Πανεπιστήμιο Πελοποννήσου
Σχολή Οικονομίας, και Τεχνολογίας
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Π.Μ.Σ. Επιστήμη Υπολογιστών

ΔΗΜΙΟΥΡΓΙΑ ΜΙΑΣ ΡΕΑΛΙΣΤΙΚΗΣ E-COMMERCE ΕΦΑΡΜΟΓΗΣ ΜΕ ΧΡΗΣΗ ΤΟΥ .NET ΚΑΙ ΤΗΣ ANGULAR

ΣΠΥΡΙΔΩΝ ΕΜ. ΡΟΔΙΤΗΣ

ΕΠΙΒΛΕΠΟΝΤΕΣ:
ΚΩΝΣΤΑΝΤΙΝΟΣ Θ. ΒΑΣΙΛΑΚΗΣ
ΚΑΘΗΓΗΤΗΣ ΠΑΝΕΠΙΣΤΗΜΙΟΥ ΠΕΛΟΠΟΝΝΗΣΟΥ
ΓΡΗΓΟΡΙΟΣ Δ. ΔΗΜΗΤΡΟΥΛΑΚΟΣ
ΜΕΛΟΣ ΕΔΙΠ Α' ΒΑΘΜΙΔΑΣ ΠΑΝΕΠΙΣΤΗΜΙΟΥ ΠΕΛΟΠΟΝΝΗΣΟΥ

ΤΡΙΠΟΛΗ, ΦΕΒΡΟΥΑΡΙΟΣ 2026

University of Peloponnese
Faculty of Economy and Technology
Department of Informatics and Telecommunications

MASTER'S THESIS

MSc in Computer Science

CREATION OF A REALISTIC E-COMMERCE APPLICATION USING .NET AND ANGULAR

SPYRIDON EM. RODITIS

SUPERVISORS:
CONSTANTINE T. VASILAKIS
PROFESSOR, UNIVERSITY OF PELOPONNESE
GRIGORIOS D. DIMITROULAKOS
1st GRADE ACADEMIC LABORATORY INSTRUCTOR, UNIVERSITY OF PELOPONNESE

TRIPOLI, FEBRUARY 2026

Ευχαριστίες

Η παρούσα διπλωματική εργασία είναι αφιερωμένη στους κ.κ. Καθηγητές του τμήματος Πληροφορικής και Τηλεπικοινωνιών του Πανεπιστημίου Πελοποννήσου, οι οποίοι καθ' όλη τη διάρκεια των σπουδών μου με ενέπνεαν και με διαφώτιζαν με τις πολύτιμες γνώσεις τους. Επίσης, θα ήθελα να ευχαριστήσω ιδιαιτέρως τους επιβλέποντες καθηγητές μου κ.κ. Κωνσταντίνο Βασιλάκη και Γρηγόρη Δημητρουλάκο, οι οποίοι μου ανέθεσαν την εκπόνηση της εν λόγω διπλωματικής εργασίας και με καθοδήγησαν ώστε να τη φέρω εις πέρας. Τέλος, θα ήθελα να ευχαριστήσω από καρδιάς την οικογένειά μου για την όλη στήριξη προς το πρόσωπό μου.

Περίληψη

Η παρούσα διπλωματική εργασία παρουσιάζει την ανάπτυξη μιας ολοκληρωμένης εφαρμογής ηλεκτρονικού καταστήματος (e-commerce application), βασισμένη σε σύγχρονες τεχνολογίες κι αρχιτεκτονικές πρακτικές των οικοσυστημάτων .NET και Angular. Στο πλαίσιο της υλοποίησης, χρησιμοποιούνται εργαλεία όπως το Microsoft Visual Studio Code για την ανάπτυξη του κώδικα, η εφαρμογή Postman για τη δοκιμή και αξιολόγηση των API endpoints, καθώς και η εφαρμογή Docker για την οργάνωση κι αποθήκευση της SQL βάσης δεδομένων, την οποία δημιουργούμε μέσω του .Net migrations system. Η αρχιτεκτονική του backend δομείται ως μία εφαρμογή .NET Core τριών αλληλένδετων projects, αξιοποιώντας το dotnet CLI, υιοθετώντας τα πιο διαδεδομένα σχεδιαστικά πρότυπα. Μερικά από αυτά που θα μας απασχολήσουν είναι το Generic Repository Pattern, το Specification Pattern και το ASP.NET Identity, το οποίο αφορά την ασφαλή διαχείριση κι εγγραφή των χρηστών στη βάση δεδομένων της εφαρμογής μας.

Στο frontend, η εφαρμογή αναπτύσσεται με χρήση της Angular, χρησιμοποιώντας τεχνολογίες όπως το Angular Material και η Tailwind CSS βιβλιοθήκη, για τη δημιουργία μίας σύγχρονης και λειτουργικής διεπαφής χρήστη. Επιπλέον, υλοποιούνται επαναχρησιμοποιήσιμα Angular components βασισμένα στις Angular Reactive Forms, καθώς και Angular Object Observers για την παρακολούθηση μεταβολών και τη δυναμική ενημέρωση της διεπαφής. Η επιχειρησιακή λειτουργικότητα της εφαρμογής μας περιλαμβάνει χαρακτηριστικά όπως είναι η σελιδοποίηση, η ταξινόμηση, η αναζήτηση και το φιλτράρισμα προϊόντων. Επίσης, γίνεται χρήση του εργαλείου Redis Server για τη διαχείριση του καλαθιού αγορών, τη δημιουργία παραγγελιών και την ολοκλήρωση ασφαλών πληρωμών μέσω της online payment platform Stripe, σύμφωνα με τα διεθνή πρότυπα 3D Secure.

Η παρούσα εργασία συνιστά ένα πλήρες και σύγχρονο παράδειγμα ανάπτυξης μίας entry-level e-commerce εφαρμογής, παρουσιάζοντας τεχνικές, εργαλεία και βέλτιστες πρακτικές που αντανακλούν τις απαιτήσεις και τις τάσεις της σύγχρονης βιομηχανίας λογισμικού.

Λέξεις κλειδιά

.NET Core, Multi-project architecture, Angular, Angular CLI, Angular Material, Tailwind CSS, Reactive Forms, Reusable components, Object observers, Repository Pattern, Specification Pattern, DbContext boundaries, ASP.NET Identity, Role-based authentication, Redis Server, Docker, Microsoft Visual Studio Code, Stripe, Postman, E-commerce application development, Web application architecture

Abstract

This Master's Thesis presents the development of a complete e-commerce application, based on modern technologies and architectural practices from the .NET and Angular ecosystems. During implementation, tools such as Microsoft Visual Studio Code are used for code development, Postman for testing and evaluating API endpoints, as well as Docker for organizing and storing the SQL database, which we create through the .NET migrations system. The backend architecture is structured as a .NET Core application, consisting of three interconnected projects, utilizing the dotnet CLI and adopting widely used design patterns. Some of the patterns discussed include the Generic Repository Pattern, the Specification Pattern, and ASP.NET Identity, which concerns the secure management and registration of users in the application's database.

On the frontend, the application is developed using Angular, employing technologies such as Angular Material and the Tailwind CSS library to create a modern and functional user interface. Additionally, reusable Angular components are implemented based on Angular Reactive Forms, along with Angular Object Observers for monitoring changes and dynamically updating the user's interface. The business functionality of the application includes features such as product pagination, sorting, searching, and filtering. Furthermore, the Redis Server tool is used for managing the shopping cart, creating orders, and completing secure payments through the Stripe online payment platform, in accordance with international 3D Secure standards.

This Master's Thesis constitutes a complete and modern example of developing an entry-level e-commerce application, presenting techniques, tools and best practices that reflect the requirements and trends of today's software industry.

Keywords

.NET Core, Multi-project architecture, Angular, Angular CLI, Angular Material, Tailwind CSS, Reactive Forms, Reusable components, Object observers, Repository Pattern, Specification Pattern, DbContext boundaries, ASP.NET Identity, Role-based authentication, Redis Server, Docker, Microsoft Visual Studio Code, Stripe, Postman, E-commerce application development, Web application architecture

Πίνακας Περιεχομένων

Πρόλογος

Η παρούσα διπλωματική εργασία επικεντρώνεται στην ανάπτυξη μίας πλήρως δομημένης διαδικτυακής εφαρμογής ηλεκτρονικού εμπορίου, αξιοποιώντας σύγχρονες τεχνολογίες κι αρχιτεκτονικά πρότυπα. Το C# υπόβαθρο της εργασίας αναπτύχθηκε βασιζόμενο στην πλατφόρμα της Microsoft, .Net Core 9, ακολουθώντας μια καθαρή αρχιτεκτονική αρθρωτού σχεδιασμού, με έναν διαχωρισμό τριών επί μέρους projects: API project, Core project και Infrastructure project. Για τη διαχείριση των δεδομένων εφαρμόστηκαν τα μοτίβα ανάπτυξης Generic Repository pattern και Specification Pattern, τα οποία εξασφάλισαν την επεκτασιμότητα, τη χαμηλή σύζευξη και την ευελιξία στη δημιουργία σύνθετων queries για κατηγορίες προϊόντων, φίλτρα αναζήτησης, ταξινόμηση και σελιδοποίηση. Η ανάπτυξη κι ο έλεγχος των API Endpoints πραγματοποιήθηκε μέσω των εφαρμογών Microsoft Visual Studio Code και Postman, ενώ το υπόβαθρο της SQL βάσης δεδομένων του ηλεκτρονικού καταστήματος υλοποιήθηκε με τη βοήθεια του .Net Migrations System και της εφαρμογής Docker.

Για τη διεπαφή χρήστη χρησιμοποιήθηκε η πλατφόρμα της Angular 20, αξιοποιώντας νέα χαρακτηριστικά του framework όπως τα Angular Signals, οι Reusable Reactive Forms, τα Observer-objects based components, οι TypeScript Guard κλάσεις και το Angular Routing, με στόχο τη βελτιστοποίηση της απόδοσης και της αρχιτεκτονικής του client μέρους της εφαρμογής. Η σχεδίαση της διεπαφής χρήστη υλοποιήθηκε με χρήστη της Tailwind CSS βιβλιοθήκης, ενώ για τη διαχείριση του καλαθιού αγορών εκμεταλλευτήκαμε τον Redis Server ως in-memory data store, επιτρέποντας τη γρήγορη κι αξιόπιστη αποθήκευση προσωρινών δεδομένων χρήστη.

Η παρούσα διπλωματική εργασία παρουσιάζει μια ολοκληρωμένη, τεχνικά δομημένη λύση, η οποία βασίζεται σε σύγχρονες αρχές σχεδιασμού λογισμικού, με στόχο την παραγωγή μιας επεκτάσιμης, αξιόπιστης και υψηλής σε απόδοση web εφαρμογής.

<u>Ακολουθεί μια σύντομη περιγραφή της δομής της διπλωματικής εργασίας ανά κεφάλαιο:</u>

Κεφάλαιο 1 – Βασικά Στοιχεία Ενός API:
Παρουσιάζεται η αρχική υλοποίηση του API project στο Microsoft Visual Studio Code, η δημιουργία των πρώτων C# κλάσεων τύπου entity, η προσθήκη του DB connection string και η χρήση του Entity Framework Migrations System για τη δημιουργία της SQL βάσης δεδομένων.

Κεφάλαιο 2 – Η Αρχιτεκτονική του API Project:
Αναλύεται η βασική δομή του API project, η εισαγωγή του Repository Pattern, η επέκταση της λειτουργικότητας της οντότητας Product, καθώς και οι διαδικασίες εισαγωγής κι ανάκτησης δεδομένων.

Κεφάλαιο 3 – Generic Repository Pattern:
Παρουσιάζεται ο σχεδιασμός και η υλοποίηση του Generic Repository Pattern, η ενσωμάτωση του Specification Pattern και η εφαρμογή evaluators κλάσεων για την επεξεργασία πολύπλοκων queries.

Κεφάλαιο 4 – Σύνθετο Φιλτράρισμα, Σελιδοποίηση, Αναζήτηση:
Αναλύονται οι μηχανισμοί σελιδοποίησης, αναζήτησης και δυναμικού φιλτραρίσματος προϊόντων μέσω επεκτάσιμων API endpoints.

Κεφάλαιο 5 – Angular:
Περιγράφεται η αρχική δομή του Angular project, η δημιουργία components, η χρήση του HTTP Client για επικοινωνία με τον API server και βασικές έννοιες της γλώσσας TypeScript.

Κεφάλαιο 6 – Angular Project – Διεπαφή Χρήστη:
Αναπτύσσονται τα κύρια UI components, η προβολή προϊόντων, το φιλτράρισμα, η ταξινόμηση, η υλοποίηση της σελιδοποίησης και της αναζήτησης στο frontend μέρος της εφαρμογής.

Κεφάλαιο 7 – Angular Routing:
Παρουσιάζεται η δημιουργία των Routing modules, νέων σελίδων και η διαμόρφωση της πλοήγησης μεταξύ βασικών τμημάτων της εφαρμογής.

Κεφάλαιο 8 – Καλάθι Αγορών (.NET Project):
Αναλύεται η ενσωμάτωση του Redis Server μέσω της εφαρμογής Docker, η δημιουργία των shopping cart κλάσεων και η υλοποίηση του Shopping Cart API.

Κεφάλαιο 9 – Καλάθι Αγορών (Angular Project):
Περιγράφονται τα Angular components του καλαθιού αγορών, η χρήση των Angular Signals για ασύγχρονες ενημερώσεις του UI, η λειτουργικότητα του Order Summary component και η επικοινωνία με τον API server.

Κεφάλαιο 10 – ASP.NET Core Identity:
Παρουσιάζεται η ενσωμάτωση του Microsoft .Net Identity για εγγραφή και είσοδο χρηστών στη βάση δεδομένων της εφαρμογής, η δημιουργία User endpoints και η προσθήκη της οντότητας User Address.

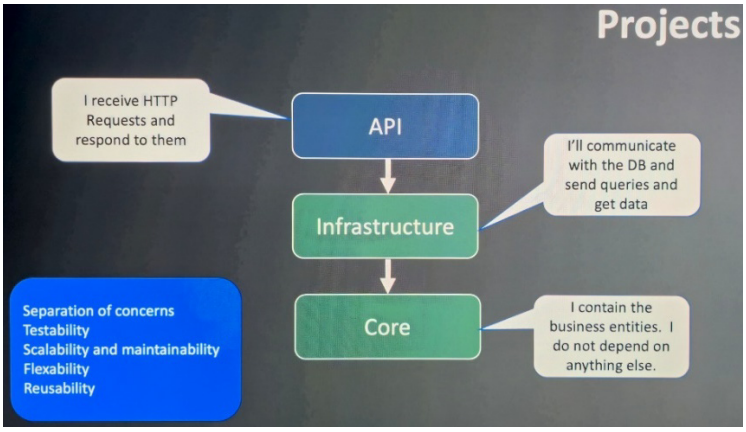
Κεφάλαιο 11 – Angular - Identity:
Αναλύεται η διαχείριση χρηστών στο frontend, η Account Angular Service, τα Login και Register components και οι Angular Guards για προβολή και διαχείριση των προστατευόμενων σελίδων.

Κεφάλαιο 12 – Τοποθέτηση και Ολοκλήρωση Παραγγελιών:
Περιγράφεται η διαδικασία του client checkout, η ενσωμάτωση της πλατφόρμας Stripe για διαδικτυακές πληρωμές, η δημιουργία σχετικών Angular components και η ολοκλήρωση της παραγγελίας ενός εγγεγραμμένου χρήστη.

Κεφάλαιο 1. Βασικά Στοιχεία του API Project

1.1 Εισαγωγή

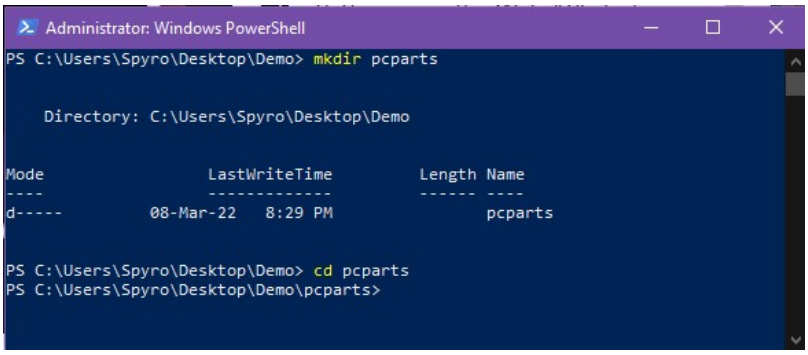
Βασικό στοιχείο ενός API - .NET CLI (command line interface) project είναι η ικανότητα του στο να μπορεί να δεχθεί HTTP αιτήματα (HTTP requests) από έναν client και να του επιστρέφει την ανάλογη απάντηση (HTTP response) από μία βάση δεδομένων. Για να συμβεί κάτι τέτοιο θα πρέπει το API project να μπορεί να επικοινωνεί με τη βάση δεδομένων μέσω του Infrastructure project κι αυτό με τη σειρά του με το Core project, το οποίο και θα φιλοξενεί τις λειτουργικές οντότητες της εφαρμογής μας. Ο λόγος για τον οποίο υλοποιούμε την εφαρμογή μας βασιζόμενη στον αρθρωτό αυτό σχεδιασμό των τριών projects (API project, Infrastructure project, Core project), είναι η δημιουργία μιας ευέλικτης web εφαρμογής την οποία θα μπορούμε να ελέγχουμε εύκολα κατά το στάδιο ανάπτυξής της, να μπορούμε να κρίνουμε εύκολα το αν και κατά πόσο είναι αποδοτική ως προς τη διαχείριση υπολογιστικών πόρων και φυσικά στο μέλλον να μπορούμε να πραγματοποιήσουμε αλλαγές (πχ να μεταβούμε από SQL server σε PostgreSQL) χωρίς να επηρεάζουμε καθολικά τη δομή της. Στο τέλος αυτού του κεφαλαίου θα έχουμε δημιουργήσει την εφαρμογή *PcParts* η οποία και θα βασίζεται στην προαναφερθείσα δομή υλοποίησης, όπως αναπαρίσταται και στην εικόνα 1.



Εικόνα 1

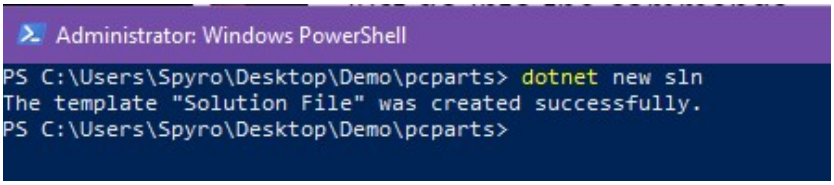
Σε περιβάλλον Microsoft Windows, ανοίγουμε ένα παράθυρο Terminal/Powershell κι εκτελούμε κατά σειρά τις εντολές των εικόνων 2 έως και 5, προκειμένου να δημιουργηθεί το API project solution, εντός του καταλόγου *Demo*.

- **mkdir pcparts**



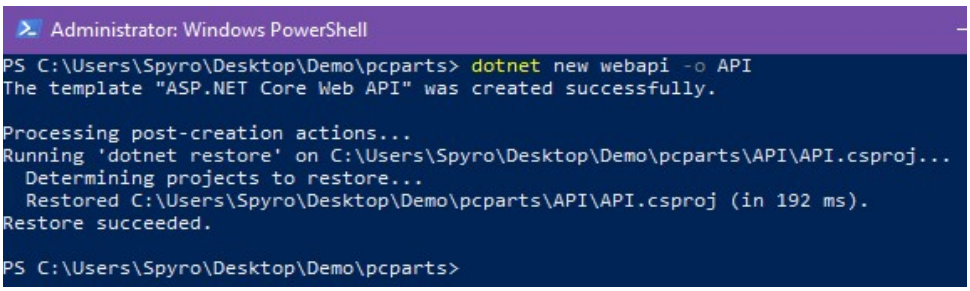
Εικόνα 2

- **dotnet new sln**



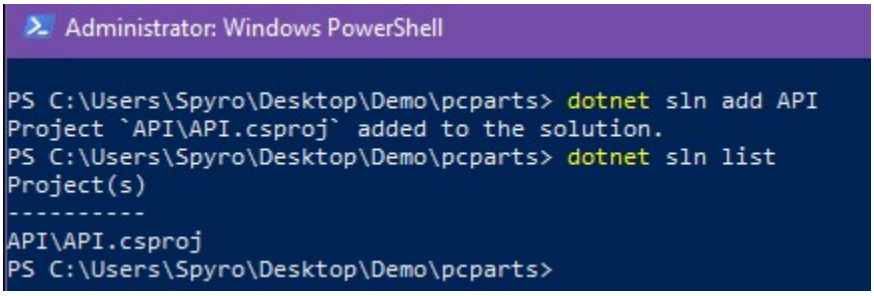
Εικόνα 3

- ***Dotnet new webapi -o API (webapi type of project inside API folder)***



Εικόνα 4

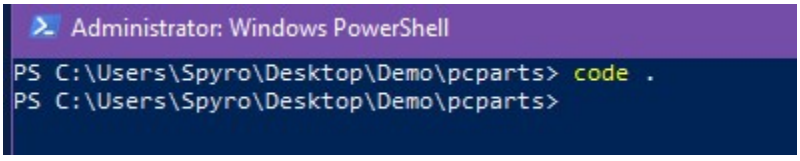
- ***Add the webapi project to the pcparts solution***



Εικόνα 5

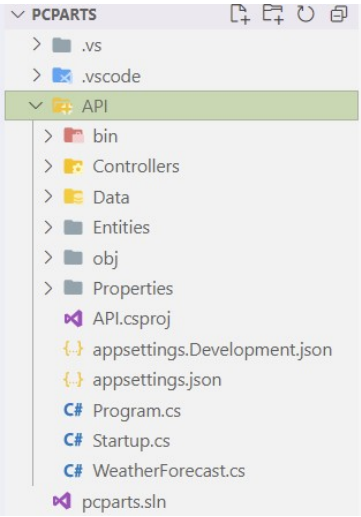
Τέλος, με την εντολή της εικόνας 6 εκκινούμε το περιβάλλον το περιβάλλον του Visual Studio Code, στο οποίο και θα υλοποιήσουμε την εφαρμογή μας.

- **Code .**



Εικόνα 6

Στην εικόνα 7 απεικονίζεται η βασική δομή των καταλόγων του *PcParts Web Application*, στο περιβάλλον του Visual Studio Code.



Εικόνα 7

Εντός του καταλόγου *Controllers* θα προσθέσουμε μια νέα C# κλάση με τίτλο *ProductsController.cs.* κάνοντας χρήση ενός *route attribute ([Route("api/[controller]")])* προκειμένου κάθε HTTP αίτημα να εξυπηρετείτε από τον συγκεκριμένο controller με βάση τη μορφή του. Το HttpGet attribute λειτουργεί ως απάντηση κι εκτελείται ο κώδικας της εικόνας 8. Ουσιαστικά, ο χρήστης δίνοντας ένα URL της μορφής *http://localhost:50001/api/products* , θα λάβει μία λίστα με όλα τα προϊόντα που θα φιλοξενούνται στη βάση δεδομένων της εφαρμογής μας. Χρησιμοποιώντας το *{id}* εντός του δεύτερου HttpGet attribute, δίνουμε τη δυνατότητα στον χρήστη να αιτηθεί μόνο ένα συγκεκριμένο προϊόν προς εμφάνιση.

ProductsController.cs

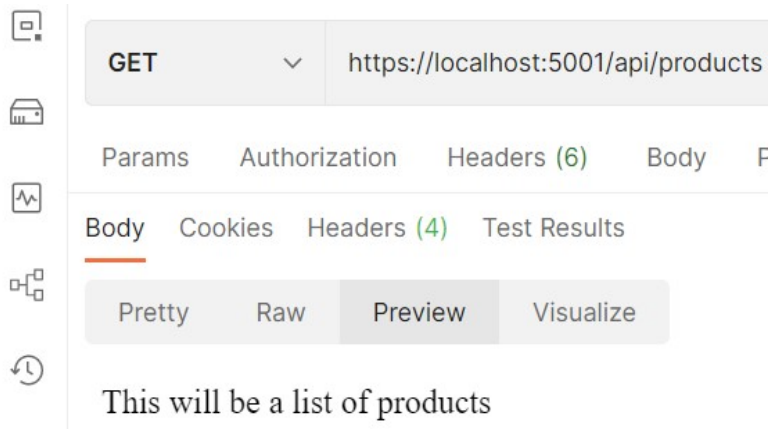
API > Controllers > ProductsController.cs > {} API.Controllers

```
1   using Microsoft.AspNetCore.Mvc;
2
3   namespace API.Controllers
4   {
5       [ApiController]
6       [Route("api/[controller]")]
        0 references
7       public class ProductsController : ControllerBase
8       {
9           [HttpGet]
            0 references
10          public string GetProducts()
11          {
12              return "This will be a list of products";
13          }
14
15          [HttpGet("{id}")]
            0 references
16          public string GetProduct(int id)
17          {
18              return "single product";
19          }
20      }
21  }
```

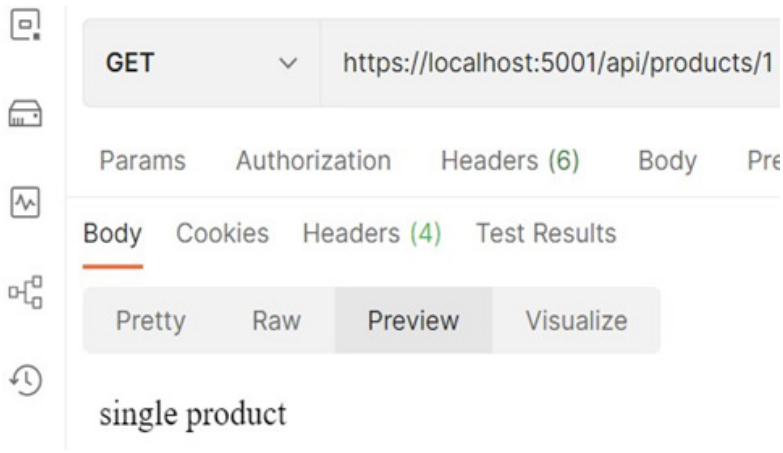
Εικόνα 8

1.2 Εκτέλεση του API Project στο Περιβάλλον του Visual Studio Code

Ανοίγοντας ένα παράθυρο τερματικού/terminal εντός του Visual Studio Code, μεταβαίνουμε στον κατάλογο API που δημιουργήσαμε νωρίτερα. Με την εντολή *dotnet run* εκτελείται ο κώδικάς μας και μέσω της εφαρμογής Postman, πληκτρολογώντας τη διαδρομή *https://localhost:5001/api/products* ή τη διαδρομή *https://localhost:5001/api/products/1* παίρνουμε ως απάντηση από τον *ProductsController* το αποτέλεσμα των εικόνων 9 και 10 αντίστοιχα.



Εικόνα 9



Εικόνα 10

1.3 Προσθήκη Μίας C# Κλάσης Τύπου Entity

Στο προηγούμενο βήμα ελέγξαμε δύο απλά *HTTP Requests* στην πλατφόρμα *Postman*. Το *API project* μας επέστρεψε 2 απαντήσεις μέσω του *ProducstController*, οι οποίες δεν ήταν άλλες από 2 γραμμές κειμένου (εικόνες 8,9,10). Σε αυτό το βήμα θα ξεκινήσουμε την προετοιμασία του *API project* ώστε να μπορεί να υποστηρίζει την άντληση δεδομένων από μία *SQL* βάση δεδομένων και την απόδοσή τους ως απάντηση σε ένα συγκεκριμένο *HTTP* αίτημα από τον *client*. Εντός του φακέλου *entities* δημιουργούμε μια *C# class* με τίτλο *Product.cs*. Η κλάση αυτή περιέχει δύο Properties, την *Id* και την *Name* (εικόνα 11).

```
C# Product.cs ×
API > Entities > C# Product.cs > ...
1   namespace API.Entities
2   {
        1 reference
3       public class Product
4       {
            0 references
5           public int Id { get; set; }
            0 references
6           public string Name {get; set;}
7
8       }
9   }
```

Εικόνα 11

Εν συνεχεία, στον κατάλογο *Data* προσθέτουμε μια νέα *C#* κλάση με τίτλο *StoreContext*.

```
C# StoreContext.cs ×
API > Data > C# StoreContext.cs > ...
1   using Microsoft.EntityFrameworkCore;
2   using API.Entities;
3
4   namespace API.Data
5   {
        1 reference
6       public class StoreContext : DbContext
7       {
            0 references
8           public StoreContext(DbContextOptions<StoreContext> options) : base(options)
9           {
10          }
            0 references
11          public DbSet<Product> Products {get; set;}
12      }
13  }
```

Εικόνα 12

Ο ρόλος της παραπάνω κλάσης είναι ιδιαιτέρως σημαντικός ως προς τη σύνδεση της κλάσης *Product* με την αντίστοιχη βάση δεδομένων. Στην ουσία η κλάση *StoreContext* κληρονομεί τη λειτουργικότητα του *.Net Core Entity Framework* από το γνώρισμα *DbContext* και τη χρήση της βιβλιοθήκης *Microsoft.EntityFrameworkCore*. Με αυτόν τον τρόπο μας επιτρέπεται η αποθήκευση διαφορετικών στιγμιότυπων των κλάσεων που βρίσκονται στον κατάλογο *Entities* (εν προκειμένω της κλάσης *Product*), να συσχετίζονται με την SQL βάση δεδομένων και τέλος να αντλούνται μέσω *Queries,* με σκοπό να αποδοθούν ως HTTP Responses στον Client. Ο Constructor *StoreContext* καθιστά δυνατή τη λήψη του connection string της βάσης δεδομένων *pcparts.db* (θα τη δημιουργήσουμε σε επόμενο βήμα) εντός της κλάσης *StoreContext*. Το Connection String γίνεται ορατό στην *Base Class* (η οποία βρίσκεται στη βιβλιοθήκη του Microsoft.EntityFrameworkCore). Με την *DbSet* property η οποία χρησιμοποιεί την οντότητα *Product* (κλάση *Product* εντός του καταλόγου Entities), δημιουργείται ο πίνακας *Products* και προστίθεται στη βάση δεδομένων, μέσω του *.Net Migrations System*.

1.4 Προσθήκη του Connection String

Στο αρχείο *appsettings.Development.json* προσθέτουμε το *SQL connection string* της *pcparts.db* βάσης δεδομένων της εφαρμογής μας (εικόνα 13).

appsettings.Development.json

API > appsettings.Development.json > ...

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "DefaultConnection": "Data source=pcparts.db"
  }

}
```

Εικόνα 13

Στη συνέχεια, εντός της κλάσης *Startup* κι εντός της μεθόδου *ConfigureServices*, εισάγουμε μια νέα service τύπου *DBContext* η οποία βαζίζεται στην κλάση *StoreContext* που δημιουργήσαμε νωρίτερα. Μέσω της *lambda expression,* το *SQL connection string* της βάσης *pcparts.db* γίνεται διαθέσιμο σε όλα τα projects της εφαρμογή μας (εικόνα 14).

```
// This method gets called by the runtime. Use this method to add services to the container.
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    //specification of connection string via lambda expression
    services.AddDbContext<StoreContext>(x => x.UseSqlite(_config.GetConnectionString("DefaultConnection")));
}
```

Εικόνα 14

1.5 Χρήση του Entity Framework Migrations System

Εντός της κλάσης *StoreContent* έχουμε μια *DbSet Property* η οποία σχετίζεται με την οντότητα *Product* και κατά την εκτέλεση του κώδικα της εικόνας 16, πρόκειται να δημιουργήσει έναν πίνακα με τίτλο *«Products»* εντός της βάσης δεδομένων pcparts.db. Ο πίνακας αυτός θα περιέχει δύο στήλες με βάση τις *Properties* της κλάσης *Product*, δηλαδή μια στήλη με τίτλο *«Id»* κι ακόμη μία με τίτλο *«Name»*. Όλα τα προαναφερθέντα υλοποιούνται αυτόματα κατά την εκτέλεση μιας *Entity Framework Core Migration* διαδικασίας. Αρχικά εγκαθιστούμε το *.NET Entity Framework Tool* στο περιβάλλον του VS Code (εικόνα 15).

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\Spyro\Desktop\Demo\pcparts\API> dotnet tool update --global dotnet-ef --version 6.0.3
Tool 'dotnet-ef' was successfully updated from version '3.1.1' to version '6.0.3'.
PS C:\Users\Spyro\Desktop\Demo\pcparts\API>
```

Εικόνα 15

Σε αυτό το σημείο μπορούμε να εκτελέσουμε την εντολή της εικόνας 16 και με αυτό τον τρόπο δημιουργείται η πρώτη εκδοχή της βάσης δεδομένων της εφαρμογής μας. Στον κατάλογο *Migrations* θα αποθηκεύονται στο εξής όλα τα στιγμιότυπα της *SQL* βάσης δεδομένων του *e-shop* μας.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\Spyro\Desktop\Demo\pcparts\API> dotnet ef migrations add InitialCreate -o Data/Migrations
Build started...
Build succeeded.
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.3 initialized 'StoreContext' using provider 'Microsoft.EntityFrameworkCore.Sqlite:6.0.2' with options: None
Done. To undo this action, use 'ef migrations remove'
PS C:\Users\Spyro\Desktop\Demo\pcparts\API>
```

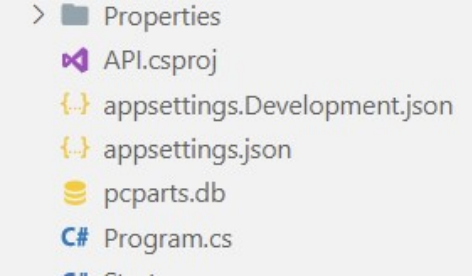
Εικόνα 16

1.6 Ενημέρωση της Βάσης Δεδομένων

Στο τερματικό/*terminal* εκτελούμε την εντολή της εικόνας 17 και προστίθεται στα περιεχόμενα της εφαρμογής το αρχείο που περιέχει τη βάση δεδομένων που δημιουργήσαμε στο προηγούμενο βήμα (εικόνα 18).

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\Spyro\Desktop\Demo\pcparts\API> dotnet ef database update
Build started...
Build succeeded.
```

Εικόνα 17



Εικόνα 18

1.7 Ανάγνωση των Περιεχομένων της Βάσης Δεδομένων μέσω του API Project

Με τη χρήση του *Nuget Package - SQL Server -*, ανοίγουμε τη βάση δεδομένων *pcparts.db* και παρατηρούμε πως περιέχει μόνο έναν κενό πίνακα με τίτλο «*Products*», όπως βλέπουμε στην εικόνα 19

Εικόνα 19

Για να ελέγξουμε τη λειτουργικότητα του *API project* θα πρέπει να εισάγουμε στον πίνακα ορισμένες εγγραφές. Για τον λόγο αυτό εφαρμόζουμε το *query* της εικόνας 20, το οποίο έχει σαν αποτέλεσμα να εισαχθούν 3 εγγραφές στον πίνακα *Products*.

INSERT INTO Products (Id, Name) Untitled-1

```
INSERT INTO Products (Id, Name)
VALUES (1,'Product 1'),
(2,'Product 2'),
(3,'Product 3');
```

SQLite

SQL ▾

Id	Name
1	Product 1
2	Product 2
3	Product 3

Εικόνα 20

Πλέον ο πίνακας *Products* δεν είναι κενός και μπορούμε να αντλήσουμε δεδομένα από αυτόν, με σκοπό να τα συμπεριλάβουμε ως απάντηση σε ένα *HTTP* αίτημα. Για τον λόγο αυτό θα προχωρήσουμε σε ορισμένες αλλαγές εντός της κλάσης *ProductsController*. Αρχικά δημιουργούμε έναν *constructor* της εν λόγω κλάσης που ως όρισμα δέχεται μια μεταβλητή τύπου *StoreContext* όπως φαίνεται στην εικόνα 21. Με αυτόν τον τρόπο έχουμε πρόσβαση στις μεθόδους και στις *properties* της οντότητας *StoreContext* και κατ' επέκταση στη βάση δεδομένων μας.

```
public class ProductsController : ControllerBase
{
    //injection of StoreContext.cs in order to have acces to its methods
    //Constructor of ProductsController
    5 references
    private readonly StoreContext _context;
    0 references
    public ProductsController(StoreContext context)
    {
        _context = context;
    }
```

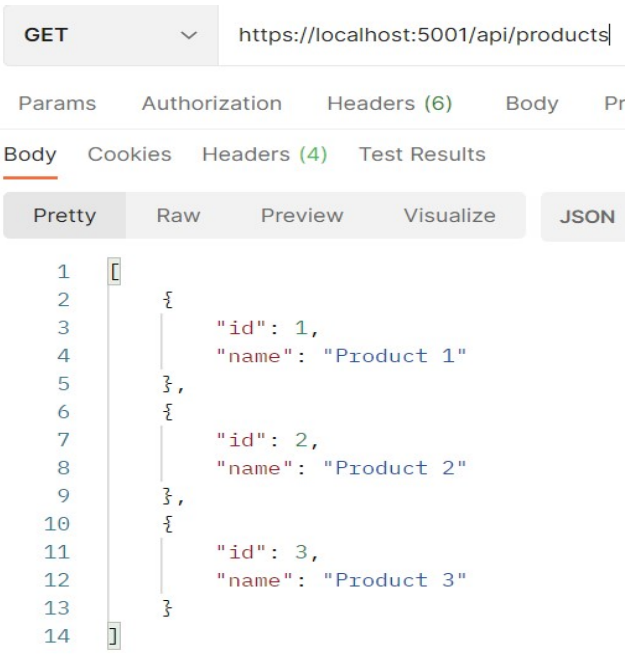
Εικόνα 21

Στη συνέχεια τροποποιούμε τις μεθόδους *GetProducts* και *GetProduct* τις οποίες είχαμε δημιουργήσει στην ενότητα 1.1, ώστε να λειτουργούν ασύγχρονα και να μπορούν να αντλούν δεδομένα από μια βάση τύπου SQL. Η μέθοδος *GetProducts* επιστρέφει μια λίστα με όλα τα δεδομένα του πίνακα *Products* της βάσης *pcparts.db*, ενώ η μέθοδος *GetProdcuct* επιστρέφει την εγγραφή που αντιστοιχεί στο όρισμα *id* που δίνεται από τον client κατά αναζήτηση του *URL* (εικόνα 22).

```
[HttpGet]
0 references
public async Task<ActionResult<List<Product>>> GetProducts()
{
    var products = await _context.Products.ToListAsync();
    return Ok(products);
}

[HttpGet("{id}")]
0 references
public async Task<ActionResult<Product>> GetProducts(int id)
{
    var products = await _context.Products.ToListAsync();
    return await _context.Products.FindAsync(id);
}
```

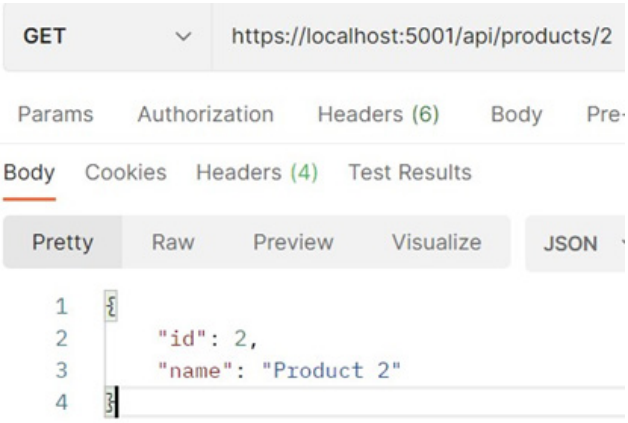
Εικόνα 22

Κατά την αναζήτηση του URL: *https://localhost:5001/api/products* στην εφαρμογή *Postman* εμφανίζονται όλες οι εγγραφές του πίνακα *Products* ως απάντηση (εικόνα 23).



Εικόνα 23

Αντίστοιχα όταν αναζητήσουμε το URL: *https://localhost:5001/api/products/2* με όρισμα τον αριθμό 2, επιστέφεται μόνο η εγγραφή με *Id* που ισούται με τον αριθμό 2, όπως βλέπουμε στην εικόνα 24.



Εικόνα 24

1.8 Δημιουργία των επιπλέων Projects

Ως τελευταία ενέργεια σε το κεφάλαιο, μας έχει απομείνει η προσθήκη του *Infrastructure project* και του *Core project*. Όπως ανέφερα και στην αρχή του κεφαλαίου, η *Web* εφαρμογή μας αποτελείται από 3 βασικά *projects* τα οποία είναι αλληλένδετα (εικόνα 1), αυτό του *API project*, του *Infrastructure project* και αυτό του *Core project*. Για τη δημιουργία λοιπόν, των δύο εναπομεινάντων *projects* ορίζουμε δύο νέες *class libraries*, όπως βλέπυμε και στις εντολές της εικόνας 25, με τα αντίστοιχα ονόματα που προαναφέρα.

```
PS C:\Users\Spyro\Desktop\Demo\pcparts> dotnet new classlib -o Core
The template "Class Library" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Users\Spyro\Desktop\Demo\pcparts\Core\Core.csproj...
  Determining projects to restore...
  Restored C:\Users\Spyro\Desktop\Demo\pcparts\Core\Core.csproj (in 50 ms).
Restore succeeded.

PS C:\Users\Spyro\Desktop\Demo\pcparts> dotnet new classlib -o Infrastructure
The template "Class Library" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Users\Spyro\Desktop\Demo\pcparts\Infrastructure\Infrastructure.csproj...
  Determining projects to restore...
  Restored C:\Users\Spyro\Desktop\Demo\pcparts\Infrastructure\Infrastructure.csproj (in 46 ms).
Restore succeeded.
```

Εικόνα 25

Για να προσθέσουμε τα 2 νέα projects στο *pcparts solution*, εκτελούμε τις εντολές της εικόνας 26.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\Spyro\Desktop\Demo\pcparts> dotnet sln add Core
Project `Core\Core.csproj` added to the solution.
PS C:\Users\Spyro\Desktop\Demo\pcparts> dotnet sln add Infrastructure
Project `Infrastructure\Infrastructure.csproj` added to the solution.
```

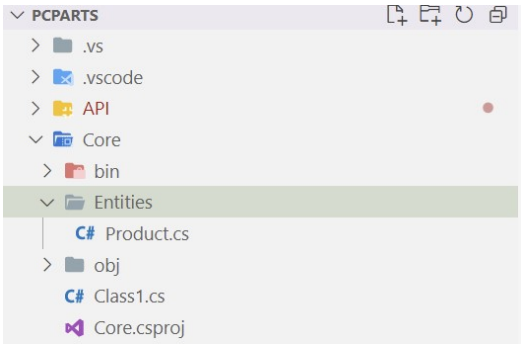
Εικόνα 26

Όπως ανέφερα στην εισαγωγή της διπλωματικής εργασίας σχετικά με τη δομή της εφαρμογής, το *API project* που δημιουργήσαμε έχει άμεση εξάρτηση από το *Infrastructure project* και αυτό με τη σειρά του από το *Core project*. Προκειμένου να ορίσω αυτές τις εξαρτήσεις και να γίνουν διαθέσιμες εντός του *pcparts solution*, εκτελώ τις εντολές της εικόνας 27.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\Spyro\Desktop\Demo\pcparts> cd API
PS C:\Users\Spyro\Desktop\Demo\pcparts\API> dotnet add reference ../Infrastructure
Reference `..\Infrastructure\Infrastructure.csproj` added to the project.
PS C:\Users\Spyro\Desktop\Demo\pcparts\API> cd..
PS C:\Users\Spyro\Desktop\Demo\pcparts> cd Infrastructure
PS C:\Users\Spyro\Desktop\Demo\pcparts\Infrastructure> dotnet add reference ../Core
Reference `..\Core\Core.csproj` added to the project.
PS C:\Users\Spyro\Desktop\Demo\pcparts\Infrastructure> cd..
PS C:\Users\Spyro\Desktop\Demo\pcparts> dotnet restore
  Determining projects to restore...
  Restored C:\Users\Spyro\Desktop\Demo\pcparts\Infrastructure\Infrastructure.csproj (in 83 ms).
  Restored C:\Users\Spyro\Desktop\Demo\pcparts\API\API.csproj (in 223 ms).
  1 of 3 projects are up-to-date for restore.
PS C:\Users\Spyro\Desktop\Demo\pcparts>
```

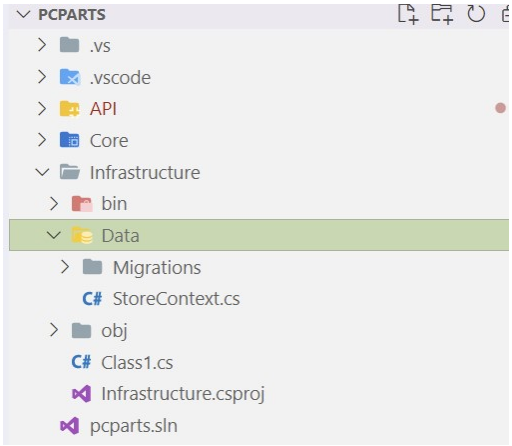
Εικόνα 27

Το ***Core project*** θα είναι υπεύθυνο για τις οντότητες (*entities*) της εφαρμογής, για τον λόγο αυτό θα μετακινήσουμε τον κατάλογο *entities* που βρίσκεται εντός του *API project* στον κατάλογο του Core project (εικόνα 28).



Εικόνα 28

Αντιστοίχως το *Infrastructure project* θα είναι υπεύθυνο στο να επικοινωνεί με τη βάση δεδομένων της εφαρμογής και να επιστρέφει απαντήσεις στο *API project*. Για τον λόγο αυτό θα μετακινήσω τον κατάλογο *Data*, ο οποίος βρίσκεται εντός του *API project*, σε αυτόν του *Infrastructure project* όπως εμφανίζεται και στην εικόνα 29.

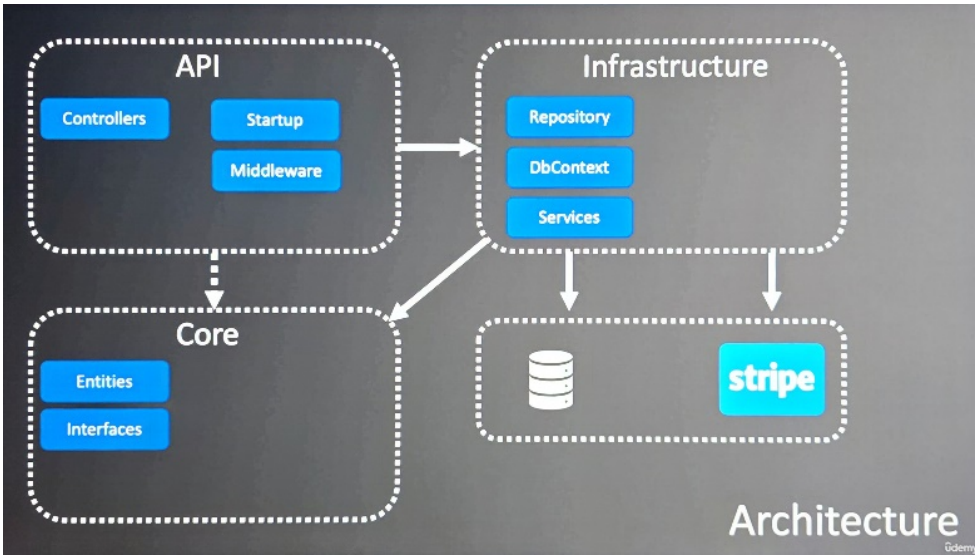


Εικόνα 29

Η αρχική δομή υλοποίησης κι ανάπτυξης της *Web* εφαρμογής μας είναι έτοιμη. Στο επόμενο κεφάλαιο θα μελετήσουμε την αρχιτεκτονική του *API project*.

Κεφάλαιο 2. Η Αρχιτεκτονική του API Project

Σε αυτό το κεφάλαιο θα επικεντρωθούμε στην αρχιτεκτονική του *API* project. Συγκεκριμένα, θα ασχοληθούμε με το κομμάτι του *ASP.Net Core Repository pattern*, το οποίο περιέχει τους απαραίτητους μηχανισμούς με τους οποίους θα μπορούμε να αποθηκεύουμε, αλλά και να αντλούμε δεδομένα από τη βάση δεδομένων μας, δια μέσου των controller κλάσεων του *API project* και του *entity framework Core*. Επίσης θα επεκτείνουμε τη λειτουργικότητα της οντότητας *Product* εντός του *Core project*, με το να προσθέσουμε νέες properties, με στόχο να εμπλουτίσουμε τα χαρακτηριστικά των προϊόντων μας. Τέλος, θα δημιουργήσουμε τους απαραίτητους μηχανισμούς προκειμένου να αυτοματοποιήσουμε τον τρόπο μαζικής εισαγωγής εγγραφών στους πίνακες της βάσης δεδομένων, μεταφέροντας τον κατάλογο *Migrations* εντός της κλάσης *Startup*. Με τη λογική αυτή κάθε φορά που εκκινούμε την εφαρμογή μας θα πραγματοποιείται έλεγχος μέσω του *Entity Framework Core* για το αν υπάρχουν ή όχι δεδομένα στην *SQL* βάση δεδομένων της εφαρμογής μας (*pcparts.db*) και στην περίπτωση μη ύπαρξης εγγραφών, τότε θα εκκινείται η διαδικασία της μαζικής εισαγωγής (*Seeding Data Process*).



Εικόνα 30

- **API – Startup app**: Περιέχει τον *ProductsController* και κατ'επέκταση το *DI container* και τις 2 κλάσεις *middleware*. Είναι υπεύθυνο για το *routing* των *HTTP* αιτημάτων.

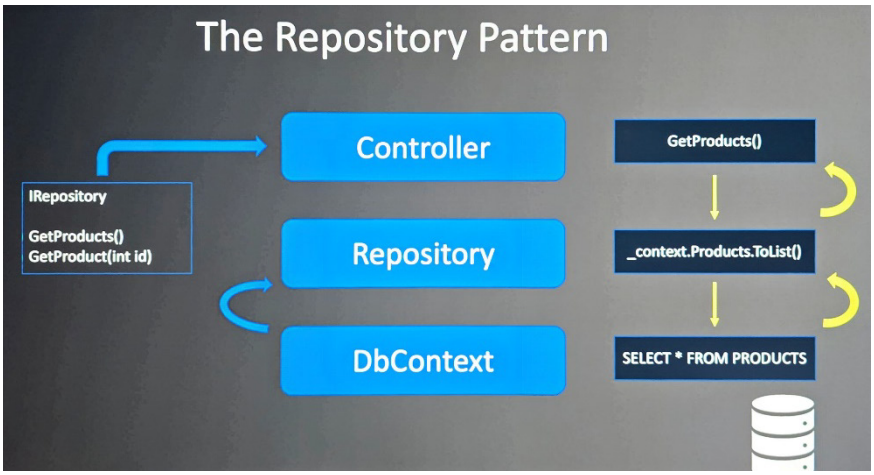
- **Infrastructure** – Περιέχει τις *DB context* κλάσεις, τις απαραίτητες *services* για την αποστολή/εκτέλεση των *queries* στη βάση δεδομένων.

- **Core** – περιέχει τις κλάσεις (*Entities*) και τα *interfaces* των *DI containers*.

2.1 Repository Pattern

Το *Repository Pattern* είναι ένα μοτίβο σχεδίασης το οποίο αφαιρεί τη λογική πρόσβασης δεδομένων, λειτουργώντας ως ενδιάμεσος μεταξύ ενός μοντέλου τομέα και των επιπέδων αντιστοίχισης δεδομένων. Σκοπός της χρήσης του *Repository Pattern* είναι η εκμετάλλευση των ιδιοτήτων του στις κλάσεις τύπου *Controller* του *API project* καθώς επίσης και στις κλάσεις τύπου *DBcontext* του *Core project*. Προς στιγμήν

έχουμε μία *ProductsController* κλάση η οποία περιλαμβάνει δύο μεθόδους, την *Getproducts* και την *GetProduct(int id).* Το *Repository pattern* θα έχει πρόσβαση σε αυτές τις μεθόδους, θα διαχωρίσει τον κώδικά τους από τον controller, θα τις συνδέσει με την κλάση *StoreContext* και αυτό θα έχει ως αποτέλεσμα μία καλύτερα δομημένη και πιο αποδοτική *web* εφαρμογή. Η εικόνα 31 αποτυπώνει τη συγκεκριμένη λογική.



Εικόνα 31

2.2 Προσθήκη του Repository Pattern και του Interface του

Στο Core project προσθέτουμε έναν νέο κατάλογο με τίτλο *Interfaces* και εντός του μια κλάση με τίτλο *IProductRepository* τύπου interface (εικόνα 32). Οι δύο ασύγχρονες μέθοδοι θα χρησιμοποιηθούν από την *Repository Pattern* κλάση *ProductRepository* την οποία θα δημιουργήσω στο επόμενο βήμα, με σκοπό την εμφάνιση των εγγραφών του πίνακα *products* της *pcparts.db* στον *client*.

```
1  using Core.Entities;
2  using System.Collections.Generic;
3  using System.Threading.Tasks;
4
5  namespace Core.Interfaces
6  {
       0 references
7      public interface IProductRepository
8      {
           0 references
9          Task<Product>GetProductByIdAsync(int id);
           0 references
10         Task<IReadOnlyList<Product>>GetProductsAsync();
11     }
12 }
```

Εικόνα 32

Ως επόμενο βήμα, έχουμε τη δημιουργία της *Repository Pattern* κλάσης εντός του *Infrastructure project*. Εντός του καταλόγου *Data* θα φτιάξουμε μια νέα κλάση με τίτλο *ProductRepository* κι εκεί θα χρησιμοποιήσουμε (implement) το *IProductRepository interface* ώστε να παραχθεί ο κώδικας της εικόνας 33.

```
namespace Infrastructure.Data
{
    1 reference
    public class ProductRepository : IProductRepository
    {
        0 references
        public Task<Product> GetProductByIdAsync(int id)
        {
            throw new NotImplementedException();
        }

        0 references
        public Task<IReadOnlyList<Product>> GetProductsAsync()
        {
            throw new NotImplementedException();
        }
    }
}
```

Εικόνα 33

Προκειμένου η κλάση *ProductRepository* να μπορεί να αξιοποιηθεί από τους *Controllers* του *API project* θα προσθέσουμε μια νέα service εντός της *startup* κλάσης όπως φαίνεται στην εικόνα 34.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    //specification of connection string via lambda expression
    services.AddDbContext<StoreContext>(x => x.UseSqlite(_config.GetConnectionString("DefaultConnection")));
    services.AddScoped<IProductRepository, ProductRepository>();
}
```

Εικόνα 34

Για να είναι λειτουργική η μεθοδολογία του *Repository Pattern*, δημιουργούμε τις μεθόδους εντός της κλάσης *ProductRepository* όπως φαίνεται στην εικόνα 35, με τη λογική που το πράξαμε στο προηγούμενο κεφάλαιο εντός της κλάσης *ProductsController*, ώστε να γίνει εφικτή η αλληλεπίδρασή της με τους *Controllers* του *API project* και την κλάση *StoreContext*, εντός του Infrastructure project.

```
using Core.Entities;
using Core.Interfaces;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Data
{
    1 reference
    public class ProductRepository : IProductRepository
    {
        3 references
        private readonly StoreContext _context;
        0 references
        public ProductRepository(StoreContext context)
        {
            _context=context;
        }

        0 references
        public async Task<Product> GetProductByIdAsync(int id)
        {
            return await _context.Products.FindAsync(id);
        }

        0 references
        public async Task<IReadOnlyList<Product>> GetProductsAsync()
        {
            return await _context.Products.ToListAsync();
        }
    }
}
```

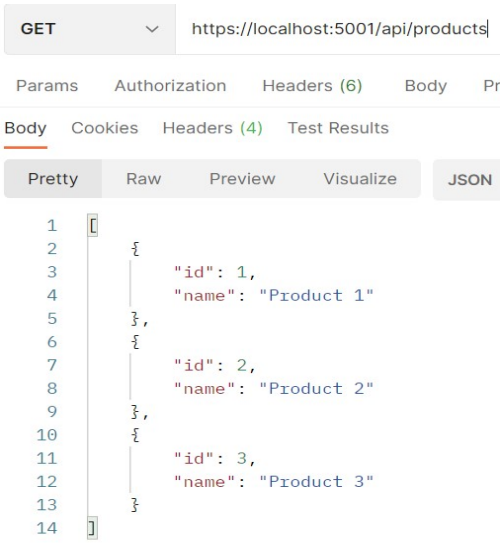
Εικόνα 35

Από πλευράς των Controllers, εντός της κλάσης *ProductsController* θα πρέπει να αντικαταστήσουμε όλες τις κλήσεις της κλάσης *StoreContext* με αυτές του *IProductRepository interface* όπως φαίνεται και στην εικόνα 36, ώστε και οι *Controllers* της εφαρμογής να μπορούν να κάνουν χρήση του μοτίβου *Repository Pattern*.

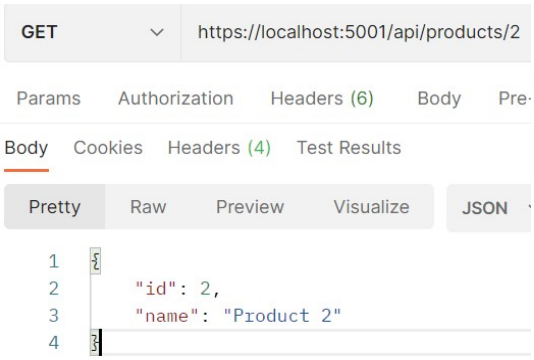
```
namespace API.Controllers
{
    [ApiController]
    [Route("api/[controller]")]

    0 references
    public class ProductsController : ControllerBase
    {
        3 references
        private readonly IProductRepository _repo;
        0 references
        public ProductsController(IProductRepository repo)
        {
            _repo = repo;
        }

        [HttpGet]
        0 references
        public async Task<ActionResult<List<Product>>> GetProducts()
        {
            var products = await _repo.GetProductsAsync();
            return Ok(products);
        }

        [HttpGet("{id}")]
        0 references
        public async Task<ActionResult<Product>> GetProduct(int id)
        {
            return await _repo.GetProductByIdAsync(id);
        }
    }
}
```

Εικόνα 36

Αναζητώντας το URL: *https://localhost:5001/api/products* στην εφαρμογή *Postman*, εμφανίζονται όλες οι εγγραφές του πίνακα *Products* ως απάντηση (εικόνα 36Α) . Αντίστοιχα όταν αναζητήσουμε το URL: *https://localhost:5001/api/products/2* με όρισμα τον αριθμό 2, επιστέφεται μόνο η εγγραφή με *Id* που ισούται με τον αριθμό 2, όπως βλέπουμε στην εικόνα 36Β, όπως ακριβώς συνέβη και στο προηγούμενο κεφάλαιο.



Εικόνα 36Α



Εικόνα 36Β

2.3 Επέκταση της Λειτουργικότητας της Οντότητας Product

Μετά και την επιτυχή προσθήκη του *Repository Pattern*, έχει έρθει η στιγμή να επεκτείνουμε την κλάση *product*. Στην ουσία θα προσθέσουμε επιπλέον στήλες στον πίνακα *products* της βάση δεδομένων. Στην εικόνα 37 παρουσιάζεται η προσθήκη μιας νέας οντότητας εντός του *Infrastructure project* κι εντός του καταλόγου *Entities*. Η νέα κλάση ονομάζεται *BaseEntity* και θα φιλοξενήσει την *property Id* της κλάσης *Product*.

```
namespace Core.Entities
{
    0 references
    public class BaseEntity
    {
        0 references
        public int Id{get; set;}
    }
}
```

Εικόνα 37

Οι νέες *properties* της οντότητας *product*, χωρίς την property *Id* πλέον, μιας και τη μετακινήσαμε στην κλάση *BaseEntity*, αποτελούν επιπλέον χαρακτηριστικά για την περιγραφή των προϊόντων και παρουσιάζονται στην εικόνα 38.

```
1   namespace Core.Entities
2   {
        10 references
3       public class Product:BaseEntity
4       {
                    1 reference
5                   public string Name {get; set;}
                    1 reference
6                   public string Description {get; set;}
                    1 reference
7                   public decimal Price {get; set;}
                    1 reference
8                   public string PictureUrl {get; set;}
                    1 reference
9                   public ProductType ProductType {get; set;}
                    1 reference
.0                  public int ProductTypeId {get; set;}
                    1 reference
.1                  public ProductBrand ProductBrand {get; set;}
                    1 reference
.2                  public int ProductBrandId {get; set;}
.3      }
.4  }
```

Εικόνα 38

Όπως φαίνεται και στην εικόνα 38, υπάρχουν δύο *properties* τύπου *ProductType* και *ProductBrand*, για τις οποίες και θα δημιουργήσουμε τις αντίστοιχες κλάσεις - οντότητες εντός του καταλόγου *Entities*, όπως δείχνουν οι εικόνες 39 και 40.

```
namespace Core.Entities
{
    1 reference
    public class ProductType:BaseEntity
    {
        0 references
        public string Name {get; set;}
    }
}
```

Εικόνα 39

```
namespace Core.Entities
{
    1 reference
    public class ProductBrand:BaseEntity
    {
        0 references
        public string Name {get; set;}
    }
}
```

Εικόνα 40

Ο λόγος που δημιουργήσαμε αυτές τις *properties* με τα αντίστοιχα *Ids* τους, είναι για να προσδιορίσουμε στον μηχανισμό του *Entity Framework* τον τρόπο που θα εφαρμόζονται μεμονωμένα *queries* για αυτές. Με πιο απλά λόγια, θέλουμε να δώσουμε στον *client* τη δυνατότητα να εμφανίζει τα προϊόντα ανά τύπο ή ανά κατηγορία εάν το επιθυμεί, με χρήση ανάλογου *Http* αιτήματος. Για να συμβεί αυτό θα πρέπει να δημιουργήσουμε δύο *DBSet* properties στην κλάση *StoreContext*, με σκοπό να εμπλουτίσουμε τη βάση δεδομένων με τους δύο νέους πίνακες, αυτόν του *ProductTypes* και αυτόν του *ProductBrands* όπως φαίνεται στην εικόνα 41.

```
namespace Infrastructure.Data
{
    6 references
    public class StoreContext : DbContext
    {
        0 references
        public StoreContext(DbContextOptions<StoreContext> options) : base(options)
        {
        }
        2 references
        public DbSet<Product> Products {get; set;}
        0 references
        public DbSet<ProductBrand> ProductBrands {get; set;}
        0 references
        public DbSet<ProductType> ProductTypes {get; set;}
    }
}
```

Εικόνα 41

2.4 Εφαρμογή μιας Νέας Migration

Για να εφαρμοστούν οι αλλαγές που πραγματοποιήσαμε στο προηγούμενο βήμα, θα πρέπει να σβήσουμε τη βάση δεδομένων που δημιουργήσαμε, καθώς επίσης και την αρχική *migration* και να εφαρμόσουμε μια νέα. Προϋπόθεση για να σβήσουμε τη βάση δεδομένων, είναι να προσδιορίσουμε το *project (Infrastructure)* στο οποίο βρίσκεται η κλάση *StoreContext* και το *startup project*, δηλαδή το API όπως φαίνεται και στην εικόνα 42.

```
PS C:\Users\Spyro\Desktop\Demo\pcparts> dotnet ef database drop -p Infrastructure -s API
Build started...
Build succeeded.
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.3 initialized 'StoreContext' using provider 'Microsoft.EntityFrameworkCore.Sqlite:6.0.2' with options: None
Are you sure you want to drop the database 'main' on server 'pcparts.db'? (y/N)
y
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.3 initialized 'StoreContext' using provider 'Microsoft.EntityFrameworkCore.Sqlite:6.0.2' with options: None
Dropping database 'main' on server 'pcparts.db'.
Successfully dropped database 'main'.
PS C:\Users\Spyro\Desktop\Demo\pcparts>
```

Εικόνα 42

Για να σβήσουμε την αρχική *migration* θα πρέπει εκ νέου να προσδιορίσουμε το *Infrastructure project*, διότι εκεί αποθηκεύονται τα στιγμιότυπα της βάσης μας (κατάλογος *Migrations*), καθώς επίσης και το *startup project* που είναι το API. Ο κώδικας που εφαρμόζουμε παρουσιάζεται στην εικόνα 43.

```
PS C:\Users\Spyro\Desktop\Demo\pcparts> dotnet ef migrations remove -p Infrastructure -s API
Build started...
Build succeeded.
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.3 initialized 'StoreContext' using provider 'Microsoft.EntityFrameworkCore.Sqlite:6.0.2' with options: None
Removing migration '20220309231143_InitialCreate'.
Removing model snapshot.
Done.
PS C:\Users\Spyro\Desktop\Demo\pcparts>
```

Εικόνα 43

Το μόνο που πρέπει να κάνουμε τώρα, είναι να εφαρμόσουμε μια νέα *migration* ώστε να δημιουργηθεί από την αρχή ένα στιγμιότυπο της βάσης δεδομένων με τα νέα χαρακτηριστικά και τους πίνακες που δημιουργήσαμε. Για να συμβεί κάτι τέτοιο εκτελούμε την εντολή της εικόνας 44. Εδώ προσδιορίσαμε το *project* στο οποίο βρίσκεται η κλάση *StoreContext*, το project που έχουμε ορίσει ως *startup* κι επιπλέον τον κατάλογο που θα αποθηκευτεί το εν λόγω *migration* στιγμιότυπο.

```
PS C:\Users\Spyro\Desktop\Demo\pcparts> dotnet ef migrations add InitiaCreate -p InfraStructure -s API -o Data/Migrations
Build started...
Build succeeded.
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.3 initialized 'StoreContext' using provider 'Microsoft.EntityFrameworkCore.Sqlite:6.0.2' with options: None
Done. To undo this action, use 'ef migrations remove'
PS C:\Users\Spyro\Desktop\Demo\pcparts>
```

Εικόνα 44

2.5 Δημιουργία της Βάσης Δεδομένων Εντός της Κλάσης Startup

Στο προηγούμενο βήμα προσθέσαμε μια νέα *migration* για τη βάση δεδομένων *pcparts.db*. Σε αυτό το βήμα θα πρέπει να δημιουργήσουμε το αρχείο της βάσης δεδομένων από την αρχή, μόνο που αυτήν τη φορά δεν θα χρησιμοποιήσουμε τις εντολές του *Entity Framework Core*. Σκοπός μας είναι να δημιουργήσουμε την κατάλληλη ακολουθία κώδικα εντός της κλάσης *Program*, η οποία βρίσκεται εντός του *API* project, ώστε κάθε φορά που εκκινούμε την εφαρμογή να γίνεται έλεγχος για το αν εφαρμόστηκε μια νέα Migration.

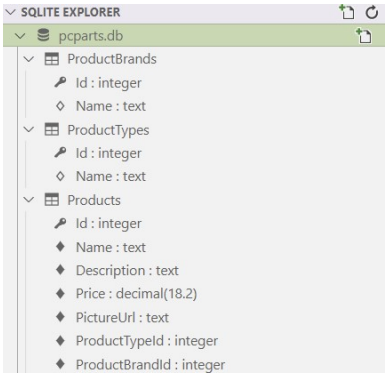
Σε περίπτωση που εφαρμοστεί μια νέα *migration* τότε εκτελείται ο κώδικας δημιουργίας του αρχείου της βάσης δεδομένων. Σύμφωνα λοιπόν με τα πρότυπα της *Microsoft* για να δημιουργήσουμε μια τέτοια λειτουργικότητα, θα πρέπει να ορίσουμε μια νέα μεταβλητή με όνομα *host* και να της αναθέσουμε τη μέθοδο *build*. Στη συνέχεια για να αποκτήσουμε πρόσβαση στην κλάση *StoreContext*, θα πρέπει να κάνουμε χρήση του *Using* statement. Το συγκεκριμένο *statement* θα δέχεται ως όρισμα μια μεταβλητή που θα ονομάζεται *scope* και θα της ανατίθεται το χρονικό περιθώριο κατά το οποίο θα εκτελείται ασύγχρονα η όλη διαδικασία δημιουργίας της βάσης. Αυτό γίνεται μέσω των *services* και του *Dependency Injection* από τις βιβλιοθήκες της *Microsoft*. Για να κάνουμε χρήση των services μέσω της μεταβλητής scope, θα ορίσουμε τη μεταβλητή *services*. Η μεταβλητή *loggerFactory* από την άλλη, θα μας χρησιμεύσει στο να δημιουργήσουμε *instances* της κλάσης *ilogger*, με σκοπό να παράξουμε μηνύματα σφαλμάτων στην περίπτωση που αποτύχει η διαδικασία. Κατόπιν, θα δημιουργήσουμε μία *try-catch* δομή εντός της οποίας θα κάνουμε χρήση της μεταβλητής *context*, ώστε να δημιουργήσουμε ένα *instance* της κλάσης *StoreContext* μέσω των *Microsoft* services κι εν τέλει με τρόπο ασύγχρονο να εφαρμόσουμε την *migration* διαδικασία που εκκρεμεί με χρήση της μεθόδου *MigrateAsync*, με τελικό στόχο τη δημιουργία του νέου αρχείου της βάσης δεδομένων. Σε περίπτωση που προκύψει κάποιο σφάλμα-εξαίρεση (*exception*), θα προβληθεί το κατάλληλο μήνυμα το οποίο θα περιλαμβάνει και τον κωδικό σφάλματος. Ο κωδικός σφάλματος αντλείται μέσω της μεταβλητής *logger* και προβάλλεται μέσω της μεθόδου *LogError*. Ο κώδικας

που περιέγραψα παρουσιάζεται στην εικόνα 45 κι εκτελείται εντός της μεθόδου *Main* με την εντολή *host.Run()*.

```
public class Program
{
    0 references
    public static async Task Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();
        using(var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;
            var loggerFactory = services.GetRequiredService<ILoggerFactory>();
            try
            {
                var context = services.GetRequiredService<StoreContext>();
                await context.Database.MigrateAsync();
            }
            catch (Exception ex)
            {
                var logger = loggerFactory.CreateLogger<Program>();
                logger.LogError(ex, "An exception occured during migration");
            }
        }
        host.Run();
    }
```

Εικόνα 45

Αφού εκκινήσουμε την εφαρμογή, εφαρμόζεται η εκκρεμής *migration* και δημιουργείται η βάση δεδομένων *pcparts.db* εκ νέου και με χρήση του SQL explorer βλέπουμε το περιεχόμενο της στην εικόνα 46. Παρατηρούμε πως η βάση δεδομένων περιέχει δύο νέους πίνακες, έναν με τίτλο *ProductBrands* κι έναν με τίτλο *ProductTypes*.



Εικόνα 46

2.6 Προσθήκη και Άντληση Εγγραφών από την Βάση Δεδομένων

Εντός του *Infrastructure project* και του καταλόγου *Data*, προσθέτουμε τον κατάλογο *SeedData*. Αφού δημιουργήσουμε τρία αρχεία *products.json, types.json, brands.json*, με όλη την απαραίτητη πληροφορία (προϊόντα, τύπους, μάρκες) το προσθέτουμε στον συγκεκριμένο κατάλογο. Κάνοντας χρήση του τερματικού/*terminal* χρησιμοποιούμε την εντολή *dotnet ef database drop -s API -p* και σβήνουμε τις εγγραφές της βάσης δεδομένων. Εντός του *Infrastructure project* δημιουργούμε μια κλάση με τίτλο

StoreContextSeed. Προσθέτουμε τη μέθοδο *SeedAsync* η οποία δέχεται ως ορίσματα μια μεταβλητή τύπου *StoreContext*, ώστε να δημιουργήσουμε ένα νέο *instance* της κλάσης *StoreContext* κι ένα τύπου *ILoggerFactory*, ώστε να προσδιορίσουμε σφάλματα που ενδέχεται να προκύψουν κατά την εκτέλεση του *Task*. Στη συνέχεια εντός της μεθόδου *SeedAsync*, δημιουργούμε ένα *try-catch block* και πραγματοποιούμε ελέγχους στους πίνακες της βάσης δεδομένων για το εάν έχουν περιεχόμενα. Σε περίπτωση που οι πίνακες είναι κενοί (όπως συμβαίνει τη δεδομένη χρονική στιγμή), κάνουμε χρήση του ανάλογου *SeedData.json* αρχείου και το αναθέτουμε σε μια μεταβλητή (*ProductsData*, *BrandsData*, *TypesData*). Κατόπιν κάνουμε *Deserialize* το αρχείο *json* σε μορφή λίστας μέσω των μεταβλητών *ProductsData*, *TypesData*, *BrandsData*, τα οποίο για στήλες έχουν τα πεδία του κάθε πίνακα της βάσης δεδομένων και τις αναθέτουμε σε μια μεταβλητή, ανάλογα με το περιεχόμενο των πινάκων που θέλουμε να ενημερώσουμε (*products*, *brands*, *types*). Τέλος, με ένα *foreach loop* γεμίζουμε τον κάθε πίνακα της βάσης δεδομένων γραμμή-γραμμή και αποθηκεύουμε τις αλλαγές. Στις εικόνες 47 και 48 παρουσιάζεται αναλυτικά η κλάση *StoreContextSeed*.

```
public class StoreContextSeed
{
    //use STATIC inside the class without a new instance to be required in order to use its method
    1 reference
    public static async Task SeedAsync(StoreContext context, ILoggerFactory loggerFactory)
    {
        try
        {
            if(!context.ProductBrands.Any())
            {
                var brandsData = File.ReadAllText("../Infrastructure/Data/SeedData/brands.json");
                var brands = JsonSerializer.Deserialize<List<ProductBrand>>(brandsData);

                foreach(var item in brands)
                {
                    context.ProductBrands.Add(item);
                }

                await context.SaveChangesAsync();
            }
```

Εικόνα 47

```
            if(!context.ProductTypes.Any())
            {
                var typesData = File.ReadAllText("../Infrastructure/Data/SeedData/types.json");
                var types = JsonSerializer.Deserialize<List<ProductType>>(typesData);

                foreach(var item in types)
                {
                    context.ProductTypes.Add(item);
                }

                await context.SaveChangesAsync();
            }

            if(!context.Products.Any())
            {
                var productsData = File.ReadAllText("../Infrastructure/Data/SeedData/products.json");
                var products = JsonSerializer.Deserialize<List<Product>>(productsData);

                foreach(var item in products)
                {
                    context.Products.Add(item);
                }

                await context.SaveChangesAsync();
            }
        }
        catch(Exception ex)
        {
            var logger = loggerFactory.CreateLogger<StoreContextSeed>();
            logger.LogError(ex.Message,"ERROR ON SEEDING DATA!");
        }
    }
}
```

Εικόνα 48

Προκειμένου να κάνουμε χρήση της κλάσης *StoreContextSeed* κάθε φορά που εκκινεί η εφαρμογή μας, θα πρέπει εντός του *try-catch block* της κλάσης *Program*, το οποίο και διαμορφώσαμε στην προηγούμενη ενότητα, να δημιουργήσουμε ένα νέο ασύγχρονο *task*. Ο κώδικας του συγκεκριμένου *task* παρουσιάζεται στην εικόνα 49.

