Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Макаренко Елена Николаевна
Должность: Ректор
Дата подписания: 29.07.2022 18:15:39
Уникальный программный ключ:
c098bc0c1041cb2a4cf926cf171d6715d99a6ae00adc8e27b55cbe1e2dbd7c78

Комплекс практических работ
по дисциплине
«Облачные технологии»

Оглавление

Практическая работа 1. Подготовка рабочего места 3

Практическая работа 2. Создание первого проекта 18

Практическая работа 3. Настройка хранилища разработки в VISUAL STUDIO 2015 39

Практическая работа 4. Хранилище данных с реляционной структурой 74

Практическая работа 5. Работа с Windows Azure Table 89

Лабораторное занятие 6. Работа с Windows Azure Blob 105

Практическая работа 7: Работа с Windows AzureQueue 127

Практическая работа 1. Подготовка рабочего места

Материал в этой работе будет посвящен формированию рабочего пространства в части облачных и мобильных технологий для сред: Visual Studio 2015 и IntelliJ IDEA. Обе среды – это интегрированные среды разработки (IDE) для языков C# и Kotlin соответственно

Установка и настройка компонентов для создания рабочей станции Visual Web Developer Workstation хорошо описана здесь: http://blogs.technet.com/b/isv_team/archive/2010/12/27/3377315.aspx

Чтобы начать работу в части мобильных технологий, загрузите IntelliJ IDEA Community Edition с сайта JetBrains по адресу https://www.jetbrains.com/idea/download.

Подготовьте рабочее место в части облачных технологий:

1. Установите VS 2015 и MS *SQL Server* 2008/12.

Установка данного инструментария подробно описана во многих ресурсах и в целом проста (см. пункты 1 - 3 в списке вспомогательных материалов).

2. Настройка *IIS для Windows 7/8/10*

Откройте панель управления (Пуск - Панель управления), а затем откройте надстройку "Программы и функции" (рисунок 1.1). Выберите меню "Включить или отключить компоненты Windows".

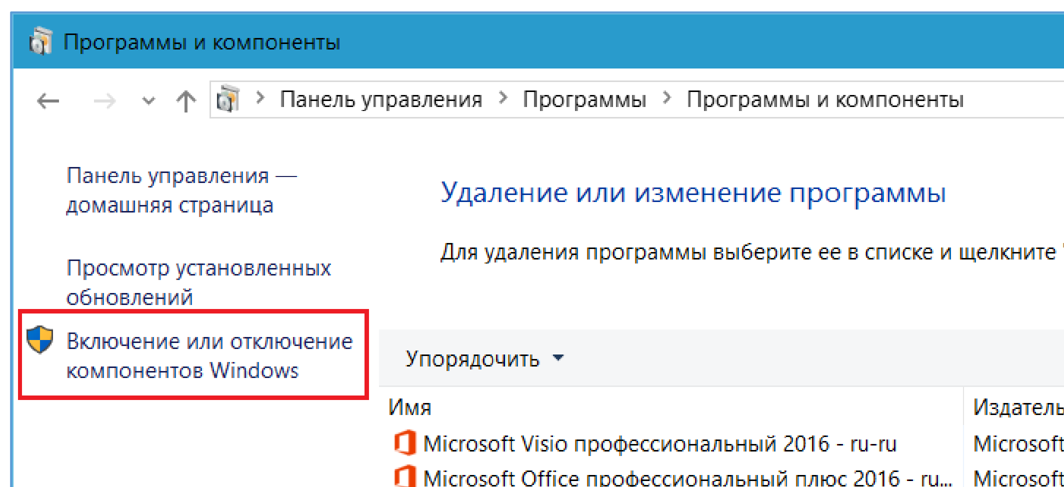


Рисунок 1.1. Снимок "Программы и компоненты"

Развернуть ". Net Framework 3.5" и включите пункт "*Enable Windows Communication Foundation over HTTP"* (рисунок 1.2).

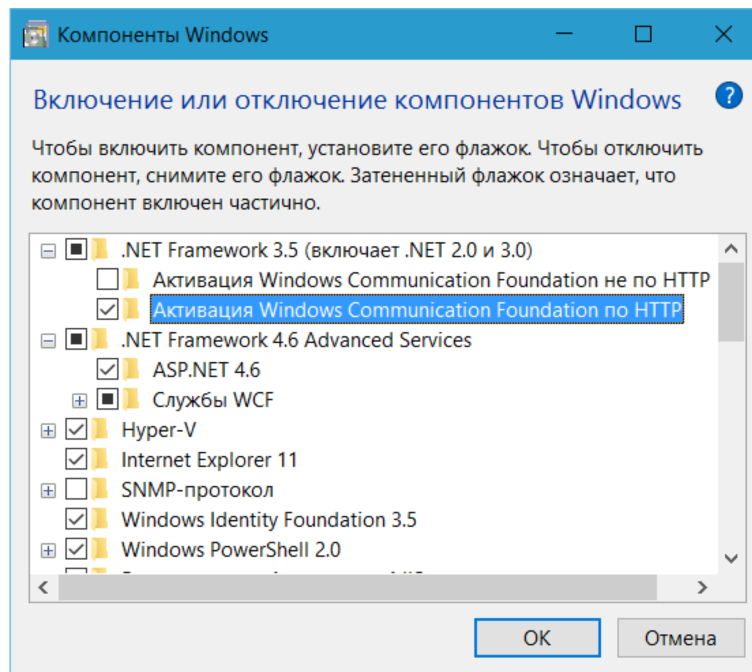


Рисунок 1.2. Компоненты окна

Последовательно разверните узлы "*IIS* Services", "Internet Services" и "Application Development Components", проверьте узлы " *ASP*. *NET*" и "*CGI*" (рисунок 1.3).

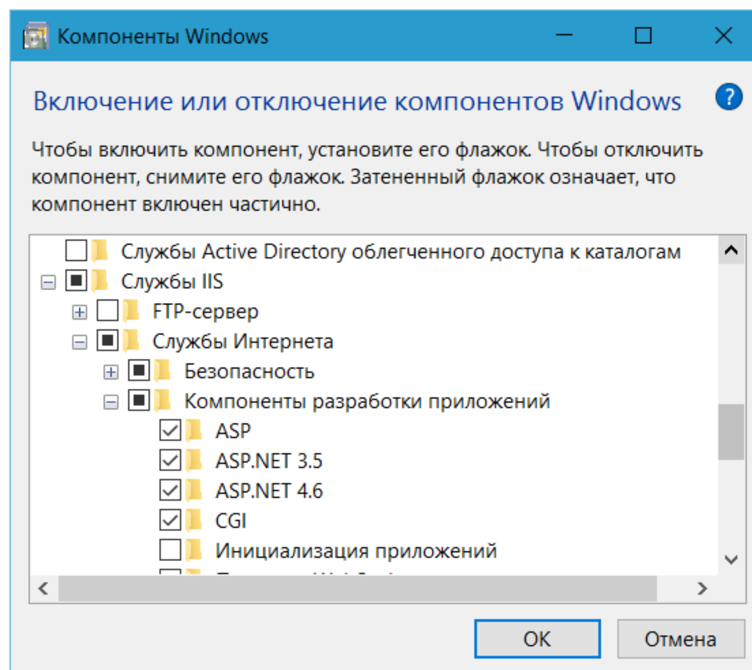


Рисунок 1.3. компоненты окна

В узле " *IIS* Services" разверните "Internet Services" и "*HTTP* Common Features". Отметьте пункт "Статическое содержимое" (рисунок 1.4).

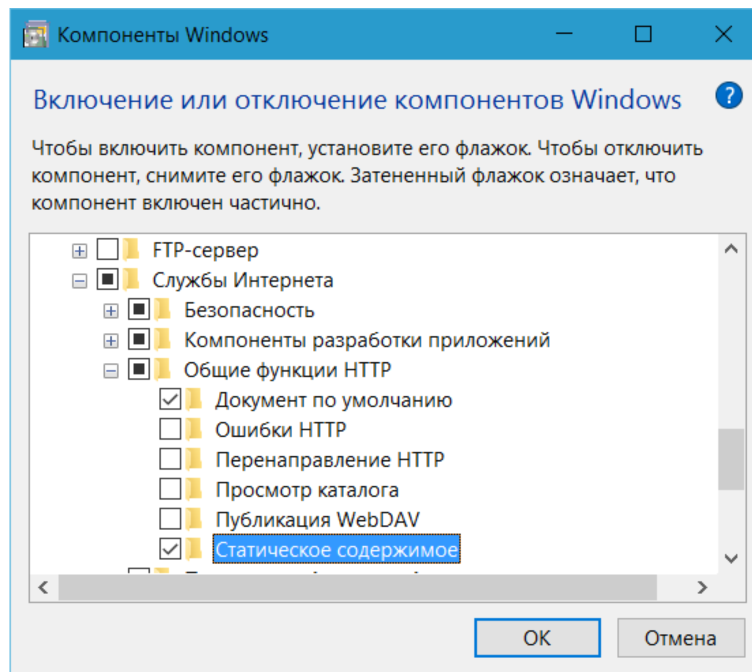


Рисунок 1.4. Компоненты окна

В узле "*IIS* Services*"* разверните "Web Management Tools" и отметьте "*IIS* Management *Console*" (рисунок 1.5).

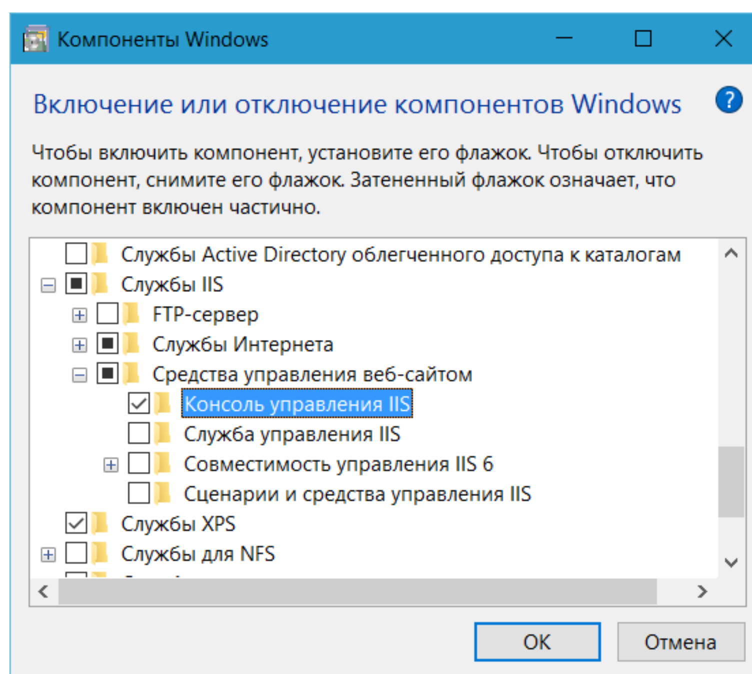


Рисунок 1.5. Компоненты окна

Затем нажмите "OK" и дождитесь завершения процесса установки отмеченных *компонентов.*

Дождитесь завершения процесса установки.

3. Установите *Windows* Azure *SDK*.

Инструменты для работы (Windows Azure SDK) можно загрузить по следующей ссылке:

На этом установка *автономных* средств разработки облачных приложений завершена.

Знание инструментария

Теперь нам нужно разобраться с тем, что мы установили и где все искать.

Эмулятор *вычислений* ("*Development Fabric*") и *эмулятор хранения* ("*Development Storage*") находятся в папке Azure ("C:\Program Files" по умолчанию). Эмуляторы находятся в подкаталоге Emulator.

Если установка инструментария успешно завершена, *модель Cloud* появится в списке проектов VS2015 (рисунок 1.6).

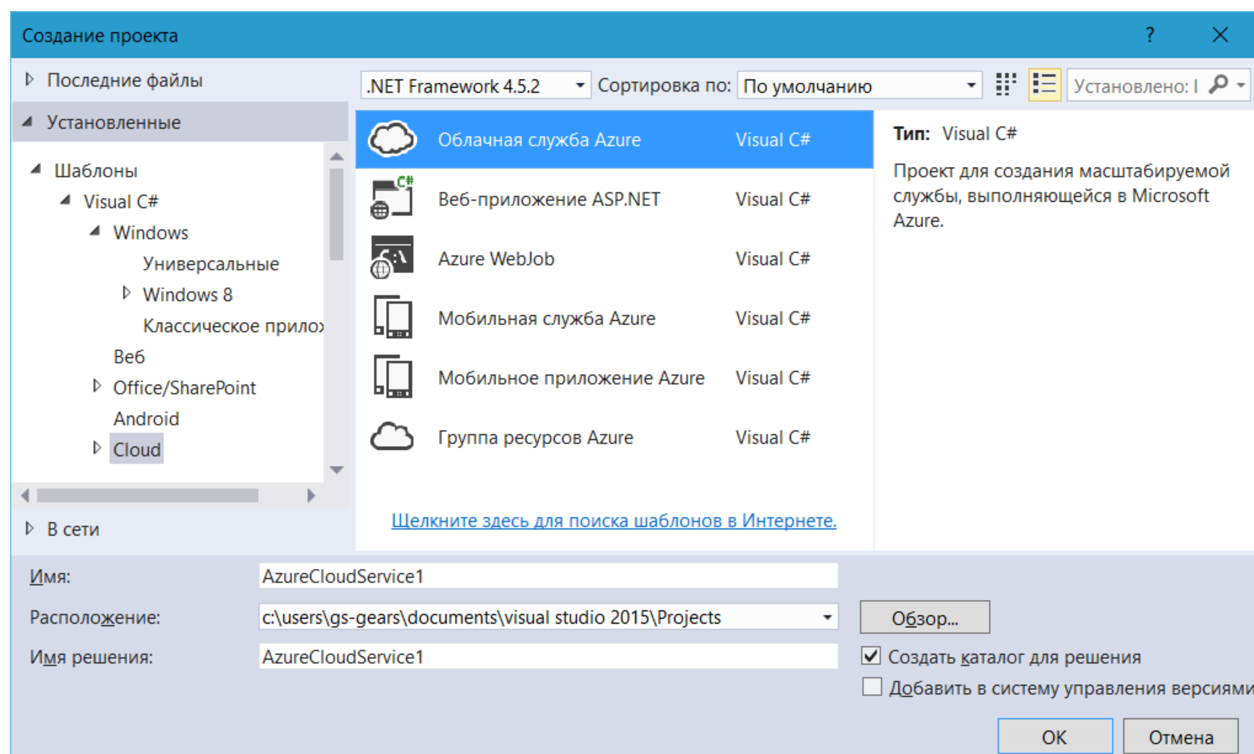


Рисунок 1.6. Модель облачного проекта

Подготовьте рабочее место в части мобильных технологий:

Следуйте инструкциям для вашей системы в инструкциях по установке и настройке на сайте JetBrains: https://www.jetbrains.com/help/idea/installation-guide.html и на сайте Surgebook, перейдя по ссылке https://www.surgebook.com/JunMidSen/book/put-java-programmista/ustanovka-intellij-idea.

Среда IntelliJ IDEA, или просто IntelliJ, помогает вам писать хорошо структурированный код на языке Kotlin. Она также упрощает процесс разработки благодаря встроенным инструментам для запуска, отладки, исследования и рефакторинга кода.

Запустите IntelliJ. Откроется окно приветствия Welcome to IntelliJ IDEA. Нажмите на Create New Project. IntelliJ отобразит новое окно New project, как

показано на рисунок 1.7. В окне New Project слева выберите Kotlin, а справа — Kotlin/JVM.

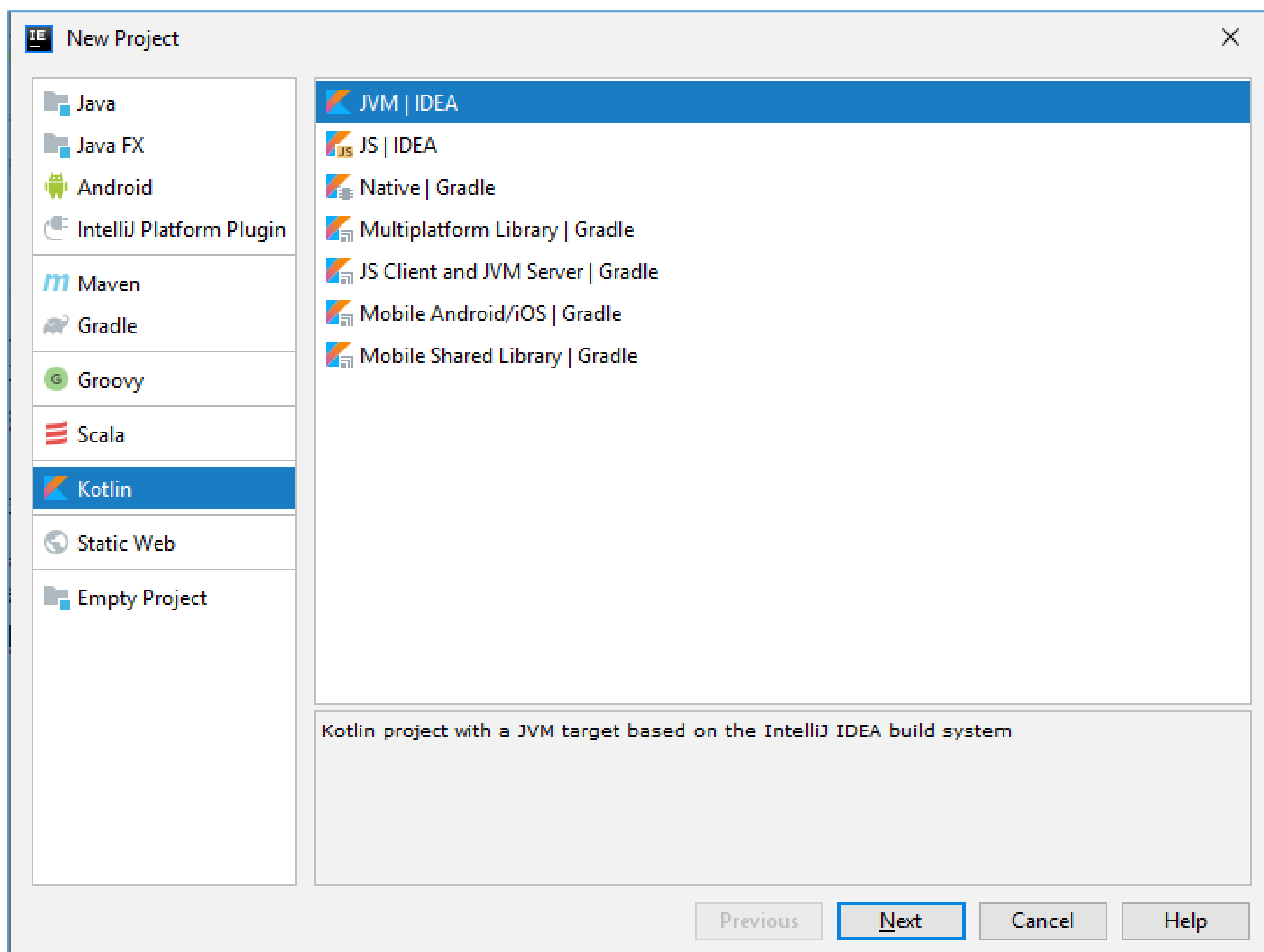


Рисунок 1.7. Окно создания нового проекта IntelliJ

Вы также можете писать код в IntelliJ на других языках, помимо Kotlin, таких как Java, Python, Scala и Groovy. Выбор Kotlin/JVM означает, что вы будете писать на Kotlin. Кроме того, Kotlin/JVM указывает на то, что вы будете писать код, который будет выполняться на виртуальной машине Java. Одним из преимуществ Kotlin является наличие набора инструментов, позволяющих писать код, работающий на разных операционных системах и на разных платформах. (В дальнейшем Java Virtual Machine будет сокращенно называться JVM. Эта аббревиатура часто используется в сообществе разработчиков Java).

Нажмите кнопку Далее в окне Новый проект. IntelliJ отобразит окно с настройками для нового проекта. Введите имя проекта "Sandbox" в поле Имя проекта. Поле местоположения проекта будет заполнено автоматически. Вы можете оставить местоположение по умолчанию или изменить его, нажав на кнопку справа от поля. Выберите Java версии 1.8 из выпадающего списка SDK Project, чтобы подключить ваш проект к Java Development Kit (JDK) версии 8.

JDK предоставляет среде IntelliJ доступ к JVM и инструментам Java, необходимым для перевода кода Kotlin в байт-код (об этом ниже). Технически, подойдет любая версия, начиная с 6-й.

Если Java 1.8 отсутствует в раскрывающемся списке SDK Project, это означает, что JDK версии 8 еще не установлен. Прежде чем продолжить, сделайте следующее: загрузите JDK 8 для вашей системы по адресу oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

Установите JDK и перезапустите IntelliJ. Повторите шаги, описанные ранее, чтобы создать новый проект.

Когда окно настроек будет выглядеть как на рисунок 1.8, нажмите Finish.

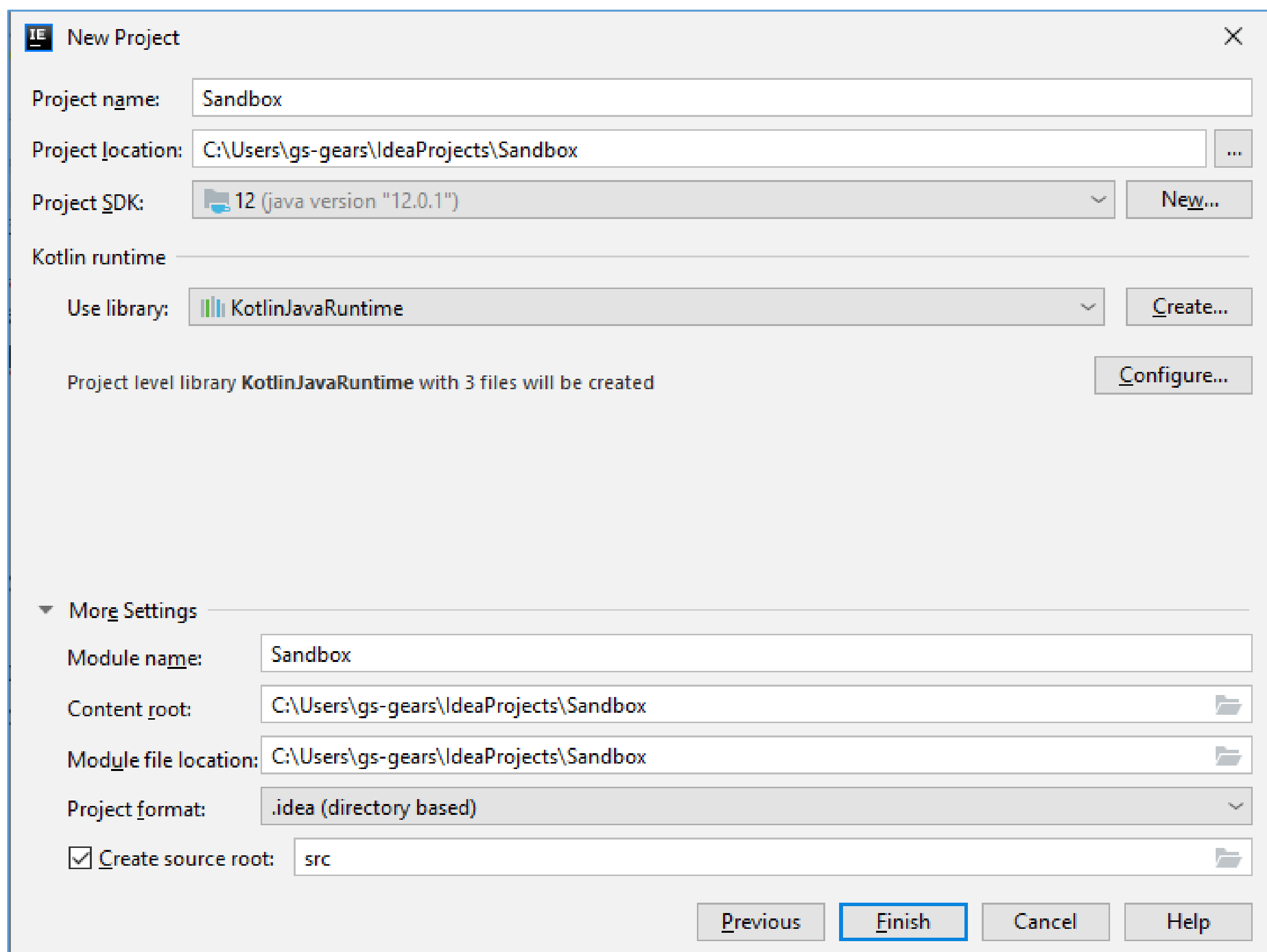


Рисунок 1.8. Настройки проекта

Выполнение файла Kotlin

Когда вы закончите формирование кода первого приложения, IntelliJ отобразит зеленую стрелку, известную как "Run Program", слева от первой строки – функции main. (Если значок не появляется или вы видите красную линию под именем файла в рамке или в любом месте введенного вами кода, это означает, что в коде есть ошибка. Проверьте код. Однако, если отображается синий красный флаг Kotlin K, это то же самое, что и "Run Program").

Нажмите кнопку Выполнить. В появившемся меню выберите Run 'HelloKt'. Это сообщит IntelliJ, что вы хотите увидеть программу в действии. После запуска IntelliJ выполнит код в фигурных скобках ({}), строка за строкой, и завершит работу. Внизу появится новое окно инструмента.

Окно инструмента выполнения, также известное как консоль, показывает информацию о том, что происходит после запуска программы, а также строки,

которые выдала программа. В консоли должно появиться сообщение Hello World! Вы также можете увидеть Process finished с кодом выхода 0, что означает, что программа завершилась успешно.

Основные конструкции Kotlin

Далее описываются возможности операторов языка и работы со строками.

1. Циклы

Цикл — управляющая конструкция, предназначенная для организации многократного исполнения набора инструкций.

Оператор цикла с предусловием while

Операторы *тела цикла* повторяются до тех пор, пока *условие* истинно. На рисунке 1.9 приведен фрагмент блок-схемы с циклом с предусловием.

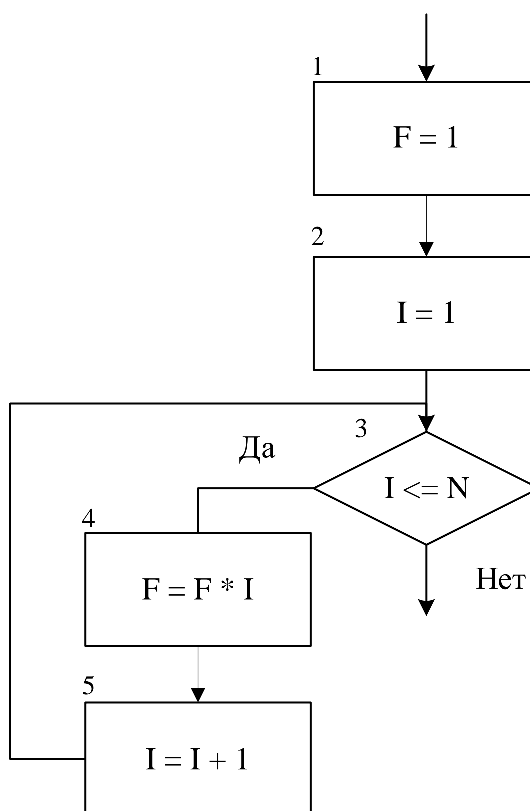


Рисунок 1.9. Фрагмент блок-схемы с циклом с предусловием

Общий вид оператора:

while (*условие*)

{

тело цикла;

}

Во время выполнения тела цикла значения каких-то переменных могут измениться — в результате условие цикла может уже не быть истинным. Если условие по-прежнему истинно, то тело цикла выполняется снова.

Условие цикла записывается как и для if: с помощью операций отношения (>, >=, <, <=, !=, ==). Сложные условия можно составлять с помощью логических операций !, &&, ||.

Один шаг цикла (выполнение тела цикла) называет **итерацией**.

Цикл while следует использовать всегда, когда какая-то часть кода должна выполниться несколько раз, причем невозможно заранее сказать, сколько именно раз.

Рассмотрим пример, в котором цикл будет выполняться, до тех пор, пока не будет введено число меньше 0:

```
import java.util.*

fun main() {
    // Считывание данных с консоли
    val sc = Scanner(System.`in`)
    println("Введите число!")
    var number = sc.nextInt()
    while (number > 0) {
        println("Было введено положительное число! Вводите дальше.")
        number = sc.nextInt()
        println("Так-так, что тут у нас...")
    }
    println("Было введено отрицательное число или ноль. Всё.")
}
```

Представим действия программного кода в виде трассировочной таблицы (таблица 1.1).

Таблица 1.1. Трассировочная таблица для примера 1

Шаг	Действие	Пояснение	Цикл
1	`number = sc.nextInt()`	`number = 10`	
2	`while (number > 0)`	`10 > 0` (Истина)	Входим в цикл, 1-я итерация
3	`println("Было введено положительное число! Вводите дальше.")`	Вывод	
4	`number = sc.nextInt()`	`number = 2`	
5	`println("Так-так, что тут у нас...")`	Вывод	
6	`while (number > 0)`	`2 > 0` (Истина)	2-я итерация
7	`println("Было введено положительное число! Вводите дальше.")`	Вывод	
8	`number = sc.nextInt()`	`number = 3`	
9	`println("Так-так, что тут у нас...")`	Вывод	
10	`while (number > 0)`	`3 > 0` (Истина)	3-я итерация
11	`println("Было введено положительное число! Вводите дальше.")`	Вывод	
12	`number = sc.nextInt()`	`number = -1`	
13	`println("Так-так, что тут у нас...")`	Вывод	
14	`while (number > 0)`	`-1 > 0` (Ложь)	Выход из цикла
15	`println("Было введено отрицательное число или ноль. Всё.")`	Вывод	

Данное объяснение позволяет понять то, как работает условие цикла while.

Оператор цикла с постусловием do

Если в оператор цикла с предусловием *while* выполняет тело цикла, пока условие остается истинным, то оператор цикла с постусловием *do* выполняет тело первый раз безусловно. После этого оно выполняется только в случае, пока условие остается истинным.

На рисунке 1.10 приведен фрагмент блок-схемы с циклом с постусловием.

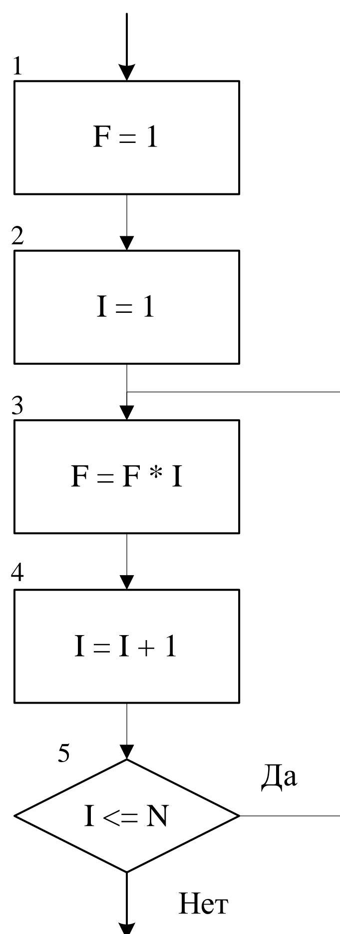


Рисунок 1.10. Фрагмент блок-схемы с циклом с постусловием

Общий вид оператора:

do
{
    тело цикла;
}
while (*условие*);

Операторы *тела цикла* повторяются до тех пор, пока *условие* истинно.

Оператор цикла с параметром for

Цикл for выполняет блок кода заданное количество раз.

Общий вид оператора:

for (*инициализирующее_выражение* **in** *интервал*)

{

тело цикла;

}

Инициализирующее_выражение выполняется только один раз в начале выполнения цикла и, как правило, инициализирует счетчик цикла.

Kotlin предусматривает *интервалы* для представления линейного набора значений.

Интервал определяется оператором .., например 1..5. Интервал включает все значения, начиная с находящегося слева от оператора .. и заканчивая находящимся справа. Например, 1..5 включает числа 1, 2, 3, 4 и 5. Интервалы могут представлять последовательности символов. Интервалы в Kotlin – закрытые или включающие, т. е. второе значение всегда является частью интервала.

Для проверки попадания заданного числа в интервал можно использовать ключевое слово **in** (внутри).

Рассмотрим пример, в котором необходимо три раза получить цены на какой-то товар и вычислить общую цену товара.

```
import java.util.*

fun main() {
    var total = 0.0
    val sc = Scanner(System.`in`)
    println("Введите цену!")
    for (i in 1..3){
        val price = sc.nextFloat()
        total += price
    }
    println("Сумма введённых чисел равна $total")
}
```

Обратите внимание, что цикл for присваивает переменной *i* (она называется итератором цикла) значения (1, потом 2...), хотя нигде нет оператора присваивания = или его родственников типа +=.

В программах, решающих абстрактные, математические задачи, допустимо называть переменные короткими и непонятными именами типа *n* или *i*. Однако этого лучше избегать. Кроме того, стоит соблюдать общепринятые договоренности: буквой *n* обычно обозначают количество чего-либо (например, итераций цикла). При этом если есть хоть какая-то определенность (например, речь идет о количестве автомобилей), то стоит и переменную назвать более понятно (например, *cars*). Буквами *i* и *j* (по-русски они традиционно читаются как «и» и «жи») обычно обозначают итераторы цикла *for*.

Операторы перехода

Оператор **break** прекращает выполнение ближайшего к нему цикла.

Оператор **continue** передает управление на проверку условия циклов **while** и **do while**, либо на *инкрементирующее_выражение* цикла **for**.

В каких ситуациях какой цикл следует использовать

Цикл while нужен, когда какой-то блок кода должен выполниться несколько раз, причем заранее неизвестно, сколько именно раз.

Цикл for нужен, когда какой-то блок кода должен выполниться несколько раз, при этом известно сколько раз еще до начала цикла.

2. Строки

В программировании текстовые данные представляются строками — упорядоченными последовательностями символов. В строках важен порядок элементов (символов). Порядок символов дает возможность пронумеровать их (проиндексировать). Индексация символов начинается с 0:

0	1	2	3	4	5	6	7	8	9
П	Р	И	В	Е	Т	М	И	Р	!

По индексу можно получить соответствующий ему символ строки. Для этого нужно после имени строки написать в квадратных скобках индекс элемента.

```
fun main() {
    val str: String = "Привет Kotlin"
    val initialLetter = str[0]
    println(initialLetter)  // сделаем то же, что println('П')
    val otherLetter = str[3]
    println(otherLetter)  // сделаем то же, что println('в')
}
```

Однако, используя индексацию, изменить какой-либо символ строки нельзя. Строка относится к неизменяемым типам данных.

```
val str: String = "Привет Kotlin"
str[5] = "д"     // Пытаемся изменить, но:
// Error: Kotlin: No set method providing array access
```

Цикл *for* можно использовать для перебора всех букв в строке.

```
fun main() {
    val hello: String = "hello, my dear friends!"
    var counts = 0
    for (s in hello){
        if (s in "aeiouy")
            counts++
    }
    print(counts)
}
```

Но, так как символы в строке проиндексированы, то еще один способ перебрать все элементы в строке: перебрать все индексы, используя интервалы.

```
fun main() {
    val hello: String = "hello, my dear friends!"
    var counts = 0
    for (i in 0 until hello.length){
        if (hello[i] in "aeiouy")
            counts++
    }
    print(counts)
}
```

При использовании интервалов, кроме оператора **..** существуют еще несколько функций их создания. Функция **downTo** создает убывающий интервал. Функция **until** создает интервал, не включающий верхнюю границу выбранного диапазона.

Основные методы работы со строками

Основные операции со строками раскрывается через методы класса String, среди которых можно выделить следующие:

- equals(): сравнивает строки с учетом регистра
- equalsIgnoreCase(): сравнивает строки без учета регистра
- regionMatches(): сравнивает подстроки в строках
- indexOf(): находит индекс первого вхождения подстроки в строку
- lastIndexOf(): находит индекс последнего вхождения подстроки в строку
- startsWith(): определяет, начинается ли строка с подстроки
- endsWith(): определяет, заканчивается ли строка на определенную подстроку
- replace(): заменяет в строке одну подстроку на другую
- reversed(): переворачивает (инвертирует) строку
- trim(): удаляет начальные и конечные пробелы
- substring(): возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса
- toLowerCase(): переводит все символы строки в нижний регистр
- toUpperCase(): переводит все символы строки в верхний регистр

С полным перечнем методом можно ознакомиться по ссылке https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/index.html

Задание 1 – Вычислить S.

$$S = \sum_{i=1}^{10} \frac{2i}{i}$$

Порядок выполнения:

a. Создание нового Kotlin файла

Добавьте в проект новый файл, щелкнув правой кнопкой мыши на папке **src** в окне инструментов проекта. Выберите New – Kotlin File/Class. Ввести имя файла, например, Task01_1. Нажать кнопку «Enter».

b. Написание кода основной программы

В появившемся окне необходимо ввести программный код решения задачи, приведенный ниже.

Листинг 1.1. Программный код решения задачи 1

```
fun main(args: Array<String>) {
    var s = 0.0
    // Объявление цикла от 1 до 10
    for (i in 1..10) {
        s += 2 * i / i.toDouble()
    }
    print("Результат: s = $s")
}
```

c. Разбор программного кода:

В строке 01 объявляем главный метод `main()`. В объявлении переменной `s` типа `double` присваиваем ей начальное значение (`s = 0.0`, инициализация переменной `s` в строке 02). В переменной `s` будет храниться сумма 10 шагов вычисления выражения. В цикле `FOR` (04-05) выполняется вычисление суммы выражения. Печать результатов производится в строке 06.

d. Построение проекта

После ввода программного кода нужно скомпилировать и отладить программу. Для этого необходимо выполнить Build – Build Project. Если в программном коде имеются ошибки, они будут выделены красным цветом.

e. Запуск программы

Чтобы просмотреть результат выполнения программы, нужно выполнить Run – Run 'Task01_1Kt' или нажать кнопку в виде треугольника (рисунок 1.11).

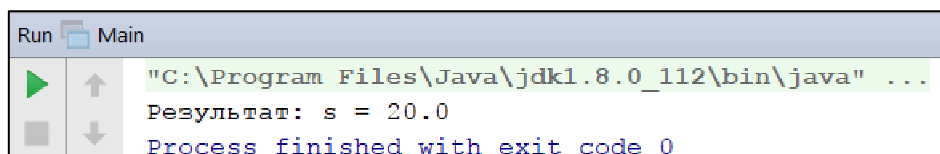


Рисунок 1.11. Результат решения задачи 1

Задание 2 – Вычислить сумму и произведение последовательности из 10 случайных чисел.

Порядок выполнения:

a. Создание нового Kotlin файла

Добавьте в проект новый файл, щелкнув правой кнопкой мыши на папке **src** в окне инструментов проекта. Выберите New – Kotlin File/Class. Ввести имя файла, например, Task01_2. Нажать кнопку «Enter».

b. Написание кода основной программы

В появившемся окне необходимо ввести программный код решения задачи, приведенный ниже.

Листинг 1.2. Программный код решения задачи 2

```
fun main(args: Array<String>) {
    var tmp = 0
    var tmp2: Long = 1
    print("Сумма\t Произведение \n")
    for (i in 0..9) {
        tmp += Math.round(Math.random() * 10).toInt()
        tmp2 = tmp2 * tmp
        print("$tmp \t\t $tmp2 \n")
    }
}
```

c. Разбор программного кода

В строке 01 объявляем главный метод `main()`. В объявлении переменной `tmp` типа `int` присваиваем ей начальное значение (`tmp=0`, инициализация переменной `tmp` в строке 02). В переменной `tmp` будет храниться сумма 10 членов последовательности из случайных чисел. Далее объявляем переменную `tmp2` типа `long`, в которой будем хранить значения произведения членов последовательности (03). В цикле `FOR` (05-08) находится сумма и произведение элементов последовательности. Генерация случайного члена последовательности осуществляется с использованием стандартной функции `Math.random()`, которая возвращает случайное число в диапазоне от 0 до 1. Далее умножаем случайное число на 10 и округляем `Math.round()` Печать результатов производится в строке 08.

d. Построение проекта

После ввода программного кода нужно скомпилировать и отладить программу. Для этого необходимо выполнить Build – Build Project. Если в программном коде имеются ошибки, они будут выделены красным цветом.

e. Запуск программы

Чтобы просмотреть результат выполнения программы, нужно выполнить Run – Run ‘Task01_2Kt’ или нажать кнопку в виде треугольника (рисунок 1.12).

Сумма	Произведение
8	8
17	136
22	2992
23	68816
28	1926848
35	67439680
37	2495268160
40	99810726400
50	4990536320000
56	279470033920000

Рисунок 1.12. Результат решения задачи 2

Библиографический список

1. Установка Visual Studio 2015. URL: http://msdn.microsoft.com/ru-ru/library/e2h7fzkw.aspx (Дата обращения 21.10.2021 г.)
2. Руководства по устранению неполадок. URL: http://msdn.microsoft.com/ru-ru/library/ee460770.aspx (Дата обращения 21.10.2021 г.)
3. Исакова С., Жемеров Д. Kotlin в действии / пер. с англ. Киселев А.Н. — М.: ДМК-Пресс, октябрь 2017 г., 402 с.
4. Скин Д., Гринхол Д. Kotlin. Программирование для профессионалов / пер. с англ. Киселев А.Н. — СПб.: Издательский дом «Питер», 2020 г., 464 с.
5. Официальная документация языка программирования Kotlin. URL: https://kotlinlang.ru/ (Дата обращения 21.10.2021 г.)

Практическая работа 2. Создание первого проекта

В этой практической работе в части облачных технологий будет описано, как развернуть веб-приложение ASP.NET в веб-приложение Azure Application Service с помощью Visual Studio 2015, в части мобильных технологий описываются возможности работы с коллекциями в языке Kotlin: списки и множества. Списки – тип данных, который представляет собой упорядоченный набор элементов с доступом по индексам. Множества – тип данных аналогичен математическим множествам, он поддерживает быстрые операции проверки наличия элемента в множестве, добавления и удаления элементов, операции объединения, пересечения и вычитания множеств.

Облачные технологии

Задание 1 - Создание проекта веб-приложения в Visual Studio

В этом задании будет создан новый проект веб-приложение ASP.NET.

1. Откройте Visual Studio 2015.
2. Нажмите **Файл > Создать > Проект**.
3. В диалоговом окне New Project последовательно щелкните на **Visual C# > Internet > ASP.NET Web Application**.
4. Убедитесь, что выбрана версия .NET Framework 4.5.2.
5. Azure Application Insights. Флажок "Add Application Insights to Project" будет установлен по умолчанию, если веб-проект создается впервые после установки Visual Studio. Если вы не хотите использовать Application Insights, снимите этот флажок.
6. Назовите приложение **MyExample** и нажмите **OK**.

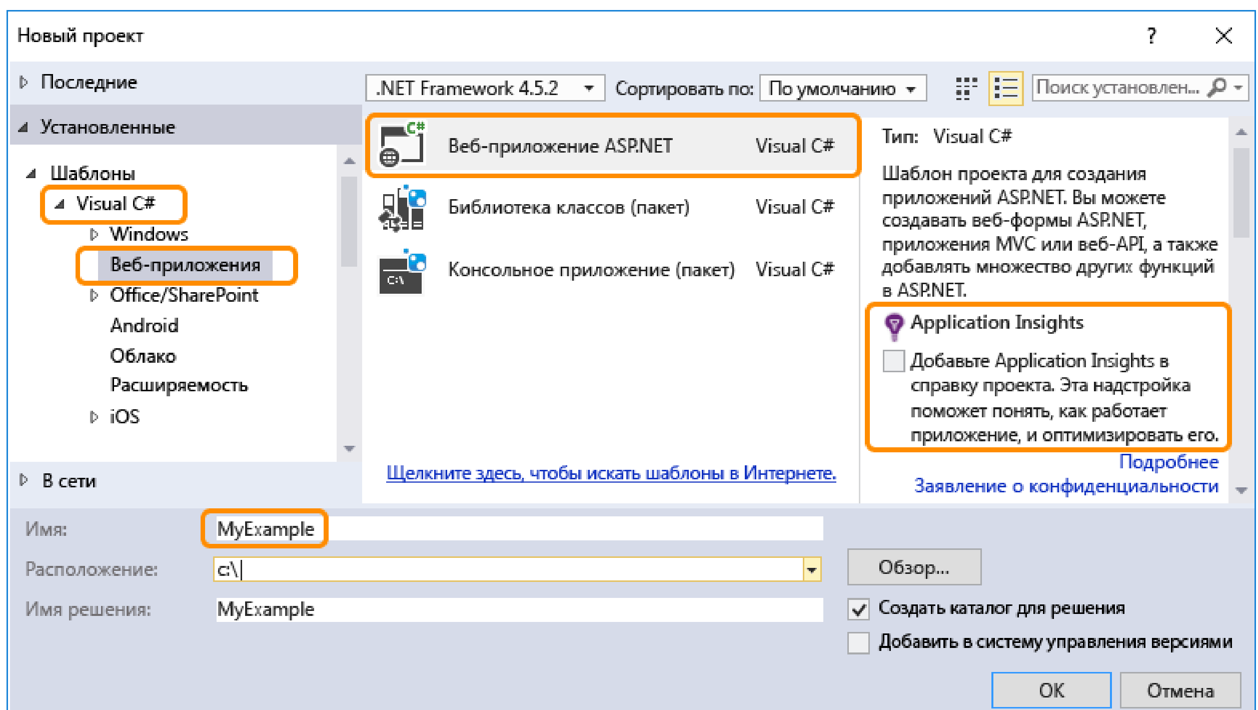


Рисунок 2.1. Модель проекта веб-приложения ASP.NET

7. В диалоговом окне **New ASP.NET Project** выберите модель **MVC**, а затем нажмите **Change Authentication Method**.

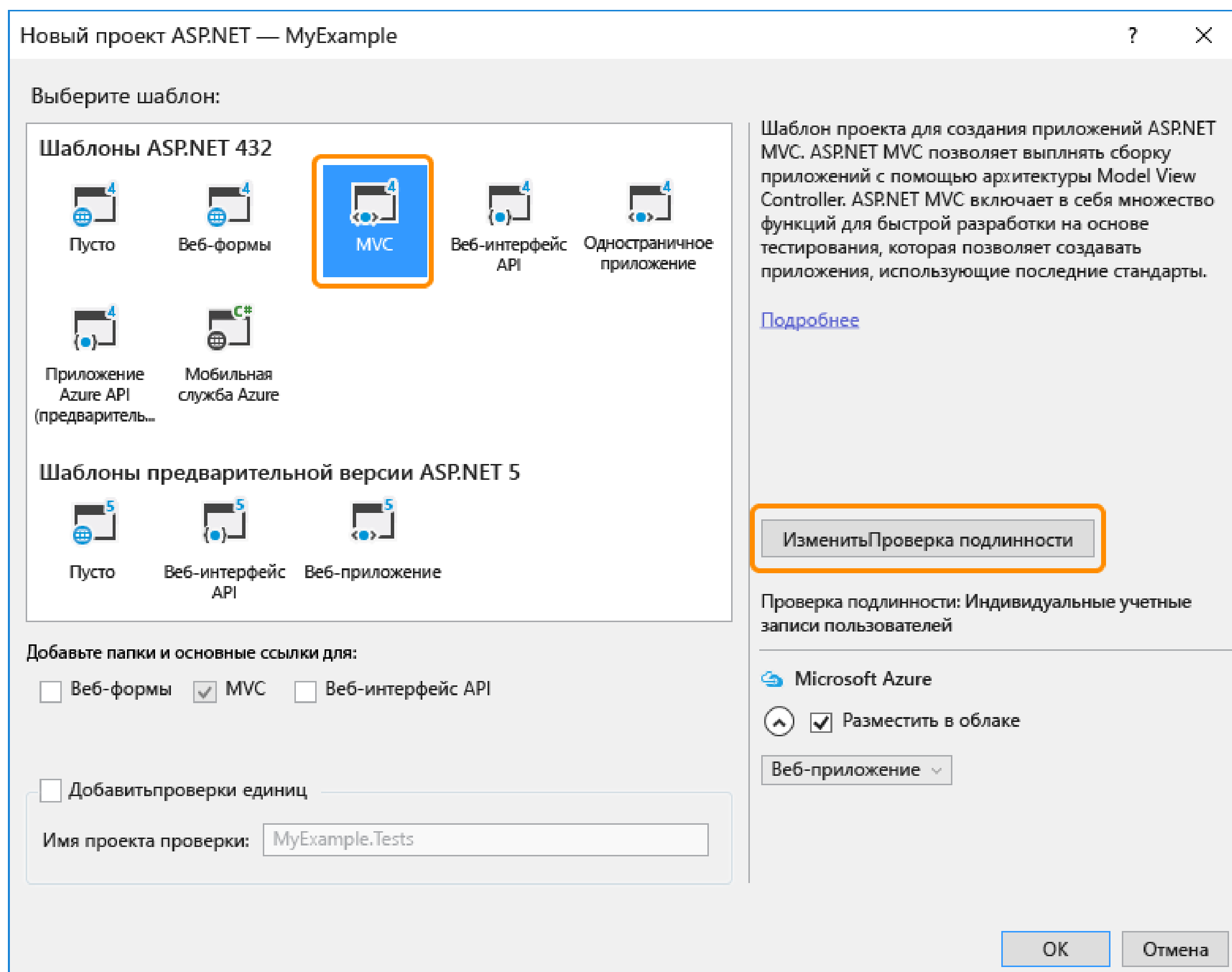


Рисунок 2.2. новый проект ASP.NET

8. В диалоговом окне **Изменение метода аутентификации** установите флажок **Без аутентификации** и нажмите **OK**.

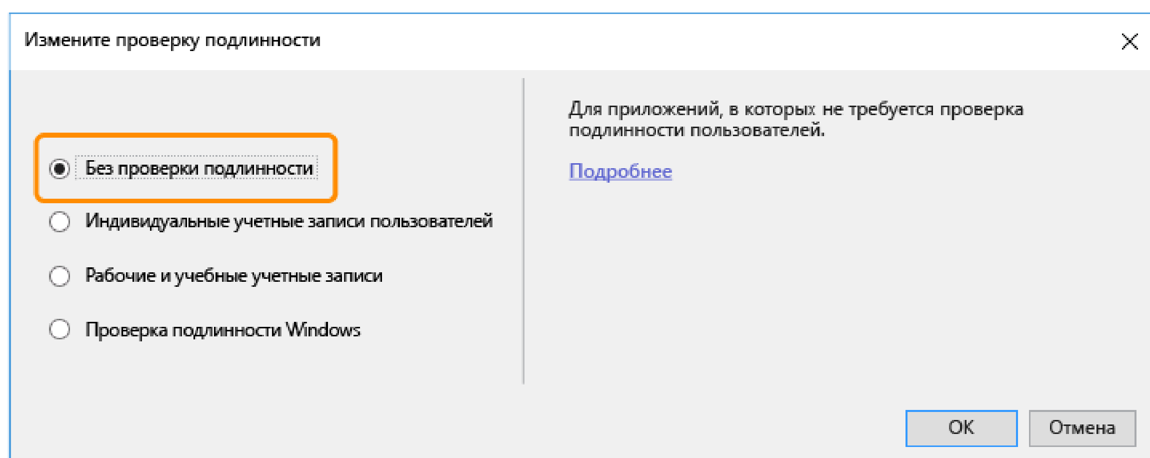


Рисунок 2.3. Окно изменения метода аутентификации

9. В диалоговом окне **New ASP.NET Project** в разделе **Microsoft Azure** установите флажок **Place in the cloud** и из выпадающего списка выберите **Application Service**.

