Lab Activity 3: RESTful HTTP API

Submit your final results to Brightspace and demonstrate your final results to your Lab Professor for 3 marks (3% of course grade)

See Brightspace for due date.

Objective

Begin learning about RESTful APIs by using Netbeans to implement a RESTful HTTP API for a clone of the address-book Contacts application, and use Postman to issue some HTTP requests to the RESTful API.

Background

There are two main ways to interact with the Enterprise Web Applications we study in this course.

Standard Web App: The first way we have seen is for a human user to interact with their web browser which displays HTML pages resulting from JSF Pages (we will study JSF Pages more later in the course). We have all used many applications this way. The human user looks at the HTML pages, and can read information from the page or enter information on the page, and can click buttons, and so on. HTTP packets (requests/responses) are the mechanism by which information travels between the browser and server.

Web Service: The second way, which is the subject of this lab exercise, is for someone else's computer program to directly interact with our application through a RESTful HTTP interface for our application. In that case we can say we are providing a Web Service. The computer program that uses our web service might be a native application running on a mobile device like a phone, or the computer program that uses our web service might be a large backend server at a bank or insurance company. The computer program performs operations with our web application (service) by sending HTTP requests to URLs, and receiving HTTP responses. There is no need for JSF pages or HTML pages because a computer program does not have eyes to look at pages. For a computer program, information is represented in JSON or XML which can be parsed by the computer directly instead of being rendered in a field on a page for a human's eyes to see.

REST Resources: Different URLs of a RESTful API represent different resources of the application. One type of resource in the address-book application is the set of all contacts in the database, another type of resource is a subset of contacts within a certain range of primary key ids, and yet another type of resource is a single contact with a certain specific primary key id. There are URLs to represent each of those resources, which are different groupings of contacts.

REST Methods: The action that is taken on a resource (when the API receives an HTTP request for the resource's URL) is given by the particular HTTP method in the HTTP request, such as GET, PUT, POST, or DELETE. The body of the HTTP requests, usually formatted as JSON or XML content, represents information necessary to carry out the request, and the body of the HTTP response similarly represents the results of the request.

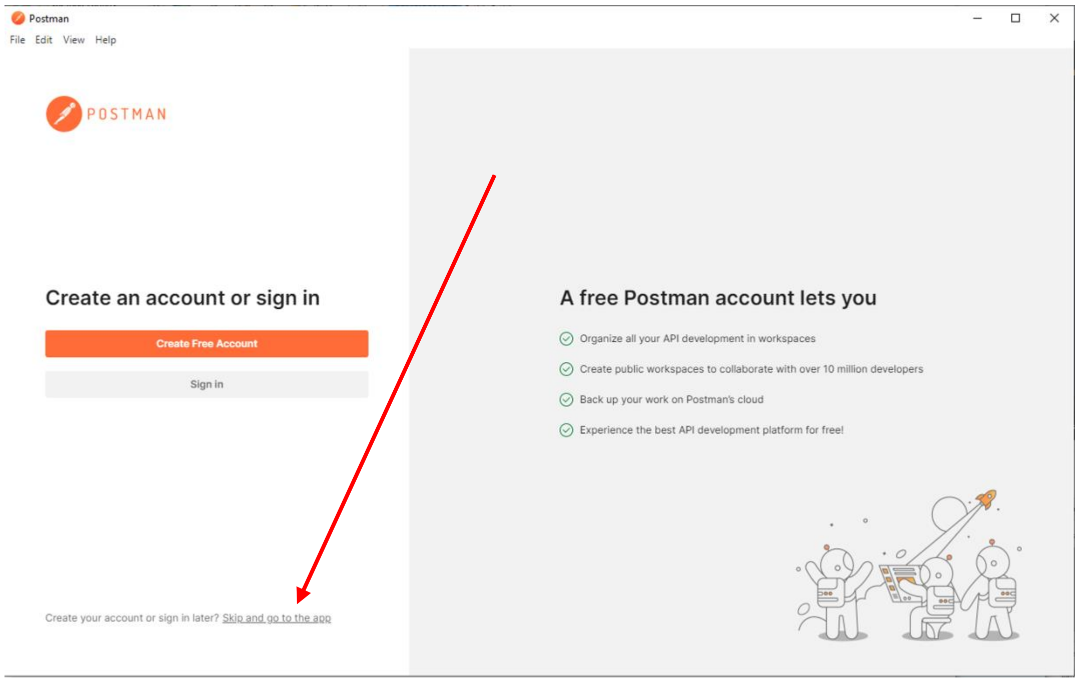
Given a project with an entity like the Contact entity, with a right-click, Netbeans can add a RESTful HTTP API to manipulate (for example POST, PUT, GET, DELETE) the following resources: the CONTACT table, a row in the contact table, a range of rows in the contact table, or the count of the number of rows in the contact table. Not all methods make sense for every resource, as it only makes sense to GET the count of the number of rows, for example.

