РАБОТА С СЕРВЕРОМ HTTP. REST-СЕРВИСЫ. SWAGGER.

JavaScript

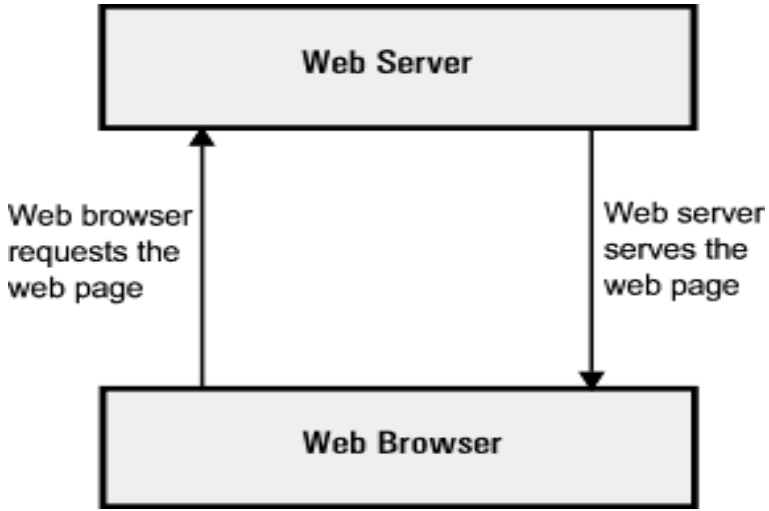
Модуль 5. Урок 1.

LUXOFT

LXFT LISTED NYSE

ВЕБ-СЕРВЕР

- Сервер выполняет экземпляр приложения (ПО), которое способно принимать запросы от клиента и соответствующим образом отвечать на них.
- Серверы работают в архитектуре клиент-сервер.



