Services project

Matteo Locatelli
Matthew Rossi
AA 2016/2017

Client side

When starting the program, the command line interface (CLI) displays to the user a list of the available actions:

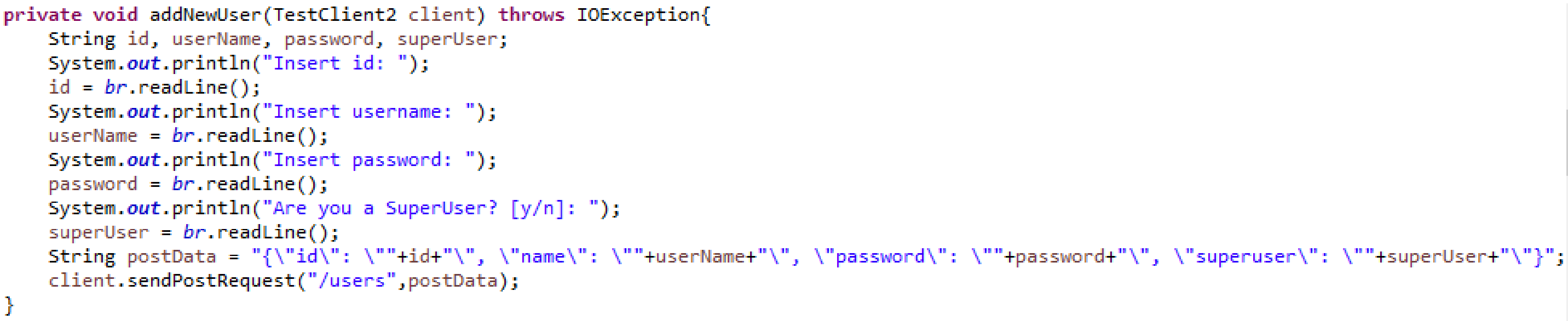
- Login
- Create user
- Exit

After logging into the system, the user can do any action required in the specification of the service, such as:

- View a user’s details
- List users
- Create a new boardgame
- View a boardgame’s details
- List boardgames
- Create a new play
- View existing plays
- Logout
- Exit

Client side

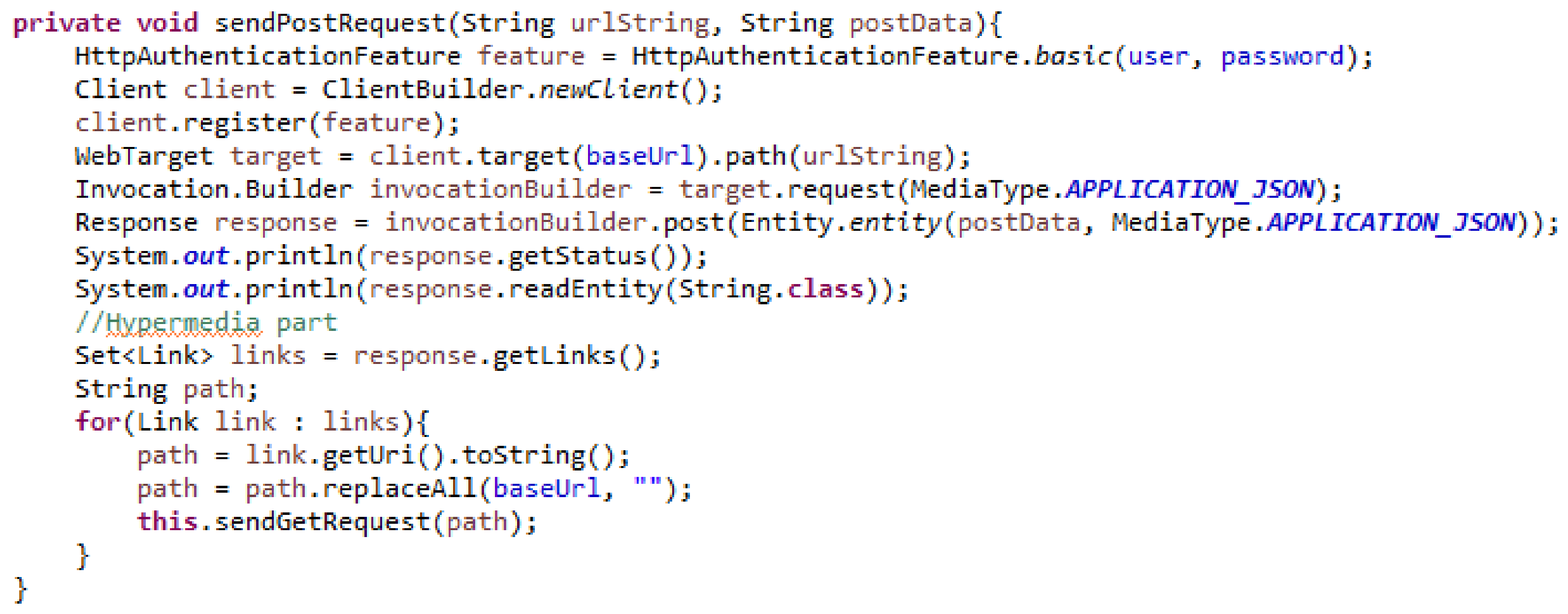
When the user selects the action he wants to do, the program sends an http request (GET or POST, depending on the type of action selected) to the server side of the application.



```java
private void addNewUser(TestClient2 client) throws IOException{
    String id, userName, password, superUser;
    System.out.println("Insert id: ");
    id = br.readLine();
    System.out.println("Insert username: ");
    userName = br.readLine();
    System.out.println("Insert password: ");
    password = br.readLine();
    System.out.println("Are you a SuperUser? [y/n]: ");
    superUser = br.readLine();
    String postData = "{\"id\": \""+id+"\", \"name\": \""+userName+"\", \"password\": \""+password+"\", \"superuser\": \""+superUser+"\"}";
    client.sendPostRequest("/users",postData);
}
```

