Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Конспект лекции №7
Поддержка архитектуры REST в Spring.

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

Предисловие

Разработчикам часто приходится заниматься разработкой сложных приложений, решающих прикладные задачи. Данные – это всего лишь сырье, обрабатываемое приложениями. Но если спросить специалистов в той или иной предметной области, что является более ценным для них, данные или программное обеспечение, они наверняка выберут данные. Данные – это кровь деловой жизни многих компаний. Программное обеспечение часто является взаимозаменяемым. Но данные, накапливаемые в течение нескольких лет, ничем не заменишь.

Не кажется ли вам странным, что, несмотря на важность данных, мы часто разрабатываем приложения, не уделяя им должного внимания? В последние годы все большую популярность обретает альтернатива традиционным веб-службам на основе протокола SOAP – архитектура передачи представлений о состоянии (Representational State Transfer, REST). Чтобы помочь разработчикам приложений на основе фреймворка Spring воспользоваться преимуществами архитектуры REST, ее поддержка была включена в версию Spring 3.0.

Самое интересное, что поддержка REST в Spring реализована поверх Spring MVC. То есть мы уже знаем многое из того, что нам потребуется для работы с REST в Spring. В этой главе мы создадим контроллеры, обслуживающие запросы на получение ресурсов RESTful , опираясь на уже имеющиеся знания о фреймворке Spring MVC. А также посмотрим, что может предложить фреймворк Spring для взаимодействия со службами REST со стороны клиента. Но прежде чем перейти к практической стороне дела, определим, что такое REST.

1 Обзор архитектуры REST

Готов поспорить, что вы не впервые слышите или читаете об архитектуре REST. В последние годы разговоры о ней идут непрерывно, и вы наверняка не раз слышали ставшие модными в мире разработки программного обеспечения отзывы об архитектуре REST как о более привлекательной альтернативе веб-службам SOAP.

Конечно, для многих применений, где веб-службы SOAP могут оказаться слишком тяжеловесными, архитектура REST является более простой альтернативой. Но проблема в том, что не все до конца понимают, что же такое REST. В результате эта архитектура оказалась окружена массой ошибочных домыслов. Прежде чем приступать к обсуждению поддержки REST в Spring, необходимо получить общее представление об архитектуре REST и возможностях ее применения.

1.1 Основы REST

Многие ошибочно считают, что архитектура REST, являющаяся способом организации «веб-служб с адресами URL», – это еще один механизм вызова

удаленных процедур (RPC), подобно SOAP, где вызовы осуществляются посредством простых обращений по протоколу HTTP и без огромных пространств имен SOAP в XML.

В действительности архитектура REST имеет весьма мало общего с RPC. Модель RPC является ориентированной на предоставление услуг и выполнение операций, тогда как архитектура REST ориентирована на предоставление доступа к ресурсам и данным. Кроме того, хотя в архитектуре REST адреса URL играют важную роль, они – лишь часть общей мозаики. Чтобы понять суть архитектуры REST, попробуем разбить эту аббревиатуру на составные части:

- *представление* – ресурсы REST могут быть представлены практически в любой форме, включая XML, формат представления объектов JavaScript (JavaScript Object Notation, JSON) или даже HTML, какая лучше подходит для потребителя ресурсов;
- *состояние* – в архитектуре REST состояние ресурсов представляет больший интерес, чем операции, которые можно выполнить над ресурсами;
- *передача* – архитектура предусматривает организацию передачи данных ресурса в некотором представлении из одного приложения в другое. Проще говоря, архитектура REST – это комплекс решений, связанных с передачей информации о состоянии ресурса в некоторой форме от сервера клиенту (или наоборот).

Отталкиваясь от этой точки зрения на архитектуру REST, постараемся избегать таких терминов, как «служба REST» или «веб-служба RESTful» и подобных им, которые порождают неправильное представление о преобладающей роли операций. Вместо этого будем говорить о ресурсах RESTful, чтобы подчеркнуть природу REST, ориентированную на ресурсы.

1.2 Поддержка REST в Spring

Некоторые ингредиенты, необходимые для экспортирования ресурсов REST, появились в Spring достаточно давно. Но в версии Spring 3.0 появились дополнительные расширения к Spring MVC, обеспечившие превосходную поддержку REST. Теперь фреймворк Spring поддерживает разработку ресурсов REST, предоставляя следующее:

- контроллеры , способные обрабатывать все типы HTTP-запросов, включая четыре основных: GET, PUT, DELETE и POST;
- новая аннотация @PathVariable, позволяющая контроллерам обрабатывать запросы к параметризованным адресам URL (URL, имеющие переменную часть пути);
- JSP-тег <form:form> из библиотеки тегов JSP, входящей в состав Spring, а также новый компонент HiddenHttpMethodFilter, позволяющий отправлять запросы PUT и DELETE из HTML-форм даже в браузерах, не поддерживающих эти HTTP-запросы;

- представления и арбитры представлений в Spring, включая новые реализации представлений для отображения моделей данных в форматах XML, JSON, Atom и RSS, позволяют представлять ресурсы в самых разных форматах;
- с помощью нового арбитра представлений ContentNegotiatingView Resolver можно выбрать формат отображения ресурса, наиболее подходящий для клиента;
- механизм отображения данных с применением представлений можно полностью обойти с помощью новой аннотации @ResponseBody и различных реализаций интерфейса HttpMethodConverter;
- аналогично, для преобразования данных HTTP-запроса в Java-объекты, передаваемые методам-обработчикам контроллера, можно воспользоваться новой аннотацией @RequestBody наряду с реализациями интерфейса HttpMethod Converter;
- класс RestTemplate, упрощающий доступ к ресурсам REST на стороне клиента.

Далее будут исследоваться эти особенности, обеспечивающие поддержку архитектуры REST в Spring, а также демонстрироваться приемы экспортирования и использования ресурсов REST. Для начала мы познакомимся с особенностями контроллера Spring MVC, ориентированного на ресурсы.

2 Создание контроллеров, ориентированных на ресурсы

Как уже было изучено, в Spring MVC используется чрезвычайно гибкая модель создания классов контроллеров. Практически любой метод с практически любой сигнатурой можно объявить обработчиком веб-запросов. Однако из-за такой гибкости фреймворк Spring MVC позволяет создавать контроллеры, которые нелучшим образом подходят для обслуживания ресурсов RESTful. Слишком просто написать контроллер, противоречащий принципам архитектуры REST.

2.1 Устройство контроллера, противоречащего архитектуре REST

Чтобы понять, как должны выглядеть контроллеры, поддерживающие архитектуру REST, будет полезно узнать, как выглядят контроллеры, противоречащие этой архитектуре. Примером такого рода контроллеров может служить контроллер DisplayStudentController, представленный в листинге ниже.

```
@Controller
@RequestMapping("/displayStudent.html") // Отображение URL,
                                        // противоречащее REST
public class DisplayStudentController {
    private final StudentService studentService;

    @Autowired
    public DisplayStudentController(StudentService studentService) {
```

```
        this.studentService = srudentService;
    }

    @RequestMapping(method=RequestMethod.GET)
    public String showStudent(@RequestParam("id") long id, Model model)
    {
        model.addAttribute(studentService.getStudentById(id));
        return "views/show";
    }
}
```

Первое, на что следует обратить внимание в листинге выше – имя класса контроллера. Конечно, это всего лишь имя. Но оно точно описывает, что делает этот контроллер. Первое слово Display – глагол. Оно наглядно показывает, что этот контроллер ориентирован на выполнение действий, а не на предоставление доступа к ресурсам.

Отметьте аннотацию @RequestMapping на уровне класса. Она говорит, что данный контроллер будет обрабатывать запросы к странице /displayStudent.html. Похоже, что этот контроллер специализируется на представлении сообщений (что подтверждается именем класса). Более того, расширение предполагает, что информация будет отображаться в формате HTML.

Сама реализация контроллера DisplayStudentController не содержит никаких ошибок. Но она противоречит принципам архитектуры REST. Контроллер ориентирован на выполнение конкретной, узкоспециализированной операции: отображение объекта Student в формате HTML. Даже имя класса явно говорит об этом.

