Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ ДЛЯ МОБИЛЬНЫХ СИСТЕМ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 1-39 03 02 «Программируемые мобильные системы»

Минск БГУИР 2019

УДК 004.42:[621.395.721.5+004.382.4](076.5)
ББК 32.972.1я73+32.971.321.4я73
Р17

А в т о р ы:

А. О. Рак, А. В. Михалькевич, А. С. Середа, Н. А. Голубов, А. П. Горбач

Р е ц е н з е н т ы:

кафедра электронных вычислительных машин и систем учреждения образования «Брестский государственный технический университет» (протокол №6 от 19.01.2018);

заведующий кафедрой автоматизированных систем управления производством учреждения образования «Белорусский государственный аграрный технический университет» кандидат технических наук, доцент А. Г. Сеньков

Разработка web-приложений для мобильных систем. Лабораторный
Р17 практикум : пособие / А. О. Рак [и др.]. – Минск : БГУИР, 2019. – 130 с. : ил.
ISBN 978-985-543-447-5.

Приведены принципы разработки и тестирования *web*-приложений для мобильных устройств на примере объектно-ориентированного языка *PHP* и системы *Android*.

УДК 004.42:[621.395.721.5+004.382.4](076.5)
ББК 32.972.1я73+32.971.321.4я73

ISBN 978-985-543-447-5

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Стремительное развитие технологий и мобильных телефонов за последние двадцать лет привело к заслуженному массовому признанию смартфонов.

Смартфоны завоевали пользователей своей многофункциональностью. Камера, телефон, плеер и другие дополнения в одном устройстве – все это справедливо пользуется высоким спросом.

Но особую популярность смартфоны обрели благодаря приложениям, которые на них можно устанавливать. Современные *web*-приложения базируются на различных операционных системах и программных платформах. Таким образом, выбор при разработке сводится к созданию нативного приложения, гибридного приложения или *web*-приложения.

Основная цель лабораторных работ, выполняемых на занятиях по дисциплине «Разработка *web*-приложений для мобильных систем», – изучение принципов разработки *web*-систем с использованием современных языков программирования для различных операционных систем.

Система, разрабатываемая с помощью этих средств, строится по трехзвенной клиент-серверной архитектуре, основные принципы которой рассматриваются в рамках лекционных занятий.

В качестве сервера приложений (*http*-сервера), сервера баз данных, который необходим при выполнении последних лабораторных работ, используются локальный *web*-сервер и среда разработки *OpenServer*.

Для выполнения поставленных задач изучаются особенности работы с различными программными платформами, фреймворками, определяющими структуру программной системы.

Логическим завершением пособия является изучение системы контроля версий и работы с удаленным репозиторием, включающее в себя инструкцию и систему распознавания ошибок.

Лабораторная работа №1. Введение в разработку *web*-приложений для мобильных устройств

Цель: ознакомиться с основными понятиями и особенностями разработки *web*-приложений.

1.1 Введение в разработку *web*-приложений

Сегодня смартфоны – основное карманное устройство более чем для миллиарда людей, поэтому мобильные *web*-приложения стали необходимостью как с технической, так и с коммерческой точки зрения. Есть несколько подходов к разработке таких приложений, и, учитывая, что нынче лидирующие компании могут за считанные месяцы уйти на второй план, а новые гаджеты появляются практически беспрерывно, современному специалисту важно разбираться в основных технологиях создания мобильных *web*-приложений [1].

В настоящее время новое приложение во многих случаях будет *web*-приложением, тогда как в прошлом обычно начинали с создания программы специально для какой-либо операционной системы, например, *Windows* или *Unix*. Сейчас же актуальна задача создания приложений, способных работать на всех мобильных устройствах, поэтому очень непросто принять решение о выборе соответствующего инструментария хотя бы по причине растущего числа платформ и инструментальных сред [1].

Мобильное приложение сильно отличается от *desktop*-приложения, поэтому необходимо учитывать данный факт при планировании процесса разработки [2].

Основными различиями мобильных приложений и desktop-приложений являются [2]:

1) мобильное устройство – это система, которая не обладает мощной начинкой, таким образом, оно не может работать как персональный компьютер;

2) тестирование мобильных приложений проводится на мобильных телефонах (*Apple, Samsung, Nokia*), в то время как *desktop*-приложения тестируются на центральном процессоре;

3) у мобильных устройств бывают разные разрешения. Размер экрана мобильного телефона меньше, чем у *desktop*;

4) выполнение и прием вызовов является основной задачей телефона, поэтому приложение не должно вмешиваться в эту важную функцию;

5) мобильные приложения имеют широкий спектр конкретных операционных систем и компонентных конфигураций (*Android, iOS, BlackBerry*);

6) операционная система мобильного телефона быстро устаревает;

7) мобильные устройства используют сетевые подключения (3*G*, 4*G*, *Wi-Fi*), широкополосное подключение к настольному персональному компьютеру (ПК) или *Wi-Fi*;

8) мобильные устройства постоянно осуществляют поиск сети, поэтому есть необходимость протестировать приложение с разной скоростью передачи данных;

9) инструменты, которые хорошо подходят для тестирования *desktop*-приложений, не полностью подходят для тестирования мобильных приложений;

10) мобильные приложения должны поддерживать несколько входных каналов (клавиатура, голос, жесты и т. д.), мультимедийные технологии и другие функции, повышающие их удобство использования.

Также при разработке обычных сайтов нередко приходится сталкиваться с тем, что те или иные вещи в разных браузерах отображаются различно: где-то поддерживается такой стандарт, а где-то нет. С этой проблемой придется столкнуться и при разработке мобильных сайтов.

1.2 Особенности *web*-программирования под мобильные системы

Понятие мобильного *web*-приложения можно трактовать по-разному. Согласно одному толкованию это приложение, работающее в *web* и спроектированное так, чтобы корректно отображаться на мобильном устройстве, а согласно другому – это приложение, созданное специально для конкретной мобильной операционной системы (ОС), которое соединяется с *web* для отправки и приема данных [1]. Чтобы разграничить эти позиции, можно воспользоваться «шкалой» от стандартных до нативных *web*-приложений. Если нативные работают со скоростью аппаратной платформы, то гибридные и мобильные исполняются поверх дополнительных уровней, которые расходуют вычислительные ресурсы, снижая быстродействие устройства.

Введем понятия стандартных и быстро реагирующих приложений, а также «мобильного *web*». Соответствующие технологии, а также их преимущества и недостатки весьма схожи, однако здесь перечислим их по отдельности, подчеркнув, что, в сущности, это разные решения. Некоторым пользователям и разработчикам нравится адаптивный интерфейс, тогда как другие отдают предпочтение специализированным приложениям для конкретных устройств [1]. Мобильно-ориентированные *web*-сайты (например, соответствующие варианты *Facebook* или *Google Docs*) обычно предлагают пользователю возможность доступа и к стандартной *web*-версии. Стандартная программа на смартфоне, вероятнее всего, будет работать медленнее, но некоторые пользователи выберут именно ее, чтобы иметь доступ ко всем возможностям сайта.

Выделим основные виды мобильных *web*-приложений:

1 *Стандартные web-приложения* [1]. Термином «*web*-приложения» обозначаются приложения, рассчитанные на исполнение в браузерах для настольных компьютеров. Они также смогут работать на мобильных устройствах, если не полагаются на специфические технологии, отсутствующие на многих мобильных устройствах (например, *Adobe Flash*).

2 *Адаптивный web-интерфейс* [1]. Приложения с адаптивным *web*-интерфейсом автоматически меняют его внешний вид в зависимости от размера устройства – чаще всего при этом используется технология *CSS* (*Cascading style sheets*). Дизайн может быть выбран сервером при доставке приложения, изменен

на уровне клиента либо обоими способами. Смысл в том, чтобы контент из одного и того же источника отображался по-разному в зависимости от особенностей конкретного устройства. Данный вариант применим как для мобильных *web*-приложений, так и для исполняемых на других видах устройств, например, на игровых консолях и телевизорах.

3 *Мобильный web* [1]. Термин «мобильный *web*» используется для случаев, когда есть специальный сайт или онлайн-сервис для доставки контента на мобильные устройства. Обычно интерфейс при этом выглядит эффектнее и работает быстрее, чем адаптивный, так как элементы (клавиши, селекторы и текстовые поля) отображаются аналогично нативным. При данном подходе может понадобиться создать несколько вариантов одного и того же сайта.

4 *Нативные приложения* [1]. Компании, разрабатывающие мобильные операционные системы, стремятся к появлению как можно большего числа нативных приложений для их ОС, и такие приложения разрабатываются с помощью инструментария, рекомендованного соответствующим поставщиком: например, на *Objective-C* и *Xcode* – для *iOS* или на *Java* и *Eclipse* – для *Android*. Для каждой ОС обычно нужен отдельный проект нативного приложения, а для этого требуется больше разработчиков. При сугубо нативном подходе приходится также иметь в виду, что в дополнение к существующим мобильным платформам постоянно появляются все новые.

5 *Гибридные приложения* [1]. Гибридным является мобильное приложение, «упакованное» в нативную оболочку. Такие приложения, как и нативные, устанавливаются из онлайн-супермаркета и имеют доступ к тем же возможностям устройства, но разрабатываются они с помощью *HTML*5, *CSS* и *JavaScript*.

Выбор вида мобильного приложения для каждой ситуации не всегда очевиден.

Сузить круг вариантов выбора поможет рассмотрение ряда технических факторов:

1 *Поддержка платформ и версий* [1]. Прежде всего следует определиться, сколько платформ и их версий предстоит поддерживать. При этом нужно принять во внимание диапазон устройств, различия в наборах средств разработки для каждого из них, а также возможности браузера на каждой платформе и имеющиеся навыки. Если цель состоит в создании приложения с поддержкой сразу множества платформ, то гибридное или мобильное *web*-приложение подойдет больше, чем нативное, которое придется разрабатывать отдельно для каждой из платформ.

2 *Возможности устройства* [1]. Необходимо учесть, какие вам понадобятся функции устройства. Если приложению нужен доступ к камере, сканеру штрихкодов, файловой системе или периферийному *Bluetooth*-устройству, то лучше воспользоваться нативным или гибридным подходом. Новейшие браузеры поддерживают аппаратное ускорение векторной графики, но пока еще не могут в полной мере пользоваться собственными графическими возможностями устройства и другими функциями, доступными через нативный *API*.

3 *Пользовательский интерфейс* [1]. У нативных приложений более функционально богатые и привлекательные пользовательские интерфейсы, быстрее реагирующие и более интерактивные, а у *web*-приложений могут быть сложности с доступом к аппаратным функциям устройства. Гибридные приложения дают разработчику больше свободы, позволяя коду *HTML* обращаться к нативным *API*, но необходимость учета применения *web*-технологий делает пользовательский интерфейс внешне не совсем соответствующим «родному».

4 *Быстродействие* [1]. Быстродействие – один из главных факторов, который нужно учитывать разработчику. Если пользовательский интерфейс богат графикой или нужна интенсивная обработка данных, то при мобильном и гибридном подходах достичь высокого быстродействия труднее, поскольку в этих случаях приложения работают поверх дополнительных уровней, расходующих ресурсы процессора. В любом случае перед началом разработки следует проверить быстродействие на прототипе или похожих уже существующих приложениях.

5 *Обновления* [1]. При разработке нативных приложений нужно учитывать, что пользователи не всегда хотят переходить на новую версию, так что, скорее всего, придется одновременно поддерживать несколько версий, что усложнит серверную часть. То же относится к гибридным приложениям, если значительная часть кода исполняется локально.

Определиться с оптимальным выбором вида мобильного приложения также поможет рассмотрение ряда нетехнических факторов:

1 *Распространение* [1]. Мобильные приложения просто распространять, но трудно рекламировать, так как вне онлайн-супермаркета их найти нелегко. Поэтому если нужно привлечь внимание пользователей, то предпочтительнее нативное или гибридное приложение. Когда вы ориентируетесь на потребителей или геймеров, можно сэкономить на маркетинге, если распространять приложение через онлайн-магазин соответствующей платформы. Тем не менее, с ростом объема приложений в подобных магазинах добиваться известности все сложнее. Если же приложение рассчитано на узкую аудиторию пользователей (например, служащих предприятия), то можно воспользоваться его частным магазином приложений. При этом следует учесть, что у общедоступных магазинов приложений есть свои ограничения, – скажем, нет возможности влиять на политику управления магазином.

2 *Цикл согласования* [1]. Процесс создания мобильных приложений может конфликтовать с методологиями скорой (*agile*) разработки, подразумевающими частый выпуск обновлений и непрерывную обратную связь с пользователями. При выборе нативного или гибридного подхода нужно иметь в виду, что частью проекта разработки станет процесс согласования, а если отклонений не будет, то время на согласование обычно минимально. Ситуация также облегчается, если вы разрабатываете приложение только для какого-то конкретного смартфона или пользуетесь локальным магазином приложений предприятия.

3 *Монетизация* [1]. С помощью платформенных магазинов приложений можно не только распространять приложения, но и добиваться окупаемости благодаря простым в использовании и надежным платежным шлюзам. Но в этой ситуации значительную долю дохода получает владелец платформы (в случае *iOS* – около 30 %), поэтому нужно тщательно взвесить, стоит ли выбирать платформенный подход.

Обобщим все перечисленные факторы, влияющие на выбор вида мобильного приложения, в таблице 1.1 [1].

Таблица 1.1 – Критерии выбора между подходами

Факторы	Нативный	Гибридный	*Web*
Цена поддержки различных платформ	Высокая	Средняя	Низкая
Доступ к возможностям устройства	Полный	Полный	Частичный
Пользовательский интерфейс	Полноценный	Полноценный	С ограничениями
Быстродействие	Очень высокое	Очень высокое	Высокое
Обновление версий клиентского приложения	Требуется	Требуется	Не требуется
Простота публикации / распространения	Средняя	Средняя	Высокая
Цикл согласования	Обязательный	Требуется в некоторых случаях	Не требуется
Монетизация через супермаркет приложений	Доступна	Доступна	Недоступна

Все виды мобильных приложений можно создавать без помощи фреймворков, однако использование готовых библиотек и инструментов упрощает процесс разработки и уменьшает трудозатраты. Фреймворки имеются практически для всех типов мобильных приложений, и большинство из них служат для разработки на *HTML*. Но можно выбирать гибридный или нативный подход и при этом создавать кроссплатформенные приложения, т. е. библиотеки и для этих случаев.

Сегодня рынок мобильных устройств – это фактически монополия платформ *iOS* и *Android*. Также представлена незначительная доля платформ *Blackberry, Windows Phone* и *Symbian*, но модели устройств уходят с рынка быстро, а аппаратные и программные технологии активно развиваются, стремительно изменяя ситуацию на рынке. Например, платформу *Firefox OS* поддерживают крупные операторы сетей мобильной связи, а поскольку приложения для нее основаны на *HTML*, то их ассортимент может вырасти достаточно быстро.

При создании нативных и гибридных приложений всегда нужно учитывать риск, связанный с отсутствием у разработчика контроля над платформой. Например, можно потерять пользователей, если владелец платформы из опасения

утраты доходов или по другой причине ограничит доступ к вашему приложению. Возможны также проблемы, если владельцы платформ будут запрещать любые приложения, разработанные на определенном фреймворке, по экономическим соображениям или из-за проблем с безопасностью.

1.3 Обзор объектно-ориентированных языков и технологий

Объектно-ориентированные языки программирования пользуются в последнее время большой популярностью среди программистов, так как они позволяют использовать преимущества объектно-ориентированного подхода не только на этапах проектирования и конструирования программных систем, но и на этапах их реализации, тестирования и сопровождения.

Первый объектно-ориентированный язык программирования ***Simula***-67 был разработан в конце 60-х гг. XX века в Норвегии [3]. Авторы этого языка очень точно почувствовали перспективы развития программирования: их язык намного опередил свое время. Однако их современники (программисты 60-х гг.) оказались не готовы воспринять ценности языка *Simula*-67, и он не выдержал конкуренции с другими языками программирования (прежде всего с языком *Fortran*). Прохладному отношению к языку *Simula*-67 способствовало и то обстоятельство, что он был реализован как интерпретируемый (а не компилируемый) язык, что было совершенно неприемлемым в 60-е гг., так как интерпретация связана со снижением эффективности (скорости выполнения) программ.

Язык ***Smalltalk*** был разработан командой *Xerox Palo Alto Research Center Learning Research Group* как программная часть *Dynabook* – фантастического проекта Алана Кея [3]. В основу были положены идеи *Simula*. *Smalltalk* является одновременно и языком программирования, и средой разработки программ. Это чисто объектно-ориентированный язык, в котором абсолютно все рассматривается как объекты; даже целые числа – это классы. *Smalltalk* является важнейшим объектно-ориентированным языком, следующим после *Simula*, поскольку он не только оказал влияние на последующие поколения языков программирования, но и заложил основы современного графического интерфейса пользователя, на которых непосредственно базируются интерфейсы *Macintosh*, *Windows* и *Motif*.

Известны пять выпусков языка *Smalltalk*, обозначаемых по году их появления: *Smalltalk*-72, -74, -76, -78, -80 [3]. Реализации 1972 и 1974 гг. заложили основу языка, в частности, идею передачи сообщений и полиморфизм, хотя в то время механизм наследования еще не появился. В последующих версиях полноправное гражданство получили классы; к этому привела точка зрения, что все состоит из объектов. *Smalltalk*-80 был перенесен на многие компьютерные платформы.

В основу языка положены две простые идеи:

- все является объектами;
- объекты взаимодействуют, обмениваясь сообщениями.

Появление ***Delphi*** не прошло незамеченным среди многочисленных пользователей компьютера [3]. Оценки экспертов, изучающих возможности этого но-

вого продукта фирмы *Borland,* обычно окрашены в восторженные тона. Основное достоинство *Delphi* состоит в том, что здесь реализованы идеи визуального программирования. Среда визуального программирования превращает процесс создания программы в приятное и легко доступное в плане понимания конструирование приложения из большого набора графических и структурных примитивов.

Система *Delphi* позволяет решать множество задач, в частности [3]:

- создавать законченные приложения для *Windows* самой различной направленности: от вычислительных и логических до графических и мультимедиа;

- быстро создавать (даже начинающим программистам) профессионально оформленный оконный интерфейс для любых приложений;

- создавать мощные системы работы с локальными и удаленными базами данных;

- создавать справочные системы (файлы *.hlp*) для своих приложений и др.

Delphi – чрезвычайно быстро развивающаяся система. Первая версия – *Delphi* 1.0 была выпущена в феврале 1995 г. А затем ежегодно выпускались новые версии, каждая последующая версия *Delphi* дополняла предыдущую. Большинство версий *Delphi* выпускается в нескольких вариантах: *Standart* – стандартном, *Professional* – профессиональном, *Client/Server* – клиент/сервер, *Enterprise* – разработка баз данных предметных областей. Варианты различаются в основном разным уровнем доступа к системам управления базами данных. Последние варианты – *Client/Server* и *Enterprise* – в этом отношении наиболее мощные.

Язык программирования ***C++*** был разработан Бьерном Страуструпом, сотрудником *AT&T Bell Laboratories* [3]. Непосредственным предшественником *C++* является *C with Classes,* созданный тем же автором в 1980 г. Язык *C with Classes,* в свою очередь, был создан под сильным влиянием *C* и *Simula*. *C++* – это в значительной степени надстройка над *C*. В определенном смысле можно назвать *C++* улучшенным *C*, тем *C*, который обеспечивает контроль типов, перегрузку функций и ряд других удобств. Но главное в том, что *C++* добавляет к *C* объектную ориентированность.

Известны несколько версий *C++* [3]. В версии 1.0 реализованы основные механизмы объектно-ориентированного программирования, такие как одиночное наследование и полиморфизм, проверка типов и перегрузка функций. В созданной в 1989 г. версии 2.0 нашли отражение многие дополнительные свойства, возникшие на базе широкого опыта применения языка многочисленным сообществом пользователей. В версии 3.0 1990 г. появились шаблоны и обработка исключений. *C++* продолжает совершенствоваться и в настоящее время. Так, в 1998 г. вышла новая версия стандарта, содержащая в себе некоторые довольно существенные изменения. Язык стал основой для разработки современных больших и сложных проектов.

Язык ***Java*** зародился как часть проекта создания передового программного обеспечения (ПО) для различных бытовых приборов [4]. Реализация проекта была запущена на языке *C++*, но вскоре возник ряд проблем, наилучшим средством борьбы с которыми было изменение самого инструмента – языка программирования. Стало очевидным, что необходим платформонезависимый язык программирования, позволяющий создавать программы, которые не приходилось бы компилировать отдельно для каждой архитектуры и можно было бы использовать на различных процессорах под различными операционными системами. Язык *Java* потребовался для создания интерактивных продуктов для сети *Internet*. Фактически, большинство архитектурных решений, принятых при создании *Java*, было продиктовано желанием предоставить синтаксис, сходный с *C* и *C++*. В *Java* используются действительно идентичные соглашения для объявления переменных, передачи параметров, операторов и для управления потоком выполнением кода. В *Java* добавлены все лучшие черты *C*.

Три ключевых элемента объединились в технологии языка *Java*, что сделало ее в корне отличительной от всего, существующего на сегодняшний день [4]:

1) *Java* предоставляет для широкого использования свои апплеты (*applets*) – небольшие, надежные, динамичные, не зависящие от платформы активные сетевые приложения, встраиваемые в страницы *web*. Апплеты *Java* могут настраиваться и распространяться потребителям с такой же легкостью, как любые документы *HTML*;

2) *Java* высвобождает мощь объектно-ориентированной разработки приложений, сочетая простой и знакомый синтаксис с надежной и удобной в работе средой разработки. Это позволяет широкому кругу программистов быстро создавать новые программы и новые апплеты;

3) *Java* предоставляет программисту богатый набор классов объектов для явного абстрагирования многих системных функций, используемых при работе с окнами, сетью и для ввода-вывода. Ключевая черта этих классов заключается в том, что они обеспечивают создание независимых от используемой платформы абстракций для широкого спектра системных интерфейсов.

Программы на *Java* транслируются в байт-код *Java*, выполняемый виртуальной машиной *Java* (*JVM*), – программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор [5].

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять *Java*-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии *Java* является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

Часто к недостаткам концепции виртуальной машины относят снижение производительности [6]. Приведем ряд усовершенствований, несколько увеличивших скорость выполнения программ на *Java*:

- применение технологии трансляции байт-кода в машинный код непосредственно во время работы программы (*JIT*-технология) с возможностью сохранения версий класса в машинном коде;

- широкое использование платформенно-ориентированного кода (*native*-код) в стандартных библиотеках;

- аппаратные средства, обеспечивающие ускоренную обработку байт-кода (например, технология *Jazelle*, поддерживаемая некоторыми процессорами фирмы *ARM*).

По данным сайта *shootout.alioth.debian.org* для семи разных задач время выполнения на *Java* составляет в среднем в полтора-два раза больше, чем для *C/C++*. В некоторых случаях *Java* быстрее, а в отдельных – в 7 раз медленне [7]. С другой стороны, для большинства из них потребление памяти *Java*-машиной было в 10–30 раз больше, чем программой на *C/C++*. Также примечательно исследование, проведенное компанией *Google*, согласно которому отмечается существенно более низшая производительность и бóльшее потребление памяти в тестовых примерах на *Java* в сравнении с аналогичными программами на *C++*.

Идеи, заложенные в концепцию, и различные реализации среды виртуальной машины *Java* вдохновили многих энтузиастов на расширение перечня языков, которые могли бы быть использованы для создания программ, исполняемых на виртуальной машине. Эти идеи нашли также выражение в спецификации общеязыковой инфраструктуры *CLI*, заложенной в основу платформы *.NET* компанией *Microsoft*.

C# (произносится «си шарп») – объектно-ориентированный язык программирования. Разработан в 1998–2001 гг. группой инженеров под руководством Андерса Хейлсберга в компании *Microsoft* как язык разработки приложений для платформы *Microsoft .NET Framework* и впоследствии был стандартизирован как *ECMA*-334 и *ISO/IEC* 23270 [8].

C# относится к семье языков с *C*-подобным синтаксисом, из них его синтаксис наиболее близок к *C++* и *Java* [8]. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, свойства, обобщенные типы и методы, итераторы, анонимные функции с поддержкой замыканий, *LINQ*, исключения, комментарии в формате *XML*.

Переняв многое от своих предшественников – языков *C++*, *Pascal*, Модула, *Smalltalk* и в особенности *Java* – *C#*, опираясь на практику их использования, исключает некоторые модели, зарекомендовавшие себя как проблематичные при разработке программных систем, например, *C#* в отличие от *C++* не поддерживает множественное наследование классов (между тем допускается множественное наследование интерфейсов).

C# разрабатывался как язык программирования прикладного уровня для *CLR* и, как таковой, зависит прежде всего от возможностей самой *CLR*. Это касается преимущественно системы типов *C#,* которая отражает *Base Class Library (BCL)* – стандартную библиотеку классов платформы *.NET Framework* [8]. Присутствие или отсутствие тех или иных выразительных особенностей языка диктуется тем, может ли конкретная языковая особенность транслировать в соответствующие конструкции *CLR*. Так, с развитием *CLR* от версии 1.1 к 2.0 значительно обогатился и сам *C#*; подобного взаимодействия следует ожидать и в дальнейшем (однако эта закономерность была нарушена с выходом *C#* 3.0, представляющего собой расширения языка, не опирающиеся на расширения платформы *.NET*). *CLR* предоставляет *C#,* как и всем другим *.NET*-ориентированным языкам, многие возможности, которых лишены «классические» языки программирования. Например, сборка мусора не реализована в самом *C#,* а производится *CLR* для программ, написанных на *C#* точно так же, как это делается для программ на *VB.NET, J#* и др.

На сегодняшний день *Hypertext Preprocessor (PHP)* – препроцессор гипертекста – является наиболее распространенным языком *web*-программирования. Подавляющее большинство сайтов и *web*-сервисов в сети Интернет написано с помощью *PHP*. По некоторым оценкам *PHP* применяется более чем на 80 % сайтов, среди которых такие сервисы, как *facebook.com, vk.com, baidu.com* и др. И такая популярность неудивительна. Простота языка позволяет быстро и легко создавать сайты и порталы различной сложности [9].

PHP был создан в 1994 г. датским программистом Расмусом Лердорфом и изначально представлял собой набор скриптов на другом языке – *Perl* [9]. Позже этот набор скриптов был переписан в интерпретатор на языке *C*. И с самого возникновения *PHP* представлял удобный набор инструментов для упрощенного создания *web*-сайтов и *web*-приложений.

PHP предоставляет следующие преимущества [9]:

- для всех имеющих наибольшее распространение операционных системам (*Windows, MacOS, Linux*) есть свои версии пакетов разработки на *PHP*, а это значит, что вы можете создавать *web*-сайты на любой из этих операционных систем;

- *PHP* может работать в связке с различными *web*-серверами: *Apache, Nginx, IIS*;

- простота и легкость освоения. Как правило, имея даже небольшой опыт в программировании на *PHP*, можно создавать простенькие *web*-сайты;

- *PHP* похож на язык *Cи*, поэтому, зная *Cи* или один из языков с *C*-подобным синтаксисом, будет проще овладеть *PHP*;

- *PHP* поддерживает работу с множеством систем баз данных (*MySQL, MSSQL, Oracle, Postgre, MongoDB* и др.);

- распространенность хостинговых услуг и их дешевизна. Как правило, хостинговые компании размещают *web*-сайты на *PHP* на *web*-серверах *Apache* или *Nginx,* которые работают на одной из операционных систем семейства *Linux*. И

web-серверы, и операционные системы на базе *Linux* бесплатны, что снижает общую стоимость использования хостинга.

PHP продолжает развиваться, выходят все новые версии, которые несут новые функции, адаптируя язык программирования к новым вызовам. И, как правило, перейти на новую версию не составляет труда.

1.4 Задания к лабораторной работе №1

1 Изучите теоретическую часть.

2 Согласно варианту, выданному преподавателем, напишите простейшую программу на одном из объектно-ориентированных языков, перечисленных в теоретической части. Выделите ключевые особенности синтаксиса языка.

3 Подготовьте отчет по лабораторной работе.

1.5 Контрольные вопросы

1 Дайте определение понятию *web*-приложение.

2 Перечислите основные отличия между мобильными и *desktop*-приложениями.

3 Перечислите виды мобильных *web*-приложений и охарактеризуйте их.

4 Перечислите современные объектно-ориентированные языки программирования.

Лабораторная работа №2. Платформы мобильных устройств

Цель: изучить основные мобильные операционные системы; ознакомиться с возможностями и особенностями операционных систем *Android, iOs* и *Windows Mobile*.

2.1 Операционная система *Android*

Android – операционная система для смартфонов, планшетных компьютеров, электронных книг, цифровых проигрывателей, «умных» наручных часов, игровых приставок, нетбуков, смартбуков, очков *Google*, телевизоров, ситем автоматического управления автомобилем и других устройств. Ее ОС основана на ядре *Linux* и собственной реализации виртуальной машины *Java* от *Google* [10]. Изначально она разрабатывалась компанией *Android Inc*., которую в 2005 г. купила *Google*. Впоследствии *Google* инициировала создание альянса *Open Handset Alliance* (*OHA*), который сейчас занимается поддержкой и дальнейшим развитием платформы. *Android* позволяет создавать *Java*-приложения, управляющие устройством через разработанные *Google* библиотеки. *Android Native Development Kit* позволяет портировать (но не отлаживать) библиотеки и компоненты приложений, написанные на *C* и других языках. ОС *Android* установлена на 86 % смартфонов (по статистике 2014 г.).

В июле 2005 г. корпорация *Google* купила компанию *Android Inc*. 5 ноября 2007 г. компания официально объявила о создании *Open Handset Alliance* (*OHA*) и анонсировала открытую мобильную платформу *Android*, а 12 ноября 2007 г. альянс представил первую версию пакета для разработчиков *Android Early Look SDK* и эмулятор *Android* [10].

23 сентября 2008 г. официально вышла первая версия операционной системы, а также первый полноценный пакет разработчика *SDK* 1.0, *Release* 1 [10].

С момента выхода первой версии в сентябре 2008 г. произошло 40 обновлений системы. Эти обновления, как правило, касаются исправления обнаруженных ошибок и добавления новой функциональности в систему.

Изначально *Google* рассчитывала давать версиям *Android* имена известных роботов, но впоследствии от этой идеи отказалась из-за проблем с авторскими правами. Каждая версия системы, начиная с 1.5, получает собственное кодовое имя на тему сладостей. Кодовые имена присваиваются латинскими буквами в алфавитном порядке.

В 2009 г. было представлено целых четыре обновления платформы. Так, в феврале вышла версия 1.1 с исправлением различных ошибок. В апреле и сентябре появились еще два обновления – *Android* 1.5 *Cupcake* и *Android* 1.6 *Donut* соответственно [10]. Обновление *Cupcake* привнесло существенные изменения: введены виртуальная клавиатура, воспроизведение и запись видео, браузер и др. В *Donut* впервые появилась поддержка различных разрешений и плотности

экрана и сетей *CDMA*. В октябре того же года вышла версия операционной системы *Android* 2.0 *Eclair* с поддержкой нескольких аккаунтов *Google*, поддержкой браузером языка *HTML*5 и других нововведений, а также после небольшого обновления в пределах версии *Android* 2.1 *Eclair* появились «живые обои» и был видоизменен экран блокировки.

В середине 2010 г. *Google* представила *Android* 2.2 под наименованием *Froyo*, а в конце 2010 г. – *Android* 2.3 *Gingerbread*. С обновлением *Froyo* стало возможно использовать смартфон в качестве точки доступа, использовать традиционную блокировку смартфона цифровым или буквенно-цифровым паролем, а обновление *Gingerbread* привнесло более полный контроль над функцией копирования и вставки, улучшение управления питанием и контроля над приложениями, поддержку нескольких камер на устройстве и т. д.

22 февраля 2011 г. была официально представлена ориентированная на интернет-планшеты платформа *Android* 3.0 *Honeycomb* [10].

Начиная с версии 3.1 обновления выходят раз в шесть месяцев. На конференции *Google (I/O)* 2014 была представлена новая версия ОС *Android* под кодовым названием *L* (*Lolipop*), которая в данное время доступна для разработчиков *Samsung Galaxy S*4/*S*5. На настоящий момент выпущено двенадцать версий системы.

По статистике на 12 января 2018 г. доли версий распределились, как представлено в таблице 2.1 [10].

Таблица 2.1 – Доля версий *Android*

Версия	Название	Год	Доля
2.3	*Gingerbread*	2010	0,4 %
4.0	*Ice Cream Sandwich*	2011	0,5 %
4.1	*Jelly Bean*	2012	1,9 %
4.2	*Jelly Bean*	2012	2,9 %
4.3	*Jelly Bean*	2013	0,8 %
4.4	*KitKat*	2013	12,8 %
5.0	*Lollipop*	2014	5,7 %
5.1	*Lollipop*	2015	19,4 %
6.0	*Marshmallow*	2015	28,6 %
7.0	*Nougat*	2016	21,1 %
7.1	*Nougat*	2016	5,2 %
8.0	*Oreo*	2017	0,5 %
8.1	*Oreo*	2017	0,2 %

С момента выхода первой версии ОС в сентябре 2008 г. система многократно перерабатывалась и обновлялась. Последняя версия вышла в августе 2017 г.

Основными нововведениями Android 8.0 Oreo стали: новые уведомления, бейджи на иконках, «картинка в картинке», ограничение фоновых процессов, динамические иконки и др. [11].

Новые уведомления. Каким именно образом изменятся уведомления, пока неизвестно. По предварительной версии, они получат новый дизайн, как это произошло в *Android* 7.0 *Nougat*. Кроме того, некоторые источники указывают, что в новую ОС войдут наработки операционной системы *Andromeda,* слухи о разработке *Google* которой ходили в прошлом году. Так, уведомления будут синхронизироваться между устройствами и сортироваться, учитывая текущее местоположение, время и используемое устройство.

Бейджи на иконках. Над иконками приложений на рабочем столе будет отображаться количество новых событий в программе. Это позволит пользователю быстро оценить состояние приложения, посмотрев на его иконку [11].

«Картинка в картинке». Ожидается возможность смотреть видео в оконном режиме, на планшетах и смартфонах. Суть режима будет заключаться в минимизированном окне воспроизведения видео, которое будет располагаться поверх всех приложений на экране. Таким образом, например, можно будет смотреть *YouTube* и одновременно пользоваться другими сервисами [11].

Ограничение фоновых процессов. В новой версии браузера *Google Chrome* 57 появилась технология, которая ограничивает нагрузку на процессор для неактивных вкладок. Следовательно, будет более автономная работа – действие фоновых процессов будет дополнительно ограничиваться ради экономии заряда.

Динамические иконки. В новой версии приложения смогут менять иконки без обновления всего приложения. Это дает возможность приложениям отображать свой статус с помощью иконки [11].

Ранее сообщалось, что в *Android* 8.0 появится функция *Gboard Copy Less,* которая будет предугадывать, когда пользователь пытается отправить адрес найденного объекта в другом приложении. Как только будет набран указывающий на это фрагмент текста *it's at, Gboard* покажет соответствующую подсказку [11].

Еще одно нововведение *Android* 8.0 – *автоматическая вставка связей с подходящими по контексту приложениями*. Например, встретившийся в сообщении электронной почты телефонный номер будет связан с окном набора номера, адрес откроет приложение карт, а дата – календарь [11].

И еще одна опция *Android* 8.0 заключается *в распознавании экранных жестов,* открывающих в любой ситуации соответствующее приложение. Например, чтобы открыть календарь, достаточно будет изобразить пальцем на экране букву *C* [11].

Важной характеристикой мобильного устройства является возможность увеличения объема памяти. В *Android*-устройствах, как правило, присутствует

MicroSD-кардридер, позволяющий осуществить быстрый перенос файлов с компьютера на телефон, минуя скоростные ограничения *USB* и других способов передачи без извлечения карты памяти. Кроме того, в *iOS* и *Windows Phone* 7 невозможна прямая передача каких-либо файлов в/из телефона, кроме как через программы синхронизации (*iTunes* и *Zune*), в то время как телефоны на *Android* экспортируют файловую систему карты памяти как *USB mass storage device* [12].

Приложения под операционную систему *Android* являются программами в нестандартном байт-коде для виртуальной машины *Dalvik*, для них был разработан формат установочных пакетов *.APK*.

