Pavel A. Stepanov, pavel@stepanoff.me
Степанов П.А.

pavel@stepanoff.me

Технологии разработки серверных информационных систем

Методическое пособие

ГУАП

Санкт-Петербург

-2018-

Оглавление

Введение 4
График выполнения лабораторных работ 5
Варианты заданий 6
Технические требования 7
Требования к отчету 8
Конфигурирование рабочего места 9
Требования к рабочему месту 9
Установка Java 9
Установка IDE 9
Краткое описание системы сборки Maven 9
Лабораторная работа №1. Разработка простого серверного приложения J2EE с использованием сервлетов 11
Архитектура простого веб-приложения. 11
Протокол http. 11
Cookie. 14
Http сессия. 15
Структура класса сервлета. 16
Структура класса фильтра. 17
Задание на лабораторную работу. 17
Лабораторная работа №2. Разработка ресурса REST/JSON сервиса 19
Предпосылки появления Web API 19
SOAP 19
JSON 20
RESTful API 21
Инструменты разработчика 23
Реализация REST средствами Spring-boot 23
Развитие REST – Query Data API 25
Задание на лабораторную работу. 26
Лабораторная работа №3. Разработка простого AJAX приложения Spring 27
Архитектура клиентского представления. 27
Технология Ajax. 27
Библиотека AngularJS. 28
Задание на лабораторную работу. 30

Лабораторная работа №4. Разработка формы логина 31
Основы авторизации и аутентификации. Виды аутентификации. 31
Виды атак на веб-приложение. 31
Снифферы и атаки “Man-in-the-middle”. 32
Защита от атак XSS и CSRF. 34
Хеширование паролей. 35
Basic и Digest аутентификация 36
Аутентификация через директории пользователей (Kerberos, LDAP). 36
Внешняя аутентификация через клиент (OpenID, OAuth). 37
SSO аутентификация. TBD 38
Реалиация аутентификации авторизации средствами Spring. 39
Задание на лабораторную работу. 44
Лабораторная работа №5. Разработка приложения с использованием thymeleaf 45
Шаблоны страниц JSP. 45
Шаблоны страниц Spring. 45
Шаблоны страниц Thymeleaf 45
Задание на лабораторную работу. 45
Лабораторная работа №6. Разработка приложения с использованием Hibernate 46
Основные понятия JDBC. TBD 46
ORM. Виды ORM. TBD 46
JPA. TBD 46
Spring-data-jpa. TBD 46
Задание на лабораторную работу. 46
Лабораторная работа №7. Разработка приложения с асинхронной очередью сообщений 47
Концепции EDA. Читатели-писатели и производители-потребители TBD 47
Zookeeper. TBD 47
Kafka. TBD 47
Задание на лабораторную работу. 47
Лабораторная работа №8. Разработка микросервиса 48
Виртуализация и связь с автоматическим развертыванием. TBD 48
Docker. TBD 48
Mesos и его экосфера. TBD 48
Kubernates и его экосфера TBD. 48

Задание на лабораторную работу. 48
Список литературы. 48
Интернет-источники. 48

Введение

Предмет “Технологии разработки серверных информационных систем” призван дать обучающимся представление о методах построения современных приложений, размещаемый в сети Интернет (интранет). Лабораторные работы по этому предмету знакомят обучающихся со следующими технологиями:

- Общее представление о современных тенденциях в разработке для web
- Структура веб-серверов, фильтры и сервлеты, сессии
- Интерфейсы JSON и SOAP. Подход REST.
- инъекция зависимостей и инверсия управления (DI, IoC, Spring)
- аспектно-ориентированное программирование (AOP, AspectJ)
- генерация страниц на сервере (Thymeleaf)
- генерация страниц на клиенте (AJAX)
- объектно-реляционные отображения (ORM, Hibernate, JPA)
- архитектура, управляемая событиями (EDA)
- Облачная архитектура
- Контейнерное развертывание

Учебные материалы, в том числе видео лекций, доступны онлайн по адресу

http://stepanoff.info/trsis/index.html

Репозиторий с примерами представлен по адресу

https://github.com/wildpierre/trsissamples

Pavel A. Stepanov, pavel@stepanoff.me

График выполнения лабораторных работ

Максимальное количество баллов за лабораторные работы – 72.

Для допуска к экзамену либо зачету требуется сдать ВСЕ лабораторные работы.

В таблице представлен максимальный рейтинг за лабораторную работу в зависимости от недели сдачи

	Лаб. 1	Лаб. 2	Лаб. 3	Лаб. 4	Лаб. 5	Лаб. 6	Лаб. 7	Лаб. 8
Нед. 1	9	9	9	9	9	9	9	9
Нед. 2	9	9	9	9	9	9	9	9
Нед. 3	8	9	9	9	9	9	9	9
Нед. 4	8	9	9	9	9	9	9	9
Нед. 5	7	8	9	9	9	9	9	9
Нед. 6	7	8	9	9	9	9	9	9
Нед. 7	6	7	8	9	9	9	9	9
Нед. 8	6	7	8	9	9	9	9	9
Нед. 9	5	6	7	8	9	9	9	9
Нед. 10	5	6	7	8	9	9	9	9
Нед. 11	4	5	6	7	8	9	9	9
Нед. 12	4	5	6	7	8	9	9	9
Нед. 13	3	4	5	6	7	8	9	9
Нед. 14	3	4	5	6	7	8	9	9
Нед. 15	2	3	4	5	6	7	8	9
Нед. 16	2	3	4	5	6	7	8	9
Нед. 17	1	2	3	4	5	6	7	8
Нед. 18	1	2	3	4	5	6	7	8

Варианты заданий

1. Книжный магазин либо библиотека
2. Поликлиника (запись на прием к врачу)
3. Расписание занятий в институте
4. Расписание поездов, самолетов, кораблей
5. Планирование покупок в магазине
6. Учет оценок студентов за лабораторные работы
7. Ведение списка литературы согласно последнему актуальному ГОСТу
8. Учет трат в бюджете семьи
9. Складской учет
10. Магазин электроники
11. Спортивные соревнования
12. Продажа автомобилей
13. Торговля акциями на бирже
14. Сдача недвижимости в аренду
15. Коллекционирование (нумизматика, филателия и пр).
16. Система безопасности предприятия (помещения, люди, права на вход)
17. Учет конфигурации сетевого оборудования сети предприятия
18. Учет показателей потребления электроэнергии и водоснабжения
19. База данных с рецептами блюд
20. База данных предметов живописи, скульптуры и пр. (музей)
21. Магазин компьютерных игр
22. Магазин музыкальных композиций
23. Учет содержимого холодильника (со сроками годности)
24. Учет медицинских лабораторных показателей пациента поликлиники
25. Ведение списка группы (ФИО, вариант задания, число сданных лабораторных, рейтинг) с учетом постоянно меняющегося количества студентов

Технические требования

Рекомендуется разрабатывать программное обеспечение на платформе Java SE 8 http://www.oracle.com/technetwork/java/javase/downloads/index.html и Spring 4. Spring рекомендуется использовать в варианте spring-boot.

В качестве базы данных рекомендуется использовать СУБД, не требующее установки, как например Derby или H2.

В качестве билд-системы рекомендуется использовать Maven 2 https://maven.apache.org/

В качестве сервера приложений рекомендуется использовать встроенный в приложение tomcat (автоматически загружается и развертывается maven/spring-boot).

Выполнение вышеуказанных рекомендаций позволит сократить набор инсталлируемых приложений до только платформы Java SE а набор загружаемых библиотек только до Maven 2. Все остальные библиотеки и ресурсы Maven загрузит и установит самостоятельно.

В качестве IDE рекомендуется использовать IntelliJ Idea, Eclipse или Netbeans

Использование других платформ разрешается, но Вы должны твердо осознавать, что можете остаться без технической поддержки преподавателем.

Требования к отчету

Отчет должен содержать

- Титульный лист
- Текст и вариант задания
- Описание разрабатываемого продукта
- Текст основных фрагментов кода
- Отчет должен быть отправлен в личный кабинет студента через систему документооборота ГУАП

Конфигурирование рабочего места

Требования к рабочему месту

На компьютере пользователя должна быть установлена операционная система, поддерживаемая Oracle Java 8 (Windows, Red Hat Linux, Ubuntu, MacOS). Полный список поддерживаемых платформ можно получить на сайте Oracle

http://www.oracle.com/technetwork/java/javase/certconfig-2095354.html

На компьютере должен присутствовать интернет. Предполагаемый объем трафика, необходимого для выполнения лабораторной работы – несколько гигабайт.

Внимание. Интернет необходим только во время разработки. Конечный результат можно демонстрировать без интернета (при условии соблюдения двух условий – используется spring-boot и встроенный tomcat).

Установка Java

Установите последнюю версию Java 8 с сайта Oracle.

http://www.oracle.com/technetwork/java/index.html

Обратите внимание, что это должен быть Java Development Kit (JDK), в не Java Runtime Environment (JRE)

Установка IDE

Установите одну из следующих IDE

Netbeans 8.2 https://netbeans.org/

IntelliJ Idea Community Edition https://www.jetbrains.com/idea/

Eclipse http://www.eclipse.org/downloads/

Краткое описание системы сборки Maven

Внимание. Все, кому больше нравится Gradle, могут использовать Gradle.

Maven https://maven.apache.org является системой сборки программных продуктов с поддержкой жизненного цикла. Ее преимуществом является наличие центрального репозитория, содержащего официально выпущенные артефакты различных проектов. В

процессе работы maven загружает актуальные версии библиотек в каталог ~/.m2 на Unix/Linux или %HOMEPATH%/.m2 на Windows и в дальнейшем компонует из них Ваш проект. Поэтому данный каталог может достигать достаточно большого размера.

Проект на базе maven состоит из pom.xml файла (называемого также pom-файлом) – дескриптора проекта и каталога src с кодом. В каталоге src присутствуют два подкаталога – main и test. Первом хранится код продукта, а во втором – тестов.

В процессе сборки могут появляться другие каталоги, в частности, каталог target, который хранит артефакты сборки.

Вызов системы сборки maven может осуществляться как через ide (обычно в этом случае детали скрыты от пользователя), так и непосредственно командой “mvn цель” в каталоге с pom-файлом. Обычно используются следующие основные цели:

Цель **compile** – компиляция проекта

Цель **package** - компиляция проекта и сборка дистрибутива

Цель **install** – компиляция проекта, сборка дистрибутива и установка его в локальный репозиторий (чтобы другие проекты могли его использовать).

См. Также https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

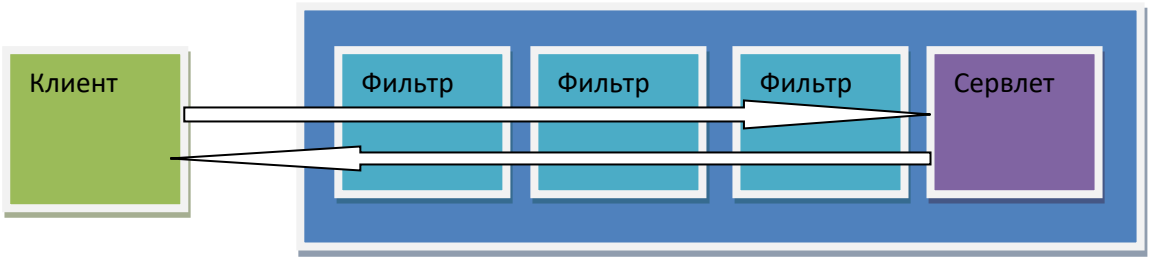
Кроме того, дополнительные используемые фреймворки могут добавлять свои цели. В частности, запуск проекта spring-boot осуществляется целью **spring-boot:run** . Некоторые IDE скрывают эти цели от пользователя и преобразуют в обычные команды (построить, запустить и т.п.).

Лабораторная работа №1. Разработка простого серверного приложения J2EE с использованием сервлетов

Архитектура простого веб-приложения.

В современной IT – индустрии одним из наиболее важных типов приложений являются веб-приложения. Значительная часть таких приложений публицует данные используя технологию сервлетов. В данной лабораторной работе студент должен создать простое веб – приложение на языке Java, использующее фильтры и сервлеты.

Простое приложение, построенное в архитектуре сервлетов, представляет собой пользовательский код, выполненный в одном из двух форматов – сервлета или фильтра. Запрос клиента по протоколу http проходит через цепочку фильтров и доходит до сервлета. Сервлет генерирует ответ и посылает его обратно клиенту. В ряде случаев один из фильтров может принять решение о том, что запрос не должен быть пропущен дальше и сгенерировать ответ самостоятельно.



Протокол http.

Протокол http (hypertext transfer protocol) является текстовым протоколом, выполненным на основе протокола tcp. Типовое общение между клиентом и сервером состоит из выполнения запроса (Request) и получения ответа (Response).

Запрос состоит из:

- URL – идентификатора запрашиваемого ресурса
- Метода запроса
- Заголовков запроса
- Тела запроса

Ответ состоит из

- Кода ответа
- Тела ответа
- Заголовков ответа

URL представляет собой адрес ресурса, которому отправляется запрос. Он формируется из протокола, имени сервера, номера порта и собственно пути, например, http://localhost:8080/mypp/myresource представляет собой получение ресурса /myapp/myresource с сервера localhost через порт 8080 по протоколу http. Данный путь является абстракцией и может обрабатываться веб-сервером по своему усмотрению, т.е. совершенно не обязательно что физически существует какой-то ресурс, соответствующий этому пути. Кроме того, URL может содержать список параметров в форме пар ключ=значение, разделенные амперсандом. Список параметров отделяется от основного пути знаком вопроса, например, http://localhost:8080/mypp/myresource?id=5

Метод запроса представляет конкретное действие, которое необходимо выполнить. Существует несколько стандартных методов, кроме того, ряд серверов позволяют определять собственные. Каждый метод имеет определенные ограничения и семантику, которая является не обязательной, но желательной.

Таблица 1. Виды стандартных методов HTTP

Метод	Стандартная семантика	Запрос имеет тело	Ответ имеет тело	Идемпотентность*	Кешируется
GET	Получить ресурс. Параметры обычно передаются через URL и их размер существенно ограничен.	Опционально	Да	Да	Да
POST	Создать ресурс. Параметры обычно передаются через тело.	Да	Да	Нет	Да
HEAD	Получить заголовки ответа GET	Нет	Нет	Да	Да
PUT	Изменить ресурс	Да	Да	Да	Нет
DELETE	Удалить ресурс	Нет	Да	Да	Нет

CONNECT	Создать тоннель	Да	Да	Нет	Нет
OPTIONS	Получить список поддерживаемых методов	Опционально	Да	Да	Нет
TRACE	Получить эхо запроса (чтобы проверить не вносят ли промежуточные сервера в него изменения)	Нет	Да	Да	Нет
PATCH	Частичная модификация ресурса	да	да	Нет	Нет

- Свойство идемпотентности означает, что несколько одинаковых запросов в рамках одной сессии приведут к тому же результату, что и один запрос.

Обычно сервер обрабатывает как минимум методы GET и POST.

Заголовки запроса – пары вида ключ-значение. Они используются для того чтобы передать серверу сопроводительную информацию. Таким образом можно указывать данные авторизации, информацию о формате запроса и тому подобную сервисную информацию. Можно также передавать серверу любые произвольные заголовки (предполагая, что пользовательский код на сервере сможет их обработать).

Наконец, тело запроса представляет ту информацию, которая отправляется на сервер. Запрос может и не иметь тела, если вся необходимая информация сосредоточена в заголовках и URL (типично для методов GET и DELETE).

Заголовки и тело ответа сервера принципиально не отличаются от заголовков и теза запроса. Важным отличием является код ответа. Код представляет собой трехзначное число. Различают пять групп кодов. Коды, начинающиеся на 1, сообщают о том что обработка запроса еще не завершена. Коды, начинающиеся с 2, представляют нормальное завершение запроса. Коды, начинающиеся с 3, представляют ответ сервера при перенаправлении на другой ресурс. Коды, начинающиеся на 4, представляют сообщения об ошибках в запросе (синтаксических или семантических). Наконец, коды, начинающиеся на 5, представляют ошибки, произошедшие на сервере в процессе обработки запроса. Соответственно, запрос, успешно завершивший свое выполнение, обычно возвращает один из следующих кодов – 200 (OK), 201 (создано) или 202 (принято).

Перечень кодов статусов http запросов можно найти, в частности, по ссылке https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP

Cookie.

Cookie (“куки”, буквально “печеньки”) – именованные фрагменты данных, которые сервер может хранить на клиентской машине и по мере необходимости перечитывать. Они специфичны для пользователя и для сайта и могут использоваться для решения таких задач, как запоминание предпочтений пользователя, информации о том, что пользователь залогинен и других задач. Куки считаются небезопасными и могут блокироваться политиками безопасности клиентского браузера, в этом случае их использование становится невозможным. Однако для нечувствительной информации использование кук существенно упрощает работу с клиентом.

Следующий демонстрационный фрагмент кода (с использованием библиотеки OkHttpClient) позволяет читать куки, которые присылает сервер:

```
public class Main {
        public static void main(String... args) throws IOException {
                class MyCookieJar implements CookieJar {
                        private List<Cookie> cookies;
                        public List<Cookie> getCookies() {
                                return cookies;
                        }

                        @Override
                        public void saveFromResponse(HttpUrl url, List<Cookie> cookies) {
                                this.cookies = cookies;
                        }

                        @Override
                        public List<Cookie> loadForRequest(HttpUrl url) {
                                if (cookies != null)
                                        return cookies;
                                return new ArrayList<>();
                        }
```

```
            }

            MyCookieJar cookieJar = new MyCookieJar();
            OkHttpClient.Builder builder = new OkHttpClient.Builder();
            builder.cookieJar(cookieJar);
            OkHttpClient client = builder.build();
            Request request = new Request.Builder().url("некоторый URL, который сюда надо поместить").get().build();
            Response response = client.newCall(request).execute();
            System.out.println("String list of cookies size:" + cookieJar.getCookies().size());
            for (Cookie cookie : cookieJar.getCookies()) {
                System.out.println("cookie " + cookie.name() + " has value " + cookie.value());
            }
        }
    }
```

Http сессия.

Хотя правильным считается иметь сервер без состояния (причины чего будут рассмотрены позднее), иногда клиенту необходимо иметь данные, которые сохраняются между запросами. Эта задача реализуется с помощью сессии, представляющей собой объект, содержащий данные, которые должны быть сохранены между запросами. Для того, чтобы каждый следующий запрос знал к какой сессии он принадлежит, сессии сопоставляется идентификатор и возникает задача его передачи с сервера на клиент и обратно, которая, в свою очередь, имеет две основных решения –через Cookie и через URL Rewriting.

В случае, если сессия реализуется через Cookie, то созданный на сервере объект ставится в соответствие некоторому идентификатору, который передается на клиент через Cookie. В дальнейшем клиент, чтобы обратиться к этой сессии, посылает идентификатор сессии обратно. Однако если клиентская система не разрешает Cookie, то используется URL Rewriting, то есть идентификатор сессии автоматически добавляется к каждому URL. Специальный механизм – HttpSession API – реализует эти сценарии автоматически и не требует от пользователя каких-либо специальных действий.

Структура класса сервлета.

Сервлет – класс, представляющий http-ресурс. Он доступен по определенному адресу, который в примере определяется верез аннотации. Необходимо иметь в виду, что все запросы по этому адресу обрабатываются одним и тем же экземпляром класса сервлета, поэтому он не должен использовать потоконебезопасные ресурсы. Базовая структура сервлета следующая:

```
@WebServlet(name = "<имя сервлета>", urlPatterns = {"<url по которому доступен
сервлет>"})
public class ExampleServlet extends HttpServlet {
        @Override
        protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
                //код, обрабатывающий HTTP метод GET
        }

        @Override
        protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
                //код, обрабатывающий HTTP метод POST
        }
}
```

В данном примере реализован сервлет, реализующий наиболее часто встречающиеся http методы GET и POST, но можно реализовывать и другие методы http, используя соответствующие члены класса – doPut, doDelete, doHead, doOptions, doTrace. Также можно переопределить обобщенный метод service, в котором можно самостоятельно определить использованный http метод и его обработать.

Типовая структура кода обрабатывающего метода следующая – он берет параметры из объекта request и формирует данные объекта response (ответа сервера). Для формирования тела ответа сервера у объекта response вызывается метод getWriter или getOutputStream в зависимости от того, является ответ текстовым или бинарным. Особенность этого метода заключается в том, что вызвать его можно только один раз, поэтому если какой-либо фильтр

предполагает менять ответ сервлета на лету, он должен передать ему какой-то другой поток вывода, а потом самостоятельно заполнить оригинальный поток вывода сервера.

Структура класса фильтра.

Фильтр – класс, содержащий код, выполняемый до и после вызова сервлетов (и некоторых других фильтров). В зависимости от положения фильтра в цепочке фильтров до и после него могут выполняться другие фильтры. Типовое применения фильтра – аудит безопасности, логирование либо замена ответа сервлета на лету.

Структура класса следующая:

```
@WebFilter("/example")
public class ExampleFilter implements Filter {
        public void init(FilterConfig filterConfig) throws ServletException  {  }
        public void destroy()  {  }

        @Override
        public void doFilter( ServletRequest request,  ServletResponse response,  FilterChain
        chain) throws IOException, ServletException {
                // Сюда вставляется пользовательский код, выполняемый перед
                остальными фильтрами и сервлетом
                chain.doFilter(request, newResponse);
                // Сюда вставляется пользовательский код, выполняемый после
                остальных фильтров и сервлета
        }
}
```

В данном примере реализован фильтр, который не делает ничего кроме передачи управления далее по цепочке фильтров.

Задание на лабораторную работу.

Выполните следующие задачи.

1 В соответствии со своим вариантом разработайте набор экранных форм приложения (порядка 5)

2 Соберите проект веб-приложения (war) на Maven (можно без использования Spring)

3 реализуйте формы средствами сервлетов. Проект должен как минимум содержать формы просмотра, добавления и удаления данных.

4 Аргументируйте почему были выбраны те или иные запросы HTTP.

5 Использовать базу данных можно, но не обязательно

Лабораторная работа №2. Разработка ресурса REST/JSON сервиса

Предпосылки появления Web API

При разработке приложений крайне неудобно возвращать всегда непосредственно HTML страницы. Это связано с тем, что таким образом мы объединяем логику доступа к данным и логику представлений в одном месте. Недостатком такого объединения может быть, например, то, что при появлении нового клиента доступ к данным необходимо реализовывать и в нем. Эта проблема может быть решена путем вынесения логики доступа в отдельную библиотеку, однако возникающие сложности по управлению конфигурациями всех программных продуктов являются причиной появления систем, в которых возможно разделение кода на отдельные компоненты, исполняемые на различных машинах, и обмен данными по четко специфицированным протоколам. Первоначальное развитие получили технологии, основанные на бинарных протоколах, такие как CORBA/IIOP и DCOM/ORPC, через которые мультимашинные приложения могли обмениваться информацией. В частности, технология CORBA в своей основе содержит ORB (Object Request Broker) – унифицированную шину данных, к которой можно подключиться и через которую передаются и получаются данные. Компонент, желающий выполнять передачу по шине, должен знать контекст адресата своего сообщения и выполнять конвертацию вызовов и данных в бинарный формат (процесс называется маршаллингом) и из бинарного формата (демаршаллинг) в соответствии с правилами, определенными для языка, на котором этот компонент написан. В результате с помощью технологии CORBA можно интегрировать коды на десятках различных языков.

Тем не менее такие подходы оказались неудобными (прежде всего из-за высокой сложности) и, как следствие, появилось понятие Web API - интерфейса программирования, предоставляемого через WEB. Обычно такой API получает данные и возвращает их в виде одного из стандартных форматов, наиболее популярными из которых являются XML и JSON (Java Script Object Notation).

SOAP

SOAP является технологией создания интерфейса приложения, основанной на XML. В основе подхода лежит идея о том, что веб-компонент публикует описание своего интерфейса на

специальном языке WSDL (т.е. XML с определенной схемой), а все его клиенты могут этот дескриптор прочитать и по нему определить правила вызова интерфейсных методов. Полностью этот язык описывается в документе https://www.w3.org/TR/wsdl/ . Пример дескриптора можно посмотреть здесь: https://www.w3.org/2001/04/wsws-proceedings/uche/wsdl.html

Получив ссылку на интерфейс, клиент SOAP-приложения может осуществлять вызов. При использовании SOAP все данные и вызовы преобразуются в специальные XML-посылки, называемые SOAP-конвертами (envelop), которые передаются между клиентом и сервером обычно через протокол HTTP или HTTPS.

JAVA-реализация SOAP называется JAX-WS. В виду того, что создавать дескрипторы непосредственно руками крайне неудобно, обычно разработка программного обеспечения на основе JAX-WS заключается в использовании разнообразных мастеров, автоматически генерирующих и обновляющих дескрипторы. Это делает код, связанный с обработкой дескрипторов, тяжело читаемым разработчиком. Еще одним важным недостатком SOAP является использование такой тяжеловесной технологии, как XML – маршалинг и демаршалинг для XML являтся очень ресурсоемкой операцией даже с использованием легковесных парсеров типа SAX. Поэтому при прочих равных условиях SOAP проигрывает в производительности более легким решениям.

JSON

В настоящее время лидирующим решением среди технологий организации интерфейсов является JSON – Java Script Object Notation. В этой технологии на клиент передается сериализованный в текст объект Java Script. При этом, хотя технически возможно передать объект вместе с кодом, обычно рассматриваются только поля данных.

Для того, чтобы получить представление о том, как работает JSON, рассмотрим пример Java класса и соответствующий ему JSON:

Простой Java класс (POJO, Plain Old Java Object)

```
@Data
public class CustomData {
```

```
        String name;
        Map<String, String> attributes;
        Set<String> array;
    }
```

И возможный JSON:

{"name": "Some Name", "attributes": { "someAttribute": "some value", "anotherAttribute ": "another value" }, "array": ["value1", "value2", "value2"]}

Как можно видеть, конечная структура JSON зависит от данных. В частности, если бы элементы массива или отображения имели сложную структуру (были массивами или объектами), то они также были бы преобразованы в JSON. Преобразование POJO в JSON и обратно происходит автоматически такими библиотеками, как Jackson (считается наиболее производительной) или Gson; однако очевидно, что обратное преобразование, т.е. из JSON в POJO не может быть выполнено единообразно (например, когда мы преобразуем поле attributes в примере выше мы не можем сказать, является результат типом Map или объектом с двумя полями, а array может быть преобразован как в Set, так и в массив). Поэтому для того, чтобы выполнить обратное преобразование, конвертерам требуется прототип того POJO объекта, в который мы собераемся демаршалить JSON.

Тем не менее, данный метод организации интерфейса имеет существенное преимущество – JavaScript может быть использован как на сервере, так и на клиенте, поэтому API, построенный на JSON, может быть напрямую использован клиентскими сценариями в браузере. Библиотеки AJAX (Jquery, AngularJS) могут получать результаты вызовов и анализировать их структуру естественным для языка браузерных сценариев образом.

RESTful API

Необходимо четко понимать, что не любой API, основанный на JSON, является RESTful. REST – это концептуальный подход; технически те же самые задачи могут быть решены JSON сервисами, которые не соблюдают некоторые или все концепции REST. В основе REST лежит ряд концепций, которым необходимо удовлетворять, а именно:

1. Модель является клиент-серверной, т.е. сервер предоставляет REST интерфейс, через который клиент осуществляет вызов (по протоколу HTTP/HTTPS);
2. Сервер не имеет информации о состоянии клиента (отсутствует сессия) – это значительно упрощает масштабирование (при этом сервер может иметь внутреннее, не публичное, состояние, например, какие-либо кеши данных);
3. Сервер может осуществлять кеширование данных (строго в соответствии с таблицей 1);
4. Клиентской системе не известно что именно находится а URL – один сервер или несколько серверов, возможно, обьединенных в иерархии;
5. Клиент манипулирует ресурсом через его представление.

Следующие ограничения при создании REST-интерфейсов зачастую игнорируются:

6. Каждое сообщение должно содержать достаточно информации о том как его обрабатывать (например, иметь заголовки, указывающие на допустимые операции и т.п.)
7. Сообщение должно содержать в себе свой описатель (в самом простом случае – ссылку, по которой доступен тот ресурс, который находится в этом сообщении). Это требование обозначается аббревиатурой HATEOAS.
8. Сервер может возвращать не только данные, но и код. Это последнее требование является опциональным.

Рассмотрим пример организации REST – интерфейса. Пусть имеется множество объектов *CustomData* из примера выше. Мы можем разместить их по адресу http://<server>/app/customdata/ . Тогда объект с конкретным именем может иметь URL http://<server>/app/customdata/<имя> или http://<server>/app/customdata/name/<имя>. Соответственно, выполнение метода GET с указанным URL будет приводить к чтению этого объекта, POST и PUT будут использоваться для создания и изменения объекта, а DELETE – его удаления. Начинающие разработчики испытывают желание решать эту задачу иначе, например, удалять объект через URL вида http://<server>/app/customdata/<имя>/delete , что

является технически возможным, но нарушающим правила REST, так как этот URL не является идентификатором ресурса.

Инструменты разработчика

Вероятно, наиболее популярным средством работы с запросами REST является postman https://www.getpostman.com/ . Это приложение позволяет, в частности, выполнять запросы с указанием тела запроса, метода и заголовков и получать назад ответы в разобранном виде.

Реализация REST средствами Spring-boot

В составе библиотеки spring-boot-starter-web находится мощное средство для организации REST. Для того, чтобы объявить бин, реализующий рест интерфейс, используется аннотация @RestController. Далее этому контроллеру можно указать базовый адрес, относительно которого будут строиться адреса его методов; для этого используется аннотация @RequestMapping(<адрес>). Сконфигурированный таким образом контроллер может декларировать методы, отображающиеся в рест – интерфейс. Для этого используются следующие основные аннотации (на методах):

@RequestMapping(value = "<адрес>", method =<метод>), где метод принимает значение константы класса RequestMethod – объявляет адрес и http метод доступа к REST – ресурсу;

@RequestParam("<имя параметра>") позволяет аннотировать параметр метода, после чего в него будет автоматически передан соответствующий параметр формы;

@RequestBody позволяет аннотировать параметр метода, после чего в него будет автоматически передан JSON (если мы используем его, а не форму, для передачи параметров). В этом случае Объект будет построен из JSON стандартным мапером, обычно Jackson.

Рассмотрим некоторые примеры. Следующий контроллер опрееляет рест-интерфейс

```
@RestController
@RequestMapping("/public/rest/resource")
public class SampleRestController {
```

```
        @RequestMapping(value = "/resource1", method = RequestMethod.GET)
        public Resource1 getResource1() { /*код*/  }
}
```

Определяет сервис /public/rest/resource/resource1 доступный методом GET без параметра. Если мы хотим добавить параметр то мы можем действовать двумя способами – передавать его в списке параметров URL /public/rest/resource/resource1?id=1 или через основную часть URL /public/rest/resource/resource1/1 . В первом случае (который должен использоваться только для http метода GET) определение будет выглядеть как

```
@RestController
@RequestMapping("/public/rest/resource")
public class SampleRestController {
        @RequestMapping(value = "/resource1/{id}", method = RequestMethod.GET)
        public Resource1 getResource1(@RequestParam("id" Long id)) { /*код*/  }
}
```

Во втором случае определение будет выглядеть так:

```
@RestController
@RequestMapping("/public/rest/resource")
public class SampleRestController {
        @RequestMapping(value = "/resource1/{id}", method = RequestMethod.GET)
        public Resource1 getResource1(@RequestParam("id" Long id)) { /*код*/  }
}
```

Заметим, что возвращаемый объект типа Resource1 будет преобразован в JSON автоматически, с возвращением кода 200. Если нам необходимо вернуть не только данные, но еще заголовки и код статуса операции, можно воспользоваться расширенным определением:

public ResponseEntity< Resource1> getResource1(**@RequestParam**("id" Long id)) { /*код*/ }

В такой версии определения метода можно вернуть данные вместе с заголовками и кодом; например, следующим образом:

return ResponseEntity.status(HttpStatus.CREATED).body(<объект типа Resource1>);

В качестве кода возврата будет использован HttpStatus.CREATED , т.е. 201. Отметим, что с точки зрения семантики REST использование этого кода должно идти совместно с RequestMethod.POST.

Развитие REST – Query Data API

Существуют подходы, основанные REST, и предназначенные для получения данных от сервера по определенным запросам, сформулированным клиентом. Прежде всего, это GraphQL https://graphql.org/learn/ . В технологии GraphQL на сервер посылается JSON с запросом в определенном формате, а назад приходит ответ, опять таки, в определенном формате. Этот подход, с одной стороны, гибче REST и позволяет экономить на передаче информации, которая не требуется клиенту, с другой стороны, перекладывает часть логики приложения на клиент, что является определенным недостатком.

Другим развивающимся подходом является OData https://www.odata.org/ . Подход представляет собой формальный язык запросов к сервису, выполненный в стандарте REST и обладающий высокой гибкостью.

ORDS (Oracle Rest Data Service) https://www.oracle.com/database/technologies/appdev/rest.html является инструментом, автоматически отображающим определенным образом сформированные http –запросы на базы данных.

Сравнительный анализ этих технологий можно прочитать по ссылке https://www.progress.com/blogs/rest-api-industry-debate-odata-vs-graphql-vs-ords (англ.).

Задание на лабораторную работу.

1. Подключите к проекту Maven фреймворк Spring (spring-boot)
2. Определите перечень Rest-сервисов, выполняющих те же действия, что и в лабораторной работе 1. Внимательно отнеситесь к вопросу какой HTTP метод использует тот или иной сервис и какие коды HTTP он может возвращать. Реализуйте эти сервисы.

Лабораторная работа №3. Разработка простого AJAX приложения Spring

Архитектура клиентского представления.

При программировании для сети Интернет приложение приобретает другую структуру, нежели при программировании для десктопов. Само приложение состоит из двух частей – серверной части, или бэкенда (backend), исполняемого где-то в интернете и клиентской части или фронтенда (frontend), исполняемого на клиентской машине в браузере. При этом фронтенд генерируется опять-таки на сервере и запрашивает с сервера ресурсы, необходимые для своей работы. Одновременно браузер скрывает от фронтенда технические детали, связанные с доступом к серверной части, а также следит за безопасностью.

Браузер обычно выполняет код двух видов – описания страниц на HTML и сценариев на JavaScript. Первые приложения таких типов имели статические клиентские страницы – при работе с ними было необходимо отправить на сервер запрос и получить в качестве ответа новую страницу. Такой подход иногда используется и сегодня (обычно для простых приложений с “легкими” страницами). Тем не менее он неудобен тем, что пользователь, нажав на кнопку, вынужден ждать полной регенерации своего представления; вместо этого пользователю было бы удобно чтобы страница реагировала только в той ее части, которая является релевантной воздействию. Есть и другие недостатки, в частности, затруднено масштабирование, “тяжелые” страницы интенсивно потребляют сетевые ресурсы и т.п.

Для решения указанной выше проблемы разрабатывались различнве подходы – это и загрузка скриптов в скрытые фреймы, XML острова данных, интеграция браузерных расширений (Flash, Silverlight, Java plugin) и пр. В настоящее время со введением стандарта HTML5 доминирующим решением является технология AJAX.

Технология Ajax.

Технология AJAX, или “Asynchronous JavaScript and XML” была предложена в середине двухтысячных годов как обобщение всех имеющихся на тот момент технологий асинхронного

доступа к ресурсам сервера. Ключевой идеей технологии было использование в браузере интерфейса XHR (XMLHttpRequest) для получения ресурсов сервера, которые могут предоставляться в дружественном браузеру формате (JSON, JavaScript); в результате сценарии браузера, написанные на JavaScript, могут обратиться к серверу, получить обьекты в формате JSON и динамически встроить результат в текущий документ браузера, используя DOM (Document Object Model). Другим вариантом действий является непосредственная доставка с сервера исполняемого кода на JavaScript.

В настоящее время имеется множество популярных библиотек, обрабатывающих пользовательский интерфейс средствами AJAX, а именно JQuery, Angular JS, React JS и другие.

Библиотека AngularJS.

AngularJS https://angularjs.org/ является библиотекой java script, использующей DOM для манипуляции документом на основании дополнительной разметки. Библиотека использует паттерн MVC, где в роли контроллера выступают коды на джаваскрипте, а в роли модели – элементы DOM.

Приложение AngularJS основывается на трех основных директивах. Директива **ng-app** определяет связь приложения с html страницей, директива **ng-model** связывает данные приложения с управляющими элементами на форме (изменение элемента приводит к изменению данных модели) и директива **ng-bind** связывает данные приложения с разметкой HTML (изменение модели приводит к изменению элемента). Последняя директива обычно записывается сокращенно- в виде двух фигурных скобок. Angular поддерживает использование для значений директивы **ng-bind** выражений на языке javascript. Кроме указанных, очень важной директивой является **ng-repeat**, которая используется для построения списков и таблиц.

Рассмотрим тривиальный пример, который использует angularJS для демонстрации своих базовых возможностей.

```
<!doctype html>
<html ng-app>
    <head>
        <script
        src="https://ajax.googleapis.com/ajax/libs/angularjs/1.
        7.5/angular.min.js"> </script>
    </head>
    <body>
        <div>
            <label>Name:</label>
            <input type="text" ng-model="yourName"
            placeholder="Enter a name here">
            <hr/>
            <h1>Hello {{yourName}}!</h1>
        </div>
    </body>
</html>
```

Страница помечена атрибутом ng-app, что информирует angular о необходимости ее обработки. Поле ввода помечено атрибутом ng-model, что информирует angular о необходимости установить листенер на изменение этого поля и запись его значения в переменную yourName. Наконец, чуть ниже в коде встречается инструкция {{yourName}}, что информирует angularJS о необходимости записать туда значение соответствующей переменной, которая, как мы помним, связана с полем ввода. В результате при изменении значения в поле ввода автоматически меняется текст в заголове <h1>.

Важное значение в AngularJS имеют объекты **$scope** и **$http**. Директива ng-controller определяет набор функций, которые отвечают за работу определенной формы, и **$scope** передается в каждую из них как параметр, указывающий на приложение, работающее с этим контроллером. Соответственно, в **$scope** можно определять глобальные переменные и т.п. Объект **$http** используется для целей получения REST ресурсов. Типовой вызов **$http** выглядит следующим образом (в данном случае для метода post):

```
$http.post(address).then($scope.successCallback, $scope.errorCallback);
```

В зависимости от того, увенчался метод успехом или нет, будет вызван либо метод $scope.successCallback либо метод $scope.errorCallback, в которые будет передан объект, представляющий ответ. Соответственно, именно в них должна находиться вся обработка, связанная с AJAX. В случае успешного завершения запроса в поле **data** обьекта, представляющего ответ, будет передан JSON, то есть объект Java Script, который может быть разобран, связан с переменными модели и, таким образом, отображен на форме.

Задание на лабораторную работу.

1. Создайте один или несколько форм и контроллеров для страниц приложения. Подключите библиотеку angularJS (желающие могут воспользоваться JQuery или иным аналогом).
2. Реализуйте задачи из лабораторной работы 1 средствами AJAX и REST-сервисов, разработанных в предыдущей лабораторной работе.

Лабораторная работа №4. Разработка формы логина

Основы авторизации и аутентификации. Виды аутентификации.

Несмотря на то, что существует определенная путаница в этих понятиях, авторизация и аутентификация – два разных процесса. Аутентификация – процесс определения того что пользователь является именно тем, кем он представляется. Авторизация – процесс определения того какие права имеет аутентифицированный пользователь.

Существует огромное количество моделей аутентификации, в частности,

1. HTTP BASIC
2. HTTP Digest
3. HTTP X.509 client certificate exchange (an IETF RFC-based standard)
4. LDAP
5. Kerberos
6. SSO
7. OpenID
8. OAuth
9. Form-based authentication

И многие другие. Ниже эти модели будут рассмотрены более подробно.

Виды атак на веб-приложение.

Веб-приложение обычно является приложением, распространяемым через незащищенную среду (интернет). Даже в случае внутренней сети предприятия или защищенных специализированных криптосетей всегла есть вероятность того, что среда передачи скомпрометирована. Поэтому, разрабатывая веб-приложение, необходимо уделить внимание следующим аспектам:

1. Защита данных при передаче (шифрование), ведущая к исключению атаки проксированием или путем перехвата пакетов данных сниФферами;
2. Защита от кросс-сайтовой подделки запроса и межсайтового скриптования;

3. Аутентификация стандартными средствами и алгоритмами, для которых подтверждена безопасность;
4. Разбиение ресурсов приложения по зонам доступа, присвоение им требований ролей и назначение ролей пользователям.

Снифферы и атаки “Man-in-the-middle”.

Сниффер (анализатор траффика) – это программа, которая переводит сетевую карту в режим, в котором она анализирует все пакеты в своем сегменте сети, безотносительно их адресата. Такое программное обеспечение может применяться администраторами сетей для обнаружения неправильной маршрутизации трафика; к сожалению, оно же может применяться злоумышленниками для перехвата трафика пользователя. Существуют инструменты обнаружения снифферов, но в тот момент, когда системный администратор обнаружит вредоносное ПО, захват чужих данных уже произойдет. Поэтому критически важно, чтобы весь трафик, содержащий логины и пароли, был зашифрован.

Другой важной проблемой является злонамеренная конфигурация прокси сервера. В случае,е сли доступ к нему получил злоумышленник, то весь трафик пользователя автоматически становится ему доступен. Такая атака называется “человек посередине” или man-in-the-middle. К счастью, современные алгоритмы шифрования устойчивы к таким атакам, что не в последнюю очередь достигается тем, что трафик подписывается цифровой подписью, удостоверенной сертификатом, заверенным сертифицированным удостоверяющим центром. Преодоление такой защиты крайне сложная задача, включающая подмену служб DNS (чтобы она указывала на фальшивый удостоверяющий центр) или подмену предымпортированных в систему сертификатов, что, вообще говоря, практически невозможно без получения физического доступа к системе или сети. К сожалению, получение сертификата безопасности стоит заметных денег, поэтому в рамках лабораторной работы мы будем использовать “самоподписанные” сертификаты, генерируемые программной keytool, входящей в состав платформы java. Такие сертификаты не могут быть проверены удостоверяющим центром и потому могут быть использованы только в целях разработки ПО. Например, такой сертификат может быть сгенерирован плагином maven:

```
<plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>keytool-maven-plugin</artifactId>
        <version>1.5</version>
        <executions>
        <execution>
                <goals>
                <goal>generateKeyPair</goal>
                </goals>
                <phase>generate-resources</phase>
        </execution>
        </executions>
        <configuration>
        <alias>tomcat</alias>
        <storetype>PKCS12</storetype>
        <keyalg>RSA</keyalg>
        <keysize>2048</keysize>
        <keystore>${project.basedir}/target/keystore.p12</keystore>
        <dname>CN=stepanoff.info, OU=stepanoff.info, O=stepanoff.info, L=SPB, ST=SPB,
        C=NO</dname>
        <validity>3650</validity>
        <password>mypassword</password>
        <storepass>mypassword</storepass>
        <workingDirectory>${basedir}</workingDirectory>
        <storetype>pkcs12</storetype>
        </configuration>
</plugin>
```

При попытке перехода на сайт, защищенный таким сертификатом, браузер будет выдавать предупреждение о невозможности его проверки. Так как мы знаем, что сами сгенерировали этот сертификат, мы можем считать переход на сайт **localhost** безопасным и пропустить предупреждение браузера; в большинстве случаев, встретив аналогичное сообщение в сети интернет, можно обоснованно предполагать, что сайт был взломан.

Защита от атак XSS и CSRF.

XSS – Cross-Site Scripting, специальный вид атаки, при котором вредоносный код пытается загрузить в себя часть доверенного сайта и выполнить в нем действия от имени пользователя. Существует много различных вариантов этой атаки, начиная с загрузки доверенного сайта во фреймы и заканчивая попытками инъекции сценариев в поля ввода доверенного сайта, в надежде что эти сценарии будут выполнены и сумеют получить доверенный доступ к сценариям самого доверенного сайта.

Загрузка доверенного сайта полностью или частями во вредоносный сайт в основном блокируется современными браузерами на основании политики same-of-origin (единого происхождения), то есть требования, чтобы все ресурсы сайта приходили с одного доменного имени. Иногда доверенный сайт использует технологию CORS (Cross-Origin Resource Sharing) позволяющую информировать браузер об исключениях в правиле same-of-origin. Использование этой технологии не рекомендуется с точки зрения безопасности; неправильное же ее употребление может привести к уязвимости сайта.

Инъекция кода javascript опасна тем, что злонамеренный код выполняется в контексте того же сайта, что и выываемые им доверенные сценарии. Ключевым средством борьбы с инъекцией скриптов является правильная обработка полей ввода и преобразование всех специальных символов в их символьные эквиваленты (< в &lt, > в &gt и тд). Подобные преобразования делают код неисполняемым. Тем не менее, все равно не рекомендуется вставлять потенциально небезопасные данные в страницу на сервере.

Еще одним типом атаки является CSRF – Cross-Site Request Forgery (межсайтовая подделка запроса). Эта атака происходит при попытке отправить форму в доверенный сайт, который может посчитать, что пользователь имеет cookie и потому залогинен, а следовательно запрос достоверен. Для борьбы с этим видом атак в Spring используется специальный токен, выдаваемый сервером, который необходимо указывать в параметрах запросов POST, PUT, PATCH и DELETE. Использование такого токена, естественно, возможно только при наличии сессии – в противном случае было бы невозможно установить, какой токен какой странице соответствует.

Хеширование паролей.

Одним из наиболее серьезных рисков при создании систем аутентификации является утечка паролей. Связано это с двумя аспектами – во-первых, злоумышленник, получивший доступ к базе, получает доступ к данным всех пользователей; во-вторых, пользователь, обладающий достаточно устойчивым паролем, имеет обыкновение использовать его на всех сайтах, соответственно, утечка пароля на одном сайте приводит к уязвимости его данных на всех остальных. Разработчики защиты могут частично побороть эту проблему за счет двухфакторной авторизации (т.е. в определенных или во всех случаях пользователь должен ввести код, пришедший ему на телефон), но двухфакторная авторизация используется далеко не всегда, а зачастую пользователи сами от нее отказываются, как от неудобной. Поэтому критически важно сделать так, чтобы доступ к базе не привел к немедленной утечке паролей.

Для того, чтобы избежать утечки, в базе хранят не пароли, а их хеши – то есть пароли, преобразованные специальной функцией (хэш-функцией) таким образом, чтобы обратное преобразование требовало огромных вычислительных ресурсов. Соответственно, для того, чтобы проверить, что пользователь ввел правильный пароль, сравниваются не сами пароли, а их хеши, и получив в свое распоряжение хэш восстановить их него пароль, не имея суперкомпьютера и большого количества времени, невозможно. Spring рекомендует использовать для хеширования алгоритм BCrypt https://en.wikipedia.org/wiki/Bcrypt

Basic и Digest аутентификация.

Basic аутентификация является простейшей формой аутентификации, не требующей никаких специальных средств. В этой модели аутентификации клиент посылает серверу запрос с заголовком Authorization , в содержимом которого присутствует слово Basic и пара login:password закодированная Base64. Заметим, что этот алгоритм является симметричным и потому абсолютно небезопасен. Сервер раскодирует пару login:password и, в зависимости от результатов проверки, авторизует или не авторизует пользователя. Другим вариантом передачи этой пары является прямое кодирование в URL, например https://login:password@адрессайта . Этот формат авторизации является устаревшим (RFC-3986).

Со своей стороны, сервер оперирует заголовками и кодом возврата. Если ресурс требует аутентификации, то сервер должен прислать статус 401 и заголовок WWW-Authenticate: Basic, в котором указывается Security Realm (абстракция, представляющая защитный механизм, которому известен этот пользователь).

Так как было бы очень неудобно каждый раз запрашивать пользователя о его логине и пароле, то браузеры их кешируют, причем конкретный алгоритм хеширования зависит от браузера. Это ставит проблему разлогинивания, которая решается различными техниками, связанными со сбросом закешированных данных пользователя.

Алгоритм Basic аутентификации считается слабозащищенным из-за передачи по сети пары логин : пароль в незашифрованном виде. Поэтому, как компромиссное решение, появилась аутентификация Digest (RFC-2069, RFC-2617), которая передает строку логин : realm : пароль, хешированную алгоритмом md5. Так как разобрать хэш-- функцию достаточно сложная вычислительная задача, Digest аутентификация сравнительно значительно надежнее, чем Basic, однако по прежнему уязвима к атакам типа “man-in-the-middle”.

Аутентификация через директории пользователей (Kerberos, LDAP).

В сложных корпоративных системах зачастую возникает проблема связанная с тем, что одному и тому же пользователю надо назначать права во многих приложениях

одновременно. В такой ситуации бывает полезно вынести центр аутентификации из приложения в стороннюю систему. Примерами таких решений могут быть Kerberos и LDAP (а также многие другие).

Kerberos является протоколом шифрования, при котором аутентификация возможна даже если между приложением и центром аутентификации лежит незащищенный канал. Такой протокол шифрования, например, используется службой Active Directory. Приложение и центр авторизации обмениваются данными о пользователе по специальному протоколу, используя симметричное шифрование Нидхема-Шредера.

LDAP (Light-Weight Directory Access Protocol, легковесный протокол каталога пользователей) является протоколом, часто используемым для централизованного хранения имен пользователей и паролей. Этот протокол не просто разрешает аутентифицировать пользователей, но еще и управлять ими, то есть добавлять, удалять и модифицировать записи.

Использование аутентификации через директории пользователей удобно в крупных компаниях, в которых необходимо иметь единое место ведения информации о пользователях. Кроме того, эта модель аутентификации значительно безопаснее, чем Basic и Digest, по многим причинам: сервер является внешним для приложения и может быть закрыт файрволлом, его безопасность подтверждена аналитиками, правила его безопасного использования четко определены и т.п.

Внешняя аутентификация через клиент (OpenID, OAuth).

В ряде случаев компании может быть неудобно поддерживать собственный сервер с аутентификационной информацией пользователей. Одной из причин этого является отсутствие специализированных персональных данных, отличающихся от других сайтов, а также то, что новому пользователю может быть неудобно проходить процесс регистрации на десятках сайтов. Так появились клиентские сервисы аутентификации, которые работают в тот момент, когда пользователь наимает на кнопку “авторизоваться через вконтакте”.

Вероятно, первым подобным стандартом был OpenID https://openid.net/what-is-openid/ . Этот стандарт предполагает, что каждый пользователь имеет уникальный FQDN, например,

https://pavel.stepanoff.info который можно использовать для аутентификации через сайт, который его выдал (т.е. в данном случае через сайт http://stepanoff.info), который выступает в качестве центра управления безопасностью. Таким образом, не нужно вести имя пользователя и пароль на сайте, поддерживающем аутентификацию через OpenID, достаточно иметь на нем профиль и авторизоваться через сайт, выдавший этот OpenID.

Более современным решением являются OAuth и OAuth2– стандарты аутентификации, определенные в RFC-6749 https://tools.ietf.org/html/rfc6749 . Ключевая разница со стандартом OpenID заключается в том, что OpenID центр аутентификации подтверждает личность пользователя, в то время как для OAuth/OAuth2 центр авторизации является также поставщиком API, через которое при наличии подтвержденных полномочий можно получать некоторую пользовательскую информацию. По этой схеме, в частности, работает аутентификация через социальные сети – авторизовавшись, пользователь разрешает сторонним приложениям читать его профиль. Появившись как развитие OpenID для некоторых специальных случаев, стандарты быстро завоевали широкую популярность (в связи с развитием социальных сетей).

Проводя сравнение аутентификации через клиент и аутентификации через сервер, можно отметить, что первая является в первую очередь только средством аутентификации, но не авторизации. Используя директории пользователей можно назначать полномочия пользователям, здесь же в качестве основной задачи ставится удостоверение факта, что пользователь – это тот, за кого он себя выдает; принципиально нет ограничений кто именно может создать свой профиль на сайте, используя OAuth2 и невозможно назначить пользователю полномочия раньше, чем этот профиль будет создан.

SSO аутентификация.

С развитием облачных технологий, особенно SaaS, возникла новая проблема – организации может быть необходимо иметь единый список пользователей, которым могут назначаться права в сторонних приложениях. Например, пользователь может работать с системами документооборота, управления предприятием и другими, предоставляемыми в рамках SaaS, однако было бы крайне неудобно управлять полномочиями в каждом приложении отдельно.

Эта задача решается с помощью аутентификации SSO – Single Sign On. В этой модели аутентификации существует некоторый провайдер, удостоверяющий личность и полномочия пользователей, интегрированный с используемыми приложениями таким образом, что, удостоверив в нем свои полномочия, пользователь получает доступ ко всем приложениям сразу. Подобные системы (Identity Management as a Service) поставляют многие компании - интеграторы.

SSO аутентификация решает сразу большое количество проблем, такие как необходимость для пользователя помнить много разных паролей, создание сертифицированной системы безопасности (так как SSO позволяет закрыть сервера файрволлом, пропускающим только аутентифицированные запросы), упрощает поддержку и обслуживание систем авторизации и аутентификации.

Реалиация аутентификации и авторизации средствами Spring.

Спринг реализует аутентификацию и авторизацию с помощью специальнного фильтра, называемого spring security. Этот фильтр внутри себя содержит свою собственную цепочку фильтров, проверяющую данные о пользователе, с которыми выполняется запрос. В случае, если они признаются недостоверными, фильтр отвергает запрос с соответствующей ошибкой (403 или 405 в зависимости от ситуации).

Для того, чтобы подключить стандартную аутентификацию и авторизацию к приложению spring-boot, на всем приложении используется аннотация **@EnableGlobalMethodSecurity(securedEnabled = true)** . После ее применения можно аннотировать отдельные методы указывая роли, которые должен иметь выполняющий их пользователь: **@Secured(<имя роли>).** Если пользователь не имеет этой роли, попытка вызова метода приведет к **AccessDeniedException**. Другим вариантом проверки полномочий является указание в параметрах метода переменной типа Principal – эта проверка является более гибкой, так как позволяет в зависимости от полномочий возвращать разные данные.

Подробно данный вопрос рассмотрен по ссылке https://spring.io/guides/topicals/spring-security-architecture/

Рассмотрим реализацию технологии form-based authorization средствами spring+thymeleaf (библиотеки рендеринга страниц на основе шаблонов). В рамках решения задачи требуется решить несколько подзадач:

1. Настроить безопасность (b security)
2. Настроить HTTPS
3. Настроить защиту CSRF
4. Настроить форму логина
5. Защитить чувствительные ресурсы

Настройка безопасности приложения осуществляется с помощью специального конфигурационного бина

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {


@Bean
@Override
public AuthenticationManager authenticationManagerBean() throws Exception {
	return super.authenticationManagerBean();
}


@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
	auth.inMemoryAuthentication()
		.withUser("guest").password("hello")
		.authorities("ROLE_USER");
}
```

```
@Override
protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/**", "/css/**", "/js/**", "/img/**",
                "/public/rest/**").permitAll()
                .and().formLogin().loginPage("/login").failureUrl("/login?error").usernamePara
                meter("login").passwordParameter("pass").permitAll()
                .and().logout().logoutUrl("/logout").logoutSuccessUrl("/login?logout").permit
                All()
                .and().exceptionHandling().accessDeniedPage("/forbidden");
}

@Autowired
public void configAuthentication(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(new
        BCryptPasswordEncoder());
}
}
```

Данный конфигурационный бин содержит несколько важных секций. Во-первых, метод authenticationManagerBean() определяет бин, который будет предоставлять сервис аутентификации. Как можно видеть, этот бин непосредственно связан с конфигурационным бином; смысл этого действия заключается в том, что настройки, определенные в конфигурационном бине, будут переданы в те места, где требуется аутентификация через автоматическое связывание.

Следующим важным методом является configure(AuthenticationManagerBuilder auth) . В этом методе можно определять способ проверки того, что пользователь принадлежит системе. В нашем случае используется “in-memory authentification”, то есть данные пользователя

закодированы непосредственно в приложении. Этот способ аутентификации можно использовать только при разработке ПО! В остальных случаях здесь должен находиться код, реализующий выбранную схему аутентификации, например, код получает пользователя из базы и проверяет его пароль. Связанный с этим метод configAuthentication(AuthenticationManagerBuilder auth) устанавливает алгоритм хеширования паролей BCrypt, и именно в этом формате пароль должен лежать в базе.

Наконец, еще один важный метод configure(HttpSecurity http) устанавливает свойства протокола, используя паттерн Builder. В нашем случае файл application.proiperties модержит следующие настройки:

server.port: 8443

server.ssl.key-store: target/keystore.p12

server.ssl.key-store-password: mypassword

server.ssl.keyStoreType: PKCS12

server.ssl.keyAlias: tomcat

что приводит к тому, что используется протокол https. Сертификат, ссылка на который идет из файла свойств, генерируется maven с помощью плагина, описанного ранее. Дальнейший код по настройке HttpSecurity указывает, какие именно страницы отвечают за логин, как называются переменные, представляющие логин и пароль, а также другие параметры. Обратите внимание, что по умолчанию CSRF (и это правильно). CSRF можно отключить с помощью команды .and().csrf().disable(), однако это сразу делает приложение незащищенным.

Обработка страницы логина в thymeleaf будет выполнена автоматически, также автоматически будет поддерживаться токен CSRF. Шаблон должен лишь содержать поля ввода, имена которых совпадают с именами, использованными при настройке HttpSecurity, а также ссылку на форму логина:

```
<div class="main-div">
    <div class="panel">
            <h2 class="form-heading">Our University Database</h2>
            <p>Please enter your login and password</p>
            <div th:if="${param.error}" class="alert alert-danger">Unknown username or password.</div>
                <div th:if="${param.logout}" class="alert alert-success">Good bye!</div>
            </div>
            <form id="Login" th:action="@{/login}" method="post">
                <div class="form-group">
                    <input type="text" class="form-control" id="login" name="login" placeholder="Login"/>
                </div>
                <div class="form-group">
                    <input type="password" class="form-control" id="pass" name="pass" placeholder="Password"/>
                </div>
                <button type="submit" class="btn btn-primary">Login</button>
            </form>
        </div>
```

Важным моментом является логаут. Для его выполнения необходимо выполнить http метод POST.

```
            <form name="logoutForm" th:action="@{/logout}" method="post">
                    <input type="submit" class="btn btn-primary" value="Sign Out"/>
            </form>
```

В составе спринг есть специальная раметка, которая позволяет понять залогинен пользователь или нет:

```
<div sec:authorize="isAnonymous()">Код рендерится если пользователь не залогинен</div>

<div sec:authorize="isAuthenticated()">Код рендерится если пользователь залогинен</div>
```

Последнее, о чем следует сказать, это связь токена CSRF с AJAX-запросами. Для того, чтобы поддержать эту функциональность нужно проинформировать AngularJS о необходимости включения заголовок CSRF во все запросы к серверу, иначе они будут отвергнуты:

```
var app = angular.module('имямодуля', []).config(function ($httpProvider) {
        csrftoken = $("meta[name='_csrf']").attr("content");
        csrfheader = $("meta[name='_csrf_header']").attr("content");
        $httpProvider.defaults.headers.common["X-CSRF-TOKEN"] = csrftoken;
});
```

Задание на лабораторную работу.

1. Переведите вебсервер на использование SSL
2. Реализуйте form-based authentication
3. Защитите ресурсы, требующие запись и введите аудит.

Pavel A. Stepanov, pavel@stepanoff.me

Лабораторная работа №5. Разработка приложения с использованием thymeleaf

Шаблоны страниц JSP.

Шаблоны страниц Spring.

Шаблоны страниц Thymeleaf.

Задание на лабораторную работу.

1 Реализуйте задачи из лабораторной работы 1 средствами Thymeleaf
2 Обратите внимание на локализацию приложения. Русский язык должен поддерживаться.

Pavel A. Stepanov, pavel@stepanoff.me

Лабораторная работа №6. Разработка приложения с использованием Hibernate

Основные понятия JDBC. TBD

JDBC является технологией, лежащей в основе более высокоуровневых средств доступа к базам данных, таких как ORM. JDBC состоит из набора стандартных классов, образующих обертку вокруг драйвера базы данных, поставляемого производителем. За счет этого обеспечивается некоторая унификация доступа к БД, хотя и в меньшей степени, нежели для ORM.

В основе работы с БД лежит класс **java.jdbc.Connection** , представляющий соединение с базой данных. Для инициализации этого класса требуется информация, специфичная для производетеля БД. Обычно при работе из среды аппсервера созданием экземпляров этого класса управляет именно контейнер, а не пользовательский код, соответственно, приложение тем или иным образом его от контейнера получает.

ORM. Виды ORM. TBD

JPA. TBD

Spring-data-jpa. TBD

Задание на лабораторную работу.

1. Добавьте базу данных, модель данных, JPA репозитории и сервисы.
2. Реализуйте операции чтения/записи из одной из предыдущих лабораторных работ (3 или 5) с помощью JPA

Pavel A. Stepanov, pavel@stepanoff.me

Лабораторная работа №7. Разработка приложения с асинхронной очередью сообщений

Концепции EDA. Читатели-писатели и производители-потребители TBD

Zookeeper. TBD

Kafka. TBD

Задание на лабораторную работу.

Целью работы является реализация простой системы распределенной репликации (“писатели-читатели”).

1. Скачайте и разверните Apache Kafka
2. Модифицируйте свое приложение со встраиваемой базой данных так, чтобы его можно было запустить в нескольких экземплярах на разных портах
3. Реализуйте в рамках своего приложения Producer и Consumer такие, что
    a. Producer при каждой операции записи оповещает соответствующий топик
    b. Consumer при получении информации из топика записывает обновление в локальную (встроенную в приложение) базу
4. Продемонстрируйте, что информация, записанная одним приложением, доступна второму приложению.

Pavel A. Stepanov, pavel@stepanoff.me

Лабораторная работа №8. Разработка микросервиса

Виртуализация и связь с автоматическим развертыванием. TBD

Docker. TBD

Mesos и его экосфера. TBD

Kubernates и его экосфера TBD.

Задание на лабораторную работу.

Подготовьте Ваше приложение к разворачиванию в облачном сервисе или компоненте Docker.

Список литературы.

Интернет-источники.

1. https://www.baeldung.com [англ.]
2.

Кафедра компьютерных технологий и программной инженерии (№43) СПбГУАП