**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ……………………………………………………………………….. 3](#_gjdgxs)

[1 ОБЩИЕ СВЕДЕНИЯ ОБ ООО «ЛЕВЕРЕКС ИНТЕРНЕШНЛ»……………. 4](#_30j0zll)

1.1 История организации……………………………………..…………… 4

[1.2 Охрана труда и техника безопасности ...……………………………... 7](#_3znysh7)

[2](#_2et92p0) ОБЗОР ТЕХНОЛОГИИ *ASP* .*NET* *CORE* и *REST API*………………………... [9](#_tyjcwt)

[2.1 Технология *ASP .NET*…………………………………………………. 9](#_3dy6vkm)

[2.2](#_1t3h5sf) [Технология](#_3dy6vkm) *REST API*………………………………………………...13

[3 РАЗРАБОТКА *WEB API*………………………………………………………..](#_17dp8vu) 16

[3.1 Постановка задачи…………………………………………………….](#_3rdcrjn) 16

[3.2](#_26in1rg) Уровень доступа к данным…………………………………………... [1](#_lnxbz9)6

3.3 Уровень бизнес-логики……………………………………………….19

[3.4](#_1ksv4uv) Уровень представления………………………………………………. [20](#_44sinio)

[3.5 Верификация *Web API*………………………………………………..](#_2jxsxqh)21

[ЗАКЛЮЧЕНИЕ………………………………………………………………….. 26](#_z337ya)

[Список использованных источников…………………………………………... 27](#_3j2qqm3)

[ПРИЛОЖЕНИЕ А](#_1y810tw) [Листинг программы……………………………………….. 28](#_4i7ojhp)



# ВВЕДЕНИЕ

Производственная практика – это обязательная составляющая образовательного процесса, необходимая для подготовки квалифицированных работников, хорошо ориентирующихся не только в профильной теории, но и в реалиях трудовых будней. Этот этап обучения обычно осуществляется вне стен вуза – на базе учреждений, соответствующих будущей специальности студента.

Место практики: ООО «Леверекс Интернешнл».

В ходе обучения важно придерживаться согласованности теории и практики. Как и любой другой этап образовательного процесса, пребывание студента на производстве преследует определенные цели и задачи.

Цели технологической практики: закрепление, расширение, углубление и систематизация теоретических знаний, а также приобретение навыков проектирования и конструирования информационных систем.

Задачи технологической практики:

* развитие и закрепление практических навыков выполнения анализа предметной области;
* приобретение практического опыта проектирования программных систем;
* развитие и закрепление практических навыков использования языков и инструментальных средств моделирования при проектировании системы;
* развитие и закрепление практических навыков создания программных систем с использованием современных сред разработки, поддерживающих возможность командной работы, контроля проекта и версий системы;
* развитие и закрепление практических навыков разработки документации к системе (технического задания, инструкций пользователя и программиста).

Индивидуальным заданием от предприятия является разработка *ASP* .*NET* *Core Web API* с использованием *Swagger*.

# 1 ОБЩИЕ СВЕДЕНИЯ ОБ ООО «ЛЕВЕРЕКС ИНТЕРНЕШНЛ»

* 1. История организации

*LeverX Group* – международная *IT*-компания, объединившая две команды: *LeverX* и *Emerline*. Сегодня здесь работают более 900 человек. Офисы компании размещены в Беларуси, США, Австрии, Польше, Украине, Латвии, Узбекистане, России [1].

*LeverX* является партнером немецкой компании *SAP*, разрабатывает и внедряет *SAP*-решения для таких крупных компаний, как *KIA Motors*, «KAMAЗ», *BMW*, *Home Credit Bank*, *PepsiCo* и других.

*Emerline* занимается разработкой *Web*- и *Mobile*-решений как для стартапов, так и для крупного бизнеса из Западной Европы и США. Кроме того, в портфеле компании есть собственные продукты: антистресс-раскраска, приложение для дрессировки собак и сканер документов.

Иностранное производственное унитарное предприятие «*LeverX Group*» г. Гомель создано в 2021 году. Учредитель – американское предприятие ООО «Леверекс Интернешнл», созданное в 2004 году. Основное направление деятельности: разработка программного обеспечения. *LeverX Group* уже более 17 лет является сертифицированным партнером немецкой компании *SAP*, а в 2020 году получила статус – *SAP Global Strategic Supplier for Technical Services*. Услуги компании оценили более 400 клиентов по всему миру. [1]. Логотип компании представлен на рисунке 1.1.



Рисунок 1.1 – Логотип компании ООО «Леверекс Интернешнл»

Также компания развивает отдельное направление разработки под брендом *Emerline*, создает собственные мобильные продукты и активно сотрудничает со стартапами. Особое внимание *Emerline* уделяет работе с инновационным стеком технологий, включая *IoT*, *AR*, *AI*, *Machine Learning* и *Data Science*.

*LeverX Group* объединяет две команды: *LeverX* и *Emerline*. Основная цель заключается в предоставлении полного цикла услуг по технологическому консалтингу, разработке и внедрению для предприятий для их успешной трансформации бизнеса. Глубокий опыт *LeverX* основан на успешном сотрудничестве с *SAP* в качестве Предпочтительного поставщика с 2004 года. Логотип команды *LeverX* показан на рисунке 1.2.



Рисунок 1.2 – Логотип команды *LeverX*

Темы привлечения талантов становятся ключевым направлением для того, чтобы быть ближе к существующим клиентам и помогать новым в решении их бизнес-задач и применении технологий. Компания *LeverX Group* проводит несколько инициатив по привлечению новых членов команды и сохранению нашей нынешней команды в:

1. налаживании сотрудничества с местными университетами
2. организацией мероприятий и встреч
3. оснащению своих офисов для организации комфортного и эффективного рабочего процесса.

Поскольку первой и главной целью компании *LeverX Group* является увеличение числа клиентов, необходима естественная миграция сотрудников между офисами. Тем не менее, эта тенденция продиктована потребностями бизнеса, такими как близость к клиентам, а не тенденциями переезда. Логотип команды *Emerline* показан на рисунке 1.3.



Рисунок 1.3 – Логотип команды *Emerline*

*LeverX Group* работает с компаниями и организациями для решения сложных задач современного делового мира. Каждый из наших партнеров является опытным, инновационным и зарекомендовавшим себя лидером в своих соответствующих областях.

* 1. Охрана труда и техника безопасности на рабочем месте

**программиста**

Охрана труда – система обеспечения безопасности жизни и здоровья работающих в процессе трудовой деятельности, включающая правовые, социально-экономические, организационные, технические, психофизиологические, санитарно-гигиенические, лечебно-профилактические, реабилитационные и иные мероприятия и средства [2].

Руководством компании уделяется большое внимание улучшению эргономики рабочих мест, обеспечению гигиены и совершенствованию организации труда, регламентации режимов труда и отдыха.

Вся деятельность в области охраны труда на предприятии регламентирована действующим законодательством Республики Беларусь, санитарными нормами и правилами, гигиеническими нормативами, предписаниями надзорных органов.

Важным фактором создания безопасных условий труда является оптимизация организации рабочих мест. Рабочее место, хорошо приспособленное к трудовой деятельности инженера, правильно и целесообразно организованное в отношении пространства, формы, размера, обеспечивает ему удобное положение при работе и высокую производительность труда при наименьшем физическом и психическом напряжении.

При правильной организации рабочего места производительность труда инженера возрастает с 8 до 20 процентов.

Рабочее место для выполнения работ в положении сидя организуется в соответствии с ГОСТ 12.2.032-78 «Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования» [3].

Согласно ГОСТ 12.2.032-78, конструкция рабочего места и взаимное расположение всех его элементов должно соответствовать антропометрическим, физическим и психологическим требованиям. Большое значение имеет также характер работы. В частности, при организации рабочего места программиста должны быть соблюдены следующие основные условия:

* оптимальное размещение оборудования, входящего и состав рабочего места;
* достаточное рабочее пространство, позволяющее осуществлять все необходимые движения и перемещения;
* уровень акустического шума не должен превышать допустимого значения.

Рабочая поза сидя вызывает минимальное утомление программиста. Рациональная планировка рабочего места предусматривает четкий порядок и постоянство размещения предметов, средств труда и документации. То, что требуется для выполнения работ чаще, расположено в зоне легкой досягаемости рабочего пространства.

Помещения для работы программиста должны иметь естественное и искусственное освещение.

