Министерство образования Республики Беларусь

Учреждение образования

БелорусскиЙ государственный университет

информатики и рАдиоэлектроники

Факультет Компьютерных систем и сетей

Кафедра Информатики

|  |
| --- |
| *К защите допустить:* |
| Заведующий кафедрой информатики |
| \_\_\_\_\_\_\_\_\_\_\_\_\_ Н.А. Волорова |

Пояснительная записка

к дипломному проекту

на тему:

**Портал для проведения фэнтези спорт турниров**

БГУИР ДП 1-40 01 03 00 001 ПЗ

|  |  |  |
| --- | --- | --- |
| Студент |  | Д. А. Абрамов |
| Руководитель |  | Е. Л. Богушевский |
| Консультанты: |  |  |
| *от кафедры информатики* |  | Е. Л. Богушевский |
| *по экономической части* |  | В. А. Палицын |
| Нормоконтролер |  | В. В. Шиманский |
|  |  |  |
| Рецензент |  | А. Д. Булова |

Минск 2017

**РЕФЕРАТ**

РАЗРАБОТКА ПОРТАЛ ДЛЯ ПРОВЕДЕНИЯ ФЭНТЕЗИ СПОРТ ТУРНИРОВ: дипломный проект / Д. А. Абрамов. – Минск: БГУИР, 2017, - п.з. – 87 с., чертежей (плакатов) – 6 л. формата А1.

Ключевые слова: ВЕБ-ПРИЛОЖЕНИЕ, ФЭНТЕЗИ СПОРТ, C#, ASP.NET CORE, ANGULAR, БАЗА ДАННЫХ.

Объектом исследования и разработки является приложение для проведения фэнтези спорт турниров.

Целью проекта является разработка программного комплекса, который является веб-приложением для организации фэнтези спорт турниров и обработки информации в процессе проведения турниров.

При разработке и внедрении приложения использовался такой стек технологий как C#, ASP.NET CORE, HTML5, CSS3, TypeScript, Angular.

В первом разделе данной работы ставится задача дипломного проектирования, устанавливается предметная область разработки, выдвигаются общие требования к созданию приложения, а также определяются перспективы развития первой версии продукта.

Второй раздел содержит общие сведения о применяемых при разработке приложения технологиях и их особенностях.

Третий раздел посвящен проектированию архитектуры приложения.

В четвертом разделе описана реализация программного продукта.

Пятый раздел содержит информацию о тестировании приложения.

В шестом разделе приведено технико-экономическое обоснование эффективности разработки и внедрения программного продукта.

В разделе заключение содержатся краткие выводы по дипломному проекту.

Дипломный проект является завершенным, поставленная задача решена в полной мере, присутствует возможность дальнейшего развития приложения и увеличение его функционала.

Министерство образования Республики Беларусь

Учреждение образования   
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет КСиС Кафедра Информатики

Специальность 1-40 01 03 Специализация 00

УТВЕРЖДАЮ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Зав. кафедрой

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_ 2017 г.

**ЗАДАНИЕ**

**по дипломному проекту (работе) студента**

Абрамова Дмитрия Александровича

(фамилия, имя, отчество)

1 Тема проекта (работы): «Портал для проведения фэнтези спорт турниров»

утверждена приказом по университету от 28.02.2017 г. № 391‑с

2 Срок сдачи студентом законченного проекта (работы): 2017 года

3 Исходные данные к проекту (работе): Тип операционной системы: Microsoft Windows; Язык программирования – C#, ASP.NET, MS SQL; Назначение разработки:

Создание портала для проведения фэнтези спорт турниров.

4 Содержание пояснительной записки (перечень подлежащих разработке вопросов):

Перечень условных обозначений, символов и терминов

Введение

1. Анализ прототипов и формирование требований к программному средству

2. Обзор используемых технологий

3. Проектирование программного средства

4. Создание программного средства

5. Тестирование программного средства

6. Технико-экономическое обоснование разработки программного средства

Заключение

Список используемых источников

Приложение А Исходный текст программы

5 Перечень графического материала (с точным указанием обязательных чертежей):

ГУИР.400103.001 ПД1 Диаграмма классов (контроллеров). Чертеж. Формат А1. лист 1.

ГУИР.400103.001 ПД2 Диаграмма классов (моделей). Чертеж. Формат А1. лист 1.

ГУИР.400103.001 ПД3 Диаграмма компонентов (front-end). Чертеж. Формат А1. лист 1.

ГУИР.400103.001 ПЛ1 Диаграмма базы данных. Плакат. Формат А1. лист 1.

ГУИР.400103.001 ПЛ2 Диаграмма вариантов использования. Плакат. Формат А1. лист 1.

ГУИР.400103.001 ПЛ3 Диаграмма деятельности. Плакат. Формат А1. лист 1.

6 Содержание задания по технико-экономическому обоснованию:

Расчет экономической эффективности от внедрения «Портала для проведения фэнтези спорт турниров»

Задание выдал: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ / В.А. Палицын /

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № п/п | Наименование этапов дипломного проекта (работы) | Объем этапа, % | Срок выполнения этапов | Примечание |
| 1 | Анализ предметной области, разработка технического задания | 15 – 20 | 03.02–16.02 |  |
| 2 | Разработка функциональных требований, проектирование архитектуры программы | 20 – 15 | 18.02–07.03 |  |
| 3 | Разработка схемы программы, алгоритмов, схемы данных | 20 – 15 | 11.03–22.03 |  |
| 4 | Разработка программного средства | 15 – 20 | 25.03–26.04 |  |
| 5 | Тестирование и отладка | 10 | 29.04–10.05 |  |
| 6 | Оформление пояснительной записки и графического материала | 20 | 13.05–31.05 |  |

КАЛЕНДАРНЫЙ ПЛАН

Дата выдачи задания \_\_\_\_\_\_\_\_\_ Руководитель \_\_\_\_\_\_\_ / Е.Л. Богушевский /

Задание принял к исполнению \_\_\_\_\_\_\_ / Д. А. Абрамов /

**СОДЕРЖАНИЕ**

[ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ 6](#_Toc483586889)

[ВВЕДЕНИЕ 7](#_Toc483586890)

[1 АНАЛИЗ ПРОТОТИПОВ И ФОРМИРОВАНИЕ ТРЕБОВАНИЙ К ПРОГРАММНОМУ СРЕДСТВУ 9](#_Toc483586891)

[1.1 Постановка задачи 9](#_Toc483586892)

[1.2 Обзор существующих аналогов 9](#_Toc483586895)

[1.2.1 DraftKing.com 9](#_Toc483586896)

[1.2.2 FanDuel.com 10](#_Toc483586897)

[1.2.3 Fanaments.com 12](#_Toc483586898)

[1.2.4 DraftOn.com 13](#_Toc483586899)

[1.2.5 Sports.ru/fantasy/ 14](#_Toc483586907)

[1.2.6 Redstarfantasy.ru 15](#_Toc483586908)

[1.3 Вывод 16](#_Toc483586909)

[2 ОБЗОР ИМПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ 17](#_Toc483586910)

[2.1 Программная платформа Microsoft .NET 17](#_Toc483586911)

[2.2 Язык программирования C# 19](#_Toc483586912)

[2.3 ASP.NET Core 23](#_Toc483586913)

[2.4 Angular 24](#_Toc483586914)

[2.5 TypeScript 25](#_Toc483586915)

[2.6 Entity Framework Core 25](#_Toc483586916)

[2.7 MS SQL Server 2017 27](#_Toc483586917)

[2.8 Visual Studio 2017 30](#_Toc483586924)

[3 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 32](#_Toc483586925)

[3.1 Слой представления 35](#_Toc483586926)

[3.2 Серверный слой 38](#_Toc483586927)

[3.3 Слой сервисов 41](#_Toc483586928)

[3.4 Слой доступа к данным 43](#_Toc483586929)

[4 СОЗДАНИЕ ПРОГРАММНОГО СРЕДСТВА 46](#_Toc483586930)

[5 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 56](#_Toc483586931)

[6 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗАРБОТКИ ПРОГРАММНОГО СРЕДСТВА 59](#_Toc483586932)

[6.1 Характеристика программного средства портала для проведения фэнтези спорт турниров 59](#_Toc483586933)

[6.2 Расчет себестоимости и отпускной цены 59](#_Toc483586934)

[ЗАКЛЮЧЕНИЕ 72](#_Toc483586937)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 73](#_Toc483586938)

[ПРИЛОЖЕНИЕ А 75](#_Toc483586939)

**ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ**

В настоящей пояснительной записке применяются следующие определения и сокращения.

API – Application Programming Interface – интерфейс программирования приложений.

DI – Dependency injection – инъекция зависимости

DOM – Document Object Model – объектная модель документа.

HTTP – HyperText Transfer Protocol – протокол передачи гипертекста.

JSON – JavaScript Object Notation – текстовый формат обмена данными, основанный на языке JavaScript.

LINQ – Language Integrated Query – интегрированные в язык запросы.

LоС – Lines of Code – количество строк кода.

MSIL – Microsoft Intermediate Language – промежуточный язык для платформы Microsoft .NET.

MVC – Model-View-Controller – модель-представление-контроллер.

MVW – Model-View-Whatever – модель-вид-что угодно.

REST – Representation State Transfer – передача состояния представления.

SPA – Single Page Application – одностраничное приложение

SQL – Structured Query Language – язык структурированных запросов.

URI – Uniform Resource Identifier – унифицированный идентификатор ресурса.

URL – Uniform Resource Locator – единый указатель ресурсов.

WWW – World Wide Web – всемирная паутина

XML – eXtensible Markup Language – расширяемый язык разметки.

БД – база данных.

ООП – объектно-ориентированное программирование.

СУБД – система управления базами данных.

# ВВЕДЕНИЕ

С самых давних времен спорт был окружен людьми, спорящими о результатах спортивных состязаний. Так повелось ещё со времён Римской империи, где делались спортивные ставки. Спустя некоторое время к ипподрому прибавились петушиные бои - развлечение для плебеев. Но только лишь в XIX веке самые разные виды спорта начали становиться предметом ставок. Вскоре практика показала, что ставить можно на всё, что бегает, прыгает, ездит и летает. Так на смену тотализаторам пришли букмекерские конторы.

Со временем любители спортивных состязаний жаждали более серьезного вызова. Так появился фэнтези спорт. Фэнтези спорт - это виртуальная игра, в которых участники составляют команду из реальных игроков того или иного чемпионата и в зависимости от того, как они выступают, получают очки. Все началось с того, что поклонники спортивных состязаний соревновались друг с другом и подсчитывали очки в конце сезона. Главный интерес заключается в том, чтобы правильно предсказывать результаты реальных спортсменов: в нужный момент «выпускать» тех, кто проводит хорошие отрезки, и сажать в запас тех, кто играет плохо. Фэнтези турниры есть для всех популярных видов спорта: можно собирать свою команду что в футбол, что в хоккей, что в баскетбол, что в бейсбол.

С распространением интернета популярность фэнтези спорт стремительно взлетела. Сейчас фэнтэзи спорт не просто игра для друзей, а индустрия с миллиардными оборотами. Правила тоже изменились: крупные игроки на этом рынке теперь предлагают игры не только длиной в целый сезон, а недельные турниры и даже дневные. Каждый игрок платит некую сумму за вход и в зависимости от типа турнира может выиграть до нескольких миллионов долларов. По данным Eilers Research, в год на вход в такие турниры игроки тратят 2,6 миллиарда долларов.

Стоит отметить географию распространения игры. Фэнтези-спорт крайне популярен в США и Канаде. В 2014 году на североамериканском континенте в него играл 41 миллион человек (для сравнения, в 2009 — 27 миллионов). На рынке в США работают две главные компании: FanDuel и DraftKings. Обе они привлекли сотни миллионов долларов инвестиций. В числе прочих FanDuel дали денег NBC и Google. Инвестором DraftKings является телеканал Fox и национальные спортивные лиги. При этом в Европе фэнтези спорт только начинает набирать популярность. Площадок проведения фэнтези спорт турниров сравнимых североамериканскими турниров на европейском рынке нет. Русскоязычные поклонники фэнтези спорта испытывают затруднения в выборе площадки из-за отсутствия языка и ограниченной поддержки видов спорта и лиг. Несмотря на это среди русскоязычных любителей спорта игра очень популярна.

В настоящем документе описаны анализ, проектирование, прототипирование и разработку портала проведения фэнтези спорт турниров.

Основными целями приложения являются предоставление площадки проведения фэнтези спорт турниров с возможностью добавления новых видов спортивных состязаний и лиг.

**1** **АНАЛИЗ ПРОТОТИПОВ И ФОРМИРОВАНИЕ ТРЕБОВАНИЙ К ПРОГРАММНОМУ СРЕДСТВУ**

* 1. **Постановка задачи**

В рамках дипломного проекта поставлена цель разработать веб-платформы с использованием технологии ASP.NET Core, предназначенной для проведения фэнтези спорт турниров. Минимальными требованиями к реализации программного продукта являются:

- реализовать удобную систему пользовательской функциональности, авторизацию, регистрацию, роли и группы пользователей, права доступа, управление доступом для администратора;

- обеспечить гибкость и расширяемость приложения;

- реализовать систему создания турниров с автоматических подсчетом очков по предоставленным статистическим данным;

- реализовать добавление администрацией новых видов спортивных состязаний и лиг.

Для достижения поставленной цели необходимо решить следующие задачи:

* изучить существующие функциональные аналоги площадок для проведения фэнтези спорт турниров;
* на основании произведенного анализа выбрать и изучить необходимые технологии для реализации дипломного проекта;
* спроектировать гибкое архитектурное решение платформы для разработки типовых сайтов.



## Обзор существующих аналогов

**1.2.1** DraftKing.com

DraftKings – одна из крупнейших дэйли фентези спорт площадок. Компания была основана в 2012 году в Бостоне Джейсоном Робинсом (CEO), Полом Либерманом (директор по маркетингу) и Мэттом Калишом (директор по аналитике). Еще в период своего становления компания выкупила двух своих конкурентов, DraftStreet и StarsStreet. Основатели DraftKings сумели собрать большой объем инвестиций, благодаря чему сайт получил мощный толчок в развитии. Крупный финансовый оборот позволяет компании регулярно проводить бесплатные соревнования с денежными призами и предоставлять щедрые бонусы на первый депозит.

На текущий момент игра на деньги в DraftKings доступна только пользователям из Штатов, Канады и Великобритании. Тем не менее, пользователи из других регионов имеют возможность принимать участие во фрироллах, то есть бесплатных турнирах. DraftKings отличается большим разнообразием видов спорта – 8. Благодаря такому решению количество игроков, отдающих свое предпочтение DraftKings, растет с каждым днем, и проект продолжает развиваться. Призовой фонд в некоторых соревнованиях может превышать миллион долларов США. Интерфейс площадки представлен на рисунке 1.1.

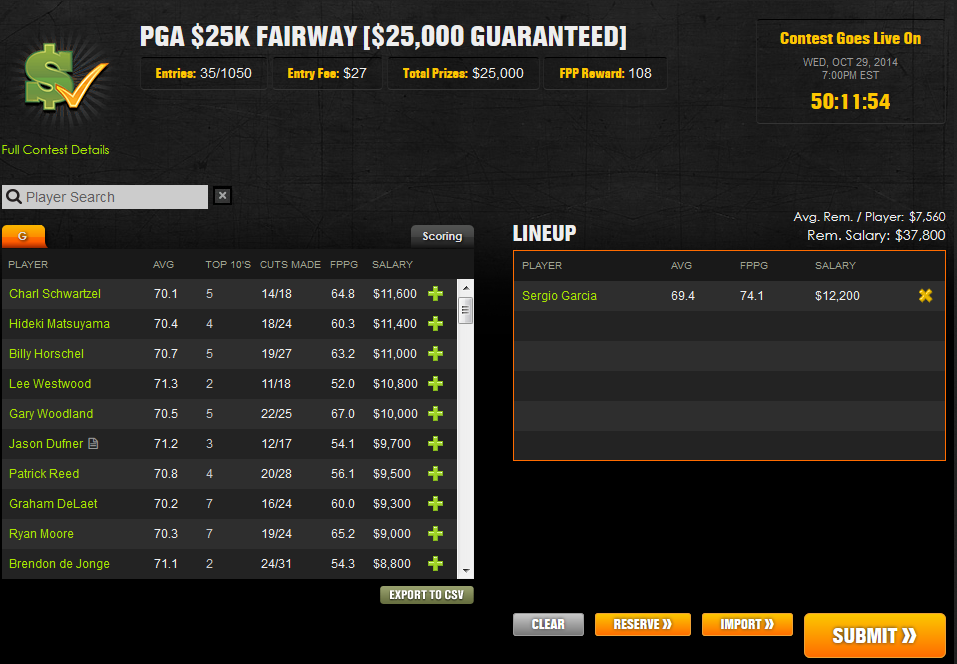


Рисунок 1.1 – DraftKings.com, пример реализации набора команды

Плюсами DraftKings являются:

- крупнейшая платформа;

- игра на деньги;

- ежедневные фрироллы;

- 8 видов спорта;

- 4 формата турниров;

- наличие британской лицензии.

Минусы платформы:

- региональные ограничения по участию;

- отсутствие сезонных турниров.

**1.2.2** FanDuel.com

FanDuel – крупнейшая площадка дэйли фентези спорта, основанный в 2009 году в США. В период создания проекта онлайн фэнтези переживал период становления и не был широко распространен. Изначально сайт развивался только на территории США и Канады, но не так давно генеральный директор FanDuel Найджел Эклс отметил, что пора переходить на новые рынки.

За восемь лет работы FanDuel приобрел большую популярность. Больше миллиона игроков отдают свое предпочтение площадке данного букмекера, а финансовый оборот превышает сумму в 300 миллионов долларов. Победители получают денежные призы, которые могут превышать сумму в сто тысяч долларов США. В настоящее время принять участие в игры могут только представители США и Канады. Для игры доступны хоккей, баскетбол, американский футбол, бейсбол и даже гольф. Интерфейс представлен на рисунке 1.2.

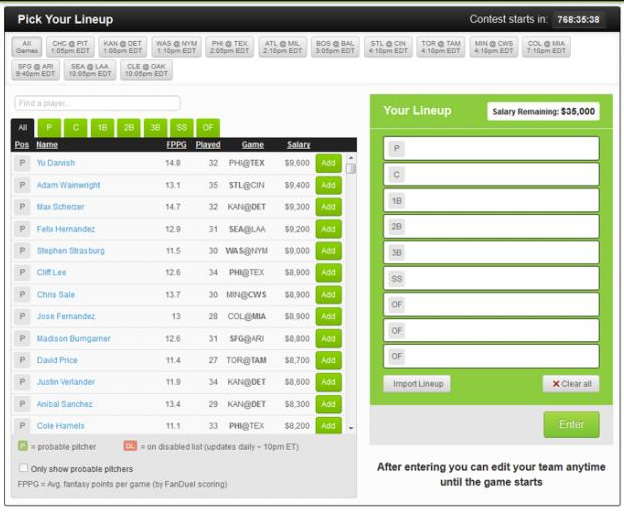


Рисунок 1.2 – FanDuel.com, пример реализации набора команды

Таким образом платформа обладает следующими плюсами:

- крупнейшая платформа;

- есть игра на деньги;

- 5 различных видов спорта;

- наличие британской лицензии;

- бесплатные соревнования.

