# 1

# Анатомия небезопасного приложения

Безопасность — пожалуй, один из самых важных архитектурных компонентов любого веб-приложения, написанного в XXI веке. В эпоху, когда вредоносные программы, преступники и недобросовестные сотрудники активно проверяют программное обеспечение на наличие *эксплойтов*, грамотное и комплексное применение средств безопасности — ключевой элемент любого проекта, за который вы будете отвечать.

Эта книга построена по принципу, который, на наш взгляд, обеспечивает удобную основу для изучения сложной темы: она предлагает рассмотреть веб-приложение на базе Spring 6 и разобраться в ключевых концепциях и стратегиях его защиты с помощью Spring Security 6. Мы дополняем этот подход, приводя примеры кода для каждой главы в виде готовых веб-приложений.

В этой главе мы рассмотрим пример сценария, который покажет несколько распространенных уязвимостей в системе безопасности. Наше путешествие начнется с изучения фундаментальных принципов безопасного программирования. В следующих главах мы уделим внимание изучению распространенных уязвимостей, таких как *SQL-инъекция*, *межсайтовый скриптинг (XSS)* и *подделка межсайтовых запросов (CSRF)*.

Если вы уже работаете со Spring Security или хотите перейти от базового применения к более сложным сценариям, эта книга обязательно окажется полезной.

В этой главе мы рассмотрим следующие темы:

- стили архитектуры программного обеспечения;
- аудит безопасности;
- решение проблем, которые выявил аудит безопасности.

Прежде чем углубляться в последующие разделы, рекомендуем получить базовое представление о фреймворке Spring. К концу этой главы вы узнаете, что делает приложение уязвимым для атак и какие основные механизмы безопасности помогут его защитить.

Если вы уже знакомы с базовой терминологией безопасности, можете перейти к главе 2 «Начало работы со Spring Security», в которой мы начинаем использовать базовую функциональность фреймворка.

# Стили архитектуры программного обеспечения

Многие компании сейчас приобретают вычислительные возможности у онлайн-платформ облачных сервисов и полагаются на облачные решения при разработке большинства приложений. Этот переход повлек за собой трансформацию дизайна приложений (рис. 1.1).