Client side

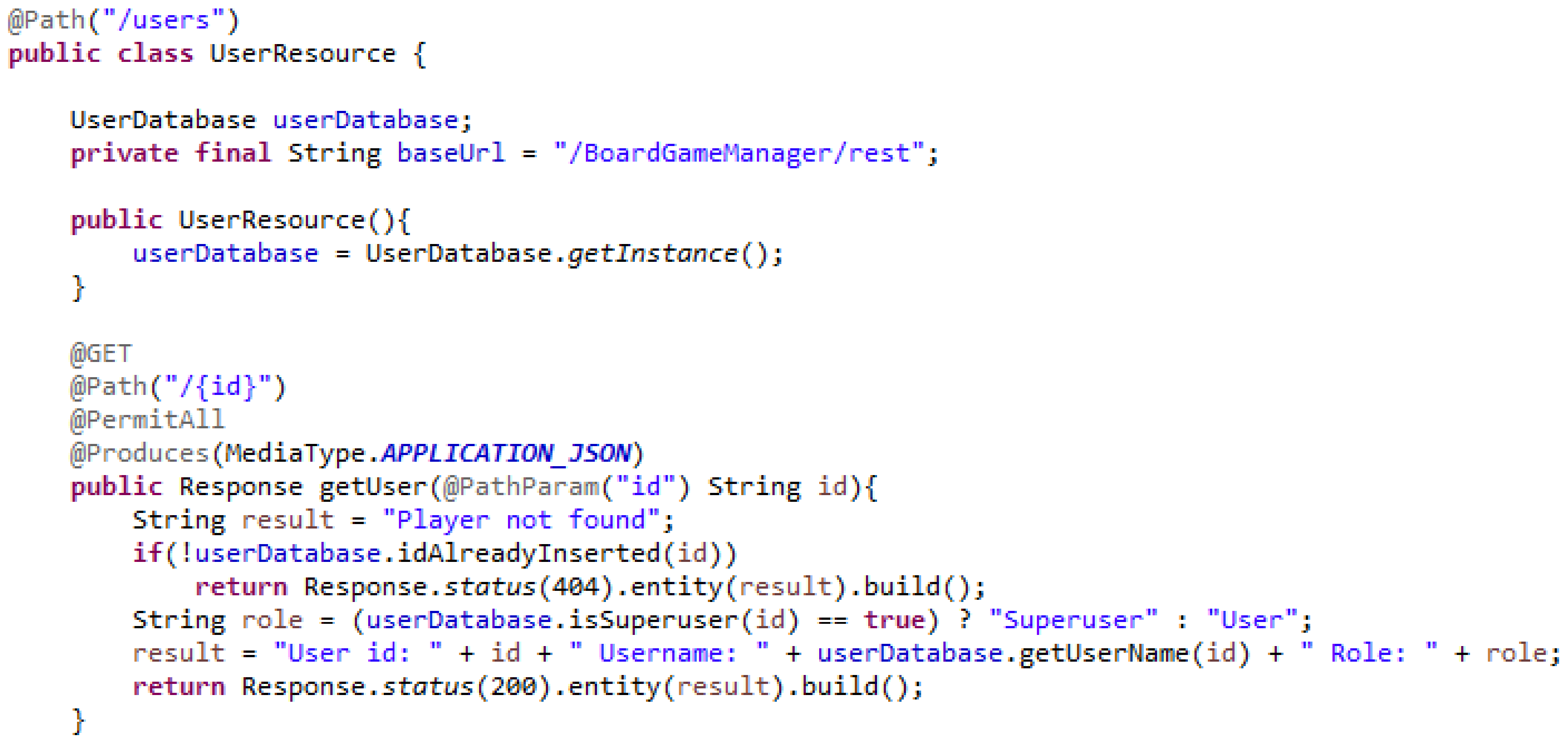
When the client sends a POST (or GET) request to the server, it waits for a response in order to read it and to display the result to the client. Also, it checks if there are any hypermedia links into the response, in order to display their content to the user.



```
private void sendPostRequest(String urlString, String postData){
    HttpAuthenticationFeature feature = HttpAuthenticationFeature.basic(user, password);
    Client client = ClientBuilder.newClient();
    client.register(feature);
    WebTarget target = client.target(baseUrl).path(urlString);
    Invocation.Builder invocationBuilder = target.request(MediaType.APPLICATION_JSON);
    Response response = invocationBuilder.post(Entity.entity(postData, MediaType.APPLICATION_JSON));
    System.out.println(response.getStatus());
    System.out.println(response.readEntity(String.class));
    //Hypermedia part
    Set<Link> links = response.getLinks();
    String path;
    for(Link link : links){
        path = link.getUri().toString();
        path = path.replaceAll(baseUrl, "");
        this.sendGetRequest(path);
    }
}
```

