University of Peloponnese
Department of Informatics and Telecommunications

Developing Client-Centric Solutions: A Hybrid Application Utilizing RESTful Web Services for Tourist Accommodations

Karapiperakis Emmanouil
2022 2023 02008

Supervisor: Nikolaos Tselikas

A thesis submitted in partial fulfillment of the University's requirements for the masters degree.

January 2025

Πανεπιστήμιο Πελοποννήσου

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Ανάπτυξη Λύσεων Προσανατολισμένων στον Πελάτη: Μια Υβριδική Εφαρμογή που Χρησιμοποιεί RESTful Υπηρεσίες Διαδικτύου για Τουριστικά Καταλύματα

Καραπιπεράκης Εμμανουήλ
2022 2023 02008

Επιβλέπων: Νικόλαος Τσελίκας

Διπλωματική Εργασία

Ιανουάριος 2025

Abstract

Customer service and the effort to provide optimal services have always been a central topic of discussion, particularly in the Tourism sector, which is one of the main pillars of the Greek economy. Every visitor invests significant time in selecting the ideal destination for their vacation and inevitably spends an important amount of money on their stay. Therefore, it is only natural that they want their experience to meet their expectations. A key element is the friendly and discreet communication with the property owner, who should be able to answer questions, provide assistance regarding the accommodation and the surrounding area when it is needed. Additionally, the continuous upgrading of services and responsiveness to customer needs are crucial for creating positive impressions and ensuring repeat visits. Furthermore, the utilization of modern technologies for better management of reservations, enhancement of the customer experience and provision of personalized recommendations can significantly improve the overall stay experience. This comprehensive approach not only contributes to increased customer satisfaction but at the same time, enhances the property's reputation, making it a highly desirable destination for future guests

For this thesis, a hybrid application was developed, aimed at improving the communication between a tourist accommodation and its clients, promoting the property and enhancing the services offered through the collection of data during the guests' stay. The application is available for both Android and iOS mobile devices, covering a wide range of devices from small to larger sizes. For prospective customers, the application offers detailed information about the property and the surrounding area, as well as the ability to express interest in booking by selecting their desired dates. Additionally, customers who have already made a reservation receive login details upon arrival, granting them access to privileges such as additional information about the interior of the property, discovery of locations in the surrounding area through dynamic maps, access to information related to their stay, communication with a virtual assistant and the ability to directly contact the host. Finally, the administrator role provides the capability to dynamically add content to the application, manage users, update availability and handle incoming booking requests. The administrator can also respond to live chats with guests and access useful statistical data, which can help in the continuous improvement of the services offered by the property.

Keywords: Tourism, Customer Service, Web Services, REST APIs, Web Sockets, Node.js, Express, Cross Platform Development, React Native, iOS, Android, Heroku, GCP