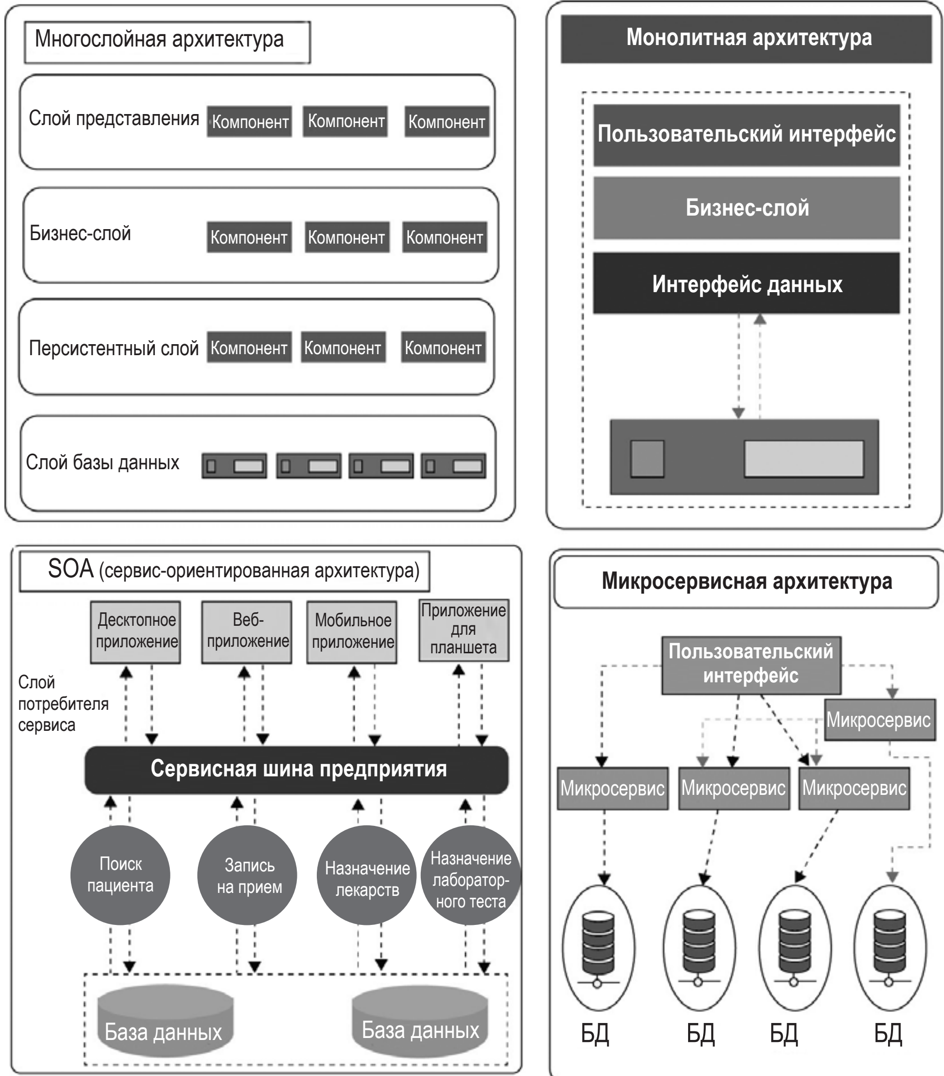


**Рис. 1.1.** Сравнение архитектур: монолитной, многослойной, сервис-ориентированной (SOA) и микросервисной

Выбор наиболее подходящей архитектуры приложения зависит от требований конкретного бизнеса. Мы рассмотрим четыре варианта архитектуры, разработанных для облегчения цифровой трансформации и адаптированных к общим потребностям бизнеса.

### Монолитная архитектура

Традиционная архитектура, в которой все приложение построено как единый и тесно интегрированный объект.

Несмотря на то что изначально его легко разрабатывать и развертывать, по мере расширения проекта могут возникнуть проблемы с масштабированием и поддержкой.

### N-уровневая архитектура (многослойная архитектура)

N-уровневая архитектура, или иерархическая структура с отдельными слоями, — это подход к проектированию программных систем, который организует приложение на нескольких уровнях, обычно четырех: *слой представления*, *бизнес-слой*, *персистентный слой* и *слой базы данных*. Эта архитектурная модель обычно используется в корпоративных приложениях, чтобы облегчить сопровождаемость за счет разделения на отдельные компоненты и поощрения модульной разработки. У каждого слоя есть определенные функции.

Шаблон проектирования *Model-View-Controller (MVC)* разделяет приложение на три взаимосвязанных компонента: модель (данные и бизнес-логика), представление (пользовательский интерфейс) и контроллер (обрабатывает вводимые пользователем данные и соответствующим образом обновляет модель и представление).

Благодаря такому разделению приложение легче масштабировать, обслуживать и адаптировать к меняющимся бизнес-требованиям.

### Сервис-ориентированная архитектура (SOA)

*Сервис-ориентированный шаблон*, также известный как *сервис-ориентированная архитектура (Service-Oriented Architecture, SOA)*, — это стиль архитектуры, в котором приложение представляет собой набор слабо связанных и независимо развертываемых сервисов.

До появления SOA в конце 1990-х годов подключение приложения к сервисам, размещенным в другой системе, было сложным процессом, требующим точечной интеграции.

### Микросервисная архитектура

Микросервисы основаны на SOA, но SOA — это не то же самое, что микросервисы.

Этот стиль архитектуры предполагает разделение приложения на небольшие автономные сервисы, взаимодействующие через API. Он обеспечивает масштабируемость, гибкость и упрощает обслуживание, но создает сложности, связанные с управлением распределенными системами.

> **Примечание**
> Исторически сложилось так, что основной упор делался на функциональность и актуальность, но сейчас большинство приложений, ориентированных на потребителя, переходят на модель *программного обеспечения как услуги (Soft ware-as-a-Service, SaaS)* и цифровые платформы. Акцент в проектировании приложений смещается в сторону улучшения пользовательского опыта, отказа от сохранения состояния и приоритета гибкости.

## Выбор между традиционными веб-приложениями и одностраничными приложениями

В современном мире есть два основных подхода к созданию веб-приложений: обычные веб-приложения, которые выполняют большую часть логики приложения на сервере, и *одностраничные приложения (single-page applications, SPA)*, которые обрабатывают большую часть логики пользовательского интерфейса в веб-браузере, а связь с веб-сервером осуществляется в основном через веб-интерфейсы. Возможен и альтернативный гибридный подход, при котором одно или несколько многофункциональных подприложений, или микрофронтендов, подобных SPA, размещаются внутри более крупного традиционного веб-приложения.

Выбирайте традиционное веб-приложение, если:

- требования к клиентской части вашего приложения просты или сводятся к режиму «только для чтения»;
- приложение должно работать в браузерах, не поддерживающих JavaScript;
- приложение предназначено для широкой аудитории и должно быть видно поисковым системам и доступно при переходе по ссылкам.

Выбирайте SPA, если:

- приложению требуется сложный пользовательский интерфейс со множеством функций;
- ваша команда разработчиков хорошо разбирается в JavaScript, Angular, ReactJS, VueJS, TypeScript или WebAssembly;
- приложение уже предоставляет API для других клиентов — внутренних или внешних.

Улучшения пользовательского опыта, обеспечиваемые подходом SPA, следует внимательно сопоставить с этими факторами.