Минусы:

- ограничения по регионам;

- отсутствие сезонных турниров.

**1.2.3** Fanaments.com

Fanaments - одна из первых площадок дэйли фэнтези в Европе, которая начала принимать игроков со всего мира. Валюта площадки – евро. Особенностью площадки является очень подробный скорринг. Каждую неделю проводятся фриролы с денежным вознаграждением. Помимо дэйли фэнтези на площадке представлен еще и пикэм – формат игры, в котором участники пытаются предугадать, с каким исходом и разницей мячей завершится несколько заданных игр.

На Fanaments существует программа вознаграждений. За каждый 1 евро, потраченный в турнирах, пользователи получают 1 EP (очки опыта) и 1 FP (очки Фанаментс). EP помогают менеджеру получить более выcокий статус, который сказывается на том, как много FP он будет получать. На FP можно приобрести различные сувенирные товары и билеты на турниры в магазине Fanaments. Интерфейс представлен на рисунке 1.3.

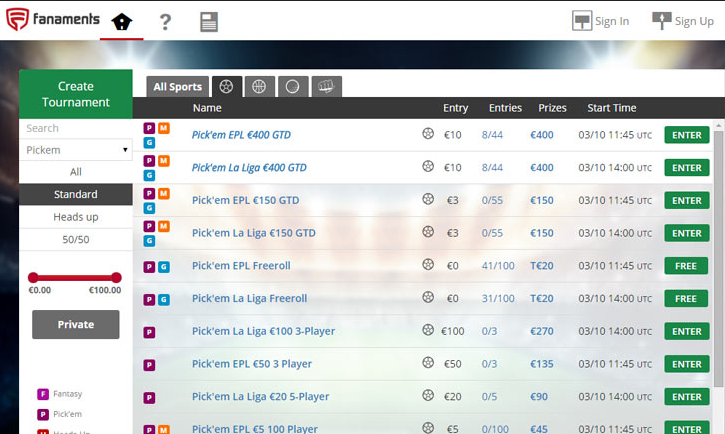


Рисунок 1.3 – Fanaments.com, пример реализации лобби

Плюсы Fanaments:

- доступен для игроков из России и стран СНГ;

- игра на деньги;

- наличие лицензии;

- 4 вида спорта;

- фрироллы;

- 5 форматoв турниров;

Минусы Fanaments:

- отсутствие сезонных турниров;

- малое количество лиг.

**1.2.4** DraftOn.com

DraftOn, недавно завершивший бета-тестирование — один из новых игроков на этом рынке, но, возможно, самый заметный.

Проект DraftOn.com делает упор исключительно на русскоязычную аудиторию и за 2016 год уже успел привлечь к дэйли фэнтези спорту почти 9000 активных пользователей. Интерфейс представлен на рисунке 1.4.

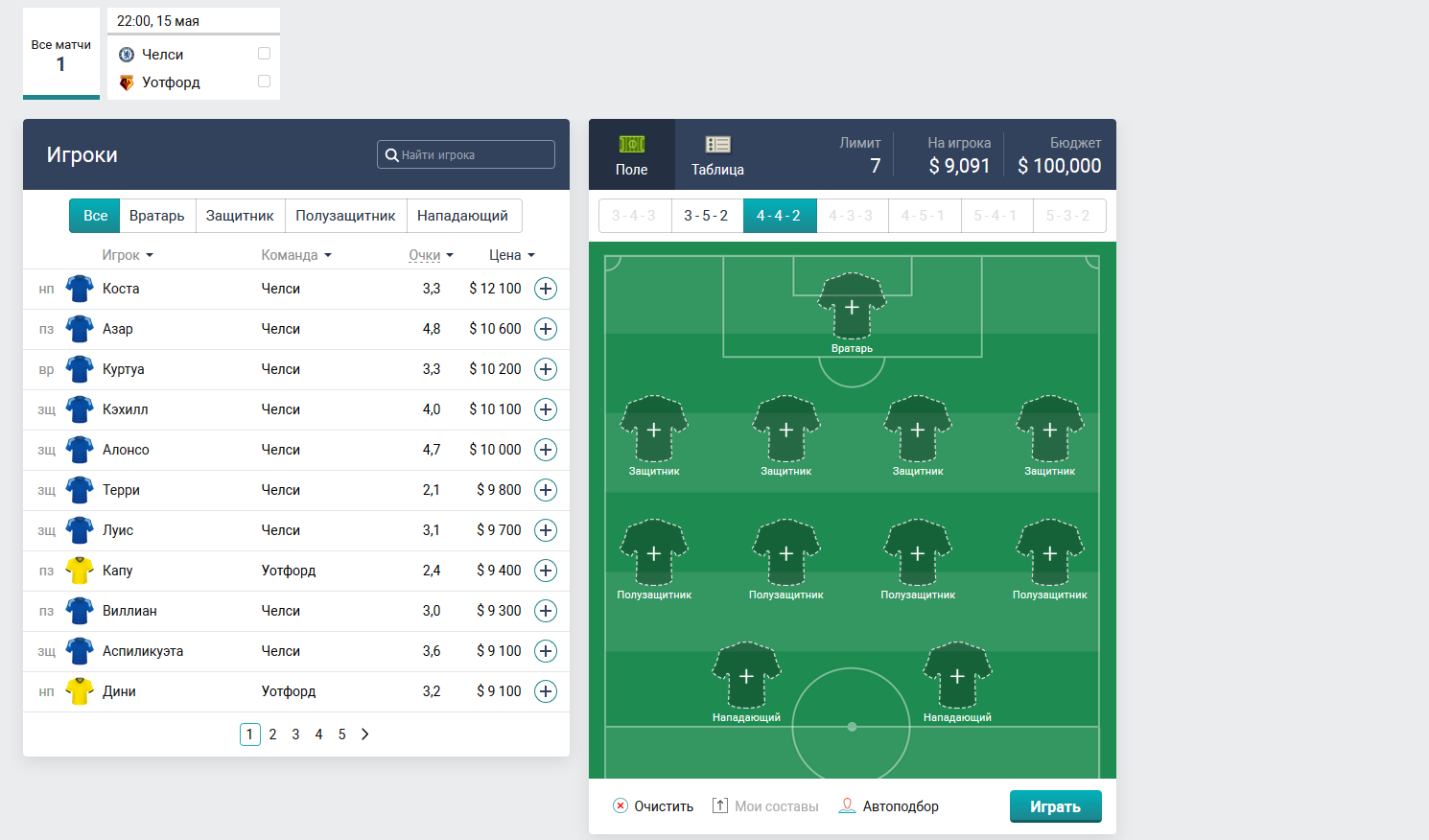


Рисунок 1.4 – DraftOn.com, пример реализации набора команды

DraftOn являются первопроходцами в России этом плане.

Исключительными особенностями сайта являются игра на российские рубли и система статусов.

Главной особенностью DraftOn является проведение игры на условные деньги (ДрафтОны), которые можно обменять на российские рубли по курсу 1:1. Вкупе с системой статусов, которая реализована на сайте вместо привычных вступительных взносов, это позволяет утверждать, что дэйли фэнтези спорт на DraftOn не является азартной игрой. Поэтому проект планирует действовать в рамках закона без получения специальных лицензий.

Площадка ориентирована на футбол. Игроки могут участвовать в соревнованиях английской, итальянской, немецкой, французской, испанской и российской лиг или же попробовать свои силы в Лиге чемпионов или Лиге Европы. На площадке уже появился хоккей, но на данный момент заметно уступает основному виду спорта.

Премиум статусы на DraftOn позволяют вступить в конкурс соответствующей категории и дают определённые преимущества. Каждый статус действует только 10 дней, по окончании которых он обнуляется, если не был использован для участия в турнире.

Как говорилось выше, помимо возможности принять участие в соответствующем турнире, каждый статус дает и определенные преимущества:

- Стандарт. Неограниченное участие в стандартных турнирах. Расстановка игроков: 4-4-2.

- Железо. Все доступное статусу стандарт. Дополнительно виды расстановки 4-4-2 и 3-5-2.

- Медь. Все преимущества предыдущих статусов. Дополнительно возможно именовать команды, обрамлять аватар, новые расстановки 4-3-3 и 3-5-2.

- Бронза. Преимущества предыдущих статусов. Менеджеру доступны новые расстановки 3-4-3 и 4-5-1.

- Серебро. Преимущества предыдущих статусов. Менеджеру доступна новая расстановка 5-4-1.

- Золото. Преимущества предыдущих статусов. Менеджеру доступна новая расстановка 5-3-2.

Плюсы DraftOn:

- восемь футбольных лиг и еврокубков;

- выигрыши без вложения средств;

Минусы платформы:

- недоступность некоторых функций;

- нет игры на деньги;

- лишь один формат турнира;

- мало видо спорта;

- нет возможности создавать турниры самостоятельно.

**1.2.5** Sports.ru/fantasy/

Sports.ru fantasy долгие годы является самой популярной площадкой для любителей фэнтези спорта в России, да и вообще в постсоветском пространстве. Фэнтези спортс ру впервые был запущен в 2009 году. Тогда это был просто текстовой интерфейс, а очки считались вручную.

На площадке представлен только сезонный фэнтези. Турниров с денежным вознаграждением нет. Однако в некоторых случаях вручаются спонсорские призы за первые места в итоговой таблице.

Фэнтези футбол на sports.ru выделяет особо. Он представлен тринадцатью лигами. Такое количество лиг редко встречается даже на крупнейших площадках проведения фэнтези спорт турниров. Самые популярные из них это Премьер Лига России и Премьер Лига Англии, насчитывающие около 20 тысяч участников каждая. Помимо этого, представлены такие виды спорта как хоккей, баскетбол, биатлон.

Плюсы площадки:

- большое количество футбольных лиг

Минусы платформы:

- нет игры на деньги;

- нет дейли турниров;

- нет возможности создавать турниры самостоятельно.



### Redstarfantasy.ru

Star Fantasy - площадка компания Red Star. Компания RedStar начала свою деятельность в 2005 году, открыв покер рум, а затем и казино, которые и по сей день широко популярны в рунете. В дальнейшем было принято решение охватить рынок фэнтези спорта. Сайт представлен на 3 языках: русском, английском и китайском. Как и остальные продукты компании, Red Star Fantasy в основном рассчитан на русскоязычную аудиторию, а наличие английского и китайского языков позволяет привлекать игроков не только из западных стран, но и из стран Азии. Еще одной интересной особенностью сайта является то, что пользователи, зарегистрировавшиеся для игры в фэнтези спорт, могут с того же счета играть в покер или казино. Интерфейс представлен на рисунке 1.5.

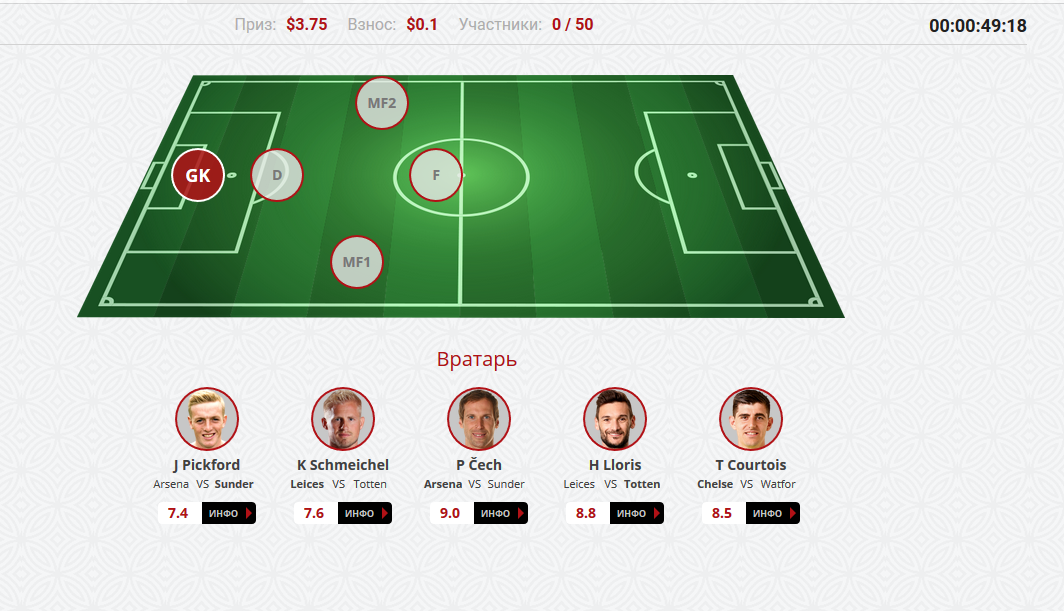


Рисунок 1.5 – Redstarfantasy.ru, пример реализации набора команды

Red Star Fantasy предоставляет своим игрокам широкий выбор лиг, так что поклонники различных видов спорта смогут найти что-то интересное для себя. Стоит отметить, что компания Red Star существует на рынке уже давно и за это время успела завоевать доверие игроков.

Плюсы проекта:

- доступность для игроков из России и СНГ;

- игра на деньги;

- бесплатные турниры;

- наличие лицензии;

Минусы:

- небольшое количество игроков;

- отсутствуют сезонные турниры;

- нельзя создавать собственные турниры.

## Вывод

Рассмотрев существующие аналоги, можно сказать, что самые крупные и интересные площадки охватывают североамериканский рынок, при этом они не торопятся выходить на европейский и азиатские рынки. Существующие площадки в Европе и Азии заметно уступают, в особенности разнообразием видов спорта. Некоторые площадки компенсируют это разнообразием лиг для представленных видов спорта, некоторые предпочитают сосредоточиться на малом количестве видов спорта и лиг, но сделать игровой процесс максимально разнообразным за счет оригинальных форм проведения турниров. Большинство площадок, позволяющих играть на деньги, также предлагают новым пользователям бонусы при первом пополнении депозита.

Выбор площадок проведения фэнтези спорт соревнований у русскоязычных пользователей невелик, так как многие площадки имеют региональные ограничения. Большинство русскоязычных пользователей попадают под них. Среди оставшихся также есть проблемы. Наиболее характерные из них:

- отсутствие русского языка;

- на площадках зачастую представлены либо дейли турниры, либо сезонные;

- скудный набор видов спорта и лиг.

Наибольшей популярностью пользуются дэйли спорт турниры. Это объясняется тем, что менеджерам не надо ждать до конца спортивного соревнования, чтобы узнать кто победил. Многие забрасывают играть посреди сезонного турнира как раз из-за длительности. Сегодня уже достаточно редко встречаются площадки проводящие сезонные турниры. Последние проводят чаще всего на официальных сайтах соответствующих лиг. Это привело к тому что энтузиасты длительных турниров вернулись к истокам – проведению турниров на форумах с ручным подсчетом очков.

# [2 ОБЗОР ИМПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ](#_Toc452938679)

## 2.1 [Программная платформа Microsoft .NET](#_Toc452938677)

В 1999 году в компании Microsoft была начана работа над новой программной платформой. Во многом необходимость разработки новой платформы была продиктована необходимостью представить свой ответ набиравшей популярность платформе Java от Sun Microsystems. В результате в 2002 году была выпущена программная платформа Microsoft .NET. С тех пор развитие не прекращалось и на данный момент номер последней версии .NET Framework 4.7.

При разработке .NET Framework учитывались следующие цели [1]:

- обеспечение согласованной объектно-ориентированной среды программирования для локального сохранения и выполнения объектного кода, для локального выполнения кода, распределенного в Интернете, либо для удаленного выполнения;

- обеспечение среды выполнения кода, минимизирующей конфликты при развертывании программного обеспечения и управлении версиями;

- обеспечение среды выполнения кода, гарантирующей безопасное выполнение кода, включая код, созданный неизвестным или не полностью доверенным сторонним изготовителем;

- обеспечение среды выполнения кода, исключающей проблемы с производительностью сред выполнения сценариев или интерпретируемого кода;

- обеспечение единых принципов работы разработчиков для разных типов приложений, таких как приложения Windows и веб-приложения;

- разработка взаимодействия на основе промышленных стандартов, которое обеспечит интеграцию кода платформы .NET с любым другим кодом.

Среди особенностей платформы .NET следует выделить наличие промежуточного языка выполнения (Common Intermediate Language, CIL). Все компиляторы, поддерживающие платформу .NET, должны транслировать код с языков высокого уровня платформы .NET на язык CIL.

По синтаксису и мнемонике язык CIL напоминает язык ассемблера. Его можно рассматривать как ассемблер виртуальной машины .NET. В то же время язык CIL содержит некоторые достаточно высокоуровневые конструкции, повышающие его уровень по сравнению с ассемблером для любой реально существующей машины, и писать код непосредственно на CIL легче, чем на ассемблере для реальных машин. Поэтому CIL можно рассматривать как своеобразный высокоуровневый ассемблер [2].

При запуске на выполнение приложения происходит JIT-компиляция (Just-In-Time) в машинный код, который затем выполняется. Основная цель использования JIT - достичь и превзойти производительность статической компиляции, сохраняя при этом преимущества динамической компиляции [3]:

- большинство тяжеловесных операций, таких как парсинг исходного кода и выполнение базовых оптимизаций, происходит во время компиляции (до развёртывания), в то время как компиляция в машинный код из байт-кода происходит быстрее, чем из исходного кода;

- байт-код более переносим (в отличие от машинного кода);

- среда может контролировать выполнение байт-кода после компиляции, поэтому приложение может быть запущено в песочнице.

- компиляторы из байт-кода в машинный код легче в реализации, так как большинство работы по оптимизации уже было проделано компилятором.

JIT, как правило, эффективней, чем интерпретация кода. К тому же в некоторых случаях JIT может показывать большую производительность по сравнению со статической компиляцией за счёт оптимизаций, возможных только во время исполнения [3]:

- Компиляция может осуществляться непосредственно для целевого процессора и операционной системы, на которой запущено приложение. Например, JIT может использовать векторные SSE2 расширения процессора, если он обнаружит их поддержку. Однако, до сих пор нет основных реализаций JIT, где этот подход бы использовался, ведь чтобы обеспечить подобный уровень оптимизации, сравнимый со статическими компиляторами, потребовалось бы либо поддерживать бинарный файл под каждую платформу, либо включать в одну библиотеку оптимизаторы под каждую платформу.

- Среда может собирать статистику о работающей программе и производить оптимизации с учётом этой информации. Некоторые статические компиляторы также могут принимать на вход информацию о предыдущих запусках приложения.

- Среда может делать глобальные оптимизации кода (например, встраивание библиотечных функций в код) без потери преимуществ динамической компиляции и без накладных расходов, присущих статическим компиляторам и линкерам.

