Summer Practice Report
Department of Computer Engineering
METU

Deniz Polat

Summer 2019

Contents

1 Introduction

For 30 workdays, from 15^{th} of July till 2^{nd} of September, I have done my summer practice at Nart Informatics, which is a web development company located in Ankara. During my summer practice, I have designed a web application named *Ne Yesek?* which allows users to create an account and interact to each other, upload different recipes and filter the meals in a user-friendly way by the ingredients of meals. This application also allows users to see the meals they can cook by using the ingredients they currently have.

2 Nart Informatics - TechNarts

2.1 Company

The company is established in 2007 and located in Middle East Technical University MET Campus, Ankara since 2009. Their mission is "providing IT products and services by analyzing the requirements of their clients in IT sector, to ensure and enhance resource and business productivity". The company is developing several products for professional business purposes. Some of their biggest projects are *Transmission Alarm Management* for Turkcell Superonline, *İlaç Takip Sistemi* and *Pharmaceutical Track & Trace System* for Republic of Turkey, Ministry of Health and *Sport News Portal* for Digiturk - LigTV. By this internship, I have learned a lot about web development tools and I had chance to observe the ongoing projects and by using all the information I gathered, I could be able to develop a small project.

2.2 People

There are 13 people working for the company and the founder of the company is Taha Yaycı, who has a Bachelor's degree in Mathematics between 1995-1999 in Middle East Technical University. Halim Görkem Gülmez is a senior backend developer and he works as the project manager of the team. After him, Yılmaz Baysal and Ömercan Gökkaya works as junior backend developers. There were also Anar Musayev as a senior frontend developer and Abdullah Akalın as junior frontend developer. İsmail Taha Aykaç, as a senior backend developer, and Shkelqim Memolla, senior frontend developer were working on some other projects.

2.2.1 My Supervisor

Halim Görkem Gülmez was my supervisor during my internship in the department of software development. He has a Bachelor's degree from Computer Engineering between 2010-2015 from Middle East Technical University and a Master's degree between 2015-2019 in Middle East Technical University, also in the department of Computer Engineering.

Contact Information:
Address: ODTU TEKNOKENT MET YERLESKESI Mustafa Kemal Mah. Dumlupinar Bulvari No:280 E Blok 2/A 06530 Cankaya, Ankara
Phone Number: 0 544 239 36 07
e-mail: gorkem.gulmez@technarts.com

2.3 Hardware and Software Systems

In general, people in TechNarts use the programming language Python in back-end and Django is mainly used as the web framefork. For frontend, developers mainly code in html, CSS and JavaScript with its tools such as jQuery. MySQL was used as the database management system for the projects and employees use Slack for communication. During the internship of all 17 interns in total, our supervisors made mini-sessions for teaching some new tools, technologies and algorithms such as Docker, Virtual Environment, TCP, UDP, path finding algorithms(such as A*), machine learning algorithms(like k-cluster and k-nn) and so on.

2.4 Working Environment

There were 2 offices for TechNarts: Main office and TechNarts Garage. These two offices were at the same floor of the building, so they were very close to each other. My desk was in Garage. All of the people were so friendly to interns, and each other, for sure. There were a small kitchen, a desk for Hasan Abi who helps with document works and setting up appointments for the company, employer's (Taha Yaycı) office, a meeting room and a big room for both employees and interns. In this big room, there were desks, one for per person, and a *resting room* which has big cushions and a massage chair in it. We all were allowed to use all these materials whenever we want. In the Garage office, there were also tables for both employees and interns, a table tennis, a dartboard, a pinball, a playstation and a whiteboard. Whenever we want, especially when we get bored or stuck with an error, we played something to relax. We go to lunch all together and employees were paying attention to us. We were asking them for help whenever we want and we were spending time together in lunch time. Also, we were able to use kitchen, eat or drink something anytime.