Искусственное освещение в помещениях эксплуатации ВДТ и ПЭВМ должно осуществляться системой общего равномерного освещения. В административно-общественных помещениях, в случаях преимущественной работы с документами, допускается применение системы комбинированного освещения (к общему освещению дополнительно устанавливаются светильники местного освещения, предназначенные для освещения зоны расположения документов).

Освещенность на поверхности стола в тоне размещения рабочего документа должна быть 300-500 лк. Местное освещение не должно создавать бликов на поверхности экрана и увеличивать освещенность экрана более 300 лк.

В качестве источников света при искусственном освещении должны применяться преимущественно люминесцентные лампы.

Конструкция рабочего стола должна обеспечивать оптимальное размещение на рабочей поверхности используемого оборудования с учетом его количества и конструктивных особенностей (размер ВДТ и ПЭВМ, клавиатуры и др.), характера выполняемой работы. При этом допускается использование рабочих столов различных конструкций, отвечающих современным требованиям эргономики.

Конструкция рабочего стула (кресла) должна обеспечивать поддержание рациональной рабочей позы при работе, позволять изменять позу с целью снижения статического напряжения мышц шейно-плечевой области и спины для предупреждения развития утомления.

Тип рабочего стула (кресла) должен выбираться в зависимости от характера и продолжительности работы.

Рабочий стул (кресло) должен быть подъемно-поворотным и регулируемым по высоте и углам наклона сиденья и спинки, а также расстоянию спинки от переднего края сиденья, при этом регулировка каждого параметра должна быть независимой, легко осуществляемой и иметь надежную фиксацию.

Экран видеомонитора должен находиться от глаз пользователя на оптимальном расстоянии 600-700 мм, но не ближе 500 мм с учетом размеров алфавитно-цифровых знаков и символов.

Высота рабочей поверхности стола должна регулироваться в пределах 680-800 мм, при отсутствии такой возможности высота рабочей поверхности стола должна составлять 725 мм.

Рабочий стол должен иметь пространство для ног высотой не менее 600 мм, шириной – не менее 500 мм, глубиной на уровне колен – не менее 450 мм и на уровне вытянутых ног – не менее 650 мм.

Клавиатуру следует располагать на поверхности стола на расстоянии не менее чем 300 мм от края, обращенного к пользователю или на специальной, регулируемой по высоте рабочей поверхности, отделенной от основной столешницы.

Помимо требований к организации рабочего места СанПиН 9-131 РБ 2000 устанавливает требования к микроклимату рабочей зоны: влажности, температуре, скорости потока воздуха и пр.

Техника безопасности – это система организационных мероприятий и технических средств, предотвращающих воздействие на работающих опасных производственных факторов.

# ОБЗОР ТЕХНОЛОГИИ *ASP .NET* И *REST API*

* 1. Технология *ASP* .*NET*

С выходом первой версии .*NET* *Framework* около десяти лет назад в сфере проектирования программного обеспечения появилось совершенно новое направление. Вдохновленные наилучшими возможностями *Java*, *COM* и веб-технологий и обученные на ошибках и ограничениях прежних технологий, разработчики в *Microsoft* решили полностью обновить свою платформу для разработки.

В результате этого появился ряд удивительно совершенных технологий для выполнения всего, начиная от построения приложений *Windows* и заканчивая выполнением запросов в базах данных, и специально ориентированный на разработку веб-сайтов инструмент под названием *ASP*.*NET*.

Сегодня *ASP*.*NET* пользуется небывалой популярностью, но особо революционной технологией уже больше не является. И хотя базовые функциональные возможности, лежащие в основе *ASP*.*NET*, на удивление, выглядят точно так же, как и десять лет назад, разработчики из *Microsoft* добавили к ним еще кое-какие дополнительные средства и абстракции кодирования более высокого уровня [4].

Также появилось и еще как минимум одно новое направление, составляющее конкуренцию традиционному программированию с использованием *ASP*.*NET*, которое получило название *ASP*.*NET* *MVC*.

Когда платформа *ASP*.*NET* была выпущена впервые, от предыдущих продуктов Microsoft и конкурирующих платформ ее отличали семь ключевых фактов. Тем, кто перешел на *ASP*.*NET* с какой-то другой платформы для разработки веб-приложений, или еще никогда не программировал в .*NET* приложений для веб-сети, материал следующих разделов позволит быстро получить общее представление об *ASP*.*NET*.

Платформа .*NET* *Framework* делится на практически неповторимый ряд функциональных частей с десятками тысяч типов (в .*NET* так называются классы, структуры, интерфейсы и другие ключевые элементы программирования). Прежде чем пытаться программировать любое приложение .*NET*, необходимо сначала получить хотя бы общее представление об этих частях и о том, почему они организованы именно таким, а не каким-то другим образом.

То, как организована предлагаемая в .*NET* *Framework* обширная коллекция функциональности, программистам традиционных *Windows*-приложений, несомненно, покажется замечательным улучшением. Каждый из тысяч доступных в .*NET* *Framework* классов размещен в логическом иерархическом контейнере, который называется пространством имен (*namespace*).

В разных пространствах имен предоставляется разная функциональность, но все вместе они предлагают функциональные возможности для практически каждого аспекта области распределенной разработки, начиная с организации очередей сообщений и заканчивая обеспечением безопасности. Весь целиком этот обширный набор инструментов называется библиотекой классов (*class* *library*).

Интересно то, что способ, которым классы .*NET* *Framework* можно использовать в *ASP.NET*, ничем не отличается от того, которым они применяются в приложениях .*NET* любого другого типа (в том числе автономных *Windows*-приложениях, *Windows*-службах, утилитах командной строки и т.д.).

Хотя в .*NET* предлагаются ориентированные специально на *Windows*- и на веб-приложения классы для построения пользовательских интерфейсов, большинство возможностей .*NET* *Framework* (начиная с получения доступа к базам данных и закачивания поддержкой многопоточной обработки) допускается использовать в приложениях любого типа. Другими словами, в .*NET* разработчикам веб-приложений предлагаются те же самые инструменты, что и разработчикам многофункциональных клиентских приложений.

Подобно всем приложениям .*NET*, приложения *ASP*.*NET* всегда компилируются. На самом деле выполнение кода на *C*# или *Visual Basic* без предварительной компиляции просто невозможно.

Приложения *ASP*.*NET* в действительности проходят через два этапа компиляции.

На первом этапе написанный код на *C*# компилируется в код на промежуточном языке, который называется *MSIL* (*Microsoft* *Intermediate Language*), или просто *IL*. Этот первый этап как раз и является одной из главных причин, по которым в .*NET* могут использоваться самые разные языки.

Дело в том, что все языки .*NET* (в том числе *C#,* *Visual Basic* и многие другие), по сути, компилируются в практически идентичный код *IL*. Этот первый этап компиляции может происходить как автоматически при первом запросе страницы, так и выполняться заранее (это называется предварительной компиляцией). Скомпилированный файл с кодом на *IL* представляет собой сборку.

Второй этап компиляции происходит непосредственно перед фактическим выполнением страницы. На этом этапе код *IL* компилируется в код на низкоуровневом машинном языке. Называется этот этап оперативной (*Just*-*In*-*Time*) компиляцией и выглядит одинаково для всех приложений .*NET* (включая, например, *Windows*-приложения).

На рисунке 2.1 показан этот состоящий из двух этапов процесс компиляции.

Процесс компиляции в .*NET* разделен на два этапа для предоставления разработчикам как можно большего удобства и мобильности. Перед тем, как создавать код на низкоуровневом машинном языке, компилятор должен знать, в какой операционной системе и на каком базовом оборудовании будет выполняться приложение (например, будет это 32- или 64-разрядная ОС *Windows*). Благодаря выполнению таких двух этапов компиляции, можно создавать скомпилированную сборку с кодом .*NET* и по-прежнему распространять ее среди более, чем одной, платформы.

Технология *ASP* предоставляет довольно слабую объектную модель. Она поддерживает весьма небольшой набор объектов, которые являются просто тонким уровнем, скрывающим детали *HTTP* и *HTML*.

Напротив, *ASP*.*NET* является самой настоящей объектно-ориентированной технологией. Она не только предоставляет коду полный доступ ко всем объектам .*NET* *Framework*, но и позволяет использовать все концепции объектно-ориентированного программирования (ООП). Например, она позволяет создавать пригодные для повторного использования классы, стандартизировать код с помощью интерфейсов, расширять существующие классы за счет наследования и объединять полезные функциональные возможности в распространяемый скомпилированный компонент.

