Assignment NO: 03

Assignment Name: Controller Rest API

Name: Md Rafiqul Islam

ID: IT17054

Theory: REST has quickly become the de-facto standard for building web services on the web because they're easy to build and easy to consume. There's a much larger discussion to be had about how REST fits in the world of microservices, but - for this tutorial - let's just look at building RESTful services. To register a new REST route, you must specify a number of callback functions to control endpoint behavior such as how a request is fulfilled, how permissions checks are applied, and how the schema for your resource gets generated. While it is possible to declare all of these methods in an ordinary PHP file without any wrapping namespace or class, all functions declared in that manner coexist in the same global scope. If you decide to use a common function name for your endpoint logic like get_items() and another plugin (or another endpoint in your own plugin) also registers a function with that same name, PHP will fail with a fatal error because the function get_items() is being declared twice.

The web and its core protocol, HTTP, provide a stack of features:

- Suitable actions (GET, POST, PUT, DELETE, …)
- Caching
- Redirection and forwarding
- Security (encryption and authentication)

Methodology:

Developers are able to draw upon 3rd party toolkits that implement these diverse specs and instantly have both client and server technology at their fingertips.

So building on top of HTTP, REST APIs provide the means to build flexible APIs that can:

- Support backward compatibility
- Evolvable APIs
- Scaleable services
- Securable services
- A spectrum of stateless to stateful services

Code:

```
package payroll;

import java.util.Objects;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

@Entity
```

```
class Employee {

  private @Id @GeneratedValue Long id;
  private String name;
  private String role;
  Employee() {}
  Employee(String name, String role) {

    this.name = name;
    this.role = role;
  }
  public Long getId() {
    return this.id;
  }
  public String getName() {
    return this.name;
  }
  public String getRole() {
    return this.role;
  }
  public void setId(Long id) {
    this.id = id;
  }
  public void setName(String name) {
    this.name = name;
  }
  public void setRole(String role) {
    this.role = role;
  }
```

```
  @Override
  public boolean equals(Object o) {
    if (this == o)
      return true;
    if (!(o instanceof Employee))
      return false;
    Employee employee = (Employee) o;
    return Objects.equals(this.id, employee.id) && Objects.equals(this.name, employee.name)
        && Objects.equals(this.role, employee.role);
  }

  @Override
  public int hashCode() {
    return Objects.hash(this.id, this.name, this.role);
  }

  @Override
  public String toString() {
    return "Employee{" + "id=" + this.id + ", name='" + this.name + '\'' + ", role='" + this.role
+ '\'' + '}';
  }
}
```

```
package payroll;
import org.springframework.data.jpa.repository.JpaRepository;
interface EmployeeRepository extends JpaRepository<Employee, Long> {}
```

Exercise/Output:

Test the Service:

visit http://localhost:8080/greeting, where you should see:

```
{"id":1,"content":"Hello, World!"}
```

Provide a name query string parameter by visiting http://localhost:8080/greeting?name=User. Notice how the value of the content attribute changes from Hello, World! to Hello, User!, as the following listing shows:

```
{"id":2,"content":"Hello, User!"}
```

This change demonstrates that the @RequestParam arrangement in GreetingController is working as expected. The name parameter has been given a default value of World but can be explicitly overridden through the query string.

Notice also how the id attribute has changed from 1 to 2. This proves that you are working against the same GreetingController instance across multiple requests and that its counter field is being incremented on each call as expected.

```
$ curl -v -X PUT localhost:8080/orders/4/complete

* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> PUT /orders/4/complete HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 405
< Content-Type: application/problem+json
< Transfer-Encoding: chunked
< Date: Mon, 27 Aug 2018 15:05:40 GMT
<
{
  "title": "Method not allowed",
  "detail": "You can't complete an order that is in the CANCELLED status"
}
```

Conclusion:

The main controller for the WordPress REST API. Routes are registered to the server within WordPress. When WP_REST_Server is called upon to serve a request, it determines which route is to be called, and passes the route callback a WP_REST_Request object. WP_REST_Server also handles authentication, and can perform request validation and permissions checks.