Instructions

1. Start with a new empty web project, as we did in Lab1 and Lab2. These empty web projects are not quite empty: there is a simple `index.html` that displays Hello World! when you run the project.
2. Create an Entity Class From Database in the new project using the CONTACT table in the same database your address-book application is using. Remember to make the **id** field auto-generate. At this point, when you deploy and run the application on your Glassfish server, you should see the same result as Step 1 above. In this lab exercise, we will not develop the human-readable pages of the web application any further, so it will always give the simple index.html when someone uses it as a normal web application with a web browser. However, our empty project now has the beginnings of a RESTful HTTP API:
    a. Notice that Netbeans created `Contact.java` (the Entity class we asked Netbeans to create) and Netbeans also created `JAXRSConfiguration.java` (configures the RESTful HTTP interface for the application). Open the `JAXRSConfiguration.java` file and find the `@ApplicationPath` annotation (and refer to iii below). The URLs for the RESTFul API resources will be in the format, http://**ServerIP**:8080/**ProjectName**/**ApplicationPath**/**Resource** where in my case,
        i. **ServerIP** is `localhost` (because my Glassfish server is running at localhost)
        ii. **ProjectName** is `mavenproject3` (I accepted the default project name when I used Netbeans to create the new project)
        iii. **ApplicationPath** is `resources` (as specified by the `@ApplicationPath` annotation)
        iv. **Resource** is the resource to be operated on (for example, the contacts table, or one contact – more on resources below)
    b. Notice that Netbeans created a file called `JavaEE8Resource.java`, which defines a resource with a `@Path` annotation, called `javaee8`. Open the `JavaEE8Resource.java` file, and find the method with the `@GET` annotation. The `@Get` annotation tells the system which runs when an HTTP GET request is received for the URL that has the `@Path` (`javaee8`) in the **Resource** position. In my case, that URL is `http://localhost:8080/mavenproject3/resources/javaee8` Your case should be similar, except your project name might be different from `mavenproject3`.
    c. Notice that Points a. and b. above mean that we already have a simple RESTful HTTP API in this new application. When a computer program issues an HTTP GET request for URL `http://localhost:8080/mavenproject3/resources/javaee8` the web application uses the `ping` method to build the HTTP Response, which contains the string, `ping`. Assuming you have run your application, you can use your web browser to create an HTTP GET request to that URL (in other words, use your web browser to browse to that URL, and that will cause your browser to send an HTTP GET request for the URL), and your web browser in this case should be able to show you the String that

came in the response from your RESTful API. Try that with your web browser, then congratulations, you have a simple RESTful API (that has only the one resource URL and can support only GET on that URL), and you used your web browser to test your API. We'll find later in this lab exercise that Postman is a better program for testing our HTTP APIs because it supports building and sending arbitrary HTTP requests, and examining the responses in detail.