Одним из наилучших примеров поддержки объектно-ориентированного поведения в *ASP*.*NET* являются серверные элементы управления. Эти элементы управления представляют собой инкапсуляцию в миниатюре. Разработчики могут манипулировать объектами элементов управления программно с использованием кода для настройки их внешнего вида, предоставления отображаемых данных и даже реагирования на события.

Вся низкоуровневая *HTML*-разметка, которую визуализируют эти элементы управления, скрывается из вида. Вместо того чтобы вынуждать разработчика писать низкоуровневый код *HTML*-разметки вручную, объекты этих элементов управления сами преобразуются в соответствующие *HTML*-элементы непосредственно перед отправкой веб-сервером страницы клиенту.

Таким образом, серверные элементы управления в *ASP*.*NET* позволяют абстрагироваться от низкоуровневых деталей программирования *HTML* и *HTTP*.

Одним из самых больших испытаний для разработчиков веб-приложений является необходимость в обеспечении поддержки множества различных браузеров. В разных браузерах, версиях и конфигурациях технологии *HTML*, *CSS* и *JavaScript* поддерживаются по-разному. Разработчики веб-приложений должны выбирать, визуализировать содержимое в соответствии с наименьшими общими характеристиками или добавлять фрагменты для принятия во внимание специфических особенностей популярных браузеров.

В *ASP*.*NET* эта проблема решена удивительно интеллектуальным способом. Хотя внутри страницы *ASP*.*NET* можно извлечь информацию о браузере клиента и его возможностях, в действительности в *ASP*.*NET* рекомендуется игнорировать такой подход и пользоваться развитым комплектом серверных веб-элементов управления.

Платформа .*NET* *Framework* включает собственную технологию доступа к данным — *ADO*.*NET*. Эта технология состоит из управляемых классов, позволяющих приложениям .*NET* подключаться к источникам данных (обычно реляционным базам данных), выполнять команды и управлять автономными данными. Маленькое чудо *ADO*.*NET* заключается в том, что эта технология позволяет писать более-менее одинаковый код для доступа к данным — как в веб-приложениях, так и в клиент-серверных настольных приложениях, и даже в однопользовательских приложениях, подключаемых к локальной базе данных.



Рисунок 2.1 – Процесс компиляции

Эти серверные элементы управления генерируют разметку адаптивным образом, принимая во внимание все возможности клиента. Примером могут служить предлагаемые в *ASP*.*NET* элементы управления верификацией, которые используют *JavaScript* и *DHTML* (динамический *HTML*) для улучшения своего поведения в случае, если они поддерживается клиентом.

* 1. Технология *REST API*

Интерфейсы прикладного программирования (*API*) предоставляют платформу и среду для приложений, которые позволяют им общаться и понимать друг друга. *API*-интерфейсы определяют способ, которым информация, передаваемая по платформам, структурирована так, чтобы приложения могли обмениваться данными и информацией.

*REST* – это стиль архитектуры *API* для распределенных систем. *REST* определяет, как данные представляются клиенту в удобном для него формате. Обмен данными происходит в формате *JSON* или *XML* (хотя сегодня более популярен формат *JSON*)[5].

Обмен данными в Интернете происходит с помощью *TCP*/*IP* (протокол управления передачей/интернет-протокол). *TCP*/*IP* – это набор протоколов связи, который описывает, как взаимодействует огромное количество компьютеров, подключенных к Интернету.

*TCP*/*IP* обеспечивает сквозную связь, которая определяет, как данные обмениваются через Интернет, как данные разбиваются на пакеты, как пакеты кодируются, адресуются, маршрутизируются и принимаются в месте назначения. Думайте об этом как о гигантской компании доставки почты, которая способна доставлять ваши посылки в любую точку мира невероятно быстро. *TCP*/*IP* определяет правила для упаковки каждой из посылок, чтобы они могли добраться до нужного человека без путаницы.

*TCP*/*IP* использует стандартную модель связи клиент-сервер, когда клиент (компьютерное устройство) запрашивает ресурс у сервера (возможно, гораздо более крупного компьютерного устройства в удаленном месте). Соединения с использованием *TCP*/*IP* не сохраняют состояния – каждый запрос от клиента к серверу рассматривается как новый, сервер никогда не запоминает клиента. Это освобождает ресурсы на сервере, чтобы сделать его быстрее, и он быстрее отвечал на несколько запросов.

*REST*, *Representational State Transfer*, является архитектурным стилем для обеспечения стандартов между компьютерными системами в сети, что облегчает для систем обмен данными друг с другом. Системы, отвечающие требованиям *REST* и часто называемые *RESTful*, характеризуются тем, что не имеют сохранения состояния и разделяют интересы клиента и сервера. Мы рассмотрим, что означают эти термины и почему они являются полезными для услуг в Интернете.

В архитектурном стиле *REST* реализация клиента и реализация сервера могут быть выполнены независимо друг от друга. Это означает, что код на стороне клиента может быть изменен в любое время без ущерба для работы сервера, а код на стороне сервера может быть изменен без влияния на работу клиента. На рисунке 2.2 показана схема работы *REST API*.

До тех пор, пока каждая сторона знает, какой формат сообщений следует направлять другой стороне, они могут храниться модульно и раздельно. Отделяя задачи пользовательского интерфейса от задач хранения данных, мы повышаем гибкость интерфейса между платформами и улучшаем расширяемость за счёт упрощения компонентов сервера. Кроме того, разделение позволяет каждому компоненту развиваться независимо.

Используя интерфейс *REST*, различные клиенты попадают в одни и те же конечные точки *REST*, выполняют те же действия и получают одинаковые ответы. Системы, которые следуют парадигме *REST*, не имеют сохранения состояния, что означает, что серверу не нужно знать о состоянии клиента и наоборот. Таким образом, и сервер, и клиент могут понять любое полученное сообщение, даже не увидев предыдущих сообщений. Это отсутствие сохранения состояния обеспечивается за счёт использования ресурсов, а не команд. Они описывают любые объекты, документы или вещи, которые могут потребоваться для хранения или отправки в другие службы.

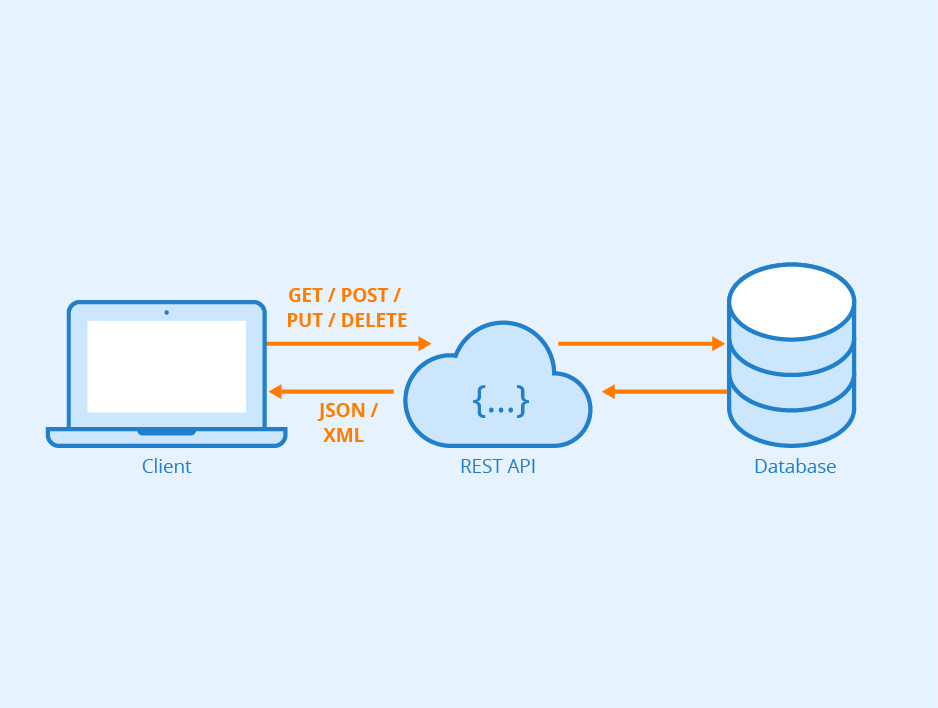


Рисунок 2.2 – Схема работы *REST API*

Системы, которые следуют парадигме *REST*, не имеют сохранения состояния, что означает, что серверу не нужно знать о состоянии клиента и наоборот. Таким образом, и сервер, и клиент могут понять любое полученное сообщение, даже не увидев предыдущих сообщений. Это отсутствие сохранения состояния обеспечивается за счёт использования ресурсов, а не команд. Они описывают любые объекты, документы или вещи, которые могут потребоваться для хранения или отправки в другие службы.

