Лёзин Илья Александрович

Лёзина Ирина Викторовна

Учебное пособие «Разработка веб-приложений с использованием Spring Boot»

УДК 004.423
ББК 32.973

Рецензенты: Гордеева Ольга Александровна, доцент кафедры программных систем, к.т.н., доцент,

Пальмов Сергей Вадимович, доцент кафедры информационных систем и технологий ФГБОУ ВО ПГУТИ, к.т.н., доцент.

В настоящем учебном пособии изложены теоретические основы и практические примеры разработки распределённых приложений с применением Spring Boot, которые могут быть применены при изучении дисциплины «Разработка web-приложений», являющейся завершающим этапом изучения технологий разработки объектно-ориентированных распределенных приложений на языке программирования Java. Учебное пособие предназначено для обучающихся по направлению подготовки бакалавриата 09.03.01 «Информатика и вычислительная техника».

Оглавление

Многоуровневая архитектура

Архитектура программного обеспечения – это структура, на базе которой создается приложение, взаимодействуют модули и компоненты всей программы. Созданием архитектур программисты занимаются уже долгое время, поэтому сейчас нам известно немало архитектурных шаблонов. Разбираться в них нужно: когда пишешь веб-приложение, проблема архитектуры становится острой, ведь в ней компонентов и модулей больше, чем в обычном приложении.

Архитектурный шаблон – это уже придуманный способ решения какой-то задачи по проектированию программного обеспечения. Если паттерны проектирования (фабричный метод, абстрактная фабрика, строитель, прототип и прочие) используются при простом написании кода, создании классов и планировании их взаимодействия, то архитектурные шаблоны задействуют на более высоком уровне абстракции – при планировании взаимодействия пользователя приложения с сервером, данными и другими компонентами проекта.

Рассмотрим клиент-серверную архитектуру. Из названия складывается впечатление, что здесь все просто и понятно. На примере чата проанализируем возможные варианты. Самый простой вариант – пользователи отправляют сообщения друг другу напрямую через интернет по IP-адресам, которые им известны (рисунок 1).

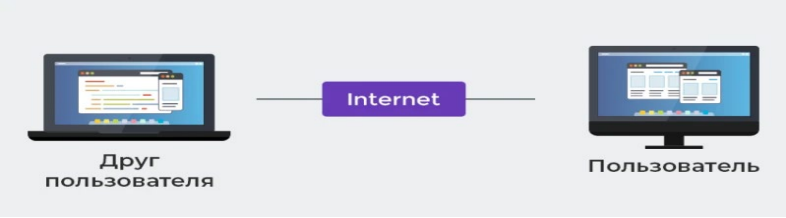

Рисунок 1 – Чат двух пользователей

Поначалу может показаться, что все отлично работает, пока не появляется еще один друг с вопросом: «А почему вы не добавите меня в свой чат?»

При необходимости добавить общего друга в чат возникает архитектурная проблема: каждому пользователю чата нужно обновить информацию о количестве пользователей, добавить IP-адрес нового юзера. А еще при отправке сообщения оно должно доставляться всем участникам. Это самые очевидные проблемы из тех, которые возникнут. Еще куча проблем будет спрятана в самом коде.

Чтобы избежать их, нужно использовать сервер, который будет хранить всю информацию о пользователях, знать их адреса (рисунок 2). Сообщение нужно будет отправить только на сервер. А он, в свою очередь, разошлет сообщение всем адресатам. При добавлении серверной части в чат начинается построение клиент-серверной архитектуры (рисунок 3).

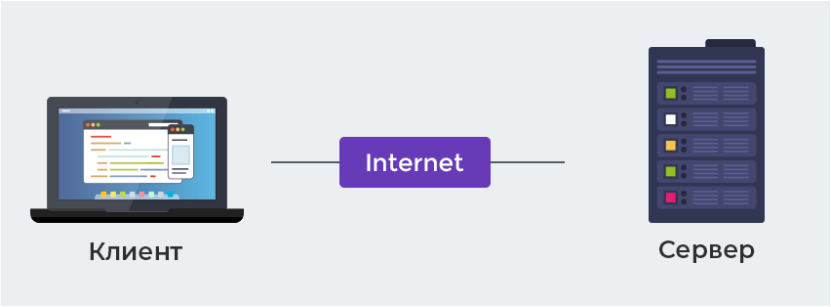

Рисунок 2 – Чат нескольких пользователей

Итак, клиент-серверная архитектура – это шаблон проектирования, основа для создания веб-приложений. Данная архитектура состоит из трех компонентов.

Клиент – из названия понятно, что это пользователь сервиса (веб-приложения), который обращается к серверу для получения какой-то информации.

Сервер – место, где располагается веб-приложение или его серверная часть. Он владеет необходимой информацией о пользователях или может ее запрашивать. Также при обращении клиента сервер возвращает ему запрашиваемую информацию.

Сеть обеспечивает обмен информацией между клиентом и сервером.

Сервер может обрабатывать огромное количество запросов от разных пользователей. То есть клиентов может быть много, а если им нужно обменяться информацией между собой, делать это придется через сервер. Таким образом, сервер получает еще одну дополнительную функцию – контроль трафика.

В случае многопользовательского чата, весь программный код будет состоять из двух модулей:

- клиентского – содержит графический интерфейс для авторизации, отправки/получения сообщений;
- серверного – веб-приложение, которое размещается на сервере и принимает сообщения от пользователей, обрабатывает их, а потом отправляет адресатам.

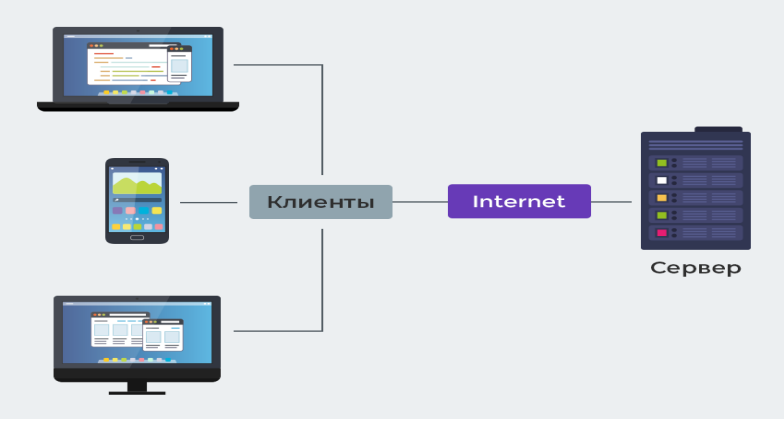

Рисунок 3 – Клиент-серверная архитектура

Для просмотра информации в интернете пользователь открывает браузер, в строке поиска вводит запрос, а в ответ получает информацию от поисковика. В этой цепочке браузер – это клиент. Он отправляет запрос с информацией о том, что ищет пользователь, серверу. Сервер обрабатывает запрос, находит наиболее релевантные результаты, упаковывает их в понятный для браузера (клиента) формат и отправляет назад. В таких сложных сервисах как поисковики серверов может быть много. Например, сервер авторизации, сервер для поиска информации, сервер для формирования ответа. Но клиент об этом ничего не знает: для него сервер является чем-то единым. Клиент знает только о точке входа, то есть, адресе сервера, которому нужно отправить запрос.

Простая клиент-серверная архитектура применяется очень редко и только для очень простых приложений. Для действительно больших и сложных проектов используются разные типы архитектур. Рассмотрим модель трехуровневой архитектуры (рисунок 4), очень похожей на клиент-серверную.

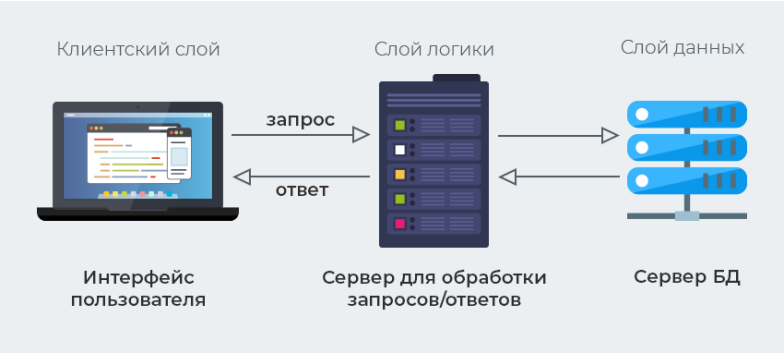

Рисунок 4 – Трёхуровневая архитектура

Это архитектурный шаблон, в котором появляется третий участник – хранилище данных. При использовании этого шаблона, три уровня принято называть слоями.

Клиентский слой – интерфейс пользователя. Это может быть веб-браузер, которому отправляются HTML-страницы, или графическое приложение, написанное с помощью JavaFX. Главное, чтобы с его помощью пользователь мог отправлять запросы на сервер и обрабатывать его ответы.

Слой логики – сервер, на котором происходит обработка запросов/ответов. Часто его еще называют серверным слоем. Также здесь происходят все логические операции: математические расчеты, операции с данными, обращения к другим сервисам или хранилищам данных.

Слой данных – сервер баз данных: к нему обращается сервер. В этом слое сохраняется вся необходимая информация, которой пользуется приложение при работе.

Таким образом, сервер принимает на себя все обязательства по обращению к данным, не давая возможности пользователю обратиться к ним напрямую.

Использование подобной архитектуры даёт немало плюсов, среди которых:

- Возможность построить защиту от SQL-инъекций – это атака на сервер, при которой передается SQL-код, и при выполнении этого кода злоумышленник может воздействовать на базу данных.
- Разграничение данных, к которым необходимо регулировать пользовательский доступ.
- Возможность модифицировать данные перед отправкой клиенту.
- Масштабируемость – возможность расширить наше приложение на несколько серверов, которые будут использовать одну и ту же базу данных.

- Меньшие требования к качеству соединения пользователя. Формируя ответ на сервере, из базы данных извлекается много различной информации, она обрабатывается, остаётся только то, что нужно пользователю. Таким образом сокращается объем информации, который будет отправлен в качестве ответа клиенту.

Протоколы HTTP/HTTPS

Перед тем, как начать разбираться в протоколах HTTP и HTTPS, нужно прояснить один момент: речь идет о протоколах передачи данных по сети на прикладном уровне модели OSI. Протоколом передачи данных называют общепринятое соглашение, благодаря которому разработчики разных сервисов отправляют информацию в едином виде.

Например, используя Google Chrome, можно получить информацию с любого сайта, потому что разработчики передают ее с помощью стандартного протокола HTTP, а браузер умеет его обрабатывать.

Единые правила очень удобны и самим разработчикам серверных частей: существует очень много библиотек, которые могут преобразовать информацию и отправить по необходимому протоколу.

Изначально HTTP задумывался как протокол передачи HTML-страниц. Долгое время так и было, но сейчас программисты передают по нему и текстовые, и мультимедийные данные. В общем, этот протокол универсальный и гибкий, и использовать его действительно просто.

Сразу стоит отметить, что HTTP-протокол состоит только из текста, нас же больше всего интересует структура, в которой расположен этот текст.

Каждое сообщение состоит из трех частей:

- Стартовая строка (Starting line) – определяет служебные данные.
- Заголовки (Headers) – описание параметров сообщения.
- Тело сообщения (Body) – данные сообщения. Должны отделяться от заголовков пустой строкой.

По HTTP-протоколу можно отправить запрос на сервер (request) и получить ответ от сервера (response). Запросы и ответы немного отличаются параметрами.

Так выглядит простой HTTP-запрос.

```
1 GET / HTTP/1.1
2 Host: servername.ru
3 User-Agent: firefox/5.0 (Linux; Debian 5.0.8; en-US; rv:1.8.1.7)
```

В стартовой строке указаны:

- GET – метод запроса;
- / – путь запроса (path);
- HTTP/1.1 – версия протокола передачи данных.

Затем следуют заголовки:

- Host – хост, которому адресован запрос;
- User-Agent – клиент, который отправляет запрос.

Тело сообщения отсутствует.

В HTTP-запросе обязательны только стартовая строка и заголовок Host. HTTP-запрос должен содержать какой-то метод. Всего их девять: GET, POST, PUT, OPTIONS, HEAD, PATCH, DELETE, TRACE, CONNECT. Самые распространенные – GET и POST.

GET – запрашивает контент у сервера. Поэтому у запросов с методом GET нет тела сообщения. Но при необходимости можно отправить параметры через path в таком формате:

https://servername.ru/page/123?name1=value1&name2=value2

Здесь:

- servername.ru– хост,
- /page/123 – путь запроса,
- ? – разделитель, обозначающий, что дальше следуют параметры запроса.

В конце перечисляются параметры в формате ключ=значение, разделенные амперсандом.

POST – публикует информацию на сервере. POST-запрос может передавать разную информацию: параметры в формате ключ=значение, JSON, HTML-код или даже файлы. Вся информация передается в теле сообщения.

```
1  POST /user/create/json HTTP/1.1
2  Accept: application/json
3  Content-Type: application/json
4  Content-Length: 28
5  Host: servername.ru
6
7  {
8    "Id": 12345,
9    "User": "John"
10 }
```

Запрос отправляется по адресу servername.ru/user/create/json, версия протокола – HTTP/1.1. Accept указывает, какой формат ответа клиент ожидает получить, Content-Type – в каком формате отправляется тело сообщения. Content-Length – количество символов в теле. HTTP-запрос может содержать много разных

заголовков. Подробнее с ними можно ознакомиться в спецификации протокола.