В следующих разделах книги мы будем использовать традиционное приложение Spring MVC в качестве примера для иллюстрации различных принципов безопасности.

> **Примечание**
> Важно отметить, что эти принципы безопасности применимы ко всем стилям архитектуры, рассмотренным в этой главе.

# Аудит безопасности

Вы разработчик программного обеспечения для онлайн-календаря «Губка Боб Круглые Штаны» (*Jim Bob Circle Pants Online Calendar — JBCP Calendar,* `JBCPCalendar.com`) и ранним утром, выпив половину своей первой чашки кофе, получаете письмо от своего руководителя:

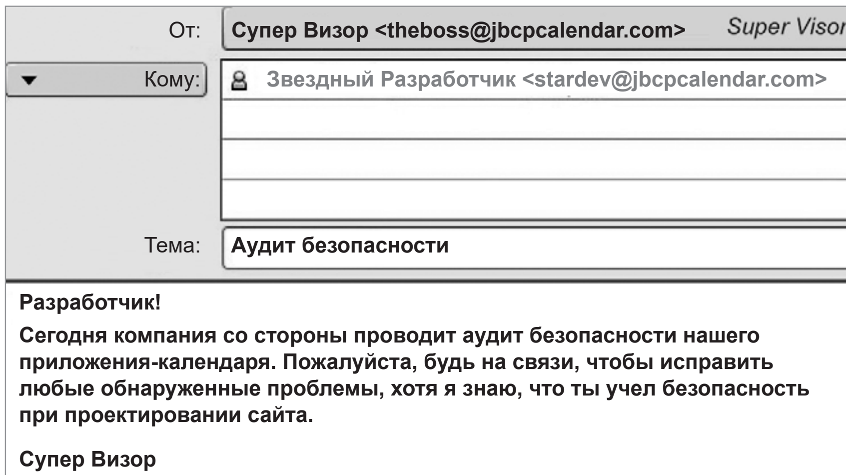


**Рис. 1.2.** Мейл от руководителя

Что? Вы не думали о безопасности, когда разрабатывали приложение? На самом деле на данный момент вы даже не уверены, что понимаете, что такое *аудит безопасности*? Похоже, вам придется многому научиться у аудиторов! Далее в этой главе мы рассмотрим, что такое аудит и какими могут быть его результаты. Для начала давайте уделим немного времени изучению приложения, которое мы проверяем.

## Изучение примера приложения

По мере чтения этой книги мы будем работать с вымышленным сценарием, но дизайн приложения и изменения, которые мы будем в него вносить, основаны

на реальном использовании приложений на базе Spring. Приложение-календарь позволяет пользователям создавать и просматривать события (рис. 1.3).

Мой календарь | Добро пожаловать | Все события | Мои события | Создать событие | H2

Информация о событии

Email участника

Дата/время события (гггг-мм-дд чч:мм)

Краткая информация

Описание

Создать | Автозаполнение

**Рис. 1.3.** Информация о событии в календаре

После ввода данных для нового события вы увидите такой экран:

Мой календарь | Добро пожаловать | Мои события | Создать событие | H2

Ниже список событий пользователя user1@example.com. При применении контроля доступа здесь появится список событий зарегистрированного пользователя.

Создать событие

| Дата/Время | Организатор | Участник | Краткая информация |
|---|---|---|---|
| 2023-07-03 00:00 | user1@example.com | admin1@example.com | Вечеринка в честь дня рождения |
| 2023-12-23 00:00 | user2@example.com | user1@example.com | Телеконференция |

**Рис. 1.4.** Краткое описание приложения-календаря

Приложение спроектировано максимально просто, чтобы мы могли сосредоточиться на важных аспектах безопасности, не отвлекаясь на детали *объектно-реляционного маппинга (ORM)* и сложные приемы создания пользовательского интерфейса. Мы предполагаем, что вы будете обращаться к дополнительным материалам в разделе «Дополнительные справочные материалы» этой книги, чтобы ознакомиться с базовой функциональностью, реализованной в примерах кода.

Код написан на Spring и Spring Security 6, но многие примеры будет относительно легко адаптировать к другим версиям Spring Security. Различия между Spring Security 4 и 6 подробно рассматриваются в главе 16 «Миграция на Spring Security 6», там же можно получить помощь в переводе примеров на синтаксис Spring Security 6.

> **Примечание**
> Пожалуйста, не берите это приложение за основу для создания настоящего онлайн-календаря. Оно специально сконструировано максимально просто, чтобы сфокусироваться на понятиях и конфигурациях, которые мы изучаем в этой книге.

В следующем разделе мы рассмотрим архитектуру приложения.

## Архитектура приложения JBCP Calendar

Веб-приложение построено по стандартной трехуровневой схеме, состоящей из веб-слоя, слоя сервисов и слоя доступа к данным, как показано на рис. 1.5.