Эти ограничения помогают *RESTful*-приложениям достигать надёжности, быстрой производительности и расширяемости, как компонентам, которые могут быть управляемы, обновлены и повторно использованы, не затрагивая систему в целом даже во время ее работы.

# РАЗРАБОТКА *WEB API*

* 1. Постановка задачи

Программной реализацией является интерфейс, написанный с помощью *ASP* .*NET* *Core*. который позволяет отдавать данные в отличном от *HTML* формате, благодаря чему им удобно пользоваться при написании собственных приложений. Основным подходом приложения является *N-layer* архитектура [6]. Главная идея этого подхода заключается в том, что приложение делится на три части:

1. уровень доступа к данным
2. уровень бизнес-логики
3. уровень представления

Общая схема такой архитектуры показана на рисунке 3.1.

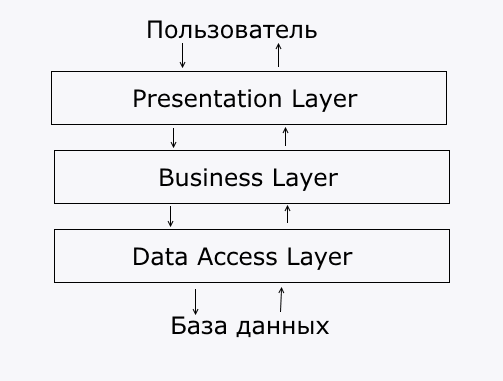


Рисунок 3.1 – Схема N-layer архитектуры

Тестировать приложение можно с помощью *Swagger*, это открытая *API*, созданная *Swashbuckle* для того, чтобы программистам бэк-энда не требовалось изучать фронт-энд, чтобы продемонстрировать работу своего кода [7].

* 1. Уровень доступа к данным

Уровень доступа к данным (или же *Data Access Layer*) хранит модели, описывающие используемые сущности, также здесь размещаются специфичные классы для работы с разными технологиями доступа к данным, например, класс контекста данных *Entity Framework*. Здесь также хранятся репозитории, через которые уровень бизнес-логики взаимодействует с базой данных.

При этом надо отметить, что крайние уровни не могут взаимодействовать между собой, то есть уровень представления не может напрямую обращаться к базе данных и даже к уровню доступа к данным, а только через уровень бизнес-логики.

Уровень доступа к данным не зависит от других уровней, уровень бизнес-логики зависит от уровня доступа к данным, а уровень представления - от уровня бизнес-логики.

Компоненты, как правило, должны быть слабо связанными (*loose* *coupling*), поэтому неотъемлемым звеном многоуровневых приложений является внедрение зависимостей.

В реализуемом *Web API* слой доступа к данным является отдельным проектом, содержащим модели, интерфейс, контекст данных и репозитории.

Модели являются кодовым представлением реальной сущности, например *Product*.*cs* и *ProductType*.*cs*, пример реализации модели продукта показан на рисунке 3.2.

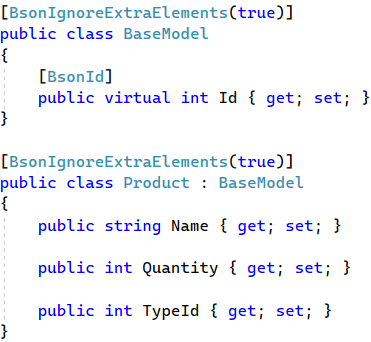


Рисунок 3.2 – Модель продукта

На рисунке видно, что модель продукта наследуется от класса базовой модели. Это нужно для того, чтобы сделать обобщенный репозиторий, что в свою очередь избавляет от необходимости копирования и повторения кода.

Интерфейс *IRepository*.*cs* в слое доступа к данным лишь задает методы, которые должен реализовать репозиторий. Он требует реализовать *CRUD*-операции – операции создания, чтения, обновления и удаления. Интерфейс требует реализовать методы чтения всех записей, чтения одной записи с определенным *id*, создания записи в таблице, обновления записи и удаления записи.

Класс контекста *ApplicationDbContext*.*cs* базы данных позволяет взаимодействовать с базой данных, используя сущностную модель классов. Этот контекст позволяет создавать и выполнять запросы, отслеживать изменения в объектах и отображать эти изменения на базу данных. Технологии *EntityFramework* требуется этот контекст для генерации запросов к базе данных. Класс *ObjectContext* взаимодействует с другими важными классами платформы *Entity Framework*, например, с классом *ObjectSet* – его объекты можно использовать в качестве коллекций данных из таблиц, *ObjectQuery* – является ядром создания запроса и т.д.

Последним основным компонентом слоя доступа к данным являются репозитории. На рисунке показана схема репозиториев Репозиторий – это коллекция. Коллекция, которая содержит сущности и может фильтровать и возвращать результат обратно в зависимости от требований вашего приложения. Где и как он хранит эти объекты является деталью реализации. На рисунке 3.3 показана схема паттерна “Репозиторий”.

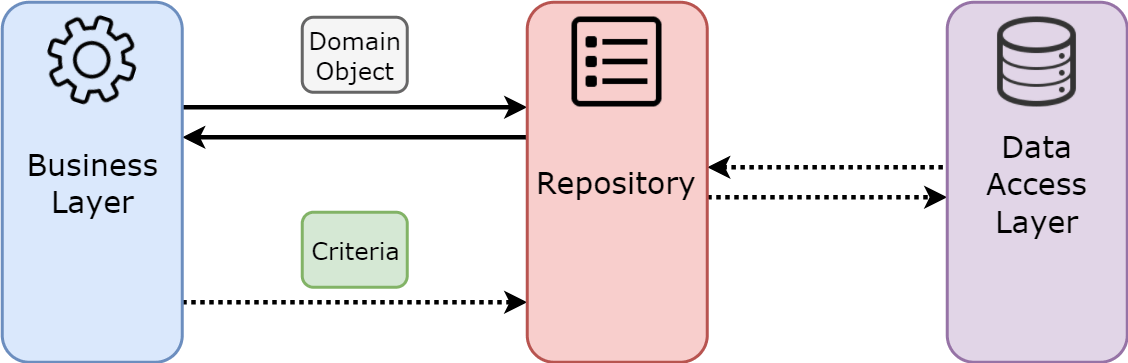


Рисунок 3.3 – Схема паттерна “Репозиторий”

В данном проекте уровня доступа к данным были реализованы репозитории *EntityRepository*.*cs* и *MongoRepository*.*cs* для доступа к двум различным базам данных – *MongoDb* и *MS SQL Server*. Оба репозитория являются обобщенными, что позволяет не создавать новые репозитории при добавлении каких-либо сущностей (таблиц) в базу данных. Доступ к *MS SQL Server* обеспечивает *Microsoft*.*EntityFrameworkCore*, который генерирует запросы на основе описанных методов. Доступ к *MongoDb* помогает реализовать *MongoDb*.*Driver*, которые тоже содержит методы, с помощью них можно получить сущности, хранящиеся в базе данных.

* 1. Уровень бизнес-логики

Уровень бизнес-логики содержит набор компонентов, которые отвечают за обработку полученных от уровня представления данных, реализует всю необходимую логику приложения, все вычисления, взаимодействует с базой данных и передает уровню представления результат. По сути, этот слой является связующим звеном между слоем, с которым взаимодействует база данных, и слоем, с которым взаимодействует пользователь.

*Business Logic Layer* или бизнес-уровень инкапсулирует всю бизнес-логику, все необходимые вычисления, получает объекты из уровня доступа к данным и передает их на уровень представления, либо, наоборот, получает данные с уровня представления и передает их на уровень данных.

На этом уровне данные, которые пользователь может занести в базу данных, должны быть проверены на валидность – пройти валидацию. Например, пользователь не сможет занести в колонку “Количество” отрицательное значение или вовсе строку. В бизнес-логике реализованного *Web API* присутствуют интерфейс службы, сама служба, общий валидатор и конкретный валидатор для каждой таблицы, так как требования к полям могут отличаться и их нельзя обобщить.

Интерфейс *IService*.*cs* лишь задает методы, которые должна реализовать служба. Служба должна реализовать те же методы, что и репозиторий, только еще и добавить валидацию.

Обобщенный валидатор *Validator*.*cs* использует *FluentValidation*, который в свою очередь предоставляет методы ограничений и правил, которые должны соблюдать объекты, чтобы пройти валидацию и попасть в базу данных. В этом классе присутствует метод, проверяющий, существует ли вообще объект, что понадобится для вставки и обновления записей. Именно для этого нам пригодилась базовая модель, имеющая лишь одно поле *Id*. На валидаторе стоит ограничение, чтобы он работал с любым типом, наследующимся от *BaseModel*.*cs*.

