МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Юго-Западный государственный университет»
(ЮЗГУ)

Кафедра информационных систем и технологий

УТВЕРЖДАЮ
Проректор по учебной работе
_____ О. Г. Локтионова
«____»_____ 2013 г.

Программирование REST-сервисов на языке Java

Методические указания по выполнению лабораторной работы
по курсу «Интернет-технологии» для студентов, обучающихся по
направлению подготовки 230400.62 «Информационные системы и
технологии»

Курск 2013

УДК 004.42, 004.75

Составитель М. В. Бородин

Рецензент
Доктор технических наук, профессор О. И. Атакищев

Программирование REST-сервисов на языке Java : методические указания по выполнению лабораторной работы / Юго-Зап. гос. ун-т; сост. М. В. Бородин. Курск, 2013. 18 с., библиогр.: 3.

В методических указаниях описывается технология разработки REST-сервисов на языке Java. Описание сопровождается примерами. Приводится список контрольных вопросов и вариантов заданий. Предназначены для студентов направления подготовки 230400.62

Текст печатается в авторской редакции

Подписано в печать . Формат 60 × 84 1/16.
Усл. печ. л. . Уч.-изд. л. . Тираж 100 экз. Заказ . Бесплатно.
Юго-Западный государственный университет.
305040, г. Курск, ул. 50 лет Октября, 94

1. Цель

Целью настоящей лабораторной работы является освоение студентами навыков разработки REST-сервисов на языке Java.

2. Задание

В соответствии с индивидуальным вариантом задания, полученным от преподавателя, требуется:

• определить состав и параметры ресурсов, реализуемых сервисом;

• реализовать сервис на языке программирования Java.

3. Содержание отчета

Отчет о выполнении работы должен включать в себя:

• титульный лист;

• вариант задания;

• описание ресурсов, реализуемых сервисом;

• исходный текст сервиса;

• тесты и описание процедуры тестирования.

4. Теоретические сведения

REST-сервис — это серверное программное обеспечение, использующее для обмена данными клиентом протокол HTTP. Программный интерфейс для создания REST-сервисов определен в спецификации JSR 311. Одной из ее реализаций является библиотека Jersey.

Пример простейшего сервиса:

```
import javax.ws.rs.*;
@Path("hello")
public class HelloResource {
    @GET
    @Produces("text/plain")
    public String getHelloText() {
        return "Hello, World!";
    }
    @GET
    @Produces("text/html")
    public String getHelloHtml() {
        return "<html>"
             + "<head><title>Hello</title></head>"
```

```
                + "<body>Hello, World!</body>"
                + "</html>";
    }
}
```

Если Web-сервер, предоставляющий доступ к этому сервису, запущен на компьютере с именем localhost, а сам сервис отображается на корневой каталог этого сервера, то строка Hello World доступна по адресу http://localhost/hello.

Запустить его с использованием встроенного Web-сервера можно так:

```
import com.sun.jersey.api.container.httpserver
        .HttpServerFactory;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
public class HttpServerPublisher {
    private HttpServerPublisher() {
    }
    public static void main(String[] args)
            throws IOException {
        HttpServer server = HttpServerFactory.create(
                "http://localhost/");
        server.start();
    }
}
```

Однако, вернемся к примеру. Аннотация Path определяет класс как ресурс и задает путь к нему. Не считая аннотаций, ресурсы являются обычными классами Java. Путь к ресурсу отсчитывается от корневого каталога сервера (если сервис представляет собой обычную программу, использующую встроенный Web-сервер) или же от корневого каталога Web-приложения (если сервис представляет собой Web-приложение, выполняющееся на сервере приложений). Путь, заданный аннотацией Path, может быть пустым.

Аннотация GET задает метод, вызываемый при поступлении GET-запроса. Тело отклика возвращается из метода. Аннотация GET является обязательной (иначе из чего будет понятно, что вызывать?).

Аннотация Produces задает тип данных отправляемых клиенту в теле отклике и не является обязательной. Несколько методов могут отличаться только аннотацией Produces.

