ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Σχεδίαση, Ανάπτυξη και Υλοποίηση διαδικτυακού περιβάλλοντος για την προώθηση και πώληση ηλεκτρικών προϊόντων

ΑΝΤΩΝΙΟΥ ΓΡΗΓΟΡΙΟΣ, ΑΔΑΜΟΠΟΥΛΟΣ ΘΕΟΔΩΡΟΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

ΤΖΙΑΛΛΑΣ ΓΡΗΓΟΡΙΟΣ
.....Βαθμίδα.....

ΣΥΝΕΠΙΒΛΕΠΩΝ
(εφόσον υπάρχει)

.....Ονοματεπώνυμο Υπεύθυνου.....
.....Βαθμίδα.....

Λαμία ..... έτος 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Σχεδίαση, Ανάπτυξη και Υλοποίηση διαδικτυακού περιβάλλοντος για την προώθηση και πώληση ηλεκτρικών προϊόντων

ΑΝΤΩΝΙΟΥ ΓΡΗΓΟΡΙΟΣ, ΑΔΑΜΟΠΟΥΛΟΣ ΘΕΟΔΩΡΟΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

ΤΖΙΑΛΛΑΣ ΓΡΗΓΟΡΙΟΣ
.....Βαθμίδα.....

ΣΥΝΕΠΙΒΛΕΠΩΝ
(εφόσον υπάρχει)

.....Ονοματεπώνυμο Υπεύθυνου.....
.....Βαθμίδα.....

Λαμία ..... έτος 2025

UNIVERSITY OF THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE & TELECOMMUNICATIONS

Design, Development, and Implementation of an Online Environment for the Promotion and Sale of Electrical Products.

ANTONIOU GRIGORIOS, ADAMOPOULOS THEODOROS

FINAL THESIS

ADVISOR

TZIALLAS GRIGORIOS
.....Position.....

CO ADVISOR

.....Full Name.....
.....Position.....

Lamia ..... year 2025

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

*1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί **χωρίς να τα περικλείω σε εισαγωγικά** και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

*2. Δέχομαι ότι η αυτολεξεί **παράθεση χωρίς εισαγωγικά**, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

GRIGORIS ANTONIOU

Ημερομηνία: ...../...../20.....

Ο - Η Δηλ.

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση
του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων
σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»

ΠΕΡΙΛΗΨΗ

Η παρούσα εργασία αφορά τη σχεδίαση, ανάπτυξη και υλοποίηση ενός ηλεκτρονικού καταστήματος (e-shop), το οποίο αναπτύχθηκε με τεχνολογίες full-stack web development. Το front-end της εφαρμογής έχει υλοποιηθεί με React.js, HTML και CSS, προσφέροντας μια δυναμική εμπειρία χρήστη, ενώ το back-end αναπτύχθηκε με Express.js και MongoDB, επιτρέποντας την αποθήκευση και διαχείριση των δεδομένων μέσω μιας NoSQL βάσης δεδομένων.

Η πλατφόρμα υποστηρίζει βασικές λειτουργίες ενός ηλεκτρονικού καταστήματος, όπως διαχείριση χρηστών, παρουσίαση προϊόντων, σύστημα αναζήτησης, φίλτρα κατηγοριών, προσθήκη προϊόντων στο καλάθι αγορών, επεξεργασία παραγγελιών.

Στο front-end, η χρήση του React.js επέτρεψε τη δημιουργία διεπαφής χρήστη βασισμένη σε components (Component-based UI) με διαχείριση μέσω hooks και context API, βελτιστοποιώντας τη λειτουργικότητα και την απόδοση της εφαρμογής. Παράλληλα, το back-end στηρίχθηκε στο Node.js με Express.js, διασφαλίζοντας RESTful API endpoints για την επικοινωνία μεταξύ client και server. Η MongoDB χρησιμοποιήθηκε για την αποθήκευση δεδομένων προϊόντων και χρηστών, με Mongoose ORM να διαχειρίζεται τις λειτουργίες της βάσης δεδομένων.

ABSTRACT

This paper deals with the design, development and implementation of an online store, which was developed using full-spectrum web development technologies. The front-end of the application was implemented using React.js, HTML and CSS, providing a dynamic user experience, while the back-end was developed using Express.js and MongoDB, allowing data to be stored and managed via a NoSQL database.

The platform supports the basic functions of an online store, such as user management, product presentation, search system, category filters, adding products to the shopping cart, order processing.

On the front-end, the use of React.js enabled the creation of a component-based UI managed through hooks and context APIs, optimizing the functionality and performance of the application. At the same time, the back-end was built in Node.js with Express.js, ensuring RESTful API endpoints for communication between client and server. MongoDB was used to store product and user data, with Mongoose ORM managing the database operations.

Table of Contents

ΚΕΦΑΛΑΙΟ 1 Εισαγωγή

1.1 Περιγραφή του E-Commerce Συστήματος

(Ενότητα 1.1)

Η παρούσα εργασία αφορά την ανάπτυξη ενός σύγχρονου e-commerce συστήματος που επιτρέπει την προώθηση και πώληση ηλεκτρονικών προϊόντων μέσω διαδικτύου. Το σύστημα ακολουθεί μια full-stack αρχιτεκτονική, αποτελούμενη από frontend και backend τμήματα, αξιοποιώντας σύγχρονες τεχνολογίες για τη δημιουργία μιας αποδοτικής και επεκτάσιμης πλατφόρμας.

1.2 Frontend (React.js, HTML, CSS, JavaScript)

(Ενότητα 1.2)

Το frontend έχει υλοποιηθεί με React.js, χρησιμοποιώντας component-based architecture για τη δημιουργία δυναμικών διεπαφών χρήστη. Η αισθητική και η διάταξη της ιστοσελίδας σχεδιάστηκαν με CSS και HTML, ενώ η διαδραστικότητα υλοποιήθηκε μέσω JavaScript και React hooks.

Οι βασικές λειτουργίες του frontend περιλαμβάνουν:

- Δυναμικό σύστημα αναζήτησης προϊόντων και φίλτρα κατηγοριών.
- Διαχείριση χρηστών (εγγραφή, σύνδεση, authentication με JWT).
- Καλάθι αγορών και σύστημα checkout.
- Εφαρμογή responsive design με προσαρμογή σε όλες τις συσκευές (mobile-first approach).

1.3 Backend (Node.js, Express.js, MongoDB)

(Ενότητα 1.3)

Το backend υλοποιήθηκε με Node.js και Express.js, επιτρέποντας τη διαχείριση των δεδομένων και τη δημιουργία RESTful API endpoints που επικοινωνούν με το frontend. Η MongoDB χρησιμοποιήθηκε ως βάση δεδομένων για την αποθήκευση πληροφοριών χρηστών, προϊόντων και παραγγελιών.

Οι κύριες λειτουργίες του backend περιλαμβάνουν:

- Διαχείριση προϊόντων (προσθήκη, ενημέρωση, διαγραφή μέσω Admin Panel).
- Διαχείριση χρηστών με authentication μέσω JSON Web Tokens (JWT).

Το σύστημα έχει σχεδιαστεί για να είναι ασφαλές, επεκτάσιμο και εύχρηστο, επιτρέποντας μελλοντικές βελτιώσεις.

ΚΕΦΑΛΑΙΟ 2 Βιβλιογραφική Επισκόπηση

2.1 Εκπαιδευτικά Βίντεο (Udemy, YouTube)

(Ενότητα 2.1.)

Τα μαθήματα στο Udemy αποτέλεσαν βασικό εργαλείο για την κατανόηση και **εφαρμογή** των full-stack τεχνολογιών που χρησιμοποιήθηκαν στην εργασία. Μέσα από αυτά, μελετήθηκαν έννοιες όπως:

- Η διαχείριση κατάστασης (State Management) με React hooks και Context API.
- Η δημιουργία RESTful APIs με Express.js και Node.js.
- Η διαχείριση βάσεων δεδομένων με MongoDB και Mongoose ORM.
- Οι αρχές ασφαλείας authentication & authorization με JWT tokens.

Αντίστοιχα, αξιοποιήθηκαν tutorials και case studies στο YouTube, σχετικά με την ανάπτυξη UI με React.js και τη βελτίωση της απόδοσης ενός e-commerce site.

2.2 Wikipedia και Άρθρα

(Ενότητα 2.2.)

Για να κατανοήσουμε καλύτερα τις τεχνολογίες που χρησιμοποιήθηκαν, ανατρέξαμε σε διάφορες πηγές πληροφόρησης, τόσο διαδικτυακές όσο και έντυπες.

Αρχικά, αξιοποιήσαμε άρθρα και αναλύσεις που εξηγούν τη λειτουργία των NoSQL βάσεων δεδομένων, εστιάζοντας ιδιαίτερα στη MongoDB. Παράλληλα, μελετήσαμε τον τρόπο με τον οποίο λειτουργούν οι single-page applications (SPA) και πώς το React.js επιτρέπει τη δημιουργία γρήγορων και δυναμικών web εφαρμογών. Επίσης, αναζητήσαμε πληροφορίες για το Node.js, το οποίο αποτελεί τη βάση του backend, κατανοώντας καλύτερα πώς λειτουργεί ως περιβάλλον εκτέλεσης της JavaScript στον server.

Πέρα από τις διαδικτυακές πηγές, συμβουλευτήκαμε και βιβλία που καλύπτουν σε βάθος τις τεχνολογίες που χρησιμοποιήθηκαν:

- "Node.js Design Patterns" – Ένα βιβλίο που αναλύει στρατηγικές και βέλτιστες πρακτικές για τη σωστή δομή εφαρμογών σε Node.js.
- "Eloquent JavaScript" – Ένα εγχειρίδιο που ξεκινά από τα βασικά της JavaScript και προχωρά σε πιο προχωρημένες τεχνικές προγραμματισμού.
- "HTML and CSS: Design and Build Websites" – Ένας οδηγός για τη δημιουργία καλοσχεδιασμένων και λειτουργικών ιστοσελίδων.
- "JavaScript & jQuery: Interactive Front-End Web Development" – Ένα βιβλίο που δείχνει πώς η JavaScript και το jQuery μπορούν να κάνουν τις ιστοσελίδες πιο διαδραστικές.

Συνδυάζοντας τις διαδικτυακές πληροφορίες με τις πιο εξειδικευμένες γνώσεις από αυτά τα βιβλία, καταφέραμε να έχουμε μια πιο ολοκληρωμένη εικόνα για την ανάπτυξη της εφαρμογής, ακολουθώντας παράλληλα τις βέλτιστες πρακτικές.

ΚΕΦΑΛΑΙΟ 3 FRONT END

3.1 Εισαγωγή

(Ενότητα 3.1.)

Στο επόμενο κεφάλαιο, θα δούμε πώς διαμορφώνεται το Front-End ενός ηλεκτρονικού καταστήματος, δηλαδή το κομμάτι της ιστοσελίδας που βλέπει και χρησιμοποιεί ο χρήστης. Είναι αυτό που κάνει την εμπειρία πλοήγησης ομαλή, γρήγορη και ευχάριστη, επιτρέποντας στους επισκέπτες να βρίσκουν εύκολα τα προϊόντα που ψάχνουν και να ολοκληρώνουν τις αγορές τους χωρίς δυσκολία.

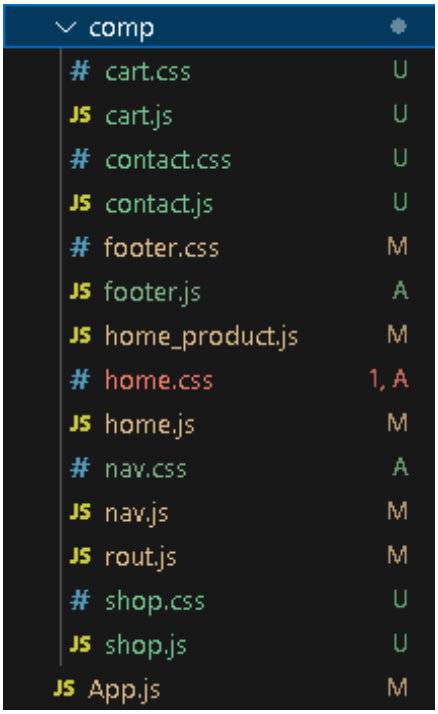
Για την ανάπτυξη του Front-End, χρησιμοποιούνται τεχνολογίες όπως HTML, CSS και JavaScript, ενώ frameworks όπως το React βοηθούν στη δημιουργία δυναμικών και διαδραστικών στοιχείων που βελτιώνουν την απόδοση της ιστοσελίδας.

Ένα σωστά σχεδιασμένο Front-End παίζει καθοριστικό ρόλο στην επιτυχία ενός e-shop, καθώς η εμφάνιση, η ταχύτητα και η ευκολία χρήσης επηρεάζουν σημαντικά τη συμπεριφορά των επισκεπτών και την πιθανότητα να προχωρήσουν σε αγορά.

3. 2 Δομή αρχείων

(Ενότητα 3.2.)

Δομή Φακέλου comp



Ο φάκελος περιέχει αρχεία CSS και JavaScript που είναι οργανωμένα ανά λειτουργικότητα, ενότητα της εφαρμογής.

Κάθε αρχείο έχει συγκεκριμένο σκοπό, όπως φαίνεται παρακάτω:

- **cart.css και cart.js:** Στυλιστική και λειτουργική υποστήριξη για το στοιχείο καλαθιού αγορών.
- **contact.css και contact.js:** Στυλ και λειτουργικότητα για τη σελίδα επικοινωνίας.
- **footer.css και footer.js:** Διαχείριση της εμφάνισης και της λειτουργικότητας του footer.
- **home.css και home_product.js:** Ειδική διαμόρφωση για την αρχική σελίδα και τη διαχείριση προϊόντων.
- **nav.css και nav.js:** Υπεύθυνα για την πλοήγηση (navigation) της ιστοσελίδας.

- **shop.css και shop.js:** Σχεδιασμένα για τη διαμόρφωση της σελίδας καταστήματος.

Βασικά Αρχεία Εφαρμογής

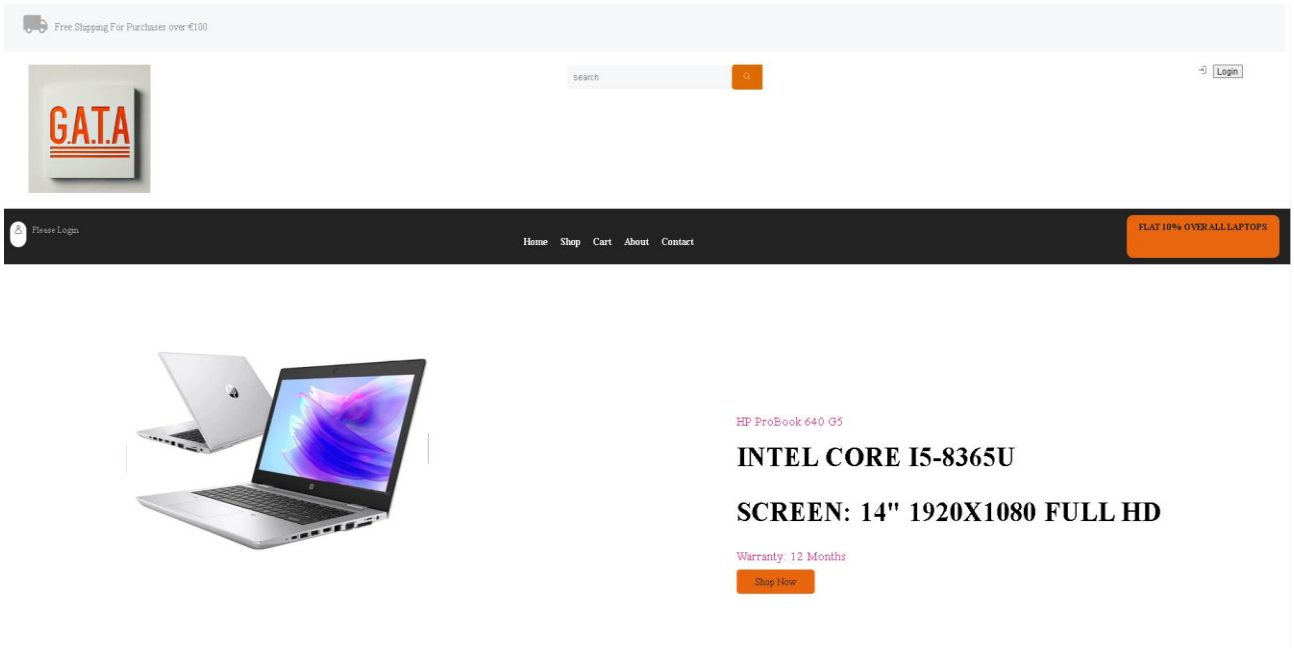
- **App.js:** Το κύριο αρχείο που συνδέει τις διαφορετικές ενότητες της εφαρμογής και διαχειρίζεται τη δρομολόγηση.
- **index.js:** Αρχείο εισόδου της εφαρμογής που φορτώνει το App.js.
- **setupTests.js:** Αρχείο που χρησιμοποιείται για ρυθμίσεις δοκιμών (testing).

Ρόλος των Αρχείων

- Τα αρχεία CSS είναι υπεύθυνα για την αισθητική και τη διάταξη των σελίδων.
- Τα αρχεία JavaScript παρέχουν τη διαδραστικότητα και τη λειτουργικότητα της εφαρμογής.
- Η οργάνωση αυτή διευκολύνει τη συντήρηση και την ανάπτυξη του Project, διαχωρίζοντας την εμφάνιση από τη λογική

3. 3 Ανάλυση του Κώδικα

(Ενότητα 3.3.)

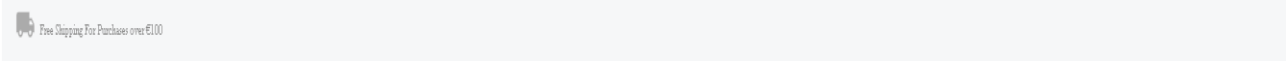


Φωτογραφία 1 -Header και Top Banner

Στην παραπάνω φωτογραφία, φαίνονται:

- Το επάνω μέρος της σελίδας περιλαμβάνει το header και το **Top Banner**.
- Υπάρχει ένα μήνυμα στην κορυφή που αναφέρει "Free Shipping For Purchases over €100".
- Το λογότυπο μας ("G.A.T.A") εμφανίζεται αριστερά.
- Στο κέντρο υπάρχει μια μπάρα αναζήτησης με κουμπί αναζήτησης (Search).
- Το κύριο μενού πλοήγησης περιλαμβάνει επιλογές όπως **Home, Shop, Cart, About, Contact**.
- Υπάρχει ένα banner που αναδεικνύει το προϊόν **HP ProBook 640 G5** με περιγραφή (οθόνη, εγγύηση, κ.λπ.) και ένα κουμπί "Shop Now".
- Στα δεξιά υπάρχει κουμπί Login, που το πατάει ο χρήστης για να συνδεθεί με τον λογαριασμό του.

Το πρώτο μήνυμα που εμφανίζεται είναι το παρακατω.



Για να δημιουργηθεί αλλά και για να μορφοποιηθεί χρειάστηκε το παρακάτω κομμάτι κώδικα.

```jsx
import React from 'react';
import { MdLocalShipping } from "react-icons/md";
import './nav.css'
import { CiSearch } from "react-icons/ci";
import { CiLogin,CiUser } from "react-icons/ci";
import { CiLogout } from "react-icons/ci";
import { useAuth0 } from "@auth0/auth0-react";
import { Link } from 'react-router-dom';




const Nav = ({search,setSearch, searchproduct}) => {
    const { loginWithRedirect, logout, user, isAuthenticated } = useAuth0();
    return (
        <>
        <div className='header'>
         <div className='top_header'>
             <div className='icon'>
             <MdLocalShipping />
             </div>
             <div className='info'>
                 <p>Free Shipping For Purchases over €100</p>
             </div>
         </div>
```

```css
.
{
    margin: 0;
    padding: 0;
    font-family: sans-serif;
    box-sizing: border-box;
}


.header{
    width: 100%;
}

.header .top_header{
    max-width: 100%;
    display: flex;
    padding: 10px 10px;
    background: #f6f7f8;
}

.header .top_header .icon{
    margin-left: 20px;
    color: #ababab;
    font-size: 43px;

}

.header .top_header .info p{
    margin-left: 10px;
    color: #8a8a8a;
}
```

Στον κώδικα **JSX** έγινε εισαγωγή (import) του εικονιδίου από τη βιβλιοθήκη react-icons. Επιπλέον, δημιουργήθηκαν τέσσερα στοιχεία με σκοπό να επιτρέψουν την μορφοποίηση του μηνύματος σε μεταγενέστερο στάδιο μέσω του **CSS styling**. Τα που δημιουργήθηκαν είναι, κατά σειρά, τα εξής: header, top_header, icon και info.

Τέλος, στον CSS κώδικα στο πεδίο των **header, top_header** δίνονται οι διαστάσεις του πεδίου που θέλουμε να φαίνεται το μήνυμα μαζί με το εικονίδιο, ενώ στο icon και info μορφοποιείται το εικονίδιο και το μήνυμα αντίστοιχα.

Για το λογότυπο, το search και το login button χρησιμοποιήθηκε ο παρακάτω κώδικας :

```
import React from 'react';
import { MdLocalShipping } from "react-icons/md";
import './nav.css'
import { CiSearch } from "react-icons/ci";
import { CiLogin,CiUser } from "react-icons/ci";
import { CiLogout } from "react-icons/ci";
import { useAuth0 } from "@auth0/auth0-react";
import { Link } from 'react-router-dom';


              (parameter) search: any
const Nav = ({search,setSearch, searchproduct}) => {
    const { loginWithRedirect, logout, user, isAuthenticated } = useAuth0();
```

```
        <div className='mid_header'>
            <div className='logo'>
                <img src='EIKONES\newlogo.jpeg' alt='newlogo'></img>
            </div>
            <div className='search_box'>
                <input type='text' value={search} placeholder='search' onChange={(e) => setSearch(e.target.value)}></input>
                <button onClick={searchproduct}><CiSearch /></button>
            </div>
                {
                    isAuthenticated ?
                    //if user is login then logout button will shown  and also user profile
                    <div className='user'>
                        <div className='icon'>

                          <CiLogout />
                        </div>
                     <div className='btn'>
                        <button onClick={() => logout({ logoutParams: { returnTo: window.location.origin } })}>Logout</button>
                     </div>
                    </div>
```

1. Λογότυπο:

- Βρίσκεται σε ένα div με className='logo'.
- Χρησιμοποιεί μια εικόνα (<img>)
- Το src δείχνει προς το αρχείο newlogo.jpeg, που βρίσκεται σε φάκελο με εικόνες (EIKONES)

2. Πλαίσιο Αναζήτησης (Search Box):

JS App.js

```
import React, { useState } from 'react';
import Nav from './comp/nav';
import {BrowserRouter} from 'react-router-dom';
import Rout from './comp/rout';
import Footer from './comp/footer';
import Homeproduct from './comp/home_product';

const App = () => {
  //add to cart
  const [cart, setCart] = useState([])
  //Shop page
  const [shop,setShop] = useState(Homeproduct)

    //shop search filter
    const [search,setSearch] = useState('')
```

```
const Nav = ({search,setSearch, searchproduct}) => {
```

```
<div className='search_box'>
    <input type='text' value={search} placeholder='search' onChange={(e) => setSearch(e.target.value)}></input>
    <button onClick={searchproduct}><CiSearch /></button>
</div>
```

Υλοποιείται σε ένα div με className='search_box'

type="text": Δηλώνει ότι πρόκειται για πεδίο εισαγωγής κειμένου.

value={search}: Συνδέεται με μια κατάσταση (state) search, που περιέχει την τρέχουσα τιμή του πεδίου αναζήτησης.

placeholder="search": Εμφανίζει το κείμενο "search" μέσα στο πεδίο, μέχρι να εισάγει κάτι ο χρήστης.

onChange={(e) => setSearch(e.target.value)}: Ενημερώνει την κατάσταση search κάθε φορά που ο χρήστης πληκτρολογεί κάτι.
Η **συνάρτηση setSearch** αποθηκεύει την τιμή του πεδίου (e.target.value) .

Στοιχείο button :
onClick={searchproduct}:

- Καλεί τη συνάρτηση searchproduct όταν ο χρήστης κάνει κλικ στο κουμπί.
- Αυτή η συνάρτηση εκτελεί την αναζήτηση βάσει της τιμής που είναι αποθηκευμένη στη μεταβλητή search.

<CiSearch />:

- Είναι ένα εικονίδιο (icon) που αντιπροσωπεύει τη λειτουργία της αναζήτησης.

```
//shop search filter
const [search,setSearch] = useState('')
```

App.js

Ανάλυση:

- search:
    - Η μεταβλητή που περιέχει την τρέχουσα τιμή της αναζήτησης.
    - Αρχική τιμή: Κενή συμβολοσειρά (''), δηλαδή δεν υπάρχει αναζήτηση στην αρχή.
- setSearch:
    - Η συνάρτηση που χρησιμοποιείται για την ενημέρωση της κατάστασης search.
    - Όποτε ο χρήστης πληκτρολογεί κάτι στο πεδίο αναζήτησης, αυτή η συνάρτηση καλείται με την καινούρια τιμή.
- useState(''):
    - Δηλώνει ότι η αρχική κατάσταση της μεταβλητής search είναι κενή ('').

Η λειτουργία αναζήτησης (search) μεταβιβάζεται μέσω props στα components της εφαρμογής.

Ο κώδικας βρίσκεται μέσα στο return της App component, και είναι υπεύθυνος για τη ρύθμιση της δρομολόγησης (BrowserRouter) και τη σύνδεση των child components με τα δεδομένα και τις λειτουργίες που χρειάζονται.

Props που περνιούνται:

- search={search}:
    - Η τρέχουσα τιμή της κατάστασης αναζήτησης (search) περνιέται ως prop στο Nav component.
    - Χρησιμοποιείται για να εμφανίζει την τιμή που πληκτρολογεί ο χρήστης στο πεδίο αναζήτησης.
- setSearch={setSearch}:
    - Η συνάρτηση για την ενημέρωση της κατάστασης search περνιέται επίσης στο Nav.
    - Στο Nav, αυτή η συνάρτηση καλείται κάθε φορά που ο χρήστης πληκτρολογεί κάτι.
- searchproduct={searchproduct}:
    - Μια συνάρτηση που εκτελεί τη λογική αναζήτησης (π.χ., φιλτράρισμα δεδομένων ή API call).
    - Αυτή καλείται στο Nav, όταν ο χρήστης πατήσει το κουμπί αναζήτησης.

Το Nav component χρησιμοποιεί αυτά τα props για να παρέχει τη λειτουργία της αναζήτησης:

- Το search εμφανίζει την τρέχουσα τιμή στο input field.
- Το setSearch ενημερώνει την κατάσταση αναζήτησης όταν αλλάζει το input field.
- Το searchproduct ενεργοποιείται με το πάτημα ενός κουμπιού.

<u>Λειτουργικότητα του Search στην Εφαρμογή:</u>

1. **Ο χρήστης πληκτρολογεί στο πεδίο αναζήτησης στο Nav:**

   Το onChange καλεί το setSearch και ενημερώνει την κατάσταση search.

2. **Ο χρήστης πατά το κουμπί αναζήτησης:**

   Το searchproduct καλείται, για να εκτελέσει τη λογική της αναζήτησης.

3. **Φιλτράρισμα δεδομένων:**

   Η τιμή search χρησιμοποιείται σε άλλες περιοχές της εφαρμογής (π.χ., στο Rout) για την εμφάνιση αποτελεσμάτων αναζήτησης.

<u>Συνοπτικά:</u>

1. Το search, το setSearch, και το searchproduct μεταβιβάζονται στο Nav για να διαχειριστούν τη λειτουργία αναζήτησης.
2. Το Nav επιτρέπει στον χρήστη να αλληλεπιδρά με την αναζήτηση μέσω ενός πεδίου εισαγωγής και ενός κουμπιού.
3. Η αναζήτηση λειτουργεί σε συνδυασμό με άλλες καταστάσεις ή δεδομένα που περνιούνται σε άλλα components μέσω του Rout.

Αυτό το κομμάτι κώδικα υλοποιεί τη λειτουργικότητα που σχετίζεται με το προφίλ του χρήστη και την κατάσταση σύνδεσής του (login/logout).

```
                <div className='user'>
                    <div className='icon'>
                    <CiLogin />
                    </div>
                     <div className='btn'>
                          <button onClick={() => loginWithRedirect()} >Login </button>
                      </div>
                 </div>

                }


            </div>
                <div className='last_header'>
                    <div className='user_profile'>
                        {
                            //user profile is shown here
                            isAuthenticated ?
                            <>
                                <div className='icon'>
                                    <CiUser />
                                </div>
                                <div className='info'>
                                    <h2>{user.name}</h2>
                                    <p>{user.email}</p>
                                </div>
```

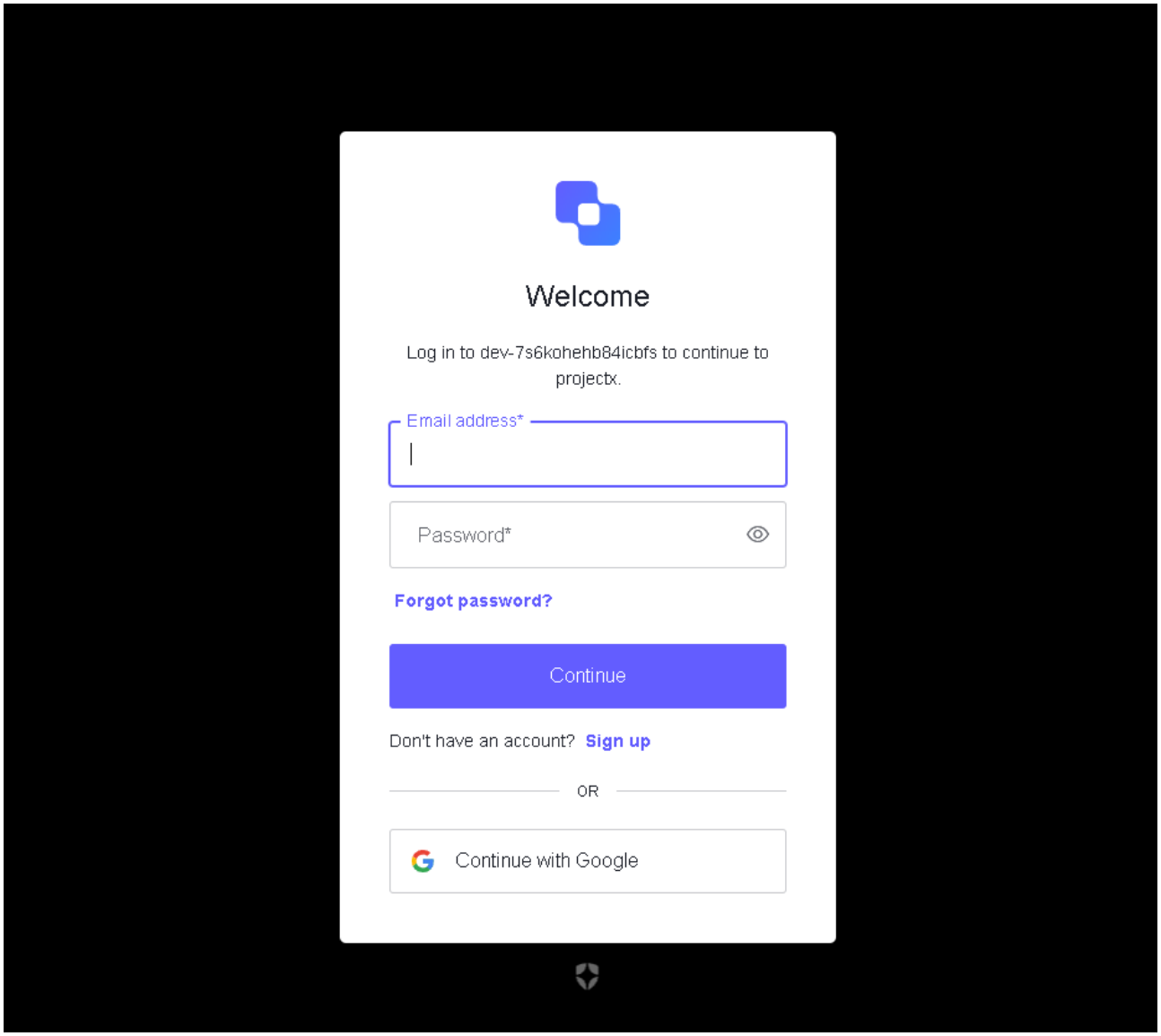
```
                    :
                    <>
                        <div className='icon'>
                            <CiUser />
                        </div>
                        <div className='info'>
                            <p>Please Login</p>
                        </div>
                    </>
                }
            </div>

```

Δομή:

Ένα div με className='user' περιέχει το εικονίδιο CiLogin και ένα κουμπί Login.

Λειτουργικότητα:

Το κουμπί καλεί τη συνάρτηση loginWithRedirect() όταν πατηθεί, ανακατευθύνοντας τον χρήστη στη σελίδα σύνδεσης.



Περίπτωση 1 : **Όταν ο χρήστης είναι συνδεδεμένος (isAuthenticated)**

- **Δομή:**
  Εμφανίζεται το εικονίδιο CiUser και πληροφορίες για το χρήστη:
    - user.name: Το όνομα του χρήστη.
    - user.email: Το email του χρήστη.
- Λειτουργικότητα:
    - Η κατάσταση isAuthenticated ελέγχει αν ο χρήστης είναι συνδεδεμένος.
    - Εμφανίζονται τα προσωπικά στοιχεία του χρήστη σε ένα h2 και ένα p.

Περίπτωση 2 : **Όταν ο χρήστης δεν είναι συνδεδεμένος**

- **Δομή**:
    - Εμφανίζεται το εικονίδιο CiUser και το μήνυμα: "Please Login".
- **Λειτουργικότητα:**
    - Το μήνυμα καλεί τον χρήστη να συνδεθεί.

Ο κώδικας ελέγχει αν ο χρήστης είναι συνδεδεμένος χρησιμοποιώντας την κατάσταση isAuthenticated.

- Αν ναι, εμφανίζει το προφίλ του χρήστη.
- Αν όχι, εμφανίζει το μήνυμα "Please Login".

Δομή του μενού πλοήγησης:

```
<div className='nav'>
    <ul>
        <li><Link to='/' className='link'>Home</Link></li>
        <li><Link to='/shop' className='link'>Shop</Link></li>
        <li><Link to='/cart' className='link'>Cart</Link></li>
        <li><Link to='/about' className='link'>About</Link></li>
        <li><Link to='/contact' className='link'>Contact</Link></li>
    </ul>
</div>
```

- Το μενού περιλαμβάνει μια λίστα (<ul>), με κάθε στοιχείο της λίστας (<li>) να περιέχει έναν σύνδεσμο (<Link>).
- Κάθε Link έχει τα εξής χαρακτηριστικά:
    - to: Δηλώνει τη διαδρομή (path) στην οποία οδηγεί ο σύνδεσμος. Για παράδειγμα:

- /: Αρχική σελίδα.
- /shop: Σελίδα καταστήματος.
- /cart: Καλάθι αγορών.
- /about: Σελίδα πληροφοριών.
- /contact: Σελίδα επικοινωνίας.

o className="link": Εφαρμόζει ένα συγκεκριμένο στυλ στους συνδέσμους μέσω CSS.

Λειτουργικότητα του μενού πλοήγησης**:**

- Το Link είναι στοιχείο του React Router και χρησιμοποιείται για να διαχειρίζεται πλοήγηση μέσα στην εφαρμογή χωρίς επαναφόρτωση της σελίδας.

Προσφορά:

```
<div className='offer'>
    <p>flat 10% over all Laptops</p>
</div>
```

Δομή:

Το στοιχείο div με className="offer" περιέχει μια παράγραφο (<p>) που αναφέρει την προσφορά:

- **"flat 10% over all Laptops"**: Ενημερώνει τους χρήστες για μια σταθερή έκπτωση 10% σε όλους τους φορητούς υπολογιστές

```
.
{
    margin: 0;
    padding: 0;
    font-family: sans-serif;
    box-sizing: border-box;
}


.header{
    width: 100%;
}

.header .top_header{
    max-width: 100%;
    display: flex;
    padding: 10px 10px;
    background: #f6f7f8;
}

.header .top_header .icon{
    margin-left: 20px;
    color: #ababab;
    font-size: 43px;

}

.header .top_header .info p{
    margin-left: 10px;
    color: #8a8a8a;
}
```

Γενικές Ρυθμίσεις

- Επαναφορά περιθωρίων και αποστάσεων όλων των στοιχείων.
- Ορισμός της γραμματοσειράς sans-serif.
- Χρήση του box-sizing: border-box για καλύτερη διαχείριση των διαστάσεων των στοιχείων.

Κεντρική Κεφαλίδα (.header)

Η κλάση .header ορίζει το συνολικό πλάτος της κεφαλίδας στο 100%.

Άνω Μέρος Κεφαλίδας (.top_header)
Τοποθέτηση στοιχείων οριζόντια (flexbox).

- Ανοιχτόχρωμο φόντο (#f6f7f8).
- Εμφάνιση εικονιδίων (.icon) και κειμένου πληροφοριών (.info).

```
}

.header .mid_header{
    display: flex;
    padding: 20px 40px;
    max-width: 100%;
    justify-content: space-between;

}

.header .mid_header .logo img{

    cursor: pointer;
    object-fit: cover;
    width: 200px;
    background-color: #fff;


}

.header .mid_header .search_box input
{
      padding: 10px 10px;
      outline: none;
      background: none;
      border-top-left-radius: 5px;
      border-bottom-left-radius: 5px;
      width: 250px;
      border: none;
      background: #f6f7f8;

}

```

Μέσο Μέρος Κεφαλίδας (.mid_header)

Διάταξη που χωρίζει τα στοιχεία σε διάφορες ενότητες:

- **Λογότυπο (**.logo**)**: Εικόνα με μέγιστο πλάτος 200px και λευκό φόντο.
- **Πεδίο Αναζήτησης (**.search_box**)**:

Είσοδος αναζήτησης με στρογγυλεμένες γωνίες.

- Κουμπί αναζήτησης με χρώμα #dd6b00 και εφέ hover (κίτρινο φόντο #fed700).

```css
.header .mid_header .search_box button
{
    padding: 10px 10px;
    border-top-left-radius: 5px;
    border-bottom-left-radius: 5px;
    border:none;
    outline: none;
    border:1px solid #dd6b00;
    width: 50px;
    background: #dd6b00;
    color: #fff;
    cursor: pointer;
    margin-right: 30px;
    transition: 0.5s;
}

.header .mid_header .search_box button:hover
{
    background: #fed700;
    border:1 px solid #fed700;
    color: #cd1e76;
}

.header .mid_header .user
{
    display: flex;
    margin-right: 30px;

}

```

```css

.header .mid_header .user .icon
{
    margin-right: 10px;
}

.header .mid_header .user .icon .btn button
{
    border: none;
    outline: none;
    font-size: 16px;
    background: none;
    cursor: pointer;
}

.header .last_header
{
    width: 100%;
    padding: 10px 10px;
    display:  flex;
   background: #232322;
   justify-content: space-between;
}

.header .last_header .user_profile
{
    display: flex;
    margin-top: 10px;
}

```

Ενότητα Χρήστη (.user):

Περιλαμβάνει εικονίδια και κουμπιά.

Κάτω Μέρος Κεφαλίδας (.last_header)

- Σκούρο φόντο (#232322).

Διαχωρισμός σε τρεις ενότητες:

1. **Προφίλ Χρήστη (**.user_profile**)**:

- Κυκλικό εικονίδιο με λευκό φόντο και πληροφορίες χρήστη.
- Επικεφαλίδα (h2) και παράγραφος (p) με διαφορετικά χρώματα κειμένου.

2. **Πλοήγηση (**.nav**)**:

- Μενού πλοήγησης σε μορφή λίστας (ul).
- Σύνδεσμοι (.link) με εφέ hover που αλλάζουν σε πορτοκαλί (#e9660f).

3. **Προσφορά (**.offer**)**:

- Εμφάνιση προσφορών με πορτοκαλί φόντο και κείμενο σε έντονο στυλ.

```css
.header .last_header .user_profile .icon
{

    height: 30px;
    padding: 5px 5px;
   background: #fff;
   border-radius: 50px;

}

.header .last_header .user_profile .info
{
    margin-left: 10px;
    line-height:1px;



}

.header .last_header .user_profile .info h2
{
    color: #fff;
    font-size: 18px;
}

.header .last_header .user_profile .info p
 {
    color: #ababab;
    font-size: 14px;
 }
```

```css
.header .last_header .nav
{
   margin-top: 15px;
}

.header .last_header .nav ul
{
      display: flex;


}

.header .last_header .nav ul li
{
   list-style: none;
}

.header .last_header .nav ul li .link
{
   margin-right: 20px;
   text-decoration: none;
   color:#fff;
   font-weight: 600;
   transition: 0.5s;


}

.header .last_header .nav ul li .link:hover
{
   color : #e9660f;

}
```

```css
}

.header .last_header .offer
{
   margin-right: 20px;
   padding: 5px 20px;
   background: #e9660f;
   border-radius: 10px;
}

.header .last_header .offer p
{
   margin-top: 5px;
   text-transform: uppercase;
   color: #232322;
   font-weight: 600;
   font-size: 14px;
}
```

Λεπτομέρειες Στιλιστικών Επιλογών

- **Εφέ Hover**:

Τα κουμπιά (button) και οι σύνδεσμοι (.link) αλλάζουν χρώματα για καλύτερη αλληλεπίδραση.

- **Flexbox Διατάξεις**:

Όλες οι ενότητες είναι ευθυγραμμισμένες και χωρίζονται οριζόντια.

- **Στρογγυλεμένες Γωνίες**:

Χρησιμοποιούνται σε διάφορα στοιχεία, όπως κουμπιά και εικονίδια, για αισθητική ομοιομορφία.

Σκοπός του παραπάνω κώδικα είναι να δημιουργεί μια λειτουργική και οπτικά ευχάριστη κεφαλίδα, ιδανική για σύγχρονες ιστοσελίδες με έμφαση στην προσβασιμότητα και την καλή οργάνωση περιεχομένου.

HP ProBook 640 G5

INTEL CORE I5-8365U

SCREEN: 14" 1920X1080 FULL HD

Warranty: 12 Months

Shop Now

JS home.js

```
<div className="home">
    <div className="top_banner">
        <div className="contant">
            <h3>HP ProBook 640 G5</h3>
            <h2>Intel Core i5-8365U</h2>
            <h2>Screen: 14" 1920x1080 Full HD </h2>
            <h3>Warranty: 12 Months</h3>
            <Link to='/shop' className="link">Shop Now</Link>

        </div>
```

Περιγραφή των Τμημάτων

- **Κεντρικό** div **με** className="home"

Περιέχει όλα τα υπόλοιπα στοιχεία και λειτουργεί ως container για την ενότητα.

- **Υποενότητα** top_banner

Περιλαμβάνει την περιοχή με το περιεχόμενο (contant)

- **Περιεχόμενο (**contant**)**

Εμφανίζει λεπτομέρειες για το προϊόν:

- <h3> **και** <h2>**:**

Προσφέρουν διαφορετικά επίπεδα επικεφαλίδων:

- HP ProBook 640 G5: Όνομα προϊόντος.
- Intel Core i5-8365U: Επεξεργαστής.
- Screen: 14" 1920x1080 Full HD: Πληροφορίες για την οθόνη.
- Warranty: 12 Months: Εγγύηση.
- Οι διαφορετικές επικεφαλίδες μπορεί να έχουν διαφορετικό στυλ (π.χ., μέγεθος, χρώμα).

Σύνδεσμος (Link)

Link **(React Router):** Δημιουργεί έναν σύνδεσμο που οδηγεί στη σελίδα /shop.

- className="link"**:** Χρησιμοποιείται για να εφαρμόζεται στυλ στον σύνδεσμο.

Εμφάνιση στην Ιστοσελίδα

- Ο χρήστης βλέπει μια διαφημιστική ενότητα για το προϊόν.
- Μπορεί να διαβάσει βασικές πληροφορίες (όνομα, χαρακτηριστικά, εγγύηση).
- Πατώντας το "Shop Now", οδηγείται στη σελίδα αγορών (/shop).

Χαρακτηριστικά του Κώδικα

- **Καθαρή Δομή:** Χωρίζει το περιεχόμενο σε τμήματα με σαφείς κλάσεις (home, top_banner, contant).
- **Δυναμικότητα με React Router:** Ο σύνδεσμος Link διευκολύνει τη γρήγορη πλοήγηση χωρίς επαναφόρτωση της σελίδας.
- **Ευελιξία:** Εύκολα επεκτάσιμο για να προστεθούν περισσότερες πληροφορίες ή στυλιστικές βελτιώσεις.

home.css

```css
.
{
    margin: 0;
    padding: 0 ;
    font-family: sans-serif;
    box-sizing: border-box;
}

.home
{
    width: 100%;
}

.home .top_banner
{


    max-width: 100%;
    height:500px;
    margin-top: 130px;
    margin-left: 200px;
    background: url(http://localhost:3000/EIKONES/test.png);
    background-size: cover;
    background-repeat: no-repeat;
    padding: 50px 60px;
}
```

Γενικές Ρυθμίσεις

Επαναφορά προεπιλεγμένων περιθωρίων και αποστάσεων.

- Ορισμός της γραμματοσειράς σε sans-serif για ομοιομορφία.
- Χρήση του box-sizing: border-box για καλύτερη διαχείριση διαστάσεων.

Κύρια Ενότητα .home

Ορίζει την ενότητα home να καταλαμβάνει το πλήρες πλάτος της σελίδας.

Τμήμα top_banner

Διαστάσεις: Το ύψος του banner ορίζεται στα 500px, με περιθώριο κορυφής 130px και αριστερό περιθώριο 200px.

- **Φόντο:**Εικόνα με διεύθυνση http://localhost:3000/EIKONES/test.png, η οποία καλύπτει ολόκληρη την ενότητα χωρίς επανάληψη.
- **Εσωτερικό περιθώριο:** Προστίθεται padding 50px (κάθετο) και 60px (οριζόντιο) για εσωτερικό κενό.

Ενότητα Περιεχομένου contant

Τοποθέτηση του περιεχομένου:

- Σχετική θέση (relative) για ευελιξία.
- Μετατόπιση 5% προς τα κάτω και 50% προς τα αριστερά.
- Εφαρμογή padding για να δημιουργηθεί κενό γύρω από το περιεχόμενο.

Στυλ Τίτλων

- **Υποτίτλος (**h3**)**
- **Κείμενο με κεφαλαιοποίηση πρώτου γράμματος κάθε λέξης.**
- Ροζ απόχρωση (#cd1e76) με λεπτή γραμματοσειρά και απόσταση γραμμάτων 1px.

Κεντρικός Τίτλος (h2)
Μετατροπή σε κεφαλαία.

- Μαύρο χρώμα (black), μέγεθος γραμματοσειράς 42px, και απόσταση γραμμάτων 1px.

Σύνδεσμος Δράσης (Call-to-Action Button)

- **Αρχικό Στυλ (**.link**)**
- **Κουμπί με πορτοκαλί φόντο (**#e9660f**) και στρογγυλεμένες γωνίες.**
- Εσωτερικό περιθώριο (padding) 10px (κάθετο) και 30px (οριζόντιο).
- Ομαλή μετάβαση χρωμάτων (transition) με διάρκεια 0.5s.

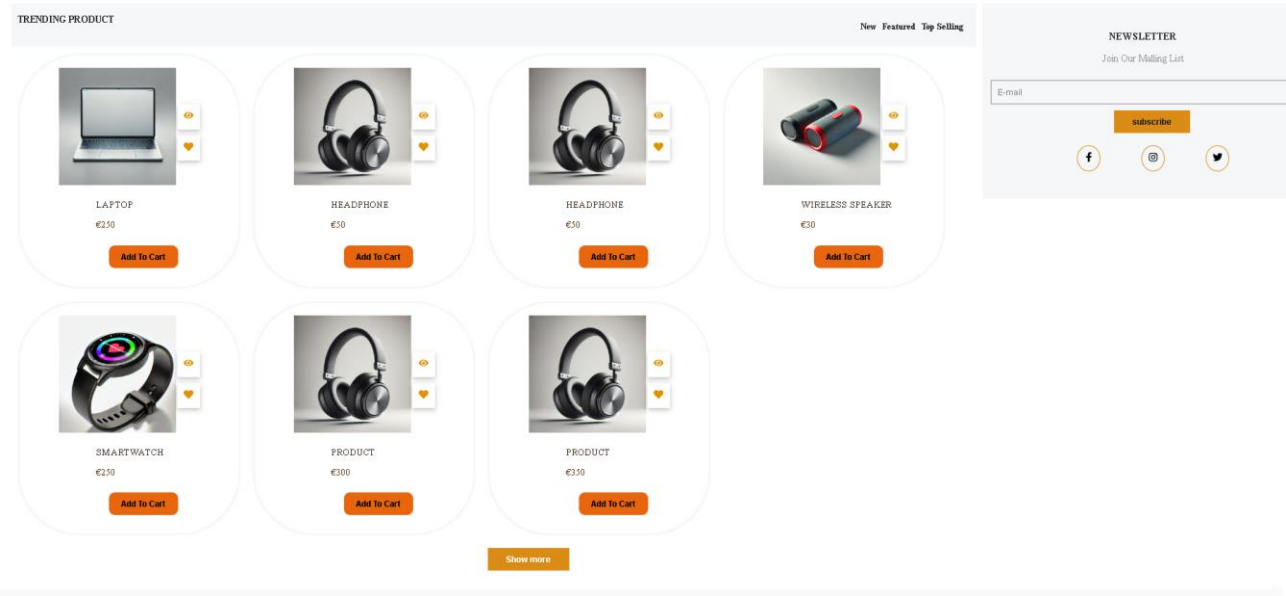
Εφέ Hover

Αλλαγή χρώματος κουμπιού και κειμένου σε σκούρο πορτοκαλί (#df9903) κατά την αιώρηση.

```css
.home .top_banner .contant
{
    width: 100%;
    position: relative;
    top: 5%;
    left: 50%;
    padding: 10px 50px;

}


.home .top_banner .contant h3{
    text-transform: capitalize;
    color: #cd1e76;
    font-weight: 300;
    letter-spacing: 1px;
}

.home .top_banner .contant h2
{
    text-transform: uppercase;
    color :black;
    letter-spacing: 1px;
    font-size: 42px;
    margin-top: 10px;
}
```

```css
.home .top_banner .contant .link
{
    margin-top: 20px;
    padding: 10px 30px;
    background: #e9660f;
    color: #232323;
    text-decoration: none;
    border-radius: 5px;
    transition: 0.5s;
}

.home .top_banner .contant .link:hover
{
    color: #df9903;
    background:  #df9903;

}
```

Ο παραπάνω CSS κώδικας αφορά τη μορφοποίηση ενότητας της σελίδας, που περιγράφεται ως home, με ιδιαίτερη έμφαση στο στοιχείο του κορυφαίου banner.



Φωτογραφία 2 (Trending Products)

Αυτή η ενότητα παρουσιάζει τα **Trending Products**.

Περιλαμβάνει:

- Πολλαπλές κάρτες προϊόντων (π.χ., Laptop, Headphone, Smartwatch).
- Κουμπιά ταξινόμησης “New , Featured,Top Selling”
- Κουμπί "Add to Cart" για κάθε προϊόν.
- Εικονίδια (Eye, Heart) για προβολή και προσθήκη στα αγαπημένα.

Στα δεξιά υπάρχει φόρμα εγγραφής στο newsletter με πεδίο εισαγωγής email και κουμπί "Subscribe".

Ανάλυση από τον κώδικα:

JS home.js

```
<div className="trending">
    <div className="container">
        <div className="left_box">
            <div className="header">

                <div className="heading">
                    <h2 onClick={() => allTrendingProduct ()}>trending product</h2>
                </div>

                <div className="cate">
                    <h3 onClick={() => filtercate ('new')}>New</h3>
                    <h3 onClick={() => filtercate ('featured')}>Featured</h3>
                    <h3 onClick={() => filtercate ('top')}>top selling</h3>

                </div>
            </div>
```

JS home.js

```
<div className="products">
    <div className="container">
        {
            trendingProduct.map((curElm) =>
            {
                return(
                    <>
                        <div className="box">
                            <div className="img_box">
                                <img src={curElm.image} alt=""></img>
                                <div className="icon">
                                    <div className="icon_box">
                                    <FaEye />


                                    </div>
                                    <div className="icon_box">
                                    <FaHeart />


                                    </div>
                                </div>
                            </div>
                            <div className="info">
                                <h3>{curElm.Name}</h3>
                                <p>€{curElm.price}</p>
                                <button className="btn" onClick={ () => addtocart (curElm)}>Add to cart</button>
                            </div>
                        </div>
                    </>
                )
            })
```

Βασικές Λειτουργίες

- allTrendingProduct**:** Φαίνεται να φορτώνει όλα τα trending προϊόντα.
- filtercate**:** Φιλτράρει τα προϊόντα με βάση την κατηγορία (π.χ., "New", "Featured").
- addtocart**:** Προσθέτει το επιλεγμένο προϊόν στο καλάθι.

Κύρια Δομή

Η ενότητα περιλαμβάνει τα εξής κύρια τμήματα:

- trending **(Outer Container):** Το βασικό που περιέχει όλη την ενότητα.
- container**:** Ένα εσωτερικό τμήμα για τη διάταξη περιεχομένου.

Ενότητα header

Περιλαμβάνει τον τίτλο και τις κατηγορίες φίλτρων:

- heading**:** Ένας τίτλος "Trending Product", ο οποίος όταν κλικάρεται, καλεί τη συνάρτηση allTrendingProduct() για να εμφανίσει όλα τα trending προϊόντα.

- cate**:** Τρεις κατηγορίες προϊόντων ("New", "Featured", "Top Selling"), με δυνατότητα φιλτραρίσματος μέσω της συνάρτησης filtercate() που δέχεται ως όρισμα την αντίστοιχη κατηγορία.

JS home.js

```
const filtercate = (x) =>
{
    const filterproduct = Homeproduct.filter((curElm) =>
    {
        return curElm.type === x
    })
    setTrendingProduct(filterproduct)
}
```

Παράμετρος x**:**

- Η συνάρτηση λαμβάνει ως όρισμα την τιμή x, η οποία αντιπροσωπεύει τον τύπο των προϊόντων που θέλουμε να φιλτράρουμε (π.χ., "new", "featured", "top").

- Homeproduct.filter**:**

- Η μέθοδος .filter() εφαρμόζει φίλτρο στον πίνακα Homeproduct.

- Για κάθε στοιχείο του πίνακα (curElm):

    - Ελέγχει αν η ιδιότητα type του στοιχείου είναι ίση με το όρισμα x.

    - Επιστρέφει μόνο τα στοιχεία που πληρούν την προϋπόθεση (δηλαδή έχουν type === x).

- **Αποτέλεσμα (**filterproduct**):**

    - Το αποτέλεσμα του φίλτρου αποθηκεύεται στη μεταβλητή filterproduct, η οποία περιέχει μόνο τα προϊόντα που ανήκουν στην κατηγορία x.

      setTrendingProduct**:**

    - Χρησιμοποιείται για να ενημερώσει την κατάσταση TrendingProduct με τα φιλτραρισμένα προϊόντα.

    - Αυτή η κατάσταση χρησιμοποιείται στον κώδικα JSX για να αποδώσει μόνο τα σχετικά προϊόντα στην ενότητα "Trending Products".

Ενότητα products

Η ενότητα αυτή δημιουργεί δυναμικά τις κάρτες των προϊόντων:

- **Δυναμική απόδοση προϊόντων:**

    - Χρησιμοποιείται η μέθοδος map() για να περάσει το κάθε στοιχείο από τον πίνακα trendingProduct και να δημιουργήσει μία κάρτα προϊόντος για κάθε αντικείμενο.

```
const [trendingProduct, setTrendingProduct] = useState(Homeproduct)
//filter of tranding product
```

    - curElm**:** Αναπαριστά το τρέχον στοιχείο του πίνακα και χρησιμοποιείται για να αποκτήσει πρόσβαση σε δεδομένα όπως εικόνα, όνομα και τιμή.

Δομή Κάρτας Προϊόντος

Η κάθε κάρτα περιλαμβάνει:

1. img_box**:**

    - Εμφανίζει την εικόνα του προϊόντος από το curElm.image.

    - **Εικονίδια:**

      Χρησιμοποιεί τα εικονίδια FaEye (προβολή προϊόντος) και FaHeart (αγαπημένα), τα οποία είναι μέρος της βιβλιοθήκης react-icons.

info**:**

- Εμφανίζει το όνομα του προϊόντος (curElm.Name).

- Εμφανίζει την τιμή του προϊόντος (curElm.price).

- **Κουμπί "Add to Cart":**

    - Καλεί τη συνάρτηση addtocart(curElm) για να προσθέσει το συγκεκριμένο προϊόν στο καλάθι.

App.js

```
const addtocart =(product) =>
{
  const exist = cart.find((x) =>{
    return  x.id === product.id
  })
  if(exist)
  {
    alert('tis product is already added in cart ')
  }
  else
  {
    setCart([...cart, {...product, qty:1}])
    alert('Added to cart')
  }

}
```

Η συνάρτηση addtocart είναι ένα prop που περνάται σε διάφορες σελίδες ή components, όπως η Home και η Shop, για να προσθέτει προϊόντα στο καλάθι αγορών

Είναι υπεύθυνη για την προσθήκη ενός προϊόντος στο καλάθι αγορών.

- Παίρνει ως παράμετρο το προϊόν που ο χρήστης θέλει να προσθέσει και ενημερώνει την κατάσταση (state) του καλαθιού.

2. **Διασύνδεση:**

rout.js

```
<Route path="shop" element={<Shop shop={shop}  Filter={Filter} allcatefilter={allcatefilter} addtocart={addtocart}/>}/>
```

Ενσωματώνεται ως prop (addtocart) στις παρακάτω διαδρομές:

- Home**:** Παρέχεται στη βασική σελίδα, όπου τα προϊόντα εμφανίζονται σε μορφή λίστας με δυνατότητα προσθήκης στο καλάθι.
- Shop**:** Παρέχεται στο κατάστημα, για να προστίθενται προϊόντα από εκεί.
- Καλείται μέσα σε κουμπί "Add to cart" σε κάθε κάρτα προϊόντος, όπως φάνηκε στον προηγούμενο JSX κώδικα.

3. **Υλοποίηση:**

- Ενημερώνει το state του καλαθιού με το νέο προϊόν.
- Ελέγχει εάν το προϊόν υπάρχει ήδη στο καλάθι και αυξάνει την ποσότητα, εάν χρειάζεται.

Εκτέλεση στη Σελίδα

Στον JSX κώδικα που παρουσιάστηκε παραπάνω:

- Η addtocart καλείται όταν ο χρήστης πατά το κουμπί **"Add to Cart"** σε μία κάρτα προϊόντος:

- Το προϊόν (curElm) περνά στη συνάρτηση addtocart για επεξεργασία.

Η συνάρτηση addtocart αποτελεί μία από τις βασικές λειτουργίες για τη διαχείριση του καλαθιού αγορών (cart) στην εφαρμογή. Ενσωματώνεται σε πολλές διαδρομές (Routes), όπως η αρχική σελίδα (Home) και το κατάστημα (Shop). Παρακάτω εξηγώ τη δομή και τον ρόλο της:

Λειτουργία addtocart

1. **Ρόλος:**

    - Η addtocart είναι υπεύθυνη για την προσθήκη ενός προϊόντος στο καλάθι αγορών.

    - Παίρνει ως παράμετρο το προϊόν που ο χρήστης θέλει να προσθέσει και ενημερώνει την κατάσταση (state) του καλαθιού.

2. **Διασύνδεση:**

    - Ενσωματώνεται ως prop (addtocart) στις παρακάτω διαδρομές:

    - Home**:** Παρέχεται στη βασική σελίδα, όπου τα προϊόντα εμφανίζονται σε μορφή λίστας με δυνατότητα προσθήκης στο καλάθι.

    - Shop**:** Παρέχεται στο κατάστημα, για να προστίθενται προϊόντα από εκεί.

    - Καλείται μέσα σε κουμπί "Add to cart" σε κάθε κάρτα προϊόντος, όπως φάνηκε στον προηγούμενο JSX κώδικα.

3. **Υλοποίηση:**

    - Ενημερώνει το state του καλαθιού με το νέο προϊόν.

    - Ελέγχει εάν το προϊόν υπάρχει ήδη στο καλάθι και αυξάνει την ποσότητα, εάν χρειάζεται.

Σύνδεση με το State του Καλαθιού

Από τη διαδρομή /cart:

- Η addtocart ενημερώνει το state του καλαθιού μέσω της setCart, οπου είναι η συνάρτηση αλλαγής του state (διαχειρίζεται το καλάθι).

App.js

```
const App = () => {
  //add to cart
  const [cart, setCart] = useState([])
```

Η addtocart παίζει κρίσιμο ρόλο για την αλληλεπίδραση του χρήστη με το καλάθι αγορών, διασφαλίζοντας τη σωστή προσθήκη προϊόντων και την ενημέρωση του state σε όλες τις σχετικές διαδρομές.

Κουμπί "Show More"

- Το κουμπί "Show More" εμφανίζεται στο τέλος της λίστας προϊόντων.
- Χρησιμοποιείται για τη φόρτωση περισσότερων προϊόντων ή την αλλαγή της προβολής τους.

NEWSLETTER

home.js

```
<div className="right_box">
    <div className="right_container">
        <div className="newsletter">
            <div className="head">
                <h3>newsletter</h3>
            </div>
            <div className="form">
                <p>join our malling list</p>
                <input type="email" placeholder="E-mail" autoComplete="off"></input>
                <button>subscribe</button>
                <div className="icon_box">
                    <div className="icon">
                <FaFacebookF />
                </div>
                <div className="icon">
                    <FaInstagram/>
                </div>
                <div className="icon">
                    <FaTwitter/>
                </div>
                </div>

                </div>
            </div>

        </div>
    </div>


</div>
```

Ο κώδικας σχετίζεται
με τη δημιουργία ενός τμήματος σελίδας που επιτρέπει στους επισκέπτες να εγγραφούν σε μια λίστα mailing και περιέχει εικονίδια κοινωνικών δικτύων.

Σκοπός Κώδικα

Ο κώδικας αυτός δημιουργεί ένα τμήμα ιστοσελίδας για την εγγραφή χρηστών σε μια λίστα mailing. Παρέχει επίσης τη δυνατότητα άμεσης σύνδεσης με κοινωνικά δίκτυα μέσω των εικονιδίων.

Λειτουργικότητα Κώδικα

- **Εγγραφή Χρηστών:** Το πεδίο κειμένου επιτρέπει την εισαγωγή του email, ενώ το κουμπί "subscribe" ενεργοποιεί τη διαδικασία εγγραφής.
- **Προβολή Κοινωνικών Δικτύων:** Τα εικονίδια επιτρέπουν στους χρήστες να μεταβούν στους αντίστοιχους λογαριασμούς κοινωνικών μέσων.

Περιγραφή Τμημάτων

1. **Επικεφαλίδα (**<h3>**):** Εμφανίζει τον τίτλο του τμήματος "Newsletter".
2. **Μήνυμα (<p>): Ενημερώνει τους χρήστες για το σκοπό της φόρμας.**
3. **Φόρμα Εισαγωγής:**

   Το πεδίο email (<input>):

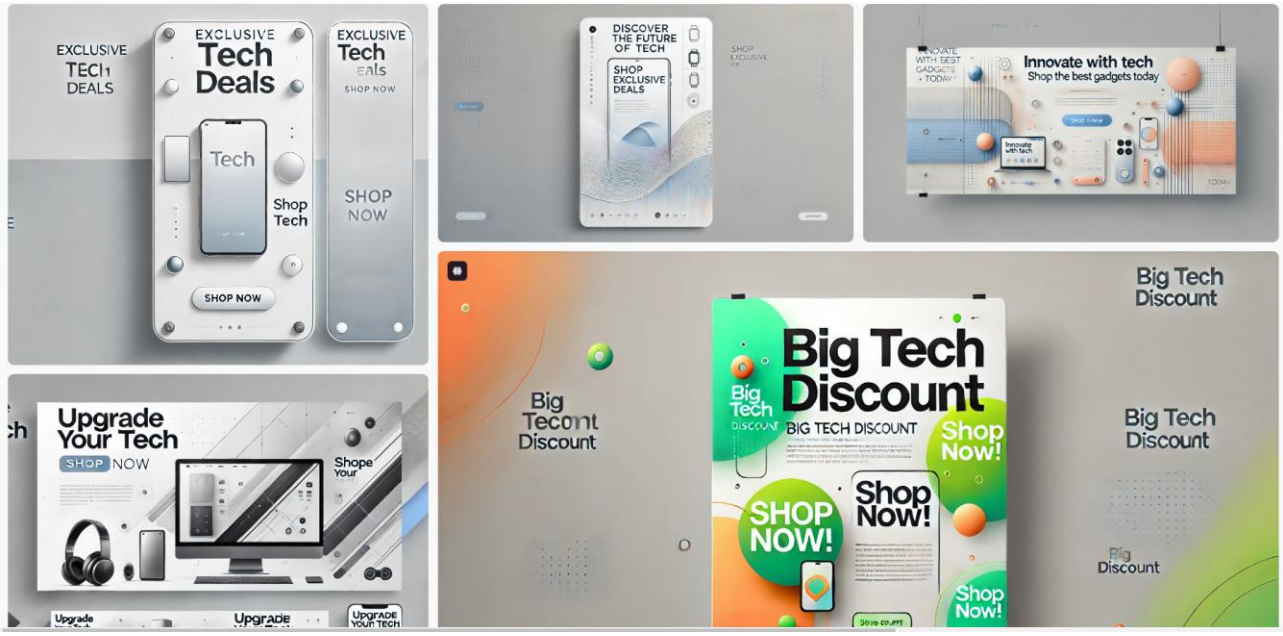
   - Υποστηρίζει μόνο email μέσω type="email".
   - Το placeholder καθοδηγεί τον χρήστη με το προτεινόμενο κείμενο.

   Το κουμπί εγγραφής (<button>):

   - Παρέχει μηχανισμό για την υποβολή του email.

4. **Εικονίδια (**FontAwesome Icons**):**
   - Προσθέτουν διαδραστικά στοιχεία για κοινωνική αλληλεπίδραση.

Το <input> ορίζει ένα πεδίο κειμένου για την εισαγωγή email, ενώ το placeholder εμφανίζει το κείμενο "E-mail" μέχρι να αρχίσει η πληκτρολόγηση. Το <button> δημιουργεί το κουμπί "subscribe".



Φωτογραφία 3 (Banner για Προσφορές)

Περιγραφή:

- Περιέχει πολλαπλά banners για προώθηση προϊόντων.
- Τα banners είναι διαφορετικών διαστάσεων και εμφανίζονται με εικόνες προϊόντων.
- Τα στοιχεία είναι στατικά (εικόνες banner) αλλά τοποθετούνται δυναμικά μέσω CSS για responsive διάταξη.

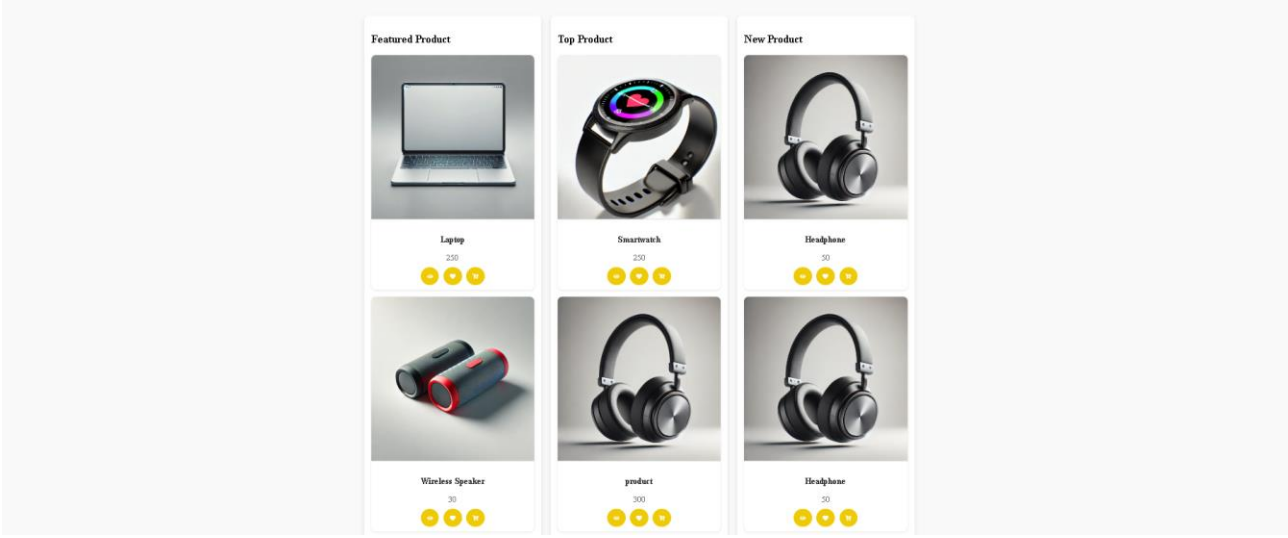
Ακολουθεί ο CSS κώδικας που χρησιμοποιήθηκε για την συγκεκριμένη διάταξη.

```css
.banners {
    padding: 20px;
    box-sizing: border-box;
    width: 100%;
    background-color: #f0f0f0; /* Ανοιχτόχρωμο φόντο για αντίθεση */
}

.banners .container {
    display: grid;
    grid-template-columns: 1fr 2fr; /* Δύο στήλες: 1/3 και 2/3 */
    gap: 15px; /* Κενό ανάμεσα στα στοιχεία */
}

.banners .left_box {
    display: grid;
    grid-template-rows: 1fr 1fr; /* Δύο γραμμές ίδιου μεγέθους */
    gap: 15px;
}

.banners .left_box .box {
    overflow: hidden;
    border-radius: 10px; /* Στρογγυλεμένες γωνίες για αισθητική */
}

.banners .left_box .box img {
    width: 100%;
    height: 100%;
    object-fit: cover; /* Διατήρηση αναλογιών χωρίς παραμόρφωση */
}
```

```css
.banners .right_box {
    display: grid;
    grid-template-rows: auto auto; /* Ευελιξία ύψους */
    gap: 15px;
}

.banners .right_box .top {
    display: grid;
    grid-template-columns: 1fr 1fr; /* Δύο ίσα banner στη σειρά */
    gap: 15px;
}

.banners .right_box .top img,
.banners .right_box .bottom img {
    width: 100%;
    height: 100%;
    object-fit: cover; /* Εξασφαλίζει σωστή τοποθέτηση */
    border-radius: 10px; /* Στρογγυλεμένες γωνίες */
}

.banners .right_box .bottom {
    overflow: hidden;
    border-radius: 10px;
}
```

```css
/* Responsive Προσαρμογές */
@media (max-width: 768px) {
    .banners .container {
        grid-template-columns: 1fr; /* Μία στήλη για μικρές οθόνες */
    }

    .banners .right_box .top {
        grid-template-columns: 1fr; /* Κάθετα τα banners */
    }
}
```

- Τα banners που εμφανίζονται οριζόντια σε μεγαλύτερες οθόνες (desktop) αναδιοργανώνονται σε **κάθετη στοίχιση** (vertical) στις οθόνες με πλάτος έως 768px.
- Η χρήση του grid-template-columns αξιοποιεί το CSS Grid για ευέλικτη και responsive σχεδίαση.
- Ο κώδικας διασφαλίζει ότι η διάταξη των banners προσαρμόζεται για μικρότερες συσκευές.



Φωτογραφία 4 (Κατηγοριοποιημένα Προϊόντα)

Περιγραφή:

Τα προϊόντα παρουσιάζονται με βάση την κατηγορία:

- **Featured Products**
- **Top Products**
- **New Products**

Κάθε κατηγορία περιέχει κάρτες προϊόντων, παρόμοιες με τα Trending Products.

Product Categories:

- Δημιουργούνται από τις καταστάσεις featuredProduct, topProduct, και newProduct μέσω του productcategory() που καλείται στο useEffect:
- Οι κάρτες παράγονται μέσω map() όπως στα Trending Products.

Κουμπιά και Εικονίδια:

Τα εικονίδια (Eye, Heart, CartArrowDown) είναι διαδραστικά και παρέχουν λειτουργίες όπως προσθήκη στο καλάθι (addtocart()).

Ανάλυση του Κώδικα

```
<div className="product_type">
    <div className="container">
        <div className="box">
         <div className="'header">
            <h2>Featured Product</h2>
         </div>
         {
           featuredProduct.map((curElm) => {
               return(
                   <>
                       <div className="productbox">
                           <div className="img-box">
                               <img src={curElm.image} alt=""></img>
                           </div>
                           <div className="detail">
                               <h3>{curElm.Name}</h3>
                               <p>{curElm.price}</p>
                               <div className="icon">
                                   <button><FaEye /></button>
                                   <button><FaHeart /></button>
                                   <button onClick={() => addtocart (curElm)}><FaCartArrowDown /></button>
                               </div>
                           </div>
                       </div>

                   </>
               )
           })
         }
```

Ο κώδικας ανήκει στην ενότητα product_type και χρησιμοποιείται για την εμφάνιση κατηγοριών προϊόντων, όπως **Featured Product**, **Top Product**, και **New Product**.

Δομή

Πλαίσιο (box)

- Κάθε κατηγορία προϊόντων περιέχεται σε ένα divμε κλάση box.
- Η επικεφαλίδα της κατηγορίας εμφανίζεται μέσα σε ένα divμε κλάση 'header, με τίτλο όπως **Featured Product**.

Χρήση του map ()

- Τα προϊόντα εμφανίζονται δυναμικά χρησιμοποιώντας το featuredProduct.map(), που διατρέχει έναν πίνακα featuredProduct και αποδίδει κάθε προϊόν.

Εμφάνιση Πληροφοριών Προϊόντος

Κάθε προϊόν εμφανίζεται μέσα σε ένα divμε κλάση productbox, που περιλαμβάνει:

- Εικόνα προϊόντος: Μέσα σε img-box.
- Λεπτομέρειες προϊόντος: Όπως το όνομα (curElm.Name) και η τιμή (curElm.price).
- Εικονίδια: Κουμπιά για ενέργειες (προβολή, αγαπημένα, προσθήκη στο καλάθι).

Σύνδεση με τη Λειτουργία addtocart

- Το κουμπί "Add to Cart" καλεί τη λειτουργία addtocart(curElm) για να προσθέσει το προϊόν στο καλάθι.

Σύνδεση με τα Άλλα Αρχεία JS

Αρχείο home.js

- Ο πίνακας featuredProduct και η μέθοδος map() ορίζονται και χρησιμοποιούνται στο home.js. Πιο συγκεκριμένα:
    - Ο featuredProduct γεμίζει μέσω της λειτουργίας productcategory() στο useEffect.
    - Το productcategory() φιλτράρει τα προϊόντα από το αρχείο Homeproduct με βάση τον τύπο τους (type === 'featured').

Άρα, τα δεδομένα για τα προϊόντα προέρχονται από το Homeproduct μέσω του home.js.

Αρχείο App.js

- Η λειτουργία addtocart ορίζεται στο App.js και περνιέται ως prop στα home.js και κατ' επέκταση στο τμήμα κώδικα που αναλύουμε.
- Ορίσαμε ότι όταν καλείται η addtocart, προσθέτει το προϊόν στον πίνακα cart που διαχειρίζεται το App.js.

Αρχείο home_product.js

- Το home_product.js είναι η βασική πηγή δεδομένων για τα προϊόντα. Εδώ αποθηκεύονται όλα τα προϊόντα με τις λεπτομέρειες τους, όπως id, Name, price, και type.
- Το home.js χρησιμοποιεί τα δεδομένα του home_product.js για να φιλτράρει και να αποδώσει τις κατηγορίες (Featured, Top, New).

ΛειτουργΙκότητα

- Δυναμική Φόρτωση Προϊόντων
- Τα προϊόντα για κάθε κατηγορία (π.χ., Featured Product) φορτώνονται και εμφανίζονται δυναμικά μέσω του πίνακα featuredProduct, ο οποίος γεμίζει στο home.js.

Σύνδεση με Καλάθι

- Το κουμπί "Add to Cart" επιτρέπει στους χρήστες να προσθέτουν προϊόντα στο καλάθι. Η λειτουργία αυτή διαχειρίζεται το App.js μέσω της addtocart.

```
<div className="box">
 <div className="'header">
    <h2>Top Product</h2>
  </div>
  {
    topProduct.map((curElm) => {
        return(
            <>
                <div className="productbox">
                    <div className="img-box">
                        <img src={curElm.image} alt=""></img>
                    </div>
                    <div className="detail">
                        <h3>{curElm.Name}</h3>
                        <p>{curElm.price}</p>
                        <div className="icon">
                            <button><FaEye /></button>
                            <button><FaHeart /></button>
                            <button onClick={() => addtocart (curElm)}><FaCartArrowDown /></button>
                        </div>
                    </div>
                </div>

            </>
        )
    })
  }
```

```
<div className="box">
 <div className="'header">
    <h2>New Product</h2>
  </div>
  {
    newProduct.map((curElm) => {
        return(
            <>
                <div className="productbox">
                    <div className="img-box">
                        <img src={curElm.image} alt=""></img>
                    </div>
                    <div className="detail">
                        <h3>{curElm.Name}</h3>
                        <p>{curElm.price}</p>
                        <div className="icon">
                            <button><FaEye /></button>
                            <button><FaHeart /></button>
                            <button onClick={() => addtocart (curElm)}><FaCartArrowDown /></button>
                        </div>
                    </div>
                </div>

            </>
        )
    })
  }


</div>
```

Ανάλυση

Δομή για Top Product:

Δημιουργείται ένα div με κλάση box που περιλαμβάνει:

- Τίτλο της κατηγορίας μέσα στο div με κλάση 'header.
- Δυναμική απόδοση προϊόντων μέσω του topProduct.map().

Δομή για New Product:

Αντίστοιχη δομή με την κατηγορία **Top Product**, αλλά χρησιμοποιεί τον πίνακα newProduct για την απόδοση προϊόντων.

Κοινά Σημεία:

Η μέθοδος map() για την απόδοση προϊόντων είναι ίδια με αυτή της κατηγορίας

Featured Product, επομένως δεν απαιτείται επανάληψη της ανάλυσης.

Κάθε προϊόν έχει παρόμοια δομή (εικόνα, όνομα, τιμή, εικονίδια ενεργειών).

Επισήμανση

Αυτές οι κατηγορίες (Top Product και New Product) λειτουργούν με πανομοιότυπο τρόπο με την κατηγορία **Featured Product**, αλλά χρησιμοποιούν διαφορετικούς πίνακες δεδομένων (topProduct και newProduct) που δημιουργούνται στο home.js.

Εφόσον η διαδικασία απόδοσης προϊόντων είναι ακριβώς η ίδια, δεν χρειάζεται περαιτέρω ανάλυση.

Φωτογραφία 5-Υποσέλιδο

JS footer.js

```
import React from "react"
import './footer.css'
import { FaPiggyBank,FaShippingFast,FaHeadphones,FaWallet } from "react-icons/fa";


const Footer = () => {
    return (
        <>
        <div className="footer">
            <div className="container">
                <div className="left-box">
                    <div className="box">
                        <div className="icon_box">
                        <FaPiggyBank />
                        </div>
                        <div className="detail">
                            <h3>Great saving</h3>
                        </div>
                    </div>
                    <div className="box">
                        <div className="icon_box">
                        <FaShippingFast />
                        </div>
                        <div className="detail">
                            <h3>24/7 support</h3>

                        </div>
                    </div>
```

```
                    <div className="box">
                        <div className="icon_box">
                        <FaWallet />
                        </div>
                        <div className="detail">
                            <h3>Money back</h3>
                        </div>
                    </div>
                </div>
            <div className="right_box">
                <div className="header">
                    <img src="EIKONES/newlogo.jpeg " alt="logo"></img>
                    <p> Ανυπομονούμε να σας εξυπηρετήσουμε ξανά σύντομα. Μη διστάσετε να επικοινωνήσετε μαζί μας για οποιαδήποτε απορία ή ανάγκη.</p>
                </div>
                <div className="bottom">
                    <div className="box">
                        <h3>Yout Account</h3>
                        <ul>
                            <li>About us</li>
                            <li>Account</li>
                            <li>Payment</li>
                            <li>sales</li>
                        </ul>
                    </div>
                    <div className="box">
                        <h3>Yout Account</h3>
                        <ul>
                            <li>Delivery</li>
                            <li>Track order</li>
                            <li>New product</li>
                            <li>Old product</li>
                        </ul>
                    </div>
```

```
                    <div className="box">
                        <h3>contact us</h3>
                        <ul>
                            <li></li>
                            <li>+(030) 6999999999</li>
                            <li>info@gata.gr</li>

                        </ul>
                    </div>
                </div>
            </div>
            </div>
        </div>
        </>
    )
}

export default Footer;
```

Ανάλυση του Footer Component

Δομή του Footer

Κύριο πλαίσιο (footer)

- Το div με κλάση footer περιέχει όλο το περιεχόμενο του υποσέλιδου.
- Εντός του, υπάρχει ένα div με κλάση container, το οποίο χωρίζεται σε:
- **Αριστερό τμήμα (left-box)**
- **Δεξί τμήμα (right_box)**

Αριστερό Τμήμα (left-box)

Το left-box εμφανίζει 3 βασικές πληροφορίες/ενότητες με τη βοήθεια εικονιδίων από τη βιβλιοθήκη react-icons.

Περιεχόμενο:

- **Great Saving:**
  - Περιέχει το εικονίδιο FaPiggyBank που υποδεικνύει οικονομική εξοικονόμηση.
  - Το μήνυμα τονίζει την αξία των προϊόντων για τους πελάτες.

24/7 Support:

  - Περιέχει το εικονίδιο FaShippingFast, τονίζοντας την υποστήριξη στους πελάτες όλο το 24ωρο.

Money Back:

  - Περιέχει το εικονίδιο FaWallet, προσφέροντας εγγύηση επιστροφής χρημάτων.

Σημαντικά Σημεία:

1. **Οπτική Εμφάνιση:** Κάθε τμήμα αποτελείται από ένα εικονίδιο και περιγραφή (τίτλος και υπότιτλος).
2. **ΛειΤουργικότητα:** Οι πληροφορίες αυτές λειτουργούν περισσότερο ως marketing messages για να χτίσουν εμπιστοσύνη στον πελάτη.

Δεξί Τμήμα (right_box)

Header:

Περιλαμβάνει:Το λογότυπο (με την εικόνα από το path EIKONES/newlogo.jpeg).Ένα κείμενο που εκφράζει ευγνωμοσύνη και ενθαρρύνει την επικοινωνία με την εταιρεία.

Bottom Section:

Περιέχει τρία διαφορετικά div με κλάση box για επιπλέον πληροφορίες.

Περιεχόμενο του Bottom Section

Your Account:

- Περιέχει links για σελίδες όπως:

    1. About Us
    2. Account
    3. Payment
    4. Sales

- Αυτά τα links βοηθούν τον χρήστη να πλοηγηθεί εύκολα σε βασικές λειτουργίες.

Your Account (Delivery & Orders):

- Παρόμοια με την προηγούμενη ενότητα, αλλά εστιάζει στη διαχείριση παραγγελιών:

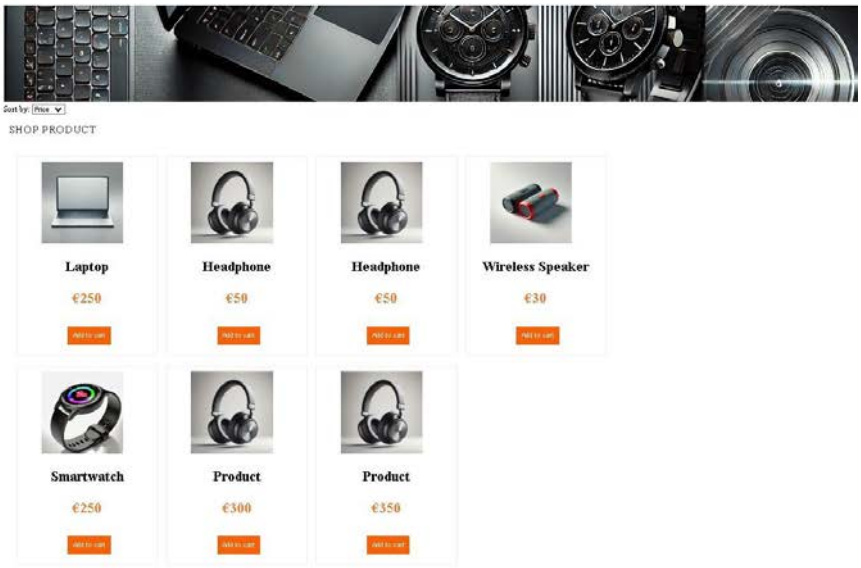
    5. Delivery
    6. Track Order
    7. New Product
    8. Old Product

Contact Us:

- Παρέχει πληροφορίες επικοινωνίας:

    9. Τηλέφωνο: "+(030) 6999999999".
    10. Email: "info@gata.gr".

Γενική Εμφάνιση



Τίτλος Σελίδας:

- Εμφανίζεται η επικεφαλίδα **# SHOP**, η οποία υποδεικνύει ότι πρόκειται για τη σελίδα του καταστήματος.

Breadcrumb Navigation:

- Η φράση **Home . shop** παρέχει breadcrumbs, που βοηθούν τους χρήστες να κατανοήσουν τη θέση τους στον ιστότοπο.

Πλευρική Στήλη (Sidebar)

Στα αριστερά, υπάρχει μια πλευρική στήλη που περιλαμβάνει τα εξής:

Κατηγορίες:

- Ο τίτλος **ALL CATEGORIES** δίνει έμφαση στην οργάνωση των προϊόντων σε κατηγορίες.
- Διαθέσιμες κατηγορίες:
    - **ALL:** Όλα τα προϊόντα.
    - **HEADPHONE: Ακουστικά.**
    - **LAPTOP: Φορητοί Υπολογιστές.**
    - **WATCH: Ρολόγια.**
    - **SPEAKER: Ηχεία.**

Προωθητική Εικόνα:

- Υπάρχει μια εικόνα που προωθεί το καλάθι αγορών με την ένδειξη **Your Cart Awaits!**.

Κεντρική Περιοχή Προβολής Προϊόντων

- Στα δεξιά, η κύρια περιοχή περιέχει τα προϊόντα προς πώληση:

Μπάρα Φιλτραρίσματος:

- Υπάρχει επιλογή φιλτραρίσματος προϊόντων βάσει **τιμής** (Sort by: Price). Αυτό επιτρέπει στους χρήστες να βλέπουν τα προϊόντα με βάση το κόστος, από φθηνότερο προς ακριβότερο ή αντίστροφα.

Λίστα Προϊόντων:

- Τα προϊόντα εμφανίζονται σε κάρτες, που περιλαμβάνουν τα εξής στοιχεία:
    - **Εικόνα Προϊόντος:** Προβάλλει το προϊόν με ευκρινή τρόπο.
    - **Όνομα Προϊόντος:** Π.χ., Laptop, Headphone, Wireless Speaker, Smartwatch.
    - **Τιμή Προϊόντος:** Π.χ., €250, €50, €30, €250.
    - **Κουμπί Add to Cart:** Δίνει τη δυνατότητα στον χρήστη να προσθέσει το προϊόν στο καλάθι του.

Διάταξη Προϊόντων:

- Τα προϊόντα είναι οργανωμένα σε μορφή grid (δικτυωτή μορφή), με συμμετρική εμφάνιση για αισθητική ευκολία.

Λειτουργικότητα

Πλευρική Στήλη (Sidebar):

- Η κατηγοριοποίηση των προϊόντων επιτρέπει εύκολη πλοήγηση.
- Η δυνατότητα επιλογής κατηγορίας βοηθά τον χρήστη να δει μόνο τα προϊόντα που τον ενδιαφέρουν.

Κεντρική Περιοχή Προϊόντων:

- Παρέχει οπτική και λειτουργική πρόσβαση στα προϊόντα.Η επιλογή φιλτραρίσματος βάσει τιμής προσθέτει ευχρηστία, κάνοντας την εμπειρία πιο φιλική προς τον χρήστη.

Κουμπί "Add to Cart":

- Το κουμπί λειτουργεί ως βασική ενέργεια για το καλάθι, ενθαρρύνοντας τον χρήστη να ξεκινήσει τη διαδικασία αγοράς.

Ανάλυση Σημαντικών Σημείων στον Κώδικα

Το Shop component είναι υπεύθυνο για την εμφάνιση της σελίδας του καταστήματος. Παρέχει λειτουργίες για την προβολή, την ταξινόμηση και τη φιλτραρισμένη απόδοση προϊόντων, καθώς και τη διαχείριση λεπτομερειών προϊόντος.

Ακολουθεί αναλυτική επισήμανση των σημαντικών σημείων:

```jsx
import React, { useState } from "react"
import './shop.css'
import { FaHeart,FaEye } from "react-icons/fa";
import { FaSkullCrossbones } from "react-icons/fa6";


const Shop = ({shop, Filter, allcatefilter,addtocart}) => {
    //toggle product detail
    const [showDetail, setShowDetail]= useState(false)
    //detail page data
    const [detail, setDetail] = useState([])
    //sort of prices
    const [sortOption, setSortOption] = useState('price');
    //detail page

    const detailpage =(product) =>
    {
        const detaildata = ([{product}])
        const productdetail = detaildata [0] ['product']
        //console.log(productdetail)
        setDetail(productdetail)
        setShowDetail(true)
    }
   // console.log(detail)
    const closedetail  = () =>
    {
        setShowDetail(false)
    }
```

```jsx

    const sortedShop = [...shop].sort((a, b) => {
        if (sortOption === 'price') {
            return a.price - b.price; // Ταξινόμηση με βάση την τιμή
        } else if (sortOption === 'name') {
            return a.Name.localeCompare(b.Name); // Ταξινόμηση με βάση το όνομα
        }
        return 0;
    });
    //setSortedShop(sortedProducts); // Ενημέρωση των ταξινομημένων προϊόντων
    //}, [sortOption, shop]); // Όταν αλλάζει το `sortOption` ή το `shop` τα προϊόντα ανανεώνονται

    return (
        <>
        {
            showDetail ?
            <>
                <div className="product_detail">
                    <button className="close_btn" onClick={closedetail}><FaSkullCrossbones /></button>
                    <div className="container">
                        <div className="img_box">
                            <img src={detail.image} alt=''></img>
                        </div>
                        <div className="info">
                            <h4># {detail.cat}</h4>
                            <h2>{detail.Name}</h2>
                            <p>{detail.Detail}</p>
                            <h3>€ {detail.price}</h3>
                            <button onClick={() => addtocart(detail)}>Add to Cart </button>
                        </div>
                    </div>

                </div>
```

```
        :null
    }



    <div className="shop">
        <h2># shop</h2>
        <p>Home . shop</p>
        <div className="container">
            <div className="left_box">
                <div className="category">
                    <div className="header">
                        <h2>all categories</h2>
                    </div>
                    <div className="box">
                        <ul>
                            <li onClick={() => allcatefilter('All')}># All</li>
                            <li onClick={() => Filter('headphone')}># Headphone</li >
                            <li onClick={() => Filter('laptop')}># Laptop</li >
                            <li onClick={() => Filter('watch')}># Watch</li >
                            <li onClick={() => Filter('speaker')}># Speaker</li >
                        </ul>
                    </div>
                </div>
```

```
                    <div className="banner">
                        <div className="img_box">
                            <img src ='EIKONES/shop_left.jpeg' alt="banner"></img>
                        </div>
                    </div>
                </div>
                <div className="right_box">
                    <div className="banner">
                        <div className="img_box">
                            <img src="EIKONES/right-banner.jpeg" alt="banner"></img>
                        </div>
                    </div>

                    <div className="sort-options">
                        <label>Sort by: </label>
                        <select onChange={(e) => setSortOption(e.target.value)} value={sortOption}>
                            <option value="price">Price</option>
                            <option value="name">Name</option>
                        </select>
                    </div>
```

```
                <div className="product_box">
                    <h2>Shop Product</h2>
                    <div className="product_container">
                        {
                            shop.map((curElm) =>
                            {
                                return(
                                    <>
                                    <div className="box">
                                        <div className="img_box">
                                            <img src={curElm.image}></img>
                                            <div className="icon">
                                            <li ><FaHeart /></li >
                                            <li onClick={() => detailpage(curElm)}><FaEye /></li >
                                            </div>
                                        </div>
                                        <div className="detail">
                                            <h3>{curElm.Name}</h3>
                                            <p>€{curElm.price}</p>
                                            <button onClick={() => addtocart (curElm)}>Add to cart</button>
                                        </div>
                                    </div>
                                    </>
                                )
                            })
                        }
                    </div>
                </div>
            </div>
        </div>
        </>
    )
```

Σημαντικές Συναρτήσεις

Λεπτομέρειες Προϊόντος (detailpage & closedetail)

detailpage:

Περιγραφή:

- Καθορίζει τα δεδομένα του προϊόντος που θα εμφανιστούν στη σελίδα λεπτομερειών.

Πώς λειτουργεί:

- Παίρνει το προϊόν που επιλέχθηκε και ενημερώνει το detail state με τις πληροφορίες του.
- Ενεργοποιεί την προβολή λεπτομερειών (setShowDetail(true)).

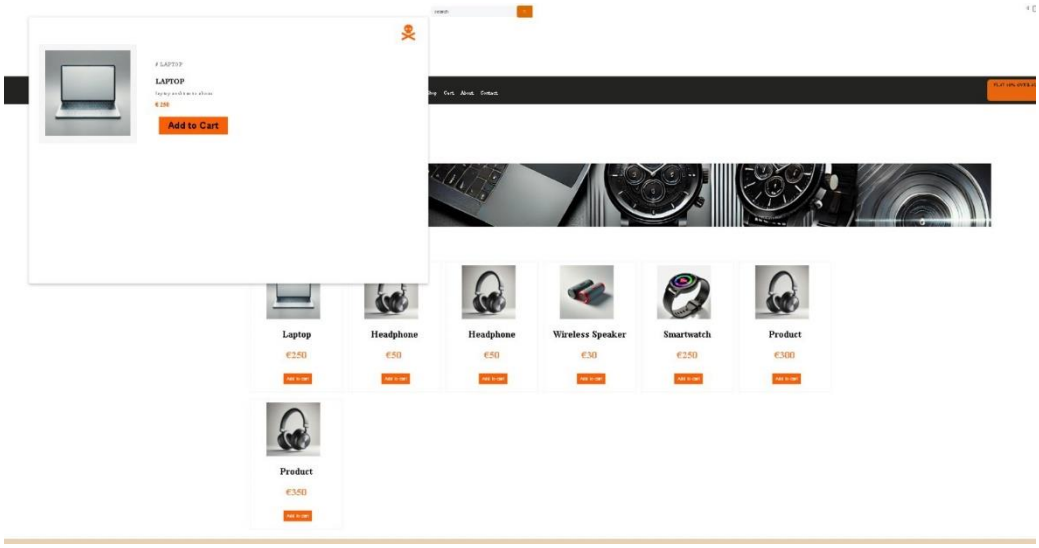
closedetail:

Περιγραφή:

- Απενεργοποιεί την προβολή λεπτομερειών προϊόντος.

Πώς λειτουργεί:

- Ορίζει το setShowDetail σε false, ώστε να κλείσει η προβολή.



Ταξινόμηση Προϊόντων (sortedShop)

Περιγραφή:

- Ταξινομεί τα προϊόντα της σελίδας με βάση την τιμή ή το όνομα.Πώς λειτουργεί:

Αντιγράφει τον πίνακα shop και τον ταξινομεί με βάση την επιλεγμένη επιλογή (sortOption).
Οι τιμές της επιλογής μπορεί να είναι:

- price: Ταξινόμηση κατά τιμή (αύξουσα).
- name: Ταξινόμηση κατά αλφαβητική σειρά του ονόματος.

Σημασία:

- Προσφέρει δυναμική ταξινόμηση των προϊόντων, διευκολύνοντας την εύρεση κατάλληλων αντικειμένων.

Φιλτράρισμα Προϊόντων (Filter & allcatefilter)

Filter:

Περιγραφή:

- Εμφανίζει προϊόντα που ανήκουν σε συγκεκριμένη κατηγορία.

Πώς λειτουργεί:

- Φιλτράρει τα προϊόντα με βάση την κατηγορία που επιλέχθηκε από το χρήστη.

allcatefilter:

Περιγραφή:

- Επαναφέρει τη λίστα προϊόντων στην αρχική της κατάσταση (όλα τα προϊόντα).

Πώς λειτουργεί:

- Εμφανίζει όλα τα προϊόντα ανεξαρτήτως κατηγορίας.

Σημασία:

Επιτρέπει στους χρήστες να φιλτράρουν προϊόντα ανά κατηγορία ή να επαναφέρουν τη λίστα.

Σχέση με Άλλα Components

- Διαχείριση του shop:

Ο πίνακας shop περνάει ως prop στο Shop component από το App.js. Περιέχει τα προϊόντα που προβάλλονται στη σελίδα.

- Επικοινωνία με το Καλάθι (addtocart):

Η συνάρτηση addtocart παρέχεται επίσης από το App.js και επιτρέπει στους χρήστες να προσθέσουν προϊόντα στο καλάθι τους απευθείας από τη σελίδα του καταστήματος.

- Διαχείριση State:

Χρησιμοποιούνται useState hooks για τη διαχείριση:

1. **Λεπτομέρειες προϊόντος:** detail.
2. **Εμφάνιση λεπτομερειών:** showDetail.
3. **Επιλογές ταξινόμησης:** sortOption.

Διαχείριση Κενών Περιπτώσεων

- Αν δεν υπάρχουν προϊόντα σε μια κατηγορία, η προβολή δεν θα αποδώσει δεδομένα.

- Το shop είναι ήδη ταξινομημένο μέσω του sortedShop, οπότε οποιοδήποτε κενό αποτέλεσμα επιστρέφει απλώς μια άδεια λίστα.

Δυναμική Διαχείριση Προϊόντων

Τα προϊόντα εμφανίζονται μέσω του map():

- Περιέχουν εικόνα, όνομα, τιμή και κουμπί "Add to Cart".
- Το κουμπί "Eye" (FaEye) καλεί τη λειτουργία detailpage, ανοίγοντας τη σελίδα λεπτομερειών.

3.4 Συμπεράσματα

(Ενότητα 3.4.)

Σημαντικές Συναρτήσεις:

- **detailpage & closedetail:** Διαχειρίζονται την προβολή λεπτομερειών προϊόντος.
- **sortedShop:** Υποστηρίζει την ταξινόμηση των προϊόντων.
- **Filter & allcatefilter:** Φιλτράρουν προϊόντα ανά κατηγορία.

Επικοινωνία:

- Το Shop επικοινωνεί άμεσα με το App.js μέσω των props shop, Filter, allcatefilter και addtocart.

User Experience:

- Προσφέρει φίλτρα κατηγοριών, ταξινόμηση και δυνατότητα προβολής λεπτομερειών.
- Το κουμπί "Add to Cart" προσθέτει προϊόντα στο καλάθι.

Δημιουργία Καλαθιού Αγορών

Το καλάθι αγορών είναι ένα βασικό στοιχείο ενός ηλεκτρονικού καταστήματος, που επιτρέπει στους χρήστες να προσθέτουν, να αφαιρούν και να διαχειρίζονται τα προϊόντα που σκοπεύουν να αγοράσουν.

#CART

SPEAKER
WIRELESS SPEAKER
PRICE: €30
TOTAL: €30

SUB TOTAL: € 30

checkout

GREAT SAVING
you are the best

24/7 SUPPORT
you are the best

MONEY BACK
you are the best

G.A.T.A

Ανυπομονούμε να σας εξυπηρετήσουμε ξανά σύντομα. Μη διστάσετε να επικοινωνήσετε μαζί μας για οποιαδήποτε απορία ή ανάγκη.

YOUT ACCOUNT YOUT ACCOUNT CONTACT US

#CART

SPEAKER
WIRELESS SPEAKER
PRICE: €30
TOTAL: €30

LAPTOP
LAPTOP
PRICE: €250
TOTAL: €500

SUB TOTAL: € 530

checkout

Ανάλυση Σημαντικών Σημείων στον Κώδικα

Ο κώδικας αφορά το **καλάθι αγορών**. Εδώ είναι οι βασικές λειτουργίες και τα σημαντικά σημεία που αναλύονται:

```
import React from 'react'
import './cart.css'
import { Link } from 'react-router-dom'
import { FaRegWindowClose } from "react-icons/fa";

const Cart = ({cart, setCart}) => {
  //increase quantity of cart product
  const incqty = (product) =>
  {
     const  exist = cart.find((x) =>
     {
       return x.id === product.id
     })
     setCart(cart.map((curElm)=>
     {
       return curElm.id === product.id ? {...exist, qty:exist.qty + 1} : curElm
     }))
}
  //decrease quantity of cart product
  const decqty =(product) =>
  {
    const  exist = cart.find((x) =>
     {
       return x.id === product.id
     })
     setCart(cart.map((curElm)=>
     {
       return curElm.id === product.id ? {...exist, qty:exist.qty - 1} : curElm
     }))
  }
```

```
//removing cart product
const removeproduct =(product) =>
{
  const exist = cart.find((x) =>
  {
    return x.id === product.id
  })
  if(exist.qty > 0 )
  {
    setCart(cart.filter((curElm) =>
    {
     return curElm.id !== product.id
    }))
  }
}
```

```
<div className='container'>
  {
    cart.map((curElm) =>
    {
      return(
        <>
        <div className='box'>
          <div className='img_box'>
            <img src={curElm.image} alt='image'></img>
          </div>
          <div className='detail'>
            <div className='info'>
            <h4>{curElm.cat}</h4>
            <h3>{curElm.Name}</h3>
            <p>Price: €{curElm.price}</p>
            <p>Total: €{curElm.price * curElm.qty}</p>
            </div>

            <div className='quantity'>
              <button onClick={() => incqty(curElm)}>+</button>
              <input type='number' value={curElm.qty}></input>
              <button onClick={() => decqty(curElm)}>-</button>
              </div>
            <div className='icon'>
              <li onClick={() =>removeproduct(curElm)}><FaRegWindowClose /></li>
            </div>

            </div>
          </div>

        </>
      )
    })
```

```
          </div>
          <div className='bottom'>
            {
              cart.length > 0 &&
              <>
                <div className='Total'>
                  <h4>Sub Total: € {total} </h4>
                </div>
                <button>
                  checkout
                </button>
              </>
            }


          </div>

        </div>

      </>

  )
}

export default Cart
```

Βασικές Συναρτήσεις

Αύξηση Ποσότητας Προϊόντος (incqty)

Περιγραφή:

- Αυξάνει την ποσότητα (qty) ενός προϊόντος στο καλάθι.

Πώς λειτουργεί:

- Ελέγχει αν το προϊόν υπάρχει στο καλάθι.
- Αν το προϊόν υπάρχει, αυξάνει την ποσότητα του κατά 1.
- Χρησιμοποιεί τη setCart για να ενημερώσει το state με τη νέα ποσότητα.
- Επιτρέπει στους χρήστες να αυξάνουν την ποσότητα προϊόντων απευθείας από το καλάθι.

Μείωση Ποσότητας Προϊόντος (decqty)

Περιγραφή:

- Μειώνει την ποσότητα (qty) ενός προϊόντος στο καλάθι.

Πώς λειτουργεί:

- Ελέγχει αν το προϊόν υπάρχει.
- Αν υπάρχει, μειώνει την ποσότητα του κατά 1.
- Αν η ποσότητα φτάσει στο 0, το προϊόν μένει στο καλάθι (δεν διαγράφεται αυτόματα).

Σημασία:

- Δίνει ευελιξία στους χρήστες να μειώνουν την ποσότητα προϊόντων.

Αφαίρεση Προϊόντος (removeproduct)

Περιγραφή:

- Αφαιρεί εντελώς ένα προϊόν από το καλάθι.

Πώς λειτουργεί:

- Ελέγχει αν το προϊόν υπάρχει.

- Αν το προϊόν υπάρχει, το αφαιρεί από τον πίνακα cart μέσω της filter.

Σημασία:

- Καθαρίζει το καλάθι αφαιρώντας προϊόντα που δεν θέλει πλέον ο χρήστης.Υπολογισμός Συνολικής Τιμής (total)

Περιγραφή:

- Υπολογίζει τη συνολική τιμή όλων των προϊόντων στο καλάθι.

Πώς λειτουργεί:

- Χρησιμοποιεί τη reduce για να υπολογίσει το γινόμενο της ποσότητας (qty) και της τιμής (price) κάθε προϊόντος και να επιστρέψει το άθροισμα.

Σημασία:

- Εμφανίζει το σύνολο των χρημάτων που πρέπει να πληρώσει ο χρήστης.

Σχέση με άλλα Components

Προέλευση των Props (cart καI SetCart):

- Το cart και το setCart έρχονται από το App.js μέσω props.
- Το App.js διαχειρίζεται το state του καλαθιού, επιτρέποντας στο Cart.js να ενημερώνει τις αλλαγές.

Σχέση με addtocart:

- Η addtocart είναι η αρχική συνάρτηση που προσθέτει προϊόντα στο καλάθι. Αν και δεν εμφανίζεται εδώ, συνεργάζεται με τις incqty και decqty για τη διαχείριση της ποσότητας.

Σχέση με shop καI home:

- Τα προϊόντα προέρχονται από το home_product.js, που εμφανίζονται στο shop. Από εκεί προστίθενται στο καλάθι.

Διαχείριση Κενών Περιπτώσεων

- Κενό Καλάθι:

Αν το καλάθι είναι άδειο (cart.length === 0), εμφανίζεται ένα μήνυμα με κουμπί
Shop now που οδηγεί τον χρήστη στη σελίδα του καταστήματος (/shop).

Σημαντικές Συναρτήσεις:

- **incqty καI decqty:** Διαχειρίζονται την ποσότητα των προϊόντων.
- **removeproduct:** Αφαιρεί προϊόντα από το καλάθι.
- **total:** Υπολογίζει τη συνολική τιμή.

Κεντρική Λειτουργία:

Το καλάθι επικοινωνεί με το App.js μέσω του cart και του setCart, διασφαλίζοντας ότι οποιαδήποτε αλλαγή ενημερώνεται σε όλη την εφαρμογή.

User Experience:

- Περιλαμβάνει λειτουργίες για αύξηση, μείωση και αφαίρεση προϊόντων.
- Παρέχει σαφή ενημέρωση για την κατάσταση του καλαθιού (κενό ή γεμάτο).
- Ο υπολογισμός της συνολικής τιμής ενισχύει την ευκολία στις αγορές.

Δυναμική Διαχείριση Προϊόντων

Εμφάνιση Προϊόντων στο Καλάθι:

- Τα προϊόντα εμφανίζονται δυναμικά μέσω του cart.map().

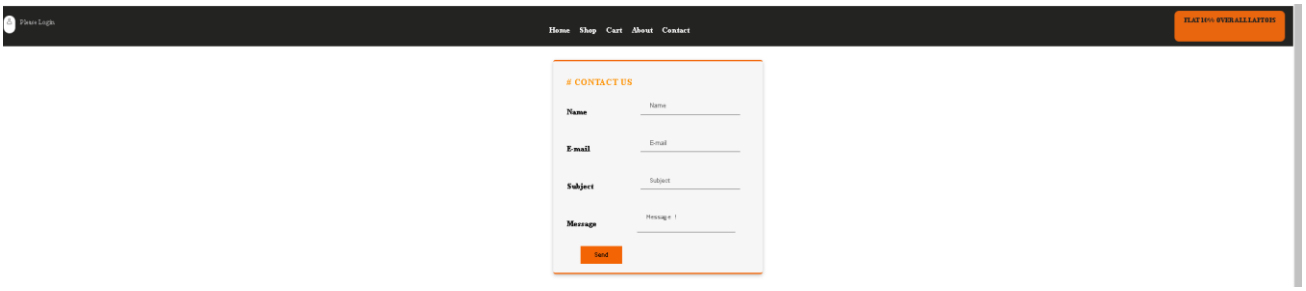
Σημασία:

- Διασφαλίζει ότι η εμφάνιση του καλαθιού προσαρμόζεται δυναμικά με βάση τα προϊόντα που υπάρχουν σε αυτό

Ανάλυση του Contact Page

Η εικόνα παρουσιάζει τη σελίδα "Contact Us", η οποία περιλαμβάνει μια φόρμα επικοινωνίας για τη συλλογή πληροφοριών από τον χρήστη.

Ακολουθεί αναλυτική περιγραφή της διάταξης και των λειτουργιών:



Φόρμα Επικοινωνίας

Η φόρμα είναι το κύριο στοιχείο της σελίδας και βρίσκεται στο κέντρο της οθόνης.

Πεδία της Φόρμας

Name:

- Πεδίο εισαγωγής του ονόματος του χρήστη.

E-mail:

- Πεδίο εισαγωγής της ηλεκτρονικής διεύθυνσης.

Subject:

- Πεδίο για τη θεματολογία του μηνύματος.

Message:

- Μεγαλύτερο πεδίο για την εισαγωγή του μηνύματος.

Κουμπί Υποβολής

- Το κουμπί **"Send"** βρίσκεται στο κάτω μέρος της φόρμας και στέλνει τα δεδομένα μέσω HTTP POST.

Λειτουργικότητα

Δυναμική Συμπεριφορά Φόρμας:

- Όταν ο χρήστης εισάγει δεδομένα στα πεδία, αυτά αποθηκεύονται δυναμικά στο state της εφαρμογής (useState).

Αποστολή Δεδομένων:

- Το κουμπί "Send" καλεί τη συνάρτηση send, η οποία:
    - Ελέγχει τα δεδομένα που έχουν εισαχθεί.
    - Αποστέλλει τα δεδομένα σε εξωτερικό database server (Firebase).
    - Ενημερώνει τον χρήστη για την επιτυχία ή την αποτυχία της αποστολής.

Μηνύματα Ενημέρωσης:

- Εμφανίζονται μηνύματα όπως:
    - **"Message Sent"** σε περίπτωση επιτυχίας.
    - **"Error Occurred: Message Failed"** σε περίπτωση αποτυχίας.

```
import React, { useState } from 'react'
import './contact.css'
import { MdSubject } from 'react-icons/md'
import { json } from 'react-router'

const Contact = () => {

    const [user,setUser] = useState(
        {
            Name:'',email:'',subject:'',Message:''
        }
    )
        let values, names
    const data = (e) =>
    {
        values = e.target.value
        names = e.target.name
        setUser({...user,[names]: values})
    }

    const send = async (e) =>
    {
        const{Name,email,subject,Message} = user
        e.preventDefault()
        const option = {
            method: 'POST',
            headers: {
                'Content-Type': 'aplication/json'
            },
            body: JSON.stringify({
                Name,email,subject,Message
            })
        }
```

```
        const send = await fetch(
            'https://projectx-5a66b-default-rtdb.firebaseio.com/Message.json',option
        )

        if (send) {
            alert ("Message Sent")
        }
        else
        {
            alert ("Error Occoured Message Sent Failed")
        }
    }

```

```
return (
  <>
      <div className='contact'>
          <div className='container'>
              <div className='form'>
                  <h2># contact us</h2>
                  <form method='POST'>
                      <div className='box'>
                          <div className='lable'>
                              <h4>Name</h4>
                          </div>
                          <div className='input'>
                              <input type='text' placeholder='Name' value={user.Name} name='Name' onChange={data}></input>
                          </div>
                      </div>
                      <div className='box'>
                          <div className='lable'>
                              <h4>E-mail</h4>
                          </div>
                          <div className='input'>
                              <input type='email' placeholder='E-mail' value={user.email} name='email' onChange={data}></input>
                          </div>
                      </div>
                      <div className='box'>
                          <div className='lable'>
                              <h4>Subject</h4>
                          </div>
                          <div className='input'>
                              <input type='text' placeholder='Subject' value={user.subject} name='subject' onChange={data}></input>
                          </div>
                      </div>
```

```
                      <div className='box'>
                          <div className='lable'>
                              <h4>Message</h4>
                          </div>
                          <div className='input'>
                              <textarea placeholder='Message !' value={user.Message} name='Message' onChange={data} > </textarea>
                          </div>
                      </div>

                      <button type='sublit' onClick={send}>Send</button>
                  </form>
              </div>
          </div>
      </div>

  </>
)
}

export default Contact
```

Ανάλυση του Component Contact

Το Contact component είναι υπεύθυνο για τη διαχείριση και αποστολή μηνυμάτων μέσω μιας φόρμας επικοινωνίας. Χρησιμοποιεί React useState για τη διαχείριση της κατάστασης και

κάνει POST αιτήματα σε μία εξωτερική βάση δεδομένων (Firebase).

Σημαντικές Συναρτήσεις

- Διαχείριση εισόδου Δεδομένων (data)

Περιγραφή:

- Χειρίζεται τις αλλαγές στις τιμές των πεδίων εισαγωγής της φόρμας (input).

Πώς λειτουργεί:

- Κάθε φορά που ο χρήστης πληκτρολογεί κάτι σε ένα πεδίο, αυτή η συνάρτηση ενημερώνει το state user με την τιμή του πεδίου.
- Το names αναφέρεται στο name του πεδίου, και το values είναι η τιμή που εισάγεται.

Σημασία:

- Δυναμική ενημέρωση του state χωρίς να απαιτείται ξεχωριστή συνάρτηση για κάθε πεδίο.

Αποστολή Μηνύματος (send)

Περιγραφή:

- Αποστέλλει τα δεδομένα της φόρμας σε μία εξωτερική βάση δεδομένων (Firebase) μέσω HTTP POST αιτήματος.

Πώς λειτουργεί:

- Παίρνει τα δεδομένα από το state user και τα μετατρέπει σε JSON μέσω της body ιδιότητας.
- Χρησιμοποιεί τη fetch για να στείλει τα δεδομένα στη διεύθυνση https://projectx-5a66b-default-rtdb.firebaseio.com/Message.json.

Σημασία:

- Συνδέει τη φόρμα με έναν server/database για τη συλλογή δεδομένων επικοινωνίας.
- Ενημερώνει τον χρήστη για την επιτυχία ή αποτυχία της αποστολής.

Διαχείριση Κατάστασης (State Management)

- Το state user διαχειρίζεται όλα τα πεδία της φόρμας.
- Αρχικοποιείται ως ένα αντικείμενο με ιδιότητες (Name, email, subject, Message) που αντιστοιχούν στα πεδία της φόρμας.

Σημασία:

- Οργανώνει τα δεδομένα εισόδου σε μία ενιαία δομή που είναι εύκολα διαχειρίσιμη.Φόρμα Εισαγωγής Δεδομένων

  Δομή Φόρμας
  Περιλαμβάνει τέσσερα πεδία:

  1. Όνομα (Name)
  2. E-mail
  3. Θέμα (Subject)
  4. Μήνυμα (Message)

  Κάθε πεδίο έχει τις ακόλουθες λειτουργίες:

1. value: Ελέγχεται από το state (user)
2. name: Χρησιμοποιείται για τη δυναμική ενημέρωση του state.
3. onChange: Καλεί τη συνάρτηση data για να ενημερώσει το state.

Κουμπί Αποστολής

- Το κουμπί "Send" καλεί τη συνάρτηση send όταν πατηθεί.
- Το method="POST" υποδεικνύει την πρόθεση αποστολής δεδομένων στον server.

.

Σχέση με Άλλα Components

- Το Contact component λειτουργεί ανεξάρτητα και δεν φαίνεται να επικοινωνεί άμεσα με άλλα components.
- Ωστόσο, η αποστολή δεδομένων σε μία εξωτερική βάση δεδομένων επιτρέπει την επεξεργασία των δεδομένων αυτών από άλλα μέρη της εφαρμογής.

Δυνατότητες Επέκτασης

Επαλήθευση Δεδομένων (Validation):

- Μπορεί να προστεθεί έλεγχος πριν από την αποστολή, ώστε να διασφαλιστεί ότι τα πεδία δεν είναι κενά ή ότι το email είναι έγκυρο.

Προβολή Κατάστασης Αποστολής:

- Χρήση ενός loading state για να ενημερώνεται ο χρήστης ότι το μήνυμα αποστέλλεται.

Εμφάνιση Επιβεβαίωσης:

- Μετά την επιτυχημένη αποστολή, μπορεί να προστεθεί μήνυμα επιβεβαίωσης στην οθόνη.

Συμπεράσματα

Σημαντικές Συναρτήσεις:

- data: Διαχειρίζεται τις αλλαγές στα πεδία της φόρμας.
- send: Αποστέλλει τα δεδομένα στον server (Firebase).

Κεντρική Λειτουργία:

- Συνδέει τη φόρμα επικοινωνίας με μία βάση δεδομένων, καθιστώντας δυνατή τη συλλογή και αποθήκευση μηνυμάτων.

User Experience:

- Εύχρηστη φόρμα με δυναμική ενημέρωση των πεδίων και ενημέρωση του χρήστη για την επιτυχία/αποτυχία αποστολής.

ΚΕΦΑΛΑΙΟ 4 - BACKEND

4.1 Εισαγωγή

(Ενότητα 4.1)

Η ανάπτυξη ενός συστήματος ηλεκτρονικού εμπορίου (**e-commerce**) απαιτεί οργανωμένη και ασφαλή διαχείριση των (**routes**) δρομολογίων **backend**. Αυτό με τη σειρά του επιτρέπει τη σωστή αλληλεπίδραση μεταξύ χρηστών, προϊόντων και κριτικών. Στην παρούσα εργασία, το backend υλοποιείται με τη χρήση **Node.js** και **Express.js,** χρησιμοποιώντας την αρχιτεκτονική **RESTful API** για τη διαχείριση των δεδομένων και την επικοινωνία με το frontend.

Τα routes που αναλύθηκαν καλύπτουν τρεις βασικές λειτουργικές περιοχές:

1. **Διαχείριση Προϊόντων (Product Routes)**
    - Υλοποιούνται (endpoints) τελικά σημεια για τη δημιουργία,την ενημέρωση, τη διαγραφή και την ανάκτηση προϊόντων.
    - Το ενδιάμεσο λογισμικό (middleware) χρησιμοποιείται για την προστασία και τον έλεγχο πρόσβασης (authentication & authorization).
    - Υποστηρίζονται nested (routes) διαδρομές για τη διαχείριση σχετικών δεδομένων (π.χ. κριτικές προϊόντων).
2. **Διαχείριση Αξιολογήσεων (Review Routes)**
    - Οι χρήστες επιτρέπεται να δημιουργούν, να διαβάζουν, να επεξεργάζονται και να διαγράφουν κριτικές προϊόντων.
    - Ορίζονται σχέσεις μεταξύ προϊόντων και κριτικών, επιτρέποντας την εμφωλευμένη δρομολόγηση (/products/:productId/reviews).
    - Ενσωματώνεται εξουσιοδότηση βάσει ρόλων (role-based authorization), ώστε μόνο οι κατάλληλοι χρήστες να μπορούν να διαχειρίζονται τις κριτικές.

3. **Διαχείριση Χρηστών (User Routes)**

- Υλοποιούνται λειτουργίες authentication & authorization μέσω JWT tokens.
- Οι χρήστες μπορούν να εγγραφούν, συνδεθούν, αλλάξουν κωδικό, ενημερώσουν ή διαγράψουν το προφίλ τους.
- Οι διαχειριστές (admin, moderator) έχουν πρόσθετες δυνατότητες διαχείρισης χρηστών.

4. Διαχειριση καλαθιου

Αυτός ο κώδικας προσθέτει μια διαδρομή που επιτρέπει στους πιστοποιημένους χρήστες να δημιουργήσουν/διαχειρίστουν μια συνεδρία checkout για ένα συγκεκριμένο προϊόν, προστατεύοντας ταυτόχρονα την πρόσβαση με έλεγχο αυθεντικοποίησης.

5.Διαχειριση παραγγελιας

Ο κώδικας αυτός καθορίζει διάφορες διαδρομές για τη δημιουργία, ανάκτηση και ενημέρωση παραγγελιών, προστατεύοντας τες με middleware για έλεγχο ταυτότητας και ρόλων (admin) όπου χρειάζεται.

Κάθε route έχει σχεδιαστεί με βάση **πρακτικών ανάπτυξης RESTful APIs**, εξασφαλίζοντας:

- **Καθαρό διαχωρισμό ευθυνών** με controllers για την επεξεργασία αιτημάτων.
- **Middleware authentication & authorization** για ασφάλεια.
- **Modular δομή**, διευκολύνοντας την επεκτασιμότητα και τη συντήρηση του κώδικα.

Στη συνέχεια, αναλύονται λεπτομερώς τα routes του backend, περιγράφοντας τη λειτουργικότητα και τις τεχνολογίες που χρησιμοποιούνται.

4.2 Διαχείριση Προϊόντων (Product Routes)- Ανάλυση Κώδικα

(Ενότητα 4.2)

Αρχική Διαμόρφωση και Εισαγωγή Modules

- express: Το Express.js framework χρησιμοποιείται για τη διαχείριση των διαδρομών του API.
- prodController: Εισάγει το **controller των προϊόντων**, που περιέχει τη λογική για τις CRUD operations (Create, Read, Update, Delete).
- authController: Περιέχει συναρτήσεις **authentication και authorization**, που ελέγχουν αν ένας χρήστης έχει τα απαραίτητα δικαιώματα για να εκτελέσει μια ενέργεια.
- reviewRouter: Ορίζεται ένα ξεχωριστό **router** για τις αξιολογήσεις (reviews) των προϊόντων, διαχωρίζοντας τη λογική σε διαφορετικά αρχεία για καλύτερη οργάνωση.

Ορισμός του Router Middleware

Το express.Router() δημιουργεί ένα νέο router object, επιτρέποντας την ομαδοποίηση και διαχείριση των διαδρομών.

Σύνδεση των Routes για τις Αξιολογήσεις (Nested Routes)

Ορίζεται ένα **nested route**, ώστε κάθε προϊόν να μπορεί να έχει τις δικές του αξιολογήσεις (/api/v1/products/:productId/reviews). Η διαδρομή αυτή συνδέεται με το **reviewRouter**, επιτρέποντας την εύκολη διαχείριση των σχετικών δεδομένων.

Στατιστικά Προϊόντων

Ορίζεται ένα route /product-stats που επιστρέφει **στατιστικά στοιχεία** για τα προϊόντα, πιθανώς χρησιμοποιώντας **aggregation pipelines της MongoDB** για ανάλυση δεδομένων.

Διαχείριση Όλων των Προϊόντων

- **GET** /api/v1/products/ → Επιστρέφει όλα τα προϊόντα.
- **POST** /api/v1/products/ → Δημιουργεί ένα νέο προϊόν, αλλά μόνο αν ο χρήστης έχει ρόλο admin ή moderator.

- Το authController.protect ελέγχει αν ο χρήστης είναι συνδεδεμένος.
- Το authController.restrictTo('admin', 'moderator') διασφαλίζει ότι μόνο χρήστες με αυτά τα δικαιώματα μπορούν να δημιουργήσουν ένα προϊόν.

Διαχείριση Συγκεκριμένου Προϊόντος

- **GET** /api/v1/products/:id → Επιστρέφει ένα προϊόν με βάση το ID του.
- **PATCH** /api/v1/products/:id → Ενημερώνει τα δεδομένα ενός προϊόντος, αλλά μόνο αν ο χρήστης έχει ρόλο admin ή moderator.
- **DELETE** /api/v1/products/:id → Διαγράφει ένα προϊόν, αλλά μόνο αν ο χρήστης έχει τα απαραίτητα δικαιώματα.

Ασφάλεια και Authorization

Το API περιλαμβάνει **authentication & authorization middleware**, το οποίο προστατεύει κρίσιμες λειτουργίες (δημιουργία, επεξεργασία, διαγραφή προϊόντων).

1. authController.protect
    - Ελέγχει αν ο χρήστης είναι συνδεδεμένος.
    - Βασίζεται σε **JWT (JSON Web Tokens)** για τον έλεγχο ταυτότητας.
2. authController.restrictTo('admin', 'moderator')
    - Ελέγχει αν ο χρήστης έχει τον κατάλληλο ρόλο (admin ή moderator).
    - Εμποδίζει την πρόσβαση σε συγκεκριμένα endpoints σε μη εξουσιοδοτημένους χρήστες.

Αρχές Σχεδίασης και Βέλτιστες Πρακτικές

Η δομή του κώδικα ακολουθεί **πρακτικές ανάπτυξης API**:

- **Διαχωρισμός λογικής**: Τα controllers, routes και middlewares είναι διαχωρισμένα σε αρχεία, βελτιώνοντας την αναγνωσιμότητα.
- **RESTful API**: Το API ακολουθεί τις αρχές των **RESTful services**, επιτρέποντας εύκολη επικοινωνία με το frontend.
- **Ασφάλεια**: Χρησιμοποιούνται **middleware authentication & role-based authorization**, αποτρέποντας την πρόσβαση σε μη εξουσιοδοτημένους χρήστες.
- **Modularization**: Ο κώδικας είναι οργανωμένος με **modular δομή**, κάνοντας την εφαρμογή πιο επεκτάσιμη.

4.3 Διαχείριση Αξιολογήσεων-Ανάλυση Κώδικα

(Ενότητα 4.3)

```
const express = require('express');
const reviewController = require('./../controllers/reviewController');
const authController = require('./../controllers/authController');

const router = express.Router({ mergeParams: true });

router.use(authController.protect);

router
    .route('/')
    .get(reviewController.getAllReviews)
    .post(
        authController.restrictTo('user'),
        reviewController.setProductUsersIds,
        reviewController.createReview
    );

router
    .route('/:id')
    .get(reviewController.getReview)
    .patch(
        authController.restrictTo('user', 'admin'),
        reviewController.updateReview)
    .delete(
        authController.restrictTo('user', 'admin'),
        reviewController.deleteReview);

module.exports = router;
```

Οι αξιολογήσεις αποτελούν βασικό στοιχείο των ηλεκτρονικών καταστημάτων, καθώς επιτρέπουν στους χρήστες να **βαθμολογούν και να σχολιάζουν προϊόντα**, βελτιώνοντας την αξιοπιστία και την εμπειρία των πελατών.

Η διαχείριση των αξιολογήσεων ακολουθεί πρότυπο **RESTful API**, επιτρέποντας στους χρήστες να δημιουργούν, ανακτούν, επεξεργάζονται και διαγράφουν αξιολογήσεις. Παράλληλα, εφαρμόζεται **authentication και role-based authorization**, ώστε να διασφαλιστεί ότι μόνο εξουσιοδοτημένοι χρήστες μπορούν να εκτελέσουν συγκεκριμένες ενέργειες.

Αρχική Διαμόρφωση και Εισαγωγή Module

- express: Το Express.js framework χρησιμοποιείται για τη διαχείριση των routes του API.
- reviewController: Περιέχει τη λογική που σχετίζεται με τις αξιολογήσεις (CRUD operations).
- authController: Περιέχει τη λογική για την **προστασία των routes μέσω authentication και authorization**.

Ορισμός του Router Middleware με Merge Params

- Χρησιμοποιείται το express.Router() για τη δημιουργία ενός νέου router object.
- Η mergeParams: true επιτρέπει στο review router να έχει πρόσβαση στις **παραμέτρους της διαδρομής (**productId**)**, ώστε να μπορεί να συνδεθεί με το **product router** και να δημιουργήσει **nested routes** (π.χ. /api/v1/products/:productId/reviews).

Authentication Middleware για Όλα τα Routes

- Όλες οι διαδρομές του **reviewRouter** προστατεύονται με το middleware authController.protect, που διασφαλίζει ότι μόνο οι συνδεδεμένοι χρήστες μπορούν να έχουν πρόσβαση στα reviews.

- Το authentication βασίζεται σε **JWT tokens**, τα οποία αποστέλλονται από τον client και ελέγχονται στο backend.

Διαχείριση Όλων των Αξιολογήσεων

- **GET** /api/v1/reviews/ → Επιστρέφει όλες τις αξιολογήσεις από τη βάση δεδομένων.
- **POST** /api/v1/reviews/ → Επιτρέπει τη δημιουργία μιας νέας αξιολόγησης, αλλά μόνο για χρήστες με ρόλο user.

Η post λειτουργία περιλαμβάνει τα εξής middleware:

- authController.restrictTo('user') → Διασφαλίζει ότι μόνο εγγεγραμμένοι χρήστες μπορούν να αφήσουν αξιολόγηση.
- reviewController.setProductUsersIds → Αυτό το middleware συμπληρώνει αυτόματα τα productId και userId, ώστε να καταγράφεται **ποιος χρήστης άφησε αξιολόγηση για ποιο προϊόν**.
- reviewController.createReview → Η κύρια συνάρτηση που χειρίζεται τη δημιουργία και αποθήκευση της αξιολόγησης στη βάση δεδομένων.

Διαχείριση Συγκεκριμένης Αξιολόγησης

- **GET** /api/v1/reviews/:id → Επιστρέφει μια αξιολόγηση με βάση το ID της.
- **PATCH** /api/v1/reviews/:id → Επιτρέπει την επεξεργασία μιας αξιολόγησης, αλλά μόνο αν ο χρήστης έχει ρόλο user **ή** admin.
- **DELETE** /api/v1/reviews/:id → Επιτρέπει τη διαγραφή μιας αξιολόγησης, αλλά μόνο για **τον χρήστη που τη δημιούργησε ή έναν admin**.

Ασφάλεια και Role-Based Access Control

Η διαχείριση των αξιολογήσεων είναι προστατευμένη μέσω των παρακάτω μηχανισμών:

Authentication (Έλεγχος Πρόσβασης)

- Όλες οι ενέργειες απαιτούν ο χρήστης να είναι **συνδεδεμένος**.
- Το **JWT authentication** χρησιμοποιείται για την ταυτοποίηση των χρηστών.

Role-Based Authorization

- Οι απλοί χρήστες (user) μπορούν να δημιουργούν αξιολογήσεις.
- Μόνο ο χρήστης που δημιούργησε την αξιολόγηση ή ένας admin μπορεί να την επεξεργαστεί ή να τη διαγράψει.

Αυτή η στρατηγική αποτρέπει **κακόβουλη επεξεργασία ή διαγραφή αξιολογήσεων από μη εξουσιοδοτημένους χρήστες**.

4.4 Διαχείριση Χρηστών (User Management)- Ανάλυση Κώδικα

(Ενότητα 4.4)

```
//2) Routes Handles
const express = require('express');
const userController = require('./../controllers/userController');
const authController = require('./../controllers/authController');



const router = express.Router(); //middleware

router.post('/signup', authController.signup);
router.post('/login', authController.login);
router.post('/forgotPassword', authController.forgotPassword);
router.patch('/resetPassword/:token', authController.resetPassword);

//Protect all routes after this middleware
router.use(authController.protect);

router.patch('/updateMyPassword', authController.updatePassword);
router.get('/me', userController.getMe, userController.getUser);
router.patch('/updateMe', userController.updateMe);
router.delete('/deleteMe', userController.deleteMe);

router.use(authController.restrictTo('admin', 'moderator'));

router
    .route('/')
    .get(userController.getAllUsers)
    .post(userController.createUser);

router
    .route('/:id')
    .get(userController.getUser)
    .patch(userController.updateUser)
    .delete(userController.deleteUser);



module.exports = router;
```

Εισαγωγή των Modules

- express: Το Express.js framework χρησιμοποιείται για τη διαχείριση των routes του API.
- userController: Περιέχει τη λογική διαχείρισης χρηστών (CRUD operations).
- authController: Περιέχει τη λογική για **authentication και authorization**.

Ορισμός του Router Middleware

- Χρησιμοποιείται το Express Router για τη δημιουργία ενός ξεχωριστού αρχείου διαχείρισης routes.

Δημιουργία Διαδρομών για Authentication

- **POST** /signup → Επιτρέπει στους νέους χρήστες να εγγραφούν στο σύστημα.
- **POST** /login → Ελέγχει τα διαπιστευτήρια των χρηστών και τους επιτρέπει να συνδεθούν.
- **POST** /forgotPassword → Επιτρέπει στους χρήστες να ζητήσουν επαναφορά του κωδικού πρόσβασης.
- **PATCH** /resetPassword/:token → Επιτρέπει στους χρήστες να αλλάξουν τον κωδικό πρόσβασης χρησιμοποιώντας έναν **μοναδικό κωδικό (token)**.

Αυτές οι λειτουργίες χρησιμοποιούν **JWT authentication** και **bcrypt για κρυπτογράφηση κωδικών**.

Προστασία των Routes μέσω Authentication Middleware

- Όλα τα routes που ακολουθούν **προστατεύονται** και μπορούν να προσπελαστούν **μόνο από συνδεδεμένους χρήστες**.

- Το **middleware** authController.protect ελέγχει αν ο χρήστης είναι αυθεντικοποιημένος, μέσω **JWT tokens**.

Λειτουργίες Χρήστη (User Profile Management)

- **PATCH** /updateMyPassword → Επιτρέπει στον χρήστη να αλλάξει τον κωδικό του.
- **GET** /me → Επιστρέφει τα προσωπικά στοιχεία του χρήστη που είναι συνδεδεμένος.
- **PATCH** /updateMe → Επιτρέπει στον χρήστη να **ενημερώσει τα προσωπικά του δεδομένα**.
- **DELETE** /deleteMe → Ο χρήστης μπορεί να **διαγράψει τον λογαριασμό του** από το σύστημα.

Περιορισμός Πρόσβασης σε Admin και Moderator

- Αυτό το middleware διασφαλίζει ότι **μόνο οι admin και moderator χρήστες μπορούν να προσπελάσουν τις επόμενες διαδρομές**.
- Αυτό είναι σημαντικό για τη **διαχείριση χρηστών** από τους διαχειριστές του συστήματος.

Διαχείριση Όλων των Χρηστών (Admin Panel)

- **GET** /api/v1/users/ → Επιστρέφει όλους τους χρήστες (μόνο για admin/moderator).
- **POST** /api/v1/users/ → Επιτρέπει τη δημιουργία νέων χρηστών από διαχειριστές.

Διαχείριση Συγκεκριμένου Χρήστη

- **GET** /api/v1/users/:id → Επιστρέφει έναν χρήστη με βάση το id.
- **PATCH** /api/v1/users/:id → Ενημερώνει τα στοιχεία ενός χρήστη.
- **DELETE** /api/v1/users/:id → Διαγράφει έναν χρήστη από τη βάση δεδομένων.

Οι παραπάνω λειτουργίες **περιορίζονται μόνο για τους admin και moderator**, διασφαλίζοντας **αποφυγή κακόβουλων ενεργειών από απλούς χρήστες**.

Ασφάλεια και Role-Based Access Control (RBAC)

Το API ακολουθεί **ασφαλείς πρακτικές** διαχείρισης χρηστών:

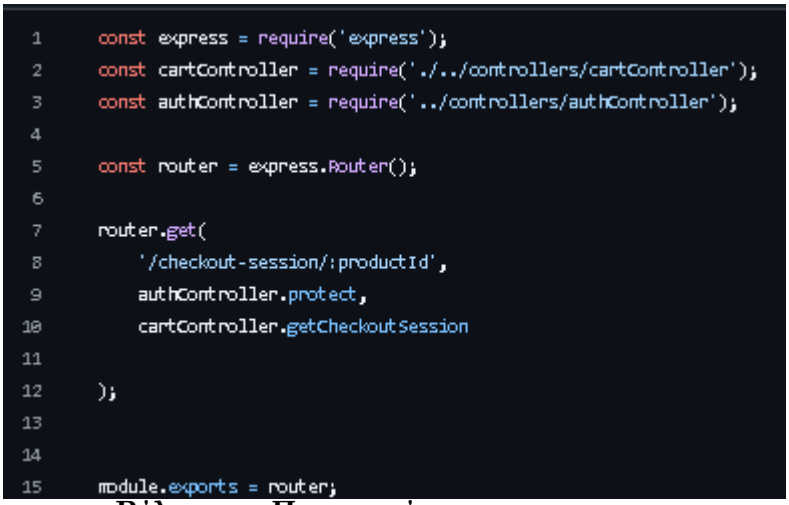
Authentication (JWT Authentication)

- Η χρήση **JWT (JSON Web Tokens)** επιτρέπει στους χρήστες να **συνδέονται και να παραμένουν αυθεντικοποιημένοι** για συγκεκριμένο χρονικό διάστημα.
- Η κρυπτογράφηση των κωδικών γίνεται με **bcrypt.js**, προστατεύοντας τα δεδομένα των χρηστών.

Authorization (Role-Based Access Control - RBAC)

- Η restrictTo('admin', 'moderator') διασφαλίζει ότι μόνο οι **εξουσιοδοτημένοι χρήστες** μπορούν να διαχειρίζονται άλλους χρήστες.

Οι απλοί χρήστες μπορούν να **επεξεργάζονται μόνο τα δικά τους δεδομένα**, αλλά δεν έχουν πρόσβαση σε άλλους χρήστες



```
const express = require('express');
const cartController = require('./../controllers/cartController');
const authController = require('../controllers/authController');

const router = express.Router();

router.get(
    '/checkout-session/:productId',
    authController.protect,
    cartController.getCheckoutSession

);


module.exports = router;
```

Αρχές Σχεδίασης και Βέλτιστες Πρακτικές

Το σύστημα διαχείρισης χρηστών έχει σχεδιαστεί με βάση τις παρακάτω **καλές πρακτικές**:

- **Separation of Concerns (Διαχωρισμός Ευθυνών)** → Τα routes, οι controllers και τα middleware είναι σε ξεχωριστά αρχεία.
- **JWT Authentication & Role-Based Authorization** → Προστασία endpoints μέσω authentication & authorization.

- **Middleware Protection** → Χρήση του authController.protect για προστασία των endpoints από μη συνδεδεμένους χρήστες.
- **Security Best Practices** → Χρήση bcrypt.js για ασφαλή αποθήκευση κωδικών
- **RESTful API Design** → Τα endpoints ακολουθούν τις πρότυπο REST για εύκολη διαχείριση.

4.5 Διαχείριση Κάρτας- Καλαθιού- Ανάλυση Κώδικα

(Ενότητα 4.5)

Εισαγωγή βιβλιοθηκών και controllers

- Χρησιμοποιείται το express για τη δημιουργία router.
- Γίνονται import δύο controllers:
    - Το cartController που χειρίζεται τη λογική του καλαθιού.
    - Το authController που χειρίζεται τον έλεγχο αυθεντικοποίησης/προστασίας.
- **Ορισμός του Router**

Δημιουργούμε ένα καινούριο Router αντικείμενο από την Express.

1. **Διαδρομή(Route):**/checkoutsession/:productidrouter.get('/checkoutsession/:productId',authController.protect,cartController.getCheckoutSession):

- Ο χρήστης πρέπει πρώτα να είναι συνδεδεμένος (χάρη στη protect μέθοδο του authController).
- Έπειτα, καλείται το getCheckoutSession του cartController, το οποίο συνήθως δημιουργεί ή επιστρέφει μια "payment session" για το συγκεκριμένο προϊόν (όπως π.χ. ενσωμάτωση με Paypal ή άλλο σύστημα πληρωμών).
- **Εξαγωγή του Router**
- Τέλος, το module.exports = router; επιτρέπει να χρησιμοποιήσεις αυτό το router σε άλλα μέρη της εφαρμογής, π.χ. app.use('/cart', cartRouter).

4.6 Διαχείριση Παραγγελίας- Ανάλυση Κώδικα

(Ενότητα 4.6)

```
const express = require('express');
const orderController = require('../controllers/orderController');
const authController = require('../controllers/authController');

const router = express.Router();

// Protect all routes after this middleware
router.use(authController.protect);

// Routes for creating an order and fetching all orders (Admin only)
router
  .route('/')
  .post(orderController.createOrder) // Create an order
  .get(authController.restrictTo('admin'), orderController.getAllOrders); // Admin: Get all orders

// Route for fetching logged-in user's orders
router.route('/myorders').get(orderController.getMyOrders);

// Routes for specific order actions
router
  .route('/:id')
  .get(orderController.getOrderById) // Get an order by ID
router
.route('/:id/pay')
//.put(authController.protect, orderController.updateOrderToPaid);
.put(orderController.updateOrderToPaid); // Mark an order as paid

router
  .route('/:id/deliver')
  .put(authController.restrictTo('admin'), orderController.updateOrderToDelivered); // Mark an order as delivered

module.exports = router;
```

τα δημιουργίας,
:

1. **Βασικό Setup**

const express = require('express');
const orderController = require('../controllers/orderController');
const authController = require('../controllers/authController');

const router = express.Router();

- Γίνεται εισαγωγή του Express για τη δημιουργία ενός νέου Router.
- Το orderController περιέχει τη λογική (business logic) για τις διάφορες πράξεις σε παραγγελίες.
- Το authController περιέχει middleware για αυθεντικοποίηση (έλεγχο token, ρόλων κ.λπ.).

2. **Προστασία Όλων των Διαδρομών**

router.use(authController.protect);

- Όλα τα endpoints που ακολουθούν απαιτούν ο χρήστης να είναι συνδεδεμένος (δηλ. να διαθέτει έγκυρο JWT ή άλλο μηχανισμό authentication).

3. **Διαδρομή** / **– Δημιουργία & Ανάκτηση Όλων των Παραγγελιών**

```
router
.route('/')
.post(orderController.createOrder) // Δημιουργία νέας παραγγελίας
.get(authController.restrictTo('admin'), orderController.getAllOrders); // Μόνο Admin μπορεί να δει όλες τις παραγγελίες
```

- **POST** /: Καλεί τη μέθοδο createOrder, η οποία καταχωρεί μια νέα παραγγελία για τον συνδεδεμένο χρήστη.

- **GET** /: Σε συνδυασμό με το authController.restrictTo('admin'), επιτρέπεται μόνο σε χρήστη με ρόλο Admin η εμφάνιση όλων των παραγγελιών (π.χ. για διαχειριστές e-shop).

4. **Διαδρομή** /myorders – **Προσωπικές Παραγγελίες**

router.route('/myorders').get(orderController.getMyOrders);

- **GET** /myorders: Επιστρέφει αποκλειστικά τις παραγγελίες που αντιστοιχούν στον ***τρέχοντα*** συνδεδεμένο χρήστη (π.χ. προβολή ιστορικού παραγγελιών).

5. **Διαδρομή** /:id – **Απόκτηση Παραγγελίας με Βάση το ID**

```
router
.route('/:id')
.get(orderController.getOrderById);
```

- **GET** /:id: Επιστρέφει μια παραγγελία συγκεκριμένου ID, εφόσον ο συνδεδεμένος χρήστης έχει δικαιώματα (συνήθως είναι είτε admin είτε ο ιδιοκτήτης της παραγγελίας).

6. **Διαδρομή** /:id/pay – **Σήμανση Παραγγελίας ως Πληρωμένη**

```
router
  .route('/:id/pay')
  .put(authController.protect, orderController.updateOrderToPaid);
```

- **PUT** /:id/pay: Καλεί την updateOrderToPaid και ενημερώνει ότι μια παραγγελία έχει εξοφληθεί (paid). Όπως και σε όλο το router, απαιτείται ο χρήστης να είναι συνδεδεμένος.

7. **Διαδρομή** /:id/deliver – **Σήμανση Παραγγελίας ως Παραδοθείσα**

```
router
  .route('/:id/deliver')
  .put(authController.restrictTo('admin'), orderController.updateOrderToDelivered);
```

- **PUT** /:id/deliver: Μόνο ένας admin μπορεί να "μαρκάρει" μια παραγγελία ως delivered, ενημερώνοντας έτσι το στάδιο της παραγγελίας.

8. **Εξαγωγή**

```
module.exports = router;
```

- Επιτρέπει την εισαγωγή του router (π.χ. ordersRouter) σε άλλο αρχείο της εφαρμογής, με app.use('/api/v1/orders', ordersRouter).

4.7 Εισαγωγή στο Product Model του E-Commerce Backend

(Ενότητα 4.7)

Το **Product Model** αποτελεί ένα από τα βασικά συστατικά ενός **e-commerce συστήματος**, καθώς αποθηκεύει και διαχειρίζεται τα δεδομένα των προϊόντων. Η υλοποίηση γίνεται με τη **MongoDB**, χρησιμοποιώντας το **Mongoose ORM**, το οποίο επιτρέπει την ευκολότερη διαχείριση της βάσης δεδομένων μέσω ενός **σχήματος (**Schema**) που καθορίζει τη δομή των προϊόντων**.

Στο συγκεκριμένο μοντέλο, εφαρμόζονται **διάφορες λειτουργίες και τεχνικές**, όπως:

- **Validation και Data Constraints** για την αποφυγή εσφαλμένων δεδομένων.
- **Indexes και Query Optimization** για ταχύτερη αναζήτηση προϊόντων.
- **Virtual Population** για σύνδεση των προϊόντων με τις αξιολογήσεις τους (reviews).

- **Middleware Functions (Pre/Post Hooks)** που εκτελούν λειτουργίες πριν ή μετά την αποθήκευση και την αναζήτηση δεδομένων.
- **Automatic Slug Generation** για δημιουργία SEO-friendly URLs.
- **Price Discount Validation** ώστε οι εκπτώσεις να είναι πάντα μικρότερες από την αρχική τιμή.

Το Product Schema είναι σχεδιασμένο με τρόπο που επιτρέπει τη **βέλτιστη απόδοση, ασφάλεια και επεκτασιμότητα**, διασφαλίζοντας ότι η διαχείριση των προϊόντων γίνεται με σωστό και αποδοτικό τρόπο. Στη συνέχεια, αναλύεται ο κώδικας και οι επιμέρους λειτουργίες που περιλαμβάνει.

```
const mongoose = require('mongoose');
const slugify = require('slugify');
const validator = require('validator');


const productSchema = new mongoose.Schema({ //pass scema as an object
    name: {
        type: String,
        required: [true, 'A product must have a name'],
       unique: true,
        trim: true,
        maxlength: [40, 'A product name must have less or equal than 40 characters'],
        minlength: [4, 'A product name must have more or equal than 4 characters'],
        //validate: [validator.isAlphanumeric, 'Product name only contain caracters and numbers']
    },
    slug: String,
    ratingsQuantity: {
        type: Number,
        default: 0
    },
    ratingsAverage: {
            type: Number,
            default:4.5,
            min: [1, 'Rating must be between 1.0 and 5 stars'],
            max: [5, 'Rating must be between 1.0 and 5.00 stars'],
            set: val => Math.round(val * 10) / 10
    },
    summary: {
        type: String,
        required: [true, 'A product must have a description']
    },
    description: {
        type: String,
        trim: true,
    },
    imageCover: {
        type: String,
        //required:[true, 'A product must have cover image']
    },
    images: [String],
    brand: {
        type: String,
        default: ''
    },
    countInStock: {
        type: Number,
        required: [true, ' A product must have
        min: 0,
        max: 255
    },
    isFeatured: {
        type: Boolean,
        default: false,
    },
        },
    createdAt: {
        type: Date,
        default: Date.now(),
        select: false
    },
    price:  {
            type: Number,
            required: [true, 'A product must have a price']
    },
    priceDiscount: {
        type: Number,
        validate: {
            validator: function(val) {
                // this only points to current doc on New document creation
                return val < this.price; //100 < 200
            },
            message: 'Discount price ({VALUE}) should be below the regular price'
        }
    },
    startDates: [Date],
    secretProduct: {
        type: Boolean,
        default: false
    },
    id: {
        type: Number
    }
   },
{
    toJSON: { virtuals: true },
    toObject: { virtuals: true}

});


productSchema.index({price: 1, ratingsAverage: -1 });
productSchema.index({ slug: 1 });


// Virtual populate
productSchema.virtual('reviews', {
    ref: 'Review',
    foreignField: 'product',
    localField: '_id'
});

// Domcument Middlewear: runs before .save() and .create()
productSchema.pre('save', function(next){
    this.slug = slugify(this.name, { lower: true });
    next();
});
```

```
// Query Middleware
// productSchema.pre('find', function(next) {
productSchema.pre(/^find/, function(next) {
    this.find({ secretProduct: { $ne: true}});

    this.start = Date.now();
    next();
});



productSchema.post(/^find/, function(docs, next) {
    console.log(`Query took ${Date.now() - this.start}  milliseconds ! `);
    console.log(docs);
    next();
});
```

Εισαγωγή των Modules

- mongoose: Βιβλιοθήκη που επιτρέπει τη διαχείριση εγγράφων MongoDB ως αντικείμενα JavaScript.
- slugify: Δημιουργεί "φιλικά" URLs από το όνομα του προϊόντο.
- validator: Παρέχει έτοιμες συναρτήσεις για έλεγχο τιμών (π.χ., alphanumeric validation).

Δημιουργία του Product Schema

- Το name είναι απαραίτητο (required: true) και πρέπει να είναι μοναδικό (unique: true).
- Χρησιμοποιείται **validation** για μήκος (minlength, maxlength).
- Το trim: true αφαιρεί τα κενά στις άκρες του string.

Υποστήριξη SEO με Slug Field

- Το slug είναι ένα URL-friendly string που δημιουργείται από το όνομα του προϊόντος.

Σύστημα Βαθμολογίας (Ratings)

- Τα προϊόντα έχουν **αξιολογήσεις (**ratingsAverage**)** και **αριθμό αξιολογήσεων (**ratingsQuantity**)**.
- Χρησιμοποιείται min και max validation για τον έλεγχο του εύρους βαθμολογίας (1-5).
- Η set συνάρτηση στρογγυλοποιεί τη βαθμολογία στο πλησιέστερο **δεκαδικό ψηφίο**.

Περιγραφή και Εικόνες Προϊόντος

- Το summary είναι υποχρεωτικό και περιέχει μια σύντομη περιγραφή του προϊόντος.
- Το imageCover αναφέρεται στην **κύρια εικόνα** του προϊόντος, ενώ το images είναι ένας πίνακας με επιπλέον εικόνες.

Απόθεμα και Κατάσταση Προϊόντος

- Το countInStock αποθηκεύει τον αριθμό των διαθέσιμων προϊόντων.
- Το isFeatured καθορίζει αν ένα προϊόν εμφανίζεται ως **προβεβλημένο** στη σελίδα.
- Το createdAt αποθηκεύει την ημερομηνία δημιουργίας, αλλά **δεν επιστρέφεται στα API responses (**select: false**)**.

Τιμολόγηση και Εκπτώσεις

- Χρησιμοποιείται **validation** για να διασφαλιστεί ότι η **τιμή προσφοράς (**priceDiscount**) είναι μικρότερη από την κανονική τιμή (**price**)**.

Indexing για Απόδοση Αναζητήσεων

- Δημιουργείται **ευρετήριο (index) για την τιμή και τη βαθμολογία**, ώστε οι αναζητήσεις να είναι ταχύτερες.
- Το slug έχει index για βελτιστοποίηση αναζητήσεων μέσω URLs.

Virtual Populate για Αξιολογήσεις

- Το virtual populate συνδέει τις **αξιολογήσεις (reviews) με τα προϊόντα**, χωρίς να αποθηκεύει αναφορές στη βάση δεδομένων.

Middleware Functions

1. Document Middleware (Pre-Save Hook)

- Πριν αποθηκευτεί ένα νέο προϊόν, δημιουργείται αυτόματα ένα **slug** βασισμένο στο όνομά του.

2. Query Middleware για Απόκρυψη Μυστικών Προϊόντων

- Κάθε find query **φιλτράρει τα "secret" προϊόντα** που δεν πρέπει να εμφανίζονται.

3. Post Query Middleware για Μέτρηση Χρόνου Αναζήτησης

- Μετράει **τον χρόνο που απαιτείται για την εκτέλεση της αναζήτησης** στη βάση δεδομένων.

4. Συμπέρασμα

Το productSchema.js αποτελεί τη **βάση της διαχείρισης προϊόντων** στο σύστημα, προσφέροντας **ασφάλεια, απόδοση και επεκτασιμότητα**.

- **Validation & Middleware** εξασφαλίζουν ακρίβεια στα δεδομένα.

- **Indexes & Virtuals** βελτιώνουν την απόδοση των queries.
- **Role-based Filtering** κρύβει συγκεκριμένα προϊόντα από αναζητήσεις.

4.8 Εισαγωγή στο Review Model του E-Commerce Backend

(Ενότητα 4.8)

Το **Review Model** αποτελεί ένα σημαντικό μέρος ενός **e-commerce συστήματος**, επιτρέποντας στους χρήστες να **αξιολογούν και να σχολιάζουν προϊόντα**. Η υλοποίηση χρησιμοποιεί **MongoDB με Mongoose**, παρέχοντας μια αποδοτική και δομημένη μέθοδο αποθήκευσης και ανάκτησης δεδομένων.

Κύριες λειτουργίες του Review Model:

- **Διαχείριση Αξιολογήσεων**: Επιτρέπει στους χρήστες να δημιουργούν, ανακτούν, ενημερώνουν και διαγράφουν κριτικές προϊόντων.
- **Συσχέτιση με Προϊόντα και Χρήστες**: Κάθε κριτική συνδέεται με ένα προϊόν (Product) και έναν χρήστη (User).
- **Επικύρωση (Validation) & Περιορισμοί**: Διασφαλίζει ότι κάθε κριτική περιλαμβάνει ένα σχόλιο (review), μια βαθμολογία (rating) μεταξύ 1-5 και είναι μοναδική ανά χρήστη και προϊόν.
- **Aggregation Middleware**: Υπολογίζει **τον μέσο όρο βαθμολογίας** και **τον αριθμό αξιολογήσεων** κάθε προϊόντος.
- Middleware για Αυτόματη Ενημέρωση: Μετά από κάθε νέα αξιολόγηση, το σύστημα επανυπολογίζει αυτόματα τη συνολική βαθμολογία του προϊόντος.

- Το μοντέλο αυτό έχει σχεδιαστεί για **απόδοση, ευελιξία και ασφάλεια**, διασφαλίζοντας ότι η διαχείριση των αξιολογήσεων γίνεται με **αυτοματοποιημένο και αποδοτικό τρόπο**. Στη συνέχεια, αναλύεται η δομή και οι επιμέρους λειτουργίες του κώδικα.

```
// review  /rating  /createdAt /ref to product /ref to user
const mongoose = require('mongoose');
const Product = require('./productModel');

const reviewSchema = new mongoose.Schema({

    review:{
        type: String,
        required: [true, 'Review can not be empty !']
    },
    rating: {
        type: Number,
        min: 1,
        max: 5
    },
    createdAt: {
        type: Date,
        default: Date.now

    },
    product: {
        type: mongoose.Schema.ObjectId,
        ref: 'Product',
        required: [true, 'Review must belong to a product.']

    },
    user: {
        type: mongoose.Schema.ObjectId,
        ref: 'User',
        required: [true, 'Review must belong to a user.']

    },

},
{
    toJSON: { virtuals: true },
    toObject: { virtuals: true}

});


reviewSchema.index({ product: 1, user: 1 }, { unique: true });

// Populate Parent referencing reviews with product and user
reviewSchema.pre(/^find/, function(next) {

```

```

    this.populate({
          path: 'user',
          select: 'name photo email'
      });

    next();
  });

reviewSchema.statics.calcAverageRatings = async function(productId) {
    const stats = await this.aggregate([
        {
            $match: {product: productId }
        },
        {
            $group: {
                _id: '$product',
                nRating: { $sum: 1 },
                avgRating: { $avg: '$rating' }
            }
        }
    ]);

    // console.log(stats);
    if (stats.length > 0) {
        await Product.findByIdAndUpdate(productId, {
            ratingsQuantity: stats[0].nRating,
            ratingsAverage: stats[0].avgRating
        });
    } else {
        await Product.findByIdAndUpdate(productId, {
            ratingsQuantity: 0,
            ratingsAverage: 4.5
        });
    }


};

reviewSchema.post('save', function(n) {
    // this points to current review

    this.constructor.calcAverageRatings(this.product);


});
```

```
92  });
93
94  // findByIdAndUpdate
95  // findByIdAndDelete
96  reviewSchema.pre(/^findOneAnd/,async function(next) {
97      this.r = await this.findOne();
98      // console.log(this.r);
99      next();
100 });
101
102 reviewSchema.post(/^findOneAnd/,async function(next) {
103     // await this.findOne(); does Not work here, query has already executed
104     await this.r.constructor.calcAverageRatings(this.r.product);
105 });
106
107
108 const Review = mongoose.model('Review', reviewSchema);
109
110 module.exports = Review;
```

Ανάλυση Κώδικα

Για την εισαγωγή των modules:

- mongoose: Χρησιμοποιείται για τη διαχείριση των εγγραφών στη βάση δεδομένων MongoDB.
- Product: Το **review schema** σχετίζεται με το προϊόν (Product), αφού κάθε αξιολόγηση αφορά συγκεκριμένο προϊόν.

Ορισμός του Review Schema

- review: Περιέχει το κείμενο της αξιολόγησης και είναι **απαραίτητο (**required: true**)**.
- rating: Περιέχει τη βαθμολογία του προϊόντος, η οποία περιορίζεται μεταξύ **1 και 5**.
- createdAt: Καταγράφει την ημερομηνία δημιουργίας της αξιολόγησης.
- product: Συσχετίζει την αξιολόγηση με ένα προϊόν, μέσω **ObjectId που αναφέρεται στο Product Model**.
- user: Συσχετίζει την αξιολόγηση με έναν χρήστη, μέσω **ObjectId που αναφέρεται στο User Model**.

- Και τα δύο πεδία είναι **required**, διασφαλίζοντας ότι κάθε αξιολόγηση ανήκει σε έναν συγκεκριμένο χρήστη και προϊόν.

2.3 Indexing για Αποφυγή Διπλών Αξιολογήσεων

- Δημιουργείται **μοναδικός συνδυασμός (product, user)**, ώστε ένας χρήστης να μην μπορεί να αξιολογήσει το ίδιο προϊόν περισσότερες από μία φορές.

Middleware για Αυτόματη Σύνδεση με Χρήστες

- Πριν εκτελεστεί οποιοδήποτε find query, οι αξιολογήσεις **αυτόματα εμπλουτίζονται με τα στοιχεία του χρήστη** (name, photo, email).

Υπολογισμός Μέσου Όρου Βαθμολογιών (Aggregation Middleware)

- Ορίζεται **στατική μέθοδος (**statics.calcAverageRatings**)** για τον υπολογισμό του **μέσου όρου βαθμολογιών** κάθε προϊόντος.
- Το aggregate φιλτράρει τις αξιολογήσεις που ανήκουν σε ένα συγκεκριμένο προϊόν και **υπολογίζει**:
    - **nRating**: Συνολικός αριθμός αξιολογήσεων.
    - **avgRating**: Μέσος όρος βαθμολογιών.

- Αν υπάρχουν αξιολογήσεις, ο **μέσος όρος βαθμολογίας** και ο αριθμός αξιολογήσεων **ενημερώνεται στο Product Model**.
- Αν **δεν υπάρχουν αξιολογήσεις**, το σύστημα **ορίζει προεπιλεγμένη τιμή (4.5)**.

Middleware για Αυτόματη Ενημέρωση Βαθμολογιών

Μετά την Αποθήκευση (post('save')**)**

- Μετά τη δημιουργία μιας αξιολόγησης, εκτελείται **αυτόματα** η calcAverageRatings για ενημέρωση της βαθμολογίας του προϊόντος.

Πριν και Μετά την Ενημέρωση ή Διαγραφή (findOneAndUpdate**,** findOneAndDelete**)**

- Πριν εκτελεστεί findOneAndUpdate ή findOneAndDelete, αποθηκεύεται η τρέχουσα αξιολόγηση (this.r).
- Μετά την ενημέρωση ή διαγραφή μιας αξιολόγησης, **επανυπολογίζεται ο μέσος όρος των αξιολογήσεων** του αντίστοιχου προϊόντος.

Αρχές Σχεδίασης και Βέλτιστες Πρακτικές

Το Review Model ακολουθεί **καλές πρακτικές ανάπτυξης**:

- **Referencing (ObjectID)**: Χρησιμοποιούνται αναφορές (product, user) αντί για ενσωμάτωση δεδομένων.
- **Indexing**: Αποτρέπει τη δημιουργία **διπλών αξιολογήσεων** από τον ίδιο χρήστη για το ίδιο προϊόν.
- **Aggregation Middleware**: Επιτρέπει **αυτόματη ενημέρωση** των βαθμολογιών του προϊόντος.
- **Population Middleware**: Εμπλουτίζει τα δεδομένα με στοιχεία χρήστη **χωρίς επιπλέον queries**.
- **Pre/Post Middleware**: Βελτιστοποιεί την ενημέρωση των δεδομένων και αποτρέπει ασυνεπή στοιχεία.

Συμπέρασμα

Το reviewSchema.js παρέχει έναν **ολοκληρωμένο μηχανισμό διαχείρισης αξιολογήσεων**, βελτιώνοντας τη λειτουργικότητα του **e-commerce συστήματος**.

Εξασφαλίζει συνέπεια μεταξύ προϊόντων και αξιολογήσεων. Βελτιστοποιεί την απόδοση μέσω indexing & population middleware. Υποστηρίζει aggregation για έξυπνη ενημέρωση των προϊόντων.

4.9 Εισαγωγή στο User Model του E-Commerce Backend

(Ενότητα 4.9)

Το **User Model** αποτελεί ένα θεμελιώδες στοιχείο στη διαχείριση των χρηστών σε ένα **backend σύστημα e-commerce**. Είναι υπεύθυνο για την αποθήκευση και διαχείριση δεδομένων των χρηστών, καθώς και για την υλοποίηση κρίσιμων λειτουργιών όπως **authentication, authorization, password encryption και account management**.

Κύριες λειτουργίες του User Model:

- **Διαχείριση προσωπικών δεδομένων**: Αποθηκεύει στοιχεία όπως **όνομα, email, ρόλο και φωτογραφία προφίλ**.
- **Authentication & Security**: Χρησιμοποιεί **κρυπτογράφηση κωδικών με bcrypt** για την προστασία των δεδομένων των χρηστών.
- **Role-Based Access Control (RBAC)**: Καθορίζει διαφορετικά επίπεδα πρόσβασης (guest,user,moderator,admin).
- **Token-based Password Reset**: Παρέχει τη δυνατότητα **επαναφοράς κωδικού** με τη χρήση **κρυπτογραφημένου resettoken**.
- **Middleware για ασφαλή διαχείριση δεδομένων**: Διαθέτει pre και post middleware functions για αυτόματες ενέργειες πριν και μετά από αλλαγές στα δεδομένα των χρηστών.

Το συγκεκριμένο **Mongoose Schema** διασφαλίζει ότι το **σύστημα χρηστών είναι ασφαλές, επεκτάσιμο και λειτουργικό**, καλύπτοντας όλες τις απαραίτητες ανάγκες ενός σύγχρονου e-commerce συστήματος. Στη συνέχεια, αναλύονται οι λειτουργίες του κώδικα και οι τεχνικές που χρησιμοποιούνται.

```
const crypto  = require('crypto');
const mongoose = require('mongoose');
const validator = require('validator');
const bcrypt = require('bcryptjs');
const catchAsync = require('../utils/catchAsync');

const userSchema = new mongoose.Schema({ //pass scema as an object
    name: {
        type: String,
        required: [true, 'Please fill your name !'],
        maxlength: [40, 'A product name must have less or equal than 40 characters'],
        minlength: [4, 'A product name must have more or equal than 4 characters'],
    },
    email: {
        type: String,
        required: [true, 'Please provide your email !'],
        unique: true,
        lowercase: true,
        validate: [validator.isEmail, 'please provide valid email ']
    },
    photo: {
          type: String
    },
    role: {
        type: String,
        enum: [ 'guest','user', 'moderator', 'admin'],
        default: 'user'
    },
    password: {
        type: String,
        required:[true, 'Please provide your password !'],
        minlength: [8, 'Password must be at least 8 characters long'],
        select: false

    },
    passwordConfirm: {
        type: String,
        required: [true, 'Please confirm your password'],
        validate: {
            // This only works on Create and Save!!
            validator: function(el) {
                return el === this.password;
            },
            message: 'Passwords are not the same'
        }
    },
    passwordChangedAt: Date,
    passwordResetToken: String,
    passwordResetExpires: Date,
    active: {
        type: Boolean,
        default: true,
        select: false,
    }
```

```
userSchema.pre('save', async function(next) {

 // Only run this function if password was actually modified
 if (!this.isModified('password')) return next();

 //hash the password with cost of 12
 this.password = await bcrypt.hash(this.password, 12);

 //Delete passwordConfirm field
 this.passwordConfirm = undefined;
 next();
});

userSchema.pre('save', function(next) {
 if (!this.isModified('password') || this.isNew) return next();

 this.passwordChangedAt = Date.now() - 1000;
 next();


});

userSchema.pre(/^find/, function(next) {
 // this points to the current query
 this.find({active: { $ne: false }  });
 next();
});

userSchema.methods.correctPassword = async function(
 candidatePassword,
  userPassword
 ) {
 return await bcrypt.compare(candidatePassword, userPassword);
};


userSchema.methods.changedPasswordAfter = function(JWTTimestamp) {
 if (this.passwordChangedAt) {
    const changedTimestamp = parseInt(
      this.passwordChangedAt.getTime() / 1000,
      10
    );


```

Ανάλυση Κώδικα

```
      // console.log(changedTimestamp, JWTTimestamp);
      return JWTTimestamp < changedTimestamp;
    }

    // False means NOT changed
    return false;
  };

  userSchema.methods.createPasswordResetToken = function () {
    const resetToken = crypto.randomBytes(32).toString('hex');

    this.passwordResetToken = crypto
    .createHash('sha256')
    .update(resetToken)
    .digest('hex');

    console.log({ resetToken }, this.passwordResetToken);

    this.passwordResetExpires = Date.now() + 10 * 60 * 1000; // 10 minutes

    return resetToken;
};
   const User = mongoose.model('User', userSchema);

   module.exports = User;
```

Για την εισαγωγή των modules :

- crypto: Χρησιμοποιείται για τη δημιουργία κρυπτογραφημένων tokens (χρησιμοποιείται για επαναφορά κωδικών).
- mongoose: Επιτρέπει τη σύνδεση και διαχείριση δεδομένων στη MongoDB.
- validator: Παρέχει έτοιμες συναρτήσεις για **validation** των email των χρηστών.
- bcryptjs: Χρησιμοποιείται για **κρυπτογράφηση κωδικών πρόσβασης**.

Ορισμός του User Schema

Ορίζεται το **σχήμα δεδομένων (**Schema**) του χρήστη**, το οποίο περιλαμβάνει βασικά στοιχεία:

Τα βασικά πεδία χρήστη είναι:

- Το **όνομα (**name**)** του χρήστη είναι **υποχρεωτικό** (required: true) και έχει **περιορισμούς μήκους** (minlength: 4, maxlength: 40).
- Το **email** είναι επίσης **υποχρεωτικό**, **μοναδικό** (unique: true), και ελέγχεται με **validator.isEmail** για τη διασφάλιση της εγκυρότητάς του.
- Το **role** καθορίζει τον **τύπο χρήστη** (guest, user, moderator, admin).
- Το προεπιλεγμένο επίπεδο πρόσβασης είναι user.
- Ο **κωδικός πρόσβασης (**password**)** είναι **υποχρεωτικός**, έχει **ελάχιστο μήκος 8 χαρακτήρες** και **δεν επιστρέφεται σε queries (**select: false**)** για λόγους ασφαλείας.
- Το passwordConfirm αποθηκεύεται προσωρινά και χρησιμοποιείται για να διασφαλίσει ότι ο χρήστης **δεν έχει κάνει λάθος στην επιβεβαίωση του κωδικού του**.
- Το passwordChangedAt καταγράφει την ημερομηνία αλλαγής κωδικού.
- Τα passwordResetToken και passwordResetExpires χρησιμοποιούνται για τη **διαδικασία επαναφοράς κωδικού πρόσβασης**.

- Το active ελέγχει αν ο χρήστης έχει **ενεργό λογαριασμό**.

Middleware για Κρυπτογράφηση Κωδικού Πρόσβασης

- **Πριν αποθηκευτεί (**pre('save')**)** ένας νέος χρήστης ή αν τροποποιηθεί ο κωδικός του, ο **κωδικός κρυπτογραφείται με bcrypt** (hashing με μέγεθος 12).
- Το passwordConfirm διαγράφεται **ώστε να μην αποθηκεύεται στη βάση δεδομένων**.

Middleware για Καταγραφή Αλλαγής Κωδικού

- Αν ο κωδικός αλλάξει, ενημερώνεται το passwordChangedAt για να διασφαλιστεί η ακύρωση των **παλιών session tokens** του χρήστη.

Middleware για Απόκρυψη Ανενεργών Χρηστών

- Πριν εκτελεστεί **οποιοδήποτε** find **query**, φιλτράρονται οι **ανενεργοί χρήστες (**active: false**)**.

Συναρτήσεις για Έλεγχο Κωδικού και Επαναφορά

1. **Έλεγχος Αντιστοίχισης Κωδικού:**

Επιστρέφει **true/false** εάν ο κωδικός που εισάγει ο χρήστης (candidatePassword) ταιριάζει με τον κρυπτογραφημένο στη βάση (userPassword).

2. **Έλεγχος Αν ο Κωδικός Έχει Αλλαγή Μετά το Token Authentication**

Αν το passwordChangedAt είναι μεταγενέστερο από το timestamp του JWT token, τότε το token είναι άκυρο.

3. **Δημιουργία Token για Επαναφορά Κωδικού**

Δημιουργεί **ένα μοναδικό token** για τη **διαδικασία επαναφοράς κωδικού**.

Συμπέρασμα

Το userSchema.js παρέχει μια **ασφαλή και επεκτάσιμη δομή διαχείρισης χρηστών**, ενσωματώνοντας **role-based authentication, κρυπτογράφηση κωδικών και reset tokens**.

4.10 Εισαγωγή στο Order Model του E-Commerce Backend

(Ενότητα 4.10)

Παραπάνω υπάρχει ένα **Mongoose schema** για την οντότητα **Order**. Αναλυτικότερα:

1. **Βασική Δομή του Schema**

- Ορίζεται το όνομα του σχήματος (orderSchema) με βάση το mongoose.Schema, περιγράφοντας όλα τα πεδία (columns) που θα υπάρχουν στην συλλογή/πίνακα των παραγγελιών.

- Στη συνέχεια δημιουργούμε το αντίστοιχο μοντέλο Order με το mongoose.model('Order', orderSchema).

2. **Κύρια Πεδία**

- **user**: Συσχετίζει την παραγγελία με έναν συγκεκριμένο χρήστη (User) μέσω του user._id.

```
user: {
  type: mongoose.Schema.Types.ObjectId,
  required: [true, 'Order must belong to a User'],
  ref: 'User',
},
```

Έτσι, το ref: 'User' επιτρέπει να κάνουμε populate δεδομένα χρήστη στην παραγγελία.

- **orderItems**: Πίνακας αντικειμένων που περιγράφουν τα προϊόντα αυτής της παραγγελίας. Κάθε στοιχείο του πίνακα περιέχει:
- **product**: Αναφορά στο Product model (π.χ. το _id από τη συλλογή των προϊόντων).
- **name**, **qty**, **price**, **imageCover** κ.λπ., που αποθηκεύουν πληροφορίες για κάθε προϊόν.
- **shippingAddress**: Στοιχεία αποστολής (διεύθυνση, πόλη, ΤΚ, χώρα).
- **paymentMethod**: Μέθοδος πληρωμής που διάλεξε ο χρήστης (π.χ. PayPal, πιστωτική κάρτα κ.λπ.).
- **paymentResult**: Πληροφορίες που μπορεί να δίνει ο πάροχος πληρωμής (π.χ. PayPal, Stripe). Περιέχει id, status, κ.ά.

3. Υπολογιστικά Πεδία Τιμών

- **itemsPrice**: Το άθροισμα της τιμής των αντικειμένων του καλαθιού.
- **taxPrice**: Φόρος (π.χ. ΦΠΑ) που υπολογίζεται στη συνολική παραγγελία.
- **shippingPrice**: Κόστος αποστολής.
- **totalPrice**: Το σύνολο: itemsPrice + taxPrice + shippingPrice. Όλα έχουν default τιμή 0.0 και σημαίνονται ως required.

4. Κατάσταση Πληρωμής & Παράδοσης

- **isPaid**: Boolean πεδίο που δείχνει αν η παραγγελία έχει εξοφληθεί.
- **paidAt**: Ημερομηνία πληρωμής (συμπληρώνεται όταν ολοκληρωθεί η πληρωμή).
- **isDelivered**: Boolean πεδίο που δείχνει αν η παραγγελία έχει παραδοθεί στον πελάτη.
- **deliveredAt**: Ημερομηνία που επιβεβαιώθηκε η παράδοση της παραγγελίας.

5. Timestamps

- Χάρη στη ρύθμιση timestamps: true στο δεύτερο όρισμα του new mongoose.Schema(...), το Mongoose καταγράφει αυτόματα πότε δημιουργήθηκε το έγγραφο (createdAt) και πότε ενημερώθηκε (updatedAt).

Συνοψίζοντας, το παραπάνω schema μοντελοποιεί μία παραγγελία, συνδέοντάς την με τον χρήστη που την έκανε, τα προϊόντα που περιέχει, τα στοιχεία αποστολής, τη μέθοδο πληρωμής κ.ά. Έτσι, μπορείς εύκολα να διαχειρίζεσαι την πορεία της παραγγελίας (π.χ. πληρωμή, παράδοση) σε μια εφαρμογή e-commerce.

```
const mongoose = require('mongoose');

const orderSchema = new mongoose.Schema(
  {
    user: {
      type: mongoose.Schema.Types.ObjectId,
      required: [true, 'Order must belong to a User'],
      ref: 'User',
    },
    orderItems: [
      {
        product: {
          type: mongoose.Schema.Types.ObjectId,
          ref: 'Product',
          required: [true, 'Order item must have a product ID']
        },
        name: { type: String, required: true },
        qty: { type: Number, required: true },
        price: { type: Number, required: true },
        imageCover: { type: String, required: true },
        // etc...
      }

    ],
    shippingAddress: {
      address: { type: String, required: true },
      city: { type: String, required: true },
      postalCode: { type: String, required: true },
      country: { type: String, required: true },
    },
    paymentMethod: {
      type: String,
      required: true,
    },
    paymentResult: {
      id: { type: String },
      status: { type: String },
      update_time: { type: String },
      email_address: { type: String },
    },
    itemsPrice: {
      type: Number,
      required: true,
      default: 0.0,
    },
    taxPrice: {
      type: Number,
      required: true,
      default: 0.0,
    },
    shippingPrice: {
      type: Number,
      required: true,
      default: 0.0,
    },
    totalPrice: {
      type: Number,
      required: true,
      default: 0.0,
    },
    isPaid: {
      type: Boolean,
      required: true,
      default: false,
    },
    paidAt: { type: Date },
    isDelivered: {
      type: Boolean,
      required: true,
      default: false,
    },
    deliveredAt: { type: Date },
  },
  {
    timestamps: true,
  }
);

const Order = mongoose.model('Order', orderSchema);

module.exports = Order;
```

4.11Αναλυτική Επισκόπηση των Controllers σε ένα Backend Σύστημα

(Ενότητα 4.11)

Οι **controllers** σε ένα backend σύστημα αποτελούν το **ενδιάμεσο επίπεδο** ανάμεσα στις **routes (διαδρομές) και τα models (δεδομένα στη βάση)**. Ο κύριος ρόλος τους είναι να **χειρίζονται τα αιτήματα του χρήστη (requests), να εκτελούν τη σχετική επιχειρησιακή λογική και να επιστρέφουν τις κατάλληλες απαντήσεις (responses)**.

Κατηγορίες Controllers σε ένα E-Commerce Σύστημα

1. **Product Controller - Διαχείριση Προϊόντων**

Εισαγω

```
//const fs = require('fs');
const Product = require('./../models/productModel');
const catchAsync = require('./../utils/catchAsync');
const AppError = require('./../utils/appError');
const factory = require('./../controllers/handlerFactory');



exports.getAllProducts = factory.getAll(Product);
exports.getProduct = factory.getOne(Product, {path: 'reviews' });
exports.createProduct = factory.createOne(Product);
exports.updateProduct = factory.updateOne(Product);
exports.deleteProduct = factory.deleteOne(Product);


exports.getProductStats = catchAsync(async (req, res, next) => {

    const stats = await Product.aggregate([
        {
            $match: {  ratingsAverage: {$gte: 4.5 } }
        },
        {
            $group: {
                _id: { $toUpper: '$difficulty'},
                num: { $sum: 1 },
                numRatings: { $sum: '$ratingsQuantity' },
                avgRating: { $avg: '$ratingsAverage' },
                avgPrice: { $avg: '$price' },
                minPrice: { $min: '$price' },
                maxPrice: { $max: '$price' }

            }
        },
        {
            $sort: { avgPrice: 1 }

        },
    ]);

    res.status(200).json({
        status: 'success',
        data: {
            stats
        }
    });
});

```

- Product είναι το μοντέλο που ορίζεται στο αρχείο productModel.js.

- factory είναι ένα αρχείο/αντικείμενο που περιλαμβάνει γενικευμένες μεθόδους CRUD (create, read, update, delete), ώστε να μην επαναλαμβάνεται κώδικας σε κάθε controller.

- catchAsync είναι μία βοηθητική συνάρτηση (utility) που τυλίγει τις async συναρτήσεις με ένα block try/catch, ώστε να περνάει αυτόματα τυχόν λάθη στην next(err) και να τα διαχειρίζεται ο global error handler.

- AppError είναι μια custom κλάση σφαλμάτων που χρησιμοποιείται για την κατασκευή εξατομικευμένων λαθών.

2. CRUD μέθοδοι

- getAllProducts: Καλεί τη συνάρτηση factory.getAll(Product). Αυτή επιστρέφει όλα τα προϊόντα με δυνατότητες φιλτραρίσματος/ταξινόμησης/σελιδοποίησης.

- getProduct: Χρησιμοποιεί τη συνάρτηση factory.getOne(Product, { path: 'reviews' }), που σημαίνει ότι πέρα από την ανάκτηση ενός συγκεκριμένου προϊόντος, κάνει και populate το πεδίο reviews (εφόσον αυτό ορίζεται στο σχήμα του μοντέλου).

- createProduct: Χρησιμοποιεί τη συνάρτηση factory.createOne(Product) για τη δημιουργία νέου προϊόντος.

- updateProduct: Χρησιμοποιεί τη συνάρτηση factory.updateOne(Product) για την ενημέρωση ενός υπάρχοντος προϊόντος.

- deleteProduct: Χρησιμοποιεί τη συνάρτηση factory.deleteOne(Product) για τη διαγραφή ενός προϊόντος.

3. Η συνάρτηση getProductStats

- Είναι κι αυτή ενσωματομένη μέσα στη catchAsync, ώστε να διαχειρίζεται τυχόν εξαιρέσεις και λάθη.

- Χρησιμοποιεί την Product.aggregate([...]) μέθοδο (Aggregation Framework του MongoDB).

- Το πρώτο στάδιο είναι το $match, όπου φιλτράρουμε (ratingsAverage: { $gte: 4.5 }) όλα τα προϊόντα που έχουν μέσο όρο αξιολόγησης (ratingsAverage) τουλάχιστον 4.5.

```
const Review = require('./../models/reviewModel');
const catchAsync = require('./../utils/catchAsync');
const factory = require('./../controllers/handlerFactory');




exports.setProductUsersIds = (req, res, next) => {
    // Allow nested routes
    if (!req.body.product) req.body.product = req.params.productId;
    if (!req.body.user) req.body.user =req.user.id;
    next();
}

exports.getAllReviews = factory.getAll(Review);
exports.getReview = factory.getOne(Review);
exports.createReview = factory.createOne(Review);
exports.updateReview = factory.updateOne(Review);
exports.deleteReview = factory.deleteOne(Review);
```

oUpper: '$difficulty' }. Αυτό

ο “difficulty”, μετατρέποντάς

ουν σε αυτήν την ομάδα.

ζει τον αριθμό των επιμέρους

ολογίζει το μέσο όρο της

ές όλων των προϊόντων της

  - minPrice: { $min: '$price' } βρίσκει τη χαμηλότερη τιμή.

  - maxPrice: { $max: '$price' } βρίσκει τη μεγαλύτερη τιμή.

  - avgPrice: { $avg: '$price' } υπολογίζει τη μέση τιμή.

- Τέλος, το στάδιο $sort: { avgPrice: 1 } ταξινομεί τα αποτελέσματα με βάση το avgPrice σε αύξουσα σειρά.

- Στο τέλος, αποστέλλονται τα αποτελέσματα (stats) στην απόκριση ως JSON, μαζί με μια βασική δομή { status: 'success', data: { stats } }.

Συνοπτικά, οι συναρτήσεις αυτές καλύπτουν βασικές λειτουργίες CRUD για το μοντέλο “Product” και η getProductStats εκτελεί μια στατιστική ανάλυση σε προϊόντα με τουλάχιστον 4.5 μέσο όρο αξιολόγησης, ομαδοποιημένα με βάση το πεδίο difficulty.

2. Review Controller - Διαχείριση Αξιολογήσεων

1. **Εισαγωγές**

- Review: Το Mongoose μοντέλο των reviews (π.χ. σχόλια/αξιολογήσεις).
- catchAsync: Βοηθητική συνάρτηση που ενσωματώνει async συναρτήσεις σε ένα try/catch block, ώστε να περνά λάθη στον global error handler.
- factory: Αρχείο/αντικείμενο που περιέχει γενικευμένες συναρτήσεις CRUD (create, read, update, delete).

2. **setProductUserIds**

- Αυτή η συνάρτηση χρησιμοποιείται για "nested routes" — δηλαδή, όταν μια διαδρομή είναι " εμφωλευμένη" κάτω από ένα άλλο resource (π.χ. /products/:productId/reviews).
- Ελέγχει αν στο req.body δεν έχει οριστεί το πεδίο product. Αν ισχύει, το γεμίζει αυτόματα με την τιμή req.params.productId που προέρχεται από το URL.
- Παρομοίως, αν στο req.body δεν υπάρχει το πεδίο user, το γεμίζει με req.user.id (άρα το user ID προκύπτει από την διαδικασία αυθεντικοποίησης).
- Τέλος καλεί το next() για να συνεχίσει η ροή στο επόμενο middleware ή controller.

```
const User = require('./../models/userModel');
const catchAsync = require('./../utils/catchAsync');
const AppError = require('./../utils/appError');
const factory = require('./../controllers/handlerFactory');


const filterObj = (obj, ...allowedFields) => {
    const newObj = {};
    Object.keys(obj).forEach(el => {
        if (allowedFields.includes(el)) newObj[el] = obj[el];
    });
    return newObj;
};

exports.getMe = (req, res, next) => {
    req.params.id = req.user.id;
    next();
};

exports.updateMe = catchAsync(async (req, res, next) => {
    // 1) Create error if user POSTs password data
    if (req.body.password || req.body.passwordConfirm) {
        return next(
            new AppError(
                'This route is not for password updates. Please use /updateMyPassword',
                400
            )
        );
    }

    // 2) Filtered oout unwanted fields names that are not allowed to be updated
    const filteredBody = filterObj(req.body, 'name', 'email');

    // 3) Update user document
    const updateUser = await User.findByIdAndUpdate(req.user.id, filteredBody, {
        new: true,
        runValidators: true
    });


    res.status(200).json({
        status: 'success',
        data: {
            user: updateUser
        }

    });
});
```

```
exports.deleteMe = catchAsync(async(req, res, next) => {
    await User.findByIdAndUpdate(req.user.id, { active: false });

    res.status(204).json({
        status: 'success',
        data: null
    });
});


exports.createUser = (req, res) => {
    res.status(500).json({
        status:'error',
        message: 'This route is not defined ! Please use /signup '
    });
};

exports.getUser = factory.getOne(User);
exports.getAllUsers = factory.getAll(User);

//Do Not update password with this
exports.updateUser = factory.updateOne(User);
exports.deleteUser = factory.deleteOne(User);
```

τον τρόπο που συνδέονται τα reviews με άλλα
εκμεταλλεύεται κοινές συναρτήσεις CRUD από
λαμβανόμενο κώδικα.

3. User Controller - Διαχείριση Χρηστών

1. **Φόρτωση απαραίτητων αρχείων και helpers**

```
const User = require('./../models/userModel');
const catchAsync = require('./../utils/catchAsync');
const AppError = require('./../utils/appError');
const factory = require('./../controllers/handlerFactory');
```

- **User**: Το μοντέλο του χρήστη (User) στο MongoDB.
- **catchAsync**: Συνάρτηση που "ενσωματώνει" τις async συναρτήσεις, ώστε αν υπάρξει σφάλμα να περάσει στο next(err) και να αναλάβει ο κεντρικός error handler.
- **AppError**: Μια κλάση σφαλμάτων για δημιουργία custom λαθών.

- **factory**: Αρχείο που περιέχει γενικευμένες (κοινές) συναρτήσεις CRUD (π.χ. getOne, getAll, updateOne, deleteOne).

2. Συνάρτηση filterObj

```
const filterObj = (obj, ...allowedFields) => {
  const newObj = {};
  Object.keys(obj).forEach(el => {
    if (allowedFields.includes(el)) newObj[el] = obj[el];
  });
  return newObj;
};
```

- Δημιουργεί ένα νέο αντικείμενο που περιλαμβάνει **μόνο** τα πεδία που βρίσκονται στη λίστα allowedFields.

- Χρήσιμο για περιπτώσεις που θέλουμε να επιτρέψουμε την ενημέρωση μόνο συγκεκριμένων πεδίων (π.χ. name, email) και όχι παραμέτρων ασφαλείας (π.χ. password).

3. **Middleware** getMe

```
exports.getMe = (req, res, next) => {
  req.params.id = req.user.id;
  next();
};
```

- Αυτή η συνάρτηση ορίζει το req.params.id ίσο με το req.user.id.

- Χρησιμοποιείται για να παίρνουμε τα στοιχεία του **συνδεδεμένου χρήστη** (logged-in user), π.χ. όταν καλούμε endpoint /me.

4. **Συνάρτηση** updateMe

```
exports.updateMe = catchAsync(async (req, res, next) => {
  // 1) Δημιουργεί σφάλμα αν ο χρήστης στέλνει password data
  if (req.body.password || req.body.passwordConfirm) {
    return next(
      new AppError(
        'This route is not for password updates. Please use /updateMyPassword',
```

```
        400
      )
    );
  }

  // 2) Φιλτράρει τα μη επιτρεπόμενα πεδία
  const filteredBody = filterObj(req.body, 'name', 'email');

  // 3) Ενημερώνει το έγγραφο του χρήστη
  const updateUser = await User.findByIdAndUpdate(req.user.id, filteredBody, {
    new: true,         // Επιστροφή του ενημερωμένου εγγράφου αντί για το παλιό
    runValidators: true // Εφαρμογή των validators που ορίζονται στο schema
  });

  // 4) Επιστρέφει τα αποτελέσματα
  res.status(200).json({
    status: 'success',
    data: {
      user: updateUser
    }
  });
});
```

- **1) Έλεγχος password**: Αν κάποιος προσπαθήσει να ενημερώσει κωδικό μέσα από αυτό το route, δείχνει σφάλμα. Η αλλαγή κωδικού πρέπει να γίνεται μέσω του /updateMyPassword.

- **2) Φιλτράρισμα**: Επιτρέπεται να αλλάξουμε μόνο name και email, τίποτε άλλο (π.χ. δεν αλλάζουμε ρόλους, κωδικούς, κλπ.).

- **3) Ενημέρωση**: Γίνεται με findByIdAndUpdate του mongoose.

- **4) Απάντηση**: Επιστρέφει τον ενημερωμένο χρήστη στο response.

5. **Συνάρτηση** deleteMe

```
exports.deleteMe = catchAsync(async (req, res, next) => {
  await User.findByIdAndUpdate(req.user.id, { active: false });

  res.status(204).json({
    status: 'success',
    data: null
  });
});
```

- Ουσιαστικά θέτει την ιδιότητα active σε false αντί να διαγράφει πραγματικά το έγγραφο από τη βάση (soft delete).

- **204**: Status code που σημαίνει “No Content”. Δεν επιστρέφεται κάποιο αντικείμενο, απλώς επιβεβαιώνεται η επιτυχία της ενέργειας.

6. createUser

```
exports.createUser = (req, res) => {
  res.status(500).json({
    status:'error',
    message: 'This route is not defined ! Please use /signup '
  });
};
```

- Σηματοδοτεί ότι αυτή η διαδρομή δεν χρησιμοποιείται για να δημιουργήσει χρήστες και ο χρήστης θα πρέπει να χρησιμοποιήσει αντίστοιχα το /signup.

7. **Χρήση** handlerFactory

```
exports.getUser = factory.getOne(User);
exports.getAllUsers = factory.getAll(User);
exports.updateUser = factory.updateOne(User);
exports.deleteUser = factory.deleteOne(User);
```

Χρησιμοποιούνται οι γενικευμένες συναρτήσεις CRUD από το handlerFactory.

- **getUser**: Επιστρέφει έναν χρήστη (π.χ. /users/:id).

- **getAllUsers**: Επιστρέφει όλους τους χρήστες (π.χ. /users).

- **updateUser / deleteUser**: Ανανεώνει ή διαγράφει χρήστη.

- Σημειώνεται ότι οι συναρτήσεις αυτές **δεν** προορίζονται να ενημερώνουν κωδικούς.

Συνολικά

Το αρχείο αυτό **χειρίζεται** τις βασικές πράξεις πάνω στο User μοντέλο: ανάγνωση, ενημέρωση και "διαγραφή" χρηστών, καθώς και τη λογική φιλτραρίσματος μόνο επιτρεπόμενων πεδίων όταν κάποιος χρήστης κάνει update το δικό του προφίλ.

Επιπλέον, περιλαμβάνει βολικά helper/middleware (π.χ. getMe) και αξιοποιεί τη λογική του handlerFactory για **μη επαναλαμβανόμενο** κώδικα.

Order Controller - Διαχείριση Παραγγελιών

```
const Order = require('../models/orderModel');
const catchAsync = require('../utils/catchAsync');
const AppError = require('../utils/appError');
const factory = require('./handlerFactory');

// @desc    Create new order
// @route   POST /api/orders
// @access  Private
exports.createOrder = catchAsync(async (req, res, next) => {
  // attach the user from the protect middleware
  req.body.user = req.user._id;

  const newOrder = await Order.create(req.body)


  res.status(201).json({
    status: 'success',
    data: newOrder
  });
});


// @desc    Get all orders (Admin only)
// @route   GET /api/v1//orders
// @access  Private/Admin
exports.getAllOrders = catchAsync(async (req, res, next) => {
  const orders = await Order.find().populate('user', 'name email');

  res.status(200).json({
    status: 'success',
    results: orders.length,
    data: {
      data: orders
    }
  });
});


// @desc    Get single order by ID
// @route   GET /api/orders/:id
// @access  Private
exports.getOrderById = factory.getOne(Order, {
  path: 'user',
  select: 'name email'
});

```

```
// @desc    Update order to paid
// @route   PUT /api/orders/:id/pay
// @access  Private
exports.updateOrderToPaid = catchAsync(async (req, res, next) => {
  const order = await Order.findById(req.params.id);

  if (!order) {
    return next(new AppError('Order not found', 404));
  }

  order.isPaid = true;
  order.paidAt = Date.now();
  order.paymentResult = {
    id: req.body.id,
    status: req.body.status,
    update_time: req.body.update_time,
    email_address: req.body.email_address,
  };

  const updatedOrder = await order.save();

  res.status(200).json({
    status: 'success',
    data: updatedOrder,
  });
});
```

```
// @desc    Update order to delivered
// @route   PUT /api/orders/:id/deliver
// @access  Private/Admin
exports.updateOrderToDelivered = catchAsync(async (req, res, next) => {
  const order = await Order.findById(req.params.id);

  if (!order) {
    return next(new AppError('Order not found', 404));
  }

  order.isDelivered = true;
  order.deliveredAt = Date.now();

  const updatedOrder = await order.save();

  res.status(200).json({
    status: 'success',
    data: updatedOrder,
  });
});

// @desc    Get logged-in user orders
// @route   GET /api/orders/myorders
// @access  Private
exports.getMyOrders = catchAsync(async (req, res, next) => {
  const orders = await Order.find({ user: req.user._id });

  // Παράδειγμα ενσωμάτωσης
res.status(200).json({
  status: 'success',
  results: orders.length,
  data: {
    data: orders
  }
});

  });
```

1.Εισαγωγές

```
const Order = require('../models/orderModel');
const catchAsync = require('../utils/catchAsync');
const AppError = require('../utils/appError');
const factory = require('./handlerFactory');
```

- **Order**: Το Mongoose μοντέλο (schema) των παραγγελιών.
- **catchAsync**: Συνάρτηση-βοηθός για χειρισμό async/await, ώστε τα σφάλματα να περνούν αυτόματα στον κεντρικό error handler.
- **AppError**: Custom κλάση σφάλματος, την οποία μπορούμε να χρησιμοποιήσουμε για να στέλνουμε συγκεκριμένα μηνύματα/κωδικούς σφάλματος.
- **handlerFactory**: Αρχείο με γενικές (κοινές) CRUD συναρτήσεις, όπως getOne, getAll, updateOne, deleteOne κ.ο.κ.

2. Δημιουργία νέας παραγγελίας (createOrder)

```
exports.createOrder = catchAsync(async (req, res, next) => {
  // Συνδέουμε τον τρέχοντα χρήστη (req.user) με την παραγγελία
  req.body.user = req.user._id;

  // Δημιουργούμε την παραγγελία
  const newOrder = await Order.create(req.body)

  res.status(201).json({
    status: 'success',
    data: newOrder
  });
});
```

- Το middleware “προστασίας” (protect) που υπάρχει σε άλλο σημείο βάζει στο req.user το αντικείμενο του συνδεδεμένου χρήστη.
- Εδώ **προσθέτουμε** το user._id στο body της παραγγελίας, ώστε η παραγγελία να συνδεθεί με τον συγκεκριμένο χρήστη.
- Δημιουργούμε (με Order.create) ένα νέο έγγραφο Order και απαντάμε με **HTTP 201** (Created).

3. Εμφάνιση όλων των παραγγελιών (getAllOrders) – μόνο για Admin

```
exports.getAllOrders = catchAsync(async (req, res, next) => {
  const orders = await Order.find().populate('user', 'name email');

  res.status(200).json({
    status: 'success',
    results: orders.length,
    data: {
      data: orders
    }
  });
});
```

- Χρησιμοποιείται πιθανότατα από admin χρήστη ή με ρόλο διαχειριστή.
- Αναζητά όλες τις παραγγελίες (Order.find()) και κάνει populate στο πεδίο user, ώστε να εμφανιστούν το name και το email του χρήστη που έκανε την κάθε παραγγελία.
- Επιστρέφει τις παραγγελίες με **HTTP 200** και εμφανίζει και το πλήθος (orders.length).

4. Εμφάνιση μίας συγκεκριμένης παραγγελίας με βάση το ID (getOrderById)

```
exports.getOrderById = factory.getOne(Order, {
  path: 'user',
  select: 'name email'
});
```

- Χρησιμοποιεί τη γενική συνάρτηση getOne από το handlerFactory.
- Παράλληλα κάνει populate στο πεδίο user της παραγγελίας, φέρνοντας πίσω μόνο name και email.
- Διαδρομή: GET /api/v1/orders/:id.

5. Ενημέρωση παραγγελίας ως πληρωμένη (updateOrderToPaid)

```
exports.updateOrderToPaid = catchAsync(async (req, res, next) => {
    const order = await Order.findById(req.params.id);

    if (!order) {
      return next(new AppError('Order not found', 404));
    }

    order.isPaid = true;
    order.paidAt = Date.now();
    order.paymentResult = {
      id: req.body.id,
      status: req.body.status,
      update_time: req.body.update_time,
      email_address: req.body.email_address,
    };

    const updatedOrder = await order.save();

    res.status(200).json({
      status: 'success',
      data: updatedOrder,
    });
  });
```

- Αναζητά την παραγγελία βάσει του req.params.id.
- Αν δεν υπάρχει, πετάει σφάλμα 404 μέσω του AppError.
- Ενημερώνει τα πεδία isPaid, paidAt και καταχωρεί στο paymentResult τα στοιχεία που έρχονται από τον πάροχο πληρωμών (π.χ. PayPal ή Stripe).
- Κάνει .save() για να αποθηκεύσει την αλλαγή και επιστρέφει τη νέα μορφή της παραγγελίας.

6. Ενημέρωση παραγγελίας ως “delivered” (updateOrderToDelivered)

```
exports.updateOrderToDelivered = catchAsync(async (req, res, next) => {
    const order = await Order.findById(req.params.id);

    if (!order) {
      return next(new AppError('Order not found', 404));
    }

    order.isDelivered = true;
    order.deliveredAt = Date.now();

    const updatedOrder = await order.save();
```

```
  res.status(200).json({
    status: 'success',
    data: updatedOrder,
  });
});
```

- Παρομοίως, βρίσκει την παραγγελία και αν δεν υπάρχει, επιστρέφει σφάλμα.
- Θέτει isDelivered = true και αποθηκεύει την παραγγελία.
- Επιστρέφει **200** με το ενημερωμένο έγγραφο.
- Συνήθως προορίζεται για admins ή υπεύθυνους που αλλάζουν την κατάσταση σε "delivered".

7. Εμφάνιση όλων των παραγγελιών του **συνδεδεμένου** χρήστη (getMyOrders)

```
exports.getMyOrders = catchAsync(async (req, res, next) => {
  const orders = await Order.find({ user: req.user._id });

  res.status(200).json({
    status: 'success',
    results: orders.length,
    data: {
      data: orders
    }
  });
});
```

- Φέρνει όλες τις παραγγελίες που έχουν user = req.user._id, δηλαδή ανήκουν στον τρέχοντα συνδεδεμένο χρήστη.
- Επιστρέφει το πλήθος των παραγγελιών του χρήστη και τα ίδια τα αποτελέσματα σε ένα αντικείμενο JSON.

8. Σχολιασμένος κώδικας (άλλες εκδοχές / απλοποιημένες υλοποιήσεις)

Στο τέλος, υπάρχει κώδικας που έχει σχολιαστεί και δείχνει:

- Πώς θα μπορούσαμε να χρησιμοποιήσουμε περισσότερο το handlerFactory για **createOne** (createOrder), **getAll** (getAllOrders) κ.λπ.
- Κάποιες placeholder συναρτήσεις για ενημέρωση πληρωμής ή παραλαβής.

Product Controller - Διαχείριση Προϊόντων

```
//const fs = require('fs');
const Product = require('./../models/productModel');
const catchAsync = require('./../utils/catchAsync');
const AppError = require('./../utils/appError');
const factory = require('./../controllers/handlerFactory');



exports.getAllProducts = factory.getAll(Product);
exports.getProduct = factory.getOne(Product, {path: 'reviews' });
exports.createProduct = factory.createOne(Product);
exports.updateProduct = factory.updateOne(Product);
exports.deleteProduct = factory.deleteOne(Product);


exports.getProductStats = catchAsync(async (req, res, next) => {

    const stats = await Product.aggregate([
        {
            $match: {  ratingsAverage: {$gte: 4.5 } }
        },
        {
            $group: {
                _id: { $toUpper: '$difficulty'},
                num: { $sum: 1 },
                numRatings: { $sum: '$ratingsQuantity' },
                avgRating: { $avg: '$ratingsAverage' },
                avgPrice: { $avg: '$price' },
                minPrice: { $min: '$price' },
                maxPrice: { $max: '$price' }

            }
        },
        {
            $sort: { avgPrice: 1 }

        },
    ]);

    res.status(200).json({
        status: 'success',
        data: {
            stats
        }
    });
});
```

1. Εισαγωγές (Imports)

- const Product = require('./../models/productModel'): Φορτώνει το **μοντέλο** Product (Mongoose Schema) από τον φάκελο models.

- const catchAsync = require('./../utils/catchAsync'): Είναι ένα helper function που τυλίγει (wrap) τις ασύγχρονες συναρτήσεις (async/await) για να παγιδεύει (catch) τυχόν **errors** και να τα προωθεί στον global error handler.

- const AppError = require('./../utils/appError'): Είναι μια προσαρμοσμένη (custom) κλάση σφάλματος που πιθανόν χρησιμοποιείται για χειρισμό συγκεκριμένων λαθών.

- const factory = require('./../controllers/handlerFactory'): Φορτώνει ένα σύνολο «εργοστασιακών» συναρτήσεων (handler factory functions) οι οποίες υλοποιούν βασικές λειτουργίες CRUD (Create, Read, Update, Delete) για οποιοδήποτε Mongoose μοντέλο.

2. Χρήση Handler Factory για CRUD

Χρησιμοποιώντας το **factory** module, αποφεύγετε να γράφετε επαναλαμβανόμενο κώδικα για τις CRUD πράξεις. Κάθε συνάρτηση είναι «τυποποιημένη» για το μοντέλο που δηλώνετε:

1. exports.getAllProducts = factory.getAll(Product);

    - Επιστρέφει όλες τις εγγραφές του μοντέλου **Product** από τη βάση.

    - Συνήθως υποστηρίζει και διάφορες επιλογές φιλτραρίσματος (query parameters), ταξινόμησης, σελιδοποίησης κ.λπ.

2. exports.getProduct = factory.getOne(Product, { path: 'reviews' });

    - Επιστρέφει **ένα** συγκεκριμένο Product με βάση το id που δίνεται σε ένα route (π.χ. /products/:id).

    - Το { path: 'reviews' } σημαίνει ότι γίνεται **populate** του σχετικού (ή εικονικού) πεδίου reviews στο αποτέλεσμα, ώστε να φέρει τα συνδεδεμένα reviews του προϊόντος.

3. exports.createProduct = factory.createOne(Product);

    - Δημιουργεί ένα νέο Product στη βάση δεδομένων.

4. exports.updateProduct = factory.updateOne(Product);

    - Ενημερώνει τα στοιχεία ενός Product με βάση το id που δίνεται.

5. exports.deleteProduct = factory.deleteOne(Product);

    - Διαγράφει ένα Product με βάση το id.

Όλα αυτά χρησιμοποιούν τις αντίστοιχες γενικές συναρτήσεις που βρίσκονται στο handlerFactory.js.

3. Συνάρτηση getProductStats

Η συνάρτηση αυτή υπολογίζει **στατιστικά** για τα προϊόντα, χρησιμοποιώντας το **MongoDB Aggregation Pipeline**:

```
exports.getProductStats = catchAsync(async (req, res, next) => {
  const stats = await Product.aggregate([
    {
      $match: { ratingsAverage: { $gte: 4.5 } }
    },
    {
      $group: {
        _id: { $toUpper: '$difficulty' },
        num: { $sum: 1 },
        numRatings: { $sum: '$ratingsQuantity' },
        avgRating: { $avg: '$ratingsAverage' },
        avgPrice: { $avg: '$price' },
        minPrice: { $min: '$price' },
        maxPrice: { $max: '$price' }
      }
    },
    {
      $sort: { avgPrice: 1 }
    }
  ]);

  res.status(200).json({
    status: 'success',
    data: {
      stats
    }
  });
});
```

Αναλυτικά:

1. $match

- Φιλτράρει τα **Products** ώστε να πάρουμε μόνο εκείνα με ratingsAverage >= 4.5.

- Επιστρέφει μόνο εκείνα τα έγγραφα (documents) που πληρούν αυτή την προϋπόθεση.

2. $group

```
const stripe = require('stripe')(process.env.STRIPE_SECRET_KEY);
const Product = require('./../models/productModel');
const catchAsync = require('./../utils/catchAsync');
const AppError = require('./../utils/appError');
const factory = require('./../controllers/handlerFactory');



exports.getCheckoutSession = catchAsync(async (req,res,next) => {
    // 1) Get the currently booked products
    const product = await Product.findById(req.params.productId);

    // 2) Create checkout session
    const session = await stripe.checkout.sessions.create({
        payment_method_types: ['card'],
        success_url: `${req.protocol}://${req.get('host')}`,
        cancel_url: `${req.protocol}://${req.get('host')}/product/${product.slug}`,
        customer_email: req.user.email,
        client_reference_id: req.params.productId,
        line_items: [
          {
            quantity: 1,
            price_data: {
              currency: 'eur',
              unit_amount: product.price * 100,
              product_data: {
                name: `${product.name} Product`,
                description: product.summary,
                //images: [`/${product.imageCover}`],
              },
            },
          },
        ],
        mode: 'payment',
      });

      //3- Send to client
      res.status(200).json({
        status: 'success',
        session,
      });

});
```

- Ομαδοποιεί τα έγγραφα σύμφωνα με κάποιο **κριτήριο**. Εδώ, το κριτήριο _id είναι {$toUpper: '$difficulty'}, άρα το grouping γίνεται με βάση το πεδίο difficulty μετατραμμένο σε κεφαλαία.

- Για κάθε ομάδα, υπολογίζει συγκεκριμένα στατιστικά:

- num: { $sum: 1 }: Μετράει τον αριθμό των εγγράφων (products) σε κάθε ομάδα.
- numRatings: { $sum: '$ratingsQuantity' }: Αθροίζει το πεδίο ratingsQuantity.
- avgRating: { $avg: '$ratingsAverage' }: Υπολογίζει τον μέσο όρο του ratingsAverage.
- avgPrice: { $avg: '$price' }: Υπολογίζει τον μέσο όρο της τιμής.
- minPrice: { $min: '$price' } και maxPrice: { $max: '$price' }: Επιστρέφουν την ελάχιστη και τη μέγιστη τιμή σε κάθε ομάδα.

3. $sort
    - Ταξινομεί τα αποτελέσματα με βάση το avgPrice, από το μικρότερο στο μεγαλύτερο (1 σημαίνει αύξουσα σειρά).
    - Το αποτέλεσμα αποθηκεύεται στη μεταβλητή stats.
    - Τέλος, στέλνουμε απάντηση (response) στον client με res.status(200).json({ ... }), μαζί με τα στατιστικά που υπολογίστηκαν.

4. Ροή Εργασιών (Workflow)

Συνήθως, αυτός ο controller συνδέεται με κάποιο **Router** (π.χ. productRoutes.js) όπου ορίζονται οι διαδρομές (routes) και οι μέθοδοι:

```
// productRoutes.js

const express = require('express');
const productController = require('../controllers/productController');

const router = express.Router();

// Για όλα τα προϊόντα
```

```
router
  .route('/')
  .get(productController.getAllProducts)   // GET /api/v1/products
  .post(productController.createProduct);  // POST /api/v1/products

// Για στατιστικά προϊόντων
router
  .route('/product-stats')
  .get(productController.getProductStats); // GET /api/v1/products/product-stats

// Για συγκεκριμένο προϊόν (με βάση το :id)
router
  .route('/:id')
  .get(productController.getProduct)       // GET /api/v1/products/:id
  .patch(productController.updateProduct)  // PATCH /api/v1/products/:id
  .delete(productController.deleteProduct);// DELETE /api/v1/products/:id

module.exports = router;
```

Και έπειτα στο βασικό αρχείο της εφαρμογής (app.js), υπάρχει:

```
// app.js
const productRouter = require('./routes/productRoutes');

app.use('/api/v1/products', productRouter);
```

Έτσι, οι κλήσεις στα endpoints (URL endpoints) εκτελούν τις αντίστοιχες συναρτήσεις του controller.

5 Cart Controller - Διαχείριση Καλαθιού

Ανάκτηση προϊόντος

- Χρησιμοποιεί το Product.findById(req.params.productId) για να βρει στη βάση δεδομένων το προϊόν με το συγκεκριμένο id. Αυτό το id προέρχεται από το URL (π.χ. /checkout-session/:productId).

Δημιουργία Stripe η PayPal Checkout Session

- Χρησιμοποιούμε το stripe.checkout.sessions.create({ ... }) για να δημιουργήσουμε μια «συνεδρία πληρωμής» (Checkout Session) στη Stripe η Paypal, αναλογα το process.env.εαν είναι STRIPE_SECRET_KEY η PAYPAL_CLIENT_ID .

- payment_method_types**:** Ορίζει τις μεθόδους πληρωμής που μπορεί να χρησιμοποιήσει ο πελάτης (π.χ. card).

- success_url**:** Το URL στο οποίο θα ανακατευθυνθεί ο χρήστης αν η πληρωμή είναι επιτυχημένη.
- Χρησιμοποιεί req.protocol (http ή https) και req.get('host') (το domain ή IP).
- cancel_url**:** Το URL στο οποίο θα ανακατευθυνθεί αν ο χρήστης ακυρώσει την πληρωμή.
- ¨Επειτα ανακατευθύνει. σε /product/:slug.
- customer_email**:** Προσθέτουμε το email του χρήστη από το req.user.email, εφόσον υπάρχει ένας συνδεδεμένος χρήστης.
- client_reference_id**:** Αποθηκεύουμε το productId ως αναφορά, ώστε αργότερα (π.χ. μέσω webhooks) να γνωρίζουμε ποιο προϊόν πληρώθηκε.
- line_items**:** Περιγράφουμε το/τα προϊόντα που θα αγοραστούν. Εδώ υπάρχει μόνο ένα προϊόν (με quantity: 1).
- price_data**:** Περιέχει πληροφορίες για τη δημιουργία τιμής (price) on-the-fly:
  - currency = 'eur'
  - unit_amount = product.price * 100 για να το μετατρέψουμε σε ευρώ
  - product_data με name και description κ.λπ.
- mode: 'payment'**:** Σημαίνει ότι είναι μια απλή, εφάπαξ πληρωμή για ένα προϊόν (σε αντίθεση με συνδρομή - subscription).

Αποστολή session **στον client**

- Επιστρέφουμε με res.status(200).json({ ... }) ένα αντικείμενο που περιέχει το session.
- Ο client (το frontend) θα πάρει το session.id και θα το χρησιμοποιήσει με το Stripe.js η OrderAction.js για να **μεταβεί** στο Checkout Page της Stripe η της Paypal.

Γιατί είναι χρήσιμο

- Σου επιτρέπει να **ενσωματώσεις πληρωμές** με κάρτα (Visa, Mastercard κτλ.) πολύ εύκολα, χωρίς να χρειαστεί να διαχειρίζεσαι ευαίσθητα δεδομένα πληρωμών στον server σου.

- Η Stripe και η PayPal παρέχει **ολοκληρωμένες σελίδες checkout** (Stripe-hosted η PayPal-hosted), οι οποίες είναι ασφαλείς και φιλικές προς το χρήστη.

- Μπορείς να συνδέσεις **webhooks** ώστε να ενημερώνεσαι αυτόματα όταν μια πληρωμή είναι επιτυχής και να προχωράς σε ενέργειες (π.χ. επιβεβαίωση παραγγελίας, αποστολή email κ.λπ.).

Συνοπτικά, ο κώδικας **δημιουργεί μια ασφαλή συνεδρία πληρωμής** μέσω Stripe η Paypal, ώστε ο χρήστης να πληρώσει online για το προϊόν που επέλεξε. Μετά την επιτυχημένη πληρωμή, ο χρήστης ανακατευθύνεται στο success_url, ενώ αν ακυρώσει, πάει στο cancel_url.

6 Auth Controller-Διαχείριση Ανθεκτικότητας

```
const crypto = require('crypto');
const { promisify } = require ('util');
const jwt = require('jsonwebtoken');
const User = require('./../models/userModel');
const catchAsync = require('./../utils/catchAsync');
const AppError = require('./../utils/appError');
const sendEmail = require('./../utils/email');

const signToken = id => {
    return jwt.sign({ id }, process.env.JWT_SECRET, {
        expiresIn: process.env.JWT_EXPIRES_IN
    });
}

const createSendToken = (user, statusCode, res) => {

    const token = signToken(user._id);
    const cookieOptions = {
        expires: new Date(
            Date.now() + process.env.JWT_COOKIE_EXPIRES_IN * 24 * 60 * 60 * 1000
        ),
        httpOnly: true
    };
    if(process.env.NODE_ENV === 'production') cookieOptions.secure = true;

    res.cookie('jwt', token, cookieOptions);

    //remove password from the output
    user.password = undefined;

    res.status(statusCode).json({
        status: 'success',
        token,
        data: {
            user
        }

    });
}
```

```
exports.signup = catchAsync(async (req, res, next) => {
    const newUser = await User.create({
        name: req.body.name,
        email: req.body.email,
        password: req.body.password,
        passwordConfirm: req.body.passwordConfirm,
        passwordChangedAt: req.body.passwordChangedAt,


    });

    createSendToken(newUser, 201, res);

});

exports.login = catchAsync(async (req, res, next) => {
    const { email, password } = req.body;

    // 1) Check if email and password exist
    if (!email || !password) {
        return next(new AppError('Please provide email and password !', 400));

    }

    // 2) Check if user exist && password is correct
    const user = await User.findOne({ email }).select('+password');


    if (!user || !(await user.correctPassword(password, user.password))) {
    return next(new AppError('Incorrect email or password', 401));
  }
    // console.log(user);

    // 3) If everithing ok, send token to client
    createSendToken(user, 200, res);

});
```

```
exports.protect = catchAsync(async (req, res, next) => {
    // 1) Getting token and check of it's there
    let token;
    if (
        req.headers.authorization &&
        req.headers.authorization.startsWith('Bearer')
    ) {
         token = req.headers.authorization.split(' ')[1];
    }

    if(!token) {
        return next(
            new AppError('You are not logged in ! Please log in to get access.', 401)
        );
    }

    // 2) Verification token
    const decoded = await promisify(jwt.verify)(token, process.env.JWT_SECRET);


    // 3) Check if user still exists
    const currentUser = await User.findById(decoded.id);
    if(!currentUser) {
        return next(
            new AppError(
                'The user belonging to this token does no longer exist.',
                401
            )
        );
    }
    // 4) Check if changed password after the token was issued
    if (currentUser.changedPasswordAfter(decoded.iat)) {
        return next(
            new AppError('User recently changed password! Please log in again.', 401)
        );
    };

    //Grand Access to Protected Route
    req.user = currentUser;
    next();
});
```

```
exports.restrictTo = (...roles) => {
    return (req, res, next) => {
        // roles ['admin', 'moderator']. role='user'
        if(!roles.includes(req.user.role)){
            //  console.log('Roles allowed:', roles);
            // console.log('User role:', req.user.role);
            return next(new AppError('You do not have permission to perform this action', 403))
        }

        next();
    };
};

exports.forgotPassword = catchAsync(async(req, res, next) => {
    // 1) Get user based on POSTed email
    const user = await User.findOne({ email: req.body.email})
    if (!user) {
        return next(new AppError('There is no user with email address.', 404));
    }

    // 2) Generate the random reset token
    const resetToken = user.createPasswordResetToken();
    await user.save({ validateBeforeSave: false });

    // 3) Send it to user's email
    const resetURL = `${req.protocol}://${req.get(
        'host'
    )}/api/v1/users/resetPassword/${resetToken}`;

    const message = `Forgot your password? Submit a PATCH request with your new password and passwordConfirm to: ${resetURL}.\nIf you didn't forget your password, please ignore this
    email !`;

    try {
        await sendEmail({
            email: user.email,
            subject: 'Your password reset token (valid for 10 min)',
            message
        });

        res.status(200).json({
            status:'success',
            message: 'Token sent to email !'
        });
    } catch(err){
        user.passwordResetToken = undefined;
        user.passwordResetExpires = undefined;
        await user.save({ validateBeforeSave: false});

        return next(
            new AppError('There was an error sending the email. Try again later !'),
            500
```

```
exports.resetPassword = catchAsync(async (req, res, next) => {
    // 1) Get  user based on the token
    const hashedToken = crypto
        .createHash('sha256')
        .update(req.params.token)
        .digest('hex');

    const user = await User.findOne({
        passwordResetToken: hashedToken,
        passwordResetExpires: { $gt:Date.now() }
    });

    // 2) If token has not expired, and there is user, set the new password
    if(!user) {
        return next(new AppError('Token is invalid or has expired', 400))
    }

    user.password = req.body.password;
    user.passwordConfirm = req.body.passwordConfirm;
    user.passwordResetToken = undefined;
    user.passwordResetExpires = undefined;
    await user.save();

    // 3) Update changePasswordAt property for the user

    // 4) Log the user in, send JWT
    createSendToken(user, 200, res);

});

exports.updatePassword = catchAsync(async (req, res, next) => {
    // 1) Get user from collection
    const user = await User.findById(req.user.id).select('+password');

    // 2) Check if POSTed current password is correct
    if (!(await user.correctPassword(req.body.passwordCurrent, user.password))) {
        return next(new AppError('Your current password is wrong.', 401))
    }

    // 3) If so, update password
    user.password = req.body.password;
    user.passwordConfirm = req.body.passwordConfirm;
    await user.save();

    // 4) Log user in, send JWT
    createSendToken(user, 200, res);
});
```

Επισκόπηση

1. **Φόρτωση βιβλιοθηκών και μοντέλων:**

- crypto: Βιβλιοθήκη της Node.js που παρέχει κρυπτογραφικές λειτουργίες (π.χ. hashing).

- jwt: Βιβλιοθήκη για τη δημιουργία/επαλήθευση JWT (JSON Web Tokens).
- User: Το Mongoose μοντέλο του χρήστη.
- catchAsync: Λειτουργία που «τυλίγει» ασύγχρονες συναρτήσεις, ώστε να γίνεται αυτόματα handling των σφαλμάτων.
- AppError: Προσαρμοσμένη κλάση σφάλματος.
- sendEmail: Συνάρτηση για την αποστολή email (π.χ. σε περιπτώσεις reset password).

2. **Sign/Verify Token**:

- signToken(id): Δημιουργεί ένα JWT που περιέχει στο payload το id του χρήστη, υπογεγραμμένο με το process.env.JWT_SECRET και με χρόνο λήξης που ορίζεται από το process.env.JWT_EXPIRES_IN.
- createSendToken(user, statusCode, res):
    - Δημιουργεί το token,
    - Ρυθμίζει τα **cookies** (π.χ. httpOnly για να μην είναι προσβάσιμα από client-side scripts, ημερομηνία λήξης κλπ.),
    - Αφαιρεί το password από το user object πριν σταλεί (ασφάλεια),
    - Επιστρέφει την απάντηση στον client (JSON) με status, token και data.

3. **Ορισμός Endpoints**:

- exports.signup: Δημιουργεί νέο χρήστη (ενημερώνει τη βάση) και στη συνέχεια καλεί createSendToken ώστε να σταλεί JWT στον client.
- exports.login: Ελέγχει email/κωδικό, επιβεβαιώνει ότι υπάρχει ο χρήστης και ότι ο κωδικός είναι σωστός, και αν όλα πάνε καλά, στέλνει πίσω JWT.
- exports.protect: **Middleware** που προστατεύει διαδρομές. Πρέπει να κληθεί πριν από τον χειριστή (controller) μιας διαδρομής. Κάνει:
    - Έλεγχο αν υπάρχει token στο Authorization header,

- Επαλήθευση (verify) του token,
- Έλεγχο αν ο χρήστης υπάρχει ακόμα στη βάση (π.χ. αν δεν έχει διαγραφεί),
- Έλεγχο αν ο χρήστης άλλαξε κωδικό μετά τη δημιουργία του token (οπότε το token δεν ισχύει),
- Αν όλα είναι καλά, αποθηκεύει τον χρήστη στο req.user και καλεί next() για να προχωρήσει η ροή.

- exports.restrictTo(...roles): **Middleware** για έλεγχο ρόλων. Π.χ. μπορείς να ορίσεις ότι μόνο admin ή moderator ρόλοι μπορούν να έχουν πρόσβαση σε μια διαδρομή.
- exports.forgotPassword: Επιτρέπει σε έναν χρήστη που ξέχασε τον κωδικό του να δημιουργήσει token επαναφοράς (reset token), το οποίο στέλνεται με email.
- exports.resetPassword: Όταν ο χρήστης πατήσει το λινκ από το email και στείλει νέους κωδικούς, ο server ελέγχει το token, επαναφέρει τον κωδικό, και στέλνει νέο JWT.
- exports.updatePassword: Επιτρέπει σε έναν χρήστη που είναι **ήδη συνδεδεμένος** να αλλάξει τον κωδικό του αφού δώσει τον τρέχοντα (σωστό) κωδικό.

4.12 Αναλυτικά Κομμάτια Κώδικα

(Ενότητα 4.12)

1. signToken και createSendToken

```
const signToken = id => {
  return jwt.sign({ id }, process.env.JWT_SECRET, {
    expiresIn: process.env.JWT_EXPIRES_IN
  });
};

const createSendToken = (user, statusCode, res) => {
  const token = signToken(user._id);

  const cookieOptions = {
    expires: new Date(
      Date.now() + process.env.JWT_COOKIE_EXPIRES_IN * 24 * 60 * 60 * 1000
```

```
    ),
    httpOnly: true
  };

  if (process.env.NODE_ENV === 'production') cookieOptions.secure = true;

  res.cookie('jwt', token, cookieOptions);

  // Αφαίρεση password από το object του χρήστη πριν σταλεί στον client
  user.password = undefined;

  res.status(statusCode).json({
    status: 'success',
    token,
    data: {
      user
    }
  });
};
```

- jwt.sign(): Δημιουργεί ένα υπογεγραμμένο token με payload { id: user._id } χρησιμοποιώντας το JWT_SECRET και ορίζει χρόνο λήξης JWT_EXPIRES_IN.
- Το token αποθηκεύεται ως cookie (όνομα jwt) και στέλνεται μέσω res.cookie.
- httpOnly: Αποτρέπει το JavaScript (στον client) να διαβάσει το cookie. Προστασία από XSS.
- secure = true: Επιτρέπει να σταλεί μόνο σε **HTTPS** συνδέσεις (ενεργοποιείται μόνο σε παραγωγή).
- Επιστρέφει JSON με token και τα στοιχεία του χρήστη (χωρίς τον κωδικό).

2. exports.signup

```
exports.signup = catchAsync(async (req, res, next) => {
  const newUser = await User.create({
    name: req.body.name,
    email: req.body.email,
    password: req.body.password,
    passwordConfirm: req.body.passwordConfirm,
    passwordChangedAt: req.body.passwordChangedAt
  });

  createSendToken(newUser, 201, res);
});
```

- Δημιουργεί ένα νέο χρήστη στη βάση (χρησιμοποιώντας τα πεδία που έρχονται από req.body).

- Καλεί τη createSendToken για να στείλει άμεσα JWT back στον client.

3. exports.login

```
exports.login = catchAsync(async (req, res, next) => {
  const { email, password } = req.body;

  // 1) Έλεγχος αν ο χρήστης έστειλε email και password
  if (!email || !password) {
    return next(new AppError('Please provide email and password!', 400));
  }

  // 2) Αναζήτηση χρήστη στη βάση & έλεγχος κωδικού
  const user = await User.findOne({ email }).select('+password');

  if (!user || !(await user.correctPassword(password, user.password))) {
    return next(new AppError('Incorrect email or password', 401));
  }

  // 3) Αν όλα είναι OK, δημιουργούμε JWT και το στέλνουμε
  createSendToken(user, 200, res);
});
```

- **Βήμα 1**: Έλεγχος για κενά email/password.

- **Βήμα 2**: Φέρνει τον χρήστη από τη βάση, συμπεριλαμβανομένου του πεδίου password (πρέπει να ορίσουμε .select('+password') επειδή στο σχήμα μπορεί να είναι κρυφό). Μετά ελέγχει τη μέθοδο user.correctPassword (που ορίζεται πιθανότατα στο userModel) για το αν ο κωδικός ταιριάζει.

- **Βήμα 3**: Αν όλα είναι καλά, καλεί createSendToken.

4. exports.protect (Middleware Προστασίας)

```
exports.protect = catchAsync(async (req, res, next) => {
  // 1) Έλεγχος αν υπάρχει token
  let token;
  if (
    req.headers.authorization &&
    req.headers.authorization.startsWith('Bearer')
  ) {
    token = req.headers.authorization.split(' ')[1];
```

```
  }

  if (!token) {
    return next(
      new AppError('You are not logged in! Please log in to get access.', 401)
    );
  }

  // 2) Επαλήθευση (verify) του token
  const decoded = await promisify(jwt.verify)(token, process.env.JWT_SECRET);

  // 3) Έλεγχος αν ο χρήστης υπάρχει ακόμα
  const currentUser = await User.findById(decoded.id);
  if (!currentUser) {
    return next(
      new AppError('The user belonging to this token does no longer exist.', 401)
    );
  }

  // 4) Έλεγχος αν ο χρήστης άλλαξε κωδικό μετά την έκδοση του token
  if (currentUser.changedPasswordAfter(decoded.iat)) {
    return next(
      new AppError('User recently changed password! Please log in again.', 401)
    );
  }

  // Αν όλα OK, δίνουμε πρόσβαση
  req.user = currentUser;
  next();
});
```

- **Βήμα 1**: Ψάχνει το token στα headers (Authorization: Bearer <token>).
- **Βήμα 2**: Χρησιμοποιεί jwt.verify (μέσω promisify) για να το επαληθεύσει με το JWT_SECRET.
- **Βήμα 3**: Φέρνει τον χρήστη από τη βάση, ελέγχει αν υπάρχει.
- **Βήμα 4**: Έχει μια custom μέθοδο changedPasswordAfter, που ελέγχει την ιδιότητα passwordChangedAt σε σχέση με την iat (Issued At) του token.
- Αν όλα είναι OK, αποθηκεύει τον χρήστη στο req.user και καλεί next().

5. exports.restrictTo(...roles)

```
exports.restrictTo = (...roles) => {
  return (req, res, next) => {
    // Π.χ. roles = ['admin', 'moderator']
    // role του χρήστη = 'user'
```

```
    if (!roles.includes(req.user.role)) {
      return next(
        new AppError('You do not have permission to perform this action', 403)
      );
    }
    next();
  };
};
```

- Δέχεται ως όρισμα έναν ή περισσότερους ρόλους (π.χ. admin, moderator).
- Αν ο ρόλος του req.user **δεν** είναι μέσα στο roles, επιστρέφει σφάλμα 403 (Forbidden). Αλλιώς, καλεί next().
- Συνδυάζεται με το protect middleware, ώστε να εξασφαλίζουμε πως πρώτα ο χρήστης είναι συνδεδεμένος και μετά ελέγχουμε τα δικαιώματά του.

6. exports.forgotPassword

```
exports.forgotPassword = catchAsync(async (req, res, next) => {
  // 1) Έλεγχος για user με συγκεκριμένο email
  const user = await User.findOne({ email: req.body.email });
  if (!user) {
    return next(new AppError('There is no user with that email address.', 404));
  }

  // 2) Δημιουργία reset token
  const resetToken = user.createPasswordResetToken();
  await user.save({ validateBeforeSave: false });

  // 3) Αποστολή email με οδηγίες επαναφοράς
  const resetURL =
`${req.protocol}://${req.get('host')}/api/v1/users/resetPassword/${resetToken}`;

  const message = `Forgot your password? Submit a PATCH request with your new
password and passwordConfirm to: ${resetURL}.\n
If you didn't forget your password, please ignore this email!`;

  try {
    await sendEmail({
      email: user.email,
      subject: 'Your password reset token (valid for 10 min)',
      message
    });

    res.status(200).json({
      status: 'success',
      message: 'Token sent to email!'
    });
  } catch (err) {
```

```
    // Σε περίπτωση αποτυχίας αποστολής email,
    // διαγράφουμε τα πεδία resetToken & resetExpires και ενημερώνουμε τη βάση
    user.passwordResetToken = undefined;
    user.passwordResetExpires = undefined;
    await user.save({ validateBeforeSave: false });

    return next(
     new AppError('There was an error sending the email. Try again later!', 500)
    );
   }
  });
```

- **Βήμα 1**: Βρίσκουμε τον χρήστη με βάση το email που στάλθηκε στο req.body.

- **Βήμα 2**: Καλεί τη μέθοδο user.createPasswordResetToken() (ορισμένη στο userModel) που δημιουργεί ένα **crypto token** και ορίζει passwordResetToken & passwordResetExpires.

- **Βήμα 3**: Στέλνει email με το λινκ /resetPassword/:resetToken. Ο χρήστης πατάει το λινκ και πάει στο επόμενο βήμα (reset password).

7. exports.resetPassword

```
  exports.resetPassword = catchAsync(async (req, res, next) => {
   // 1) Κάνουμε hash το token που υπάρχει στο URL
   const hashedToken = crypto
    .createHash('sha256')
    .update(req.params.token)
    .digest('hex');

   // 2) Βρίσκουμε τον χρήστη με αυτό το hashed token, και που δεν έχει λήξει
(passwordResetExpires > τώρα)
   const user = await User.findOne({
    passwordResetToken: hashedToken,
    passwordResetExpires: { $gt: Date.now() }
   });

   // 3) Αν δεν υπάρχει user ή έχει λήξει το token, σφάλμα
   if (!user) {
    return next(new AppError('Token is invalid or has expired', 400));
   }

   // 4) Διαφορετικά, ορίζουμε τον νέο κωδικό
   user.password = req.body.password;
   user.passwordConfirm = req.body.passwordConfirm;
   user.passwordResetToken = undefined;
   user.passwordResetExpires = undefined;
   await user.save();
```

```
  // 5) Αυτόματα ενημερώνεται το passwordChangedAt (στο userModel) και στέλνουμε νέο JWT
  createSendToken(user, 200, res);
});
```

- Παίρνει το resetToken από το URL, το κάνει hash και ψάχνει στη βάση το αντίστοιχο user.
- Αν βρεθεί, ενημερώνει τον κωδικό (password, passwordConfirm) και «καθαρίζει» τα passwordResetToken & passwordResetExpires.
- Τέλος, καλεί createSendToken για να συνδέσει αυτόματα τον χρήστη.

8. exports.updatePassword

```
exports.updatePassword = catchAsync(async (req, res, next) => {
  // 1) Παίρνουμε τον τρέχοντα χρήστη από το req.user.id (το έχουμε από το protect middleware)
  const user = await User.findById(req.user.id).select('+password');

  // 2) Ελέγχουμε αν ο currentPassword που έστειλε ταιριάζει
  if (!(await user.correctPassword(req.body.passwordCurrent, user.password))) {
    return next(new AppError('Your current password is wrong.', 401));
  }

  // 3) Αν ταιριάζει, θέτουμε νέο κωδικό και αποθηκεύουμε
  user.password = req.body.password;
  user.passwordConfirm = req.body.passwordConfirm;
  await user.save();

  // 4) Στέλνουμε νέο token
  createSendToken(user, 200, res);
});
```

- Χρησιμοποιείται αφού ο χρήστης είναι ήδη **logged in**.
- Ελέγχουμε τον τρέχοντα (παλιό) κωδικό, εάν είναι σωστός ενημερώνουμε με το νέο κωδικό.
- Στέλνουμε νέο JWT (άρα κάνει ουσιαστικά re-login).

Πώς λειτουργεί στη ροή της εφαρμογής

- **signup**: Καταχώριση και άμεση σύνδεση χρήστη.

- **login**: Έλεγχος email/κωδικού, αποστολή token μέσω cookie + JSON response.
- **Προστασία σε routes** με protect:
    - Ελέγχουμε το token,
    - Επαληθεύουμε τον χρήστη & την ισχύ του token,
    - Καλούμε τον επόμενο handler (π.χ. για ανάγνωση/δημιουργία δεδομένων).
- **restrictTo**: Έλεγχος ρόλου για διαδρομές που είναι μόνο για admin κ.λπ.
- **forgotPassword/resetPassword**: Διαδικασία ανάκτησης κωδικού (send email -> reset link -> ορισμός νέου κωδικού).
- **updatePassword**: Χρήστης που είναι ήδη συνδεδεμένος, αλλά θέλει να αλλάξει κωδικό.

Τεχνικά Χαρακτηριστικά των Controllers

1. **Χρήση async και await για Διαχείριση Αιτημάτων**

Όλες οι λειτουργίες των controllers είναι **ασύγχρονες**, καθώς επικοινωνούν με τη βάση δεδομένων.

Παράδειγμα ασύγχρονης function:

- Η catchAsync βοηθά στη διαχείριση των λαθών.

2. **Χρήση Middleware για Authentication & Authorization**

Οι controllers συνεργάζονται με middleware για να διασφαλίσουν ότι μόνο εξουσιοδοτημένοι χρήστες εκτελούν συγκεκριμένες ενέργειες.

Παράδειγμα προστασίας route:

- Το authController.protect διασφαλίζει ότι ο χρήστης είναι συνδεδεμένος.
- Το authController.restrictTo('admin') επιτρέπει μόνο στους διαχειριστές να δημιουργήσουν προϊόντα.

3. **Χρήση Query Filtering & Sorting**

Οι controllers επιτρέπουν δυναμικό φιλτράρισμα και ταξινόμηση δεδομένων από τη βάση.

Παράδειγμα:

- Το sort('-price') ταξινομεί τα προϊόντα από το πιο ακριβό στο πιο φθηνό.

Συμπέρασμα

Οι **controllers** είναι υπεύθυνοι για τη διαχείριση των αιτημάτων στο backend, διασφαλίζοντας την **ορθή λειτουργία του API**. Παρέχουν **επαλήθευση δεδομένων, επικοινωνία με τη βάση δεδομένων, διαχείριση ασφάλειας και βελτιστοποίηση ερωτημάτων**, καθιστώντας το backend **ευέλικτο, ασφαλές και αποδοτικό**.

4.13 Αναλυτική Επισκόπηση των Utils σε ένα Backend Σύστημα

(Ενότητα 4.13)

Στο backend ενός **e-commerce συστήματος**, τα **utils (utilities)** είναι **βοηθητικά αρχεία** που περιέχουν **επαναχρησιμοποιούμενες συναρτήσεις** για να βελτιώνουν την αποδοτικότητα του κώδικα. Οι **utilities** βοηθούν στην **οργάνωση, διατήρηση και επεκτασιμότητα του backend**, επιτρέποντας στους developers να αποφεύγουν **επανάληψη κώδικα** και να εφαρμόζουν **καθαρές αρχές προγραμματισμού**.

Τι Κάνουν τα Utils

- Απλοποιούν σύνθετες λειτουργίες και τις κάνουν πιο επαναχρησιμοποιήσιμες.
- Διαχωρίζουν βοηθητικές συναρτήσεις από το κύριο business logic του backend.
- Αυξάνουν την αναγνωσιμότητα και συντηρησιμότητα του κώδικα.
- Διαχειρίζονται errors, logs, email services, API responses, security tokens, κρυπτογράφηση και άλλα.

4.14 Κατανόηση και Επεξήγηση της κλάσης APIFeatures για φιλτράρισμα, ταξινόμηση, περιορισμό πεδίων και σελιδοποίηση σε MONGOOSE

(Ενότητα 4.14)

```
class APIFeatures {
  constructor(query, queryString) {
    this.query = query;
    this.queryString = queryString;
  }

  filter() {
    const queryObj = { ...this.queryString };
    const excludedFields = ['page', 'sort', 'limit', 'fields'];
    excludedFields.forEach(el => delete queryObj[el]);

    // 1B) Advanced filtering
    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, match => `$${match}`);

    this.query = this.query.find(JSON.parse(queryStr));

    return this;
  }

  sort() {
    if (this.queryString.sort) {
      console.log(this.queryString.sort)
      const sortBy = this.queryString.sort.split(',').join(' ');
      this.query = this.query.sort(sortBy);
    } else {
      this.query = this.query.sort('-createdAt');
    }

    return this;
  }

  limitFields() {
    if (this.queryString.fields) {
      const fields = this.queryString.fields.split(',').join(' ');
      this.query = this.query.select(fields);
    } else {
      this.query = this.query.select('-__v');
    }

    return this;
  }

  paginate() {
    const page = this.queryString.page * 1 || 1;
    const limit = this.queryString.limit * 1 || 100;
    const skip = (page - 1) * limit;

    this.query = this.query.skip(skip).limit(limit);

    return this;
  }
}
module.exports = APIFeatures;
```

Αναλυτική Επεξήγηση

1. Κατασκευαστής (Constructor)

```
constructor(query, queryString) {
  this.query = query;
  this.queryString = queryString;
}
```

- query: Είναι το **mongoose query** αντικείμενο. Π.χ. κάτι σαν Product.find().

- queryString: Είναι τα **query parameters** από το URL. Για παράδειγμα, αν έχουμε /api/v1/products?sort=price&limit=5, τότε queryString μπορεί να είναι ένα αντικείμενο JS σαν { sort: 'price', limit: '5' }.

Αποθηκεύουμε αυτά τα δύο στην κλάση, ώστε σε κάθε μέθοδο (filter, sort κ.λπ.) να τροποποιούμε το this.query με βάση το this.queryString.

2. filter()

```
filter() {
  const queryObj = { ...this.queryString };
  const excludedFields = ['page', 'sort', 'limit', 'fields'];
  excludedFields.forEach(el => delete queryObj[el]);

  // 1B) Advanced filtering
  let queryStr = JSON.stringify(queryObj);
  queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, match => `$${match}`);

  this.query = this.query.find(JSON.parse(queryStr));
  return this;
}
```

- **Αντιγραφή των query params** στο queryObj, ώστε να μη μεταβληθεί το this.queryString.

- excludedFields: Είναι πεδία που χρησιμοποιούνται για άλλους σκοπούς (π.χ. page, sort, limit, fields) και **δεν** θέλουμε να τα θεωρήσουμε ως φίλτρα στη MongoDB. Για κάθε τέτοιο πεδίο, το διαγράφουμε από το queryObj.

- **Advanced filtering**:

    - **Μετατρέπουμε το queryObj σε συμβολοσειρά (JSON.stringify).**

    - **Κάνουμε replace σε μοτίβα όπως gte, gt, lte, lt προσθέτοντας μπροστά $, ώστε να γίνει σύνταξη MongoDB.**

    - **π.χ. "price[gte]": 100 -> "price": { "$gte": 100 }**

    - **Τέλος, μετατρέπουμε ξανά σε αντικείμενο JS (JSON.parse) και καλούμε .find(...) με αυτά τα κριτήρια πάνω στο this.query.**

    - **Παράδειγμα: Αν ο χρήστης έγραψε ?price[gte]=100, το price[gte] γίνεται price: { $gte: 100 }.**

3. sort()

```
sort() {
  if (this.queryString.sort) {
    const sortBy = this.queryString.sort.split(',').join(' ');
    this.query = this.query.sort(sortBy);
  } else {
    this.query = this.query.sort('-createdAt');
  }
  return this;
}
```

- Ελέγχουμε αν υπάρχει το sort στα query params (π.χ. ?sort=price).
- Αν υπάρχουν πολλαπλά κριτήρια (π.χ. ?sort=price,ratings), τότε χρησιμοποιούμε .split(',').join(' ') για να μετατρέψουμε "price,ratings" σε "price ratings", που είναι αποδεκτή σύνταξη για το .sort() της Mongoose.
- Αν **δεν** έχει οριστεί sort από τον χρήστη, ταξινομούμε με βάση το -createdAt (φθίνουσα ταξινόμηση κατά ημερομηνία δημιουργίας, ώστε να φαίνονται τα πιο πρόσφατα πρώτα).

4. limitFields()

```
limitFields() {
  if (this.queryString.fields) {
    const fields = this.queryString.fields.split(',').join(' ');
    this.query = this.query.select(fields);
  } else {
    this.query = this.query.select('-__v');
  }
  return this;
}
```

- Ο χρήστης μπορεί να ζητήσει συγκεκριμένα πεδία (π.χ. ?fields=name,price), οπότε τα διαχωρίζουμε με κόμμα και τα μετατρέπουμε σε μορφή 'name price'.
- Καλούμε .select(...) στη mongoose query για να επιστρέψουμε μόνο εκείνα τα πεδία.
- Αν δεν οριστεί κάτι, εξ ορισμού εξαιρούμε το __v (το πεδίο που βάζει η MongoDB για εκδόσεις εγγράφων).

5. paginate()

```
paginate() {
  const page = this.queryString.page * 1 || 1;
```

```
  const limit = this.queryString.limit * 1 || 100;
  const skip = (page - 1) * limit;

  this.query = this.query.skip(skip).limit(limit);
  return this;
}
```

- Μετατρέπουμε τα page και limit από string σε αριθμούς (* 1) κι αν δεν υπάρχουν, θέτουμε τις προεπιλογές (page = 1, limit = 100).

- Υπολογίζουμε το skip, δηλαδή πόσα έγγραφα θα **παραλείψουμε** (π.χ. αν είμαστε στη σελίδα 2, skip = (2 - 1) * limit = limit).

- Χρησιμοποιούμε .skip() και .limit() στη mongoose query για να εφαρμόσουμε σελιδοποίηση.

Πώς χρησιμοποιείται

Συνήθως την κλάση αυτή την καλείς μέσα σε κάποιον controller, για παράδειγμα:

```
exports.getAllProducts = async (req, res) => {
  // 1) Δημιουργία instance της APIFeatures
  const features = new APIFeatures(Product.find(), req.query)
    .filter()
    .sort()
    .limitFields()
    .paginate();

  // 2) Εκτέλεση της query
  const products = await features.query;

  // 3) Απάντηση
  res.status(200).json({
    status: 'success',
    results: products.length,
    data: {
      products
    }
  });
};
```

- Product.find(): αρχικό query χωρίς φίλτρα.

- Έπειτα καλείς **διαδοχικά** τις μεθόδους filter(), sort(), limitFields() και paginate() της κλάσης. Κάθε μέθοδος **τροποποιεί** την this.query.

- Τελικά, κάνεις await features.query για να εκτελέσεις τη query και να πάρεις τα δεδομένα.

Συμπέρασμα

Η κλάση APIFeatures είναι ένας **συνηθισμένος σχεδιαστικός τρόπος** (design pattern) που βοηθά:

- Να **φιλτράρεις** δεδομένα βάσει συγκεκριμένων παραμέτρων (π.χ. price[gte]=100).
- Να **ταξινομείς** με βάση ένα ή περισσότερα πεδία (π.χ. ?sort=price,-ratings).
- Να **περιορίζεις** τα πεδία που επιστρέφονται (π.χ. ?fields=name,price).
- Να υλοποιείς **σελιδοποίηση** (π.χ. ?page=2&limit=10).

Μέσω αυτής της κλάσης, ο controller παραμένει λιτός, ενώ η ευθύνη της επεξεργασίας των query params και της διαμόρφωσης της MongoDB query συγκεντρώνεται

4.15 Συνάρτηση Χειρισμού Σφαλμάτων για Ασύγχρονες Express Συναρτήσεις

(Ενότητα 4.15)

Τι κάνει:

- Πρόκειται για μια **Higher-Order Function** (συνάρτηση που επιστρέφει συνάρτηση).
- Δέχεται μια ασύγχρονη συνάρτηση fn (ένα controller ή middleware) και την ενσωματώνει με .catch(next).
- Με τον τρόπο αυτό, οποιοδήποτε **σφάλμα** προκύψει μέσα στην fn θα προωθηθεί στο Express error handler μέσω του next().

Γιατί χρησιμοποιείται

- Στα ασύγχρονα route handlers (με async/await) σε Express, εάν προκύψει λάθος μέσα σε ένα await, πρέπει να το «σπρώξουμε» χειροκίνητα (π.χ. με next(err)), ειδάλλως ο server μπορεί να μην προλάβει να το πιάσει.

- Η τεχνική αυτή (**catchAsync**, **asyncWrapper** κτλ.) **απαλλάσσει** από το να γράφουμε διαρκώς try/catch μπλοκ σε κάθε route handler.

- Παράδειγμα

Αν έχεις ένα controller π.χ.:

```
async function getAllUsers(req, res, next) {
  const users = await User.find();
  res.status(200).json({ status: 'success', data: { users } });
}
```

χωρίς τη συγκεκριμένη τεχνική, θα χρειαζόσουν try/catch μπλοκ για να πιάσεις τα σφάλματα. Με το παραπάνω wrapper μπορείς να κάνεις:

```
js
ΑντιγραφήΕπεξεργασία
const catchAsync = require('./catchAsync'); // το αρχείο με το snippet

app.get('/users', catchAsync(getAllUsers));
```

Έτσι, αν ρίξει σφάλμα η getAllUsers, θα πιαστεί από το .catch(next), και το Express θα το επεξεργαστεί στον κεντρικό error handler.

Οφέλη

- Κρατάει τον κώδικα **καθαρό** και **συνοπτικό**.

- Σε βοηθά να **μην επαναλαμβάνεις** συνεχώς try/catch σε κάθε ασύγχρονο handler.

- Εξασφαλίζει ότι όλα τα σφάλματα θα περάσουν σωστά στο **error-handling middleware** του Express.

Συνοπτικά, το snippet αυτό είναι ένας **helper** για να «τυλίγει» ασύγχρονες συναρτήσεις σε Express, επιτρέποντας την **αυτόματη** προώθηση των σφαλμάτων.

Συμπέρασμα

Τα **utils** αποτελούν **βασικά βοηθητικά εργαλεία** που συμβάλλουν στη **δομή, ασφάλεια και απόδοση** ενός backend συστήματος.

- **Αυτοματοποιούν διαδικασίες όπως error handling, authentication, email sending και passwordencryption**.

- **Βελτιώνουν την αναγνωσιμότητα και αποτρέπουν τον επαναλαμβανόμενο κώδικα**.

- **Επιτρέπουν την εύκολη διαχείριση δεδομένων και βελτιστοποίηση queries στη βάση δεδομένων**.

4.16 Συνδεσμολογία Συναρτήσεων

(Ενότητα 4.16)

Σχετικά με τις σταθερές σε λίστα που εμφανίζονται στο αρχείο (**constants**) όπως: REGISTER_SUCCESS, LOGIN_FAIL, PRODUCT_LIST_REQUEST, κ.λπ.

Αυτές οι σταθερές ονομάζονται action types. Ο ρόλος τους είναι να περιγράφουν μοναδικά τα διάφορα “γεγονότα” ή “ενέργειες” που μπορούν να συμβούν στην εφαρμογή μας. Είναι σαν "ετικέτες" που υποδηλώνουν στο υπόλοιπο σύστημα τι συμβαίνει, ώστε το state (κατάσταση) της εφαρμογής να μπορεί να ενημερωθεί ανάλογα.

Για παράδειγμα:

- REGISTER_SUCCESS: δηλώνει ότι μια προσπάθεια εγγραφής χρήστη ολοκληρώθηκε με επιτυχία.

- PRODUCT_LIST_REQUEST: δηλώνει ότι ξεκινάμε αίτημα για να φορτώσουμε λίστα προϊόντων.

- ADD_TO_CART: δηλώνει ότι προσθέτουμε ένα προϊόν στο καλάθι.

Χωρίς αυτές τις σταθερές, θα μπορούσαμε να χρησιμοποιήσουμε “απλές” συμβολοσειρές (π.χ. "REGISTER_SUCCESS" απευθείας), αλλά η χρήση σταθερών βοηθά στην αποφυγή ορθογραφικών λαθών (typos) και διευκολύνει τη συντήρηση του κώδικα.

Τι είναι τα Actions

Τα Actions στο Redux είναι απλά αντικείμενα (JavaScript objects) που έχουν συνήθως δύο βασικά πεδία:

- **type**: το είδος (type) της ενέργειας. Εδώ χρησιμοποιούμε τις σταθερές μας, όπως REGISTER_SUCCESS ή PRODUCT_LIST_REQUEST.

- **payload** (προαιρετικό): τα δεδομένα που μεταφέρει το action στο reducer (π.χ. στοιχεία χρήστη, λίστα προϊόντων, κ.λπ.).

Ένα παράδειγμα action θα μπορούσε να είναι:

```
{
 type: ADD_TO_CART,
 payload: {
  productId: 123,
  quantity: 2
 }
}
```

Αυτό το action "λέει" στο Redux ότι θέλουμε να προσθέσουμε στο καλάθι το προϊόν με ID 123, σε ποσότητα 2. Τα actions είναι οι μόνες πληροφορίες που "διανέμονται" από την εφαρμογή στο store (δηλαδή στον κεντρικό "αποθηκευτικό χώρο" του Redux) για να αλλάξει η κατάσταση.

Τι κάνουν οι Reducers

Οι ***Reducers*** είναι καθαρές συναρτήσεις (pure functions) που δέχονται δύο ορίσματα:

- Το τρέχον state (δηλαδή την τρέχουσα κατάσταση).

- Το action (δηλαδή την ενέργεια που περιγράφει τι συνέβη).

Με βάση τον τύπο της ενέργειας (action.type) και τα δεδομένα μέσα στο action.payload, ο reducer επιστρέφει ένα ***νέο*** state. Για παράδειγμα:

```
const cartReducer = (state = { cartItems: [] }, action) => {
 switch (action.type) {
  case ADD_TO_CART:
   // Ενημερώνουμε το state με βάση το payload του ADD_TO_CART
   return {
    ...state,
    cartItems: [...state.cartItems, action.payload]
   };
```

```
    case REMOVE_FROM_CART:
      return {
        ...state,
        cartItems: state.cartItems.filter(
          (item) => item.productId !== action.payload.productId
        )
      };

    default:
      return state;
  }
};
```

Σημειώνεται ότι:

- Ο reducer εξετάζει το action.type.
- Αν είναι ADD_TO_CART, προσθέτει το νέο αντικείμενο στο cart (χρησιμοποιώντας το payload).
- Αν είναι REMOVE_FROM_CART, αφαιρεί το αντικείμενο που ζητήθηκε.
- Αν δεν αναγνωρίζει το action.type, απλώς επιστρέφει το προηγούμενο state (default).

Το σημαντικό στην αρχιτεκτονική του Redux είναι ότι **κανείς άλλος** πέρα από τον reducer δεν μπορεί να τροποποιήσει το state. Οι ***actions*** περιγράφουν τι πρέπει να αλλάξει και ο ***reducer*** εφαρμόζει τον "κανόνα" για το πώς θα γίνει αυτή η αλλαγή, επιστρέφοντας πάντα ένα νέο state (ποτέ δεν κάνουμε απευθείας μεταβολή στο υπάρχον state).

<u>Πώς δένουν όλα μαζί</u>

- Ο χρήστης (ή κάποια λειτουργία) προκαλεί μια ενέργεια (π.χ. dispatch({ type: ADD_TO_CART, payload: { productId: 123, quantity: 2 } })).
- Το Redux περνά το action στους reducers.
- Ο αντίστοιχος reducer "βλέπει" το type του action, παίρνει τα απαραίτητα δεδομένα (payload) και υπολογίζει το νέο state.
- Το Redux ενημερώνει το store με το νέο state.
- Η διεπαφή (UI) ξανακάνει render, εμφανίζοντας την ενημερωμένη κατάσταση (π.χ. ένα παραπάνω προϊόν στο καλάθι).

Σύνοψη

- Τα action types είναι σταθερές που δηλώνουν τις διάφορες ενέργειες οι οποίες μπορούν να συμβούν στην εφαρμογή.
- **Τα** actions είναι αντικείμενα που περιγράφουν τι συνέβη ή τι θέλουμε να συμβεί (type και δεδομένα).
- **Οι** reducers είναι υπεύθυνοι να επεξεργαστούν τις ενέργειες και να επιστρέψουν ένα νέο state με βάση το παλιό state και τα δεδομένα του action.

ΚΕΦΑΛΑΙΟ 5 Συμπεράσματα

5.1 Συμπέρασμα για το Frontend

(Ενότητα 5.1)

Το frontend αποτελεί τη διεπαφή χρήστη ενός e-commerce συστήματος, καθώς αποτελεί το κομμάτι που αλληλεπιδρά άμεσα με τους χρήστες. Στη συγκεκριμένη υλοποίηση, χρησιμοποιήθηκε το React.js, ένα frontend framework που προσφέρει component-based αρχιτεκτονική, επιτρέποντας την επαναχρησιμοποίηση και την ευελιξία του κώδικα.

Για τη διασφάλιση ενός μοντέρνου και λειτουργικού σχεδιασμού, εφαρμόστηκαν τεχνικές **responsive design** με χρήση CSS και flexbox/grid, ώστε η εμφάνιση του site να προσαρμόζεται σε διαφορετικά μεγέθη οθονών. Παράλληλα, η διαχείριση της κατάστασης της εφαρμογής γίνεται με **React hooks** και το **Context API**, προσφέροντας αποτελεσματική οργάνωση των δεδομένων και αποτρέποντας περιττές επαναφορτώσεις (re-renders).

Η επικοινωνία με το backend πραγματοποιείται μέσω **RESTful API calls** με χρήση της βιβλιοθήκης Axios, εχωντας ως αποτελεσμα τη γρήγορη και ασφαλή ανταλλαγή δεδομένων. Επιπλέον, με το **lazy loading** και το **code splitting** βελτιώθηκε η απόδοση και μειώθηκε ο χρόνος φόρτωσης της εφαρμογής, ώστε ο χρήστης να απολαμβάνει μια πιο ομαλή εμπειρία. Συνολικά, το frontend προσφέρει μια διαδραστική εμπειρία, με έμφαση την ομαλή χρήση της εφαρμογής.

5.2 Συμπέρασμα για το Backend

(Ενότητα 5.2)

Το backend αποτελεί τη «ραχοκοκαλιά» ενός e-commerce συστήματος, διαχειρίζοντας την αποθήκευση, την ανάκτηση και την ασφάλεια των δεδομένων. Σε αυτή την υλοποίηση, χρησιμοποιήθηκε το **Node.js** με **Express.js** για τη δημιουργία ενός RESTful API. Τα δεδομένα αποθηκεύονται σε **MongoDB**, ενώ μέσω του **Mongoose ORM** είναι δυνατή η εύκολη και ευέλικτη διαχείριση της βάσης NoSQL.

Βασικές λειτουργίες του backend είναι η **πιστοποίηση (authentication)** με JWT, η **εξουσιοδότηση (authorization)** με RBAC, οι πράξεις CRUD, ο έλεγχος και η εγκυρότητα δεδομένων (data validation), καθώς και ο χειρισμός σφαλμάτων (error handling). Επιπλέον, αξιοποιείται η δημιουργία index στη βάση δεδομένων για ταχύτερα queries, ενώ τα **middlewares** παρέχουν έλεγχο πρόσβασης και κρυπτογράφηση κωδικών (encryption) για μέγιστη ασφάλεια.

Με μια **modular** αρχιτεκτονική και έναν δομημένο κώδικα, το backend είναι έτοιμο να επεκταθεί όποτε χρειαστεί, επιτρέποντας την εύκολη ενσωμάτωση τρίτων υπηρεσιών (π.χ. πληρωμές μέσω Stripe ή PayPal). Επιπλέον, χάρη στη χρήση **async** functions και **catch** handlers, τα αιτήματα διαχειρίζονται αποδοτικά, εξασφαλίζοντας αξιοπιστία, ασφάλεια και υψηλή απόδοση.

5.3 Γενικά Συμπέρασμα

(Ενότητα 5.3)

Η δημιουργία ενός e-shop προϊόντων τεχνολογίας, με **Front-End (HTML, CSS, React)** και **Back-End (Node.js, Express, MongoDB**), αντιπροσωπεύει μια σύγχρονη προσέγγιση στην ανάπτυξη διαδικτυακών εφαρμογών. Το Front-End, το πρώτο σημείο επαφής με τον χρήστη, διευκολύνει την προβολή προϊόντων και την αλληλεπίδραση (π.χ. προσθήκη στο καλάθι), ενώ το Back-End, από την άλλη, αναλαμβάνει την διαχείριση αιτημάτων να επικοινωνεί με τη βάση δεδομένων και να επιστρέφει δεδομένα στο Front-End. Η επιτυχής συνεργασία αυτών των δύο τμημάτων διασφαλίζει μια ομαλή εμπειρία αγορών, εξασφαλίζοντας παράλληλα επεκτασιμότητα και αξιοπιστία.

Σε επίπεδο σχεδίασης, η HTML δίνει τη βασική δομή της σελίδας, ενώ η CSS επιτρέπει την αισθητική προσαρμογή (χρώματα, γραμματοσειρές, responsive συμπεριφορά). Η React αναλαμβάνει να κάνει τη διεπαφή πιο δυναμική και διαδραστική, προσφέροντας reusable components και διαχειρίζοντας αποδοτικά την κατάσταση (state) της εφαρμογής, ιδιαίτερα σε ενέργειες όπως το φιλτράρισμα προϊόντων ή η επικύρωση δεδομένων πριν την αποστολή στον server. Η ταχύτητα απόκρισης και η σαφήνεια του κώδικα αυξάνονται, ενώ ο διαχωρισμός των UI στοιχείων σε αυτόνομα components διευκολύνει τη συντήρηση και την περαιτέρω εξέλιξη του κώδικα.

Στο Back-End, η χρήση του Node.js επιτρέπει στον προγραμματιστή να αξιοποιήσει την JavaScript και στην πλευρά του server, συνδέοντας το front-end περιβάλλον με την ιδια γλώσσα προγραμματισμου. Το Express.js προσφέρει δομή για το routing και τα middlewares (π.χ. έλεγχος ταυτότητας, επεξεργασία αρχείων), ενώ το MongoDB επιτρέπει την αποθήκευση προϊόντων, χρηστών και παραγγελιών, μέσω εγγράφων (documents). Αυτό το σχήμα μπορεί να προσαρμόζεται εύκολα σε πιθανές μελλοντικές αλλαγές, όπως νέα πεδία (π.χ. προωθητικές ενέργειες ή κριτικές πελατών), χωρίς να απαιτούνται περίπλοκες τροποποιήσεις σε σχήματα πινάκων.Τέλος η ασφάλεια ενισχύεται μέσω κρυπτογράφησης

κωδικών και tokens (JWT), ενώ μηχανισμοί ελέγχου όπως το rate-limiting ή τα web application firewalls μειώνουν την πιθανότητα κακόβουλων επιθέσεων.

Στο μέλλον, το σύστημα μπορεί να επεκταθεί με επιπλέον λειτουργίες, όπως **microservices** αρχιτεκτονικού μοντέλου, ώστε διαφορετικές πτυχές του e-shop (διαχείριση αγορών, σύστημα επιβράβευσης, real-time ενημερώσεις) να χωρίζονται σε ανεξάρτητες υπηρεσίες. Επιπλέον, η ενσωμάτωση τεχνολογιών όπως το **GraphQL** για πιο ευέλικτα queries, **caching** (π.χ. Redis) για ταχύτερη πρόσβαση σε κρίσιμα δεδομένα ή ακόμα και **internationalization** για πολυγλωσσική υποστήριξη. Σε επίπεδο DevOps, εργαλεία όπως το **Docker** και pipelines CI/CD μπορούν να κάνουν τη διαδικασία ενημερώσεων και βελτιώσεων πιο ομαλή και αξιόπιστη.

Αξίζει να σημειωθεί ότι η επιλογή JavaScript (React για το Front-End και Node.js για το Back-End) δεν είναι η μοναδική. Άλλα languages και frameworks (π.χ. Python με Django/Flask, PHP με Laravel, Ruby με Rails, Java με Spring Boot) προσφέρουν επίσης ολοκληρωμένες λύσεις. Εντούτοις, η χρήση της JavaScript από την πλευρά του πελάτη και του διακομιστή διευκολύνει τους νέους προγραμματιστές, επιτρέποντάς τους να εξοικειωθούν με μία γλώσσα για ολόκληρη την εφαρμογή.

Συνολικά, η συγκεκριμένη επιλογή τεχνολογιών έγινε με εκπαιδευτικά κριτήρια, καθώς η JavaScript είναι διαδεδομένη και κατάλληλη τόσο για το frontend όσο και για το backend. Έτσι, οποιοσδήποτε μαθαίνει ή βελτιώνει τις δεξιότητές του σε αυτήν, μπορεί εύκολα να σχεδιάσει, να υλοποιήσει και να επεκτείνει μια πλήρη, σύγχρονη διαδικτυακή εφαρμογή, όπως ένα ολοκληρωμένο e-shop.

ΒΙΒΛΙΟΓΡΑΦΙΑ

Διαδικτυακές Πηγές (Udemy)

1. **Udemy (n.d.).** *Course Link 1*. https://www.udemy.com/share/101qYw3@dMMTU5N4NPaK_AKdRz62DOeXHUaCd6WiDOcnnhck2NTBeSsq5tBAE0PsTMqD8OLnfQ==/
2. **Udemy (n.d.).** *Course Link 2*. https://www.udemy.com/share/101Way3@R7X1Ug8vV7WmtoLGXMN9CCGJ74QRRKgt08p0r9tqFWClqrwq-cdn-c25GxdU1_z0rw==/
3. **Udemy (n.d.).**Course Link 3. https://www.udemy.com/course/mern-ecommerce/?couponCode=LETSLEARNNOW
4. **Udemy (n.d.).**Course Link 4. https://www.udemy.com/course/mern-stack-front-to-back/?couponCode=LETSLEARNNOW

Βιβλία

1. **Casciaro, M. & Mammino, L. (2020).** *Node.js Design Patterns (3rd Edition)*. Packt Publishing.
   - Εστιάζει στις αρχές σχεδίασης (design patterns) για εφαρμογές Node.js και πώς αυτές μπορούν να εφαρμοστούν σε πραγματικά project.
2. **Haverbeke, M. (2018).** *Eloquent JavaScript: A Modern Introduction to Programming (3rd Edition)*. No Starch Press.
   - Παρέχει μια εκτενή εισαγωγή στη JavaScript, καλύπτοντας θεμελιώδη αλλά και πιο προχωρημένα χαρακτηριστικά της γλώσσας.
3. **Duckett, J. (2011).** *HTML and CSS: Design and Build Websites*. John Wiley & Sons.
   - Ιδανικό για αρχάριους και όχι μόνο, με έμφαση στην καλή δομή HTML και στο ευδιάκριτο στιλ CSS.
4. **Duckett, J. (2014).** *JavaScript & jQuery: Interactive Front-End Web Development*. John Wiley & Sons.
   - Επικεντρώνεται στο πώς η JavaScript και η βιβλιοθήκη jQuery μπορούν να χρησιμοποιηθούν για τη δημιουργία διαδραστικών στοιχείων στο front-end.
5. **Meyer, E. & Weyl, E. (2017).** *CSS: The Definitive Guide (4th Edition)*. O'Reilly Media.
   - Ένας από τους πιο λεπτομερείς οδηγούς για το CSS, κατάλληλος για όσους θέλουν να εμβαθύνουν.