```
try
{
    var context = services.GetRequiredService<StoreContext>();
    await context.Database.MigrateAsync();
    await StoreContextSeed.SeedAsync(context, loggerFactory);
}
```

Εικόνα 49

Όταν εκκινήσουμε την εφαρμογή μας, καλείται το παραπάνω *task* και αφού γίνουν οι κατάλληλοι έλεγχοι εντός της κλάσης *StoreContextSeed* ξεκινάει η εισαγωγή στοιχείων στους πίνακες της βάσης δεδομένων. Ενδεικτικά στην εικόνα 50 παρουσιάζεται μέρος των περιεχόμενων του πίνακα *products* της *pcparts.db*.

SQL ▾ 1 / 1 1 - 18 of 18

Id	Name	Description	Price	PictureUrl	ProductTypeId	ProductBrandId
1	Intel Core i7-12700K 2.7GHz	12 Core Desktop CPU - Socket 1700 in a BOX	419.99	images/products/12700k.png	1	1
2	Intel Core i9-12900K 2.4GHz	16 Core Desktop CPU - Socket 1700 in a BOX	599.99	images/products/12900k.png	1	1
3	AMD Ryzen 7 5800X 3.8GHz	8 Core Desktop CPU - Socket AM4 in a BOX	345	images/products/5800x.png	1	2
4	AMD Ryzen 9 5950X 3.4GHz	16 Core Desktop CPU - Socket AM4 in a BOX	649	images/products/5950x.png	1	2
5	Asus GeForce RTX 3070 Ti 8GB GDDR6X ROG Strix OC	Graphics Card PCI-E x16 4.0 with 2 HDMI and 3 DisplayPort	1073.64	images/products/ASUS_rtx_3070ti.png	2	3

Εικόνα 50

Στην εφαρμογή *Postman* αναζητούμε το URL: https://localhost:5001/api/products και λαμβάνουμε ως απάντηση το περιεχόμενο της εικόνας 51. Η επιτυχής μαζική εισαγωγή εγγραφών στους πίνακες της *pcparts.db*, αλλά και η άντλησή τους μέσω *HTTP* αιτημάτων είναι γεγονός!

GET https://localhost:5001/api/products

Params Authorization Headers (6) Body Pre-request Script Tests Settings

ody Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "name": "Intel Core i7-12700K 2.7GHz",
        "description": "12 Core Desktop CPU - Socket 1700 in a BOX",
        "price": 419.99,
        "pictureUrl": "images/products/12700k.png",
        "productType": null,
        "productTypeId": 1,
        "productBrand": null,
        "productBrandId": 1,
        "id": 1
```


Εικόνα 51

Όσον αφορά την άντληση στοιχείων από τους πίνακες *ProductBrands* και *ProductTypes*, θα πρέπει αρχικά να δημιουργήσουμε δύο νέα *tasks* εντός του *interface IProductRepository*, όπως φαίνεται στην εικόνα 52.

```
4 references
public interface IProductRepository
{
    1 reference
    Task<Product>GetProductByIdAsync(int id);
    1 reference
    Task<IReadOnlyList<Product>>GetProductsAsync();
    0 references
    Task<IReadOnlyList<ProductBrand>>GetProductBrandsAsync();
    0 references
    Task<IReadOnlyList<ProductType>>GetProductTypesAsync();
}
```

Εικόνα 52

Στη συνέχεια θα πρέπει να δημιουργήσουμε δύο νέα ασύγχρονα *tasks* στην κλάση *ProductRepository* ώστε να εφαρμόζονται *queries* στους πίνακες που επιθυμούμε μέσω του *Repository Pattern* (Εικόνα 53).

```
0 references
public async Task<IReadOnlyList<ProductType>> GetProductTypesAsync()
{
      return await _context.ProductTypes.ToListAsync();
}

0 references
public async Task<IReadOnlyList<ProductBrand>> GetProductBrandsAsync()
{
      return await _context.ProductBrands.ToListAsync();
}
```

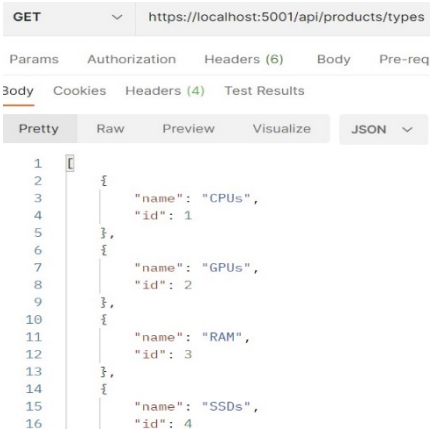
Εικόνα 53

Για να μπορέσουμε να εφαρμόσουμε ένα *HTTP request* που θα αντλεί δεδομένα και θα μας επιστρέφει ως απάντηση περιεχόμενα των δύο πινάκων, θα πρέπει εντός της κλάσης *ProductsControler* να δημιουργήσουμε τις αντίστοιχες μεθόδους. Στην εικόνα 54 παρουσιάζεται ο κώδικας των μεθόδων *GetProductBrands* και *GetProductTypes*.

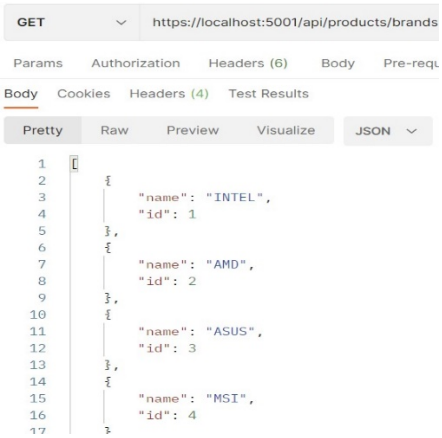
```
    [HttpGet("brands")]
    0 references
    public async Task<ActionResult<IReadOnlyList<ProductBrand>>> GetProductBrands()
    {
        return Ok(await _repo.GetProductBrandsAsync());
    }

    [HttpGet("types")]
    0 references
    public async Task<ActionResult<IReadOnlyList<ProductType>>> GetProductTypes()
    {
        return Ok(await _repo.GetProductTypesAsync());
    }
}
```

Εικόνα 54

Στην εφαρμογή *Postman* αναζητούμε τα URLs: https://localhost:5001/api/products/types ώστε να εμφανιστούν στην οθόνη οι τύποι των προϊόντων της εφαρμογής μας και το δεύτερο είναι το https://localhost:5001/api/products/brands που θα μας εμφανίσει τις μάρκες τους. Στις εικόνες 55 και 56 παρουσιάζονται τα αποτελέσματα που λαμβάνουμε ως απάντηση στα HTTP αιτήματά μας από την API server.



Εικόνα 55



Εικόνα 56

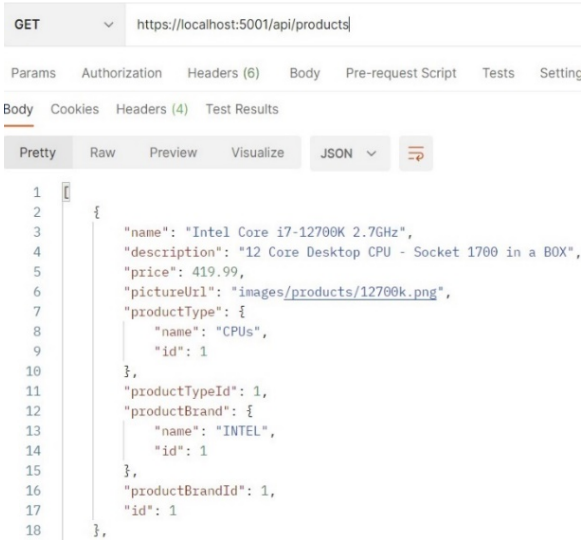
2.7 Eager Lodading

Η τεχνική του *Eager Loading* αφορά τις *navigation properties* που κάθε *web* εφαρμογή διαθέτει. Εφαρμόζοντας την τεχνική του *Eager Loading* υποδεικνύουμε στο *.NET Core* την επιθυμία μας να φορτώσουμε μια *navigation property* ταυτόχρονα με την οντότητα που την περιέχει. Αν παρατηρήσουμε πιο προσεκτικά την εικόνα 51 της προηγούμενης ενότητας, θα διαπιστώσουμε πως τα πεδία *productType* και *productBrand* τα οποία ανήκουν στις properties *ProductType* και *ProductBrand* της κλάσης *product* αντίστοιχα, έχουν τιμή η οποία είναι ίση με *null*. Με τη χρήση του *Eager Loading* θα φέρουμε την πληροφορία που μας χρειάζεται από τις οντότητες *ProductBrands* και *ProductTypes*, ταυτόχρονα με το φόρτωμα της οντότητας *Product*. Αρχικά θα μεταβούμε στην κλάση *ProductRepository* και θα τροποποιήσουμε τις μεθόδους *GetProductByIdAsync* και *GetProductAsync* όπως φαίνεται στην εικόνα 57. Στην ουσία εμπλουτίζουμε το *query* που θα εφαρμοστεί μέσω του *Repository Pattern*, όταν ένα *HTTP* αίτημα φτάσει στις δύο αυτές μεθόδους. Με το να συμπεριλάβουμε τα πεδία *ProductType* και *ProductBrand,* διασφαλίζουμε την περίπτωση οι τιμές που θα λάβουμε ως απάντηση για τα αντίστοιχα πεδία του πίνακα *products* δεν θα είναι null.

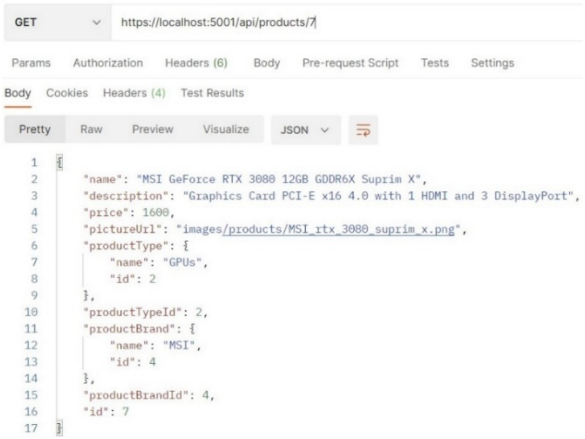
```
public async Task<IReadOnlyList<Product>> GetProductsAsync()
{
      return await _context.Products
            .Include(p=>p.ProductType)
            .Include(p=>p.ProductBrand)
            .ToListAsync();
}

1 reference
public async Task<Product> GetProductByIdAsync(int id)
{
      return await _context.Products
            .Include(p=>p.ProductType)
            .Include(p=>p.ProductBrand)
            .FirstOrDefaultAsync(p=>p.Id == id);
}
```

Εικόνα 57

Στην εφαρμογή *Postman*, με το να αιτηθούμε το URL: https://localhost:5001/api/products και το URL: https://localhost:5001/api/products/7 θα διαπιστώσουμε πως πλέον δεν λαμβάνουμε την τιμή null σε κανένα από τα προηγούμενα πεδία, όπως δείχνουν και οι εικόνες 58 και 59 αντίστοιχα.



Εικόνα 58



Εικόνα 59

Κεφάλαιο 3. Generic Repository Pattern

Σε αυτό το κεφάλαιο θα προσπαθήσουμε να βελτιώσουμε τη δομή της εφαρμογής μας, καθιστώντας την ικανή στο να μπορεί να δεχθεί μελλοντικές προσθήκες κι αναβαθμίσεις, χωρίς να επηρεάζονται όλες οι οντότητες και οι κλάσεις που ήδη έχουμε δημιουργήσει. Στο προηγούμενο κεφάλαιο χρησιμοποιήσαμε το μοτίβο ανάπτυξης *Repository Pattern* για τον ίδιο λόγο, το οποίο αν και λειτουργικό δεν μπορεί να εφαρμοστεί σε όλες τις οντότητες ανεξαιρέτως και με τον ίδιο αποτελεσματικό τρόπο. Με πιο απλά λόγια, εάν υποθέσουμε πως εκτός της οντότητας *Product,* είχαμε κι άλλες εκατό οντότητες για διαφορετικούς λόγους ύπαρξης η κάθε μία, θα έπρεπε να κατασκευάσουμε εκατό *interfaces* κλάσεις κι εκατό *repository* κλάσεις για την κάθε μία. Αυτό θα είχε ως αποτέλεσμα μία εφαρμογή με χιλιάδες γραμμές διπλότυπου κώδικα, μη αποδοτική και περίπλοκη ως προς τη διαδικασία της αποσφαλμάτωσης. Θέλοντας να αποφύγουμε τα παραπάνω, θα δοκιμάσουμε να βελτιώσουμε το μοτίβο ανάπτυξης της εφαρμογής κάνοντας χρήση του *Generic Repository Pattern*. Έτσι, θα επιτύχουμε την καλύτερη δομή κώδικα, την καθολική εφαρμογή του σε όλες τις οντότητες (*entities)*/κλάσεις της εφαρμογής μας και το πιο αποδοτικό φιλτράρισμα κατά την άντληση στοιχείων από τη βάση δεδομένων *pcparts.db*.

3.1 Δημιουργία του Generic Repository Interface

Όπως ακριβώς και στην περίπτωση του *Repository Pattern*, έτσι και τώρα θα δημιουργήσουμε μία νέα κλάση τύπου *Interface* εντός του καταλόγου *Interfaces* του *Core project*, την οποία και θα ονομάσουμε *IGenericRepository*. Καταρχάς προσδιορίζουμε τον τύπο δεδομένων που χρησιμοποιεί το συγκεκριμένο *interface*, τα οποία είναι τύπου *BaseEntity* και δηλώνονται ως <*T*>. Εν συνεχεία, ορίζουμε όλα τα απαραίτητα *Tasks* και τις μεθόδους, ώστε να μπορούμε να αντλήσουμε δεδομένα από τη βάση της εφαρμογής. Κατά σειρά ορισμού, θα μπορούμε να αιτούμαστε: Τα προϊόντα με βάση το *Id* τους, όλα τα προϊόντα σε μορφή λίστας, θα έχουμε τη δυνατότητα να ενημερώνουμε, να προσθέτουμε αλλά και να διαγράφουμε ένα προϊόν και τέλος, να μπορούμε να αποθηκεύουμε τυχόν αλλαγές στη βάση δεδομένων και να ελέγχουμε την ύπαρξη ενός προϊόντος με βάση το *Id* του. Στην εικόνα 60 παρουσιάζεται η κλάση/*Interface IGeneriRepository*.

IGenericRepository.cs ×

Core > Interfaces > IGenericRepository.cs > ...

```
1  using Core.Entities;
2
3  namespace Core.Interfaces;
4
5  //type T is based on BaseEntity properties types
   0 references
6  public interface IGenericRepository<T> where T : BaseEntity
7  {
       0 references
8      Task<T?> GetByIdAsync(int id); //? optional
       0 references
9      Task<IReadOnlyList<T>> ListAllAsync(); //lists all products
       0 references
10     void Add(T entity); //CRUD operations
       0 references
11     void Update(T entity);
       0 references
12     void Remove(T entity);
       0 references
13     Task<bool> SaveAllAsync(); //check if we made changes to db
       0 references
14     bool Exists(int id); //checks if product id exists
15
16 }
```

Εικόνα 60

Ακολουθώντας τη λογική του *Respository Pattern*, σειρά έχει ο ορισμός της υπηρεσίας τύπου *IGenericRepository (Service)* εντός της κλάσης *Program*, η οποία βρίσκεται στο *API project*. Με τον ορισμό αυτής της υπηρεσίας, καθιστούμε δυνατή τη χρήση όλων των αλληλένδετων κλάσεων που χρησιμοποιούν το *Generic Repository Pattern* (εικόνα 61).

```
22  //Generic Repository Pattern Service
23  builder.Services.AddScoped(typeof(IGenericRepository<>), typeof(GenericRepository<>));
24
```

Εικόνα 61

3.2 Η Κλάση Generic Repository

Ακολούθως, έχουμε τη δημιουργία μιας *Implementation* κλάσης εντός του *Infrastructure project* και του καταλόγου *Data*. Η κλάση αυτή ονομάζεται *GenericRepository*, δέχεται δεδομένα τύπου <*T*> (όπου T = BaseEntity type = Product), χρησιμοποιεί την κλάση *StoreContext* καθώς και το *Interface IGenericRepository*, το οποίο δημιουργήσαμε στο προηγούμενο βήμα. Τα δεδομένα τύπου <*T*> ουσιαστικά αντικαθιστούν το όρισμα *product* που είχαμε δώσει στις αντίστοιχες κλάσεις του *Repository Pattern* κι έτσι γενικεύουμε τη λειτουργικότητα της εφαρμογής μας για οποιαδήποτε οντότητα προσθέσουμε στο μέλλον. Εντός της κλάσης *IGenericRepository* ορίζουμε τη λειτουργικότητα των μεθόδων του *IGenericRepository Interface*, ορίζοντας τον τύπο δεδομένων *[Set<T>.()]* που περνούν στην κλάση *StoreContext* και των μεταβλητών *id* και *entity*, που χρησιμοποιούνται στην κλάση *IGenericRepository*. Ο κώδικας της κλάσης *GenericProductRepository* παρουσιάζεται στις εικόνες 62 και 63.

GenericRepository.cs ●

Infrastructure > Data > GenericRepository.cs > ...

```
1   using Core.Entities;
2   using Core.Interfaces;
3   using Microsoft.EntityFrameworkCore;
4
5   namespace Infrastructure.Data;
6
7   //use of IgenericRepository and T only usable with BaseEntity
        1 reference
8   public class GenericRepository<T>(StoreContext context) : IGenericRepository<T> where T : BaseEntity
9   {
            1 reference
10      public void Add(T entity) //product addition
11      {
12          context.Set<T>().Add(entity);
13      }
14
            1 reference
15      public bool Exists(int id)//checks product existance
16      {
17          return context.Set<T>().Any(x => x.Id == id);//Checks product id via BaseEntity property
18      }
19
            1 reference
20      public async Task<T?> GetByIdAsync(int id) //finds and returns all products of specific id
21      {
22          return await context.Set<T>().FindAsync(id);
23      }
24
```

Εικόνα 62

```
        1 reference
25      public async Task<IReadOnlyList<T>> ListAllAsync() //finds and returns all products
26      {
27          return await context.Set<T>().ToListAsync();
28      }
29
        1 reference
30      public void Remove(T entity) //removes a product from db
31      {
32          context.Set<T>().Remove(entity);
33      }
34
        1 reference
35      public async Task<bool> SaveAllAsync() //checks if DB modified
36      {
37          return await context.SaveChangesAsync() > 0;
38      }
39
40      //saves cahnges applied to DB
        1 reference
41      public void Update(T entity)
42      {
43          context.Set<T>().Attach(entity); //set of the entity - tells Entity Framework to track it
44          context.Entry(entity).State = EntityState.Modified; //saves changes
45      }
46  }
```

Εικόνα 63

3.3 Χρήση του Generic Repository Pattern Εντός του ProductsController

Πριν μεταβούμε στο περιβάλλον της εφαρμογής Postman προκειμένου να ελέγξουμε τη μέχρι τώρα λειτουργικότητα της εφαρμογής μας, θα πρέπει να πραγματοποιήσουμε ορισμένες αλλαγές στην κλάση *ProductsController*, εντός του *API project*. Αρχικά θα πρέπει να αντικαταστήσουμε το *IProductRepositry Interface* με το αντίστοιχο *IGenericRepository* (εικόνα 64).

```
[ApiController] //Attribute that improves developer experience during the bulding c
[Route("api/[controller]")] //let server know where to send the incoming http reque
0 references
public class ProductsController(IGenericRepository<Product> repo) : ControllerBase
{
```

Εικόνα 64

Ακολούθως, δημιουργούμε τη μέθοδο *GetProducts*, με τρία προαιρετικά ορίσματα (*brand, type, sort*) τα οποία και θα αναλύσουμε αργότερα ως προς τη λειτουργικότητα και τη χρηστικότητά τους. Προς το παρόν η μέθοδος αυτή θα μας επιστρέφει ασύγχρονα όλα τα προϊόντα από τη βάση δεδομένων της εφαρμογής μας, κάνοντας χρήση της μεθόδου *ListAllAsync* την οποία ορίσαμε εντός της κλάσης *GenericRepository*. Στην εικόνα 65 παρουσιάζεται ο κώδικας της μεθόδου *GetProducts*.

```
[HttpGet] //return asynchronously a task (action result) in http format as a list (a list of products in our ex
//url: api/products
//public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
0 references
public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts(string? brand, string? type, string? sort)
{
    //return await context.Products.ToListAsync();   //asynchronous query in order to return a list of produc
    //after IProduct RepositoryPattern use
    return Ok(await repo.ListAllAsync());
}
```

Εικόνα 65A

Με την ίδια λογική, δημιουργούμε τη μέθοδο *GetProduct*, η οποία δέχεται ως όρισμα έναν ακέραιο αριθμό με όνομα *id,* το οποίο και χρησιμοποιείται ως όρισμα στη μέθοδο *GetByIdAsync*, προκειμένου να ελεγχθεί η ύπαρξή του εντός της βάσης δεδομένων και να επιστραφεί το ανάλογο προϊόν ασύγχρονα, εφόσον υπάρχει. Στην εικόνα 65Β παρουσιάζεται η εν λόγω μέθοδος.

```
[HttpGet("{id:int}")] //url api/products/id of product
0 references
public async Task<ActionResult<Product>> GetProduct(int id)
{
      //var product = await context.Products.FindAsync(id);
      var product = await repo.GetByIdAsync(id);
      //in case id is null/not found in db
      if (product == null) return NotFound();

      return product;
}
```

Εικόνα 65Β

Θέλοντας να διανθίσουμε τη λειτουργικότητα της εφαρμογής μας, θα προχωρήσουμε στη δημιουργία τριών νέων μεθόδων που υπηρετούν τη λογική *CR.U.D.* Με το ακρωνύμιο *CR.U.D.* (*Create, Update, Delete*) αναφερόμαστε στην υποδομή μιας *Web* εφαρμογής, στο να μπορεί να υποστηρίξει τη δημιουργία, την ενημέρωση και τη διαγραφή οντοτήτων από τη βάση δεδομένων της. Έχοντας υπόψιν τη συγκεκριμένη λογική η επόμενη μέθοδος εντός της κλάσης *ProductsController*, είναι η *CreateProduct*. Δέχεται ως όρισμα μια μεταβλητή τύπου *Product* και μέσω της μεθόδου *Add* δημιουργεί ένα νέο προϊόν εντός της *pcparts.db*. Σε περίπτωση που αυτό καταστεί αδύνατο, επιστρέφεται το ανάλογο μήνυμα προς τον χρήστη. Σε αντίθετη περίπτωση, μέσω της μεθόδου *CreatedAtAction*, η οποία χρησιμοποιεί την *GetProduct* ως παράμετρο, επιστρέφεται το νέο προϊόν στον χρήστη (εικόνα 66). Παρομοίως, προκύπτει και η δημιουργία των μεθόδων *UpdateProduct* και *DeleteProduct*. Εκμεταλλευόμενες τις μεθόδους *ProductExists*, *Update*, *Remove* και *SaveAllAsync* οι δύο αυτές μέθοδοι μας δίνουν τη δυνατότητα να ενημερώνουμε ή να διαγράφουμε από τη βάση δεδομένων της εφαρμογής μας ένα προϊόν. Και στις δύο περιπτώσεις προηγείται έλεγχος για την ύπαρξη ενός προϊόντος με το δεδομένο *id* από τον χρήστη μέσω της μεθόδου *ProductExists*. Επίσης, στην περίπτωση που δεν πραγματοποιηθεί η ενημέρωση της βάσης δεδομένων (Η μέθοδος *SaveAllAsync* μας επιστρέφει την τιμή *false*) επιστρέφεται ανάλογο μήνυμα προς τον χρήστη. Οι δύο αυτές μέθοδοι παρουσιάζονται στις εικόνες 67 και 68 αντίστοιχα.

```
[HttpPost] //http post endpoint
//query that adds/creates a product into db
0 references
public async Task<ActionResult<Product>> CreateProduct(Product product)
{
      //context.Products.Add(product);
      repo.Add(product);
      if(await repo.SaveAllAsync())
      {     //returns the created product using the GetProduct method
            return CreatedAtAction("GetProduct", new { id = product.Id }, product);
      }
      return BadRequest("Product update failure!");
}
```

Εικόνα 66

```
//update db with new product addition
[HttpPut("{id:int}")] //specify the attribute type
0 references
public async Task<ActionResult> UpdateProduct(int id, Product product)
{
      if (product.Id != id || !ProductExists(id))
            return BadRequest("Cannot update this product!");

      //context.Entry(product).State = EntityState.Modified;
      repo.Update(product);

      if(await repo.SaveAllAsync())
      {
            return NoContent();
      }

      return BadRequest("Problem updating he product!!!");
}
```

Εικόνα 67

```
//update db with removal of a product
[HttpDelete("{id:int}")]
0 references
public async Task<ActionResult> DeleteProduct(int id)
{
      //var product = await context.Products.FindAsync(id);
      var product = await repo.GetByIdAsync(id);
      if (ProductExists(id) == false || product == null) return NotFound();

      //context.Products.Remove(product);
      repo.Remove(product);

      //await context.SaveChangesAsync();
      if (await repo.SaveAllAsync())
      {
            return NoContent();
      }

      return BadRequest("Problem deleting the product!!!");
}
```

Εικόνα 68

Στην εικόνα 69 βλέπουμε και τον ορισμό της μεθόδου *ProductExists* η οποία χρησιμοποιεί τη μέθοδο *Exists*, την οποία ορίσαμε εντός της κλάσης *GenericRepository*.

```
//check function iinside db for same id while updating or deleting a product
2 references
private bool ProductExists(int id)
{
      //return context.Products.Any(x => x.Id == id);
      return repo.Exists(id);
}
```

Εικόνα 69

Τελική προσθήκη εντός της κλάσης *ProducstController* αποτελούν οι μέθοδοι *GetBrands* και *GetTypes*. Θέλοντας να δώσουμε τη δυνατότητα στον χρήστη να αντλεί τις διάφορες κατηγορίες των προϊόντων από τη βάση δεδομένων (CPUs, GPUs, RAMs, SSDs, Mobos) ή τη μάρκα τους, προσθέτουμε τις δύο αυτές μεθόδους (εικόνα 70). Προς το παρόν δεν είναι λειτουργικές λόγω των περιορισμών της δομής του *Generic Repository Pattern*, ακριβώς όπως συμβαίνει και στην περίπτωση της μεθόδου *GetProducts*, αναφορικά με τα ορίσματα *type, brand, sort*. Την αδυναμία αυτή θα την επιλύσει ένα νέο μοτίβο ανάπτυξης, το *Specification Pattern*, τη λογική και τις δυνατότητες του οποίου θα εξετάσουμε στις επόμενες ενότητες.

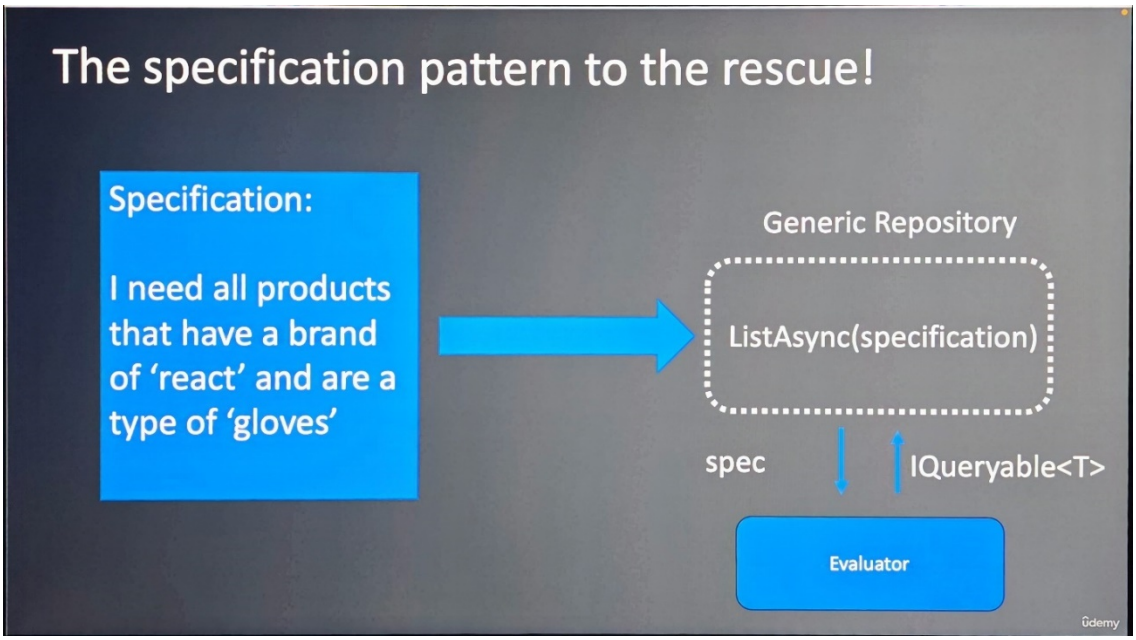
```
//query products brands list
[HttpGet("brands")]
0 references
public async Task<ActionResult<IReadOnlyList<string>>> GetBrands()
{
      return Ok();
}

//query products types list
[HttpGet("types")]
0 references
public async Task<ActionResult<IReadOnlyList<string>>> GetTypes()
{
      return Ok();
}
```

Εικόνα 70

3.4 Specification Pattern

Ακολουθώντας ως μοτίβο ανάπτυξης της εφαρμογής μας το *Generic Repository Pattern*, παρατηρούμε πως ορισμένες μέθοδοι εντός της κλάσης *ProductsController* παρουσιάζουν προβλήματα ως προς την υλοποίησή τους και κατά συνέπεια η εφαρμογή μας εμφανίζει αδυναμίες ως προς τη λειτουργικότητά της. Στην προηγούμενη ενότητα, αναφερθήκαμε στις μεθόδους *GetBrands* και *GetTypes* ως παράδειγμα μιας τέτοιας αδυναμίας, οι οποίες έχουν ως ρόλο την επιστροφή των προϊόντων στον *client* βάσει συγκεκριμένων κριτηρίων, τα οποία και παραθέτει ως όρισμα στο URL που αναζητά. Όταν λοιπόν ερχόμαστε αντιμέτωποι με την αδυναμία μιας μεθόδου η οποία, παραδείγματος χάριν, δέχεται ως όρισμα μία *generic* έκφραση τύπου <*T*> βάσει συγκεκριμένων κριτηρίων (*Brands, Types* κτλ.) και μας επιστρέφει μία τιμή *boolean* σε μια αφηρημένη μορφή *IQueryable<T>*, καταλαβαίνουμε πως πρέπει να γίνουμε πολύ πιο σαφείς ως προς τα *queries* που εκτελούμε στη βάση δεδομένων μας. Το φαινόμενο των «αφηρημένων» queries στην περίπτωση του *Generic Repository Pattern*, αναφέρεται συχνά κι ως «*Leaky Abstraction*». Απόρροια των όσων ανέφερα είναι η αναζήτηση μιας συγκεκριμένης φόρμας/μοτίβου υλοποίησης που βοηθά το *Generic Repository Pattern* στο να μπορεί να διαχειρίζεται πιο περίπλοκα αιτήματα (πχ *queries* σχετικά με την άντληση των *brands* των προϊόντων της pcparts.db) από τον εκάστοτε *client*, προς τις οντότητες της οποιασδήποτε *Web* εφαρμογής και πίσω. Το *Specification Pattern* αποτελεί την ενδεδειγμένη λύση στο πρόβλημά μας κι ο ρόλος που θα διαδραματίσει στην εφαρμογή μας είναι απλός. Μέσω των κλάσεων που θα υλοποιήσουμε και θα ακολουθούν το συγκεκριμένο μοτίβο, θα επιτύχουμε κάθε αίτημα (query) να περιγράφεται μέσω ενός αντικειμένου (*object*), το οποίο και θα χρησιμοποιείται από το *Generic Repository Pattern*.

Στη συνέχεια μέσω μιας κλάσης (*evaluator class*) το αντικείμενο (*object*) θα αξιολογείται βάσει του *Entity Framework Expressions Tree* και θα αποστέλλεται εκ νέου στην εκάστοτε μέθοδο του *Generic Repository Pattern*, σε *IQueryable<T>* μορφή με σκοπό να αντλήσουμε την πληροφορία που επιθυμούμε από τη βάση δεδομένων της εφαρμογής μας. Στην εικόνα 71 αναπαρίσταται η φιλοσοφία του *Specification Pattern*.



Εικόνα 71

3.5 Specification Κλάσεις

Εντός του καταλόγου *Interfaces*, του *Core project* της εφαρμογής μας, θα προσθέσουμε μία ακόμη κλάση τύπου *Interface* με τίτλο *ISpecification*. Αρχικά θα ορίσουμε ως παράμετρο του *interface* τον τύπο <*T*> κι έπειτα θα προσπαθήσουμε να μεταφράσουμε σε μορφή *query* μία *generic expression* τύπου *Func*, η οποία δέχεται ως όρισμα ένα αντικείμενο τύπου *Product* κι επιστρέφει μία *Boolean* τιμή σε μορφή *IQueryable<T>*. Την έκφραση (*generic expression*) αυτή την ονομάζουμε *Criteria* (εικόνα 72) και θα χρησιμοποιηθεί (*Implemented*) από την *evaluator* κλάση, μέσω των μεθόδων της *implementation* κλάσης *BaseSpecification*.

ISpecification.cs

Core > Interfaces > ISpecification.cs > ...

```
1   using System.Linq.Expressions;
2
3   namespace Core.Interfaces;
4
    0 references
5   public interface ISpecification<T>
6   {
        0 references
7       Expression<Func<T, bool>> Criteria { get; }
8   }
```

Εικόνα 72

Εντός του *Core Project* δημιουργούμε έναν νέο κατάλογο με τίτλο *Specifications*. Εντός του καταλόγου *Specifications* ορίζουμε μία νέα κλάση με τίτλο *BaseSpecification*. Η κλάση αυτή δέχεται ως παράμετρο τον τύπο *<T>* και κάνει *implement* την κλάση *ISpecification*, την οποία κι ορίσαμε στο προηγούμενο βήμα. Δημιουργώντας έναν *primary constructor* της κλάσης *BaseSpecification* έχουμε τη δυνατότητα να της περάσουμε ως όρισμα μία έκφραση *func* τύπου *<T>* (πχ *query.Where(x => x.Brand == brand*), η οποία επιστρέφει μια *boolean* τιμή κι αυτό το πραγματοποιούμε μέσω της *generic expression Criteria* (εικόνα 73).

BaseSpecification.cs

Core > Specifications > BaseSpecification.cs > ...

```
1  using System.Linq.Expressions;
2  using Core.Interfaces;
3
4  namespace Core.Specifications;
5
6  //primary constructor implementing the is[ecification expression criteria
   0 references
7  public class BaseSpecification<T>(Expression<Func<T, bool>> criteria) : ISpecification<T>
8  {
       1 reference
9      public Expression<Func<T, bool>> Criteria => criteria;
10 }
```

Εικόνα 73

Σε αυτό το σημείο θα πρέπει να λάβουμε υπόψιν τη διαδικασία του *evaluation* της έκφρασης που έχουμε δημιουργήσει. Εντός του *Infrastructure project* και του καταλόγου *Data*, θα ορίσουμε μία νέα κλάση με τίτλο *SpecificationEvaluator*. Η κλάση αυτή δέχεται έναν *type parameter <T>* της μορφής *BaseEntity*. Στη συνέχεια ορίζουμε μία μέθοδο με τίτλο *GetQuery* η οποία δέχεται ως όρισμα μία μεταβλητή τύπου *IQueryable<T>* (*query*) και μία τύπου *ISpecification<T>* (*spec*) κι επιστρέφει μία μεταβλητή τύπου *IQueryable<T>*. Με τον τρόπο αυτό η μέθοδος *GetQuery* ελέγχει την περίπτωση που το δεδομένο *query (spec.Criteria)* πληροί τα κριτήρια εντός του *Entity Framework Expressions Tree* κι επιστρέφεται σε μορφή *IQueryable<T>*. Ο κώδικας της κλάσης *SpecificationEvaluator* παρουσιάζεται στην εικόνα 74.

SpecificationEvaluator.cs

Infrastructure > Data > SpecificationEvaluator.cs > ...

```
1  using Core.Entities;
2  using Core.Interfaces;
3
4  namespace Infrastructure.Data;
5
   0 references
6  public class SpecificationEvaluator<T> where T : BaseEntity
7  {
8      //This method evaluates the given criteria to the one of the ISpecification's expression
       0 references
9      public static IQueryable<T> GetQuery(IQueryable<T> query, ISpecification<T> spec)
10     {
11         if (spec.Criteria != null)
12         {
13             query = query.Where(spec.Criteria); //FOR EXAMPLE checks if x => x.Brand == brand
14         }
15
16         return query;
17     }
18 }
```

Εικόνα 74

3.6 Ενημέρωση του Generic Repository Pattern

Αφού έχουμε δημιουργήσει τη βασική υποδομή για τη χρήση του *Specification Pattern* μέσω των τριών κλάσεων *ISpecification, BaseSpecification* και *SpecificationEvaluator*, έχει ήρθε η ώρα να προσαρμόσουμε τις κλάσεις του *Generic Repository Pattern*, ώστε να μπορούν να εκμεταλλευτούν τη λειτουργικότητά του. Αρχικά θα μεταβούμε στην κλάση (*Interface*) *IGenericRepository* και θα δημιουργήσουμε δύο νέες μεθόδους, την *GetEntityWithSpec* και την *ListAsync*. Οι δύο αυτές μέθοδοι μπορούν να επιστρέψουν ένα ή μία λίστα προϊόντων αντίστοιχα, δεχόμενες την παράμετρο *spec* τύπου *ISpecification<T>.* Με άλλα λόγια, μπορούν να δεχτούν ως όρισμα μία έκφραση τύπου *query* μέσω του *ISpecification interface* (εικόνα 75).

```
//Specification Pattern methods
0 references
Task<T?> GetEntityWithSpec(ISpecification<T> spec);
0 references
Task<IReadOnlyList<T>> ListAsync(ISpecification<T> spec);
```

Εικόνα 75

Εν συνεχεία θα μεταβούμε στην κλάση *GenericRepository*, ώστε να κάνουμε *implement* τις νέες μεθόδους της κλάσης (*Interface*) *IGenericRepository*. Επίσης θα ορίσουμε μία *helper* μέθοδο με τίτλο *ApplySpecification*, η οποία δέχεται ως όρισμα μία μεταβλητή τύπου *ISpecification<T> (spec)*. Ο ρόλος αυτής της μεθόδου είναι να αποστέλλει το *ISpecification<T>* όρισμά της στη μέθοδο *GetQuery* (την οποία ορίσαμε εντός της κλάσης *SpecificationEvaluator*) και να το επιστρέφει ως *evaluated query* (εικόνα 76).

```
//returns an evaluated query via specification pattern classes
0 references
private IQueryable<T> ApplySpecification(ISpecification<T> spec)
{
    return SpecificationEvaluator<T>.GetQuery(context.Set<T>().AsQueryable(), spec);
}
```

Εικόνα 76

Εφόσον κάναμε *implement* τις μεθόδους *GetEntityWithSpec* , *ListAsync* και παράλληλα ορίσαμε την *helper* μέθοδο *ApplySpecification*, έχουμε τη δυνατότητα να επιστρέφουμε μία λίστα προϊόντων ή και ένα προϊόν δεδομένου ενός *specification* κριτηρίου, όπου στην περίπτωσή μας είναι τύπου *where* (πχ *query.Where x => x.Brand == brand*). Στην εικόνα 77 παρατηρούμε την *GetEntityWithSpec* μέθοδο να μας επιστρέφει ένα (ή και κανένα εφόσον δεν πληρούνται τα *Specification criteria*) προϊόν, μέσω ενός *evaluated query*, εκμεταλλευόμενη την *helper* μέθοδο *ApplySpecification*. Με την ίδια λογική και η μέθοδος *ListAsync* μας επιστρέφει μία λίστα προϊόντων φιλτραρισμένα από ένα *specification* κριτήριο.

```
//specification pattern methods implementation -- start
1 reference
public async Task<T?> GetEntityWithSpec(ISpecification<T> spec)
{
    return await ApplySpecification(spec).FirstOrDefaultAsync();
}
1 reference
public async Task<IReadOnlyList<T>> ListAsync(ISpecification<T> spec)
{
    return await ApplySpecification(spec).ToListAsync();
}
```

Εικόνα 77

3.7 Εφαρμογή του Specification Pattern

Στην ενότητα 3.3 αναφερθήκαμε στη μέθοδο *GetProducts* και τα προαιρετικά ορίσματα *brand* και *type*. Η αδυναμία του *Generic Repository Pattern* στον να ερμηνεύει και να αποδίδει πολύπλοκα *queries* καθιστά την επιστροφή μίας λίστας προϊόντων ή ενός προϊόντος (πχ μάρκας Samsung και τύπου SSD) από τη βάση δεδομένων, αδύνατη. Στις τελευταίες ενότητες της διπλωματικής εργασίας καλύπτουμε το μοτίβο ανάπτυξης *Specification Pattern* με στόχο την επίλυση του συγκεκριμένου προβλήματος. Πριν μεταβούμε στο *API Project* και την κλάση *ProductsController*, θα πρέπει εντός του καταλόγου *Specifications* του *Core Project*, να δημιουργήσουμε μία νέα κλάση με τίτλο *ProductSpecification*. Η κλάση αυτή κληρονομεί τις ιδιότητες της κλάσης *BaseSpecification* η οποία μας δίνει πρόσβαση στις *properties* της οντότητας Product. Εντός του πεδίου ορισμού του *constructor* της, θέτουμε τα ορίσματα *brand* και *type* (τα οποία δίνονται προαιρετικά από τον *client*) κι εφόσον υπάρχουν, διαμορφώνεται μία έκφραση που θα χρησιμοποιηθεί για τη δημιουργία ενός *query* μέσω της μεθόδου *GetProducts*, εντός του *ProductsController*. Ο κώδικας της κλάσης *ProductSpecification* παρουσιάζεται στην εικόνα 78.

ProductSpecification.cs

Core > Specifications > ProductSpecification.cs > ...

```
1  using Core.Entities;
2
3  namespace Core.Specifications;
4
   1 reference
5  public class ProductSpecification : BaseSpecification<Product>
6  {
7      //creation of an expression using (optionally) the brand and type from client
8      //in oredr to parse it to products controller class methods and add it to specifi queries
       0 references
9      public ProductSpecification(string? brand, string? type) : base(x =>
10         (string.IsNullOrWhiteSpace(brand) || x.Brand == brand) &&
11         (string.IsNullOrWhiteSpace(type) || x.Type == type)
12     )
13     {
14
15     }
16 }
```

Εικόνα 78

Πλέον μπορούμε να μεταβούμε στον *ProductController* του *API Project* και να ενημερώσουμε τη μέθοδο *GetProducts* ως εξής. Ορίζουμε μία μεταβλητή με όνομα *spec* στην οποία αποθηκεύουμε το αποτέλεσμα της μεθόδου *ProductSpecification*. Στη συνέχεια χρησιμοποιούμε τη μέθοδο *ListAsync* της *IGenericRepository* κλάσης, που πλέον εκμεταλλεύεται τη λειτουργικότητα του *Specification Pattern*. Εφόσον αξιολογηθεί (*Evaluated*) το *query* και εφαρμοστεί στη βάση δεδομένων, αποθηκεύεται το αποτέλεσμα του (σε μορφή *IQueryable<T>*) στη μεταβλητή *products*. Τέλος, στον *client* εμφανίζεται η επιθυμητή πληροφορία σε μορφή λίστας. Αναλυτικά, στην εικόνα 79 έχουμε τον κώδικα της μεθόδου *GetProducts*.

```
//http endpoints
[HttpGet] //return asynchronously a task (action result) in http format as a list (a list of products in our ex
//url: api/products
//public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
0 references
public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts(string? brand, string? type, string? sort)
{
    //return await context.Products.ToListAsync();   //asynchronous query in order to return a list of produc
    //after IProduct RepositoryPattern use

    var spec = new ProductSpecification(brand, type); //using specifications to format the query
    var products = await repo.ListAsync(spec); //applying the query to infrastructure repository classes

    return Ok(products); //returns the diltered list of products to the client
}
```

Εικόνα 79

Ένα ακόμη σημείο στο οποίο θα πρέπει να σταθούμε, είναι η προσθήκη της δυνατότητας του *sorting*. Ο *client* θα έχει τη δυνατότητα να αντλεί πληροφορία από τη βάση δεδομένων θέτοντας το κριτήριο της ταξινόμησης των προϊόντων σύμφωνα με την αξία ή την ονομασία τους. Εξάλλου δεν πρέπει να ξεχνάμε πως εκτός από τα ορίσματα *brand* και *type*, η μέθοδος *GetProducts* διαθέτει ένα ακόμη όρισμα με τίτλο *sort*. Για τον λόγο αυτό θα μεταβούμε εκ νέου στην *interface* κλάση *ISpecification*, όπου δημιουργούμε δύο νέες *expressions* στην ίδια λογική της *expression Criteria*. Σε αυτήν την περίπτωση δεν γνωρίζουμε τον τύπο των δεδομένων που επιστρέφεται, γι' αυτό και χρησιμοποιούμε τον τύπο *object*. Με την έκφραση *OrderBy* θα καλύψουμε την περίπτωση της ταξινόμησης κατά αύξουσα σειρά εμφάνισης και με την έκφραση *OrderByDescending* την περίπτωση της εμφάνισης κατά φθίνουσα σειρά. Την ενημερωμένη κλάση *ISpecification* τη βλέπουμε στην εικόνα 80.

ISpecification.cs

Core > Interfaces > ISpecification.cs > ...

```
1  using System.Linq.Expressions;
2
3  namespace Core.Interfaces;
4
   7 references
5  public interface ISpecification<T>
6  {
7      //getting the brand or type info (if exists) in order to evaluate the query
       3 references
8      Expression<Func<T, bool>>? Criteria { get; }
       0 references
9      Expression<Func<T, object>>? OrderBy { get; } //getting info for sorting
10     //using "object" cause we dont know what's the ruturn type (string,decimal...)
       0 references
11     Expression<Func<T, object>>? OrderByDescending { get; }
12 }
```

Εικόνα 80

Ακολούθως, ενημερώνουμε την κλάση *BaseSpecification* ώστε να υποστηρίζει τη λειτουργικότητα του *sorting* βασιζόμενη στις δύο νέες *expressions* που ορίσαμε εντός του *interface ISpecification* (*Implementation process* – εικόνα 81).

```
15     //sorting info - expressions support implementation
       4 references
16     public Expression<Func<T, object>>? OrderBy { get; private set; }
17
       4 references
18     public Expression<Func<T, object>>? OrderByDescending { get; private set; }
19
       0 references
20     protected void AddOrderBy(Expression<Func<T, object>> orderByExpression)
21     {
22         OrderBy = orderByExpression;
23     }
       0 references
24     protected void AddOrderByDescending(Expression<Func<T, object>> orderByDescExpression)
25     {
26         OrderByDescending = orderByDescExpression;
27     }
28 }
```

Εικόνα 81

Σειρά έχει η διαδικασία του *evaluation*, όπως ακριβώς και στην περίπτωση του «*where query*». Θα μεταβούμε στην κλάση *SpecificationEvaluator* όπου οι νέες εκφράσεις θα αξιολογηθούν και θα προστεθούν στο *sorting query*. Ουσιαστικά γίνεται έλεγχος για το αν η μεταβλητή *spec* εμπεριέχει την έκφραση

OrderBy ή την *OrderByDescending* κι αν αυτό συμβαίνει προστίθεται στο τελικό *query* προς τη βάση δεδομένων (εικόνα 82).

SpecificationEvaluator.cs

Infrastructure > Data > SpecificationEvaluator.cs > ...

```
using Core.Entities;
using Core.Interfaces;

namespace Infrastructure.Data;

1 reference
public class SpecificationEvaluator<T> where T : BaseEntity
{
    //This method evaluates the given criteria to the one of the ISpecification's expression
    1 reference
    public static IQueryable<T> GetQuery(IQueryable<T> query, ISpecification<T> spec)
    {
        if (spec.Criteria != null)
        {
            query = query.Where(spec.Criteria); //FOR EXAMPLE checks if x => x.Brand == brand
        }
        //FOR EXAMPLE  "priceAsc" => query.OrderBy(x => x.Price)
        if(spec.OrderBy != null)
        {
            query = query.OrderBy(spec.OrderBy);

        }
        //FOR EXAMPLE "priceDesc" => query.OrderByDescending(x => x.Price)
        if (spec.OrderByDescending != null)
        {
            query = query.OrderByDescending(spec.OrderByDescending);
        }
        return query;
    }
}
```

Εικόνα 82

Τελευταία στάση πριν τον *ProductsController*, είναι η κλάση *ProductSpecification*. Εκεί θα προσθέσουμε ένα *Switch Statement* το οποίο καλύπτει όλες τις περιπτώσεις της *sorting* διαδικασίας, η οποία επιθυμούμε να εφαρμόζεται στις *product properties* (*Price*, *Name*), μέσω των μεθόδων *AddOrderBy* και *AddOrderByDescending*, τις οποίες ορίσαμε εντός της κλάσης *BaseSpecification* (εικόνα 83).

```
    {
        //sorting case
        switch (sort)
        {
            case "priceAsc":
                AddOrderBy(x => x.Price);
                break;
            case "priceDesc":
                AddOrderByDescending(x => x.Price);
                break;
            default:
                AddOrderBy(x => x.Name);
                break;
        }
    }
}
```

Εικόνα 83

Τέλος, εντός του *ProductsController* και της κλάσης *GetProducts*, προσθέτουμε το όρισμα *sort* στη μέθοδο *ProductSpecification* (εικόνα 84).

```
24      [HttpGet] //return asynchronously a task (action result) in http format as a list (a list of products in our e
25      //url: api/products
26      //public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
            0 references
27      public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts(string? brand, string? type, string? sort)
28      {
29          //return await context.Products.ToListAsync();   //asynchronous query in order to return a list of produ
30          //after IProduct RepositoryPattern use
31
32          var spec = new ProductSpecification(brand, type, sort); //using specifications to format the query
33          var products = await repo.ListAsync(spec); //applying the query to infrastructure repository classes
34
35          return Ok(products); //returns the filtered list of products to the client
36      }
```

Εικόνα 84

Θεωρητικά, ολοκληρώσαμε την υλοποίηση του *Specification Pattern* καθώς και την ενημέρωση του *Generic Repository Pattern* για την υποστήριξή του. Πλέον η εφαρμογή μας μπορεί να δέχεται ένα *HTTP* αίτημα εντός του *API Project* από τον *Client* και να εξυπηρετείτε μέσω του *ProductsController*. Το αίτημα αυτό φιλτράρεται από τη μέθοδο *GetProducts*, για τις περιπτώσεις που εμπεριέχει στοιχεία που αφορούν το *Sorting* ή το *Filtering* (*Brand, Type*) ενός ή περισσότερων προϊόντων. Η πληροφορία αυτή μεταφέρεται σε ένα νέο Instance της κλάσης *ProductSpecification* η οποία κάνει *derive* όλες τις οντότητες και τα πεδία της κλάσης *BaseSpecification*, η οποία με τη σειρά της μας δίνει πρόσβαση στην οντότητα *Product* μέσω των *Properties* και των μεθόδων της. Εντός του *constructor* της κλάσης *ProductSpecification*, χτίζονται οι εκφράσεις τύπου *Where(x => x.Brand == brand)* ή *Where(x => x.Type == type)*, καθώς και η λειτουργικότητα του *Sorting*. Η έκφραση που προκύπτει μεταφέρεται στην κλάση *SpecificationEvaluator*, με χρήση του *Generic Repository Pattern* και της μεθόδου *ListAsync*, εντός της μεθόδου *GetProducts* του *ProductsController*. Με χρήση της μεθόδου *ApplySpecification* εντός της κλάσης *GenericRepository*, έχουμε την κλήση της μεθόδου *GetQuery*, την οποία ορίσαμε εντός της *SpecificationEvaluator* κλάσης. Η έκφραση μεταφράζεται σε μορφή *Specific Expression Tree Query* προς τη βάση δεδομένων και το αποτέλεσμα (*ListAsync*) αποθηκεύεται σε μια μεταβλητή (*products*) εντός της μεθόδου *GetProducts*. Η μεταβλητή (*products*) περιέχει τη λίστα με την πληροφορία που αιτήθηκε ο *client*.

3.8 Projection

Κλείνοντας την ενότητα 3.3 αναφερθήκαμε σε δύο μεθόδους, την *GetBrands* και την *GetTypes*. Θέλοντας να αντλήσουμε όλες τις κατηγορίες προϊόντων ή τις μάρκες τους μέσα από τη βάση δεδομένων *pcparts.db* θα εκμεταλλευτούμε μία ακόμη ιδιότητα του *Microsoft Entity Framework*, η οποία ονομάζεται *Projection*. Χρησιμοποιώντας το *Projection* θα καταφέρουμε να μετατρέψουμε το επιστρεφόμενο αποτέλεσμα από ένα *ListAsync type<T> Query* σε μία μορφή *ListAsync type<TResult>*. Με πιο απλά λόγια θα δοκιμάσουμε να αντλήσουμε μέσα από μία οντότητα τύπου *Product*, μία «υποπληροφορία» τύπου *Brand* ή *Type*. Αρχικά θα πρέπει να ενημερώσουμε το *Specification Pattern*. Ανοίγοντας το *Interface Ispecification* θα δημιουργήσουμε ένα νέο *Instance* το οποίο δέχεται ένα όρισμα τύπου <*T*> και μας επιστρέφει μία πληροφορία τύπου <*TResult*> / *Projection Result*, ενώ ταυτόχρονα υποστηρίζει και τις ήδη υπάρχουσες εκφράσεις τις οποίες ορίσαμε στην προηγούμενη ενότητα (εικόνα 85).

```
13
14  //Projection result (TYPES + BRANDS case)
15  //Takes type<T> item - Returns <TResult> item
16  //supports all expressions from ISpecification interface
    0 references
17  public interface ISpecification<T, TResult> : ISpecification<T>
18  {
        0 references
19      Expression<Func<T, TResult>>? Select { get; }
20  }
```

Εικόνα 85

Ακολούθως, ανοίγουμε την *BaseSpecification* κλάση και την ενημερώνουμε δημιουργώντας ένα νέο *instance* (εικόνα 86), προκειμένου να μας επιστρέφει ένα αποτέλεσμα τύπου <*TResult*> με σκοπό την ενημέρωση της *Evaluator* κλάσης, για τη δημιουργία του κατάλληλου *Specific Expression Tree Query* , ενώ ταυτόχρονα διατηρεί όλη τη λειτουργικότητα του *Specification Pattern* που ήδη έχουμε δημιουργήσει.

```
//projection support (BRANDS,TYPES)
//Derives from ISPECIFICATION projection instance + BaseSpecification instance
//Returns item of TResult type & supports all the expressions and functions above
0 references
public class BaseSpecification<T, TResult>(Expression<Func<T, bool>> criteria)
    : BaseSpecification<T>(criteria), ISpecification<T, TResult>
{
    //implementation of ISpecification interface
    2 references
    public Expression<Func<T, TResult>>? Select { get; private set; }
    //Adds the select into the expression
    0 references
    protected void AddSelect(Expression<Func<T, TResult>> selectExpression)
    {
        Select = selectExpression;
    }
}
```

Εικόνα 86

Στην κλάση *SpecificationEvaluator* θα πρέπει να δημιουργήσουμε μία νέα μορφή της μεθόδου *GetQuery* (*Overloaded Version* – εικόνα 87), ώστε να μπορεί να μας επιστρέφει ένα αποτέλεσμα τύπου <*TResult*>.

```
    //Projection case (Brands + Types)
    0 references
    public static IQueryable<TResult> GetQuery<TSpec, TResult>
    (IQueryable<T> query, ISpecification<T, TResult> spec)
    {
        if (spec.Criteria != null)
        {
            query = query.Where(spec.Criteria); //FOR EXAMPLE checks if x => x.Brand == brand
        }
        //FOR EXAMPLE  "priceAsc" => query.OrderBy(x => x.Price)
        if(spec.OrderBy != null)
        {
            query = query.OrderBy(spec.OrderBy);

        }
        //FOR EXAMPLE "priceDesc" => query.OrderByDescending(x => x.Price)
        if (spec.OrderByDescending != null)
        {
            query = query.OrderByDescending(spec.OrderByDescending);
        }

        //Brands or Types query
        var selectQuery = query as IQueryable<TResult>;

        if (spec.Select != null)
        {
            selectQuery = query.Select(spec.Select);
        }

        return selectQuery ?? query.Cast<TResult>() ; //in case selectquery is null
    }
}
```

Εικόνα 87

Έχοντας πραγματοποιήσει τις απαραίτητες αλλαγές στις κλάσεις του *Specification Pattern*, σειρά παίρνουν οι κλάσεις του *Generic Repository Pattern*. Ξεκινώντας από την *Interface* κλάση *IGenericRepository*, θα ορίσουμε δύο νέα *Tasks* για τις «*Projection*» εκδοχές των μεθόδων *GetEntityWithSpec* και *ListAsync*. Οι δύο μέθοδοι διατηρούν την ίδια ακριβώς ονομασία, παρόλα αυτά ορίζονται ως τύπου *TResult* κι επιστρέφουν ένα αποτέλεσμα επίσης τύπου *TResult*. Ουσιαστικά αποτελούν δύο *Overloaded* εκδοχές των *Specification Pattern* μεθόδων, δεχόμενες διαφορετικού τύπου ορίσματα (εικόνα 88).

```
15      //projection case methods
        0 references
16      Task<TResult?> GetEntityWithSpec<TResult>(ISpecification<T, TResult> spec);
        0 references
17      Task<IReadOnlyList<TResult>> ListAsync<TResult>(ISpecification<T, TResult> spec);
18
```

Εικόνα 88

Σειρά έχει η *implementation* κλάση της *IgenericRepository*, η *GenericRepository*. Κάνουμε *implement* τις νέες μεθόδους που ορίσαμε στο προηγούμενο βήμα και δημιουργούμε μία *<TResult> overloaded* εκδοχή της μεθόδου *ApplySpecification*. Και οι τρεις μέθοδοι αποτελούν *overloaded* εκδοχές αυτών του *Generic Repository Pattern*, με τη διαφορά πως όλες είναι τύπου *TResult* και υποστηρίζουν το *Projection* (εικόνα 89).

```
65      //objection case (Brands + Types) - version 2 overloaded methods
        1 reference
66      public async Task<TResult?> GetEntityWithSpec<TResult>(ISpecification<T, TResult> spec)
67      {
68          return await ApplySpecification(spec).FirstOrDefaultAsync();
69      }
70
        1 reference
71      public async Task<IReadOnlyList<TResult>> ListAsync<TResult>(ISpecification<T, TResult> spec)
72      {
73          return await ApplySpecification(spec).ToListAsync();
74      }
75
76      //second version (Objection case) returns IQueryable of type TResult
77      //is used by ListAsync & GetEntityWithSpec overloaded methods
        2 references
78      private IQueryable<TResult> ApplySpecification<TResult>(ISpecification<T, TResult> spec)
79      {
80          return SpecificationEvaluator<T>.GetQuery<T,TResult>(context.Set<T>().AsQueryable(), spec);
81      }
82  }
```

Εικόνα 89

Σε αυτό το σημείο θα πρέπει να μεταβούμε στην κλάση *ISpecification* για να ορίσουμε μία *Boolean property* και θα της αποδώσουμε το όνομα *IsDistinct* (εικόνα 90).

```
        0 references
12      bool IsDistinct { get; } //Brands & types case
13  }
```

Εικόνα 90

Ακολούθως θα ανοίξουμε την κλάση *BaseSpecification*, ώστε να κάνουμε *implement* τη νέα μέθοδο και να την ορίσουμε (εικόνα 91).

```
BaseSpecification.cs
Core > Specifications > BaseSpecification.cs > BaseSpecification > BaseSpecification
using System.Linq.Expressions;
using Core.Interfaces;

namespace Core.Specifications;

//primary constructor implementing the ISpecification expression criteria
5 references
public class BaseSpecification<T>(Expression<Func<T, bool>>? criteria) : ISpecification<T>
{
    //empty constructor definition for use in ProductSpecification class
    //when we are using primary constructor syntax
    //criteria (brand,type,sorting via query are optional!)
    0 references
    protected BaseSpecification() : this(null) {} //in case no criteria are given
    5 references
    public Expression<Func<T, bool>>? Criteria => criteria;

    //sorting info - expressions support implementation
    6 references
    public Expression<Func<T, object>>? OrderBy { get; private set; }

    6 references
    public Expression<Func<T, object>>? OrderByDescending { get; private set; }

    2 references
    public bool IsDistinct { get; private set; }

    2 references
    protected void AddOrderBy(Expression<Func<T, object>> orderByExpression)
    {
        OrderBy = orderByExpression;
    }
    1 reference
    protected void AddOrderByDescending(Expression<Func<T, object>> orderByDescExpression)
    {
        OrderByDescending = orderByDescExpression;
    }

    0 references
    protected void ApplyDistinct()
    {
        IsDistinct = true;
    }
}
```

Εικόνα 91

Εν συνεχεία θα μεταβούμε στην κλάση *SpecificationEvaluator* για να ορίσουμε κι εκεί τη λειτουργικότητα της μεθόδου *Distinct* (εικόνες 92 και 93).

```
public class SpecificationEvaluator<T> where T : BaseEntity
{
    //This method evaluates the given criteria to the one of the ISpecification's expression
    1 reference
    public static IQueryable<T> GetQuery(IQueryable<T> query, ISpecification<T> spec)
    {
        if (spec.Criteria != null)
        {
            query = query.Where(spec.Criteria); //FOR EXAMPLE checks if x => x.Brand == brand
        }
        //FOR EXAMPLE  "priceAsc" => query.OrderBy(x => x.Price)
        if (spec.OrderBy != null)
        {
            query = query.OrderBy(spec.OrderBy);

        }
        //FOR EXAMPLE "priceDesc" => query.OrderByDescending(x => x.Price)
        if (spec.OrderByDescending != null)
        {
            query = query.OrderByDescending(spec.OrderByDescending);
        }

        if (spec.IsDistinct)
        {
            query = query.Distinct();
        }

        return query;
    }
```

Εικόνα 92

```
//Projection case (Brands + Types) + specif. pattern
1 reference
public static IQueryable<TResult> GetQuery<TSpec, TResult>
(IQueryable<T> query, ISpecification<T, TResult> spec)
{
    if (spec.Criteria != null)
    {
        query = query.Where(spec.Criteria); //FOR EXAMPLE checks if x => x.Brand == brand
    }
    //FOR EXAMPLE  "priceAsc" => query.OrderBy(x => x.Price)
    if (spec.OrderBy != null)
    {
        query = query.OrderBy(spec.OrderBy);

    }
    //FOR EXAMPLE "priceDesc" => query.OrderByDescending(x => x.Price)
    if (spec.OrderByDescending != null)
    {
        query = query.OrderByDescending(spec.OrderByDescending);
    }

    //Brands or Types query
    var selectQuery = query as IQueryable<TResult>;

    if (spec.Select != null)
    {
        selectQuery = query.Select(spec.Select);
    }

    //distinct versions for types & brands
    if (spec.IsDistinct)
    {
        selectQuery = selectQuery?.Distinct(); //optional in case of beeing null
    }

    return selectQuery ?? query.Cast<TResult>(); //in case select query is null
}
```

Εικόνα 93

Σε αυτό το βήμα θα πρέπει να δημιουργήσουμε δύο νέες κλάσεις εντός του καταλόγου *Specifications*. Η πρώτη κλάση ονομάζεται *BrandListSpecification* κι ορίζεται σύμφωνα με τον κώδικα της εικόνας 94 ενώ η δεύτερη ονομάζεται *TypeListSpecification* κι ακολουθεί τη λογική της πρώτης (εικόνα 95).

BrandListSpecification.cs

Core > Specifications > BrandListSpecification.cs > ...

```
1   using Core.Entities;
2
3   namespace Core.Specifications;
4
    1 reference
5   public class BrandListSpecification : BaseSpecification<Product, string>
6   {
        0 references
7       public BrandListSpecification()
8       {
9           AddSelect(x => x.Brand);
10          ApplyDistinct();
11      }
12  }
```

Εικόνα 94

TypeListSpecification.cs

Core > Specifications > TypeListSpecification.cs > ...

```
1   using Core.Entities;
2
3   namespace Core.Specifications;
4
    1 reference
5   public class TypeListSpecification : BaseSpecification<Product, string>
6   {
        0 references
7       public TypeListSpecification()
8       {
9           AddSelect(x => x.Type);
10          ApplyDistinct();
11      }
12  }
```

Εικόνα 95

Πλέον μπορούμε να μεταβούμε στην κλάση *ProductsController* και πιο συγκεκριμένα στις μεθόδους *GetBrands* και *GetTypes*. Για την *GetBrands* θα κάνουμε *implement* τη μέθοδο *BrandListSpecification* κι αντιστοίχως για την *GetTypes*, θα κάνουμε *implement* την *evaluator* κλάση *TypeListSpecifiation*. Το αποτέλεσμα του *query* που μας επιστρέφεται το αποθηκεύουμε σε μια μεταβλητή (*spec*) και το επιστρέφουμε στον *client* με χρήση της μεθόδου *ListAsync* (εικόνα 96).

```
//query products brands list
[HttpGet("brands")]
0 references
public async Task<ActionResult<IReadOnlyList<string>>> GetBrands()
{
      var spec = new BrandListSpecification();

      return Ok(await repo.ListAsync(spec));
}

//query products types list
[HttpGet("types")]
0 references
public async Task<ActionResult<IReadOnlyList<string>>> GetTypes()
{
      var spec = new TypeListSpecification();

      return Ok(await repo.ListAsync(spec));
}
```

Εικόνα 96

3.9 Εξετάζοντας την Λειτουργικότητα της Εφαρμογής

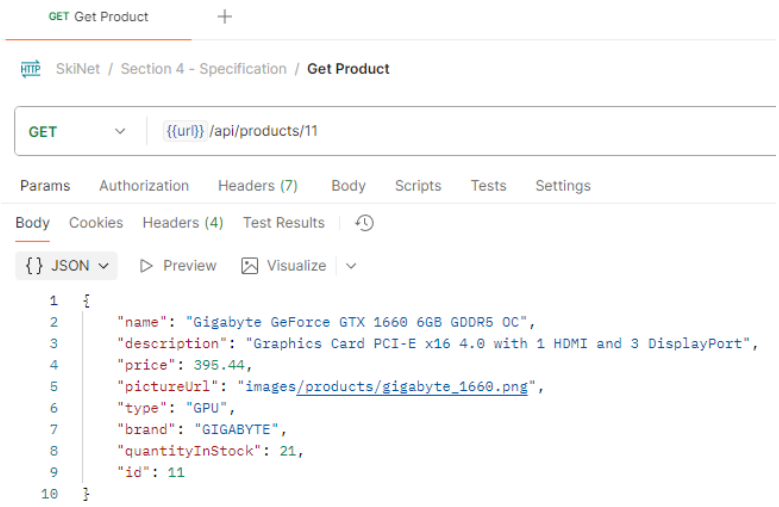
Στην τελευταία ενότητα αυτού του κεφαλαίου θα δοκιμάσουμε τη μέχρι στιγμής λειτουργικότητα της εφαρμογής μας, μετά και την προσθήκη των κλάσεων *Generic Repository Pattern*, των κλάσεων του *Specification Pattern* καθώς και τη χρήση του *Projection*. Χρησιμοποιώντας το περιβάλλον της εφαρμογής *Postman*, θα εκτελέσουμε την αναζήτηση συγκεκριμένων *URLs* και θα παρουσιάσουμε τα αποτελέσματά τους στις εικόνες που ακολουθούν.

- https://localhost:5001/api/products - Εμφάνιση όλων των προϊόντων της βάσης δεδομένων.

Εικόνα 97

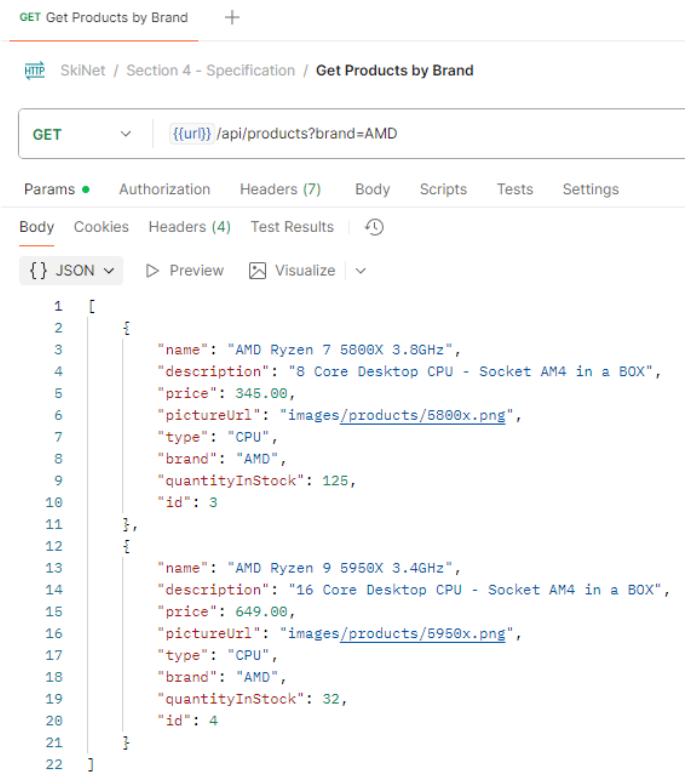
Εικόνα 98

- https://localhost:5001/api/products/11 - Εμφάνιση του προϊόντος με Id που ισούται με 11.



Εικόνα 99

- https://localhost:5001/api/products?brand=AMD – Εμφάνιση όλων των προϊόντων μάρκας AMD



Εικόνα 100

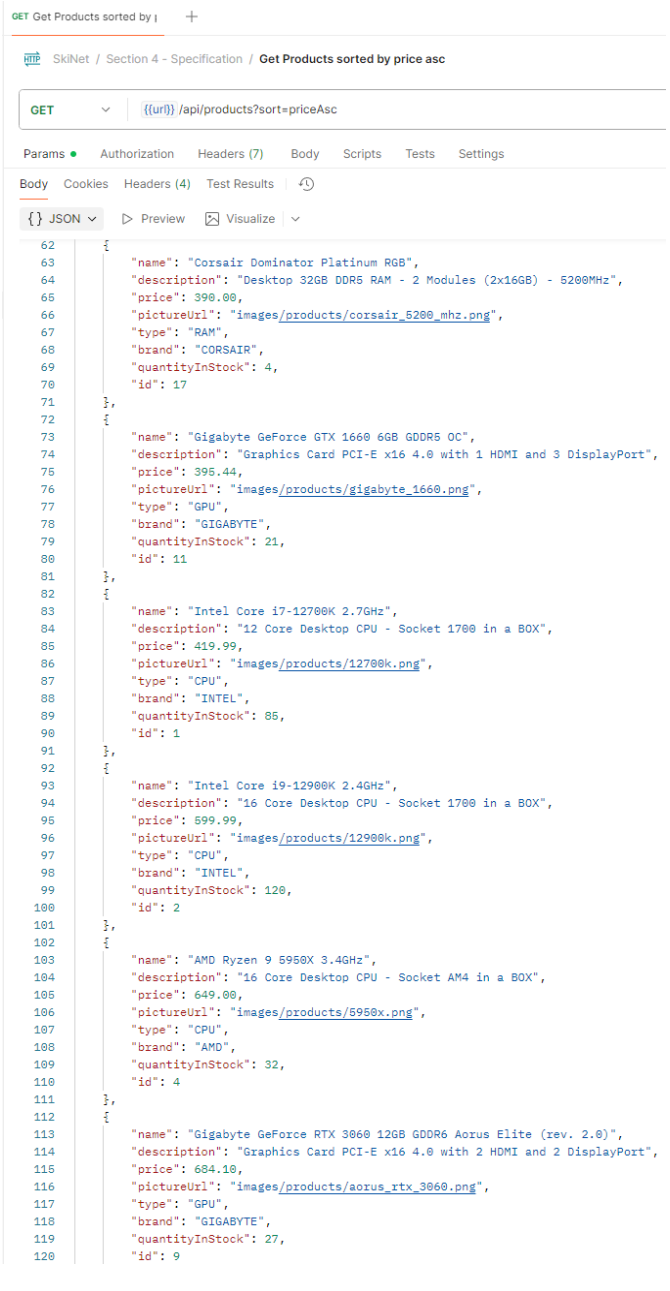
- https://localhost:5001/api/products?type=GPU – Εμφάνιση όλων των προϊόντων τύπου GPU


GET Get Products by Type
SkiNet / Section 4 - Specification / Get Products by Type
GET
{{url}} /api/products?type=GPU
Params
Authorization
Headers (7)
Body
Scripts
Tests
Settings
Body
Cookies
Headers (4)
Test Results
{} JSON
Preview
Visualize
[
{
"name": "Asus GeForce RTX 3070 Ti 8GB GDDR6X ROG Strix OC",
"description": "Graphics Card PCI-E x16 4.0 with 2 HDMI and 3 DisplayPort",
"price": 1073.64,
"pictureUrl": "images/products/ASUS_rtx_3070ti.png",
"type": "GPU",
"brand": "ASUS",
"quantityInStock": 10,
"id": 5
},
{
"name": "Asus GeForce RTX 3080 Ti 12GB GDDR6X ROG Strix LC OC",
"description": "Graphics Card PCI-E x16 4.0 with 2 HDMI and 3 DisplayPort",
"price": 2490.90,
"pictureUrl": "images/products/ASUS_rtx_3080ti.png",
"type": "GPU",
"brand": "ASUS",
"quantityInStock": 12,
"id": 6
},
{
"name": "Gigabyte GeForce GTX 1660 6GB GDDR5 OC",
"description": "Graphics Card PCI-E x16 4.0 with 1 HDMI and 3 DisplayPort",
"price": 395.44,
"pictureUrl": "images/products/gigabyte_1660.png",
"type": "GPU",
"brand": "GIGABYTE",
"quantityInStock": 21,
"id": 11
},
{
"name": "Gigabyte GeForce RTX 3060 12GB GDDR6 Aorus Elite (rev. 2.0)",
"description": "Graphics Card PCI-E x16 4.0 with 2 HDMI and 2 DisplayPort",
"price": 684.10,
"pictureUrl": "images/products/aorus_rtx_3060.png",
"type": "GPU",
"brand": "GIGABYTE",
"quantityInStock": 27,
"id": 9
},
{
"name": "Gigabyte GeForce RTX 3060 Ti 8GB GDDR6 Aorus Elite (rev. 2.0)",
"description": "Graphics Card PCI-E x16 4.0 with 2 HDMI and 2 DisplayPort",
"price": 850.00,
"pictureUrl": "images/products/gigabyte_rtx_3060ti.png",
"type": "GPU",
"brand": "GIGABYTE",
"quantityInStock": 35,
"id": 10
},
{
"name": "MSI GeForce RTX 3080 12GB GDDR6X Suprim X",
"description": "Graphics Card PCI-E x16 4.0 with 1 HDMI and 3 DisplayPort",
"price": 1600.00,
"pictureUrl": "images/products/MSI_rtx_3080_suprim_x.png",
"type": "GPU",
"brand": "MSI",
"quantityInStock": 5,

Εικόνα 101

- https://localhost:5001/api/products?sort=priceAsc – Ταξινόμηση όλων των προϊόντων κατά αύξουσα τιμή πώλησης.