По умолчанию под каждый HTTP-запрос создается новый объект ресурсного класса. Это избавляет от необходимости синхронизации доступа к полям (если они нестатические); однако, накладывает довольно жесткие требования на время инициализации и на суммарный размер полей. Если необходимо сохранять какие-либо данные между запросами, то можно воспользоваться статическими полями, или, что лучше, использовать, какой-либо другой класс.

Изменим определение метода getHelloText следующим образом:

```
@GET
@Produces("text/plain")
public String getHelloText(
        @QueryParam("name")
        @DefaultValue("World")
        String name) {
    return "Hello, " + name + "!";
}
```

Вызывать этот метод можно, например, обратившись по адресу http://localhost/hello?name=John+Smith. Аннотация QueryParam показывает, что через параметр name методу getHelloText будет передано значение параметра name из строки запроса. Аннотация DefaultValue задает значение, которое будет передаваться в метод в том случае, если в строке запроса параметра name нет.

Изменим метод getHelloText так, чтобы при формировании тела отклика учитывались предпочтения клиента (вернее пользователя), относительно языка:

```
@Context
private Request request;
@GET
@Produces("text/plain")
public String getHelloText() {
    Locale[] languages = HelloTranslator
            .getSupportedLanguages();
    List<Variant> variants = Variant
            .languages(languages).add().build();
    Variant variant = request.selectVariant(variants);
    if (null != variant) {
        return HelloTranslator.getTranslation(
                variant.getLanguage());
    } else {
```

```
            throw new WebApplicationException(
                    Response.Status.NOT_ACCEPTABLE);
        }
}
```

С помощью аннотации Context обозначаются поля, также параметры методов и конструкторов, через которые передается информация о контексте выполнения. В данном случае информация о HTTP-запросе инъецируется в переменную request (при использовании аннотации Context тип инъецируемой информации определяется типом поля или параметра). Для инъекции могут использоваться также ранее упомянутые аннотации QueryParam, HeaderParam и подобные им.

В объектах класса Variant инкапсулируется информация вариантах содержимого, включающая тип содержимого, язык и кодирование (на самом деле — сжатие). Метод selectVariant выбирает наиболее предпочтительный вариант.

Выброс исключения WebApplicationException приводит к отправке клиенту отклика с соответствующим кодом статуса.

Следующий пример представляет собой сервис, имитирующий работу с базой данных фильмов:

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
@Path("movie")
public class MovieCatalogResource {
    @GET
    @Produces("application/xml")
    public MovieList getMovies() {
        return MovieCatalog.getCatalog().getMovies();
    }
    @POST
    @Consumes("application/xml")
    public void addMovie(Movie movie) {
        MovieCatalog.getCatalog().addMovie(movie);
    }
    @Path("{id}")
    @GET
    @Produces("application/xml")
    public Movie getMovieById(
        @PathParam("id") Integer id) {
        Movie movie =
                MovieCatalog.getCatalog().getMovie(id);
        if (null != movie) {
```