Sever side

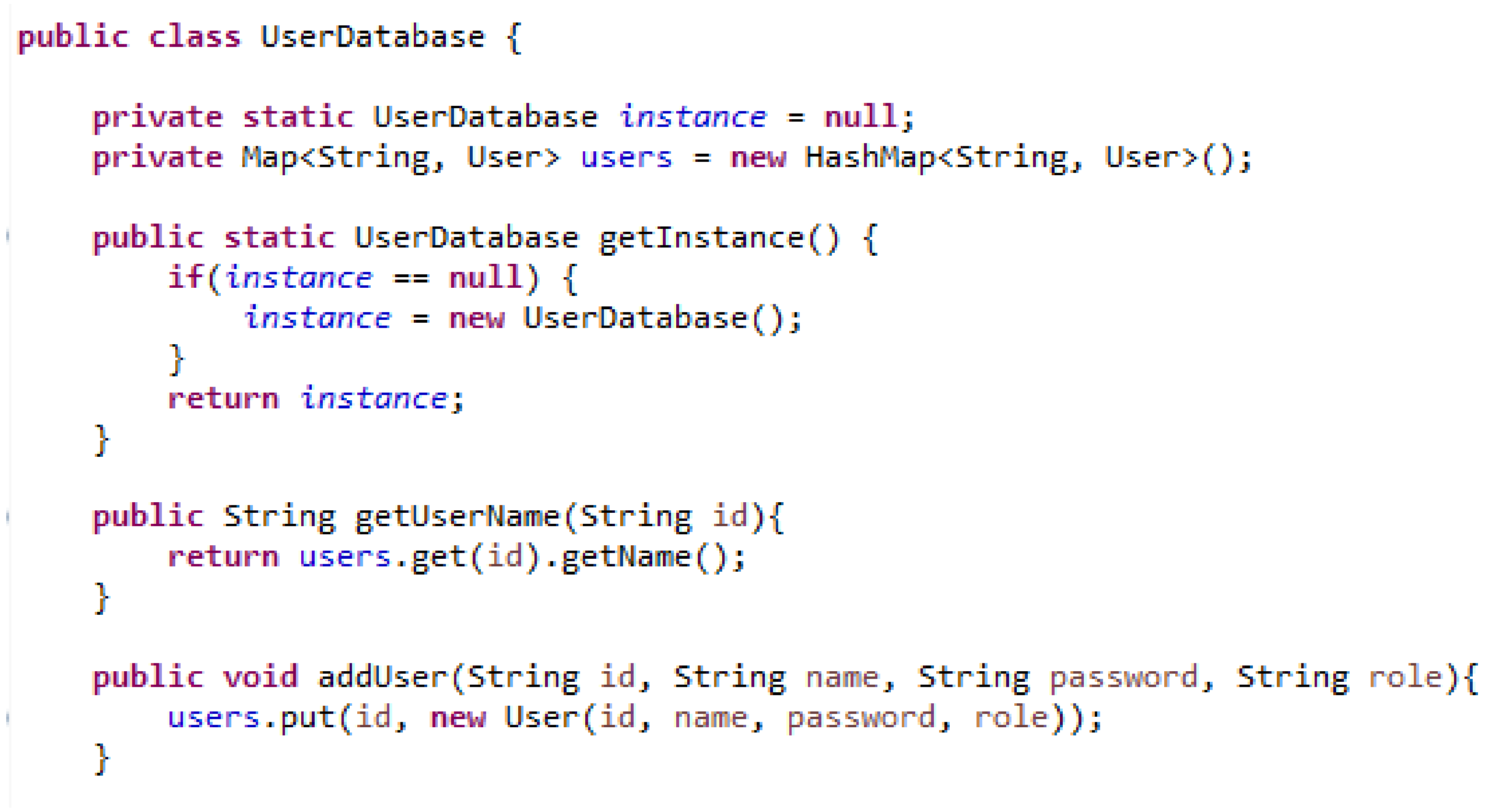
At the server side of the application, there are the resources needed (i.e. users, boardgames and plays) which have all the methods required to send the client the response when it makes a request.



```java
@Path("/users")
public class UserResource {

    UserDatabase userDatabase;
    private final String baseUrl = "/BoardGameManager/rest";

    public UserResource(){
        userDatabase = UserDatabase.getInstance();
    }

    @GET
    @Path("/{id}")
    @PermitAll
    @Produces(MediaType.APPLICATION_JSON)
    public Response getUser(@PathParam("id") String id){
        String result = "Player not found";
        if(!userDatabase.idAlreadyInserted(id))
            return Response.status(404).entity(result).build();
        String role = (userDatabase.isSuperuser(id) == true) ? "Superuser" : "User";
        result = "User id: " + id + " Username: " + userDatabase.getUserName(id) + " Role: " + role;
        return Response.status(200).entity(result).build();
    }
```

Sever side

There is also a database for each of the resources (like UserDatabase) that contains the actual data and all the methods used by the resource.