Εικόνα 102



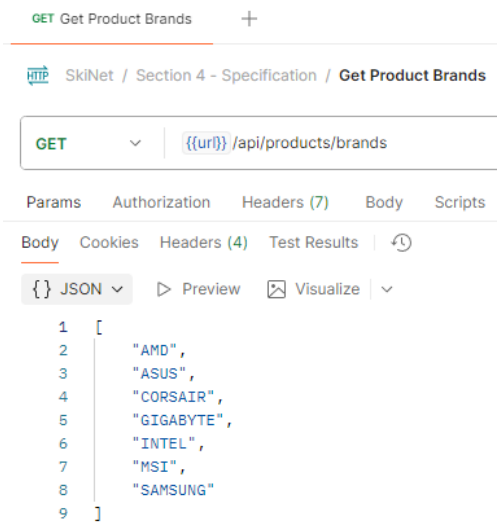
Εικόνα 103

- https://localhost:5001/api/products?sort=priceDesc - Ταξινόμηση όλων των προϊόντων κατά φθίνουσα τιμή πώλησης.

Εικόνα 104

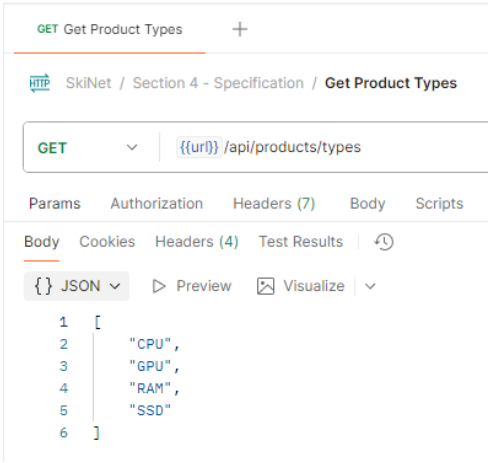
Εικόνα 105

- https://localhost:5001/api/products/brands - Παρουσίαση με όλες τις μάρκες των προϊόντων της βάσης δεδομένων.



Εικόνα 106

- https://localhost:5001/api/products/types - Παρουσίαση με όλους τους τύπους των προϊόντων της βάσης δεδομένων.



Εικόνα 107

Κεφάλαιο 4. Σύνθετο Φιλτράρισμα, Σελιδοποίηση, Αναζήτηση

Σε αυτό το κεφάλαιο θα υλοποιήσουμε τις λειτουργικότητες της σελιδοποίησης (*Pagination*), του σύνθετου φιλτραρίσματος των προϊόντων (*Filtering*) και της βασικής αναζήτησης προϊόντων (*Searching*). Βασιζόμενοι στα μοτίβα ανάπτυξης *Generic Repository Pattern* και *Specification Pattern*, δώσαμε τη δυνατότητα του βασικού φιλτραρίσματος στον χρήστη μέσω της μεθόδου *GetProducts*, εντός του *ProductsController*. Τα ορίσματα αυτής της μεθόδου θα δοκιμάσουμε να τα αποθηκεύσουμε εντός ενός αντικειμένου (*Object*). Αρχικά, εντός του καταλόγου *Specifications*, του *Core Project*, θα ορίσουμε μία νέα κλάση με τίτλο *ProductSpecParams*. Εκεί θα ορίσουμε τις απαραίτητες *Properties* και τα *Backing Fields*, ώστε να υλοποιήσουμε τη λειτουργικότητα του σύνθετου φιλτραρίσματος με λέξεις κλειδιά (*Brand or Type or both of them*) καθώς και της ταξινόμησης, όπως βλέπουμε στην εικόνα 108. Ουσιαστικά η κλάση αυτή δέχεται το αίτημα από τον *client* (*Get*) και αναλύει τα περιεχόμενα του κατακερματίζοντάς το ανάλογα με τις λέξεις κλειδιά (Χρήση της μεθόδου *Split*). Στη συνέχεια το αποτέλεσμα αποθηκεύεται (*Set*) σε μορφή συμβολοσειράς στην κατάλληλη *Property (_types*, *_brands*).

ProductSpecParams.cs

Core > Specifications > ProductSpecParams.cs > ...

```
1   namespace Core.Specifications;
2
    2 references
3   public class ProductSpecParams
4   {
        2 references
5       private List<string> _brands = []; //set to an emprty string list
6       //declare our properties
7       // using private backing field scope
        3 references
8       public List<string> Brands
9       {
10          get => _brands; //example: brand=SAMSUNG,AMD,INTEL
11          set
12          {   //separates the above string
13              _brands = value.SelectMany(X => X.Split(',',
14                  StringSplitOptions.RemoveEmptyEntries)).ToList();
15          }
16      }
17
        2 references
18      private List<string> _types = []; //set to an emprty string list
19      //declare our properties
20      // using private backing field scope
        1 reference
21      public List<string> Types
22      {
23          get => _types; //example: types=GPU,CPU,RAM
24          set
25          {   //separates the above string
26              _types = value.SelectMany(X => X.Split(',',
27                  StringSplitOptions.RemoveEmptyEntries)).ToList();
28          }
29      }
30
31      //sorting optional parameter
        1 reference
32      public string? Sort { get; set; }
33  }
```

Εικόνα 108

Επόμενο βήμα, η μετάβασή μας στην κλάση *ProductSpecification* ώστε να παραμετροποιήσουμε τα περιεχόμενά της, με σκοπό να εκμεταλλευτούμε την πληροφορία που μας μεταφέρεται από την κλάση *ProductSpecParams*. Ουσιαστικά, αντικαθιστούμε τις παραμέτρους *brand*, *type*, *sort* με την παράμετρο *ProductSpecParams specParams*, καθώς επίσης μεταβάλουμε και τον ορισμό των ελέγχων αναφορικά με την *BaseSpecification* κλάση, όπως παρατηρούμε στην εικόνα 109.

ProductSpecification.cs

Core > Specifications > ProductSpecification.cs > ...

```
1  using Core.Entities;
2
3  namespace Core.Specifications;
4
   2 references
5  public class ProductSpecification : BaseSpecification<Product>
6  {
7      //creation of an expression using (optionally) the brand and type from client
8      //in order to parse it to products controller class methods and add it to specific queries
       1 reference
9      public ProductSpecification(ProductSpecParams specParams) : base(x =>
10         (specParams.Brands.Count == 0 || specParams.Brands.Contains(x.Brand)) &&
11         (specParams.Types.Count == 0 || specParams.Types.Contains(x.Type))
12     )
13     {
14         //sorting case
15         switch (specParams.Sort)
16         {
17             case "priceAsc":
18                 AddOrderBy(x => x.Price);
19                 break;
20             case "priceDesc":
21                 AddOrderByDescending(x => x.Price);
22                 break;
23             default:
24                 AddOrderBy(x => x.Name);
25                 break;
26         }
27     }
28 }
```

Εικόνα 109

Τέλος, μεταβαίνουμε στον *ProductsController* και τη μέθοδο *GetProducts*. Ομοίως, θα αντικαταστήσουμε τα ορίσματα της μεθόδου με το *object ProductSpecparams specParams* και λόγω της φύσεως ενός τέτοιου ορίσματος, θα πρέπει να εισάγουμε και το *attribute* [*FromQuey*] για τον *API controller*, ώστε να ξέρει που να αναζητήσει την πληροφορία από τη συγκεκριμένη παράμετρο (εικόνα 110).

```
//http endpoints
[HttpGet] //return asynchronously a task (action result) in http format as a list (a list of products in our
//url: api/products
//public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
0 references
public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts([FromQuery] ProductSpecParams specParams)
{

    var spec = new ProductSpecification(specParams); //using specifications to format the query
    var products = await repo.ListAsync(spec); //applying the query to infrastructure repository classes

    return Ok(products); //returns the filtered list of products to the client
}
```

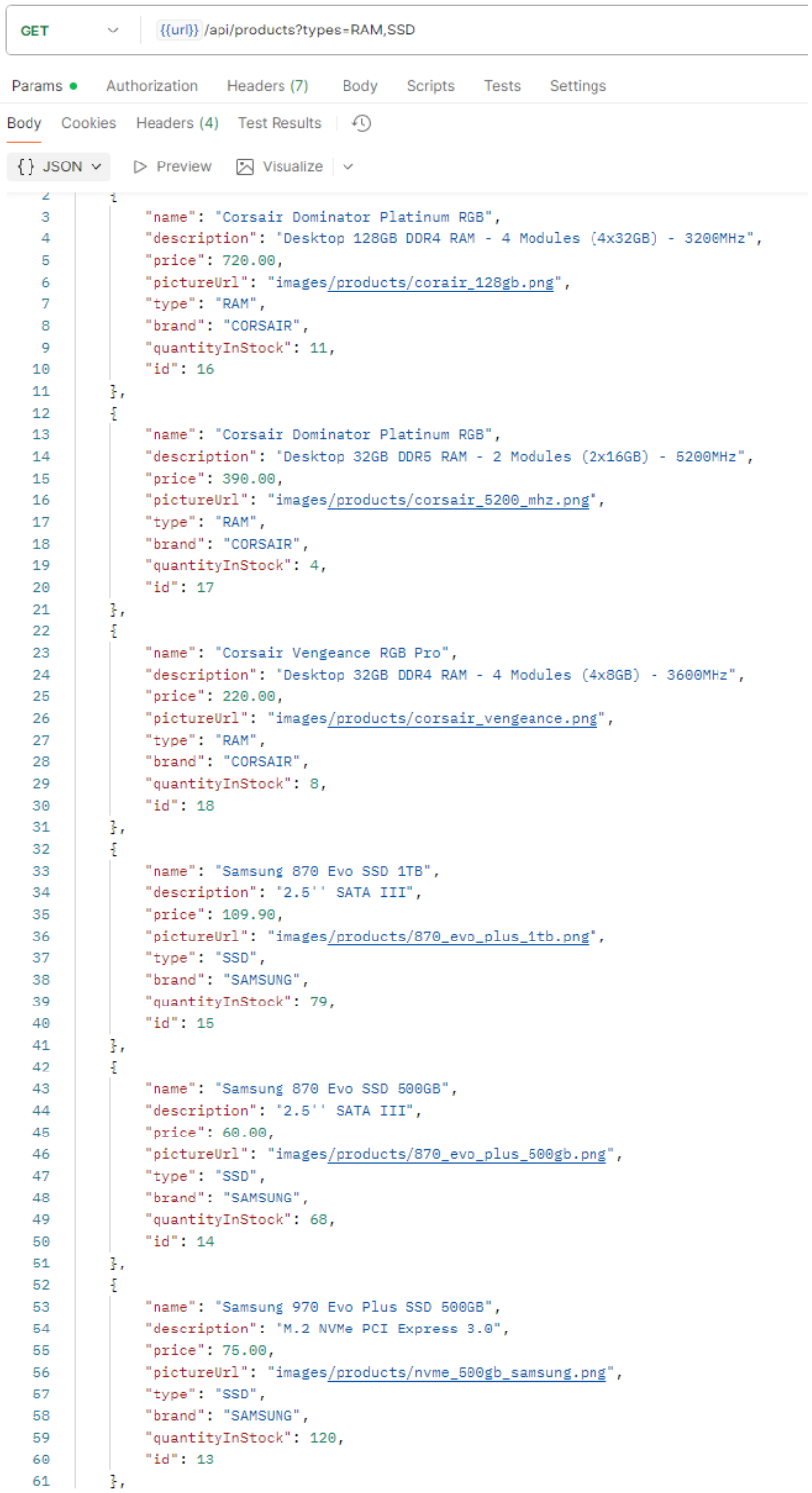
Εικόνα 110

Εκτελώντας μία αναζήτηση, στην εφαρμογή *Postman*, για προϊόντα από συγκεκριμένες μάρκες (εικόνα 111) ή τύπους (εικόνα 112), ή εκτελώντας σύνθετη αναζήτηση ανά μάρκα και τύπο (εικόνα 113), επιστρέφονται τα αποτελέσματα των εικόνων όπως παρουσιάζονται ακολούθως.

- https://localhost:5001/api/products?brands=AMD,INTEL

Εικόνα 111

- https://localhost:5001/api/products?types=RAM,SSD



Εικόνα 112

- https://localhost:5001/api/products?brands=AMD,ASUS&types=CPU,GPU

Εικόνα 113

4.1 Σελιδοποίηση

Σε αυτό το σημείο θα ξεκινήσουμε την υλοποίηση μίας ακόμη πολύ σημαντικής λειτουργικότητας για την εφαρμογή μας, αυτήν της σελιδοποίησης. Κάθε *e-commerce web application* οφείλει να υποστηρίζει τη δυνατότητα της σελιδοποίησης και να παρουσιάζει τον κατάλογο των προϊόντων της ως συγκεκριμένο αριθμό και ομαδοποιημένα ανά σελίδα. Εδώ θα εφαρμόσουμε τη λογική του *Skip – Take*. Με άλλα λόγια, ο χρήστης θα δίνει τον αριθμό της σελίδας την οποία θέλει να προβάλει και η εφαρμογή μας θα παρακάμπτει τόσα προϊόντα όσα χρειάζονται για να προβάλει αυτά της ζητούμενης σελίδας (*Offset Pagination*). Εάν λοιπόν, για παράδειγμα, ο χρήστης επιθυμεί να μεταβεί στη σελίδα τρία και κάθε σελίδα έχει μέγεθος έξι (προβάλλονται έξι προϊόντα ανά σελίδα), θα πρέπει να γίνει *skip* δώδεκα προϊόντων και να λάβουμε (*take*) τα αμέσως επόμενα έξι. Αυτά τα δεδομένα σε συνάρτηση με τις σταθερές που θα ορίσουμε, θα σχηματίσουν την υποδομή και τους κανόνες της σελιδοποίησης στην εφαρμογή μας. Ξεκινώντας, θα ανοίξουμε την *Interface* κλάση *ISpecification* και θα προσθέσουμε τις *Properties* της εικόνας 114.

```
14      //pagination properties
        0 references
15      int Take { get; }
        0 references
16      int Skip { get; }
        0 references
17      bool IsPagingEnabled { get; }
18
19  }
```

Εικόνα 114

Στη συνέχεια ανοίγουμε την κλάση *BaseSpecification* ώστε να κάνουμε *Implement* τις νέες *properties* (εικόνα 115).

```
22      //pagination properties implementation
        1 reference
23      public int Take { get;  private set;}
24
        1 reference
25      public int Skip { get;  private set;}
26
        1 reference
27      public bool IsPagingEnabled { get;  private set;}
```

Εικόνα 115

Στην ίδια κλάση ορίζουμε τη λειτουργικότητα των τριών *Implemented Properties* όπως βλέπουμε στην εικόνα 116.

```
43      //PAGINATION PROPERTIES SETUP
        0 references
44      protected void ApplyPaging(int skip, int take)
45      {
46          Skip = skip;
47          Take = take;
48          IsPagingEnabled = true;
49      }
```

Εικόνα 116

Ακολούθως θα μεταβούμε στην κλάση *SpecificationEvaluator* προκειμένου να ενημερώσουμε τον τρόπο ενημέρωσης των *queries*, προσθέτοντας τη λειτουργικότητα της σελιδοποίησης στη μέθοδο *Getquery* (εικόνα 117). Το ίδιο θα πράξουμε και στην *overloaded - Projection GetQuery* μέθοδο, εντός της ίδιας κλάσης (εικόνα 118).

```
33          //pagination query setup
34          if (spec.IsPagingEnabled)
35          {
36              query = query.Skip(spec.Skip).Take(spec.Take);
37          }
```

Εικόνα 117

```
76          //pagination query setup
77          if (spec.IsPagingEnabled)
78          {
79              selectQuery = selectQuery?.Skip(spec.Skip).Take(spec.Take);
80          }
```

Εικόνα 118

Αφού έχουμε προχωρήσει στις απαραίτητες προσθήκες εντός των *Specification Pattern* κλάσεων, θα μεταβούμε στην κλάση *ProductSpecParams*. Εδώ θα ορίσουμε τους *Pagination Operators* (σταθερές) που

θα μας χρειαστούν, ώστε να ορίσουμε τα κριτήρια λειτουργίας της σελιδοποίησης. Καταρχάς, θα ορίσουμε τον μέγιστο αριθμό προϊόντων που μπορεί να αιτηθεί ο *client* ανά σελίδα, καθώς και το προκαθορισμένο μέγεθος (αριθμό προϊόντων) που είναι δυνατόν να προβληθούν σε μία σελίδα. Εν συνεχεία, θα ορίσουμε μια *backing field property* όπου καθορίζεται το μέγεθος της σελίδας με βάση το αίτημα του χρήστη, με τον περιορισμό να μην μπορεί να υπερβεί το μέγιστο μέγεθός της, το οποίο και ορίσαμε στη μεταβλητή *MaxPageSize* (εικόνα 119).

```
//pagination Skip - Take (Offset Pagination logic)
//max item number could be requested
2 references
private const int MaxPageSize = 50;
0 references
public int PageIndex { get; set; } = 1; //default page number
2 references
private int _pageSize = 6; //default page size
2 references
public int PageSize //backing field property to set the size of page
{
    get => _pageSize;
    set => _pageSize = (value > MaxPageSize) ? MaxPageSize : value;
}
```

Εικόνα 119

Εντός του *Core Project* και πιο συγκεκριμένα εντός της κλάσης *ProductSpecification*, θα χρησιμοποιήσουμε τη μέθοδο *ApplyPaging*, στην οποία ορίζουμε τον αριθμό (*Skip*) των προϊόντων που θέλουμε να παρακαμφθούν και τον αριθμό (*Take*) των προϊόντων που θέλουμε να προβληθούν. Πολλαπλασιάζοντας τον αριθμό (*Skip*) με τον αριθμό της σελίδας στην οποία βρισκόμαστε (*PageIndex*), μειωμένο κατά ένα (*1*), προκύπτει ο αριθμός της σελίδας που θα μεταβούμε με βάση τον αριθμό (*Take*), των προϊόντων που θέλουμε να προβληθούν (εικόνα 120).

```
14          //offset pagination method
15          ApplyPaging(specParams.PageSize * (specParams.PageIndex - 1), specParams.PageSize);
16
```

Εικόνα 120

Αυτήν την πληροφορία (*Pagination Information*), καθώς και τη λίστα των προϊόντων που προκύπτει, θα τη στείλουμε μέσω ενός αντικειμένου (*Object*) σε μία νέα κλάση. Την κλάση αυτή την ονομάζουμε *Pagination* και την ορίζουμε εντός ενός νέου καταλόγου με τίτλο *RequestHelpers*. Τον κατάλογο *RequestHelpers* τον δημιουργούμε εντός του *API Project*. Στην εικόνα 121 δημιουργούμε όλες τις απαραίτητες οντότητες (*PageIndex, PageSize, Count, Data*) εντός της *generic* κλάσης *Pagination*.

Pagination.cs ×

API > RequestHelpers > Pagination.cs > ...

```
1   namespace API.RequestHelpers;
2
3   //primary constructor methodology for generic class pagination
    0 references
4   public class Pagination<T>(int pageIndex, int pageSize, int count, IReadOnlyList<T> data )
5   {
6       //pagination properties to reutn to the client
        0 references
7       public int PageIndex { get; set; } = pageIndex;
        0 references
8       public int PageSize { get; set; } = pageSize;
        0 references
9       public int Count { get; set; } = count; //total number of items requested
        0 references
10      public IReadOnlyList<T> Data { get; set; } = data;
11  }
```

Εικόνα 121

Το πρόβλημα που ανακύπτει είναι η ανάγκη υπολογισμού του συνολικού αριθμού προϊόντων, που προκύπτει κατά το φιλτράρισμα του *HTTP* αιτήματος από τον *client*. Αυτός ο αριθμός θέλουμε να υπολογίζεται αφού έχει ολοκληρωθεί το φιλτράρισμα για τη δημιουργία του κατάλληλου *query* προς τη βάση, να αποθηκεύεται στην οντότητα *Count* κι αυτό να συμβαίνει πριν τη διαδικασία της σελιδοποίησης. Για τον λόγο αυτό θα μεταβούμε στην *Interface* κλάση *IGenericRepository*, όπου θα δημιουργήσουμε μία νέα μέθοδο με τίτλο *CountAsync*. Στην ουσία πρόκειται για ένα *Task* τύπου *Int*, το οποίο δέχεται μία παράμετρο *ISpecification<T>* (εικόνα 121Α).

```
24      //total items number count for pagination
        0 references
25      Task<int> CountAsync(ISpecification<T> spec);
```

Εικόνα 121Α

Προκειμένου να χρησιμοποιήσουμε τη μέθοδο *CountAsync*, θα ανοίξουμε την *Interface* κλάση *ISpecification* όπου δημιουργούμε μία νέα οντότητα. Η νέα οντότητα είναι τύπου *IQueryable<T>,* την ονομάζουμε *ApplyCriteria* και ως παράμετρο δέχεται μία μεταβλητή τύπου *IQueryable<T>* με όνομα *query* (εικόνα 122).

```
14      //pagination properties
        4 references
15      int Take { get; }
        4 references
16      int Skip { get; }
        4 references
17      bool IsPagingEnabled { get; }
18      //criteria for total items count for pagination
        0 references
19      IQueryable<T> ApplyCriteria(IQueryable<T> query);
```

Εικόνα 122

Επόμενος σταθμός, η κλάση *BaseSpecification*. Εδώ θα κάνουμε *Implement* τη νέα μέθοδο *ApplyCriteria*. Ελέγχουμε την περίπτωση που το αίτημα από τον *client* περιέχει κριτήρια αναζήτησης (εικόνα 123).

```
28      //if we have criteria for total items count to use for pagination
        1 reference
29      public IQueryable<T> ApplyCriteria(IQueryable<T> query)
30      {
31          if (Criteria != null)
32          {
33              query = query.Where(Criteria);
34          }
35
36          return query;
37      }
```

Εικόνα 123

Εν συνεχεία, στην κλάση *GenericRepository* θα κάνουμε *Implement* τη μέθοδο *CountAsync*. Ο ρόλος αυτής της μεθόδου είναι να αποθηκεύσει, προσωρινά, το περιέχον κριτήρια αναζήτησης *query* σε μία μεταβλητή. Ακολούθως το εφαρμόζει στη βάση δεδομένων προκειμένου να καταμετρήσει το σύνολο των αποτελεσμάτων που επιστρέφεται. Τέλος τον αριθμό αυτό τον επιστρέφει στον *ProducsController* (εικόνα 124).

```
83      //count items and returns the number to productscontroller for pagination
        1 reference
84      public async Task<int> CountAsync(ISpecification<T> spec)
85      {
86          var query = context.Set<T>().AsQueryable(); //takes the query wtih specs
87          query = spec.ApplyCriteria(query); //apply the query to db
88          return await query.CountAsync(); //use the query result to count the items
89      }
```

Εικόνα 124

Τέλος, στον *ProductsController* θα αποθηκεύσουμε τον αριθμό των προϊόντων που προέκυψαν σε μία μεταβλητή (*count*), εκμεταλλευόμενοι τη μέθοδο *CountAsync*. Ακολούθως ορίζουμε ένα νέο *Instance* της κλάσης *Pagination* (έχοντας πλέον διαθέσιμες όλες τις παραμέτρους) και το αποθηκεύουμε στη μεταβλητή *pagination*. Το περιεχόμενο της μεταβλητής *Pagination* επιστρέφεται στον *client* (εικόνα 125).

```
public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts([FromQuery]ProductSpecParams specParams)
{

    var spec = new ProductSpecification(specParams); //using specifications to format the query
    var products = await repo.ListAsync(spec); //applying the query to infrastructure repository classes
    //gets total num of products
    // based on specs (query with criteria) for pagination
    var count = await repo.CountAsync(spec);

    var pagination = new Pagination<Product>(specParams.PageIndex,
        specParams.PageSize, count, products);

    return Ok(pagination); //returns the filtered list of products to the client
}
```

Εικόνα 125

Θέλοντας να δοκιμάσουμε τη λειτουργικότητα της σελιδοποίησης, θα μεταβούμε την εφαρμογή *Postman* και θα αναζητήσουμε το URL: https://localhost:5001/api/products?pageSize=3&pageIndex=5. Ουσιαστικά από ένα σύνολο δεκαοκτώ προϊόντων (σύνολο προϊόντων στη βάση δεδομένων pcparts.db) και μέγεθος σελίδας ίσο με τρία, αιτούμαστε μετάβαση στα προϊόντα της σελίδας πέντε. Το αποτέλεσμα του αιτήματός μας παρουσιάζεται στην εικόνα 126.

Εικόνα 126

4.2 Base API Controller

Θέλοντας να βελτιώσουμε την αποδοτικότητα της εφαρμογής μας και τη δομή του κώδικα εντός του *ProductsController*, θα δημιουργήσουμε έναν νέο *Controller* με τίτλο *BaseApiController.* Ο νέος *controller* θα οριστεί εντός του *API Project* και συγκεκριμένα εντός του καταλόγου *Controllers*. Από τη στιγμή που οι *Controllers* της εφαρμογής μας θα μοιράζονται τον ίδιο κώδικα για συγκεκριμένες λειτουργικότητες (για παράδειγμα την εμφάνιση προϊόντων, τη σελιδοποίηση, τα *HTTP Attributes*), θα μεταφέρουμε τον κώδικα που υλοποιεί τα παραπάνω στον *BaseApiController* (εικόνα 127). Ουσιαστικά πρόκειται για ένα ασύγχρονο *Task* με τίτλο *CreatePagedResult* τύπου *<T>* το οποίο δέχεται ως *attributes*, ένα *IGenericRepository<T> object*, ένα *ISpecifiaction<T> object*, τη μεταβλητή *pageIndex* και τη μεταβλητή *pageSize*, όπου *<T>* είναι τύπου *BaseEntity*, δηλαδή *Product*. Στη συνέχεια με χρήση της μεθόδου *Listasync*, η οποία μέσω του *Generic Repository Pattern object repo* και του *Specification Pattern object spec*, αποθηκεύει τη φιλτραρισμένη λίστα των προϊόντων στη μεταβλητή *items*. Ομοίως, με χρήση της μεθόδου *CountAsync*, θα αποθηκεύσουμε τον αριθμό αυτών των προϊόντων στη μεταβλητή *count*. Έχοντας διαθέσιμα όλα τα απαραίτητα ορίσματα για την *new instanced* κλάση *Pagination*, αποθηκεύουμε στη μεταβλητή *pagination* το αποτέλεσμά της, το οποίο επιστρέφεται στον εκάστοτε *controller*.

BaseApiController.cs ●

API > Controllers > BaseApiController.cs > ...

```
1   using API.RequestHelpers;
2   using Core.Entities;
3   using Core.Interfaces;
4   using Microsoft.AspNetCore.Mvc;
5
6   namespace API.Controllers;
7
8
9   //derives from aspnetcore mvc
10  //controller which returns API responses in jason format
11  [ApiController] //Attribute that improves developer experience during the bulding of an
12  [Route("api/[controller]")] //let server know where to send the incoming http request -
        1 reference
13  public class BaseApiController : ControllerBase //Base class for MVC controllers
14  {
15      //task which returns the items list and pagination to controllers that derives from
            1 reference
16      protected async Task<ActionResult> CreatePagedResult<T>(IGenericRepository<T> repo,
17          ISpecification<T> spec, int pageIndex, int pageSize) where T : BaseEntity
18      {
19          var items = await repo.ListAsync(spec); //lists items from query
20          var count = await repo.CountAsync(spec); //counts the total number of these iten
21
22          var pagination = new Pagination<T>(pageIndex, pageSize, count, items); //creates
23
24          return Ok(pagination); //returns result to other controller(s)
25      }
26  }
```

Εικόνα 127

Στην εικόνα 128 παρουσιάζονται οι αλλαγές στον *ProductsController* και στη μέθοδο *GetProducts*, ύστερα από τον ορισμό του *BaseApiController*.

```
8   //derives from BaseApiController
    0 references
9   public class ProductsController(IGenericRepository<Product> repo) : BaseApiController //using primary constructor meth. to implement IProductRepository interface/class
10  {
11      //http endpoints
12      [HttpGet] //return asynchronously a task (action result) in http format as a list (a list of products in our example) via GetProducts() method
13      //url: api/products
14      //public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
        0 references
15      public async Task<ActionResult<IReadOnlyList<Product>>> GetProducts([FromQuery] ProductSpecParams specParams)
16      {
17
18          var spec = new ProductSpecification(specParams); //using specifications to format the query
19          //returns CreatePagedResult result derived from BaseApiController
20          return await CreatePagedResult(repo,spec,specParams.PageIndex,specParams.PageSize); //returns the filtered list of products to the client with pagination
21      }
```

Εικόνα 128

Θέλοντας να ελέγξουμε τη λειτουργικότητα της εφαρμογής μας, στην πλατφόρμα του *Postman* θα αναζητήσουμε το παρακάτω URL: https://localhost:5001/api/products?pageSize=3&pageIndex=4 . Στην εικόνα 129 παρατηρούμε πως από ένα σύνολο 18 προϊόντων, μας επιστρέφονται τρία (*Page Size* ίσο με τρία), τα οποία βρίσκονται στη σελίδα τέσσερα.

GET {{url}} /api/products?pageSize=3&pageIndex=4

Params • Authorization Headers (7) Body Scripts Tests Settings

Body Cookies Headers (4) Test Results

{} JSON ▷ Preview Visualize


```
{
    "pageIndex": 4,
    "pageSize": 3,
    "count": 18,
    "data": [
        {
            "name": "Gigabyte GeForce RTX 3060 Ti 8GB GDDR6 Aorus Elite (rev. 2.0)",
            "description": "Graphics Card PCI-E x16 4.0 with 2 HDMI and 2 DisplayPort",
            "price": 850.00,
            "pictureUrl": "images/products/gigabyte_rtx_3060ti.png",
            "type": "GPU",
            "brand": "GIGABYTE",
            "quantityInStock": 35,
            "id": 10
        },
        {
            "name": "Intel Core i7-12700K 2.7GHz",
            "description": "12 Core Desktop CPU - Socket 1700 in a BOX",
            "price": 419.99,
            "pictureUrl": "images/products/12700k.png",
            "type": "CPU",
            "brand": "INTEL",
            "quantityInStock": 85,
            "id": 1
        },
        {
            "name": "Intel Core i9-12900K 2.4GHz",
            "description": "16 Core Desktop CPU - Socket 1700 in a BOX",
            "price": 599.99,
            "pictureUrl": "images/products/12900k.png",
            "type": "CPU",
            "brand": "INTEL",
            "quantityInStock": 120,
            "id": 2
        }
    ]
}
```


Εικόνα 129

4.3 Αναζήτηση

Στην τελευταία ενότητα αυτού του κεφαλαίου, θα εξετάσουμε την προσθήκη της λειτουργικότητας για αναζήτηση προϊόντων στη βάση δεδομένων *pcparts.db*. Ο χρήστης θα έχει τη δυνατότητα εισαγωγής μίας λέξεως «κλειδί», η οποία θα αναζητάται εντός της *property Name* της οντότητας *Product*, για το εκάστοτε προϊόν το οποίο εμπεριέχεται στο φιλτραρισμένο *query* προς τη βάση δεδομένων. Ουσιαστικά το «*Search Function*» είναι μία *Where clause*, ένα ακόμη φίλτρο που εφαρμόζεται στο εκάστοτε *query*. Ανοίγοντας την κλάση *ProductSpecParams*, θα δημιουργήσουμε μία νέα *property* με χρήση του backing field ορισμού. Στη μεταβλητή _search αποθηκεύεται η συμβολοσειρά προς αναζήτηση, σε μορφή πεζών χαρακτήρων (εικόνα 130).

```
45      //sorting optional parameter
        1 reference
46      public string? Sort { get; set; }
47
48      //searching backing field property
        2 references
49      private string? _search;
        0 references
50      public string Search
51      {
52          get => _search ?? ""; //key string or empty string case
53          set => _search = value.ToLower(); //string saved in lowercase
54      }
```

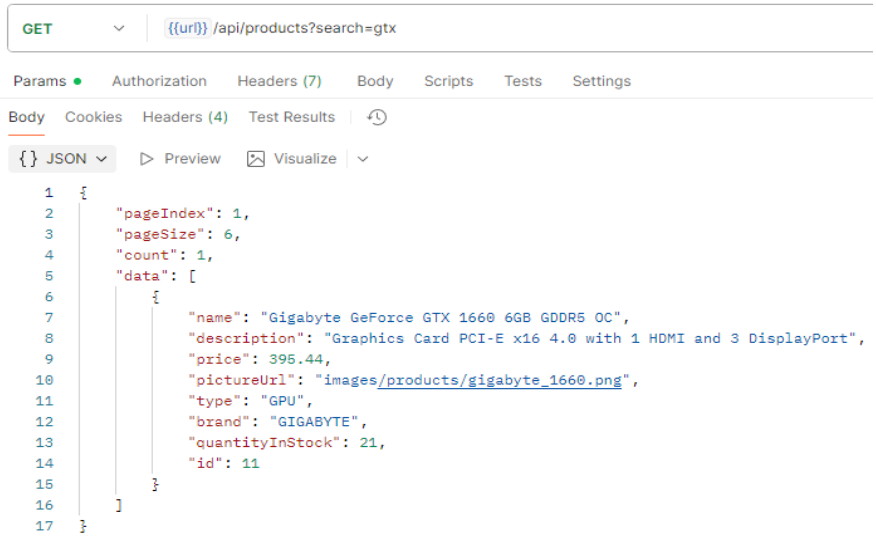
Εικόνα 130

Ακολούθως, θα μεταβούμε στην κλάση *ProductSpecification* και θα προσθέσουμε μία ακόμη συνθήκη (*Condition*) ως παράμετρο στον *primary cosnstructor* της κλάσης. Ουσιαστικά ελέγχουμε το ενδεχόμενο να έχει δοθεί «λέξη κλειδί» προς αναζήτηση. Σε περίπτωση που έχει δοθεί, συγκρίνεται με το όνομα κάθε προϊόντος κι αν βρεθεί προϊόν το οποίο πληροί τα κριτήρια αναζήτησης, προστίθεται στη λίστα του φιλτραρισμένου *query* (εικόνα 131).

```
public ProductSpecification(ProductSpecParams specParams) : base(x => //query formatting conditions
    (string.IsNullOrEmpty(specParams.Search) || x.Name.ToLower().Contains(specParams.Search)) &&
    (specParams.Brands.Count == 0 || specParams.Brands.Contains(x.Brand)) &&
    (specParams.Types.Count == 0 || specParams.Types.Contains(x.Type))
)
```

Εικόνα 131

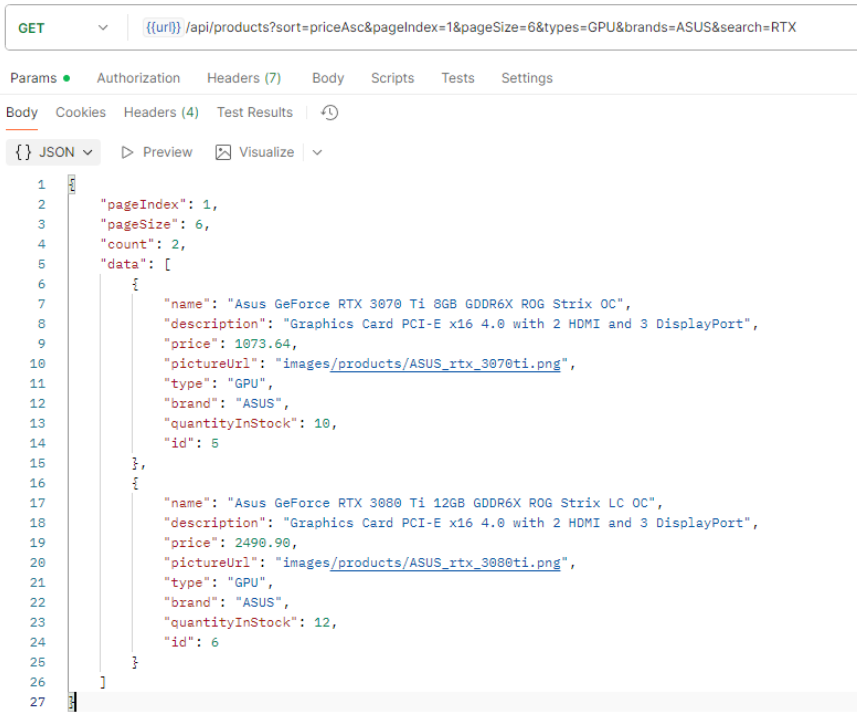
Στο περιβάλλον της εφαρμογής *Postman*, θα αναζητήσουμε ένα προϊόν που περιέχει στον τίτλο του τον όρο «*gtx*», χρησιμοποιώντας το URL: https://localhost:5001/api/products?search=gtx . Στην εικόνα 132 παρουσιάζεται το αποτέλεσμα που επιστρέφεται από την εφαρμογή μας. Το μοναδικό προϊόν που περιέχει την «λέξη κλειδί» εμφανίζεται στην οθόνη μας, επομένως η λειτουργικότητα της αναζήτησης προστέθηκε επιτυχώς και πλέον είναι διαθέσιμη στον *client*.



Εικόνα 132

Κλείνοντας αυτό το κεφάλαιο, θα δοκιμάσουμε την αναζήτηση ενός σύνθετου URL, με σκοπό να δοκιμάσουμε το σύνολο των μέχρι στιγμής δυνατοτήτων της εφαρμογής μας. Θα αιτηθούμε όλα τα προϊόντα από τη βάση δεδομένων *pcparts.db*, κατά αύξον κόστος, τύπου «*GPU*», μάρκας «*ASUS*», που περιέχουν τον όρο «*RTX*» στην περιγραφή του ονόματός τους, σε μέγεθος σελίδας έξι και προβολή της σελίδας νούμερο ένα. Στην εικόνα 133 παρουσιάζονται τα αποτελέσματα που επιστρέφονται στον *client*.

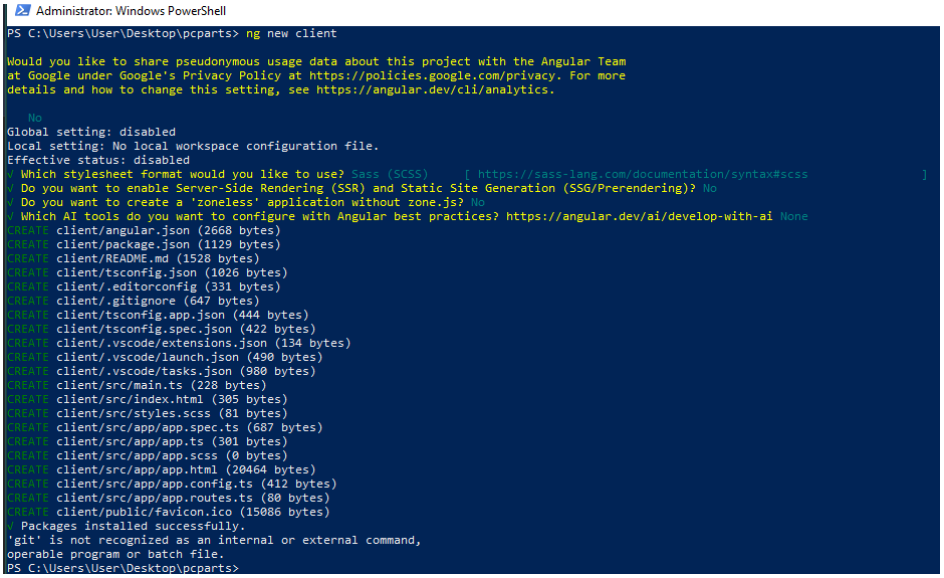
- URL:https://localhost:5001/api/products?sort=priceAsc&pageIndex=1&pageSize=6&types=GPU&brands=ASUS&search=RTX



Εικόνα 133

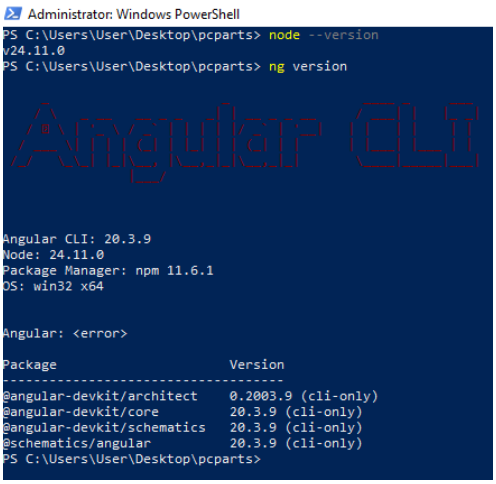
Κεφάλαιο 5. Angular

Η Angular είναι μια πλατφόρμα (*framework*) ανάπτυξης ιστοσελίδων, που βασίζεται στην *TypeScript*, για τη δημιουργία εφαρμογών ιστού μονής σελίδας (*single-page web applications).* Αναπτύχθηκε από την *Google* και παρέχει μια σουίτα εργαλείων, ένα *framework* που βασίζεται σε στοιχεία και βιβλιοθήκες για κοινές εργασίες, όπως η δρομολόγηση και η διαχείριση φορμών, βοηθώντας τους προγραμματιστές να δημιουργούν γρήγορες κι επεκτάσιμες εφαρμογές. Στην περίπτωση της εφαρμογής μας, η *Angular* θα μας βοηθήσει στο «χτίσιμο» της διεπαφής χρήστη (*User Interface*) και κατ' επέκταση στην επικοινωνία του *API* με τον *Client*. Αρχικά, εκτελώντας τις εντολές της εικόνας 134, θα εγκαταστήσουμε το *command line interface (CLI)* και τις απαραίτητες βιβλιοθήκες της πλατφόρμας, εντός του καταλόγου *pcparts*.



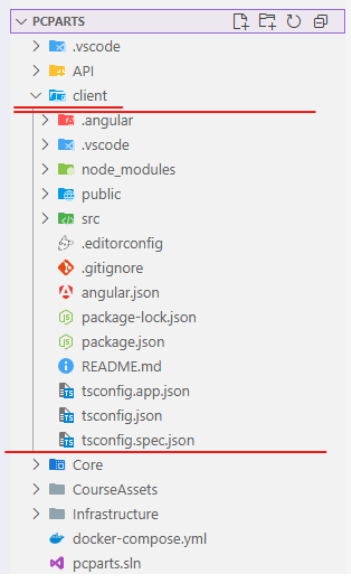
Εικόνα 134

Στην εικόνα 135, εκτελώντας τις εντολές *node –version* και *ng version,* λαμβάνουμε την πληροφορία για το ποια έκδοση της πλατφόρμας *Angular* διαθέτει ο υπολογιστής μας, καθώς και για τα υπόλοιπα εργαλεία που είναι διαθέσιμα μετά την εγκατάστασή της.



Εικόνα 135

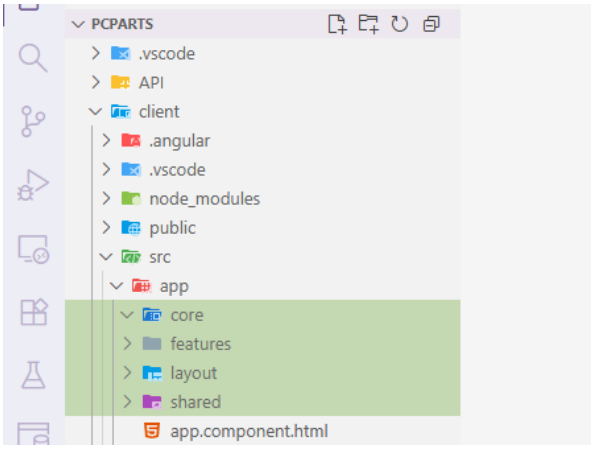
Πλέον, εκτελώντας στο περιβάλλον του *Visual Studio Code* την εφαρμογή μας κι ανοίγοντας τον *File Explorer*, παρατηρούμε την προσθήκη του καταλόγου *Client* με όλα τα απαραίτητα αρχεία κώδικα της πλατφόρμας *Angular* (εικόνα 136).



Εικόνα 136

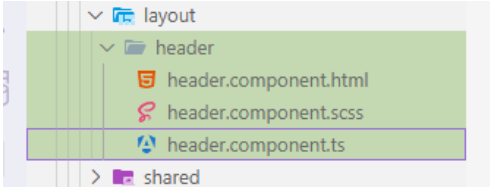
5.1 Δόμηση Καταλόγων και Δημιουργία Angular Components

Ξεκινώντας το «χτίσιμο» της εφαρμογής *Angular* (*Angular Application*), θα πρέπει να δημιουργήσουμε μία σειρά νέων καταλόγων. Στην εικόνα 137 βλέπουμε την παρουσία τεσσάρων νέων καταλόγων, εντός του καταλόγου *app*. Αυτοί είναι, ο κατάλογος ***core***, ο κατάλογος ***features***, ο κατάλογος ***layout*** και ο κατάλογος ***shared***. Ο κατάλογος *core*, θα περιέχει αντικείμενα τύπου *Angular Services*, *Angular Guards* και *Interceptors*. Ο κατάλογος *shared* θα είναι υπεύθυνος για όλα τα κοινόχρηστα στοιχεία της εφαρμογής *Angular*, καθώς και τις *Angular directives* αλλά και τα *Angular Models*. Ο κατάλογος *features* θα περιέχει χαρακτηριστικά όπως, το *shopping cart feature (καλάθι αγορών)* της εφαρμογής μας, το *shop feature* (κατάστημα αγορών) και το *checkout feature* (παραγγελίες και πληρωμές). Ο κατάλογος *layout* θα περιέχει τα αντικείμενα τα οποία ορίζουν τον *header* της εφαρμογής μας.



Εικόνα 137

Στον κατάλογο *layout* θα δημιουργήσουμε το πρώτο *Angular component*, το οποίο θα το ονομάσουμε *header component*. Εντός του περιβάλλοντος *Visual Studio Code*, ανοίγουμε το τερματικό και εισάγουμε την εντολή *ng g c layout/header --skip-tests.* Στην εικόνα 138 βλέπουμε τα αρχεία που δημιουργήθηκαν εντός του καταλόγου *header*.

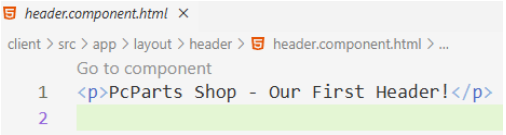


Εικόνα 138

Το περιεχόμενο ενός τυπικού (*auto generated*) Angular Component παρουσιάζεται στην εικόνα 139. Ένα *Angular Component* με τίτλο *header*, το οποίο περιέχει ένα *component decorator (@)*, έναν *selector* με το πρόθεμα *app-* (με αυτόν τον τρόπο θα το εισάγουμε και σε άλλα *components*), καθώς και *Url paths* για τα σχετιζόμενα σε αυτό αρχεία.

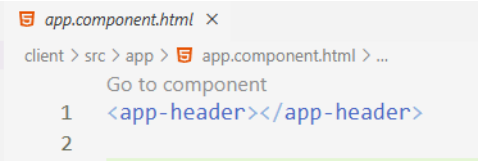
Εικόνα 139

Ανοίγοντας το αρχείο *header.component.html*, προσθέτουμε την *html* παράγραφο της εικόνας 140.



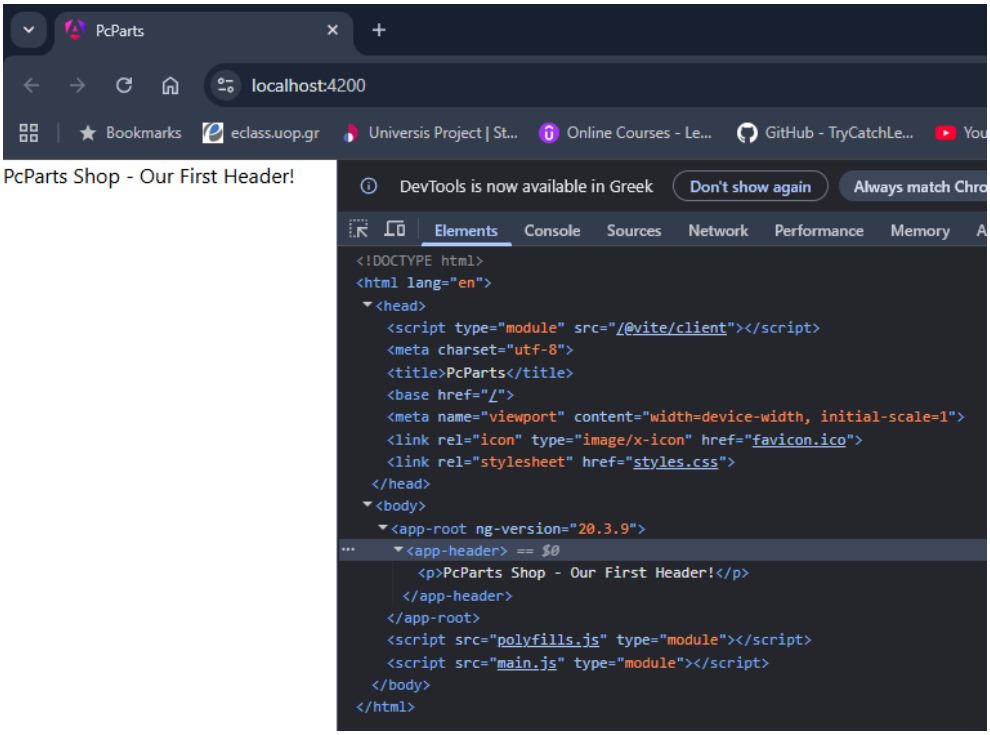
Εικόνα 140

Τέλος, μεταβαίνουμε στο αρχείο *app.component.html* Και κάνουμε κλήση του *header.component.ts* μέσω της έκφρασης που βλέπουμε στην εικόνα 141.



Εικόνα 141

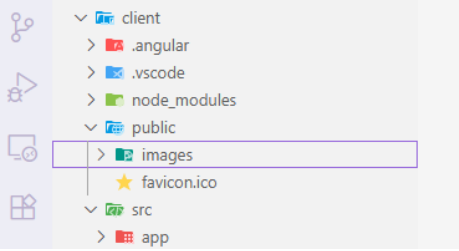
Ανοίγοντας το τερματικό κι εκτελώντας την εντολή *ng serve*, ανοίγει μία νέα καρτέλα στον φυλλομετρητή μας (*browser*), όπου εμφανίζεται το περιεχόμενο της εικόνας 142. Ουσιαστικά έχουμε το πρώτο αποτέλεσμα σε μορφή *HTML* το οποίο μας επιστρέφει η *Angular* εφαρμογή μας.



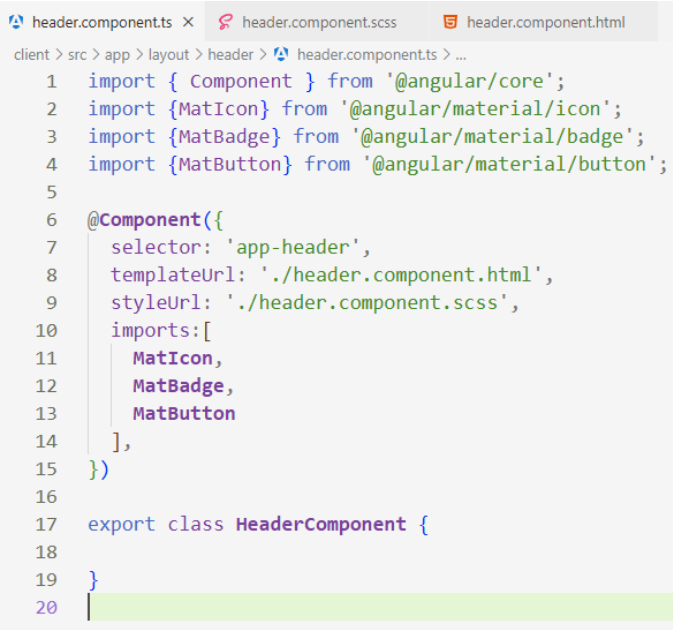
Εικόνα 142

5.2 Βελτιστοποίηση του Header Angular Component

Χρησιμοποιώντας τις βιβλιοθήκες *Tailwind*, οι οποίες συμπεριλαμβάνονται στον αντίστοιχο κατάλογο του *Angular project*, θα προσπαθήσουμε να βελτιώσουμε την εμφάνιση του πρώτου μας *component*. Χρησιμοποιώντας *Tailwind Utility Attributes* εντός του αρχείου *header.component.html*, δημιουργούμε ένα *border* για τον *header* μας, με *padding* που ισούται με τρία *pixels* και καταλαμβάνει όλο το πλάτος της οθόνης. Στη συνέχεια, κάνουμε χρήση της κλάσης *flex,* η οποία μας δίνει τη δυνατότητα εισαγωγής και διαμόρφωσης των πρώτων συνδέσμων της αρχικής μας σελίδας, *Home*, *Shop* και *Contact*, στο κέντρο της *navigation bar*. Με την ίδια λογική γίνεται και η εισαγωγή του *Tailwind* εικονιδίου *shoping_cart*, αλλά και τον κουμπιών *Login* και *Register*, σε στοίχιση δεξιά στην *navigation bar*. Στην εικόνα 143 παρουσιάζεται ο κατάλογος *Images*, όπου εκεί θα φιλοξενηθεί όλο το στατικό περιεχόμενο της εφαρμογής μας, όπως τα λογότυπα και οι εικόνες των προϊόντων του *e-shop*. Επίσης, στην εικόνα 144 βλέπουμε την εισαγωγή (*import*) των απαραίτητων *Angular-Material* βιβλιοθηκών εντός του *header.component.ts* αρχείου, οι οποίες είναι υπεύθυνες για την εμφάνιση των *Angular* κουμπιών στην ιστοσελίδα μας. Στην εικόνα 145, εντός του αρχείου *header.component.css* παρουσιάζεται ο κώδικας παραμετροποίησης των *mat-badge* κουμπιών του αρχείου *header.component.html*, αλλά και του εικονιδίου *shopping_cart*, το οποίο είναι τύπου *mat-icon*. Ουσιαστικά με του ορισμούς αυτούς ρυθμίζουμε την «στρογγυλάδα» των κουμπιών *Login* και *Register*, αλλά και το μέγεθος του εικονιδίου *shopping_cart*. Στην εικόνα 146, παρουσιάζεται ο κώδικας *HTML* του *header.component.html*.



Εικόνα 143



Εικόνα 144

Εικόνα 145

header.component.ts | header.component.scss | header.component.html

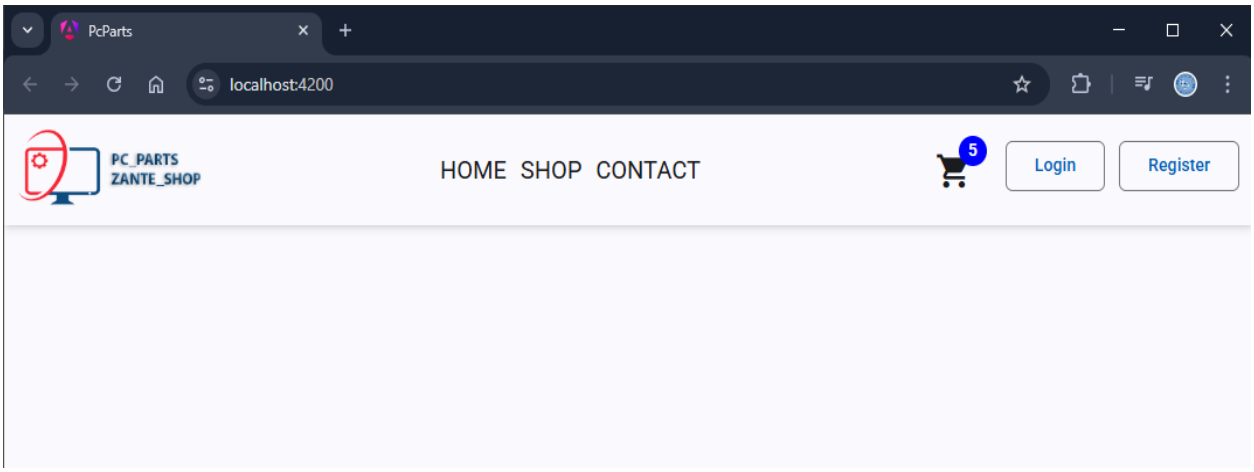
client > src > app > layout > header > header.component.html > ...

Go to component

```
<header class="shadow-md p-3 w-full">
    <div class="flex align-middle items-center justify-between max-w-screen-2xl mx-auto">
        <img src="/images/logo.png" alt="pcparts logo" class="max-h-16">
        <nav class="flex gap-3 my-2 uppercase text-xl">
            <a>Home</a>
            <a>Shop</a>
            <a>Contact</a>
        </nav>
        <div class="flex gap-3 align-middle">
            <a matBadge="5" matBadgeSize="large" class="custom-badge mt-2 mr-4">
                <mat-icon>shopping_cart</mat-icon>
            </a>
            <button mat-stroked-button>Login</button>
            <button mat-stroked-button>Register</button>
        </div>
    </div>
</header>
```

Εικόνα 146

Κλείνοντας την τρέχουσα ενότητα, εκτελούμε το *Angular project* μέσω του τερματικού κι έχουμε ως αποτέλεσμα την εμφάνιση των πρώτων κουμπιών και κατ' επέκταση της *navigation bar* στον φυλλομετρητή μας (εικόνα 147).



Εικόνα 147

Πρακτικά, η εμφάνιση των κουμπιών και η διάταξη την οποία καταφέραμε, δεν μας προσφέρει τίποτε παραπάνω από την παρουσία ενός απλού *header* στην αρχική σελίδα της εφαρμογής μας. Το λειτουργικό κομμάτι θα το εξετάσουμε στις επόμενες ενότητες.

5.3 Αιτήματα HTTP Μέσω της Εφαρμογής Angular

Προκειμένου να εκτελεστεί ένα *HTTP* αίτημα μέσω του *Angular project*, θα πρέπει να δημιουργήσουμε έναν *Angular Http client*. Εντός του αρχείου *app.config.ts* θα προσθέσουμε έναν νέο *provider* (*Angular*

μέθοδο) με τίτλο *provideHttpClient*. Η μέθοδος αυτή μπορεί να χρησιμοποιηθεί σε οποιοδήποτε *Angular component* μέσω του *dependency injection* (εικόνα 148Α).

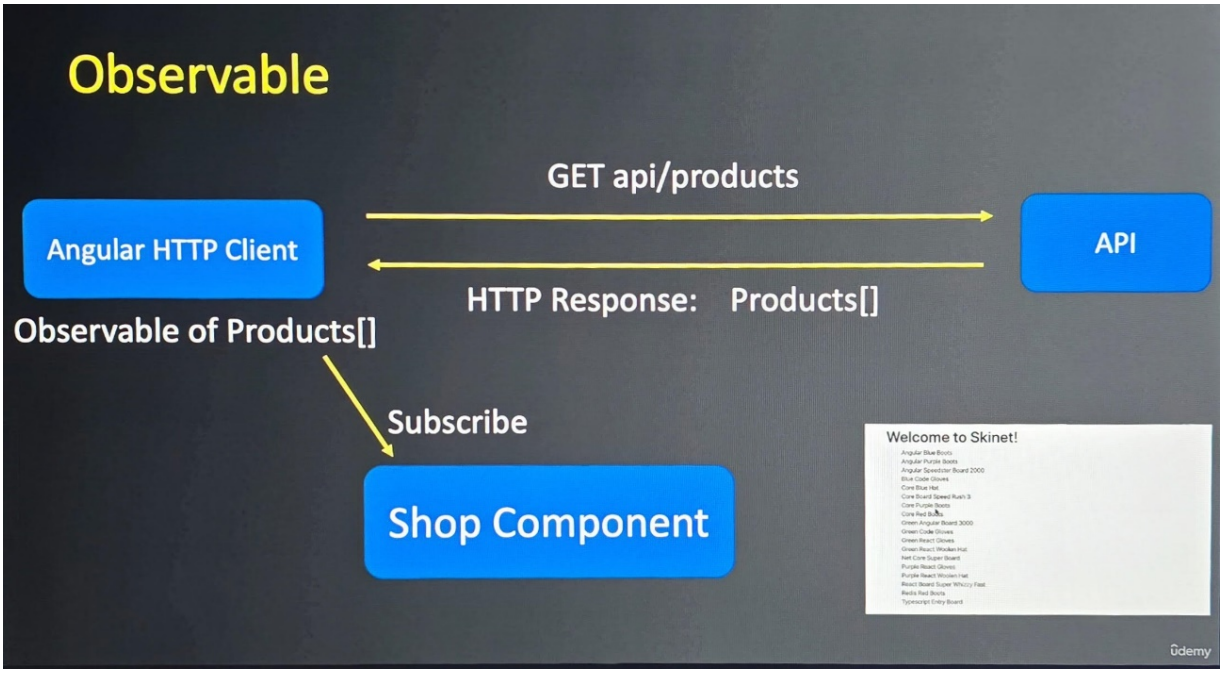
app.config.ts

client > src > app > app.config.ts > ...

```
1  import { ApplicationConfig, provideBrowserGlobalErrorListeners,
2  import { provideRouter } from '@angular/router';
3  import { routes } from './app.routes';
4  import { provideHttpClient } from '@angular/common/http';
5
6  export const appConfig: ApplicationConfig = {
7    providers: [
8      provideBrowserGlobalErrorListeners(),
9      provideZoneChangeDetection({ eventCoalescing: true }),
10     provideRouter(routes),
11     provideHttpClient()
12   ]
13 };
14
```

Εικόνα 148Α

Στη συνέχεια, θα ανοίξουμε το αρχείο *app.component.ts* κι εντός της κλάσης *AppComponent* θα προσθέσουμε την *property baseUrl*, η οποία απευθύνεται στον *API Server*. Ακολούθως, θα πρέπει να κάνουμε *inject* τον *Http client* κι αυτό θα γίνει μέσω της αντίστοιχης μεθόδου *inject*, η οποία ορίζεται εντός των βιβλιοθηκών (*components*) της *Angular* (*@angular/core*). Για την εκτέλεση του αιτήματος *HTTP* προς τον *API server*, θα πρέπει να χρησιμοποιήσουμε την τεχνική του *initialization lifecycle event*. Αυτό σημαίνει πως κάθε *Angular component* έχει έναν κύκλο ζωής συμβάντων (*lifecycle of events*) κι όταν ο *constructor* του αρχικοποιείται, τότε θεωρείται η καταλληλότερη χρονική στιγμή για την εκτέλεση ενός αιτήματος *HTTP*. Για τον λόγο αυτό, θα κάνουμε *implement* τη μέθοδο *OnInit* εντός του *constructor* της κλάσης *AppComponent*. Εν συνεχεία, θα γίνει η χρήση της μεθόδου *OnInit* ως εξής: Χρησιμοποιώντας την property *HTTP* μέσω της μεθόδου *get*, εισάγουμε την *property baseUrl* προκειμένου προκειμένου να αιτηθούμε μία λίστα αντικειμένων (*objects*) από τον *ProductsController* του *API project*. Η μέθοδος *get* μας επιστρέφει ένα ***observable object*** (εικόνα 148Β), το οποίο για να παρακολουθήσουμε και να αντλήσουμε πληροφορία από αυτό, θα πρέπει να εκμεταλλευτούμε τη μέθοδο *subscribe*. Μέσω της *subscribe* μπορούμε να εξάγουμε την επιθυμητή πληροφορία από τον *API server*. Τέλος, με την *call back function log* θα ελέγξουμε την ακεραιότητα της πληροφορίας που εξάγουμε, την πιθανή ύπαρξη σφαλμάτων και εν κατακλείδι την προβολή ενός μηνύματος για την ολοκλήρωση του αιτήματός μας. Ο κώδικας που ορίζει όλα τα παραπάνω εμφανίζεται στην εικόνα 148Γ, καθώς και το αποτέλεσμα που μας δίνει εντός της κονσόλας του φυλλομετρητή μας, στην εικόνα 149.



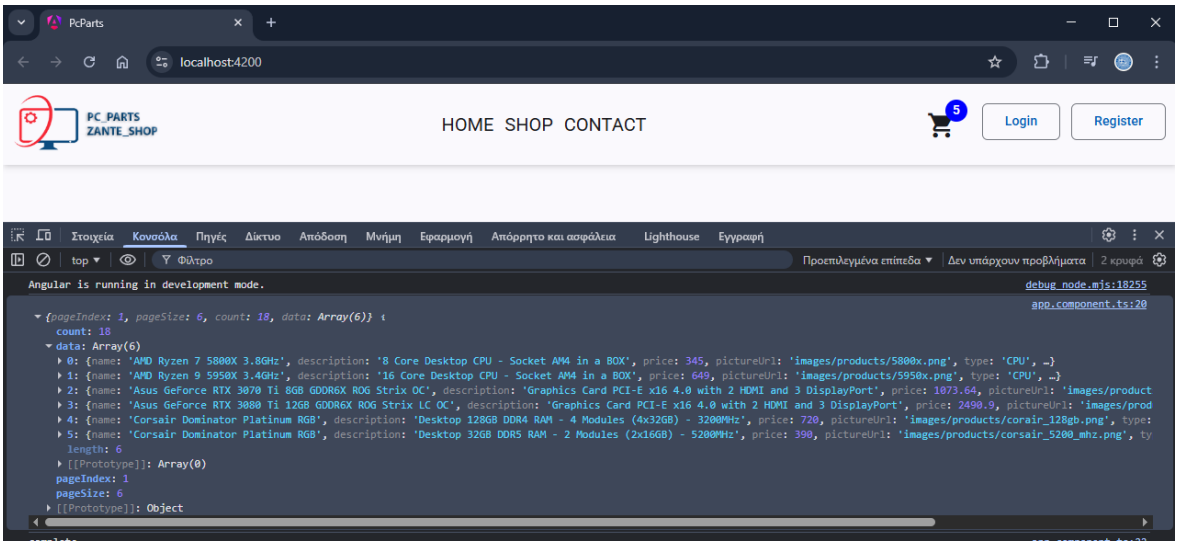
Εικόνα 148Β

app.component.ts | Settings | app.component.html

client > src > app > app.component.ts > ...

```
1  import { Component, inject, OnInit} from '@angular/core';
2  import { RouterOutlet } from '@angular/router';
3  import { HeaderComponent } from "./layout/header/header.component";
4  import { HttpClient } from '@angular/common/http';
5
6  @Component({
7    selector: 'app-root',
8    standalone: true,
9    imports: [RouterOutlet,HeaderComponent],
10   templateUrl: './app.component.html',
11   styleUrl: './app.component.scss'
12 })
13 export class AppComponent implements OnInit{ //constructor initialization
14   baseUrl = 'https://localhost:5001/api/' //http request to API server property
15   private http = inject(HttpClient); //injection of httpclient
16   title = 'PcParts';
17
18   ngOnInit(): void{//lifecycle event init
19     this.http.get(this.baseUrl + 'products').subscribe({//callback func to ask items fro productscontroller
20     next: data => console.log(data),//get data
21     error: error => console.log(error),//test for errors
22     complete: () => console.log('complete')//end of process message into browser console
23     })
24   }
25 }
```

Εικόνα 148Γ



Εικόνα 149

Στην προηγούμενη εικόνα, παρατηρούμε πως στην κονσόλα του φυλλομετρητή μας εξάγεται όλη η διαθέσιμη πληροφορία από τον *API server* και τη βάση δεδομένων της εφαρμογής μας, η λίστα όλων των προϊόντων συμπεριλαμβανομένων των χαρακτηριστικών τους, καθώς και πληροφορίες σχετικά με τη σελιδοποίηση την οποία δημιουργήσαμε στο προηγούμενο κεφάλαιο.

Η πληροφορία της εικόνας 149, θα πρέπει να προβληθεί στην αρχική σελίδα της εφαρμογής μας και κατ' επέκταση στον browser μας. Ανοίγοντας εκ νέου το αρχείο *app.component.ts*, θα ορίσουμε μία νέα *class property* με τίτλο *products*, τύπου *any* και θα την αρχικοποιήσουμε με τη μορφή ενός άδειου πίνακα. Ύστερα, την πληροφορία που μας δίνει ο *ProductsController*, θα την αποθηκεύσουμε στη συγκεκριμένη μεταβλητή. Λόγω του ότι η πληροφορία που μας επιστρέφεται είναι ένα *observable object*, θα πρέπει να ορίσουμε τη μέθοδο *get* ως τύπου *any* (εικόνα 150).

```
17    products: any[] = [];
18
19    ngOnInit(): void{//lifecycle event init
20      this.http.get<any>(this.baseUrl + 'products').subscribe({//callback func
21      next: response => this.products=response.data,//store data to array prod
22      error: error => console.log(error),//test for errors
23      complete: () => console.log('complete')//end of process message into bro
24      })
```

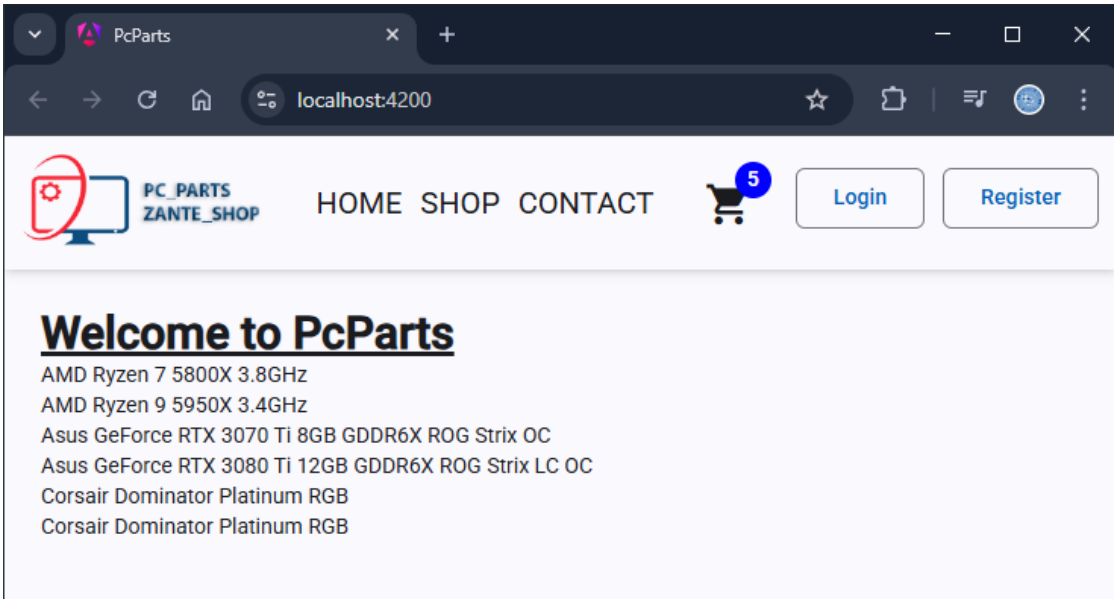
Εικόνα 150

Η επόμενή μας κίνηση είναι να μεταφερθούμε στο αρχείο *app.component.html*, όπου θα ορίσουμε μία *unordered list* και μέσω ενός *for loop* θα προσπελάσουμε όλα τα αντικείμενα που εμπεριέχονται στον πίνακα *products*, τον οποίο ορίσαμε εντός του *app.component.ts*. Ο κώδικας του *app.component.html* παρουσιάζεται στην εικόνα 151.

```
4   <div class="container mt-6">
5       <h1 class="text-3xl font-bold underline">Welcome to {{title}}</h1>
6       <ul>
7           @for (product of products; track product.id) {
8               <li>{{product.name}}</li>
9           }
10      </ul>
11  </div>
```

Εικόνα 151

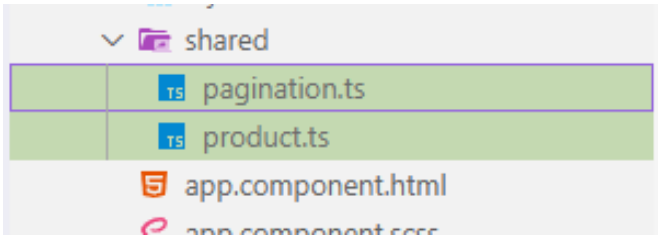
Η πληροφορία που εμφανίζεται σε μορφή λίστας στον *browser*, δεν είναι άλλη από το όνομα του εκάστοτε προϊόντος (εικόνα 152). Λόγω των ρυθμίσεων της σελιδοποίησης, το *default* μέγεθος των σελίδων είναι ίσο με έξι, γι' αυτό και τα προϊόντα που παρουσιάζονται στη σελίδα ένα (*page_index* =1) είναι τα πρώτα έξι, σε αλφαβητική σειρά.



Εικόνα 152

5.4 TypeScript

Η TypeScript (TS) είναι μια γλώσσα προγραμματισμού υψηλού επιπέδου, η οποία προσθέτει στατική πληκτρολόγηση με *optional type annotations* στην *JavaScript*. Έχει σχεδιαστεί για την ανάπτυξη μεγάλων εφαρμογών, ενώ κατά τη διαδικασία μεταγλώττισής της δύναται να μετατραπεί σε *JavaScript*. Η *Angular* βασίζεται στην *TypeScript*, γι' αυτό και στη συνέχεια θα εκμεταλλευτούμε πολλές από τις δυνατότητές της. Αρχικά, θα ορίσουμε δύο νέα *Types* (*TypeScript Interfaces*) εντός του καταλόγου *shared* (εικόνα 153). Το πρώτο θα έχει όνομα *product.ts* κι εντός του θα ορίσουμε όλες τις *properties* που απαρτίζουν ένα αντικείμενο τύπου *product* (εικόνα 154).



