

Hellenic Mediterranean University

School of Engineering

Electrical and Computer Engineering

Bachelor Thesis

Developing a RESTful API website with secure payments for horse booking - in Epirus, Greece - with Node.js, Express and Mongo DB.

Eleftheria Barka

Academic supervisor: Papadakis Nikolaos

Heraklion 2024

Table of contents

Abstract....4
Περίληψη....5
Chapter 1: Introduction....7
1.1 Background and Context....7
1.2 Thesis Organization....7
Chapter 2: Literature Review....8
2.1 Equestrian Tourism in the Digital Age....8
2.2 Online Horse Booking Systems in Epirus Greece....9
2.3 Frontend Technologies (HTML, CSS, JavaScript, PUG templates)....9
2.4 Backend Technologies (Node.js, Express, npm, nodemon, dotenv, morgan, slugify, MongoDB, Mongoose, NDB Debugger)....11
Chapter 3: System Architecture and Tools....12
3.1 Overview of Pegasus Epirus Equitours....12
3.2 Version Control: Git and Github....12
3.3 Frontend Development....15
3.3.1 PUG Templates....15
3.3.2 CSS Styling....17
3.4.3 JavaScript....18
App.js....20
3.5 Backend Development....21
3.5.1 MVC....21
3.5.2 CRUD....29
3.6 Node.js and Express....34
3.7 Database Management (MongoDB, Mongoose)....34
3.7.1 Mongo DB....35
3.7.2 Mongoose....37
3.7.3 Secure Payment Implementation with Stripe....38
3.8 Development Tools and Environments....39
3.8.1 Visual Studio Code....39
3.8.2 Postman....40
Chapter 4: User Manual....42
4.1 User....48
4.2 Administrator....50
Chapter 5: Summary....54
5.1 Summary of Technologies Used in Pegasus Epirus Equitours....54
Frontend Technologies....54
Backend Technologies....54
Middleware and Dependencies....55
Other Tools....55
5.2 Conclusion....56
References....57

Table of figures

Figure 0 Logo.....1
Figure 3.1 Header 's pug.....16
Figure 3.2 Home page of website.....17
Figure 3.3 Favicon.....18
Figure 3.4 Logo.....18
Figure 3.5 Server's code.....19
Figure 3.6 App.js.....20
Figure 3.7 MongoDB account.....35
Figure 3.8 MongoDB cluster.....36
Figure 3.9 MongoDB schema.....36
Figure 3.10 Visual Studio Code extensions.....40
Figure 3.11 Postman enviroment.....41
Figure 4.1 Pegasus homepage.....43
Figure 4.2 Pegasus photo gallery.....44
Figure 4.3 About us section of the website.....44
Figure 4.4 Facts about Pegasus sections.....45
Figure 4.5 Footer section.....45
Figure 4.6 Pegasus tours page.....46
Figure 4.7 Tour's details page.....46
Figure 4.8 About Tzoumerka tour.....47
Figure 4.9 Tzoumerka photos.....47
Figure 4.10 Tzoumerka reviews.....48
Figure 4.11 Login page.....48
Figure 4.12 My reviews.....49
Figure 4.13 Edit my reviews.....49
Figure 4.14 Leave a review.....49
Figure 4.15 Stripe payment page.....50
Figure 4.16 Administrator page of all reviews.....50
Figure 4.17 Administrator edits a user page.....51
Figure 4.18 Administrator edits a tour page.....51
Figure 4.19 Administrator edits a review page.....52
Figure 4.20 Administrator and all users page.....53
Figure 4.21 Administrator and all bookings page.....53

Abstract

The purpose of this thesis is the utilization of internet technologies and their possibilities in the field of equestrian tourism in Epirus Region in Greece. Through a systematic approach to the development of the "Pegasus Epirus Equitours" website, an overview of various frond and back end technologies is offered. This website is a solution for those looking for horseback riding experiences, but it can also be the chance for an increase in equestrian tourism and proper development of Epirus region in that field.

Pegasus Epirus Equitours is developed on the principles of a RESTful API, which is ideal for data exchange between server and client (user), as it can communicate and retrieve content dynamically. Specifically, resource-based endpoints are identified by a URI, allowing users to interact with various aspects of the site, from browsing available horseback riding tours to securely booking, and paying for a horseback riding tour. Moreover, RESTful communication ensures that each client request contains the necessary information, thereby enhancing system scalability and responsiveness.

The architecture of the website follows the Model-View-Controller (MVC) model. Specifically, the backend architecture was developed using Node.js, Express, MongoDB and Mongoose technologies. Mongoose schemas define the resource structure, while the data is stored in a MongoDB database, ensuring consistency and reliability. The View level (user interface) was built with HTML, CSS, JavaScript and PUG templates in the frontend while user interactions and dynamic content are achieved through JavaScript. The Controller level (application logic) is developed through the Express framework, and the middlewares, routes and sessions to maintain the state of the user that allows the specific framework the creation of a RESTful API. In addition, controllers process requests and call model functions to determine the appropriate user-side view each time.

User features: user authentication, browsing, booking functionality, control system, payment processing, profile customization (update name, photo, email, password), review editing, view bookings and reviews.

Admin features: tours/users/reviews management (edit their fields and update them), see all bookings, profile customization (similar to users). This thesis focuses on creating an efficient website, with user-friendly interfaces, smooth interactions and security methods.

Finally, by addressing the needs of both users and administrators, the *Pegasus Epirus Equitours* website aims to redefine the equestrian tourism experience through the possibilities offered by technology.

Keywords: equestrian tourism, RESTful booking website, horse riding booking tours, horse riding Epirus

Περίληψη

Σκοπός της παρούσας πτυχιακής εργασίας είναι η αξιοποίηση των τεχνολογιών του διαδικτύου και των δυνατοτήτων τους στον τομέα του ιππικού τουρισμού στην Περιφέρεια Ηπείρου, στην Ελλάδα. Συγκεκριμένα, η ανάπτυξη της ιστοσελίδας *Pegasus Epirus Equitours* έγινε μέσω μιας συστηματικής προσέγγισης, ώστε να προσφέρει μια επισκόπηση διαφόρων τεχνολογιών front και back end. Η ιστοσελίδα είναι μια λύση για όσους αναζητούν εμπειρίες ιππασίας, ενώ μπορεί επίσης να αποτελέσει την ευκαιρία για την αύξηση του ιππικού τουρισμού και τη σωστή ανάπτυξη της περιοχής της Ηπείρου σε αυτόν τον τομέα.

Η ιστοσελίδα *Pegasus Epirus Equitours* αναπτύχθηκε με βάση τις αρχές ενός RESTful API, το οποίο είναι ιδανικό για ανταλλαγή δεδομένων μεταξύ διακομιστή και πελάτη (χρήστη), καθώς μπορεί να επικοινωνεί και να ανακτά περιεχόμενο δυναμικά. Συγκεκριμένα, τα τελικά σημεία (endpoints) βασίζονται σε πόρους και προσδιορίζονται από ένα URI, επιτρέποντας στους χρήστες να αλληλεπιδρούν με διάφορες πτυχές του ιστότοπου, από την περιήγηση σε διαθέσιμες εκδρομές με άλογα έως την ασφαλή κράτηση και την πληρωμή για μια εκδρομή ιππασίας. Επιπλέον, η επικοινωνία RESTful διασφαλίζει ότι κάθε αίτημα χρήστη περιέχει τις απαραίτητες πληροφορίες, ενισχύοντας έτσι την επεκτασιμότητα και την ανταπόκριση του συστήματος.

Η αρχιτεκτονική του ιστότοπου ακολουθεί το μοντέλο Model-View-Controller (MVC). Συγκεκριμένα, η αρχιτεκτονική του στο backend αναπτύχθηκε χρησιμοποιώντας τεχνολογίες Node.js, Express, MongoDB και Mongoose. Τα σχήματα Mongoose ορίζουν τη δομή των πόρων, ενώ τα δεδομένα αποθηκεύονται σε μια βάση δεδομένων MongoDB, διασφαλίζοντας συνέπεια και αξιοπιστία. Η διεπαφή στην πλευρά του χρήστη δημιουργήθηκε με πρότυπα HTML, CSS, JavaScript και PUG στο frontend, ενώ οι αλληλεπιδράσεις των χρηστών και το δυναμικό περιεχόμενο επιτυγχάνονται μέσω JavaScript. Η λογική της εφαρμογής αναπτύσσεται μέσω του framework Express και των ενδιάμεσων λογισμικών, των δρομολογίων και των συνεδριών για τη διατήρηση της κατάστασης του χρήστη, που επιτρέπει στο συγκεκριμένο framework τη δημιουργία ενός RESTful API. Επιπλέον, οι ελεγκτές επεξεργάζονται αιτήματα και καλούν συναρτήσεις του μοντέλου με σκοπό να καθορίζουν κάθε φορά την κατάλληλη προβολή από την πλευρά του χρήστη.

Χαρακτηριστικά χρήστη: έλεγχος ταυτότητας χρήστη, περιήγηση, λειτουργικότητα κράτησης, επεξεργασία πληρωμών, επεξεργασία προφίλ (ενημέρωση ονόματος, φωτογραφίας, email, κωδικού πρόσβασης), επεξεργασία κριτικών, προβολή κρατήσεων και κριτικών.

Χαρακτηριστικά διαχειριστή: διαχείριση περιηγήσεων/χρηστών/κριτικών (δυνατότητα επεξεργασίας και ενημέρωσης), προβολή όλων των κρατήσεων, προσαρμογή προφίλ (παρόμοια με του χρήστη).

Τέλος, η παρούσα πτυχιακή εργασία εστιάζει στη δημιουργία μιας φιλικής διεπαφής προς τον χρήστη, με σωστή αλληλεπίδραση μεταξύ χρήστη και server και μεθόδους ασφαλείας. Έτσι, αντιμετωπίζοντας τις ανάγκες τόσο των χρηστών όσο και των διαχειριστών,

ο ιστότοπος *Pegasus Epirus Equitour* στοχεύει στον επαναπροσδιορισμό της εμπειρίας του ιππικού τουρισμού μέσω των δυνατοτήτων που προσφέρει η τεχνολογία.

Λέξεις-κλειδιά: ιππικός τουρισμός, ιστοσελίδα κρατήσεων RESTful, εκδρομές κρατήσεων ιππασίας, ιππασία Ήπειρος

Chapter 1: Introduction

1.1 Background and Context

Equestrian tourism is part of cultural and outdoor nature activities, which has recently experienced rising popularity worldwide. The combination of horseback riding and exploring appeals to adventurers, nature lovers and those looking for a unique travel experience. However, despite its charm, the management and accessibility of equestrian tourism services face challenges in Greece [1].

Specifically, in the picturesque region of Epirus in Greece, known for its rich history, wild landscapes and vibrant equestrian culture, the potential for equestrian tourism is immense. The area offers a variety of trails and historic sites, making it ideal for equestrian adventures. However, the lack of a proper and RESTful website for online booking and secure transactions is an obstacle for both the providers of such services and the enthusiasts of such activities [5].

Thus, the absence of a RESTful website for equestrian tourism in Epirus creates difficulty in the booking process, limiting the potential development of the Epirus region in Greece. Existing systems do not offer user-friendly interfaces, real-time updates and secure payments. This gap reduces the overall experience for users and poses challenges for businesses[6].

The *Pegasus Epirus Equitours* website seeks to address these challenges by providing a solution for users to explore, book and experience horse tours in various landscapes of Epirus, while ensuring secure payment processes.

1.2 Thesis Organization

This thesis is organized into the following chapters, each contributing to understanding of the development, implementation, and impact of the *Pegasus Epirus Equitours* website:

Chapter 2: Literature Review

Chapter 3: System Architecture and Tools

Chapter 4: User Manual

Chapter 5: Summary

The subsequent chapters delve into the theoretical aspects, practical implementations, and results of the *Pegasus Epirus Equitours* project.

Chapter 2: Literature Review

2.1 Equestrian Tourism in the Digital Age

Equestrian tourism has seen a tremendous rise and change in the digital age, leveraging technology to enhance the overall experience for enthusiasts of such activities. The integration of digital tools, such as online platforms and mobile applications, has made it easier to access information about horse tours, trail rides and related services. This section explores the evolving landscape of equestrian tourism, highlighting the role of technology in connecting riders to a variety of equestrian experiences [1].

Equestrian Tourism in the Digital Age refers to the integration of digital technologies and online platforms to enhance and promote it, and involves activities such as horseback riding tours, horse safaris, and other horse-related experiences [5]. In the digital age, technological advancements have significantly impacted various industries, including tourism. Equestrian tourism, traditionally reliant on in-person interactions and local marketing efforts, has seen a transformation with the advent of digital tools and platforms [7].

Firstly, the online booking platforms allow enthusiasts to discover and book equestrian experiences online. Websites and mobile apps can provide comprehensive information about available tours, horse breeds, trail details, and safety measures [5]. Also, the use of virtual reality (VR) or high-quality multimedia content enables potential visitors to virtually experience equestrian tours before making a reservation, so, it can help in marketing and setting realistic expectations for participants[7].

Moreover, social media platforms play a crucial role in promoting equestrian tourism. Equestrian businesses can utilize platforms like Instagram, Facebook, and Twitter to share stunning content, customer testimonials, and updates about upcoming tours, as social media provides a direct engagement with the audience. In addition, mobile applications can provide on-the-go assistance for participants during equestrian tours. These apps may include GPS-based trail maps, safety guidelines, and emergency contact information to enhance the overall experience and ensure the safety of participants. Moreover, digital platforms can offer e-learning modules and educational resources about horse care, riding techniques, and the history of equestrian traditions. This adds an educational component to the tourism experience and can attract a broader audience.

On the other hand, equestrian tourism businesses can leverage data analytics to gain insights into customer preferences, booking patterns, and overall market trends. This data-driven approach allows businesses to tailor their offerings and marketing strategies for better results. Also, the online platforms enable participants to share their experiences through reviews and ratings. Positive reviews can significantly impact the reputation of equestrian tourism businesses, attracting more customers.

The integration of digital tools not only enhances the accessibility and reach of equestrian tourism but also contributes to the overall growth and sustainability of the industry in the modern, technology-driven era [3].

The advent of online booking systems has changed the tourism industry, providing convenience and efficiency for both service providers and consumers. Examining established

models and best practices in online booking systems, this section delves into the key features and functionalities that contribute to a seamless booking experience. Understanding the dynamics of online reservations becomes paramount in the context of equestrian tourism, laying the foundation for the design and implementation of the *Pegasus Epirus Equitours* website [2].

2.2 Online Horse Booking Systems in Epirus Greece

The 3 highest rated providers in 2023 that offer horseback riding tours on the continent based on the TripAdvisor page have been selected and they are in order of best ratingp[34]:

1. Equestrian Park Preveza
2. Via Natura
3. Pony Club Acheron

Equestrian Park Preveza: offers riding lessons, therapeutic riding and horse walks, however, it does not have a website, only a Facebook page, and communication between users and the park is done by phone or through the page.

Via Natura: offers horse tours on its website and has a nice interface, in particular, it provides information about the horse riding tour, its duration and the periods it takes place. However, here again the communication is done by phone or email to book a tour, while there is no possibility for the user to see comments by previous users who visited it. In addition, no log in, or online booking is available through the website.

Pony Club Acheron: offers horse tours on Acheron River. Particularly, its website has a fairly simple interface, specifically, it provides information bout the tour, its duration and the price. However, here again the communication takes place by phone, while there is no possibility for the user to login, book a tour online or see other people's comments.

To summarize, as we can see, none of the above websites offers the possibility of booking an excursion electronically, nor does it allow the user to create an account on the website, securely book a tour online, leave a comment and save their tours, their comments and to process them. *Pegasus Epirus Equitours* tries to cover this gap, where through the selection of the appropriate technologies it will offer a RESTFul navigation experience to the user and may be the beginning for the proper exploitation of the region of Epirus and the increase of alternative tourism.

2.3 Frontend Technologies (HTML, CSS, JavaScript, PUG templates)

Front-end development forms the user experience in the website, influencing user interaction and website's design. Some of the fundamental front-end technologies, include the HTML,

CSS, and JavaScript languages, along with the utilization of PUG templates for dynamic content rendering. The front-end development is very important in developing a visually appealing interface for the *Pegasus Epirus Equitour*s website.

HTML (HyperText Markup Language) is the standard markup language for creating web pages and web applications. It was first introduced by Tim Berners-Lee in 1991 as a means to structure and format documents for the World Wide Web. Basically, HTML provides the basic structure of a web page by using markup tags to define elements such as headings, paragraphs, links, images, and more. In addition, it forms the backbone of web content and works in conjunction with other technologies to create a complete user experience [12].

CSS (Cascading Style Sheets) was introduced in the late 1990s as a way to separate the presentation of a web page from its structure. The separation of style from content allows for more flexible and efficient web design. CSS is used to style HTML elements and control their layout on a web page. It also gives developers the opportunity to define the visual presentation, including aspects such as colors, fonts, spacing, and positioning. By using CSS, web designers can create responsive interfaces [10].

At the same time, JavaScript was developed by Netscape in the mid-1990s, created to add interaction and dynamic behavior to web pages. Over the years, it has evolved into a powerful programming language as it is a client-side scripting language that runs in the browser. In this way, it allows developers to manipulate the Document Object Model (DOM), handle events, create animations, and perform various actions to enhance user interactions on web pages[9]. On the other hand, DOM represents the structure of a document as a tree of objects, where each object corresponds to a part of the document, such as elements, attributes, and text. The DOM provides a way for programs to dynamically update the content, structure, and style of web documents. Specifically, the key aspects of the DOM include tree structure, nodes, hierarchy, traversal and manipulation, events, dynamic updates and cross browser compatibility.

In particular, DOM represents an HTML or XML document as a hierarchical tree structure and each node in the tree corresponds to an element, attribute, or piece of text in the document and the root of the tree is the Document object, representing the entire web page [11]. Also, the elements, attributes, and text within an HTML document are represented as nodes in the DOM tree, for instance some node types include element nodes for HTML elements, attribute nodes for element attributes, and text nodes for text content. Another important part is the hierarchy, as nodes are organized in a parent-child hierarchy, each node may have parent nodes, child nodes, and sibling nodes. So, the parentNode, childNodes, firstChild, and nextSibling properties allow navigation within the DOM tree [9].

Another important aspect, is the traversal and manipulation, as JavaScript is commonly used to interact with the DOM it allows developers to use methods and properties provided by the DOM API to traverse the tree and manipulate its nodes. For example, some examples of manipulation include adding, removing, or modifying elements and attributes dynamically [17]. In addition, events of the DOM provides a mechanism for handling events, such as user interactions (clicks, key presses) or changes in the document (loading, resizing) as event listeners can be attached to specific nodes to respond to events [12]. Also, one of the

main purposes of the DOM is to enable dynamic updates to web documents, so scripts can modify the content and structure of a page in response to user actions or other events. Plus, DOM provides a standardized interface that can be implemented and with the advent of server-side JavaScript (Node.js), it can also be used for server-side scripting [17].

Also, another technology is the Pug templates, formerly known as Jade, which is a template engine for Node.js and browsers, created by TJ Holowaychuk and later renamed to Pug. Pug simplifies the process of writing HTML by using indentation and a concise syntax. Pug is used to generate HTML markup dynamically. It also allows developers to write HTML templates using a more concise and readable syntax. Pug templates are often used in conjunction with Node.js for server-side rendering, providing a streamlined way to generate HTML content dynamically[17].

In summary, HTML, CSS, and JavaScript are the main technologies used to build and design responsive websites. Pug templates, on the other hand, serve as a tool to simplify the use of HTML markup, especially in the context of server-side rendering within Node.js websites. Each of these technologies plays a crucial role in modern web development, of dynamic and user-friendly web applications[23].

2.4 Backend Technologies (Node.js, Express, npm, nodemon, dotenv, morgan, slugify, MongoDB, Mongoose, NDB Debugger)

This subsection delves into back-end development that handles data storage, retrieval, and overall system functionality. Especially, gives the inside of the back-end technologies that were used in the development of the website, such as Node.js for server-side, JavaScript, Express for building the API, npm for package management, MongoDB as the NoSQL database, and Mongoose for schema-based modeling. Also, the inclusion of tools like nodemon, dotenv, morgan, slugify, and the NDB Debugger enhances the development, debugging, and operational aspects of the website[5].

Building on existing research and case studies, this section reviews and analyzes platforms that have ventured into the intersection of technology and equestrian tourism. Examining their successes, challenges, and user feedback provides valuable insights into effective strategies, potential pitfalls, and innovative features. Understanding the lessons learned from previous endeavors in this domain informs the development process and decision-making for the "Pegasus Epirus Equitours" platform.

This comprehensive literature review sets the foundation for the subsequent chapters, informing the design, development, and evaluation of the proposed equestrian tourism platform.

Chapter 3: System Architecture and Tools

3.1 Overview of Pegasus Epirus Equitours

The architecture of *Pegasus Epirus Equitours* is designed to ensure a scalable and secure equestrian tourism website. The system architecture follows the Model-View-Controller (MVC) model, dividing the website into following components:

(a) the *Model* for data management and business logic

(b) the *View* for handling presentation and user interface and

(c) the *Controller* for managing user input and interactions between the Model and View.

3.2 Version Control: Git and Github

Version control in software development offers several benefits to programmers and teams collaborating on a project, as version control systems (VCS) like Git keep track of changes made to files over time. In this way, it offers a record that allows developers to view and revert to previous versions of the code. Also, it helps in understanding how the code base has developt when specific changes were inserted [8].

In a collaborative environment, multiple developers may work on the same code base at the same time and version control controls this collaboration by providing a repository where changes can be synchronized. In this way, developers can work on their branches and merge changes seamlessly, preventing conflicts, allowing developers to create independent lines of development. Moreover, this is useful especially when working on new features, bug fixes, or experimental changes without affecting the main code base as it allows branches can be merged back into the main branch when ready[30].

In addition, it is an integral part of continuous integration and continuous deployment (CI/CD) workflows. Specifically, CI/CD systems can automatically trigger builds, run tests, and deploy applications based on changes pushed to the version control repository, ensuring a streamlined and automated development process[30].

In this thesis both Git and Github was used to made the hole process much more easier as the repository acted as a backup, reducing the risk of data loss in the development process of *Pegasus Epirus Equitours*.

Git and Github

Git

Git is a version control system that allows developers to track changes in source code during software development, meaning that every developer has a local copy of the entire project history. Specifically, it was created by Linus Torvalds in 2005 and has since become one of the most widely used version control systems [30]. More over, Git offers a branching model which is lightweight and efficient that gives programmers the possibility to create branches for new features and bug fixes, without affecting the main code base.

Also, includes a commit history for recording changes made to files over time. Due to this fact, each commit has a unique identifier, a timestamp, and an associated message that describes the changes made [33]. In addition, Git uses a staging area that allows to selectively choose which changes to include in the next commit[8]. Finally, it is an open-source project, offering the possibility for anyone to inspect and contribute to[33].

GitHub

GitHub is a web-based platform built around Git that provides additional collaboration and project management features. It was launched in 2008 and has become a central hub for developers, hosting millions of repositories across various projects[33]. More specific, provides hosting for Git repositories, making it easy for developers to create, clone, and collaborate on projects as the repositories can be public or private, depending on the visibility settings chosen by the project owner [31].

In addition, it offers collaboration features such as pull requests, issues, and discussions. The so-called pull requests allow developers to propose changes and initiate code reviews, while issues help track tasks, enhancements, and bugs [32]. Also, it offers developers the possibility to fork repositories on GitHub, creating their own copy of the project, that could be modified independently, and the changes can be proposed back to the original project through pull requests. Due to that features, cloning allows users to download a copy of a repository to their local machine[33]. Also, it allows developers to automate workflows, run tests, and deploy applications directly from GitHub [31]. More over, it provides security features such as dependency scanning, code scanning, and secret scanning, that help identify and address security vulnerabilities in projects [31] and also a large community where everyone can follow projects, star repositories, and contribute to open-source projects.

In summary, Git and GitHub together provide a powerful ecosystem for version control and project management in software development. Git serves as the underlying version control system, while GitHub enhances the collaboration and social aspects, providing a centralized platform for hosting, sharing, and contributing to code [33]. And that’s the

reason they were chosen to be used in this thesis for better control of the data, files, and history changes of the code.

Implementation

I began by downloading and installing Visual Studio Code from the official website.I followed the installation instructions for my operating system (Windows 11) and set up VS Code on my machine.

Next, I downloaded and installed Git from the official website. During the installation process, I chose the default settings and then to confirm that both VS Code and Git were installed correctly, I opened a terminal and ran the following commands:

bash>

code --version

git --version

Then , I configured my Git username and email by executing the following commands in the terminal, replacing "my name" and "myemail@example.com" (with my actual name and email):

bash>

git config --global user.name "my name"

git config --global user.email "myemail@example.com"

Also, I created a new folder for my project, by opening VS Code (View > Terminal), I initialized a Git repository with the command:

bash>

git init

I logged in my GitHub account at GitHub and clicked on the "+" icon in the top right corner and then I selected “New repository” and a new repository on GitHub was created, then I chose to initialize it with a README file. In my Visual Studio Code, I opened the source control tab (Ctrl + Shift + G) on the sidebar and by clicking on the "Initialize Repository" button, I initialized the repository in VS Code.

After, I committed my initial changes with a commit message by clicking the check mark icon. Also, I copied the HTTPS or SSH URL of my GitHub repository and then in the terminal, I ran the following command, by replacing <repository-url> with the URL I copied:

bash >

git remote add origin <repository-url>

To push my code to GitHub, I ran the following commands in the terminal:

bash>

git push origin main

Finally, with my VS Code project linked to a Git repository on GitHub, I started coding and made changes to my code, staged them, committed them, and pushed them to GitHub as needed through out the development of the *Pegasus Epirus Equitours* website.

3.3 Frontend Development

The frontend development of *Pegasus Epirus Equitours* website focuses on delivering a user-friendly interface using a combination of CSS, Pug templates and JavaScript language.

3.3.1 PUG Templates

Pug is a template engine for Node.js and browsers that simplifies the process of writing HTML [29] and uses indentation to represent the structure of the HTML, making the code more concise and readable. It also eliminates the need for closing tags and relies on indentation to define the hierarchy of elements and minimizes the amount of typing required compared to traditional HTML. The elimination of closing tags, quotes, and other syntactic elements can lead to shorter and more expressive code.

Moreover, Pug templates are compiled into HTML at runtime, allowing to write templates in a more dynamic way, something that can be really beneficial for generating HTML based on data or conditions. At the same time, it allows to embed JavaScript directly into the template for performing logic, iterating over data, or dynamically generation of content within the template itself [29].

In addition, Pug supports the use of mixins and includes, reusable components and layouts that can make the hole process easier, especial when dealing with complex projects.
Additionally, Pug is often used with Express.js, a popular Node.js web application framework, as Express has built-in support for Pug, making it easy to render Pug templates in response to HTTP requests [29].

In the same way, the View component is also structured using PUG templates, ensuring a dynamic and adaptable layout for users exploring horse tours.

_header.pug



Figure 3.1 Header 's code pug template

Specifically:

- Header element: <header class="header" id="main-header"> contains three sections the tours navigation, the logo, and the user navigation.
- Tours navigation section: <nav class="nav nav--tours"> contains two anchor links: "HOME" and "EXPLORE TOURS".
- Logo section: contains an image element with the source "/img/logo.png" and alt text "Pegasus logo."
- User navigation section: <nav class="nav nav--user"> checks if there is a user:
- If there is a user, displays a "Log out" link and a user profile link with an image and the user's name.
- If there is no user, displays "Log in" and "Sign up" links.

The rendering of the template depends on the values of the user object, if there's a user, it will show the user-related navigation, otherwise, it will show the login and signup options. Also, the ${user.photo} and ${user.name} are placeholders for dynamic values coming from the server.

The front end result of header:



Figure 3.2 Website's homepage

3.3.2 CSS Styling

As we know the quality of a style from a user experience perspective involves considering various factors such as readability, usability, and overall visual appeal. Due to that, the chosen font size and line spacing enhancing readability and also the high contrast between text color (#111111) and background color (#fcfbf6) helps it to be easy and clear in readingr. At the same time, the use of a serif font ('Life Savers') can add a touch of personality.

```
body {
font-size: 16px;
line-height: 1.5;
}
```

The color scheme (primary color #575307, secondary color #c3c486) creates a balanced in colors as consistent use of colors and spacing throughout the design provides a cohesive and pleasing design. Also, the layout is well-structured, with appropriate padding and margins, with a navigation menu and buttons having clear style, making it easy for users to understand and interact with the content. In addition, the hover effects on buttons and images add a subtle interactive element. Also, the use of media queries in the CSS create a responsive design that adapts to different screen sizes:

(@media (max-width: 768px) { /* Styles for smaller screens */ }

Additionally, effective use of images and icons (e.g., ion-icons), use of styles, such as colors, fonts, and spacing, give a professional touch to the websites design. Moreover, the "Explore Tours" button is appropriately styled, making it stand out, and the hover effect adds interactivity.

```
.cta-button {
background-color: #575307;
color: #fcfbf6;
```

```
padding: 10px 20px;
border: none;
border-radius: 5px;
}
.cta-button:hover {
background-color: #c3c486;
}
```

Moreover, visual elements, including graphics, favicon and the logo, are created using Canva to enhance brand recognition.

Favicon and logo:



Figure 3.3 Favicon



Figure 3.4 Logo

3.4.3 JavaScript

The decision to use JavaScript on the website is due to the need for dynamic and interactive features and javascript is ideal with its client-side interactivity that offers. Specificaly, JavaScript is commonly used to create interactive and dynamic elements on the client side that include features such as image sliders, form validation, interactive maps, and real-time updates without requiring page reloads[11].

At the same time, JavaScript offer the possibility to enhance the website's UI and can be used to implement animations, transitions, and other visual effects that contribute to a more engaging user interface[8]. On the other hand, it's asynchronous behavior helps for handling operations that don't block the main thread and this is particularly important for tasks like making asynchronous HTTP requests (AJAX), which allow the website to retrieve or send data to a server without refreshing the entire page. In addition, JavaScript is often used for client-side form validation by providing instant feedback to users when they submit forms and reducing the possibility of errors[15]. Also, allow the website to fetch and display real-time

data without reloading the page as it dynamically update content on the page based on user interactions (updating tour, users, reviews and booking data)[11].

To summarize JavaScript can contribute to security measures on the client side. For example, it can be used to sanitize user input, prevent cross-site scripting (XSS) attacks, and enforce security policies. Also, it is supported by all web browsers, ensuring that the interactive features and functionalities work consistently across different platforms [15]. JavaScript acts as the Controller, by dynamic content updates, form validation, and asynchronous communication with the back end.



```
// Load environment variables from config file
dotenv.config({ path: './config.env' });
// Import the Express app
const app = require('./app');

// Database configuration

// Replace the placeholder '<password>' in the database connection string with the actual password
const DB = process.env.DATABASE.replace(
  '<password>',
  process.env.DATABASE_PASSWORD,
);
// Connect to the MongoDB database using Mongoose
mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
    useUnifiedTopology: true,
  })
  .then(() => {
    console.log(`DB connection successfull`);
  });

// Set up and start the Express app

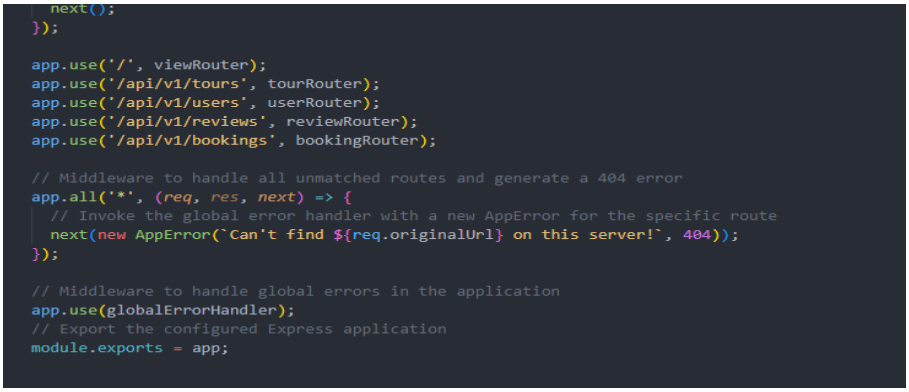
// Define the port for the app to listen on, using the specified port in the environment variables or defaulting to 30
const port = process.env.PORT || 3000;
```

Figure 3.5 Server's code

The server.js sets up a connection to the MongoDB database ‘pegasus’ using Mongoose, loads environment variables from the configuration file, and starts the Express app to run the web server. The script begins by setting up an event listener for the 'uncaughtException' event. If an unhandled exception occurs, it logs the error details and exits the process with a status of 1. Also, the dotenv module is used to load environment variables from the 'config.env' file.

The database connection string is constructed using environment variables, particularly DATABASE and DATABASE_PASSWORD. The script defines the port on which the Express app will listen and checks for the specified port in environment variables or defaults to 3000, as the app is started and listens on the specified port. Finally, a success message is logged once the database connection is established otherwise if an unhandled promise rejection occurs, it logs the error details, closes the server, and exits the process.

App.js



```
  next();
});

app.use('/', viewRouter);
app.use('/api/v1/tours', tourRouter);
app.use('/api/v1/users', userRouter);
app.use('/api/v1/reviews', reviewRouter);
app.use('/api/v1/bookings', bookingRouter);

// Middleware to handle all unmatched routes and generate a 404 error
app.all('*', (req, res, next) => {
  // Invoke the global error handler with a new AppError for the specific route
  next(new AppError(`Can't find ${req.originalUrl} on this server!`, 404));
});

// Middleware to handle global errors in the application
app.use(globalErrorHandler);
// Export the configured Express application
module.exports = app;
```

Figure 3.6 App.js

Various modules are imported, including path, express, morgan (for logging), helmet (for HTTP security headers), express-mongo-sanitize (for sanitizing against NoSQL query injection), xss-clean (for sanitizing against XSS attacks), hpp (for preventing HTTP parameter pollution), and cookie-parser (for parsing cookies).

Data sanitization middlewares (mongoSanitize and xss) are applied to prevent NoSQL query injection and XSS attacks. At the same time, the hpp middleware is used to prevent HTTP parameter pollution by whitelisting specific parameters. The application defines routes using app.use() for various resources (tours, users, reviews, bookings), along with a view router for non-API routes. An error handling middleware (globalErrorHandler) is used to handle errors globally.

3.5 Backend Development

The backend development of *Pegasus Epirus Equitours*, representing the Model and Controller, manages data processing, storage, and communication with external services.

3.5.1 MVC

MVC (Model-View-Controller) architecture [16]:

1. **Models**: define the structure of data (MongoDB schemas).
2. **Controllers**: implement logic and handle HTTP requests.
3. **Views**: handle the rendering of web pages

userModel.js

The userModel.js defines the MongoDB schema and model for the user in a website, using Mongoose:

Schema:

```
const userSchema = new mongoose.Schema({
  // Fields for the user document
});
```

The schema defines the structure of the user document in the MongoDB collection and includes fields such as name, email, photo, role, password, passwordConfirm, passwordChangedAt, passwordResetToken, and passwordResetExpires.

Schema Fields:

- name: user's name.
- email: user's email, which must be unique.
- photo: user's profile photo, with a default value of 'default.png'.
- role: user's role, which must be one of 'user' or 'admin', with a default value of 'user'.
- password: user's password, which is hashed using bcrypt with a cost factor of 12.
- passwordConfirm: used for confirming the password during registration, with validation to ensure it matches the password.

- passwordChangedAt: timestamp indicating when the user's password was last changed.
- passwordResetToken: token generated for resetting the user's password.
- passwordResetExpires: expiration date for the password reset token.
- active: boolean field indicating whether the user account is active.

Specifically, two pre-save middleware functions are defined and run before saving a user document. The first middleware hashes the password and removes the passwordConfirm field. On the other hand, the second middleware updates the passwordChangedAt field if the password is modified. Also, a pre-find middleware is defined using that sets a filter to exclude inactive users from query results.

Instance methods:

- correctPassword: instance method that compares a provided password with the user's hashed password using bcrypt.compare.
- ChangedPasswordAfter: instance method that checks if the user's password was changed after a specified timestamp.

Password reset token creation:instance method that generates a random token, hashes it, and sets the passwordResetToken and passwordResetExpires fields.

Model :

```
const User = mongoose.model('User', userSchema);

module.exports = User;
```

Finally, a Mongoose model named User is created based on the defined schema, and the model is exported for use in other parts of the website.

tourModel.js

Schema:

```
const tourSchema = new mongoose.Schema({
  // Field definitions for the "Tour" document
}
```

The schema defines the structure of the "Tour" document in the MongoDB collection and includes fields such as name, slug, duration, difficulty, ratingsAverage, ratingsQuantity, price, summary, description, imageCover, images, startDates. At the samet time, a pre-save middleware is defined to generate a slug for the tour's name before saving it to the database by using the slugify library.

Model:

```
const Tour = mongoose.model('Tour', tourSchema);

module.exports = Tour;
```

reviewModel.js

```
const reviewSchema = new mongoose.Schema({

}, {

  toJSON: { virtuals: true },

  toObject: { virtuals: true },

});
```

The schema defines the structure of the "Review" document in the MongoDB collection and includes fields such as review, rating, tour (referring to the Tour model), and user (referring to the User model). Specifically, the schema includes fields for the review text, rating (with a minimum of 1 and a maximum of 5), date that it was created, reference to the specific tour, and the user who wrote the review. Also, an index is defined on the tour and user fields with the { unique: true } option to ensure that a user can only submit one review per tour. More over, a static method is used to calculate and update the average ratings for a given tour by aggregating review data. Also, these middleware functions are used to store a reference to the current review (this.r) before and after updating or deleting a review. This reference is then used to recalculate average ratings for the associated tour.

Model:

```
const Review = mongoose.model('Review', reviewSchema);

module.exports = Review;
```

bookingModel.js

Schema:

```
const bookingSchema = new mongoose.Schema({

  // Field definitions for the "Booking" document

});
```

Schema fields: The schema includes fields for the associated tour, user, booking price, creation date, and a flag indicating whether the booking has been paid (paid).

Pre-find Middleware: A pre-find middleware is defined to populate the user field and, in addition, populate the tour field with only the name property.

Model creation:

const Booking = mongoose.model('Booking', bookingSchema);

module.exports = Booking;

Validation: ensuring that a booking must be associated with both a tour and a user, a price is required for the booking.

Default values: default values are provided for the createdAt field (set to the current date/time) and the paid field (set to true by default).

viewRouter

The viewRouter handles routes that render views rather than processing data (which is done by API routes).

Home route: router.get('/', authController.isLoggedIn, viewsController.getAboutUs);

Tour Route: router.get('/tour/:slug', authController.isLoggedIn, viewsController.getTour);

Login Route: router.get('/login', authController.isLoggedIn, viewsController.getLoginForm);

Account Route: router.get('/me', authController.protect, viewsController.getAccount);

Admin-Only Routes:

// Users

router.get('/users', authController.protect, userController.getAllUsers);

router.get('/users/edit/:id', authController.protect, userController.getUserAsAdmin);

router.patch('/api/v1/users/edit/:id',authController.protect,
userController.updateUserAsAdmin);

// Tours

router.get('/tours', authController.protect, tourController.getAllTours);

router.get('/tours/:id', authController.protect, tourController.getTourAsAdmin);

router.patch('/api/v1/tours/:id', authController.protect, tourController.updateTour);

// Reviews

router.get('/reviews', authController.protect, reviewController.getAllReviews);

router.get('/reviews/edit/:id', authController.protect, reviewController.getReviewAsAdmin);

router.patch('/api/v1/reviews/edit/:id',authController.protect,
reviewController.updateReviewAsAdmin);

// Bookings

router.get('/all-bookings', authController.protect, bookingController.getAllBookings);

These routes are restricted to admin users and handle various administrative actions, such as managing users, tours, reviews, and bookings. The viewRouter is responsible for rendering different views in your application. It utilizes controllers (viewsController, userController, tourController, reviewController, bookingController) to handle specific functionalities and interacts with authentication middleware (authController) to ensure proper user authentication and authorization for certain routes.

userRouter

router.post('/signup', authController.signup);

router.post('/login', authController.login);

router.get('/logout', authController.logout);

router.post('/forgotPassword', authController.forgotPassword);

router.patch('/resetPassword/:token', authController.resetPassword);

These routes handle user authentication, including signing up, logging in, logging out, and password reset functionalities.

Protected routes:

// Protects all routes after this middleware

router.use(authController.protect);

This middleware ensures that all the routes below it are protected, meaning the user must be authenticated to access them.

Update password Route: router.patch('/updateMyPassword', authController.updatePassword);

Allows users to update their password after authentication.

User profile routes:

```
router.get('/loggeduser', userController.getLoggedUser, userController.getUser);
router.patch(
  '/updateLoggedUser',
  userController.uploadUserPhoto,
  userController.updateLoggedUser,
);
router.delete('/deleteLoggedUser', userController.deleteLoggedUser);
```

These routes are related to the user's own profile where each user can retrieve its profile information, update its profile details, and delete its own account.

Admin-Only routes:

```
router.use(authController.restrictTo('admin'));
router
  .route('/users')
  .get(userController.getAllUsers)
  .post(userController.createUser);

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

These routes are restricted to admin users, where they can view all users, create a new user, and perform CRUD operations on a specific user.

Admin user edit route:

```
router
  .route('/users/edit/:id')
  .get(authController.protect, viewsController.getUserAsAdmin)
```

```
  .patch(
    authController.protect,
    authController.restrictTo('admin'),
    userController.updateUserAsAdmin,
  );
```

This route allows an admin to view and edit the details of a specific user, with userRoutes module to be the responsible for handling various user-related actions, including authentication, profile management, and administrative actions, helps controllers to execute specific functionalities and ensures proper authentication and authorization.

tourRoutes

The reviewRouter at the endpoint /:tourId/reviews handles routes related to reviews for a specific tour where the are defined for handling CRUD operations on tours.

- GET /tours: get all tours.
- POST /tours: create a new tour, the route is protected and restricted to admin role.
- GET /tours/:id: get a specific tour.
- PATCH /tours/:id: update a specific tour, the route is protected and restricted to admin.
- DELETE /tours/:id: delete a specific tour, the route is also protected and restricted to admin and guide roles.

ReviewRoutes

The authController.protect middleware is used to protect all routes in this router, ensuring that only authenticated users can access them.

- GET /: get all reviews, this route is protected.
- POST/:create a new review, this route is restricted to users (authController.restrictTo('user')) and uses the reviewController.setTourUserIds middleware to set the tour and user IDs for the review.
- GET /:id: get a specific review.
- PATCH /:id: update a specific review. This route is restricted to users and admins (authController.restrictTo('user', 'admin')).
- DELETE /:id: delete a specific review, this route is also restricted to users and admins

Admin Review Routes:

- GET /reviews: get all reviews
- POST /reviews: create a new review
- GET /reviews/edit/:id: get a specific review
- PATCH /reviews/edit/:id: update a specific review as an admin, protected and restricted to admins.
- DELETE /reviews/edit/:id: delete a specific review as an admin (

BookingRoutes

CRUD Routes for Bookings:

```
router
  .route('/all-bookings')
  .get(bookingController.getAllBookings)
  .post(bookingController.createBooking);


router
  .route('/:id')
  .get(bookingController.getBooking)
  .patch(bookingController.updateBooking)
  .delete(bookingController.deleteBooking);
```

These routes handle CRUD (Create, Read, Update, Delete) operations for bookings that include retrieving all bookings, creating a new booking, getting a specific booking by ID, updating a booking, and deleting a booking. In summary, routes handling bookings, tours, users, including creating checkout sessions, managing bookings, and enforcing authentication and authorization.

3.5.2 CRUD

CRUD stands for Create, Read, Update, and Delete, it represents the four basic operations that can be performed on data, especially in a database. These operations allow to manage the lifecycle of websites's data [24].

- Create (C): adding new data, typically implemented using the HTTP POST method.
- Read (R): to retrieve existing data from your system, typically implemented using the HTTP GET method[19][21][22].
- Update (U): to modify existing data in your system, typically implemented using the HTTP PUT or PATCH method [19][21][22].
- Delete (D): to remove existing data from your system, typically implemented using the HTTP DELETE method. [24][23]

In the website's backend, these CRUD operations map to specific routes that handle the corresponding actions and they are fundamental in building applications and websites that interact with data. Also, as they provide a standardized way to perform essential actions on data, make the website consistent and easy to understand. In the meaning of RESTful APIs, these operations are often mapped to HTTP methods, and their implementation follows RESTful principles. Below is an overview of the main functionalities implemented [19][21] [22] in *Pegasus Epirus Equitours*:

Forms

User Signup:

- creates a new user based on the provided input (name, email, password, password confirmation).
- sends a welcome email to the user using the Email class.
- generates a JSON Web Token (JWT) using the signToken function.
- sends the JWT in an HTTP-only cookie.
- returns a JSON response with the user data and the JWT.
- user Login (exports.login):
- validates if the email and password are provided.
- finds the user with the provided email.
- checks if the provided password matches the stored hashed password.

- generates a JWT and sends it in an HTTP-only cookie.
- returns a JSON response with the user data and the JWT. [19][21][22]

User Logout (exports.logout):

- clears the JWT cookie to log the user out.
- user Authentication Middleware (exports.protect)
- checks for the presence of a JWT in the request headers or cookies.
- verifies the JWT.
- checks if the user associated with the token still exists.
- checks if the user changed the password after the token was issued.
- grants access to protected routes and sets req.user and res.locals.user. [19][21][22]

Authorization : restricts access to certain roles (admin or user) for specific actions.

Check if user is logged in:

- checks if there is a valid JWT cookie in the request.
- if so, sets res.locals.user to the current user.

Forgot password:

- generates a password reset token.
- sends a password reset email with a link containing the token.

Reset password:

- finds the user based on the hashed reset token.
- checks if the token is not expired.
- sets the new password for the user.
- clears the password reset token fields.
- logs in the user by sending a new JWT.

Update password:

- checks the correctness of the current password.
- updates the password and password confirmation.
- logs in the user by sending a new JWT.

Utility functions:

signToken: generates a JWT based on the user's id

createSendToken: sends a JWT in an HTTP-only cookie and responds with user data.

bookingController

For handling the booking and payment process using Stripe.

Get checkout session:

- retrieves the currently booked tour based on the tourId parameter.
- creates a checkout session using the Stripe API.
- defines payment method types, success and cancel URLs, customer email, and line items.
- returns the created session as a JSON response.

Create booking checkout:

- temporarily creates a booking without real payment
- extracts tour, user, and price from the query parameters
- creates a booking using the Booking model
- redirects back to the original URL, removing query parameters

Check admin role:

- check if the logged-in user has an admin role
- if the user has an admin role, it proceeds to the next middleware.
- if not, it responds with an unauthorized status.

Get all bookings:

- verify the admin role

- if the user has an admin role, retrieves all bookings from the database using Booking.find().
- renders the 'bookings' pug template with the booking data.

Booking CRUD operations:

- exports.createBooking: creates a new booking
- exports.getBooking: retrieves a specific booking
- exports.updateBooking: updates a booking
- exports.deleteBooking: deletes a booking

Finally, the Stripe payment gateway handles the checkout process.

reviewController

Additionally, implementation for role-based access needed to ensure that certain operations are only accessible to users with an admin role [19][21][22].

Check admin role (exports.checkAdminRole):

- middleware function to check if the logged-in user has an admin role.
- if the user has an admin role, it proceeds to the next middleware.
- if not, it responds with an unauthorized status.

Get All Reviews (exports.getAllReviews):

- uses the checkAdminRole middleware to verify the admin role.
- if the user has an admin role, retrieves all reviews from the database
- renders the 'allReviews' pug template with the review data.

Get review as admin (exports.getReviewAsAdmin):

- uses the checkAdminRole middleware to verify the admin role
- if the user has an admin role, retrieves a specific review by its id
- renders the 'editReview' pug template

Update review as admin (exports.updateReviewAsAdmin):

- middleware function to update a review as an admin.
- extracts review and rating from the request body
- converts review and rating to strings if they are objects
- creates an object (updateFields) with the fields to update
- finds and updates the review in the database using Review.findByIdAndUpdate().
- handles validation errors and other errors appropriately

Review CRUD Operations:

- exports.getReview: retrieves a specific review
- exports.createReview: creates a new review
- exports.updateReview: updates a review
- exports.deleteReview: deletes a review

  [19][21][22]

CRUD operations provide a structured way to handle reviews, including user authentication, role-based access control, and reusable CRUD operations.

TourController

It includes functionalities for image upload, handling routes for tours, and performing various operations related to tours, such as getting statistics, monthly plans, tours within a certain distance, and distances between locations [19][21][22][24].

Tour CRUD Operations:

- exports.getAllTours: retrieves all tours from the database
- exports.getTourAsAdmin: retrieves a specific tour by ID after checking the admin role.
- exorts.updateTour: updates a tour based on the provided data.

userController

Includes functionalities for user authentication, profile update, deletion, and user management by administrators [19][21][22][24].

User CRUD Operations for Admins:

- Middleware (exports.checkAdminRole): checks if the logged-in user has an admin role.
- exports.getAllUsers: retrieves all users from the database after checking the admin role.
- exports.getUserAsAdmin: retrieves a specific user by ID after checking the admin role.
- exports.updateUserAsAdmin: updates a user's profile as an admin, filtering out unwanted fields.

3.6 Node.js and Express

Node.js and Express together form the Controller, handling the flow of data and communication between the Model and View. Firstly, Node.js enables server-side JavaScript, while Express provides a robust framework for building APIs [27][24][28]. At the same time, middleware and dependencies enhance the Controller's capabilities, including security measures and optimizations for data processing. Specifically, security measures include middleware such as Helmet for setting HTTP headers, Express Rate Limit for request throttling, and XSS-clean for preventing cross-site scripting attacks [27]. Finally, other dependencies, such as bcryptjs for password hashing, JSON Web Token for user authentication, and Mongoose for MongoDB interaction, contribute to a secure and efficient backend[28].

3.7 Database Management (MongoDB, Mongoose)

The database management with MongoDB, a NoSQL database, and Mongoose, serves as the Model, managing data storage, retrieval, and the definition of data schemas.

3.7.1 Mongo DB

I signed up for a free MongoDB Atlas account, making use of the cloud-based database service that provides a cost-effective solution for managing databases. Once registered, I created a new cluster in the free tier's features. In this cluster, a collection of servers storing my data, was configured by selecting the cloud provider, region, and adjusting other settings with helpful tips guiding me through the process.

Specifically, configuring the free cluster involved creating a database user with the necessary permissions, as this user was essential for connecting to the database. Firstly, to access the cluster, I clicked the "Connect" button and followed the provided instructions. On the other hand, connecting to MongoDB Shell allowing me to perform basic CRUD operations in a manner similar to a locally hosted MongoDB server. Security and network configuration were of important considerations, even with the free tier. So, I utilized features like IP whitelist settings to control access to my cluster, ensuring only authorized entities could interact with the database.

Throughout my use of the free MongoDB Atlas option, I found the MongoDB Atlas documentation to be a valuable resource, as it covered a wide range of topics, providing guidance on setting up the cluster, optimizing performance, and maintaining security, and of course ensuring a smooth experience.

My Mongo DB account:

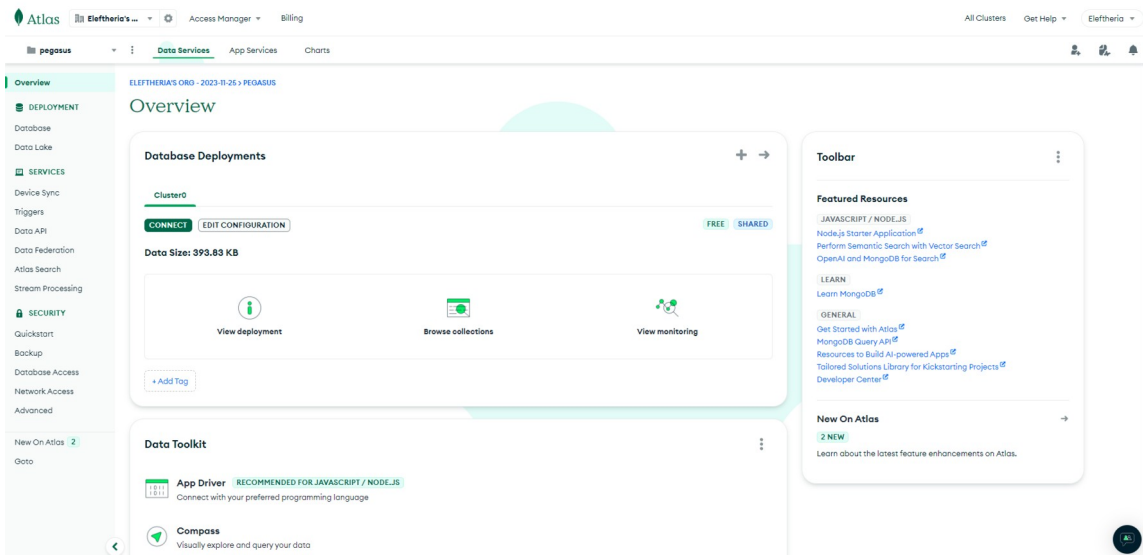


Figure 3.7 MongoDB account

My MongoDB Cluster:

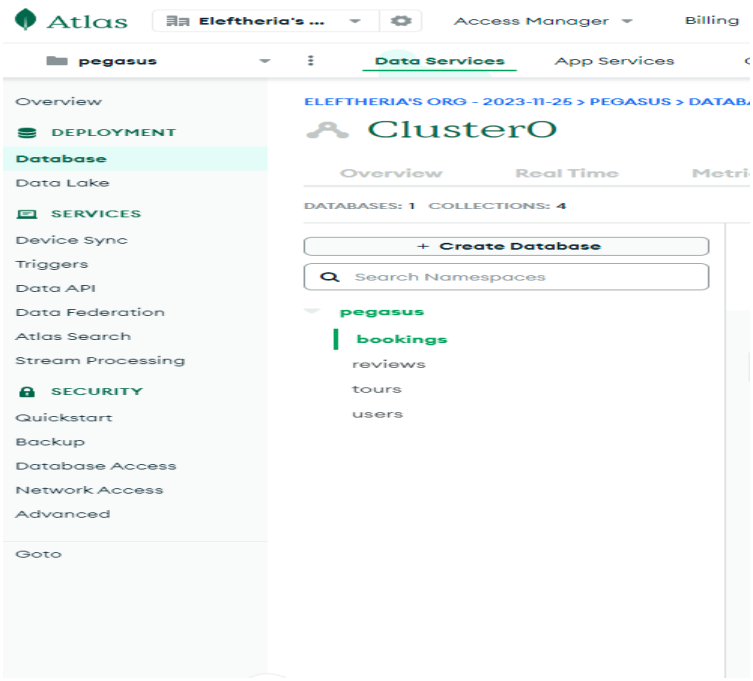


Figure 3.8 MongoDB cluster

My Mongo DB schema:

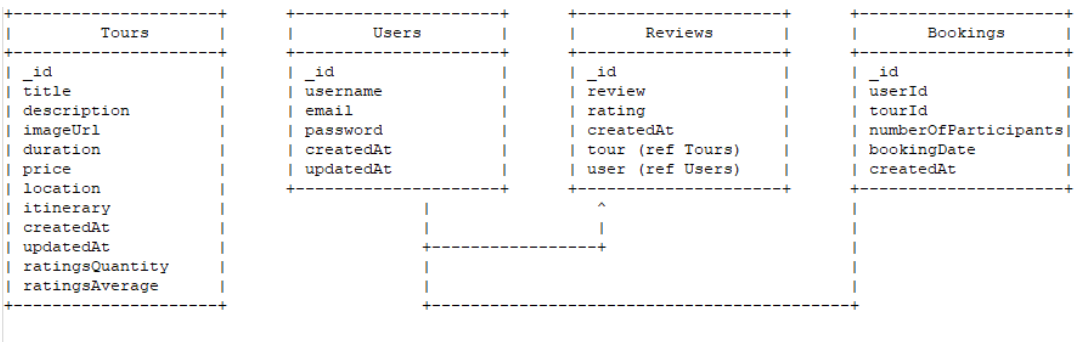


Figure 3.9 MongoDB schema

3.7.2 Mongoose

In the development of the *Pegasus Epirus Equitours* website, Mongoose library with MongoDB and Node.js provided a structured and efficient approach to handle data modeling, validation, and database operations.

One significant use of Mongoose in the project was the definition of schemas. Particularly, schemas giving the structure and validation rules for MongoDB collections. For instance, the tourSchema was created to define the structure of the Tour collection, specifying fields like title, description, and others. This schema ensured data consistency and integrity by enforcing validation rules.

Moreover, Mongoose facilitated the creation of models based on these schemas, as Models served as constructors, enabling documents conforming to the defined structure. The Tour model, for instance, allowed for the creation and retrieval of tour documents in the database.

```
const Tour = mongoose.model('Tour', tourSchema);
```

The following code snippet illustrates the creation of a new tour document:

```
const newTour = new Tour({
  title: 'Pindus',
  description: 'Explore ancient trails and mythical landscapes in Pindus...',
});
newTour.save();
```

Mongoose middleware functions were instrumental in performing operations before or after certain events, such as document saving. In the *Pegasus Epirus Equitours*, pre-save middleware was employed to calculate average ratings for reviews and update related data.

Another crucial aspect, was to establish relationships between different collections by using Mongoose. So trough references, relationships were defined between the Review collection and other collections like Tour and User. This allowed navigation between related data.

```
const reviewSchema = new mongoose.Schema({
  // ... other fields
  tour: {
    type: mongoose.Schema.ObjectId,
    ref: 'Tour',
```

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

Moreover, Mongoose provided a powerful framework, enabling the execution of complex queries and data transformations, that was successfully utilized in the website. In conclusion, Mongoose played a pivotal role in shaping the data layer of Pegasus Epirus Equitours, providing an expressive interface for MongoDB interactions, schema definition, middleware implementation, relationship establishment, and aggregation operations. The integration of Mongoose contributed to the efficiency of the website's backend, ensuring a reliable and performance user experience.

3.7.3 Secure Payment Implementation with Stripe

Secure payment transactions are facilitated through the integration of Stripe, a trusted payment gateway, and this implementation takes place within the Controller layer, ensuring the security and integrity of financial transactions [27]. The Stripe module is imported and configured with the Stripe secret key obtained from the environment variables.

Checkout session:

- responsible for creating a checkout session when a user initiates the payment process.
- retrieves the details of the selected tour from the database
- uses the Stripe API to create a checkout session, specifying payment details, success and cancel URLs, customer email, and line items.
- session is created
- session object is then sent as a response to the client

Booking Checkout :

- extracts the tour, user, and price details from the query parameters.
- if these details are present, it creates a booking

Middleware: there is a middleware function that checks if the user has an admin role.

Admin booking retrieval: for retrieving all bookings (admin-only access), checks if the user has an admin role (exports.checkAdminRole), and if so, it retrieves all bookings from the database and renders a template.

Admin routes: specifically designed for admin access to booking-related functionalities.

To summarize, the code implements secure payment processing using the Stripe API as it creates a checkout session for a selected tour, handles successful payment events, and includes admin-specific functionality for managing bookings [27].

3.8 Development Tools and Environments

The development process is supported by various tools and environments to enhance efficiency, collaboration, and testing.

3.8.1 Visual Studio Code

Visual Studio Code serves as the primary integrated development environment (IDE), providing features such as code highlighting, debugging, and version control integration. To begin with, Visual Studio Code is a lightweight, open-source code editor developed by Microsoft, highly extensible as it is has gained widespread popularity across different platforms, including Windows, macOS, and Linux [26].

Specifically, Visual Studio Code is available for Windows, macOS, and Linux, making it accessible and provides features like syntax highlighting, auto-completion, and intelligent code suggestions to enhance the coding experience. Also, it includes built-in Git support for version control, allowing to manage source code repositories directly within the editor, and there are extensions available for various programming languages, frameworks, and tools. Moreover, it has built-in support for debugging with breakpoints, call stacks, and an interactive console.

Also, VSCode includes an integrated terminal, allowing us to run command-line tools and scripts directly within the editor, which supports task automation using the task runner, making it easier to run build tasks and other scripts. Of course it gives the opportunity to customize the editor's appearance, including themes, icons, and other settings. Known for its performance, flexibility, and the active community that continuously contributes to its improvement, it's widely used for a variety of software development tasks, from web development to system programming [26].

To summarize, I chose to use it with some really helpful extensions that really made the hole process mush more easier. Firstly, I installed GitHub Pull Requests and Issues v0.78.1 extensions to help me ‘commit’ and ‘push’ much more easier. For my JavaScript I

installed ESLint v2.4.2 and Prettier - Code formatter v10.1.0 for all the project code, and also I installed DotENV v1.0.1 for my file system [26].

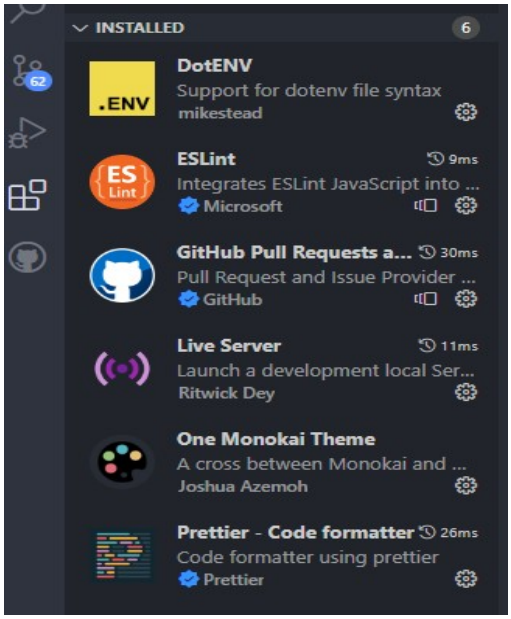


Figure 3.10 Visual Studio Code extensions

3.8.2 Postman

Postman is utilized for testing and validating APIs, ensuring seamless communication between the frontend and backend components. Postman is a popular collaboration platform for API development that simplifies the process of developing, testing, and documenting APIs. It provides a user-friendly interface to design, build, and test APIs, making it an essential tool for developers and teams working on web services and APIs [25].

Allows developers to create and execute API requests, supporting various HTTP methods like GET, POST, PUT, DELETE, etc. It supports both simple HTTP requests and complex scenarios with multiple requests, parameterization, and scripting. Users can easily construct HTTP requests with various parameters such as headers, query parameters, request bodies, and authentication details. Requests can be organized into collections, which are groups of related API calls. This helps in managing and structuring API tests and workflows. It also supports the use of environment variables, allowing developers to parameterize requests and easily switch between different environments (e.g., development, testing, production) [25].

Postman provides a testing framework that allows users to write and execute automated tests for APIs. Test scripts can be written in JavaScript, enabling developers to perform validations and assertions. Users can define assertions to verify the response of API requests, ensuring that the API behaves as expected. Assertions can be based on status codes, response bodies, headers, and more. _Postman allows the creation of mock servers, which are simulated servers that respond to API requests. This is useful for testing APIs before they are fully developed or when certain conditions need to be simulated. In addition, provides monitoring capabilities to track the performance of APIs over time. It allows users to set up monitors to regularly test APIs and collect data on response times, uptime, and more [25].

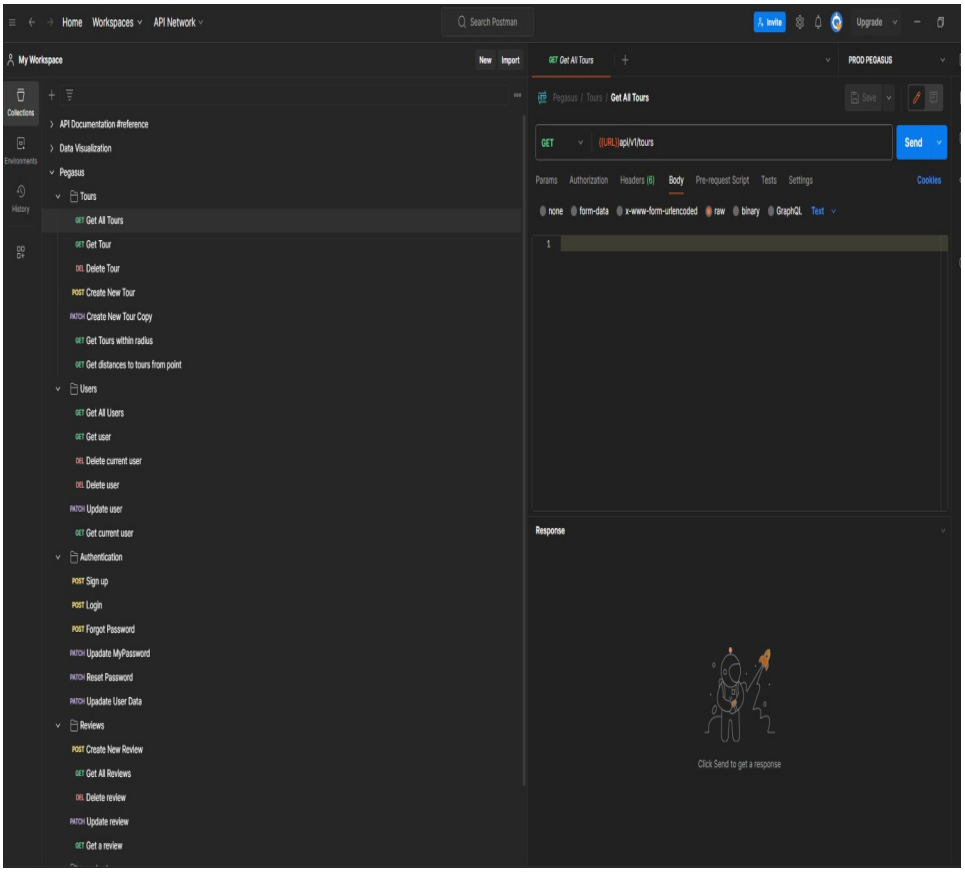


Figure 3.11 Postman enviroment

Postman simplifies the process of generating API documentation. Developers can create detailed documentation for their APIs, including information about requests, responses, and examples. Postman is available as both a web application and a desktop application, making it accessible to users on different platforms. The platform has a large user community, and its extensive features contribute to its widespread adoption in the software development community [25].

Postman, played a crucial role in testing and validating various endpoints of the Pegasus Epirus Equitours backend. I downloaded the desktop application and first I tested all my routes and CRUD operations there both for development and production environment.

Chapter 4: User Manual

- We open the VS Code in View>Termninal> **npm start** and this starts the nodemon server.js script.

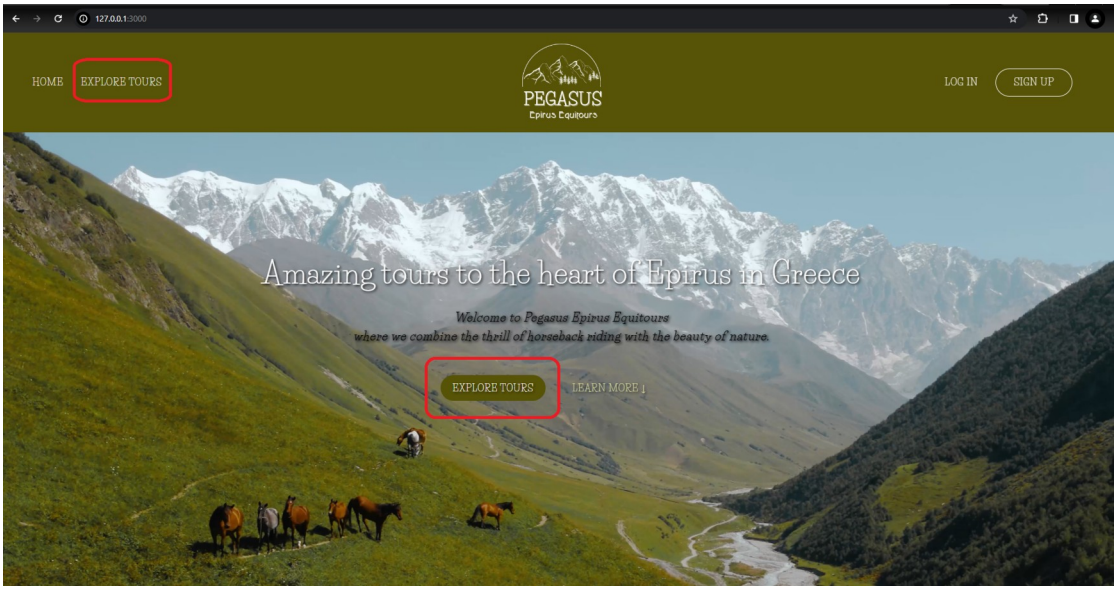


Figure 4.1 Pegasus homepage

- We open a browser and type our localhost ip in port 3000
- This is the **Homepage**:
- We can click **Explore tours** on the Header or in the button in the center of the page



Figure 4.2 Pegasus photo gallery

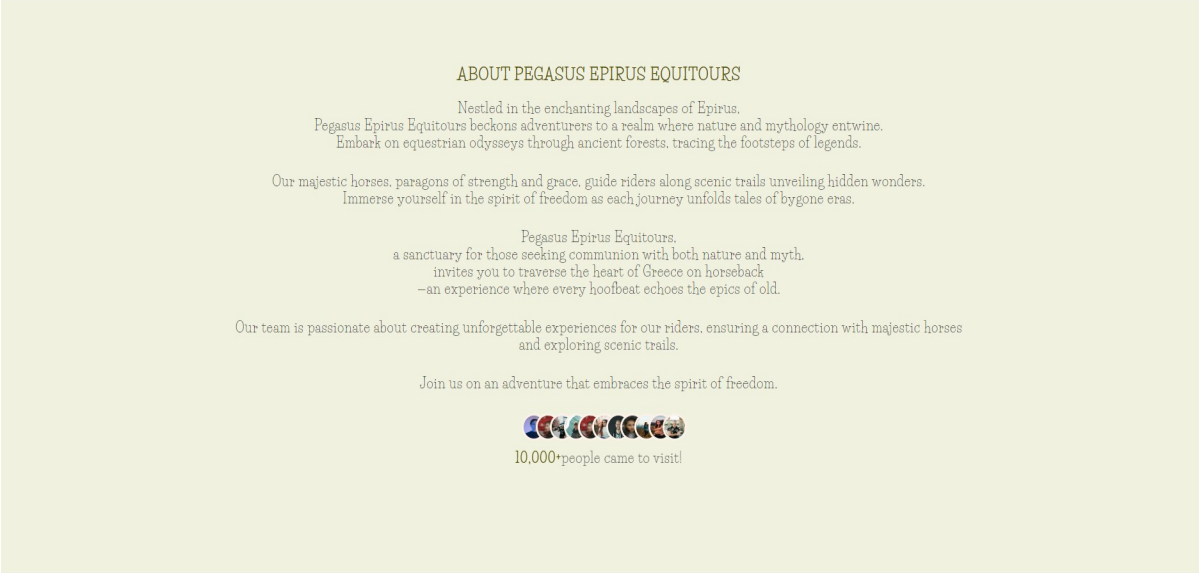


Figure 4.3 About us section of the website



Figure 4.4 Facts about Pegasus sections

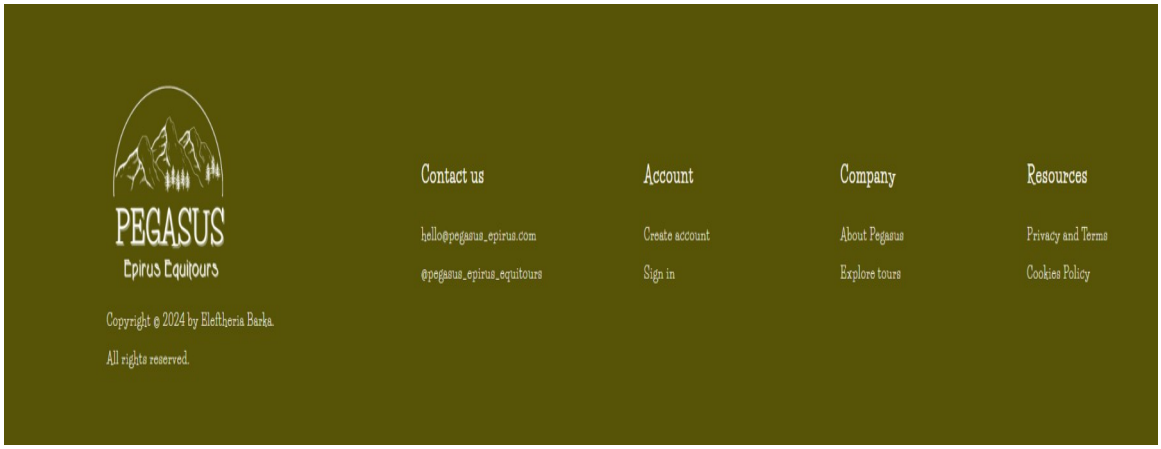


Figure 4.5 Footer section

Both buttons directs us on the tours page:

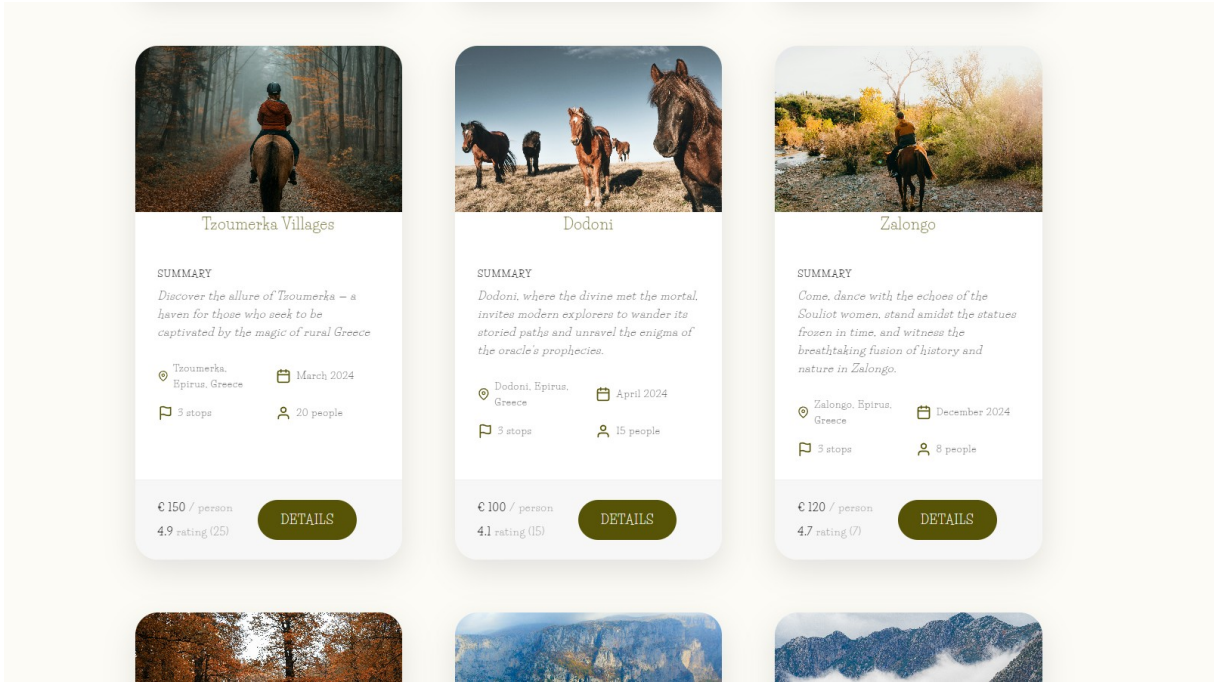


Figure 4.6 Pegasus tours page



Figure 4.7 Tour's details page



Figure 4.8 About Tzoumerka tour



Figure 4.9 Tzoumerka photos

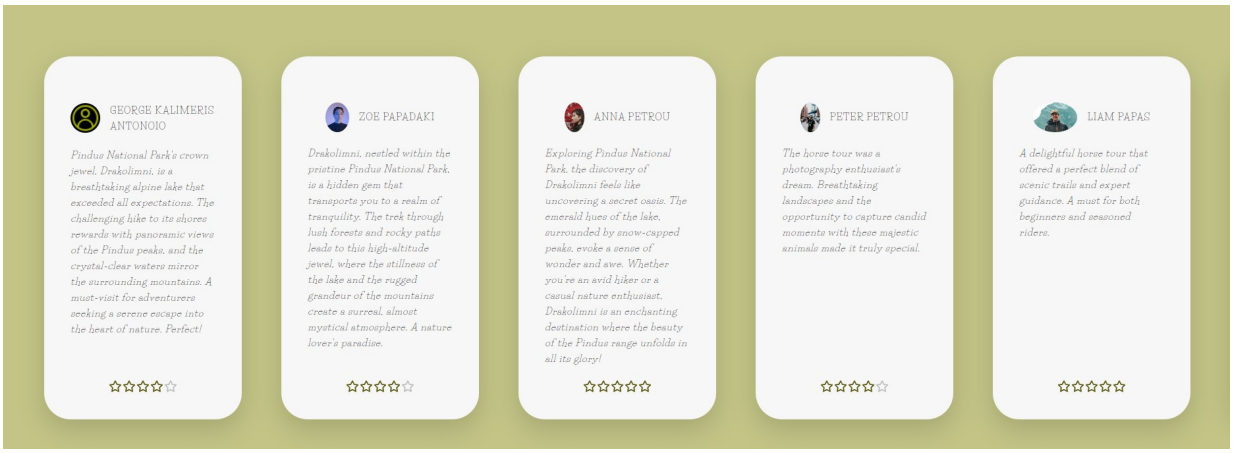


Figure 4.10 Tzoumerka reviews

4.1 User

Now the **Log in button** directs us the login form where we can login as **user** or as admin:

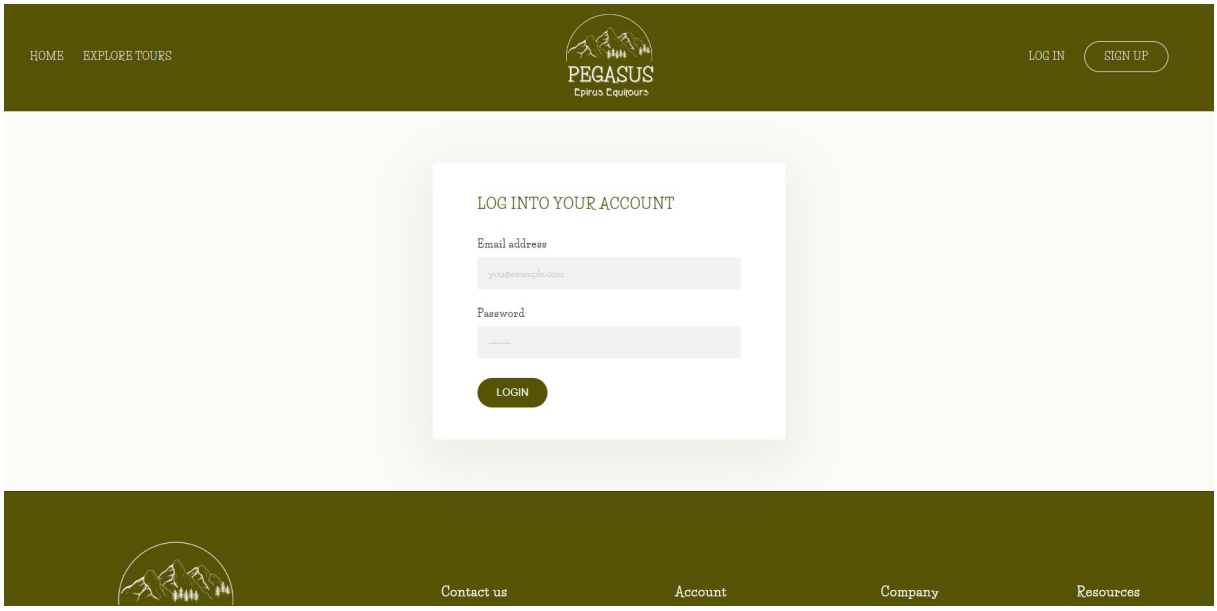


Figure 4.11 Login page

My reviews

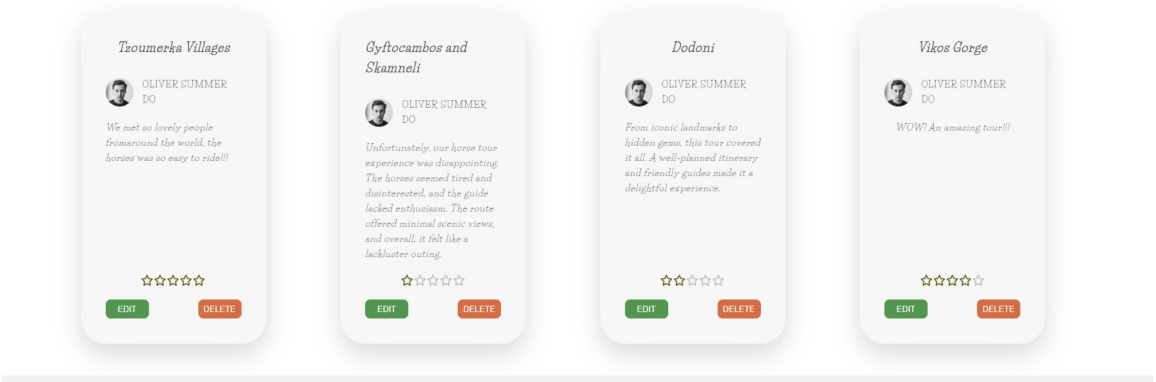


Figure 4.12 My reviews

Edit my review

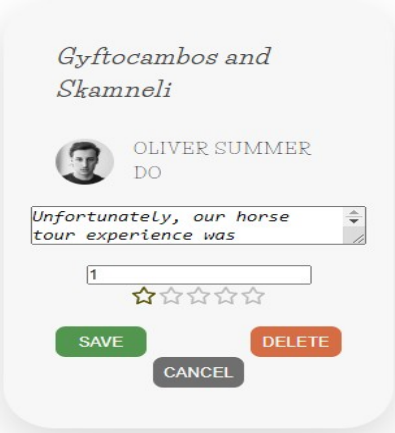


Figure 4.13 Edit my reviews

Leave a review

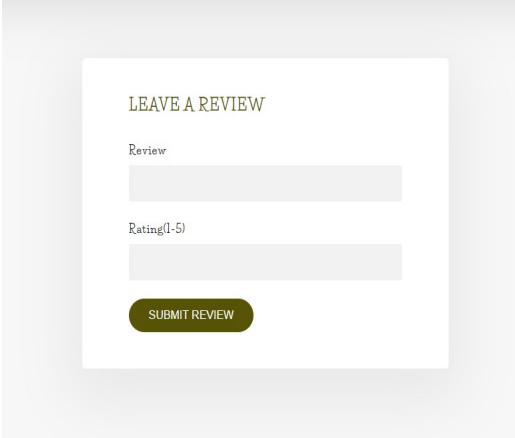


Figure 4.14 Leave a review

Payment Form supported by Stripe:

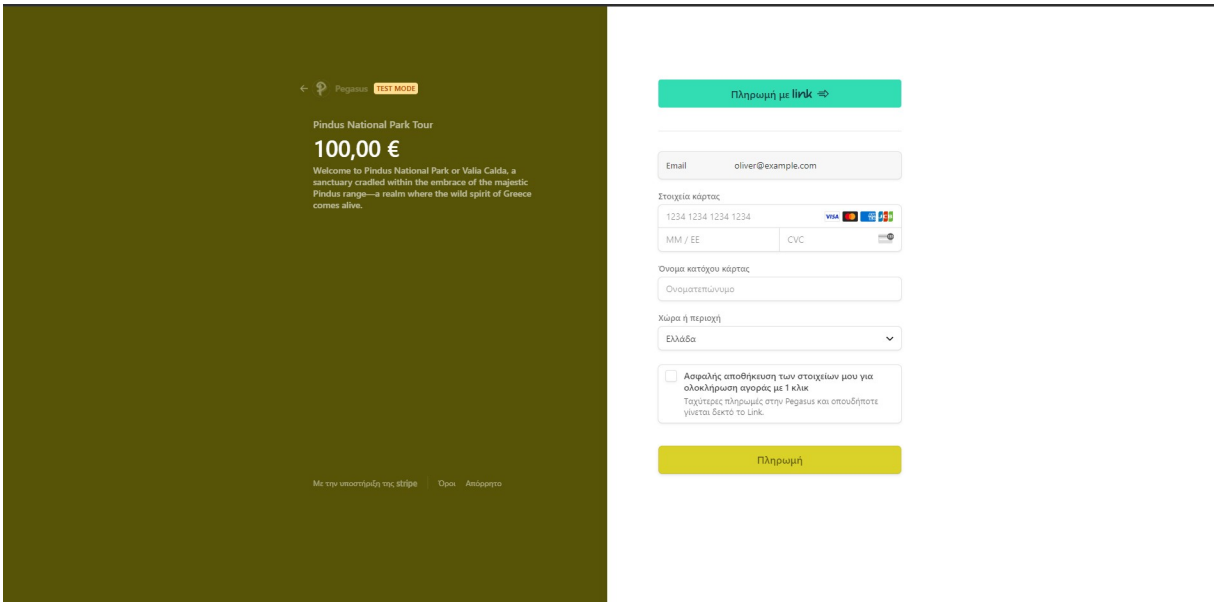


Figure 4.15 Stripe payment page

4.2 Administrator

All reviews

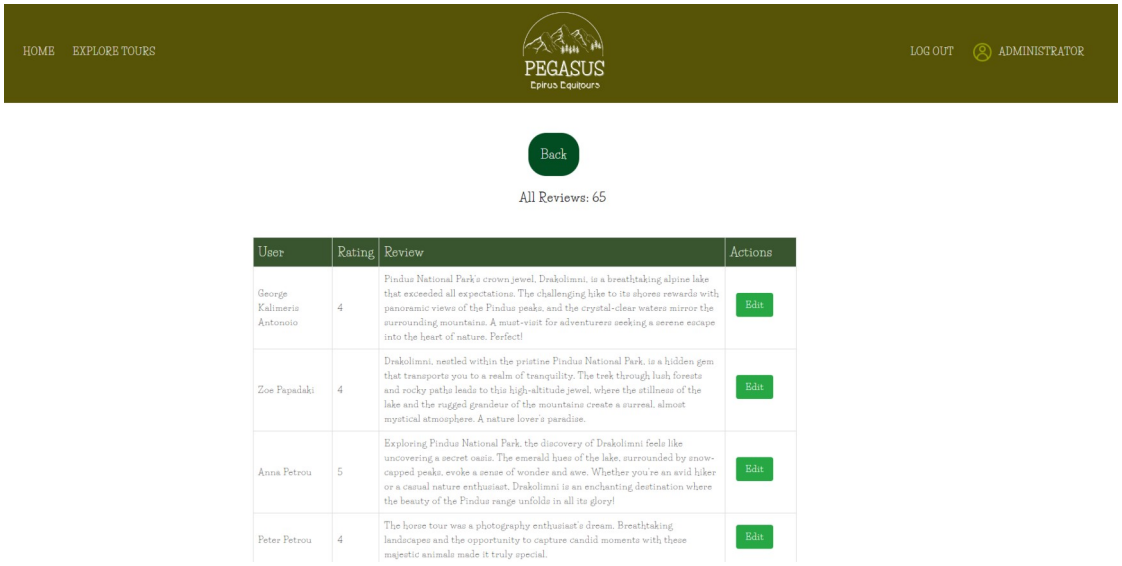


Figure 4.16 Administrator page of all reviews

Edit a user

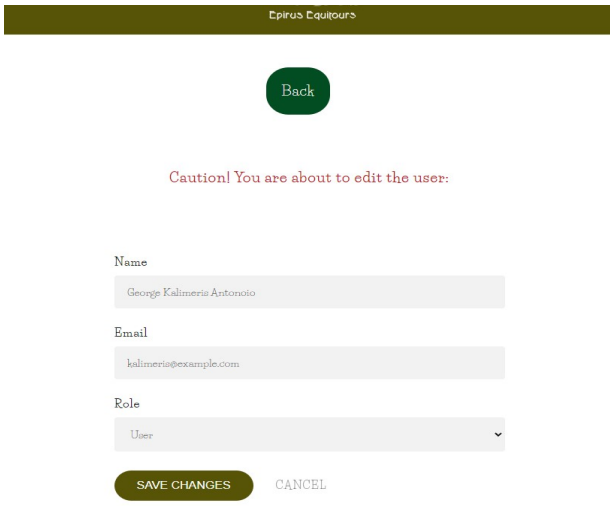


Figure 4.17 Administrator edits a user page

Edit a Tour



Figure 4.18 Administrator edits a tour page

Edit a review

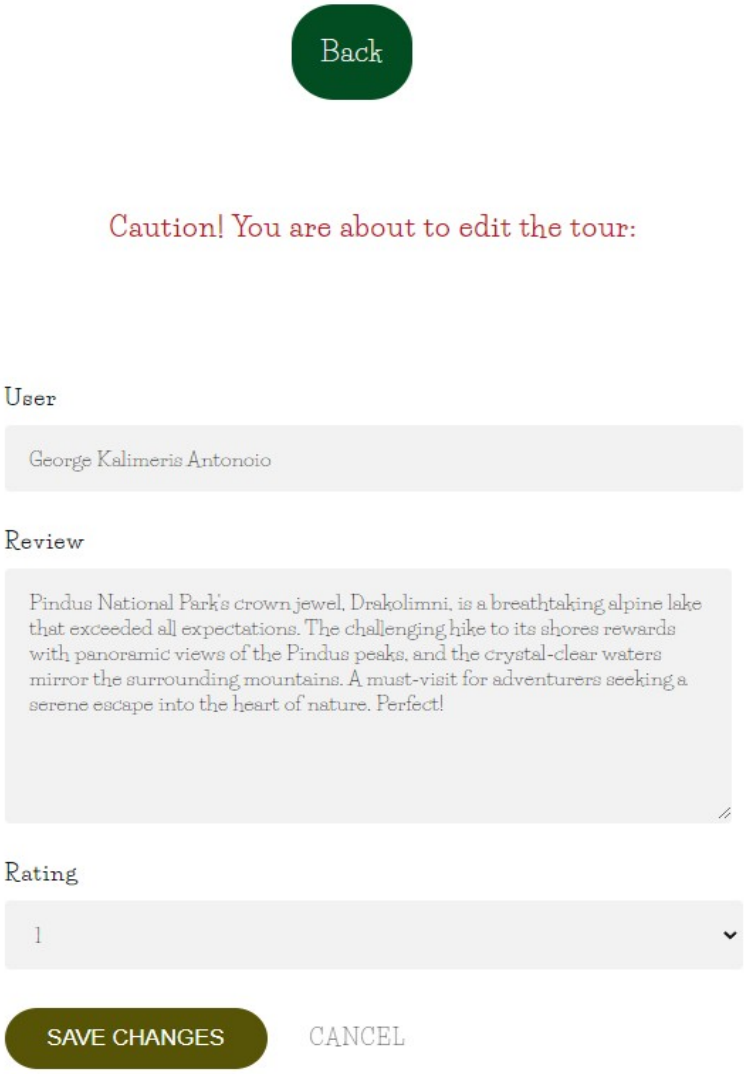


Figure 4.19 Administrator edits a review page

See all users

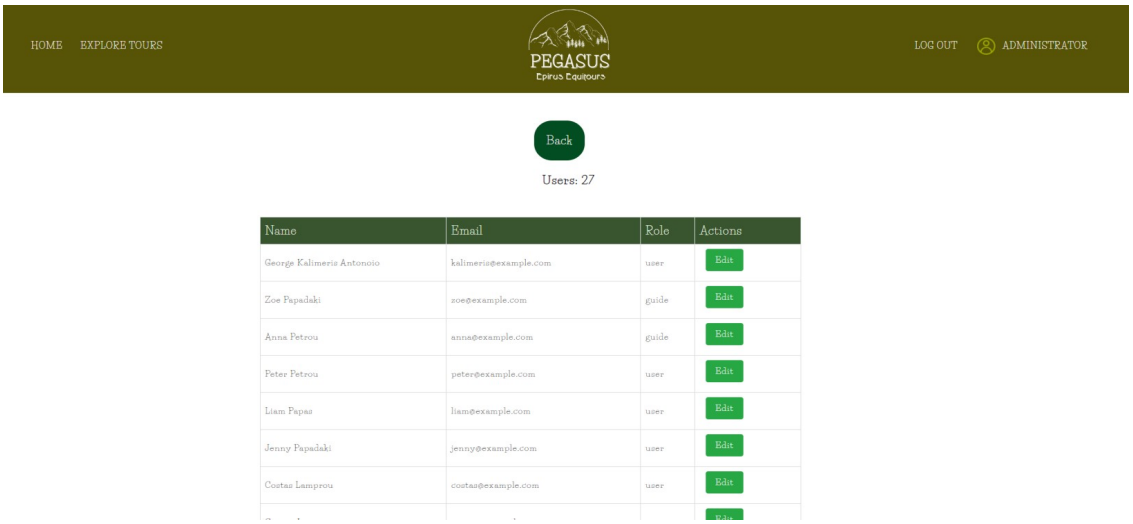


Figure 4.20 Administrator and all users page

See all bookings

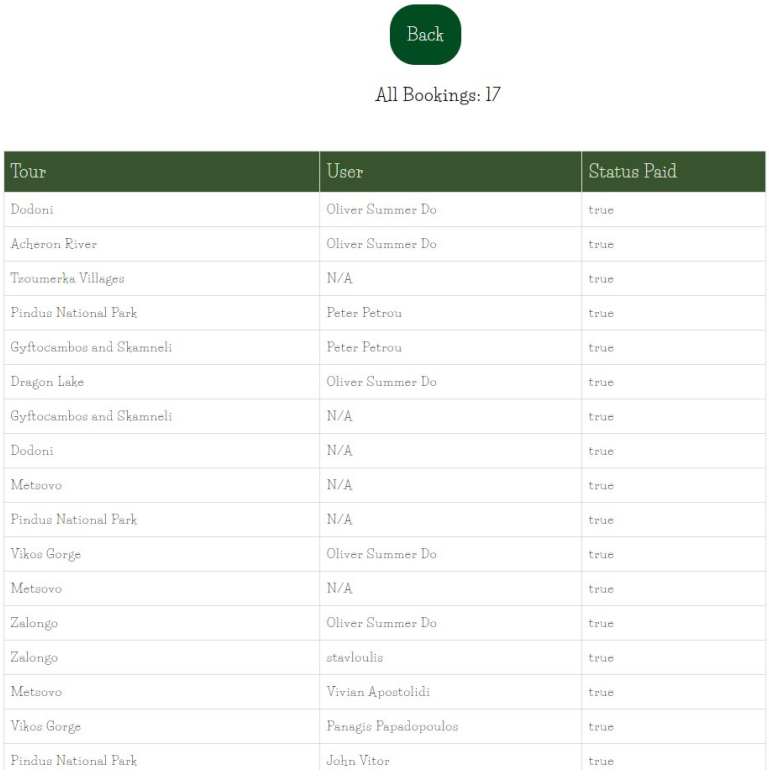


Figure 4.21 Administrator and all bookings page

Chapter 5: Summary

5.1 Summary of Technologies Used in Pegasus Epirus Equitours

Frontend Technologies

1. HTML (HyperText Markup Language):
    - Foundation for structuring web content.
    - Used for creating the basic structure of web pages.
2. CSS (Cascading Style Sheets):
    - Responsible for styling and visually enhancing the website.
    - Defines the layout, colors, and overall visual presentation.
3. JavaScript:
    - Adds interactivity and dynamic features to the website.
    - Used for client-side scripting to enhance user experience.
4. Pug Templates:
    - A templating engine for Node.js.
    - Simplifies HTML syntax, aiding in the dynamic rendering of content.

Backend Technologies

1. Node.js:
    - JavaScript runtime for server-side development.
    - building scalable and high-performance website
2. Express:
    - web application framework for Node.js.
    - simplifies the web development
3. MongoDB:
    - NoSQL database used for storing and retrieving data.
    - provides flexibility and scalability, especially for large datasets.
4. Mongoose:
    - an Object Data Modeling (ODM) library for MongoDB and Node.js.
    - simplifies interactions with MongoDB, providing a schema-based solution.

Middleware and Dependencies

1. npm (Node Package Manager):
    - manages and installs project dependencies.
    - ensures a streamlined process for acquiring and managing external packages.
2. nodemon:
    - monitors changes in the source code and automatically restarts the server.
    - enhances the development workflow by reducing manual server restarts.
3. dotenv:
    - loads environment variables from a .env file into process.env.
    - keeps sensitive information secure and allows for configuration management.
4. morgan:
    - HTTP request logger middleware for Node.js.
    - logs request details, aiding in debugging and performance analysis.
5. slugify:
    - converts strings into URL-friendly slugs.
    - useful for creating readable and SEO-friendly URLs

Other Tools

1. Postman:
    - API development and testing tool.
    - facilitates the testing of API endpoints and ensures communication between frontend and backend.
2. Visual Studio Code:
    - integrated development environment (IDE).
    - offers features for code editing, debugging, and version control.
3. Mongo Compass:
    - GUI for MongoDB.
    - allows for visual interaction with MongoDB databases.

5.2 Conclusion

Overall, the designing phases was a creative journey of learning technologies to develop the front end and back end architecture and as the database schema were visualize the structure and the flow of the website the utilization of Mongoose schemas and Pug templates played a pivotal role in defining a dynamic data model and a responsive front end.

Moreover, as the architectural design took shape, Node.js, Express, and MongoDB, the back end development were structured and Mongoose with data modeling helped with the database. On the front end part, CSS and JavaScript, with Pug templates made a dynamic content rendering with the incorporation of secure payment through Stripe further extend the website's capabilities. In the testing phase, Postman helped to ensure not only the functionality but also the security and usability of the website, as it helped in API testing by validating various endpoints and scenarios.

Pegasus Epirus Equitours combines powerful technologies both in front end and backend, tools that is adeal to create a dynamic, and user friendly website for equestrian tourism experiences. Moreover, by providing an amazing front end experience, Pegasus Epirus Equitours becomes a gateway for enthusiasts to explore the rich landscapes of Epirus. On the technological aspect, the website stands as a testament to innovative web development practices.Combines cutting-edge technologies, secure payment integrations, and user-friendly design, that sets a benchmark for the intersection of technology and tourism.

In conclusion, Pegasus Epirus Equitours is not just a website, it's a journey into the heart of Epirus, by combining mythology with modern technology. As the website evolves, it promises not just a ride through landscapes but a continued exploration of possibilities in equestrian tourism and web development.

References

1. Newsome, David & Smith, Amanda & Moore, Susan. (2008). Horse Riding in Protected Areas: A Critical Review and Implications for Research and Management. Current Issues in Tourism, 11 (2). pp. 144-166.. 11. 10.2167/cit336.0.
2. Mikhailova, Alina & Zinnatova, Mariya. (2023). Role of Horse Riding in Changing the Psycho-Emotional State of a Person. Lurian Journal. 3. 41-50. 10.15826/Lurian.2022.3.4.3.
3. Martin Vodopivec & Maja Konečnik Ruzzier. (2022) Opportunities in Identifying and Marketing Windsport Tourism Destinations: High-Resolution Wind Analysis.Sustainability 14:24, pages 16589.
4. Vranousis, L. – Sfiroeras, V. (1997) "Η Ήπειρος κατά τους νεώτερους χρόνους, Δημογραφικές εξελίξεις – Οι Ηπειρώτες της Διασποράς, 257 – 264 ΗΠΕΙΡΟΣ 4000 χρόνια ελληνικής ιστορίας και πολιτισμού.“
5. Gkiokas, G. (1997) "Δυνατότητες τουριστικής ανάπτυξης της Ηπείρου"
6. Nitsiakos, V. (1998) "Νομός Ιωαννίνων, Σύγχρονη Πολιτισμική Γεωγραφία, Νομαρχιακή Αυτοδιοίκηση Ιωαννίνων, Ιωάννινα."
7. Nitsiakos, V. , Arapoglou, M., Tsounis, Gr. (2000) "Epirus – Following the River Paths," Η ΚΑΘΗΜΕΡΙΝΗ, Επτά Ημέρες, Sunday, July 9th, 2000.
8. K. Lei, Y. Ma and Z. Tan, "Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js," 2014 IEEE 17th International Conference on Computational Science and Engineering, Chengdu, China, 2014, pp. 661-668, doi: 10.1109/CSE.2014.142.
9. Mozilla Developer Network. "Document Object Model (DOM)." [https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model]
10. Mozilla Developer Network. "Learn CSS." [https://developer.mozilla.org/en-US/docs/Learn/CSS]
11. Mozilla Developer Network. "Fetch API." [https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API]
12. Mozilla Developer Network. "HTML DOM API." [https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API]
13. Mozilla Developer Network. "Secure Contexts." [https://developer.mozilla.org/en-US/docs/Web/Security/Secure_Contexts]

14. Mozilla Developer Network. "Web CSS." [https://developer.mozilla.org/en-US/docs/Web/CSS]

15. Mozilla Developer Network. "Web JavaScript." [https://developer.mozilla.org/en-US/docs/Web/JavaScript]

16. Mozilla Developer Network."MVC (Model-View-Controller)." [https://developer.mozilla.org/en-US/docs/Glossary/MVC]

17. Mozilla Developer Network."Using the Document Object Model." [https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Using_the_Document_Object_Model]

18. Mozilla Developer Network. "Express/Node.js Tutorial: Local Library Website." [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Tutorial_local_library_website]

19. Mozilla Developer Network. "Express/Node.js Tutorial: Skeleton Website." [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/skeleton_website]

20. Mozilla Developer Network. "Express/Node.js Tutorial: Mongoose." [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose]

21. Mozilla Developer Network. "Express/Node.js Tutorial: Routes." [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes]

22. Mozilla Developer Network."Express/Node.js Tutorial: Displaying Data." [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Displaying_data]

23. Mozilla Developer Network. "Express/Node.js Tutorial: Forms." [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/forms]

24. Mozilla Developer Network. "Express/Node.js Tutorial: Deployment." [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/deployment]

25. Postman Learning. "Postman Documentation Overview." [https://learning.postman.com/docs/introduction/overview/]

26. Visual Studio Code. "Visual Studio Code Documentation." [https://code.visualstudio.com/docs]

27. Stripe. "Stripe API Documentation." [https://stripe.com/docs/api]

28. MongoDB."MongoDB Manual Reference." [https://www.mongodb.com/docs/manual/reference/]

29. Pug."Pug - Getting Started." [https://pugjs.org/api/getting-started.html]

30. Git. "Git Documentation." [https://git-scm.com/docs]

31. GitHub. (2022, November 28). "GitHub REST API." [https://docs.github.com/en/rest?apiVersion=2022-11-28]

32. GitHub. "GitHub Actions Documentation." [https://docs.github.com/en/actions]

33. GitHub. "GitHub Pages Documentation." [https://docs.github.com/en/pages]

34. Tripadvisor. (n.d.). Retrieved September 09, 2023, from https://www.tripadvisor.com.gr/