

UserResource - God Class

[Link to chat 1](#)

[Link to chat 2](#)

[Manual Refactoring](#)

File: books-android/app/src/main/java/com/sismics/books/resource/UserResource.java

Missing modularization, hierarchy, and god class

Chat 1: Before being told the smells

Sure, let's start by refactoring the `UserResource` class to address the identified design smells:

```
package com.sismics.books.rest.resource;

import javax.ws.rs.*;
import javax.ws.rs.core.*;
import java.util.*;

@Path("/user")
public class UserResource extends BaseResource {
    private final UserService userService;

    public UserResource(UserService userService) {
        this.userService = userService;
    }

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    public Response register(UserRegistrationRequest request) {
        userService.register(request);
        return Response.ok().entity(buildSuccessResponse()).build();
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response updateProfile(UserProfileUpdateRequest request) {
        userService.updateProfile(request, principal.getName());
        return Response.ok().entity(buildSuccessResponse()).build();
    }

    @POST
    @Path("/{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response adminUpdateProfile(@PathParam("username") String
username, UserProfileUpdateRequest request) {
        checkAdminPermissions();
        userService.adminUpdateProfile(username, request);
    }
}
```

```

        return Response.ok().entity(buildSuccessResponse()).build();
    }

    @GET
    @Path("/check_username")
    @Produces(MediaType.APPLICATION_JSON)
    public Response checkUsernameAvailability(@QueryParam("username")
String username) {
        boolean available = userService.isUsernameAvailable(username);
        return
Response.ok().entity(buildCheckUsernameResponse(available)).build();
    }

    @POST
    @Path("/login")
    @Produces(MediaType.APPLICATION_JSON)
    public Response login(UserLoginRequest request) {
        String authToken = userService.login(request);
        return
Response.ok().entity(buildLoginResponse(authToken)).cookie(buildCookie(auth
Token)).build();
    }

    @POST
    @Path("/logout")
    @Produces(MediaType.APPLICATION_JSON)
    public Response logout() {
        String authToken = extractAuthTokenFromRequest();
        userService.logout(authToken);
        return
Response.ok().entity(buildSuccessResponse()).cookie(buildEmptyCookie()).bui
ld();
    }

    @DELETE
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteOwnAccount() {
        userService.deleteAccount(principal.getName());
        return Response.ok().entity(buildSuccessResponse()).build();
    }

    @DELETE
    @Path("/{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response adminDeleteAccount(@PathParam("username") String
username) {
        checkAdminPermissions();
        userService.adminDeleteAccount(username);
        return Response.ok().entity(buildSuccessResponse()).build();
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getOwnProfileInfo() {

```

```

        UserProfileInfoResponse response =
userService.getUserProfileInfo(principal.getName());
        return Response.ok().entity(response).build();
    }

    @GET
    @Path("/{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getProfileInfo(@PathParam("username") String username)
    {
        checkAdminPermissions();
        UserProfileInfoResponse response =
userService.getUserProfileInfo(username);
        return Response.ok().entity(response).build();
    }

    // Other methods for listing users, managing sessions, etc.

    // Helper methods for response construction
    private Map<String, Object> buildSuccessResponse() {
        Map<String, Object> response = new HashMap<>();
        response.put("status", "ok");
        return response;
    }

    private Map<String, Object> buildCheckUsernameResponse(boolean
available) {
        Map<String, Object> response = new HashMap<>();
        response.put("status", available ? "ok" : "ko");
        response.put("message", available ? "Username available" :
"Username already registered");
        return response;
    }

    private Map<String, Object> buildLoginResponse(String authToken) {
        Map<String, Object> response = new HashMap<>();
        response.put("status", "ok");
        response.put("auth_token", authToken);
        return response;
    }

    private NewCookie buildCookie(String authToken) {
        int maxAge = TokenBasedSecurityFilter.TOKEN_LONG_LIFETIME;
        return new NewCookie(TokenBasedSecurityFilter.COOKIE_NAME,
authToken, "/", null, null, maxAge, false);
    }

    private NewCookie buildEmptyCookie() {
        return new NewCookie(TokenBasedSecurityFilter.COOKIE_NAME, null);
    }

    // Other helper methods
    private void checkAdminPermissions() {
        if (!hasAdminPermissions()) {

```

```

        throw new ForbiddenClientException();
    }
}

private boolean hasAdminPermissions() {
    // Implement logic to check if the user has admin permissions
    // For example, by checking the user's roles or permissions
    return false; // Placeholder implementation
}

private String extractAuthTokenFromRequest() {
    // Implement logic to extract the auth token from the request
    // For example, by accessing cookies or headers
    return null; // Placeholder implementation
}
}

```

In this refactored version:

- The **UserResource** class has been simplified and divided into methods that handle specific functionalities such as registration, profile updates, login/logout, etc.
- Each resource method now takes a single request object containing the necessary parameters, reducing the complexity of method signatures and improving readability.
- Business logic has been moved to a separate **UserService** class, which encapsulates user management operations and promotes better separation of concerns.
- Helper methods have been introduced to construct response entities and manage cookies, reducing code duplication and improving maintainability.
- Placeholder implementations have been provided for methods related to authentication and authorization, which can be replaced with actual logic to check user permissions and extract authentication tokens from requests.
- The **BaseResource** superclass is assumed to provide common functionality such as authentication and access to principal information.