Εικόνα 153

product.ts

client > src > app > shared > models > product.ts > ...

```ts
export type Product ={
    id: number;
    name: string;
    description: string;
    price: number;
    pictureUrl: string;
    type: string;
    brand: string;
    quantityInStock: number;
}
```

Εικόνα 154

Το δεύτερο *type* θα ονομάζεται *pagination.ts* κι εντός του θα ορίσουμε όλες τις *properties* που αφορούν ένα αντικείμενο σελιδοποίησης (εικόνα 154). Η μόνη διαφορά σε σχέση με το προηγούμενο *type*, είναι πως στην περίπτωση της σελιδοποίησης ορίζεται ως τύπου <*T*>. Αυτό συμβαίνει διότι η πληροφορία που μας επιστρέφει ο *API server* μέσω του *ProductsController* σε μορφή *Json* λίστας, θα πρέπει να αποθηκεύεται εντός της *data property* και κατ' επέκταση στα κελιά του πίνακά της (εικόνα 155).

pagination.ts

client > src > app > shared > pagination.ts > ...

```ts
export type Pagination<T> ={
    pageIndex: number;
    pageSize: number;
    count: number;
    data: T[];
}
```

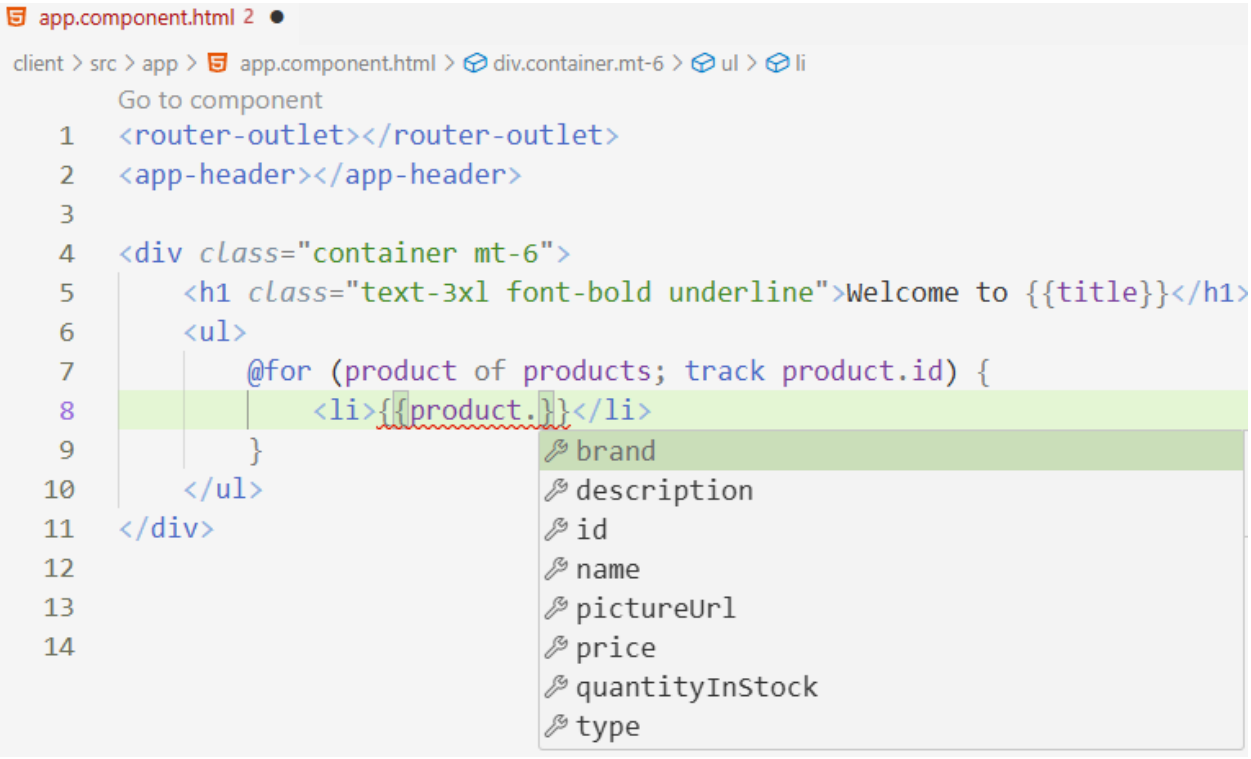
Εικόνα 155

Στη συνέχεια, θα ανοίξουμε το αρχείο *app.component.ts* στο οποίο θα ορίσουμε τα δύο νέα *Types* (*TypeScript Interfaces*). Αρχικά στη μεταβλητή *products* δίνουμε τον τύπο *Product* (κενού πίνακα αντικειμένων τύπου *Product*). Ακολούθως, στη μέθοδο *get* ορίζουμε την επιστρεφόμενη μορφή δεδομένων ως *Pagination*, η οποία με τη σειρά της είναι τύπου *Product*. Τέλος, στη μεταβλητή *products* αποθηκεύεται η πληροφορία που μας επιστρέφει ο *API server*, μέσω της οντότητας *response.data*, την οποία ορίσαμε προηγουμένως (εικόνα 156).

```ts
  products: Product[] = [];

  ngOnInit(): void{//lifecycle event init using types
    this.http.get<Pagination<Product>>(this.baseUrl + 'products').subscribe({//
    next: response => this.products=response.data,//store data to array product
    error: error => console.log(error),//test for errors
    complete: () => console.log('complete')//end of process message into brows
    })
```

Εικόνα 156Α

Τέλος, εντός του *app.component.html* αρχείου, έχουμε τις δυνατότητες του *Intellisense* και του *Type Safety*. Πιο απλά, έχουμε πρόσβαση σε όλα τα πεδία του *Product type* χρησιμοποιώντας τα κατάλληλα *Html attributes*, καθώς και τη δυνατότητα της αυτόματης συμπλήρωσης κι ελέγχου του κώδικα μας κατά τη συγγραφή (εικόνα 156Β).



Εικόνα 156Β

Κεφάλαιο 6. Angular Project – Διεπαφή Χρήστη

Στο τρέχον κεφάλαιο, θα ασχοληθούμε εκτενέστερα με τις *Angular Services* και με την εν γένει δημιουργία της διεπαφής χρήστη (*User Interface*). Μέσω των *Angular Services*, θα επιτύχουμε την καλύτερη λειτουργία των HTTP αιτημάτων καθ' όλη τη διάρκεια του κύκλου ζωής (*application lifecycle*) της εφαρμογής μας. Επιπροσθέτως, συνδυάζοντας τα *Angular Material Components* και την *Tailwind CSS*, θα επιτύχουμε το επιθυμητό αποτέλεσμα αναφορικά με την εμφάνιση της ιστοσελίδας μας.

6.1 Angular Services

Προς το παρόν, εντός του *TypeScript* αρχείου *app.component.ts* εκτελούμε το πρώτο μας HTTP αίτημα προς τον *API server*, εκμεταλλευόμενοι τη μέθοδο *ngOnInit*. Ωστόσο, ο ενδεδειγμένος τρόπος για την εκτέλεση πολλαπλών *HTTP* αιτημάτων, από τα οποία πολλά μπορεί να έχουν ακριβώς τον ίδιο κώδικα, δεν είναι ο συγκεκριμένος. Στις *Angular* εφαρμογές χρησιμοποιούνται ευρέως οι *Angular services* για να πραγματοποιούμε *HTTP* αιτήματα, καθώς μας δίνουν τη δυνατότητα του διαμοιρασμού δεδομένων και λειτουργικότητας σε όλο το φάσμα μίας *Angular* εφαρμογής. Επιπλέον, οι *Angular services* έχουν τη μορφή *singleton*. Αυτό σημαίνει πως εγγράφονται (*registered*) και δημιουργούνται κατά τη διάρκεια εκκίνησης της εφαρμογής και διατηρούν τη λειτουργικότητά τους έως τον τερματισμό της. Για τους λόγους που αναλύσαμε, θα ξεκινήσουμε με τη δημιουργία μίας *Angular service* με τίτλο *shop*. Εκτελώντας την εντολή *ng g s core/services/shop --skip-tests* , εντός του καταλόγου *core* δημιουργείται ο κατάλογος *services* κι εντός του, η *Angular service* με τίτλο *shop*. Ο κώδικας της εικόνας 157 αποτυπώνει τη δομή μίας *Angular service*. Το *@Injectable attribute* σημαίνει πως μία *Angular service* μπορεί να χρησιμοποιηθεί από οποιοδήποτε *component* στην εφαρμογή μας και το *provideIn: root*, δηλώνει πως μία τέτοια *service* εγγράφεται (*registered*) κατά την εκκίνηση της εφαρμογής μας.

shop.service.ts ●

client > src > app > core > services > shop.service.ts > ...

```
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4    providedIn: 'root',
5  })
6
7  export class ShopService {
8
9  }
```

Εικόνα 157

Στη συνέχεια, θα χρησιμοποιήσουμε τις μεταβλητές *baseUrl* και *http* του αρχείου *app.component.ts*, καθώς και τη λειτουργικότητα της μεθόδου *get* και θα τα ορίσουμε εντός της *TypeScript* κλάσης *ShopService*. Τη λειτουργικότητα της μεθόδου *get*, τη μεταφέρουμε εντός μίας νέας μεθόδου με τίτλο *GetProducts*, η οποία μας επιστρέφει την *paginated* πληροφορία από τον *API server*, σε οποιοδήποτε *Angular component* κάνει *inject* τη συγκεκριμένη *service* (εικόνα 158).

```
@Injectable({
  providedIn: 'root',
})

export class ShopService {
  baseUrl = 'https://localhost:5001/api/' //http request to API server pr
  private http = inject(HttpClient); //injection of httpclient

  getProducts(){
    return  this.http.get<Pagination<Product>>(this.baseUrl + 'products')
  }
}
```

Εικόνα 158

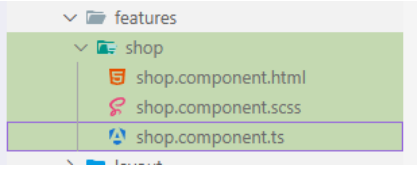
Τέλος, θα μεταβούμε στο αρχείο *app.component.ts* και με τις αλλαγές που παρουσιάζονται στην εικόνα 159, ουσιαστικά κάνουμε *inject* την πρώτη μας *Angular service* εντός του συγκεκριμένου *component*. Η λειτουργικότητα της εφαρμογής μας διατηρείται ακριβώς όπως και στην εικόνα 152, με τη διαφορά πως το *HTTP* αίτημα πραγματοποιείται από μία *Angular service* κι όχι από ένα *Angular component*.

```
export class AppComponent implements OnInit{ //constr
  private shopService =inject(ShopService); //injecti
  title = 'PcParts';
  products: Product[] = [];

  ngOnInit(): void{//lifecycle event init using types
    this.shopService.getProducts().subscribe({ //usin
      next: response => this.products=response.data,//
      error: error => console.log(error),//test for e
    })
  }
}
```

Εικόνα 159

6.2 Shop Angular Component

Προχωρώντας με τον σχεδιασμό της διεπαφής χρήστη, θα δημιουργήσουμε ένα νέο *Angular component*, με τίτλο *shop*. Εκτελώντας την εντολή *ng g c features/shop –skip-tests* στο τερματικό του *Visual Studio Code*, δημιουργούμε το προαναφερθέν *component* εντός του καταλόγου *features* (εικόνα 160).

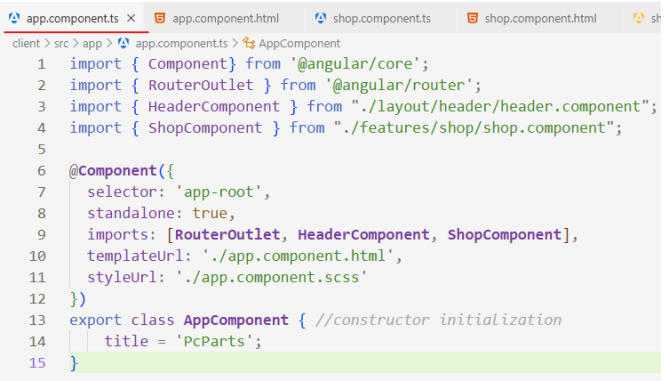


Εικόνα 160

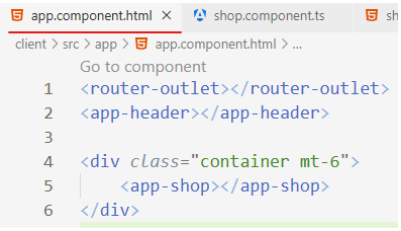
Στη συνέχεια, θα δημιουργήσουμε έναν νέο κατάλογο με τίτλο *products*, τον οποίο θα συμπεριλάβουμε εντός του καταλόγου *images* (εικόνα 161). Εδώ θα προσθέσουμε όλες τις φωτογραφίες των προϊόντων της βάσης δεδομένων *pcparts.db*.

Εικόνα 161

Ακολούθως, θα μεταβούμε στο *app.component.ts* αρχείο και θα αφαιρέσουμε εξ ολοκλήρου τη δυνατότητα άντλησης δεδομένων μέσω της *shop Angular service* (εικόνα 162). Αντ' αυτού θα κάνουμε *import* το νέο *component* που δημιουργήσαμε στο προηγούμενο βήμα. Επομένως, ο κώδικας του *app.component.html* αρχείου θα μεταβληθεί αναλόγως, σύμφωνα με την εικόνα 163.



Εικόνα 162



Εικόνα 163

Εντός του νέου *shop component* και πιο συγκεκριμένα στο αρχείο *TypeScript*, θα ορίσουμε εκ νέου τη λειτουργικότητα άντλησης δεδομένων μέσω της *shop service*. Ο κώδικας του *shop.component.ts* παρουσιάζεται στην εικόνα 164.

shop.component.ts | shop.component.html | shop.service.ts

client > src > app > features > shop > shop.component.ts > ...

```ts
import { Component, inject, OnInit } from '@angular/core';
import { ShopService } from '../../core/services/shop.service';
import { Product } from '../../shared/product';
import { MatCard } from '@angular/material/card';

@Component({
  selector: 'app-shop',
  imports: [
    MatCard
  ],
  templateUrl: './shop.component.html',
  styleUrl: './shop.component.scss',
})
export class ShopComponent implements OnInit {
  private shopService =inject(ShopService); //injection of angu
  title = 'PcParts'
  products: Product[] = [];

  ngOnInit(): void{//lifecycle event init using types
    this.shopService.getProducts().subscribe({ //using angular s
      next: response => this.products=response.data,//store data
      error: error => console.log(error),//test for errors
    })
  }
}
```

Εικόνα 164

Εκμεταλλευόμενοι τα *Tailwind attributes*, θα δοκιμάσουμε να παρουσιάσουμε τις εικόνες των προϊόντων μας σε 5 στήλες. Επίσης θα τους δώσουμε μία εμφάνιση τύπου «κάρτας». Ορίζοντας ένα *for loop* εντός του αρχείου *shop.component.html*, προσπελαύνουμε όλη την πληροφορία που μας έχει αποδώσει το *shop.component.ts*, βασιζόμενοι στο *id* κάθε προϊόντος. Με χρήση ενός *img tag* παρατίθενται όλες οι εικόνες των διαθέσιμων προϊόντων της βάσης δεδομένων *pcparts.db*, μέσω της *property pictureUrl*. Ο κώδικας του *shop.component.html* φαίνεται στην εικόνα 165.

shop.component.html | shop.service.ts

client > src > app > features > shop > shop.component.html > div.grid.grid-cols-5.gap-5 > mat-card > img

Go to component

```html
<div class="grid grid-cols-5 gap-5">
    @for (product of products; track product.id) {
        <mat-card appearance="raised">
            <img src="{{product.pictureUrl}}" alt="{{product.name}}">
        </mat-card>
    }
</div>
```

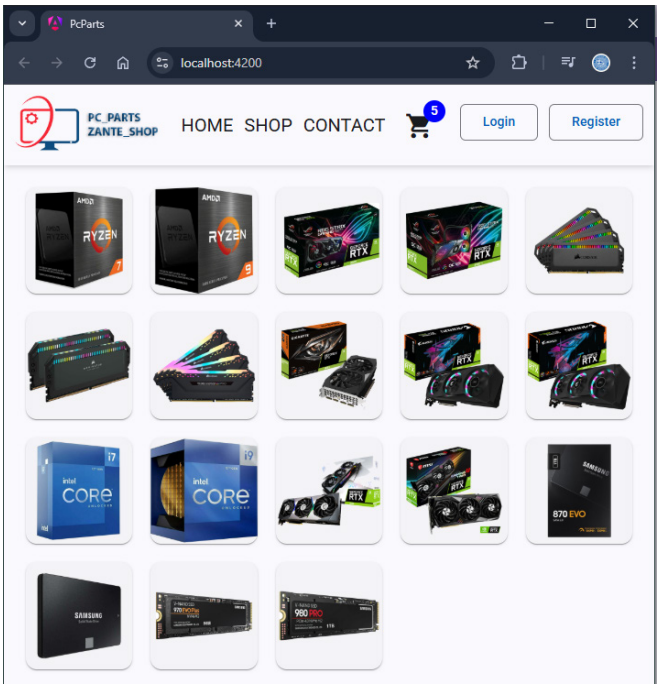
Εικόνα 165

Τέλος, θέλοντας να παρουσιάσουμε το σύνολο των προϊόντων που έχουν εισαχθεί στη βάση δεδομένων της εφαρμογής μας, θα μεταβούμε στην *shop Angular service* και θα διανθίσουμε το *query attribute* σύμφωνα με την εικόνα 166Α. Ουσιαστικά παρακάμπτουμε το πεδίο *_pageSize* που είχαμε ορίσει εντός της κλάσης *ProductSpecParams* και ορίζουμε το μέγεθος της σελίδας σε δεκαοκτώ αντικείμενα αντί για έξι.

```ts
getProducts(){
  return  this.http.get<Pagination<Product>>(this.baseUrl + 'products?pageSize=18')
}
```

Εικόνα 166Α

Εκτελώντας την εφαρμογή, θα λάβουμε από τον φυλλομετρητή μας το αποτέλεσμα της εικόνας 166Β.



Εικόνα 166Β

6.3 Product-item Angular Component

Σε αυτήν την ενότητα θα δημιουργήσουμε ένα *child-component*, ως προς το *shop component* το οποίο δημιουργήσαμε στην προηγούμενη ενότητα. Εκτελώντας την εντολή *ng g c features/shop/product-item --skip-tests* στο τερματικό μας, δημιουργούμε ένα νέο *Angular component* με τίτλο *product-item*, εντός του καταλόγου *shop*. Το συγκεκριμένο *component* αφορά αποκλειστικά και μόνο τα προϊόντα και την παρουσίασή τους στο περιβάλλον του *browser* μας. Ξεκινώντας, θα μεταβούμε στο αρχείο *TypeScript* με τίτλο *product-item.component.ts*, όπου θα κάνουμε *import* την *Angular* βιβλιοθήκη *MatCard* κι εντός του *export constructor* του, θα ορίσουμε μία *Input property* τύπου *Product* με τίτλο *product*. Με αυτήν τη μέθοδο αποκτάμε πρόσβαση στο *parent Angular component*, το οποίο δεν είναι άλλο από το *shop component* της προηγούμενης ενότητας (εικόνα 167).

product-item.component.ts × product-item.component.html shop.component.ts

client > src > app > features > shop > product-item > product-item.component.ts > ...

```ts
import { Component, Input } from '@angular/core';
import { MatCard } from '@angular/material/card';
import { Product } from '../../../shared/product';


@Component({
  selector: 'app-product-item',
  imports: [
    MatCard, //import card appearance
],
  templateUrl: './product-item.component.html',
  styleUrl: './product-item.component.scss',
})

export class ProductItemComponent {
  @Input() product!: Product; //access to parent component
}
```

Εικόνα 167

Ύστερα, θα ανοίξουμε το *template shop.component.html* και θα μεταβάλουμε τον κώδικά του, ώστε το αντικείμενο *product* να λαμβάνεται από το *child template product-item.component.html*. Χρησιμοποιώντας το *Angular attribute <app-product-item [product] = "product">*, λαμβάνουμε την πληροφορία της οντότητας *Product* από τη μεταβλητή *product* του *product-item.component* (εικόνα 168).

product-item.component.ts product-item.component.html shop.component.html ×

nt > src > app > features > shop > shop.component.html > div.grid.grid-cols-5.gap-5 > app-product-item

Go to component

```ts
<div class="grid grid-cols-5 gap-5">
    @for (product of products; track product.id) {
        <app-product-item [product]="product"></app-product-item>
    }
</div>
```

Εικόνα 168

Έχοντας πλέον πρόσβαση στην οντότητα *Product*, μέσω του *parent Angular component*, κι εφόσον η τιμή που λαμβάνουμε δεν είναι ίση με την τιμή *null [@if(product)]*, έχουμε τη δυνατότητα να παραθέσουμε όλα τα προϊόντα που αιτείται ο *client* στον φυλλομετρητή μας. Επιπλέον, κάνοντας χρήση των κατάλληλων *button attributes*, των *CSS* βιβλιοθηκών της *Angular* και της *Tailwind*, προσθέτουμε ως πληροφορία σε κάθε κάρτα προϊόντος, την τιμή πώλησής του, κάνοντας χρήση του *currency pipe module* για προβολή των τιμών σε ευρώ, αλλά και την επιλογή "*Add to Cart*". Ο κώδικας του *product-item.component.html* template παρουσιάζεται στην εικόνα 169.

product-item.component.html

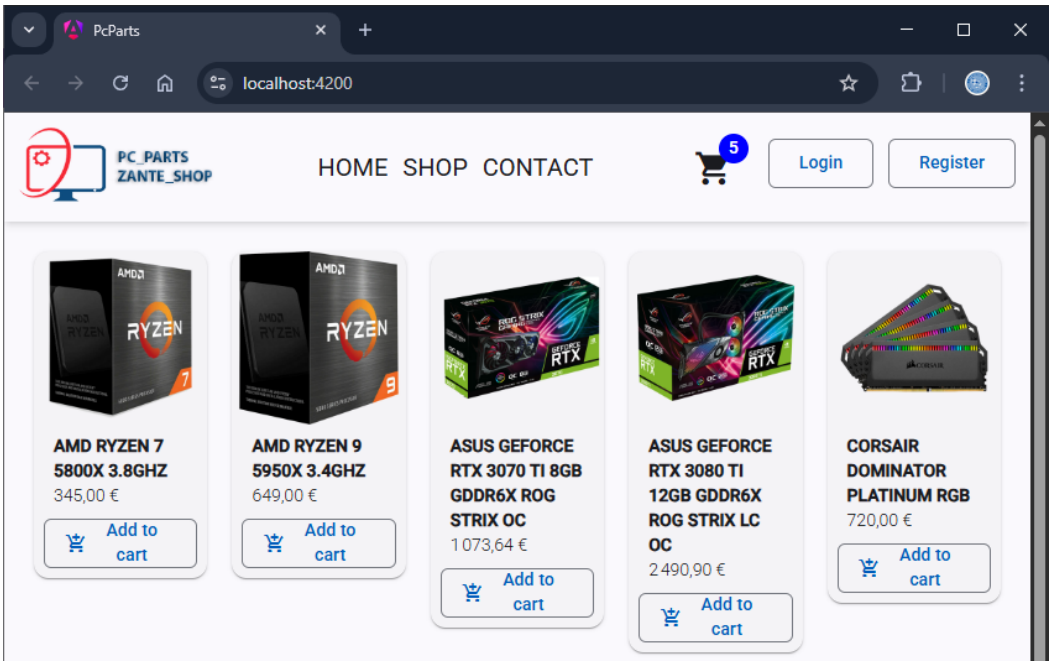
client > src > app > features > shop > product-item > product-item.component.html > ...

Go to component

```
@if(product){
    <mat-card appearance="raised">
        <img src="{{product.pictureUrl}}" alt="{{product.name}}">
        <mat-card-content class="mt-2">
            <h2 class="text-sm font-semibold uppercase">{{product.name}}</h2>
            <p class="font-light">{{product.price|currency:'EUR':'symbol':'1.2-2':'fr'}}</p>
        </mat-card-content>
        <mat-card-actions>
            <button mat-stroked-button class="w-full">
                <mat-icon>add_shopping_cart</mat-icon>
                Add to cart
            </button>
        </mat-card-actions>
    </mat-card>
}
```

Εικόνα 169

Εκτελώντας την εφαρμογή μας, έχουμε το αποτέλεσμα του *browser* στην εικόνας 170.



Εικόνα 170

6.4 Εμφάνιση Προϊόντων ανά Τύπο και Μάρκα

Σε αυτήν την ενότητα θα προσθέσουμε ακόμη μία δυνατότητα που ξεκινήσαμε να «χτίζουμε» στα προηγούμενα κεφάλαια, αυτήν του φιλτραρίσματος των προϊόντων ανά τύπο και μάρκα (*types & brands filtering*). Στη λογική της αποθήκευσης της συγκεκριμένης πληροφορίας άπαξ και σε μορφή πίνακα συμβολοσειρών, θα εκμεταλλευτούμε τη λειτουργικότητα της *shop Angular service*. Στην εικόνα 171 παρουσιάζουμε τον ορισμό δύο κενών τύπου *string* μεταβλητών σε μορφή πίνακα, με όνομα *types* και *brands* αντίστοιχα. Σε αυτές, μέσω των μεθόδων *getTypes* και *getBrands*, γίνεται χρήση των *observable objects*

και εκτελείται το κατάλληλο HTTP αίτημα προς τον API server. Κατά την εκκίνηση της εφαρμογής μας αντλείται η επιθυμητή πληροφορία, αποθηκεύεται στις αντίστοιχες μεταβλητές άπαξ και κατόπιν ελέγχων εκτελείται το συγκεκριμένο αίτημα.

```
15  types: string[]=[]; //set variables to save brands & types
16  brands: string[]=[]; //once the app starts
17
18  getProducts(){
19    return  this.http.get<Pagination<Product>>(this.baseUrl + 'products?pageSize=18')
20  }
21
22  getBrands(){ //using observable to get items brands and save them to variable brand
23    if(this.brands.length>0) return; //checks if already have the brands info
24    return this.http.get<string[]>(this.baseUrl + 'products/brands').subscribe({
25      next: response => this.brands = response
26    })
27  }
28
29  getTypes(){ ////using observable to get items types and save them to variable types
30    if(this.types.length>0) return; //checks if already have the types info
31    return this.http.get<string[]>(this.baseUrl + 'products/types').subscribe({
32      next: response => this.types = response
33    })
34  }
```

Εικόνα 171

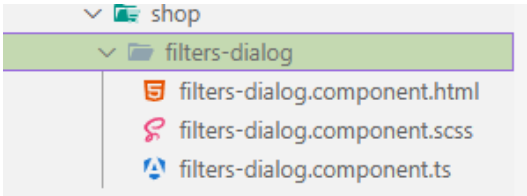
Προχωρώντας, θα ανοίξουμε το αρχείο *shop.component.ts* και θα προσθέσουμε τον απαραίτητο κώδικα, ώστε να μπορούμε να καλούμε τις δύο νέες μεθόδους (εικόνα 172). Ουσιαστικά έχουμε τη δημιουργία μίας νέας μεθόδου με τίτλο *initializeShop*, εντός της οποίας ορίζεται η μέχρι στιγμής λειτουργικότητα όλης της *Angular* εφαρμογής μας. Η συγκεκριμένη μέθοδος εκτελεί τις μεθόδους *getBrands*, *getProducts* και *getTypes* κατά την εκκίνηση της εφαρμογής μας, ενώ η ίδια με τη σειρά της καλείται από τη μέθοδο *ngOnInit*, την οποία είχαμε ορίσει στο προηγούμενο κεφάλαιο.

```
18  ngOnInit(): void{
19    this.initializeShop();
20  }
21
22  initializeShop() { //lifecycle event init using typ
23    this.shopService.getBrands();
24    this.shopService.getTypes();
25    this.shopService.getProducts().subscribe({ //usi
26      next: response => this.products=response.data,
27      error: error => console.log(error),//test for
28    })
29  }
```

Εικόνα 172

Αυτό στο οποίο θα εστιάσουμε τώρα, είναι ο τρόπος με τον οποίο θα εισάγουμε τη λειτουργικότητα του φιλτραρίσματος στη διεπαφή χρήστη. Στόχος μας είναι η δημιουργία ενός κουμπιού, το οποίο όταν επιλεγεί από τον χρήστη θα ανοίγει ένα παράθυρο επιλογών. Εκεί θα παρουσιάζονται ως «*tick-boxes*» οι τύποι και οι μάρκες όλων των προϊόντων της εφαρμογής μας και ο χρήστης θα μπορεί να επιλέξει όσα από αυτά επιθυμεί. Αναλόγως με τον συνδυασμό φίλτρων, θα επιλέγει ένα άλλο κουμπί ώστε να εφαρμόσει την αναζήτηση που ο ίδιος όρισε. Ξεκινώντας, θα ανοίξουμε ένα παράθυρο τερματικού και με την

εντολή *ng g c features/shop/filters-dialog --skip-tests* , θα δημιουργήσουμε ένα νέο *Angular component* εντός του καταλόγου *shop*. Το όνομα του νέου *component* θα είναι *filters-dialog* (εικόνα 173).



Εικόνα 173

Εντός του νέου *component* θα κάνουμε *inject* την *shop Angular service*, ώστε να έχουμε πρόσβαση στις λίστες των τύπων και των μαρκών των προϊόντων μας, εντός του *filters-dialog.component.html template* (εικόνα 174).

filters-dialog.component.ts ×

client > src > app > features > shop > filters-dialog > filters-dialog.component.ts > ...

```ts
import { Component } from '@angular/core';
import { inject } from '@angular/core/primitives/di';
import { ShopService } from '../../../core/services/shop.service';
import { MatDivider } from '@angular/material/divider';
import { MatListOption , MatSelectionList } from '@angular/material/list';
import { MatButton } from '@angular/material/button';

@Component({
  selector: 'app-filters-dialog',
  imports: [
    MatDivider,
    MatSelectionList,
    MatListOption,
    MatButton
],
  templateUrl: './filters-dialog.component.html',
  styleUrl: './filters-dialog.component.scss',
})
export class FiltersDialogComponent {
  shopService = inject(ShopService);
}
```

Εικόνα 174

Ακολούθως, θα μεταβούμε στο αντίστοιχο *html template*, *filters-dialog.component.html*. Εδώ θα ορίσουμε τον *html* κώδικα του παραθύρου-διαλόγου που θα εμφανίζεται στον χρήστη. Χρησιμοποιώντας τις βιβλιοθήκες *MatDivider*, *MatSelectionList*, *MatlistOption* και *MatButton*, διαμορφώνουμε ένα παράθυρο διαλόγου το οποίο εμφανίζεται στο μέσον της οθόνης του χρήστη. Σε δύο στήλες θα παρουσιάζονται σε μορφή «*check-boxes*» όλες οι διαθέσιμες μάρκες και οι τύποι των προϊόντων. Τη συγκεκριμένη πληροφορία την αντλούμε μέσω της *Angular shop service*, την οποία χρησιμοποιούμε εντός ενός *for-loop* στον κώδικα *html*. Τέλος, στο δεξί κάτω μέρος του εν λόγω παραθύρου θα εμφανίζεται το κουμπί «*Apply*

Filters», ώστε ο χρήστης να μπορεί να επιβεβαιώνει και να εκτελεί την αναζήτησή του. Ο κώδικας του *filters-dialog.component.html template* παρουσιάζεται στην εικόνα 175.

filters-dialog.component.html

client > src > app > features > shop > filters-dialog > filters-dialog.component.html > ...

```
<div>
    <h3 class="text-3xl text-center pt-6 mb-3">Filters</h3>
    <mat-divider></mat-divider>
    <div class="flex p-6">
        <div class="w-1/2">
            <h4 class="font-semibold text-xl text-primary">Brands</h4>
            <mat-selection-list>
                @for (brand of shopService.brands; track $index) {
                    <mat-list-option>{{brand}}</mat-list-option>
                }
            </mat-selection-list>
        </div>

        <div class="w-1/2">
            <h4 class="font-semibold text-xl text-primary">Types</h4>
            <mat-selection-list>
                @for (type of shopService.types; track $index) {
                    <mat-list-option>{{type}}</mat-list-option>
                }
            </mat-selection-list>
        </div>
    </div>
    <div class="flex justify-end p-4">
        <button mat-flat-button>Apply Filters</button>
    </div>
</div>
```

Εικόνα 175

Το συγκεκριμένο παράθυρο διαλόγου θα πρέπει να το διαχειριζόμαστε στο *shop.component.ts* αρχείο, ώστε να του αποστέλλουμε δεδομένα. Για να συμβεί κάτι τέτοιο, θα κάνουμε *inject* μία *Angular Modal Dialog service*, η οποία ονομάζεται *MatDialog* και θα την αναθέσουμε στη μεταβλητή *dialogService*. Στη συνέχεια θα ορίσουμε τη μέθοδο *OpenFiltersDialog*, η οποία θα χρησιμοποιεί την προαναφερθείσα *service* και θα ανοίγει το παράθυρο διαλόγου σε ελάχιστο πλάτος 500 *pixels* (εικόνα 176), χρησιμοποιούμενη από το *shop.component.html template*.

```
20  export class ShopComponent implements OnInit {
21    private shopService =inject(ShopService); //injection of angular s
22    private dialogService = inject(MatDialog); //modal dialog service
23    products: Product[] = [];
24
25
26    ngOnInit(): void{
27      this.initializeShop();
28    }
29
30    initializeShop() { //lifecycle event init using types
31      this.shopService.getBrands();
32      this.shopService.getTypes();
33      this.shopService.getProducts().subscribe({ //using angular servic
34        next: response => this.products=response.data,//store data to a
35        error: error => console.log(error),//test for errors
36      })
37    }
38
39    openFiltersDialog(){ //method which calls filtersdialogcomponent to
40      const dialogRef = this.dialogService.open(FiltersDialogComponent,
41        {minWidth: '500px'})
42    }
```

Εικόνα 176

Στο *shop.component.html template*, διαμορφώνουμε τον κώδικα (εικόνα 177) προκειμένου να προσθέσουμε ένα κουμπί (*mat-stroked-button*) με τίτλο *Filters*. Το συγκεκριμένο κουμπί θα εμφανίζεται στο επάνω – δεξί μέρος της οθόνης του *browser* κι όταν ο χρήστης το επιλέγει, εκκινεί η διαδικασία κλήσης της μεθόδου *openFiltersDialog*. Αυτό έχει σαν αποτέλεσμα την κλήση του *filters-dialog.component.html template*.

shop.component.html

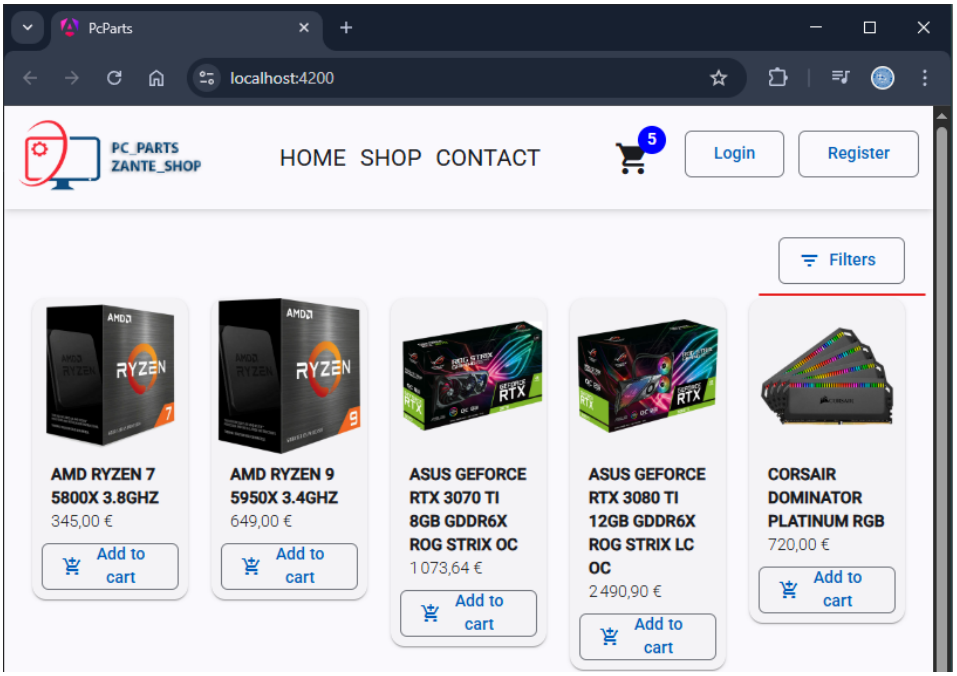
client > src > app > features > shop > shop.component.html > ...

Go to component

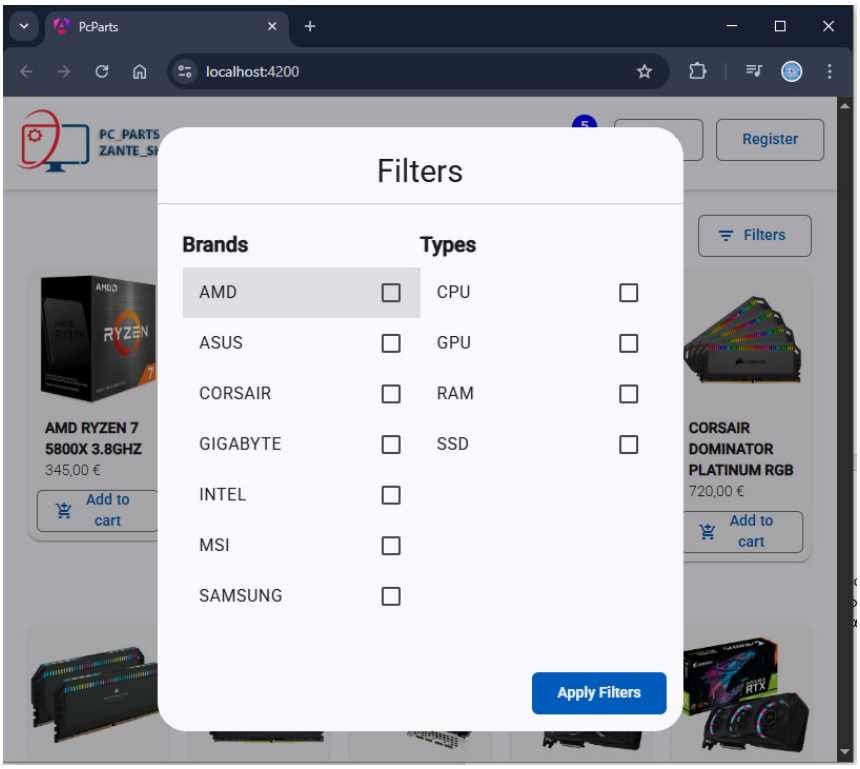
```
1   <div class="flex flex-col gap-3">
2       <div class="flex justify-end">
3           <button mat-stroked-button (click)="openFiltersDialog()">
4               <mat-icon>filter_list</mat-icon>
5               Filters
6           </button>
7       </div>
8       <div class="grid grid-cols-5 gap-5">
9           @for (product of products; track product.id) {
10              <app-product-item [product]="product"></app-product-item>
11          }
12      </div>
13  </div>
```

Εικόνα 177

Εκτελώντας την εφαρμογή μας, θα διαπιστώσουμε την ύπαρξη του νέου κουμπιού με τίτλο «*Filters*», στο επάνω – δεξί μέρος του παραθύρου του φυλλομετρητή μας (εικόνα 178). Κάνοντας «κλικ» στο συγκεκριμένο κουμπί, παίρνουμε το αποτέλεσμα της εικόνας 179.



Εικόνα 178



Εικόνα 179

Το παράθυρο διαλόγου με τίτλο «*Filters*» είναι πλέον διαθέσιμο προς τον χρήστη. Παρόλα αυτά εκκρεμεί η καθ' αυτή λειτουργικότητά του, διότι ακόμη κι αν επιλέξει ο χρήστης ορισμένα φίλτρα και πατήσει το κουμπί «*Apply Filters*», δεν θα συμβεί απολύτως τίποτα στο περιβάλλον του φυλλομετρητή, αλλά ούτε και στην *Angular* εφαρμογή μας. Συνεχίζοντας την ανάπτυξη της συγκεκριμένης λειτουργικότητας, θα ανοίξουμε το αρχείο *shop.component.ts* (εικόνα 180) και θα ορίσουμε τις μεταβλητές *selectedBrands* και *selectedTypes*. Οι δύο αυτές μεταβλητές θα μας χρησιμεύσουν στο να αποθηκεύουμε όλα τα φίλτρα που ορίζει ο χρήστης μέσω του *filters-dialog.component.html template*. Ακολούθως δημιουργούμε τη μέθοδο *openFiltersDialog*. Εκεί, μέσω της μεταβλητής *dialogRef*, περνάμε τις δύο μεταβλητές στα *filters-dialog components* της εφαρμογής μας. Τη στιγμή που μας επιστρέφεται η πληροφορία από τα συγκεκριμένα *components*, μεταβαίνουμε στη μέθοδο *afterclosed*, στην οποία κάνουμε χρήση ενός *observable* αντικειμένου. με αυτόν τον τρόπο μπορούμε να αποθηκεύουμε εντός του *shop component* τα φίλτρα που ορίζει ο χρήστης κάθε φορά που κλείνει το παράθυρο διαλόγου και στη συνέχεια μέσω της *shop Angular service* και της μεθόδου *getProducts*, να παραθέτουμε το τελικό αποτέλεσμα του φιλτραρίσματος στον φυλλομετρητή μας.

```
25    selectedBrands: string[] = []; //store selected items brands filter
26    selectedTypes: string[] = []; //store selected items types filter
27
28    ngOnInit(): void{
29      this.initializeShop();
30    }
31
32    initializeShop() { //lifecycle event init using types
33      this.shopService.getBrands();
34      this.shopService.getTypes();
35      this.shopService.getProducts().subscribe({ //using angular service
36        next: response => this.products=response.data,//store data to array products
37        error: error => console.log(error),//test for errors
38      })
39    }
40
41    openFiltersDialog(){ //method which calls filtersdialogcomponent to open
42      const dialogRef = this.dialogService.open(FiltersDialogComponent, {
43        minWidth: '500px',//set minimum filters dioalog window width
44        data:{//receive and send data from/to filter dialog component
45          selectedBrands: this.selectedBrands,
46          selectedTypes: this.selectedTypes
47        }
48      });
49      dialogRef.afterClosed().subscribe({ //using observer object to log data and afterClosed
50      // in order to observe the received data after apply filters button click
51        next: result => { //after filters dialog window is closed (apply filters button clicke
52          if(result){
53            this.selectedBrands = result.selectedBrands;
54            this.selectedTypes = result.selectedTypes;
55            //apply filters to product list by calling the shop Angular service
56            this.shopService.getProducts(this.selectedBrands, this.selectedTypes).subscribe({
57              next: response => this.products = response.data,
58              error: error => console.log(error)
59            })
```

Εικόνα 180

Στο filters-dialog.component.ts αρχείο, κάνουμε *inject* την κλάση *MatDialogRef* με τύπο *FiltersDialogComponent* και την αναθέτουμε στη μεταβλητή *dialogRef*. Με τη συγκεκριμένη ενέργεια επιτυγχάνουμε πρόσβαση στα δεδομένα χρήστη (φίλτρα) μέσω του filters-dialog.component.html template. Αφού μας επιστραφούν τα δεδομένα από το εν λόγο *template*, τα αναθέτουμε στη μεταβλητή *data*. Η πληροφορία στη μεταβλητή *data* αποθηκεύεται εκ νέου μέσω ενός *injection token*, το οποίο ονομάζεται *MAT_DIALOG_DATA*. Εφόσον έχουμε αποθηκευμένα τα φίλτρα μας στις μεταβλητές *selectedBrands* και *selectedTypes*, μέσω της μεθόδου *applyFilters* αποστέλλονται στο *shop.component.ts* αρχείο. Στην εικόνα 181 παρουσιάζεται ο κώδικας της *TypeScript* κλάσης *FiltersDialogComponent*.

```
22  export class FiltersDialogComponent {
23    shopService = inject(ShopService); //inject shop service to templ
24    //inject MatdialogRef to pass filters into dialog template
25    private dialogRef = inject(MatDialogRef<FiltersDialogComponent>);
26    //inject MAT_DIALOG_DATA in order to pass data to dialog templat
27    data =  inject(MAT_DIALOG_DATA);
28
29    //store data variables form filters-dialog template
30    selectedBrands: string[] = this.data.selectedBrands;
31    selectedTypes: string[] = this.data.selectedTypes;
32
33    //method to apply filters when closing the filters dialog window
34    //when pressing apply filters button from template
35    applyFilters(){
36      this.dialogRef.close({//closes filters dialoge and data is stor
37        selectedBrands: this.selectedBrands, //data is sent to shop.c
38        selectedTypes: this.selectedTypes //in order filters applyanc
39      })
40    }
41  }
```

Εικόνα 181

Χρησιμοποιώντας την τεχνική *Angular Two-Way Binding*, εντός των *Angular attributes* του *template filters-dialog.component.html*, έχουμε πρόσβαση στις οντότητες *selectedBrands* και *selectedTypes* του *filters-dialog.component.ts*. Τα *attributes [(ngModel)]* και *[Multiple]*, εκτελούν έναν ρόλο παρατηρητή (*tracker*) καθώς ο *client* ορίζει τα φίλτρα εντός του παραθύρου διαλόγου. Καθώς ο χρήστης επιλέγει τα επιθυμητά «*check-boxes*», η εν λόγω πληροφορία αποθηκεύεται στις μεταβλητές selectedB*rands* και selected*Types* αντίστοιχα. Τέλος, η συγκεκριμένη πληροφορία θα μεταβιβαστεί στο αρχείο *filters-dialog.component.ts*, καθώς ο χρήστης επιλέγει το κουμπί «*Apply Filters*» και αυτομάτως καλείται η μέθοδος *applyFilters*. Στην εικόνα 182 παρουσιάζεται ο κώδικας του *filters-dialog.component.html* αρχείου.

filters-dialog.component.html ● shop.service.ts

client > src > app > features > shop > filters-dialog > filters-dialog.component.html > ...

Go to component

```
<div>
  <h3 class="text-3xl text-center pt-6 mb-3">Filters</h3>
  <mat-divider></mat-divider>
  <div class="flex p-4">
    <div class="w-1/2">
      <h4 class="font-semibold text-xl text-primary">Brands</h4>
      <mat-selection-list [(ngModel)]="selectedBrands" [multiple]="true">
        @for (brand of shopService.brands; track brand) {
          <mat-list-option [value]="brand">{{brand}}</mat-list-option>
        }
      </mat-selection-list>
    </div>
    <div class="w-1/2">
      <h4 class="font-semibold text-xl text-primary">Types</h4>
      <mat-selection-list [(ngModel)]="selectedTypes" [multiple]="true">
        @for (type of shopService.types; track type) {
          <mat-list-option [value]="type">{{type}}</mat-list-option>
        }
      </mat-selection-list>
    </div>
  </div>
  <div class="flex justify-end p-4">
    <button mat-flat-button (click)="applyFilters()">Apply Filters</button>
  </div>
</div>
```

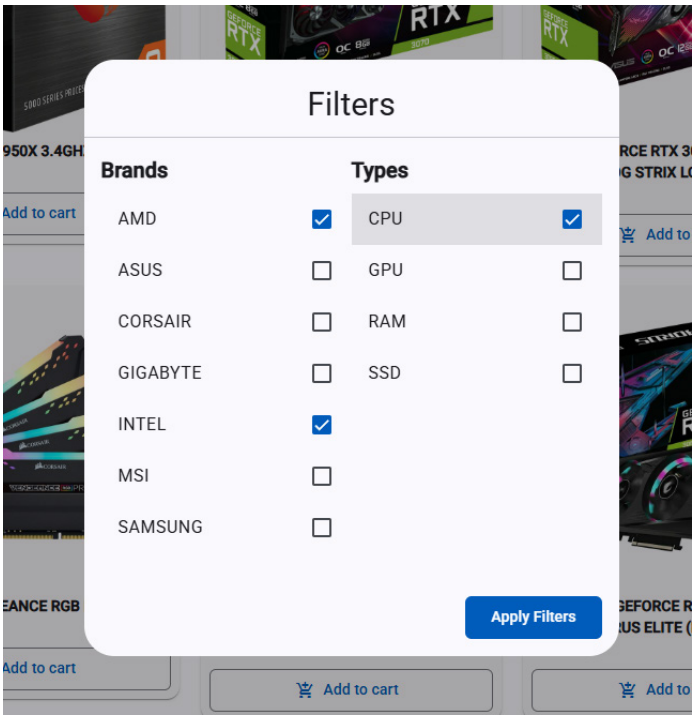
Εικόνα 182

Τέλος, μεταφερόμαστε στην *shop Angular service* (εικόνα 183) της εφαρμογής μας, προκειμένου να ενημερώσουμε το *HTTP request*. Ορίζουμε τις μεταβλητές *types* και *brands*, οι οποίες θα αποτελέσουν και προαιρετικά ορίσματα για τη μέθοδο *getProducts*. Ορίζουμε τη μεταβλητή με τίτλο *params* στην οποία αποθηκεύονται οι παράμετροι του *HTTP* αιτήματος. Εφόσον έχουν επιλεγεί φίλτρα από τον χρήστη, η πληροφορία που μας επιστρέφεται από το *filters-dialog component* δεν είναι ίση με null. Για τον λόγο αυτό χρησιμοποιούμε τις μεθόδους *append* και *join* ώστε να συλλέξουμε και να διαχωρίσουμε τα φίλτρα ανά κατηγορία (τύποι & μάρκες) και να τα αναθέσουμε στη μεταβλητή *params*. Θέτοντας το μέγιστο μέγεθος της σελίδας μας σε δεκαοχτώ προϊόντα, η μεταβλητή *params* χρησιμοποιείται εντός των ορισμάτων της μεθόδου *get*, προκειμένου να διαμορφωθεί το τελικό *HTTP* αίτημα προς τον *API server*.

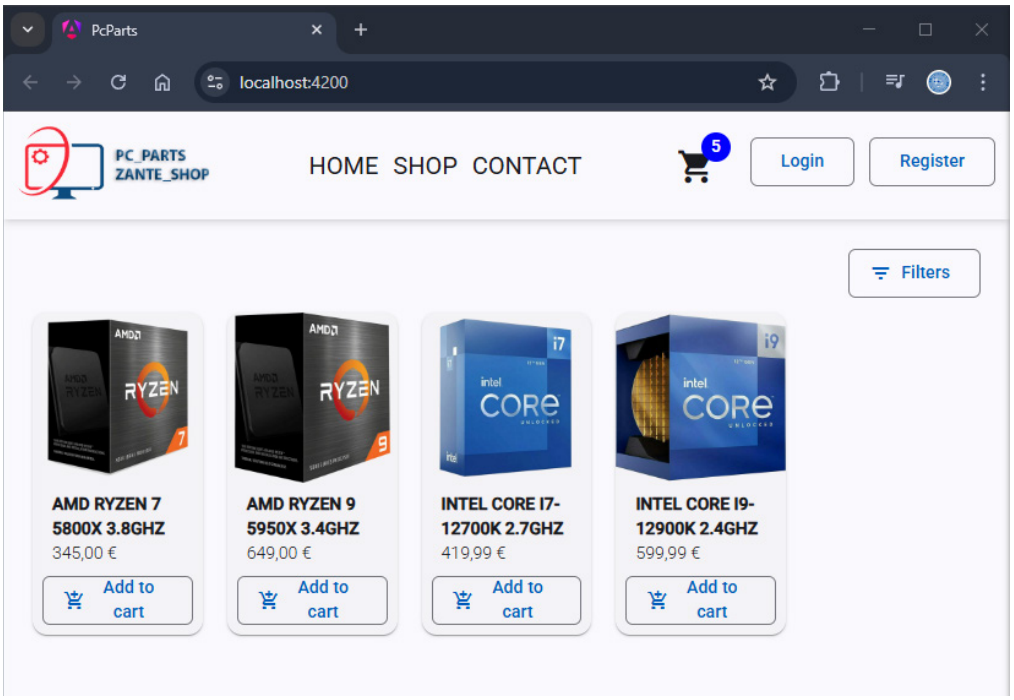
```
getProducts(brands?: string[], types?: string[]){
  let params = new HttpParams();

  if(brands && brands.length>0){
    params = params.append('brands', brands.join(','));
  }

  if(types && types.length>0){
    params = params.append('types', types.join(','));
  }

  params = params.append('pageSize', 18);

  return this.http.get<Pagination<Product>>(this.baseUrl + 'products', {params});
}
```

Εικόνα 183

Θέλοντας να ελέγξουμε τη λειτουργικότητα της εφαρμογής μας, στην εικόνα 184 θα εφαρμόσουμε ορισμένα φίλτρα αναζήτησης. Θα δοκιμάσουμε την αναζήτηση όλων των προϊόντων μάρκας *AMD* και μάρκας *INTEL* , τα οποία ανήκουν στην κατηγορία *CPU*. Όπως παρατηρούμε στην εικόνα 185, η λειτουργικότητα του σύνθετου φιλτραρίσματος στην εφαρμογή *Angular* έχει ολοκληρωθεί με επιτυχία.



Εικόνα 184



Εικόνα 185

6.5 Ταξινόμηση Προϊόντων

Στο αρχείο *shop.component.ts* (Εικόνα 186) θα ορίσουμε δύο νέες μεταβλητές, την *selectedSort* και την *sortOptions*. Στην πρώτη, θέτουμε ως όρισμα τη συμβολοσειρά *name*, ενώ στη δεύτερη δημιουργούμε μία στήλη με περιγραφή *name* και μία ακόμη με περιγραφή *value*. Οι τιμές που αναθέτουμε στις δύο αυτές στήλες, αντιπροσωπεύουν τις επιλογές που θα έχει ο χρήστης ώστε να μπορεί να ταξινομεί τη σειρά εμφάνισης των προϊόντων στον φυλλομετρητή. Στη συνέχεια, δημιουργούμε τη μέθοδο *onSortChange*, στην οποία θέτουμε ως όρισμα ένα *Angular* αντικείμενο τύπου *MatSelectionChange*. Η συγκεκριμένη μέθοδος θα χρησιμοποιηθεί από το αντίστοιχο *Angular html template*, προκειμένου ο χρήστης μέσα από μία λίστα τριών επιλογών (αλφαβητικά - αύξουσα τιμή - φθίνουσα τιμή), να ταξινομεί τα προϊόντα. Τέλος, με την κλήση της μεθόδου *getProducts* γίνεται η εμφάνιση των ταξινομημένων προϊόντων στον *browser*.

```
35    selectedSort: string ='name';
36    sortOptions = [
37      {name:'Alphabetical', value:'name'},
38      {name:'Price: Low-High', value:'priceAsc'},
39      {name:'Price: High-Low', value:'priceDesc'}
40    ]
41
42  ngOnInit(): void{
43    this.initializeShop();
44  }
45
46  initializeShop() { //lifecycle event init using types
47    this.shopService.getBrands();
48    this.shopService.getTypes();
49    this.getProducts();
50  }
51
52  getProducts(){//using angular service
53    //apply filters to product list by calling the shop Angular service
54    this.shopService.getProducts(this.selectedBrands, this.selectedTypes, this.selectedSort).subscribe({
55      next: response => this.products = response.data,//store data to array products
56        error: error => console.error(error)//test for errors
57    })
58  }
59
60  onSortChange(event: MatSelectionListChange){ //sorting products
61    const selectedOption = event.options[0];//default alphabetically
62    if(selectedOption){//receive data from template
63      this.selectedSort = selectedOption.value;
64      this.getProducts();
65    }
66  }
```

Εικόνα 186

Ακολούθως, εντός της *shop.service* θα εισάγουμε ένα νέο προαιρετικό όρισμα στη μέθοδο *getProducst*, τύπου *string* με όνομα *sort*. Στο σώμα της μεθόδου γίνεται έλεγχος της τιμής του κι ακολούθως ανατίθεται στη μεταβλητή *params*, χρησιμοποιώντας τη μέθοδο *append*, με τελικό στόχο να προστεθεί στο *HTTP* αίτημα προς τον *API server* (εικόνα 187).

```
18  getProducts(brands?: string[], types?: string[], sort?: string){
19    let params = new HttpParams();
20
21    if(brands && brands.length>0){
22      params = params.append('brands', brands.join(','));
23    }
24
25    if(types && types.length>0){
26      params = params.append('types', types.join(','));
27    }
28
29    if(sort){
30      params= params.append('sort',sort);
31    }
32
33    params = params.append('pageSize', 18);
34
35    return this.http.get<Pagination<Product>>(this.baseUrl + 'products', {params});
36  }
```

Εικόνα 187

Τελευταία ενέργεια για να καταστεί εφικτή η λειτουργικότητα της ταξινόμησης στον *client*, είναι η δημιουργία του ανάλογου κουμπιού στην ιστοσελίδα της εφαρμογής μας και η αλληλεπίδρασή του με τα αντίστοιχα *Angular components* εντός του *shop.component.html template*. Κάνοντας χρήση των αναλόγων *Angular* και *Tailwind attributes*, δημιουργούμε ένα νέο κουμπί με τίτλο *Sort* δίπλα σε αυτό του *Filters*. Όταν ο χρήστης το επιλέγει, αυτό αλληλεπιδρά μέσω της οντότητας *[matMenuTriggerFor]*, με την *template reference* μεταβλητή *sortMenu* (εικόνα 188).

shop.component.html

client > src > app > features > shop > shop.component.html > ...

```
    Go to component
1   <div class="flex flex-col gap-3">
2       <div class="flex justify-end pb-1 gap-3">
3           <button mat-stroked-button (click)="openFiltersDialog()">
4               <mat-icon>filter_list</mat-icon>
5               Filters
6           </button>
7           <button mat-stroked-button [matMenuTriggerFor]="sortMenu">
8               <mat-icon>filter_list</mat-icon>
9               Sort
10          </button>
11      </div>
12      <div class="grid grid-cols-5 gap-5">
13          @for (product of products; track product.id) {
14              <app-product-item [product]="product"></app-product-item>
15          }
16      </div>
17  </div>
18
```

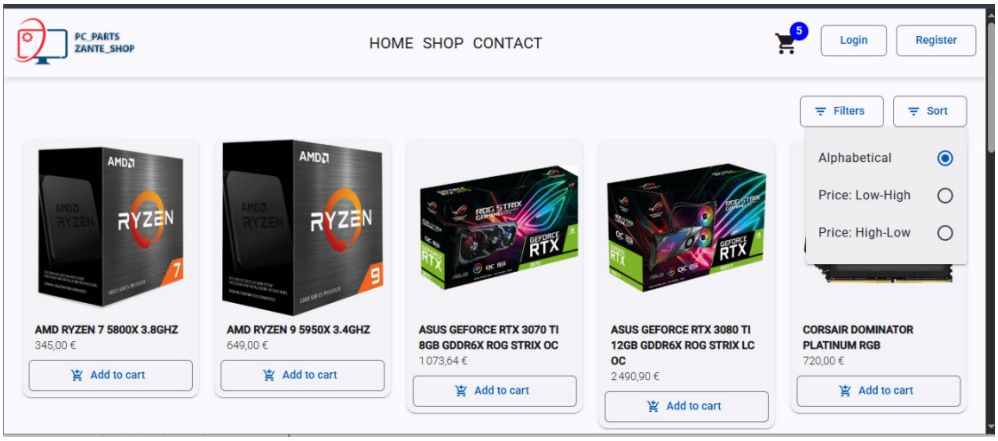
Εικόνα 188

Εντός του ίδιου *template* γίνεται και ο ορισμός της *sortmenu* μεταβλητής. Στην εικόνα 189, έχουμε τη δημιουργία μίας *drop-down* λίστας μονής επιλογής, όπου ο χρήστης επιλέγει (*event trigger*) τον τρόπο ταξινόμησης (μετά τη διάσχιση του πίνακα *sortOptions*) και μέσω της μεθόδου *onSortChange*, αποστέλλεται η πληροφορία *($event)* στο *shop.component.ts*.

```
19  <mat-menu #sortMenu="matMenu">
20      <mat-selection-list [multiple]="false" (selectionChange)="onSortChange($event)">
21          @for (sort of sortOptions; track $index) {
22              <mat-list-option [value]="sort.value" [selected]="selectedSort===sort.value">
23                  {{sort.name}}
24              </mat-list-option>
25          }
26      </mat-selection-list>
27  </mat-menu>
```

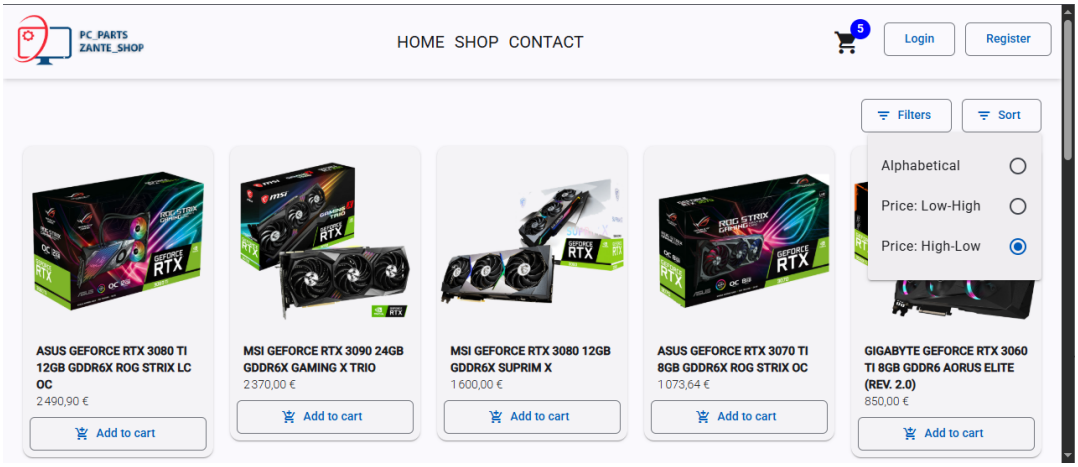
Εικόνα 189

Εκκινώντας την εφαρμογή μας, διαπιστώνουμε την ύπαρξη του νέου κουμπιού *Sort* (εικόνα 190), με την επιλογή της ταξινόμησης κατά αλφαβητική σειρά να είναι προκαθορισμένη.



Εικόνα 190

Κάνοντας την επιλογή ταξινόμησης κατά φθίνουσα τιμή (*Price High-Low*), έχουμε στην οθόνη του φυλλομετρητή μας το αποτέλεσμα της εικόνας 191.



Εικόνα 191

6.6 Αναδιαμόρφωση του API Request

Ο μεγάλος αριθμός ορισμάτων στη μέθοδο *getProducts*, εντός του *shop.component.ts* αρχείου, αλλά κι εντός της *shop Angular service*, μας αναγκάζει να αναδιαμορφώσουμε την προσέγγισή μας ως προς τη σύνταξη του εκάστοτε *HTTP* request προς τον *API server*. Με τις προσθήκες των λειτουργιών της σελιδοποίησης και της αναζήτησης να βρίσκονται εν όψη, θα δημιουργήσουμε μία νέα κλάση όπου θα είναι υπεύθυνη για τη φιλοξενία όλων των οντοτήτων της *Angular* εφαρμογής μας, οι οποίες είναι απαραίτητες για το φιλτράρισμα, την ταξινόμηση, τη σελιδοποίηση και την αναζήτηση όλων των προϊόντων της βάσης δεδομένων *pcparts.db*. Η νέα *TypeScript* κλάση ονομάζεται *shopParams.ts* και τη δημιουργούμε εντός του υποφάκελο *models*, του καταλόγου *shared*. Οι *properties* που ορίζουμε εντός της κλάσης *ShopParams* παρουσιάζονται στην εικόνα 192.

shopParams.ts

client > src > app > shared > models > shopParams.ts >

```
1  export class ShopParams{
2      //filtering,sorting,pagin
3      brands: string[] = [];
4      types: string[] = [];
5      sort: string='name';
6      pageNumber: number = 1;
7      pageSize: number = 10;
8      search: string = '';
9  }
10
```

Εικόνα 192

Στις εικόνες 193 έως και 196, παρουσιάζονται οι αλλαγές στις μεθόδους *getProducts*, *onSortChange*, *openFiltersDialog* και *afterClosed*, ύστερα από την αρχικοποίηση και χρήση της οντότητας *ShopParams* εντός του *shop.component.ts* αρχείου.

```
38    shopParams = new ShopParams(); //initialization
```

Εικόνα 193

```
50    getProducts(){//using angular service
51      //apply filters to product list by calling the shop Angula
52      this.shopService.getProducts(this.shopParams).subscribe({
53        next: response => this.products = response.data,//store
54          error: error => console.error(error)//test for errors
55      })
56    }
```

Εικόνα 194

```
58    onSortChange(event: MatSelectionListChange){ //sorting products
59      const selectedOption = event.options[0];//default alphabetically
60      if(selectedOption){//receive data from template
61        this.shopParams.sort = selectedOption.value;
62        this.getProducts();
63      }
64    }
```

Εικόνα 195

```
66    openFiltersDialog(){ //method which calls filtersdialogcomponent to o
67      const dialogRef = this.dialogService.open(FiltersDialogComponent, {
68        minWidth: '500px',//set minimum filters dioalog window width
69        data:{//receive and send data from/to filter dialog component
70          selectedBrands: this.shopParams.brands,
71          selectedTypes: this.shopParams.types
72        }
73      });
74      dialogRef.afterClosed().subscribe({ //using observer object to log
75      // in order to observe the received data after apply filters button
76        next: result => { //after filters dialog window is closed (apply
77          if(result){
78            this.shopParams.brands = result.selectedBrands;
79            this.shopParams.types = result.selectedTypes;
80            this.getProducts();
81          }
```

Εικόνα 196

Στην εικόνα 197, παρουσιάζεται η αλλαγή εντός του *shop.component.html template*.

```
19  <mat-menu #sortMenu="matMenu">
20      <mat-selection-list [multiple]="false" (selectionChange)="onSortChange($event)">
21          @for (sort of sortOptions; track $index) {
22              <mat-list-option [value]="sort.value" [selected]="shopParams.sort === sort.value">
23                  {{sort.name}}
24              </mat-list-option>
25          }
26      </mat-selection-list>
27  </mat-menu>
```

Εικόνα 197

Τέλος, στην εικόνα 198 βλέπουμε τις απαραίτητες αλλαγές εντός της *Angular shop service*.

```
getProducts(shopParams: ShopParams){
  let params = new HttpParams();

  if(shopParams.brands.length>0){
    params = params.append('brands',shopParams.brands.join(','));
  }

  if(shopParams.types.length>0){
    params = params.append('types',shopParams.types.join(','));
  }

  if(shopParams.sort){
    params= params.append('sort',shopParams.sort);
  }

  params = params.append('pageSize', 18);

  return this.http.get<Pagination<Product>>(this.baseUrl + 'products', {params});
}
```

Εικόνα 198

Πρακτικά, πέραν της βελτίωσης ως προς τη συνεκτικότητα και τη δομή του κώδικα, δεν πραγματοποιήθηκε καμία μεταβολή ως προς τη λειτουργικότητα της εφαρμογής μας.

6.7 Σελιδοποίηση Προϊόντων

Σε αυτήν την ενότητα θα προσθέσουμε τη λειτουργικότητα της σελιδοποίησης στην *Angular* εφαρμογή μας. Ξεκινώντας, θα μεταβούμε στο *Angular shop.service.ts* αρχείο, όπου θα προσθέσουμε δύο νέες παραμέτρους στη μέθοδο *getProducts*, με στόχο αυτές να προστεθούν στο *HTTP* αίτημά μας (εικόνα 199). Οι παράμετροι αυτές δεν είναι άλλες από το μέγεθος της σελίδας και τον αριθμό της σελίδας, τις οποίες καθορίζει ο χρήστης.

```
34      params = params.append('pageSize', shopParams.pageSize);
35      params = params.append('pageIndex', shopParams.pageNumber);
```

Εικόνα 199

Συνεχίζοντας, θα ανοίξουμε το αρχείο *shop.component.ts* κι εκεί θα κάνουμε *initialize* τη μεταβλητή *products* δίνοντάς της τον τύπο *Pagination<Product>*. Επίσης, θα ορίσουμε τη μεταβλητή *pageSizeOptions*, ώστε ο χρήστης να μπορεί να επιλέγει το μέγεθος (αριθμό προϊόντων ανά σελίδα) της εκάστοτε σελίδας (εικόνα 200).

```
34    products?: Pagination<Product>;
35
36    sortOptions = [
37     {name:'Alphabetical', value:'nar
38     {name:'Price: Low-High', value:
39     {name:'Price: High-Low', value:
40    ]
41    shopParams = new ShopParams(); //
42    pageSizeOptions = [5,10,15,20];
```

Εικόνα 200

Ακολούθως, ορίζουμε τη μέθοδο *handlePageEvent*. Η μέθοδος αυτή δέχεται ως όρισμά της μία κλάση τύπου *MatPaginatorModule* με όνομα *PageEvent*. Μέσω αυτής της μεθόδου, μόλις ο χρήστης κάνει μία επιλογή εντός του *shop.component.html template*, ενημερώνεται αυτόματα ο αριθμός και το μέγεθος της τρέχουσας σελίδας και η πληροφορία μεταφέρεται στο εν λόγω *component*. Όσον αφορά τον αριθμό της σελίδας που μας επιστρέφεται από τον *API server*, λόγω του ότι το *indexing* εκκινεί από τον αριθμό μηδέν, εντός του *Angular project* οφείλουμε να προσθέτουμε τον αριθμό ένα κατά την εκάστοτε καταχώρηση της επιλογής του χρήστη, ώστε κάθε φορά να επιστρέφεται ο σωστός αριθμός τρέχουσας σελίδας (εικόνα 201).

```
62    //event which triggered from template
63    handlePageEvent(event: PageEvent){//when users chan
64      this.shopParams.pageNumber = event.pageIndex + 1;
65      this.shopParams.pageSize = event.pageSize;//itmes
66      this.getProducts();
67    }
```

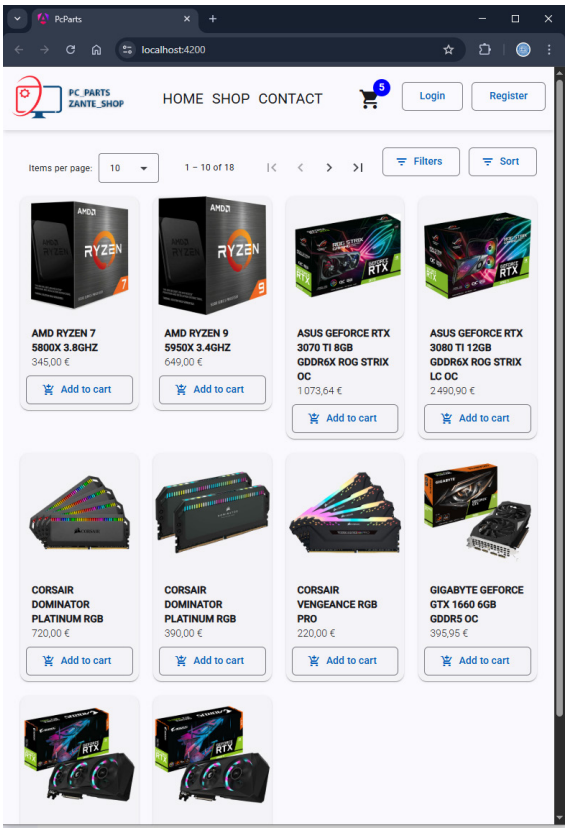
Εικόνα 201

Κλείνοντας με την παραμετροποίηση της σελιδοποίησης, μεταβαίνουμε στο *Angular template shop.component.html*. Εντός του πεδίου των φίλτρων αναζήτησης και ταξινόμησης, θα προσθέσουμε τα απαραίτητα κουμπιά για τη σελιδοποίηση. Δημιουργούμε ένα ενιαίο * element* κι εντός του ορίζουμε ένα *Angular mat-paginator component*. Στην εικόνα 202, παρουσιάζονται όλες οι οντότητες ενός τέτοιου *component* τις οποίες και αρχικοποιούμε. Κατά σειρά, αυτές αφορούν το σύνολο των προϊόντων στης σελίδας (λαμβάνοντας υπόψιν ενδεχόμενο φιλτράρισμα), το μέγεθος της σελίδας, κουμπιά μετάβασης στην πρώτη και την τελευταία σελίδα καθώς και τον αριθμό της τρέχουσας σελίδας (εικόνα 202).

```
<div class="flex flex-col gap-3">
    <div class="flex justify-between">
        <mat-paginator
            (page)="handlePageEvent($event)"
            [length]="products?.count"
            [pageSize]="shopParams.pageSize"
            [showFirstLastButtons]="true"
            [pageSizeOptions]="pageSizeOptions"
            [pageIndex]="shopParams.pageNumber - 1"
            aria-label="Select page"
        >
        </mat-paginator>
```

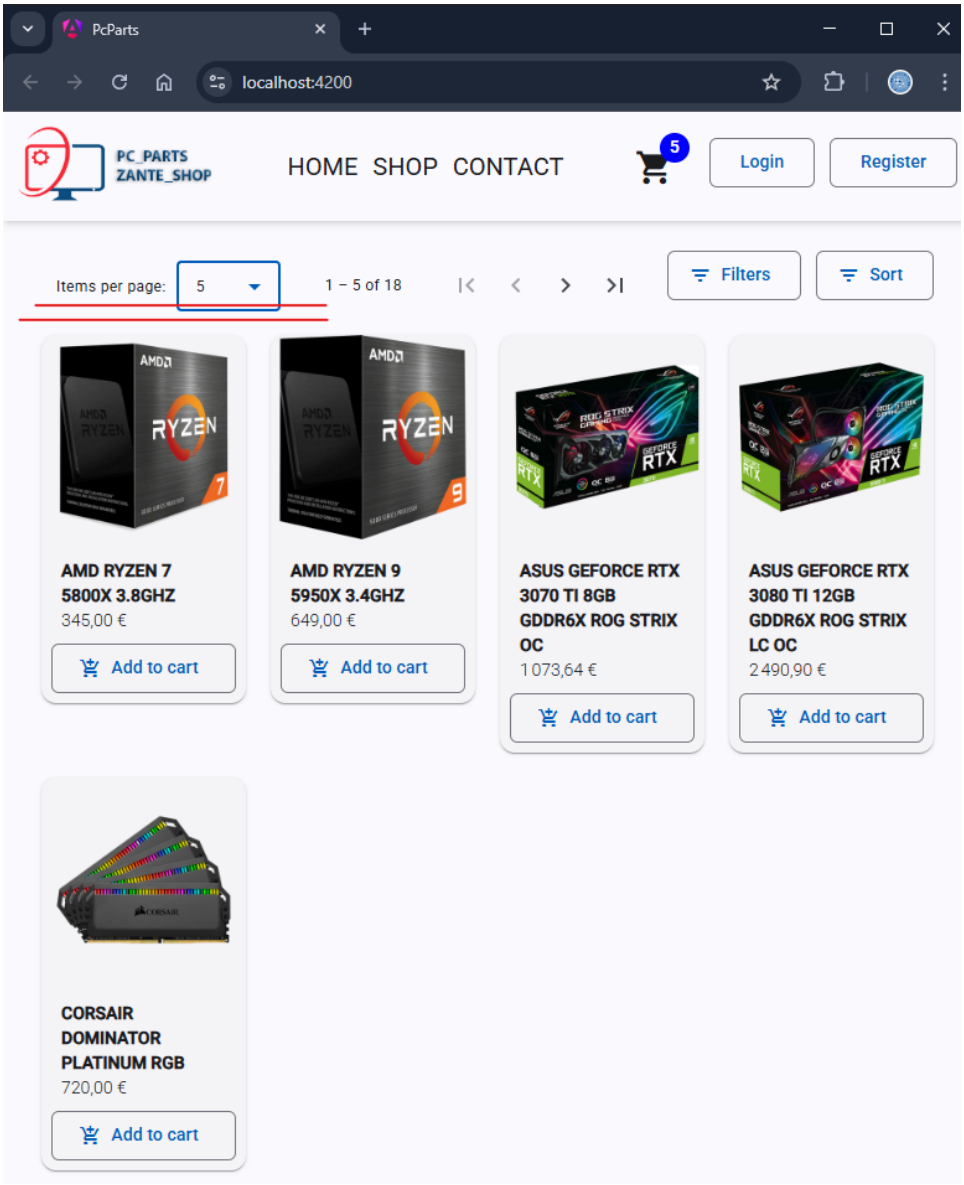
Εικόνα 202

Εκτελώντας την *Angular* εφαρμογή μας, εμφανίζεται το αποτέλεσμα της εικόνας 203, το οποίο αποτυπώνει το *pagination template* σε περιβάλλον *browser*.



