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

pavel@stepanoff.info

Технологии разработки серверных информационных систем

Методическое пособие

ГУАП

Санкт-Петербург

-2018-

Оглавление

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.info

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

Кафедра компьютерных технологий и программной инженерии (№43) СПбГУАП

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 (изменение с течением времени гемоглобина в крови, сахара, АЛТ, АСТ и тому подобных параметров)
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.).

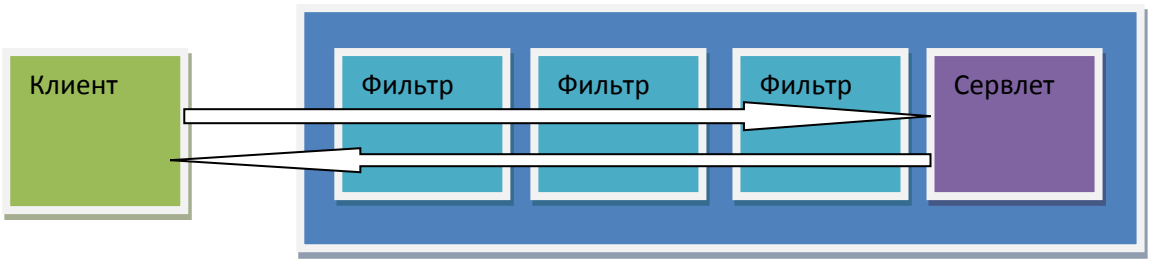
Pavel A. Stepanov, pavel@stepanoff.info

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 желвательной.

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.

Следущий 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

Pavel A. Stepanov, pavel@stepanoff.info

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 обменивающиеся 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:

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 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 некоторый формальный язык запросов к сервису, выполненный в стандарте RESTи 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

Задание на лабораторную работу.

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

Лабораторная работа №4. Разработка формы логина

Задание на лабораторную работу.

1. Добавьте в приложение поддержку одной из моделей безопасности средствами spring security (предпочтительно Digest)
2. Защитите ресурсы, требующие запись и введите аудит.

Лабораторная работа №5. Разработка приложения с использованием thymeleaf

Задание на лабораторную работу.

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

Лабораторная работа №6. Разработка приложения с использованием Hibernate

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

Лабораторная работа №7. Разработка приложения с асинхронной очередью сообщений

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.

Лабораторная работа №8. Разработка микросервиса

Задание на лабораторную работу.

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