Теперь, когда вы узнали, как выглядит контроллер, противоречащий архитектуре REST, посмотрим, как выглядит контроллер, поддерживающий эту архитектуру. Для начала исследуем, как обрабатываются запросы к адресам URL ресурсов.

2.2. Обработка адресов URL в архитектуре RESTful

Адреса URL – это первое, о чем думает большинство людей, начиная работать с архитектурой REST. В конце концов, все, что делается в архитектуре REST, делается через URL. Самое забавное, что многие адреса URL обычно делают совсем не то, для чего они предназначены.

URL – это аббревиатура от Uniform Resource Locator (унифицированный указатель ресурсов). Согласно этому названию, URL должен служить ссылкой на ресурс. Более того, все адреса URL являются также идентификаторами URI, или Uniform Resource Identifiers (унифицированные идентификаторы ресурсов). Если это так, тогда от конкретного URL можно ожидать, что он не только является ссылкой на ресурс, но является еще и его идентификатором.

Тот факт, что URL определяет местоположение ресурса, выглядит вполне естественным. В конце концов, в течение многих лет мы привыкли вводить адреса URL в адресную строку браузера, чтобы отыскать требуемую информацию в Интернете. Но мы не привыкли считать, что URL является также уникальным идентификатором ресурса. Никакие два ресурса не могут размещаться по одному и тому же адресу URL, поэтому URL можно считать средством идентификации ресурса.

Многие адреса URL ни на что не указывают и ничего не идентифицируют – они выражают требования. Вместо того чтобы идентифицировать ресурс, они требуют выполнения некоторого действия.

Например, на рис. 1 изображен пример URL, обрабатываемого методом displayStudent() контроллера DisplayStudentController.

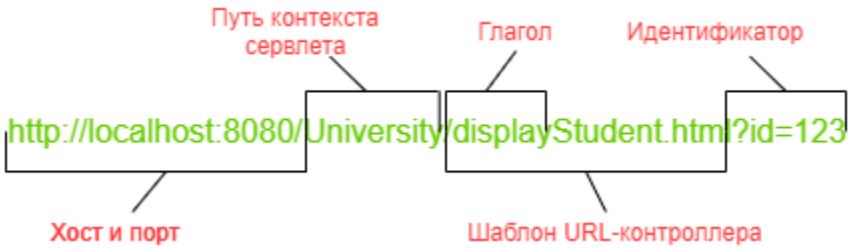

Рисунок 1 — URL, противоречащий архитектуре REST, ориентированный на выполнение операции, а также не ссылающийся и не идентифицирующий ресурс

Как показано на рис. 1, этот URL не ссылается и не идентифицирует какой-либо ресурс. Он требует, чтобы сервер отобразил объект Student. Единственный фрагмент URL, который хоть что-то идентифицирует, – это параметр запроса id. Базовая часть URL – глагол, выражающий требование. Все это говорит, что данный URL противоречит архитектуре REST. Прежде чем писать контроллеры, обрабатывающие адреса URL, полностью соответствующие архитектуре REST, необходимо выяснить, как должны выглядеть такие URL.

2.2.1 Характеристики URL, соответствующих архитектуре REST

В противоположность адресам URL, противоречащим архитектуре REST, адреса URL, соответствующие ей, целиком и полностью признают, что протокол HTTP предназначен для передачи ресурсов. Например, на рис. 2 показано, как мог бы выглядеть предыдущий URL после приведения его в согласие с архитектурой REST.

Единственное, что остается неясным в этом URL, – что он делает. Это обусловлено тем, что он ничего не делает. Он идентифицирует ресурс. В частности, он ссылается на ресурс, представляющий объект Student. Что будет делаться с этим ресурсом, это уже отдельная тема, зависящая от типа HTTP-запроса.

Рисунок 2 — URL, указывающий на ресурс и идентифицирующий его

Этот URL не только ссылается на ресурс, но и уникально идентифицирует его. Он в равной степени является адресом URL и идентификатором URI. Для полной идентификации ресурса используется весь URL, а не параметр запроса.

В действительности новый URL вообще не имеет параметров запроса. Несмотря на то что параметры все еще считаются допустимым способом передачи информации на сервер, теперь они являются лишь руководством для сервера, помогающим ему воспроизвести ресурс. Параметры запроса не должны использоваться для идентификации ресурса.

И последнее замечание, касающееся адресов URL в стиле RESTful: обычно они имеют иерархическую организацию. При чтении слева направо более общие понятия сменяются более специализированными. В данном примере URL содержит несколько уровней, любой из которых определяет ресурс.

http://localhost:8080 – определяет доменное имя и порт. Хотя в нашем приложении с этим URL не связано никаких ресурсов, нет объективных причин, почему этого нельзя было бы сделать.

http://localhost:8080/Univesity – определяет контекст сервлета приложения. Данный URL является более специализированным – он определяет приложение, выполняющееся на сервере.

http://localhost:8080/University/students – определяет ресурс, представляющий список объектов Student внутри приложения University

http://localhost:8080/University/students/123 – наиболее специализированный URL, определяющий конкретный ресурс Student. Самая интересная особенность адресов URL в стиле RESTful заключается в том, что пути в них являются параметризованными. В то время как URL, противоречащие архитектуре REST, несут входные данные в виде параметров запроса, адреса URL в стиле RESTful несут эти же данные внутри пути в URL. Для обработки запросов с такими URL необходимо иметь возможность, позволяющую методам-обработчикам контроллеров извлекать входные данные из пути URL.

2.2.2 Встраивание параметров в адреса URL

Для поддержки параметризованных путей в адресах URL в версию Spring 3.0 была добавлена новая аннотация @PathVariable . Чтобы познакомиться с ней в действии, рассмотрим класс StudentController, новый контроллер Spring MVC, реализующий подход, ориентированный на ресурсы, и обрабатывающий запросы на получение объектов Student.

```
@Controller
@RequestMapping("/students") // Обрабатывает запросы к URL /students
public class StudentController {

	private StudentService studentService;

	@Inject
	public StudentController(StudentService studentService) {
		this.studentService = studentService;
	}

	@RequestMapping(value="/{id}", // Используется переменная-
                                       заполнитель
		method=RequestMethod.GET)
	public String getStudent(@PathVariable("id") long id, Model model) {
		model.addAttribute(studentService.getStudentById(id));
		return "students/view";
	}
}
```

Как показано в листинге выше, класс StudentController снабжен аннотацией @RequestMapping, указывающей, что данный контроллер будет обрабатывать запросы на получение ресурсов Student – запросов с адресами URL, начинающимися со строки /students.

Пока существует только один метод-обработчик, getStudent(). Аннотация @RequestMapping, которой отмечен этот метод, в паре с аннотацией @RequestMapping на уровне класса превращает его в обработчик запросов типа GET с адресами URL вида /students/{id}.

Возможно, вас заинтересовали странные фигурные скобки в шаблоне URL. Конструкция {id} – это переменная-заполнитель, посредством которой изменяющаяся часть URL передается методу. Она соответствует аннотации @PathVariable, которой отмечен параметр id метода.

Таким образом, если приложение получит запрос GET с URL http://localhost:8080/University/students/123, метод getStudent() будет вызван со значением 123 в параметре id. Затем объект отыщет соответствующий параметру объект Student и поместит его в модель.