Περίληψη

Η εξυπηρέτηση πελατών και η προσπάθεια παροχής βέλτιστων υπηρεσιών αποτελούσαν πάντα κεντρικό θέμα συζήτησης, ειδικά στον τομέα του Τουρισμού, ο οποίος αποτελεί έναν από τους κύριους πυλώνες της ελληνικής οικονομίας. Κάθε επισκέπτης επενδύει σημαντικό χρόνο για να επιλέξει τον ιδανικό προορισμό για τις διακοπές του και αναπόφευκτα, διαθέτει ένα μη ευκαταφρόνητο χρηματικό ποσό για τη διαμονή του. Είναι επομένως λογικό να περιμένει ότι η εμπειρία του θα ανταποκριθεί στις προσδοκίες του. Απαραίτητο στοιχείο είναι η φιλική και διακριτική επικοινωνία με τον ιδιοκτήτη του καταλύματος, ο οποίος πρέπει να είναι σε θέση να απαντά σε απορίες, να προσφέρει βοήθεια σχετικά με το κατάλυμα και την ευρύτερη περιοχή, οποιαδήποτε στιγμή χρειαστεί. Παράλληλα, η συνεχής αναβάθμιση των υπηρεσιών και η ανταπόκριση στις ανάγκες των πελατών αποτελούν κλειδί για τη δημιουργία θετικών εντυπώσεων και τη διασφάλιση της επαναλαμβανόμενης επισκεψιμότητας. Επιπλέον, η αξιοποίηση σύγχρονων τεχνολογιών για την καλύτερη διαχείριση των κρατήσεων, τη βελτίωση της εμπειρίας του πελάτη και την παροχή εξατομικευμένων προτάσεων, δύναται να βελτιώσει τη συνολική εμπειρία διαμονής. Η προσέγγιση αυτή δεν συμβάλλει μόνο στην ικανοποίηση των πελατών, αλλά και στην ενίσχυση της φήμης του καταλύματος, καθιστώντας το έναν επιθυμητό προορισμό για μελλοντικούς επισκέπτες. Για τον συγκεκριμένο σκοπό, στο πλαίσιο της διπλωματικής εργασίας θα αναπτυχθεί μια υβριδική εφαρμογή, η οποία θα αποσκοπεί στη βελτίωση της επικοινωνίας ενός τουριστικού καταλύματος με τους πελάτες του, στην προώθηση του καταλύματος και στην ενίσχυση των παρεχόμενων υπηρεσιών μέσω της συλλογής δεδομένων κατά τη διαμονή των επισκεπτών. Η εφαρμογή θα στηρίζεται στη RESTful αρχιτεκτονική και θα είναι διαθέσιμη τόσο για κινητές συσκευές Android όσο και iOS, καλύπτοντας ένα ευρύ φάσμα συσκευών από μικρού έως μεγαλύτερου μεγέθους. Για τους υποψήφιους πελάτες, η εφαρμογή θα προσφέρει πληροφορίες για το κατάλυμα και τη γύρω περιοχή, καθώς και τη δυνατότητα να εκδηλώσουν ενδιαφέρον για κράτηση επιλέγοντας τις επιθυμητές ημερομηνίες. Επιπλέον, οι πελάτες που θα έχουν ήδη κάνει κράτηση θα λαμβάνουν κατά την άφιξή τους στοιχεία σύνδεσης στην εφαρμογή, αποκτώντας πρόσβαση σε προνόμια, όπως επιπλέον πληροφορίες για το εσωτερικό του καταλύματος, ανακαλύψεις τοποθεσιών στη γύρω περιοχή μέσω δυναμικών χαρτών, πρόσβαση σε πληροφορίες σχετικές με τη διαμονή τους, επικοινωνία με εικονικό βοηθό και δυνατότητα άμεσης επικοινωνίας με τον οικοδεσπότη. Τέλος, ο ρόλος του διαχειριστή θα προσφέρει τη δυνατότητα προσθήκης δυναμικού περιεχομένου στην εφαρμογή, διαχείρισης χρηστών, ενημέρωσης της διαθεσιμότητας και διαχείρισης εισερχόμενων αιτημάτων κράτησης. Ο διαχειριστής θα έχει επίσης τη δυνατότητα να συμμετέχει στη ζωντανή συνομιλία με τους επισκέπτες και να αποκτά πρόσβαση σε χρήσιμα στατιστικά στοιχεία, τα οποία μπορούν να συμβάλουν στη συνεχή βελτίωση των υπηρεσιών καθώς και των παροχών που προσφέρει το ίδιο το κατάλυμα.

Λέξεις Κλειδιά: Τουρισμός, Εξυπηρέτηση Πελατών, Υπηρεσίες Διαδικτύου, REST APIs, , Web Sockets, Node.js, Express, Cross Platform Development, React Native, iOS, Android, Heroku, GCP

Contents

List of Figures

1 Introduction

Villa Agapi is a tourist accommodation located in Greece, specifically on the island of Crete, with over 20 years of experience in the tourism industry. Every year, it welcomes hundreds of guests, offering high-quality services. To keep up with technological advancements, the host recognized the need for an app that not only promotes the property but also enhances the guest experience. Given that the host already collaborates with companies managing bookings, the primary purpose of the app is to serve guests after they have made their reservations, with most guests learning about its availability upon arrival. Since travelers often do not have access to their personal computers while on vacation but rarely part with their mobile devices, priority was given to creating a mobile-friendly solution. Additionally, since guests come from various countries, providing multi-language support is essential to ensure a smooth and personalized experience for everyone.

Considering the dominance of iOS and Android in the mobile market, the app should support both platforms, including older devices and various screen sizes. To achieve this, a hybrid mobile application was built using React Native, allowing it to run smoothly on both iOS and Android from a single codebase.

To further enrich the app's features, such as dynamic maps and virtual assistant services, the Google Cloud Platform was integrated, offering support for these functionalities. As the app is dynamic, a server was created using Node.js with the Express library, allowing guests to access Villa Agapi's resources through REST APIs. The relational database PostgreSQL was chosen to store essential guest and service data. To facilitate real-time communication, Socket.io was used alongside Node.js and Express, providing live updates and event-driven interactions. The backend infrastructure, as outlined, is hosted on the Heroku platform.

This multi-language, user-friendly app ensures that Villa Agapi can offer its international guests a seamless, technologically advanced experience, enhancing their stay and overall satisfaction.

2 RESTful Web Services

2.1 Introduction to Web Services

Every day a user gains access to a huge amount of information, whether by opening an app on their phone or visiting a website through a web browser. Without actively taking any steps, the user obtains information provided by various services, raising questions about the source of the information and the way it is transmitted from one end to the other.
This communication is facilitated through Application Programming Interfaces (APIs), allowing businesses to share their resources with the public and providing users, who can be anywhere in the world, access to them. The services that support this bidirectional communication via the internet are known as *web services*. As the internet is used as the medium for this information transmission, the HTTP protocol plays a crucial role in achieving this communication, serving as the foundation for the *REST* (Representational State Transfer) architecture, a type of web service used in developing web applications for exchanging information between client and server.