Для работы над приложениями доступно множество библиотек:

- *Bionic* (библиотека стандартных функций, несовместимая с *glibc*);

- мультимедийные библиотеки на базе *PacketVideo OpenCORE* (поддерживают такие форматы, как *MPEG*-4, *H*.264, *MP*3, *AAC, AMR, JPEG* и *PNG*);

- *SGL* (движок двухмерной графики); *OpenGL ES* 1.0 *ES* 2.0 (движок трехмерной графики);

- *Surface Manager* (обеспечивает для приложений доступ к 2*D*/*3D)*;

- *WebKit* (готовый движок для *web*-браузера; обрабатывает *HTML, JavaScript*);

- *FreeType* (движок обработки шрифтов);

- *SQLite* (легковесная СУБД, доступная для всех приложений);

- *SSL* (протокол, обеспечивающий безопасную передачу данных по сети).

По сравнению с обычными приложениями Linux приложения Android подчиняются дополнительным правилам:

- *Content Providers* – обмен данными между приложениями;

- *Resource Manager* – доступ к таким ресурсам, как файлы *XML, PNG, JPEG;*

- *Notification Manager* – доступ к строке состояния;

- *Activity Manager* – управление активными приложениями [13].

21 октября 2008 г. альянс *OHA* опубликовал исходный код платформы *Android* на открытом исходном коде *Android*: и операционную систему, и промежуточное ПО (*middleware*), и основные конечные приложения, написанные на *Java*. Общий объем исходного кода *Android* составил 2,1 ГБ. «Предпочтительной лицензией» на исходный код *Android* является лицензия *Apache* 2.0. После выпуска *Android* 3.0 *Honeycomb* президент мобильного подразделения *Google* Энди Рубин заявил о том, что открытие исходного кода новой версии системы будет отложено по причине того, что система была плохо готова для запуска на коммуникаторах и требует значительных оптимизаций. Это решение вызвало критику аналитиков. Например, обозреватель *ZDNet* Кристофер Доусон назвал такой ход *Google* разочаровывающим. Но согласно данным компанией обещаниям *Google* открыла осенью 2011 г. исходные коды следующей версии системы – *Android* 4.0 *Ice Cream Sandwich* [10].

22 октября 2008 г. *Google* объявила об открытии онлайн-магазина приложений для ОС *Android – Android Market*. По соглашению разработчики получают

70 % прибыли, операторы сотовой связи – 30 %. В феврале 2009 г. для разработчиков из США и Великобритании появилась возможность брать плату за свои приложения в *Android Market*. Компания *Sony Ericsson* первая запустила собственный канал в онлайн-магазине приложений *Android Market*. В нем представлены приложения и игры, которые рекомендованы компанией [10].

На декабрь 2011 г. с момента создания *Android Market* было скачано 10 млрд приложений.

В марте 2012 г. компания *Google* объединила мультимедийные сервисы «Книги», *Android Market,* «Музыка» и др. в единый сервис *Google Play*. Интернет-магазин *Google Play* работает в 190 странах и насчитывает более 700 тыс. приложений, за время работы сервиса зарегистрировано около 25 млрд скачиваний.

В марте 2011 г. *Android Market* оказался в центре громкого скандала после обнаружения в каталоге магазина вредоносных приложений, которые тотчас были удалены из магазина и устройств пользователей компанией *Google Inc*. Компания заявила, что вступит в контакт с партнерами для решения о выпуске срочного обновления, закрывающего уязвимости, а также гарантировала то, что приняла ряд мер, препятствующих появлению подобного вредоносного ПО в каталоге приложений. Несмотря на это, в *Google Play* до сих пор встречается ряд приложений с вредоносным ПО, и это число со временем возрастает [10].

2.2 Операционная система *iOS*

Операционная система *iOS,* в отличие от *Windows Phone* и *Google Android,* выпускается только для устройств, производимых фирмой *Apple*: для смартфона *iPhone,* медиапроигрывателя *iPod Touch* и планшетного компьютера *iPad*. Данная операционная система занимает сегодня лидирующие позиции на рынке мобильных платформ.

iOS достаточно проста в использовании: в ней нетрудно разобраться даже самому пользователю-непрофессионалу. Однако за внешней простотой скрывается довольно мощная система. На базе *iOS* работают смартфоны, медиаплееры и планшеты, поэтому можно с уверенностью сказать, что пользователь одного устройства с данной операционной системой не столкнется со сложностями использования другого устройства ввиду большой схожести функционала.

9 января 2007 г. Стив Джобс представил на одной из презентаций *Apple* на выставке-конференции *Macworld Conference* & *Expo* первый *iPhone* на базе *iOS*. С того момента прошло больше 7 лет, и многие поколения *iPhone*, *iPod Touch* и *iPad* кардинально изменили внешнюю оболочку смартфонов и планшетных компьютеров [14]. *iOS* – одна из старейших мобильных платформ, однако это вовсе не значит, что в ней не хватает функций, инструментов или же мощности. Совсем наоборот, корпорация *Apple* смогла зарекомендовать *iOS* одной из самых функциональных и поддерживаемых операционных систем, существующих на данный момент. Каждый год специалисты и разработчики совершенствуют *iOS*, о

чем говорит увеличивающееся количество пользователей данной мобильной платформы.

Отличительными особенностями платформы iOS являются [15]: конфиденциальность данных, высокий уровень безопасности, огромное количество встроенных функций.

Конфиденциальность данных. Никакая программа без вашего согласия не получит доступ к личным данным, т. е. только с вашего ведома сторонние приложения получат доступ к адресной книге, вашему местонахождению, фото- и видеофайлам.

Высокий уровень безопасности. Разработчики ОС постарались максимально обезопасить систему от возможного заражения вредоносным программным обеспечением.

Огромное количество встроенных функций. В систему *iOS* включено огромное количество полезных для пользователя функций, что по достоинству оценили владельцы мобильных устройств от *Apple*.

До выхода первого *iPhone Apple* стояла на пороге трудностей, как финансовых, так и прогрессивных. Стив Джобс искал возможность улучшить положение руководимой им корпорации. Решением данной проблемы он видел необходимость разработки мобильной платформы для первого смартфона. Так, Стив Джобс создал два конкурентных лагеря: одно подразделение пыталось усовершенствовать и доработать платформу *Mac OS X* для мобильных устройств, а другое подразделение работало над новой операционной системой плееров *iPod* [16].

Скотт Форстал работал, ориентируясь на *Mac OS X*. Его подразделение, состоящее из 15 разработчиков, занималось тестированием урезанной версии данной ОС на устройствах с меньшими функциональными возможностями и ограниченным временем автономной работы, нежели настольные компьютеры. Тони Фадел был руководителем второго подразделения, которое занималось разработкой операционной системы для *iPod*, пытаясь наделить ее функциями смартфона [16].

Конкуренция этих двух групп была достаточно жесткой, так как оба менеджера боролись за репутацию в компании. В итоге подразделение Форстала одержало победу, и первая внутренняя версия *iOS* вышла в свет. Она не поддерживала загрузку каких-либо сторонних приложений. В корпорации предполагали, что набор основных сервисов напишет сама *Apple*, остальные будут дополнены *Google*, а разработчики напишут *HTML* под *web*-браузер *Safari* [16].

Еще до презентации *iPhone* Скотт Форстал поддержал Стива Джобса в том, что у *Apple* нет необходимости создавать экосистему, потому что считалось, что для мобильного устройства будет достаточно просто объединить плеер *iPod* с новым быстрым *web*-браузером. Оставалось решить, что делать с популярными развлечениями, такими как просмотр видео с *YouTube*. В этом случае было предусмотрено сотрудничество с одним из лидеров рынка – компанией *Google*, которая написала приложение для *iPhone*.

Концепция изменилась, когда пользователи *iPhone* стали устанавливать джейлбрейк-расширения с целью иметь возможность запустить неавторизированные приложения от сторонних разработчиков. *Apple* начала заниматься написанием официального инструментария, с помощью которого каждый программист может разработать приложение для *iPhone*. А также началось создание раздела *App Store* в *iTunes*, где можно скачать или приобрести необходимое пользователю приложение [16].

Изначально данную мобильную платформу называли *iPhone OS*, и под таким названием она была известна следующие три года. Ее переименовали 7 июня 2010 г. в *iOS* ввиду того, что *iPhone* уже был не единственным устройством, поддерживающим *iOS*.

Традиционно каждый год на конференции *WWDC Apple* представляет новые мобильные устройства и новые версии операционной системы. В таблице 2.2 приведен список версий ОС [16].

Таблица 2.2 – Список версий *iOS*

Версия	Номер сборки	Дата релиза	Последняя версия для мобильных устройств
3.1.3	7*E*18	2010-02-02	*iPhone* (1-е поколение); *iPod touch* (1-е поколение)
4.2.1	8*C*148	2010-11-22	*iPhone* 3*G; iPod touch* (2-е поколение)
5.1.1	9*B*206	2012-05-07	*iPod touch* (3-е поколение); *iPad* (1-е поколение)
6.1.6	10*B*400	2014-02-21	*iPhone* 3*GS; iPod touch* (4-е поколение)
7.1.2	11*D*257 11*D*258	2014-06-30	*iPhone* 4
9.3.5	13*G*36	2016-08-25	*iPhone* 4s, *iPod touch* (5-е поколение); *iPad* 2, *iPad* (3-е поколение); *iPad mini*
10.3.3	14*F*89 14*F*90 14*F*91 14*G*60	2017-07-19	*iPhone* 5; *iPhone* 5c; *iPad* 4
11.2.1	15*C*153	2017-12-14	*iPhone* 5*s*, *iPhone* 6, *iPhone* 6 *Plus*, *iPhone* 6*s*, *iPhone* 6*s Plus*, *iPhone* 7, *iPhone* 7 *Plus* , *iPhone* 8, *iPhone* 8 *Plus*, *iPhone SE*, *iPhone X* *iPad Air*, *iPad Air* 2, *iPad* 7, *iPad mini* 2, *iPad mini* 3, *iPad mini* 4, *iPad Pro* *iPod touch* 6

Оперативная система *iOS* – универсальный помощник в учебе, работе и повседневной жизни. Благодаря встроенным функциям мобильное устройство способно помочь разобраться с самыми сложными задачами.

Среди встроенных функций следует выделить следующие [15]:

1) *Touch ID*. Благодаря данной технологии пользователь не сможет получить доступ к чужому мобильному устройству. Доступ к данным телефона или планшета будет только у владельца, которого устройство распознает по отпечатку пальца;

2) *VoiceOver*. Эта функция позволяет пользоваться разработкой *Apple* людям с плохим зрением и слепым. В основе этой технологии лежит сопровождение озвучкой всех действий, которые выполняет пользователь;

3) *Made for iPhone*. С помощью данной функцией можно улучшить звук в *Bluetooth*, который можно использовать как для разговора, так и для прослушивания музыки;

4) *гид-доступ*. Данное приложение дает возможность отключить ряд программ, кроме избранных. Эта функция может быть полезной для родителей, которые желают ограничить доступ детей к тем или иным программам устройства, и людям, которые имеют проблемы с восприятием;

5) *приложение «Полиглот»*. Уникальная функция, которая позволяет пользоваться телефоном или планшетом людям, не владеющим английским языком. С помощью программы «Полиглот» можно переключить раскладку клавиатуры более чем на 50 языков. Кроме того, приложение может распознавать более 20 языков на слух.

Операционная система Apple iOS обладает следующим рядом особенностей [15]:

1) *высокая скорость работы*. Платформа *iOS* обладает высокой скоростью работы. Динамика использования интерфейса способна удивить того, кто впервые взял гаджет от *Apple* в свои руки;

2) *интуитивно понятный интерфейс*. Даже самый не опытный пользователь сможет быстро и легко разобраться со всеми особенностями платформы. Простота, удобство и многофункциональность интерфейса превращают *iOS* в одну из самых надежных и популярных платформ;

3) *удобная файловая система*. Для того чтобы найти любой необходимый вам файл, достаточно совершить несколько нажатий пальцем на экран. Файловая система максимально проста и понятна;

4) *наличие большого количества приложений для ОС*. С момента выхода первой версии платформы и до сегодняшнего дня было создано множество специальных приложений для решения различных задач. Кроме того, количество развлекательных программ под *iOS* огромно и разнообразно, любой пользователь найдет и скачает на *iTunes* что-то для себя;

5) *постоянное повышение функциональности*. Благодаря регулярному обновлению функциональность мобильного устройства постоянно повышается. Разработчики ОС ведут работу непрерывно.

Видно, что корпорация *Apple* вложила много сил на создание и усовершенствование системы *iOS*. Если первая версия имела лишь несколько важных функций, то последняя стала практически идеальной для современного пользователя:

включает в себя набор полезных функций, стильный дизайн и удобна в использовании. С каждой версией *iOS* совершенствовалась все больше и больше, привлекая новых пользователей и закрепляя за собой прежних. По статистике *iOS* – вторая в мире по популярности мобильная платформа, которая постепенно набирает все больше и больше известность и признание.

2.3 Операционная система *Windows Phone*

Windows Phone – мобильная операционная система, разработанная *Microsoft* и вышедшая 11 октября 2010 г. 21 октября начались поставки первых устройств на базе новой платформы. В России телефоны с *Windows Phone* начали продаваться 16 сентября 2011 г., первым из которых стал *HTC 7 Mozart*. 9 октября 2017 г. исполнительный директор *Microsoft* Джо Бельфиор заявил о прекращении создания новых устройств и обновлений *Windows* 10 *Mobile* [17].

Операционная система является преемником *Windows Mobile,* хотя и несовместима с ней. *Windows Phone* – ОС с полностью новым интерфейсом и впервые с интеграцией сервисов *Microsoft*: игрового *Xbox Live* и медиаплеера *Zune*. В отличие от предшествующей системы *Windows Phone* в большей степени ориентирован на рынок потребителей, чем на корпоративную сферу [17].

Работа над масштабным обновлением *Windows Mobile* могла начаться еще в 2004 г. под рабочим названием *Photon,* но процесс двигался медленно, и в результате проект был закрыт. В 2008 г. *Microsoft* переформировала команду *Windows Mobile* и начала разработку новой мобильной операционной системы. Выход продукта под названием *Windows Phone* был анонсирован на 2009 г., но в связи с некоторыми отсрочками *Microsoft* решила разработать *Windows Mobile* 6.5 в качестве промежуточной версии. Еще одной причиной для ее создания стала несовместимость новой операционной системы с приложениями *Windows Mobile*. Старший продакт-менеджер *Windows Mobile* Ларри Либерман также объяснил это стремлением *Microsoft* по-новому взглянуть на рынок мобильных телефонов, учитывая как интересы конечных пользователей, так и корпоративных сетей [17].

Windows Phone 7. 15 февраля 2010 г. в Барселоне впервые был анонсирован выход новой мобильной операционной системы – *Windows Phone* 7. Первая версия была официально представлена 11 октября 2010 г., а 21 октября смартфоны на новой платформе появились в продаже в Европе и Азиатско-Тихоокеанском регионе. *Windows Phone* 7 была доступна на пяти языках [17].

Windows Phone 7.5. В феврале 2011 г. на *Mobile World Congress* 2011 корпорация *Microsoft* впервые анонсировала следующее обновление *Windows Phone,* а уже в апреле того же года на конференции *MIX*2011 в Лас-Вегасе познакомила с подробностями того же обновления, которое получило название *Mango*. Позже компания официально заявила, что это обновление операционной системы получит порядковый номер 7.5.

Windows Phone 8. 20 июня 2012 г. на организованной *Microsoft* конференции под названием *Windows Phone Summit* была анонсирована *Windows Phone* 8 [17].

Windows Phone 8.1. Презентация нового обновления прошла 2 апреля 2014 г. на ежегодном мероприятии компании *Microsoft – Build* 2014 [17].

Windows Phone 10. 21 января 2015 г. *Microsoft* официально представила следующее поколение мобильной ОС *Windows* 10 *Mobile* для устройств с диагональю экрана до девяти дюймов. Операционная система является специальной редакцией *Windows* 10 и обеспечивает единую экосистему для разного рода устройств при помощи *Universal Windows Platform* [17].

Для устройств на *Windows Phone* предусмотрен интернет-магазин программ и игр *Windows Phone Store* (ранее *Windows Phone Marketplace*), доступный в 60 странах. Покупка или установка этих приложений возможна через раздел *Marketplace* на телефоне или через браузер. В конце июня 2012 г. компания *Microsoft* официально заявила, что в магазине количество приложений превысило 100 тыс. На июнь 2015 г. количество приложений в *Windows Phone Store* составило 380 тыс. [17].

В октябре 2017 г. *Microsoft* прекратила развивать операционную систему для мобильных устройств *Windows Phone* (*WP*). Вице-президент корпорации Джо Белфиоре объявил, что *Microsoft* не будет обновлять *WP* за исключением элементов, связанных с безопасностью. Компания пыталась заинтересовать разработчиков приложений для *WP*, спонсировало их, но количество пользователей оказалось слишком мало, чтобы инвестировать в платформу. По данным *IDC* в первом квартале 2017 г. *Windows* была установлена всего на 0,1 % смартфонов в мире. За год эта доля сократилась на 0,7 процентных пунктов [17].

2.4 Задания к лабораторной работе №2

1 Изучите теоретическую часть.

2 Проведите сравнительный анализ операционных систем *Android, iOs* и *Windows Phone*. Выделите основные достоинства и недостатки, приведите различия в разработке приложений для каждой операционной системы.

3 Подготовьте отчет по лабораторной работе.

2.5 Контрольные вопросы

1 Дайте краткую характеристику операционной системе *Android*.

2 Дайте краткую характеристику операционной системе *iOs*.

3 Дайте краткую характеристику операционной системе *Windows Phone*.

Лабораторная работа №3. Шаблоны проектирования для мобильных устройств

Цель: изучить основные шаблоны проектирования; рассмотреть отношения и взаимодействия между классами или объектами в данных шаблонах проектирования.

3.1 Шаблоны проектирования для мобильных устройств

Паттерн – повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста [18].

Самым широко используемым шаблоном является *Model – View – Controller* (*MVC*). *MVC* – это фундаментальный паттерн, который нашел применение во многих технологиях, дал развитие новым технологиям и каждый день облегчает жизнь разработчикам [19].

Впервые паттерн *MVC* появился в языке *SmallTalk* [19]. Разработчики должны были придумать архитектурное решение, которое позволяло бы отделить графический интерфейс от бизнес-логики, а бизнес логику от данных. Таким образом, в классическом варианте *MVC* состоит из трех частей, которые и дали ему название (рисунок 3.1).

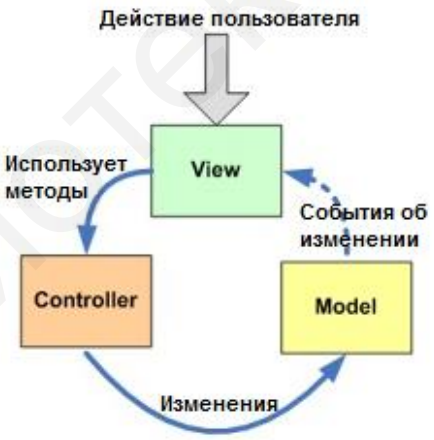


Рисунок 3.1 – Схема паттерна *MVC* [19]

Под моделью (*Model*) обычно понимают часть, содержащую в себе функциональную бизнес-логику приложения [19]. Модель должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна и способе его отображения. Достигается результат, позволяющий менять представление данных, то, как они отображаются, не затрагивая саму модель.

Модель обладает следующими признаками [19]:

- модель – это бизнес-логика приложения;
- модель обладает знаниями о себе самой и не знает о контроллерах и представлениях;
- для некоторых проектов модель – это просто слой данных (*DAO*, база данных, *XML*-файл);
- для некоторых проектов модель – это менеджер базы данных, набор объектов или просто логика приложения.

В обязанности представления (*View*) входит отображение данных, полученных от модели [19]. Однако представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом к данным «только на чтение».

Представление обладает следующими признаками [19]:

- в представлении реализуется отображение данных, которые извлекаются из модели любым способом;
- в некоторых случаях, представление может иметь код, который реализует некоторую бизнес-логику.

Контроллер (*Controller*) интерпретирует действия пользователя, оповещая модель о необходимости изменений [19]. Контроллер управляет запросами пользователя (получаемые в виде запросов *HTTP GET* или *POST*, когда пользователь нажимает на элементы интерфейса для выполнения различных действий). Его основная функция – вызывать и координировать действие необходимых ресурсов и объектов, нужных для выполнения действий, задаваемых пользователем. Обычно контроллер вызывает соответствующую модель для задачи и выбирает подходящий вид.

Самое очевидное преимущество, которое мы получаем от использования концепции *MVC*, – это четкое разделение логики представления (интерфейса пользователя) и логики приложения [20].

Поддержка различных типов пользователей, которые используют различные типы устройств, является общей проблемой наших дней. Предоставляемый интерфейс должен различаться, если запрос приходит с персонального компьютера или с мобильного телефона. Модель возвращает одинаковые данные, единственное различие заключается в том, что контроллер выбирает различные виды для вывода данных.

Помимо изолирования видов от логики приложения концепция *MVC* существенно уменьшает сложность больших приложений. Код получается гораздо более структурированным, и тем самым облегчается поддержка, тестирование и повторное использование решений [20].

Model – View – Presenter

Model – View – Presenter (*MVP*) – шаблон, который впервые появился в *IBM*, а затем использовался в *Taligent* в 1990-х гг. *MVP* является производным от

MVC, при этом имеет несколько иной подход. В *MVP* представление не тесно связано с моделью, как это было в *MVC* [21].

Данный подход позволяет создавать абстракцию представления. Для этого необходимо выделить интерфейс представления с определенным набором свойств и методов. Презентер, в свою очередь, получает ссылку на реализацию интерфейса, подписывается на события представления и по запросу изменяет модель.

На рисунке 3.2 [19] показана структура *MVP*.

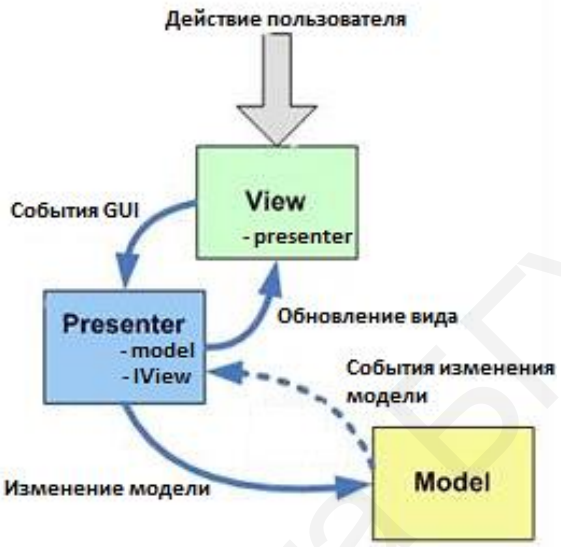


Рисунок 3.2 – Схема шаблона *Model – View – Presenter*

Presenter занял место контроллера и отвечает за перемещение данных, введенных пользователем, а также за обновление представления при изменениях, которые происходят в модели [21]. *Presenter* общается с представлением через интерфейс, который позволяет увеличить тестируемость, так как модель может быть заменена на специальный макет для модульных тестов. Так же, как и для *MVC*, давайте рассмотрим структуру *MVP* со стороны бизнес-логики приложения.

Признаки Presenter [21]:

- двухсторонняя коммуникация с представлением;
- представление взаимодействует напрямую с презентером путем вызова соответствующих функций или событий экземпляра презентера;
- презентер взаимодействует с *View* путем использования специального интерфейса, реализованного представлением;
- один экземпляр презентера связан с одним отображением.

Каждое представление должно реализовывать соответствующий интерфейс. Интерфейс представления определяет набор функций и событий, необходимых для взаимодействия с пользователем (например, *IView.ShowErrorMessage(string msg)*). Презентер должен иметь ссылку на реализацию соответствующего интерфейса, которую обычно передают в конструкторе [21].

Логика представления должна иметь ссылку на экземпляр презентера. Все события представления передаются для обработки в презентер и практически никогда не обрабатываются логикой представления (в том числе создания других представлений).

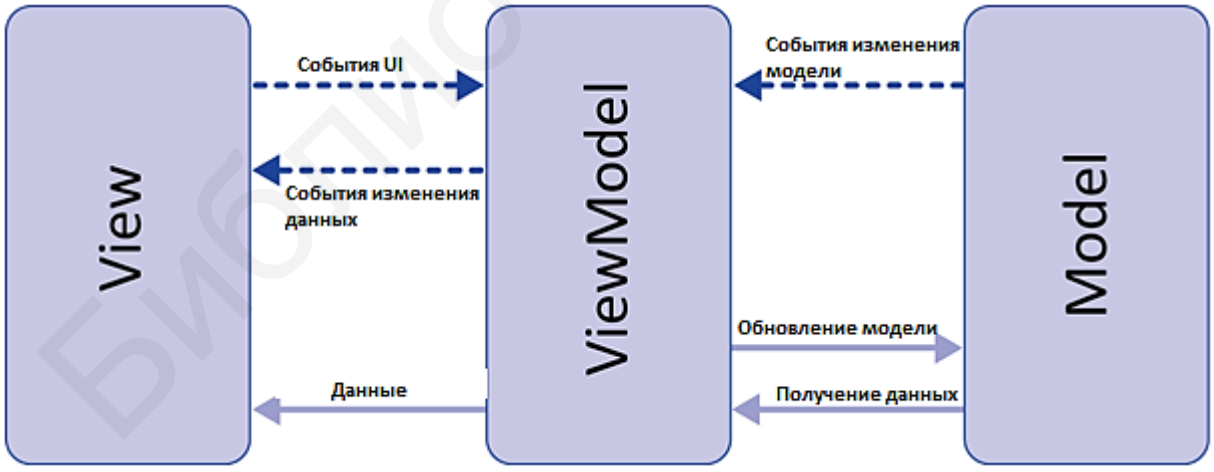
Model – View – ViewModel

Шаблон *Model – View – ViewModel* (*MVVM*) применяется при проектировании архитектуры приложения. Первоначально был представлен сообществу Джоном Госсманом в 2005 г. как модификация шаблона *Presentation Model*. *MVVM* ориентирован на современные платформы разработки, такие как *Windows Presentation Foundation, Silverlight* от компании *Microsoft* [22].

MVVM удобно использовать вместо классического *MVC* и ему подобных в тех случаях, когда в платформе, на которой ведется разработка, присутствует «связывание данных» [22].

Связывание данных – это процесс, который устанавливает соединение между *UI* (пользовательским интерфейсом) приложения и бизнес-логикой. Если настройки и уведомления установлены правильно, данные отражают изменения, когда они сделаны. Это может также значить, что, когда *UI* изменяется, лежащие в его основе данные будут отражать эти изменения [22].

Основная идея *MVVM Pattern'a* – отделить *View*, он же (*UI*) от логики, которая может происходить в рамках формы (*View*) и произвести «связывание данных» между этими двумя слоями. Это необходимо, так как позволяет изменять их отдельно друг от друга. Например, программист задает логику работы с данными, а дизайнер соответственно работает с пользовательским интерфейсом. Схематично работа паттерна представлена на рисунке 3.3 [19].



Риунок 3.3 – Схема паттерна *Model – View – ViewModel*

Модель (*Model*) так же, как в классическом паттерне *MVC,* представляет собой фундаментальные данные, необходимые для работы приложения (классы, структуры).

Вид/представление (*View*) так же, как в классическом паттерне *MVC*, – это графический интерфейс, т. е. окно, кнопки и т. п.

Модель вида (*ViewModel*, что означает *Model of View*) является, с одной стороны, абстракцией вида, а с другой – предоставляет обертку данных из модели, которые подлежат связыванию. То есть она содержит модель, которая преобразована к виду, а также содержит в себе команды, которыми может пользоваться вид, чтобы влиять на модель.

Реализация *MVVM*- и *MVP*-паттернов, на первый взгляд, выглядит достаточно простой, схожей. Однако для *MVVM* связывание представления с *View*-моделью осуществляется автоматически, а для *MVP* – необходимо программировать. *MVC* имеет больше возможностей по управлению представлением [19].

Общие правила выбора паттерна:

1) ***MVVM:***

а) используется в ситуации, когда возможно связывание данных без необходимости ввода специальных интерфейсов представления (т. е. отсутствует необходимость реализовывать *IView*);

б) частым примером является технология *WPF*;

2) ***MVP:***

а) используется в ситуации, когда невозможно связывание данных (нельзя использовать *Binding*);

б) частым примером может быть использование *Windows Forms*;

3) ***MVC:***

а) используется в ситуации, когда связь между представление и другими частями приложения невозможна (и вы не можете использовать *MVVM* или *MVP*);

б) частым примером использования может служить *ASP.NET MVC*.

Стоит отметить, что *MVC* часто трактуют просто как разделение трех уровней приложения и никак не регламентируют связи между ними. Поэтому довольно часто встречаются диаграммы (выше была приведена одна из таких), на которых модель и представление связаны стрелками, хотя очевидно, что таким образом теряются полезные свойства масштабируемости при использовании разных представлений и иерархичность контроллеров.

3.2 Изучение и применение на практике архитектурного шаблона проектирования *HMVC*

Выглядящий в теории идеальным, *MVC* на практике сталкивается с рядом проблем. Для начала вспомним, какую основную проблему он должен разрешить: разделив приложение на три различных аспекта (контроллер, вид и модель), добиться минимальной зависимости между ними, а также зависимости между различными частями приложения. Рассмотрим пример: есть модель чего-либо, вид для отображения данных и контроллер для осуществления действий в ответ на действия пользователя (рисунок 3.4) [23].

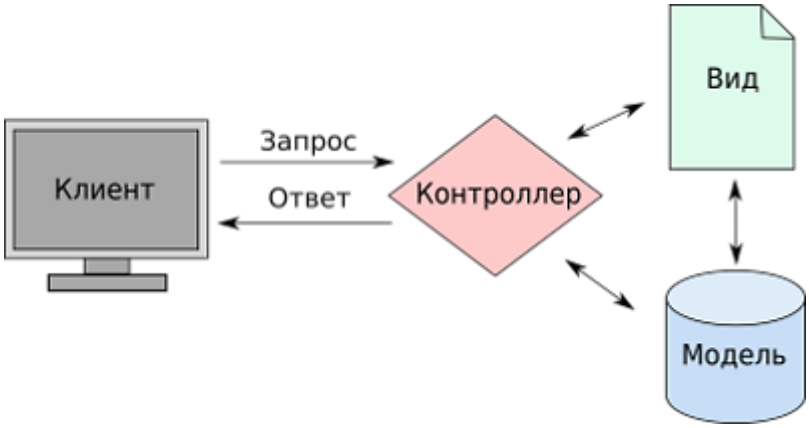


Рисунок 3.4 – Шаблон *MVC*

Проблемы MVC

Проблема первая

На практике обычно приходится оперировать одновременно несколькими моделями, например, статьями, пользователями, комментариями. В принципе, это не критично, паттерн *MVC* это предусматривает, но это увеличивает количество зависимостей: вид и контроллер зависят более чем от одной модели, а от одной модели зависят более одного вида и контроллера [23].

Проблема вторая

Для отображения данных из разных моделей хотелось бы использовать виды, созданные специально для них. Например, выглядит логичным отображение комментариев одинаково для статей и товаров. Это *MVC* не предусматривает, но этот паттерн частично обходится использованием шаблонов. То есть используется один вид, который получает данные из моделей, а для отображения их отображения используется комбинация нескольких шаблонов. И снова увеличивается количество зависимостей [23].

Проблема третья

Иногда действия надо выполнить не с одной моделью, а с несколькими одновременно. Например, при удалении пользователя надо удалить все его статьи и комментарии. В результате приходится создавать контроллер, который описывает операции не только с моделью, к которой он относится (в примере – пользователи), но и с моделями, к которым непосредственного отношения он не имеет. Таким образом, появляются неочевидные зависимости [23].

Основная идея HMVC

Поскольку проблемы возникают из-за того, что вместо *MVC* получается что-то наподобие *MMMVVVCC*, где каждая модель, вид и контроллер могут принадлежать разным подсистемам, ответ очевиден – вернуться к *MVC*, в котором есть лишь по одной модели, виду и контроллеру [23].

Первый принцип *HMVC*: в приложении используются только жестко фиксированные триады модель – вид – контроллер, которые взаимодействуют с остальными подсистемами исключительно через контроллер.

Из этого вытекает второй принцип: для организации более сложных комбинаций используются иерархии триад (рисунок 3.5) [23].

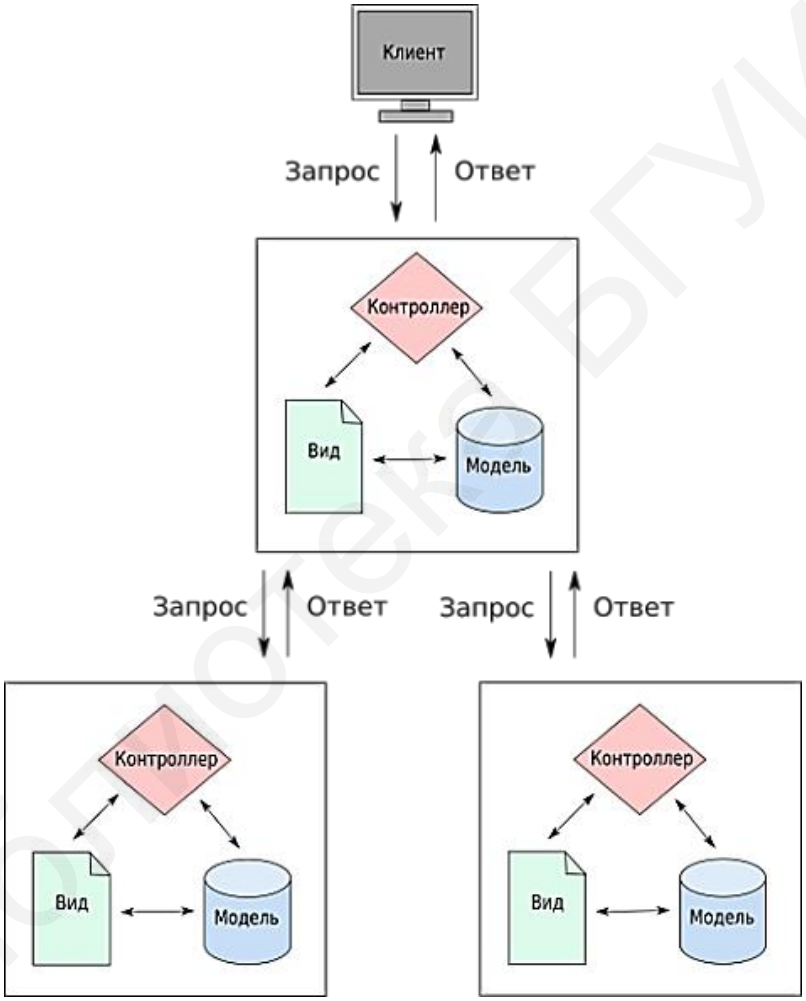


Рисунок 3.5 – Триада *HMVC*

На первый взгляд, может показаться, что для возможности реализации *HMVC* достаточно осуществить вызов контроллера из другого контроллера. Но в приложении поведение зависит не просто от команды, переданной контроллеру, а от *http*-запроса в целом. И модель, и вид могут сами анализировать запрос и некоторым образом изменять свое поведение. Поэтому требуется возможность

передать запрос другому контроллеру, причем необязательно тот же, что был получен.

Методы реализации

1 Клиент-серверный *HMVC*

Самый простой способ отправить *http*-запрос – использовать для этого браузер. В этом случае даже не требуется какая-то особая поддержка фреймворком. И этот подход используется повсеместно, называется *AJAX*. Рассмотрим пример со статьей и комментариями. Можно использовать одну базовую триаду модель – вид – контроллер для отображения основного содержимого страницы (например, текста статьи), которая остальные необходимые фрагменты (например, комментарии) получит при помощи *ajax*-запросов к таким же триадам (рисунок 3.6) [23].