3 Information About My Project - Ne Yesek?

As I mentioned in the introduction part of this report, I have designed a new web application and worked on it. In this web application, users are able to create accounts and edit their profile, upload new recipes to the system, reach their recipes from their profile page, follow/unfollow other users, favorite recipes, make comments to recipes and so on. There is a *Fridge* section, which allows users to save the foods that the user currently has to the system and see the recipes which can be cooked by using these ingredients. Moreover, user can edit

his/her profile page and change the settings for *allergic items*, which makes the recipes including allergic items invisible for that user.

3.1 Preliminary Work Phase

Before I have started this project, I have gathered some information about the tools that I will be using for the project. I had programmed in Python before, but I have had no idea about web projects. Therefore, in the first week of my summer practice, I have watched tutorials in order to learn Django and I used the official Django website, djangoproject, as a guideline. I have completed the example app of djangoproject, the polls app. I also gathered enough information about html and css to survive while working on basics of the frontend. Therefore, I knew what kind of projects I can work on, so I decided about my project, *Ne Yesek?* Moreover, I have installed Python3, virtual environment, MySQL and pip. Hence, when the first week of my internship was over, my working environment was ready and I had sufficient information on how I'm going to proceed.

3.2 *Ne Yesek?* From the Eyes of User

When a user opens *Ne Yesek?* for the first time, or she clicks the brand part on the left of the navigation bar when (s)he is not authenticated, (s)he finds the index page where (s)he finds the buttons to login or sign up which looks like:

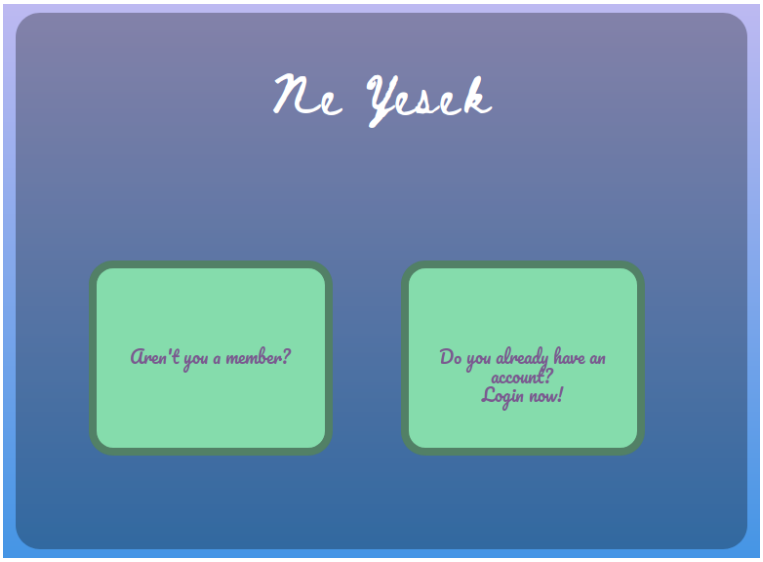
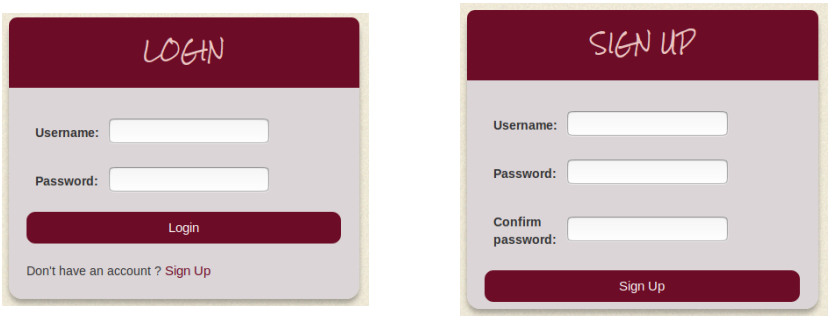


