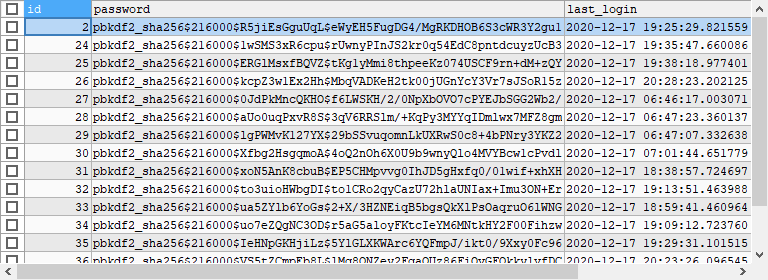
For the security protection, we did it through several ways.

**Password security:**

First of all, we store the user’s password after encryption. Instead of storing the User ID and password in the Customer table, we stored it in the user table provided by the Django Framework. And use the create\_user, login() and logout() API function from Django API to achieve the user’s account operation.

user = User.objects.create\_user(username=email, password=password1, is\_staff=True)  
user.groups.add(2)  
logout(request)  
login(request, user)

And after the user has been created, the password will be encrypted by Django using hash function.



**Transaction:**

Then, we need to deal with the transaction. Since there might be multiple queries in one transaction, we need to meet the ACID principle. For doing that, here’s an example:

cust = Customer(emailID=email, firstName=firstName, lastName=lastName, phoneNo=phoneNo, streetAddr=streetAddr,city=city, state=state, zipcode=zipcode, customerType='I')  
indiU = individualCustomer(customerID=cust, driverLicenceNo=DLNO, insuranceCompany=insuranceCompany, insurancePolicyNo=insPolicyNo)  
try: #Transaction begin  
 with transaction.atomic():  
 cust.save()  
 indiU.save()  
 messages.error(request, 'Success!')  
 user = User.objects.create\_user(username=email, password=password1, is\_staff=True)  
 user.groups.add(3)  
 logout(request)  
 login(request, user)  
 return redirect("/admin")  
except DatabaseError: #Rollback  
 messages.error(request, 'Failed! Please check your information!')  
 return redirect("/register")

In this transaction, a user has been created. As I mentioned before, The password is separated from the Customer table. And the individualCustomer will also be created. So there are three objects being created and save into the database. By doing the transaction.atomic(), all the objects will be saved into the database if there’s no problem, but when there is an error, all three objects will roll back.

And to prevent the customer to register with some false information, I added some check in the registration process. Like checking if the email is valid, password and re-entered password are matched, no blank input etc.

**Permission**  
In order to prevent the user from obtaining permissions that do not belong to him, we created three group: Individual Customer, Corporation Customer and Employee. Each class have different permissions on different resources. For example, in the rental service part, Customer should only be able to add or view a service, but not change them after the rental service has been made.

And the employee should have all the permissions except view or change customer’s password.

**Sql Injection**

Then for preventing the sql injection, we used the Objective Relational Mapper, which is a in-build API provided by Django. Django then converts the Python query to SQL query and communicates with the database.

A simple select query is like this:

cust = Customer.objects.filter(emailID=request.user.username).first()

**CSRF attack**

For csrf attack, we use csrf token in the html. We also used form to send POST request to the back end.

<form action="/registerIndi/" class="form-inline" style="margin-left : 430px" method="post">**{% csrf\_token %}**

Since this is the first time that we create a website totally by ourselves, this project really taught us a lot. The project helps us have a better understanding on the Database and more familiar to the operations like creating triggers, procedures etc. We also learned a new programming language Python. By using the framework Django to connect the database and the front end, we got a full view on the website development, learned more about how the website we browse every day works. And during the debugging, we learned not only the solution of the problem, but also the way to solve the problem.