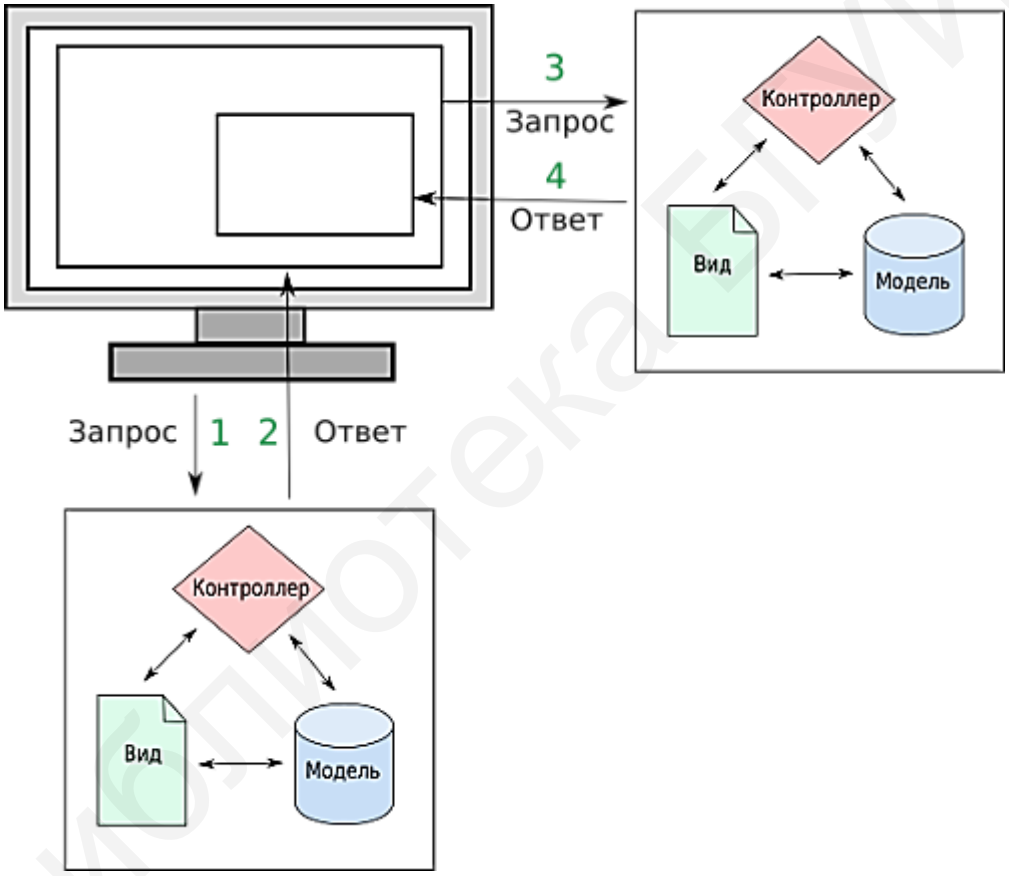


Рисунок 3.6 – Триада *HMVC* с *ajax*-запросами

Такой подход позволяет для некоторых частей страницы использовать *http*-кэширование (кэширование данных в кэше браузера, прокси-сервера либо проксирующего *http*-сервера), а часть загружать в режиме *real-time*. Например, страница статьи может быть загружена по частям следующим образом [23]:

- сама страница с текстом статьи статична и по возможности извлекается из кэша браузера, в котором может храниться довольно долго;

- комментарии постоянно обновляются и поэтому не кэшируются и каждый раз запрашиваются с *web*-сервера;

- блок последних новостей обновляется время от времени, может извлекаться из кэша, но хранится в нем не так долго, как статья.

Положительными сторонами данного подхода являются:

- нет необходимости поддержки фреймворком;

- возможность гибкого *http*-кэширования;

- возможность отправить запрос на другой *web*-сервер, распределив таким образом нагрузку между несколькими серверами.

Отрицательными сторонами, возможно, будут являться:

- необходимость писать *java*-скрипты;

- увеличение количества запросов от браузера серверу, что может увеличить время загрузки страницы и нагрузку на *web*-сервер.

2 Сервер-серверный *HMVC*

Следующим по простоте реализации действием является отправка запроса *web*-сервером самому себе. Для этого может использоваться *curl* либо другая библиотека, способная отправлять *http*-запросы. Этот подход похож на предыдущий, но разница в том, что запросы отправляет не браузер пользователя, а сам *web*-сервер в процессе формирования документа [23].

В качестве примера снова можно рассмотреть статью с комментариями. Основой страницы является статья, поэтому для ее отображения используются контроллер, вид и модель статьи. Но если в классическом *MVC* для отображения комментариев внутри вида статьи используется обращение к модели и виду комментариев, то *HTMV* предусматривает отправку запроса контроллеру комментариев. В данном случае посредством *http*-запроса, который инициирует запуск параллельного процесса, выдающего готовый блок комментариев для статьи (рисунок 3.7) [23].

Как и в предыдущем случае, при таком подходе возможно использование *http*-кэширования, но для этого необходимо применить проксирующий *http*-сервер [23].

Положительными сторонами данного подхода являются:

- клиенту отдается готовая страница;

- возможность гибкого *http*-кэширования;

- возможность отправить запрос на другой *web*-сервер, распределив таким образом нагрузку между несколькими серверами.

3 Внутрисерверный *HMVC*

Под термином «внутрисерверный» подразумевается, что все происходит внутри процесса приложения. Как и в предыдущем случае, триады модель – вид – контроллер общаются между собой посредством запросов, и восприниматься они должны ими точно так же, как обычные *http*-запросы, однако передачу этих запросов обеспечивает фреймворк. С точки зрения программиста, два последних варианта различаются между собой несущественно. В хорошем фреймворке разница должна быть лишь в одном параметре, который указывает,

должен подзапрос быть внутренним (внутри процесса) или внешним (вызывать настоящий *http*-запрос) [23].

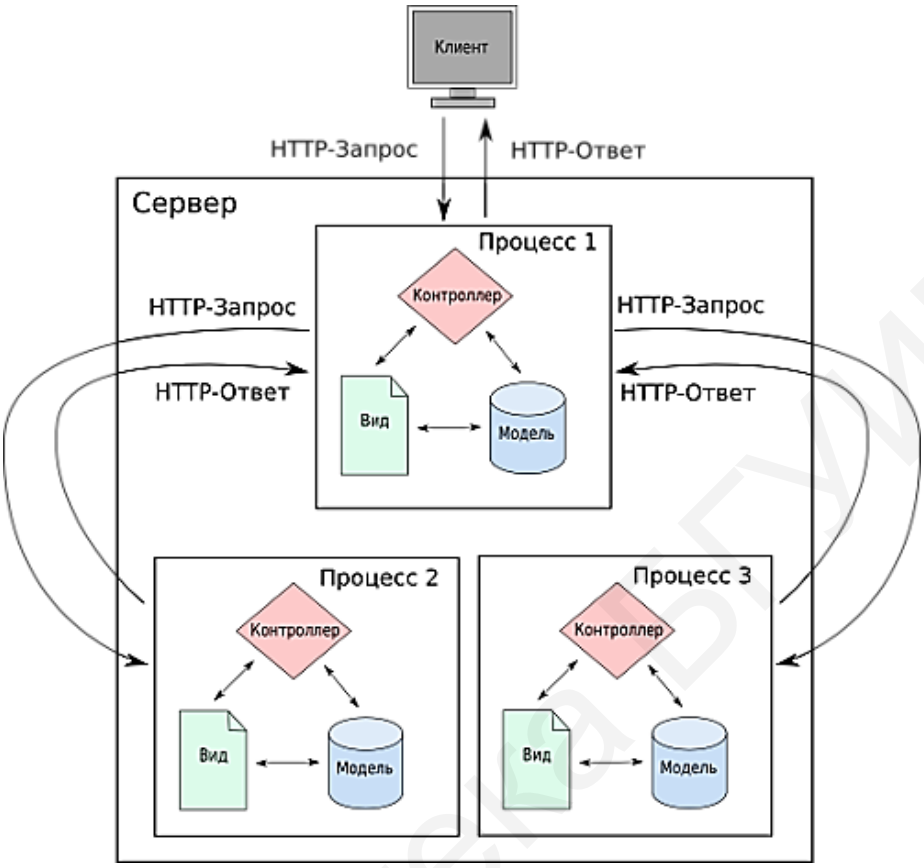


Рисунок 3.7 – Отправка запроса сервером

3.3 Задания к лабораторной работе №3

1 Изучите теоретическую часть.

2 Разработайте простейший сайт согласно варианту, выданному преподавателем. При разработке используйте основные принципы шаблона проектирования *HMVC*.

3 Подготовьте отчет по лабораторной работе.

3.4 Контрольные вопросы

1 Дайте определение понятию «паттерн».
2 Дайте характеристику шаблону *Model – View – Controller* (*MVC*).
3 Дайте характеристику шаблону *Model – View – Presenter* (*MVP*).
4 Дайте характеристику шаблону *Model – View – ViewModel* (*MVVM*).
5 Перечислите проблемы, с которыми сталкивается шаблон *MVC*.
6 Перечислите основные принципы шаблона *HMVC*.
7 Перечислите виды *HMVC*.

Лабораторная работа №4. Мобильные технологии и инструментарий

Цель: изучить основные принципы программирования под мобильное устройство; провести классификацию мобильных устройств; изучить основные среды разработки приложений *Android Studio* и *PHPStorm.*

4.1 Программирование под устройство

Операционная система *Android* построена на уникальной основе, которая предполагает полную открытость исходного кода. Данная операционная система находится в свободном распространении. Это упрощает разработчикам возможность получить непосредственный доступ к исходному коду *Android* и понять, как правильно реализованы свойства и функции разных приложений. Абсолютно любой пользователь может принять непосредственное участие в совершенствовании операционной системы *Android*. Для этого достаточно отправить отчет об обнаруженных ошибках либо принять участие в одной из групп. В сети Интернет доступны разные приложения *Android* с открытым исходным кодом, предлагаемые крупнейшей корпорацией *Google* и рядом сторонних производителей.

Открытость платформы способствует очень быстрому ее обновлению. В отличие от закрытой системы *iOS* компании *Apple*, доступной только на устройствах *Apple*, система *Android* доступна на устройствах десятков производителей оборудования (*OEM, Original Equipment Manufacturer*) и телекоммуникационных компаний по всему земному шару. Эти операционные системы жестко конкурируют между собой, что идет на пользу конечному потребителю [24].

При разработке приложений для *Android* используется объектно-ориентированный язык программирования *Java* – на данный момент это один из наиболее распространенных языков программирования. Использование языка *Java* стало вполне обоснованным выбором для платформы *Android*, потому что это мощный, свободный и открытый язык, известный миллионам разработчиков. Программисты – специалисты языка *Java* – могут очень быстро освоить *Android*-программирование, используя интерфейсы *Google Android API* (*Application Programming Interface*) и другие разработки независимых фирм [24].

Язык *Java* является объектно-ориентированным, предоставляет разработчикам доступ к мощным библиотекам классов, ускоряющих разработку тех или иных приложений. Программирование графического интерфейса пользователя является управляемым событием. Помимо непосредственного написания кода приложений, можно воспользоваться специальными средами разработки приложений *Eclipse* и *Android Studio,* позволяющими собирать графический интерфейс из готовых объектов, таких как различные кнопки и текстовые поля, перетаскивая их в определенные места экрана, добавляя различные подписи и изменяя их размеры [24]. Эти среды разработки позволяют быстро и удобно создавать, тестировать и отлаживать приложения *Android*.

В современном мире универсальные смартфоны *Android* сочетают в себе функции мобильных телефонов, интернет-клиентов, *MP*3-плееров, игровых консолей, цифровых фотоаппаратов и др. Эти портативные устройства оборудованы полноцветными мультисенсорными экранами. Простое прикосновение пальцев позволяет легко переключаться между использованием телефона, запуском приложений, воспроизведением музыки, просмотром *web*-страниц и т. д. На экране может отображаться виртуальная клавиатура для ввода электронной почты и текстовых сообщений, а также ввода данных в приложениях (некоторые устройства *Android* также оснащаются физическими клавиатурами [24].

В комплект поставки устройств *Android* входят различные встроенные приложения, набор которых зависит от поставляемого устройства, производителя или оператора мобильной связи. Обычно это приложения «Телефон» (*Phone*), «Почта» (*EMail*), «Браузер» (*Browser*), «Камера» (*Camera*) и ряд других приложений.

Web-службы (*web-services*) представляют из себя программные компоненты, хранящиеся на одном компьютере, к которым могут обращаться приложения (или другие программные компоненты) с другого компьютера по сети Интернет. На базе *web*-служб могут создаваться так называемые мэшапы (*mashups*), которые ускоряют разработку приложений путем комбинирования разных *web*-служб, используемые в различных организациях, и с различными типами вводимой информации. Например, сайт 100 *Destinations* объединяет фотографии и публикации в *Twitter* с картами *Google Maps,* чтобы пользователи могли познакомиться с разными странами по фотографиям других пользователей [24].

На сайте *Programmableweb* размещен каталог 9400 *API* и 7000 мэшапов, а также руководства и примеры кода по самостоятельному созданию мэшапов. По данным *Programmableweb* для создания мэшапов самыми популярными *API* на сегодняшний день являются *Google Maps, Twitter* и *YouTube* [24].

4.2 Виды мобильных устройств

Мобильные устройства предполагают использование любого вида компьютера в движущейся среде [25]. Непосредственно в движении могут быть использованы такие устройства, как ноутбуки, портативные мини-компьютеры, микрокомпьютеры и мобильные телефоны, а также цифровые камеры и *MP*3-проигрыватели. Все мобильные устройства используют технологии беспроводного соединения, такие, как *LAN, Wi-Fi, GPRS* и позже введенный *MAN*.

Мобильные устройства можно классифицировать по двум тематическим категориям [25]:

- портативные устройства;
- переносные устройства.

Портативные устройства фактически относятся к проводным соединениям. Сами портативные устройства мобильны, но, чтобы воспользоваться ими, существует одна потребность – подключение их к сетевым портам. Это означает,

что портативными устройствами можно воспользоваться везде, где есть доступ к сетевым портам [25].

В настоящее время мобильные устройства также называют *переносными устройствами*. Это – истинная беспроводная связь. Мало того, что сами устройства подвижны, так еще и ими воспользоваться можно практически везде. Сегодня портативные устройства находятся на стадии исчезновения; а мобильные устройства стали неотъемлемой частью жизни человека [25].

Впервые мобильные устройства были использованы в транспортных средствах. Нынешние спидометры были первыми компьютеризированными устройствами. Практически в каждом современном транспортном средстве существует несколько мобильных устройств, расположенных на приборной панели [25].

Широкое распространение на сегодняшний день имеют такие мобильные устройства:

- сотовый телефон;
- карманный персональный компьютер;
- электронная книга;
- планшет.

Сотовый телефон – мобильный телефон, предназначенный для работы в сетях сотовой связи, использует приемопередатчик радиодиапазона и традиционную телефонную коммутацию для осуществления телефонной связи на территории зоны покрытия сотовой сети [26].

В настоящее время сотовая связь – самая распространенная из всех видов мобильной связи, поэтому, как правило, мобильным телефоном называют именно сотовый телефон, хотя *мобильными телефонами*, помимо сотовых, являются также спутниковые телефоны, радиотелефоны и аппараты магистральной связи.

Сотовый телефон – сложное высокотехнологичное электронное устройство, включающее в себя:

- приемопередатчик на поддиапазоны 1–2 ГГц (*GSM*) и 2–4 ГГц (*UMTS*) СВЧ-диапазона;
- специализированный контроллер управления;
- дисплей;
- интерфейсные устройства;
- аккумулятор.

Большинство аппаратов имеет свой уникальный номер, например, *IMEI* – международный идентификатор мобильного устройства. *IMEI* присваивается при производстве сотового телефона и состоит из пятнадцати цифр; он записывается в немодифицируемую часть прошивки телефона. Сам этот номер отпечатан на этикетке телефона под аккумулятором, также на упаковочной коробке к телефону (под штрихкодом) [26].

Размер экрана современных телефонов может достигать 5,3″ (13,5 см). Поэтому в зависимости от требований, которые вы предъявляете к аппарату, необходимо выбирать дисплей соответствующих размеров. Если одной из функций

телефона, помимо звонков, является выход в сеть Интернет или чтение электронных книг, то рекомендуется выбирать телефоны с большим сенсорным экраном – от 3,5″. С другой стороны, если при использовании необходимы максимальное удобство и компактность, то стоит выбрать устройство с меньшим размером диагонали. Максимальный размер диагонали, не вызывающий дискомфорта, – 3,5…4″ [26].

Кроме выбора размера, необходимо обратить внимание на разрешающую способность дисплея: количество точек на экране (чем их больше, тем четче изображение) и технологию его выполнения.

Можно выделить три основных технологии производства экранов для мобильных устройств:

- *LCD* (либо *SLCD*);
- *TFT* (или его разновидность *IPS*);
- *AMOLED*.

К преимуществам *LCD*-дисплеев можно отнести небольшие размер и массу, отсутствие видимого мерцания, дефектов фокусировки и сведения лучей, помех от магнитных полей, проблем с геометрией изображения и четкостью. *IPS*-экраны обладают хорошими углами обзора и повышенной разрешающей способностью (большим количеством точек на экране). Особенностью *AMOLED*-дисплеев являются их высокая яркость, контрастность и хорошая цветопередача [26].

В случае если на телефоне предполагается прослушивание музыки, то в нем непременно должен быть соответствующий выход для подключения наушников. Как правило, если телефон поддерживает данную функцию, то наушники (посредственного качества) идут с ним в комплекте. Желательно, чтобы данный разъем был стандартным, т. е. 3,5 мм (*mini-jack*) – для возможности подключения любых наушников [26].

Карманный персональный компьютер (КПК) – портативное вычислительное устройство, обладающее широкими функциональными возможностями. Из-за небольших размеров КПК часто называют *наладонником*. Изначально КПК предназначались для использования в качестве электронных органайзеров. С «классического» КПК невозможно совершать звонки, и КПК не является мобильным телефоном, поэтому к настоящему времени классические КПК практически полностью вытеснены *коммуникаторами* – КПК с модулем сотовой связи и смартфонами [27].

В английском языке словосочетание «карманный ПК» (*Pocket PC*) является торговой маркой фирмы *Microsoft*, т. е. относится лишь к одной из разновидностей КПК, а не обозначает весь класс устройств. *Palm PC* – «наладонный компьютер» – также является конкретной торговой маркой. Для обозначения всего класса устройств в английском языке используется аббревиатура *PDA* [27].

КПК можно условно разделить на две категории:

- первые тип – это устройства с клавиатурой, похожие на маленький ноутбук. Из-за небольших размеров клавиатуры набирать на ней текст не очень

удобно, к тому же нажатие клавиш у большинства моделей недостаточно мягкое. Поэтому при необходимости ввести большой объем текста лучше всего воспользоваться настольным компьютером с клавиатурой обычного размера, а затем с помощью специального устройства сопряжения, входящего в комплект поставки почти всех КПК, перевести текст в память карманного компьютера [27];

- второй тип карманных компьютеров – устройства без клавиатуры. В этом случае ввод информации производится путем написания на сенсорном экране специальным пером букв или символов, которые сразу переводятся программой распознавания в текстовый файл. Кроме того, можно воспользоваться «экранной клавиатурой»: вызвать на экран изображение миниатюрной клавиатуры и ввести текст, нажимая пером на нарисованные клавиши [27].

Современные КПК снабжены цветными либо черно-белыми жидкокристаллическими дисплеями, которые, как правило, имеют подсветку. У большинства моделей экран сенсорный. Он не только предоставляет пользователям возможность рукописного ввода текста, но и заменяет привычный манипулятор – мышь. Рисовать на таком экране простые изображения и выбирать пункты меню даже проще, чем на настольном ПК [27].

Вместо винчестера и других механических устройств, служащих для хранения программ и данных, в карманных компьютерах используется энергонезависимая память на микросхемах, например, флэш-память [27]. Это позволяет запускать программы практически моментально, так как не приходится тратить время на поиск нужной дорожки на диске. К сожалению, в настоящее время флэш-память стоит довольно дорого, так что комплектация КПК большим объемом памяти существенно увеличивает их общую стоимость.

Все КПК подключаются проводами к настольным компьютерам. Кроме того, большинство из них имеет еще и инфракрасный порт, который используется не только для передачи данных на другой компьютер, но и для соединения с мобильным телефоном в целях получения и отправки электронной почты или факсимильных сообщений. Ко многим моделям карманных компьютеров можно подключать модем [27].

Электронная книга – это книга, в которой информация представлена в электронном виде [28]. До недавнего времени электронные книги существовали только в программной интерпретации, во всевозможных форматах, как обычных (например, *.txt, .doc, .htm, .chm, .pdf, .rtf, .djvu*), так и специфических (например, *.fb2*). Некоторые программные электронные книги созданы как самостоятельные приложения в формате исполняемых *exe*-файлов. Сравнительно недавно стали появляться электронные книги в аппаратной интерпретации, так называемые цифровые гаджеты (устройства для чтения электронных книг).

Важным фактором является разрешение экрана (для 5–6-дюймовых книг – обычно 800×600 пикселей, для 9–10-дюймовых – 1200×825). В книгах с дисплеем *E-Ink* указывается количество градаций серого цвета (обычно 16). Чем выше эти показатели, тем более четкую картинку мы увидим [28].

Наличие встроенной памяти обычно позволяет сохранить более 1000 книг, но при желании сохранить в устройстве большое количество любимых книг следует обратить внимание на наличие слота для карт памяти.

Показатели частоты процессора и емкости оперативной памяти дадут вам представление о быстродействии электронного устройства. Эта информация будет интересна, если вы планируете одновременно читать несколько книг, слушать при этом фоновую музыку и пользоваться доступом *Wi-Fi* в сеть Интернет [28].

Емкость аккумулятора – важный показатель, который обычно обеспечивает около 7–8 тыс. перелистываний страниц.

Из важных дополнительных функций электронных книг выделим наличие установленного словаря, возможности преобразования текста в речь. Из развлекательных, но одновременно полезных функций отметим возможность подключать к книгам электронные накопители (флешки), поддержку мультимедиа-, видео-, аудио-, фотоформатов.

Наличие в электронной книге открытой операционной системы (к примеру, *Android*) превращает электронное устройство, по сути, в небольшой планшет, позволяя загружать и устанавливать дополнительные программы для обучения либо развлечения [28].

Поддержка *Wi-Fi* и *Bluetooth* могут позволить быстро пополнить вашу библиотеку как через сеть Интернет, так и с другого устройства.

Планшет – это устройство с сенсорным экраном, которое имеет размеры обычной книги и которое выполняет функции компьютера [29].

«Планшетники» имеют меньший размер и вес, чем ноутбуки, что делает их транспортабельными. Из-за размеров и вычислительные мощности планшетов меньше, но при этом время автономной работы больше. А по сравнению со смартфонами они имеют больший размер экрана, что комфортнее при долгом просмотре видео или текстовой информации [29].

Первые модели планшетных компьютеров еще назывались *Tablet PC* и *Slate PC*. *Tablet PC* отличались от других планшетов тем, что на них уже устанавливалась операционная система с компьютеров (*Windows XP, Windows* 7).

А уже ближе к 2010 г. появились и *Slate PC* (тонкий ПК). Они были меньшего размера и еще сохраняли совместимость с *IBM PC*, что позволяло использовать программное обеспечение с обычного компьютера. На них уже устанавливались операционные системы, переработанные для сенсорных экранов [29].

Революционным этапом развития планшетов можно назвать 2010 г. Именно в этом году фирма *Apple* выпустила свой первый интернет-планшет под названием *iPad* [29]. Его появление доказало выигрышность мнения, что планшеты должны потреблять контент, а не создавать. *Tablet PC* и *Slate PC* ушли в прошлое, рынок с тех пор прочно завоевали интернет-планшеты. С того времени все устройства с сенсорным экраном размером диагонали дисплея от 7 до 12 дюймов без аппаратных клавиатуры и мышки называются *планшетами или планшетными компьютерами*. Исключением являются электронные книги, которые по размерам также подходят под это описание.

На функциональные возможности планшетного компьютера влияют размеры экрана и производительность, зависящая от процессора и объема оперативной памяти.

Так как сегодняшние планшеты являются по большей части устройствами для потребления контента, а не его создания, то и выполняемые функции этим и определяются.

Сегодня планшеты предоставляют следующие возможности [29]:

- интернет-серфинг – позволяет получать информацию с сети Интернет (текстовую, видео, фото, музыку), посещение сайтов;
- в сети Интернет пользование социальными сетями (Вконтакте, *Facebook, Twitter* и др.);
- электронная почта;
- общение по *Skype* или по другой программе;
- просмотр видео (фильмы любого разрешения, клипы, обучающие фильмы и др.);
- прослушивание музыки;
- игры (возможность загружать с хранилищ производителей операционных систем как платные, так и бесплатные);
- фото- и видеосъемка на встроенные камеры с последующей обработкой несложными редакторами на планшете;
- работа с текстовыми документами, с таблицами и другими документами (учитывая, что работать нужно с экранной клавиатурой или дополнительно подключать аппаратную клавиатуру);
- чтение электронных книг разного формата (*.fb2, .pdf, .djvu* и др.);
- *GPS*-навигация.

Функции планшетников можно расширить, установив дополнительные приложения с магазинов приложений от *Apple, Google, Microsoft*. Эти магазины компании создавали для своих операционных систем, поэтому, смотря у кого какая ОС, тот магазин и выбираете [29].

Стоит помнить, что полностью заменить персональный компьютер планшет не сможет из-за ограничений на производительность. Он скорее является дополнением к ПК, которое всегда можно взять с собой в дорогу или путешествие. Именно мобильность и является одним из главных достоинств планшетного компьютера.

4.3 *Android Studio*

Android Studio – интегрированная среда разработки производства *Google*, с помощью которой разработчикам становятся доступны инструменты для создания приложений на платформе *Android OS*. *Android Studio* можно установить на *Windows, Mac* и *Linux* [30]. Учетная запись разработчика приложений в *Google Play App Store* стоит 25 дол. *Android Studio* создавалась на базе *IntelliJ IDEA* (рисунок 4.1) [31].

IDE можно загрузить и пользоваться ей бесплатно. В ней присутствуют макеты для создания *UI*, с чего обычно начинается работа над приложением. В *Studio* содержатся инструменты для разработки решений для смартфонов и планшетов, а также новые технологические решения для *Android TV*, *Android Wear*, *Android Auto*, *Glass* и дополнительные контекстуальные модули [30].

Среда *Android Studio* предназначена как для небольших команд разработчиков мобильных приложений (даже поодиночке), так и для крупных международных организаций с *GIT* или другими подобными системами управления версиями. Опытные разработчики смогут выбрать инструменты, которые больше подходят для масштабных проектов.

Решения для *Android* разрабатываются в *Android Studio* с использованием *Java* или *C++*. В основе рабочего процесса *Android Studio* заложен концепт непрерывной интеграции, позволяющий сразу же обнаруживать имеющиеся проблемы. Продолжительная проверка кода обеспечивает возможность эффективной обратной связи с разработчиками. Такая опция позволяет быстрее опубликовать версию мобильного приложения в *Google Play App Store*. Для этого присутствует также поддержка инструментов *LINT*, *Pro-Guard* и *App Signing* [30].

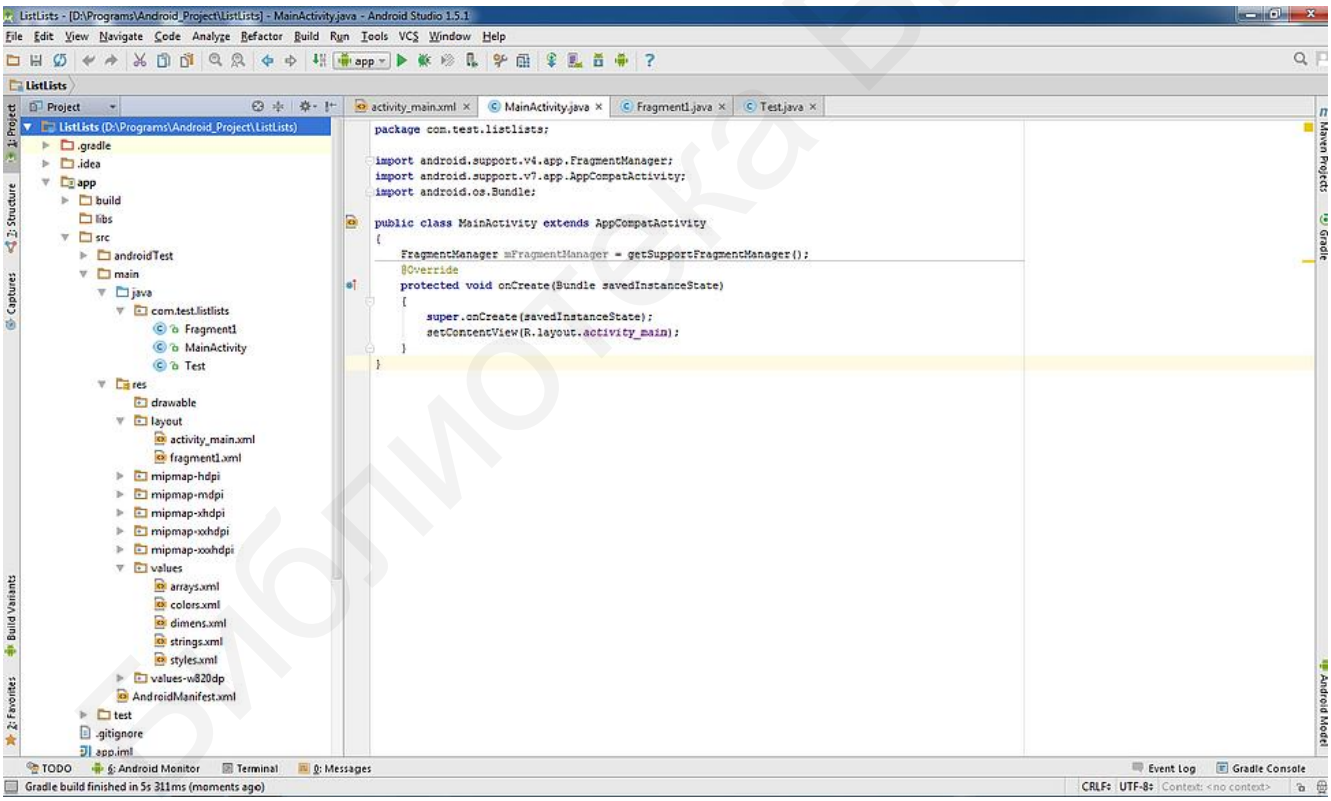


Рисунок 4.1 – Окно представления *Android Studio*

С помощью средств оценки производительности определяется состояние файла с пакетом прикладных программ. Визуализация графики дает возможность узнать, соответствует ли приложение ориентиру *Google* в 16 мс. С помощью инструмента для визуализации памяти разработчик узнает, когда его при-

ложение будет использовать слишком много оперативной памяти и когда произойдет «сборка мусора». Инструменты для анализа батареи показывают, какая нагрузка приходится на устройство [30].

Дополнительно ко всем возможностям, ожидаемым от *IntelliJ*, в *Android Studio реализованы* [31]:

- поддержка сборки приложения, основанной на *Gradle*;
- специфичный для *Android* рефакторинг и быстрое исправление дефектов;
- *lint*-инструменты для поиска проблем с производительностью, юзабилити, совместимостью версий и др.;
- возможности *ProGuard* (утилита для сокращения, оптимизации и обфускации кода) и подписи приложений;
- основанные на шаблонах «мастера» для создания общих для *Android* конструкций и компонентов;
- *WYSIWYG*-редактор, работающий с экранов и разрешений различных размеров, окно предварительного просмотра, показывающее запущенное приложение сразу на нескольких устройствах и в реальном времени;
- встроенная поддержка облачной платформы *Google*.

Android Studio совместима с платформой *Google App Engine* для быстрой интеграции в облаке новых *API* и функций [30]. В среде разработки вы найдете различные *API*, такие как *Google Play*, *Android Pay* и *Health*. Присутствует поддержка всех платформ *Android*, начиная с *Android* 1.6. Есть варианты *Android*, которые существенно отличаются от версии *Google Android*. Самая популярная из них – это *Amazon Fire OS*. В *Android Studio* можно создавать *APK* для этой ОС. Поддержка *Android Studio* ограничивается онлайн-форумами.

4.4 *PHPStorm*

Программное обеспечение *JetBrains PHPStorm* представляет собой специализированное средство *web*-разработки, ориентированные на *web*-приложения и другие виды программ, которые можно создавать с помощью языка *PHP* и с использованием *HTML*, *JavaScript* и *CSS*. Решение *PHPStorm* осуществляет развертывание и синхронизацию проектов через протокол *FTP*. Среда *PHPStorm* предлагает функции автоматического завершения языковых конструкций *PHP* в коде, инспектирование кода, различные алгоритмы рефакторинга и быструю навигацию по коду [32].

Реализованный в *PHPStorm* графический *PHP*-отладчик поддерживает условные точки останова, отслеживание значений и автоматизированный вход в отладку отдельных процедур. Для тестирования приложений поддерживается каркас тестовых модулей *PHPUnit* и графический интерфейс для запуска тестов. При редактировании кода выделяются конструкции синтаксиса, осуществляется расширенное форматирование конфигурации, выявление ошибок в режиме реального времени и завершение кода [32]. *PHPStorm*-редактор учитывает комментарии к коду при его завершении, автоматически выбирая оптимальное решение

проблемы. *PHP*-рефакторинг и редактирование шаблонов гарантируют изменение проекта в кратчайшие сроки. *PHPStorm* позволяет визуализировать код в иерархичном виде и обеспечивает быструю навигацию по всем элементам [32].

Благодаря применению *PHPUnit*-тестов можно быстро просматривать результаты генерации кода отдельных блоков или всего приложения. Если тест был проведен неудачно, продукт позволяет просматривать отдельные кодовые строки, в которых бала обнаружена ошибка. *PHPStorm* обеспечивает отладку кода *JavaScript* и предоставляет широкий диапазон возможностей: нахождение точки останова в *HTML* и *JavaScript*, настройка параметров точки останова, тестирование синтаксиса кода в режиме реального времени.

Инспекции заботятся о верификации кода, анализируя проект целиком во время разработки. Поддержка *PHPDoc, code* (*re)arranger*, форматирование кода с конфигурацией стиля кода и другие возможности помогают разработчикам писать системный и легко поддерживаемый код [32].

Данным программным обеспечением поддерживаются передовые технологии *web*-разработки, включая *HTML*5, *CSS, Sass, SCSS, Less, Stylus, Compass, CoffeeScript, TypeScript, ECMAScript Harmony*, шаблоны *Jade, Zen Coding, Emmet*, и, конечно же, *JavaScript*.

PHPStorm включает в себя всю функциональность *WebStorm* (*HTML*/*CSS*-редактор, *JavaScript* редактор) и добавляет полнофункциональную поддержку *PHP* и баз данных/*SQL*.

Ключевыми возможностями PHPStorm являются [32]:

- интеллектуальный редактор *PHP*-кода с подсветкой синтаксиса, автодополнением кода, расширенными настройками форматирования кода, предотвращением ошибок на лету;
- поддержка *PHP* 7.0, 5.6, 5.5, 5.4 и 5.3, генераторы, сопрограммы и все синтаксические улучшения;
- *PHP*-рефакторинги, *code (re)arranger*, детектор дублируемого кода;
- поддержка *Vagrant, Composer,* встроенный *REST*-клиент, *Command Line Tools, SSH*-консоль;
- поддержка фреймворков (*MVC view* для *Symfony*2, *Yii*) и специализированные плагины для ведущих *PHP*-фреймворков (*Symfony, Magento, Drupal, Yii, CakePHP* и др.);
- визуальный отладчик для *PHP*-приложений, валидация конфигурации отладчика, *PHPUnit* с покрытием кода (поддержка *PHPUnit* 5), а также интеграция с профилировщиком;
- *HTML, CSS, JavaScript*-редактор. Отладка и модульное тестирование для *JS*. Поддержка *HTML5, CSS, Sass, SCSS, Less, Stylus, Compass, CoffeeScript, TypeScript, ECMAScript Harmony, Emmet* и других передовых технологий *web*-разработки;
- полный набор инструментов для фронтенд-разработки;

- поддержка стилей кода, встроенные стили *PSR*1/*PSR*2, *Symfony*2, *Zend*, *Drupal* и др.;

- интеграция с системами управления версиями, включая унифицированный интерфейс;

- удаленное развертывание приложений и автоматическая синхронизация с использованием *FTP*, *SFTP*, *FTPS*;

- *Live Edit* (изменения в коде можно мгновенно просмотреть в браузере без перезагрузки страницы);

- *PHP UML*;

- интеграция с баг-трекерами;

- инструменты работы с базами данных, *SQL*-редактор;

- кросс-платформенность (*Windows*, *Mac OS X*, *Linux*).

