DISTRIBUTED SYSTEMS

Lab 2

João Leitão, Sérgio Duarte, Pedro Camponês

GOALS

In the end of this lab you should be able to:

- Understand what a WebService REST is
- Know how to develop a WS REST and Server in Java (using JAX-RS)
- Know how to develop a REST Client in Java (using JAX-RX)
- Use Docker to test your service using your clients

GOALS

In the end of this lab you should be able to:

- **Understand what a WebService REST is**
- Know how to develop a WS REST and Server in Java (using JAX-RS)
- Know how to develop a REST Client in Java (using JAX-RX)
- Use Docker to test your service using your clients

REST : REPRESENTATIONAL STATE TRANSFER

Architectural Pattern to access Information

Fundamental Approach:

An application is perceived as a collection of resources.

The key implications of this are:

- A resource is identified by a URI/URL
- The URL returns a document with a representation of the resource
- A URL can refer to a collection of resources
- It is possible to refer to other resources (from a resource) using links

REST : REPRESENTATIONAL STATE TRANSFER EXAMPLE

Consider an application that is used to manage contact cards.

- A contact card is a resource and each contact card has an URL associated.
- The URL of a card will return a representation of that card (could be a textual representation of the fields of the card) – name of the person, phone, e-mail, postal address – but it could also be a binary representation.
- An URL can represent the whole collection of contact cards managed by an application.
- A contact card could contain the URL of another card, for instance to refer to the spouse of that person.

REST PROTOCOL

A client-server protocol that is **stateless**: each request contains all the information that is necessary to process the request.

- This implies that the server does not need to keep track of relations among different requests
- It makes the interaction pattern of systems using rest simple
- It allows to do transparent caching

REST Protocol

The REST interface is **uniform**: all resources are accessed by a set of well-defined HTTP operations:

- **POST**: Creates a new resource
- **GET**: Obtains (a representation of) an existing resource
- **PUT**: Updates or Replaces an existing resource
- **DELETE**: Eliminates an existing resource

GOALS

In the end of this lab you should be able to:

- Understand what a WebService REST is
- **Know how to develop a WS REST and Server in Java (using JAX-RS)**
- Know how to develop a REST Client in Java (using JAX-RX)
- Use Docker to test your service using your clients

DEVELOPMENT OF A WEB SERVICE REST IN JAVA

In the Distributed Systems Course we are using the Jersey (JAX-RS) framework, which highly simplifies the development of REST services in Java.

- When using this framework, we instrument our code through simple annotations in our Java code (e.g., @PATH, @GET, @POST, @DELETE, …)
- Java Reflection is taken advantage by the Jersey runtime to derive code automatically based on those annotations.

Want to know more? https://eclipse-ee4j.github.io/jersey/

DEVELOPMENT OF A WEB SERVICE REST IN JAVA

There are a few dependencies that our code will have. As discussed last week these will be handled by Maven. The dependencies are inserted in the pom.xml file:

```
<dependencies>

  <dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>3.1.1</version>
  </dependency>

  <dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-jdk-http</artifactId>
    <version>3.1.1</version>
  </dependency>

  <dependency>
    <groupId>org.glassfish.jersey.inject</groupId>
    <artifactId>jersey-hk2</artifactId>
    <version>3.1.1</version>
  </dependency>

  <dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
    <version>4.0.5</version>
  </dependency>

</dependencies>
```

DEVELOPMENT OF A WEB SERVICE REST IN JAVA

We are going to show how to develop a Web Service REST by example.

Our example, *not accidentally*, is based on the construction of a simple user management service.

- Users are identifier by a unique identifier (username).
- Users have a password that protects changes to their data, a full name, and an e-mail address.
- In this example, we are allowing the service to associate and return an avatar image (png format) as a file stored in the filesystem where the server executes. This should be handled by a different service; we will tackle that next week.
- Users are going to be our main resource in this example.

Modelling our User Resource as a Java Class

Standard Java Class

```
package lab2.api;
public class User {
        private String email;
        private String userId;
        private String fullName;
        private String password;

        public User(){
        }
        public User(String userId, String fullName, String email, String password) {
                super();
                this.email = email;
                this.userId = userId;
                this.fullName = fullName;
                this.password = password;
        }
        @Override
        public int hashCode() { … }
        @Override
        public boolean equals(Object obj) { … }
        //Defaults getters and setters for the fields
        public String getEmail() { … }
        public void setEmail(String email) { … }
        public String getUserId() { … }
        ….
```

Modelling our Message Resource as a Java Class

Standard Java Class

```
package lab2.api;
public class User {
        private String email;
        private String userId;
        private String fullName;
        private String password;

        public User(){
        }
        public User(String userId, String fullName, String email, String password) {
                super();
                this.email = email;
                this.userId = userId;
                this.fullName = fullName;
                this.password = password;
        }
        @Override
        public int hashCode() { … }
        @Override
        public boolean equals(Object obj) { … }
        //Defaults getters and setters for the fields
        public String getEmail() { … }
        public void setEmail(String email) { … }
        public String getUserId() { … }
        ….
```

Private Fields (but you must create standard getters and setters)

Modelling our Message Resource as a Java Class

```
package lab2.api;
public class User {
        private String email;
        private String userId;
        private String fullName;
        private String password;

        public User(){
        }
        public User(String userId, String fullName, String email, String password) {
                super();
                this.email = email;
                this.userId = userId;
                this.fullName = fullName;
                this.password = password;
        }
        @Override
        public int hashCode() { … }
        @Override
        public boolean equals(Object obj) { … }
        //Defaults getters and setters for the fields
        public String getEmail() { … }
        public void setEmail(String email) { … }
        public String getUserId() { … }
        ….
```

Standard Java Class

You can have any number of constructors…

Modelling our Message Resource as a Java Class

```
package lab2.api;
public class User {
        private String email;
        private String userId;
        private String fullName;
        private String password;

        public User(){
        }
        public User(String userId, String fullName, String email, String password)
                super();
                this.email = email;
                this.userId = userId;
                this.fullName = fullName;
                this.password = password;
        }
        @Override
        public int hashCode() { … }
        @Override
        public boolean equals(Object obj) { … }
        //Defaults getters and setters for the fields
        public String getEmail() { … }
        public void setEmail(String email) { … }
        public String getUserId() { … }
        ….
```

Standard Java Class

But you should have a default constructor without arguments.

(in this case all class attributes get a value of null)

Modelling our Message Resource as a Java Class

```
package lab2.api;
public class User {
        private String email;
        private String userId;
        private String fullName;
        private String password;

        public User(){
        }
        public User(String userId, String fullName, String email, String password)
                super();
                this.email = email;
                this.userId = userId;
                this.fullName = fullName;
                this.password = password;
        }
        @Override
        public int hashCode() { … }
        @Override
        public boolean equals(Object obj) { … }
        //Defaults getters and setters for the fields
        public String getEmail() { … }
        public void setEmail(String email) { … }
        public String getUserId() { … }
        ….
```

Standard Java Class

But you should have a default constructor without arguments.

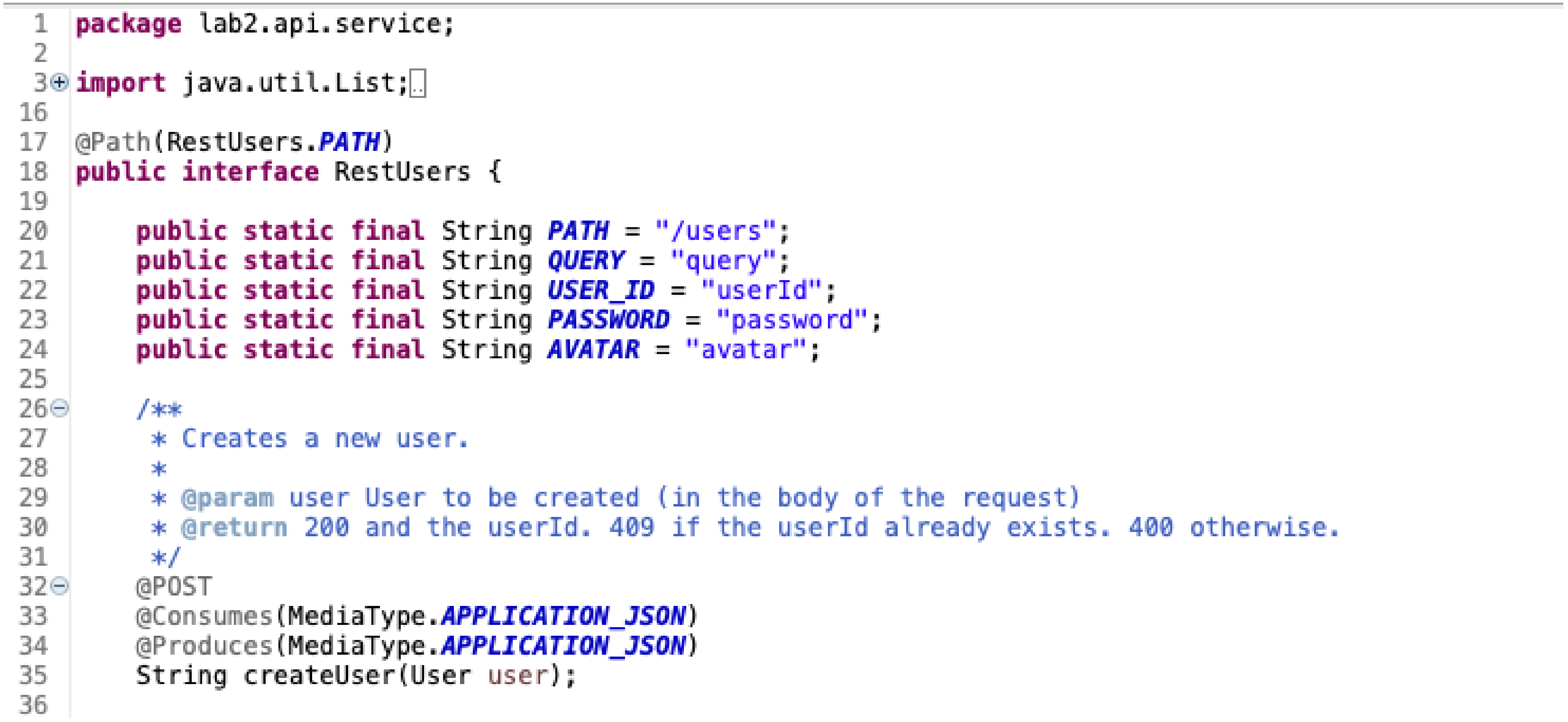
(in this case all class attributes get a value of null)

Default constructor and getters/setters are important to allow the serialization and deserialization of this class over the network.

Defining the Service Interface

```
package lab2.api.service;

@Path(RestUsers.PATH)
public interface RestUsers {

		public static final String PATH = "/users";
		public static final String QUERY = "query";
		public static final String USER_ID = "userId";
		public static final String PASSWORD = "password";
		public static final String AVATAR = "avatar";

		@POST
		@Consumes(MediaType.APPLICATION_JSON)
		@Produces(MediaType.APPLICATION_JSON)
		String createUser(User user);

		@GET
		@Path("/{" + USER_ID + "}")
		@Produces(MediaType.APPLICATION_JSON)
		User getUser(@PathParam(USER_ID) String userId,
@QueryParam(PASSWORD) String password);

		@PUT
		@Path("/{" + USER_ID + "}")
		@Consumes(MediaType.APPLICATION_JSON)
		@Produces(MediaType.APPLICATION_JSON)
		User updateUser(@PathParam(USER_ID) String userId,
@QueryParam(PASSWORD) String password, User user);

		@DELETE
		@Path("/{" + USER_ID + "}")
		@Produces(MediaType.APPLICATION_JSON)
		User deleteUser(@PathParam(USER_ID) String userId,
@QueryParam(PASSWORD) String password);

		@GET
		@Produces(MediaType.APPLICATION_JSON)
		List<User> searchUsers(@QueryParam(QUERY) String pattern);

		@PUT
		@Path("{" + USER_ID + "}/" + AVATAR)
		@Consumes(MediaType.APPLICATION_OCTET_STREAM)
		void associateAvatar(@PathParam(USER_ID) String userId,
@QueryParam(PASSWORD) String password, byte[] avatar);

		@DELETE
		@Path("{" + USER_ID + "}/" + AVATAR)
		void removeAvatar(@PathParam(USER_ID) String userId,
@QueryParam(PASSWORD) String password);

		@GET
		@Path("{" + USER_ID + "}/" + AVATAR)
		@Produces(MediaType.APPLICATION_OCTET_STREAM)
		byte[] getAvatar(@PathParam(USER_ID) String userId);

}
```

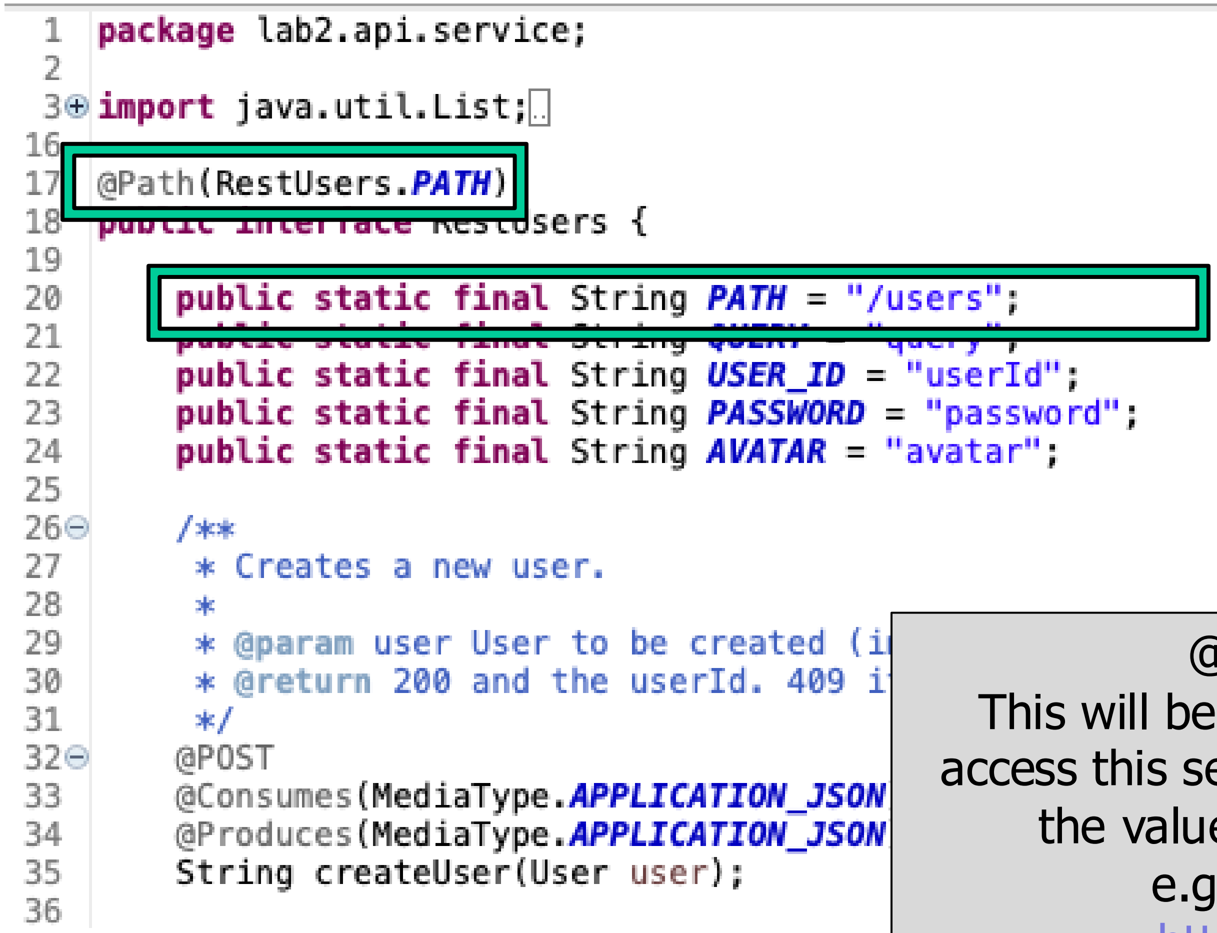
DEFINING THE SERVICE INTERFACE



```java
package lab2.api.service;

import java.util.List;

@Path(RestUsers.PATH)
public interface RestUsers {

    public static final String PATH = "/users";
    public static final String QUERY = "query";
    public static final String USER_ID = "userId";
    public static final String PASSWORD = "password";
    public static final String AVATAR = "avatar";

    /**
     * Creates a new user.
     *
     * @param user User to be created (in the body of the request)
     * @return 200 and the userId. 409 if the userId already exists. 400 otherwise.
     */
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    String createUser(User user);

```

Standard Java Interface enriched with Jersey annotations and identifying the methods supported by your service.

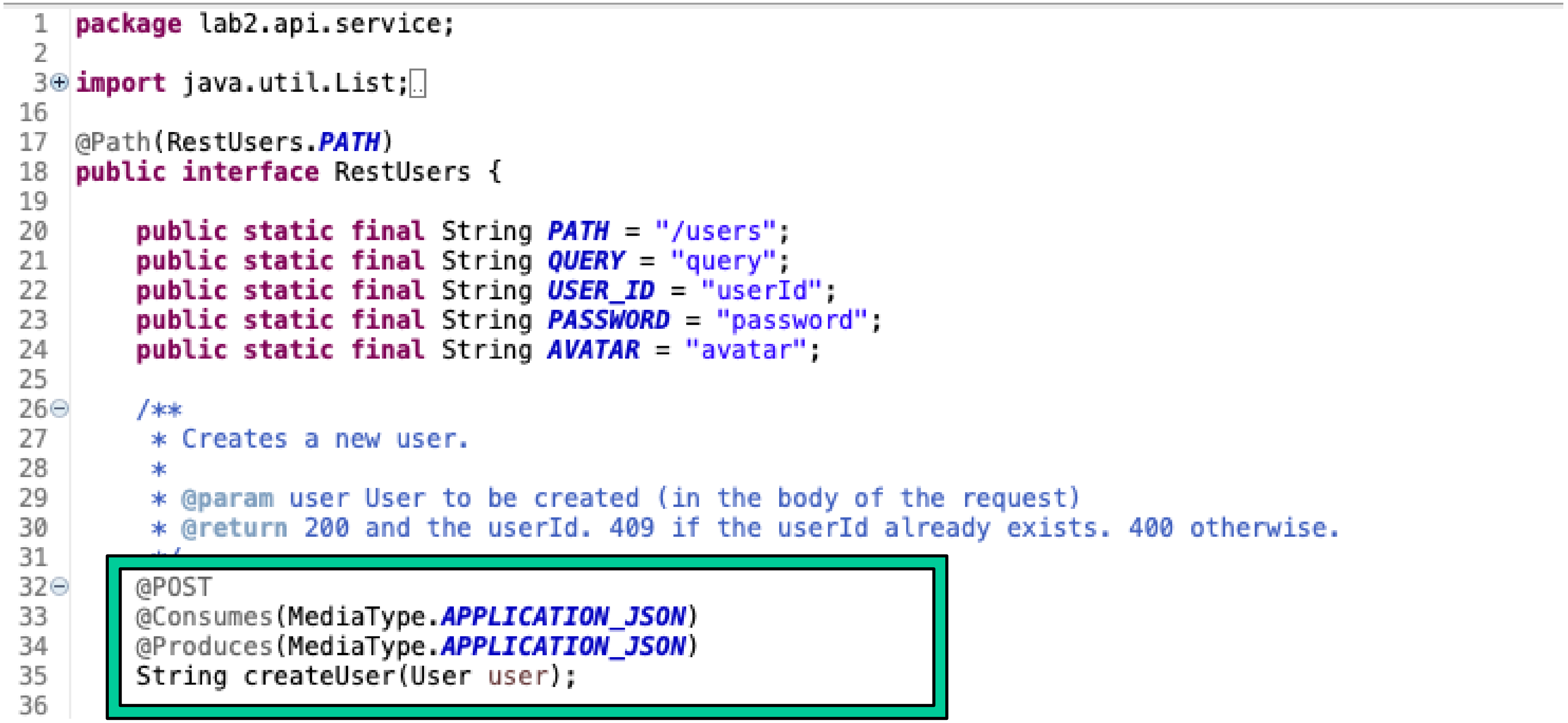
DEFINING THE SERVICE INTERFACE



```
package lab2.api.service;

import java.util.List;

@Path(RestUsers.PATH)
public interface RestUsers {

    public static final String PATH = "/users";
    public static final String QUERY = "query";
    public static final String USER_ID = "userId";
    public static final String PASSWORD = "password";
    public static final String AVATAR = "avatar";

    /**
     * Creates a new user.
     *
     * @param user User to be created (i
     * @return 200 and the userId. 409 i
     */
    @POST
    @Consumes(MediaType.APPLICATION_JSON
    @Produces(MediaType.APPLICATION_JSON
    String createUser(User user);

```

@Path(STRING VALUE)
This will be used to define the URL used to access this service. It will be the Server URL + the value provided in this annotation.
e.g., if the Server URL was
http://myserver:8080/rest
this service would be accessed by URLs starting with:
http://myserver:8080/rest/users