Конкретные валидаторы *ProductValidator*.*cs* и *ProductTypeValidator*.*cs* в конструкторе имеют правила, чтобы строки названий не были пустыми, а также чтобы количественные характеристики были большими или равными нуля. Также валидаторы имеют различные сеты правил на случай создания и добавления записей. Это значит, что при добавлении записи идет проверка, является ли имя уникальным, а при обновлении записи идет проверка, существует ли вообще такая запись, чтобы ее обновить, а также проверка, является ли новое имя записи уникальным. Эти проверки реализованы с помощью *LINQ* запросов, что уменьшает количество кода, а также помогает быстрее его понять.

Служба *Service*.cs отвечает за реализацию *CRUD*-операций, совмещенных с валидацией. Если валидация пройдена, то есть данные валидны, вызывается метод репозитория и данные уже попадают в базу. При ошибке валидации выбрасывается исключение с сообщением, в чем проблема.

* 1. Уровень представления

*Presentation layer* (уровень представления) – это тот уровень, с которым непосредственно взаимодействует пользователь. Этот уровень включает компоненты пользовательского интерфейса, механизм получения ввода от пользователя. Применительно к *ASP*.*NET* *Core* на данном уровне расположены представления и все те компоненты, который составляют пользовательский интерфейс (в данном случае *Swagger*), а также модели представлений, контроллеры, объекты контекста запроса.

В уровне представления присутствуют настройки запуска *launchSettings*.*json* и настройки приложения *appSettings*.*json*. В настройках запуска задается *URL* приложения и *URL* при запуске. Например, начальный *URL* адрес выглядит как *https*:/*localhost*:5002/*swagger*, чтобы сразу переходить к тестированию приложения с помощью *Swagger’а*. На рисунке 3.4 показано, как выглядит начальное окно без настройки *URL* при старте. Это шаблонный стиль *ASP*.*NET* *Core*, который обычно используют для перед написанием дизайна.

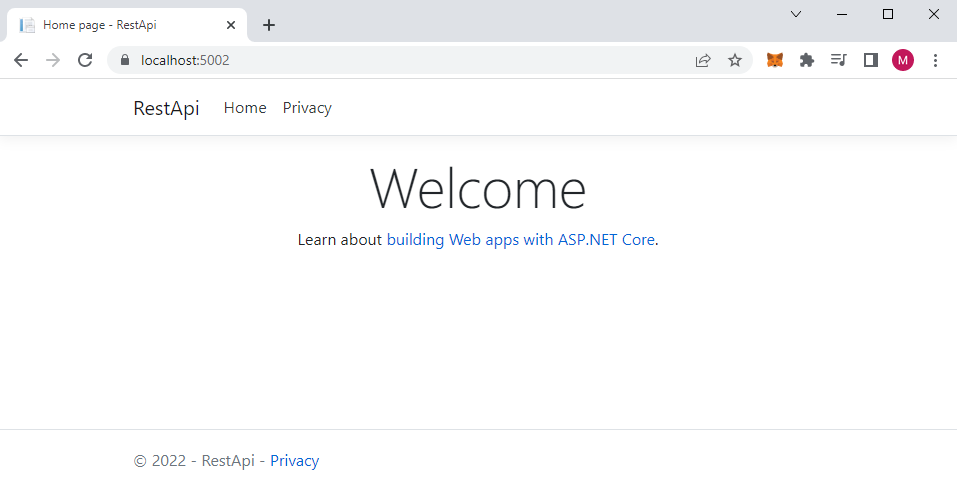


Рисунок 3.4 – Стартовое окно по умолчанию в *ASP*.*NET* *Core*

В настройках приложения указаны строки подключения к базам данных, а конкретно к локальной базе данных Mongo и локальной базе данных *MS SQL Server*. Пример настроек строк показан на рисунке 3.5.

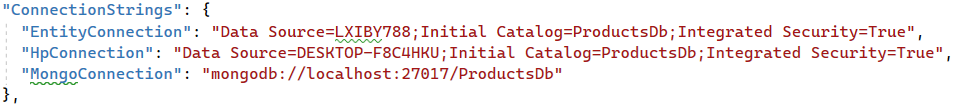


Рисунок 3.5 – Строки подключения в настройках *appSettings*.*json*

Эти настройки представлены в виде *JSON*-файла, который удобно читается системным классом *Microsoft*.*Configuration*.

Слой представления содержит контроллеры *ProductController*.*cs* и *ProductTypeController*.*cs*, которые содержат эндпоинты (методы) для чтения, записи, обновления и удаления. Они отвечают за возврат результата операции в виде *JSON*-файла, который отображается в графическом интерфейсе *Swagger’а*.

Также в проекте по умолчанию созданы *Pages*, или же представления, для отображения информации в браузере на основе *html*. Представления имеют расширения .*cshtml*, которое позволяет совмещать код *C*# и *HTML*.

Приложение запускает класс *Program*.*cs*, который в свою очередь берет настройки из класса *Startup*.*cs*, в котором присутствуют два метода – *Configure* и *ConfigureServices*. В методе конфигурации сервисов к *services* добавляются службы, связи, контроллеры, а также связи. Можно сказать, этот метод показывает приложению, какие службы будут использоваться. А метод конфигурации приложения в свою очередь уже использует эти службы.

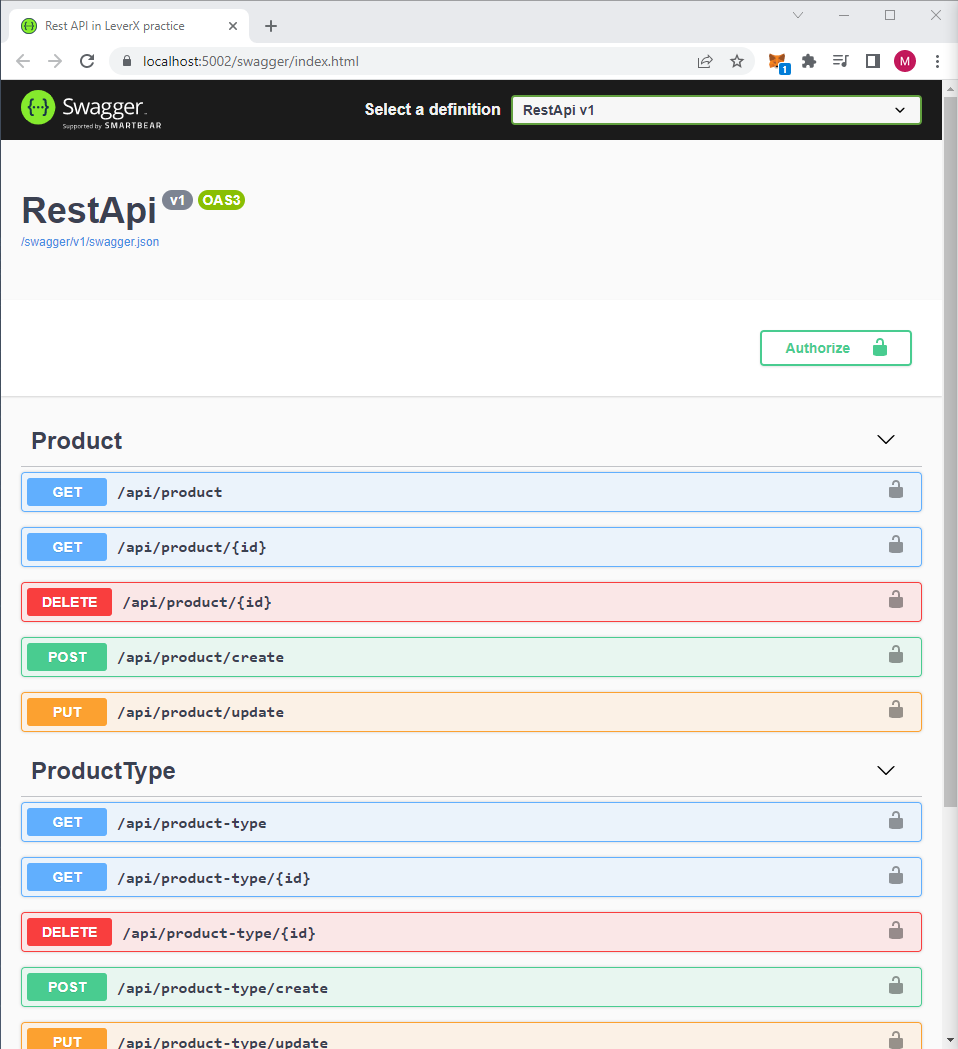
В конфигурации сервисов добавлены методы для внедрения зависимостей *AddTransient*<>(), которые задают конкретные реализации интерфейсов, что будут использованы в месте, где нужно подставить интерфейс. Например, в данном приложении прописаны такие связи как: подставлять *ApplicationDbContext*.*cs* при использовании *DbContext*; подставлять конкретный *Validator.cs* при использовании интерфейса *IValidator*.*cs*; подставлять конкретный *Service*.*cs* при использовании интерфейса *IService*.*cs*; подставлять конкретный репозиторий (либо *EntityRepository*.*cs*, либо *MongoRepository*.*cs* в зависимости от того, с какой базой данных требуется работать) при использовании интерфейса *IRepository*.*cs.*

Также в классе *Startup*.*cs* объявляется, что будет использоваться *Swagger* и какие его настройки применять.