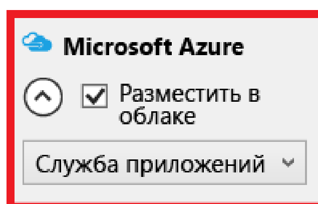


Рисунок 2.4. Опция Microsoft Azure

Эти опции указывают, что Visual Studio создаст веб-приложение Azure для вашего веб-проекта.

10. Нажмите кнопку **OK**.

Задание 2 - Создание ресурсов Azure

В этом задании вы укажете ресурсы Azure, которые хотите создать.

1. В диалоговом окне **Создание службы приложений** нажмите **Добавить учетную запись**, а затем войдите в Azure с идентификатором и паролем учетной записи, которую вы используете для управления подпиской Azure.

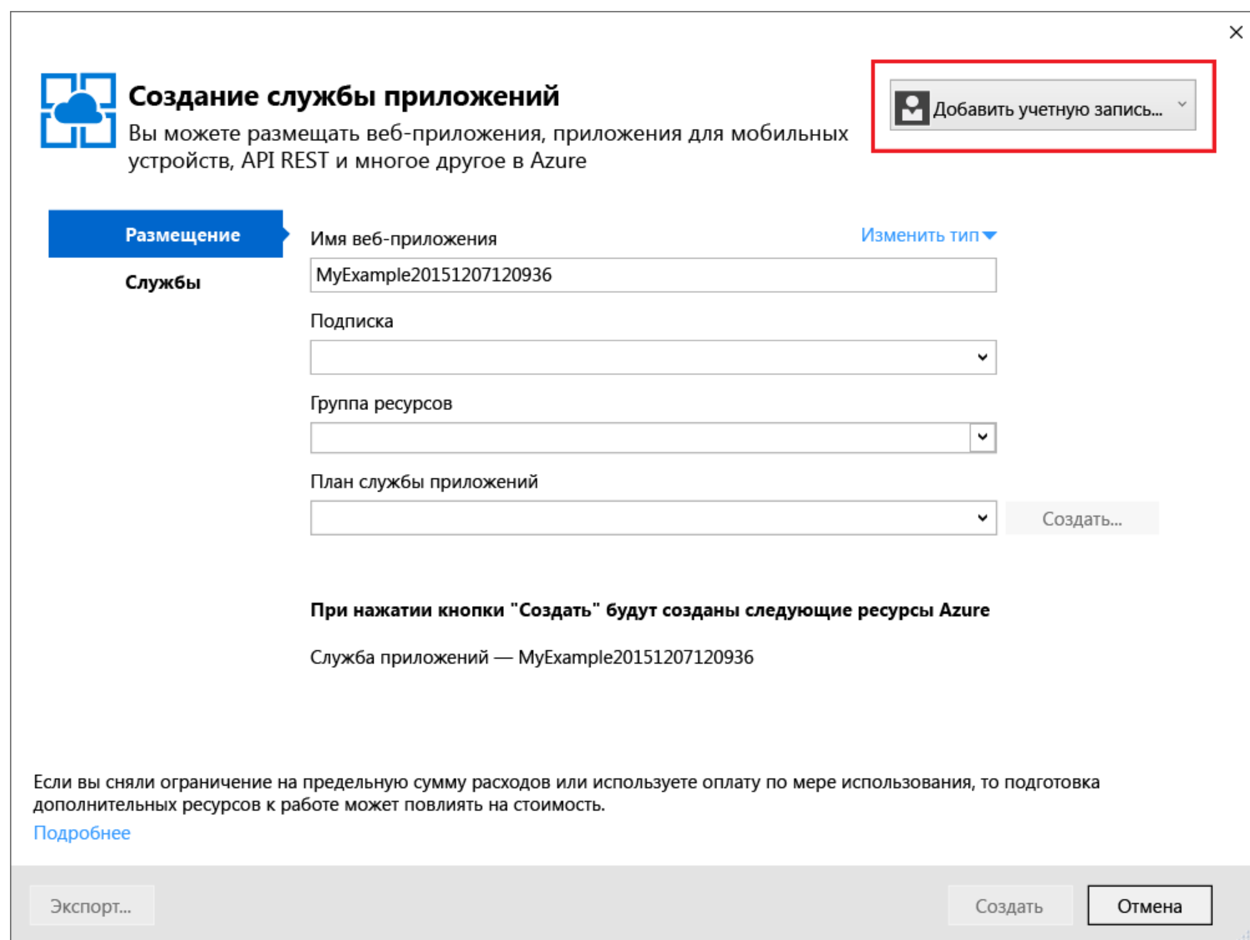


Рисунок 2.5. Экран для создания служб приложений

Если вы уже вошли в Azure на том же компьютере, вы можете не увидеть кнопку **Добавить учетную запись.** В этом случае вы можете пропустить этот шаг или повторно ввести свои учетные данные, если это необходимо.

2. В поле **Web Application Name** введите имя, уникальное для домена *azurewebsites.net*. Вы можете назвать приложение MyExample и добавить уникальный номер справа (например, MyExample810) или использовать уникальное имя, которое создается автоматически. Если кто-то уже использует введенное вами имя, вместо зеленого восклицательного знака справа появится красный восклицательный знак. Это означает, что вам необходимо ввести другое имя. URL приложения состоит из этого имени и домена *.azurewebsites.net*. Например, если имя MyExample810, URL будет следующим: myexample810.azurewebsites.net.

3. Нажмите кнопку **New** рядом с полем **Resource Group** и введите имя MyExample (или любое другое имя).

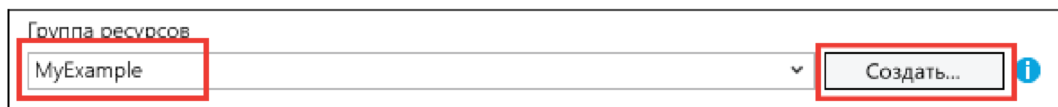


Рисунок 2.6. Создание группы ресурсов

Группа ресурсов — это набор ресурсов Azure, таких как веб-приложения, базы данных и виртуальные машины. Рекомендуется создать новую группу ресурсов для этого задания, чтобы все ресурсы Azure, созданные для работы с этим приложением, могли быть удалены одновременно.

4. **Нажмите** кнопку **New** рядом с выпадающим списком **Application Service Plan**.

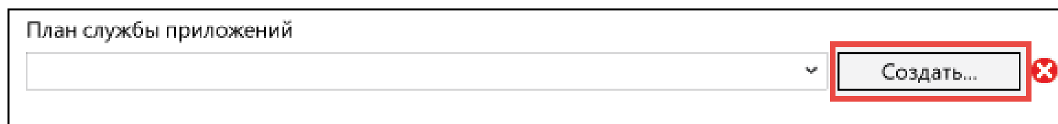


Рисунок 2.7. Создание плана обслуживания приложений

Появится диалоговое окно **Configure Application Service Plan**.

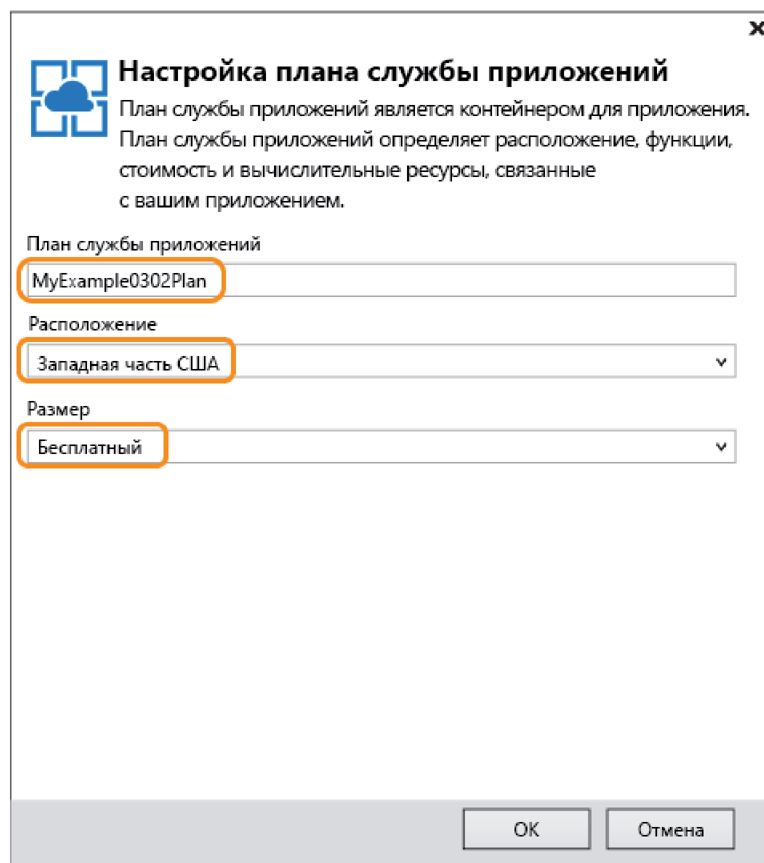


Рисунок 2.8. Экран конфигурации плана обслуживания приложений

План обслуживания приложения определяет компьютерные ресурсы, на которых будет работать ваше веб-приложение. Например, если вы выберете "бесплатный" уровень, ваше приложение будет работать на общих виртуальных машинах, а если вы выберете некоторые платные уровни, ваше приложение будет работать на выделенных виртуальных машинах.

5. В диалоговом окне **Configure Application Service Plan** введите MyExamplePlan или другое имя в соответствующее поле.
6. В раскрывающемся списке **Местоположение** выберите ближайшее местоположение. Этот параметр определяет, в каком центре обработки данных Azure будет выполняться приложение. Для работающего приложения необходимо, чтобы сервер располагался как можно ближе к клиентам, обращающимся к нему. Это позволит минимизировать задержку.
7. В раскрывающемся списке **Размер** выберите **Свободный**.
8. В диалоговом окне **Configure Application Service Plan** нажмите **OK**.
9. В диалоговом окне **Создание службы приложений** нажмите кнопку **Новый**.

Задание 3 - Проверка ресурсов Azure в Visual Studio

В этом задании вы должны проверить правильность созданного вами приложения.

1. В окне **Solution Browser** отображаются файлы и папки нового проекта.

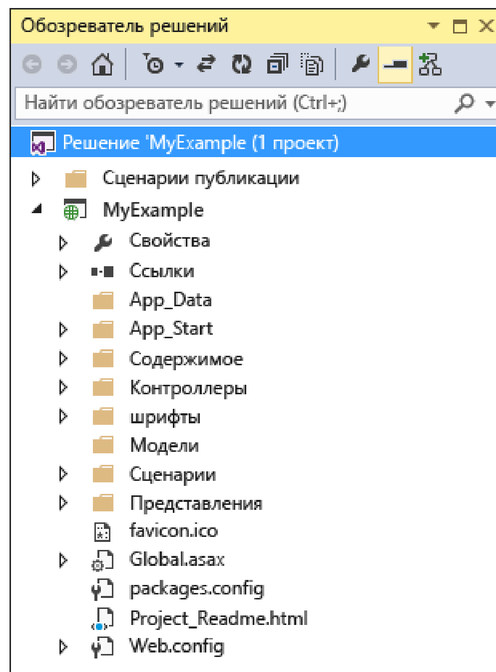


Рисунок 2.9. Окно браузера решения MyExample

2. В окне **Azure Application Service Action** будет показано, что ресурсы службы приложений были созданы в Azure.

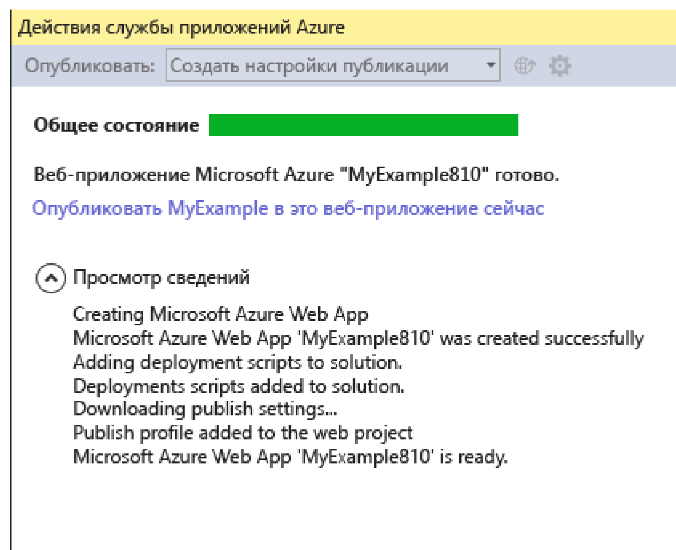


Рисунок 2.10. Окно действий службы приложений Azure

3. В **Cloud Explorer** вы можете просматривать и управлять своими ресурсами Azure, включая только что созданное веб-приложение.

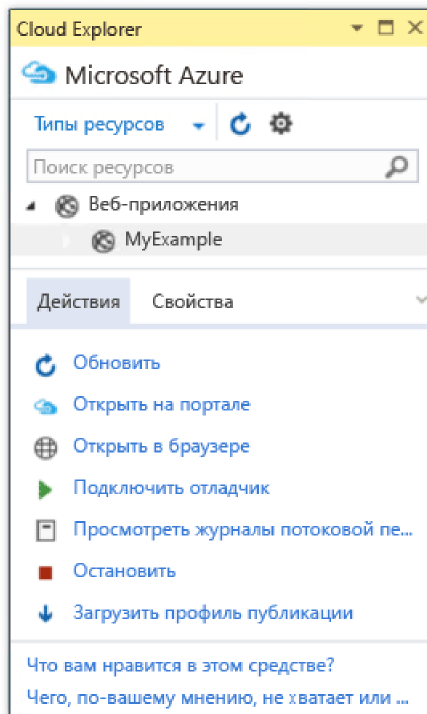


Рисунок 2.11. Экран Cloud Explorer

Задание 4 - Реализация веб-проекта в Azure

В этом задании веб-проект будет развернут на ресурсе веб-приложения, созданном в Azure Application Service.

1. В **обозревателе решений** щелкните правой кнопкой мыши на проекте и выберите **Publish**.

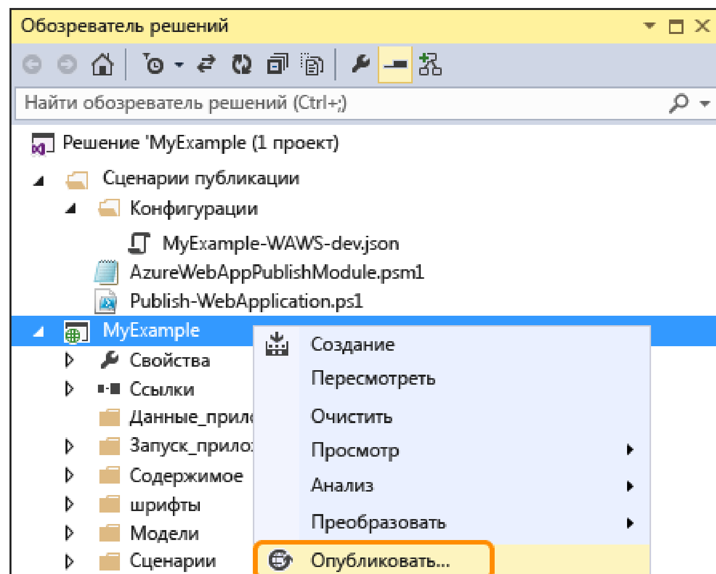


Рисунок 2.12. Пункт меню "Публикация

Затем откроется окно **мастера веб-публикации,** в котором будет отображен *профиль публикации,* содержащий параметры для развертывания веб-проекта в новом веб-приложении.

Обратите внимание: профиль публикации содержит имя пользователя и пароль для развертывания. Эти учетные данные создаются автоматически. Вводить их не обязательно. Пароль хранится в зашифрованном виде в скрытом пользовательском файле в папке Properties.

2. На вкладке **Подключение** мастера **публикации веб-сайта** нажмите **Далее**.

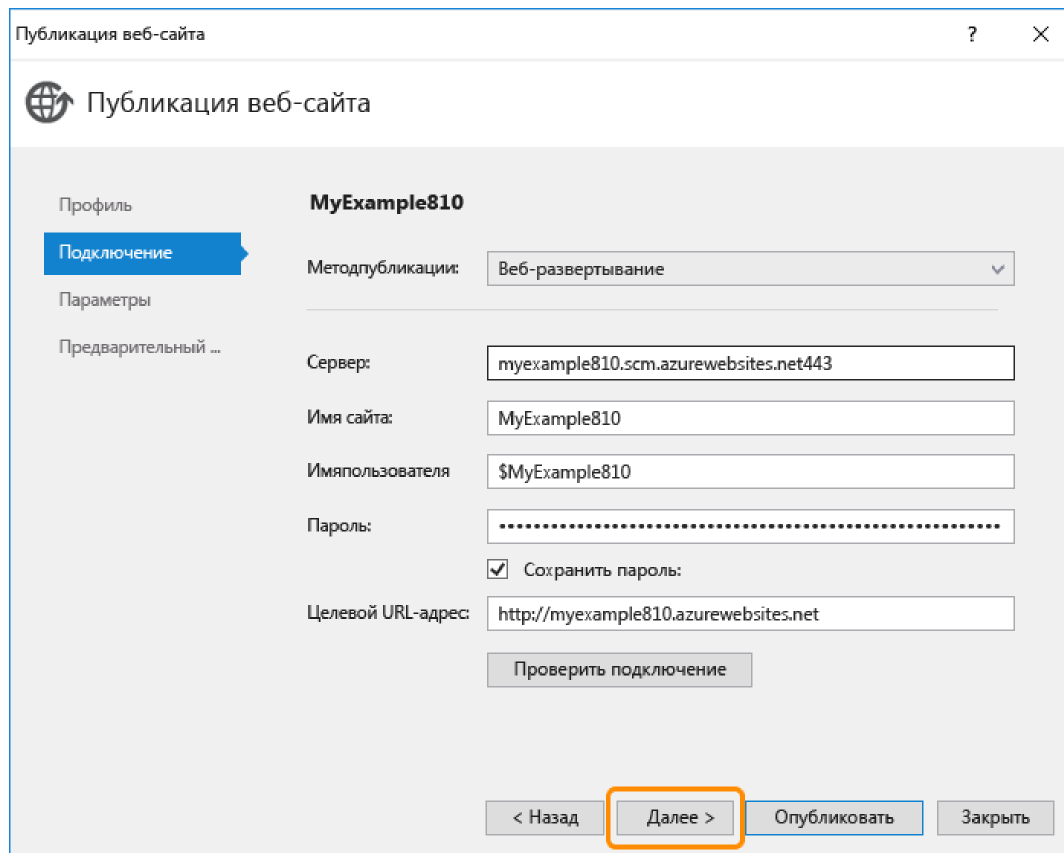


Рисунок 2.13. Окно публикации веб-сайта: Вкладка входа в систему

3. На вкладке **Параметры** нажмите **Далее**.

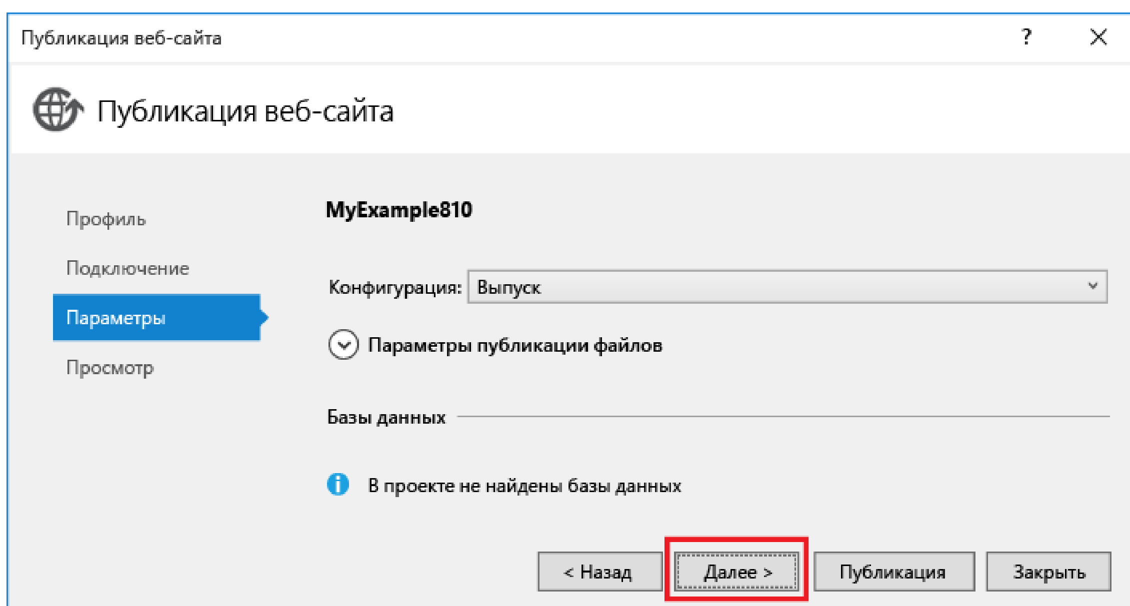


Рисунок 2.14. Окно публикации веб-сайта: Вкладка Параметры

4. На вкладке **Предварительный просмотр** нажмите кнопку **Опубликовать**.

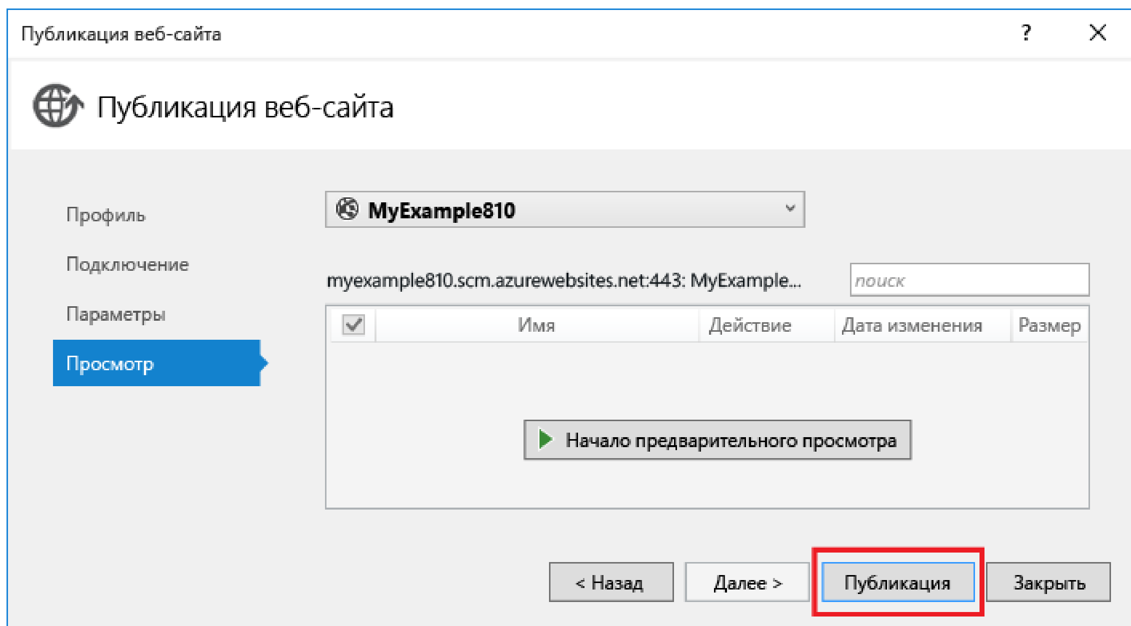


Рисунок 2.15. Окно публикации веб-сайта: Вкладка "Вид".

Когда вы нажмете кнопку **Publish**, Visual Studio начнет копировать файлы на сервер Azure. Это может занять несколько минут.

В окнах **Azure Application Service Exit** и **Actions** отображается не только информация о действиях, предпринятых во время развертывания, но и отчет об успешном завершении развертывания.

После успешного развертывания URL-адрес развернутого веб-приложения автоматически открывается в браузере по умолчанию, и созданное приложение теперь работает в облаке. URL в адресной строке браузера указывает на то, что веб-приложение загружается из Интернета.

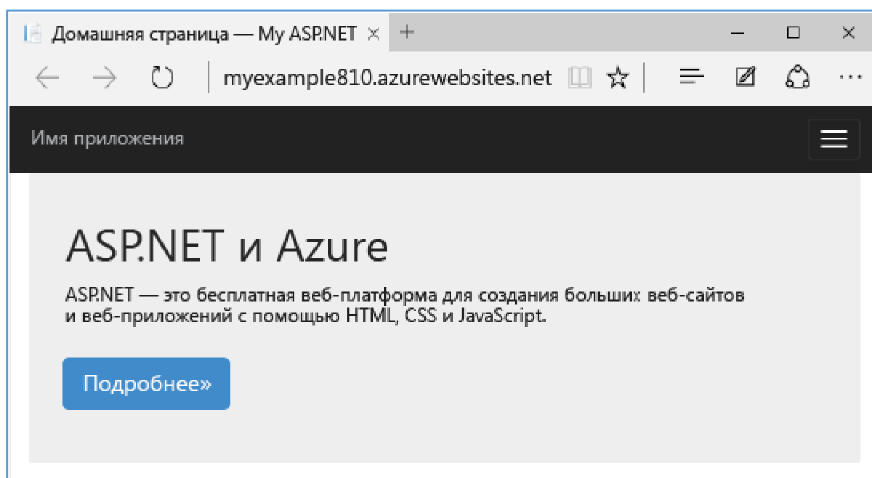


Рисунок 2.16. Результат успешной публикации

Мобильные технологии

Типы коллекций

Стандартная библиотека Kotlin предоставляет исчерпывающий набор инструментов для управления коллекциями – группами с переменным количеством элементов (возможно, нулевых), которые имеют общее значение для решаемой проблемы.

Коллекции являются общей концепцией для большинства языков программирования. Они обычно содержит несколько объектов (это число также может быть нулевым) одного и того же типа. Объекты в коллекции называются элементами (*elements*) или объектами (*items*). Например, все студенты в общежитии составляют коллекцию, которую можно использовать для расчета их среднего возраста. Следующие типы коллекций актуальны для Kotlin:

– *List (список)* — это упорядоченная коллекция с доступом к элементам по индексам – целым числам, отражающим их положение. Элементы могут встречаться в списке более одного раза. Примером списка является предложение: это группа слов, их порядок важен, и они могут повторяться;

– *Set (множество)* — представляет собой коллекцию уникальных элементов. Он отражает математическую абстракцию множества: группа объектов без повторений. Как правило, порядок набора элементов не имеет значения. Например, алфавит — это набор букв;

– *Map (карта или словарь)* — представляет собой набор пар ключ-значение. Ключи уникальны, и каждый из них соответствует ровно одному значению. Значения могут быть повторяющимися. Map полезны для хранения логических связей между объектами, например, идентификатором сотрудника и его положением.

Kotlin позволяет управлять коллекциями независимо от того, какой тип объектов хранится в них. Другими словами, вы добавляете String в список Strings точно так же, как если бы вы использовали Ints или пользовательский класс. Таким образом, стандартная библиотека Kotlin предлагает универсальные интерфейсы, классы и функции для создания, заполнения и управления коллекциями любого типа.

Интерфейсы коллекции и связанные функции находятся в пакете *kotlin.collections*. Давайте рассмотрим его содержание.

Стандартная библиотека Kotlin предоставляет реализации для базовых типов коллекций: *Set*, *List* и *Map*. Пара способов создания представляет каждый тип коллекции:

– *read-only* (только для чтения), который предоставляет операции для доступа к элементам коллекции.

– *mutable* (изменяемый), который расширяет соответствующий способов создания только для чтения операциями записи: добавление, удаление и обновление элементов.

> **Обратите внимание**
>
> Изменение *mutable* коллекции не требует, чтобы она была *var*: операции записи изменяют один и тот же объект *mutable* коллекции, поэтому ссылка не изменяется. Хотя, если вы попытаетесь переназначить коллекцию *val*, вы получите ошибку компиляции.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.add("five")                               // Все хорошо
    //numbers = mutableListOf("six", "seven")         // Ошибка компиляции
}
```

Ниже на рисунке 2.17 приведена схема интерфейсов коллекции Kotlin.

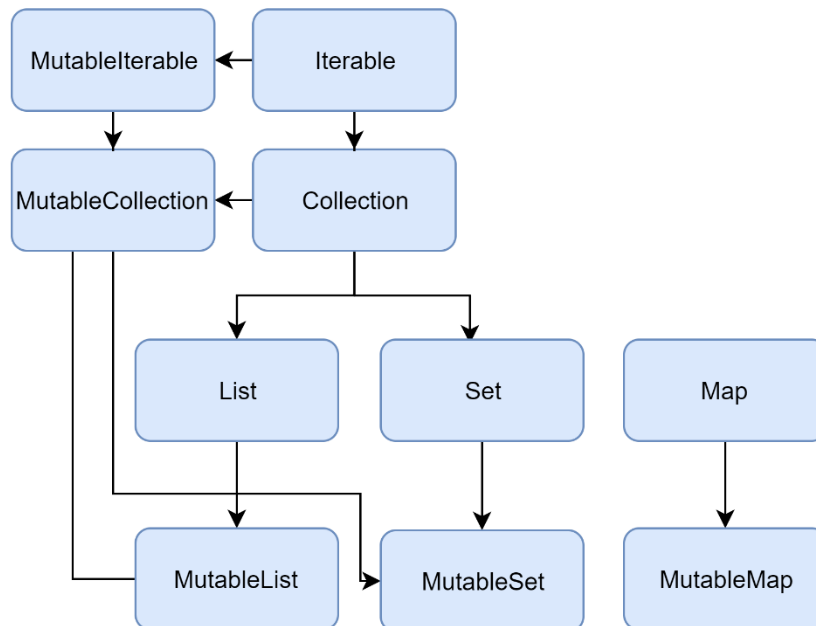


Рисунок 2.17. Иерархия интерфейсов коллекции

Давайте пройдемся по интерфейсам и их реализациям.

Коллекция

Collection<T> является корнем иерархии коллекции. Этот интерфейс представляет общее поведение коллекции только для чтения: получение размера, проверка членства элемента и т. д. Коллекция наследуется от интерфейса *Iterable<T>*, который определяет операции для итерации элементов. Вы можете использовать коллекцию как параметр функции, которая применяется к различным типам коллекций. Для более конкретных случаев используйте наследники *Collection*: *List* и *Set*.

```
fun printAll(strings: Collection<String>) {
    for(s in strings)
        print("$s ")
    println()
}

fun main() {
    val stringList = listOf("one", "two", "one")
    printAll(stringList)

    val stringSet = setOf("one", "two", "three")
    printAll(stringSet)
}
```

MutableCollection – это коллекция с операциями записи, такими как добавление и удаление.

```
fun List<String>.getShortWordsTo(shortWords: MutableList<String>, maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength }
    // Убираем артикли
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}

fun main() {
    val words = "A long time ago in a galaxy far far away".split(" ")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
    println(shortWords)
}
```

Список (List)

List<T> хранит элементы в указанном порядке и обеспечивает индексированный доступ к ним. Индексы начинаются с нуля – индекса первого элемента – и переходят к *lastIndex*, который вычисляется как *list.size - 1*.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    println("Количество элементов: ${numbers.size}")
    println("Третий элемент: ${numbers.get(2)}")
    println("Четвертый элемент: ${numbers[3]}")
    println("Индекс элемента \"два\" ${numbers.indexOf("two")}")
}
```

Элементы списка (включая нули) могут дублироваться: список может содержать любое количество одинаковых объектов или вхождений одного объекта. Два списка считаются равными, если они имеют одинаковые размеры и структурно равные элементы в одинаковых позициях.

```
class Person(var s: String, var age: Int)

fun main() {
    val boris = Person("Борис", 31)
    val people = listOf<Person>(Person("Алексей", 20), boris, boris)
    val people2 = listOf<Person>(Person("Алексей", 20), Person("Борис", 31), boris)
    println(people == people2)
    boris.age = 32
    println(people == people2)
}
```

```
false
false
```

MutableList – это список с набором операций записи, например, для добавления или удаления элемента в определенной позиции.

```
fun main() {
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    numbers.removeAt(1)
    numbers[0] = 0
    numbers.shuffle() /*Случайно перемешивает элементы в списке*/
    println(numbers)
}
```

```
[5, 0, 4, 3]
```

Как видите, в некоторых аспектах списки очень похожи на массивы. Однако есть одно важное отличие: размер массива определяется при инициализации и никогда не изменяется; в свою очередь, список не имеет предопределенного размера; размер списка может быть изменен в результате операций записи: добавления, обновления или удаления элементов.

В Kotlin реализацией *List* по умолчанию является *ArrayList*, который можно рассматривать как массив с изменяемым размером.

Множество (Set)

Set<T> хранит уникальные элементы; их порядок, как правило, не определен.

Нулевые элементы также уникальны: набор может содержать только один ноль. Два набора равны, если они имеют одинаковый размер, и для каждого элемента набора есть равный элемент в другом наборе.

```
fun main() {
    val numbers = setOf(1, 2, 3, 4)
    println("Количество элементов: ${numbers.size}")
    if (numbers.contains(1))
        println("1 принадлежит множеству")

    val numbersBackwards = setOf(4, 3, 2, 1)
    println("Множества равны: ${numbers == numbersBackwards}")
}
```

MutableSet – это множество с операциями записи из *MutableCollection*.

Реализация по умолчанию *Set* – *LinkedHashSet* – сохраняет порядок вставки элементов. Следовательно, функции, которые зависят от порядка, такие как *first()* или *last()*, возвращают предсказуемые результаты на таких множествах.

```
fun main() {
    val numbers = setOf(1, 2, 3, 4)
    val numbersBackwards = setOf(4, 3, 2, 1)

    println(numbers.first() == numbersBackwards.first())
    println(numbers.first() == numbersBackwards.last())
}
```

```
false
true
```

Альтернативная реализация – *HashSet* – ничего не говорит о порядке элементов, поэтому вызов таких функций для данной реализации возвращает непредсказуемые результаты. Однако *HashSet* требует меньше памяти для хранения того же количества элементов.

Операции над двумя множествами

Есть три операции, которые из двух множеств делают новое множество: объединение, пересечение, разность.

Операция объединение приведена на рисунке 2.18.

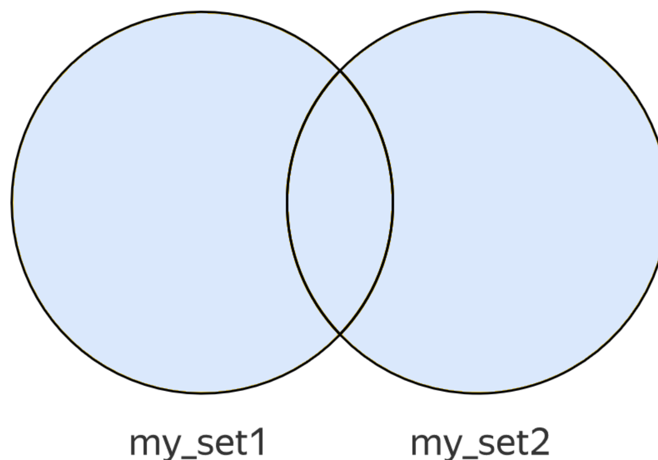


Рисунок 2.18. Операция объединение множеств

Объединение двух множеств включает в себя все элементы, которые есть хотя бы в одном из них. Для этой операции существует метод *union*:

```
val union = my_set1 union my_set2
```

Или можно использовать оператор +:

```
val union = my_set1 + my_set2
```

Операция пересечение приведена на рисунке 2.19.

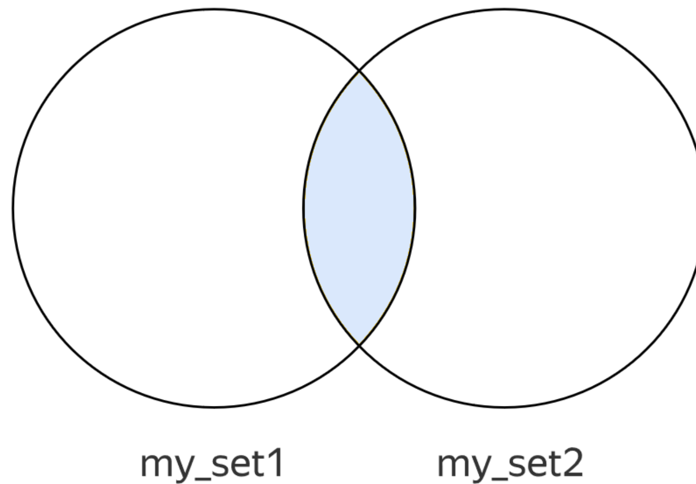


Рисунок 2.19. Операция пересечение множеств

Пересечение двух множеств включает в себя все элементы, которые есть в обоих множествах:

```
val intersection = my_set1 intersect my_set2
```

Операция разность приведена на рисунке 2.20.

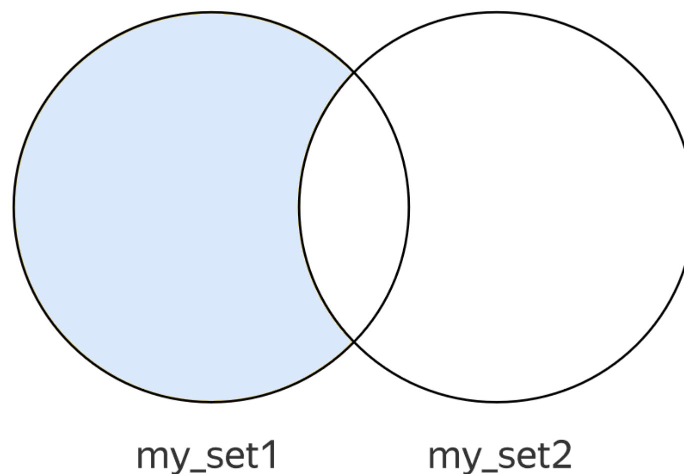


Рисунок 2.20. Операция разность множеств

Разность двух множеств включает в себя все элементы, которые есть в первом множестве, но которых нет во втором:

```
val diff_2 = my_set1 subtract my_set2
```

Или аналог:

```
val diff = my_set1 - my_set2
```

Задание 5 – Дан список *List* из 10 целых чисел. Составить программу нахождения суммы максимального и минимального элементов.

Блок–схема решения задачи приведена на рисунке 2.21.

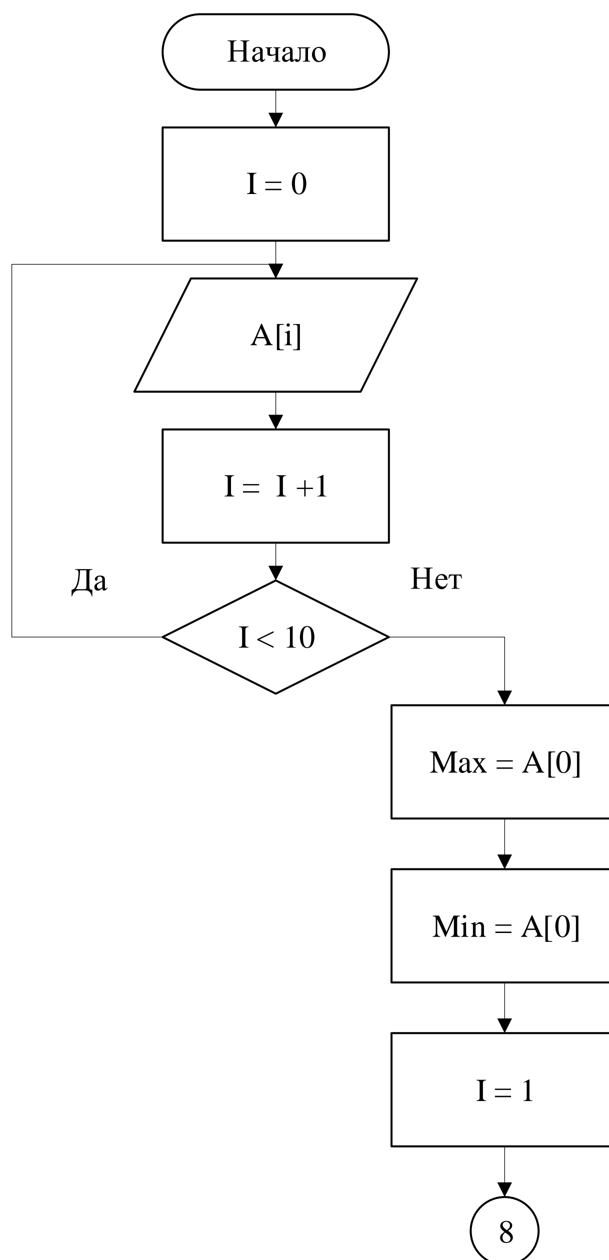
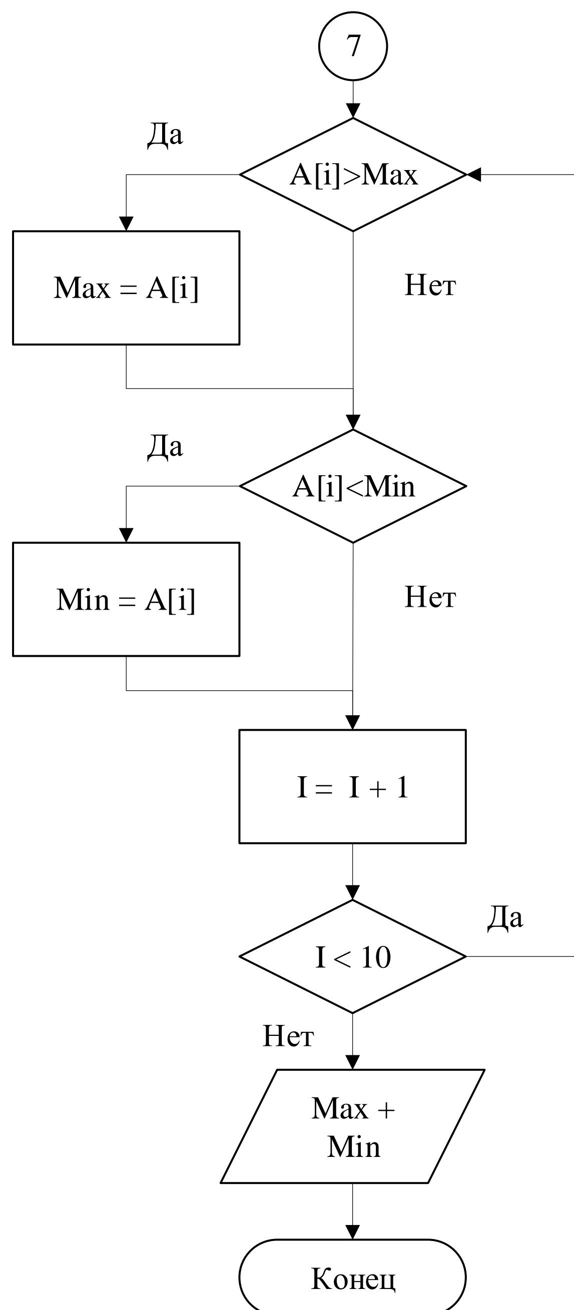


Рисунок 2.21. Блок–схема решения задачи 5



Продолжение рисунка 2.21

Порядок выполнения:

a. Создание нового Kotlin файла

Добавьте в проект новый файл, щелкнув правой кнопкой мыши на папке **src** в окне инструментов проекта. Выберите New – Kotlin File/Class. Ввести имя файла, например, Task02_1. Нажать кнопку «Enter».

b. Написание кода основной программы

В появившемся окне необходимо ввести программный код решения задачи, приведенный ниже.

Листинг 2.1. Программный код решения задачи 5

```
import java.util.*

fun main() {
    // Объект считывания данных с консоли
    val sc = Scanner(System.`in`)
    val A = mutableListOf<Int>() // Создание списка
    println("Введите последовательность из 10 цифр ")
    // Цикл заполнения списка A
    for (i in 0 until 10){
        A.add(sc.nextInt())
    }
    var max = A[0]
    var min = A[0]
    // Цикл перебора элементов списка A
    for (element in A){
        if (element > max) {
            max = element
        }
        if (element < min) {
            min = element
        }
    }
    println("Сумма Макс и Мин элементов списка = ${max + min}")
}
```

c. Построение проекта

После ввода программного кода нужно скомпилировать и отладить программу. Для этого необходимо выполнить Build – Build Project. Если в программном коде имеются ошибки, они будут выделены красным цветом.

d. Запуск программы

Чтобы просмотреть результат выполнения программы, нужно выполнить Run – Run ‘Task02_1Kt’ или нажать кнопку в виде треугольника (рисунок 2.22).

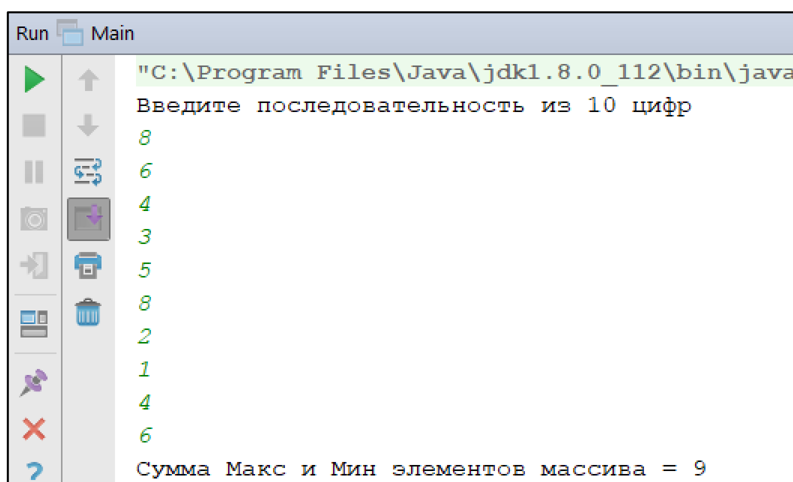


Рисунок 2.22. Результат решения задачи 5

Задание 6 – Дан список List из N целых чисел. Составить программу нахождения суммы нечетных элементов.

Блок–схема решения задачи приведена на рисунке 2.23.

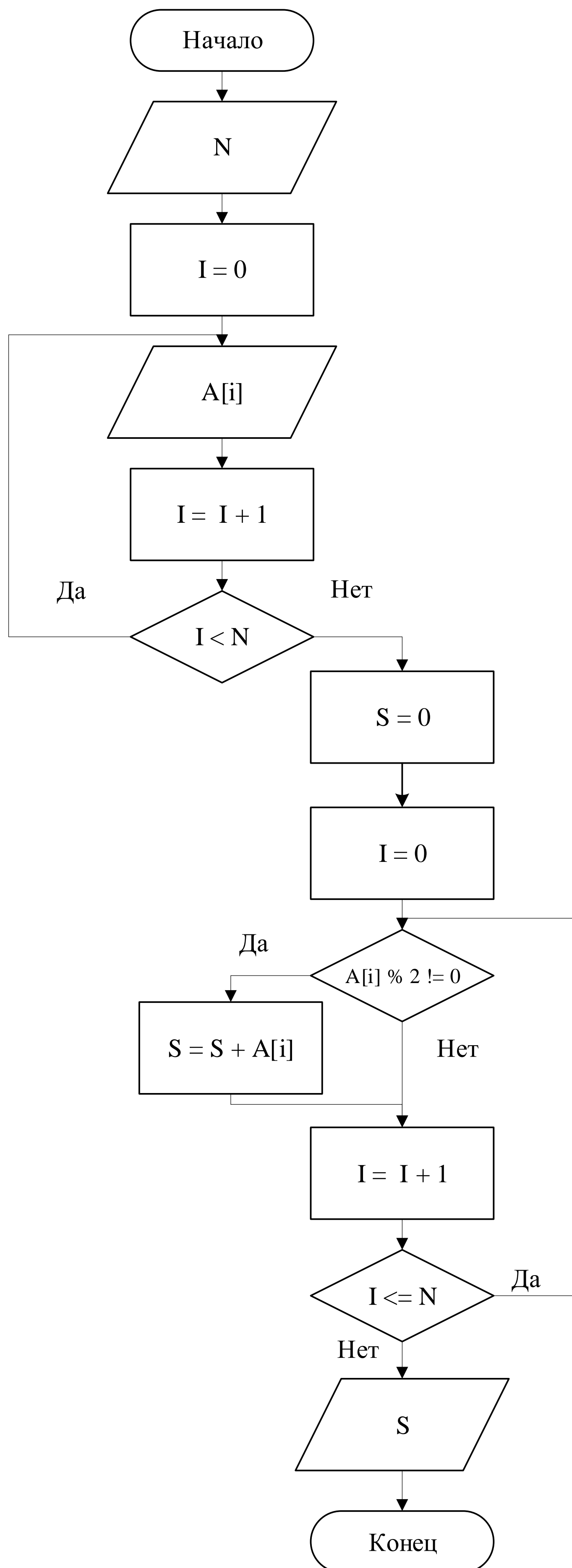


Рисунок 2.23. Блок–схема решения задачи 6

Порядок выполнения:

a. Создание нового Kotlin файла:

Добавьте в проект новый файл, щелкнув правой кнопкой мыши на папке **src** в окне инструментов проекта. Выберите New – Kotlin File/Class. Ввести имя файла, например, Task03_2. Нажать кнопку «Enter».

b. Написание кода основной программы

В появившемся окне необходимо ввести программный код решения задачи, приведенный ниже.