Возможно, вы заметили, что имя id трижды используется в сигнатуре метода. Оно используется не только как имя переменой-заполнителя в шаблоне

URL и как параметр аннотации @PathVariable, оно также используется как имя фактического параметра метода. В данном случае это всего лишь совпадение. Однако если имя параметра метода совпадает с именем переменной-заполнителя в шаблоне URL, тогда можно воспользоваться преимуществом наличия принятых соглашений и опустить параметр аннотации @PathVariable. Например:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public String getStudent(@PathVariable long id, Model model) {
        model.addAttribute(studentService.getStudentById(id));
         return "students/view";
}
```

Когда аннотация @PathVariable указывается без параметра, ее параметром становится имя аннотированного параметра метода. Независимо от того, как будет определяться имя переменной-заполнителя, явно или неявно, аннотация @PathVariable позволяет писать методы контроллеров для обработки запросов, в которых адреса URL идентифицируют ресурс, а не описывают некоторую операцию. С другой стороны, запросы в стиле RESTful – это методы HTTP, которые применяются к адресам URL. Посмотрим, как методы HTTP могут играть роль глаголов, описывающих действия, в запросах REST.

2.3. Выполнение операций в стиле REST

Как упоминалось выше, архитектура REST применяется для передачи информации о состоянии ресурса. Поэтому для описания операций над ресурсами достаточно лишь небольшого количества глаголов – глаголов, требующих передачи информации о состоянии ресурса. Наиболее типичными операциями над ресурсами являются создание, чтение, изменение и удаление ресурса. Глаголы, интересующие нас в данном случае (послать (post), получить (get), вставить (put) и удалить (delete)), непосредственно соответствуют четырем методам протокола HTTP, перечисленным в таблице ниже.

Метод	Описание	Безопасный	Идемпотентный
GET	Извлекает ресурс с сервера. Ресурс идентифицируется адресом URL в запросе	Да	Да
POST	Посылает данные на сервер для обработки процессором, ожидающим поступления запросов по адресу URL в запросе	Нет	Нет
PUT	Помещает данные в ресурс на сервере, идентифицируемый адресом URL в запросе	Нет	Да
DELETE	Удаляет ресурс, идентифицируемый адресом URL в запросе	Нет	Да
OPTIONS	Запрашивает дополнительные параметры для взаимодействия с сервером	Да	Да

HEAD	Напоминает метод GET, за исключением того, что в ответ возвращаются только заголовки – содержимое не должно возвращаться в теле ответа	Да	Да
TRACE	В ответ на этот запрос сервер должен вернуть его тело обратно клиенту	Да	Да

Каждый из HTTP-методов характеризуется двумя чертами: безопасность и идемпотентность. Метод считается безопасным, если он не изменяет состояние ресурса. Идемпотентный метод может изменять или не изменять состояние ресурса, но повторные запросы, следующие за первым, не должны оказывать влияния на состояние ресурса. По определению все безопасные методы являются также идемпотентными, но не все идемпотентные методы являются безопасными.

Важно отметить, что хотя фреймворк Spring поддерживает все методы протокола HTTP, это не освобождает разработчика от обязанности следить, чтобы реализация методов контроллеров соответствовала семантике HTTP-методов. Иными словами, метод, обрабатывающий GET-запросы, должен только возвращать ресурс – он не должен изменять или удалять его.

Четыре первых HTTP-метода из перечисленных в таблице выше час то отображаются в CRUD-операции (Create/Read/Update/Delete – создать/прочитать/изменить/удалить). В частности, метод GET выполняет операцию чтения, а метод DELETE – операцию удаления. И даже при том, что методы PUT и POST могут использоваться для выполниния других операций, отличных от операций изменения и создания, обычно принято использовать их по прямому назначению.

Выше уже демонстрировался пример обработки GET-запросов. Метод getStudent() класса StudentController снабжен аннотацией @RequestMapping, в которой атрибуту method присвоено значение GET. Атрибут method определяет, какой HTTP-метод будет обрабатываться данным методом контроллера.

2.3.1 Изменение ресурса с помощью PUT-запросов

Что касается метода PUT, его семантика полностью противоположна методу GET. В противоположность GET-запросу, требующему передать информацию о состоянии ресурса клиенту, PUT-запрос передает информацию о состоянии ресурса на сервер.

Например, следующий метод putStudent() предназначен для приема объекта Student в составе PUT-запроса:

```
@RequestMapping(value="/{id}", method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void putStudent(@PathVariable("id") long id,
                       @Valid Student student {
        studentService.saveStudent(student);
}
```

Метод putStudent() отмечен аннотацией @RequestMapping как любой другой метод-обработчик. Фактически данная аннотация @RequestMapping почти ничем не отличается от аннотации для метода getStudent). Единственное отличие заключается в том, что атрибуту method присвоено значение PUT.

Поскольку это единственное отличие, следовательно, метод putStudent() будет обрабатывать запросы с адресами URL вида /students/{id}, аналогично методу getStudent(). Напомню, что URL идентифицирует ресурс, а не операцию над ним. Поэтому URL, идентифицирующий объект Student, будет тем же самым что для метода GET, что для метода PUT.

Кроме того, putStudent() метод отмечен аннотацией, не встречавшейся нам до сих пор. Аннотация @ResponseStatus определяет код состояния HTTP, который должен быть установлен в ответе, отправляемом клиенту. В данном случае значение HttpStatus.NO_CONTENT указывает, что клиенту необходимо вернуть код состояния HTTP 204. Этот код означает, что запрос был успешно обработан, но тело ответа не содержит никакой дополнительной информации.

2.3.2 Обработка DELETE-запросов

Иногда бывает необходимость не просто изменить ресурс, а вообще удалить его. В случае с приложением University, например, можно дать некоторым лицам возможность удалять информацию о студентах, в случае отчисления или перевода из одного университета в другой. Когда надобность в ресурсе отпадает, можно задействовать HTTP-метод DELETE.

Для демонстрации обработки DELETE-запросов в Spring MVC добавим в класс StudentController новый метод-обработчик, который в ответ на DELETE-запросы будет удалять ресурсы Student:

```
@RequestMapping(value="/{id}", method=RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteStudent(@PathVariable("id") long id) {
        studentService.deleteStudent(id);
}
```

И снова аннотация @RequestMapping оказалась очень похожей на аналогичные аннотации методов getStudent() и putStudent(). Она отличается только атрибутом method , который на этот раз получил значение DELETE. Шаблон URL, идентифицирующий ресурс, остался прежним.

Подобно методу putStudent(), метод deleteStudent() также отмечен аннотацией @ResponseStatus, чтобы известить клиента об успешной обработке запроса и об отсутствии дополнительной информации в теле ответа.

2.3.3 Создание ресурса с помощью POST-запроса

В каждой компании найдется свой человек, отличающийся свободомыслием и несогласием с остальными. Среди HTTP-методов таким

несогласием отличается метод POST . Он не подчиняется общепринятым правилам. Он опасен и конечно же не идемпотентен. Этот HTTP-метод нарушает, кажется, все правила, но благодаря этому он выполняет работу, непосильную для других HTTP-методов.

Чтобы увидеть, как действует метод POST, посмотрим, как он выполняет работу, которую ему часто поручают, – создание нового ресурса. Метод createStudent(), представленный в листинге ниже, реализует обработку POST-запросов и создает новые ресурсы Student.