2.2 APIs

APIs (Application Programming Interfaces) are a set of rules and protocols that facilitate communication between two software components. They provide a software-to-software interface defining the contract for applications to communicate with each other without requiring user interaction. Each transaction involving APIs consists of three main parts:

- Sending a request
- Processing the request
- Returning a response

APIs enable seamless interaction between different software systems, allowing them to exchange data and functionalities efficiently.

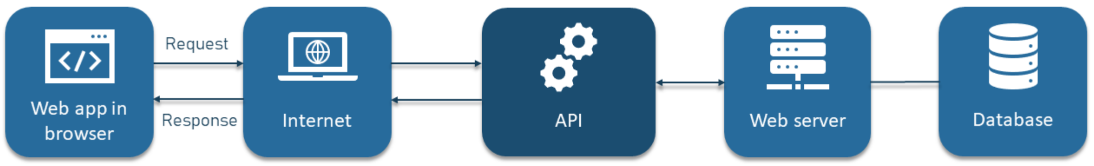


Figure 1 - API Transaction

2.3 Web Services

A *Web Service* is a software system or component designed to support machine-to-machine interaction over a network. In simple terms, it is an API that utilizes the internet for data transmission, encapsulating the following assertion:

- Every web service implements an API.
- However, not every API necessarily constitutes a web service.

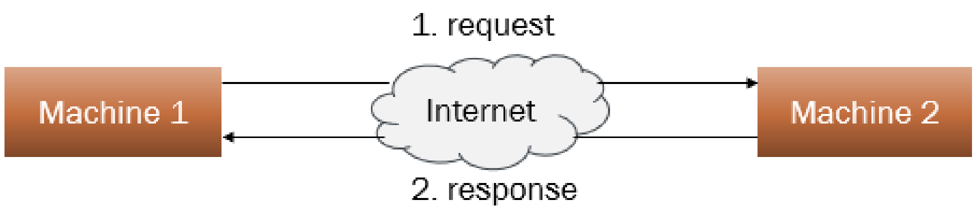


Figure 2 - Web Services

2.4 HTTP

HTTP (Hypertext Transfer Protocol) forms the cornerstone of the World Wide Web, facilitating the retrieval and display of web pages through hyperlinks [4]. Operating at the application layer of the *OSI* (Open Systems Interconnection) model, it enables the seamless exchange of information between interconnected devices across the internet. This protocol establishes the rules and conventions governing communication between web browsers and servers, ensuring data transfer and user interaction with online resources.

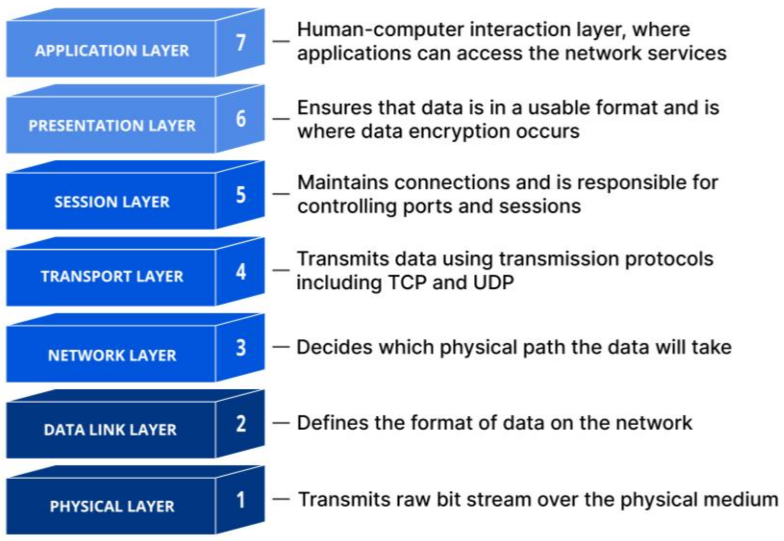
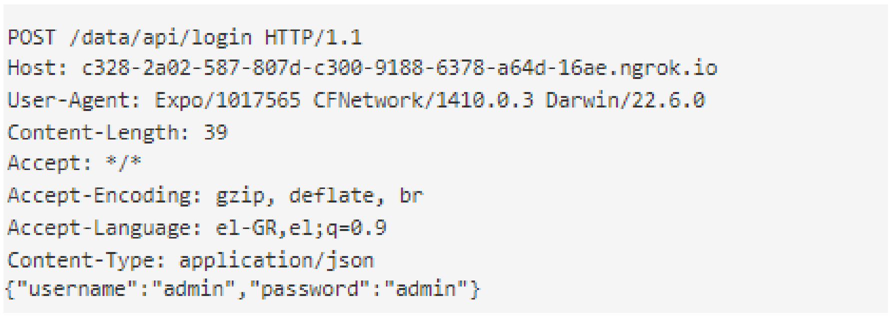