Платформа Microsoft .NET состоит из общеязыковой среды разработки (Common Language Runtime, CLR) и общирного набора библиотек.

Общеязыковая среда разработки выполняет большое количество различных задач [1]:

- Управление памятью, выполнением потоков, выполнением кода, проверкой безопасности кода, компиляцией и другими системными службами. Эти средства являются внутренними для управляемого кода, который выполняется в среде CLR.

- Среда выполнения обеспечивает управление доступом для кода. Например, пользователи могут доверить исполняемому приложению, внедренному в веб-страницу, воспроизведение анимации на экране или звукозаписи, не позволяя ему при этом получить доступ к личным данным, файловой системе или сети. Таким образом, средства безопасности CLR предоставляют подлинному развернутому в Интернете программному обеспечению исключительно богатые функции.

- Обеспечение наджености кода, реализуя инфраструктуру строгой типизации и проверки кода, которую называют системой общих типов (CTS). Система общих типов обеспечивает самоописание всего управляемого кода. Различные языковые компиляторы корпорации Microsoft и независимых изготовителей создают управляемый код, удовлетворяющий системе общих типов. Это означает, что управляемый код может принимать другие управляемые типы и экземпляры, при этом обеспечивая правильность типов и строгую типизацию.

- Исключение многих часто возникающие проблемы с программным обеспечением. Например, среда выполнения автоматически управляет размещением объектов и ссылками на объекты, освобождая их, когда они больше не используются. Автоматическое управление памятью исключает две наиболее часто возникающие ошибки приложений: утечки памяти и недействительные ссылки на память.

- Среда выполнения также повышает продуктивность разработчиков. Например, программисты могут писать приложения на привычном языке разработки, при этом используя все преимущества среды выполнения, библиотеку классов и компоненты, написанные другими разработчиками на других языках. Это доступно любому производителю компиляторов, обращающихся к среде выполнения. Языковые компиляторы, предназначенные для платформы .NET Framework, делают средства .NET Framework доступными для существующего кода, написанного на соответствующих языках, существенно облегчая процесс переноса существующих приложений/

- Хотя среда выполнения разрабатывалась для будущего программного обеспечения, она также поддерживает сегодняшнее и вчерашнее программное обеспечение. Взаимодействие управляемого и неуправляемого кодов позволяет разработчикам использовать необходимые компоненты COM и библиотеки DLL.

- Среда выполнения разработана для повышения производительности. Хотя общеязыковая среда выполнения предоставляет многие стандартные службы времени выполнения, управляемый код никогда не интерпретируется. Средство компиляции по требованию (JIT) позволяет всему управляемому коду выполняться на машинном языке компьютера. Между тем диспетчер памяти устраняет возможность фрагментации памяти и увеличивает объем адресуемой памяти для дополнительного повышения производительности.

## 2.2 **Язык программирования C#**

Выбор языка программирования имеет огромное влияние, так как программисты будут погружены в него с начала конструирования программного средства до самого конца жизненного цикла.

Исследования показали, что выбор языка программирования несколькими способами влияет на производительность труда программистов и качество создаваемого ими кода.

Если язык хорошо знаком программистам, то они работают более производительно.

Программисты, использующие языки высокого уровня, достигают более высокой производительности и создают более качественный код, чем программисты, работающие с языками низкого уровня.

C# является универсальным, безопасным в отношении типов, объектно-ориентированным языком программирования. Цель C# заключается в обеспечении продуктивности работы программистов. Для этого в языке соблюдается баланс между простотой, выразительностью и надежностью. С самой первой версии главным архитектором языка C# был Андерс Хейлсберг (создатель Turbo Pascal и архитектор Delphi). Язык C# нейтрален в отношении платформ, но проектировался для эффективной работы с платформой Microsoft .NET Framework.

Особенностями языка C# c объектно-ориентированной точки зрения являются [4]:

- Унифицированная система типов. Фундаментальным строительным блоком в C# является инкапсулированная единица данных и функций, называющаяся типом. Язык имеет унифицированную систему типов, где все типы, независимо от того, представляют ли они бизнес-объекты или примитивные сущности вроде чисел, совместно используют один и тот же базовый набор функциональности.

- Классы и интерфейсы. В рамках традиционной объектно-ориентированной парадигмы единственной разновидностью типа является класс. В C# присутствуют типы многих других видов, один из которых представляет собой интерфейс. Интерфейс похож на класс за исключением того, что он только описывает члены. Реализация для этих членов поступает из типов, которые реализуют данный интерфейс.

- Свойства, методы и события. В чистой объектно-ориентированной парадигме все функции называются методами. В C# методы представляют собой только одну разновидность функций-членов, и различают также свойства и события (помимо прочих). Свойства – функции-члены, которые инкапсулируют фрагмент состояния объекта. События – функции-члены, которые упрощают выполнение действий при изменении состояния объекта.

C# главным образом является объектно-ориентированным языком, но также заимствует ряд парадигм функционального программирования [4]:

- Функции могут трактоваться как значения. За счет применения делегатов язык C# позволяет передавать функции как значения в и из других функций.

- Для чистоты в C# поддерживаются шаблоны. Основная черта функционального программирования заключается в том, чтобы избегать использования переменных, значения которых изменяются, отдавая предпочтение декларативным шаблонам. В C# имеются ключевые средства, содействующие таким шаблонам, в том числе возможность оперативного написания неименованных функций, которые «захватывают» переменные (лямбда-выражения), и возможность спискового или реактивного программирования через выражения запросов.

С# является типобезопасным языком. Это значит, что экземпляры типов могут взаимодействовать только через определяемые ими протоколы, подобным образом обеспечивая внутреннюю согласованность каждого типа. Поддерживаются статическая типизация и типизация времени выполнения. Статическая типизация обеспечивает безопасность типов на этапе компиляции, что позволяет избежать значительной категории ошибок еще до запуска программы, а также повышает производительность труда программистов [4].

По историческим причинам C# применялся в основном для написания приложений, выполняющихся на платформах Windows. Однако на сегодняшний день есть возможность писать и под другие платформы, такие как Mac OS X, iOS, Android.

C# постоянно эволюционирует. Команда разработки языка стремится взять лучшее из существующих парадигм и приемов программирования для создания максимально удобного и продуктивного средства написания кода. Далее приведен краткий обзор версий языка и их особенностей.

Первая версия C# была в значительной схожа с Java 1.4, но имея свои особенности как свойства (выглядящие в коде как поля объекта, но на деле вызывающие при обращении к ним методы класса), индексаторы (подобные свойствам, но принимающие параметр как индекс массива), события, делегаты, циклы foreach, структуры, передаваемые по значению, автоматическое преобразование встроенных типов в объекты при необходимости (boxing), атрибуты, встроенные средства взаимодействия с неуправляемым кодом (DLL, COM) и прочее [5].

Кроме того, в C# решено было перенести некоторые возможности C++, отсутствовавшие в Java: беззнаковые типы, перегрузку операторов (с некоторыми ограничениями, в отличие от C++), передача параметров в метод по ссылке, методы с переменным числом параметров, оператор goto (с ограничениями). Также в C# оставили ограниченную возможность работы с указателями — в местах кода, специально обозначенных словом unsafe и при указании специальной опции компилятору [5].

Вторая версия языка предоставила следующие возможности [5]:

- Частичные типы (разделение реализации класса более чем на один файл).

- Обобщённые, или параметризованные типы (generics). В отличие от шаблонов C++, они поддерживают некоторые дополнительные возможности и работают на уровне виртуальной машины. Вместе с тем, параметрами обобщённого типа не могут быть выражения, они не могут быть полностью или частично специализированы, не поддерживают шаблонных параметров по умолчанию, от шаблонного параметра нельзя наследоваться.

- Новая форма итератора, позволяющая создавать сопрограммы с помощью ключевого слова yield.

- Анонимные методы, обеспечивающие функциональность замыкания.

- Обнуляемые типы-значения, представляющие собой те же самые типы-значения, способные принимать также значение null. Такие типы позволяют улучшить взаимодействие с базами данных через язык SQL.

- Поддержка 64-разрядных вычислений, что кроме всего прочего, позволяет увеличить адресное пространство и использовать 64-разрядные примитивные типы данных.

Третья версия языка привнесла следующие возможности [5]:

- ключевые слова select, from, where, позволяющие делать запросы из XML документов, коллекций;

- инициализация объекта вместе с его свойствами;

- деревья выражений: лямбда-выражения теперь могут представляться в виде структуры данных, доступной для обхода во время выполнения, тем самым позволяя транслировать строго типизированные C#-выражения в другие домены (например, выражения SQL);

- анонимные типы.

Новые возможности в четвертой версии [5]:

- возможность использования позднего связывания;

- именованные и опциональные параметры;

- контракты в коде;

- библиотека параллельных задач TPL (Task Parallel Library).

Новые возможности в пятой версии C# [5]:

- Шаблон TAP (Task-based Asynchronous Pattern). TAP использует один метод для представления инициализации и завершения асинхронной операции.

- Асинхронные методы (async и await) - как реализация шаблона TAP.

Шестая версия языка обладает следующими особенностями [5]:

- null-условные операции;

- функции сжатые до выражений (expression-bodied functions)

- инициализаторы свойств;

- инициализаторы индексов;

- интерполяция строк;

- фильтры исключений;

- импорт статических функций из класса.

Новые возможности седьмой версии языка:

- Сопоставление с шаблоном. Вводится понятие шаблона (pattern), который представляет собой синтаксическую конструкцию, позволяющую проверить соответствие переменной определенной форме и извлечь из нее информацию.

- Кортежи.

- Локальные функции. Теперь функцию, которая используется только в теле какого-либо метода можно объявить прямо в теле этого метода.

- Улучшения литералов. Были добавлены бинарные литералы и символ разделителя (\_) в числовых литералах.

- Локальные переменные и возвращаемые значения по ссылке. Расширена функциональность ключевого слова ref. Теперь можно возвратить данные из метода или сохранить их в локальной переменной по ссылке.

- Расширение списка типов, возвращаемых асинхронными методами.

## 2.3 ASP.NET Core

Платформа ASP.NET Core представляет технологию от компании Microsoft, предназначенную для создания различного рода веб-приложений: от небольших веб-сайтов до крупных веб-порталов и веб-сервисов.

ASP.NET Core представляет собой продолжения развития платформы ASP.NET, но в то же время это не очередной релиз, а переосмысление всей платформы, ее революционное изменение.

ASP.NET Core построен на основе кросс-платформенной среды .NET Core, которая может быть развернута на основных популярных операционных системах: Windows, Mac OS X, Linux, что позволяет запускать веб-приложения не только на ОС Windows, но и на Linux и Mac OS. А для развертывания веб-приложения можно использовать традиционный IIS, либо кросс-платформенный веб-сервер Kestrel.

Хотя ASP.NET Core преимущественно нацелено на использование .NET Core, но фреймворк также может работать и с полной версией фреймворка .NET.

ASP.NET Core характеризуется расширяемостью. Фреймворк построен из набора относительно независимых компонентов. Теперь у пользователя есть возможность использовать встроенную реализацию этих компонентов, либо расширить их с помощью механизма наследования, либо вовсе создать и применять свои компоненты со своим функционалом.

Также было упрощено управление зависимостями и конфигурирование проекта. Фреймворк теперь имеет свой легковесный контейнер для внедрения зависимостей, и больше нет необходимости применять сторонние контейнеры, такие как Autofac, Ninject. Хотя при желании их также можно продолжать использовать.

При разработке в Visual Studio 2015/2017 проекты приложений имеют встроенную поддержку с такими популярными инструментами, как Bower, Grunt, Gulp, который позволяют управлять скриптами JavaScript и стилями CSS, автоматизировать и оптимизировать процесс веб-разработки [6].

Для обработки запросов теперь используется новый конвейер HTTP, который основан на компонентах Katana и спецификации OWIN. А его модульность позволяет легко добавить свои собственные компоненты.

Если суммировать, то можно выделить следующие ключевые отличия ASP.NET Core от предыдущих версий ASP.NET [7]:

- новый легковесный и модульный конвейер HTTP-запросов;

- возможность развертывать приложение как на IIS, так и в рамках своего собственного процесса;

- использование платформы .NET Core и ее функциональности;

- распространение пакетов платформы через NuGet;

- интегрированная поддержка для создания и использования пакетов NuGet;

- единый стек веб-разработки, сочетающий Web UI и Web API;

- конфигурация для упрощенного использования в облаке;

- встроенная поддержка для внедрения зависимостей;

- расширяемость;

- кроссплатформенность: возможность разработки и развертывания приложений ASP.NET на Windows, Mac и Linux;

- развитие как open source, открытость к изменениям. [8]

## 2.4 Angular

Angular - JavaScript-фреймворк от компании Google с открытым исходным кодом. Предназначен для разработки клиентских приложений. Прежде всего он нацелен на разработку SPA-решений (Single Page Application), то есть одностраничных приложений. Является наследником другого фреймворка AngularJS, но Angular это не новая версия, а принципиально новый фреймворк. Его цель - расширение браузерных приложений на основе MVC-шаблона, а также упрощение тестирования и разработки.

Новая версия изначально носила название AngularJS 2, но это вносило определенную путаницу. Чтобы обозначить, что новая версия есть принципиально новый путь развития, а дальнейшие версии будут лишь расширять возможности, а не уходить в сторону, было принято решение называть эту и последующие версии Angular.

На данный момент последняя версия имеет версию 4. При этом было принято решение не выпускать версию 3 чтобы не вносить путаницу из-за существования пакета роутинга версии 3.3.0.

В отличие от многих JavaScript-фреймворков Angular адаптирует и расширяет HTML. Таким образом получается подход разработки, основанный на данных. Двусторонняя привязка позволяет автоматически синхронизировать модель и представление. В результате Angular уменьшает роль DOM-манипуляций и улучшает тестируемость.

Angular, в отличие от предшественника, имеет много уникальных возможностей [9]:

- упрощенный синтаксис связывания;

- в качестве основного языка был принят TypeScript, но это не единственный поддерживаемый язык, помимо этого можно писать на привычном JavaScript и Dart; - модульность, за счет которой значительная часть функциональности ядра была перемещена на модули, за счет чего было достигнуто создания более легкого и быстрого ядра;

- упрощена маршрутизация;

- улучшена инъекция зависимостей.

## 2.5 TypeScript

В 2012 году компания Microsoft представила язык программирования TypeScript, расширяющий возможности JavaScript. Разработчиком языка выступил Андерс Хейлсберг создавший ранее Turbo Pascal, Delphi и C#.

Необходимость в создании языка была обусловлена недостатками JavaScript при создании крупномасштабных приложений в первую очередь самой компании Microsoft, а также других пользователей JavaScript. В результате TypeScript стал обладать определенными отличиями от JavaScript, такими как статическая типизация, поддержка полноценных классов в духе объектно-ориентированного программирования, а также поддержкой подключения модулей, что призвано повысить производительность труда программистов, облегчить читаемость, рефакторинг и повторное использование кода, помочь осуществлять поиск ошибок на этапе разработки и компиляции [10].

Спецификации языка открыты и опубликованы в рамках соглашения Open Web Foundation Specification Agreement (OWFa 1.0).

TypeScript является обратно совместимым с JavaScript и компилируется в последний. Фактически, после компиляции программу на TypeScript можно выполнять в любом современном браузере или использовать совместно с серверной платформой Node.js. Планируется, что в силу полной обратной совместимости адаптация существующих приложений на новый язык программирования может происходить поэтапно, путём постепенного определения типов, используемых в решении [10].

## 2.6 Entity Framework Core

Entity Framework Core представляет собой объектно-ориентированную, легковесную и расширяемую технологию от компании Microsoft для доступа к данным. EF Core является ORM-инструментом (object-relational mapping - отображения данных на реальные объекты). То есть EF Core позволяет работать базами данных, но представляет собой более высокий уровень абстракции: EF Core позволяет абстрагироваться от самой базы данных и ее таблиц и работать с данными независимо от типа хранилища. Если на физическом уровне происходит оперирование таблицами, индексами, первичными и внешними ключами, но на концептуальном уровне происходят уже операции над объектами [11].

Entity Framework Core поддерживает множество различных систем баз данных. При наличии необходимого провайдера можно работать с любой базой данных. Компания Microsoft уже предоставляет ряд провайдеров для популярных решений, таких как:

- MS SQL Server;

- SQLite;

- MySQL.

Возможность использования любой базы данных обеспечивается за счет такого архитектурного решения, как универсальных API работы с данными. Таким образом код работы с данными абстрагируется от базы данных используемой в решении. Это решение позволяет перейти от одной СУБД к другой, не меняя код работы непосредственно с данными, а только изменить настройки подключения и конфигурации к соответствующему провайдеру.

Entity Framework Core перенял многие идеи от предыдущих ORM-инструментов компании Microsoft, таких как линейка Entity Framwork. Стоит отметить, что Entity Framework Core не является новой версией по отношению к Entity Framework 6. Это принципиально новая технология. По этой причине у Entity Framework используется своя система версий. На данный момент старшая версия – 1.1. Эта версия была выпущена в конце 2016 года. Технология продолжает активно развиваться.

Как технология доступа к данным Entity Framework Core может использоваться на различных платформах стека .NET. Это и стандартные платформы типа Windows Forms, консольные приложения, WPF, ASP.NET 4.6/4.5. Это и новые технологии как UWP и ASP.NET Core. При этом кроссплатформенная природа EF Core позволяет задействовать ее не только на ОС Windows, но и на Linux и Mac OS X.

Центральной концепцией Entity Framework является понятие сущности или entity. Сущность определяет набор данных, которые связаны с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их коллекциями.

Entity Framework позволяет работать как с уже существующими базами данных, так и может создавать их по моделям.

Любая сущность, как и любой объект из реального мира, обладает рядом свойств. Например, если сущность описывает человека, то мы можем выделить такие свойства, как имя, фамилия, рост, возраст. Свойства необязательно представляют простые данные типа int или string, но могут также представлять и более комплексные типы данных. И у каждой сущности может быть одно или несколько свойств, которые будут отличать эту сущность от других и будут уникально определять эту сущность. Подобные свойства называют ключами.

При этом сущности могут быть связаны ассоциативной связью один-ко-многим, один-ко-одному и многие-ко-многим, подобно тому, как в реальной базе данных происходит связь через внешние ключи.

Отличительной чертой Entity Framework Core, как технологии ORM, является использование запросов LINQ для выборки данных из БД. С помощью LINQ появляется возможность создавать различные запросы на выборку объектов, в том числе связанных различными ассоциативными связями. А Entity Framework Core при выполнении запроса осуществляет трансляцию выражения LINQ в выражения, понятные для конкретной СУБД (как правило, в выражения SQL) [12].

## 2.7 MS SQL Server 2017

Microsoft SQL Server — система управления реляционными базами данных (РСУБД), разработанная корпорацией Microsoft. Основной используемый язык запросов — Transact-SQL, создан совместно Microsoft и Sybase. Transact-SQL является реализацией стандарта ANSI/ISO по структурированному языку запросов (SQL) с расширениями. Используется для работы с базами данных размером от персональных до крупных баз данных масштаба предприятия; конкурирует с другими СУБД в этом сегменте рынка [13].