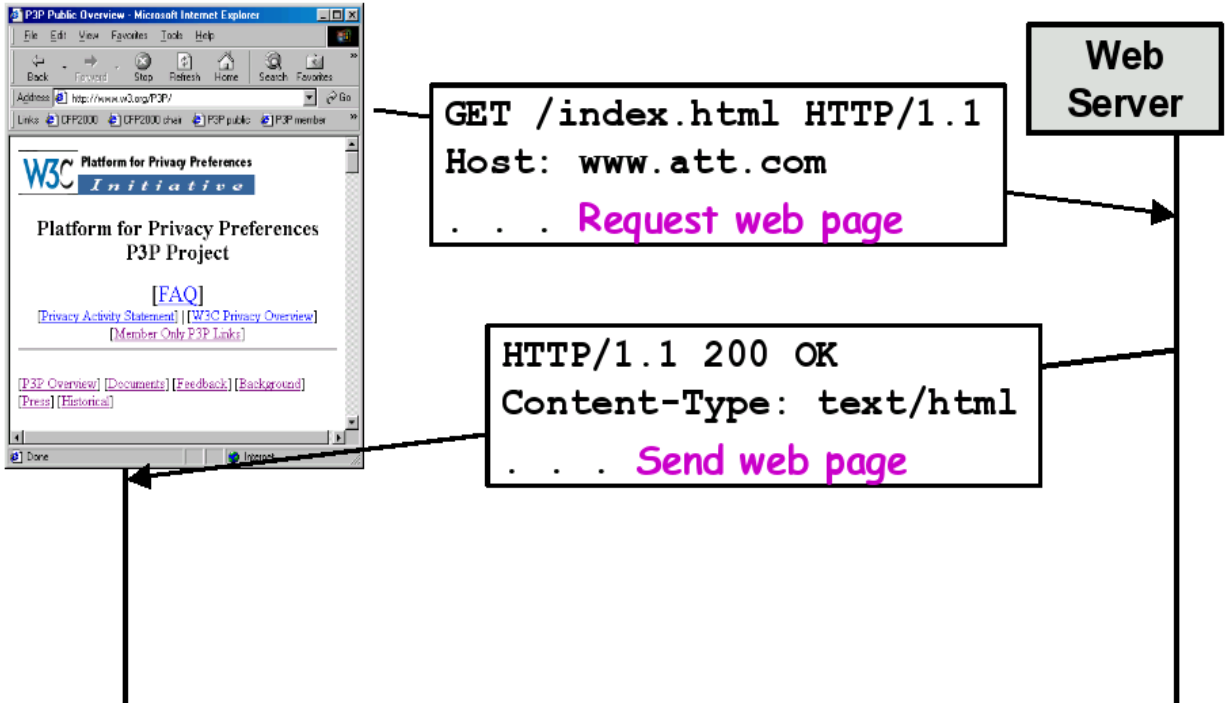
ВЕБ-СЕРВЕР

- Apache (Apache HTTP Server) – бесплатный межплатформенный веб-сервер
- IIS (Internet Information Services) – веб-сервер, созданный Microsoft для использования с семейством продуктов Windows
- Nginx (engine-x) – обратный прокси-сервер, использующий протокол HTTP с открытым исходным кодом
- Tomcat (Apache Tomcat) – популярный Java-сервер

ПРОТОКОЛ HTTP

- HyperText Transfer Protocol
- Работает по принципу «запрос-ответ»
- Содержит информацию о запрашиваемом ресурсе (странице, файле, изображении)
- Содержит некоторую информацию об отправителе (версия браузера, предпочтительный способ кодировки и т. д.)
- Содержит информацию о методе HTTP, т. е. о запрашиваемом действии

ПРОТОКОЛ HTTP



МЕТОДЫ

- **GET** – запрашивает представление указанного ресурса. Данные отправляются в виде URL-переменных и в ограниченном объеме (2–8 Кбайт).

  /path/resource?param1=value1&param2=value2

- **POST** – запрашивает отправку данных. Данные отправляются в теле запроса без ограничений по объему.

  /path/resource

МЕТОДЫ

- **OPTIONS** – возвращает методы HTTP, поддерживаемые сервером для указанного URL.
- **HEAD** – запрашивает ответ, идентичный тому, который соответствует запросу GET, но без тела ответа.
- **PUT** – запрашивает сохранение вложенной сущности в предоставленном идентификаторе URI.
- **PATCH** – применяет частичные изменения к ресурсу.
- **DELETE** – удаляет указанный ресурс.
- **TRACE** – возвращает отправителю полученный запрос.
- **CONNECT** – конвертирует запрос соединения в прозрачный туннель TCP/IP.

КОДЫ СОСТОЯНИЯ ПРОТОКОЛА HTTP

- HTTP всегда возвращает код состояния в ответе.
- Код состояния — основная информация о результатах обработки запроса.
    - **200** – OK
    - **500** – Internal Server Error (Внутренняя ошибка сервера)
    - **403** – Forbidden (Недопустимый)
    - **404** – Not Found (Не найдено)
    - **405** – Method Not Allowed (Недопустимый метод)

ФОРМЫ HTML

- Доступ к форме и элементам формы
- Определение элементов формы
- Общие свойства элементов формы
- Элементы формы

ПРИМЕР ФОРМЫ

```
<form action="/employees" method="POST">
   <input id="nameSearch" placeholder="Имя" size="30">
   <input id="surnameSearch" placeholder="Фамилия" size="30">

   Поиск по менеджеру:
   <select id="managerSearch"></select>

   <p>
      <input type="submit" value="Отправить">
      <input type="reset" value="Сбросить форму">
   </p>
</form>
```

По умолчанию форма отправляется по протоколу POST на адрес, который указан в Action.