3. In Step 1 above, we gave the project no real user interface as a web application. In that state, there are no JSF pages and no Controller named bean (no `ContactController` class). On the API side, in Step 2c above, we looked at the simple "hello-world-equivalent" RESTful HTTP API that Netbeans created for us when we created the `Contact` Entity in the project. Unlike the web application which is meant to be used by humans with a web browser, the API is meant to be used by machines. So far the API can support only a GET request to a `javaee8` resource, and the response is always that associated string. At this point we do have a `Contact` Entity class, but no `ContactFacade` class. Can you find a `persistence.xml` file? Configure `persistence.xml` so that this new project uses the same database that your address-book application uses, and configure it to NOT drop the tables.
4. Create the default Netbeans RESTful HTTP API for performing CRUD operations on the `Contact` class in your new project: select the project and right click `New->RESTful Web Services From Entity Class`, and go through the wizard. Build, Deploy, and Run your project, and Netbeans will load the web application in your browser (the web app still shows the hello-world page, but the RESTful API now has more capabilities).
5. Inspect the changes Netbeans made to your project in Step 4 above. You'll notice there is a `ContactFacadeREST.java`. Open this file, and look for the @Path annotation on the root resource (the class), and a method with a @GET annotation but no additional @Path annotation, as we did in Step 2b. This @GET annotation without a @Path annotation applies to the root resource, the class, which has a @Path annotation. Can you construct the URL for the resource given by the @Path annotation as we did in Step 2b? Can you determine from looking at the method with the @GET annotation what will happen when you send an HTTP GET request to that URL (supposed to be done by a machine, but you can browse to that URL with your web browser to issue a GET method as a quick test).
6. We will study details of how the RESTful API works, and how to program enhancements to it, in a future assignment. The remainder of this lab exercise involves getting ready to test a RESTful HTTP API once we begin making modifications. For testing our API, we will use the Postman API testing application, available for free download here: https://www.getpostman.com/downloads/
7. Install the Postman API testing application, and run it locally **without signing in**. There is a button to download without making an account or signing in. When you run Postman for the first time, you should see a window similar to the screenshot below, where you can click "Skip and go to the app", and on the next screen, "Create a Request":

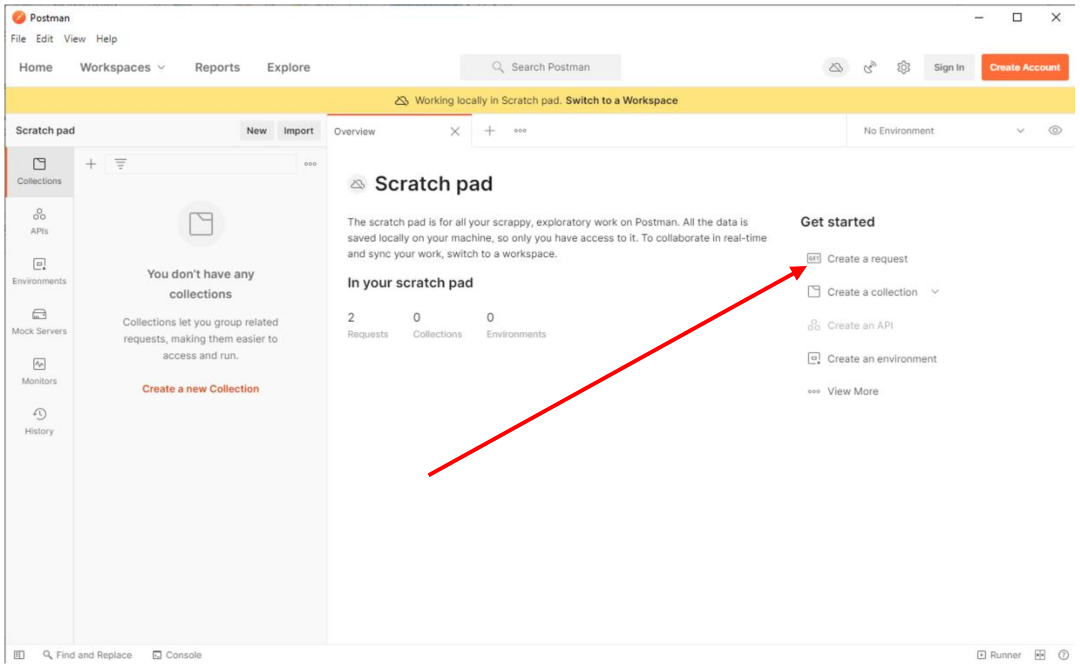


Create an account or sign in

Create your account or sign in later? Skip and go to the app

A free Postman account lets you

- Organize all your API development in workspaces
- Create public workspaces to collaborate with over 10 million developers
- Back up your work on Postman's cloud
- Experience the best API development platform for free!