* 1. Верификация *Web API*

Работа с базами данных является основой задачей ведения бизнеса, ведь продавцы должны иметь информацию о том, кто и что сделал с каким-то товаром и так далее.

При запуске приложения запускается страница сваггера, на которой можно увидеть все эндпоинты, их *URL*, их доступность и много другой полезной информации для разработчика. На рисунке 3.6 показан начальный экран приложения.

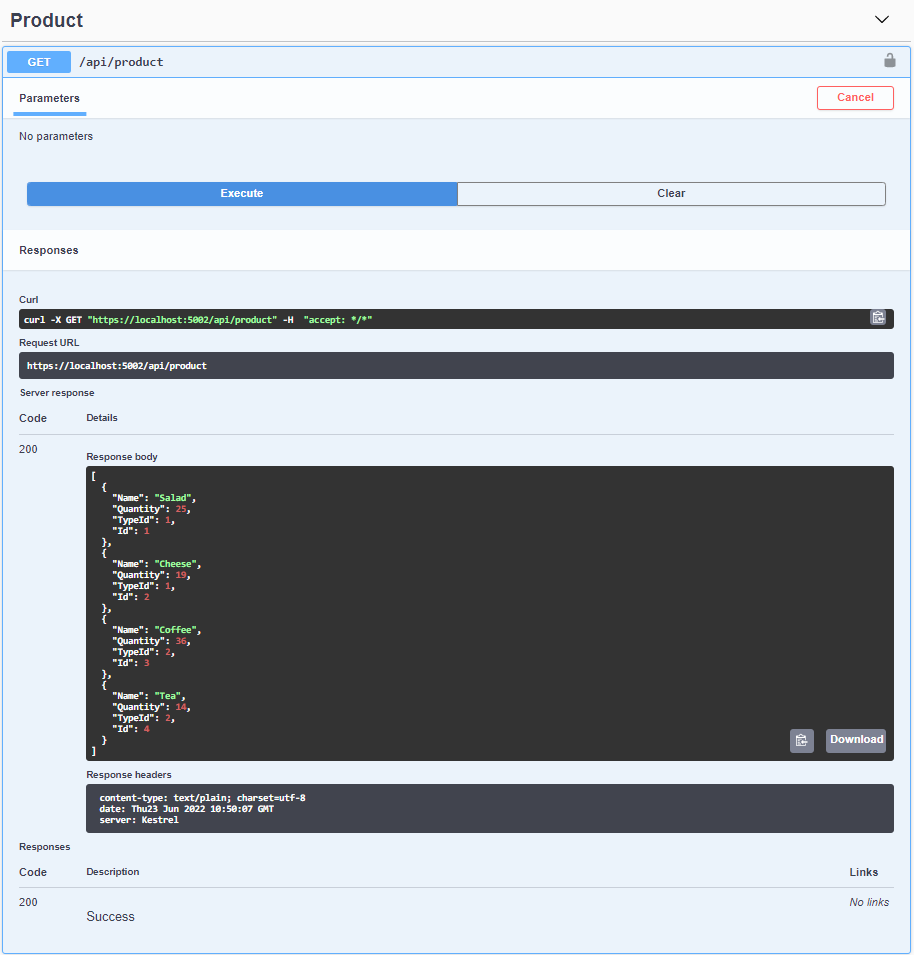


# 

# Рисунок 3.6 – Начальный экран приложения

# 

Операция *GetAll*, представленная по адресу /*api*/*product* получает из таблицы все объекты и отображает их на экране в виде *json*-файла. На рисунке 3.7 продемонстрированы все объекты таблицы *Products* базы данных *MS SQL Server*.

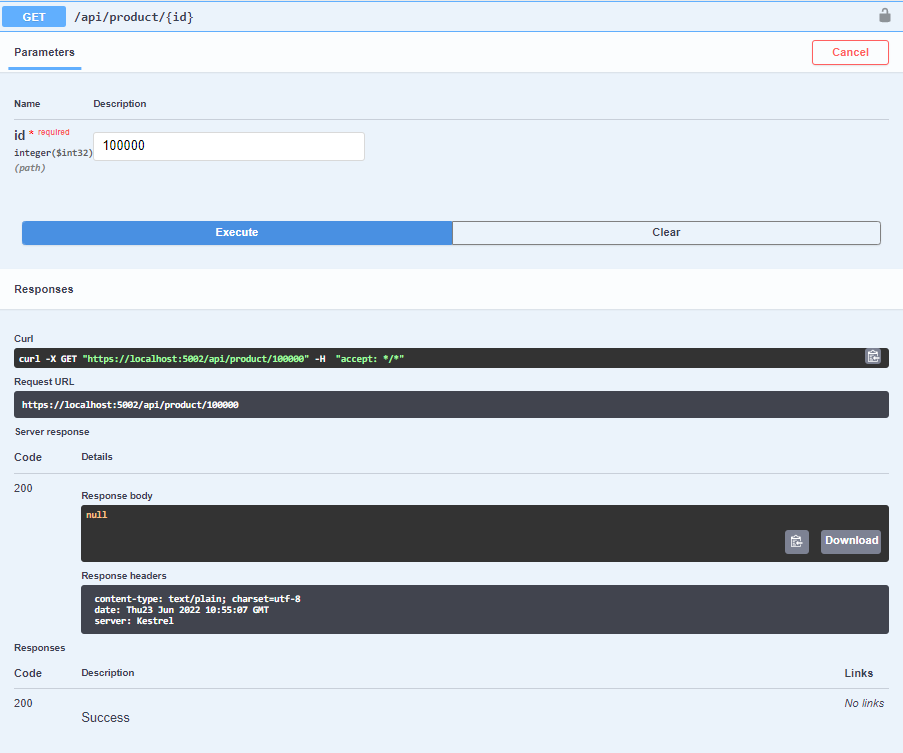


# 

# Рисунок 3.7 – Все объекты таблицы *Products* базы данных *MS SQL Server*

Помимо этого, отображается код, полученный при выполнении запроса. Если данные были получены и проблем нет, то код равен 200, а это значит, что операция выполнена успешно.

Эндпоинт получения с адресом /*api*/*product*/{*id*} получает данные по заданному *id*. Если *id* указан неверно, то есть объекта с таким *id* не существует в базе данных, то сваггер возвращает *null*-объект. Это продемонстрировано на рисунке 3.8.

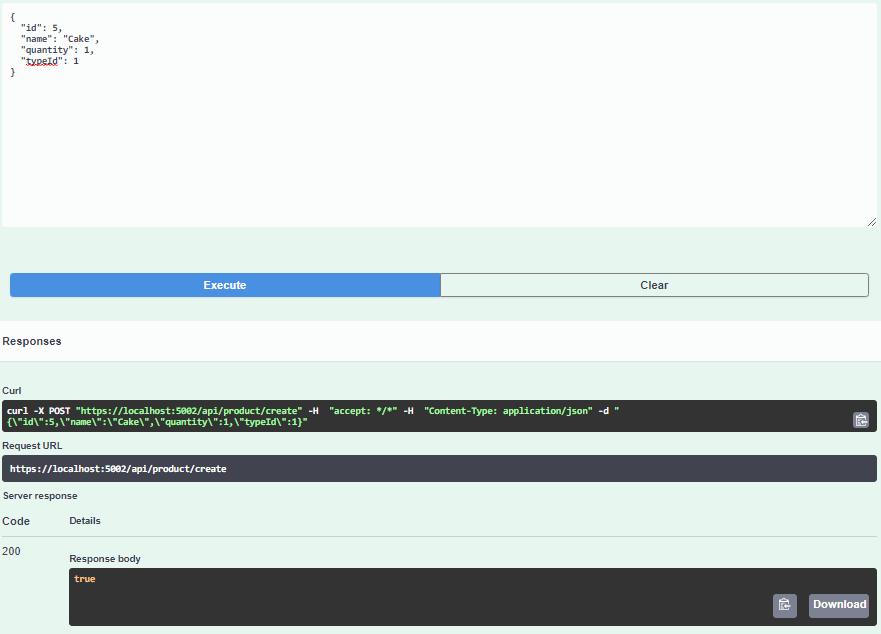


# 

# Рисунок 3.8 – Возврат пустого объекта при указании несуществующего *id*

Процедура удаления из базы данных происходит по тому же принципу, что и чтение по *id*. При указании несуществующего *id* возвращается ответ *false*, это означает, что никакой объект не был удален.