APPLICATION LAYER 7 — Human-computer interaction layer, where applications can access the network services
PRESENTATION LAYER 6 — Ensures that data is in a usable format and is where data encryption occurs
SESSION LAYER 5 — Maintains connections and is responsible for controlling ports and sessions
TRANSPORT LAYER 4 — Transmits data using transmission protocols including TCP and UDP
NETWORK LAYER 3 — Decides which physical path the data will take
DATA LINK LAYER 2 — Defines the format of data on the network
PHYSICAL LAYER 1 — Transmits raw bit stream over the physical medium

Figure 3 - OSI Model

2.5 HTTP Request

An HTTP request defines how a client, such as a web browser, requests access to information from a server. A properly formatted HTTP request consists of:

- The request line
- HTTP headers
- The body



```
POST /data/api/login HTTP/1.1
Host: c328-2a02-587-807d-c300-9188-6378-a64d-16ae.ngrok.io
User-Agent: Expo/1017565 CFNetwork/1410.0.3 Darwin/22.6.0
Content-Length: 39
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: el-GR,el;q=0.9
Content-Type: application/json
{"username":"admin","password":"admin"}
```

Figure 4 - HTTP Request

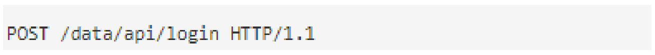
2.6 HTTP Request Line

The Request line typically consists of three parts:

- HTTP method: A command that informs the server of the action expected to be performed.
- URL path: The path identifying the server's resource (e.g., /data/api/login).
- HTTP version number: The version of the HTTP protocol specification that the request follows.

In addition, a request line may or may not include:

- Query string: Additional text information managed by the server, following the path and separated by the '?' symbol.
- Scheme and host components of the URL: Known as an absolute URI, usually used when the request needs to pass through a proxy server.



```
POST /data/api/login HTTP/1.1
```

Figure 5 - Request Line

2.7 HTTP Methods

The HTTP method defines the action expected to be performed by the server:

- GET: Exclusively used for retrieving data from a server using the request's URI as an identifier. It can include parameters in the URL (as part of the path or query string) or headers but not a body. Considered a safe method as it retrieves data without modifying the server's resource in any way. Upon successful completion, GET typically returns data in JSON or XML format to the client, as described by the accept header set by the client. Additionally, incoming requests may or may not include additional headers such as *If-Modified-Since*, *If-Range* and *If-Match*, making GET a conditional method. The server responds only when the conditions described in the request are satisfied, reducing unwanted network usage. Upon success, the GET method returns a status code 200 (OK), while errors may result in:
    - 400 (Bad request): Indicates an incorrect request.
    - 401 (Unauthorized): Indicates the user is not authenticated.

    - 404 (Not found): Indicates the requested resource does not exist, the path specified in the request does not correspond to an existing server resource.
- POST: This method allows headers and a body. It is typically used in two scenarios:
    - Creating data within a collection (database).
    - Executing business logic described within server controllers. Input to these functions usually comprises data included in the request body. Upon success, the server typically responds with status code 201 (Created), indicating data creation. In cases where no data was inserted, it may return 204 (No Content). However, POST method does not necessarily equate to data creation, it can also simply execute functionality, in which case the server may respond with 200 (OK).
- PUT: Updates an existing record. If the record described in the request does not exist, a new one is created. Known as Upsert (from update/insert), sending the entire body each time. Usually, the identifier indicating the record to update is included in the URI. The server's response depends on the action taken:
    - 201 (Created): Indicates a new resource was created.
    - 200 (OK) or 204 (No content): Indicates successful update of an existing resource.
- PATCH: Similar to PUT, but updates only parts of a resource by sending instructions on how to complete it, rather than the entire body as with PUT.
- DELETE: Used to remove data according to the URI. Upon success, returns either 200 (OK) or 204 (No content). Using status code 204 is recommended when returning the representation of the deleted resource is unnecessary, which helps avoid unnecessary network congestion and improves performance.
- HEAD: Similar to GET, but only returns response lines and headers. It does not transfer entity data, only metadata, thereby reducing network bandwidth usage. Often used for checking purposes after modifications and for verifying link validity and accessibility.
- OPTIONS: Determines the capabilities of an HTTP server, including supported headers for interacting with a specific resource. Although OPTIONS does not perform CRUD (Create, Read, Update, Delete) operations, it provides the client with information on how to interact with the specific resource. The client can specify a URL address for the method, allowing it to refer to a particular resource. An asterisk (*) can be used when the client wishes to obtain information about the entire server.

2.7.1 Idempotent and Safe Methods

Some HTTP methods can be called multiple times without causing any issues, always returning the same result and without producing unintended side effects, while others cannot. This categorization leads to defining methods as Idempotent and/or Safe:

- Idempotent: Characterized by the fact that the method, when called multiple times, always results in the same outcome and does not produce side effects.
- Safe: Characterized by the fact that it does not modify the state of a resource.