Working locally in Scratch pad. Switch to a Workspace

You don't have any collections

Collections let you group related requests, making them easier to access and run.

Scratch pad

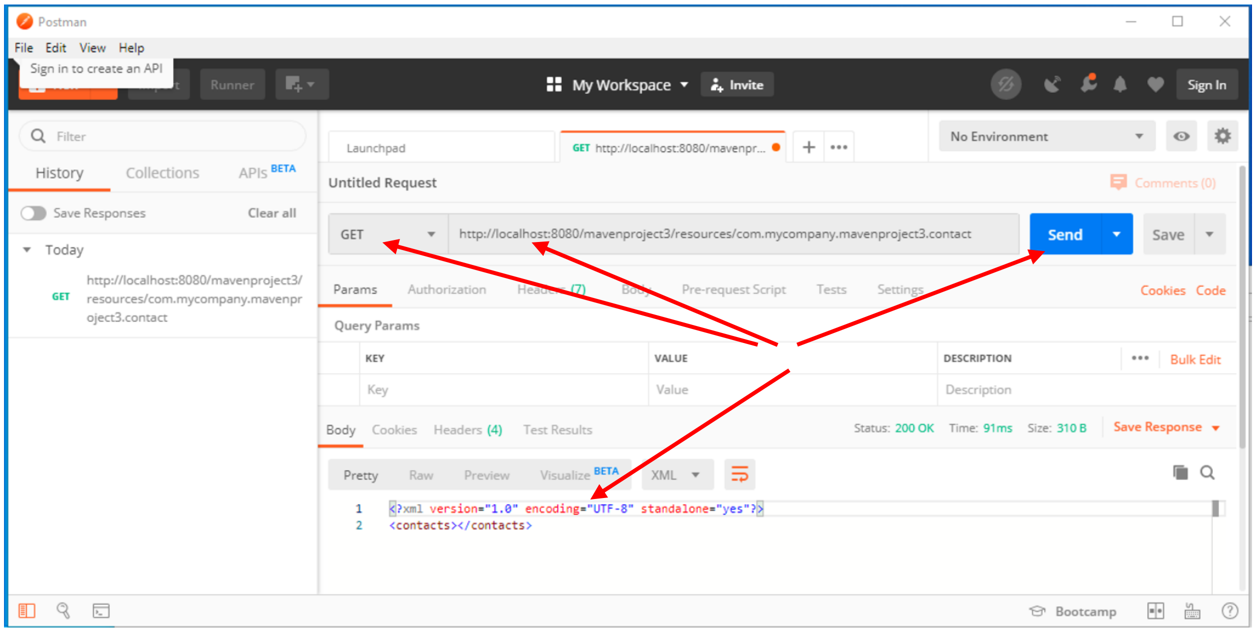
The scratch pad is for all your scrappy, exploratory work on Postman. All the data is saved locally on your machine, so only you have access to it. To collaborate in real-time and sync your work, switch to a workspace.