```
@RequestMapping(method=RequestMethod.POST)
                                // Обрабатывает POST-запросы
@ResponseStatus(HttpStatus.CREATED)
                                // Возвращает ответ HTTP 201
public @ResponseBody Student createStudent(@Valid Student student,
BindingResult result, HttpServletResponse response) throws BindException {
        if(result.hasErrors()) {
                throw new BindException(result);
        }
        studentService.saveStudent(student);
                                // Указать местоположение ресурса
        response.setHeader("Location", "/student/" + student.getId());
        return student; // Вернуть ресурс
}
```

Первое, на что следует обратить внимание в этом методе – аннотация @RequestMapping , отличающаяся от аналогичных аннотаций, встречавшихся до сих пор. В отличие от них, данная аннотация не имеет атрибута value . Это означает, что определение шаблона адресов URL, обрабатываемых методом createStudent(), целиком и полностью возлагается на аннотацию @RequestMapping на уровне класса.

Точнее, метод createStudent() будет обрабатывать запросы, соответствующие шаблону URL /students.

Обычно идентичность ресурса определяется сервером. Поскольку в данном случае создается новый ресурс, нет никакой возможности определить его идентификатор в URL. То есть запросы GET, PUT и DELETE воздействуют непосредственно на ресурс, идентифицируемый адресом URL, а запрос POST вынужден использовать URL, который не является ссылкой на создаваемый ресурс (нельзя определить адрес URL несуществующего ресурса).

И снова метод отмечен аннотацией @ResponseStatus, определяющей код состояния в ответе, отправляемом клиенту. На этот раз возвращается код состояния HTTP 201 (создано), свидетельствующий, что ресурс был успешно создан. При возврате кода состояния HTTP 201 вместе с ним необходимо вернуть клиенту и URL нового ресурса. Поэтому в конце метода createStudent() определяется заголовок Location, содержащий URL ресурса.

Хотя это и не обязательно, но в теле ответа с кодом HTTP 201 можно вернуть полное представление ресурса. Поэтому, подобно методу getStudent(), обрабатывающему GET-запросы, данный метод завершается, возвращая новый объект Student. Этот объект будет трансформирован в некоторое представление, которое сможет быть использовано клиентом. Неясным пока остается сам процесс трансформации. Или как будет выглядеть представление. Рассмотрим букву R в аббревиатуре REST: representation (представление).

3. Представление ресурсов

Представление – важный аспект архитектуры REST, определяющий форму ресурсов при взаимодействиях между клиентом и сервером. Любой ресурс может быть представлен практически в любой форме. Если потребитель ресурса предпочитает формат JSON, ресурс может быть представлен в формате JSON. Если потребитель испытывает слабость к угловым скобкам, тот же самый ресурс может быть представлен в формате XML. Большинство людей, просматривающих ресурсы в веб-браузере, предпочтут получать их в формате HTML (или, может быть, PDF, Excel или каком-то другом удобочитаемом формате). Сам ресурс при этом не изменяется – изменяется только его представление.

Важно помнить, что обычно контроллеры никак не определяют формат представления ресурса. Контроллеры обрабатывают ресурс в терминах Java-объектов. Но как только контроллер завершит свою работу, ресурс тут же будет трансформирован в формат, ожидаемый клиентом.

Фреймворк Spring предоставляет два способа преобразования ресурса из представления на языке Java в представление, которое может быть отправлено клиенту:

- отображение с помощью представлений на основе договоренностей;
- преобразование HTTP-сообщений;

Ранее мы уже обсуждали арбитры представлений и знаем, как реализовать отображение с помощью представлений. Поэтому сначала посмотрим, как обеспечить отображение ресурса в требуемый формат, выбирая представление или арбитра представлений, исходя из договоренностей с клиентом.

3.1. Договоренность о представлении ресурса

Как рассказывалось в предыдущих темах, метод-обработчик контроллера обычно возвращает логическое имя представления. Даже если метод не возвращает это имя непосредственно (например, если метод вообще ничего не возвращает), тогда логическое имя представления определяется на основе адреса URL в запросе. Затем сервлет DispatcherServlet передает имя представления арбитру представлений, предлагая ему определить конкретное представление, которое должно использоваться для отображения результатов.

В веб-приложениях, взаимодействующих с человеком, обычно всегда выбирается представление, отображающее результаты в формате HTML. Выбор представления осуществляется по одному параметру – имени представления.

Что касается преобразования имен представлений в фактические представления отображения ресурсов, в операцию выбора добавляется еще один параметр. Представление не только должно соответствовать имени, но и отображать данные в формате, требуемом клиенту. Если клиент требует представить ресурс в формате XML, то представление, возвращающее данные в формате HTML, не годится, даже если оно соответствует указанному имени.

В состав Spring входит класс ContentNegotiatingViewResolver – специализированный арбитр представлений, который при выборе представления учитывает также, в каком формате клиент желает получить ресурс. Подобно любым другим арбитрам представлений, он настраивается в виде компонента в контексте приложения Spring, как показано в листинге ниже.

```
<bean id="contentNegotiationManager" class="org.springframework.web
                                .accept.ContentNegotiationManagerFactoryBean">
        <property name="favorPathExtension" value="true" />
        <property name="ignoreAcceptHeader" value="true"/>
        <property name="useJaf" value="false"/>
        <property name="defaultContentType" value="text/html" />

        <property name="mediaTypes">
                <map>
                        <entry key="html" value="text/html"/>
                        <entry key="json" value="application/json"/>
                </map>
        </property>
</bean>

<bean class="org.springframework.web.servlet.view.
                                        ContentNegotiatingViewResolver">
        <property name="contentNegotiationManager"
                 ref="contentNegotiationManager"/>
                <property name="viewResolvers">
        <list>
            <bean class="org.springframework.web.servlet.view.freemarker.
                                                    FreeMarkerViewResolver"/>
       </list>
</property>
</bean>
```

Чтобы понять принцип действия арбитра ContentNegotiatingView Resolver, необходимо знать, что договоренность о формате ресурса выполняется в два этапа:

1. Определяется тип возвращаемых данных.

2. Отыскивается представление для данного типа.

Рассмотрим подробнее каждый из этих этапов, чтобы лучше понять механизм действия ContentNegotiatingViewResolver. Начнем с определения типа возвращаемых данных.

3.1.1 Определение запрошенного типа возвращаемых данных

Первый этап в процессе определения формата представления ресурса заключается в выяснении требований клиента. На первый взгляд кажется, что в этом нет ничего сложного, поскольку клиент может явно указать требуемый формат в заголовке Accept запроса.

К сожалению, заголовок Accept не является достаточно надежным для этого. Если клиент пользуется веб-браузером, нет никакой гарантии, что клиент желает получить именно то, что браузер отправляет в заголовке Accept. Обычно веб-браузеры указывают форматы, доступные для восприятия человеком (такие как text/html) и не предоставляют возможности указать другой формат (за исключением случаев использования расширений, предназначенных для разработчиков).

Арбитр ContentNegotiatingViewResolver учитывает содержимое заголовка Accept и выбирает то или иное представление в соответствии с ним, но только после того, как попытается определить расширение файла в URL. Если в конце URL присутствует расширение файла, будет выполнена попытка отыскать соответствующий ему элемент в свойстве mediaTypes. Свойство mediaTypes – это отображение, ключами в котором являются расширения имен файлов, а значениями – типы содержимого. При обнаружении совпадения используется соответствующий тип содержимого. При таком подходе расширения имен файлов пользуются более высоким приоритетом по отношению к заголовку Accept.

Если имеющееся расширение не соответствует ни одному поддерживаемому типу содержимого, тогда в учет принимается содержимое заголовка Accept. Но если запрос не имеет заголовка Accept, тогда используется тип содержимого, определяемый свойством defaultContentType.

В качестве примера предположим, что арбитр ContentNegotiatingViewResolver настроен, как показано в листинге выше, и ему предлагается определить тип содержимого по запросу, содержащему расширение файла .json. В данном случае этому расширению соответствует элемент с ключом json в свойстве mediaTypes. Поэтому будет выбран тип содержимого application/json.

А теперь допустим, что запрос содержит расширение .www . Этому расширению не соответствует ни один элемент в свойстве mediaTypes. В отсутствие совпадения ContentNegotiatingViewResolver попытается определить тип содержимого по заголовку Accept в запросе. Запросы, отправляемые браузером Firefox, содержат типы text/html, application/xhtml+xml, application

/xml и */*. Если запрос не содержит заголовка Accept, тогда будет выбран тип text/html, согласно значению свойства defaultContentType.

3.1.2 Изменение порядка определения типа содержимого

Порядок определения типа содержимого, описанный выше, соответствует стратегии по умолчанию. Однако существуют дополнительные возможности, позволяющие влиять на этот порядок:

- присвоив свойству favorPathExtension значение false, можно заставить ContentNegotiatingViewResolver игнорировать расширение файла в URL;
- добавив фреймворк Java Activation Framework (JAF) в библиотеку классов (classpath), можно заставить ContentNegotiatingViewResolver обращаться к JAF за помощью в определении типа содержимого по расширению имени файла, если в свойстве mediaTypes не будет найден соответствующий элемент;
- если присвоить свойству favorParameter значение true, тип содержимого будет определяться путем сопоставления параметра format в запросе (если присутствует) с элементами в свойстве mediaTypes (кроме того, имя параметра можно изменить, определив свойство parameterName);
- присвоив свойству ignoreAcceptHeader значение true, можно исключить из рассмотрения заголовок Accept.

Например, допустим, что свойству favorParameter было присвоено значение true:

<property name="favorParameter" value="true" />

Теперь запросы, в которых URL не содержит расширения имени файла, будут соответствовать типу содержимого application/json, если URL включает параметр запроса format со значением json.

После того как ContentNegotiatingViewResolver определит тип содержимого, можно приступать к поиску представления, которое сможет отобразить данные в содержимое этого типа.

3.1.3 Поиск представления

В отличие от других арбитров представлений, ContentNegotiatingView Resolver не определяет представление непосредственно, а предоставляет другим арбитрам представлений возможность выбрать наиболее подходящее представление, соответствующее требованиям клиента. Если не оговаривается иное, он будет использовать любые арбитры представлений, имеющиеся в контексте приложения. Но имеется возможность явно перечислить арбитры представлений, которые следует задействовать, перечислив их в свойстве viewResolvers.

ContentNegotiatingViewResolver опросит все арбитры представлений, предложив им определить представление по логическому имени, и поместит полученные представления в список кандидатов. Кроме того, если определено

свойство defaultView , представление, указанное в нем, также будет добавлено в конец списка.

После составления списка кандидатов ContentNegotiatingViewResolver выполнит обход по всем запрашиваемым типам содержимого, пытаясь отыскать соответствующее представление из числа кандидатов. Поиск производится до первого совпадения.

Если представление определить не удалось, ContentNegotiatingViewResolver вернет пустую ссылку (null). Или, если свойство useNotAcceptable StatusCode имеет значение true, будет возвращено представление с кодом состояния HTTP 406 (Not Acceptable).

Прием определения договоренности о формате представления ресурса поволяет создавать новые представления вдобавок к уже имеющимся HTML-представлениям. При определении ресурсов в стиле RESTful, потребляемых программными клиентами, возможно, имеет смысл создать контроллер, который будет осведомлен, что данные потребляются как ресурс другим приложением. В этом случае можно задействовать инструменты преобразования HTTP-сообщений и аннотацию @ResponseBody.

3.2 Преобразование HTTP-сообщений

Как было изучено в теме «MVC», типичный метод контроллера Spring MVC завершается записью некоторых данных в модель и определением логического имени представления для отображения этих данных. Несмотря на большое разнообразие способов заполнения модели данными и идентификации представлений, все методы-обработчики контроллеров, встречавшиеся нам до сих пор, следовали этому основному шаблону.

Но когда задача контроллера состоит в том, чтобы воспроизвести ресурс в некотором формате, существует более прямой путь к цели, минуя модели и представления. При таком подходе к реализации метода-обработчика возвращаемый им объект автоматически преобразуется в формат, запрошенный клиентом.

Использование этого нового приема начинается с применения аннотации @ResponseBody к методу-обработчику контроллера.

3.2.1 Возврат ресурса в теле ответа

Обычно , когда метод-обработчик возвращает Java-объект (любого типа, отличного от String), этот объект помещается в модель для последующего отображения в представлении. Но если метод отметить аннотацией @ResponseBody, тогда возвращаемый им объект будет передан механизму преобразования HTTP-сообщений и превращен в формат, требуемый клиентом.

Например, рассмотрим следующий метод getStudent() класса StudentController:

```
@RequestMapping(value = "/{username}", method = RequestMethod.GET,
                headers = {"Accept=text/xml, application/json"})
```

```
public @ResponseBody Student getStudent(@PathVariable String username)
{
        return studentService.getStudent(username);
}
```

Аннотация @ResponseBody сообщает фреймворку Spring, что возвращаемый объект следует отправить клиенту как ресурс, преобразовав его в некоторый формат, доступный для клиента. Точнее, ресурс должен быть преобразован в формат в соответствии с содержимым заголовка Accept. Если в запросе отсутствует заголовок Accept, тогда предполагается, что клиент способен принимать ресурсы в любом формате.

Что касается заголовка Accept, обратите внимание на аннотацию @RequestMapping перед методом getStudent(). Атрибут headers указывает, что этот метод будет обрабатывать только запросы, в которых заголовок Accept включает text/xml или application/json. Любые другие запросы, даже если это будут GET-запросы, в которых URL соответствует указанному шаблону, не будут обрабатываться данным методом. Они либо будут обработаны другим методом (если имеется соответствующий метод), либо клиенту будет отправлен ответ с кодом состояния HTTP 406 (Not Acceptable).

Преобразование произвольных Java-объектов, возвращаемых методами обработчиками, в представление, доступное для клиента, выполняется одним из преобразователей HTTP-сообщений, входящих в состав Spring.

С ними можете ознакомиться самостоятельно. (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jms/support/converter/MessageConverter.html)

Например, допустим, что в заголовке Accept запроса клиент сообщил, что он способен принимать данные в формате application/json. Допустим также, что в библиотеке классов приложения присутствует библиотека Jackson JSON. В этом случае объект, возвращаемый методом-обработчиком, можно передать преобразователю MappingJacksonHttpMessageConverter для преобразования его в формат JSON перед передачей клиенту. С другой стороны, если заголовок в запросе указывает, что клиент предпочитает формат text/xml, преобразование данных в формат XML можно поручить преобразователю Jaxb2Root ElementHttpMessageConverter.

Обратите внимание, что преобразователи HTTP-сообщений регистрируются по умолчанию, поэтому их не требуется настраивать отдельно. Однако для их поддержки может понадобиться добавить дополнительные библиотеки в библиотеку классов приложения (classpath). Например, для преобразования данных в формат JSON и обратно с помощью преобразователя MappingJacksonHttpMessageConverter необходимо будет добавить библиотеку Jackson JSON Processor.

3.2.2 Прием ресурса в теле запроса

Другому участнику диалога в стиле RESTful, клиенту, может потребоваться отправить на сервер объект в формате JSON, XML или каком-то другом. Было бы неудобно получать эти объекты в исходном виде в методе контроллера и пытаться преобразовать их вручную. К счастью, аннотация @RequestBody позволяет выполнить все необходимые преобразования объектов, отправленных клиентом, как это делает аннотация @ResponseBody с объектами, возвращаемыми клиентам.

Допустим, что клиент отправил запрос PUT с данными для объекта Student в формате JSON. Чтобы принять это сообщение как объект Student, достаточно всего лишь отметить соответствующий параметр типа Student метода-обработчика аннотацией @RequestBody:

```
@RequestMapping(value = "/{username}",
method = RequestMethod.PUT, headers = "Content-Type=application/json")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void updateStudent(@PathVariable String username, @RequestBody Student student) {
    studentService.saveStudent(student);
}
```

Получив запрос, фреймворк Spring MVC определит, что этот запрос должен обрабатывать метод updateStudent(). Но полученное сообщение имеет формат JSON, а метод ожидает получить объект Student. В этом случае для преобразования сообщения из формата JSON в объект Student можно было бы задействовать преобразователь MappingJacksonHttpMessageConverter. Чтобы этот выбор состоялся, должны быть выполнены следующие условия:

- заголовок Content-Type запроса должен иметь значение application/ json;
- библиотека Jackson JSON должна находиться в библиотеке классов приложения.

Возможно, вы обратили внимание, что метод updateStudent() также отмечен аннотацией @ResponseStatus. После обработки запроса PUT не требуется возвращать клиенту какие-либо данные, а аннотирование метода updateStudent() таким способом обеспечивает отправку клиенту кода состояния HTTP 204 (No Content).

К настоящему моменту у нас имеются несколько контроллеров Spring MVC с методами-обработчиками для обработки запросов на получение ресурсов. Нам предстоит обсудить еще ряд тем, касающихся определения RESTful API с помощью Spring MVC, и мы вернемся к этому обсуждению в разделе. Но перед этим немного отвлечемся и посмотрим, как можно использовать класс RestTemplate для создания клиентов, потребляющих ресурсы.

4 Клиенты REST

Обычно в нашем представлении веб-приложения ассоциируются с пользовательским интерфейсом в веб-браузере. Но к веб-приложениям, реализующим работу с ресурсами в стиле RESTful, это не относится. Сам факт передачи данных через Всемирную паутину еще не означает, что эти данные обязательно должны отображаться в окне веб-браузера. Более того, можно даже написать веб-приложение, взаимодействующее с другим веб-приложением посредством RESTful API.

Разработка программ, взаимодействующих с ресурсами в стиле RESTful, может оказаться весьма утомительным занятием, требующим писать массу шаблонного кода. Например, допустим, что потребовалось написать некоторый клиентский программный код, который пользовался бы прикладным интерфейсом извлечения списка студентов из приложения. В листинге ниже представлено одно из возможных решений этой задачи.