Если нужно отправлять по другому протоколу, используется атрибут method:
method="GET"

ПРИМЕР ФОРМЫ ДЛЯ ОТПРАВКИ ЧЕРЕЗ AJAX

```
<form>
  <input id="nameSearch" placeholder="Имя" size="30">
  <input id="surnameSearch" placeholder="Фамилия" size="30">

  Поиск по менеджеру:
  <select id="managerSearch"></select>

  <p>
    <button type=button onclick="searchEmployeeUI()"
        id="searchEmployeesButton">Найти сотрудников</button>
    <input type="reset" value="Сбросить форму">
  </p>
</form>
```

В AJAX-форме нет необходимости указывать action, т.к. форма все равно отправляется с использованием JavaScript

```
function searchEmployeeUI() {
  const name = document.getElementById("nameSearch").value;
  const surname = document.getElementById("surnameSearch").value;
  const managerId = document.getElementById("managerSearch").value;

  const request = { name, surname, managerId} ;

  fetch('/employees/search', {
    method: 'post',
    headers: {
     'Accept': 'application/json',
     'Content-Type': 'application/json'
    },
    body: JSON.stringify(request)
  }).then(res=>res.json())
  .then(res => console.log(res));}
}
```

Функция отвечает за сбор данных с формы и последующую его обработку.
Например, мы можем сконструировать JSON и использовать метод fetch для отправки данных на сервер.

АРХИТЕКТУРА REST

Реализация Web-сервисов REST следует четырем базовым принципам проектирования:

- Явное использование HTTP-методов
- Несохранение состояния
- Предоставление URI, аналогичных структуре каталогов
- Передача данных в XML, JavaScript Object Notation (JSON) или в обоих форматах

REST И CRUD-ОПЕРАЦИИ

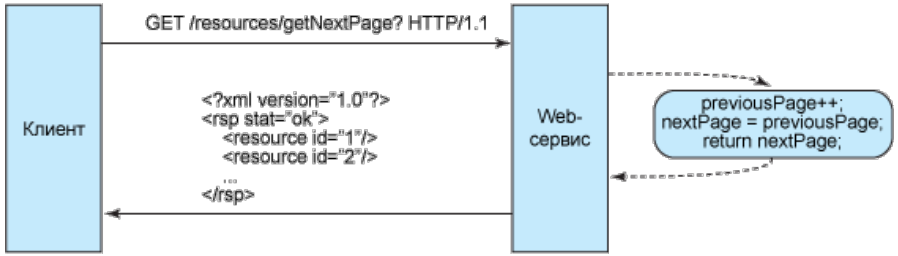
Основной принцип проектирования REST устанавливает однозначное соответствие между операциями create, read, update и delete (CRUD) и HTTP-методами.

Согласно этому соответствию:

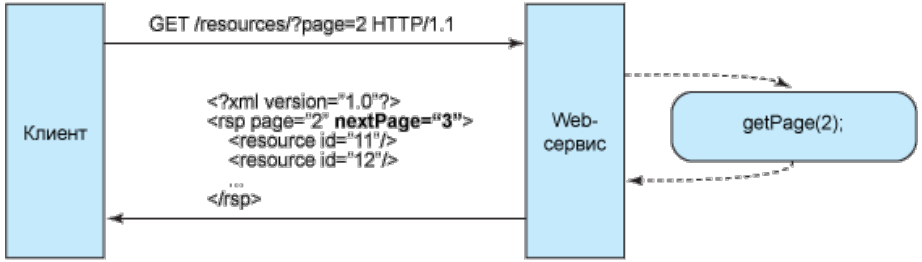
- Для создания ресурса на сервере используется POST
- Для извлечения ресурса используется GET
- Для изменения состояния ресурса или его обновления используется PUT
- Для удаления ресурса используется DELETE

НЕСОХРАНЕНИЕ СОСТОЯНИЯ

Для удовлетворения постоянно растущих требований к производительности Web-сервисы REST должны быть масштабируемыми. Это может быть достигнуто только при использовании stateless-сервисов (без сохранения состояния).



Сохранение состояния: сервис помнит о клиенте и о том, какую страницу клиент запрашивал последней



Нет сохранения состояния: вся необходимая информация передается каждый раз

НЕСОХРАНЕНИЕ СОСТОЯНИЯ И МАСШТАБИРУЕМОСТЬ

Клиент отправляет полные запросы, которые могут обрабатываться независимо от других запросов.