После получения запроса, сервер его обрабатывает и отправляет ответ клиенту.

```
1  HTTP/1.1 200 OK
2  Content-Type: text/html; charset=UTF-8
3  Content-Length: 98
4
5  <html>
6    <head>
7      <title>An Example Page</title>
8    </head>
9    <body>
10     <p>Hello World</p>
11   </body>
12 </html>
```

Стартовая строка в ответе содержит версию протокола (HTTP/1.1), Код статуса (200), Описание статуса (OK). В заголовках – тип и длина контента. В теле ответа – HTML-код, который браузер отрендерит в HTML-страницу.

Response Status Code всегда трехзначный, и первая цифра кода указывает категорию ответа:

- 1xx – информационный. Запрос получен, сервер готов к продолжению;
- 2xx – успешный. Запрос получен, понятен и обработан;
- 3xx – перенаправление. Следующие действия нужно выполнить для обработки запроса;
- 4xx – ошибка клиента. Запрос содержит ошибки или не отвечает протоколу;

- 5xx – ошибка сервера. Сервер не смог обработать запрос, хотя был составлен верно.
- Вторая и третья цифры в коде детализируют ответ:
- 200 OK – запрос получен и успешно обработан;
- 201 Created – запрос получен и успешно обработан, в результате чего создан новый ресурс или его экземпляр;
- 301 Moved Permanently – запрашиваемый ресурс был перемещен навсегда, и последующие запросы к нему должны происходить по новому адресу;
- 307 Temporary Redirect – ресурс перемещен временно. Пока к нему можно обращаться, используя автоматическую переадресацию;
- 403 Forbidden – запрос понятен, но нужна авторизация;
- 404 Not Found – сервер не нашел ресурс по этому адресу;
- 501 Not Implemented – сервер не поддерживает функциональность для ответа на этот запрос;
- 505 HTTP Version Not Supported – сервер не поддерживает указанную версию HTTP-протокола.

Вдобавок к статус-коду ответа также отправляется описание статуса, благодаря которому интуитивно понятно, что значит конкретный статус.

HTTP-протокол очень практичен: в нем предусмотрено большое количество заголовков, используя которые можно настроить гибкое общение между клиентом и сервером. Все заголовки запросов и ответов, методы запросов и статус-коды ответов можно узнать в официальной спецификации протокола, которая описывает все нюансы.

С развитием технологий и активным перемещением персональных данных в интернет пришлось задуматься о том, как

обеспечить дополнительную защиту информации, которую клиент передает серверу. В результате появился протокол HTTPS.

HTTPS синтаксически идентичен протоколу HTTP, то есть использует те же стартовые строки и заголовки. Единственные отличия – дополнительное шифрование и порт по умолчанию (443).

HTTPS шифруется между HTTP и TCP, то есть между прикладным и транспортным уровнями.

Современный стандарт шифрования – по протоколу TLS. Шифрование происходит перед тем, как информация попадает на транспортный уровень. В HTTPS шифруется абсолютно вся информация, кроме хоста и порта, куда отправлен запрос.

Для перевода сервера на использование HTTPS протокола вместо HTTP не нужно менять код сервера. Включение этой функциональности происходит в контейнерах сервлетов.

Чтобы увидеть HTTP-запросы, можно открыть Google Chrome, нажать F12 и выбрать вкладку Network. Тут будут отображаться все запросы и ответы, отправленные и полученные браузером.

Spring Framework

Spring Framework – универсальный фреймворк с открытым исходным кодом для Java-платформы.

Spring Framework обеспечивает комплексную модель разработки и конфигурации для современных бизнес-приложений на Java – на любых платформах. Ключевой элемент Spring – поддержка инфраструктуры на уровне приложения: основное внимание уделяется технологиям бизнес-приложений, поэтому

разработчики могут сосредоточиться на бизнес-логике без лишних настроек в зависимости от среды исполнения.

Spring Framework может быть рассмотрен как коллекция меньших фреймворков или фреймворков во фреймворке. Большинство этих фреймворков может работать независимо друг от друга, однако они обеспечивают большую функциональность при совместном их использовании. Эти фреймворки делятся на структурные элементы типовых комплексных приложений.

Spring Core – центральная часть всего фреймворка. Контейнер Inversion of Control (IoC) отвечает за конфигурирование компонентов приложений и управление жизненным циклом Java-объектов. Сюда же относятся логика управления ресурсами и интернационализация (i18n), валидация объектов, связывание данных, преобразование типов, ведение логов и прочее.

Фреймворк аспектно-ориентированного программирования (Spring AOP) работает с функциональностью, которая не может быть реализована возможностями объектно-ориентированного программирования на Java без потерь.

Фреймворк доступа к данным (Spring Data) работает с системами управления реляционными и нереляционными базами данных на Java-платформе, используя JDBC- и ORM-средства и обеспечивая решения задач, которые повторяются в большом числе окружений, построенных на Java.

Фреймворк управления транзакциями отвечает за координацию различных API управления транзакциями и инструментарий настраиваемого управления транзакциями для объектов Java.

Фреймворк Spring MVC – это каркас, основанный на HTTP и сервлетах, предоставляющий множество возможностей для расширения и настройки (customization).

Фреймворк удалённого доступа (Spring Remoting) – это конфигурируемая передача Java-объектов через сеть в стиле RPC, поддерживающая RMI, CORBA, HTTP-based протоколы, включая web-сервисы (SOAP).

Фреймворк аутентификации и авторизации (Spring Security) предоставляет конфигурируемый инструментарий процессов аутентификации и авторизации, поддерживающий много популярных и ставших индустриальными стандартами протоколов и инструментов.

Фреймворк работы с сообщениями (Spring Messaging) обеспечивает конфигурируемую регистрацию объектов-слушателей сообщений для прозрачной обработки сообщений из очереди сообщений с помощью JMS, улучшенная отправка сообщений по стандарту JMS API.

Фреймворк тестирования (Spring Testing) – это каркас, поддерживающий классы для написания модульных и интеграционных тестов.

Spring Boot

Из-за громоздкой конфигурации зависимостей настройка Spring для корпоративных приложений превратилась в весьма утомительное и подверженное ошибкам занятие. Особенно это относится к приложениям, которые используют несколько сторонних библиотек.

Каждый раз, создавая очередное корпоративное Java-приложение на основе Spring, вам необходимо повторять одни и те же рутинные шаги по его настройке:

- в зависимости от типа создаваемого приложения (Spring MVC, Spring JDBC, Spring ORM и т.д.) импортировать необходимые Spring-модули;
- импортировать библиотеку web-контейнеров (в случае web-приложений);
- импортировать необходимые сторонние библиотеки (например, Hibernate, Jackson), при этом необходимо найти версии, совместимые с указанной версией Spring;
- конфигурировать компоненты DAO (источники данных, управление транзакциями и т.д.);
- конфигурировать компоненты web-слоя (диспетчер ресурсов, view resolver);
- определить класс, который загрузит все необходимые конфигурации.

Авторы Spring решили предоставить разработчикам некоторые утилиты, которые автоматизируют процедуру настройки и ускоряют процесс создания и развертывания Spring-приложений, под общим названием Spring Boot.

Spring Boot – это полезный проект, целью которого является упрощение создания приложений на основе Spring. Он позволяет наиболее простым способом создать web-приложение, требуя от разработчиков минимум усилий по его настройке и написанию кода.

Spring Boot обладает большим функционалом, но его наиболее значимыми особенностями являются: управление зависимостями, автоматическая конфигурация и встроенные контейнеры сервлетов.

Чтобы ускорить процесс управления зависимостями, Spring Boot неявно упаковывает необходимые сторонние зависимости для каждого типа приложения на основе Spring и предоставляет их

разработчику посредством так называемых starter-пакетов (spring-boot-starter-web, spring-boot-starter-data-jpa и т.д.)

Starter-пакеты представляют собой набор удобных дескрипторов зависимостей, которые можно включить в свое приложение. Это позволит получить универсальное решение для всех, связанных со Spring технологий, избавляя программиста от лишнего поиска примеров кода и загрузки из них требуемых дескрипторов зависимостей (пример таких дескрипторов и стартовых пакетов будет показан ниже).

Например, если вы хотите начать использовать Spring Data JPA для доступа к базе данных, просто включите в свой проект зависимость spring-boot-starter-data-jpa и все будет готово (вам не придется искать совместимые драйверы баз данных и библиотеки Hibernate).

Если вы хотите создать Spring web-приложение, просто добавьте зависимость spring-boot-starter-web, которая подтянет в проект все библиотеки, необходимые для разработки Spring MVC-приложений, таких как spring-webmvc, jackson-json, validation-api и Tomcat.

Другими словами, Spring Boot собирает все общие зависимости и определяет их в одном месте, что позволяет разработчикам просто использовать их, вместо того, чтобы изобретать колесо каждый раз, когда они создают новое приложение.

Следовательно, при использовании Spring Boot, файл pom.xml содержит намного меньше строк, чем при использовании его в Spring-приложениях.

Второй превосходной возможностью Spring Boot является автоматическая конфигурация приложения.

После выбора подходящего starter-пакета, Spring Boot попытается автоматически настроить Spring-приложение на основе добавленных вами jar-зависимостей.

Например, если вы добавите spring-boot-starter-web, Spring Boot автоматически сконфигурирует такие зарегистрированные бины, как DispatcherServlet, ResourceHandlers, MessageSource.

Если вы используете spring-boot-starter-jdbc, Spring Boot автоматически регистрирует бины DataSource, EntityManagerFactory, TransactionManager и считывает информацию для подключения к базе данных из файла application.properties.

Если вы не собираетесь использовать базу данных и не предоставляете никаких подробных сведений о подключении в ручном режиме, Spring Boot автоматически настроит базу в памяти, без какой-либо дополнительной конфигурации с вашей стороны (при наличии H2 или HSQL библиотек).

Автоматическая конфигурация может быть полностью переопределена в любой момент с помощью пользовательских настроек.

Каждое Spring Boot web-приложение включает встроенный web-сервер. Разработчикам теперь не надо беспокоиться о настройке контейнера сервлетов и развертывании приложения на нем. Теперь приложение может запускаться само, как исполняемый jar-файл с использованием встроенного сервера.

Если вам нужно использовать отдельный HTTP-сервер, для этого достаточно исключить зависимости по умолчанию. Spring Boot предоставляет отдельные starter-пакеты для разных HTTP-серверов.

Создание автономных web-приложений со встроенными серверами не только удобно для разработки, но и является допустимым решением для приложений корпоративного уровня и

становится все более полезно в мире микросервисов. Возможность быстро упаковать весь сервис (например, аутентификацию пользователя) в автономном и полностью развертываемом артефакте, который также предоставляет API – делает установку и развертывание приложения значительно проще.

Spring Boot также предоставляет возможность воспользоваться онлайн-инициализатором приложений, расположенном по адресу https://start.spring.io/.

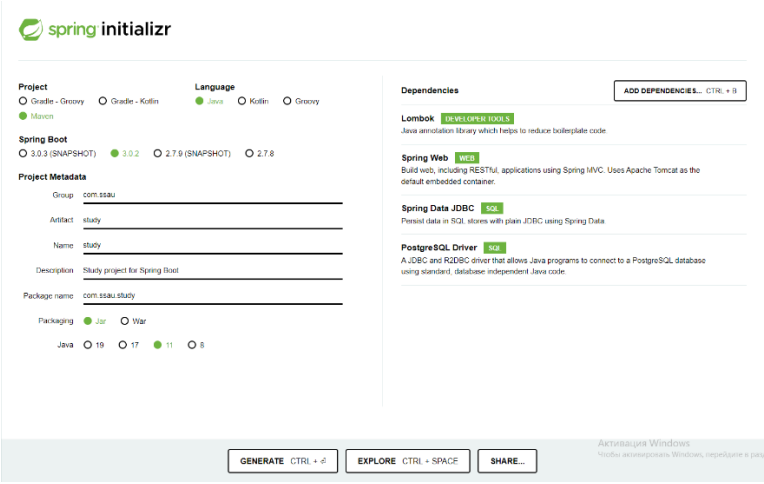

Рисунок 5 – Онлайн-инициализатор приложений

Он позволяет выбрать все основные настройки приложения и генерирует минимально необходимый код для его запуска. Пример на рисунке 5 показывает, что проект настроен на использование языка Java и сборку Maven. Проект будет собираться в виде jar-архива, версия языка 17 (несмотря на выбор 11-й версии, Spring Boot 3+ подразумевает версию Spring Framework 6+, которая работает с Java 17 и выше). В качестве дополнительных зависимостей указаны Spring Web, Spring Data JDBC и драйвер для СУБД PostgreSQL.

Сгенерированный проект имеет вид, представленный на рисунке 6.

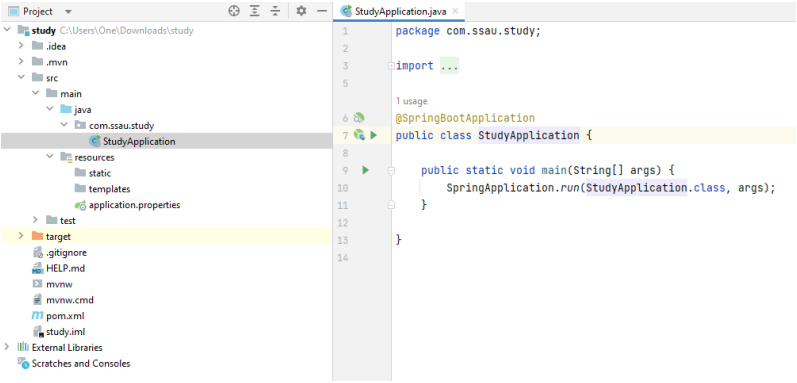

Рисунок 6 – Сгенерированный проект

Зачем нужен Maven

Maven – это инструмент для автоматической сборки проектов на основе описания их структуры в специальных файлах на языке POM (Project Object Model) в формате XML. Чаще всего Maven используют Java-разработчики, однако после установки дополнительных плагинов его также можно применять для работы на PHP, Ruby, Scala, C/C++ и Groovy.

Функциональность системы сборки Maven шире, чем компилятора исходного кода. В процессе работы приложения Apache Maven вызывает компилятор и при этом автоматически управляет зависимостями и ресурсами, например:

- загружает подходящие версии пакетов;
- размещает изображения, аудио- и видеофайлы в нужных папках;
- подгружает сторонние библиотеки.

Автоматическая сборка приложения особенно важна на этапах разработки, отладки и тестирования – Maven помогает собрать код и ресурсы в исполняемое приложение без IDE (среды разработки). При этом система сборки отличается гибкостью:

- может использоваться в IDE – Eclipse, IntelliJ IDEA, NetBeans и других;
- не зависит от операционной системы;
- не требует установки – архив с программой можно распаковать в любой директории;
- все необходимые параметры имеют оптимальные настройки по умолчанию;
- упрощает организацию командной работы и документирование;
- запускает библиотеки для модульного тестирования;
- обеспечивает соблюдение стандартов;
- имеет огромное количество плагинов и расширений.

Также существуют две другие системы сборки Java-приложений – Ant и Gradle, однако Maven пользуется наибольшей популярностью и является стандартом индустрии.

Репозиторий – это место для хранения и обновления файлов проекта. Директория на компьютере разработчика, в которой Maven хранит все jar-файлы отдельного проекта, библиотеки и необходимые модули (зависимости), называется локальным репозиторием. По умолчанию он располагается в папке .m2 текущего пользователя.

Центральный репозиторий – это общее онлайн-хранилище, здесь находятся все библиотеки, плагины и модули, созданные разработчиками сообщества Maven. Если во время сборки проекта система не находит нужную библиотеку (зависимость) в

локальном репозитории разработчика, она автоматически обращается в центральный репозиторий Maven.

Помимо центрального репозитория, можно указывать дополнительные репозитории.

В среде Maven «собранные» проекты называются артефактами, а не приложениями или программами. Термин выбран потому, что готовый проект не всегда является исполняемым приложением – он может быть модулем, плагином или библиотекой.

Для описания структуры проектов Apache Maven применяет разновидность языка XML под названием POM (сокращение от Project Object Model, «объектная модель проекта»).

Основные теги POM:

- project – базовый тег, содержит всю информацию о приложении;
- modelVersion – генерируется автоматически, текущая версия – 4.0.0;
- groupId – пакет, к которому принадлежит приложение, с добавлением имени домена;
- artifactId – уникальный ID артефакта;
- version – создается и обновляется автоматически, во время разработки к номеру версии добавляется суффикс -SNAPSHOT.

В сгенерированном примере файл pom.xml имеет вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.2</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.ssau</groupId>
<artifactId>study</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>study</name>
<description>Study project for Spring Boot</description>
<properties>
    <java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jdbc</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
```

```
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Maven выполняет сборку последовательными фазами.

Проверка – validate. Фреймворк проверяет, корректен ли проект и предоставлена ли вся необходимая для сборки информация.

Компиляция – compile. Maven компилирует исходники проекта.

Тест – test. Проверка скомпилированных файлов. Обычно используется библиотека JUnit.

Сборка проекта – package. По умолчанию осуществляется в формате JAR. Этот параметр можно изменить, добавив в project тег packaging.

Интеграционное тестирование – integration-test. Maven обрабатывает и при необходимости распаковывает пакет в среду, где будут выполняться интеграционные тесты.

Верификация – verify. Артефакт проверяется на соответствие критериям качества.

Инсталляция – install. Артефакт попадает в локальный репозиторий. Теперь его можно использовать в качестве зависимости.

Размещение проекта в удалённом репозитории – deploy. Это финальная стадия работы.

Эти фазы упорядочены и выполняются поочерёдно. Если необходимо собрать проект, система последовательно проведёт оценку, компиляцию и тестирование, и только после этого сборку. Помимо этого есть две фазы, выполняющиеся отдельно, только прямой командой. Это очистка – clean, удаляющая предыдущие сборки, и создание документации для сайта – site.

Секция build не является обязательной, в ней указываются плагины, которые используются при сборке. Плагинов для Maven тысячи, и они могут значительно облегчить разработчикам жизнь, например, плагин для проверки покрытия кода модульными тестами.

В проектах чуть серьёзнее, чем вывод приветствия в консоль, приходится использовать внешние ресурсы. Maven способен автоматически обрабатывать файлы ресурсов и размещать их в

сборке проекта. Для этого их нужно разместить в папке src/main/resources. Файлы будут упакованы с сохранением внутренней структуры каталогов.

Для успешного запуска сгенерированного приложения не хватает ресурсов с настройками подключения к СУБД PostgreSQL.

Настройки можно указывать в формате файла свойств (в файле application.properties):

spring.datasource.url=jdbc:postgresql://host:port/db
spring.datasource.username=postgres
*spring.datasource.password=********

Или в формате YAML (в файле application.yml):

```
spring:
  datasource:
    url: jdbc:postgresql:// host:port/db
    username: postgres
    password: ********
```

После настройки соединения проект готов к запуску. В консоль при запуске будет выведено примерно следующее.

com.ssau.study.StudyApplication : Starting StudyApplication using Java 17.0.6 with PID 10460 (C:\Users\One\Downloads\study\target\classes started by One in C:\Users\One\Downloads\study)
com.ssau.study.StudyApplication : No active profile set, falling back to 1 default profile: "default"
.s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JDBC repositories in DEFAULT mode.
.s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 9 ms. Found 0 JDBC repository interfaces.
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)

o.apache.catalina.core.StandardService : Starting service [Tomcat]
o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.5]
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 767 ms
com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection org.postgresql.jdbc.PgConnection@642a16aa
com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
com.ssau.study.StudyApplication : Started StudyApplication in 1.575 seconds (process running for 2.073)

Приложение уже запускается, но пока что ничего не делает. Чтобы сделать его полноценным сервером с поддержкой REST API, необходимо добавить слои взаимодействия с базой данных и контроллер, который будет реагировать на HTTP-запросы от клиента.

Контроллер.

```
package com.ssau.study.controller;
import ...;
@RestController
@RequestMapping("/api/students")
public class StudentController {
   @Autowired
   private StudentRepository studentRepository;
   @GetMapping("/count")
```

```
    public int count() {
        return studentRepository.count();
    }
    @GetMapping
    public List<Student> findAll() {
        return studentRepository.findAll();
    }
    @GetMapping("/{name}")
    public List<Student> findAllByName(@PathVariable String name) {
        return studentRepository.findAllByName(name);
    }
}
```

Сущностный класс.

```
package com.ssau.study.entity;

import lombok.Getter;
import lombok.Setter;

import java.util.Date;
@Getter
@Setter
public class Student {
    private long id;
    private String name;
    private Date birthdate;
    private int number;
}
```

Репозиторий.

```
package com.ssau.study.repository;
```

```
import com.ssau.study.entity.Student;
import java.util.List;
public interface StudentRepository {
    int count();
    List<Student> findAll();
    List<Student> findAllByName(String name);
}
```

Реализация репозитория.

```
package com.ssau.study.repository;
import ...;
@Repository
@RequiredArgsConstructor
public class JdbcStudentRepository implements StudentRepository {
    private final JdbcTemplate jdbcTemplate;
    private final NamedParameterJdbcTemplate
namedParameterJdbcTemplate;
    private RowMapper<Student> studentMapper = (rs, rowNum) -> {
        Student student = new Student();
        student.setId(rs.getLong("id"));
        student.setName(rs.getString("name"));
        student.setBirthdate(rs.getDate("birthdate"));
        student.setNumber(rs.getInt("number"));
        return student;
    };
...
@Override
    public int count() {
        return jdbcTemplate.queryForObject("select count(*) from
         public.students", Integer.class);
    }
```

```
    @Override
    public List<Student> findAll() {
        return jdbcTemplate.query("select * from public.students",
tudentMapper);
    }
    @Override
    public List<Student> findAllByName(String name) {
        return namedParameterJdbcTemplate.query(
                "select * from public.students where name ilike '%' || :name ||
'%'",
                Collections.singletonMap("name", name), studentMapper);
    }
}
```

Как это работает (рисунок 7):

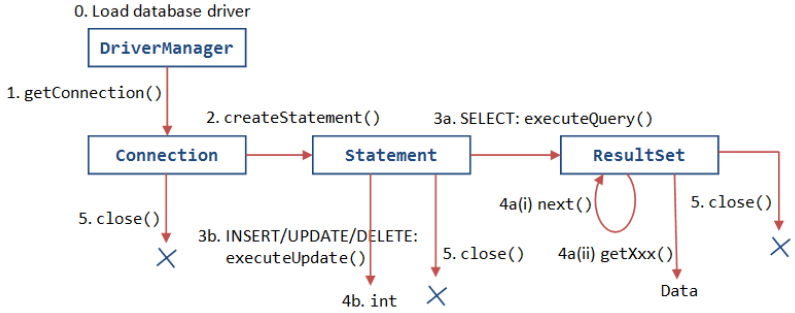

Рисунок 7 – Схема работы сервера с поддержкой REST API

А если кодом.

```
Class.forName("org.postgresql.Driver");

try (Connection connection = DriverManager.getConnection(url,
username, password);
        Statement statement = connection.createStatement();
        ResultSet resultSet = statement.executeQuery("select * from
```

```
public.students")) {
    while (resultSet.next()) {
        ...
    }
}

try (Connection connection = DriverManager.getConnection(url,
properties);
        PreparedStatement statement = connection.prepareStatement(
                "select * from public.students where id = ?")) {
    statement.setLong(1, id);
    ResultSet resultSet = statement.executeQuery();
    if (resultSet.next()) {
        ....
    }
}
```

Результаты выполнения запросов.

http://localhost:8080/api/students


```
[{
  "id":1,
  "name":"Иванов Пётр Сергеевич",
  "birthdate":"2001-01-01",
  "number":123456
},
{
  "id":2,
  "name":"Петров Сергей Иванович",
  "birthdate":"2002-02-02",
  "number":123457
}]
```

http://localhost:8080/api/students/count

2

http://localhost:8080/api/students/Пётр

[{"id":1,"name":"Иванов Пётр Сергеевич","birthdate":"2001-01-01","number":123456}]

Spring IoC

Spring Framework обладает собственной реализацией принципа инверсии управления (IoC). IoC также известен как внедрение зависимостей (DI). Это процесс, в котором объекты определяют свои зависимости (то есть другие объекты, с которыми они работают) только через аргументы конструктора, аргументы фабричного метода или свойства, которые устанавливаются для экземпляра объекта после его создания или возврата из фабричного метода. Затем контейнер внедряет эти зависимости при создании bean-компонента. Этот процесс, по сути, является обратным (отсюда и название Inversion of Control) самому bean-компоненту, управляющему созданием экземпляров или расположением его зависимостей с помощью прямого конструирования классов или механизма, такого как шаблон Service Locator.

Интерфейс BeanFactory предоставляет расширенный механизм настройки, способный управлять любым типом объекта.

ApplicationContext – это расширение BeanFactory. Что он привносит:

- Упрощенная интеграция с функциями АОП Spring.
- Обработка ресурсов сообщений (для использования в интернационализации).
- Публикация событий.

- Специальные контексты прикладного уровня, такие как WebApplicationContext, для использования в веб-приложениях.

BeanFactory предоставляет структуру конфигурации и базовые функции, а ApplicationContext добавляет больше функций энтерпрайз-уровня.

ApplicationContext включает в себя функционал BeanFactory и используется при описании контейнера IoC Spring.

В Spring объекты, формирующие основу вашего приложения и управляемые контейнером Spring IoC, называются bean-компонентами.

Компонент – это объект, который создается, собирается и управляется контейнером Spring IoC. В противном случае bean-компонент – это просто один из многих объектов вашего приложения. Bean-компоненты и зависимости между ними отражаются в метаданных конфигурации, используемых контейнером.

Интерфейс org.springframework.context.ApplicationContext представляет контейнер Spring IoC и отвечает за создание, настройку и сборку компонентов.

Контейнер получает инструкции о том, какие объекты создавать, настраивать и собирать, считывая метаданные конфигурации. Метаданные конфигурации представлены в формате XML, аннотациях Java или коде Java.

Он позволяет вам описывать объекты, из которых состоит ваше приложение, и взаимозависимости между этими объектами.

Несколько реализаций интерфейса ApplicationContext поставляются с Spring. В автономных приложениях обычно создается экземпляр ClassPathXmlApplicationContext или FileSystemXmlApplicationContext.

Хотя XML был традиционным форматом для определения метаданных конфигурации, вы можете указать контейнеру использовать аннотации или код Java в качестве формата метаданных, предоставив небольшой объем конфигурации XML, чтобы декларативно включить поддержку этих дополнительных форматов метаданных.

В большинстве приложений не требуется писать явный пользовательский код для создания одного или нескольких экземпляров контейнера Spring IoC.

Например, для веб-приложения обычно достаточно восьми (или около того) строк стандартного XML-кода веб-дескриптора в файле web.xml приложения.

Если вы используете Spring Tools для Eclipse (среда разработки на основе Eclipse), вы можете легко создать эту шаблонную конфигурацию несколькими щелчками мыши или нажатиями клавиш.

Классы вашего приложения объединяются с метаданными конфигурации, так что после создания и инициализации ApplicationContext у вас есть полностью настроенная и исполняемая система или приложение.

Контейнер Spring IoC использует метаданные конфигурации. Эти метаданные конфигурации определяют, как разработчик приложения указывает контейнеру Spring создавать экземпляры, настраивать и собирать объекты в приложении.

Метаданные конфигурации традиционно предоставляются в простом и интуитивно понятном формате XML, будем его использовать для передачи ключевых концепций и функций контейнера Spring IoC.

Метаданные на основе XML – не единственная разрешенная форма метаданных конфигурации. Сам контейнер Spring IoC полностью отделен от формата, в котором фактически

записываются эти метаданные конфигурации. В наши дни многие разработчики выбирают конфигурацию на основе Java для своих приложений Spring.

Альтернативные виды конфигурации:

- определение bean-компонентов с использованием метаданных конфигурации на основе аннотаций.
- определение bean-компонентов, внешних по отношению к классам вашего приложения, используя файлы Java, а не файлы XML (аннотации @Configuration, @Bean, @Import и @DependsOn).

Конфигурация Spring состоит как минимум из одного и, как правило, более одного определения bean-компонента, которым должен управлять контейнер.

Метаданные конфигурации на основе XML настраивают эти bean-компоненты как элементы <bean/> внутри элемента <beans/> верхнего уровня. Конфигурация Java обычно использует методы с аннотациями @Bean в классе @Configuration.

Эти определения компонентов соответствуют реальным объектам, из которых состоит ваше приложение. Как правило, вы определяете объекты сервисного уровня, объекты персистентного уровня, такие как репозитории или объекты доступа к данным (DAO), объекты представления, такие как веб-контроллеры, объекты инфраструктуры, такие как JPA EntityManagerFactory, очереди JMS и т. д. Как правило, детализированные доменные объекты в контейнере не настраиваются, поскольку за создание и загрузку доменных объектов обычно отвечают репозитории и бизнес-логика.

Пример конфигурации приведён далее.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
      https://www.springframework.org/schema/beans/spring-
beans.xsd">
   <bean id="..." class="...">
      <!-- collaborators and configuration for this bean go here -->
   </bean>
   <bean id="..." class="...">
      <!-- collaborators and configuration for this bean go here -->
   </bean>
   <!-- more bean definitions go here -->
</beans>
```

Путь или пути расположения, предоставляемые конструктору ApplicationContext, представляют собой строки ресурсов, которые позволяют контейнеру загружать метаданные конфигурации из различных внешних ресурсов, таких как локальная файловая система, Java CLASSPATH и т. д.

Рассмотрим пример, в котором сервисный уровень состоит из класса PetStoreServiceImpl и двух объектов доступа к данным типов JpaAccountDao и JpaItemDao (на основе стандарта объектно-реляционного сопоставления JPA). Элемент имени свойства ссылается на имя свойства JavaBean, а элемент ref ссылается на имя другого определения компонента. Эта связь между элементами id и ref выражает зависимость между взаимодействующими объектами. Так выглядит создание контекста:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("services.xml", "daos.xml");
```

Содержимое файла services.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
      https://www.springframework.org/schema/beans/spring-
beans.xsd">
   <!-- services -->
   <bean id="petStore"
class="org.springframework.samples.jpetstore.services.PetStoreServic
eImpl">
      <property name="accountDao" ref="accountDao"/>
      <property name="itemDao" ref="itemDao"/>
      <!-- additional collaborators and configuration for this bean go
here -->
   </bean>
   <!-- more bean definitions for services go here -->
</beans>
```

Настройки daos.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
      https://www.springframework.org/schema/beans/spring-
beans.xsd">
   <bean id="accountDao"
class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao
">
      <!-- additional collaborators and configuration for this bean go
here -->
```

```
    </bean>
    <bean id="itemDao"
class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">
        <!-- additional collaborators and configuration for this bean go
here -->
    </bean>
    <!-- more bean definitions for data access objects go here -->
</beans>
```

Определения bean-компонентов могут охватывать несколько XML-файлов. Часто каждый отдельный файл конфигурации XML представляет собой логический уровень или модуль в вашей архитектуре.

Вы можете использовать конструктор контекста приложения для загрузки определений bean-компонентов из всех этих XML-фрагментов. Этот конструктор принимает несколько местоположений ресурсов. В качестве альтернативы используйте одно или несколько вхождений элемента <import/> для загрузки определений bean-компонентов из другого файла или файлов.

В следующем примере определения внешних компонентов загружаются из трех файлов: services.xml, messageSource.xml и themeSource.xml. Все пути расположения относятся к файлу определения, выполняющему импорт, поэтому services.xml должен находиться в том же каталоге или пути к классам, что и файл, выполняющий импорт, а messageSource.xml и themeSource.xml должны находиться в расположении ресурсов ниже расположения импортируемого файла. Содержимое импортируемых файлов, включая элемент <beans/> верхнего уровня, должно быть допустимым определением компонента XML в соответствии со схемой Spring.

Описания бинов в формате XML, файл beans.xml:

```
<beans>
   <import resource="services.xml"/>
   <import resource="resources/messageSource.xml"/>
   <import resource="/resources/themeSource.xml"/>
   <bean id="bean1" class="..."/>
   <bean id="bean2" class="..."/>
</beans>
```

В качестве еще одного примера внешних метаданных конфигурации определения bean-компонентов также могут быть выражены в Spring Groovy Bean Definition DSL, известном из среды Grails.

Обычно такая конфигурация хранится в файле «.groovy».

Этот стиль конфигурации во многом эквивалентен определениям компонентов XML и даже поддерживает пространства имен конфигурации XML Spring. Он также позволяет импортировать файлы определения компонентов XML с помощью директивы importBeans.

Альтернативное описание beans.groovy.

```
beans {
   dataSource(BasicDataSource) {
      driverClassName = "org.hsqldb.jdbcDriver"
      url = "jdbc:hsqldb:mem:grailsDB"
      username = "sa"
      password = ""
      settings = [mynew:"setting"]
   }
   sessionFactory(SessionFactory) {
      dataSource = dataSource
   }
   myService(MyService) {
```

```
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

Основными признаками и частями Java-конфигурации IoC контейнера являются классы с аннотацией @Configuration и методы с аннотацией @Bean.

Аннотация @Bean используется для указания того, что метод создает, настраивает и инициализирует новый объект, управляемый Spring IoC контейнером.

Такие методы можно использовать как в классах с аннотацией @Configuration, так и в классах с аннотацией @Component (или её наследниках).

Класс с аннотацией @Configuration говорит о том, что он является источником определения бинов.

Простейшая конфигурация выглядит следующим образом.

```
package ssau;
import org.springframework.context.annotation.Configuration;
/**
 * Конфигурационный класс Spring IoC контейнера
 */
@Configuration
public class AppConfiguration {
}
```

Когда методы с аннотацией @Bean определены в классах, не имеющих аннотацию @Configuration, то относятся к обработке в легком режиме, то же относится и к классам с аннотацией

@Component. Иначе, такие методы относятся к полному режиму обработки.

В отличие от полного, в легком режиме @Bean методы не могут просто так объявлять внутренние зависимости. Поэтому, в основном предпочтительно работать в полном режиме, во избежание трудноуловимых ошибок.

Для обычной Java-конфигурации применяется AnnotationConfigApplicationContext, в качестве аргумента к которому передается класс, либо список классов с аннотацией @Configuration, либо с любой другой аннотацией JSR-330, в том числе и @Component.

```
public class Starter {
   private static final Logger logger = 
LogManager.getLogger(Starter.class);
   public static void main(String[] args) {
      logger.info("Starting configuration...");
      ApplicationContext context =
            new
AnnotationConfigApplicationContext(AppConfiguration.class);
   }
}
```

Чтобы объявить бин, достаточно указать аннотацию @Bean тому методу, который возвращает тип бина как в классах с аннотацией @Configuration, так и в классах с аннотацией @Component (или её наследниках). Например, определим интерфейс какого-нибудь сервиса и его реализацию.

```
package ssau.services;
public interface GreetingService {
   String sayGreeting();
}
```

```
package ssau.services;
public class GreetingServiceImpl implements GreetingService {
   @Override
   public String sayGreeting() {
      return "Greeting, user!";
   }
}
```

Теперь, для того, чтобы объект с типом GreetingService был доступен для использования, необходимо описать его в конфигурации следующим образом:

```
@Configuration
public class AppConfiguration {
   @Bean
   GreetingService greetingService() {
      return new GreetingServiceImpl();
   }
}
```

А для того, чтобы использовать его, достаточно выполнить следующее:

```
public class Starter {
   public static void main(String[] args) {
      ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfiguration.class);
      GreetingService greetingService =
context.getBean(GreetingService.class);
   }
}
```

Метод getBean() может принимать в качестве аргумента как класс, так и названия бина, либо другие варианты, с которыми вы можете ознакомится в документации. Однако такой подход не рекомендуется использовать в production-конфигурациях, т.к. для подобных целей существует механизм Dependency Injection (DI), собственно говоря, для чего и предназначен Spring IoC контейнер.

Именовать бины принято в соответствии со стандартным соглашением по именованию полей Java-классов. Т.е. имена бинов должны начинаться со строчной буквы и быть в "верблюжьей" нотации.

По умолчанию, так, как будет назван метод определения бина, по такому имени и нужно получать бин через метод getBean() или автоматическое связывание. Однако вы можете переопределить это имя или указать несколько псевдонимов, через параметр name аннотации @Bean. Выглядеть это будет примерно так:

```
@Bean(name = "gServiceName")
@Bean(name = {"gServiceName", "gServiceAnotherNamed"})
```

Иногда полезно предоставить более подробное описание бина, например, в целях мониторинга. Для этого существует аннотация @Description.

Для управления контейнером жизненным циклом бина, вы можете реализовать метод afterPropertiesSet() интерфейса InitializingBean и метод destroy() интерфейса DisposableBean.

Метод afterPropertiesSet() позволяет выполнять какие-либо действия после инициализации всех свойств бина контейнером, метод destroy() выполняется при уничтожении бина контейнером. Однако их не рекомендуется использовать, поскольку они дублируют код Spring.

Как вариант, предпочтительно использовать методы с JSR-250 аннотациями @PostConstruct и @PreDestroy. Также существует

вариант определить аналогичные методы как параметры аннотации @Bean, например, так:
@Bean(initMethod = "initMethod", destroyMethod = "destroyMethod")

При совместном использовании методов, интерфейсов и аннотаций, описанных выше, учитывайте их порядок вызовов. Для методов инициализации порядок будет следующий:

- Методы с аннотациями @PostConstruct в порядке их определения в классе.
- Метод afterPropertiesSet().
- Метод, указанный в параметре initMethod аннотации @Bean.

Для методов разрушения порядок будет следующий:

- Методы с аннотациями @PreDestroy в порядке их определения в классе.
- Метод destroy().
- Метод, указанный в параметре destroyMethod аннотации @Bean.

Spring IoC контейнеру требуются метаданные для конфигурации. Одна из таких аннотаций – это @Bean.

Другой основной аннотацией является @Component, а также её наследники @Repository, @Service и @Controller. Все они являются общими шаблонами для любых компонентов, управляемыми контейнером.

@Repository, @Service и @Controller рекомендуется использовать в тех случаях, когда вы можете отнести аннотируемый класс к определенному слою, например DAO, либо когда вам необходима поддержка функциональности, которую предоставляет аннотация. Также эти аннотации могут иметь

дополнительный смысл в будущих версиях Spring Framework. В остальных же случаях достаточно использовать аннотацию @Component.

Для того, чтобы ваша конфигурация могла знать о таких компонентах и вы могли бы их использовать, существует специальная аннотация для класса вашей конфигурации @ComponentScan.

По умолчанию, такая конфигурация сканирует на наличие классов с аннотацией @Component и его потомков в том пакете, в котором сама находится, а также в подпакетах.

Однако, если вы хотите, чтобы сканирование было по определенным каталогам, то это можно настроить, просто добавив в аннотацию @ComponentScan параметр basePackages с указанием одного или нескольких пакетов. Выглядеть это будет примерно таким образом:

```
@ComponentScan(basePackages = "ssau.services")
```

Классу GreetingServiceImpl при этом необходимо добавить аннотацию @Component.

Аннотация @Required применяется к setter-методу бина и указывает на то, чтобы соответствующее свойство метода было установлено на момент конфигурирования значением из определения бина или автоматического связывания. Если же значение не будет установлено, будет выброшено исключение. Использование аннотации позволит избежать NullPointerException в процессе использования свойства бина.

Пример использования:

```
package ssau.services;
public class GreetingServiceImpl implements GreetingService {
   private ApplicationContext context;
   @Required
```

```
    public void setContext(ApplicationContext context) {
        this.context = context;
    }
}
```

Когда вы создаете определение бинов, вы создаете правило для создания экземпляров класса, который определяет бин. Важно понять, что определение бинов является правилом, потому что оно означает, множество экземпляров какого класса вы можете создать по этому правилу.

Вы можете контролировать не только какие зависимости и значения конфигурации вы можете подключить в объекте, который создан из определения бина, но также область видимости из того же определения бина. Это мощный и гибкий подход, при котором вы можете выбрать область видимости создаваемых объектов. Изначально, Spring Framework поддерживает несколько вариантов, некоторые доступны, только если вы используете web-aware ApplicationContext. Также вы можете создать свою собственную область видимости.

Для того чтобы указать область видимости бина, отличный от singleton, необходимо добавить аннотацию @Scope("область_видимости") методу объявления бина или классу с аннотацией @Component:

```
@Component
@Scope("prototype")
public class GreetingServiceImpl implements GreetingService {
 //...
}
```

singleton – по умолчанию. Spring IoC контейнер создает единственный экземпляр бина. Как правило, используется для бинов без сохранения состояния (stateless).

prototype – Spring IoC контейнер создает любое количество экземпляров бина. Новый экземпляр бина создается каждый раз, когда бин необходим в качестве зависимости, либо через вызов getBean(). Как правило, используется для бинов с сохранением состояния (stateful).

request – жизненный цикл экземпляра ограничен единственным HTTP запросом; для каждого нового HTTP запроса создается новый экземпляр бина. Действует, только если вы используете web-aware ApplicationContext.

session – жизненный цикл экземпляра ограничен в пределах одной и той же HTTP Session. Действует, только если вы используете web-aware ApplicationContext.

global session – жизненный цикл экземпляра ограничен в пределах глобальной HTTP Session (обычно при использовании portlet контекста). Действует, только если вы используете web-aware ApplicationContext.

application – жизненный цикл экземпляра ограничен в пределах ServletContext. Действует, только если вы используете web-aware ApplicationContext.

Как упоминалось выше, классы с аннотацией @Configuration указывают на то, что они являются источниками определения бинов, public-методов с аннотацией @Bean.

Кода бин имеет зависимость от другого бина, то зависимость выражается просто как вызов метода:

```
@Configuration
@ComponentScan
public class AppConfiguration {
   @Bean
```

```
    BeanWithDependency beanWithDependency() {
        return new BeanWithDependency(greetingService());
    }
    @Bean
    GreetingService greetingService() {
        return new GreetingServiceImpl();
    }
}
```

Однако работает такое взаимодействие только в @Configuration-классах, в @Component-классах такое не работает.

Большая часть приложений строится по модульной архитектуре, разделенная по слоям, например DAO, сервисы, контроллеры и др. Создавая конфигурацию, можно также её разбивать на составные части, что также улучшит читабельность и понимание архитектуры вашего приложения. Для этого в конфигурацию необходимо добавить аннотацию @Import, в параметрах которой указываются другие классы с аннотацией @Configuration.

```
@Configuration
public class AnotherConfiguration {
    @Bean
    BeanWithDependency beanWithDependency() {
        return new BeanWithDependency();
    }
}
@Configuration
@ComponentScan
@Import(AnotherConfiguration.class)
public class AppConfiguration {
    @Bean
```

```
    GreetingService greetingService() {
        return new GreetingServiceImpl();
    }
}
```

Зачастую встречаются случаи, когда бин в одной конфигурации имеет зависимость от бина в другой конфигурации. Поскольку конфигурация является источником определения бинов, то разрешить такую зависимость не является проблемой, достаточно объявить поле класса конфигурации с аннотацией @Autowired.

```
@Configuration
public class AnotherConfiguration {
    @Autowired
    GreetingService greetingService;
    @Bean
    BeanWithDependency beanWithDependency() {
        return new BeanWithDependency(greetingService);
    }
}
```

Классы с аннотацией @Configuration не стремятся на 100% заменить конфигурации на XML, при этом, если вам удобно или имеется какая-то необходимость в использовании XML конфигурации, то к вашей Java-конфигурации необходимо добавить аннотацию @ImportResource, в параметрах которой необходимо указать нужное вам количество XML-конфигураций.

Java-конфигурация:

```
@Configuration
@ImportResource("classpath:/lessons/xml-config.xml")
public class AppConfiguration {
```

```
    @Value("${jdbc.url}")
    String url;
 //...
}
```

Содержимое файла xml-config.xml:

```
<beans>
    <context:property-placeholder
location="classpath:/jdbc.properties"/>
</beans>
```

Файл свойств jdbc.properties:

```
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
```

IoC контейнер выполняет разрешение зависимостей бинов в следующем порядке:

- Создается и инициализируется ApplicationContext с метаданными конфигурации, которые описывают все бины. Эти метаданные могут быть описаны через XML, Java-код или аннотации.
- Для каждого бина и его зависимостей вычисляются свойства, аргументы конструктора или аргументы статического фабричного метода, либо обычного(без аргументов) конструктора. Эти зависимости предоставляются бину, когда он (бин) уже создан. Сами зависимости инициализируются рекурсивно, в зависимости от вложенности в себе других бинов. Например, при инициализации бина A, который имеет зависимость B, а B зависит от C, сначала инициализируется бин C, потом B, а уже потом A.

- Каждому свойству или аргументу конструктора устанавливается значение или ссылка на другой бин в контейнере.
- Для каждого свойства или аргумента конструктора подставляемое значение конвертируется в тот формат, который указан для свойства или аргумента. По умолчанию Spring может конвертировать значения из строкового формата во все встроенные типы, такие как int, long, String, boolean и др.

Spring каждый раз при создании контейнера проверяет конфигурацию каждого бина. И только бины с областью видимости singleton создаются сразу вместе со своими зависимостями, в отличие от остальных, которые создаются по запросу и в соответствии со своей областью видимости. В случае цикличной зависимости (когда класс A требует экземпляр B, а классу B требуется экземпляр A) Spring IoC контейнер обнаруживает её и выбрасывает исключение BeanCurrentlyInCreationException.

Spring контейнер может разрешать зависимости между бинами через autowiring (автоматическое связывание). Данный механизм основан на просмотре содержимого в ApplicationContext и имеет следующие преимущества:

- Автоматическое связывание позволяет значительно сократить количество инструкций для указания свойств или аргументов конструктора.
- Автоматическое связывание позволяет обновлять конфигурацию, несмотря на развитие ваших объектов. К примеру, вам необходимо добавить зависимость в классе и эта зависимость может быть разрешена без необходимости модификации конфигурации. Поэтому

автоматическое связывание может быть особенно полезным при разработке, не исключая возможность переключения на явное описание, когда кодовая база будет стабильна.

Примеры связывания:

```
public class AutowiredClass {
   @Autowired
   @Qualifier("main")
   private GreetingService greetingService;
   @Autowired // в виде массива или коллекции
   private GreetingService[] services;
   @Autowired // Map, где ключами являются имена бинов,
значения - сами бины
   private Map<String, GreetingService> serviceMap;
   @Autowired // необязательная
   public AutowiredClass(@Qualifier("main") GreetingService service)
{}
   @Autowired // обычный метод с произвольным названием
аргументов и их количеством
   public void prepare(GreetingService prepareContext){/* что-то
делаем... */}
   @Autowired // традиционный setter-метод
   public void setContext(GreetingService service) {
this.greetingService = service; }
}
```

Профиль является именованной группой определений бинов, зарегистрированных контейнером только в том случае, если профиль активен.

Профиль определения бинов позволяет зарегистрировать различные бины для различных окружений.

Например, у вас есть два стенда приложения, testing, где вы проводите тестирование на наличие ошибок, и production, собственно работающее приложение.

При этом у вас наверняка будут различаться какие-то параметры в окружениях, такие как подключение к БД и др.

Профили вам очень помогут, т.к. вам не придется каждый раз изменять что-либо и пересобирать для каждого окружения.

Конфигурация с профилями:

```
@Configuration
public class AppConfig {
   // Для testing
   @Bean
   @Profile("testing")
   public DataSource devDataSource() {
      return new EmbeddedDatabaseBuilder()
         .setType(EmbeddedDatabaseType.HSQL)
         .addScript("classpath:schema.sql")
         .addScript("classpath:test-data.sql")
         .build();
   }
   // Для production
   @Bean
   @Profile("production")
   public DataSource productionDataSource() throws Exception {
      Context ctx = new InitialContext();
      return (DataSource)
ctx.lookup("java:comp/env/jdbc/datasource");
   }
}
```

Если конфигурация имеет аннотацию @Profile, то все бины в ней и конфигурации, указанные в аннотации @Import, будут действовать, если активен соответствующий профиль, указанный в @Profile конфигурации.

Например, если конфигурация имеет аннотацию @Profile({"p1", "p2"}), то она будет действовать, если профили "p1" или "p2" будут активны. При @Profile({"p1", "!p2"}) конфигурация действует, если профиль "p1" активен или профиль "p2" не активен.

Профили можно активировать несколькими способами, программно:

```
AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("p1");
ctx.register(SomeConfig.class, StandaloneDataConfig.class,
JndiDataConfig.class);
ctx.refresh();
```

Установкой значения переменной среды:

```
-Dspring.profiles.active="p1,p2"
```

Через системное свойство JVM и даже через JNDI.

Вы также можете указать профиль по умолчанию через @Profile("default"), которое будет действовать только в том случае, если не задано ни одно из других свойств. Также вы можете задать свои значения профилей по умолчанию через метод Environment#setDefaultProfiles, либо через переменную среды spring.profiles.default.

Свойства играют важную роль практически во всех приложениях и могут иметь происхождение из различных

источников: свойства файлов, системные свойства JVM, окружение операционной системы, JNDI и др.

Свойства представлены набором объектов PropertySource и значения в итоге получаются из тех же источников, как System.getProperties() и System.getenv().

Для получения значений применяется несколько способов.

Получение свойств:

```
@Configuration
@PropertySource("classpath:/com/${my.placeholder:default/path}/app
.properties")
public class AppConfig {
   @Autowired
   Environment env;
   @Bean
   public TestBean testBean() {
      TestBean testBean = new TestBean();
      testBean.setName(env.getProperty("testbean.name"));
      return testBean;
   }
}
```

В условной конфигурации мы сообщаем контекст билдеру, что данную конфигурацию мы создаем, только в случае наличия положительного значения константы project.mq.enabled в файле настроек.

```
@ConditionalOnProperty(value="project.mq.enabled",
matchIfMissing = false)
@Configuration
public class JmsConfig {
 ...
}
```

Теперь перейдем к зависимым бинам и пометим их аннотацией ConditionalOnBean, которая не даст Spring создать бины, зависимые от нашей конфигурации.

```
@ConditionalOnBean(JmsConfig.class)
@Component
public class JmsConsumer {
 ...
}
```

Виды @Conditional

- ConditionalOnBean – условие выполняется, если присутствует нужный бин в BeanFactory.
- ConditionalOnClass – условие выполняется, если нужный класс есть в classpath.
- ConditionalOnCloudPlatform – условие выполняется, когда активна определенная платформа.
- ConditionalOnExpression – условие выполняется, когда SpEL выражение вернуло положительное значение.
- ConditionalOnJava – условие выполняется, когда приложение запущено с определенной версией JVM.
- ConditionalOnJndi – условие выполняется, только если через JNDI доступен определенный ресурс.
- ConditionalOnMissingBean – условие выполняется, в случае если нужный бин отсутствует в BeanFactory.
- ConditionalOnMissingClass – условие выполняется, если нужный класс отсутствует в classpath.
- ConditionalOnNotWebApplication – условие выполняется, если контекст приложения не является веб контекстом.

- ConditionalOnProperty – условие выполняется, если в файле настроек заданы нужные параметры.
- ConditionalOnResource – условие выполняется, если присутствует нужный ресурс в classpath.
- ConditionalOnSingleCandidate – условие выполняется, если bean-компонент указанного класса уже содержится в BeanFactory и он единственный.
- ConditionalOnWebApplication – условие выполняется, если контекст приложения является веб контекстом.

Автоматическая настройка Spring Boot пытается автоматически настроить приложение Spring на основе добавленных вами зависимостей jar. Например, если HSQLDB находится в вашем пути к классам (classpath), и вы не настроили вручную никаких компонентов соединения с базой данных, то Spring Boot автоматически конфигурирует базу данных в памяти.

Если Вам необходимо включить автоматическую настройку, добавьте аннотации @EnableAutoConfiguration или @SpringBootApplication к одному из ваших классов @Configuration.

Вам следует добавлять только одну аннотацию @SpringBootApplication или @EnableAutoConfiguration. Как правило, рекомендуется добавлять один или другой только в ваш основной класс @Configuration.

Автоконфигурация неинвазивна. В любой момент вы можете начать определять свою собственную конфигурацию для замены определенных частей автоконфигурации. Например, если вы добавите свой собственный DataSource бин, поддержка встроенной базы данных по умолчанию отступит.

Если вам нужно выяснить, какая автоконфигурация применяется и почему, запустите ваше приложение с ключом --

debug. Это включает журналы отладки для выбора основных регистраторов и записывает отчет о состоянии в консоль.

ORM

ORM (Object-Relational Mapping, объектно-реляционное отображение, или преобразование) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных» (рисунок 8).

Использование реляционной базы данных для хранения объектно-ориентированных данных приводит к семантическому разрыву, заставляя программистов писать программное обеспечение, которое должно уметь обрабатывать данные в объектно-ориентированном виде, а хранить эти данные в реляционной форме.

Эта постоянная необходимость в преобразовании между двумя разными формами данных не только сильно снижает производительность, но и создаёт трудности для программистов, так как обе формы данных накладывают ограничения друг на друга.

Рисунок 8 – ORM

ORM решения устраняют необходимость в преобразовании объектов для хранения в реляционных базах данных.

Некоторые пакеты решают эту проблему, предоставляя библиотеки классов, способных выполнять такие преобразования автоматически. Имея список таблиц в базе данных и объектов в программе, они автоматически преобразуют запросы из одного вида в другой. В результате запроса объекта «человек» необходимый SQL-запрос будет сформирован и выполнен, а результаты «волшебным» образом преобразованы в объекты «номер телефона» внутри программы.

С точки зрения программиста система должна выглядеть как постоянное хранилище объектов. Он может просто создавать объекты и работать с ними как обычно, а они автоматически будут сохраняться в реляционной базе данных.

На практике всё не так просто и очевидно. Все системы ORM обычно проявляют себя в том или ином виде, уменьшая в некотором роде возможность игнорирования базы данных. Более того, слой транзакций может быть медленным и неэффективным (особенно в терминах сгенерированного SQL). Всё это может привести к тому, что программы будут работать медленнее и использовать больше памяти, чем программы, написанные «вручную».

Но ORM избавляет программиста от написания большого количества кода, часто однообразного и подверженного ошибкам, тем самым значительно повышая скорость разработки. Кроме того, большинство современных реализаций ORM позволяют программисту при необходимости самому жёстко задать код SQL-запросов, который будет использоваться при тех или иных действиях (сохранение в базу данных, загрузка, поиск и т. д.) с постоянным объектом.

JPA (Java/Jakarta Persistence API) – это спецификация Java EE и Java SE, описывающая систему управления сохранением Java объектов в таблицы реляционных баз данных в удобном виде.

Сама Java не содержит реализации JPA, однако существует много реализаций данной спецификации от разных компаний (открытых и нет). Это не единственный способ сохранения Java объектов в базы данных (ORM систем), но один из самых популярных в Java мире.

Hibernate – одна из самых популярных открытых реализаций последней версии спецификации (JPA 2.1). Даже скорее самая популярная, почти стандарт де-факто. То есть JPA только описывает правила и API, а Hibernate реализует эти описания, впрочем у Hibernate (как и у многих других реализаций JPA) есть дополнительные возможности, не описанные в JPA (и не переносимые на другие реализации JPA).

JPA (Java Persistence API) и Java Data Objects (JDO) – две спецификации сохранения Java объектов в базах данных. Если JPA сконцентрирована только на реляционных базах, то JDO – более общая спецификация, которая описывает ORM для любых возможных баз и хранилищ. В принципе, можно рассматривать JPA как специализированную на релятивистских базах часть спецификации JDO, даже при том, что API этих двух спецификаций не полностью совпадает. Также отличаются «разработчики» спецификаций – если JPA разрабатывается как JSR, то JDO сначала разрабатывался как JSR, теперь разрабатывается как проект Apache JDO.

Entity класс студента (Student):

```
package com.ssau.study.orm;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import java.util.Date;
```

```
@Entity
@Table(name = "students", schema = "public")
@Getter
@Setter
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    private String name;

    @Temporal(value = TemporalType.DATE)
    private Date birthdate;

    private int number;

    @ManyToOne(fetch = FetchType.EAGER, cascade =
CascadeType.ALL)
    @JoinColumn(name = "group_id", foreignKey =
@ForeignKey(ConstraintMode.NO_CONSTRAINT))
    private Group group;
}
```

Entity класс группы (Group):

```
package com.ssau.study.orm;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import java.util.List;
```

```
@Entity
@Table(name = "groups", schema = "public", uniqueConstraints = {
@UniqueConstraint(columnNames = "name") })
@Getter
@Setter
public class Group {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "name", columnDefinition = "text", length = 100,
nullable = false, unique = true)
    private String name;

    @OneToMany(mappedBy = "group", fetch = FetchType.LAZY,
cascade = CascadeType.ALL)
    private List<Student> students;
}
```

Spring Data JPA

Spring Data JPA, часть более крупного семейства Spring Data (рисунок 9), упрощает реализацию репозиториев на основе JPA. Этот модуль имеет дело с расширенной поддержкой уровней доступа к данным на основе JPA (рисунок 10). Это упрощает создание приложений на базе Spring, использующих технологии доступа к данным.

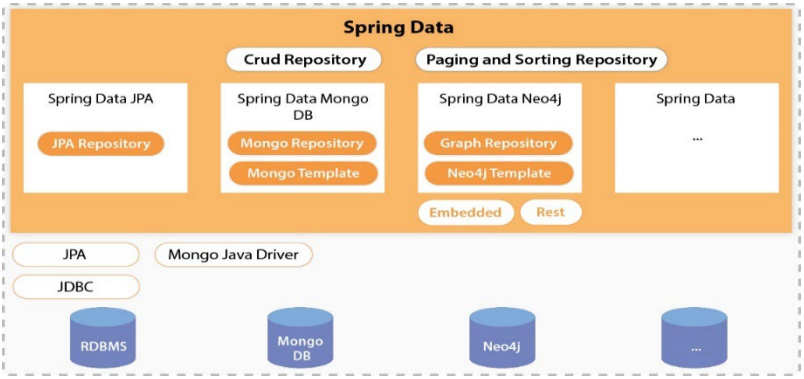

Рисунок 9 – Spring Data

Реализация слоя доступа к данным приложения долгое время была громоздкой. Необходимо было писать слишком много стандартного кода для выполнения простых запросов, а также для выполнения разбиения на страницы и прочего. Spring Data JPA стремится значительно улучшить реализацию уровней доступа к данным, сократив усилия до того объема, который действительно необходим (рисунок 11). Как разработчик, вы пишете свои интерфейсы репозитория, включая пользовательские методы поиска, а Spring автоматически предоставит реализацию.

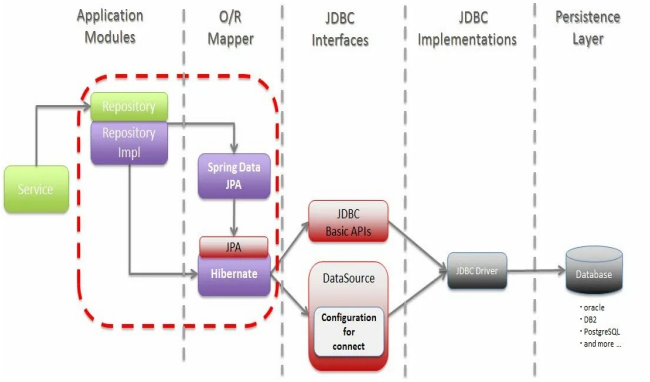

Рисунок 10 – Spring Data JPA

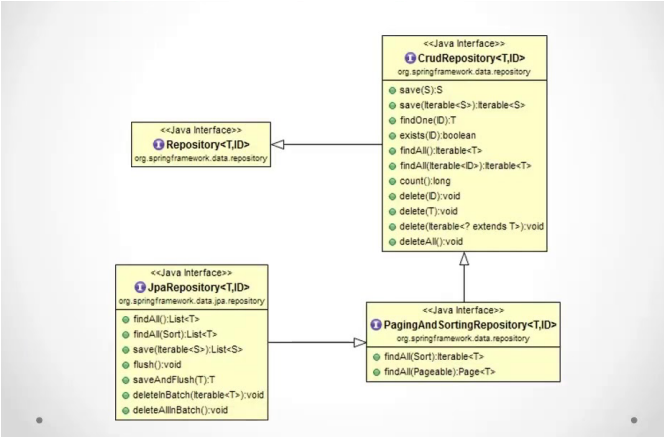

Рисунок 11 – Иерархия интерфейсов

Функции:

- Усовершенствованная поддержка создания репозиториев на основе Spring и JPA.
- Поддержка предикатов Querydsl и, следовательно, типобезопасных запросов JPA.
- Прозрачный аудит сущностных классов.
- Поддержка разбивки на страницы, динамическое выполнение запросов, возможность интеграции пользовательского кода доступа к данным.
- Проверка аннотированных запросов @Query во время начальной загрузки.
- Поддержка сопоставления сущностей на основе XML.
- Конфигурация репозитория на основе JavaConfig путем введения @EnableJpaRepositories.

StudentRepository.java для работы со студентами:

```
package com.ssau.study.jpa;

import com.ssau.study.orm.Student;
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import org.springframework.data.jpa.repository.Query;

import java.util.List;

public interface StudentRepository extends
      JpaRepository<Student, Long> {
   @Query(value = "select * from public.students where name ilike '%'
|| :name || '%'", nativeQuery = true)
   List<Student> selectByName(String name);
}
```

GroupRepository.java для работы с группами:

```
package com.ssau.study.jpa;

import com.ssau.study.orm.Group;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface GroupRepository extends
      JpaRepository<Group, Long> {
   List<Group> findAllByNameContainingIgnoreCase(String name);
}
```

Примеры запросов JPQL:

```
SELECT a FROM Author a ORDER BY a.firstName, a.lastName;
SELECT DISTINCT a FROM Author a INNER JOIN a.books b
WHERE b.publisher.name = :name;
SELECT DISTINCT pr FROM Person pr LEFT JOIN pr.phones ph
WHERE ph IS NULL OR ph.type = :type;
```

```
SELECT DISTINCT pr FROM Person pr LEFT JOIN FETCH
pr.phones;
SELECT count(c), sum(c.duration), min(c.duration), max(c.duration),
avg(c.duration) FROM Call c;
SELECT length(p.name) FROM Person p;
UPDATE Person p SET p.name = :newName WHERE p.name =
:oldName;
DELETE FROM Person p WHERE p.name = :name;
```

Именованные запросы:

```
@NamedQueries({
  @NamedQuery(name = "Book.findAllJPQL",
    query = "SELECT b FROM Book b ORDER BY b.title DESC"),
  @NamedQuery(name = "Book.findByTitleJPQL",
    query = "SELECT b FROM Book b WHERE b.title = ?1"),
  @NamedQuery(name =
 "Book.findByTitleAndPagesGreaterThanJPQL",
    query = "SELECT b FROM Book b WHERE b.title = :title AND
 b.pages > :pages")
})
@NamedNativeQueries({
  @NamedNativeQuery(name = "Book.findAllNative",
    query = "SELECT * FROM book b ORDER BY b.title DESC",
    resultClass = Book.class),
  @NamedNativeQuery(name = "Book.findByIsbnNative",
    query = "SELECT * FROM book b WHERE b.isbn = :isbn",
    resultClass = Book.class)
})
@Entity
public class Book {
    // ...
```

```
}
```

Именованные запросы в Spring Data:

```
public interface BookRepository extend
    JpaRepository<Book, Long> {
  // named queries declared with `@NamedQuery`
  List<Book> findAllJPQL();

  List<Book> findByTitleJPQL(String title);

  List<Book> findByTitleAndPagesGreaterThanJPQL(
    @Param("title") String title, @Param("pages") int pages);

  // named queries declared with `@NamedNativeQuery`
  @Query(nativeQuery = true)
  List<Book> findAllNative();

  @Query(nativeQuery = true)
  List<Book> findByIsbnNative(@Param("isbn") String isbn);
}
```

Примеры сгенерированных запросов:

```
interface PersonRepository extends JpaRepository<Person, Long> {
  List<Person>        findByEmailAddressAndLastname(EmailAddress
emailAddress, String lastname);

  // Enables the distinct flag for the query
  List<Person>      findDistinctPeopleByLastnameOrFirstname(String
lastname, String firstname);
  List<Person>      findPeopleDistinctByLastnameOrFirstname(String
lastname, String firstname);
```

```
    // Enabling ignoring case for an individual property
    List<Person> findByLastnameIgnoreCase(String lastname);

    // Enabling ignoring case for all suitable properties
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String
lastname, String firstname);

    // Enabling static ORDER BY for a query
    List<Person> findByLastnameOrderByFirstnameAsc(String
lastname);
    List<Person> findByLastnameOrderByFirstnameDesc(String
lastname);

    // Limiting results
    Page<Person> queryFirst10ByLastname(String lastname, Pageable
pageable);
    Slice<Person> findTop3ByLastname(String lastname, Pageable
pageable);
    List<Person> findFirst10ByLastname(String lastname, Sort sort);
}
```

На слое сервисов расположена бизнес-логика обработки данных:

```
package com.ssau.study.service;

import com.ssau.study.dto.GroupPojo;
import com.ssau.study.dto.StudentPojo;
import com.ssau.study.jpa.GroupRepository;
import com.ssau.study.jpa.StudentRepository;
import com.ssau.study.orm.Group;
```

```
import com.ssau.study.orm.Student;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;

@Service
@RequiredArgsConstructor
public class GroupService {
    private final GroupRepository groupRepository;
    private final StudentRepository studentRepository;

    public List<GroupPojo> findAll(String name) {
        List<GroupPojo> result = new ArrayList<>();
        for (Group group : name == null ? groupRepository.findAll() :
groupRepository.findAllByNameContainingIgnoreCase(name)) {
            result.add(GroupPojo.fromEntity(group));
        }
        return result;
    }

    public StudentPojo create(long groupId, StudentPojo pojo) {
        Student student = StudentPojo.toEntity(pojo);

student.setGroup(groupRepository.findById(groupId).orElseThrow());
        return StudentPojo.fromEntity(studentRepository.save(student));
    }
}
```

POJO классы служат для передачи данных между слоями, а также между клиентом и сервером:

```
package com.ssau.study.dto;

import com.ssau.study.orm.Student;
import lombok.Getter;
import lombok.Setter;

import java.util.Date;

@Getter
@Setter
public class StudentPojo {
    private long id;
    private String name;
    private Date birthdate;
    private int number;

    public static StudentPojo fromEntity(Student student) {
        StudentPojo pojo = new StudentPojo();
        pojo.setId(student.getId());
        pojo.setName(student.getName());
        pojo.setBirthdate(student.getBirthdate());
        pojo.setNumber(student.getNumber());
        return pojo;
    }

    public static Student toEntity(StudentPojo pojo) {
        Student student = new Student();
        student.setId(pojo.getId());
        student.setName(pojo.getName());
```

```
        student.setBirthdate(pojo.getBirthdate());
        student.setNumber(pojo.getNumber());
        return student;
    }
}

package com.ssau.study.dto;

import com.ssau.study.orm.Group;
import com.ssau.study.orm.Student;
import lombok.Getter;
import lombok.Setter;

import java.util.ArrayList;
import java.util.List;

@Getter
@Setter
public class GroupPojo {
    private long id;
    private String name;
    private List<StudentPojo> students;

    public static GroupPojo fromEntity(Group group) {
        GroupPojo pojo = new GroupPojo();
        pojo.setId(group.getId());
        pojo.setName(group.getName());

        List<StudentPojo> students = new ArrayList<>();
        pojo.setStudents(students);
        for (Student student : group.getStudents()) {
```

```
            students.add(StudentPojo.fromEntity(student));
        }

        return pojo;
    }
}
```

Контроллер предназначен для обработки запросов, пришедших по определённому адресу:

```
package com.ssau.study.controller;

import com.ssau.study.dto.GroupPojo;
import com.ssau.study.dto.StudentPojo;
import com.ssau.study.service.GroupService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/v2")
public class GroupController {
    @Autowired
      private GroupService groupService;

