УДК 004.77:004.424
DOI: 10.46548/21vek-2020-0951-0012

ВАРИАНТ АРХИТЕКТУРЫ ОБЛАЧНОЙ СИСТЕМЫ ДЛЯ ПОЛЬЗОВАТЕЛЬСКИХ ДАННЫХ

Пащенко Татьяна Юрьевна, кандидат экономических наук, доцент,
доцент кафедры «Менеджмент и экономическая безопасность»
Пензенский государственный университет
(440026, Россия, Пенза, улица Красная, 40, e-mail: tania.paschenko@gmail.com)
Мартышкин Алексей Иванович, кандидат технических наук, доцент,
доцент кафедры «Вычислительные машины и системы»
Лобов Роман Александрович, аспирант кафедры
«Вычислительные машины и системы»
Пензенский государственный технологический университет
(440039, Россия, Пенза, проезд Байдукова/ул. Гагарина, д. 1а/11,
e-mails: alexey314@yandex.ru, popovich058@yandex.ru)

Аннотация. В статье рассматривается архитектура облачной системы для хранения пользовательских данных. В настоящее время рынок облачных сервисов является одними из самых активно развивающихся направлений в IT-индустрии. Целью работы является исследование архитектурах подходов для построения облачных сервисов, обзор технологий с помощью которых может быть реализовано такое решение. Для достижения поставленной цели проведено изучение существующих решений, помогающих в разработке. Также в статье проводится сравнение технологий, с помощью которых возможно построение облачной системы с «нуля». В рамках исследования использованы современные технологии, инструменты и архитектурные подходы. Основным подходом к построению системы выбрана многоуровневая архитектура. Общим технологическим стеком прототипа выбрана программная платформа .NET и язык C#. Для разработки серверного компонента выбрана архитектурная парадигма MVC. Кроме серверного компонента, отвечающего за хранение файлов и других данных, спроектированы веб и настольные решения, использующиеся пользователем для работы. Система может быть использована несколькими пользователями и обладает минимальным набором необходимых для работы функций. Результатом исследований является прототип облачного сервиса, позволяющий хранить пользовательские данные. В заключении приведены основные результаты и выводы по проведенной работе.

Ключевые слова: облачная система, пользовательские данные, хранилище, технологический стек, архитектурная парадигма MVC, прототип облачного сервиса, платформа .NET.

A VARIANT OF THE CLOUD SYSTEM ARCHITECTURE FOR USER DATA

Pashchenko Tatyana Yuryevna, candidate of economic Sciences,
associate Professor of sub-department «Management and economic security»
Penza State University
(440026, Russia, Penza, Krasnaya Street, 40, e-mail: tania.paschenko@gmail.com)
Martyshkin Alexey Ivanovich, candidate of technical sciences, docent,
associate Professor of sub-department «Computers and systems»
Lobov Roman Aleksandrovich, postgraduate of sub-department
«Computers and systems»
Penza state technological University
(440039, Russia, Penza, Baydukov Proyezd / Gagarin Street, 1a/11, e-mails:
alexey314@yandex.ru, popovich058@yandex.ru)

Abstract. This article discusses the architecture of a cloud system for storing user data. Currently, the cloud services market is one of the most actively developing areas in the IT industry. The purpose of this work is to study the architecture of approaches for building cloud services, review the technologies that can be used to implement such a solution. To achieve this goal, we studied existing solutions that help in development. The article also compares the technologies that can be used to build a cloud system from scratch. The research uses modern technologies, tools, and architectural approaches. The main approach to building the system is a multi-level architecture. The software platform was chosen as the General technological stack of the prototype .NET and the C# language. The MVC architectural paradigm was chosen for the development of the server component. In addition to the server component responsible for storing files and other data, web and desktop solutions are designed that are used by the user for work. The system can be used by several users and has a minimal set of functions required for operation. The result of the research is a prototype of a cloud service that allows storing user data. In conclusion, the main results and conclusions of the work are presented.