Это требует от клиентского приложения использования в полном объеме HTTP-заголовков, определенных интерфейсом Web-сервиса, и отправки полных представлений ресурсов в теле запроса.

Клиентское приложение отправляет запросы, которые практически ничего не знают о

- предшествующих запросах,
- существовании сеанса на сервере,
- способности сервера добавлять контекст в запрос
- состоянии приложения, сохраняющемся между запросами

Результат – отличная масштабируемость – клиенту все равно, с каким сервером работать:

ОПЕРАЦИИ CRUD И REST-СЕРВИСЫ

Операция:	Create	Read	Update	Delete
HTTP Метод:	POST	GET	PUT	DELETE
/records	Создать новую запись	Получить список всех записей	-	-
/records/5	-	Получить данные о записи 5	Обновить данные записи 5	Удалить запись 5

В данном примере наглядно показана структура URL которую следует наследовать:
/:collection/
/:collection/:id

В первом случае мы будем манипулировать набором элементов, во втором — какой-то конкретной записью под неким ID.

REST: ПРИМЕР API ТЕЛЕФОННОЙ КНИГИ

URI	HTTP Method	Описание
/rest/phones	GET	Возвращает всю телефонную книгу в формате JSON
/rest/phones/{id}	GET	Возвращает информацию о контакте по id
/rest/phones/name/{name}	GET	Возвращает информацию о контакте по имени
/rest/phones	POST	Добавляет новый контакт
/rest/phones/{id}	DELETE	Удаляет контакт по id
/rest/phones/{id}	PUT	Обновляет данные контакта

ПРИМЕР: СПИСОК ПОЛЬЗОВАТЕЛЕЙ

Пусть у нас есть некий ресурс, содержащий список пользователей:

id	firstName	lastName
1	Ivan	Ivanov
2	Petr	Petrov
3	Sidr	Sidorov
..	..	..

Чтение всех записей:

>> GET /users/

<< HTTP/1.1 200 OK

```
[
  {id:1, firstName:"Ivan", lastName:"Ivanov"},
  {id:2, firstName:"Petr", lastName:"Petrov"},
  {id:3, firstName:"Sidr", lastName:"Sidorov"}
]
```

Чтение одной записи:

>> GET /users/2

<< HTTP/1.1 200 OK

{id:2, firstName:"Petr", lastName:"Petrov"}

Запись не найдена:

>> GET /users/42

<< HTTP/1.1 404 Not Found

ПОЛУЧЕНИЕ ЧАСТИ ЗАПИСЕЙ

Количество записей в базе данных может быть неподъёмным для одного запроса, так что лучше их разбивать на страницы, но для реализации постраничной навигации никакого стандарта не принято.

Вот пример получения части записей:

>> GET /users/?offset=0&limit=2

```
<< HTTP/1.1 200 OK
{
  rows:[
    {id:1, firstName:"Ivan", lastName:"Ivanov"},
    {id:2, firstName:"Petr", lastName:"Petrov"}
  ],
  meta:{
    total:3,
    offset:0,
    limit:2
  }
}
```

СОЗДАНИЕ ЗАПИСИ

Создание новой записи – отправка данных **формы**

>> POST /users/

firstName=Petrik&lastName=Petrenko

<< HTTP/1.1 201 Created

<< Location: /users/4

Создание новой записи – отправка **JSON**:

>> POST /users/

>> Content-Type:application/json

{"firstName":"Petrik","lastName":"Petrenko"}

ИЗМЕНЕНИЕ И УДАЛЕНИЕ ЗАПИСИ

Для **изменения** данных RESTful предполагает использования двух методов **PUT** и **PATCH**, различия в них лишь в том, что **PUT** предполагает **замену записи полностью**, а **PATCH** должен **обновлять** лишь данные, которые пришли в запросе.