```
            return movie;
        } else {
            throw new WebApplicationException(
                    Response.Status.NOT_FOUND);
        }
    }
    @Path("{id}")
    @PUT
    @Consumes("application/xml")
    public void updateMovieById(
                @PathParam("id") Integer id, Movie movie) {
        if (!MovieCatalog.getCatalog().updateMovie(
                id, movie)) {
            throw new WebApplicationException(
                    Response.Status.NOT_FOUND);
        }
    }
    @Path("{id}")
    @DELETE
    public void deleteMovieById(
            @PathParam("id") Integer id) {
        if (!MovieCatalog.getCatalog().deleteMovie(id)) {
            throw new WebApplicationException(
                    Response.Status.NOT_FOUND);
        }
    }
}
```

Методы, обрабатывающие HTTP-запросы могут возвращать объекты любого класса, аннотированного XmlRootElement. Такие объекты автоматически преобразовываются в XML-данные.

Аннотации POST, PUT и DELETE, а также HEAD и OPTIONS аналогичны по назначению аннотации GET (и также являются обязательными).

Аннотация Consumes задает тип данных, получаемых от клиента в теле запроса.

Собственно данные тела запроса передаются методу через не имеющий аннотаций параметр (методы, обрабатывающие HTTP-запросы не могут иметь более одного параметра без аннотаций). К типам параметров, через которые в методы передаются тела запросов, предъявляются те же требования, что и к типам значений возвращаемых методами, которые отправляет клиентом данные в телах откликов. Метод, обрабатывающий HTTP-запрос может и не

возвращать никакого значения; в этом случае клиенту отправляется отклик, не имеющий тела (разумеется, при обработке GET-запросов это недопустимо).

Аннотация Path может применяться не только к классам, но и к методам. В этом случае полный путь к методу получается сцеплением путей, указанных в обеих аннотациях. Путь, задаваемый аннотацией Path, может включать в себя подстановки (записываемые в фигурных скобках). По умолчанию на месте подстановки в HTTP-запросе могут быть любые символы, кроме наклонной черты. Подстановки используются совместно с аннотацией PathParam, благодаря которой часть пути, оказавшаяся на месте подстановки, передается через параметр либо инъецируется в поле. В данном случае обращение по адресу http://localhost/movies/101 приведет к вызову метода getMovieById с параметром id, равным 101.

Помимо методов, непосредственно обрабатывающих HTTP-запросы, ресурс может содержать методы, возвращающие вложенные ресурсы. Переработаем класс MovieCatalogResource так, чтобы запросы к конкретному фильму обрабатывал бы вложенный ресурс:

```
@Path("movie")
public class MovieCatalogResource {
    @GET
    @Produces("application/xml")
    public MovieList getMovies() {
        return MovieCatalog.getCatalog().getMovies();
    }
    @POST
    @Consumes("application/xml")
    public void addMovie(Movie movie) {
        MovieCatalog.getCatalog().addMovie(movie);
    }
    @Path("{id}")
    public MovieResource movieById(
            @PathParam("id") Integer id) {
        return new MovieResource(id);
    }
}
public class MovieResource {
    private Integer id;
    public MovieResourceV2(Integer id) {
        this.id = id;
```

```
    }
    @GET
    @Produces("application/xml")
    public Movie get() {
        Movie movie =
                MovieCatalog.getCatalog().getMovie(id);
        if (null != movie) {
            return movie;
        } else {
            throw new WebApplicationException(
                    Response.Status.NOT_FOUND);
        }
    }
    @PUT
    @Consumes("application/xml")
    public void update(Movie movie) {
        if (!MovieCatalog.getCatalog().updateMovie(
                id, movie)) {
            throw new WebApplicationException(
                    Response.Status.NOT_FOUND);
        }
    }
    @DELETE
    public void delete() {
        if (!MovieCatalog.getCatalog().deleteMovie(id)) {
            throw new WebApplicationException(
                    Response.Status.NOT_FOUND);
        }
    }
}
```

Отличительным признаком метода, возвращающего вложенный ресурс, является наличие аннотации Path при отсутствии аннотаций HTTP-методов (т. е. GET, POST и т. д.). Классы вложенных ресурсов, в отличие от классов обычных ресурсов, не требуют аннотации Path.

Усовершенствуем Web-сервис базы данных о фильмах:

```
@Path("movie")
public class MovieCatalogResource {
    @Context
    private UriInfo uriInfo;
    @Context
    private Request request;
    @GET
    @Produces("application/xml")
```

```
    public MovieList getMovies() {
        return MovieCatalog.getCatalog().getMovies();
    }
    @POST
    @Consumes("application/xml")
    public Response addMovie(Movie movie) {
        Integer id = MovieCatalog.getCatalog().addMovie(
                movie);
        URI uri = uriInfo.getAbsolutePathBuilder()
                .path("{id}")
                .buildFromMap(
                    Collections.singletonMap("id", id));
        return Response.created(uri).build();
    }
    @Path("{id}")
    public MovieResource movieById(
            @PathParam("id") Integer id) {
        return new MovieResource(request, id);
    }
}
```

Объект класса Response инкапсулирует всю информацию об отклике, включая, как тело отклика, так и код статуса и набор заголовков. Метод created присваивает статусу отклика значение CREATED и добавляет в него заголовок Location (из этого заголовка клиент узнает адрес созданной записи).

Для построения адреса созданной записи о фильме используется инъецированный объект типа UriInfo инкапсулирующий информацию об адресе того ресурса, запрос к которому обрабатывается в данный момент. При той части адреса, которая представляет собой пусть можно использовать подстановки, так же как и в аннотации Path.

Следующее усовершенствование — поддержка условных запросов.