HTTP Method	Idempotent	Safe Method
GET	YES	YES
HEAD	YES	YES
OPTION	YES	YES
DELETE	YES	NO
PUT	YES	NO
PATCH	NO	NO
POST	NO	NO

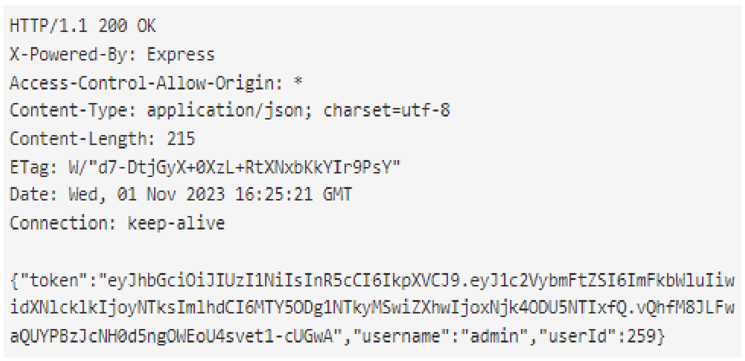
2.8 HTTP Response

An HTTP response is sent from a server to a client with the purpose of providing the requested resource, confirming the completion of a requested action, or notifying about an error encountered while processing the request consists of:

- The status line
- HTTP Headers
- The body

The status line on the other hand comprises three parts:

1. HTTP Version
2. Status Code
3. Reason Phrase



```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Content-Type: application/json; charset=utf-8
Content-Length: 215
ETag: W/"d7-DtjGyX+0XzL+RtXNxbKkYIr9PsY"
Date: Wed, 01 Nov 2023 16:25:21 GMT
Connection: keep-alive

{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiw
idXNlcklkIjoyNTksImlhdCI6MTY5ODg1NTkyMSwiZXhwIjoxNjk4ODU5NTIxfQ.vQhfM8JLFw
aQUYPBzJcNH0d5ngOWEoU4svet1-cUGwA","username":"admin","userId":259}
```

Figure 6 - HTTP Response

2.9 HTTP Status Code

HTTP defines 40 different status codes through which the client is informed about the outcome of its request. These are categorized into the following groups:

- 1xx Informational
- 2xx Success
- 3xx Redirection
- 4xx Client Error
- 5xx Server Error

Below is a table featuring commonly used status codes [1]:

Status Code	**Reason Phrase**	**Description**
200	OK	Indicates that the request has been processed successfully.
201	Created	Indicates that the request has been processed and a new resource has been created successfully.
202	Accepted	Indicates that the request has been received by the server and is being processed asynchronously.
204	No Content	Indicates that the response body has been purposely left blank.
301	Moved Permanently	Indicates that a new permanent URI has been assigned to the client's requested resource.
303	See Others	Indicates that the response to the request can be found in a different URI.
304	Not Modified	Indicates that the resource has not been modified for the conditional GET request of the client.
307	Use Proxy	Indicates that the request should be accessed through a proxy URI specified in the Location field.
400	Bad Request	Indicates that the request had some malformed syntax error due to which it could not be understood by the server. Probable reason is missing mandatory parameters or syntax error.
401	Unauthorized	Indicates that the request could not be authorized, possibly due to missing or

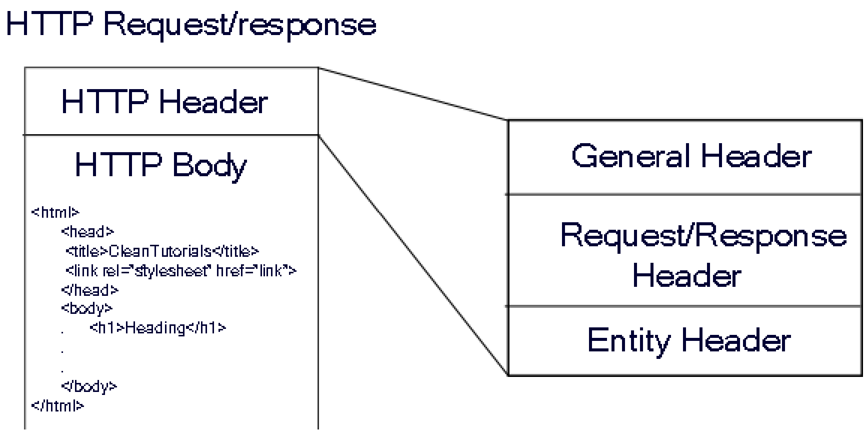
		incorrect authentication token information.
403	Forbidden	Indicates that the request was understood by the server but it could not be processed due to some policy violation or the client does not have access to the requested resource.
404	Not Found	Indicates that the server did not find anything matching the request URI.
405	Method Not Allowed	Indicates that the method specified in the request line is not allowed for the resource identified by the request URI.
408	Request Timeout	Indicates that the server did not receive a complete request within the time it was prepared to wait.
409	Conflict	Indicates that the request could not be processed due to a conflict with the current state of the resource.
414	Request URI Too Long	Indicates that the request URI length is longer than the allowed limit for the server.
415	Unsupported Media Type	Indicates that the request format is not supported by the server.
429	Too Many Requests	Indicates that the client sent too many requests within the time limit than it is allowed to.
500	Internal Server Error	Indicates that the request could not be processed due to an unexpected error in the server.
501	Not Implemented	Indicates that the server does not support the functionality required to fulfill the request.
502	Bad Gateway	Indicates that the server, while acting as a gateway or proxy, received an invalid response from the back-end server.
503	Service Unavailable	Indicates that the server is currently unable to process the request due to temporary overloading or maintenance of the server. Trying the request at a later time might result in success.