GPT is able to correctly identify the design smells in the code and is able to fix them by creating separate classes to create modularization and hierarchy. It however fails to divide the classes further based on additional responsibility like CRUD.

Chat 2: After being told that the design smells:

To address the missing modularization, hierarchy, and god class design smells in the provided code, we can refactor it by introducing a more modular structure and separating concerns into smaller, more specialized classes. Here's a refactored version of the **UserResource** class:

```

package com.sismics.books.rest.resource;

import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

```

```

import org.codehaus.jettison.json.JSONException;
import org.codehaus.jettison.json.JSONObject;

@Path("/user")
public class UserResource extends BaseResource {
    private final UserRegistrationService userRegistrationService;
    private final UserUpdateService userUpdateService;
    private final UserAuthenticationService userAuthenticationService;
    private final UserDeletionService userDeletionService;
    private final UserInfoService userInfoService;
    private final UserSessionService userSessionService;

    public UserResource() {
        this.userRegistrationService = new UserRegistrationService();
        this.userUpdateService = new UserUpdateService();
        this.userAuthenticationService = new UserAuthenticationService();
        this.userDeletionService = new UserDeletionService();
        this.userInfoService = new UserInfoService();
        this.userSessionService = new UserSessionService();
    }

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    public Response register(
        @FormParam("username") String username,
        @FormParam("password") String password,
        @FormParam("locale") String localeId,
        @FormParam("email") String email) throws JSONException {
        return userRegistrationService.register(username, password,
        localeId, email);
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response update(
        @FormParam("password") String password,
        @FormParam("email") String email,
        @FormParam("theme") String themeId,
        @FormParam("locale") String localeId,
        @FormParam("first_connection") Boolean firstConnection) throws
    JSONException {
        return userUpdateService.update(password, email, themeId, localeId,
        firstConnection, principal);
    }

    @POST
    @Path("{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response update(
        @PathParam("username") String username,
        @FormParam("password") String password,
        @FormParam("email") String email,
        @FormParam("theme") String themeId,
        @FormParam("locale") String localeId) throws JSONException {

```

```

        return userService.update(username, password, email, themeId,
        localeId, principal);
    }

    @GET
    @Path("check_username")
    @Produces(MediaType.APPLICATION_JSON)
    public Response checkUsername(
        @QueryParam("username") String username) throws JSONException {
        return userService.checkUsername(username);
    }

    @POST
    @Path("login")
    @Produces(MediaType.APPLICATION_JSON)
    public Response login(
        @FormParam("username") String username,
        @FormParam("password") String password,
        @FormParam("remember") boolean longLasted) throws JSONException
    {
        return userService.login(username, password,
        longLasted);
    }

    @POST
    @Path("logout")
    @Produces(MediaType.APPLICATION_JSON)
    public Response logout() throws JSONException {
        return userService.logout(request);
    }

    @DELETE
    @Produces(MediaType.APPLICATION_JSON)
    public Response delete() throws JSONException {
        return userService.delete(principal);
    }

    @DELETE
    @Path("{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response delete(@PathParam("username") String username) throws
    JSONException {
        return userService.delete(username, principal);
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response info() throws JSONException {
        return userService.getInfo(authenticate(), principal);
    }

    @GET
    @Path("{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)

```

```

    public Response view(@PathParam("username") String username) throws
JSONException {
        return userInfoService.view(username, authenticate(), principal);
    }

    @GET
    @Path("list")
    @Produces(MediaType.APPLICATION_JSON)
    public Response list(
        @QueryParam("limit") Integer limit,
        @QueryParam("offset") Integer offset,
        @QueryParam("sort_column") Integer sortColumn,
        @QueryParam("asc") Boolean asc) throws JSONException {
        return userInfoService.list(limit, offset, sortColumn, asc,
authenticate(), principal);
    }

    @GET
    @Path("session")
    @Produces(MediaType.APPLICATION_JSON)
    public Response session() throws JSONException {
        return userSessionService.getSessionInfo(authenticate(), request,
principal);
    }

    @DELETE
    @Path("session")
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteSession() throws JSONException {
        return userSessionService.deleteSession(authenticate(), request,
principal);
    }
}

```

In this refactoring:

- Each endpoint's functionality is delegated to a separate service class.
- Each service class handles a specific aspect of user management, such as registration, authentication, deletion, etc.
- The `BaseResource` class (not provided) presumably contains common functionality shared by all resource classes.

This refactoring improves modularity, promotes separation of concerns, and avoids the god class anti-pattern by breaking down the monolithic `UserResource` class into smaller, specialized components.

After telling it the design smells, it is able to refactor the code correctly and also creates separate classes based on separation of additional responsibility and was able to produce code which is very similarly structured to our own