Добавление и обновление происходит с помощью отправки объекта базы данных, представленного в виде *json*. Это продемонстрировано на рисунке 3.9.



# 

# Рисунок 3.9 – Добавление объекта в базу данных

Остальные эндпоинты работают аналогичным образом и не требуют обзора.

# 

# ЗАКЛЮЧЕНИЕ

Прохождение производственной практики является важным элементом учебного процесса по подготовки специалиста в области программирования.

Во время её прохождения будущий программист применяет полученные в процессе обучения знания, умения и навыки на практике.

Основными задачами производственной практики являются:

* получения практического опыта работы в качестве программиста;
* улучшения качества профессиональной подготовки;
* закрепление полученных знаний по общим и специальным дисциплинам;
* проверка умения студентов пользоваться персональным компьютером.

Широкий охват отраслей, с которыми пришлось сталкиваться на практике, позволил лучше усвоить изученный теоретический материал, полученный на занятиях в колледже.

Были освоены некоторые тонкости применения ПО на практике, была изучена работа некоторых программ, подпрограммы, которые были необходимы при разработке индивидуального задания.

Практическая деятельность помогла научиться самостоятельно решать определенный круг задач, возникающих в ходе работы программиста.

В процессе прохождения технологической практики было разработано *Web API*, схожее с реальными задачами клиентов, поставленными перед компанией.

# Список использованных источников

1. Официальный сайт ООО «Леверекс Интернешнл» [Электронный ресурс]. Режим доступа: https://leverx.ru/company/about – Дата доступа: 02.07.2022.
2. Юридический словарь [Электронный ресурс]. Режим доступа: http://multilang.pravo.by/ru/Term/Index/525?langName=ru&ch=Все&size=25&page=4&type=3 – Дата доступа: 02.07.2022.
3. Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования : ГОСТ 12.2.032-78. – Введ. 01.01.1979 – Москва : ИПК Издательство стандартов, 2001. – 27с.
4. Основы *ASP*.*NET* [Электронный ресурс]. Режим доступа: https://professorweb.ru/my/ASP\_NET/base/level1/base\_aspnet\_index.php – Дата доступа: 02.07.2022.
5. Основы *REST*: теория и практика [Электронный ресурс]. Режим доступа: https://tproger.ru/articles/osnovy-rest-teorija-i-praktika/ – Дата доступа: 02.07.2021.
6. Многоуровневая архитектура [Электронный ресурс]. Режим доступа: https://metanit.com/sharp/mvc5/23.5.php – Дата доступа: 02.07.2022.
7. Проектирование *API* с помощью *Swagger* и *OpenAPI* / Джош Понелат. // Аллея науки. – 2017. – №16.

# ПРИЛОЖЕНИЕ А

(обязательное)

# Листинг программы

Листинг *IRepository.cs*:

using DataAccessLayer.Models;

using System.Collections.Generic;

namespace DataAccessLayer.Interfaces

{

public interface IRepository<T> where T : BaseModel

{

IEnumerable<T> GetAll();

T Get(int id);

void Create(T item);

void Update(T item);

void Delete(int id);

}

}

Листинг *BaseModel.cs*:

using MongoDB.Bson.Serialization.Attributes;

namespace DataAccessLayer.Models

{

[BsonIgnoreExtraElements(true)]

public class BaseModel

{

[BsonId]

public virtual int Id { get; set; }

}

}

Листинг *Product.cs*:

using MongoDB.Bson.Serialization.Attributes;

namespace DataAccessLayer.Models

{

[BsonIgnoreExtraElements(true)]

public class Product : BaseModel

{

public string Name { get; set; }

public int Quantity { get; set; }

public int TypeId { get; set; }

}

}

Листинг *ProductType.cs*:

using MongoDB.Bson.Serialization.Attributes;

namespace DataAccessLayer.Models

{

[BsonIgnoreExtraElements(true)]

public class ProductType : BaseModel

{

public string Name { get; set; }

}

}

Листинг *AppicationDbContext.cs*:

using DataAccessLayer.Models;

using Microsoft.EntityFrameworkCore;

namespace DataAccessLayer

{

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)

{

}

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

}

public DbSet<Product> Products { get; set; }

public DbSet<ProductType> ProductTypes { get; set; }

}

}

Листинг *EntityRepository.cs*:

using DataAccessLayer.Interfaces;

using Microsoft.EntityFrameworkCore;

using System.Collections.Generic;

using System.Linq;

using DataAccessLayer.Models;

namespace DataAccessLayer.Repositories

{

public class EntityRepository<T> : IRepository<T> where T : BaseModel

{

private readonly DbContext \_context;

public EntityRepository(DbContext context)

{

\_context = context;

}

public IEnumerable<T> GetAll()

{

var result = \_context.Set<T>().ToList();

return result;

}

public T Get(int id)

{

var result = \_context.Set<T>().Find(id);

return result;

}

public void Create(T item)

{

\_context.Set<T>().Add(item);

\_context.SaveChanges();

}

public void Update(T item)

{

\_context.Entry(item).State = EntityState.Modified;

\_context.SaveChanges();

}

public void Delete(int id)

{

var item = \_context.Set<T>().Find(id);

\_context.Set<T>().Remove(item);

\_context.SaveChanges();

}

}

}

Листинг *MongoRepository.cs*:

using DataAccessLayer.Interfaces;

using DataAccessLayer.Models;

using MongoDB.Bson;

using MongoDB.Driver;

using System.Collections.Generic;

namespace DataAccessLayer.Repositories

{

public class MongoRepository<T> : IRepository<T> where T : BaseModel

{

private readonly IMongoCollection<T> \_collection;

public MongoRepository()

{

string connectionString = "mongodb://localhost:27017/ProductsDb";

var connection = new MongoUrlBuilder(connectionString);

MongoClient client = new MongoClient(connectionString);

IMongoDatabase database = client.GetDatabase(connection.DatabaseName);

\_collection = database.GetCollection<T>($"{typeof(T).Name}s");

}

public IEnumerable<T> GetAll()

{

var result = \_collection.Find(new BsonDocument()).ToList();

return result;

}

public T Get(int id)

{

var filter = Builders<T>.Filter.Eq("\_id", id);

var result = \_collection.Find(filter).First();

return result;

}

public void Create(T item)

{

\_collection.InsertOne(item);

}

public void Update(T item)

{

\_collection.ReplaceOne(

new BsonDocument("\_id", item.Id),

item,

new ReplaceOptions() { IsUpsert = false });

}

public void Delete(int id)

{

var filter = Builders<T>.Filter.Eq("\_id", id);

\_collection.FindOneAndDelete(filter);

}

}

}

Листинг *IService.cs:*

using System.Collections.Generic;

namespace BusinessLogicLayer.Interfaces

{

public interface IService<T>

{

List<T> GetAll();

T Get(int id);

void Create(T item);

void Update(T item);

void Delete(int id);

}

}

Листинг *Service.cs:*

using BusinessLogicLayer.Interfaces;

using FluentValidation;

using System;

using System.Collections.Generic;

using System.Linq;

using DataAccessLayer.Interfaces;

using DataAccessLayer.Models;

namespace BusinessLogicLayer.Services

{

public class Service<T> : IService<T> where T : BaseModel

{

private readonly IRepository<T> \_repository;

private readonly IValidator<T> \_validator;

public Service(IRepository<T> repository, IValidator<T> validator)

{

\_repository = repository ?? throw new ArgumentNullException(nameof(repository));

\_validator = validator ?? throw new ArgumentNullException(nameof(validator));

}

public List<T> GetAll()

{

var entities = \_repository.GetAll();

return entities.ToList();

}

public T Get(int id)