In your scratch pad

2 Requests 0 Collections 0 Environments

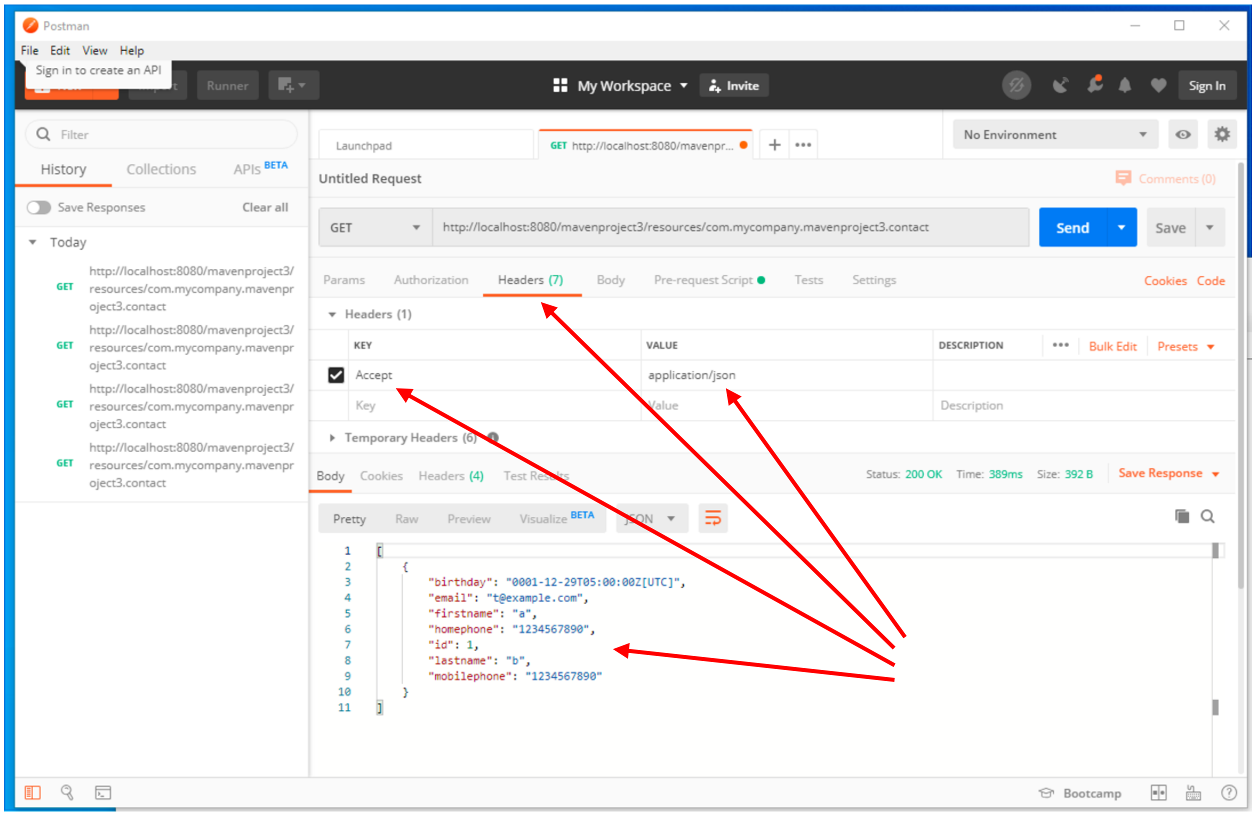
Get started

8. Issue a GET request to the URL for the Contacts resource. In the screenshot below, you can see the response in this case is XML representing an empty list of Contacts:


Postman
File Edit View Help
Sign in to create an API
Runner
My Workspace
Invite
Sign In
Filter
History Collections APIs BETA
Save Responses Clear all
Today
GET http://localhost:8080/mavenproject3/resources/com.mycompany.mavenproject3.contact
Launchpad
GET http://localhost:8080/mavenpr...
No Environment
Untitled Request
Comments (0)
GET
http://localhost:8080/mavenproject3/resources/com.mycompany.mavenproject3.contact
Send
Save
Params Authorization Headers (7) Body Pre-request Script Tests Settings
Cookies Code
Query Params
KEY VALUE DESCRIPTION
Bulk Edit
Key Value Description
Body Cookies Headers (4) Test Results
Status: 200 OK Time: 91ms Size: 310 B
Save Response
Pretty Raw Preview Visualize BETA XML
Bootcamp