>> PUT /users/2

>> Content-Type:application/json

{"firstName":"Petr","lastName":"Petrenko"}

<< HTTP/1.1 200 OK

>> PATCH /users/2

lastName=Petrov

<< HTTP/1.1 200 OK

Удаление записи – DELETE:

>> DELETE /users/3

<< HTTP/1.1 204 No Content

ФОРМАТ ВОЗВРАЩАЕМЫХ ДАННЫХ: XML, JSON, XHTML

Клиентское приложение может запросить данные в разных форматах:

MIME-тип	Тип содержимого
JSON	application/json
XML	application/xml
XHTML	application/xhtml+xml

Чаще всего используют JSON формат как наиболее простой и достаточно популярный

Формат можно задать, передав HTTP-заголовок Accept.
Но нужно, чтобы сервер поддерживал запрошенный формат

>> GET /users
>> Accept: application/json

Запрос формата, который не поддерживается:

>> GET /users
>> Accept: application/xhtml+xml
<< HTTP/1.1 406 Not Acceptable
<< Content-Type:application/json
{"accept": ["application/json", "application/javascript"]}

ФУНКЦИЯ FETCH

```
let promise = fetch(url[, options]);
```

```
fetch('/user/1')
  .then(function(response) {
    console.log(response.headers.get('Content-Type'));
    // application/json; charset=utf-8
    console.log(response.status); // 200

    return response.json();
  })
  .then(function(user) {
    console.log(user.name); // Ivan
  })
  .catch( console.log );
```

Объект **response** кроме доступа к заголовкам **headers**, статусу **status** и некоторым другим полям ответа, даёт возможность прочитать его тело, в желаемом формате.

Варианты включают в себя:

response.arrayBuffer()
response.blob()
response.formData()
response.json()
response.text()

ФУНКЦИЯ FETCH

let promise = fetch(url[, options]);

url – URL, на который сделать запрос,

options – необязательный объект с настройками запроса.

Свойства **options**:

method – метод запроса,

headers – заголовки запроса (объект),

body – тело запроса: FormData, Blob, строка и т.п.

mode – одно из: «same-origin», «no-cors», «cors», указывает, в каком режиме кросс-доменности предполагается делать запрос.

credentials – одно из: «omit», «same-origin», «include», указывает, пересылать ли куки и заголовки авторизации вместе с запросом.

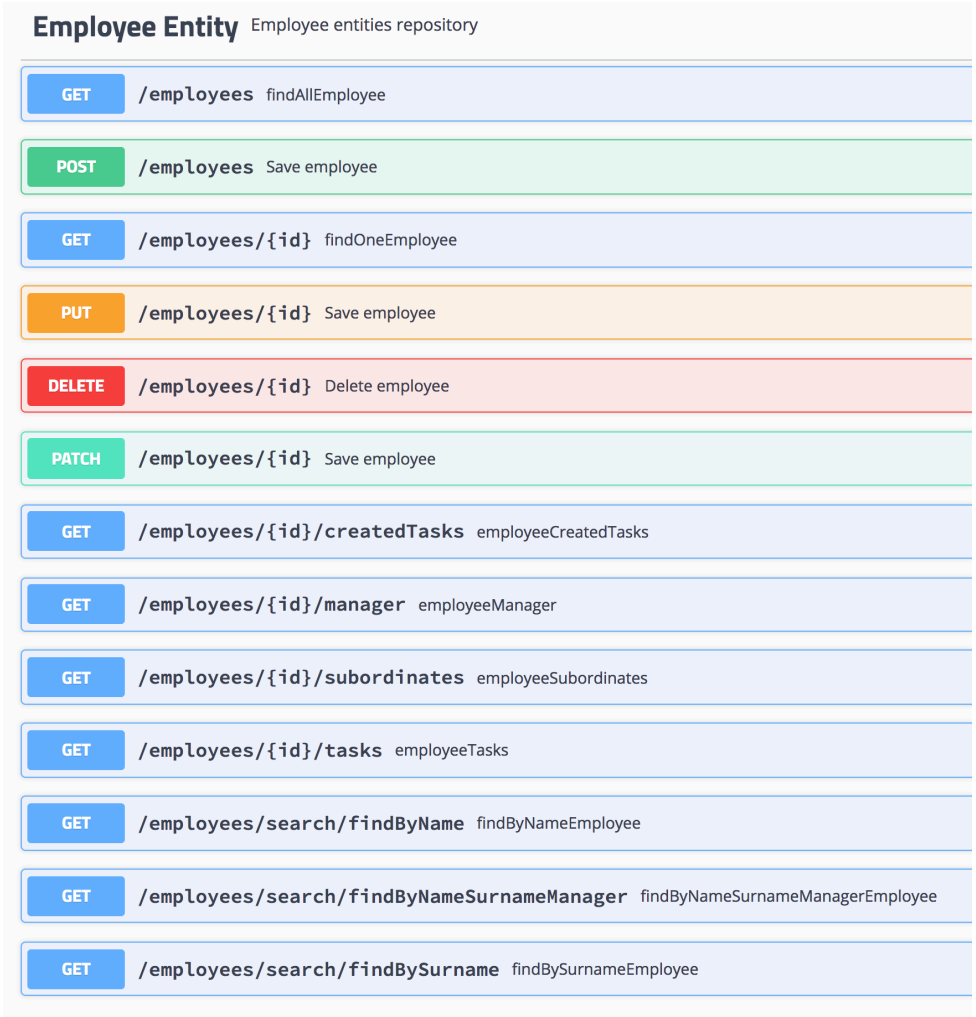
cache – одно из «default», «no-store», «reload», «no-cache», «force-cache», «only-if-cached», указывает, как кешировать запрос.