Листинг 2.2. Программный код решения задачи 6

```
import java.util.*

fun main() {
    // Объект считывания данных с консоли
    val sc = Scanner(System.`in`)
    val A = mutableListOf<Int>() // Создание списка
    var sum = 0
    print("Введите размерность списка: ");
    val n = sc.nextInt();
    println("Введите последовательность из $n-ти цифр:");
    // цикл заполнения списка A
    for (i in 0 until n){
        A.add(sc.nextInt())
    }
    // Цикл перебора элементов списка A
    for (element in A)
        if (element % 2 != 0)
            sum += element

    println("Сумма нечетных элементов списка = $sum")
}
```

c. Построение проекта:

После ввода программного кода нужно скомпилировать и отладить программу. Для этого необходимо выполнить Build – Build Project. Если в программном коде имеются ошибки, они будут выделены красным цветом.

d. Запуск программы

Чтобы просмотреть результат выполнения программы, нужно выполнить Run – Run ‘Task02_2Kt’ или нажать кнопку в виде треугольника (рисунок 2.24).

```
Run    Main
"C:\Program Files\Java\jdk1.8.0_112\bin\java"
Введите размерность массива 10
Введите последовательность из 10-ти цифр
1
2
3
5
7
8
9
5
4
5
Сумма нечетных элементов массива = 35
```

Рисунок 2.24. Результат решения задачи 6

Библиографический список

a. Начало работы с Облачными службами Azure (классическая версия) и ASP.NET. URL: https://docs.microsoft.com/ru-ru/azure/cloud-services/cloud-services-dotnet-get-started (Дата обращения 17.12.2021 г.)

b. Исакова С., Жемеров Д. Kotlin в действии / пер. с англ. Киселев А.Н. — М.: ДМК-Пресс, октябрь 2017 г., 402 с.

c. Скин Д., Гринхол Д. Kotlin. Программирование для профессионалов / пер. с англ. Киселев А.Н. — СПб.: Издательский дом «Питер», 2020 г., 464 с.

d. Официальная документация языка программирования Kotlin. URL: https://kotlinlang.ru/ (Дата обращения 21.10.2021 г.)

Практическая работа 3. Настройка хранилища разработки в VISUAL STUDIO 2015

В этой практической работе в части облачных технологий будет показано, как развернуть веб-приложение ASP.NET в Azure, а затем подключить его к базе данных Azure SQL, в части мобильных технологий будут продемонстрированы возможности группировки команд в функции — участки кода, которые можно использовать многократно. Рассмотрим, как можно сделать так, чтобы код функции работал по-разному в зависимости от параметров.

Облачные технологии

Задание 1 – Развернуть приложение ASP.NET, которое работает на Azure и подключается к базе данных SQL (рисунок 3.1).

Azure Web Apps — это высокомасштабируемая, самообучающаяся служба веб-хостинга.

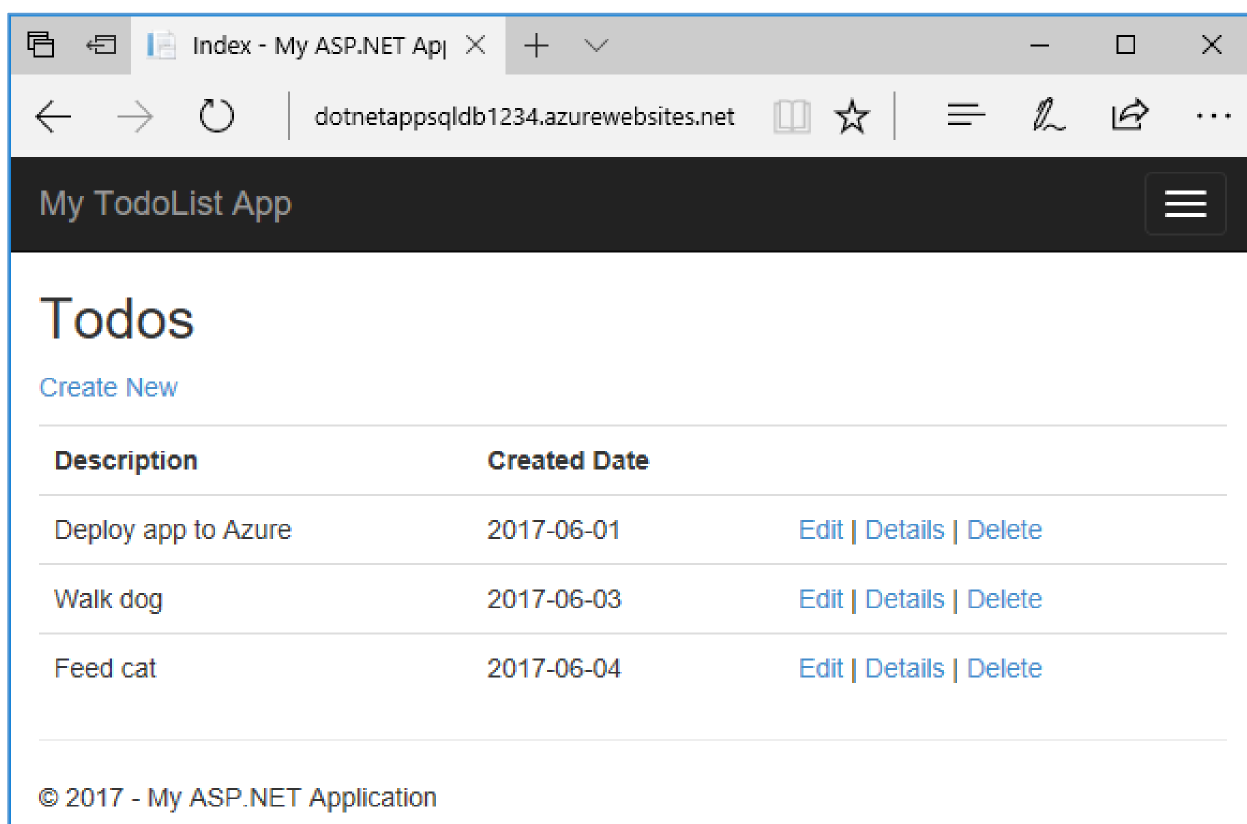


Рисунок 3.1. Приложение ASP.NET, работающее на Azure и подключенное к базе данных SQL

Скачивание примера приложения

- Скопируйте пример проекта из папки Cloud Technologies на локальный диск.
- Извлеките (разархивируйте) файл *dotnet-sqldb-tutorial-master.zip*.

Этот пример проекта содержит простое ASP.NET MVC CRUD-приложение, основанное на Entity Framework Code First.

Запуск приложения

Откройте файл *dotnet-sqldb-tutorial-master/DotNetAppSqlDb.sln* в Visual Studio.

Введите Ctrl+F5, чтобы запустить приложение без отладки. Это приложение откроется в браузере по умолчанию. Нажмите ссылку **Создать,** чтобы создать различные элементы в *списке задач* (рисунок 3.2).

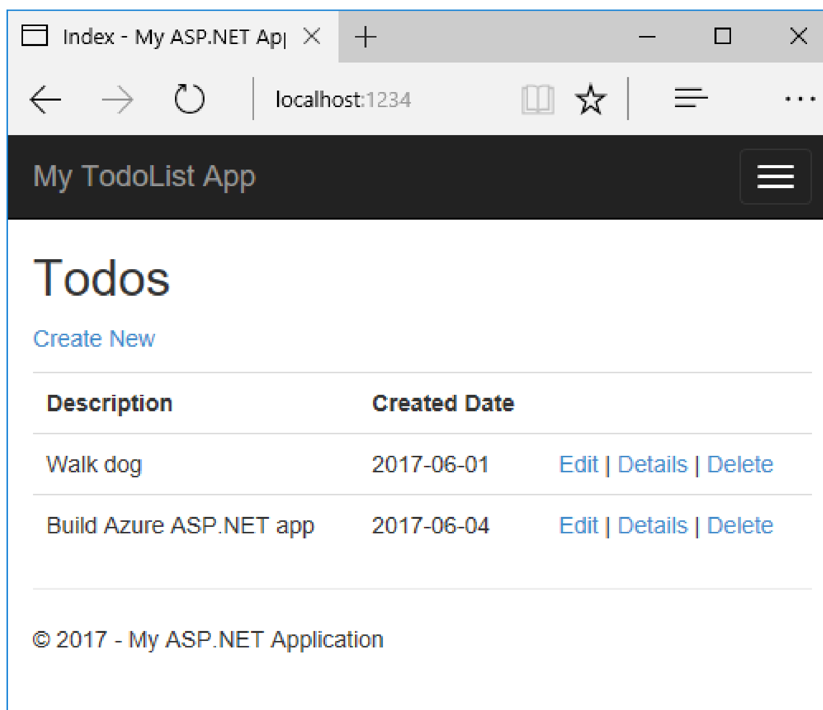


Рисунок 3.2. Добавление списка задач

Проверьте ссылки **Редактировать**, **Подробности** и **Удалить**.

Для подключения к базе данных приложение использует контекст базы данных. В этом примере контекст базы данных использует строку подключения MyDbConnection. Строка подключения определяется в файле *Web.config*. На него также ссылаются в файле *Models/MyDatabaseContext.cs*. Имя строки подключения понадобится позже в этой работе при подключении веб-приложения Azure к базе данных SQL.

Публикация в Azure с базой данных SQL

Щелкните правой кнопкой мыши проект **DotNetAppSqlDb** в **обозревателе решений** и выберите **Publish** (рисунок 3.3).

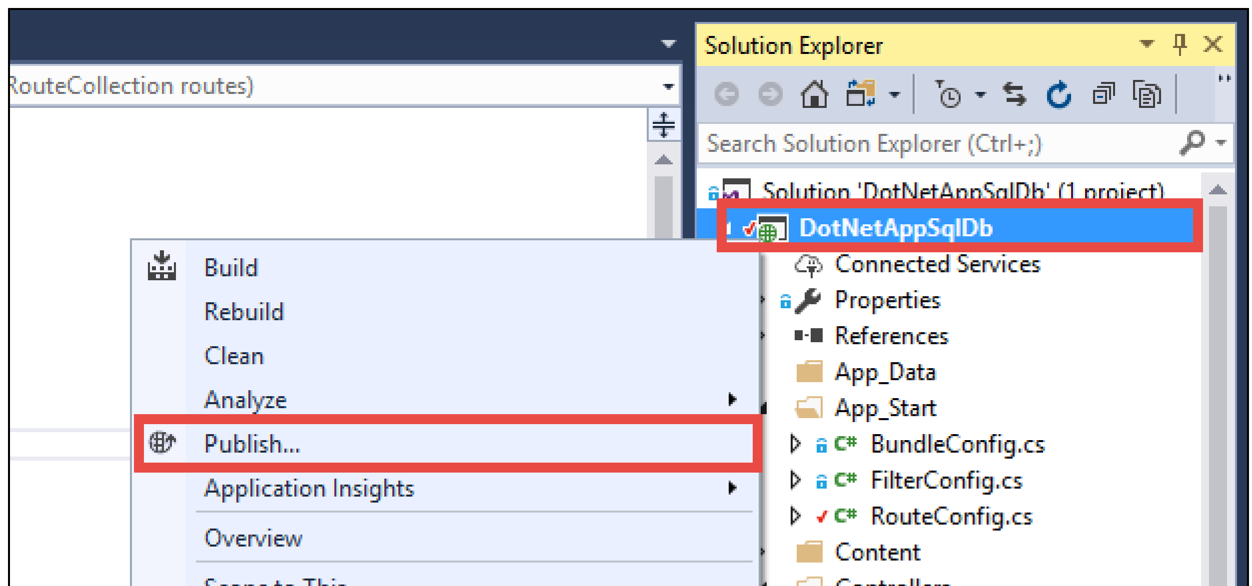


Рисунок 3.3. Публикация запроса в Azure

Выберите **Microsoft Azure Application Service** и нажмите **Publish** (рисунок 3.4).

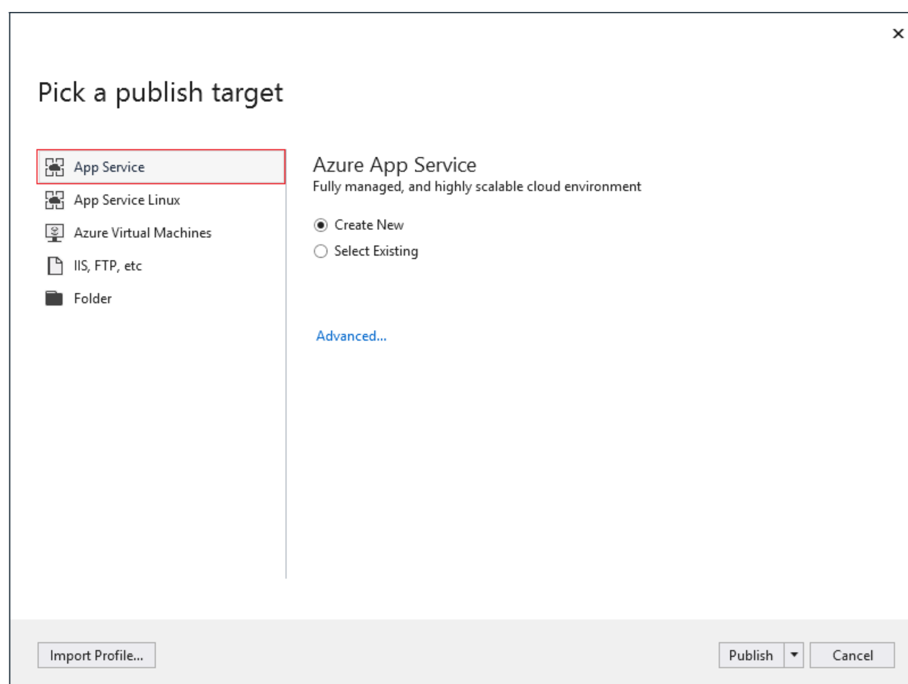


Рисунок 3.4. Элемент службы приложений Microsoft Azure

После публикации откроется диалоговое окно **Create Application Service,** которое можно использовать для создания всех ресурсов Azure, необходимых для запуска веб-приложения ASP.NET в Azure.

Войти в Azure

В диалоговом окне **Создание службы приложений** нажмите **Добавить новую учетную запись,** а затем войдите в подписку Azure. Если вы уже вошли в свою учетную запись Microsoft, убедитесь, что она содержит подписку Azure.

Если нет, нажмите на него, чтобы добавить нужную учетную запись (рисунок 3.5).

> Примечание
> Если вы уже вошли в систему, не нажимайте пока кнопку Создать.

Настройка имени веб-приложения

Вы можете использовать созданное имя веб-приложения или присвоить ему уникальное имя (допустимые символы: a-z, 0-9 и -). Это имя используется как часть URL приложения по умолчанию (<имя_приложения>.azurewebsites.net, где <имя_приложения> - имя вашего веб-приложения). Он должен быть глобально уникальным для всех приложений Azure (рисунок 3.6).

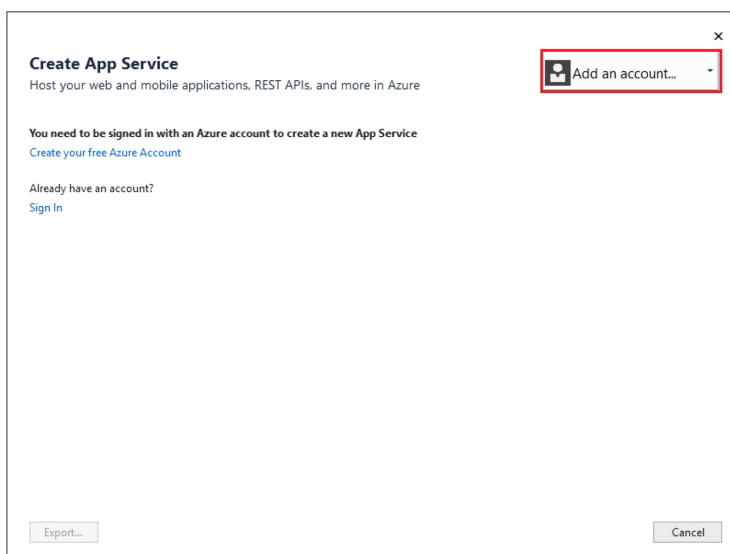


Рисунок 3.5. Окно создания службы приложения

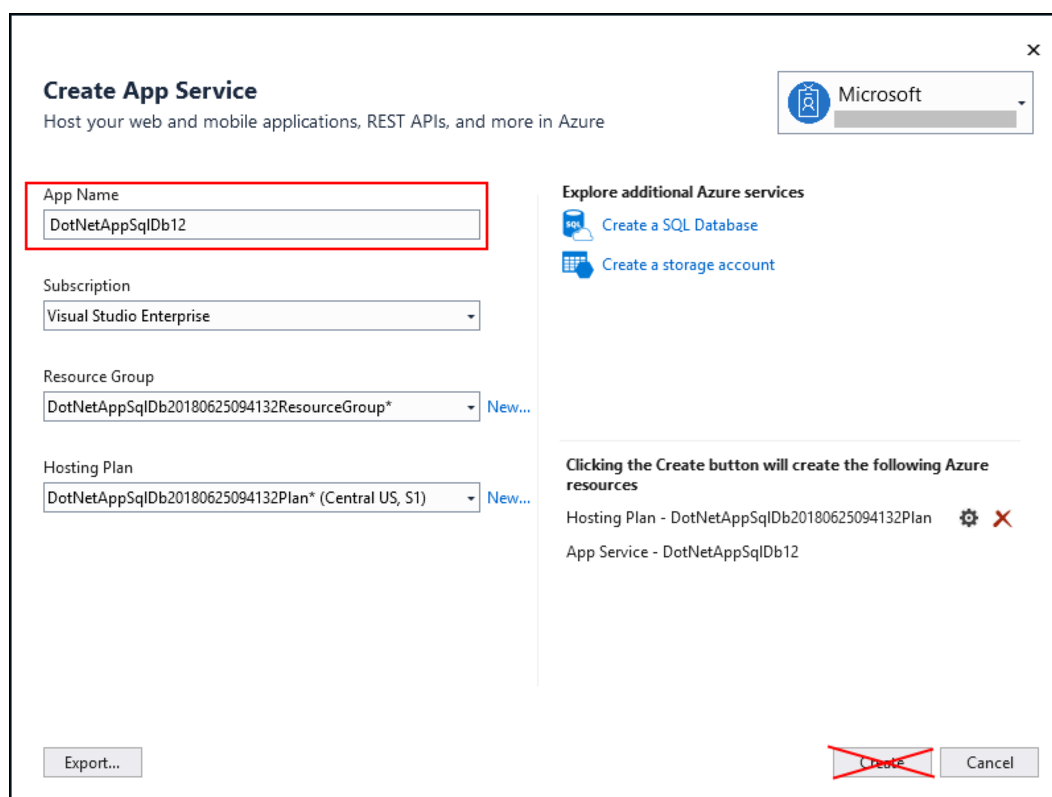


Рисунок 3.6. Настройка имени веб-приложения

Создание группы ресурсов

Пул ресурсов — это логический контейнер, в котором развернуты и управляются ресурсы Azure (веб-приложения, базы данных и учетные записи хранения). Например, впоследствии можно удалить весь пул ресурсов с помощью простого действия.

Рядом с **группой ресурсов** нажмите кнопку **Создать** (рисунок 3.7).

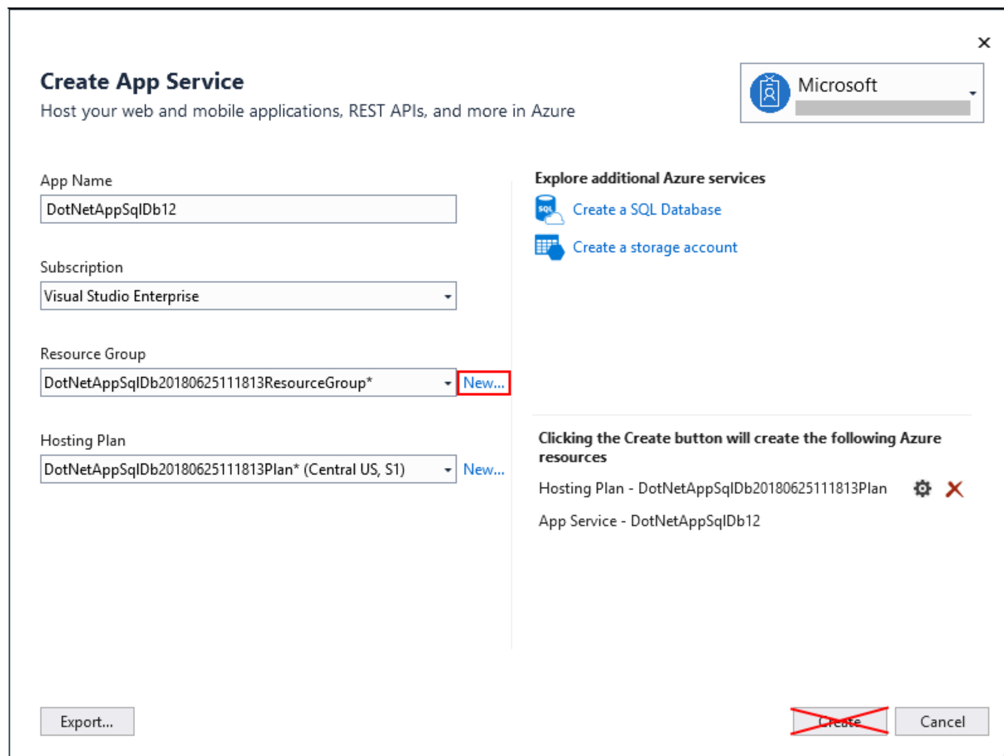


Рисунок 3.7. Создание группы ресурсов

Присвойте группе ресурсов имя **myResourceGroup**.

Создание плана обслуживания приложений

План обслуживания приложений определяет местоположение, размер и функции фермы веб-серверов, на которой размещается приложение. Вы можете сэкономить деньги при размещении нескольких приложений, настроив совместное использование одного плана обслуживания приложений веб-приложениями.

Планы обслуживания приложений определяют эти компоненты:

- регион (например, Северная Европа, Восточная часть США, Юго-Восточная Азия);
- Размер выборки (малый, средний, большой)
- Количество копий для масштабирования (от 1 до 20)
- SKU ("Free", "General", "Basic", "Standard" или "Premium").

Рядом с **планом обслуживания приложений** нажмите кнопку **Новый**.

В диалоговом окне **Configure Application** Service **Plan** настройте новый **план обслуживания приложений**, задав следующие параметры (рисунок 3.8).

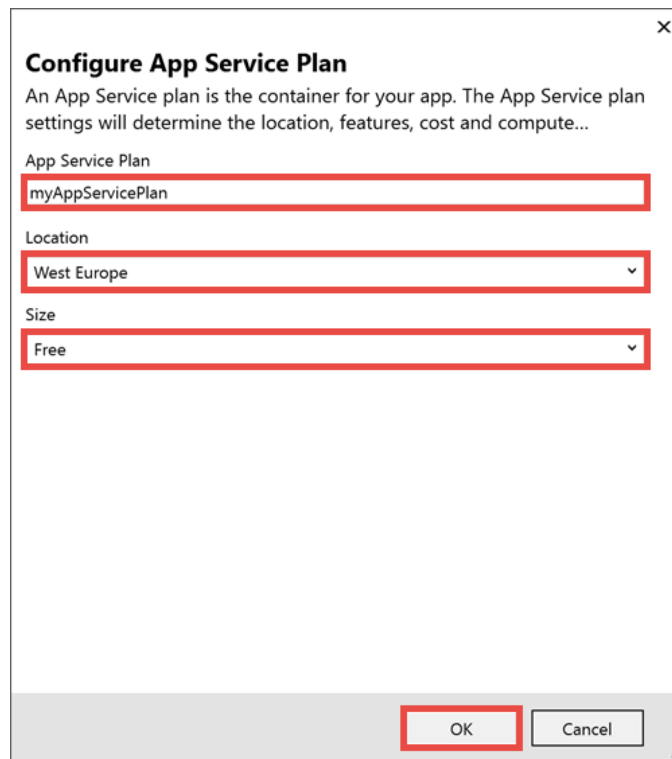


Рисунок 3.8. Создание плана обслуживания приложений

Создание экземпляра SQL Server

Перед созданием базы данных необходимо создать логический сервер Azure SQL Database Logical Server. Логический сервер содержит группу баз данных, которыми можно управлять как кластером.

Нажмите **Создать базу данных SQL** (рисунок 3.9).

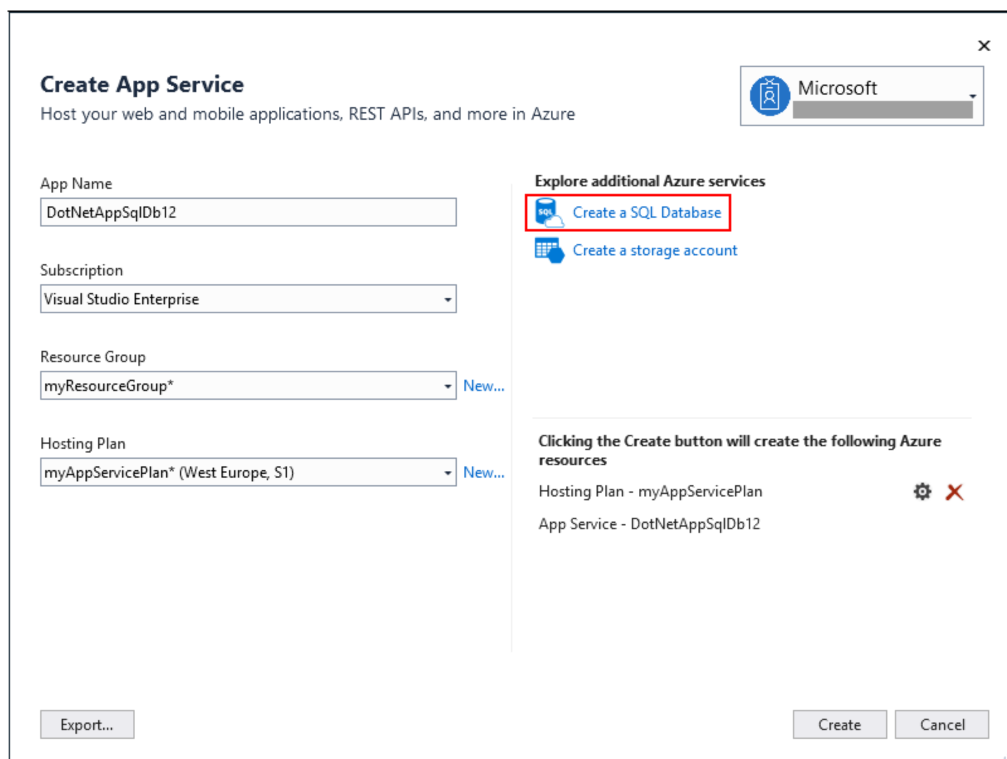


Рисунок 3.9. Создание экземпляра SQL Server

В окне **конфигурации базы данных SQL** нажмите кнопку **New** рядом с **SQL Server**.

Создается уникальное имя сервера. Это имя используется как часть URL по умолчанию для логического сервера (<имя_сервера>.base_base.windows.net). Он должен быть уникальным для всех логических экземпляров сервера в Azure. Имя сервера можно изменить, но для целей данной статьи рекомендуется использовать созданное значение (рисунок 3.10).

Добавьте имя пользователя и пароль администратора. Требования к сложности паролей см. в статье Политика паролей.

Запомните это имя пользователя и пароль. Они понадобятся вам позже для управления экземпляром логического сервера.

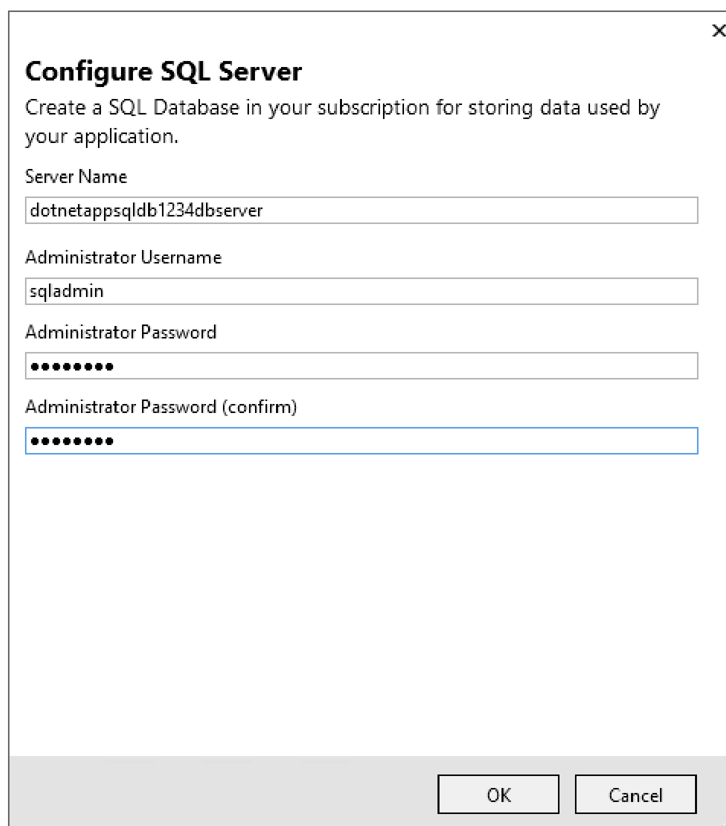


Рисунок 3.10. Конфигурация базы данных SQL

Затем выберите **OK**. Не закрывайте пока диалоговое окно **Configure SQL database**.

Важно!

Хотя пароль строки подключения скрыт (в Visual Studio и службе приложения), он все равно где-то хранится, что делает приложение уязвимым для атак. Затем служба приложения может использовать учетные данные управляемой службы, полностью устраняя необходимость хранить секреты в коде или конфигурации приложения.

Создание базы данных SQL

В диалоговом окне **Настройка базы данных SQL** (рисунок 3.11):

- в поле **Имя базы данных** не изменяйте имя, созданное по умолчанию.
- введите *MyDbConnection* в поле **Имя строки подключения**. Это имя должно совпадать с именем строки подключения, указанной в *Models\MyDatabaseContext.cs*.
- нажмите кнопку **OK**.

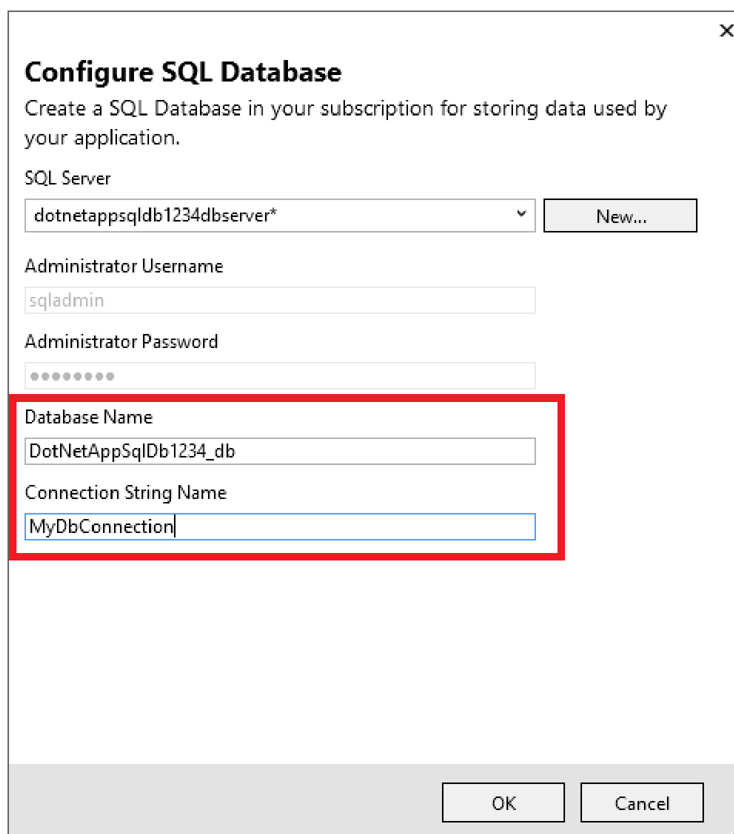


Рисунок 3.11. Конфигурация базы данных SQL

В диалоговом окне **Create Application Service** будут отображены настроенные ресурсы. Нажмите кнопку **Создать** (рисунок 3.12).

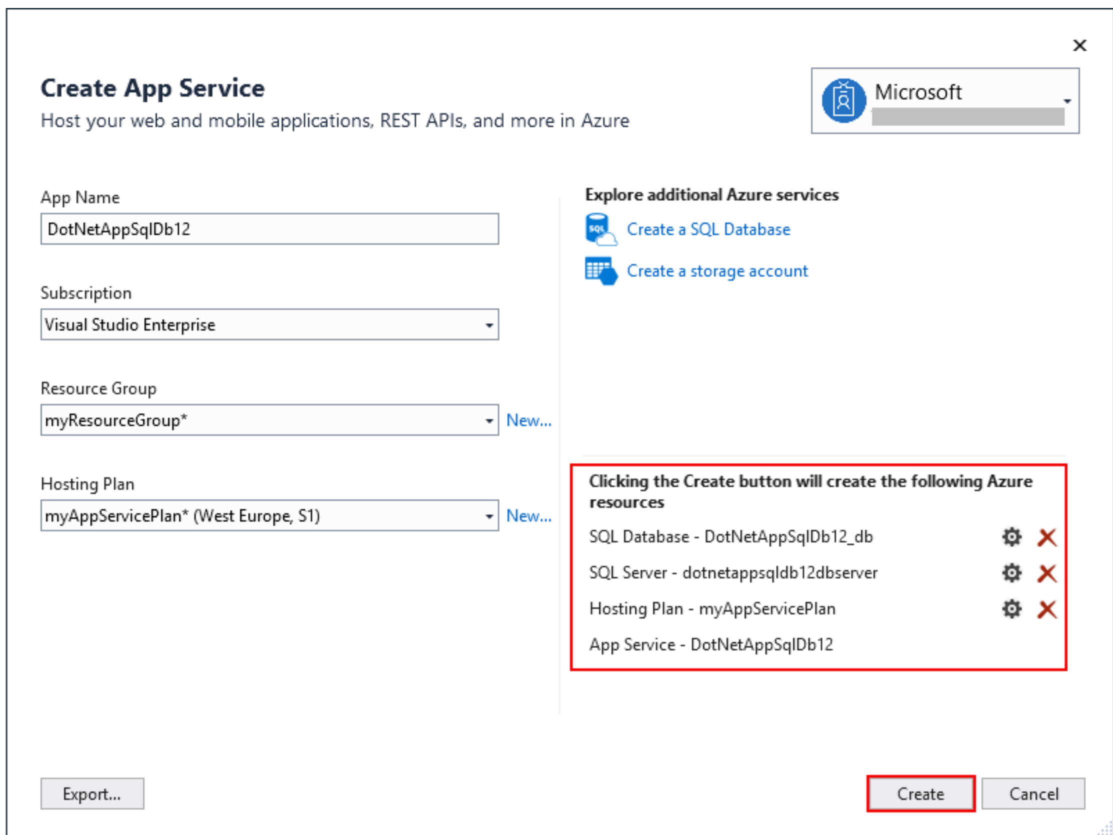


Рисунок 3.12. Проверка конфигурации окна "Создание службы приложений

Когда мастер завершит создание ресурсов Azure, он опубликует приложение ASP.NET в Azure. Откроется браузер по умолчанию с URL-адресом развернутого приложения.

Добавьте несколько элементов в список дел (рисунок 3.13).

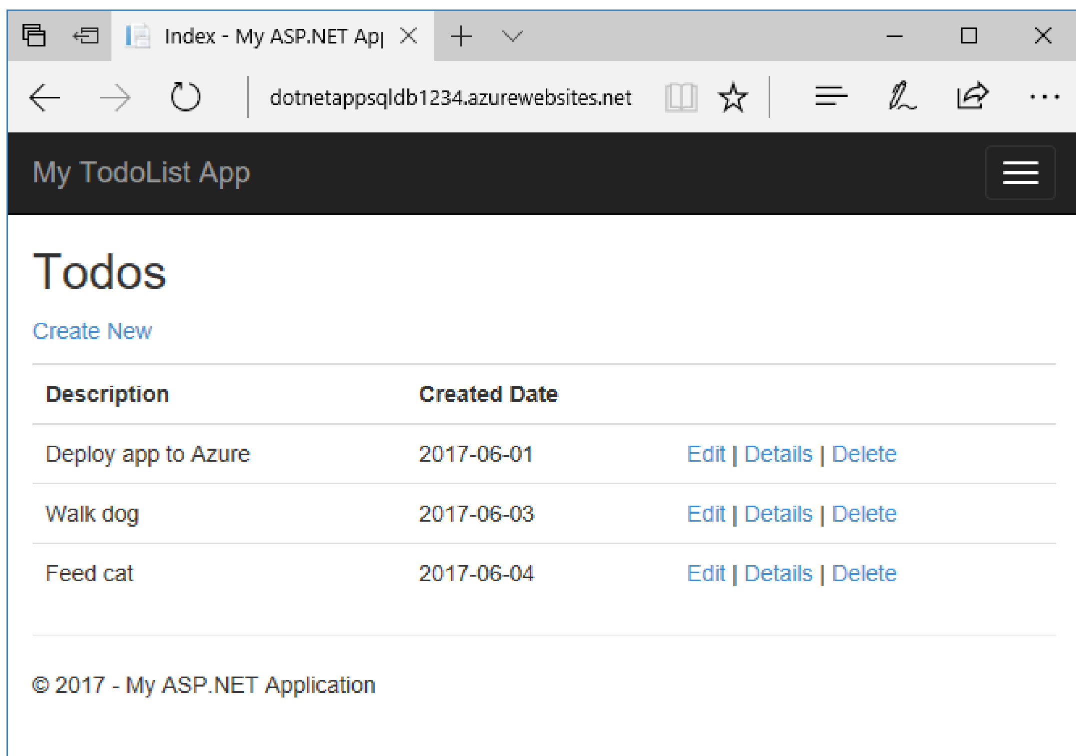


Рисунок 3.13. Добавление списка дел

Приложение ASP.NET, управляемое данными, работает на службе Azure Application Service.

Локальный доступ к базе данных SQL

В браузере объектов SQL Server Visual Studio вы можете легко просматривать и управлять базой данных SQL.

Создание подключения к базе данных

В меню **Вид** выберите **SQL Server Object Browser**.

В верхней части **браузера объектов SQL Server Object Browser** нажмите **Add SQL Server**.

Настройка подключения к базе данных

В диалоговом окне **"Подключение"** разверните узел **Azure**. Здесь показаны все экземпляры базы данных SQL в Azure. Выберите базу данных SQL, созданную ранее. Созданное вами ранее соединение автоматически появится внизу. Введите пароль администратора для ранее созданной базы данных и нажмите **Подключиться** (рисунок 3.14).

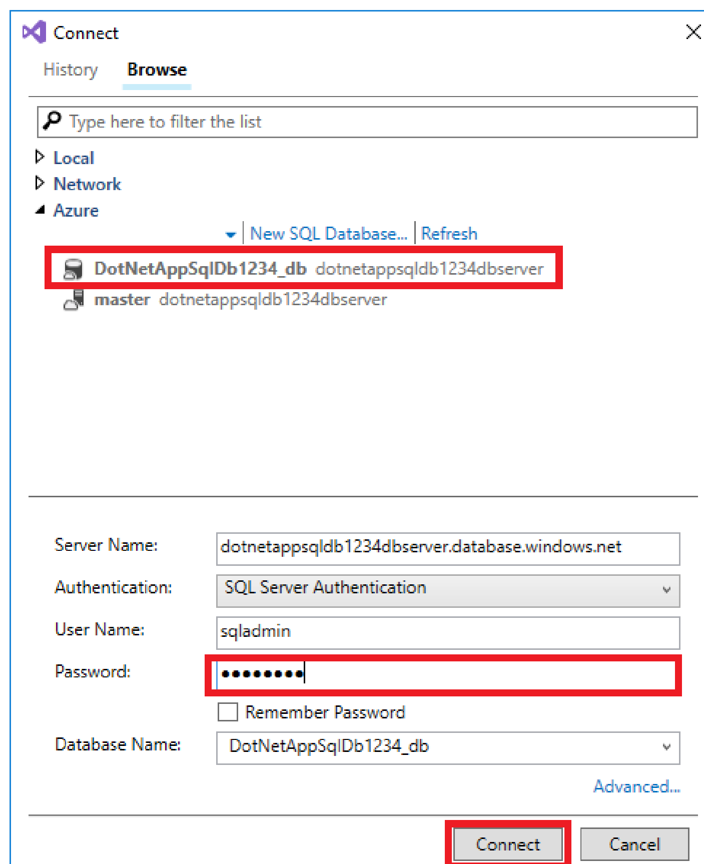


Рисунок 3.14. Настройка подключения к базе данных

Разрешить подключение клиентов с вашего компьютера

Откроется диалоговое окно **Create New Firewall Rule (Создать новое правило брандмауэра)**. По умолчанию только службы Azure, такие как ваше

веб-приложение Azure, могут подключаться к экземпляру SQL DB. Чтобы подключиться к базе данных, создайте правило брандмауэра на экземпляре SQL DB. Это правило разрешает подключения с публичного IP-адреса вашего локального компьютера.

В диалоговом окне уже содержится публичный IP-адрес компьютера.

Убедитесь, что флажок **Добавить IP-адрес моего клиента** установлен и нажмите **OK** (рисунок 3.15).

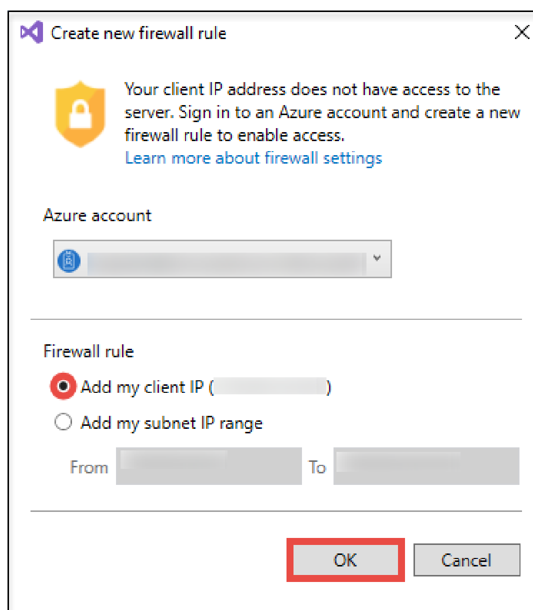


Рисунок 3.15. Разрешить клиентам подключаться

После завершения настройки брандмауэра для экземпляра базы данных SQL ваше соединение появится в **SQL Server Object Explorer**.

С его помощью можно выполнять самые распространенные операции с базой данных: запросы, создание представлений и хранимых процедур и многое другое.

Разверните узел подключения, выберите **Базы данных →<ваша база данных> Таблицы →**. Щелкните правой кнопкой мыши таблицу Todoes и выберите **View data** (рисунок 3.16).

Модификация приложений с помощью миграций Code First

Вы можете обновить базу данных и веб-приложение в Azure с помощью знакомых инструментов Visual Studio. В этом шаге вы измените схему базы данных с помощью Code First Migrations в Entity Framework и опубликуете ее в Azure.

Для получения дополнительной информации об использовании Entity Framework Code First Migrations смотрите раздел Начало работы с Entity Framework 6 Code First с MVC 5.

Обновление модели данных

Откройте файл *Models.cs* в редакторе кода. Добавьте следующие активы в класс ToDo:

bool public done { get; set; }

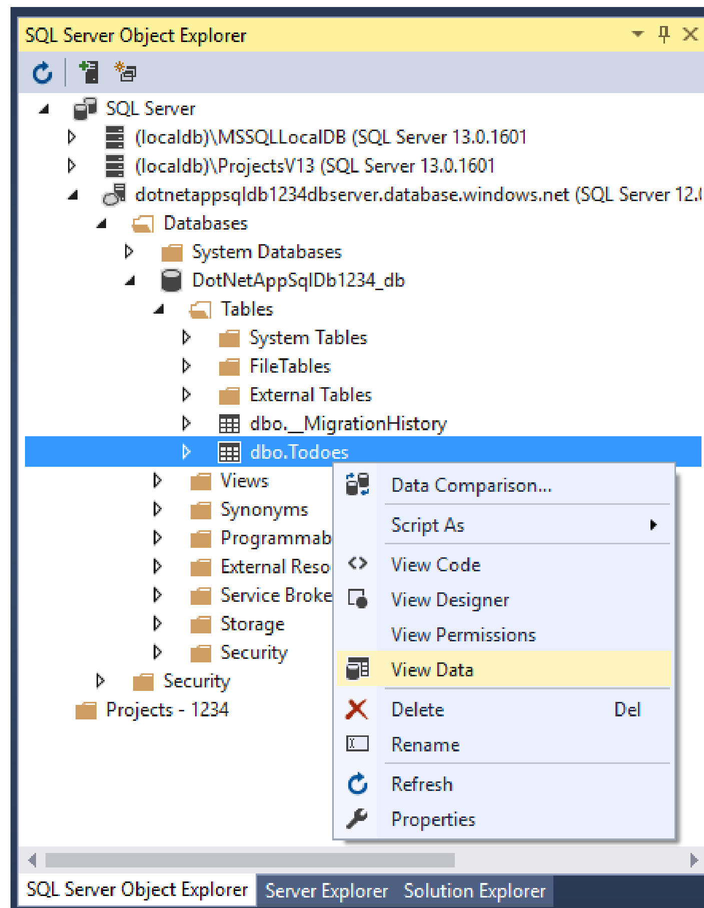


Рисунок 3.16. Браузер объектов SQL Server

Локальная реализация миграций Code First

Выполните несколько команд для обновления локальной базы данных.

В меню **Инструменты** выберите **NuGet Package Manager** > **Package Manager Console**.

В окне консоли управления пакетами активируйте миграции Code First:

Миграционное разрешение

Добавить миграцию:

Add-Migration AddProperty

Обновление локальной базы данных:

Обновление базы данных

Введите Ctrl+F5, чтобы запустить приложение. Отметьте ссылки "Изменить", "Информация" и "Создать".

Если приложение загружается без ошибок, Code First Migrations успешно активирован. Однако ваша страница не изменилась, потому что новое свойство еще не используется в логике приложения.

Использование нового свойства

Внесите некоторые изменения в код, чтобы использовалось свойство Done. Для простоты мы просто изменим представления "Оглавление" и "Создать", чтобы увидеть свойство в действии.

Откройте файл *ControllersAllController.cs*.

Найдите метод Create() в строке 52 и добавьте Done в список свойств для атрибута Bind. Когда это будет сделано, сигнатура метода Create() должна выглядеть следующим образом

```
Public ActionResult Create([Bind(Include = "Description,CreatedDate,Done")] Все)
```

Откройте файл *Views\Todos.cshtml*.

В коде Razor вы должны увидеть элемент <div class="form-group">, который использует модель. Description, и еще один элемент <div class="form-group">, который использует model.CreatedDate. Сразу после этих двух элементов добавьте еще один элемент <div class="form-group">, который использует модель. Выполнено:

```
<div class="form-group">
    @Html.LabelFor(model => model.Done, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        <div class="checkbox">
            @Html.EditorFor(model => model.Done)
            @Html.ValidationMessageFor(model => model.Done, "", new { @class = "text-danger" })
        </div>
    </div>
</div>
```

Откройте файл *Views\Todoses\Index.cshtml.*

Найдите пустой элемент <th></th>. Добавьте следующий код бритвы над этим элементом:

```
<th>
    @Html.DisplayNameFor(model => model.Done)
</th>
```

Найдите элемент <td>, содержащий вспомогательные методы Html.ActionLink(). *Над* элементом <td> добавьте еще один элемент <td> со следующим кодом Razor:

```
<td>
    @Html.DisplayFor(modelItem => item.Done)
</td>
```

Это все, что вам нужно сделать, чтобы увидеть изменения в представлениях Index и Create.

Введите Ctrl+F5, чтобы запустить приложение.

Теперь вы можете добавить элемент из списка задач и поставить галочку в поле "**Выполнено"**. После этого задание должно появиться на главной странице как выполненное. Помните, что поле "Готово" не отображается в представлении редактирования, поскольку вы не изменили представление редактирования.

Миграции Code Enablement First в Azure

Теперь, когда изменения кода, включая миграции базы данных, успешно завершены, вы можете опубликовать изменения в веб-приложении Azure и обновить базу данных SQL для использования миграций Code First.

Как и раньше, щелкните проект правой кнопкой мыши и выберите **Publish (Опубликовать)**. Нажмите **Настроить, чтобы** открыть параметры публикации (рисунок 3.17).

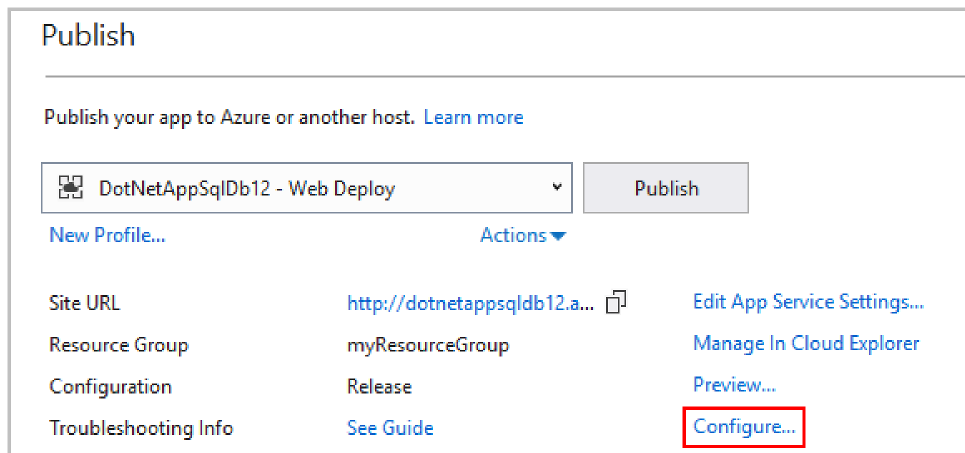


Рисунок 3.17. Настройка параметров публикации

В мастере нажмите кнопку **Далее**.

Убедитесь, что строка подключения для базы данных SQL в **MyDatabaseContext (MyDbConnection)** заполнена. вам может понадобиться выбрать **базу данных myToDoAppDb из** выпадающего списка.

Установите флажок **Запускать миграции кода первыми (при запуске приложения)** и нажмите **Сохранить** (рисунок 3.18).

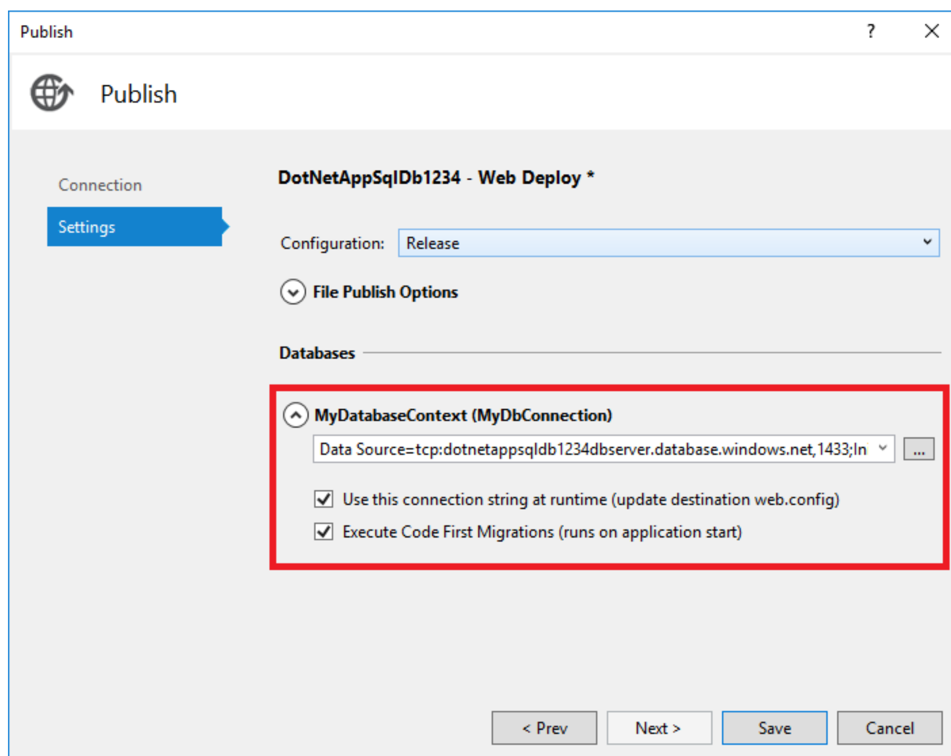


Рисунок 3.18. Настройка параметров публикации. Код первой миграции

Публикация поправок

Включив миграции Code First в веб-приложении Azure, опубликуйте изменения кода.