SQL Server 2016 — самый крупный рывок вперед за всю историю развития платформы управления данными Майкрософт. Обеспечьте оперативную аналитику транзакционных и аналитических данных с помощью масштабируемой платформы баз данных, которая обладает всем необходимым: от беспрецедентной производительности при обработке в памяти, инновационных средств безопасности и высокой доступности до расширенных функций аналитики, расширяющих возможности критически важных приложений.

Особенности Microsoft SQL Server 2016:

- Always Encrypted (т.е. «Всегда зашифрованные») - это функционал, позволяющий хранить некоторые данные (например, персональные данные) в зашифрованном виде, при этом ключ шифрования хранится на клиенте, в результате происходит разделения между теми, кто владеет данными, и может просматривать их, например, клиент, персональные данные которого зашифрованы, и тех, кто управляет данными, но не должен иметь доступа к ним, например, администраторы или программисты SQL Server. Данная возможность обеспечивает защиту данных, как в состоянии покоя, так и в движении, т.е. в процессе передачи.

- Real-Time Operational Analytics («Оперативная аналитика в режиме реального времени») – это возможность SQL сервера обрабатывать как рабочую OLTP нагрузку, так и аналитические запросы на одних и тех же таблицах базы данных одновременно, т.е. в режиме реального времени. Традиционный подход к аналитике имеет несколько недостатков таких как: сложность реализации ETL процессов, затраты на приобретение дополнительных хранилищ данных и лицензий, а также один из самых главных недостатков, который для многих предприятий является неприемлемым, это задержка данных. Это связанно с тем, что задание ETL обычно выполняется один раз в сутки, например, в конце рабочего дня и соответственно анализировать эти данные можно только на следующий день. Возможность Real-Time Operational Analytics позволяет в некоторых случаях устранить все вышеперечисленные недостатки, т.е. уже нет необходимости в ETL и хранилищах данных. Одним из нюансов Real-Time Operational Analytics является то, что данная возможность нацелена на работу с одним источником данных, который участвует как в OLTP нагрузке, так и в аналитике. В случае необходимости анализировать данные из нескольких источников, потребность в отдельном хранилище данных не исчезает.

- SQL Server R Services (In-database Analytics) - это новый компонент SQL сервера который позволяет выполнять анализ внутри базы данных с использованием языка R. R - это язык программирования и библиотеки с открытым исходным кодом, который широко используется как средство для анализа данных. Все возможности языка R, включая возможность использования достаточно большого количества пакетов с открытым исходным кодом для решения повседневных задач, интегрированы в SQL Server 2016, что позволяет избежать проблем с их интеграцией в приложения. Кроме этого пользователи R Services могут использовать библиотеку алгоритмов ScaleR – это набор функций масштабируемых для обработки сотен миллиардов строк посредством параллельных вычислений, что обеспечивает производительность, которую невозможно достичь при использовании распространенных пакетов с открытым исходным кодом.

- PolyBase – это технология позволяющая посылать запросы как к реляционным, так и не реляционным данным, а также осуществлять импорт и экспорт этих данных. Другими словами в SQL Server теперь есть возможность обращаться к данным в Hadoop или Blob-хранилище Azure.

- AlwaysOn Availability Groups – в SQL Server 2016 есть улучшения групп доступности AlwaysOn, а именно это: возможность иметь до трех синхронных реплик, циклическая балансировка нагрузки на вторичные реплики, поддержка распределенных транзакций (DTC - Distributed Transaction Coordinator), поддержка групп управляемых учётных записей сервисов (gMSA - Group Managed Service Accounts), а также улучшена пропускная способность репликации журнала.

- Stretch Database – это технология, которая позволяет хранить «холодные» данные в облаке Azure с возможностью быстрого доступа к этим данным (под «холодными» данными здесь понимается данные, которые редко используются или вовсе не используются). Другими словами, если у на предприятии в базе данных есть таблица или таблицы, в которых хранятся исторические данные, достигающие терабайтных объемов при этом этот объем постоянно увеличивается, а нужно обеспечить сохранность этих данных, да так чтобы к ним можно было мгновенно обратиться, то технология Stretch Database в этом поможет. Преимущества при использовании Stretch Database следующие: отсутствие необходимости в приобретение дополнительных систем хранения данных, мгновенный доступ к «холодным» данным, обычный способ обращения к «холодным» данным, т.е. отсутствует необходимость в модификации запросов и приложений, перенос «холодных» строк (например, если в одной и той же таблице есть и «горячие», и «холодные» данные, можно перенести в облако Azure только «холодные»). Все что нужно, для того чтобы начать использовать Stretch Database необходимо наличие SQL Server 2016 и подписка на облако Azure для создания новой базы данных SQL Server Stretch Database;

- SSRS Mobile Reports – в службы Reporting Services добавлена возможность создавать отчеты, оптимизированные для просмотра на мобильных устройствах.

- Усовершенствования для Columnstore Indexes - в SQL Server 2016 есть ряд улучшений для колоночных индексов, например, кластерный columnstore индекс теперь поддерживает один или несколько некластеризованных индексов rowstore, таблица теперь может иметь один обновляемый некластеризованный индекс columnstore, а также теперь можно создать один columnstore индекс на таблице, оптимизированной для памяти.

- Улучшения In-Memory OLTP – данная технология также получила новые функциональные возможности такие как: поддержка операторов UNION, UNION ALL, DISTINCT, поддержка UNIQUE индексов, ограничений CHECK и FOREIGN KEY, триггеров, а также вложенных запросов;

- Database Scoped Configurations – эта возможность позволяет конфигурировать ряд параметров конфигурации базы данных на индивидуальном уровне базы данных.

- Live Query Statistics - SQL Server Management Studio 2016 предоставляет возможность просмотра живого плана выполнения активного запроса, другими словами, теперь можно в режиме реального времени смотреть, как выполняется запрос. С помощью данной возможности мы можем легко определить, какая часть запроса выполнятся достаточно долго и соответственно на основе этого скорректировать запрос, т.е. оптимизировать его.

- Query Store – данная функция автоматически фиксирует историю запросов, планов и статистики во время выполнения, и сохраняет их для обзора, что позволяет администраторам баз данных отслеживать ресурсоёмкие запросы и оптимизировать их.

- System-Versioned Temporal Tables – это новый тип временных таблиц, который позволяет сохранять полную историю изменений данных и получать информацию об этих данных на любой момент времени, а не только на текущий.

- Встроенная поддержка JSON – в SQL Server 2016 появилась возможность обрабатывать данные в формате JSON, т.е. теперь мы стандартными средствами можем экспортировать и импортировать JSON данные.

- Row-Level Security (RLS) – данная технология позволяет контролировать доступ к строкам в таблице базы данных на основе характеристик пользователя, выполняющего запрос (например, членство в группе или контекст выполнения). Другими словами, можно сделать так, чтобы один или несколько пользователей могли видеть (и соответственно обрабатывать) только определенные строки в таблице.

- Dynamic Data Masking – это функция динамической маскировки данных. Целью этой функции ограничить раскрытие конфиденциальных данных, запрещая пользователям, которые не должны иметь доступ к этим данным, просматривать их. Эта функция не шифрует данные, поэтому рекомендуется использовать Dynamic Data Masking в сочетание с другими технологиями безопасности, например, с Encryption или Row-Level Security, чтобы лучше защитить свои данные.



## Visual Studio 2017

Microsoft Visual Studio — линейка продуктов компании Microsoft, включающих интегрированную среду разработки программного обеспечения и ряд других инструментальных средств. Данные продукты позволяют разрабатывать как консольные приложения, так и приложения с графическим интерфейсом, в том числе с поддержкой технологии Windows Forms, а также веб-сайты, веб-приложения, веб-службы как в родном, так и в управляемом кодах для всех платформ, поддерживаемых Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone .NET Compact Framework и Silverlight [14].

Особенности Visual Studio:

- Повышение производительности труда программиста. Visual Studio включает в себя редактор исходного кода с поддержкой технологии интеллектуального дополнения IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и отладчик машинного уровня. Остальные встраиваемые инструменты включают в себя редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных [14].

- Поиск и исправление ошибок. Такие функции, как динамическое модульное тестирование, помощники по исправлению ошибок и выполнение до щелчка сокращают цикл разработки, уменьшают риски регрессии и выявляют основные причины дополнительных ошибок.

- Интеграция с облаком. Встроенные средства обеспечивают полную интеграцию со всеми приложениями .NET Core, Azure, службами, контейнерами Docker и другим ПО. Интерфейс схож с центром данных Azure.

- Эффективная совместная работа. Помогает управлять командными проектами, размещенными у поставщиков Visual Studio Team Services, Team Foundation Server или GitHub. Функция «Открыть любую папку» помогает открыть практически любой файл кода, не создавая формальный проект или решение.

- Поддержка плагинов. Visual Studio позволяет создавать и подключать сторонние дополнение для расширения функциональность практически на каждом уровне, включая добавление поддержки систем контроля версий исходного кода (как, например, Subversion и Visual SourceSafe), добавление новых наборов инструментов (например, для редактирования и визуального проектирования кода на предметно- ориентированных языках программирования) или инструментов для прочих аспектов процесса разработки программного обеспечения (например, клиент Team Explorer для работы с Team Foundation Server).

**3 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА**

Создание любого программного продукта требует предварительного создания архитектуры приложения, отвечающего всем требованиям и обеспечивающим высокий уровень внешних и внутренних характеристик, такие как корректность, практичность, гибкость и т.д. Несмотря на то что внутренние характеристика недоступны конечному пользователю, они оказывают непосредственное влияние на продуктивность разработки и качество конечного продукта. Процесс создания архитектуры приложения включает в себя принятие многих решений на основании широкого диапазона факторов. Ни одно решение не является единственно верным, но каждое оказывает огромное влияние на итоговый результат.

Не существует общепринятого термина «архитектура программного обеспечения». Тем не менее, когда дело касается практики, то для большинства разработчиков и так понятно какой код является хорошим, а какой плохим. Хорошая архитектура это прежде всего выгодная архитектура, делающая процесс разработки и сопровождения программы более простым и эффективным. Программу с хорошей архитектурой легче расширять и изменять, а также тестировать, отлаживать и понимать. То есть, на самом деле можно сформулировать список вполне разумных и универсальных критериев:

- Эффективность системы. В первую очередь программа, конечно же, должна решать поставленные задачи и хорошо выполнять свои функции, причем в различных условиях. Сюда можно отнести такие характеристики, как надежность, безопасность, производительность, способность справляться с увеличением нагрузки (масштабируемость).

- Гибкость системы. Любое приложение приходится менять со временем. Чем быстрее и удобнее можно внести изменения в существующий функционал, чем меньше проблем и ошибок это вызовет — тем гибче и конкурентоспособнее система. Поэтому в процессе разработки старайтесь оценивать то, что получается, на предмет того, как вам это потом, возможно, придется менять. Изменение одного фрагмента системы не должно влиять на ее другие фрагменты.

- Расширяемость системы. Возможность добавлять в систему новые сущности и функции, не нарушая ее основной структуры. На начальном этапе в систему имеет смысл закладывать лишь основной и самый необходимый функционал (принцип YAGNI — you ain’t gonna need it, «Вам это не понадобится»). Но при этом архитектура должна позволять легко наращивать дополнительный функционал по мере необходимости. Причем так, чтобы внесение наиболее вероятных изменений требовало наименьших усилии.

- Требование, чтобы архитектура системы обладала гибкостью и расширяемостью (то есть была способна к изменениям и эволюции) является настолько важным, что сформулировано в виде отдельного принципа - «Принципа открытости/закрытости» (Open-Closed Principle): Программные сущности (классы, модули, функции и т.п.) должны быть открытыми для расширения, но закрытыми для модификации. Иными словами, должна быть возможность расширить/изменить поведение системы без изменения/переписывания уже существующих частей системы. Это означает, что приложение следует проектировать так, чтобы изменение его поведения и добавление новой функциональности достигалось бы за счет написания нового кода (расширения), и при этом не приходилось бы менять уже существующий код. В таком случае появление новых требований не повлечет за собой модификацию существующей логики, а сможет быть реализовано прежде всего за счет ее расширения. Именно этот принцип является основой «плагинной архитектуры» (Plugin Architecture).

- Масштабируемость процесса разработки. Возможность сократить срок разработки за счёт добавления к проекту новых людей. Архитектура должна позволять распараллелить процесс разработки, так чтобы множество людей могли работать над программой одновременно.

- Тестируемость. Код, который легче тестировать, будет содержать меньше ошибок и надежнее работать. Но тесты не только улучшают качество кода. Многие разработчики приходят к выводу, что требование «хорошей тестируемости» является также направляющей силой, автоматически ведущей к хорошему дизайну, и одновременно одним из важнейших критериев, позволяющих оценить его качество: "Используйте принцип «тестируемости» класса в качестве «лакмусовой бумажки» хорошего дизайна класса. Существует целая методология разработки программ на основе тестов, называемая - разработка через тестирование (Test-Driven Development, TDD).

- Возможность повторного использования. Систему желательно проектировать так, чтобы ее фрагменты можно было повторно использовать в других системах.

- Хорошо структурированный, читаемый и понятный код.

- Сопровождаемость. Над программой, как правило, работает множество людей — одни уходят, приходят новые. После написания сопровождать программу тоже, как правило, приходится людям, не участвовавшем в ее разработке. Поэтому хорошая архитектура должна давать возможность относительно легко и быстро разобраться в системе новым людям. Проект должен быть хорошо структурирован, не содержать дублирования, иметь хорошо оформленный код и желательно документацию. И по возможности в системе лучше применять стандартные, общепринятые решения привычные для программистов. Чем экзотичнее система, тем сложнее ее понять другим (Принцип наименьшего удивления — Principle of least astonishment. Обычно, он используется в отношении пользовательского интерфейса, но применим и к написанию кода) [15].

Архитектура ПО часто описывается как организация или структура системы, где система разбивается на набор модулей, выполняющих определенную функцию или набор функций. Иначе говоря, основное назначение архитектуры – организация компонентов с целью обеспечения определенной функциональности. Такую организацию функциональности часто называют группировкой компонентов по функциональным областям. На рисунке 3.1 представлена типовая архитектура приложения, компоненты которого сгруппированы по функциональным областям.

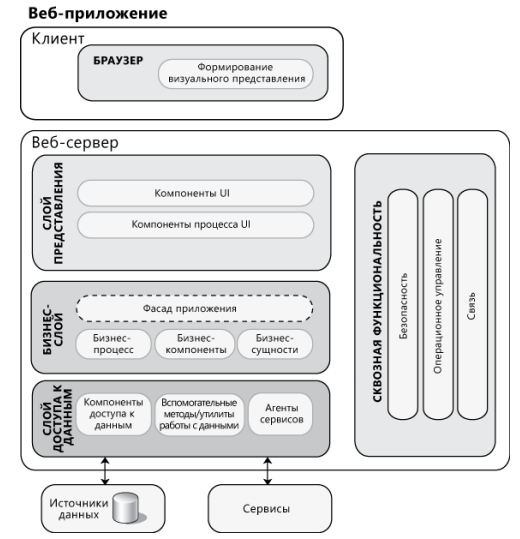


Рисунок 3.1 –Типовая архитектура ПО.

Архитектура должна:

- соответствовать требованиям;

- раскрывать структуры системы, но скрывать детали реализации;

- реализовывать все варианты использования и сценарии.

Проектируемая система представляет собой совокупность следующих слоев:

- слой представления;

- серверный слой;

- слой сервисов;

- слой доступа к данным.

## 3.1 Слой представления

Слой представления - это единственный слой, который непосредственно связан с пользователем. Его цель состоит в предоставлении пользователю информации и взаимодействию с ним, формирует запросы к серверу, обрабатывает его ответы. Исходя из этих задач можно сказать, что данный слой должен отвечать многим требованиям, таким как удобство использования, дизайн и другие. Неудачно выполненный слой представления способен свести пользовательский интерес к минимуму, при том что все остальные части программного продукта могут выполнены идеально.

Пользовательский интерфейс (GUI - graphical user interface) часто проектируется на этапе выработки требований. Если это не так, то его следует определить на этапе разработки архитектуры. Архитектура должна описывать главные элементы формата страниц. Удобство пользовательского интерфейса может в итоге определить популярность или провал программного комплекса.

Архитектура должна быть модульной, чтобы пользовательский интерфейс можно было изменить, не затрагивая бизнес-правил и модулей программы, отвечающих за вывод данных. Например, архитектура должна обеспечивать возможность сравнительно легкой замены группы классов интерактивного интерфейса на группу классов интерфейса командной строки. Такая возможность весьма полезна, так как интерфейс командной строки удобен для тестирования программного средства на уровне блоков или подсистем.

При разработке пользовательского интерфейса можно выделить определенные задачи:

- Разработка интерфейса должна начинаться с определения задачи или набора задач, для которых продукт предназначен.

- Простое должно оставаться простым. Не стоит усложнять интерфейс, нужно постоянно думать как сделать его проще и понятнее.

- Пользователи не должны думать над тем, как устроена программа. Все что они видят – интерфейс. С точки зрения потребителя именно интерфейс является конечным продуктом.

- Интерфейс должен быть ориентирован на пользователя, отвечая его нуждам и учитывая его слабости.

- Необходимо учитывать поведения и привычки пользователей. Не стоит менять хорошо всем известные элементы пользовательского интерфейса на неожиданные, а новые стоит делать интуитивно понятными.

- Разработка интерфейса должны исходить из принципа наименьшего возможного количества действия со стороны пользователя.

Чтобы свести время ожидания пользователем загрузки страниц к минимуму, было принято решение построить программный продукт в виде одностраничного приложения (Single Page Application, SPA). Архитектура одностраничных приложений представлена на рисунке 3.2.

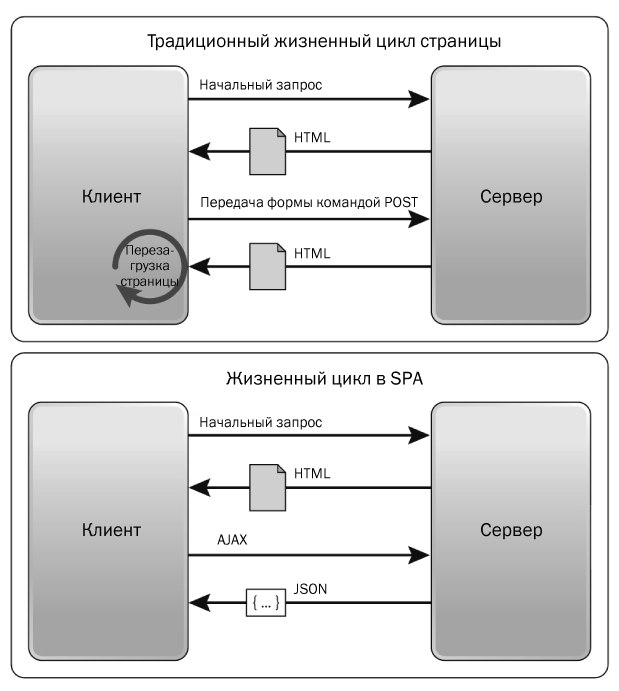


Рисунок 3.2 – Архитектура одностраничных приложений

Исходя из вышесказанного было принято решение реализовать слой представления при помощи JavaScript-фреймворка Angular. Эта клиентская технология позволяет выполнять логику на клиенте, что положительно сказывается на взаимодействии пользователя с программой.