redirect – можно поставить «follow» для обычного поведения при коде 30x (следовать редиректу) или «error» для интерпретации редиректа как ошибки.

SWAGGER API

Запустив сервер, вы сможете обратиться к системе документирования REST-сервисов:

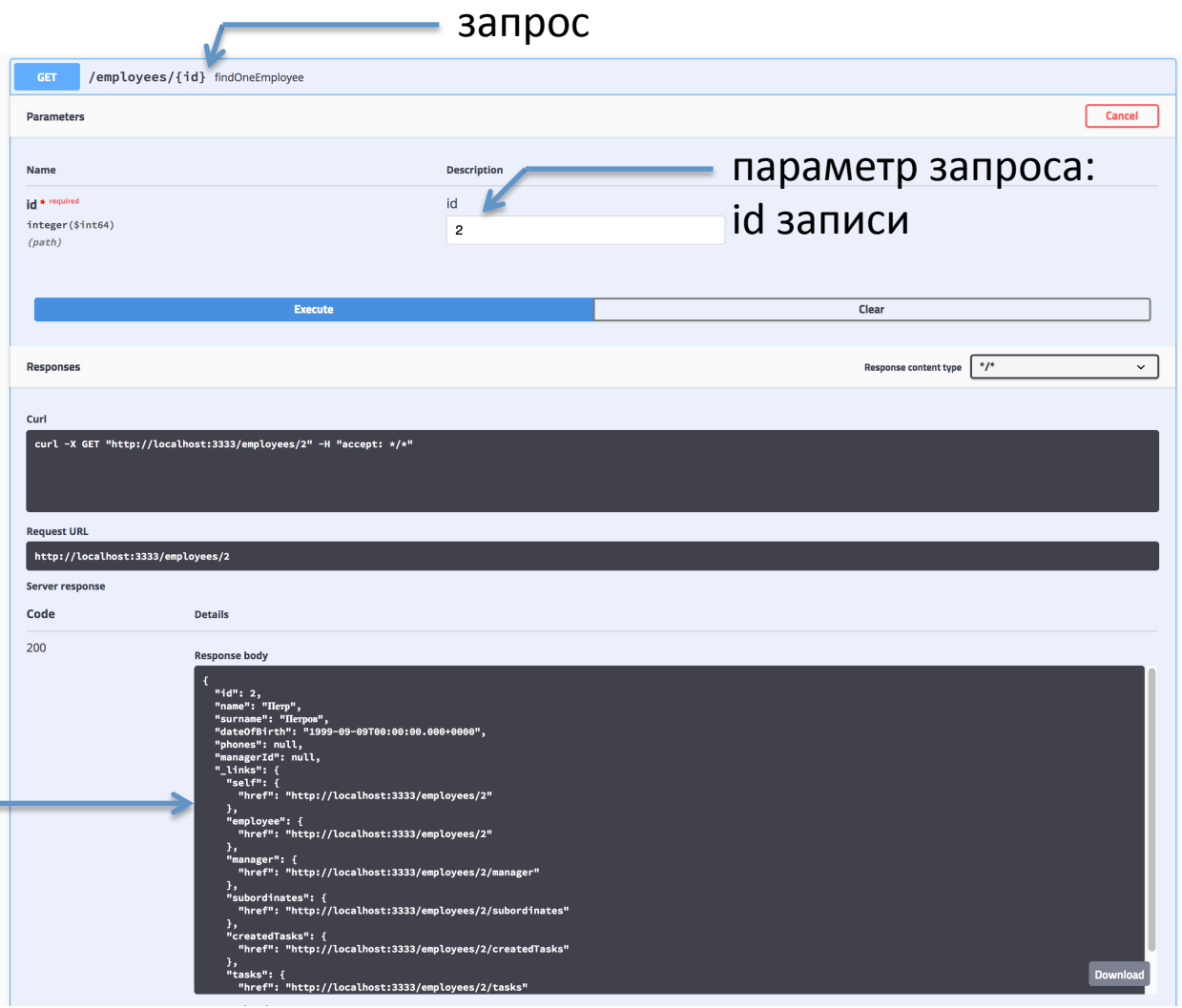
http://localhost:3333/swagger-ui.html



SWAGGER

Вы можете выбрать любой пункт, нажать **Try it out**, ввести недостающие параметры, нажать **Execute** и получить ответ от сервера.

В данном примере мы получаем информацию о сотруднике 2 в формате JSON.



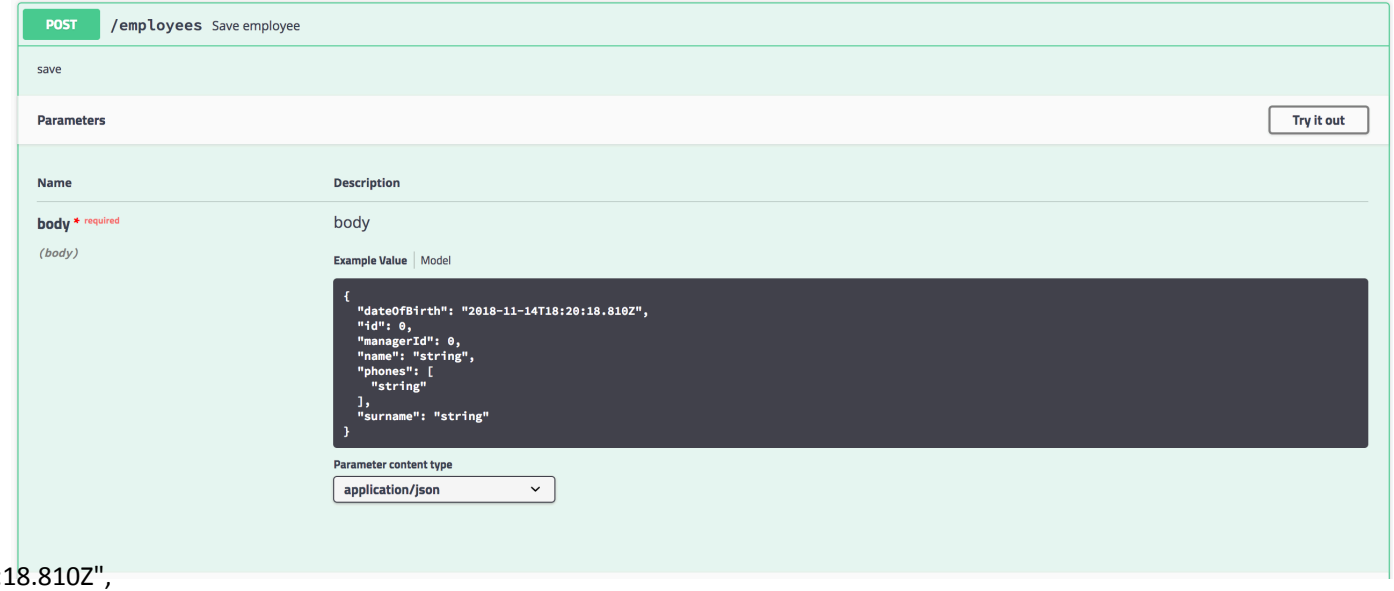
ОТВЕТ ОТ СЕРВЕРА


```
{
  "_embedded": {
  "employees": [
    {
      "id": 2,
      "name": "Петр",
      "surname": "Петров",
      "dateOfBirth": "1999-09-09T00:00:00.000+0000",
      "phones": null,
      "managerId": null,
      "_links": {
          "self": {
            "href": "http://localhost:3333/employees/2"
          },
          "employee": {
            "href": "http://localhost:3333/employees/2"
          },
          "manager": {
            "href": "http://localhost:3333/employees/2/manager"
          },
          "subordinates": {
            "href": "http://localhost:3333/employees/2/subordinates"
          },
          "createdTasks": {
            "href": "http://localhost:3333/employees/2/createdTasks"
          },
          "tasks": {
            "href": "http://localhost:3333/employees/2/tasks"
          }
      }
    },
    {
      "id": 3,
      …
```