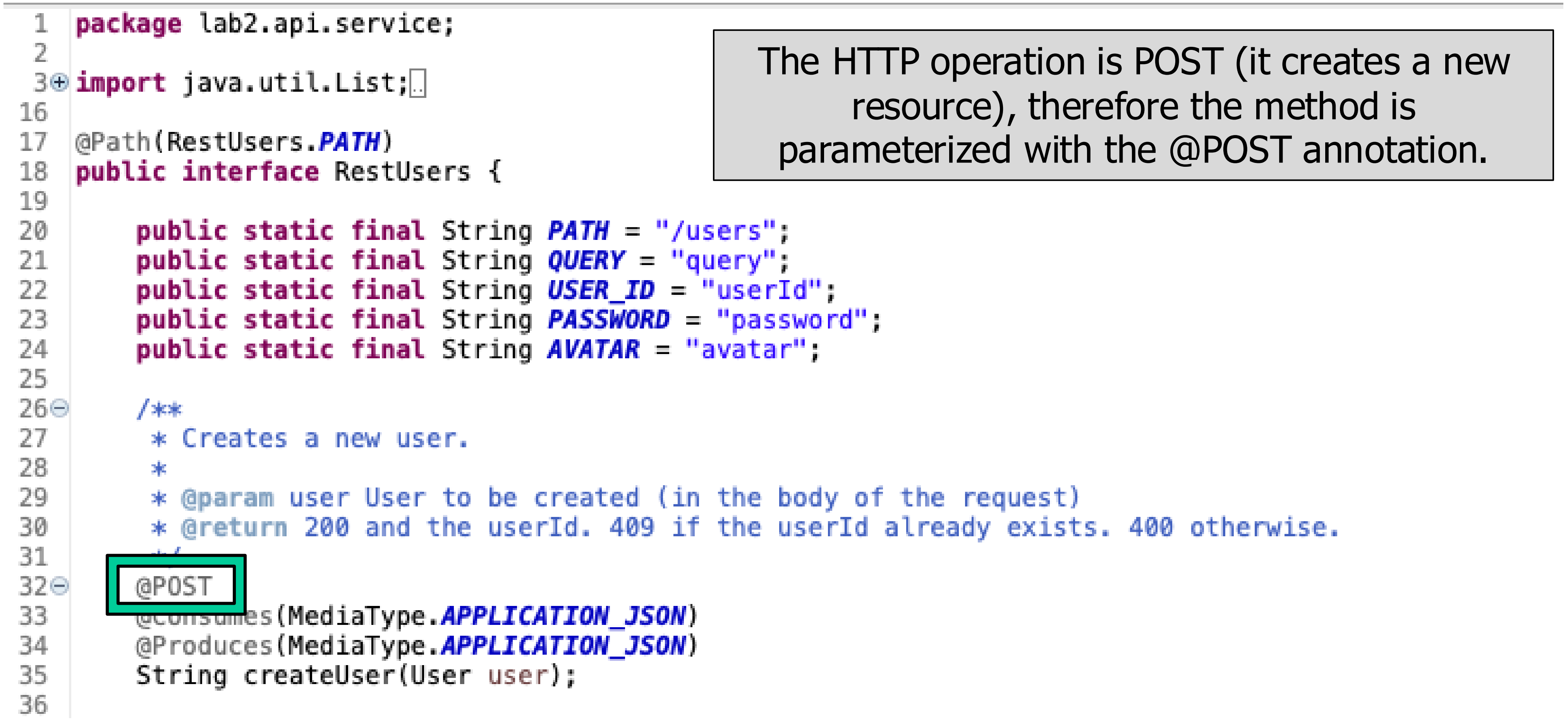
Defining the Service Interface



```java
package lab2.api.service;

import java.util.List;

@Path(RestUsers.PATH)
public interface RestUsers {

    public static final String PATH = "/users";
    public static final String QUERY = "query";
    public static final String USER_ID = "userId";
    public static final String PASSWORD = "password";
    public static final String AVATAR = "avatar";

    /**
     * Creates a new user.
     *
     * @param user User to be created (in the body of the request)
     * @return 200 and the userId. 409 if the userId already exists. 400 otherwise.
     */
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    String createUser(User user);

```

This method will allow to create (i.e., store) a new user in the server.

DEFINING THE SERVICE INTERFACE

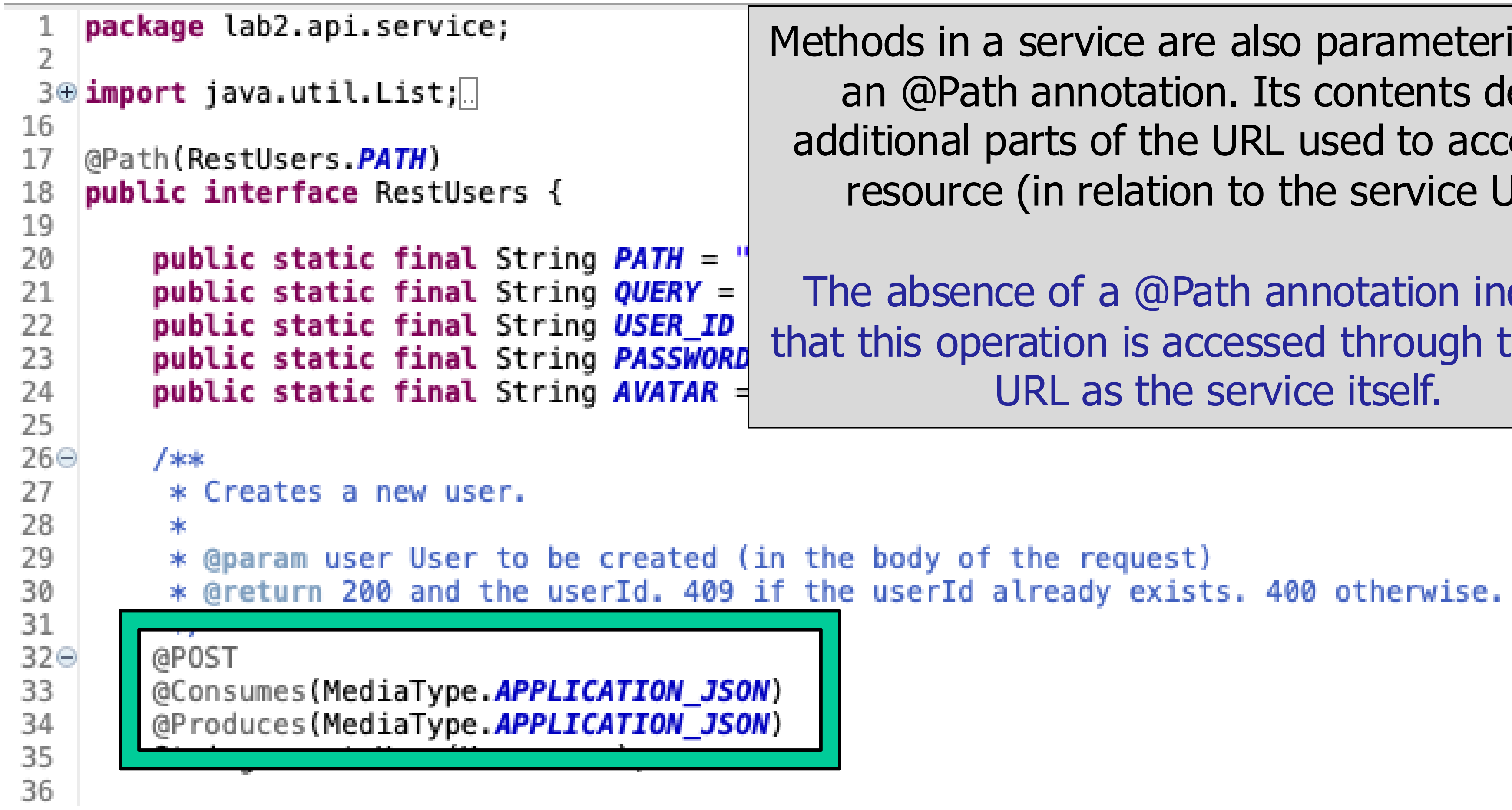


```java
package lab2.api.service;

import java.util.List;

@Path(RestUsers.PATH)
public interface RestUsers {

    public static final String PATH = "/users";
    public static final String QUERY = "query";
    public static final String USER_ID = "userId";
    public static final String PASSWORD = "password";
    public static final String AVATAR = "avatar";

    /**
     * Creates a new user.
     *
     * @param user User to be created (in the body of the request)
     * @return 200 and the userId. 409 if the userId already exists. 400 otherwise.
     */
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    String createUser(User user);

```

The HTTP operation is POST (it creates a new resource), therefore the method is parameterized with the @POST annotation.

This method will allow to create (i.e., store) a new user in the server.

DEFINING THE SERVICE INTERFACE



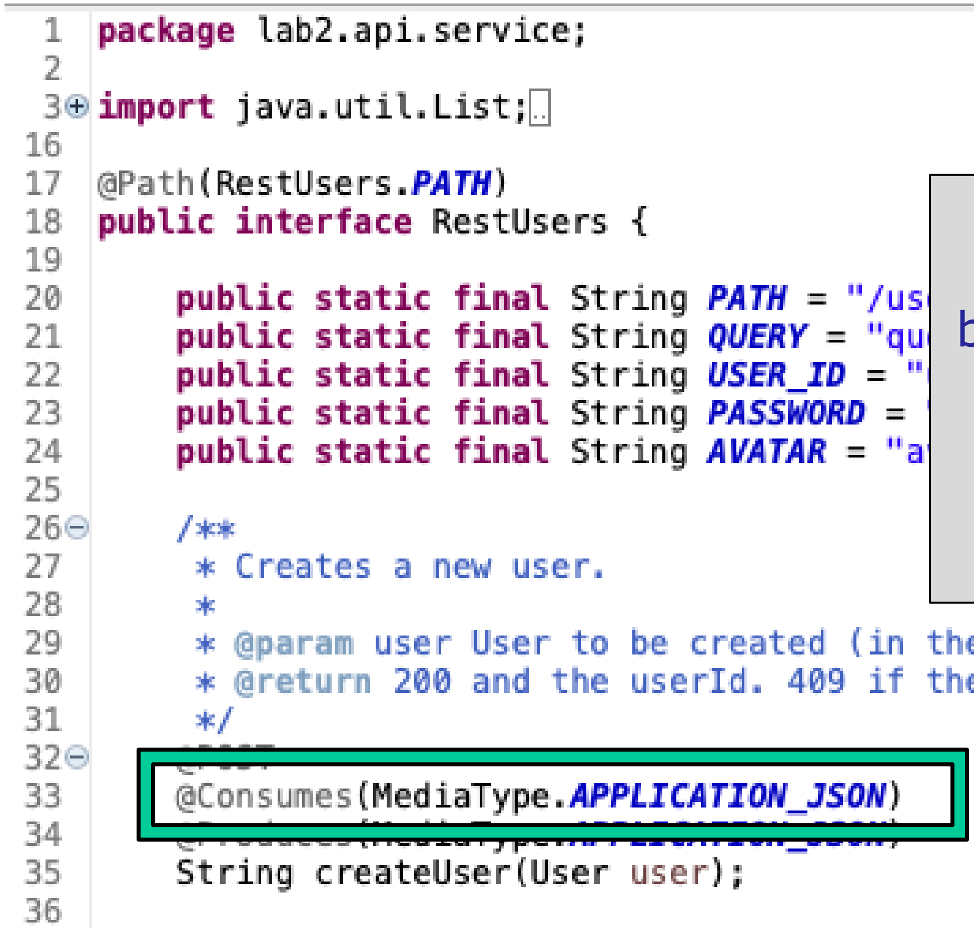
```
package lab2.api.service;

import java.util.List;

@Path(RestUsers.PATH)
public interface RestUsers {

    public static final String PATH = "
    public static final String QUERY =
    public static final String USER_ID
    public static final String PASSWORD
    public static final String AVATAR =

    /**
     * Creates a new user.
     *
     * @param user User to be created (in the body of the request)
     * @return 200 and the userId. 409 if the userId already exists. 400 otherwise.

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
```

Methods in a service are also parameterized with an @Path annotation. Its contents define additional parts of the URL used to access this resource (in relation to the service URL).

The absence of a @Path annotation indicates that this operation is accessed through the same URL as the service itself.

This method will allow to create (i.e., store) a new user in the server.

DEFINING THE SERVICE INTERFACE



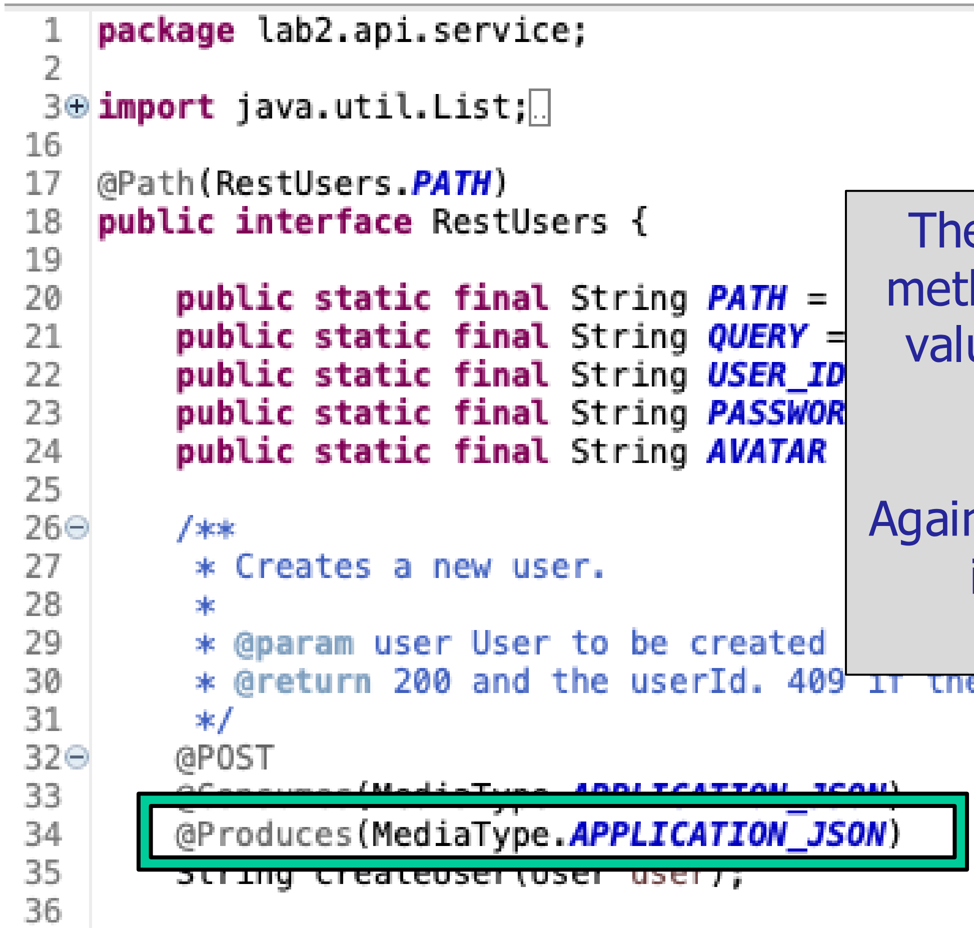
```java
package lab2.api.service;

import java.util.List;

@Path(RestUsers.PATH)
public interface RestUsers {

    public static final String PATH = "/us
    public static final String QUERY = "qu
    public static final String USER_ID = "
    public static final String PASSWORD =
    public static final String AVATAR = "a

    /**
     * Creates a new user.
     *
     * @param user User to be created (in the body of the request)
     * @return 200 and the userId. 409 if the userId already exists. 400 otherwise.
     */
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    String createUser(User user);

```

The @Consumes annotation indicates that this method will receive an argument through the body of the HTTP request (The parameter user).

We typically encode Java objects sent in the body of an HTTP request in JavaScript Object Notation (JSON)

This method will allow to create (i.e., store) a new user in the server.

DEFINING THE SERVICE INTERFACE



```java
package lab2.api.service;

import java.util.List;

@Path(RestUsers.PATH)
public interface RestUsers {

    public static final String PATH =
    public static final String QUERY =
    public static final String USER_ID
    public static final String PASSWOR
    public static final String AVATAR

    /**
     * Creates a new user.
     *
     * @param user User to be created
     * @return 200 and the userId. 409 if the userId already exists. 400 otherwise.
     */
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    String createUser(User user);

```

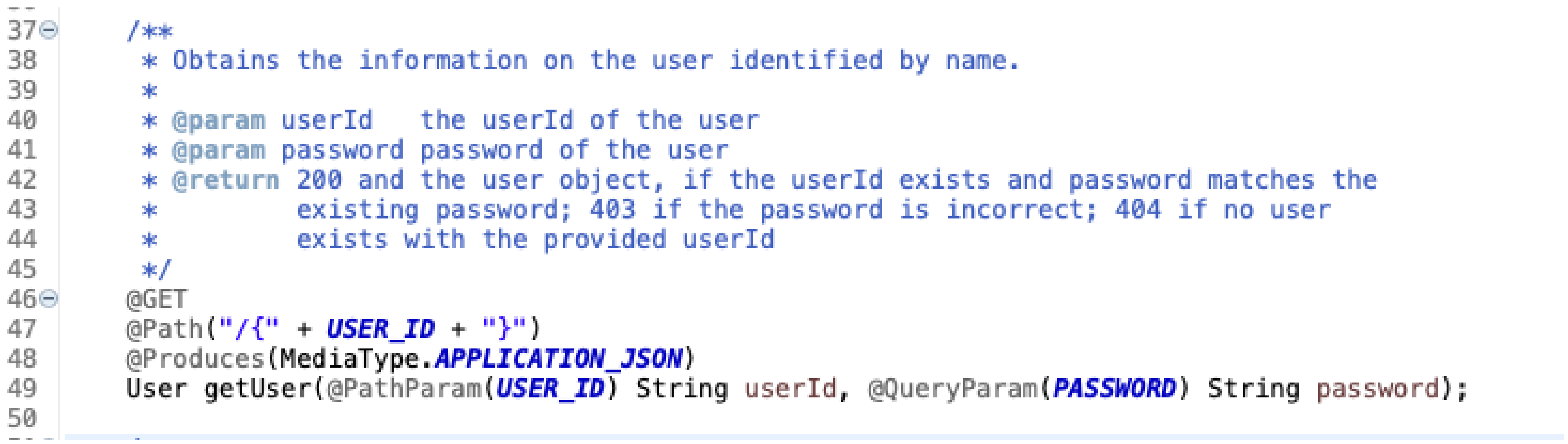
The @Produces annotation indicates that this method will return a value (in this case a String value) that will be encoded in the body of the HTTP response sent back to the client.

Again, we typically encode native java types sent in the body of an HTTP request/reply in JavaScript Object Notation (JSON)

This method will allow to create (i.e., store) a new user in the server.

