1. Creating an virtual enviroment (git bash)

* Command creating env: python -m venv enviroment\_name
* Activating virutal enviroment using command: source env/Scripts/activate
* To see all installed packages we use command: pip freeze
* To deactivate virtual env we use command: deactivate

1. Intsaling Django in venv (git bash; Django version 4.2.2)

* To install Django we use command: pip install django
* To create django project: django-admin startproject greatkart . (with the dot in the end)
* To activate django server: python manage.py runserver

1. Running http respone (visual studio code)

* Before changing anything in the files, we need to change python intepreter in VSC to the one created as virtual enviroment.
* Adding to file urls.py one more url link and importing file views
* Creating folder views.py in which is placed the template to the site from bootstrap.

1. Importing bootstrap templates

* Importing template files to main folder
* Creating folder templates in main project folder, in which is created a file named home.html
* Changes in the folder greatkart, file settings.py. Setting the variable TEMPLATES argument DIRS to ‘templates’ incializing the previously created folder
* Copying the code from index.html to home.html
* Creating folder named static and moving to it all folders from template (css, fonts, images, js)
* Creating new variables in file settings.py:
* STATIC\_ROOT which refers to the static folder
  + STATICFILES\_DIRS which refers to the static folder
* Running command „python manage.py collectstatic” in gitbush which will copy all files from the static folder ale create another one in the root folder.

1. Dividing sections from html file

* Loading the static folder to home.html file using DTL (Django Template Language). DTL is used for generating dynamic content and dividing parts of code into sections which are then conected with each other through refrences.
* Method I:
  + Adding a line of code {% load static %} in the beginning of the html file
  + Changing all the refrences to css links and js scripts. The pattern is {% static ‘link\_or\_script\_path’ %}.
  + Changing all the refrences to images. The pattern is {% static ‘image\_path’ %}.
* Creating another html file named base in folder templates.
* Extracting to base.html the head and header section of home.html.
* Joining two html files with {% extends ‘base.html’ %} placed in the home.html
* Using {% block content %} and {% endblock %} to place in right place the code from the other file.
* Method II:
* Creating a folder named inlcudes
* Placing two files footer.html and navbar.html
* Cutting out the footer and nav sections from html files and placing them acordingly as the names of files
* Placing {% include 'includes/navbar.html' %} and {% include 'includes/footer.html' %} before and after the section block in the base.html file
* \* Adding to some files {% load static %} that need refrence to this folder.

1. Creating app (django-admin-panel)

* Creating a folder named category with „python manage.py startapp category” (bash). Its a Django framework command that generates a structure for new app.
* Registering the category app in INSTALLED\_APPS list in file settings.py
* Django.db models enables to define datamodels that we can use to keep, download, modify or delete data in database. It enables interaciton with database and it lets to manipulate data in Django app.
* Creating a category class in models.py in order to keep data in database
* Importing and registering model in admin.py
* Installing pillow framework for images
* Command „python manage.py makemigrations” creates migration files. It is used to propagate changes in the models into the database schema.
* Running created migration file „python manage.py migrate”
* Creating a superuser to enable entering into /admin site with command „winpty python manage.py createsuperuser” (bash)
* To change the spelling of „Categorys” to „Categories” in admin panel, we create meta class in models.py

1. Custom user model

* Creating new user app named accounts „python manage.py startapp accounts” (bash)
* In the main folder greatkart, we add in file settings.py another argument to variable INSTALLED\_APPS
* Creating accounts models in file models.py
* Importing AbstractBaseUser to assure basic user model implementation, and BaseUserManager to manage those models
* Creating mandatory fileds such as join date or last login date
* Creating superadmin model in file models.py
* Creating if cases for mandatory fields
* Creating user creator function
* Creating superuser creator function
* Setting superuser privileges to True
* Adding a new variable AUTH\_USER\_MODEL to file settings.py in greatkart folder with the path to the Account class
* Registering the model in admin.py file in accounts folder
* Deleting the previous database because it may contain old data that would result in a conflict with data
* Making migrations to the app with new functions
* Creating a superuser in bash

