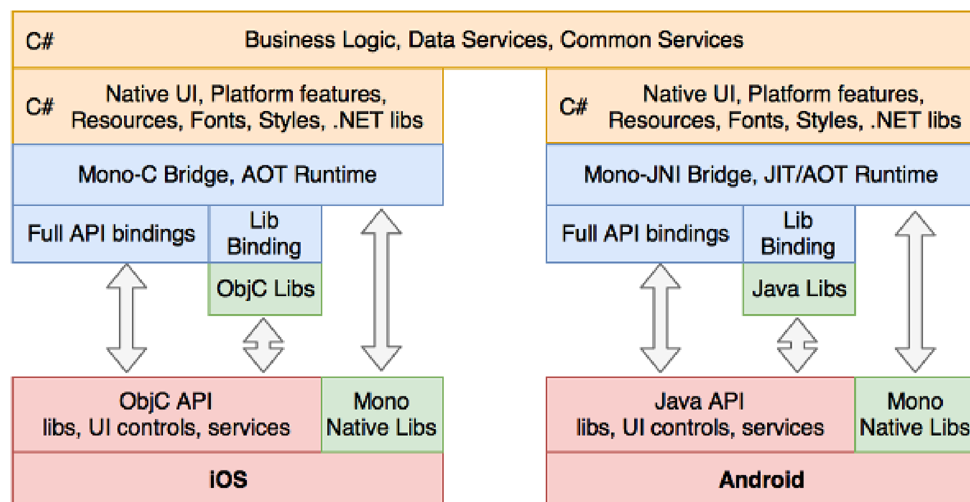

Рис. 5. Архитектура Xamarin

Xamarin.Forms представляет собой платформу, позволяющую вести разработку кроссплатформенных приложений под Android, iOS и Windows 10 с учетом различий в исполнении приложений[22]. С точки зрения исполнения приложений между iOS и Android есть одно ключевое различие – способ их предварительной компиляции. Как известно, для выполнения приложений в Android используется виртуальная Java-машина Dalvik. Нативные приложения, которые пишутся на Java, компилируются в некий промежуточный байт-код, интерпретируемый в команды процессора в момент исполнения программы (т.е. аналогично тому, как работает CLR в .NET). Это так называемая Just-in-time-компиляция (компиляция на лету). В iOS используется другая модель компиляции – Ahead-of-Time (компиляция перед исполнением). Xamarin учитывает это различие, предоставляя отдельные компиляторы для каждой из этих платформ, позволяющих на выходе получать настоящие нативные приложения, которые выполняются вне контекста браузера и могут использовать все аппаратные и программные ресурсы платформы. Xamarin позволяет создавать одну единую логику приложения с применением C# и .NET сразу для всех трех платформ – Android, iOS, UWP.

Контрольные вопросы

1. Понятие нативной разработки, примеры инструментов.
2. Понятие кроссплатформенной разработки, примеры инструментов.
3. Сравнение нативной и кроссплатформенной разработки: плюсы и минусы.

[22] URL: https://habr.com/ru/post/188130.

4. Понятие платформы: аппаратная и программная платформы.
5. Виды аппаратных платформ.
6. Виды программных платформ.
7. Архитектуры кроссплатформенных инструментов разработки.
8. Сравнение кроссплатформенных инструментов разработки.
9. Проблема фрагментации, причины и следствие.
10. История развития платформ.

2. ФРЕЙМВОРК ДЛЯ КРОССПЛАТФОРМЕННОЙ РАЗРАБОТКИ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ XAMARIN.FORMS

2.1. Основы Xamarin

2.1.1. Обзор и основные компоненты платформы

Xamarin представляет собой фреймворк для кроссплатформенной разработки мобильных приложений для ОС iOS, Android и Windows 10/11. Обычно в качестве среды разработки используют Microsoft Visual Studio, а в качестве основного языка разработки – C#. Таким образом, приложения с использованием фреймворка Xamarin можно писать на персональных компьютерах с установленной ОС Windows или на компьютерах Apple Mac.

Платформа Xamarin реализует уровень абстракции, обеспечивающий управление взаимодействием между общим кодом и кодом базовой платформы[23]. Приложение Xamarin выполняется в управляемой среде, которая реализует такие возможности, как выделение памяти и сборка мусора. Можно сказать, что Xamarin – это не только инструмент, но некий подход к процессу разработки. С помощью этого фреймворка разработчик может написать всю бизнес-логику на одном языке (или использовать существующий код приложения), но при этом получить характеристики производительности, оформление и поведение, характерные для каждой поддерживаемой платформы. Благодаря Xamarin в среднем до 90 % кода приложения можно использовать без изменений. Приложения Xamarin компилируются в собственные пакеты приложений, например в файлы с расширением .apk для Android или .ipa для iOS.

Преимущества использования Xamarin:

– при разработке создается единый код для всех платформ;

– предоставляется прямой доступ к нативным API каждой платформы;

– при создании приложений можно использовать платформу .NET и язык программирования C#;

– Xamarin.Forms поддерживает несколько платформ.

В основе Xamarin лежит среда .NET, которая автоматически обрабатывает такие задачи, как выделение памяти, сборка мусора и обеспечение взаимодействия с базовыми платформами. На рис. 6 показана общая архитектура кроссплатформенного приложения Xamarin.

Основные компоненты (платформы): Xamarin.Android, Xamarin.iOS, Xamarin.Forms.

Приложения **Xamarin.Android**[24] компилируются из языка C# в промежуточный язык (IL), который при запуске приложения претерпевает Just-in-Time-компиляцию (JIT) в машинную сборку. Приложения Xamarin.Android работают

---

[23] Гринштейн С. Просто и понятно о Xamarin: как разработать кроссплатформенное мобильное приложение. Пошаговая инструкция. URL: https://highload.today/ponyatno-o-xamarin-kak-razrabotat-krossplatformennoe-mobilnoe-prilozhenie-poshagovaya-instruktsiya.

[24] URL: https://learn.microsoft.com/ru-ru/xamarin/get-started/what-is-xamarin.

в среде выполнения Mono параллельно с виртуальной машиной среды выполнения Android (ART). Xamarin предоставляет привязки .NET к пространствам имен Android.* и Java.* Среда выполнения Mono обращается к этим пространствам имен с использованием управляемых вызываемых оболочек (MCW) и предоставляет среде выполнения ART вызываемые программы-оболочки Android (ACW), благодаря чему обе среды могут вызывать код друг друга (рис. 7).

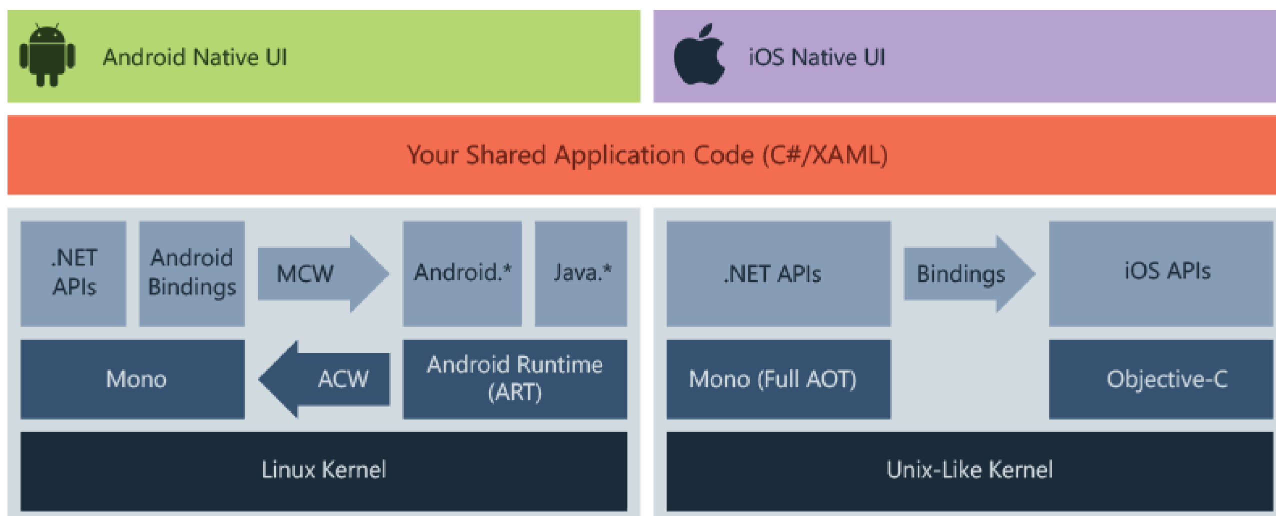

Рис. 6. Архитектура Xamarin.Forms

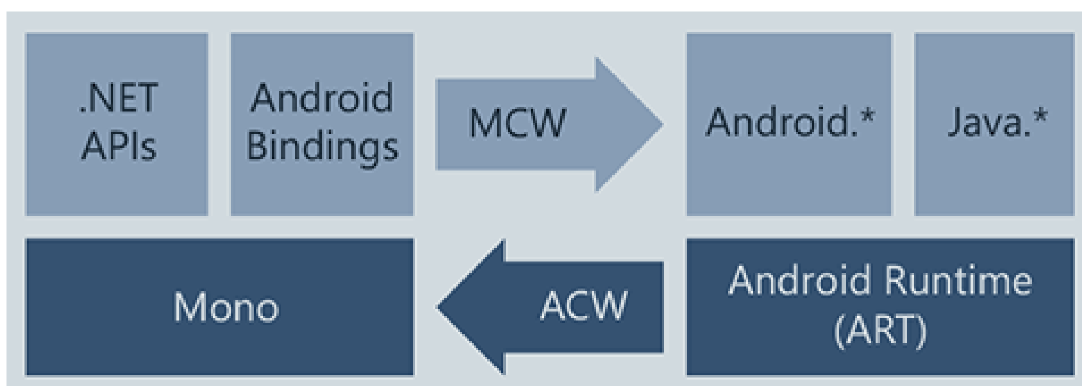

Рис. 7. Структура проекта Xamarin.Android

Для разработки приложений Android требуется установить пакеты SDK для Java и Android. Пакеты SDK предоставляют компилятор, эмулятор и другие средства, необходимые для сборки, развертывания и тестирования. Java, пакет SDK для Android Google и средства Xamarin можно установить и запустить в Windows и macOS.

Приложения **Xamarin.iOS** проходят полную Ahead-of-Time-компиляцию (AOT) из языка C# в собственный код сборки ARM. Xamarin использует селекторы для предоставления кода Objective-C управляемому коду C# и регистраторы для предоставления управляемого кода C# коду Objective-C. Селекторы и регистраторы в совокупности называются «привязками» и обеспечивают взаимодействие между Objective-C и C# (рис. 8).

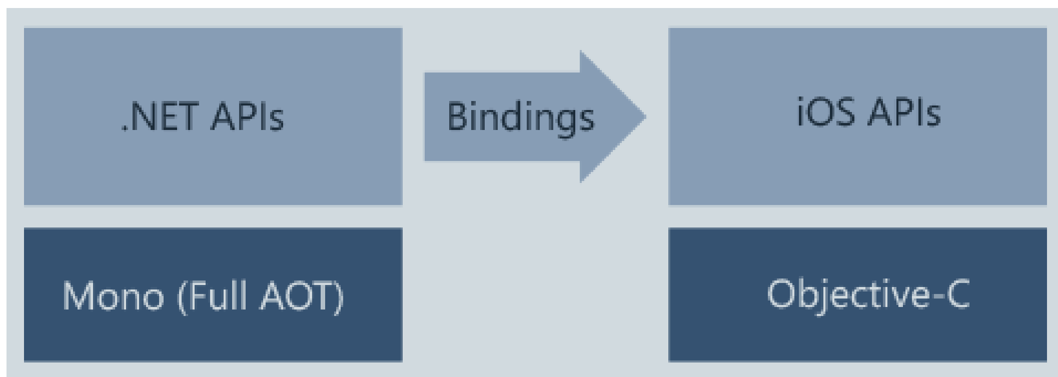

Рис. 8. Структура проекта Xamarin.iOS

Для разработки приложений для iOS требуется компьютер Mac под управлением macOS. Visual Studio можно использовать для разработки и развертывания приложений iOS с помощью Xamarin, однако компьютер Mac необходим для целей сборки и лицензирования.

Интегрированная среда разработки Xcode Apple должна быть установлена для предоставления компилятора и симулятора для тестирования. Вы можете бесплатно протестировать свои устройства, но при создании приложений для распространения (например, App Store) необходимо присоединиться к программе разработчиков Apple.

Xamarin.Forms – это платформа пользовательского интерфейса с открытым кодом. С помощью Xamarin.Forms разработчики могут создавать приложения для Xamarin.iOS, Xamarin.Android и Windows на основе общей базы кода. Xamarin.Forms позволяет разработчикам создавать пользовательские интерфейсы в XAML с помощью кода программной части в C#. Эти пользовательские интерфейсы на каждой платформе подготавливаются к просмотру как собственные элементы управления. Приведем некоторые примеры функций, предоставляемых Xamarin.Forms: язык пользовательского интерфейса XAML; привязка данных; жесты; произведенный эффект; задание стиля.

Таким образом, Xamarin.Forms является одним из компонентов Xamarin, на базе которого можно создать кроссплатформенный пользовательский интерфейс с помощью XAML. Строго говоря, можно использовать Xamarin без Xamarin.Forms, создавая интерфейс посредством классов-оберток над нативными классами, но Xamarin без Forms нельзя использовать для кроссплатформенного создания пользовательского интерфейса.

2.1.2. Методы создания общего кода

Существует три метода совместного использования кода между кроссплатформенными приложениями: библиотеки .NET Standard; общие проекты; переносимые библиотеки классов. Рассмотрим эти методы.

Библиотеки .NET Standard предоставляют четко определенный набор библиотек базовых классов, на которые можно ссылаться в разных типах проектов, включая кроссплатформенные проекты, такие как Xamarin.Android и Xamarin.iOS (рис. 9).

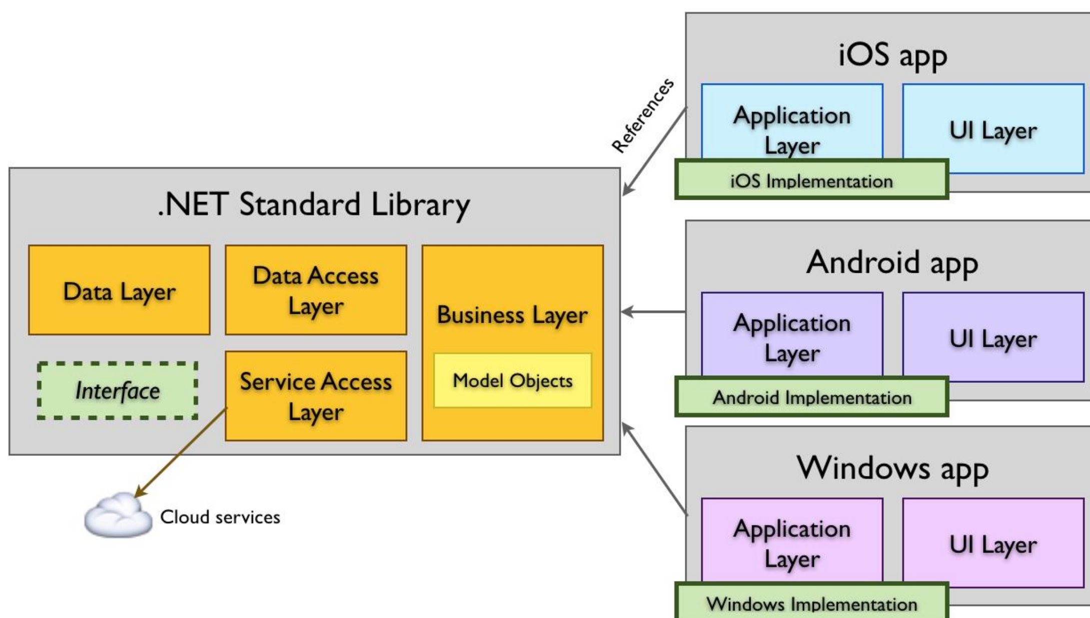

Рис. 9. Схема работы метода «библиотеки .NET Standard»

К преимуществам данного похода можно отнести следующее:

– возможность совместно использовать код в нескольких проектах;

– операции рефакторинга всегда обновляют все затронутые ссылки;

– большая часть библиотеки базовых классов .NET доступна.

Недостаток данного подхода – нельзя использовать директивы компилятора, такие как `#if __IOS__`.

Общие проекты содержат файлы кода и ресурсы, включенные в любой проект, ссылающийся на них. Общий проект не компилируется сам по себе, он существует исключительно как группировка файлов исходного кода, которые могут быть включены в другие проекты. Когда на общий проект ссылается другой проект, то код собирается как часть этого проекта. Концептуальная архитектура показана на рис. 10, где каждый проект содержит все общие исходные файлы.

Кроссплатформенное приложение, поддерживающее iOS, Android и Windows, потребует проекта приложения для каждой платформы. Общий код будет находиться в Shared Project. К преимуществам такого подхода можно отнести:

– возможность совместно использовать код в нескольких проектах;

– можно в рамках общего кода использовать директивы компилятора, такие как `#if __IOS__` или `#if __ANDROID__` для написания платформозависимого кода;

– проекты приложений могут содержать ссылки на платформы, которые может использовать общий код.

Недостатки метода:

– рефакторинг, влияющий на код внутри директив компилятора, не обновляет код внутри этих директив;

– общий проект не имеет «выходной» сборки, в отличие от большинства других типов проектов, поэтому во время компиляции файлы обрабатываются как часть ссылающегося проекта и компилируются в этой сборке.

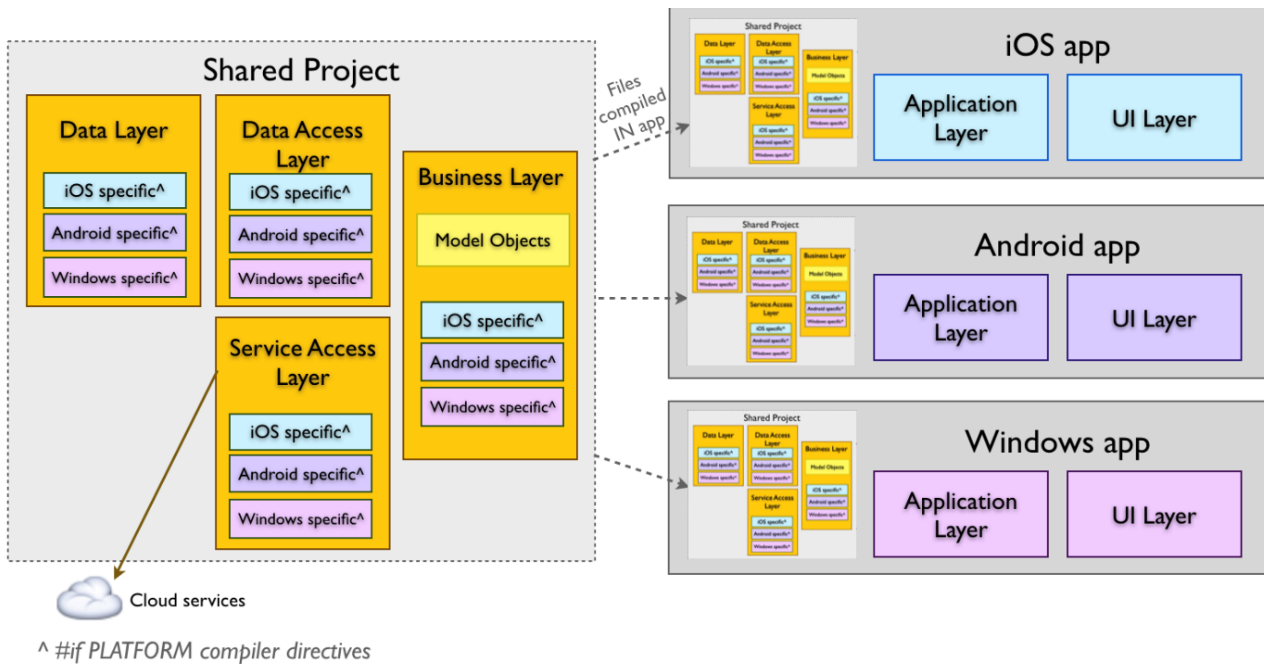

Рис. 10. Общие проекты

В рамках метода **«переносимые библиотеки классов»** создаются библиотеки переносимых классов (PCL), предназначенные для платформ, которые необходимо поддерживать, и использующие интерфейсы для обеспечения специфичной для платформы функциональности. Концептуальная архитектура данного подхода приведена на рис. 11. Шаблон PCL на текущий момент времени считается устаревшим (заменен на «библиотеки .NET Standard») и не рекомендуется к применению.

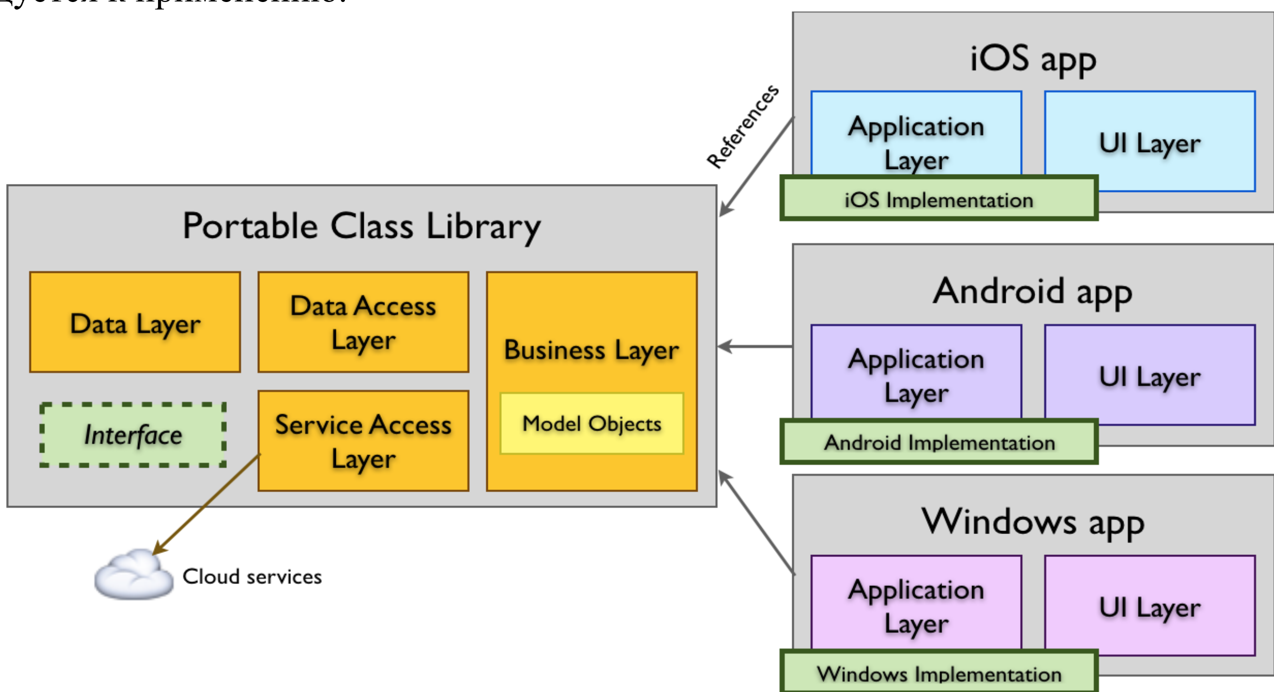

Рис. 11. Концептуальная архитектура PLC

В рамках данного подхода можно эффективно совместно использовать код в нескольких проектах, а операции рефакторинга всегда обновляют все затронутые ссылки. К недостаткам следует отнести то, что нельзя использовать директивы компилятора, а для работы доступно только то подмножество платформы .NET Framework, которое определено выбранным профилем.

2.1.3. Поддерживаемые платформы и платформозависимый код

Xamarin поддерживает все типы устройств, доступные на поддерживаемых платформах, поэтому помимо телефонов, приложения также могут работать на следующих устройствах: ipad; планшеты с ОС Android; планшеты и настольные компьютеры под управлением Windows 10/11.

При разработке кроссплатформенных программных продуктов необходимо учитывать, что не все характеристики и функции поддерживаемых платформ являются одинаковыми (универсальными), целый ряд из них платформозависимы. В большинстве случаев к универсальным можно отнести такие основные элементы:

– организация меню, оконных форм, вкладок и т.д.;
– работа с данными (списки, прокрутка и т.д.);
– формы работы с данными, представления, элементы и т.д.;
– навигация по страницам приложения.

При проектировании приложения можно создать общий код для этих функций.

Во вторую группу попадают функции и характеристики, которые в большинстве реализаций присутствуют на каждой платформе, но существенно отличаются в зависимости от конкретной реализации:

1. Размеры экрана. Некоторые платформы (например, iOS) обладают стандартизированными размерами экрана и относительно просты в использовании. Устройства Android имеют большое разнообразие размеров экрана, что затрудняет построение «единого» интерфейса приложения.

2. Клавиатуры. Некоторые устройства имеют физические клавиатуры, а другие – только программные.

3. Касание и жесты. Поддержка ОС для распознавания жестов зависит от версий каждой ОС. Более ранние версии Android имеют очень ограниченную поддержку сенсорных операций, т.е. для поддержки старых устройств может потребоваться отдельный код.

4. Push-уведомления. На каждой платформе существуют различные возможности и реализации.

К третьей группе следует отнести функции для конкретных устройств. При работе с такими функциями нужно обязательно предусмотреть код для проверки, если такая функция есть на устройстве. При ее отсутствии необходимо дать пользователю альтернативу (например, при невозможности задействовать GPS / Глонасс для определения географического расположения можно предоставить пользователю средства для выбора расположения вручную на карте). К данной группе относятся:

1. Камера. Функциональные возможности камеры сильно различаются на разных устройствах. В некоторых устройствах вообще может не быть камеры, а в других есть несколько камер (например, передняя и задняя камеры). Некоторые камеры поддерживают запись видео, а другие только фото и т.д.

2. Определение географического положения. Поддержка GPS/Глонасс или определение расположения по мобильной сети/Wi-Fi реализуются не на всех устройствах. Приложениям также необходимо ориентироваться на различные уровни точности, поддерживаемые каждым методом.

3. Акселерометр, гироскоп и компас. Эти функции встречаются не на каждой мобильной платформе, поэтому приложения почти всегда должны предоставлять альтернативную версию.

4. Радиочастотная связь ближнего действия (NFC).

В случае, если проект организован по методу Shared Project, то для написания платформозависимого кода можно использовать условную компиляцию, например:

```
#if WINDOWS_UWP
// код для Windows 10
#elif __ANDROID__
// код для Android
#elif __IOS__
// код для iOS
#endif
```

При создании проекта с организацией кода по типу .NET Standard целесообразно использовать класс *Device*[25]. В этом случае свойство *RuntimePlatform* класса *Device* позволит получить программную платформу, на которой запущено приложение (ОС).

Свойство *Idiom* можно использовать для изменения макетов или функций в зависимости от устройства, на котором выполняется приложение. Перечисление *TargetIdiom*[26]:

– *Desktop* – приложение работает на настольном компьютере;

– *Phone* – ширина устройства мобильных платформ, на котором выполняется приложение, меньше 600 DIP;

– *Tablet* – ширина устройства мобильных платформ, на котором выполняется приложение, больше 600 DIP;

– *TV* – приложение работает в ОС Tizen на телевизоре Tizen;

– *Unsupported* – приложение работает на неподдерживаемом устройстве;

– *Watch* – приложение работает в часах Tizen.

Для работы с программно-аппаратными функциями мобильных платформ можно использовать Xamarin.Essentials. Он обеспечивает единый кроссплатформенный API-интерфейс, который предоставляет доступ из общего кода для любого

---

[25] Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/api/xamarin.forms.device?view=xamarin-forms.

[26] Там же.

приложения iOS, Android и универсальной платформы Windows независимо от используемого метода создания пользовательского интерфейса. Xamarin.Essentials предоставляется в виде пакета NuGet[27].

2.1.4. Класс Application и жизненный цикл приложения

Класс *Application*[28] является ядром приложения Xamarin. Когда вы запускаете либо останавливаете свое приложение (класс *Application*), все остальное также останавливается или запускается. App.xaml – первая стартовая точка приложения. Класс *Application* предлагает следующие возможности, которые предоставляются в используемом по умолчанию подклассе App проекта:

– свойство *MainPage* используется для задания начальной страницы для приложения;

– сохраняемый словарь *Properties* нужен для хранения простых значений в рамках изменений состояния жизненного цикла;

– статическое свойство *Current* содержит ссылку на текущий объект приложения.

Он также предоставляет методы жизненного цикла и события модальной навигации. Жизненный цикл мобильного приложения более сложный и отличается от настольных приложений. На мобильном устройстве приложение работает либо на переднем плане, либо в фоновом режиме. Рассмотрим жизненный цикл приложения на примере Android.

Жизненный цикл приложения в Android жестко контролируется системой и зависит от нужд пользователя, доступных ресурсов и т.д.[29] Схема жизненного цикла приложения Android приведена на рис. 12.

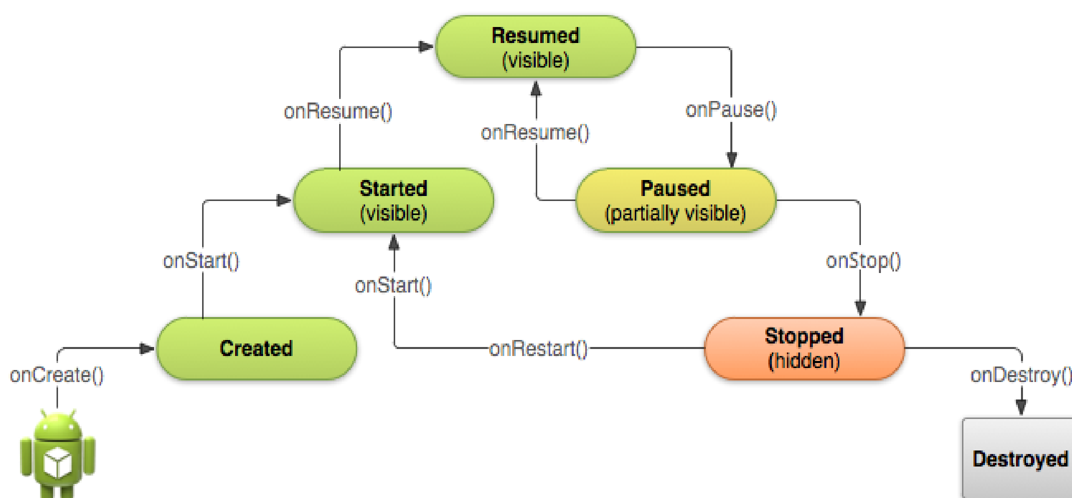

Рис. 12. Жизненный цикл приложения Android

---

[27] Рассмотрение пакета Xamarin.Essentials выходит за рамки данного учебного пособия, более подробно с его функциональными возможностями можно ознакомиться на сайте проекта. URL: https://github.com/xamarin/Essentials.

[28] Microsoft Learn.

[29] Жизненный цикл приложения на Android. URL: http://developer.alexanderklimov.ru/android/theory/lifecycle.php.

Например, пользователь хочет запустить браузер. Решение о запуске приложения принимает система. Хотя последнее слово и остается за системой, она подчиняется определенным заданным и логическим правилам, позволяющим определить, можно ли загрузить, приостановить приложение или прекратить его работу. Если в данный момент пользователь работает с определенным окном, система отдает приоритет соответствующему приложению. И наоборот, если окно невидимо и система решает, что работу приложения необходимо остановить, чтобы мобилизовать дополнительные ресурсы, будет прекращена работа приложения, имеющего более низкий приоритет.

В связи с этим класс *Application* содержит три виртуальных метода, которые можно переопределить для реагирования на изменения жизненного цикла:

– *OnStart* вызывается при запуске приложения;

– *OnSleep* вызывается каждый раз, когда приложение переводится в фоновый режим;

– *OnResume* вызывается каждый раз, когда приложение выводится из фонового режима и его работа возобновляется.

События *OnCreate* и *OnDestroy* в Xamarin не обрабатываются. Кроме того, в классе *Application* можно обработать следующие события модальной навигации:

– *ModalPushing* возникает при отправке страницы в модальном режиме;

– *ModalPushed* возникает после отправки страницы в модальном режиме;

– *ModalPopping* возникает при извлечении страницы в модальном режиме;

– *ModalPopped* возникает после извлечения страницы в модальном режиме.

2.2. Проект «Мобильное приложение (Xamarin.Forms)»

2.2.1. Создание проекта и интерфейс среды разработки

В качестве среды разработки будет использоваться Visual Studio 2022 с установленной рабочей нагрузкой «Разработка мобильных приложений на Net».

Для создания приложения откроем Visual Studio и выберем из списка «создание мобильного приложения Xamarin.Forms» (язык C#) (рис. 13).

В следующей оконной форме система предложит ввести название проекта и его месторасположение (рис. 14). Назовем проект MyProject.

Далее нужно будет выбрать базовый шаблон и поддерживаемые платформы (рис. 15).

Настройки шаблона приложения позволяют выбрать один из двух базовых каркасов пользовательского интерфейса и навигации между страницами. «**Всплывающий элемент»** представляет интерфейс навигации, основанный на всплывающих окнах и вкладках, навигация представлена боковым меню. Шаблон **«С вкладками»** создаст каркас приложения, которое будет использовать вкладки для навигации между страницами. Шаблон **«Пустой»** создаст каркас приложения с минимальной функциональностью, с одной экранной формой без навигации.

Создадим пустой проект, выберем все поддерживаемые платформы. На рис. 16 показано главное окно среды разработки с созданным проектом. Обратите внимание на три выделенные области: обозреватель решения, окно редактирования исходного кода и настройка сборки. Рассмотрим их более подробно.

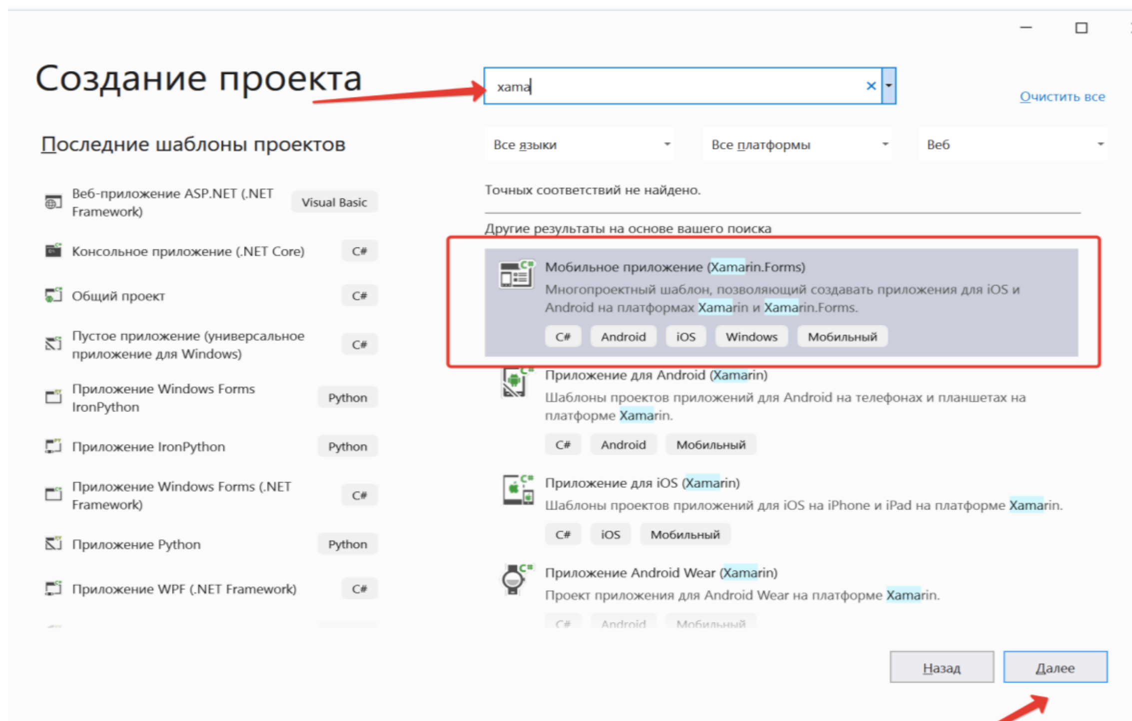

Рис. 13. Окно создания нового проекта

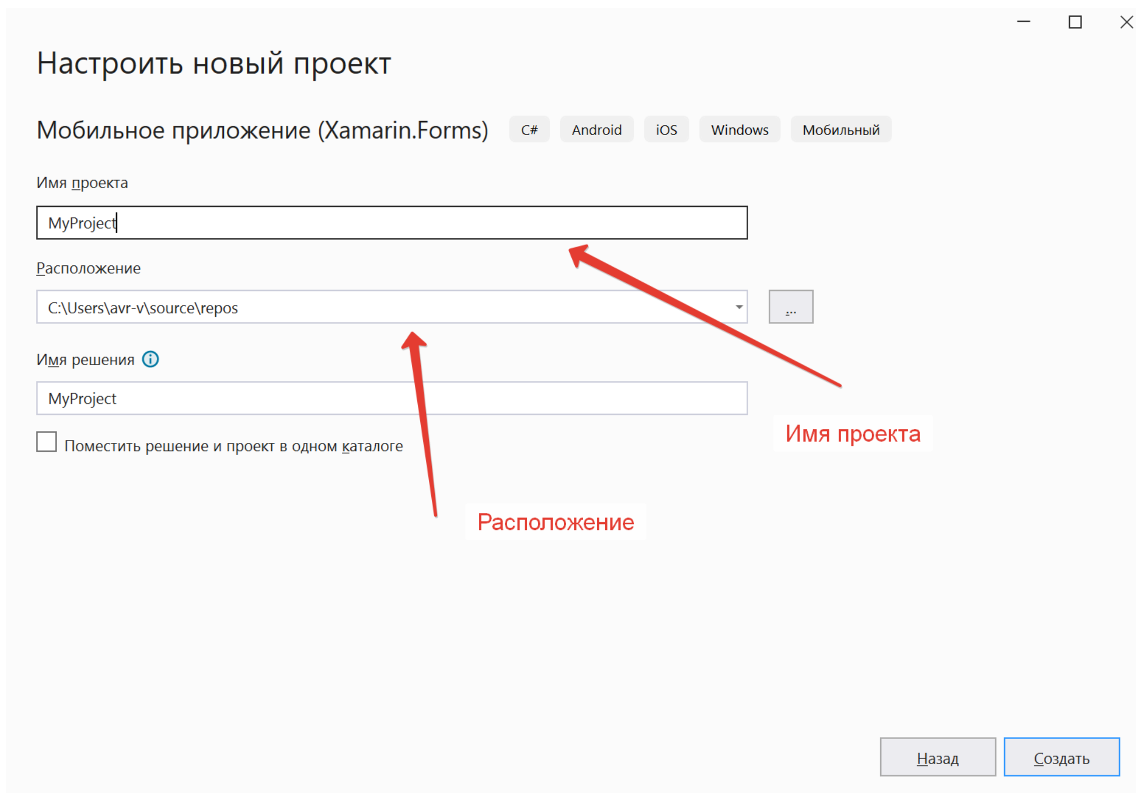

Рис. 14. Настройки нового проекта

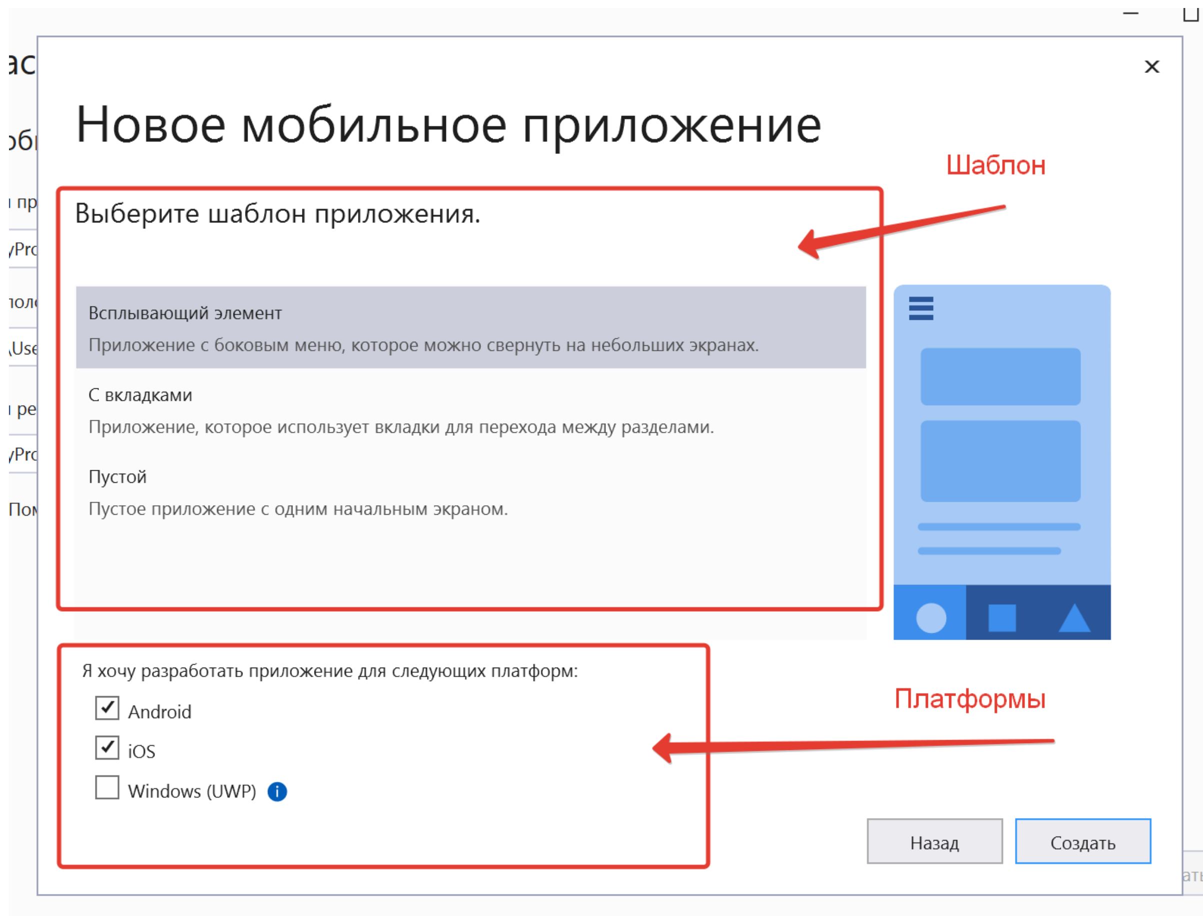

Рис. 15. Шаблоны и поддерживаемые платформы

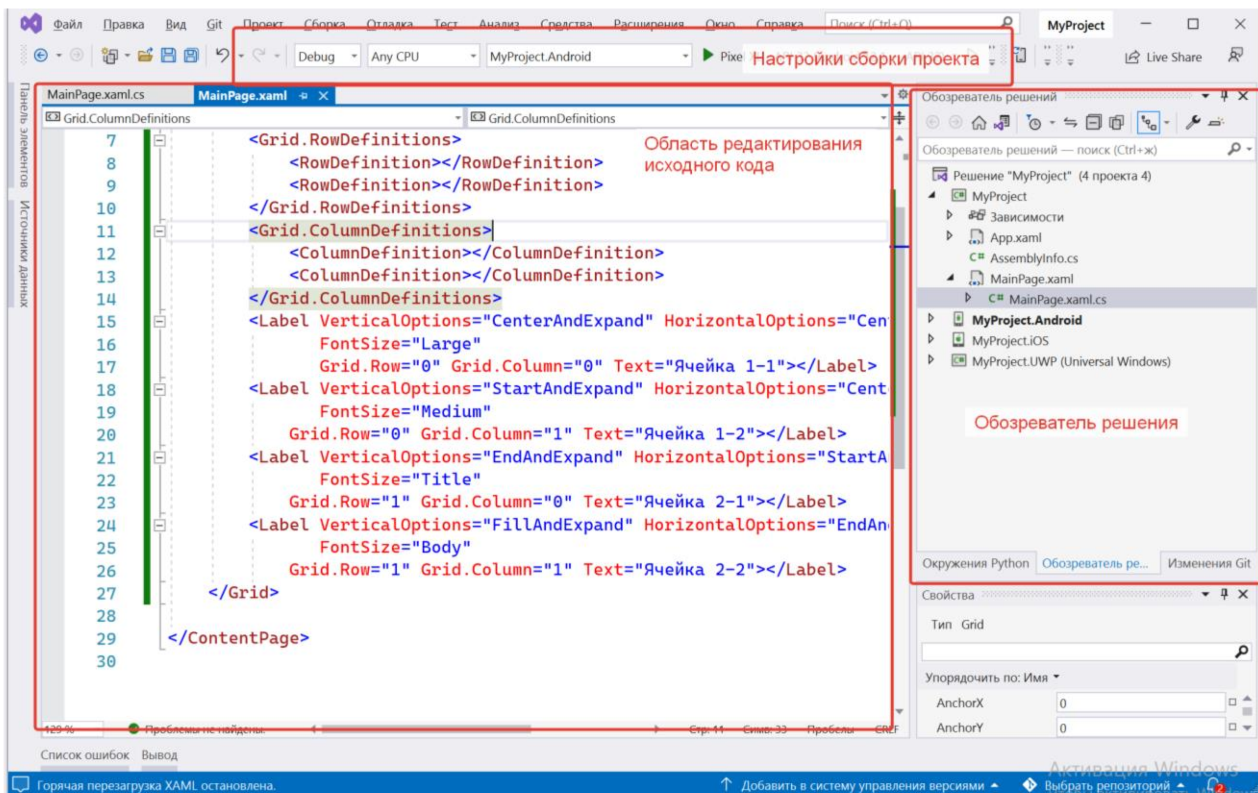

Рис. 16. Главное окно среды разработки

Обозреватель решений нужен для выполнения работ с созданным решением: добавление/удаление из него проектов, просмотр файлов и взаимодействие с кодом. Создавая или открывая приложение или просто отдельный файл, система Visual Studio использует концепцию *решения (solution)* для связывания всех компонентов в единое целое. Как правило, решение состоит из одного или нескольких проектов, каждый из которых содержит множество элементов, связанных с ним.