Для работы с Angular необходимо установить сервер Node.js и пакетный менеджер npm, если они отсутствуют на рабочей машине. Для установки можно использовать программу установки node.js. Вместе с сервером она также установить и npm. При этом особого какого-то знания для работы с NodeJS и npm не требуется. После установки node.js и npm необходимо установить и настроить Angular. Для этого нужно создать конфигурационные файлы.

Файл package.json устанавливает пакеты и зависимости, которые будут использоваться проектом.

Для определения кода приложения применяется язык TypeScript, поэтому в папке проекта был создан новый файл tsconfig.json. Данный файл определяет настройки для компилятора TypeScript.

Приложение Angular состоит из модулей. Модульная структура позволяет легко подгружать и задействовать только те модули, которые непосредственно необходимы. И каждое приложение имеет как минимум один корневой модуль. Поэтому в большинстве проектах на Angular будет пристуствовать файл с модулем app.module.ts

Компоненты представляют основные строительные блоки приложения Angular. Каждое приложение Angular имеет как минимум один компонент. Компонент управляет отображением представления на экране.

## 3.2 Серверный слой

Серверный слой программного продукта представляет собой API, используемый для взаимодействия клиента и сервера. Для реализации было принято решение использовать ASP.NET Core WebAPI.

API повсеместно применяется в программировании. API определяет интерфейс, предоставляемый программой, при этом API позволяет абстрагироваться от реализации. При необходимости из API можно строить иерархии, таким образом API верхнего уровня использует нижележащий API, тот, в свою очередь, может взаимодействовать с другим API.

В контексте веб-программирования, API представляет собой набор HTTP-запросов, а также определение структуры HTTP-ответов, для выражения которых могут быть использованы XML или JSON форматы. WebAPI является практически синонимом веб-службы, хотя в последнее время за счет тенденции Web 2.0 осуществлен переход от SOAP к REST типу коммуникации [16].

В дипломные проекты было принято решение реализовывать RESTful на серверном слое.

REST (Representational State Transfer — «передача состояния представления») - архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. В определённых случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) это приводит к повышению производительности и упрощению архитектуры.

В сети Интернет вызов удалённой процедуры может представлять собой обычный HTTP-запрос (обычно «GET» или «POST»; такой запрос называют «REST-запрос»), а необходимые данные передаются в качестве параметров запроса.

Для веб-служб, построенных с учётом REST (то есть не нарушающих накладываемых им ограничений), применяют термин «RESTful».

В отличие от веб-сервисов (веб-служб) на основе SOAP, не существует «официального» стандарта для RESTful веб-API. Дело в том, что REST является архитектурным стилем, в то время как SOAP является протоколом. Несмотря на то, что REST не является стандартом сам по себе, большинство RESTful-реализаций используют стандарты, такие как HTTP, URL, JSON и XML.

Существует шесть обязательных ограничений для построения распределённых REST-приложений по Филдингу.

Выполнение этих ограничительных требований обязательно для REST-систем. Накладываемые ограничения определяют работу сервера в том, как он может обрабатывать и отвечать на запросы клиентов. Действуя в рамках этих ограничений, система приобретает такие желательные свойства как производительность, масштабируемость, простота, способность к изменениям, переносимость, отслеживаемость и надежность.

Если сервис-приложение нарушает любое из этих ограничительных условий, данную систему нельзя считать REST-системой.

Обязательными условиями-ограничениями являются:

- Модель клиент-сервер. Первым ограничением применимым к нашей гибридной модели является приведение архитектуры к модели клиент-сервер. Разграничение потребностей является принципом, лежащим в основе данного накладываемого ограничения. Отделение потребности интерфейса клиента от потребностей сервера, хранящего данные, повышает переносимость кода клиентского интерфейса на другие платформы, а упрощение серверной части улучшает масштабируемость. Наибольшее же влияние на всемирную паутину, пожалуй, имеет само разграничение, которое позволяет отдельным частям развиваться независимо друг от друга, поддерживая потребности в развитии интернета со стороны различных организаций.

- Отсутствие состояния. Протокол взаимодействия между клиентом и сервером требует соблюдения следующего условия: в период между запросами клиента никакая информация о состоянии клиента на сервере не хранится. Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Состояние сессии при этом сохраняется на стороне клиента. Информация о состоянии сессии может быть передана сервером какому-либо другому сервису (например, в службу базы данных) для поддержания устойчивого состояния, например, с целью, и на период установления аутентификации. Клиент инициирует отправку запросов, когда он готов (возникает необходимость) перейти в новое состояние. Во время обработки клиентских запросов считается, что клиент находится в переходном состоянии. Каждое отдельное состояние приложения представлено связями, которые могут быть задействованы при следующем обращении клиента.

- Кэширование. Как и во Всемирной паутине, клиенты а также промежуточные узлы могут выполнять кэширование ответов сервера. Ответы сервера в свою очередь должны иметь явное или неявное обозначение как кэшируемые или некэшируемые с целью предотвращения получения клиентами устаревших или неверных данных в ответ на последующие запросы. Правильное использование кэширования способно полностью или частично устранить некоторые клиент-серверные взаимодействия, ещё более повышая производительность и расширяемость системы.

- Единообразие интерфейса. Наличие унифицированного интерфейса является фундаментальным требованием дизайна REST-сервисов. Унифицированные интерфейсы позволяют каждому из сервисов развиваться независимо.

- Слои. Клиент обычно не способен точно определить взаимодействует ли он напрямую с сервером, или же с промежуточным узлом в связи с иерархической структурой сетей (слои). Применение промежуточных серверов способно повысить масштабируемость за счет балансировки нагрузки и распределенного кэширования. Промежуточные узлы также могут подчиняться политике безопасности с целью обеспечения конфиденциальности информации [16].

Реализация выше приведенных требований позволяет получить следующие преимущества:

- надёжность (за счёт отсутствия необходимости сохранять информацию о состоянии клиента, которая может быть утеряна);

- производительность (за счёт использования кэша);

- масштабируемость;

- прозрачность системы взаимодействия (особенно необходимая для приложений обслуживания сети);

- простота интерфейсов;

- портативность компонентов;

- лёгкость внесения изменений;

- способность эволюционировать, приспосабливаясь к новым требованиям.

Для реализации логики API используются классы ASP.NET Core WebAPI, называемые контроллерами. Эти классы содержат в себе логику методов WebAPI, находятся в папке Controllers решения. Все эти методы поддерживают архитектуру REST, таким образом для взаимодействия с ними используются методы HTTP:

* GET;
* POST;
* PUT;
* DELETE.

Во избежание ненужных затрат на передачу данных по сети используются вью модели. Так как обычные модели могут быть избыточны, возникает необходимость хранить только необходимую информацию, для этого и создаются вью модели. Такой подход позволяет упрощает сериализацию данных для удобной передачи по сети, уменьшает траффик. Зачастую для одной модели представлено две вью модели. Одна из них содержит необходимый минимум информации для отображения в списках, а другая более развернутая – для работы со связями между сущностями.

## 3.3 Слой сервисов

Зачастую программные продукты подразумевают применение разного рода интерфейсов к хранимым данным и реализуемой логике — загрузчиков данных, интерфейсов пользователя, шлюзов интеграции и т.д. Несмотря на различия в назначении, подобные интерфейсы часто нуждаются в одних и тех же функциях взаимодействия с приложением для манипулирования данными и выполнения бизнес-логики. Функции могут быть весьма сложными и способны включать транзакции, охватывающие многочисленные ресурсы, а также операции по координации реакций на действия. Описание логики взаимодействия в каждом отдельно взятом интерфейсе сопряжено с многократным повторением одних и тех же фрагментов кода. Сервисный слой и его возможные клиенты представлены на рисунке 3.3.

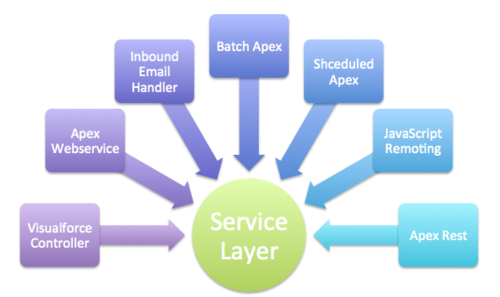


Рисунок 3.3 – Сервисный слой и его возможные клиенты

Слой определяет границы приложения и множество операций, предоставляемых им для интерфейсных клиентских слоев кода. Он инкапсулирует бизнес-логику приложения, управляет транзакциями и координирует реакции на действия.

Слой служб может быть реализован несколькими способами, удовлетворяющими всем упомянутым выше условиям. Различия проявляются в методах распределения ответственности вне интерфейса слоя служб. Прежде чем окунуться в особенности альтернативных реализаций, позвольте осветить некоторые основополагающие аспекты.

Слой сервисов представляет собой типовое решение по организации бизнес-логики. Многие проектировщики предпочитают разносить бизнес-логику по двум категориям: логика домена (domain logic) имеет дело только с предметной областью как таковой (примером могут служить стратегии вычисления зачтенного дохода по контракту), а логика приложения (application logic) описывает сферу ответственности приложения (например, уведомляет пользователей и сторонние приложения о протекании процесса вычисления доходов). Логику приложения часто называют также "логикой рабочего процесса", несмотря на то что под "рабочим процессом" часто понимаются совершенно разные вещи.

Модель предметной области более предпочтительна в сравнении со сценарием транзакции, поскольку исключает возможность дублирования бизнес-логики и позволяет бороться со сложностью с помощью классических проектных решений. Но размещение логики приложения в "чистых" классах домена чревато нежелательными последствиями. Во-первых, классы домена допускают меньшую вероятность повторного использования, если они реализуют специфическую логику приложения и зависят от тех или иных прикладных инструментальных пакетов. Во-вторых, смешивание логики обеих категорий в контексте одних и тех же классов затрудняет возможность новой реализации логики приложения с помощью специфических инструментальных средств, если необходимость такого шага становится очевидной. По этим причинам слой служб предусматривает распределение "разной" логики по отдельным слоям, что обеспечивает традиционные преимущества расслоения, а также большую степень свободы применения классов домена в разных приложениях.

Двумя базовыми вариантами реализации слоя служб являются создание интерфейса доступа к домену (domain facade) и конструирование сценария операции (operation script). При использовании подхода, связанного с интерфейсом доступа к домену, слой служб реализуется как набор "тонких" интерфейсов, размещенных "поверх" модели предметной области. В классах, реализующих интерфейсы, никакая бизнес-логика отражения не находит — она сосредоточена исключительно в контексте модели предметной области. Тонкие интерфейсы устанавливают границы и определяют множество операций, посредством которых клиентские слои взаимодействуют с приложением, обнаруживая тем самым характерные свойства слоя служб.

Создавая сценарий операции, вы реализуете слой служб как множество более "толстых" классов, которые непосредственно воплощают в себе логику приложения, но за бизнес-логикой обращаются к классам домена. Операции, предоставляемые клиентам слоя служб, реализуются в виде сценариев, создаваемых группами в контексте классов, каждый из которых определяет некоторый фрагмент соответствующей логики. Подобные классы, расширяющие супертип слоя (Layer Supertype) и уточняющие объявленные в нем абстрактные характеристики поведения и сферы ответственности, формируют "службы" приложения (в названиях служебных типов принято употреблять суффикс "Service"). Слой служб и заключает в себе эти прикладные классы. Преимуществом использования слоя служб является возможность определения набора общих операций, доступных для применения многими категориями клиентов, и координация откликов приложения на выполнение каждой операции. В сложных случаях отклики могут включать в себя логику приложения, передаваемую в рамках атомарных транзакций с использованием нескольких ресурсов. Таким образом, если у бизнес-логики приложения есть более одной категории клиентов, а отклики на варианты использования передаются через несколько ресурсов транзакций, использование слоя служб с транзакциями, управляемыми на уровне контейнера, становится просто необходимым, даже если архитектура приложения не является распределенной [17].

На данном этапе также необходимо задумать о данных, которые будут возвращать сервисы. Передача сущностей, полученных напрямую из базы данных не является лучшим решением, ввиду того что данные сущности часто содержать много лишней информации, и как правиле совершенно ненужной для клиента. Во избежание ненужных затрат на передачу ненужных данным используется еще один паттерн проектирования Data Transfer Object. Данный шаблон позволяет уменьшить количество передаваемых данных за счет хранения в сущностях только данных необходимых для вызова информации. Так же такой подход упрощает сериализации данным для удобной передачи по сети. Сущности такого типа хранят в себе только простые типы данных, а вместо ссылки на другие объекты только их идентификаторы. Как правило, на одну сущность уровня данных создаются две такие сущности. Одна с минимальным необходимым набором данных предназначенная для отображения сущности в списках. Вторая более развернутая для работы со связями между сущностей.

## 3.4 Слой доступа к данным

При проектировании программного продукта большое значение приобретает вопрос организации доступа к данным. Тому есть ряд причин:

- ценовые или прочие политики поставщиков хранилищ данных регулярно меняются, но предприятия, использующие данные хранилища, не всегда согласны с этими изменениями;

- в процессе развития программного продукта имеющееся решение по хранению данных может перестать удовлетворять его потребностям;

- в ходе различных экономических процессов предприятию может быть не целесообразно использовать текущее решение хранения данных;

- технологии хранения данных развиваются, появляются новые средства, предназначенные для решения специализированных задач;

- в рамках проектов Open Source вырастают дешевые или даже бесплатные альтернативы дорогим коммерческим решениям.

Все это может привести к тому, что в какой-то момент жизненного цикла программного продукта может возникнуть желание или необходимость сменить технологию хранения данных либо начать использовать новые технологии одновременно с текущими. Однако если при проектировании автоматизированных систем их бизнес-логика не была отделена от работы с хранилищами данных, то смена инструмента хранения может привести к дорогостоящей и плохо управляемой миграции.

На стадии проектирования слоя уровня к данным стоит определиться с хранилищем данных. Существуют следующие основные направления SQL (реляционные) и NoSQL (нереляционные) решения. Различия между ними заключаются в том, как они спроектированы, какие типы данных поддерживают, как хранят информацию.

Представленная в 70-х, реляционная модель предлагает математический способ структуризации, хранения и использования данных. Отношения дают возможность группировки данных как связанных наборов, представленных в виде таблиц, содержащих упорядоченную информацию (например, имя и адрес человека) и соотносящих значения и атрибуты (его номер паспорта). Благодаря десятилетиям исследований и разработки РСУБД работают производительно и надёжно. В сочетании с большим опытом использования администраторами реляционные базы данных стали выбором, гарантирующим защиту информации от потерь.

При использовании реляционного подхода база данных должна соответствовать требованиям ACID (Atomicity, Consistency, Isolation, Durability — атомарность, непротиворечивость, изолированность, долговечность). Это позволяет уменьшить вероятность неожиданного поведения системы и обеспечить целостность базы данных. Достигается подобное путём жёсткого определения того, как именно транзакции взаимодействуют с базой данных. Это отличается от подхода, используемого в NoSQL-базах, которые ставят во главу угла гибкость и скорость, а не 100% целостность данных.

Несмотря на строгие принципы формирования и обработки данных, РСУБД могут быть весьма гибкими, если приложить немного усилий.

NoSQL-способ структуризации данных заключается в избавлении от ограничений при хранении и использовании информации. Базы данных NoSQL, используя неструктурированный подход, предлагают много эффективных способов обработки данных в отдельных случаях (например, при работе с хранилищем текстовых документов).

Реляционные системы управления базами данных берут своё название от реализуемой модели — реляционной. Сейчас они остаются, да и ещё какое-то время будут, самым популярным выбором для надёжного, безопасного и производительного хранения данных.

Сравним эти подходы:

- Структура и тип хранящихся данных. SQL/реляционные базы данных требуют наличия однозначно определённой структуры хранения данных, а NoSQL базы данных таких ограничений не ставят.

- Запросы. Вне зависимости от лицензии, РСУБД реализуют SQL-стандарты, поэтому из них можно получать данные при помощи языка SQL. Каждая NoSQL база данных реализует свой способ работы с данными.

- Масштабируемость. Оба решения легко растягиваются вертикально (например, путём увеличения системных ресурсов). Тем не менее, из-за своей современности, решения NoSQL обычно предоставляют более простые способы горизонтального масштабирования (например, создания кластера из нескольких машин).

- Надёжность. Если речь заходит о надежности, то SQL решения имеют значительное преимущество, обусловленное их долголетием.

- Поддержка. РСУБД имеют очень долгую историю. Они очень популярны, и поэтому получить поддержку, платную или нет, очень легко. Поэтому, при необходимости, решить проблемы с ними гораздо проще, чем с NoSQL, особенно если проблема сложна по своей природе.

- Хранения и доступ к сложным структурам данных. По своей природе реляционные базы данных предполагают работу с сложными ситуациями, поэтому и здесь они превосходят NoSQL-решения.

Изучив плюсы и минусы существующих способов хранения данных было принято решение использовать реляционных подход, так как этот подход обеспечивает целостность данных, надежность в обращении. Несмотря на то что NoSQL проще масштабировать, при использовании SQL решение является предпочтительным в данной случае.

**4 СОЗДАНИЕ ПРОГРАММНОГО СРЕДСТВА**

В данном разделе описывается способ реализации дипломного проекта.

Для реализации клиентской части написано Angular приложение. Данное приложение находится в сборке Fantasy.Client.

Подход создания приложения, определяемый Angular, предусматривает создание шаблонов и компонентов. Шаблон представляет кусок разметки HTML с вкраплениями кода Angular. Фактически шаблон это и есть представление, которое увидит пользователь при работе с приложением. Компонент управляет отображением представления на экране. Каждый компонент должен иметь один шаблон.

Помимо отображения информации пользователю, данный слой содержит в себе клиентскую валидацию данных. Это позволяет моментально уведомить пользователя, что введенные данные некорректны. Но валидация реализована и на серверном слое, так как клиентскую валидацию не составляет труда обойти.

Рассмотрим пример реализации одной из страниц дипломного проекта, а именно страницы набора игроков в команду. Шаблон содержит следующий код, представленный ниже.