Keywords: cloud system, user data, storage, technology stack, MVC architectural paradigm, cloud service prototype, .NET platform.

Введение. В последние годы развитие облачных сервисов носит взрывной характер, что связано с все более широким распространением сети Интернет. Появляются сервисы, позволяющие хранить пользовательский контент на удаленных серверах, что в свою очередь позволяет освободить место на накопителях, а также исключает возможность потери данных. Наиболее известными в России сервисами для хранения пользовательских данных являются «Яндекс Диск», «*Google* Диск» и др. В основном, перечисленные сервисы предоставляют малое количество дискового пространства для хранения данных, что ведет к лишним затратам в случае, если пользователь хочет хранить большое количество данных в облаке.

Целью работы является исследование архитектурных подходов для построения облачных сервисов, обзор технологий с помощью которых может быть реализовано такое решение. Для достижения цели проводится изучение существующих решений, помогающих в разработке. Приводится сравнение технологий, с помощью которых возможно построение облачной системы. Результатом исследования является прототип облачного сервиса, позволяющий хранить данные пользователя.

К сожалению, нет ничего бесплатного. Это касается и сферы *IT*, в которой понятие себестоимости играет не менее важную роль, чем в других областях. Также немаловажную роль сегодня играет конфиденциальность информации, как показывает практика, даже самая надежно защищенная информация может быть скомпрометирована. Можно сказать, что публичные облачные сервисы для хранения данных не очень подходят для хранения конфиденциальной информации. Сама концепция облачного хранилища данных удобна. Реальна ситуация, в которой небольшая компания, занимающаяся, скажем работой с фото, хочет централизованно хранить свой архив снимков. Логично, что сотрудниками удобнее работать через общую базу фотографий, при этом синхронизирующуюся между их компьютерами. Проблема построения приватного облака, в том числе выбора его архитектуры и технологического стека, является достаточно важной, так как может быть применима в реальной жизни и носит практический характер [1].

Материалы и результаты исследования. В настоящее время существует достаточно большое количество платформ и языков разработки веб-приложений. А также достаточно большое количество средств для разработки настольных приложений. Выделяют две большие группы – компилируемые и интерпретируемые [2, 3]. В случае с сервисом, работающим с большим количеством данных и большим количеством пользователей, компилируемые языки получаются преимущество, за счет уже скомпилированных в машинные коды исходников. Что позволяет им работать более производительно. Одной из наиболее популярных технологий разработки веб-приложений является *ASP.NET MVC* [4 – 7]. Это веб-фреймворк от компании *Microsoft*, который поддерживает программирование на языке *C#*, а также позволяет без труда использовать весь стек разработки от данной компании. В основе этого технологического стека лежит виртуальная машина *CLR*, преобразовывающая написанный программный код в промежуточный *IL*-язык, следующим шагом транслируемый в коды процессора [8]. В случае с настольным клиентом используем язык *C#* [9 – 12], платформу *.NET* и технологию *WPF* [13], которая позволяет строить настольные приложения для платформы *Windows*. Использование одной и той же платформы, позволит переиспользовать некоторые фрагменты программного кода (например, подсчет хэш-функции), что в свою очередь уменьшит время разработки прототипа решения [14, 15].

Сегодня одной из самых популярных моделей для построения приложений является трехуровневая архитектура – модель информационной системы, предполагающая, что в ней будет как минимум три слоя абстракций – клиент, сервер и СУБД. Клиентом в этом случае является компонент системы доступный пользователю системы. Сервером является модуль системы второго слоя и содержит основную часть бизнес-логики. Также является связующем звеном в цепи между сервером баз данных и клиентом. Сервер баз данных отвечает за хранение данных и является отдельным модулем. Этот слой обычно реализуют средствами СУБД. Только сервер приложений имеет доступ к серверу баз данных, доступ клиента полностью исключен.

Сервер приложений, также можно представить в виде слоев абстракций. Которые помогут уменьшить стоимость разработки и поддержки решения. А также повысить его стабильность. Условно, сервер приложений можно поделить на следующие слои:

1. *Presentation Layer (PL)* – уровень, с которым происходит взаимодействие пользователя.