На странице публикации нажмите **Опубликовать**.

Попробуйте добавить новые задачи, установив флажок "**Выполнено**". Эти задания должны появиться на главной странице как выполненные (рисунок 3.19).

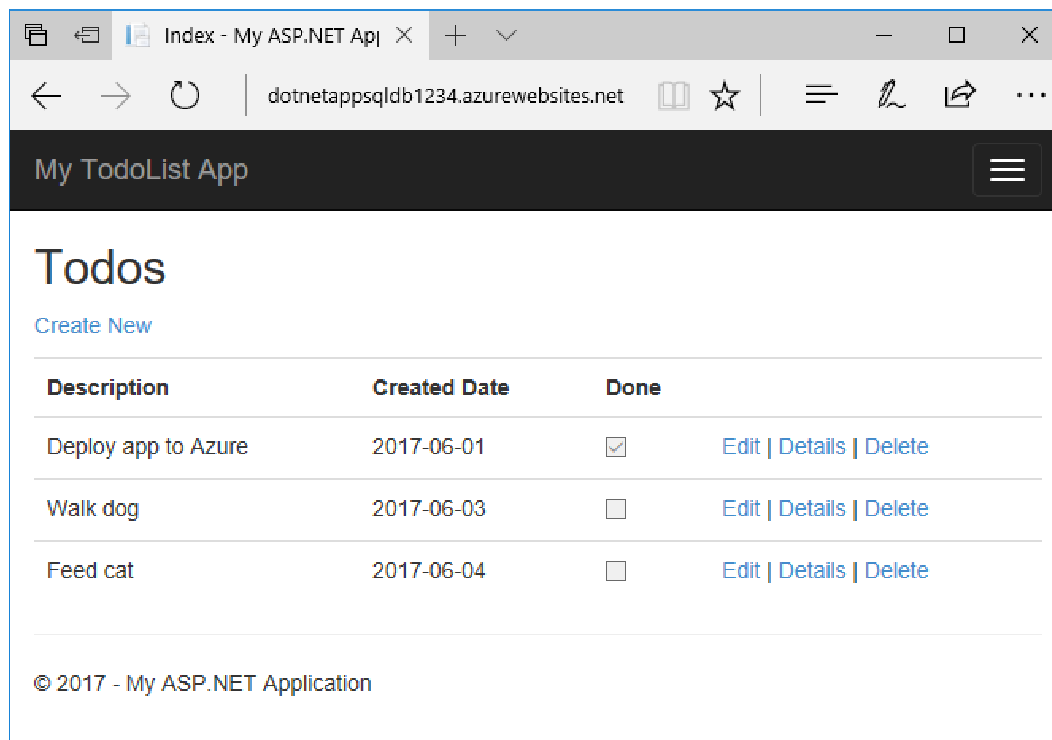


Рисунок 3.19. Опубликованные изменения в приложении

Все существующие элементы списка задач продолжают отображаться. Существующие данные в базе данных SQL не теряются при повторной публикации приложения ASP.NET. Кроме того, Code First Migrations изменяет только схему данных, оставляя существующие данные нетронутыми.

Передача записей о подаче заявления

Последующие сообщения можно передавать непосредственно из веб-приложения Azure в Visual Studio.

Откройте файл *ControllersAllController.cs*.

Каждое действие начинается с метода Trace.WriteLine(). Этот код добавлен, чтобы показать, как добавить сообщения трассировки в веб-приложение Azure.

Откройте браузер сервера

В меню **View (Вид)** выберите **Server** Browser (Браузер **сервера)**. Регистрация для веб-приложения Azure может быть включена в браузере **сервера.**

Активация передачи регистрации

В **браузере сервера** выберите **Azure** > **Служба приложений**.

Реализует **myResourceGroup,** которая была создана при создании веб-приложения.

Щелкните правой кнопкой мыши на веб-приложении и выберите **View Transmission Logs** (рисунок 3.20).

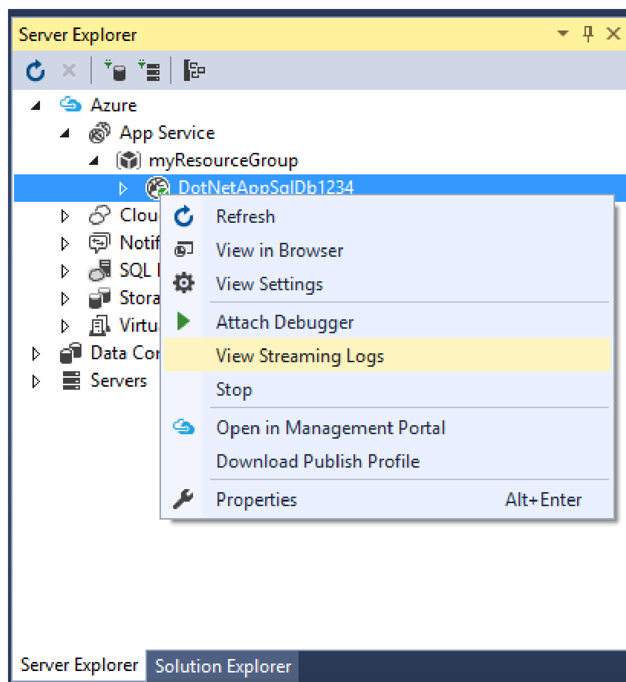


Рисунок 3.20. Меню просмотра журнала передач

Теперь записи переносятся в окно **вывода** (рисунок 3.21).

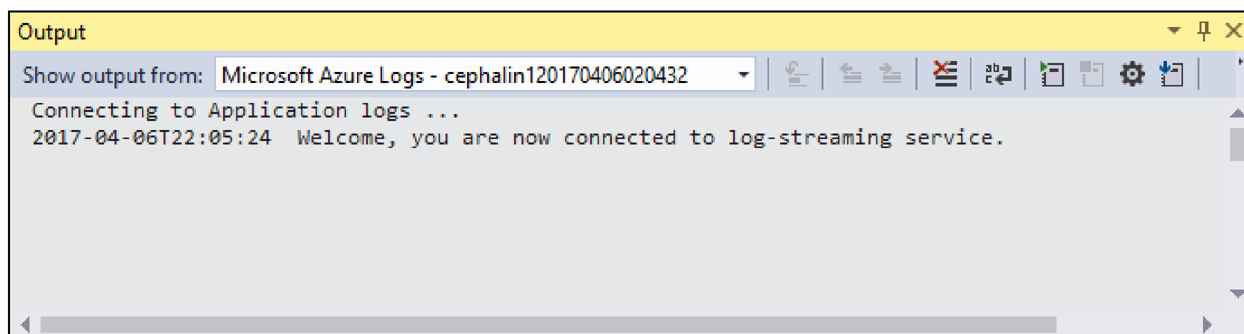


Рисунок 3.21. Окно вывода

Однако сообщения трассировки по-прежнему не отображаются. Это происходит потому, что при первом выборе пункта меню **View Streaming Logs** веб-приложение устанавливает уровень трассировки на Error, где регистрируются только события ошибок (с помощью метода Trace.TraceError()).

Изменение уровней трассировки

Чтобы изменить уровень скрининга для выдачи других скрининговых сообщений, вернитесь в **браузер сервера**.

Снова щелкните правой кнопкой мыши на веб-приложении и выберите **View Settings**.

В раскрывающемся списке **Журнал приложений (Файловая система)** выберите **Сведения**. Выберите команду **Сохранить** (рисунок 3.22).

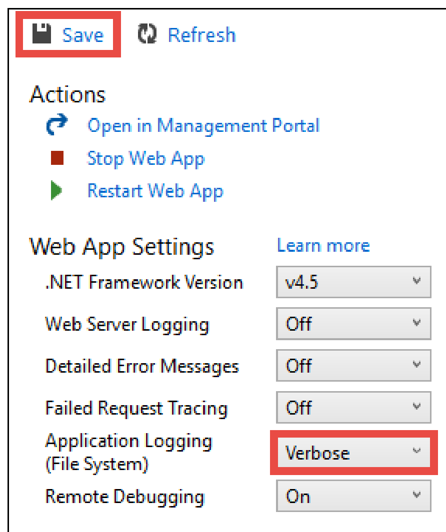


Рисунок 3.22. Изменение уровней следов

Совет

Вы можете поэкспериментировать с различными уровнями трассировки, чтобы увидеть, какие типы сообщений отображаются для каждого уровня. Например, информационный уровень включает все записи, созданные Trace.TraceInformation(), Trace.TraceWarning() и Trace.TraceError(), но не включает записи, созданные Trace.WriteLine().

В браузере снова перейдите к веб-приложению по адресу *http://<имя приложения>.azurewebsites.net*, а затем щелкните правой кнопкой мыши приложение в списке задач Azure. Сообщения трассировки теперь передаются в окно **вывода** в Visual Studio.

Приложение: 2017-04-06-T23:30:41 PID[8132] Verbose GET /Todos/Index
Приложение: 2017-04-06-T23:30:43 PID[8132] Verbose GET /All/Create
Application: 2017-04-06-T23:30:53 PID[8132] Verbose POST /All/Create
Приложение: 2017-04-06-T23:30:54 PID[8132] Verbose GET /Todos/Index

Деактивация передачи регистрации

Чтобы остановить службу передачи журнала, нажмите кнопку **Stop Watch** в окне **вывода** (рисунок 3.23).

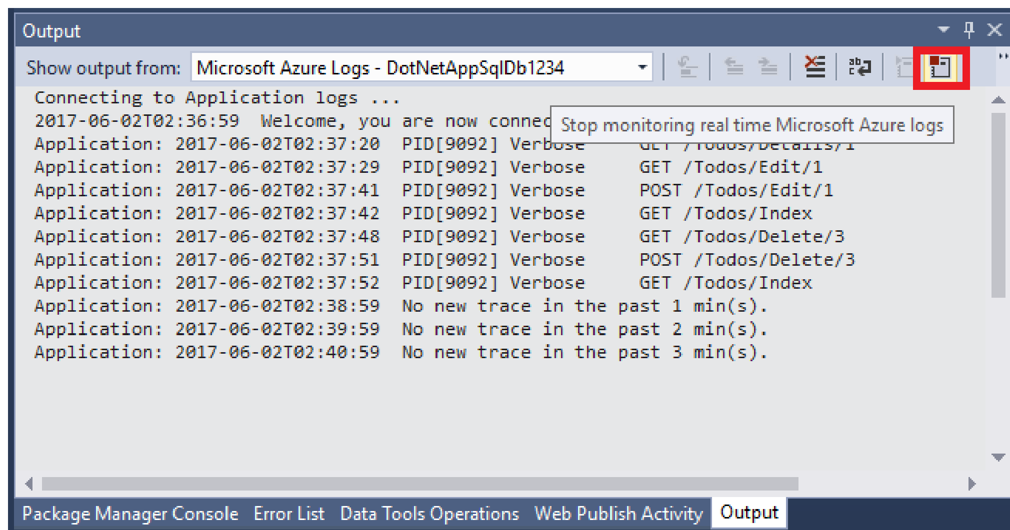


Рисунок 3.23. Окно вывода

Управление веб-приложениями Azure

Просмотрите портал Azure, чтобы увидеть созданное вами веб-приложение.

В левом меню выберите **Application Service,** а затем нажмите на имя вашего веб-приложения Azure (рисунок 3.24).

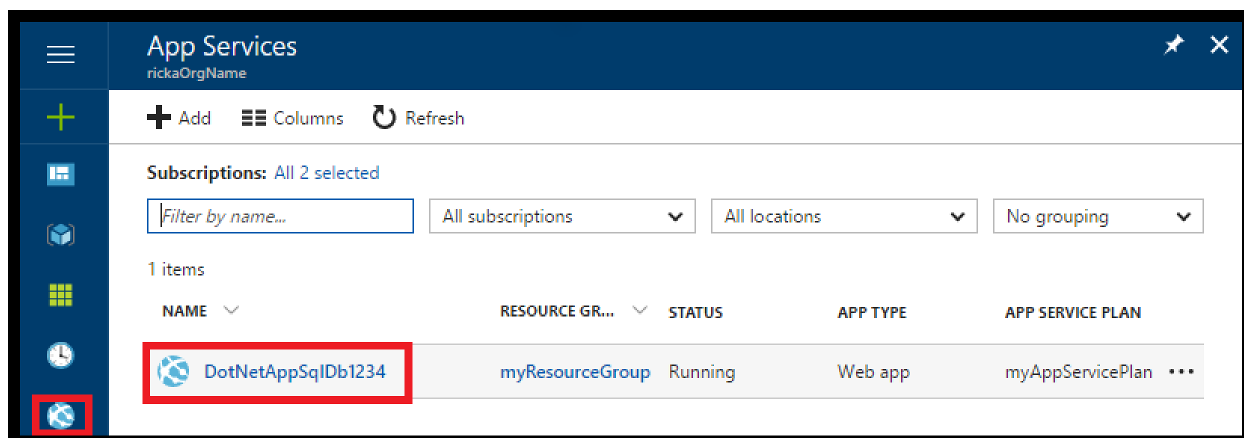


Рисунок 3.24. Категория прикладных услуг

Вы попадете на страницу веб-приложения.

По умолчанию на портале отображается страница "**Обзор**". Здесь вы можете управлять работой приложения. Вы также можете выполнять основные задачи управления: обзор, завершение, запуск, перезапуск и удаление. Вкладки в левой части страницы показывают различные страницы конфигурации, которые вы можете открыть (рисунок 3.25).

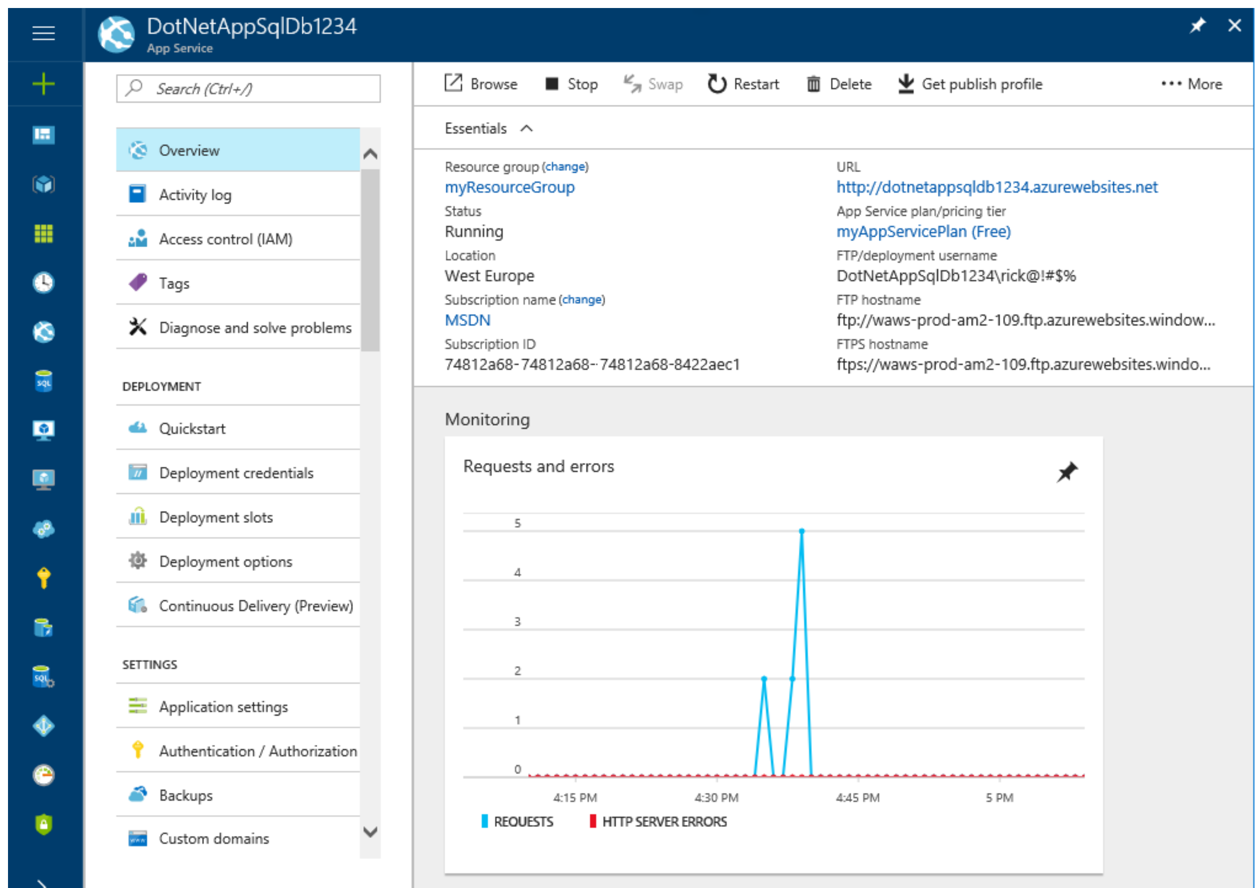


Рисунок 3.25. Обзор службы приложений

Расчистка ресурсов

В предыдущем шаге вы создали ресурсы Azure в группе ресурсов. Если вы считаете, что эти ресурсы не понадобятся вам в будущем, вы можете удалить их, удалив группу ресурсов.

1. На странице **Обзор** веб-приложения Azure Portal нажмите ссылку **myResourceGroup** в разделе **Группы ресурсов**.
2. На странице группы ресурсов убедитесь, что удаляемые ресурсы перечислены.
3. Нажмите **Удалить**, введите **myResourceGroup в** текстовое **поле** и снова нажмите **Удалить**.

Мобильные технологии

Функция как способ группировать команды и именовать участки кода

Работа любой компьютерной программы — это выполнение процессором большого набора элементарных инструкций. В машинном коде, с которым работает процессор, все команды очень простые:

– считать из оперативной памяти одно целое число в специальную ячейку;
– прибавить к одному числу значение другой ячейки;
– сравнить ячейку с нулем;
– вернуться на пару команд назад и пр.

Команды машинного кода не могут вывести окошко программы или проиграть аудиофайл, не могут посчитать среднюю оценку в классе или загрузить страничку из Интернета. Машинный код не умеет полноценно работать даже с обычными строками или списками и не может выполнять сложные математические расчеты. Однако программа целиком все это делает, потому что состоит из множества команд, которые в комбинации дают нужный эффект.

Многие из команд Kotlin, которые вы уже знаете, требуют от процессора выполнения десятков команд. Если бы программист писал их вручную, даже простейшие программы создавались бы несколько дней. При этом даже опытному программисту было бы очень легко допустить ошибку.

Рассмотрим программу, в которой последовательно запрашивается имя, а затем выводится приветствие по имени для трех живых существ:

```
import java.util.*

fun main() {
    val sc = Scanner(System.`in`)
    print("Как тебя зовут?")
    val name_1 = sc.next()
    print("Привет $name_1")

    print("А тебя?")
    val name_2 = sc.next()
    print("Привет $name_2")

    print("А твоего пса?")
    val name_3 = sc.next()
    print("Привет $name_3")
}
```

```
Как тебя зовут?
Вася
Привет Вася
А тебя?
Коля
Привет Коля
А твоего пса?
Шарик
Привет Шарик
```

Программа небольшая, но уже видно много проблем:

Во-первых, три раза приходится повторять фактически одно и то же

Во-вторых, приходится вводить разные имена переменных, чтобы ничего не перепутать и не поприветствовать кого-нибудь неправильным именем

В-третьих, если вы захотите исправить приветствие на более официальное — например, «Здравствуйте», вам придется внести одинаковые исправления сразу в трех разных местах

Даже в такой маленькой программе можно при исправлении допустить опечатку. А представьте, что фраза используется десять раз в разных местах большой программы — тогда придется искать каждое приветствие и исправлять его.

Было бы здорово иметь возможность устранить дублирование кода: в каком-то одном месте сообщить компилятору, что именно мы понимаем под словом «поприветствовать», а затем попросить компилятор использовать определение термина «поприветствовать» там, где мы его попросим об этом.

Итак, сформулируем, чего мы хотим добиться:

Один раз определить, что значит «поприветствовать», т. е. сгруппировать и поименовать повторяющийся кусок кода

Многократно в дальнейшем «ссылаться» на это определение везде, где нам только потребуется

Замечательно, что язык Kotlin действительно обладает такими выразительными возможностями — **функциями**.

Функция

Функция — набор команд, сгруппированный особым образом. Команды в этом наборе выполняются последовательно, но воспринимаются как единое целое. При этом функция может возвращать (или не возвращать) свой результат.

Для того чтобы использовать какую-нибудь собственную функцию, вначале необходимо ее **объявить**, т. е. рассказать, что именно она будет делать. В нашем примере мы объявим функцию *greet* с помощью ключевого слова *fun*.

```
fun greet(sc: Scanner) {
    val name = sc.next()
    print("Привет $name")
}
```

Далее мы можем использовать нашу функцию *greet* всякий раз, когда в нашем коде возникает необходимость кого-нибудь поприветствовать:

```
fun main() {
    val sc = Scanner(System.`in`)
    print("Как тебя зовут?")
    greet(sc)

    print("А тебя?")
    greet(sc)

    print("А твоего пса?")
    greet(sc)
}
```

Вызов функции

Обращение к ранее объявленной функции с целью выполнения ее команд называется вызовом.

В нашем примере функция *greet* один раз объявляется, а затем три раза вызывается.

Что мы получили в результате:

Код сократился и стал понятнее. Теперь нам не нужно выискивать, где какая переменная заводится, где и для чего она используется. Функция сама говорит, что она делает: *greet* — «поприветствовать».

Нам не приходится заводить несколько разных переменных.

Чтобы поменять приветствие во всей программе, достаточно изменить одну строчку.

Итак, функции нужны, чтобы группировать команды, а заодно — чтобы не писать один и тот же код несколько раз.

Например, достаточно один раз написать функцию *greet* и потом пользоваться ею постоянно. Польза от этого особенно очевидна, когда функция действительно сложная и используется много раз в разных местах программы. Например, загрузку данных из Интернета или отрисовку персонажа компьютерной игры удобно оформлять в виде отдельных функций.

Еще одна важная вещь состоит в том, что функции имеют **имена**.

Благодаря им программу можно сделать понятной не только компьютеру, но и человеку. Тут все так же, как с именами переменных: если переменная имеет ничего не говорящее название, сложно угадать, что в ней хранится. Если участок кода не сгруппирован в функцию, иногда приходится буквально дешифрировать, для чего он нужен в программе. А если он оформлен в виде функции, название функции само подскажет, что делает этот код.

Проиллюстрируем сказанное на примере. Попробуйте угадать, что делает такой код:

```
fun main() {
    val t = listOf(-5, -10, 1, 11, 20, 25, 27, 23, 18, 8, 2, -3)
    var s = 0.0
    var mm = 1000
    var mx = -1000
    for (e in t) {
        s += e
        if (e < mm)
            mm = e
        if (e > mx)
            mx = e
    }
    println(s / t.size.toDouble())
    println(mm)
    println(mx)
}
```

После некоторых нетривиальных усилий по дешифровке можно понять, что этот код вычисляет среднее, минимальное и максимальное значение списка:

```
9.75
-10
27
```

А вот тот же самый код, но переработанный с помощью встроенных функций и хороших названий переменных:

```
fun main() {
    val temperatures = listOf(-5, -10, 1, 11, 20, 25, 27, 23, 18, 8, 2, -3)
    val average_temperature = temperatures.sum().toDouble() /
            temperatures.size.toDouble()

    println(average_temperature)
    println(temperatures.min())
    println(temperatures.max())
}
```

Определение простейших функций

Давайте подытожим то, что мы знаем о функциях.

Заголовок и тело функции

У каждой функции есть заголовок (его обычно называют сигнатурой) и тело. Сигнатура описывает, как функцию вызывать, а тело описывает, что эта функция делает.

Сигнатура содержит имя функции, аргументы (то есть параметры), которые передаются в функцию и тип возвращаемого значения.

Записывается это так:

fun <имя функции>([аргументы]): <тип возвращаемого значения>{
<тело функции>
}

Начальные знания о локальных переменных

В тот момент, когда вы вызываете функцию *greet*, начинают выполняться команды, написанные в теле функции. Когда работа функции доходит до конца, исполнение программы продолжается со строки, которая вызывала функцию.

Обратите внимание: теперь в программе используется только одна переменная — *name*. Как же так? Ведь мы договорились, что не будем использовать одну и ту же переменную для разных имен? На самом деле мы не используем одну и ту же переменную. При каждом вызове функции эта переменная создается заново, а в конце работы функции — прекращает свое существование. Это очень важный момент.

Область видимости переменной

Снаружи функции *greet* переменная *name* вообще не существует. Таким образом, функция очерчивает тот участок программы, где переменная нужна и используется. Этот участок, в котором переменная **живет**, называется областью видимости переменной (по-английски — *scope*).

Благодаря ограничению области видимости переменной программисту не нужно беспокоиться, не «всплывет» ли эта переменная в другом месте программы. Изменяя переменную внутри функции, программист понимает, что он может что-то испортить **только внутри** функции, но не поломает работу остальной программы. Можно сказать, что вся работа с переменной локализована, т. е. сосредоточена внутри функции.

Локальные и глобальные переменные

Переменные, создаваемые внутри функций, недоступны извне и существуют только внутри функции. Они называются локальными.

Создаваемые вне функции переменные могут быть доступны из функций. Они являются глобальными.

По возможности избегайте использования глобальных переменных для предотвращения конфликтов.

Аргументы функций

Большая часть программ требует выполнения разных действий. Например, функция *print* (а это именно функция) должна каждый раз выводить на экран разные сообщения — в зависимости от переданных аргументов.

Аргументы функций

Аргументы (параметры) могут изменять поведение функции. В зависимости от конкретного аргумента она возвращает разный результат, а значит, выполняет внутри немного различного действия.

Рассмотрим функцию, которая должна выводить на экран содержимое списка, печатая каждый элемент на своей строчке. Вряд ли нам захочется заводить функцию, которая раз за разом выводит содержимое одного и того же списка. Скорее, нужна функция, которая может распечатать список. Конкретный список мы передаем функции в качестве параметра при ее вызове. Функция же работает с тем, что ей передали.

Выглядит это так:

```
fun print_array(array: List<String>) {
    for (element in array)
        println(element)
}

fun main() {
    print_array(listOf("Михаил", "Ольга", "Светлана"))
}
```

Важно!

В момент вызова функции ей необходимо передать вычисленные аргументы. Если аргументы не вычислены, они вычисляются слева направо.

Связь между математическими функциями и функциями в Kotlin

Функции, которые мы писали до сих пор, выводили значение на экран. Однако значение, выведенное на экран, полезно только для человека. Сама программа никак не может его использовать. Если бы функции могли выводить результаты своей работы только на экран, их было бы почти невозможно комбинировать.

Каждая функция может не только выполнять действия, но и выдавать какой-то результат, который потом можно использовать в программе — например, записать в переменную. Вы еще не делали таких функций, но уже не раз пользовались ими. Попробуйте вспомнить несколько примеров.

Если вы посмотрите примеры таких функций, вы увидите среди них много математических... функций.

Функция в математике — такое преобразование, которое из одного значения или набора значений делает другое значение. Например, функция квадратного корня делает из числа его корень. Функция $f(a,b) = (a+b)^2$ делает из двух чисел квадрат их суммы. Фактически единственное важное свойство математической функции заключается в том, что каждому набору аргументов она сопоставляет значение, и каждый раз вычисление функции на одних и тех же аргументах дает один и тот же результат. Сколько бы раз вы ни вычисляли корень из шестнадцати, каждый раз будете получать четыре.

Заметьте: математическая функция не обязана работать с числами. Например, в математике можно встретить такую функцию — число перестановок букв в слове. Это функция, которая принимает аргументом строку, а возвращает число. Или функцию пересечения множеств, которая берет в качестве аргументов два множества и возвращает тоже множество.

Чем функции в программировании похожи и чем отличаются от функций в математике?

Функция в языке Kotlin — некий алгоритм, который выполняется каждый раз одинаково. Для большинства функций возвращаемое значение, как и в случае математической функции, зависит только от аргументов.

У функций в Kotlin, как вы знаете, тоже есть список аргументов: иногда одно значение, иногда несколько, а иногда он и вовсе пустой. У функций также есть возвращаемое значение — значение всегда ровно одно.

Возвращаемые значения

Для того чтобы функция вернула значение, используется оператор *return*. Использовать его очень просто. Давайте напишем функцию *double_it*, которая удваивает значение:

```
fun double_it(x: Float): Float {
    return x * 2F
}
```

Эта функция получила число с плавающей точкой *x* в качестве аргумента, умножила его на 2 и вернула результат в основную программу. Значением, которое функция вернула, можно воспользоваться. При этом тип возвращаемого значения указывается после двоеточия в заголовке функции.

Например, мы можем посчитать длину окружности с использованием этой функции:

```
fun main() {
    val radius = 3F
    val length = double_it(3.14F) * radius
    print(length)
}
```

Когда компилятор дойдет до *double_it(3.14)*, начнет исполняться код функции. Когда он дойдет до слова *return*, значение, которое указано после *return*, будет подставлено в программе вместо вызова функции. Можно сказать, что сразу после того, как функция досчитается, вычисление превратится в такое:

```
val length = 6,28 * radius
```

Важно!

Заметьте: функция *double_it* ничего не выводит на экран. Она выполняет вычисления и сообщает их не пользователю, как мы делали раньше, а другой части программы.

Если нам потребуется не просто вернуть удвоенное число, а еще и вывести его на экран, лучше не добавлять *print* внутрь функции. Ведь если вы добавите *print* в функцию, уже никак не сможете вызвать эту функцию в «тихом» режиме, чтобы она ничего не печатала. Вместо этого лучше сначала вернуть результат, а потом уже распечатать его во внешней программе. Вот так:

```
print(double_it(3.14F))
```

Или, если вам нужно еще как-то использовать вычисленное значение, можно завести специальную переменную, хранящую результат вычисления.

```
val double_pi = double_it(3.14F)
print(double_pi)
val length = double_pi * radius
```

Функция удваивания числа, конечно, совершенно бесполезна.

А теперь давайте рассмотрим чуть более сложный пример — вычисление суммы элементов списка:

```
fun sumArr(arr: List<Int>):Int {
    var result = 0
    for (element in arr)
        result += element
    return result
}

fun main() {
    print(sumArr(listOf(1, 2, 3, 4)))
}
```

Здесь мы используем очень распространенный способ написания функций: создаем вспомогательную переменную, а затем возвращаем ее значение.

Но мы ведь недавно говорили, что локальные переменные живут только внутри функции. Если переменная *result* исчезнет, почему результат — число 10 — никуда не пропадает?

Объект (значение) может существовать, даже когда нет переменной, в которой он хранится. Когда мы записываем число в *result*, мы фактически создаем объект числа и даем ему временное имя *result*. Потом, когда мы пишем *return result*, мы возвращаем не переменную. Как и в большинстве конструкций языка (кроме, пожалуй, присваивания), вместо переменной подставляется ее значение: таким образом, мы возвращаем объект «число 10». У этого объекта нет имени, что не мешает функции *print* использовать его и напечатать 10 на экране.

Однако, если ни программист, ни программа не имеют возможности пользоваться объектом, этот объект становится не нужен. После того как функция *print* отработала, доступ к результату вычисления пропал, ведь мы никуда не сохранили этот результат. На объект «число 10» нет ссылок, поэтому компилятор может его «выкинуть».

Сборка мусора

Это называется «сборка мусора»: Kotlin автоматически избавляется от всех объектов, которые невозможно использовать.

Множественные точки возврата из функции

Часто бывают ситуации, когда в зависимости от входных данных нужно выполнить различные наборы команд. Например, когда мы считаем модуль, в случае отрицательного числа нужно взять число со знаком минус, а в случае неотрицательного числа (положительное или ноль) мы берем само число.

Давайте запишем это в виде функции my_abs(x: Int).

```
fun my_abs(x: Int): Int {
    val result: Int
    if (x >= 0)
        result = x
    else
        result = -x
    return result
}
```

Но если вдуматься: зачем ждать конца функции, когда мы уже вычислили результат и совершили все необходимые действия? Давайте завершим функцию сразу, ведь *return* именно для этого создан: он не только возвращает значение функции, но и возвращает нас из функции в основную программу. После вызова оператора *return* выполнение кода функции заканчивается. Раз так, давайте немного упростим программу. Сначала мы перенесем *return* к тому месту, где результат получен:

```
fun my_abs(x: Int): Int {
    val result: Int
    if (x >= 0) {
        result = x
        return result }
    else {
        result = -x
        return result }
}
```

А теперь можно заметить, что переменная *result* лишняя: когда вы подставляете значение *result*, вместо *result* вы легко можете подставить сразу результат.

```
fun my_abs(x: Int): Int {
    if (x >= 0)
        return x
    else
        return -x
}
```

Можно обратить внимание, что в последней записи *else*-часть становится нам вообще не нужна, и тогда сократить функцию до вот такой:

```
fun my_abs(x: Int): Int {
    if (x >= 0)
        return x
    return -x
}
```

Заметьте, что любую функцию можно написать с одним-единственным оператором *return*, но часто использовать несколько точек выхода из функции просто удобно.

Возврат из глубины функции

Множественные точки возврата из функции позволяют нам упростить обработку и более сложных структур, например, вложенных списков. Наша следующая программа будет проверять, есть ли в матрице элемент, отличающийся от искомого не больше чем на число *eps*.

Матрица записывается как список списков. Мы предполагаем, что наша функция будет работать с большими матрицами, поэтому нам не хочется тратить лишнее время на проверку. Мы будем прекращать поиск, как только нашли подходящий элемент. Давайте для начала разберемся, как бы мы действовали без множественных операторов *return*.

```
fun matrix_has_close_value(matrix: List<List<Int>>, value: Int, eps: Double): Boolean {
    var found = false
    for (row in matrix) {
        for (cell in row) {
            if (abs(cell - value) <= eps) {
                found = true
                break }
        }
        if (found)
            break
    }
    if (found)
        return true
    else
        return false
}
```

Как видите, нам приходится прилагать некоторые усилия, чтобы выйти из нескольких уровней вложенности. Каждый уровень вложенности — дополнительное препятствие на пути к завершению функции. Ему мешают не только циклы, как в этом примере, но и условные операторы.

Перепишем теперь функцию с учетом того, что, как только мы нашли элемент, мы уже знаем, что ответ — True (т. е. элемент содержится в матрице). А если мы закончили перебор элементов и так и не нашли ни одного элемента, ответ False.

```
fun matrix_has_close_value(matrix: List<List<Int>>, value: Int, eps: Double): Boolean {
    for (row in matrix)
        for (cell in row)
            if (abs(cell - value) <= eps)
                return true
    return false
}
```

Хотя return false не заключен ни в какое условие, выполняется он только тогда, когда элемент не найден. Если элемент найден, мы сразу выходим из функции и до этой строки просто не доходим. Оператор *return* очень удобен, когда нужно выйти из глубины функции.

Что можно возвращать из функции

В функциях, которые не возвращают значение, тоже можно использовать *return*. Если написать *return* без аргументов, функция просто сразу завершит свою работу (без *return* функция завершает работу, когда выполнит последнюю команду).

Результат возвращает любая функция, даже если в ней нет слова *return*. Результатом такой функции будет Unit.

Если в функции использован *return* без аргументов, это фактически эквивалентно return Unit.

Задание 2 – Дан список *List* из 10 целых чисел. Написать функцию, которая принимает на вход список и возвращает сумму максимального и минимального элементов.

Порядок выполнения:

a. Создание нового Kotlin файла

Добавьте в проект новый файл, щелкнув правой кнопкой мыши на папке **src** в окне инструментов проекта. Выберите New – Kotlin File/Class. Ввести имя файла, например, Task03_1. Нажать кнопку «Enter».

b. Написание кода основной программы

В появившемся окне необходимо ввести программный код решения задачи, приведенный ниже.

Листинг 3.1. Программный код решения задачи 2

```
import java.util.*

private fun maxMinElement(arr: List<Int>): Int {
    var max = arr[0]
    var min = arr[0]
    // Цикл перебора элементов списка myList
    for (element in arr) {
        if (element > max) {
            max = element
        }
        if (element < min) {
            min = element
        }
    }
    return max + min
}

fun main() {
    // Объект считывания данных с консоли
    val sc = Scanner(System.`in`)
    val myList = mutableListOf<Int>() // Создание списка
    println("Введите последовательность из 10 цифр ")
    // цикл заполнения списка myList
    for (i in 0 until 10){
        myList.add(sc.nextInt())
    }
    println("Сумма Макс и Мин элементов списка = ${maxMinElement(myList)}")
}
```

c. Построение проекта

После ввода программного кода нужно скомпилировать и отладить программу. Для этого необходимо выполнить Build – Build Project. Если в программном коде имеются ошибки, они будут выделены красным цветом.

d. Запуск программы

Чтобы просмотреть результат выполнения программы, нужно выполнить Run – Run ‘Task03_1Kt’ или нажать кнопку в виде треугольника (рисунок 3.26).

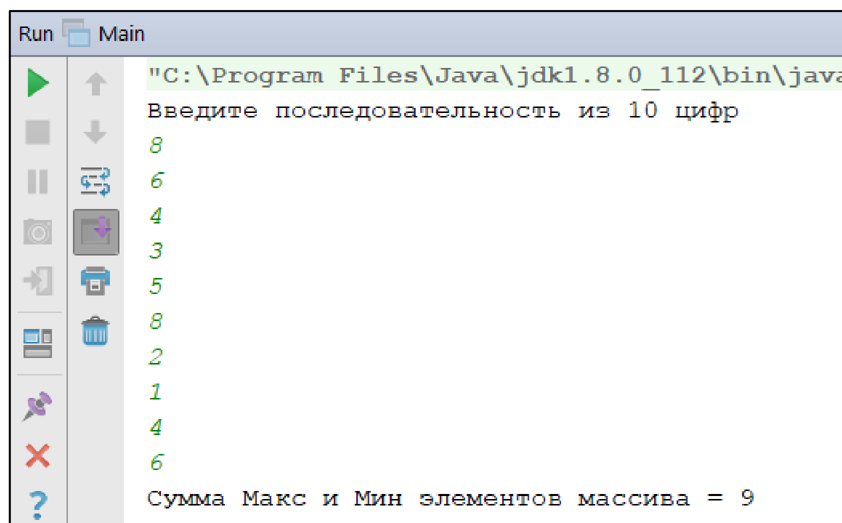


Рисунок 3.26. Результат решения задачи 2

Задание 3 – Написать перегруженную функцию, которая суммирует два, или три целых числа, или все элементы списка целых чисел.

Порядок выполнения:

a. Создание нового Kotlin файла:

Добавьте в проект новый файл, щелкнув правой кнопкой мыши на папке **src** в окне инструментов проекта. Выберите New – Kotlin File/Class. Ввести имя файла, например, Task03_2. Нажать кнопку «Enter».

b. Написание кода основной программы

В появившемся окне необходимо ввести программный код решения задачи, приведенный ниже.

Листинг 3.2. Программный код решения задачи 4

```
import java.util.*
// Функция суммирование 2-х целых чисел
fun Sum(a: Int, b: Int): Int {
    return a + b
}
// Функция суммирование 3-х целых чисел
fun Sum(a: Int, b: Int, c: Int): Int {
    return a + b + c
}
// Функция суммирование последовательности целых чисел
fun Sum(arr: List<Int>): Int {
    var s = 0
    for (element in arr) {
        s += element
    }
    return s
}

fun main() {
    val sc = Scanner(System.`in`)
    println("Введите значение: ")
    var numberOne = sc.nextInt() // Ввод значения в переменную
    println("Введите значение: ")
    var numberTwo = sc.nextInt() // Ввод значения в переменную
    println("Сумма чисел = ${Sum(numberOne, numberTwo)}")
    //////////////////////////////////////////////////////

    println("Введите значение: ")
    numberOne = sc.nextInt() // Ввод значения в переменную
    println("Введите значение: ")
    numberTwo = sc.nextInt() // Ввод значения в переменную
    println("Введите значение: ")
    var numberThree = sc.nextInt() // Ввод значения в переменную
    println("Сумма чисел = ${Sum(numberOne, numberTwo, numberThree)}")
    //////////////////////////////////////////////////////

    val myList = listOf(1, 2, 3, 4, 5)
    println("Сумма списка значений = ${Sum(myList)}")
}
```

c. Построение проекта:

После ввода программного кода нужно скомпилировать и отладить программу. Для этого необходимо выполнить Build – Build Project. Если в программном коде имеются ошибки, они будут выделены красным цветом.

d. Запуск программы

Чтобы просмотреть результат выполнения программы, нужно выполнить Run – Run 'Task03_2Kt' или нажать кнопку в виде треугольника (рисунок 3.27).

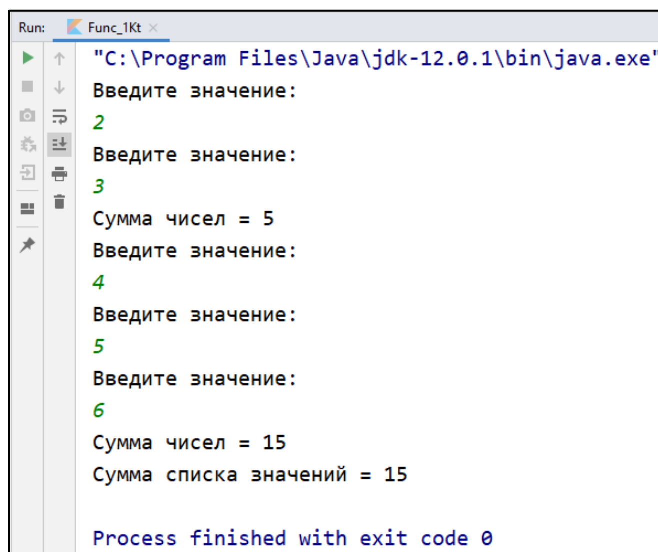


Рисунок 3.27. Результат решения задачи 3

Задание 4 – Дан список *List* из N целых чисел. Написать функцию, которая принимает на вход список и модифицирует его таким образом, чтобы элементы равные 5 были заменены на 0.

Порядок выполнения:

a. Создание нового Kotlin файла:

Добавьте в проект новый файл, щелкнув правой кнопкой мыши на папке **src** в окне инструментов проекта. Выберите New – Kotlin File/Class. Ввести имя файла, например, Task04_3. Нажать кнопку «Enter».

b. Написание кода основной программы

В появившемся окне необходимо ввести программный код решения задачи, приведенный ниже.

Листинг 3.3. Программный код решения задачи 3

```
import java.util.*

private fun changeList(myList: MutableList<Int>) {
    for (i in 0 until myList.size)
        if (myList[i] == 5)
            myList[i] = 0
}

fun main() {
    // Объект считывания данных с консоли
    val sc = Scanner(System.`in`)
    val myList = mutableListOf<Int>() // Создание списка
    print("Введите размерность списка: ");
    val n = sc.nextInt();
    println("Введите последовательность из $n-ти цифр:");
    // цикл заполнения списка myList
    for (i in 0 until n){
        myList.add(sc.nextInt())
    }
    changeList(myList)
    // цикл вывода элементов списка myList
    for (element in myList)
        print("$element ")
}
```

c. Построение проекта:

После ввода программного кода нужно скомпилировать и отладить программу. Для этого необходимо выполнить Build – Build Project. Если в программном коде имеются ошибки, они будут выделены красным цветом.

d. Запуск программы

Чтобы просмотреть результат выполнения программы, нужно выполнить Run – Run 'Task03_3Kt' или нажать кнопку в виде треугольника (рисунок 4.3).

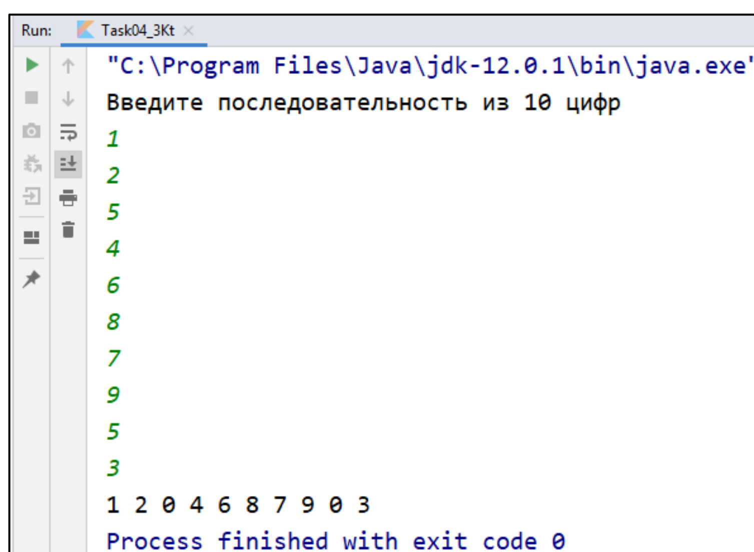


Рисунок 4.3. Результат решения задачи 4

Библиографический список

e. Начало работы с Облачными службами Azure (классическая версия) и ASP.NET. URL: https://docs.microsoft.com/ru-ru/azure/cloud-services/cloud-services-dotnet-get-started (Дата обращения 17.12.2021 г.)

f. Исакова С., Жемеров Д. Kotlin в действии / пер. с англ. Киселев А.Н. — М.: ДМК-Пресс, октябрь 2017 г., 402 с.

g. Скин Д., Гринхол Д. Kotlin. Программирование для профессионалов / пер. с англ. Киселев А.Н. — СПб.: Издательский дом «Питер», 2020 г., 464 с.

h. Официальная документация языка программирования Kotlin. URL: https://kotlinlang.ru/ (Дата обращения 21.10.2021 г.)

Практическая работа 4. Хранилище данных с реляционной структурой

В этой практической работе в части облачных технологий будет продемонстрировано создание хранилища с простой структурой данных, в части мобильных технологий будут рассмотрены особенности работы с функциями высшего порядка в Kotlin. Рассмотрен базовый синтаксис объявления лямбда-выражений, показано, как объявлять анонимные функции. Наконец, разберемся с основной обработки исключений.

Облачные технологии

Задание 1 – Выпуск репозитория разработки

Давайте рассмотрим *эмулятор* хранилища. *По* умолчанию *эмулятор хранилища* устанавливается в папку devfabric по адресу: "C:/Program Files/Microsoft SDKs/AzureNemulator/devfabric".

Существует два файла . *exe*:

1. **DFInit. exe** - инициализирует локальное хранилище и устанавливает *права* доступа к нему. При выполнении этого *файла*, если нет ошибок, должно появиться следующее окно (рисунок 4.1):

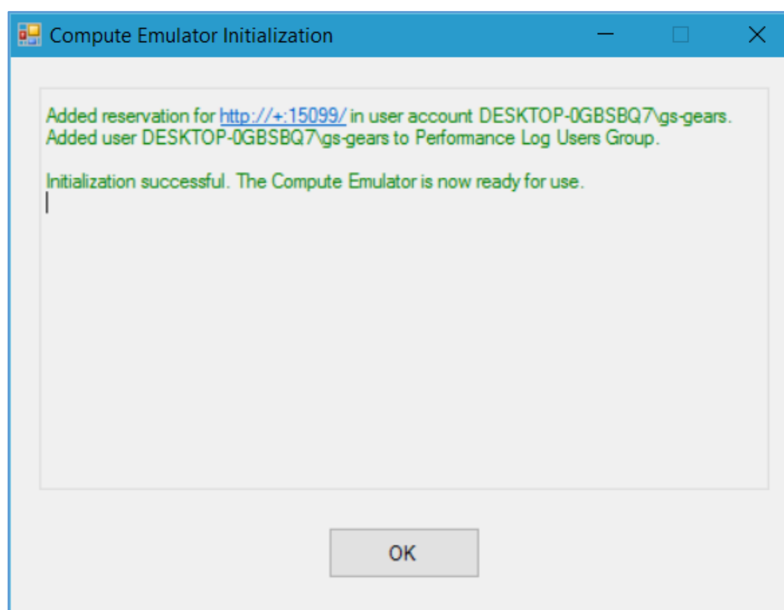


Рисунок 4.1. Эмулятор компьютера

2. **DFService.exe** - непосредственно запускает эмулятор облачного хранилища. Запустив его и подождав некоторое время, вы заметите, что в правом нижнем углу появился значок Windows *Azure*. Чтобы открыть *интерфейс* эмулятора, нужно щелкнуть правой кнопкой мыши на значке Windows *Azure* и выбрать *"Показать интерфейс эмулятора* хранилища*"*. Если ошибок нет, должно появиться следующее окно (рисунок 4.2).

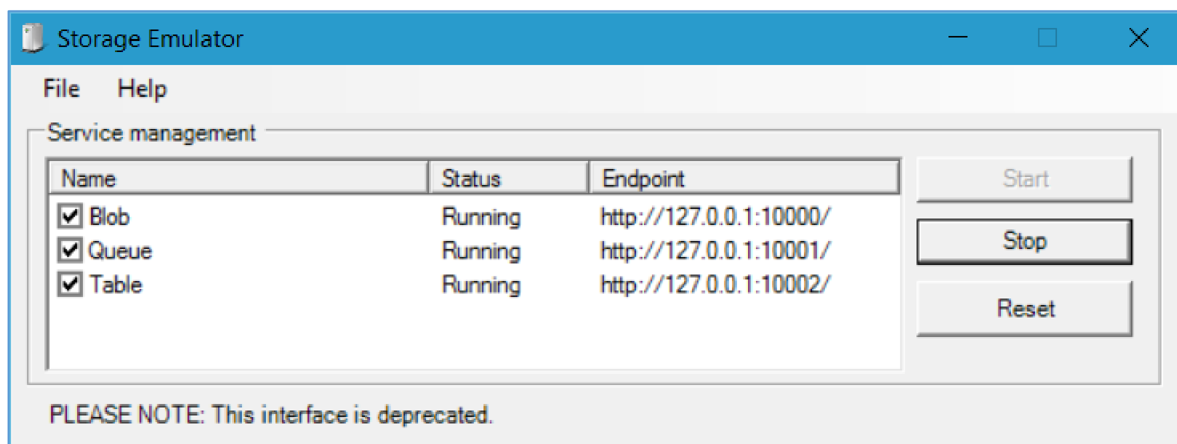


Рисунок 4.2. Эмулятор хранения данных

В окне *эмулятора хранилища* отображается состояние и конечные точки служб эмулятора: *Blob*, *Queue* и *Table*. Службы могут быть запущены, остановлены или перезапущены с потерей хранящихся в них данных.