Εικόνα 203

Στην προηγούμενη εικόνα εμφανίζονται τα πρώτα δέκα προϊόντα της βάσης δεδομένων (προκαθορισμένο μέγεθος σελίδας) κατά αλφαβητική σειρά. Επιλέγοντας τον αριθμό πέντε από τις επιλογές του *pagination template*, παίρνουμε το αποτέλεσμα της εικόνας 204.



Εικόνα 204

Ακριβώς δίπλα από την επιλογή του μεγέθους της σελίδας, έχουμε την ένδειξη για τον αριθμό των προϊόντων στην τρέχουσα σελίδα, αλλά και τον αριθμό των προϊόντων που βρίσκονται στις επόμενες σελίδες και είναι διαθέσιμα για προσπέλαση. Τέλος, ο χρήστης έχει τη δυνατότητα της περιήγησης σε επόμενη ή προηγούμενη σελίδα, καθώς και τη μετάβαση απευθείας στην πρώτη ή την τελευταία διαθέσιμη σελίδα, με βάση τα φίλτρα και την ταξινόμηση που ο ίδιος έχει ορίσει.

6.8 Αναζήτηση Προϊόντων

Ολοκληρώνοντας το τρέχον κεφάλαιο, θα προσθέσουμε ακόμη μία λειτουργικότητα στην *Angular* εφαρμογή μας. Η λειτουργικότητα αυτή αφορά την αναζήτηση προϊόντων από τον χρήστη, εισάγοντας μία λέξη κλειδί στο περιβάλλον του *browser*. Ξεκινώντας, θα ορίσουμε μία ακόμη μέθοδο εντός της *TypeScript* κλάσης *ShopComponent*, με τίτλο *onSearchChange*. Εντός της συγκεκριμένης μεθόδου ορίζουμε τον αριθμό τρέχουσας σελίδας σε ένα και στη συνέχεια καλούμε τη μέθοδο *getProducts* για την προβολή των προϊόντων στον *client* (εικόνα 205).

```
65    //Searching event
66    onSearchChange(){
67      this.shopParams.pageNumber = 1;
68      this.getProducts();
69    }
```

Εικόνα 205

Εν συνεχεία, εντός της *shop.service.ts* εισάγουμε την παράμετρο *search* στη μεταβλητή *params*, εφόσον δεν είναι κενή δεδομένων (εικόνα 206).

```
if(shopParams.search){
  params = params.append('search', shopParams.search);
}
```

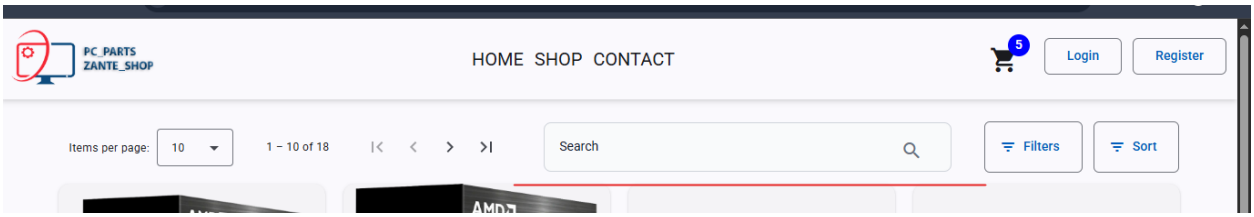
Εικόνα 206

Τέλος, εντός του *shop.component.html template* δημιουργούμε ένα *form group*, ανάμεσα στο *mat-paginator - pagination template* και στο *html div element* που περιέχει τα κουμπιά ταξινόμησης και φιλτραρίσματος. Εκμεταλλευόμενοι την ιδιότητα του *Angular Two-Way Binding*, χρησιμοποιούμε τη μεταβλητή (*event emiter*) (*ngSubmit*) για να καλέσουμε τη μέθοδο *onSearchChange* (εικόνα 205). Ακολούθως, δημιουργούμε το προς εμφάνιση στον *browser* πλαίσιο αναζήτησης, όπου ο χρήστης εισάγει έναν όρο προς αναζήτηση κάνοντας «κλικ» στο εικονίδιο *mat-icon search*. Μόλις ο χρήστης εκτελέσει την αναζήτηση ενεργοποιείται (*event trigger*) η *directive [(ngModel)]* και η πληροφορία μεταφέρεται στη μεταβλητή *shopParams.search* και κατ' επέκταση στην *TypeScript* κλάση *ShopComponent* (εικόνα 207).

```
<form #searchForm="ngForm"
      (ngSubmit)="onSearchChange()"
      class="relative flex items-center w-full max-w-md mx-4">
  <input
    type="search"
    class="block w-full p-4 text-sm text-gray-900 border border-gray-300 rounded-lg bg-gray-50
     focus:border-blue-500 focus:ring-blue-500"
    placeholder="Search"
    name="search"
    [(ngModel)]="shopParams.search" />
  <button mat-icon-button type="submit"
      class="absolute inset-y-0 right-8 top-2 flex items-center pl-3">
      <mat-icon class="text-gray-500">search</mat-icon>
  </button>
</form>
```

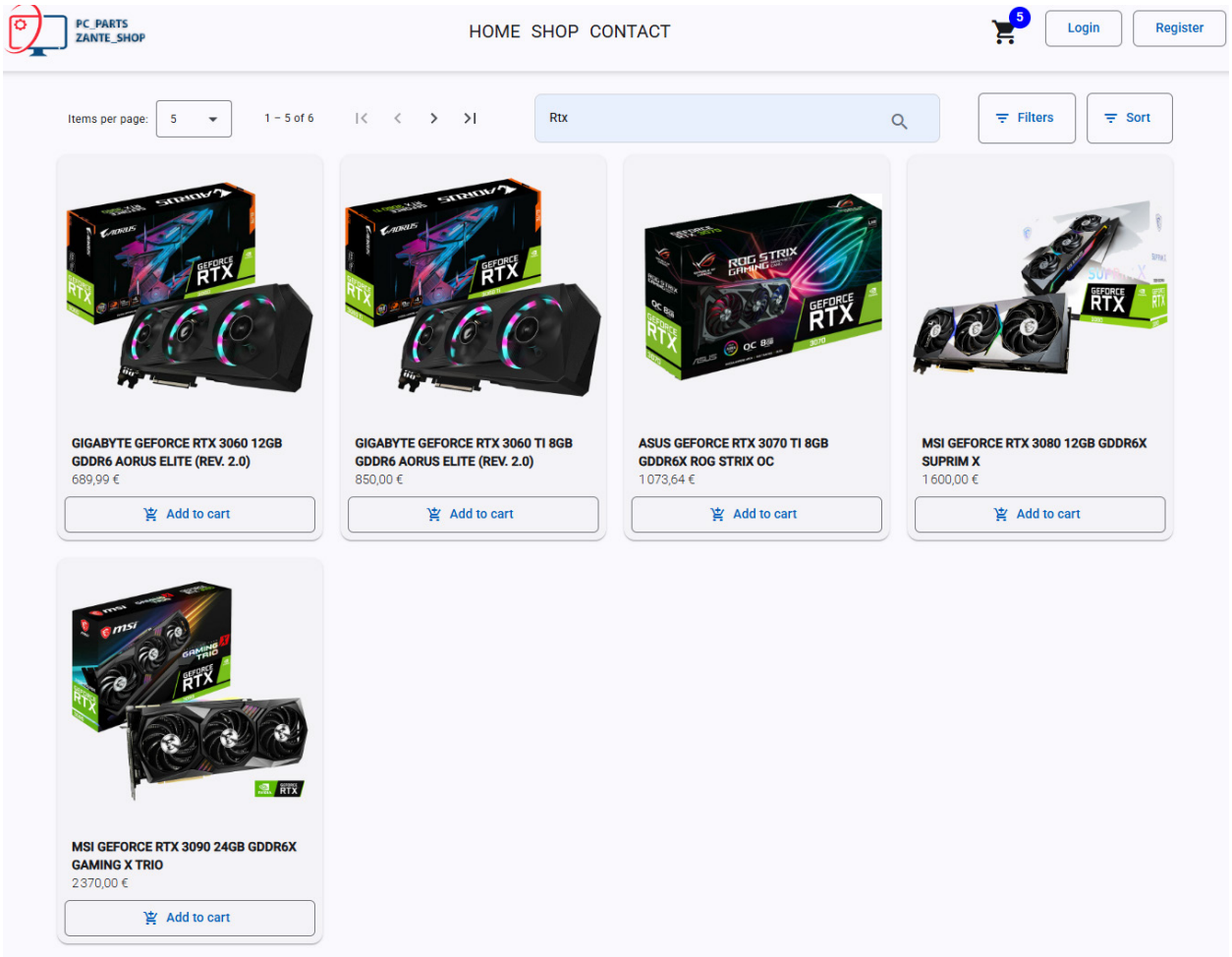
Εικόνα 207

Κατά την εκτέλεση της εφαρμογής μας, παρατηρούμε την προσθήκη του πλαισίου αναζήτησης στην οθόνη του *browser* (εικόνα 208).

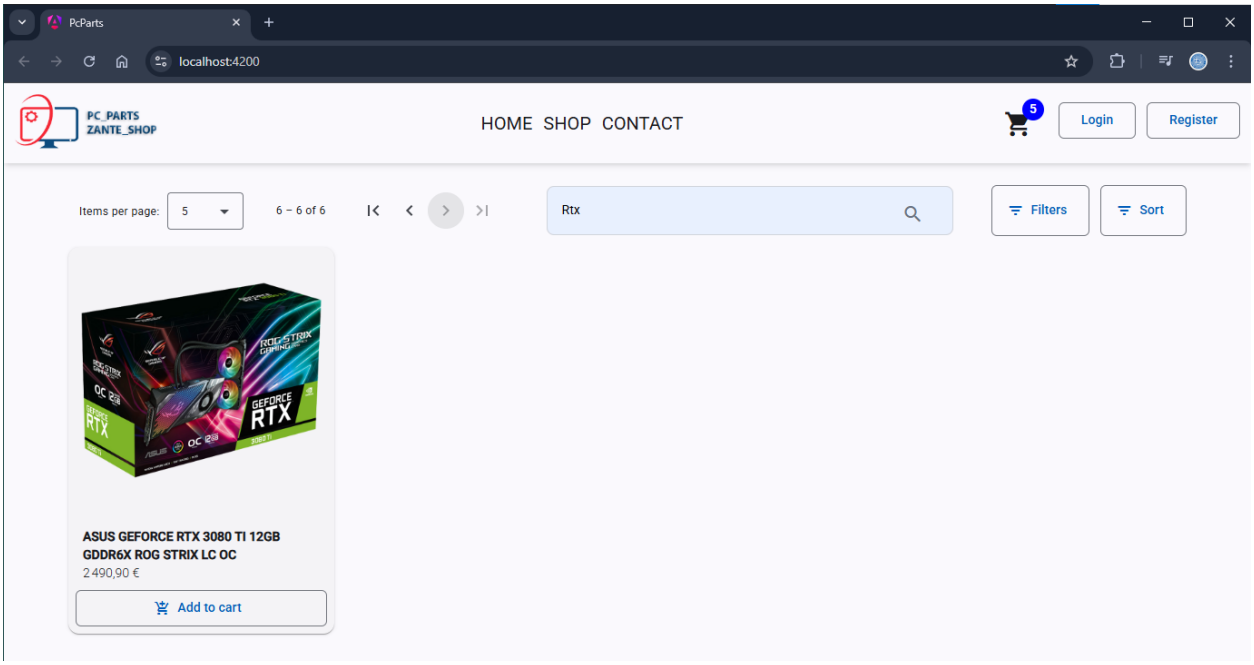


Εικόνα 208

Δοκιμάζοντας την αναζήτηση του όρου «*Rtx*», σε μέγεθος σελίδας πέντε προϊόντων και ταξινόμηση κατά αύξουσα τιμή πώλησης, λαμβάνουμε έξι αποτελέσματα τα οποία παρουσιάζονται στις εικόνες 209 και 210 αντίστοιχα.



Εικόνα 209


PcParts
localhost:4200
PC_PARTS
ZANTE_SHOP
HOME SHOP CONTACT
5
Login
Register
Items per page:
5
6 – 6 of 6
Rtx
Filters
Sort
ASUS GEFORCE RTX 3080 TI 12GB GDDR6X ROG STRIX LC OC
2 490,90 €
Add to cart

Εικόνα 210

Κεφάλαιο 7. Angular Routing

Οι εφαρμογές μονής σελίδας (*single page applications*), όπως στην περίπτωσή μας είναι η *e-commerce application* που αναπτύσσουμε, χρειάζονται τη λειτουργικότητα του *routing* εντός της δομής τους. Το *routing* (δρομολόγηση) είναι αρμόδιο για την εναλλαγή των σελίδων που παρουσιάζονται στον *client* μετά από κάθε *HTTP* αίτημά του. Το *Angular Routing* φιλοξενεί τη βιβλιοθήκη *(@angular/router*) για τη διαχείριση της πλοήγησης σε εφαρμογές *Angular* και αποτελεί βασικό μέρος του περιβάλλοντος ανάπτυξης του οικοσυστήματος. Η συγκεκριμένη βιβλιοθήκη είναι προ εγκατεστημένη σε οποιοδήποτε *Angular project*, δημιουργείται μέσω του *Angular CLI* (*Command Line Interface*), όπως φυσικά είναι και το *PcParts project*.

7.1 Δημιουργία Νέων Components και Προσθήκη των Angular Routes

Ανοίγοντας ένα παράθυρο τερματικού εντός του *Visual Studio Code*, θα εκτελέσουμε τις εντολές *ng g c features/home --skip-tests* και *ng g c features/product-details --skip-tests* κατά σειρά, όπως βλέπουμε στην εικόνα 211. Εκτελώντας αυτές τις δύο εντολές, δημιουργούμε δύο νέα *Angular components* εντός του καταλόγου *features*, με τίτλο *home.componet.ts* και *product-details.component.ts* αντίστοιχα.

```
PS C:\Users\User\Desktop\pcparts\client> ng g c features/home --skip-tests
CREATE src/app/features/home/home.component.ts (219 bytes)
CREATE src/app/features/home/home.component.scss (0 bytes)
CREATE src/app/features/home/home.component.html (20 bytes)
PS C:\Users\User\Desktop\pcparts\client> ng g c features/product-details --skip-tests
CREATE src/app/features/product-details/product-details.component.ts (262 bytes)
CREATE src/app/features/product-details/product-details.component.scss (0 bytes)
CREATE src/app/features/product-details/product-details.component.html (31 bytes)
PS C:\Users\User\Desktop\pcparts\client>
```

Εικόνα 211

Εντός του *home.component.html template*, θα φιλοξενηθεί ο απαραίτητος κώδικας για τη διαμόρφωση της αρχική σελίδας της *web* εφαρμογής μας. Προς το παρόν, θέλοντας να ελέγξουμε τη λειτουργικότητά της μέσω του φυλλομετρητή μας, θα προσθέσουμε ένα απλό κείμενο (εικόνα 212). Το ίδιο θα πράξουμε και για το *product-details.html template* (εικόνα 213), με τη διαφορά πως το συγκεκριμένο αρχείο θα φιλοξενήσει τον κώδικα που διαμορφώνει τη σελίδα που θα περιέχει τις λεπτομέρειες που περιγράφουν ένα προϊόν.

```
home.component.html ×
client > src > app > features > home > home.component.html > p.text-4xl
    Go to component
1   <p class="text-4xl">HOME PAGE will be constructed here!</p>
```

Εικόνα 212

```
tes.ts   product-details.component.html ●   home.component.html
: > app > features > product-details > product-details.component.html > ...
Go to component
<p class="text-4xl">PRODUCT DETAILS page will be constructed here!</p>
```

Εικόνα 213

Στη συνέχεια, θα ανοίξουμε το αρχείο *app.routes.ts* κι εκεί θα προσθέσουμε τέσσερα νέα *URL paths* εντός του πίνακα *Routes* (εικόνα 214). Η παράμετρος *path* καθορίζει τη διαδρομή του *URL* που μας επιστρέφει την αρχική σελίδα της εφαρμογής μας, το δεύτερο *path* μας επιστρέφει τη σελίδα του *e-shop*, το τρίτο μας οδηγεί στη σελίδα με τη λεπτομερή περιγραφή ενός προϊόντος βάσει του *id* του, ενώ το τελευταίο καλύπτει την οποιαδήποτε άλλη αναζήτηση μη υποστηριζόμενου *URL* από τον *client*, κάνοντας ανακατεύθυνση (*redirect*) στην αρχική σελίδα του *online* καταστήματός μας.

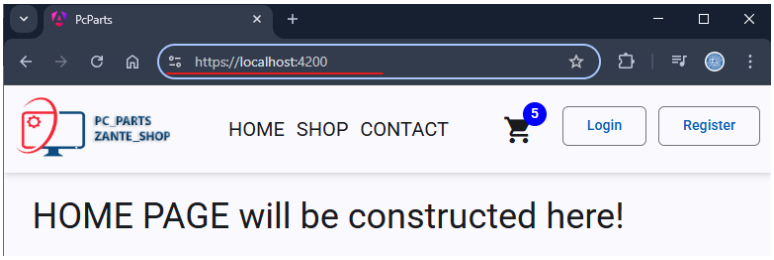
app.routes.ts | product-details.component.html | home.component.html

client > src > app > app.routes.ts > ...

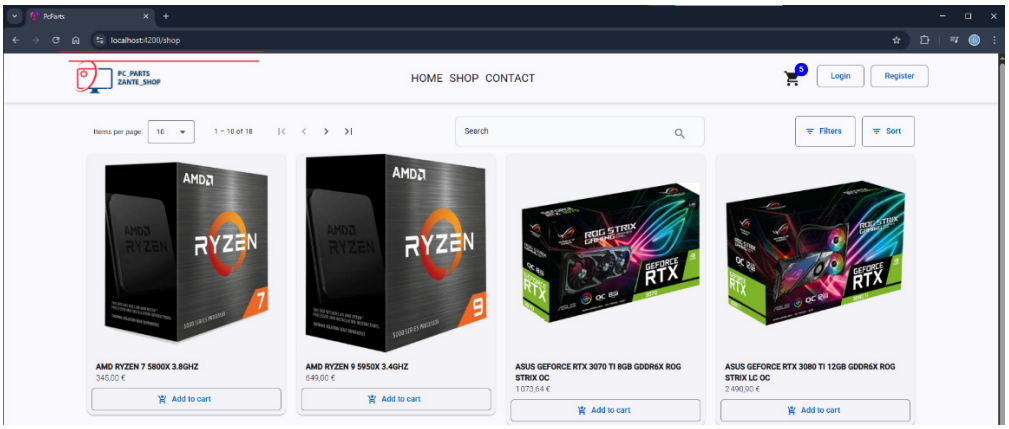
```
1  import { Routes } from '@angular/router';
2  import { HomeComponent } from './features/home/home.compon
3  import { ShopComponent } from './features/shop/shop.compon
4  import { ProductDetailsComponent } from './features/produc
5
6  export const routes: Routes = [
7      {path:'', component: HomeComponent},
8      {path:'shop', component: ShopComponent},
9      {path:'shop/:id', component: ProductDetailsComponent},
10     {path:'**', redirectTo:'', pathMatch:'full'}
11 ];
```

Εικόνα 214

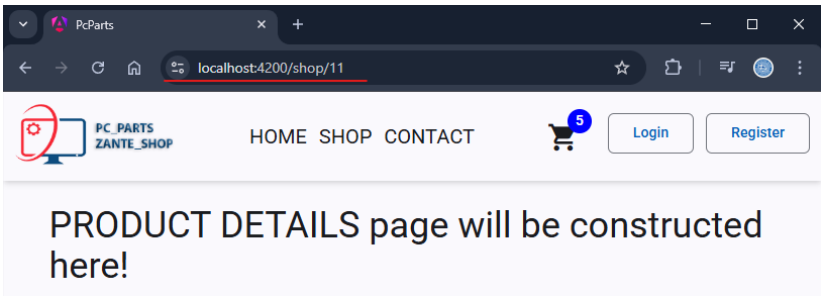
Εισάγοντας τα *URLs*: https://localhost:4200/ (εικόνα 215) - https://localhost:4200/shop (εικόνα 216) - https://localhost:4200/shop/11 (εικόνα 217), λαμβάνουμε στον *browser* τα παρακάτω αποτελέσματα.



Εικόνα 215



Εικόνα 216



Εικόνα 217

7.2 Προσθήκη Συνδέσμων Ανακατεύθυνσης

Έχοντας δημιουργήσει δύο νέα *components*, τα οποία ουσιαστικά αποτελούν δύο νέες σελίδες για την εφαρμογή μας, θα πρέπει να παράσχουμε και τα κατάλληλα *links* (συνδέσμους) εντός του φυλλομετρητή, ώστε ο χρήστης να είναι σε θέση να τα επιλέγει και να μπορεί να πλοηγείτε στην εκάστοτε σελίδα του *e-shop* μας. Ανοίγοντας τον κώδικα του αρχείου *header.component.html*, θα δοκιμάσουμε να προσδώσουμε χαρακτήρα «*link*» στα κουμπιά *Home*, *Shop* και στο *attribute* που περιέχει τον σύνδεσμο με το λογότυπο του ψηφιακού μας καταστήματος. Αρχικά, χρησιμοποιούμε τα *routerLink attributes* για να μας δώσουν τη δυνατότητα πρόσβασης στο εκάστοτε *component*, βασιζόμενα στον κανόνα δόμησης *URL («/» ή «/shop»)* εντός του αρχείου *app.routes.ts*. Στη συνέχεια, με τα *attributes routerLinkActive* και *routerLinkActiveOptions = "{exact:true}"*, πετυχαίνουμε τον χρωματισμό του ενεργού *link* κατά την επιλογή του από τον χρήστη. Τέλος, μέσω του *attribute text-3xl* και *font-bold* δίνουμε έναν πιο έντονο χαρακτήρα κι ένα μεγαλύτερο μέγεθος στη γραμματοσειρά των κουμπιών *Home*, *Shop* και *Contact*. Ο κώδικας του *home.component.html template* παρουσιάζεται στην εικόνα 218.

header.component.html

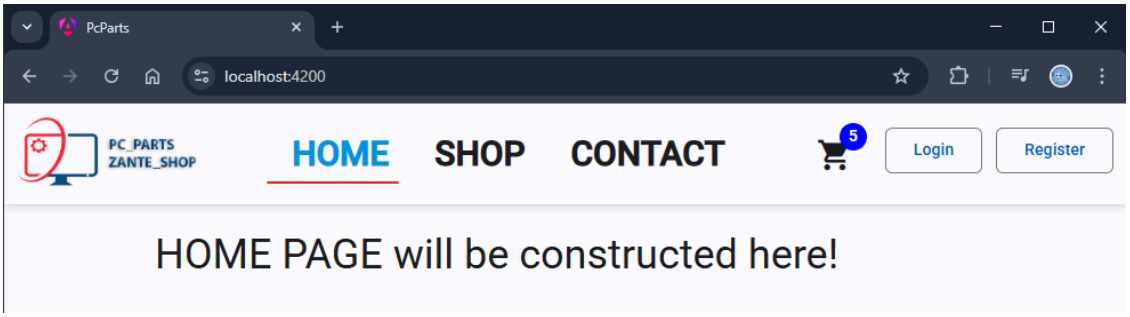
client > src > app > layout > header > header.component.html > header.shadow-md.p-3.w-full

Go to component

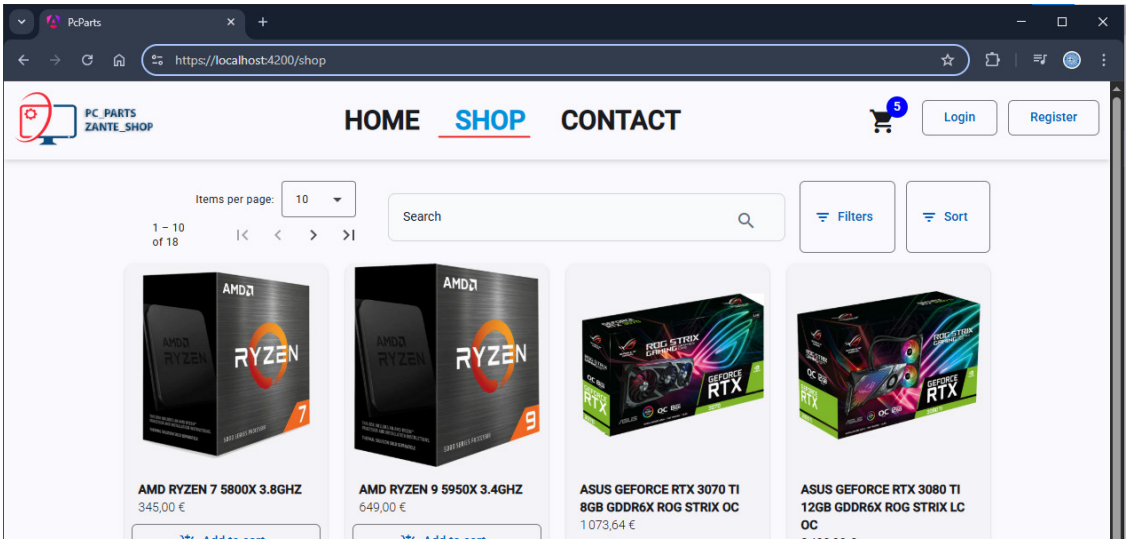
```
<header class="shadow-md p-3 w-full">
    <div class="flex align-middle items-center justify-between max-w-screen-2xl mx-auto">
        <img routerLink="/" src="/images/logo.png" alt="pcparts logo" class="max-h-16">
        <nav class="flex gap-x-10 my-2 uppercase text-3xl font-bold">
            <a routerLink="/"
            routerLinkActive="active"
            [routerLinkActiveOptions]="{exact:true}"
            >Home</a>
            <a routerLink="/shop"
            routerLinkActive="active">Shop</a>
            <a>Contact</a>
        </nav>
        <div class="flex gap-3 align-middle">
                <a matBadge="5" matBadgeSize="large" class="custom-badge mt-2 mr-4">
                    <mat-icon>shopping_cart</mat-icon>
                </a>
            <button mat-stroked-button>Login</button>
            <button mat-stroked-button>Register</button>
        </div>
    </div>
</header>
```

Εικόνα 218

Εισάγοντας τα *URLs*: https://localhost:4200/ (εικόνα 219) - https://localhost:4200/shop (εικόνα 220), διαπιστώνουμε την ορθή λειτουργία των *links*, καθώς και τις αλλαγές στη γραμματοσειρά τους.



Εικόνα 219



Εικόνα 220

7.3 Διαμόρφωση της Σελίδας Προϊόντος

Κάθε *web* εφαρμογή της μορφής *e-shop*, δίνει τη δυνατότητα στον χρήστη να επιλέγει οποιοδήποτε προϊόν και να προβάλλει τη λεπτομερή περιγραφή του. Έτσι, και στην περίπτωση της εφαρμογής *PcParts*, θα διαμορφώσουμε το *product-details component* για να φιλοξενήσει τη συγκεκριμένη λειτουργικότητα. Ξεκινώντας, θα ανοίξουμε την *shop Angular service* και θα προσθέσουμε τη μέθοδο *getProduct*. Μέσω της συγκεκριμένης μεθόδου, προσθέτουμε το *id* του προϊόντος που επιλέγει ο *client* για προβολή των λεπτομερειών του και το συμπεριλαμβάνουμε στις παραμέτρους του *HTTP* αιτήματος προς τον *API server* (εικόνα 221).

```
44    getProduct(id: number){
45      return this.http.get<Product>(this.baseUrl +'products/' + id);
46    }
```

Εικόνα 221

Στη συνέχεια θα ανοίξουμε το αρχείο *product-details.component.ts*, όπου κάνουμε *inject* τις κλάσεις *ShopService* και *ActivatedRoute*. Η πρώτη, μας δίνει πρόσβαση στη μέθοδο *getProduct*, ώστε μέσω του *observable object* να αντλήσουμε το *id* του επιλεγμένου από τον χρήστη προϊόντος. Η δεύτερη, μας δίνει τη δυνατότητα να χρησιμοποιήσουμε τις *Angular* οντότητες *snapshot* και *paramMap*, οι οποίες μέσω του *template product-details.component.html*, καταγράφουν το «κλικ» επάνω στην εικόνα του εκάστοτε προϊόντος με σκοπό να αντλήσουμε το *id* του. Τέλος, εφόσον έχουμε αντλήσει το *id* του προϊόντος, μέσω του *observable object* στη μέθοδο *loadProduct*, αναθέτουμε όλη την πληροφορία που το περιγράφει και την αποθηκεύουμε στη μεταβλητή *product* (εικόνα 222).

```
export class ProductDetailsComponent implements OnInit{
  //access to shopservice and getprodut method
  private shopService = inject(ShopService);
  //get product id from selected item's link route
  private activatedRoute = inject(ActivatedRoute);
  product?: Product; //product initialization

  ngOnInit(): void {
    this.loadProduct(); //getting the specific product
  }

  loadProduct(){ //using observable object in order to get the ID from
    //activated route method and returns the specific product
    const id = this.activatedRoute.snapshot.paramMap.get('id');
    if(!id) return;
    this.shopService.getProduct(+id).subscribe({ //+ typecasts string to number
      next: product => this.product = product,
      error: error => console.log(error)
    })
```

Εικόνα 222

Ακολούθως, μεταφερόμαστε στο *template product-item.html* όπου χρησιμοποιούμε τα *attributes routerlink* και την κλάση *product-card*, τα οποία μας δίνουν τη δυνατότητα να επιλέγουμε τη φωτογραφία ενός προϊόντος, να δημιουργείται ένα *hover-effect* και κάνοντας «κλικ», να αντλούμε το *id* του συγκεκριμένου προϊόντος, μέσω του *URL* ανακατεύθυνσης προς τη σελίδα του *product-details.html template*.

```
<mat-card appearance="raised" routerLink="/shop/{{product.id}}" class="product-card">
    <img src="{{product.pictureUrl}}" alt="{{product.name}}">
    <mat-card-content class="mt-4">
        <h2 class="text-sm font-semibold uppercase">{{product.name}}</h2>
        <p class="font-light">{{product.price|currency:'EUR':'symbol':'1.2-2':'fr'}}</p>
    </mat-card-content>
```

Εικόνα 223

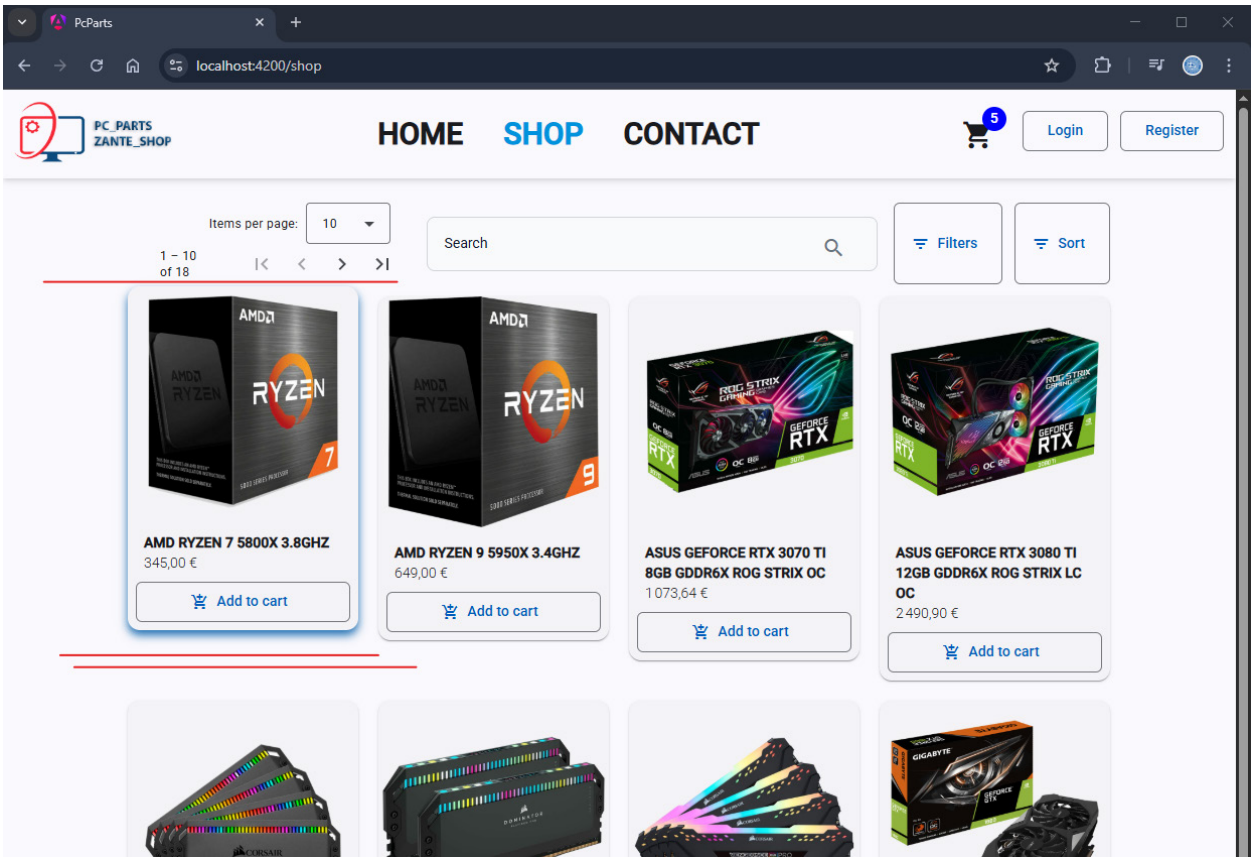
Προκειμένου να δημιουργηθεί το *styling* αναφορικά με το *hover effect*, κατά το πέρασμα του δείκτη του ποντικιού επάνω από την εικόνα του εκάστοτε προϊόντος, θα ορίσουμε την *SCSS* κλάση *product-card* εντός του αρχείου *product-item.scss* (εικόνα 224). Περνώντας τον δείκτη του ποντικιού επάνω από την κάρτα με την εικόνα ενός προϊόντος, γίνεται μία ανοδική μετακίνηση της κάρτας στον κατακόρυφο άξονα *Y* κατά 10 *pixels*, καθώς και μία ελαφρά σκίαση μπλε απόχρωσης περιμετρικά της κάρτας (εικόνα 225). Με αυτόν τον τρόπο, ο χρήστης αντιλαμβάνεται καλύτερα το προϊόν που είναι επιλεγμένο για προβολή των λεπτομερειών του.

product-item.component.scss

client > src > app > features > shop > product-item > product-item.component.scss > ...

```scss
.product-card{
    transition: transform 0.2s, box-shadow 0.2s;
}

.product-card:hover{
    transform: translateY(-10px);
    box-shadow: 0 5px 10px rgba(13, 101, 173, 0.925);
    cursor: pointer;
}
```

Εικόνα 224



Εικόνα 225

Τέλος, θα διαμορφώσουμε τον *html* κώδικα εντός του αρχείου *product-details.html*, όπως παρουσιάζεται στην εικόνα 226. Κάνοντας χρήση των κατάλληλων *Angular - Tailwind* και *html* attributes, δημιουργούμε δύο κύριες στήλες. Στην πρώτη περιλαμβάνεται η εικόνα του προϊόντος, ενώ στη δεύτερη παρουσιάζονται σε τρία επίπεδα, ο τίτλος του προϊόντος, η τιμή πώλησης, τα κουμπιά «*Add to cart*» και «*Insert Quantity*», καθώς και η διαθεσιμότητά του. Στο τελευταίο επίπεδο, κάτω από ένα *Angular mat-divider attribute*, τοποθετείται η αναλυτική περιγραφή του προϊόντος.

product-details.component.html × product-details.component.scss

client > src > app > features > product-details > product-details.component.html > ...

Go to component

```
@if(product){
    <section class="py-8">
        <div class="max-w-screen-2xl px-4 mx-auto">
            <div class="grid grid-cols-2 gap-8">
                <div class="max-w-xl mx-auto">
                    <img class="w-full" src="{{product.pictureUrl}}" alt="product picture">
                </div>
                <div>
                    <h1 class="text-4xl font-semibold text-blue-900">{{product.name}}</h1>
                    <div class="mt-4 items-center gap-4 flex">
                        <p class="text-3xl font-extrabold text-gray-900">
                            {{product.price|currency:'EUR':'symbol':'1.2-2':'fr'}}
                        </p>
                    </div>
                    <div class="flex gap-3 mt-6">
                        <button class="button gap-2.5 font-semibold" mat-flat-button >
                            <mat-icon>shopping_cart</mat-icon>
                            Add to cart
                        </button>
                        <mat-form-field appearance="outline" class="flex">
                            <mat-label>Insert Quantity</mat-label>
                            <input matInput type="number">
                        </mat-form-field>
                        <button class="text-black font-medium text-2xl">
                            <a>In Stock: <a class=" text-emerald-500 ">{{product.quantityInStock}}</a></a>
                        </button>
                    </div>
                    <mat-divider></mat-divider>
                    <p class="mt-3 text-gray-500 text-2xl">
                    {{product.description}}
                    </p>
                </div>
            </div>
        </div>
    </section>
}
```

Εικόνα 226

Το μέγεθος του *mat-icon shoping_cart*, καθώς και το χρώμα του συγκεκριμένου κουμπιού, καθορίζονται στο αρχείο *product-details.scss* (εικόνα 227).

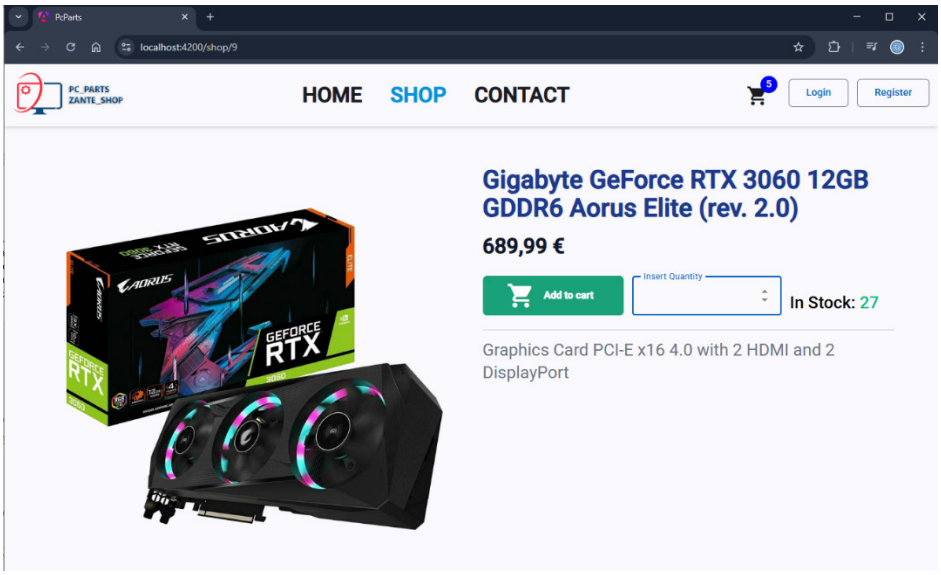
product-details.component.scss ×

client > src > app > features > product-details > product-details.component.scss > ...

```
.mat-icon{ //cart icon
    font-size: 40px;
    width: 35px;
}

.button{
    width: 200px;
    height: 56px;
    background-color: rgba(12, 156, 113, 0.945)//add to cart button
}
```

Εικόνα 227

Εκτελώντας την εφαρμογή μας κι επιλέγοντας ένα προϊόν, μεταφερόμαστε στη σελίδα προϊόντος με το αποτέλεσμα του *browser* να παρουσιάζεται στην εικόνα 228.



Εικόνα 228

7.4 Διαμόρφωση της Αρχικής Σελίδας

Βελτιώνοντας την εμφάνιση της αρχικής σελίδας της εφαρμογής μας, θα προσθέσουμε τον *html* κώδικα της εικόνας 229, εντός του *home.component.html* αρχείου. Πρακτικά, προσθέτουμε μία εικόνα υποδοχής τύπου *wallpaper*, καθώς κι ένα κουμπί-σύνδεσμο για την οθόνη *Shop* του ηλεκτρονικού μας καταστήματος. Χρησιμοποιώντας τα *Angular attributes mask-radial-from-neutral-200* και *bg-linear-to-r*, δίνουμε χρωματικά οπτικά εφέ στα κείμενα *«Welcome to Pcparts!»* και *«Go to shop».*

home.component.html

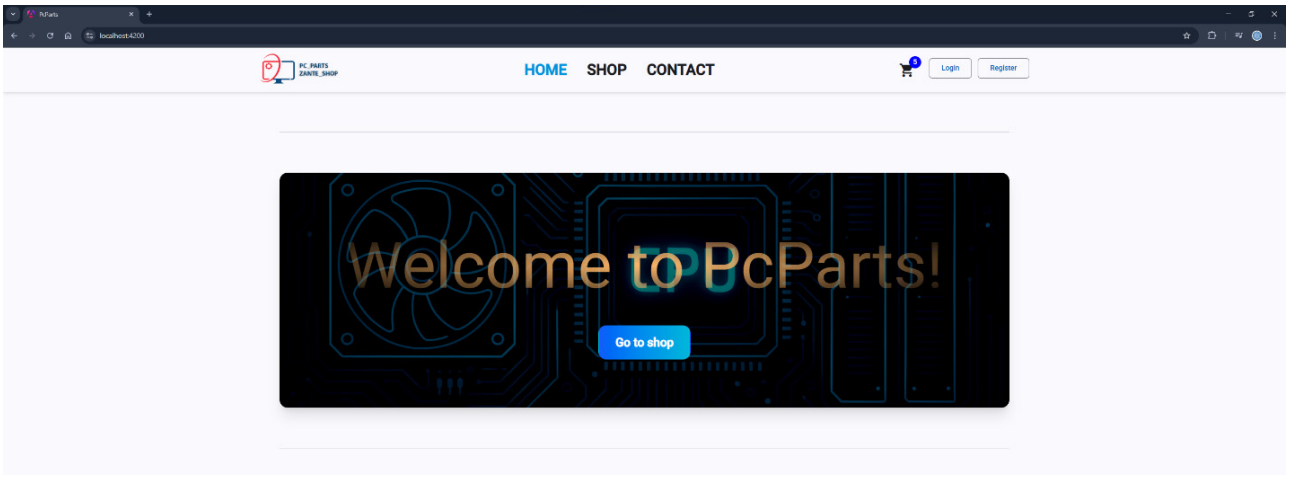
client > src > app > features > home > home.component.html > ...

Go to component

```
<div class="max-w-screen-2xl mx-auto px-4 mt-20">
    <mat-divider></mat-divider>
    <div class="flex flex-col items-center py-16 justify-center mt-20 rounded-2xl
        shadow-xl relative">
        <img class="absolute inset-0 w-full h-full object-cover rounded-2xl"
            src="../images/homescreenbg.png" alt="ai pcparts generated image">
        <div class="flex flex-col p-8 rounded-2xl items-center relative">
            <h1 class="my-6 mask-radial-from-neutral-200 text-orange-400 text-9xl">
                Welcome to PcParts!
            </h1>
            <button class="bg-linear-to-r from-orange-900 to-orange-500 font-semibold
                    text-2xl text-white
                    rounded-2xl px-8 py-4 border-2 border-transparent mt-8"
                routerLink="/shop">
                Go to shop
            </button>
        </div>
    </div>
    <div>
        <mat-divider class="mt-20"></mat-divider>
    </div>
</div>
```

Εικόνα 229

Στην εικόνα 230, παρουσιάζεται η αρχική οθόνη της *web* εφαρμογής μας.



Εικόνα 230

7.5 Διαμόρφωση της Σελίδας Επικοινωνίας

Το τελευταίο *component* προς διαμόρφωση με βάση τα κουμπιά – *links*, *Home*, *Shop* και *Contact*, είναι αυτό του *contact-details*. Ανοίγοντας ένα παράθυρο τερματικού, εκτελούμε την εντολή *ng g c features/contact-details --skip-tests* και δημιουργείται ένα νέο *component* με τίτλο *contact-details*, εντός του καταλόγου *features*. Στην εικόνα 231 παρουσιάζεται η προσθήκη ενός ακόμη *path* εντός του αρχείου *app.routes.ts (contact)*, το οποίο πατώντας τον σύνδεσμο *Contact* εντός του φυλλομετρητή μας, ανακατευθύνει τον χρήστη μέσω του *URL*: https://localhost:4200/contact, στην οθόνη του *contact-details.html template*.

app.routes.ts

client > src > app > app.routes.ts > ...

```
1  import { Routes } from '@angular/router';
2  import { HomeComponent } from './features/home/home.component';
3  import { ShopComponent } from './features/shop/shop.component';
4  import { ProductDetailsComponent } from './features/product-details/product-details.component';
5  import { ContactDetailsComponent } from './features/contact-details/contact-details.component';
6
7  export const routes: Routes = [
8      {path:'', component: HomeComponent},
9      {path:'shop', component: ShopComponent},
10     {path:'shop/:id', component: ProductDetailsComponent},
11     {path:'contact', component: ContactDetailsComponent},
12     {path:'**', redirectTo:'', pathMatch:'full'}
13 ];
```

Εικόνα 231

Ακολούθως, ανοίγουμε το *template contact-details.html* αρχείο και προσθέτουμε τον κώδικα της εικόνας 232. Στην ουσία δημιουργούμε μία απλή φόρμα *html*, χρησιμοποιώντας τα *attributes mat-icons*, *label*, *input* και *form* και παρουσιάζουμε στον χρήστη όλες τις απαραίτητες πληροφορίες για να επικοινωνεί με το διαδικτυακό μας κατάστημα. Η λειτουργικότητα της συγκεκριμένης σελίδας είναι καθαρά αισθητική και δεν προσφέρει κάποια νέα δυνατότητα στην *Angular* εφαρμογή μας.

contact-details.component.html

client > src > app > features > contact-details > contact-details.component.html > ...

Go to component

```
<!DOCTYPE html>
<html lang="en">
<div class="flex align-middle items-center justify-between">
            <label class="text-2xl flex align-middle"><mat-icon class="material-symbols-outlined">location_on</mat-icon>
             Zakynthos Island</label>
             <label class="text-2xl flex align-middle"><mat-icon class="material-symbols-outlined">add_road</mat-icon>
             Vanato Street</label>
             <label class="text-2xl flex align-middle"><mat-icon class="material-symbols-outlined">phone</mat-icon>
             26950 26950</label>
             <label class="text-2xl flex align-middle"><mat-icon class="material-symbols-outlined">mail</mat-icon>
             contact@pcparts.com
             </label>
</div>
<body class="pb-85">
    <div class="contact-form-container">
        <h2 class="text-5xl align-middle font-bold text-blue-600 ">Contact Us</h2>
        <form id="contactForm">
            <div class="form-group">
                <label for="name">Name</label>
                <input  type="text" id="name"
                       name="name"
                       placeholder="Your Name" required>
                <span class="error-message" id="nameError"></span>
            </div>
            <div class="form-group">
                <label for="email">Email</label>
                <input type="email" id="email"
                       name="email"
                       placeholder="Your Email" required>
                <span class="error-message" id="emailError"></span>
            </div>
            <div class="form-group">
                <label for="phone">Phone</label>
                <input type="tel" id="phone"
                       name="phone"
                       placeholder="Your Phone Number" required>
                <span class="error-message" id="phoneError"></span>
            </div>
            <div class="form-group">
                <label for="message">Message</label>
                <textarea id="message"
                          name="message"
                          placeholder="Your Message"
                          rows="5" required></textarea>
                <span class="error-message"
                     id="messageError"></span>
            </div>
            <button type="submit"
                    class="submit-button text-2xl"  routerLink="/">
              Send Message
            </button>
        </form>
    </div>
    <script src="scripts.js"></script>
</body>
</html>
```

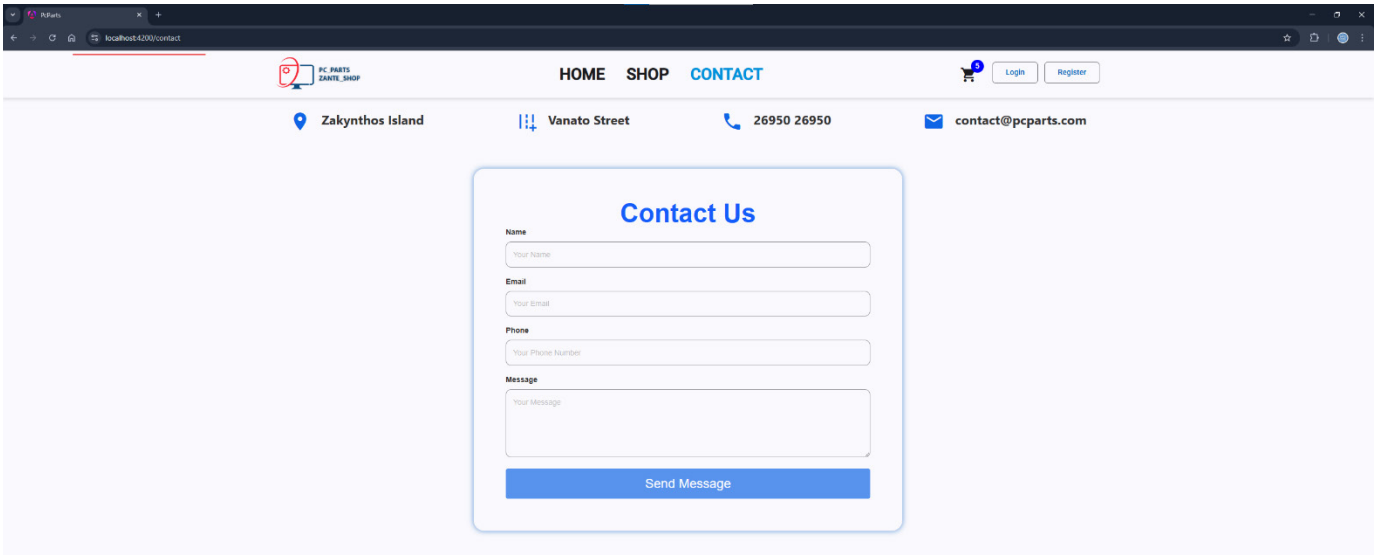
Εικόνα 232

Εντός του αρχείου *header.component.html*, προσθέτουμε τα *attributes routerLink* και *routerlinkActive* προκειμένου να δημιουργηθεί το *link* προς τη σελίδα *Contact* και να προστεθεί ο χρωματισμός κατά την επιλογή του αντίστοιχου κουμπιού (εικόνα 233).

```
<nav class="flex gap-x-10 my-2 uppercase text-3xl font-bold">
    <a routerLink="/"
    routerLinkActive="active"
    [routerLinkActiveOptions]="{exact:true}"
    >Home</a>
    <a routerLink="/shop"
    routerLinkActive="active">
    Shop</a>
    <a routerLink="/contact"
    routerLinkActive="active">
    Contact</a>
</nav>
```

Εικόνα 233

Το περιεχόμενο της σελίδας *Contact* (https://localhost:4200/contact), το ενεργό κουμπί-*Link* καθώς και ο σύνδεσμος που ανακατευθύνει τον χρήστη, παρουσιάζονται στην εικόνα 234.



Εικόνα 234

Κεφάλαιο 8. Καλάθι Αγορών – .Net Project

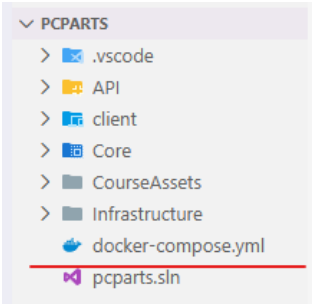
Κάθε διαδικτυακή εφαρμογή της μορφής «*e-commerce*», παρέχει στον χρήστη τη δυνατότητα προσθήκης προϊόντων σε ένα ηλεκτρονικό καλάθι αγορών. Ο χρήστης έχει τη δυνατότητα να πλοηγείται στις διαφορετικές σελίδες προϊόντων, να επιλέγει οποιοδήποτε προϊόν είναι διαθέσιμο για αγορά και να το προσθέτει στο συγκεκριμένο καλάθι.

8.1 Οι εφαρμογές Docker και Redis Server

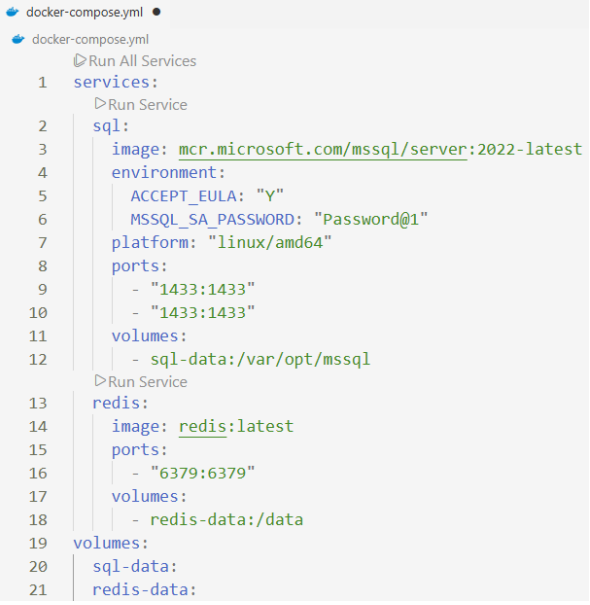
Στην εφαρμογή PcParts, θα χρησιμοποιήσουμε τον *Remote Dictionary Server - Redis* για να αποθηκεύουμε προσωρινά το καλάθι αγορών του εκάστοτε χρήστη. Ο *Redis* αποτελεί ιδανική επιλογή προσωρινής αποθήκευσης *Key/Value* τιμών στην προσωρινή μνήμη του υπολογιστή και αποτελεί μία μορφή *quick-response* βάσης δεδομένων (εικόνα 235). Επίσης, παράλληλα με τη δημιουργία του *pcparts-redis container*, θα δημιουργήσουμε και το *pcparts-sql container*, όπου θα μεταφέρουμε τη βάση δεδομένων της εφαρμογής μας σε ένα ενιαίο *container* με τίτλο *pcparts*. Για να συμβούν όλα τα παραπάνω, θα πρέπει να κατεβάσουμε την εφαρμογή *Docker* (*personal version*) από το δίκτυο. Η συγκεκριμένη εφαρμογή δύναται να φιλοξενεί *SQL* βάσεις δεδομένων και μας παρέχει τη δυνατότητα διαχείρισής τους από το *Docker CLI*, το οποίο και θα προσθέσουμε στο περιβάλλον του *Visual Studio Code*.

Εικόνα 235

Εφόσον εγκαταστήσουμε την εφαρμογή *Docker* στον υπολογιστή μας, εντός του *pcparts project* δημιουργούμε το αρχείο *docker-compose.yml* (εικόνα 236). Εκεί, ορίζουμε τις *Docker services* με όλες τις απαραίτητες παραμέτρους (*platform, DB volumes, password, ports, image name*), ώστε να έχουμε τη δυνατότητα επικοινωνίας της *.Net* εφαρμογής μας από το περιβάλλον ανάπτυξης του *Visual Studio Code*, με το περιβάλλον της εφαρμογής *Docker* (εικόνα 237).

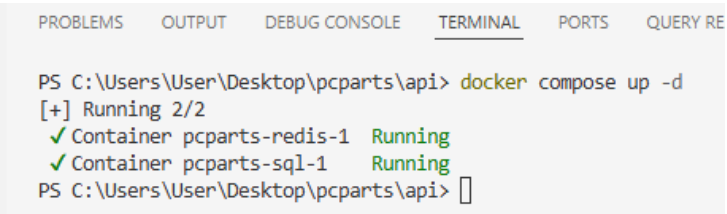


Εικόνα 236

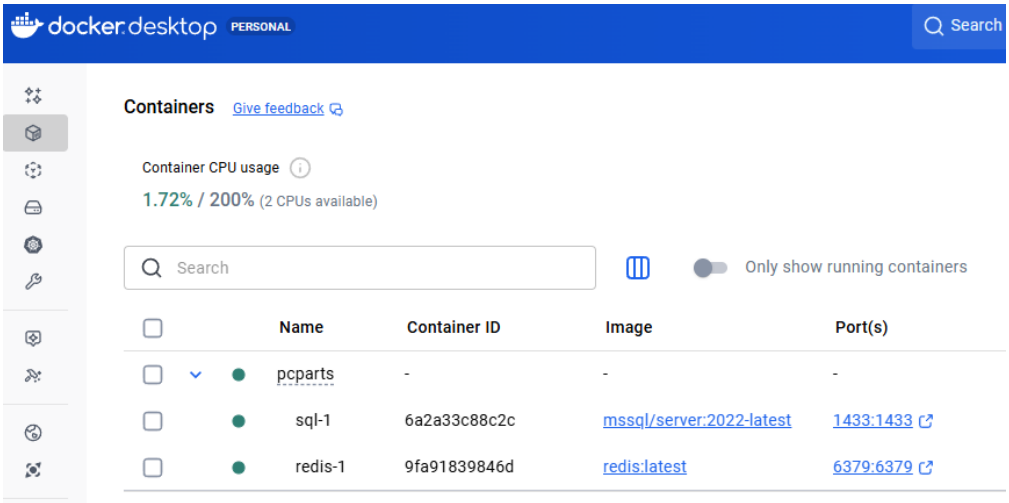


Εικόνα 237

Σε ένα παράθυρο τερματικού εντός του *Visual Studio Code*, εκτελούμε την εντολή *docker compose up – d* (εικόνα 238) και δημιουργείται ένα κυρίως *container* με τίτλο *pcparts* στην εφαρμογή *Docker*, το οποίο περιέχει ένα κενό *instance* μίας *SQL* βάσης δεδομένων με τίτλο *sql-1*, καθώς και το *redis instance* με την ονομασία *redis-1* (εικόνα 239).

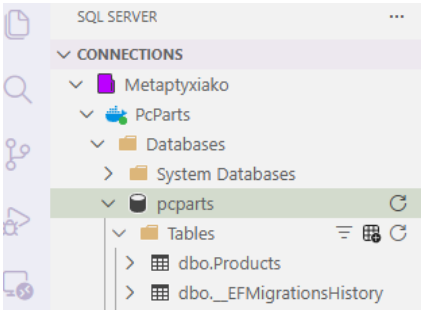


Εικόνα 238



Εικόνα 239

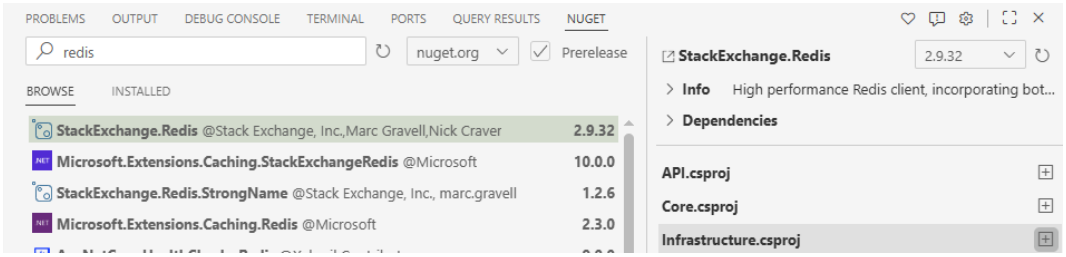
Εκτελώντας την εντολή *dotnet watch*, μέσα από τον κατάλογο API, εφαρμόζεται μία ακόμη dotnet ef migration και η βάση δεδομένων της εφαρμογής μας δημιουργείται εκ νέου. Στο παράθυρο διαχείρισης των SQL συνδέσεων του Visula Studio Code, παρατηρούμε την επιτυχή σύνδεση με τη βάση δεδομένων pcparts, καθώς και τη δημιουργία του πίνακα με όλη την πληροφορία που έχει να κάνει με τα προϊόντα (εικόνα 240).



Εικόνα 240

8.2 Χρήση του Redis Server εντός του .Net Project

Θέλοντας να δημιουργήσουμε, αλλά και να διασφαλίσουμε την απρόσκοπτη επικοινωνία της *.Net* εφαρμογής μας με τον *Redis server*, θα εγκαταστήσουμε το *Nuget Package stack.echange.redis* στο *Infastructure project* (εικόνα 241).



Εικόνα 241

Στη συνέχεια, θα μεταφερθούμε στο περιβάλλον του *API project* για να ορίσουμε μία νέα *.Net singleton service* εντός του αρχείου *Program.cs* (εικόνα 242). Κάθε φορά που εκκινεί η εφαρμογή μας θα χρησιμοποιούμε το ίδιο *instance* της συγκεκριμένης υπηρεσίας και μέσω του *.Net interface IConnectionMultiplexer* περνάμε ένα συγκεκριμένο μοτίβο παραμετροποίησης. Αυτό το μοτίβο περιλαμβάνει την αποθήκευση του *connection string* στη μεταβλητή *connString* κι εφόσον δεν παρουσιάζονται σφάλματα, μέσω των κλάσεων *ConfigurationOptions* και *ConnectionMultiplexer*, οι οποίες περιλαμβάνονται στο *StackExchange.Redis namespace*, μας επιστρέφεται το τελικό *Redis connection string*.

```
17  builder.Services.AddSingleton<IConnectionMultiplexer>(config =>
18  {
19      var connectionString = builder.Configuration.GetConnectionString("Redis")
20          ?? throw new Exception("Cannot get redis connection string");
21      var configuation = ConfigurationOptions.Parse(connectionString, true);
22      return ConnectionMultiplexer.Connect(configuation);
23  });
```

Εικόνα 242

Το *Redis connection string* το ορίζουμε εντός του αρχείου *appsettings.development.json*, όπως παρουσιάζεται στην εικόνα 243.

```
"ConnectionStrings": {
  "DefaultConnection":"Server=localhost,1433;Database=pcparts;User Id=SA;Password=Password@1;TrustServerCertificate=True",
  "Redis":"localhost"
}
```

Εικόνα 243

8.3 Δημιουργία των Shopping Cart Κλάσεων

Προχωρώντας με την παραμετροποίηση του καλαθιού αγορών εντός του *Core project* και συγκεκριμένα εντός του καταλόγου *Entities*, θα δημιουργήσουμε δύο νέες *C#* κλάσεις. Η πρώτη ονομάζεται *ShoppingCart* και περιλαμβάνει τις οντότητες *Id* και *Items*. Η *property Items* αποτελεί μία κενή λίστα τύπου *CartItem*, τύπο τον οποίο ορίζουμε αμέσως μετά. Οι οντότητες αυτές θα προσδιορίζουν το εκάστοτε καλάθι χρήστη εντός του *Redis server* (εικόνα 244Α).

ShoppingCart.cs 1 ×

Core > Entities > ShoppingCart.cs > ...

```
using System;

namespace Core.Entities;

0 references
public class ShoppingCart
{
    0 references
    public required string Id { get; set; }
    0 references
    public List<CartItem> Items { get; set; } = [];
}
```

Εικόνα 244Α

Η δεύτερη κλάση ονομάζεται *CartItem* και περιέχει όλες τις οντότητες που περιγράφουν κάθε προϊόν που βρίσκεται στη βάσης δεδομένων μας (εικόνα 244Β). Οι οντότητες αυτές θα χρησιμοποιηθούν για την άντληση της πληροφορίας, σχετικά με το ποια προϊόντα έχει προσθέσει ο χρήστης στο ψηφιακό καλάθι αγορών του.

CartItem.cs ×

Core > Entities > CartItem.cs > ...

```
using System;

namespace Core.Entities;

1 reference
public class CartItem
{
    0 references
    public int ProductId { get; set; }
    0 references
    public required string ProductName { get; set; }
    0 references
    public decimal Price { get; set; }
    0 references
    public int Quantity { get; set; }
    0 references
    public required string PictureUrl { get; set; }
    0 references
    public required string Brand { get; set; }
    0 references
    public required string Type { get; set; }
}
```

Εικόνα 244Β

8.4 Δημιουργία της Shopping Cart Service

Σειρά έχει η δημιουργία μίας *Interface* κλάσης εντός του καταλόγου *Interfaces*, με στόχο την αποστολή και άντληση δεδομένων από και προς το ηλεκτρονικό καλάθι αγορών μας, το οποίο φιλοξενείται στον *Redis server*. Η συγκεκριμένη κλάση ονομάζεται *ICartService* και περιέχει τις μεθόδους *GetCartAsync* *SetCartAsync* και *DeleteCartAsync* (εικόνας 245).

ICartService.cs

Core > Interfaces > ICartService.cs > ...

```
1  using System;
2  using Core.Entities;
3
4  namespace Core.Interfaces;
5
   0 references
6  public interface ICartService
7  {
       0 references
8      Task<ShoppingCart?> GetCartAsync(string key);
       0 references
9      Task<ShoppingCart?> SetCartAsync(ShoppingCart cart);
       0 references
10     Task<bool> DeleteCartAsync(string key);
11 }
```

Εικόνα 245

Στη συνέχεια, εντός του *Infrastructure project* δημιουργούμε έναν νέο κατάλογο δίνοντάς του το όνομα *Services*. Εκεί ορίζουμε την *implementation C#* κλάση *CartService*. Εκτός της χρησιμοποίησης του *ICartService interface*, η συγκεκριμένη κλάση δέχεται ως παράμετρο (*Injects*) την κλάση *IConnectionMultiplexer*, εφόσον θα επικοινωνεί με τον *Redis server*. Αρχικά, ορίζουμε τη μεταβλητή τύπου *IDatabase* με όνομα *_database*, στην οποία αποθηκεύουμε το *connection path* προς τον *Redis server*. Ακολούθως, εντός της μεθόδου *DeleteCartAsync*, βασιζόμενη στο δεδομένο κλειδί (*key*) καλαθιού, χρησιμοποιούμε τη μέθοδο *KeyDeleteAsync* για να διαγράψουμε τη συγκεκριμένη εγγραφή καλαθιού από τον *Redis server* (εφόσον υπάρχει). Προχωρώντας στη μέθοδο *GetCartAsync*, βασιζόμενοι εκ νέου στο δεδομένο κλειδί εγγραφής (εφόσον υπάρχει) μέσω του *JsonSerializer*, λαμβάνουμε τις πληροφορίες που περιγράφουν τη συγκεκριμένη εγγραφή καλαθιού εντός του *Redis server*. Τέλος, στη μέθοδο *SetCartAsync* αναθέτουμε την εισαγωγή μίας νέας εγγραφής καλαθιού στον *Redis server*, η οποία παραμένει ενεργή για διάστημα 30 ημερών. Μετά το πέρας των 30 ημερών διαγράφεται αυτόματα από τη βάση δεδομένων *Redis*. Στις εικόνες 246 και 247 παρουσιάζεται ο κώδικας της κλάσης *CartService*.

```
CartService.cs ●
Infrastructure > Services > CartService.cs > CartService > CartService
1  using System.Text.Json;
2  using Core.Entities;
3  using Core.Interfaces;
4  using StackExchange.Redis;
5
6  namespace Infrastructure.Services;
7
8  //implements ICart service methods
9  //injects the IConnectionMultiplexer class to get in touch with Redis server
   0 references
10 public class CartService(IConnectionMultiplexer redis) : ICartService
11 {
12     //connects with redis
       3 references
13     private readonly IDatabase _database = redis.GetDatabase();
       1 reference
14     public async Task<bool> DeleteCartAsync(string key)
15     {
16         //delete specified (by key) cart from Redis if exists
17         return await _database.KeyDeleteAsync(key);
18     }
```

Εικόνα 246

```
20     public async Task<ShoppingCart?> GetCartAsync(string key)
21     {
22         //retrieves specified cart's data from Redis if exists
23         var data = await _database.StringGetAsync(key);
24
25         return data.IsNullOrEmpty ? null : JsonSerializer.Deserialize<ShoppingCart>(data!);
26     }
27
       1 reference
28     public async Task<ShoppingCart?> SetCartAsync(ShoppingCart cart)
29     {
30         //creates a new cart intro in Redis
31         var created = await _database.StringSetAsync(cart.Id,
32             JsonSerializer.Serialize(cart), TimeSpan.FromDays(30));
33
34         if (!created) return null;
35
36         return await GetCartAsync(cart.Id);
37     }
38 }
```

Εικόνα 247

Εφόσον τις συγκεκριμένες κλάσεις τις διαχειριζόμαστε ως *.Net services* βασισμένες στην *Interface* κλάση *ICartService*, θα πρέπει να μεταβούμε στο αρχείο *program.cs* και να δηλώσουμε τη νέα *singleton service* που της αφορά (εικόνα 248).

```
33 });
34 //cartservice
35 builder.Services.AddSingleton<ICartService,CartService>();
36
```

Εικόνα 248

8.5 Δημιουργία του Shopping Cart Controller

Έχοντας δημιουργήσει τις απαραίτητες κλάσεις για τη λειτουργία του καλαθιού αγορών εντός της *.Net* εφαρμογής μας, θα πρέπει να δημιουργήσουμε και τα απαραίτητα *API endpoints* για να εκμεταλλευτούμε τη συγκεκριμένη προσθήκη. Για να συμβεί κάτι τέτοιο, θα μεταβούμε στο *API project* κι εντός του καταλόγου *Controllers*, θα δημιουργήσουμε μία νέα κλάση τύπου *Controller* με όνομα *CartController*, η οποία χρησιμοποιεί τις οντότητες της κλάσης *BaseApiController* και κάνει *inject* την *Shopping Cart service*.

Εντός της συγκεκριμένης κλάσης ορίζουμε τρία *endpoints*, ένα για κάθε ενέργεια που αφορά τη δημιουργία, τη διαγραφή και την περιγραφή ενός καλαθιού στον *Redis server*. Στις εικόνες 249 και 250 παρουσιάζεται ο κώδικας του νέου *controller*.

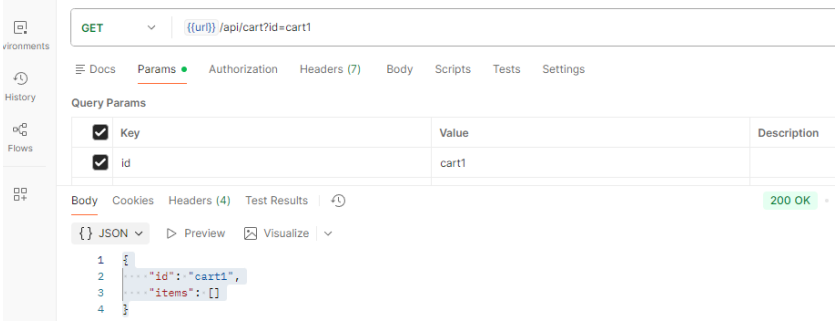
```
5   namespace API.Controllers;
6
7   //injection of cartservice
8   //derives from BaseApiController
    0 references
9   public class CartController(ICartService cartService) : BaseApiController
10  {
11      [HttpGet] //gets or if not exist insert new cart id into redis server db
        0 references
12      public async Task<ActionResult<ShoppingCart>> GetCartById(string id)
13      {
14          var cart = await cartService.GetCartAsync(id);
15
16          return Ok(cart ?? new ShoppingCart{Id = id});
17      }
```

Εικόνα 249

```
19      [HttpPost] //insert cart info into redis db specific cart key
        0 references
20      public async Task<ActionResult<ShoppingCart>> UpdateCart(ShoppingCart cart)
21      {
22          var updatedCart = await cartService.SetCartAsync(cart);
23
24          return Ok(updatedCart);
25      }
26
27      [HttpDelete] //delete a cart key row from redis server db baed on given id
        0 references
28      public async Task DeleteCart(string id)
29      {
30          await cartService.DeleteCartAsync(id);
31      }
32  }
```

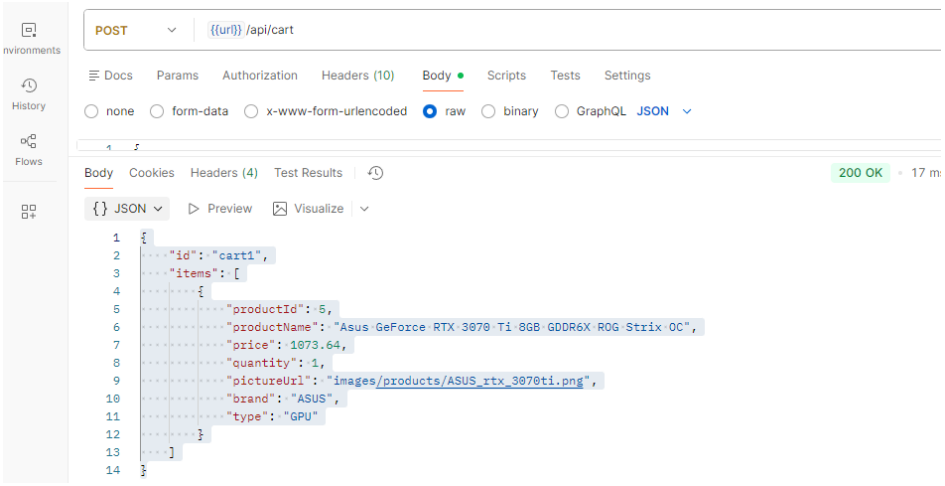
Εικόνα 250

Θέλοντας να δοκιμάσουμε τη λειτουργικότητα των *API endpoints* εντός του *CartController*, θα μεταβούμε στο περιβάλλον της εφαρμογής *Postman*. Αρχικά θα εκτελέσουμε ένα *HTTP get request*, αναζητώντας το «καλάθι» με *id* που ισούται με το *string cart1*. Από τη στιγμή που δεν υπάρχει διαθέσιμο στην *Redis* βάση δεδομένων, θα εισαχθεί ως νέα εγγραφή και θα μας επιστραφεί ως κενή εγγραφή (εικόνα 251).