2. *Business Logic Layer (BLL)* – включает компоненты, отвечающие за обработку данных от *DAL*.

3. *Data Access Layer (DAL)* – данный слой хранит в себе модель данных, которая описывает используемые сущности и предметную область.

Данное описание слоев носит лишь общий и абстрактный смысл. Так как количество уровней может быть больше трех. Например, у *DAL* может быть дополнительны уровень, отвечающий за кеширование. А у *BLL* дополнительный уровень, куда вынесены специальные вычисления. Общий принцип останется неизменным.

Архитектурный подход *MVC* является наиболее подходящим для написания систем в формате трехуровневой архитектуры. Концептуально, с помощью *MVC* части программы можно разделить на 3 логические и программные группы: модель; представление; контроллер. *MVC* идеально соответствует трехуровневой архитектуре, так как через нее легко описать уровни приложения.

Создание прототипа. На рисунке 1 приведена диаграмма вариантов использования для веб-интерфейса системы [16, 17]. Данная диаграмма описыва-

ет основные действия, которые пользователь может проводить с веб-интерфейсом, описывающая взаимодействие пользователя со списком файлов, путем его просмотра и скачки необходимых файлов. Кроме того, пользователь, при желании, может изменить пароль. Завершение работы системы также входит в сферу деятельности пользователя, так как он может разлогировать из приложения.

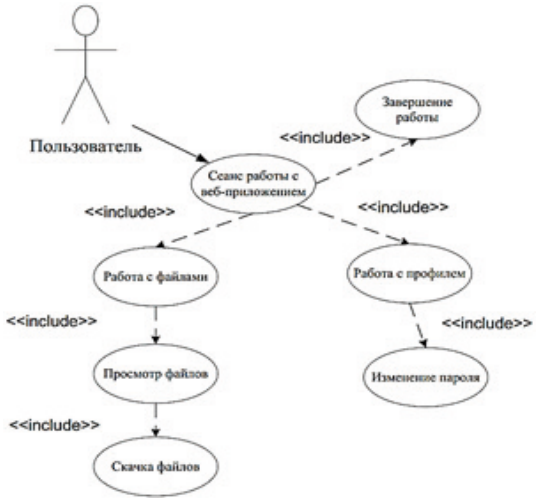


Рисунок 1 – Диаграмма вариантов использования веб-интерфейса

На рисунке 2 приведена диаграмма вариантов использования для настольной части проекта, описывающая основные действия, которые может проводить пользователь.

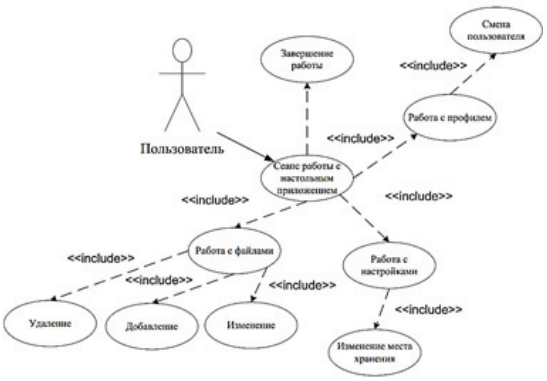


Рисунок 2 – Диаграмма вариантов использования настольного интерфейса

Реализация слоя доступа к данным. Используется паттерн разработки слоя доступа к данным «*Repository*» и «*IUnitOfWork*». На рисунке 3 показана первая итерация интерфейса базового репозитория.

```
using System.Collections.Generic;
using System.Threading.Tasks;

namespace FileStorage.Database.Interfaces
{
    public interface IGenericRepository<TEntity> where TEntity : class
    {
        Task<List<TEntity>> Get();
        Task<TEntity> GetById(int id);
        void Insert(TEntity item);
        void Remove(TEntity item);
    }
}
```