Важно отметить, что порты, указанные для каждой из служб, должны быть свободными. В случае возникновения ошибки при запуске репозитория разработки, вполне вероятно, что указанные порты "прослушивают" другие приложения. Например, как было замечено и проверено в нескольких источниках, клиент *mtorrent,* запущенный до запуска эмулятора, конфликтует со *службой Blob,* предотвращая *доступ к* последней.

Задание 2 – Подключение к репозиторию разработки

Для подключения к эмулятору хранилища при разработке приложения (в нашем случае в среде *Visual Studio*) после создания проекта необходимо перейти к свойствам Web Paper, а затем во вкладке *Options* добавить *параметр* строки подключения (DataConnectionString) и указать *значение* "UseDevelopmentStorage=true" (рисунок 4.3).

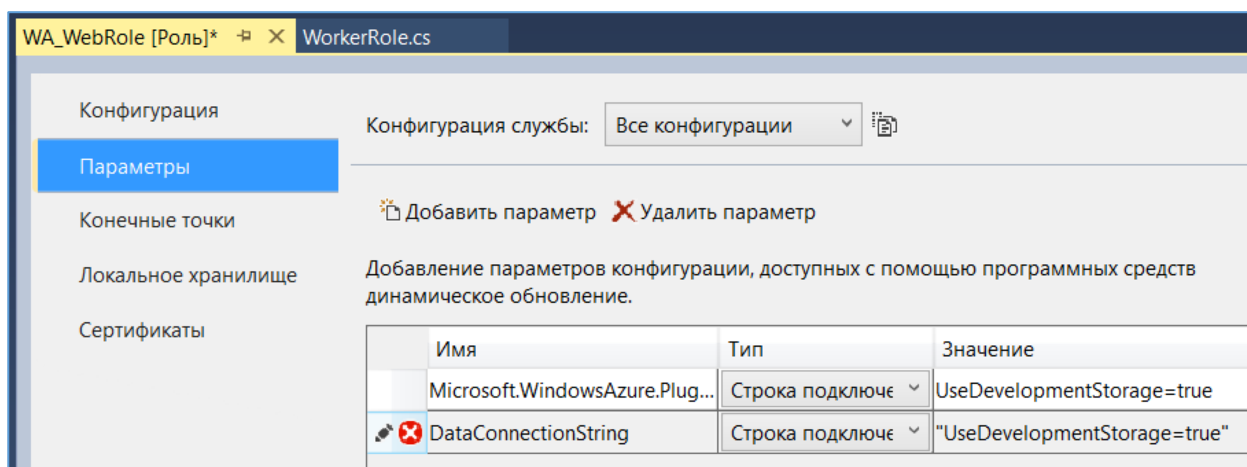


Рисунок 4.3. Свойства веб-функции

При работе непосредственно с хранилищем *Windows Azure* на той же вкладке при *создании цепочки* соединений нажмите кнопку "...". (рисунок 4.4.).

	Имя	Тип	Значение
	Microsoft.WindowsAzure.Plug...	Строка подключе	UseDevelopmentStorage=true
▸	DataConnectionString	Строка подключе	UseDevelopmentStorage=true

Рисунок 4.4. формирование цепочки связей

И в появившемся окне задайте подключение и используемую учетную запись *Windows Azure* (рисунок 4.5).

Создание строки подключения к хранилищу

Подключиться с помощью:
- (•) Эмулятор хранилища Microsoft Azure
- ( ) Ваша подписка
- ( ) Введенные вручную учетные данные

Учетная запись Майкрософт

Выберите подписку и связанную с ней учетную запись хранилища.

Подписка:
DreamSpark

Имя учетной записи:

☑ Использовать HTTPS (рекомендуется)

Предварительный просмотр строки подключения:
UseDevelopmentStorage=true

Заявление о конфиденциальности — ОК — Отмена

Рисунок 4.5. Создание окна соединительных строк

Запустите Server Browser (*См. меню* - Server Browser), и вы увидите репозиторий *Windows Azure* и репозиторий *Development* - который является отображением эмулятора (рисунок 4.6).

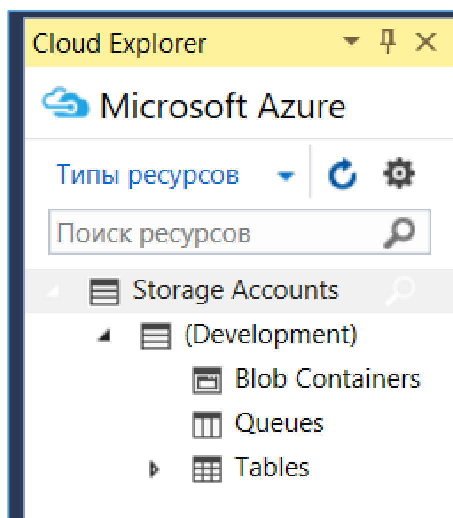


Рисунок 4.6. Облачный проводник

Вы можете использовать браузер сервера для просмотра содержимого определенных таблиц или контейнеров двоичных объектов.

Вы также можете использовать Server Explorer (кнопка Settings) для подключения или изменения *учетной записи Windows Azure и* активации существующих подписок (рисунок 4.7).

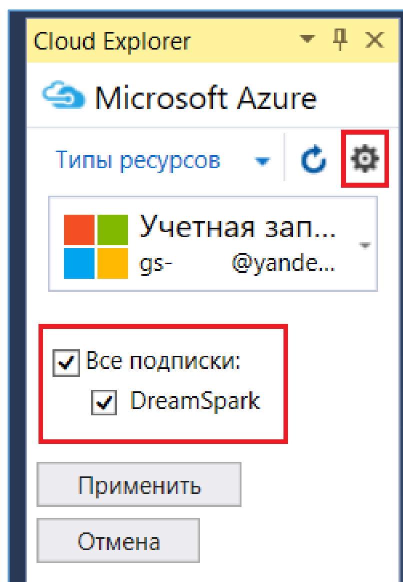


Рисунок 4.7. Конфигурация облачного браузера

Включение подписки DreamSpark предоставляет следующие возможности для использования *Windows Azure* (рисунок 4.8):

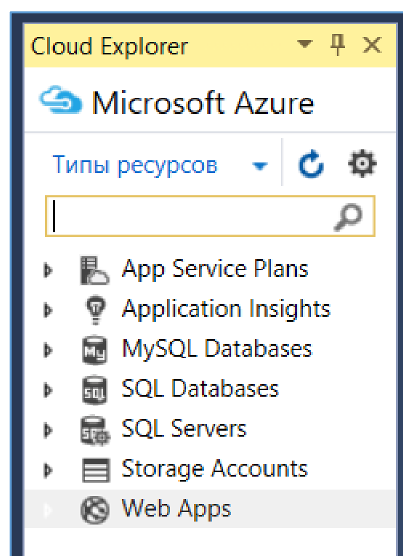


Рисунок 4.8. Облачный проводник

Задание 3 – Создайте хранилище с простой структурой данных

Чтобы продемонстрировать, как подключиться к хранилищу данных, рассмотрим небольшой пример создания хранилища с простой структурой - это будет *таблица* - *список* клиентов.

1. создайте проект консольного приложения Windows под названием *ConsoleAppAzureTable*: *меню Файл* - Создать - Проект - Windows - ConsoleApp (рисунок 4.9).

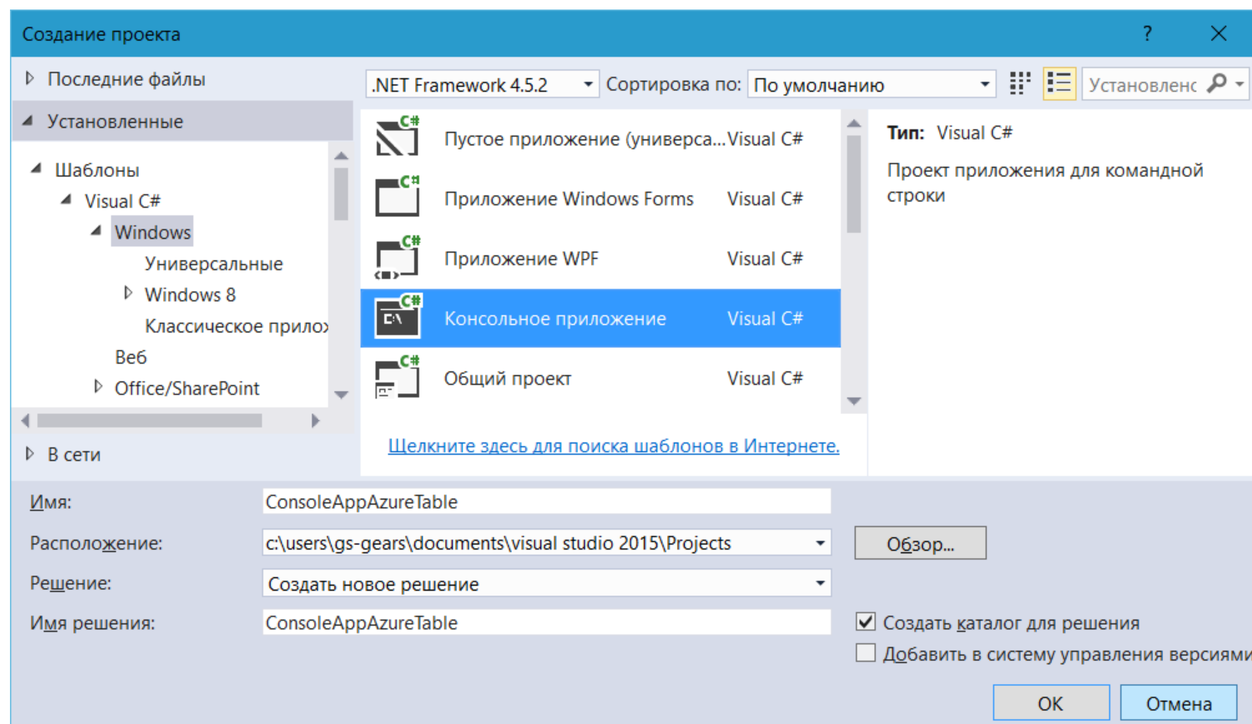


Рисунок 4.9. Создание нового проекта

Наше *приложение* подключится к эмулятору хранилища, создаст таблицу *Customer*, если она не существует, и добавит туда произвольную *запись*.

Установка необходимых пакетов через NuGet

В проект необходимо установить два пакета: Microsoft Azure Storage Service Client Library for .NET (пакет обеспечивает программный доступ к ресурсам хранения данных) и Microsoft Azure Configuration Manager Library for .NET (пакет предоставляет класс для разбора цепочки соединений конфигурационного файла, независимо от среды выполнения приложения). Выполним следующие действия:

Щелкните правой кнопкой мыши на имени проекта в обозревателе решений и выберите NuGet Package Management.

2. Найдите в Интернете "WindowsAzure.Storage" и нажмите Установить, чтобы установить клиентскую библиотеку службы хранения и зависимые компоненты.

3. Найдите в Интернете "ConfigurationManager" и нажмите Установить, чтобы установить Azure Configuration Manager.

Конфигурация цепи звеньев хранения

Клиентская библиотека Azure Storage for .NET поддерживает использование строки подключения для настройки конечных точек и учетных данных для доступа к службам хранения. Рекомендуется хранить строку подключения к хранилищу в конфигурационном файле.

Чтобы настроить строку подключения эмулятора хранилища, откройте файл app.config в обозревателе решений Visual Studio и добавьте содержимое элемента *<appSettings>*, показанного в листинге 4.1.

Листинг 4.1. Файл app.config

```
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
    </startup>
  <appSettings>
    <add key="StorageConnectionString" value="UseDevelopmentStorage=true;" />
  </appSettings>
</configuration>
```

Добавить объект в сетку

Сущности отображаются на объекты C# с помощью настраиваемого класса, производного от TableEntity. Чтобы добавить сущность в таблицу, создайте класс, определяющий свойства сущности. Следующий код определяет класс сущности, который использует имя клиента в качестве строкового ключа и имя клиента в качестве ключа раздела. Вместе ключ раздела и строковый ключ сущности уникально идентифицируют сущность в таблице. Сущности с одинаковым ключом раздела могут быть запрошены быстрее, чем с разными ключами раздела, но использование разных ключей раздела обеспечивает лучшую масштабируемость для одновременных операций. Любое свойство, которое хранится в службе таблиц, должно быть открытым свойством поддерживаемого типа, предоставляющим методы для его получения и установки. Кроме того, тип сущности *должен* предоставлять конструктор без параметров.

Нам нужно добавить в проект *класс CustomerEntity,* который будет описывать структуру сущностей нашей таблицы. Код класса показан в листинге 4.2.

Листинг 4.2. Код класса CustomerEntity

```
using Microsoft.WindowsAzure.Storage.Table;
namespace ConsoleAppAzureTable
{
    public class CustomerEntity : TableEntity
    {
        // Свойства класса
        public string Email { get; set; }
        public string PhoneNumber { get; set; }

        // Конструкторы класса
        public CustomerEntity() { }
        public CustomerEntity(string lastName, string firstName)
        {
            this.PartitionKey = lastName;
            this.RowKey = firstName;
        }
    }
}
```

Табличные операции с сущностями выполняются с помощью объекта CloudTable, созданного ранее в разделе "Создание таблицы". Операция, которая

должна быть выполнена, представлена объектом TableOperation. Следующий пример кода показывает создание объекта CloudTable и объекта CustomerEntity. Для подготовки операции создается объект TableOperation для вставки сущности customer в таблицу. Наконец, операция выполняется вызовом CloudTable.Execute (листинг 4.3).

Листинг 4.3. Код класса Program

```
using Microsoft.Azure;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
namespace ConsoleAppAzureTable
{
    class Program
    {
        static void Main(string[] args)
        {
            // Определение учетной записи из строки подключения
            CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
            CloudConfigurationManager.GetSetting("StorageConnectionString"));

            // Создание клиента таблицы Windows Azure Table
            CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

            // Создание объекта CloudTable, представляющего таблицу "Address"
            CloudTable table = tableClient.GetTableReference("Customers");

            // Создание таблицы, если она не существует
            table.CreateIfNotExists();
            /* Определение сущности, в том числе свойств ключ строки и ключ секции,
            унаследованных от родительского TableEntity */
            CustomerEntity customer1 = new CustomerEntity("Harp", "Walter");
            customer1.Email = "Walter@contoso.com";
            customer1.PhoneNumber = "425-555-0101";

            // Создание объекта TableOperation для вставки объекта класса CustomerEntity
            TableOperation insertOperation = TableOperation.Insert(customer1);

            // Выполните операцию вставки
            table.Execute(insertOperation);
        }
    }
}
```

Запустите *приложение*, убедившись, что оно работает без ошибок.

В Cloud Manager обновите вкладку *Tables* и убедитесь, что приложение создало таблицу *Customers* (рисунок 4.10).

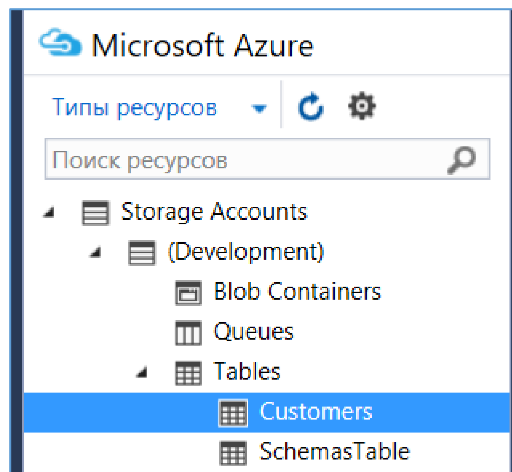


Рисунок 4.10. Таблица клиентов в Cloud Explorer

Щелкните правой кнопкой мыши на таблице и *выберите Открыть редактор таблиц*. Вы увидите, что сущность, которую мы определили, добавлена в таблицу (рисунок 4.11).

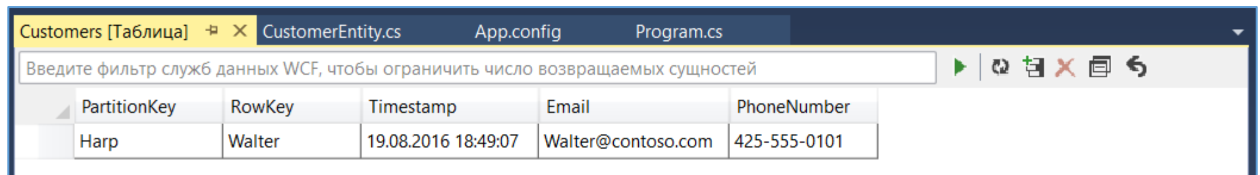


Рисунок 4.11. Редактор таблицы клиентов

Более подробно работа с *Windows Azure* Storage будет рассмотрена на следующих практических занятиях.

Мобильные технологии

Функции высшего порядка и лямбды

Функции **Kotlin** являются *first-class*, что означает, что они могут храниться в переменных и структурах данных, передаваться в качестве аргументов и возвращаться из других функций более высокого порядка. Можно работать с функциями любым способом, который возможен для других не функциональных значений.

Чтобы облегчить это, **Kotlin**, как статически типизированный язык программирования, использует семейство типов функций для представления функций и предоставляет набор специализированных языковых конструкций, таких как лямбда-выражения.

Функции высшего порядка

Функция более высокого порядка — это функция, которая принимает функции в качестве параметров или возвращает функцию.

Хороший пример — это идиома функционального программирования для коллекций, которая принимает начальное значение аккумулятора и функцию объединения и строит свое возвращаемое значение, последовательно

комбинируя текущее значение аккумулятора с каждым элементом сбора, заменяя аккумулятор:

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

В приведенном выше коде параметр *combine* имеет тип функции *(R, T) -> R*, поэтому он принимает функцию, которая принимает два аргумента типов *R* и *T* и возвращает значение типа *R*. Она вызывается внутри for-цикла, и возвращаемое значение затем присваивается *accumulator*.

Чтобы вызвать *fold*, нужно передать ему экземпляр типа функции в качестве аргумента, и лямбда-выражение (более подробно описанные ниже):

```
fun main() {
    val items = listOf(1, 2, 3, 4, 5)

    // Лямбды - это кодовые блоки, заключенные в фигурные скобки.
    items.fold(0, {
        // Когда лямбда имеет параметры,
        // они идут первыми, а затем '->'
            acc: Int, i: Int ->
        print("acc = $acc, i = $i, ")
        val result = acc + i
        println("result = $result")
        // Последнее выражение в лямбде считается возвращаемым значением:
        result
    })

    // Типы параметров в лямбде необязательны, если они могут быть выведены:
    val joinedToString = items.fold("Elements:", { acc, i -> "$acc $i" })

    println("joinedToString = $joinedToString")
}
```

Синтаксис лямбда-выражений

Полная синтаксическая форма лямбда-выражений, таких как *literals of function types*, может быть представлена следующим образом:

```
val sum = { x: Int, y: Int -> x + y }
```

Лямбда-выражение всегда заключено в скобки {...}, объявление параметров при таком синтаксисе происходит внутри этих скобок и может включать в себя аннотации типов (опционально), тело функции начинается после знака ->. Если тип возвращаемого значения не Unit, то в качестве возвращаемого типа принимается последнее (а возможно и единственное) выражение внутри тела лямбды.

Если мы вынесем все необязательные объявления, то, что останется, будет выглядеть следующим образом:

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

Обычное дело, когда лямбда-выражение имеет только один параметр. Если **Kotlin** может определить сигнатуру метода сам, он позволит нам не объявлять этот единственный параметр, и объявит его сам под именем it:

```
val ints = listOf<Int>(1, 2, -3, 4, -5)
ints.filter { it > 0 } // Эта константа имеет тип '(it: Int) -> Boolean'
```

Можно явно вернуть значение из лямбды, используя qualified return синтаксис:

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

Обратите внимание, что функция принимает другую функцию в качестве своего последнего параметра, аргумент лямбда-выражения в таком случае может быть принят вне списка аргументов, заключённого в скобках.

Анонимные функции

Единственной особенностью синтаксиса лямбда-выражений, о которой ещё не было сказано, является способность определять и назначать возвращаемый функцией тип. В большинстве случаев в этом нет особой необходимости, потому что он может быть вычислен автоматически. Однако, если у вас есть потребность в определении возвращаемого типа, вы можете воспользоваться альтернативным синтаксисом:

```
fun(x: Int, y: Int): Int = x + y
```

Параметры функции и возвращаемый тип обозначаются таким же образом, как в обычных функциях. Правда, тип параметра может быть опущен, если его значение следует из контекста:

```
ints.filter(fun(item) = item > 0)
```

Аналогично и с типом возвращаемого значения: он вычисляется автоматически для функций-выражений или же должен быть определён вручную (если не является типом Unit) для анонимных функций, которые имеют в себе блок.

Обратите внимание, что параметры анонимных функций всегда заключены в круглые скобки (...). Приём, позволяющий оставлять параметры вне скобок, работает только с лямбда-выражениями.

Одним из отличий лямбда-выражений от анонимных функций является поведение оператора return (non-local returns). Слово return , не имеющее метки (@), всегда возвращается из функции, объявленной ключевым словом fun. Это означает, что return внутри лямбда-выражения возвратит выполнение к функции, включающей в себя это лямбда-выражение. Внутри анонимных функций оператор return, в свою очередь, выйдет, собственно, из анонимной функции.

Замыкания

Лямбда-выражение или анонимная функция (так же, как и локальная функция или object expression) имеет доступ к своему замыканию, то есть к переменным, объявленным вне этого выражения или функции. В отличие от Java, переменные, захваченные в замыкании, могут быть изменены:

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

Классы исключений

Все исключения в **Kotlin** являются наследниками класса Throwable. У каждого исключения есть сообщение, трассировка стека, а также причина, по которой это исключение вероятно было вызвано.

Для того, чтобы возбудить исключение явным образом, используйте оператор *throw*

```
throw MyException("Hi There!")
```

Оператор *try* позволяет перехватывать исключения

```
try {
    // some code
}
catch (e: SomeException) {
    // handler
}
finally {
    // optional finally block
}
```

В коде может быть любое количество блоков *catch* (такие блоки могут и вовсе отсутствовать). Блоки *catch* выполняются по порядку, пока не будет найден тип исключения, который соответствует выброшенному исключению (оно не должно точно соответствовать; класс выброшенного исключения может быть подклассом объявленного), и при этом будет выполнен только один блок *catch*.

Блок *finally* (если есть) выполняется в конце, независимо от того, какой будет результат: либо после успешного завершения блока *try*, либо после выполнения блока *catch* (даже если блок *catch* выдает другое исключение), или если соответствующее перехватывание не найдено. Однако, должен быть использован как минимум один блок *catch* или *finally*.

Try — это выражение

Ключевое слово *try* является выражением, то есть оно может иметь возвращаемое значение.

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException) {
        null }
```

Возвращаемым значением будет либо последнее выражение в блоке *try*, либо последнее выражение в блоке *catch* (или блоках). Содержимое *finally* блока никак не повлияет на результат *try*-выражения.

Использование интерфейса List<T>

Тип List<out T> в Kotlin — интерфейс из пакета kotlin.collections, который предоставляет read-only операции, такие как size, get, и другие. Так же, как и в Java, он наследуется от Collection<T>, а значит и от Iterable<T>. Методы, которые изменяют список, добавлены в интерфейс MutableList<T>. То же самое относится и к Set<out T>/MutableSet<T>.

В следующем примере используется класс *Student* в качестве типа элементов, добавляемых к коллекции, хранящей информацию о студентах университета. Этот класс имеет два атрибута — полное имя студента и номер группы. Значения атрибутов передаются экземпляру класса в параметрах конструктора. Метод *ToString()* переопределен так, чтобы возвращать имя студента и номер группы.

```
class Student(var fullName: String, var group: String) {
    override fun toString(): String {
        return "$fullName группа $group"
    }
}
```

Переменная *students* определена как имеющая тип *MutableList<Student>*. Поскольку объект списка *MutableList<T>* создается с конкретным классом *Student* в качестве параметра, только объекты типа *Student* могут быть добавлены в коллекцию методом *add( )*. В интерфейсе *MutableList<T>* функция *add( )* определена как *add(element: E)*. В следующем примере создаются и добавляются в коллекцию шесть студентов. Затем с помощью функции *forEach* выполняется итерация по коллекции и с помощью лямбда-выражения каждый элемент выводится в консоль.

```
fun main() {
    val students: MutableList<Student> = mutableListOf<Student>()
    students.add(Student("Акинина Ольга", "331"))
    students.add(Student("Викулов Михаил", "331"))
    students.add(Student("Анисимова Наталья", "321"))
    students.add(Student("Квасов Виктор", "321"))
    students.add(Student("Илларионов Евнений", "334"))
    students.add(Student("Кузнецова Татьяна", "334"))

    students.forEach{ println(it)}
}
```

На рисунке 4.12 приведен результат работы программы.

```
Run: StudentKt
"C:\Program Files\Java\jdk-12.0.1\bin\java.exe"
Акинина Ольга группа 331
Викулов Михаил группа 331
Анисимова Наталья группа 321
Квасов Виктор группа 321
Илларионов Евнений группа 334
Кузнецова Татьяна группа 334
```

Рисунок 4.12. Вывод данных списка

Используя интерфейс *MutableList<T>*, можно не только добавлять элементы и обращаться к ним через перечислитесь; этот интерфейс также имеет функции для вставки и удаления элементов, очистки коллекции и копирования элементов в массив. Ниже описана еще более мощная функциональность. Пакет *kotlin.collections* представляет функции для поиска и преобразования элементов, для изменения их порядка на противоположный и тому подобных действий.

Задание 4 – Выполнить поиск элементов. Найти всех студентов в коллекции, которые учатся в группе 321.

Пакет *kotlin.collections* представляет функции расширения *first( )* и *filter( )* со следующими объявлениями:

```
fun <T> Iterable<T>.first(predicate: (T) -> Boolean): T
fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T>
```

Обе функции требуют аргумента *predicate: (T) -> Boolean*. Тип *(T) -> Boolean* является функцией, которая ссылается на метод-предикат. Предикат — это метод, возвращающий булевское значение. Если предикат возвращает *true*, значит соответствие обнаружено и элемент найден. Если он возвращает *false*, то элемент не добавляется к результату поиска. В соответствии со своим определением, *predicate: (T) -> Boolean* должен иметь единственный аргумент типа *T*. Функция *first( )* возвращает первый элемент, соответствующий предикату, а *filter( )* — все соответствующие предикату элементы в списке-коллекции.

Чтобы найти нужных студентов, которые учатся в группе 321, необходимо воспользоваться функцией *filter( )*. Для данной функции требуется создать объект предиката, например с помощью лямбда-выражения. *filter( )* возвращает список типа *List<Student>*, который используется в цикле *forEach* для итерации по всем найденным студентам и вывода их в консоль. В задаче требуется выдать на экран только определенных студентов, удовлетворяющих предикату, то для этого функции *forEach( )* и *filter( )* можно комбинировать. Функция *forEach( )* применяется к списку, возвращенному *filter( )*:

```
students.filter { it.group == "321" }.forEach { println(it) }
```

Результат работы программы приведен на рисунке 4.13.

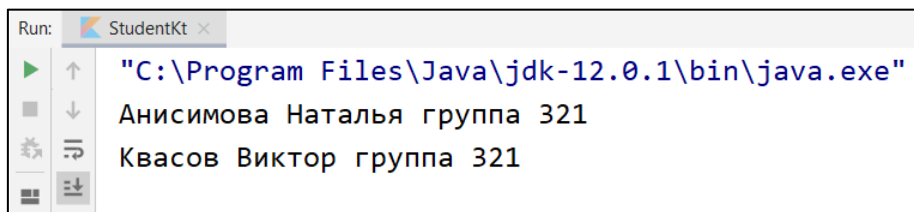


Рисунок 4.13. Результаты поиска студентов группы 321

Сортировка

Пакет kotlin.collections содержит функции, которые позволяют сортировать элементы. Функции расширения *sortBy( )* и *sortWith( )* со следующими объявлениями:

```
fun <T, R : Comparable<R>> MutableList<T>.sortBy(crossinline selector: (T) -> R?): Unit
fun <T> MutableList<T>.sortWith(comparator: kotlin.Comparator<in T>): Unit
```

Если нужно отсортировать по определенным свойствам данного объекта, можно использовать *sortBy*. Метод *sortBy* позволяет передавать функцию селектора в качестве аргумента. Функция селектора получит объект и должна вернуть значение, по которому требуется отсортировать:

```
students.sortBy{ it.fullName }
students.forEach { println(it) }
```

Если нужно, чтобы результат был возвращен как новый список, тогда следует использовать метод *sortedBy* вместо метода *sortBy*.

Результат работы программы приведен на рисунке 4.14.

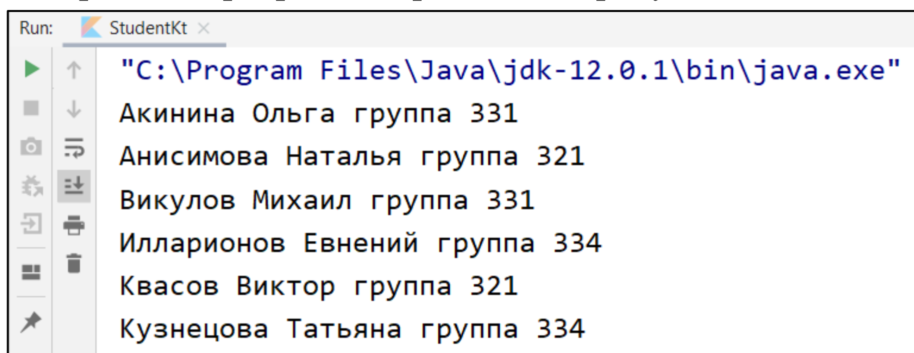


Рисунок 4.14. Сортированный список студентов по фамилии

Для более сложного использования (например, для объединения нескольких правил) можно использовать метод *sortWith()*. Метод *sortWith* позволяет передать объект *Comparator* в качестве аргумента.

Comparator<T> — это функция для метода, который имеет два параметра типа *T* и возвращает тип *int*. Если значения параметров равны, возвращается ноль, если первый параметр меньше второго, должно быть возвращено значение меньше нуля; в противном случае возвращается значение больше нуля.

```
students.sortedWith(compareBy<Student> { it.fullName }.thenBy { it.group })
    .forEach { println(it) }
```

Для сортировки порядке убывания можно использовать метод *reverse()*.

Библиографический список

1. Начало работы с Облачными службами Azure (классическая версия) и ASP.NET. URL: https://docs.microsoft.com/ru-ru/azure/cloud-services/cloud-services-dotnet-get-started (Дата обращения 17.12.2021 г.)
2. Исакова С., Жемеров Д. Kotlin в действии / пер. с англ. Киселев А.Н. — М.: ДМК-Пресс, октябрь 2017 г., 402 с.
3. Скин Д., Гринхол Д. Kotlin. Программирование для профессионалов / пер. с англ. Киселев А.Н. — СПб.: Издательский дом «Питер», 2020 г., 464 с.
4. Официальная документация языка программирования Kotlin. URL: https://kotlinlang.ru/ (Дата обращения 21.10.2021 г.)

Практическая работа 5. Работа с Windows Azure Table

В этой практической работе в части облачных технологий будет продемонстрирована работа с Windows Azure Table, в части мобильных технологий будут рассмотрены особенности работы с классами в Kotlin, базовый синтаксис объявления класса, узнаем, как объявлять методы и свойства, как использовать основные и вторичные конструкторы. Разберемся с основной наследования классов.

Облачные технологии

Задание 1 – Создание хранилища Windows Azure

Хранилище *Windows Azure* подходит для хранения реляционных данных и использует для этого возможности Windows *Azure Table*. Однако само Windows *Azure* Table Storage не хранит данные относительно одного и того же.

Давайте продемонстрируем самый простой и очевидный способ: *чтение данных* из реляционной *базы данных* строка за строкой и *запись их* в таблицу *Windows Azure*.

Во-первых, нам понадобится реляционная *база данных*. В нашей базе данных будут храниться 3 связанные сущности: *адрес*, *компания* и *сотрудник*.

В этой практической работе мы будем использовать *SQL Management Studio* для подключения к серверу *sqlexpress* и выполнения запросов.

Создайте базу данных с помощью запроса, как показано на рисунках 5.1 и 5.2, нажав кнопку "*Выполнить*".

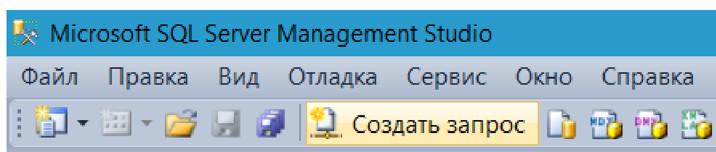


Рисунок 5.1. Создание запроса

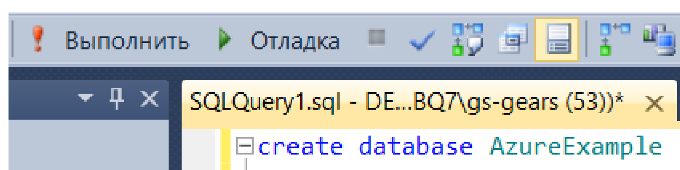


Рисунок 5.2. Выполнение запроса

Далее необходимо создать таблицы, связать их вместе и заполнить *тестовым набором* данных. Вы можете сделать это самостоятельно или просто выполнить операцию, показанную в листинге 5.1.

Листинг 5.1. Скрипт создания таблиц в БД AzureExample

```
use AzureExample

create table Address(
AddresID int identity(1,1) primary key,
Country nvarchar(max),
City nvarchar(max),
Street nvarchar(max),
House int)
```

```
create table Firm(
FirmID int identity(1,1) primary key,
NameOf nvarchar(max),
Telephone nvarchar(10),
Email nvarchar(max),
AddressKey int not null
)

create table Employee(
EmployeeID int identity(1,1) primary key,
FirstName nvarchar(max),
LastName nvarchar(max),
Telephone nvarchar(10),
FirmKey int
)

Alter table dbo.Firm add constraint
  FK_Firm_Address foreign key
  (
  AddressKey
  ) references dbo.Address
  (
  AddresID
  ) on update no action
   On delete no action

alter table dbo.Employee add constraint
FK_Employee_Firm foreign key
(
FirmKey
)
references dbo.Firm
(
FirmID
)
on update no action
on delete no action

insert into Address
values ('RF', 'Tomsk', 'Evergreen Terrace', '247')
insert into Firm
values ('Firm1', 'xxx-xx-xx', 'firm1@testmail.com', '1')
insert into Firm
values ('Firm2', 'xxx-xx-xx', 'firm2@testmail.com', '1')
insert into Employee
values ('Ivan', 'Ivanov', 'x-xxx-xx','1')
insert into Employee
values ('Victor', 'Romanov', 'x-xxx-xx','1')
insert into Employee
values ('Alex', 'Petrov', 'x-xxx-xx','2')
```

В результате получаются три таблицы. *Диаграмма* данных в созданной базе данных показана ниже (рисунок 5.3).

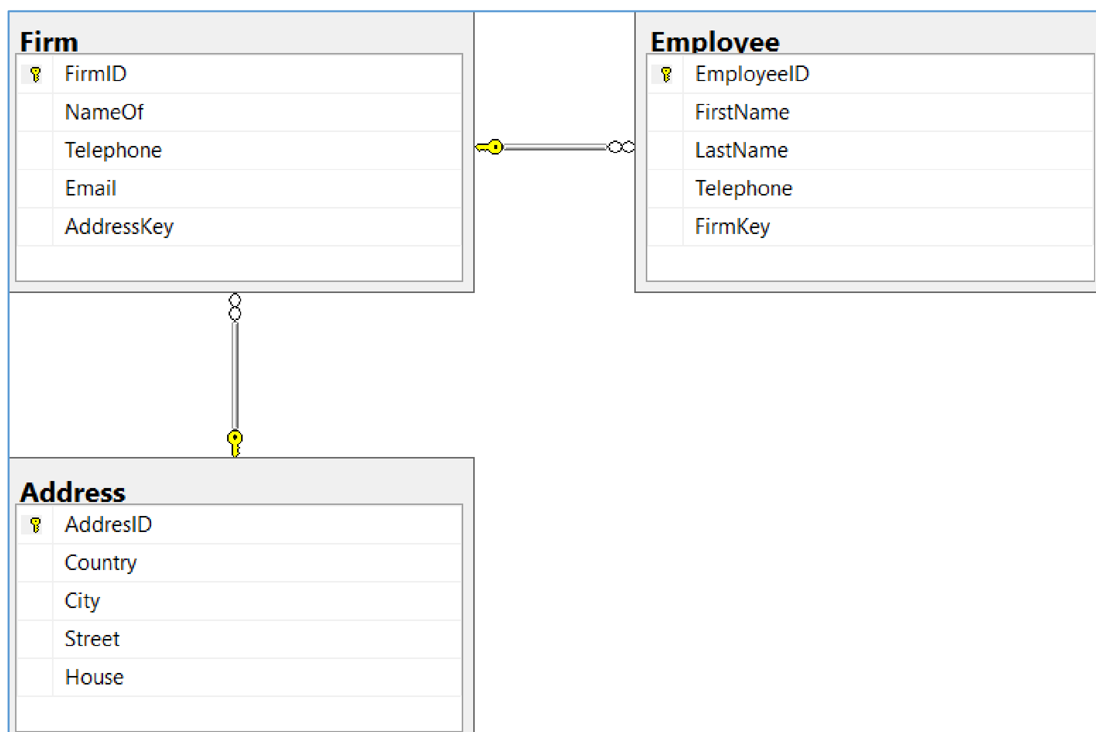


Рисунок 5.3. Диаграмма базы данных

Теперь необходимо перенести эту реляционную структуру в таблицу. Для этого необходимо сопоставить таблицу *Windows Azure* с коллекцией связанных сущностей.

Создайте таблицу *Windows Azure "Relational"*, которая будет содержать сущности *Address*, *Company* и *Employee*.

Ключ раздела должен соответствовать строке в таблице *Azure* с экземпляром сущности реляционной базы данных, т.е. для всех сотрудников *ключ* раздела должен быть *"Employee"*.

Значения ключей атрибутов в реляционных таблицах будут значениями RowKey таблицы *"Relational"*, поэтому пара *ключ* раздела - *ключ* строки будет уникальным идентификатором, указывающим, что строка *Azure* - принадлежит определенной сущности *в* исходной *базе данных* и определяющим *экземпляр сущности*.

Значения остальных атрибутов должны быть переданы без изменений.

Нет необходимости создавать новый *проект VS*, поэтому мы изменим *Program.cs* из проекта, созданного в предыдущей практической работе.

Кроме того, мы должны создать классы - соответствующие сущностям в реляционной *базе данных*.

Листинг 5.2. Адрес class.cs

```
class Address : TableEntity
{
    public Address(string partitionKey, string rowKey) :
        base(partitionKey, rowKey) {
    }
    public Address() { }
    public string country { get; set; }
    public string city { get; set; }
    public string street { get; set; }
```

```
        public int house { get; set; }
    }
```

Листинг 5.3. Класс Firm.cs

```
    class Firm : TableEntity
    {
        public Firm(string partitionKey, string rowKey) :
            base(partitionKey, rowKey)
        {
        }
        public Firm() { }
        public string nameof { get; set; }
        public string telephone { get; set; }
        public string email { get; set; }
        public int adresskey { get; set; }

    }
```

Листинг 5.4. Класс Employee.cs

```
    class Employee : TableEntity
    {
        public Employee(string partitionKey, string rowKey) :
            base(partitionKey, rowKey)
        {
        }
        public Employee() { }
        public string firstname { get; set; }
        public string lastname { get; set; }
        public string telephone { get; set; }
        public int firmkey { get; set; }

    }
```

Листинг 5.5. Основные методы перечислены ниже:

```
        static void Main(string[] args)
        {
            // Определение учетной записи из строки подключения
            CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
            CloudConfigurationManager.GetSetting("StorageConnectionString"));

            // Создание клиента таблицы Windows Azure Table
            CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

            // Создание объекта CloudTable, представляющего таблицу "Address"
            CloudTable table = tableClient.GetTableReference("Relational");

            // Создание таблицы, если она не существует
            table.CreateIfNotExists();

            // Определение параметров подключения к БД, измените строку подключения
            // соответствующим образом, для соединения с вашим sql - сервером
            SqlConnection conn = new SqlConnection("Data Source=DESKTOP-0GBSBQ7;" +
                "Initial Catalog = AzureExample; Integrated Security = true; ");

            // Импорт данных из таблицы Address
            // Указываем команду для чтения данных из базы
            SqlCommand cmd = new SqlCommand("SELECT * FROM Address", conn);
            conn.Open();
            SqlDataReader adr = cmd.ExecuteReader();
```

```
            while (adr.Read())
            {
                // Создание объекта TableOperation для вставки объекта класса Address
                TableOperation insertOperation = TableOperation.Insert(new Address
                {
                    PartitionKey = "Address",
                    RowKey = adr["AddresID"].ToString(),
                    country = adr["Country"].ToString(),
                    city = adr["City"].ToString(),
                    street = adr["Street"].ToString(),
                    house = Convert.ToInt32(adr["House"].ToString())
                });

                // Выполнение операции вставки
                table.Execute(insertOperation);
            }
            adr.Close();

            // Импорт данных из таблицы Firm
            cmd.CommandText = "SELECT * FROM Firm";
            SqlDataReader frm = cmd.ExecuteReader();
            while (frm.Read())
            {
                TableOperation insertOperation = TableOperation.Insert(new Firm
                {
                    PartitionKey = "Firm",
                    RowKey = frm["FirmID"].ToString(),
                    nameof = frm["NameOf"].ToString(),
                    telephone = frm["Telephone"].ToString(),
                    email = frm["Email"].ToString(),
                    adresskey = Convert.ToInt32(frm["AddressKey"].ToString())
                });
                table.Execute(insertOperation);
            }
            frm.Close();

            // Импорт данных из таблицы Employee
            cmd.CommandText = "SELECT * FROM Employee";
            SqlDataReader emp = cmd.ExecuteReader();
            while (emp.Read())
            {
                TableOperation insertOperation = TableOperation.Insert(new Employee
                {
                    PartitionKey = "Employee",
                    RowKey = emp["EmployeeID"].ToString(),
                    firstname = emp["FirstName"].ToString(),
                    lastname = emp["LastName"].ToString(),
                    telephone = emp["Telephone"].ToString(),
                    firmkey = Convert.ToInt32(emp["FirmKey"].ToString())
                });
                table.Execute(insertOperation);
            }
            emp.Close();
            // Закрытие подключения
            conn.Close();
        }
```

Запустите *приложение* и дождитесь окончания его работы. В Cloud Manager обновите вкладку *Tables* и убедитесь, что приложение создало

реляционную таблицу. Если вы решите просмотреть его, вы увидите, что данные из *реляционной базы данных были* успешно перенесены в таблицу (рисунок 5.4).

Relational [Таблица] | Employee.cs | Firm.cs | Address.cs | App.config | Program.cs

Введите фильтр служб данных WCF, чтобы ограничить число возвращаемых сущностей

PartitionKey	RowKey	Timestamp	country	city	street	house	firstname	lastname	telephone	firmkey	nameof
Address	1	19.08.2016 20:2...	RF	Tomsk	Evergre...	247					
Employee	1	19.08.2016 20:2...					Ivan	Ivanov	x-xxx-xx	1	
Employee	2	19.08.2016 20:2...					Victor	Romanov	x-xxx-xx	1	
Employee	3	19.08.2016 20:2...					Alex	Petrov	x-xxx-xx	2	
Firm	1	19.08.2016 20:2...							xxx-xx-xx		Firm1
Firm	2	19.08.2016 20:2...							xxx-xx-xx		Firm2

Рисунок 5.4. Редактор реляционных таблиц

Мобильные технологии

Классы

Класс — это пользовательский тип данных (user defined type, UDT), который состоит из данных (часто называемых атрибутами или свойствами) и функциями для выполнения с этими данными различных действий (эти функции обычно называются методами).

Классы в Kotlin, как и в других языках программирования, служат для разделения кода и повышения независимости отдельных частей программы друг от друга.

Классы объявляются с использованием ключевого слова class:

```
class Invoice { /*...*/ }
```

Объявление класса состоит из имени класса, заголовка класса (с указанием параметров его типа, основного конструктора и т. д.) и тела класса, заключенного в фигурные скобки. И заголовок, и тело являются необязательными; если у класса нет тела, фигурные скобки могут быть опущены.

```
class Empty
```

Конструкторы

Конструктор класса — это специальный метод, который вызывается каждый раз при создании нового экземпляра класса (объекта) и служит для того, чтобы подготовить объект к дальнейшему использованию.

Класс в Kotlin может иметь первичный конструктор и один или несколько вторичных конструкторов. Основной конструктор является частью заголовка класса: он идет после имени класса (и необязательных параметров типа).

```
class Person constructor(firstName: String) { /*...*/ }
```

Если основной конструктор не имеет аннотаций или модификаторов видимости, ключевое слово *constructor* может быть опущено:

```
class Person (firstName: String) { /*...*/ }
```

Основной конструктор не может содержать код. Код инициализации может быть размещен в блоках инициализатора, которые имеют префикс с ключевым словом *init*.

Во время инициализации экземпляра блоки инициализатора выполняются в том же порядке, в котором они отображаются в теле класса, с чередованием с инициализаторами свойства:

```
class InitOrderDemo(name: String) {
    val firstProperty = "Первое свойство: $name".also(::println)

    init {
        println("Первый инициализирующий блок, который выводит ${name}")
    }

    val secondProperty = "Второе свойство: ${name.length}".also(::println)

    init {
        println("Второй инициализирующий блок, который выводит ${name.length}")
    }
}

fun main() {
    InitOrderDemo("hello")
}
```

```
First property: hello
First initializer block that prints hello
Second property: 5
Second initializer block that prints 5
```

Обратите внимание, что параметры первичного конструктора могут использоваться в блоках инициализатора. Они также могут использоваться в инициализаторах свойств, объявленных в теле класса:

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

Фактически, для объявления свойств и их инициализации из основного конструктора, Kotlin имеет лаконичный синтаксис:

```
class Person(val firstName: String, val lastName: String, var age: Int) { /*...*/ }
```

Подобно обычным свойствам, свойства, объявленные в основном конструкторе, могут быть изменяемыми (*var*) или только для чтения (*val*).

Вторичные конструкторы

Класс также может объявлять вторичные конструкторы, которые помечаются ключевым словом *constructor*:

```
class Person {
    var children: MutableList<Person> = mutableListOf<Person>();
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

Если у класса есть первичный конструктор, каждый вторичный конструктор должен делегировать первичному конструктору, прямо или косвенно, через другой вторичный конструктор(ы). Делегирование другому конструктору того же класса выполняется с помощью ключевого слова *this*:

```
class Person(val name: String) {
    var children: MutableList<Person> = mutableListOf<Person>();
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Обратите внимание, что код в блоках инициализатора фактически становится частью основного конструктора. Делегирование первичному конструктору происходит в качестве первого оператора вторичного конструктора, поэтому код во всех блоках инициализатора и инициализаторах свойства выполняется перед телом вторичного конструктора. Даже если у класса нет первичного конструктора, делегирование все равно происходит неявно, и блоки инициализатора все еще выполняются:

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}

fun main() {
    Constructors(1)
}
```

```
Init block
Constructor 1
```

В классах можно определить любое количество конструкторов. Эти специальные методы классов позволяют создавать объекты данного класса, настраивая при этом их исходное состояние нужным вам образом.

Создание экземпляров классов

Чтобы создать экземпляр класса, мы вызываем конструктор как обычную функцию:

```
fun main() {
    val invoice = Invoice()
    val customer = Customer("Иван Иванов")
}
```

Обратите внимание, что в Kotlin нет ключевого слова *new* для создания экземпляра класса.

Классы могут содержать:

– конструкторы и инициализаторы;
– функции;
– свойства;
– вложенные и внутренние классы;
– объявления объектов.

Наследование

Все классы в Kotlin имеют общий суперкласс *Any*, который является суперклассом по умолчанию для класса без объявленных супертипов:

```
class Example // Неявно наследуется от Any
```

Any содержит три метода: *equals()*, *hashCode()* и *toString()*. Таким образом, они определены для всех классов Kotlin.

Чтобы объявить явный супертип, поместите тип после двоеточия в заголовок класса:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

Если у производного класса есть первичный конструктор, базовый класс может (и должен) быть инициализирован прямо здесь, используя параметры первичного конструктора.

Переопределяемые методы

Kotlin требует явных модификаторов для переопределяемых элементов (они называются открытыми *open*), а для переопределений *override*:

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}
```

Модификатор переопределения *override* необходим для *Circle.draw()*. Если бы он отсутствовал, компилятор жаловался бы. Если в функции нет модификатора *open*, такого как *Shape.fill()*, объявление метода с той же сигнатурой в подклассе является недопустимым либо с переопределением *override*, либо без него. Модификатор *open* не действует при добавлении к членам конечного класса (т.е. класса без модификатора *open*).

Порядок инициализации производного класса

Во время создания нового экземпляра производного класса инициализация базового класса выполняется в качестве первого шага (которому предшествует только оценка аргументов для конструктора базового класса) и далее происходит запуск логики инициализации производного класса.

