Lab report #3

SLH - Restaurant Grading Application

Alexis Martins

January 27, 2024

Contents

1 Authentication

1.1 Hash passwords

By default, the passwords are in clear text in the database. We should add a mechanism to hash the password during registration and login.

I added a cryptographic module that implements Argon2 hashing and verification.

```rust
pub fn hash_password(password: &str) -> Result<String,  argon2::
    password_hash::Error> {
    let salt = SaltString::generate(&mut OsRng);
    let argon2 = Argon2::default();
    let password_hash = argon2.hash_password(password.as_bytes(), &salt)?;
    Ok(password_hash.to_string())
}

pub fn verify_password(password: &str, hash: &str) -> bool {
    let argon2 = Argon2::default();
    let password_hash = match PasswordHash::new(hash) {
        Ok(h) => h,
        Err(_) => return false,
    };
    argon2.verify_password(password.as_bytes(), &password_hash).is_ok()
}
```

We can now use it in our functions, for instance the registration function.

```rust
match is_password_valid(&password) {
    true => {
        password = hash_password(password.as_str()).unwrap();
    },
    false => {
        println!("Mot de passe trop faible");
        return ShouldContinue::No;
    }
}
```

1.2 Login code reorganization

There were multiple problems in the default implementation of this function.

First the code wasn't safe to use, it could crash between the user's hands if he mistyped his credentials. This comes with the call to `get` that wasn't properly handled in case of a non-existing user.

When I added the code to hash the password, I was careful to make the code time-constant at execution. So I still do a "dummy" verification even if the user doesn't exist.

```rust
let mut user = User::default();
let ok : bool = match User::get(&username) {
    Some(user_bd) => {
        user = user_bd;
        verify_password(&password, &user.password)
    },
    None => {
        verify_password(&password, &default_hash());
        false
```

```
    }
};

if ok {
    loop_menu(|| user_menu(&user));
} else {
    println!("Nom d'utilisateur ou mot de passe incorrect");
}

ShouldContinue::Yes
```

2 Authorization

2.1 Access control

I created an authorization module that contains all the Casbin integration for Rust.

```
lazy_static! {
    static ref ENFORCER: Mutex<Enforcer> = Mutex::new(
        Runtime::new().unwrap().block_on(async {
            Enforcer::new("./casbin/model.conf", "./casbin/policy.csv").
               await.unwrap()
        })
    );
}

pub fn check_permission(sub: &User, obj: &str, act: Action) -> CasbinResult<
   bool> {
    let enforcer = ENFORCER.lock().unwrap();
    enforcer.enforce((sub, obj, act))
}
```

Then the Casbin configuration is pretty simple. I tried to adapt to the three cases described in the lab's instructions. My model is basic, the only particularity comes from the matcher. I decided to manage the admin rights directly from there, because he can basically do everything in the application.

Note I created in the `main.rs` file an enumeration for the actions, because it was cleaner. This is the reason why I have to access my action property with the `name` attribute. I also decided to use the object to pass the establishment name as argument to Casbin. I did this choice, because the establishment is the only thing that matters to decide if a user can do certain actions and thought it was better than creating a full review object that doesn't apply to all cases. I also decided to directly check if the user is admin in the matcher, because he can basically do everything on the application.

```
[request_definition]
r = sub, obj, act
[policy_definition]
p = sub_rule, act
[policy_effect]
e = some(where (p.eft == allow))
[matchers]
m = eval(p.sub_rule) && r.act.name == p.act || r.sub.role.name == "Admin"
```

For the rules file, I translated the instructions to Casbin rules. This way a reviewer has only access to the endpoints to read his own reviews and write a review. And for the owner the rules are a bit more sophisticated

since we had to check the establishment. I decided to create the rule to read their own reviews even if it's already check in the code, because one day this could change and a verification would be missing.

```
p, r.sub.role.name == "Reviewer", ReadOwn
p, r.sub.role.name == "Reviewer", Write
p, r.sub.role.name == "Owner", ReadOwn
p, r.sub.role.name == "Owner" && r.sub.role.owned_establishment != r.obj, Write
p, r.sub.role.name == "Owner" && r.sub.role.owned_establishment == r.obj, Read
```

Finally, in my code I just had to call the `check_permission` function to verify if the user can do the action he wants to do. This was done in each function that needed a permission check (list_own_reviews, add_review, list_establishment_reviews and delete_review).

```
if let Ok(authorized) = check_permission(&user, &establishment, Action::Write) {
    if !authorized {
        println!("Vous n'avez pas la permission d'ajouter un avis sur cet établissement");
        return ShouldContinue::Yes;
    }
} else {
    println!("Erreur côté serveur, veuillez réessayer plus tard");
    return ShouldContinue::Yes;
}
```

2.2 Delete review and administration rights

To delete a review, we must be administrator. In the initial implementation, we juste asked the user if he was admin, but now I manage it directly through Casbin. This means that the following lines were replaced with a Casbin management.

```
let is_admin = Confirm::new("Êtes-vous administrateur ?")
    .with_default(true)
    .prompt()?;

if !is_admin {
    bail!("vous n'êtes pas administrateur")
}
```

3 Error management

3.1 Delete a review that doesn't exist

When we tried to delete a review of a non-existing restaurant in the database, the system crashes. I decided to add the following verification to prevent this behavior.

```
match Review::get(&name, &establishment) {
    Some(review) => {
        println!("Avis supprimé avec succés");
        review.delete()
    },
    None => {
        println!("Aucun avis trouvé");
    }
};
```

3.2 Add a review to a restaurant we already reviewed

Trying to add a review to a restaurant we already reviewed was causing a crash at the save. I added a check to verify that we didn't overwrite any existing review.

```
let review = if let Some(_) = Review::get(&user.name, &establishment) {
    println!("Vous avez déjà un avis sur cet établissement.");
    return ShouldContinue::Yes;
} else {
    Review::new(&establishment, &user.name, &comment, grade)
};
```

Note that there is also a check to the save in case there is a problem with the database.