504	Gateway Timeout	Indicates that the server, while acting as a gateway or proxy, did not receive a timely response from the back-end server.

2.10 HTTP Headers

HTTP headers are used to transmit additional information between the client and server via request and response headers, respectively. They are further categorized into four groups:

- Entity headers: Contain metadata regarding the body of resources, such as Content-Length.
- General headers: Provide information applicable to both requests and responses, such as connection details.
- Client request headers: Included only in requests sent to the server. Information related to authorization, encoding and preferred language of the client are included in this category.
- Server response headers: Included only in responses sent from the server back to the client. Information such as the age of the response and ETag for caching purposes are characteristic examples.



```
<html>
      <head>
       <title>CleanTutorials</title>
       <link rel="stylesheet" href="link">
      </head>
      <body>
      .     <h1>Heading</h1>
      .
      .
      </body>
</html>
```

Figure 7 - HTTP Headers

2.11 HTTP Body

The HTTP body consists of data sent either when a client makes a request or when a server responds. It can be an image or text file, a video, or simply text. When sending text data, proper formatting is required for it to be successfully transmitted and processed by the server. The most widely used text formats include [5]:

- JSON
- XML

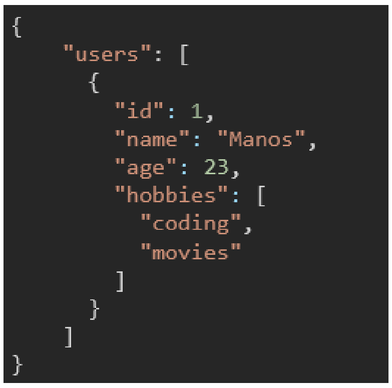
2.11.1 JSON

JSON (JavaScript Object Notation) stands out as a lightweight and flexible data exchange format. It is commonly used in web services over the HTTP protocol. JSON is human-readable and easy for machines to parse and generate. Although derived from a subset of the JavaScript programming

language, JSON is independent from JavaScript itself, relying instead on conventions widely adopted in programming communities such as C#, C++, Java, Python and others.
Structurally, JSON revolves around two fundamental elements:

- A collection of name/value pairs, records and structures.
- An ordered list of values, represented as arrays, vectors, lists, or sequences.

In JSON, an object contains an unordered set of name/value pairs enclosed in curly braces '{}' with each pair defined by a colon ':' and separated by commas ',' When using JSON as a text format in an HTTP request, the header should be appropriately set to Content-Type: application/json.



```json
{
    "users": [
      {
        "id": 1,
        "name": "Manos",
        "age": 23,
        "hobbies": [
          "coding",
          "movies"
        ]
      }
    ]
}
```

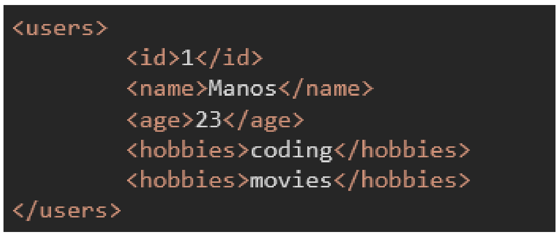
Figure 8 - JSON example

2.11.2 XML

XML (eXtensible Markup Language) serves as a flexible and extensible markup format for data exchange. Renowned for its human-readable nature, XML facilitates both manual and automated processing. In contrast to JSON, it is not tied to any specific programming language and provides a general way to structure and organize data. It has gained prominence as a W3C standard and is widely used across various domains, including network services, configuration files and data representation.

XML structures data using tags enclosed in angle brackets '<>' to form elements. These elements can have attributes and contain nested elements, enabling hierarchical representation and parent-child relationships. While JSON emphasizes simplicity and ease of use, XML's strength lies in its flexibility and ability to represent complex hierarchical structures in a standardized and structured manner.