JetBrains также предоставляет другую, более мощную, интегрированную среду разработки *IntelliJ IDEA*, которая может предоставить такую же функциональность, как и *PhpStorm,* с помощью *плагинов.*

4.5 Задания к лабораторной работе №4

1 Изучите теоретическую часть.
2 Установите среду разработки *Android Studio*.
3 Изучите основные возможности, предоставляемые *Android Studio*.
4 Установите среду разработки *PHPStorm.*
5 Изучите основные возможности, предоставляемые *PHPStorm.*
6 Подготовьте отчет по лабораторной работе.

4.6 Контрольные вопросы

1 Классифицируйте мобильные устройства и назовите основные их особенности.
2 Охарактеризуйте среду разработки *Android Studio*.
3 Охарактеризуйте среду разработки *PHPStorm.*

Лабораторная работа №5. Эмулятор. Ресурсы *Android*

Цель: изучить программные средства эмуляции операционной системы *Anroid*; рассмотреть ресурсы *Anroid*; основные методы работы с файловой системой *Anroid*; ознакомиться с методами подключения библиотек и виджетов в приложение для *Anroid*; изучить активности и манифест операционной системы *Anroid*.

5.1 Установка эмулятора

Для работы эмулятора операционной системы *Android* необходимо наличие на компьютере виртуальной машины *Java* (*Java Runtime Environment*), которую можно скачать на официальном сайте *Java*. Скачав эту виртуальную машину на свой компьютер, ее следует установить, следуя подсказкам мастера установки.

Собственно, сам эмулятор *Android* можно скачать с официального сайта *Android Studio*, где следует выбрать 32- или 64-разрядную версию программы, соответствующую той *Windows*-платформе, на которую она будет устанавливаться. После загрузки необходимо распаковать из архива какую-либо директорию [33].

Чтобы запустить эмулятор, вначале следует создать виртуальное устройство, работа которого будет эмулироваться. Для этого следует перейти в распакованной директории (по умолчанию имя директории может быть, например, *adt-bundle-windows-x*86-20130729, а можно и переименовать его) в поддиректорию */sdk/tools* и запустить файл *android.bat*, так что полный путь к файлу будет, например, такой: *C*: \ *Program Files\adt-bundle-windows-x*86-20130729\ *sdk\tools\android.bat* [33].

В результате появится окно менеджера *SDK* (*Android SDK Manager*), которое представлено на рисунке 5.1.

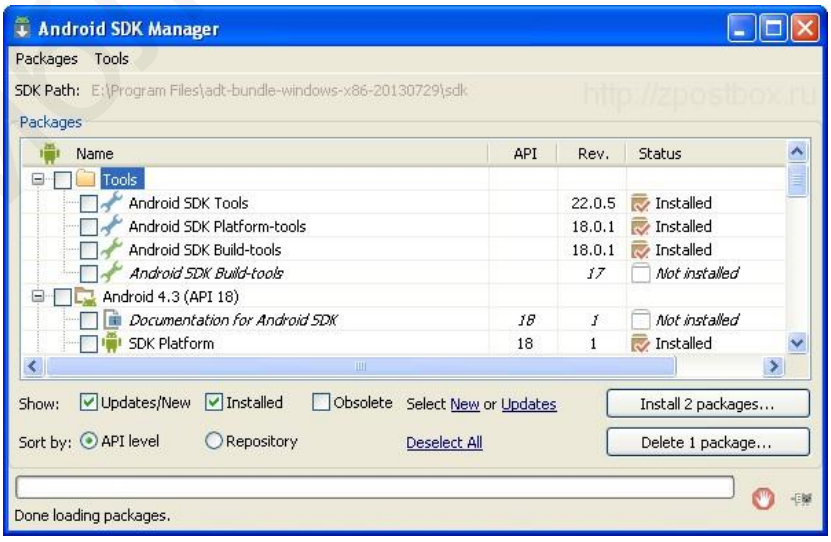


Рисунок 5.1 – Стартовое окно *Android SDK Manager*

В этом окне из системного меню *tools* следует выбрать пункт *Manage AVDs* (*Android Virtual Devices* (*AVD*) – виртуальные устройства). Появится окно менеджера виртуальных устройств *Android Virtual Device Manager*, вид которого представлен на рисунке 5.2.

Рисунок 5.2 – Окно менеджера виртуальных устройств *Android Virtual Device Manager*

Нажав кнопку *New...*, переходим к окну создания нового виртуального устройства (рисунок 5.3).

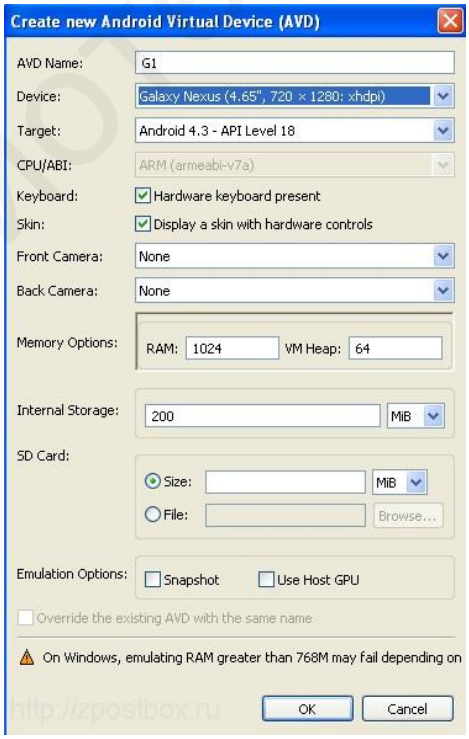


Рисунок 5.3 – Создание нового виртуального устройства

В этом окне следует дать какое-либо название новому виртуальному устройству *AVD Name* (в данном примере выбрано имя *G*1) и выбрать устройство *Device* (здесь выбрано устройство *Galaxy Nexus*). Остальные параметры можно оставить по умолчанию. Следует только отметить, что величина *Memory Option – RAM* (в данном случае 1024) не должна превышать количество свободной оперативной памяти компьютера, на котором запущен эмулятор, иначе запустить эмулятор не удастся. В этом случае величину *RAM* следует уменьшить [33].

После ввода параметров нового устройства и нажатии кнопки *OK* появится окно с параметрами нового виртуального устройства (рисунок 5.4).

Рисунок 5.4 – Параметры созданного виртуального устройства

Теперь, закрыв все ранее открытые окна, переходим непосредственно к запуску эмулятора. Для этого следует создать ярлык для программы *emulator.exe*, расположенной в этой же директории */sdk/tools*. В качестве параметров запуска необходимо указать следующее: *emulator.exe -avd G*1. Здесь *G*1 – это имя созданного виртуального устройства. На рисунке 5.5 представлены свойства ярлыка.

Вместо ярлыка можно также создать командный *bat* – пакетный файл, расположив его в директории программы */sdk/tools* с содержимым *emulator.exe-avd G*1 [33].

Щелкнув по ярлыку или командному файлу, можно запустить эмулятор (рисунок 5.6), при этом появится окно с подсветкой бегущей надписи *Android*.

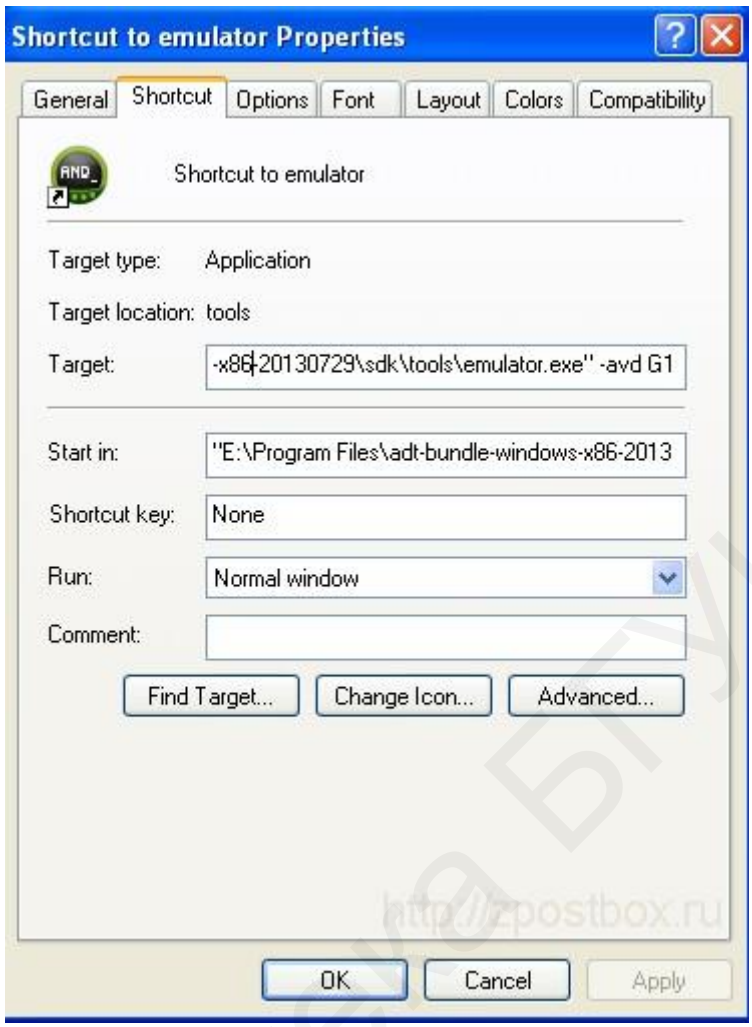


Рисунок 5.5 – Свойства ярлыка эмулятора

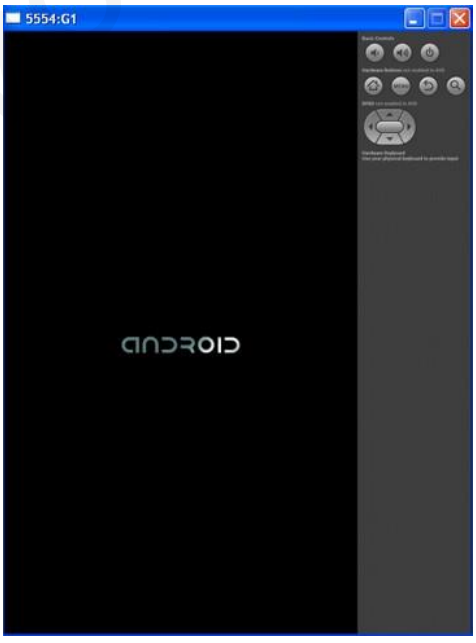


Рисунок 5.6 – Стартовое окно эмулятора *Android*

Загрузка системы займет некоторое время. После загрузки окно эмулятора *Android* будет иметь вид, представленный на рисунке 5.7.

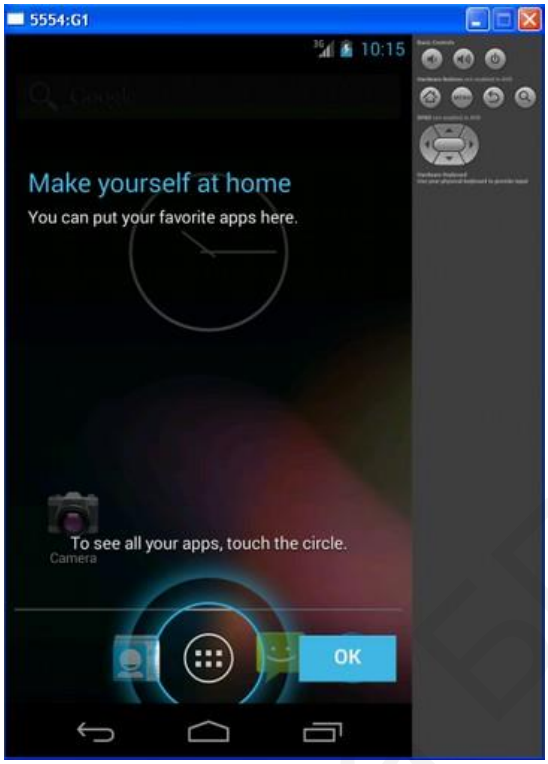


Рисунок 5.7 – Запущенный эмулятор *Android*

В данном случае на правом поле вверху видны несколько кнопок, из них активны только верхние три (для данного виртуального устройства). Самая правая верхняя кнопка включает и выключает питание устройства, при включении на экране появляется рисунок замка на экране блокировки.

Установленный эмулятор *Android* позволяет запускать и тестировать разрабатываемые приложения без мобильного устройства на базе *Android*.

5.2 Виды эмуляторов

Рассмотрим следующие виды эмуляторов:

- эмулятор в составе *SDK*;
- *Genymotion*;
- *Android x*86;
- *Bluestacks*.

Эмулятор в составе *SDK*

Входит в состав *Android Studio.*

К достоинствам эмулятора в составе SDK можно отнести [34]:

- кроссплатформенное решение;
- вход в состав *SDK*, отсутствие необходимости стороннего инструментария;

- конфигурируемость (размер памяти, подключение камеры и т. д);
- доступ по *telnet* для настройки параметров сети, батареи и т. д;
- плагин для *Eclipse*, легкий доступ через *adb*;
- обновление сразу после выхода новой версии *Android*.

К недостаткам эмулятора в составе SDK можно отнести [34]:

– медлительность, если не использовать *HAXM*;
– не является *ARM*, если использовать *HAXM*.

Genymotion

Это проприетарная реализация, выросшая из проекта *AndroVM*. По сути это виртуальная машина на *VirtualBox*. *Genymotion* достаточно удобен, быстр, имеет *Java API* для тестов. При создании устройства из сети загружается его образ. *APK* можно устанавливать, перетянув их на окно с виртуальной машиной [34].

Достоинства Genymotion:

- наличие кроссплатформенного решения;
- быстрота;
- большое количество дополнительных инструментов (контроль заряда, акселерометра, *API* для тестов и т. д.);
- плагин для *Eclipse*, легкий доступ через *adb*.

Недостатки Genymotion:

- платный для компаний (и это главный «минус»);
- не является *ARM*;
- достаточно долгий выход актуальных версий *Android*.

Android x86

Проект по портированию *Android* на платформу *x*86. Распространяется в виде образа *iso*, можно запустить/установить в виртуальной машине. Имеется возможность поставить на устройство с *x*86 процессором (например, на ноутбук) [34].

Работает быстро, но есть множество проблем из-за того, что это виртуальная машина. Например, привязывание мыши внутри окна виртуальной машин, доступ к *adb* только по сети и т. д.

Для использования в *VirtualBox* нужно отключать *Mouse Integration*, иначе в виртуальной машине не видно курсора.

*Достоинства Android x*86:

- наличие кроссплатформенного решения (везде, где есть *VirtualBox*);
- быстрота.

*Недостатки Android x*86:

- неудобный доступ к *adb*;
- неудобства, связанные с использованием *VM* – например, привязка мыши;
- не является *ARM*;
- очень долгий выход актуальных версий.

Bluestacks

Позиционируется как плеер приложений для *Windows, Mac* и *TV*. Запускает приложения, имеет доступ к магазину приложений. Неудобен для разработки и тестирования. Так, *apk* ставятся специальным инструментом из комплекта, но доступ к *adb* можно получить [34].

Достоинства Bluestacks:

- наличие кроссплатформенного решения (правда, только *Mac* и *Windows*);
- быстрота.

Недостатки Bluestacks:

- неудобно ставить приложения;
- возникновение проблем с различными версиями *Android*;
- нет версии для *Linux*.

Выбор конкретного вида эмулятора *Android* продиктован поставленными задачами. К примеру, «мобильные» игроманы по достоинству оценят эмуляторы, специально рассчитанные на запуск игр – *BlueStacks, Droid4X* и *Nox APP Player*. Для разработчиков незаменимым окажется *Genymotion*, сочетающий в себе отличную производительность и возможность протестировать работоспособность своего приложения на разнообразных устройствах, отличающихся как размерами дисплеев, так и техническими характеристиками.

5.3 Примеры использования ресурсов

Рассмотрим способы применения, формат и синтаксис основных типов ресурсов приложения *Android*, которые вы можете предоставить в каталоге проекта *res* (это относится как к *Eclipse*, так и к *Android Studio*).

Проанализируем следующие типы рессурсов:

- *Animation Resources*;
- *Property Animation*;
- *View Animation*;
- ресурс *Color*;
- ресурс *Color State List*;
- ресурсы типа *Drawable*.

Animation Resources

Здесь задаются ресурсы для анимации [35]. *Tween*-анимации (анимации с использованием промежуточных кадров) сохраняются в папке *res/anim/*, и доступ к ним осуществляется с помощью класса *R.anim*. *Frame*-анимации (покадровые анимации) сохраняются в папку *res/drawable/* (доступ через класс *R.drawable*) [35].

Ресурс анимации может задать два типа анимаций [35]:

– *Property Animation* (анимация свойства). В этом случае анимация происходит через модификацию значений свойств объекта через установленный период времени с использованием класса *Animator*;

– *View Animation* (анимация представления). Этот тип включает в себя еще два подтипа анимаций, которые вы можете создать в рабочем окружении *View*: *Tween animation* – анимация создается через серию преобразований одной картинки в классе *Animation*; *Frame animation* – в этом случае анимация будет показана как последовательность чередующихся друг за другом изображений, что делается с помощью класса *AnimationDrawable*.

Property Animation

Анимация создается в файле *XML* [35]. Она будет модифицировать свойства целевого объекта в определенные промежутки времени. Модифицируемыми свойствами могут быть, к примеру, цвет заднего плата (*background color*) или значение *alpha*.

View Animation

Фреймворк *view animation* поддерживает оба типа анимаций: *tween* и *frame by frame*; и тот и другой типы декларируются в *XML* [35]. Далее описываются оба этих вида анимаций.

Анимация *Tween* задается в *XML* и выполняет такие преобразования, как поворот, затенение (*fading*), перемещение (*moving*) и растягивание/сжатие (*stretching*) для элементов графики.

Анимация *Frame* также задается в *XML* и работает как показ последовательности изображений (точно так же, как устроен классический кинематограф).

Ресурс *Color*

Color – это целое число, кодирующее цвет [35]. Ресурс цвета *Color* – это же целое число, которое задано в *XTML*. Цвет указывается виде значения *RGB* и канала альфа (*alpha channel* – число, которое кодирует прозрачность цвета). Можно использовать ресурс цвета в любом месте, которое принимает значение цвета в шестнадцатеричном виде. Также можно использовать ресурс цвета там, где подразумевается использование в *XML* рисуемого (*drawable*) ресурса.

Ресурс *Color State List*

Это так называемый цветовой список состояний. *ColorStateList* является объектом, заданным в *XML*, который вы можете применить как цвет, но он в реальности будет менять цвета в зависимости от состояния объекта *View*, к которому применен [35]. Например, виджет кнопки *Button* может существовать в одном из некоторых различных состояний (кнопка нажата, получила фокус или ни то, ни другое), так что вы можете, используя цветовой список состояний (*color state list*), предоставить разный цвет для каждого состояния кнопки.

Вы можете описать список состояний в файле *XML*. Каждый цвет задается как элемент *<item>* внутри одного элемента *<selector>*. Каждый *<item>* использует разные атрибуты для описания состояния, которое должно быть использовано [35].

Во время изменения каждого состояния список пересекается от начала до конца, и первый *item*, который соответствует текущему состоянию, будет использован для раскраски. Выбор не основывается на «наилучшем совпадении», берется просто первое подходящее по минимальным критериям состояние.

Ресурсы типа *Drawable*

«Рисуемые» (*drawable*) ресурсы составляют главную концепцию для графики, которая может быть «отрисована» на экране и которую можно запросить через такое *API* как *getDrawable(int)* или применить к другому *XML*-ресурсу с помощью атрибутов наподобие *android*:*drawable* и *android*:*icon* [35].

Есть несколько разных типов drawable-ресурсов:

- *Bitmap File* – растровая картинка, графический файл с расширением *.png*, *.jpg* или *.gif*. Создается с помощью класса *BitmapDrawable*;

- *Nine-Patch File* – это файл *PNG* с растягиваемыми регионами, используется для изменения размеров изображения, базируясь на его содержимом (.9.*png*). Создается с помощью класса *NinePatchDrawable*;

- *Layer List* – список слоев, специальный *drawable*-ресурс, который управляет массивом из других *drawable*-ресурсов. Этот список отрисовывается в порядке следования элементов в массиве, так что элемент с самым большим индексом будет нарисован на самом верху. Создается с помощью класса *LayerDrawable*;

- *State List* – список состояний, файл *XML*, который содержит ссылки на различные растровые (*bitmap*) графические изображения для разных состояний (например, может использоваться для анимации кнопки, когда она нажата). Создается с помощью класса *StateListDrawable*;

- *Level List* – список уровней, файл *XML*, определяющий *drawable*-ресурс, который управляет некоторым количеством альтернативных *Drawables*, каждому из которых присвоено максимальное числовое значение. Создается с помощью класса *LevelListDrawable*;

- *Transition Drawable* – файл *XML*, описывающий *drawable*, который может перетекать (*cross-fade*) между двумя *drawable*-ресурсами. Создается с помощью класса *TransitionDrawable*;

- *Inset Drawable* – файл *XML*, определяющий *drawable*, который будет вставлен в другой *drawable* с указанным расстоянием. Это полезно, когда для *View* (вид, представление) нужен рисунок заднего плана (*background drawble*), который меньше, чем реальные границы *View*;

- *Clip Drawable* – файл *XML*, определяющий *drawable*, который отсекает другой *drawable* на основе значения текущего уровня. Создается с помощью класса *ClipDrawable*;

- *Scale Drawable* – файл *XML*, задающий *drawable*, который меняет размер другого *Drawable* на основе значения его текущего уровня. Создается с помощью класса *ScaleDrawable*.

5.4 Работа с файловой системой *Android*

ОС *Android* построена на основе *Linux*. Этот факт находит свое отражение в работе с файлами. Так, в путях к файлам в качестве разграничителя в *Linux* использует слеш «/», а не обратный слеш «\» (как в *Windows*). А все названия

файлов и каталогов являются регистрозависимыми, т. е. *data* – это не то же самое, что и *Data* [36].

Приложение *Android* сохраняет свои данные в каталоге */data/data/<название_пакета>/* и, как правило, относительно этого каталога будет идти работа.

Для работы с файлами абстрактный класс android.content.Context определяет ряд методов [36]:

- *deleteFile(String name)*: удаляет определенный файл;
- *fileList()*: получает все файлы, которые содержатся в подкаталоге */files* в каталоге приложения;
- *getCacheDir()*: получает ссылку на подкаталог *cache* в каталоге приложения;
- *getDir(String dirName, int mode)*: получает ссылку на подкаталог в каталоге приложения; если такого подкаталога нет, то он создается;
- *getExternalCacheDir()*: получает ссылку на папку */cache* внешней файловой системы устройства;
- *getExternalFilesDir()*: получает ссылку на каталог */files* внешней файловой системы устройства;
- *getFileStreamPath(String filename)*: возвращает абсолютный путь к файлу в файловой системе;
- *openFileInput(String filename)*: открывает файл для чтения;
- *openFileOutput (String name, int mode)*: открывает файл для записи.

Все файлы, которые создаются и редактируются в приложении, как правило, хранятся в подкаталоге */files* в каталоге приложения.

Для непосредственного чтения и записи файлов применяются также стандартные классы *Java* из пакета *java.io*.

Система позволяет создавать файлы с двумя разными режимами: *MODE_PRIVATE* – файлы могут быть доступны только владельцу приложения (режим по умолчанию); *MODE_APPEND* – данные могут быть добавлены в конец файла.

5.5 Формат хранения данных *JSON*

JSON – текстовый формат обмена данными, основанный на *JavaScript*.

За счет своей лаконичности по сравнению с *XML* формат *JSON* может быть более подходящим для сериализации сложных структур. Если говорить о *web*-приложениях, то в таком ключе он уместен в задачах обмена данными как между браузером и сервером (*AJAX*), так и между самими серверами (программные *HTTP*-сопряжения) [37].

Поскольку формат *JSON* является подмножеством синтаксиса языка *JavaScript*, то он может быть быстро десериализован встроенной функцией *eval()*. Кроме того, возможна вставка вполне работоспособных *JavaScript*-функций. В языке *PHP*, начиная с версии 5.2.0, поддержка *JSON* включена в ядро в виде функций *json_decode()* и *json_encode()*, которые сами преобразуют типы данных *JSON* в соответствующие типы *PHP*, и наоборот.

JSON-текст представляет собой (в закодированном виде) одну из двух структур [37]:

1) набор пар «ключ:значение». В различных языках это реализовано как объект, запись, структура, словарь, хэш-таблица, список с ключом или ассоциативный массив. Ключом может быть только строка (регистрозависимая – имена с буквами в разных регистрах считаются разным), значением – любая форма;

2) упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

Это универсальные структуры данных, как правило, любой современный язык программирования поддерживает их в той или иной форме. Они легли в основу *JSON*, так как он используется для обмена данными между различными языками программирования [37].

В качестве значений в JSON могут быть использованы [37]:

- объект – это неупорядоченное множество пар «ключ:значение», заключенное в фигурные скобки «{ }». Ключ описывается строкой, между ним и значением стоит символ «:». Пары «ключ:значение» отделяются друг от друга запятыми;
- массив (одномерный) – это упорядоченное множество значений. Массив заключается в квадратные скобки «[ ]». Значения разделяются запятыми;
- число;
- литералы *true, false* и *null*;
- строка – это упорядоченное множество из нуля или более символов юникода, заключенное в двойные кавычки. Символы могут быть указаны с использованием *escape*-последовательностей, начинающихся с обратной косой черты «\» (поддерживаются варианты \", \\, \/, *t*, *n*, *r*, *f* и *b*), или записаны шестнадцатеричным кодом в кодировке *Unicode* в виде *uFFFF*.

Строка очень похожа на одноименный тип данных в языках *C* и *Java*. Число тоже очень похоже на *C*- или *Java*-число, за исключением того, что используется только двоичный формат. Пробелы могут быть вставлены между любыми двумя синтаксическими элементами [37].

Для работы с форматом *JSON* нет встроенных средств, но есть большое количество библиотек и пакетов, которые можно использовать для данной цели. Одним из наиболее популярных является пакет *com.google.code.gson* [37].

5.6 Подключение и использование сторонних библиотек

Android Studio позволяет подключить три типа библиотек в свой проект [38]:

1) из репозитория *Maven*;

2) библиотеку в виде собранного *.jar* файла;

3) библиотеку в виде исходных кодов.

Открываем окно структуры проекта (*File → Project Structure*) и переходим к основному модулю проекта – слева находится секция *Modules*, следует щелкнуть по названию модуля этого приложения (рисунок 5.8) [38].

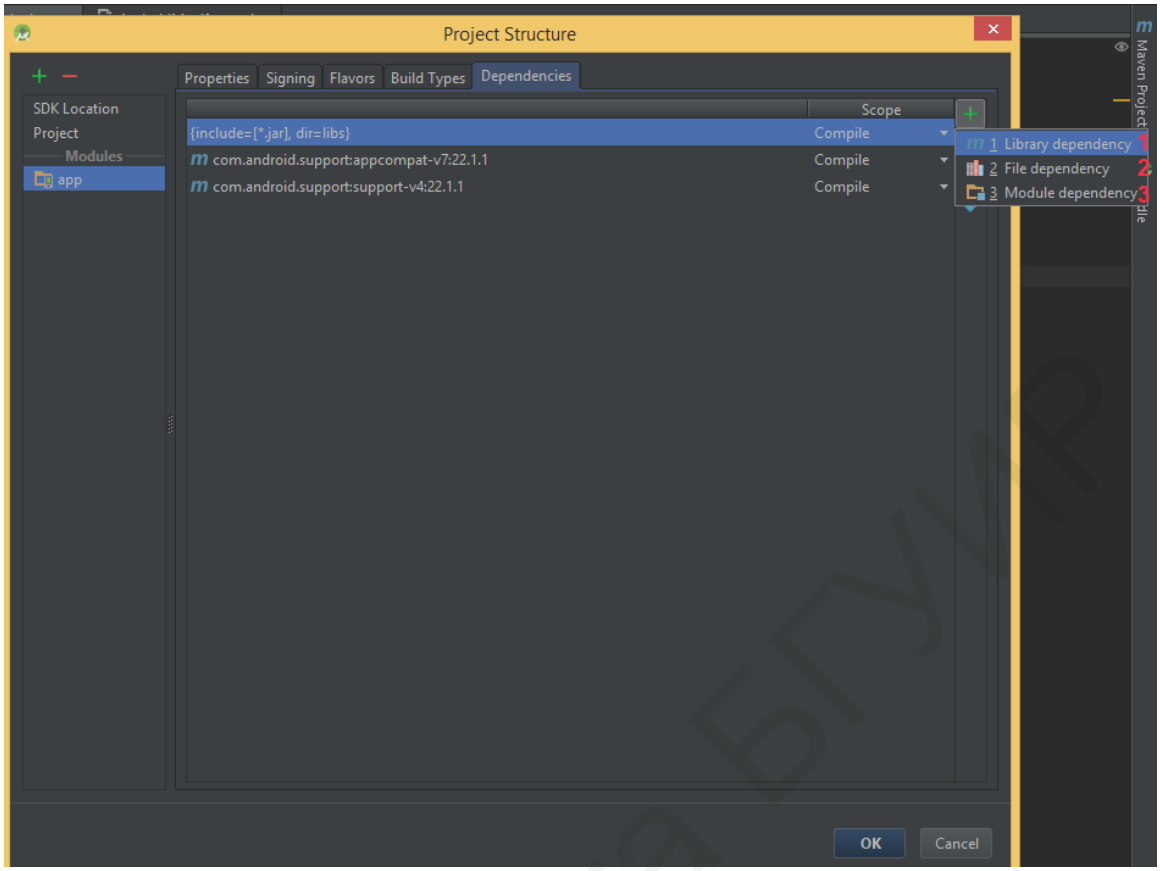


Рисунок 5.8 – Окно *Project Structure*

В открывшемся окне переходим к вкладке *Dependencies*. Здесь можно управлять подключенными зависимостями: добавить новые, удалить ненужные и перемещать их по иерархии.

Для того чтобы добавить зависимость, нажимаем «плюс» справа и видим три пункта. Рассмотрим их подробно [38].

1 *Library dependency* – добавление библиотеки из внешнего репозитория

Обычно по умолчанию в конфиге *gradle* всего проекта в качестве внешнего репозитория используется *JCenter*, и разработчики-профессионалы складывают свои труды туда. Но можно прописать в этом конфиге и другие репозитории с библиотеками, если таковые требуются.

При выборе пункта 1 появляется окно поиска (рисунок 5.9) [38], в котором можно набрать имя требуемой библиотеки, и, если она присутствует в репозитории, она отобразится. Нажатием кнопки *OK* подключаем библиотеку, и после того, как *gradle* проведет необходимые манипуляции по скачиванию и импорту, она будет доступна в проекте [38].

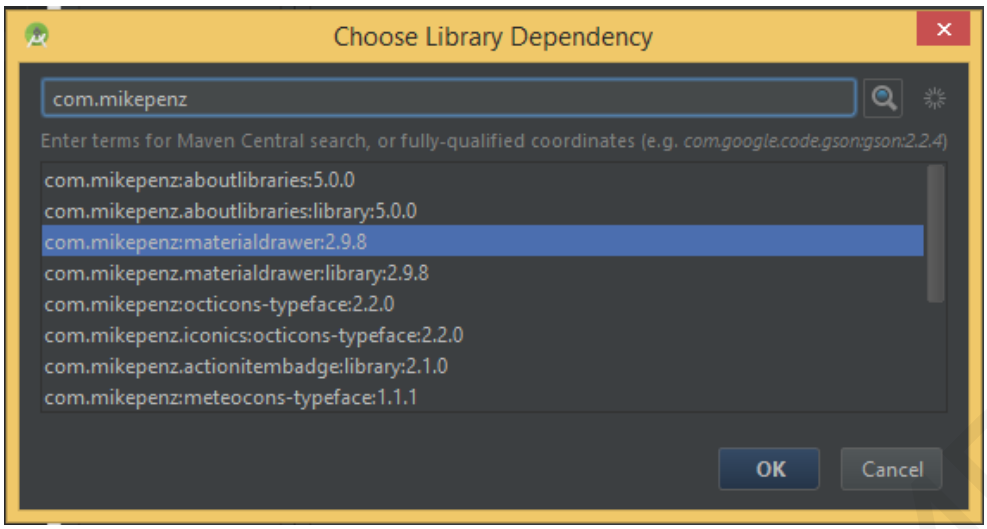


Рисунок 5.9 – Окно *Library dependency*

При добавлении библиотеки из репозитория вручную необходимо прописать в секции *dependencies*{} конфига *build.gradle* приложения ссылку на нужную библиотеку [38];

2 *File dependency* – добавление библиотеки из локального скомпилированного *jar*-файла

Перед началом импорта локальной библиотеки в виде *jar*-файла необходимо скопировать сам файл библиотеки в папку */libs* вашего проекта.

При выборе пункта 2 появится стандартный диалог выбора файла, в котором необходимо указать собственно сам файл подключаемой библиотеки (рисунок 5.10) [38].

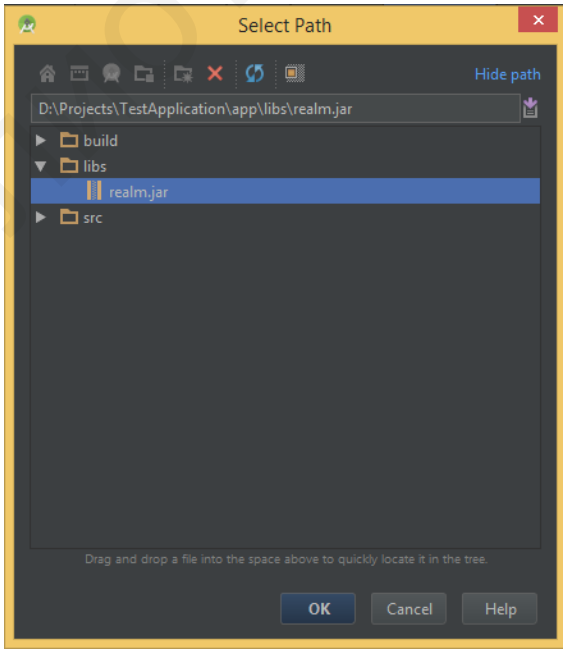


Рисунок 5.10 – Окно *File dependency*

При нажатии кнопки *OK gradle* проведет некоторые манипуляции по импорту, которые займут какое-то время, после чего библиотека будет доступна в проекте [38].

Для того чтобы вручную сделать то же самое – добавить библиотеку на основе *jar*-файла, в секции *dependencies*{} конфига *build.gradle* приложения необходимо прописать следующее [38]:

```
dependencies {
   compile files('libs/realm.jar')
}
```

Для того чтобы подключить все библиотеки из папки /libs/ разом, нужно сделать так [38]:

```
dependencies {
   compile fileTree(include: ['*.jar'], dir: 'libs')
}
```

3 *Module dependency* – добавление библиотеки из исходных кодов.

Перед импортом библиотеки из исходных кодов необходимо скачать сами исходные коды и распаковать их в какой-либо каталог.

Для избежания проблем при добавлении библиотек следует учесть [38]:

- исходные коды должны быть *gradle*-проектом *Android Studio/IntelliJ IDEA*;
- имена основного модуля и модуля подключаемой библиотеки не должны совпадать;
- подключаемый исходник является библиотекой, а не приложением (в *build.gradle указано: apply plugin: 'com.android.library'*), можно исправить самостоятельно в конфиге *gradle*;
- минимальные версии *SDK* вашего проекта и библиотеки должны совпадать, можно исправить самостоятельно в конфигах *gradle* проекта или библиотеки;
- версии *BuildTools* проекта и библиотеки должны совпадать (рекомендуется), либо обе версии должны быть установлены в вашем *SDK* (не рекомендуется). Лучшее решение – всегда использовать самую последнюю версию *BuildTools*.

Сначала импортируем нужную библиотеку в свой проект в качестве модуля. Нажимаем *File → New → Import Module*, указываем путь до корневой папки с разархивированными исходниками и следуем другим указаниям мастера импорта. Данный процесс отображен на рисунке 5.11 [38].