DEFINING THE SERVICE INTERFACE

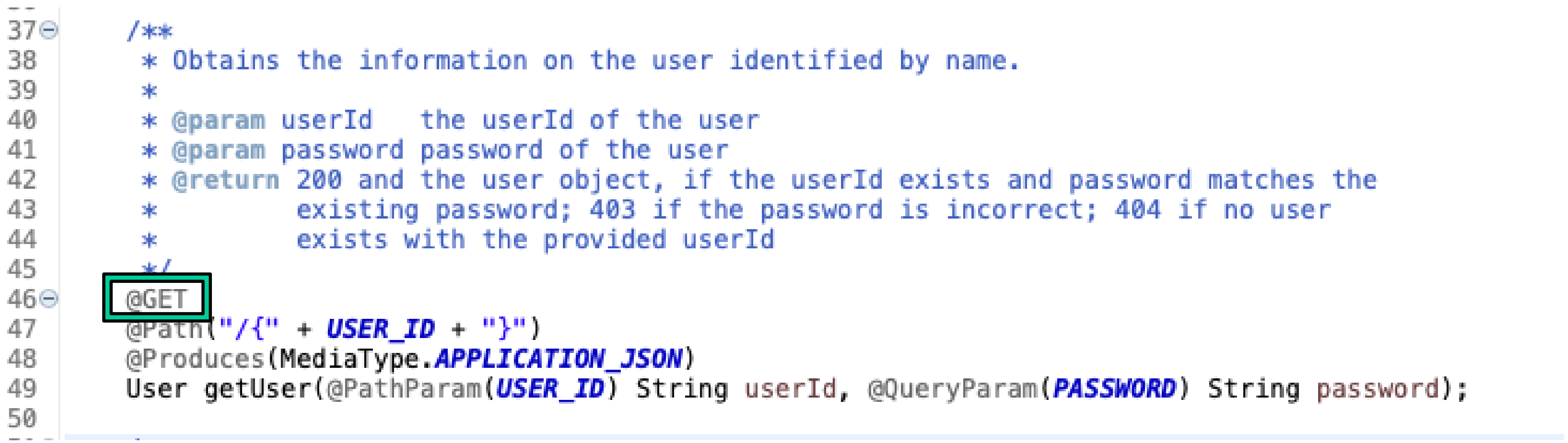
This method will allow to, respectively, access an existing user in the server



```java
	/**
	 * Obtains the information on the user identified by name.
	 *
	 * @param userId   the userId of the user
	 * @param password password of the user
	 * @return 200 and the user object, if the userId exists and password matches the
	 *         existing password; 403 if the password is incorrect; 404 if no user
	 *         exists with the provided userId
	 */
	@GET
	@Path("/{" + USER_ID + "}")
	@Produces(MediaType.APPLICATION_JSON)
	User getUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);
```

DEFINING THE SERVICE INTERFACE

This method will allow to, respectively, access an existing user in the server

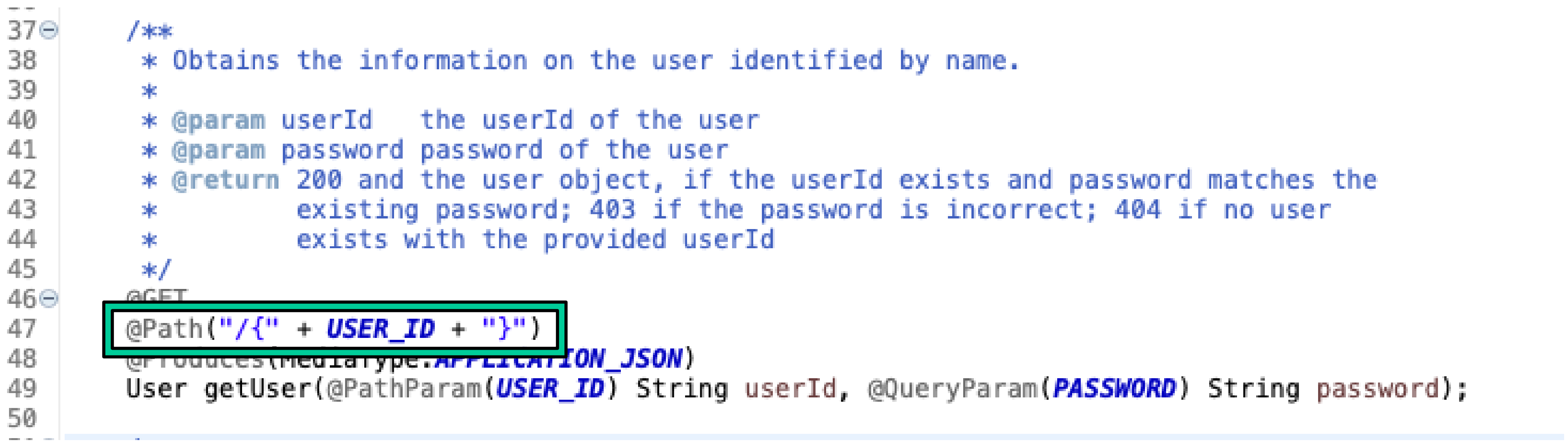


```java
	/**
	 * Obtains the information on the user identified by name.
	 *
	 * @param userId   the userId of the user
	 * @param password password of the user
	 * @return 200 and the user object, if the userId exists and password matches the
	 *         existing password; 403 if the password is incorrect; 404 if no user
	 *         exists with the provided userId
	 */
	@GET
	@Path("/{" + USER_ID + "}")
	@Produces(MediaType.APPLICATION_JSON)
	User getUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);
```

The method exposes a representation of a resource (GET Operations). Therefore, it must be parameterized with the @GET annotation.

DEFINING THE SERVICE INTERFACE

This method will allow to, respectively, access an existing user in the server

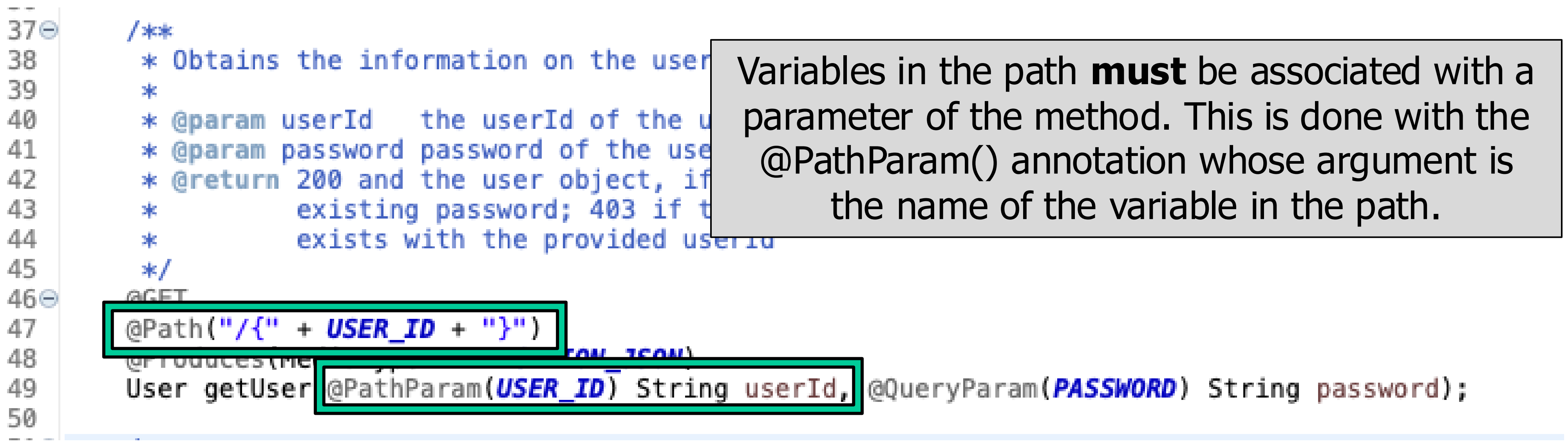


```
    /**
     * Obtains the information on the user identified by name.
     *
     * @param userId   the userId of the user
     * @param password password of the user
     * @return 200 and the user object, if the userId exists and password matches the
     *         existing password; 403 if the password is incorrect; 404 if no user
     *         exists with the provided userId
     */
    @GET
    @Path("/{" + USER_ID + "}")
    @Produces(MediaType.APPLICATION_JSON)
    User getUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);
```

Note that the @Path annotation has a value within {} (the constant USER_ID that has the value ”userId”). This indicates that this part of the path will be a variable – named in this case userId.

DEFINING THE SERVICE INTERFACE

This method will allow to, respectively, access an existing user in the server



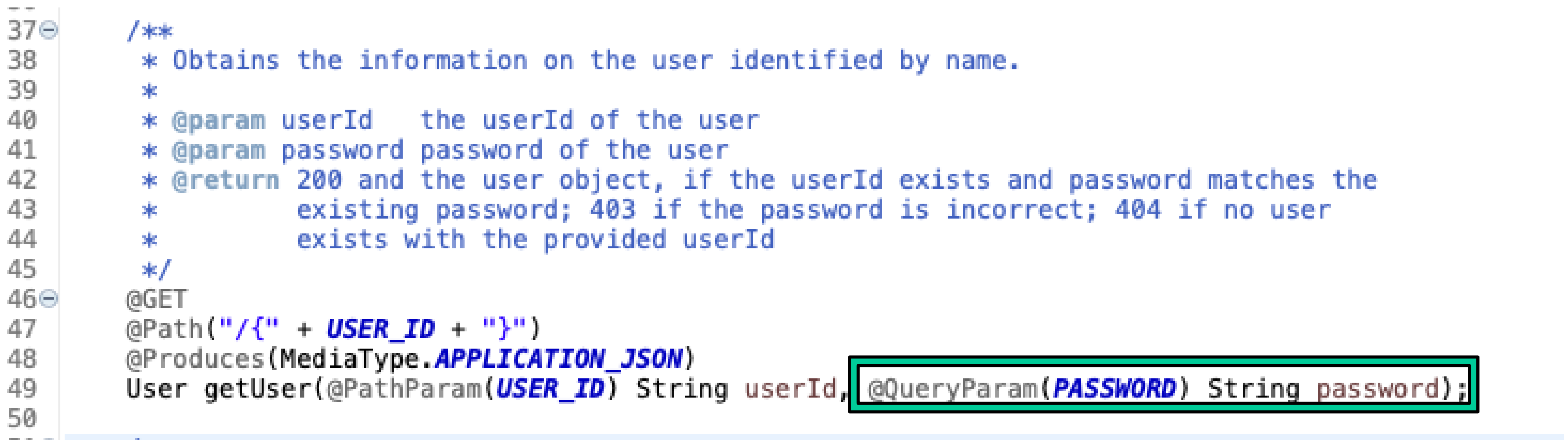
```
/**
 * Obtains the information on the user
 *
 * @param userId   the userId of the u
 * @param password password of the use
 * @return 200 and the user object, if
 *         existing password; 403 if t
 *         exists with the provided userId
 */
@GET
@Path("/{" + USER_ID + "}")
@Produces(Me                 ON_JSON)
User getUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);
```

Variables in the path **must** be associated with a parameter of the method. This is done with the @PathParam() annotation whose argument is the name of the variable in the path.

Jersey will process the URL automatically and assign it to that method parameter.
You can have multiple path variables mapped with @PathParam in the same path.
Only native types (including String) can be passed as Path parameters (i.e., Java classes and byte[] have to be passed through the body of the HTTP request using the corresponding annotation @Consumes)

DEFINING THE SERVICE INTERFACE

This method will allow to, respectively, access an existing user in the server

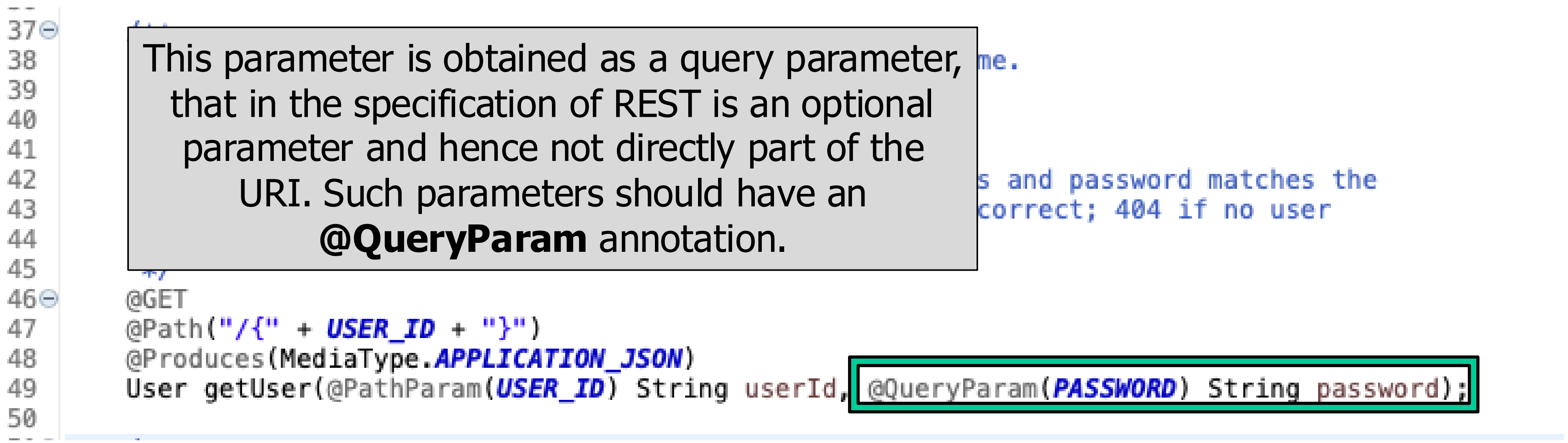


```java
    /**
     * Obtains the information on the user identified by name.
     *
     * @param userId   the userId of the user
     * @param password password of the user
     * @return 200 and the user object, if the userId exists and password matches the
     *         existing password; 403 if the password is incorrect; 404 if no user
     *         exists with the provided userId
     */
    @GET
    @Path("/{" + USER_ID + "}")
    @Produces(MediaType.APPLICATION_JSON)
    User getUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);
```

Notice that this parameter as no corresponding element in the @Path annotation…

DEFINING THE SERVICE INTERFACE

This method will allow to, respectively, access an existing user in the server



This parameter is obtained as a query parameter, that in the specification of REST is an optional parameter and hence not directly part of the URI. Such parameters should have an **@QueryParam** annotation.

```
    /**
me.
s and password matches the
correct; 404 if no user
     */
    @GET
    @Path("/{" + USER_ID + "}")
    @Produces(MediaType.APPLICATION_JSON)
    User getUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);
```

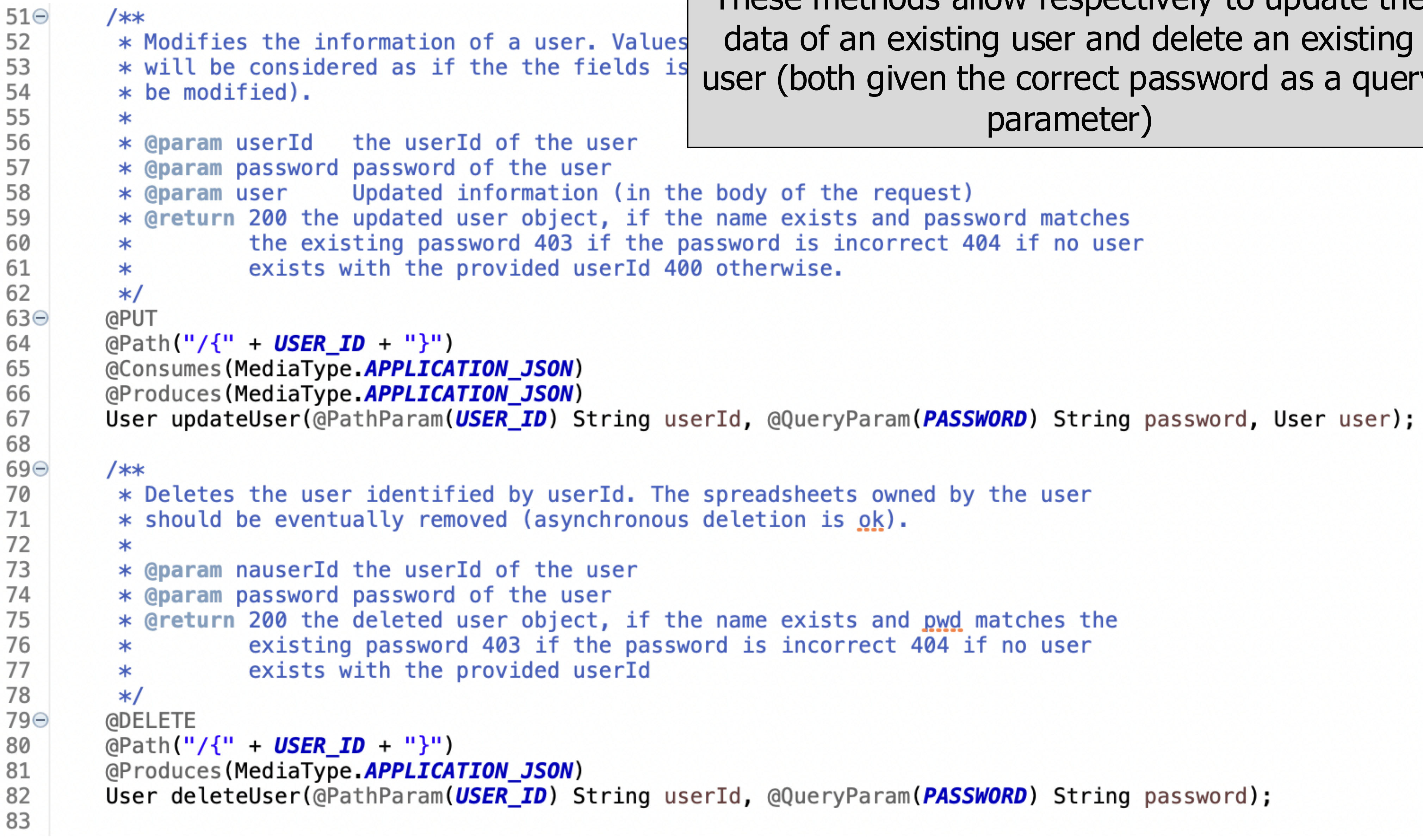
@QueryParam takes as argument the name of the optional parameter (in this case the constant PASSWORD whose value is "password").

Note that query parameters are not part of the @Path (otherwise they would be mandatory)

They can be passed in the URL using the ? character. E.g., http://myserver:8080/rest/users/jleitao?password=123456

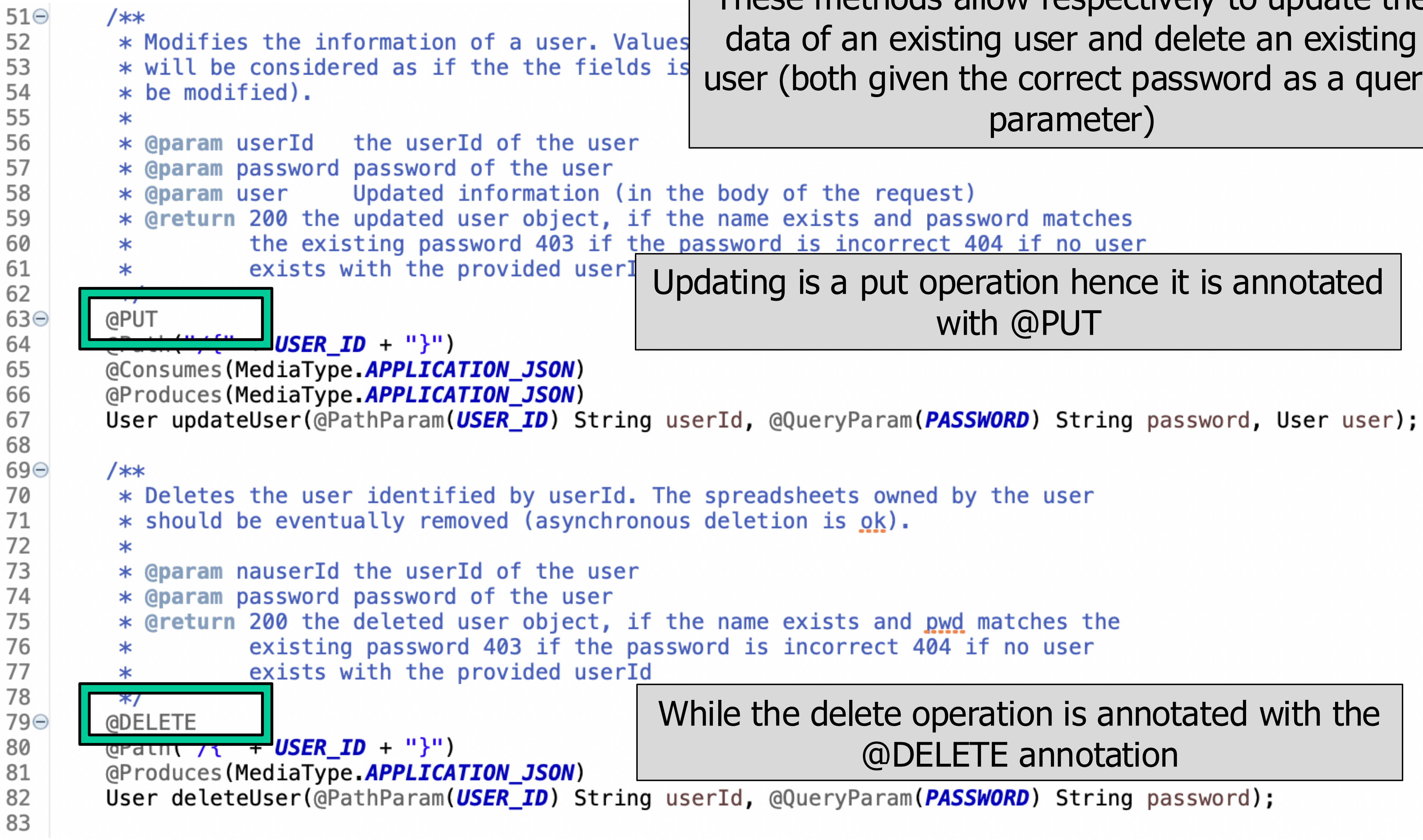
DEFINING THE SERVICE INTERFACE

These methods allow respectively to update the data of an existing user and delete an existing user (both given the correct password as a query parameter)



```java
    /**
     * Modifies the information of a user. Values
     * will be considered as if the the fields is
     * be modified).
     *
     * @param userId   the userId of the user
     * @param password password of the user
     * @param user     Updated information (in the body of the request)
     * @return 200 the updated user object, if the name exists and password matches
     *         the existing password 403 if the password is incorrect 404 if no user
     *         exists with the provided userId 400 otherwise.
     */
    @PUT
    @Path("/{" + USER_ID + "}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    User updateUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password, User user);

    /**
     * Deletes the user identified by userId. The spreadsheets owned by the user
     * should be eventually removed (asynchronous deletion is ok).
     *
     * @param nauserId the userId of the user
     * @param password password of the user
     * @return 200 the deleted user object, if the name exists and pwd matches the
     *         existing password 403 if the password is incorrect 404 if no user
     *         exists with the provided userId
     */
    @DELETE
    @Path("/{" + USER_ID + "}")
    @Produces(MediaType.APPLICATION_JSON)
    User deleteUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);
```