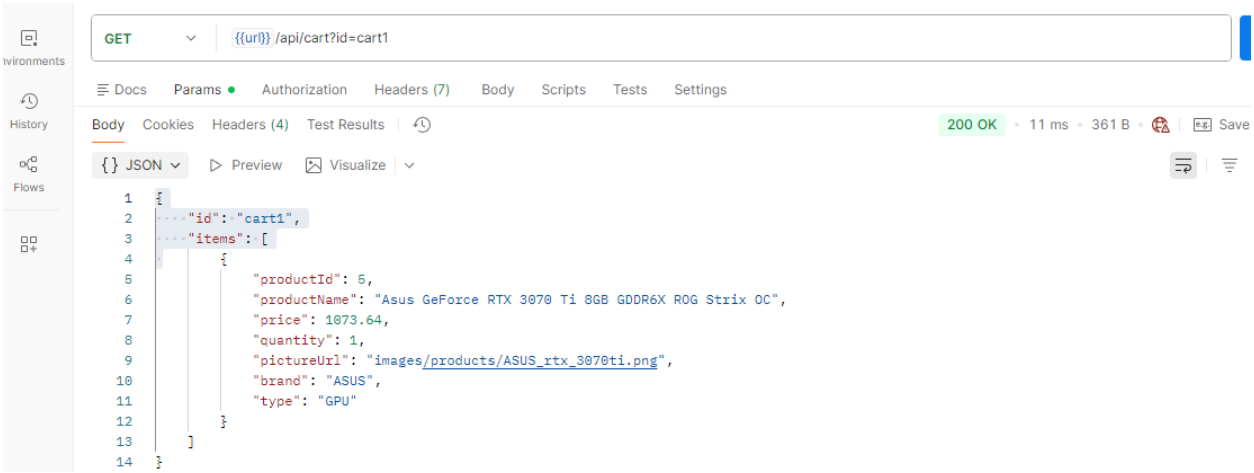
Εικόνα 251

Εν συνεχεία θα εκτελέσουμε ένα *HTTP* post αίτημα προς το καλάθι με *id* που ισούται εκ νέου με *cart1*. Θέτοντας χειροκίνητα όλη την πληροφορία εντός της *item property* στο περιβάλλον του *Postman*, το καλάθι αποκτά ένα προϊόν και λαμβάνουμε το αποτέλεσμα της εικόνας 252.



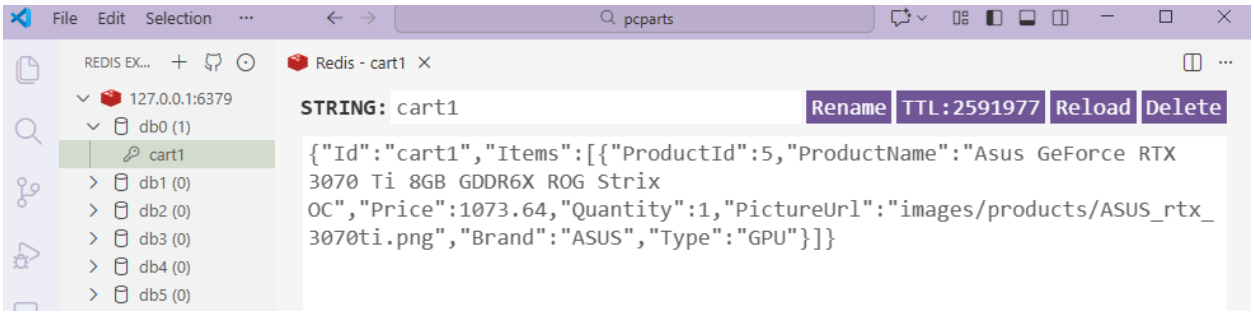
Εικόνα 252

Δοκιμάζοντας εκ νέου το *HTTP get* αίτημα της εικόνας 251, διαπιστώνουμε πως έχει προστεθεί ένα προϊόν εντός του καλαθιού με *id* που ισούται με το αλφαριθμητικό *cart1* (εικόνα 253).



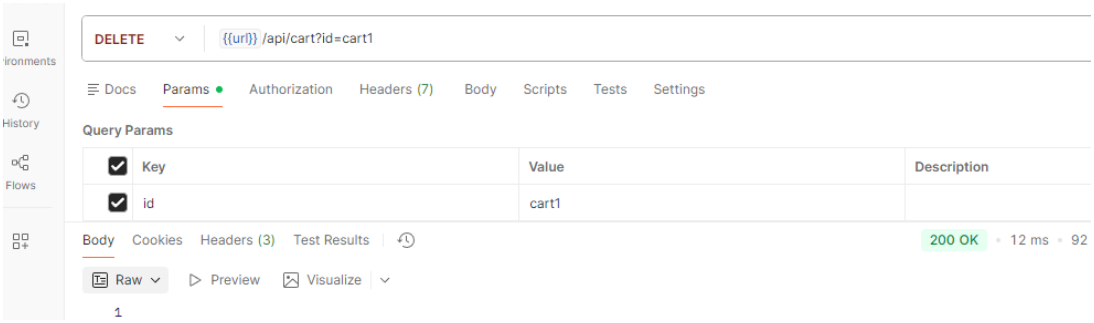
Εικόνα 253

Στο περιβάλλον του *Visual Studio Code*, θα προσθέσουμε την επέκταση *Redis Explorer*. Κατά την εκτέλεσή της παρατηρούμε πως στη θέση db0, έχει εισαχθεί όλη η πληροφορία που αφορά το καλάθι με *id* που ισούται με *cart1*, συμπεριλαμβανομένης της πληροφορίας *TTL (time to live)*, η οποία υποδεικνύει το χρονικό διάστημα μέχρι την αυτόματη διαγραφή της συγκεκριμένης εγγραφής (εικόνα 254).

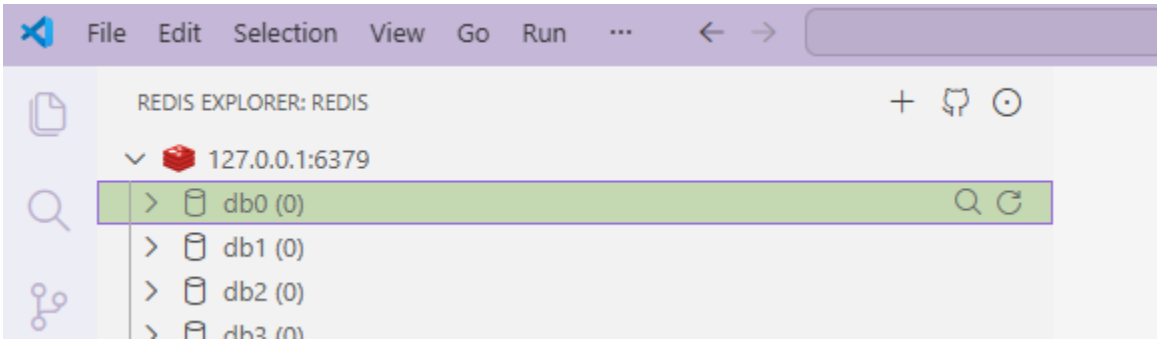


Εικόνα 254

Τέλος, εκτελώντας το *HTTP delete* αίτημα για το καλάθι αγορών με *id* που ισούται με *cart1*, λαμβάνουμε το αποτέλεσμα της εικόνας 255, το οποίο είναι το 200 OK. Στην εικόνα 256, παρουσιάζεται το αντίστοιχο αποτέλεσμα στο περιβάλλον του *Redis explorer*. Η εγγραφή καλαθιού *cart1* έχει διαγραφεί οριστικά.



Εικόνα 255



Εικόνα 256

Κεφάλαιο 9. Καλάθι Αγορών – Angular Project

Στο τρέχων κεφάλαιο θα συνεχίσουμε την ανάπτυξη του ηλεκτρονικού καλαθιού αγορών, καθώς μεταφερόμαστε στο περιβάλλον του *Angular project*. Θα εξετάσουμε τεχνικές όπως τα *Angular Signals*, θα βελτιώσουμε τα *hardcoded urls* εντός της εφαρμογής μας χρησιμοποιώντας *environmental variables*, καθώς επίσης θα ολοκληρώσουμε την αλληλεπίδραση χρήστη – διεπαφής – *Angular* εφαρμογής, αναφορικά με τη διαδικασία προσθήκης προϊόντων στο καλάθι αγορών.

9.1 Δημιουργία των Cart Components

Ξεκινώντας από τον κατάλογο *Core*, εντός του *Angularp Project*, εκτελούμε την εντολή *ng g s core/services/cart --skip-tests* στο τερματικό, προκειμένου να δημιουργήσουμε μία νέα *Angular service* με όνομα *cart*, εντός του υποφάκελου *services*. Ακολούθως, εκτελώντας την εντολή *ng g c features/cart --skip-tests* , δημιουργούμε το *cart component* εντός του καταλόγου *features*. Στη συνέχεια, ανοίγουμε το αρχείο *app.routes.ts* και προσθέτουμε ένα ακόμη *path*, αναφερόμενο στο νέο *cart component* (εικόνα 257).

```
19    {path:'cart', component: CartComponent},
20    {path:'**', redirectTo:'not-found', pathMatch:'full'}
21  ];
```

Εικόνα 257

Ακολούθως, ανοίγουμε το *html template header.component.html*, όπου κάνοντας χρήση των *attributes routerlink* και *routerlinkActive*, εντός του *<a> element* που περιλαμβάνει το *shopping_cart* εικονίδιο, το μετατρέπουμε σε σύνδεσμο που μας οδηγεί στο *cart.component.html template* (εικόνα 258).

```
<a routerLink="/cart" routerLinkActive="active" matBadge="5"
   matBadgeSize="large" class="custom-badge mt-2 mr-4">
   <mat-icon>shopping_cart</mat-icon>
</a>
```

Εικόνα 258

Ανοίγοντας ένα παράθυρο τερματικού, εκτελούμε την εντολή *npm install nanoid*. Με αυτήν μας την κίνηση εγκαθιστούμε την εφαρμογή *Nano ID* (*Random Number Generator Package*), η οποία θα μας βοηθήσει στο να αποδίδουμε τυχαίους αριθμούς/κλειδιά (*ids*), στις διάφορες παραγγελίες που εισάγονται στον *Redis server*. Επόμενο βήμα αποτελεί η δημιουργία ενός αρχείου – μοντέλου, το οποίο θα μας δίνει τη δυνατότητα να κάνουμε αναπαραγωγή (*replication*) των οντοτήτων που περιλαμβάνονται σε όλες τις κλάσεις του *API project* και αφορούν τη λειτουργία του καλαθιού αγορών. Το αρχείο αυτό το ονομάζουμε *cart.ts*, το τοποθετούμε εντός του καταλόγου *models* και ο *TypeScript* κώδικας που το περιγράφει παρουσιάζεται στην εικόνα 259.

cart.ts

client > src > app > shared > models > cart.ts > ...

```ts
import {nanoid} from "nanoid";
//api cart properties replication
export type CartType = {
  id: string;
  items: CartItem[];
}

export type CartItem = {
  productId: number;
  productName: string;
  price: number;
  quantity: number;
  pictureUrl: string;
  brand: string;
  type: string;
}

export class Cart implements CartType {
  id = nanoid(); //random numbers generator for cart id
  items: CartItem[] = [];
}
```

Εικόνα 259

9.2 Angular Signals

Τα *Angular Signals* αποτελούν ένα νέο διαδραστικό μοντέλο, το οποίο επιτρέπει στους προγραμματιστές να δημιουργούν εφαρμογές παρακολουθώντας και αντιδρώντας σε αλλαγές δεδομένων με λεπτή ακρίβεια. Βελτιώνουν την απόδοση της εφαρμογής επιτρέποντας στην *Angular* να ενημερώνει μόνο τα μέρη του κώδικα στα *components* που έχουν αλλάξει, αντί να ελέγχει ολόκληρο το δέντρο των *components*. Με απλά λόγια, ένα *Angular Signal* είναι μια τιμή που ειδοποιεί μόνο τα συνδεόμενα *components* σε αυτήν, κάθε φορά που αλλάζει το περιεχόμενό της. Στην περίπτωση της εφαρμογής *PcParts* και συγκεκριμένα για το *Angular project*, εμείς θα εκμεταλλευτούμε τα *Angular Signals* για να αποθηκεύουμε την πληροφορία που μας αποστέλλει ο *Redis server* αναφορικά με τα καλάθια αγορών, στην προσωρινή μνήμη της *client* εφαρμογής μας. Ξεκινώντας, θα εκτελέσουμε την εντολή *ng g environments* στο τερματικό ώστε να δημιουργηθούν τα *TypeScript* αρχεία της εικόνας 260.

Εικόνα 260

Εντός του αρχείου *environment.development.ts*, εισάγουμε τον κώδικα της εικόνας 261.

environment.development.ts ●

client > src > environments > environment.development.ts > ...

```ts
//replace the hardcoded URLs inside compo
export const environment = {
  production: false,
  baseUrl: 'https://localhost:5001/api/'
};
```

Εικόνα 261

Με αυτόν τον τρόπο αναθέτουμε το *hardcoded URL*, που χρησιμοποιούμε εντός των *Angular components*, σε μία μεταβλητή. Στη συνέχεια ανοίγουμε το *cart.service.ts* αρχείο και προσθέτουμε τη συγκεκριμένη μεταβλητή, προσδιορίζοντας ότι βρισκόμαστε στο στάδιο ανάπτυξης της εφαρμογής. Στη συνέχεια κάνουμε *inject* τη μέθοδο *HttpClient* κι ορίζουμε τη μεταβλητή *cart* ως *Angular Signal*, το οποίο μας παρέχεται από την *Angular Core* βιβλιοθήκη. Το σήμα (*signal*) αυτό μας επιστρέφει δεδομένα τύπου *Cart* ή την τιμή *null*. Ακολούθως, δημιουργούμε τις μεθόδους *getCart* και *setCart* εντός της κλάσης *CartService*. Εκμεταλλευόμενοι την *Angular Signal* μεταβλητή *Cart*, το *pipe object* ώστε να παρακολουθούμε την πορεία της *signal* μεταβλητής μας, αλλά και τις *Angular HTTP* μεθόδους *get* και *post*, η υπηρεσία *CartService* αλλά και οποιοδήποτε άλλο *component* το αιτηθεί, μπορεί να θέτει και να λαμβάνει ένα νέο καλάθι ευρισκόμενο εντός του *Redis server* (εικόνα 262).

cart.service.ts 1 ●

client > src > app > core > services > cart.service.ts > CartService

```ts
import { computed, inject, Injectable, signal } from '@angular/core';
import { environment } from '../../../environments/environment';
import { HttpClient } from '@angular/common/http';
import { Cart, CartItem } from '../../shared/models/cart';
import { Product } from '../../shared/models/product';
import { map } from 'rxjs';


@Injectable({
  providedIn: 'root'
})

export class CartService {
  baseUrl = environment.baseUrl;
  private http = inject(HttpClient); //inject Httpclient method
  cart = signal<Cart | null>(null); //set cart property sto Angular Signal


  getCart(id: string) { //getting the cart with specific id from redis using angular pipe + signal
    return this.http.get<Cart>(this.baseUrl + 'cart?id=' + id).pipe(
      map(cart => {
        this.cart.set(cart);
        return cart;
      })
    )
  }

  setCart(cart: Cart) { //setting a cart to redis using angular observer + signal
    return this.http.post<Cart>(this.baseUrl + 'cart', cart).subscribe({
      next: cart => this.cart.set(cart)
    })
  }
```

Εικόνα 262

Λαμβάνοντας υπόψιν τα δύο μοντέλα που αναπτύξαμε σε προηγούμενες ενότητες (*product.ts* και *cart.ts* αρχεία), διαθέτουμε τρεις τρόπους προσθήκης ενός προϊόντος στο καλάθι αγορών. Ο χρήστης θα μπορεί μέσα από το περιβάλλον της σελίδας *Shop*, πατώντας απευθείας το κουμπί «*Add to cart*» στην κάρτα προϊόντος, ομοίως και από τη σελίδα της λεπτομερούς περιγραφής προϊόντος με το αντίστοιχο κουμπί, να προσθέτει προϊόντα στο καλάθι του (μοντέλο *product*). Η Τρίτη επιλογή θα είναι μέσα από ένα υφιστάμενο καλάθι, να μπορεί να προσθέτει ποσότητες σε ένα ήδη υπάρχον προϊόν (μοντέλο *cart*).

Εντός της *CartService* κλάσης δημιουργούμε μία νέα μέθοδο με όνομα *addItemToCart*. Τα ορίσματα αυτής της μεθόδου αποτελούν, η μεταβλητή *item* που είναι τύπου *CartItem* ή *ProductItem* και η μεταβλητή *quantity*, την οποία κι αρχικοποιούμε με την τιμή ένα. Στη συνέχεια, ορίζουμε τη σταθερά *cart* η οποία μέσω της μεθόδου *getCart*, μας δίνει την πληροφορία για το εάν υφίσταται το συγκεκριμένο καλάθι ή όχι εντός του *Redis server*. Σε περίπτωση που δεν υπάρχει, τότε καλείται η μέθοδος *createCart* και το δημιουργεί αναθέτοντας του ένα τυχαίο *id* (μέθοδος *nanoId*) και το αποθηκεύει στην προσωρινή μνήμη του *browser*. Εν συνεχεία, πραγματοποιούμε έλεγχο μέσω της μεθόδου *isProduct*, αναφορικά με τον τύπο της μεταβλητής *item*. Στην περίπτωση που μιλάμε για ένα *Product item*, η μέθοδος *isProduct* μας επιστρέφει την τιμή *true*. Αυτό σημαίνει πως το αντικείμενο *item* διαθέτει μία *property* της μορφής *Product.id*. Επομένως, ο έλεγχος είναι αληθής και καλείται η μέθοδος *mapProductToItem*, στην οποία δίνουμε ως όρισμα το αντικείμενο *item* και μας επιστρέφει αντιστοιχισμένες σε μορφή *CartIem* όλες τις οντότητες οι οποίες το περιγράφουν. Εφόσον διαχειριζόμαστε, εν τέλει, αντικείμενα της μορφής *CartItem*, καλούμε τη μέθοδο *addOrUpdateItem* με ορίσματα την *property cart.items* (πλέον) και τις παραμέτρους *item* και *quantity*. Σε περίπτωση που το συγκεκριμένο προϊόν υπάρχει στο υφιστάμενο καλάθι, τότε μεταβάλλεται αναλόγως και η ποσότητά του (βασιζόμενοι στην τιμή της παραμέτρου *quantity*), σε αντίθετη περίπτωση προστίθεται ένα νέο προϊόν στο καλάθι, συμπεριλαμβανομένων όλων των οντοτήτων που το διέπουν (παράμετρος *item*). Τέλος, καλείται η μέθοδος *setCart* για να ενημερωθεί ο *Redis server*. Αναλυτικά το, μέχρι στιγμής, περιεχόμενο της *TypeScript* κλάσης *CartService*, παρουσιάζεται στις εικόνες 263 και 264.

cart.service.ts ●

client > src > app > core > services > cart.service.ts > CartService > cart

```
11  export class CartService {
31
32    addItemToCart(item: CartItem | Product, quantity = 1) {//two models = two types
33      const cart = this.cart() ?? this.createCart();//check if cart exists if not c
34      if (this.isProduct(item)) {//checks if item is of type PRODUCT or CART
35        item = this.mapProductToCartItem(item);//if item is of type Product
36      }//then we map the Product properties to CartItem properties using the above
37      cart.items = this.addOrUpdateItem(cart.items, item, quantity);//adds item to
38      this.setCart(cart);//sets the cart
39    }
40
41
42    //if item exists update the cart item quantity - if not sets the quantity and i
43    private addOrUpdateItem(items: CartItem[], item: CartItem, quantity: number) {
44      const index = items.findIndex(i => i.productId === item.productId);
45      if (index === -1) {
46        item.quantity = quantity;
47        items.push(item);
48      } else {
49        items[index].quantity += quantity;
50      }
51      return items;
52    }
53
54    //if item is of type Poduct then remap its properties to match CartItem propert
55    private mapProductToCartItem(product: Product): CartItem {
56      return {
57        productId: product.id,
58        productName: product.name,
59        price: product.price,
60        quantity: 0,
61        pictureUrl: product.pictureUrl,
62        brand: product.brand,
63        type: product.type
64      };
65    }
```

Εικόνα 263

```
66
67    //decalres if item is of type PRODUCT OR CART based on the property
68    //Product.id - If item has the product.id property is of type PRODUCT else r
69    private isProduct(item: CartItem | Product): item is Product {
70      return (item as Product).id !== undefined;
71    }
72
73    //create a cart item in redis with random id
74    //saved in local browser storage
75    private createCart() {
76      const cart = new Cart();
77      localStorage.setItem('cart_id', cart.id);
78      return cart;
79    }
80  }
```

Εικόνα 264

Εντός του αρχείου *product-item.component.ts* θα κάνουμε *inject* την κλάση *CartService* (εικόνα 265).

```
26  export class ProductItemComponent {
27    @Input() product!: Product; //access to
28    cartService = inject(CartService); //in
29  }
```

Εικόνα 265

Τέλος, εντός του *html template product-item.component.html*, θα προσθέσουμε δύο *click elements* τα οποία αφορούν την κάρτα προϊόντος στη σελίδα *Shop* της εφαρμογής μας. Το πρώτο, παρακάμπτει το *routerLink element* προκειμένου να μη γίνεται ανακατεύθυνση στη σελίδα της λεπτομερούς περιγραφής του προϊόντος. Το δεύτερο καλεί τη μέθοδο *addItemToCart*, την οποία ορίσαμε νωρίτερα εντός της κλάσης *CartService*. Πλέον, ο χρήστης κάνοντας «κλικ» στο κουμπί «*Add to cart*», προσθέτει ένα προϊόν στο καλάθι αγορών του ή ενημερώνει το ήδη υπάρχον καλάθι αγορών του (εικόνα 266).

```
8        <mat-card-actions (click)="$event.stopPropagation()">
9            <button (click)="cartService.addItemToCart(product)"
10           mat-stroked-button class="w-full">
11               <mat-icon>add_shopping_cart</mat-icon>
12               Add to cart
13           </button>
14       </mat-card-actions>
```

Εικόνα 266

9.3 Ενημέρωση του Εικονιδίου shopping_cart

Σε αυτήν την ενότητα θα ενημερώσουμε το εικονίδιο «*shopping_cart*» στη μπάρα πλοήγησης, με σκοπό να παρουσιάζει το άθροισμα των ποσοτήτων, το οποίο αφορά τα προϊόντα που βρίσκονται εντός του καλαθιού αγορών. Ξεκινώντας από την κλάση *CartService*, ορίζουμε μία νέα μεταβλητή τύπου *Angular Signal* με τίτλο *itemCount* και της αναθέτουμε μία τιμή. Η τιμή αυτή προκύπτει ύστερα από τη χρήση των *Angular* μεθόδων *computed* και *reduce*. Η μέθοδος *computed* δέχεται ως όρισμα μία τιμή *Signal* και βασιζόμενη στα κριτήρια που υπολογίζονται από τη μέθοδο *reduce* (άθροισμα συνολικής ποσότητας αντικειμένων, βάσει της οντότητας *quantity*, ξεκινώντας από την τιμή μηδέν), μας επιστρέφει μία *Signal* τιμή (*sum*) ή *null*, εφόσον το καλάθι είναι κενό. Ο ορισμός της μεταβλητής *itemCart* παρουσιάζεται στην εικόνα 267.

```
itemCount = computed(() => { //computes the total amount of items in cart
  return this.cart()?.items.reduce((sum,item)=> sum + item.quantity,0)
});
```

Εικόνα 267

Επόμενος σταθμός, το αρχείο *header.components.ts*. Εδώ κάνουμε *inject* την *TypeScript* κλάση *CartService*, ώστε να αποκτήσουμε πρόσβαση στην *signal* μεταβλητή που ορίσαμε προηγουμένως (εικόνα 268).

```
export class HeaderComponent {
  busyService = inject(BusyService) //inject this service in order the loading bar to work
  cartService = inject(CartService) //injects cartservice to access count of items in cart
}
```

Εικόνα 268

Ακολούθως, θα μεταφερθούμε στο αντίστοιχο *html template* για να ενημερώσουμε το εικονίδιο *shopping_cart*, εκμεταλλευόμενοι την *signal* μεταβλητή *itemCount*. Όπως βλέπουμε στην εικόνα 269, στο *attribute matBadge* κάνουμε χρήση της *itemCount()* μέσω της οντότητας *cartService*, στην οποία ορίσαμε την *injected Angular Service CartService*. Ο λόγος που στη συγκεκριμένη περίπτωση χρησιμοποιούμε παρενθέσεις στην κλήση της *itemCount*, είναι επειδή πρόκειται για *Angular Signal* μεταβλητή.

```
<a routerLink="/cart"
routerLinkActive="active"
matBadge="{{cartService.itemCount()}}"
    matBadgeSize="large" class="custom-badge mt-2 mr-4">
    <mat-icon>shopping_cart</mat-icon>
</a>
```

Εικόνα 269

Τέλος, θέλοντας να δώσουμε έμφαση στον αριθμό των προϊόντων που συνοδεύει το εικονίδιο *shopping_cart*, θα μεταβούμε στο αρχείο *styles.css* και θα ορίσουμε έναν κόκκινο χρωματισμό ως φόντο, αναφερόμενο στο *Angular element mat.Badge* (εικόνα 270).

```
@include mat.badge-overrides(( //custom
    background-color: rgb(163, 0, 0)
));
```

Εικόνα 270

Η προηγηθείσα παραμετροποίηση, έχει ως αποτέλεσμα τη μεταβολή του εικονιδίου *shopping_cart* και την ένδειξη του συνολικού αριθμού προϊόντων εντός αυτού. Στην εικόνα 271 παρατηρούμε το εν λόγω εικονίδιο, του οποίου η εικόνα αφορά ένα κενό καλάθι αγορών.

Εικόνα 271

9.4 Ενημέρωση των Cart Components

Έχοντας προχωρήσει την υλοποίηση του καλαθιού αγορών, αναφορικά με την εισαγωγή προϊόντων, θα ασχοληθούμε με την εμφάνιση της σελίδας καλαθιού στον χρήστη. Ξεκινώντας, μέσω του τερματικού θα εκτελέσουμε την εντολή *ng g c features/cart/cart-item --skip-tests*, με την οποία δημιουργούμε ένα νέο *component* εντός του καταλόγου *cart*, με την ονομασία *cart-item*. Εντός του νέου αρχείου cart-*item.component.ts* ορίζουμε την *signal* μεταβλητή *item*, στην οποία αναθέτουμε την *required signal* τιμή τύπου *CartItem*, μέσω της *Angular Core* μεθόδου *input* (εικόνα 272).

client > src > app > features > cart > cart-item > cart-item.component.ts > ...

```
1  import { Component, input } from '@angular/core';
2  import { CartItem } from '../../../shared/models/cart';
3
4  @Component({
5    selector: 'app-cart-item',
6    imports: [],
7    templateUrl: './cart-item.component.html',
8    styleUrl: './cart-item.component.scss',
9  })
10 export class CartItemComponent {
11   item = input.required<CartItem>(); //required signal val
12 }
```

Εικόνα 272

Στη συνέχεια, θα μεταφερθούμε στο αντίστοιχο *html template* ώστε να διαμορφώσουμε την παρουσίαση των προϊόντων που βρίσκονται στο καλάθι αγορών. Κάνοντας χρήση των κατάλληλων *html*, *Angular* και *Tailwind attributes*, παρουσιάζουμε κάθε προϊόν βασιζόμενοι στην *signal* μεταβλητή *item()*, έχοντας πρόσβαση στις *properties* της οντότητας *CartItem*. Ουσιαστικά δημιουργούμε *rounded*, μπλε περιγράμματος και λευκού φόντου πλαίσια, εντός των οποίων παραθέτουμε την εικόνα και την περιγραφή κάθε προϊόντος θέτοντάς τα ως *links* προς τη σελίδα λεπτομερούς περιγραφής του. Επιπλέον, εισάγουμε τα κουμπιά διαγραφής, προσθήκης κι αφαίρεσης ποσοτήτων για το εκάστοτε προϊόν, καθώς επίσης και την αξία του. Στις εικόνες 273 και 274, παρουσιάζεται ο κώδικας του *cart-item.component.html template*.

```
<div class="rounded-lg border border-blue-500 bg-white p-4 shadow-sm mb-4 ml-50">
    <div class="flex items-center justify-between gap-8">
        <a routerLink="/shop/{{item().productId}}" class="shrink order-1">
            <img class="h-25 w-25" src="{{item().pictureUrl}}"
                alt="product image">
        </a>
        <div class="flex items-center justify-between order-3">
            <div class="flex items-center align-middle gap-6">
                <button mat-icon-button>
                    <mat-icon class="text-red-600">remove</mat-icon>
                </button>

                <div class="font-semibold text-xl mb-1">
                    {{item().quantity}}</div>

                <button mat-icon-button>
                    <mat-icon class="text-green-600">add</mat-icon>
                </button>
            </div>
```

Εικόνα 273

```

            <div class="text-end order-4 w-32 ml-5">
                <p class="font-bold text-xl text-gray-900">
                    {{item().price | currency:'EUR':'symbol':'1.2-2':'fr'}}</p>
            </div>
        </div>
        <div class="w-full flex flex-col flex-3 space-y-4 order-2 max-w-md">
            <a routerLink="/shop/{{item().productId}}" class="font-medium">
                {{item().productName}}
            </a>
            <div class="flex items-center gap-4">
                <button mat-button class="text-red-700 flex gap-2 items-center">
                    <mat-icon>delete</mat-icon>
                    <span>Delete</span>
                </button>
            </div>
        </div>
    </div>
</div>
```

Εικόνα 274

Ακολούθως, κάνουμε *inject* την *CartService* κλάση εντός της κλάσης *CartComponent*, στο αρχείο *cart.component.ts* (εικόνα 275).

```
13  export class CartComponent {
14    cartService = inject(CartService);
15  }
```

Εικόνα 275

Εντός του αντίστοιχου *html template*, χρησιμοποιούμε την *signal* μεταβλητή *cart()* και μέσω της οντότητας *items*, διατρέχουμε εντός ενός *for-loop* όλα τα προϊόντα που έχουν εισαχθεί στο καλάθι κι έχουν αποθηκευτεί στον *Redis server*. Η παρουσίαση των προϊόντων γίνεται μέσω του *child cart-item component* (εικόνα 276).

```
<section>
    <div class="mx-auto max-w-full">
      <div class="flex w-full items-start gap-6 mt-32">
        <div class="w-3/4">
          @for (item of cartService.cart()?.items; track item.productId) {
            <app-cart-item [item]="item"></app-cart-item>
          }
        </div>
      </div>
    </div>
  </section>
```

Εικόνα 276

9.5 Δημιουργία του Order Summary Component

Εκτός της παρουσίασης των προϊόντων στο καλάθι αγορών, ο χρήστης θα πρέπει να έχει εικόνα για τη συνολική αξία των προϊόντων του, καθώς και για το αν υπάρχει κάποια έκπτωση ή χρέωση μεταφορικών στην παραγγελία που θα πραγματοποιήσει. Για τους λόγους που μόλις αναλύσαμε, θα προσθέσουμε ακόμη ένα νέο component στην εφαρμογή μας εκτελώντας την εντολή *ng g c shared/components/order-summary --skip-tests*. Το νέο *component* ονομάζεται *order-summary* και τοποθετείται εντός του καταλόγου *shared*. Ανοίγοντας το *html template* του νέου *component* θα προσθέσουμε τον κώδικα της εικόνας 277.

```
<div class="mx-auto max-w-4xl flex-1 space-y-6 w-full">
  <div class="space-y-4 rounded-lg border border-blue-500 bg-white p-4 shadow-sm">
    <p class="text-xl font-semibold">Order summary</p>
    <div class="space-y-4">
      <div class="space-y-2">
        <dl class="flex items-center justify-between gap-4">
          <dt class="font-medium text-gray-500">Subtotal</dt>
          <dd class="font-medium text-gray-900">0,00 €</dd>
        </dl>
        <dl class="flex items-center justify-between gap-4">
          <dt class="font-medium text-gray-500">Discount</dt>
          <dd class="font-medium text-green-500">-0,00 €</dd>
        </dl>
        <dl class="flex items-center justify-between gap-4">
          <dt class="font-medium text-gray-500">Delivery fee</dt>
          <dd class="font-medium text-gray-900">0,00 €</dd>
        </dl>
      </div>
      <dl class="flex items-center justify-between gap-4 border-t border-gray-200 pt-2">
        <dt class="font-bold text-gray-900">Total</dt>
        <dd class="font-semibold text-gray-900">0,00 €</dd>
      </dl>
    </div>

    <div class="flex flex-col gap-2">
      <button routerLink="/checkout" mat-flat-button>Checkout</button>
      <button routerLink="/shop" mat-button>Continue Shopping</button>
    </div>
  </div>

  <div class="space-y-4 rounded-lg border border-blue-500 bg-white shadow-sm">
    <form class="space-y-2 flex flex-col p-2">
      <label class="mb-2 block text-sm font-medium">Do you have a voucher code?</label>
      <mat-form-field appearance="outline">
        <mat-label>Voucher code</mat-label>
        <input matInput type="text">
      </mat-form-field>

      <button mat-flat-button>Apply code</button>
    </form>
  </div>
</div>
```

Εικόνα 277

Με τον παραπάνω κώδικα πετυχαίνουμε τη δημιουργία δύο νέων *rounded* πλαισίων, μπλε περιγράμματος και λευκού φόντου. Το πρώτο, κατά σειρά εμφάνισης, περιέχει τις πληροφορίες για το υποσύνολο της αξίας του καλαθιού, την έκπτωση που ενδέχεται να υπάρχει, το κόστος αποστολής καθώς και τη συνολική αξία του καλαθιού. Ακριβώς από κάτω, χρησιμοποιώντας ένα νέο *div attribute* και κάνοντας χρήση των *routerlink elements*, δημιουργούμε το κουμπί «*Checkout*» και το κουμπί «*Continue Shopping*». Στο δεύτερο πλαίσιο, το οποίο τοποθετείται ακριβώς από κάτω από το πρώτο, χρησιμοποιούμε το *mat-form-field element* όπου ο χρήστης μπορεί να εισάγει τον κωδικό από ένα εκπτωτικό κουπόνι. Πατώντας το *mat-flat-button* κουμπί «*Apply code*», θα μπορεί να εφαρμόσει έκπτωση στη συνολική αξία του καλαθιού του. Προς το παρόν, υλοποιήσαμε καθαρά και μόνο το εμφανισιακό μέρος του εν λόγω *template*. Στην

επόμενη ενότητα θα ενημερώσουμε και τη λειτουργικότητά του. Συνεχίζοντας, θα μεταφερθούμε στο *cart.component.html* αρχείο και θα προσθέσουμε ένα νέο * element*, σύμφωνα με την εικόνα 278. Πρακτικά, χρησιμοποιούμε το ένα τέταρτο της οθόνης μας για να παρουσιάσουμε το περιεχόμενο του *order-summary component*, δίπλα ακριβώς από το περιεχόμενο του *cart-item component*, το οποίο καταλαμβάνει τα τρία τέταρτα αυτής.

```
9           <div class="w-1/4">
10            <app-order-summary></app-order-summary>
11          </div>
```

Εικόνα 278

9.5.1 Υπολογισμός των Συνολικών Αξιών του Καλαθιού Αγορών

Με στόχο τον εμπλουτισμό, από λειτουργικής άποψης, του κώδικα *summary.component.html template*, θα μεταβούμε στο *cart.service.ts* αρχείο και θα δημιουργήσουμε μία νέα οντότητα. Η οντότητα αυτή ονομάζεται *totals*, εκμεταλλεύεται τη λειτουργικότητα των *signal* αντικειμένων *computed* και *cart*, μέσω των οποίων υπολογίζεται η μερική αξία του συνόλου των προϊόντων εντός του καλαθιού. Η τιμή αυτή προκύπτει κατά την εκτέλεση της *callback* μεθόδου *reduce*, η οποία υπολογίζει τον αριθμό των τεμαχίων κάθε προϊόντος στο καλάθι αγορών και για το κάθε ένα ξεχωριστά τη συνολική του αξία. Το άθροισμα όλων των αξιών διαμορφώνει το μερικό κόστος του καλαθιού, το οποίο αποθηκεύεται στη σταθερά *subtotal*. Στις σταθερές *shipping* και *discount*, προς το παρόν αναθέτουμε την τιμή μηδέν, ενώ στη μεταβλητή *total* το άθροισμα όλων των προαναφερθέντων μεταβλητών (εικόνα 279).

```
83  totals = computed(()=> {//calculating the order summary values via signal variables (cart)
84    const cart = this.cart();//using cart signal variable
85    if (!cart) return null;//if the cart is empty or not found
86    const subtotal = cart.items.reduce((sum, item) => sum + item.price * item.quantity, 0);
87    const shipping = 0;
88    const discount = 0;
89    return {
90      subtotal,
91      shipping: 0,
92      discount: 0,
93      total: subtotal + shipping - discount
94    };
95  })
```

Εικόνα 279

Εντός του *order-summary.component.html* αρχείου, το μόνο που έχουμε να κάνουμε είναι η αντικατάσταση των *hardcoded* τιμών στο πεδίο κάθε τιμής της λίστας *Order Summary*, με τις αντίστοιχες τιμές που μας δίνει η *Angular Signal based* οντότητα *totals*. Στην εικόνα 280 βλέπουμε τις απαραίτητες αλλαγές στον κώδικα html.

```
<dl class="flex items-center justify-between gap-4">
  <dt class="font-medium text-gray-500">Subtotal</dt>
  <dd class="font-medium text-gray-900">{{cartService.totals()?.subtotal | currency:'EUR':'symbol':'1.2-2':'fr'}}</dd>
</dl>
<dl class="flex items-center justify-between gap-4">
  <dt class="font-medium text-gray-500">Discount</dt>
  <dd class="font-medium text-green-500">-{{cartService.totals()?.discount | currency:'EUR':'symbol':'1.2-2':'fr'}}</dd>
</dl>
<dl class="flex items-center justify-between gap-4">
  <dt class="font-medium text-gray-500">Delivery fee</dt>
  <dd class="font-medium text-gray-900">{{cartService.totals()?.shipping | currency:'EUR':'symbol':'1.2-2':'fr'}}</dd>
</dl>
'div>
ll class="flex items-center justify-between gap-4 border-t border-gray-200 pt-2">
<dt class="font-bold text-gray-900">Total</dt>
<dd class="font-semibold text-gray-900">{{cartService.totals()?.total | currency:'EUR':'symbol':'1.2-2':'fr'}}</dd>
```

Εικόνα 280

9.6 Μέθοδοι removeItemFromCart και deleteCart

Σε αυτήν την ενότητα θα ορίσουμε δύο νέες μεθόδους, την *removeItemFromCart* και την *deleteCart*, εντός της *Angular service cart*. Η πρώτη (εικόνα 281), δέχεται ως ορίσματα έναν αριθμό ο οποίος αντιπροσωπεύει το *id* ενός προϊόντος κι έναν ακόμη, ο οποίος αντιπροσωπεύει την ποσότητα των τεμαχίων προς αφαίρεση για αυτό το προϊόν από το καλάθι αγορών. Εφόσον ελέγξουμε την ύπαρξη του καλαθιού μέσω της *signal* μεταβλητής *cart*, ελέγχουμε και την ύπαρξη του συγκεκριμένου προϊόντος εντός αυτού. Από τη στιγμή που η ποσότητα των τεμαχίων εντός του καλαθιού υπερκαλύπτει την ποσότητα προς αφαίρεση, τότε μειώνουμε αναλόγως την ποσότητα των τεμαχίων αυτού του προϊόντος. Σε αντίθετη περίπτωση, διαγράφουμε εξ ολοκλήρου την εγγραφή που έχει να κάνει με το συγκεκριμένο προϊόν. Στην περίπτωση που το καλάθι μας περιείχε ένα προϊόν και το αφαιρέσαμε εξ ολοκλήρου, καλείται η μέθοδος *deleteCart* (εικόνα 282), η οποία βάσει του *cart id* κι ενός *observable* αντικειμένου, διαγράφει το καλάθι αγορών από τον *Redis server* και την προσωρινή μνήμη του φυλλομετρητή μας.

```
44   removeItemFromCart(productId: number, quantity = 1) {
45     const cart = this.cart(); //get cart from signal variable cart
46     if (!cart) return;//if cart is empty or not exists returns
47     const index = cart.items.findIndex(i => i.productId === productId);/
48     if (index !== -1) {//if true then the specified item of specific id
49       if (cart.items[index].quantity > quantity) { //checks if reduce qu
50         cart.items[index].quantity -= quantity;//than existing item quan
51       } else {//if not sets the new quantity
52         cart.items.splice(index, 1);//eslse deletes the item from cart
53       }
54       if (cart.items.length === 0) {//if no items in cart after quantity
55         this.deleteCart();//deletes the whole cart
56       } else {
57         this.setCart(cart);//sets the cart in redis server
58       }
59     }
60   }
```

Εικόνα 281

```
62  //deletes the whole cart from redis server and local browser storage
63  deleteCart() {
64    this.http.delete(this.baseUrl + 'cart?id=' + this.cart()?.id).subscribe({
65      next: () => {
66        localStorage.removeItem('cart_id');//deletes local storage
67        this.cart.set(null);//sets signal value to null
68      }
69    });
70  }
```

Εικόνα 282

Ακολούθως, μεταφερόμαστε στην *TypeScript* κλάση *CartItemComponent* και κάνουμε *inject* την *TypeScript* κλάση *CartService*. Ύστερα, ορίζουμε τρεις μεθόδους, την *incrementQuantity*, την *decrementQuanity* και την *deleteItemFromCart*. Έχοντας στο μυαλό μας την παραμετροποίηση του *cart-item.component.html template* (εικόνες 273 & 274), δημιουργούμε τις τρεις αυτές μεθόδους ώστε ο χρήστης να έχει τη δυνατότητα να μειώνει ή να αυξάνει την ποσότητα των τεμαχίων ενός προϊόντος, ή ακόμη και να το διαγράφει με το πάτημα ενός κουμπιού. Στην εικόνα 283 παρουσιάζονται οι νέες μέθοδοι.

```
22  cartService = inject(CartService); //injects cartservice to access to methods
23
24  incrementQuantity() {//+ button in template
25    this.cartService.addItemToCart(this.item());
26  }
27
28  decrementQuantity() {//- button in template
29    this.cartService.removeItemFromCart(this.item().productId, 1);
30  }
31
32  deleteItemFromCart() {//delete button in template
33    this.cartService.removeItemFromCart(this.item().productId, this.item().quantity);
34  }
```

Εικόνα 283

Στις εικόνες 284 και 285 παρουσιάζονται οι απαραίτητες αλλαγές, με την εισαγωγή των *click events* στα αντίστοιχα *Angular mat-buttons*, αλλά και της εισαγωγής των νέων μεθόδων εντός του *html template cart-item.component.html*.

```
<button mat-icon-button (click)="decrementQuantity()">
    <mat-icon class="text-red-600">remove</mat-icon>
</button>
<div class="font-semibold text-xl mb-1">{{item().quantity}}</div>
<button mat-icon-button (click)="incrementQuantity()">
    <mat-icon class="text-green-600">add</mat-icon>
</button>
```

Εικόνα 284

```
<button (click)="deleteItemFromCart()" mat-button
    class="text-red-700 flex gap-2 items-center">
    <mat-icon>delete</mat-icon>
    <span>Delete</span>
</button>
```

Εικόνα 285

9.7 Ενημέρωση της Σελίδας Προϊόντος

Σε αυτήν την ενότητα θα ασχοληθούμε με το *product-details component*, με στόχο την αναβάθμιση της λειτουργικότητας του αντίστοιχου *template*. Όπως αναφέραμε και στην αρχή αυτού του κεφαλαίου, ο χρήστης δύναται να προσθέτει προϊόντα μέσα από τη σελίδα *Shop*, του ηλεκτρονικού μας καταστήματος, αλλά και μέσα από τη σελίδα λεπτομερούς περιγραφής του εκάστοτε προϊόντος. Στο σημείο αυτό, θα μεταβούμε στον κώδικα του αρχείου *product-details.component.ts* και θα κάνουμε *inject* την κλάση *CartService*. Στη συνέχεια, δημιουργούμε τη μέθοδο *UpdateQuantityInCart* με σκοπό να ελέγχουμε την περίπτωση που ένα προϊόν υπάρχει ήδη στο καλάθι αγορών ή όχι, χρησιμοποιώντας την *signal* μεταβλητή *cart*. Η μέθοδος αυτή καλείται εντός της μεθόδου *loadProduct*, ώστε να αρχικοποιηθεί η ποσότητα του εκάστοτε προϊόντος στο καλάθι αγορών. Αμέσως μετά, δημιουργούμε τη μέθοδο *getButtonText*, η οποία μας επιστρέφει ένα *string*, βασιζόμενο στην ποσότητα ενός προϊόντος εντός του καλαθιού αγορών. Όταν το προϊόν έχει ήδη προστεθεί στο καλάθι, το αλφαριθμητικό που επιστρέφεται είναι το «*Update cart*», ενώ στην αντίθετη περίπτωση είναι το «*Add to cart*». Τέλος, ορίζουμε τη μέθοδο *updateCart*. Με τη συγκεκριμένη μέθοδο επιτυγχάνουμε την ενημέρωση του καλαθιού αγορών, βασιζόμενοι στις τιμές που εισάγει ο χρήστης. Αν, χάριν παραδείγματος, έχουμε ένα προϊόν που έχει τρία τεμάχια στο καλάθι, ο χρήστης εισάγει τον αριθμό δύο κι επιλέξει «*Update cart*», τότε ο αριθμός των τεμαχίων θα τεθεί σε δύο. Αντίστοιχα, εάν έχουμε τέσσερα τεμάχια ενός προϊόντος στο καλάθι κι ο χρήστης εισάγει την ποσότητα δέκα, τότε τα τεμάχια του συγκεκριμένου προϊόντος θα τεθούν σε δέκα. Με πιο απλά λόγια ανεξαρτήτως της αρχικής ποσότητας των προϊόντων εντός του καλαθιού, μέσα από τη σελίδα προϊόντος ο χρήστης μπορεί να θέτει τον ακριβή αριθμό των τεμαχίων, βάσει της ποσότητας που ο ίδιος επιλέγει. Στην περίπτωση που εισάγει τον αριθμό μηδέν, τότε το προϊόν αφαιρείται από το καλάθι. Στην εικόνα 286 παρουσιάζονται οι προσθήκες στον κώδικα της κλάσης *ProductDetailsComponent*.

```
63  updateCart(){// sets the amount of a product in shopping cart or deletes
64    if(!this.product) return;
65    if(this.quantity > this.quantityInCart){
66      const itemsToAdd = this.quantity - this.quantityInCart;
67      this.quantityInCart += itemsToAdd; //adds if given quantity is great
68      this.cartService.addItemToCart(this.product, itemsToAdd);
69    }else{
70      const itemsToRemove = this.quantityInCart - this.quantity;
71      this.quantityInCart -= itemsToRemove;
72      this.cartService.removeItemFromCart(this.product.id, itemsToRemove);
73    }//is smallrer that cart's
74  }
75
76
77  updateQuantityInCart() { //sets the quantity variable based on cart signa
78    this.quantityInCart = this.cartService.cart()?.items.
79      find(item => item.productId === this.product?.id)?.quantity || 0;
80    this.quantity = this.quantityInCart || 1;
81  }
82
83  getButtonText(){//returns the message on add to cart button based on ite
84    return this.quantityInCart>0 ? 'Update cart' : 'Add to cart'
85  }
```

Εικόνα 286

Τέλος, εντός του *product-details.html* αρχείου, το οποίο αφορά τον σχεδιασμό και τη λειτουργικότητα της σελίδας προϊόντος του ηλεκτρονικού μας καταστήματος, προχωράμε στις εξής αλλαγές: Αρχικά, μέσω ενός *click event attribute* χρησιμοποιούμε τη μέθοδο *updateCart* κι εκμεταλλευόμενοι τη λειτουρ-

γικότητα των *Angular Signalς* μεταβλητών, θέτουμε τιμές στις ποσότητες ενός προϊόντος, το οποίο προστίθεται ή αφαιρείται από το καλάθι. Στη συνέχεια, αναφορικά με το *shopping_cart mat-button*, το θέτουμε ως ανενεργό στην περίπτωση που η ποσότητα, εντός καλαθιού, ενός προϊόντος είναι ίση με την ποσότητα που θέτει ο χρήστης, καθώς επίσης και στην περίπτωση της τιμής μηδέν, όταν το προϊόν δεν έχει προστεθεί στο καλάθι αγορών. Χρησιμοποιώντας τη μέθοδο *getButtonText*, εμφανίζεται στο *button shopping_cart* το ανάλογο μήνυμα, «*Add to cart*» ή «*Update cart*». Στο *mat-form-field* πεδίο, εκμεταλλευόμαστε το *Angular two-way binding*, ορίζουμε μία ελάχιστη και μία μέγιστη τιμή εισαγωγής από τον χρήστη, αποθηκεύουμε την τιμή αυτή στη μεταβλητή *quantity* και ύστερα αποστέλλεται στο *component product-details* για ενημέρωση του καλαθιού από τις υπεύθυνες μεθόδους. Επίσης, θέλοντας να καθοδηγήσουμε τον χρήστη και να του δώσουμε μία εικόνα για το εκάστοτε προϊόν και το καλάθι αγορών του μέσα από τη σελίδα *product details*, κάτω από τα πεδία εισαγωγής ποσότητας και υποβολής, αναφέρουμε πόσα τεμάχια του συγκεκριμένου προϊόντος βρίσκονται στο καλάθι του τη δεδομένη χρονική στιγμή. Οι προσθήκες στο *product-details.html template* παρουσιάζονται στην εικόνα 287.

```
<div class="flex gap-3 mt-6">
    <button [disabled]="quantity === quantityInCart"
        (click)="updateCart()"
        class="button gap-2.5 font-semibold" mat-flat-button >
        <mat-icon>shopping_cart</mat-icon>
        {{getButtonText()}}
    </button>
    <mat-form-field appearance="outline" class="flex">
        <mat-label>Insert Quantity</mat-label>
        <input matInput min="0" max="2" [(ngModel)]="quantity" type="number">
    </mat-form-field>
    <button class="text-black font-medium text-2xl">
        <a>In Stock: <a class=" text-emerald-500 ">{{product.quantityInStock}}</a
    </button>
</div>
<p class="text-amber-500">
    You have {{quantityInCart}} piece(s) of this item inside your cart!
</p>
```

Εικόνα 287

9.8 Προσθήκη των Checkout Components

Ο χρήστης ολοκληρώνοντας την προσθήκη των προϊόντων στο καλάθι αγορών έχει δύο επιλογές, είτε να συνεχίσει την περιήγησή του στην εφαρμογή μας, είτε να μεταφέρεται επιλέγοντας το κουμπί «*Checkout*» στη σελίδα ολοκλήρωσης της παραγγελίας του. Με τις εντολές *ng g c features/checkout --skip-tests* και *ng g s core/services/checkout --skip-tests*, δημιουργούνται τα απαραίτητα *components* αλλά και η *Angular service*, τα οποία θα μας βοηθήσουν στη συνέχεια με την υλοποίηση της διαδικασίας «*Login*», «*Register*» και κατόπιν του «*Checkout*» στα επόμενα κεφάλαια. Καθαρά για λόγους παρουσίασης, ανοίγουμε το *checkout.component.html template* και προσθέτουμε το μήνυμα της εικόνας 288.

checkout.component.html

client > src > app > features > checkout > checkout.component.html > div.mt-25 > h1.text-5xl

Go to component

```
1  <div class="mt-25">
2      <h1 class="text-5xl">Only authorized/registered users have access to this page</h1>
3  </div>
```

Εικόνα 288

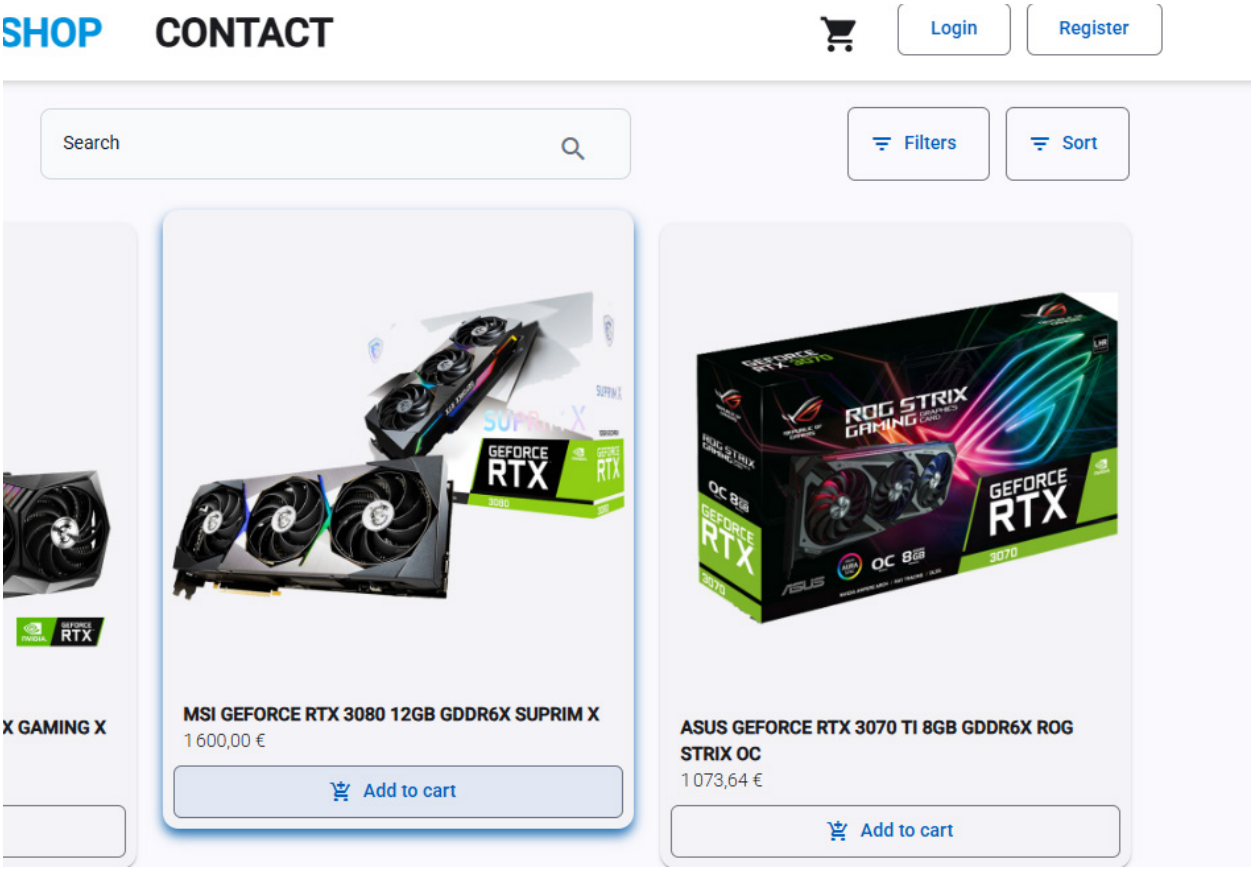
Ακολούθως, προσθέτουμε ένα ακόμη *path* στο αρχείο *app.routes.ts*, το οποίο ανακατευθύνει τον χρήστη στη σελίδα του *checkout component* (εικόνα 289).

```
18      {path:'checkout', component: CheckoutComponent},
```

Εικόνα 289

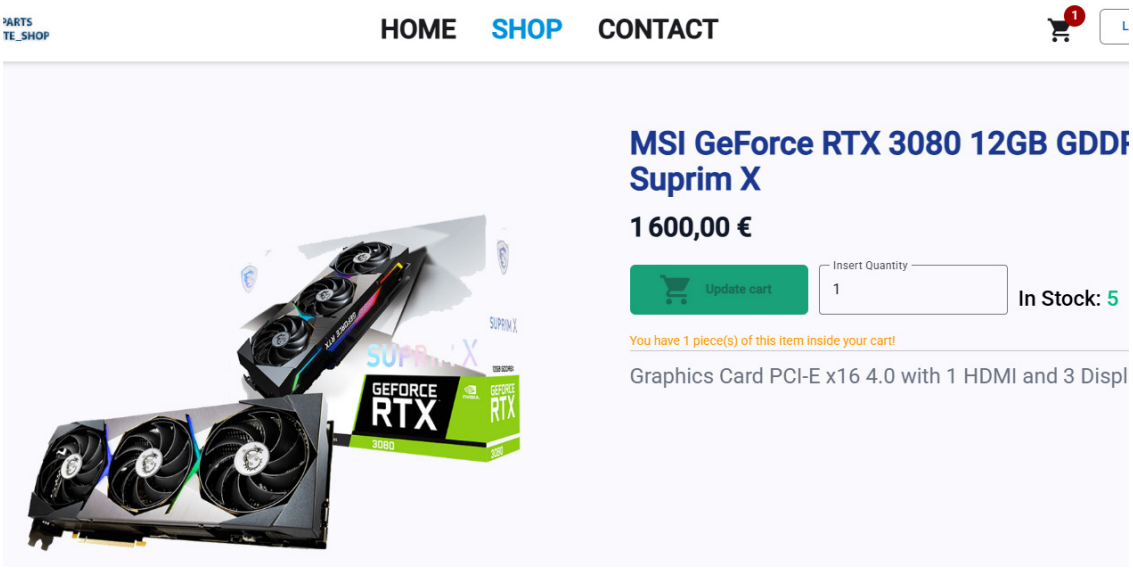
9.9 Σύνοψη της Λειτουργικότητας του Καλαθιού αγορών

Φτάνοντας στο κλείσιμο του τρέχοντος κεφαλαίου, μέσα από μία σειρά εικόνων, θα παρουσιάσουμε τις νέες λειτουργικότητες αναφορικά με το καλάθι αγορών της εφαρμογής PcParts. Στην εικόνα 290, ο χρήστης βρίσκεται στη σελίδα *Shop* του ηλεκτρονικού μας καταστήματος. Επιλέγοντας το κουμπί «*Add to cart*» στην κάρτα προϊόντος, προσθέτει το προϊόν *MSI GeForce RTX 3080 12GB GDDR6X Suprim X* στο καλάθι του.



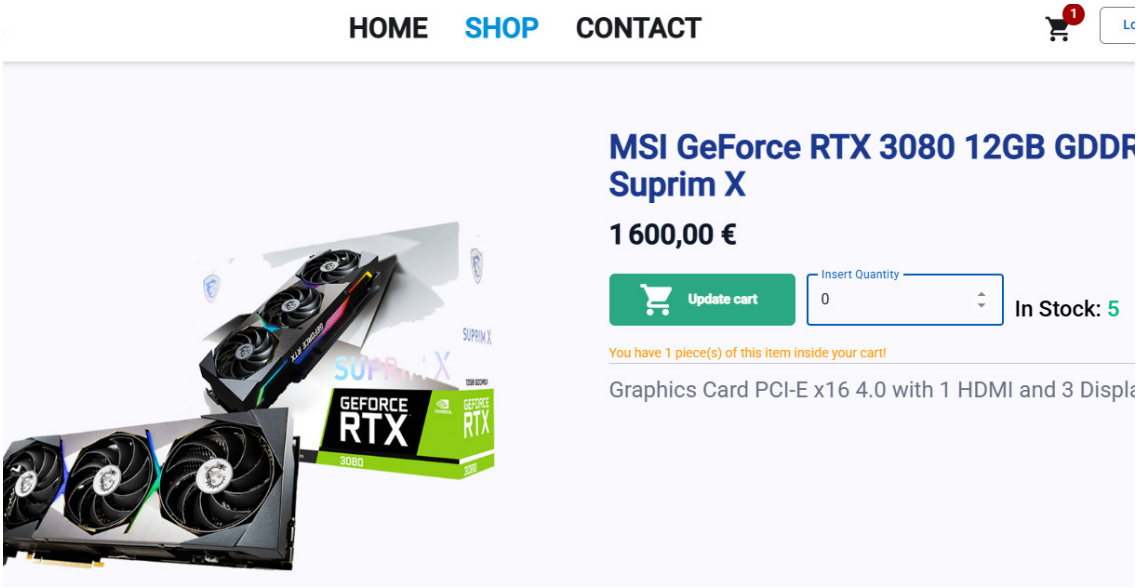
Εικόνα 290

Επιλέγοντας την κάρτα προϊόντος, ο χρήστης έχει μπροστά του τη σελίδα που παρουσιάζεται στην εικόνα 291. Το καλάθι του περιέχει ένα προϊόν (αριθμός ένα, σε κόκκινο φόντο, στο εικονίδιο *shopping_cart* της μπάρας πλοήγησης) και την πληροφορία, σε κίτρινο χρωματισμό, για το ότι ήδη έχει προσθέσει ένα τεμάχιο του συγκεκριμένου προϊόντος στο καλάθι του. Παρατηρούμε επίσης το μήνυμα «*Update cart*» στο πράσινο *disabled* πλήκτρο. Αυτό συμβαίνει διότι η ποσότητα ένα, ήδη έχει προστεθεί στο καλάθι αγορών του και οι μόνες επιλογές εισαγωγής είναι η τιμή δύο ή εισαγωγή της τιμής μηδέν, με σκοπό να αδειάσει το καλάθι του.



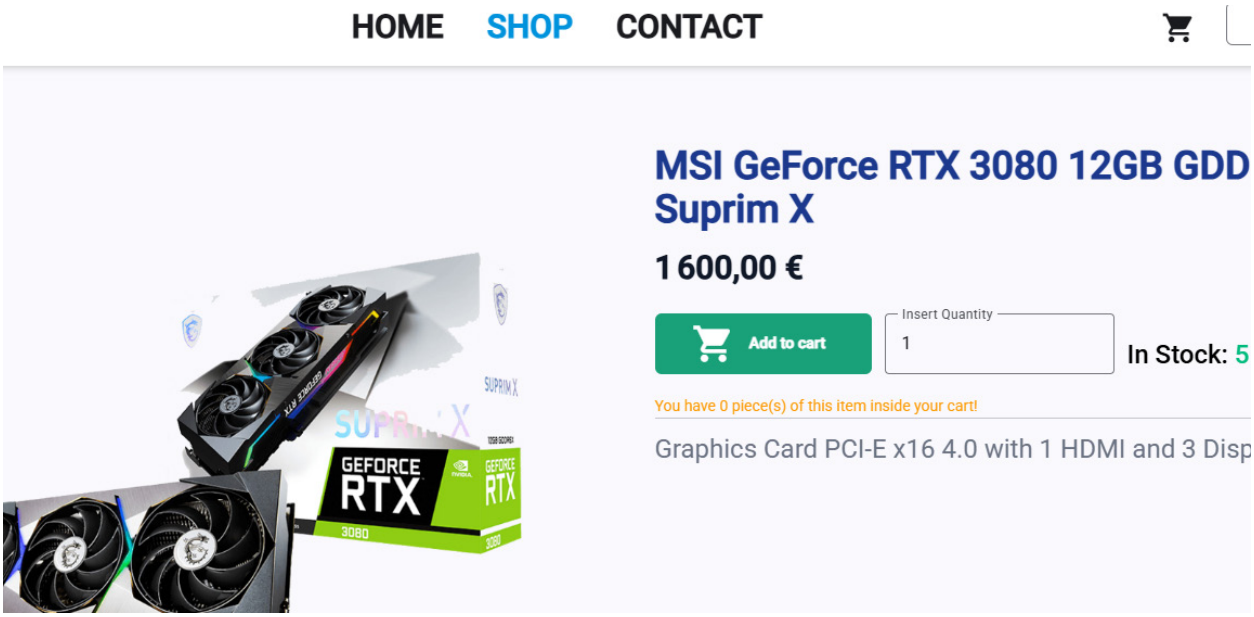
Εικόνα 291

Δοκιμάζοντας να εισάγουμε την επιλογή μηδέν, έχουμε το αποτέλεσμα της εικόνας 292. Το κουμπί «*Update cart*» είναι διαθέσιμο προς επιλογή.



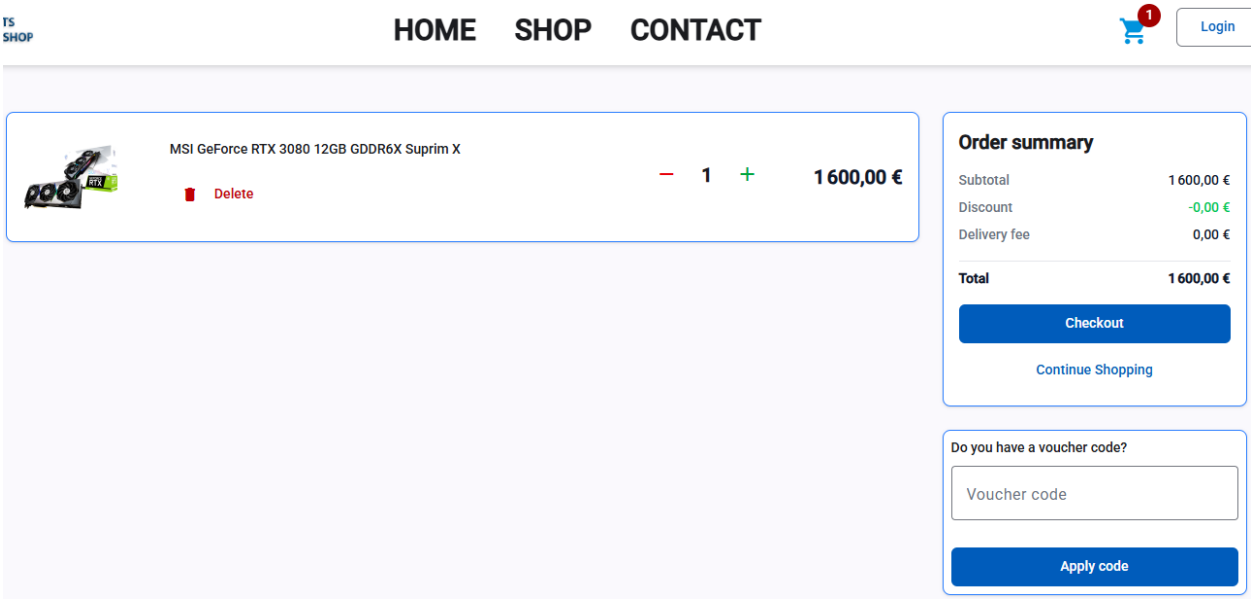
Εικόνα 292

Υποβάλλοντας την τιμή μηδέν, παίρνουμε το αποτέλεσμα της εικόνας 293. Το πλήκτρο υποβολής περιέχει το κείμενο «*Add to cart*» και είναι διαθέσιμο για επιλογή. Το εικονίδιο *shopping_cart* μεταβλήθηκε στην αρχική του κατάσταση. Ο χρήστης λαμβάνει το μήνυμα πως το συγκεκριμένο προϊόν δεν βρίσκεται στο καλάθι του.



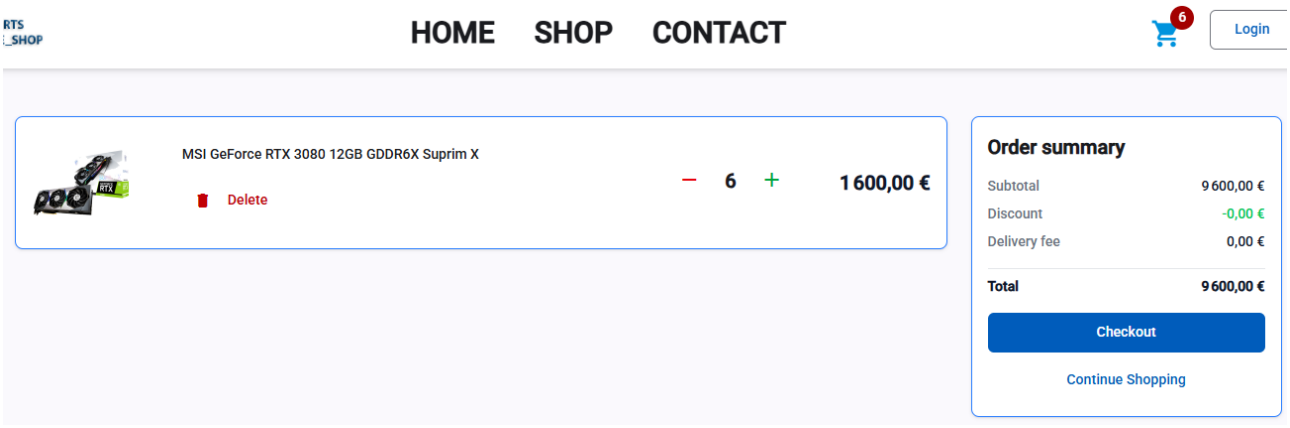
Εικόνα 293

Κάνοντας εκ νέου εισαγωγή ενός τεμαχίου στο καλάθι κι επιλέγοντας το εικονίδιο *shopping_cart*, μεταφερόμαστε στη σελίδα του *cart component* (εικόνα 294). Το προϊόν απεικονίζεται εντός του πλαισίου που έχουμε ορίσει σύμφωνα με το *cart-item.component.html template*.

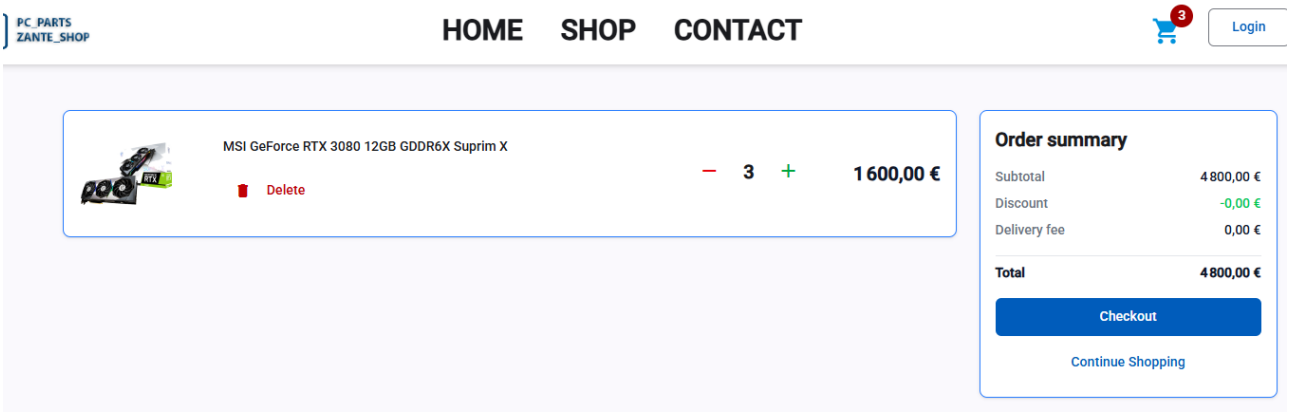


Εικόνα 294

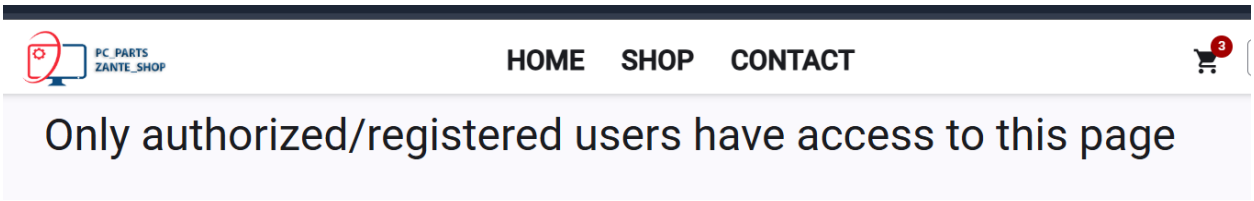
Ο χρήστης διαθέτει τις επιλογές αύξησης και μείωσης των τεμαχίων του προϊόντος, καθώς και την επιλογή της διαγραφής του από το τρέχον καλάθι αγορών. Στην εικόνα 295 αυξάνουμε την ποσότητα των τεμαχίων κατά πέντε μονάδες, ενώ στην εικόνα 296 τη μειώνουμε κατά τρεις μονάδες. Σε όλες τις εικόνες παρατηρούμε τη μεταβολή της συνολικής αξίας του καλαθιού στο πεδίο «*Order summary*». Στην εικόνα 297 επιλέγουμε το κουμπί «*Checkout*» κι εμφανίζεται η σελίδα του *checkout component*.