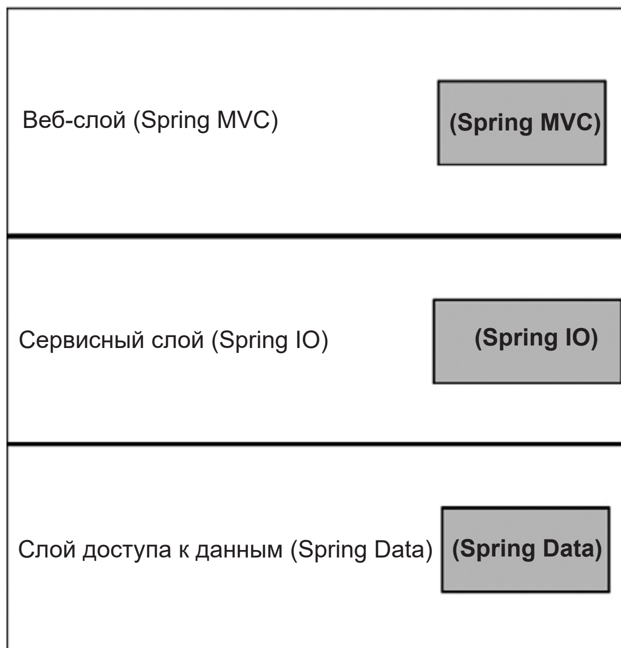


**Рис. 1.5.** Архитектура приложения JBCP Calendar

Дополнительный материал об архитектурах MVC вы можете найти в разделе «Полезные ссылки».

Веб-слой инкапсулирует код и функциональность MVC. В этом примере мы будем применять фреймворк Spring MVC, но с таким же успехом можно использовать *Spring Web Flow (SWF)*, *Apache Struts* или даже веб-стек, совместимый со Spring, например *Apache Wicket*.

В типичном веб-приложении, использующем Spring Security, большая часть настройки и расширения кода происходит в веб-слое. Например, класс `EventsController` преобразует HTTP-запрос в процесс сохранения события в базе данных. Если у вас мало опыта работы с веб-приложениями и Spring MVC в частности, было бы разумно внимательно изучить базовый код и убедиться, что вы понимаете его, прежде чем мы перейдем к более сложным темам. Опять же, мы постарались сделать веб-сайт максимально простым, а приложение-календарь взяли только для того, чтобы дать сайту понятный заголовок и создать легкую структуру.

> **Примечание**
> Подробные инструкции по настройке примера приложения вы найдете в разделе «Дополнительные справочные материалы».

Сервисный слой инкапсулирует бизнес-логику приложения. В нашем приложении мы применяем `DefaultCalendarService` как очень простой интерфейс для слоя доступа к данным, чтобы проиллюстрировать некоторые моменты, связанные с защитой методов сервисов приложений. Сервисный слой используется для работы со Spring Security API и с нашим Calendar API в рамках одиночного вызова метода. Мы рассмотрим это более подробно в главе 3 «Пользовательская аутентификация».

В типичном веб-приложении сервисный слой включает в себя проверку бизнес-правил, композицию и декомпозицию бизнес-объектов, а также сквозную функциональность, такую как аудит.

Слой доступа к данным содержит код, отвечающий за манипуляции с содержимым таблиц базы данных. Во многих приложениях Spring именно здесь можно увидеть ORM, такие как Hibernate или JPA. Слой доступа к данным предоставляет сервисному слою API на основе объектов. В нашем приложении мы используем базовую функциональность JDBC для обеспечения персистентности в базе данных H2 в памяти. Например, `JdbcEventDao` сохраняет объекты событий в базе данных.

В настоящих веб-приложениях для доступа к данным применяются более комплексные решения. Но поскольку ORM и доступ к данным вообще, как правило, сбивают с толку некоторых разработчиков, мы решили максимально упростить эту область для большей ясности.

В следующем разделе мы рассмотрим результаты аудита.

## Рассмотрение результатов аудита

Давайте вернемся к нашей электронной почте и посмотрим, как продвигается аудит. О-о-о... результаты выглядят не очень хорошо:

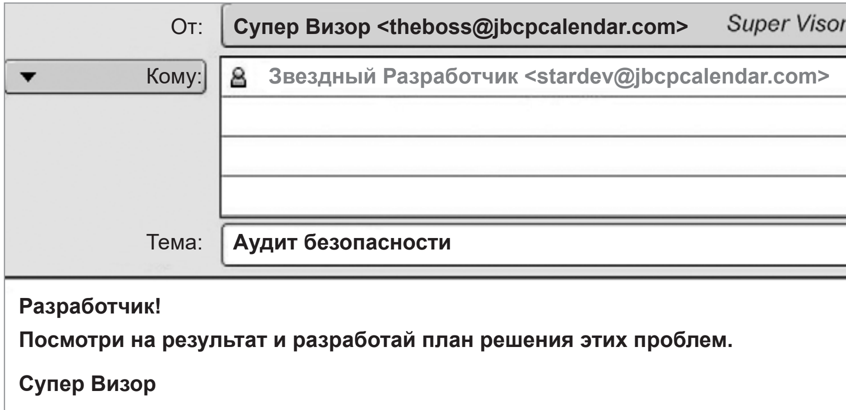


**Рис. 1.6.** Письмо о результатах аудита

*Результаты аудита приложения. Это приложение демонстрирует небезопасное поведение:*

- *непреднамеренное повышение привилегий из-за отсутствия защиты URL-адреса и общей аутентификации;*
- *ненадлежащее использование авторизации или ее отсутствие;*
- *отсутствует защита учетных данных базы данных;*
- *личная или конфиденциальная информация легко доступна или не зашифрована;*
- *ненадежная защита на транспортном уровне из-за отсутствия SSL-шифрования.*

*Уровень риска высокий. Мы рекомендуем отключить это приложение до тех пор, пока эти проблемы не будут устранены.*

Ой! Такой результат выглядит плачевно для нашей компании. Нам лучше взяться за решение этих проблем как можно быстрее.

Компании (их партнеры или заказчики) часто нанимают сторонних специалистов по безопасности для проверки эффективности защиты своего программного обеспечения путем сочетания белого взлома, анализа исходного кода, а также официальных или неформальных бесед с разработчиками и архитекторами приложений.

*Белый*, или *этичный, хакинг* — это не злонамеренный взлом, а действия профессионалов, которых компании специально нанимают, чтобы научиться лучше защищаться от угроз.

Как правило, цель *аудита безопасности* — предоставить руководству или клиентам уверенность в том, что были соблюдены основные методы безопасной разработки для обеспечения целостности и сохранности данных и функций системы заказчика. В зависимости от отрасли, для которой предназначено программное обеспечение, аудитор может также протестировать его, используя отраслевые стандарты или метрики соответствия.

**Примечание**

Два конкретных стандарта безопасности, с которыми вы наверняка столкнетесь на определенном этапе своей карьеры, — это *стандарт безопасности данных индустрии платежных карт (PCI DSS)* и *Закон о конфиденциальности и подотчетности в сфере медицинского страхования (HIPAA)*[1]. Оба этих стандарта призваны обеспечить безопасность специфической конфиденциальной информации (такой, как данные кредитных карт и медицинская информация) путем сочетания технологических процессов и программного обеспечения.

Во многих других отраслях и странах действуют аналогичные правила в отношении *конфиденциальной или позволяющей установить личность информации (PII)*. Несоблюдение этих стандартов не только неэтично, но и может повлечь за собой серьезную ответственность для вас или вашей компании в случае нарушения безопасности (не говоря уже о плохой репутации).

Получение результатов *аудита безопасности* может открыть вам новые горизонты. Внесение необходимых улучшений — прекрасная возможность для самообразования и совершенствования программного обеспечения, а также внедрения методов и политики, которые сделают ваш продукт безопасным.

В следующем разделе мы рассмотрим выводы аудитора и разработаем план по их устранению.

# Устранение недочетов после аудита безопасности

В этом разделе мы подробно рассмотрим результаты аудита безопасности, чтобы пролить свет на уязвимости и проблемные области в системе безопасности нашего приложения. Мы проанализируем полученные данные и перейдем к изучению различных эффективных стратегий и подходов для защиты от угроз и снижения выявленных рисков. Эта глава — дорожная карта для повышения надежности приложения и обеспечения его устойчивости к потенциальным уязвимостям.

---

[1] В России похожий нормативный правовой акт — Федеральный закон «О персональных данных» 152-ФЗ. — *Примеч. науч. ред.*

## Аутентификация

Аутентификация — это одна из двух ключевых концепций безопасности, которые вы должны усвоить при разработке защищенных приложений (вторая — авторизация). Аутентификация определяет, кто пытается запросить ресурс. Вы могли встречаться с аутентификацией в повседневной онлайн- и офлайн-жизни, причем в совершенно разных контекстах:

- *Аутентификация на основе учетных данных*: когда вы входите в свою учетную запись электронной почты в интернете, вы, скорее всего, указываете свое имя пользователя и пароль. Поставщик услуг электронной почты сопоставляет ваше имя пользователя с уже имеющимся именем пользователя в своей базе данных и проверяет, совпадает ли ваш пароль с тем, который у него записан. Эти учетные данные используются системой электронной почты для подтверждения того, что вы — пользователь системы. Сначала мы будем применять этот тип аутентификации для защиты конфиденциальных областей приложения JBCP Calendar. С технической точки зрения почтовая система может проверять учетные данные не только в базе данных, но и где угодно, например на корпоративном сервере каталогов, таком как *Microsoft Active Directory*. В этой книге рассмотрено несколько примеров такого рода интеграции.
- *Двухфакторная аутентификация*: когда вы снимаете деньги в банкомате своего банка, вы прикладываете карту и вводите свой личный идентификационный номер, прежде чем вам разрешат получить наличные или провести другие операции. Этот тип аутентификации похож на аутентификацию по имени пользователя и паролю, за исключением того, что имя пользователя закодировано на магнитной полосе карты. Сочетание физической карты и введенного пользователем PIN-кода позволяет банку убедиться, что вы имеете доступ к счету. Комбинация пароля и физического устройства (вашей пластиковой карты для банкомата) — широко распространенная форма двухфакторной аутентификации. В профессиональной среде, где безопасность очень важна, такие устройства часто используются для доступа к высокозащищенным системам, особенно при работе с финансовой информацией или личными данными. Аппаратное устройство, такое как *RSA SecurID*, сочетает в себе собственно устройство, работающее по времени, и серверное программное обеспечение, что делает подобную среду чрезвычайно сложной для взлома.
- *Аппаратная аутентификация*: когда вы утром заводите свой автомобиль, вы вставляете металлический ключ в замок зажигания и поворачиваете его, чтобы машина завелась. Хотя этот пример не похож на два других, правильное совпадение выступов на ключе и тумблеров в замке зажигания — это форма аппаратной аутентификации.

Существуют десятки форм аутентификации, которые могут решить проблемы безопасности программного и аппаратного обеспечения, — у каждой из них есть

свои плюсы и минусы. В первой половине этой книги мы рассмотрим некоторые из этих методов в том виде, в каком они применимы к Spring Security. В нашем приложении нет никакой аутентификации, поэтому аудит включал риск непреднамеренного повышения привилегий.

Обычно программная система делится на две высокоуровневые области — неаутентифицированную (анонимную, не прошедшую проверку подлинности) и аутентифицированную, как показано на снимке экрана (рис. 1.7).

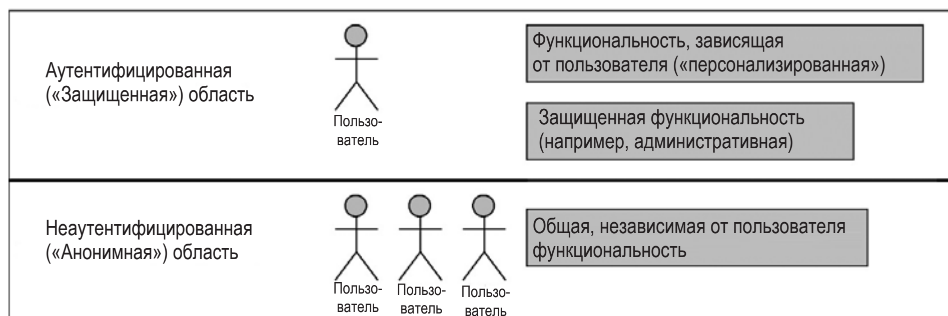


**Рис. 1.7.** Высокоуровневые области в программной системе

Функциональность приложения в анонимной области не зависит от личности пользователя (например, страница приветствия для онлайн-приложения).

В анонимных областях:

- пользователю не нужно входить в систему или иным образом идентифицировать себя;
- не отображается конфиденциальная информация, например имена, адреса, кредитные карты и заказы;
- не предоставляется функциональность для управления общим состоянием системы или ее данными.

Неаутентифицированные области системы предназначены для всех пользователей, даже тех, кого мы еще не идентифицировали. Однако, возможно, для идентификации пользователей в этих областях появится дополнительная функциональность (например, вездесущий текст `Здравствуйте {Имя}`). Выборочное отображение контента для аутентифицированных пользователей полностью поддерживается с помощью библиотеки тегов Spring Security и рассматривается в главе 11 «Детализированный контроль доступа».

Мы решим эту проблему и реализуем аутентификацию на основе форм, используя возможности автоматической настройки Spring Security, в главе 2 «Начало работы со Spring Security». После этого рассмотрим другие способы выполнения аутентификации (которые обычно связаны с интеграцией системы с корпоративными или другими внешними хранилищами аутентификации).

Следующий раздел посвящен *авторизации*.

## Авторизация

Авторизация — это вторая из двух основных концепций безопасности, важных для реализации и понимания безопасности приложений. Авторизация просматривает информацию, которая была подтверждена при аутентификации, чтобы определить, должен ли быть предоставлен доступ к определенному ресурсу. Построенная вокруг модели авторизации для данного приложения, система авторизации разделяет функциональность и данные приложения таким образом, чтобы их доступность можно было контролировать, сопоставляя комбинацию привилегий, функциональных возможностей и данных с конкретными пользователями. Сбой в работе нашего приложения на этом этапе аудита показывает, что роль пользователя не ограничивает функциональность приложения. Представьте, что в вашем интернет-магазине возможность просматривать, отменять или изменять заказы и информацию о клиентах доступна любому пользователю сайта!