<div class="container">

<h1 class="line-title-center raleway inner-style">

Лобби

</h1>

<div class="row marginTop">

<div class="col-xs-9">

<label>Сортировать по:</label>

<select class="form-control" ng-init="sortBy = 'none'" ng-model="sortBy">

<option value="none">-- Выбор --</option>

<option value="league">Лиге</option>

<option value="tournament\_type">Типу турнира</option>

<option value="players">Количеству игроков</option>

<option value="start">Началу</option>

</select>

</div>

<div class="col-xs-4">

<label>Выбор категории:</label>

<select class="form-control" ng-model="leagues">

<option value="">Все</option>

<option ng-repeat="league **in** leagues">**{{**league.name**}}**</option>

</select>

</div>

<div class="block\_container clearfix">

<div class="content-filtered row">

<div \*ngIf="isFiltering === true" class="loading-tournametns-container">

<div class="loading-tournamesnt">

<div class="indeterminate"></div>

</div>

</div>

</div>

<div class="content-filter">

<div class="row ajax-content-more">

<table>

<thead>

<tr>

<td>Лига</td>

<td>Тип турнира</td>

<td>Количество игроков</td>

<td>Начало</td>

</tr>

</thead>

<tbody>

<tr \*ngFor="let tournament of tournaments | filter:

{name: query, league: { name: leagues }} | orderBy:sortBy">

<td>**{{**tournament.league**}}**</td>

<td>**{{**tournament.type**}}**</td>

<td>**{{**tournament.players**}}**</td>

<td>**{{**tournament.start**}}**</td>

<td \*ngIf="**{{**tournament.isOpen**}}** === true" (click)="getMore()" data-itemcount="10" data-page="1" data-model="Tournament" data-params="" class="btn btn-default btn-index display-block more\_button">Вступить</td>

</tr>

</tbody>

</table>

<div class="text-center col-xs-12 more\_button\_block marginBottom">

<div \*ngIf="isShowMoreButton === true" (click)="getMore()" data-itemcount="10" data-page="1" data-model="Tournament" data-params="" class="btn btn-default btn-index display-block more\_button">Еще турнуры</div>

</div>

</div>

</div>

<div \*ngIf="isShowMoreButton === false" class="loading-tournaments-container">

<div class="loading-tournaments">

<div class="indeterminate">

</div>

</div>

</div>

</div>

</div>

Для этого шаблона также есть и компонент.

@Component({

selector: 'tournaments',

moduleId: module.id,

templateUrl: './tournaments.component.html',

providers: [ TournamentService, LeageService, AuthService],

styleUrls: ['./tournament.component.css']

})

export class TournamentsComponent implements OnInit {

errorMessage: string;

isShowMoreButton: boolean = true;

isFiltering: boolean = false;

pageIndex: number = 0;

tournaments: TournamentFilter[];

leages: LeageFilter[];

muscles: MuscleFilter[];

mode = 'Observable';

constructor(private tournamentService: TournamentService,

private leagueService: LeagueService) { }

ngOnInit() {

this.getTournaments();

}

public getMore() {

this.pageIndex++;

this.getTournaments(this.pageIndex);

}

public getTournaments(pageIndex: number = 0) {

this.isShowMoreButton = false;

this.tournamentService.getTournaments(pageIndex)

.subscribe(

(tournaments) => {

if (!this.tournaments) {

this.tournaments = tournaments;

this.getLeagues;

}

else {

this.tournaments = this.tournaments.concat(tournaments);

}

this.isShowMoreButton = true;

},

error => this.errorMessage = <any>error);

}

}

В приложение часто встречается логика, разделяемая несколькими компонентами. Так как дублирование кода лишь усложняет процесс разработки, то данные места реализованы с помощью сервисов. Пример сервиса представлен ниже.

@Injectable()

export class TournamentService extends BaseCRUDService<tournamentFilter, tournament> {

constructor(authHttp: AuthHttp)

{

super(authHttp, 'tournaments');

}

getTournaments(pageIndex: number): Observable<tournamentFilter[]> {

let params: URLSearchParams = new URLSearchParams();

params.set('pageIndex', pageIndex.toString());

let requestOptions = new RequestOptions();

requestOptions.search = params;

return super.getAll(requestOptions);

}

}

Результат данного кода представлен на рисунке 5.1.

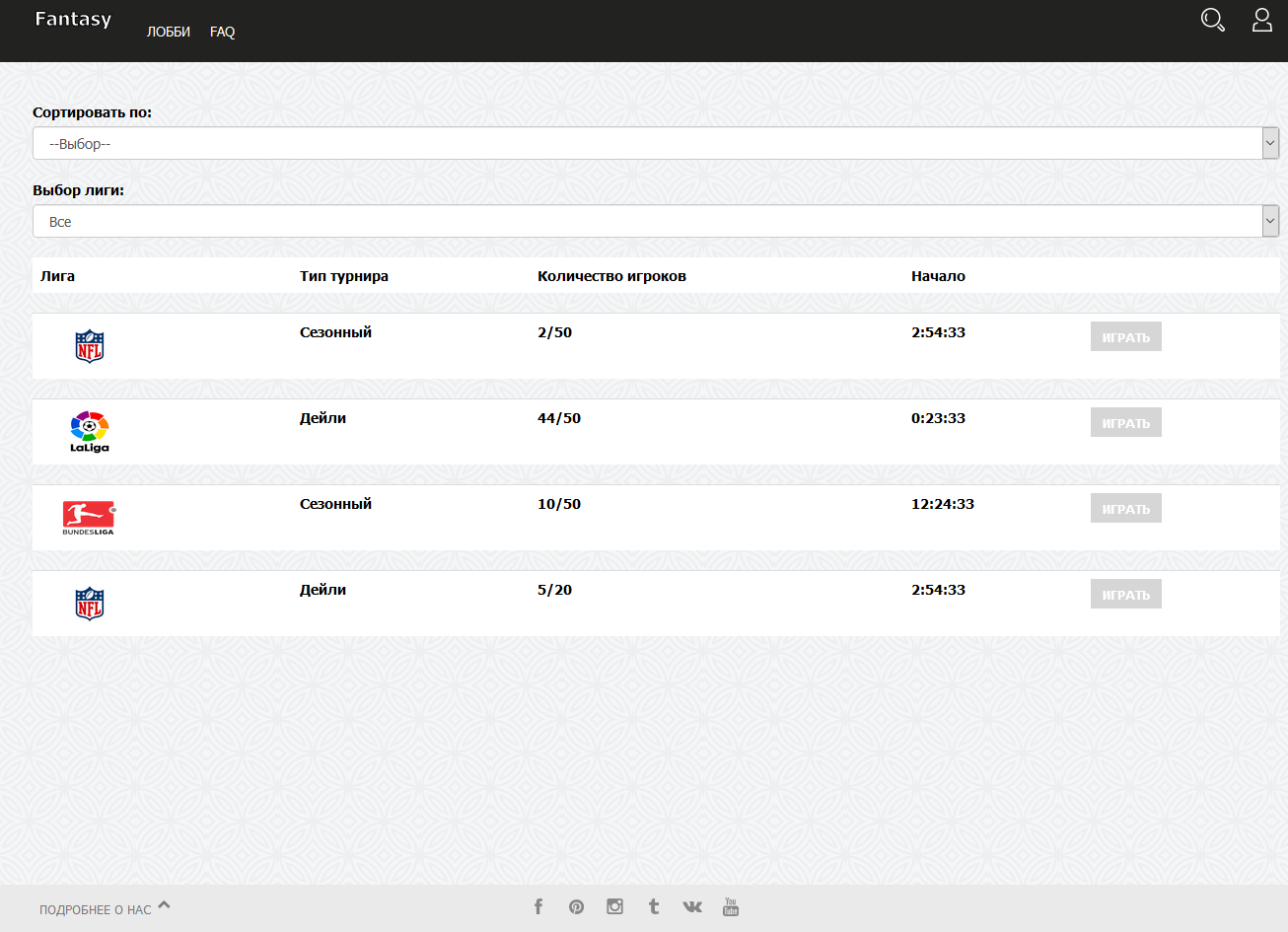


Рисунок 5.1 – Страница игрового лобби

Так как клиентская часть приложения получает данные от серверной части, перейдем к ее описанию.

Общение с сервером происходит посредством HTTP методов, определенных в контроллерах приложения. Ниже представлен один из таких контроллеров. Стоит отметить, что в самом контроллере не выполняется никаких операций, он выступает интерфейс взаимодействия с системой. Данный слой представлен в сборке Fantasy.API.

Помимо самого кода API есть код вьюмоделей, которые отдаются клиентом серверу. Они расположены в сборке Fantasy.ViewModels.

public class TournamentController : Controller

{

protected readonly TournamentService \_tournamenTournamentService;

protected readonly FantasySportsContext \_fantasySportsContext;

protected readonly IMapper \_mapper;

public TournamentController(TournamentService tournamenTournamentService, FantasySportsContext fantasySportsContext, IMapper mapper)

{

\_tournamenTournamentService = tournamenTournamentService;

\_fantasySportsContext = fantasySportsContext;

\_mapper = mapper;

}

[HttpGet]

public virtual IActionResult Get()

{

if (filterModel.PageSize == null)

{

filterModel.PageSize = \_adminOptions.DefaultPageSize;

}

var entities = \_tournamentService.Get(filterModel);

var models = \_mapper.Map<IEnumerable<TournmentVM>, List<Tournment>>(entities);

return Ok(models);

}

[HttpGet("{id}")]

public virtual IActionResult Get(long id)

{

var entity = \_tournamentService.Get(id);

if (entity == null)

{

return NotFound();

}

var model = \_mapper.Map<TournmentVM, TGetByIdModel>(entity);

return Ok(model);

}

[Authorize(Roles = "admin")]

[HttpPost]

public virtual IActionResult Post([FromBody]TPostModel value)

{

if (value == null)

{

return BadRequest();

}

else if (ModelState.IsValid == false)

{

return BadRequest(ModelState);

}

else

{

var entity = \_mapper.Map<TPostModel, TournmentVM>(value);

\_tournamenTournamentService.Add(entity);

\_fantasySportsContext.SaveChanges();

var modelForReturn = \_mapper.Map<TournmentVM, Tournment>(entity);

return CreatedAtAction("Get", new { id = entity.Id }, modelForReturn);

}

}

[Authorize(Roles = "admin")]

[HttpPut("{id}")]

public virtual IActionResult Put(long id, [FromBody]TPutModel value)

{

if (value == null || id != value.Id)

{

return BadRequest();

}

if (\_tournamenTournamentService.Get(id) == null)

{

return NotFound();

}

var entity = \_mapper.Map<TPutModel, TournmentVM>(value);

\_tournamenTournamentService.Update(entity);

\_fantasySportsContext.SaveChanges();

return NoContent();

}

[Authorize(Roles = "admin")]

[HttpDelete("{id}")]

public virtual IActionResult Delete(long id)

{

if (\_tournamenTournamentService.Get(id) == null)

{

return NotFound();

}

\_tournamenTournamentService.Remove(id);

\_fantasySportsContext.SaveChanges();

return NoContent();

}

}

Взаимодействие с базой данных реализовано посредством Entity Framework Core. Ключевым является класс, наследуемый от DbContext. Этот класс определяет контекст данных, используемый для взаимодействия с базой данных. В нем реализованы поля DbSet<TEntity>, представляющие набор объектов, которые хранятся в базе данных. Ниже представлен контекст данных приложения.

public class FantasySportContext : DbContext

{

public DbSet<Tournament> Tournaments { get; set; }

public DbSet<Athlete> Athletes { get; set; }

public DbSet<Sport> Sports { get; set; }

public DbSet<Scoring> Scorings { get; set; }

public DbSet<GameAction> GameActions { get; set; }

<другие поля DbSet>

public DbSet<User> Users { get; set; }

public DbSet<Role> Roles { get; set; }

public DbSet<UserRole> UserRoles { get; set; }

}

Для унификации работы с данными и облегчения задачи модульного тестирования был реализован паттерн репозиторий посредством создания базового класса BaseRepository. Данный класс включает в себя основную логику работы с базой данных. Классы потомки могут расширить способы взаимодействия.

public abstract class RepositoryBase<T> : IRepository<T> where T : class

{

private EBankContext \_dataContext;

protected IDbSet<T> DbSet;

public RepositoryBase(IDatabaseFactory databaseFactory)

{

DatabaseFactory = databaseFactory;

DbSet = DataContext.Set<T>();

}

protected IDatabaseFactory DatabaseFactory { get; }

protected EBankContext DataContext => \_dataContext ?? (\_dataContext = DatabaseFactory.Context);

public virtual void Add(T entity) => DbSet.Add(entity);

public virtual void Delete(Expression<Func<T, bool>> where)

{

var entities = DbSet.Where(where).AsEnumerable();

foreach (var e in entities)

DbSet.Remove(e);

}

public virtual void Delete(T entity) => DbSet.Remove(entity);

public virtual int Count() => DbSet.Count();

public virtual int Count(Expression<Func<T, bool>> where) => DbSet.Count(where);

public virtual T Get(Expression<Func<T, bool>> where) => DbSet.FirstOrDefault(where);

public virtual IEnumerable<T> GetAll() => DbSet.ToList();

public virtual T GetById(string id) => DbSet.Find(id);

public virtual T GetById(long id) => DbSet.Find(id);

public virtual IEnumerable<T> GetMany(Expression<Func<T, bool>> where) => DbSet.Where(where).ToList();

public virtual void Update(T entity)

{

DbSet.Attach(entity);

DataContext.Entry(entity).State = EntityState.Modified;

}

}

Что касается слоя сервисов, то он представляет собой набор интерфейсов, наследующихся от интерфейса IBaseService. Далее эти интерфейсы реализуют классы интерфейсов, содержащих логику работы приложения.

Код интерфейса IBaseService:

public interface IBaseService<TEntity>

where TEntity: BaseEntity

{

IEnumerable<TEntity> Get();

TEntity Get(long id);

void Add(TEntity entity);

void Remove(TEntity entity);

void Remove(long id);

void Update(TEntity entity);

}

Пример реализации сервиса для управления ролями:

public class RoleService : IRoleService

{

private IUnitOfWork \_unitOfWork;

private IRoleRepository \_roleRepository;

public RoleService(IUnitOfWork unitOfWork, IRoleRepository roleRepository)

{

\_unitOfWork = unitOfWork;

\_roleRepository = roleRepository;

}

public void Add(Role role)

{

if (role == null) throw new ArgumentNullException(nameof(role));

try

{

\_roleRepository.Add(role);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public void Delete(int id)

{

try

{

\_roleRepository.Delete(r => r.Id == id);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public void Delete(Role role)

{

if (role == null) throw new ArgumentNullException(nameof(role));

try

{

\_roleRepository.Delete(role);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public Role GetById(int id)

{

try

{

return \_roleRepository.Get(r => r.Id == id);

}

catch (Exception)

{

throw;

}

}

public IEnumerable<Role> GetAll()

{

try

{

return \_roleRepository.GetAll().ToList();

}

catch (Exception)

{

throw;

}

}

public void Update(Role role)

{

if (role == null) throw new ArgumentNullException(nameof(role));

try

{

\_roleRepository.Update(role);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;}}

# 5 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

В ходе разработки программного обеспечения неизбежно появляются ошибки. Это могут быть как изначально неверно принятые решения, так и ошибки появившиеся в ходе внесения дополнительной функциональности. В любом случае важным является процесс обнаружения поведения программного продукта несоответствующего требованиям. Причем чем раньше была найдена ошибка, тем проще ее будет исправить. На поздних этапах процесс исправления затрагивает слишком много вещей, что может привести к внесению новых ошибок.

Для оптимизации процесса разработки было принято решения воспользоваться техникой разработки через тестирование (Test-driven Development, TDD). Разработка через тестирование требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода. Тест содержит проверки условий, которые могут либо выполняться, либо нет. Когда они выполняются, говорят, что тест пройден. Прохождение теста подтверждает поведение, предполагаемое программистом. На практике модульные тесты покрывают критические и нетривиальные участки кода. Это может быть код, который подвержен частым изменениям, код, от работы которого зависит работоспособность большого количества другого кода, или код с большим количеством зависимостей.

Использование разработки через тестирование предполагает использование повторяющихся очень коротких циклов разработки:

- добавление теста;

- запуск всех тестов: убедиться, что новые тесты не проходят;

- написать код;

- запуск всех тестов: убедиться, что все тесты проходят;

- рефакторинг;

- повторить цикл.

Разработка через тестирование предлагает больше, чем просто проверку корректности, она также влияет на дизайн программы. Изначально сфокусировавшись на тестах, проще представить, какая функциональность необходима пользователю. Таким образом, разработчик продумывает детали интерфейса до реализации. Тесты заставляют делать свой код более приспособленным для тестирования. Например, отказываться от глобальных переменных, одиночек (singletons), делать классы менее связанными и легкими для использования. Сильно связанный код или код, который требует сложной инициализации, будет значительно труднее протестировать. Модульное тестирование способствует формированию четких и небольших интерфейсов. Каждый класс будет выполнять определенную роль, как правило, небольшую. Как следствие, зависимости между классами будут снижаться, а зацепление повышаться.

Несмотря на то, что при разработке через тестирование требуется написать большее количество кода, общее время, затраченное на разработку, обычно оказывается меньше. Тесты защищают от ошибок. Поэтому время, затрачиваемое на отладку, снижается многократно. Большое количество тестов помогает уменьшить количество ошибок в коде. Устранение дефектов на более раннем этапе разработки, препятствует появлению хронических и дорогостоящих ошибок, приводящих к длительной и утомительной отладке в дальнейшем.

Тесты позволяют производить рефакторинг кода без риска его испортить. При внесении изменений в хорошо протестированный код риск появления новых ошибок значительно ниже. Если новая функциональность приводит к ошибкам, тесты, если они, конечно, есть, сразу же это покажут. При работе с кодом, на который нет тестов, ошибку можно обнаружить спустя значительное время, когда с кодом работать будет намного сложнее. Хорошо протестированный код легко переносит рефакторинг. Уверенность в том, что изменения не нарушат существующую функциональность, придает уверенность разработчикам и увеличивает эффективность их работы. Если существующий код хорошо покрыт тестами, разработчики будут чувствовать себя намного свободнее при внесении архитектурных решений, которые призваны улучшить дизайн кода.

Разработка через тестирование способствует более модульному, гибкому и расширяемому коду. Это связано с тем, что при этой методологии разработчику необходимо думать о программе как о множестве небольших модулей, которые написаны и протестированы независимо и лишь потом соединены вместе. Это приводит к меньшим, более специализированным классам, уменьшению связанности и более чистым интерфейсам.

Поскольку пишется лишь тот код, что необходим для прохождения теста, автоматизированные тесты покрывают все пути исполнения. Например, перед добавлением нового условного оператора разработчик должен написать тест, мотивирующий добавление этого условного оператора. В результате получившиеся в результате разработки через тестирование тесты достаточно полны: они обнаруживают любые непреднамеренные изменения поведения кода [18].

Однако не стоит забывать, что при таком подходе, равно как и при любом подходе к тестированию, невозможно обнаружить все ошибки. При выполнении написанных во время итераций юнит-тестов происходит тестирование каждого из модулей по отдельности. Это означает, что ошибки интеграции, системного уровня, функций, исполняемых в нескольких модулях, не будут определены. Кроме того, данная технология бесполезна для проведения тестов на производительность.