Εικόνα 295

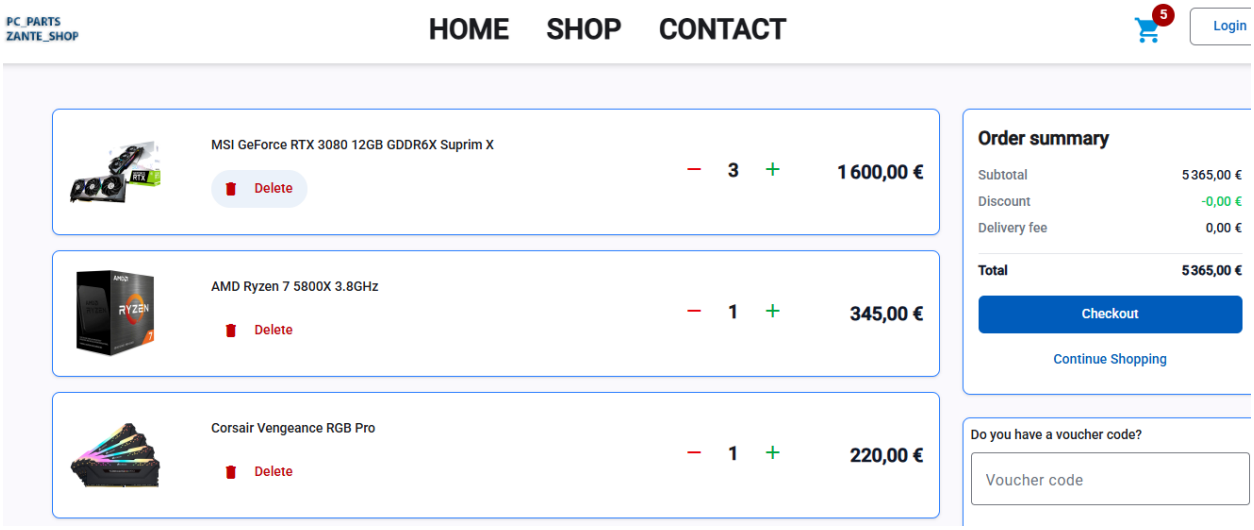


Εικόνα 296

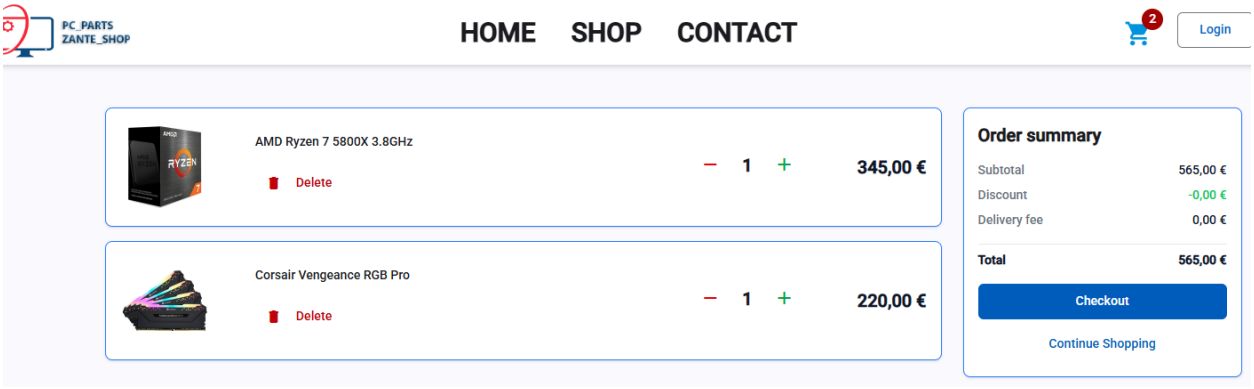


Εικόνα 297

Επιλέγοντας εκ νέου το εικονίδιο *shopping_cart* στην εικόνα 298 κι αφού έχουμε προσθέσει δύο νέα προϊόντα εντός του, κάνουμε κλικ το κουμπί «*Delete*» του προϊόντος *MSI GeForce RTX 3080 12GB GDDR6X Suprim X*. Το συγκεκριμένο προϊόν διαγράφεται κι ενημερώνεται αντίστοιχα το περιεχόμενο και η συνολική αξία του καλαθιού. Το αποτέλεσμα της ενέργειάς μας παρουσιάζεται στην εικόνα 299.



Εικόνα 298



Εικόνα 299

Κεφάλαιο 10. ASP.NET Core Identity

Σε αυτό το κεφάλαιο θα ασχοληθούμε με τις βιβλιοθήκες και τα *endpoints* του *ASP.NET Core Identity*. Το *.NET Core Identity* είναι ένα αρθρωτό σύστημα σχεδιασμένο για το *ASP.NET Core framework*, το οποίο παρέχει *APIs* για τη διαχείριση χρηστών και λειτουργίες ασφαλείας, όπως είναι ο έλεγχος ταυτότητας κι εξουσιοδότησης, εγγραφή χρήστη (*Register Process*) και σύνδεση χρήστη (*Login Process*). Επιτρέπει στους προγραμματιστές να διαχειρίζονται τις πληροφορίες ασφαλείας των χρηστών, τους κωδικούς πρόσβασης, τους ρόλους και τις αξιώσεις τους. Επίσης, Επαληθεύει τις ταυτότητες των χρηστών, χρησιμοποιώντας έλεγχο ταυτότητας που βασίζεται στα *cookies,* για εφαρμογές ιστού ή έλεγχο ταυτότητας που βασίζεται σε *tokens* για *API* εφαρμογές.

10.1 Χρήση του ASP.NET Core Identity στο API Project

Στην περίπτωση της διαδικτυακής εφαρμογής μας, θα εκμεταλλευτούμε τη λειτουργικότητα του *.Net Core Identity*, προκειμένου να διαχειριστούμε και να δημιουργήσουμε τη λειτουργικότητα της ταυτοποίησης εγγεγραμμένων χρηστών στην εφαρμογή μας, την εγγραφή χρηστών στην εφαρμογή μας, καθώς και τη διαχείριση των αιτημάτων ασφαλείας τους. Εκμεταλλευόμενοι τη λειτουργία των *cookies* και των βιβλιοθηκών του *Microsoft.AspNetCore.Identity*, θα σχεδιάσουμε τις απαραίτητες κλάσεις και τις μεθόδους διαχείρισης κι αποθήκευσης των απαραίτητων δεδομένων ασφαλείας της εφαρμογής μας. Ξεκινώντας, εντός του *Core project* και του καταλόγου *Entities*, δημιουργούμε την κλάση *AppUser*, η οποία βασίζεται στην κλάση *Microsoft.AspNetCoreIdentiy*, *IdentityUser* (εικόνα 300). Εντός της κλάσης αυτής προσθέτουμε δύο *optional string properties* με όνομα *FirstName* και *Lastname*.

AppUser.cs × Program.cs

Core > Entities > AppUser.cs > ...

```
1   using Microsoft.AspNetCore.Identity;
2
3   namespace Core.Entities;
4
    3 references
5   public class AppUser : IdentityUser
6   {
        0 references
7       public string? FirstName {get; set;}
        0 references
8       public string? LastName {get; set;}
9   }
```

Εικόνα 300

Ολοκληρώνοντας την παραμετροποίηση ώστε η εφαρμογή μας να έχει πρόσβαση στις βιβλιοθήκες του *Microsoft.AspNetCore.Identity*, μεταβαίνουμε στο αρχείο *Program.cs*, εντός του *API project*. Εκεί ορίζουμε τις απαραίτητες *services* και *extensions*, προκειμένου να αποκτήσουμε πρόσβαση στα *identity endpoints* (εικόνα 301).

```
37  //identity services
38  builder.Services.AddAuthorization();
39  builder.Services.AddIdentityApiEndpoints<AppUser>()
40      .AddEntityFrameworkStores<StoreContext>();
41
42
43
44  var app = builder.Build();
45
46  // Configure the HTTP request pipeline.
47  app.UseMiddleware<ExceptionMiddleware>(); //handling
48  //allows to send data to client throw ANGULAR port on
49  app.UseCors(x => x.AllowAnyHeader().AllowAnyMethod().
50  app.MapControllers();
51
52  //identity mapping
53  app.MapControllers();
54  app.MapIdentityApi<AppUser>();
```

Εικόνα 301

10.2 Δημιουργία του Register Endpoint

Ξεκινώντας, εντός του καταλόγου *DTOs*, στο *API project*, δημιουργούμε μία νέα κλάση με τίτλο *RegisterDto*. Εντός τις κλάσης αυτής ορίζουμε τα απαραίτητα πεδία χρήστη, τα οποία απαιτούνται για τη διαδικασία του *registration* (εικόνα 302).

RegisterDto.cs ● AccountController.cs

API > DTOs > RegisterDto.cs > ...

```
1   using System;
2   using System.ComponentModel.DataAnnotations;
3
4   namespace API.DTOs;
5
    1 reference
6   public class RegisterDto
7   {
        1 reference
8       [Required] public string FirstName { get; set; } = string.Empty;
        1 reference
9       [Required] public string LastName { get; set; } = string.Empty;
        2 references
10      [Required] public string Email { get; set; } = string.Empty;
        1 reference
11      [Required] public string Password { get; set; } = string.Empty;
12  }
```

Εικόνα 302

Στη συνέχεια, εντός του καταλόγου *Controllers*, δημιουργούμε έναν νέο *C# controller* με τίτλο *AccountController*. Η κλάση *AccountController* βασίζεται στην αντίστοιχη *BaseApiController* της εφαρμογής μας, καθώς επίσης κάνει *inject* τις απαραίτητες *Identity* κλάσεις, *SignInManager* και *AppUser*.

Ακολούθως δημιουργούμε μία *HttpPost Task* μέθοδο με όνομα *Register*, η οποία δέχεται ως όρισμα μία μεταβλητή τύπου *RegisterDto*, την οποία και ορίσαμε προηγουμένως. Ο χρήστης δίνει όλες τα απαραίτητες παραμέτρους κι εφόσον πληρούνται τα κριτήρια, μέσω των *.Net Core* και *.Net Core Identiy* κλάσεων και μεθόδων (*SignInManager*, *UserManager*, *CreateAsync*), δημιουργείται μία εγγραφή χρήστη (εικόνα 303).

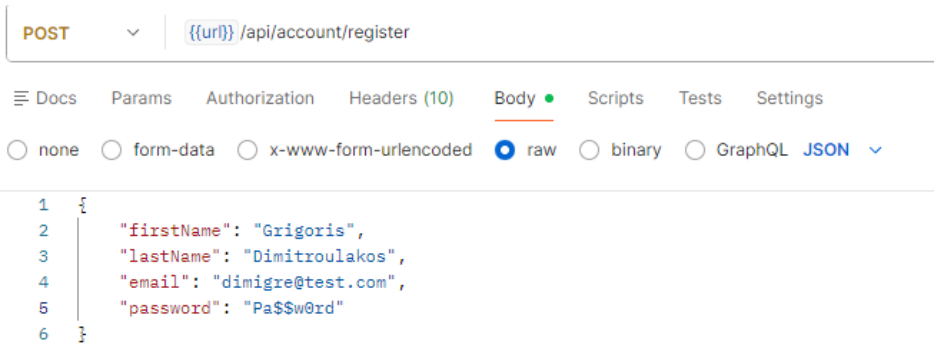
AccountController.cs

API > Controllers > AccountController.cs > AccountController > AccountController

```
1  using API.DTOs;
2  using Core.Entities;
3  using Microsoft.AspNetCore.Authorization;
4  using Microsoft.AspNetCore.Identity;
5  using Microsoft.AspNetCore.Mvc;
6
7  namespace API.Controllers;
8
9  //injecting the signinmanagers from microsoft identity to manage the users
   0 references
10 public class AccountController(SignInManager<AppUser> signInManager) : BaseApiController
11 {
12     [HttpPost("register")] // api/regsiter
       0 references
13     public async Task<ActionResult> Register(RegisterDto registerDto)//access properties of Re
14     {
15         var user = new AppUser //create new user
16         {
17             FirstName = registerDto.FirstName,
18             LastName = registerDto.LastName,
19             Email = registerDto.Email,
20             UserName = registerDto.Email
21         };
22         //saves the result of registration of new user
23         var result = await signInManager.UserManager.CreateAsync(user, registerDto.Password);
24
25         if (!result.Succeeded)
26         {
27             return BadRequest(result.Errors);
28         }
29
30         return Ok();
31     }
32 }
```

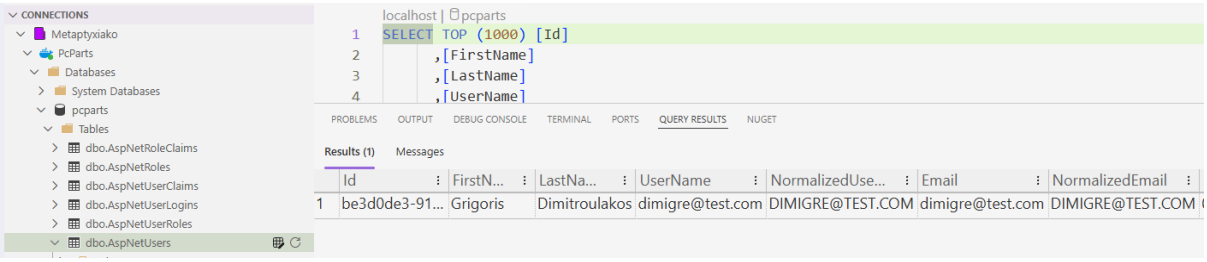
Εικόνα 303

Δοκιμάζοντας ένα HTTP post αίτημα στο περιβάλλον της εφαρμογής *Postman*, χρησιμοποιώντας το *URL*: *api/account/register*, εισάγουμε μία εγγραφή χρήστη στη βάση δεδομένων *pcaparts.db*, μέσω του *register endpoint* (εικόνα 304).



Εικόνα 304

Η νέα εγγραφή χρήστη φιλοξενείται στη βάση δεδομένων *pcparts.db* και συγκεκριμένα στον πίνακα *dbo.AspNetUsers* (εικόνα 305) . Όλοι οι πίνακες *AspNet* δημιουργούνται μέσω του *dotnet migrations system*, ύστερα από την προσθήκη των παραμέτρων (*Identity services* και *MapIdentityApi*) στην κλάση *Program*.



Εικόνα 305

10.3 Προσθήκη των User Endpoints

Συνεχίζοντας την προσθήκη *identity endpoints* εντός του *AccountController.cs* αρχείου, θα δημιουργήσουμε έναν νέο κατάλογο εντός του *API project*, με τίτλο *Exntensions*. Εκεί ορίζουμε μία νέα *public static* κλάση με όνομα *ClaimsPrincipleExtensions*. Στη συγκεκριμένη κλάση προσθέτουμε δύο νέες μεθόδους, την *GetEmail* Και την ασύγχρονη *Task*, τύπου *AppUser* μέθοδο *GetUserByEmail*. Η πρώτη μας επιστρέφει το email ενός χρήστη από τη βάση δεδομένων *pcparts.db* και η δεύτερη, εκμεταλλευόμενη τη συγκεκριμένη πληροφορία, επιστρέφει την πληροφορία που περιγράφει τον χρήστη στο *endpoint* που θα την αιτηθεί (εικόνα 306).

ClaimsPrincipleExtensions.cs × AccountController.cs

API > Extensions > ClaimsPrincipleExtensions.cs > ClaimsPrincipleExtensions

```
1  using System;
2  using System.Security.Authentication;
3  using System.Security.Claims;
4  using Core.Entities;
5  using Microsoft.AspNetCore.Identity;
6  using Microsoft.EntityFrameworkCore;
7
8  namespace API.Extensions;
9
10 //all of them static as no new instance of this class is going to be needed
   0 references
11 public static class ClaimsPrincipleExtensions
12 {   //returns user based on the returning email
       1 reference
13     public static async Task<AppUser> GetUserByEmail(this UserManager<AppUser> userManager, ClaimsPrincipal user)
14     {
15         var userToReturn = await userManager.Users.FirstOrDefaultAsync(x =>
16             x.Email == user.GetEmail());
17
18         if (userToReturn == null) throw new AuthenticationException("User not found");
19
20         return userToReturn;
21     }
22
23     //returns user email
       1 reference
24     public static string GetEmail(this ClaimsPrincipal user)
25     {
26         var email = user.FindFirstValue(ClaimTypes.Email)
27             ?? throw new AuthenticationException("Email claim not found");
28         return email;
29     }
30 }
```

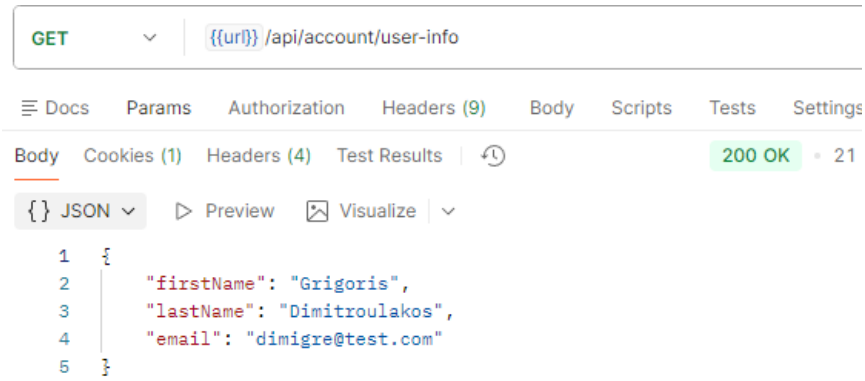
Εικόνα 306

Βασιζόμενοι στην κλάση *ClaimsPrincipleExtensions*, εντός του *Account Controller* δημιουργούμε ένα νέο *HttpGet endpoint* με τίτλο *GetUserInfo*, το οποίο παρουσιάζεται στην εικόνα 307.

```
44     //endpoint that returns user info for logged in user to client app
45     [HttpGet("user-info")]
       0 references
46     public async Task<ActionResult> GetUserInfo()
47     {
48         if (User.Identity?.IsAuthenticated == false) return NoContent();
49
50         var user = await signInManager.UserManager.GetUserByEmail(User);
51
52         return Ok(new
53         {
54             user.FirstName,
55             user.LastName,
56             user.Email,
57         });
58     }
```

Εικόνα 307

Δοκιμάζοντας τη λειτουργία του νέου *endpoint* στο περιβάλλον της εφαρμογής *Postman*, διαπιστώνουμε την ορθή λειτουργία του, καθώς μας επιστρέφεται η πληροφορία του χρήστη ο οποίος φιλοξενείται στη βάση δεδομένων της εφαρμογής μας.



Εικόνα 308

Ακολουθεί η προσθήκη δύο ακόμη *endpoints* εντός του *Account Controller*, με τίτλο *GetAuthState* (εικόνα 309) και *Logout* (εικόνα 310) αντίστοιχα. Το πρώτο, μας επιστρέφει την πληροφορία για το αν ένας χρήστης είναι ενεργά συνδεδεμένος στη βάση δεδομένων της εφαρμογής μας, μία δεδομένη χρονική στιγμή.

```
//endpoint which returns true if user is authenticated
[HttpGet("auth-status")]
0 references
public ActionResult GetAuthState()
{
    return Ok(new
    {   //? cause we may not have the user loged in
        IsAuthenticated = User.Identity?.IsAuthenticated ?? false
    });
}
```

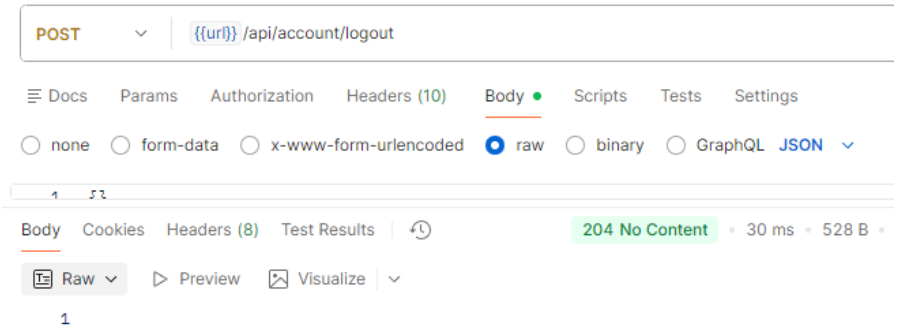
Εικόνα 309

Το δεύτερο, εφόσον ο χρήστης είναι σε κατάσταση «*Loged In*», τον αποσυνδέει από τη βάση δεδομένων *pcparts.db*.

```
[Authorize] //authorization required
[HttpPost("logout")] //in order to log out
0 references
public async Task<ActionResult> Logout()
{
    await signInManager.SignOutAsync();
    return NoContent();
}
```

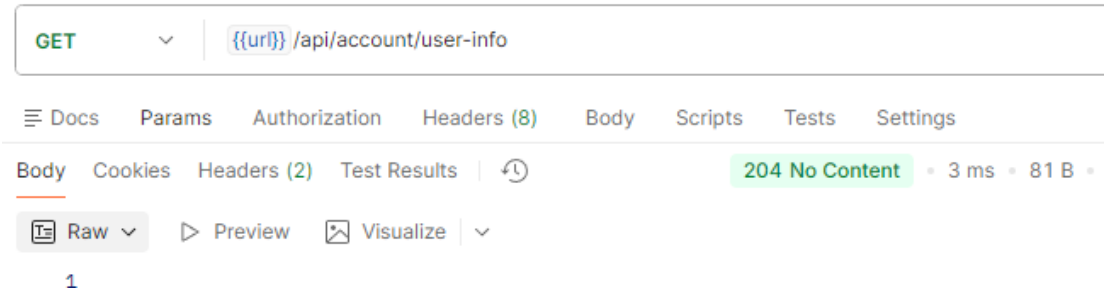
Εικόνα 310

Δοκιμάζοντας την προσπέλαση του *Logout endpoint* στην εφαρμογή *Postman*, λαμβάνουμε ως απάντηση «*204 No Content*», πράγμα που σημαίνει πως η ενεργή συνεδρία του χρήστη με τη βάση δεδομένων έχει ολοκληρωθεί (εικόνα 311).



Εικόνα 311

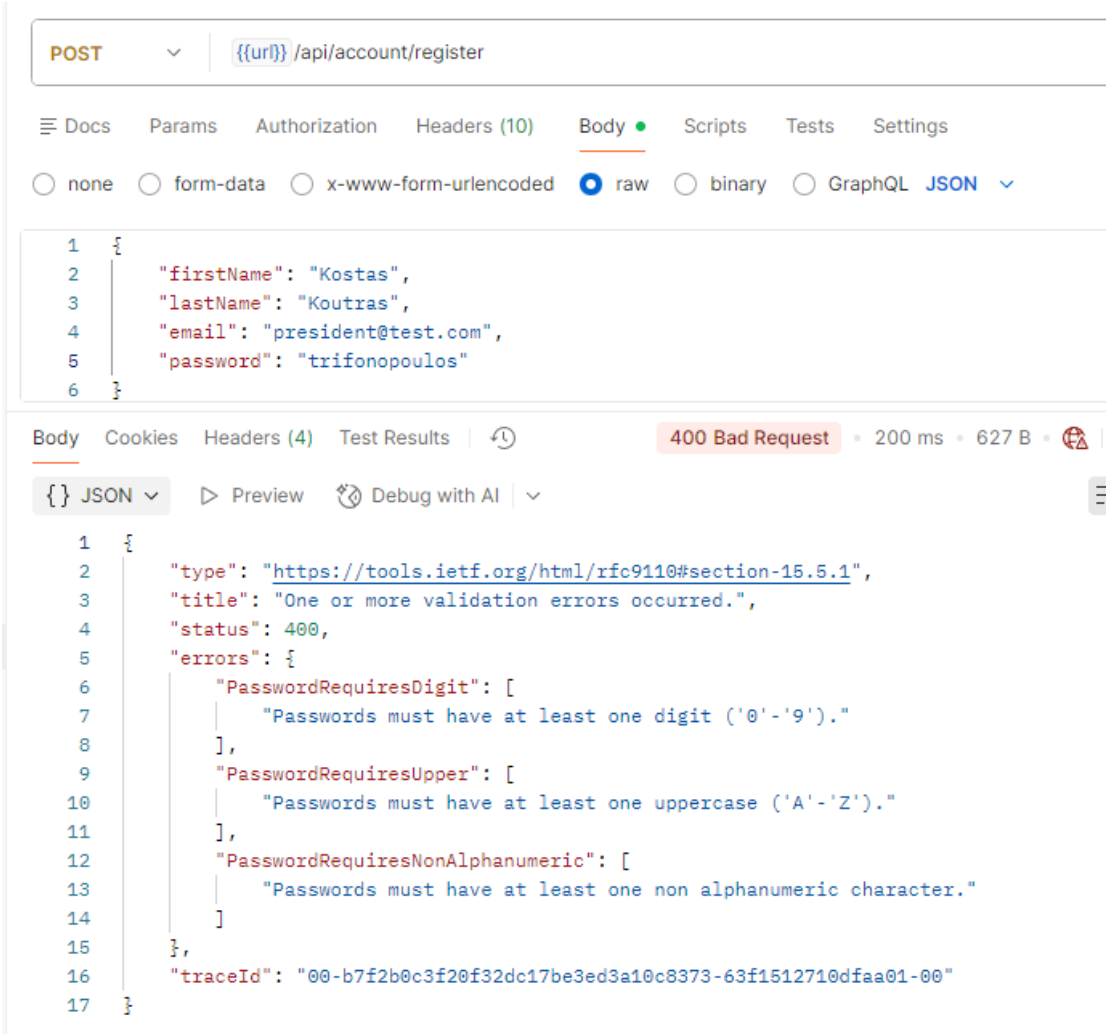
Εφόσον ο χρήστης έχει αποσυνδεθεί, δοκιμάζουμε εκ νέου να προσπελάσουμε το *endpoint GetUserInfo*. Το αποτέλεσμα που μας επιστρέφεται είναι το ίδιο με πριν. Ο χρήστης δεν έχει κάποια ενεργεί συνεδρία στη βάση δεδομένων της εφαρμογής μας, επομένως είναι *unauthorized* και δεν μας επιστρέφεται η ζητούμενη πληροφορία σχετικά με τα στοιχεία του (εικόνα 312).



Εικόνα 312

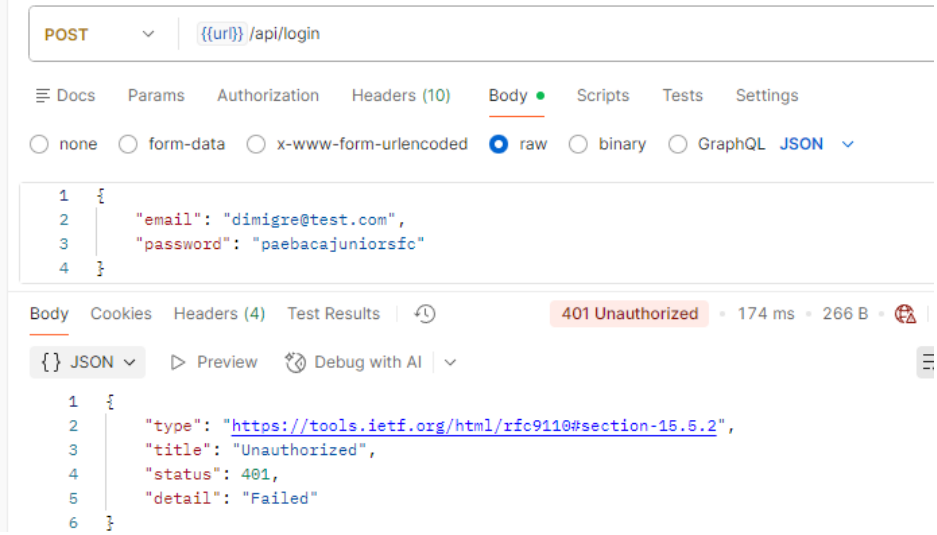
10.4 Σφάλματα Ταυτοποίησης

Κάθε φορά που ένας χρήστης επιθυμεί την εγγραφή του στη βάση δεδομένων μίας *web* εφαρμογής, θα πρέπει να έχει υπόψιν του ορισμένα κριτήρια. Τα απαραίτητα πεδία ορθής συμπλήρωσης (όνομα, επώνυμο, διεύθυνση, email, κωδικός πρόσβασης) και τα κριτήρια ασφαλείας, σχετικά με τη δημιουργία ενός ασφαλούς κωδικού εισόδου, είναι μερικά από αυτά. Το *Microsoft .Net.Core Identity* παρέχει στον προγραμματιστή έτοιμα, όλα τα απαραίτητα εργαλεία για την ασφαλή διαχείριση δεδομένων που αφορούν τους χρήστες. Επομένως, προσπαθώντας να εγγράψουμε έναν νέο χρήστη στη βάση δεδομένων μας, χρησιμοποιώντας έναν «αδύναμο» κωδικό ασφαλείας, θα λάβουμε το αποτέλεσμα της εικόνας 313.



Εικόνα 313

Επιπροσθέτως, κάνοντας μία προσπάθεια να συνδεθούμε με τα στοιχεία του μοναδικού εγγεγραμμένου χρήστη στη βάση δεδομένων μας, χρησιμοποιώντας όμως λανθασμένο κωδικό πρόσβασης, λαμβάνουμε τον μήνυμα της εικόνας 314Α, το οποίο αφορά τις περιπτώσεις σφαλμάτων «*401 Unauthorized Access*».



Εικόνα 314Α

Έχοντας την ανάγκη να επιστρέφουμε τα συγκεκριμένα μηνύματα στον χρήστη, μέσω της *Angular - client* εφαρμογής μας, θα μεταβούμε στον *Account Controller* κι εντός του *Register endpoint*, θα χρησιμοποιήσουμε την οντότητα *ModelState*. Μέσω ενός *foreach loop* αποτυπώνουμε όλα τα στιγμιότυπα εκτέλεσης του μοντέλου μας και στην περίπτωση σφάλματος, αυτό μεταφέρεται στη μέθοδο *ValidationProblem*, η οποία επιστρέφει όλα τα σφάλματα τύπου «*400 Client Errors*» στην *client* εφαρμογή μας (εικόνα 314Β).

```
13  public class AccountController(SignInManager<AppUser> signInManager) : BaseApiController
14  {
15      [HttpPost("register")] // api/regsiter
        0 references
16      public async Task<ActionResult> Register(RegisterDto registerDto)//access properties of Re
17      {
18          var user = new AppUser //create new user
19          {
20              FirstName = registerDto.FirstName,
21              LastName = registerDto.LastName,
22              Email = registerDto.Email,
23              UserName = registerDto.Email
24          };
25          //saves the result of registration of new user
26          var result = await signInManager.UserManager.CreateAsync(user, registerDto.Password);
27
28          if (!result.Succeeded)
29          {
30              foreach (var error in result.Errors)
31              {
32                  ModelState.AddModelError(error.Code, error.Description);
33              }
34
35              return ValidationProblem();
36          }
37
38          return Ok();
39      }
```

Εικόνα 314Β

10.5 Προσθήκη της Οντότητας Address

Θέλοντας να διανθίσουμε την πληροφορία χρήστη εντός της βάσης δεδομένων *pcararts.db*, θα δημιουργήσουμε μία νέα *C# entity* στον αντίστοιχο κατάλογο *Entities*, του *Core project*. Την κλάση αυτή την ονομάζουμε *Address* και ο κώδικας που την περιγράφει παρουσιάζεται στην εικόνα 315. Οι *properties* τις οποίες ορίζουμε, αφορούν τη διεύθυνση, την πόλη, την περιοχή, τον ταχυδρομικό κώδικα και τη χώρα του χρήστη, οντότητες που θα μας χρησιμεύσουν για την ολοκλήρωση της εκάστοτε παραγγελίας.

Address.cs × AppUser.cs StoreContext.cs

Core > Entities > Address.cs > Address

```
1  namespace Core.Entities;
2
   2 references
3  public class Address : BaseEntity //derives from ba
4  {
       0 references
5      public required string Line1 { get; set; }
       0 references
6      public string? Line2 { get; set; }
       0 references
7      public required string City { get; set; }
       0 references
8      public required string State { get; set; }
       0 references
9      public required string PostalCode { get; set; }
       0 references
10     public required string Country { get; set; }
11 }
```

Εικόνα 315

Ακολούθως, στην κλάση *AppUser* κάνουμε *implement* την κλάση *Address*, ορίζοντας μία νέα *optional property* τύπου *Address* (εικόνα 316).

Address.cs AppUser.cs × StoreContext.cs

Core > Entities > AppUser.cs > ...

```
1  using Microsoft.AspNetCore.Identity;
2
3  namespace Core.Entities;
4
   7 references
5  public class AppUser : IdentityUser
6  {
       2 references
7      public string? FirstName {get; set;}
       2 references
8      public string? LastName {get; set;}
       0 references
9      public Address? Address {get; set;}
10 }
11
```

Εικόνα 316

Τέλος, εντός της κλάσης *StoreContext* θα προσθέσουμε την οντότητα της εικόνας 317, προκειμένου να έχουμε τη δυνατότητα υποβολής *queries* προς τη βάση δεδομένων, αναφορικά με τον πίνακα *Addresses* ο οποίος δημιουργείται μέσω του *dotnet migration system*.

```
16     //have access to address in our db during migr
       0 references
17     public DbSet<Address> Addresses { get; set; }
```

Εικόνα 317

Ξεκινώντας την παραμετροποίηση της εφαρμογής μας, προκειμένου να αποθηκεύονται τα στοιχεία διεύθυνσης χρήστη στη βάση δεδομένων *pcparts.db*, δημιουργούμε μία νέα κλάση με όνομα *AddressDto*, εντός του καταλόγου *DTOs*, του *API project* (εικόνα 318). Η συγκεκριμένη κλάση περιλαμβάνει όλα τα πεδία της κλάσης *Address*, αρχικοποιώντας τις *required properties* με μία κενή συμβολοσειρά.

AddressDto.cs

API > DTOs > AddressDto.cs > ...

```
1   using System.ComponentModel.DataAnnotations;
2
3   namespace API.DTOs;
4
    0 references
5   public class AddressDto
6   {
        0 references
7       [Required] public string Line1 { get; set; } = string.Empty;
        0 references
8       public string? Line2 { get; set; }
        0 references
9       [Required] public string City { get; set; } = string.Empty;
        0 references
10      [Required] public string State { get; set; } = string.Empty;
        0 references
11      [Required] public string PostalCode { get; set; } = string.Empty;
        0 references
12      [Required] public string Country { get; set; } = string.Empty;
13  }
14
```

Εικόνα 318

Ακολούθως ορίζουμε τη μέθοδο *GetUserByEmailWithAddress*, η οποία μας επιστρέφει την πληροφορία του χρήστη βασιζόμενη στο *email* εγγραφής του (όπως και η *GetUserByEmail* μέθοδος, εικόνα 306), με τη διαφορά ότι μας επιστρέφεται και η πληροφορία που αφορά τα στοιχεία της οντότητας *Address*. Τη νέα αυτή μέθοδο την ορίζουμε εντός της κλάσης *ClaimsPrincipleExtensions* (εικόνα 319).

```
    //returns user info + address based on user email
    2 references
    public static async Task<AppUser> GetUserByEmailWithAddress(this UserManager<AppUser> userManager, ClaimsPrincipal user)
    {
        var userToReturn = await userManager.Users
            .Include(x => x.Address) //AppUser propertie
            .FirstOrDefaultAsync(x => x.Email == user.GetEmail());
        if (userToReturn == null) throw new AuthenticationException("User not found");

        return userToReturn;
    }
```

Εικόνα 319

Στη συνέχεια, δημιουργούμε μία νέα *extension* κλάση με όνομα *AddressMappingExtension*, εντός του καταλόγου *Extensions*, του *API project*. Ο ρόλος αυτής της κλάσης είναι να ενημερώνει αμφίδρομα τις οντότητες *Address* και *AddressDto* κάθε φορά που ο χρήστης επιθυμεί την εισαγωγή ή την ενημέρωση των παραμέτρων του αχρείου διευθύνσεώς του. Στην εικόνα 320 παρουσιάζεται ο κώδικας της νέα *static* κλάσης *AddressMappingExtension*. Εντός του σώματος της *AddressMappingExtensions class*, ορίζουμε τη στατική μέθοδο *AdressDto* με τίτλο *ToDto* και όρισμα τη μεταβλητή *address* τύπου *Address*.

Με αυτόν τον τρόπο η πληροφορία της οντότητας *Address*, μεταβιβάζεται στις *properties* της οντότητας *AddressDto*. Επόμενη, είναι η *static* τύπου *Address* μέθοδος *ToEntity*, η οποία εκτελεί το ακριβώς αντίθετο δρομολόγιο συγκριτικά με τη μέθοδο *ToDto*. Οι *properties* της οντότητας *Address*, ενημερώνονται από τις *properties* της *extension* κλάσης *AddressDto*. Τέλος, ορίζεται η μέθοδος *UpdateFromDto*, η οποία ενημερώνει με νέες εγγραφές τις *properties* της οντότητας *Address*, εγγραφές που παρέχονται από τις *properties* της κλάσης *AddressDto*.

AddressMappingExtensions.cs

API > Extensions > AddressMappingExtensions.cs > AddressMappingExtensions

```
using API.DTOs;
using Core.Entities;

namespace API.Extensions;

0 references
public static class AddressMappingExtensions
{
    2 references
    public static AddressDto? ToDto(this Address? address)
    {
        if (address == null) return null;

        return new AddressDto
        {
            Line1 = address.Line1,
            Line2 = address.Line2,
            City = address.City,
            State = address.State,
            Country = address.Country,
            PostalCode = address.PostalCode
        };
    }

    1 reference
    public static Address ToEntity(this AddressDto addressDto)
    {
        if (addressDto == null) throw new ArgumentNullException(nameof(addressDto));

        return new Address
        {
            Line1 = addressDto.Line1,
            Line2 = addressDto.Line2,
            City = addressDto.City,
            State = addressDto.State,
            Country = addressDto.Country,
            PostalCode = addressDto.PostalCode
        };
    }

    1 reference
    public static void UpdateFromDto(this Address address, AddressDto addressDto)
    {
        if (addressDto == null) throw new ArgumentNullException(nameof(addressDto));
        if (address == null) throw new ArgumentNullException(nameof(address));

        address.Line1 = addressDto.Line1;
        address.Line2 = addressDto.Line2;
        address.City = addressDto.City;
        address.State = addressDto.State;
        address.Country = addressDto.Country;
        address.PostalCode = addressDto.PostalCode;
    }
}
```

Εικόνα 320

Τελευταίος σταθμός πριν τον έλεγχο της λειτουργικότητας των πεπραγμένων μας, αποτελεί η κλάση *AccountController*. Στην εικόνα 321, παρουσιάζεται η χρήση της μεθόδου *GetUserByEmailWithAddress*, εντός του *GetUserInfo endpoint*, αντί της μεθόδου *GetUserByEmail*.

```
//endpoint that returns user info for logged in user to client app
[HttpGet("user-info")]
0 references
public async Task<ActionResult> GetUserInfo()
{
    if (User.Identity?.IsAuthenticated == false) return NoContent();

    var user = await signInManager.UserManager.GetUserByEmailWithAddress(User);

    return Ok(new
    {
        user.FirstName,
        user.LastName,
        user.Email,
        Address = user.Address?.ToDto()//optional
    });
}
```

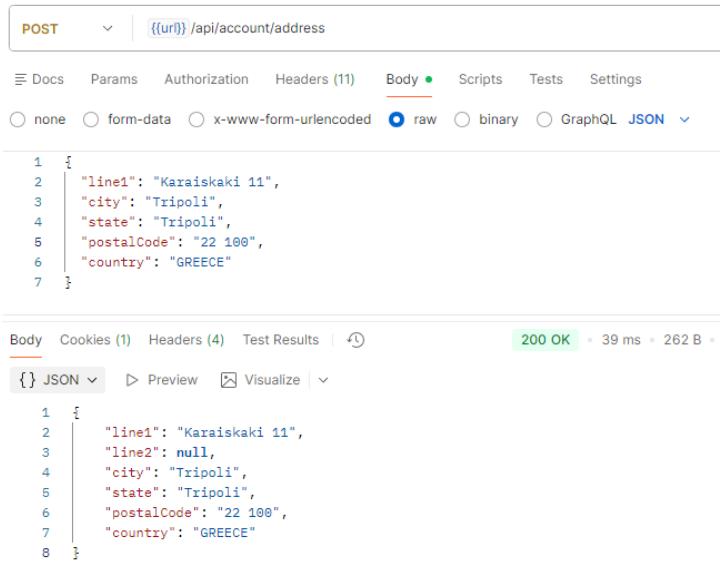
Εικόνα 321

Επιπλέον, ορίζουμε το νέο *endpoint CreateOrUpdateAddress*, με το οποίο έχουμε τη δυνατότητα ενημέρωσης της βάσης δεδομένων της εφαρμογής μας, αναφορικά με τα στοιχεία διεύθυνσης ενός χρήστη. Στην περίπτωση που τα στοιχεία διεύθυνσης δεν υφίστανται, καλείται η μέθοδος *ToEntity* και ενημερώνεται ανάλογα ο αντίστοιχος πίνακας στη βάση δεδομένων. Αντίστοιχη είναι και η λειτουργία της μεθόδου *UpdateFromDto*, με τη διαφορά ότι την καλούμε στην περίπτωση ενημέρωσης των υπαρχόντων στοιχείων χρήστη στη βάση δεδομένων. Τέλος, μας επιστρέφεται η *address* πληροφορία μέσω της μεθόδου *ToDto*. Στην εικόνα 322 παρουσιάζεται ο κώδικας της νέας ασύγχρονης μεθόδου CreateOrUpdate-Address.

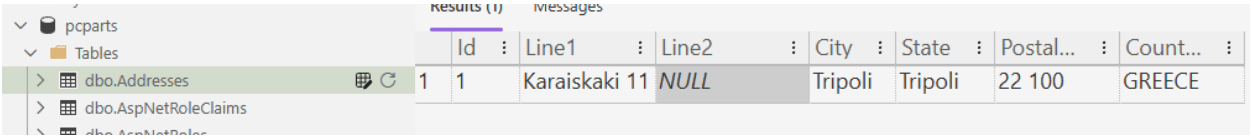
```
[Authorize]//address endpoint
[HttpPost("address")]
0 references
public async Task<ActionResult<Address>> CreateOrUpdateAddress(AddressDto addressDto)
{
    var user = await signInManager.UserManager.GetUserByEmailWithAddress(User);

    if (user.Address == null) //new address entry
    {
        user.Address = addressDto.ToEntity();
    }
    else
    {   //existing address in db update method
        user.Address.UpdateFromDto(addressDto);
    }

    var result = await signInManager.UserManager.UpdateAsync(user);

    if (!result.Succeeded) return BadRequest("Problem updating user address!");

    return Ok(user.Address.ToDto());//returns the address info to user
}
```

Εικόνα 322

Έχοντας ολοκληρώσει την προσθήκη του *CreateOrUpdateAddress endpoint*, μεταφερόμαστε στο περιβάλλον της εφαρμογής Postman. Εκτελώντας ένα *HTTP* Post request προς το CreateOrUpdateAddress endpoint, μέσω του *URL*: https://localhost:5001/api/account/address, συμπληρώνοντας χειροκίνητα τα στοιχεία διεύθυνσης του μοναδικού μας *registered* χρήστη, λαμβάνουμε το αποτέλεσμα της εικόνας 323. Η εισαγωγή των στοιχείων διεύθυνσης στη βάση δεδομένων πραγματοποιήθηκε επιτυχώς (εικόνα 324).

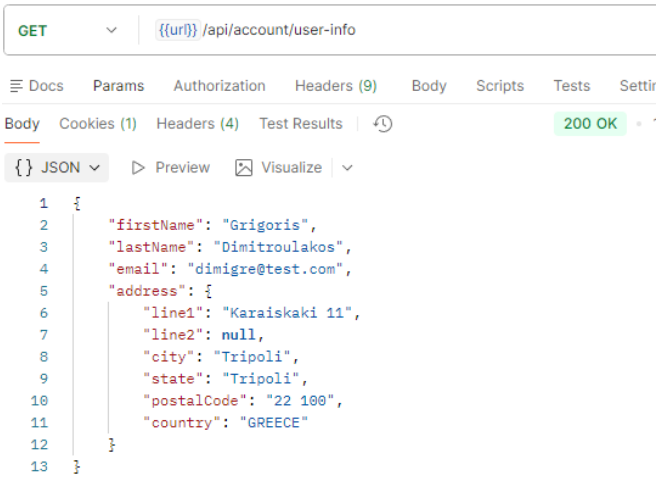


Εικόνα 323



Εικόνα 324

Εκτελώντας ένα *HTTP Get request* προς το *endpoint GetUserInfo*, μέσω του *URL*: https://localhost:5001/api/account/user-info, μας επιστρέφεται ολοκληρωμένη η πληροφορία του εγγεγραμμένου χρήστη από τη βάση δεδομένων *pcparts.db* (εικόνα 325).



Εικόνα 325

Κεφάλαιο 11. Angular – Identity

Στο τρέχον κεφάλαιο θα μελετήσουμε τη δημιουργία των φορμών εγγραφής, σύνδεσης και αποσύνδεσης ενός χρήστη, στη βάση δεδομένων της εφαρμογής μας. Πρακτικά θα ασχοληθούμε με την υλοποίηση των λειτουργιών *Register*,*Login* και *Logout*, κάνοντας χρήση των *Angular reactive forms* (*form builder attributes*), των *Angular guard components*, καθώς και του *Angular client validation* ώστε να ενημερώνεται ο χρήστης για λανθασμένη εισαγωγή στοιχείων στις ανάλογες φόρμας.

11.1 Δημιουργία της Account Service

Αρχικά, ξεκινάμε με την εκτέλεση των εντολών της εικόνας 326, στο τερματικό του *Visula Studio Code*, εντός του καταλόγου *client*, της *Angular* εφαρμογής μας. Με αυτόν τον τρόπο δημιουργούνται, η *Account Angular service*, τα *login* και τα *register components*.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  QUERY RESULTS  NUGET

PS C:\Users\USER\Desktop\pcparts> cd client
PS C:\Users\USER\Desktop\pcparts\client> ng g s core/services/account --skip-tests
CREATE src/app/core/services/account.service.ts (127 bytes)
PS C:\Users\USER\Desktop\pcparts\client> ng g c features/account/login --skip-tests
CREATE src/app/features/account/login/login.component.ts (223 bytes)
CREATE src/app/features/account/login/login.component.scss (0 bytes)
CREATE src/app/features/account/login/login.component.html (21 bytes)
PS C:\Users\USER\Desktop\pcparts\client> ng g c features/account/register --skip-tests
CREATE src/app/features/account/register/register.component.ts (235 bytes)
CREATE src/app/features/account/register/register.component.scss (0 bytes)
CREATE src/app/features/account/register/register.component.html (24 bytes)
PS C:\Users\USER\Desktop\pcparts\client>
```

Εικόνα 326

Ακολούθως, θα μεταβούμε στο αρχείο *app.routes.ts*, για να δημιουργήσουμε δύο νέα *paths* που θα ανακατευθύνουν τον χρήστη, αναλόγως την περίσταση, στο κάθε ένα από τα νέα δύο *components* που δημιουργήσαμε προηγουμένως (εικόνα 327).

```
24    { path: 'account/login', component: LoginComponent},
25    { path: 'account/register', component: RegisterComponent },
```

Εικόνα 327

Το κυρίως κομμάτι της υλοποίησής μας εκκινεί από την *Angular* κλάση *AccountService*. Ορίζουμε τις μεταβλητές *baseUrl* και *http*, κάνοντας *inject* την *Angular* κλάση *HttpClient*. Δημιουργούμε την *Angular signal* μεταβλητή *currentUser*, η οποία είναι τύπου *User* (εικόνα 328).

user.ts

client > src > app > shared > models > user.ts > ...

```
1  export type User = {
2    firstName: string;
3    lastName: string;
4    email: string;
5    address: Address;
6  }
7
8  export type Address = {
9    line1: string;
10   line2?: string;
11   city: string;
12   state: string;
13   country: string;
14   postalCode: string;
15 }
```

Εικόνα 328

Στη συνέχεια, ορίζουμε τις μεθόδους *login, register, getUserInfo, logout και updateAddress*. Η μέθοδος *login* αποστέλλει ένα *HTTP post* αίτημα προς το *.Net.Core Identity Login endpoint*, στον *API server*, με τα στοιχεία σύνδεσης του χρήστη (*email-username* και κωδικό) και μας επιστρέφει την ανάλογη απάντηση. Η μέθοδος *register*, έχει παρόμοια λειτουργία με αυτήν της μεθόδου *login*, με τη διαφορά πως αποστέλλονται τα στοιχεία για την πρώτη εγγραφή ενός χρήστη στη βάση δεδομένων μας, στο *endpoint Register*, του *AccountController*. Στην ίδια λογική κινούνται και οι μέθοδοι *logout*, *updateAddress* και *getAuthState*, στέλνοντας τα κατάλληλα *HTTP requests* στα αντίστοιχα *endpoints* του *Account Controller*, για την αποσύνδεση του τρέχοντος συνδεδεμένου χρήστη, την ενημέρωση των στοιχείων διεύθυνσής του, αλλά και για τον έλεγχο της κατάστασης σύνδεσής του στη βάση δεδομένων μας, μία δεδομένη χρονική στιγμή. Όσον αφορά τη μέθοδο *getUserInfo*, κάνοντας χρήση του *Angular Pipe*, δημιουργούμε μία *observable situation* μέσω της *signal* μεταβλητής *currentUser* κι εφόσον έχουμε έναν συνδεδεμένο χρήστη στην πλατφόρμα μας, επιστρέφονται τα στοιχεία του από τον *API server*, μέσω του *GetUserInfo endpoint*. Ο κώδικας της *Account service* παρουσιάζεται στην εικόνα 329.

account.service.ts

client > src > app > core > services > account.service.ts > AccountService

```ts
import { inject, Injectable, signal } from '@angular/core';
import { environment } from '../../../environments/environment';
import { HttpClient, HttpParams } from '@angular/common/http';
import { Address, User } from '../../shared/models/user';
import { map } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class AccountService {
  baseUrl = environment.baseUrl;
  private http = inject(HttpClient);
  currentUser = signal<User | null>(null);

  login(values: any) {
    let params = new HttpParams();
    params = params.append('useCookies', true);
    return this.http.post<User>(this.baseUrl + 'login', values, {params});
  }

  register(values: any) {
    return this.http.post<User>(this.baseUrl + 'account/register', values);
  }

  getUserInfo() {
    return this.http.get<User>(this.baseUrl + 'account/user-info').pipe(
      map(user => {
        this.currentUser.set(user);
        return user;
      })
    );
  }

  logout() {
    return this.http.post(this.baseUrl + 'account/logout', {});
  }

  updateAddress(address: Address) {
    return this.http.post(this.baseUrl + 'account/address', address);
  }
}
```

Εικόνα 329

11.1.1 Login Components

Στην εικόνα 330 παρουσιάζεται ο κώδικας του *login component*. Αρχικά, κάνουμε *inject* τις κλάσεις *AccountService* και *Router*, όπως έχουμε δει και σε προηγούμενα κεφάλαια. Ένα νέο στοιχείο, είναι η χρήση της *private* μεταβλητής *fb*, στην οποία κάνουμε *inject* την κλάση *FormBuilder*. Η συγκεκριμένη κλάση επιτρέπει στα *Angular components* την ανταλλαγή δεδομένων, μέσω του *Angular two-way binding*, όταν ο χρήστης εισάγει στοιχεία σε μία *Angular* φόρμα πεδίων. Στην περίπτωση του *login component*, μέσω της μεταβλητής *fb*, δημιουργούμε μία *loginForm* οντότητα εντός της κλάσης *LoginComponent* με δύο πεδία, *email* και *password*. Έτσι, όταν ο χρήστης εισάγει την απαραίτητη πληροφορία στην *login* φόρμα, καλείται η μέθοδος *onSubmit*, η οποία μέσω της *signal* μεταβλητής *accountService*, και της μεθόδου *login*, δημιουργεί ένα *observable object*. Το συγκεκριμένο αντικείμενο μέσω της μεθόδου *getUserInfo*, λαμβάνει τα δεδομένα που εισήγαγε ο χρήστης και τα αποδίδει σε μορφή *query* στον API *server*. Στην περίπτωση που ο χρήστης επιβεβαιωθεί, μέσω της μεθόδου *navigateByUrl* ανακατευθύνεται στη σελίδα *Shop* της εφαρμογής μας.

auth-guard.ts | login.component.ts

ent > src > app > features > account > login > login.component.ts > LoginComponent

```typescript
import { MatButton } from '@angular/material/button';
import { MatCard } from '@angular/material/card';
import { MatFormField, MatLabel } from '@angular/material/form-field';
import { MatInput } from '@angular/material/input';
import { ActivatedRoute, Router } from '@angular/router';
import { AccountService } from '../../../core/services/account.service';

@Component({
  selector: 'app-login',
  imports: [ReactiveFormsModule, MatCard, MatFormField, MatInput, MatButton, MatLabel],
  templateUrl: './login.component.html',
  styleUrl: './login.component.scss'
})
export class LoginComponent {
  private fb = inject(FormBuilder);
  private accountService = inject(AccountService);
  private router = inject(Router);

  loginForm = this.fb.group({
    email: [''],
    password: ['']
  })

  onSubmit() {
    this.accountService.login(this.loginForm.value).subscribe({
      next: () => {
        this.accountService.getUserInfo().subscribe();
        this.router.navigateByUrl('/shop');
      }
    });
  }
}
```

Εικόνα 330

Τέλος, μεταφερόμαστε στο *html template login* (εικόνα 331). Εδώ, δημιουργούμε μία *Angular mat-card*, η οποία περιέχει τα *attributes form*, *mat-from-field* και *mat-label*. Στην ουσία, πρόκειται για μία φόρμα δύο πεδίων εισαγωγής (*email & password*), τα οποία παρουσιάζονται εντός μίας κάρτας *Angular* σε λευκό φόντο και μέσω των οντοτήτων *loginForm* και *onSubmit* (*Angular two-way binding* μέσω των *attributes [formGroup]* και *(ngSubmit)*), αποστέλλουν την πληροφορία στο *login component*. Επίσης μέσω ενός *routerlink attribute*, κάτω από το *mat-button Sign in*, ο χρήστης έχει τη δυνατότητα να μεταβεί στη σελίδα του *register component*, σε περίπτωση που δεν έχει ήδη ενεργό λογαριασμό.

```
<mat-card class="max-w-lg mx-auto mt-32 p-8 bg-white">
    <form [formGroup]="loginForm" (ngSubmit)="onSubmit()">
        <div class="text-center mb-6">
            <h1 class="text-3xl font-semibold text-primary">Login</h1>
        </div>
        <mat-form-field appearance="outline" class="w-full mb-4">
            <mat-label>Email address</mat-label>
            <input matInput formControlName="email" type="email"
                placeholder="name@example.com" />
        </mat-form-field>
        <mat-form-field appearance="outline" class="w-full mb-6">
            <mat-label>Password</mat-label>
            <input matInput formControlName="password" type="password"
                placeholder="Password" />
        </mat-form-field>
        <button mat-flat-button type="submit" color="primary"
            class="w-full py-2">Sign in</button>
        <a routerLink="/account/register">
            <p class="pt-7 text-shadow-blue-800 underline">Don't have an account? Register here!</p>
        </a>
    </form>
</mat-card>
```

Εικόνα 331

Έχοντας ολοκληρώσει την παραμετροποίηση των *login components*, θα μεταβούμε στα header components προκειμένου να προσαρμόσουμε την παραπάνω λειτουργικότητα. Ξεκινώντας από το αρχείο *header.component.ts* (εικόνα 332), κάνουμε *inject* τις κλάσεις *AccountService* και *Router*. Ορίζουμε τη μέθοδο *logout* και μέσω ενός *observable object* αποστέλλουμε την πληροφορία (τιμή *null*), στο *Logout endpoint* του *Account controller*, εντός του *API project*. Ακολούθως, ο χρήστης ανακατευθύνεται στην αρχική οθόνη (*Home page*), της εφαρμογής μας.

```
accountService = inject(AccountService);//injects accountservice to access signal
private router = inject(Router);//injects router to direct to a page

logout() {
  this.accountService.logout().subscribe({//subscribed to current user signal
    next: () => {
      this.accountService.currentUser.set(null);//sets user to loged out state
      this.router.navigateByUrl('/');//redirects user to home page
    }
  });
}
```

Εικόνα 332

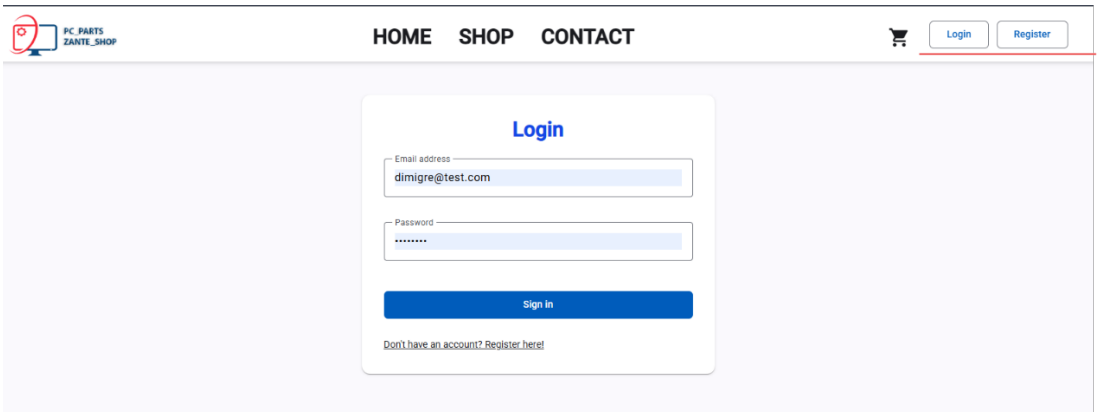
Εν συνεχεία, ανοίγουμε το *header.component.html* αρχείο και προσθέτουμε τον κώδικα της εικόνας 333. Εντός μίας *@if* συνθήκης, ελέγχουμε το περιεχόμενο της *signal* μεταβλητής *currentUser* και στην περίπτωση που δεν είναι *null* (ο χρήστης έχει πραγματοποιήσει επιτυχημένο *login* στην πλατφόρμα μας), στη

θέση των *mat-stroked-buttons Login* και *Register*, παρουσιάζεται το *username* του (δηλαδή το *email* εγγραφής του) κι ένα *drop-down* (*mat-menu*), με τα *routerlink buttons shopping_cart* και *Logout*.

```
@if (accountService.currentUser()) {
  <div class="px-6">
    <button mat-button [matMenuTriggerFor]="menu">
      <mat-icon>arrow_drop_down</mat-icon>
      <span>{{accountService.currentUser()?.email}}</span>
    </button>
    <mat-menu #menu="matMenu" class="px-3">
      <button mat-menu-item routerLink="/cart">
        <mat-icon>shopping_cart</mat-icon>
          My cart
      </button>
      <mat-divider></mat-divider>
      <button mat-menu-item (click)="logout()">
        <mat-icon>logout</mat-icon>
          Logout
      </button>
    </mat-menu>
  </div>
} @else {
      <button mat-stroked-button routerLink="/account/login">Login</button>
      <button mat-stroked-button routerLink="/account/register">Register</button>
    }
```

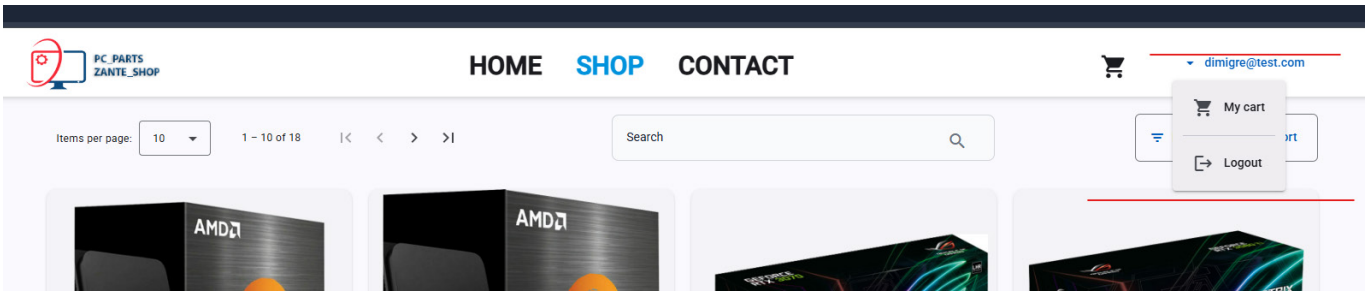
Εικόνα 333

Στην εικόνα 334 παρουσιάζεται η σελίδα *login* της εφαρμογής μας.



Εικόνα 334

Στην εικόνα 335, παρουσιάζεται η αλλαγή στο *header component*, καθώς ο χρήστης έχει εισέλθει επιτυχώς στην πλατφόρμα κι έχει πραγματοποιηθεί ανακατεύθυνση στη σελίδα *Shop*.



Εικόνα 335

11.1.2 Register Components

Έχοντας ορίσει τα *register components* (εικόνα 326), μεταβαίνουμε στο αρχείο *register.component.ts*. Στην εικόνα 336 παρουσιάζεται ο κώδικας της κλάσης *RegisterComponent*. Ακολουθώντας την ίδια λογική με την περίπτωση του *login*, πραγματοποιούμε *inject* των κλάσεων *FormBuilder*, *AccountService* και *Router*. Μία νέα προσθήκη, αποτελεί η *Angular* κλάση *SnackbarService*, η οποία εμφανίζει τα *client side validation errors* στο περιβάλλον του *browser*. Μέσω των *Angular reactive forms* ορίζουμε την οντότητα *registerForm*, η οποία περιέχει όλα τα *required* πεδία που πρέπει να συμπληρωθούν από τον χρήστη, προκειμένου να πραγματοποιηθεί με επιτυχία η εγγραφή του στη βάση δεδομένων *pcparts.db*. Τέλος, ορίζουμε τη μέθοδο *submit*, η οποία μέσω της *object observable accountService*, «παρακολουθεί» την πορεία του *HTTP* αιτήματος για εγγραφή νέου χρήστη προς το *Register endpoint* και βάσει της απάντησης που λαμβάνουμε από τον *API server*, εμφανίζεται μήνυμα επιτυχούς εγγραφής μέσω της *SnacbarService* και πραγματοποιείται ανακατεύθυνση στη σελίδα *login* της εφαρμογής μας.

register.component.ts register.component.html text-input.component.ts text-input.component.html

client > src > app > features > account > register > register.component.ts > ...

```
3  import { MatButton } from '@angular/material/button';
4  import { MatCard } from '@angular/material/card';
5  import { Router } from '@angular/router';
6  import { AccountService } from '../../../core/services/account.service';
7  import { SnackbarService } from '../../../core/services/snackbar.service';
8  import { TextInputComponent } from "../../../shared/components/text-input/text-input.component";
9
10 @Component({
11   selector: 'app-register',
12   imports: [
13     MatCard,
14     ReactiveFormsModule,
15     MatButton,
16     TextInputComponent
17 ],
18   templateUrl: './register.component.html',
19   styleUrl: './register.component.scss'
20 })
21 export class RegisterComponent {
22   private fb = inject(FormBuilder);
23   private accountService = inject(AccountService);
24   private router = inject(Router);
25   private snack = inject(SnackbarService);
26   validationErrors: any[] = [];
27
28   registerForm = this.fb.group({
29     firstName: ['', Validators.required],
30     lastName: ['', Validators.required],
31     email: ['', [Validators.required, Validators.email]],
32     password: ['', [Validators.required]]
33   })
34
35   onSubmit() {
36     this.accountService.register(this.registerForm.value).subscribe({
37       next: () => {
38         this.snack.success('Registration successful - you can now login!');
39         this.router.navigateByUrl('/account/login');
40       },
41       error: err => this.validationErrors = err
42     });
43   }
```

Εικόνα 336

Πριν δημιουργήσουμε τον κώδικα του *register.component.html template*, θα πρέπει να δημιουργήσουμε το *component text-input*, εκτελώντας τις εντολές της εικόνας 337.

```
PS C:\Users\USER\Desktop\pcparts\client> ng g c shared/components/text-input --skip-tests
CREATE src/app/shared/components/text-input/text-input.component.ts (242 bytes)
CREATE src/app/shared/components/text-input/text-input.component.scss (0 bytes)
CREATE src/app/shared/components/text-input/text-input.component.html (26 bytes)
PS C:\Users\USER\Desktop\pcparts\client>
```

Εικόνα 337

Ο ρόλος του αρχείου *text-input.component.ts* (εικόνα 338) είναι ο εξής: Στη σκέψη δημιουργίας μίας νέας *reactive* φόρμας για την εγγραφή χρήστη στην πλατφόρμα μας, θα πρέπει να λάβουμε υπόψιν ενδεχόμενα σφάλματα κατά την εισαγωγή των στοιχείων του. Μερικά παραδείγματα είναι η εισαγωγή «αδύναμων» κωδικών πρόσβασης, η λανθασμένη μορφή *email* καθώς και η παράλειψη συμπλήρωσης απαιτούμενων πεδίων (όνομα, επίθετο κτλ.). Για τον λόγο αυτό, δημιουργούμε το εν λόγω *component* και ουσιαστικά κατασκευάζουμε μία «*custom*» δομή ελέγχου των πεδίων για *reactive Angular forms*, την οποία και θα εφαρμόσουμε στο *register html template*. Μέσω του *text-input component*, αποκτούμε πλήρη πρόσβαση (*Angular* μέθοδοι *@Self* και *control*) σε μία *Angular reactive* φόρμα και στα *attributes* που τη συνθέτουν. Εντός της κλάσης *TextInputComponent*, εμείς το μόνο που κάνουμε είναι να ορίσουμε τα πεδία *(@Input label*, *@Input type*) στα οποία θα εφαρμόσουμε τα κριτήρια ελέγχου κατά την εισαγωγή της πληροφορίας του χρήστη.

text-input.component.ts × text-input.component.html

client > src > app > shared > components > text-input > text-input.component.ts > TextInputComponent

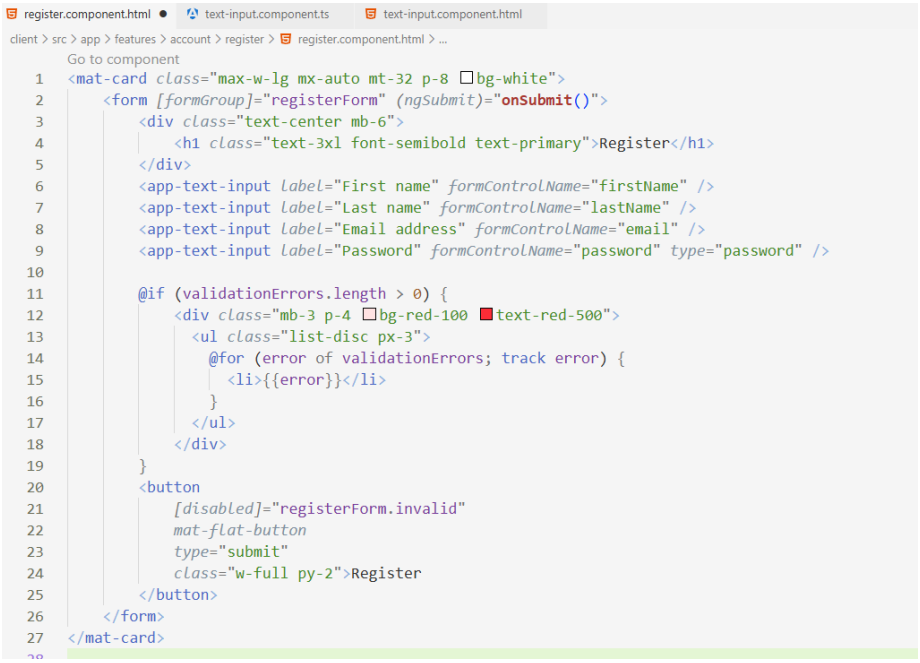
```ts
import { Component, Input, Self } from '@angular/core';
import { FormControl, NgControl, ReactiveFormsModule } from '@angular/forms';
import { MatError, MatFormField, MatLabel } from '@angular/material/form-field';
import { MatInput } from '@angular/material/input';

@Component({
  selector: 'app-text-input',
  imports: [
    MatFormField,
    MatInput,
    MatError,
    MatLabel,
    ReactiveFormsModule
  ],
  templateUrl: './text-input.component.html',
  styleUrl: './text-input.component.scss'
})
export class TextInputComponent {
    @Input() label = '';
    @Input() type = 'text';

    constructor(@Self() public controlDir: NgControl) {
      this.controlDir.valueAccessor = this;
    }

    writeValue(obj: any): void {
    }

    registerOnChange(fn: any): void {
    }

    registerOnTouched(fn: any): void {
    }

    get control() {
      return this.controlDir.control as FormControl;
    }
}
```

Εικόνα 338

Στην εικόνα 339 παρουσιάζεται ο κώδικας του *text-input.component.html template*. Μέσω του *two-way binding* της *Angular*, επιτυγχάνουμε τον έλεγχο της πληροφορίας που εισάγει ο χρήστης, χρησιμοποιώντας τις οντότητες [*type*] , [*formControl*] και {{*label*}}, τις οποίες και δηλώσαμε προηγουμένως στο αντίστοιχο *TypeScript component*. Όταν ορίζουμε μία νέα *reactive form* με τα *attributes* <*app-text-input*>, γίνεται έλεγχος για την εισαγωγή δεδομένων κι αν προκύψει σφάλμα τότε ο χρήστης ενημερώνεται αναλόγως μέσω της ίδιας της της φόρμας, εντός του *browser*.

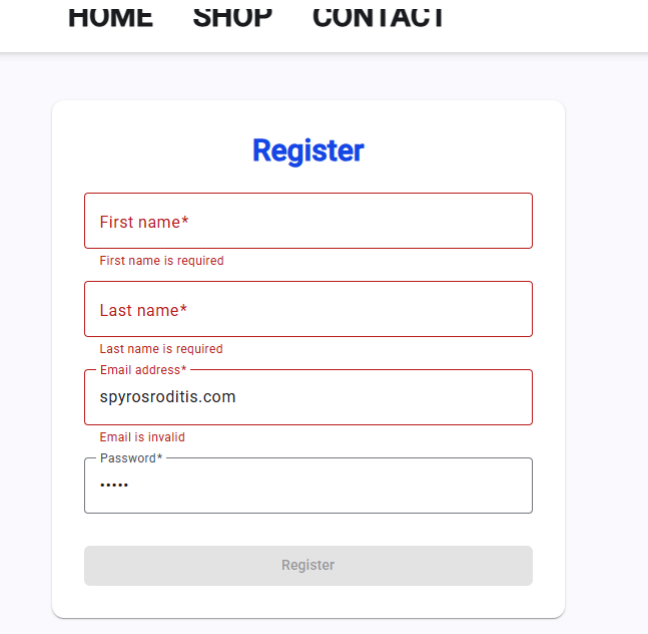
Εικόνα 339

Βασιζόμενοι στη λειτουργικότητα του *text-input component*, δημιουργούμε μία *mat-card* φόρμα για την εγγραφή χρήστη στην πλατφόρμα μας. Τα απαραίτητα πεδία προς έλεγχο είναι: Το όνομα, το επίθετο, το email κι ο κωδικός πρόσβασής του. Όσο τα *forrmControlName* πεδία εντός του <*app-text-input*> *attribute* δεν συμπληρώνονται σωστά, ή παραλείπονται από τον χρήστη, η *@if* δομή απαριθμεί τα σφάλματα, ορίζει το χρώμα του *input cell* σε κόκκινο κι εμφανίζεται η περιγραφή του σφάλματος στον *client*. Άπαξ και συμπληρωθούν επιτυχώς όλα τα πεδία, το *attribute* [*disabled*]=”*registerForm.invalid*” δίνει τιμή *false* και το *mat-flat-button Register* γίνεται διαθέσιμο προς τον χρήστη (*active/clickable*), για να υποβάλλει την εγγραφή του στον *API server*. Η υλοποίηση *του text-input.component.html template*, παρουσιάζεται στην εικόνα 340.



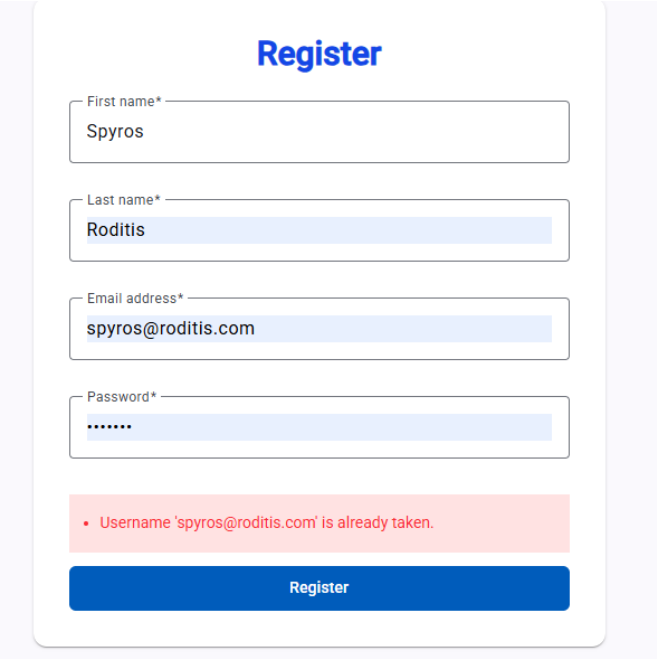
Εικόνα 340

Ενδεικτικά, στις εικόνες 341, 342 και 343 παρουσιάζονται τα παραδείγματα μίας ανεπιτυχούς συμπλήρωσης της φόρμας εγγραφής (*register reactive form*) χρήστη, εντός του φυλλομετρητή μας. Στην πρώτη περίπτωση, έχουμε παράλειψη συμπλήρωσης των πεδίων *firtsName* και *lastName*, ενώ το *email* δεν έχει την επιτρεπόμενη μορφή.



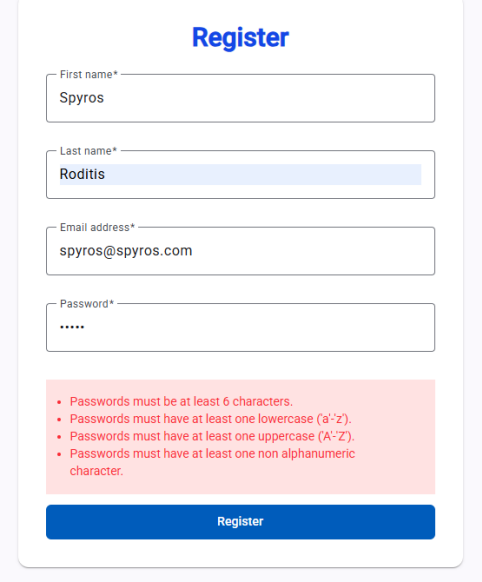
Εικόνα 341

Στη δεύτερη περίπτωση, κατά το πάτημα του κουμπιού «*Register*», εμφανίζεται μήνυμα για την εισαγωγή ενός *username* (email), το οποίο ανήκει ήδη σε άλλον εγγεγραμμένο χρήστη.



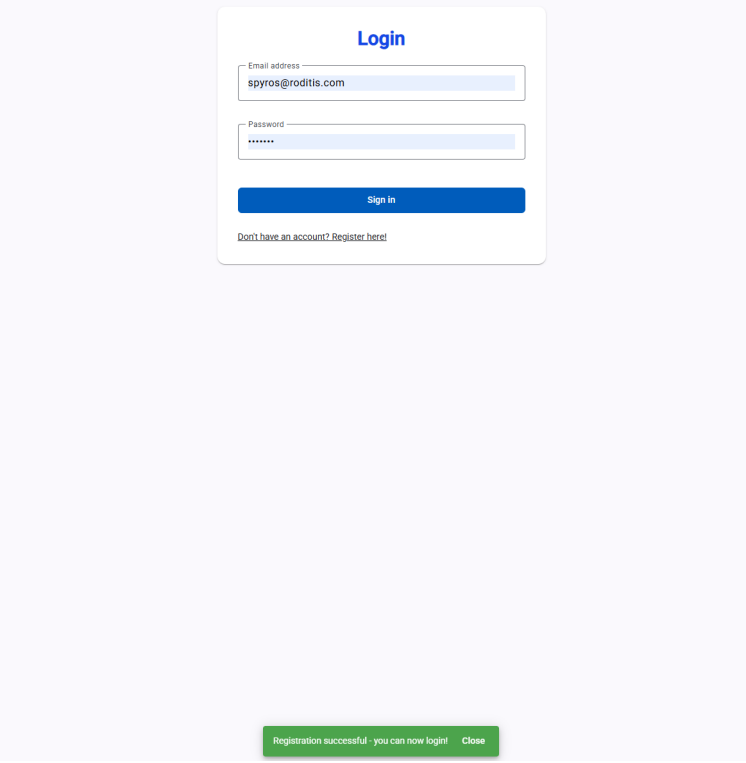
Εικόνα 342

Ενώ στην τρίτη περίπτωση, παρουσιάζεται σφάλμα στο ενδεχόμενο προσπάθειας ενός χρήστη, ο οποίος επιθυμεί να κάνει εγγραφή χρησιμοποιώντας έναν «αδύναμο» κωδικό ασφαλείας.



Εικόνα 343

Στην εικόνα 344 παρουσιάζεται η προσπάθεια μίας επιτυχημένης εγγραφής χρήστη, με το ανάλογο *snackBarService* μήνυμα επιτυχίας από τον *client*, να εμφανίζεται στη σελίδα *Login* της εφαρμογής μας.



Εικόνα 344

11.2 Δημιουργία των Angular Guards

11.2.1 Σελίδα Checkout

Ένα ζήτημα προς επίλυση, αποτελεί η πρόσβαση μη εγγεγραμμένων χρηστών σε σελίδες που δεν είναι επιθυμητό. Αρχικά, θα γίνει λόγος για τη σελίδα του *Checkout*, όταν ο χρήστης έχει επιλέξει τα προϊόντα που επιθυμεί και τα έχει προσθέσει στο καλάθι του, τότε έχει τη δυνατότητα να πατήσει το κουμπί «*Checkout*» (εικόνα 296) και να μεταφερθεί στην ανάλογη σελίδα (εικόνα 297). Στόχος μας είναι να αποτρέψουμε τους μη εγγεγραμμένους χρήστες να μεταβαίνουν στη συγκεκριμένη σελίδα κι αυτό μπορεί να επιτευχθεί μέσω των *Angular guards* κλάσεων. Στην εικόνα 345 παρουσιάζεται η εντολή δημιουργίας του πρώτου *Angular guard* στην *client* εφαρμογή μας, με τίτλο *auth* (εκ του *authenticated*).