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<contacts></contacts>
```

9. Use the address-book application to create at least one contact in your Contacts table, so that you can retrieve an non-empty list of contacts. The following screenshot shows a GET request when a single contact resides in the table. The header of this request had "accept : application/json" , which is why the response shows JSON in this case, instead of XML in Step 8.



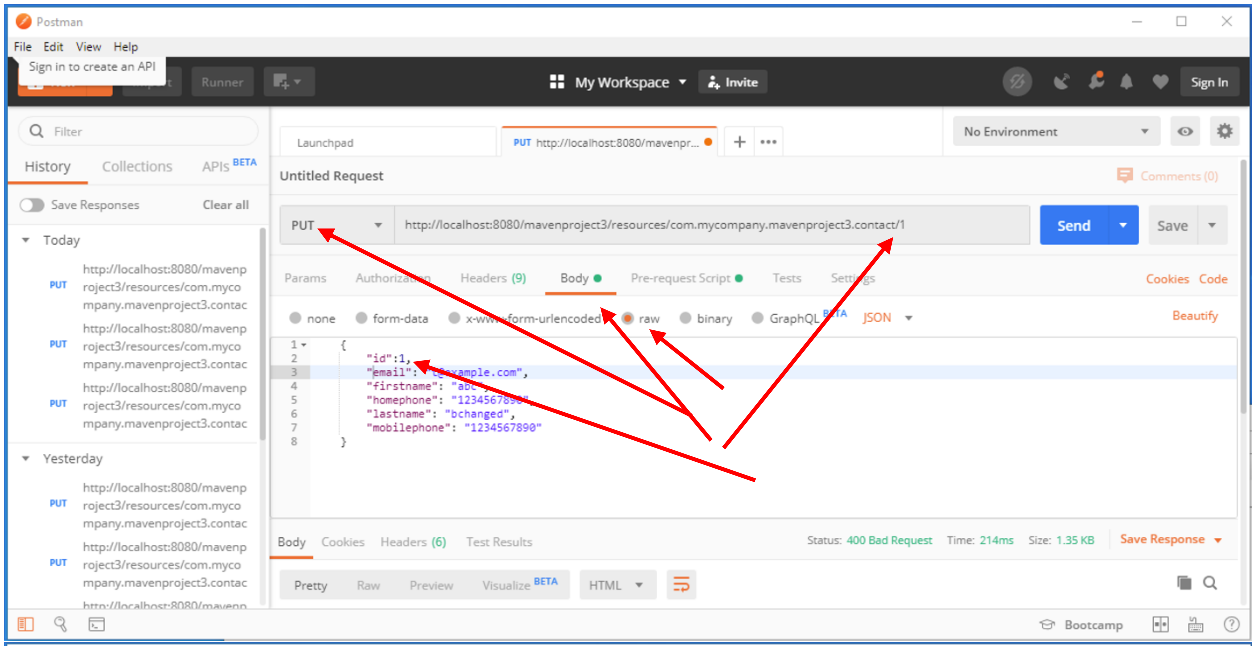
GET http://localhost:8080/mavenproject3/resources/com.mycompany.mavenproject3.contact

KEY	VALUE	DESCRIPTION
✓ Accept	application/json	
Key	Value	Description

Status: 200 OK Time: 389ms Size: 392 B