DEFINING THE SERVICE INTERFACE



```
    /**
     * Modifies the information of a user. Values
     * will be considered as if the the fields is
     * be modified).
     *
     * @param userId   the userId of the user
     * @param password password of the user
     * @param user     Updated information (in the body of the request)
     * @return 200 the updated user object, if the name exists and password matches
     *         the existing password 403 if the password is incorrect 404 if no user
     *         exists with the provided userI
     */
    @PUT
    @Path("/{" + USER_ID + "}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    User updateUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password, User user);

    /**
     * Deletes the user identified by userId. The spreadsheets owned by the user
     * should be eventually removed (asynchronous deletion is ok).
     *
     * @param nauserId the userId of the user
     * @param password password of the user
     * @return 200 the deleted user object, if the name exists and pwd matches the
     *         existing password 403 if the password is incorrect 404 if no user
     *         exists with the provided userId
     */
    @DELETE
    @Path("/{" + USER_ID + "}")
    @Produces(MediaType.APPLICATION_JSON)
    User deleteUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);
```

These methods allow respectively to update the data of an existing user and delete an existing user (both given the correct password as a query parameter)

Updating is a put operation hence it is annotated with @PUT

While the delete operation is annotated with the @DELETE annotation

DEFINING THE SERVICE INTERFACE

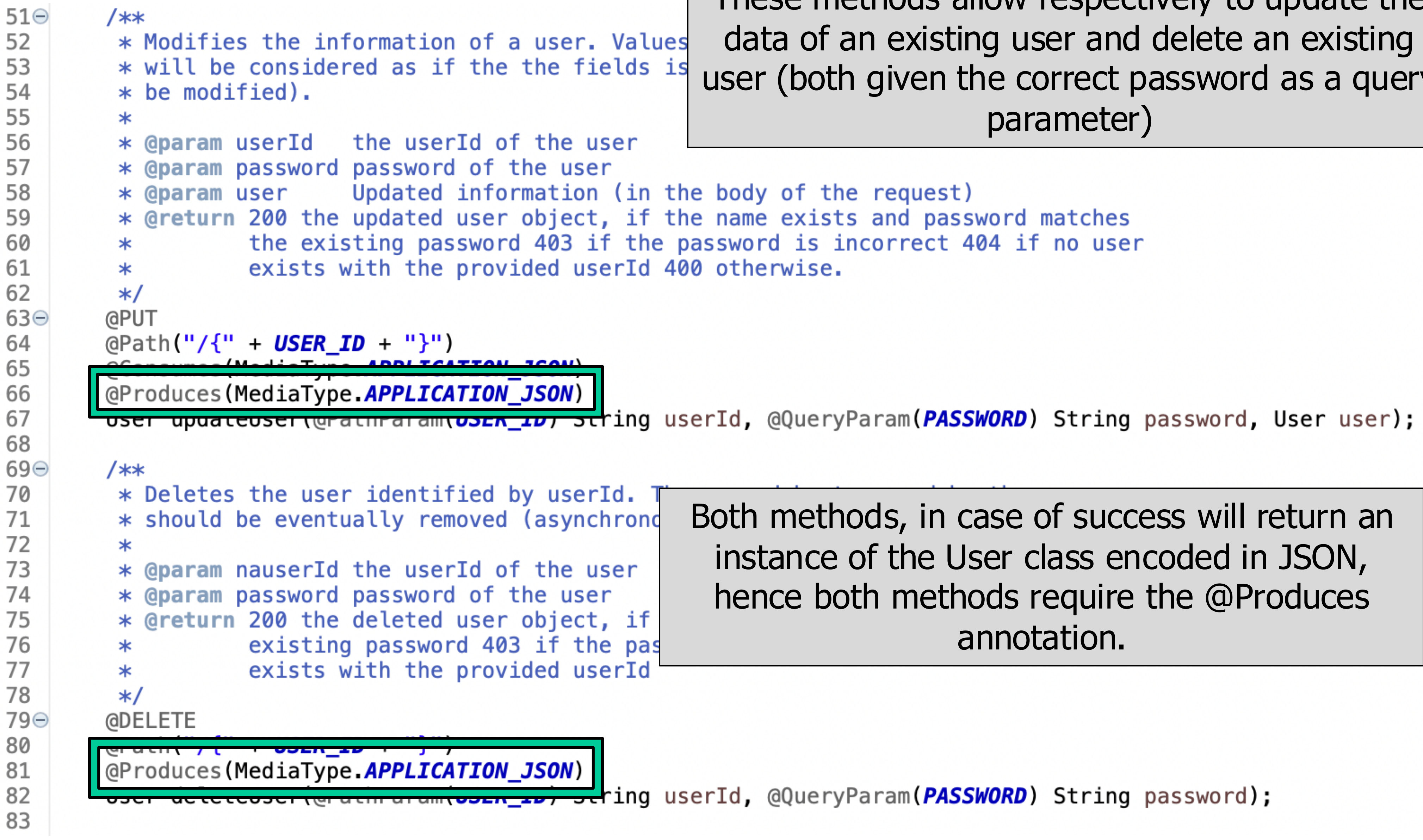
These methods allow respectively to update the data of an existing user and delete an existing user (both given the correct password as a query parameter)

```
/**
 * Modifies the information of a user. Values
 * will be considered as if the the fields is
 * be modified).
 *
 * @param userId   the userId of the user
 * @param password password of the user
 * @param user     Updated information (in the body of the request)
 * @return 200 the updated user object, if the name exists and password matches
 *         the existing password 403 if the password is incorrect 404 if no user
 *         exists with the provided userId 400 otherwise.
 */
@PUT
@Path("/{" + USER_ID + "}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
User updateUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password, User user);

/**
 * Deletes the user identified by userId. T
 * should be eventually removed (asynchrono
 *
 * @param nauserId the userId of the user
 * @param password password of the user
 * @return 200 the deleted user object, if
 *         existing password 403 if the pas
 *         exists with the provided userId
 */
@DELETE
@Path("/{" + USER_ID + "}")
@Produces(MediaType.APPLICATION_JSON)
User deleteUser(@PathParam(USER_ID) String
```

The PUT operation can take a body parameter, and as such it is annotated with the @Consumes annotation. This body parameter (received in JSON format) is converted to a Java instance and associated with the method parameter that has no annotation.

Remember that at most you can receive one body parameter (on PUT and POST operations only)

DEFINING THE SERVICE INTERFACE



```
    /**
     * Modifies the information of a user. Values
     * will be considered as if the the fields is
     * be modified).
     *
     * @param userId   the userId of the user
     * @param password password of the user
     * @param user     Updated information (in the body of the request)
     * @return 200 the updated user object, if the name exists and password matches
     *         the existing password 403 if the password is incorrect 404 if no user
     *         exists with the provided userId 400 otherwise.
     */
    @PUT
    @Path("/{" + USER_ID + "}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    User updateUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password, User user);

    /**
     * Deletes the user identified by userId. T
     * should be eventually removed (asynchrono
     *
     * @param nauserId the userId of the user
     * @param password password of the user
     * @return 200 the deleted user object, if
     *         existing password 403 if the pas
     *         exists with the provided userId
     */
    @DELETE
    @Path("/{" + USER_ID + "}")
    @Produces(MediaType.APPLICATION_JSON)
    User deleteUser(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);
```

These methods allow respectively to update the data of an existing user and delete an existing user (both given the correct password as a query parameter)

Both methods, in case of success will return an instance of the User class encoded in JSON, hence both methods require the @Produces annotation.

DEFINING THE SERVICE INTERFACE

This method allows to obtain a list of users (password should be removed before exposing these resources to users) with an optional parameter that is a query.

If no pattern is provided, then all users should be returned.

```
/**
 * Returns the list of users for which
 * (of the user), case-insensitive. The
 * query must be set to the empty string "".
 *
 * @param pattern substring to search
 * @return 200 when the search was successful, regardless of the number of hits
 *         (including 0 hits). 400 otherwise.
 */
@GET
@Produces(MediaType.APPLICATION_JSON)
List<User> searchUsers(@QueryParam(QUERY) String pattern);
```

DEFINING THE SERVICE INTERFACE

This method allows to obtain a list of users (password should be removed before exposing these resources to users) with an optional parameter that is a query.

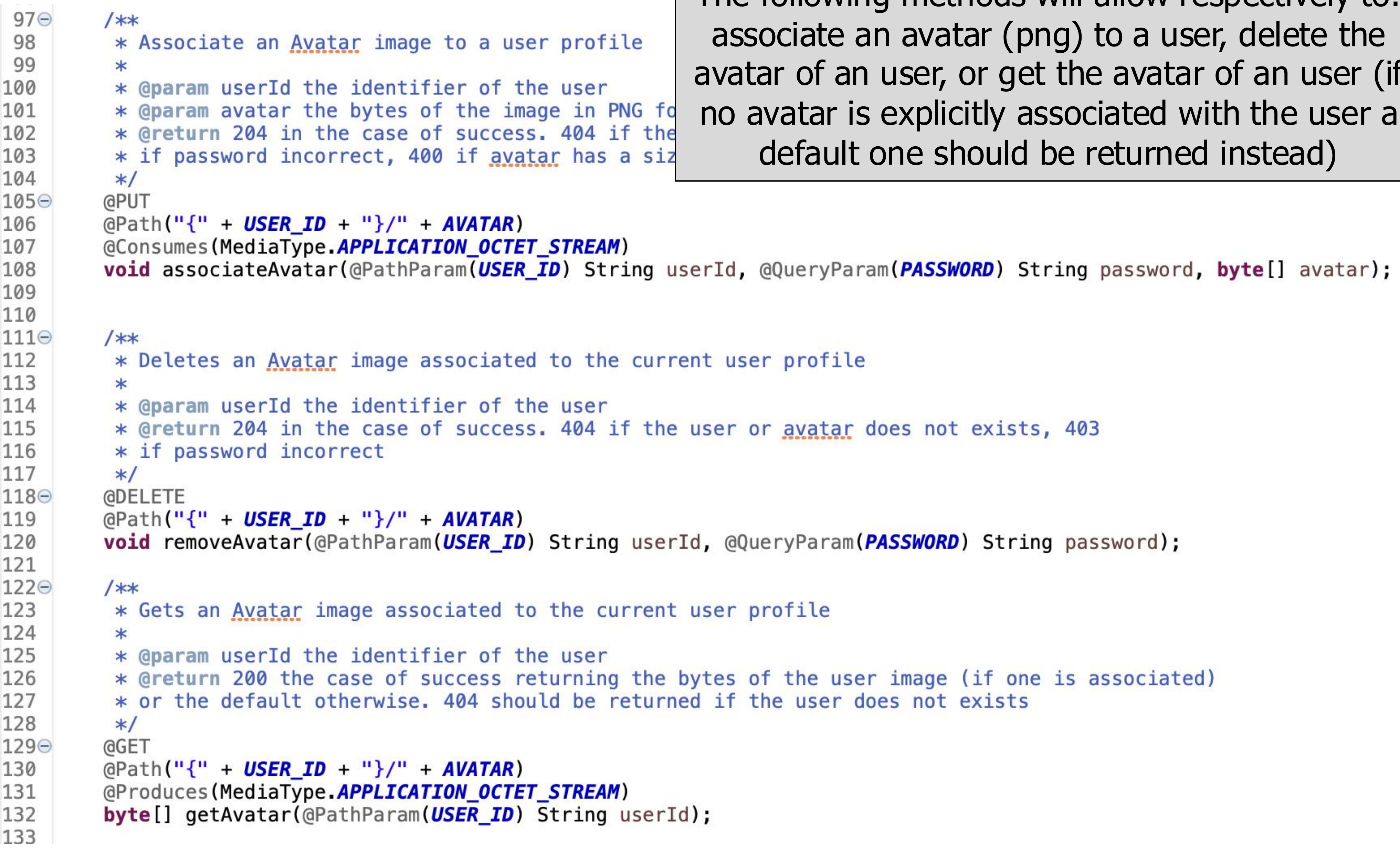
If no pattern is provided, then all users should be returned.

```
/**
 * Returns the list of users for which t
 * (of the user), case-insensitive. The password of the users returned by the
 * query must be set to the empty string "".
 *
 * @param pattern substring to search
 * @return 200 when the search was successful, regardless of the number of hits
 *          (including 0 hits). 400 otherwise.
 */
@GET
@Produces(MediaType.APPLICATION_JSON)
List<User> searchUsers(@QueryParam(QUERY) String pattern);
```

The query parameter has the value “query” which is the value of the constant QUERY.

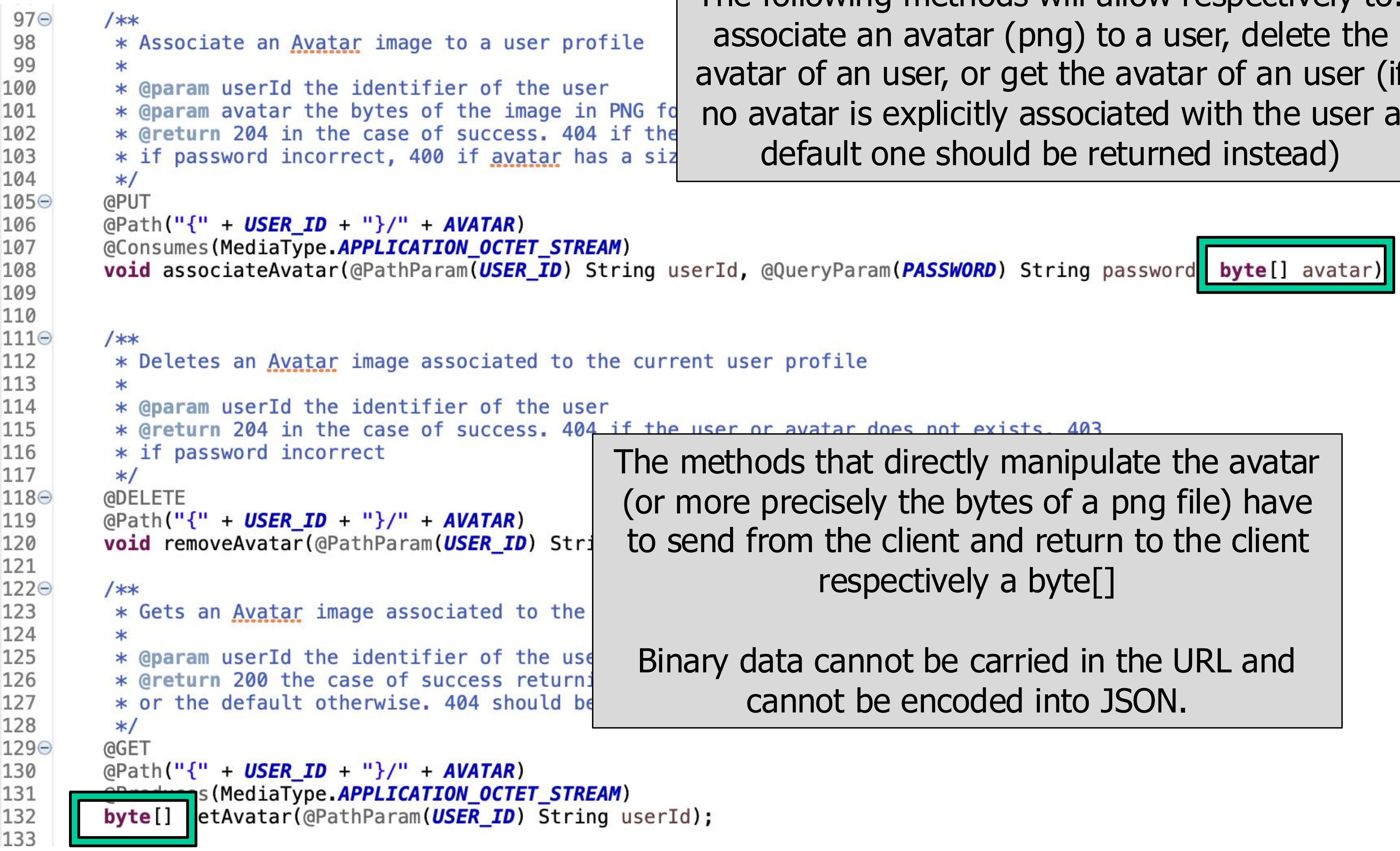
Defining the Service Interface

The following methods will allow respectively to: associate an avatar (png) to a user, delete the avatar of an user, or get the avatar of an user (if no avatar is explicitly associated with the user a default one should be returned instead)



```java
    /**
     * Associate an Avatar image to a user profile
     *
     * @param userId the identifier of the user
     * @param avatar the bytes of the image in PNG fo
     * @return 204 in the case of success. 404 if the
     * if password incorrect, 400 if avatar has a siz
     */
    @PUT
    @Path("{" + USER_ID + "}/" + AVATAR)
    @Consumes(MediaType.APPLICATION_OCTET_STREAM)
    void associateAvatar(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password, byte[] avatar);


    /**
     * Deletes an Avatar image associated to the current user profile
     *
     * @param userId the identifier of the user
     * @return 204 in the case of success. 404 if the user or avatar does not exists, 403
     * if password incorrect
     */
    @DELETE
    @Path("{" + USER_ID + "}/" + AVATAR)
    void removeAvatar(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password);

    /**
     * Gets an Avatar image associated to the current user profile
     *
     * @param userId the identifier of the user
     * @return 200 the case of success returning the bytes of the user image (if one is associated)
     * or the default otherwise. 404 should be returned if the user does not exists
     */
    @GET
    @Path("{" + USER_ID + "}/" + AVATAR)
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    byte[] getAvatar(@PathParam(USER_ID) String userId);

```

DEFINING THE SERVICE INTERFACE



```
    /**
     * Associate an Avatar image to a user profile
     *
     * @param userId the identifier of the user
     * @param avatar the bytes of the image in PNG fo
     * @return 204 in the case of success. 404 if the
     * if password incorrect, 400 if avatar has a siz
     */
    @PUT
    @Path("{" + USER_ID + "}/" + AVATAR)
    @Consumes(MediaType.APPLICATION_OCTET_STREAM)
    void associateAvatar(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password byte[] avatar)


    /**
     * Deletes an Avatar image associated to the current user profile
     *
     * @param userId the identifier of the user
     * @return 204 in the case of success. 404 if the user or avatar does not exists. 403
     * if password incorrect
     */
    @DELETE
    @Path("{" + USER_ID + "}/" + AVATAR)
    void removeAvatar(@PathParam(USER_ID) Stri

    /**
     * Gets an Avatar image associated to the
     *
     * @param userId the identifier of the use
     * @return 200 the case of success returni
     * or the default otherwise. 404 should be
     */
    @GET
    @Path("{" + USER_ID + "}/" + AVATAR)
    s(MediaType.APPLICATION_OCTET_STREAM)
    byte[] etAvatar(@PathParam(USER_ID) String userId);
```

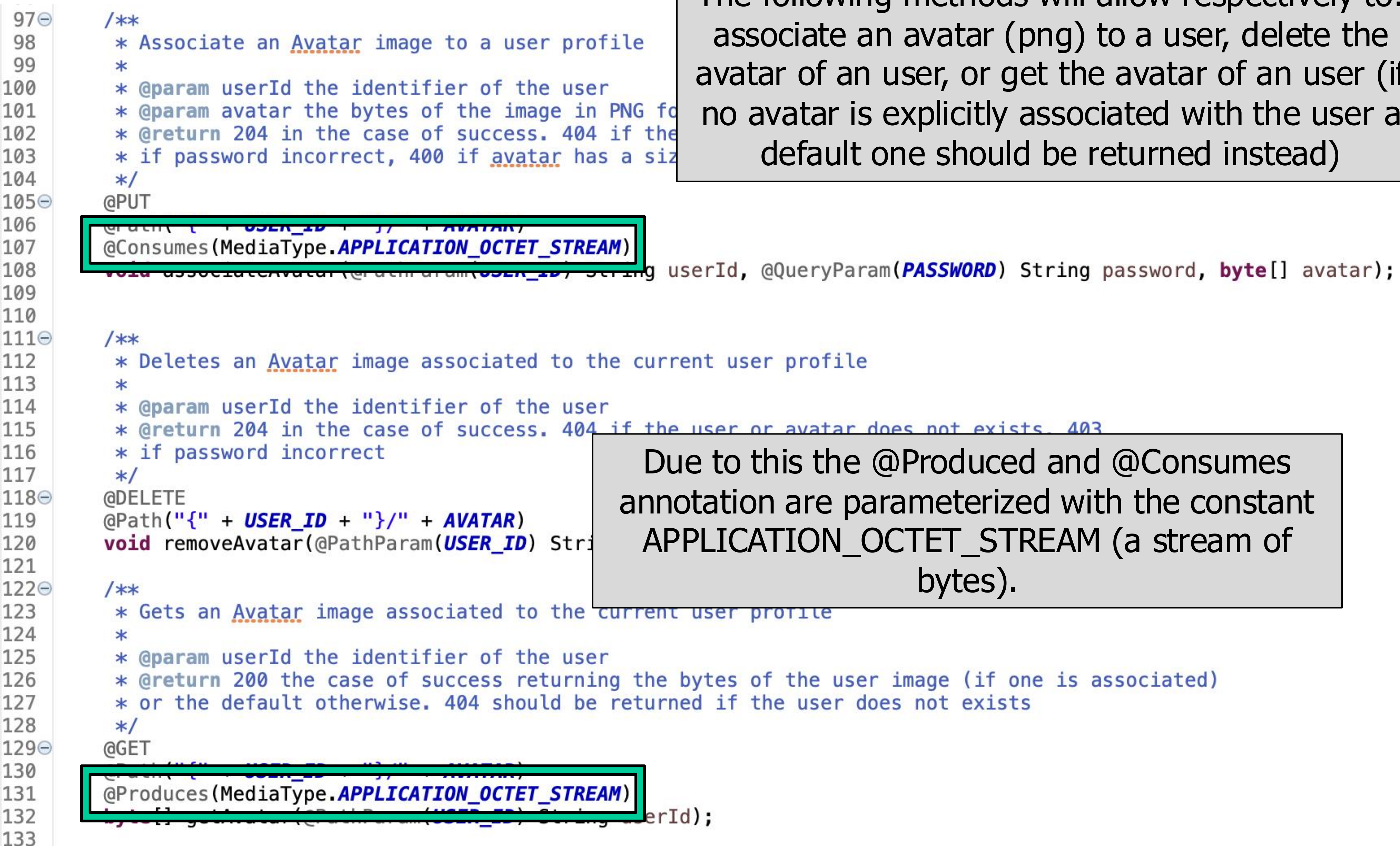
The following methods will allow respectively to: associate an avatar (png) to a user, delete the avatar of an user, or get the avatar of an user (if no avatar is explicitly associated with the user a default one should be returned instead)

The methods that directly manipulate the avatar (or more precisely the bytes of a png file) have to send from the client and return to the client respectively a byte[]

Binary data cannot be carried in the URL and cannot be encoded into JSON.