Авторизация обычно включает в себя два отдельных аспекта, которые в совокупности описывают доступность защищенной системы:

- Первый — это маппинг (сопоставление) аутентифицированного субъекта с одним или несколькими правами (часто называемыми *ролями*). Например, обычный пользователь вашего веб-сайта может рассматриваться как обладатель прав посетителя, в то время как администратор сайта может быть наделен административными правами.
- Второй — это назначение проверок полномочий для защищенных ресурсов системы. Обычно это делается при разработке системы через явное объявление в коде или через параметры конфигурации. Например, экран, позволяющий просматривать события других пользователей, должен быть доступен только тем, у кого есть административные права.

> **Примечание**
> Защищенным ресурсом может быть любой аспект системы, который должен быть доступен при определенных условиях на основании прав пользователя.

Защищенными ресурсами веб-приложения могут быть отдельные веб-страницы, целые разделы веб-сайта или части отдельных страниц. В свою очередь, защищенными бизнес-ресурсами могут быть вызовы методов классов или отдельные бизнес-объекты.

Можно представить себе проверку прав, которая анализирует субъекта (личность текущего пользователя или системы, взаимодействующей с приложением), просматривает его учетную запись и определяет, действительно ли он администратор. Если такая проверка установит, что субъект, пытающийся получить

доступ к защищенной области, действительно администратор, запрос будет выполнен успешно. Однако если субъект не обладает достаточными правами, запрос должен быть отклонен.

Давайте подробнее рассмотрим пример конкретного защищенного ресурса — страницы All events (Все события). Страница All events требует административного доступа (мы ведь не хотим, чтобы обычные пользователи просматривали события других пользователей) и поэтому ожидает определенного уровня прав пользователя, получающего к ней доступ.

Чтобы понять, как может приниматься решение, когда администратор сайта пытается получить доступ к защищенному ресурсу, представьте, что сравнение фактических полномочий с требуемыми выражается в терминах теории множеств. Мы можем изобразить это решение как *диаграмму Венна* для пользователя с правами администратора (рис. 1.8).

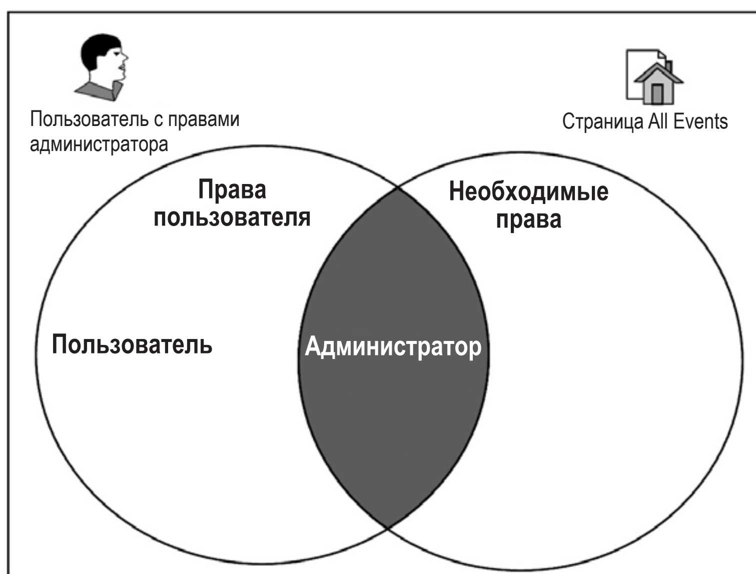


**Рис. 1.8.** Диаграмма Венна для пользователя с правами администратора

*Права пользователя* (пользователей и администраторов) пересекаются с *необходимыми правами* (администраторов) для страницы, поэтому пользователю предоставляется доступ.

Для неавторизованного пользователя это выглядит так, как на рис. 1.9.

У нас симметричная разность. Наборы прав не пересекаются и не имеют общих элементов. Поэтому пользователю отказано в доступе к странице. Таким образом, мы продемонстрировали основной принцип авторизации доступа к ресурсам.