Whether used for configuration files or data exchange between heterogeneous systems, XML remains a robust and adaptable choice in the realm of markup languages.



```xml
<users>
        <id>1</id>
        <name>Manos</name>
        <age>23</age>
        <hobbies>coding</hobbies>
        <hobbies>movies</hobbies>
</users>
```

Figure 9 - XML example

2.12 REST API

The term REST (Representational State Transfer) and its associated principles were introduced by Roy Fielding in his doctoral dissertation titled "Architectural Styles and the Design of Network-based Software Architectures" [3] presented in 2000. Fielding was among the authors of the HTTP/1.1 specification and played a pivotal role in shaping the World Wide Web as it exists today. REST is a widely adopted architectural style for designing web applications. It leverages the HTTP protocol, applying defined headers and employs four fundamental functions: Create, Read, Update, Delete (CRUD). These correspond to HTTP methods and align with actions that can be performed on a database system.

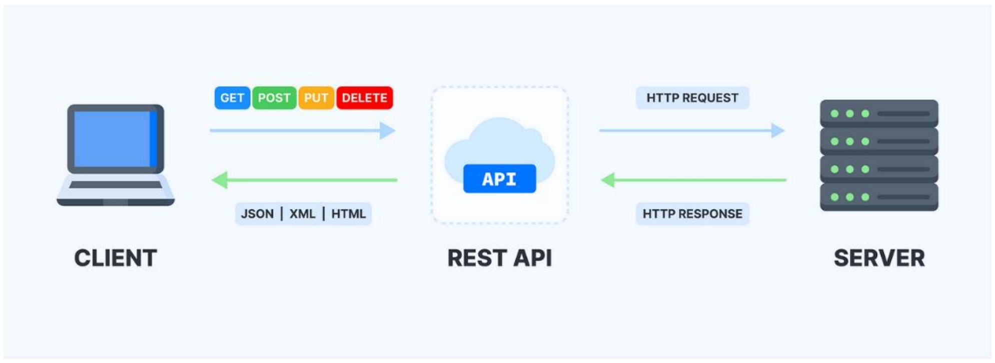


Figure 10 - REST API

2.13 REST API Design Patterns

There are some best practices to follow when designing a REST API [6] some of which are listed below:

- Accepting and Responding with JSON: JSON format has become standard for data transfer, compatible with nearly all internet technologies. On the client side, JavaScript includes built-in methods like the fetch API for encoding and decoding JSON. Similarly, server-side technologies have libraries for JSON decoding, simplifying data handling without extensive conversions. While other data transport methods exist, each with their own advantages, XML, for instance, can replace JSON but lacks broad framework support without data transformation into a more usable format, often ending up as JSON. Conversely, JSON's versatility and readability make it the standard choice. The difficulty of handling XML data from the client side, especially in browsing programs, makes it a challenge to transmit data normally. Additionally, JSON's smaller size in comparison to XML results in faster data transmission rates improving overall performance. Finally, to ensure that the response of the REST API application is interpreted as JSON by the clients, the correct specification of the Content-Type in the response header is required. After the submission of the request, the Content-Type must be set to “application/json”. This parameter informs clients that the response includes data in JSON format, allowing them to correctly interpret and manage the information.

- Using HTTP Methods for CRUD Operations: It is crucial to utilize well-known HTTP protocol methods such as GET, POST, PUT, DELETE for CRUD operations, rather than corresponding functions.
- Implementing Nested Endpoints: To achieve better organization, grouping requests containing related information is recommended. Therefore, when an object can involve an additional object, a corresponding endpoint must be designed for that scenario.
- Proper Error Handling: The REST API should be designed to prevent and manage errors correctly. In case an error is identified, an appropriate message should be returned, along with the corresponding status code following the HTTP protocol rules (400, 404, 500, etc.).
- Emphasizing Security Practices: Information exchanged between client and server must be confidential, as private data is usually sent and received. Therefore, the use of SSL/TLS is considered necessary. Additionally, measures must be taken to ensure that each user receives only the necessary information. This is achieved by establishing roles among users or adding authentication to requests.
- Temporary Data Storage (Caching) for Performance Improvement: Data retrieval from local cache is possible instead of constantly querying the database whenever data for a user needs to be retrieved. By using local memory, users receive their data more quickly, although caution is required as data may not be correctly updated (outdated).
- Adherence to Naming Best Practices: Some certain informal rules should be followed [7] when naming REST APIs, some of which are outlined below:
    - Avoid the use of capital letters in URIs.
    - Use a forward-slash (/) in the URI path to indicate hierarchical relationships between resources. For example, if we want to expose the hobbies of a specific user, a URI like /users/{id}/hobbies could be used.
    - Avoid using a trailing forward-slash in the URI to prevent confusion for users.

```
/users/{id}/hobbies/               //bad practice

/users/{id}/hobbies                //good practice
```

- Prefer the use of hyphens (-) instead of underscores (_) when combining words in the URI. This practice improves the readability of URIs.

```
application_users/{id}/favorite_colors        //bad practice

application-users/{id}/favorite-colors        //good practice
```

- Avoid using file extensions in URIs, as they affect the clarity of requests, increase the length of URIs unnecessarily and generally do not provide any advantage.

```
application-users/{id}/favorite-colors/list.json       //bad practice

application-users/{id}/favorite-colors/list            //good practice
```