Рисунок 3 – Структура проекта с файлами для работы с базой данных

На интерфейс наложено ограничение, обусловленное спецификой языка *C#*, его использование возможно только с классами. Интерфейс обладает четырьмя методами. *Get* – забирает лист сущностей. *GetById* – забирает сущность из базы данных, при этом происходит поиск по идентификатору сущности. *Insert* – вставка данных сущности в таблицу в БД. *Remove* – удаление сущности из базы данных. На рисунке 4 изображена реализация базового интерфейса, ей стал класс *GenericeRepository*.

```
public abstract class GenericRepository<TEntity, TContext> : IGenericRepository<TEntity>
    where TEntity : class
    where TContext : DbContext
{
    private readonly DbSet<TEntity> _dbSet;
    protected TContext Context { get; }

    protected GenericRepository(TContext dbContext)...
    public async Task<List<TEntity>> Get()...
    public async Task<TEntity> GetById(int id)...
    public void Insert(TEntity item)...
    public void Remove(TEntity item)...
}
```

Рисунок 4 – Реализация репозитория

Из рисунка 4 видно, что базовый репозиторий сделан максимально шаблонным классом, т.е. предполагается его переиспользование. Также, оставлена возможность использовать его в рамках разных контекстов. Данные шаги позволят сделать этот слой гибким и переиспользуемым. Возможна ситуация, когда приложение будет использовать несколько контекстов, тогда не придется прибегать к копированию кода, а просто использовать существующий. На рисунке 5 изображена реализация данного интерфейса.

```
public class UserRepository : GenericRepository<User, FileStorageContext>, IUserRepository
{
    public UserRepository(FileStorageContext dbContext) : base(dbContext)
    {
    }

    public async Task<User> GetByLogin(string login)
    {
        return await Context.Users.FirstOrDefaultAsync(x => x.Login == login);
    }
}
```

Рисунок 5 – Реализация репозитория для User

Для реализации паттерна «Модуль работы» создан интерфейс *IUnitOfWork* и его наследник *UnitOfWork*, содержащий в себе вызов метода *SaveChanges* из контекста. На рисунке 6 предоставлена схема *IUnitOfWork* и *UnitOfWork*.

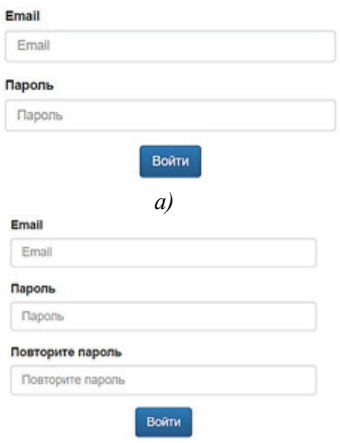
```
public interface IUnitOfWork
{
    Task SaveAsync();
}

public class UnitOfWork : IUnitOfWork
{
    private readonly DbContext _dbContext;

    public UnitOfWork(DbContext context)
    {
        _dbContext = context;
    }

    public async Task SaveAsync()
    {
        await _dbContext.SaveChangesAsync();
    }
}
```

Рисунок 6 – Реализация паттерна

Реализация авторизации и аутентификации. Выбор инструмента для механизма работы аутентификации и авторизации носит важную роль, так как с его помощью, пользователям предоставляется доступ к ресурсам. *Json Web Tokens (JWT)* – открытый индустриальный стандарт, описывающий возможность безопасной передачи служебной информации о правах, между двумя клиентами. Главное преимущество данной технологии, в том, что с ее помощью уменьшается количество запросов в базу данных, для проверки данных пользователя, а также его прав [18].

Реализация пользовательского интерфейса. Сегодня существует несколько фреймворков для построения *SPA*-приложений. Наиболее популярных из них – *Angular*, *React* и *VueJs*. Для построения интерфейса выбран фреймворк *React*, так как прототип решения не должен обладать богатой функциональностью. Предполагается наличие формы входа и регистрации, с помощью которой пользователь будет заходить в систему, а также страницы, содержащей список файлов, где нужный из которых сможет скачать пользователь. Для работы прототипу необходимо наличие четырех страниц в веб-версии. Это страницы авторизации, регистрации, небольшой личный кабинет, а также же список всех файлов пользователя с возможностью их скачивания.