```
public class MovieResource {
    private Request request;
    private Integer id;
    public MovieResource(Request request, Integer id) {
        this.request = request;
        this.id = id;
    }
    @GET
    @Produces("application/xml")
    public Response get() {
```

```
    Movie movie =
            MovieCatalog.getCatalog().getMovie(id);
    if (null != movie) {
        EntityTag movieTag = getMovieTag(movie);
        Response.ResponseBuilder responseBuilder
                = request.evaluatePreconditions(
                    movieTag);
        if (null == responseBuilder) {
            responseBuilder = Response.ok(movie)
                    .tag(movieTag);
        }
        return responseBuilder.build();
    } else {
        throw new WebApplicationException(
                Response.Status.NOT_FOUND);
    }
}
@PUT
@Consumes("application/xml")
public void update(Movie movie) {
    MovieCatalog catalog = MovieCatalog.getCatalog();
    for (;;) {
        Movie oldMovie = catalog.getMovie(id);
        if (null != oldMovie) {
            EntityTag oldMovieTag
                    = getMovieTag(oldMovie);
            Response.ResponseBuilder responseBuilder
                    = request.evaluatePreconditions(
                        oldMovieTag);
            if (null != responseBuilder) {
                throw new WebApplicationException(
                        responseBuilder.build());
            }
            if (catalog.updateMovie(
                    id, movie, oldMovie)) {
                return;
            }
        } else {
            throw new WebApplicationException(
                    Response.Status.NOT_FOUND);
        }
    }
}
@DELETE
public void delete() {
    MovieCatalog catalog = MovieCatalog.getCatalog();
    for (;;) {
```

```
            Movie oldMovie = catalog.getMovie(id);
            if (null != oldMovie) {
                EntityTag oldMovieTag = getMovieTag(
                        oldMovie);
                Response.ResponseBuilder responseBuilder
                        = request.evaluatePreconditions(
                            oldMovieTag);
                if (null != responseBuilder) {
                    throw new WebApplicationException(
                            responseBuilder.build());
                }
                if (catalog.deleteMovie(id, oldMovie)) {
                    return;
                }
            } else {
                throw new WebApplicationException(
                        Response.Status.NOT_FOUND);
            }
        }
    }
    private static EntityTag getMovieTag(Movie movie) {
        return new EntityTag(Integer.toHexString(
                movie.hashCode()));
    }
}
```

Во вложенных ресурсах инъекции не осуществляются, поэтому контекст запроса приходится передавать через параметр.

Тэг инкапсулируется объектом класса EntityTag. В данном примере тэг создается на основе хэш-кода объекта, представляющего информацию о фильме.

Для проверки того, нужно ли выполнять условный запрос используется метод evaluatePreconditions; если он возвращает построитель отклика, то выполнять не нужно.

Дополнительные параметры методов updateMovie и deleteMovie необходимы для реализации оптимистической блокировки на уровне «базы данных» (именно ради оптимистической блокировки HTTP-методы PUT и DELETE и сделаны условными).

Рассмотрим теперь пример, в котором в теле отклика должно передаваться изображение.