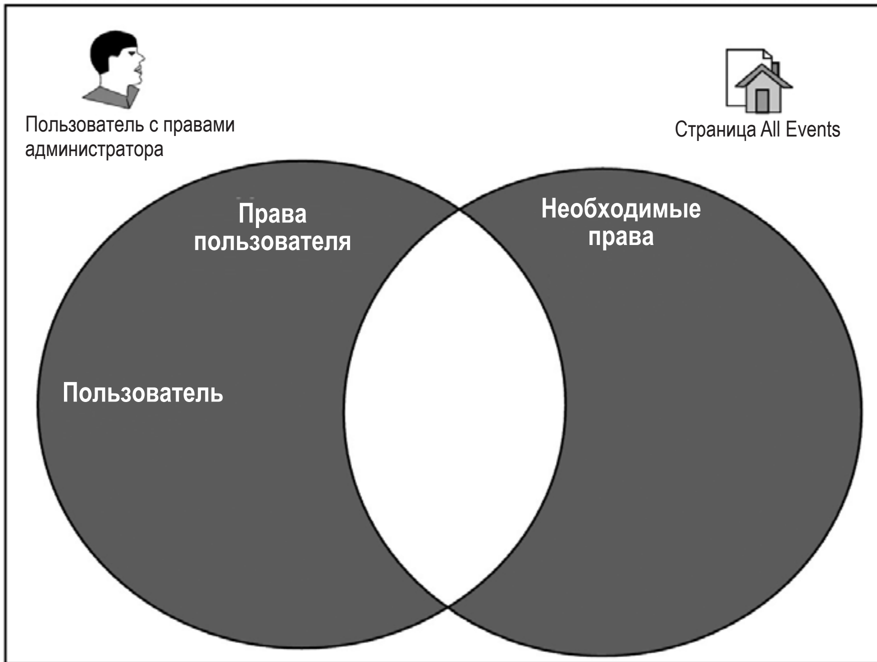


**Рис. 1.9.** Диаграмма Венна для гостя (неавторизованного пользователя)

В действующем приложении это решение принимает реальный код — пользователь получает разрешение или отказ на доступ к запрашиваемому защищенному ресурсу. Мы рассмотрим основную проблему авторизации с помощью инфраструктуры авторизации Spring Security в главе 2 «Начало работы со Spring Security», а затем более продвинутую авторизацию в главе 12 «Списки управления доступом (ACL)» и главе 13 «Пользовательская авторизация».

Теперь, когда мы рассмотрели концепцию авторизации, изучим, как применить ее к безопасности баз данных.

## Безопасность учетных данных базы данных

В терминологии Spring учетные данные базы данных обычно обозначают информацию, необходимую для установления соединения между приложением Spring и базой данных. Эти учетные данные включают:

- имя пользователя: имя пользователя или идентификатор пользователя, связанный с учетной записью базы данных, который приложение Spring использует для подключения;
- пароль: соответствующий пароль для указанного имени пользователя, обеспечивающий аутентификацию для доступа к базе данных;

- URL-адрес базы данных: URL-адрес, указывающий местоположение и подробные сведения о базе данных. Он включает информацию о хосте, порте и имени базы данных.

Изучив исходный код приложения и файлы конфигурации, аудиторы обнаружили, что пароли пользователей хранятся в файлах конфигурации открытым текстом, что значительно облегчает злоумышленнику, имеющему доступ к серверу, получение контроля над приложением.

Поскольку приложение содержит личные и финансовые данные, несанкционированный доступ пользователя к любым данным может привести к краже или подделке личных данных компании. Защита доступа к учетным данным должна быть нашим главным приоритетом, и в первую очередь мы должны сделать так, чтобы одна точка отказа в системе безопасности не поставила под угрозу всю систему.

Настройку уровней доступа к базе данных в Spring Security для хранения учетных данных, для которого требуется *подключение к JDBC*, мы рассмотрим в главе 4 «Аутентификация на основе JDBC». В этой же главе мы изучим встроенные методы для повышения безопасности паролей, хранящихся в базе данных.

После изучения вопроса о безопасности учетных данных базы данных мы перейдем к анализу выявленных проблем с конфиденциальной информацией.

## Конфиденциальная информация

Персональные данные или конфиденциальная информация легко доступны или не зашифрованы. Аудиторы отметили, что некоторые важные и конфиденциальные фрагменты данных были полностью не зашифрованы и нигде не маскируются в системе. К счастью, существуют простые шаблоны проектирования и инструменты, которые позволяют нам надежно защитить эту информацию благодаря поддержке *аспектно-ориентированного программирования (AOP)* на основе аннотаций в Spring Security.

## Защита на транспортном уровне

Защита на транспортном уровне ненадежна из-за отсутствия SSL-шифрования.

В реальном мире немыслимо, чтобы онлайн-приложение, содержащее конфиденциальную информацию, работало без SSL-защиты, но, к сожалению, JBCP Calendar находится именно в такой ситуации. SSL-защита гарантирует, что связь между клиентом браузера и сервером веб-приложения будет надежно защищена от многих видов несанкционированного доступа и слежки.

В разделе «Дополнительные справочные материалы» мы рассмотрим основные варианты использования защиты на транспортном уровне как части определения защищенной структуры приложения.