```
class Derived(
    name: String,
    val lastName: String
) : Base(name.capitalize().also { println("Аргумент для класса Base: $it") }) {

    init { println("Инициализация производного класса") }

    override val size: Int =
        (super.size + lastName.length).also {
            println("Инициализация значения в классе Derived: $it") }
}
fun main() {
    println("Создание класса Derived(\"hello\", \"world\")")
    val d = Derived("hello", "world")
}
```

```
Создание класса Derived("hello", "world")
Аргумент для класса Base: Hello
Инициализация базового класса
Инициализация значения в классе Base: 5
Инициализация производного класса
Инициализация значения в классе Derived: 10
```

Это означает, что ко времени выполнения конструктора базового класса свойства, объявленные или переопределенные в производном классе, еще не инициализированы. Если любое из этих свойств используется в логике инициализации базового класса (прямо или косвенно, через другую переопределенную реализацию открытого члена), это может привести к некорректному поведению или сбою во время выполнения. Поэтому при разработке базового класса следует избегать использования открытых членов в конструкторах, инициализаторах свойств и блоках инициализации.

Рассмотрим программную систему, которая осуществляет учет данных по двум группам сотрудников: менеджерам и продавцам. Требуется создать программу, которая обеспечивает:

– ввод с клавиатуры общих данных по сотрудникам:

- идентификационного номера;
- фамилии;
- имени;
- отчества;
- заработная плата;

– для менеджеров ввод с клавиатуры данных, о имеющихся у них опционов на акции (целое число);
– для продавцов ввод с клавиатуры данных, об объемах продаж;
– вывод на экран данных по сотрудникам.

В процессе анализа предметной области был выявлен один общий родительский класс *Employee*, который имеет данные и функциональность для всех сотрудников. Этот класс должен отвечать за ввод общих данных по сотрудникам и вывод их на экран. Классами более специфическими по отношению к классу *Employee* являются классы *Manager* и *SalePerson*. Классы *Manager* и *SalePerson* можно сконструировать на базе класса *Employee*, расширив данные и функциональность последнего. В таком варианте класс *Employee* нужно рассматривать как родительский класс, а классы *Manager* и *SalePerson*, как дочерние. Так класс *Manager* должен дополнять данные класса *Employee* в части имеющихся у них опционов на акции, а класс *SalePerson* − в части учета объема продаж. Кроме того, класс *Manager* должен обеспечивать ввод и вывод информации по опционам, а класс *SalePerson* – по объемам продаж.

Диаграмма классов предметной области на языке моделирования *UML* приведена на рисунке 5.5.

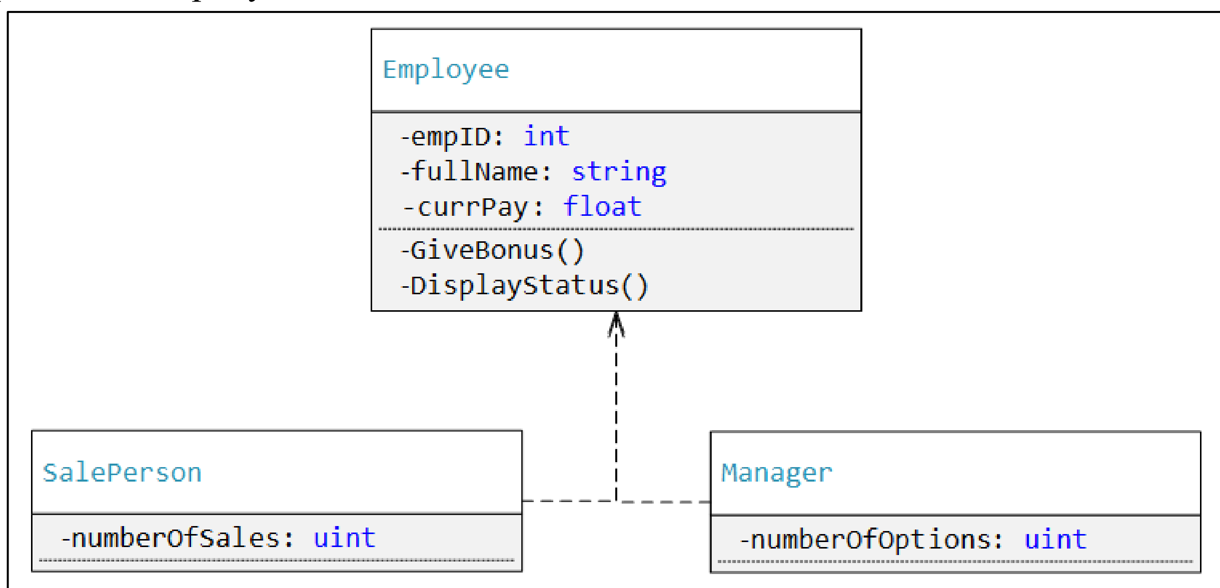


Рисунок 5.5. Диаграмма классов предметной области

Как видно из рисунка, сотрудниками (*Employee*) являются и продавец (*SalePerson*), и менеджер (*Manager*). В модели классического наследования

базовые классы (в нашем случае *Employee*) используются для того, чтобы определить характеристики, которые будут общими для всех производных классов. Производные классы (такие как *SalePerson* и *Manager*) расширяют унаследованную функциональность за счет своих собственных, специфических членов.

Задание 2 – Провести последовательную разработку и тестирование программной системы, описание которой приведено выше.

a. Создание нового Kotlin файла:

Добавьте в проект новый файл, щелкнув правой кнопкой мыши на папке **src** в окне инструментов проекта. Выберите New – Kotlin File/Class. Выбрать тип Class и ввести имя класса **Employee** (Сотрудник). Нажать кнопку «Enter».

b. Написание кода класса Employee

В появившемся окне необходимо ввести программный код, приведенный в листинге 5.6.

Листинг 5.6. Программный код класса Employee

```
// Класс "Сотрудник"
open class Employee {
    // Поля класса
    var empId = 0               // Идентификатор
    var fullName = ""           // Имя Фамилия
    var currPay = 0f            // Зар. плата

    // Методы класса
    fun giveBonus(amount: Float){
        currPay += amount
    }

    fun displayStatus(){
        println("Код:        $empId")
        println("ФИО:        $fullName")
        println("Зарплата:   $currPay")
    }
}
```

c. Написание кода класса SalePerson

Добавим в проект аналогичным образом класс *SalePerson,* содержащий данные об объеме продаж (*numberOfSales*) и отвечающий за ввод объема продаж и вывод его на экран, при этом указав в качестве базового класса созданный ранее класс *Employee*, при этом пометив его как *open*. В языке Kotlin указатель на базовый класс выглядит как двоеточие (:).

В появившемся окне необходимо ввести программный код, приведенный в листинге 5.7.

Листинг 5.7. Программный код класса SalePerson

```
class SalePerson: Employee() {
    var numberOfSales = 0   // Объем продаж
}
```

d. Написание кода класса Manager

Аналогичным образом добавим в проект класс *Manager,* содержащий данные о имеющихся у менеджеров опционов на акции (*numberOfOptions*) и отвечающий за ввод и вывод данных.

Листинг 5.8. Программный код класса Manager

```
class Manager: Employee() {
    var numberOfOptions: Int = 0    // Количество опционов на акции
}
```

e. Создание нового Kotlin файла:

Добавьте в проект новый файл, щелкнув правой кнопкой мыши на папке **src** в окне инструментов проекта. Выберите New – Kotlin File/Class. Ввести имя файла, например, Program. Нажать кнопку «Enter».

В данном файле будет определена функция main() и заданы значения для полей и осуществлен вывод информации о сотрудниках на экран.

Листинг 5.9. Программный код функции main()

```
fun main() {
    val manager = Manager()
    manager.empId = 1
    manager.fullName = "Иванов Иван Иванович"
    manager.currPay = 25000F
    manager.numberOfOptions = 500
    manager.giveBonus(5000F)
    manager.displayStatus()

    val salePerson = SalePerson()
    salePerson.empId = 2
    salePerson.fullName = "Петров Петр Петрович"
    salePerson.currPay = 15000F
    salePerson.numberOfSales = 40
    salePerson.giveBonus(3000F)
    salePerson.displayStatus()
}
```

Выполните компиляцию проекта и запустите main() на выполнение.

```
Код:       1
ФИО:       Иванов Иван Иванович
Зарплата:  30000.0
Код:       2
ФИО:       Петров Петр Петрович
Зарплата:  18000.0
```

Рисунок 5.6. Результат работы листинга 5.9

Унаследованный метод *GiveBonus()* производными классами *SalePerson* и *Manager* пока работает абсолютно одинаково в отношении обоих объектов. Но если, при поощрении продавцов нужно учесть их объем продаж, а менеджерам помимо денежного поощрения нужно выдать дополнительные опционы на акции, то появляется проблема: «Как заставить один и тот же метод по-разному реагировать на объекты разных классов?»

Для решения этой задачи в ООП предусмотрено понятие полиморфизма. Полиморфизм позволяет переопределять реакцию объекта производного класса на метод, определенный в базовом классе. Для реализации полиморфизма необходимо использовать ключевые слова Kotlin: **open** и **override**. Если базовый класс определяет метод, который должен быть замещен в производном классе, этот метод должен быть объявлен как *open*.

Переопределяемая функция (метод) — это функция, которая объявляется в базовом классе с использованием ключевого слова open и переопределяется в одном или нескольких производных классах. Таким образом, каждый производный класс может иметь собственную версию переопределяемой функции.

f. Формирование переопределяемых функций в классе Employee

В классе *Employee* объявим переопределяемый метод для начисления бонусов к заработной плате сотрудников и переопределяемый метод вывода информации.

Листинг 5.10. Формирование переопределяемых функций класса Employee

```
/**
 * Переопределяемый метод для начисления бонусов
 */
open fun giveBonus(amount: Float){
    currPay += amount
}

/**
 * Переопределяемый метод вывода информации
 */
open fun displayStatus(){
    println("Код:       $empId")
    println("ФИО:       $fullName")
    println("Зарплата:  $currPay")
}
```

g. Переопределение функций в классе Manager

Чтобы переопределить функцию, необходимо заново определить ее в производном классе, используя ключевое слово **override**.

Листинг 5.11. Программный код класса Manager

```
class Manager: Employee() {
    var numberOfOptions: Int = 0    // Количество опционов на акции

    /**
     * Переопределенный метод для начисления бонусов
     */
    override fun giveBonus(amount: Float){
        // Использование кода функции родительского класса
        super.giveBonus(amount)
        // Опционы на акции: увеличиваем их количество
        numberOfOptions += amount.toInt() * 10
    }

    /**
     * Переопределенный метод вывода информации
     */
    override fun displayStatus(){
        super.displayStatus()
        println("Количество опционов:  $numberOfOptions")
    }
}
```

h. Переопределение функций в классе SalePerson

Чтобы переопределить функцию, необходимо заново определить ее в производном классе, используя ключевое слово **override**.

Листинг 5.12. Программный код класса SalePerson

```
class SalePerson: Employee() {
    var numberOfSales = 0   // Объем продаж

    /**
     * Переопределенный метод для начисления бонусов
     */
    override fun giveBonus(amount: Float){
        val salesBonus = when(numberOfSales){
            in 0..100 -> 10F
            in 101..200 -> 15F
            else -> 20F
        }

        // Использование кода функции родительского класса
        super.giveBonus(amount * salesBonus)
    }

    /**
     * Переопределенный метод вывода информации
     */
    override fun displayStatus(){
        super.displayStatus()
        println("Объем продаж:  $numberOfSales")
    }
}
```

Обратите внимание, что в определении каждого из замещенных методов используется вызов этого же метода базового класса. Таким образом, нет

необходимости заново определять всю логику замещенного метода в производном классе: вполне достаточно воспользоваться вариантом метода по умолчанию, определенном в базовом классе, дополнив его нужными действиями.

Метод *DisplayStatus()* также определен как переопределяемый *open*, и он замещен в производных классах таким образом, чтобы показывать текущий объем продаж, если речь идет о продавце, или количество опционов, имеющееся в настоящее время в распоряжении менеджера.

Теперь, когда для каждого из производных классов определены собственные варианты этих двух методов, объекты разных классов ведут себя по-разному.

i. Запуск программы

Чтобы просмотреть результат выполнения программы, нужно выполнить Run – Run ‘ProgramKt’ или нажать кнопку в виде треугольника (рисунок 5.7).

```
Код:        1
ФИО:        Иванов Иван Иванович
Зарплата:   30000.0
Количество опционов:   50500
Код:        2
ФИО:        Петров Петр Петрович
Зарплата:   45000.0
Объем продаж:   40
```

Рисунок 5.7. Результат выполнения программы

Библиографический список

1. Миграция в облачные службы (расширенная поддержка) с помощью портала Azure. URL: https://docs.microsoft.com/ru-ru/azure/cloud-services-extended-support/in-place-migration-portal?toc=/azure/cloud-services/toc.json (Дата обращения 17.12.2021 г.)
2. Исакова С., Жемеров Д. Kotlin в действии / пер. с англ. Киселев А.Н. — М.: ДМК-Пресс, октябрь 2017 г., 402 с.
3. Скин Д., Гринхол Д. Kotlin. Программирование для профессионалов / пер. с англ. Киселев А.Н. — СПб.: Издательский дом «Питер», 2020 г., 464 с.
4. Официальная документация языка программирования Kotlin. URL: https://kotlinlang.ru/ (Дата обращения 21.10.2021 г.)

Практическая работа 6. Работа с Windows Azure Blob

В этой практической работе вы создадите простое приложение GuestBook, которое продемонстрирует ряд возможностей платформы Windows Azure, включая использование веб-ролей и ролей приложений, хранение двоичных и табличных объектов и постановку в очередь.

Приложение "Гостевая книга" использует веб-функции для создания пользовательского интерфейса, который позволяет как просматривать содержимое гостевой книги, так и добавлять в нее новые записи. Каждая запись включает имя, текстовое сообщение и изображение. Приложение также использует функцию приложения, которая генерирует эскизы для изображений, добавленных пользователями.

Когда пользователь добавляет новую запись, веб-функция загружает соответствующее изображение в магазин бинарных объектов, а затем добавляет новую сущность в табличный магазин, содержащую информацию, введенную пользователем, и ссылку на изображение в магазине бинарных объектов. При обращении к ней веб-функция форматирует эту информацию, чтобы пользователь мог просмотреть содержимое гостевой книги.

После сохранения изображения и добавления сущности в табличное хранилище веб-функция помещает рабочий элемент в очередь, указывая, что изображение необходимо обработать. Прикладная функция извлекает рабочий элемент из очереди, извлекает изображение из магазина двоичных объектов и создает эскиз - уменьшенную версию исходного изображения. Использование очередей является рекомендуемым подходом для организации взаимодействия облачных сервисов. Преимущество организации слабосвязанных элементов заключается в том, что их можно тестировать и масштабировать отдельно.

Задание 1 – Создание проекта в Visual Studio

В этом задании вы создадите новый проект типа Cloud Service.

1. Откройте Visual Studio от имени администратора: выберите **Пуск | Все программы | Microsoft Visual Studio 2015**, щелкните правой кнопкой мыши на ярлыке **Microsoft Visual Studio 2015** и выберите **Запуск от имени администратора**.
2. Если появится диалоговое окно **Контроль учетных записей пользователей**, нажмите **Продолжить**.
3. В меню **Файл** выберите **Новый**, а затем **Проект**.
4. В диалоговом окне **New Project** разверните список **Installed Templates** и выберите соответствующий узел для предпочитаемого языка (Visual C#) и выберите **Cloud**. Выберите шаблон проекта **Windows Azure Cloud Service**, дайте проекту имя (поле **Name)** - в нашем случае **GuestBook**, задайте местоположение (поле **Location**), **переименуйте** Solution в **GuestBook** и убедитесь, что установлен флажок **Create directory** for the solution. Нажмите **OK,** чтобы создать проект.

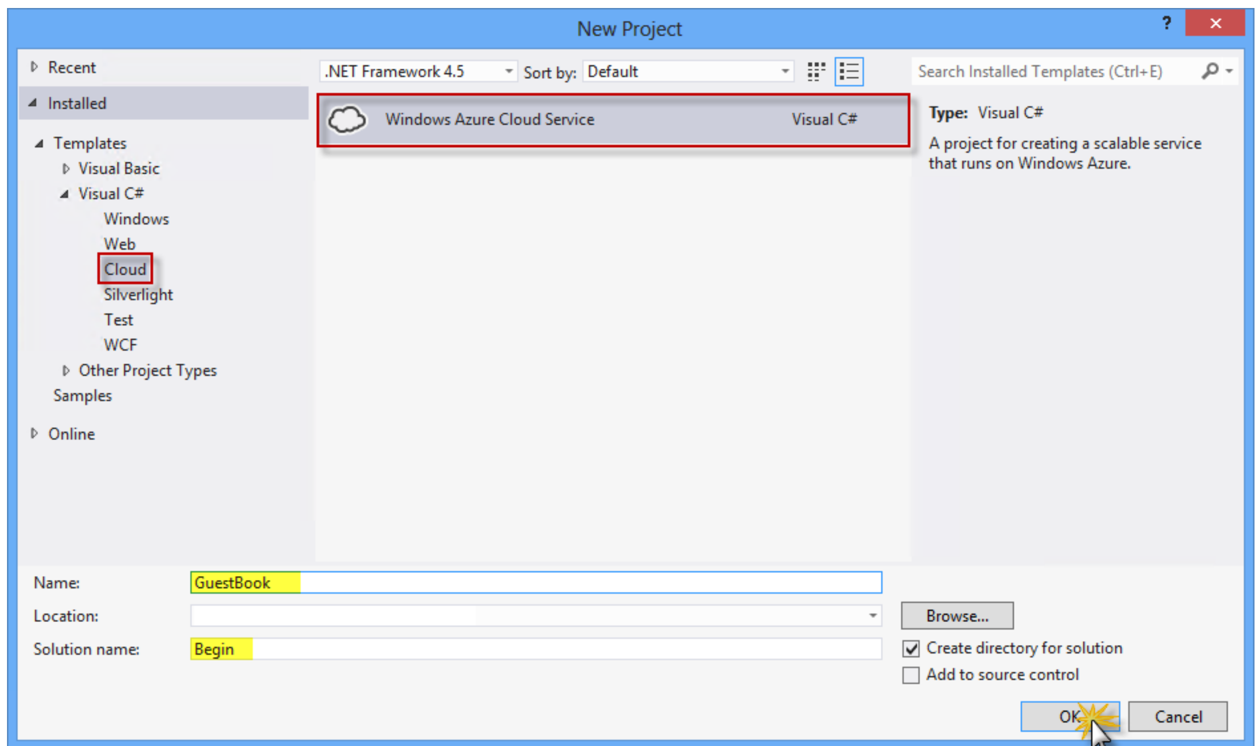


Рисунок 6.1. Создание нового проекта для Windows Azure

5. В диалоговом окне **New Windows Azure Project** в панели **Roles** разверните вкладку с предпочтительным языком (Visual C#), выберите **ASP.NET Web Role** и нажмите кнопку со стрелкой вправо (>), чтобы добавить экземпляр этого типа роли в ваше решение. Перед закрытием этого диалога выберите добавленную роль в правой панели, нажмите на значок карандаша и переименуйте роль в **GuestBook_WebRole**. Нажмите **OK,** чтобы создать решение.

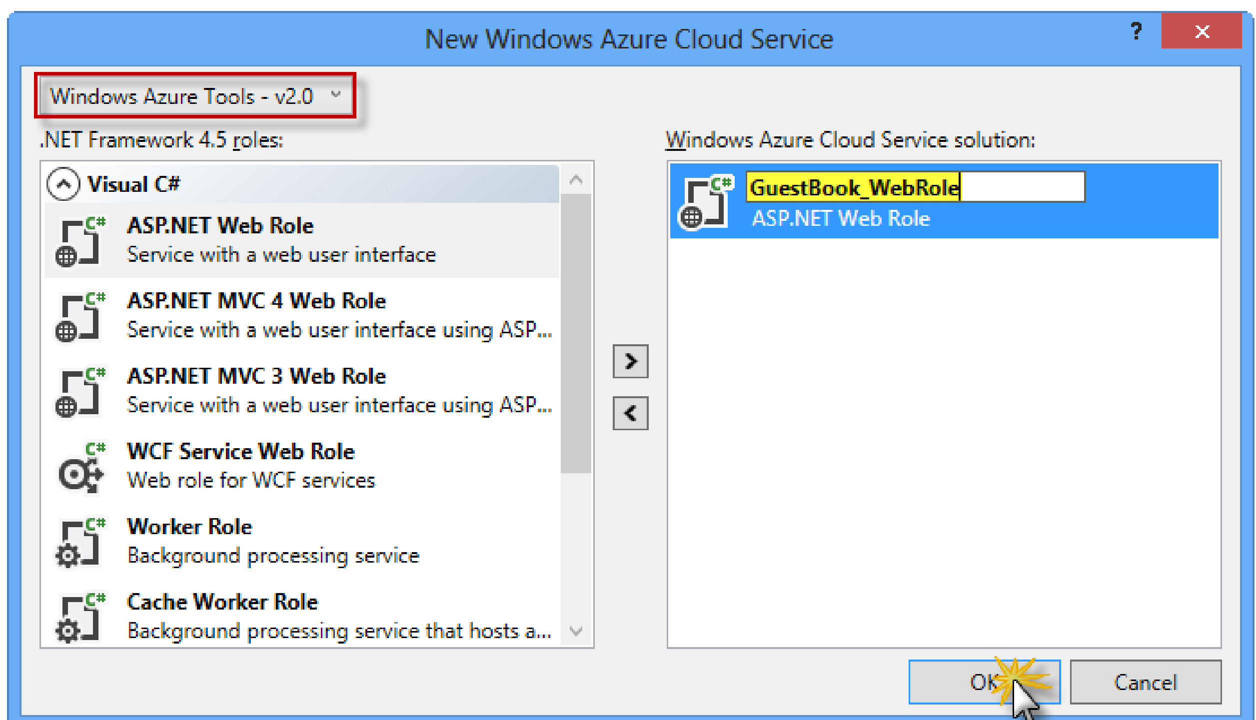


Рисунок 6.2. Добавление ролей в проект Windows Azure

6. В Solution Explorer отображается структура созданного решения.

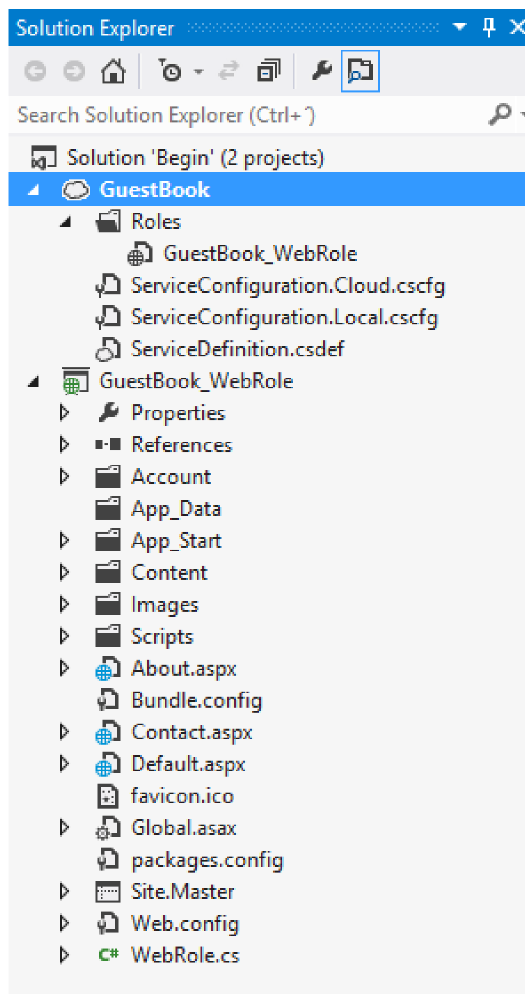


Рисунок 6.3. Проводник решения, показывающий структуру решения гостевой книги

Примечание: Созданное решение содержит два проекта. Первый проект, называемый GuestBook, содержит конфигурацию ролей, составляющих облачное приложение. Он включает файл определения службы, ServiceDefinition.csdef, который содержит метаданные, используемые фабрикой Windows Azure: список ролей и их уровень доверия, конечные точки, опубликованные ролями, используемое локальное хранилище и сертификаты. В этом же файле определяется список настроек, специфичных для конкретного приложения. Файл конфигурации службы, ServiceConfiguration.cscfg, определяет количество экземпляров каждого типа роли, значения параметров, перечисленных в файле определения службы. Разделение настроек между файлами определения и конфигурации позволяет обновлять настройки, включая изменение количества экземпляров ролей, без перезапуска приложения.

Узел ролей в облачном проекте позволяет настроить список ролей (веб-, прикладных или их комбинацию) и связанных с ними проектов. Добавление и настройка ролей на этом узле обновляет содержимое файлов ServiceDefinition.csdef и ServiceConfiguration.cscfg.

Второй проект под названием GuestBook_WebRole представляет собой традиционное приложение ASP.NET, модифицированное для работы в среде Windows Azure. Он содержит дополнительный класс, который обеспечивает точку входа для приложения и методы, управляющие запуском, стартом и остановкой роли.

Задание 2 - Создание модели данных для работы с табличным хранилищем

Приложение хранит записи гостевой книги в табличном хранилище Windows Azure. Табличное хранилище предоставляет частично структурированные таблицы, содержащие наборы сущностей. Каждая сущность содержит первичный ключ и набор свойств, набранных в формате "свойство-значение".

В дополнение к свойствам, используемым вашей моделью, каждая сущность в магазине таблиц должна иметь два ключевых свойства: **PartitionKey** и **RowKey**. Вместе эти два свойства образуют первичный ключ и уникально идентифицируют каждую сущность в таблице. Кроме того, каждая сущность имеет системное свойство **Timestamp**, которое используется службой для хранения даты последнего изменения. Это поле предназначено для использования службой и не может быть изменено в приложении. **TableService** Client API предоставляет класс **TableServiceEntity**, в котором необходимые свойства уже определены. Хотя вы можете унаследовать от него свой собственный класс, это не обязательно.

API службы таблиц поддерживает REST API, предоставляемый технологией WCF Data Services (ранее ADO. NET Data Services Framework), что позволяет использовать клиентскую библиотеку WCF Data Services (ранее . NET Client Library) и работать с хранилищем таблиц с помощью . NET.

Табличное хранилище не содержит никакой информации о схеме таблиц, что позволяет хранить в одной таблице сущности с разными наборами свойств. В то же время, такой подход не рекомендуется, и в демонстрационном приложении гостевой книги таблицы хранят данные с фиксированной схемой.

Для работы с данными в хранилище с помощью клиентской библиотеки служб данных WCF необходимо определить класс, наследующий от **TableServiceContext**, который, в свою очередь, является потомком класса **DataServiceContext, описанного** в разделе "Службы данных WCF". API хранения таблиц позволяет приложениям создавать таблицы на основе описанных контекстных классов. Чтобы это было возможно, каждый контекстный класс должен публиковать таблицы как свойства типа **IQueryable<** ***SchemaClass*****>** , где ***SchemaClass*** - это конкретный класс, описывающий сущность, хранящуюся в таблице.

В этом задании вы создадите схему хранения сущностей, которая потребуется для приложения "Гостевая книга". Затем вы создадите классы, которые позволят вам использовать службы данных WCF для доступа к табличному хранилищу. Чтобы выполнить задание, вы создадите объект,

который будет использоваться ASP. NET и обеспечит функциональность для чтения, обновления и удаления записей.

1. Создайте проект для размещения классов схем. Для этого в меню **Файл** выберите пункт **Добавить**, а затем **Новый проект**.

2. В диалоговом окне **Add New Project** на панели **Installed Templates** разверните узел, соответствующий предпочитаемому языку, выберите категорию **Windows**, укажите шаблон проекта **Class Library**. Переименуйте его в **GuestBook_Data**, не меняя предложенного расположения, и нажмите **OK.**

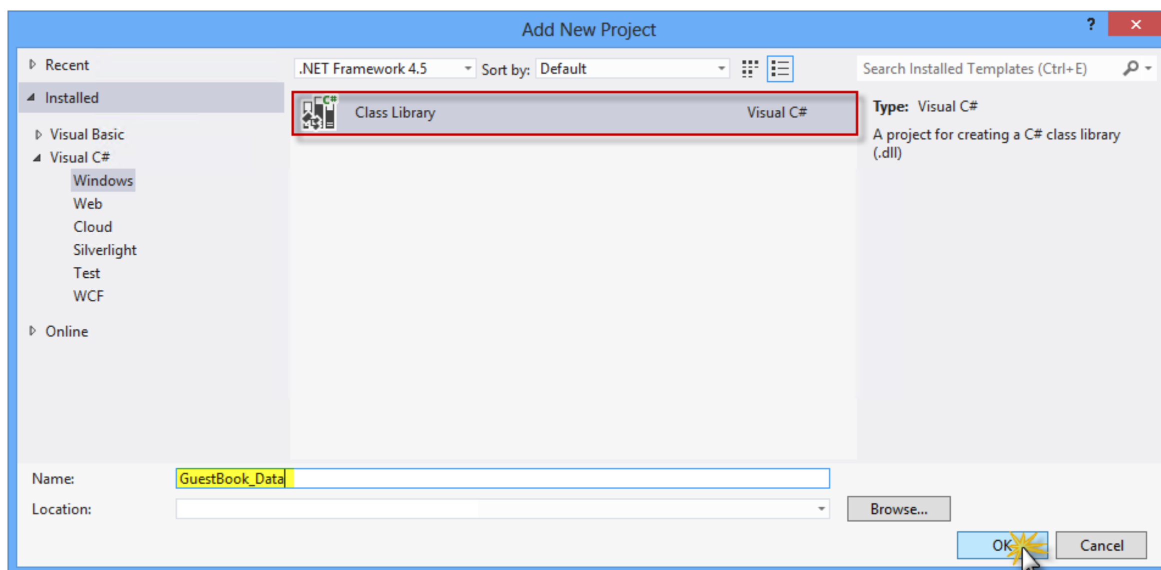


Рисунок 6.4. Создание проекта для хранения классов сущностей

3. Удалите созданный класс. Для этого щелкните правой кнопкой мыши файл **Class1.cs** и выберите **Delete**. Нажмите **OK** в диалоговом окне подтверждения.

4. Добавить в проект **GuestBook_Data** ссылку на . NET клиентская библиотека для служб данных WCF. Щелкните правой кнопкой мыши узел проекта **GuestBook_Data** в **Solution Explorer**, выберите **Add Reference**, перейдите к **. Данные. Услуги. Вкладка Клиент** и нажмите **OK**.

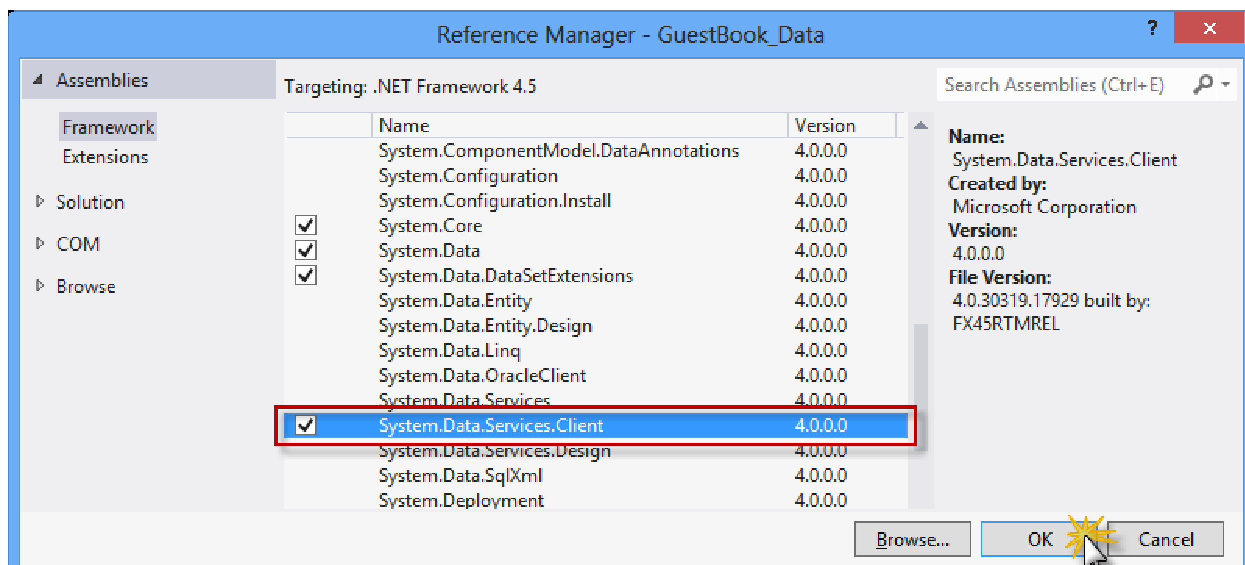


Рисунок 6.5. Добавление подключения к компоненту System.Data.Service.Client

5. Повторите предыдущий шаг и добавьте ссылку на сборку Windows Azure Storage Client API, выбрав компонент **Microsoft.WindowsAzure.Storage** и **Microsoft.WindowsAzure.Configuration**.

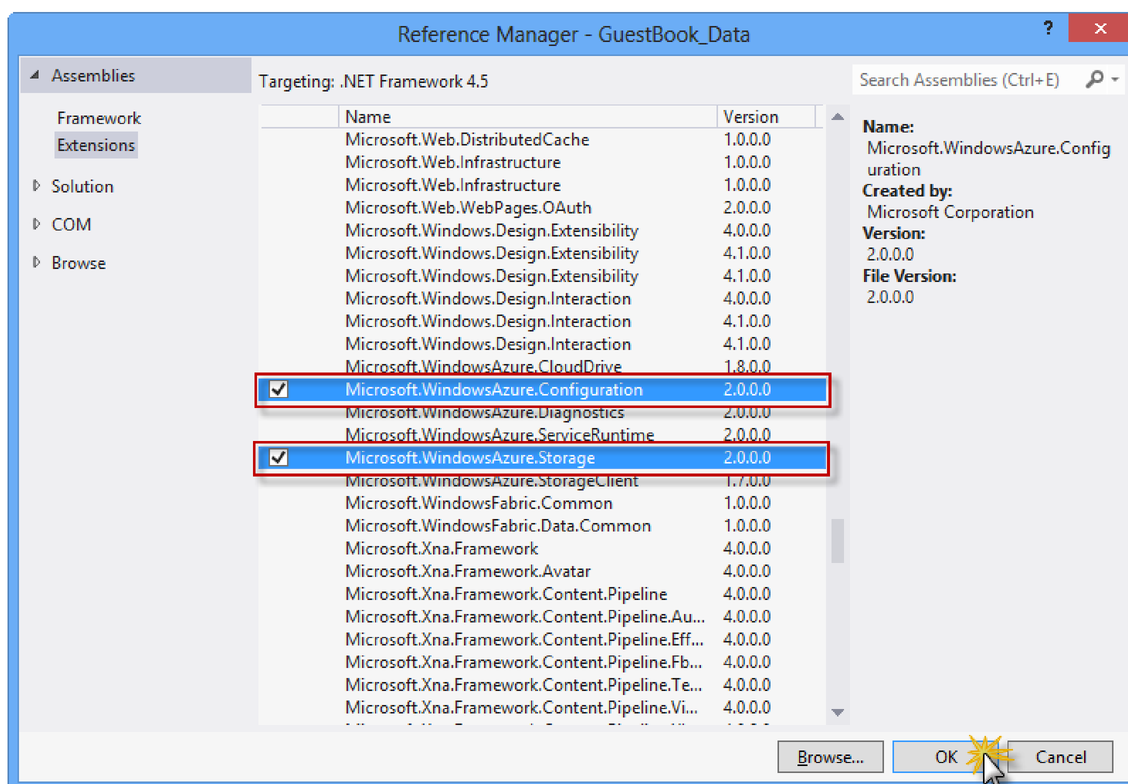


Рисунок 6.6. Добавление подключения к компонентам *Microsoft.WindowsAzure.Storage* и *Microsoft.WindowsAzure.Configuration.*

6. Прежде чем сохранить сущность в таблице, необходимо описать схему для этой сущности. Для этого щелкните правой кнопкой мыши проект **GuestBook_Data** в панели **Solution Explorer**, выберите **Add** и затем **Class**. В диалоговом окне **Add New Item** укажите имя - **GuestBookEntry.cs** и нажмите кнопку **Add**.

7. В верхней части файла свяжите пространство имен

```
using Microsoft.WindowsAzure.Storage.Table;
```

8. Откройте **файл GuestBookEntry. cs** и обновите объявление класса **GuestBookEntry**, сделав его открытым и унаследованным от класса **TableEntity**.

```
public class GuestBookEntry : TableEntity
{
}
```

Примечание: Класс TableEntity объявлен в библиотеке API клиента хранилища. Он содержит системные свойства PartitionKey, RowKey и TimeStamp, необходимые для каждой сущности.

Свойства PartitionKey и RowKey вместе определяют DataServiceKey, который уникально идентифицирует каждую сущность в таблице.

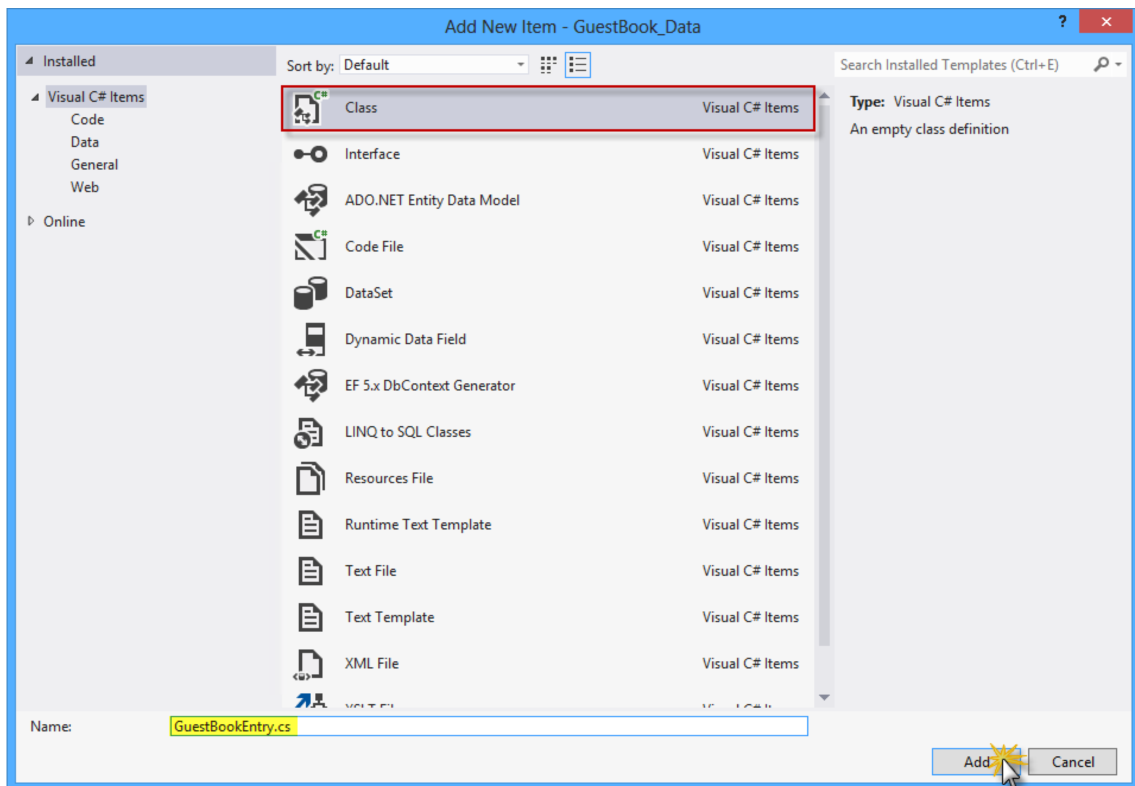


Рисунок 6.7. Добавление класса GuestBookEntry

9. Добавьте конструктор в класс **GuestBookEntry**, который инициализирует свойства **PartitionKey** и **RowKey**.

```
public GuestBookEntry()
{
  PartitionKey = DateTime.UtcNow.ToString("MMddyyyy");

  // Row key allows sorting, so we make sure the rows come back in time order
  RowKey = string.Format("{0:10}_{1}", DateTime.MaxValue.Ticks - 
DateTime.Now.Ticks, Guid.NewGuid());
}
```

Примечание: В приложении для разделения гостевой книги дата используется в качестве значения свойства PartitionKey, в результате чего создается отдельный раздел для сущностей, созданных в каждый день. Вы должны выбрать значение PartitionKey таким образом, чтобы оно позволило вам воспользоваться преимуществами балансировки нагрузки.

Свойству RowKey присваивается инвертированное значение даты и времени с добавлением идентификатора (GUID), чтобы сделать его уникальным. Внутри разделов данные сортируются по значению свойства RowKey, поэтому данный метод генерации немедленно отсортирует записи в нужном порядке - новые записи будут располагаться выше старых.

10. Чтобы завершить описание класса **GuestBookEntry**, добавьте **Message, GuestName**, **PhotoUrl** и **ThumbnailUrl,** сохранив полезную нагрузку.

```
public string Message { get; set; }
public string GuestName { get; set; }
public string PhotoUrl { get; set; }
public string ThumbnailUrl { get; set; }
```

11. Сохраните файл **GuestBookEntry. cs**.

12. Далее необходимо создать класс, позволяющий взаимодействовать с табличным магазином с помощью служб данных WCF. Для этого щелкните правой кнопкой мыши проект **GuestBook_Data** в панели **Solution Explorer**, выберите **Add**, а затем **Class**. В диалоговом окне **Add New Item** задайте имя (поле **Name**) **GuestBookDataContext.cs** и нажмите кнопку **Add**.

13. В созданном классе обновите его объявление так, чтобы он был открыт и наследовался от класса **TableServiceContext**.

```
public class GuestBookDataContext
    : Microsoft.WindowsAzure.Storage.Table.DataServices.TableServiceContext
{
}
```

14. Теперь добавьте конструктор, который инициализирует базовый класс.

```
public class GuestBookDataContext
  :  Microsoft.WindowsAzure.Storage.Table.DataServices.TableServiceContext
{
    public
GuestBookDataContext(Microsoft.WindowsAzure.Storage.Table.CloudTableClient client) :
base(client)
    {
    }
}
```

Примечание: Класс TableServiceContext наследуется от DataServiceContext из библиотеки WCF Data Services, хранит данные для доступа к хранилищу таблиц, а также реализует механизм обратного вызова и запросы операций.

15. Добавьте свойство в класс **GuestBookDataContext** для таблицы сущностей типа **GuestBookEntry**. Для этого вставьте в класс следующий код (выделенный).

```
public class GuestBookDataContext
  :  Microsoft.WindowsAzure.Storage.Table.DataServices.TableServiceContext
{
    ...
    public IQueryable<GuestBookEntry> GuestBookEntry
    {
        get {
            return this.CreateQuery<GuestBookEntry>("GuestBookEntry");
        }
    }
}
```

Примечание: Для создания таблиц можно использовать метод CreateTablesFromModel класса CloudTableClient. Когда вы передаете этому методу в качестве параметра экземпляр наследуемого класса DataServiceContext (или TableServiceContext), он находит свойства, возвращая IQueryable< T>, где T определяет класс таблицы, и создает соответствующие таблицы в магазине.

16. Теперь вам понадобится объект, который можно использовать для связи данных в ASP. NET. В панели **Solution Explorer** щелкните правой кнопкой мыши проект **GuestBook_Data**, выберите **Add**, а затем **Class**. В диалоговом окне **Add New Item** измените имя на **GuestBookDataSource. cs** (для проекта Visual C#) или **GuestBookDataSource. vb** (для проекта Visual Basic) и нажмите **Add**.

17. Импортируйте пространства имен **Microsoft.WindowsAzure, Microsoft.WindowsAzure.Storage** и **Microsoft.WindowsAzure.Storage.Table во** вновь добавленный класс.

```
using Microsoft.WindowsAzure;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
```

18. Сделайте класс **GuestBookDataSource** публичным и добавьте в него свойства для класса контекста и информации об учетной записи хранилища, как показано ниже.

```
public class GuestBookDataSource
{
    private static CloudStorageAccount storageAccount;
    private GuestBookDataContext context;
}
```

19. Теперь добавьте статический конструктор, как показано в следующем фрагменте кода (выделенная часть). Код в конструкторе создает таблицы на основе описания класса **GuestBookDataContext**.

```
public class GuestBookDataSource
{
    ...
    static GuestBookDataSource()
    {
        storageAccount = 
CloudStorageAccount.Parse(CloudConfigurationManager.GetSetting("DataConnectionString"))
;

        CloudTableClient cloudTableClient = 
storageAccount.CreateCloudTableClient();
        CloudTable table = cloudTableClient.GetTableReference("GuestBookEntry");
        table.CreateIfNotExists();
    }
}
```

20. Добавляет конструктор к классу **GuestBookDataSource**, который инициализирует класс, используемый для доступа к данным.

```
public class GuestBookDataSource
{
    ...
    public GuestBookDataSource()
    {
        this.context = new
GuestBookDataContext(storageAccount.CreateCloudTableClient());
    }
}
```

21. Теперь добавьте метод, который возвращает содержимое таблицы *GuestBookEntry*.

```
public class GuestBookDataSource
{
    ...

    public IEnumerable<GuestBookEntry> GetGuestBookEntries()
    {
        CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
        CloudTable table = tableClient.GetTableReference("GuestBookEntry");

        TableQuery<GuestBookEntry> query = new
TableQuery<GuestBookEntry>().Where(TableQuery.GenerateFilterCondition("PartitionKey",
QueryComparisons.Equal, DateTime.UtcNow.ToString("MMddyyyy")));

        return table.ExecuteQuery(query);
    }
}
```

Примечание: Метод GetGuestBookEntries возвращает сегодняшние записи, для чего используется выражение LINQ с выбором по значению свойства PartitionKey. Веб-роль использует этот метод для связи данных и отображения содержимого гостевой книги.

22. Теперь добавьте метод для добавления записи в таблицу *GuestBookEntry*.

```
public class GuestBookDataSource
{
    ...
    public void AddGuestBookEntry(GuestBookEntry newItem)
    {
        TableOperation operation = TableOperation.Insert(newItem);
        CloudTable table =
context.ServiceClient.GetTableReference("GuestBookEntry");
        table.Execute(operation);
    }
}
```

23. Наконец, добавьте метод, который обновляет URL-адрес уменьшенного изображения.

```
    public class GuestBookDataSource
    {
        ...
        public void UpdateImageThumbnail(string partitionKey, string rowKey, string
thumbUrl)
        {
            CloudTable table =
context.ServiceClient.GetTableReference("GuestBookEntry");
            TableOperation retrieveOperation =
TableOperation.Retrieve<GuestBookEntry>(partitionKey, rowKey);

            TableResult retrievedResult = table.Execute(retrieveOperation);
            GuestBookEntry updateEntity = (GuestBookEntry)retrievedResult.Result;

            if (updateEntity != null)
            {
                updateEntity.ThumbnailUrl = thumbUrl;

                TableOperation replaceOperation =
TableOperation.Replace(updateEntity);
                table.Execute(replaceOperation);
            }
        }
    }
```

Примечание: Метод UpdateImageThumbnail находит сущность по ключу раздела и ключу строки, обновляет значение свойства, уведомляет контекст об обновленных данных, а затем вызывает процедуру сохранения.

24. Сохраните файл **GuestBookDataSource.cs**

Задание 3 - Создание веб-роли для отображения содержимого гостевой книги и добавления записей

В этом задании вы доработаете веб-роль, созданную в задании 1. Изменения коснутся пользовательского интерфейса, после чего вы сможете отображать содержимое гостевой книги. Вы не будете настраивать содержимое страницы вручную, а возьмете существующую страницу, найденную в каталоге **Assets** с материалами для этого упражнения. Затем вы добавите код, который будет хранить сущности в таблице, а изображения в хранилище двоичных объектов.

1. Добавьте ссылку на проект **GuestBook_Data** в webRole. В панели **Solution Explorer** щелкните правой кнопкой мыши узел проекта **GuestBook_WebRole**, выберите **Add Reference,** переключитесь на вкладку **Projects,** выберите проект **GuestBook_Data** и нажмите **OK**.
2. Страница **Default. aspx была** сгенерирована при создании веб-функции. Он будет заменен на другой, ранее подготовленный для вас. Чтобы удалить страницу, на панели **Solution Explorer** щелкните правой кнопкой мыши файл **aspx** в проекте **GuestBook_WebRole** и выберите **Delete**.
3. Добавьте ранее подготовленную страницу к webRole. Для этого щелкните правой кнопкой мыши проект **GuestBook_WebRole в Solution**

Explorer, выберите **Add | Existing Item**. В диалоговом окне Add Existing Item перейдите в каталог **Cloud Technologies,** удерживая клавишу **CTRL**, выберите все файлы и нажмите кнопку **Add.**

> **Примечание**: в каталоге Lab.5 есть пять файлов для добавления в проект - Default.aspx с кодом и файлами макета, CSS и изображение.

4. Откройте файл кода для домашней страницы проекта **GuestBook_WebRole**. Для этого щелкните правой кнопкой мыши на файле **Default. aspx** и выберите **View code**.
5. Добавьте следующие объявления пространства имен.

```
using System.IO;
using System.Net;
using Microsoft.WindowsAzure;
using Microsoft.WindowsAzure.ServiceRuntime;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Blob;
using Microsoft.WindowsAzure.Storage.Queue;
using GuestBook_Data;
```

6. Добавляет объявления свойств в класс **Default**.

```
public partial class Default : System.Web.UI.Page
{
    private static bool storageInitialized = false;
    private static object gate = new object();
    private static CloudBlobClient blobStorage;

    ...
}
```

7. Добавьте обработчик события **SignButton_Click** со следующим содержанием.

```
public partial class Default : System.Web.UI.Page
{
    ...

    protected void SignButton_Click(object sender, EventArgs e)
    {
        if (this.FileUpload1.HasFile)
        {
            this.InitializeStorage();

            // upload the image to blob storage
            string uniqueBlobName = string.Format("guestbookpics/image_{0}{1}",
Guid.NewGuid().ToString(), Path.GetExtension(this.FileUpload1.FileName));
            CloudBlockBlob blob =
blobStorage.GetContainerReference("guestbookpics").GetBlockBlobReference(uniqueBlobName
);
            blob.Properties.ContentType =
this.FileUpload1.PostedFile.ContentType;
            blob.UploadFromStream(this.FileUpload1.FileContent);
```

```
                    System.Diagnostics.Trace.TraceInformation("Uploaded image '{0}' to
blob storage as '{1}'", this.FileUpload1.FileName, uniqueBlobName);

                    // create a new entry in table storage
                    GuestBookEntry entry = new GuestBookEntry() { GuestName =
this.NameTextBox.Text, Message = this.MessageTextBox.Text, PhotoUrl =
blob.Uri.ToString(), ThumbnailUrl = blob.Uri.ToString() };
                    GuestBookDataSource ds = new GuestBookDataSource();
                    ds.AddGuestBookEntry(entry);
                    System.Diagnostics.Trace.TraceInformation("Added entry {0}-{1} in
table storage for guest '{2}'", entry.PartitionKey, entry.RowKey, entry.GuestName);
                }

                this.NameTextBox.Text = string.Empty;
                this.MessageTextBox.Text = string.Empty;

                this.DataList1.DataBind();
        }
```

Примечание: Для добавления записи обработчик сначала вызывает метод InitializeStorage, чтобы убедиться, что контейнер двоичного объекта существует и к нему можно получить доступ. Этот метод будет создан для вас в ближайшее время.

Затем метод получает ссылку на контейнер, создает уникальное имя для объекта и создает сам объект, а затем загружает в него изображение, предоставленное пользователем. Тип содержимого, свойство ContentType, устанавливается на тип загруженного файла. При чтении содержимого двоичного объекта этот тип будет использоваться для корректного отображения изображения.

В конце создается сущность типа GuestBookEntry, которую вы описали в предыдущем задании, инициализируется данными, введенными пользователем, а затем сохраняется с помощью методов класса GuestBookDataSource.

Последним шагом процедуры является связывание данных для обновления содержимого страницы.

8. Обновляет содержимое метода **Timer1_Tick в соответствии со** следующим текстом.

```
public partial class Default : System.Web.UI.Page
{
    ...
    protected void Timer1_Tick(object sender, EventArgs e)
    {
        this.DataList1.DataBind();
    }
    ...
}
```

Обратите внимание: таймер заставляет вас регулярно обновлять содержимое страницы.

9. Найдите обработчик события **Page_Load** и обновите его содержимое в соответствии с приведенным ниже примером, чтобы активировать таймер.

```
public partial class Default : System.Web.UI.Page
{
    ...

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!Page.IsPostBack)
        {
            this.Timer1.Enabled = true;
        }
    }

    ...
}
```

10. Добавьте реализацию метода **InitializeStorage в** соответствии со следующим фрагментом.

```
public partial class _Default : System.Web.UI.Page
{
    ...

    private void InitializeStorage()
    {
        if (storageInitialized)
        {
            return;
        }

        lock (gate)
        {
            if (storageInitialized)
            {
                return;
            }

            try
            {
                // read account configuration settings
                var storageAccount = 
CloudStorageAccount.Parse(CloudConfigurationManager.GetSetting("DataConnectionString"))
;

                // create blob container for images
                blobStorage = storageAccount.CreateCloudBlobClient();
                CloudBlobContainer container = 
blobStorage.GetContainerReference("guestbookpics");
                container.CreateIfNotExists();

                // configure container for public access
                var permissions = container.GetPermissions();
                permissions.PublicAccess = 
BlobContainerPublicAccessType.Container;
                container.SetPermissions(permissions);
            }
            catch (WebException)
```

```
                    {
                        throw new WebException("Storage services initialization failure.
"
                            + "Check your storage account configuration settings. If
running locally, "
                            + "ensure that the Development Storage service is
running.");
                    }

                    storageInitialized = true;
                }
            }
        }
```

Примечание: Метод InitializeStorage сначала проверяет, не был ли он запущен ранее. Затем он считывает информацию об учетной записи хранилища из файла конфигурации, создает контейнер образа и устанавливает для него емкость общего доступа.

11. Поскольку веб-роль использует службы хранения Windows Azure, необходимо хранить учетные данные для подключения к ним. Чтобы создать новую конфигурацию, в панели **Solution Explorer** разверните узел **Roles**, расположенный в проекте **GuestBook,** и дважды щелкните роль **GuestBook_WebRole**. В конфигурации роли перейдите на вкладку **Конфигурация**. Нажмите кнопку **Добавить конфигурацию**, укажите "*DataConnectionString*" в качестве имени (колонка **Name**), измените тип на *ConnectionString*, а затем нажмите кнопку с многоточием.

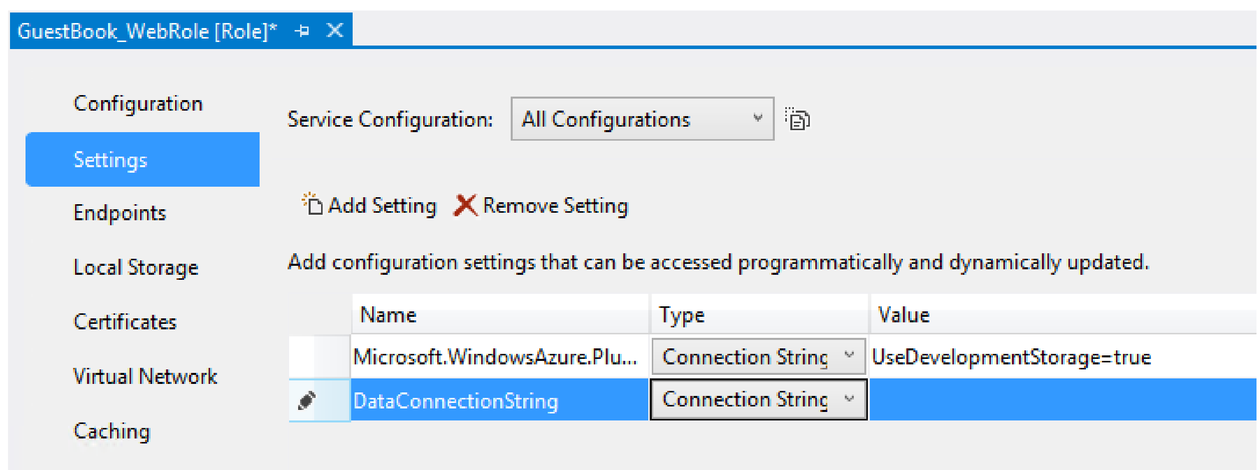


Рисунок 6.8. Добавление информации об учетной записи хранилища в файл конфигурации

12. В диалоговом окне **String Storage Account Connection** выберите **Use Windows Azure storage** emulator и нажмите **OK**.

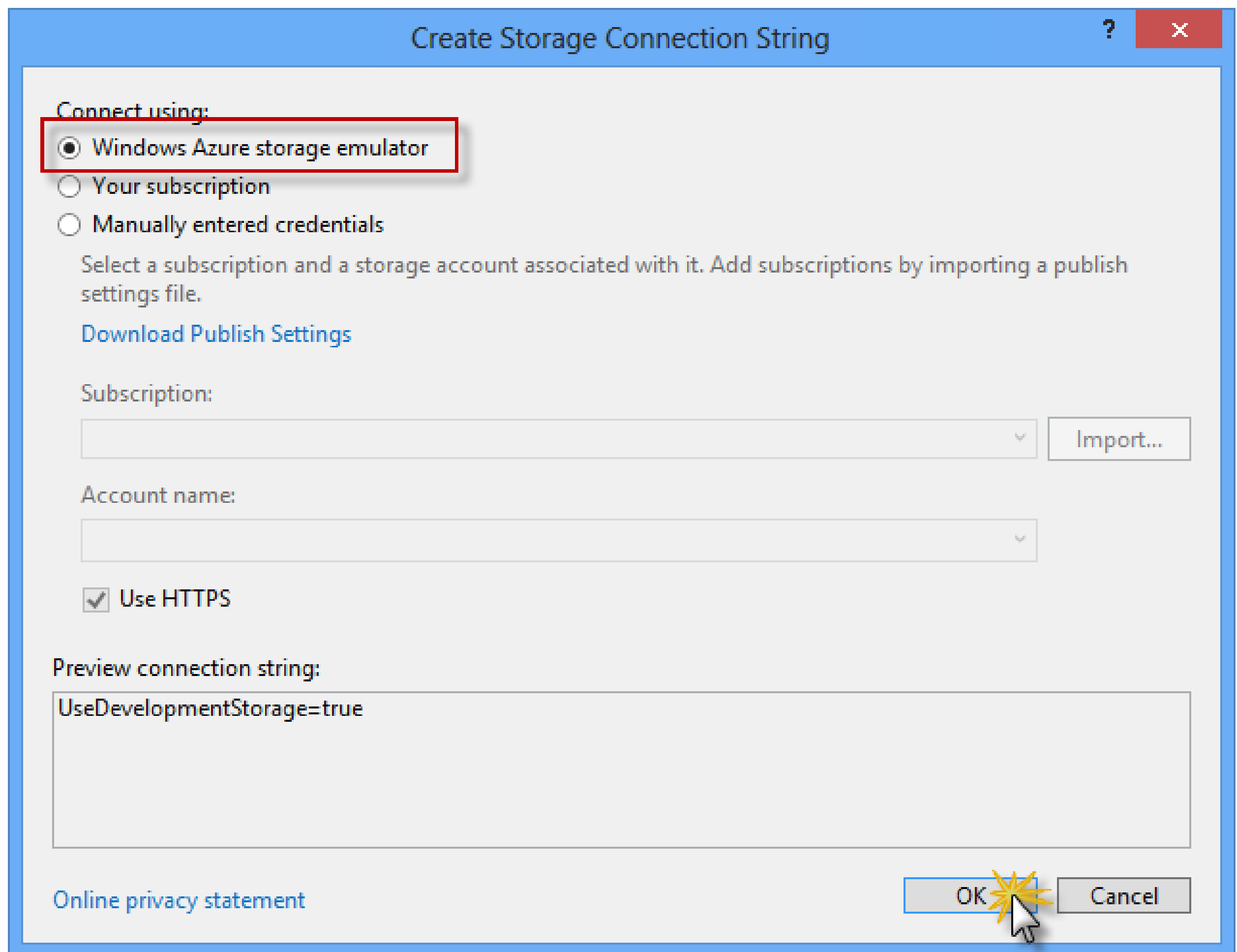


Рисунок 6.9. Создание цепной связи с эмулированным хранилищем

Примечание: Учетные записи хранилищ (табличные, бинарные объекты и объекты очередей) создаются отдельно, поэтому для их использования необходимо создать их заранее через портал администрирования. В этом упражнении вы будете использовать эмулированный репозиторий, который позволяет разрабатывать и отлаживать приложения локально.

Чтобы использовать эмулятор хранилища, в строке подключения необходимо указать UseDevelopmentStorage=true. Для подключения к реальному хранилищу необходимо указать протокол, имя хранилища и пароль:

<Configuration name="DataConnectionString" value="DefaultEndpointsProtocol=https; AccountName=YourAccountName; AccountKey=YourAccountKey" />

где YourAccountName - имя службы хранения, а YourAccountKey - ключ.

13. Нажмите комбинацию клавиш **CTRL + S,** чтобы сохранить изменения в файле конфигурации.

Задание 4 – Использование очередей для организации фоновой обработки данных

Это задание обновит веб-роль, чтобы поставить рабочие элементы в очередь для обработки в фоновом режиме. Эти элементы будут оставаться в

очереди до тех пор, пока их не извлечет прикладная функция. Прикладная функция извлекает следующий элемент в очереди и создает миниатюру для каждого изображения, добавленного пользователем.

1. Откройте файл кода из файла **по умолчанию.aspx**. Для этого щелкните по нему правой кнопкой мыши и выберите **View code**.

2. Добавляет экземпляр класса **CloudQueueClient,** используемого для взаимодействия с очередью (выделенный фрагмент), в класс **Default.**

```
public partial class _Default : System.Web.UI.Page
{
    private static bool storageInitialized = false;
    private static object gate = new object();
    private static CloudBlobClient blobStorage;
    private static CloudQueueClient queueStorage;

    ...
}
```

3. Теперь обновите код, инициализирующий хранилище, так, чтобы он дополнительно создавал очередь, если она не существует, и инициализировал переменную, добавленную ранее. Для этого найдите метод **InitializeStorage** и добавьте следующий код (выделенный фрагмент) сразу после кода, который устанавливает контейнер бинарного объекта.

```
public partial class _Default : System.Web.UI.Page
{
    ...

    private void InitializeStorage()
    {
        ...

        try
        {
            ...

            // create queue to communicate with worker role
            queueStorage = storageAccount.CreateCloudQueueClient();
            CloudQueue queue = queueStorage.GetQueueReference("guestthumbs");
            queue.CreateIfNotExists();
        }
        catch (WebException)
        {
            ...
        }

        ...
    }
}
```

Примечание: Обновленный код создает очередь, используемую веб-функцией для передачи задач функции приложения.

4. Наконец, добавьте код, который помещает рабочие элементы в очередь. Найдите обработчик **SignButton_Click** и вставьте приведенный ниже код (выделенный фрагмент) после того, как новая сущность будет добавлена в таблицу shop.

```
protected void SignButton_Click(object sender, EventArgs e)
{
    if (this.FileUpload1.HasFile)
    {
        ...

        // queue a message to process the image
        var queue = queueStorage.GetQueueReference("guestthumbs");
        var message = new CloudQueueMessage(string.Format("{0},{1},{2}",
uniqueBlobName, entry.PartitionKey, entry.RowKey));
        queue.AddMessage(message);
        System.Diagnostics.Trace.TraceInformation("Queued message to process blob
'{0}'", uniqueBlobName);
    }

    this.NameTextBox.Text = string.Empty;
    this.MessageTextBox.Text = string.Empty;

    this.DataList1.DataBind();
}
```

Внимание: обновленный код получает ссылку на очередь под названием "guestthumbs". Он создает новое сообщение, состоящее из строки, разделенной запятыми, содержащей имя двоичного объекта с изображением, ключ раздела и ключ строки для записи в магазин таблиц. Затем метод добавляет полученное сообщение в очередь.

Проверка

Эмулятор Windows Azure (ранее известный как Development Fabric или devfabric) позволяет разрабатывать и тестировать приложения локально. В этом задании вы запустите приложение "Гостевая книга" локально, а затем добавите несколько записей в гостевую книгу.

Другие возможности, предлагаемые инструментарием Windows Azure для Microsoft Visual Studio, включают браузер Windows Azure Storage, который позволяет подключиться к учетной записи хранилища и просмотреть содержащиеся в ней таблицы и бинарные объекты.

1. Нажмите **F5** для запуска. Локальный эмулятор Windows Azure запустится после завершения сборки. Чтобы отобразить доступный пользовательский интерфейс, щелкните правой кнопкой мыши значок Windows Azure на панели задач и выберите **Показать пользовательский интерфейс эмулятора компьютера.**

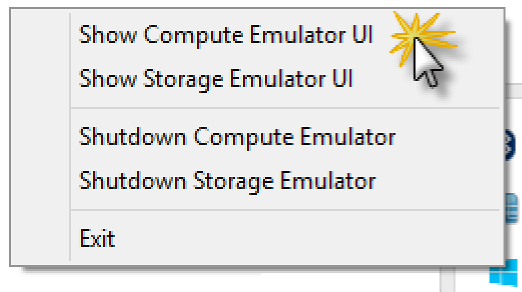


Рисунок 6.10. Экран интерфейса эмулятора

Примечание: Первый запуск эмулятора займет значительно больше времени, чем все последующие, поскольку он инициализирует необходимые базы данных и таблицы. Для просмотра хода выполнения этой процедуры можно использовать диалоговое окно Development Storage Initialisation.

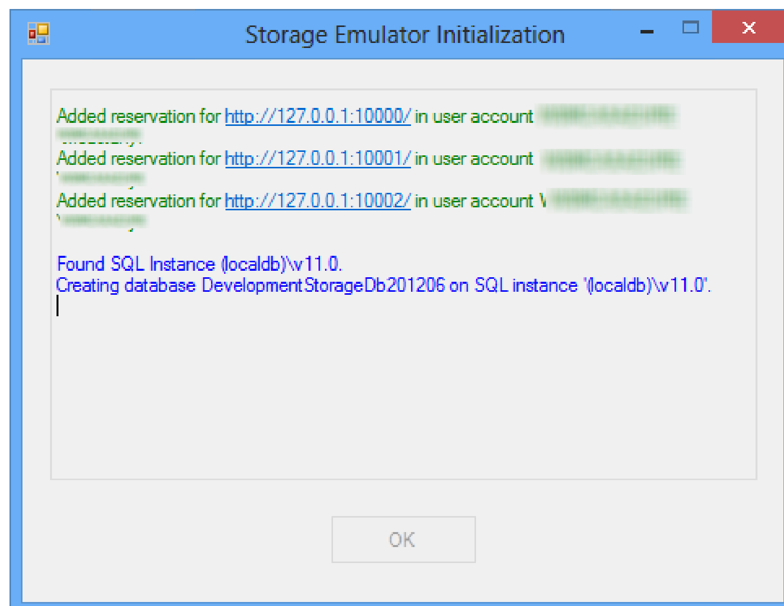


Рисунок 6.11. Процесс загрузки эмулятора хранилища

2. Переключитесь на Internet Explorer для просмотра приложения гостевой книги.
3. Добавьте новую запись в гостевую книгу. Для этого введите свое имя и текст сообщения, выберите фотографию и нажмите кнопку с карандашом.

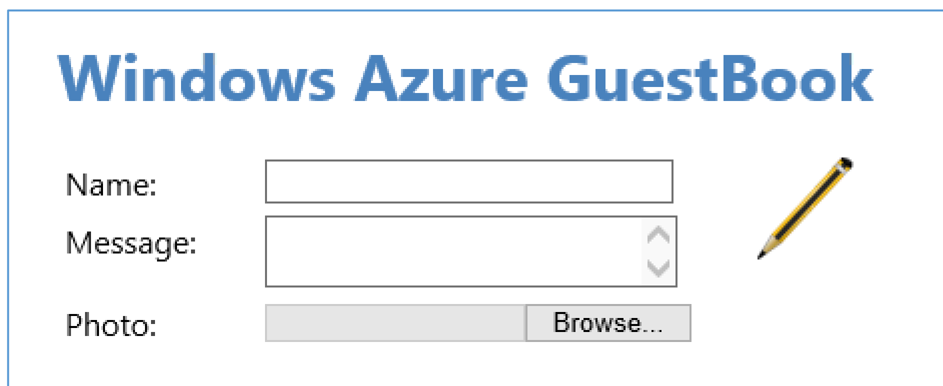


Рисунок 6.12. Домашняя страница гостевой книги проекта

Примечание: загрузка изображений высокого разрешения позволит вам проверить готовое изображение, так как функция приложения создаст для них миниатюры.

После добавления записи веб-роль создает объект в табличном магазине и загружает изображение в магазин двоичных объектов. Таймер страницы обновляется каждые 5 секунд, поэтому новая запись будет отображаться в течение этого периода времени. Пока миниатюра не создана, на странице будет отображаться исходное изображение.

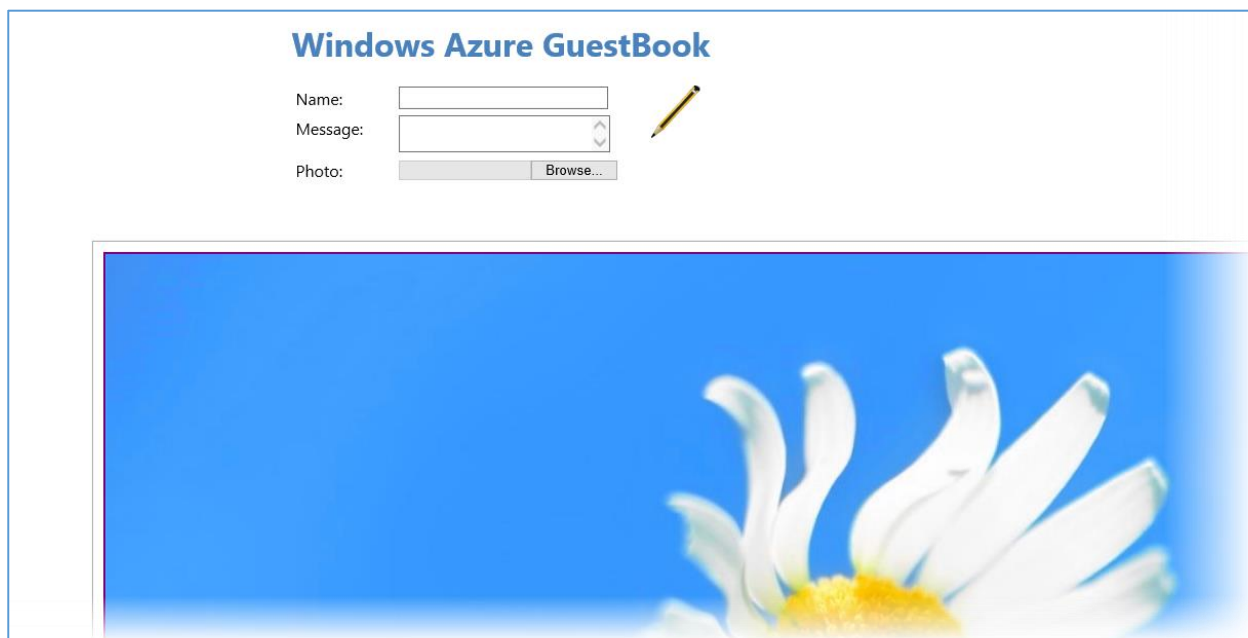


Рисунок 6.13. Приложение для гостевой книги, показывающее добавленное изображение в исходном размере

4. Чтобы открыть Storage Explorer в Visual Studio 2015, используйте меню **View | Server Explorer**, затем разверните узел **Windows Azure Storage**.

5. Узел **хранилища Windows Azure** показывает список всех зарегистрированных учетных записей, включая локальное хранилище, помеченное как **(Разработка).**

6. Разверните узел **(Разработка)**, затем **Таблицы**. Убедитесь, что существует таблица *GuestBookEntry,* созданная приложением.

GuestBookEntry [Table]

Enter a WCF Data Services filter to limit the entities returned

PartitionKey	RowKey	Timestamp	Message	GuestName	PhotoUrl	ThumbnailUrl
05222013	12520332283576...	22/05/2013 07:2...	Image	Image	http://127.0.0.1:...	http://127.0.0.1:...

Рисунок 6.14. Просмотр таблиц в локальном хранилище Windows Azure

7. Дважды щелкните таблицу *GuestBookEntry*, чтобы отобразить ее содержимое. Видно, что он содержит значения как для введенных пользователем свойств (*GuestName*, *Message*, *PhotoUrl* и *ThumbnailUrl*), так и для системных свойств (*PartitionKey*, *RowKey* и *Timestamp*). Свойства *PhotoUrl* и *ThumbnailUrl* в

настоящее время ссылаются на один и тот же объект. В следующем упражнении вы создадите функцию приложения, которая генерирует эскизы и обновляет записи с соответствующим URL.

GuestBookEntry [Table]

Enter a WCF Data Services filter to limit the entities returned

PartitionKey	RowKey	Timestamp	GuestName	Message	PhotoUrl	ThumbnailUrl
09182012	12520543427620...	9/18/2012 3:47:...	Image	Image	http://127.0.0.1:...	http://127.0.0.1:...

Рисунок 6.15. Просмотр содержимого таблицы с помощью Windows Azure Tools for Visual Studio

8. Теперь разверните узел **Blobs**. Внутри находятся контейнеры - в нашем случае это один контейнер для *фотографий гостевой книги*, который содержит наши изображения.

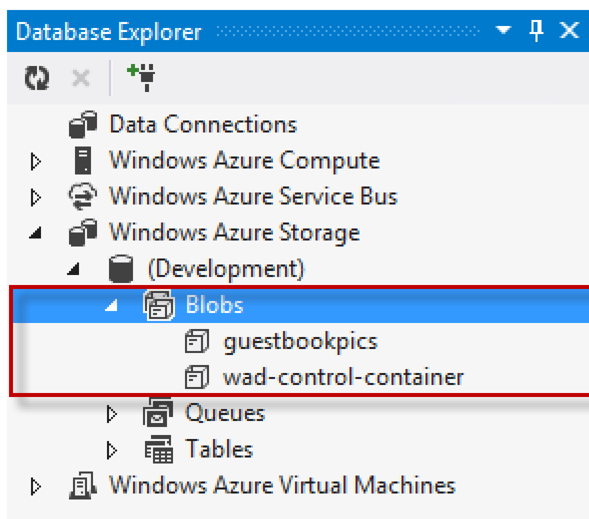


Рисунок 6.16. Просмотр двоичного хранилища с помощью инструментов Windows Azure для Visual Studio

9. Дважды щелкните на контейнере *гостевой книги*. Результат содержит запись для каждого из изображений, добавленных выше.

guestbookpics [Container]

Filter by prefix (case-sensitive)

Name	Size	Last Modified (UTC)	Content Typ...	URL
guestbookpics/image_d3469da5-98c7-420d-8781-3e8bc8873baa.jpg	31,6...	22/05/2013 08:54:02 p....	image/jpeg	h...

Рисунок 6.17. Просмотр содержимого двоичного контейнера

10. Для каждой сущности в хранилище двоичных объектов отдельное поле содержит ее тип содержимого, что позволяет Visual Studio выбрать соответствующий инструмент для ее просмотра. Чтобы просмотреть содержимое сущности, дважды щелкните любую сущность.

11. Нажмите **SHIFT + F5,** чтобы остановить процесс отладки.

Библиографический список

1. Общие сведения о хранилище BLOB-объектов Azure. URL: https://docs.microsoft.com/ru-ru/azure/storage/blobs/storage-blobs-introduction (Дата обращения 17.12.2021 г.)

2. Обзор хранилища BLOB-объектов Azure. URL: https://docs.microsoft.com/ru-ru/azure/storage/blobs/storage-blobs-overview (Дата обращения 17.12.2021 г.)

Практическая работа 7. Работа с Windows Azure Queue

В этой практической работе в части облачных технологий вы создадите роль приложения, которая будет получать сообщения из очереди, в которую их помещает веб-роль. Роль приложения извлекает изображение из хранилища двоичных объектов, создает для него миниатюру, которую затем также сохраняет как двоичный объект. В части мобильных технологий будет построено приложение, которое называется DroidQuest. Оно представляет собой викторину на тему: «хорошо ли пользователь знает Android». Пользователь отвечает на вопрос, нажимая кнопку «Да» или «Нет», а DroidQuest мгновенно сообщает ему результат.

Облачные технологии

Роль приложения выполняется в фоновом режиме – его можно сравнить со службой Windows.

Задание 1 – Создание прикладной функции для фоновой обработки данных

В этом задании вы добавите роль приложения в решение и улучшите ее, чтобы она могла получать сообщения из очереди и обрабатывать их.

1. Запустите Visual Studio от имени администратора, если вы не делали этого раньше.
2. В меню **Файл** выберите Открыть | **Проект/Решение**. В диалоговом окне **Open Project** перейдите в подкаталог с решением из предыдущей практической работы и продолжите.
3. В **Solution Explorer** щелкните правой кнопкой мыши на узле **Roles** проекта **Guestbook** и выберите **Add | New Worker Role Project**.
4. В диалоговом окне **Add New Project Role** выберите категорию **Worker Role** и шаблон **Worker Role**. Переименуйте добавленную роль в **GuestBook_WorkerRole** и нажмите **Add** (рисунок 7.1).
5. В новом проекте добавьте ссылку на модель данных. Для этого щелкните правой кнопкой мыши проект **GuestBook_WorkerRole** в панели **Solution Explorer,** выберите **Add Reference,** переключитесь на вкладку **Projects**, выберите **GuestBook_Data** и нажмите **OK**.
6. Теперь добавьте ссылку на **System.** В диалоговом окне **Добавить ссылку** переключитесь на **.NET** выберите **System. Компонент дизайна** и нажмите **OK**.
7. Откройте файл **WorkerRole. cs** проекта **GuestBook_WorkerRole** и добавьте следующие объявления пространства имен.

```
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.IO;
using GuestBook_Data;
using Microsoft.WindowsAzure.Storage.Queue;
using Microsoft.WindowsAzure.Storage.Blob;
```

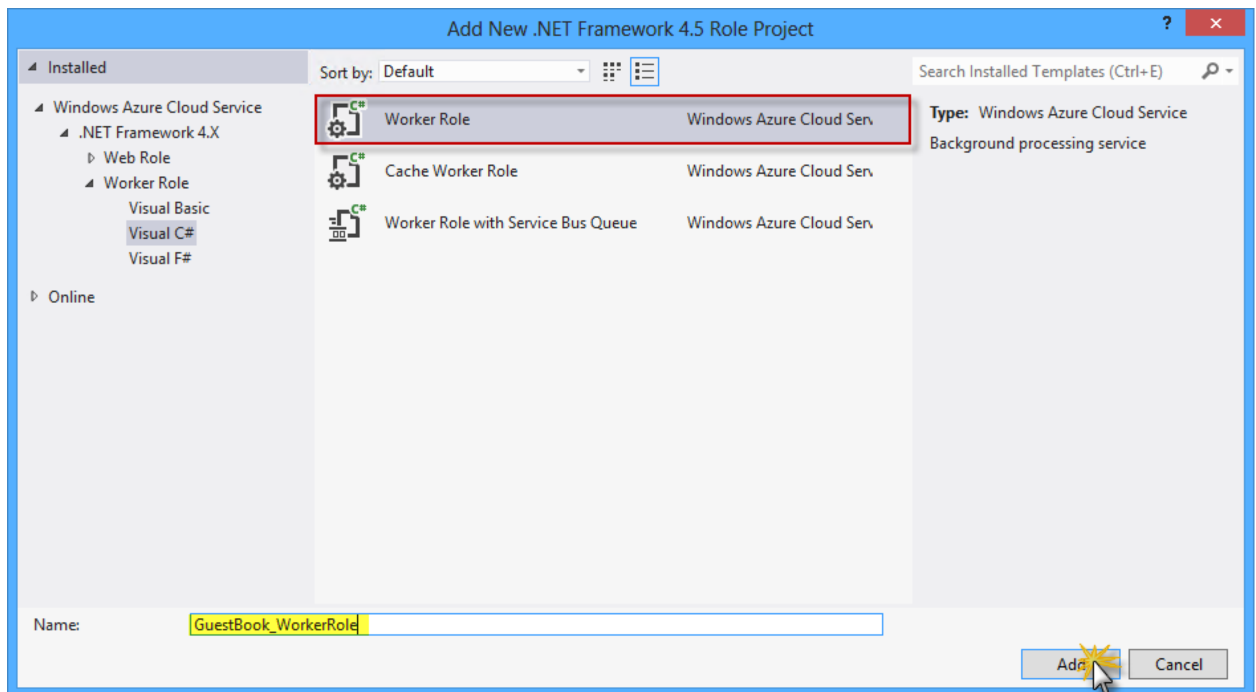


Рисунок 7.1. Добавление прикладного документа к решению

8. Добавляет экземпляры классов бинарного хранилища и очереди к классу **WorkerRole.**

```
public class WorkerRole : RoleEntryPoint
{
    private CloudQueue queue;
    private CloudBlobContainer container;

    ...
}
```

9. Добавьте следующий фрагмент в метод **OnStart** перед вызовом метода **OnStart** базового класса.

```
public class WorkerRole : RoleEntryPoint
{
  ...

  public override bool OnStart()
  {
    // Set the maximum number of concurrent connections
    ServicePointManager.DefaultConnectionLimit = 12;

    // read storage account configuration settings
    var storageAccount =
CloudStorageAccount.Parse(CloudConfigurationManager.GetSetting("DataConnectionString"))
;

    // initialize blob storage
    CloudBlobClient blobStorage = storageAccount.CreateCloudBlobClient();
    this.container = blobStorage.GetContainerReference("guestbookpics");

    // initialize queue storage
```

```
        CloudQueueClient queueStorage = storageAccount.CreateCloudQueueClient();
        this.queue = queueStorage.GetQueueReference("guestthumbs");

        Trace.TraceInformation("Creating container and queue...");

        bool storageInitialized = false;
        while (!storageInitialized)
        {
            try
            {
                // create the blob container and allow public access
                this.container.CreateIfNotExists();
                var permissions = this.container.GetPermissions();
                permissions.PublicAccess = BlobContainerPublicAccessType.Container;
                this.container.SetPermissions(permissions);

                // create the message queue(s)
                this.queue.CreateIfNotExists();

                storageInitialized = true;
            }
            catch (StorageException e)
            {
                var requestInformation = e.RequestInformation;
                var errorCode =
requestInformation.ExtendedErrorInformation.ErrorCode;//errorCode =
ContainerAlreadyExists
                var statusCode =
(System.Net.HttpStatusCode)requestInformation.HttpStatusCode;//requestInformation.HttpS
tatusCode = 409, statusCode = Conflict
                if (statusCode == HttpStatusCode.NotFound)
                {
                    Trace.TraceError(
                      "Storage services initialization failure. "
                      + "Check your storage account configuration settings. If
running locally, "
                      + "ensure that the Development Storage service is running.
Message: '{0}'",
                      e.Message);
                    System.Threading.Thread.Sleep(5000);
                }
                else
                {
                    throw;
                }
            }
        }

        return base.OnStart();
      }
    }
```

10. Замените тело метода **Run** следующим фрагментом.

```
    public class WorkerRole : RoleEntryPoint
    {
      ...

      public override void Run()
      {
        Trace.TraceInformation("Listening for queue messages...");
```

```
            while (true)
            {
                try
                {
                    // retrieve a new message from the queue
                    CloudQueueMessage msg = this.queue.GetMessage();
                    if (msg != null)
                    {
                        // parse message retrieved from queue
                        var messageParts = msg.AsString.Split(new char[] { ',' });
                        var imageBlobName = messageParts[0];
                        var partitionKey = messageParts[1];
                        var rowkey = messageParts[2];
                        Trace.TraceInformation("Processing image in blob '{0}'.",
imageBlobName);

                        string thumbnailName =
System.Text.RegularExpressions.Regex.Replace(imageBlobName, "([^\\.]+)(\\.[^\\.]+)?$",
"$1-thumb$2");

                        CloudBlockBlob inputBlob =
this.container.GetBlockBlobReference(imageBlobName);
                        CloudBlockBlob outputBlob =
this.container.GetBlockBlobReference(thumbnailName);

                        using (Stream input = inputBlob.OpenRead())
                        using (Stream output = outputBlob.OpenWrite())
                        {
                            this.ProcessImage(input, output);

                            // commit the blob and set its properties
                            outputBlob.Properties.ContentType = "image/jpeg";
                            string thumbnailBlobUri = outputBlob.Uri.ToString();

                            // update the entry in table storage to point to the
thumbnail
                            GuestBookDataSource ds = new GuestBookDataSource();
                            ds.UpdateImageThumbnail(partitionKey, rowkey,
thumbnailBlobUri);

                            // remove message from queue
                            this.queue.DeleteMessage(msg);

                            Trace.TraceInformation("Generated thumbnail in blob
'{0}'.", thumbnailBlobUri);
                        }
                    }
                    else
                    {
                        System.Threading.Thread.Sleep(1000);
                    }
                }
                catch (StorageException e)
                {
                    Trace.TraceError("Exception when processing queue item. Message:
'{0}'", e.Message);
                    System.Threading.Thread.Sleep(5000);
                }
            }
        }
```

```
  ...
}
```

11. Наконец, добавьте метод в класс **WorkerRole,** который создает миниатюру для указанного изображения.

```
public class WorkerRole : RoleEntryPoint
{
  ...

  public void ProcessImage(Stream input, Stream output)
  {
    int width;
    int height;
    var originalImage = new Bitmap(input);

    if (originalImage.Width > originalImage.Height)
    {
      width = 128;
      height = 128 * originalImage.Height / originalImage.Width;
    }
    else
    {
      height = 128;
      width = 128 * originalImage.Width / originalImage.Height;
    }

    Bitmap thumbnailImage = null;

    try
    {
      thumbnailImage = new Bitmap(width, height);

      using (Graphics graphics = Graphics.FromImage(thumbnailImage))
      {
        graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
        graphics.SmoothingMode = SmoothingMode.AntiAlias;
        graphics.PixelOffsetMode = PixelOffsetMode.HighQuality;
        graphics.DrawImage(originalImage, 0, 0, width, height);
      }

      thumbnailImage.Save(output, ImageFormat.Jpeg);
    }
    finally
    {
      if (thumbnailImage != null)
      {
          thumbnailImage.Dispose();
      }
    }
  }
}
```

Примечание: В приведенном выше примере для простоты используются типы из сборки System.Drawing. Эта сборка предназначена для использования в приложениях Windows Forms и не совместима с другими типами приложений. Если вы решите использовать эту сборку в производственном решении Windows Azure, пожалуйста, тщательно протестируйте эту функциональность.

12. Роль приложения также использует службы хранения Windows Azure, поэтому вам нужно будет добавить соответствующие параметры в файл конфигурации, как и для веб-роли. Чтобы добавить конфигурацию, разверните узел **Roles** проекта **GuestBook**, дважды щелкните **GuestBook_WorkerRole** и перейдите на вкладку **Settings.** Нажмите кнопку **Добавить конфигурацию**, назовите конфигурацию "*DataConnectionString*", измените **тип** на *ConnectionString* и нажмите кнопку с многоточием. В диалоговом окне **Строка подключения учетной записи хранилища** выберите **Использовать эмулятор хранилища Windows Azure** и нажмите **OK**. Нажмите **CTRL + S,** чтобы сохранить изменения.

Проверка

Теперь запустите обновленное приложение локально, чтобы протестировать новую функциональность. Проверьте, что функция приложения генерирует эскизы для представленных изображений.

1. Нажмите **F5,** чтобы запустить приложение локально.
2. Переключитесь на Internet Explorer. Если вы запускали приложение ранее, вы увидите предыдущие записи, включая изображения в оригинальном размере. Одновременно с добавлением сущностей в хранилище таблиц в очередь добавлялись сообщения, которые также никуда не делись.
3. Подождите некоторое время, пока прикладная функция обработает поставленные в очередь сообщения и создаст эскизы. Когда обработка изображения будет завершена, и страница обновится, на ее месте появится миниатюра исходного изображения.

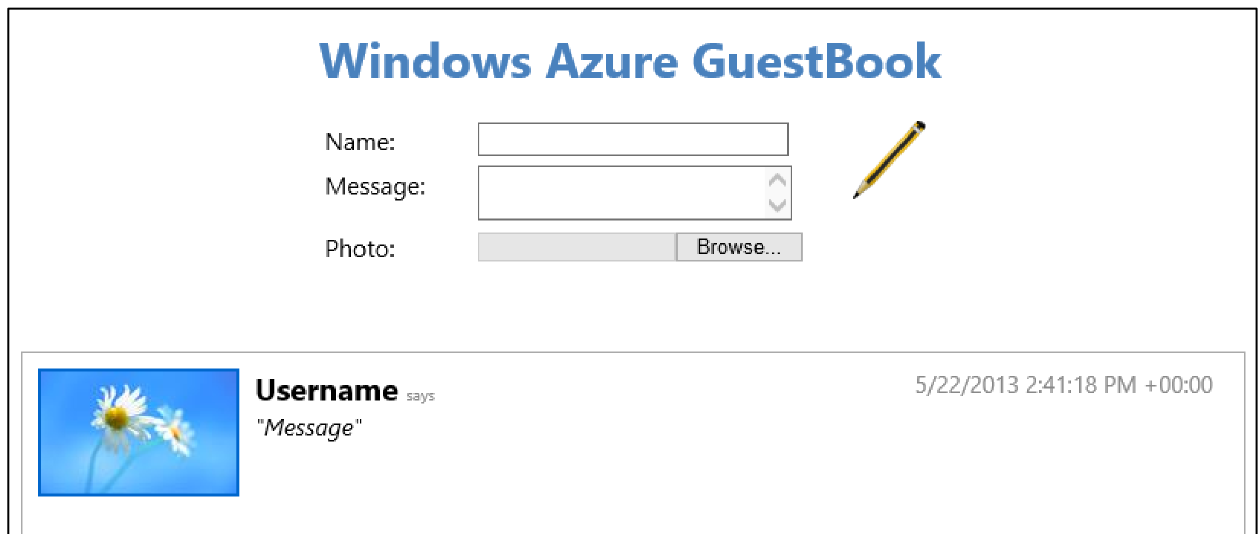


Рисунок 7.2. Страница, показывающая эскиз, созданный бумагой приложения

4. Разверните узел **хранилища Windows Azure** в панели **Server Explorer** и дважды щелкните контейнер *фотографий гостевой книги*. Теперь контейнер содержит дополнительные объекты, соответствующие созданным эскизам.

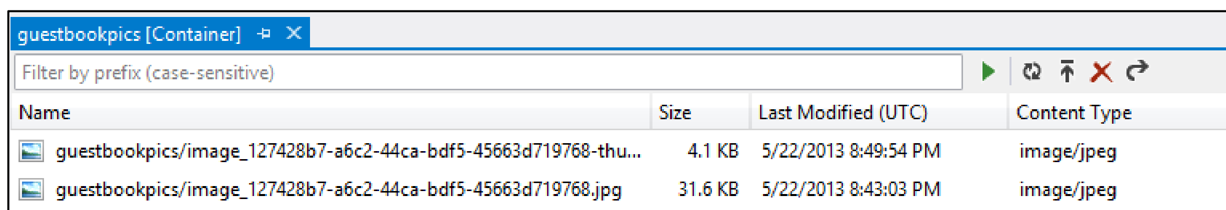


Рисунок 7.3. Контейнер, содержащий исходное изображение и созданную миниатюру

5. Добавьте еще несколько записей в гостевую книгу. Убедитесь, что записи обновляются через несколько секунд, что свидетельствует о правильной работе фонового процесса.
6. Нажмите **SHIFT** + **F5,** чтобы остановить процесс отладки.

Мобильные технологии

Приложение DroidQuest состоит из *активности* (activity) и *макета* (layout).

Активность представлена экземпляром Activity — класса из Android SDK. Она отвечает за взаимодействие пользователя с информацией на экране.

Чтобы реализовать функциональность, необходимую приложению, разработчик пишет субклассы Activity. В простом приложении бывает достаточно одного субкласса; в сложном приложении их может потребоваться несколько.

DroidQuest — простое приложение, поэтому в нем используется всего один субкласс Activity с именем QuestActivity. Класс QuestActivity управляет пользовательским интерфейсом, изображенным на рисунке 7.4.

Макет определяет набор объектов пользовательского интерфейса и их расположение на экране. Приложение DroidQuest включает файл макета с именем activity_quest.xml. Разметка XML в этом файле определяет пользовательский интерфейс, изображенный на рисунке 7.4.

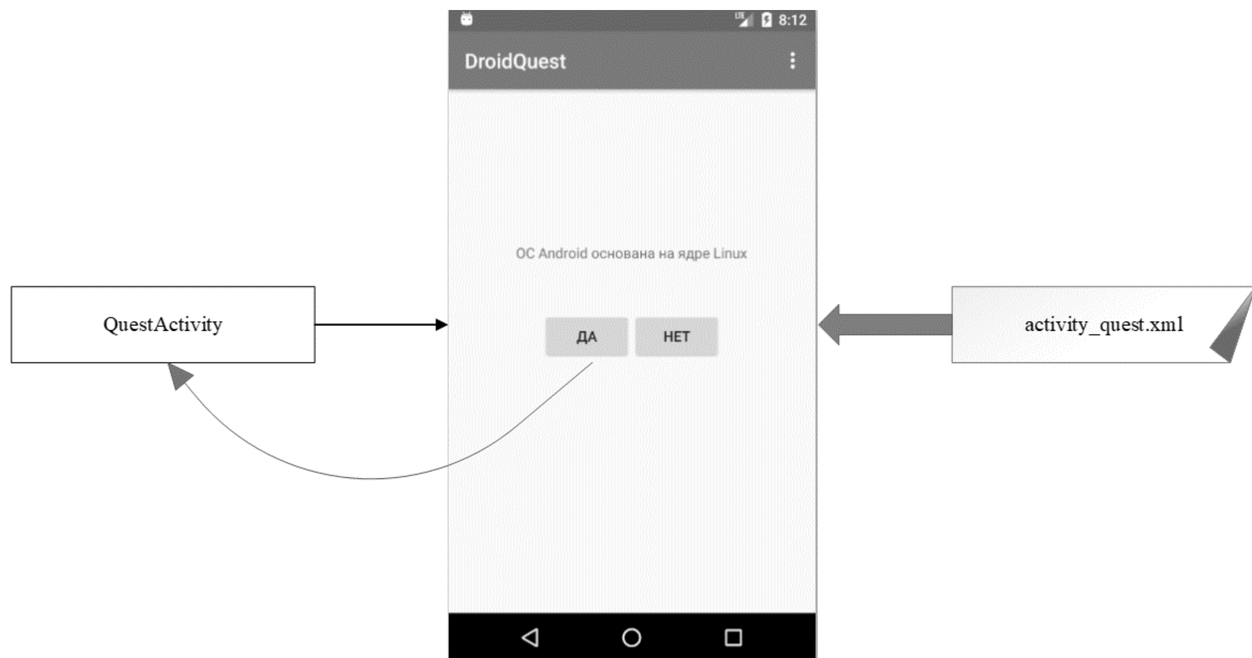


Рисунок 7.4. QuestActivity управляет интерфейсом, определяемым в файле activity_quest.xml

Задание 1 – Создание проекта Android

Проект Android содержит файлы, из которых состоит приложение. Чтобы создать новый проект, откройте Android Studio. Если Android Studio запускается на компьютере впервые, то на экране появляется диалоговое окно с приветствием (рисунок 7.5).

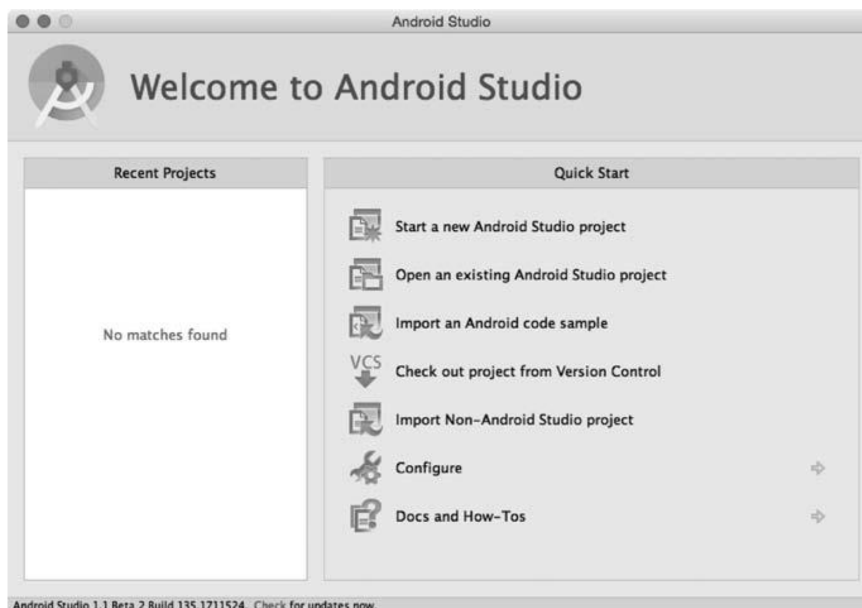


Рисунок 7.5. Добро пожаловать в Android Studio

Выберите в диалоговом окне команду *Start a new Android Studio project*. Если диалоговое окно не отображается при запуске, значит, ранее уже

создавались другие проекты. В таком случае выполните команду File→New Project....

Открывается мастер создания проекта. На первом экране мастера введите имя приложения **DroidQuest** (рисунок 7.6). В поле Company Domain введите строку ***android.rsue.ru*** (*android* – фамилия студента на латинице); сгенерированное имя пакета (Package Name) при этом автоматически меняется на *ru.rsue.android.driodquest*. В поле Project location введите любую папку своей файловой системы на свое усмотрение.

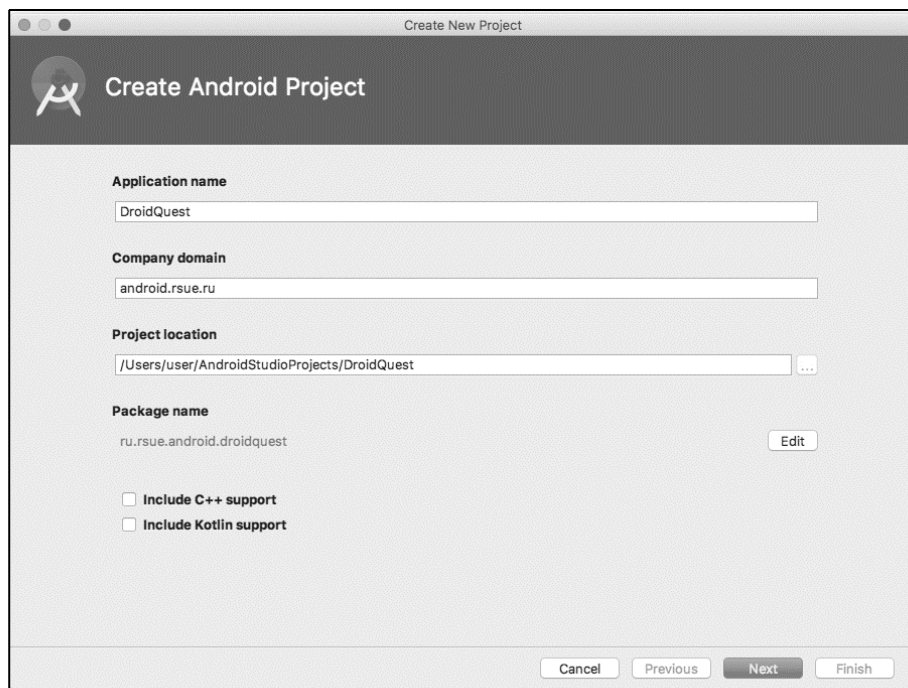


Рисунок 7.6. Создание нового проекта

Обратите внимание: в имени пакета используется схема «обратного DNS», согласно которой доменное имя организации записывается в обратном порядке с присоединением суффиксов дополнительных идентификаторов. Эта схема обеспечивает уникальность имен пакетов и позволяет различать приложения на устройстве и в Google Play.

Щелкните на кнопке Next. На следующем экране можно ввести дополнительную информацию об устройствах, которые необходимо поддерживать. Приложение DroidQuest будет поддерживать только телефоны, поэтому установите только флажок Phone and Tablet. Выберите в списке минимальную версию SDK API 21: Android 5.0 (Lollipop) (рисунок 7.7).

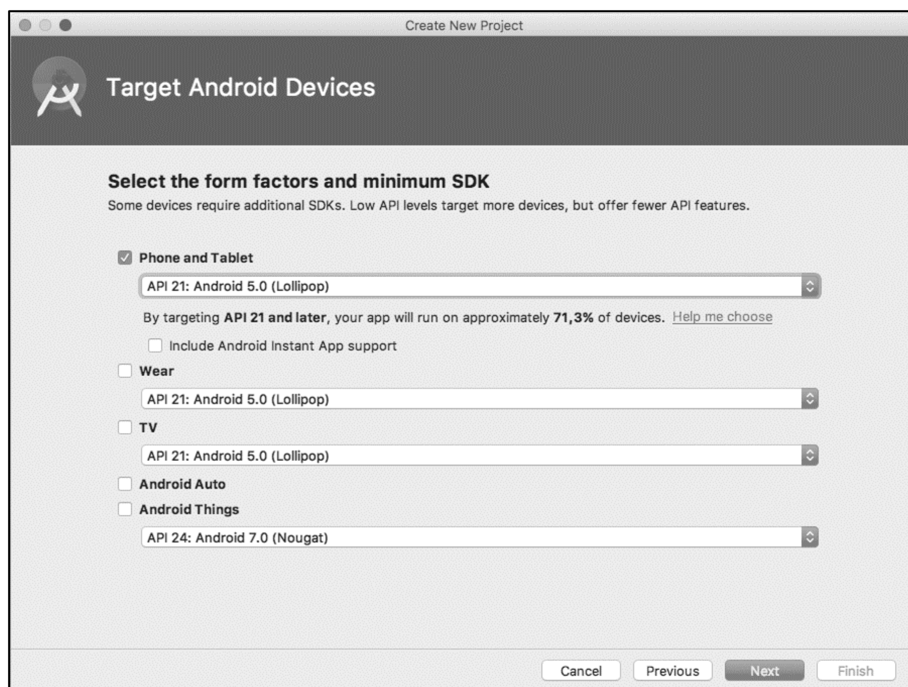


Рисунок 7.7. Определение поддерживаемых устройств

Щелкните на кнопке Next.

На следующем экране необходимо выбрать шаблон первого экрана DroidQuest (рисунок 7.8). Выберите пустую активность (Blank Activity / Empty Activity) и щелкните на кнопке Next.

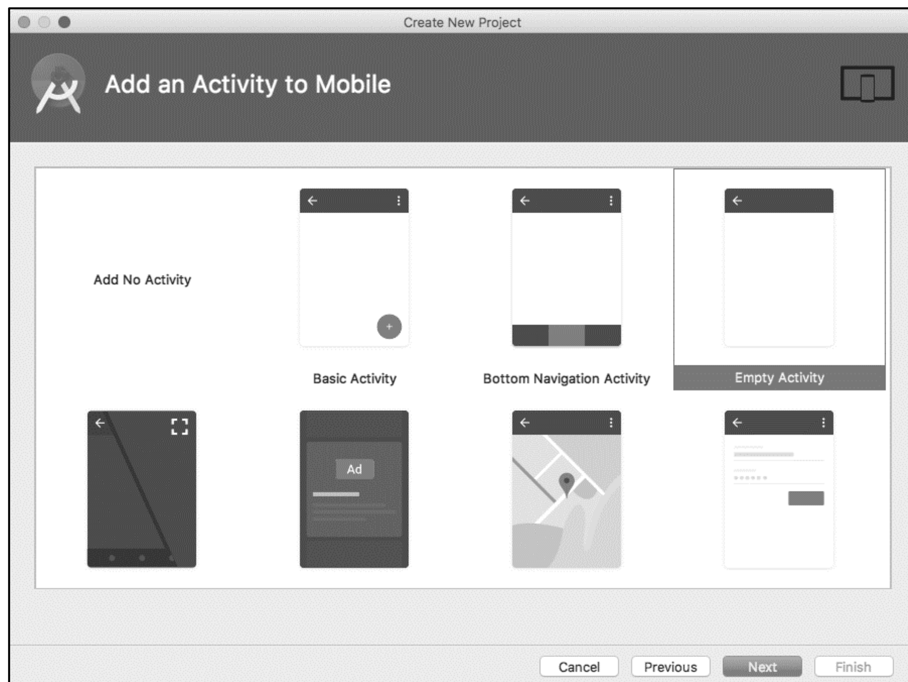


Рисунок 7.8. Выбор типа активности

В последнем диалоговом окне мастера введите имя субкласса активности QuestActivity (рисунок 7.9). Обратите внимание на суффикс Activity в имени класса. Его присутствие не обязательно, но это очень полезное соглашение, которое стоит соблюдать.

Имя макета автоматически заменяется на activity_quest в соответствии с переименованием активности. Имя макета записывается в порядке, обратном имени активности; в нем используются символы нижнего регистра, а слова разделяются символами подчеркивания.

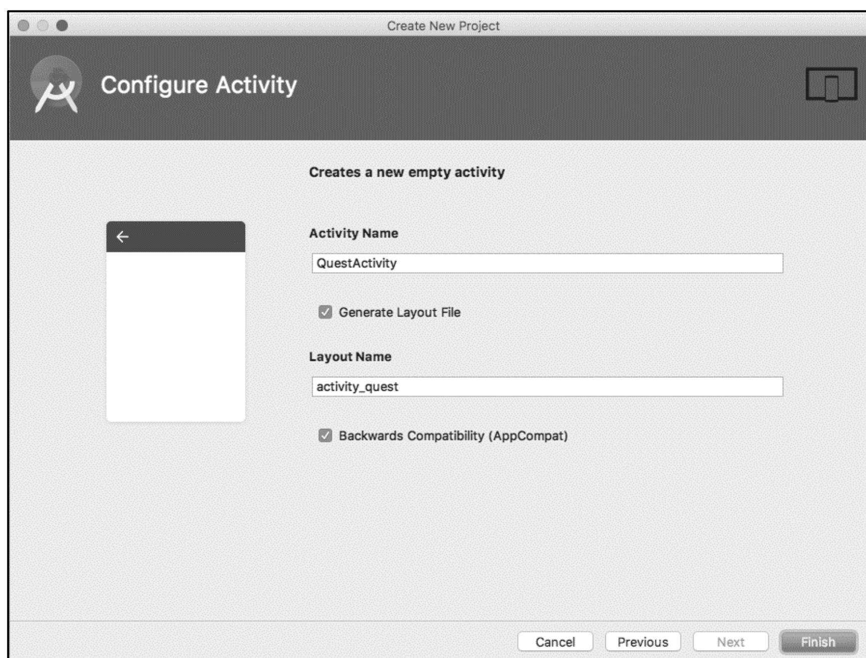


Рисунок 7.9. Настройка новой активности

Щелкните на кнопке Finish. Среда Android Studio создает и открывает новый проект.

Навигация в Android Studio

Среда Android Studio открывает созданный проект в окне, изображенном на рисунке 7.10.

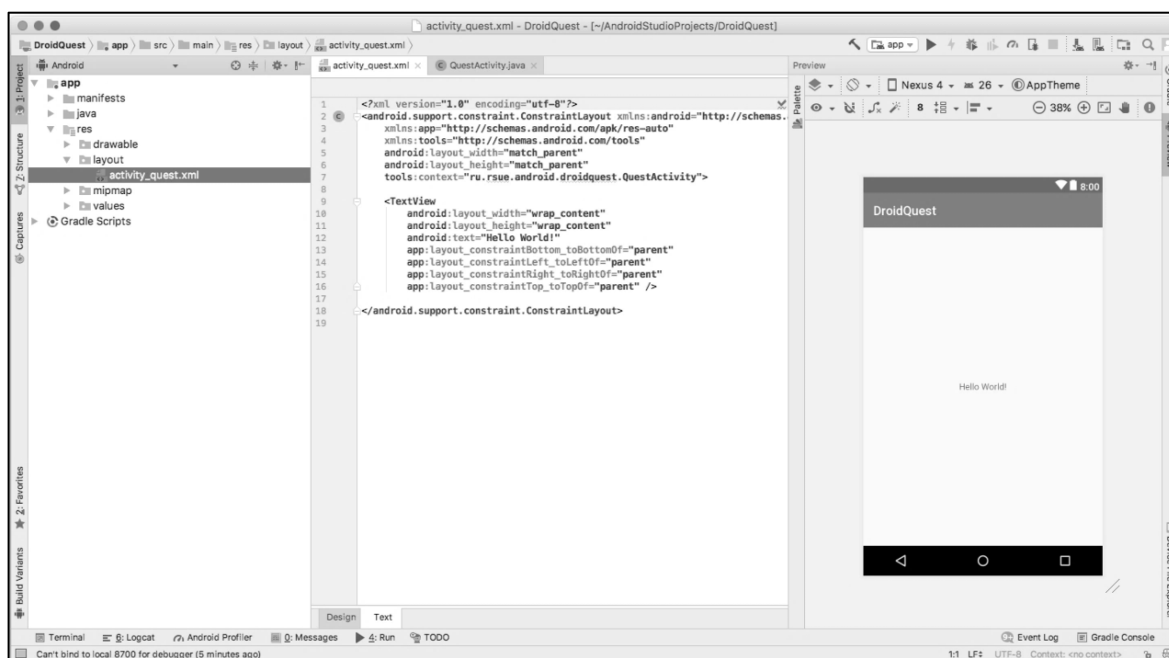


Рисунок 7.10. Окно нового проекта

Различные части окна проекта называются *панелями*.

Слева располагается *окно инструментов Project*. В нем выполняются операции с файлами, относящимися к проекту.

В середине находится панель *редактора*, в котором Android Studio открывает в файл activity_quest.xml. (Если в редакторе выводится изображение, щелкните на вкладке Text в нижней части панели.) На панели в правой части окна показано, как выглядит содержимое файла в режиме предварительного просмотра.

Видимостью различных панелей можно управлять, щелкая на их именах на полосе кнопок инструментов у левого, правого или нижнего края экрана. Также для многих панелей определены специальные комбинации клавиш. Если полосы с кнопками не отображаются, щелкните на серой квадратной кнопке в левом нижнем углу главного окна или выполните команду View→Tool Buttons.

Задание 2 – Построение макета пользовательского интерфейса

На данный момент файл activity_quest.xml определяет разметку для активности по умолчанию. Разметка шаблона часто изменяется, но XML будет выглядеть примерно так, как показано в листинге 7.1.

Листинг 7.1. Разметка для активности

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Макет активности по умолчанию определяет два *виджета* (widgets): RelativeLayout и TextView.

На рисунке 7.11 показано, как выглядят на экране виджеты ConstraintLayout и TextView, определенные в листинге 7.1.

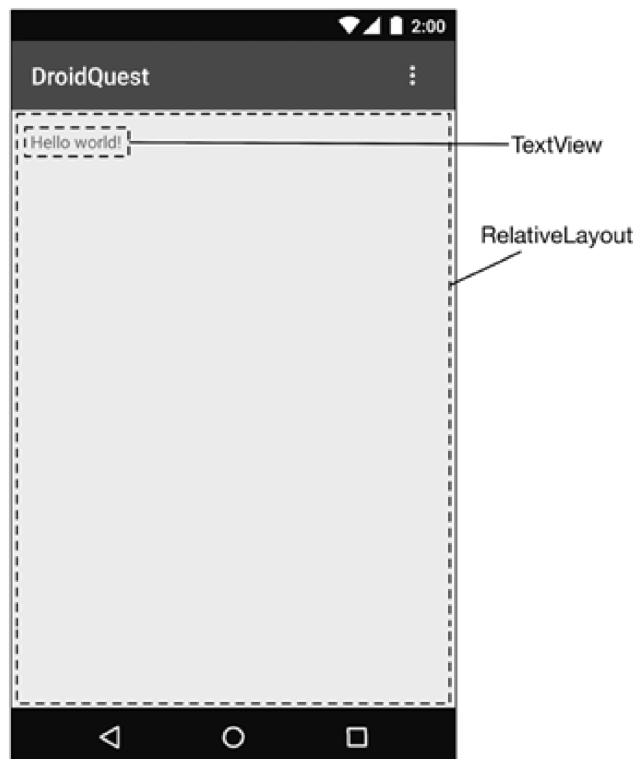


Рисунок 7.11. Виджеты по умолчанию на экране

В интерфейсе QuestActivity будут задействованы пять виджетов: вертикальный виджет LinearLayout; TextView; горизонтальный виджет LinearLayout; две кнопки Button.

На рисунке 7.12 показано, как из этих виджетов образуется интерфейс QuestActivity.

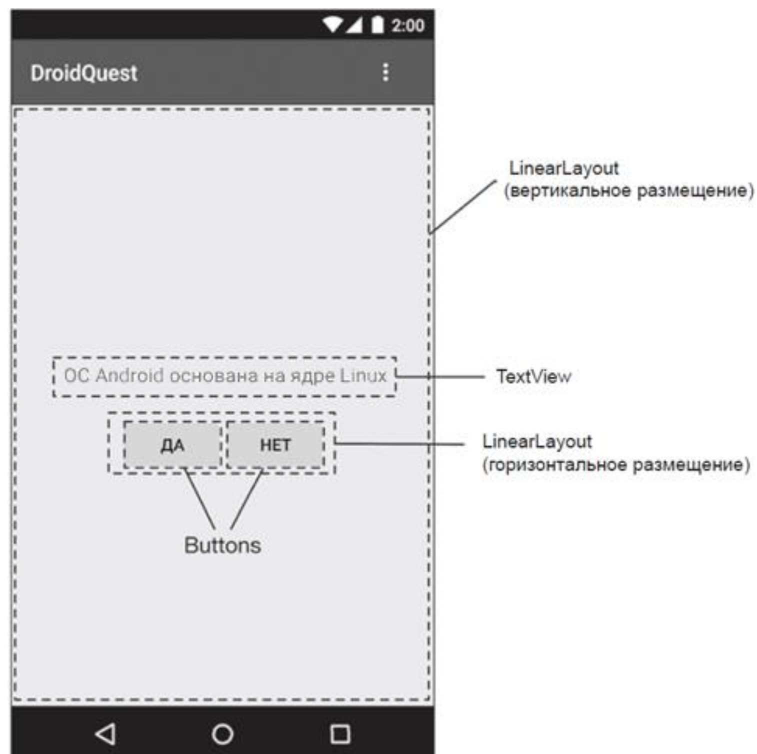


Рисунок 7.12. Запланированное расположение виджетов на экране

Теперь нужно определить эти виджеты в файле activity_quest.xml.

Внесите в файл activity_quest.xml изменения, представленные в листинге 7.2 – Разметка XML, которую нужно удалить, выделена перечеркиванием, а добавляемая разметка XML выделена жирным шрифтом.

Листинг 7.2. Определение виджетов в XML (activity_quest.xml)

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     android:gravity="center"
     android:orientation="vertical" >
     <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        android:text="@string/question_text" />
     <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/true_button" />
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/false_button" />
     </LinearLayout>
</LinearLayout>
```