```
@Path("chequers")
public class ChequersResource {
    @GET
```

```
    @Produces("image/gif")
    public StreamingOutput get() {
        return new StreamingOutput() {
            public void write(OutputStream output)
                    throws IOException {
                ImageIO.write(Chequers.createImage(20, 20),
                        "gif", output);
            }
        };
    }
}
```

Используя интерфейс StreamingOutput, метод обработки запроса может не возвращать тело запроса, а записывать его в поток.

Усовершенствуем метод get, позволив клиенту самому определять наиболее подходящий для него формат изображения:

```
@Context
private Request request;
@GET
public Response get() {
    String[] mimeTypes = ImageIO.getWriterMIMETypes();
    MediaType[] mediaTypes =
            new MediaType[mimeTypes.length];
    for (int i = 0; i < mimeTypes.length; ++i) {
        mediaTypes[i] = MediaType.valueOf(mimeTypes[i]);
    }
    Variant variant = request.selectVariant(
            Variant.mediaTypes(mediaTypes).add().build());
    if (null != variant) {
        final String mimeType = variant.getMediaType()
                .toString();
        return Response.ok(new StreamingOutput() {
            public void write(OutputStream output)
                    throws IOException {
                ImageWriter writer = ImageIO
                        .getImageWritersByMIMEType(
                        mimeType)
                        .next();
                try {
                    ImageOutputStream imageOutput = ImageIO
                            .createImageOutputStream(
                            output);
                    writer.setOutput(imageOutput);
                    writer.write(
                            Chequers.createImage(20, 20));
                    imageOutput.flush();
```

```
                } finally {
                    writer.dispose();
                }
            }
        }, variant).build();
    }
    throw new WebApplicationException(
            Response.Status.NOT_ACCEPTABLE);
}
```

Поскольку тип возвращаемого содержимого зависит от запроса использовать аннотацию Produces нельзя; это приводит к необходимости явно добавлять заголовок Content-Type (вторым параметром метода ok).

Еще пример:

```
@Path("download/{path}")
public class DownloadResource {
    private File baseDirectory = new File("C:\\Temp");
    @GET
    public Response get(@PathParam("path") String path)
            throws IOException {
        File file = new File(baseDirectory, path);
        if (DownloadUtilities.isFileWithinDirectory(
                file, baseDirectory)) {
            return Response.ok(file, DownloadUtilities
                .getMediaTypeForFile(file))
                .build();
        } else {
            throw new WebApplicationException(
                    Response.Status.FORBIDDEN);
        }
    }
}
```

Метод обработки HTTP-запроса может возвращать имя локального файла. В этом случае клиенту в теле отклика передается содержимое этого файла.

Усовершенствуем и этот пример, добавив возможность получать файлы из подкаталогов

```
@Path("download/{path: .+}")
public class DownloadResourceV2 {
    private File baseDirectory = new File("C:\\Temp");
    @GET
    public Response get(
```

```
            @PathParam("path") List<String> path)
            throws IOException {
        File file = baseDirectory;
        for (String pathSegment : path) {
            file = new File(file, pathSegment);
        }
        if (DownloadUtilities.isFileWithinDirectory(
                file, baseDirectory)) {
            return Response.ok(file, DownloadUtilities
            .getMediaTypeForFile(file))
            .build();
        } else {
            throw new WebApplicationException(
                    Response.Status.FORBIDDEN);
        }
    }
}
```

Для каждой подстановки, указанной в аннотации Path, может быть указан шаблон в виде регулярного выражения.

Если типом параметра, аннотированного PathParam, QueryParam, HeaderParam, MatrixParam и FormParam, является List, Set или SortedSet, то значение, извлекаемое соответствующей аннотацией, автоматически разбивается на составные части. Способ разбиения зависит от источника значения.

Просмотреть описание REST-сервиса можно, обратившись оп адресу http://localhost/application.wadl.

Множество типов, которые можно использовать при описании тела запроса и тела отклика можно расширить. Определим класс, позволяющий методам, обрабатывающим HTTP-запросы, возвращать графические изображения.