1. Changing password field in django admin panel to read only

* Importing in file admin.py the UserAdmin module which provides an ready to go admin configuration panel for the user model.
* Creating class AccountAdmin in admin.py file

1. Configuring media files

* Adding two varaibles MEDIA and MEDIA\_ROOT to settings.py file in greatkart folder. Those two files describe the path to images, movies etc.
* Adding „static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)” into urls.py to set path for the media files (folder media in root folder).
* Data is stored in default db.sqlite file created by Django framework

1. Pre-populate Category Slug

* Changing the type of sług variable to SlugField in models.py (category folder)
* Migrating files (bash)
* Creating class CategoryAdmin in admin.py (category folder) and registering it.

1. Creating store app (for products)

* Creating store app in bash
* Adding store app into setting.py in variable INSTALLED\_APPS as one of arguments
* Creating class product in models.py (store folder) with the necessary fields.
* Registering the app in admin.py (store folder) and creating duplicate action for sług field and registering it.
* Migrate changes

1. Adding products via admin panel
2. Adding products to template

* Importing product module from models.py in store folder to views.py placed in greatkart folder
* Adding context variable and returning it
* Deleting from home.html (templates folder) all the static displayed products, leaving only one
* Adding inside the home.html file, DTL code that runs eight times (numer of products added) for loop, displaying the same static product (for now)
* Displaying product name, price and image on the template using DTL code.

1. Adding Store Page

* Creating file urls.py in store folder
* Redirecting the store page in urls.py (greatkart folder)
* Importing views module into urls.py in store folder
* Creating function in views.py file in store folder
* Creating new folder and file named store and store.html in templates folder
* Importing template to created html store file
* Impotring header and footer from templates using DTL
* Copying code from store.html (greatkart\_templates folder) two sections: PAGETOP and SECTION into store.html (templates/store folder) inside the block and end content lines.
* Loading static for and changing the path of static images to display them on the store site

1. Adding products to store page

* Creating view in views.py (store folder) and rendering it
* Adding DTL for loop and reference code to product names, images and prices in the store.html (templates/store folder)
* Creating another variable „product\_count” in views.py (store folder) that counts the number or products
* Passing this variable through render function to store.html

1. Displaying products by category

* Adding path to urls.py in store folder
* Creating views in views.py (store folder)
* Importing category.models with category class

1. Making context processors for displaying categories

* Creating context processors file in category folder
* Creating function in context\_processors.py that stores all the categories and return them in dictionary format
* Adding this newly created python into the setting.py file (greatkart folder) into the TEMPLATES variable. This enables to use the menu\_links variable in any template we want.
* Adding a for loop and category name in the navbar section in templates folder and navabar.html (only displaying)
* Creating a function get\_url in the models.py (category folder) that gathers all of the sług category names
* The reverse django function takes on or more argumnets and returns a string representing the URL that matches the given URL patern
* Adding this function into the navbar reference (href = ) in the navbar.html file (templates/includes folder)

--- 27.06.2023 ---

1. Listing all the categories

* Adding to store.html in templates/store folder a for loop creating the category list on the left side of the store page
* Adding url to those interactive categories
* Adding one more list element in the store.html to display all categories, so basically retunring to the store page
  + This command „{% url 'store' %}” lets to add static url
* Adding the same „all categories” element but into the navbar placed in navbar.html (templates/includes folder)

1. Adding product detail subpage

* Creating url link in urls.py (store folder)
* Creating a function in models.py (store folder) that returns url links for all products
* Creating a view in views.py (store folder) to render the distinict parameters to destined site
* Creating new html file named product\_detail in store folder and importing to it necessary code from greatkart\_template folder

1. Single product view

* Adding to the function product\_detail() placed in views.py (store folder), variable that gets the parameters of a product which is curently displaye at the detail page
* Adding to product\_detail.html (templates/store folder) DTL code to display basic information about the name and price of the chosen product

1. Getting url for products

* Creating a function get\_url() in models.py store (folder) that creates the url links to the products detail page.
* Adding DTL code into store.html and home.html files to connect interactive parts of the page with links to other subpages (href=”{{url\_link}}”)
* Adding additional DTL code into store.html and home.html files to connect interactive parts such as logo or buttons with links to other subpages

1. Changing Banner
2. Out of stock product

* Adding if condition in DTL code into the product\_detail.html file (templates/store folder) to display whether a product is out of stock or is available.

1. Added .gitignore file to not include unnecessary file into the git repository while commit

--- 27.06.2023 ---

--- 28.06.2023 ---

1. Cart app and cart page

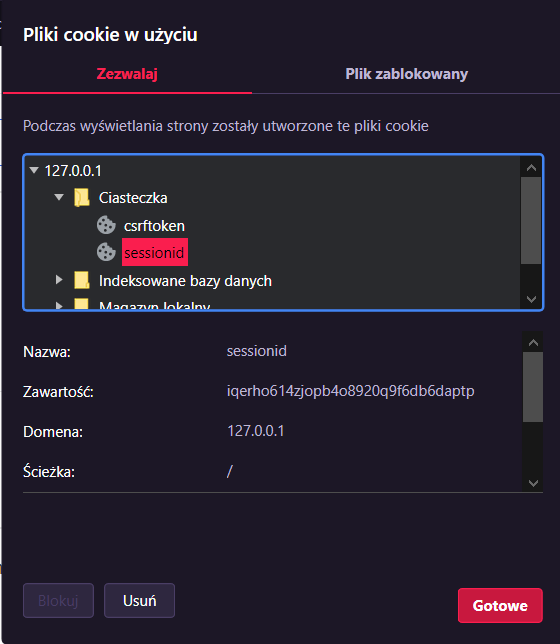
* Creating cart app via bash named „carts”
* Adding new app to settings.py (greatkart folder) as one of the argument in INSTALLED\_APPS variable
* Creating urls.py in the carts app and adding default path to cart site
* Creating another path in urls.py (greatkart folder) that redirects the request to urls.py in carts folder
* Adding a view in views.py (carts folder)
* Adding cart.html file (templates/store folder) and importing the template code into it

1. Cart and cart items models

* Adding models to models.py (carts folder)
* Registering the models in admin panel via the admin.py file (carts folder)

1. Add to cart functionality without logging in with sessions key

* Adding a function to redirect the connection to cart page when clicking the buton „add to cart”
* Adding the DTL code into product\_detail.html (templates/store folder)
* Session keys are stored in the cookies of the website. So the next step is to get those session keys from the cookies folder of the website.





* Creating views on views.py (carts folder)
  + Function \_cart\_id() is a private function that takes as argument a request. It assigns to cart variable a session key which is linked to the request. If the request doesn’t have a session key in databse it creates and returns the session key.
  + Function add\_cart takes request and id of product as arguments. It gathers all the information about the product which id is equal to the selected product identificator. If a cart doesnt exist it creates one with the assigned session key. Otherwise it collets all the information from the available cart that is linked with the correct session key. The chosen product and cart are used then to select which cart item should be placed in the cart. If the object is already added it just add the quantity of product by one. Otherwise the product is added to the cart.

1. Cart view for getting cart items

* Adding to function cart() in views.py (carts folder)
  + Function cart() gives back all the selected products, total price of the products and the number of products selected in cart.

1. Implementing data in the cart page

* Using the DTL code to implement collected data into the cart.html (templates/store folder)
* Creating a function sub\_total in models.py (carts folder) to count the price of the product instances
* Adding DTL code to increase the quantity of a product by clicking on the button on the site

1. Calculataing tax and total calculation

* Adding two variables tax and grand\_total in views.py (carts folder) to calculate the tax and final price of products and pass it to the cart.html file

1. Decrement and remove buton

* Adding function remove\_cart and delete\_cart in views.py (carts folder)
  + Function remove\_cart gathers the information about the products in the session cart. After that it cheks if the quantity of a distinct product is greater than 1 and if the condition is fulfilled it deincrements the quantity of the product by one Otherwise it deletes the product from the session cart page. In the end it redirects to cart page
  + Function delete\_cart gathers the information about the products in the session cart. Then it the deletes the chosen product from the cart
* Adding two urls paths to urls.py (carts folder)
* Adding DTL code into cart.html file (templates/store folder) for the remove product and decrease quantity buttons

1. Check for empty page

* Adding DTL if condition into the cart.html file (templates/store folder) that whenever the cart is empty it displays infomation and button that redirects to the store page.

1. Fixing „add to cart” buttons in store page

* Fixed the „add to cart” buttons in store page using DTL code that redirects to cart page whenever someone picks a product.
* Fixed the „continue shoping” button in the cat page that redirects to store page
* Added to view product detail page in the cart page whenever someone clicks on the product name.

1. Check if product is added to cart

* Adding to product detail page a button that indicates when a product is already in the cart and a button which redirects to cart page.
* Adding to function product\_detail in views.py (store folder) a variable that retrurn true or false whenever a product exist in the cart or not.
* Adding this variable into the context dictionary and using DTL code, implementing it in product\_detail.html file (templates/store folder)

1. Adding a counter to the navbar cart icon

* Creating a contect\_processors.py file in carts folder
* Creating a function counter in the contect\_processors.py file
  + This function counts the number of elements in the cart and returns it as dictionary data type.
* Adding this function path into the settings.py (greatkart folder) in TEMPLATES variable as one of the aruguments.
* Adding the returned parametr as DTL code to navbar.py file (templates/includes folder)

1. Changing the button in store page to view details about the product
2. Paiginator

* Importing to views.py file (store folder) some django paginator modules
* Adding the paginator varaiables into the store function in store.py file (store folder)
* Adding DTL code into the store.html file (templates/store folder) that enables six products to appear per page

--- 28.06.2023 ---

--- 29.06.2023 ---

1. Fixing products warning and empty cart issue

* Fixing products warning by adding a order\_by function at the end of code that gets product objects in views.py (store folder)
* Fixing the empty cart problem by adding ObjectDoesNotExist module inside the except block in the views.py (store folder)

1. Search function

* Design the url path to search „127.0.0.1:8000/store/search/?keyword=searched\_item”
* Adding path to search site into urls.py (store folder)
* Creating a new function search in views.py (store folder)
  + Search function takes as argument request and checks if in the looks for a keyword written inside the search bar. If the keyword exists it displays all the products which have this keyword pharse inside the product description or name fields. Then it renders the result to store.html file being the store page.
* Adding a category at the beginning of the link to store and product detail paths in urls.py file (store folder). This operation is made to solve the conflict between those paths and the search path.
* Adding the name attribute to search input in the navabar.html file (templates/includes folder)
* Adding the call function DTL code for the search button in the navabar.html file (templates/includes folder)
* Adding if condition when given products exists into the store.html (templates/includes folder)

1. Product variation preparation (color and size)

* Putting the choose color and select size elements from product\_detail.html (templates/store folder) file into a form
* Changing the display of the chose color and size elements to drop down list box
* Adding new class Variation that lets to create multiple variations of products in models.py (store folder)
* Registering the model in admin.py file (store folder) and making a migrations
* Adding some variations into the admin panel

--- 29.06.2023 ---

--- 04.07.2023 ---

1. Dynamic fetch color

* Displaying the created variations via admin panel on the product detail page
* Adding the DTL code into product\_detail.html file (templates/store folder) which dynamically displays color varitaions of products.

1. Variation manager for variation model

* Adding Variation manager class into the models.py file (store folder)
  + Class VariationManager has two function inside which return color or size depeding on which is called in the product\_detail.html file (templates / store folder)
* Adding DTL code to the product\_detail.html file (templates/store folder) that displays that displays the variation of a product
* Adding to product class a objects variable that enables the variation manager class
* Changing the method of the form to POST
  + Adding the expression „{% csrf\_token %}” to create a unique value generated token for each session to validate if the form submission originated from the same website and user.
* Adding an if condition which checks the request method to be POST in the views.py file (carts folder)

1. Getting the instance of variation

* Adding a for loop into the views.py (carts folder) that looks for any information inside the request such as color, size, brand, token etc.
* Adding the try block in views.py file (carts folder) which checks if the chosen variation of the product does exist in the database
* Changing in the models.py file (store folder) the str function to return the variation value instead of the product name
* Adding a list variable in views.py file (carts folder) which stores the variation of products to further display them in the cart
* Adding into the CartItem class in the models.py file (carts folder) a variations variable
* Making a migrations
* Changing the return of product in models.py file (carts folder) from \_\_str\_\_ type to \_\_unicode\_\_
* Adding displays lists to admin panel for Cart and CartItem in the admin.py file (carts folder)

1. Adding variation in cart item

* Checking in views.py (cart folder) if the list of product variations is empty. If the list contains some variations then all of them are saved as unique products.
* Displaying the info about products variations in the cart page using DTL code inside the cart.html file (templates/store folder)

1. Grouping cart item variations

* Creating a groping method in views.py (carts folder) in the „add\_cart” function.
  + The first if condition in the add\_cart function checks if the request method is POST. Then it adds to the variable called „products\_varation” size and color chosen in the web page with product details.
  + The variable called „is\_cart\_item\_exist” checks if the chosen product and cart exist. If it does then into the list variable called „ex\_var\_list” are gathered all existing variations of the chosen product. Also the id’s of the variations are stored in the id list variable called „id”.
  + Next the if condtion checks if the chosen variation of product exists in the already created ones. If it exists then it is chosen from the cart that it exists in by the id and the quantity of it is incremented by one. Otherwise if the chosen variation of product doesn’t exist then it is being created and added to cart with quantity equal to one.

1. Decrement quantity and remove buttons

* Adding additional parameter to urls.py file (carts folder) called „/<int:cart\_items\_id>/” to the url links that sends the information from the cart.html file template to the remove\_cart and delete\_cart function located in views.py (carts folder) whenever one of them is called.
* In the remove\_cart and delete\_cart functions the cart item gets chosen by product name, cart id and cart item id. Then depending on the action it is being decremented or remove completly from the session cart.
* Also the button that makes the product quantitiy incremented by is set in form with POST method to enable the call of the add\_cart function placed in the views.py (cart folder)

**Registration proces**

1. Registration preparation

* Creating a urls.py in accounts folder and adding paths to registration, login and logout pages. Also adding to urls.py file (greatkart folder) path to the urls.py in accounts folder.
* Adding in the views.py file (accounts folder) functions for register, login and logout.
* Creating a new folder accounts in the templates and adding to it two html files called login and register.
* Adding to those two html file prepared templates
* Creating a django model form
  + Creating a forms.py file in accounts folder that is a django generated form
* Inicilizing the form in the views.py (accounts folder)

--- 04.07.2023 ---

--- 05.07.2023 ---

1. Implementing model forms and editing \_\_init\_\_ method

* Adjusting the form in register.html file (templates/accounts folder) by displaying the generated django form using DTL code.
* Creating passoword inputs fields in the forms.py file (accounts folder) and displaying them in the register site
* Creating a placeholder for the password input fields in the forms.py file (accounts folder)
* Creating a function \_\_init\_\_ in the forms.py file (accounts folder)
  + Function \_\_init\_\_ executes without being called out. The super class has access to call the RegistrationForm class. The \*args parameter collects extra positional argumnets such as (1,2,3) and the \*\*kwargs collects extra keyword argumnets as dictionary {‘name’ : ‘Pawel’, ‘age’ : 22}. Then the for loop goes through every single field from the RegistrationForm class and changes it css class to be adequate to the web design.
  + Adding additional placeholders for other fields

1. Making view and editing model from clean method to check

* Updating the register function in views.py (accounts folder)
  + Checking if the sent request method is post. Then sending the request info about the user to the RegistrationForm class for validation’
  + After validation the data from the form is collected into seperate variables with generated username from the email
  + Creating the user with Account class and create\_user function from MyAccountManager class.
  + If the method of the request is GET the it only shows the registration form page
* Creating a new function called clean in the forms.py file (accounts folder)
  + This function checks if the password and confirm password fields are the same
  + If the passwords fields aren’t the same then there is a error raised which is displayed in the register site using DTL code in the register.html file (accounts folder)

1. Django Message Alerts

* Creating alert.html file (templates/includes folder) which displays a message when a regiin the registration proces.
* Adding in views.py (accounts folder) a message sucession function that sends the infomartion about succesful registration into the registration site
* Displaying the succesful registration message in the register.html file (templates/accounts folder)

1. User Login Functionality

* Adding to login and logout functions in views.py file (accounts folder) the functionality to login and logout from the account.
* Adding a dekorator in views.py file (accounts folder) for the logout function so only the authenticated user can access it.
* Including alert.html in the login.html file (templates/accounts folder) using the DTL code.

1. Account activation proces using the authentication via email (token validation)

* Creating the user activation script in the register function in views.py file (accounts folder)
  + Importing needed modules from Django
  + Using those modules to create an email message which is then sent via SMTP that is configurated in the settings.py file (greatkart folder)
* Creating a account\_verification\_email.html file (templates/accounts folder) which will be used as a message with the token authentication via url link.
  + The DTL code {% autoescape off %} is used to prevent the variables getting out from the sent message. This situation may occur while a Cross Site Scripting attack.

--- 05.07.2023 ---

--- 06.07.2023 ---

1. Account activation part 2

* Adding a url path in urls.py (accounts folder) for the activation function
* Decoding the uid and token in the activation function
* Checking if user exists and the token has been activated. If the condition is met then the user status becames active and the message is sent to the user that account verification ran successfully. Otherwise it displays an error.
* In the login.html file (templates/accounts folder) we check if the url link which takes user from the registration site to login site has in it the verification phrase. If it does than it indicates that the activation email was succefully sent to the registered user.
* IMPORATANT NOTES
  + The date\_joined and last\_login variables in the models.py file, Account class (accounts folder) should be designed as „models.DateTimeField” otherwise the function that generates token will display an error.
  + Password for email host in the settings.py file (greatkart folder) for the gmail account should be generated via passwords apps that are located in the gmail profile security configuration. Also the two step verification should be enabled.

1. Preparing dashboard for users

* Creating dashboard.html file (templates/accounts folder) that displays the user dashboard whenever he is logged in
* Creating a simple function „dashboard” in the views.py file (accounts folder) that renders the created html file.
* Creating two paths to those functions in the urls.py file (accounts folder)
* Seperating the dashboard navbar from dashboard.html file to another dashboard\_navbar.html file and then including it in the orginal dashboard file
* Adding a logout functionality to dashboard page

1. Forgot password process and password validation

* Creating a fogotPassword function in the views.py (accounts folder) (similar to register)
  + Function checks if the request method is POST. Then the function pulls the email from the form and checks if it exists in database. After that the email message is being formulated to the user with a link to the reset password page.
* Creating a forgotPassword and reset\_password html files (templates/accounts folder)
  + The forgotPassword.html contains the input field with user address email that is being sent to the forgotPassword function
  + This file also contains a if statment that activates whenever the user clicks the url link from his email. Then it displays new password creator.
  + The reset\_password.html is being sent as a email message to the user with the reset password link
* Creating a validateEmail function in views.py (accounts) which encodes the user id and receives a token which was created in the forgotPassword function. Whenever the function detects user and token it redirects to the forgotPassword page.
* Creating a changePassword function in the views.py (accounts folder)
  + Function checks if the request method is POST. Then the function pulls the email ,password and confirm password values from the form. Then it checks if the password are identical and if they are then it changes the old one with the new password. Otherwise it redirects back to the password reset site so the user may try again.
* Adding additional paths to urls.py file (accounts folder)

--- 06.07.2023 ---