```
 public Student[] retrieveStudents(String name) {
try {
        HttpClient httpClient = new DefaultHttpClient();
        String studentsUrl = "http://localhost:8080/University/students/"
                                + name; // Подготовить URL
        HttpGet getRequest = new HttpGet(studentsUrl); // Создать запрос
        getRequest.setHeader( new BasicHeader("Accept", "application/json"));
        HttpResponse response = httpClient.execute(getRequest); // Выполнить
        HttpEntity entity = response.getEntity(); // Извлечь результаты
        ObjectMapper mapper = new ObjectMapper();
        return mapper.readValue(entity.getContent(), Student[].class);
} catch (IOException e) {
         throw new StudentClientException("Unable to retrieve Students", e);
}
}
```

Как видите, чтобы получить ресурс REST, необходимо немало потрудиться. И это при том, что мы немного сжульничали, воспользовавшись библиотекой Jakarta Commons HTTP Client, чтобы создать запрос, и библиотекой Jackson JSON Processor для преобразования ответа.

Взглянув поближе на метод retrieveSetudents(), можно заметить, что здесь не так много программного кода связано с реализацией конкретной функциональностью. Если теперь написать другой метод, извлекающий другой ресурс REST, обнаружится, что эти два метода имеют совсем немного отличий.

Более того, метод выполняет множество операций, каждая из которых может возбудить исключение IOException . Поскольку IOException является контролируемым исключением, я вынужден предусмотреть либо его обработку, либо повторное его возбуждение. В данном случае я предпочел перехватить исключение и взамен возбудить неконтролируемое исключение StudentClientException .

С таким количеством шаблонного кода, связанного с извлечением ресурса, было бы неплохо иметь возможность инкапсулировать его и обеспечить различные варианты выполнения с помощью параметров. Именно эту цель преследует класс RestTemplate , входящий в состав фреймворка Spring. Подобно тому, как JdbcTemplate прячет неудобства работы с данными при помощи JDBC, RestTemplate освобождает программиста от реализации рутинных операций при работе с ресурсами RESTful.

Чуть ниже будет показано, как можно переписать метод retrieveStudents() и с помощью RestTemplate существенно уменьшить объем шаблонного кода. Но сначала познакомимся со всеми REST-операциями, которые может предложить класс RestTemplate.

4.1. Операции класса RestTemplate

Ранее в таблице был представлен список из семи методов HTTP, применяемых для взаимодействия с ресурсами RESTful. Эти методы играют роль глаголов в диалоге в стиле RESTful.

Класс RestTemplate определяет 33 метода и использует все методы HTTP для взаимодействия с ресурсами REST. К сожалению, в этой главе не так много места, чтобы можно было подробно рассмотреть все 33 метода. Но, как оказывается, в действительности класс поддерживает всего 11 уникальных операций, каждая из которых имеет три реализации в виде перегруженных методов. Список из 11 уникальных операций, поддерживаемых классом RestTemplate, представлены в списке:

- **delete()** Выполняет запрос HTTP DELETE к ресурсу с указанным URL
- **exchange()** Выполняет HTTP-запрос требуемого типа к ресурсу с указанным URL и возвращает экземпляр ResponseEntity, содержащий объект, отображающий тело ответа
- **execute()** Выполняет HTTP-запрос требуемого типа к ресурсу с указанным URL, возвращает объект, отображающий тело ответа
- **getForEntity()** Выполняет запрос HTTP GET и возвращает экземпляр ResponseEntity, содержащий объект, отображающий тело ответа
- **getForObject()** Выполняет запрос HTTP GET и возвращает объект, отображающий тело ответа
- **headForHeaders()** Выполняет запрос HTTP HEAD к ресурсу с указанным URL и возвращает заголовки ответа
- **optionsForAllow()** Выполняет запрос HTTP OPTIONS и возвращает заголовок Allow для указанного URL
- **postForEntity()** Выполняет запрос HTTP POST и возвращает экземпляр ResponseEntity, содержащий объект, отображающий тело ответа
- **postForLocation()** Выполняет запрос HTTP POST и возвращает URL нового ресурса

- **postForObject()** Выполняет запрос HTTP POST и возвращает объект, отображающий тело ответа put() Выполняет запрос HTTP PUT, отправляя измененный ресурс с указанным URL

Класс RestTemplate использует все методы HTTP, за исключением TRACE. Кроме того, методы execute() и exchange() предлагают возможность выполнения любых HTTP-запросов. Каждая операция из представленных в списке реализована в форме трех перегруженных методов:

- один принимает URL в виде java.net.URI без поддержки параметризованных адресов URL;
- один принимает URL в виде строки с параметрами URL в ви де экземпляра Map;
- один принимает URL в виде строки, с параметрами URL в виде списка аргументов переменной длины.

Познакомившись с 11 операциями, предоставляемыми классом RestTemplate, и особенностями их использования, вы без труда сможете создавать собственные клиентские приложения для работы с ресурсами REST. Познакомимся поближе с операциями, предлагаемыми классом RestTemplate, которые используют четыре основных метода HTTP: GET, PUT, DELETE и POST. И начнем знакомство с операций getForObject() и getForEntity(), использующих метод GET.

4.2. Чтение ресурсов

Вы могли заметить, что в списке выше перечислены два метода, выполняющих GET-запросы: getForObject() и getForEntity(). Как отмечалось выше, каждый из этих методов имеет три перегруженные версии. Ниже приводятся сигнатуры трех версий метода getForObject( ):

```
<T> T getForObject(URI url, Class<T> responseType)
        throws RestClientException;
<T> T getForObject(String url, Class<T> responseType, Object... uriVariables)
        throws RestClientException;
<T> T getForObject(String url, Class<T> responseType, Map<String, ?> uriVariables)
        throws RestClientException;
```

И сигнатуры трех версий метода getForEntity( ):

```
<T> ResponseEntity<T> getForEntity(URI url, Class<T> responseType)
        throws RestClientException;
<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType, Object... uriVariables)
        throws RestClientException;
<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType, Map<String, ?> uriVariables)
        throws RestClientException;
```

За исключением типа возвращаемого значения, методы getForObject() являются зеркальным отражением методов getForEntity(). И в действительности они действуют точно так же. Оба метода выполняют GET-запрос, чтобы извлечь ресурс с указанным URL. И оба отображают этот ресурс в объект некоторого типа, определяемого параметром responseType. Единственное отличие состоит в том, что метод getForObject() просто возвращает объект требуемого типа, а метод getForEntity() возвращает этот объект наряду с дополнительной информацией об ответе.

Рассмотрим сначала более простой метод getForObject(). Затем узнаем, как получить дополнительную информацию из ответа на GET-запрос с помощью метода getForEntity().

4.2.1 Извлечение ресурсов

Метод getForObject() является сугубо прагматичным способом извлечения ресурсов. С его помощью программа запрашивает ресурс и получает его в форме объекта указанного типа. В качестве простого примера использования метода getForObject() рассмотрим еще одну версию метода retrieveStudents():