```
PS C:\Users\USER\Desktop\pcparts\client> ng g g core/guards/auth --skip-tests
✓ Which type of guard would you like to create? CanActivate
CREATE src/app/core/guards/auth-guard.ts (133 bytes)
PS C:\Users\USER\Desktop\pcparts\client>
```

Εικόνα 345

Μία *Angular guard* κλάση, όπως η *authGuard* (εικόνα 346), μας επιτρέπει να παρακολουθούμε μέσω ενός *pipe observable object*, την τρέχουσα κατάσταση σύνδεσης του χρήστη, εκμεταλλευόμενη τη μέθοδο *getAuthState* στην κλάση *AccountService*. Εφόσον ο τρέχων χρήστης δεν έχει πραγματοποιήσει *login* ή δεν έχει εγγραφεί στην πλατφόρμα μας, στην προσπάθειά του να ολοκληρώσει την πιθανή αγορά του πατώντας στο κουμπί «*Checkout*», ανακατευθύνεται στη σελίδα *Login*. Μόλις ο χρήστης κάνει είσοδο στην πλατφόρμα μας, ανακατευθύνεται απευθείας στη σελίδα *checkout*.

auth-guard.ts × login.component.ts

client > src > app > core > guards > auth-guard.ts > ...

```ts
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AccountService } from '../services/account.service';
import { map, of } from 'rxjs';

export const authGuard: CanActivateFn = (route, state) => {
  const accountService = inject(AccountService);
  const router = inject(Router);

  if (accountService.currentUser()) {
    return of(true);
  } else {
    return accountService.getAuthState().pipe(
      map(auth => {
        if (auth.isAuthenticated) {
          return true;
        } else {
          router.navigate(['/account/login'], {queryParams: {returnUrl: state.url}});
          return false;
        }
      })
    );
  }
};
```

Εικόνα 346

Για να ολοκληρωθεί η συγκεκριμένη διαδικασία, θα πρέπει να μεταβούμε στο αρχείο *app.routes.ts* και να υποδείξουμε το *path* το οποίο θα επιτηρεί η *Angular guard* κλάση. Στην εικόνα 347 βλέπουμε τον τρόπο με τον οποίο προσθέτουμε έναν *Angular guard* στο *checkout path*, κάνοντας χρήση της *property canAcivate*, με παράμετρο το όνομα της *Angular guard* κλάσης *authGuard*.

```
{path:'checkout', component: CheckoutComponent, canActivate: [authGuard]},
```

Εικόνα 347

11.2.2 Κενό Καλάθι Αγορών

Κινούμενοι στο ίδιο μήκος κύματος όπως στην ενότητα 11.2.1, θα δημιουργήσουμε και μία δεύτερη *Angular guard* κλάση, όπως βλέπουμε και στην εντολή της εικόνας 348. Η νέα αυτή κλάση ονομάζεται *cartempty* κι αποτρέπει τους χρήστες (εγγεγραμμένους και μη), από το να έχουν πρόσβαση στη σελίδα *cart*, στην περίπτωση που το καλάθι αγορών είναι κενό.

```
PS C:\Users\USER\Desktop\pcparts\client> ng g g core/guards/cartempty --skip-tests
✓ Which type of guard would you like to create? CanActivate
CREATE src/app/core/guards/cartempty-guard.ts (138 bytes)
PS C:\Users\USER\Desktop\pcparts\client>
```

Εικόνα 348

Στην εικόνα 349 παρουσιάζεται ο κώδικας της κλάσης *cartEmptyGuard*. Στη συγκεκριμένη περίπτωση δεν απαιτείται η χρήση κάποιου *observable object*, καθώς ο έλεγχος γίνεται απευθείας στο καλάθι αγορών, ανεξαρτήτως προϊόντων ή εγγραφής του τρέχοντος χρήστη. Στην περίπτωση που το καλάθι είναι κενό, επιστρέφεται μήνυμα στον χρήστη από την *Angular* κλάση *SnackbarService* και γίνεται ανακατεύθυνση στη σελίδα *cart*. Σε αντίθετη περίπτωση (*return true*), το *cart path* «απελευθερώνεται» από την εποπτεία της *guard* κλάσης και παρουσιάζονται τα προϊόντα του καλαθιού στον χρήστη.

cartempty-guard.ts × app.routes.ts ×

client > src > app > core > guards > cartempty-guard.ts > ...

```ts
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { CartService } from '../services/cart.service';
import { SnackbarService } from '../services/snackbar.service';

export const cartEmptyGuard: CanActivateFn = (route, state) => {
  const cartService = inject(CartService);
  const router = inject(Router);
  const snack = inject(SnackbarService);

  if (!cartService.cart() || cartService.cart()?.items.length === 0) {
    snack.error('Your cart is empty');
    router.navigateByUrl('/cart');
    return false;
  }
  return true;
};
```

Εικόνα 349

Η παρέμβαση στο cart.*component.html template*, έχει να κάνει με την προσθήκη της δομής *@if* (εικόνα 350). Μετακινήσαμε τον ήδη υπάρχον κώδικα εντός της δομής *@if*, πράγμα που σημαίνει πως για όσο το καλάθι αγορών παραμένει κενό, στον χρήστη επιστρέφεται το περιεχόμενο του *empty-state.component.html template* (εικόνα 351).

cart.component.html

client > src > app > features > cart > cart.component.html > ...

Go to component

```
<section>
  @if (cartService.cart()?.items?.length! > 0) {
  <div class="mx-auto max-w-7xl">
    <div class="flex w-full items-start gap-6 mt-32">
      <div class="w-3/4">
        @for (item of cartService.cart()?.items; track item.productId) {
          <app-cart-item [item]="item"></app-cart-item>
        }
      </div>
      <div class="w-1/4">
        <app-order-summary></app-order-summary>
      </div>
    </div>
  </div>
  } @else {
    <app-empty-state></app-empty-state>
  }
</section>
```

Εικόνα 350

Ουσιαστικά πρόκειται για ένα πλαίσιο λευκού φόντου στο μέσον της οθόνης του χρήστη, στο κέντρο του οποίο εμφανίζεται το *mat-icon shopping_cart* κι ένα *routerLink button* με τίτλο «*Go shopping*», το οποίο εφόσον το επιλέξει ο χρήστης ανακατευθύνεται στη σελίδα *shop* της εφαρμογής μας.

empty-state.component.html

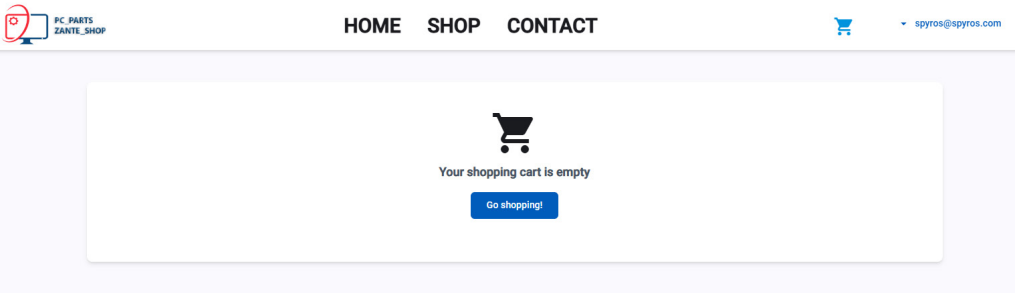
client > src > app > shared > components > empty-state > empty-state.component.html > ...

Go to component

```
<div class="max-w-7xl mx-auto mt-32 px-10 py-4 bg-white rounded-lg shadow-md w-full">
    <div class="flex flex-col items-center justify-center py-12 w-full">
        <mat-icon class="icon-display mb-8">shopping_cart</mat-icon>
        <p class="text-gray-600 text-lg font-semibold mb-4">
            Your shopping cart is empty
        </p>
        <button routerLink="/shop" mat-flat-button>Go shopping!</button>
    </div>
</div>
```

Εικόνα 351

Στην εικόνα 352 παρουσιάζεται το περιεχόμενο της σελίδας *cart* (*cart.component.html template*), στην περίπτωση που το καλάθι αγορών είναι άδειο.



Εικόνα 352

Κεφάλαιο 12. Τοποθέτηση και Ολοκλήρωση Παραγγελιών

Το τελευταίο κεφάλαιο της διπλωματικής μου εργασίας, αφορά την ανάπτυξη της λειτουργικότητας η οποία σχετίζεται με τον τρόπο τοποθέτησης και ολοκλήρωσης της ηλεκτρονικής αγοράς του χρήστη, στο περιβάλλον του ηλεκτρονικού μας καταστήματος *PcParts*. Για την υλοποίηση της συγκεκριμένης δομής, θα βασιστούμε σε μία ευρέως διαδεδομένη ηλεκτρονική πλατφόρμα συναλλαγών, η οποία ονομάζεται *Stripe*. Η *Angular* πλατφόρμα, στις τελευταίες της εκδόσεις (v.20 τη δεδομένη χρονική στιγμή) έχει εισάγει όλες τις απαραίτητες βιβλιοθήκες της *Stripe*, με στόχο οι προγραμματιστές να μπορούν να προσαρμόσουν εύκολα, αποτελεσματικά και με ασφάλεια τη λειτουργικότητα των *online* συναλλαγών στην εκάστοτε *e-commerce web* εφαρμογή τους.

12.1 Ενημέρωση του .Net Core Project

Ξεκινώντας, θα μεταβούμε στο *Core project* κι εντός του καταλόγου *Entities*, θα δημιουργήσουμε μία νέα κλάση με όνομα *DeliveryMethod*. Εδώ προσθέτουμε όλες τις απαραίτητες οντότητες (εικόνα 353) οι οποίες περιγράφουν την εκάστοτε μέθοδο αποστολής εντός της βάσης *pcparts.db*. Κάθε μέθοδος αποστολής έχει μία περιγραφή, ένα όνομα, ένα χρονικό διάστημα παράδοσης, μία τιμή κι έναν μοναδικό αριθμό *id*, τον οποίο μας δίνει η κλάση *BaseEntity*.

DeliveryMethod.cs

Core > Entities > DeliveryMethod.cs > ...

```
1  using System;
2
3  namespace Core.Entities;
4
   7 references
5  public class DeliveryMethod : BaseEntity
6  {
       0 references
7      public required string ShortName { get; set; }
       0 references
8      public required string DeliveryTime { get; set; }
       0 references
9      public required string Description { get; set; }
       2 references
10     public decimal Price { get; set; }
11 }
```

Εικόνα 353

Στην εικόνα 354, παρουσιάζεται ο τρόπος με τον οποίο θα ενημερώσουμε το υπάρχον σχήμα της βάσης δεδομένων *pcparts.db*, υποδεικνύοντας εντός της κλάσης *program.cs* τη διεύθυνση του αρχείου *delivery.json* (εικόνα 355), στο οποίο έχουμε προσθέσει τις τέσσερεις διαθέσιμες μεθόδους αποστολής προϊόντων από το ηλεκτρονικό μας κατάστημα. Όπως ακριβώς πραγματοποιήσαμε και στην περίπτωση του πίνακα *products*, εφαρμόζεται μία νέα *.Net migration* κατά την εκκίνηση της εφαρμογής μας και προστίθεται στη βάση δεδομένων ο νέος πίνακας με όνομα *deliveryMethods*, διανθισμένος με τα στοιχεία του *json* αρχείου με τίτλο *delivery*.

```
//seed data from deliver.json file of delivery methods if empty
if (!context.DeliveryMethods.Any())
{
    var dmData = await File.ReadAllTextAsync("../Infrastructure/Data/SeedData/delivery.json");
    var methods = JsonSerializer.Deserialize<List<DeliveryMethod>>(dmData);

    if (methods == null) return;//checks for errors in path or file delivery.json

    context.DeliveryMethods.AddRange(methods);//seeds the db table deliverymethods

    await context.SaveChangesAsync();
}
```

Εικόνα 354

delivery.json

Infrastructure > Data > SeedData > delivery.json > ...

```
[
  {
    "ShortName": "UPS1",
    "Description": "Fastest delivery time",
    "DeliveryTime": "1-2 Days",
    "Price": 10
  },
  {
    "ShortName": "UPS2",
    "Description": "Get it within 5 days",
    "DeliveryTime": "2-5 Days",
    "Price": 5
  },
  {
    "ShortName": "UPS3",
    "Description": "Slower but cheap",
    "DeliveryTime": "5-10 Days",
    "Price": 2
  },
  {
    "ShortName": "FREE",
    "Description": "Free! You get what you pay for",
    "DeliveryTime": "1-2 Weeks",
    "Price": 0
  }
]
```

Εικόνα 355

12.2 Ενημέρωση του Angular Project

Στο *Angular project* και συγκεκριμένα εντός του *cart.ts* αρχείου, εισάγουμε δύο νέες οντότητες, την *paymentMethod* και την *deliveryMethod* (εικόνα 356). Η πρώτη αφορά την *Stripe* δομή εντός του *Angular project*, ενώ η δεύτερη προκύπτει βάσει των ενεργειών μας στην προηγούμενη ενότητα.

cart.ts

client > src > app > shared > models > cart.ts > Cart

```
1  import {nanoid} from "nanoid";
2  //api cart properties replication
3  export type CartType = {
4    id: string;
5    items: CartItem[];
6    paymentMethodId?: string;
7    deliveryMethodId?: number;
8
```

Εικόνα 356

Με τον ίδιο τρόπο ορίζουμε τις δύο αυτές μεταβλητές κι εντός της *TypeScript* κλάσης *Cart* (εικόνα 357).

```
23  export class Cart implements CartType {
24    id = nanoid(); //random numbers generator for cart id
25    items: CartItem[] = [];
26    deliveryMethodId?: number;
27    paymentMethodId?: string;
28
```

Εικόνα 357

12.2.1 Προσθήκη ενός Angular Mat-Stepper Menu

Έχοντας δημιουργήσει τη σελίδα προβολής του ηλεκτρονικού καλαθιού αγορών, σειρά παίρνει και η ενημέρωση του *html template checkout.component.html*. Έχοντας υπόψιν τα βήματα διεκπεραίωσης της παραγγελίας του χρήστη, θα προσθέσουμε ένα *Angular Mat-stepper menu*, το οποίο θα διευκολύνει και θα καθοδηγεί τον χρήστη έως την πληρωμή της παραγγελίας του. Ουσιαστικά πρόκειται για ένα μενού τεσσάρων βημάτων (*Address form*, *Delivery form*, *Payment form*, *Review form*), δίπλα στο οποίο εμφανίζεται η σύνοψη της παραγγελίας. Ο κώδικας με τον οποίο δημιουργούμε τα παραπάνω βρίσκεται στην εικόνα 358.

checkout.component.html

client > src > app > features > checkout > checkout.component.html > div.flex.mt-32.gap-6

Go to component

```
<div class="flex mt-32 gap-6">
    <div class="w-3/4">
      <mat-stepper #stepper class="bg-white border border-blue-600 shadow-sm">
          <mat-step>
              Address form
          </mat-step>
          <mat-step>
              Delivery form
          </mat-step>
          <mat-step>
              Payment form
          </mat-step>
          <mat-step>
              Review form
          </mat-step>
      </mat-stepper>
    </div>
    <div class="w-1/4">
        <app-order-summary></app-order-summary>
    </div>
</div>
```

Εικόνα 358

Από τη στιγμή που ο χρήστης επιλέγει το κουμπί *Checkout* από τη σελίδα του καλαθιού αγορών, μεταφέρεται στην αντίστοιχη σελίδα. Για τον λόγο αυτό θέλουμε να απενεργοποιούμε το εν λόγω κουμπί εντός του *checkout.component.html template*. Για να συμβεί κάτι τέτοιο, θα μεταβούμε στην κλάση *OrderSummaryComponent* και θα εισάγουμε (*inject*) την κλάση *Location*. Μέσω της *location* μεταβλητής μπορούμε ανά πάσα στιγμή να γνωρίζουμε το *URL* μονοπάτι το οποίο έχει αιτηθεί και βρίσκεται ο ενεργός χρήστης (εικόνα 359).

```
export class OrderSummaryComponent {
  cartService = inject(CartService);
  location = inject(Location);//checks current url
}
```

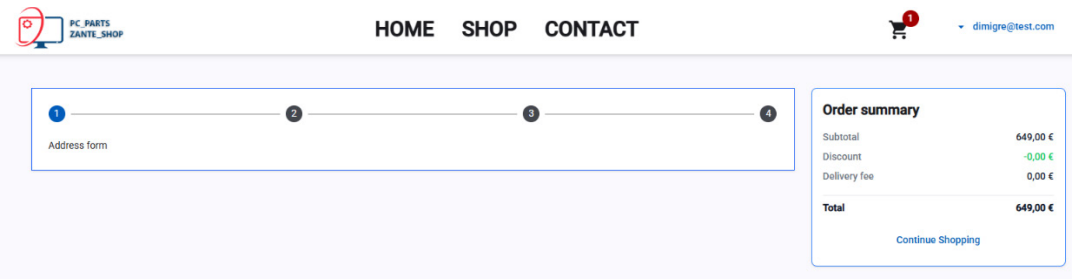
Εικόνα 359

Ακολούθως, μεταφερόμαστε στο αντίστοιχο *html template* και προσθέτουμε τον έλεγχο για το αν ο χρήστης παρακολουθεί τη σελίδα του *checkout* (εικόνα 360). Σε αυτήν την περίπτωση το κουμπί *Checkout* δεν είναι διαθέσιμο εντός του *order-summary component* (εικόνα 361).

```
    <div class="flex flex-col gap-2">
      @if(location.path()!=='/checkout'){
        <button routerLink="/checkout" mat-flat-button>Checkout</button>
      }
```

Εικόνα 360

Σε αυτό το σημείο, η εφαρμογή μας διαθέτει ένα μενού τεσσάρων βημάτων στη σελίδα του *checkout*. Στις επόμενες ενότητες θα ενημερώσουμε τη λειτουργικότητα του εκάστοτε βήματος.



Εικόνα 361

12.2.2 Η Stripe Angular Service

Όπως αναφέραμε και στην εισαγωγή του τρέχοντος κεφαλαίου, για τη λειτουργικότητα της περάτωσης της παραγγελίας ενός χρήστη, θα βασιστούμε στις *Stripe - Angular* βιβλιοθήκες οι οποίες μας παρέχονται αυτούσιες μέσω του *Angular Framework*. Ένα εργαλείο το οποίο θα προσθέσουμε στη δομή του προγράμματός μας, είναι η φόρμα συμπλήρωσης των στοιχείων αποστολής της παραγγελίας του χρήστη. Η *Stripe* μας δίνει μία τέτοια προσχεδιασμένη φόρμα, βασιζόμενη στη χώρα του χρήστη (από τα δεδομένα τοποθεσίας του λειτουργικού μας συστήματος), παρόλα αυτά εμείς επιθυμούμε την αυτόματη εισαγωγή των πληροφοριών του χρήστη από τη βάση δεδομένων μας. Για να συμβεί κάτι τέτοιο θα μεταβούμε στην *Stripe Angular service* και θα αντικαταστήσουμε τις *default* τιμές της συγκεκριμένης φόρμας στην *Stripe* μέθοδο *createAddressElement*, με αυτές που λαμβάνουμε από τον πίνακα *Address* της βάσης δεδομένων *pcparts.db* (εικόνα 362).

```
//address element for checkout
  async createAddressElement() {
    if (!this.addressElement) {//stripe address element
      const elements = await this.initializeElements();
      if (elements) {
        const user = this.accountService.currentUser();//signal variable
        //load stripe default address values (country based)
        let defaultValues: StripeAddressElementOptions['defaultValues'] = {};
        //load user NAME and Surname
        if (user) {
          defaultValues.name = user.firstName + ' ' + user.lastName;
        }
        //if user has provided address on register process ignores default values
        if (user?.address) {
          defaultValues.address = {
            line1: user.address.line1,
            line2: user.address.line2,
            city: user.address.city,
            state: user.address.state,
            country: user.address.country,
            postal_code: user.address.postalCode
          };
        }
        //else load default fro shipping fields
        const options: StripeAddressElementOptions = {
          mode: 'shipping',
          defaultValues //passing user values to stripe
        };//creates the form to send to components
        this.addressElement = elements.create('address', options);
      } else {
        throw new Error('Elements instance has not been loaded');
      }
    }
    return this.addressElement;
  }
```

Εικόνα 362

Στη συνέχεια ανοίγουμε το αρχείο *checkout.component.ts* και στην αντίστοιχη *TypeScript* κλάση *checkoutComponent* κάνουμε *implement* την οντότητα *Angular lifecycle hook OnInit*. Μέσω της μεθόδου

ngOnInit επιτυγχάνουμε την αρχικοποίηση του *address-element*, εντός του *html checkout template*, χρησιμοποιώντας εντός της *Stripe* φόρμας τα δεδομένα χρήστη, παρακάμπτοντας αυτά που μας προσφέρει η *Stripe* από τις *online* βιβλιοθήκες της (εικόνα 363).

```
22  export class CheckoutComponent implements OnInit {
23    private stripeService = inject(StripeService);
24    private snackBar = inject(SnackbarService);
25    addressElement?: StripeAddressElement;
26
27    async ngOnInit() {
28      try {
29        this.addressElement = await this.stripeService.createAddressElement();
30        this.addressElement.mount("#address-element");
31      } catch (error:any) {
32        this.snackBar.error(error.message)
33      }
34
35    }
```

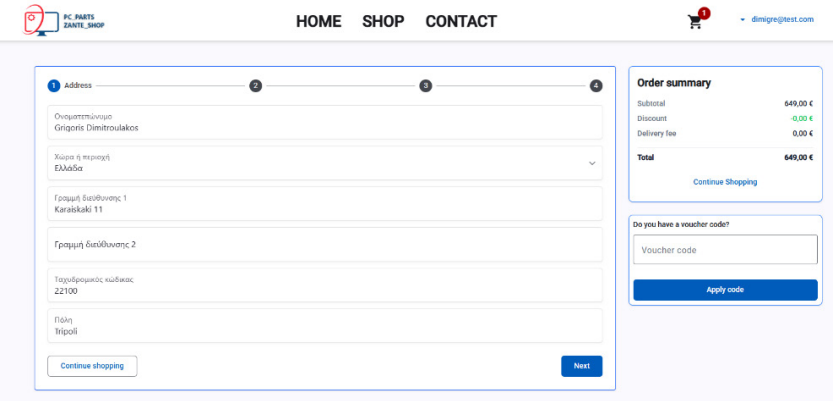
Εικόνα 363

Στο αντίστοιχο *html template*, προσθέτουμε εντός του * element*, το *address-element* που ορίσαμε στο προηγούμενο βήμα, το *routerlink mat-stroked-button Continue shopping*, το οποίο ανακατευθύνει τον χρήστη στη σελίδα των προϊόντων, καθώς και το *mat-stepper Next button*, το οποίο τον οδηγεί στο επόμενο βήμα του *Angular mat-stepper menu* (εικόνα 364).

```
<mat-stepper #stepper class="bg-white border border-blue-600 rounded shadow-sm">
    <mat-step label="Address">
        <div id="address-element"></div>
        <div class="flex justify-between mt-6">
            <button routerLink="/shop" mat-stroked-button>Continue shopping</button>
            <button matStepperNext mat-flat-button>Next</button>
        </div>
    </mat-step>
```

Εικόνα 364

Έχοντας πραγματοποιήσει σύνδεση στην πλατφόρμα μας με τον υφιστάμενο λογαριασμό χρήστη που δημιουργήσαμε στο προηγούμενο κεφάλαιο, δοκιμάζουμε την προσθήκη ενός προϊόντος στο καλάθι. Όταν επιλέγουμε το κουμπί *Checkout*, μέσω του *order summary component*, οδηγούμαστε στη σελίδα του *checkout*, στην οποία σχεδιάσαμε το *Angular mat-stepper menu*. Πλέον, στο πρώτο επίπεδο του συγκεκριμένου μενού, προβάλλεται η πληροφορία που αφορά τα στοιχεία διεύθυνσης του ενεργού χρήστη (εικόνα 365).



Εικόνα 365

Στο σημείο αυτό, τίθεται το εξής ερώτημα: Τί γίνεται στην περίπτωση που ο χρήστης δεν έχει αποθηκευμένα δεδομένα διεύθυνσης στη βάση δεδομένων μας κι επιθυμεί να τα εισάγει και να τα αποθηκεύσει για μελλοντικές αγορές; Την απάντηση σε αυτό το ερώτημα την παρέχει για ακόμη μια φορά η *Stripe Angular Service*. Εντός του αρχείου *Stripe.Service.ts* δημιουργούμε τρεις νέες μεθόδους, την *onSaveAddressCheckboxChange* και τις ασύγχρονες *onStepChange* και *getAdressFromStripeAddress*. Η πρώτη μέθοδος αποτελεί έναν *event handler* και καλύπτει την περίπτωση όπου ο χρήστης εισάγει τα στοιχεία διεύθυνσής του κι επιλέγει το *checkbox* αποθήκευσης, πριν μεταβεί στο επόμενο βήμα του *stepper menu*. Αυτή η πληροφορία επιστρέφεται στη μέθοδο *onStepChange* και καθώς ο χρήστης μεταβαίνει στο επόμενο επίπεδο του *stepper menu* (*if event.selectedIndex == 1*), καλείται η μέθοδος *getAddressFromStripeAddress*. Η τελευταία μέθοδος, αντλεί το περιεχόμενο των πεδίων της *Stripe* φόρμας και τα αποδίδει στη μεταβλητή *address*. Τέλος, καλείται η μέθοδος *updateAddress* μέσω της *AccountService* και τα δεδομένα διεύθυνσης χρήστη αποθηκεύονται στον αντίστοιχο πίνακα της βάσης δεδομένων. Στην εικόνα 366 παρουσιάζεται ο κώδικας των προαναφερθέντων.

```
//checks if save saveAddress checkbox is checked
onSaveAdressCheckboxChange(event: MatCheckboxChange){
  this.saveAddress = event.checked; //returns true if checkbox is checked
}

//stepper starts count from 0 stage
async onStepChange(event: StepperSelectionEvent) {
  if (event.selectedIndex === 1) { //if stepper moved forword (address -> delivery)
    if (this.saveAddress) {
      const address = await this.getAddressFromStripeAddress();//calls function
      //if we have the address from function above calls updateAddres method to
      //parse it to the API using Angular method firtsValueFrom
      address && firstValueFrom(this.accountService.updateAddress(address));
    }
  }
}

//extracts address from stripe form and returns it
private async getAddressFromStripeAddress() {
  const result = await this.addressElement?.getValue();
  const address = result?.value.address;

  if (address) {
    return {
      line1: address.line1,
      line2: address?.line2 || undefined,
      city: address.city,
      state: address.state,
      country: address.country,
      postalCode: address.postal_code
    }
  } else return null;
}
```

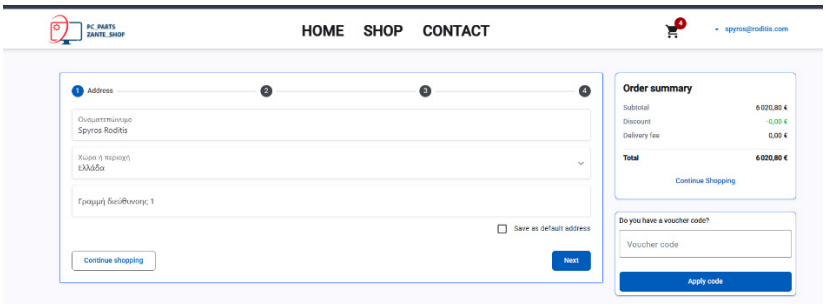
Εικόνα 366

Στο αρχείο *checkout.component.html*, κάνοντας χρήση των *Angular event attributes* (*selectionChange*), *[checked]* και *(change),* καλούμε τις μεθόδους του προηγούμενου βήματος. Πρακτικά, δημιουργούμε το *checkbox* «*Save as default address*» κάτω και δεξιά από την *Stripe* φόρμα διεύθυνσης και μόλις αυτό επιλεγεί από τον χρήστη, αποστέλλονται τα κατάλληλα *events* προς την *Stripe service*, τη στιγμή μετάβασής του στο επόμενο βήμα (εικόνα 367).

```
<mat-stepper #stepper
(selectionChange)="onStepChange($event)"
class="bg-white border border-blue-600 rounded shadow-sm">
    <mat-step label="Address">
        <div id="address-element"></div>
        <div class="flex justify-end mt-1">
            <mat-checkbox
              [checked]="saveAddress"
              (change)="onSaveAdressCheckboxChange($event)"
            >
            Save as default address
            </mat-checkbox>
        </div>
```

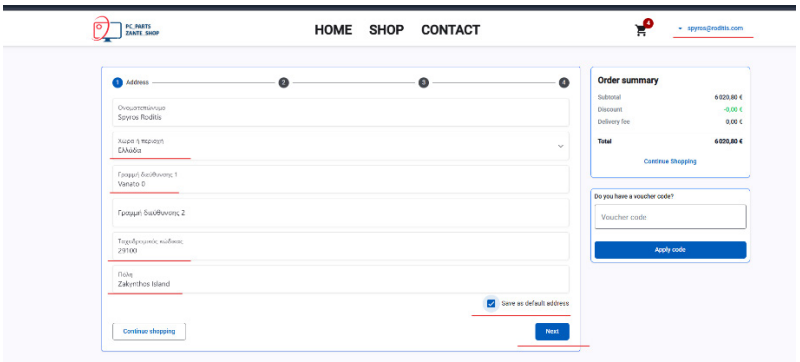
Εικόνα 367

Στην εικόνα 368 παρουσιάζεται η εικόνα της σελίδας *checkout*, στην οποία εμφανίζεται ένας νέος ενεργός χρήστης. Όπως παρατηρούμε, δεν υπάρχουν διαθέσιμα στοιχεία διεύθυνσης στα αντίστοιχα πεδία της *Stripe* φόρμας.



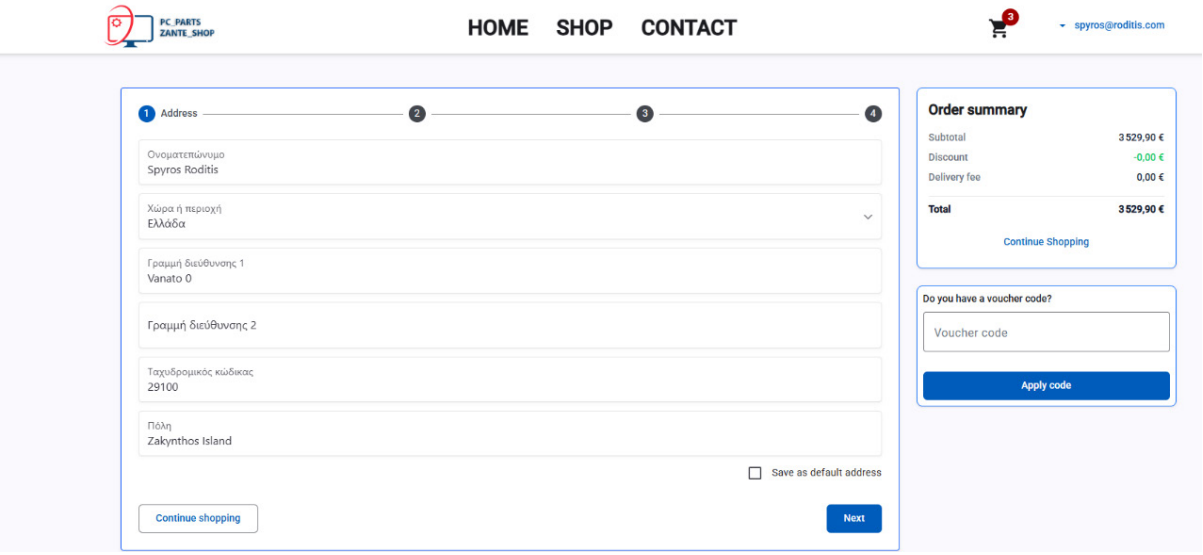
Εικόνα 368

Στην εικόνα 369 ο χρήστης επιλέγει την *Stripe* φόρμα, εισάγει όλες τις απαραίτητες πληροφορίες στα αναδυόμενα πεδία κι επιλέγει το *checkbox* «*Save as default address*».



Εικόνα 369

Στην εικόνα 370, εφόσον έχουμε πραγματοποιήσει *logout* κι ακολούθως *login* στην πλατφόρμα μας, ακολουθούμε εκ νέου τη διαδικασία έως τη σελίδα του *checkout*. Αυτήν τη φορά η φόρμα με τα στοιχεία διεύθυνσης επιστρέφεται συμπληρωμένη με τα στοιχεία του πίνακα *Address* από τον *API server*.



Εικόνα 370

12.3 Δημιουργία του Checkout-Delivery Component

Στο δεύτερο επίπεδο του *Angular mat-stepper menu*, ο χρήστης καλείται να επιλέξει μία από τις τέσσερεις διαθέσιμες μεθόδους αποστολής της παραγγελίας του. Σε αυτό το σημείο, θα χρειαστεί να δημιουργήσουμε ένα νέο *Angular component*, με τίτλο *checkout-delivey*, σύμφωνα με την εντολή της εικόνας 371.

```
PS C:\Users\USER\Desktop\pcparts\client> ng g c features/checkout-delivery --skip-tests
CREATE src/app/features/checkout-delivery/checkout-delivery.component.ts (270 bytes)
CREATE src/app/features/checkout-delivery/checkout-delivery.component.scss (0 bytes)
CREATE src/app/features/checkout-delivery/checkout-delivery.component.html (33 bytes)
PS C:\Users\USER\Desktop\pcparts\client>
```

Εικόνα 371

Αμέσως μετά, εντός του καταλόγου *models* θα δημιουργήσουμε ένα νέο *TypeScript* μοντέλο, αντίστοιχο με αυτό του *Core project*, με όνομα *DeliveryMethod* (εικόνα 372).

```
ent > src > app > shared > models > deliveryMethod.ts >
1  export type DeliveryMethod = {
2    shortName: string;
3    deliveryTime: string;
4    description: string;
5    price: number;
6    id: number;
7  }
```

Εικόνα 372

Εν συνεχεία, μεταφερόμαστε στην *checkout service* και δημιουργούμε τη μέθοδο *getDeliveryMethods*. Με τη συγκεκριμένη μέθοδο αιτούμαστε από τον *API server* όλες τις διαθέσιμες μεθόδους αποστολής

από τον πίνακα *DeliveryMethods*, τις αποθηκεύουμε στη μεταβλητή *deliveryMethods* και τις παρουσιάζουμε στον *client* κατά φθίνουσα τιμή κόστους (εικόνα 373).

```
private http = inject(HttpClient);
deliveryMethods: DeliveryMethod[] = [];//set delivery methods array to be empty

getDeliveryMethods() {
  if (this.deliveryMethods.length > 0) return of(this.deliveryMethods);//if already have th
  //if not ask them from the API observable pipe
  return this.http.get<DeliveryMethod[]>(this.baseUrl + 'payments/delivery-methods').pipe(
    map(delivmethods => {//and sort them from most expensive to least expensive
      this.deliveryMethods = delivmethods.sort((a, b) => b.price - a.price);
      return delivmethods;
    })
  );
}
```

Εικόνα 373

Ακολούθως, θα μεταβούμε στην κλάση *CartServce* κι εκεί θα ορίσουμε την *Angular signal* μεταβλητή *selectedDeliver*, η οποία μάς επιστρέφει μία μέθοδο αποστολής ή την τιμή *null* (εικόνα 374).

```
//signal to watch the selected delivery method from user at checkout
selectedDelivery = signal<DeliveryMethod | null>(null);
```

Εικόνα 374

Εντός της ίδιας κλάσης, στη σταθερά *delivery* αναθέτουμε το περιεχόμενο της *signal* μεταβλητής *selectedDelivery*. Με αυτόν τον τρόπο, εφόσον το περιεχόμενο της *signal* μεταβλητής δεν είναι ίσο με *null*, μεταβάλλεται το κόστος των μεταφορικών σύμφωνα με την αξία της επιλεγμένης μεθόδου αποστολής (εικόνα 375).

```
totals = computed(()=> {//calculating the order summary values via signal variables (cart)
  const cart = this.cart();//using cart signal variable
  const delivery = this.selectedDelivery();
  if (!cart) return null;//if the cart is empty or not found
  const subtotal = cart.items.reduce((sum, item) => sum + item.price * item.quantity, 0);

  const shipping = delivery ? delivery.price : 0; //if delivery is not null then charge ship
  const discount = 0;
  return {
    subtotal,
    shipping,
    discount,
    total: subtotal + shipping - discount
  };
})
```

Εικόνα 375

Επόμενος σταθμός μας είναι η νέα κλάση *CheckoutDeliveryComponent*, στην οποία κάνουμε *implement* την *lifecycle hook* οντότητα *OnInit*. Όπως και στην περίπτωση της κλάσης *Checkout*, έτσι κι εδώ θέλουμε να αρχικοποιήσουμε το περιεχόμενο των *signal* μεταβλητών με τις *default* τιμές που έχουμε ορίσει.

Για τον λόγο αυτό, εντός της μεθόδου *ngOnInit* χρησιμοποιούμε ένα *observable object*, ώστε μέσω της *signal* μεταβλητής *cart* να αποδίδουμε κάθε φορά την επιλεγμένη μέθοδο αποστολής στη μεταβλητή *method* και στον χρήστη μέσω του *checkout-delivery.html template*. Με άλλα λόγια, κάθε φορά που ο

χρήστης μεταβαίνει στο βήμα «*Shipping*» του *mat-stepper menu*, αντικρίζει τέσσερεις μεθόδους αποστολής, εκ των οποίων καμία δεν είναι επιλεγμένη (αρχικοποίηση των *components* και *signals* μέσω της *ngOnInit*). Κάθε φορά που επιλέγει μία μέθοδο αποστολής συμβαίνουν τρία πράγματα: Πρώτον, ενημερώνεται η *CartService* κλάση με την επιλεγμένη μέθοδο αποστολής (υπολογισμός κόστους μεταφορικών για το σύνολο της παραγγελίας). Δεύτερον, μέσω της μεθόδου *updateDeliveryMethod* ενημερώνεται το επιλεγμένο πεδίο της συγκεκριμένης μεθόδου αποστολής εντός του *checkout-delivery.component.html template*. Τρίτον, ενημερώνεται το περιεχόμενο του καλαθιού αναφορικά με το κόστος του στον *Redis server*. Το περιεχόμενο της κλάσης *CheckoutDeliveryComponent* παρουσιάζεται στην εικόνα 376.

```
18  export class CheckoutDeliveryComponent implements OnInit {
19    checkoutService = inject(CheckoutService);
20    cartService = inject(CartService);
21
22    ngOnInit() {
23      this.checkoutService.getDeliveryMethods().subscribe({
24        next: methods => {//subscribe to signal cart() in order to find
25          if (this.cartService.cart()?.deliveryMethodId) {
26            const method = methods.find(d => d.id === this.cartService.cart()?.deliveryMethodId);
27            if (method) {//selected Delivery signal inside cart service with delivery method id
28              this.cartService.selectedDelivery.set(method);
29            }
30          }
31        }
32      });
33    }
34
35    updateDeliveryMethod(method: DeliveryMethod){
36      //updates the selectedDelivery signal inside cart service with selected
37      //delivery method ID
38      this.cartService.selectedDelivery.set(method);
39      const cart = this.cartService.cart();
40      if(cart){//also update cart in redis
41        cart.deliveryMethodId = method.id;
42        this.cartService.setCart(cart);
43      }
44    }
```

Εικόνα 376

Στο αρχείο *checkout-delivery.component.html* δημιουργούμε ένα *mat-radio buttons group* που περιέχει τις διαθέσιμες μεθόδους αποστολής των προϊόντων. Μέσω της *signal* μεταβλητής *selectedDelivery().?id* και του *attribute [value]*, δίνουμε ως όρισμα στη μέθοδο *updateDeliveryMethod* το *id* της επιλεγμένης μεθόδου αποστολής. Καθώς έχουμε το *id* της επιλεγμένης μεθόδου αποστολής από τον χρήστη, μέσω ενός *@for loop* και της μεταβλητής *method* διατρέχουμε συνεχώς τα *events* που προκύπτουν ανάλογα με την επιλογή του χρήστη.

Έτσι, κάθε φορά που ο χρήστης δίνει μία διαφορετική επιλογή, αναζητάται το *id* της εκάστοτε μεθόδου αποστολής, αποθηκεύεται στη μεταβλητή *method*, συγκρίνεται με το περιεχόμενο της *signal* μεταβλητής *selectedDelivery* και τέλος η επιλογή του παρουσιάζεται ως *checked button* στο *mat-radio button group*. Εντός κάθε πεδίου επιλογής εμφανίζονται το όνομα, η περιγραφή, το κόστος και ο χρόνος παράδοσης της κάθε υπηρεσίας αποστολής. Στην εικόνα 377 παρουσιάζεται ο κώδικας του συγκεκριμένου *html template*.

checkout-delivery.component.html

client > src > app > features > checkout-delivery > checkout-delivery.component.html > ...

Go to component

```
<div class="w-full">
    <mat-radio-group
        [value]="cartService.selectedDelivery()?.id"
        (change)="updateDeliveryMethod($event.value)"
        class="grid grid-cols-2 gap-4">
        @for (method of checkoutService.deliveryMethods; track method.id) {
        <label
            class="p-3 border border-gray-400 cursor-pointer w-full h-full hover:bg-blue-200">
            <mat-radio-button
                [value]="method"
                [checked]="cartService.selectedDelivery() === method"
                class="w-full h-full">
                <div class="flex flex-col w-full h-full">
                    <strong>{{ method.shortName }} - {{ method.price | currency:'EUR':'symbol':'1.2-2':'fr'}}</strong>
                    <span class="text-sm">{{ method.description }}</span>
                    <span class="text-sm">{{ method.deliveryTime }}</span>
                </div>
            </mat-radio-button>
        </label>
        }
    </mat-radio-group>
</div>
```

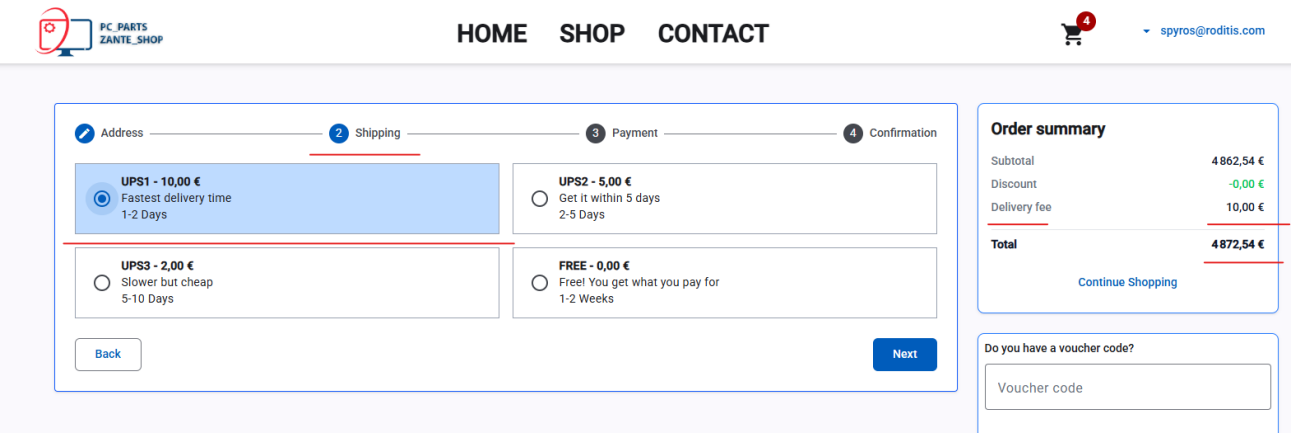
Εικόνα 377

Πριν εκτελέσουμε την εφαρμογή μας στο περιβάλλον του φυλλομετρητή, απομένει η εισαγωγή του *checkout-deliver component* στο *html template checkout*. Στην εικόνα 378 παρουσιάζεται η προσθήκη του εν λόγω *component*, χρησιμοποιώντας την ονομασία του εντός των *Angular attributes*.

```
<mat-step label="Shipping">
    <app-checkout-delivery></app-checkout-delivery>
    <div class="flex justify-between mt-6">
        <button matStepperPrevious mat-stroked-button>Back</button>
        <button matStepperNext mat-flat-button>Next</button>
    </div>
</mat-step>
```

Εικόνα 378

Τέλος, εκτελούμε την εφαρμογή μας και παρατηρούμε την εισαγωγή του *checkout-delivery component* στη σελίδα του *checkout* (εικόνα 379). Ο χρήστης έχει τη δυνατότητα τεσσάρων μεθόδων αποστολής, επιλέγοντας αυτήν που επιθυμεί μέσω ενός *radio-buttons menu*. Παράλληλα με την επιλογή του διαμορφώνεται αναλόγως και το τελικό κόστος της παραγγελίας του.



Εικόνα 379

12.4 Stripe PaymentElement Method

Στο τρίτο επίπεδο του *Angular mat-stepper menu*, θα προσθέσουμε τη δυνατότητα εισαγωγής των στοιχείων μίας πιστωτικής/χρεωστικής κάρτας τύπου *Visa* ή *Mastercard* από τον χρήστη. Κάθε ηλεκτρονικό κατάστημα παρέχει τουλάχιστον μία μέθοδο πληρωμής προς τον χρήστη, έτσι και στην περίπτωση *της e-commerce* εφαρμογής μας θα παρέχουμε μία τέτοια δυνατότητα, εκμεταλλευόμενοι την *Stripe paymentservice*. Παρά το γεγονός πως αναπτύσσουμε μία εφαρμογή σε επίπεδο *localhost server* και καμία συναλλαγή δεν ανταποκρίνεται στην πραγματικότητα, η πλατφόρμα *Stripe* μας δίνει τη δυνατότητα εισαγωγής μεθόδων κι εργαλείων σε προγραμματιστικό περιβάλλον ανάπτυξης, προκειμένου να μάθουμε και να βιώσουμε το οικοσύστημα της συγκεκριμένης *online payment processing* πλατφόρμας. Ξεκινώντας, θα ανοίξουμε το *checkout.component*, όπου εντός της *TypeScript* κλάσης *CheckoutComponent* θα ορίσουμε μία νέα μεταβλητή με όνομα *paymentElement*, τύπου *StripePaymentELemet*. Ακολούθως, εντός της *Angular* μεθόδου *ngOnInit* αναθέτουμε ασύγχρονα στη συγκεκριμένη μεταβλητή, το περιεχόμενο που μας επιστρέφει η *Stipe* μέθοδος *createPaymentElement*. Πλέον, όπως και στην περίπτωση της *addressElement*, έχουμε τη δυνατότητα χρήσης των *Stripe* πεδίων για συναλλαγές μέσω κάρτας, εντός του αντίστοιχου *html template*. Αυτό επιτυγχάνεται για ακόμη μία φορά, με τη χρήση της *Angular* μεθόδου *mount*, στην οποία δίνουμε ως όρισμα το αλφαριθμητικό *'#payment-element'*, το οποίο μπορεί να χρησιμεύσει ως *id* τιμή ενός *html class element* (εικόνα 380).

```
36   paymentElement?: StripePaymentElement; ////////////////////////////////
37
38   async ngOnInit() {
39     try {
40       this.addressElement = await this.stripeService.createAddressElement();
41       this.addressElement.mount("#address-element");
42
43       this.paymentElement = await this.stripeService.createPaymentElement();
44       this.paymentElement.mount("#payment-element");
45
46     } catch (error:any) {
47       this.snackBar.error(error.message)
48     }
49
50   }
```

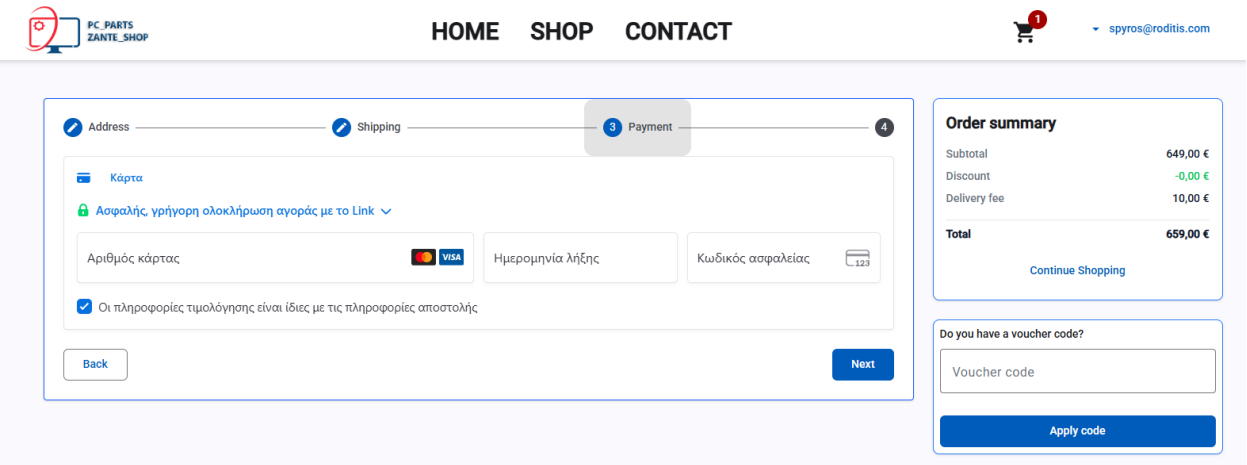
Εικόνα 380

Εφόσον δημιουργήσαμε το *html class attribute* με *id* που ισούται με το χαρακτηριστικό *#payment-element*, κάνουμε χρήση του εντός του * attribute* στο *html template checkout*. Το συγκεκριμένο κομμάτι κώδικα εισάγεται εντός του τρίτου *mat-step button*, όπως βλέπουμε στην εικόνα 381.

```
<mat-step label="Payment">
    <div id="payment-element"></div>
    <div class="flex justify-between mt-6">
        <button matStepperPrevious mat-stroked-button>Back</button>
        <button matStepperNext mat-flat-button>Next</button>
    </div>
</mat-step>
```

Εικόνα 381

Δοκιμάζοντας για ακόμη μία φορά την εφαρμογή μας στο περιβάλλον του *browser*, διαπιστώνουμε πως στο τρίτο βήμα του *mat-stepper menu* με τίτλο «*Payment*», εμφανίζονται τα πεδία *Stripe* για εισαγωγή των στοιχείων μίας χρεωστικής/πιστωτικής κάρτας (εικόνα 382). Η εμφάνιση των πεδίων, ο τύπος της κάρτας και οι έλεγχοι εγκυρότητας των στοιχείων της, εφαρμόζονται αποκλειστικά από την *Angular Stripe service* σε συνδυασμό με την *online* πλατφόρμα του συγκεκριμένου εργαλείου.



Εικόνα 382

12.5 Δημιουργία του Checkout-Review Component

Φτάνοντας αισίως στο τέταρτο επίπεδο του *Angular mat-stepper menu*, επιθυμούμε να παρουσιάσουμε στον χρήστη μία σύνοψη των πεπραγμένων του, σχετικά με τα στοιχεία χρέωσης – αποστολής, τον τρόπο πληρωμής - αποστολής, καθώς και για το σύνολο της παραγγελίας του ανά προϊόν, πριν επικυρώσει την οριστική πληρωμή της παραγγελίας του. Για τους παραπάνω λόγους θα δημιουργήσουμε το νέο *Angular component* «*checkout-review*», σύμφωνα με την εντολή της εικόνας 383.

```
PS C:\Users\USER\Desktop\pcparts\client> ng g c features/checkout/checkout-review --skip-tests
CREATE src/app/features/checkout/checkout-review/checkout-review.component.ts (262 bytes)
CREATE src/app/features/checkout/checkout-review/checkout-review.component.scss (0 bytes)
CREATE src/app/features/checkout/checkout-review/checkout-review.component.html (31 bytes)
PS C:\Users\USER\Desktop\pcparts\client>
```

Εικόνα 383

Εντός της νέα κλάσης *CheckoutReviewComponent*, κάνουμε *inject* τις *CartService* και *AccountService*. Στη συνέχεια ορίζουμε τον *userInfo* array, στον οποίο μέσω της μεθόδου *getCurrentUserAddressForShipping*, αναθέτουμε όλη την πληροφορία διεύθυνσης χρήστη από τον *API server* και τη βάση δεδομένων μας (εικόνα 384).

checkout-review.component.ts

client > src > app > features > checkout > checkout-review > checkout-review.component.ts > ...

```ts
import { Component, inject } from '@angular/core';
import { CartService } from '../../../core/services/cart.service';
import { CurrencyPipe } from '@angular/common';
import { AccountService } from '../../../core/services/account.service';

@Component({
  selector: 'app-checkout-review',
  imports: [
    CurrencyPipe
  ],
  templateUrl: './checkout-review.component.html',
  styleUrl: './checkout-review.component.scss',
})
export class CheckoutReviewComponent {
  cartService = inject(CartService);
  currentUser = inject(AccountService);

   userInfo: any[] = []; //any cause Null or String
   address = null;

 getCurrentUserAddressForShipping(){

    this.userInfo[0] = this.currentUser.currentUser()?.firstName;
    this.userInfo[1] = this.currentUser.currentUser()?.lastName;
    this.userInfo[2] = this.currentUser.currentUser()?.email;
    this.userInfo[3] = this.currentUser.currentUser()?.address.line1;
    this.userInfo[4] = this.currentUser.currentUser()?.address.line2;
    this.userInfo[5] = this.currentUser.currentUser()?.address.city;
    this.userInfo[6] = this.currentUser.currentUser()?.address.state;
    this.userInfo[7] = this.currentUser.currentUser()?.address.postalCode;
    this.userInfo[8] = this.currentUser.currentUser()?.address.country;

    return this.userInfo.join('  -  ');
  }

}
```

Εικόνα 384

Εν συνεχεία, ανοίγουμε το αρχείο κώδικα του αντίστοιχου *html template* όπου εκεί μέσω των κατάλληλων *html* και *Angular attributes*, δημιουργούμε ένα * * μενού δύο επιπέδων. Στο πρώτο επίπεδο κάνουμε χρήση της μεθόδου *getCurrentUserAddressForShipping* και παρουσιάζουμε όλες τις λεπτομέρειες χρέωσης και διεύθυνσης του χρήστη. Στο δεύτερο επίπεδο μέσω ενός *@for loop* και της *signal* μεταβλητής *cart*, διασχίζουμε το τρέχων καλάθι του χρήστη και παρουσιάζουμε τη φωτογραφία, την ποσότητα και την τιμή του εκάστοτε προϊόντος εντός αυτού (εικόνα 385).

```
Go to component
<div class="mt-4 w-full">
  <h4 class="text-lg font-semibold underline">Billing and delivery information</h4>
  <dl>
    <dt class="font-medium">Shipping address and user information:</dt>
    <dd class="mt-1 text-gray-500 flex">
      <span>{{getCurrentUserAddressForShipping()}}</span>
    </dd>
    <dt class="font-medium">Payment details:</dt>
    <dd class="mt-1 text-gray-500 flex">
      <span>Payment: Credit Card</span>
    </dd>
  </dl>
</div>
<div class="mt-6 mx-auto">
  <div class="border-b border-gray-200">
    <table class="w-full text-center">
      <tbody class="divide-y divide-gray-200">
        @for (item of cartService.cart()?.items; track item.productId) {
        <tr>
          <td class="py-4">
            <div class="flex items-center gap-4">
              <img src="{{item.pictureUrl}}" alt="product image"
                   class="w-10 h-10 font-semibold">
              <span>{{ item.productName }}</span>
            </div>
          </td>
          <td class="p-5 font-semibold">x{{ item.quantity }}</td>
          <td class="p-5 text-right font-medium">{{ item.price |currency:'EUR':'symbol':'1.2-2':'fr' }}</td>
        </tr>
        }
      </tbody>
    </table>
  </div>
</div>
```

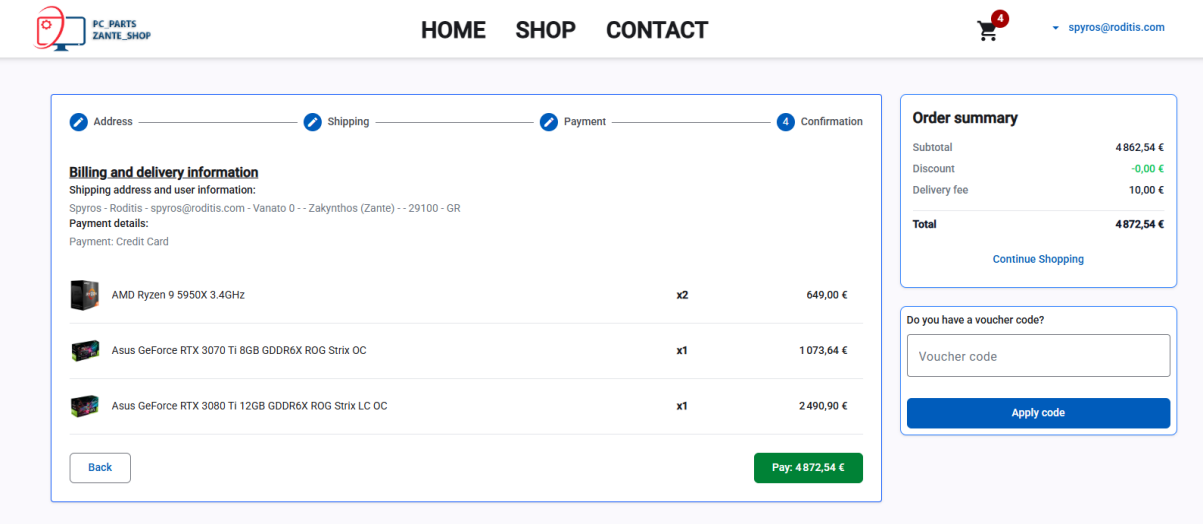
Εικόνα 385

Στο τέταρτο και τελευταίο επίπεδο του *mat-stepper menu*, ορίζουμε το *label confirmation* κι ακριβώς από κάτω εισάγουμε το *app-checkout-review attribute*, ώστε να εισαχθεί το νέο *component* στη σελίδα *checkout*. Επίσης, τη θέση του *next mat-flat-button* παίρνει ένα νέο κουμπί πράσινης απόχρωσης, στο οποίο μέσω της *cartService signal* μεταβλητής *totals*, παρουσιάζουμε το συνολικό κόστος της παραγγελίας. Δίπλα στο αναγραφόμενο ποσό, για ακόμη μια φορά κάνουμε χρήση της *Angular Currency pipe*, για να εμφανίζεται το σύμβολο νομισματικής μονάδας του ευρώ (€) (εικόνα 386).

```
      <mat-step label="Confirmation">
          <app-checkout-review></app-checkout-review>
          <div class="flex justify-between mt-6">
            <button matStepperPrevious mat-stroked-button>Back</button>
            <button matStepperNext mat-flat-button class="bg-green-500">
                Pay {{cartService.totals()?.total |currency:'EUR':'symbol':'1.2-2':'fr'}}
            </button>
          </div>
      </mat-step>
  </mat-stepper>
```

Εικόνα 386

Στην εικόνα 387 παρουσιάζεται το αποτέλεσμα που βλέπει ο χρήστης στο περιβάλλον του φυλλομετρητή του, καθώς έχει φτάσει στο τελευταίο επίπεδο του *mat-stepper menu*.



Εικόνα 387

12.6 Δημιουργία του Checkout-Success Component

Η τελευταία ενότητα αυτού του κεφαλαίου, αφορά τη δημιουργία ενός ακόμη *component* με τίτλο *checkout-success*. Καθώς ο χρήστης επιλέγει το κουμπί *Pay* εντός του *mat-stepper*, θα πρέπει να μεταφέρεται στη σελίδα σύνοψης κι επιτυχημένης καταχώρησης της παραγγελίας του. Στην εικόνα 388 παρουσιάζουμε την εντολή δημιουργίας του νέου *component*.

```
PS C:\Users\USER\Desktop\pcparts\client> ng g c features/checkout/checkout-success --skip-tests
CREATE src/app/features/checkout/checkout-success/checkout-success.component.ts (266 bytes)
CREATE src/app/features/checkout/checkout-success/checkout-success.component.scss (0 bytes)
CREATE src/app/features/checkout/checkout-success/checkout-success.component.html (32 bytes)
PS C:\Users\USER\Desktop\pcparts\client>
```

Εικόνα 388

Εντός του αρχείου *app.routes.ts* προσθέτουμε το *URL path* προς το συγκεκριμένο *component*, καθώς και της *Angular guard* κλάσεις *authGuard* και *cartEmptyGuard* (εικόνα 389).

```
25    {path:'success', component: CheckoutSuccessComponent,canActivate: [authGuard,cartEmptyGuard]},
```

Εικόνα 389

Στην κλάση *CheckoutSuccessComponent* κάνουμε *inject* τις κλάσεις *CartService* και *AccountService*, καθώς και την κλάση *Date* στη μεταβλητή *today* (εικόνα 390).

```
19  export class CheckoutSuccessComponent {
20
21    cartService = inject(CartService);
22    currentUser = inject(AccountService);
23
24    today: number = Date.now(); //get current date
25
26  }
```

Εικόνα 390

Τέλος, δημιουργούμε ένα *<section> menu* με * attributes* τριών επιπέδων. Πρακτικά πρόκειται για μία σελίδα ενημερωτικού χαρακτήρα προς τον χρήστη, στην περίπτωση που έχει ολοκληρωθεί η διαδικασία πληρωμής και η παραγγελία του έχει καταχωρηθεί επιτυχώς στην πλατφόρμα μας. Ουσιαστικά δεν πρόκειται για μία επιπλέον λειτουργικότητα της εφαρμογής μας, παρά μόνο για μία εικονική αποτίμηση των ενεργειών του. Στο πρώτο επίπεδο της σελίδας εμφανίζεται ένα ευχαριστήριο μήνυμα για την εικονική καταχώρηση της παραγγελίας και στο δεύτερο επίπεδο, σε γκρίζο φόντο και σε χρωματισμό *text-gray-500*, οι τίτλοι *Date*, *Payment method*, *Payment Location* και *Amount*. Δίπλα σε κάθε τίτλο παρουσιάζεται το περιεχόμενο της συγκεκριμένης πληροφορίας σε *text-gray-900* απόχρωση, μέσω των οντοτήτων *Date* (για την ημερομηνία σε πραγματικό χρόνο) και *currentUser* για τα στοιχεία διεύθυνσης. Τέλος, στο τρίτο επίπεδο δημιουργούμε το *routerLink mat-stroked button* «*Continue Shopping*», στο οποίο εισάγουμε το *(click)="cartService.deleteCart()" attribute*. Με αυτόν τον τρόπο ο χρήστης επιστρέφει στη σελίδα *Shop* του ηλεκτρονικού μας καταστήματος και ταυτόχρονα διαγράφεται το υπάρχον καλάθι αγορών του από τον *Redis server*, τα *Cookies* του φυλλομετρητή του και τα *Cart related components* της *Angular* εφαρμογής μας (εικόνα 391).

checkout-success.component.html

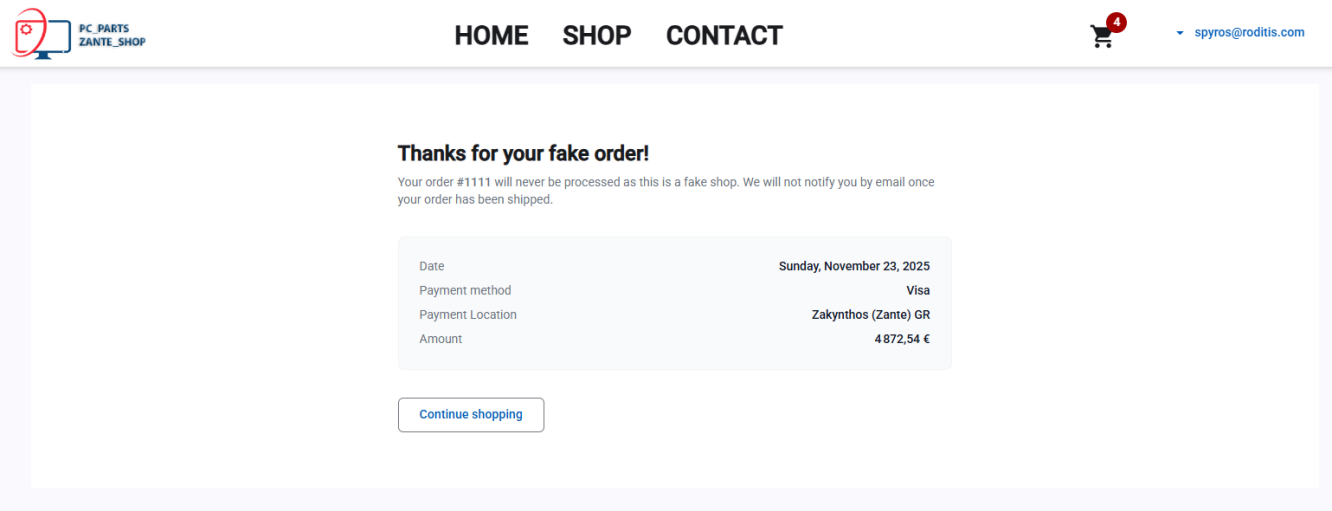
client > src > app > features > checkout > checkout-success > checkout-success.component.html > ...

Go to component

```
@if(cartService.cart()?.id){
<section class="bg-white py-16">
    <div class="mx-auto max-w-2xl px-4">
        <h2 class="font-semibold text-2xl mb-2">Thanks for your fake order!</h2>

        <p class="text-gray-500 mb-8">Your order <span
            class="font-medium">#1111</span> will never be processed as this
            is a fake shop. We will not notify you by email once your order has
            been shipped.</p>
        <div
            class="space-y-2 rounded-lg border border-gray-100 bg-gray-50 p-6 mb-8">
            <dl class="flex items-center justify-between gap-4">
                <dt class="font-normal text-gray-500">Date</dt>
                <dd class="font-medium text-gray-900 text-end">
                    {{today | date:'fullDate'}}
                </dd>
            </dl>
            <dl class="flex items-center justify-between gap-4">
                <dt class="font-normal text-gray-500">Payment method</dt>
                <dd class="font-medium text-gray-900 text-end">Visa</dd>
            </dl>
            <dl class="flex items-center justify-between gap-4">
                <dt class="font-normal text-gray-500">Payment Location</dt>
                <dd class="font-medium text-gray-900 text-end">
                    {{currentUser.currentUser()?.address?.city}}
                    {{currentUser.currentUser()?.address?.country}}
                </dd>
            </dl>
            <dl class="flex items-center justify-between gap-4">
                <dt class="font-normal text-gray-500">Amount</dt>
                <dd class="font-medium text-gray-900 text-end">
                    {{cartService.totals()?.total |currency:'EUR':'symbol':'1.2-2':'fr'}}
                </dd>
            </dl>
        </div>
        <div class="flex items-center space-x-4">
            <button routerLink="/shop" mat-stroked-button (click)="cartService.deleteCart()">
                Continue shopping</button>
        </div>
    </div>
</section>
}
```

Εικόνα 391

Εκκινώντας την εφαρμογή *Angular*, στο περιβάλλον του φυλλομετρητή μας μεταβαίνουμε στο τέταρτο επίπεδο του *mat-stepper menu* «*Confirmation*» κι επιλέγουμε το πράσινο κουμπί *Pay*. Αμέσως, καθώς βρισκόμαστε σε περιβάλλον ανάπτυξης της εφαρμογής και σε *localhost server*, ο χρήστης ανακατευθύνεται στη νέα σελίδα του *checkout-success component*. Στην εικόνα 392 παρουσιάζεται το αποτέλεσμα μίας ολοκληρωμένης παραγγελίας στην εφαρμογή *PcParts*.



Εικόνα 392

Συμπεράσματα

Η παρούσα διπλωματική εργασία ανέδειξε τη διαδικασία σχεδιασμού και υλοποίησης μιας σύγχρονης, ολοκληρωμένης *e-commerce* διαδικτυακής εφαρμογής, αξιοποιώντας σύγχρονες αρχιτεκτονικές πρακτικές και τεχνολογίες, τόσο στο *backend* όσο και στο *frontend* περιβάλλον της. Μέσα από την ανάπτυξη μιας καθαρής κι επεκτάσιμης αρχιτεκτονικής στο *.NET Core 9*, την εφαρμογή σχεδιαστικών προτύπων όπως τα *Repository*, *Generic Repository* και *Specification Pattern*, καθώς και την αξιοποίηση του *Entity Framework*, επιτεύχθηκε μια στιβαρή κι ευέλικτη βάση δεδομένων καθώς και το *API Project*.

Στο *frontend*, η εφαρμογή της *Angular 20* αξιοποίησε σύγχρονες δυνατότητες όπως τα *Angular Signals*, οι *Reactive Forms*, το *modular routing* και η *Tailwind CSS*, επιτρέποντας τη δημιουργία μιας δυναμικής κι εύχρηστης διεπαφής χρήστη. Η ενσωμάτωση του *Redis Server* ως *in-memory data store* βελτίωσε σημαντικά τη διαχείριση του καλαθιού αγορών, ενώ η υλοποίηση μηχανισμών *authentication* και *authorization* μέσω του *ASP.NET Core Identity* προσέθεσε ασφάλεια κι αξιοπιστία στο σύστημα.

Η συνολική υλοποίηση αποδεικνύει ότι ο συνδυασμός σύγχρονων εργαλείων, δομημένων προτύπων και βέλτιστων πρακτικών, επιτρέπει την ανάπτυξη ενός επεκτάσιμου, υψηλής απόδοσης και παραγωγικού συστήματος ηλεκτρονικού εμπορίου. Η εργασία αυτή μπορεί να αποτελέσει βάση για μελλοντικές επεκτάσεις, όπως βελτιστοποίηση της απόδοσης, προσθήκη νέων επιχειρησιακών λειτουργιών ή εφαρμογή προηγμένων τεχνικών ασφαλείας και αυτοματοποίησης.

Παραπομπές

Βιβλιογραφία

[1] Freeman, A., & Jones, S. (2022). *Pro ASP.NET Core 6: Develop cloud-ready web applications using MVC, Blazor, and Razor Pages*. Apress.

[2] Freeman, A., & Vogels, J. (2022). *Pro Angular: Build powerful and dynamic web apps*. Apress.

[3] Freeman, A., & Le, S. (2021). *Pro ASP.NET Core Identity: Under the hood with authentication and authorization in ASP.NET Core 5 and 6 applications*. Apress. [4] A. Troelsen, Pro C# 5.0 and the .NET 4.5 Framework, Sixth. Berkeley, Ca Apress, 2012.

[4] Freeman, A. (2019). *Essential TypeScript: From beginner to pro* (1st ed.). Apress.

[5] J. Duckett, HTML & CSS: Design and Build Websites (HTML and CSS). John Wiley & Sons Incorporated, 2011.

[6] J. Smith, Entity Framework Core in Action. Manning Publications Company, 2018.

Ιστοσελίδες

- Cummings, N. (2025, July). *Learn to Build an E-Commerce App with .NET Core & Angular* [Online course]. Udemy. https://www.udemy.com/course/learn-to-build-an-e-commerce-app-with-net-core-and-angular/?coupon
- Microsoft. (2024). *ASP.NET Core Identity Documentation*, https://learn.microsoft.com/aspnet/core/security/authentication/identity
- Microsoft. (2024). *ASP.NET Core Documentation*, https://learn.microsoft.com/aspnet/core
- Microsoft. (2024). *Entity Framework Core Documentation*, https://learn.microsoft.com/ef/core
- Microsoft. (2024). *.NET Documentation*, https://learn.microsoft.com/dotnet
- Angular Team. (2024). *Angular Documentation*, https://angular.io/docs
- Redis Ltd. (2024). *Redis Documentation*, https://redis.io/docs
- Stripe. (2024). *Stripe API Reference*, https://stripe.com/docs/api