Сравните XML с пользовательским интерфейсом, изображенным на рисунке 7.13 – Каждому виджету в разметке соответствует элемент XML. Имя элемента определяет тип виджета. Каждый элемент обладает набором *атрибутов* XML. Атрибуты можно рассматривать как инструкции по настройке виджетов.

Иерархия представлений

Виджеты входят в иерархию объектов View, называемую *иерархией представлений*. На рисунке 7.13 изображена иерархия виджетов для разметки XML из листинга 7.2.

Корневым элементом иерархии представлений в этом макете является элемент LinearLayout. В нем должно быть указано пространство имен XML ресурсов Android *http://schemas.android.com/apk/res/android*.

LinearLayout наследует от субкласса View с именем ViewGroup. Виджет ViewGroup предназначен для хранения и размещения других виджетов. LinearLayout используется в тех случаях, когда необходимо выстроить виджеты в один столбец или строку. Другие субклассы ViewGroup — FrameLayout, TableLayout и RelativeLayout.

Если виджет содержится в ViewGroup, он называется *потомком* (child) ViewGroup. Корневой элемент LinearLayout имеет двух потомков: TextView и другой элемент LinearLayout. У LinearLayout имеются два собственных потомка Button.

Атрибуты виджетов

Рассмотрим некоторые атрибуты, используемые для настройки виджетов.

`android:layout_width` и **`android:layout_height`**

Атрибуты `android:layout_width` и `android:layout_height`, определяющие ширину и высоту, необходимы практически для всех разновидностей виджетов. Как правило, им задаются значения `match_parent` или `wrap_content`:

- `match_parent` — размеры представления определяются размерами родителя;
- `wrap_content` — размеры представления определяются размерами содержимого.

(Иногда в разметке встречается значение `fill_parent`. Это устаревшее значение эквивалентно `match_parent`.)

В корневом элементе `LinearLayout` атрибуты ширины и высоты равны match_parent. Элемент `LinearLayout` является корневым, но у него все равно есть родитель — представление, которое предоставляет Android для размещения иерархии представлений данного приложения.

У других виджетов макета ширине и высоте задается значение wrap_content. На рисунке 7.13 показано, как в этом случае определяются их размеры.

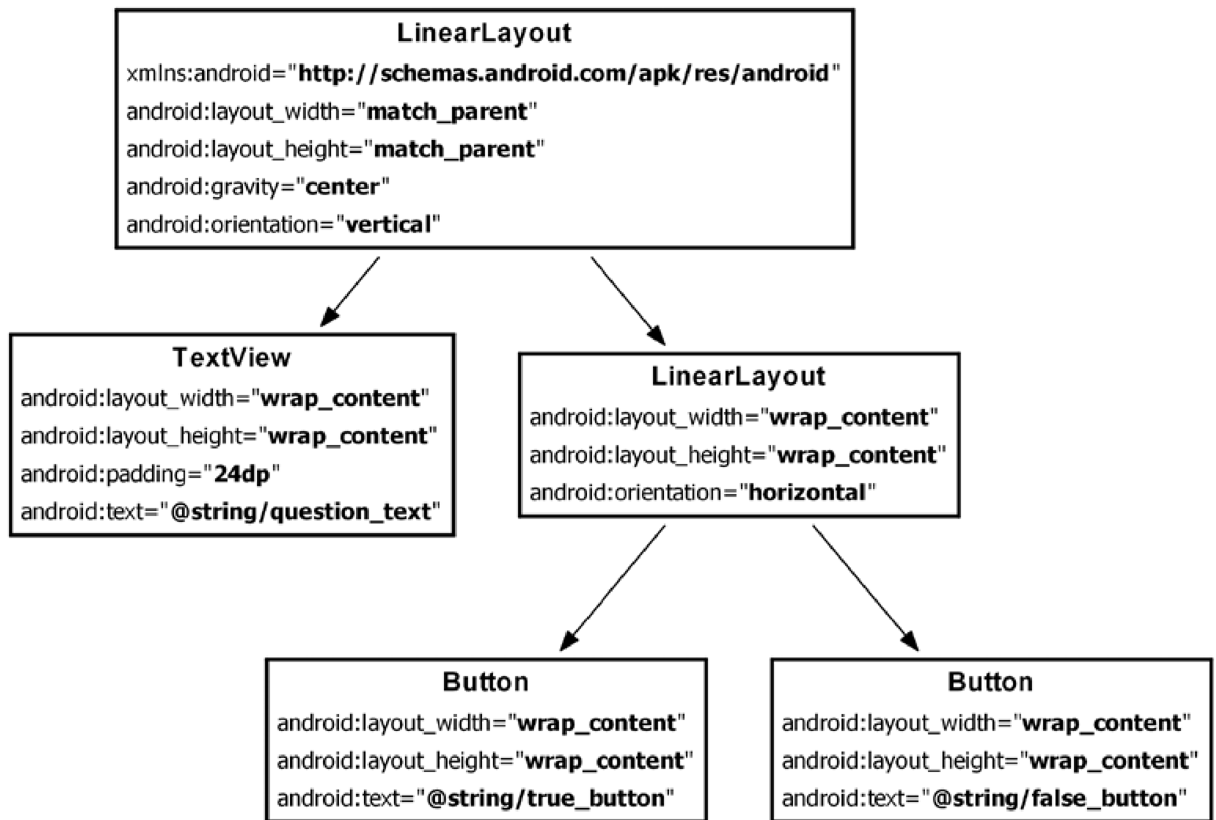


Рисунок 7.13. Иерархия виджетов и атрибутов

Виджет TextView содержит чуть больше текста из-за атрибута `android:padding="24dp"`. Этот атрибут приказывает виджету добавить заданный отступ вокруг содержимого при определении размера, чтобы текст вопроса не соприкасался с кнопкой (dp — это пикселы, не зависящие от плотности (density-independent pixels)).

– **`android:orientation`**

Атрибут `android:orientation` двух виджетов `LinearLayout` определяет, как будут выстраиваться потомки — по вертикали или горизонтали. Корневой элемент `LinearLayout` имеет вертикальную ориентацию; у его потомка `LinearLayout` горизонтальная ориентация.

Порядок определения потомков определяет порядок их отображения на экране. В вертикальном элементе `LinearLayout` потомок, определенный первым, располагается выше остальных. В горизонтальном элементе LinearLayout первый потомок является крайним левым. (Если только на устройстве не используется язык с письменностью справа налево, например, арабский или иврит; в этом случае первый потомок будет находиться в крайней правой позиции.)

– **android:text**

Виджеты TextView и Button содержат атрибуты android:text. Этот атрибут сообщает виджету, какой текст должен в нем отображаться.

Обратите внимание: значения атрибутов представляют собой не строковые литералы, а ссылки на строковые ресурсы.

Строковый ресурс — строка, находящаяся в отдельном файле XML, который называется *строковым файлом*. Виджету можно назначить фиксированную строку (например, `android:text="True"`), но так делать не стоит. Лучше размещать строки в отдельном файле, а затем ссылаться на них, так как использование строковых ресурсов упрощает локализацию.

Строковые ресурсы, на которые ссылается activity_quest.xml, еще не существуют. Это необходимо исправить.

Создание строковых ресурсов

Каждый проект включает строковый файл по умолчанию с именем strings.xml. Найдите в окне Project каталог app/res/values, раскройте его и откройте файл strings.xml.

В шаблон уже включено несколько строковых ресурсов. Удалите неиспользуемую строку с именем hello_world и добавьте три новые строки для макета.

Листинг 7.3. Добавление строковых ресурсов (strings.xml)

```
<resources>
      <string name="app_name">DroidQuest</string>
      <string name="hello_world">Hello world!</string>
      <string name="question_text">ОС Android основана на ядре
            Linux</string>
      <string name="true_button">Да</string>
      <string name="false_button">Нет</string>
      <string name="action_settings">Settings</string>
</resources>
```

Теперь по ссылке @string/false_button в любом файле XML проекта DroidQuest будем получать строковый литерал "Нет" на стадии выполнения.

Сохраните файл strings.xml. Если в файле activity_quest.xml оставались ошибки, связанные с отсутствием строковых ресурсов, они должны исчезнуть. (Если ошибки остались, проверьте оба файла — возможно, где-то допущена опечатка.)

Строковый файл по умолчанию называется strings.xml, но ему можно присвоить любое имя. Проект может содержать несколько строковых файлов. Если файл находится в каталоге res/values/, содержит корневой элемент resources и дочерние элементы string, строки будут найдены и правильно использованы приложением.

Предварительный просмотр макета

Макет готов, и его можно просмотреть в графическом конструкторе (рисунок 7.14). Прежде всего убедитесь в том, что файлы сохранены и не содержат ошибок. Затем вернитесь к файлу activity_quest.xml и откройте панель Preview при помощи вкладки в правой части редактора.

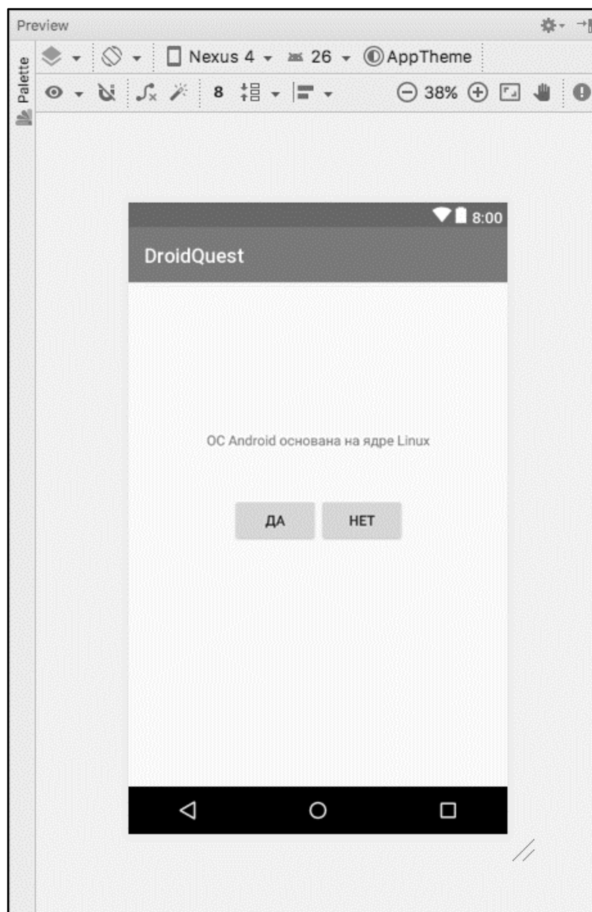


Рисунок 7.14. Предварительный просмотр в графическом конструкторе макетов (activity_quest.xml)

От разметки XML к объектам View

При создании проекта DroidQuest был автоматически создан субкласс Activity с именем QuestActivity. Файл класса QuestActivity находится в каталоге app/java (в котором хранится Java/Kotlin-код проекта).

В окне инструментов Project откройте каталог app/java, а затем содержимое пакета ru.rsue.android.driodquest. Откройте файл QuestActivity.kt и просмотрите его содержимое (листинг 7.4).

Листинг 7.4. Файл класса QuestActivity по умолчанию (QuestActivity.kt)

```
package ru.rsue.droidquest
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class QuestActivity: AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_quest)
    }
}
```

(AppCompatActivity — это субкласс, наследующий от класса Android Activity и обеспечивающий поддержку старых версий Android.)

(Если не все директивы import отображены, щелкните на знаке ⊕ слева от первой директивы import, чтобы раскрыть список.)

Файл может содержать три метода Activity: `onCreate(Bundle)`, `onCreateOptionsMenu(Menu)` и `onOptionsItemSelected(MenuItem)`.

Метод `onCreate(Bundle)` вызывается при создании экземпляра субкласса активности. Такому классу нужен пользовательский интерфейс, которым он будет управлять. Чтобы предоставить классу активности его пользовательский интерфейс, следует вызвать следующий метод Activity:

```
public void setContentView(int layoutResID);
```

Этот метод *заполняет* (inflates) макет и выводит его на экран. При заполнении макета создаются экземпляры всех виджетов в файле макета с параметрами, определяемыми его атрибутами. Чтобы указать, какой именно макет следует заполнить, необходимо передать идентификатор ресурса макета.

Ресурсы и идентификаторы ресурсов

Для обращения к ресурсу в коде используется его идентификатор ресурса. Макету приложения назначен идентификатор ресурса `R.layout.activity_quest`.

Чтобы просмотреть текущие идентификаторы ресурсов проекта DroidQuest, необходимо сначала изменить режим представления проекта. По умолчанию Android Studio использует режим представления Android (рисунок 7.15). В этом режиме истинная структура каталогов проекта Android скрывается, чтобы можно было сосредоточиться на тех файлах и папках, которые чаще всего нужны программисту.

Найдите раскрывающийся список в верхней части окна инструментов Project и выберите вместо режима Android режим Project. В этом режиме файлы и папки проекта представлены в своем фактическом состоянии.

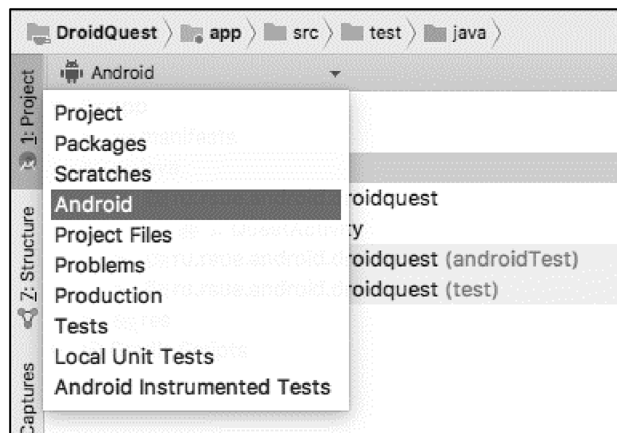


Рисунок 7.15. Изменение режима представления проекта

Чтобы просмотреть ресурсы приложения DroidQuest, раскройте содержимое каталога app/build/generated/source/r/debug. В этом каталоге найдите имя пакета проекта и откройте файл R.java из этого пакета. Поскольку этот файл генерируется процессом сборки Android, не следует его изменять, о чем деликатно предупреждает надпись в начале файла.

Листинг 7.5. Текущие идентификаторы DroidQuest

```
/* AUTO-GENERATED FILE. DO NOT MODIFY.
... */
package ru.rsue.android.driodquest;
public final class R {
      public static final class anim {
         ...
      }
      ...
      public static final class id {
         ...
      }
      public static final class layout {
         ...
         public static final int activity_quest=0x7f030017;
      }
      public static final class mipmap {
         public static final int ic_launcher=0x7f030000;
      }
      public static final class string {
         ...
         public static final int app_name=0x7f0a0010;
         public static final int correct_toast=0x7f0a0011;
         public static final int false_button=0x7f0a0012;
         public static final int incorrect_toast=0x7f0a0013;
         public static final int question_text=0x7f0a0014;
         public static final int true_button=0x7f0a0015;
      }
```

```
}
```

Файл R.java может быть довольно большим; в листинге 7.5 значительная часть его содержимого не показана.

Теперь понятно, откуда взялось имя R.layout.activity_quest — это целочисленная константа с именем activity_quest из внутреннего класса layout класса R.

Строкам также назначаются идентификаторы ресурсов. В приложении еще не было ссылок на строки в коде, но эти ссылки обычно выглядят так:

```
setTitle(R.string.app_name);
```

Android генерирует идентификатор ресурса для всего макета и для каждой строки, но не для отдельных виджетов из файла activity_quest.xml. Не каждому виджету нужен идентификатор ресурса.

Прежде чем генерировать идентификаторы ресурсов, переключитесь обратно в режим представления Android. Чтобы сгенерировать идентификатор ресурса для виджета, включите в определение виджета атрибут android:id. В файле activity_quest.xml добавьте атрибут android:id для каждой кнопки.

Листинг 7.6. Добавление идентификаторов кнопок (activity_quest.xml)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
           ... >
      <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:padding="24dp"
      android:text="@string/question_text" />
      <LinearLayout
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:orientation="horizontal">
          <Button
             android:id="@+id/true_button"
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
             android:text="@string/true_button" />
          <Button
             android:id="@+id/false_button"
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
             android:text="@string/false_button" />
      </LinearLayout>
</LinearLayout>
```

Обратите внимание: знак + присутствует в значениях android:id, но не в значениях android:text. Это связано с тем, что идентификаторы *создаются*, а на строки только *ссылаемся*.

Задание 3 – Подключение виджетов к программе

Теперь, когда кнопкам назначены идентификаторы ресурсов, к ним можно обращаться в QuestActivity. Все начинается с добавления двух переменных.

Введите следующий код в QuestActivity.kt. (Не используйте автозавершение; введите его самостоятельно.) После сохранения файла выводятся два сообщения об ошибках.

Листинг 7.7. Добавление полей (QuestActivity.kt)

```
class QuestActivity : AppCompatActivity() {

    private lateinit var mTrueButton: Button
    private lateinit var mFalseButton: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_quest)
    }
}
```

Обратите внимание на префикс **m** у имен двух полей (переменных экземпляров). Этот префикс соответствует схеме формирования имен Android.

Наведите указатель мыши на красные индикаторы ошибок. Они сообщают об одинаковой проблеме: «Не удается разрешить символическое имя Button» (Cannot resolve symbol ‘Button’).

Чтобы избавиться от ошибок, следует импортировать класс `android.widget.Button` в QuestActivity.kt. Введите следующую директиву импортирования в начале файла:

```
import android.widget.Button;
```

А можно пойти по простому пути и поручить эту работу Android Studio. Нажмите Alt+Enter — волшебство IntelliJ приходит на помощь. Новая директива import теперь появляется под другими директивами в начале файла. Этот прием часто бывает полезным, если код работает не так, как положено.

Теперь можно подключить виджеты-кнопки. Процедура состоит из двух шагов:

- получение ссылок на заполненные объекты View;
- назначение для этих объектов слушателей, реагирующих на действия пользователя.

Получение ссылок на виджеты

В классе активности можно получить ссылку на заполненный виджет, для чего используется следующий метод Activity:

```
public View findViewById(int id);
```

Метод получает идентификатор ресурса виджета и возвращает объект View.

В файле QuestActivity.kt по идентификаторам ресурсов кнопок можно получить заполненные объекты и присвоить их полям. Учтите, что возвращенный объект View перед присваиванием необходимо преобразовать в Button.

Листинг 7.8. Получение ссылок на виджеты (QuestActivity.kt)

```
class QuestActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_quest)

        mTrueButton = findViewById(R.id.true_button)
        mFalseButton = findViewById(R.id.false_button)
    }
}
```

Назначение слушателей

Приложения Android обычно *управляются событиями* (event-driven). В отличие от программ командной строки или сценариев, такие приложения запускаются и ожидают наступления некоторого события, например нажатия кнопки пользователем. (События также могут инициироваться ОС или другим приложением, но события, инициируемые пользователем, наиболее очевидны.)

Когда приложение ожидает наступления конкретного события, оно «прослушивает» данное событие. Объект, создаваемый для ответа на событие, называется *слушателем* (listener). Такой объект реализует *интерфейс слушателя* данного события.

Android SDK поставляется с интерфейсами слушателей для разных событий, поэтому вам не придется писать собственные реализации. В данном случае прослушиваемым событием является «щелчок» на кнопке, поэтому слушатель должен реализовать интерфейс View.OnClickListener.

В файле QuestActivity.kt включите следующий фрагмент кода в метод onCreate(…) непосредственно после присваивания.

Листинг 7.9. Назначение слушателя для кнопки True (QuestActivity.kt)

```
  ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_quest)

        mTrueButton = findViewById(R.id.true_button)
```

```
        mTrueButton.setOnClickListener(View.OnClickListener {
            fun OnClick(view: View){
                // Пока ничего не делает, но скоро будет!
            }
        })
        mFalseButton = findViewById(R.id.false_button)
    }
}
```

(Если появиться ошибка «View cannot be resolved to a type», воспользуйтесь комбинацией Alt+Enter для импортирования класса View.)

В листинге 7.9 назначается слушатель, информирующий о нажатии виджета Button с именем mTrueButton. Метод `setOnClickListener(OnClickListener)` получает в аргументе слушателя, а конкретнее — объект, реализующий `OnClickListener`.

Анонимные внутренние классы

Слушатель реализован в виде *анонимного внутреннего класса*. Возможно, синтаксис не очевиден; просто запомните: все, что заключено во внешнюю пару круглых скобок, передается `setOnClickListener(OnClickListener)`. В круглых скобках создается новый безымянный класс, вся реализация которого передается вызываемому методу.

```
mTrueButton.setOnClickListener(View.OnClickListener {
    fun OnClick(view: View){
        // Пока ничего не делает, но скоро будет!
    }
})
```

Все слушатели будут реализованы в виде анонимных внутренних классов. В этом случае реализация методов слушателя находится непосредственно там, где это необходимо, и уменьшаются затраты ресурсов на создание именованного класса, который будет использоваться только в одном месте.

Так как анонимный класс реализует `OnClickListener`, он должен реализовать единственный метод этого интерфейса `onClick(View)`. Интерфейс слушателя требует, чтобы метод `onClick(View)` был реализован, но не устанавливает никаких правил относительно того, *как именно* он будет реализован. В Kotlin реализация данного метода может быть опущена, а использовав лямбда-аргумент, который можно вынести из скобок, программный код слушателя может быть сокращен до следующего вида:

```
mTrueButton.setOnClickListener {
    // Пока ничего не делает, но скоро будет!
}
```

Назначьте аналогичного слушателя для кнопки False.

Листинг 7.10. Назначение слушателя для кнопки False (QuestActivity.kt)

```
...
    mTrueButton = findViewById(R.id.true_button)
    mTrueButton.setOnClickListener {
        // Пока ничего не делает, но скоро будет!
    }
    mFalseButton = findViewById(R.id.false_button)
    mFalseButton.setOnClickListener{
        // Пока ничего не делает, но скоро будет!
    }
}
```

Задание 4 – Уведомления

В данном приложении каждая кнопка будет выводить на экран временное *уведомление* (toast) — короткое сообщение, которое содержит какую-либо информацию для пользователя, но не требует ни ввода, ни действий (рисунок 7.16).

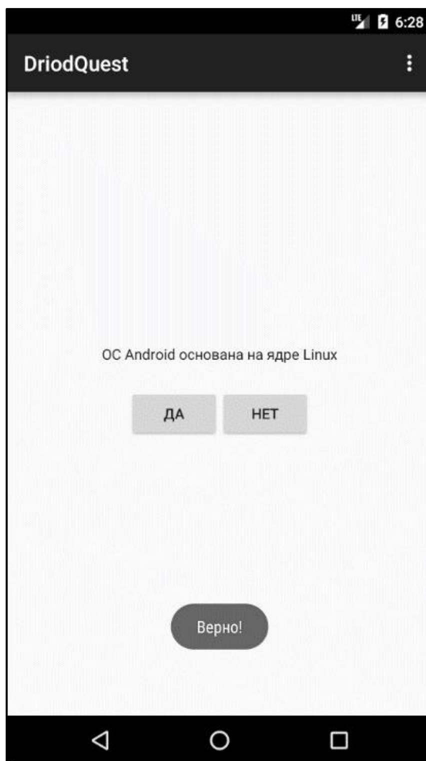


Рисунок 7.16. Уведомление с информацией для пользователя

Уведомления будут сообщать пользователю, правильно ли он ответил на вопрос.

Для начала следует вернуться к файлу strings.xml и добавить строковые ресурсы, которые будут отображаться в уведомлении.

Листинг 7.11. Добавление строк уведомлений (strings.xml)

```
<resources>
    ...
```

```
    <string name="false_button">Нет</string>
    <string name="correct_toast">Верно!</string>
    <string name="incorrect_toast">Не верно!</string>
    <string name="menu_settings">Settings</string>
</resources>
```

Уведомление создается вызовом следующего метода класса Toast:

```
public static Toast makeText(Context context, int resId, int duration);
```

Параметр Context обычно содержит экземпляр Activity (Activity является субклассом Context). Во втором параметре передается идентификатор ресурса строки, которая должна выводиться в уведомлении. Параметр Context необходим классу Toast для поиска и использования идентификатора ресурса строки. Третий параметр обычно содержит одну из двух констант Toast, определяющих продолжительность пребывания уведомления на экране.

После того как объект уведомления будет создан, вызовите Toast.show(), чтобы уведомление появилось на экране.

В классе QuestActivity вызов `makeText(…)` будет присутствовать в слушателе каждой кнопки (листинг 7.12).

Автозавершение

Начните вводить новый код из листинга 7.12 – Когда будет набрана точка после класса Toast, на экране появляется список методов и констант класса Toast.

Чтобы выбрать одну из рекомендаций, используйте клавиши ↑ или ↓ и нажмите клавиши Tab или Return/Enter. Выберите в списке рекомендаций метод `makeText(Context context, int resID, int duration)`. Механизм автозавершения добавляет полный вызов метода.

Задайте параметры метода makeText так, как показано в листинге 7.12.

Листинг 7.12. Создание уведомлений (QuestActivity.kt)

```
...
mTrueButton = findViewById(R.id.true_button)
mTrueButton.setOnClickListener {
    Toast.makeText(this, R.string.correct_toast,
        Toast.LENGTH_SHORT).show()
    // Пока ничего не делает, но скоро будет!
}
mFalseButton = findViewById(R.id.false_button)
mFalseButton.setOnClickListener{
    Toast.makeText(this, R.string.incorrect_toast,
        Toast.LENGTH_SHORT).show()
    // Пока ничего не делает, но скоро будет!
}
```

В вызове `makeText(…)` экземпляр QuestActivity передается в аргументе Context. Но просто передать this нельзя. В этом месте кода определяется анонимный класс, где `this` обозначает `View.OnClickListener`.

Благодаря использованию автозавершения не придется ничего специально делать для импортирования класса Toast. Когда происходит соглашение на рекомендацию автозавершения, необходимые классы импортируются автоматически.

Сохраните внесенные изменения.

Задание 5 – Создание нового класса. Обновление уровня представления

В окне инструментов Project щелкните правой кнопкой мыши на пакете ru.rsue.android.driodquest и выберите команду New→Kotlin Class. Введите имя класса Question и щелкните на кнопке OK (рисунок 7.17).

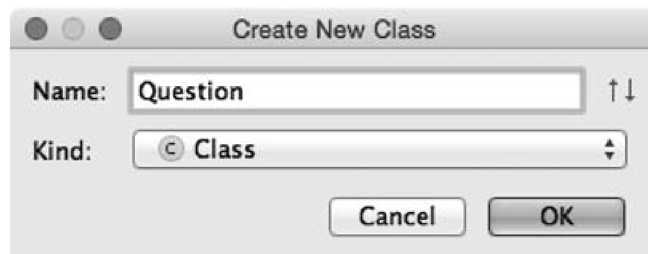


Рисунок 7.17. Создание класса Question

Объявите созданный класс как data класс и добавьте в конструктор два поля.

Листинг 7.13. Добавление класса Question (Question.kt)

```
data class Question (val textResId: Int,
                     val answerTrue: Boolean)
```

Класс Question содержит два вида данных: текст вопроса и правильный ответ (да/нет).

Класс Question готов.

Обновление уровня представления

Обновим уровень представления DroidQuest и включим в него кнопку «Далее».

В Android объекты уровня представления обычно заполняются на основе разметки XML в файле макета. Весь макет DroidQuest определяется в файле activity_quest.xml. В него следует внести изменения, представленные на рисунке 7.18. (Для экономии места на рисунке не показаны атрибуты виджетов, оставшихся без изменений.)

Итак, на уровне представления необходимо внести следующие изменения:

– удалить атрибут android:text из TextView. Жестко запрограммированный текст вопроса не должен присутствовать в определении;

– назначить TextView атрибут android:id. Идентификатор ресурса необходим виджету для того, чтобы задать его текст в коде QuestActivity;

– добавить новый виджет Button как потомка корневого элемента LinearLayout.

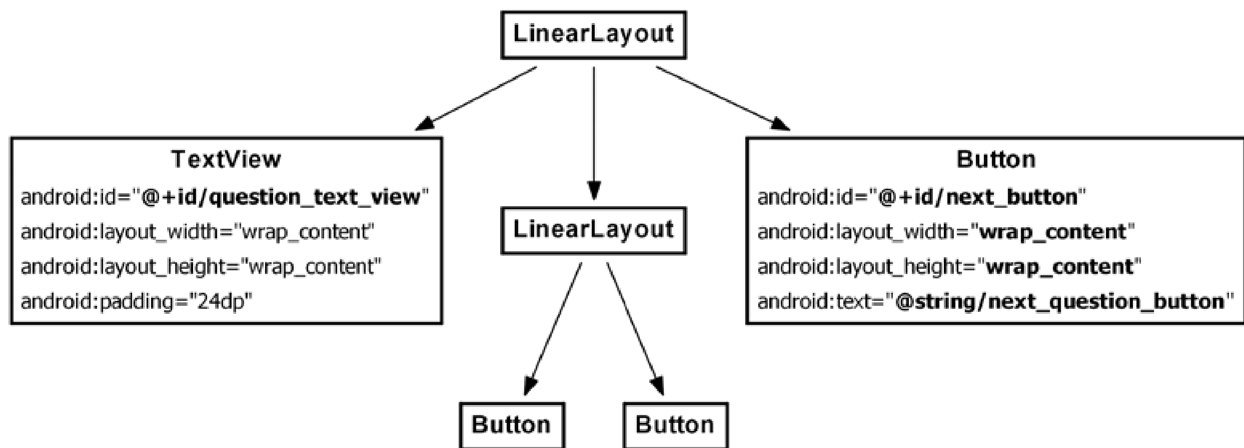


Рисунок 7.18. Добавление новой кнопки

Вернитесь к файлу activity_quest.xml и выполните все перечисленные действия.

Листинг 7.14. Новая кнопка и изменения в TextView (activity_quest.xml)

```
<LinearLayout
        ... >
    <TextView
    android:id="@+id/question_text_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="24dp"
    android:text="@string/question_text"
    />
    <LinearLayout
        ... >
    ...
    </LinearLayout>
    <Button
    android:id="@+id/next_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/next_button" />
</LinearLayout>
```

Сохраните файл activity_quest.xml. Возможно, на экране появится ошибка с сообщением об отсутствующем строковом ресурсе.

Вернитесь к файлу res/values/strings.xml. Удалите строку вопроса и добавьте строку для новой кнопки.

Листинг 7.15. Обновленные строки (strings.xml)

```
...
      <string name="app_name">DroidQuest</string>
      <string name="question_text">ОС Android основана на ядре Linux
            </string>
      <string name="true_button">Да</string>
      <string name="false_button">Нет</string>
      <string name="next_button">Далее</string>
      <string name="correct_toast">Верно!</string>
...
```

Далее в файл strings.xml необходимо добавить строки с вопросами по ОС Android, которые будут предлагаться пользователю.

Листинг 7.16. Добавление строк вопросов (strings.xml)

```
...
      <string name="incorrect_toast">Не верно!</string>
      <string name="action_settings">Settings</string>
      <string name="question_android">ОС Android основана на ядре
            Linux.</string>
      <string name="question_linear">Разметка LinearLayout позиционирует
            свои дочерние элементы в строки и столбцы.</string>
      <string name="question_service">Компонент Android-приложения Service
            предоставляют доступ к данным.</string>
      <string name="question_res">Ресурсы приложения содержатся в отдельном
            файле XML.</string>
      <string name="question_manifest">AndroidManifest.xml предоставляет
            системе основную информацию о программе.</string>
...
```

Сохраните файлы. Вернитесь к файлу activity_quest.xml и ознакомьтесь с изменениями макета в графическом конструкторе.

Задание 6 – Обновление уровня контроллера

В предыдущей работе в контроллере DroidQuest — QuestActivity — не происходило почти ничего. Он отображал макет, определенный в файле activity_quest.xml, назначал слушателей для двух кнопок и организовывал выдачу уведомлений.

Теперь, когда у нас появились дополнительные вопросы, классу QuestActivity придется приложить дополнительные усилия для связывания уровней модели и представления DroidQuest.

Откройте файл QuestActivity.kt. Добавьте переменные для TextView и новой кнопки Button. Также создайте список объектов Question и переменную для индекса списка.

Листинг 7.17. Добавление переменных и списка Question (QuestActivity.kt)

```
class QuestActivity : AppCompatActivity() {

    private lateinit var mTrueButton: Button
    private lateinit var mFalseButton: Button
    private lateinit var mNextButton: Button
    private lateinit var mQuestionTextView: TextView
    private val mQuestionBank = listOf(
        Question(R.string.question_android, true),
        Question(R.string.question_linear, false),
        Question(R.string.question_service, false),
        Question(R.string.question_res, true),
        Question(R.string.question_manifest, true)
    )
    private var mCurrentIndex = 0
...
```

Программа несколько раз вызывает конструктор Question и создает список объектов Question.

Мы собираемся использовать mQuestionBank, mCurrentIndex и методы доступа Question для вывода на экран серии вопросов.

Начнем с получения ссылки на TextView и задания тексту виджета вопроса с текущим индексом.

Листинг 7.18. Подключение виджета TextView (QuestActivity.kt)

```
class QuestActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_quest)
        mQuestionTextView = findViewById(R.id.question_text_view)
        var question = mQuestionBank[mCurrentIndex].textResId
        mQuestionTextView.setText(question)
        mTrueButton = findViewById(R.id.true_button)
        ...
      }
}
```

Сохраните файлы и проверьте возможные ошибки. Запустите программу DroidQuest. Первый вопрос из списка должен отображаться в виджете TextView.

Получите ссылку на кнопку, назначьте ей слушателя View.OnClickListener. Этот слушатель будет увеличивать индекс и обновлять текст TextView.

Листинг 7.19. Подключение новой кнопки (QuestActivity.kt)

```
class QuestActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_quest)
        ...
        mFalseButton.setOnClickListener {
            Toast.makeText(
                this, R.string.incorrect_toast, Toast.LENGTH_SHORT
            ).show()
        }
        mNextButton = findViewById(R.id.next_button)
        mNextButton.setOnClickListener {
            mCurrentIndex = (mCurrentIndex + 1) % mQuestionBank.size
            question = mQuestionBank[mCurrentIndex].textResId
            mQuestionTextView.setText(question)
        }
    }
}
```

Обновление переменной mQuestionTextView осуществляется в двух разных местах. Лучше выделить этот код в закрытый метод, как показано в листинге 7.20, и вызвать этот метод в слушателе mNextButton и в конце onCreate(Bundle) для исходного заполнения текста в представлении активности.

Листинг 7.20. Инкапсуляция в методе updateQuestion() (QuestActivity.kt)

```
class QuestActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_quest)
        mQuestionTextView = findViewById(R.id.question_text_view)
        var question = mQuestionBank[mCurrentIndex].textResId
        mQuestionTextView.setText(question)
        mTrueButton = findViewById(R.id.true_button);
        ...
        mNextButton = findViewById(R.id.next_button)
        mNextButton.setOnClickListener {
            mCurrentIndex = (mCurrentIndex + 1) % mQuestionBank.size
            var question = mQuestionBank[mCurrentIndex].textResId
            mQuestionTextView.setText(question)
```

```
            updateQuestion()
        }
        updateQuestion()
    }

    private fun updateQuestion() {
        val question = mQuestionBank[mCurrentIndex].textResId
        mQuestionTextView.setText(question)
    }
}
```

Запустите DroidQuest и протестируйте новую кнопку «Далее».

В текущем состоянии приложение DroidQuest считает, что на все вопросы ответ должен быть положительным; исправим этот недостаток. Для этого будет реализован закрытый метод для инкапсуляции кода вместо того, чтобы вставлять одинаковый код в двух местах.

Сигнатура метода, который будет добавлен в QuestActivity, выглядит так:

```
private fun checkAnswer(userPressedTrue: Boolean)
```

Метод получает логическую переменную, которая указывает, какую кнопку нажал пользователь: True или False. Ответ пользователя проверяется по ответу текущего объекта Question. Наконец, после определения правильности ответа метод создает уведомление для вывода соответствующего сообщения.

Включите в файл QuestActivity.kt реализацию checkAnswer(boolean), приведенную в листинге 7.21.

Листинг 7.21. Добавление метода checkAnswer(Boolean) (QuestActivity.kt)

```
class QuestActivity : AppCompatActivity {
     ...
    private fun updateQuestion() {
        val question = mQuestionBank[mCurrentIndex].textResId
        mQuestionTextView.setText(question)
    }

    private fun checkAnswer(userPressedTrue: Boolean) {
        val answerIsTrue = mQuestionBank[mCurrentIndex].answerTrue
        val messageResId = if (userPressedTrue == answerIsTrue) {
            R.string.correct_toast
        } else {
            R.string.incorrect_toast
        }
        Toast.makeText(this, messageResId, Toast.LENGTH_SHORT).show()
    }
}
```

Включите в слушателя кнопки вызов checkAnswer(boolean), как показано в листинге 7.22.

Листинг 7.22. Вызов метода checkAnswer(boolean) (QuestActivity.kt)

```
class QuestActivity : AppCompatActivity {
      ...
      override fun onCreate(savedInstanceState: Bundle?) {
          ...
          mTrueButton = findViewById(R.id.true_button);
          mTrueButton.setOnClickListener {
              Toast.makeText(
                  this, R.string.correct_toast, Toast.LENGTH_SHORT
              ).show()
              checkAnswer(true)
          }
          mFalseButton = findViewById(R.id.false_button)
          mFalseButton.setOnClickListener {
              Toast.makeText(
                  this, R.string.incorrect_toast, Toast.LENGTH_SHORT
              ).show()
              checkAnswer(false)
          }
          mNextButton = findViewById(R.id.next_button)
      ...
      }
}
```

Программа DroidQuest снова готова к работе. Давайте запустим ее на

Библиографический список

1. Начало работы с Хранилищем очередей Azure с помощью .NET. URL: https://docs.microsoft.com/ru-ru/azure/storage/queues/storage-dotnet-how-to-use-queues?tabs=dotnet (Дата обращения 21.11.2021 г.)
2. Исакова С., Жемеров Д. Kotlin в действии / пер. с англ. Киселев А.Н. — М.: ДМК-Пресс, октябрь 2017 г., 402 с.
3. Скин Д., Гринхол Д. Kotlin. Программирование для профессионалов / пер. с англ. Киселев А.Н. — СПб.: Издательский дом «Питер», 2020 г., 464 с.
4. Официальная документация языка программирования Kotlin. URL: https://kotlinlang.ru/ (Дата обращения 21.11.2021 г.)

Практическая работа 8. Конец