```
@Provider
@Produces("image/*")
public class ImageBodyWriter
        implements MessageBodyWriter<RenderedImage> {
    public boolean isWriteable(Class<?> c,
            Type t, Annotation[] a, MediaType mt) {
        return RenderedImage.class.isAssignableFrom(c)
                && ImageIO.getImageWritersByMIMEType(
                    mt.toString()).hasNext();
    }
    public long getSize(RenderedImage ri, Class<?> c,
            Type t, Annotation[] a, MediaType mt) {
        return -1;
```

```
    }
    public void writeTo(RenderedImage ri, Class<?> c,
            Type t, Annotation[] a, MediaType mt,
            MultivaluedMap<String, Object> h,
            OutputStream os)
            throws IOException, WebApplicationException {
        ImageWriter iw = ImageIO.getImageWritersByMIMEType(
                mt.toString()).next();
        try {
            ImageOutputStream ios = ImageIO
                    .createImageOutputStream(os);
            iw.setOutput(ios);
            iw.write(ri);
            ios.flush();
        } finally {
            iw.dispose();
        }
    }
}
```

Аннотация Provider показывает, что данный класс предназначен для расширения возможностей среды, обеспечивающей выполнение REST-сервисов. Как конкретно класс расширяет возможности среды, определяется тем, какие интерфейсы он реализует. В ходе инициализации среда автоматически создает объекты классов аннотированных Provider.

Интерфейс MessageBodyWriter определяет методы для преобразования объекта в тело отклика. Когда среде требуется преобразовать объект в тело отклика, она обращается ко всем реализациям интерфейса MessageBodyWriter и путем вызова метода isWriteable ищет первую реализацию, которая поддерживает такое преобразование для заданного класса объекта и типа содержания. Как только реализация найдена среда использует метод getSize для получения размера тела отклика в байтах и метод writeTo для записи тела отклика.

Определим класс, ответственный за формирование отклика в том случае, если метод, обрабатывающий HTTP-запрос, выбрасывает исключение MovieNotFoundException.

```
@Provider
public class MovieNotFoundExceptionMapper
        implements ExceptionMapper<MovieNotFoundException>{
    public Response toResponse(
```

```
            MovieNotFoundException exception) {
        return Response
                .status(Response.Status.NOT_FOUND)
                .type(MediaType.TEXT_PLAIN)
                .entity("There is no such movie")
                .build();
    }
}
```

5. Контрольные вопросы

1. Что такое REST-сервис?
2. Как запустить REST-сервис?
3. Для чего используется аннотация Path?
4. Для чего используются аннотации GET и POST?
5. Каким образом в метод-обработчик передается тело запроса?
6. Каким образом из метода-обработчика возвращается тело отклика?
7. Для чего используются аннотации Produces и Consumes?
8. Как в методе-обработчике получить значения параметров запроса?
9. Для чего используется класс Variant?
10. Как вернуть отклик с сообщение об ошибке?
11. Для чего используется класс Response?
12. Для чего используется класс UriInfo?
13. Что называют подстановкой (в пути к ресурсу)?
14. Как реализовать условную обработку запросов?
15. В каких случаях следует реализовывать интерфейс StreamingOutput?
16. По какому адресу следует обратиться, чтобы просмотреть описание REST-сервиса?
17. Что нужно сделать, чтобы передать в теле запроса или отклика объект пользовательского класса?

6. Варианты заданий

1. Чтение файлов на удаленном компьютере
2. Запись файлов на удаленном компьютере
3. Просмотр дерева каталогов на удаленном компьютере
4. Рекурсивный поиск файла на удаленном компьютере
5. Выполнение команды ОС на удаленном компьютере

6. Показ на удаленном компьютере графического изображения
7. Получение снимка экрана удаленного компьютера
8. Имитация нажатия клавиши на клавиатуре удаленного компьютера
9. Игра с удаленным компьютером в крестики-нолики
10. Отгадывание «задуманного» удаленным компьютером целого числа
11. Удаленная база данных отдела кадров
12. Удаленная база данных деканата
13. Удаленная база данных склада
14. Удаленное тестирование знаний
15. Электронная почта

7. Список литературы

1. Sandoval J. RESTful Java Web Services — Packt Publishing, 2009. — 256 p.
2. Massé M. REST API Design Rulebook — O'Reilly, 2012. — 114 p.
3. Daigneau R.Service Design Patterns : Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services — Addison-Wesley, 2012 — 354 p.