HATEOAS-ссылки: что еще можно сделать с ресурсом: URL возможных операций

SWAGGER

Чтобы добавить данные на сервер, можно воспользоваться POST-запросом. Например, здесь мы отправляем на сервер JSON, содержащий информацию о добавляемом сотруднике.



Вот значение-пример:

```
{
  "dateOfBirth": "2018-11-14T18:20:18.810Z",
  "id": 0,
  "managerId": 0,
  "name": "string",
  "phones": [
    "string"
  ],
  "surname": "string"
}
```

Нам нужно заполнить только необходимые данные.
id заполнять не нужно, т.к. он генерируется автоматически.
Вот пример данных:

```
{
  "dateOfBirth": "2000-01-01T18:20:18.810Z",
  "name": "Степан",
  "surname": "Степанчиков"
}
```

SWAGGER

Запрос:

curl -X POST "http://localhost:3333/employees" -H "accept: */*" -H "Content-Type: application/json" -d "{ \"dateOfBirth\": \"2000-01-01T18:20:18.810Z\", \"name\": \"Степан\", \"surname\": \"Степанчиков\"}"

Ответ сервера:

код: 201

тело ответа:


```
{
  "id": 76,
  "name": "Степан",
  "surname": "Степанчиков",
  "dateOfBirth": "2000-01-01T18:20:18.810+0000",
  "phones": [],
  "managerId": null,
```


заголовки ответа:

content-type: application/hal+json;charset=UTF-8
date: Wed, 14 Sep 2018 18:23:44 GMT
location: http://localhost:3333/employees/76
transfer-encoding: chunked


```
"_links": {
  "self": {
    "href": "http://localhost:3333/employees/76"
  },
  "employee": {
    "href": "http://localhost:3333/employees/76"
  },
  "subordinates": {
    "href": "http://localhost:3333/employees/76/subordinates"
  },
  "createdTasks": {
    "href": "http://localhost:3333/employees/76/createdTasks"
  },
  "manager": {
    "href": "http://localhost:3333/employees/76/manager"
  },
  "tasks": {
    "href": "http://localhost:3333/employees/76/tasks"
  }
 }
}
```