```
public Students[] retrieveStudents(String name) {
 return new RestTemplate().getForObject(
                "http://localhost:8080/University/students/{name}",
                 Student[].class, name);
}
```

Применение класса RestTemplate позволило сократить метод до нескольких строк (и строк было бы еще меньше, если бы их не нужно было переносить, чтобы уместить по ширине книжной страницы).

Новая версия метода retrieveStudents() начинается с создания экземпляра RestTemplate (при желании можно было бы использовать внедренный экземпляр), а затем вызывается метод getForObject(), чтобы извлечь список сообщений. То есть запрашивается массив объектов Student. После получения массива он возвращается вызывающей программе.

Обратите внимание, что для конструирования URL в этой новой версии метода retrieveStudent() не используется операция конкатенации строк. Вместо этого используется тот факт, что RestTemplate принимает параметризованные адреса URL. Переменная-заполнитель {name} в URL будет замещена значением параметра name метода. Последний аргумент метода getForObject( ) – это список аргументов переменной длины, где каждый аргумент замещает переменные-заполнители в URL в порядке их следования.

Здесь отсутствует явное преобразование данных в формате JSON в объект. Преобразование тела ответа в требуемый объект выполняется методом getForObject() автоматически. Это преобразование выполняется с привлечением

все тех же преобразователей HTTP-сообщений, о которых говорилось ранее, которые Spring MVC использует в методах-обработчиках, отмеченных аннотацией @ResponseBody.

В этом методе также отсутствует обработка исключений. И вовсе не потому, что метод getForObject() не может возбуждать исключения, а потому, что любые исключения, возбуждаемые им, являются неконтролируемыми. Если в методе getForObject() что-то пойдет не так, он возбудит неконтролируемое исключение RestClientException. При желании его можно перехватить, но компилятор не вынуждает делать это.

4.2.2 Извлечение метаданных из ответа

В качестве альтернативы методу getForObject() класс RestTemplate предлагает метод getForEntity( ). Этот метод действует практически так же, как метод getForObject(). Но если метод getForObject() возвращает только ресурс (преобразованный в Java-объект с помощью преобразователя HTTP-сообщений), то метод getForEntity() возвращает тот же самый объект, помещенный внутрь объекта ResponseEntity. Объект ResponseEntity несет в себе также дополни- тельную информацию об ответе, такую как код состояния HTTP и заголовки ответа.

Объект ResponseEntity можно использовать, чтобы извлечь значение какого-либо заголовка ответа. Например, представьте, что помимо ресурса необходимо также определить, когда ресурс изменялся в последний раз. Допустим, что сервер предоставляет такую информацию в заголовке Last-Modified. Тогда, чтобы получить требуемую информацию, можно воспользоваться методом getHeaders( ), как показано ниже:

```
Date lastModified = new Date(response.getHeaders().getLastModified());
```

Метод getHeaders() возвращает объект HttpHeaders, имеющий несколько вспомогательных методов для извлечения заголовков, включая метод getLastModified( ), возвращающий количество миллисекунд, прошедших с 1 января 1970 года.

Кроме метода getLastModified(), класс HttpHeaders содержит следующие методы:

```
public List<MediaType> getAccept() { ... }
public List<Charset> getAcceptCharset() { ... }
public Set<HttpMethod> getAllow() { ... }
public String getCacheControl() { ... }
public long getContentLength() { ... }
public MediaType getContentType() { ... }
public long getDate() { ... }
public String getETag() { ... }
public long getExpires() { ... }
public long getIfNotModifiedSince() { ... }
public List<String> getIfNoneMatch() { ... }
```

```
public long getLastModified() { ... }
public URI getLocation() { ... }
public String getPragma() { ... }
```

Более универсальный способ доступа к HTTP-заголовкам обеспечивают методы get( ) и getFirst( ). Оба принимают строковый аргумент, определяющий имя заголовка. Метод get() возвращает список строковых значений – по одному для каждого заголовка. Метод getFirst() возвращает значение только первого заголовка. Чтобы получить код состояния HTTP, можно воспользоваться методом getStatusCode( ). Например, взгляните на реализацию метода retrieveStudents() в листинге ниже.

```
public Student[] retrieveStudents(String name) {
        ResponseEntity<Student[]> response = new
RestTemplate().getForEntity(
                "http://localhost:8080/University/students/{name}",
                Student[].class, name);
        if(response.getStatusCode() == HttpStatus.NOT_MODIFIED) {
                throw new NotModifiedException();
        }
        return response.getBody();
}
```

Если сервер вернет код состояния HTTP 304, это будет свидетельствовать о том, что содержимое на сервере не изменялось с момента последнего обращения клиента. В этом случае метод возбудит исключение NotModifiedException , чтобы сообщить, что клиент должен извлечь данные из своего кэша.

4.3 Изменение ресурсов

Для выполнения операции PUT над ресурсом RestTemplate предлагает набор из трех простых методов put( ). Как и все методы класса RestTemplate, метод put() имеет три версии:

```
void put(URI url, Object request) throws RestClientException;
void put(String url, Object request, Object... uriVariables)
                throws RestClientException;
void put(String url, Object request, Map<String, ?> uriVariables)
                throws RestClientException;
```

Простейшая версия метода put() принимает объект java.net.URI, идентифицирующий ресурс (и определяющий его местоположение), отправляемый на сервер, и Java-объект, представляющий этот ресурс.

Например, ниже показано, как можно использовать URI-версию метода put(), чтобы изменить ресурс Student на сервере:

```
public void updateStudent(Student student) throws StudentException {
        try {
        String url = "http://localhost:8080/University/students/"+ student.getId();
        new RestTemplate().put(new URI(url), student);
} catch (URISyntaxException e) { throw new StudentUpdateException("Unable to update Student", e); }
}
```

Несмотря на простую сигнатуру метода, возникают некоторые сложности с аргументом типа java.net.URI. Во-первых, чтобы создать URL изменяемого объекта Student, необходимо использовать операцию конкатенации строк. Во-вторых, поскольку есть вероятность сконструировать недопустимый URI, который затем передается конструктору класса URI, приходится перехватывать исключение URISyntaxException (даже если мы совершенно уверены в допустимости URI).

Другие версии метода put(), основанные на использовании строк, избавляют от неудобств, связанных с созданием экземпляра URI, включая необходимость перехватывать исключения. Более того, эти версии метода позволяют определять URI в виде шаблона, позволяя передавать переменные части шаблона. Ниже представлена измененная версия метода updateStudent(), использующая один из методов put(), основанных на применении строк:

```
public void updateStudent(Student student) throws StudentException {
        restTemplate.put("http://localhost:8080/University/students/{id}",
                student, student.getId());
}
```

Теперь URI определяется как простой строковый шаблон. Когда RestTemplate отправляет PUT-запрос, вместо {id} в шаблон URI будет подставлено значение, возвращаемое методом student.getId(). Подобно методам getForObject() и getForEntity(), последний аргумент этой версии метода put() является списком аргументов переменной длины, замещающих переменные-заполнители в шаблоне в порядке их следования.

Во всех версиях put() во втором аргументе передается Java-объект, представляющий изменяемый ресурс, который будет отправлен на сервер методом PUT. В данном случае это объект Student. Для преобразования объектов Student перед отправкой на сервер класс RestTemplate будет использовать преобразователи HTTP-сообщений.

Формат преобразования объекта в значительной степени зависит от типа объекта, переданного методу put(). Если передать методу строковое значение, он будет использовать StringHttpMessageConverter: значение будет записано непосредственно в тело запроса, а в качестве типа содержимого будет выбрано значение text/plain.

Поскольку в данном случае будут отправляться объекты Student, нам необходим преобразователь сообщений, способный преобразовывать произвольные объекты. Если в библиотеке классов приложения присутствует библиотека Jackson JSON, для записи объектов Student в формате application/json будет использоваться преобразователь MappingJacksonHttpMessageConverter.

4.4 Удаление ресурсов

Когда ресурс становится ненужным, его можно удалить вызовом метода delete() класса RestTemplate. Подобно методу put(), метод delete( ) имеет три версии, сигнатуры которых приводятся ниже:

```
void delete(String url, Object... uriVariables)
        throws RestClientException;
void delete(String url, Map<String, ?> uriVariables)
        throws RestClientException;
void delete(URI url)
        throws RestClientException;
```

Вне всяких сомнений, метод delete() является самым простым из всех методов класса RestTemplate. Единственное, что требуется передать ему, – это URI удаляемого ресурса. Например, чтобы удалить объект Student с указанным идентификатором, достаточно вызвать метод delete(), как показано ниже:

```
public void deleteStudent(long id) {
    try {
        restTemplate.delete(
            new URI("http://localhost:8080/University/students/" +
id));
    } catch (URISyntaxException wontHappen) { }
}
```

Достаточно просто, но здесь снова приходится использовать операцию конкатенации строк, чтобы создать объект URI. При этом конструктор может возбудить контролируемое исключение URISyntax-Exception, что вынуждает нас перехватывать его. Поэтому попробуем воспользоваться более простой версией метода delete(), чтобы избавиться от неудобств:

```
public void deleteStudent(long id) {
    restTemplate.delete("http://localhost:8080/University/students/{id}",
id));
}
```

Так намного лучше. Не находите?

Теперь, после знакомства с набором наиболее простых методов класса RestTemplate, перейдем к знакомству с более сложными методами, поддерживающими запросы HTTP POST.

4.5. Создание новых ресурсов

Можно заметить, что класс RestTemplate включает три разных метода для выполнения POST-запросов. Если умножить это число на три (по количеству версий каждого метода), получится девять методов, посылающих данные на сервер методом POST.

Имена двух из этих методов покажутся вам знакомыми. Методы postForObject() и postForEntity() выполняют POST-запросы почти так же, как методы getForObject() и getForEntity() выполняют GET- запросы. Другой метод, getForLocation(), является уникальным в этом отношении.

4.5.1 Прием объектов в ответах на POST-запросы

Представим , что нам необходимо воспользоваться классом RestTemplate, чтобы отправить на сервер новый объект Student. По скольку это совершенно новый объект Student, он пока не известен серверу. Поэтому официально он не является ресурсом REST и не имеет собственного URL. Кроме того, идентификатор его не будет известен клиенту, пока он не будет создан на сервере.

Один из способов отправить новый ресурс на сервер заключает- ся в использовании метода postForObject( ) класса RestTemplate. Метод postForObject() имеет три версии со следующими сигнатурами:

<T> T postForObject(URI url, Object request, Class<T> responseType)
throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType, Object... uriVariables)
throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType, Map<String, ?> uriVariables)
throws RestClientException;

Во всех версиях в первом параметре передается адрес URL, куда должен быть отправлен ресурс, во втором параметре – отправляемый объект, в третьем – Java-тип объекта, который, как ожидается, должен быть возвращен обратно. В версиях, которые принимают URL в виде строки, четвертый параметр определяет значения переменных-заполнителей в шаблоне URL (либо в виде списка аргументов переменной длины, либо в виде отображения Map).

В приложении University новые ресурсы Student должны отправляться по адресу http://localhost:8080/University/students, где находится метод-обработчик, сохраняющий объекты. Поскольку этот URL не содержит переменных, можно использовать любую версию метода postForObject(). Но в интересах простоты и чтобы избежать необходимости перехватывать исключения, которые могут возбуждаться при конструировании нового объекта URI, мы реализуем эту операцию, как показано ниже:

```
public Student postStudentForObject(Student student) {
RestTemplate rest = new RestTemplate();
return rest.postForObject("http://localhost:8080/University/students",
        student, Student.class);
}
```

Метод postStudentForObject() получает новый объект Student и с помощью postForObject() отправляет его на сервер. В ответ он получает объект Student и возвращает его вызывающей программе.

Как и при использовании метода getForObject(), может возникнуть потребность исследовать некоторые метаданные, поступающие вмес те с ответом. В этом случае предпочтительнее будет использовать метод postForEntity(). Метод postForEntity() имеет три версии, сигнатуры которых являются почти точным отражением сигнатур версий метода postForObject( ):

```
<T> ResponseEntity<T> postForEntity(URI url, Object request, Class<T>
        responseType) throws RestClientException;
<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Object... uriVariables)
        throws RestClientException;
<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Map<String, ?> uriVariables)
        throws RestClientException;
```

Предположим, что помимо возвращаемого обратно ресурса Student необходимо также извлечь из ответа значение заголовка Location. В этом случае можно воспользоваться методом postForEntity(), как показано ниже:

```
RestTemplate rest = new RestTemplate();
ResponseEntity<Student> response = rest.postForEntity(
"http://localhost:8080/University/students", student, Student.class);
Student student = response.getBody();
URI url = response.getHeaders().getLocation();
```

Подобно методу getForEntity(), метод postForEntity() возвращает объект ResponseEntity<T>. Извлечь ресурс (в данном случае – объект Student) из этого объекта можно с помощью его метода getBody(). А с помощью метода getHeaders() можно извлечь объект HttpHeaders и уже с его помощью получить доступ к различным HTTP-заголовкам ответа. В примере выше значение заголовка Location извлекается вызовом метода getLocation(), который возвращает его в виде объекта java.net.URI.

4.5.2 Прием местоположения ресурса после выполнения POST-запроса

Метод postForEntity() удобно использовать для приема отправленного ресурса и заголовков ответа. Но часто бывает так, что нет необходимости

принимать отправленный ресурс обратно (в конце концов, этот ресурс уже имеется в приложении). Если значение заголовка Location – это все, что представляет интерес, тогда проще будет воспользоваться методом postForLocation() класса RestTemplate.

Подобно другим методам, выполняющим POST-запросы, метод postForLocation( ) отправляет ресурс на сервер в теле POST-запроса. Но вместо объекта ресурса метод postForLocation() возвращает местоположение вновь созданного ресурса. Три версии этого метода имеют следующие сигнатуры:

```
URI postForLocation(String url, Object request, Object... uriVariables)
        throws RestClientException;
URI postForLocation( String url, Object request, Map<String, ?> uriVariables)
        throws RestClientException;
URI postForLocation(URI url, Object request)
        throws RestClientException;
```

Для демонстрации метода postForLocation() попробуем реализовать отправку объекта Student еще раз. На этот раз потребуем вернуть URL ресурса:

```
public String postStudent(Student student) {
RestTemplate rest = new RestTemplate();
 return rest.postForLocation("http://localhost:8080/Student/student",
                        student).toString();
}
```

Здесь адрес URL передается в виде строки (в данном случае шаблон URL не содержит переменных-заполнителей). Если после создания ресурса сервер вернет его URL в заголовке Location ответа, то метод postForLocation() вернет этот URL в виде строки.

5 Отправка форм в стиле RESTful

Так как тема оказалась и без того перегруженной, то, при желании, для ознакомления с темой «Отправка форм в стиле RESTful» вы можете обратиться к преподавателю за дополнительным материалом.

6 В заключение

Архитектура RESTful позволяет интегрировать приложения, основываясь на веб-стандартах, сохраняя взаимодействия простыми и естественными. Ресурсы в системе идентифицируются адресами URL, управляются с помощью методов HTTP и отображаются в форму, наиболее соответствующую требованиям клиента.

В этой теме вы узнали, как писать контроллеры Spring MVC, обрабатывающие запросы управления ресурсами RESTful. Используя шаблоны параметризованных адресов URL и связывая методы контроллеров с

конкретными методами HTTP, можно обеспечить обработку запросов GET, POST, PUT и DELETE на выполнение операций с ресурсами приложения.

В ответ на запросы фреймворк Spring может обеспечить представление данных из ресурсов в формате, наиболее соответствующем требованиям клиента. При использовании представлений можно воспользоваться услугами арбитра представлений ContentNegotiatingViewResolver, который выберет представление из числа предлагаемых другими арбитрами представлений, лучше всего удовлетворяющее потребности клиента. Кроме того, метод-обработчик контроллера можно отметить аннотацией @ResponseBody, избавившись тем самым от этапа выбора представления, и поручить одному из нескольких преобразователей HTTP-сообщений превратить возвращаемое значение в ответ, который будет отправлен клиенту.

Для поддержки диалога в стиле REST на стороне клиента фреймворк Spring предоставляет класс RestTemplate, обеспечивающий механизм взаимодействия с ресурсами RESTful на основе шаблонов. А если клиентом является браузер, отсутствие поддержки HTTP- методов PUT и DELETE может быть восполнено с помощью сервлет-фильтра HiddenHttpMethodFilter.