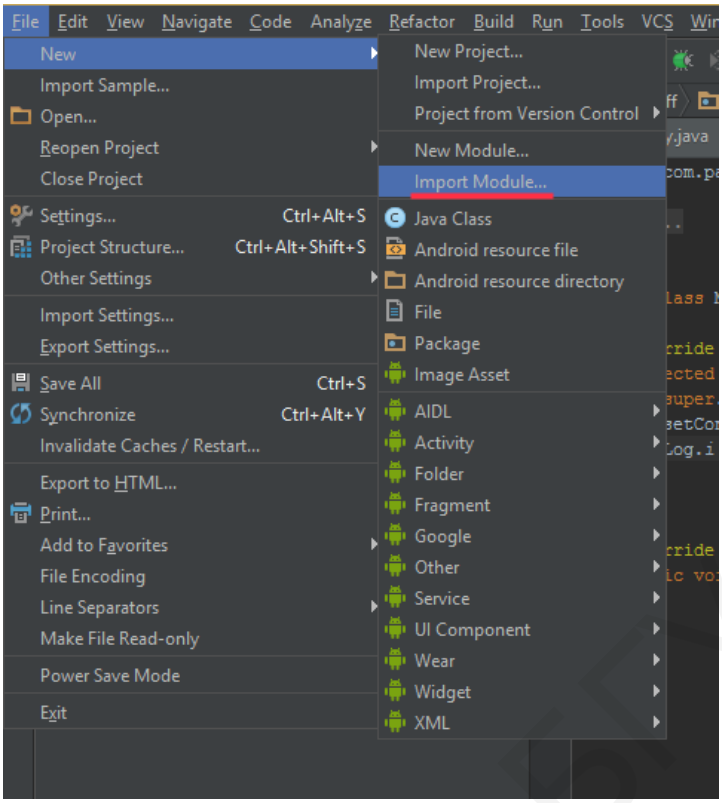


Рисунок 5.11 – Импорт библиотеки в качестве модуля

После выбираем пункт 3 – появляется окно выбора модуля, выбираем нужный вариант и нажимаем кнопку *OK*. Таким образом будет проведена работа по импорту в данный проект и библиотека становится доступной в проекте [38].

5.7 Манифест. Обзор файла *manifest.xml*

Файл манифеста *AndroidManifest.xml* предоставляет основную информацию о программе системе [39]. Каждое приложение должно иметь свой файл *AndroidManifest.xml*. Редактировать файл манифеста можно вручную, изменяя *XML*-код или через визуальный редактор *Manifest Editor* (редактор файла манифеста), который позволяет осуществлять визуальное и текстовое редактирование файла манифеста приложения [39].

Назначение манифеста [39]:

- объявляет имя *Java*-пакета приложения, который служит уникальным идентификатором;
- описывает компоненты приложения (деятельность, службы, приемники широковещательных намерений и контент-провайдеры, что позволяет вызывать классы, которые реализуют каждый из компонентов) и объявляет их намерения;
- содержит список необходимых разрешений для обращения к защищенным частям *API* и взаимодействия с другими приложениями;
- объявляет разрешения, которые сторонние приложения обязаны иметь для взаимодействия с компонентами данного приложения;

- объявляет минимальный уровень *API Android*, необходимый для работы приложения;

- перечисляет связанные библиотеки.

Файл манифеста инкапсулирует всю архитектуру *Android*-приложения, его функциональные возможности и конфигурацию. В процессе разработки приложения вам придется постоянно редактировать данный файл, изменяя его структуру и дополняя новыми элементами и атрибутами [39].

Рассмотрим структуру файла *AndroidManifest.xml file* [40].

<*manifest*> – корневой элемент, содержащий полное описание вашего пакета. В него могут включаться следующие элементы [40]:

<*uses-permission*> – описывает права, необходимые для того, чтобы ваша программа работала корректно. То есть, если вы в своей программе хотите использовать доступ к данным *GPS*, то в этой секции вы должны явно это указать, например: <*uses-permission android:name=»android.permission.ACCESS _GPS»* /> Манифест может вообще не содержать этот элемент [40];

<*permission*> – в этой секции описываются права, которые должны запросить другие приложения для доступа к вашему. Манифест может вообще не содержать этот элемент [40];

<*instrumentation*> – описывает код компонентов инструментария, доступный для тестирования функционала этого или другого приложений. Манифест может вообще не содержать этот элемент [40];

<*application*> – корневой элемент, содержащий описание компонентов уровня приложения доступных в пакете. Этот элемент может содержать глобальные и/или значения по умолчанию, такие как иконка программы, название, тема оформления, необходимые права доступа и т. д. Манифест может вообще не содержать этот элемент. Под ним также могут располагаться нуль или более других описаний [40];

<*activity*> – деятельность – это основной компонент приложения, взаимодействующий с пользователем. Первое окно, которое видят пользователи при запуске программы, – это и есть деятельность, и большинство других окон будут реализованы как отдельные деятельности, описанные тэгом <*activity*>. Каждая деятельность должна иметь собственный тэг <*activity*> в файле манифеста. Если деятельность не описана в манифесте, то нет возможности ее запускать. Подобное приложение вызовет ошибку [40]. Для поддержки позднего поиска своей деятельности следует включить один или более <*intent-filter*> элементов для описания действий, которые деятельность поддерживает [40];

<*intent-filter*> – описывается определенный тип значений намерений, которые компоненты поддерживают в качестве фильтров намерений. В дополнение, различные типы значений могут быть указаны под этим элементом. Атрибуты могут быть указаны для получения уникального названия, иконки или другой информации для действия, которое было описано [40];

<*action*> – действие намерений, которое компонент поддерживает [40];

<*category*> – категория намерений, которое поддерживает компонент [40];

<*data*> – так указываются поддерживаемые типы *Intent data MIME type, Intent data URI scheme, Intent data URI authority* или *Intent data URI path* [40];

<*meta-data*> – добавляет описание метаинформации к вашей деятельности, клиенты которой могут получить ее через *ComponentInfo.metaData* [40];

<*receiver*> – широковещательный приемник (*BroadcastReceiver*), который позволяет приложению узнавать о изменениях с данными или действиями, которые случились, даже если программа не запущена. Так же как и <*activity*>, вы можете указать один или более <*intent-filter*> или значения <*meta-data*>, которые получатель поддерживает [40];

<s*ervice*> – сервис – это компонент, который может быть запущен в фоне на произвольное количество времени. Так же как и в теге <*activity*>, опционально вы можете указать один или более <*intent-filter*> или <*meta-data*> элементов, которые поддерживает сервис [40];

<*provider*> – провайдер содержимого (*ContentProvider*) – это компонент, который управляет доступом к данным вашей программы, предоставляя его другим приложениям. Вы также можете указать один или более элементов <*meta-data*> [40].

5.8 Активности

Термин *Activity* еще не прижился в русском языке у разработчиков. Некоторые используют термин «активность», другие – «деятельность» (далее – активность) [41].

Разработчики старшего поколения воспринимают активность как форму. Простые приложения состоят из одной активности. Более сложные приложения могут иметь несколько окон, т. е. они состоят из нескольких активностей, которыми надо уметь управлять и которые могут взаимодействовать между собой [41].

Активность, которая запускается первой, считается главной. Из нее можно запустить другую активность. Причем не только относящуюся к данному приложению, но и из другого приложения. Пользователю будет казаться, что все запускаемые им активности являются частями одного приложения, хотя на самом деле они могут быть определены в разных приложениях и работают в разных процессах. Попробуйте воспринимать активности как страницы разных сайтов, открываемых в браузерах по ссылке.

Обычно активность занимает весь экран устройства, но это не является обязательным требованием. Можно создавать полупрозрачные и плавающие окна активностей. С развитием *Android* такой подход набирает обороты [41].

Чтобы создать активность, нужно «унаследоваться» от класса *Activity* и вызвать метод *onCreate()*. В результате мы получим пустой экран, что бесполезно. Поэтому в активность добавляют компоненты, фрагменты с помощью разметки.

Активность имеет жизненный цикл: начало, когда *Android* создает экземпляр активности, промежуточное состояние и конец, когда экземпляр уничтожается системой и освобождает ресурсы.

Активности свойственны три состояния [41]:

1) активная (*active* или *running*) – активность находится на переднем плане экрана. Пользователь может взаимодействовать с активным окном;

2) приостановленная (*paused*) – активность потеряла фокус, но все еще видима пользователю. То есть активность находится сверху и частично перекрывает данную активность. Приостановленная активность может быть уничтожена системой в критических ситуациях при нехватке памяти;

3) остановленная (*stopped*) – если данная активность полностью закрыта другой активностью. Она больше не видима пользователю и может быть уничтожена системой, если память необходима для более важного процесса.

Если активность, которая была уничтожена системой, нужно снова показать на экране, она должна быть полностью перезапущена и восстановлена в своем предыдущем состоянии.

Android SDK включает набор классов, наследованных от *Activity*. Они предназначены для упрощения работы с виджетами, которые часто встречаются в обычном пользовательском интерфейсе. Перечислим некоторые (наиболее полезные) из них [41]:

- *MapActivity* инкапсулирует обработку ресурсов, необходимых для поддержки элемента *MapView* внутри активности [41];

- *ListActivity* – обертка для класса *Activity*, главная особенность которой – виджет *ListView*, привязанный к источнику данных, и обработчики, срабатывающие при выборе элемента из списка [41];

- *ExpandableListActivity* – то же самое, что и *ListActivity*, но вместо *ListView* поддерживает *ExpandableListView* [41];

- *TabActivity* позволяет разместить несколько активностей или представлений в рамках одного экрана, используя вкладки для переключения между элементами [41].

5.9 Подключение виджетов

Виджеты представляют собой структурные элементы, из которых составляется пользовательский интерфейс. Виджет может выводить текст или графику, взаимодействовать с пользователем или размещать другие виджеты на экране. Кнопки, текстовые поля, флажки – все это разновидности виджетов.

Чтобы создать простейший виджет, понадобятся три детали [42]:

1) *Layout*-файл;

2) *XML*-файл с метаданными;

3) класс, наследующий *AppWidgetProvider*.

Layout-файл

В нем формируется внешний вид виджета. Все аналогично *layout*-файлам для *Activity* и фрагментов, только набор доступных компонентов здесь ограничен следующим списком [42]:

- *FrameLayout*;
- *LinearLayout*;
- *RelativeLayout*;
- *GridLayout*;
- *AnalogClock*;
- *Button*;
- *Chronometer*;
- *ImageButton*;
- *ImageView*;
- *ProgressBar*;
- *TextView*;
- *ViewFlipper*;
- *ListView*;
- *GridView*;
- *StackView*;
- *AdapterViewFlipper*;

XML-файл с метаданными

В нем задаются различные характеристики виджета. Ему свойственны следующие параметры [42]:

- *layout*-файл, чтобы виджет знал, как он будет выглядеть;
- размер виджета, чтобы виджет знал, сколько места он должен занять на экране;
- интервал обновления, чтобы система знала, как часто ей надо будет обновлять виджет.

Класс, наследующий AppWidgetProvider

В этом классе нам надо будет реализовать *Lifecycle*-методы виджета.

Приведем схему создания виджета и его *lifecycle*-методов. *Activity* нам не понадобится, поэтому следует отключить *Create Activity* в визарде создания нового проекта (убрать галочку) [42].

Создадим проект без *Activity*:
Project name: P1171_SimpleWidget
Build Target: Android 2.3.3
Application name: SimpleWidget
Package name: ru.startandroid.develop.p1171simplewidget

Добавим строки в **strings.xml**:
<string name=*"widget_name"*>My first widget</string>
<string name=*"widget_text"*>Text in widget</string>

Создаем layout-файл **widget.xml**:
<?xml version=*"1.0"* encoding=*"utf-8"*?>
<RelativeLayout

```
 xmlns:android=http://schemas.android.com/apk/res/android
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical">
<TextView
 android:id="@+id/tv"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:background="#6600ff00"
 android:gravity="center"
 android:text="@string/widget_text"
 android:textColor="#000"
 android:textSize="18sp">
</TextView>
</RelativeLayout>
```

RelativeLayout, а внутри зеленый *TextView* с текстом по центру. То есть виджет просто будет показывать текст на зеленом фоне [42].

Создаем файл метаданных ***res/xml/widget_metadata.xml***:

```
<appwidget-provider
 xmlns:android=http://schemas.android.com/apk/res/android
 android:initialLayout="@layout/widget"
 android:minHeight="40dp"
 android:minWidth="110dp"
 android:updatePeriodMillis="2400000">
</appwidget-provider>
```

В атрибуте ***initialLayout*** указываем *layout*-файл для виджета.

Атрибуты ***minHeight*** и ***minWidth*** содержат минимальные размеры виджета по высоте и ширине.

Есть определенный алгоритм расчета этих цифр. При размещении виджета экран делится на ячейки, и виджет занимает одну или несколько из этих ячеек по ширине и высоте. Чтобы конвертнуть ячейки в *dp*, используется формула $70 \cdot n - 30$, где n – это количество ячеек. То есть, если, например, хотим, чтобы виджет занимал две ячейки в ширину и одну в высоту, то высчитываем ширину – $70 \cdot 2 - 30 = 110$ – и высоту – $70 \cdot 1 - 30 = 40$. Эти полученные значения и будем использовать в атрибутах *minWidth* и *minHeight* [42].

Атрибут ***updatePeriodMillis*** содержит некоторое количество миллисекунд (мс). Это интервал обновления виджета. Тут можно указать интервал хоть от 5 с, но чаще чем один раз в 30 мин (1 800 000 раз), виджет обновляться все равно не будет – это системное ограничение. Для примера поставим интервал 40 мин (2 400 000 раз).

Приступаем к созданию класса, наследующего *AppWidgetProvider*.

```
package ru.startandroid.develop.p1171simplewidget;

import java.util.Arrays;

import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.Context;
import android.util.Log;

public class MyWidget extends AppWidgetProvider {

  final String LOG_TAG = "myLogs";

  @Override
  public void onEnabled(Context context) {
    super.onEnabled(context);
    Log.d(LOG_TAG, "onEnabled");
  }

  @Override
  public void onUpdate(Context context, AppWidgetManager appWidgetManager,
      int[] appWidgetIds) {
    super.onUpdate(context, appWidgetManager, appWidgetIds);
    Log.d(LOG_TAG, "onUpdate " + Arrays.toString(appWidgetIds));
  }

  @Override
  public void onDeleted(Context context, int[] appWidgetIds) {
    super.onDeleted(context, appWidgetIds);
    Log.d(LOG_TAG, "onDeleted " + Arrays.toString(appWidgetIds));
  }

  @Override
  public void onDisabled(Context context) {
    super.onDisabled(context);
    Log.d(LOG_TAG, "onDisabled");
  }

}
```

onEnabled вызывается системой при создании первого экземпляра виджета (мы ведь можем добавить в *Home* несколько экземпляров одного и того же виджета) [42].

onUpdate вызывается при обновлении виджета. На вход, кроме контекста, метод получает объект *AppWidgetManager* и список *ID*-экземпляров виджетов, которые обновляются. Именно этот метод обычно содержит код, который обновляет содержимое виджета. Для этого нужен будет *AppWidgetManager*, который мы получаем на вход [42].

onDeleted вызывается при удалении каждого экземпляра виджета. На вход, кроме контекста, метод получает список *ID*-экземпляров виджетов, которые удаляются [42].

onDisabled вызывается при удалении последнего экземпляра виджета.

Во всех методах выводим в лог одноименный текст и список *ID* для *onUpdate* и *onDeleted* [42].

В *onUpdate* создается код для какого-то обновления виджета. То есть, если виджет отображает, например, текущее время, то при очередном обновлении (вызове *onUpdate*) надо получать текущее время и передавать эту информацию в виджет для отображения [42].

Осталось немного подрисовать манифест (рисунок 5.12).

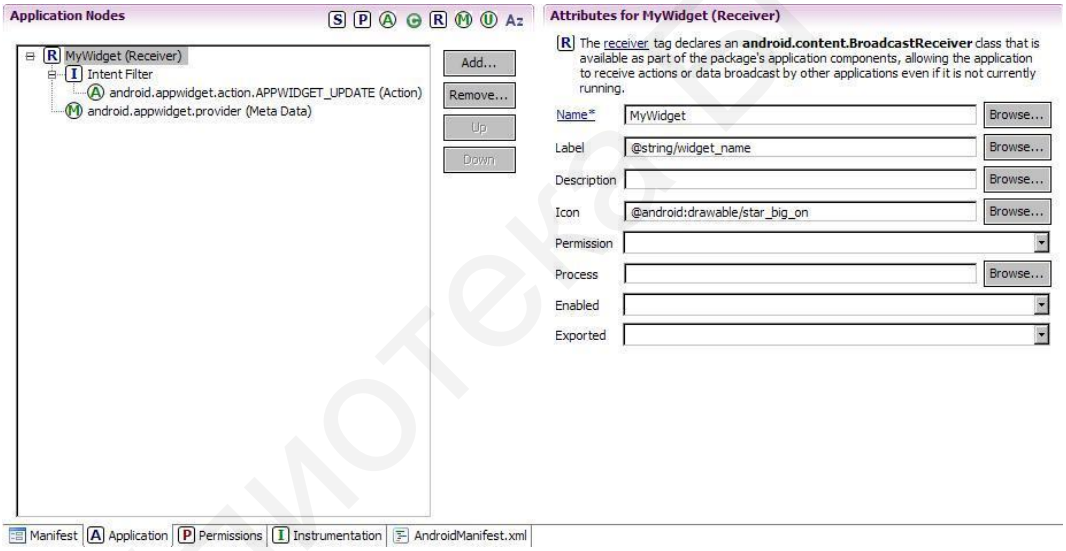


Рисунок 5.12 – Манифест

Добавьте туда класс *Receiver* [42]:

- укажите для него свои ***label*** и ***icon***. Этот текст и эту иконку вы увидите в списке выбираемых виджетов, когда будете добавлять виджет на экран;
- настройте для него фильтр с ***action*** = ***android.appwidget.action.APPWIDGET_UPDATE;***
- добавьте метаданные с именем ***android.appwidget.provider*** и указанием файла метаданных ***xml/widget_metadata.xml*** в качестве ресурса.

После этого в манифесте должна получиться примерно такая секция *Receiver*:

```
<receiver
 android:name="MyWidget"
 android:icon="@android:drawable/star_big_on"
 android:label="@string/widget_name">
<intent-filter>
<action
 android:name="android.appwidget.action.APPWIDGET_UPDATE">
</action>
</intent-filter>
<meta-data
 android:name="android.appwidget.provider"
 android:resource="@xml/widget_metadata">
</meta-data>
</receiver>
```

Виджет готов. Все сохраняем и запускаем. Никаких *Activity*, разумеется, не всплывет. В консоли должен появиться текст:

```
\P1171_SimpleWidget\bin\P1171_SimpleWidget.apk installed on device
Done!
```

Открываем диалог создания виджета и находим в списке наш виджет с иконкой и текстом, которые указывали в манифесте для *Receiver* (рисунок 5.13).

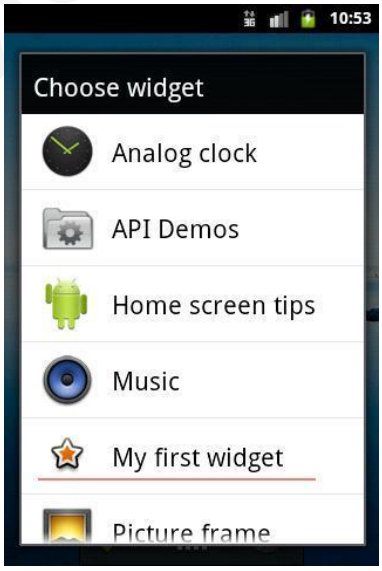


Рисунок 5.13 – Список виджетов

Выбираем его и добавляем на экран (рисунок 5.14).

Рисунок 5.14 – Виджет на рабочем столе

Виджет появился. Смотрим логи:

onEnabled
onUpdate [8]

Сработал ***onEnabled***, так как добавили **первый** экземпляр виджета. И сразу после этого добавления сработал метод ***onUpdate*** для конкретного экземпляра. Видим, что ему назначен *ID* = 8. То есть система добавила экземпляр виджета на экран и вызвала метод обновления с указанием *ID*-экземпляра.

Добавим еще один экземпляр (рисунок 5.15).

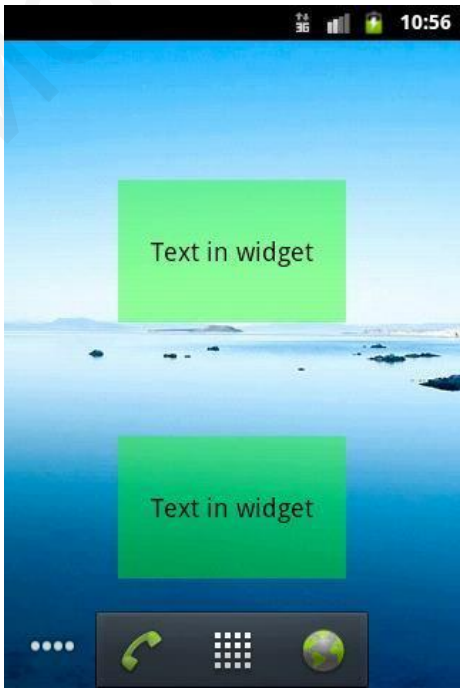


Рисунок 5.15 – Добавление второго виджета на рабочий стол

Смотрим лог:

onUpdate [9]

onEnabled не сработал, так как добавляемый экземпляр виджета уже **не первый**. ***onUpdate*** же снова отработал для нового добавленного экземпляра и получил на вход *ID* = 9.

Теперь рассмотрим удаление с экрана двух этих экземпляров виджета. Сначала удаляется последний. В логах увидим:

onDeleted [9]

Сработал ***onDeleted*** и получил на вход *ID* удаляемого экземпляра виджета. Удаляем первый экземпляр. В логах отображается:

onDeleted [8]
onDisabled

Снова сработал ***onDeleted***: нас оповестили, что экземпляр виджета с *ID* = 8 был удален. И сработал ***onDisabled***, т. е. был удален **последний** экземпляр виджета, больше работающих экземпляров не осталось.

Виджет обновляется (получает вызов метода *onUpdate*) один раз в 40 мин. Добавьте снова пару виджетов на экран и подождите. Когда они снова обновятся, в логах это отразится.

Так создается и работает простой виджет. Обратим внимание на некоторые нюансы.

Класс *AppWidgetProvider* является расширением класса *BroadcastReceiver* (в манифесте мы его и прописали как *Receiver*). Он просто получает от системы сообщение в *onReceive*, определяет по значениям из *Intent*, какое именно событие произошло (добавление, удаление или обновление виджета), и вызывает соответствующий метод (*onEnabled*, *onUpdate* и пр.) [42].

В манифесте для *Receiver*-класса настроили фильтр с *action*, который ловит события *update*. Каким же образом этот *Receiver* ловит остальные события (например, удаление)? Служба помощи пишет об этом так [42]:

The <intent-filter> element must include an <action> element with the android:name attribute. This attribute specifies that the AppWidgetProvider accepts the ACTION_APPWIDGET_UPDATE broadcast. This is the only broadcast that you must explicitly declare. The AppWidgetManager automatically sends all other App Widget broadcasts to the AppWidgetProvider as necessary.

То есть *ACTION_APPWIDGET_UPDATE* – это единственный *action*, который необходимо прописать явно. Остальные события *AppWidgetManager* каким-то образом сам доставит до нашего *AppWidgetProvider*-наследника [42].

Если мы расположим рядом несколько экземпляров виджета, увидим следующее его состояние, представленное на рисунке 5.16.

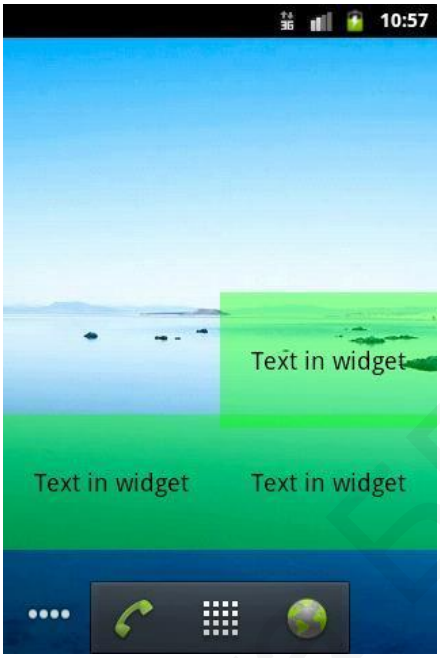


Рисунок 5.16 – Наложение виджетов

Делаем вывод, что надо бы сделать отступы.

Добавим *android:padding="8dp"* к *RelativeLayout* в нашем *layout*-файле (рисунок 5.17).

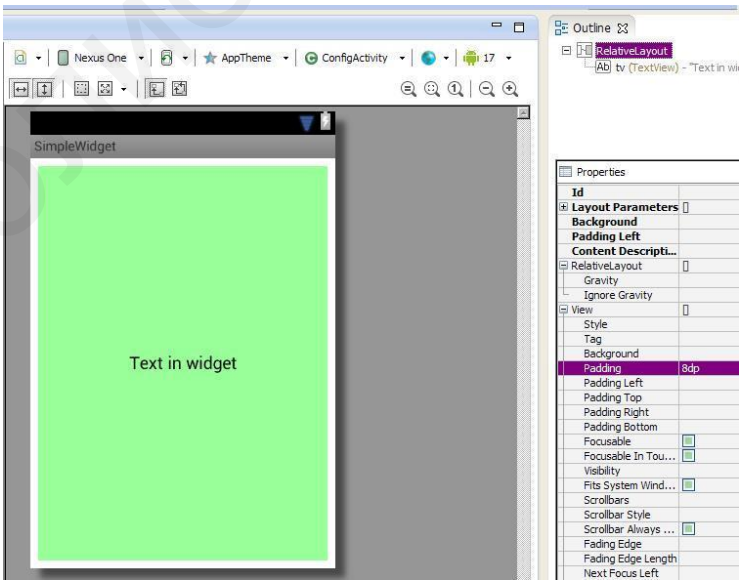


Рисунок 5.17 – Добавление отступов

Сохраняем, запускаем.

Виджеты на экране поменялись автоматически и теперь отображаются корректно (рисунок 5.18).

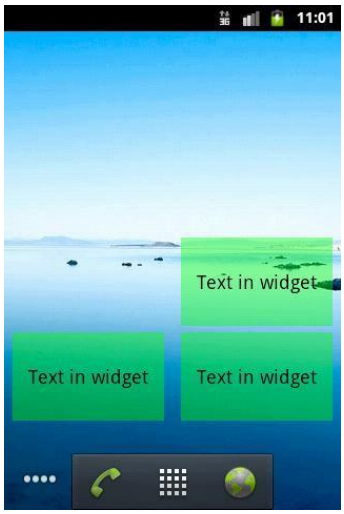


Рисунок 5.18 – Проверка работы виджетов с отступами

Помним, что для них сработал *onUpdate,* и это зафиксировано в логах. В метод был передан массив *ID* всех работающих экземпляров виджета.

В *Android* 4.0 (*API Level* 14) и более поздних версиях это неудобство с отступами было устранено, и необходимость делать отступы вручную отпала [42].

Давайте проверим. Уберите ранее добавленный в *RelativeLayout* отступ и укажите в манифесте ***android:targetSdkVersion*** версию 14 (или выше), чтобы система знала, что можно использовать стандартные возможности, а не режим совместимости.

Все сохраняем, запускаем виджет на эмуляторе с *Android* 4.0. Добавим три экземпляра (рисунок 5.19).

Рисунок 5.19 – Эмуляция виджета в *Android* 4.0

Так, система сама делает отступы между виджетами.

Получается, что для версий ниже 4 надо делать отступ в *layout*, а для старших версий – не надо. Служба помощи дает подсказку, как сделать так, чтобы ваш виджет корректно работал со всеми версиями. Для этого используются квалификаторы версий.

В *layout* для *RelativeLayuot* указываете:

android:padding="@dimen/widget_margin"

И создаете два файла:

res/values/dimens.xml с записью
<dimen name=*"widget_margin"*>8dp</dimen>
res/values-v14/dimens.xml с записью:
<dimen name=*"widget_margin"*>0dp</dimen>

В манифесте ***android:targetSdkVersion*** должна быть версия 14 или выше.

Таким образом, на старых версиях (без системного отступа) отступ будет 8 *dp*, а на новых – 0 *dp,* и останется только системный отступ.

5.10 Выделение контроллеров, моделей и элементов представления приложения

Не существует универсально уникального шаблона *Model – View – Controller* (*MVC*). *MVC* представляет собой концепцию, а не прочную структуру программирования. Реализовать свой собственный *MVC* можно на любой платформе [43].

Достоинства MVC [43]:

- когда нам приходится выделять крупные проекты в разработке программного обеспечения, обычно используется *MVC*, потому что это универсальный способ организации проектов;
- новые разработчики могут быстро адаптироваться к проекту;
- шаблон помогает в разработке больших проектов и кроссплатформенных проектов.

Структура MVC в основном такова [43]:

- модель: что показывать. Это может быть источник данных (например, сервер, исходные данные в приложении);
- вид: как он отображается. Это может быть *xml*. Таким образом, он действует как фильтр представления. Представление привязано к его модели (или части модели) и получает данные, необходимые для презентации;
- контроллер: обработка событий, таких как ввод данных пользователем. Это будет деятельность.

Важная особенность *MVC – мы можем изменить структурные элементы*: *либо модель, либо представление, либо контроллер (все еще не влияющие на другие)* [43].

Так, изменение цвета в представлении, размере вида или позиции вида не повлияет на модель или контроллер.

Изменение модели (вместо данных, полученных с сервера, извлекаем данные из активов) не повлияет на представление и контроллер.

Изменение контроллера (логики в активности) не повлияет на модель и представление.

Реализация структуры *MVC* делает приложение сложнее и не очень хорошо сочетается с другими частями *Android*.

При применении *MVC, MVVM* или *Presentation Model* для *Android*-приложения создается четкий структурированный проект для модульных тестов – что еще более важно.

Необходимо запускать тесты для *Android*, а не обычные тесты *JUnit*, которые требуют больше времени для запуска и делают блок-тесты несколько непрактичными. Для этого можно использовать *RoboBinding* – привязанную к данным платформу *Presentation Model* для платформы *Android*.

RoboBinding помогает вам писать код пользовательского интерфейса, который легче читать, тестировать и поддерживать. *RoboBinding* устраняет необходимость в ненужном коде, таком как *addXXListener*, или «сдвигает» логику пользовательского интерфейса на модель представления, которая является *POJO* и может быть протестирована с помощью обычных тестов *JUnit*. *RoboBinding* поставляется с более чем 300 тестов *JUnit* для обеспечения его качества [43].

5.11 Задания к лабораторной работе №5

1 Изучите теоретическую часть.
2 Согласно варианту, полученному у преподавателя, создайте виджет.
3 Проведите эмуляцию работы виджета в *Android Studio.*
4 Оформите отчет по лабораторной работе.

5.12 Контрольные вопросы

1 Дайте определение понятию «эмулятор».
2 Назовите основные этапы установки эмулятора *Android.*
3 Перечислите виды эмуляторов. Назовите их достоинства и недостатки.
4 Перечислите основные ресурсы *Android.*
5 Назовите методы для работы с файлами.
6 Охарактеризуйте формат хранения данных *JSON*.
7 Назовите основные этапы подключения библиотек в *Android Studio.*
8 Назовите назначение файла *manifest.xml* и охарактеризуйте его.
9 Дайте определение понятию «активность».
10 Опишите выделение контроллеров, моделей и элементов представления приложения.

Лабораторная работа №6. Архитектура приложений. Вспомогательные библиотеки

Цель: изучить особенности архитектуры приложений для операционной системы *Android*; выделить особенности операционной *Android* как платформы для разработки приложений.

6.1 Обзор архитектуры приложения *Android*

Экосистема средств разработки под *Android* развивается очень быстро. Каждую неделю кто-то создает новые инструменты, обновляет существующие библиотеки, пишет новые статьи или выступает с докладами [44].

В 2012 г. структура проектов выглядела очень просто. Не было никаких библиотек для работы с сетью, и только *AsyncTask* оказывал помощь [44]. Рисунок 6.1 показывает примерную архитектуру тех решений.

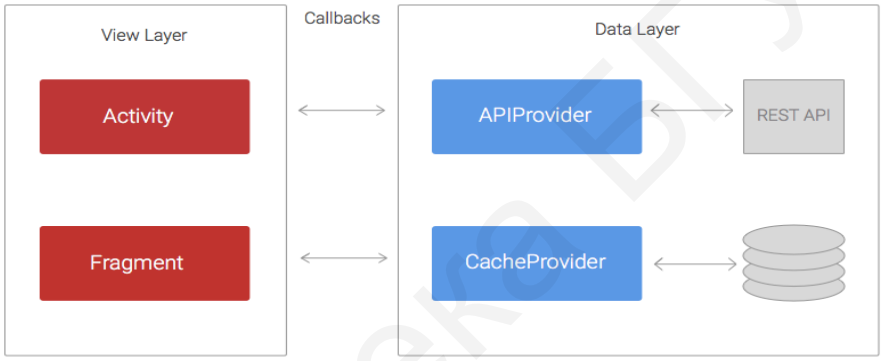


Рисунок 6.1 – Архитектура проектов в 2012 г.

Код был разделен на два уровня: уровень данных (*data layer*), который отвечал за получение/сохранение данных, получаемых как через *REST API*, так и через различные локальные хранилища, и уровень представления (*view layer*), отвечающий за обработку и отображение данных [44].

APIProvider предоставляет методы, позволяющие *Activity* и *Fragment* взаимодействовать с *REST API*. Эти методы используют *URLConnection* и *AsyncTask*, чтобы выполнить запрос в фоновом потоке, а потом доставляют результаты в *Activity* через функции обратного вызова. Аналогично работает и *CacheProvider*: есть методы, которые достают данные из *SharedPreferences* или *SQLite*, и есть функции обратного вызова, которые возвращают результаты [44].

Главная проблема такого подхода состоит в том, что уровень представления имеет слишком много ответственности. Представим простой сценарий, в котором приложение должно загрузить список постов из блога, закэшировать их в *SQLite*, а потом отобразить в *ListView*. *Activity* должна выполнить следующее [44]:

1 Вызвать метод *APIProvider#loadPosts*(*Callback*).

2 Подождать вызов метода *onSuccess*() в переданном *Callback'e* и потом вызвать *CacheProvider#savePosts*(*Callback*).

3 Подождать вызов метода *onSuccess*() в переданном *Callback'e* и потом отобразить данные в *ListView*.

4 Отдельно обработать две возможные ошибки, которые могут возникнуть как в *APIProvider*, так и в *CacheProvider*.

И это еще простой пример. На практике может случиться так, что *API* вернет данные не в том виде, в котором их ожидает наш уровень представления, а значит, *Activity* должна будет как-то трансформировать и/или отфильтровать данные прежде, чем сможет с ними работать. Или, например, *loadPosts*() будет принимать аргумент, который нужно откуда-то получить (например, адрес электронной почты, который запросим через *Play Services SDK*). Наверняка *SDK* будет возвращать адрес асинхронно, через функцию обратного вызова, а значит, теперь есть три уровня вложения функций обратного вызова [44].

Ситуация начала меняться в 2014 г., когда вышла версия *RxJava*. Решение проблемы вложенных функций обратного вызова, похоже, найдено [44]. Если коротко, *RxJava* позволяет управлять данными через асинхронные потоки (прим. переводчика: в данном случае имеются в виду потоки как *streams*, не путать с *threads* – потоками выполнения), и предоставляет множество операторов, которые можно применять к потокам, чтобы трансформировать, фильтровать, или же комбинировать данные так, как нужно.

Приняв во внимание все проблемы, с которыми столкнулись за два прошедших года, была продумана архитектура нового приложения, представленная на рисунке 6.2 [44].

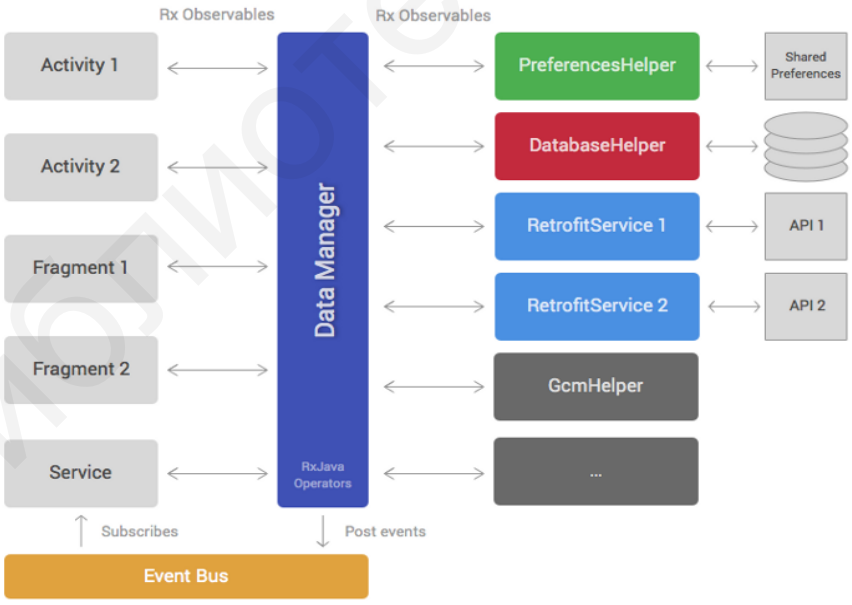


Рисунок 6.2 – Архитектура нового приложения

Код все так же разделен на два уровня: на уровень данных, который содержит *DataManager* и набор классов-помощников, и уровень представления, который состоит из классов *Android SDK*, таких как *Activity*, *Fragment*, *ViewGroup* и т. д.