DEFINING THE SERVICE INTERFACE

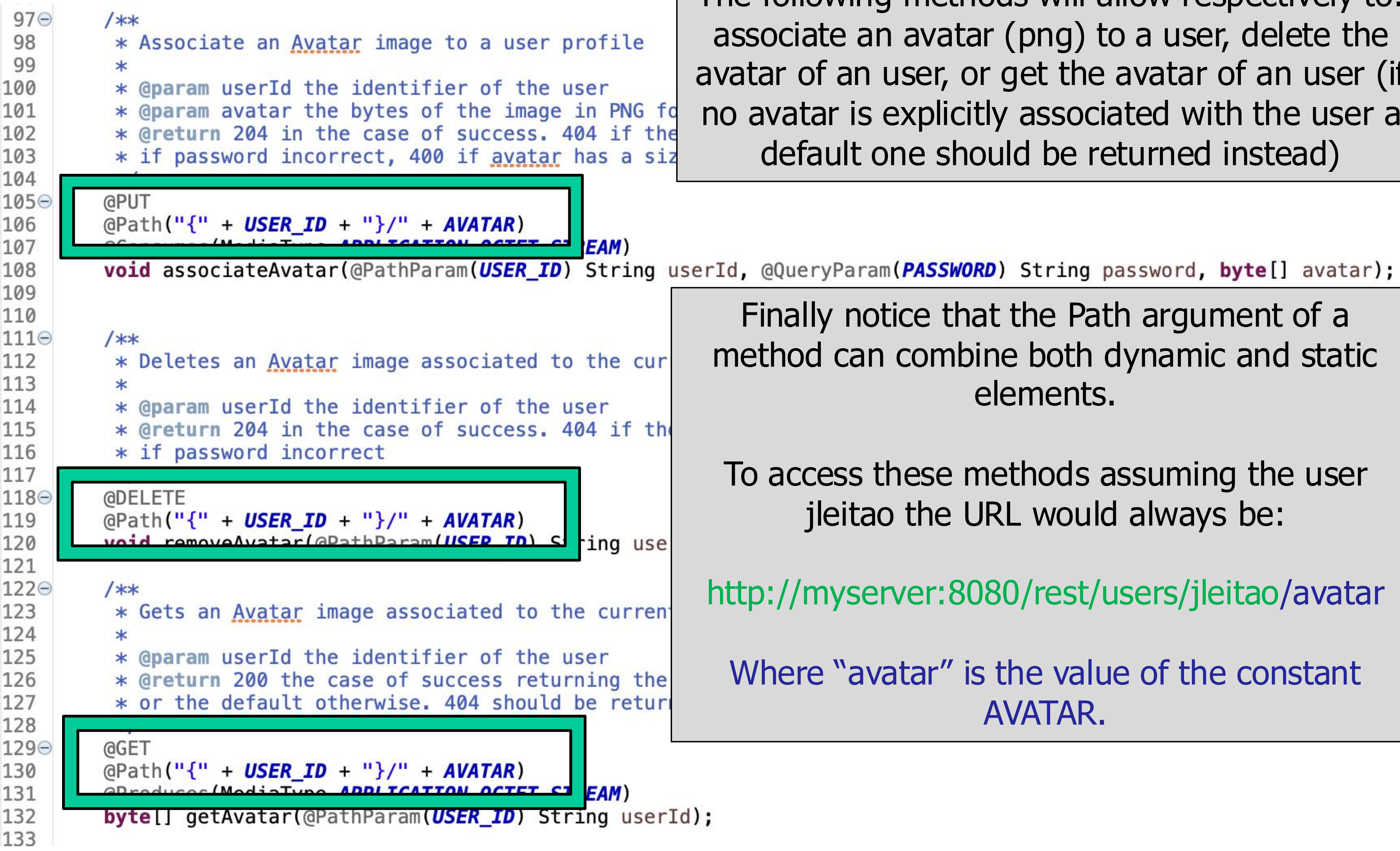
The following methods will allow respectively to: associate an avatar (png) to a user, delete the avatar of an user, or get the avatar of an user (if no avatar is explicitly associated with the user a default one should be returned instead)



```
    /**
     * Associate an Avatar image to a user profile
     *
     * @param userId the identifier of the user
     * @param avatar the bytes of the image in PNG fo
     * @return 204 in the case of success. 404 if the
     * if password incorrect, 400 if avatar has a siz
     */
    @PUT
    @Path("{" + USER_ID + "}/" + AVATAR)
    @Consumes(MediaType.APPLICATION_OCTET_STREAM)
    void associateAvatar(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password, byte[] avatar);


    /**
     * Deletes an Avatar image associated to the current user profile
     *
     * @param userId the identifier of the user
     * @return 204 in the case of success. 404 if the user or avatar does not exists. 403
     * if password incorrect
     */
    @DELETE
    @Path("{" + USER_ID + "}/" + AVATAR)
    void removeAvatar(@PathParam(USER_ID) Stri

    /**
     * Gets an Avatar image associated to the current user profile
     *
     * @param userId the identifier of the user
     * @return 200 the case of success returning the bytes of the user image (if one is associated)
     * or the default otherwise. 404 should be returned if the user does not exists
     */
    @GET
    @Path("{" + USER_ID + "}/" + AVATAR)
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    byte[] getAvatar(@PathParam(USER_ID) String userId);
```

Due to this the @Produced and @Consumes annotation are parameterized with the constant APPLICATION_OCTET_STREAM (a stream of bytes).

DEFINING THE SERVICE INTERFACE



```
    /**
     * Associate an Avatar image to a user profile
     *
     * @param userId the identifier of the user
     * @param avatar the bytes of the image in PNG fo
     * @return 204 in the case of success. 404 if the
     * if password incorrect, 400 if avatar has a siz
     */
    @PUT
    @Path("{" + USER_ID + "}/" + AVATAR)
    @Consumes(MediaType.APPLICATION_OCTET_STREAM)
    void associateAvatar(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password, byte[] avatar);


    /**
     * Deletes an Avatar image associated to the cur
     *
     * @param userId the identifier of the user
     * @return 204 in the case of success. 404 if the
     * if password incorrect
     */
    @DELETE
    @Path("{" + USER_ID + "}/" + AVATAR)
    void removeAvatar(@PathParam(USER_ID) String use

    /**
     * Gets an Avatar image associated to the curren
     *
     * @param userId the identifier of the user
     * @return 200 the case of success returning the
     * or the default otherwise. 404 should be retur
     */
    @GET
    @Path("{" + USER_ID + "}/" + AVATAR)
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    byte[] getAvatar(@PathParam(USER_ID) String userId);

```

The following methods will allow respectively to: associate an avatar (png) to a user, delete the avatar of an user, or get the avatar of an user (if no avatar is explicitly associated with the user a default one should be returned instead)

Finally notice that the Path argument of a method can combine both dynamic and static elements.

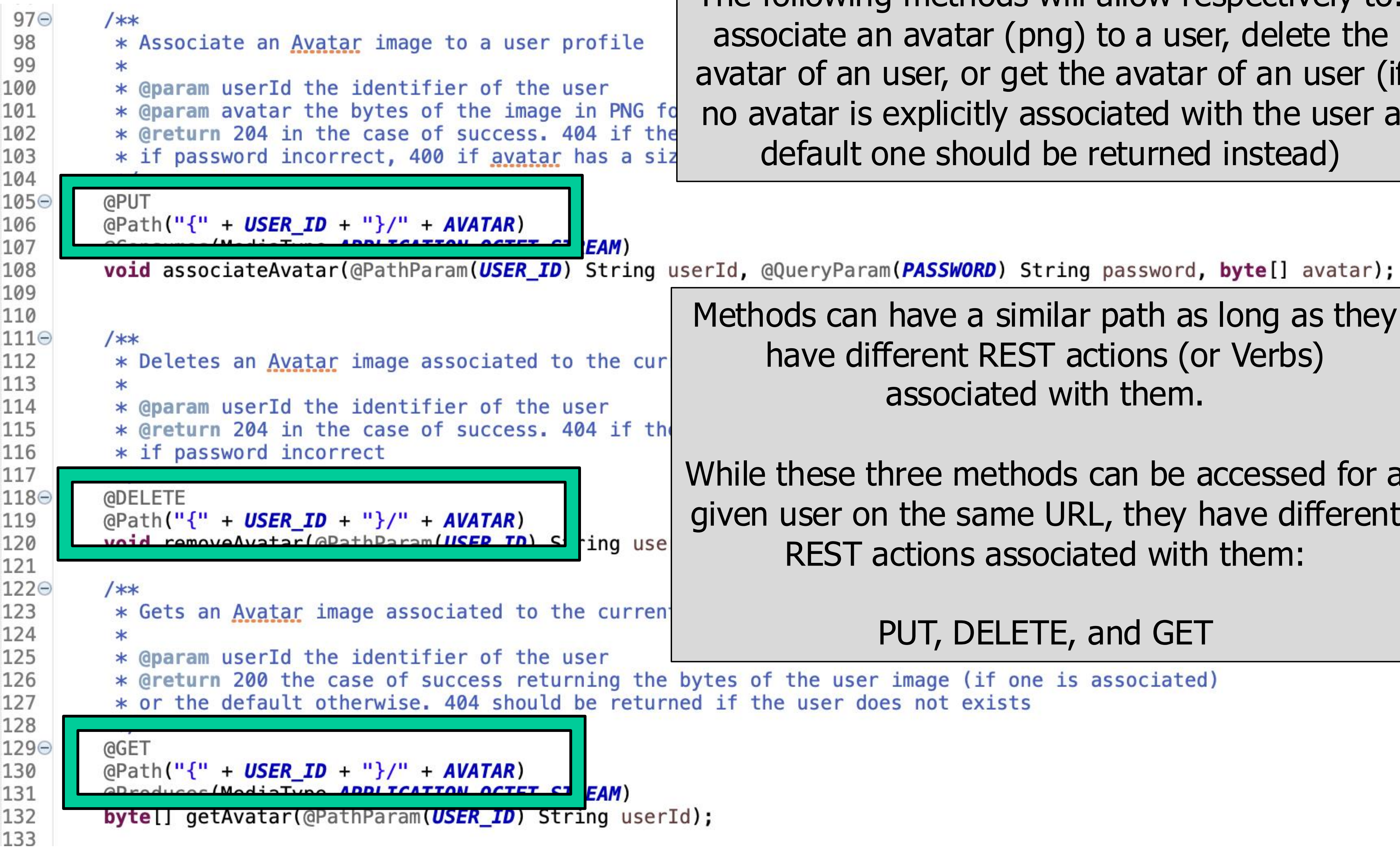
To access these methods assuming the user jleitao the URL would always be:

http://myserver:8080/rest/users/jleitao/avatar

Where "avatar" is the value of the constant AVATAR.

DEFINING THE SERVICE INTERFACE

The following methods will allow respectively to: associate an avatar (png) to a user, delete the avatar of an user, or get the avatar of an user (if no avatar is explicitly associated with the user a default one should be returned instead)



```
    /**
     * Associate an Avatar image to a user profile
     *
     * @param userId the identifier of the user
     * @param avatar the bytes of the image in PNG fo
     * @return 204 in the case of success. 404 if the
     * if password incorrect, 400 if avatar has a siz
     */
    @PUT
    @Path("{" + USER_ID + "}/" + AVATAR)
    @Consumes(MediaType.APPLICATION_OCTET_STREAM)
    void associateAvatar(@PathParam(USER_ID) String userId, @QueryParam(PASSWORD) String password, byte[] avatar);


    /**
     * Deletes an Avatar image associated to the cur
     *
     * @param userId the identifier of the user
     * @return 204 in the case of success. 404 if the
     * if password incorrect
     */
    @DELETE
    @Path("{" + USER_ID + "}/" + AVATAR)
    void removeAvatar(@PathParam(USER_ID) S ing use

    /**
     * Gets an Avatar image associated to the curren
     *
     * @param userId the identifier of the user
     * @return 200 the case of success returning the bytes of the user image (if one is associated)
     * or the default otherwise. 404 should be returned if the user does not exists
     */
    @GET
    @Path("{" + USER_ID + "}/" + AVATAR)
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    byte[] getAvatar(@PathParam(USER_ID) String userId);

```

Methods can have a similar path as long as they have different REST actions (or Verbs) associated with them.

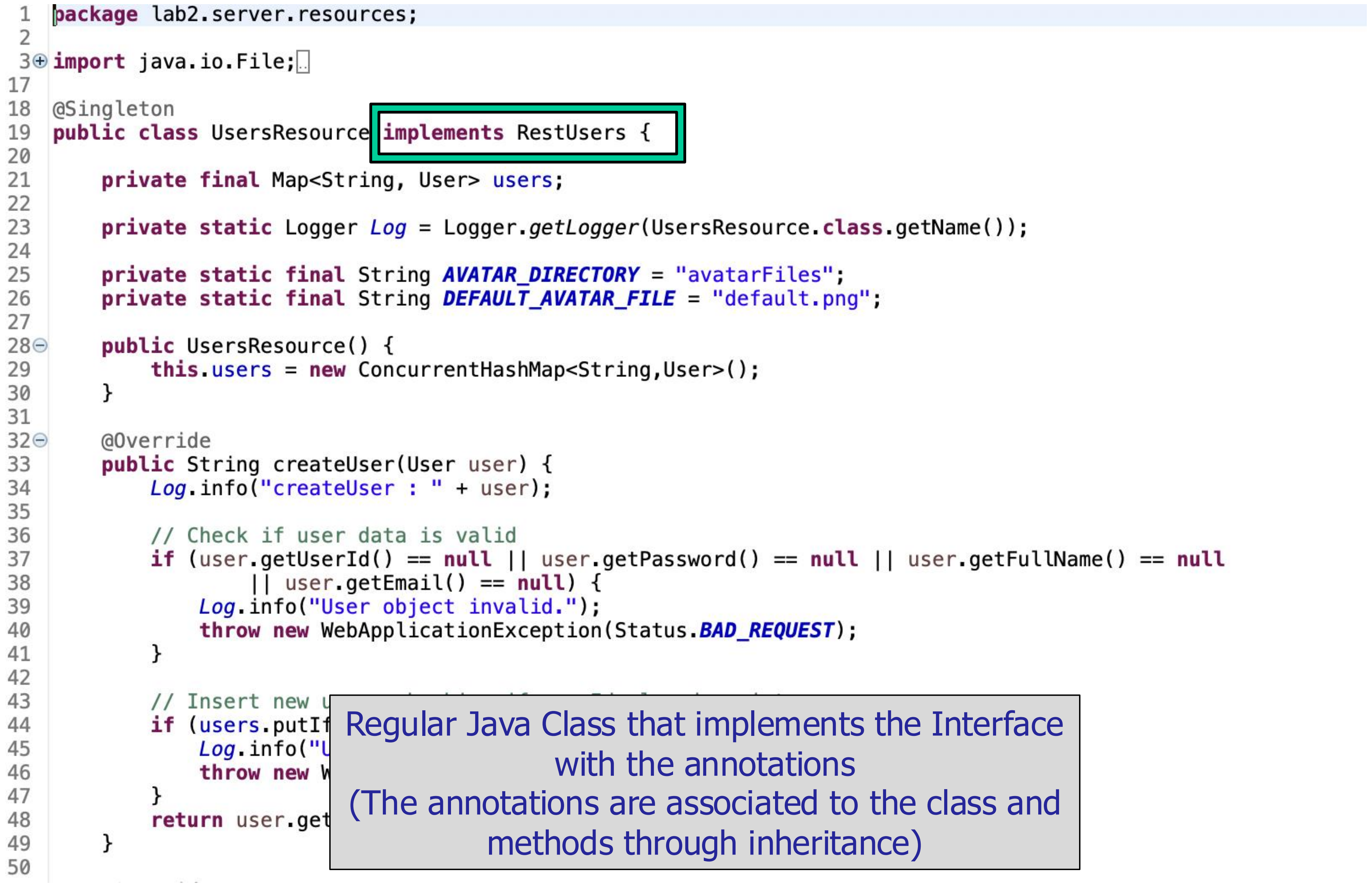
While these three methods can be accessed for a given user on the same URL, they have different REST actions associated with them:

PUT, DELETE, and GET

More about Annotations and Methods

- GET and DELETE are similar.
  - They should avoid to send information in the body of the request (and hence usually do not have a @Consumes Annotation).
- POST and PUT are similar.
  - They should always send a representation of the resource being manipulated in the body of the HTTP request (and hence usually have a @Consumes Annotation).
- GET should always return a representation of a resource.
  - (Therefore, a @Produces Annotation is always present).

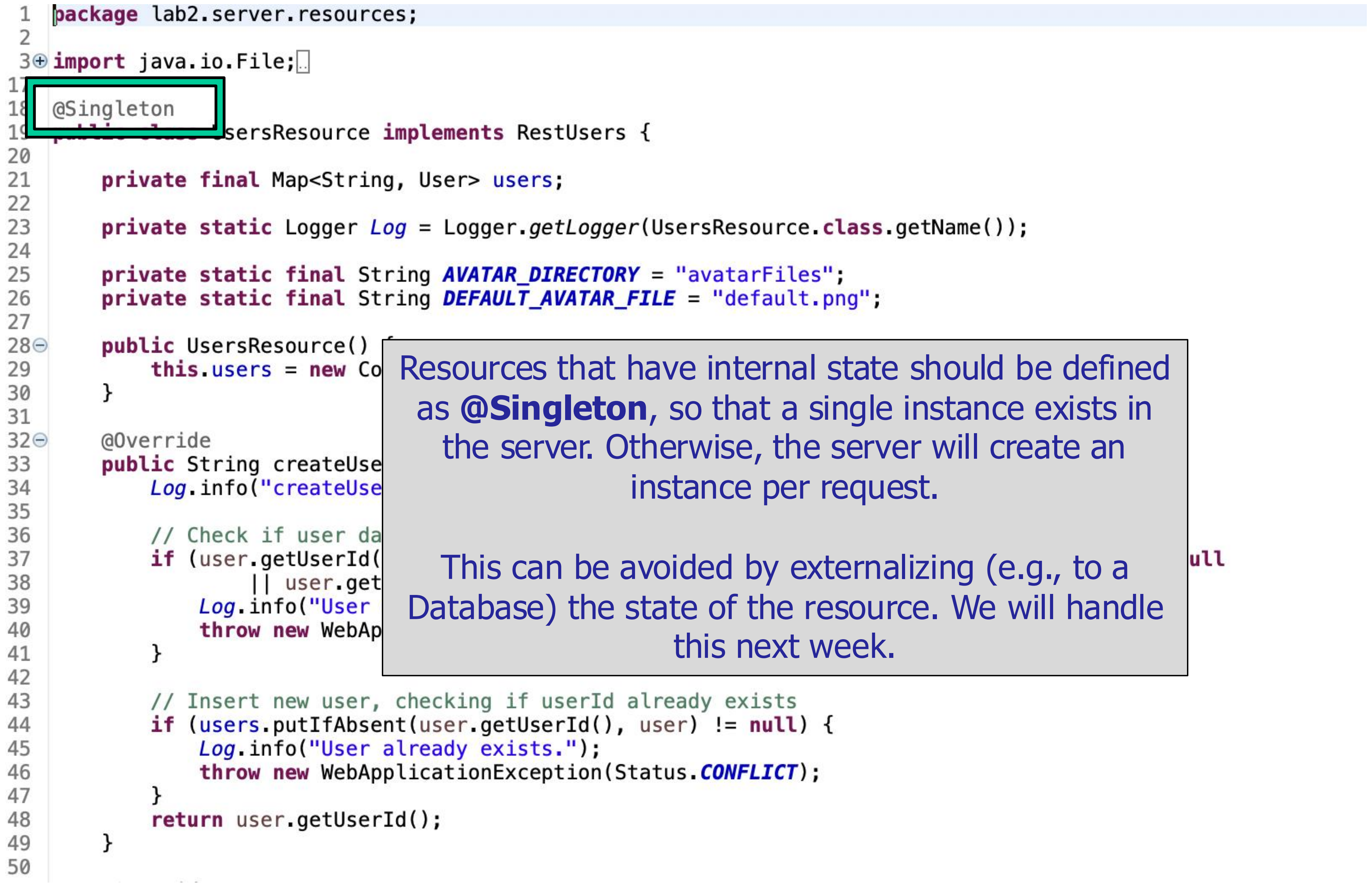
IMPLEMENTING THE SERVICE



```java
package lab2.server.resources;

import java.io.File;

@Singleton
public class UsersResource implements RestUsers {

    private final Map<String, User> users;

    private static Logger Log = Logger.getLogger(UsersResource.class.getName());

    private static final String AVATAR_DIRECTORY = "avatarFiles";
    private static final String DEFAULT_AVATAR_FILE = "default.png";

    public UsersResource() {
        this.users = new ConcurrentHashMap<String,User>();
    }

    @Override
    public String createUser(User user) {
        Log.info("createUser : " + user);

        // Check if user data is valid
        if (user.getUserId() == null || user.getPassword() == null || user.getFullName() == null
                || user.getEmail() == null) {
            Log.info("User object invalid.");
            throw new WebApplicationException(Status.BAD_REQUEST);
        }

        // Insert new u
        if (users.putIf
            Log.info("U
            throw new W
        }
        return user.get
    }
```

Regular Java Class that implements the Interface with the annotations
(The annotations are associated to the class and methods through inheritance)

IMPLEMENTING THE SERVICE



```
package lab2.server.resources;

import java.io.File;

@Singleton
public class UsersResource implements RestUsers {

    private final Map<String, User> users;

    private static Logger Log = Logger.getLogger(UsersResource.class.getName());

    private static final String AVATAR_DIRECTORY = "avatarFiles";
    private static final String DEFAULT_AVATAR_FILE = "default.png";

    public UsersResource() {
        this.users = new Co
    }

    @Override
    public String createUse
        Log.info("createUse

        // Check if user da
        if (user.getUserId(                                                                  ull
                || user.get
            Log.info("User
            throw new WebAp
        }

        // Insert new user, checking if userId already exists
        if (users.putIfAbsent(user.getUserId(), user) != null) {
            Log.info("User already exists.");
            throw new WebApplicationException(Status.CONFLICT);
        }
        return user.getUserId();
    }
```

Resources that have internal state should be defined as **@Singleton**, so that a single instance exists in the server. Otherwise, the server will create an instance per request.