Figure 1: Index page of *Ne Yesek?*

and when the user clicks on any of them, login or sign up pages comes, respectively. These boxes look like:



(a) Login Box

(b) Sign Up Box

Figure 2: Authentication Boxes

Once the user is authenticated, button with the brand logo on navigation bar does not open the index page anymore, but opens the profile of the user instead:

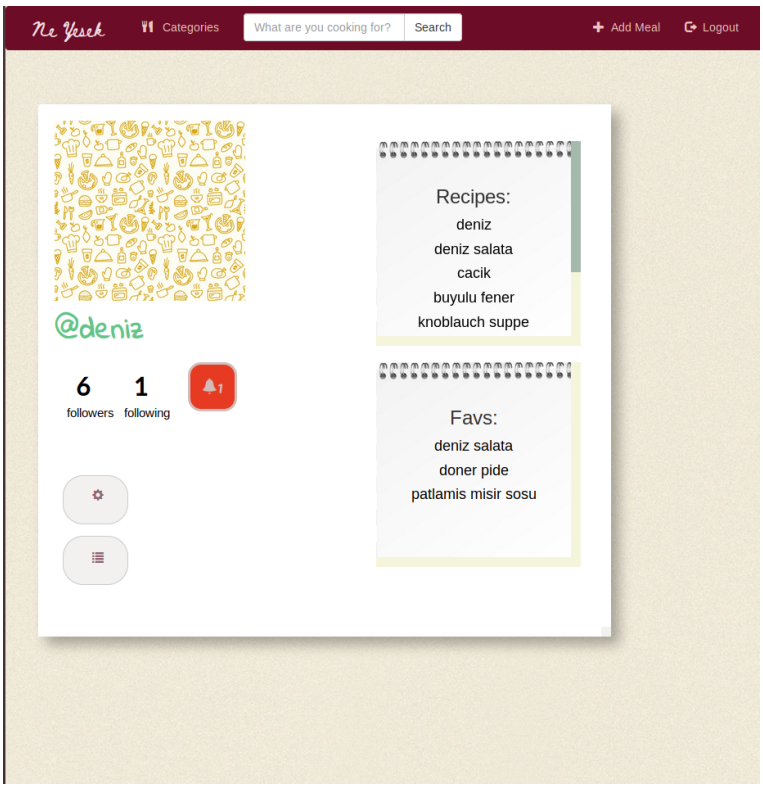


Figure 3: Profile page of the user

The user can easily reach the meals that (s)he has created and the ones (s)he added to his/her favorites from the right-hand side of the profile card. Under profile picture and username, one can reach his/her followers and people following him/her easily, and next to these buttons, there is a *notifications* part where one sees the *waiting comments* which I have mentioned before. Below them, there is profile settings and fridge buttons. The former one is for updating profile picture and/or allergic ingredients, where the latter one shows the user's *"items in his/her fridge"*. From fridge, one can see the meals which could be cooked by using only the items in fridge. The fridge page looks like:

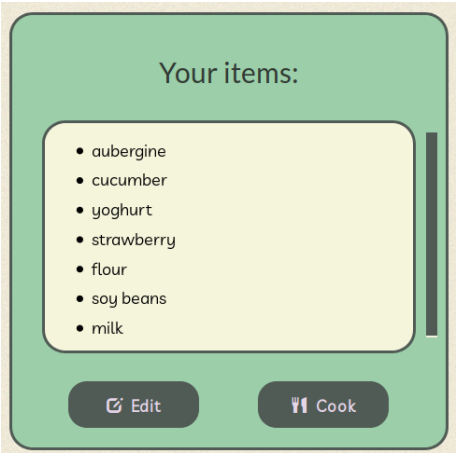


Figure 4: *My Fridge* card of the user

For anyone, whether authenticated or not, the meal cards under any category looks like:

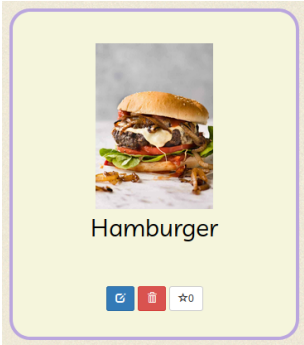


Figure 5: *Meal* card

There are edit, delete and favorite buttons on the card where all of them require login and the first 2 of them does not allow you to do the operation until

you are the one that created the meal.