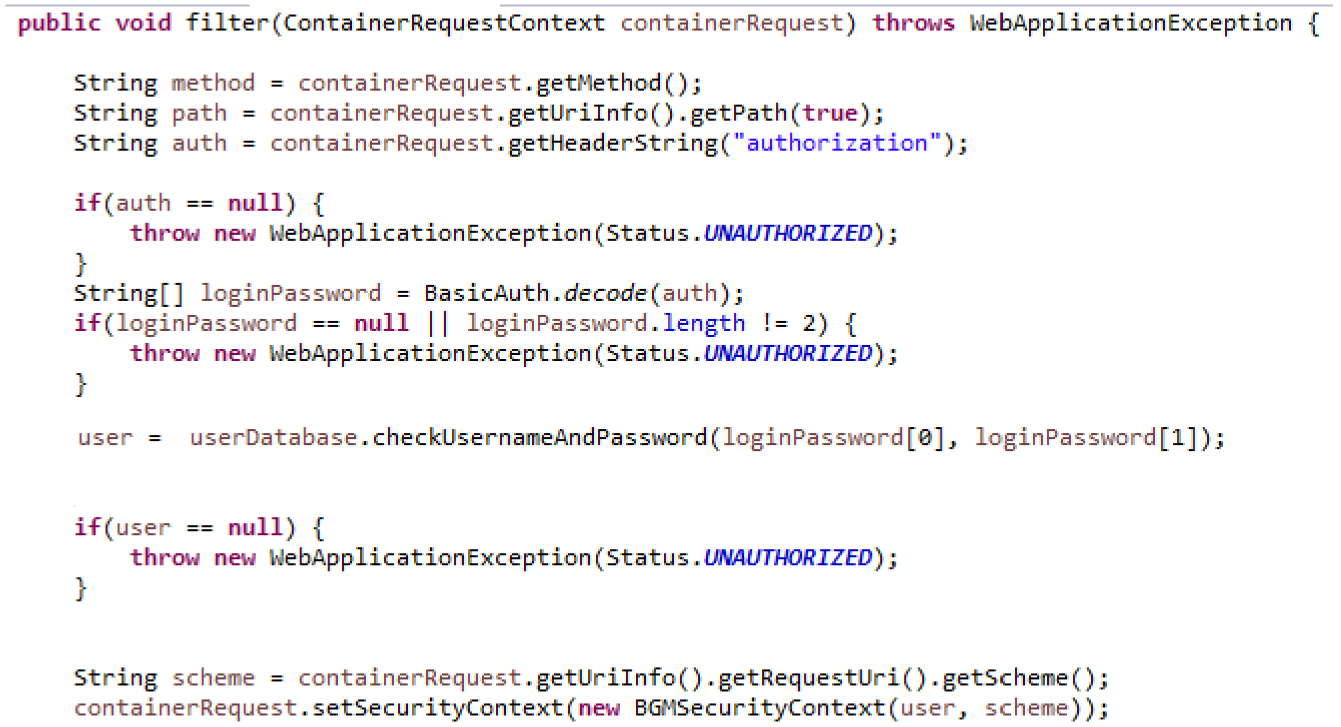


```java
public class UserDatabase {

    private static UserDatabase instance = null;
    private Map<String, User> users = new HashMap<String, User>();

    public static UserDatabase getInstance() {
        if(instance == null) {
            instance = new UserDatabase();
        }
        return instance;
    }

    public String getUserName(String id){
        return users.get(id).getName();
    }

    public void addUser(String id, String name, String password, String role){
        users.put(id, new User(id, name, password, role));
    }
```

Sever side

In order to guarantee the fact that only power users can create new resources, there is an authorization filter that is used to:

- Ensure that a user uses the GET or POST methods that he has the privileges to access. This restriction is defined in the resource code, using the annotation @PermitAll and @RolesAllowed(“Superuser”)
- Ensure that a user can log into the system only if the username and password he inserts during the login phase are correct.



```
public void filter(ContainerRequestContext containerRequest) throws WebApplicationException {

    String method = containerRequest.getMethod();
    String path = containerRequest.getUriInfo().getPath(true);
    String auth = containerRequest.getHeaderString("authorization");

    if(auth == null) {
        throw new WebApplicationException(Status.UNAUTHORIZED);
    }
    String[] loginPassword = BasicAuth.decode(auth);
    if(loginPassword == null || loginPassword.length != 2) {
        throw new WebApplicationException(Status.UNAUTHORIZED);
    }

    user =  userDatabase.checkUsernameAndPassword(loginPassword[0], loginPassword[1]);


    if(user == null) {
        throw new WebApplicationException(Status.UNAUTHORIZED);
    }


    String scheme = containerRequest.getUriInfo().getRequestUri().getScheme();
    containerRequest.setSecurityContext(new BGMSecurityContext(user, scheme));
```