Классы-помощники (третья колонка в диаграмме) имеют очень ограниченные области ответственности, и реализуют их в определенной последовательности. Например, большинство проектов имеют классы для доступа к *REST API*, чтения данных из базы данных (БД) или взаимодействия с *SDK* от сторонних производителей. У разных приложений будет разный набор классов-помощников, но наиболее часто используемыми будут следующие [44]:

- *PreferencesHelper*: работает с данными в *SharedPreferences*;
- *DatabaseHelper*: работает с *SQLite*;
- сервисы *Retrofit*, выполняющие обращения к *REST API*. Использование *Retrofit* вместо *Volley* предпочтительнее, потому что он поддерживает работу с *RxJava*, причем *API* у него лучше.

Многие методы классов-помощников возвращают *RxJava Observables* [44].

DataManager является центральной частью новой архитектуры. Он широко использует операторы *RxJava* для того, чтобы комбинировать, фильтровать и трансформировать данные, полученные от помощников. Задача *DataManager* состоит в том, чтобы освободить *Activity* и *Fragment* от работы по «причесыванию» данных – он будет производить все нужные трансформации внутри себя и отдавать наружу данные, готовые к отображению.

Компоненты уровня представления будут просто вызывать этот метод и подписываться на возвращенный им *Observable*. Как только подписка завершится, посты, возвращенные полученным *Observable*, могут быть добавлены в *Adapter*, чтобы отобразить их в *RecyclerView* или чем-то подобном [44].

Последний элемент этой архитектуры – это *event bus*. *Event bus* позволяет нам запускать сообщения о неких событиях, происходящих на уровне данных, а компоненты, находящиеся на уровне представления, могут подписываться на эти сообщения. Например, метод *signOut*() в *DataManager* может запустить сообщение, оповещающее о том, что соответствующий *Observable* завершил свою работу, и тогда *Activity*, подписанные на это событие, могут перерисовать свой интерфейс, чтобы показать, что пользователь вышел из системы [44].

Достоинства данного подхода следующие [44]:

- *Observables* и операторы из *RxJava* избавляют от вложенных функций обратного вызова;
- *DataManager* берет на себя работу, которая ранее выполнялась на уровне представления, разгружая таким образом *Activity* и *Fragment*;
- перемещение части кода в *DataManager* и классы-помощники делает *unit*-тестирование *Activity* и *Fragment* более простым;
- ясное разделение ответственности и выделение *DataManager* как единственной точки взаимодействия с уровнем данных делает всю архитектуру адаптированной к тестированию. Классы-помощники, или *DataManager*, могут быть легко подменены на специальные заглушки.

В то же время у подхода наблюдаются недостатки [44]:

- в больших и сложных проектах *DataManager* может стать слишком «раздутым», и поддержка его существенно затруднится;

- хотя компоненты уровня представления (такие как *Activity* и *Fragment*) и были сделаны более легковесными, они все еще содержат заметное количество «логики, крутящейся около управления подписками *RxJava*, анализа ошибок и пр.».

В *Android*-сообществе начали набирать популярность отдельные архитектурные шаблоны, такие как *MVP*, или *MVVM* (рисунок 6.3 [44]). Было обнаружено, что *MVP* может привнести значимые изменения в архитектуру проектов. Так как код уже разделили на два уровня (данных и представления), введение *MVP* выглядело натурально. Необходимо было добавить новый уровень *presenter* и перенести в него часть кода из представлений.

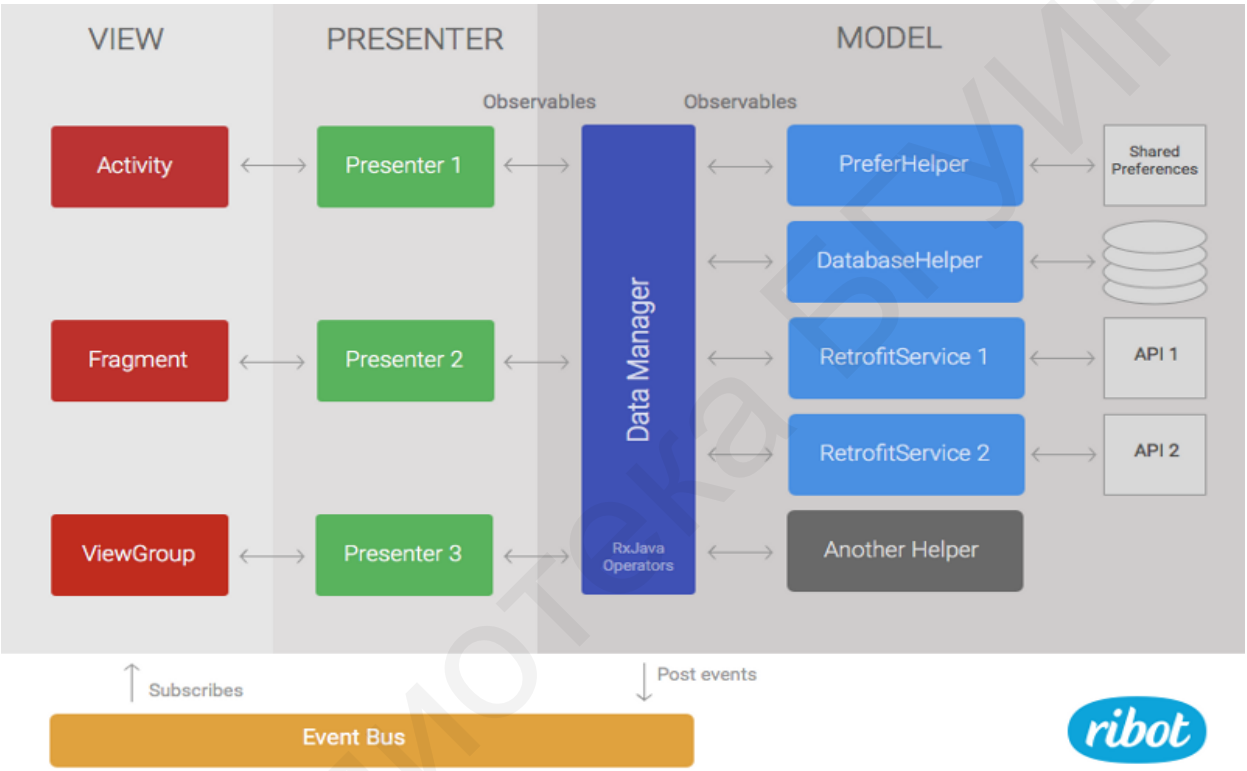


Рисунок 6.3 – Архитектурный шаблон *MVP*

Уровень данных остается неизменным, но теперь он называется моделью, чтобы соответствовать имени соответствующего уровня из *MVP* [44].

Presenter отвечают за загрузку данных из модели и вызов соответствующих методов на уровне представления, когда данные загружены. *Presenter* подписываются на *Observables*, возвращаемые *DataManager*. Следовательно, они должны работать с такими сущностями, как подписки и планировщики. Более того, они могут анализировать возникающие ошибки или применять дополнительные операторы к потокам данных, если необходимо. Например, если нужно отфильтровать некоторые данные и этот фильтр скорее всего нигде больше использоваться не будет, есть смысл вынести этот фильтр на уровень *presenter*, а не *DataManager* [44].

Как и в предыдущей архитектуре, уровень представления содержит стандартные компоненты из *Android SDK*. Разница в том, что теперь эти компоненты не подписываются напрямую на *Observables*. Вместо этого они имплементируют интерфейс *MvpView* и предоставляют список внятных и понятных методов, таких как *showError*() или *showProgressIndicator*(). Компоненты уровня представления отвечают также за обработку взаимодействия с пользователем (например, события нажатия) и вызов соответствующих методов в *presenter*. Например, если у нас есть кнопка, которая загружает список постов, *Activity* должна будет вызвать в *OnClickListener'e* метод *presenter.loadTodayPosts*() [44].

Достоинства данного подхода следующие [44]:

- *Activity* и *Fragment* становятся еще более легковесными, так как их работа сводится теперь к отрисовке/обновлению пользовательского интерфейса и обработке событий взаимодействия с пользователем;
- писать *unit*-тесты для *presenter* очень легко – нужно просто замокировать уровень представления. Раньше этот код был частью уровня представления, и провести его *unit*-тестирование не представлялось возможным. Архитектура становится еще легче тестируемой;
- если *DataManager* становится слишком «раздутым», то мы всегда можем перенести часть кода в *presenter*.

Уникальной архитектуры не существует по сей день. Однако экосистема *Android* развивается очень быстро.

6.2 Подключение и использование сторонних библиотек

Очень часто при создании приложения приходится пользоваться сторонними библиотеками при решении тех или иных задач. Но не все начинающие *Android*-разработчики умеют подключать их к своему проекту. Рассмотрим, как подключаются библиотеки и какие при этом возникают ошибки. В качестве *IDE* будет использоваться *Eclipse* [45].

Разберем три случая [45]:

- подключение *jar*;
- подключение проекта с исходниками;
- что делать, если пункт 2 не сработал.

Случай первый: библиотека поставляется в jar-файле [45]

Это самый легкий вариант подключения библиотеки. На сегодняшний день не очень распространен и, скорее всего, будет терять популярность. Одной из причин является невозможность «запаковки» ресурсов вместе с библиотекой. Не так давно появился новый формат библиотек – *aar*. Это тот же *jar*-файл, только с ресурсами.

Тут тоже есть два способа подключения *jar*-библиотеки к проекту.

Первый способ [45]:

1 Раскройте дерево проекта в *Package Explorer*.

2 Перетащите в папку *libs* ваш *jar*-файл (если папки *libs* нет, то создайте ее.

3 В открывшемся окне выберите пункт *Copy files* и нажмите кнопку *OK*.

4 Нажмите правой кнопкой мыши на вашем проекте и выберите пункт *Refres*.

Второй способ [45]:

1 Нажмите правой кнопкой мыши на вашем проекте и выберите пункт *Properties*.

2 В открывшемся окне слева есть меню, выберите в нем пункт *Java Build Path*.

3 Выберите вкладку *Libraries*.

4 Нажмите кнопку *Add External JARs…*

5 В открывшемся окне выберите файл с вашей библиотекой и нажмите *OK*.

6 Нажмите кнопку *OK*.

Случай второй: библиотека поставляется в виде проекта [45]

1 Откройте *File > Import*, выберите *Android > Existing Android Code into Workspace*.

2 Нажмите кнопку *Browse*.

3 В открывшемся окне найдите папку с библиотекой, нажмите *OK*.

4 Поставьте галочку на пункте *Copy projects into workspace*.

5 Нажмите *Finish*.

6 Нажмите правой кнопкой мыши на вашем проекте и выберите пункт *Properties*.

7 В открывшемся окне слева есть меню, выберите в нем пункт *Android*.

8 Нажмите кнопку *Add*.

9 В открывшемся окне выберите вашу библиотеку и нажмите *OK*.

Третий случай: при импортировании библиотеки Eclipse выдает ошибку [45]

Если при импортировании библиотеки в *workspase Eclipse* выдает ошибку типа «…» или какую-нибудь еще, то выполняем следующие действия [45]:

1 Выберите *File > New > Android Application Project*.

2 В полях *Application Name* и *Project Name* напишите названия библиотеки, например, *PullToRefreshLibrary*.

3 Нажмите *Next*.

4 Уберите галочку с пунктов *Create custom launcher icon* и *Create activity*. Поставьте галочку на пункте *Mark this project as a library*. Нажмите *Finish*.

5 Удалите папки *src*, *libs* и *res*, а также файл *AndroidManifest.xml*.

6 Перетащите папки *src*, *libs*, *res* и *AndroidManifest.xml* из папки библиотеки в папку проекта.

7 В открывшемся окне выберите пункт *Copy files* and *folders*.

8 Выполните действия из второго случая, начиная с пункта 6.

Операционная система *Android* предоставляет мощный фундамент для разработки приложений, которые отлично работают на множестве разнообразных

устройств и форм-факторов. Трудно создавать «идеальные» приложения в условиях сложных циклов «жизни» объектов и отсутствия рекомендованной архитектуры приложения [45].

6.3 Задания к лабораторной работе №6

1 Изучите теоретическую часть.

2 К одному из ранее созданных проектов подключите сторонние библиотеки, использовав способы, представленные в пункте 6.2. Воспользуйтесь библиотеками из открытых источников сети Интернет.

3 Сделайте вывод об удобности данных методов.

4 Оформите отчет по лабораторной работе.

6.4 Контрольные вопросы

1 Охарактеризуйте архитектуру *Android*-приложения.

2 Перечислите классы-помощники.

3 Дайте характеристику архитектуре *MVP*.

4 Перечислите основные этапы подключения сторонних библиотек в проект.

Лабораторная работа №7. Локальный сервер *OpenServer*. База данных *MySQL*

Цель: изучить программное обеспечение *OpenServer*; рассмотреть принцип использования базы данных *MySQL* для серверных задач; изучить управление базой *MySQL* данных через *phpMyAdmin*.

7.1 Установка и настройка базы данных *MySQL*

Сервер баз данных *MySQL* является одним из наиболее популярных среди серверов баз данных с открытым исходным кодом, используемых при разработке *web*-приложений.

Рассмотрим последовательность действий по настройке сервера базы данных *MySQL* версии 5.6 в ОС *Windows*. Сведения о конфигурации *MySQL* не рассматриваются, приводится только последовательность необходимых шагов [46].

Начало загрузки

Последнюю версию *MySQL* можно скачать на официальном сайте. Перейдя в раздел *Downloads* и выбрав необходимый продукт, скачиваем его на персональный компьютер [46].

Начало установки

После завершения загрузки запустите программу установки следующим образом [46]:

1 Щелкните правой кнопкой мыши загруженный установочный файл и выберите пункт «Выполнить» – запустится программа установки *MySQL*.

2 На панели приветствия выберите «Установить продукты *MySQL*».

3 На панели информации о лицензии ознакомьтесь с лицензионным соглашением, установите флажок принятия и нажмите кнопку «Далее».

4 На панели «Найти последние продукты» нажмите кнопку «Выполнить». После завершения операции нажмите кнопку «Далее».

5 На панели «Тип настройки» выберите параметр «Пользовательская», а затем нажмите кнопку «Далее».

6 На панели «Выбор компонентов обеспечения» убедитесь, что выбран *MySQL Server 5.6.x,* и нажмите кнопку «Далее».

7 На панели «Проверить требования» нажмите кнопку «Далее».

8 На панели «Установка» нажмите кнопку «Выполнить». После успешного завершения установки сервера на панели «Установка» отображается информационное сообщение, нажмите кнопку «Далее».

9 На странице «Настройка» нажмите кнопку «Далее».

10 На первой странице конфигурации сервера *MySQL* (1/3) установите следующие параметры [46]:

а) **тип конфигурации сервера**. Выберите вариант «Компьютер для разработки»;

б) **включите поддержку сети *TCP/IP***. Убедитесь, что флажок установлен, и задайте следующие параметры ниже:

- **номер порта**. Укажите порт подключения. По умолчанию установлено значение 3306; не следует изменять его без необходимости;
- **откройте порт брандмауэра для доступа к сети**. Выберите исключение добавления брандмауэра для указанного порта;
- **расширенная настройка.** Выберите флажок «Показать расширенные параметры» для отображения дополнительной страницы конфигурации для настройки расширенных параметров для экземпляра сервера (если требуется).

Примечание – При выборе этого параметра необходимо перейти к панели для установки параметров сети, где будет отключен брандмауэр для порта, используемого сервером *MySQL*.

11 Нажмите кнопку «Далее».

12 На второй странице конфигурации сервера *MySQL* (2/3) установите следующие параметры [46]:

а) **пароль учетной записи *root***;

б) **пароль *root* для *MySQL***. Введите пароль пользователя *root*;

в) **повторите ввод пароля**. Повторно введите пароль пользователя **root**.

Примечание – Пользователь *root* – это пользователь, который имеет полный доступ к серверу баз данных *MySQL* – создание, обновление и удаление пользователей и т. д. Запомните пароль пользователя *root* (администратора) – он понадобится вам при создании примера базы данных;

г) **учетные записи пользователя *MySQL***. Нажмите кнопку «Добавить пользователя» для создания учетной записи пользователя. В диалоговом окне «Сведения о пользователе *MySQL*» введите имя пользователя, роль базы данных и пароль (например, *!phpuser*). Нажмите кнопку *ОК*. Нажмите кнопку «Далее».

13 На третьей странице конфигурации сервера *MySQL* (3/3) установите следующие параметры [46]:

а) **имя службы *Windows***. Укажите имя службы *Windows*, которая будет использоваться для экземпляра сервера *MySQL*;

б) **запустите сервер *MySQL* при запуске системы**. Не снимайте этот флажок, если сервер *MySQL* требуется для автоматического запуска при запуске системы;

в) **запуск службы *Windows* в качестве**. Возможны следующие варианты:

- **стандартная системная учетная запись**. Рекомендуется для большинства сценариев;
- **нестандартный пользователь**. Существующая учетная запись пользователя рекомендуется для сложных сценариев. Нажмите кнопку «Далее».

14 На странице «Обзор конфигурации» нажмите кнопку «Далее».

15 После успешного завершения настройки на панели «Завершение» появляется информационное сообщение. Нажмите кнопку «Завершить».

Примечание – Для проверки успешной настройки запустите диспетчер задач. Если *MySQLd-nt.exe* присутствует в списке «Процессы», сервер базы данных запущен [46].

7.2 Наполнение базы

Полезным и удобным инструментом, который позволяет создавать базы данных *MySQL* и работать с ними, является *Denwer*. Он также позволяет тестировать код *PHP*.

Рассмотрим наполнение базы данных *MySQL* с помощью *web*-приложения для администрирования систем управления базами данных (СУБД) *MySQL phpMyAdmin*.

Перейдем на страницу администрирования базы данных *MySQL* (рисунок 7.1), введя адрес: *localhost/tools/phpmyadmin/* в командной строке браузера [47].

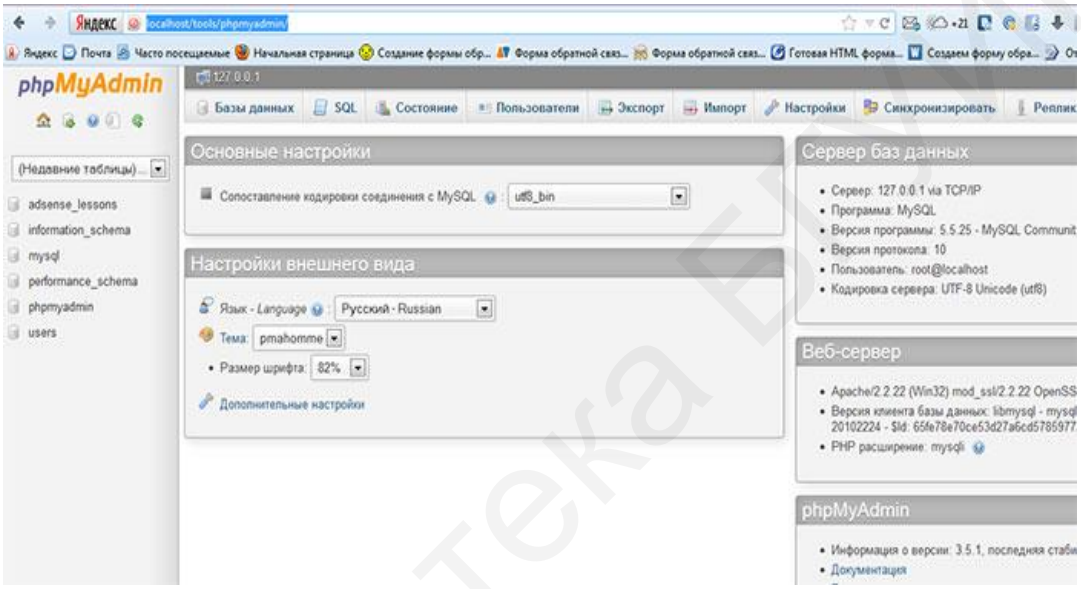


Рисунок 7.1 – Стартовая страница *phpMyAdmin*

В левой колонке находятся имеющиеся базы данных, в центральной части – основные настройки (здесь вы можете изменить язык, вид, кодировку). Верхние вкладки предназначены для различных задач.

Создание базы данных *MySQL*

Для того чтобы создать новую базу данных, необходимо нажать на верхнюю вкладку «Базы данных» – на центральном поле откроется список всех имеющихся баз данных *MySQL* [47].

Необходимо создать новую. Для этого в поле «Создать базу данных» нужно вписать название создаваемой базы и нажать на кнопку «Создать» (рисунок 7.2).

После нажатия кнопки «Создать» база данных добавится в список баз данных в панели слева и на центральном поле. Теперь выберите новую базу данных, щелкнув по ее названию [47].

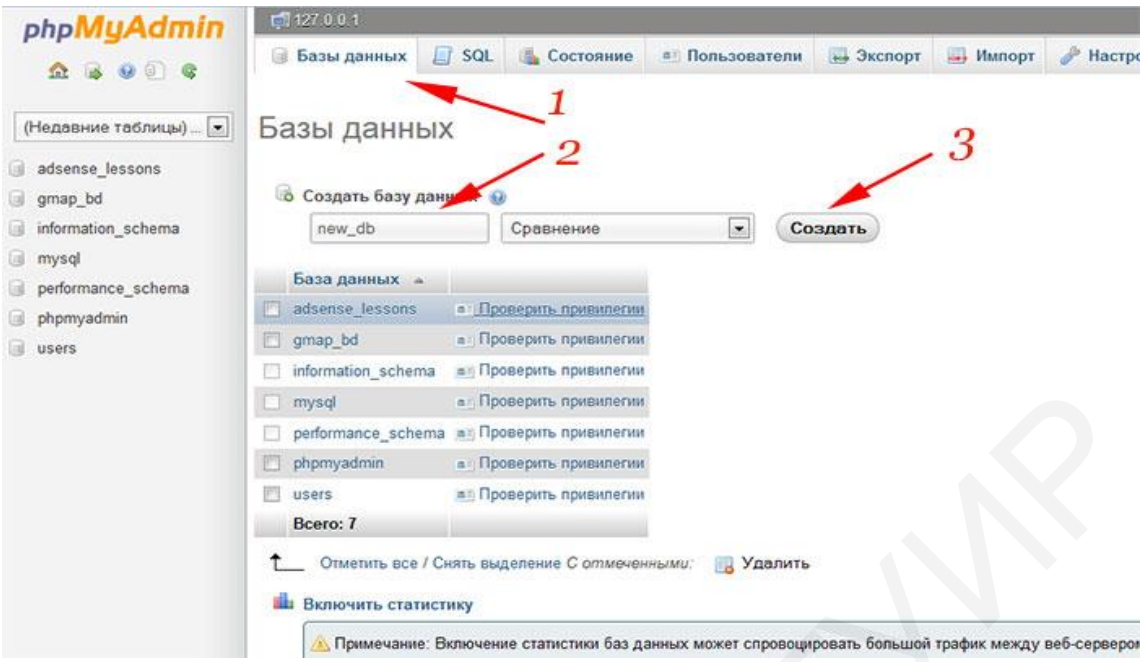


Рисунок 7.2 – Создание новой базы данных

Здесь будет предложено создать таблицу (рисунок 7.3). Для этого заполните поля «Имя» и «Количество столбцов» и нажмите *ОК*.

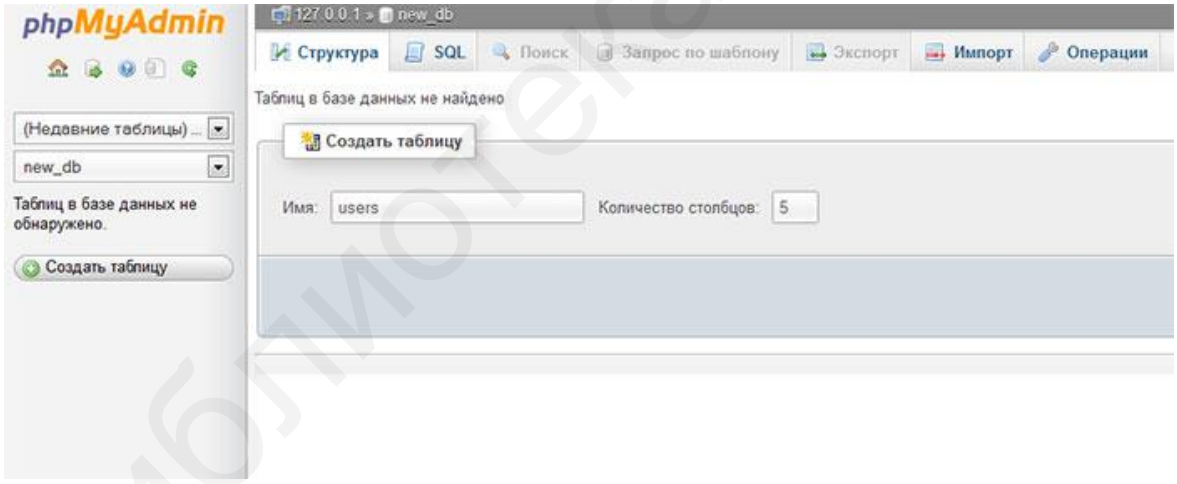


Рисунок 7.3 – Создание новой таблицы

После этого откроется страница для заполнения полей новой таблицы базы данных (рисунок 7.4). Здесь каждому полю нужно присвоить имя, тип хранимых данных, длину (если требуется для данного атрибута) и для такого поля, как идентификатор (*id*), также требуется указать автоинкремент и первичный ключ. Это должно выглядеть так, как на скриншоте (см. рисунок 7.4).

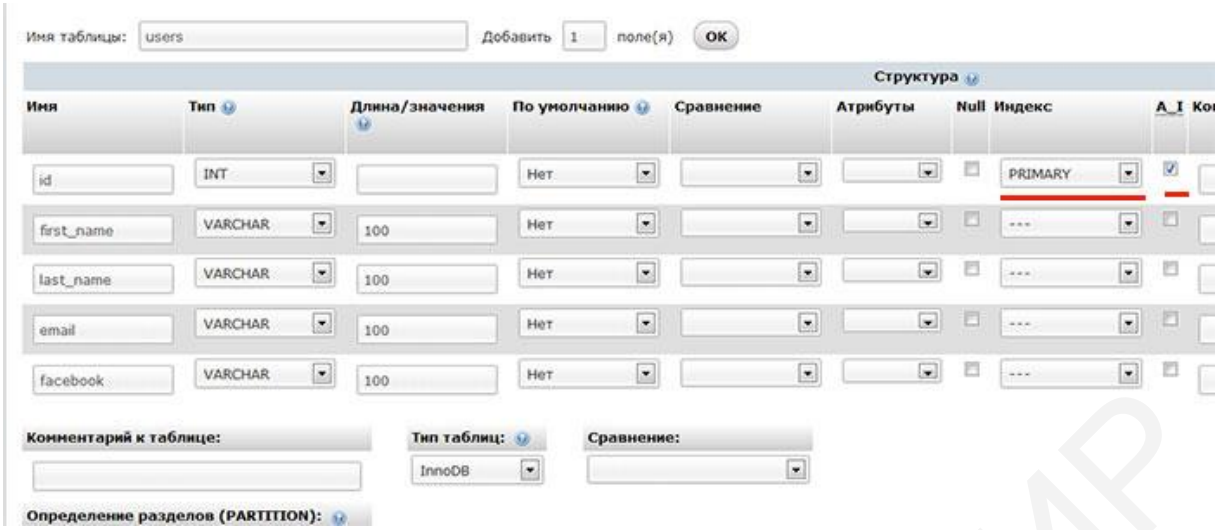


Рисунок 7.4 – Страница для заполнения полей новой таблицы

Существуют и разные типы данных, предназначенные для хранения даты, текста и других данных.

Далее нажимаем кнопку «Сохранить», и открывается созданная таблица, в которой пока нет ни одной записи (рисунок 7.5). Таблица появится в панели слева и на центральной части экрана. Щелкните по ее имени, чтобы увидеть структуру.

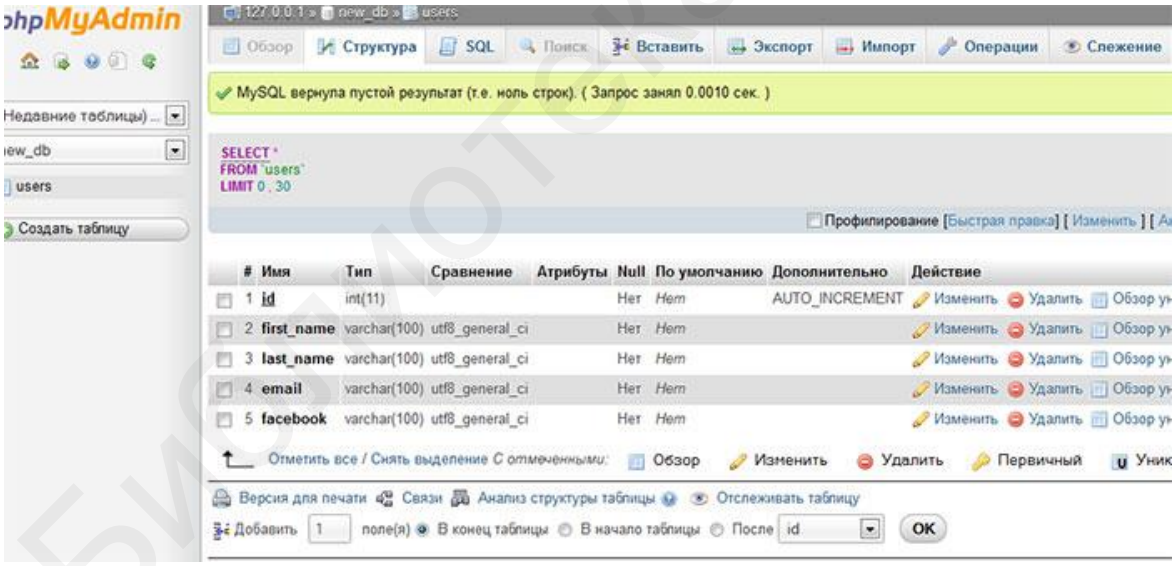


Рисунок 7.5 – Структура таблицы

Здесь можно удалить, изменить и добавить поле. Интерфейс интуитивно понятный, и сделать это не составит какого-либо труда.

Вставка нового элемента в таблицу базы данных

Для этого щелкните по верхней вкладке «Вставить» – откроется страница для вставки нового элемента в таблицу базы (рисунок 7.6). Заполните все поля (кроме поля *id*, оно будет заполняться автоматически) и нажмите кнопку *OK* [47].

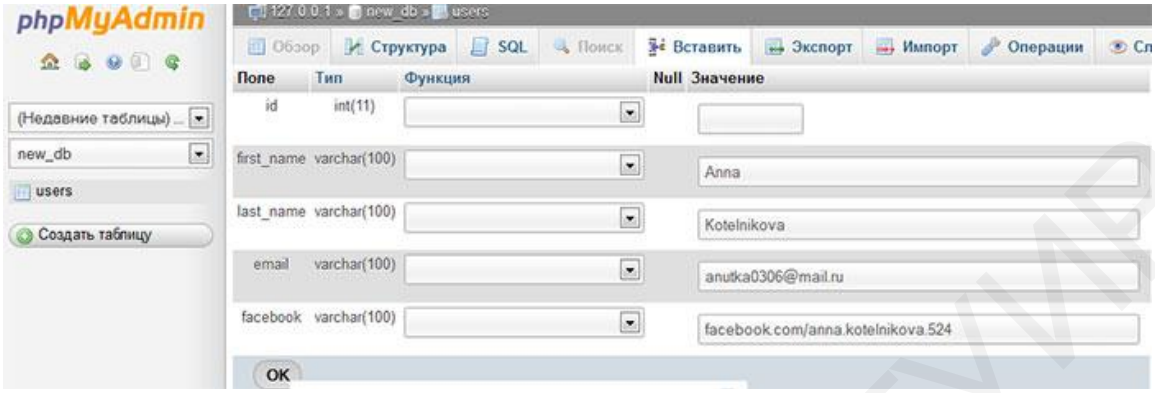


Рисунок 7.6 – Вставка нового элемента в таблицу

После нажатия *OK* перейдите к вкладке «Обзор» (она находится вверху) – вы увидите новый добавленный элемент в таблице базы данных *MySQL* (рисунок 7.7).

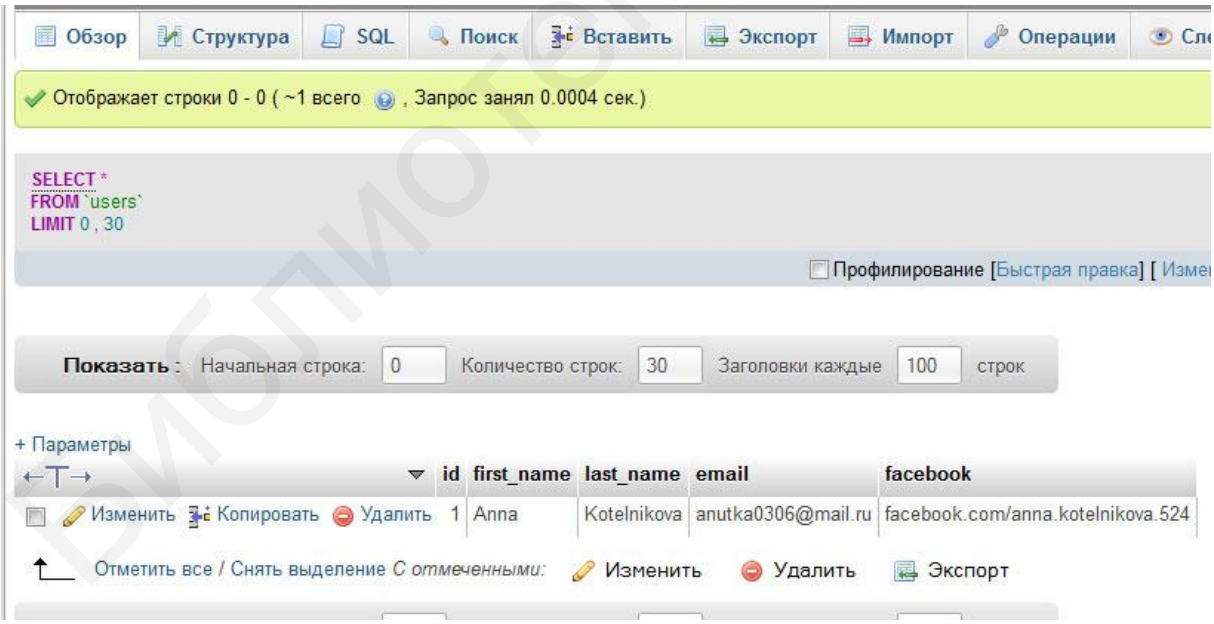


Рисунок 7.7 – Просмотр элемента таблицы базы данных

На этой же страничке есть возможность удалить или изменить добавленный элемент из таблицы базы данных.

Создать нового пользователя для базы данных – значит, создать ему имя и пароль и выставить определенные привилегии. Информация об имени

пользователя и пароле понадобится, когда будем соединяться с базой при помощи *PHP*-скрипта.

Итак, создаем нового пользователя для базы данных (рисунок 7.8). Для этого нажимаем вверху на название базы данных, после этого в верхних вкладках появится пункт «Привилегии», щелкаем по нему.