Реализация веб-интерфейса. Ввиду того, что для прототипа нет необходимости в сложных настройках и конфигурациях, то используем прямое подключение скриптов через *CDN*. В данном случае, скрипты собраны и сформированы кем-то еще, а пользователи получают ссылки на них. В таком случае, в проект достаточно подключить эти ссылки через специальные тэги. С помощью этого шага уменьшается головная боль для разворачивания инфраструктуры и сборки. На рисунке 7а изображена страница входа, на которой пользователь вводит электронную почту и пароль. После чего, он авторизовывается в решении. На рисунке 7б изображена страница регистрации пользователя, отличается она от страницы авторизации лишь наличием дополнительно поля ввода для подтверждения пароля [19].



а)

б)

Рисунок 7 – Реализованная страница входа (а) и регистрации пользователя (б)

Реализовано приложение, постоянно мониторящее файлы и при необходимости отправляющее их на сервер. Для этого предусмотрено 3 окна:

1. Окно настроек – отвечает за установления папки, файлы в которой будут постоянно проверяться на изменения, а также за проверку ситуации, когда добавляются новые файлы.

2. Окно входа в приложение – данное окно отвечает за ввод электронной почты и пароля с помощью которых производится вход в систему. Также данное окно производит их сохранение

3. Контекстное меню –меню всплывающее окно в трее системы которое отвечает за показ окна настроек. Также с его помощью можно произвести разлогирование из программы и полное закрытие программы.

В рамках настольного приложения, предполагается постоянный мониторинг файлов, находящихся в папке, указанной в настройках настольного клиента. При этом прототип не будет поддерживать вложенные каталоги. Мониторинг будет реализовываться с помощью встроенных средств *.NET-Framework*. Данное средство мониторит события, происходящие в файловой системе. После этого будет происходить обновление или сохранение файла на сервере.

Вполне реальна ситуация, когда клиент с одним логином и паролем работает с нескольких компьютеров. Достаточно обычной ситуацией будет изменение или удаление файлов на одном из компьютеров. Допустим на первом произошло добавление и удаление файла, при это эти файлы должны исчезнуть и со второго. Чтобы это стало возможным необходимо сообщаться клиенту о событиях, происходящих на сервере. Наиболее популярными способами сделать это являются *long poll* (длинные запросы) и технология *WebSockets* [20]. Длинные запросы можно описать как очередь ожидающих запросов. Здесь запрос отправляется на сервер, при этом соединение с сервером не закрывается до того, момента, пока на сервере не отработает какое-либо событие, после отработки которого, оно отправляется как ответ на запрос. После получения ответа, клиент сразу отправляет новый ожидающий запрос.

При первом запуске программы предполагается загрузка всех недостающих файлов со стороны сервера. Для этого предполагается делать нескольких действий. Первым шагом будет получение списка всех файлов пользователя в виде *JSON*-файла. На рисунке 8 приведен пример данного файла.

Рисунок 8 – Пример списка файлов, полученных с сервера

Возможна ситуация, когда в папке уже есть некоторые файлы. Для этого необходимо сделать проверку соответствий пары «хэш-название», между полученными данными и файлами, находящимися в папке. Если возникнет ситуация, когда пары совпадают, необходимо просто добавить записи в локальную базу данных клиента. Если совпадений не найдено, тогда необходимо посчитать хэш файлов, присвоить им идентификаторы и отправить на сервер. Файлы, которые присутствуют на сервере, необходимо скачать на сторону клиента, при этом делать это необходимо в многопоточном режиме, что позволит загрузить большое количество файлов в относительное короткое время. Для этого серверу будут уходить запросы с идентификаторами файлов, после этого будет происходить их скачка. Следующим шагом станет выставление им имен и сохранение в файловой системе. После данных шагов, файлы в системе будут готовы к использованию.