2.2.2. Структура проекта Xamarin

В нашем случае созданное решение состоит из четырех проектов (если были выбраны все три доступные для разработки платформы): общий с именем проекта (MyProject) и трех для соответствующих платформ (MyProject.Android, MyProject.iOS, MyProject.UWP) (рис. 17).

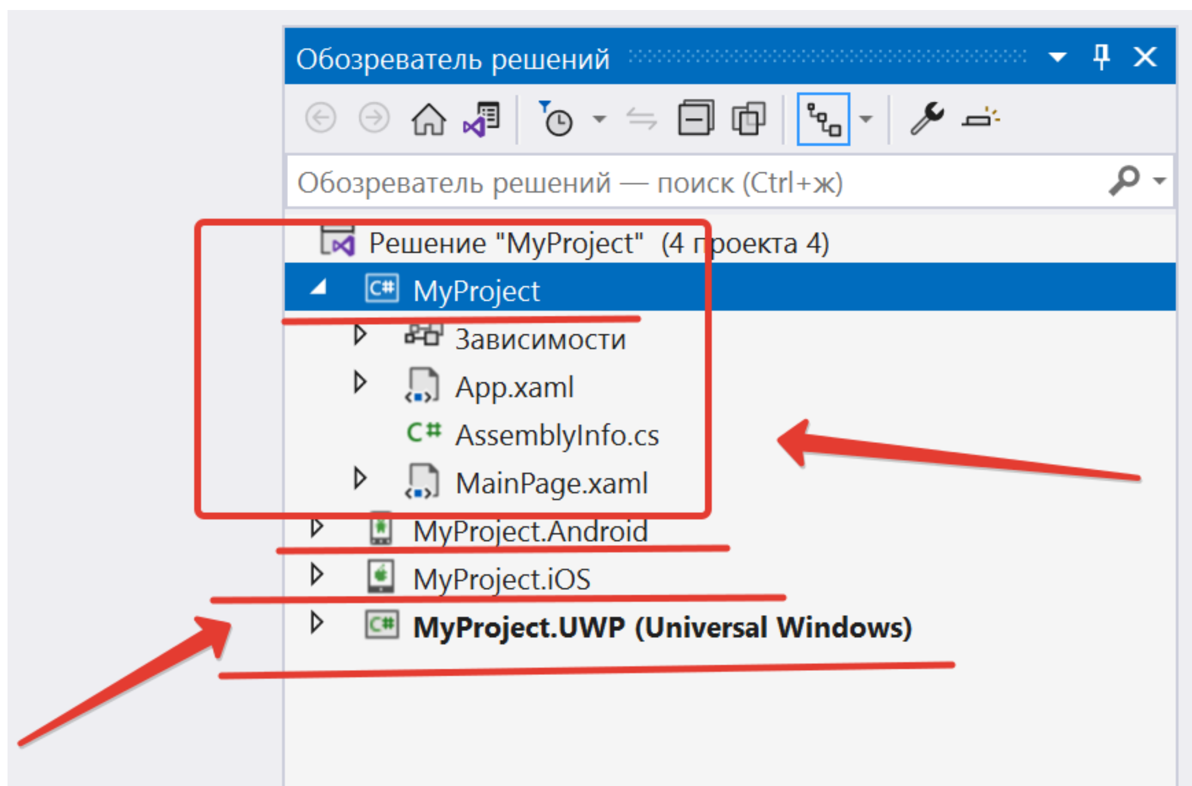

Рис. 17. Структура решения

Главным является самый верхний проект (MyProject). Он обычно содержит весь код и определение интерфейса, которые затем будут использовать все остальные проекты. По умолчанию в нем генерируются следующие файлы[30]:

1. App.xaml – файл, который определяет общие ресурсы для всего приложения. Данная разметка не имеет визуального представления и не является страницей приложения. В файле App.xaml вы можете хранить данные и настройки

[30] Пугачев С.В., Шериев А.М., Кичинский К.А. Разработка приложений для Windows 8 на языке C#. СПб. : БХВ-Петербург, 2013. 416 с.

для всего приложения. Кроме того, в данном файле удобно определять стили и подключать ресурсы, используемые на нескольких страницах приложения, хотя это и не обязательно.

2. App.xaml.cs – файл с кодом C#, с которого начинается выполнение приложения. В данном файле обрабатываются события уровня приложения, такие как запуск, активация, в том числе активация по поисковому запросу, и деактивация приложения. Кроме того, в App.xaml.cs вы можете перехватывать необработанные исключения и отлавливать ошибки навигации между страницами. Также в этом файле задается «первая» страница приложения.

3. MainPage.xaml – файл с визуальным интерфейсом для единственной страницы MainPage в виде XAML. Название страницы выбрано произвольно, к нему не предъявляется никаких требований. Важно только указать в файле App.xaml.cs, какую страницу первой увидит пользователь. По умолчанию в качестве такой страницы указана MainPage.xaml.

4. MainPage.xaml.cs – файл, который содержит логику MainPage на языке C#. Если вам требуется добавить на какую-либо страницу код, вы можете использовать для этого файл с расширением xaml.cs. В данном случае это файл MainPage.xaml.cs. Однако существуют и другие подходы, например паттерн проектирования MVVM (Model-View-ViewModel), когда файл кода страницы оказывается почти пустым, а бизнес-логика определяется в других классах.

5. AssemblyInfo.cs – файл с кодом на языке C#, который используется для установки настроек приложения.

По двойному щелчку на файле в обозревателе решения файл открывается в центральной области среды разработки.

Внимание! *Xamarin.Forms не поддерживает предварительную демонстрацию кода (превью), однако можно воспользоваться функцией горечей перезагрузки. Горячая перезагрузка при сохранении XAML-файла отражает все изменения в работающем приложении. Кроме того, будут поддерживаться состояние навигации и данные.*

2.2.3. Отладка проекта

Далее обратим внимание на функции компиляции и сборки приложения. Разработчик может выбрать систему, под которую будет собираться проект (рис. 18). По умолчанию предлагается сделать сборку под Windows (UWP).

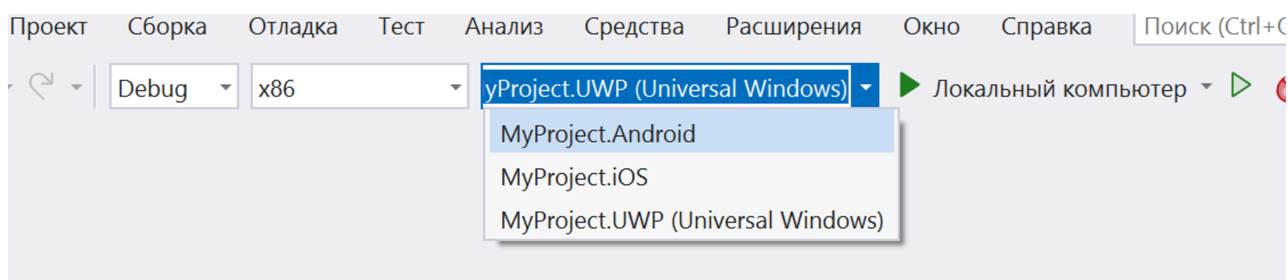

Рис. 18. Выбор платформы для сборки

Внимание! *Компиляция под UWP – это единственный вариант тестирования приложения без участия эмуляторов. Компиляция под Android требует предварительной установки и настройки эмулятора, а запуск по iOS возможен только при наличии физического устройства.*

Запустим проект под UWP. Через некоторое время откроется окно приложения (рис. 19).

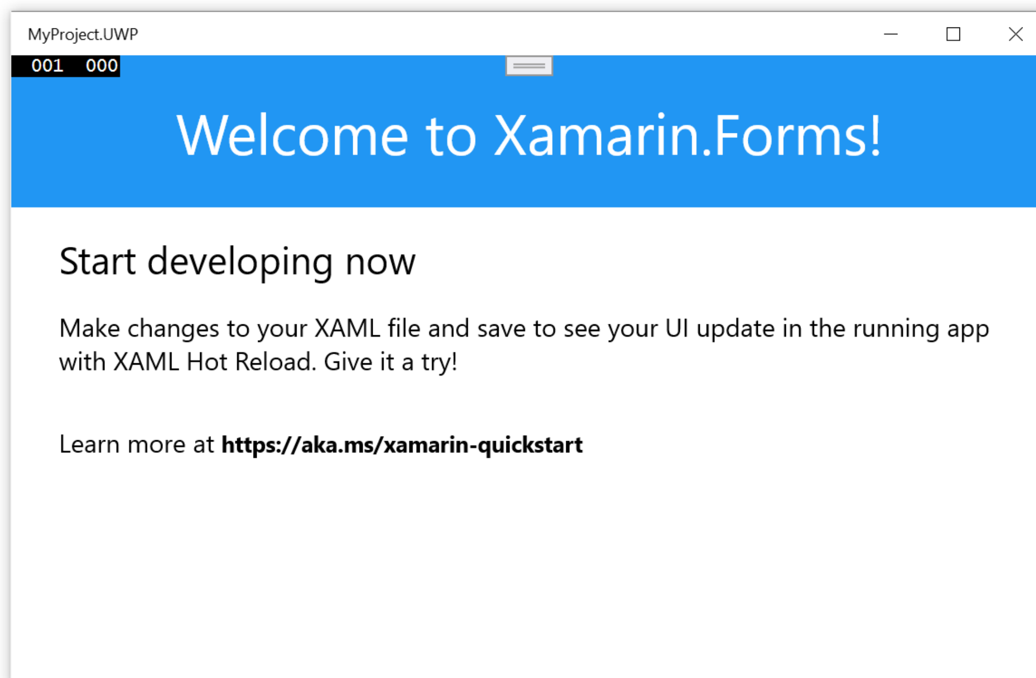

Рис. 19. Выполняющееся приложение под Windows

Если мы выберем иную платформу (например, Android), то процесс запуска пойдет чуть сложнее и дольше, особенно если система не поддерживает аппаратное ускорение эмуляторов. Обязательно обратите внимание на сообщение (рис. 20). Если на компьютере не активировано ускорение hyper-v, включите его.

Внимание! *У процессора должна быть поддержка аппаратной виртуализации. Не забудьте включить ее в BIOS. У Intel эта технология называется Intel-VT (может обозначаться как VMX), а у AMD – AMD-V (SVM). Hyper-v доступен только в 64-разрядной Windows, не ниже профессионального издания.*

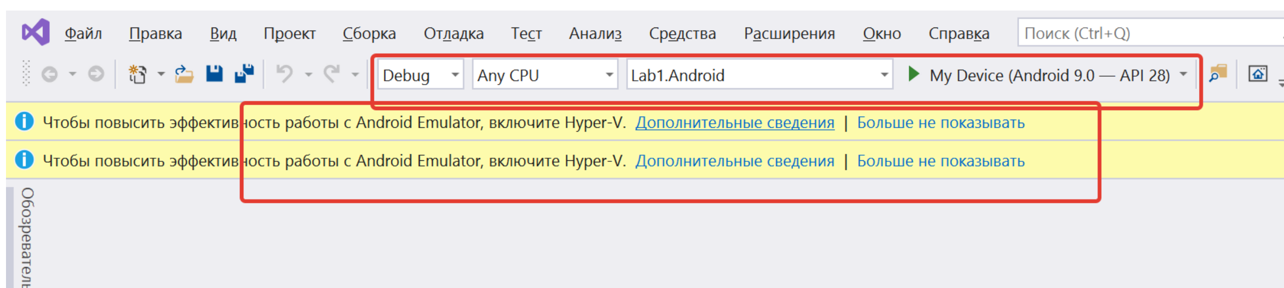

Рис. 20. Включение аппаратного ускорения

Затем можно выбрать один из доступных в системе эмуляторов. Используя «диспетчер устройств Android» можно создать несколько разных устройств с отличающимися характеристиками (рис. 21).

Вначале происходит запуск эмулятора, а затем уже «устанавливается» и «запускается» собираемое приложение (рис. 22).

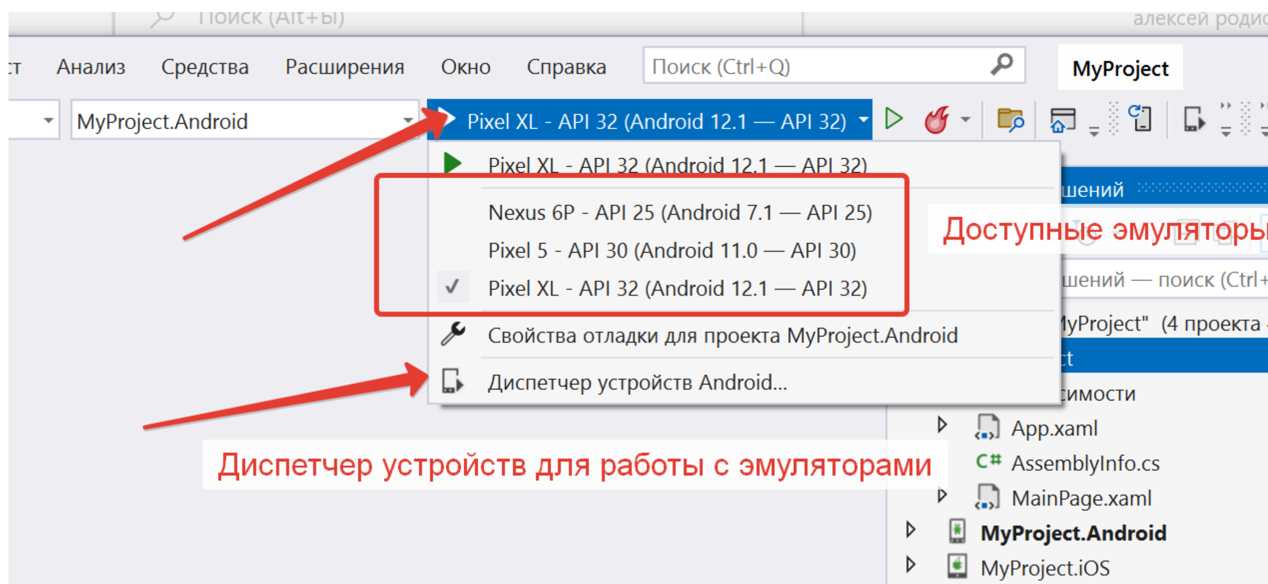

Рис. 21. Выбор и настройка эмуляторов

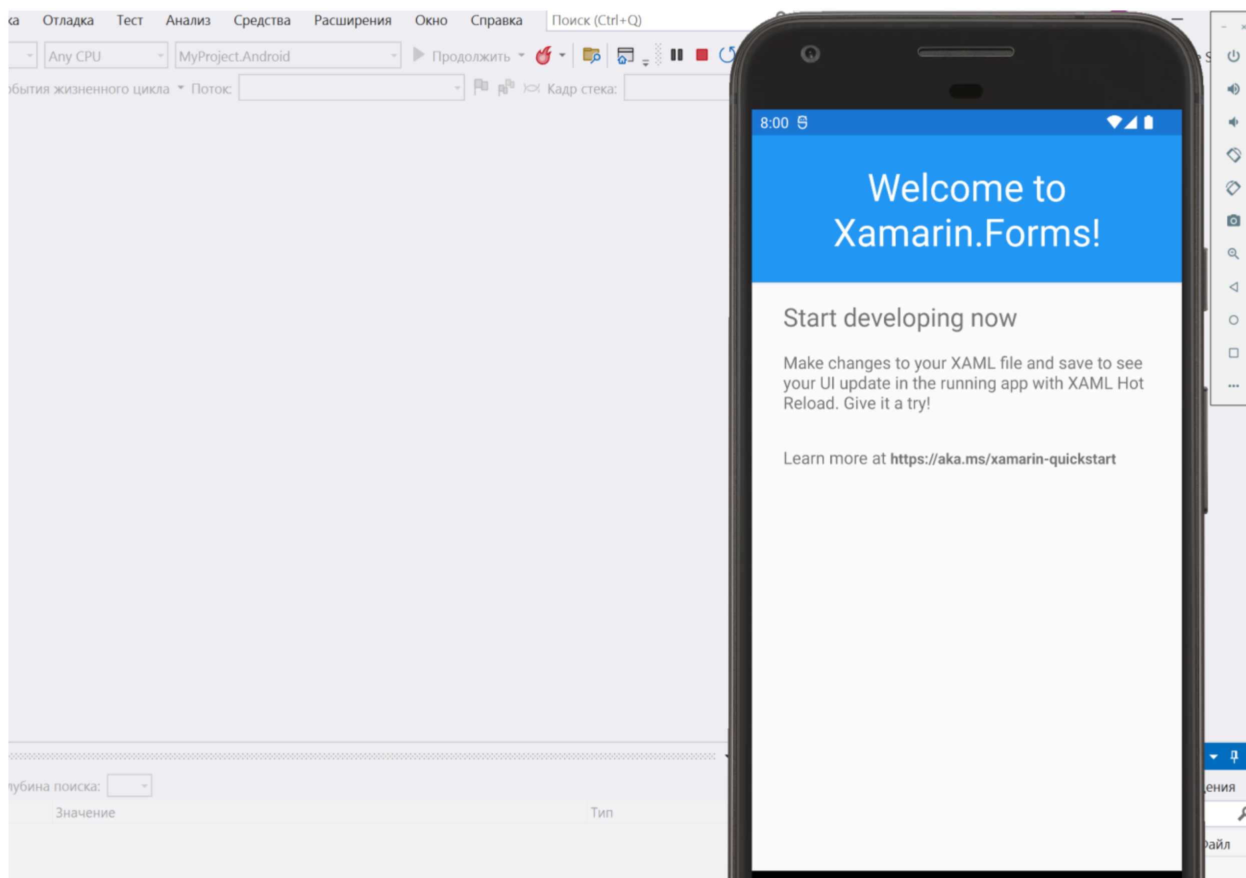

Рис. 22. Выполняющееся приложение под Android в эмуляторе

2.3. Язык eXtensible Application Markup Language

2.3.1. Основы eXtensible Application Markup Language

Язык eXtensible Application Markup Language (XAML) – это язык на основе XML, созданный корпорацией Microsoft в качестве альтернативы программированию кода для создания экземпляров и инициализации объектов, а также организации этих объектов в иерархиях типа «родители – потомки». XAML был адаптирован к нескольким технологиям в .NET Framework, но самую большую роль он играет в определении макета пользовательских интерфейсов в Windows Presentation Foundation (WPF), Silverlight, универсальной платформы Windows (UWP) и Xamarin. Кроме того, XAML хорошо подходит для реализации популярной архитектуры приложений MVVM (Model-View-ViewModel): XAML определяет представление, связанное с кодом ViewModel через привязки данных на основе XAML.

В XAML-файле Xamarin.Forms разработчик может определять пользовательские интерфейсы, используя все доступные представления, макеты и страницы, а также пользовательские классы. XAML может быть скомпилирован или внедрен в исполняемый файл. XAML имеет несколько преимуществ по сравнению с эквивалентным кодом:

– часто является более лаконичным и удобочитаемым, чем эквивалентный код;

– иерархия типа «родители – потомки», встроенная в XML, позволяет XAML имитировать более наглядную иерархию «родители – потомки» объектов пользовательского интерфейса;

– XAML-код может быть написан как программистами, так и дизайнерами, что возможно благодаря наличию средств визуального проектирования.

Существуют и недостатки, в основном связанные с ограничениями, встроенными в языки разметки. XAML не может содержать:

– код (все обработчики событий должны быть определены в файле кода);

– циклы для повторяющейся обработки (однако некоторые элементы, в частности *ListView*, могут создавать несколько дочерних элементов на основе объектов в его *ItemsSource*-коллекции);

– условную обработку (однако привязка данных может ссылаться на преобразователь привязки на основе кода, который допускает условную обработку).

Кроме того, XAML не может создавать экземпляры классов, не определяющих конструктор без параметров, а также вызывать методы.

Таким образом, можно сказать, что XAML – это, по сути, XML, в котором есть некоторые уникальные функции синтаксиса. Наиболее важные из них – элементы свойств, вложенные свойства, расширения разметки.

2.3.2. Структура файла eXtensible Application Markup Language

Файл с разметкой XAML представляет собой обычный файл xml. Откроем файл MainPage.xaml, который содержит разметку страницы приложения. В

Xamarin визуальный интерфейс состоит из страниц, все, что пользователь видит на экране устройства, – это страница. Приложение может иметь одну или несколько страниц. Первой строкой идет стандартное определение xml-файла:

```
<?xml version="1.0" encoding="utf-8" ?>
```

Далее в файле определен элемент *ContentPage*, который представляет обычную страницу (про другие типы страниц см. *2.6.3. Типы страниц в Xamarin.Forms*):

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       x:Class="MyProject.MainPage">
  <StackLayout>
    <Frame BackgroundColor="#2196F3" Padding="24" CornerRadius="0">
      <Label Text="Welcome to Xamarin.Forms!"
HorizontalTextAlignment="Center" TextColor="White" FontSize="36"/>
    </Frame>
    <Label Text="Start developing now" FontSize="Title"
Padding="30,10,30,10"/>
    <Label Text="Make changes to your XAML file and save to see your UI
update in the running app with XAML Hot Reload. Give it a try!"
FontSize="16" Padding="30,0,30,0"/>
    <Label FontSize="16" Padding="30,24,30,0">
      <Label.FormattedText>
        <FormattedString>
          <FormattedString.Spans>
            <Span Text="Learn more at "/>
            <Span Text="https://aka.ms/xamarin-quickstart"
FontAttributes="Bold"/>
          </FormattedString.Spans>
        </FormattedString>
      </Label.FormattedText>
    </Label>
  </StackLayout>
</ContentPage>
```

В определении корневого элемента *ContentPage* подключаются пространства имен с помощью атрибутов `xmlns`. Только одно пространство имен может быть базовым, это пространство используется с префиксом x: `xmlns:x`. Это значит, что те свойства элементов, которые заключены в этом пространстве имен, будут использоваться с префиксом x – `x:Name` или `x:Class`.

Атрибут `x:Class="MyProject.MainPage"` указывает на класс, представляющий данную страницу. Откроем файл MainPage.xaml.cs:

```
using Xamarin.Forms;

namespace MyProject
{
  public partial class MainPage : ContentPage
  {
    public MainPage()
    {
      InitializeComponent();
    }
  }
}
```

Класс *MainPage* является производным от класса *ContentPage* (наследуется), при этом оно определено как *partial* – частичный класс. Объявив класс частичным, мы можем иметь несколько файлов с определением одного и того же класса, и при компиляции все эти определения будут скомпилированы в одно. Второй файл создается при сборке проекта на основе файла xaml: Visual Studio анализирует XAML-файл для создания файла кода C# (в данном случае MainPage.xaml) и на его основе генерирует вторую часть класса (в виде файла с кодом доступна для просмотра в папке obj\Debug-решения, он имеет название MainPage.xaml.g.cs, где «g» означает «созданный») (рис. 23).

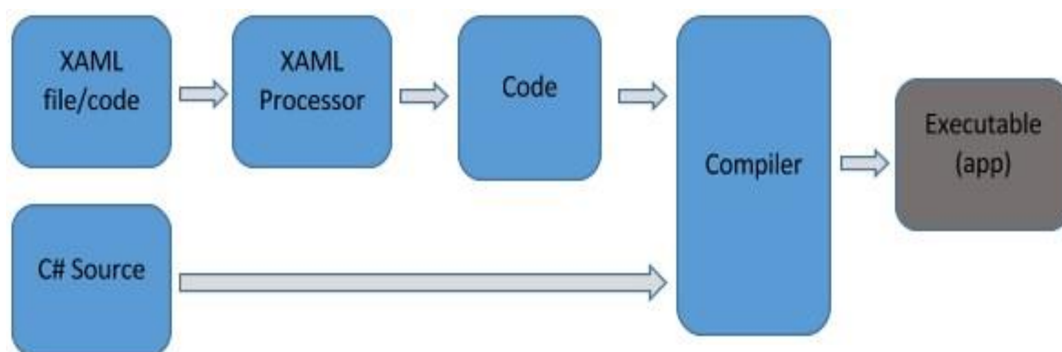

Рис. 23. Схема «сборки» страницы

Внутри элемента *ContentPage* могут быть объявлены элементы интерфейса – по умолчанию мы видим элемент компоновки *StackLayout*[31], внутри которого уже размещаются другие пользовательские элементы (контейнер *Frame*, надписи *Label*)[32].

---

[31] Про элементы компоновки будет подробно рассказано в параграфе 3.2.

[32] Обзор пользовательских элементов дан в параграфе 3.1.

2.3.3. Элементы свойств

XAML предлагает декларативную схему определения различных элементов и их свойств. Каждый элемент, как и любой элемент XML, должен иметь открывающий и закрывающий тег, например:

```
<Label></Label>
```

Допустимо использовать сокращенную запись:

```
<Label/>
```

Каждый элемент в XAML представляет объект определенного класса C#, а атрибуты элементов соотносятся со свойствами этих классов. В XAML свойства классов обычно устанавливаются в виде XML-атрибутов:

```
<Label Text="Make changes to your XAML file and save to see your UI
update in the running app with XAML Hot Reload. Give it a try!"
FontSize="16"
Padding="30,0,30,0">
</Label>
```

В этом коде *Label* является элементом объекта. Это объект Xamarin.Forms, выраженный как элемент XML. *Text, FontSize, Padding* являются атрибутами свойств. Они представляют собой Xamarin.Forms-свойства, выраженные в виде XML-атрибутов.

Есть и второй вариант записи, который принято называть «синтаксис элемента свойства»:

```
<Label>
      <Label.Text>
        Второй вариант
      </Label.Text>
      <Label.FontSize>
        16
      </Label.FontSize>
</Label>
```

Данный синтаксис может показаться избыточным, но синтаксис элемента свойства становится очень важен, если значение свойства слишком сложное, чтобы оно было выражено как простая строка. В тегах элементов свойств можно создать экземпляр другого объекта и задать его свойства.

Например, очень часто в проектах используется элемент *Grid*. *Grid* имеет два свойства с именами *RowDefinitions* и *ColumnDefinitions*. Эти свойства имеют

тип *RowDefinitionCollection* и *ColumnDefinitionCollection*, которые являются коллекциями *RowDefinition* объектов и *ColumnDefinition*. Для задания этих коллекций необходимо использовать синтаксис элемента свойства.

2.3.4. Присоединенные свойства

Для указания, в какой ячейке размещаются элементы в таблице (*Grid*), используются так называемые «присоединенные свойства». Для создания присоединенного свойства в теге для каждого дочернего элемента *Grid* нужно указать строку и столбец этого дочернего объекта, используя следующие атрибуты (значения этих атрибутов по умолчанию равны 0): *Grid.Row* и *Grid.Column.*

Присоединенные свойства задаются в XAML-файлах как атрибуты, содержащие класс и имя свойства, разделенные точкой. Они называются *присоединенными*, так как определяются одним классом (в данном случае *Grid*), но присоединены к другим объектам (в данном случае дочерним элементам *Grid*).

Можно также указать, что дочерний элемент охватывает более одной строки или столбца с этими атрибутами: *Grid.RowSpan* и *Grid.ColumnSpan.*

Эти два атрибута имеют значения по умолчанию 1.

Практический пример 1.
Создание одностраничного приложения

Задача. Создайте проект «Мобильное приложение (Xamarin)». Приложение должно содержать таблицу с двумя строками и двумя столбцами, в каждой ячейке таблицы должна быть размещена надпись.

Решение.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       x:Class="MyProject.MainPage">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition></RowDefinition>
      <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition></ColumnDefinition>
      <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Grid.Column="0" Text="Ячейка 1-1"></Label>
    <Label Grid.Row="0" Grid.Column="1" Text="Ячейка 1-2"></Label>
    <Label Grid.Row="1" Grid.Column="0" Text="Ячейка 2-1"></Label>
    <Label Grid.Row="1" Grid.Column="1" Text="Ячейка 2-2"></Label>
  </Grid>
</ContentPage>
```

Скриншот работающей программы представлен на рис. 24.

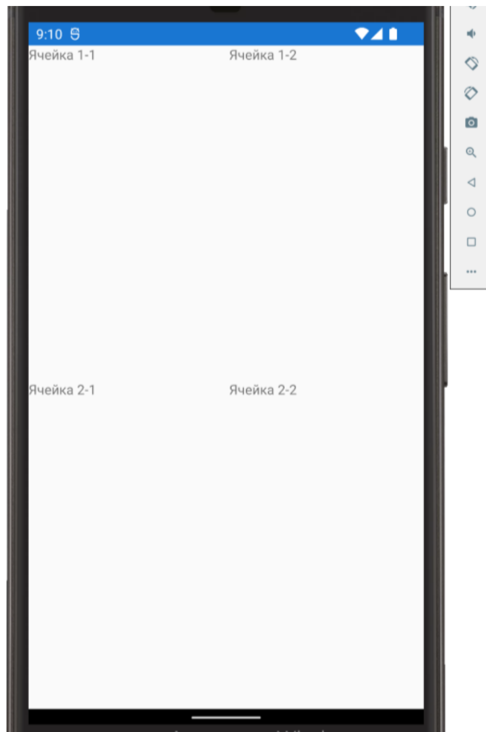

Рис. 24. Результат выполнения приложения

2.3.5. Позиционирование элементов

Позиционирование элементов в XAML задается относительно родительского элемента (элемента-контейнера). Для позиционирования элементов на странице можно использовать два типа отступов: внутренние и внешние (рис. 25).

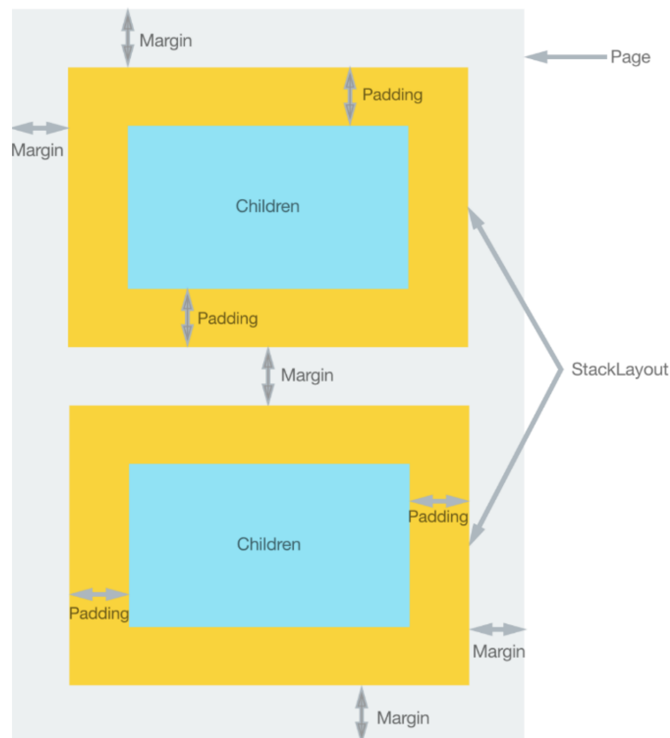

Рис. 25. Схема позиционирования элементов

Атрибут *Padding* устанавливает внутренние отступы – от внутреннего содержимого элемента до его границ. Атрибут *Margin* определяет внешний отступ элемента от других элементов или контейнера. Оба этих свойства определяются через структуру *Thickness*, которая имеет свойства *Left, Top, Right, Bottom*. В результате задать ее определение мы можем тремя способами:

– указать одно значение для отступов со всех сторон;

– указать два значения для отступов по горизонтали (слева и справа) и вертикали (сверху и снизу);

– указать четыре значения для отступов для каждой из сторон.

2.3.6. Выравнивание элементов

Выравнивание элементов в XAML задается относительно родительского элемента (элемента-контейнера). Все элементы, используемые при создании интерфейса, имеют два свойства – *HorizontalOptions* и *VerticalOptions*. Они управляют выравниванием элемента соответственно по горизонтали и по вертикали.

В качестве значения они принимают структуру *LayoutOptions*[33]. Данная структура имеет ряд свойств, которые хранят объекты опять же *LayoutOptions*:

– *Start* – выравнивание по левому краю (выравнивание по горизонтали) или по верху (выравнивание по вертикали);

– *Center* – элемент выравнивается по центру;

– *End* – выравнивание по правому краю (выравнивание по горизонтали) или по низу (выравнивание по вертикали);

– *Fill* – элемент заполняет все пространство контейнера;

– *StartAndExpand* – аналогичен опции *Start* с применением растяжения;

– *CenterAndExpand* – аналогичен опции *Center* с применением растяжения;

– *EndAndExpand* – аналогичен опции *End* с применением растяжения;

– *FillAndExpand* – аналогичен опции *Fill* с применением растяжения.

Практический пример 2.
Позиционирование и выравнивание элементов на форме

Задача. Внесите изменения в проект, созданный в первом примере, продемонстрируйте разные способы выравнивания надписей, измените размеры надписей.

Решение.

Внесем изменение в пример с таблицей, добавим выравнивание (атрибуты *VerticalOptions* и *HorizontalOptions*), изменим размер текста надписей (атрибут *FontSize*):

```
...
<Label VerticalOptions="CenterAndExpand"
HorizontalOptions="CenterAndExpand"
        FontSize="Large"
```

[33] Microsoft Learn.

```
        Grid.Row="0" Grid.Column="0" Text="Ячейка 1-1"></Label>
    <Label VerticalOptions="StartAndExpand"
HorizontalOptions="CenterAndExpand"
        FontSize="Medium"
      Grid.Row="0" Grid.Column="1" Text="Ячейка 1-2"></Label>
    <Label VerticalOptions="EndAndExpand"
HorizontalOptions="StartAndExpand"
        FontSize="Title"
      Grid.Row="1" Grid.Column="0" Text="Ячейка 2-1"></Label>
    <Label VerticalOptions="FillAndExpand"
HorizontalOptions="EndAndExpand"
        FontSize="Body"
      Grid.Row="1" Grid.Column="1" Text="Ячейка 2-2"></Label>
...
```

Результат представлен на рис. 26.

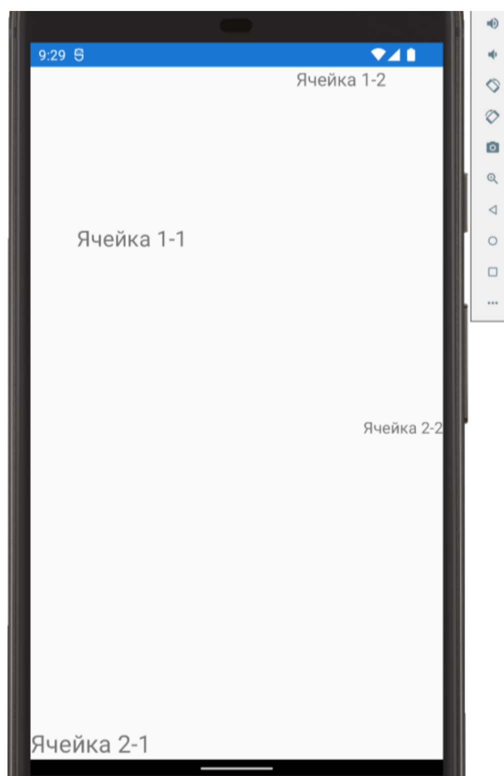

Рис. 26. Выравнивание элементов

2.3.7. Взаимодействие с кодом приложения

Файлы XAML позволяют нам определить визуальный интерфейс, но для создания логики, например для определения обработчиков событий элементов управления, все равно придется воспользоваться кодом C#. И вместе с файлом разметки XAML MainPage.xaml Visual Studio по умолчанию также создает файл отделенного кода MainPage.xaml.cs, который содержит логику на C#, связанную с файлом MainPage.xaml.

Внимание! *Нередко в приложении требуется обратиться к какому-нибудь элементу управления, который определен в коде XAML. Для этого надо установить у элемента в XAML атрибут x:Name.*

Часто для произведения какого-либо действия в приложении используются кнопки, представленные классом *Button*. Кнопка является базовым элементом управления Xamarin.Forms и обычно отображается как прямоугольник, в котором содержится короткая текстовая строка с названием выполняемой операции/команды (при этом кнопка может отображать растровое изображение и сочетать текст и изображение). Пользователь может нажать кнопку пальцем (если мобильное устройство поддерживает сенсорный интерфейс) или щелкнуть по ней мышью, чтобы инициировать команду.

Для кнопки можно задать обработчик нажатия для события *Clicked*. Событие возникает при отпускании пальца или кнопки мыши с кнопки.

При отображении кнопки на форме она занимает все пространство, которое будет доступно/разрешено (по умолчанию *Button* будет занимать полную ширину и высоту родительского элемента).

Базовым свойством кнопки является *Text*, которое позволяет задать текст, отображаемый в *Button*.

Практический пример 3.
Реализация обработчика событий

Задача. Внесите изменения в проект, созданный в первом примере, добавьте кнопку и обработчик события нажатия кнопки. При нажатии должны измениться все надписи.

Решение.

Внесем в наш предыдущий пример дополнение, добавим еще одну строку в таблицу, объединим в ней ячейки и поместим в нее кнопку. Также добавим к текстовым полям имена с помощью атрибута *x:Name*.

Для создания обработчика событий разместите элемент *Button* в последней строке таблицы, введите атрибут *Clicked,* поставьте двойные кавычки и нажмите пробел – среда разработки автоматически предложит добавить новый обработчик (или выбрать существующий, если он уже написан). Дважды щелкните по строке «новый обработчик событий» (рис. 27).

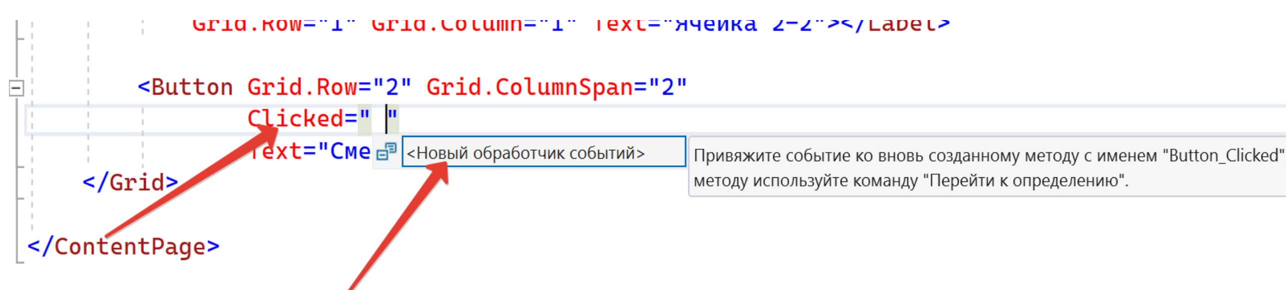

Рис. 27. Создание обработчика событий

Код XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       x:Class="MyProject.MainPage">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition></RowDefinition>
      <RowDefinition></RowDefinition>
      <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition></ColumnDefinition>
      <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label x:Name="l1"
      VerticalOptions="CenterAndExpand"
        HorizontalOptions="CenterAndExpand"
        FontSize="Large"
        Grid.Row="0" Grid.Column="0" Text="Ячейка 1-1"></Label>
    <Label x:Name="l2"
      VerticalOptions="StartAndExpand"
        HorizontalOptions="CenterAndExpand"
        FontSize="Medium"
      Grid.Row="0" Grid.Column="1" Text="Ячейка 1-2"></Label>
    <Label x:Name="l3"
      VerticalOptions="EndAndExpand"
        HorizontalOptions="StartAndExpand"
        FontSize="Title"
      Grid.Row="1" Grid.Column="0" Text="Ячейка 2-1"></Label>
    <Label x:Name="l4"
      VerticalOptions="FillAndExpand"
        HorizontalOptions="EndAndExpand"
        FontSize="Body"
      Grid.Row="1" Grid.Column="1" Text="Ячейка 2-2"></Label>

    <Button Grid.Row="2" Grid.ColumnSpan="2"
        Clicked="Button_Clicked"
        Text="Сменить текст надписей"></Button>
  </Grid>
</ContentPage>
```

Теперь для кнопки создадим обработчик событий. Откроем файл MainPage.xaml.cs. Мы увидим, что в классе для обработки события нажатия кнопки создана процедура *private void Button_Clicked (object sender, System.EventArgs e).*

Изменим файл следующим образом:

```
public partial class MainPage : ContentPage
  {
```

```
    public MainPage()
    {
      InitializeComponent();
    }

    private void Button_Clicked(object sender, System.EventArgs e)
    {
      l1.Text = "Новый текст";
      l2.Text = "Новый текст";
      l3.Text = "Новый текст";
      l4.Text = "Новый текст";
    }
  }
```

Результат нажатия кнопки представлен на рис. 28.

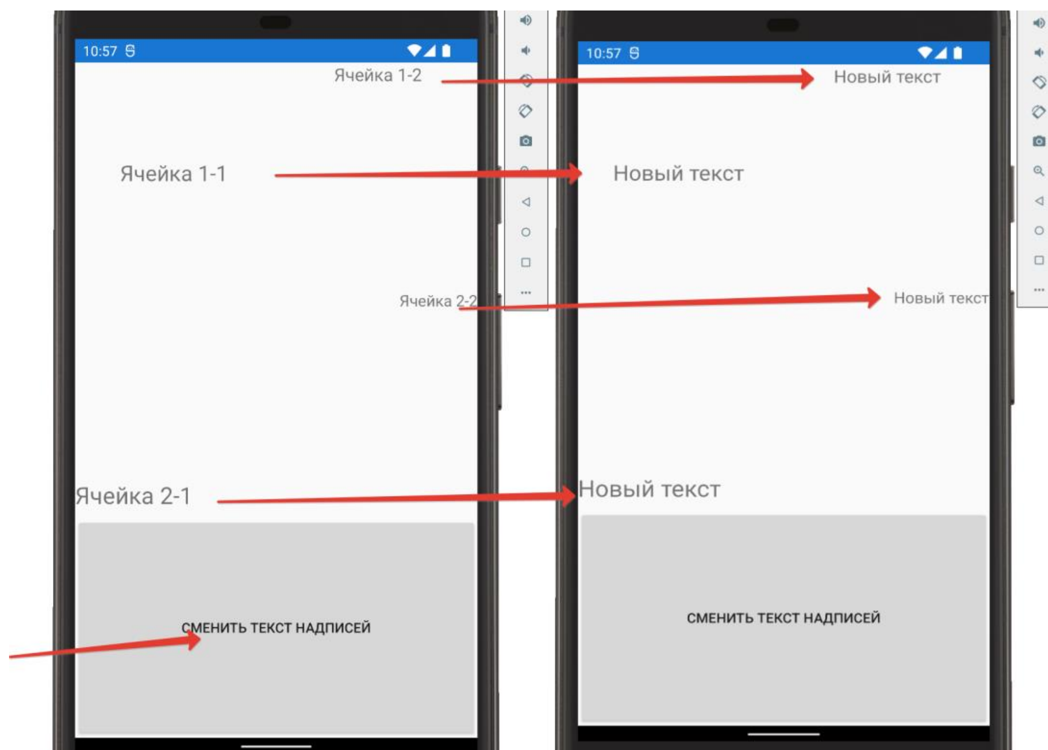

Рис. 28. Приложение с кнопкой

Контрольные вопросы

1. Основные компоненты Xamarin.Forms.
2. Поддерживаемые платформы Xamarin.Forms.
3. Структура проекта Xamarin.Forms.
4. Язык XAML. Определение, назначение, примеры применения.
5. Основы синтаксиса XAML.
6. Объектные элементы XAML, синтаксис атрибутов (свойства), синтаксис элемента свойства.
7. События и код программной части XAML. Взаимодействие с кодом приложения.

8. Расширение разметки XAML.

9. Префиксы. Префикс «x:». Пользовательские префиксы и пользовательские типы.

10. Присоединенные свойства и присоединенные события.

11. Различия платформ с OnPlatform (Device).

12. Методы создания общего кода.

13. Жизненный цикл приложения. Основные события.

14. Основы позиционирования и выравнивания элементов.

3. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС В XAMARIN.FORMS

Пользовательский интерфейс при применении фреймворка Xamarin.Forms создается из объектов, которые сопоставляются с собственными элементами управления каждой целевой платформы. Это позволяет приложениям для конкретных платформ (iOS, Android и UWP) использовать Xamarin.Forms-код, содержащийся в .NET стандартной библиотеке.

Когда приложение запускается на любой платформе, оно отображает один экран, соответствующий странице в Xamarin.Forms. Страница Xamarin.Forms обычно занимает весь экран, а все типы страниц являются производными от класса *Page*. Как правило, страницы содержат макет, а все типы макетов являются производными от класса *Layout*. Макет обычно включает представления и, возможно, другие макеты, а все типы представлений в конечном итоге являются производными от класса *View*. Наконец, ячейки – это специализированные элементы управления, которые используются при отображении данных в *TableView*- и *ListView*-элементах управления. Страницы, макеты, представления и ячейки в конечном счете являются производными от класса *Element*.

Достаточно часто макеты называются элементами компоновки, а представления – элементами управления. Рассмотрим основные из них.

3.1. Основные элементы управления

3.1.1. Элементы для работы с текстом

В Xamarin для ввода/вывода текстовой информации можно использовать три элемента: *Label, Entry и Editor*.

Элемент *Label*[34] представляет обычную текстовую метку, которая выводит информацию с помощью свойства *Text*. *Label* удобен для создания заголовков и меток к элементам ввода. Простое создание элемента:

```
...
<Label Text="Это просто надпись" HorizontalOptions="Center" />
...
```

Результат выполнения кода представлен на рис. 29.

Элемент *Label* поддерживает настройки, изменяющие внешний вид выводимого текста:

```
    ...
<Label Text="Это надпись золотым цветом, курсивное начертание, размер 22, подчеркнуто"
        TextColor="Gold"
        FontAttributes="Italic"
        FontSize="24"
```

[34] Microsoft Learn.

```
        TextDecorations="Underline"
        HorizontalOptions="Center" />
...
```

Здесь задействованы атрибуты:

– *TextColor* – установка цвета текста;

– *FontAttributes* – установка начертания текста;

– *FontSize* – размер шрифта;

– *TextDecorations* – стиль подчеркивания.

Результат выполнения кода представлен на рис. 30.

Можно объединять несколько форматов в одной надписи, используя свойство *FormattedText* и элемент *Span* (рис. 31).

```
...
<Label TextColor="Blue"
        FontSize="Large">
        <Label.FormattedText>
          <FormattedString>
            <Span Text="Текст 1 " />
            <Span Text="Текст 2" FontAttributes="Bold" />
            <Span Text="Текст 3" TextDecorations="Strikethrough" />
          </FormattedString>
        </Label.FormattedText>
</Label>
...
```

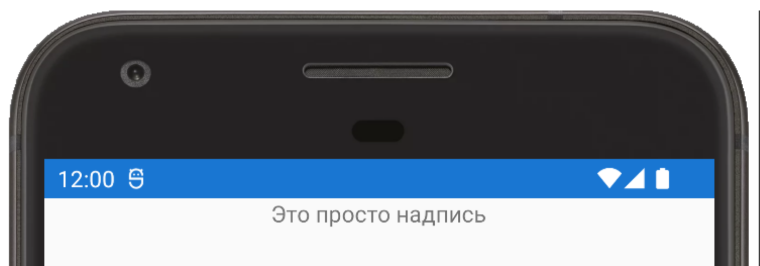

Рис. 29. Пример надписи Label

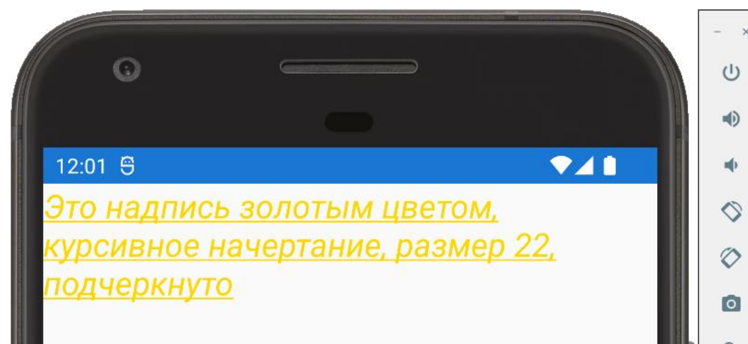

Рис. 30. Пример оформления текста в надписи Label

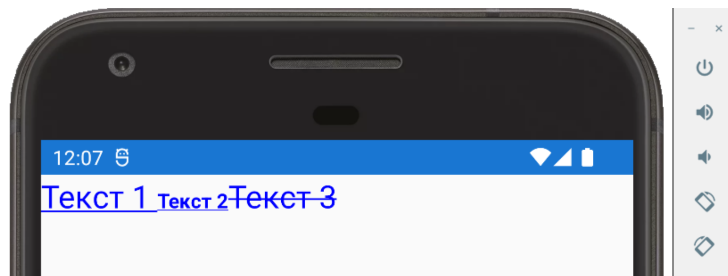

Рис. 31. Несколько форматов в надписи Label

Элемент *Entry*[35] представляет собой текстовое поле для ввода однострочной записи:

```
<Entry Placeholder="Введите текст" />
```

Результат представлен на рис. 32.

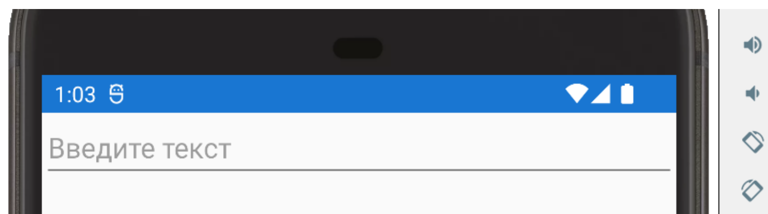

Рис. 32. Элемент Entry

Атрибут *Placeholder* позволяет задать текст-подсказку, который исчезает в момент начала ввода текста в текстовое поле.

Элемент *Entry* имеет ряд настроек поведения. Создадим текстовое поле для ввода пароля:

```
...
<Entry Placeholder="Введите пароль"
      MaxLength="12"
      IsSpellCheckEnabled="false"
      IsTextPredictionEnabled="false"
      IsPassword="true" />
...
```

Результат представлен на рис. 33.

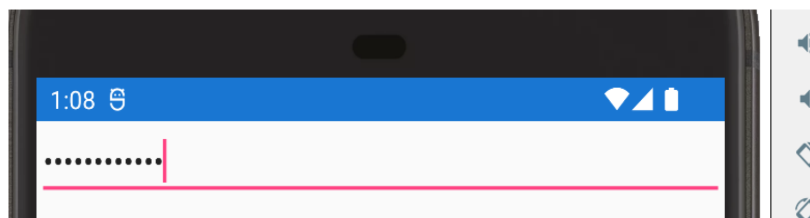

Рис. 33. Текстовое поле для ввода пароля

[35] Microsoft Learn.

Задействованные атрибуты:

– *MaxLength* – ограничение допустимой длины строки;

– *IsSpellCheckEnabled* – отключение автоматической проверки орфографии;

– *IsTextPredictionEnabled* – отключение автоматического дополнения текста;

– *IsPassword* – скрытие вводимого текста.

Кроме этого для ввода текста можно задать вид клавиатуры, который будет наиболее удобен в конкретном случае. Для этого используется свойство *Keyboard*, которое может принимать одно из следующих значений: *Default*, *Text*, *Chat*, *Url*, *Email*, *Telephone*, *Numeric*. Создадим форму для ввода ФИО пользователя и телефонного номера:

```
...
<Entry
	Placeholder="Введите ФИО"
	Keyboard="Text"
/>
<Entry
	Placeholder="Введите телефонный номер, 10 цифр"
	MaxLength="10"
	Keyboard="Telephone">
</Entry>
...
```

Результат представлен на рис. 34.

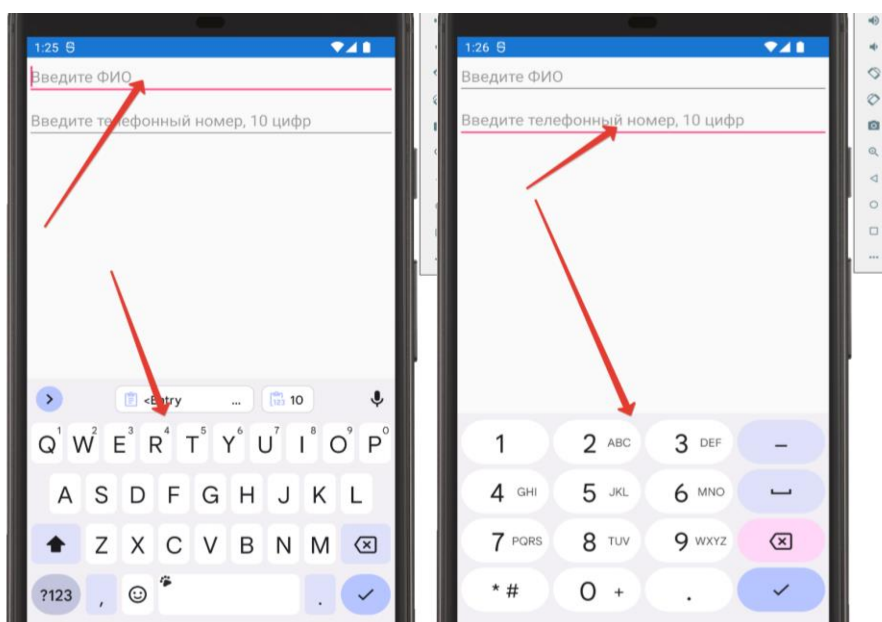

Рис. 34. Использование разных видов клавиатур

Элемент *Editor*[36] представляет собой многострочное поле ввода, принципы его работы аналогичны элементу *Entry*. Создадим поле для многострочного ввода:

[36] Microsoft Learn.

```
...
<Editor
      Placeholder="Поле для ввода многострочного текста"
      HeightRequest="200"
      BackgroundColor="AliceBlue"
      TextColor="Blue"
/>
...
```

Результат представлен на рис. 35.

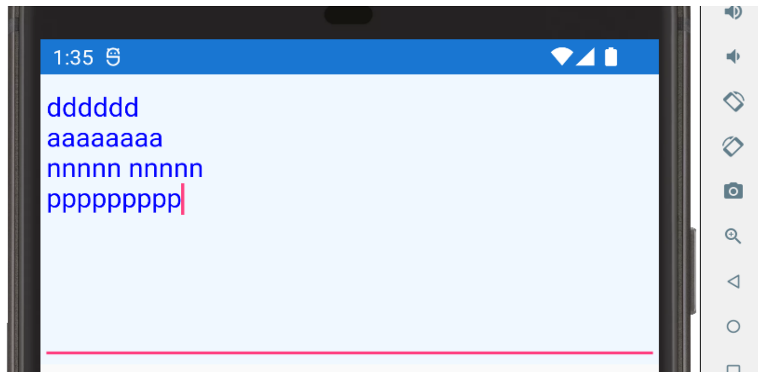

Рис. 35. Поле для многострочного ввода Editor

Задействованные атрибуты:

– *HeightRequest* задает высоту элемента;

– *BackgroundColor* задает фоновый цвет элемента;

– *TextColor* задает цвет шрифта.

Кроме того, высоту элемента можно задать автоматически – свойство *AutoSize*. Если его установить в значение *TextChanges*, то высота элемента *Editor* будет увеличиваться в процессе заполнения текстом, и уменьшаться, когда текст удаляется.

Элементы *Entry* и *Editor* поддерживают два основных обработчика событий: *TextChanged* и *Completed*. Первый возникает при вводе текста, второй – при завершении ввода. Продемонстрируем это на примере, напишем приложение, содержащее три текстовых элемента. При вводе текста в первый элемент он должен дублироваться во втором, но не в третьем. После завершения ввода текст должен появиться в третьем элементе, и исчезнуть во втором.

Решение.

Код XAML:

```
...
<StackLayout>
      <Editor Placeholder="Введите текст"
        TextChanged="Editor_TextChanged"
```

```
        Completed="Editor_Completed"
        ></Editor>
      <Label x:Name="l1"
          Text="Текст во время ввода"
          ></Label>
      <Label x:Name="l2"
          Text="Текст после ввода"
          ></Label>
</StackLayout>
...
```

Код C#:

```
...
private void Editor_TextChanged(object sender, TextChangedEventArgs e)
{
     l1.Text = ((Editor)sender).Text;
}
private void Editor_Completed(object sender, EventArgs e)
{
     l1.Text = "";
     l2.Text = ((Editor)sender).Text;
}
...
```

Результат выполнения программы представлен на рис. 36.

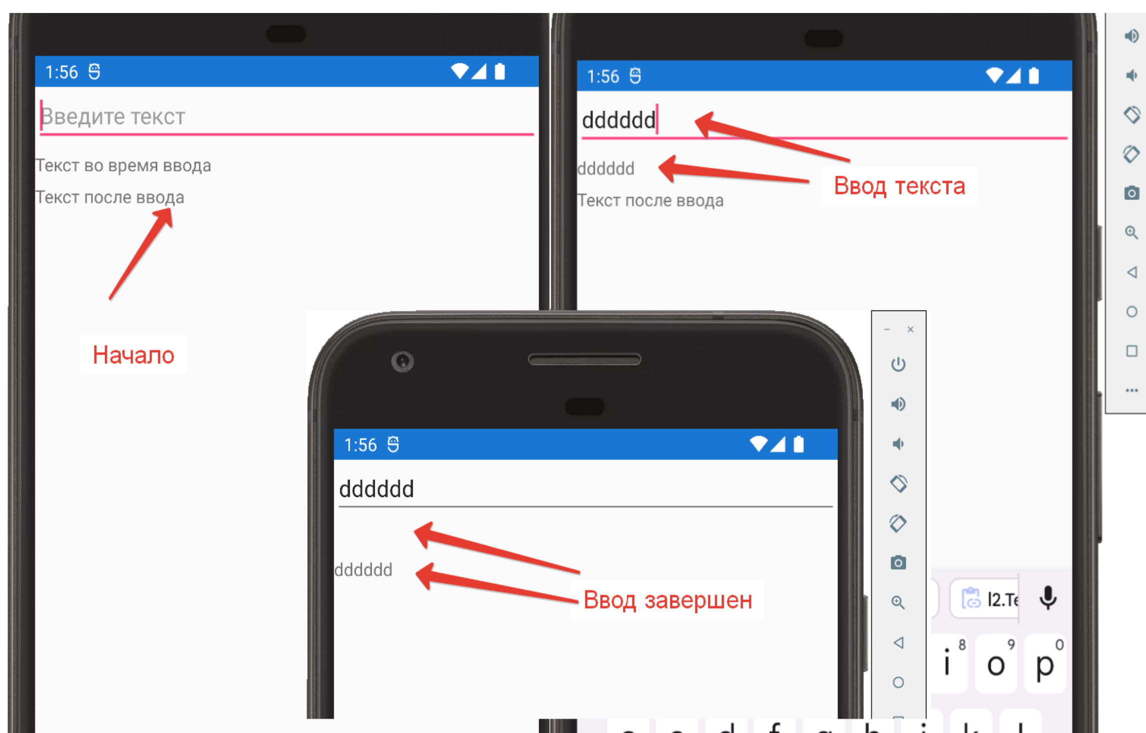

Рис. 36. Результат работы программы

3.1.2. Элементы для работы с числовыми значениями

Для установки числовых значений можно использовать элементы *Stepper* и *Slider*. Элемент *Stepper*[37] позволяет задавать числовые значения дискретно из определенного диапазона. Визуально представляет собой две кнопки, помеченные знаками «–» и «+». Возвращает число, тип данных double. Имеет четыре базовых свойства:

– *Increment* задает число (шаг) для изменения выбранного значения со значением по умолчанию 1;

– *Minimum* – минимум диапазона со значением по умолчанию 0;

– *Maximum* – максимальное значение диапазона со значением по умолчанию 100;

– *Value* – позволяет установить или получить текущее выбранное значение.

Основное событие – *ValueChanged* – позволяет отслеживать изменение значения. Продемонстрируем работу элемента.

Код XAML:

```
...
 <StackLayout>
      <Label x:Name="l1"
        ></Label>
      <Stepper
        Minimum="10"
        Maximum="100"
        Increment="10"
        ValueChanged="Stepper_ValueChanged"
        >
      </Stepper>
    </StackLayout>
...
```

Код C#:

```
...
private void Stepper_ValueChanged(object sender, ValueChangedEventArgs e)
    {
      l1.Text = e.NewValue.ToString();
    }
...
```

Результат работы программы представлен на рис. 37. В данном примере с помощью *Stepper* можно выбирать значения в диапазоне от 10 до 100 с шагом 10.

Элемент *Slider*[38] визуально представляет собой ползунок для выбора из диапазона непрерывных значений. При создании *Slider* можно установить следующие свойства:

[37] Microsoft Learn.

[38] Там же.

– *Minimum* – минимальное значение диапазона со значением по умолчанию 0;

– *Maximum* – максимальное значение диапазона со значением по умолчанию 1;

– *ThumbColor* задает цвет указателя текущего значения;

– *MinimumTrackColor* задает цвет ползунка до указателя значения;

– *MaximumTrackColor* задает цвет ползунка после указателя значения;

– *ThumbImageSource* позволяет задать изображение, используемое как ползунок, в качестве значения используется объект типа *ImageSource*.

Пример задания ползунка.

Код XAML:

```
...
<StackLayout>
      <Label x:Name="l1"
        ></Label>
      <Slider x:Name="sl"
        Maximum="100"
        Minimum="10"
        Value="50"
        ThumbColor="Blue"
        MinimumTrackColor="Red"
        MaximumTrackColor="Black"
        ValueChanged="Slider_ValueChanged"
        >
      </Slider>
</StackLayout>
...
```

Код C#:

```
...
private void Slider_ValueChanged(object sender, ValueChangedEventArgs e)
    {
      l1.Text = e.NewValue.ToString();
    }
    ...
```

Результат работы программы представлен на рис. 38.

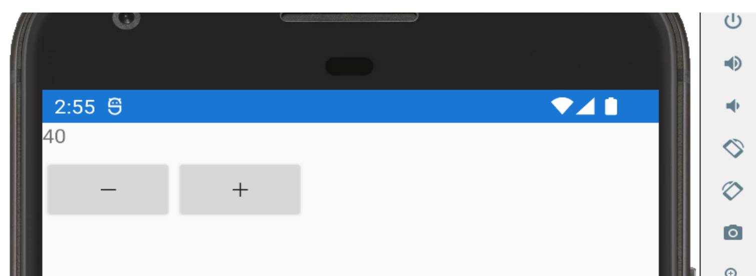

Рис. 37. Установка значения с использованием Stepper

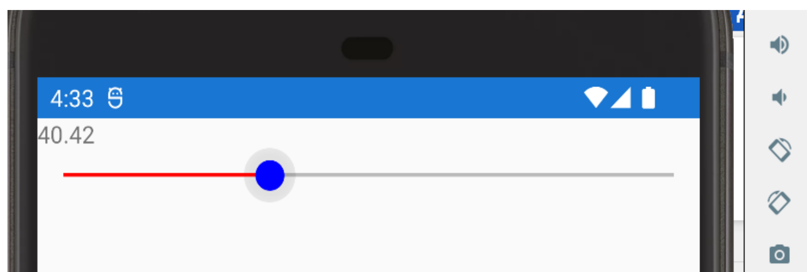

Рис. 38. Установка значения элементом Slider

3.1.3. Элементы оформления

К элементам оформления можно отнести *Frame* и *BoxView*. *Frame*[39] представляет собой контейнер, который может содержать другие элементы и обычно используется для визуального оформления. Основные свойства элемента:

– *BorderColor* позволяет задать цвет границы фрейма с помощью структуры Color;

– *CornerRadius* позволяет задать радиус границы фрейма в виде значения типа float;

– *HasShadow* позволяет установить, будет ли фрейм отбрасывать тень.

Изменим с помощью контейнера *Frame* визуальное оформление элемента Entry, создав поле для ввода текста с закругленными краями:

```
...
<Frame
      BorderColor ="DarkBlue"
      CornerRadius="15"
      HasShadow="True"
      Margin="20"
      >
      <Entry Placeholder="Введите текст"></Entry>
</Frame>
...
```

Результат представлен на рис. 39.

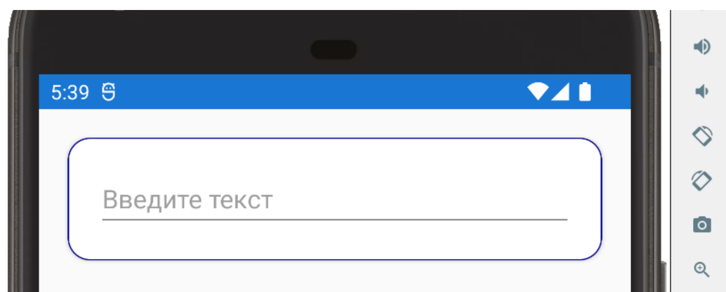

Рис. 39. Элемент для ввода текста с закругленными краями

[39] Microsoft Learn.

BoxView[40] позволяет отобразить на форме простой прямоугольник указанной ширины, высоты и цвета. *BoxView* можно использовать для оформления, простого графического оформления и взаимодействия с пользователем. Основные свойства:

– *Color* позволяет задать цвет;
– *CornerRadius* позволяет задать радиус границы;
– *WidthRequest* позволяет задать ширину элемента;
– *HeightRequest* позволяет задать высоту элемента.

Пример:

```
...
<BoxView Color="Blue"
       Margin="20"
       CornerRadius="10"
       WidthRequest="100"
       HeightRequest="100"
       VerticalOptions="Center"
       HorizontalOptions="Center">
...
```

Результат представлен на рис. 40.

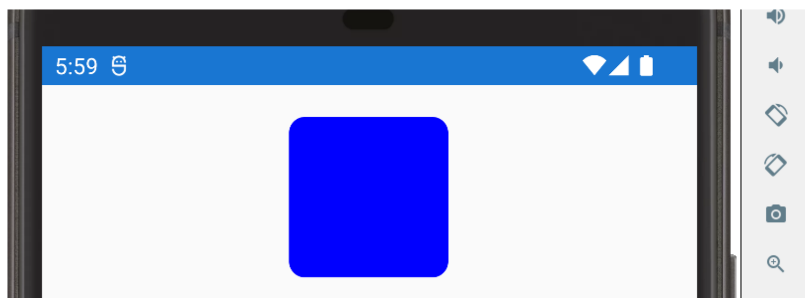

Рис. 40. Квадрат со скругленными углами, элемент BoxView

3.1.4. Элементы выбора значений

К элементам выбора значений можно отнести *CheckBox, Switch*, *RadioButton* и *Picker*.

CheckBox[41] – элемент графического пользовательского интерфейса, позволяющий пользователю управлять параметром с двумя состояниями (включено и отключено). *CheckBox* можно группировать с помощью какого-либо контейнера (например, *Frame*). *CheckBox* содержит свойство *IsChecked*, которое позволяет установить (и получить) значение типа bool (стоит или не стоит галочка). При установке значение вызывается событие *CheckedChanged.*

Пример работы элемента (*CheckBox* будет управлять кнопкой, кнопка активна только тогда, когда стоит галочка).

---

[40] Microsoft Learn.
[41] Там же.

Код XAML:

```
...
<Frame Margin="10">
      <StackLayout Orientation="Horizontal">
            <CheckBox
                  IsChecked="True"
                  CheckedChanged="CheckBox_CheckedChanged"
            >
            </CheckBox>
            <Button
                  x:Name="bb"
                  Text="Кнопка">
            </Button>
      </StackLayout>
</Frame>
...
```

Код C# обработчика события:

```
...
private void CheckBox_CheckedChanged(object sender,
CheckedChangedEventArgs e)
    {
      bb.IsEnabled = ((CheckBox)sender).IsChecked;
    }
...
```

Результат работы программы представлен на рис. 41.

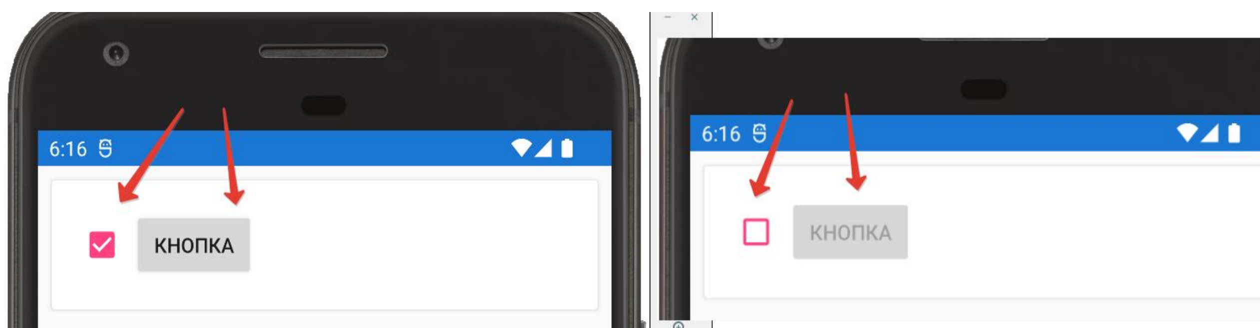

Рис. 41. Пример работы элемента CheckBox

Аналогичным функционалом обладает другой элемент – *Switch*[42], который представляет собой кнопку-переключатель. *Switch* может находиться в двух состояниях: включенном и выключенном. Основные свойства:

– *IsToggled* определяет, находится ли *Switch* во включенном состоянии или в выключенном;

[42] Microsoft Learn.

– *ThumbColor* позволяет задать цвет кнопки переключателя;

– *OnColor* позволяет задать цвет переключателя во включенном состоянии.

Для отслеживания изменения состояния элемента *Switch* используется событие *Toggled.*

Пример работы элемента.

Код XAML:

```
...
<Frame Margin="10">
        <StackLayout Orientation="Horizontal">
          <Switch
            IsToggled="True"
            OnColor="Aqua"
            ThumbColor="Gold"
            Toggled="Switch_Toggled"
            ></Switch>
          <Button
            x:Name="bb"
            Text="Кнопка"></Button>
        </StackLayout>
</Frame>
...
```

Код C#:

```
...
 private void Switch_Toggled(object sender, ToggledEventArgs e)
    {
      bb.IsEnabled = ((Switch)sender).IsToggled;
    }
...
```

Результат работы программы представлен на рис. 42.

Элемент *RadioButton*[43] представляет кнопку-переключатель, которая позволяет выбрать один вариант из группы вариантов. Обратите внимание, что обычно на форме используется не один элемент *RadioButton,* а группа. Основные свойства:

– *Content* определяет содержимое кнопки в виде текста или объекта *View*, отображается справа от *RadioButton*;

– *GroupName* определяет имя группы, к которой принадлежит данная *RadioButton*;

– *Value* определяет некоторое значение (тип object), связанное с *RadioButton*;

– *IsChecked* определяет, находится ли *RadioButton* в отмеченном или неотмеченном состоянии;

[43] Microsoft Learn.

– *GroupName* позволяет сгруппировать кнопки и задает название группы.

Также для группировки можно использовать присоединенные свойства класса *RadioButtonGroup* (задаются атрибутом в элементе-контейнере):

– *GroupName* позволяет сгруппировать кнопки и задает название группы;

– *SelectedValue* позволяет получить выбранный объект *RadioButton.*

Для отслеживания изменения состояния класс *RadioButton* определяет событие *CheckedChanged*.

Пример работы элемента (выбор одного значения из нескольких).

Код XAML:

```
...
 <Frame Margin="10">
        <StackLayout Orientation="Vertical">
          <Label Text="Выберите один из элементов списка"></Label>
          <Label x:Name="Rez"></Label>
          <RadioButton Content="Элемент 1" GroupName="group1"
CheckedChanged="RadioButton_CheckedChanged"></RadioButton>
          <RadioButton Content="Элемент 2" GroupName="group1"
CheckedChanged="RadioButton_CheckedChanged"></RadioButton>
          <RadioButton Content="Элемент 3" GroupName="group1"
CheckedChanged="RadioButton_CheckedChanged"></RadioButton>
          <RadioButton Content="Элемент 4" GroupName="group1"
CheckedChanged="RadioButton_CheckedChanged"></RadioButton>
        </StackLayout>
    </Frame>
...
```

Код C#:

```
...
private void RadioButton_CheckedChanged(object sender,
CheckedChangedEventArgs e)
    {
      var radio = (RadioButton)sender;
      var rez = $"Ваш выбор: {radio.Content}";
      Rez.Text = rez;
    }
...
```

Результат работы программы представлен на рис. 43.

Элемент *Picker*[44] визуально представляет собой обычное текстовое поле, при нажатии на которое открывается выпадающий список для выбора. Основные свойства:

– *ItemsSource* определяет коллекцию отображаемых элементов;

– *SelectedIndex* позволяет задать или получить индекс выделенного элемента;

---

[44] Microsoft Learn.

– *SelectedItem* позволяет задать или получить выбранный элемент.

Для отслеживания изменения (выбора элемента) класс *Picker* определяет событие *SelectedIndexChanged*.

Пример работы с элементом *Picker*.

Код XAML:

```
...
<Frame Margin="10">
        <StackLayout Orientation="Vertical">
          <Label Text="Выберите один из элементов списка"></Label>
          <Label x:Name="Rez"></Label>
          <Picker x:Name="ElList"
            Title="Выберите один из элементов"
              SelectedIndexChanged="Picker_SelectedIndexChanged"
              >
            <Picker.Items>
              <x:String>Элемент 1</x:String>
              <x:String>Элемент 2</x:String>
              <x:String>Элемент 3</x:String>
              <x:String>Элемент 4</x:String>
            </Picker.Items>
          </Picker>
        </StackLayout>
</Frame>
...
```

Код C#:

```
...
 private void Picker_SelectedIndexChanged(object sender, EventArgs e)
    {
      var rez = $"Ваш выбор: {ElList.SelectedItem}";
      Rez.Text = rez;
    }
...
```

Результат работы программы представлен на рис. 44.

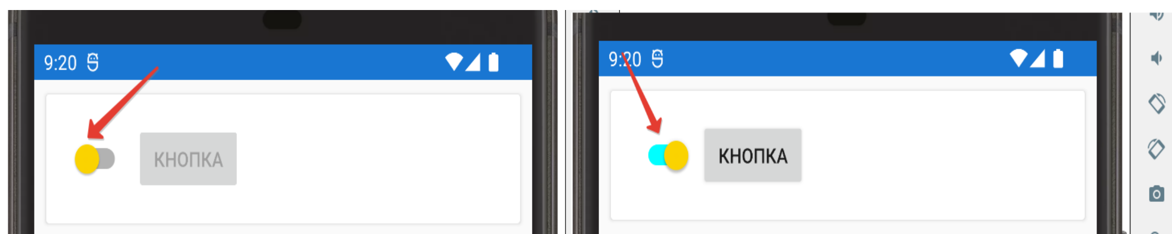

Рис. 42. Пример работы элемента Switch

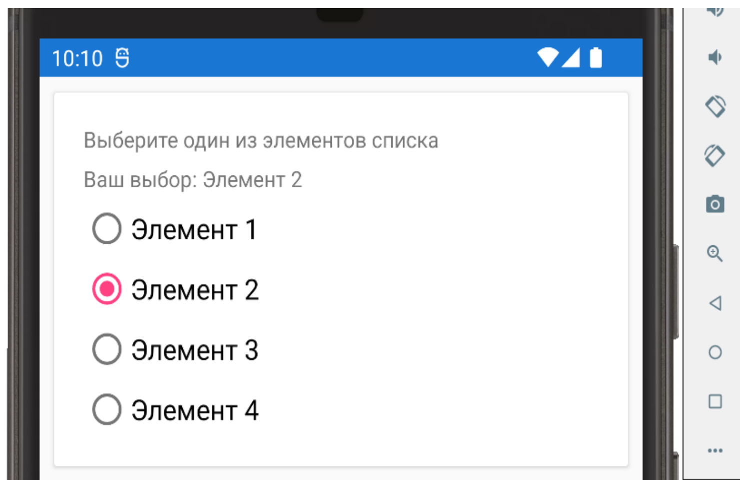

Рис. 43. Выбор одного элемента из четырех с использованием RadioButton

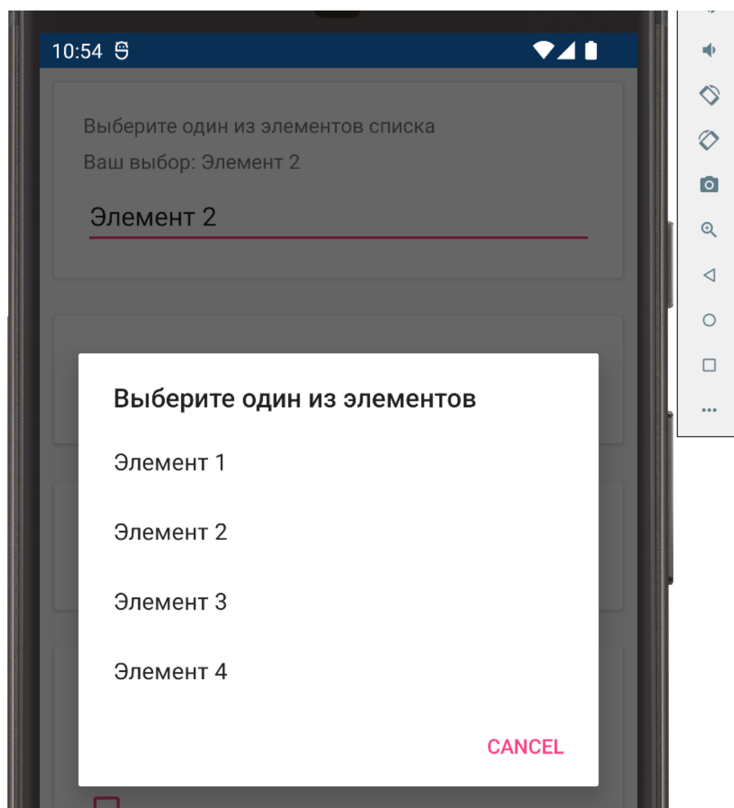

Рис. 44. Выбор значения (элемента списка) с использованием Picker

Список может быть задан в виде коллекции *IList<T>*.
Код XAML:

```
...
 <Frame Margin="10">
        <StackLayout Orientation="Vertical">
          <Label Text="Выберите один из элементов списка"></Label>
          <Label x:Name="Rez"></Label>
          <Picker x:Name="ElList"
            Title="Выберите один из элементов"
              SelectedIndexChanged="Picker_SelectedIndexChanged"
              >
```

```
        </Picker>
      </StackLayout>
    </Frame>
...
```

Код C#:

```
public partial class CheckRadio : ContentPage
  {
private List<string> Lists = new List<string>();
    public CheckRadio()
    {
      InitializeComponent();
      Lists.Add("Элемент 1");
      Lists.Add("Элемент 2");
      Lists.Add("Элемент 3");
      Lists.Add("Элемент 4");
      ElList.ItemsSource = Lists;
    }
    private void Picker_SelectedIndexChanged(object sender, EventArgs e)
    {
      var rez = $"Ваш выбор: {ElList.SelectedItem}";
      Rez.Text = rez;
    }}
```

Результат работы этого кода будет аналогичным (см. рис. 44).

Кроме того, среди свойств *Picker* следует отметить свойство *ItemDisplayBinding*, которое возвращает или задает привязку, выбирающую свойство, отображающееся для каждого объекта в списке элементов.

3.1.5. Элементы для работы с датой и временем

Для работы с датой и временем используются два элемента – соответственно *DatePicker* и *TimePicker*.

С использованием элемента *DatePicker*[45] пользователь приложения может выбирать дату. Основные свойства:

– *MaximumDate* позволяет задать максимальную дату, по умолчанию значение свойства – 31 декабря 2100 г.;

– *MinimumDate* позволяет задать минимальную дату, по умолчанию значение свойства – 1 января 1900 г.;

– *Date* возвращает или задает дату;

– *Format* определяет формат даты, принимает стандартные для .NET форматы.

Для класса *DatePicker* определено событие *DateSelected*, которое срабатывает при выборе новой даты.

[45] Microsoft Learn.

Элемент *TimePicker*[46] представляет собой элемент управления для отображения времени. Основные свойства:

– *Time* возвращает/задает выбранное время;

– *Format* определяет формат даты, принимает стандартные для .NET форматы.

Для класса *TimePicker* определено событие *PropertyChanged*.

Пример формы для выбора даты и времени.

Код XAML:

```
...
<StackLayout>
      <Frame Margin="10">
        <StackLayout Orientation="Vertical">
          <Label Text="Выберите дату и время начала
мероприятия:"></Label>
          <Label x:Name="Rez"></Label>
          <DatePicker
            MinimumDate="01/01/2022"
            MaximumDate="12/31/2022"
            Format="dd MMMM yyyy">
          </DatePicker>
          <TimePicker
            Format="HH:mm">
          </TimePicker>
        </StackLayout>
      </Frame>
    </StackLayout>
...
```

Результат работы программы представлен на рис. 45.

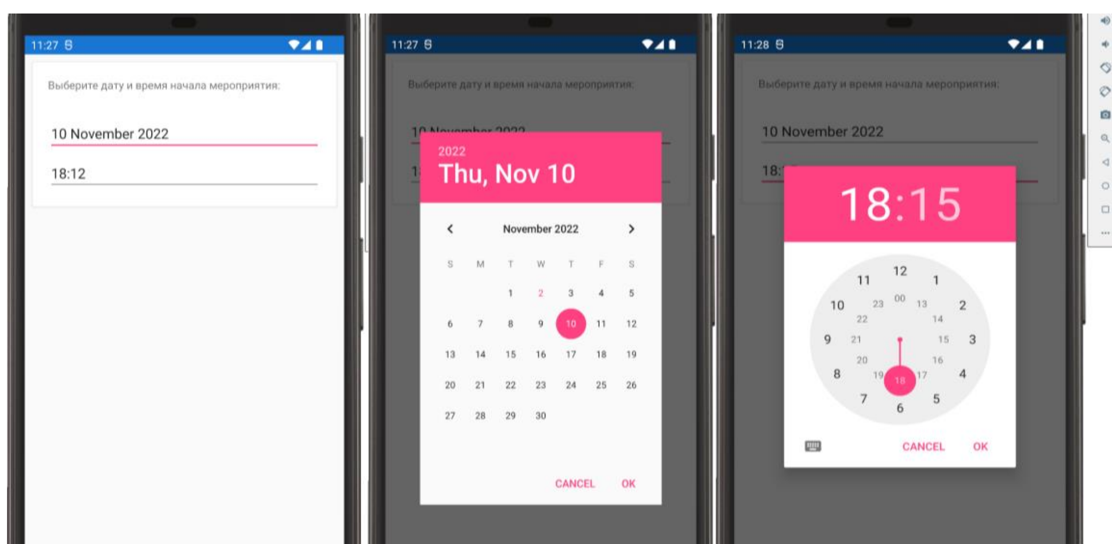

Рис. 45. Пример работы элементов DatePicker и TimePicker

[46] Microsoft Learn.

3.1.6. Мультимедиаэлементы

К элементам для работы с мультимедиа можно отнести элементы *Image* и *WebView*. Элемент *Image*[47] предназначен для отображения изображений на странице. Он имеет несколько важных свойств:

– *Source* задает источник изображения (это может быть файл, URI-ресурса, поток);

– *Aspect* позволяет задать стиль изображения в пределах границ, в которых оно отображается (растянуть или обрезать при несовпадении формата изображения и элемента).

Свойство *Source* задается посредством создания объекта класса *ImageSource*, экземпляр которого можно получить с помощью статических методов для каждого типа источника изображения:

– *FromFile* (можно указать имя файла или путь к файлу, которые разрешаются на каждой платформе);

– *FromUri* (можно указать ссылку на интернет-адрес в виде объекта URI);

– *FromResource* позволяет указать идентификатор ресурса для файла изображения, внедренного в проект приложения или библиотеки .NET Standard, с действием сборки *EmbeddedResource*;

– *FromStream* позволяет указать поток, предоставляющий данные изображения.

Свойство *Aspect* определяет, как будет масштабироваться изображение в соответствии с областью отображения:

– *Fill* задает растягивание изображения, чтобы полностью заполнить область отображения (это может привести к искажению изображения);

– *AspectFill* обрезает изображение таким образом, чтобы оно заполняло область отображения, сохраняя аспект (т.е. без искажений);

– *AspectFit* увеличивает изображение так, чтобы все изображение помещалось в область отображения с пустыми местами, добавленными в верхнюю, нижнюю или боковые стороны в зависимости от соотношения сторон исходного изображения.

При локальном размещении изображения (на устройстве) следует учитывать особенности каждой платформы. Для того чтобы использовать одно изображение во всех приложениях, на каждой платформе нужно применять одно и то же имя файла, оно должно быть допустимым именем ресурса Android (буквы, цифры, символы, подчеркивания). Расположение файлов изображений зависит от платформы.

В iOS, начиная с iOS 9[48], Apple рекомендует использовать *наборы образов в каталоге активов*[49], которые должны содержать все версии образа, необходимые для поддержки различных устройств и коэффициентов масштабирования для приложения.

---

[47] Microsoft Learn.

[48] До iOS 9 изображения обычно помещались в папку «Ресурсы» с действием сборки *BundleResource*.

[49] URL: https://learn.microsoft.com/ru-ru/xamarin/ios/app-fundamentals/images-icons/displaying-an-image?tabs=windows.

В Android изображения следует поместить в каталог resources/drawable с помощью действия сборки *AndroidResource*. Также можно предоставить версии изображений с высоким и низким уровнем DPI (в соответствующих подкаталогах ресурсов, таких как drawable-ldpi, drawable-hdpi и drawable-xhdpi).

В приложениях универсальной платформы Windows (UWP) по умолчанию изображения должны размещаться в корневом каталоге приложения с помощью действия сборки «содержание».

Создадим страницу для вывода фотографии. Вначале разместим файл с фотографией в папке проекта (для примера только для проектов Android и UWP) (рис. 46). В свойствах установим необходимые параметры (действие при сборке и копировать в выходной каталог).

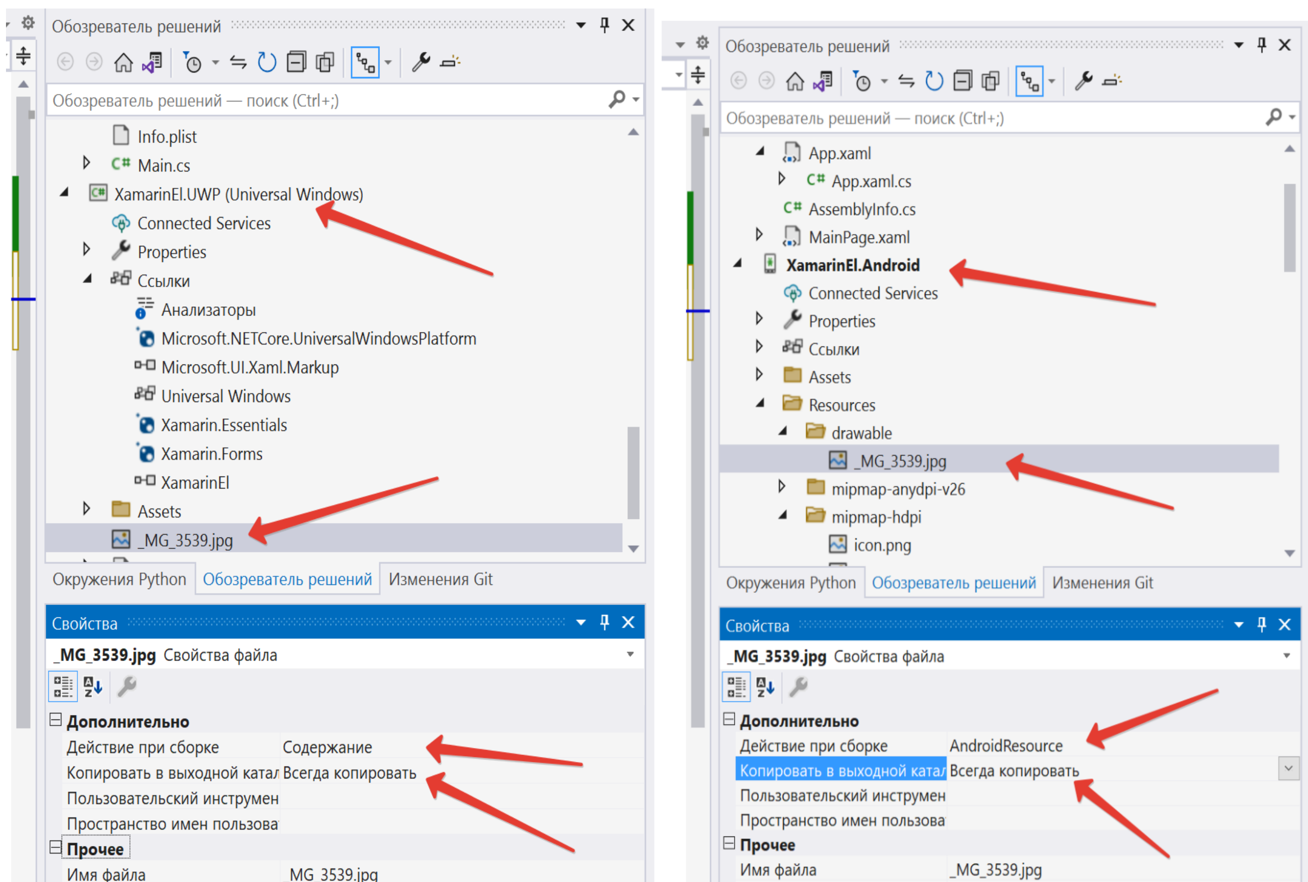

Рис. 46. Расположение файла с изображением в проектах UWP (слева) и Android (справа)

Код XAML:

```
...
<Frame Margin="10">
<StackLayout>
    <Label Text="Фотография" FontSize="Large"></Label>
    <Frame BorderColor="Blue"
   >
<Image Aspect="AspectFit"
```

```
Source="_MG_3539.jpg">
</Image>
    </Frame>
</StackLayout>
</Frame>
...
```

Результат представлен на рис. 47.

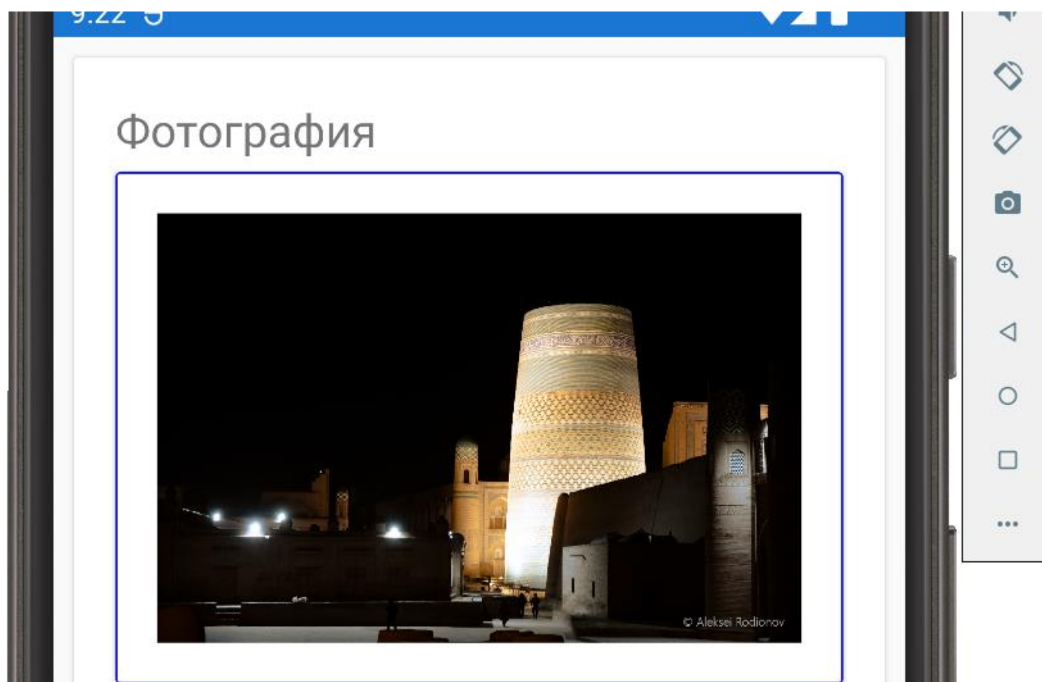

Рис. 47. Отображение фотографии с использованием Image

Элемент *WebView*[50] используется для отображения веб-содержимого в приложении. *WebView* поддерживает следующие типы содержимого:

1. Веб-сайты. *WebView* имеет полную поддержку веб-сайтов, написанных с помощью HTML, CSS, включая поддержку JavaScript.

2. Документы. Так как *WebView* реализуется с помощью собственных компонентов на каждой платформе, *WebView* может отображать документы в форматах, поддерживаемых базовой платформой.

3. Строки на языке HTML. *WebView* может отображать HTML-строки из памяти.

4. Локальные файлы. *WebView* может представлять любой из типов контента, указанных выше, внедренных в приложение.

Обратите внимание, что URL-адреса должны быть записаны с указанием протокола (т.е. они должны иметь «http://» или «https://» в начале).

Элемент *WebView* поддерживает навигацию с помощью нескольких методов и свойств:

[50] Microsoft Learn.

1. *GoForward()*. Если свойство *CanGoForward* имеет значение true, вызов *GoForward* позволяет перейти к следующей посещенной странице.

2. *GoBack()*. Если свойство *CanGoBack* имеет значение true, вызов *GoBack* перейдет на последнюю посещенную страницу.

3. *CanGoBack* (true, если есть страницы, на которые нужно вернуться, false – если браузер находится по исходному URL-адресу);

– *CanGoForward* (true, если пользователь переместился назад и может перейти к уже посещенной странице).

На страницах *WebView* не поддерживаются мультисенсорные жесты. Важно убедиться, что содержимое оптимизировано для мобильных устройств и отображается без необходимости масштабирования.

Класс *WebView* определяет следующие события, помогающие реагировать на изменения в состоянии:

– *Navigating* возникает при начале загрузки новой страницы;

– *Navigated* возникает при загрузке страницы и остановке навигации;

– *ReloadRequested* возникает при запросе на перезагрузку текущего содержимого.

При использовании *WebView* следует убедиться, что для каждой платформы заданы необходимые разрешения. Если это не выполнить, то *WebView* на некоторых платформах может работать в режиме отладки, но не будет работать в релизной версии. Это связано с тем, что некоторые разрешения (например, для доступа к Интернету в Android) устанавливаются по умолчанию в Visual Studio в режиме отладки, но отсутствуют в релизе:

1. UWP. Требуется доступ к Интернету (клиентский и сервер) при отображении сетевого содержимого.

2. Android. Требуется Интернет только при отображении содержимого из сети. Локальное содержимое не требует специальных разрешений.

3. iOS. Не требует специальных разрешений.

Простой просмотр веб-страницы.

Код XAML:

```
...
<Frame Margin="10">
<StackLayout>
    <Label Text="Просмотр web-страницы" FontSize="Large"></Label>
    <WebView WidthRequest="400" HeightRequest="600"
    Source="https://ya.ru">
    </WebView>
</StackLayout>
</Frame>
...
```

Результат работы программы представлен на рис. 48.

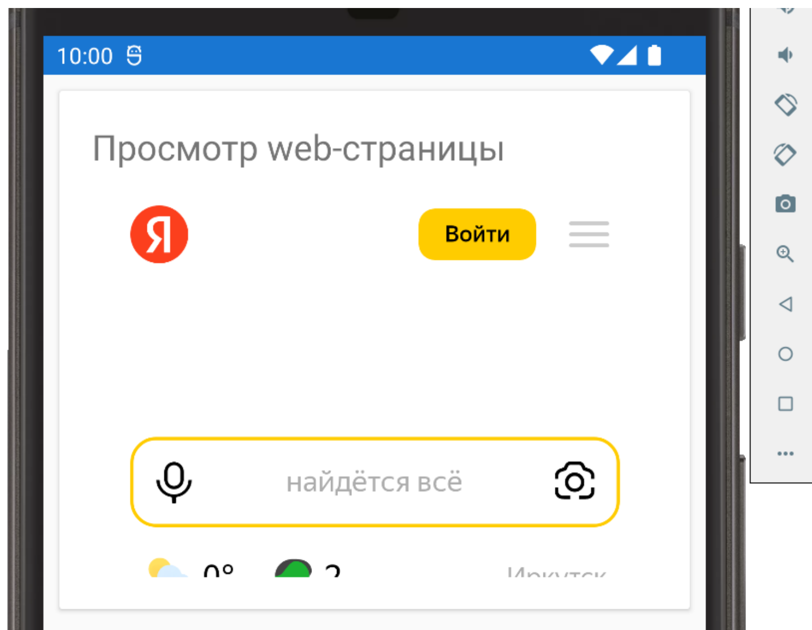

Рис. 48. Просмотр страницы ya.ru в приложении с использованием элемента WebView

Обратите внимание, что в отличие от большинства других элементов, для *WebView* требуется явно указывать значения *HeightRequest* и *WidthRequest*, если он содержится в *StackLayout* или *RelativeLayout*.

Использование *WebView* для отображения форматированной с использованием html текстовой строки.

Код XAML:

```
...
<Frame Margin="10">
                <StackLayout>
                    <Label Text="Просмотр web-страницы"
FontSize="Large"></Label>
                    <WebView WidthRequest="400" HeightRequest="600"
                            x:Name="web">
                    </WebView>
                </StackLayout>
</Frame>
...
```

Код C#:

```
...
public MultiMed()
        {
            InitializeComponent();
            var htmlSource = new HtmlWebViewSource();
            htmlSource.Html = @"<html><body>
                              <h1>Заголовок 1</h1>
```

```
                        <h2>Заголовок 2</h2>
                        <table>
                              <tr>
                              <td>Ячейка 1</td> <td>Ячейка 2</td>
                              </tr>
                              </table>
                        <p>Текст абзаца</p>
                        </body></html>";
            web.Source = htmlSource;
        }
...
```

Результат работы приложения приведен на рис. 49.

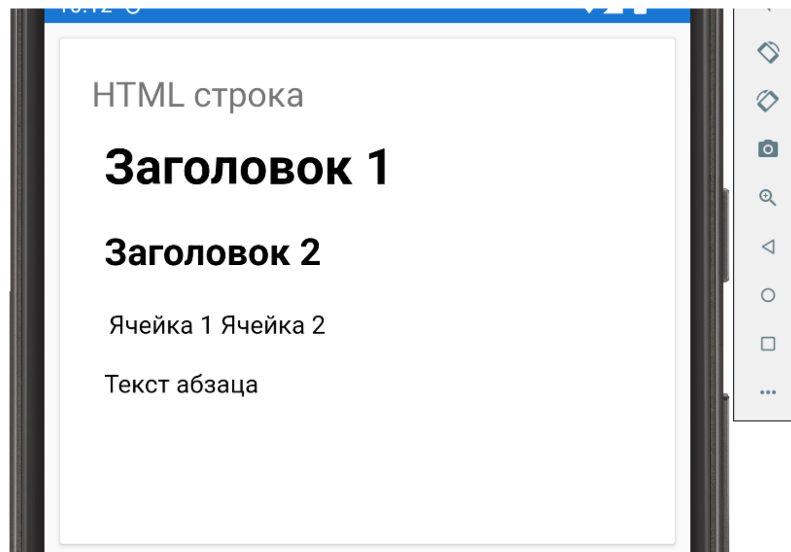

Рис. 49. Просмотр форматированной html-строки в приложении с использованием элемента WebView

3.1.7. Элементы – индикаторы процесса выполнения

К индикаторам процесса выполнения какого-либо действия можно отнести элементы *ActivityIndicator* и *ProgressBar*.

ActivityIndicator используется, когда время выполнения задачи и доля уже выполненной работы неизвестны. Ключевым свойством индикатора является свойство IsRunning, значение которого запускает или останавливает индикатор. В зависимости от платформы принимает вид «бегущего» круга или полосы.

Пример:

```
...
<StackLayout>
            <ActivityIndicator IsRunning="true">
            </ActivityIndicator>
        </StackLayout>
...
```

Результат представлен на рис. 50, слева для Android, справа для Windows.

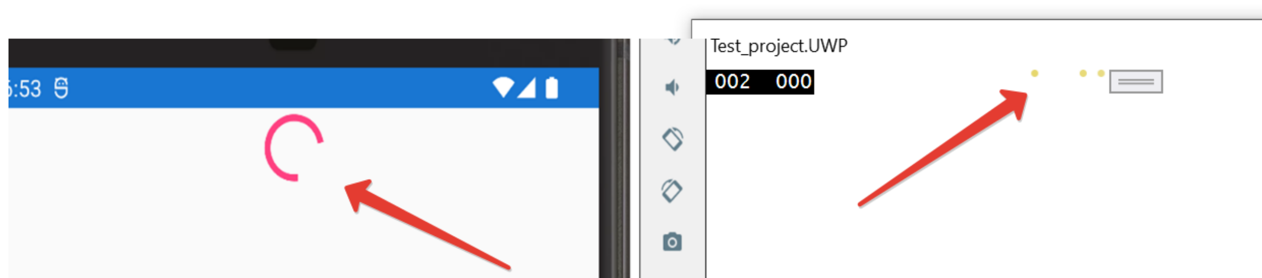

Рис. 50. Пример работы ActivityIndicator

Элемент *ProgressBar* служит для того, чтобы дать пользователю информацию о ходе выполнения какой-либо задачи, при условии, что масштаб задачи и доля уже выполненной работы известны хотя бы приблизительно. Представляет собой горизонтальную полосу, заполненную в процентах. Основные свойства:

– *Progress* – значение, представляющее текущий ход выполнения в виде значения от 0 до 1;

– *ProgressColor* задает цвет полосы.

Также элемент имеет метод *ProgressTo,* который запускает анимацию движения индикатора в течение определенного времени до нужного значения.

Пример запуска индикатора, который должен полностью заполниться за 2 сек по нажатию кнопки.

Код XAML:

```
...
<StackLayout>
            <ProgressBar x:Name="pb" ProgressColor="Green">
            </ProgressBar>
            <Button Text="Старт" Clicked="Button_Clicked"></Button>
        </StackLayout>
</ContentPage.Content>
...
```

Код C#:

```
...
private async void Button_Clicked(object sender, EventArgs e)
{
      await pb.ProgressTo(1, 2000, Easing.Linear);
}
...
```

Результат представлен на рис. 51.

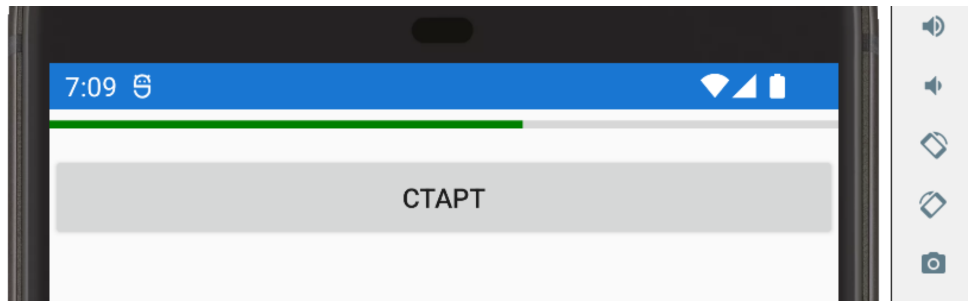

Рис. 51. Демонстрация работы ProgressBar

3.1.8. Прокрутка элементов

В случае, когда количество элементов на форме много, может сложиться ситуация, что не все элементы смогут разместиться на ней. В этом случае целесообразно создать возможность «прокрутки» формы в вертикальном или горизонтальном направлениях. Для этого можно использовать элемент *ScrollView*[51]. Основные свойства элемента:

– *Content* определяет содержимое области прокрутки;

– *HorizontalScrollBarVisibility* – значение типа *ScrollBarVisibility*, которое «показывает» полосу горизонтальной прокрутки;

– *VerticalScrollBarVisibility* – значение типа *ScrollBarVisibility*, которое «показывает» полосу вертикальной прокрутки;

– *Orientation* устанавливает направление прокрутки (*ScrollOrientation*), по умолчанию имеет значение *Vertical* (вертикальная прокрутка).

Создадим форму с множеством строк.

Код XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Views.ScrolPage">
    <ContentPage.Content>
        <ScrollView VerticalScrollBarVisibility="Always">
            <StackLayout>
                <Label Text="Строка 1" FontSize="Large"></Label>
                <Label Text="Строка 2" FontSize="Large"></Label>
                <Label Text="Строка 3" FontSize="Large"></Label>
                <Label Text="Строка 4" FontSize="Large"></Label>
                <Label Text="Строка 5" FontSize="Large"></Label>
                <Label Text="Строка 6" FontSize="Large"></Label>
                <Label Text="Строка 7" FontSize="Large"></Label>
                <Label Text="Строка 8" FontSize="Large"></Label>
                <Label Text="Строка 9" FontSize="Large"></Label>
                <Label Text="Строка 10" FontSize="Large"></Label>
                <Label Text="Строка 11" FontSize="Large"></Label>
                <Label Text="Строка 12" FontSize="Large"></Label>
                <Label Text="Строка 13" FontSize="Large"></Label>
                <Label Text="Строка 14" FontSize="Large"></Label>
```

[51] Microsoft Learn.

```
            <Label Text="Строка 15" FontSize="Large"></Label>
            <Label Text="Строка 16" FontSize="Large"></Label>
            <Label Text="Строка 17" FontSize="Large"></Label>
            <Label Text="Строка 18" FontSize="Large"></Label>
            <Label Text="Строка 19" FontSize="Large"></Label>
            <Label Text="Строка 20" FontSize="Large"></Label>
            <Label Text="Строка 21" FontSize="Large"></Label>
            <Label Text="Строка 22" FontSize="Large"></Label>
         </StackLayout>
      </ScrollView>
   </ContentPage.Content>
</ContentPage>
```

Запустив программу, мы увидим, что на экране размещается только 19 строк. Использование контейнера *ScrollView* позволяет решить эту проблем за счет появления справа от формы полосы прокрутки. С ее помощью можно «прокрутить» форму до нужной строки (рис. 52).

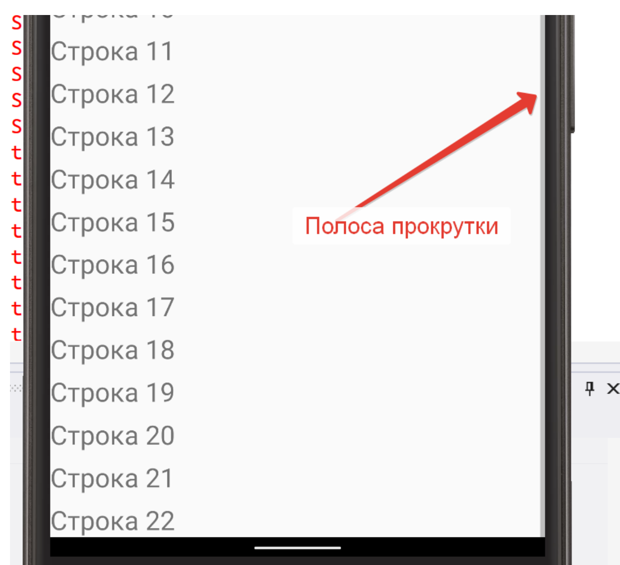

Рис. 52. Полоса прокрутки

3.1.9. Элементы для отображения коллекций

В данном разделе рассмотрим два типовых элемента – *ListView* и *TableView*. *ListView* предназначен для представления списков данных, особенно длинных списков, требующих прокрутки. Элемент *ListView* имеет ряд компонентов, доступных для реализации собственных функциональных возможностей каждой платформы.

1. Компоненты верхнего и нижнего колонтитулов отображаются в начале и конце списка, отдельно от данных списка.

2. Группы. Данные в элементе управления *ListView* можно сгруппировать для упрощения навигации.

3. Ячейки. Каждая ячейка соответствует строке данных. Встроенные ячейки можно задавать как «универсальным» *ViewCell,* так и «типизированным», например:

– *SwitchCell* – это элемент управления, представляющий собой переключатель;

– *EntryCell* используется, когда нужно отобразить текстовые данные, которые пользователь может изменить;

– *TextCell* – элемент управления, который используется для отображения текста с необязательной второй строкой для подробного текста;

– *ImageCell* – элемент управления, похожий на *TextCells*, но включает изображение слева от текста.

Стили взаимодействия, например:

1. Обновление. По запросу *ListView* может обновить содержимое.

2. Контекстные действия. Позволяют разработчику указывать пользовательские действия для отдельных элементов списка. Например, можно обработать действие с длинным касанием в Android и т.п.

3. Выбор. Позволяет разработчику присоединять функциональные возможности к событиям выбора и отмены выбора элементов списка.

ListView связывается с набором данных через свойство *ItemsSource*. Пример использования элемента *ListView* (более детальный пример будет рассмотрен во второй части пособия, после изучения привязки):

```
...
<ContentPage.Content>
        <ListView x:Name="lw">
        </ListView>
</ContentPage.Content>
...
```

Код C#:

```
...
    public partial class ListPage : ContentPage
    {
        public List<Student> Students { get; set; }
        public ListPage()
        {
            InitializeComponent();
            Students = new List<Student>
            {
                new Student{Fio="Иванов Иван Иванович", Group="Ис-10-1"},
                new Student{Fio="Петров Петр Петрович", Group="Ис-10-1"},
                new Student{Fio="Сидоров Сидор Сидорович", Group="Ис-10-
2"}
            };
            lw.ItemsSource = Students;
        }
```

```
    }
    public class Student
    {
        public string Fio { get; set; }
        public string Group { get; set; }
        public override string ToString()
        {
            return $"{Fio} {Group}";
        }
    }
…
```

Элемент *TableView* позволяет отображать данные в табличной форме, но, в отличие от *ListView*, не использует единый для всех данных шаблон (*DataTemplate*) и не поддерживает задание источника данных (*ItemSource*)

TableView можно использовать для создания списка параметров или формы для сбора информации.

Дочерние элементы в *TableView* упорядочиваются по разделам. Корневым элементом *TableView* является *TableRoot* – родительский объект для одного или нескольких *TableSection*. Каждый из *TableSection* состоит из заголовка и одного или нескольких *ViewCell* (аналогично *ListView*).

Внешний вид элемента задается свойством *TableIntent*. Возможные значения перечисления:

– *Data* используется при отображении записей данных, представляет собой таблицу, которая может содержать произвольное число подобных записей данных;

– *Form* используется в том случае, если таблица предназначена для отображения сведений, которые обычно содержатся в формах;

– *Menu* предназначена для использования в качестве меню для выделенных фрагментов;

– *Settings* – таблица, содержащая набор переключателей или других изменяемых параметров конфигурации.

Ячейки таблицы генерируют ряд событий: *EntryCell* при завершении ввода генерирует событие *Completed*, *SwitchCell* при переключении состояния – событие *OnChanged,* *ViewCell* при касании/нажатии – событие *OnViewCellTapped.*

Пример использования *TableView* для создания формы настройки.

Код XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Views.TablePage">
    <ContentPage.Content>
        <TableView Intent="Settings">
            <TableRoot>
                <TableSection >
                    <TableSection.Title>
```

```
                        <x:String>Настройки</x:String>
                    </TableSection.Title>
                    <SwitchCell Text="Настройка 1" />
                    <SwitchCell Text="Настройка 2" On="true" />
                    <EntryCell Label="Строка подключения:"
Text=""></EntryCell>
                </TableSection>
            </TableRoot>
        </TableView>
    </ContentPage.Content>
</ContentPage>
```

Результат представлен на рис. 53.

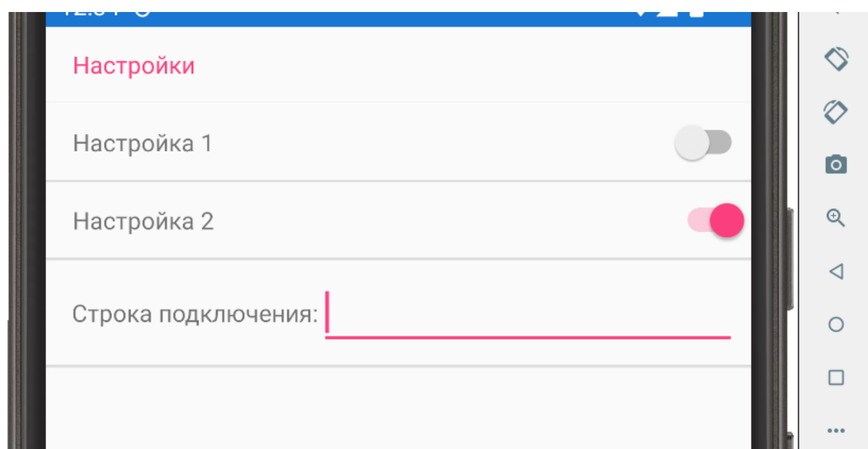

Рис. 53. Использование TableView для создания страницы с настройками приложения

Пример использования *TableView* для создания формы:

```
...
<TableView Intent="Form">
<TableRoot>
    <TableSection >
<TableSection.Title>
    <x:String>Информация об участнике</x:String>
</TableSection.Title>
<EntryCell Label="Фамилия:"  Placeholder="Введите фамилию"></EntryCell>
<EntryCell Label="Имя:"  Placeholder="Введите имя"></EntryCell>
<EntryCell Label="Отчество:"  Placeholder="Введите отчество"></EntryCell>
    </TableSection>
    <TableSection >
<TableSection.Title>
    <x:String>Регистрация в системе</x:String>
</TableSection.Title>
<EntryCell Label="Логин:"  Text=""></EntryCell>
<EntryCell Label="Пароль:"  Text=""></EntryCell>
<SwitchCell Text="Сохранить пароль" />
    </TableSection>
```

```
</TableRoot>
...
```

Результат выполнения кода представлен на рис. 54.

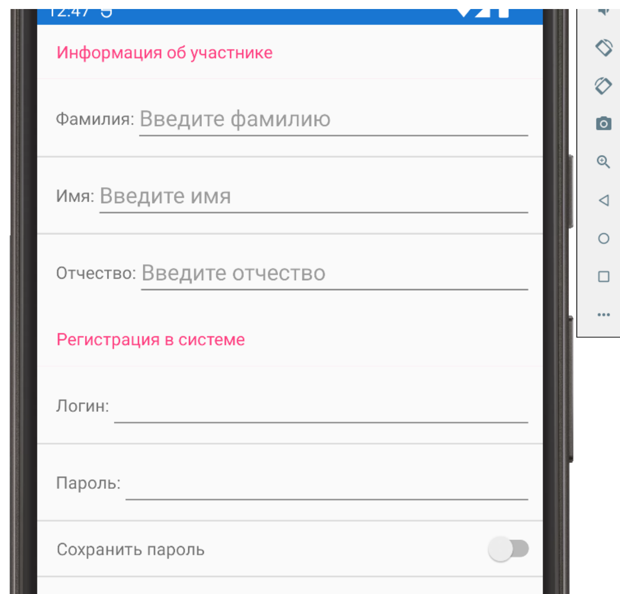

Рис. 54. Форма регистрации с использованием TableView

Практический пример 4.
Форма регистрации на мероприятие

Задача. Разработать форму регистрации на мероприятие. Необходимо собрать информацию об участнике, адресе проживания, форме участия.

Решение.

Реализуем класс для участника мероприятия *PartEvent*. Дополнительно создадим перечисления для указания пола участника и типа участия. Класс *Address* будем использовать для работы с адресом проживания.

Код C#:

```
...
public enum Pol
{
	Мужской = 0, Женский = 1,
}
public enum TypePart
{
	Личный_доклад = 0, Онлайн_доклад=1, Публикация =2
}
public class Address
	{
		public string Post { get; set; }
		public string City { get; set; }
		public string Adress1 { get; set; }
	}
```

```
public class PartEvent
     {
          public int Id { get; set; }
          public string FIO { get; set; }
          public DateTime DateofBirth { get; set; }
          public Pol Pol { get; set; }
          public TypePart Type { get; set; }
          public string Email { get; set; }
          public Address Address { get; set; }
          public PartEvent()
          {
               Address = new Address();
          }

          public override string ToString()
          {
               var rez = string.Format(
                              @"ФИО:{0}
                              Возраст:{1}
                              Пол:{2}
                              Тип участия:{3}
                              Адрес для связи:{4}
                              Почтовый индекс:{5}
                              Город:{6}
                              Дом и улица:{7}
                           ", FIO, DateofBirth.ToString("dd MMMM
yyyy"),
                           Pol, Type.ToString(), Email,
                           Address.Post, Address.City,
Address.Adress1);

               return rez;
          }
...
```

Код XAML на форме:

```
...
<StackLayout>
    <TableView Intent="Form" HasUnevenRows="True">
<TableRoot>
    <TableSection Title="Информация об участнике" >
<EntryCell Label="ФИО:"  Placeholder="Введите ФИО участника"
Completed="EntryCell_Completed"></EntryCell>
<ViewCell>
    <StackLayout Padding="15,10" Orientation="Horizontal" >
<Label Text="Дата рождения:"
   VerticalOptions="Center"
   ></Label>
<DatePicker DateSelected="DatePicker_DateSelected"
    VerticalOptions="Center" HorizontalOptions="EndAndExpand"
```

```
Format="dd MMMM yyyy"
WidthRequest="200"
MinimumDate="01/01/1960"></DatePicker>
    </StackLayout>
</ViewCell>
<ViewCell>
    <StackLayout Padding="15,10" Orientation="Horizontal">
<Label Text="Пол:"
   VerticalOptions="Center"
   ></Label>
<Label Text="Мужской" VerticalOptions="Center"
HorizontalOptions="EndAndExpand"></Label>
<Switch Toggled="Switch_Toggled"
    OnColor="Gray" ThumbColor="Blue" VerticalOptions="Center"
HorizontalOptions="End"></Switch>
<Label Text="Женский" VerticalOptions="Center"
HorizontalOptions="End"></Label>
    </StackLayout>
</ViewCell>
    </TableSection>
    <TableSection Title="Адрес участника">
<EntryCell Label="Почтовый индекс:"
   Completed="EntryCell_Completed_1"
       Placeholder="Введите почтовый индекс"
       Keyboard="Numeric"></EntryCell>
<EntryCell Label="Город:"
   Completed="EntryCell_Completed_2"
       Placeholder="Введите название города"
       Keyboard="Text"></EntryCell>
<EntryCell Label="Адрес:"
   Completed="EntryCell_Completed_3"
       Placeholder="Улица, дом, квартира"
       Keyboard="Default"></EntryCell>
    </TableSection>
    <TableSection Title="Тип участия">
<ViewCell>
    <StackLayout Padding="15,10" Orientation="Horizontal">
<Label Text="Выберите тип участия:"
   VerticalOptions="Center"
   ></Label>
<Picker x:Name="PType"
SelectedIndexChanged="PType_SelectedIndexChanged"
WidthRequest="200"
   VerticalOptions="Center"
   HorizontalOptions="EndAndExpand"></Picker>
    </StackLayout>
</ViewCell>
<EntryCell Label="E-mail"
   Completed="EntryCell_Completed_4"
       Placeholder="E-mail для связи"
       Keyboard="Email"></EntryCell>
    </TableSection>
```

```
</TableRoot>
    </TableView>
    <Button Margin="15,10" Text="Регистрация" Clicked
="Button_Clicked"></Button>
</StackLayout>
```

Когда пользователь начнет заполнять поля формы, будут срабатывать обработчики событий. Информация станет «фиксироваться» в полях экземпляра класса *PartEvent*. Так как это только демонстрационная форма, то код регистрации мы писать пока не будем, а на кнопку «регистрация» повесим функцию демонстрации всплывающего окна с полной информацией о сделанной записи.

Код C# (конструктор класса страницы *TablePage* и обработчики событий):

```
XamlCompilation(XamlCompilationOptions.Compile)]
public partial class TablePage : ContentPage
{
public PartEvent PartEvent { get; set; }
public TablePage()
{
    InitializeComponent();
    PartEvent = new PartEvent();

    foreach (TypePart item in Enum.GetValues(typeof(TypePart)))
    {
PType.Items.Add(item.ToString());
    }
}
private void EntryCell_Completed(object sender, EventArgs e)
{
    PartEvent.FIO=((EntryCell)sender).Text;
}
private async void Button_Clicked(object sender, EventArgs e)
{
    await App.Current.MainPage.DisplayAlert("Информация об участнике",
PartEvent.ToString(), "OK");
}
private void DatePicker_DateSelected(object sender, DateChangedEventArgs
e)
{
    PartEvent.DateofBirth = e.NewDate;
}
private void Switch_Toggled(object sender, ToggledEventArgs e)
{
    if (e.Value)
PartEvent.Pol = Pol.Женский;
    else
PartEvent.Pol = Pol.Мужской;
}
private void EntryCell_Completed_1(object sender, EventArgs e)
```

```
{
    PartEvent.Address.Post= ((EntryCell)sender).Text;
}

private void EntryCell_Completed_2(object sender, EventArgs e)
{
    PartEvent.Address.City = ((EntryCell)sender).Text;
}
private void EntryCell_Completed_3(object sender, EventArgs e)
{
    PartEvent.Address.Adress1 = ((EntryCell)sender).Text;
}
private void EntryCell_Completed_4(object sender, EventArgs e)
{
    PartEvent.Email = ((EntryCell)sender).Text;
}
private void PType_SelectedIndexChanged(object sender, EventArgs e)
{
    var s = ((Picker)sender).SelectedIndex;
    PartEvent.Type = (TypePart)s;
}
}
}
```

На рис. 55 приведена демонстрация работы приложения.

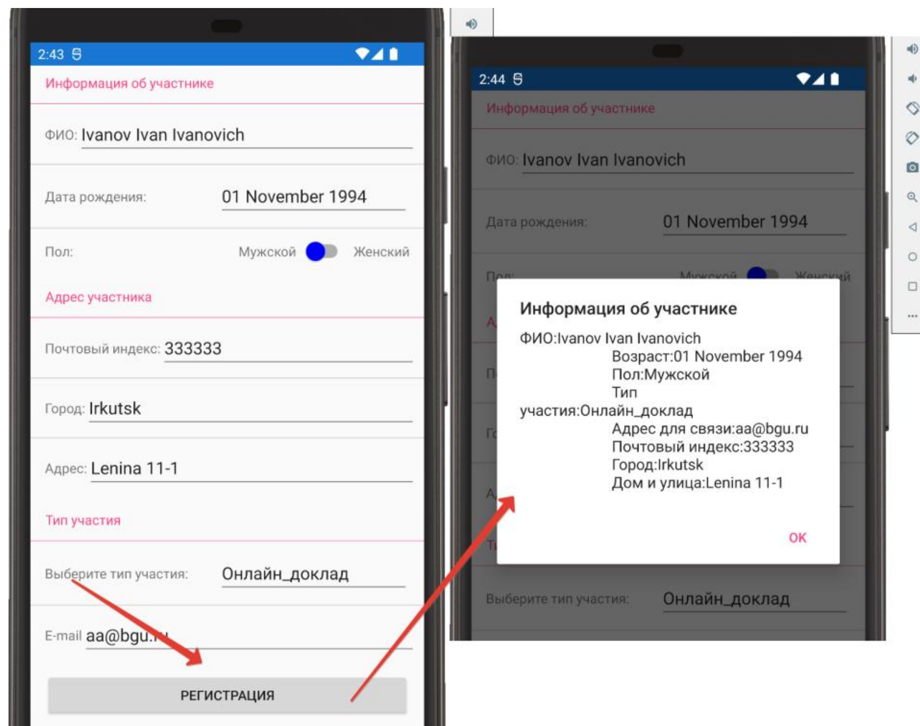

Рис. 55. Форма регистрации на мероприятие

Заполнив форму и нажав кнопку «регистрация», мы увидим информационное окно о сделанной записи.

3.2. Макеты страниц и элементы компоновки

3.2.1. Введение в компоновку

Элементы компоновки позволяют упорядочивать и группировать элементы управления пользовательского интерфейса в приложении. Выбор элемента компоновки зависит от того, какие элементы должны быть на форме и как интерфейс должен отвечать на воздействия (увеличение/уменьшение размеров формы и т.д.). Обратите внимание, что часто для создания нужной оконной формы необходимо комбинировать элементы компоновки (вложение элементов друг в друга). Основные элементы компоновки:

– *StackLayout* упорядочивает элементы в одномерном стеке по горизонтали или по вертикали;

– *AbsoluteLayout* служит для статичного позиционирования элементов (позиция задается координатами верхнего левого угла дочернего элемента относительно левого верхнего угла родительского элемента);

– *Grid* используется для отображения элементов в таблице, строки и столбцы таблицы могут иметь пропорциональные или абсолютные размеры;

– *RelativeLayout* используется для размещения элементов относительно других элементов (по умолчанию элемент размещается в левом верхнем углу макета, все элементы размещаются относительно предыдущих элементов);

– *FlexLayout* используется для того, чтобы дочерние элементы отображались по горизонтали или по вертикали в стеке, при этом есть возможность сделать вложенные элементы «гибкими», они могут сжиматься и растягиваться по заданным правилам, занимая нужное пространство.

На рис. 56 приведены схемы основных элементов компоновки, которые можно использовать при проектировании и разработке пользовательских интерфейсов.

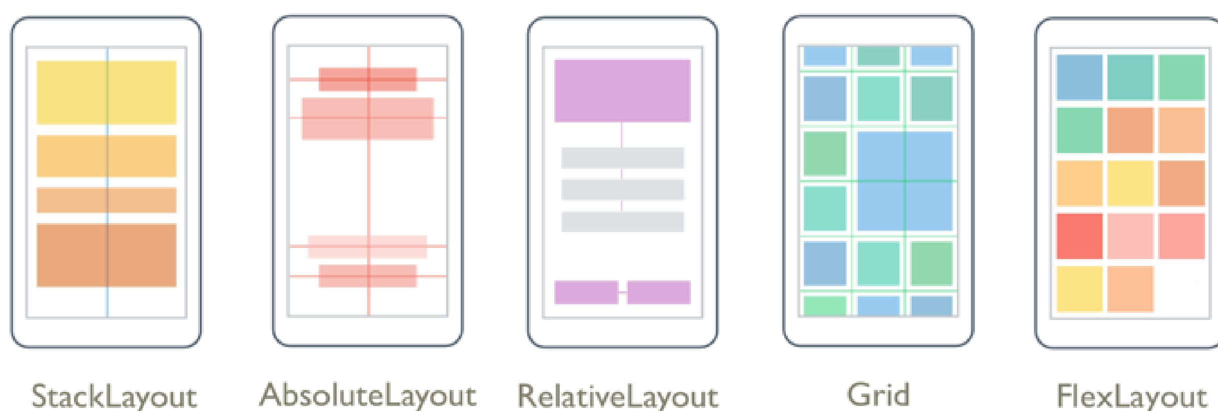

Рис. 56. Основные элементы компоновки

Все элементы компоновки имеют свойство *Children.* Это свойство есть у всех классов-контейнеров, с его помощью можно как получить вложенные элементы, так и добавить новые. Коллекция *Children* представляет собой обычный список, поэтому к ней допустимы все операции, определенные для списков, в

том числе и *Linq*. В качестве примера получим значения всех элементов *Entry*, расположенных в контейнере *StackLayout* и просуммируем их.

Код XAML:

```
...
<StackLayout x:Name="Form">
    <Label Text="Введите X"></Label>
    <Entry x:Name="x"></Entry>
    <Label Text="Введите Y"></Label>
    <Entry x:Name="y"></Entry>
    <Button Text="Расчет" Clicked="Button_Clicked"></Button>
    <StackLayout Orientation="Horizontal" >
<Label Text="Сумма  X + Y = "></Label>
<Label x:Name="rez"></Label>
    </StackLayout>
</StackLayout>
...
```

Код C#:

```
...
private void Button_Clicked(object sender, EventArgs e)
{
    var listentry = Form.Children.OfType<Entry>();
    var s = listentry.Select(t => int.Parse(t.Text)).
    Aggregate((a, b) => a + b);
}
...
```

Скриншот программы представлен на рис. 57.

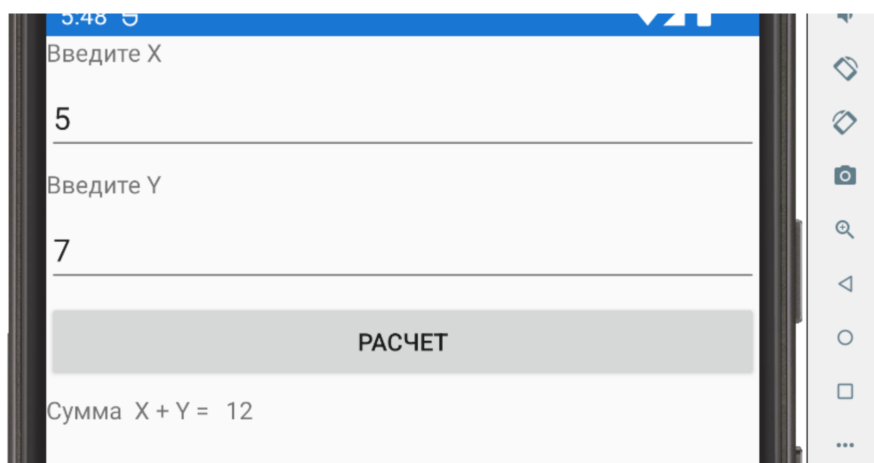

Рис. 57. Демонстрация работы с коллекцией Children

В следующих параграфах в качестве дочерних элементов будут выступать элементы *BoxView* для наглядности. Вместо них могут быть абсолютно любые элементы, и в том числе сами элементы компоновки.

3.2.2. Элемент Grid

Элемент *Grid* позволяет создать макет экранной формы, в которой все дочерние элементы размещаются в строках и столбцах таблицы. По умолчанию *Grid* содержит одну строку и один столбец. *Grid* часто используют в качестве базового (родительского) элемента компоновки, в ячейках которой содержатся другие элементы компоновки. Элемент *Grid* не следует путать с таблицами, он не предназначен для представления табличных данных. Класс *Grid* определяет следующие свойства:

– *Column* – присоединенное (статическое) свойство, которое задает позицию (номер колонки) для дочерних элементов (значение по умолчанию – 0);

– *Row* – присоединенное (статическое) свойство, которое задает позицию (номер строки) для дочерних элементов (значение по умолчанию – 0);

– *ColumnDefinitions* получает или задает упорядоченную коллекцию объектов *ColumnDefinition*, определяющих макет столбцов в *Grid*;

– *RowDefinitionsc* получает или задает упорядоченную коллекцию объектов *RowDefinition*, определяющих макет строк в *Grid*;

– *ColumnSpacing* указывает расстояние между столбцами сетки (значение по умолчанию – 6);

– *ColumnSpan* – присоединенное свойство, позволяющее объединить несколько столбцов в родительском объекте *Grid*;

– *RowSpacing* указывает расстояние между строками сетки (значение по умолчанию – 6);

– R*owSpan* – присоединенное свойство, позволяющее объединить несколько строк в родительском объекте *Grid*.

Для дочерних элементов при размещении в ячейках можно задать выравнивание с помощью свойств *VerticalOptions* и *HorizontalOptions*: *Start* (с начала), *Center* (в центре), *End* (с конца).

В качестве примера создадим таблицу из пяти строк и двух столбцов. Высота первой и последней строки – 100, вторая, третья и четвертая строки делят остальное пространство в соотношении 2:1:2. Ширина первого столбца – 150, второй столбец занимает все остальное пространство. Для наглядности в качестве дочерних элементов будем использовать *BoxView*, причем один элемент будет занимать три строки, а другой – два столбца.

Код XAML:

```
…
<Grid>
    <Grid.RowDefinitions>
<RowDefinition Height="100"></RowDefinition>
<RowDefinition Height="2*"></RowDefinition>
<RowDefinition Height="1*"></RowDefinition>
<RowDefinition Height="2*"></RowDefinition>
<RowDefinition Height="100"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
```

```
<ColumnDefinition Width="150"></ColumnDefinition>
<ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <BoxView Color="Aqua" Grid.RowSpan="3" Grid.Column="0"></BoxView>
    <BoxView Color="DarkRed" Grid.Row="0" Grid.Column="1"></BoxView>
    <BoxView Color="LightPink" Grid.Row="1" Grid.Column="1"></BoxView>
    <BoxView Color="Red" Grid.Row="2" Grid.Column="1"></BoxView>
    <BoxView Color="Blue" Grid.Row="3" Grid.ColumnSpan="2"></BoxView>
    <BoxView Color="Green" Grid.Row="4" Grid.Column="0"></BoxView>
    <BoxView Color="Orange" Grid.Row="4" Grid.Column="1"></BoxView>
</Grid>
...
```

Результат представлен на рис. 58.

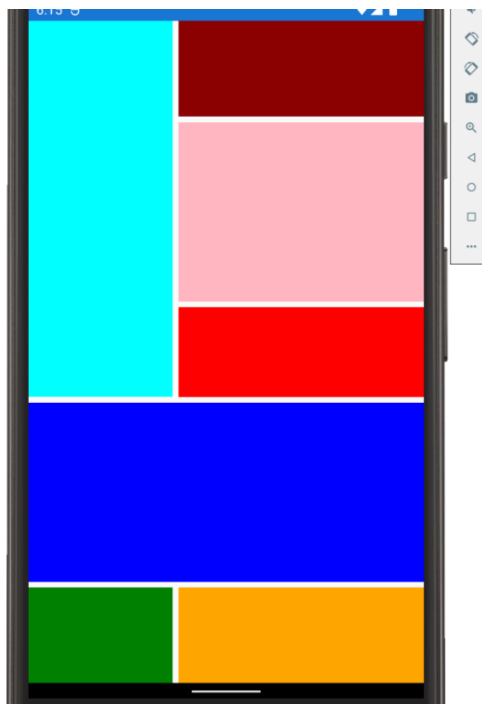

Рис. 58. Демонстрация работы элемента Grid

Кроме того, дочерние представления могут совместно размещаться в ячейках *Grid*. Порядок отображения дочерних элементов в XAML определяется очередностью записи в макете:

```
...
<Grid>
    <Grid.RowDefinitions>
<RowDefinition Height="100"></RowDefinition>
<RowDefinition Height="1*"></RowDefinition>
<RowDefinition Height="3*"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
<ColumnDefinition Width="150"></ColumnDefinition>
```

```
<ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <BoxView Color="Orange" Grid.Row="0" Grid.Column="0"></BoxView>
    <Label Text="Ячейка 0-0" FontSize="Large" Grid.Row="0"
Grid.Column="0"
   VerticalTextAlignment="Center" HorizontalTextAlignment="Center"
   ></Label>
    <BoxView Color="DarkRed" Grid.Row="0" Grid.Column="1"></BoxView>
    <Label Text="Ячейка 0-1" FontSize="Large" Grid.Row="0"
Grid.Column="1"
   VerticalTextAlignment="Center" HorizontalTextAlignment="Center"
   TextColor="White"
   ></Label>
    <BoxView Color="LightPink" Grid.Row="1"
Grid.ColumnSpan="2"></BoxView>
    <Label Text="Объединение по столбцам" FontSize="Large" Grid.Row="1"
Grid.ColumnSpan="2"
   VerticalTextAlignment="Center" HorizontalTextAlignment="Center"
   ></Label>
    <BoxView Color="Red" Grid.Row="2" Grid.Column="0"></BoxView>
    <Label Text="Ячейка 2-0" FontSize="Large" Grid.Row="2"
Grid.Column="0"
   VerticalTextAlignment="Start" HorizontalTextAlignment="Center"
   ></Label>
    <BoxView Color="Blue" Grid.Row="2" Grid.Column="1"></BoxView>
    <Label Text="Ячейка 2-1" FontSize="Large" Grid.Row="2"
Grid.Column="1"
   VerticalTextAlignment="Start" HorizontalTextAlignment="Center"
   TextColor="White"
   ></Label>
</Grid>
...
```

Результат работы представлен на рис. 59.

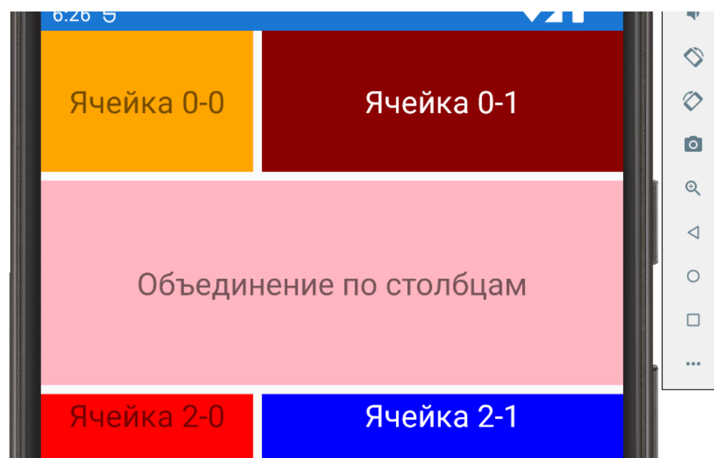

Рис. 59. «Наложение» дочерних элементов

3.2.3. Элемент StackLayout

Элемент *StackLayout* упорядочивает дочерние элементы по горизонтали или по вертикали. По умолчанию *StackLayout* имеет вертикальную ориентацию. *StackLayout* можно использовать в качестве родительского макета, содержащего другие дочерние элементы компоновки. Класс *StackLayout* определяет следующие свойства:

– *Orientation* задает направление, в котором размещаются дочерние представления (значение по умолчанию этого свойства равно Vertical);

– Spacing задает расстояние между дочерними элементами (значение по умолчанию этого свойства – 6).

Размер и положение дочерних элементов в пределах *StackLayout* зависит от значений их свойств: *HeightRequest*, *WidthRequest*, *HorizontalOptions* и *VerticalOptions*. При вертикальной ориентации *StackLayout* дочерние представления расширяются, чтобы заполнить всю доступную ширину, если их размер не задан явным образом. При горизонтальной ориентации *StackLayout* дочерние элементы расширяются, чтобы заполнить доступную высоту, если их размер не задан явным образом.

Код XAML:

```
…
<StackLayout>
    <BoxView Color="Aqua"></BoxView>
    <BoxView Color="DarkRed"></BoxView>
    <BoxView Color="LightPink" ></BoxView>
    <BoxView Color="Red" ></BoxView>
    <BoxView Color="Blue" ></BoxView>
    <BoxView Color="Green" ></BoxView>
    <BoxView Color="Orange" ></BoxView>
</StackLayout>
…
```

Пример для горизонтальной ориентации:

```
…
<StackLayout Orientation="Horizontal">
    <BoxView Color="Aqua"></BoxView>
    <BoxView Color="DarkRed"></BoxView>
    <BoxView Color="LightPink" ></BoxView>
    <BoxView Color="Red" ></BoxView>
    <BoxView Color="Blue" ></BoxView>
    <BoxView Color="Green" ></BoxView>
    <BoxView Color="Orange" ></BoxView>
</StackLayout>
…
```

Демонстрация работы программы приведена на рис. 60.

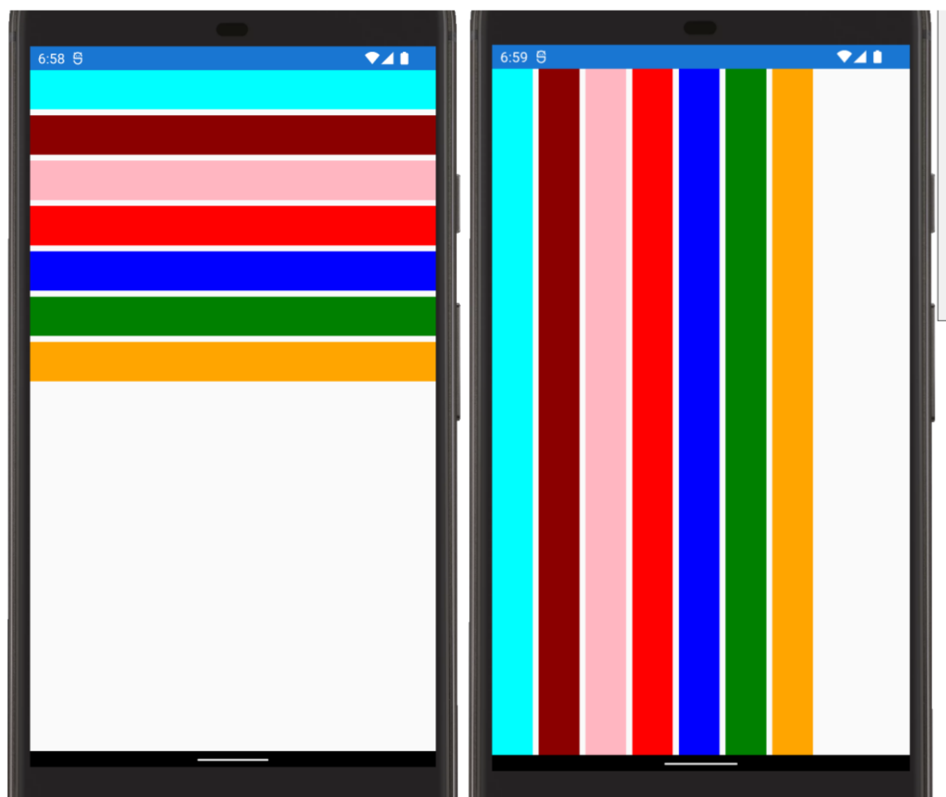

Рис. 60. Вертикальная и горизонтальная ориентации элемента StackLayout

3.2.4. Элемент AbsoluteLayout

Элемент *AbsoluteLayout* используется для размещения дочерних элементов в заранее определенных областях экранной формы. Позиция дочерних элементов задается координатами верхнего левого угла дочернего элемента относительно верхнего левого угла родительского элемента. Размеры элемента могут быть заданы значениями ширины и высоты элементов. При этом, несмотря на название, в *AbsoluteLayout* можно использовать для указания позиции и размеров элементов относительные значения. *AbsoluteLayout* следует рассматривать в первую очередь как макет «специального назначения» и использовать его в том случае, когда можно заранее определить положение и размеры дочерних элементов при условии, что они не повлияют на другие элементы и композицию страницы, особенно при ее «динамическом» изменении. Класс *AbsoluteLayout* определяет следующие свойства:

– *LayoutBounds* – присоединенное свойство, определяющее положение и размер дочернего элемента;

– *LayoutFlags* – присоединенное свойство, определяющее, считать ли границы, размеры и позиции пропорциональными.

Положение и размер элементов определяется в *AbsoluteLayout* путем задания присоединенного свойства *AbsoluteLayout.LayoutBounds* для каждого дочернего элемента с использованием абсолютных или относительных значений. Абсолютные и относительные значения можно использовать совместно в любой

комбинации. Свойство *AbsoluteLayout.LayoutBounds* можно задать с помощью двух форматов, независимо от того, используются ли абсолютные или относительные значения:

– *x, y*, где *x* и *y* указывают положение левого верхнего угла дочернего элемента относительно его родительского элемента (дочерний элемент не ограничен по размеру);

– *x, y, width, height*, где *x*- и *y*-значения указывают позицию верхнего левого угла дочернего элемента относительно родительского элемента, а *width* и *height* задают размер дочернего элемента.

С целью задания относительных значений для дочерних элементов можно использовать специальный флаг *AbsoluteLayout.LayoutFlags*. Возможные значения флага:

– *All* – все размеры интерпретируются относительно;

– *HeightProportional* – свойство высоты рассчитывается относительно высоты родительского элемента;

– *WidthProportional* – свойство ширины рассчитывается относительно ширины родительского элемента;

– *XProportional* – координата (свойство) *x* рассчитывается относительно оставшегося места по ширине родительского элемента;

– *YProportional* – координата (свойство) *y* рассчитывается относительно оставшегося места по высоте родительского элемента;

– *PositionProportional* – эквивалентно *XProportional* и *YProportional* одновременно;

– *SizeProportional* – эквивалентно *WidthProportional* и *HeightProportional* одновременно;

– *None* – все размеры интерпретируются абсолютно.

При использовании относительных значений размеры и позиция задаются в диапазоне от 0 до 1. Пример:

```
…
<AbsoluteLayout>
    <BoxView Color="Blue"
     AbsoluteLayout.LayoutBounds="50, 40, 300, 5"
     ></BoxView>
    <BoxView Color="Blue"
     AbsoluteLayout.LayoutBounds="50, 50, 300, 5"
     ></BoxView>
    <BoxView Color="Blue"
     AbsoluteLayout.LayoutBounds="70, 20, 5, 100"
     ></BoxView>
    <BoxView Color="Blue"
     AbsoluteLayout.LayoutBounds="80, 20, 5, 100"
     ></BoxView>
    <Label Text="Заголовок" FontSize="40"
     AbsoluteLayout.LayoutBounds="100, 55"
     ></Label>
```

```
    <BoxView Color="Orange"
     AbsoluteLayout.LayoutBounds="0.5, 150, 0.7, 15"
     AbsoluteLayout.LayoutFlags="XProportional,WidthProportional"
     ></BoxView>
    <BoxView Color="Green"
     AbsoluteLayout.LayoutBounds="20, 150, 5, 0.5"
     AbsoluteLayout.LayoutFlags="HeightProportional"
     ></BoxView>
    <BoxView Color="Red"
     AbsoluteLayout.LayoutBounds="1,0.5,25,100"
     AbsoluteLayout.LayoutFlags="PositionProportional"
     ></BoxView>
</AbsoluteLayout>
...
```

Скриншот работающей программы представлен на рис. 61.

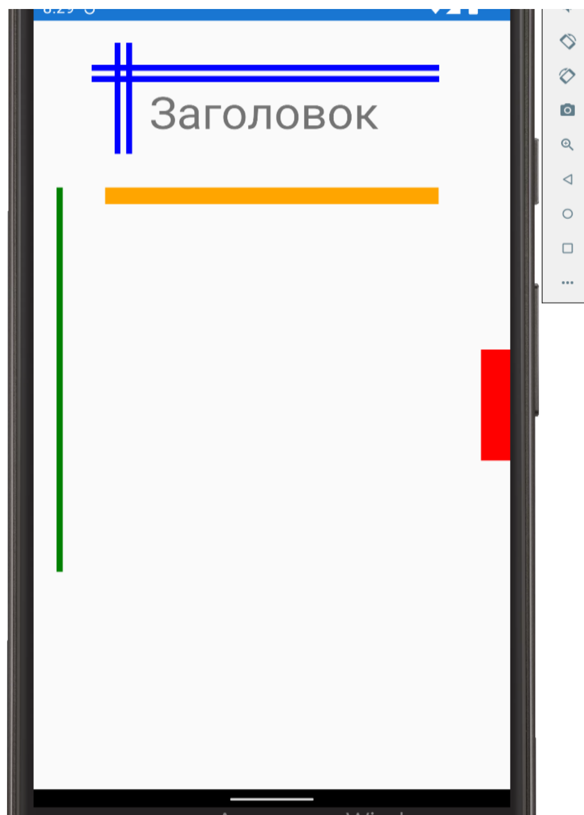

Рис. 61. Абсолютное позиционирование

Обратите внимание на задание относительных размеров и/или позиции. Например, для оранжевого прямоугольника флаг задан как *XProportional, WidthProportional*. Соответственно, позиция элемента по координате *x* и по ширине определятся относительно. Тогда значение свойства *AbsoluteLayout.LayoutBounds* «***0.5***, *150*, ***0.7***, *15*» означает, что элемент располагается по центру экранной формы (***0.5*** *по x)* и занимает по ширине 70 % формы (***0.7*** *по Width*). При этом позиции по координате *y* и по высоте заданы абсолютно: 150 и 15 соответственно.

Для красного прямоугольника флаг задан значением *PositionProportional*. Это говорит от том, что размеры элемента будут определены абсолютно, а координаты заданы относительными значениями. Тогда, установив значение свойства *AbsoluteLayout.LayoutBounds* как «***1, 0.5**, 25, 100*», мы размещаем элемент по правой стороне, а по высоте – по центру.

Запустим приложение под Windows, посмотрим, как элементы будут позиционироваться при другом соотношении сторон формы (рис. 62).

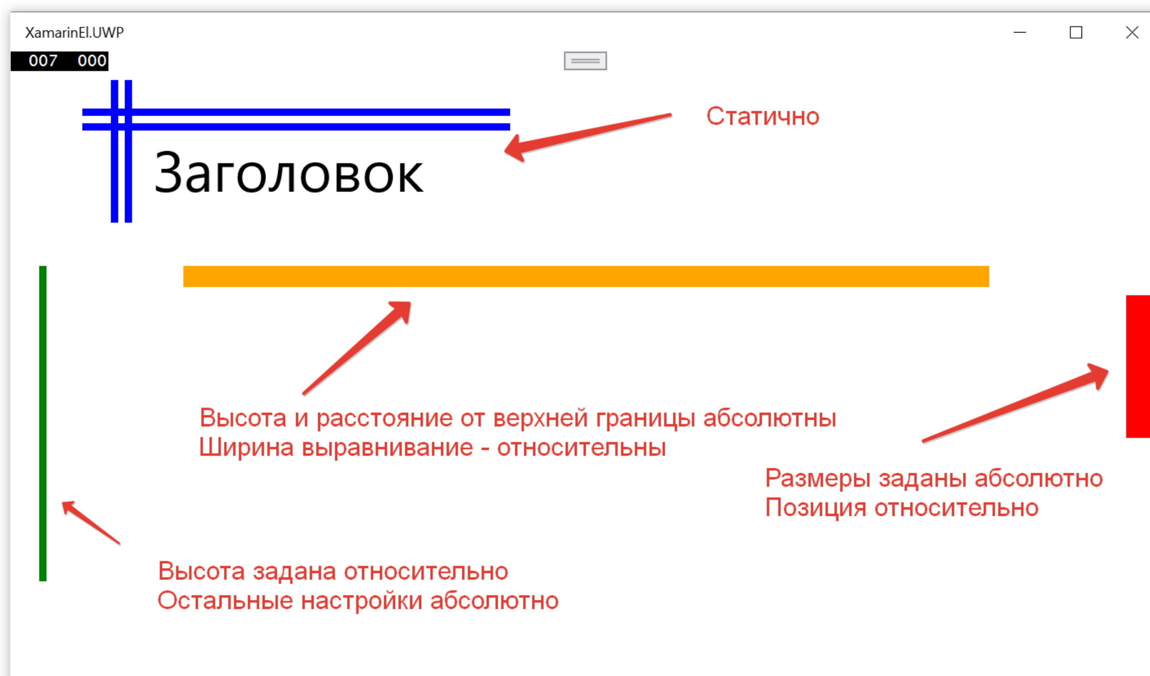

Рис. 62. Позиционирование элементов при запуске под UWP

3.2.5. Элемент RelativeLayout

Элемент *RelativeLayout* задает позиционирование вложенных элементов относительно сторон контейнера или других элементов. *RelativeLayout* позволяет создавать пользовательские интерфейсы, масштабируемые пропорционально размерам экрана в устройствах. Класс *RelativeLayout* определяет следующие свойства:

– *XConstraint* – присоединенное свойство, задает расположение по оси *x*;

– *YConstraint* – присоединенное свойство, задает расположение по оси *y*;

– *WidthConstraint* – присоединенное свойство, задает ширину элемента;

– *HeightConstraint* – присоединенное свойство, задает высоту элемента.

Обратите внимание, что в отличие от *AbsoluteLayout* в *RelativeLayout* координаты дочерних элементов можно задавать не числами, а с помощью «ограничений», представленных классом *ConstraintExpression*, включающего следующие свойства:

– *Type* – тип ограничения, который указывает, применяется ограничение относительно контейнера или других элементов;

– *Property* – свойство, на основании которого устанавливается ограничение;

– *Factor* – множитель, на который умножается длина между границами контейнера;

– *Constant* – смещение относительно контейнера или элемента;

– *ElementName* – название элемента, к которому применяется ограничение.

Сами ограничения (*Constraint*) могут быть константой, либо заданы относительно родительского элемента. Тип ограничения представлен перечислением *ConstraintType*, который определяют следующие элементы:

– *RelativeToParent* задает ограничение, которое относится к родительскому объекту;

– *RelativeToView* задает ограничение, которое относится к представлению (или элементу);

– *Constant* задает постоянное ограничение.

Пример использования элемента *RelativeLayout*:

```
...
<RelativeLayout>
    <BoxView x:Name="bb1"
Color="Orange"
     RelativeLayout.XConstraint="{ConstraintExpression Type=Constant}"
     RelativeLayout.YConstraint="{ConstraintExpression
Type=RelativeToParent, Property=Height, Factor=0.05}"
     RelativeLayout.WidthConstraint="{ConstraintExpression
Type=RelativeToParent, Property=Width, Factor=0.33}"
     RelativeLayout.HeightConstraint="{ConstraintExpression
Type=RelativeToParent, Property=Height, Factor=0.9}"
     ></BoxView>
    <BoxView Color="Blue"
     RelativeLayout.XConstraint="{ConstraintExpression Type=Constant,
Constant=50}"
     RelativeLayout.YConstraint="{ConstraintExpression
Type=RelativeToParent,  Property=Height, Factor=0.5,Constant=-25}"
     RelativeLayout.WidthConstraint="{ConstraintExpression Type=Constant,
Constant=50}"
     RelativeLayout.HeightConstraint="{ConstraintExpression
Type=Constant, Constant=50}"
     ></BoxView>
    <BoxView Color="Black"
     RelativeLayout.XConstraint="{ConstraintExpression
Type=RelativeToParent, ElementName=bb1, Property = Width, Factor=0.5}"
     RelativeLayout.YConstraint="{ConstraintExpression
Type=RelativeToParent,  Property=Height, Factor=0.275}"
     RelativeLayout.WidthConstraint="{ConstraintExpression Type=Constant,
Constant=50}"
     RelativeLayout.HeightConstraint="{ConstraintExpression
Type=RelativeToView, ElementName=bb1, Property = Height, Factor=0.5}"
     ></BoxView>
</RelativeLayout>
...
```

Результат работы приложения в ОС Windows представлен на рис. 63, а в Android – на рис. 64.

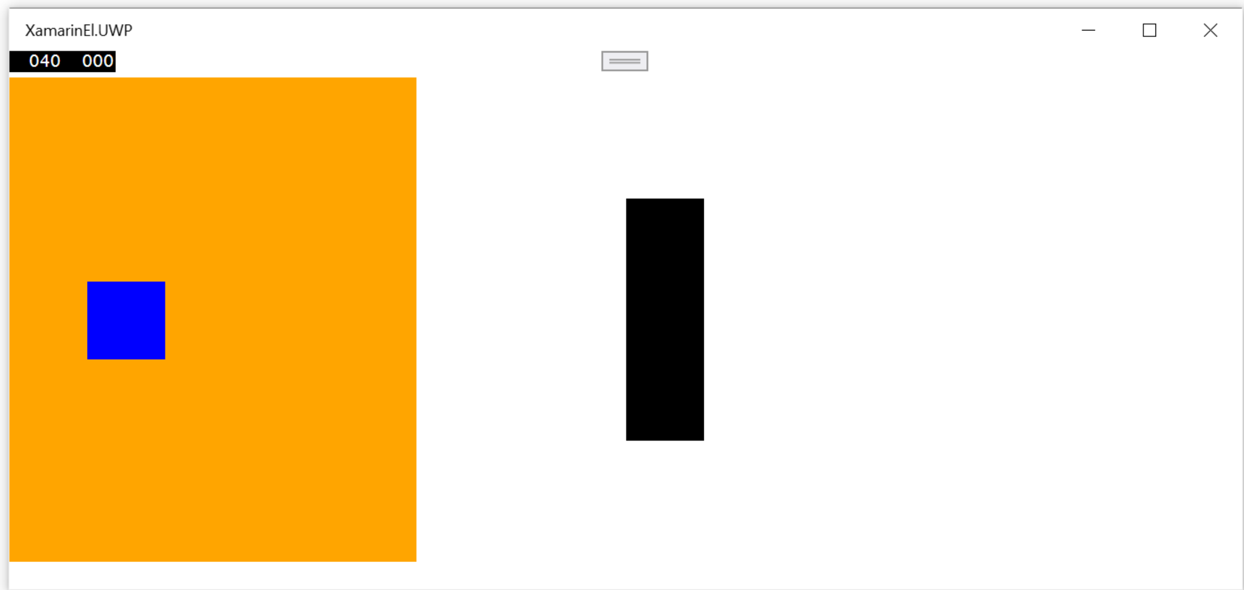

Рис. 63. Результат работы программы под Windows

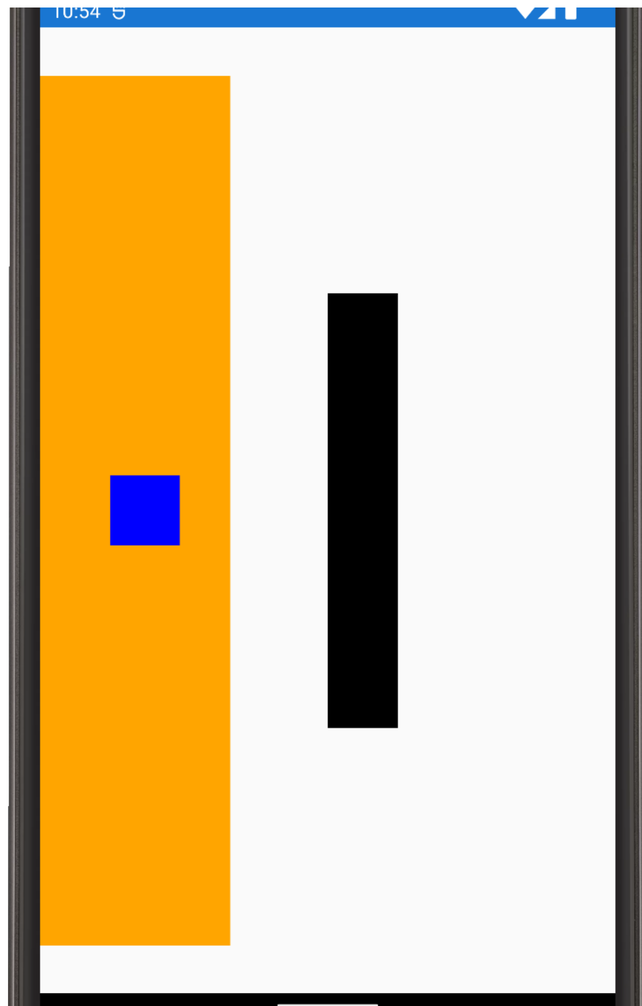

Рис. 64. Результат работы программы под Android

3.2.6. Элемент FlexLayout

Элемент *FlexLayout* появился в Xamarin.Forms в третьей версии. Его поведение основано на модуле CSS Flexible[52], который иногда называют «гибким макетом». *FlexLayout* обеспечивает эффективный способ размещения, выравнивания и распределения пространства между элементами в контейнере, при этом их размер может быть неизвестен и/или динамически изменяться. Основная идея *Flex Layout*[53] состоит в том, чтобы дать контейнеру возможность изменять ширину/высоту его элементов и порядок их следования, чтобы наилучшим образом заполнить доступное пространство (главным образом для отображения на всех типах устройств с любым размером экрана). Flex-контейнер расширяет элементы, чтобы заполнить доступное пространство, или сжимает их, чтобы предотвратить переполнение.

Класс *FlexLayout* определяет следующие свойства:

– *Direction* задает направление упорядочивания элементов в контейнере, значение по умолчанию – по строкам (*Row*) (чтобы разместить элементы в столбце, значение свойства устанавливается в *Column*, также возможно указание «реверсивных» значений, в этом случае упорядочивание будет идти не с левого, а с правого края формы);

– *AlignItems* указывает, как элементы выравниваются по оси пересечения.

Возможные значения свойства:

– *Center* указывает выравнивание по центру;

– *End* указывает, что дочерние элементы будут помещены у конца родительского элемента;

– *Start* указывает, что дочерние элементы будут помещены у начала родительского элемента;

– *Stretch* позволит растянуть все дочерние элементы от начала до конца родительского элемента;

– *JustifyContent* указывает порядок упорядочивания элементов на главной оси.

Последний параметр также имеет свои возможные значения:

– *Center* указывает, что дочерние элементы будут сгруппированы в центре родительского объекта;

– *End* указывает, что дочерние элементы будут выравниваться по концу строки;

– *SpaceAround* указывает, что дочерние элементы будут заполнять строку так, что между ними останется интервал в две единицы, а в начале и конце – интервал в одну единицу;

– *SpaceBetween* указывает, что дочерние элементы будут заполнять строку так, что между ними останутся равные интервалы, а в начале и конце интервалов не будет;

---

[52] Консорциум Всемирной паутины. URL: https://www.w3.org/TR/css-flexbox-1.

[53] Полное руководство по Flexbox. URL: https://pcnews.ru/blogs/%5Bperevod%5D_polnoe_rukovodstvo_po_flexbox-924967.html#gsc.tab=0.

– *SpaceEvenly* указывает, что между дочерними элементами будут такие же интервалы, как между крайними элементами и ближайшими краями родительского объекта;

– *Start* указывает, что дочерние элементы будут выравниваться по началу строки;

– *Wrap* используется если в строке слишком много элементов, в этом случае установка этого параметра приводит к переносу элементов в следующую строку.

Возможные значения *Wrap*:

– *NoWrap* указывает, что дочерние элементы будут упакованы в одну строку или столбец;

– *Reverse* указывает, что строки дочерних элементов будут перенесены в направлении, противоположном естественному направлению переноса для языковой локали;

– *Wrap* – указывает, что строки дочерних элементов будут перенесены в естественном направлении для языковой локали.

Кроме того, в классе определены следующие статические свойства (присоединенные):

– *Order* можно использовать для изменения порядка следования элементов, порядок задается простым числовым перечислением;

– *Basis* задает размер элемента (высоту при расположении в столбец или ширину при расположении в строку);

– *Grow* указывает, как элемент будет растягиваться по ширине или высоте контейнера;

– *Shrink* указывает, как элементы будут изменяться в размерах, если пространства контейнера недостаточно, чтобы разместить их все.

Пример переноса элементов по строкам в случае, если ширины экрана не хватает для размещения элементов:

```
…
<FlexLayout Direction="Row" Wrap="Wrap" >
     <BoxView FlexLayout.Basis="100" Color="Aqua"></BoxView>
     <BoxView FlexLayout.Basis="100" Color="DarkRed"></BoxView>
     <BoxView FlexLayout.Basis="100" Color="LightPink" ></BoxView>
     <BoxView FlexLayout.Basis="100" Color="Red" ></BoxView>
     <BoxView FlexLayout.Basis="100" Color="Blue" ></BoxView>
     <BoxView FlexLayout.Basis="100" Color="Green" ></BoxView>
     <BoxView FlexLayout.Basis="100" Color="Orange" ></BoxView>
</FlexLayout>
…
```

Результат представлен на рис. 65.

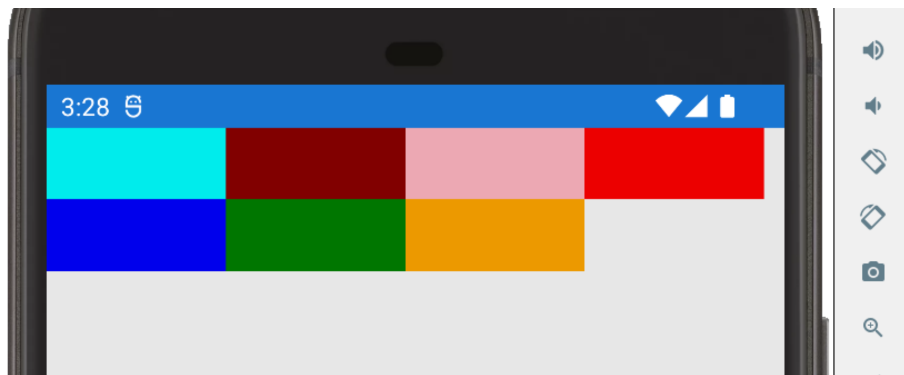

Рис. 65. Упорядочивание дочерних элементов с переносом по строке

Использование реверсивного значения для свойства *Direction:*

```
...
<FlexLayout Direction="RowReverse" Wrap="Wrap" >
     <BoxView FlexLayout.Basis="100" Color="Aqua"></BoxView>
     <BoxView FlexLayout.Basis="100" Color="DarkRed"></BoxView>
     <BoxView FlexLayout.Basis="100" Color="LightPink" ></BoxView>
     ...
</FlexLayout>
...
```

Результат представлен на рис. 66.

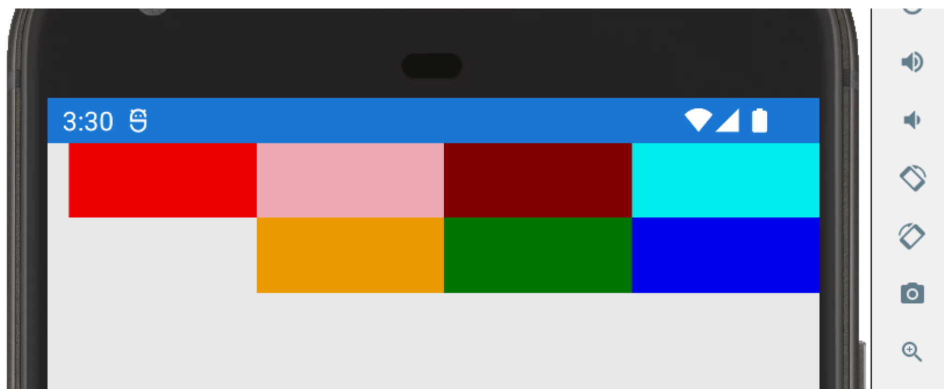

Рис. 66. Упорядочивание дочерних элементов (RowReverse) с переносом по строке

Свойство *FlexLayout.Basis* задает ширину элемента (так как в данном примере упорядочивание идет по строкам). Значение может быть задано относительно. Например, создадим элементы с шириной, указанной в процентах от ширины контейнера (высоту зададим статично):

```
...
<FlexLayout Direction="Row" Wrap="Wrap" >
     <BoxView FlexLayout.Basis="35%" HeightRequest="100"
          Color="Aqua"></BoxView>
```

```
    <BoxView FlexLayout.Basis="30%" HeightRequest="100"
          Color="DarkRed"></BoxView>
    <BoxView FlexLayout.Basis="40%" HeightRequest="100"
          Color="LightPink" ></BoxView>
    <BoxView FlexLayout.Basis="50%" HeightRequest="100"
          Color="Red" ></BoxView>
    <BoxView FlexLayout.Basis="20%" HeightRequest="100"
          Color="Blue" ></BoxView>
    <BoxView FlexLayout.Basis="35%" HeightRequest="100"
          Color="Green" ></BoxView>
    <BoxView FlexLayout.Basis="30%" HeightRequest="100"
          Color="Orange" ></BoxView>
</FlexLayout>
...
```

Результат представлен на рис. 67 и 68.

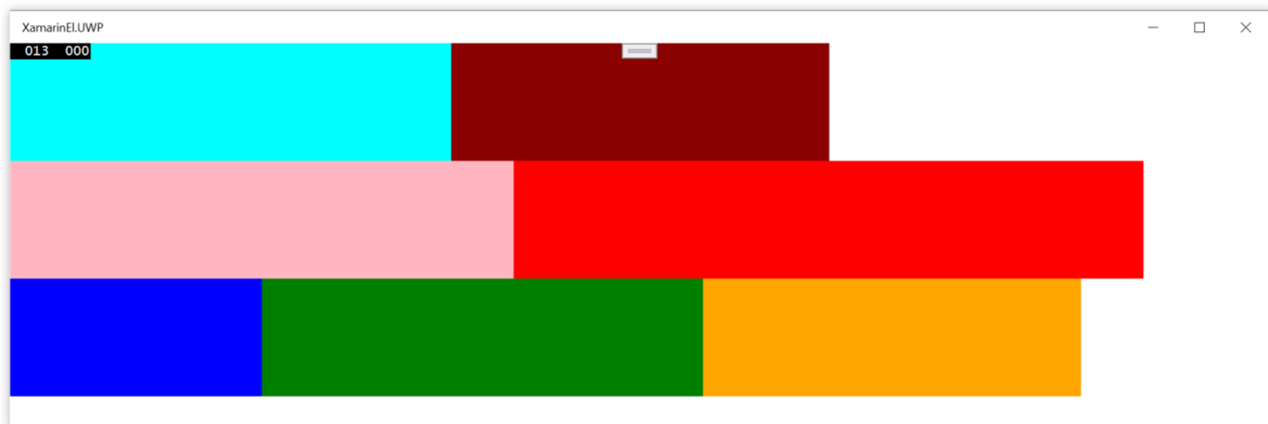

Рис. 67. Результат при компиляции под Windows

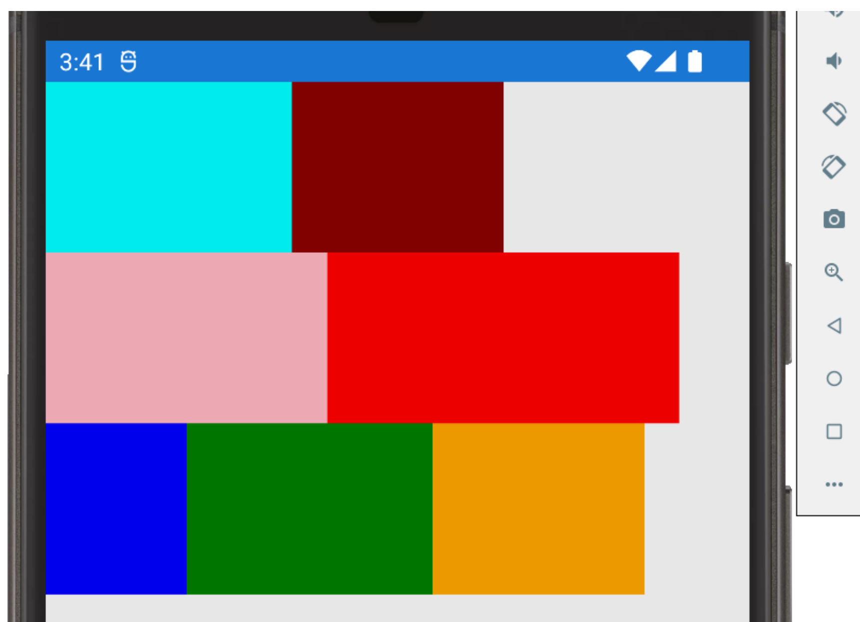

Рис. 68. Результат при компиляции под Android

Порядок следования блоков можно изменить. Пример изменения порядка элементов:

```
...
<FlexLayout Direction="Row" Wrap="Wrap" >
     <BoxView FlexLayout.Basis="100" FlexLayout.Order="1"
           Color="Aqua"></BoxView>
     <BoxView FlexLayout.Basis="100" FlexLayout.Order="7"
           Color="DarkRed"></BoxView>
     <BoxView FlexLayout.Basis="100" FlexLayout.Order="2"
           Color="LightPink" ></BoxView>
     <BoxView FlexLayout.Basis="100" FlexLayout.Order="3"
           Color="Red" ></BoxView>
     <BoxView FlexLayout.Basis="100" FlexLayout.Order="6"
           Color="Blue" ></BoxView>
     <BoxView FlexLayout.Basis="100" FlexLayout.Order="5"
           Color="Green" ></BoxView>
     <BoxView FlexLayout.Basis="100" FlexLayout.Order="4"
           Color="Orange" ></BoxView>
</FlexLayout>
...
```

Результат работы программы представлен на рис. 69.

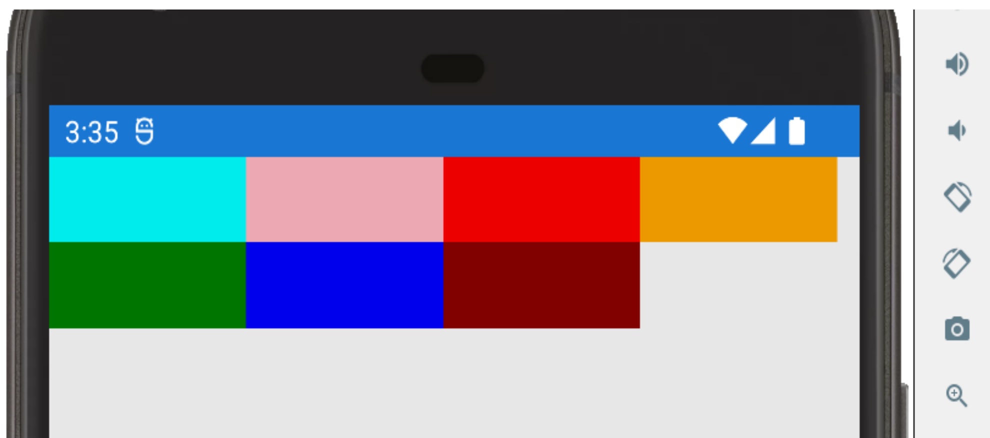

Рис. 69. Пример упорядочивания элементов с использованием присоединенного свойства FlexLayout.Order

Более сложный пример: создадим макетную страницу в формате шапка/две колонки (меню и контент)/подвал:

```
...
<FlexLayout Direction="Column"  >
    <Frame Margin="10">
```

```
        <Label Text="Шапка" FontSize="Large" TextColor="Black"
        HorizontalOptions="Center"></Label>
    </Frame>
    <FlexLayout FlexLayout.Grow="1">
        <Frame FlexLayout.Grow="1" Margin="5,5,10,5">
            <StackLayout>
        <Label Text="Основной контент" FontSize="Medium"
        TextColor="Black"></Label>
        <Label Text="Строка текста1"></Label>
        <Label Text="Строка текста2"></Label>
            </StackLayout>
        </Frame>
        <Frame FlexLayout.Basis="120" Margin="10,5,5,5"
             FlexLayout.Order="-1">
            <StackLayout>
        <Label Text="Меню" FontSize="Medium"
        TextColor="Black"></Label>
        <Label Text="Меню1"></Label>
        <Label Text="Меню2"></Label>
            </StackLayout>
        </Frame>
    </FlexLayout>
    <Frame Margin="10" BackgroundColor="White">
<Label Text="Подвал" FontSize="Large" TextColor="Black"
HorizontalOptions="Center"></Label>
    </Frame>
</FlexLayout>
...
```

Результат представлен на рис. 70 и 71.

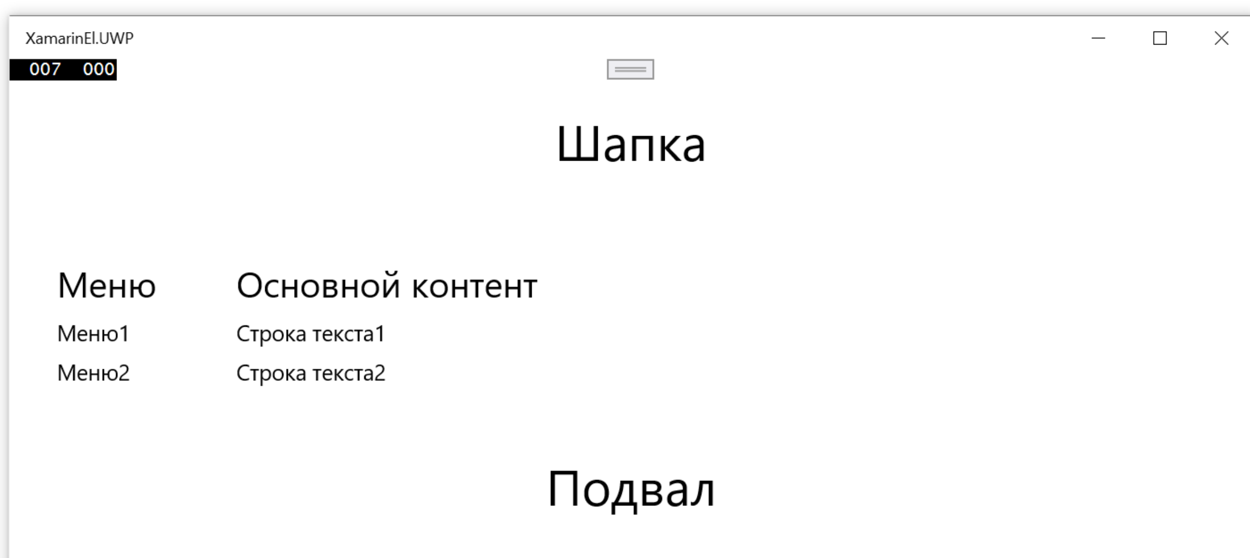

Рис. 70. Результат при компиляции под Windows

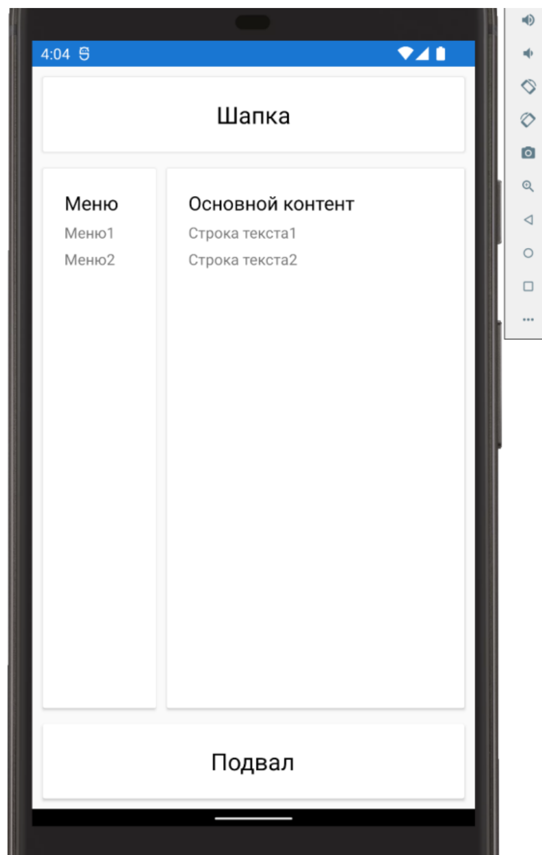

Рис. 71. Результат при компиляции под Android

Практический пример 5.
Создание элементов программным путем

При работе с XAML не следует забывать, что это просто декларативный язык, с помощью которого можно создавать объекты. По своей сути это еще один способ описания объектов на основе XML. Но все объекты, которые мы создаем с использованием XAML, являются экземплярами соответствующих классов, и работать с ними можно и без XAML напрямую из кода C#, либо комбинируя эти способы.

Задача. Нужно создать форму для работы с матрицей размерностью N*M.

Решение.

Очевидно, что эту задачу можно решить, используя элемент компоновки *Grid* для создания разметки под матрицу и элементы *Entry* для ввода значений матрицы. Проблема в том, что мы не знаем значений N и M, поэтому не можем заранее создать разметку *Grid.* Поэтому пойдем следующим путем: на языке XAML создадим базовую разметку страницы, а «генерацию» самой матрицы сделаем на языке C#.

Код XAML:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
```

```
     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
     x:Class="XamarinEl.Views.ExampleMatrixPage">
    <ContentPage.Content>
<StackLayout>
    <Frame Margin="10">
<StackLayout>
    <Label
    FontSize="Large"
    Text="Работа с матрицей"
    ></Label>
    <Label Text="Количество строк:"></Label>
    <StackLayout Orientation="Horizontal">
<Label x:Name="Nv"
    Text="0" FontSize="Large" VerticalOptions="Center"></Label>
<Stepper x:Name="Nst" Maximum="10" Minimum="2"
 ValueChanged="Nst_ValueChanged"
    ></Stepper>
    </StackLayout>
    <Label Text="Количество столбцов:"></Label>
    <StackLayout Orientation="Horizontal">
<Label x:Name="Mv"
    Text="0" FontSize="Large" VerticalOptions="Center"></Label>
<Stepper  x:Name="Mst" Maximum="10" Minimum="2"
  ValueChanged="Mst_ValueChanged"
    ></Stepper>
    </StackLayout>
    <Button Text="Создать"
    Clicked="Button_Clicked"
    ></Button>
</StackLayout>
    </Frame>
    <Frame Margin="10">
<Grid x:Name="Matrix"
      MinimumWidthRequest="400" MinimumHeightRequest="400"
      VerticalOptions="CenterAndExpand"
HorizontalOptions="CenterAndExpand"
></Grid>
    </Frame>
</StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Код C#:

```
namespace XamarinEl.Views
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class ExampleMatrixPage : ContentPage
    {
```

```
private int n;
private int m;

private int N { get => n; set { n = value;
Nv.Text = value.ToString();
Nst.Value = value;
    }
}
private int M { get => m; set { m = value;
Mv.Text = value.ToString();
Mst.Value = value;
    }
}
private Entry[,] Entries { get; set; }
public ExampleMatrixPage()
{
    InitializeComponent();
    N = 5; M = 5;

}
private void Nst_ValueChanged(object sender, ValueChangedEventArgs e)
{
    N= (int)Nst.Value;
}
private void Mst_ValueChanged(object sender, ValueChangedEventArgs e)
{
    M= (int)Mst.Value;
}
private void Button_Clicked(object sender, EventArgs e)
{
    Entries = new Entry[N, M];
    Matrix.Children.Clear();
    Matrix.ColumnDefinitions.Clear();
    Matrix.RowDefinitions.Clear();
    for (int i = 0; i < N; i++)
    {
Matrix.RowDefinitions.Add(new RowDefinition { Height=50});
for (int j = 0; j < M; j++)
{
    Matrix.ColumnDefinitions.Add(new ColumnDefinition { Width=60 });
    Entries[i, j] = new Entry { Placeholder= $"X({i+1},{j+1})"};
    Grid.SetRow(Entries[i, j],i);
    Grid.SetColumn(Entries[i, j], j);
    Matrix.Children.Add(Entries[i, j]);
}
    }
}
    }
}
```

Задача генерации матрицы возложена на обработчик события *Button_Clicked*. В этом методе мы создаем массив элементов *Entries*, по значениям N и M программным путем формируем строки и столбы *Matrix* (объект класса *Grid*, созданный на XAML) и располагаем элементы массива в ячейках таблицы. Обратите внимание на свойство *Children* класса *Grid.* Это свойство есть у всех классов-контейнеров, оно позволяет работать с дочерними элементами как с коллекцией элементов.

Результат работы программы представлена на рис. 72.

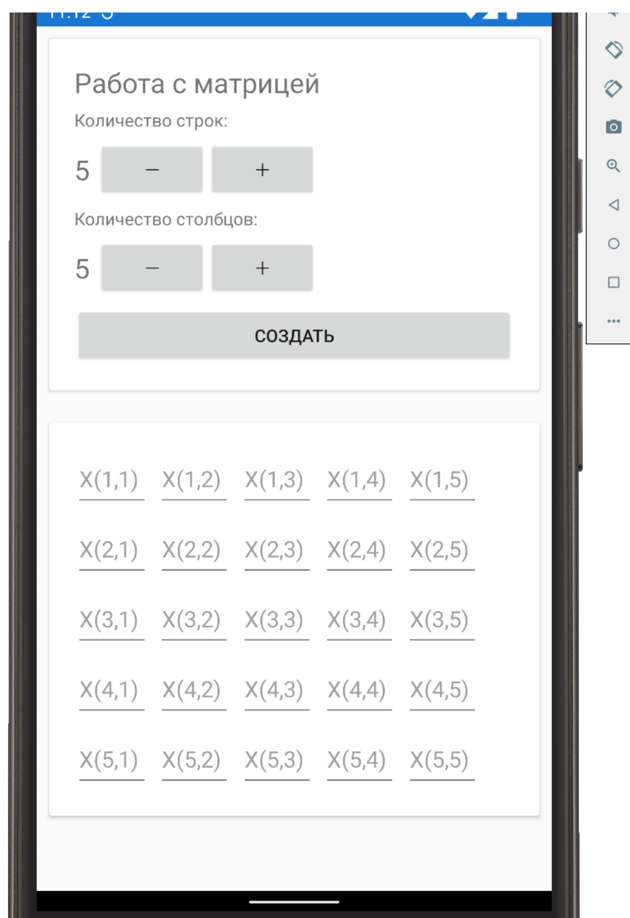

Рис. 72. Форма генерации матрицы

При нажатии на кнопку «создать» будут сгенерированы поля для ввода значений элементов матрицы указанного размера.

3.3. Многостраничные приложения в Xamarin.Forms

3.3.1. Навигация между страницами

При создании многостраничного приложения всегда встает вопрос о разработке простой и удобной навигации между страницами приложения. Xamarin предлагает несколько подходов к созданию навигации. В данном параграфе рассмотрим использование класса *NavigationPage,* а в параграфе 3.3.4 – применение оболочки *Shell*.

Класс *NavigationPage* обеспечивает иерархическую навигацию, при которой пользователь может переходить по страницам вперед и назад по своему желанию. Этот класс реализует навигацию на основе стека объектов *Page* по методу LIFO (последний зашел – первый вышел).

Для перехода с одной страницы на другую приложение помещает новую страницу в стек навигации, где она становится активной (рис. 73).

Рис. 73. Создание стека навигации

Для возврата к предыдущей странице приложение выбирает текущую страницу из стека навигации, после чего активной становится верхняя страница в стеке (рис. 74).

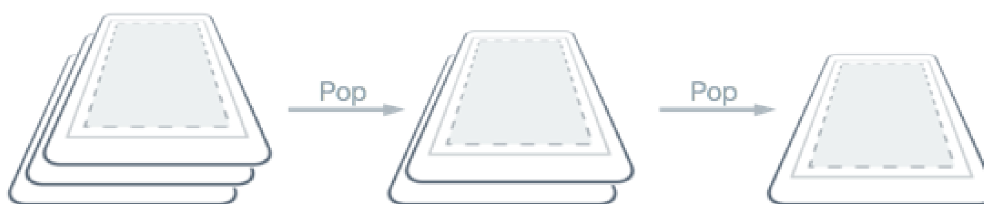

Рис. 74. Извлечение страниц из стека навигации

Методы навигации предоставляются свойством *Navigation* для любых страниц, производных от класса *Page*. Эти методы дают возможность отправлять страницы в стек навигации, извлекать страницы из стека навигации, а также выполнять операции со стеком:

– *PushAsync*(Page page);
– *PushModalAsync*(Page page);
– *PopAsync*();
– *PopModalAsync*().

Макет *NavigationPage* зависит от платформы.

В iOS панель навигации находится в верхней части страницы, где отображается заголовок и кнопка «назад», которая возвращает на предыдущую страницу.

В Android панель навигации находится в верхней части страницы, где отображается заголовок, значок и кнопка «назад», которая возвращает на предыдущую страницу. Значок определяется в атрибуте [Activity], который оформляет класс *MainActivity* в проекте, зависящем от платформы Android.

На универсальной платформе Windows панель навигации расположена в верхней части страницы, где отображается заголовок.

Кроме того, при навигации можно использовать два события: *OnAppearing()* и *OnDisappearing()*.

Событие *OnAppearing* возникает, когда пользователь перешел на эту страницу или вернуться на нее с другой в стеке. Событие *OnDisappearing* возникает, когда пользователь переходит с этой страницы на другую.

Создадим приложение, состоящие из трех страниц – одной главной и двух дочерних. Для этого добавим в проект папку Pages, а вней сформируем три страницы (рис. 75).

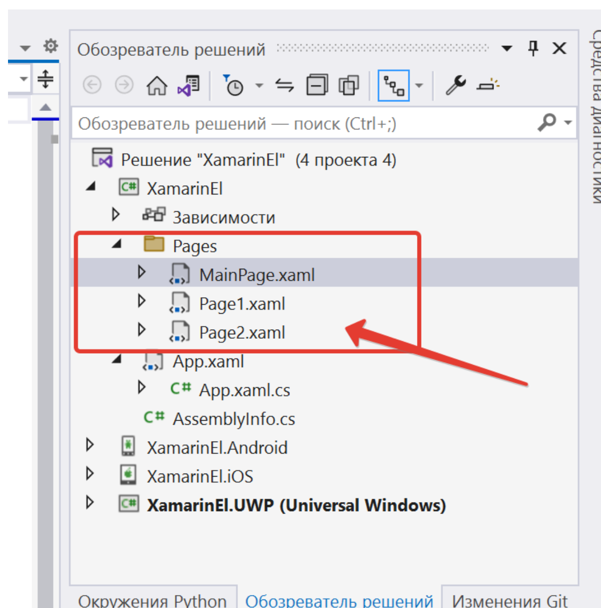

Рис. 75. Структура проекта с навигацией между страницами

Активируем навигацию в проекте. Для этого зададим корневую страницу в файле App.xaml.cs:

```
...
using XamarinEl.Pages;
...
public App()
        {
            InitializeComponent();
            MainPage = new NavigationPage(new MainPage());
        }
...
```

Код страницы *MainPage*:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Pages.MainPage"
                Title="Главная страница" >
    <ContentPage.Content>
        <StackLayout
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="CenterAndExpand">
```

```
            <Label Text="Это главная страница!"
                   FontSize="Large"
                   Margin="10">
            </Label>
            <StackLayout Orientation="Horizontal">
                <Button Text="На страницу 1" Margin="10"
                        Clicked="Button_Clicked"></Button>
                <Button Text="На страницу 2" Margin="10"
                        Clicked="Button_Clicked_1"></Button>
            </StackLayout>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Код обработчиков событий нажатия кнопок на странице *MainPage*:

```
...
private async void Button_Clicked(object sender, EventArgs e)
        {
            await Navigation.PushAsync(new Page1());
        }

        private async void Button_Clicked_1(object sender, EventArgs e)
        {
            await Navigation.PushModalAsync(new Page2());
        }
...
```

Результат работы программы представлен на рис. 76 (переход на обычную страницу) и рис. 77 (переход на модальную страницу).

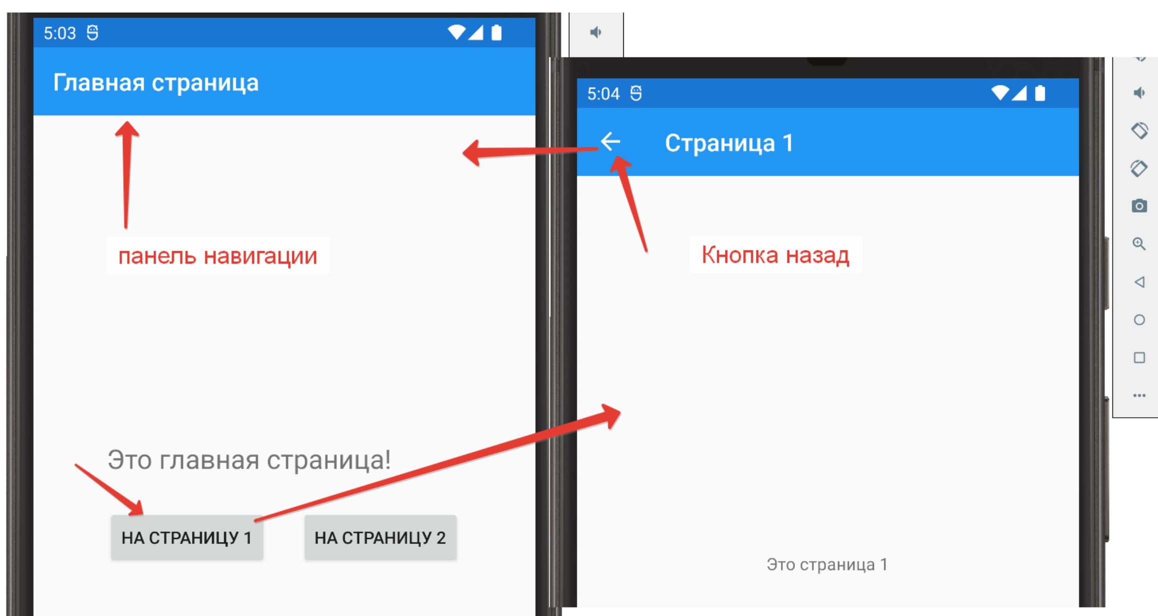

Рис. 76. Навигация, переход между двумя обычными страницами

Обратите внимание, что вторую страницу мы сделали модальной. На модальной странице пользователь должен выполнить отдельную задачу, причем он не может уйти с этой страницы, пока задача не будет выполнена или отменена. На модальной странице отсутствует панель навигации. Логику возврата с модальной страницы разработчик должен определить самостоятельно. Часто модальную страницу используют для создания страницы аутентификации пользователя в системе.

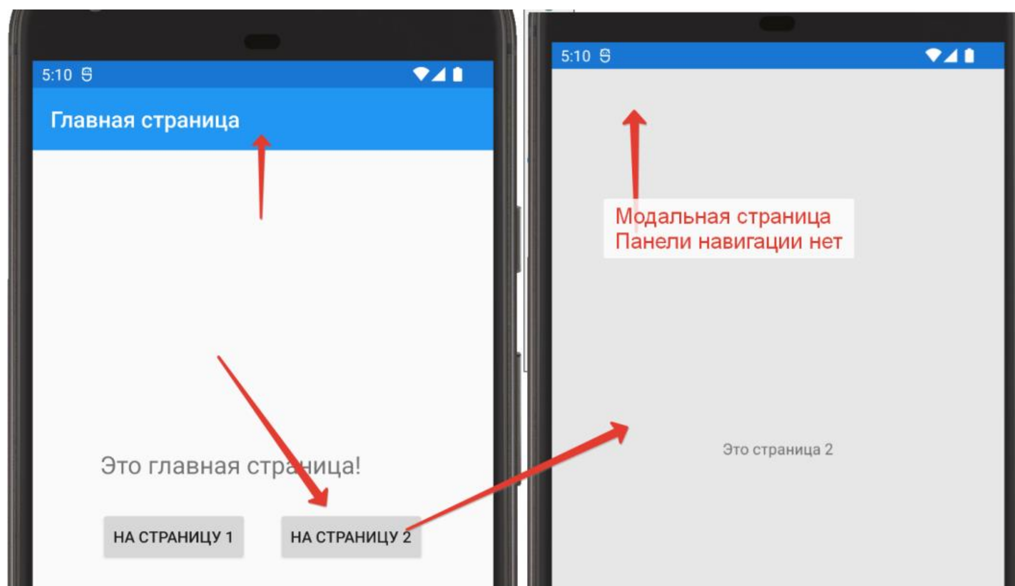

Рис. 77. Навигация, переход на модальную страницу

С помощью *Navigation* можно не только переходить между страницами, но и передавать между ними информацию. Для этого нужно использовать конструктор страницы или привязку. Применение конструктора для передачи параметров рассмотрим в следующем параграфе, а про привязку будет рассказано во второй части учебного пособия.

Практический пример 6.
Многостраничное приложение

Задание. Создадим приложение из трех страниц для работы с пользователями приложения. На главной странице должен размещаться список пользователей, доступный только аутентифицированным пользователям. Также должны быть реализованы страницы авторизации и редактирования пользователя.

Внимание! *Приведенный в примере код следует рассматривать только как демонстрацию переходов и обмена информацией между страницами, и не использовать его по «прямому» назначению.*

Определим следующие классы:

```
public class AutUser
    {
```

```
        public int Id { get; set; }
        public string Fio { get; set; }
        public string Login { get; set; }
        public string Password { get; set; }
        public override string ToString()
        {
            return $"{Id}.{Fio}\t{Login}\t{Password}";
        }
    }
    public static class PUser
    {
        private static ObservableCollection<AutUser> AutUsers = new
ObservableCollection<AutUser>();
        static PUser()
        {
            Ini();
        }
        public static ObservableCollection<AutUser> GetUsers()
        {
            return AutUsers;
        }
        public static void AddUser(AutUser autUser)
        {
            int n = AutUsers.Last().Id;
            autUser.Id = n+1;
            AutUsers.Add(autUser);
        }
        public static void AddUser(string fio, string login, string
password)
        {
            int n = AutUsers.Last().Id;
            var u = new AutUser { Id=n+1, Fio = fio, Login = login,
Password = password };
            AutUsers.Add(u);
        }
        public static void EditUser(AutUser autUser)
        {
            var u = AutUsers.First(t => t.Id == autUser.Id);
            u.Fio = autUser.Fio;
            u.Login = autUser.Login;
            u.Password = autUser.Password;
        }
        public static void EditUser(int id, string fio, string login,
string password)
        {
            var u = AutUsers.First(t => t.Id == id);
            u.Fio = fio;
            u.Login = login;
            u.Password = password;
        }
```

```
        public static void DeleteUser(int id)
        {
            var u = AutUsers.First(t => t.Id == id);
            AutUsers.Remove(u);
        }
        public static (bool IsUser, string FIO) GetUser(string login,
string password)
        {
            var u = AutUsers.FirstOrDefault(x => x.Login == login &&
x.Password == password);
            if (u != null)
                return (true, u.Fio);
            else
                return (false, "");
        }
        private static void Ini()
        {
            if (AutUsers.Count==0)
            {
                AutUsers.Add(new AutUser {Id=1, Fio="Админ админович",
Login="Admin", Password="Admin" });
            }
        }

    }
```

Класс *AutUser* будет описывать пользователя приложения, класс *PUser* содержит список всех пользователей и методы работы с ними: добавление, редактирование, удаление пользователей. Метод *GetUser* будет использоваться для аутентификации пользователя по логину и паролю. Метод *Ini()* класса *PUser* будет инициализировать одного «начального» пользователя приложения (администратора).

Разметка страницы *MainPage*:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
     x:Class="XamarinEl.Pages.MainPage"
     Title="Список пользователей">
    <ContentPage.Content>
<StackLayout
    VerticalOptions="StartAndExpand"
    HorizontalOptions="CenterAndExpand">
    <Frame Margin="10">
<StackLayout Orientation="Horizontal">
    <Label x:Name="NameUser" HorizontalOptions="StartAndExpand"
   Text="" VerticalOptions="CenterAndExpand"
   FontSize="Large" Margin="10">
```

```
    </Label>
    <Button x:Name="TButton"  Text="Вход" Margin="5"
HorizontalOptions="End"
    VerticalOptions="CenterAndExpand"
    Clicked="Button_Clicked_1"></Button>
</StackLayout>
    </Frame>
    <Frame Margin="10">
<StackLayout>
    <Label Text="Список пользователей" FontSize="Large"></Label>
    <Label x:Name="LStatus"
Text="Доступен только авторизированым пользователям!"
TextColor="Red"></Label>
    <Grid x:Name="LUser">
<Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition Height="50"></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition Width="100"></ColumnDefinition>
    <ColumnDefinition Width="100"></ColumnDefinition>
    <ColumnDefinition Width="100"></ColumnDefinition>
</Grid.ColumnDefinitions>
<ListView x:Name="ListUser" IsPullToRefreshEnabled="True"
  Grid.ColumnSpan="4"
  ItemSelected="ListUser_ItemSelected">
</ListView>
<Button x:Name="BAdd"
    Text="Добавить" Grid.Row="1" Grid.Column="1"
    Clicked="Button_Clicked"></Button>
<Button x:Name="BEdit"
    Text="Редактировать" IsEnabled="False" Grid.Row="1" Grid.Column="2"
    Clicked="BEdit_Clicked"></Button>
<Button x:Name="BDel" IsEnabled="False"
    Text="Удалить" Grid.Row="1" Grid.Column="3"
    Clicked="Button_Clicked_2"></Button>
    </Grid>
</StackLayout>
    </Frame>
</StackLayout>
    </ContentPage.Content>
</ContentPage>
```

На странице разместится два фрейма. В первом фрейме будет кнопка авторизации в приложении и поля для вывода ФИО авторизованного пользователя. Скриншот главной страницы для неавторизованного пользователя представлен на рис. 78.

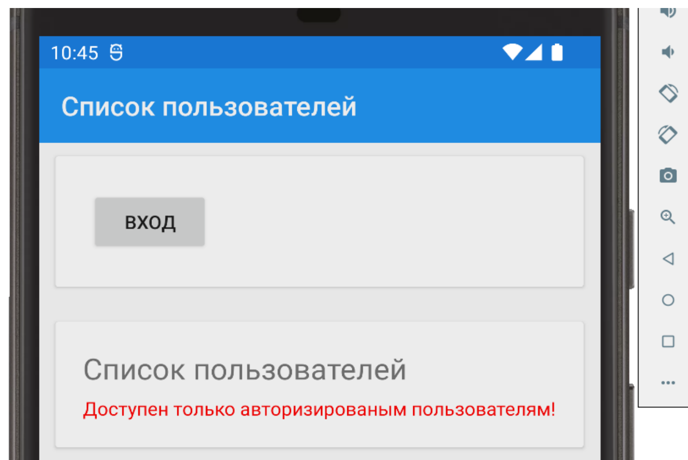

Рис. 78. Главная страница приложения

Обратите внимание, что для вывода списка пользователей мы задействовали элемент *ListView*. Он позволяет выводить на страницу сведения в списочной форме на основе заданного источника данных (свойство *ItemsSource*). Более подробно элементы для работы со списками и способы работы с ними будут рассмотрены во второй части учебного пособия.

Внесем изменение в код страницы. Добавим свойство *AutStatus*, при изменении которого будем устанавливать метку с текущим статусом аутентификации пользователя. Кроме этого добавим свойство *FioUser* для установки текстового поля – ФИО авторизованного пользователя.

Для того чтобы установить новый статус и передать ФИО аутентифицированного пользователя при успешной аутентификации, передадим страницу в конструктор страницы *Page2* при переходе на нее. Также реализуем необходимые обработчики событий.

Код в файле *MainPage.xaml.cs*:

```
public partial class MainPage : ContentPage
    {
        private bool autStatus;
        private string fioUser;
        public bool AutStatus
        {
            get => autStatus;
            set
            {
                autStatus = value;
                LUser.IsVisible = value;
                LStatus.IsVisible = !value;
                NameUser.IsVisible = value;
                if (value)
                {
                    TButton.Text = "Выход";
                }
                else
                {
                    TButton.Text = "Вход";
```

```
                }
            }
        }
        public string FioUser { get => fioUser; set {
                fioUser = value;
                NameUser.Text = value;
            }
        }
        private AutUser EditAutUser { get; set; }
        public MainPage()
        {
            InitializeComponent();
            AutStatus = false;
            ListUser.ItemsSource = PUser.GetUsers();
        }

        protected override void OnAppearing()
        {
            BEdit.IsEnabled = false;
            BDel.IsEnabled = false;
            ListUser.SelectedItem = null;
        }
        private async void Button_Clicked(object sender, EventArgs e)
        {
            if (AutStatus)
            {
                await Navigation.PushAsync(new Page1(-1));
            }
        }
        private async void Button_Clicked_1(object sender, EventArgs e)
        {
            if (!AutStatus)
            {
                await Navigation.PushModalAsync(new Page2(this));
            }
            else
            {
                AutStatus = false;
            }
        }
        private void Button_Clicked_2(object sender, EventArgs e)
        {
            var id = 
int.Parse(ListUser.SelectedItem.ToString().Split('.')[0]);
            PUser.DeleteUser(id);
        }
        private void ListUser_ItemSelected(object sender, 
SelectedItemChangedEventArgs e)
        {
            BEdit.IsEnabled = true;
            BDel.IsEnabled = true;
        }
```

```
        private async void BEdit_Clicked(object sender, EventArgs e)
        {
            var id =
int.Parse(ListUser.SelectedItem.ToString().Split('.')[0]);
            if (AutStatus)
            {
                await Navigation.PushAsync(new Page1(id));
            }
        }
    }
```

Разметка страницы аутентификации:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Pages.Page2"
             Title="Страница 2">
    <ContentPage.Content>
        <StackLayout VerticalOptions="CenterAndExpand"
                HorizontalOptions="CenterAndExpand">
            <Entry x:Name="l"  Placeholder="Введите логин"
                 />
            <Entry x:Name="p"
                Placeholder="Введите пароль" IsPassword="True"
                 />
            <Label x:Name="err" TextColor="Red"></Label>
            <Button Text="Авторизация" Clicked="Button_Clicked"></Button>
            <Button Text="Назад" Clicked="Button_Clicked_1"></Button>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Экранная форма страницы авторизации пользователя представлена на рис. 79.

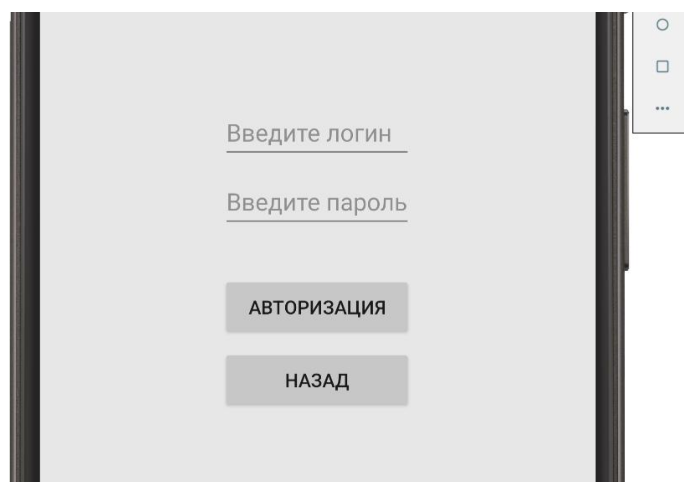

Рис. 79. Страница авторизации

Код страницы аутентификации:

```
public partial class Page2 : ContentPage
    {
        private ContentPage _page;
        public Page2(ContentPage page)
        {
            InitializeComponent();
            _page = page ;
        }
        private async void Button_Clicked(object sender, EventArgs e)
        {
            var u = PUser.GetUser(l.Text,p.Text);
            if (u.IsUser)
            {
                err.Text = "";
                (_page as MainPage).FioUser = u.FIO;
                (_page as MainPage).AutStatus = true ;
                await Navigation.PopModalAsync();
            }
            else
            {
                (_page as MainPage).FioUser = "";
                (_page as MainPage).AutStatus = false;
                err.Text = "Пользователя не существует";
            }
        }
        private async void Button_Clicked_1(object sender, EventArgs e)
        {
            await Navigation.PopModalAsync();
        }
    }
```

На странице определено поле типа *ContentPage*. Оно будет принимать страницу, с которой был осуществлен переход на страницу аутентификации и, соответственно, изменен конструктор по умолчанию.

Метод *Button_Clicked* определяет обработчик события нажатия кнопки аутентификации, выполняется проверка, есть ли пользователь с введенным логином и паролем, и если проверка прошла успешно, то на странице, с которой был осуществлен переход, устанавливаем свойство *FioUser* как ФИО пользователя, прошедшего аутентификацию, свойство *AutStatus,* что все прошло успешно, и возвращаемся на эту страницу. Если логин или/и пароль не верны, то выводим сообщение, что пользователь не найден, и остаемся на странице. По нажатию кнопки «назад» можно просто вернуться на прошлую страницу без прохождения процедуры аутентификации.

Разметка страницы добавления или редактирования пользователя:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

```
            x:Class="XamarinEl.Pages.Page1"
            Title="Страница 1">
    <ContentPage.Content>
        <StackLayout>
            <TableView>
                <TableView.Root>
                    <TableSection Title="Информация о пользователе">
                        <EntryCell x:Name="FIO" Label="ФИО пользователя"
></EntryCell>
                        <EntryCell x:Name="Login" Label="Логин"
></EntryCell>
                        <EntryCell x:Name="Pass" Label="Пароль"
></EntryCell>
                    </TableSection>
                </TableView.Root>
            </TableView>
            <Button Text="Сохранить" Clicked="Button_Clicked"></Button>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Скриншот страницы редактирования пользователей представлен на рис. 80.

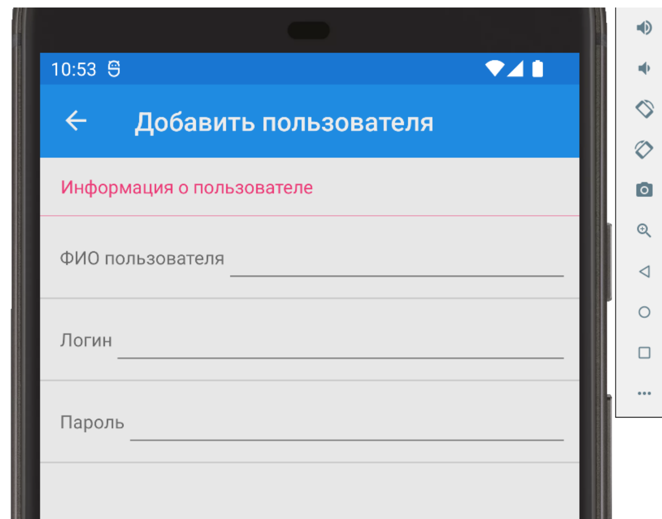

Рис. 80. Страница добавления/редактирования пользователя

Код страницы:

```
public partial class Page1 : ContentPage
    {
        private AutUser AutUser { get; set; }
        private bool IsEdit { get; set; }
        public Page1(int id)
        {
            InitializeComponent();
            if (id == -1)
```

```
            {
                AutUser = new AutUser();
                this.Title = "Добавить пользователя";
                IsEdit = false;
            }
            else
            {
                AutUser = PUser.GetUsers().First(t => t.Id == id);
                FIO.Text = AutUser.Fio;
                Login.Text = AutUser.Login;
                Pass.Text = AutUser.Password;
                this.Title = "Редактировать пользователя";
                IsEdit = true;
            }
        }
        private async void Button_Clicked(object sender, EventArgs e)
        {
            if (IsEdit)
            {
                PUser.EditUser(AutUser.Id, FIO.Text, Login.Text,
Pass.Text);
            }
            else
            {
                AutUser.Fio = FIO.Text;
                AutUser.Login = Login.Text;
                AutUser.Password = Pass.Text;
                PUser.AddUser(AutUser);
            }
            await Navigation.PopAsync();
        }
    }
```

Результат работы программы представлен на рис. 81.

При нажатии на кнопку «вход» переходим на страницу авторизации (*Page2*). Если допущена ошибка в логине и/или пароле, выводится соответствующее сообщение. Если логин и пароль введены верно, возвращаемся на главную страницу, в верхнем фрейме появляется надпись – ФИО авторизованного пользователя, текст кнопки меняется на «выход». Также становится виден второй фрейм со списком пользователей. При нажатии кнопки «добавить» открывается страница для создания нового пользователя (*Page1*). Кнопки «редактировать» и «удалить» станут активны после того, как в списке пользователей будет выбрана одна запись. В этом случае переход также идет на страницу *Page1*, но при переходе на нее мы передаем идентификатор выделенного пользователя. В результате данные пользователя подставляются в поля редактирования.

Однако при редактировании мы столкнемся со следующей проблемой: изменение значений полей не будет отражаться в списке (в отличие от добавления/удаления пользователей). Это наглядно иллюстрируют рис. 82 и 83.

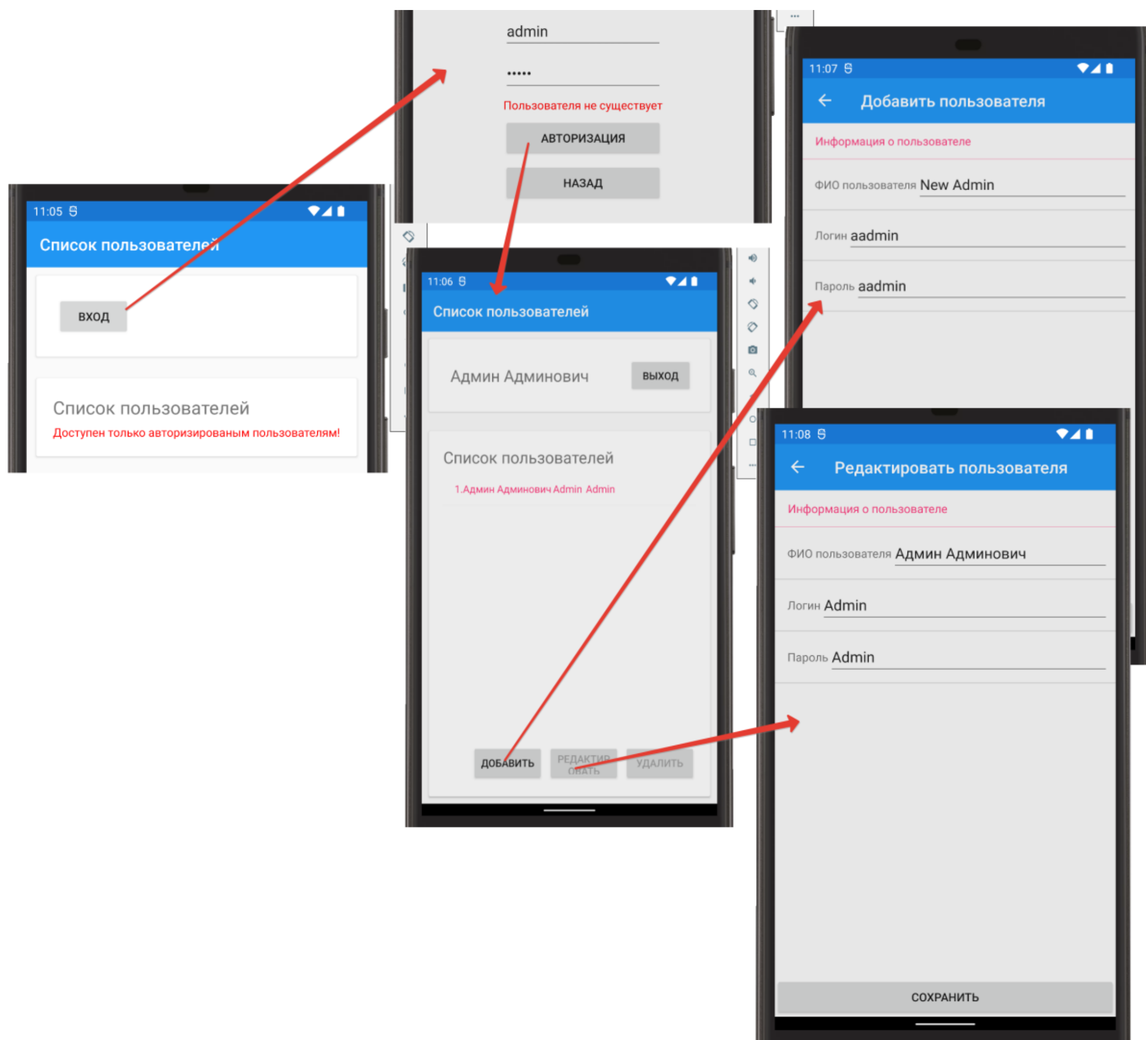

Рис. 81. Пример работы приложения

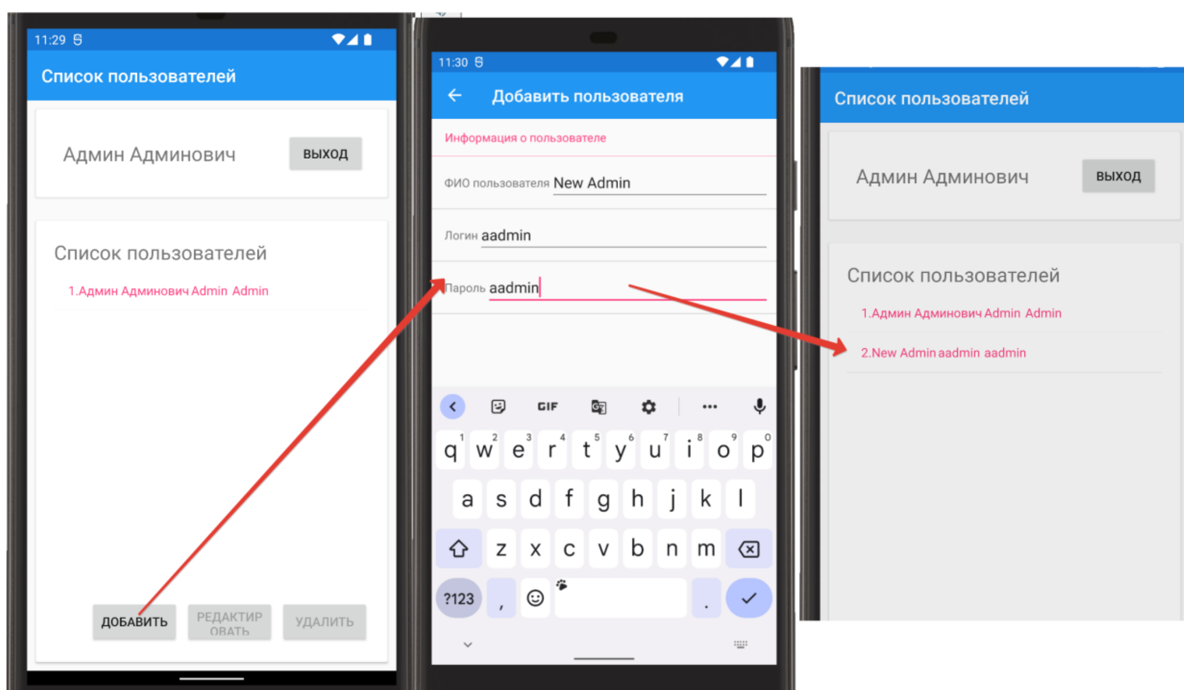

Рис. 82. Добавление записи

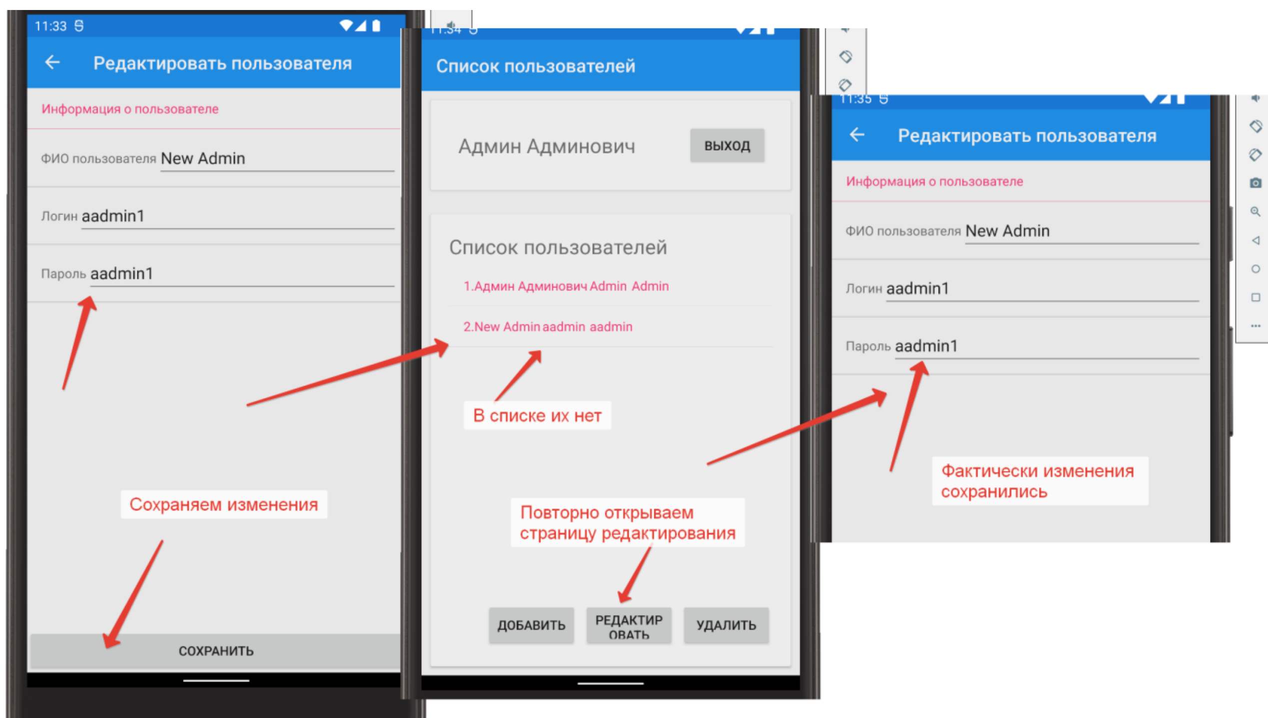

Рис. 83. Редактирование записей

Дело в том, что, во-первых, источники данных для элемента *ListView* должны «уметь уведомлять» представление о произошедших изменениях. Для этого они должны реализовывать интерфейсы *INotifyPropertyChanged* (для базовых классов) и *INotifyCollectionChanged* (для коллекций). В классе *PUser* для создания коллекции мы использовали не стандартный список *List<T>*, а коллекцию класса *ObservableCollection<T>* (реализует интерфейс *INotifyCollectionChanged*, в связи с чем обладает свойством уведомлять об изменениях), поэтому при добавлении нового элемента в коллекцию он автоматически отображается на странице.

Класс *AutUser* интерфейс *INotifyPropertyChanged* не реализует. Во-вторых, реализация должна идти не через метод *ToString()*, а через механизм привязки *Binding* к свойствам класса. Так как рассмотрение механизма привязки будет во второй части пособия, то в качестве временного решения можно внести следующее изменение в код метода – *OnAppearing* (рис. 84, строки 52 и 53).

```
            Ссылок: 0
50          protected override void OnAppearing()
51          {
52              ListUser.ItemsSource = null;
53              ListUser.ItemsSource = PUser.GetUsers();
54              BEdit.IsEnabled = false;
55              BDel.IsEnabled = false;
56              ListUser.SelectedItem = null;
57          }
```

Рис. 84. Изменение, внесенное в код метода

Таким образом мы заставим страницу при каждом переходе на нее сбрасывать, а затем заново устанавливать источник данных, тем самым принудительно обновляя все значения. Решение с привязкой будет представлено во второй части учебного пособия.

3.3.2. Типы страниц в Xamarin.Forms

Базовым классом для всех страниц в Xamarin является *Page*. В прошлых параграфах при создании приложений мы познакомились с основной страницей, представленной классом *ContentPage*, и навигационной страницей *NavigationPage*. Кроме этих двух типов страниц при создании приложения можно использовать следующие:

– *TabbedPage* – состоит из списка вкладок и большой области сведений, где каждая вкладка загружает содержимое в область сведений;

– *CarouselPage* – страница, в рамках которой пользователи могут «пролистывать» содержимое из стороны в сторону, таким образом проходя по всем по страницам;

– *MasterDetailPage* – страница, управляющая двумя страницами связанных сведений – всплывающей страницей, которая представляет элементы, и страницей сведений, которая отображает сведения об элементах на всплывающей странице.

Схематично все страницы изображены на рис. 85.

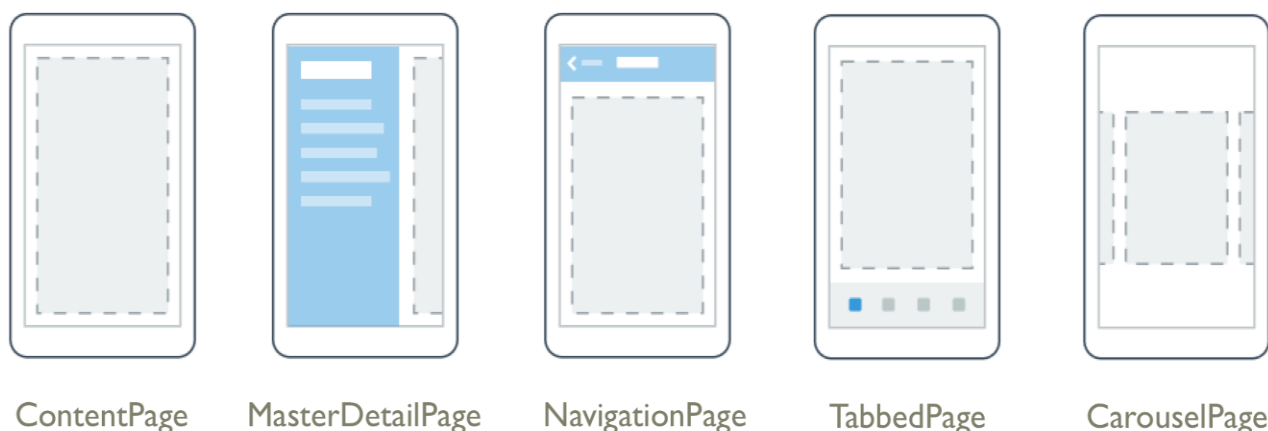

Рис. 85. Основные типы страниц

Используя страницу *MasterDetailPage,* можно создать приложение с главным меню.

Внимание! *MasterDetailPage считается устаревшим типом страницы и не рекомендуется к использованию в новых проектах. Вместо MasterDetailPage следует использовать FlyoutPage и оболочку Shell.*

Для демонстрации добавим в проект четыре страницы: одна будет главной (по сути меню приложения), три других – обычные страницы (страницы деталей). Структура проекта представлена на рис. 86.

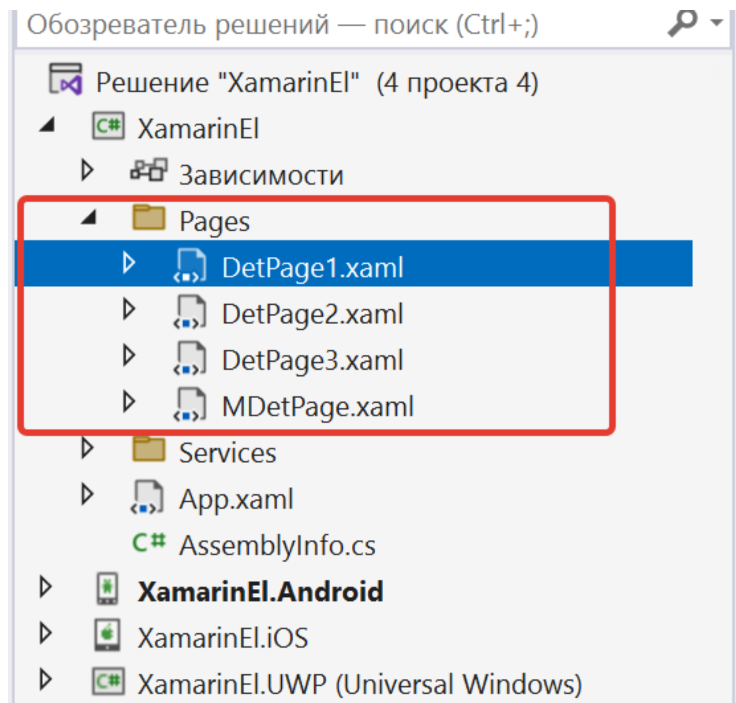

Рис. 86. Структура проекта

Разметка главной страницы *MDetPage*:

```
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Pages.MDetPage">
    <MasterDetailPage.Master >
        <ContentPage Padding="10" BackgroundColor="White" Title="Главная
страница" Icon="hamburger.png">
            <ContentPage.Content>
                <StackLayout Margin="5">
                    <Label Text="Главное меню" FontSize="Large">
                    </Label>
                    <Button  Text="Страница 1"
Clicked="Button_Clicked"></Button>
                    <Button  Text="Страница 2"
Clicked="Button_Clicked_1"></Button>
                    <Button  Text="Страница 3"
Clicked="Button_Clicked_2"></Button>
                </StackLayout>
            </ContentPage.Content>
        </ContentPage>
    </MasterDetailPage.Master>
    <MasterDetailPage.Detail>
        <ContentPage Padding="10">
            <ContentPage.Content>
                <StackLayout Margin="5">
                    <Label Text="Detail Page">
                    </Label>
                </StackLayout>
```

```
            </ContentPage.Content>
        </ContentPage>
    </MasterDetailPage.Detail>
</MasterDetailPage>
```

Код страницы *MDetPage*:

```
  public partial class MDetPage : MasterDetailPage
    {
        public MDetPage()
        {
            InitializeComponent();
            Detail = new NavigationPage(new DetPage1());
            IsPresented = false;
        }
        private void Button_Clicked(object sender, EventArgs e)
        {
            Detail = new NavigationPage(new DetPage1());
            IsPresented = false;
        }
        private void Button_Clicked_1(object sender, EventArgs e)
        {
            Detail = new NavigationPage(new DetPage2());
            IsPresented = false;
        }
        private void Button_Clicked_2(object sender, EventArgs e)
        {
            Detail = new NavigationPage(new DetPage3());
            IsPresented = false;
        }
    }
```

Свойство *IsPresented* указывает, должно ли меню *MasterDetail* быть скрыто после касания.

Страницы *DetPage1*, *DetPage2 и DetPage3* для примера будут содержать различающийся порядковым номером текст с названием страницы (на примере *DetPage1*):

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Pages.DetPage1"
             Title="Страница 1">
    <ContentPage.Content>
        <StackLayout>
            <Label Text="Это страница 1"
```

```
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="CenterAndExpand" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Пример работы приложения приведен на рис. 87 для Android и рис. 88 при запуске под ОС Windows.

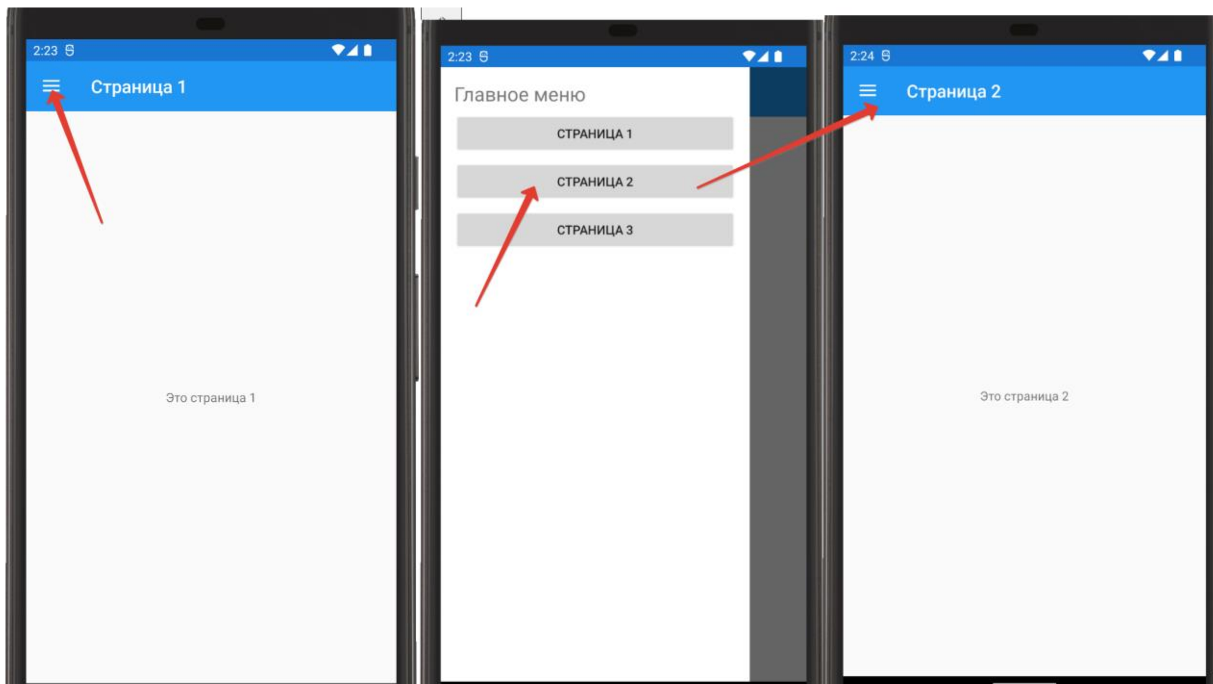

Рис. 87. Пример MasterDetailPage

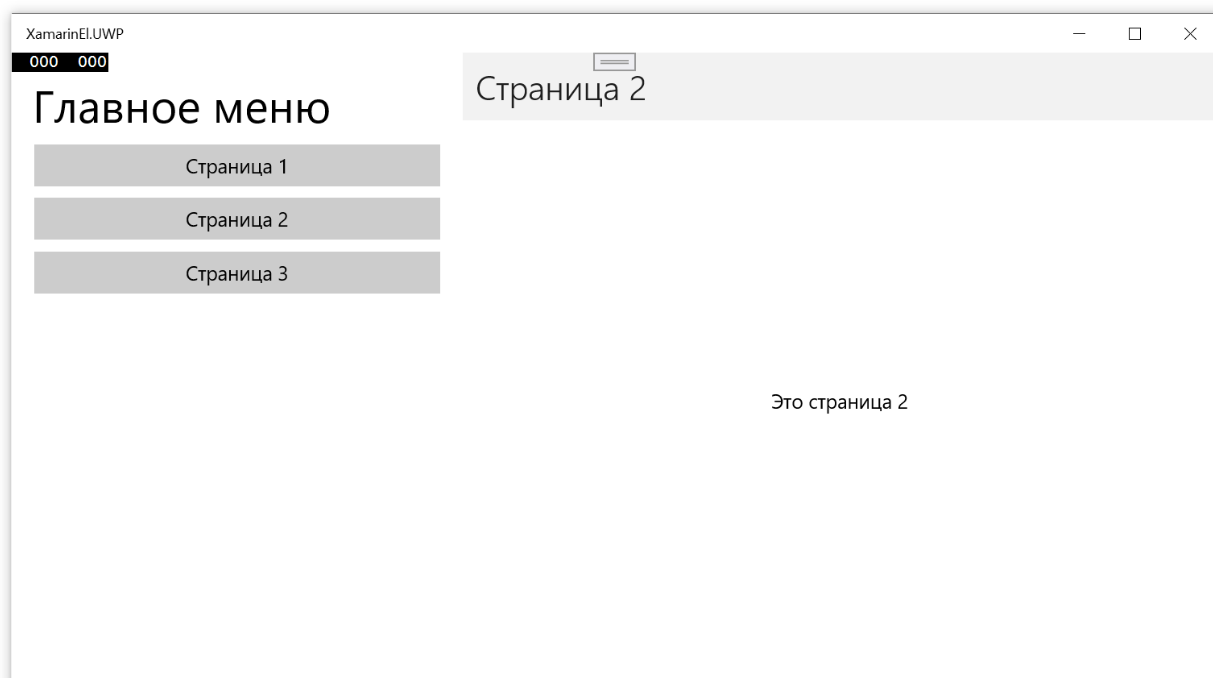

Рис. 88. Пример MasterDetailPage на операционной системе Windows

TabbedPage[54] позволяет создать пользовательский интерфейс страницы с вкладками, состоит из списка вкладок и области сведений. В iOS список вкладок отображается в нижней части экрана, а также в области данных сверху. На устройствах с ОС Android и Windows вкладки отображаются в верхней части экрана. Пользователь может прокрутить коллекцию вкладок, которые находятся в верхней части экрана, если эта коллекция слишком велика, чтобы поместиться на одном экране.

Добавим в проект страницу *TabbedPage,* разметка страницы:

```
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Pages.TPage">
    <ContentPage Title="Страница 1" >
        <ContentPage.Content>
            <StackLayout>
                <Label Text="Страница 1" FontSize ="Large"></Label>
            </StackLayout>
        </ContentPage.Content>
    </ContentPage>
    <ContentPage Title="Страница 2" >
        <ContentPage.Content>
            <Frame Margin="10">
                <StackLayout Orientation="Vertical">
                    <Label Text="Выберите один из элементов 
списка"></Label>
                    <Label x:Name="Rez"></Label>
                    <RadioButton Content="Элемент 1" GroupName="group1" 
></RadioButton>
                    <RadioButton Content="Элемент 2" GroupName="group1" 
></RadioButton>
                    <RadioButton Content="Элемент 3" GroupName="group1" 
></RadioButton>
                    <RadioButton Content="Элемент 4" GroupName="group1" 
></RadioButton>
                </StackLayout>
            </Frame>
        </ContentPage.Content>
    </ContentPage>
    <ContentPage Title="Страница 3" >
        <Frame Margin="10">
            <StackLayout>
                <Label Text="Фотография" FontSize="Large"></Label>
                <Frame BorderColor="Blue" >
                    <Image Aspect="AspectFit" Source="_MG_3539.jpg">
                    </Image>
                </Frame>
            </StackLayout>
```

[54] Microsoft Learn.

```
        </Frame>
    </ContentPage>
</TabbedPage>
```

Результат представлен на рис. 89.

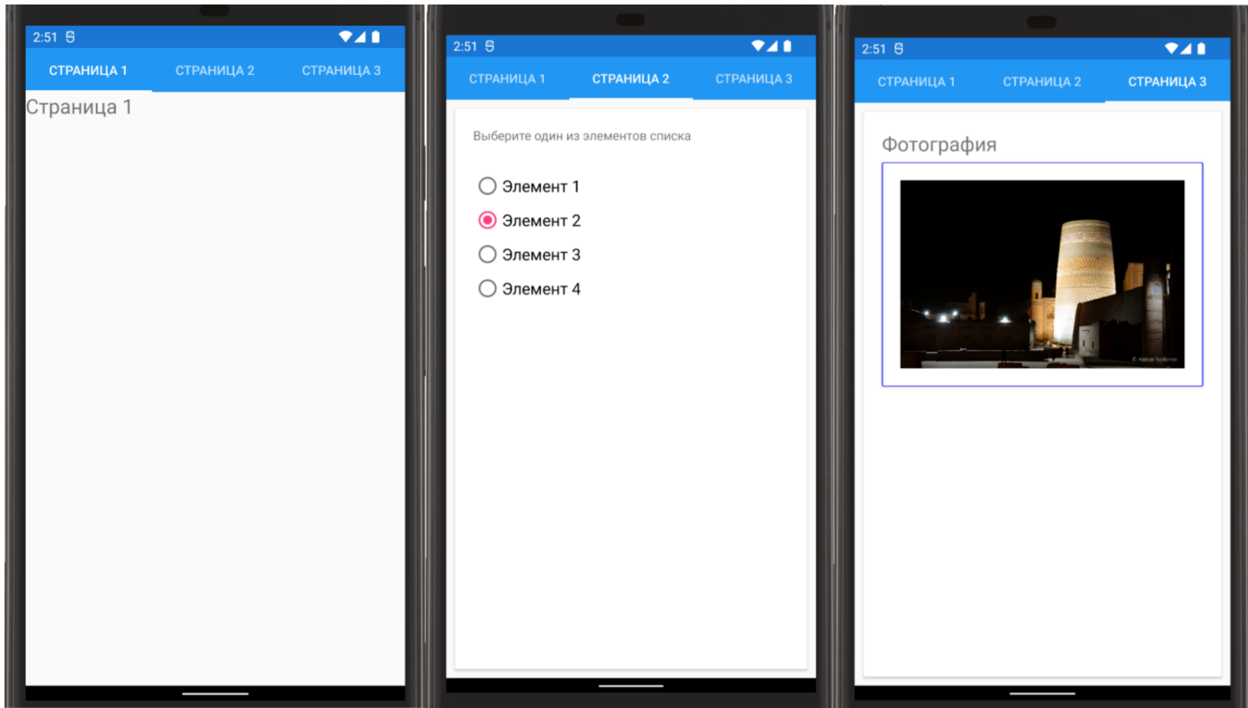

Рис. 89. Пример TabbedPage

CarouselPage[55] позволяет создать интерфейс, который можно «пролистывать» при помощи манипулятора или просто пальцем на сенсорном экране:

```
<CarouselPage  xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Pages.CPage">
    <ContentPage Title="Страница 1" >
        <ContentPage.Content>
            <Frame Margin="10">
                <StackLayout>
                    <Label Text="Страница 1" FontSize ="Large"></Label>
                    <Label Text="Детали страницы 1"></Label>
                </StackLayout>
            </Frame>
        </ContentPage.Content>
    </ContentPage>
    <ContentPage Title="Страница 2" >
        <ContentPage.Content>
            <Frame Margin="10">
```

[55] Microsoft Learn.

```
                <StackLayout Orientation="Vertical">
                    <Label Text="Выберите один из элементов списка"></Label>
                    <Label x:Name="Rez"></Label>
                    <RadioButton Content="Элемент 1" GroupName="group1" ></RadioButton>
                    <RadioButton Content="Элемент 2" GroupName="group1" ></RadioButton>
                    <RadioButton Content="Элемент 3" GroupName="group1" ></RadioButton>
                    <RadioButton Content="Элемент 4" GroupName="group1" ></RadioButton>
                </StackLayout>
            </Frame>
        </ContentPage.Content>
    </ContentPage>
    <ContentPage Title="Страница 3" >
        <Frame Margin="10">
            <StackLayout>
                <Label Text="Фотография" FontSize="Large"></Label>
                <Frame BorderColor="Blue" >
                    <Image Aspect="AspectFit" Source="_MG_3539.jpg">
                    </Image>
                </Frame>
            </StackLayout>
        </Frame>
    </ContentPage>
</CarouselPage>
```

Результат представлен на рис. 90.

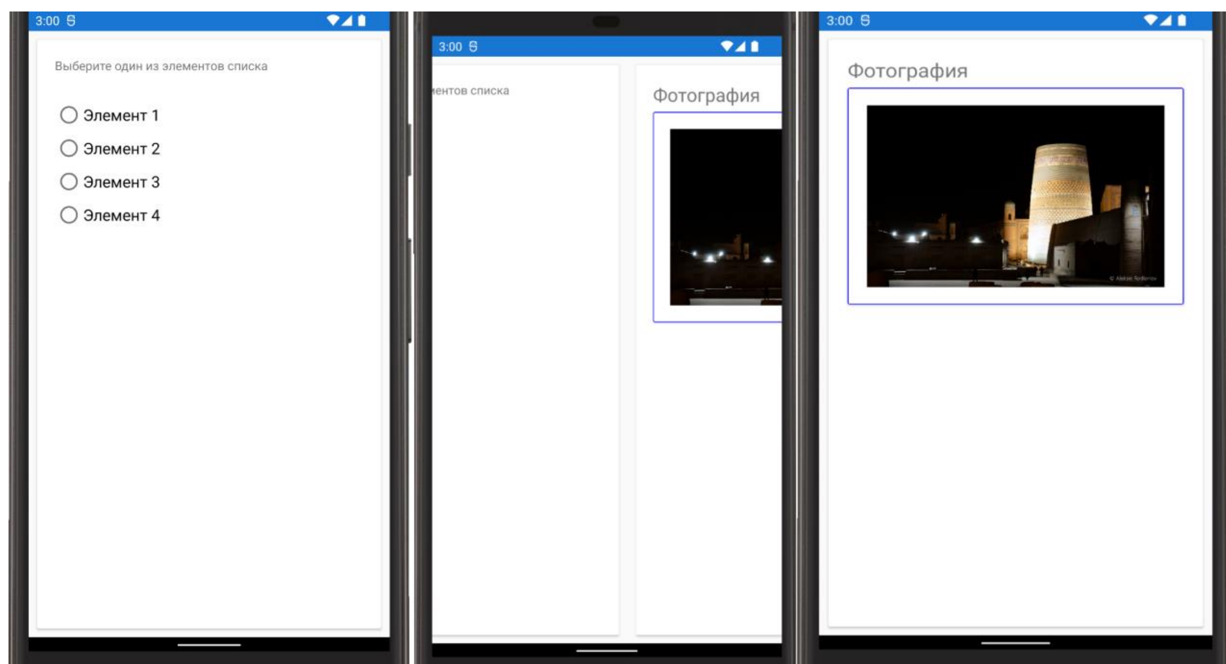

Рис. 90. «Пролистывание» страниц в CarouselPage

3.3.3. Xamarin.Forms Shell

Начиная с версии Xamarin.Forms 4.0 разработчикам стала доступна оболочка *Shell*[56], предназначенная для упрощения создания кроссплатформенных приложений и включающая в себя следующий функционал: боковое меню, вкладки, навигация, поиск.

Класс *Shell*[57] наследуется от класса *Page* и, по сути, представляет собой контейнер, который может содержать следующие объекты:

– *FlyoutItem* представляет собой один или несколько элементов всплывающего меню и применяется, если шаблон навигации требует использования всплывающего меню;

– *TabBar* представляет собой панель вкладок, используется, если навигация в приложении начинается с нижней панели вкладок;

– *Tab* – сгруппированное содержимое с навигацией по нижним вкладкам;

– *ShellContent* – объекты *ContentPage* для каждой вкладки.

Оболочка *Shell* также включает встроенные функции поиска, предоставляемые классом *SearchHandler*. Чтобы добавить на страницу функцию поиска, создайте производный объект *SearchHandler*. После этого в верхней части страницы появится поле поиска.

Внимание! *Оболочка Shell на Xamarin.Forms полностью доступна в iOS и Android, но на универсальной платформе Windows (UWP) оболочка является экспериментальной и для ее активации необходимо добавить следующую строку кода в класс App в проекте UWP перед вызовом Forms.Init:*

global::Xamarin.Forms.Forms.SetFlags("Shell_UWP_Experimental");

FlyoutItem представляет собой элемент навигационного меню, который пользователи могут выбирать для перехода на разные части приложения. Каждый *FlyoutItem* может содержать одну или несколько вкладок *Tab*, и каждая вкладка – одну или несколько страниц *ShellContent*.

Основные свойства:

– *FlyoutItem.Title* определяет заголовок всплывающего элемента;

– *FlyoutItem.Icon* определяет значок всплывающего элемента;

– *FlyoutItem.FlyoutDisplayOptions* определяет, как всплывающий элемент и его дочерние элементы отображаются во всплывающем окне (главном меню).

Возможные значения:

– *AsSingleItem* указывает, что все объекты *Tab* в *FlyoutItem* в меню представлены одним элементом;

– *AsMultipleItems* указывает на то, что сам элемент и его дочерние элементы будут отображаться во всплывающем меню как группа элементов.

Ширину и высоту всплывающего окна можно настроить, установив значения присоединенных свойств *Shell.FlyoutWidth* и *Shell.FlyoutHeight.*

---

[56] Microsoft Developer Blogs. URL: https://devblogs.microsoft.com/xamarin/introducing-xamarin-forms-4-0-the-era-of-shell.

[57] Microsoft Learn.

Пример использования *FlyoutItem*:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Pages.ShPage">
    <FlyoutItem Title="Просто текст">
        <Tab>
            <ShellContent>
                <ContentPage Title="Страница 1" >
                    <ContentPage.Content>
                        <Frame Margin="10">
                            <StackLayout>
                                <Label Text="Страница 1" FontSize
="Large"></Label>
                                <Label Text="Детали страницы 1"></Label>
                            </StackLayout>
                        </Frame>
                    </ContentPage.Content>
                </ContentPage>
            </ShellContent>
        </Tab>
    </FlyoutItem>
    <FlyoutItem Title="Выбор значения">
        <Tab>
            <ShellContent>
                <ContentPage Title="Страница 2" >
                    <ContentPage.Content>
                        <Frame Margin="10">
                            <StackLayout Orientation="Vertical">
                                <Label Text="Выберите один из элементов
списка"></Label>
                                <Label x:Name="Rez"></Label>
                                <RadioButton Content="Элемент 1"
GroupName="group1" ></RadioButton>
                                <RadioButton Content="Элемент 2"
GroupName="group1" ></RadioButton>
                                <RadioButton Content="Элемент 3"
GroupName="group1" ></RadioButton>
                                <RadioButton Content="Элемент 4"
GroupName="group1" ></RadioButton>
                            </StackLayout>
                        </Frame>
                    </ContentPage.Content>
                </ContentPage>
            </ShellContent>
        </Tab>
    </FlyoutItem>
    <FlyoutItem Title="Фотография">
        <Tab>
            <ContentPage Title="Страница 3" >
```

```
                <Frame Margin="10">
                    <StackLayout>
                        <Label Text="Фотография"
FontSize="Large"></Label>
                        <Frame BorderColor="Blue" >
                            <Image Aspect="AspectFit"
Source="_MG_3539.jpg">
                            </Image>
                        </Frame>
                    </StackLayout>
                </Frame>
            </ContentPage>
        </Tab>
    </FlyoutItem>
</Shell>
```

Результат работы программы представлен на рис. 91.