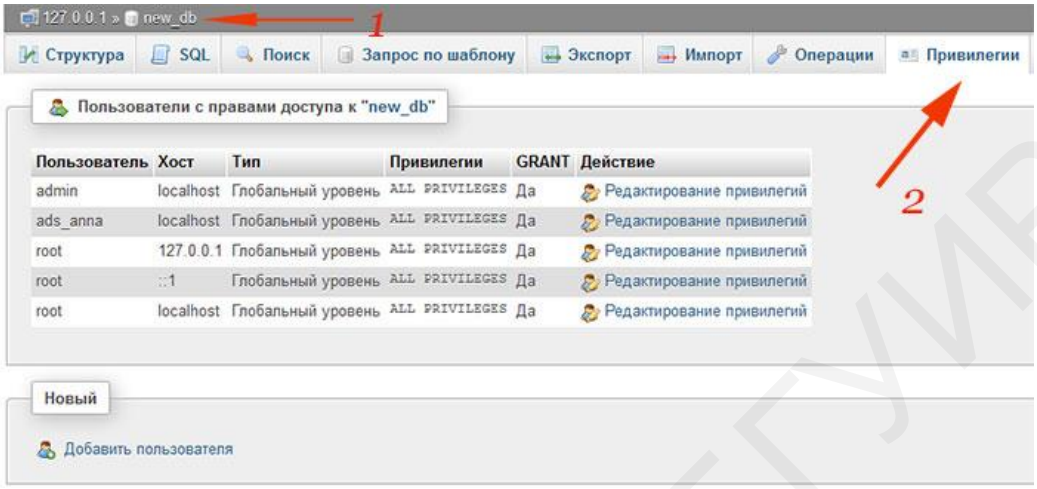


Рисунок 7.8 – Создание нового пользователя базы данных

Нажимаем «Добавить пользователя». Откроется страница с полями, которые нужно заполнить (имя пользователя, хост, пароль и подтверждение пароля) (рисунок 7.9). В качестве хоста нужно выбрать локальный хост.

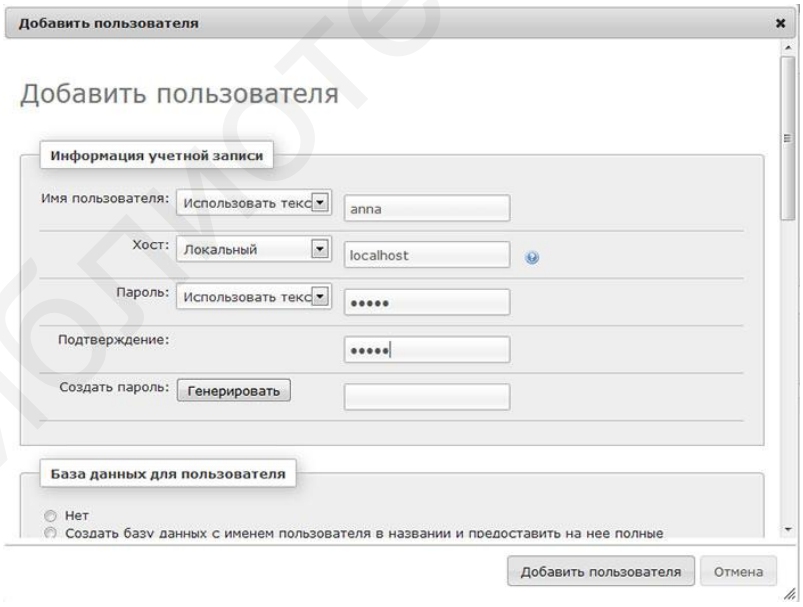


Рисунок 7.9 – Добавление нового пользователя

После чего нажимаем «Добавить пользователя», и новый пользователь будет добавлен. Будет видно сообщение, что добавлен пользователь к базе данных *new_db* со всеми привилегиями [47].

Здесь также можете редактировать привилегии, нажав на редактирование привилегий. Это может понадобиться в том случае, если кто-то еще должен иметь доступ к базе, но необходимо ограничить этого человека в привилегиях (например, он не может удалять данные). Тогда создаете нового пользователя для базы данных, но выставляете ему определенные привилегии.

Чтобы удалить базу данных, нужно снова перейти к вкладке «Базы данных», выбрать базу для удаления и нажать «Удалить» (рисунок 7.10).

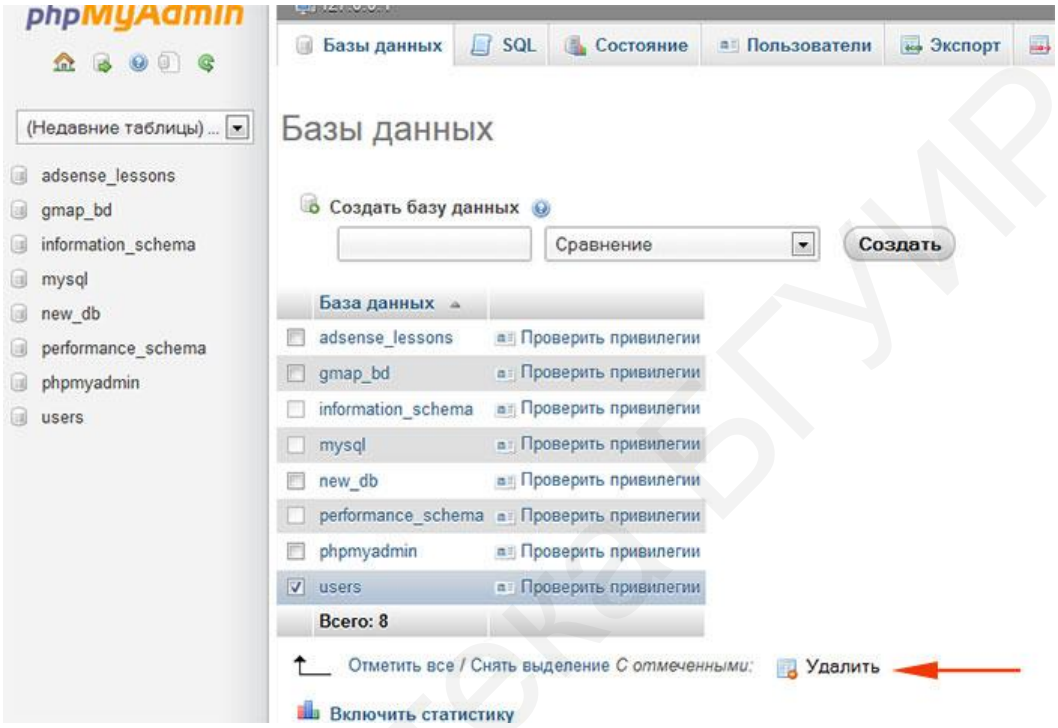


Рисунок 7.10 – Удаление базы данных

Базы данных *MySQL* и *web*-интерфейс *phpMyAdmin* позволяют реализовывать проекты различной сложности схем данных.

4.1 Задания к лабораторной работе №7

1 Изучите теоретическую часть.

2 Согласно варинту, полученному от преподавателя, создайте и наполните базу данных.

3 Оформите отчет по лабораторной работе.

4.2 Контрольные вопросы

1 Перечислите основные этапы установки базы данных *MySQL*.

2 Перечислите основные этапы наполнения базы данных *MySQL*.

Лабораторная работа №8. Язык программирования *PHP*. Фреймворк *Laravel*

Цель: изучить основные принципы разработки приложений на объектно-ориентированном языке программирования *PHP*; рассмотреть основные фреймворки языка *PHP*.

8.1 Язык программирования PHP

PHP (рекурсивный акроним словосочетания *PHP* (*Hypertext Preprocessor*) – это скриптовый язык программирования общего назначения с открытым исходным кодом [48]. Он сконструирован специально для ведения *web*-разработок, и его код может внедряться непосредственно в *HTML*.

Рассмотрим синтаксис языка *PHP*.

Классы и объекты

Пример объявления класса:

```
class Article {
// тело класса
}
```

Имена классов в *PHP* принято писать с заглавной буквы.

Объект – совокупность конкретных данных и функций для их обработки. Фактически объект – это переменная, тип данных которой задается соответствующим классом.

Для объявления объекта используется ключевое слово *new*:

```
$a = new Article();
```

Свойства и методы

Класс может содержать методы и свойства.

Методы – это функции класса.

Свойства – это переменные класса. Значением свойств может быть пустое значение, число, строка или массив.

```
class Article
{
var $id;
var $myint = 3;
var $mystr = 'str';
var $myarr = array('Jesse');
```

```
public function content(){
// содержимое метода
}
}
```

При попытке присвоить любое другое значение происходит аварийное завершение *PHP*.

Чтобы присвоить переменной неконстантное значение, нужно выполнить присвоение внутри метода класса:

```
class Article{
 var $str;
 public function update($id){
 $this->str = ‘My id is ’.$id;
 }
}
```

где $*this* – специальная переменная, содержащая ссылку на объект текущего класса.

Символ → служит для обращения к методам и свойствам класса через объект:

```
$a = new Article();
echo $a->id;
echo $a->content()
```

Наследование

При наследовании используется ключевое слово *extents*:

```
class Foo
{
    public function printItem($string)
    {
        echo 'Foo: ' . $string . PHP_EOL;
    }

    public function printPHP()
    {
        echo 'PHP is great.' . PHP_EOL;
    }
}
```

```
class Bar extends Foo
{
   public function printItem($string)
   {
      echo 'Bar: ' . $string . PHP_EOL;
   }
}
$foo = new Foo();
$bar = new Bar();
$foo->printItem('baz'); // Выведет: 'Foo: baz'
$foo->printPHP();       // Выведет: 'PHP is great'
$bar->printItem('baz'); // Выведет: 'Bar: baz'
$bar->printPHP();       // Выведет: 'PHP is great'
```

Полиморфизм

Полиморфизм – взаимозаменяемость объектов с одинаковым интерфейсом. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию, например, реализация класса может быть изменена в процессе наследования [49].

Рассмотрим свойство полиморфности классов на основе следующего примера:

```
class A {
// Выводит, функция какого класса была вызвана
function Test() { echo "Test from A\n"; }
// Тестовая функция – просто переадресует на Test()
function Call() { Test(); }
}
class B extends A {
// Функция Test() для класса B
function Test() { echo "Test from B\n"; }
}
$a=new A();
$b=new B();
$a->Call(); // выводит "Test from A"
$b->Test(); // выводит "Test from B"
$b->Call(); // Внимание! Выводит "Test from B"!
```

Спецификаторы доступа в *PHP*

Модификатор *public* позволяет обращаться к свойствам и методам отовсюду. Модификатор *private* позволяет обращаться к свойствам и методам только внутри текущего класса. Модификатор *protected* позволяет обращаться к

свойствам и методам только текущего класса и класса, который наследует свойства и методы родительского класса [50]:

```
class Person {
 public $name;
 protected $age;
 private $salary;
 public function __construct(){ }
 protected function set_age(){ }
 private function set_salary(){ }
}
```

Статические методы

Помимо символа «→», для обращения к методам или свойствам можно использовать символ «::» (двойное двоеточие). Этот синтаксис предназначен для обращения к статическим методам класса. Статические методы работают независимо друг от друга и от других методов и свойств. В статическом методе не может использоваться ссылка *$this* [51].

Статический метод выглядит так:

```
class Article{
 public static function getUser(){
  // содержимое метода.
 }
}
Article::getUser()
```

Внутри класса, в котором определен статический метод, для обращения к нему используется обозначение *self*::

```
class Article{
 public static function getuser(){
  // содержимое метода.
 }
 public function moreuser(){
  return self::getuser();
 }
}
```

Константы класса

Константы определяются как свойства класса, но с использованием метки const [51]:

```
class Math{
 const pi = 3.14159;
}
// math::pi
```

К константам, как и к статическим свойствам, можно обращаться без предварительного создания объекта, при этом используется синтаксис двух двоеточий(::).

Для обращения к константе внутри метода класса используется префикс *self*::

```
class Math{
 const pi = 3.14159;
 protected $radius;
 public function cirle(){
  return self::pi * $this→radius;
 }
}
```

Предопределенные константы

Предопределенные константы используются для получения информации о вашем коде. Имя такой константы пишется заглавными буквами между сдвоенными подчеркиваниями, например, *__LINE__* и *__FILE__* .

Вот несколько полезных предопределенных констант, доступных в *PHP* [51]:

__LINE__ возвращает номер строки в исходном файле, где используется константа;

__FILE__ представляет имя файла, включая полный путь;

__DIR__ представляет только путь к файлу;

__CLASS__ представляет имя текущего класса;

__FUNCTION__ представляет имя текущей функции;

__METHOD__ представляет имя текущего метода;

__NAMESPACE__ представляет имя текущего пространства имен.

Константа __NAMESPACE__:

```
<?php
namespace MySampleNS;
echo "Пространство имен: " . __NAMESPACE__;
// Пространство имен: MySampleNS
```

Конструктор

Метод с именем *__construct()* используется для построения объекта.

Использование конструктора [51]:

```
class user{
 public $user;
 public function __construct($username){
  $this->user = $username;
 }
}
```

Теперь при создании объекта нужно в конструктор передать значение входящего параметра *$username*.

Если неконстантное значение должно присваиваться переменной при создании объекта, оно присваивается в конструкторе (*__construct*) класса.

Использование встроенного конструктора:

```
$user = new user('Alex');
```

Деструктор

__destruct() – «магический» метод, который вызывается, когда объект уничтожается коллектором *PHP*. Данный метод не принимает аргументов и обычно используется для выполнения специальных операций, например, для закрытия соединения с базой данных [51]:

```
<?php
class MySample
{
 public function __destruct() {
 echo__CLASS__ . " вызвал деструктор.";
 }
}
$obj = new MySample; // MySample вызвал деструктор
```

«Магические» методы

PHP оставляет за собой право все методы, начинающиеся с __, считать «магическими». Не рекомендуется использовать имена методов с __ в *PHP*, если не желаете использовать соответствующий «магический» функционал [51].

Имена методов *__construct(), __destruct(), __call(), __callStatic(), __get(), __set(), __isset(), __unset(), __sleep(), __wakeup(), __toString(), __invoke(), __set_state(), __clone() и __debugInfo()* зарезервированы для «магических» методов в *PHP*. Не стоит называть свои методы этими именами, если вы не хотите использовать их «магическую» функциональность.

Иногда возникает необходимость либо выполнить какой-то код при отсутствии нужного нам метода, либо узнать, какой метод пытались вызвать, либо ис-

пользовать другое *API* для вызова нужного нам метода. С этой целью и существуют методы *__call()* и *__callStatic()* – они перехватывают обращение к несуществующему методу в контексте объекта и в статическом контексте, соответственно.

Каждый из этих «магических» методов принимает два параметра:

- имя метода, который мы пытаемся вызвать;
- список, содержащий параметры вызываемого метода.

Ключ – номер параметра вызываемого метода, значение – собственно сам параметр.

Методы __call() и __callStatic():

```
<?php
class OurClass
{
 public function __call($name,array $params)
 {
 echo 'Вы хотели вызвать $Object->'.$name.', но его не существует, и сейчас выполняется '.__METHOD__.'()<br>'
 .PHP_EOL;
 return;
 }

 public static function __callStatic($name,array $params)
 {
 echo 'Вы хотели вызвать '.__CLASS__.'::'.$name.', но его не существует, и сейчас выполняется '.__METHOD__.'()';
 return;
 }
}

$Object=new OurClass;
#Вы хотели вызвать $Object->DynamicMethod, но его не существует, и сейчас выполняется OurClass::__call()
$Object->DynamicMethod();
#Вы хотели вызвать OurClass::StaticMethod, но его не существует, и сейчас выполняется OurClass::__callStatic()
OurClass::StaticMethod();
?>
```

Этот прием лучше всего использовать при создании объектно-реляционных отношений (*ORM*). Предположим, мы хотим, чтобы приложение могло возвращать пользователя по различным критериям поиска: *id, email, phone, login* и т. д. Для этого можно создать по одному методу на каждый критерий. Однако

код этих методов получится в основном идентичным. Для решения такой задачи воспользуемся «магическим» методом *__callStatic()*.

Реализация *ORM*-отношений с использованием метода *__callStatic()*:

```
class Users{
 static function find($args){
 // Здесь реализуется логика запроса:
 // SELECT * FROM users WHERE $args[‘field’] = $args[‘value’]
 }
 static function __callStatic($method, $args){
 if(preg_match(‘/^findBy(.+)$/’, $method, $matches)){
 return static::find(array(‘field’=>$matches[1],
 ‘value’=>$args[0]));
 }
 }
}
$user = User::findById($id);
$user = User::findByEmail($email)
```

При вызове *findById() PHP* передает запрос *__callStatic()*. Внутри метода результат ищет запросы, начинающиеся с *findBy*, и извлекает остальные символы. Полученное значение и аргумент функции передается методу *Users::find()*, который, используя полученные данные, выполняет запрос.

Копирование и клонирование объектов

Чтобы присвоить один объект другому, можно воспользоваться оператором «=» [51]:

```
$tom = new user;
$tom→load_info(‘tom’);
$jhon = $tom;
```

Таким образом, модификация одного объекта приводит к изменению другого.

Копирование объектов по назначению осуществляется ключевым словом *clone*. При этом создается независимый объект с тем же содержимым (методы и свойства). Изменение одного объекта не приводит к изменению другого.

Для управления процессом клонирования можно использовать «магический» метод *__clone()*.

Если метод *__clone()* не существует, *PHP* автоматически предоставляет поверхностную копию переменной, хранящейся в *$this*.

Управление процессом клонирования:

```
class Person{
 public function __clone(){
   $this->name = clone $this->name;
 }
}
```

Переопределение обращений к свойствам

Для перехвата обращений к свойствам используются методы *__get()* и *__set()* [51].

Предположим, класс *User* хранит переменные в массиве *$data*.

Использование геттеров и сеттеров:

```
class User{
 private $data = array();
 public function __get($prop){
 if(isset($this->data[$prop])){
 return $this->data[$prop];
 }else{
 return false; }
 }
 public function __set($prop, $value){
 $this->data[$prop] = $value;
 }
}
```

Применение этих методов и массива для хранения данных упрощает инкапсуляцию данных объекта. Вместо того чтобы писать пару методов доступа для каждого свойства класса, достаточно прописать методы *__set()* и *__get()*. Дело в том, что операции чтения и записи не выполняются с переменными напрямую, а проходят через методы доступа [51].

Определение свойств через объект:

```
$jhon = new User;
 $jhon->email = ‘jhon@mail.com’;
 echo $jhon->email;
```

Кроме сокращения количества методов, специальные методы *__set()* и *__get()* упрощают реализацию централизованной проверки входных и выходных данных. Так можно ограничить использование свойств класса заранее определенным списком [51].

Ограничение свойств класса:

```
class User{
 private $data = array(‘email’=>false, ‘login’=>false);
 public function __get($prop){
 if(isset($this->data[$prop])){
 return $this->data[$prop];
 }else{
 return false;
 }}
 public function __set($prop, $value){
 if(isset($this->data[$prop])){
 $this->data[$prop] = $value;
 }else{
 return false;
 }
 }
}
```

Интерфейсы и имплементы

Результат сходного поведения в разных классах реализуют интерфейсы и имплементы [51].

Определение интерфейса:

```
interface NameInterface{
 public function getName();
 public function setName();
}
class Book implements NameInterface{
 private $name;
 public function getName(){
 return $this->name;
 }
 public function setName($name){
 return $this->name = $name;
 }
}
```

Если нужно проверить, реализует ли класс конкретный интерфейс, можно воспользоваться функцией *class_implements()* [51].

Функция *class_implements()*:

```
class Book implemets NameInterface{ }
$interface = class_implements('Book');
if(isset($interface['NameInterface'])){
 // Book реализует инрефейс NameINterface
}
```

Также возможно воспользоваться классами *Reflection*:

```
class Book implemets NameInterface{
}

$rc = new ReflectionClass('Book');
if($rc->implementsInterface('NameInteface')){
 // Book реализует интерфейс
}
```

Типажи

Если вы захотите включить код, реализующий интерфейс, в другой класс, определите типаж (*trait*) и в другом классе объявите его использование. Типажи применяются для совместного использования кода [51]:

```
trait NameTrait{
 private $name;
 public function getName(){

 }
 public function setName(){
 return $this→name = $name;
 }
}

class Book{
 use NameTrait;
}
class Picture{
 use NameTrait;
}
$book = new Book;
$book→setName('Практика разработки сайтов');
echo $book->getName();
```

Абстрактный класс и метод

Абстрактным называется класс, от которого нельзя создать объекты [51]:

```
abstract class Database{
 abstract public function connect(){}
}
```

Абстрактные классы должны содержать, как минимум, один метод с ключевым словом *abstract()*. Другие методы могут быть и неабстрактными [51].

Если класс содержит абстрактный метод, то и сам класс должен быть объявлен абстрактным.

Абстрактные методы реализуются не внутри абстрактного класса, а внутри произвольного класса, расширяющего абстрактного родителя.

Если произвольный класс не реализует все абстрактные методы родительского класса, он тоже является абстрактным. Поэтому другой класс должен осуществить дальнейшее субклассирование.

К абстрактным методам предъявляются следующие требования:

- они не могут быть закрытыми (*private*), потому что они должны использоваться при наследовании;
- абстрактные методы не могут использоваться совместно с ключевым словом *final*, потому что они должны переопределяться.

Окончательные (*final*) методы и классы

PHP 5 представляет ключевое слово *final*, разместив которое перед объявлениями методов класса, можно предотвратить их переопределение в дочерних классах. Если же сам класс определяется с этим ключевым словом, то он не сможет быть унаследован [51].

Пример использования окончательного метода:

```
class BaseClass {
 public function test() {
 echo "Вызван метод BaseClass::test()\n";
 }

 final public function moreTesting() {
  echo "Вызван метод BaseClass::moreTesting()\n";
  }
 }
```

Попытка переопределения метода *moreTesting()* закончится фатальной ошибкой: метод *BaseClass::moretesting()* не может быть переопределен.

Окончательным может быть также класс.

Использование окончательного класса:

```
final class BaseClass {
 public function test() {
 echo "Вызван метод BaseClass::test()\n";
 }
 // В данном случае неважно, укажете ли вы этот метод как final или нет
 final public function moreTesting() {
 echo "BaseClass::moreTesting() called\n";
 }
}
```

Попытка наследования от этого класса закончится фатальной ошибкой: класс *ChildClass* не может быть унаследован от окончательного класса *BaseClass*.

Множественный вызов методов одного класса

В *PHP* возможен последовательный вызов методов одного класса. Чтобы цепочки вызовов работали, следует возвращать *$this* из каждого сцепляемого метода [51].

Класс, реализующий цепочку методов:

```
class Mail {
 protected $data;
 public function from($from){
 $data[‘from’] = $from;
 return $this;
 }
 public function send(){
 print_r($this->data);
 return true;
 }
}
```

8.2 Фреймворки *PHP*

Фреймворк – программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта [52].

Достоинства *PHP*-фреймворков [53]:

- ускоряют процесс разработки;
- помогают писать структурированный код, пригодный для повторного использования;

- позволяют легко масштабировать проекты;
- соблюдают схему *MVC* (*Model – View – Controller*, модель – представление – контроллер);
- поощряют современные практики разработки, например, объектно-ориентированное программирование.

Рассмотрим основные фреймфорки *PHP*.

FuelPHP

FuelPHP – полноценный настраиваемый *PHP*-фреймворк, который поддерживает не только архитектуру *MVC*, но еще и *HMVC* (*Hierarchical* – иерархическая *MVC*). У него есть опциональный класс под названием *Presenter* (прошлое название – *ViewModel*) между слоями *Controller* и *View*, в котором содержится логика, необходимая для генерации представлений [53].

Фреймворк *FuelPHP* является модульным, его можно расширять. В фреймворке уделено внимание вопросам безопасности: реализована фильтрация данных, вносимых пользователем, запросов, а также выводимых данных. У него есть свой фреймворк авторизации и множество других содержательных функций, а также довольно обширная документация.

SLIM

Slim – микрофреймворк, идеально подходящий для небольших проектов или приложений, где полноценный фреймворк покажется лишним. Его используют многие *PHP*-разработчики для создания *RESTful API* и сервисов [53]. Среди функций *Slim* – кэширование *HTTP* на стороне клиента, *URL*-маршрутизация, шифрование сессий и *cookie*, а также мгновенные сообщения по *HTTP*-запросам. Документация полная и сделана качественно.

PHALCON

Phalcon был создан в 2012 г. и быстро стал популярным среди *PHP*-разработчиков. Его считают очень быстрым, так как он написан на *C* и *C++* с целью достигать наивысшего возможного уровня оптимизации производительности. Знания *C* не нужны, так как вся функциональность заключена в *PHP*-классы, которые можно использовать с любыми целями [53].

Так как *Phalcon* изначально был создан как расширение на *C*, его архитектура оптимизирована на низком уровне, что значительно снижает расходование ресурсов, типичное для приложений, основанных на схеме *MVC*. *Phalcon* не только повышает скорость выполнения, но и снижает уровень затрат ресурсов. У этого *PHP*-фреймворка есть и много других замечательных функций: универсальный автозагрузчик, менеджмент ресурсов, безопасность, перевод, кэширование и т. д. Документация для *Phalcon* довольна обширна, а использовать его несложно.

CAKEPHP

Фреймворку *CakePHP* уже десять лет, а он все еще популярен. Работе с ним легко обучиться, а шаблонирование – быстрое и настраиваемое [53]. Встроенная функция *CRUD* очень помогает при взаимодействии с базой данных. В по-

следнем релизе – *CakePHP* 3.*x* – улучшились управление сессиями и модульность (они разъединили несколько компонентов), а также расширились возможности создания большего количества отдельных библиотек.

Выбирайте *CakePHP*, если *web*-приложению требуется высокий уровень безопасности, *CakePHP* содержит такие функции безопасности:

- валидация ввода;
- защита от атак с использованием внедряемого *SQL* (*SQL injection*);
- предотвращение межсайтового скриптинга;
- защита от подделки межсайтовых запросов и др.

***ZEND FRAMEWORK* 2**

Zend Framework 2 – фреймворк с открытым исходным кодом, использующийся для разработки *web*-приложений и сервисов на *PHP* 5.3+ [53]. Он использует в полном обьеме объектно-ориентированный код и большинство новых функций *PHP* 5.3: пространства имен, позднее статическое связывание, лямбда-функции и замыкания. *Zend* – надежное решение со множеством вариантов конфигурации. Обычно его не рекомендуют использовать для небольших приложений, а вот для крупных проектов этот фремворк наиболее подходящий.

Среди функций *Zend Framework* 2:

- инструменты криптографического кодирования;
- простой в использовании *drag-and-drop*-редактор с поддержкой фронтенд-технологий (*HTML, CSS, JavaScript*);
- мгновенная онлайн-отладка;
- инструменты для *unit*-тестирования;
- мастер конфигурации базы данных.

Создатели этого фреймворка учли методологию *Agile*, что позволяет создавать высококачественные приложения для корпоративных клиентов.

***YII* 2**

Yii выбирают, чтобы повысить производительность сайта. Он быстрее всех остальных *PHP*-фреймворков, так как использует технологию загрузки по требованию (*lazy loading*) [53]. *Yii* 2 полностью объектно-ориентированный и основан на принципе *Don't-Repeat-Yourself* («не повторяйся»), так что основа для кода будет чистая и логичная.

Yii 2 интегрирован с *jQuery* и поставляется с набором *AJAX*-функций. Механизмы скиннинга и выбора тем здесь просты, так что фреймворк понравится тем, кто ранее занимался фронтенд-разработкой. Здесь также есть мощный генератор исходного кода *Gii*, который способствует объектно-ориентированному программированию и быстрому прототипированию, а также предоставляет *web*-интерфейс, в котором можно интерактивно генерировать нужный код.

PHPIXIE

PHPixie – сравнительно новый фреймворк, созданный в 2012 г. для сайтов-визиток [53]. Как и *FuelPHP*, *PHPixie* поддерживает схему *HMVC*. Он построен на независимых компонентах, которые можно использовать даже без самого фреймворка. Модули компонентов *PHPixie* полностью протестированы и требуют минимум других компонентов для своей работы.

Функций *PHPixie*:

- работа с БД на уровне объектов (*ORM*);
- кэширование;
- валидация ввода;
- аутентификация и возможности для авторизации.

С *PHPixie* можно использовать язык разметки *HAML*, легко вносить изменения в структуру БД, а также у него есть продуманная система маршрутизации.

CODEIGNITER

CodeIgniter – PHP-фреймворк с десятилетним стажем и очень простым процессом установки, требующим минимальной конфигурации, так что начать будет просто [53]. Станет хорошим выбором, если существует риск конфликта разных *PHP*-версий: он идеально работает практически на всех платформах виртуального и выделенного хостингов.

CodeIgniter не во всем опирается на схему *MVC*. Использование контроллеров обязательно, а моделей и представлений – нет, так что можно применять собственные стандарты оформления кода и наименований. По сути это простой фреймворк на 2 МБ, но при желании можно добавлять сторонние плагины, если требуется более сложная функциональность.

SYMFONY 2

Компоненты фреймворка *Symfony* 2 используются во многих замечательных проектах, например, *Drupal* 8 *CMS, phpBB и Laravel*. У *Symfony* большое сообщество разработчиков и огромное количество преданных поклонников [53].

Компоненты *Symfony* – это *PHP*-библиотеки, которые можно использовать повторно и выполнять с их помощью множество задач.

Задачи *Symfony* 2:

- создание форм;
- конфигурация объектов,
- маршрутизация;
- аутентификация;
- создание шаблонов и др.

Любой компонент можно установить с помощью *Composer*-менеджера пакетов для *PHP*. А в разделе *Showcase* на сайте указаны проекты, выполненные с использованием этого фреймворка.

8.3 *Laravel*

Laravel – бесплатный *web*-фреймворк с открытым кодом, предназначенный для разработки с использованием архитектурной модели *MVC*. *Laravel* выпущен под лицензией *MIT*. Исходный код проекта размещается на *GitHub* [54].

По результатам опроса *sitepoint.com* в декабре 2013 г. о самых популярных *PHP*-фреймворках *Laravel* занял место самого многообещающего проекта на 2014 г.

В 2015 г. по результатам опроса *sitepoint.com* по использованию *PHP*-фреймворков среди программистов *Laravel* занял первое место в номинациях «Фреймворк корпоративного уровня» и «Фреймворк для личных проектов».

Ключевые особенности, лежащие в основе архитектуры *Laravel* [54]:

- пакеты (от англ. *packages*) – позволяют создавать и подключать модули в формате *Composer* к приложению на *Laravel*. Многие дополнительные возможности уже доступны в виде таких модулей;

- *Eloquent ORM* – реализация шаблона проектирования *ActiveRecord* на *PHP*. Позволяет строго определить отношения между объектами базы данных. Стандартный для *Laravel* построитель запросов *Fluent* поддерживается ядром *Eloquent*;

- логика приложения – часть разрабатываемого приложения, объявленная либо при помощи контроллеров, либо маршрутов (функций-замыканий). Синтаксис объявлений похож на синтаксис, используемый в каркасе *Sinatra*;

- обратная маршрутизация – связывает между собой генерируемые приложением ссылки и маршруты, позволяя изменять последние с автоматическим обновлением связанных ссылок. При создании ссылок с помощью именованных маршрутов *Laravel* автоматически генерирует конечные *URL*;

- *REST-контроллеры* – дополнительный слой для разделения логики обработки *GET*- и *POST*-запросов *HTTP* [54];

- автозагрузка классов – механизм автоматической загрузки классов *PHP* без необходимости подключать файлы их определений в *include*. Загрузка по требованию предотвращает загрузку ненужных компонентов; загружаются только те из них, которые действительно используются;

- составители представлений (от англ. *view composers*) – блоки кода, которые выполняются при генерации представления (шаблона);

- инверсия управления (от англ. *Inversion of Control*) – позволяет получать экземпляры объектов по принципу обратного управления. Также может использоваться для создания и получения объектов-одиночек [54];

- миграции – система управления версиями для баз данных. Позволяет связывать изменения в коде приложения с изменениями, которые требуется внести в структуру БД, что упрощает развертывание и обновление приложения [54];

- модульное тестирование (*unit*-тесты) – играет очень большую роль в *Laravel*, который сам по себе содержит большое число тестов для предотвращения регрессий (ошибок вследствие обновления кода или исправления других ошибок) [54];

- страничный вывод (от англ. *pagination*) – упрощает генерацию страниц, заменяя различные способы решения этой задачи единым механизмом, встроенным в *Laravel* [54].

Установка фреймфорка *Laravel* для каждой аппаратной платформы имеет свои различия. Детальную инструкцию по установке можно найти на официальном сайте разработчика.

8.4 Разработка проекта и тестирование

Процесс написания проекта можно разбить на следующие этапы [55]:
- написание кода;
- тестирование кода;
- сборка проекта и развертывание кода.

Процесс написания кода

Разработка проекта возможна в любой операционной системе, в том числе и на *Windows*. Для этого необходима «хорошая» *IDE* (интегрированная среда разработки (от англ. *Integrated Development Environment*)), например, *PhpStorm*. Можно использовать текстовый редактор *Atom* или *Sublime Text*. Конечно, можно писать код и в обычном блокноте, например, *Notepad++*, но *IDE* использовать целесообразнее [55].

Кроме *IDE*, при нанаписании кода необходимо установить *Composer* (*Composer* – это пакетный менеджер уровня приложений для языка программирования *PHP*, который предоставляет средства по управлению зависимостями в *PHP*-приложении). Именно через *Composer* устанавливается (обновляется) *Laravel*, добавляются (обновляются) дополнительные пакеты в *web*-проект. *Composer* – это очень важный и полезный инструмент.

После подготовительной работы идет сам процесс написания кода с использованием возможностей *IDE* и фреймворка [55].

Тестирование кода

Для тестирования *web*-проекта нет необходимости загружать файлы на *FTP*-сервер, устанавливать локальный *Apache* (тот же *Denwer* или *XAMPP*), как делалось много лет назад. Это неправильно и не избавляет от ошибок в коде. На сегодняшний день с этой задачей справляются соответствующие инструменты, которые сэкономят много времени [55].

Так, *Laravel* предлагает установить *Homestead*.

Homestead – это образ операционной системы *Ubuntu*, в которой уже установлено все необходимое для работы [55].

Для установки образа необходимы *Vagrant* и *VirtualBox*. Благодаря данному образу разработчик точно знает, какие модули надо установить и как поведет себя код на *Ubuntu*. Также можете установить любой дополнительный софт [55].

Если кратко, то в системе появятся общие папки с кодом, которые будут доступны внутри образа *Ubuntu*, и записываться код будет именно внутри *Ubuntu*.

В браузере вводится *site.app*, и браузер отображает сайт из *Ubuntu*. При этом также возможен доступ к *Ubuntu* по *SSH*.

У начинающих программистов установка и настройка *Homestead* занимает немало времени, но эти действия обязательны.

Стоит отметить, что *Homestead* можно установить не только на *Linux*, но и на *Windows*.

Будем считать, что *Homestead* установлен, и сайт со свежей версией *Laravel* открывается в браузере.

Разработчик не должен писать код без тестов. Тесты дают подтверждение и уверенность, что все работает так, как задумалось. Времени на написание тестов жалеть не стоит. Каждый разработчик-профессионал обязательно пишет тесты своего кода [55].

Laravel предлагает нам инструменты для полноценного тестирования *web*-проекта со всех сторон. Причем тестирования разнообразны: можно создать временную базу данных, проверить заполнение *HTML*-форм, проверить загрузку файлов, даже содержание *PHP*-сессий и отправку писем [55].

Laravel создан для качественного тестирования всех возможностей проекта.

В *Laravel* тесты находятся в папке *tests* и выполняются командой *phpunit* в консоли либо сразу из *IDE*.

Тесты бывают нескольких типов [55]:

- функциональные – *Feature*-тесты;
- модульные – *Unit*-тесты.

Feature-тесты – функциональные тесты.

Тесты, которые проверяют функционал *web*-проекта, например, регистрацию пользователей, отправку уведомлений, заполнение *web*-форм, загрузку файлов. Они позволяют нам проверить, какие именно данные отображаются в браузере: так, не нужно заполнять *web*-формы вручную, чтобы узнать, работают ли они.

Также можно проводить тестирование с помощью *Laravel Dusk*, не просто отправляя *HTTP*-запросы, а используя реальный движок браузера *Chromium*. В этом помогает *Laravel Dusk* [55].

Unit-тесты – модульные тесты.

Эти типы тестов проверяют логику приложения, каждую функцию, отлавливают события, определяют, отправлено ли письмо, а также сверяют текст письма, проверяют, добавлено ли задание в очередь сообщений и различные поломки, если неудачно изменен код.

Каждая функция проекта должна иметь свои тесты, а когда завершен проект, то все тесты должны успешно запускаться.

При изменении функционала можно дописать тесты. Это гарантия от ошибок и помощь при диагностике проблемы.