```
[
    {
        "birthday": "0001-12-29T05:00:00Z[UTC]",
        "email": "t@example.com",
        "firstname": "a",
        "homephone": "1234567890",
        "id": 1,
        "lastname": "b",
        "mobilephone": "1234567890"
    }
]
```

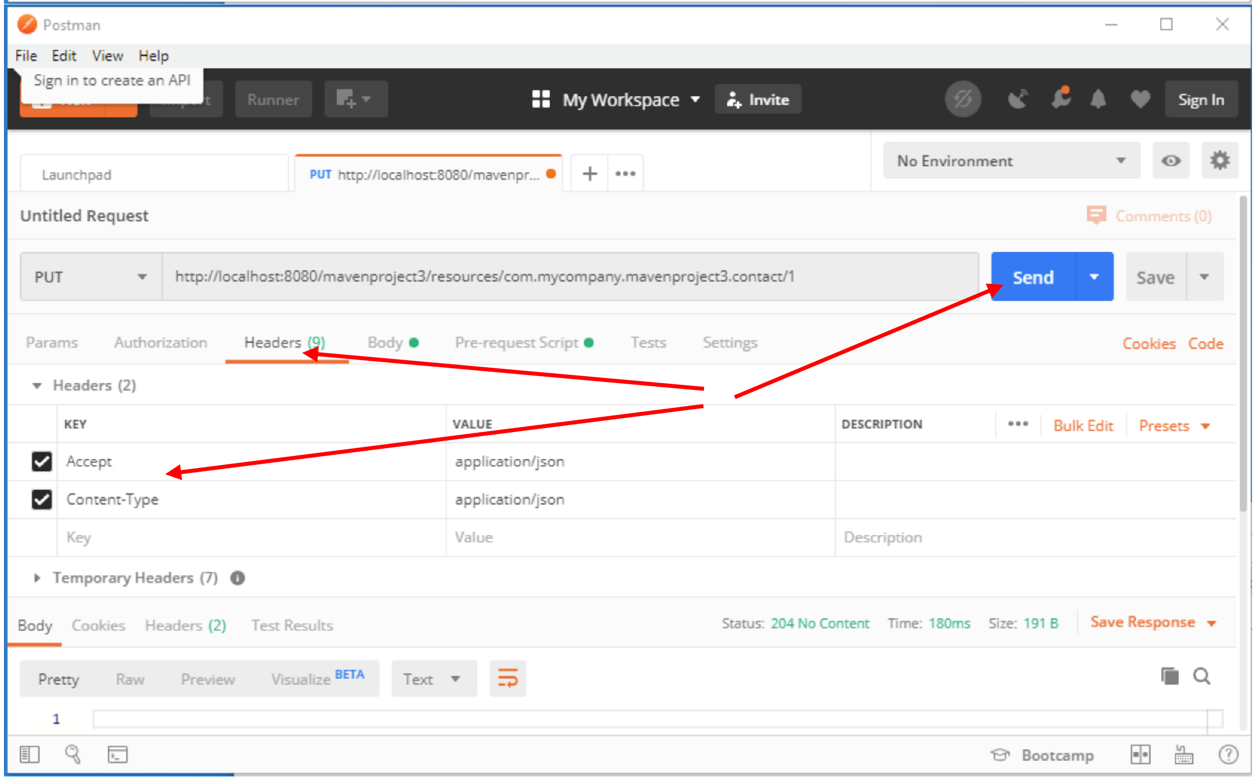
10. Can you work out, with Postman, how to issue a request that will make a change to a contact in the database? Hint: one possibility is to build a HTTP PUT request with a body containing a JSON representation of new fields for changing a contact, and the URL will end with the id of that contact. If we are encoding the body in JSON, we put "Content-type: application/json" in the request header. The following screenshots help show how to do this. Notice that when you're building a new request that has content in its body, you need to use the body input box, which will appear (when you select the body tab) above the response box that contained the response in Step 8.



PUT http://localhost:8080/mavenproject3/resources/com.mycompany.mavenproject3.contact/1

```
{
    "id":1,
    "email": "t@example.com",
    "firstname": "abc",
    "homephone": "1234567890",
    "lastname": "bchanged",
    "mobilephone": "1234567890"
}
```

Status: 400 Bad Request   Time: 214ms   Size: 1.35 KB



PUT http://localhost:8080/mavenproject3/resources/com.mycompany.mavenproject3.contact/1

Headers (2)

KEY	VALUE	DESCRIPTION
✓ Accept	application/json	
✓ Content-Type	application/json	

Status: 204 No Content   Time: 180ms   Size: 191 B

Demonstration

11. Show your Lab Instructor how you can issue a GET request to your API with Postman, and receive a list of all the contacts in the database.
12. Show your Lab Instructor how you can issue a request to your API with Postman that changes data in your contact table (you can decide what change to make, as long as something in the database changes).