On meal detail page, I used accordion and resizer tools from jQuery UI. There are 4 sections, named Details, Ingredients, Recipe and Comments(number_of_comments). If the meal is added by the current user, then (s)he sees one more section, which is named Waiting Comments. This page is as in Figure 6:

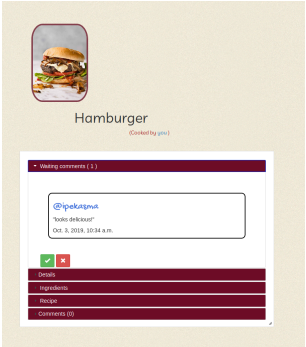


Figure 6: Detail page of a meal

If the user approves comment, then it will be visible on Comments section. Under the meal name, one can see who added the meal, go the cooker's profile, see his/her recipes and follow him/her.

4 Implementation Phase

My supervisor has guided me during this project and I asked questions to anyone whenever I got stuck. We were exchanging ideas with other interns and all the employees were eagered to share their creative ideas about our projects. For the backend side of my project, i used virtual environment, as I mentioned before. I get virtualenv by using pip from terminal. Activating and deactivating were so easy, and could be done with a few lines of command in terminal:

```
# create virtual environment:
$ virtualenv -p /usr/bin/python3 <name_of_venv>

# activate virtual environment:
$ <source_of_venv> <name_of_venv>/bin/activate

# deactivate virtual environment:
$ deactivate
```

In order to start the project, run my server on local host and start my app; I used the following commands:

```
# start project:
$ django-admin startproject <name_of_project>

# in order to run the server, go to the source directory which
    includes manage.py and run the following command:
$ python manage.py runserver

# start the app:
$ python manage.py startapp <name_of_app>
```

4.1 Backend

I started the backend part of my project by designing the connections of my classes first. First of all, I needed to create a class to represent the meal objects. Then, since I wanted to categorize these meals, I need an object to stand for category. In addition, every meal would have ingredients in it for sure, so I needed to represent the ingredient objects. Last of all, according to my "before starting programming" plans, I needed to represent the user objects. Later on, I decided to create an object for comments, I'm going to explain all of them in the "models" part in detail.

4.1.1 Models

Models part is where you create the structure of your objects and create the schemas for your database. In Django, it is appropriate to design your models in *models.py* file. When you make changes on your models that is going to affect the schema of a table in your database, you need to *make migrations*, i.e. you need to synchronize your code and your database. By doing that, your database is kind of being updated. You can do it easily by running these 2 commands in terminal:

```
$ python manage.py makemigrations <your_app_label>
$ python manage.py migrate
```

The former one basically generates the SQL commands for preinstalled apps (which can be viewed in installed apps in settings.py) and your newly created apps' model which you add in installed apps, where the latter one executes those SQL commands in database file. Therefore, after executing migrate all the tables of your installed apps are created in your database file.

I created 5 different objects, or models, for this project and they are as follows:

Category I designed a Category object, which identifies the category of a meal. A new category can be added only by *admin* user, who is basically me, by using Django's admin page. This object has only two attributes: category name and category image.

Ingredients I have the Ingredients object, which stands for the ingredients of a meal. This object will be used in order to allow the user to disable the meals including a specific ingredient or on the contrary, listing meals including only specific ingredients later on. This object has only one atttribute, ingredient name. As the *Category* model, this model also can only be edited/added by *admin* user.

Meal Since this project is about meals, the main object was the "Meal" object, so this is the most *detailed* object of my models. Since every meal is made of some ingredients, I added in "ing" attribute to Meal object and I connected Ingredients object(s) to Meal object with this instance by using ManyToManyField as:

```
ing = models.ManyToManyField(Ingredients)
```

The reason why I used ManyToManyField was that the same ingredient could be in varius meals and a simple meal could include several ingredients, as it could be easily understood by the field's name. Then, I made up a "category" attribute to show the category that the meal object belongs to, and I connected Meal object with Category object by using ForeignKey as:

```
category = models.ForeignKey(Category, on_delete=models.CASCADE)
```

By expressing the category attribute like that, I made the connection between Category and Meal objects as follows: every meal definitely belongs to a specific category, i.e. there cannot be a Meal object such that it belongs to two different categories, and if the Category object of a specific Meal object is deleted, all the Meal objects under this deleted Category objects should also be deleted.

After that, in order to allow the users to reach the meals that they have added, I designed a "user" attribute as follows:

```
user = models.ForeignKey(User, on_delete=models.CASCADE)
```

(Here, the "User" model belongs to django.contrib.auth.models package, which means I haven't created this model. Yet, I designed a profile object and connected it with this User object which I'm going to explain later.)

Later on, I added price, meal image, meal name, recipe, cooking time and favorite count attributes to make it more detailed.

User Profile As I mentioned before, I created a User Profile object and connected it to Django's User model with this line:

```
user = models.ForeignKey(User, on_delete=models.CASCADE)
```

When someone creates a new account, a new django User object is created; yet, a User Profile object is not. In order to create a User Profile object automatically, at the time when the User object is created, I have written a simple function as follows:

```
def create_profile(sender, **kwargs):
    if kwargs['created']:
        user_profile = UserProfile.objects.create(user=kwargs['
    instance'])
    for user in User.objects.all():
        UserProfile.objects.get_or_create(user=user)

post_save.connect(create_profile, sender=User)
```

In the User Profile object, I added an instance named "allergic", which keeps the allergic ingredients in order to disable the meals including these allergic ingredients to that specific user profile, and so user, respectively. I connected this object with ingredients by this line:

```
allergic = models.ManyToManyField(Ingredients, blank=True,
    related_name='allergic')
```

Here, "blank=True" part allows a user to not to have an allergic item, which is so normal, by allowing the table in the database to have a blank row for the "allergic" column.
With a similar approach, I kept the information "what does the user currently have in his/her hand" with the instance *fridge* by this line of code:

```
fridge = models.ManyToManyField(Ingredients, blank=True,
    related_name='fridge')
```

which works pretty similar with the instance *allergic*. For both of these two lines, the connection is between UserProfile and Ingredients, so in order to , I needed to add the "related_name" part to let Django distinguish these two attributes.
I have 3 more attributes working with the same principle, connected with ManyToManyField, to keep the followers of a UserProfile object, followings of the object and favorited meals of the object. The former two of these attributes are connected to User object and the latter is connected to Meal object, respectively. I keep the count of both followers and followings by these functions:

```
def follower_count(self):
    return self.followers.count()

def following_count(self):
    return self.following.count()
```

in order to show these numbers in user's profile page.
Last of all, I have the attribute *profile_picture* to let the user have a photo on his/her profile page. Since Django's User model already has username and password instances and I connected each UserProfile object with related User object, I haven't made fields for these two.

Comment This is the last Model that I have in my project and I decided to add this model on the last days of my internship with the recommendation of my supervisor. He told me that instead of using ManyToManyField, I could create a new object, and so a new table, which makes the connection with two ForeignKey connections. Which means, instead of adding a new column to a table, I could create a *transition table*, and make the connections in this table. In order to learn this new approach, I decided to allow users to make comments to any meal. This model is as follows:

```python
class Comment(models.Model):
    user_profile = models.ForeignKey(UserProfile, on_delete=models.
    CASCADE)
    meal = models.ForeignKey(Meal, on_delete=models.CASCADE,
    related_name='comments')
    comment_text = models.TextField(max_length=1000)
    date = models.DateTimeField(default=timezone.now)
    is_approved = models.BooleanField(default=False)

    def approve_comment(self):
        self.is_approved = True
        self.save()

    def __str__(self):
        return self.comment_text
```

Here, *user_profile* attribute keeps who has made the comment, *meal* keeps to which meal the user has made the comment, *comment_text* keeps the text of the comment, *date* keeps when the comment has made. The last instance, *is_approved* is a bool, used as a flag, which helps me to provide that when the user ***a*** has made a comment to the user ***b***'s meal, this comment will not appear at the same time, but the comment will *fall* to the *waiting comments* section instead. In this section, if the user ***b*** approves the comment, then the comment will be visible to everyone. If ***b*** disapproves the comment, then it will be deleted. Until ***b*** decides whether or not (s)he will approve the comment, it will wait in the ***waiting comments*** section of this meal, which is visible only for the user ***b***. ***approve_comment*** is the function which changes the ***is_approved*** flag, basically. Last of all, ***__str__*** function is to display an object in the Django admin site and as the value inserted into a template when it displays an object, according to Django's official website.

4.1.2 Views

Views part is where all the action happens. When I first started my project, I was using Django's generic views, such as DeleteView, DetailView, ListView, UpdateView, CreateView and so on. After that, I started having trouble and my supervisor suggested me not to use generic views but write all get and post operations myself. It seemed so hard at first since doing fundamental things with generic views was so easy, but when the skeleton of my project was ready and I was working on tiny specialities, I realized that I need to overwrite lots of methods of generic views. Then I quit using generic views and I have written get and post methods myself. Most of my views are class-based; yet, there are a little function based views, too.
As I did with my models, I cannot write all my views here since they are much longer than my models(more than 500 lines of code), yet instead of that, I will give some specific examples and explain them.

```
@user_login_required
def follow_unf(request, id):
    user_to_follow = get_object_or_404(User, pk=id)
    others_profile = UserProfile.objects.get(user=user_to_follow)
    user_profile = UserProfile.objects.get(user=request.user)
    following = user_profile.following.all()
    if user_to_follow in following:
        user_profile.following.remove(user_to_follow)
        others_profile.followers.remove(request.user)
        user_to_follow.save()
        others_profile.save()
    else:
        user_profile.following.add(user_to_follow)
        others_profile.followers.add(request.user)
        others_profile.save()
        user_to_follow.save()
    return redirect('athome:profile', id)
```

For instance, this view is an example of one of my function-based views. It takes request and id as arguments, id from url, and if the logged in user is following that user, then it unfollows when this view is called; follows otherwise. After that, it redirects the same page with updated information. The first line is called as ***decorator***, which is written by me, and if the user who sends the request has not logged in, it renders the user to login page. If the user is already logged in, then it allows the user to do the operation follow/unfollow. It is necessary, because an anonymous user shouldn't be able to follow another user. That is, if a user has not logged in, then how can we know who is (s)he and save the following information to whom? The decorator includes a few lines of codes with a very simple logic and it is as follows:

```
def user_login_required(function):
    @wraps(function)
    def wrap(request, *args, **kwargs):
        if request.user.is_authenticated:
            return function(request, *args, **kwargs)
        else:
            return redirect('athome:login')
```

As you see, if the user making the request is authenticated, it allows that user to do the operation and if not, it redirects to login page.
Here is another example, which is written as a class-based view. This view allows a user to change some information on his/her profile such as profile picture or allergic ingredients. On 3^{rd} line, the keyword *form_class* is used to define which form we need for that view, where I'm going to explain what is a *form* later. In addition, the *method_decorator* keyword is used in order to use the dispatch method, which works like a judge and decides what is the method of the request. Since the decorator is a function in this example where the view is class-based, we direct the decorator to dispatch so it kind of *distributes* the decorator to other methods.

```
1  @method_decorator(user_login_required, name='dispatch')
2  class ProfileSettings(View):
3      form_class = SettingsForm
4      template_name = 'athome/profilesettings_form.html'
5
6      def get(self, request, pk):
7          form = self.form_class(None)
8          allergic = Ingredients.objects.all()
9          sel_all = UserProfile.objects.get(user=request.user).
       allergic.all()
10         return render(request, self.template_name, {'form': form, '
       allergic': allergic, 'sel_all': sel_all})
11
12     def post(self, request, pk):
13         form = self.form_class(request.POST, request.FILES)
14         if form.is_valid():
15             user_profile = UserProfile.objects.get(user=request.
       user)
16             if form.cleaned_data['profile_picture']:
17                 user_profile.profile_picture = form.cleaned_data['
       profile_picture']
18             user_profile.allergic.clear()
19             for allergic in form.cleaned_data['allergic']:
20                 user_profile.allergic.add(allergic)
21             user_profile.save()
22         return redirect('athome:profile', request.user.id)
```

In this view, if the method of request is *get*, then the ***get*** method is called. In this method, we supply an empty form to the user, by the keyword "None" on line 7. Then, we pass a context to use in html file by using *render*, which basically takes request, a context and a template as arguments here, and does the operations in template file(html) with the given context(a dictionary here). On the other hand, if the method is *post*, then we need to take the information from user(as on line 13), and then save them. We check if the form is valid, i.e. has no errors, then since the user is able to change only one of these settings and remain the other unchanged, we check whether the user changed his/her profile picture and/or allergic ingredients or not. Then, we save the new information to the database.

4.1.3 Forms

A form can briefly be explained as the way to get information from the user. In Django, we mostly define a form in *forms.py* file. For instance, the ProfileSettings view which I have given as an example under Views title as a class-based view, uses the following form class:

```
class SettingsForm(forms.ModelForm):
    class Meta:
        model = UserProfile
        fields = ('profile_picture', 'allergic')
```

This simple lines tell the computer that the form is going to have two fields, which are attributes of the class UserProfile. In other words, it says *"You are going to take a UserProfile instance, and for that instance, you are going to be given two informations, for* ***profile_picture*** *and* ***allergic*** *fields, and you are going to do your manipulations according to these informations."* to the ProfileSettings view.

4.1.4 Urls

In Django, when you are working on a web project, collecting your url paths in *urls.py* file is the common approach. The programmer needs to give the url path of a page, so when you *clicked* to something, or manually written a url path on web, it opens a new page, or shows a new view on website. In other words, we can say that urls made the connections of different views between each other. To create the path of a url, one can use *path()* function in current version of Django. Here, you can see an example path retrieved from my project:

```
path('accounts/profile/<int:id>/followers/', views.FollowersView.
    as_view(), name='followers'),
```

This path calls the *FollowersView* from views.py file. Here, *int:pk* part is a parameter that I passed, which helps me to find whose followers I'm looking for, by using the user's primary key.

4.2 Frontend

In frontend part of my project, I used CSS and HTML mostly, and a little bit JavaScript. I defined class names for divs and buttons in HTML and then designed the look of these classes in CSS file. I used JavaScript in order to make a dropdown box *on submit* and make pop-up *Are you sure?* or *You are not allowed* boxes for delete and update buttons.

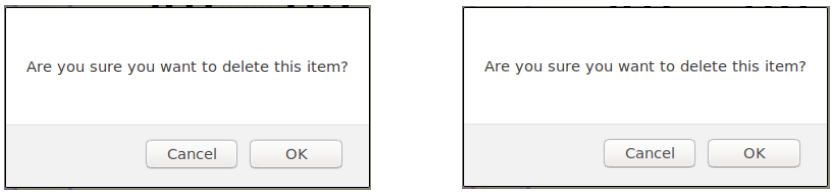
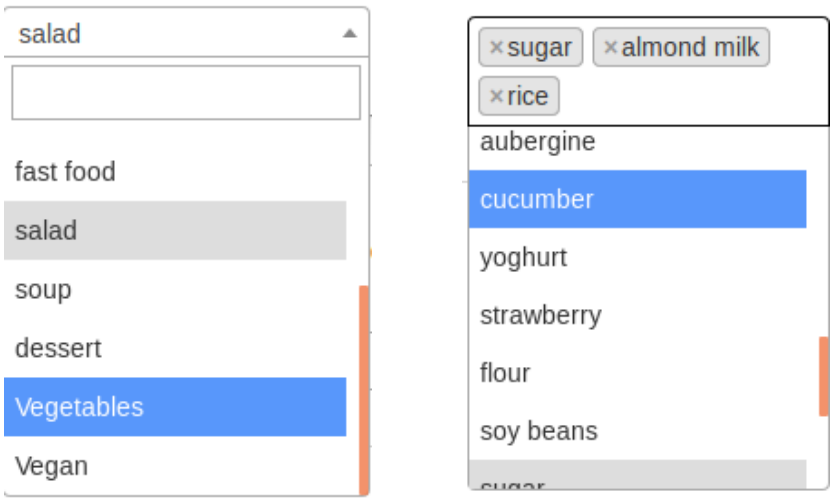


Figure 7: Pop-up warning messages

In my forms, for ManyToManyField parts, such as ingredients field when adding a new meal to the system, in order to make it more user-friendly, I used Select2, which is a jQuery based replacement for select boxes. The type of the Select2 tool that I used in this project has a search box, and a cross mark which helps user to deselect an item. I also used a different style of Select2 for the relations that I crated using ForeignKey.



(a) DropDown Select2 Box

(b) ManyToManyField Select2 Box

Figure 8: Select2 Selection Boxes

When a user clicks any of the Categories page, a dropdown box to let the user sort items comes. For this dropdown box, I used a simple JavaScript code in order to vanish submit button for this box and make it *on click*, as I mentioned before.

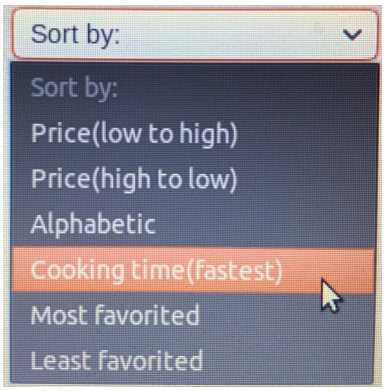


Figure 9: A dropdown box for sorting items

Last of all, I used some packages from jQuery user interface, such as accordion and resizer, which you have seen in *Section 3.2*.

5 Conclusion

In conclusion, during this internship, I have learned how to build a web application using Django and I explored what one is able to do in web sector. Since I felt so confident and happy after this internship, it was very beneficial and unforgettable for me. That was the first time that I have ever been working on a project on my own from the beginning till the end and beyond all the academic stuff I learn from school, learning how to work on a project which I designed, built and tried to apply every idea that I keep producing everyday was a very informative and self-satisfying experience for me. I have learned a lot from this summer practice, including how to be patient and how to *learn* anything besides all of the huge academic knowledge about programming that I get. All people working in the company were willing to teach something to interns and each other, that's why I thought that this process was very beneficial for me and all the other interns in the company.

6 References

About jQuery UI (n.d.) Retrieved from https://jqueryui.com/

Getting Started — Select2 (n.d.) Retrieved from https://select2.org/

Nart Informatics, TechNarts, 2007. Retrieved from https://technarts.com/

Why Django?, (n.d.) Retrieved from https://www.djangoproject.com/