3.3 Register an existing user

In the initial code, there was a verification missing to check if the user we tried to register already existed or not. I added a check to verify that we didn't overwrite any existing user.

```
let user = if let Some(user_bd) = User::get(&username) {
    println!("Erreur lors de la création de l'utilisateur");
    user_bd
} else {
    User::new(&username, &password, role)
};
```

There is also a check to the save in case there is a problem with the database.

3.4 Wrong error message in login and register

In the login function, the initial implementation was explicitly telling the user if the username exists or not. I decided to remove this information and to replace it by something more general just telling that the password or the username was incorrect.

```rust
// Rest of the login function checking if the user exists

if ok {
    loop_menu(|| user_menu(&user));
} else {
    println!("Username ou mot de passe incorrect");
}
```

The register also had problems with the error management. It was telling the user if the username was already taken or not. The problem in this application comes from the fact that we only have a username to identify the user. So we can't use any mechanism like telling the user we will send him a confirmation email if the account isn't already taken or something like that. The only thing we can do in order to not reveal too much information is to tell the user that the registration failed without telling him too much information about the reason.

```rust
// Rest of the registration function
let user = if let Some(_) = User::get(&username) {
    println!("Erreur lors de la création de l'utilisateur");
    return ShouldContinue::Yes;
} else {
    User::new(&username, &password, role)
};

match user.save() {
    Ok(_) => println!("Utilisateur créé avec succés"),
    Err(_) => println!("Erreur lors de la création de l'utilisateur"),
}
```

3.5 Manage Inquire errors

When I read the documentation and examples of the Inquire library, I saw that it was possible to have errors. They were handled in the examples, but not in the initial code. I decided to copy the examples and to add the error management to the application. So a typical inquire prompt is now handled as follows.

```rust
let username = match Text::new("Entrez votre nom d'utilisateur : ")
    .prompt() {
    Ok(input) => input.trim().to_ascii_lowercase(),
    Err(_) => {
        println!("Erreur lors de la saisie du nom d'utilisateur.");
        return ShouldContinue::Yes;
    }
};
```

4 Input validation

4.1 Password validation

The complexity of the passwords isn't verified, we should add a sort of policy for valid and strong passwords. I decided to use the same as the previous lab with `zxcvbn`. This is the function that is used to verify the password complexity. It checks the length of the password and its complexity taking in account the username.

```
pub fn is_password_valid(username: &str, password : &str) -> bool {
    if password.len() < MIN_PASSWORD_LENGTH || password.len() >
       MAX_PASSWORD_LENGTH {
        return false;
    }
    let estimate = zxcvbn(&password, &[username]).unwrap();
    return estimate.score() >= ZXCVBN_THRESHOLD
}
```

In the code, this will be used as follows.

```
match is_password_valid(&username, &password) {
    true => {
        password = hash_password(password.as_str()).unwrap();
    },
    false => {
        println!("Mot de passe trop faible");
        return ShouldContinue::No;
    }
}
```

4.2 Input validation of grades

It was possible to grade the restaurants with grades out of the defined range (1..5). I added to the system an input validation function fo the grades.

```
pub fn is_grade_valid(grade : u8) -> bool {
    grade >= MIN_GRADE_REVIEW && grade <= MAX_GRADE_REVIEW
}
```

In the function to add a new review, we can then add the following lines after the grade input.

```
if !is_grade_valid(grade) {
    println!("Note invalide ! Il faut une note entre {} et {}",
       MIN_GRADE_REVIEW, MAX_GRADE_REVIEW);
    return Ok(ShouldContinue::Yes);
}
```

4.3 Input validation for textual inputs

The application doesn't apply any filtering on textual inputs. It was hard to chose what was the best input validation for the cases like the comments, restaurant names and usernames. I decided to mainly check the length of the inputs. So each input named above has its own constant for the minimum and maximum length. I check this everytime the user inputs something related to these attributes.

```rust
pub const COMMENT_MIN_LENGTH: usize = 1;
pub const COMMENT_MAX_LENGTH: usize = 1024;
pub const USERNAME_MIN_LENGTH: usize = 1;
pub const USERNAME_MAX_LENGTH: usize = 32;
pub const ESTABLISHMENT_MIN_LENGTH: usize = 1;
pub const ESTABLISHMENT_MAX_LENGTH: usize = 32;
```

I did a function for each of these attributes to check the length of the input.

```rust
pub fn is_username_length_valid(input: &str) -> bool {
    input.len() >= USERNAME_MIN_LENGTH && input.len() <= USERNAME_MAX_LENGTH
}
```

Then I use these functions in the code to check the inputs.

```rust
if !is_username_length_valid(&username) {
    println!("Erreur lors de la saisie du nom d'utilisateur, il doit faire
        entre {} et {} caractéres.", USERNAME_MIN_LENGTH,
        USERNAME_MAX_LENGTH);
    return ShouldContinue::Yes;
}
```

4.4 Names formatting

The application doesn't apply any filtering on restaurant names and usernames. I decided to add a trim, a convert to ascii and a convert to lowercase for these two attributes. This way we can't have multiple users or restaurants with the "same" name. I did this by adding to the input prompts the following methods.

```rust
.trim()
.to_ascii_lowercase();
```

5 Unit tests

I also added some unit tests to the application. Every external module has a bunch of unit tests to verify the correct behavior of the functions.

Figure 1: Average Casbin user