Для реализации модульных тестов было принято решение использовать фреймворк xUnit.net. xUnit.net разработан в сотрудничестве с одним из авторов NUnit, Джимом Ньюкирком. При разработке этого фреймворка большой акцент был уделен его расширяемости и гибкости.

Ниже представлен пример теста написанного при разработке программного продукта.

public class TeamServiceTest

{

private TeamService \_teamService;

public TeamServiceTest()

{

\_teamService = new TeamService();

}

[Fact(DisplayName = "Add athlete to team")]

public void AddAthleteToItem()

{

var athlete = new Athlete();

var team = new Team();

\_teamService.AddAthleteToTeam(athlete, team);

Assert.True(team.Contains(athlete));

}

}

# 6 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗАРБОТКИ ПРОГРАММНОГО СРЕДСТВА

## 6.1 Характеристика программного средства портала для проведения фэнтези спорт турниров

Программные продукты являются продукцией, реализуемой покупателям по рыночным ценам. Широкое применение вычислительной техники требует постоянного обновления и совершенствования программных средств. Выбор эффективных проектов программных средств связан с их экономической оценкой и расчетом экономического эффекта. Поэтому разрабатываемые средства должны быть не только совершенными в техническом плане, но и выгодными с точки зрения экономики.

Программное средство, разрабатываемое в рамках рассматриваемого дипломного проекта, предназначено для автоматизации процесса проведения турниров по фэнтези спорту. Основная цель заключается в автоматизации процесса подсчета фэнтези очков получаемые за результативные действия игроков реальных команд в реальных соревнованиях.

Главной особенностью продукта является минимизация вмешательства пользователя, что позволяет максимально снизить вероятность возникновения ошибок из-за человеческого фактора. Помимо этого, следует учесть, что скорость получения результата будет значительно превосходить скорость, с которой комиссар лиги мог бы рассчитать вручную.

Написанное в ходе данного дипломного проекта программное средство предназначено для пользователей, увлеченных как фэнтези спортом, так и обычными спортивными соревнованиями. Возможность добавления любых видов спорта и лиг позволяет охватить максимально возможное количество целевой аудитории.

Воспользоваться программным средством смогут как небольшие компании, которые не могут себе позволить полноценные решения, однако при всем этом заинтересованные в анализе своей работы, так и компании, которые позволить подобные вложения себе могут, но перед этим желают убедиться в необходимости приобретения.

**6.2 Расчет себестоимости и отпускной цены**

Экономический эффект от разработки программного продукта определяется чистой прибылью, остающейся в распоряжении организации-разработчика после реализации данного программного продукта.

С учетом того, что стадии проектирования определяются только для крупных проектов (более 3 исполнителей), то для данного проекта стадии не определяются.

Исходные данные для расчета сметы затрат и отпускной цены представлены в таблице 6.1.

Таблица 6.1 – Исходные данные

|  |  |  |
| --- | --- | --- |
| Наименование | Условное  обозна­­чение | Значение |
| Категория сложности |  | 3 |
| Коэффициент сложности, ед. | Кс | 1,12 |
| Степень использования при разработке стандартных модулей, ед. | Kт | 0,6 |
| Коэффициент новизны, ед. | Кн | 0,7 |
| Продолжительность рабочего дня, ч. | Тч | 8 |
| Месячная тарифная ставка первого разряда, ВYN | ТМ1 | 31 |
| Коэффициент премирования, ед. | K | 1,2 |
| Норматив дополнительной заработной платы, ед. | Нд | 25 |
| Норматив отчислений в ФСЗН, % | Нсз | 34 |
| Отчисления в Белгосстрах, % | Нк | 0,6 |
| Цена одного машиночаса, BYN | Нмв | 1,2 |

Общий объём программного продукта определяется исходя из коли­чества и объёма функций, реализованных в программе:

|  |  |
| --- | --- |
|  | (6.1) |

где Vi – объём отдельной функции ПО, LоС;

n – общее число функций.

Каталог аналогов программного обеспечения предназначен для предварительной оценки объёма ПО методом структурной аналогии. Уточненный объём ПО рассчитывает­ся по формуле:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | (6.2) | |
|  | | | | |
| где | Vi | – | уточнённый объём отдельной функции ПО, LоС; | |
|  | n | – | общее число функций. | |

Уточненный объём некоторых функций изменился, и общий объём ПО составил Vo *=* 32910 LоС, общий уточнен­ный общем ПО – Vy *=* 29531 LоС.

Перечень и объём функций программного модуля перечислен в таблице 6.2.

Таблица 6.2 - Перечень и объем функций программного модуля

|  |  |  |  |
| --- | --- | --- | --- |
| № функции | Наименование (содержание) функции | Объем функции (LОС) | |
| По каталогу Vi | УточненныйVyi |
| 1 | 2 | 3 | 4 |
| 109 | Организация ввода/вывода информации в интерактивном режиме | 320 | 320 |  |
| 102 | Контроль, предварительная обработка и ввод информации | 450 | 450 |  |
| 201 | Генерация структуры базы данных | 4300 | 4300 |
| 204 | Обработка наборов и записей баз данных | 2670 | 2670 |
| 205 | Обслуживание базы данных в интерактивном режиме | 6950 | 5215 |
| 207 | Манипулирование данными | 9550 | 9550 |
| 208 | Организация поиска и поиск в базе данных | 5480 | 3836 |
| 210 | Загрузка базы данных | 2780 | 2780 |
| 506 | Обработка ошибочных и сбойных ситуаций | 410 | 410 |
|  | Итого | 32910 | 29531 |

Перечень и объём функций программного модуля перечислен в таблице 6.2. По приведенным данным уточненный объём некоторых функций изменился, и общий объём ПО составил Vo *=* 32910 LоС, общий уточнен­ный общем ПО – Vy *=* 29531 LоС.

По уточненному объёму ПО и нормативам затрат труда в расчете на единицу объёма определяются нормативная и общая трудоемкость разработки ПО. Согласно укрупненным нормам времени на разработку нормативная трудоемкость разрабатываемого проекта составля­ет Тн = 540 чел./дн.

Нормативная трудоемкость служит основой для оценки общей трудоемкости Тo. Используем формулу (6.3) для оценки общей трудоемкости для небольших проектов:

|  |  |
| --- | --- |
|  | (6.3) |
|  |  |
| где Kc – коэффициент, учитывающий сложность ПО;  Kт – поправочный коэффициент, учитывающий степень использования при разработке стандратных модулей;  Кн – коэффициент, учитывающий степень новизны ПО. | |

Дополнительные затраты труда на разработку ПО учитываются через коэффициент сложности, который вычисляется по формуле:

|  |  |
| --- | --- |
|  | (6.4) |
| где Ki – коэффициент, соответствующий степени повышения слож­­­­­­­­­­­­­ности ПО за счет конкретной характеристики;  n – количество учитываемых характеристик. | |

Наличие двух характеристик сложности позволяет вычислить коэффициент сложности позволяет установить Кс = 1,18. Разрабатываемое ПО использует стандартные компоненты. Степень использования стандартных компонентов определяется коэффициентом использования стандартных модулей – Кт. Согласно справочным данным указанный коэффициент равен Кт = 0,6. Трудоемкость создания ПО также зависит от его новизны и наличия аналогов. Разрабатываемое ПО не является новым, существуют аналогичные разработки у различных компаний, которые, однако уступают доступной функциональности. Влияние степени новизны определяется коэффициентом новизны – Кн. Согласно спра­вочным данным для разрабаты­ваемого ПО Кн = 0,7. Подставив приведенные выше коэффициенты для разрабатываемого ПО в формулу (6.3) получим общую трудоемкость разра­ботки:

|  |  |  |
| --- | --- | --- |
|  | (6.5) |  |

На основе общей трудоемкости и требуемых сроков реализации проекта вычисляется плановое количество исполнителей. Численность исполнителей проекта рассчитывается по формуле

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | (6.6) |
|  | | | | |
| где | To | – | объём отдельной функции ПО, LоС; | |
|  | Фэф | – | общее число функций. | |
|  | Тр | – | срок разработки проекта, лет. | |

Эффективный фонд времени работы одного разработчика вычисляет­ся по формуле

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | (6.7) |
|  | | | | |
| где | Дг | – | количество дней в году, дн.; | |
|  | Дп | – | количество праздничных дней в году, не совпадающих с выходными днями, дн.; | |
|  | Дв | – | количество выходных дней в году, дн.; | |
|  | До | – | количество дней отпуска, дн. | |

Согласно данным, приведённым в производственном календаре для пятидневной рабочей недели в 2017 году для Беларуси, фонд рабочего времени Фэф = 236 дней.

Учитывая срок разработки Тр = 0,33 года, общую трудоёмкость и фонд эффективного времени одного работника, полученные ранее, можно рассчитать численность исполнителей проекта

|  |  |
| --- | --- |
|  | (6.8) |
|  | |

Вычисленные оценки показывают, что для выполнения запланированного проекта в указанные сроки необходимо трое исполнителей.

Основная заработная плата исполнителей на конкретное ПО рассчитывается по формуле:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | (6.9) |
|  | | | | |
| где |  | – | часовая тарифная ставка *i-*го исполнителя, руб./час; | |
|  | Тч | – | общее число функций; | |
|  | Фп | – | плановый фонд рабочего времени *i*-го исполнителя, дн.; | |
|  | К | – | коэффициент премирования. | |

Расчет основной заработной платы представлен в таблице 6.3.

Таблица 6.3 -Расчет основной заработной платы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Категория исполнителя | Среднедневная заработная плата, руб. | Плановый фонд рабочего времени, дн. | Коэффициент премирования | Основная заработная плата, руб. |
| C# разработчик | 40 | 97 | 1,2 | 4656 |
| Фронтенд разработчик | 35 | 97 | 1,2 | 4074 |
| Разработчик баз данных | 35 | 55 | 1,2 | 2310 |
| Итого с премией (20%), Зо | - | - | - | 11040 |

Дополнительная заработная плата исполнителей проекта определяется по формуле:

|  |  |
| --- | --- |
|  | (6.10) |

где НД – норматив дополнительной заработной платы (20%).

Дополнительная заработная плата составит:

|  |  |
| --- | --- |
|  | (6.11) |

Отчисления в фонд социальной защиты населения и на обязательное страхование (ЗС) определяются в соответствии с действующими законодательными актами по формуле:

|  |  |
| --- | --- |
|  | (6.12) |

где НСЗ – норматив отчислений в фонд социальной защиты населения и на обязательное страхование (34 + 0,6%).

|  |  |  |
| --- | --- | --- |
|  |  | (6.13) |

Расходы по статье «Машинное время» (РМ) включают оплату машинного времени, необходимого для разработки и отладки ПС, и определяются по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (6.14) |

|  |  |  |  |
| --- | --- | --- | --- |
| где | Цм | – | цена одного машиночаса, руб.; |
|  | Тч | – | время работы в день, часов; |
|  | Ср | – | длительность проекта, дней. |

Стоимость машино-часа на предприятии составляет 1,2 руб. Разработка проекта займет 93 дня.

|  |  |  |
| --- | --- | --- |
|  |  | (6.15) |

Затраты по статье «Накладные расходы» (РН), связанные с необходимостью содержания аппарата управления, вспомогательных хозяйств и опытных (экспериментальных) производств, а также с расходами на общехозяйственные нужды (РН), определяются по формуле

|  |  |  |
| --- | --- | --- |
|  |  | (6.16) |

где НРН – норматив накладных расходов (50%).

Накладные расходы составят:

|  |  |  |
| --- | --- | --- |
|  |  | (6.17) |

Общая сумма расходов по всем статьям сметы (Сп) на ПО рассчитывается по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (6.18) |
|  |  | (6.19) |

Прибыль ПС рассчитывается по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (6.20) |

|  |  |
| --- | --- |
| где | Пс – прибыль от реализации ПО заказчику, руб.; |
|  | Урп – уровень рентабельности ПО, %. |

|  |  |  |
| --- | --- | --- |
|  |  | (6.21) |

Прогнозируемая отпускная цена ПС вычисляется по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (6.22) |

Подставив данные в формулу (6.19) получаем цену ПО без налогов:

|  |  |  |
| --- | --- | --- |
|  |  | (6.23) |

Налог на добавленную стоимость (НДСi):

|  |  |  |
| --- | --- | --- |
|  |  | (6.24) |

|  |  |  |
| --- | --- | --- |
| где | Ндс – норматив НДС, % | |
|  | |  | | (6.25) |

Прогнозируемая отпускная цена (Цo):

|  |  |  |
| --- | --- | --- |
|  |  | (6.26) |

Подставляя известные данные в формулу (6.23) получаем прогнози­руемую отпускную цену:

|  |  |  |
| --- | --- | --- |
|  |  | (6.27) |

Кроме того, организация-разработчик осуществляет затраты на сопровождение ПС (РС), которые определяются по нормативу (НС), где Нс – норматив расходов на сопровождение и адаптацию (20%).

|  |  |  |
| --- | --- | --- |
|  |  | (6.28) |

где Рс – расходы на сопровождение и адаптацию ПС в целом по организации (руб.);

СР – смета расходов в целом по организации без расходов на сопровождение и адаптацию (руб.).

|  |  |  |
| --- | --- | --- |
|  |  | (6.29) |

Таблица 6.4 - Исходные данные для расчета экономии ресурсов в связи с применением нового ПО

| Наименование показателя | Обозначение | Единицы измерения | Значение показателя | | Наименование источника информации |
| --- | --- | --- | --- | --- | --- |
| в базовом варианте | в новом варианте |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 1. Капитальные вложения, включая затраты пользователя на приобретение ПО | Kпp | руб. |  | 85031,4 | Договор заказчика с разработчиком |
| 2. Затраты на освоение ПО | Кос | руб. |  | 5061,4 | Договор заказчика с разработчиком |
| 3. Затраты на сопровождение ПО | Кс | руб. |  | 10122,8 | Договор заказчика с разработчиком |

Продолжение таблицы 6.4

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | | 3 | 4 | 5 | 6 |
| 4. Затраты на укомплектование ВТ техническими средствами в связи с  внедрением нового ПО | Ктс | | руб. |  | 6100,0 | Сметы затрат на внедрение |
| 5. Затраты на пополнение оборотных средств в связи с эксплуатацией нового ПО | | Коб | руб. |  | 2000,0 | Сметы затрат на внедрение |
| 6. Время простоя сервиса,обусловленное ПО, в день | | П1,П2 | мин | 50 | 10 | Расчетные данные пользователя и паспорт ПО |
| 7. Стоимость одного часа простоя | | Сп | руб. | 30,1 | 30,1 | Расчетные данные пользователя и паспорт ПО |
| 8. Среднемесячная ЗП одного  программиста | | Зсм | руб. | 820 | 820 | Расчетные данные пользователя |
| 9. Коэффициент начислений на зарплату | | Kнз |  | 1,5 | 1,5 | Рассчитывается по данным пользователя |
| 10. Среднемесячное  количество рабочих дней | | Д р | день |  | 21,5 | Принято для расчета |
| 11. Количество типовых задач, решаемых за год | | Зт1 ,Зт2 | задача | 1800 | 1800 | План пользователя |
| 12. Объем выполняемых работ | | А1, А2 | задача | 1800 | 1800 | План пользователя |
| 13. Средняя трудоемкость работ на задачу  ззззззззз1задачу зада- чу | | Tc1,  Тс2 | Человеко-часов | 6,0 | 0,87 | Рассчитывается по данным пользователя |
| 14. Количество часов работы ты в день | | Tч | ч | 8 | 8 | Принято для расчета расчета |
| 15. Ставка налога на прибыль | | Hп | % |  | 18 |  |

Экономия затрат на заработную платув расчете на 1 задачу (Cзe):

|  |  |  |
| --- | --- | --- |
| , |  | (6.30) |

|  |  |
| --- | --- |
| где | Зcм – среднемесячная заработная плата программиста (руб.); |
|  | Tc1, Tc2 – снижение трудоемкости работ в расчете на 1 задачу (человеко-часов);  Tч− количество часов работы в день (ч);  Дp− среднемесячное количество рабочих дней. |

Подставим значения в формулу и получим:

|  |  |  |
| --- | --- | --- |
|  |  | (6.31) |

Экономия заработной платы при использовании нового ПО (руб.):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | |  | (6.32) | |
| где | Сз− экономия заработной платы; | | |
|  | А2− количество типовых задач, решаемых за год (задач). | | |

Подставим значения в формулу 6.32 и получим:

|  |  |  |
| --- | --- | --- |
|  |  | (6.33) |

Экономия с учетом начисления на зарплату (Сн):

|  |  |  |
| --- | --- | --- |
|  |  | (6.34) |

Экономия за счет сокращения простоев сервиса (Сс) рассчитывается по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (6.35) |

где Дрг – плановый фонд работы сервиса (дней).

Найдем по формуле 6.35:

|  |  |  |
| --- | --- | --- |
|  |  | (6.36) |

Общая готовая экономия текущих затрат, связанных с использованием  
нового ПО (Со), рассчитывается по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (6.37) |

Подставим значения в формулу 6.37 и получим:

|  |  |  |
| --- | --- | --- |
|  |  | (6.38) |

Внедрение нового ПО позволит пользователю сэкономить на текущих затратах, т.е. практически получить на эту сумму дополнительную прибыль. Для пользователя в качестве экономического эффекта выступает лишь чистая прибыль − дополнительная прибыль, остающаяся в его распоряжении (ΔПЧ), которая определяется по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (6.39) |

где Hп− ставка налога на прибыль (%).

Подставим значения в формулу 6.39 и получим:

|  |  |  |
| --- | --- | --- |
|  |  | (6.40) |

В процессе использования нового ПО чистая прибыль в конечном итоге  
возмещает капитальные затраты. Однако полученные при этом суммы результатов (прибыли) и затрат (капитальных вложений) по годам приводят к единому времени − расчетному году (за расчетный год принят 2017-й год) путем умножения результатов и затрат за каждый год на коэффициент дисконтирования α. Используются коэффициенты: 2017 г. – 1, 2018-й – 0,8696, 2019-й – 0,7561, 2020 г. – 0,6575. Все рассчитанные данные экономического эффекта сводятся в таблицу 6.5.