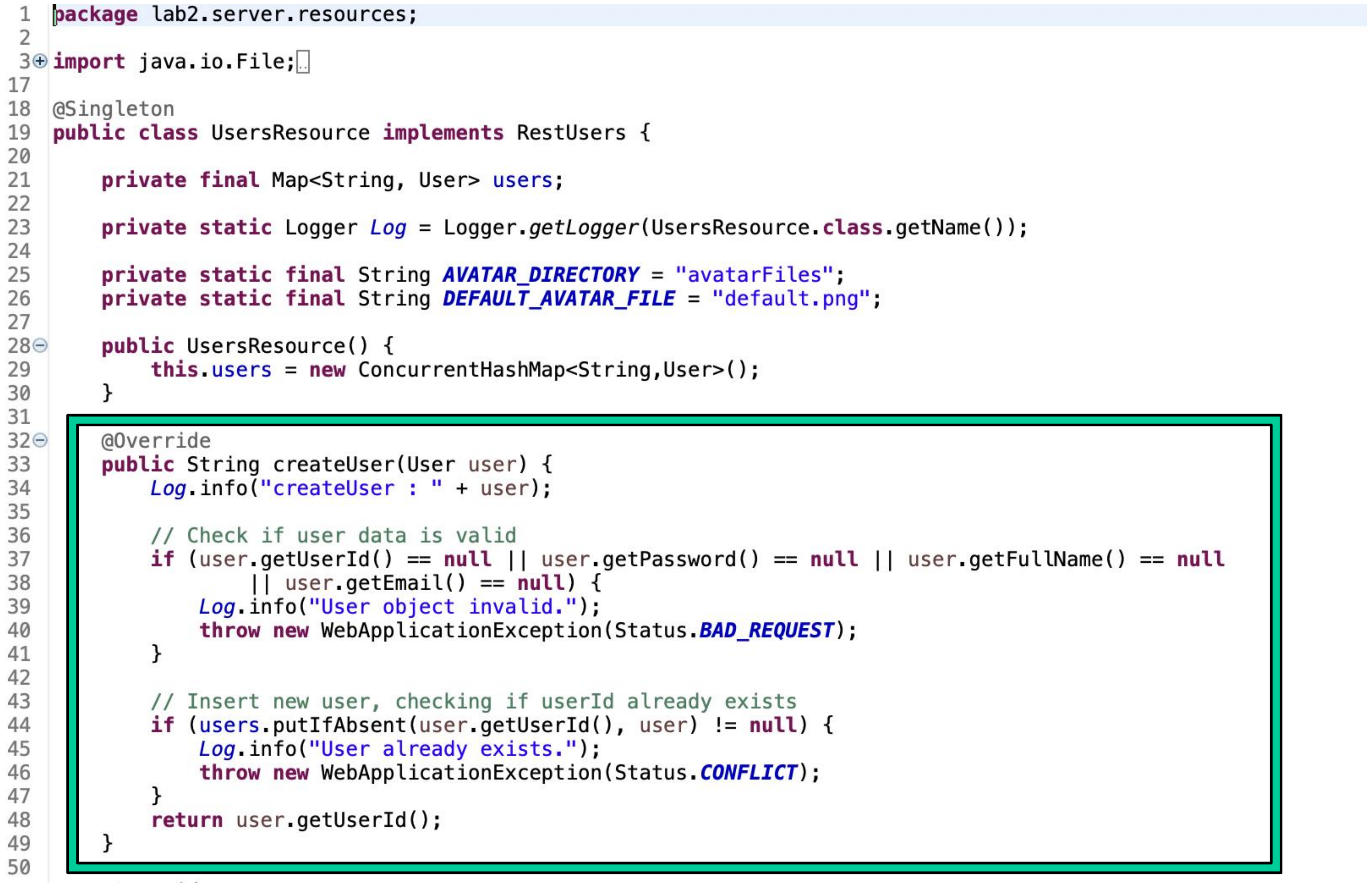
This can be avoided by externalizing (e.g., to a Database) the state of the resource. We will handle this next week.

IMPLEMENTING THE SERVICE

```java
package lab2.server.resources;

import java.io.File;

@Singleton
public class UsersResource implements RestUsers {

    private final Map<String, User> users;

    private static Logger Log = Logger.getLogger(UsersResource.class.getName());

    private static final String AVATAR_DIRECTORY = "avatarFiles";
    private static final String DEFAULT_AVATAR_FILE = "default.png";

    public UsersResource() {
        this.users = new ConcurrentHashMap<String,User>();
    }

    @Override
    public String createUser(User user) {
        Log.info("createUser : " + user);

        // Check if user data is valid
        if (user.getUserId() == null || user.getPassword() == null || user.getFullName() == null
                || user.getEmail() == null) {
            Log.info("User object invalid.");
            throw new WebApplicationException(Status.BAD_REQUEST);
        }

        // Insert new u
        if (users.putIf
            Log.info("U
            throw new W
        }
        return user.get
    }

```

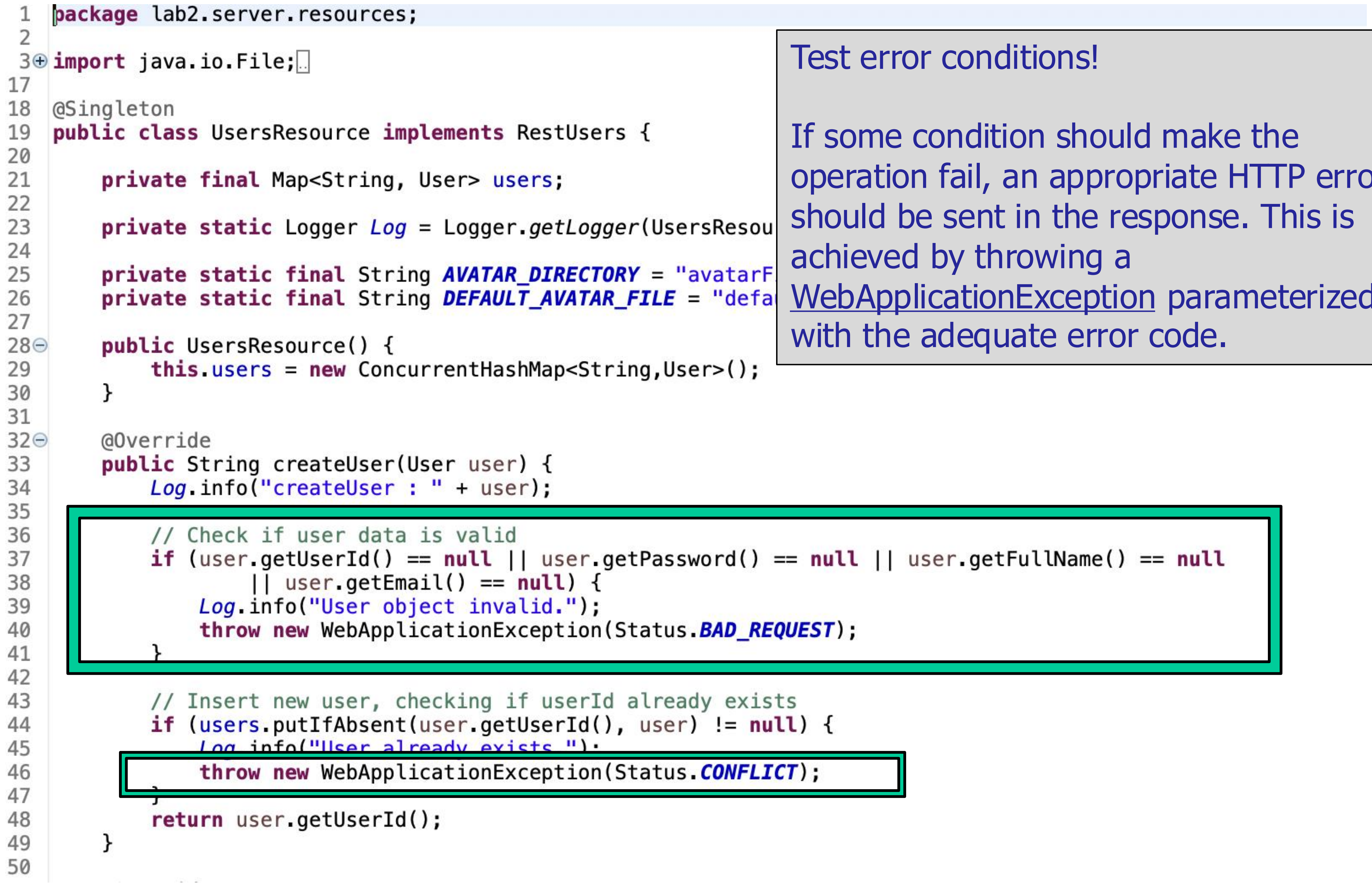
Regular Java Class that implements the Interface with the annotations
(The annotations are associated to the class and methods through inheritance)

Implementing the Service: Create User



```java
package lab2.server.resources;

import java.io.File;

@Singleton
public class UsersResource implements RestUsers {

    private final Map<String, User> users;

    private static Logger Log = Logger.getLogger(UsersResource.class.getName());

    private static final String AVATAR_DIRECTORY = "avatarFiles";
    private static final String DEFAULT_AVATAR_FILE = "default.png";

    public UsersResource() {
        this.users = new ConcurrentHashMap<String,User>();
    }

    @Override
    public String createUser(User user) {
        Log.info("createUser : " + user);

        // Check if user data is valid
        if (user.getUserId() == null || user.getPassword() == null || user.getFullName() == null
                || user.getEmail() == null) {
            Log.info("User object invalid.");
            throw new WebApplicationException(Status.BAD_REQUEST);
        }

        // Insert new user, checking if userId already exists
        if (users.putIfAbsent(user.getUserId(), user) != null) {
            Log.info("User already exists.");
            throw new WebApplicationException(Status.CONFLICT);
        }
        return user.getUserId();
    }

```

IMPLEMENTING THE SERVICE: CREATE USER



```java
package lab2.server.resources;

import java.io.File;

@Singleton
public class UsersResource implements RestUsers {

    private final Map<String, User> users;

    private static Logger Log = Logger.getLogger(UsersResou

    private static final String AVATAR_DIRECTORY = "avatarF
    private static final String DEFAULT_AVATAR_FILE = "defa

    public UsersResource() {
        this.users = new ConcurrentHashMap<String,User>();
    }

    @Override
    public String createUser(User user) {
        Log.info("createUser : " + user);

        // Check if user data is valid
        if (user.getUserId() == null || user.getPassword() == null || user.getFullName() == null
                || user.getEmail() == null) {
            Log.info("User object invalid.");
            throw new WebApplicationException(Status.BAD_REQUEST);
        }

        // Insert new user, checking if userId already exists
        if (users.putIfAbsent(user.getUserId(), user) != null) {
            Log.info("User already exists.");
            throw new WebApplicationException(Status.CONFLICT);
        }
        return user.getUserId();
    }
```

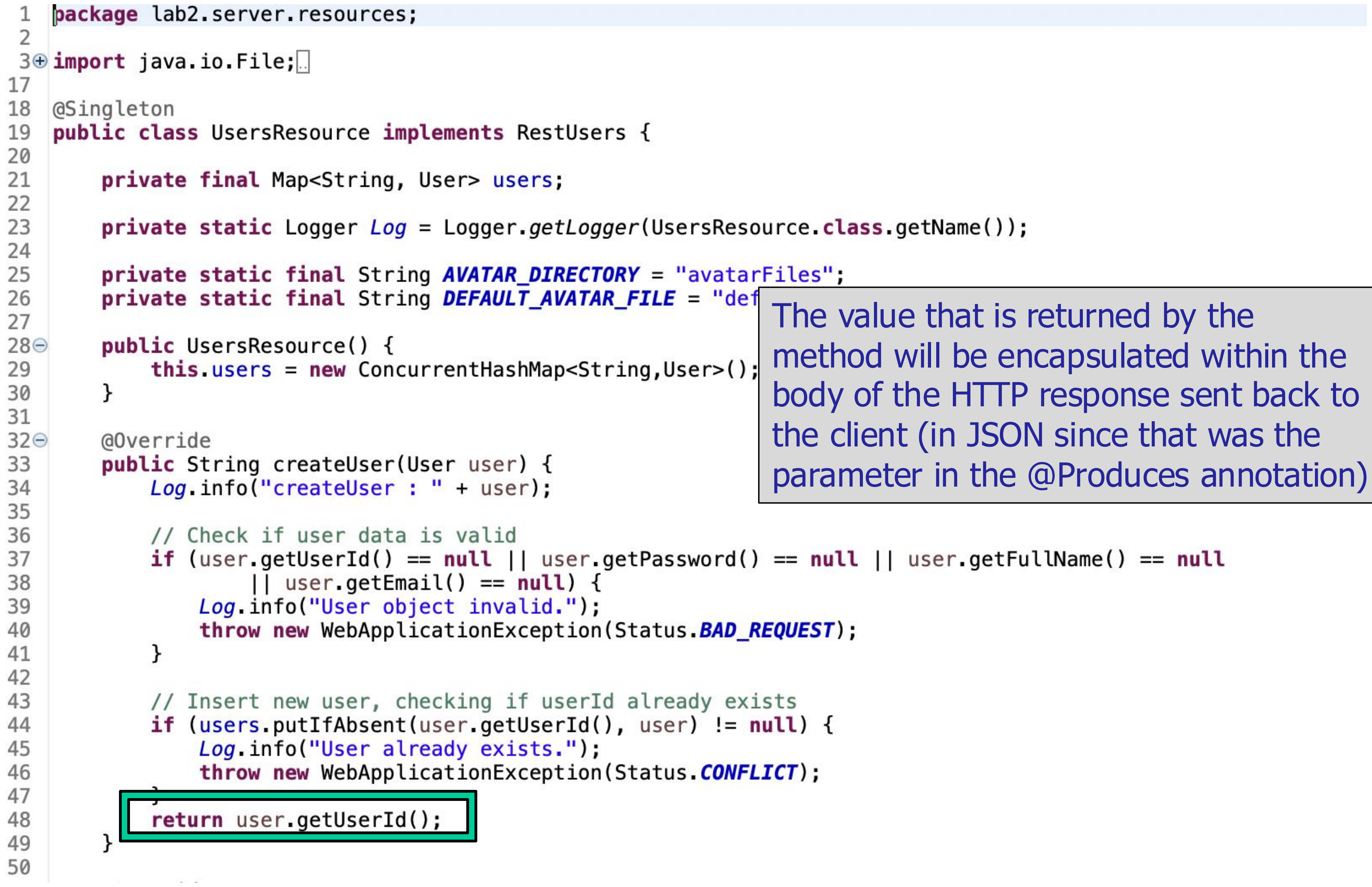
Test error conditions!

If some condition should make the operation fail, an appropriate HTTP error should be sent in the response. This is achieved by throwing a WebApplicationException parameterized with the adequate error code.

IMPORTANT HTTP RESPONSE CODES

- Range 100 – 199: Information (rarely seen)
- Range 200 – 299: Success
  - 200: OK (the operation was successful, and the reply contains information)
  - 204: No Content (the operation was successful but there is no information returned).
- Range 300 – 399: Redirection: additional action is required
  - 301: Moved Permanently (the resource is now represented by a new URL, which is provided in this answer)
- Range 400 – 499: Client Error (e.g., preparing request)
  - 404: Page/Resource not found
  - 409: Conflict – executing the request violates logic rules
- Range 500 – 599: Server Error
  - 500: Internal Server Error – usually means an unhandled exception was thrown while executing request

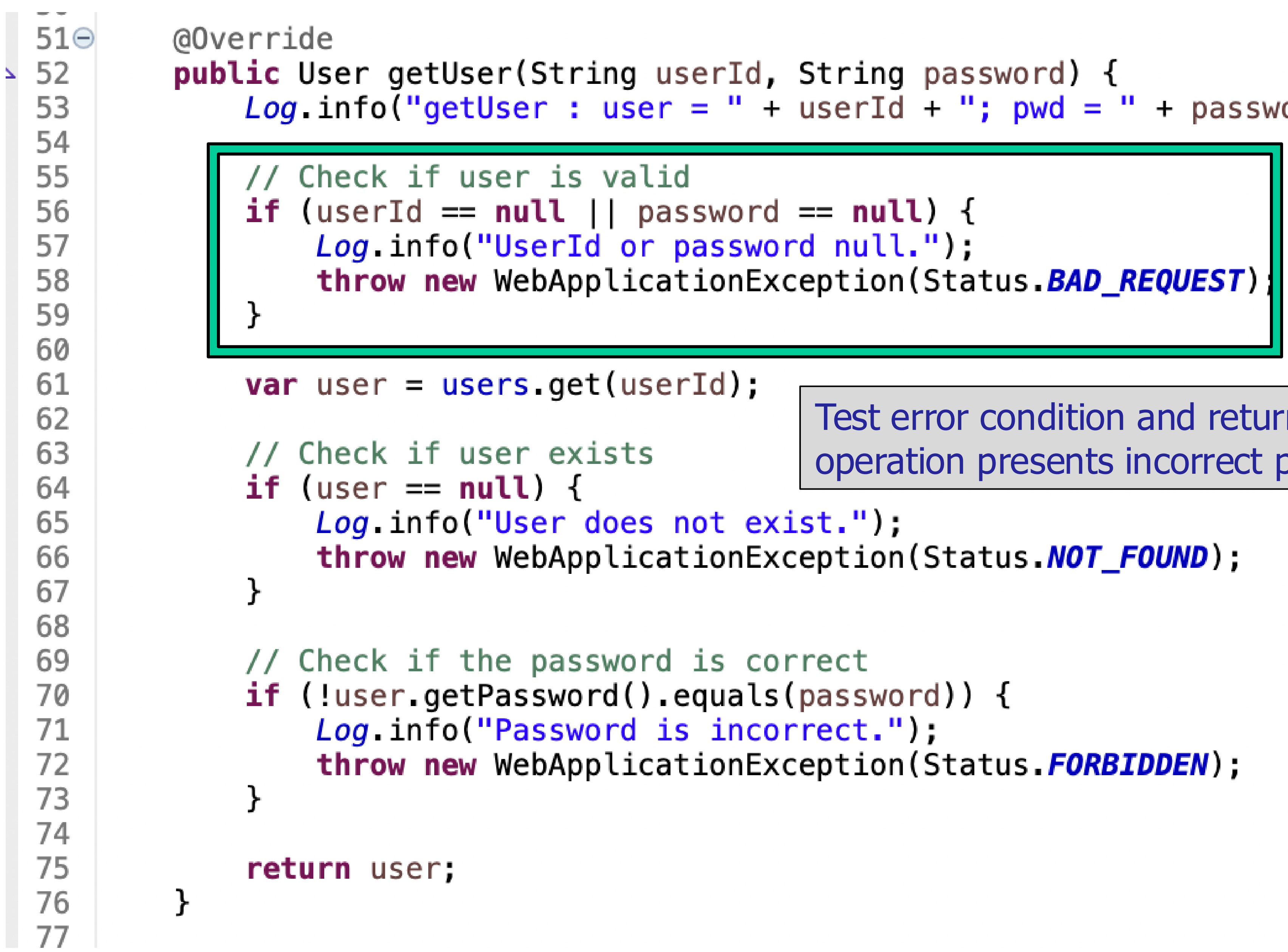
IMPLEMENTING THE SERVICE: POST MESSAGE



```java
package lab2.server.resources;

import java.io.File;

@Singleton
public class UsersResource implements RestUsers {

    private final Map<String, User> users;

    private static Logger Log = Logger.getLogger(UsersResource.class.getName());

    private static final String AVATAR_DIRECTORY = "avatarFiles";
    private static final String DEFAULT_AVATAR_FILE = "def

    public UsersResource() {
        this.users = new ConcurrentHashMap<String,User>();
    }

    @Override
    public String createUser(User user) {
        Log.info("createUser : " + user);

        // Check if user data is valid
        if (user.getUserId() == null || user.getPassword() == null || user.getFullName() == null
                || user.getEmail() == null) {
            Log.info("User object invalid.");
            throw new WebApplicationException(Status.BAD_REQUEST);
        }

        // Insert new user, checking if userId already exists
        if (users.putIfAbsent(user.getUserId(), user) != null) {
            Log.info("User already exists.");
            throw new WebApplicationException(Status.CONFLICT);
        }
        return user.getUserId();
    }

```

The value that is returned by the method will be encapsulated within the body of the HTTP response sent back to the client (in JSON since that was the parameter in the @Produces annotation)

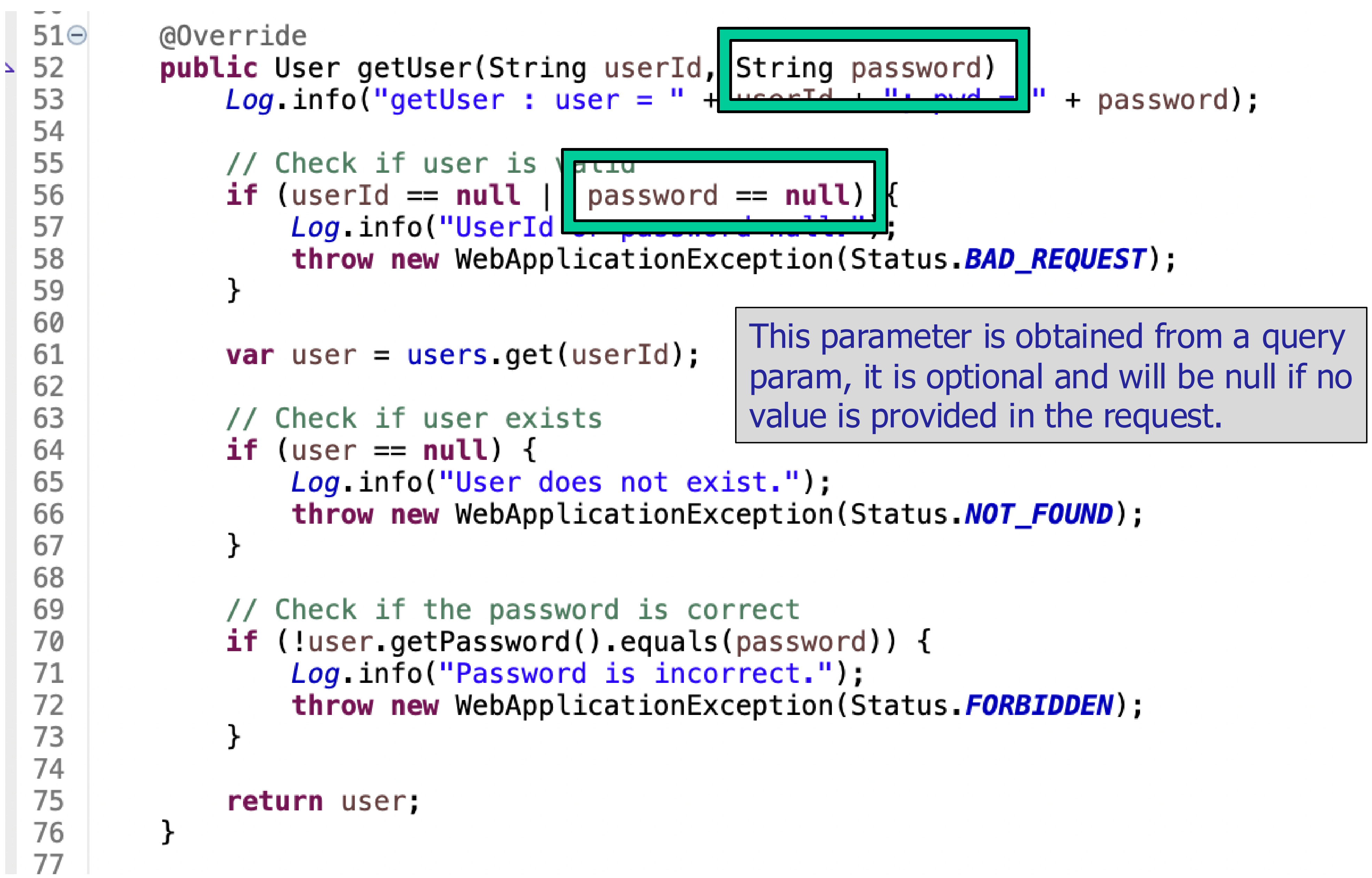
Implementing the Service: Get User



```java
    @Override
    public User getUser(String userId, String password) {
        Log.info("getUser : user = " + userId + "; pwd = " + password);

        // Check if user is valid
        if (userId == null || password == null) {
            Log.info("UserId or password null.");
            throw new WebApplicationException(Status.BAD_REQUEST);
        }

        var user = users.get(userId);

        // Check if user exists
        if (user == null) {
            Log.info("User does not exist.");
            throw new WebApplicationException(Status.NOT_FOUND);
        }

        // Check if the password is correct
        if (!user.getPassword().equals(password)) {
            Log.info("Password is incorrect.");
            throw new WebApplicationException(Status.FORBIDDEN);
        }

        return user;
    }
```

Test error condition and return 400 if the operation presents incorrect parameters.

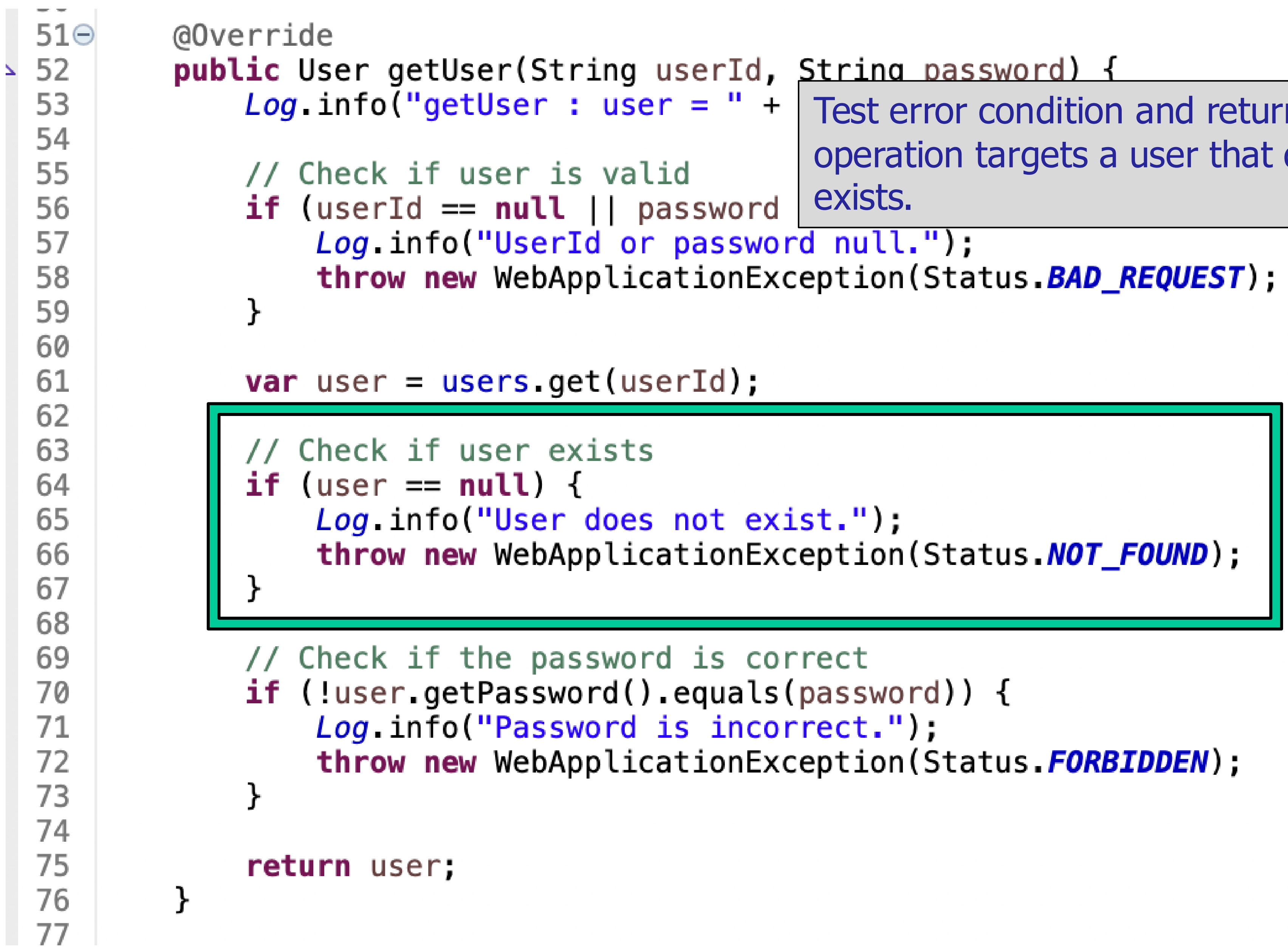
Implementing the Service: Get User



```java
    @Override
    public User getUser(String userId, String password) {
        Log.info("getUser : user = " + userId + "; pwd = " + password);

        // Check if user is valid
        if (userId == null || password == null) {
            Log.info("UserId or password null.");
            throw new WebApplicationException(Status.BAD_REQUEST);
        }

        var user = users.get(userId);

        // Check if user exists
        if (user == null) {
            Log.info("User does not exist.");
            throw new WebApplicationException(Status.NOT_FOUND);
        }

        // Check if the password is correct
        if (!user.getPassword().equals(password)) {
            Log.info("Password is incorrect.");
            throw new WebApplicationException(Status.FORBIDDEN);
        }

        return user;
    }
```

This parameter is obtained from a query param, it is optional and will be null if no value is provided in the request.

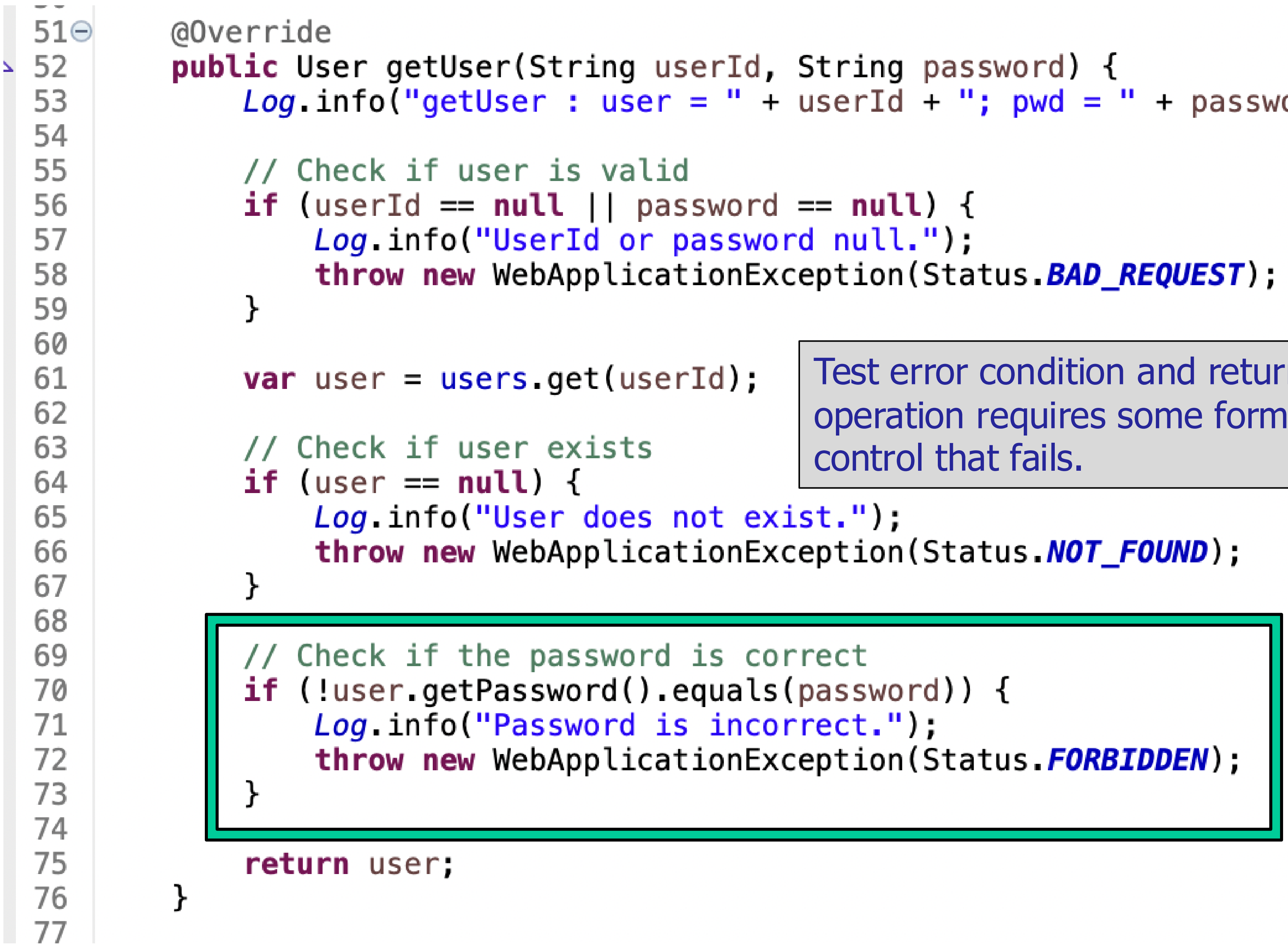
IMPLEMENTING THE SERVICE: GET USER



```
    @Override
    public User getUser(String userId, String password) {
        Log.info("getUser : user = " +

        // Check if user is valid
        if (userId == null || password
            Log.info("UserId or password null.");
            throw new WebApplicationException(Status.BAD_REQUEST);
        }

        var user = users.get(userId);

        // Check if user exists
        if (user == null) {
            Log.info("User does not exist.");
            throw new WebApplicationException(Status.NOT_FOUND);
        }

        // Check if the password is correct
        if (!user.getPassword().equals(password)) {
            Log.info("Password is incorrect.");
            throw new WebApplicationException(Status.FORBIDDEN);
        }

        return user;
    }
```

Test error condition and return 404 if the operation targets a user that does not exists.

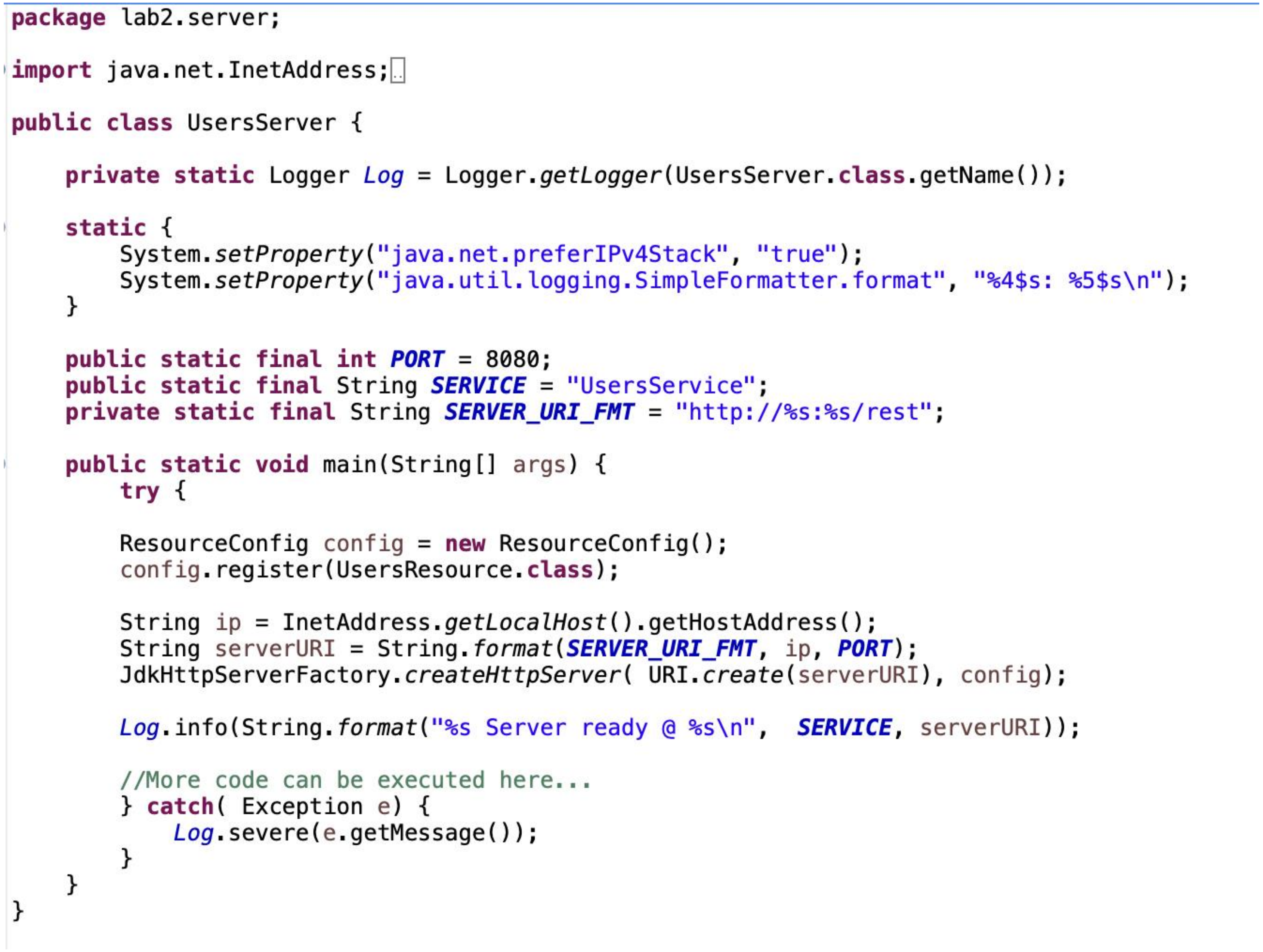
IMPLEMENTING THE SERVICE: GET USER



```java
    @Override
    public User getUser(String userId, String password) {
        Log.info("getUser : user = " + userId + "; pwd = " + password);

        // Check if user is valid
        if (userId == null || password == null) {
            Log.info("UserId or password null.");
            throw new WebApplicationException(Status.BAD_REQUEST);
        }

        var user = users.get(userId);

        // Check if user exists
        if (user == null) {
            Log.info("User does not exist.");
            throw new WebApplicationException(Status.NOT_FOUND);
        }

        // Check if the password is correct
        if (!user.getPassword().equals(password)) {
            Log.info("Password is incorrect.");
            throw new WebApplicationException(Status.FORBIDDEN);
        }

        return user;
    }
```

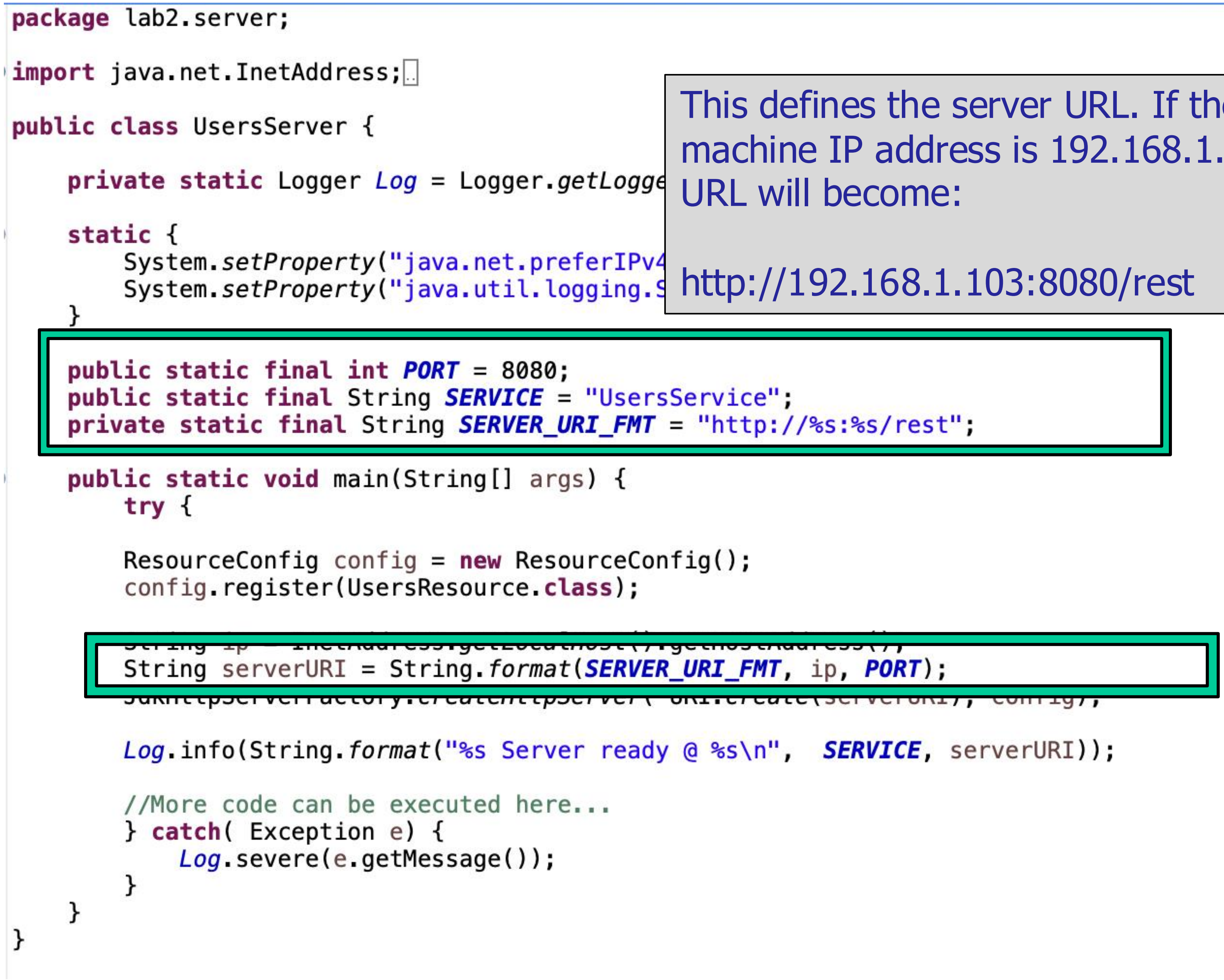
Test error condition and return 403 if the operation requires some form of access control that fails.