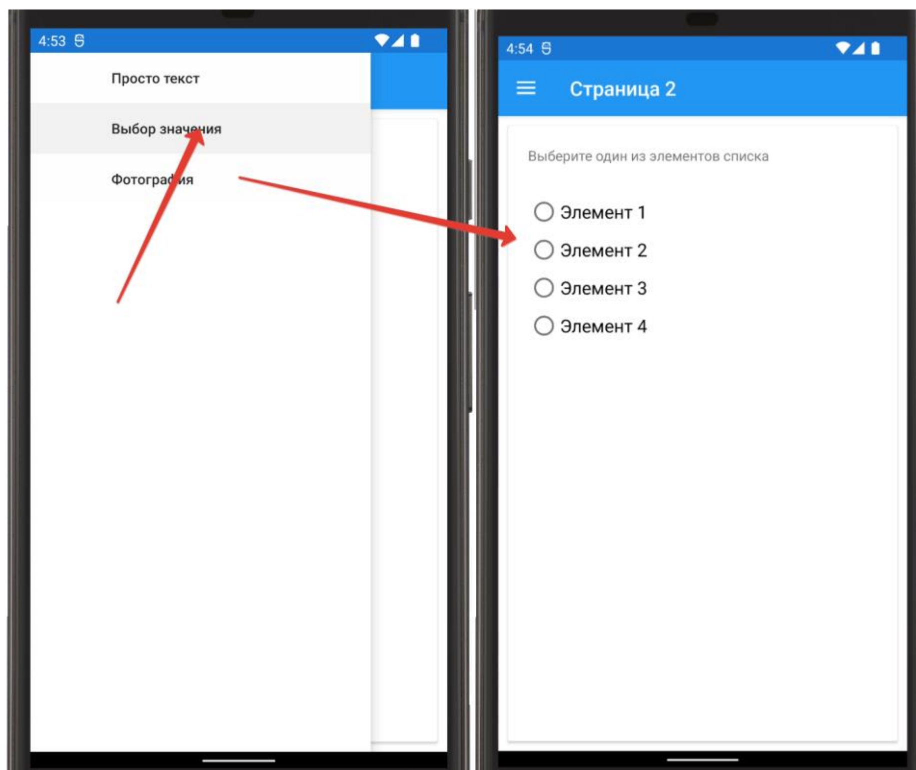

Рис. 91. Элемент FlyoutItem

Заголовок всплывающего окна можно задать с помощью свойства *Shell.FlyoutHeader*:

```
...
<Shell.FlyoutHeader>
        <Frame BackgroundColor="LightBlue">
            <Label Text="Главное меню" TextColor="White"
FontSize="Large"></Label>
```

```
        </Frame>
</Shell.FlyoutHeader>
...
```

Результат представлен на рис. 92.

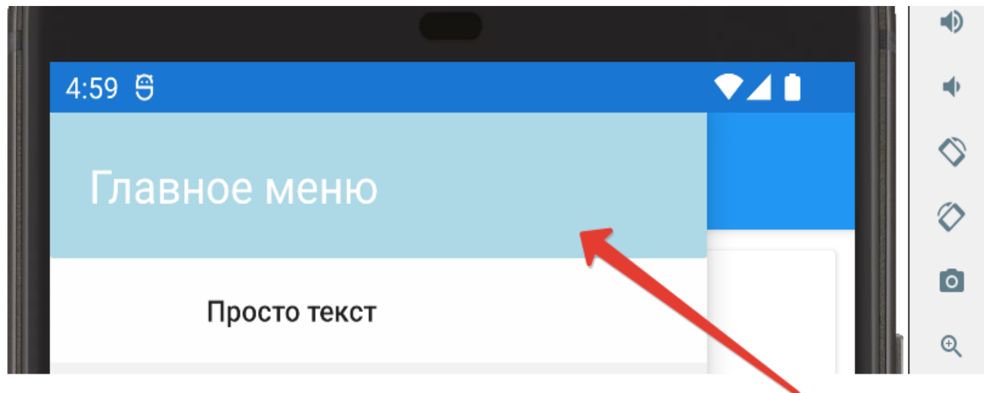

Рис. 92. Заголовок всплывающего меню

По умолчанию положение заголовка всплывающего меню будет зафиксировано, а содержимое меню будет «прокручиваться» в том случае, если элементов много. Это поведение можно изменить, задав свойство *Shell.FlyoutHeaderBehavior*:

– *Default* означает, что для полос прокрутки будет использоваться поведение, установленное для платформы по умолчанию;

– *Fixed* означает, что заголовок всплывающего меню все время остается видимым и не изменяется;

– *Scroll* указывает, что заголовок всплывающего меню пропадает с экрана, прокручиваясь вместе с другими элементами;

– *CollapseOnScroll* указывает, что заголовок всплывающего меню сворачивается до заглавия во время прокрутки элементов.

Для всплывающего меню может быть установлен нижний колонтитул:

```
...
<Shell.FlyoutFooter>
        <Frame BackgroundColor="Blue"
               HeightRequest="150"  FlowDirection="LeftToRight">
            <Label  TextColor="White"  FontSize="Medium"
                  VerticalOptions="CenterAndExpand"
                   HorizontalOptions="CenterAndExpand"
                 Text="Нижний колонтитул" >
            </Label>
        </Frame>
</Shell.FlyoutFooter>
...
```

Результат представлен на рис. 93.

Для запуска проекта под Windows добавим необходимую строку в файл App.xaml.cs в проекте для UWP (рис. 94). Результат представлен на рис. 95.

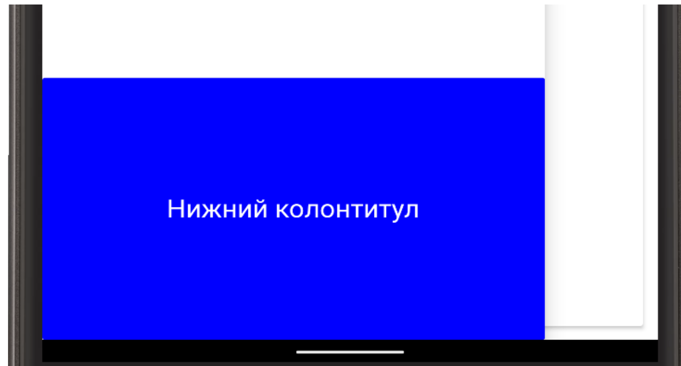

Рис. 93. Нижний колонтитул

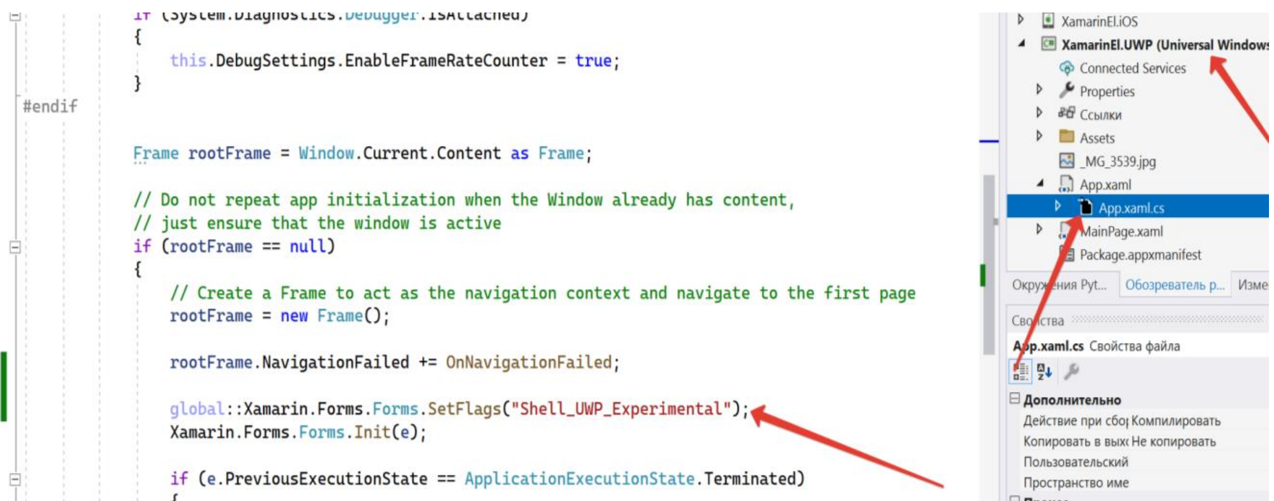

Рис. 94. «Активация» оболочки Shell под Windows

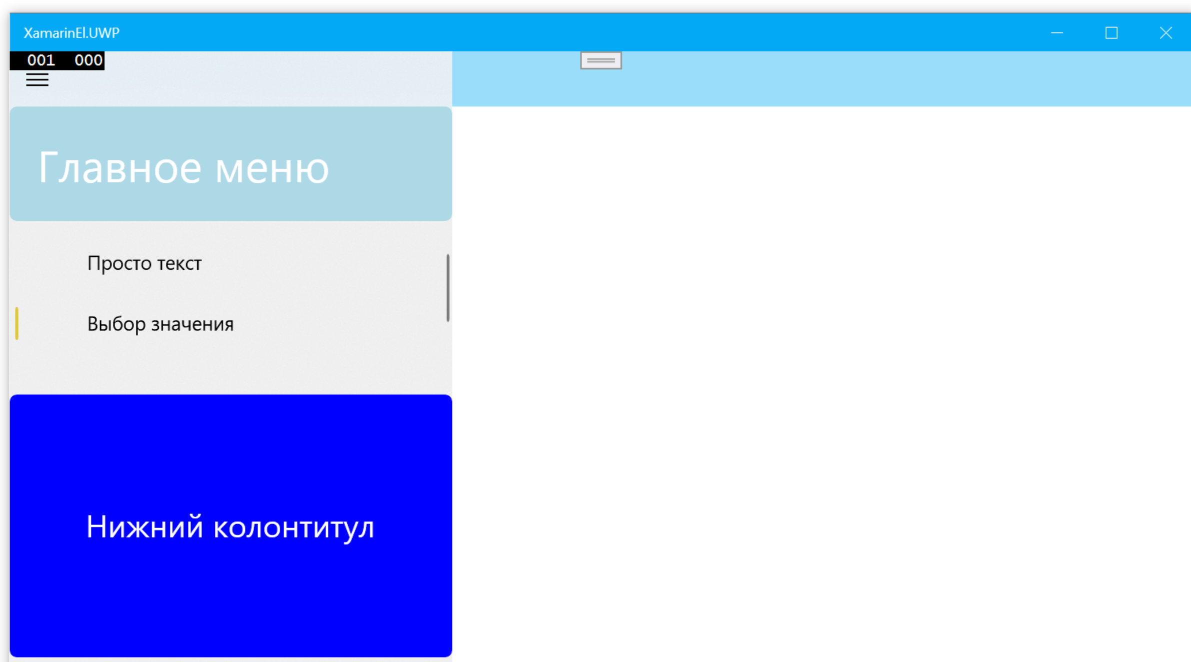

Рис. 95. Приложение с оболочкой Shell, запущенное под Windows

Для навигации с использованием вкладок в *Shell* можно использовать объект *TabBar*, который в свою очередь может содержать один или несколько объектов *Tab*, при этом каждый объект *Tab* представляет вкладку на нижней панели. Каждый объект *Tab* может содержать один или несколько объектов *ShellContent*, а каждый объект *ShellContent* отображает один объект *ContentPage*. Если *Tab* содержит более одного объекта *ShellContent*, перемещение по объектам *ContentPage* осуществляется с помощью верхней панели вкладок. Пример использования *TabBar*:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:views="clr-namespace:XamarinEl.Pages"
             x:Class="XamarinEl.Pages.ShTabPage">
    <TabBar>
        <Tab Title="Вкладка 1">
            <ShellContent>
                <ContentPage Title="Страница 1" >
                    <ContentPage.Content>
                        <Frame Margin="10">
                            <StackLayout>
                                <Label Text="Страница 1" FontSize
="Large"></Label>
                                <Label Text="Детали страницы 1"></Label>
                            </StackLayout>
                        </Frame>
                    </ContentPage.Content>
                </ContentPage>
            </ShellContent>
        </Tab>
        <Tab Title="Вкладка 2">
            <ShellContent Title="Страница 2">
                <ContentPage Title="Страница 2" >
                    <ContentPage.Content>
                        <Frame Margin="10">
                            <StackLayout Orientation="Vertical">
                                <Label Text="Выберите один из элементов
списка"></Label>
                                <Label x:Name="Rez"></Label>
                                <RadioButton Content="Элемент 1"
GroupName="group1" ></RadioButton>
                                <RadioButton Content="Элемент 2"
GroupName="group1" ></RadioButton>
                                <RadioButton Content="Элемент 3"
GroupName="group1" ></RadioButton>
                                <RadioButton Content="Элемент 4"
GroupName="group1" ></RadioButton>
                            </StackLayout>
                        </Frame>
                    </ContentPage.Content>
```

```
            </ContentPage>
        </ShellContent>
        <ShellContent Title="Страница 3">
            <ContentPage Title="Страница 3" >
                <Frame Margin="10">
                    <StackLayout>
                        <Label Text="Фотография"
FontSize="Large"></Label>
                        <Frame BorderColor="Blue" >
                            <Image Aspect="AspectFit"
Source="_MG_3539.jpg">
                            </Image>
                        </Frame>
                    </StackLayout>
                </Frame>
            </ContentPage>
        </ShellContent>
    </Tab>
  </TabBar>
</Shell>
```

Результат работы программы представлен на рис. 96.

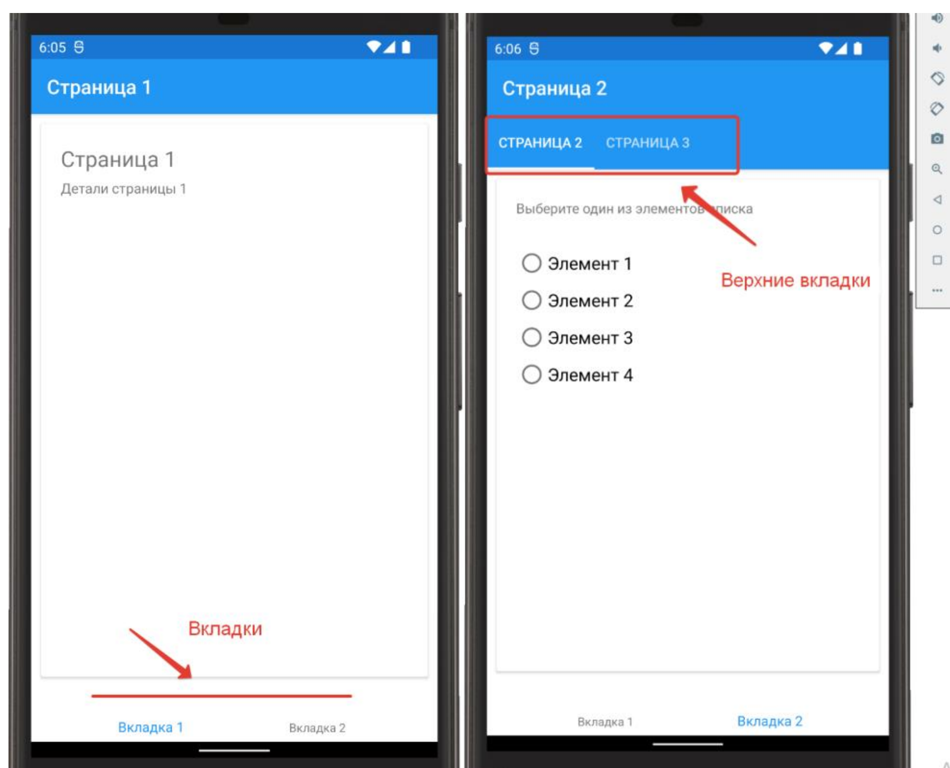

Рис. 96. Элемент TabBar

Оболочка *Shell* имеет встроенные функции поиска, предоставляемые классом *SearchHandler*. Чтобы добавить функцию поиска на страницу, необходимо реализовать класс, производный от *SearchHandler*, и переопределить методы *OnQueryChanged* и *OnItemSelected*. Метод *OnQueryChanged* срабатывает при

вводе пользователем текста в поисковое поле и принимает два аргумента – *oldValue* и *newValue*, которые содержат предыдущий и новый поисковый запрос соответственно. Метод *SelectedItem* выполняется в момент выбора пользователем результата поиска и принимает в качестве параметров выбранный пользователем объект.

Поле поиска по умолчанию является видимым и полностью развернутым. Это поведение можно изменить с помощью свойства *SearchHandler.SearchBoxVisibility*. Возможные значения:

– *Hidden* – поле поиска не отображается и недоступно;

– *Collapsible* – поле поиска является скрытым, пока пользователь не выполнит действие, открывающее его;

– *Expanded* – поле поиска является видимым и полностью развернутым.

Создадим необходимые классы (для примера возьмем список с группой студентов):

```
namespace XamarinEl.Services
{
    public class Student
    {
        public int Id { get; set; }
        public string Fio { get; set; }
        public string Group { get; set; }
    }
    public class Students : List<Student>
    {
        public Students()
        {
            Add(new Student {Id=1, Fio= "Иванов Иван Иванович",
Group="Ис-101-1" });
            Add(new Student {Id=2, Fio= "Петров Иван Иванович", Group=
"Ис-101-1" });
            Add(new Student {Id=3, Fio= "Сидоров Иван Иванович", Group=
"Ис-101-1" });
            Add(new Student {Id=4, Fio= "Иванов Петр Иванович", Group=
"Ис-101-1" });
            Add(new Student {Id=5, Fio= "Козлов Иван Иванович", Group=
"Ис-101-2" });
            Add(new Student {Id=6, Fio= "Мамонтов Иван Иванович", Group=
"Ис-101-2" });
            Add(new Student {Id=7, Fio= "Кузнецов Иван Иванович", Group=
"Ис-101-2" });
            Add(new Student {Id=8, Fio= "Буянов Иван Иванович", Group=
"Ис-101-2" });
        }
    }

    public static class ServicesStudent
    {
```

```
        public static IList<Student> Students = new Students();
    }
    public class StudentSearchHandler : SearchHandler
    {
        protected override void OnQueryChanged(string oldValue, string
newValue)
        {
            base.OnQueryChanged(oldValue, newValue);

            if (string.IsNullOrWhiteSpace(newValue))
            {
                ItemsSource = null;
            }
            else
            {
                ItemsSource = ServicesStudent.Students
                    .Where(x =>
x.Fio.ToLower().Contains(newValue.ToLower()))
                    .Select(t => t.Fio).ToList<string>();
            }
        }
        protected override async void OnItemSelected(object item)
        {
            base.OnItemSelected(item);
            var i = item as string;
            if (i is null)
                return;
            await App.Current.MainPage.DisplayAlert("Вы выбрали", i,
"Закрыть");
        }
    }
}
```

Для представления возьмем страницу с вкладками из прошлого примера и добавим на первую страницу объект *Shell.SearchHandler* (новый код выделен полужирным начертанием):

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:serv="clr-namespace:XamarinEl.Services"
             x:Class="XamarinEl.Pages.ShTabPage">
    <TabBar>
        <Tab Title="Вкладка 1">
            <ShellContent>
                <ContentPage Title="Страница 1" >
                    <Shell.SearchHandler>
                        <serv:StudentSearchHandler
                                      Placeholder="ФИО студента"
```

```
                        ShowsResults="true"
            ></serv:StudentSearchHandler>
        </Shell.SearchHandler>
        <ContentPage.Content>
...
```

Результат работы программы приведен на рис. 97 и 98.

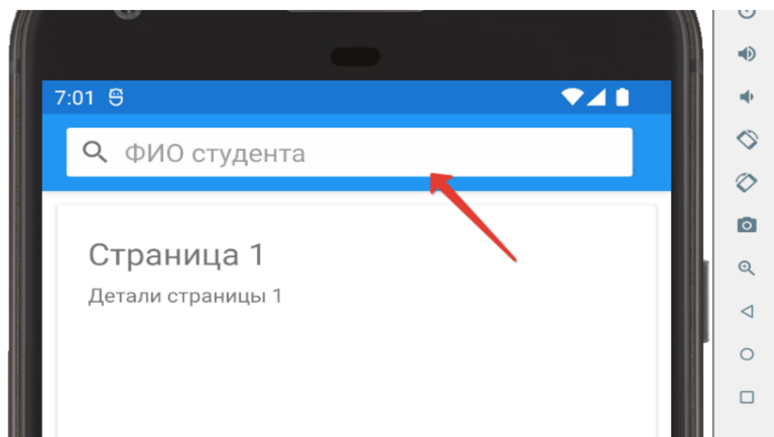

Рис. 97. Строка поиска при запуске на платформе Android

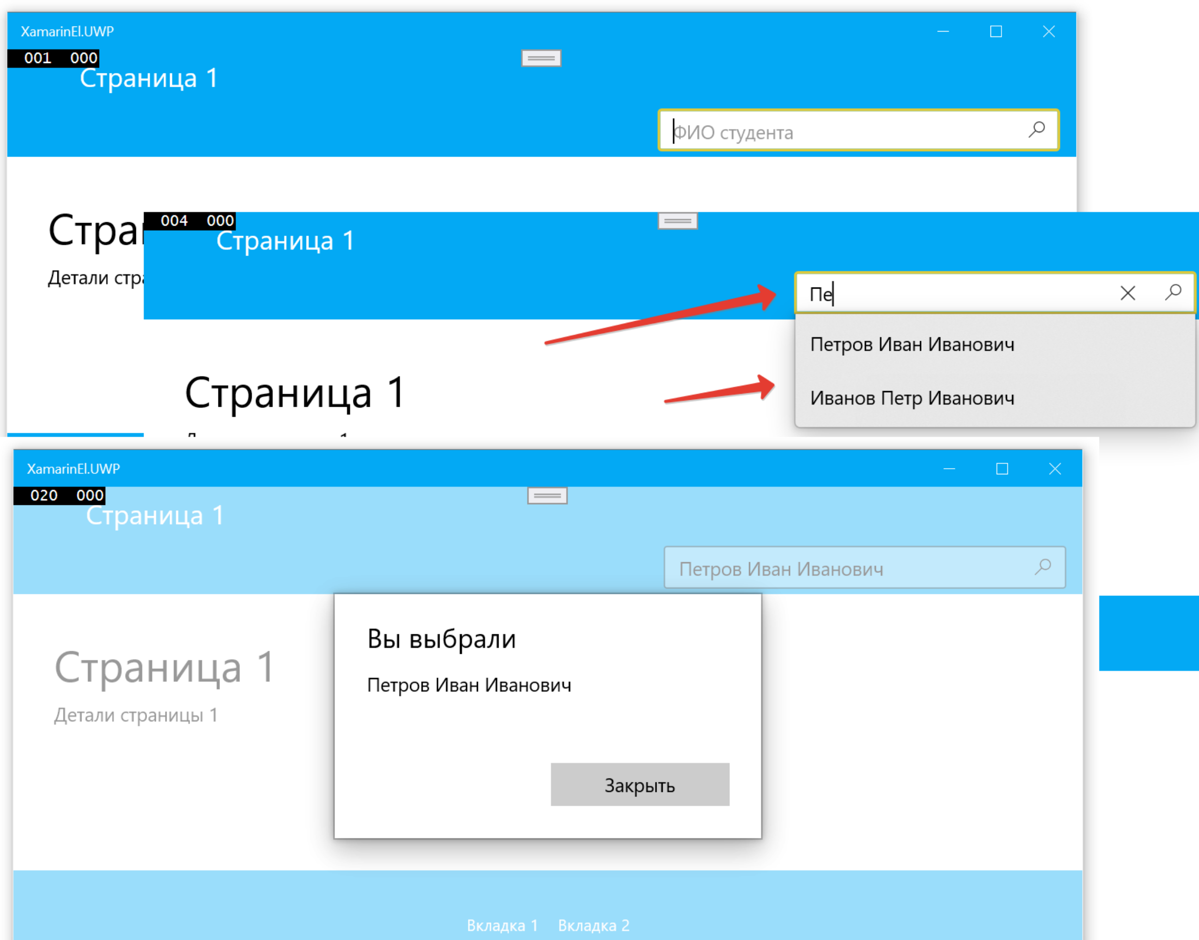

Рис. 98. Пример работы поиска на платформе Windows

Также оболочка *Shell* реализует навигацию на основе *URI*[58]. URI-навигации могут иметь три компонента:

1. Маршрут, определяющий путь к содержимому, которое существует в визуальной иерархии оболочки.

2. Страница. Страницы не отражены в визуальной иерархии оболочки и могут добавляться в стек навигации из любого места в приложении оболочки.

3. Один или несколько параметров запроса. Параметры запроса могут передаваться странице назначения при переходе на нее.

Структура URI:

```
//маршрут/страница?параметры_запроса
```

Маршруты могут быть явно определены во *FlyoutItem*, *ShellContent*, *TabBar* и *Tab* посредством свойства *Route*. Кроме того, в конструкторе страницы-оболочки *Shell* или в любой другой части кода, которая выполняется перед вызовом маршрута, можно зарегистрировать маршруты для любых страниц сведений, не представленных в визуальной иерархии оболочки. Это выполняется с помощью метода *Routing.RegisterRoute*.

Класс *Shell* определяет следующие свойства, связанные с навигацией:

– *BackButtonBehavior* – присоединенное свойство, которое определяет поведение кнопки «назад»;

– *CurrentItem* – выбранный элемент;

– *CurrentPage* – текущая страница;

– *CurrentState* – текущее состояние навигации для *Shell*;

– *Current* является приведенным по типу псевдонимом для *Application.Current.MainPage*.

Навигация выполняется путем вызова метода GoToAsync класса *Shell*. Когда планируется действие навигации, срабатывает событие *Navigating*, а после завершения навигации – событие *Navigated*. Для выполнения обратной навигации можно указать «..» в качестве аргумента для метода GoToAsync. Также при выполнении программной навигации на основе URI можно передавать данные в параметрах запроса. Продемонстрируем это на примере списка студентов.

Создадим приложение, состоящее из трех страниц: первая страница – оболочка *Shell*, вторая страница – список студентов, третья – детальная информация о выбранном студенте (функции редактирования записи реализовывать не будем, хотя они подразумеваются).

[58] Унифицированный (единообразный) идентификатор ресурса. Схемы ресурсов можно посмотреть на официальном сайте IANA. URL: https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml.

За основу возьмем код ServicesStudent из прошлого примера, слегка его модифицировав (приведем только измененные классы, изменения выделим полужирным начертанием):

```
    public class Student
    {
        public int Id { get; set; }
        public string Fio { get; set; }
        public string Group { get; set; }
        public override string ToString()
        {
            return $"{Id}. {Fio} {Group}";
        }
    }
public static class ServicesStudent
    {
        public static IList<Student> Students = new Students();
        public static Student GetStudent(int id)
        {
            return Students.FirstOrDefault(t => t.Id == id);
        }
    }

public class StudentSearchHandler : SearchHandler
    {
        protected override void OnQueryChanged(string oldValue, string
newValue)
        {
            base.OnQueryChanged(oldValue, newValue);

            if (string.IsNullOrWhiteSpace(newValue))
            {
                ItemsSource = null;
            }
            else
            {
                ItemsSource = ServicesStudent.Students
                    .Where(x =>
x.Fio.ToLower().Contains(newValue.ToLower()))
                    .ToList<Student>();
            }
        }
        protected override async void OnItemSelected(object item)
        {
            base.OnItemSelected(item);
            var i = item as Student;
            if (i is null)
                return;
            await
Shell.Current.GoToAsync($"///Main/StudDetail?id={i.Id}");
```

```
        }
    }
```

Отметим, что в классе *StudentSearchHandler* мы изменили метод *OnItemSelected*, теперь при выборе записи из списка поиска будет идти переход на страницу *StudDetail*.

Разметка страницы *Shell*:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:pages="clr-namespace:XamarinEl.Pages"
             x:Class="XamarinEl.Pages.ShMaster">
    <TabBar>
        <Tab>
            <ShellContent Route="Main" >
                <pages:ShList></pages:ShList>
            </ShellContent>
        </Tab>
    </TabBar>
</Shell>
```

На странице оболочки *Shell* добавим один *TabBar* (в этом случае вкладки внизу не появятся), в котором разместим страницу со списком.

Код страницы:

```
public partial class ShMaster : Shell
    {
        public ShMaster()
        {
            InitializeComponent();
            Routing.RegisterRoute("Main/StudDetail", typeof(ShDetail));

        }
        protected override async void
OnNavigating(ShellNavigatingEventArgs args)
        {
            base.OnNavigating(args);
            if
(args.Target.Location.OriginalString.Contains("StudDetail"))
            {
                ShellNavigatingDeferral token = args.GetDeferral();
                var result = await DisplayActionSheet("Редактировать
запись?", "Yes", "No");
                if (result != "Yes")
                {
                    args.Cancel();
                }
                token.Complete();
```

```
        }
    }
}
```

В конструкторе страницы в таблицу маршрутизации добавим страницу деталей. Продемонстрируем также работу метода *OnNavigating*, который будет срабатывать, когда генерируется событие *Navigating*. Использование этого метода позволяет перехватывать навигацию и переопределять ее. Например, можно отменить переход назад, если на странице есть несохраненные данные, или реализовать отложенный переход (возможность завершить или отменить переход в зависимости от пользовательского выбора), что и было сделано в нашем примере.

Метод *GetDeferral* для объекта *ShellNavigatingEventArgs* возвращает токен *ShellNavigatingDeferral* с методом *Complete*, который можно использовать для завершения запроса о переходе. Переход отменяется вызовом метода *Cancel* для объекта *ShellNavigatingEventArgs*. При этом задавать вопрос о переходе мы будем только в том случае, если идет переход на страницу *StudDetail.* Остальные переходы будут завершаться автоматически, без запроса.

Разметка страницы со списком студентов:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:serv="clr-namespace:XamarinEl.Services"
             x:Class="XamarinEl.Pages.ShList">
    <Shell.SearchHandler>
        <serv:StudentSearchHandler
              Placeholder="ФИО студента"
              ShowsResults="true"
              ></serv:StudentSearchHandler>
    </Shell.SearchHandler>

    <ContentPage.Content>
        <ListView x:Name="Lstud" ItemSelected="Lstud_ItemSelected">
        </ListView>
    </ContentPage.Content>
</ContentPage>
```

На этой странице мы разместили поле для поиска студента по его ФИО, а также сам список студентов.

Код страницы:

```
public partial class ShList : ContentPage
    {
        public ShList()
        {
            InitializeComponent();
            Lstud.ItemsSource = ServicesStudent.Students;
```

```
        }

        private async void Lstud_ItemSelected(object sender,
SelectedItemChangedEventArgs e)
        {
            var id = int.Parse(e.SelectedItem.ToString().Split('.')[0]);
            await Shell.Current.GoToAsync($"StudDetail?id={id}") ;
        }
    }
```

На странице устанавливаем для элемента *ListView* источник данных и реализуем обработчик события нажатия на элемент списка.

Разметка страницы деталей:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinEl.Pages.ShDetail"
             Title="Редактировать">
    <ContentPage.Content>
        <StackLayout>
            <TableView>
                <TableView.Root>
                    <TableRoot>
                        <TableSection>
                            <EntryCell x:Name="eid"
Label="Идентификатор"></EntryCell>
                            <EntryCell x:Name="efio"
Label="ФИО"></EntryCell>
                            <EntryCell x:Name="egroup"
Label="Группа"></EntryCell>
                        </TableSection>
                    </TableRoot>
                </TableView.Root>
            </TableView>
            <Button Text="Сохранить"></Button>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Код страницы:

```
[QueryProperty(nameof(IdStudent), "id")]
    public partial class ShDetail : ContentPage
    {
        public int IdStudent
        {
            set
            {
                LoadStudent(value);
            }
```

```
        }
        public ShDetail()
        {
            InitializeComponent();
        }
        public void LoadStudent(int id)
        {
            var s = ServicesStudent.GetStudent(id);
            eid.Text = s.Id.ToString();
            efio.Text = s.Fio;
            egroup.Text = s.Group;
        }
    }
```

Для получения данных навигации «принимающий» класс дополняем атрибутом *QueryPropertyAttribute*. Таких атрибутов должно быть столько же, сколько передаваемых параметров. Первый аргумент задает имя свойства класса принимающей страницы, которое будет получать данные, а второй аргумент задает идентификатор параметра запроса. Таким образом, атрибут *QueryPropertyAttribute* в примере выше указывает, что свойство *IdStudent* будет получать данные из параметра запроса **id**, переданного в URI, через вызов метода *GoToAsync*. Метод *Set* свойств *IdStudent* вызывает метод *LoadStudent* для получения объекта *Student.*

Пример работы приложения приведен на рис. 99 и 100.

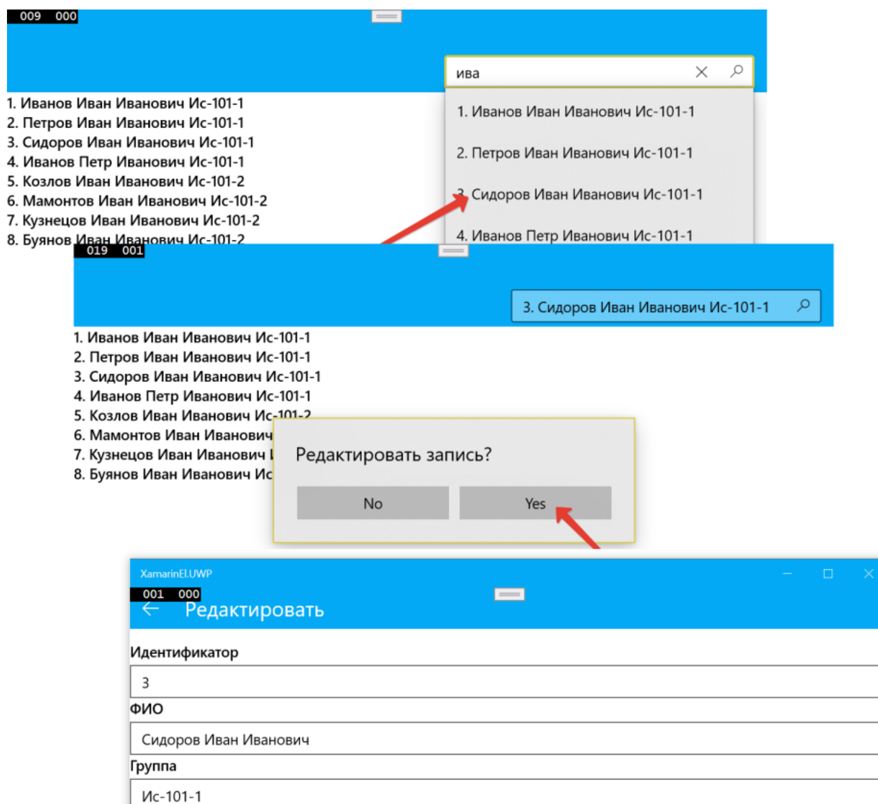

Рис. 99. Работа приложения под Windows, переход на редактирование записи от списка поиска

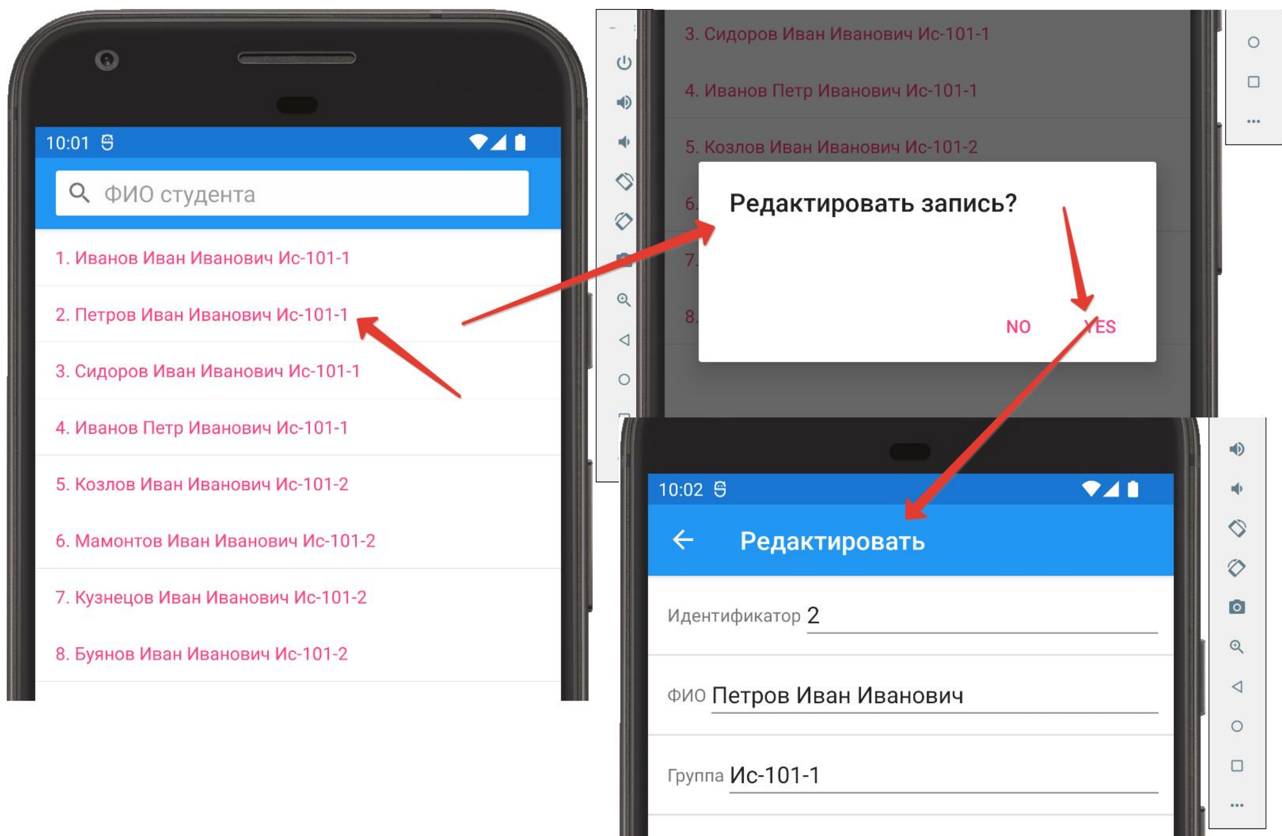

Рис. 100. Работа приложения под Android, переход на редактирование записи от основного списка

Кроме того, есть другой вариант получения данных: реализация в принимающем классе интерфейса *IQueryAttributable*. Интерфейс *IQueryAttributable* указывает, что реализующий класс должен реализовать метод *ApplyQueryAttributes*. У этого метода есть аргумент *query* типа *IDictionary<string, string>*, который содержит любые данные, передаваемые в процессе навигации. Каждый ключ в словаре – это идентификатор параметра запроса, а его значение – значение параметра запроса. Преимущество использования этого подхода заключается в том, что данные навигации можно обрабатывать с помощью одного метода.

Контрольные вопросы

1. Классификация элементов управления.
2. Иерархия элементов управления.
3. Основные типы страниц, их поведение и назначение.
4. Основные типы макетов (элементов компоновки), их поведение и назначение.
5. Основные типы представлений (элементов управления), их поведение и назначение.
6. Элементы для редактирования текста.
7. Элементы для установки значений.
8. Элементы для обозначения действий.

9. Элементы для отображения коллекций.
10. Основные типы ячеек, их поведение и назначение.
11. Основы навигации между страниц Xamarin.Forms.
12. Оболочка Shell, основные элементы.
13. Модальная страница Xamarin.Forms. Ее создание, поведение, навигация.
14. Навигация с использованием оболочки Shell.
15. Передача информации между страницами, основные способы.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

Байдачный С.С. Silverlight 4: Создание насыщенных Web-приложений / С.С. Байдачный. – Москва : Солон-Пресс, 2010. – 278 с.

Ватсон Б. C# 4.0 на примерах / Б. Ватсон. – Санкт-Петербург : БХВ-Петербург, 2011. – 608 с.

Прайс М.Дж. C# 7 и .NET Core. Кроссплатформенная разработка для профессионалов / М.Дж. Прайс : пер. с англ. М. Сагалович, С. Черников. – 3-е изд. – Санкт-Петербург : Питер, 2018. – 636 с.

Пугачев С.В. Разработка приложений для Windows 8 на языке C# / С.В. Пугачев, А.М. Шериев, К.А. Кичинский. – Санкт-Петербург : БХВ-Петербург, 2013. – 416 с.

Троелсен Э. Язык программирования C# 7 и платформы .NET и .NET Core / Э. Троелсен, Ф. Джепикс. – 8-е изд. – Москва : Санкт-Петербург : Диалектика, 2019. – 1328 с.

Черников В.Н. Разработка мобильных приложений на C# для iOS и Android / В.Н. Черников. – Москва : ДМК Пресс, 2020. – 188 с.

Учебное издание

Родионов Алексей Владимирович

КРОССПЛАТФОРМЕННЫЕ ИНСТРУМЕНТАЛЬНЫЕ СИСТЕМЫ: РАЗРАБОТКА ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ XAMARIN.FORMS (ЧАСТЬ 1)

Учебное пособие

В трех частях

Часть 1

Подготовлено к печати Т.И. Кочульской

Дизайн обложки А.А. Мартыновой

ИД № 06318 от 26.11.01.
Подписано в печать 19.05.23. Формат 60×90 1/16. Бумага офсетная. Печать цифровая. Усл. печ. л. 9,0. Тираж 300 экз. (1-й з-д 1–30). Заказ .

Издательский дом ФГБОУ ВО «БГУ».
Отпечатано в ИПО ФГБОУ ВО «БГУ».
664003, г. Иркутск, ул. Ленина, 11.
http://bgu.ru.