Таблица 6.5 - Расчет экономического эффекта от использования нового программного средства

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Показатели | Единицы  измерения | Годы | | | |
| 2017 | 2018 | 2019 | 2020 |
| *Результаты* |  |  |  |  |  |
| Прирост прибыли за счет экономии затрат (Пч) | руб. |  | 54508 | 54508 | 54508 |
| То же с учетом фактора времени | руб. |  | 47400 | 41213,50 | 35839 |
| *Затраты* |  |  |  |  |  |
| Приобретение ПО (Кпр) | руб. | 85031,4 |  |  |  |
| Сопровождение (Кс) | руб. | 10122,8 |  |  |  |
| Всего затрат | руб. | 95154,2 |  |  |  |
| То же с учетом фактора времени | руб. | 95154,2 |  |  |  |
| *Экономический эффект* |  |  |  |  |  |
| Превышение результата над затратами | руб. | -95154,2 | 47400 | 41213,50 | 35839 |
| То же с нарастающим итогом | руб. | -95154,2 | -47754.2 | -6540.7 | 29298,3 |
| Коэффициент приведения | единицы | 1 | 0,8696 | 0,7561 | 0,6575 |

Реализация проекта ПО позволит заказчику снизить трудоемкость решения задач, сократить простои сервиса и минимизировать возникновение ошибок из-за человеческого фактора.

На основании приведённых выше расчетов и полученных результатов, можно сказать, что данный проект, то есть разработка и применение данного программного продукта, является экономически выгодной, так как, несмотря на значительные капиталовложения, уже через 3 года после внедрения данного программного средства, предприятие-заказчик покрывает затраты на приобретение и начинает получать прибыль.

Стоит отметить, что доходы от программного средства исчисляются не только оплатой со стороны заказчика, но также и как выплаты по партнерским программа, а также выплаты за счет показа на портале различной рекламы, что позволяет с уверенностью сказать, что разработка проекта представляется эффективной и полезной для заказчика

**ЗАКЛЮЧЕНИЕ**

В результате работы над дипломным проектом был разработан портал для проведения фэнтези спорт турниров.

Для успешной реализации программного продукта был проведен процесс исследования предметной области, анализ существующих аналогов. В результате были получены достоинства и недостатки данных решений, что было учтено при разработке дипломного проекта.

Также была изучена экономическая сторона разработки программного средства, а именно рассчитан экономический эффект от внедрения проекта и показатели эффективности использования программного продукта как для заказчиков, так и для пользователей. Полученные значения позволяют сказать, что разработка данного программного средства целесообразна.

Обобщение полученных на этом этапе знаний о проекте позволило выбрал технологии для разработки. В ходе работы над проектом были использованы язык C# с использование технологий Angular 2 совместно с языком TypeScript, ASP.NET Core Web Api, ASP.NET Core MVC, Entity Framework Core.

Результатом работы над дипломным проектом является разработанный портал для проведения фэнтези спорт турниров. Портал предоставляет следующие возможности:

* добавление видов спорта;
* добавление лиг, команд и игроков;
* создание открытых и закрытых турниров;
* подсчет результатов.

Так же, в ходе работы над дипломным проектом были получены новые знания и структурированы старые.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] MSDN [Электронный ресурс]. – Электронные данные. – Режим доступа: https://msdn.microsoft.com/ru-ru/library/zw4w595w(v=vs.110).aspx.

[2] CIL [Электронный ресурс]. – Электронные данные. – Режим доступа:

<https://ru.wikipedia.org/wiki/Common_Intermediate_Language>.

[3] JIT [Электронный ресурс]. – Электронные данные. – Режим доступа:

<https://ru.wikipedia.org/wiki/JIT>.

[4] Албахари, Джозеф. C# 6.0. Справочник. Полное описание / Джозеф Албахари, Бен Албахари. – 6-е изд. – ООО “И.Д. Вильямс”, 2016. – 1040 стр.

[5] C Sharp [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <http://ru.wikipedia.org/wiki/C_Sharp>.

[6] ASP.NET Core [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://metanit.com/sharp/aspnet5/1.1.php>.

[7] API [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://ru.wikipedia.org/wiki/API>.

[8] ASP.NET Core [Электронный ресурс]. – Электронные данные. – ­Режим доступа: https://docs.microsoft.com/en-us/aspnet/core/.

[9] Angular [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://en.wikipedia.org/wiki/Angular_(application_platform)>.

[10] TypeScript [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://ru.wikipedia.org/wiki/TypeScript>.

[11] Metanit [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://metanit.com/sharp/entityframeworkcore/1.1.php>.

[12] MS SQL Server [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://ru.wikipedia.org/wiki/Microsoft_SQL_Server>.

[13] MSDN [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://www.microsoft.com/ru-ru/sql-server/sql-server-2016>.

[14] Microsoft Visual Studio [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio>.

[15] Создание архитектуры программы или как спроектировать табуретку [Электронный ресурс]. – Электронные данные. – ­Режим доступа: https://habrahabr.ru/post/276593.

[16] REST [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://ru.wikipedia.org/wiki/REST>.

[17] Фаулер, Мартин. Архитектура корпоративных программных приложений – ООО “И.Д. Вильямс”, 2006. – 544 стр.

[18] TDD [Электронный ресурс]. – Электронные данные. – ­Режим доступа: <https://en.wikipedia.org/wiki/Test-driven_development>.

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Исходный текст программы**

namespace Fantasy.Services.Services

{

public class JwtAuthService : IJwtAuthService

{

private JwtAuthOptions \_jwtAuthOptions;

public JwtAuthService(IOptions<UserOptions> jwtAuthOptions)

{

\_jwtAuthOptions = jwtAuthOptions.Value.Jwt;

}

public string GetSerializedToken(User user)

{

ClaimsIdentity identity = null;

if (user != null)

{

var claims = new List<Claim>

{

new Claim(ClaimsIdentity.DefaultNameClaimType, user.Login)

};

if(user.UserRoles != null)

{

foreach (var userRole in user.UserRoles)

{

claims.Add(new Claim(ClaimsIdentity.DefaultRoleClaimType, userRole.Role.Name));

}

}

identity = new ClaimsIdentity(claims, "Token", ClaimsIdentity.DefaultNameClaimType, ClaimsIdentity.DefaultRoleClaimType);

}

var now = DateTime.Now;

var jwt = new JwtSecurityToken(

issuer: \_jwtAuthOptions.Issuer,

audience: \_jwtAuthOptions.Audience,

notBefore: now,

claims: identity.Claims,

expires: now.Add(TimeSpan.FromMinutes(\_jwtAuthOptions.Lifetime)),

signingCredentials: new SigningCredentials(\_jwtAuthOptions.GetSymmetricSecurityKey(), SecurityAlgorithms.HmacSha256));

var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);

var response = new JwtToken

{

token = encodedJwt,

username = identity.Name

};

return JsonConvert.SerializeObject(response, new JsonSerializerSettings { Formatting = Formatting.Indented });

}

}

}

public class RoleService : IRoleService

{

private IUnitOfWork \_unitOfWork;

private IRoleRepository \_roleRepository;

public RoleService(IUnitOfWork unitOfWork, IRoleRepository roleRepository)

{

\_unitOfWork = unitOfWork;

\_roleRepository = roleRepository;

}

public void Add(Role role)

{

if (role == null) throw new ArgumentNullException(nameof(role));

try

{

\_roleRepository.Add(role);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public void Delete(int id)

{

try

{

\_roleRepository.Delete(r => r.Id == id);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public void Delete(Role role)

{

if (role == null) throw new ArgumentNullException(nameof(role));

try

{

\_roleRepository.Delete(role);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public Role GetById(int id)

{

try

{

return \_roleRepository.Get(r => r.Id == id);

}

catch (Exception)

{

throw;

}

}

public IEnumerable<Role> GetAll()

{

try

{

return \_roleRepository.GetAll().ToList();

}

catch (Exception)

{

throw;

}

}

public void Update(Role role)

{

if (role == null) throw new ArgumentNullException(nameof(role));

try

{

\_roleRepository.Update(role);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public Role GetByName(string name)

{

if (name == null) throw new ArgumentNullException(nameof(name));

try

{

return \_roleRepository.Get(r => r.Name == name);

}

catch (Exception)

{

throw;

}

}

[Route("api/[controller]")]

public class UsersController : Controller

{

private readonly IJwtAuthService \_jwtAuthService;

[AllowOnlyAnonymous]

[HttpPost("register")]

public IActionResult Register([FromBody] UserRegisterModel userRegister)

{

if (userRegister == null)

{

return BadRequest();

}

else if (ModelState.IsValid == false)

{

return BadRequest(ModelState);

}

else

{

var user = \_tournamentService.GetUser(userRegister);

if(user != null)

{

if(user.Login == userRegister.Login)

{

ModelState.AddModelError(nameof(user.Login), \_localizer["LoginAlreadyExists"]);

}

if(user.Email == userRegister.Email)

{

ModelState.AddModelError(nameof(user.Login), \_localizer["EmailAlreadyExists"]);

}

return BadRequest(ModelState);

}

var entity = \_mapper.Map<UserRegisterModel, User>(userRegister);

\_tournamentService.Add(entity);

\_fantasySportsContext.SaveChanges();

return Ok();

}

}

[AllowOnlyAnonymous]

[HttpPost("login")]

public IActionResult Login([FromBody] UserLoginModel user)

{

if (user == null)

{

return BadRequest();

}

else if (ModelState.IsValid == false)

{

return BadRequest(ModelState);

}

var foundUser = \_tournamentService.GetUser(user);

if (foundUser == null)

{

ModelState.AddModelError("", \_localizer["WrongCredentials"]);

return BadRequest(ModelState);

}

return Ok(\_jwtAuthService.GetSerializedToken(foundUser));

}

}

public class UserRegisterModel

{

[Required(AllowEmptyStrings = false, ErrorMessage = "Не введен логин")]

public string Login { get; set; }

[Required(AllowEmptyStrings = false, ErrorMessage = "Не введена почта")]

[EmailAddress(ErrorMessage = "Неверный формат")]

public string Email { get; set; }

[Required(AllowEmptyStrings = false, ErrorMessage = "Не введен пароль")]

[StringLength(255, ErrorMessage ="Слишком короткий пароль", MinimumLength = 6)]

public string Password { get; set; }

[Required(AllowEmptyStrings = false, ErrorMessage = "Не введено подтверждение пароля")]

[StringLength(255, ErrorMessage = "Слишком короткий пароль", MinimumLength = 6)]

[Compare("Password", ErrorMessage = "Пароль и подтверждение отличаются")]

public string ConfirmPassword { get; set; }

}

public class UserLoginModel

{

[Required(AllowEmptyStrings = false, ErrorMessage = " Не введен логин ")]

public string LoginOrEmail { get; set; }

[Required(AllowEmptyStrings = false, ErrorMessage = " Не введен пароль ")]

[StringLength(255, ErrorMessage = " Слишком короткий пароль ", MinimumLength = 6)]

public string Password { get; set; }

}

public class RepositoryBase<T> : IRepository<T> where T : class

{

private EBankContext \_dataContext;

protected IDbSet<T> DbSet;

public RepositoryBase(IDatabaseFactory databaseFactory)

{

DatabaseFactory = databaseFactory;

DbSet = DataContext.Set<T>();

}

protected IDatabaseFactory DatabaseFactory { get; }

protected EBankContext DataContext => \_dataContext ?? (\_dataContext = DatabaseFactory.Context);

public virtual void Add(T entity) => DbSet.Add(entity);

public virtual void Delete(Expression<Func<T, bool>> where)

{

var entities = DbSet.Where(where).AsEnumerable();

foreach (var e in entities)

DbSet.Remove(e);

}

public virtual void Delete(T entity) => DbSet.Remove(entity);

public virtual int Count() => DbSet.Count();

public virtual int Count(Expression<Func<T, bool>> where) => DbSet.Count(where);

public virtual T Get(Expression<Func<T, bool>> where) => DbSet.FirstOrDefault(where);

public virtual IEnumerable<T> GetAll() => DbSet.ToList();

public virtual T GetById(string id) => DbSet.Find(id);

public virtual T GetById(long id) => DbSet.Find(id);

public virtual IEnumerable<T> GetMany(Expression<Func<T, bool>> where) => DbSet.Where(where).ToList();

public virtual void Update(T entity)

{

DbSet.Attach(entity);

DataContext.Entry(entity).State = EntityState.Modified;

}

}

public class TournamentController : Controller

{

protected readonly TournamentService \_tournamenTournamentService;

protected readonly FantasySportsContext \_fantasySportsContext;

protected readonly IMapper \_mapper;

public TournamentController(TournamentService tournamenTournamentService, FantasySportsContext fantasySportsContext, IMapper mapper)

{

\_tournamenTournamentService = tournamenTournamentService;

\_fantasySportsContext = fantasySportsContext;

\_mapper = mapper;

}

[HttpGet]

public virtual IActionResult Get()

{

if (filterModel.PageSize == null)

{

filterModel.PageSize = \_adminOptions.DefaultPageSize;

}

var entities = \_tournamentService.Get(filterModel);

var models = \_mapper.Map<IEnumerable<TournmentVM>, List<Tournment>>(entities);

return Ok(models);

}

[HttpGet("{id}")]

public virtual IActionResult Get(long id)

{

var entity = \_tournamentService.Get(id);

if (entity == null)

{

return NotFound();

}

var model = \_mapper.Map<TournmentVM, TGetByIdModel>(entity);

return Ok(model);

}

[Authorize(Roles = "admin")]

[HttpPost]

public virtual IActionResult Post([FromBody]TPostModel value)

{

if (value == null)

{

return BadRequest();

}

else if (ModelState.IsValid == false)

{

return BadRequest(ModelState);

}

else

{

var entity = \_mapper.Map<TPostModel, TournmentVM>(value);

\_tournamenTournamentService.Add(entity);

\_fantasySportsContext.SaveChanges();

var modelForReturn = \_mapper.Map<TournmentVM, Tournment>(entity);

return CreatedAtAction("Get", new { id = entity.Id }, modelForReturn);

}

}

[Authorize(Roles = "admin")]

[HttpPut("{id}")]

public virtual IActionResult Put(long id, [FromBody]TPutModel value)

{

if (value == null || id != value.Id)

{

return BadRequest();

}

if (\_tournamenTournamentService.Get(id) == null)

{

return NotFound();

}

var entity = \_mapper.Map<TPutModel, TournmentVM>(value);

\_tournamenTournamentService.Update(entity);

\_fantasySportsContext.SaveChanges();

return NoContent();

}

[Authorize(Roles = "admin")]

[HttpDelete("{id}")]

public virtual IActionResult Delete(long id)

{

if (\_tournamenTournamentService.Get(id) == null)

{

return NotFound();

}

\_tournamenTournamentService.Remove(id);

\_fantasySportsContext.SaveChanges();

return NoContent();

}

}

public class UserService : IUserService

{

private IUnitOfWork \_unitOfWork;

private IUserRepository \_userRepository;

public UserService(IUnitOfWork unitOfWork, IUserRepository userRepository)

{

\_unitOfWork = unitOfWork;

\_userRepository = userRepository;

}

public void Add(User user)

{

if (user == null) throw new ArgumentNullException(nameof(user));

try

{

\_userRepository.Add(user);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public User GetByEmail(string email)

{

if (email == null) throw new ArgumentNullException(nameof(email));

try

{

return \_userRepository.Get(u => u.Email == email);

}

catch (Exception)

{

throw;

}

}

public User GetByUserName(string username)

{

if (username == null) throw new ArgumentNullException(nameof(username));

try

{

return \_userRepository.Get(u => u.Username == username);

}

catch (Exception ex)

{

throw;

}

}

public User GetById(long id)

{

try

{

return \_userRepository.GetById(id);

}

catch (Exception)

{

throw;

}

}

public void Update(User user)

{

if (user == null) throw new ArgumentNullException(nameof(user));

try

{

\_userRepository.Update(user);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public void Delete(long id)

{

try

{

\_userRepository.Delete(u => u.Id == id);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public void Delete(User user)

{

if (user == null) throw new ArgumentNullException(nameof(user));

try

{

\_userRepository.Delete(user);

\_unitOfWork.Commit();

}

catch (Exception)

{

throw;

}

}

public long GetMaxId()

{

try

{

var users = \_userRepository.GetAll().ToList();

return users.Any() ? users.Max(x => int.Parse(x.Username)) : Init();

}

catch (Exception)

{

throw;

}

}

public string CreatePassword()

{

try

{

var regex = new Regex("^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*[0-9])\\S{8,50}$");

var dictionary = Security.GetNewDictinary();

string password;

var random = new Random();

do

{

password = string.Empty;

for (byte i = 0; i < 20; i++)

{

password += dictionary[random.Next(dictionary.Length)];

}

} while (!regex.IsMatch(password));

return password;

}

catch (Exception)

{

throw;

}

}

}

}

public class Role

{

public int Id { get; set; }

public string Name { get; set; }

public virtual ICollection<User> Users { get; set; }

}

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | | | | | *Наименование* | *Дополнительные сведения* | | | | |
|  | | | | | Текстовые документы |  | | | | |
|  | | | | |  |  | | | | |
| БГУИР ДП 1–40 01 03 00 001 ПЗ | | | | | Пояснительная записка | 86 c | | | | |
|  | | | | | Отзыв руководителя |  | | | | |
|  | | | | | Рецензия |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | | Графические документы |  | | | | |
|  | | | | |  |  | | | | |
| ГУИР.400103.001 ПД1 | | | | | Диаграмма классов | Формат А1 | | | | |
|  | | | | | (контроллеров). Чертеж. |  | | | | |
|  | | | | |  |  | | | | |
| ГУИР.400103.001 ПД2 | | | | | Диаграмма классов | Формат А1 | | | | |
|  | | | | | (моделей). Чертеж. |  | | | | |
|  | | | | |  |  | | | | |
| ГУИР.400103.001 ПД3 | | | | | Диаграмма компонентов | Формат А1 | | | | |
|  | | | | | (front-end). Чертеж. |  | | | | |
|  | | | | |  |  | | | | |
| ГУИР.400103.001 ПЛ1 | | | | | Диаграмма базы данных. | Формат А1 | | | | |
|  | | | | | Плакат. |  | | | | |
|  | | | | |  |  | | | | |
| ГУИР.400103.001 ПЛ2 | | | | | Диаграмма вариантов | Формат А1 | | | | |
|  | | | | | использования. Плакат. |  | | | | |
|  | | | | |  |  | | | | |
| ГУИР.400103.001 ПЛ3 | | | | | Диаграмма деятельности. | Формат А1 | | | | |
|  | | | | | Плакат. |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  | | | | |  |  | | | | |
|  |  |  |  |  | ***БГУИР ДП 1- 40 01 03 00 001 Д1*** | | | | | |
|  |  |  |  |  |
|  |  |  |  |  |
| ***Изм*** | ***Л.*** | ***№ докум.*** | ***Подп.*** | ***Дата*** | ***Портал для проведения фэнтези спорт турниров.***  ***Ведомость дипломного  проекта*** |  | | | ***Лист*** | ***Листов*** |
| ***Разраб.*** | | ***Абрамов Д.А.*** |  |  | ***Т*** |  |  | ***86*** | ***86*** |
| ***Пров.*** | | ***Богушевский Е.Л.*** |  |  | ***Информатика***  ***гр. 253503*** | | | | |
| ***Т.контр.*** | | ***Богушевский Е.Л.*** |  |  |
| ***Н.контр.*** | | ***Шиманский В.В.*** |  |  |
| ***Утв.*** | | ***Волорова Н.А.*** |  |  |