IMPLEMENTING THE SERVICE: SERVER CODE (MAIN)



```java
package lab2.server;

import java.net.InetAddress;

public class UsersServer {

    private static Logger Log = Logger.getLogger(UsersServer.class.getName());

    static {
        System.setProperty("java.net.preferIPv4Stack", "true");
        System.setProperty("java.util.logging.SimpleFormatter.format", "%4$s: %5$s\n");
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "UsersService";
    private static final String SERVER_URI_FMT = "http://%s:%s/rest";

    public static void main(String[] args) {
        try {

        ResourceConfig config = new ResourceConfig();
        config.register(UsersResource.class);

        String ip = InetAddress.getLocalHost().getHostAddress();
        String serverURI = String.format(SERVER_URI_FMT, ip, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config);

        Log.info(String.format("%s Server ready @ %s\n",  SERVICE, serverURI));

        //More code can be executed here...
        } catch( Exception e) {
            Log.severe(e.getMessage());
        }
    }
}
```

IMPLEMENTING THE SERVICE: SERVER CODE (MAIN)

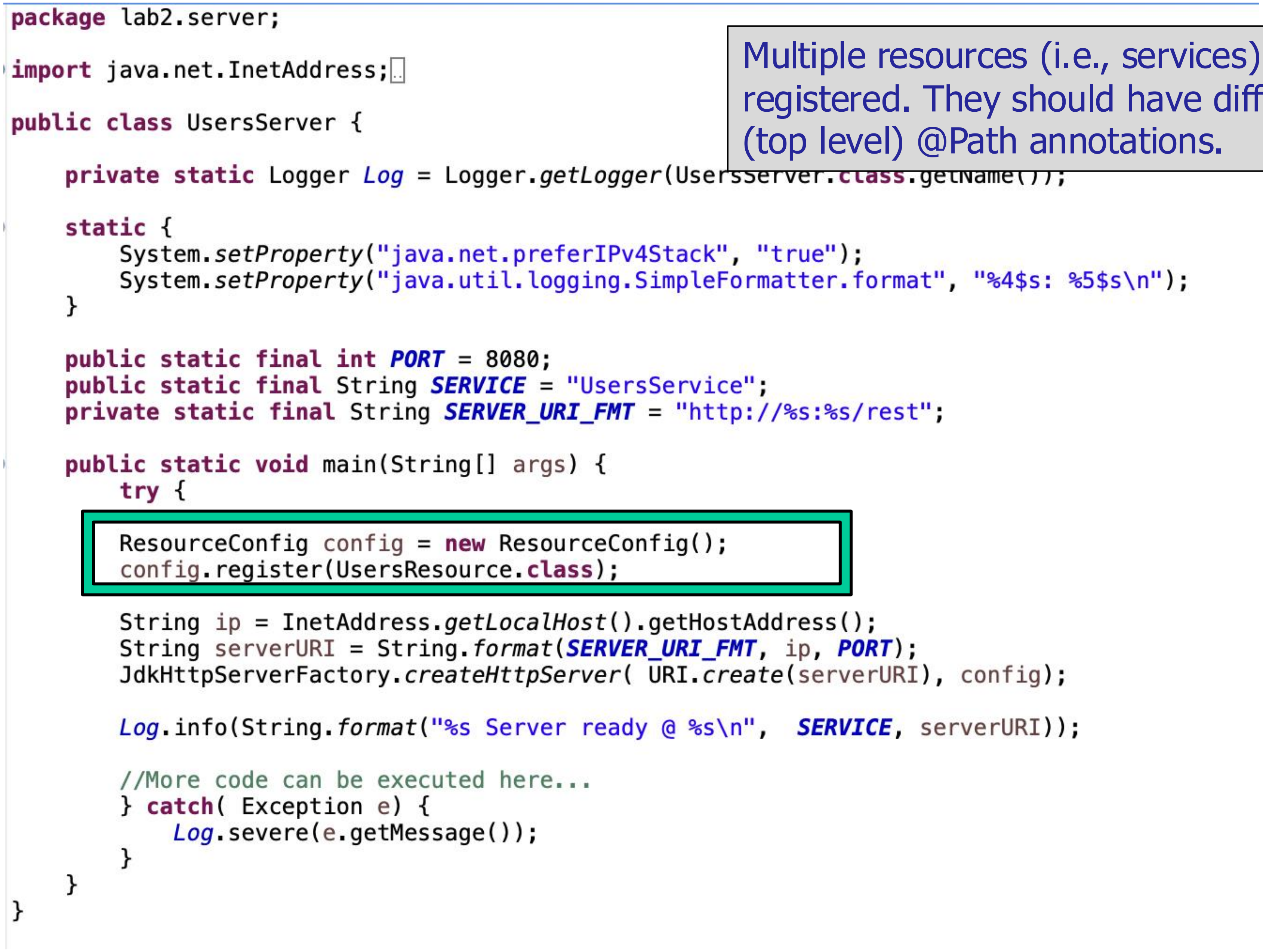


```
package lab2.server;

import java.net.InetAddress;

public class UsersServer {

    private static Logger Log = Logger.getLogge

    static {
        System.setProperty("java.net.preferIPv4
        System.setProperty("java.util.logging.S
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "UsersService";
    private static final String SERVER_URI_FMT = "http://%s:%s/rest";

    public static void main(String[] args) {
        try {

        ResourceConfig config = new ResourceConfig();
        config.register(UsersResource.class);

        String ip = InetAddress.getLocalHost().getHostAddress();
        String serverURI = String.format(SERVER_URI_FMT, ip, PORT);
        JdkHttpServerFactory.createJdkHttpServer( URI.create(serverURI), config);

        Log.info(String.format("%s Server ready @ %s\n",  SERVICE, serverURI));

        //More code can be executed here...
        } catch( Exception e) {
            Log.severe(e.getMessage());
        }
    }
}
```

This defines the server URL. If the machine IP address is 192.168.1.103 the URL will become:

http://192.168.1.103:8080/rest

Implementing the Service: Server Code (Main)



```
package lab2.server;

import java.net.InetAddress;

public class UsersServer {

    private static Logger Log = Logger.getLogger(UsersServer.class.getName());

    static {
        System.setProperty("java.net.preferIPv4Stack", "true");
        System.setProperty("java.util.logging.SimpleFormatter.format", "%4$s: %5$s\n");
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "UsersService";
    private static final String SERVER_URI_FMT = "http://%s:%s/rest";

    public static void main(String[] args) {
        try {

        ResourceConfig config = new ResourceConfig();
        config.register(UsersResource.class);

        String ip = InetAddress.getLocalHost().getHostAddress();
        String serverURI = String.format(SERVER_URI_FMT, ip, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config);

        Log.info(String.format("%s Server ready @ %s\n",  SERVICE, serverURI));

        //More code can be executed here...
        } catch( Exception e) {
            Log.severe(e.getMessage());
        }
    }
}
```

Multiple resources (i.e., services) can be registered. They should have different (top level) @Path annotations.

Implementing the Service: Server Code (Main)

This effectively starts the server (with their own threads to handle client requests).

```java
package lab2.server;

import java.net.InetAddress;

public class UsersServer {

    private static Logger Log = Logger.getLogger(UsersServer.class.getName());

    static {
        System.setProperty("java.net.preferIPv4Stack", "true");
        System.setProperty("java.util.logging.SimpleFormatter.format", "%4$s: %5$s\n");
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "UsersService";
    private static final String SERVER_URI_FMT = "http://%s:%s/rest";

    public static void main(String[] args) {
        try {

        ResourceConfig config = new ResourceConfig();
        config.register(UsersResource.class);

        String ip = InetAddress.getLocalHost().getHostAddress();
        String serverURI = String.format(SERVER_URI_FMT, ip, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config);

        Log.info(String.format("%s Server ready @ %s\n",  SERVICE, serverURI));

        //More code can be executed here...
        } catch( Exception e) {
            Log.severe(e.getMessage());
        }
    }
}
```

GOALS

In the end of this lab you should be able to:

- Understand what a WebService REST is
- Know how to develop a WS REST and Server in Java (using JAX-RS)
- **Know how to develop a REST Client in Java (using JAX-RX)**
- Use Docker to test your service using your clients

Implementing the Client: CreateUserClient (1/2)

```java
public class CreateUserClient {

    public static void main(String[] args) throws IOException {
        if( args.length != 5) {
            System.err.println( "Use: java " + CreateUserClient.class.getCanonicalName() + " url userId fullName email password");
            return;
        }

        String serverUrl = args[0];
        String userId = args[1];
        String fullName = args[2];
        String email = args[3];
        String password = args[4];

        User usr = new User( userId, fullName, email, password);
        System.out.println("Sending request to server.");
        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( RestUsers.PATH );

        Response r = target.request()
                .accept(MediaType.APPLICATION_JSON)
                .post(Entity.entity(usr, MediaType.APPLICATION_JSON));

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
            System.out.println("Success, created user with id: " + r.readEntity(String.class) );
        else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

}
```

The first part of the client only gathers the multiple parameters from the user provided in the command line. And creates a User instance.

The interesting part is after this

IMPLEMENTING THE CLIENT: CREATEUSERCLIENT (2/2)

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);

WebTarget target = client.target( serverUrl ).path( RestUsers.PATH );

Response r = target.request()
        .accept(MediaType.APPLICATION_JSON)
        .post(Entity.entity(usr, MediaType.APPLICATION_JSON));

if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
    System.out.println("Success, created user with id: " + r.readEntity(String.class) );
else
    System.out.println("Error, HTTP error status: " + r.getStatus() );
```

We start by creating a ClientConfig (later on, this can be used to control the behavior of the client) and from that generate an instance of a Client.

IMPLEMENTING THE CLIENT: CREATEUSERCLIENT (2/2)

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);

WebTarget target = client.target( serverUrl ).path( RestUsers.PATH );

Response r = target.request()
        .accept(MediaType.APPLICATION_JSON)
        .post(Entity.entity(usr, MediaType.APPLICATION_JSON));

if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
    System.out.println("Success, created user with id: " + r.readEntity(String.class) );
else
    System.out.println("Error, HTTP error status: " + r.getStatus() );
```

We then create a WebTarget instance, whose base target is the server URL (e.g., http://192.168.1.103:8080/rest). We then can concatenate any number of other elements to the URL. Here we are just adding the path corresponding to the service (enclosed in the top level @Path annotation of the users service): e.g., http://192.168.1.103:8080/rest/users

IMPLEMENTING THE CLIENT: CREATEUSERCLIENT (2/2)

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);

WebTarget target = client.target( serverUrl ).path( RestUsers.PATH );

Response r = target.request()
        .accept(MediaType.APPLICATION_JSON)
        .post(Entity.entity(usr, MediaType.APPLICATION_JSON));

if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
    System.out.println("Success, created user with id: " + r.readEntity(String.class) );
else
    System.out.println("Error, HTTP error status: " + r.getStatus() );
```

From the target, we create a request, which we parameterize with the .accept() method to state what is the format in which we can receive the return value in the body of the HTTP response (must match the @Produces annotation on the server).

This is optional and is only performed when the endpoint returns some value.

IMPLEMENTING THE CLIENT: CREATEUSERCLIENT (2/2)

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);

WebTarget target = client.target( serverUrl ).path( RestUsers.PATH );

Response r = target.request()
        .accept(MediaType.APPLICATION_JSON)
        .post(Entity.entity(usr, MediaType.APPLICATION_JSON));

if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
    System.out.println("Success, created user with id: " + r.readEntity(String.class) );
else
    System.out.println("Error, HTTP error status: " + r.getStatus() );
```

Finally, we execute the post method, because the endpoint were are trying to use is a POST HTTP operation. As an argument we can encode the parameter that is passed in the body of the HTTP request using the Entity class. The second argument must match the annotation @Consumes on the server side. The argument of Post is optional.

The invocation of post effectively executes the request to the server and waits for a response to be returned.

IMPLEMENTING THE CLIENT: CREATEUSERCLIENT (2/2)

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);

WebTarget target = client.target( serverUrl ).path( RestUsers.PATH );

Response r = target.request()
        .accept(MediaType.APPLICATION_JSON)
        .post(Entity.entity(usr, MediaType.APPLICATION_JSON));
```

```
if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
    System.out.println("Success, created user with id: " + r.readEntity(String.class) );
else
    System.out.println("Error, HTTP error status: " + r.getStatus() );
```

We can now process the reply received from the server. We start by checking the HTTP response code (OK – 200), and check if the body of the reply contains an object.

If so, we access the contents of the body with the method readEntity, which is parameterized with the class we want to read from the body (in this case String, the identifier of the user that was created).

If the request failed, we print the HTTP response code.

IMPLEMENTING THE CLIENT: GETUSERCLIENT

```java
public class GetUserClient {


    public static void main(String[] args) throws IOException {

        if( args.length != 3) {
            System.err.println( "Use: java " + CreateUserClient.class.getCanonicalName() + " url userId password");
            return;
        }

        String serverUrl = args[0];
        String userId = args[1];
        String password = args[2];

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( RestUsers.PATH );

        Response r = target.path( userId )
                .queryParam(RestUsers.PASSWORD, password).request()
                .accept(MediaType.APPLICATION_JSON)
                .get();

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() ) {
            System.out.println("Success:");
            User usr = r.readEntity(User.class);
            System.out.println( "User : " + usr);
        } else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

    }

}
```

The client to execute the get operation is very similar, except that:

(1) we have an additional path component with the message identifier (which is passed in the URL as a path parameter)

IMPLEMENTING THE CLIENT: GETUSERCLIENT

```
public class GetUserClient {

    public static void main(String[] args) throws IOException {

        if( args.length != 3) {
            System.err.println( "Use: java " + CreateUserClient.class.getCanonicalName() + " url userId password");
            return;
        }

        String serverUrl = args[0];
        String userId = args[1];
        String password = args[2];

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( RestUsers.PA

        Response r = target.path( userId )
                .queryParam(RestUsers.PASSWORD, password).request()
                .accept(MediaType.APPLICATION_JSON)
                .get();

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() ) {
            System.out.println("Success:");
            User usr = r.readEntity(User.class);
            System.out.println( "User : " + usr);
        } else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

    }

}
```

The client to execute the get operation is very similar, except that:

(1) we have an additional path component with the message identifier (which is passed in the URL as a path parameter)

(2) We have a query parameter named password

IMPLEMENTING THE CLIENT: GETUSERCLIENT

```java
public class GetUserClient {

    public static void main(String[] args) throws IOException {

        if( args.length != 3) {
            System.err.println( "Use: java " + CreateUserClient.class.getCanonicalName() + " url userId password");
            return;
        }

        String serverUrl = args[0];
        String userId = args[1];
        String password = args[2];

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( RestUsers.PA

        Response r = target.path( userId )
                .queryParam(RestUsers.PASSWORD, password).request()
                .accept(MediaType.APPLICATION_JSON)
                .get();

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity()
            System.out.println("Success:");
            User usr = r.readEntity(User.class);
            System.out.println( "User : " + usr);
        } else
            System.out.println("Error, HTTP error status: " + r.getStatu

    }

}
```

The client to execute the get operation is very similar, except that:

(1) we have an additional path component with the message identifier (which is passed in the URL as a path parameter)

(2) We have a query parameter named password

(3) Instead of the method post, since this operation is a get, we use that method instead (with no parameters since get should not have a body parameter).

IMPLEMENTING THE CLIENT: GETUSERCLIENT

```java
public class GetUserClient {

    public static void main(String[] args) throws IOException {

        if( args.length != 3) {
            System.err.println( "Use: java " + CreateUserClient.class.getCanonicalName() + " url userId password");
            return;
        }

        String serverUrl = args[0];
        String userId = args[1];
        String password = args[2];

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( RestUsers.PATH );

        Response r = target.path( userId )
                .queryParam(RestUsers.PASSWORD, password).request()
                .accept(MediaType.APPLICATION_JSON)
                .get();

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() ) {
            System.out.println("Success:");
            User usr = r.readEntity(User.class);
            System.out.println("User : " + usr);
        } else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

    }

}
```

When processing the response received from the server, we can use the method readEntity parameterized with the class of the instance we expect to receive to generate a Java instance from the body of the http response.

Implementing the Client: SearchUserClient

```
public class SearchUserClient {

    public static void main(String[] args) throws IOException {

        if( args.length != 2) {
            System.err.println( "Use: java " + CreateUserClient.class.getCanonicalName() + " url query");
            return;
        }

        String serverUrl = args[0];
        String query = args[1];

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( RestUsers.PATH );

        Response r = target.path("/").queryParam( RestUsers.QUERY, query).request()
                .accept(MediaType.APPLICATION_JSON)
                .get();

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() ) {
            List<User> users = r.readEntity(new GenericType<List<User>>() {});
            System.out.println("Success: (" + users.size() + " users)");
            users.stream().forEach( u -> System.out.println( u));
        } else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

    }

}
```

This method returns a List of messages. To receive an object that has a generic type (such as List), we use the GenericType interface in the readEntity method.

GOALS

In the end of this lab you should be able to:

- Understand what a WebService REST is
- Know how to develop a WS REST and Server in Java (using JAX-RS)
- Know how to develop a REST Client in Java (using JAX-RX)
- **Use Docker to test your service using your clients**

TESTING WITH DOCKER

1. **Build the image (on your project folder run):**
   - mvn clean compile assembly:single docker:build
2. **If you don’t have it yet, create the docker network sdnet**
   - docker network create -d bridge sdnet
3. **Run the server in a named container (with port forwarding)**
   - docker run -h serv --name serv --network sdnet -p 8080:8080 sd2425-lab2-xxxxx-yyyyy

NOTE: The server will output its URL – you will need it in the clients.

TESTING WITH DOCKER

4. **Run another container in interactive mode (to execute clients) <u>in a second terminal window</u>**
   - docker run -it --network sdnet sd2425-lab2-xxxxx-yyyyy /bin/bash

   You should get a terminal within a second docker container.

   You can check the files in the directory with ls, and execute commands, including java commands to start the clients.

TESTING WITH DOCKER

5. **Run the client to post a message <u>in the second container</u>**
   - java -cp sd2425.jar lab2.clients.CreateUserClient http://172.18.0.2:8080/rest jleitao Joao jc.leitao@fct.unl.pt password

   - Don't forget that you should use the URL of your own server obtained when you started the first docker container.

   When the client receives the answer from the server, if it is successful, it will provide you a confirmation that the user was created.

TESTING WITH DOCKER

6. **Run the client to get a message <u>in the second container</u>**

   - java -cp sd2425.jar lab2.clients.GetUserClient http://172.18.0.2:8080/rest jleitao password

   Again remember that you should use the correct server URL

   If successful the client, after getting the data from the server, should print the obtained User instance (using the representation defined by the toString method).

Testing with Docker

7. **Use your browser to access your service (optional)**
   - Since we exposed the port 8080 of the first docker container, all TCP connections reaching your localhost on this port will be redirected to that docker container and the Java server running there.
   - You can use the URL associated with the method to search for users (with or without the optimal 'search' query parameter and obtain a JSON representation of users in your browser.

   E.g., http://localhost:8080/rest/users/?query=j

7. **Try the other clients that are provided in the second container (optional)**
   - Not all clients are completed…

EXERCISE

1. Complete all the operations missing in the server. Don't forget to:
    1. Add the Jersey annotations where you need them.
    2. Complete the implementation of the service
    3. You cannot delete something that does not exists.
2. Complete the clients that are incomplete, you can take advantage of the provided ones, to exercise all operations.
3. Test your implementation using docker.
4. Integrate the Discovery class from last week to enable all clients to obtain the server URL automatically (adjust user provided parameters accordingly).