Заключение. В ходе исследования спроектирован прототип облачного сервиса для хранения пользовательских данных. Реализовано приложение, мониторящее файлы и при необходимости отправляющее их на сервер. Кроме серверного компонента, отвечающего за хранение файлов и других данных, спроектированы веб и настольные решения. Возможно использование системы несколькими пользователями, она обладает минимальным набором необходимых для работы функций.

СПИСОК ЛИТЕРАТУРЫ:

1. Ананченко И.В., Гришечко Е.К. Разработка клиент-серверных приложений, работающих в облачных средах. Успехи современной науки и образования. – 2016. – Т.2. – № 4. – С. 124-126.

2. Thinh L. V., Middleware to Integrate Mobile Devices, Sensors and Cloud Computing/ Bouzefrane S, Attar A. // ScienceDirect. – 2015. – №52. – С.234

3. Фаулер М. Шаблоны корпоративных приложений – 3-е изд. – М.: Вильямс, 2010. – 544 с.

4. Симан М. Внедрение зависимостей в .NET – 1-е изд. – СПб.: Питер, 2012. – 204 с.

5. How to create simple blog using ASP.NET MVC [Электронный ресурс]. URL: http://www.prideparrot.com/blog/archive/2012/12/how_to_create_a_simple_blog_part1 (дата обращения: 14.10.2020).

6. How to create a blog using ASP.NET MVC [Электронный ресурс]. URL: https://stackoverflow.com/questions/4254822/how-to-create-a-blog-in-asp-net-and-not-asp-net-mvc (дата обращения: 14.10.2020).

7. Фриман М. ASP.NET MVC 5 с примерами на C# 5.0 для профессионалов – 2-е изд. – М.: Вильямс, 2013. – 351 с.

8. Asankhaya Sharma. How to choose the right Technology Stack for your Startup? [Электронный ресурс]. URL: https://www.linkedin.com/pulse/20140404155507-16837833-how-to-choose-the-right-technology-stack-for-your-startup (дата обращения: 14.10.2020).

9. Маклин Хол Г. Адаптивный код на C#. Проектирование классов и интерфейсов, шаблоны и принципы SOLID – СПб.: Питер, 2015. – 432 с.

10. Рихтер Д. CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C# – СПб.: Питер, 2015. – 432 с.

11. Шилдт Г. Полное руководство C# 4.0 – 3-е изд. – М.: Вильямс, 2011. – 487 с.

12. Троелсен Э. C# и платформа .Net. Библиотека программиста. – СПб.: Питер, 2002. – 800 с.

13. МакДональд М. WPF. Windows Presentation Foundation в.NET 4.5 с примерами на C# 5.0 для профессионалов / Мэтью Мак-Дональд; [пер. с англ. Ю. Н. Артеменко]. – 4-е изд. – Москва [и др.]: Вильямс, 2013. – 1018 с.

14. Архитектура REST [Электронный ресурс]. URL: https://habrahabr.ru/post/38730/ (дата обращения: 14.10.2020).

15. Зачем нуженRest? [Электронный ресурс]. URL: http://eax.me/rest/ (дата обращения: 14.10.2020).

16. How to choose the right Technology for your Web Application? [Электронный ресурс]. URL: https://medium.com/the-resonant-web/best-tech-stacks-for-web-development-e35229e0b473 (дата обращения: 14.10.2020).

17. Guthrie S., Somasegar S. Microsoft Application Architecture Guide, 2nd Edition 2009

18. BigTable: A Disturbed Storage System for Structured Data [Электронный ресурс]. URL: http://static.googleusercontent.com/media/research.google.com/ru//archive/bigtable-osdi06.pdf (дата обращения: 14.10.2020).

19. Ричардсон Л. RESTful Web APIs – O'REILLY, 2013. – 193с

20. Ричардсон Л. RESTful WebServices Cookbook: Solutions for Improving Scaalability and Simplicity – O'REILLY, 2010. – 320 с.

Статья поступила в редакцию 05.11.2020
Статья принята к публикации 11.12.2020