{

var entity = \_repository.Get(id);

return entity;

}

public void Create(T entity)

{

\_validator.ValidateAndThrow(entity);

if (\_validator.Validate(entity, options => options.IncludeRuleSets("BeforeCreating")).IsValid)

\_repository.Create(entity);

}

public void Update(T entity)

{

\_validator.ValidateAndThrow(entity);

if (\_validator.Validate(entity, options => options.IncludeRuleSets("BeforeUpdating")).IsValid)

\_repository.Update(entity);

}

public void Delete(int id)

{

\_repository.Delete(id);

}

}

}

Листинг *ProductTypeValidator.cs:*

using DataAccessLayer.Interfaces;

using DataAccessLayer.Models;

using FluentValidation;

using System.Linq;

namespace BusinessLogicLayer.Validators

{

public class ProductTypeValidator : Validator<ProductType>

{

public ProductTypeValidator(IRepository<ProductType> repository) : base(repository)

{

\_repository = repository;

RuleFor(x => x.Name).NotNull().NotEmpty();

RuleSet("BeforeCreating", () =>

{

RuleFor(x => x).Must(x => IsNameUnique(x));

});

RuleSet("BeforeUpdating", () =>

{

RuleFor(x => x).Must(x => IsNameUnique(x));

RuleFor(x => x).Must(x => IsEntityExists(x));

});

}

private bool IsNameUnique(ProductType type)

{

var result = \_repository.GetAll().All(x => x.Name != type.Name);

return result;

}

}

}

Листинг *ProductValidator.cs:*

using DataAccessLayer.Interfaces;

using DataAccessLayer.Models;

using FluentValidation;

using System.Linq;

namespace BusinessLogicLayer.Validators

{

public class ProductValidator : Validator<Product>

{

public ProductValidator(IRepository<Product> repository) : base(repository)

{

\_repository = repository;

RuleFor(x => x.Quantity).GreaterThanOrEqualTo(0);

RuleFor(x => x.TypeId).GreaterThanOrEqualTo(0);

RuleFor(x => x.Name).NotNull().NotEmpty();

RuleSet("BeforeCreating", () => { RuleFor(x => x).Must(x => IsNameUnique(x)); });

RuleSet("BeforeUpdating", () =>

{

RuleFor(x => x).Must(x => IsNameUnique(x));

RuleFor(x => x).Must(x => IsEntityExists(x));

});

}

private bool IsNameUnique(Product product)

{

var result = \_repository.GetAll().All(x => x.Name != product.Name);

return result;

}

}

}

Листинг *Validator.cs:*

using DataAccessLayer.Interfaces;

using DataAccessLayer.Models;

using FluentValidation;

namespace BusinessLogicLayer.Validators

{

public class Validator<T> : AbstractValidator<T> where T : BaseModel

{

protected IRepository<T> \_repository;

public Validator(IRepository<T> repository)

{

\_repository = repository;

}

protected bool IsEntityExists(T entity)

{

var result = \_repository.Get(entity.Id);

if (result == null)

return false;

return true;

}

}

}

Листинг *Program.cs:*

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Hosting;

namespace RestApi

{

public class Program

{

public static void Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

}

}

Листинг *Startup.cs:*

using BusinessLogicLayer.Interfaces;

using BusinessLogicLayer.Services;

using BusinessLogicLayer.Validators;

using DataAccessLayer;

using DataAccessLayer.Interfaces;

using DataAccessLayer.Repositories;

using FluentValidation;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using Microsoft.OpenApi.Models;

using System;

using System.Collections.Generic;

using DataAccessLayer.Models;

using IdentityServer4.AccessTokenValidation;

using Swashbuckle.AspNetCore.SwaggerUI;

using Microsoft.AspNetCore.Http;

namespace RestApi

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; private set; }

// This method gets called by the runtime. Use this method to add services to the container.

public void ConfigureServices(IServiceCollection services)

{

services.AddRazorPages();

services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("EntityConnection")));

services.AddHttpClient();

services.AddTransient<DbContext, ApplicationDbContext>();

services.AddTransient<IValidator<Product>, ProductValidator>();

services.AddTransient<IValidator<ProductType>, ProductTypeValidator>();

services.AddTransient(typeof(IRepository<>), typeof(EntityRepository<>));

services.AddTransient(typeof(IService<>), typeof(Service<>));

services.AddControllers();

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "RestApi", Version = "v1" });

c.AddSecurityDefinition("oauth2", new OpenApiSecurityScheme

{

Type = SecuritySchemeType.OAuth2,

Flows = new OpenApiOAuthFlows

{

Password = new OpenApiOAuthFlow

{

TokenUrl = new Uri("https://localhost:44315/connect/token")

}

}

});

c.AddSecurityRequirement(new OpenApiSecurityRequirement

{

{

new OpenApiSecurityScheme

{

Reference = new OpenApiReference

{

Type = ReferenceType.SecurityScheme,

Id = "oauth2"

},

Scheme = "oauth2",

Name = "Bearer",

In = ParameterLocation.Header

},

new List<string>()

}

});

});

services.AddOptions().Configure<UserApiOptions>(builder => builder.UserApiAddress = Configuration["IdentityUrl"]);

services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

var builder = new ConfigurationBuilder()

.SetBasePath(env.ContentRootPath)

.AddJsonFile("appsettings.json");

builder.AddEnvironmentVariables();

Configuration = builder.Build();

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "RestApi v1");

c.DocumentTitle = "Rest API in LeverX practice";

c.DocExpansion(DocExpansion.List);

c.OAuthClientId("client\_id\_restapi");

c.OAuthClientSecret("client\_secret\_restapi");

});

}

else

{

app.UseExceptionHandler("/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapRazorPages();

endpoints.MapControllers();

});

}

}

}

Листинг *ProductController.cs:*

using BusinessLogicLayer.Interfaces;

using DataAccessLayer.Models;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using Newtonsoft.Json;

using System;

namespace RestApi.Controllers

{

[Route("api/product/")]

[ApiController]

//[Authorize]

public class ProductController : ControllerBase

{

private readonly IService<Product> \_service;

public ProductController(IService<Product> service)

{

\_service = service;

}

[HttpGet("")]

public Object GetAll()

{

var data = \_service.GetAll();

var json = JsonConvert.SerializeObject(data, Formatting.Indented,

new JsonSerializerSettings()

{

ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore

}

);

return json;

}

[HttpGet("{id}")]

public Object Get(int id)

{

var data = \_service.Get(id);

var json = JsonConvert.SerializeObject(data, Formatting.Indented,

new JsonSerializerSettings()

{

ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore

}

);

return json;

}

[HttpPost("create")]

public Object Create([FromBody] Product product)

{

try

{

\_service.Create(product);

return true;

}

catch (Exception)

{

return false;

}

}

[HttpPut("update")]

public bool Update(Product product)

{

try

{

\_service.Update(product);

return true;

}

catch (Exception)

{

return false;

}

}

[HttpDelete("{id}")]

public bool Delete(int id)

{

try

{

\_service.Delete(id);

return true;

}

catch (Exception)

{

return false;

}

}

}

}

Листинг *ProductTypeController.cs:*

using BusinessLogicLayer.Interfaces;

using DataAccessLayer.Models;

using Microsoft.AspNetCore.Mvc;

using Newtonsoft.Json;

using System;

using Microsoft.AspNetCore.Authorization;

namespace RestApi.Controllers

{

[Route("api/product-type")]

[ApiController]

public class ProductTypeController : ControllerBase

{

private readonly IService<ProductType> \_service;

public ProductTypeController(IService<ProductType> service)

{

\_service = service;

}

[HttpGet("")]

public Object GetAll()

{

var data = \_service.GetAll();

var json = JsonConvert.SerializeObject(data, Formatting.Indented,

new JsonSerializerSettings()

{

ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore

}

);

return json;

}

[HttpGet("{id}")]

public Object Get(int id)

{

var data = \_service.Get(id);

var json = JsonConvert.SerializeObject(data, Formatting.Indented,

new JsonSerializerSettings()

{

ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore

}

);

return json;

}

[HttpPost("create")]

public Object Create([FromBody] ProductType productType)

{

try

{

\_service.Create(productType);

return true;

}

catch (Exception)

{

return false;

}

}

[HttpPut("update")]

public bool Update(ProductType productType)

{

try

{

\_service.Update(productType);

return true;

}

catch (Exception)

{

return false;

}

}

[HttpDelete("{id}")]

public bool Delete(int id)

{

try

{

\_service.Delete(id);

return true;

}

catch (Exception)

{

return false;

}

}

}

}