Unit-тестирование позволяет избежать ошибок в логике приложения.

Стоит отметить, что существует методика разработки *TDD (test-driven development)* – разработка через тестирование. Сначала пишутся тесты, а затем постепенно реализуется код. Когда все тесты выполнены, то можно сказать, что завершено написание кода [55].

Кроме тестов, существуют еще другие помощники для анализа производительности *web*-приложения.

Laravel предлагает использовать *Laravel Debugbar* [55].

Это специальный пакет, который отображается на сайте в режиме отладки. С помощью него можно отследить все *SQL*-запросы к базе данных с целью их дальнейшей оптимизации.

Сборка проекта

После создания *web*-проекта и прохождения тестов необходимо подготовить проект к размещению на сервере [55].

Laravel предоставляет нам *Laravel Mix*.

Laravel Mix использует *Webpack* и умеет работать с *CSS, JS, Less, Saas, Stylus, PostCSS* [55].

Это замечательный инструмент, который, используя специальный сборщик модулей *Webpack*, собирает вместе все *JS*- и *CSS*-файлы, а также, самое главное, умеет создавать версии этих файлов.

Таким образом, каждая сборка нашего проекта позволяет иметь разные названия *JS*- и *CSS*-файлов в *HTML*-коде, что решает проблему с кэшированием при изменении содержимого файла [55].

В шаблоне нашего проекта пишем

```
<link rel="stylesheet" href="{{ mix('/css/app.css') }}">
```

После сборки он превращается:

```
<link href="/css/app.289df32d2d2c47df3b16.css" rel="stylesheet">
```

При этом браузер посетителя сразу загрузит новый файл с сайта.

Стоит отметить, что *Laravel* замечательно работает с прогрессивным *Javascript*-фреймворком *Vue* и позволяет очень удобно создавать *web*-приложения на базе *JS*-фреймворка. При этом каждый компонент можно удобно размещать в отдельном файле.

Развертывание кода (*deploy*)

Обычно после сборки проекта его файлы необходимо загрузить на сервер и обновить структуру таблиц в базе данных [55].

Не станем использовать *FTP* и *phpMyAdmin*, иначе, пока вносятся изменения, все пользователи, которые зайдут на сайт *web*-проекта, увидят множество ошибок об отсутствии каких-то файлов или полей в БД [55].

Есть очень простое и грамотное решение, которое позволяет обновлять код *web*-проекта без его отключения, и ни один пользователь при этом не получит сообщения об ошибке.

Первое, что нам необходимо изучить, – *Git* [55].

Git – это распределенная система управления версиями файлов.

С помощью *Git* можно отслеживать изменения в файлах, возвращать старую версию файлов и работать в команде над одним и тем же кодом, при этом ничего не перепутав.

Можно создать либо общедоступный код, либо приватный (для приватных репозиториев он платный).

Также можно использовать другой сервис – *BitBucket,* который позволяет бесплатно создавать приватные репозитории с кодом.

Кроме этого, сам *Git* можно настроить так, чтобы при внесении изменений происходили определенные действия [55]:

- запуск тестов проекта через *Travis CI*;
- форматирование кода по стандарту;
- анализ качества кода через инструмент.

Таким образом, весь код *web*-проекта будет храниться в *Git,* причем всегда будет качественный и проверенный.

Например, если необходимо внести изменения в официальный код *PHP*-фреймворка *Laravel,* то при внесении изменений автоматически запускаются тесты, которые проверяют работу фреймворка, учитывая новый код.

Ранее говорилось о процессе развертывания *web*-приложения. Именно для этого нам и необходим *Git*. С локальной машины загружается код *web*-приложения в *Git,* после чего произойдет автоматический запуск развертывания приложения на сервере.

Laravel Forge – сервер надежный. Для автоматического развертывания из *Git* оказывает помощь сервис *Laravel Forge* [55].

Через *Laravel Forge* можное создать виртуальный сервер в *DigitalOcean, Linode* или указать доступ к собственному серверу. При этом будет настроено абсолютно все необходимое ПО для работы *PHP*-фреймворка *Laravel* [55].

Laravel Forge автоматически устанавливает обновления, связанные с безопасностью системы. Также *Forge* легко установит бесплатный *SSL*-сертификат от *Let's Encrypt*.

Можно дать сервису *Laravel Forge* доступ к *Git*-репозиторию, и при каждом изменении в коде на сервере будет автоматически развернута его свежая версия.

В случае необходимости использования, например, десяти серверов, *Laravel Forge* может установить балансировщик нагрузки, создать десять виртуальных серверов, на каждый сервер копировать код из *Git* и запустить проект [55].

8.5 Задания к лабораторной работе №8

1 Изучите теоретическую часть.

2 Согласно варианту, выданному преподователем, разработайте *web*-приложение с использованием фреймворка *Laravel*. Примените при разработке паттерн *MVC*.

3 Реализуйте все этапы разработки и тестирования, представленные в пунтке 8.4.

4 Оформите отчет по лабораторной работе.

8.6 Контрольные вопросы

1 Перечислите основные особенности языка программирования *PHP*.

2 Назовите достоинства *PHP*-фреймворков.

3 Охарактеризуйте фреймворк *Laravel*.

4 Перечислите основные этапы разработки и тестирования проекта.

Лабораторная работа №9. *GIT. Github.com*

Цель: изучить систему контроля версий *GIT*; освоить основные возможности удаленного репозитория *Github*.

9.1 Система контроля версий *GIT*

Система контроля версий – это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определенной версии.

Git – распределенная система управления версиями [56].

Система спроектирована как набор программ, специально разработанных с учетом их использования в скриптах. Это позволяет удобно создавать специализированные системы контроля версий на базе *Git* или пользовательские интерфейсы. Например, *Cogito* является именно таким примером оболочки к репозиториям *Git*, а *StGit* использует *Git* для управления коллекцией исправлений (патчей).

Git поддерживает быстрое разделение и слияние версий, включает инструменты для визуализации и навигации по нелинейной истории разработки. Как и *Darcs*, *BitKeeper*, *Mercurial*, *Bazaar* и *Monotone*, *Git* предоставляет каждому разработчику локальную копию всей истории разработки, изменения копируются из одного репозитория в другой [56].

Удаленный доступ к репозиториям *Git* обеспечивается *git-daemon*, *SSH*- или *HTTP*-сервером. *TCP*-сервис *git-daemon* входит в дистрибутив *Git* и является наряду с *SSH* наиболее распространенным и надежным методом доступа. Метод доступа по *HTTP*, несмотря на ряд ограничений, очень популярен в контролируемых сетях, потому что позволяет использовать существующие конфигурации сетевых фильтров [56].

*Я*дро *Git* представляет собой набор утилит командной строки с параметрами. Все настройки хранятся в текстовых файлах конфигурации. Такая реализация делает *Git* легко переносимой на любую платформу и дает возможность легко интегрировать *Git* в другие системы (в частности, создавать графические *git*-клиенты с любым желаемым интерфейсом) [56].

Репозиторий *Git* представляет собой каталог файловой системы, в котором находятся файлы конфигурации репозитория, файлы журналов, хранящие операции, выполняемые над репозиторием, индекс, описывающий расположение файлов, и хранилище, содержащее собственно файлы. Структура хранилища файлов не отражает реальную структуру хранящегося в репозитории файлового дерева, она ориентирована на повышение скорости выполнения операций с репозиторием. Когда ядро обрабатывает команду изменения (неважно, при локальных изменениях или при получении патча от другого узла), оно создает в хранилище новые файлы, соответствующие новым состояниям измененных файлов. Важно, что никакие операции не изменяют содержимого уже существующих в хранилище файлов [56].

По умолчанию репозиторий хранится в подкаталоге с названием «.git» в корневом каталоге рабочей копии дерева файлов, хранящегося в репозитории. Любое файловое дерево в системе можно превратить в репозиторий *git*, отдав команду создания репозитория из корневого каталога этого дерева (или указав корневой каталог в параметрах программы). Репозиторий может быть импортирован с другого узла, доступного по сети. При импорте нового репозитория автоматически создается рабочая копия, соответствующая последнему зафиксированному состоянию импортируемого репозитория (т. е. не копируются изменения в рабочей копии исходного узла, для которых на том узле не была выполнена команда *commit*) [56].

Нижний уровень *git* является так называемой файловой системой с адресацией по содержимому (*content-addressed file system*). Инструмент командной строки *git* содержит ряд команд по непосредственной манипуляции этим репозиторием на низком уровне. Эти команды не нужны при нормальной работе с *git* как с системой контроля версий, но необходимы для реализации сложных операций (ремонт поврежденного репозитория и т. д.), а также дают возможность создать на базе репозитория *git* свое приложение [56].

Для каждого объекта в репозитории вычисляется *SHA*-1-хэш, и именно он становится именем файла, содержащего данный объект в директории *.git/objects*. Для оптимизации работы с файловыми системами, не использующими деревья для директорий, первый байт хэша становится именем поддиректории, а остальные – именем файла в ней, что снижает количество файлов в одной директории (лимитирующий фактор производительности на таких устаревших файловых системах) [56].

Все ссылки на объекты репозитория, включая ссылки на один объект, находящийся внутри другого объекта, являются *SHA*-1-хэшами.

Кроме того, в репозитории существует директория *refs*, которая позволяет задать читаемые человеком имена для каких-то объектов *git*. В командах *git* оба вида ссылок – читаемые человеком из *refs*, и нижележащие в *SHA*-1 – полностью взаимозаменяемы [56].

В классическом сценарии в репозитории *git* есть три типа объектов: файл, дерево и *commit*. Файл – это какая-то версия какого-то пользовательского файла, дерево – совокупность файлов из разного поддиректория, *commit* – некоторая дополнительная информация (например, родительский(е) *commits*, а также комментарий).

В репозитории иногда производится сборка мусора, во время которой устаревшие файлы заменяются на «дельты» между ними и актуальными файлами (актуальная версия файла хранится неинкрементально, инкременты используются только для выполнения шагов назад), после чего данные «дельты» складываются в один большой файл, к которому строится индекс. Это снижает требования по месту на диске.

Репозиторий *git* бывает локальный и удаленный [56]. Локальный репозиторий – это поддиректория *.git*, которая создается (в пустом виде) командой *git*

init и (в полном виде с немедленным копированием содержимого родительского удаленного репозитория и простановкой ссылки на родителя) командой *git clone*.

Практически все обычные операции с системой контроля версий, такие, как коммит и слияние, производятся только с локальным репозиторием. Удаленный репозиторий можно только синхронизировать с локальным как вверх (*push*), так и вниз (*pull*) [56].

Наличие полностью всего репозитория проекта локально у каждого разработчика дает *git* ряд преимуществ перед *SVN*. Так, например, все операции, кроме *push* и *pull*, можно осуществлять без наличия интернет-соединения.

Мощной возможностью *git* являются ветви, реализованные куда более полно, чем в *SVN*. Создать новую ветвь так же просто, как и совершить коммит. По сути, ветвь *git* есть не более чем именованная ссылка, указывающая на некий коммит в репозитории (используется поддиректория *refs*). *Commit* без создания новой ветви всего лишь передвигает эту ссылку на себя, а коммит с созданием ветви оставляет старую ссылку на месте, но создает новую на новый коммит и объявляет ее текущей. Заменить локальные девелоперские файлы на набор файлов из иной ветви, тем самым перейдя к работе с ней – так же тривиально [56].

Также поддерживаются субрепозитории с синхронизацией текущих ветвей в них.

Команда *push* передает все новые данные (те, которых еще нет в удаленном репозитории) из локального репозитория в репозиторий удаленный. Для исполнения этой команды необходимо, чтобы удаленный репозиторий не имел новых коммитов в себя от других клиентов, иначе *push* завершается ошибкой, и придется делать *pull* и слияние [56].

Команда *pull* обратна команде *push*. В случае, если одна и та же ветвь имеет независимую историю в локальной и в удаленной копии, *pull* немедленно переходит к слиянию [56].

Слияние в пределах разных файлов осуществляется автоматически (все это поведение настраивается), а в пределах одного файла – стандартным трехпанельным сравнением файлов. После слияния нужно объявить конфликты как разрешенные [56].

Результатом всего этого является новое состояние в локальных файлах у того разработчика, что осуществил слияние. Ему нужно немедленно сделать *commit*, при этом в данном объекте коммита в репозитории окажется информация о том, что *commit* – это результат слияния двух ветвей, который имеет два родительских коммита.

Кроме слияния, *git* поддерживает еще и *rebase* – операцию перемещения. Эта операция – получение набора всех изменений в ветви *A* с последующим их «накатом» на ветвь *B*. В результате ветвь *B* продвигается до состояния *AB*. В отличие от слияния, в истории ветви *AB* не останется никаких промежуточных коммитов ветви *A* (только история ветви *B* и запись о самом *rebase*, что упрощает интеграцию крупных и очень крупных проектов) [56].

Также *git* имеет временный локальный индекс файлов. Это промежуточное хранилище между собственно файлами и очередным *commit* (*commit* делается только из этого индекса). С помощью этого индекса осуществляется добавление новых файлов (*git add* добавляет их в индекс, они попадут в следующий *commit*), а также *commit* не всех измененных файлов (*commit* делается только тем файлам, которым был сделан *git add*). После *git add* можно редактировать файл далее – получатся три копии одного и того же файла, последняя – в индексе (та, что была на момент *git add*) и в последнем *commit* [56].

Имя ветви по умолчанию – *master* [56].

Имя удаленного репозитория по умолчанию, создаваемое *git clone* во время типичной операции «взять имеющийся проект с сервера себе на машину» – *origin*.

Таким образом, в локальном репозитории всегда есть ветвь *master*, которая является последним локальным *commit*, и ветвь *origin/master*, которая является последним состоянием удаленного репозитория на момент завершения исполнения последней команды *pull* или *push*.

Команда *fetch* (частичный *pull*) берет с удаленного сервера все изменения в *origin/master* и переписывает их в локальный репозиторий, продвигая метку *origin/master* [56].

Если после этого *master* и *origin/master* разошлись по сторонам, то необходимо сделать слияние, установив *master* на результат слияния (команда *pull* представляет собой *fetch+merge*). Далее возможно сделать *push*, отправив результат слияния на сервер и установив на него *origin/master* [56].

Git использует сеть только для операций обмена с удаленными репозиториями.

Возможно использование следующих протоколов [56]:

- *git*:// – открытый протокол, требующий наличия на сервере запущенного «демона» (поставляется вместе с *git*). Протокол не имеет средств аутентификации пользователей;
- *ssh*:// – использует аутентификацию пользователей с помощью пар ключей, а также встроенный в *UNIX*-систему «основной» *SSH*-сервер (*sshd*). Со стороны сервера требуется создание аккаунтов, шелл-код у которых будет некая команда *git*;
- *http(s)*:// – использует внутри себя инструмент *curl* (для *Windows* поставляется вместе с *git*) и его возможности *HTTP*-аутентификации, как и его поддержку *SSL* и сертификатов.

В последнем случае необходимо наличие на сервере некоего ПО *web*-приложения, которое будет исполнять роль прослойки между командами *git* на сервере и *HTTP*-сервером. Такие прослойки существуют как под *Linux*, так и под *Windows* (например, *WebGitNet*, разработанный на *ASP.NET MVC* 4).

Кроме поддержки серверной стороны команд *push* и *pull*, такие *web*-приложения могут также давать доступ только на чтение к репозиторию через *web*-браузер.

Преимущества *git перед другими DVCS* [56]:

1) высокая производительность;

2) развитые средства интеграции с другими *VCS*, в частности, с *CVS*, *SVN* и *Mercurial*. Помимо разнонаправленных конвертеров репозиториев, имеющиеся в комплекте программные средства позволяют разработчикам использовать *git* при размещении центрального репозитория в *SVN* или *CVS*, кроме того, *git* может имитировать *cvs*-сервер, обеспечивая работу через клиентские приложения и поддержку в средах разработки, специально не поддерживающих *git*;

3) продуманная система команд, позволяющая удобно встраивать *git* в скрипты;

4) репозитории *git* могут распространяться и обновляться общесистемными файловыми утилитами архивации и обновления, такими как *rsync*, благодаря тому, что фиксации изменений и синхронизации не меняют существующие файлы с данными, а только добавляют новые (за исключением некоторых служебных файлов, которые могут быть автоматически обновлены с помощью имеющихся в составе системы утилит). Для раздачи репозитория по сети достаточно любого *web*-сервера.

В числе недостатков git обычно называют [56]:

1) отсутствие сквозной нумерации *commits* монотонно непрерывно возрастающими целыми числами. Во многих проектах используется автоматическое получение номера этой версии (например, командой *svnversion*), построение .H-файла на основе этого числа, и далее его использование при создании штампа версии исполняемого файла, некоторых вшитых в него строк и т. д.;

2) отсутствие переносимой на другие операционные системы поддержки путей в кодировке *Unicode* в *Microsoft Windows (для версий msysgit до* 1.8.1*)*. Если путь содержит символы, отличные от *ANSI*, то их поддержка из командной строки требует специфических настроек, которые не гарантируют правильного отображения файловых имен при пользовании тем же репозиторием из других ОС. Одним из способов решения проблемы для *git* 1.7 является использование специально пропатченного консольного клиента. Другой вариант – использование графических утилит, работающих напрямую через *API*, таких как *TortoiseGit* [56];

3) некоторое неудобство для пользователей, переходящих с других *VCS*, таких как *SVN*, вызывают команды *git*, ориентированные на наборы изменений, а не на файлы. Например, команда *add*, которая в большинстве систем управления версиями производит добавление файла к проекту, в *git* подготавливает к фиксации сделанные в файлах изменения. При этом сохраняется не патч, описывающий изменения, а новая версия целевого файла;

4) использование для идентификации ревизий хэшей *SHA*1, что приводит к необходимости оперировать длинными строками вместо коротких номеров версий, как во многих других системах (хотя в командах допускается использование неполных хэш-строк) [56];

5) большие накладные расходы при работе с проектами, в которых делаются многочисленные несвязанные между собой изменения файлов. При работе в таком режиме размеры наборов изменений становятся достаточно велики и происходит быстрый рост объема репозиториев [56];

6) большие затраты времени по сравнению с файл-ориентированными системами, на формирование истории конкретного файла, истории правок конкретного пользователя, поиска изменений, относящихся к заданному месту определенного файла [56];

7) отсутствие отдельной команды переименования/перемещения файла, которая отображалась бы в истории как соответствующее единое действие. Существующий скрипт *git mv* фактически выполняет переименование, копирование файла и удаление его на старом месте, что требует специального анализа для определения, что в действительности файл был просто перенесен (этот анализ выполняется автоматически командами просмотра истории). Однако, учитывая тот факт, что наличие специальной команды для переименования/перемещения файлов технически не вынуждает пользователя использовать именно ее (и, как следствие, в этом случае возможны разрывы в истории), поведение *git* может считаться преимуществом [56];

8) система работает только с файлами и их содержимым, и не отслеживает пустые каталоги.

В ряде публикаций, относящихся преимущественно к 2005–2008 гг., можно встретить также нарекания в отношении документации *git*, отсутствия удобной *Windows*-версии и удобных графических клиентов. В настоящее время эта критика неактуальна: существует версия *git* на основе *MinGW* («родная» сборка под *Windows*), и несколько высококачественных графических клиентов для различных операционных систем, в частности, под *Windows* имеется клиент *TortoiseGit*, идеологически очень близкий к широко распространенному *TortoiseSVN* – клиенту *SVN*, встраиваемому в оболочку *Windows* [56].

9.2 Удаленный репозиторий *github.com*

Удаленные репозитории представляют собой версии вашего проекта, сохраненные в сети Интернет или еще где-то в сети [57]. У вас может быть несколько удаленных репозиториев, каждый из которых может быть доступен для чтения или чтения-записи. Взаимодействие с другими пользователями предполагает управление удаленными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удаленными ветками, объявлять их отслеживаемыми или нет и т. д. В данном разделе мы рассмотрим некоторые из этих навыков [57].

Просмотр удаленных репозиториев

Для того чтобы просмотреть список настроенных удаленных репозиториев, вы можете запустить команду *git remote* [57]. Она выведет названия доступных удаленных репозиториев. Если вы клонировали репозиторий, то увидите как минимум *origin* – имя по умолчанию для исходного репозитория:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100 % (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100 % (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

Вы можете также указать ключ *-v*, чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию [57]:

```
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
```

Если у вас больше одного удаленного репозитория, команда выведет их все. Например, для репозитория с несколькими настроенными удаленными репозиториями в случае совместной работы нескольких пользователей вывод команды может выглядеть примерно так [57]:

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Это означает, что мы можем легко получить изменения от любого из этих пользователей. Возможно, что некоторые из репозиториев доступны для записи, и в них можно отправлять свои изменения, хотя вывод команды не дает никакой информации о правах доступа.

Добавление удаленных репозиториев

В предыдущих разделах мы уже упоминали и приводили примеры добавления удаленных репозиториев, сейчас рассмотрим эту операцию подробнее [57]. Для того чтобы добавить удаленный репозиторий и присвоить ему имя (shortname), просто выполните команду:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
pb     https://github.com/paulboone/ticgit (fetch)
pb     https://github.com/paulboone/ticgit (push)
```

Теперь вместо указания полного пути вы можете использовать *pb*. Например, если вы хотите получить изменения, которые есть в *pull,* но нет у вас, вы можете выполнить команды [57]:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100 % (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100 % (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]     master     -> pb/master
 * [new branch]     ticgit     -> pb/ticgit
```

Ветка *Master* из репозитория *pull* сейчас доступна вам под именем *pb/master*. Вы можете «слить» ее с одной из ваших веток или переключить на нее локальную ветку, чтобы просмотреть содержимое ветки *pull* [57].

Получение изменений из удаленного репозитория – *fetch* и *pull*

Как вы только что узнали, для получения данных из удаленных проектов следует выполнить следующее [57]:

```
$ git fetch [remote-name]
```

Данная команда связывается с указанным удаленным проектом и забирает все те данные проекта, которых у вас еще нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удаленного проекта. Теперь эти ветки в любой момент могут быть просмотрены или слиты [57].

Когда вы клонируете репозиторий, команда *clone* автоматически добавляет этот удаленный репозиторий под именем *origin*. Таким образом, *git fetch origin* извлекает все наработки, отправленные *push* на этот сервер после того, как вы клонировали его (или получили изменения с помощью *fetch*). Важно отметить, что команда *git fetch* забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную «объединить» эти данные с вашими, когда вы будете готовы [57].

Если у вас есть ветка, настроенная на отслеживание удаленной ветки, то вы можете использовать команду *git pull*, чтобы автоматически получить изменения из удаленной ветви и «слить» их со своей текущей ветвью. Этот способ более простой или удобный. К тому же по умолчанию команда *git clone* автоматически настраивает вашу локальную ветку *Master* на отслеживание удаленной ветки *master* на сервере, с которого вы клонировали (подразумевается, что на удаленном сервере есть ветка *Master*). Выполнение *git pull*, как правило, извлекает (*fetch*) данные с сервера, с которого вы изначально клонировали, и автоматически пытается «слить» (*merge*) их с кодом, над которым вы в данный момент работаете [57].

Отправка изменений в удаленный репозиторий (*push*)

Когда вы хотите поделиться своими наработками, вам необходимо отправить их в главный репозиторий *push*. Команда для этого действия простая [57]:

```
git push [remote-name] [branch-name]
```

Чтобы отправить вашу ветку *Master* на сервер *Origin* (повторимся, что клонирование, как правило, настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок на сервер [57]:

```
$ git push origin master
```

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду *push*. Если клонируют одновременно несколько человек, затем один выполняет команду *push*, а далее эту же команду выполняет другой, то ваш *push* точно будет отклонен. В этом случае следует сначала вытянуть (*pull*) их изменения и объединить со своими. Только после этого будет позволено выполнить *push* [57].

Просмотр удаленного репозитория

Если хотите получить побольше информации об одном из удаленных репозиториев, вы можете использовать команду *git remote show [remote-name]*. Выполнив эту команду с некоторым именем, например, *origin*, вы получите результат, аналогичный следующему [57]:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Она выдает *URL* удаленного репозитория, а также информацию об отслеживаемых ветках. Эта команда сообщает вам, что, если вы, находясь на ветке *master*, выполните *git pull*, ветка *master* с удаленного сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдает список всех полученных ею ссылок [57].

Это был пример простой ситуации, и, наверняка вы встречались с чем-то подобным. Однако если вы используете *Git* активно, то можете увидеть гораздо большее количество информации от *git remote show*:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master tracked
    dev-branch tracked
    markdown-strip tracked
    issue-43 new (next fetch will store in remotes/origin)
    issue-45 new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11  stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
```

```
  master merges with remote master
Local refs configured for 'git push':
  dev-branch pushes to dev-branch (up to date)
  markdown-strip pushes to markdown-strip (up to date)
  master  pushes to master (up to date)
```

Данная команда показывает, какая именно локальная ветка будет отправлена на удаленный сервер по умолчанию при выполнении *git push*. Она также показывает, каких веток с удаленного сервера у вас еще нет, какие ветки все еще есть у вас, но уже удалены на сервере. И для нескольких веток показано, какие удаленные ветки будут в них влиты при выполнении *git pull* [57].

Удаление и переименование удаленных репозиториев

Для переименования ссылок в новых версиях *Git'a* можно выполнить *git remote rename*, это изменит сокращенное имя, используемое для удаленного репозитория. Например, если вы хотите переименовать *pb* в *paul*, вы можете это сделать при помощи *git remote rename* [57]:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Стоит упомянуть, что это также меняет для вас имена удаленных веток. То, к чему вы обращались как *pb/master*, теперь стало *paul/master* [57].

Если по какой-то причине вы хотите удалить ссылку (вы сменили сервер или больше не используете определенное зеркало, или, возможно, контрибьютор перестал быть активным), то можете использовать *git remote rm*:

```
$ git remote rm paul
$ git remote
origin
```

Всего на *GitHub* используют 337 уникальных языков программирования. Из них самым популярным на сервисе снова стал *JavaScrpit*: на нем выполнено 2,3 млн запросов на добавление кода (*pull-request*). Второе место занял *Python* с 1 млн запросов, который лишь немного обошел *Java* с 986 тыс. запросов (третье место). Десятку лидеров замыкает *C*, у которого 239 тыс. *pull*-реквестов [57].

Самой популярной темой в тегах репозиториев стало машинное обучение, далее – игры и *iOS*. Самым популярным репозиторием по количеству веток стал *Tensorflow* (его модели обучения заняли пятую позцию в этом же рейтинге), за которым следуют *Bootstrap*, *gitignore*, *jekyll-now* и *Vuejs*.

GitHub отмечает, что 48 % новых пользователей сервиса называют себя студентами, 28 % – профессионалами, а еще 22 % используют *GitHub* в качестве хобби. При этом 45 % новичков начали изучать программирование параллельно с открытием профиля на сайте [57].

9.3 Задания к лабораторной работе №9

1 Изучите теоретическую часть.

2 Создайте четыре ветки любого ранее разработанного приложения при помощи *GIT*. Ветки должны быть таких типов, как *Master, Development, Test, Release, Pre-release*. Приложение должно иметь характерные отличия для каждой ветки.

3 Оформите отчет по лабораторной работе.

9.4 Контрольные вопросы

1 Что такое системы контроля версий?
2 Назовите основные особенности *Git.*
3 Охарактеризуйте команды *push* и *pull.*
4 Перечислите протоколы, применяемые в *Git.*
5 Перечислите основные операции репозитория *github.com*.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Разработка мобильных *web*-приложений – Открытые системы. СУБД [Электронный ресурс]. – Режим доступа : https://www.osp.ru/os/2013/09/13038284/.

2 Как тестировать мобильное приложение? *IT*-академия *Stormnet* [Электронный ресурс]. – Режим доступа : http://www.it-courses.by/testing-mobile-application/.

3 Объектно-ориентированные языки программирования. Объектно-ориентированное программирование [Электронный ресурс]. – Режим доступа : https://bourabai.ru/alg/oop11.htm.

4 *Java* – Прогопедия [Электронный ресурс]. – Режим доступа : http://progopedia.ru/language/java/.

5 Основная информация о языке *Java* – Университет ИТМО [Электронный ресурс]. – Режим доступа : https://neerc.ifmo.ru/wiki/index.php?titleОсновная информация_о_языке_Java.

6 *Java*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Java.

7 *Java – StudFiles* [Электронный ресурс]. – Режим доступа : https://studfiles.net/preview/2874314/page:4/.

8 *C Sharp*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/C_Sharp.

9 Введение в PHP. Общий обзор языка программирования *PHP* – *METANIT.COM*. Сайт о программировании [Электронный ресурс]. – Режим доступа : https://metanit.com/web/php/1.1.php.

10 Операционная система *Android* корпорации *Google*. Операционные системы вычислительных машин [Электронный ресурс]. – Режим доступа : http://bourabai.kz/os/android.htm.

11 «Картинка в картинке», счетчик уведомлений на иконках и другие функции, которые *Android* 8.0 скопирует у *iOS – MacDigger* [Электронный ресурс]. – Режим доступа : http://www.macdigger.ru/iphone-ipod/kartinka-v-kartinke-schetchik-uvedomlenij-na-ikonkax-i-drugie-funkcii-kotorye-android-8-0-skopiruet-u-ios.html.

12 Обзор программного обеспечения *Android*-систем – *CyberLeninka* [Электронный ресурс]. – Режим доступа : https://cyberleninka.ru/article/n/obzor-programmnogo-obespecheniya-android-sistem.

13 *Android*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Android.

14 *iOs*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/IOS.

15 Операционная система *iOs* – ipadinsider [Электронный ресурс]. – Режим доступа : http://ipadinsider.ru/raznoe/operacionnaya-sistema-ios.html.

16 Операционная система *iOs*: история, прогресс, совершенствование – Научный форум [Электронный ресурс]. – Режим доступа : https://nauchforum.ru/studconf/tech/xviii/4989.

17 *Windows Phone*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Windows_Phone.

18 Шаблон проектирования. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Шаблон_ проектирования.

19 Паттерны для новичков: *MVC vs MVP vs MVVM* – Хабрахабр [Электронный ресурс]. – Режим доступа : https://habrahabr.ru/post/215605/.

20 Концепция *MVC* для чайников – *RUSELLER.COM*. Частная коллекция качественных материалов для тех, кто делает сайты [Электронный ресурс]. – Режим доступа : https://ruseller.com/lessons.php?id=666.

21 Паттерн *MVP – Professor Web. Net & Web Programming* [Электронный ресурс]. – Режим доступа : https://professorweb.ru/my/WPF/documents_WPF/level36/36_4.php.

22 *Model – View – ViewModel*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Model-View-ViewModel.

23 Паттерн *HMVC* в *web*-разработке – Хабрахабр [Электронный ресурс]. – Режим доступа : https://habrahabr.ru/post/212065/.

24 Особенности *Android – itmagick.ru* [Электронный ресурс]. – Режим доступа : http://itmagick.ru/osobennosti-android/.

25 Мобильные устройства – ДамОтвет.ру [Электронный ресурс]. – Режим доступа : http://computer.damotvet.ru/mobile/228040.htm.

26 Сотовый телефон. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Сотовый_телефон.

27 Карманный персональный компьютер. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Карманный_персональный_компьютер.

28 Электронная книга. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Электронная_книга.

29 Что такое планшет, и какие его функции – Мобильные компьютеры [Электронный ресурс]. – Режим доступа : http://planshetniypc.ru/plpc.html.

30 *Android Studio IDE* от *Google – WINFOX* [Электронный ресурс]. – Режим доступа : http://wnfx.ru/android-studio-ide-ot-google/.

31 Введение в разработку мобильных приложений [Электронный ресурс]. – Режим доступа : https://www.intuit.ru/studies/courses/12643/1191/lecture/21980?page=3.

32 *JetBrains PhpStorm – I.T. PRO Professional IT Solutions* [Электронный ресурс]. – Режим доступа : https://itpro.ua/product/jetbrains-phpstorm/?tab=description.

33 Как установить эмулятор Андроида на ПК с *Windows* [Электронный ресурс]. – Режим доступа : http://zpostbox.ru/kak_ustanovit_emulyator_androida_na_pk_s_windows.html.

34 Что делать, если под рукой нет *Android*-устройства? Обзор *Android*-эмуляторов – Хабрахабр [Электронный ресурс]. – Режим доступа : https://habrahabr.ru/post/218739/.

35 Типы ресурсов приложения *Android* – *microsin.net* [Электронный ресурс]. – Режим доступа : http://microsin.net/programming/android/android-application-resource-types.html.

36 Работа с файловой системой – *METANIT.COM*. Сайт о программировании [Электронный ресурс]. – Режим доступа : https://metanit.com/java/android/13.1.php.

37 *JSON*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/JSON.

38 Подключение сторонней библиотеки в проект *Android Studio* – *stack overflow* [Электронный ресурс]. – Режим доступа : https://ru.stackoverflow.com/questions/Подключение-сторонней-библиотеки-в-проект-android-studio.

39 Файл манифеста AndroidManifest.xml – Освой программирование играючи. Сайт Александра Климова [Электронный ресурс]. – Режим доступа : http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php.

40 *AndroidManifest.xml* – что это и зачем он нужен? [Электронный ресурс]. – Режим доступа : http://androidteam.ru/dev/particles/androidmanifest-xml.html.

41 *Activity* (Активность, Деятельность) – Освой программирование играючи. Сайт Александра Климова [Электронный ресурс]. – Режим доступа : http://developer.alexanderklimov.ru/android/theory/activity-theory.php.

42 Урок 117. Виджеты. Создание. *Lifecycle* – *Start Android* – учебник по *Android* для начинающих и продвинутых [Электронный ресурс]. – Режим доступа : http://startandroid.ru/ru/uroki/vse-uroki-spiskom/195-urok-117-vidzhety-sozdanie-lifecycle.html.

43 Шаблон MVC на *Android* – *stack.io* [Электронный ресурс]. – Режим доступа : http://qaru.site/questions/16100/mvc-pattern-on-android.

44 Архитектура *Android* приложений – Хабрахабр [Электронный ресурс]. – Режим доступа : https://habrahabr.ru/post/278815/.

45 Подключение библиотеки к проекту в *Eclipse* – *JAVA-HELP*. сайт для разработчиков приложений под *ANDROID* [Электронный ресурс]. – Режим доступа : http://java-help.ru/add-library-in-project-eclipse/.

46 Настройка сервера баз данных *MySQL* в операционной системе *Windows* [Электронный ресурс]. – Режим доступа : https://netbeans.org/kb/docs/ide/install-and-configure-mysql-server_ru.html?print=yes#download.

47 Создание базы данных *MySQL* – *website-create.ru*. Блог о создании *web*-сайтов [Электронный ресурс]. – Режим доступа : http://website-create.ru/web-zametki/sozdanie-saitov/28-sozdanie-baz-dannih.html.

48 *PHP*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/PHP.

49 Инкапсуляция, наследование, полиморфизм [Электронный ресурс]. – Режим доступа : http://codrob.ru/lesson/26.

50 Модификаторы доступа и инкапсуляция *METANIT.COM*. Сайт о программировании [Электронный ресурс]. – Режим доступа : https://metanit.com/java/tutorial/3.3.php.

51 ООП в PHP – *Web*-программирование [Электронный ресурс]. – Режим доступа : https://web.obmenka.by/1/115.

52 Фреймворк. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Фреймворк.

53 10 лучших *PHP*-фреймворков для *web*-проектов – *Kultprosvet* [Электронный ресурс]. – Режим доступа : https://kultprosvet.net/ru/blog/10-luchshih-php-freymvorkov-dlya-veb-proektov.

54 *Laravel*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Laravel.

55 *Laravel* – экосистема, а не просто *PHP*-фреймворк – Хабрахабр [Электронный ресурс]. – Режим доступа : https://habrahabr.ru/post/334776/.

56 *Git*. Википедия – свободная энциклопедия [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Git.

57 Основы *Git* – Работа с удаленными репозиториями – *Git* [Электронный ресурс]. – Режим доступа : https://git-scm.com/book/ru/v2/Основы-Git-Работа-с-удаленными-репозиториями.

Св. план 2018, поз. 2

Учебное издание

Рак Алексей Олегович
Михалькевич Александр Викторович
Середа Александр Сергеевич и др.

РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ ДЛЯ МОБИЛЬНЫХ СИСТЕМ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редактор *Е. В. Иванюшина*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *В. М. Задоля*

Подписано в печать 16.11.2019. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 7,79. Уч.-изд. л. 8,2. Тираж 50 экз. Заказ 276.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск