Contents

ANALYSYS

Background.

The client

Client A wants to create an educational website about healthy lifestyle, nutrition and fitness and needs content for the website. One of the features he would like to implement is an application which would help people plan their diet and make it more healthy.

The problem

The client believes that the main reasons for majority of people not eating a healthy diet are:

- Ignorance about healthy nutrition principles
- Lack of time or will for self-education and planning the diet
- Inability to calculate calories properly.
- Complexity of nutrition as science

The first reason is about people who underestimate the health benefits of healthy foods and long-term effects of junk food. The second and fourth reason means that a lot of time should be invested, to become good at the subject. Things like “Micro and Macro nutrients”, “Glycemic index” and “Insulin” are only the tip of the iceberg. In addition, there are many controversies in health and fitness industry, which increases the ignorance. And the third reason is about people who try to help their health, but fail due to the lack of precision. Most common problem is about people underestimating the number of calories they take and fail to lose weight.

How the problem is going to be solved

The main goal is to eliminate as fully as possible the four main reasons by creating a program which will make things much simpler for users. It is going to improve the ignorance issue by providing examples of healthy meals, every time user generate a meal plan with the aid of the app. So users can learn which foods are good for them. The second issue will be solved completely, because all the necessary calculations and adjustments will be automated and done quickly. Users would only have to spend few minutes on inputs. The under and over estimations, random errors and other human factor issues will no longer occur, because the program not only performs precise calculations, but also gives user systematic approach, where every single gram of foods is taken is taken into account. And finally, the complexity of the subject does not matter, because all the research is done for him already and he can implement healthy principles in his diet by just pressing some buttons and following the plan.

Who the problem is being solved for

Any visitor/customer of the Client A’s website can use the application. It solves the problem for anyone, who wants to achieve certain fitness or health goals, but doesn’t know how it should be achieved. Such goals could be gaining or losing weight or cleaning the diet. The program is also suitable and very useful for very busy people, who works hard on their business goals and can’t afford spending much time on doing all the research and calculations.

Investigation.

The understanding of how the application should look like, how it should be programmed and what principles should be used comes from the following sources:

- Dialogue with the client
- Scientific peer-reviewed publications
- Personal experience of the client and author
- List of core objectives

The list of core objectives describes the abilities which should be implemented in the program. Some of them are more important than others and the goals with the high level of importance are essential success criteria. Dialogue with the client in the form of interviews or discussions are used as confirmations or clarifications of methodology of the program. For example, which formula calculating calories should be used and why. Unless some details were mentioned in the interviews, the relevance of using the proposed method can be based on the scientific studies.

Interview with the client

Interview abstract:

Two main features: calorie calculator and meal planner. Calculator has simple and advanced mode. Both versions calculate approximation of daily energy and macronutrients intake. Meal planner uses this values, number of meals, favorite food and other inputs to suggest well-balanced and healthy meals for user.

Full interview(Most important parts are underlined):

Developer: What is your vision of the program?

Client: In my vision we end up with simple to use scientifically reliable and helpful resource. The first feature is calorie calculator with two modes, simple and advanced, and the second is special meal planner which will create an example set of healthy meals based on the results of calorie calculator.

D: How is it possible to make up meal plan based on the calories?

C: The calorie calculators can calculate not only the calories intake, but also the amount of the core macronutrients intake: proteins, fats and carbohydrates.

D: And how does it work?

C: Usually the intake of macros is measured in grams per kilogram of bodyweight, so it is very different from person to person. Different diet styles have different proportions of the proteins, fats and carbs. The standard proportion is 1.5-2g of protein, 3-7g of carbs and 0,7-2g of fats. There is a variety of diet schemas suitable for improving shape and health. You can always ask me.

D: OK. Could you clarify the macronutrients part. What about "good" and "bad" proteins, fats or carbohydrates? How deep into details shall we go?

C: You're right. Proteins from various sources have different composition of amino-acids and some protein is higher in quality than other. Fats and Carbs also have some specifics which should be considered by people who cares about health. But the problem is that It is impossible to take absolutely everything into account. First, foods properties would vary, because there are many brands, farms, plantations. Second, the big problem is obesity, which is solved by restricting calories and macros, not the micronutrients. And finally, people with advanced knowledge about health and nutrition already know about things you're talking about, while the program I am looking for should help people without huge background knowledge. So the sufficient precision is less than you may think.

D: What methods would you suggest for macros approximation?

C: For proteins I would take into account only the animal sources, such as meat, eggs and diary. Most protein from plants has lack of essential amino acids, so it shouldn't be in those 1.5grams. For carbs, there are complex carbs, which are healthy, because they are slowly absorbed, while simple carbs are unhealthy in large amounts, so it is important to restrict high sugar and low in fibre food to small amount. As for fats, if they are restricted, they don't really cause health problems. In addition, I want you to make sure that meal planner recommends people eat enough vegetables. For now we may not worry about all the vitamins and minerals, but in the future I will be more than happy to see the advanced system which would adjust each vitamin in the diet up to a gram, although it is not critical.

D: How many meals a day is healthy?

C: It doesn't matter at all. If the calories intake is the same, having three big meals or six small meals doesn't have a huge impact on weight loss or gain. Let the person choose the number of meals he feels comfortable having.

D: Which formulas for calorie calculation should be used?

C: It's a big question. Total Energy Expenditure (TEE) depends on four main factors: Base Metabolic Rate(BMR), Non-Exercise Associated Thermogenesis(NEAT), Exercise Associated Thermogenesis(EAT) and Thermogenic Effect of Feeding(TEF). I would like BMR to be calculated by using Katch – McArdle formula. The problem with any formula is that it is likely to overestimate the required calories, but this is the best tool we can have. TEF is not essential, unless you really will. It is impossible to be precise about it and ignoring it in planning nutrition doesn't dramatically affect burning fat and building muscle. As for NEAT and EAT, I would like user to be able to fully describe his training activity and give approximate description of how active he is in life, but that's for advanced version, for users who will spend more time and get more precise values. For quick option multiplying by a certain coefficient will be more than enough. To lose or gain weight caloric surplus or deficit should be created and it should be 10-20% of TEE.

D: Could you tell more about detailed training activity description?

C: Of course. The average energy expenditure of different sports and other activities have been calculated. It is measured in kilocalories per kilogram-hour, i.e. the higher the weight of the person and the longer the training process, the higher the energy expenditure.

D: Excuse me, did you say KILO-calories?

C: I see, where your confusion comes from. We often use the word "calorie" when we talk about energy in food. But apparently the actual "calories" are kilocalories. On the food packaging we usually see "kcal" unit being used. One calorie is an energy required to heat up one gram of water by one degree Kelvin, it's like nothing. I have just spent tens, maybe hundreds of calories, but zero kilocalories. So make sure you use the right unit.

D: I got it. What else is important to include in the meal planner or calculator?

C: I think I didn't mention that carbohydrates have glycemic index. If it is more than 50, food should be minimised, it is impossible to avoid, because there are many high-glycemic foods around us, but make sure, that meal planner doesn't suggest eating a lot of that kind of food in one meal. Also, for healthy eating meal planner should avoid obvious junk food, like French fries or hotdogs and other fast food, because of trans fats. I would also like that meal planner suggest food which user would feel comfortable to stick too. There may be something else, but I am not sure, I think that's all for now.

D: Ok. Please, let me know if you recall something important. I have the last question. How would you call the application?

C: I would call it FreshLife228. I think that's a good name, because a lot of people may start a new life with the aid of this program.

D: Thank you very much.

C: Thank you. if you get more questions during development you can always contact me.

Specification of the core objectives

Most of the following objectives come from the interview. Some

ID	Need	Description	Importance
A	Login system	Ability to login and logout	High
B	Calories calculator	Ability to give good approximation of number of user's daily maintenance calories based on the user inputs.	High
C	Advanced options for calories calculator	Ability for user to choose between simple or advanced ways of calculating daily calories demand. Advanced option should allow user to fully describe his activity and get a more precise value	High
D	Daily Macronutrients	User should get the approximate value of amount of proteins, carbohydrates and fats his body needs daily	High

E	Favourite foods	Ability to mark certain foods as favourite, so nutrition planner would consider user preferences	High
F	Food ignore list	Ability to add food to the ignore list (Due to allergies or taste preferences). The food in that list will not be suggested in the meal plans created by this user.	High
G	Search	Ability to search foods when user wants to mark them as favourite or ignored.	Medium
H	Suggest choice	In “add food”, the application suggest user few food options he might like to add.	Medium
I	User-Friendly Interface	Application should be simple to work with and test users shouldn't have problems with it.	High
J	Nutrition Planner	Ability to create meal plans based on the values of user’s energy and macros requirements and a number of meals he would like to have per day.	High
K	Favourite food only	Ability to create meal plans containing only user’s favourite food	Medium
L	Send user to the “Add Food”	If the user doesn’t have favourite food or its number is insufficient, application should ask user to add some food.	Medium
M	Save meal plans to the database	Ability to save just created meal plan to the database	High
N	User meal plans	Ability to have all meal plans of the user displayed. User must to have ability to view the details of the particular plan or delete plan.	High
O	Different styles of diet	Make dietary plans in different styles of diet, for example ketogenic, paleo, low-carb.	Low
P	Data security	User passwords should be stored as hashed values. Allowing users create only passwords with sufficient level of complexity	High

Investigation of the problem

According to some of the core objectives it becomes obvious, that a lot of data have to be stored: The personal details of the user, properties of the food, meals and plans information, in addition all the relationships between all these things have to be represented and stored. One of the first ways to implement it which comes to mind is relational database. Such type of database model has a lot of advantages. First, Its simplicity and ease of data retrieval. Data is organized naturally with in the model, simplifying the development. Accessing data does not require navigating a rigid path through a hierarchy or tree. The data can be accessed with use of queries and user can filter the data he wants to display based on the content of its columns. Second, relational database model is flexible and easily scalable and extensible. Theoretically the number of rows and columns can be infinite. The growth of the database is only limited by the hardware of the computer and is sometimes limited by relational database managing system (DBMS). Third, with the aid of special constraints and database normalization data integrity can be preserved. Validity checks ensure that the required data is present and its values satisfy the acceptable range. If the database is well normalized, user can add, alter or delete data and be assured that no anomalies occur during creating, updating and deleting and the performance speed remains sufficiently high even when the database size increases.

The alternative approaches are XML database or Graph database. The last one immediately gets rejected, mainly because of lack of experience of working with such things. Meanwhile the XML

database model is not the best option, because it is more suitable for strictly hierarchical databases with one-to-one of one-to-many relationships, which can be imagined like a tree. Although there is a "User-Plan-Meal" hierarchy, that is not the case for the current situation, because there are many-to-many relationships between food and meals and food and people. So XML model gets rejected as well.

Additional research

This sub-section is made for the purpose of providing some extra information in popular style about specifics of the problem from the nutrition science point of view. It also shows the relevance of some methods used in the program and expands and clarifies some information from the interview.

Micro and macro nutrients.

Macronutrients are Proteins, Fats and Carbohydrates. Micronutrients are vitamins and minerals. Macros hold energy, i.e. calories, micronutrients don't have any energy, but most of them are vital for good health.

Protein is essential for building muscle and many other tissues. It consists of 22 different amino acids of which 9 are essential, 7 are conditionally essential and 4 non-essential. The quality of the protein is determined by its amino-acid composition. Protein from animal sources usually have balanced for human consumption composition, while the protein from plants sources usually has poor quality.

Fat is the most energy-dense macronutrient (9kcal/g). It is also important for absorbing some vitamins, healthy skin, hairs, nails and teeth. Saturated fat is solid and usually found in meat. Monounsaturated fat is liquid and found in oils or avocado. Polyunsaturated fatty acids, omegas-3 are rare, the richest source is salmon. Consuming a lot of any type of fat can be unhealthy, especially saturated fats. Omega-3 is healthy, but most people under-intake it. The biggest threat is trans-fat, which is found in fast food. It takes 50 days for human body to process it.

Carbohydrates are sources of energy for the body, but alike fats, the carbohydrate energy can be used for high-intensity activity. All carbohydrates are chains of glycose molecules. Carbohydrates with many glycose molecules in chain are called complex and digested slowly. Sugar is a simple carbohydrate which is absorbed quickly and that means that blood sugar grows dramatically and huge amount of extra sugar stored as fat. That's why slow carbs are healthy and sugar is a threat. Fibre is a carbohydrate with structure, which can't be digested, so technically doesn't have calories. It slows down the absorption of carbs, which is good for weight loss. There's also polyols, which are considered as carbohydrates, but they are actually sugar alchohols, which are used as sugar substitutes for diabetic people. It is not the best thing to consume, but in small amounts it is safe.

Vitamins are simple organic substances essential for living organism. Both deficit or big surplus of vitamins can cause serious problems. Fat-soluble vitamins can be stored in fat cells, water-soluble vitamins cannot be stored, so they have to be consumed every day in recommended dozes.

Minerals are metals and salts, non-organic substances. Just like vitamins, they are essential for living organisms.

Katch-McArdle formula:

Katch-McArdle formula, suggested by client A, has advantages over the other two popular formulas, such as Harris-Benedict and Mifflin-St Jeor, because their parameters are weight, height and age, while the proposed approach uses weight and body fat ratio. Using bodyfat percentage is more accurate, because the amount of energy required to support muscle tissue and fat (adipose) tissue is different. Other methods don't even consider tissue difference, so athletes with big muscle mass and person with high bodyfat can be treated the same.

Katch-McArdle Equation

$$BMR = 370 + 21.6 * \left(Weight * \frac{100 - BF\%}{100}\right)$$

Weight-kg BF% - Body Fat Percentage

Some more terminology:

Thermogenic effect of feeding – the amount of energy expenditure above the basal metabolic rate due to the cost of processing food for use and storage.

Glycemic Index – shows how quickly carbohydrates in foods are absorbed. The reference food is pure glucose which has GI = 100. The higher the glycemic index of the food consumed, the higher and quicker the blood sugar grows. When the sugar is too high, human body tries to reduce it by storing extra sugar as fat. Low-glycemic food is better for weight loss. Consuming products with GI > 50 in large amounts is not a good idea.

Choosing the tools for problem solution.

Front-end part of the application

The front-end part is going to be implemented with HTML, CSS and Java Script. Since the application is planned as a part of the website, options of creating GUI with the aid of Python libraries like Tkinter or Pygame and other not web-based options were rejected.

Back-end part of the application

Satisfaction of majority of the core objectives require a database for successful implementation. Any programming language can be used for handling databases. But instead of crossing the ocean by swimming why not choosing a fast motor boat. And the fastest boats, best tools for database handling are PHP and Python.

Pros and Cons of PHP and Python as database handling tool

PHP pros:

- Speed and simplicity
- Handling databases and other web developer needs are what the PHP is specialized for
- Big choice of frameworks and large community

PHP cons:

- Poor PHP experience. It will slow down the development dramatically

Python pros:

- Good level of Python experience and knowledge
- Wide choice of libraries
- Do more with less code
- Python Django

Python cons:

- Speed limitations
- Not as good as PHP for web developer needs

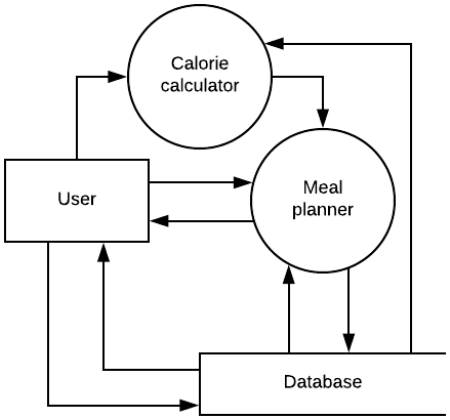
Conclusion

According to the unbiased opinion, PHP is ahead of Python in terms of web-development and database handling needs. But the fact that personal PHP experience is too low overcomes all the advantages of PHP. Python is chosen for back-end database tool, because learning and revising library is easier and faster than learning new language.

Program modelling.

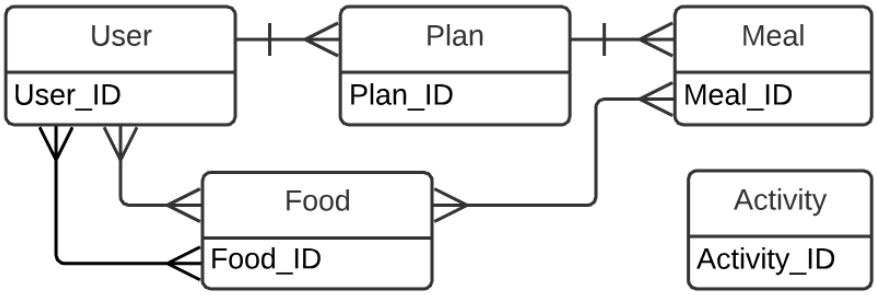
The following section is made to provide sufficient and abstract description of the internal processes of the program, such as data handling and storage, program inputs and outputs and algorithms.

Context Diagram



The image on the right shows the level 0 Data Flow Diagram. It show how data is processed within a system.

Entity relationship diagram



The entities within the database interact with respect of the following principles:

- User can have many plans, but plan is owned by only one user
- Plans can have many meals, but meal exists only within one plan
- User can have many favorite foods. Same food can be favored by many users
- User can Ignore many foods. Same food can be ignored by many users
- Food can be both favourite and ignored in the same time.
- Meals consist on many foods. Food is used in many meals.
- Food has a type

In addition, there's Activity, which is a lonely entity. It doesn't interact with others entities within the database, but the database is used for storing Activity attributes, to make it easier to update them or add new in the future.

First Prototype

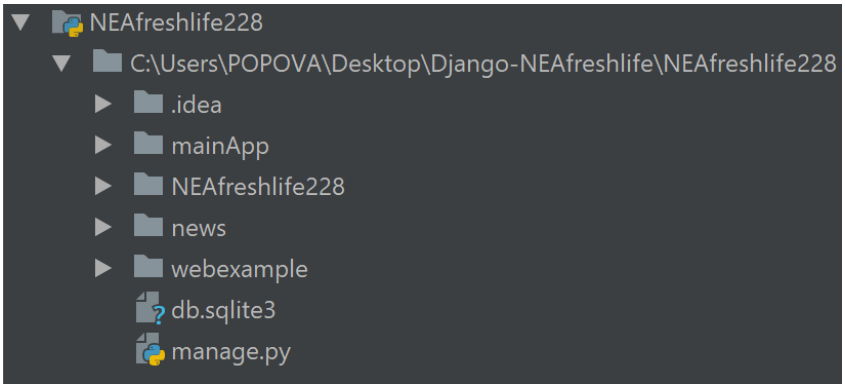
The purpose of the following prototype is to show that proposed application approach is realistic and success criteria are achievable. The bullet points of what the first prototype should be able to do are listed below:

- Successful running of the application on the local server
- All components(Python(Django) + HTML + CSS + JS + Jinja + Sqlite should be engaged
- Data from Python displayed as raw HTML
- Implemented Many-to-many relationship in the database
- Viewing data from database. Clear demonstration that an entity instance and its relative instance can be accessed
- Basic Registration and Login system

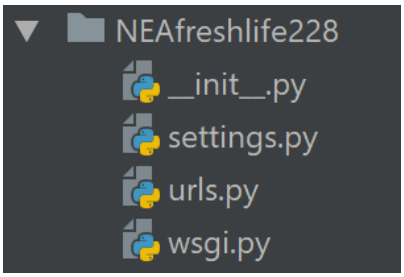
Prototype overview

Django applications have very complicated file structure. It is impossible to describe everything and it is also not relevant for just a prototype. Only key the points are going to be reviewed.

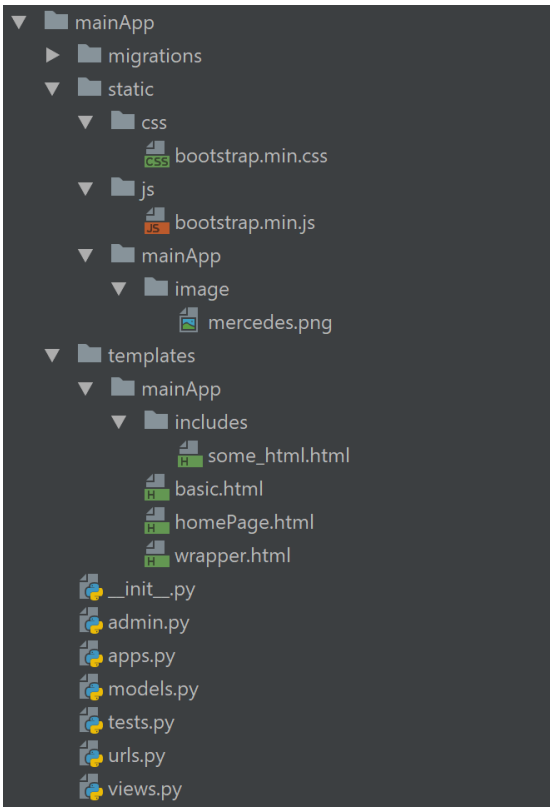
Apps



The application is called NEAfreshlife228. It has three created apps: mainApp, news and webexample. It also has NEAfreshlife228 app itself, which was created automatically by Django.

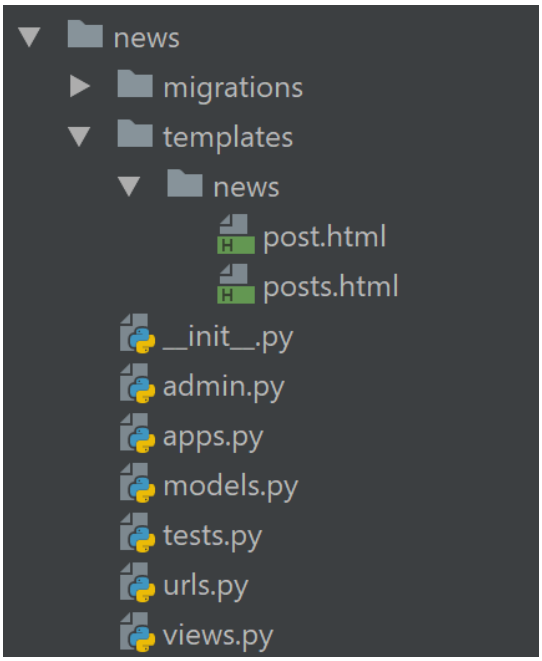


mainApp



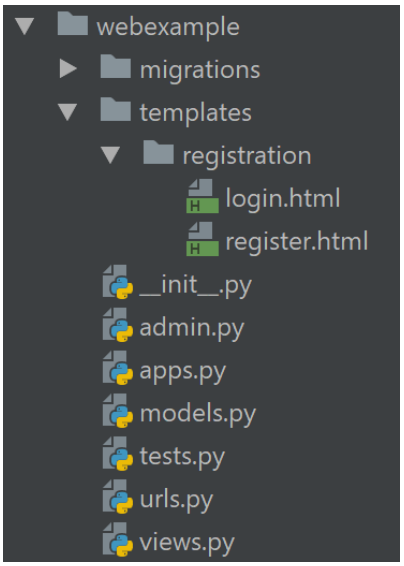
The mainApp is the home page of the prototype. It has the main template file *wrapper.html* and few more small HTML files which extend *wrapper.html*, so different pages use the code from the same file.

news app



The news app is the app where the test database was implemented. The app views articles as a list in *posts.html* and a single article in *post.html*.

webexample app

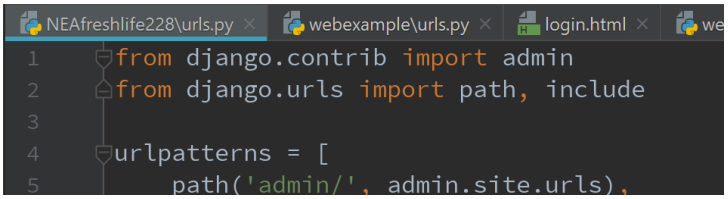


Webexample app was the first app created with intention to just test Django, output “Hello world” and nothing more. Later on it was decided to hold files related to user authentication.

Evidence of all key components being used

Python Django:

The big part of the file structure, most python files were automatically created after inserting a command: “django-admin startproject NEAfreshlife228” in the Windows PowerShell, which is a way to start a typical Django application.



In addition, methods from Django library are widely used across the application.

HTML, CSS, JS and Jinja:

HTML, CSS and Java Script are well known trinity of front-end web development. Jinja is a template design engine, which allows to display context variables taken from python or database as raw html, makes it possible for html files used by other html files in several ways. It can make many other useful things.

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Page title</title>
    {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}" type="text/css">
</head>
<body>
    <div class="jumbotron">
        <div class="container">
            <h1>{% include "mainApp/includes/some_html.html" %}</h1>

            {% block content %}
            {% endblock %}

            <p>
                <a href="/" class="btn btn-success btn-lg">Main page</a>
                <a href="/news" class="btn btn-primary btn-lg">News »</a>
                <a href="/contact" class="btn btn-info btn-lg">Contacts »</a>
            </p>
        </div>
    </div>
    <div class="container">
        <img src="{% static 'mainApp/image/mercedes.png' %}" class="img-thumbnail">
        <hr>
        © All rights reserved
    </div>
</body>
</html>

```

mainApp/templates/mainApp/wrapper.html

This html file is the main template of the web based application. For visual part of the page, Bootstap was used. It is an online library of CSS styles and JS functional elements. Wrapper.html loads bootstrap.min.css file and uses its styles. Statements in “{% %}” brackets is Jinja. The red arrow points on the most important Jinja element. It is a block tag which is opened and closed immediately afterwards, It is empty, when looking at the wrapper.html alone, but in fact the content of the file will be inherited by other html files.

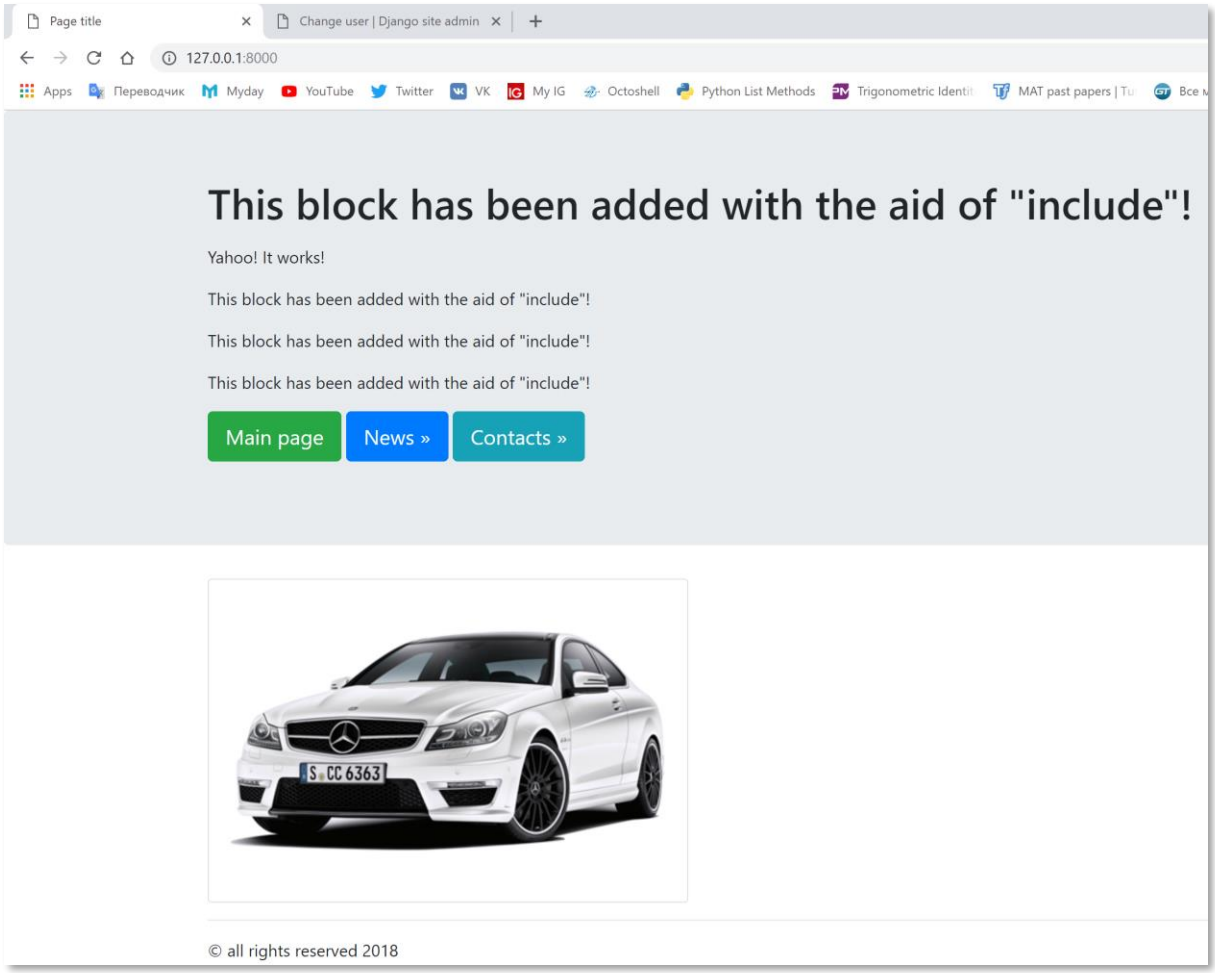
As an example the homepage:

wrapper.html | homePage.html

```
{% extends "mainApp/wrapper.html" %}

{% block content %}

<p>Yahoo! It works!</p>

{% include "mainApp/includes/some_html.html" %}
{% include "mainApp/includes/some_html.html" %}
{% include "mainApp/includes/some_html.html" %}

<script> alert('hello world')</script>
{% endblock %}
```

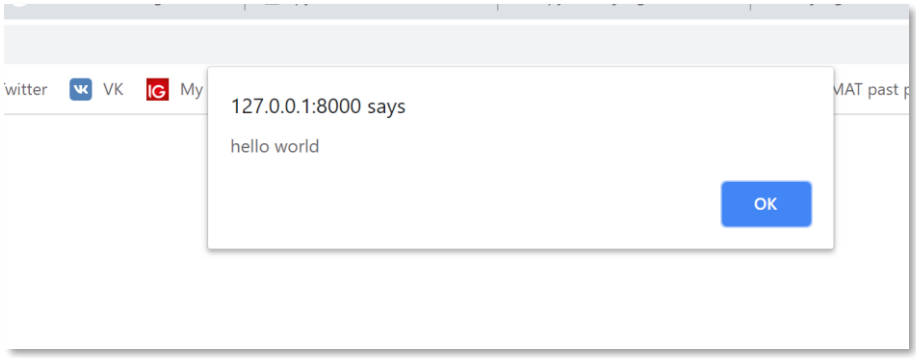
mainApp/templates/mainApp/homePage.html

The first line tells that this file is an extension of the wrapper.html, i.e. the browser first goes to wrapper.html and prints it out, when it comes back to the homePage.html, and everything inside block content tag goes inside similar block in the wrapper.



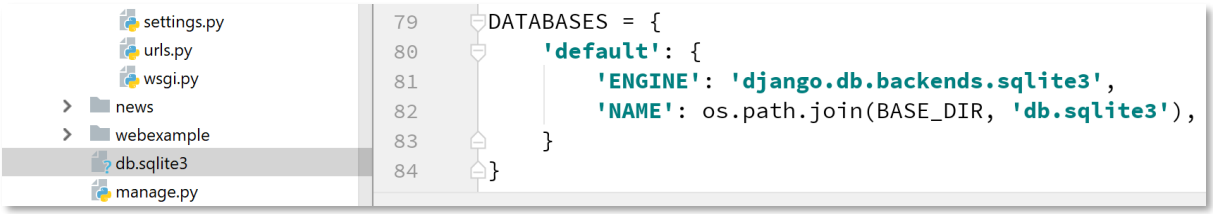
View in the browser. Everything between level 1 geader and three buttons is homePage.html. The rest is taken from the wrapper.html. In fact, every html page within the application will use wrapper.html via the same technique.

Note, that before the page is loaded, the short message occurs. It is caused by <script></script> tag in the homePage.html. It is the laziest way of using JavaScript, but it is sufficient to show that application can use the tool.



SQLite:

The application is using SQLite as a database engine. The following component is not used explicitly, i.e. no SQLite queries been hardcoded. The database is operated with the aid of special classes and methods from Django library.



Screenshot from NEAfreshlife228/settings.py

However, the settings.py file, which contains meta data of the application, clearly shows on the lines 79-83 that the sqlite3 is set as the database engine. In addition db.sclite3 file is in the project directory.

Displaying data from Python as HTML

If the user is on the Home Page, he can click on the 'Contacts' button. It sends him to 'contacts/' url Here's what will happen in the case and some explanations why.

mainApp/urls.py:

```
1  from django.urls import path, include
2  from . import views
3
4  urlpatterns = [
5      path('', views.index, name="index"),
6      path('contact/', views.contact, name="contact"),
7  ]
```

The code works out the page addresses.
Line 6: "If the url has 'contact/' use contact() method from mainApp/views.py file"

mainApp/views.py:

```
1  from django.shortcuts import render
2
3
4  def index(request):
5      return render(request, 'mainApp/homePage.html')  # additional parameter dictionary
6
7
8  def contact(request):
9      return render(request,
10                   'mainApp/basic.html',
11                   {'values': ["if you have any questions feel free to call me",  # additional parameter dictionary
12                               "(777)999-456-13",
13                               "contact@gmail.com"]})
14
```

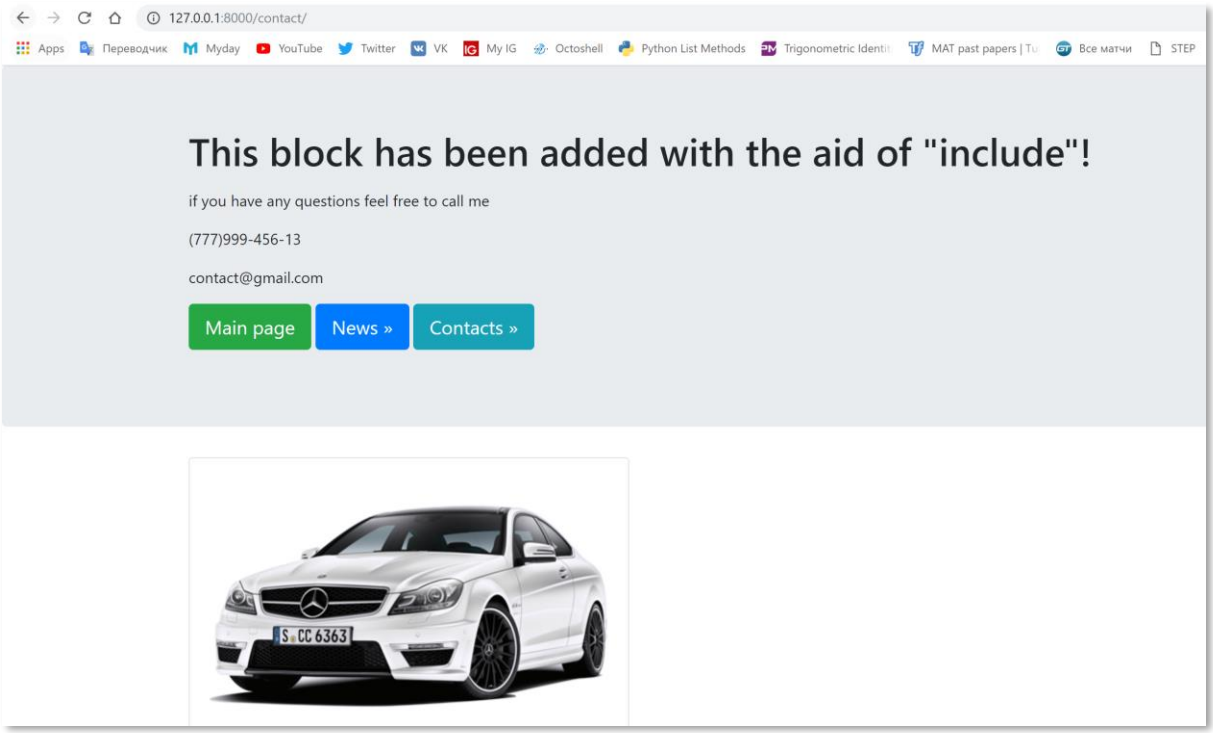
Lines 8-13: contact() method. It does two key things: it calls(opens) *mainApp/basic.html* and it sends it a dictionary. The dictionary has one element, which is called 'values' and it is a list with 3 items of contact information. The dictionary declares 'values' as a context variable within the html file.

mainApp/templates/mainApp/basic.html:

```
1  {% extends "mainApp/wrapper.html" %}
2
3  {% block content %}
4      {% for val in values %}
5          <p>{{val}}<p>
6      {% endfor %}
7  {% endblock %}
```

Lines 4-6: Jinja for loop tag is used to list all three items of 'values' list in three separate paragraph.

The resultant page looks like this:



Conclusion: It has just been showed that it is possible to display data from Python as raw HTML.

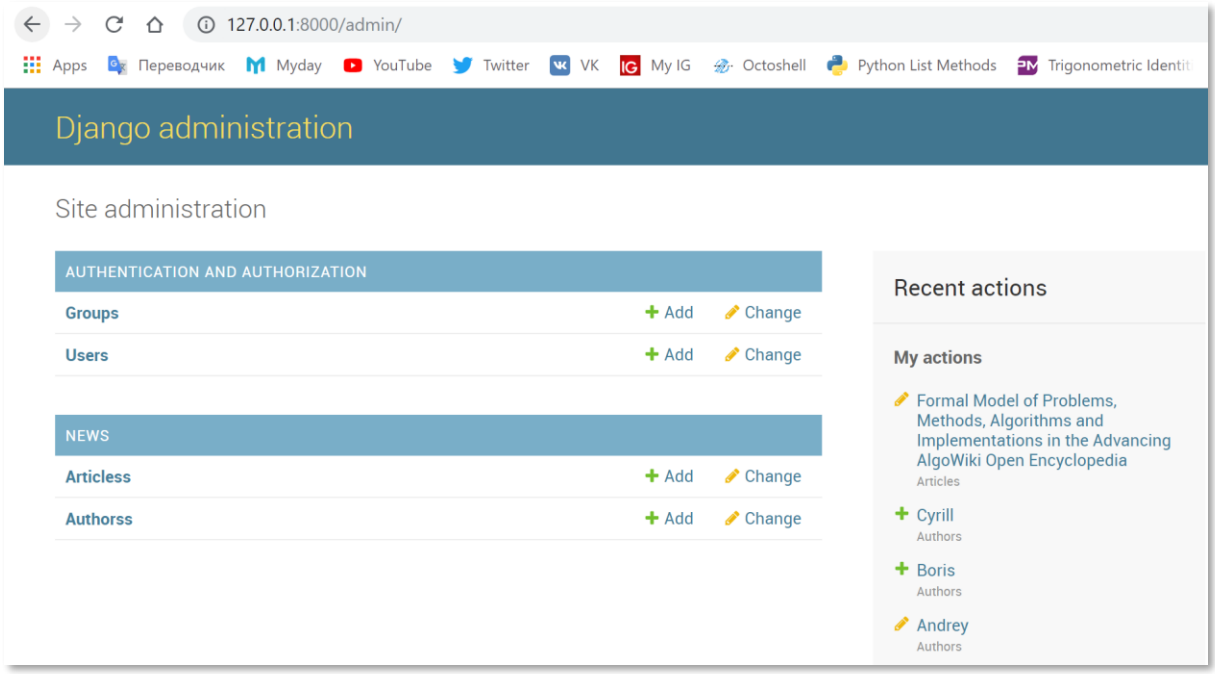
Simple database. Many-to-many relationship implementation.

news/models.py

```
1   from django.db import models
2
3   # this is news/models.py
4
5
6   class Authors(models.Model):
7       name = models.CharField(max_length=120)
8       # Articles = models.ManyToManyField(Articles)
9
10      def __str__(self):
11          return self.name
12
13
14  class Articles(models.Model):
15      title = models.CharField(max_length=120)
16      post = models.TextField()
17      date = models.DateTimeField()
18      authors = models.ManyToManyField(Authors)
19
20      def __str__(self):
21          return self.title
```

Django models were used for the database. Each table is defined as a class. But all these classes inherit from django.db.model.Model class (lines 6 and 9). For test purposes the simple database has been created. It has two entities: Articles and Authors. Articles have three attributes: title, the text content(post) and the date(lines 15-17). Authors just have name(line 7). Line 18 creates many-to-many

field, which in the database itself is implemented as a separate junction table (but it never explicitly will be shown, unless outside CMS being used), From python objects point of view, ManyToManyField() method allows two entities access each other. Class models.Model has auto-incremented id field as a default, so primary key attribute doesn't have to be defined.



New data can be added, updated or deleted in Django Admin dashboard. It can also be hardcoded in python shell.

Viewing data from database

The test items has been added to the database:

There are four articles with title date and some text in it. The goal is to display the data in two ways:

- Display article titles and dates as a list
- Display one single article and display the names of all its authors

```
1   from django.urls import path, re_path, include  #
2   from news.views import (
3       ArticleListView,
4       ArticleDetailView,
5   )
6
7   urlpatterns = [
8       path('', ArticleListView.as_view()),  # class views
9       path('<int:pk>', ArticleDetailView.as_view()),
10  ]
```

news/urls.py
Line 8-9: if after 'news/' there's an empty string in the url, use ArtlicleListView.as_view(). If there's an integer which is a primary key, use ArticleDetailedView.as_view(), the views are defined below.

news/views.py

```
1   from django.views.generic import DetailView, ListView
2   from news.models import Articles, Authors
3
4
5   class ArticleListView(ListView):
6       template_name = 'news/posts.html'
7       queryset = Articles.objects.all().order_by("-date")[:20]
8
9
10  class ArticleDetailView(DetailView):
11      template_name = 'news/post.html'
12      model = Articles
```

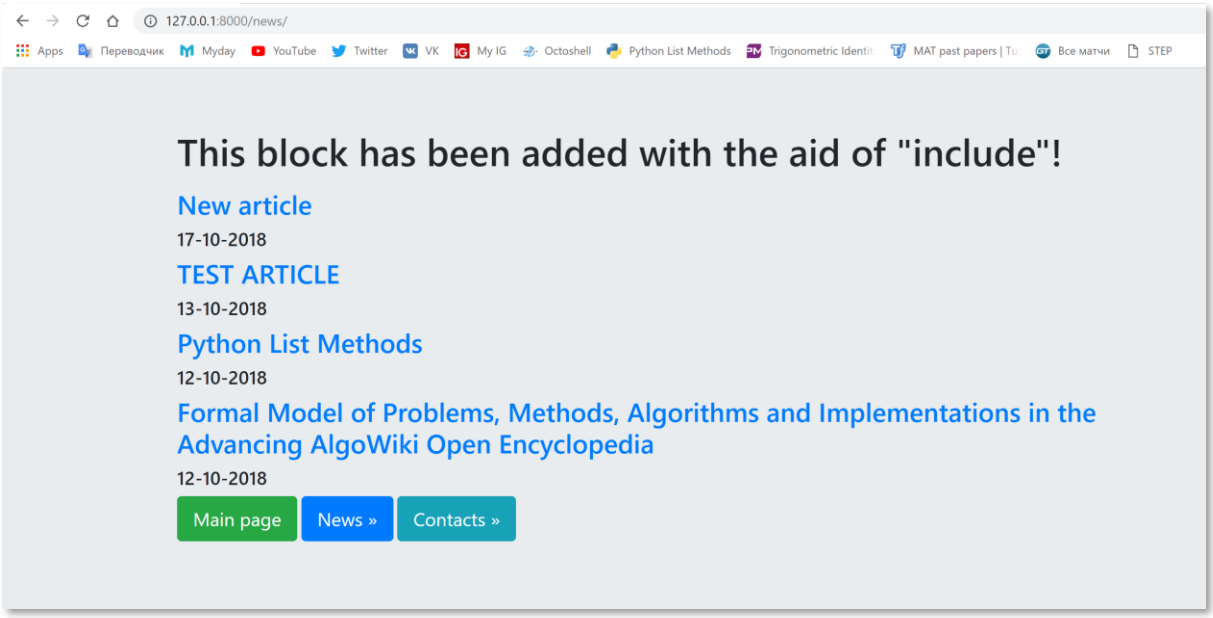
Lines 5-12: defining the class based views.
Line 6 and 11 shows which html file should be used in each of the views.

1) List view:

news/templates/news/posts.html

```
1   {% extends "mainApp/wrapper.html" %}
2
3   {% block content %}
4       {% for post in object_list %}
5           <a href="/news/{{post.id}}"><h3>{{post.title}}</h3></a>
6           <h5>{{post.date|date:"d-m-Y"}}</h5>
7       {% endfor %}
8   {% endblock %}
```

Lines 4-7: for loop displays the title and date of each of the object in the list. The title is a link to the article own page.



List view in the browser.

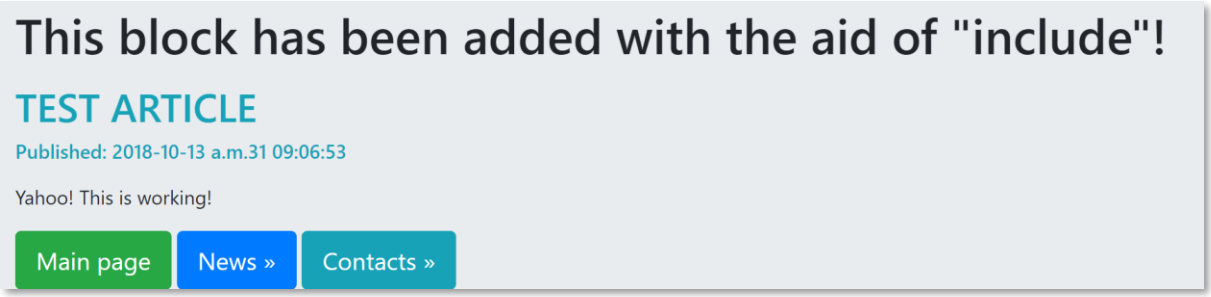
2) Detail view

News/templates/news/post.html

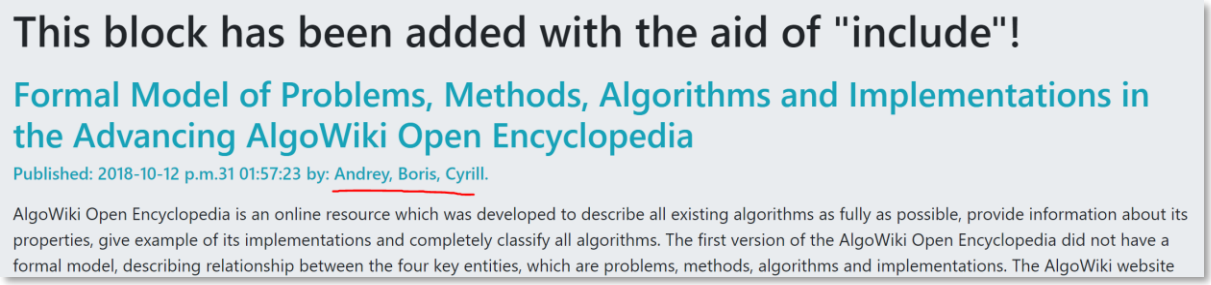
```
1   {% extends "mainApp/wrapper.html" %}
2   {% block content %}
3
4       <h2 class="text-info">{{articles.title}}</h2>
5       <h6 class="text-info">
6           Published: {{articles.date|date:"Y-m-d at h:i:s"}}
7           {% if articles.authors.all %}
8               by:
9               {% for author in articles.authors.all %}
10                  {% if forloop.counter == articles.authors.all|length %}
11                      {{author.name}}.
12                  {% else %}
13                      {{author.name}},
14                  {% endif %}
15              {% endfor %}
16          {% endif %}
17      </h6>
18      <p>{{articles.post|safe|linebreaks}}</p>
19
20  {% endblock %}
```

Line 4: display title of the article
Line 6: display the date and time in a special way (achieved by filter)
Line 7-16: “if article has authors, display them”
Line 9-15: lists all the authors in a line. If clause inside the loop is used to separate author names with coma, but the last author has a ‘.’ symbol after it.
Line 18: display the text (post) of the article.

Here is the result:



news/5
The article doesn't have authors, so it doesn't mention any of them



news/1
The article has three authors, so it lists them out (underlined with red)

User authentication.
When Django application is created, the special app called django.contrib.auth creates automatically.

```
1   from django.contrib import admin
2   from django.urls import path, include
3
4   urlpatterns = [
5       path('admin/', admin.site.urls),
6       path('', include('mainApp.urls')),
7       path('news/', include('news.urls')),
8       path('accounts/', include('django.contrib.auth.urls')),
9       path('webexample/', include('webexample.urls'))
10  ]
```

NEAfreshlife228/urls.py

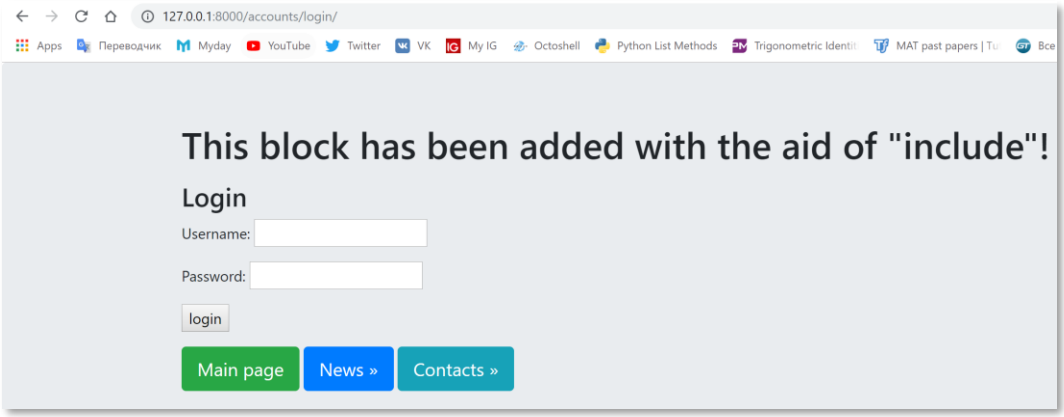
In addition the model(table) for users is also created by Django. So the remaining part of the login system is to create view for registration and html files for login and registration.

1) LOGIN:

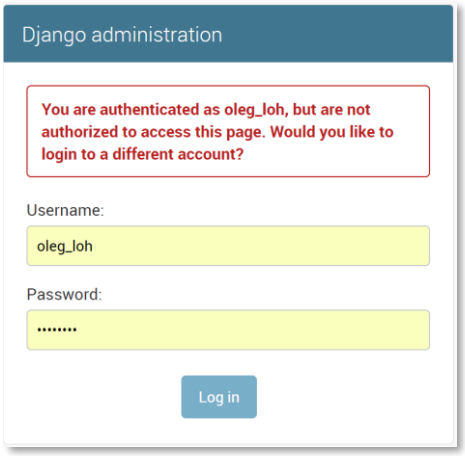
webexample/templates/registration/login.html

```
1   {% extends "mainApp/wrapper.html" %}
2   {% block content %}
3       <h3>Login</h3>
4
5       {% if form.errors %}
6       <p>There's something wrong with what you entered!</p>
7       {% endif %}
8       {% if next %}
9       <p>Hey, you can't access that page</p>
10      {% endif %}
11
12      <form method="post" action="{% url 'login' %}">
13          {% csrf_token %}
14          <p>Username: {{ form.username }}</p>
15          <p>Password: {{ form.password }}</p>
16          <p><input type="submit" value="login"></p>
17          <input type="hidden" name="next" value="{{ next }}">
18      </form>
19  {% endblock %}
```

Lines 12-18: the login form.
Line 12: action attribute = {% url 'login' %} refers to the login view, which doesn't have to be created, but has to be referred in the login form.



This is how the form looks in the browser. The way to ensure that authorization actually happens is coding pages to make them able to recognize users and print their name, for example.



…or trying to access django administration, while being authorized as non-admin user. Lazy, but sufficient evidence.

2) REGISTER

webexample/views.py

```
13  def register(request):
14      if request.method == 'POST':
15          form = UserCreationForm(request.POST)
16          if form.is_valid():
17              form.save()
18              username = form.cleaned_data['username']
19              password = form.cleaned_data['password1']
20              user = authenticate(username=username, password=password)
21              login(request, user)
22              return redirect('index')
23      else:
24          form = UserCreationForm()
25      context = {'form': form}
26      return render(request, 'registration/register.html', context)
```

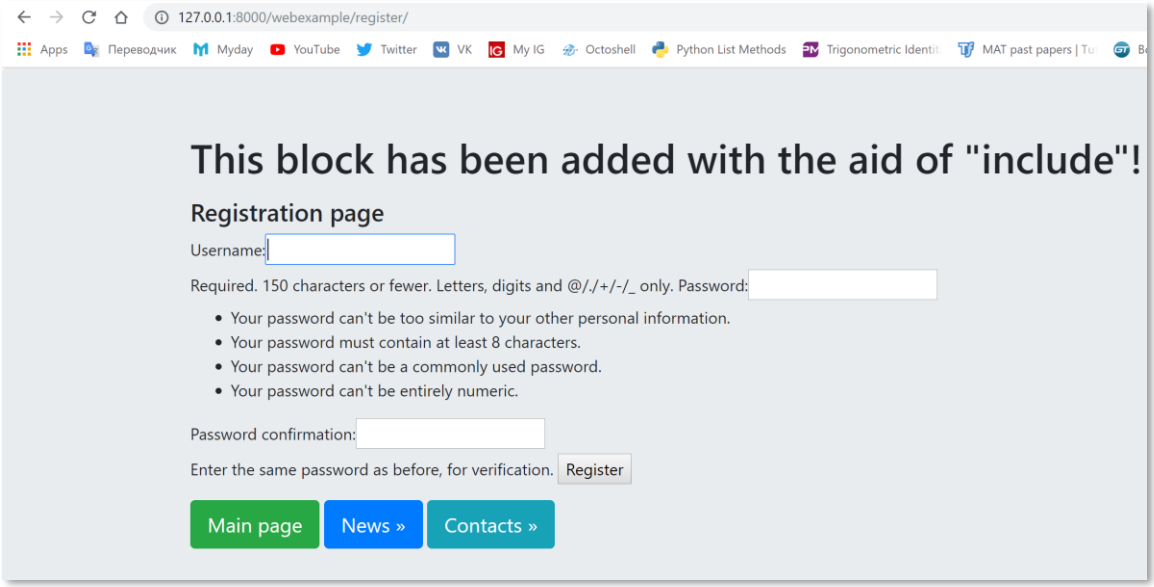
register method-based view.
Lines 14-22 are used when the form is filled and submitted. They add new user to the database
Lines 24-25: Creating form and declaring it as context variable for register.html
Line 26: Accessing register.html file and sening it a dictionary.

```
1   {% extends "mainApp/wrapper.html" %}
2   {% block content %}
3
4       <h4>Registration page</h4>
5
6       <form method="post" action="{% url 'register' %}">
7           {% csrf_token %}
8
9           {% if form.errors %}
10          <p>There are errors in the form!</p>
11          {% endif %}
12
13          {{ form }}
14
15          <input type="submit" value="Register"/>
16      </form>
17
18  {% endblock %}
```

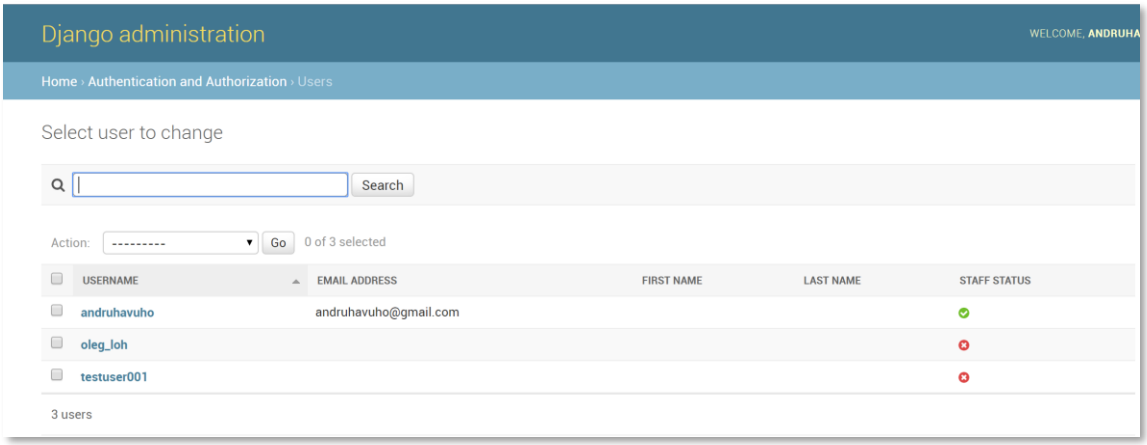
webexample/templates/registration/register.html

Line 6: form has method attribute equal ‘post’. It is related to line 14 of the previous screenshot.
Lines 6-16: register form html tag, Line 13: the form itself

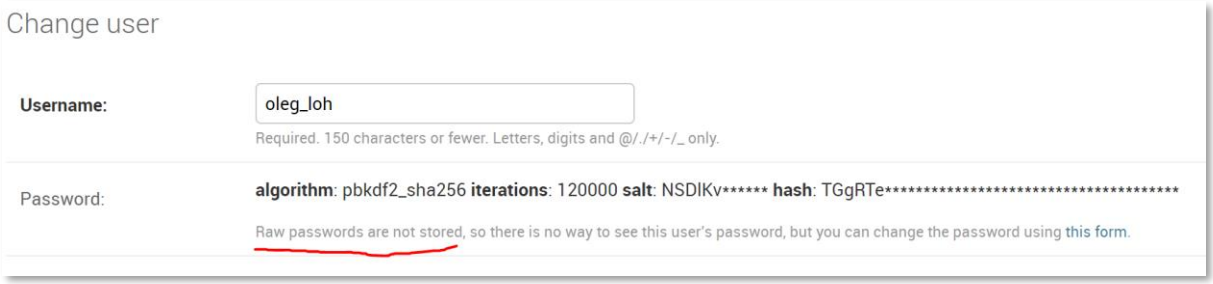
View from the browser



Evidence of new users appearing in the database can be found in the Django administration dashboard:



The good thing about the python Django that it already takes care about safety of Users' personal information, so that even admin can't see their passwords.

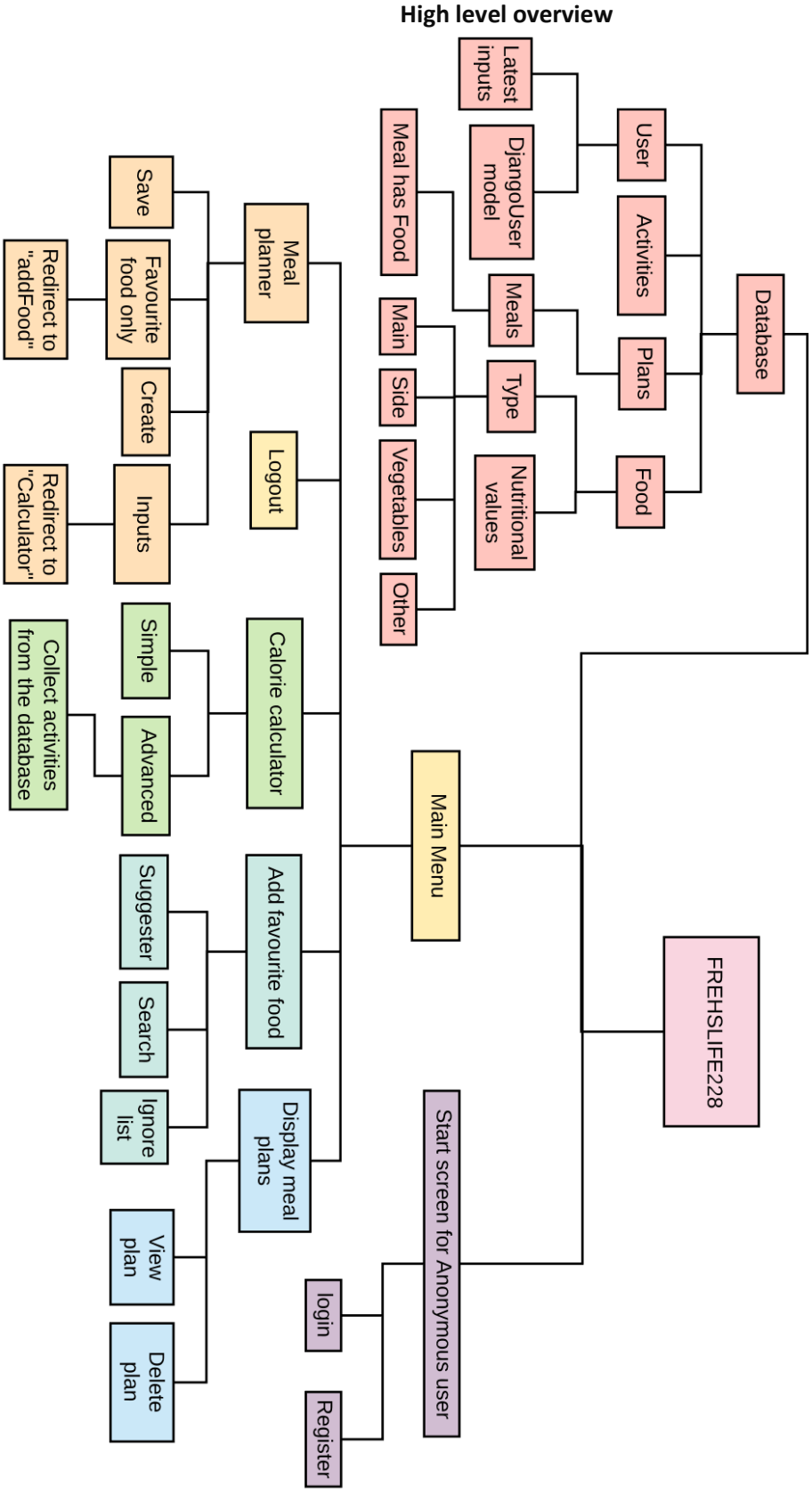


Prototype conclusion.

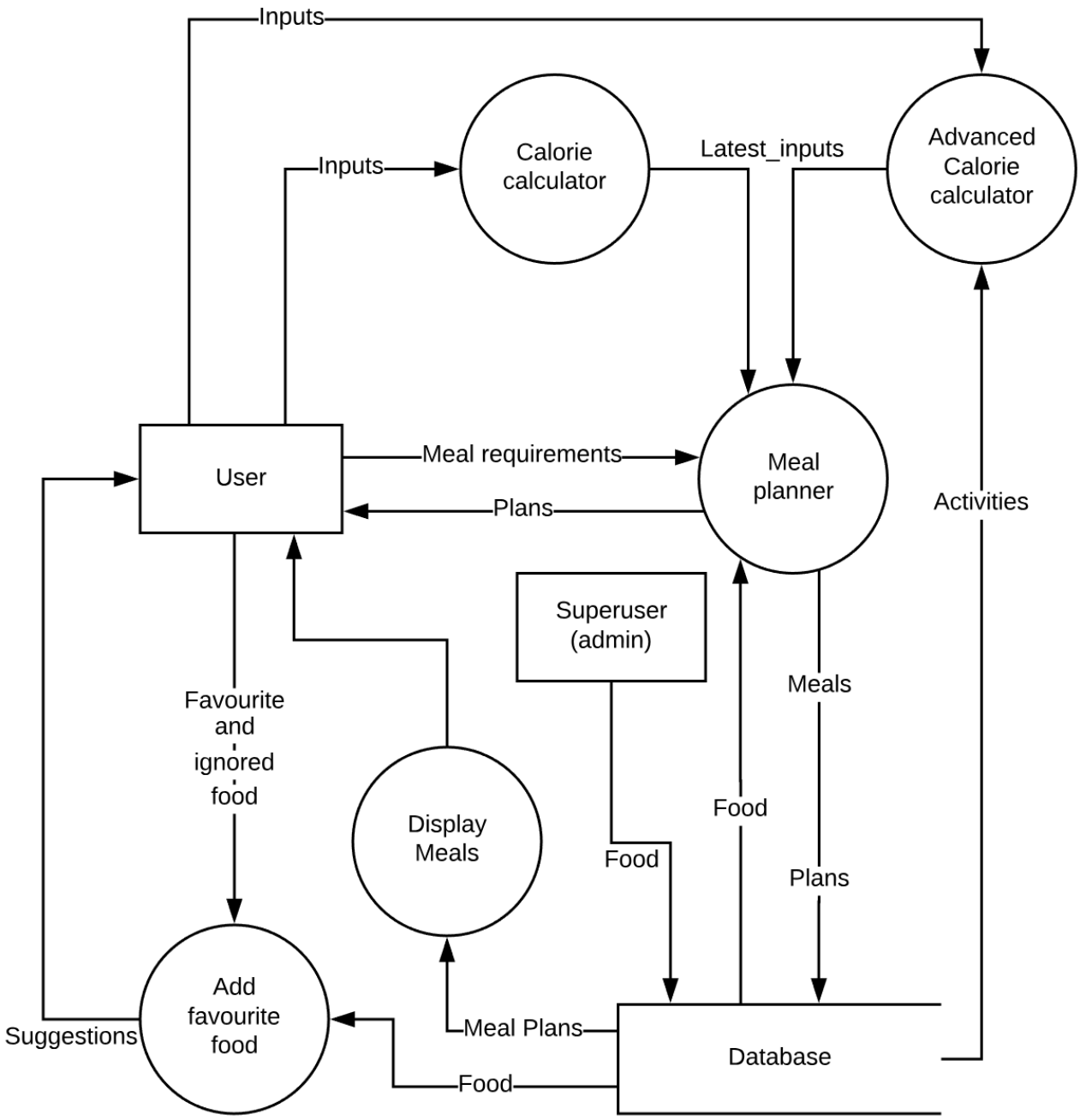
As a result, all the bullet points from prototype goals list are satisfied. The planned application is doable. All global ideas are already implemented in the prototype, so it can be used as a draft for the application itself.

DESIGN

Program Overview.



Data Flow Diagram

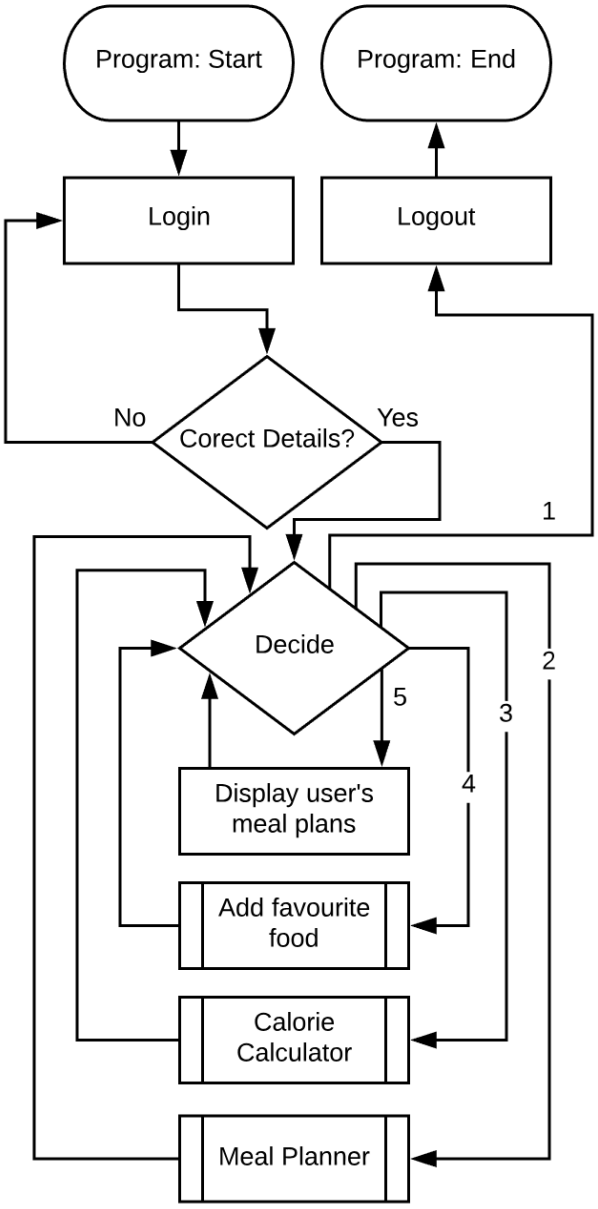


Algorithms description

Flow charts

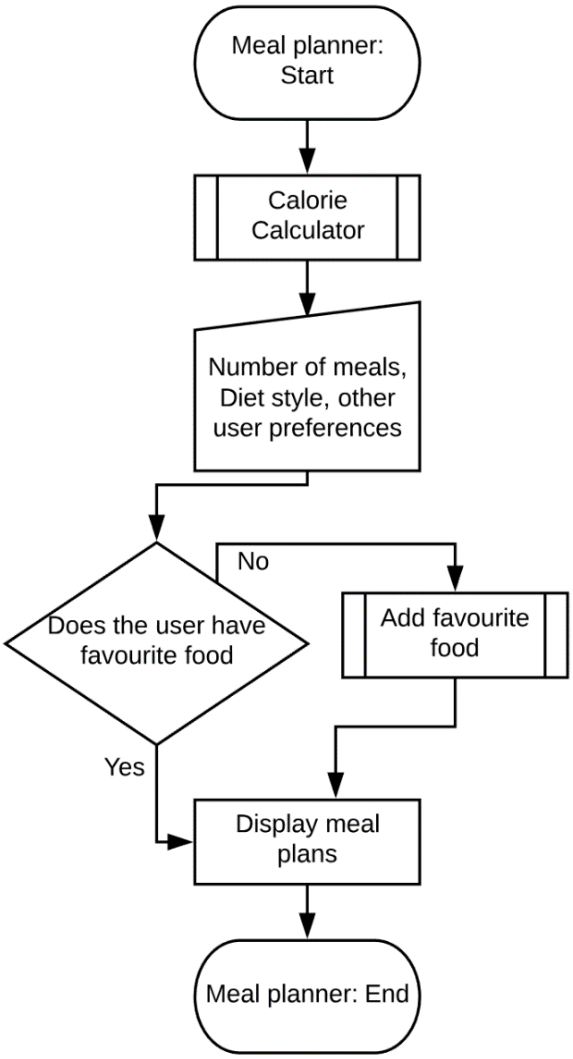
Flow chart is one of the best ways to give visual representation of the algorithms. There are four diagrams in this subsection with one describing the full application and three representing smaller processes. Note that the diagrams are abstract and level of precision is not close to absolute.

Global flow chart



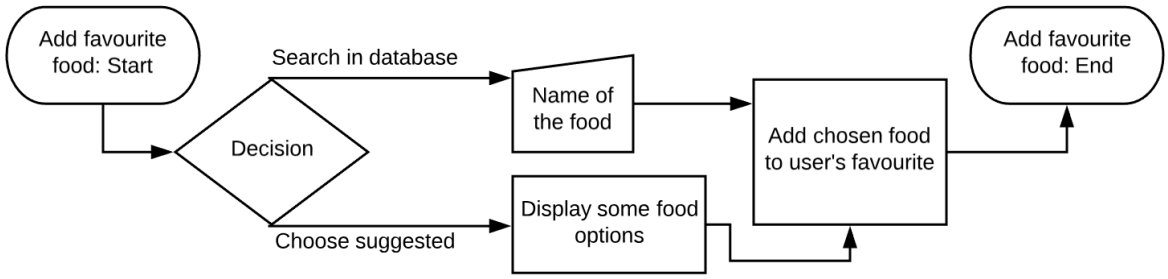
The image above shows the flow chart which implies that as the program starts, user has to login to his personal account, and if the personal details are correct, he will see a main menu with four options: Logout(1), Meal Planner(2), Calorie Calculator(3), Add Favorite food(4) and Display meal plans(5). The options 2-5 are looped, which means, that by the end of each process the user comes back to the main menu until the user logs out.

Meal Planner flow chart



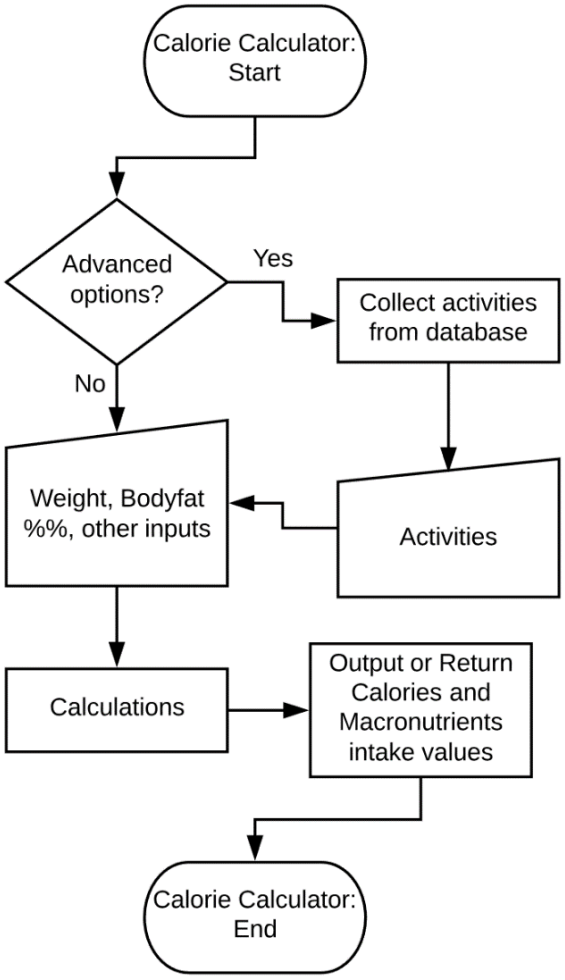
The process of meal planning always starts with calculating calories, because it is essential information for meal planner. After these values been returned by the Calorie Calculator process, user has to input the details representing his personal preferences for the following meal, such as number of meals he would like to have, the style of diet. The next step is working out the meals themselves, but if the current user doesn't have favorite food in the database, the program will suggest him add some. After the meals have been worked out they get displayed for the user.

Add Favorite Food flowchart



And according to the diagram on the top. There are two ways to choose favorite food: via automatic suggestions and via type search.

Calorie Calculator flow chart



Calorie calculator is a process which require user to input his weight, bodyfat ratio and describe his activity. In case if the user chose to use advanced option, he creates a mini-list of activities he takes part in during a week. This and some other possible inputs are processed and the total energy expenditure gets calculated. If the calculator was used alone, the values are displayed. If it was used as a part of the meal planner, the values are returned to the meal planner.

Pseudocode functions

Pseudocode is useful when it comes to the codes design and planning ahead the programming techniques and methods. The following screenshots reveal more details about some of the algorithms described in flowcharts.

Simple Calculator subroutine

```
SUBROUTINE SimpleCalc()
    Weight <-- USERINPUT
    BodyFatRatio <-- USERINPUT

    LBM <-- (100 - BodyFatRatio) / 100
    BMR <-- 370 + 21.6 * Weight * LBM

    Choice <-- UserMultipleChoice()
    ActivityCoef <-- 1

    IF choice = 1 THEN
        ActivityCoef <-- 1.20
    ELSE IF choice = 2 THEN
        ActivityCoef <-- 1.35
    ELSE IF choice = 3 THEN
        ActivityCoef <-- 1.55
    ELSE IF choice = 4 THEN
        ActivityCoef <-- 1.75
    ELSE IF choice = 5 THEN
        ActivityCoef <-- 1.95
    ELSE
        Error()
    END IF

    TEE <-- BMR * ActivityCoef
    Prots <-- Weight * 1.5          # Proteins
    Fats <-- Weight * 0.7
    Carbs <-- (TEE - Prots * 4 - Fats * 9) / 4

    RETURN TEE, Prots, Fats, Carbs
END SUBROUTINE
```

The *SimpleCalc()* algorithm is quite simple. It starts by taking the values of weight and body fat percentage and calculates the Base Metabolic Rate (BMR) using the Katch McArdle equation. But the Total Energy Expenditure (TEE) is found by multiplying BMR by the coefficient representing user's life activity. *UserMultipleChoice()* is an abstract function which interacts with the multiple choice question on the interface and returns a certain coefficient, depending on the choice made by user. The coefficient is between 1.20 and 1.95 and the bigger it is, the more active the person is. After TEE been found, the recommended intake of Proteins, fats and Carbohydrates is worked out. In the given code, the assumption was made that 1.5 grams of protein and 0.7 grams of fat per kilogram of bodyweight are recommended. Proteins and Fats are found first and Carbohydrates are found in the way that all the remaining calories come from them.

Advanced Calculator subroutine

```
# Activities are taken from DataBase.
# Activities example: [[name],[Epkgh],[time/hours]]
# [[["football"],[4.4],[2]],[['yoga'],[1.8],[2.5]]]

SUBROUTINE AdvancedCalc()
    Weight <-- USERINPUT
    BodyFatRatio <-- USERINPUT

    LBM <-- (100 - BodyFatRatio) / 100
    BMR <-- 370 + 21.6 * Weight * LBM

    EAT <-- 0
    Time <-- 0
    FOR i <-- 0 TO LEN(Activities)
        EE <-- Activities[i][1] * Activities[i][2]
        EAT <-- EAT + EE * Weight
        Time <-- TIme + Activities[i][2]
    END FOR
    EATcoef <-- Time * 0.05

    Choice <-- UserMultipleChoice()
    NEATcoef <-- 1

    IF choice = 1 THEN
        NEATcoef <-- 1.20
    ELSE IF choice = 2 THEN
        NEATcoef <-- 1.25
    ELSE IF choice = 3 THEN
        NEATcoef <-- 1.30
    ELSE IF choice = 4 THEN
        NEATcoef <-- 1.35
    ELSE IF choice = 5 THEN
        NEATcoef <-- 1.40
    ELSE
        Error()
    END IF

    TEE <-- BMR * (NEATcoef + EATcoef) + EAT
    Prots <-- Weight * 1.5
    Fats <-- Weight * 0.7
    Carbs <-- (TEE - Prots * 4 - Fats * 9) / 4

    RETURN TEE, Prots, Fats, Carbs
END SUBROUTINE
```

The *AdvancedCalc()* function is more similar to *SimpleCalc()* than different, so a lot of statements are the same. However, Advanced Calculator uses different slightly method for finding TEE. The FOR loop has been added. It is assumed that the activities are already defined and stored in a list and it is also assumed that the list has the configuration described in the comment statements on top of the screenshot. Every iteration of the loop finds the energy spent on the activity depending on the time and weight and adds it to the total Exercise Associated Thermogenesis (EAT). Activity coefficient has been split on two components: *EATcoef* and *NEATcoef.* The NEAT coefficient is responsible for Non-Exercise activity and depends on the multiple choice question. The EAT coefficient depends on the

number of hours of training. It is relevant because the training boosts metabolism, not only spends energy in the process.

Meal Planner subroutine

```
SUBROUTINE MealPlan(Weight, TEE,  # i.e. Calories
                    MealNum, MealPlanName,
                    Prots <-- 1.5, Fats <-- 0.7,
                    Carbs <-- False)
    Vegetables <-- Round(350 * Weight/70, 2) # 2s.f.
    Prots <-- Weight * Prots
    Fats <-- Weight * Fats

    # IF NOT Fats THEN
    #     Fats <-- (TEE - Prots * 4 - Carbs * 9) / 4
    # ELSE
    #     Fats <-- Weight * Fats
    # END IF

    IF NOT Carbs THEN
        Carbs <-- (TEE - Prots * 4 - Fats * 9) / 4
    ELSE
        Carbs <-- Weight * Carbs

    VegPerMeal <-- Vegetables / MealNum
    ProtsPerMeal <-- Prots / MealNum
    CarbsPerMeal <-- Carbs / MealNum
    FatsPerMeal <-- Fats / MealNum

    AddMealPlanToDataBase(MealPlanName)

    FOR i <-- 0 TO MealNum
        # Food objects
        MainCourse <-- FavouriteMain()
        Garnish <-- FavouriteGar()
        Vegetable <-- FavouriteVeg()
        Other <-- FavouriteOther()

        MainCourse.Serving <-- ProtsPerMeal / MainCourse.Prot
        Vegetable.Serving <-- VegPerMeal / 100

        a <-- FatsPerMeal - Vegetable.Serving * Vegetable.Fats
        b <-- Garnish.Fats / Garnish.Carbs
        c <-- CarbsPerMeal + Vegetable.Serving * Vegetable.Fats
        d <-- MainCourse.Serving * MainCourse.Carbs / MainCourse.Fats
        e <-- Other.Fats - Garnish.Fats * Other.Carbs / Garnish.Carbs
        Other.Serving <-- ( a - b * (c - d) ) / e
        f <-- Other.Serving * Other.Carbs
        Garnish.Serving <-- (c - d - f)/Garnish.Carbs

        # Conversion to grams
        MainCourse.Serving <-- MainCourse.Serving * 100
        Garnish.Serving <-- Garnish.Serving * 100
        Vegetable.Serving <-- Vegetable.Serving * 100
        Other.Serving <-- Other.Serving * 100

        Meal = [MainCourse, Garnish, Vegetable, Other]
        AddMealToPlan(MealPlanName, "Meal" + INT_TO_STRING(i), Meal)
    END FOR
END SUBROUTINE
```

The MealPlan() subroutine has the purpose of creating the menus for the meals and determine the servings. The function has compulsory parameters, such as weight, number of calories, number of meals and the name of the plan. There are optional parameters of number of grams of proteins, fats and carbohydrates per kilogram of bodyweight. They have default values of 1.5 for proteins and 0.7 for fats, which relates to the assumption made in previous algorithms. The Carbohydrates have default

value ‘False’, which means that it is chosen as “flexible macronutrient” or main source of energy and its value is adjusted, depending on the number of fats, proteins and calories. It is assumed that either fats or carbs have that dependent role, i.e. initially equal boolean False. In the code there’s a commented statement which can be used if fats are chosen as “flexible nutrient”. It might be used in the future for ketogenic diet for example. *AddMealPlanToTheDataBase()* is an abstract function which has self-explanatory name and represents all the necessary procedures for adding meal plan to the database. The meals are added to the meal plan by a FOR loop, there one iteration is one meal. For every meal the program picks one dish from each of the four categories: Main course, garnish, vegetable and other. It is assumed that foods are represented with objects, which have fields (attributes) representing the nutritional values of food. All dishes have their own properties and the such servings have to be calculated that the total amount of nutrients satisfies the required intake. The variables *a, b, c, d, e* and *f* are not necessary, but they are used in the code to break down one huge line of math into something more readable. The math statements from this algorithm solve some simultaneous equations there servings are unknowns and nutritional properties are constants. The found values should be multiplied by 100 to get the values in grams. At the end of each iteration the determined values are stored in the database which is represented with *AddMealToPlan()* abstract function.

Improved way to evaluate best fit servings of food.

```
A <-- matrix([  <!--# content of nutrients in food-->
    [Main.protein,     Side.protein,     Veg.protein,     Other_p],
    [Main.carbs,       Side.carbs,       Veg.carbs,       Other_c],
    [Main.fat,         Side.fat,         Veg.fat,         Other_f],
    [Main.healthscore, Side.healthscore, Veg.healthscore, Other_h],
    [Main.calories,    Side.calories,    Veg.calories,    Other_kcal],
])

b <-- column_vector([   <!-- # requirements column vector  -->
    prots,
    carbs,
    fats,
    healthscore,
    calories,
])

r <-- lsq_linear(A, b, bounds<--(0.4, 6))  <!--# Constrained least squares -->

Main_serving <-- r[0]
Side_serving <-- r[1]
Veg_serving <-- r[2]
Other_serving <-- r[3]
```

The image above shows the pseudocode of the part of the meal planner algorithm. The code assign matrix A, which is a representation of all equations (for each nutrient or value) in the system and column vector b with the requirements of each of the value. Then the algorithm finds the least squares approximation of the solution of the equation with constraints such that servings cannot be less than 40g and more than 600g. The advantage of such method is that it solves the problem no matter if the solution exist or not. Lsq_linear() is real method which exist in Python library Scipy.optimize.

Database Design

Step-by-step database normalization

The tables below show the example test data and the process of playing with it in order to show how the database evolved during the normalization.

Flat-file approach. Non-normalized data.

The table below shows displays the data about one user having a meal plan with five meals. All data is fitted in one table. Note that not all the features (Activities, junction tables) of the database are included in this example.

Username	Password	Plan_Name	Meal1	Meal2	Meal3	Meal4	Meal5
Andrey228	oljegloh1	big plan	oatmeal 150g, 3 milk 200ml	whole-wheat biscuits 100g, whey isolate 30g,	chicken filet 100g, pasta 250g, vegetables 130g	chicken filet 50g, salmon 50g, pasta 100g, vegetables 100g	chicken filet 100g, pasta 250g, vegetables 130g

It is not even in the first normal form, because meal descriptions are lists and are not atomic values. The advantage of denormalized data is that it is the easiest for humans to read. But it has many obvious disadvantages. In this particular example, the most noticeable con of the table is its lack of flexibility to storing meal plans with more than five meals. Data normalization would solve that problem. In addition, it makes the database more suitable for machines to operate, increases efficiency and also can reduce the chance of anomalies, caused by deleting or altering tables. Example of anomaly can be a piece of data, which became orphaned, when related data was accidentally deleted. Data normalization also ensures that the data dependency makes sense.

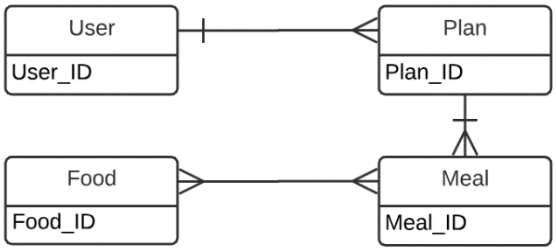
First normal form (1NF) .

In order to upgrade the table above to 1NF the following conditions have to be satisfied: Atomic (single) values in each cell, same type of entries in columns rows are uniquely identified.

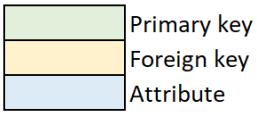
Username	Password	Plan_ Name	Meal index	dish1	Portion 1	dish2	Portion 2	dish3	Portion 3	dish4	Portion 4
Andrey228	oljegloh1	Big plan	1	oatmeal	150g	milk	100ml	null	null	null	null
Andrey228	oljegloh1	Big plan	2	Whole wheat biscuits	100g	whey isolate	30g	null	null	null	null
Andrey228	oljegloh1	Big plan	3	chicken filet	100g	pasta	250g	vegetables	130g	null	null
Andrey228	oljegloh1	Big plan	4	chicken filet	50g	salmon	50g	pasta	100g	vegetables	100g
Andrey228	oljegloh1	Big plan	5	chicken filet	100g	pasta	250g	vegetables	130g	null	null

The *Username* column makes sure each row is unique. The table above can already be operated much quicker than the previous denormalized one. But there are serious shortcomings in how the data is stored. The same Username, Password and Plan_Name are stored five times. There are some addition problems, because if new meals are added, Username and Password has to be entered every time. In addition, there are several null values stored due to the fact, that different meals can have different number of dishes. In addition, portion columns entries have VARCHAR (string) type, which is not handy because the numbers are used in calculations and converting them from strings all the time is inefficient. The table cannot be improved significantly without splitting it apart and adding more dimensions in it.

2NF and 3NF



In order to change from 1NF to 2NF all non-key columns have to depend on the key. The ER-diagram on the right clearly shows that there are four key entities which are dependent on their own key: User, Plan, Meal and Food. It also shows that one person may have many plans, plans have many meals and food and meals have many-to-many relationship, i.e. the same food can be ingredient of several meals and one meal consists of several foods. As a matter of facts, the database needs four tables responsible for the four entities and one extra table representing transactions between meals and food. *Meal_has_Food* is a junction table, which has an attribute Portion, because it belongs neither to food nor to meal but to the instance of the transaction between two.



User table:

User

User_ID	Username	Password
1	Andrey228	oljegloh1

User is the first entity of the database. The primary key is auto-incrementing integer. Username and password are fields which will be used for user authentication. Passwords created by users will be hashed and only then stored for security.

Plan table:

Plan

Plan_ID	User_ID	Plan_Name	Created
1	1	Big plan	2018-10-14 15:00:00*

Plan is the second entity. It is owned by one user. It has its own ID and a name and also stores an Identifier of the owner to refer him. Plan has its name and Datetime attribute has been added, because it might be useful for the user.
*datetime field representation may in fact be stored in a different form.

Meal table:

Meal

Meal_ID	Plan_ID	Meal_index
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5

Meal is the third entity in the database. It is owned by Plan entities in the same manner as User owns Plan. It has an index arrtibute, which represents the order number of a meal in a plan.

Food table:

Food

Food_ID	Food_Name	Unit
1	pasta	g

2	salmon	g
3	vegetables	g
4	chicken filet	g
5	oatmeal	g
6	Whole wheat biscuits	g
7	milk	ml
8	whey isolate	g

From an attribute in the table Food evolved into an entity. *Food_ID* and *Food_Name* have already existed in the 2NF version of the database, but it received a new attribute, *Unit*. Units are stored separately, because different food are measured in different units. In the actual database food will have much more attributes (proteins, fats, calories, …)

Meal-has-food table:

Meal_has_food

Meal_ID	Food_ID	Portion_size
1	5	150
1	7	150
2	8	100
2	10	30
3	6	100
3	3	250
3	5	130
4	6	50
4	4	50
4	3	130
4	5	100
5	6	100
5	3	250
5	5	130

There's also Meal_has_Food table, which is a junction table with additional fields used to represent m:n relationship between the Meal and Food entities. This table use two foreign keys as a composite primary keys, because the attribute *Portion* depends on the combination the two.

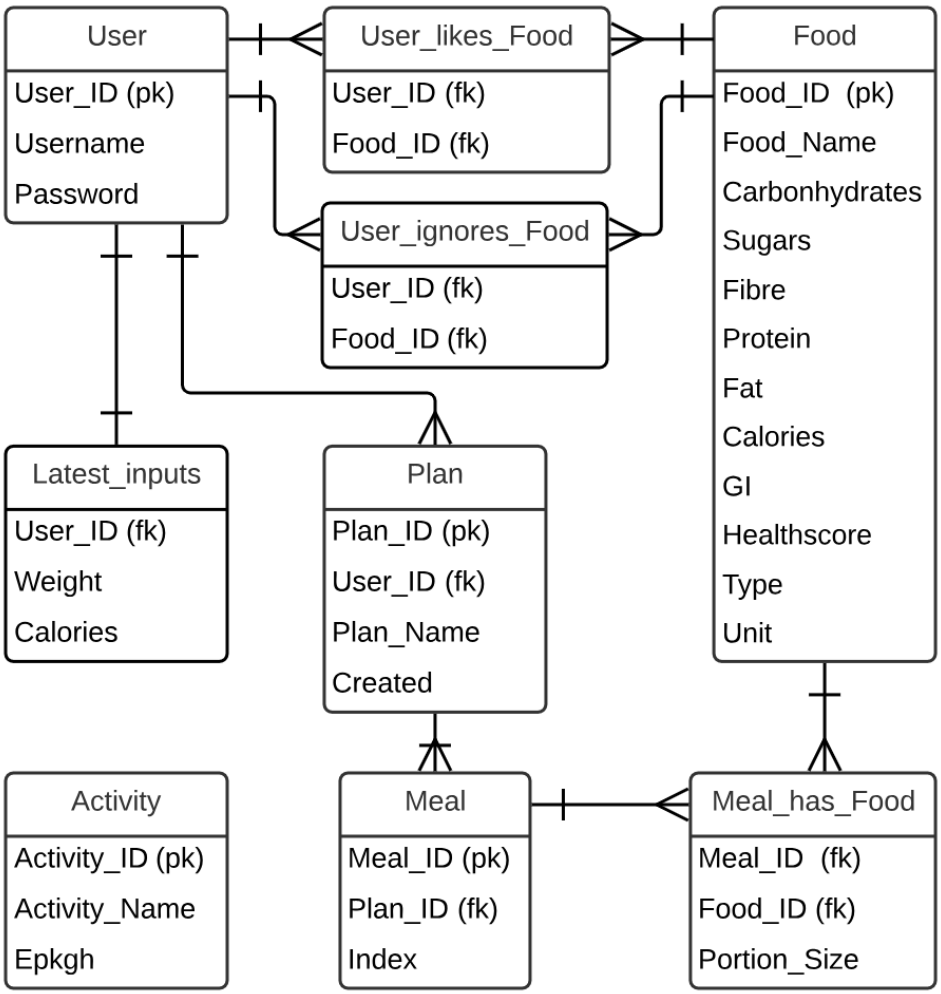
The database in 3rd Normal Form has to be in 2NF and has no transitive dependencies, i.e. each column has to be determined only by the primary key and nothing else. The set of tables above does not have any transitive dependencies, so it is already in the Third Normal Form. Normalizing up to a 3NF eliminates the chance of most common anomalies and is sufficient for most tasks in business or life. So the database does not require further normalization. However the database is not complete, because it doesn't stores all the necessary information.

Complete Database

The following things have to be added into the database: Food needs to have more attributes beyond just its name, such as nutritional values. Another transaction table (junction table) needed to represent many-to-many relationship between users and food, i.e. store user's favorite food. Food

has to have type. In addition, activities need their own table. Although they are not related to the food and meal plans, they have to be stored somewhere and the database is the best place, because it makes it easier to alter.

Entity attribute model



The EAM diagram above shows the complete version of the database. Junction tables have the combination of two foreign keys as a composite primary key of the tables. Most attributes have self-explanatory names, apart from “Epkgh”, which stands for: “energy per kilogram-hour”.

Entry data types

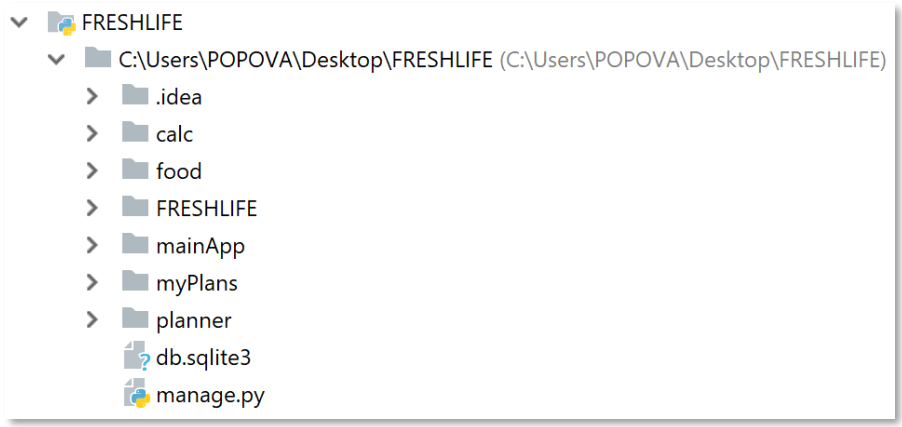
Entry	Proposed Type	Django.db.models type
User_ID	INT	IntegerField()
Username	Varchar(150)	CharField(maxlength=150)
Password	-	Django password Field
Plan_ID	INT	IntegerField()
Plan_Name	Varchar(45)	CharField(maxlength=45)
Meal_Index	time	IntegerField()
Portion_Size	INT	IntegerField()
Activity_ID	INT	IntegerField()
Activity_Name	Varchar(45)	CharField(maxlength=45)

Epkgh	INT	FloatField()
Food_ID	INT	IntegerField()
Food_Name	Varchar(45)	CharField(maxlength=45)
Carbohydrates	INT	IntegerField()
Sugars	INT	IntegerField()
Fibre	INT	IntegerField()
Protein	INT	IntegerField()
Fat	INT	IntegerField()
Energy	INT	IntegerField()
GI	INT	IntegerField()
Healthscore	INT	IntegerField()
Type	Varchar(45)	CharField(maxlength=45)
Unit	Varchar(45)	CharField(maxlength=45)
Weight	INT	FloatField()
Calories	INT	FloatField()
Created	DateTime	DateTimeField(default=datetime.now)

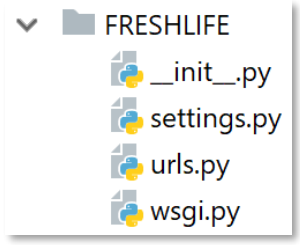
The table above determine the data type of each of the key and attribute in the database.

File structure and organization.

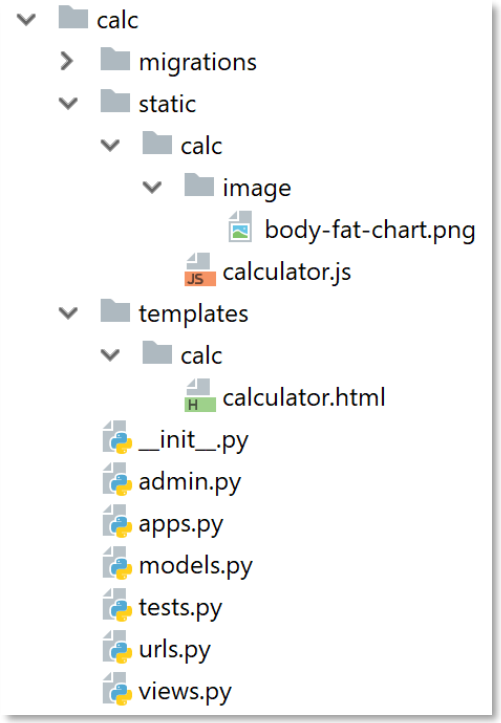
First level of the folders

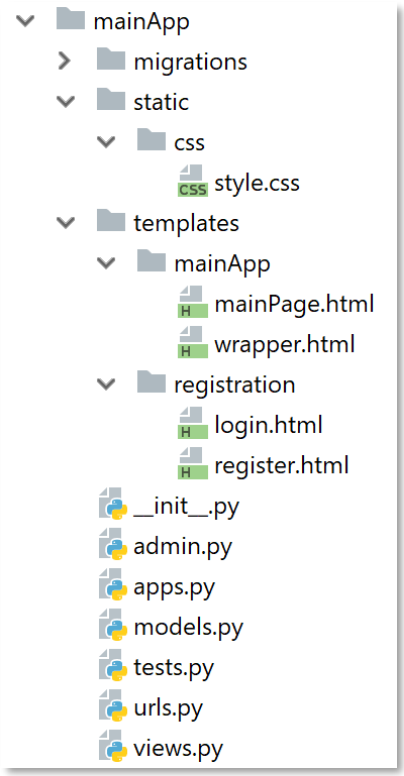


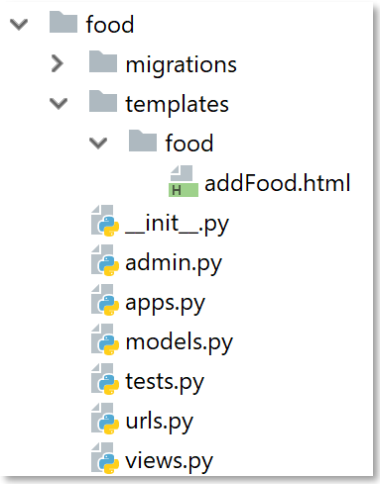
Main directory. FRESHLIFE

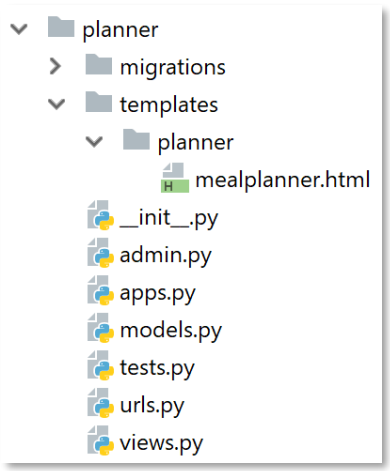


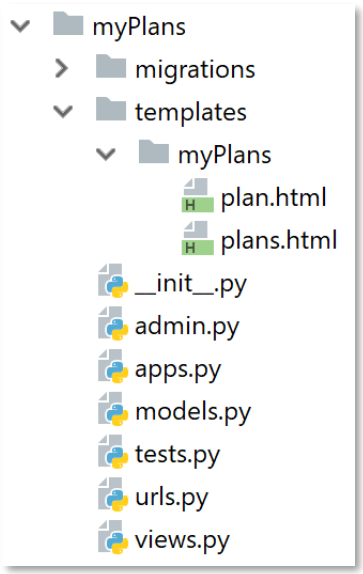
Apps.











App name	Purpose	Database models
mainApp	Main menu and authentication	-
calc	Calculator page with both simple and advance versions	Activities, Latest_inputs
food	Add Favourite food	Food
planner	Meal planner	Plan, Meal, Meal_has_food
myPlans	Display User's plans	-

Global variables of the Front-End

Variable	Type	Purpose	File
actIdCounter	integer	Activity Identifier Counter. Hold the numeric part of the id of the "div" of the HTML structure, created by the user. The structure represents user's activity. It is incremented every time user creates that "div", so every single one has unique id.	calc/static/calc/ calculator.js
EATperKG	float	Hold the value of the Energy per kilogram associated with user's training activity. It is dynamically changed by two different functions activated by two different buttons.	calc/static/calc/ calculator.js

Back-end part of the application doesn't have global variables, except ***settings.py***. There are many variables, which represent a certain settings, but they were created automatically by the Django. Another exception is app_name variable (type is string). It has a purpose to access functions using namespaces instead of hardcoding the path. Every app has its own app_name.
Example:
mainApp/urls.py

```
4   app_name = "mainApp"
5
6   urlpatterns = [
7       path('', views.mainPage, name='mainPage'),
8       path('register/', views.register, name='register'),
9       path('logout/', views.logout_user, name='logout_user'),
10  ]
```

Expressions in form 'app_name:name' (name of the view) like this:

```
return redirect('mainApp:mainPage')
```

Would activate "views.mainPage" in the mainApp without need to hardcode the path of the page.

Functions of the Server-side.

Function	Main purpose	Data type of return	Location
mainPage	Render the main menu page	render	mainApp/views.py
register	Render user creation form and redirect to the main menu page if it was valid when submitted	render, or redirect	mainApp/views.py
logout_user	Logout user and redirect to the login page	HttpResponseRedirect	mainApp/views.py
calculator	Render the calculator	render or redirect	calc/views.py
serving_master	Evaluate food servings through recursion	Tuple	planner/views.py
meal_planner	Render two forms: for creating meal plan and for saving it.	render	planner/views.py
user_plans	Display meal plans owned by the user	render	myPlans/views.py
user_plan	Render the view of one plan	render	myPlans/views.py
delete_plan	Delete plan owned by the user	redirect	myPlans/views.py
addfood	Render tools for finding food in the database: suggestions and search-bar	render	food/views.py
add_to_favourite	Change relationship between food and user	HttpResponseRedirect	food/views.py
remove_from_favourite	Change relationship between food and user	HttpResponseRedirect	food/views.py
add_to_ignore	Change relationship between food and user	HttpResponseRedirect	food/views.py
remove_from_ignore	Change relationship between food and user	HttpResponseRedirect	Food/views.py

In addition. All views functions, which render pages, in the beginning have an authentication check. If the user is “Anonymous-User”, function returns HttpResponceRedirect to the login view. To use the features of the application user has to log in. This is done to not give not Authenticated users a chance to use the content.

Queries.

The database engine used by the application is SQLite. However all the queries are performed with the aid of built in django methods. No raw SQL has been used.

As an example, the code below shows the query “Get all the plans of the current logged in user and sort it by the date (such that newest are first)”

```
8       plans = request.user.plan_set.all().order_by("-date")
```

With this or several other similar Django methods application performs the following queries from the database (Described on English):

1. Select all activities
2. Select Latest_inputs of the request user
3. Select all food, except the food ignored by the user
4. Select user’s favourite food, except the food ignored by the user
5. Working with samples described in 3 or 4:
    a. From the sample, select all the food with type “Main”
    b. From the sample, select all the food with type “Side”
    c. From the sample, select all the food with type “Vegetables”
    d. From the sample, select all the food with type “Other”
        - From each of these categories, select random Food
6. Select food with carbohydrates content more than or equal to 25
7. Select food with carbohydrates content less than 25
8. Select food with fat content more than or equal to 30
9. Select food with fat content less than 30
10. Select food with fat content more than or equal to 8
11. Select food with fat content less than 8
12. Select all the Plans, where Owner is the request user
13. Select the Plan with pk, equal to the integer in the URL
    - Select all the meals of that plan
        - For each meal: Select all Meal_has_Foods
14. Select food with pk, equals to the given integer

15. Select all Food

16. Select Food with pk, equal to the integer in the URL

17. Select Food with Names having the value of User input in the search textbox.

Hardware

Freshlife228 has technical requirements of a typical web-based application, i.e. they are very low. However, the least powerful hardware, on which the application was executed was Acer Aspire V5-571P-6642. It has processor Intel Core i5-3317U. Hence, the safest estimate of hardware choice, suitable for the application is Acer Aspire V5-571P-6642, or any other more powerful computer.

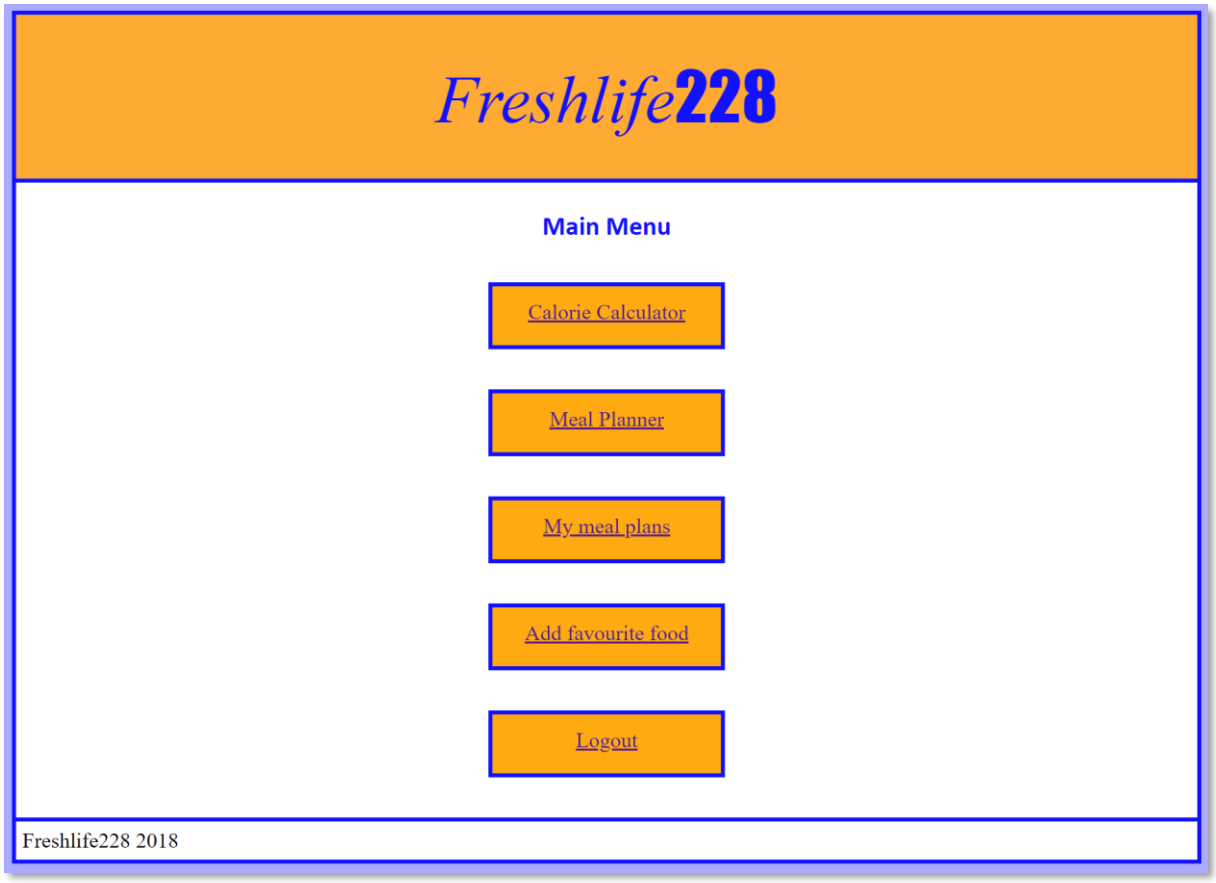
User Interface.

The following images are the screen captures of the GUI of the latest version of the application. All pages share the common structure. Blue background, main block in the middle with orange header and small footer saying “Freshlife228 2018”. The content block is between header and footer. Common color scheme is simple. It’s blue white and orange. Orange is used for clickable objects, like buttons. Header is also clickable, clicking on it always brings to the main menu page.

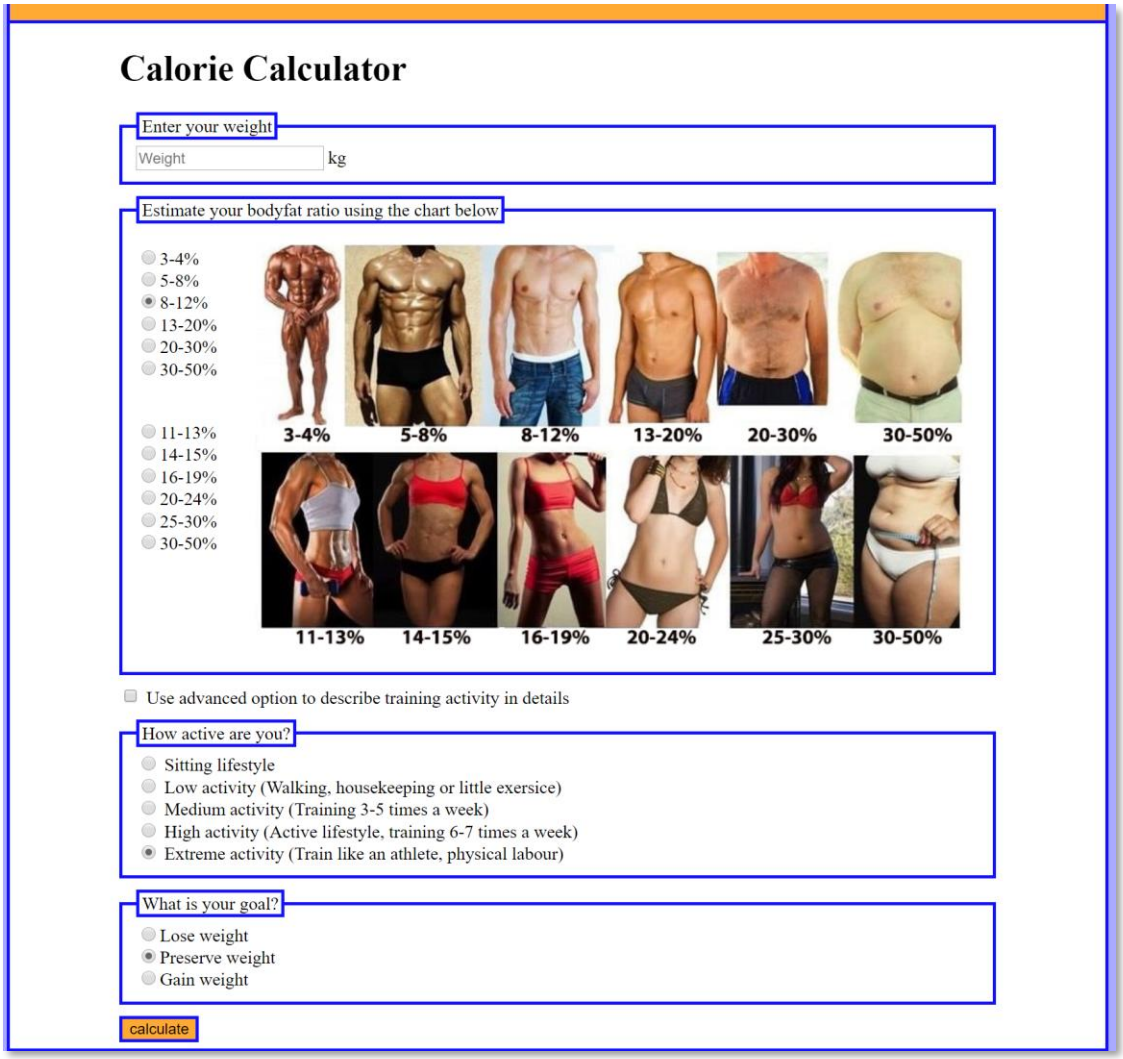
Login page

Register page

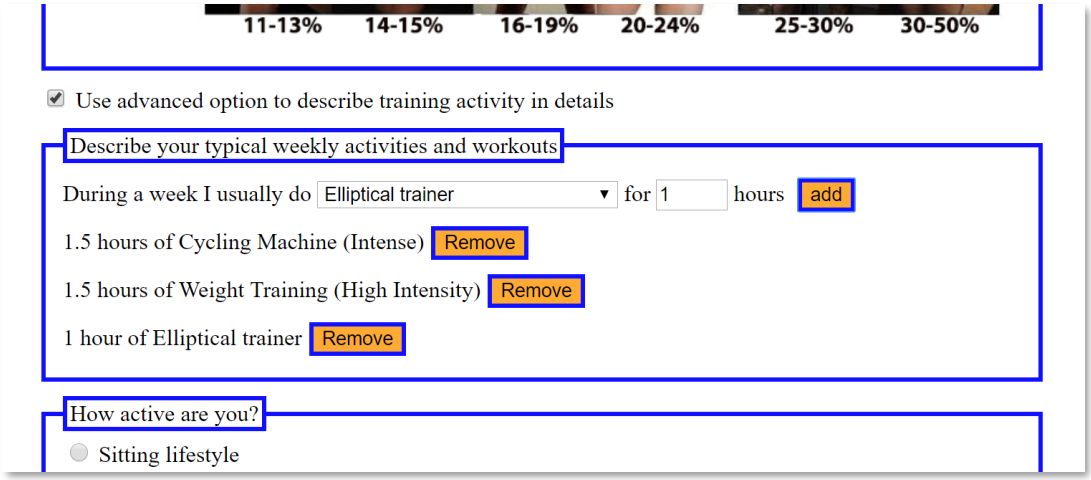
Main menu page



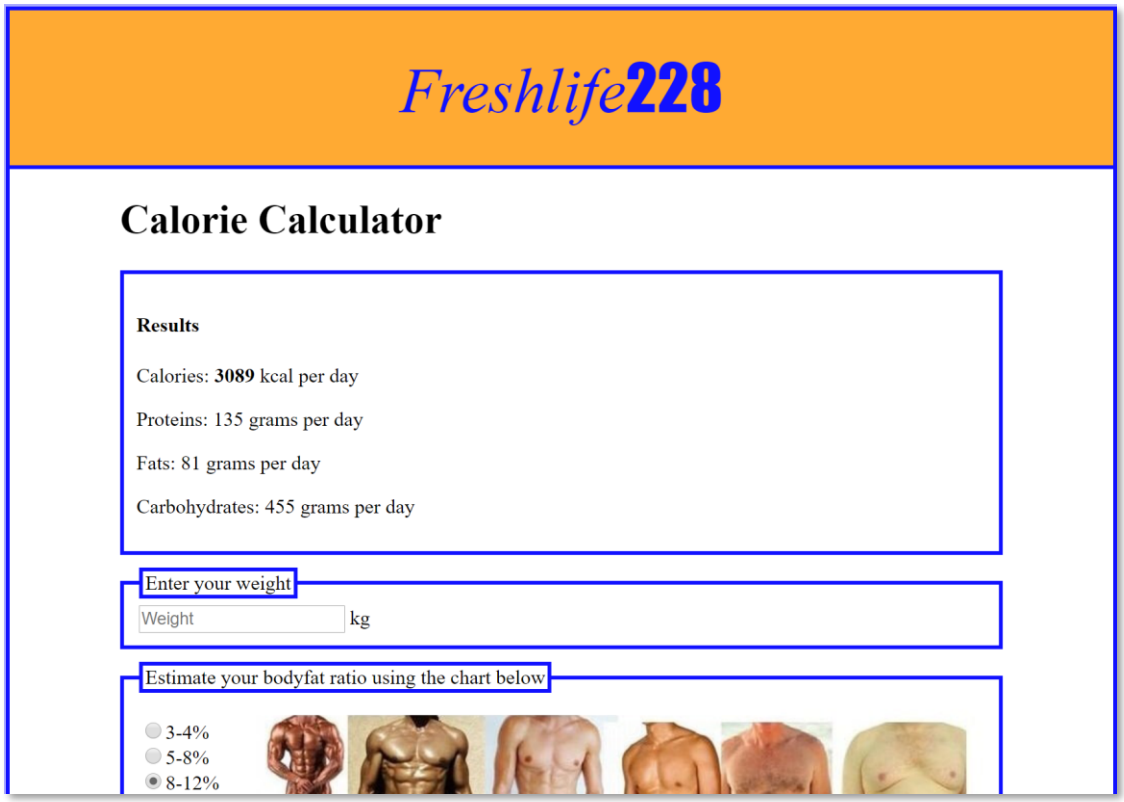
Calorie Calculator Page



If User decided to use advanced options



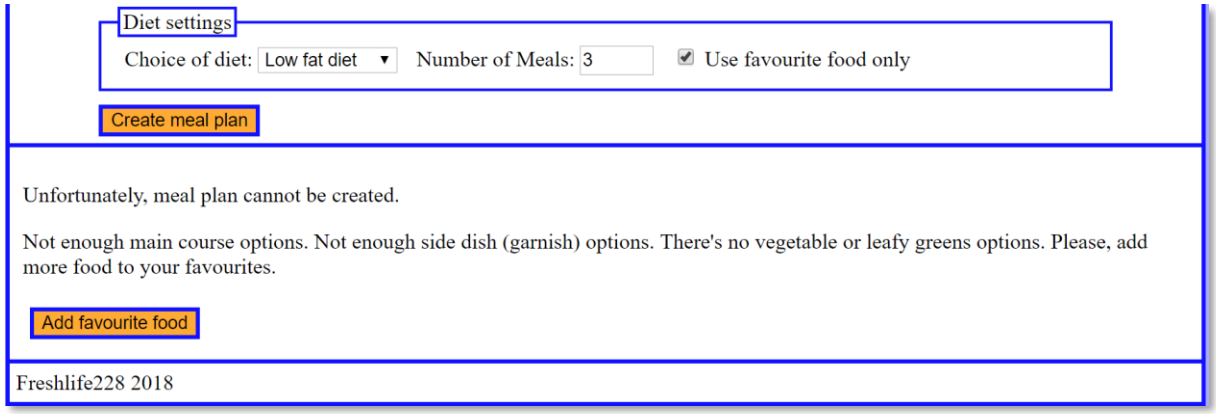
If after user pressed "Calculate"



Meal planner page

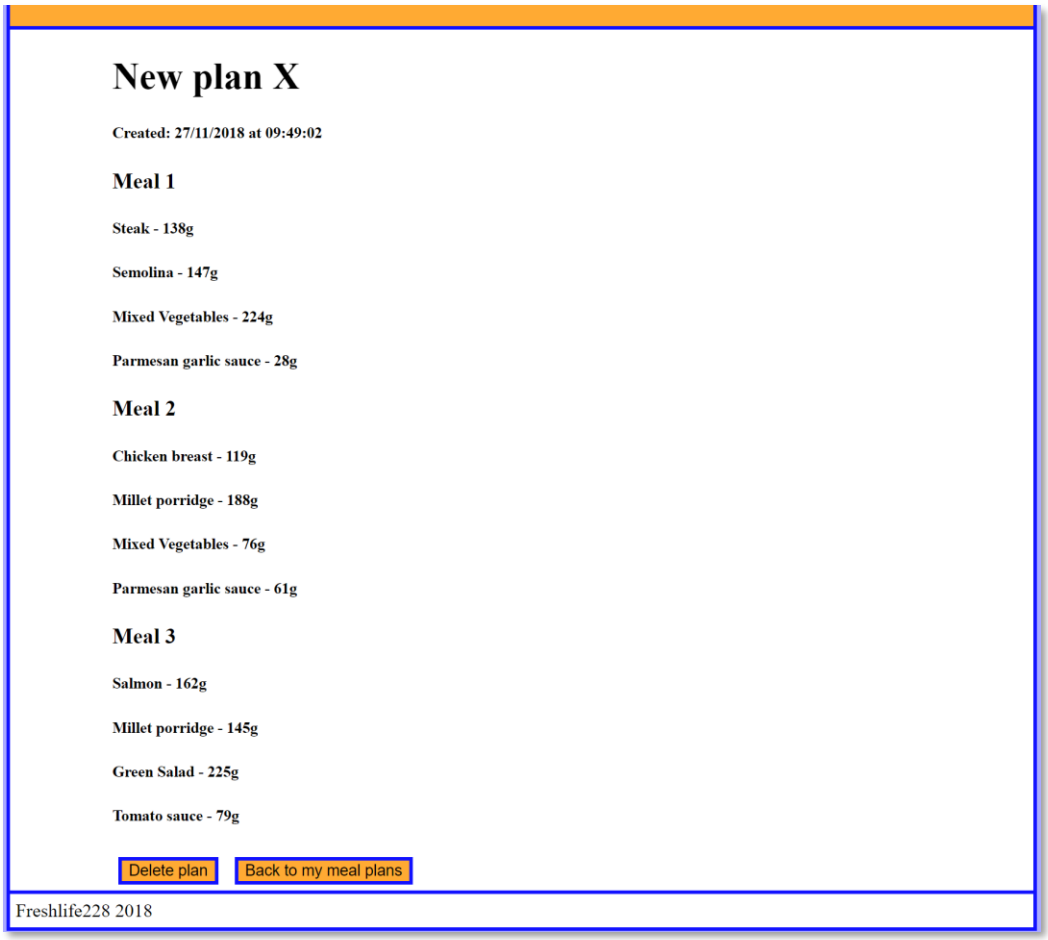
When user clicks "Create meal plan"

When chooses "favourite food only" and doesn't have enough options

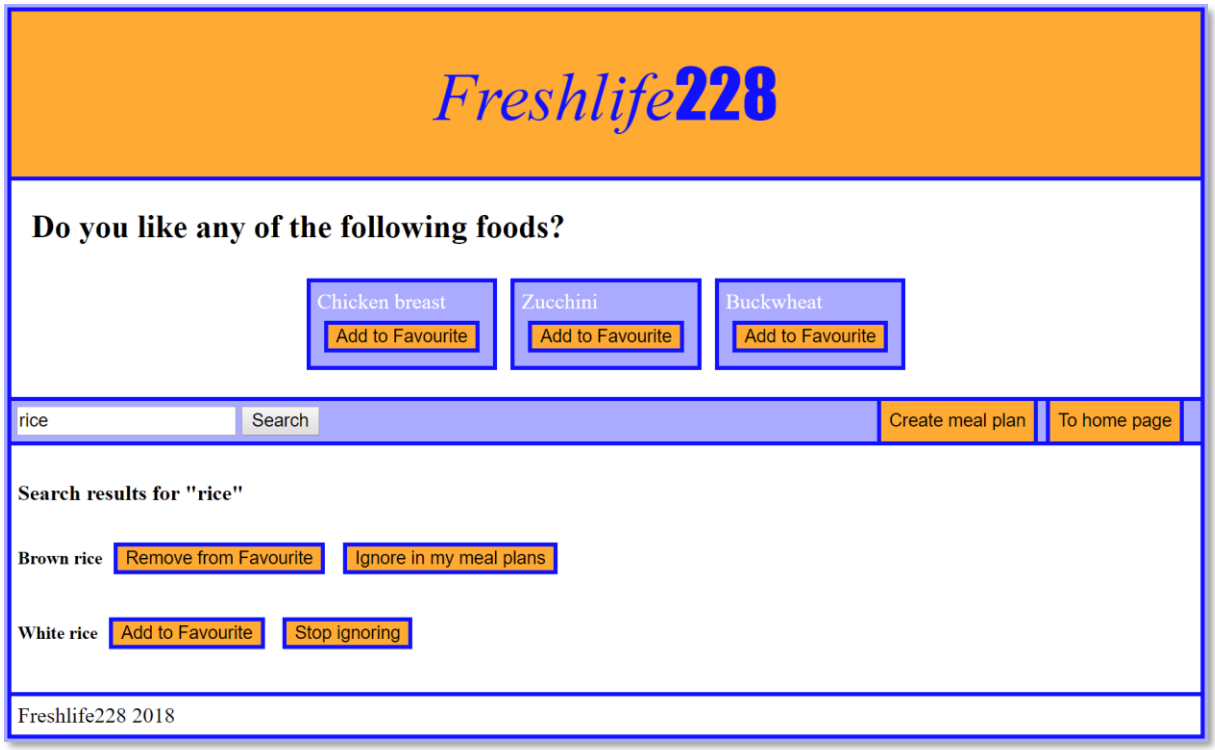


Meal plans viewer page

When user clicks "View plan"



Add Favourite food



In the image the current logged in user has “Brown rice” in his favourites and “White rice” in his Ignore list

TECHNICAL SOLLUTION

Completeness of the solution

A. Login system

Need	Description	Importance
Login system	Ability to login and logout (and register)	High

```
def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data['username']
            password = form.cleaned_data['password1']
            user = authenticate(username=username, password=password)
            login(request, user)
            return redirect('mainApp:mainPage')
    else:
        form = UserCreationForm()
    context = {'form': form}
    return render(request, 'registration/register.html', context)


def logout_user(request):
    logout(request)
    return HttpResponseRedirect('/accounts/login')
```

For logging in and logging out, application uses login() and logout() view from django.contrib.auth.views

B. Calorie Calculator and
C. Advanced option

Need	Description	Importance
Calories calculator	Ability to give good approximation of number of user's daily maintenance calories based on the user inputs.	High
Advanced options for calories calculator	Ability for user to choose between simple or advanced ways of calculating daily calories demand. Advanced option should allow user to fully describe his activity and get a more precise value	High

```python
def calculator(request, user_came_from_planner=0):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    activities_list = Activity.objects.all().order_by("name")  # taking activities from db
    calories = prots = fats = carbs = 0  # context vars
    if request.method == "POST":
        weight = float(request.POST.get('weight'))
        bodyfat = float(request.POST.get('bodyfat'))
        activity_choice = int(request.POST.get('activity'))
        goal = float(request.POST.get('goal'))
        advanced_checkbox = request.POST.get('adCalc')

        bmr = 370 + 21.6 * weight * (100 - bodyfat) / 100
        if advanced_checkbox:
            eat_per_kg = float(request.POST.get('sneaky'))  # hidden input in calculator.html
            calories += int(eat_per_kg * weight / 7)  # Exercise Energy Expenditure is added here
            common_difference = 0.05  # (1.15, 1.20. 1.25, ...)
        else:
            common_difference = 0.15  # (1.15, 1.30, 1.45, ...)
        # arithmetic sequence instead of "if" clause for determining activity coeficient
        activity_coef = 1.15 + common_difference * (activity_choice - 1)

        calories += int(bmr * activity_coef * goal)
        prots = int(weight * 1.5)  # Proteins
        fats = int(weight * 0.9)
        carbs = int((calories - prots * 4 - fats * 9) / 4)  # Carbohydrates
        # ----------------------------------------------------------------------
        # new code: updating User's Latest_inputs
        user = request.user
        try:  # plan a, if everything is OK
            if user.latest_inputs:
                user.latest_inputs.weight = weight
                user.latest_inputs.calories = calories
                user.save()
        except ObjectDoesNotExist:  # plan b, if User "lost" his latest_inputs
            Latest_inputs(user=user, weight=weight, calories=calories).save()
        # ----------------------------------------------------------------------
        if user_came_from_planner:  # also new: coming back to meal planner
            return redirect('planner:meal_planner')
    context = {
        'calories': calories,
        'prots': prots,
        'fats': fats,
        'carbs': carbs,
        'activities': activities_list,
    }
    return render(request, 'calc/calculator.html', context)
```

D. Daily Macronutrients

Need	Description	Importance
Daily Macronutrients	User should get the approximate value of amount of proteins, carbohydrates and fats his body needs daily	High

```python
            # Calculating daily macronutrients
            prots = diet[0] * weight
            fats = diet[1] * weight
            carbs = diet[2] * weight
            if not fats:  # one of the macros is always not fixed
                fats = (calories - prots * 4 - carbs * 4) / 9
            elif not carbs:
                carbs = (calories - prots * 4 - fats * 9) / 4
```

E. Favourite foods

Need	Description	Importance
Favourite foods	Ability to mark certain foods as favourite, so nutrition planner would consider user preferences	High

```
47  def add_to_favourite(request, pk):
48      food = Food.objects.get(pk=pk)
49      food.loved_by.add(request.user)
50      return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
51
52
53  def remove_from_favourite(request, pk):
54      user = request.user
55      food = Food.objects.get(pk=pk)
56      if user in food.loved_by.all():
57          food.loved_by.remove(user)
58      return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
```

F. Ignore list

Need	Description	Importance
Food ignore list	Ability to add food to the ignore list (Due to allergies or taste preferences). The food in that list will not be suggested in the meal plans created by this user.	High

```
61  def add_to_ignore(request, pk):
62      food = Food.objects.get(pk=pk)
63      food.ignored_by.add(request.user)
64      return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
65
66
67  def remove_from_ignore(request, pk):
68      food = Food.objects.get(pk=pk)
69      user = request.user
70      if user in food.ignored_by.all():
71          food.ignored_by.remove(user)
72      return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
```

G. Search and

H. Suggest

Need	Description	Importance
Search	Ability to search foods when user wants to mark them as favourite or ignored.	Medium
Suggest choice	In “add food”, the application suggest user few food options he might like to add.	Medium

```python
def addfood(request):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    # Suggestion block part
    suggest_list = []
    possible_suggestions = Food.objects.all()  # query
    already_favored_list = list(request.user.favourite_food.all())
    ignore_list = list(request.user.ignored_food.all())
    possible_suggestions = possible_suggestions.exclude(name__in=already_favored_list)
    possible_suggestions = possible_suggestions.exclude(name__in=ignore_list)

    length = len(possible_suggestions)  # length of narrowed list
    if length:
        randindex = randint(1, length)  # old randpk
        if length > 3:
            number_of_suggestions = 3
            r = randint(1, length - 1)  # just a random integer
        else:
            number_of_suggestions = length  # two or less items to add
            r = 1  # not that random anymore, due to empty randrange error when length < 1
        for i in range(number_of_suggestions):
            suggest_list.append(
                possible_suggestions[(randindex + r * i) % length]
            )
    # Search block part
    queryset_list = Food.objects.all().order_by("name")
    query = request.GET.get('query')
    if query:
        queryset_list = queryset_list.filter(name__icontains=query)
    # Final part of the view
    context = {
        "suggest_list": suggest_list,
        "query": query,
        "query_list": queryset_list
    }
    return render(request, 'food/addFood.html', context)
```

I. User-Friendliness

Need	Description	Importance
User-Friendly Interface	Application should be simple to work with and test users shouldn't have problems with it.	High

According to User Interface section, which shows the screen captures of the application GUI, User works with application which uses typical elements (Inputs, buttons, text boxes). All the buttons have noticeable colour and have text on, so It was attempted to make their purpose as clear as possible.

In addition, majority of the testers said that overall the application was easy and intuitive to use with some of them pointing on one not critical downside, which have been solved immediately (see User-Friendliness test report on page 93)

J. Nutrition planner

Need	Description	Importance
Nutrition Planner	Ability to create meal plans based on the values of user's energy and macros requirements and a number of meals he would like to have per day.	High

```
        #  if recursion failed or even hasn't been attempted
        if use_least_squares or recursion_has_failed:
            # print("Using Least Squares")
            if Other != 0:
                Other_p = Other.protein
                Other_f = Other.carbs
                Other_c = Other.fat
                Other_h = Other.healthscore
                Other_kcal = Other.calories
            else:
                Other = Other_p = Other_f = Other_c = Other_h = Other_kcal = 0

            A = array([  # content of nutrients in food
                [Main.protein,     Side.protein,     Veg.protein,     Other_p],
                [Main.carbs,       Side.carbs,       Veg.carbs,       Other_c],
                [Main.fat,         Side.fat,         Veg.fat,         Other_f],
                [Main.healthscore, Side.healthscore, Veg.healthscore, Other_h],
                [Main.calories,    Side.calories,    Veg.calories,    Other_kcal],
            ])
            b = array([  # requirements column vector
                prots,
                carbs,
                fats,
                healthscore,
                calories,
            ])
            servings = lsq_linear(A, b, bounds=(0.17, 4)).x.tolist()
            # print("r =", servings)

```

In addition: Technique-Recursion

K. Favourite food

Need	Description	Importance
Favourite food only	Ability to create meal plans containing only user's favourite food	Medium

- Checkbox is on the Meal Planner page

```
    if request.POST.get('fav_checkbox'):
        favourites = list(request.user.favourite_food.all())
        Main_choices = Main_choices.filter(name__in=favourites)
        Side_choices = Side_choices.filter(name__in=favourites)
        Veg_choices = Veg_choices.filter(name__in=favourites)
        Other_choices = Other_choices.filter(name__in=favourites)
        # checking if all requirements are satisfied
        if len(Main_choices) < 2:  # two or more Mains required
            reasons += "Not enough main course options. "
        if len(Side_choices) < 2:  # two or more Sides required
            reasons += "Not enough side dish (garnish) options. "
        if not Veg_choices:  # at least one source of vegetables required
            reasons += "There's no vegetable or leafy greens options. "
        # If any problems detected, user's favourites are insufficient
        if reasons:
            enough_favourites = False
```

L. Send to add food

Need	Description	Importance
Send user to the "Add Food"	If the user doesn't have favourite food or its number is insufficient, application should ask user to add some food.	Medium

- Client-side view here

```
69      {% if not enough_favourites %}
70          <p>Unfortunately, meal plan cannot be created.</p>
71          <p>{{ reasons }}Please, add more food to your favourites.</p>
72          <a href="{% url 'food:addfood' %}">
73              <button type="button">Add favourite food</button> <!--Button to the Add Food-->
74          </a>
75      {% elif plan %}
```

M. Save plan

Need	Description	Importance
Save meal plans to the database	Ability to save just created meal plan to the database	High

```
220     if request.method == 'POST' and 'save' in request.POST:  # "save" button was clicked
221
222         new_plan = Plan(  # creating plan with name provided by user
223             owner=request.user,
224             name=request.POST.get('plan_name'),  # user input
225         )
226         new_plan.save()
227         #  nested looping for, there meal number and food "indecies", m and f, start with 1, not 0
228         for m in range(1, int(request.POST.get('meal_num')) + 1):
229             meal = Meal(plan=new_plan, index=m) # creating m-th meal
230             meal.save()
231             for f in range(1, int(request.POST.get('three_or_four')) + 1):
232                 food_name = request.POST.get('food_{}{}'.format(m, f))  # finding
233                 serving = int(request.POST.get('serving_{}{}'.format(m, f)))
234                 Meal_has_Food(  # linking f-th food to m-th meal
235                     meal=meal,
236                     food=Food.objects.get(name=food_name),
237                     portion_size=serving,
238                     ).save()
```

N. Display meal plans

Need	Description	Importance
User meal plans	Ability to have all meal plans of the user displayed. User must have ability to view the details of the particular plan or delete plan.	High

```python
def user_plans(request):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    # user plans from latest to oldest
    plans = request.user.plan_set.all().order_by("-date")
    context = {
        'plans': plans,
    }
    return render(request, "myPlans/plans.html", context)


def user_plan(request, index):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    # taking one element from user plans (from latest to oldest)
    # 1 is subtracted from index, because it starts from 0, while
    # forloop counter starts from 1
    plan = request.user.plan_set.all().order_by("-date")[index - 1]
    context = {
        "plan": plan,
        "meals": plan.meal_set.all().order_by("index")
    }
    return render(request, "myPlans/plan.html", context)
```

```
{% block content %}
<div id="plans">
    <h1>My meal plans</h1>
    {% for plan in plans %}
        <div class="plan_in_list">
            <a class="header3" href="{% url 'myPlans:plan' index=forloop.counter %}">
                <h3>{{plan.name}} </h3>
            </a>
            <h5>Created: {{plan.date|date:"d/m/Y"}} at {{plan.date|date:"h:i:s"}}</h5>
            <a href="{% url 'myPlans:plan' index=forloop.counter %}">
                <button>View plan</button> <!--View button-->
            </a>
            <a href="{% url 'myPlans:delete' pk=plan.pk %}">
                <button>Delete</button>  <!--Delete button-->
            </a>
        </div>
    {% endfor %}
    <a href="{% url 'mainApp:mainPage' %}">
        <button>Back</button>
    </a>
</div>
{% endblock %}
```

```
{% block content %}
<div id="plan">
    <h1>{{plan.name}}</h1>
    <h5>Created: {{plan.date|date:"d/m/Y"}} at {{plan.date|date:"h:i:s"}}</h5>
    {% for meal in meals %}
        <h3>Meal {{meal.index}}</h3>
        {% for ingridient in meal.meal_has_food_set.all %}
            <h5>{{ingridient.food.name}} - {{ingridient.portion_size}}{{ingridient.food.unit}} </h5>
        {% endfor %}
    {% endfor %}
    <a href="{% url 'myPlans:delete' pk=plan.pk %}">
        <button>Delete plan</button>  <!--Delete button-->
    </a>
    <a href="{% url 'myPlans:plans' %}">
        <button>Back to my meal plans</button>
    </a>
</div>
{% endblock %}
```

Cntrl+click on HTML code to see the client-side view

O. Different styles of diet

Need	Description	Importance
Different styles of diet	Make dietary plans in different styles of diet, for example ketogenic, paleo, low-carb.	Low

```
<div class="setting"><!--Types of diet-->
    Choice of diet:
    <select name="diet"><!--Choice between three diets-->
        <option value="[1.5, 0.9, 0]">Low fat diet</option>
        <option value="[2, 0, 1.3]">Low carb diet</option>
        <option value="[2.0, 1.2, 0]">High protein diet</option>
    </select>
</div>
```

P. Data security

Need	Description	Importance
Data security	User passwords should be stored as hashed values. Allowing users create only passwords with sufficient level of complexity	High

As a result. All of the core objectives have been satisfied in sufficient amount

Techniques Used Reference tables

AQA Group A technical skills

Index	Technique	Purpose / description	Files and their page reference
1	Complex data model of the database	5 entities, 3 many-to-many fields of which 1 uses intermediate model, 1 One-to-one field	Entire files: food/models.py (page 141), calc/models.py (page 124), planner/models.py (page 128)
2	List operations	Get length of a list, Add element, Check membership, etc	Implementation is spread across many files
3	Recursion	Evaluate the values of the servings of food in a meal	planner/views.py (page 129) lines 11-71 and lines 13-167
4	Complex mathematical/ scientific: Matrix operations (with optimization elements)	Evaluate the best-fit values of the servings of food in a meal	planner/views.py (page 129) lines 183-197
5	OOP. Inheritance	Creating own classes based on Django parent class	Every class definition in this files: food/models.py (page 141), calc/models.py (page 124), planner/models.py (page 128)
6	OOP. Association	Make Latest_inputs, One-to-one field of the User model, created and altered on cascade with corresponding User model. *(In addition all models are associated via foreign keys or many-to-many fields)*	calc/models.py (page 124), *food/models.py (page 141),* *planner/models.py (page 128)*

AQA Group B technical skills

Index	Technique	Purpose	Files and their page reference
7	Dictionaries	Send context variables from Server to HTML, so Jinja2 is able to parse them	Every *.../views.py* file (pages 129, 125, 141, 119, 137)
8	Multi-dimensional arrays (Python lists)	Organise data in special way, represent mathematical matrix	planner/views.py (page 129)
9	OOP. Six defined classes.	Representation of the database entities or transaction tables as classes in Python	Entire files: food/models.py (page 141), calc/models.py (page 124), planner/models.py (page 128)
10	Generation of objects based on simple OOP model	Allow user to create/alter User, Plan, Meal, Meal_has_Food and Latest_inputs instances.	calc/views.py (page 125) lines 40-41 mainApp/views.py (page 119) lines 216-232
11	Simple user defined algorithms. Search engine.	Allow user search food he wants to mark as favourite or ignored	food/views.py (page 141) lines 31-35
12	Simple user defined algorithms (eg a range of mathematical / statistical calculations), i.e. Calorie Calculator	Calculate User's Daily Caloric intake	calc/views.py (page 125) (almost entire file)
13	Server-side scripting using request object	-	All views.py files (pages 129, 125, 141, 119 and 137)

AQA Group C technical skills

Index	Technique	Purpose	Files and their page reference
14	Simple mathematical: Arithmetic sequence	Evaluate coefficient in the calorie calculator algorithm	calc/views.py (page 125) lines 22-26
15	Simple mathematical: Finding average	Find average Exercise Associated Thermogenesis	calc/views.py (page 125) line 21

Other technical skills

Index	Technique	Purpose	Files and their page reference
16	Exception handling	Solve ObjectDoesNotExist and RecursionError types of errors	calc/views.py (page 125) lines 35-40 planner/views.py (page 129) lines 162-169
17	Java Script algorithms interacting with HTML DOM	Allow user dynamically create and delete HTML structures, representing his weekly training activity.	Entire file: *calc/static/calc/calculator.js* (page 120)

Techniques Used. Descriptions, Justifications

1. Complex Database Model

The code below shows defines the tables and relationships between the entities.

calc/models.py

```
class Activity(models.Model):
    name = models.CharField(max_length=45)
    epkgh = models.FloatField()

    def __str__(self):
        return self.name


class Latest_inputs(models.Model):  # additional information related to user
    user = models.OneToOneField(User, on_delete=models.CASCADE, primary_key=True)
    weight = models.FloatField(null=True)
    calories = models.FloatField(null=True)

    @receiver(post_save, sender=User)
    def create_latest_inputs(sender, instance, created, **kwargs):
        if created:
            Latest_inputs.objects.create(user=instance)

    @receiver(post_save, sender=User)
    def save_latest_inputs(sender, instance, **kwargs):
        instance.latest_inputs.save()

    def __str__(self):
        return "{}'s inputs".format(self.user.username)
```

Definition of two classes: Activity and Latest_Inputs. Activity is a single table, isolated from other Database. Lates_inputs is linked to the User model (django.contrib.auth.models.User) and adds extra fields to the user.

food/models.py

```
class Food(models.Model):
    name = models.CharField(max_length=45)
    carbs = models.IntegerField()
    sugars = models.IntegerField()  # not used yet
    fibre = models.IntegerField()  # not used yet
    protein = models.IntegerField()
    fat = models.IntegerField()
    calories = models.IntegerField()
    gi = models.IntegerField()  # not used yet
    healthscore = models.IntegerField()
    type = models.CharField(max_length=45)
    unit = models.CharField(max_length=45, default="g")

    loved_by = models.ManyToManyField(User,
                                      blank=True,
                                      related_name="favourite_food")
    ignored_by = models.ManyToManyField(User,
                                        blank=True,
                                        related_name="ignored_food")

    def __str__(self):
        return self.name

    def __repr__(self):
        return self.name
```

Definition of Food model. It has many nutritional values fields, name, type and unit. Types are: Main, Side, Vegetables and Other. In addition it has two many-to-many fields related with User. (lines 18 and 21)

Note that not all of them have been used in the current application. 3 fields: sugars, fibre and gi has not been used in this version of application and can be ignored. Now they are nothing, but plans for the future.

planner/models.py

```
7   class Plan(models.Model):
8       name = models.CharField(max_length=45)
9       owner = models.ForeignKey(User, on_delete=models.CASCADE)
10      date = models.DateTimeField(default=datetime.now)  # when created
11
12      def __str__(self):
13          return self.name
14
15
16  class Meal(models.Model):
17      index = models.IntegerField(null=True)  # order number in the plan
18      plan = models.ForeignKey(Plan, on_delete=models.CASCADE)
19      food = models.ManyToManyField(Food, through='Meal_has_food')
20
21      def __str__(self):
22          return "{}--{}".format(self.plan, self.index)
23
24
25  class Meal_has_Food(models.Model):  # intermediary model (junction table)
26      meal = models.ForeignKey(Meal, on_delete=models.CASCADE)
27      food = models.ForeignKey(Food, on_delete=models.CASCADE)
28      portion_size = models.IntegerField()
29
30      def __str__(self):  # str help work with normalized data
31          if self.food.unit == "null":
32              unit = ""
33          else:
34              unit = self.food.unit
35          return "{}--{} {}{}".format(self.meal, self.food,
36                                      self.portion_size, unit)
37          # example str: "TESTPLAN--3--Tomato sauce 30g"
```

Definition of Plan and Meal. Meal is owned by Plan which is owned by User via Foreignkey. In Line 19 Meal has Many-to-Many field with Food, but since each instance of the relationship has additional field, Intermediary model *Meal_has_Food* is used. All objects have __str__() methods to make them look readable in Django Administration Board.

- However, defining the class itself doesn't do anything to the database. In order to actually apply the create things or apply changes migrations have to be made. Programmer only types in two special command in the command prompt (Windows Powershell):

```
PS C:\Users\POPOVA\desktop\FRESHLIFE> python manage.py makemigrations
No changes detected
PS C:\Users\POPOVA\desktop\FRESHLIFE> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, calc, contenttypes, food, planner, sessions
Running migrations:
  No migrations to apply.
```

In this example nothing have been changed.

- Migrations themselves are created automatically by manage.py after the first command and are applied after the second.

Example of auto-created migrations:

```python
# Generated by Django 2.1.1 on 2018-11-14 07:27

from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Activity',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=45)),
                ('epkgh', models.FloatField()),
            ],
        ),
    ]

```

But even at this level SQLite statements cannot be found. The evidence of the SQL statements is hidden deeply in the Django package. For example:

C:\Users\POPOVA\AppData\Local\Programs\Python\Python36-32\Lib\site-packages\django-2.1.1-py3.6.egg\django\db\backends\base\creation.py

```python
    def _execute_create_test_db(self, cursor, parameters, keepdb=False):
        cursor.execute('CREATE DATABASE %(dbname)s %(suffix)s' % parameters)
```

2. List operations

Converting iterable object Queryset into list:

```python
        already_favored_list = list(request.user.favourite_food.all())
```

Adding element of the list and also retrieving an element from a list at a certain index:

```python
        for i in range(number_of_suggestions):
            suggest_list.append(
                possible_suggestions[(randindex + r * i) % length]
            )
```

Converting string representation of a list to the python list:

```python
        diet = ast.literal_eval(request.POST.get('diet'))  # string representation of a list
```

Example: "[1.5, 1, 0]" → [1.5, 1, 0]

Retrieving the values of list length:

```
18      length = len(possible_suggestions)  # length of narrowed list
```

Converting 2-D and 1-D lists to numpy.array:

```
193             A = array([  # content of nutrients in food
194                 [Main.protein,     Side.protein,     Veg.protein,     Other_p],
195                 [Main.carbs,       Side.carbs,       Veg.carbs,       Other_c],
196                 [Main.fat,         Side.fat,         Veg.fat,         Other_f],
197                 [Main.healthscore, Side.healthscore, Veg.healthscore, Other_h],
198                 [Main.calories,    Side.calories,    Veg.calories,    Other_kcal],
199             ])
200             b = array([  # requirements column vector
201                 prots,
202                 carbs,
203                 fats,
204                 healthscore,
205                 calories,
206             ])
```

3. Recursion

```
11  def serving_master(
12          prots, fats, carbs,
13          calories, healthscore,
14          Main, Side, Veg, Other=None
15  ):
16      # Measures to make nutrients non-negative
17      if prots < 0: prots = 0
18      if fats < 0: fats = 0
19      if carbs < 0: carbs = 0
20      if calories < 0: calories = 0
21      if healthscore < 0: healthscore = 0
22
23      # Making sure certain percentage of nutritional value comes from
24      # the right source, and calibrating nutrients after each assignment
25      Veg_s = 0.6 * healthscore / Veg.healthscore
26      prots -= Veg.protein * Veg_s
27      carbs -= Veg.carbs * Veg_s
28      fats -= Veg.fat * Veg_s
29      healthscore -= Veg.healthscore * Veg_s
30
31      Main_s = 0.6 * prots / Main.protein + 0.05 * fats / Main.fat
32      prots -= Main.protein * Main_s
33      carbs -= Main.carbs * Main_s
34      fats -= Main.fat * Main_s
35      healthscore -= Main.healthscore * Main_s
36
37      Side_s = 0.7 * carbs / Side.carbs
38      prots -= Side.protein * Side_s
39      carbs -= Side.carbs * Side_s
40      fats -= Side.fat * Side_s
41      healthscore -= Side.healthscore * Side_s
42
43      # calibrating calories
44      calories -= Main.calories * Main_s +\
45          Side.calories * Side_s + Veg.calories * Veg_s
```

```
46
47      if Other:
48          Other_s = 0.05 * calories / Other.calories
49          prots -= Other.protein * Other_s
50          carbs -= Other.carbs * Other_s
51          fats -= Other.fat * Other_s
52          healthscore -= Other.healthscore * Other_s
53          calories -= Other.calories * Other_s
54      else:
55          Other_s = 0
56
57      # print(prots,"|", fats, "|", carbs, "|", calories, "|", healthscore)
58      if calories > 200 or prots > 10 or carbs > 10 or fats > 25:
59          # if actual macronutrients values are not close enough
60          # to the expected values, another iteration is executed
61          print("ONE MORE ITERATION!")
62          extra_servings = serving_master(
63              prots, fats, carbs,
64              calories, healthscore,
65              Main, Side, Veg, Other,
66          )
67          Main_s += extra_servings[0]
68          Side_s += extra_servings[1]
69          Veg_s += extra_servings[2]
70          Other_s += extra_servings[3]
71      return Main_s, Side_s, Veg_s, Other_s  # servings of each category
```

The function above is one of the methods of evaluating the values of the servings of each food type in the meal. It receives the food choices and macronutrient requirements, makes sure that a part of the certain macronutrient comes from a food which normally is rich in that nutrient (From example protein from Main, which is a meat). At the end of the iteration only a part of the requirements is satisfied. Values of amounts of undefined(unsatisfied) nutrients is sent to another iteration and so on until the servings converge to a point close enough to the expected, (max possible calories margin = -200, carbs and prots = -10, fats = -30, line 58).

Note that this method was not proposed in the Pseudocode The reason why there is an extra serving evaluating algorithm is that some algorithms are more suitable for particular diet. For example serving_master() is good at high carb diets, while it is not very successful for non-standart diets. In addition.

4. **Matrix operation (with elements of optimization)**

```
6   from numpy import array
7   from scipy.optimize import lsq_linear
```

```
171             #  if recursion failed or even hasn't been attempted
172             if use_least_squares or recursion_has_failed:
173                 # print("Using Least Squares")
174                 if Other != 0:
175                     Other_p = Other.protein
176                     Other_f = Other.carbs
177                     Other_c = Other.fat
178                     Other_h = Other.healthscore
179                     Other_kcal = Other.calories
180                 else:
181                     Other = Other_p = Other_f = Other_c = Other_h = Other_kcal = 0
182 
183                 A = array([  # content of nutrients in food
184                     [Main.protein,     Side.protein,     Veg.protein,     Other_p],
185                     [Main.carbs,       Side.carbs,       Veg.carbs,       Other_c],
186                     [Main.fat,         Side.fat,         Veg.fat,         Other_f],
187                     [Main.healthscore, Side.healthscore, Veg.healthscore, Other_h],
188                     [Main.calories,    Side.calories,    Veg.calories,    Other_kcal],
189                 ])
190                 b = array([  # requirements column vector
191                     prots,
192                     carbs,
193                     fats,
194                     healthscore,
195                     calories,
196                 ])
197                 servings = lsq_linear(A, b, bounds=(0.17, 4)).x.tolist()
198                 # print("r =", servings)
199 
```

The code above is a part of a big meal planner algorithm. It is a python version of the last proposed Pseudo code algorithms. It defines matrix A and column vector b, which describe equation Ar = b. But just solving simultaneous equations doesn't work, because matrix can be singular. In this case it is not even square matrix. But it is possible to analytically find the point which is as close to the sollution as possible. The method is called Least squares approximation. Lsq_linear() is a version of this algorithm, which allows to constraint the values of the solution. So in line 197 of the picture above the bounds are 0.17 and 4, which means that servings should be between 17 and 400 grams (17 grams is the average mass of a table spoon, which is the smallest serving size which makes sense)

5. OOP. Inheritance

All models definitions have already been shown in Technique 1 – Complex Database Model. All of them are subclasses of the class Model (from Django.db.models library). They inherit auto-incrementing integer ID, methods which allow to query them using Python and many other useful abilities.

6. OOP. Association

Coming back to the planner/models.py

```
class Latest_inputs(models.Model):  # additional information related to user
    user = models.OneToOneField(User, on_delete=models.CASCADE, primary_key=True)
    weight = models.FloatField(null=True)
    calories = models.FloatField(null=True)

    @receiver(post_save, sender=User)
    def create_latest_inputs(sender, instance, created, **kwargs):
        if created:
            Latest_inputs.objects.create(user=instance)

    @receiver(post_save, sender=User)
    def save_latest_inputs(sender, instance, **kwargs):
        instance.latest_inputs.save()

    def __str__(self):
        return "{}'s inputs".format(self.user.username)
```

Latest_inputs is associated with User model via create_latest_inputs() and save_latest_inputs(). This methods create/save Latest_inputs instance, than corresponding User instance has been created/saved.

In addition, all models (except Activity) are interlinked with either ManyToManyField() or ForeignKey() and can access each other.

```
            user.latest_inputs.weight = weight
            user.latest_inputs.calories = calories
            user.save()
```

In the example above, weight and calories are retrieved from Latest inputs, which are accessed through user (instance of User)

7. Dictionaries

Dictionaries are widely user in all of the views.py files. Here's a typical example:

calc/views.py

```
    context = {
        'calories': calories,
        'prots': prots,
        'fats': fats,
        'carbs': carbs,
        'activities': activities_list,
    }
    return render(request, 'calc/calculator.html', context)
```

Method based view sends dictionary "context" to template calculator.html inside render()

Meanwhile in calc/templates/calc.calculator.html context variables are parsed inside double curly parentheses "{{ }}".

```
    {% if calories %}
        <div id="results">
            <h4>Results</h4>
            <p>Calories: <strong>{{ calories }}</strong> kcal per day</p>
            <p>Proteins: {{ prots }} grams per day</p>
            <p>Fats: {{ fats }} grams per day</p>
            <p>Carbohydrates: {{ carbs }} grams per day</p>
        </div>
    {% endif %}
```

So they can be displayed on the web page with the aid of template language Jinja and HTML. Without the dictionary, Jijna2 is unable to parse “calories”, “prots” and other things.

8. Multi-dimensional Python lists

Sending 2-D List as argument of the array().

```
A = array([  # content of nutrients in food
    [Main.protein,     Side.protein,     Veg.protein,     Other_p],
    [Main.carbs,       Side.carbs,       Veg.carbs,       Other_c],
    [Main.fat,         Side.fat,         Veg.fat,         Other_f],
    [Main.healthscore, Side.healthscore, Veg.healthscore, Other_h],
    [Main.calories,    Side.calories,    Veg.calories,    Other_kcal],
])
```

2-D List, representing Meal.

```
meal = [
    [Main, Main_s, Main.unit],
    [Side, Side_s, Side.unit],
    [Veg, Veg_s, Veg.unit]
]
```

When the meal is appended to the Plan list, 3-D list is formed. Plan list can contain as many meals as user wants.

Iterating through 3-D List `plan` and displaying its elements with Jinja

```
            {% for meal in plan %} <!-- display the meals in plan-->
                <div class="meal_div">
                    <h3>Meal {{ forloop.counter }}</h3>
                    {% with outer=forloop %}
                        {% for food in meal %} <!-- display food in the meal -->
                            <h4>{{ food.0 }} - {{ food.1 }} {{ food.2 }}</h4>
                            <input class="stealth"
                                   name="food_{{outer.counter}}{{forloop.counter}}"
                                   type="text"
                                   value="{{ food.0 }}"><!--hidden input of the food name-->
                            <input class="stealth"
                                   name="serving_{{outer.counter}}{{forloop.counter}}"
                                   type="number"
                                   value="{{ food.1 }}"><!--hidden input of the food serving size-->
                        {% endfor %}
                    {% endwith %}
                </div>
            {% endfor %}
```

9. Use of OOP

In technique with index 1 (Complex Database Model) all the code with class definition has been displayed.
Fields of the objects and methods of their parent class Model are widely used in the application.

Example of working with objects:

Working with User and Food which are related together:

```
65  def remove_from_ignore(request, pk):
66      food = Food.objects.get(pk=pk)
67      user = request.user
68      if user in food.ignored_by.all():
69          food.ignored_by.remove(user)
70      return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
```

Working with User and his Latest_inputs:

```
36              if user.latest_inputs:
37                  user.latest_inputs.weight = weight
38                  user.latest_inputs.calories = calories
39                  user.save()
```

Working with Querysets of food, i.e. iterable object with multiple objects in it (objects must be subclasses of Model):

```
97          # making a queryset of the acceptable food choices and making sure it's sufficient
98          ignored = list(request.user.ignored_food.all())  # ignored food
99          choices = Food.objects.all().exclude(name__in=ignored)
100
101         Main_choices = choices.filter(type="Main")  # splitting shoices in categories
102         Side_choices = choices.filter(type="Side")
103         Veg_choices = choices.filter(type="Vegetables")
104         Other_choices = choices.filter(type="Other")
```

10. Generating objects

Since all the objects represent data in the database, Object is generated every time a data is stored, to it. User is able to create objects by filling forms or by pressing some buttons.

Examples:

Creating Plan, which belongs to the user, Meals in that plan and every meal to the food in that meal.

```
        new_plan = Plan(  # creating plan with name provided by user
            owner=request.user,
            name=request.POST.get('plan_name'),  # user input
        )
        new_plan.save()
        #  nested looping for, there meal number and food "indecies", m and f, start with 1, not 0
        for m in range(1, int(request.POST.get('meal_num')) + 1):
            meal = Meal(plan=new_plan, index=m)  # creating m-th meal
            meal.save()
            for f in range(1, int(request.POST.get('three_or_four')) + 1):
                food_name = request.POST.get('food_{}{}'.format(m, f))  # finding
                serving = int(request.POST.get('serving_{}{}'.format(m, f)))
                Meal_has_Food(  # linking f-th food to m-th meal
                    meal=meal,
                    food=Food.objects.get(name=food_name),
                    portion_size=serving,
                ).save()
```

Creating an Instance of User with User model form…

```
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
```

… which is filled by the user when he registers.

```
<form method="post" action="{% url 'mainApp:register' %}">
    {% csrf_token %}
    {% if form.errors %}
    <p>There are errors in the form!</p>
    {% endif %}
    <!--{{ form.as_p }}-->
    <p><label>Username:</label>  {{form.username}}</p>
    <p><label>Password:</label>  {{form.password1}}</p>
    <p><label>Confirm password:</label>  {{form.password2}}</p>

    <span class="helptext">
        Username must contain less than 150 characters. <br/>
        Acceptable characters are: letters, digits and @/./+/-/_ only.
    </span>
    <ul>
        <li>Your password can't be too similar to your other personal information.</li>
        <li>Your password must contain at least 8 characters.</li>
        <li>Your password can't be a commonly used password.</li>
        <li>Your password can't be entirely numeric.</li>
    </ul>
    <button type="submit">Register</button>
</form>
```

11. Search engine

User enters text in special textbox and presses submit-button
HTML

```
    <form style="width:50%;float:left; margin: 4px;" method="GET" action="">
        <input type="text" name="query"
               placeholder="Search food"
               value="{{ request.GET.query }}"/>
        <input type="submit" value="Search"/>
    </form>
```

Server finds all objects, where name contains the text in the textbox

Python

```
    # Search block part
    queryset_list = Food.objects.all().order_by("name")
    query = request.GET.get('query')
    if query:
        queryset_list = queryset_list.filter(name__icontains=query)
```

queryset_list is sent to the HTML in a dictionary as 'query_list'...

HTML

```
{% for item in query_list %} <!--food item-->
    <h5>
        {{item.name}}
        {% if user not in item.loved_by.all %} <!--ability to add favourite-->
            <a href="{% url 'food:add_to_favourite' pk=item.pk %}">
                <button type="button">Add to Favourite</button>
            </a>
        {% else %} <!--... or remove from favourite if already added-->
            <a href="{% url 'food:remove_from_favourite' pk=item.pk %}">
                <button type="button">Remove from Favourite</button>
            </a>
        {% endif %} <!-- end of the first button-->
        {% if user not in item.ignored_by.all %} <!--ability to add ignore list-->
            <a href="{% url 'food:add_to_ignore' pk=item.pk %}">
                <button type="button">Ignore in my meal plans</button>
            </a>
        {% else %} <!--... or remove from ignore list if already added-->
            <a href="{% url 'food:remove_from_ignore' pk=item.pk %}">
                <button type="button">Stop ignoring</button>
            </a>
        {% endif %}
    </h5>
{% endfor %}
```

Food objects and two buttons next to it are displayed in the "Add food page".

12. Calorie Calculator

```python
def calculator(request, user_came_from_planner=0):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    activities_list = Activity.objects.all().order_by("name")  # taking activities from db
    calories = prots = fats = carbs = 0  # context vars
    if request.method == "POST":
        weight = float(request.POST.get('weight'))
        bodyfat = float(request.POST.get('bodyfat'))
        activity_choice = int(request.POST.get('activity'))
        goal = float(request.POST.get('goal'))
        advanced_checkbox = request.POST.get('adCalc')

        bmr = 370 + 21.6 * weight * (100 - bodyfat) / 100
        if advanced_checkbox:
            eat_per_kg = float(request.POST.get('sneaky'))  # hidden input in calculator.html
            calories += int(eat_per_kg * weight / 7)  # Exercise Energy Expenditure is added here
            common_difference = 0.05  # (1.15, 1.20. 1.25, ...)
        else:
            common_difference = 0.15  # (1.15, 1.30, 1.45, ...)
        # arithmetic sequence instead of "if" clause for determining activity coeficient
        activity_coef = 1.15 + common_difference * (activity_choice - 1)

        calories += int(bmr * activity_coef * goal)
        prots = int(weight * 1.5)  # Proteins
        fats = int(weight * 0.9)
        carbs = int((calories - prots * 4 - fats * 9) / 4)  # Carbohydrates
```

```python
        # ----------------------------------------------------------------------
        # Updating User's Latest_inputs
        user = request.user
        try:  # plan a, if everything is OK
            if user.latest_inputs:
                user.latest_inputs.weight = weight
                user.latest_inputs.calories = calories
                user.save()
        except ObjectDoesNotExist:  # plan b, if User "lost" his latest_inputs
            Latest_inputs(user=user, weight=weight, calories=calories).save()
        # ----------------------------------------------------------------------
        if user_came_from_planner:  # coming back to meal planner
            return redirect('planner:meal_planner')
    context = {
        'calories': calories,
        'prots': prots,
        'fats': fats,
        'carbs': carbs,
        'activities': activities_list,
    }
    return render(request, 'calc/calculator.html', context)
```

Server collects the values from the calculator form. Then the modified version of the Calculator Pseudocode Subroutines is executed. After that user's latest inputs are updated/created. Then all the calculated nutrients go to the dictionary and are sent to the Calculator page. If user came to the calculator from meal planner page, instead of creating the dictionary, server sends him back to the meal planner as soon as the Latest_inputs are updated.

13. Server-side scripting

Associating url-patterns with apps/methods in views.py files

FRESHLIFE.urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('mainApp.urls')),
    path('accounts/', include('django.contrib.auth.urls')),
    path('food/', include('food.urls')),
    path('calc/', include('calc.urls')),
    path('planner/', include('planner.urls')),
    path('my_plans/', include('myPlans.urls')),
]
```

mainApp.urls

```
urlpatterns = [
    path('', views.mainPage, name='mainPage'),
    path('register/', views.register, name='register'),
    path('logout/', views.logout_user, name='logout_user'),
]
```

And the similar situation in every app urls.py file

Examples of working with request object:

Funcitons changing the relationships between request user and Food.

```
def remove_from_favourite(request, pk):
    user = request.user
    food = Food.objects.get(pk=pk)
    if user in food.loved_by.all():
        food.loved_by.remove(user)
    return HttpResponseRedirect(request.META.get('HTTP_REFERER'))


def add_to_ignore(request, pk):
    food = Food.objects.get(pk=pk)
    food.ignored_by.add(request.user)
    return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
```

Collecting the values of the *request* from the form via method POST:

```
if request.method == 'POST' and 'create' in request.POST:  # "create" button was clicked
    # getting data from the form and converting to the right data types
    weight = float(request.POST.get('weight'))
    calories = float(request.POST.get('calories'))
    diet = ast.literal_eval(request.POST.get('diet'))  # string representation of a list --> list
    meal_num = int(request.POST.get('meal_num'))
```

14. Arithmetic sequence

In the calc/views,py (see previous technique for full algorithm)

```
17          if advanced_checkbox:
18              eat_per_kg = float(request.POST.get('sneaky'))  # hidden input in calculator.html
19              calories += int(eat_per_kg * weight / 7)  # Exercise Energy Expenditure is added here
20              common_difference = 0.05  # (1.15, 1.20. 1.25, ...)
21          else:
22              common_difference = 0.15  # (1.15, 1.30, 1.45, ...)
23          # arithmetic sequence instead of "if" clause for determining activity coeficient
24          activity_coef = 1.15 + common_difference * (activity_choice - 1)
```

According to the Simple Calculator and Advanced Calculator Pseudocode Subroutines, "IF" clause was used to determine the value of the activity_coefficient. But having two big if clauses in the algorithm (for simple and advanced) is not a good idea, because they take a lot of space and hence doesnot make algorithm readable. Instead, It was noticed that possible values of this coefficient grow linearly and hence it is possible to describe them as arithmetic sequence. In this case both Simple and Advanced options use the same line24, but the value of common difference depends on whether the user checked the advanced option checkbox or not.

15. Finding average

```
17          if advanced_checkbox:
18              eat_per_kg = float(request.POST.get('sneaky'))  # hidden input in calculator.html
19              calories += int(eat_per_kg * weight / 7)  # Exercise Energy Expenditure is added here
```

User inputs his weekly activities, but calculator evaluates the daily recommended energy intake. Hence the energy spendings on sports and other activities per one day are found by taking average. Eat_per_kg is the value of the total activity energy expenditure over a week and found by Java Script algorithm.

16. Exception handling

a) In calc/views.py

```
32          user = request.user
33          try:  # plan a, if everything is OK
34              if user.latest_inputs:
35                  user.latest_inputs.weight = weight
36                  user.latest_inputs.calories = calories
37                  user.save()
38          except ObjectDoesNotExist:  # plan b, if User "lost" his latest_inputs
39              Latest_inputs(user=user, weight=weight, calories=calories).save()
```

The code above is the update of the latest_inputs of the request user. It handles the exception, taken from Django.core.exceptions. If the User has Latest_inputs field, it is updated. If not, the Latest_inputs is created for him. Though Latest_inputs has two "magic" methods which associate it to the User (see OOP.association), Sometimes the error might be raized. For example, old User model instances which were created before Latest_inputs have been defined. They don't have that field. Or in case if the Field was accidentally deleted.

b) In planner/views.py

```
161                 #  trying to use recursion
162                 recursion_has_failed = False
163                 if not use_least_squares:
164                     try:
165                         servings = serving_master(
166                             prots, fats, carbs,
167                             calories, healthscore,
168                             Main, Side, Veg, Other,
169                         )
170                     except RecursionError:
171                         recursion_has_failed = True
```

Another example of exception handling. There are two methods for evaluating servings: Recursive serving_master() and scipy.optimize.lsq_linear(). The first one has more 'natural' outputs, i.e. returned servings are closer to the meal plan which a person would create. But recursive approach is only suitable for diets where carbohydrates prevail fats. Sometimes maximum recursion depth error is raised. Meanwhile Least Square Approximation doesn't errors (at least it didn't raise anything after more than 150 test clicks on the "Create meal plan" button).

If the intake of fats is more or equal than 2 grams per kg, recursion is not even attempted ("Low carb diet"). In case of "Low fat diet" Recursion is tried first and always or almost always successful (during a debugging process 100 runs of low carb diet didn't show recursion fail). In case of "High protein diet" recursion is tried first and occasionally it fails, in which case Leat_Squares is used as defensive choice.

17. HTML DOM Interaction algorithms

The idea is the following:

Create the div, put there the text about the activity. The information about the duration and "epkgh"-value must be wrapped by tag, so they will be accessible, using getElementsByTagName() method. But "epkgh" should not be visible, because this number makes no sense for most users. The div should also have the button, which removes it

Example of structure created by the addAct()

```
▼<div id="act1">
    <span>1</span>
    " hour of Yoga"
    <button type="button" onclick="delAct(1)">Remove</button>
    <span style="visibility: hidden;">3.2</span>
  </div>
```

The set of JavaScript functions not only creates and removes HTML, but also changes the value of EATperKg variable, which stands for Energy expenditure during user's workouts. The value goes into the number box, which is a part of the form, but invisible for the user. Then the user submits the form, this value is submitted as well, but user may not even know about it.

Calc/static/calc/calculator.js

```
1   function removeElement(elementId) {
2       var element = document.getElementById(elementId);
3       element.parentNode.removeChild(element);
4   }
5   function updateSneakyInput() {
6       sneakyInput = document.getElementById("sneaky");
7       sneakyInput.value = EATperKg.toFixed(2); // two decimal places
8   }
9   function advancedOption(){
10      var checkBox = document.getElementById("advanced_check");
11      var adCalc = document.getElementById("advanced_calc");
12      if (checkBox.checked == true){
13          adCalc.style.display = "block";
14      } else {
15          adCalc.style.display = "none";
16      }
17  }
```

Lines 1-4: function which removes the element and needs it id. Is used to remove activities.
Lines 5-8: function which updates the value inside the secret input box.
    Line 6: accessing text box by its id in html
    Line 7: changing its value
Lines 9-17: Checkbox operating function
    Lines 10 and 11: accessing the checkbox and the container (with advanced option content) respectively
    Lines 12-16: If clause. If the check box is checked, entire div becomes visible, else its becomes invisible.

```
18  function addAct(){
19      var time = document.getElementById("hours").value; // time from the time input
20      if (time){
21          var select = document.getElementById("sel"); // select element
22          var name = select.options[select.selectedIndex].innerHTML; // name of chosen activity
23          var epkgh = select.options[select.selectedIndex].value; // value of chosen activity
24          var container = document.getElementById("user_act-s"); // div where new activities created
25
26          //creating div for new activity
27          var newDiv = document.createElement("div");
28          actIdCounter = actIdCounter + 1; // incrementing the identifier of the block
29          newDiv.id = "act" + actIdCounter;
30          //creating spans for time and epkgh, because this values must be accessible
31          var timeSpan = document.createElement("span");
32          timeSpan.innerHTML = time;
33          var epkghSpan = document.createElement("span");
34          epkghSpan.innerHTML = epkgh;
35          epkghSpan.setAttribute("style", "visibility: hidden;"); // user don't need to see it
36          // but it has to be stored nearby for delAct() fuction
37
38          //creating button
39          var clickme = document.createElement("button"); // creating delete button
40          clickme.innerHTML = "Remove";
41          clickme.type = "button";
42          clickme.setAttribute("onclick", "delAct(" + actIdCounter + ")");
43          // arranging div elements in order with plain text inbetween
44          newDiv.appendChild(timeSpan)
45          newDiv.innerHTML += " hour"; if (time != 1){newDiv.innerHTML += "s"};
46          newDiv.innerHTML += " of " + name;
47          newDiv.appendChild(clickme);
48          newDiv.appendChild(epkghSpan);
49          container.appendChild(newDiv); // adding new created div to the container div
50
51          //calculating calories contribution
52          time = parseFloat(time);
53          epkgh = parseFloat(epkgh);
54          EATperKg += epkgh * time;
55          updateSneakyInput();
56      } else { alert("Please, enter the number of hours"); }
57  }
```

Line 19: accessing the value of number of hours, entered by the user.
Line 20-57: if clause/validity check. If the entry is absent, user will be asked to enter it (line 56)
Line 21-23, accessing select tag and taking the value(epkgh) of it and the name of chosen activity
Line 24, accessing the special empty div there all activities are going to appear and storing it in container variable

Lines 26-29: Creating div element and setting its id-value. Please notice, that its id-value depend on the actIdCounter variable, which is in fact holds the value of the number of the last created activity-div. before creating new div it is incrementing(line 28), so the new div will have different id. As a result the id-attribute of the div tag is will not repeat.

Lines 31-35: Creating two spans and filling them with data, making epkghSpan hidden.

Lines 39-42: Creating button, setting its text, type and onclick values. Onclick value is delAct(id) method, which deletes activity div with corresponding id. In the example of what the function addAct() creates (image of the html) the div has id=”act**1**” and onclick=”delAct(**1**)”, which is related.

Lines 44-49: Filling the activity div with content and appending that div to the “parent” div.

Lines 52-55: Changing the type of data to float, adding their product to the EATperKg, which stands for “Energy per kg”, to get the actual calories, it should be multiplied by weight, which is done in updated calculator view in views.py.

Line 55: Updating the hidden input with new EATperKg value.

```
58  function delAct(id){
59      // collecting data from spans and converting to float
60      var time  = document.getElementById("act" + id).getElementsByTagName("span")[0].innerHTML;
61      var epkgh = document.getElementById("act" + id).getElementsByTagName("span")[1].innerHTML;
62      time = parseFloat(time);
63      epkgh = parseFloat(epkgh);
64
65      EATperKg -= epkgh * time; //calculating calories contribution
66
67      updateSneakyInput();
68
69      removeElement("act" + id); //removing activity note
70  }
```

The last function in the script file, delAct(), which deletes the div with activity when “remove” button is pressed. Entry argument id is the number inside div id attribute, like “act1” or ”act222”.

Line 60-61. This function also has time and epkgh attributes, but they are taken from the spans inside the div with id attribute, which has form: “actX”, there X is a number.

Line 62-67: the same as lines 52-55, but subtracting the energy expenditure change.
Line 69: using defined function to remove the div with a single activity

Additional explanations and justifications

Hidden inputs approach for saving plans.
How the meal planner displays and saves a meal plan.

There are two buttons on the Meal planner page: “create” and “save”. Then the user presses “create” button, objects are not created, Plan details are displayed as a text for client perspective. But in fact, it is not just a text, but a form, which helps to create object.

This is the code of the form. User sees it as several heading tags.

```
<form method="post"> <!--Form with hidden inputs which are filled by the server.-->
    <!--It contains all the information for the DB to be saved. User just presses 'save'-->
    {% csrf_token %}
        <!-- hidden information about plan -->
        <input class="stealth" name="meal_num" type="number" value="{{ plan|length }}">
        <input class="stealth" name="three_or_four" type="number" value="{{ plan.1|length }}">
        {% for meal in plan %} <!-- display the meals in plan-->
            <div class="meal_div">
                <h3>Meal {{ forloop.counter }}</h3>
                {% with outer=forloop %}
                    {% for food in meal %} <!-- display food in the meal -->
                        <h4>{{ food.0 }} - {{ food.1 }} {{ food.2 }}</h4>
                        <input class="stealth"
                               name="food_{{outer.counter}}{{forloop.counter}}"
                               type="text"
                               value="{{ food.0 }}"><!--hidden input of the food name-->
                        <input class="stealth"
                               name="serving_{{outer.counter}}{{forloop.counter}}"
                               type="number"
                               value="{{ food.1 }}"><!--hidden input of the food serving size-->
                    {% endfor %}
                {% endwith %}
            </div>
        {% endfor %}
    <fieldset> <!-- visible part of the form -->
        <legend>You can save the plan here</legend>
        <input type="text" name="plan_name" placeholder="Name of the meal plan" required>
        <input type="submit" name="save" class="ocb" value="save"> <!--SAVE button-->
    </fieldset>
</form>
```

The form is submitted by the “save” button.
This is the python code, which takes the values from the form and using them creates the plan, meals and Meal_has_Food entities.

```
    if request.method == 'POST' and 'save' in request.POST:  # "save" button was clicked

        new_plan = Plan(  # creating plan with name provided by user
            owner=request.user,
            name=request.POST.get('plan_name'),  # user input
        )
        new_plan.save()
        #  nested looping for, there meal number and food "indecies", m and f, start with 1, not 0
        for m in range(1, int(request.POST.get('meal_num')) + 1):
            meal = Meal(plan=new_plan, index=m)  # creating m-th meal
            meal.save()
            for f in range(1, int(request.POST.get('three_or_four')) + 1):
                food_name = request.POST.get('food_{}{}'.format(m, f))  # finding
                serving = int(request.POST.get('serving_{}{}'.format(m, f)))
                Meal_has_Food(  # linking f-th food to m-th meal
                    meal=meal,
                    food=Food.objects.get(name=food_name),
                    portion_size=serving,
                    ).save()

    context = {  # sending the data and logic to the mealplanner.html
        "enough_favourites": enough_favourites,
        "reasons": reasons,
        "plan": plan,
        "meal_num": meal_num,
    }
    return render(request, 'planner/mealplanner.html', context)
```

Class “stealth” makes elements invisible for the user (display:none;). Jinja creates several text and number boxes for the data, which is must-know in order to create a plan. The following data has its own invisible text box:

- Number of meals. It is essential, because python algorithm needs to know how many iterations to perform.

- Three or Four. This is not the best name, but it shows how many categories of Food are used in meal. Food categories are Main, Side, Vegetables, Other. Other is optional, others are compulsory for a meal. Presence or absence of Other in a meal affects the number of iterations inside a nested loop (3 or 4).

- For each meal, there are two input boxes:
    - Food name. Important, so that python will be able to find the Food instance by name
    - Serving of that food in the meal. Important entry for creating Meal_has_Food instance.

As a result User creates not only a text of the plan, but also a form, which size and number of fields can vary depending on the created plan. The form is mostly invisible and user only sees "save" button and a text box, where he enters the name of the plan.

The purpose of such approach is to separate the creation and saving of the plan. As a result User saves the plan only when he wants to and database does not gets filled with unwanted data.

Tests

Test Plan

Test	Purpose/description	Objective	Entries	Expected result	Actual result
1	Create users and see them appear in the database (appear in the Django Admin pane)	Login system (A)	Username, Password, button click	Successfully added user	Successfully
2	Calculate daily energy expenditure and macronutrients	Calories calculator(B) Daly macronutrients (D)	Number, radio buttons choice, submit button click (calculator form)	Number in the range 2800-3000 Proteins = 115-145 Fats = 70-90 Carb-es = 380-430 (grams per day)	Result: 2890 kcal per day Proteins = 135 Fats = 81 Carb-es = 405 (grams per day)
3	Activate advanced option and calculate calories. Compare the result with simple version of calculator	Advanced options for calories calculator (C)	Everything in 2 + checkbox and several button clicks	No problems with adding/removing activities. Number in the range 2800-3000	No problems. Calories: 2820
4	See the reflection of adding to favourite in the database.	Favourite foods (E)	Button click	Django administration shows changes	Django administration shows changes
5	See the reflection of adding to ignore in the database. Make sure "addFood" doesn't suggest ignored food and ignored food is not in the meal plans	Food ignore list (F)	Button click	Django administration shows changes	Django administration shows changes
6	Search some food	Search (G)	Textbox entry and button click	Search results are food which contain the text in the entry.	Search results are food which contain the text in the entry.
7	Make sure that "add food" doesn't suggest the food user marked as ignored or favourite	Suggest choice (H)	Marking all food as favourite or ignored and reloading page	Doesn't suggest anything	Doesn't suggest anything

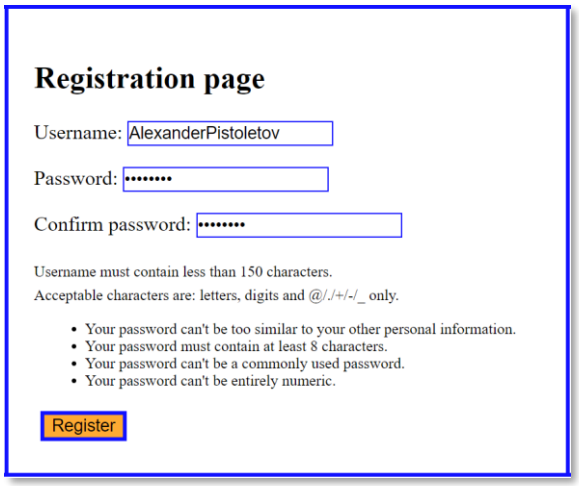
8	Independent users test the application and give feedback	User-Friendly Interface (I)		Majority say that the application is easy to use	Majority said that the app is easy to use, but some changes in the program has been done according to the feedback.
9	Create meal plans. Trying every diet option	Nutrition Planner (J) Different styles of diet (O)	Meal plan form filled and submitted	Meal plans have proper number of meals, every meal has Main course, side choice and Salad/vegetable choice No crashes, negative values or extremely big values of servings.	All satisfied
10	Create meal plans based on only the favourite food	Favourite food only (K)	Meal plan form filled and submitted	All food in the meal plan is in favourites	All food in the meal plan is in favourites
11	See what happens if meal plan cannot be created	Send user to the "Add Food" (L)	Meal plan form filled and submitted	Short message and "add food" button appear on the screen	Short message and "add food" button appear on the screen
12	Try to save meal plan and view it in the "My meal plans".	Save meal plans to the database (M) User meal plans (N)	Text entry and button click	Mela plan appears in Django admin and in user's "My plans"	Mela plan appears in Django admin and in user's "My plans"

Tests reports.

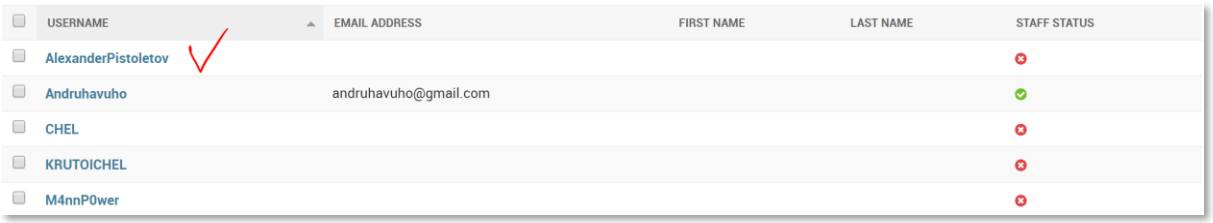
- The tests are ordered like in the test plan, but not necessarily chronologically.

1. Login system

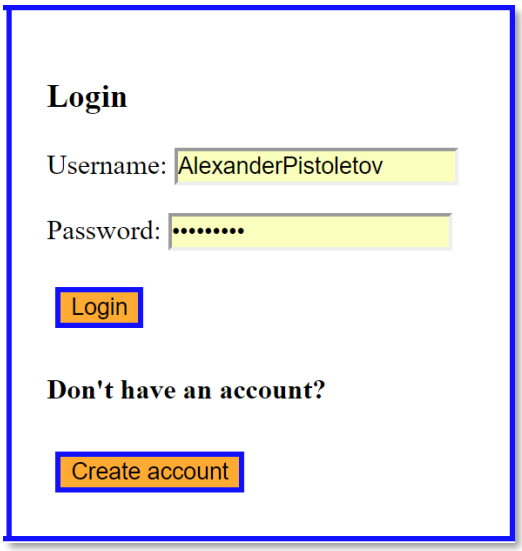
Description	Attempt to create new User by registration, then logout and login back.
Entries	Username, Password, Confirm password, Form submission (Registration). Button click (Logout) Username, Password, Form submission (Login)
Expectation	Successful creation of User, row representing the User should appear in the Django Admin page. After registration User should be redirected to the Main menu. After logging out and logging in The same user should be authenticated
Actual result	All expectations have been satisfied



Meanwhile in the Django administration (opened in incognito window)



Logging in



Another evidence that user have been created is that during Logging in Google Chrome suggested to auto fill the fields (textboxes became yellow).

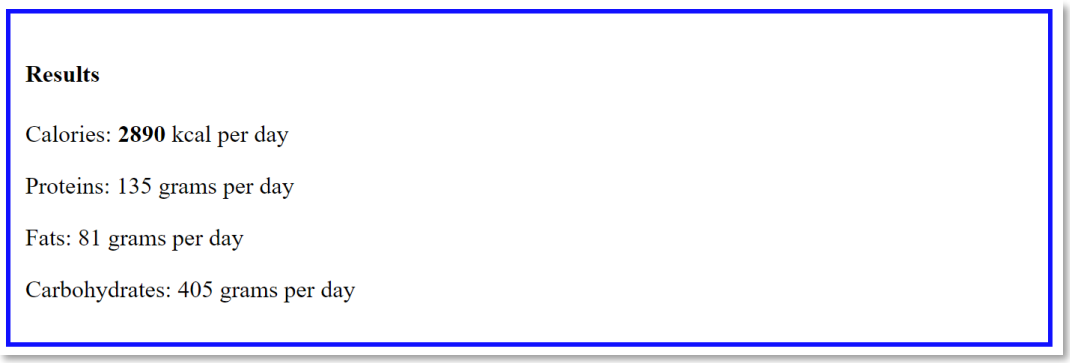
2. Calorie calculator

Description	Use the calculator to evaluate the values of the Daily energy intake
Entries	Weight=”90”, Bodyfat percentage radio buttons = “13-20%” Activity radio buttons = “Medium activity” Goal radio buttons = “Preserve weight” “Calculate” button click

Expectation	Acceptable ranges of values Calories = 2800-3000 Proteins = 115-145 Fats = 70-90 Carb-es = 380-430 (grams per day)
Actual result	• 2890 kcal per day • Proteins = 135 • Fats = 81 • Carb-es = 405 (grams per day)

Image bellow shows entries

Image bellow shows results



Additional trials:

Entries	Outcomes
• Weight=”90”, • Bodyfat percentage radio buttons = “8-12%” • Activity radio buttons = “High activity” • Goal radio buttons = “Gain weight”	• 3217 kcal per day • Proteins = 112 • Fats = 67 • Carb-es = 541 (grams per day)
• Weight=”105”, • Bodyfat percentage radio buttons = “20-30%” • Activity radio buttons = “Low activity” • Goal radio buttons = “Lose weight”	• 2423 kcal per day • Proteins = 157 • Fats = 94 • Carb-es = 237 (grams per day)
• Weight=”88”, • Bodyfat percentage radio buttons = “13-20%” • Activity radio buttons = “Medium activity” • Goal radio buttons = “Lose weight”	• 2554 kcal per day • Proteins = 132 • Fats = 79 • Carb-es = 328 (grams per day)
• Weight=”65”, • Bodyfat percentage radio buttons = “16-19%” • Activity radio buttons = “Low activity” • Goal radio buttons = “Lose weight”	• 1683 kcal per day • Proteins = 90 • Fats = 54 • Carb-es = 209 (grams per day)

3. Advanced option of the calorie calculator

Description	Use the calculator with advanced option to evaluate the values of the Daily energy intake
Entries	Weight = ”90”, Bodyfat percentage radio buttons = “13-20%” Advanced option checkbox. Activities are: • 2 hours of Weight training • 3 hours of Intense weight training Activity radio buttons = “Medium activity” Goal radio buttons = “Preserve weight”

	“Calculate” button click
Expectation	No problems with advancing option performace Acceptable ranges of values: • Calories = 2800-3000 • Proteins = 115-145 • Fats = 70-90 • Carbohydrates = 380-430 (grams per day)
Actual result	2820 kcal per day Proteins = 135 Fats = 81 Carbohydrates = 387 (grams per day)

Image bellow shows user entries:

90 kg

Estimate your bodyfat ratio using the chart below

- 3-4%
- 5-8%
- 8-12%
- 13-20%
- 20-30%
- 30-50%

- 11-13%
- 14-15%
- 16-19%
- 20-24%
- 25-30%
- 30-50%

☑ Use advanced option to describe training activity in details

Describe your typical weekly activities and workouts

During a week I usually do Weight Training (High Intensity) ▾ for 3 hours add

2 hours of Weight training Remove

3 hours of Weight Training (High Intensity) Remove

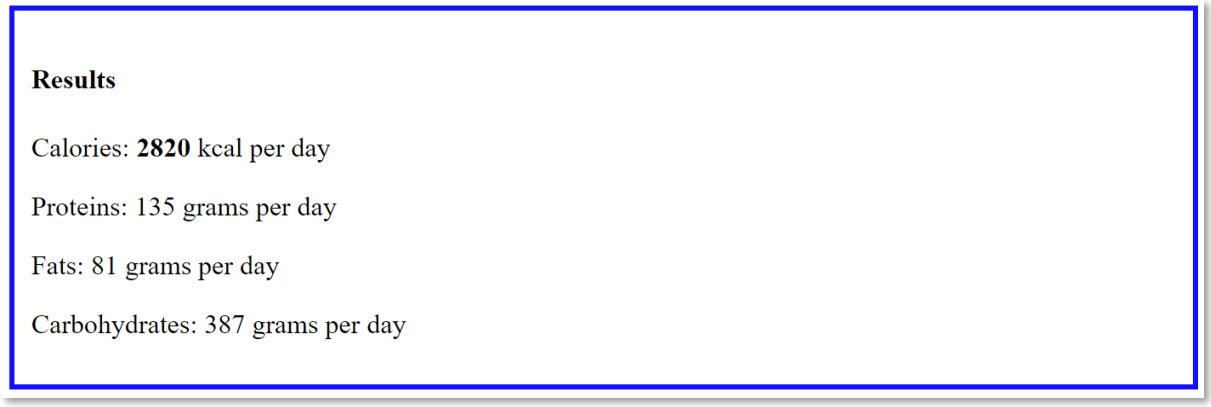
How active are you?

- Sitting lifestyle
- Low activity (Walking, housekeeping or little exersice)
- Medium activity (Training 3-5 times a week)
- High activity (Active lifestyle, training 6-7 times a week)
- Extreme activity (Train like an athlete, physical labour)

What is your goal?

- Lose weight
- Preserve weight

The image bellow shows the result:



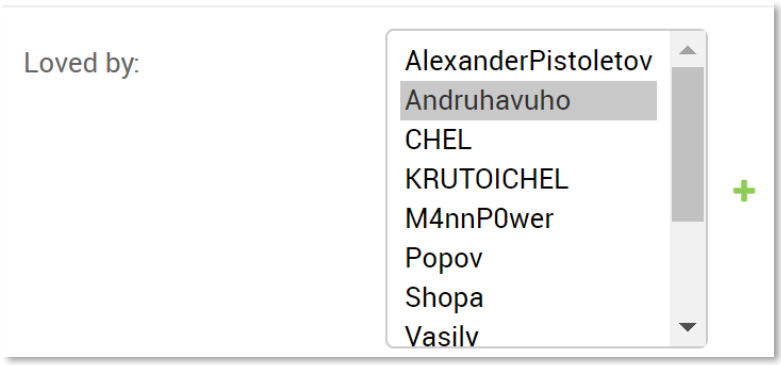
4. Favourite foods

Description	See if changes in the database occur when certain buttons are pressed.
Entries	"Add to favourite" button click, "Remove from favourite" button click,
Expectation	Django administration show changes
Actual result	All as expected

Logged in as User with username AlexanderPistoletov:
BEFORE clicking "Add to favourite":

Client perspective

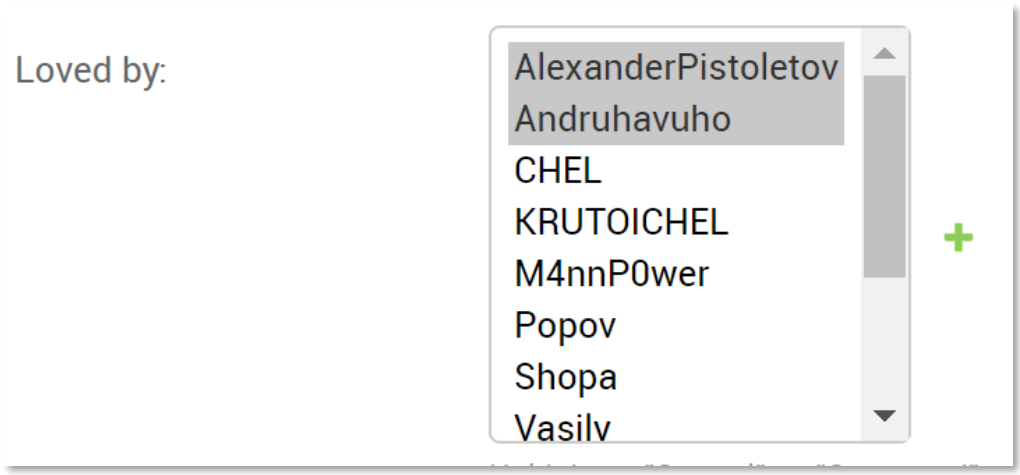
Admin perspective



AFTER clicking “Add to favourite”:

Client perspective:

Admin perspective:



After pressing “Add to favourite” button near Asparagus, Django administration site highlighted User AlexanderPistoletov in the *Loved_by* field of the Asparagus. In addition, the text on the button changed to “Remove from favourite”. Clicking “Remove from favourite” has exactly opposite action.

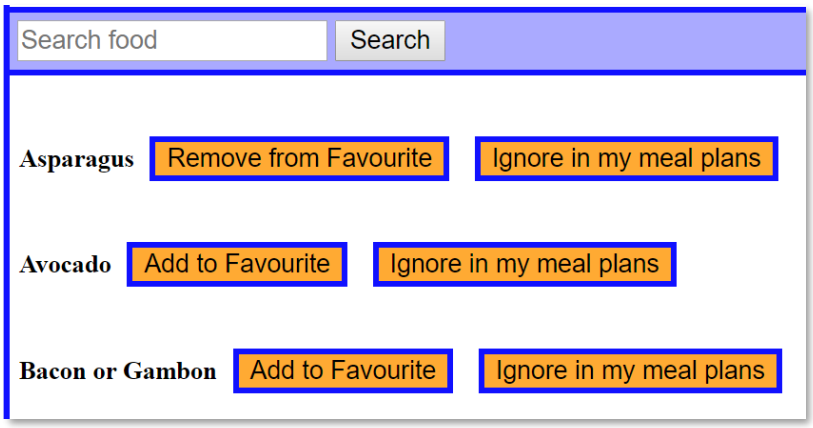
5. Ignore list

Description	See if changes in the database occur when certain buttons are pressed.
Entries	“Ignore in my meal plans” button click, “Stop ignoring” button click,
Expectation	Django administration show changes
Actual result	All as expected

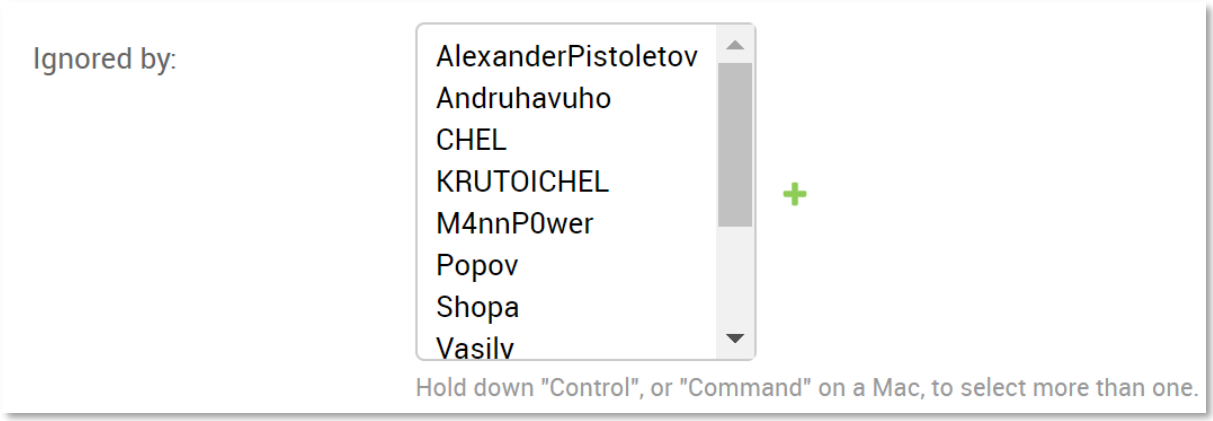
User AlexanderPistoletov is logged in. Admin is logged in in the incognito window.

BEFORE clicking “Ignore in my meal plans”:

Client’s Perspective:



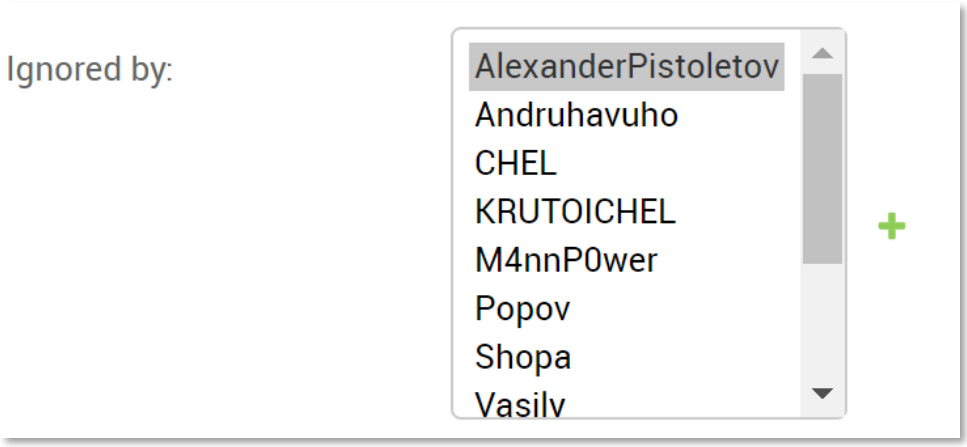
Admin’s perspective



AFTER clicking “Ignore in my meal plans”:

Client’s perspective

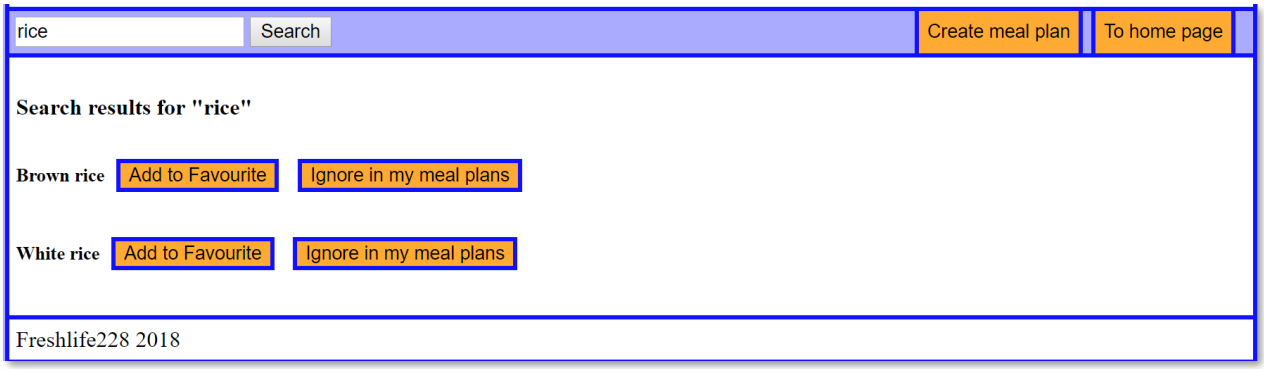
Admin's perspective

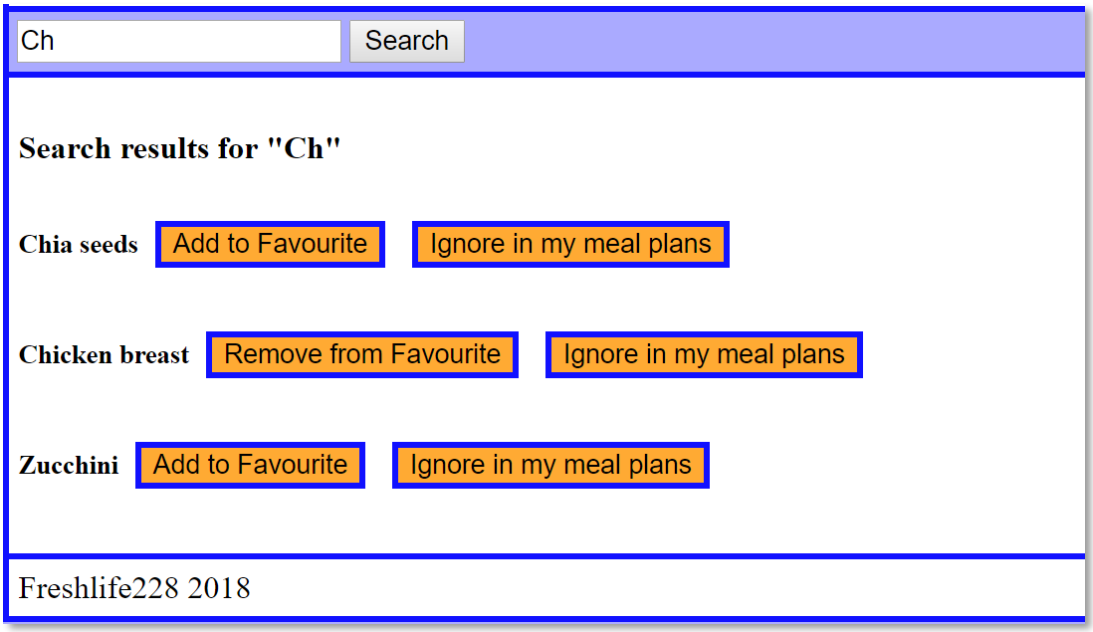


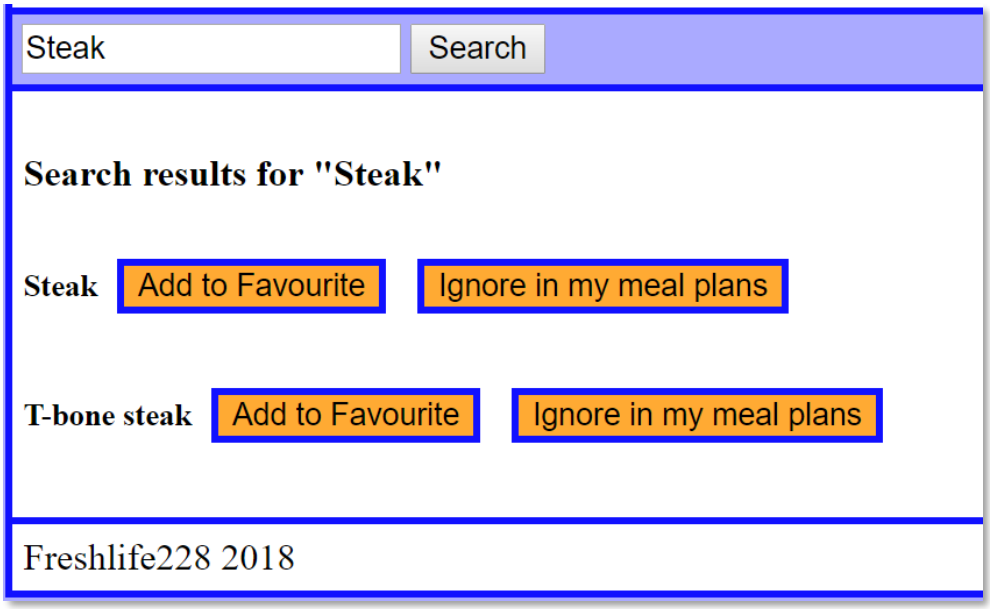
After pressing "Ignore in my meal plans" button near Asparagus, Django administration site highlighted User AlexanderPistoletov in the *Ignored_by* field of the Asparagus. In addition, the text on the button changed to "Stop ignoring". Clicking "Stop ignoring" has exactly opposite action.

6. Search

Description	Use searching form to find some food
Entries	• "rice" • "Ch" • "Steak"
Expectation	• White rice and Brown rice • Chicken breast, Chia seeds and Zucchini • Steak and T-bone steak
Actual result	All as expected (see images bellow)


rice
Search
Create meal plan
To home page
Search results for "rice"
Brown rice
Add to Favourite
Ignore in my meal plans
White rice
Add to Favourite
Ignore in my meal plans
Freshlife228 2018

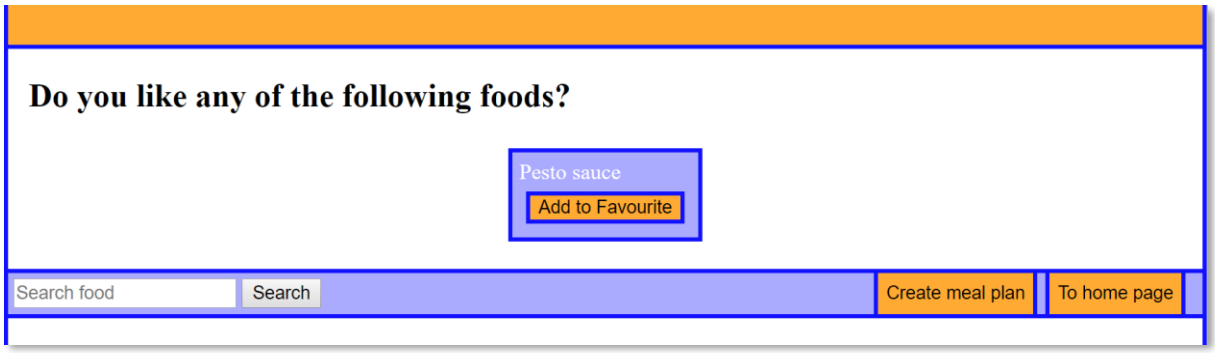

Ch
Search
Search results for "Ch"
Chia seeds
Add to Favourite
Ignore in my meal plans
Chicken breast
Remove from Favourite
Ignore in my meal plans
Zucchini
Add to Favourite
Ignore in my meal plans
Freshlife228 2018


Steak
Search
Search results for "Steak"
Steak
Add to Favourite
Ignore in my meal plans
T-bone steak
Add to Favourite
Ignore in my meal plans
Freshlife228 2018

7. Suggest

Description	Try to mark all food as ignored or favourite and see what will be suggested
Entries	Reloading page after every food is marked
Expectation	Suggestion block will not be displayed
Actual result	Suggestion block is not displayed

When there's only one food not in favourite or in the Ignorelist, Suggestion block shows only one item instead of usual three:

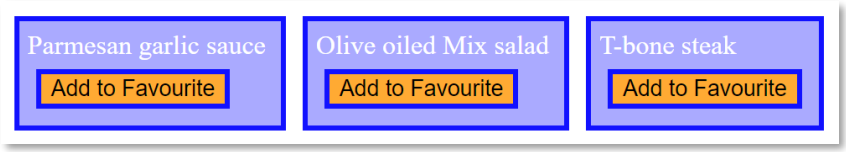


After adding Pesto Sauce, there's no possible suggestions so nothing is suggested:

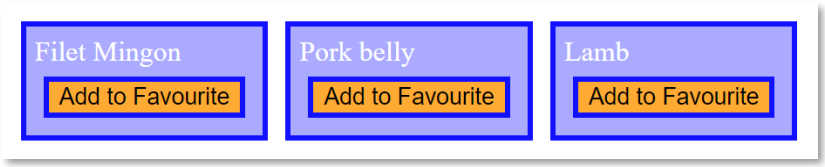
Additional test:

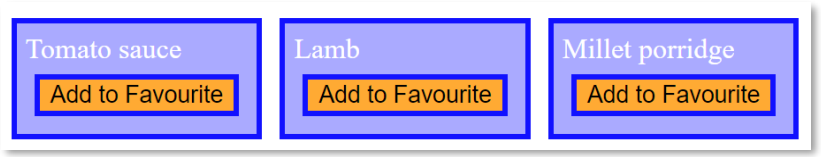
Description	Update page several times to see if the suggestion algorithms suggest different food every time
Entries	Reloading pages pages. Only Pasta, Brown rice, Chicken breast, Salmon and Mixed Vegetables added to favourite food
Expectation	Suggestion block shows different foods but not any of the listed one cell above. In addition, the same food must not be suggested twice in one run. In other words, all three suggestions must be unique.
Actual result	Within ten reloads, none of the Pasta, Brown rice, Chicken breast, Salmon and Mixed Vegetables were suggested.

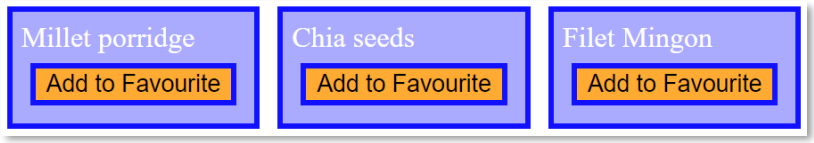
Outcomes

1. 

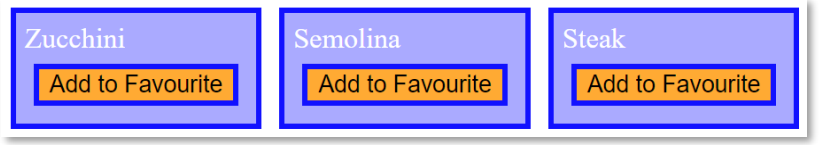
2. 

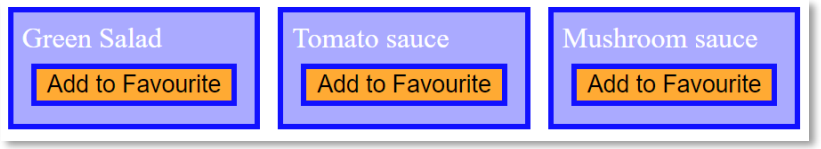
3. 

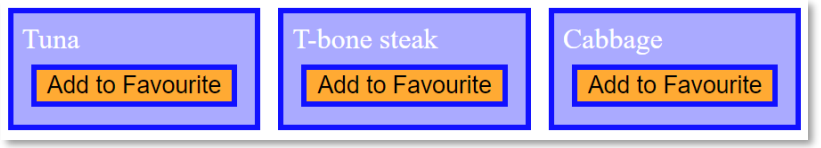
4. 

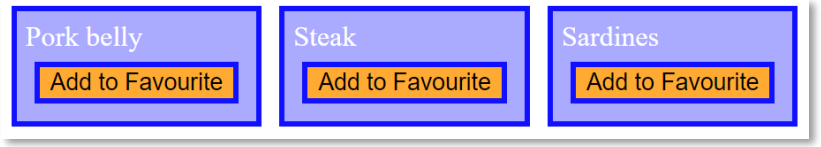
5. 

6. 

7. 

8. 

9. 

10. 

8. User-Friendliness

Description	5 people try to use the application without any guidance.
Expectation	At least 4 people say that the application is easy to use.
Actual result	5 people said that the program has a friendly interface, but 2 of them said that they didn't know how to come back to the Main Page from calculator and meal planner.
Changes in the program	Additional navigation button "Go to Main Menu" is added to calculator and meal planner.

Changes in the program

The changes are adding link with button inside to the heading tag and changing some css, so meal planner and calculator look more the same than they are different.

Calculator:

Before:

```
{% block content %}
<div id="area">
    <h1>Calorie Calculator</h1>
    {% if calories %} <!--Display results-->
```

After:

```
{% block content %}
<div id="area">
    <h1>
        Calorie Calculator
        <a href="{% url 'mainApp:mainPage' %}">
            <button type="button" style="float:right;"> Go to Main Menu</button>
        </a>
    </h1>
    {% if calories %} <!--Display results-->
```

Meal planner:

Before:

```
{% block content %}
<h1 style="text-align: center;">Meal Planner</h1>
<div id="setup">
    <form method="post" style="..."> <!--form for meal details-->
        {% csrf_token %}
```

After:

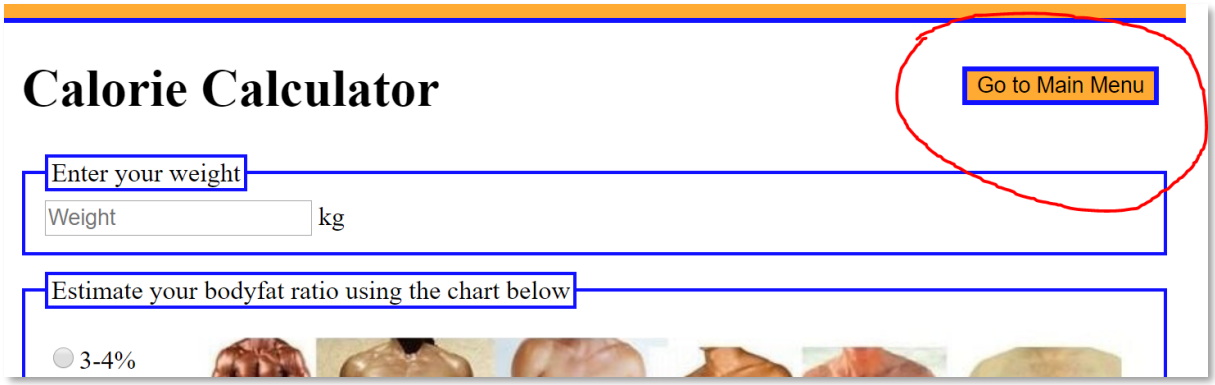
```
{% block content %}
<div id="setup">
    <h1>
        Meal Planner
        <a href="{% url 'mainApp:mainPage' %}">
            <button type="button" style="float:right;"> Go to Main Menu</button>
        </a>
    </h1>
    <form method="post" style="..."> <!--form for meal details-->
        {% csrf_token %}
```

1. Added the button structure inside <h1>.
2. Removed the text-align of the <h1>. Now it's "left" by default.
3. Header is now inside the main container (id="setup") with the rest of the content

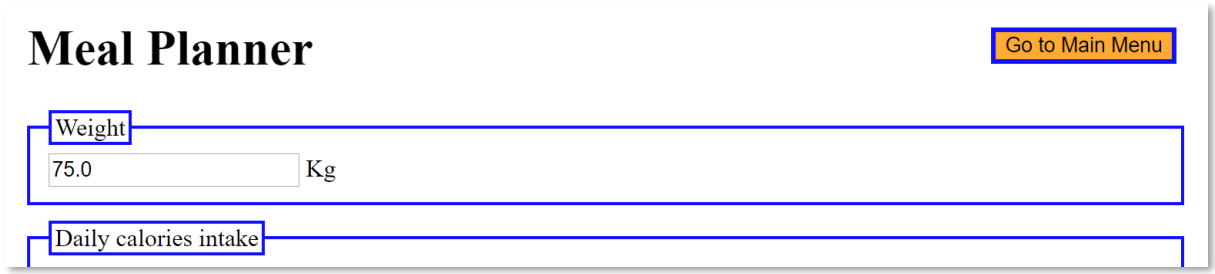
---

Updated User Interface.
How the slight change affected the appearance of the GUI.

New Calculator:



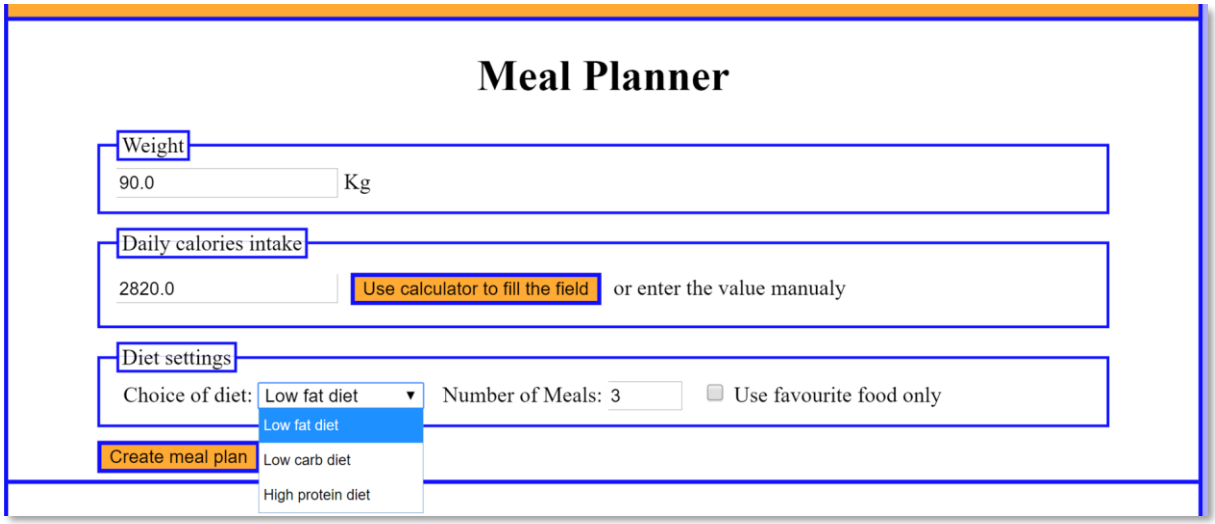
New Meal planner:



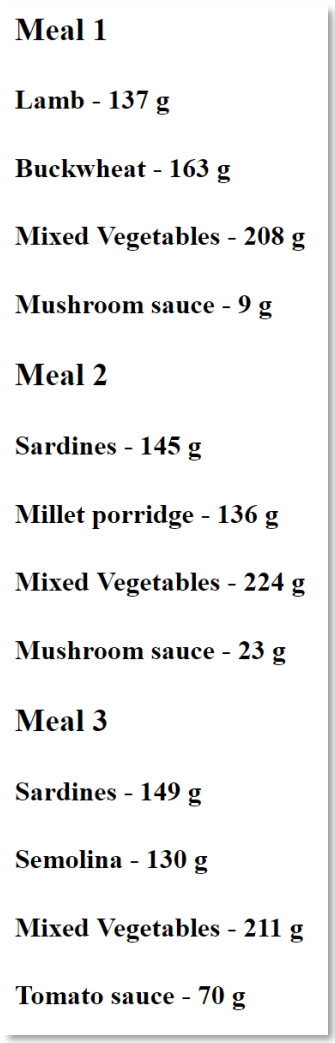
9. Nutrition planner and Different styles of diet

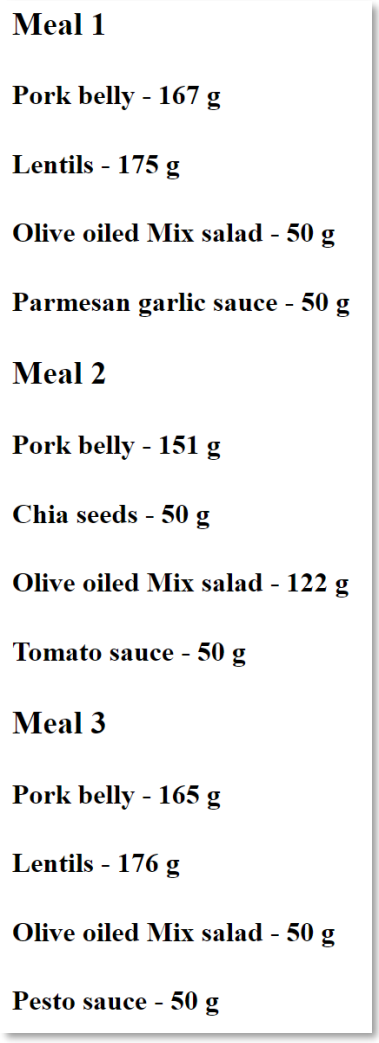
Description	Using Meal planner to create some meals. Trying every diet options and different meal number options.
Entries	• Weight = 90.0 • Calories = 2820.0 • Type of diet (every option was tried) • Number of meals = 3 and 4
Expectation	Meal plans have proper number of meals, every meal has Main course, side choice and Salad/vegetable choice No crashes, negative values or extremely big values of servings.
Actual result	All expectations are satisfied. Plans with different meal numbers and types of diet have been created. Serving values are within acceptable range

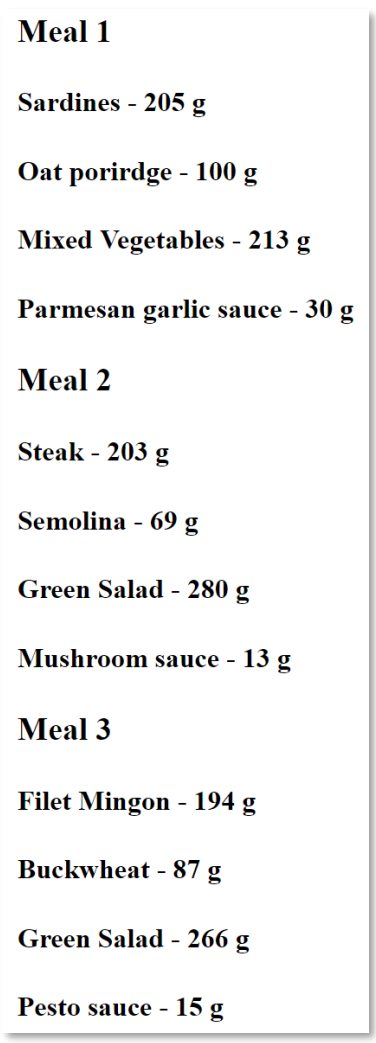
The image below shows the form for meal generation. Every time different style of diet has been used.



The following three images show the meals generated for Low fat diet, Low carb diet and High protein diet respectively.







Low fat diet with four meals:

Meal 1

Steak - 114 g

Brown rice - 93 g

Green Salad - 201 g

Tomato sauce - 30 g

Meal 2

Lamb - 108 g

Pasta - 110 g

Zucchini - 210 g

Parmesan garlic sauce - 9 g

Meal 3

Salmon - 124 g

Semolina - 101 g

Mixed Vegetables - 144 g

Mushroom sauce - 16 g

Meal 4

Chicken breast - 109 g

Brown rice - 117 g

Cabbage - 210 g

Parmesan garlic sauce - 26 g

Meal 1

Steak - 105 g

White rice - 107 g

Zucchini - 200 g

Tomato sauce - 37 g

Meal 2

Steak - 101 g

Millet porridge - 120 g

Cabbage - 208 g

Pesto sauce - 8 g

Meal 3

Sardines - 115 g

Pasta - 99 g

Mixed Vegetables - 168 g

Parmesan garlic sauce - 21 g

Meal 4

T-bone steak - 106 g

Millet porridge - 125 g

Zucchini - 208 g

Pesto sauce - 10 g

The program can split your calories in up to 9 meals:

Meal 1

Pork belly - 50 g

Chia seeds - 17 g

Olive oiled Mix salad - 39 g

Parmesan garlic sauce - 17 g

Meal 2

Pork belly - 50 g

Chia seeds - 17 g

Olive oiled Mix salad - 37 g

Pesto sauce - 17 g

Meal 3

Pork belly - 56 g

Lentils - 58 g

Olive oiled Mix salad - 17 g

Pesto sauce - 17 g

Meal 4

Pork belly - 50 g

Chia seeds - 17 g

Olive oiled Mix salad - 39 g

Parmesan garlic sauce - 17 g

Meal 5

Pork belly - 57 g

Lentils - 58 g

Olive oiled Mix salad - 17 g

Parmesan garlic sauce - 17 g

Meal 6

Pork belly - 56 g

Lentils - 58 g

Olive oiled Mix salad - 17 g

Pesto sauce - 17 g

Meal 7

Pork belly - 50 g

Chia seeds - 17 g

Olive oiled Mix salad - 38 g

Mushroom sauce - 17 g

Meal 8

Pork belly - 50 g

Chia seeds - 17 g

Olive oiled Mix salad - 37 g

Pesto sauce - 17 g

Meal 9

Pork belly - 56 g

Lentils - 58 g

Olive oiled Mix salad - 17 g

Pesto sauce - 17 g

Additional trials with different meal numbers and diets, but for the same calories:

Meal 1

Tuna - 128 g

Pasta - 51 g

Cabbage - 138 g

Pesto sauce - 17 g

Meal 2

Filet Mingon - 91 g

Buckwheat - 70 g

Zucchini - 130 g

Pesto sauce - 7 g

Meal 3

Salmon - 89 g

Oat porirdge - 40 g

Green Salad - 100 g

Mushroom sauce - 8 g

Meal 4

Tuna - 127 g

White rice - 47 g

Cabbage - 138 g

Pesto sauce - 18 g

Meal 5

Tuna - 100 g

Buckwheat - 48 g

Mixed Vegetables - 80 g

Tomato sauce - 32 g

Meal 6

Salmon - 89 g

Brown rice - 37 g

Green Salad - 100 g

Parmesan garlic sauce - 10 g

Meal 1

Chicken breast - 95 g

Pasta - 70 g

Green Salad - 168 g

Parmesan garlic sauce - 21 g

Meal 2

Salmon - 105 g

Oat porirdge - 92 g

Green Salad - 132 g

Mushroom sauce - 11 g

Meal 3

Chicken breast - 87 g

Brown rice - 102 g

Zucchini - 166 g

Pesto sauce - 18 g

Meal 4

Chicken breast - 91 g

Millet porridge - 77 g

Green Salad - 157 g

Tomato sauce - 49 g

Meal 5

Filet Mingon - 83 g

White rice - 74 g

Mixed Vegetables - 127 g

Tomato sauce - 36 g

Meal 1

Sardines - 235 g

White rice - 213 g

Cabbage - 423 g

Tomato sauce - 132 g

Meal 2

Tuna - 322 g

Brown rice - 266 g

Zucchini - 415 g

Tomato sauce - 164 g

Meal 1

Pork belly - 243 g

Chia seeds - 17 g

Olive oiled Mix salad - 369 g

Pesto sauce - 17 g

Meal 2

Pork belly - 243 g

Chia seeds - 17 g

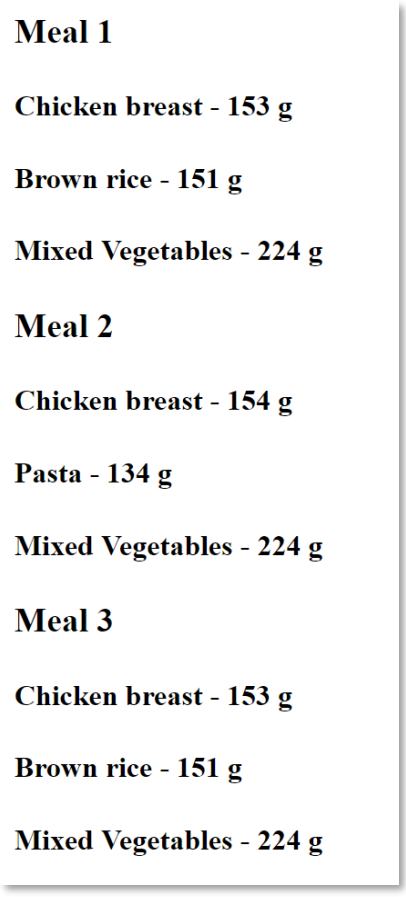
Olive oiled Mix salad - 369 g

Pesto sauce - 17 g

10. Favourite food only

Description	Create meal plan which contains on only favourite food
Entries	• Pasta, Brown rice, Chicken breast, Salmon and Mixed Vegetables added to favourite food • Weight = 90.0 • Calories = 2820.0 • Type of diet = Low fat diet • Number of meals = 3 • Favourite food only checkbox checked • "Create" button click (form submission)
Expectation	Meals must contain nothing but: Pasta, Brown rice, Chicken breast, Salmon and Mixed Vegetables
Actual result	All as expected.

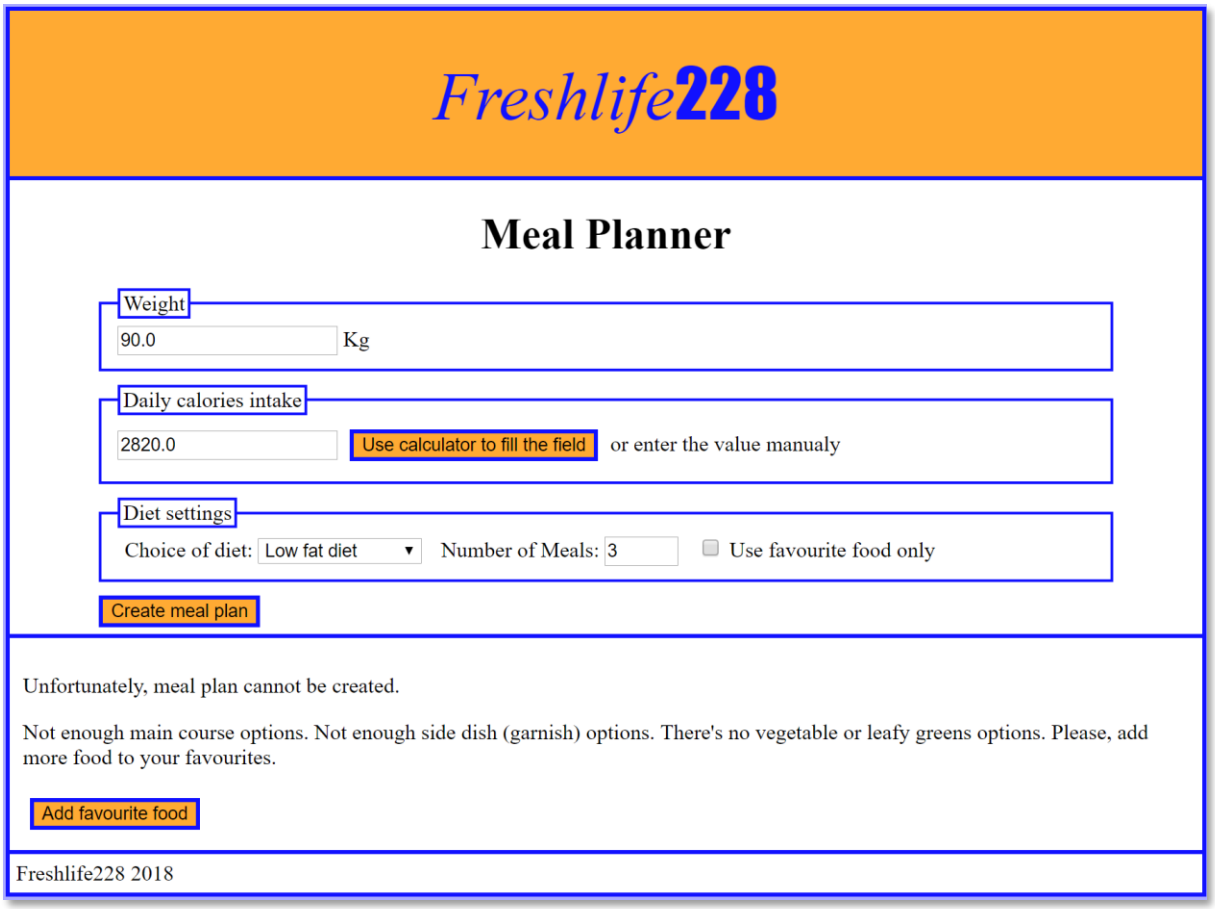
The images bellow shows the meal plans generated with the given entries (several trials were made)



11. Send to add food

Description	Try to create meal plan without favourite food, while checking "favourite food only" checkbox
Entries	• Removed all food from favourites • Favourite food only checkbox

	• Weight = 90.0 • Calories = 2820.0 • Type of diet = Low fat diet • Number of meals = 3 • checked "Create" button click (form submission)
Expectation	Short message and "add food" button appear on the screen
Actual result	All as expected. Short message and "add food" button appeared on the screen

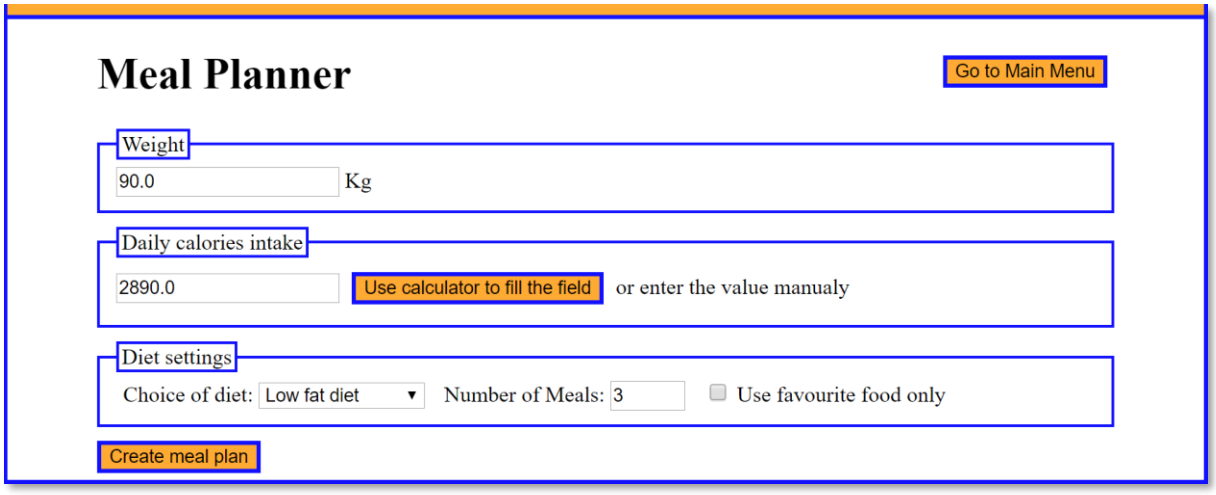


12. Save plan, Display meal plans

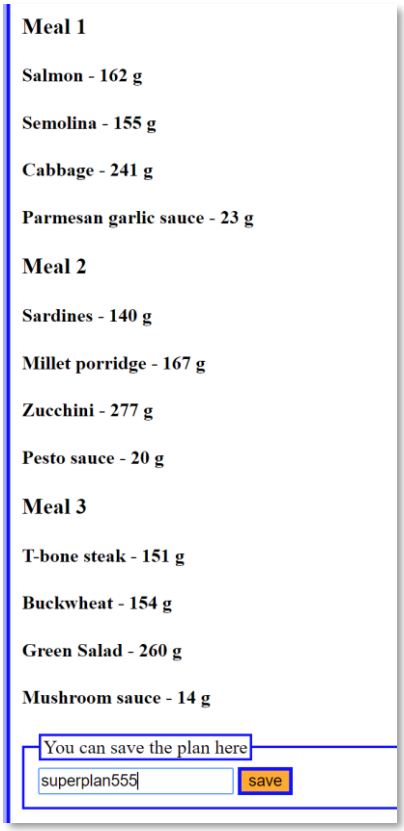
Description	Creating plan. Saving it as "superplan555". Looking how it is displayed in Django Admin. Going to the "My meal plans" in the application and seeing how the meal plan is displayed and comparing with its display, when it was first created. In addition, testing the ability to delete meal plan.
Entries	1. Creating plan • Weight = 90.0 • Calories = 2820.0 • Type of diet = Low fat diet • Number of meals = 3 • "Create" button click 2. Saving plan

	• Name of the plan = “superplan555” • “Save” button click 3. Diplaying all plans • Clicking on the orange header (coming back to main page) • “My meal plans” button click 4. Viewing the plan • Clicking “View plan” button 5. Deleting plan • “Delete” button click
Expectation	✓ In Django Administration, 1 new Plan, 3 new Meals and 12 new Meal_has_foods have to appear ✓ In the application the plan viewed in the meal plan viewer should show the same food and servings as it did in the meal planner. ✓ When the meal plan is deleted, the user should be redirected to the page, which says that there are no plans created
Actual result	All expectations have been satisfied.

Meal planner Entries:

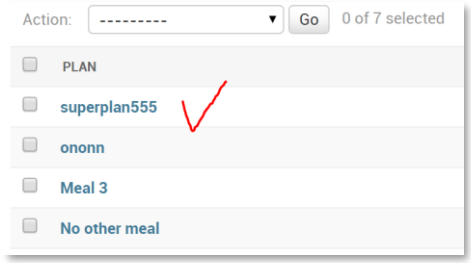


Generated Meal plan:

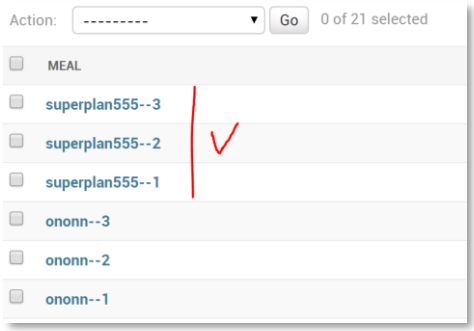


After the “Save” button have been clicked:

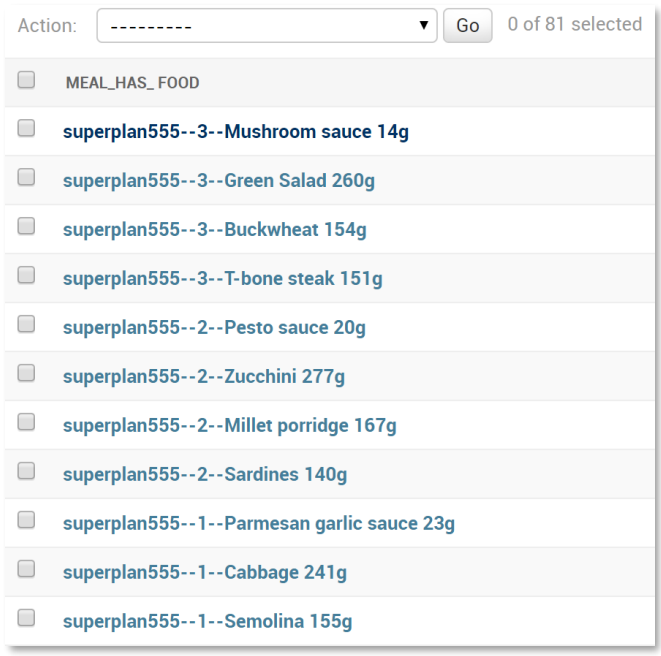
Django administration - Plans:



Django administration – Meals:



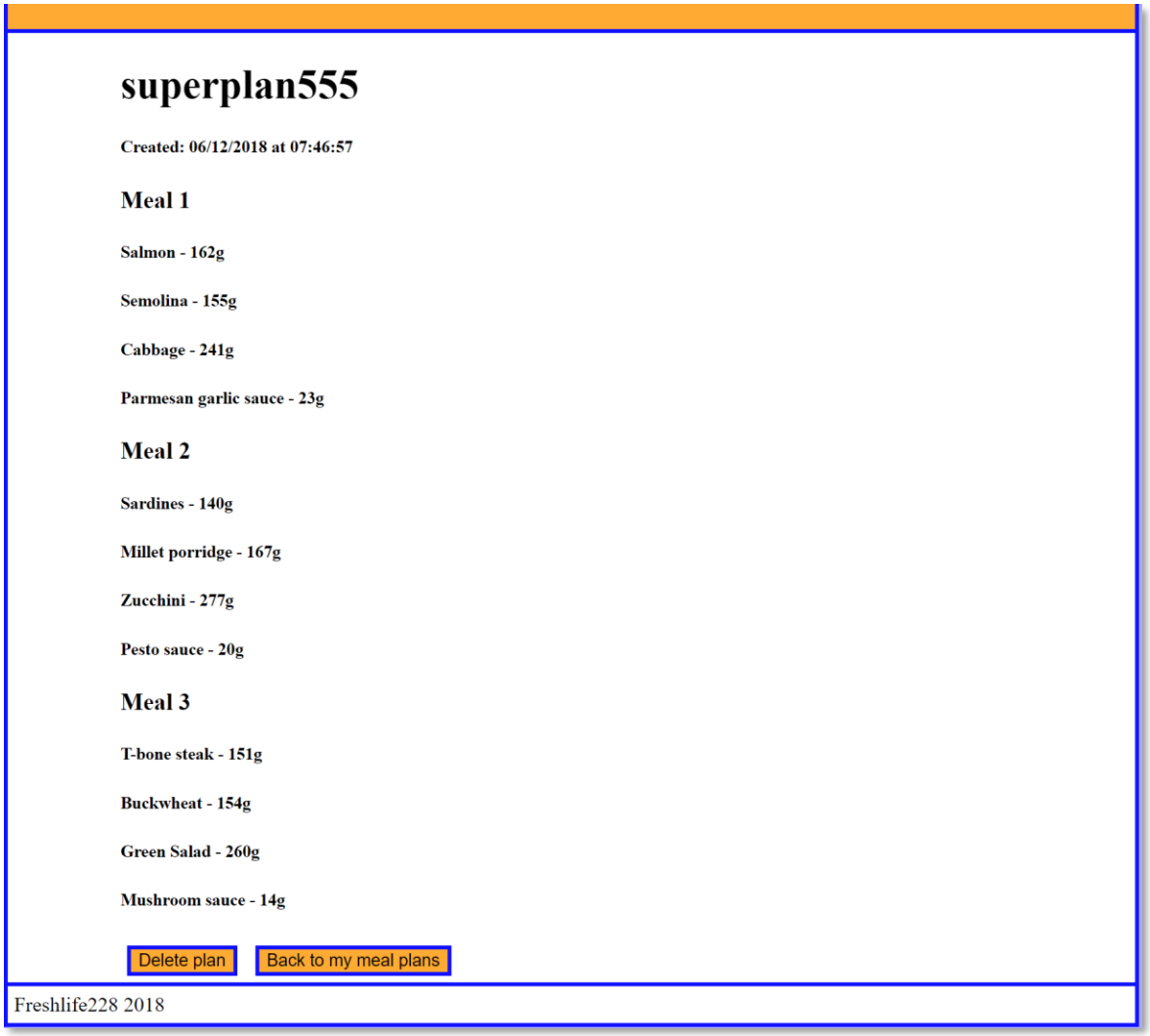
Django administration – Meal_has_Foods:



Meal and Meal_has_food have special string representation (__str__ method), which features their Plan, so it is easy to see, which of the entries relates to the test.

Meanwhile in the application, that's what happens, when "My meal plans" page is opened:

“View plan” button is clicked:



“Delete plan” has been clicked:

As a result all the expectations have been satisfied.

Evaluation

Core objective reflection

Need	Description	Importance	Pass/Fail
A. Login system	Ability to login and logout	High	Pass
With the aid of Django built in User creation form, login and logout views, users are now able to create accounts, login and logout. User has username and password, as well as other data, related to it.			
B. Calories calculator	Ability to give good approximation of number of user's daily maintenance calories based on the user inputs.	High	Pass
The application gives their users ability to calculate their calories. The Katch-McArdle's formula has been used, which is more accurate in comparison other methods. Radio button fields were useful to make the form easy to fill, so test users didn't find extimating their bodyfat ration difficult.			
C. Advanced options for calories calculator	Ability for user to choose between simple or advanced ways of calculating daily calories demand. Advanced option should allow user to fully describe his activity and get a more precise value	High	Pass
On the calculator page user can click on the checkbox and additional field-set appears. The field-set uses JavaScript to calculate its contribution. User chooses an activity and sets a number of hours he does it during a week. He can easily add and remove the notes, representing his training activity by pressing special buttons. Choosing the advanced option affects the way calories are calculated and user gets more personalised result.			
D. Daily Macronutrients	User should get the approximate value of amount of proteins, carbohydrates and fats his body needs daily	High	Pass
Calorie calculator not only calculates calories, but also proteins fats and carbohydrates and displays the values in grams per day. Meal planner is one step ahead. Calculating macronutrients is a part of its algorithms, but instead of showing the values to users, it returns user the meal plans with the values of servings of food, which takes the nutritional values of food into account, which may be much more sensible for some users.			
E. Favourite foods	Ability to mark certain foods as favourite, so nutrition planner would consider user preferences	High	Pass
User has a menu which allows to add food to favourite, which has been implemented through the Many-to-Many relationship between User and Food models.			
F. Food ignore list	Ability to add food to the ignore list (Due to allergies or taste preferences). The food in that list will not be suggested in the meal plans created by this user.	High	Pass
The same menu, "Add food", also allows to add food to ignore list, which has also been implemented through the Many-to-Many relationship between User and Food models. Food related with the user via such relationship is excluded from Meal Planner possible food suggestions.			

G. Search	Ability to search foods when user wants to mark them as favourite or ignored.	Medium	Pass
In the “Add food” page, user can search for the food he would like to mark as favourite or ignored. Server looks for food, which contains the value typed in the search textbox inside the name of the food.			
H. Suggest choice	In “add food”, the application suggest user few food options he might like to add.	Medium	Pass
In the “Add food” page, apart from search algorithm, There is a section which suggests three random foods to add to favourite. Food which is already in the Favourite or ignore list, is not suggested by this field.			
I. User-Friendly Interface	Application should be simple to work with and test users shouldn't have problems with it.	High	Pass
The application tries its best to be as simple as possible. Buttons and clickable objects have contrast colours and the text on the buttons is as explicit as possible. During the test # 8 (page 93), test users confirmed that the app is user-friendly and simple, with an exception which was fixed later on. The raised issue was the struggle to navigate back to the main menu. Coming back to the Main Page could be done by just clicking on the logo on the top of every page, but it was not obvious for some users, so additional navigation buttons were added.			
J. Nutrition Planner	Ability to create meal plans based on the values of user’s energy and macros requirements and a number of meals he would like to have per day.	High	Pass
Meal Planner page allows users to create the meal plans. Users can choose between different diets and adjust the number of meals. Meal planner has a flexible algorithm, which evaluates the servings via recursion or analytically and hence it can use the one which is better for a particular diet. With the aid of exception handling and defensive programming, Meal planner didn’t crash during the tests			
K. Favourite food only	Ability to create meal plans containing only user’s favourite food	Medium	Pass
Meal planner form has a check box, which allows to use only favourite food of the user. That might sufficiently			
L. Send user to the “Add Food”	If the user doesn’t have favourite food or its number is insufficient, application should ask user to add some food.	Medium	Pass
If the user checked the “Favourite food only” checkbox and in the same time he doesn’t have a favourite food or doesn’t have enough favourite food, i.e. in case if meal planner cannot be created, Meal planner displays a message to the user, which tells him, which type of food he should add and a quich navigation button to the “Add food” page.			
M. Save meal plans to the database	Ability to save just created meal plan to the database	High	Pass
In the Meal Planner when the plan has been created, “Save” button appears on the page. The tests are saved and can be viewed in the meal plan viewer.			
N. User meal plans	Ability to have all meal plans of the user displayed. User must to have ability to view the details of the particular plan or delete plan.	High	Pass

Every user can go to “My meal plans” and display the list of all plans he owns. User can press “View plan” to see the details of the meal plan or delete the plan.			
O. Different styles of diet	Make dietary plans in different styles of diet, for example ketogenic, paleo, low-carb.	Low	Pass
By the current moment program has three diet options: “Low fat diet”, “High protein diet” and “Low carb diet”, which cover the most common needs people on diet usually have, such as “Cut on carbs” or “Cut on fats”.			
P. Data security	User passwords should be stored as hashed values. Allowing users create only passwords with sufficient level of complexity	High	Pass
User can create an account only via the built in Django User Creation Form, which applies hashing and hence raw passwords are not stored in the system.			

Customer feedback.

The client commented as follows:

“I liked what I have seen as a final result. I had an idea and the program made it real in a pretty accurate way. I liked the way calculator was made. The body-fat chart was a good idea, because it makes estimating much easier. Meal planner does a good job, I like how it generates meals in just few clicks. I also find it good that user are not overloaded with complicated information and just get what they need to have. The application already has all the functionality and user experience and after we add more food choices and improve some visual aspects. I think it has potential and I will be happy to keep working with Andrey as soon as he finishes his A-levels.”

Further development

Plans and Ideas.

Although the initial core objectives are satisfied, the website needs more to become complete and competitive product. Some of the things which could have been implemented if more time was given are the following:

- Image fields of food. It would make the website better appearance and make clearer for users how the food should look like.
- More diet options. Expand the choice of diets meal planner can work with so users have a bigger choice. It means that meal planner algorithm should be improved so it is more flexible.
- Handling advanced nutritional properties of food, such as vitamins, minerals, fibre and sugar content and etc. Taking this things into account will make the meals generated by the Meal Planner much healthier.
- Food library, where user can read its nutritional information and healthy benefits and/or interesting facts. It will attract users who just want to learn more about food.
- Training program maker. Helping users organise their training. In addition, letting Meal Planner know the exact training program of a user will help to create most suitable nutrition plans
- Progress tracker of the customer’s fitness goal. Tracking the progress of the user’s shape and training results, so the user can see his progress
- Better looking design of the user interface.

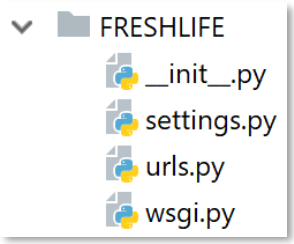
There's plenty of things which would be great extension of that is already created. The achieved result is just a beginning. A lot of work is ahead and it is hoped not to wait for long.

Conclusion.
Overall, I believe the finished program meets the client's requirements. The website does not look as professional as I would like it to be, but the website is fully functional and does everything expected of it. Given more time, I would improve its appearance and ease of use.

Appendix A (Code listing)

Note that if a file code has been shown in the directory, but the code of the file haven't been shown, then the file was auto-created or empty. Migration files in migration folders and some other file are not shown here, because they were autocreated. Files like mainApp/models.py are not shown, because it is empty (mainApp doesn't have database entities).

FRESHLIFE



Settings.py

```
"""Django settings for FRESHLIFE project...."""

import os

# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/2.1/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = '=([redacted]'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []


# Application definition

```

```
INSTALLED_APPS = [
    'mainApp',
    'calc',
    'planner',
    'myPlans',
    'food',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
            # 'builtins': [
            #     'django.contrib.staticfiles.templatetags.staticfiles',
            # ],
        },
    },
]
```

```
78  WSGI_APPLICATION = 'FRESHLIFE.wsgi.application'
79
80
81  # Database
82  # https://docs.djangoproject.com/en/2.1/ref/settings/#databases
83
84  DATABASES = {
85      'default': {
86          'ENGINE': 'django.db.backends.sqlite3',
87          'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
88      }
89  }
```

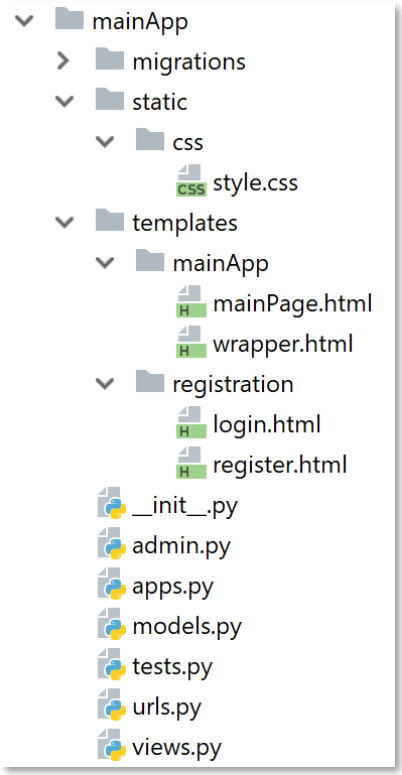
```
92  # Password validation
93  # https://docs.djangoproject.com/en/2.1/ref/settings/#auth-password-validators
94
95  AUTH_PASSWORD_VALIDATORS = [
96      {
97          'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
98      },
99      {
100         'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
101     },
102     {
103         'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
104     },
105     {
106         'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
107     },
108 ]
```

```
111 # Internationalization
112 # https://docs.djangoproject.com/en/2.1/topics/i18n/
113
114 LANGUAGE_CODE = 'en-us'
115 TIME_ZONE = 'UTC'
116 USE_I18N = True
117 USE_L10N = True
118 USE_TZ = True
119
120
121 # Static files (CSS, JavaScript, Images)
122 # https://docs.djangoproject.com/en/2.1/howto/static-files/
123
124 STATIC_URL = '/static/'
125
126 LOGIN_REDIRECT_URL = '/'
```

Urls.py

```
1    """FRESHLIFE URL Configuration..."""
16   from django.contrib import admin
17   from django.urls import path, include
18   # first level urls
19   urlpatterns = [
20       path('admin/', admin.site.urls),
21       path('', include('mainApp.urls')),
22       path('accounts/', include('django.contrib.auth.urls')),
23       path('food/', include('food.urls')),
24       path('calc/', include('calc.urls')),
25       path('planner/', include('planner.urls')),
26       path('my_plans/', include('myPlans.urls')),
27   ]
```

mainApp



Static/css/style.css

```
* {box-sizing: border-box;}
/* a {text-decoration: none;} */
body {
    background-color: #aaaaff;
}
button {
    background-color: #ffaa33;
    margin: 5px;
    border: 3px solid #1111ff;
} button:active {background-color: #cc7700;}

.ocb { /*Orange Clickable Box*/
   background-color: #ffaa33;
   border: 3px solid #1111ff;
} .ocb:active {background-color: #ee9922;}

fieldset {
    margin: 10px auto;
    border: 3px solid #1111ff;
} legend {
    border: 3px solid #1111ff;
}
```

```
#pagesheet {
    margin:0 20% 0 20%;
    min-height:100px;
    background-color: silver;
    float: center;
}
.big {
    display: block;
    clear: both;
    width:100%;
}
#header {
    border: 3px solid #1111ff;
    background-color: #ffaa33;
}
#fl228 {
    text-align: center;
    margin: 30px 0 30px 0;
    font-size: 50px;
    color: #1111ff;
}
```

```
#Freshlife {font-style: italic;}
#_228 {
    font-family: Impact, Charcoal, sans-serif;
}
#BLOCK {
    float: left;
    background-color: #ffffff;
    border: 3px solid #1111ff;
    border-top: none;
    min-height: 100px;
}
#footer {
    background-color: #ffffff;
    border: 3px solid #1111ff;
    border-top: none;
    padding: 5px;
    /*min-height: 100px;*/
}
```

Templates/mainApp/wrapper.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>{% block title %}{% endblock %}</title>
    {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}" type="text/css"/>
    {% block style %}{% endblock %} <!-- for css files or code-->
</head>
<body>
    <div id="pagesheet">
        <a style="..." href="/">
            <div class="big" id="header">
                <p id="fl228"><span id="Freshlife"> Freshlife</span><span id="_228">228</span></p>
            </div>
        </a>
        <div class="big" id="BLOCK">
            {% block content %}
            {% endblock %}
        </div>
        <div class="big" id="footer">
            Freshlife228 2018
        </div>
    </div>
</body>
</html>
```

Templates/mainApp/mainPage.html

```
{% extends "mainApp/wrapper.html" %}

{% block title %}Freshlife228 - Main Menu{% endblock %}

```

```
{% block style %}
<style>
#mainMenuTitle {
    color: #1111ff;
    font-size:20px;
    font-weight:bold;
    text-align:center;
    font-family: calibri;
    margin: 0 auto;
} #mm {margin-top: 20px;}
#buttons {
    width: 20%;
    margin: 0 auto;
}
.cent {
    display: block;
    margin: 5px auto;
    /*text-align: center;*/
}
.diva a {
    color: #1111ff;
    display: block;
}
.button {
    margin-top: 30px;
    height: 50px;
    background-color: #ffaa11;
    border: 3px solid #1111ff;
    padding: 5px;
    text-align: center;
} #logoutbutton {
    margin-bottom: 30px;
}
</style>
{% endblock %}
```

```
{% block content %}
<div id="buttons">
    <div id="mainMenuTitle">
        <p class="cent" id="mm">Main Menu</p>
    </div>
    <a class="diva" href="calc/">
        <div class="button">
            <p class="cent">Calorie Calculator</p>
        </div>
    </a>
    <a class="diva" href="planner/">
        <div class="button">
            <p class="cent">Meal Planner</p>
        </div>
    </a>
    <a class="diva" href="my_plans/">
        <div class="button">
            <p class="cent">My meal plans</p>
        </div>
    </a>
    <a class="diva" href="food/">
        <div class="button">
            <p class="cent">Add favourite food</p>
        </div>
    </a>
    <a class="diva" href="logout/">
        <div class="button" id="logoutbutton">
            <p class="cent">Logout</p>
        </div>
    </a>
</div>
{% endblock %}
```

Templates/registration/login

```
{% extends "mainApp/wrapper.html" %}
{% block title %}Freshlife228 - Login{% endblock %}
{% load staticfiles %}
{% block style %}
<style>
    #loginform {
        border: 3px solid #1111ff;
        border-top: none; border-bottom: none;
        width: 33%; margin:0 auto; padding: 20px;
    }
</style>
{% endblock %}

{% block content %}
<div id="loginform">
    <h3>Login</h3>

    {% if form.errors %}
    <p>There's something wrong with what you entered!</p>
    {% endif %}
    {% if next %}
    <p>Hey, you can't access that page</p>
    {% endif %}

    <form method="post" action="{% url 'login' %}">
        {% csrf_token %}
        <p>Username: {{ form.username }}</p>
        <p>Password: {{ form.password }}</p>
        <p><button type="submit">Login</button></p>
        <input type="hidden" name="next" value="{{ next }}">
    </form>
    <div>
        <h4> Don't have an account?</h4>
        <a href="{% url 'mainApp:register' %}">
            <button>Create account</button>
        </a>
    </div>
</div>
{% endblock %}
```

Templates/registration/register

```
{% extends "mainApp/wrapper.html" %}
{% block title %}Freshlife228 - Login{% endblock %}
{% load staticfiles %}

{% block style %}
<style>
    #registerform {
        border: 3px solid #1111ff;
        border-top: none; border-bottom: none;
        width: 50%;
        margin:0 auto; padding: 20px;
    }
    ul, .helptext {
       font-size:12px;
    }
    input {border: 1px solid #1111ff;}
</style>
{% endblock %}
```

```
{% block content %}
<div id="registerform">
    <h2>Registration page</h2>

    <form method="post" action="{% url 'mainApp:register' %}">
        {% csrf_token %}
        {% if form.errors %}
        <p>There are errors in the form!</p>
        {% endif %}
        <!--{{ form.as_p }}-->
        <p><label>Username:</label>  {{form.username}}</p>
        <p><label>Password:</label>  {{form.password1}}</p>
        <p><label>Confirm password:</label>  {{form.password2}}</p>

        <span class="helptext">
            Username must contain less than 150 characters. <br/>
            Acceptable characters are: letters, digits and @/./+/-/_ only.
        </span>
        <ul>
            <li>Your password can't be too similar to your other personal information.</li>
            <li>Your password must contain at least 8 characters.</li>
            <li>Your password can't be a commonly used password.</li>
            <li>Your password can't be entirely numeric.</li>
        </ul>
        <button type="submit">Register</button>
    </form>
</div>
{% endblock %}
```

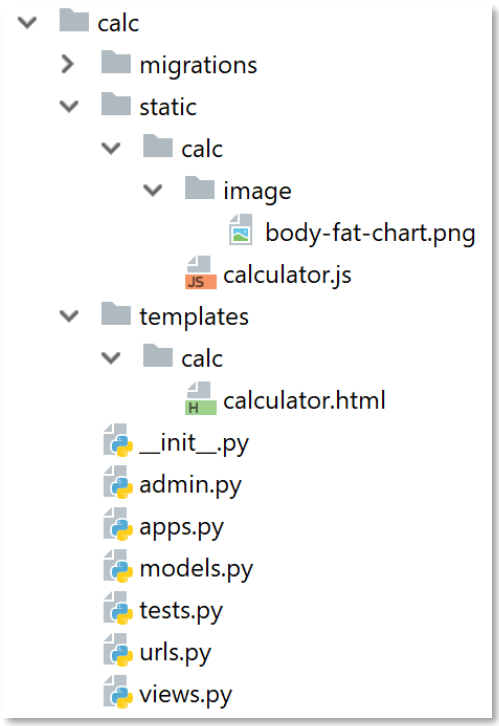
Urls.py

```python
from django.urls import path, include
from . import views

app_name = "mainApp"

urlpatterns = [
    path('', views.mainPage, name='mainPage'),
    path('register/', views.register, name='register'),
    path('logout/', views.logout_user, name='logout_user'),
]
```

Views.py

```python
from django.shortcuts import render, redirect
from django.shortcuts import HttpResponseRedirect
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import authenticate, login, logout


def mainPage(request):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    else:
        return render(request, 'mainApp/mainPage.html')


def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data['username']
            password = form.cleaned_data['password1']
            user = authenticate(username=username, password=password)
            login(request, user)
            return redirect('mainApp:mainPage')
    else:
        form = UserCreationForm()
    context = {'form': form}
    return render(request, 'registration/register.html', context)


def logout_user(request):
    logout(request)
    return HttpResponseRedirect('/accounts/login')
```

Calc



Static/calc/calculator.js

```
function removeElement(elementId) {
    var element = document.getElementById(elementId);
    element.parentNode.removeChild(element);
}
function updateSneakyInput() {
    sneakyInput = document.getElementById("sneaky");
    sneakyInput.value = EATperKg.toFixed(2); // two decimal places
}
function advancedOption(){
    var checkBox = document.getElementById("advanced_check");
    var adCalc = document.getElementById("advanced_calc");
    if (checkBox.checked == true){
        adCalc.style.display = "block";
    } else {
        adCalc.style.display = "none";
    }
}
```

```
function addAct(){
    var time = document.getElementById("hours").value; // time from the time input
    if (time){
        var select = document.getElementById("sel"); // select element
        var name = select.options[select.selectedIndex].innerHTML; // name of chosen activity
        var epkgh = select.options[select.selectedIndex].value; // value of chosen activity
        var container = document.getElementById("user_act-s"); // div where new activities created

```

```
        //creating div for new activity
        var newDiv = document.createElement("div");
        actIdCounter = actIdCounter + 1; // incrementing the identifier of the block
        newDiv.id = "act" + actIdCounter;
        //creating spans for time and epkgh, because this values must be accessible
        var timeSpan = document.createElement("span");
        timeSpan.innerHTML = time;
        var epkghSpan = document.createElement("span");
        epkghSpan.innerHTML = epkgh;
        epkghSpan.setAttribute("style", "visibility: hidden;"); // user don't need to see it
        // but it has to be stored nearby for delAct() fuction

        //creating button
        var clickme = document.createElement("button"); // creating delete button
        clickme.innerHTML = "Remove";
        clickme.type = "button";
        clickme.setAttribute("onclick", "delAct(" + actIdCounter + ")");
        // arranging div elements in order with plain text inbetween
        newDiv.appendChild(timeSpan)
        newDiv.innerHTML += " hour"; if (time != 1){newDiv.innerHTML += "s"};
        newDiv.innerHTML += " of " + name;
        newDiv.appendChild(clickme);
        newDiv.appendChild(epkghSpan);
        container.appendChild(newDiv); // adding new created div to the container div

        //calculating calories contribution
        time = parseFloat(time);
        epkgh = parseFloat(epkgh);
        EATperKg += epkgh * time;
        updateSneakyInput();
    } else { alert("Please, enter the number of hours"); }
}
```

```
function delAct(id){
    // collecting data from spans and converting to float
    var time  = document.getElementById("act" + id).getElementsByTagName("span")[0].innerHTML;
    var epkgh = document.getElementById("act" + id).getElementsByTagName("span")[1].innerHTML;
    time = parseFloat(time);
    epkgh = parseFloat(epkgh);

    EATperKg -= epkgh * time; //calculating calories contribution

    updateSneakyInput();

    removeElement("act" + id); //removing activity note
}
var actIdCounter = 0;
var EATperKg = 0;
```

Templates/calc/calculator.html

```
{% extends "mainApp/wrapper.html" %}
{% block title %}Freshlife228 - Calorie Calculator{% endblock %}
{% load staticfiles %}

{% block style %}
<style>
    #area {
        margin: 5px auto;
        width:80%;
    }
    #bfradio {
        width:13%; float:left;
        margin: 15px auto;
        }
    #bfimg {width:87%; float:left;}
    #advanced_calc {
        display: none;
    }
    #sneaky {
        display: none;
    }
    #results {
        border: 3px solid #1111ff;
        padding: 10px;
    }
    #hours {width:50px;}
</style>
{% endblock %}
```

```
{% block content %}
<div id="area">
    <h1>
        Calorie Calculator
        <a href="{% url 'mainApp:mainPage' %}">
            <button type="button" style="float:right;"> Go to Main Menu</button>
        </a>
    </h1>
    {% if calories %} <!--Display results-->
        <div id="results">
            <h4>Results</h4>
            <p>Calories: <strong>{{ calories }}</strong> kcal per day</p>
            <p>Proteins: {{ prots }} grams per day</p>
            <p>Fats: {{ fats }} grams per day</p>
            <p>Carbohydrates: {{ carbs }} grams per day</p>
        </div>
    {% endif %}
```

```
    <form method="POST">
        {% csrf_token %}
        <fieldset><!--Weight fieldset-->
            <legend>Enter your weight</legend>
            <input id="weight" type="number" step="0.01" placeholder="Weight" name="weight" required> kg
        </fieldset>
```

```
        <fieldset><!--Body fat percentage radio button multiple choice-->
            <legend>Estimate your bodyfat ratio using the chart below</legend>
            <div id="bfradio">
                <label><input type="radio" name="bodyfat" value="3.5">3-4%</label> <br/>
                <label><input type="radio" name="bodyfat" value="6.5">5-8%</label> <br/>
                <label><input type="radio" name="bodyfat" value="10" checked>8-12%</label> <br/>
                <label><input type="radio" name="bodyfat" value="16.5">13-20%</label> <br/>
                <label><input type="radio" name="bodyfat" value="25">20-30%</label> <br/>
                <label><input type="radio" name="bodyfat" value="40">30-50%</label> <br/>
                    <br><br>
                <label><input type="radio" name="bodyfat" value="12">11-13%</label> <br/>
                <label><input type="radio" name="bodyfat" value="14.5">14-15%</label> <br/>
                <label><input type="radio" name="bodyfat" value="17.5">16-19%</label> <br/>
                <label><input type="radio" name="bodyfat" value="22">20-24%</label> <br/>
                <label><input type="radio" name="bodyfat" value="27.5">25-30%</label> <br/>
                <label><input type="radio" name="bodyfat" value="40">30-50%</label> <br/>
            </div>
            <div id="bfimg">
                <img src="{% static 'calc/image/body-fat-chart.png' %}" width="100%">
            </div>
        </fieldset>
```

```
        <label><!--checkbox for advanced option-->
            <input id="advanced_check" name="adCalc"
                   type="checkbox" onclick="advancedOption()">
            Use advanced option to describe training activity in details
        </label>
        <div id="advanced_calc"><!--advanced options div. Initially it is in display:none; state-->
            <fieldset>
                <legend>Describe your typical weekly activities and workouts</legend>
                <div id="actbar">
                    During a week I usually do
                    <select id="sel">
                        {% for item in activities %}
                            <option id="opt" value="{{ item.epkgh }}">{{ item.name }}</option>
                        {% endfor %}
                    </select>
                    for <input id="hours" type="number" min="0.5" step="0.5"/> hours
                    <button type="button" onclick="addAct()">add</button>
                    <input id="sneaky" name="sneaky" type="number" value="0" step="0.01">
                </div>
                <div id="user_act-s"></div> <!-- User activities are dropped here -->
            </fieldset>
        </div>
```

```
        <fieldset><!--Multiple choice question about general activity-->
            <legend>How active are you?</legend>
            <label><input type="radio" name="activity" value="1">
                Sitting lifestyle</label> <br/>
            <label><input type="radio" name="activity" value="2">
                Low activity (Walking, housekeeping or little exersice)</label> <br/>
            <label><input type="radio" name="activity" value="3">
                Medium activity (Training 3-5 times a week)</label> <br/>
            <label><input type="radio" name="activity" value="4">
                High activity (Active lifestyle, training 6-7 times a week)</label> <br/>
            <label><input type="radio" name="activity" value="5" checked>
                Extreme activity (Train like an athlete, physical labour)</label> <br/>
        </fieldset>
```

```
        <fieldset><!--Multiple choice question about the goal-->
            <legend>What is your goal?</legend>
            <label><input type="radio" name="goal" value="0.9">Lose weight</label> <br/>
            <label><input type="radio" name="goal" value="1" checked>Preserve weight</label> <br/>
            <label><input type="radio" name="goal" value="1.1">Gain weight</label> <br/>
        </fieldset>
        <input class="ocb" name="calculate" type="submit" value="calculate"><!--submit form button-->
    </form>
</div>
<script src="{% static 'calc/calculator.js' %}"></script> <!--external scripts-->
{% endblock %}
```

Admin.py

```python
from django.contrib import admin
from .models import Activity, Latest_inputs

# registering the models from calc/models.py
admin.site.register(Activity)
admin.site.register(Latest_inputs)
```

Models.py

```python
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver


class Activity(models.Model):
    name = models.CharField(max_length=45)
    epkgh = models.FloatField()

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

```python
class Latest_inputs(models.Model):  # additional information related to user
    user = models.OneToOneField(User, on_delete=models.CASCADE, primary_key=True)
    weight = models.FloatField(null=True)
    calories = models.FloatField(null=True)

    @receiver(post_save, sender=User)
    def create_latest_inputs(sender, instance, created, **kwargs):
        if created:
            Latest_inputs.objects.create(user=instance)

    @receiver(post_save, sender=User)
    def save_latest_inputs(sender, instance, **kwargs):
        instance.latest_inputs.save()

    def __str__(self):
        return "{}'s inputs".format(self.user.username)
```

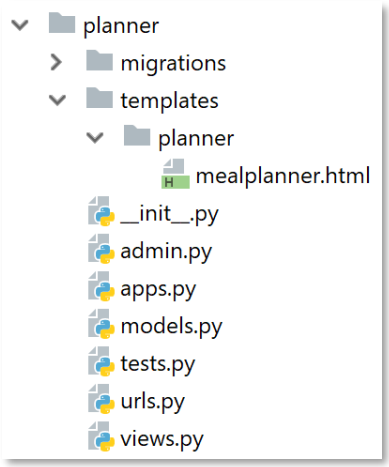
Urls.py

```python
from django.urls import path, include
from . import views

app_name = 'calc'

urlpatterns = [
    path('', views.calculator, name='calculator'),
    path('<int:user_came_from_planner>/', views.calculator, name='calculator2'),
]  # calculator2 redirects user to the meal planner instead of showing results
```

Views.py

```python
from django.shortcuts import render, redirect, HttpResponseRedirect
from .models import Activity, Latest_inputs
from django.core.exceptions import ObjectDoesNotExist


def calculator(request, user_came_from_planner=0):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    activities_list = Activity.objects.all().order_by("name")  # taking activities from db
    calories = prots = fats = carbs = 0  # context vars
    if request.method == "POST":
        weight = float(request.POST.get('weight'))
        bodyfat = float(request.POST.get('bodyfat'))
        activity_choice = int(request.POST.get('activity'))
        goal = float(request.POST.get('goal'))
        advanced_checkbox = request.POST.get('adCalc')

        bmr = 370 + 21.6 * weight * (100 - bodyfat) / 100
        if advanced_checkbox:
            eat_per_kg = float(request.POST.get('sneaky'))  # hidden input in calculator.html
            calories += int(eat_per_kg * weight / 7)  # Exercise Energy Expenditure is added here
            common_difference = 0.05  # (1.15, 1.20. 1.25, ...)
        else:
            common_difference = 0.15  # (1.15, 1.30, 1.45, ...)
        # arithmetic sequence instead of "if" clause for determining activity coeficient
        activity_coef = 1.15 + common_difference * (activity_choice - 1)

        calories += int(bmr * activity_coef * goal)
        prots = int(weight * 1.5)  # Proteins
        fats = int(weight * 0.9)
        carbs = int((calories - prots * 4 - fats * 9) / 4)  # Carbohydrates
        # ----------------------------------------------------------------------
        # Updating User's Latest_inputs
        user = request.user
        try:  # plan a, if everything is OK
            if user.latest_inputs:
                user.latest_inputs.weight = weight
                user.latest_inputs.calories = calories
                user.save()
        except ObjectDoesNotExist:  # plan b, if User "lost" his latest_inputs
            Latest_inputs(user=user, weight=weight, calories=calories).save()
        # ----------------------------------------------------------------------
        if user_came_from_planner:  # coming back to meal planner
            return redirect('planner:meal_planner')
    context = {
        'calories': calories,
        'prots': prots,
        'fats': fats,
        'carbs': carbs,
        'activities': activities_list,
    }
    return render(request, 'calc/calculator.html', context)
```

Planner



Templates/planner/mealplanner.html

```
{% extends "mainApp/wrapper.html" %}
    {% load staticfiles %}
{% block title %}Freshlife228 - Meal planner{% endblock %}

{% block style %}
<style>
    a {text-decoration: none;}
    #setup {
        width:85%; margin:auto;
    }
    .setting{margin:5px; display:inline;}
    #meal_display {
        border-top: 3px solid #1111ff;
        padding: 10px;
    }
    .stealth {
        display: none;
    }
</style>
{% endblock %}
```

```
{% block content %}
<div id="setup">
    <h1>
        Meal Planner
        <a href="{% url 'mainApp:mainPage' %}">
            <button type="button" style="float:right;"> Go to Main Menu</button>
        </a>
    </h1>
    <form method="post" style="..."> <!--form for meal details-->
        {% csrf_token %}
        <fieldset> <!--Weight-->
            <legend>Weight</legend>
            <input name="weight" type="number" value="{{ user.latest_inputs.weight }}"
                   min="1" step="0.01" placeholder="weight" required> Kg
        </fieldset>
```

```
        <fieldset> <!--values of weight and calories have auto fill (latest_inputs)-->
            <legend>Daily calories intake</legend>
            <input name="calories" type="number"
                   value="{{ user.latest_inputs.calories }}"
                   min="0" required>
            <a href="{% url 'calc:calculator2' user_came_from_planner=1 %}">
                <button type="button">Use calculator to fill the field</button>
            </a> or enter the value manualy
        </fieldset>
        <fieldset id="settings"><!--Diet entries-->
            <legend>Diet settings</legend>
            <div class="setting"><!--Types of diet-->
                Choice of diet:
                <select name="diet"><!--Choice between three diets-->
                    <option value="[1.5, 0.9, 0]">Low fat diet</option>
                    <option value="[2, 0, 1.3]">Low carb diet</option>
                    <option value="[2.0, 1.2, 0]">High protein diet</option>
                </select>
            </div>
            <div class="setting"><!--Number of meals entry. Default: 3-->
                Number of Meals:
                <input type="number" name="meal_num" value="3" min="1" max="9"> <!--number of meals-->
            </div>
            <div class="setting"><!-- Favourite food checkbox here-->
                <label><input type="checkbox" name="fav_checkbox">
                    Use favourite food only</label>
            </div>
        </fieldset>
        <input type="submit" name="create" class="ocb" value="Create meal plan"> <!--CREATE button-->
    </form>
</div>
```

```
<div id="meal_display">
    {% if not enough_favourites %}
        <p>Unfortunately, meal plan cannot be created.</p>
        <p>{{ reasons }}Please, add more food to your favourites.</p>
        <a href="{% url 'food:addfood' %}">
            <button type="button">Add favourite food</button> <!--Button to the Add Food-->
        </a>
```

```
    {% elif plan %}
    <form method="post"> <!--Form with hidden inputs which are filled by the server.-->
        <!--It contains all the information for the DB to be saved. User just presses 'save'-->
        {% csrf_token %}
            <!-- hidden information about plan -->
            <input class="stealth" name="meal_num" type="number" value="{{ plan|length }}">
            <input class="stealth" name="three_or_four" type="number" value="{{ plan.1|length }}">
```

```
            {% for meal in plan %} <!-- display the meals in plan-->
                <div class="meal_div">
                    <h3>Meal {{ forloop.counter }}</h3>
                    {% with outer=forloop %}
                        {% for food in meal %} <!-- display food in the meal -->
                            <h4>{{ food.0 }} - {{ food.1 }} {{ food.2 }}</h4>
                            <input class="stealth"
                                   name="food_{{outer.counter}}{{forloop.counter}}"
                                   type="text"
                                   value="{{ food.0 }}"><!--hidden input of the food name-->
                            <input class="stealth"
                                   name="serving_{{outer.counter}}{{forloop.counter}}"
                                   type="number"
                                   value="{{ food.1 }}"><!--hidden input of the food serving size-->
                        {% endfor %}
                    {% endwith %}
                </div>
            {% endfor %}
```

```
        <fieldset> <!-- visible part of the form -->
            <legend>You can save the plan here</legend>
            <input type="text" name="plan_name" placeholder="Name of the meal plan" required>
            <input type="submit" name="save" class="ocb" value="save"> <!--SAVE button-->
        </fieldset>
    </form>
    {% endif %}
</div>
{% endblock %}
```

Admin.py

```
from django.contrib import admin
from . import models
# Registering models from planner/models.py
admin.site.register(models.Plan)
admin.site.register(models.Meal)
admin.site.register(models.Meal_has_Food)
```

Models.py

```
from django.db import models
from django.contrib.auth.models import User
from food.models import Food
from datetime import datetime


class Plan(models.Model):
    name = models.CharField(max_length=45)
    owner = models.ForeignKey(User, on_delete=models.CASCADE)
    date = models.DateTimeField(default=datetime.now)  # when created

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

```python
class Meal(models.Model):
    index = models.IntegerField(null=True)  # order number in the plan
    plan = models.ForeignKey(Plan, on_delete=models.CASCADE)
    food = models.ManyToManyField(Food, through='Meal_has_food')

    def __str__(self):
        return "{}--{}".format(self.plan, self.index)


class Meal_has_Food(models.Model):  # intermediary model (junction table)
    meal = models.ForeignKey(Meal, on_delete=models.CASCADE)
    food = models.ForeignKey(Food, on_delete=models.CASCADE)
    portion_size = models.IntegerField()

    def __str__(self):  # str help work with normalized data
        if self.food.unit == "null":
            unit = ""
        else:
            unit = self.food.unit
        return "{}--{} {}{}".format(self.meal, self.food,
                                    self.portion_size, unit)
        # example str: "TESTPLAN--3--Tomato sauce 30g"
```

Urls.py

```python
from django.urls import path, include
from . import views

app_name = "planner"

urlpatterns = [
    path('', views.meal_planner, name='meal_planner')
]
```

Views.py

```python
from django.shortcuts import render
from django.http import HttpResponseRedirect
from food.models import Food
from .models import Plan, Meal, Meal_has_Food
from random import randint
import ast
from numpy import array
from scipy.optimize import lsq_linear

```

```
10
11  def serving_master(
12          prots, fats, carbs,
13          calories, healthscore,
14          Main, Side, Veg, Other=None
15  ):
16      # Measures to make nutrients non-negative
17      if prots < 0: prots = 0
18      if fats < 0: fats = 0
19      if carbs < 0: carbs = 0
20      if calories < 0: calories = 0
21      if healthscore < 0: healthscore = 0
22
```

```
23      # Making sure certain percentage of nutritional value comes from
24      # the right source, and calibrating nutrients after each assignment
25      Veg_s = 0.6 * healthscore / Veg.healthscore
26      prots -= Veg.protein * Veg_s
27      carbs -= Veg.carbs * Veg_s
28      fats -= Veg.fat * Veg_s
29      healthscore -= Veg.healthscore * Veg_s
30
31      Main_s = 0.6 * prots / Main.protein + 0.05 * fats / Main.fat
32      prots -= Main.protein * Main_s
33      carbs -= Main.carbs * Main_s
34      fats -= Main.fat * Main_s
35      healthscore -= Main.healthscore * Main_s
36
37      Side_s = 0.7 * carbs / Side.carbs
38      prots -= Side.protein * Side_s
39      carbs -= Side.carbs * Side_s
40      fats -= Side.fat * Side_s
41      healthscore -= Side.healthscore * Side_s
42
```

```
43      # calibrating calories
44      calories -= Main.calories * Main_s +\
45          Side.calories * Side_s + Veg.calories * Veg_s
46
47      if Other:  # Other may not be in the meal
48          Other_s = 0.05 * calories / Other.calories
49          prots -= Other.protein * Other_s
50          carbs -= Other.carbs * Other_s
51          fats -= Other.fat * Other_s
52          healthscore -= Other.healthscore * Other_s
53          calories -= Other.calories * Other_s
54      else:
55          Other_s = 0
56
```

```
    # print(prots,"|", fats, "|", carbs, "|", calories, "|", healthscore)
    if calories > 200 or prots > 10 or carbs > 10 or fats > 25:
        # if actual macronutrients values are not close enough
        # to the expected values, another iteration is executed
        print("ONE MORE ITERATION!")
        extra_servings = serving_master(
            prots, fats, carbs,
            calories, healthscore,
            Main, Side, Veg, Other,
        )
        Main_s += extra_servings[0]
        Side_s += extra_servings[1]
        Veg_s += extra_servings[2]
        Other_s += extra_servings[3]
    return Main_s, Side_s, Veg_s, Other_s  # servings of each category
```

```
def meal_planner(request):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    enough_favourites = True
    reasons = ""  # reason why not enough favourites, message for user
    plan = []  # list, where all meals lists would go
    meal_num = 0  # some context vars
    if request.method == 'POST' and 'create' in request.POST:  # "create" button was clicked
        # getting data from the form and converting to the right data types
        weight = float(request.POST.get('weight'))
        calories = float(request.POST.get('calories'))
        diet = ast.literal_eval(request.POST.get('diet'))  # string representation of a list --> list
        meal_num = int(request.POST.get('meal_num'))

```

```
        # Calculating daily macronutrients
        prots = diet[0] * weight
        fats = diet[1] * weight
        carbs = diet[2] * weight
        if not fats:  # one of the macros is always not fixed
            fats = (calories - prots * 4 - carbs * 4) / 9
        elif not carbs:
            carbs = (calories - prots * 4 - fats * 9) / 4

        # making a queryset of the acceptable food choices and making sure it's sufficient
        ignored = list(request.user.ignored_food.all())  # ignored food
        choices = Food.objects.all().exclude(name__in=ignored)

        Main_choices = choices.filter(type="Main")  # splitting shoices in categories
        Side_choices = choices.filter(type="Side")
        Veg_choices = choices.filter(type="Vegetables")
        Other_choices = choices.filter(type="Other")

```

```
        # Narrowing choices of some categories depending on the diet.
        use_least_squares = False
        if carbs / weight < 3:  # measure for low carb diets
            Side_choices = Side_choices.filter(carbs__lt=25)  # lt --> less than
        else:
            Side_choices = Side_choices.filter(carbs__gte=25)  # gte --> more or equal
        if fats / weight >= 2:  # measure for fat-rich diets
            Main_choices = Main_choices.filter(fat__gte=30)
            Veg_choices = Veg_choices.filter(fat__gte=8)
            use_least_squares = True  # recursion cannot solve this type of diet
        else:
            Main_choices = Main_choices.filter(fat__lt=30)
            Veg_choices = Veg_choices.filter(fat__lt=8)

```

```
        # if "favourite food only" was checked only favourite foods are taken
        if request.POST.get('fav_checkbox'):  # but there are some requirements
            favourites = list(request.user.favourite_food.all())
            Main_choices = Main_choices.filter(name__in=favourites)
            Side_choices = Side_choices.filter(name__in=favourites)
            Veg_choices = Veg_choices.filter(name__in=favourites)
            Other_choices = Other_choices.filter(name__in=favourites)
            # checking if all requirements are satisfied
            if len(Main_choices) < 2:  # two or more Mains required
                reasons += "Not enough main course options. "
            if len(Side_choices) < 2:  # two or more Sides required
                reasons += "Not enough side dish (garnish) options. "
            if not Veg_choices:  # at least one source of vegetables required
                reasons += "There's no vegetable or leafy greens options. "
            # If any problems detected, user's favourites are insufficient
            if reasons:
                enough_favourites = False

        # organising meals
        if enough_favourites:  # unless user checked the checkbox AND his favourites are insufficient

            # Amount of nutrients per one meal
            prots = prots / meal_num
            fats = fats / meal_num
            carbs = carbs / meal_num
            calories = calories / meal_num  # energy per meal
            healthscore = 40 / meal_num

            # main for loop, where servings of each meal are
            for i in range(meal_num):
                # taking random item from each category
                Main = Main_choices[randint(0, len(Main_choices) - 1)]
                Side = Side_choices[randint(0, len(Side_choices) - 1)]
                Veg = Veg_choices[randint(0, len(Veg_choices) - 1)]
                if Other_choices:  # Other is different, it doesn't have to be in a plan
                    Other = Other_choices[randint(0, len(Other_choices) - 1)]
                else:
                    Other = 0

                #  trying to use recursion
                recursion_has_failed = False
                if not use_least_squares:
                    try:
                        servings = serving_master(
                            prots, fats, carbs,
                            calories, healthscore,
                            Main, Side, Veg, Other,
                        )
                    except RecursionError:
                        recursion_has_failed = True
```

```
171                 #  if recursion failed or even hasn't been attempted
172                 if use_least_squares or recursion_has_failed:
173                     # print("Using Least Squares")
174                     if Other != 0:
175                         Other_p = Other.protein
176                         Other_f = Other.carbs
177                         Other_c = Other.fat
178                         Other_h = Other.healthscore
179                         Other_kcal = Other.calories
180                     else:
181                         Other = Other_p = Other_f = Other_c = Other_h = Other_kcal = 0
182 
183                     A = array([  # content of nutrients in food
184                         [Main.protein,     Side.protein,     Veg.protein,     Other_p],
185                         [Main.carbs,       Side.carbs,       Veg.carbs,       Other_c],
186                         [Main.fat,         Side.fat,         Veg.fat,         Other_f],
187                         [Main.healthscore, Side.healthscore, Veg.healthscore, Other_h],
188                         [Main.calories,    Side.calories,    Veg.calories,    Other_kcal],
189                     ])
190                     b = array([  # requirements column vector
191                         prots,
192                         carbs,
193                         fats,
194                         healthscore,
195                         calories,
196                     ])
197                     servings = lsq_linear(A, b, bounds=(0.17, 4)).x.tolist()
198                     # print("r =", servings)
199 
```

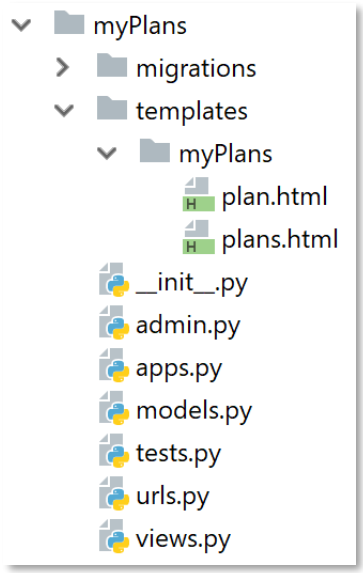
Line 197: lsq_linear retutns a lot of things and “x” is solution attribute. It has numpy.array() type, so it is converted to the list via tolist()

```
200                 Main_s = int(servings[0] * 100)  # servings converted to grams or ml
201                 Side_s = int(servings[1] * 100)  # and rounded to the integer
202                 Veg_s = int(servings[2] * 100)
203                 Other_s = int(servings[3] * 100)
204                 # Organising food choices, servings and units in a list
205                 meal = [
206                     [Main, Main_s, Main.unit],
207                     [Side, Side_s, Side.unit],
208                     [Veg, Veg_s, Veg.unit]
209                 ]
210                 if Other_s and Other:
211                     meal.append([Other, Other_s, Other.unit])
212                 plan.append(meal)  # appending meal list to the plan list
213 
```

```
214     if request.method == 'POST' and 'save' in request.POST:  # "save" button was clicked
215
216         new_plan = Plan(  # creating plan with name provided by user
217             owner=request.user,
218             name=request.POST.get('plan_name'),  # user input
219         )
220         new_plan.save()
221         #  nested looping for, there meal number and food "indecies", m and f, start with 1, not 0
222         for m in range(1, int(request.POST.get('meal_num')) + 1):
223             meal = Meal(plan=new_plan, index=m)  # creating m-th meal
224             meal.save()
225             for f in range(1, int(request.POST.get('three_or_four')) + 1):
226                 food_name = request.POST.get('food_{}{}'.format(m, f))  # finding
227                 serving = int(request.POST.get('serving_{}{}'.format(m, f)))
228                 Meal_has_Food(  # linking f-th food to m-th meal
229                     meal=meal,
230                     food=Food.objects.get(name=food_name),
231                     portion_size=serving,
232                 ).save()
233
```

```
234     context = {  # sending the data and logic to the mealplanner.html
235         "enough_favourites": enough_favourites,
236         "reasons": reasons,
237         "plan": plan,
238         "meal_num": meal_num,
239     }
240     return render(request, 'planner/mealplanner.html', context)
241
```

myPlans



Templates/myPlans/plan.html

```
{% extends "mainApp/wrapper.html" %}
{% block title %}{{ user.username }}'s {{plan.name}}{% endblock %}

{% block style %}
<style>
    a {text-decoration:none;}
    #plan {
        width: 80%;
        margin: auto;
    }
</style>
{% endblock %}
```

```

{% block content %}
<div id="plan">
    <h1>{{plan.name}}</h1>
    <h5>Created: {{plan.date|date:"d/m/Y"}} at {{plan.date|date:"h:i:s"}}</h5>
    {% for meal in meals %}
        <h3>Meal {{meal.index}}</h3>
        {% for ingridient in meal.meal_has_food_set.all %}
            <h5>{{ingridient.food.name}} - {{ingridient.portion_size}}{{ingridient.food.unit}} </h5>
        {% endfor %}
    {% endfor %}
    <a href="{% url 'myPlans:delete' pk=plan.pk %}">
        <button>Delete plan</button>  <!--Delete button-->
    </a>
    <a href="{% url 'myPlans:plans' %}">
        <button>Back to my meal plans</button>
    </a>
</div>
{% endblock %}
```

Templates/myPlans/plans.html

```
{% extends "mainApp/wrapper.html" %}
{% block title %}{{ user.username }}'s plans{% endblock %}

{% block style %}
<style>
    a {text-decoration:none;}
    .header3{text-decoration: underline;}
    h1 {text-align:center;}
    #plans {
        width: 80%;
        margin: auto; padding:;
    }
    .plan_in_list {
        border: 2px solid #11f;
        margin: 5px;
        padding: 0 20px;
    }
</style>
{% endblock %}

```

```
{% block content %}
<div id="plans">
    <h1>My meal plans</h1>
    {% if plans %}
        {% for plan in plans %}
            <div class="plan_in_list">
                <a class="header3" href="{% url 'myPlans:plan' index=forloop.counter %}">
                    <h3>{{plan.name}} </h3>
                </a>
                <h5>Created: {{plan.date|date:"d/m/Y"}} at {{plan.date|date:"h:i:s"}}</h5>
                <a href="{% url 'myPlans:plan' index=forloop.counter %}">
                    <button>View plan</button> <!--View button-->
                </a>
                <a href="{% url 'myPlans:delete' pk=plan.pk %}">
                    <button>Delete</button>  <!--Delete button-->
                </a>
            </div>
        {% endfor %}
    {% else %}
        <h3>There's no meal plans created</h3>
    {% endif %}
    <a href="{% url 'mainApp:mainPage' %}">
        <button>Back</button>
    </a>
</div>
{% endblock %}
```

Urls.py

```
from django.urls import path
from . import views

app_name = 'myPlans'

urlpatterns = [
    path('', views.user_plans, name='plans'),
    path('<int:index>', views.user_plan, name='plan'),
    path('delete/<int:pk>', views.delete_plan, name='delete'),
]
```

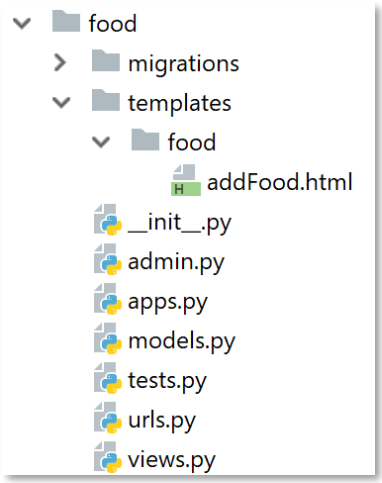
Views.py

```python
from django.shortcuts import render, redirect
from django.http import HttpResponseRedirect
from planner.models import Plan


def user_plans(request):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    # user plans from latest to oldest
    plans = request.user.plan_set.all().order_by("-date")
    context = {
        'plans': plans,
    }
    return render(request, "myPlans/plans.html", context)
```

```python

def user_plan(request, index):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    # taking one element from user plans (from latest to oldest)
    # 1 is subtracted from index, because it starts from 0, while
    # forloop counter starts from 1
    plan = request.user.plan_set.all().order_by("-date")[index - 1]
    context = {
        "plan": plan,
        "meals": plan.meal_set.all().order_by("index")
    }
    return render(request, "myPlans/plan.html", context)
```

```python

def delete_plan(request, pk):
    Plan.objects.get(pk=pk).delete()  # finding plan by pk and deleting it
    return redirect("myPlans:plans")
    # return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
```

Food



Templates/food/addFood.html

```
{% extends "mainApp/wrapper.html" %}
{% block title %}Freshlife228 - Add favourite food{% endblock %}

{% block style %}
<style>
    a {text-decoration:none;}
    #suggestblock {
        padding:0 15px;
        min-height:50px;
        border-bottom:3px solid #1111ff;
    } #suggestions {
        width: auto; max-width:60%;
        margin:0 auto 15px auto;
        display: table;
    } .suggested_item {
        color: #fff; padding: 5px;
        display: inline-block;
        min-height:40px;
        background-color:#aaaaff;
        border:3px solid #1111ff;
        margin:5px;
    }
```

```
    #searchblock {min-height:50px;}
    #searchbar {
        min-height:33px; padding: 0px;
        background-color:#aaaaff;
        border-bottom:3px solid #1111ff;
    } .nav_button {
        float:right;
        height:36px;
        margin: -3px 3px;
    }
    #inner-searchblock {padding:5px;}
</style>
{% endblock %}
```

```
{% block content %}
<div id="suggestblock">
    {% if suggest_list %}  <!--if there are suggestions-->
    <h2>Do you like any of the following foods?</h2>
    <div id="suggestions">
        {% for item in suggest_list %} <!--list of suggestions-->
            <div class="suggested_item">
                {{item.name}} <br/>
                <a href="{% url 'food:add_to_favourite' pk=item.pk %}">
                    <button type="button">Add to Favourite</button>
                </a>
            </div>
        {% endfor %}
    </div>
    {% endif %}
</div>
```

```
<div id="searchblock">
    <div id="searchbar"><!--search bar-->
        <form style="width:50%;float:left; margin: 4px;" method="GET">
            <input type="text" name="query"
                   placeholder="Search food"
                   value="{{ request.GET.query }}"/>
            <input type="submit" value="Search"/>
        </form>
        <div style="width:48%;float:left;"> <!--navigation buttons div-->
            <a href="/"><!--Main menu button-->
                <button class="nav_button" type="button">To home page</button>
            </a>
            <a href="/planner"><!--Meal planner button-->
                <button class="nav_button" type="button">Create meal plan</button>
            </a>
        </div> <!--end of new navigation buttons div-->
    </div>
```

```
    <div id="inner-searchblock"><!--Display searched elements-->
        {% if query%}
        <h4>Search results for "{{ query }}"</h4>
        {% endif %}
        {% for item in query_list %} <!--food items-->
            <h5><!--food name, favourites button, ignore list button.-->
                {{item.name}}
                {% if user not in item.loved_by.all %} <!--ability to add favourite-->
                    <a href="{% url 'food:add_to_favourite' pk=item.pk %}">
                        <button type="button">Add to Favourite</button>
                    </a>
                {% else %} <!--... or remove from favourite if already added-->
                    <a href="{% url 'food:remove_from_favourite' pk=item.pk %}">
                        <button type="button">Remove from Favourite</button>
                    </a>
                {% endif %} <!-- end of the first button-->
                {% if user not in item.ignored_by.all %} <!--ability to add ignore list-->
                    <a href="{% url 'food:add_to_ignore' pk=item.pk %}">
                        <button type="button">Ignore in my meal plans</button>
                    </a>
                {% else %} <!--... or remove from ignore list if already added-->
                    <a href="{% url 'food:remove_from_ignore' pk=item.pk %}">
                        <button type="button">Stop ignoring</button>
                    </a>
                {% endif %}
            </h5>
        {% endfor %}
    </div>
</div>
{% endblock %}
```

Admin.py

```
from django.contrib import admin
from food.models import Food

admin.site.register(Food)
```

Models.py

```python
from django.db import models
from django.contrib.auth.models import User


class Food(models.Model):
    name = models.CharField(max_length=45)
    carbs = models.IntegerField()
    sugars = models.IntegerField()  # not used yet
    fibre = models.IntegerField()  # not used yet
    protein = models.IntegerField()
    fat = models.IntegerField()
    calories = models.IntegerField()
    gi = models.IntegerField()  # not used yet
    healthscore = models.IntegerField()
    type = models.CharField(max_length=45)
    unit = models.CharField(max_length=45, default="g")

    loved_by = models.ManyToManyField(User,
                                      blank=True,
                                      related_name="favourite_food")
    ignored_by = models.ManyToManyField(User,
                                        blank=True,
                                        related_name="ignored_food")

    def __str__(self):
        return self.name

    def __repr__(self):
        return self.name
```

Urls.py

```python
from django.urls import path
from . import views

app_name = 'food'

urlpatterns = [
    path('', views.addfood, name='addfood'),
    path('add_to_favourite/<int:pk>', views.add_to_favourite, name='add_to_favourite'),
    path('remove_from_favourite/<int:pk>', views.remove_from_favourite, name='remove_from_favourite'),
    path('add_to_ignore/<int:pk>', views.add_to_ignore, name='add_to_ignore'),
    path('remove_from_ignore/<int:pk>', views.remove_from_ignore, name='remove_from_ignore'),
]
```

Views.py

```python
from random import randint


def addfood(request):
    if not request.user.is_authenticated:
        return HttpResponseRedirect('/accounts/login')
    # Suggestion block part
    suggest_list = []
    possible_suggestions = Food.objects.all()  # query
    already_favored_list = list(request.user.favourite_food.all())
    ignore_list = list(request.user.ignored_food.all())
    possible_suggestions = possible_suggestions.exclude(name__in=already_favored_list)
    possible_suggestions = possible_suggestions.exclude(name__in=ignore_list)

```

```
    length = len(possible_suggestions)  # length of narrowed list
    if length:
        randindex = randint(1, length)  # old randpk
        if length > 3:
            number_of_suggestions = 3
            r = randint(1, length - 1)  # just a random integer
        else:
            number_of_suggestions = length  # two or less items to add
            r = 1  # not that random anymore, due to empty randrange error when length < 1
        for i in range(number_of_suggestions):
            suggest_list.append(
                possible_suggestions[(randindex + r * i) % length]
            )
```

```
    # Search block part
    queryset_list = Food.objects.all().order_by("name")
    query = request.GET.get('query')
    if query:
        queryset_list = queryset_list.filter(name__icontains=query)
    # Final part of the view
    context = {
        "suggest_list": suggest_list,
        "query": query,
        "query_list": queryset_list
    }
    return render(request, 'food/addFood.html', context)
```

```

def add_to_favourite(request, pk):
    food = Food.objects.get(pk=pk)
    food.loved_by.add(request.user)
    return HttpResponseRedirect(request.META.get('HTTP_REFERER'))

```

```

def remove_from_favourite(request, pk):
    user = request.user
    food = Food.objects.get(pk=pk)
    if user in food.loved_by.all():
        food.loved_by.remove(user)
    return HttpResponseRedirect(request.META.get('HTTP_REFERER'))

```

```

def add_to_ignore(request, pk):
    food = Food.objects.get(pk=pk)
    food.ignored_by.add(request.user)
    return HttpResponseRedirect(request.META.get('HTTP_REFERER'))

```

```
64
65  def remove_from_ignore(request, pk):
66      food = Food.objects.get(pk=pk)
67      user = request.user
68      if user in food.ignored_by.all():
69          food.ignored_by.remove(user)
70      return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
```

Appendix B (Bibliography).

- Django documentation.
  - https://docs.djangoproject.com/en/2.1/
  - Django models
    - https://docs.djangoproject.com/en/2.1/topics/db/models/

- HTML, CSS, JavaScript
  - W3schools
    - https://www.w3schools.com/
  - Ruseller
    - https://ruseller.com/
  - Bootstrap. HTML, CSS, and JS library.
    - https://getbootstrap.com/

- Template language Jinja2
  - http://jinja.pocoo.org/docs/2.10/

- Python Numpy library
  - http://www.numpy.org/
  - Linear Algebra module (Linalg)
    - https://docs.scipy.org/doc/numpy/reference/routines.linalg.html
  - Array
    - https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html#numpy.array

- Python Scipy library
  - https://www.scipy.org/

- Pyhton Scipy.optimize
  - https://docs.scipy.org/doc/scipy/reference/optimize.html
  - Least Square algorithm (lsq_linear)
    - https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.lsq_linear.html
    - Source code: https://github.com/scipy/scipy/blob/v0.18.1/scipy/optimize/_lsq/lsq_linear.py#L36-L317
  - Non-Negative Least Squares (nnls)
    - https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.nnls.html

- Python Ast library (Abstract Syntax Trees)
  - https://docs.python.org/2/library/ast.html
  - Literal_eval
    - https://docs.python.org/2/library/ast.html#ast-helpers
- Wikipedia
  - Database Normalisation
    - https://en.wikipedia.org/wiki/Database_normalization
  - Least Squares
    - https://en.wikipedia.org/wiki/Least_squares
  - Non-Negative Least Squares
    - https://en.wikipedia.org/wiki/Non-negative_least_squares

- Useful Python tutorials
  - Derek Banas
    - https://www.youtube.com/watch?v=N4mEzFDjqtA
  - Corey Shafer
    - https://www.youtube.com/watch?v=YYXdXT2l-Gg&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU

- Useful Python Django tutorials
  - CodingEnterpreneurs
    - https://www.youtube.com/watch?v=uu98pqiUJU8&list=PLEsfXFp6DpzTD1BD1aWNxS2Ep06vIkaeW
  - Pretty Printed
    - https://www.youtube.com/watch?v=QVX-etwgvJ8&list=PLXmMXHVSvS-DQfOsQdXkzEZyD0Vei7PKf
  - Max Goodridge
    - https://www.youtube.com/watch?v=Fc2O3_2kax8&list=PLw02n0FEB3E3VSHjyYMcFadtQORvl1Ssj

- Useful JavaScript tutorial
  - The Net Ninja
    - https://www.youtube.com/watch?v=qoSksQ4s_hg&list=PL4cUxeGkcC9i9Ae2D9Ee1RvylH38dKuET