    @GetMapping("/groups")
    public List<GroupPojo> findAll() {
        return groupService.findAll(null);
    }

    @GetMapping("/groups/{name}")
```

```
    public List<GroupPojo> findAllByName(@PathVariable String name) {
        return groupService.findAll(name);
    }

    @PostMapping("/groups/{groupId}/students")
    public StudentPojo createStudent(@PathVariable long groupId, @RequestBody StudentPojo pojo) {
        return groupService.create(groupId, pojo);
    }
}
```

Spring Security

Spring Security – это мощная и настраиваемая среда аутентификации и контроля доступа. Это стандарт де-факто для защиты приложений на основе Spring.

Spring Security – это фреймворк, ориентированный на обеспечение как аутентификации, так и авторизации приложений Java. Как и во всех проектах Spring, реальная сила Spring Security заключается в том, насколько легко его можно расширить для удовлетворения пользовательских требований.

Возможности:

- Комплексная и расширяемая поддержка аутентификации и авторизации.
- Защита от атак, таких как фиксация сеанса, кликджекинг, подделка межсайтовых запросов и т.д.
- Интеграция API сервлетов.
- Дополнительная интеграция с Spring Web MVC.
- И многое-многое другое…

Далее приведена конфигурация Spring Boot в Maven для настроек безопасности:

```
<dependencies>
  <!-- ... other dependency elements ... -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
</dependencies>
<dependencies>
  <!-- ... other dependency elements ... -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
  </dependency>
</dependencies>
```

Spring Boot конфигурируется автоматически:

- Включает конфигурацию Spring Security по умолчанию, которая создает фильтр сервлета в виде bean-компонента с именем springSecurityFilterChain.
- Этот bean-компонент отвечает за всю безопасность (защиту URL-адресов приложений, проверку введенных имени пользователя и паролей, перенаправление на форму входа и т. д.) в вашем приложении.

- Создает bean-компонент UserDetailsService с именем пользователя и случайно сгенерированным паролем, который регистрируется в консоли.
- Регистрирует фильтр с помощью bean-компонента с именем springSecurityFilterChain в контейнере сервлетов для каждого запроса.

Spring Boot мало что настраивает, но многое делает. Кратко об особенностях:

- Требует авторизованного пользователя для любого взаимодействия с приложением.
- Создаёт форму входа по умолчанию.
- Разрешает пользователю с именем пользователя и паролем, зарегистрированным в консоли, пройти аутентификацию с помощью аутентификации на основе форм.
- Защищает хранилище паролей с помощью BCrypt.
- Позволяет пользователю выйти из системы.
- Предотвращает CSRF-атаки.
- Защищает от фиксации сеанса.
- Интеграция заголовков безопасности (интеграция X-Content-Type-Options, интеграция X-XSS-Protection, интеграция X-Frame-Options для предотвращения кликджекинга).
- Интеграция со следующими методами Servlet API:
  - HttpServletRequest#getRemoteUser();
  - HttpServletRequest#getUserPrincipal();
  - HttpServletRequest#isUserInRole(java.lang.String);
  - HttpServletRequest#login(java.lang.String, java.lang.String);
  - HttpServletRequest#logout().

CSRF (cross-site request forgery – «межсайтовая подделка запроса», также известна как XSRF) – вид атак на посетителей веб-сайтов, использующий недостатки протокола HTTP. Если жертва заходит на сайт, созданный злоумышленником, от её лица тайно отправляется запрос на другой сервер (например, на сервер платёжной системы), осуществляющий некую вредоносную операцию (например, перевод денег на счёт злоумышленника). Для осуществления данной атаки жертва должна быть аутентифицирована на том сервере, на который отправляется запрос, и этот запрос не должен требовать какого-либо подтверждения со стороны пользователя, которое не может быть проигнорировано или подделано атакующим скриптом.

Данный тип атак, вопреки распространённому заблуждению, появился достаточно давно, первые уязвимости были обнаружены в 2000 году. А сам термин ввёл Питер Уоткинс в 2001 году.

Основное применение CSRF – вынуждение выполнения каких-либо действий на уязвимом сайте от лица жертвы (изменение пароля, секретного вопроса для восстановления пароля, почты, добавление администратора и т. д.).

Атака может осуществляться путём размещения на веб-странице ссылки или скрипта, пытающегося получить доступ к сайту, на котором атакуемый пользователь заведомо (или предположительно) уже аутентифицирован.

Фиксация сеанса – это очень распространенный и наиболее частый тип атаки, при котором злоумышленник может создать сеанс, зайдя на сайт, а затем убедить другого пользователя войти в систему с помощью того же сеанса (отправив ему ссылку, содержащую сеанс, идентификатор в качестве параметра, например).

Фиксации сеанса – это своего рода уязвимость, когда злоумышленник обманом заставит вас войти в приложение, а

затем использует ваш сеанс, чтобы получить доступ к тому же сайту.

Это отличается от перехвата сеанса. При перехвате сеанса злоумышленник украдет ваш аутентифицированный сеанс, чтобы получить доступ к приложению.

При фиксации сеанса злоумышленник сначала получит действительный сеанс из приложения, а затем перенаправит пользователя на страницу входа, чтобы вы могли войти в систему, а злоумышленник мог использовать сеанс для входа в приложение.

Кликджекинг - это механизм обмана пользователей интернета, при котором злоумышленник может получить доступ к конфиденциальной информации или даже получить доступ к компьютеру пользователя, заманив его на внешне безобидную страницу или внедрив вредоносный код на безопасную страницу.

Принцип основан на том, что поверх видимой страницы располагается невидимый слой, в который и загружается нужная злоумышленнику страница, при этом элемент управления (кнопка, ссылка), необходимый для осуществления требуемого действия, совмещается с видимой ссылкой или кнопкой, нажатие на которую ожидается от пользователя.

Возможны различные применения технологии – от подписки на ресурс в социальной сети до кражи конфиденциальной информации и совершения покупок в интернет-магазинах за чужой счёт.

Фильтры безопасности вставляются в FilterChainProxy с помощью API SecurityFilterChain (рисунок 12). Порядок экземпляров фильтра имеет значение. Обычно нет необходимости знать порядок фильтров, однако бывают случаи, когда это полезно.

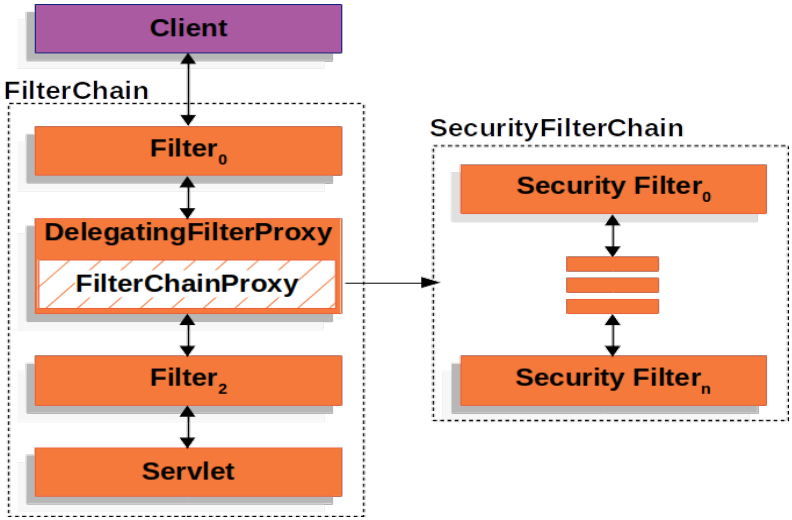

Рисунок 12 – SecurityFilterChain

Далее приведен полный упорядоченный список фильтров Spring Security:

ForceEagerSessionCreationFilter, ChannelProcessingFilter, WebAsyncManagerIntegrationFilter, SecurityContextPersistenceFilter, HeaderWriterFilter, CorsFilter, CsrfFilter, LogoutFilter, OAuth2AuthorizationRequestRedirectFilter, Saml2WebSsoAuthenticationRequestFilter, X509AuthenticationFilter, AbstractPreAuthenticatedProcessingFilter, CasAuthenticationFilter, OAuth2LoginAuthenticationFilter, Saml2WebSsoAuthenticationFilter, UsernamePasswordAuthenticationFilter, DefaultLoginPageGeneratingFilter, DefaultLogoutPageGeneratingFilter, ConcurrentSessionFilter, DigestAuthenticationFilter, BearerTokenAuthenticationFilter, BasicAuthenticationFilter, RequestCacheAwareFilter, SecurityContextHolderAwareRequestFilter, JaasApiIntegrationFilter, RememberMeAuthenticationFilter, AnonymousAuthenticationFilter,

OAuth2AuthorizationCodeGrantFilter, SessionManagementFilter, ExceptionTranslationFilter, FilterSecurityInterceptor, SwitchUserFilter

Мы рискуем зайти слишком далеко, чтобы подробно рассмотреть каждый фильтр в этой цепочке, но вот пояснения для некоторых из этих фильтров. Вы можете просмотреть исходный код Spring Security, чтобы понять другие фильтры.

- BasicAuthenticationFilter: пытается найти HTTP-заголовок базовой аутентификации в запросе и, если он найден, пытается аутентифицировать пользователя с помощью имени пользователя и пароля заголовка.
- UsernamePasswordAuthenticationFilter: пытается найти параметр запроса имени пользователя/пароля/тело POST и, если он найден, пытается аутентифицировать пользователя с помощью этих значений.
- DefaultLoginPageGeneratingFilter: создает для вас страницу входа, если вы явно не отключили эту функцию. Этот фильтр – причина, по которой вы получаете страницу входа по умолчанию при включении Spring Security.
- DefaultLogoutPageGeneratingFilter: создает для вас страницу выхода, если вы явно не отключили эту функцию.
- FilterSecurityInterceptor: делает вашу авторизацию.

На рисунке 13 приведена схема механизма обработки исключений.

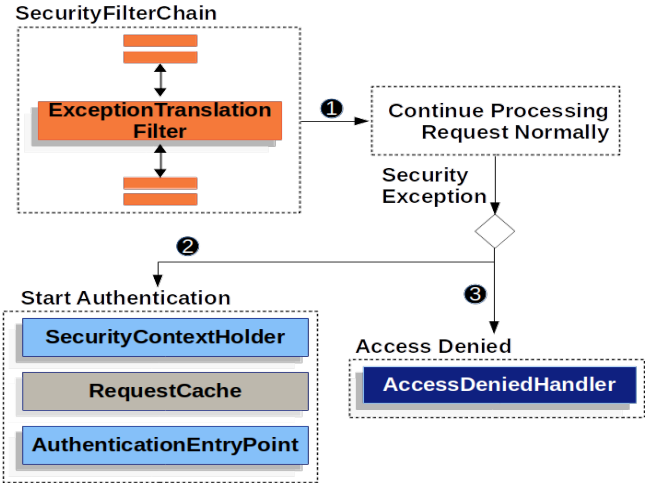

Рисунок 13 – Обработка исключений

Spring Security обеспечивает всестороннюю поддержку аутентификации. Механизмы аутентификации:

- Username and Password – как пройти аутентификацию с помощью имени пользователя/пароля.
- OAuth 2.0 Login – вход OAuth 2.0 с помощью OpenID Connect и нестандартный вход OAuth 2.0 (например, GitHub).
- SAML 2.0 Login – вход в систему SAML 2.0.
- Remember Me – как запомнить пользователя после истечения сеанса.
- JAAS Authentication – аутентификация с помощью JAAS.
- Сценарии Pre-Authentication – аутентификация с помощью внешнего механизма, такого как SiteMinder или безопасность Java EE, но Spring Security используется для авторизации и защиты.
- X509 Authentication – аутентификация X509.

Основные архитектурные компоненты Spring Security, используемые для сервлетов аутентификации:

- SecurityContextHolder – это место, где Spring Security хранит сведения о том, кто прошел проверку подлинности.
- SecurityContext – получается из SecurityContextHolder и содержит Authentication текущего аутентифицированного пользователя.
- Authentication – может подаваться на вход в AuthenticationManager для предоставления учетных данных, полученных от пользователя при аутентификации, или текущего пользователя из SecurityContext.
- GrantedAuthority – полномочие, которое предоставляется принципалу при аутентификации (т. е. роли, области действия и т. д.).
- AuthenticationManager – API, который определяет, как фильтры Spring Security выполняют аутентификацию.
- ProviderManager – наиболее распространенная реализация AuthenticationManager.
- AuthenticationProvider – используется ProviderManager для выполнения определенного типа аутентификации.
- Запрос учетных данных с AuthenticationEntryPoint – используется для запроса учетных данных от клиента (т. е. перенаправление на страницу входа в систему, отправка ответа WWW-Authenticate и т. д.)
- AbstractAuthenticationProcessingFilter – базовый фильтр, используемый для аутентификации. Он дает представление о высокоуровневой последовательности аутентификации и о том, как все части работают вместе.

В основе модели аутентификации Spring Security лежит SecurityContextHolder (рисунок 14). Он содержит SecurityContext.

Контейнер для контекста безопасности SecurityContextHolder – это хранилище, где Spring Security держит сведения о том, кто прошел аутентификацию.

Spring Security не заботится о том, как заполняется SecurityContextHolder. Если он содержит значение, он используется в качестве текущего аутентифицированного пользователя.

Рисунок 14 – SecurityContextHolder

SecurityContext содержит объект Authentication.

Интерфейс Authentication служит двум основным целям в Spring Security:

- Входные данные для AuthenticationManager для предоставления учетных данных, предоставленных пользователем для аутентификации. При использовании в этом сценарии isAuthenticated() возвращает false.
- Представляет текущего аутентифицированного пользователя. Текущую аутентификацию можно получить из SecurityContext.
- Объект Authentication содержит:
- принципал: идентифицирует пользователя. При аутентификации с использованием имени

пользователя/пароля он часто является экземпляром UserDetails.

- учетные данные: зачастую это пароль. Во многих случаях очищается после аутентификации пользователя, чтобы предотвратить утечку.
- полномочия: экземпляры GrantedAuthority представляют собой разрешения высокого уровня, предоставляемые пользователю. Два примера – роли и области действия.

Фильтр AbstractAuthenticationProcessingFilter (рисунок 15) используется в качестве базового фильтра для аутентификации учетных данных пользователя. Прежде чем учетные данные могут быть аутентифицированы, Spring Security обычно запрашивает учетные данные с помощью AuthenticationEntryPoint.

Затем данный фильтр может аутентифицировать любые отправленные ему запросы аутентификации.

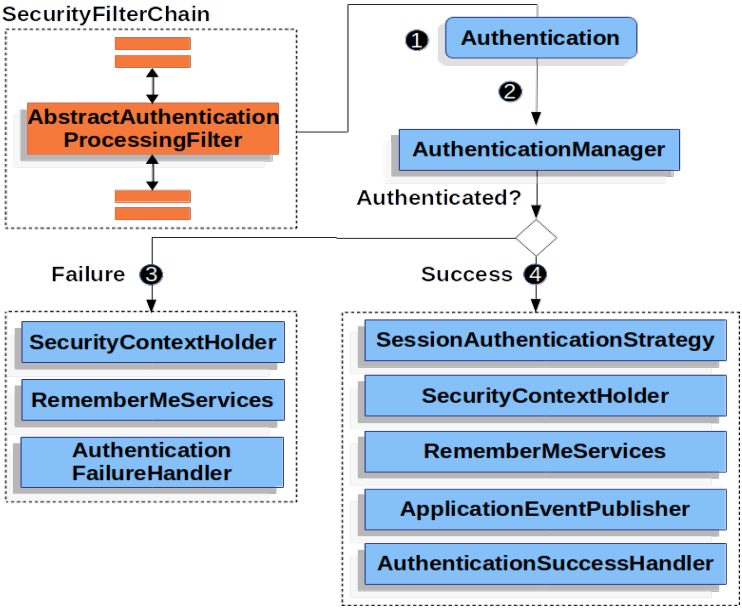

Рисунок 15 – AbstractAuthenticationProcessingFilter

Одним из наиболее распространенных способов аутентификации пользователя является проверка имени пользователя и пароля (Username/Password Authentication).

Spring Security обеспечивает всестороннюю поддержку аутентификации с помощью имени пользователя и пароля и предоставляет встроенные механизмы для чтения имени пользователя и пароля из HttpServletRequest:

- Form;
- Basic;
- Digest.

Конфигурируем Spring Security:

```
@Configuration
@EnableWebSecurity // (1)
public class WebSecurityConfig { // (1)
   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception { // (2)
      http
            .csrf().disable()
            .authorizeHttpRequests((requests) -> requests
              .requestMatchers("/", "/home", "/login").permitAll() // (3)
              .requestMatchers("/api/admin/*").hasRole("ADMIN") // (4)
                  .anyRequest().authenticated() // (5)
            )
            .formLogin((form) -> form
              .loginPage("/login.html") // (6)
              .loginProcessingUrl("/login") // (6)
              .defaultSuccessUrl("/index.html") // (6)
              .failureUrl("/login.html?error=true") // (6)
              .permitAll()
            )
```

```
        .logout(LogoutConfigurer::permitAll); // (7)

    return http.build();
}
...
```

1. Обычный бин Spring @Configuration с аннотацией @EnableWebSecurity.

2. Кастомизируя бин SecurityFilterChain, вы получаете приятный маленький DSL, с помощью которого вы можете настроить свою аутентификацию.

3. Запросы, идущие к /, /home и /login, разрешены – пользователю не нужно проходить аутентификацию.

4. Запросы, идущие по адресам /api/admin/*, требуют, чтобы пользователь был аутентифицирован и имел роль ADMIN. Использование antMatcher допускает в маске пути наличие подстановочных знаков (*, **,?).

5. Любой другой запрос требует, чтобы пользователь сначала прошел аутентификацию, т. е. пользователь должен войти в систему.

6. Вы разрешаете вход в форму (имя пользователя/пароль в форме) с пользовательской страницей входа (/login.html, т.е. не автоматически сгенерированной Spring Security). Обработка введённых пользователем данных делается по адресу /login. После успешного входа пользователь перенаправляется на страницу /index.html. Любой должен иметь возможность получить доступ к странице входа в систему без необходимости сначала входить в систему (permitAll; в противном случае у нас была бы «Ловушка 22»).

7. То же самое касается страницы выхода.

Конфигурируем дальше:

```
...
```

```
    @Bean
    public PasswordEncoder passwordEncoder() { // (8)
        return new BCryptPasswordEncoder();
    }

    @Bean
    public UserDetailsService userDetailsService() { // (9)
        UserDetails user =
                User.withUsername("user")
                        .password(passwordEncoder().encode("password"))
                        .roles("USER")
                        .build();
        UserDetails admin =
                User.withUsername("admin")
                        .password(passwordEncoder().encode("password"))
                        .roles("ADMIN")
                        .build();

        return new InMemoryUserDetailsManager(user, admin);
    }
}
```

8. Определим простой BCryptPasswordEncoder как бин в нашей конфигурации. Это стандартный бин, который будет использоваться Spring Security для шифрования паролей. Старые реализации, такие как SHAPasswordEncoder, требуют, чтобы клиент задал значение «соли» при кодировании пароля. Вместо этого BCrypt сгенерирует случайную «соль». Это важно понимать, потому что это означает, что каждый вызов будет иметь разный результат, поэтому нам нужно кодировать пароль только один раз. Создание бина со скоупом «одиночка» решает данный вопрос.

9. В данном случае для реализации UserDetailsService используется механизм in-memory хранения предварительно заданных пользователей. Это стандартный подход для демо-приложений, когда не требуется реализация полного функционала по работе с пользователями. Разумеется, для больших приложений требуется обеспечивать работу с базой через репозитории Spring Data JPA, чтобы информация о пользователях не пропадала вместе с электричеством.

Страница resources/static/login.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Login Page</title>
</head>
<body>
  <form action="/login" method="post">
    Пользователь: <input type="text" name="username"
id="username" />
    <br/>
    Пароль: <input type="password" name="password"
id="password" />
    <br/>
    <input type="submit" value="Войти" />
  </form>
</body>
</html>
```

Аннотация @Secured используется для указания списка ролей в методе. Таким образом, пользователь может получить доступ к этому методу только в том случае, если у него есть хотя бы одна

из указанных ролей. Здесь аннотация @Secured("ROLE_VIEWER") определяет, что только пользователи с ролью ROLE_VIEWER могут выполнять метод getUsername.

```
@Secured("ROLE_VIEWER")
public String getUsername() {
   SecurityContext securityContext =
SecurityContextHolder.getContext();
   return securityContext.getAuthentication().getName();
}
```

Кроме того, мы можем определить список ролей в аннотации @Secured.

```
@Secured({ "ROLE_VIEWER", "ROLE_EDITOR" })
public boolean isValidUsername(String username) {
   return userRoleRepository.isValidUsername(username);
}
```

В этом случае в конфигурации указано, что если у пользователя есть ROLE_VIEWER или ROLE_EDITOR, этот пользователь может вызвать метод isValidUsername. Аннотация @Secured не поддерживает язык выражений Spring (SpEL).

Аннотация @RolesAllowed является эквивалентной аннотацией JSR-250 аннотации @Secured. По сути, мы можем использовать аннотацию @RolesAllowed аналогично @Secured. Таким образом, мы могли бы переопределить методы getUsername и isValidUsername.

```
@RolesAllowed("ROLE_VIEWER")
public String getUsername2() {
   //...
}
```

```
@RolesAllowed({ "ROLE_VIEWER", "ROLE_EDITOR" })
public boolean isValidUsername2(String username) {
   //...
}
```

Точно так же только пользователь с ролью ROLE_VIEWER может выполнить getUsername2. Опять же, пользователь может вызвать isValidUsername2 только в том случае, если у него есть хотя бы одна из ролей ROLE_VIEWER или ROLER_EDITOR.

Аннотации @PreAuthorize и @PostAuthorize обеспечивают управление доступом на основе выражений: предикаты можно писать с помощью SpEL (Spring Expression Language).

Аннотация @PreAuthorize проверяет данное выражение перед входом в метод, тогда как аннотация @PostAuthorize проверяет его после выполнения метода и может изменить результат.

Рассмотрим метод getUsernameInUpperCase.

```
@PreAuthorize("hasRole('ROLE_VIEWER')")
public String getUsernameInUpperCase() {
   return getUsername().toUpperCase();
}
```

@PreAuthorize("hasRole('ROLE_VIEWER')") имеет то же значение, что и @Secured("ROLE_VIEWER").

Следовательно, аннотацию @Secured({"ROLE_VIEWER", "ROLE_EDITOR"}) можно заменить на:

```
@PreAuthorize("hasRole('ROLE_VIEWER') or
hasRole('ROLE_EDITOR')")
public boolean isValidUsername3(String username) {
   //...
}
```

Более того, мы можем использовать аргумент метода как часть выражения.

```
@PreAuthorize("#username == authentication.principal.username")
public String getMyRoles(String username) {
   //...
}
```

Здесь пользователь может вызывать метод getMyRoles только в том случае, если значение аргумента имя пользователя совпадает с именем пользователя текущего принципала.

Стоит отметить, что выражения @PreAuthorize можно заменить выражениями *@PostAuthorize.*

```
@PostAuthorize("#username == authentication.principal.username")
public String getMyRoles2(String username) {
   //...
}
```

Однако в этом примере авторизация происходила после выполнения целевого метода. Аннотация @PostAuthorize предоставляет возможность доступа к результату метода.

```
@PostAuthorize("returnObject.username ==
authentication.principal.nickName")
public CustomUser loadUserDetail(String username) {
   return userRoleRepository.loadUserByUserName(username);
}
```

Здесь метод loadUserDetail будет успешно выполнен только в том случае, если имя пользователя возвращенного CustomUser равно псевдониму текущего принципала проверки подлинности.

Spring Security предоставляет аннотацию @PreFilter для фильтрации аргумента коллекции перед выполнением метода.

```
@PreFilter("filterObject != authentication.principal.username")
public String joinUsernames(List<String> usernames) {
   return usernames.stream().collect(Collectors.joining(";"));
}
```

В этом примере мы объединяем все имена пользователей, кроме того, который прошел проверку подлинности. Здесь в нашем выражении мы используем имя filterObject для представления текущего объекта в коллекции.

Однако, если метод имеет более одного аргумента, который является типом коллекции, нам нужно использовать свойство filterTarget, чтобы указать, какой аргумент мы хотим отфильтровать.

```
@PreFilter(value = "filterObject != authentication.principal.username", filterTarget = "usernames")
public String joinUsernamesAndRoles(List<String> usernames, List<String> roles) {
   return usernames.stream().collect(Collectors.joining(";"))
      + ":" + roles.stream().collect(Collectors.joining(";"));
}
```

Кроме того, мы также можем отфильтровать возвращаемую коллекцию метода, используя аннотацию @PostFilter.

```
@PostFilter("filterObject != authentication.principal.username")
public List<String> getAllUsernamesExceptCurrent() {
   return userRoleRepository.getAllUsernames();
}
```

В этом случае имя filterObject относится к текущему объекту в возвращаемой коллекции. С этой конфигурацией Spring Security

будет перебирать возвращаемый список и удалять все значения, соответствующие имени пользователя принципала.

Обычно мы оказываемся в ситуации, когда мы защищаем разные методы, используя одну и ту же конфигурацию безопасности. В этом случае мы можем определить мета-аннотацию безопасности.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasRole('VIEWER')")
public @interface IsViewer {
}
```

Затем мы можем напрямую использовать аннотацию @IsViewer для защиты нашего метода.

```
@IsViewer
public String getUsername4() {
   //...
}
```

Мета-аннотации безопасности – отличная идея, потому что они добавляют больше семантики и отделяют бизнес-логику от структуры безопасности.

Если одна и та же аннотация безопасности используется для каждого метода в одном классе, можно рассмотреть возможность размещения этой аннотации на уровне класса.

```
@Service
@PreAuthorize("hasRole('ROLE_ADMIN')")
public class SystemService {
   public String getSystemYear(){
      //...
   }
```

```
    public String getSystemDate(){
        //...
    }
}
```

Также можно использовать несколько аннотаций безопасности в одном методе.

```
@PreAuthorize("#username == authentication.principal.username")
@PostAuthorize("returnObject.username == authentication.principal.nickName")
public CustomUser securedLoadUserDetail(String username) {
    return userRoleRepository.loadUserByUserName(username);
}
```

Библиографический список

1. JavaRush. Часть 2. Поговорим немного об архитектуре ПО [сайт] – URL: https://javarush.com/groups/posts/2519-chastjh-2-pogovorim-nemnogo-ob-arkhitekture-po (дата обращения 20.09.2023).

2. JavaRush. Часть 3. Протоколы HTTP/HTTPS [сайт] – URL: https://javarush.com/groups/posts/2521-chastjh-3-protokolih-httphttps (дата обращения 20.09.2023)

3. Javastudy. Spring Core [сайт] – URL: https://javastudy.ru/frameworks/spring/spring-core/ (дата обращения 21.09.2023)

4. Habr. Введение в Spring Boot: создание простого REST API на Java [сайт] – URL: https://habr.com/ru/post/435144/ (дата обращения 22.09.2023)

5. Skillfactory. Maven [сайт] – URL: https://blog.skillfactory.ru/glossary/maven/ (дата обращения 22.09.2023)