- Avoid using CRUD operations in URI names. The purpose of each URI is to identify the resources offered, not to describe the expected action. This purpose is fulfilled by the HTTP methods themselves.

```
HTTP GET data/api/users/{id}                    //get a specific user

HTTP PUT data/api/users/{id}                    //update a specific user

HTTP DELETE data/api/users/{id}                 //delete a specific user
```

- Use query parameters for filtering and controlling collections. Often, only specific information from a resource is required. Therefore, it is reasonable to perform sorting, pagination and filtering actions using query parameters instead of composing new requests.

```
data/api/users                          //get all users

data/api/users?country=GREECE           //get only users from Greece
```

2.14 API Security

The resources provided by servers through APIs may be accessible to the general public or restricted to those with appropriate authorization. To appropriately restrict access, APIs implement identity verification (authentication) and permission granting mechanisms (authorization).
Different security threats to APIs can be categorized into the following categories:

- Authentication
- Authorization
- Message or content-level attacks
- Man-in-the-middle attack
- DDoS attacks (distributed denial-of-service)

2.15 Authentication & Authorization

The concepts of authentication and authorization, although related, are often confused. Authentication determines the identity of a user attempting to access a server resource, whereas authorization defines the permissions and access level of the user making a request. For example, in the scenario where someone wants to attend a concert. If they are not certified, entry will be denied. Therefore, they must obtain a ticket by going to the box office and presenting their ID to verify their details, which may include age restrictions. This process is called authentication, where the user must prove their identity to gain access to a resource (entry to the concert). Upon successful verification, they receive a ticket, which the holder can use to enter the concert. This process is called authorization. It's worth noting that the ticket does not display personal details of the holder, such as name or age, it simply confirms that the holder now has access to the resource.
Two well-known and commonly used categories for authentication and authorization are API Keys and Username & Password.

2.15.1 API Keys

An API key is used to identify the application consuming an API. API keys provide a simple authentication mechanism for applications, allowing an API to determine which applications are using it. They consist of a series of random characters and numbers provided by the API provider and is unique to each user (application). When a developer decides to integrate an API into their application, they receive the key generated by the provider and incorporate it into their code, including it in the authorization headers of each request sent to the service. This way, the provider can verify if the request for resources is valid and respond accordingly.

2.15.2 Basic Authentication

Username and password authentication is the most common form of authentication. In this method, the client presents the server with a unique username and a secret password. The server compares the received credentials with those stored in its database and grants access to resources only after successful authentication and authorization

For a request to a REST API, the client can send these credentials through the request headers using the Basic Authentication Scheme:

- The username and password are combined with a colon in between: *username:password*.
- The resulting string is encoded using Base64.
- The authorization method and a space (Basic) are prefixed to the encoded string in the "Authorization" header.

This way, the server can verify the credentials and respond accordingly.

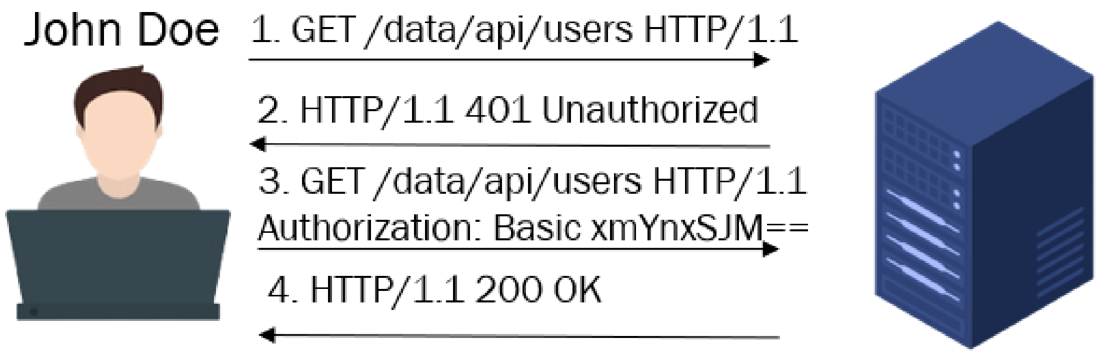

John Doe
1. GET /data/api/users HTTP/1.1
2. HTTP/1.1 401 Unauthorized
3. GET /data/api/users HTTP/1.1
Authorization: Basic xmYnxSJM==
4. HTTP/1.1 200 OK

Figure 11 - Basic Authentication Scheme

2.16 CORS

Cross-Origin Resource Sharing (CORS) [8] is a security mechanism based on HTTP headers, allowing a server to specify any origin other than its own from which a browser should permit resource loading. An example of this is XMLHttpRequest and the Fetch API, which follow the same-origin policy. For security reasons, browsers prohibit cross-origin HTTP requests initiated by scripts. This means a website using these APIs can request resources only from the same origin from which the website was loaded, unless the response from other sources includes the necessary CORS headers.

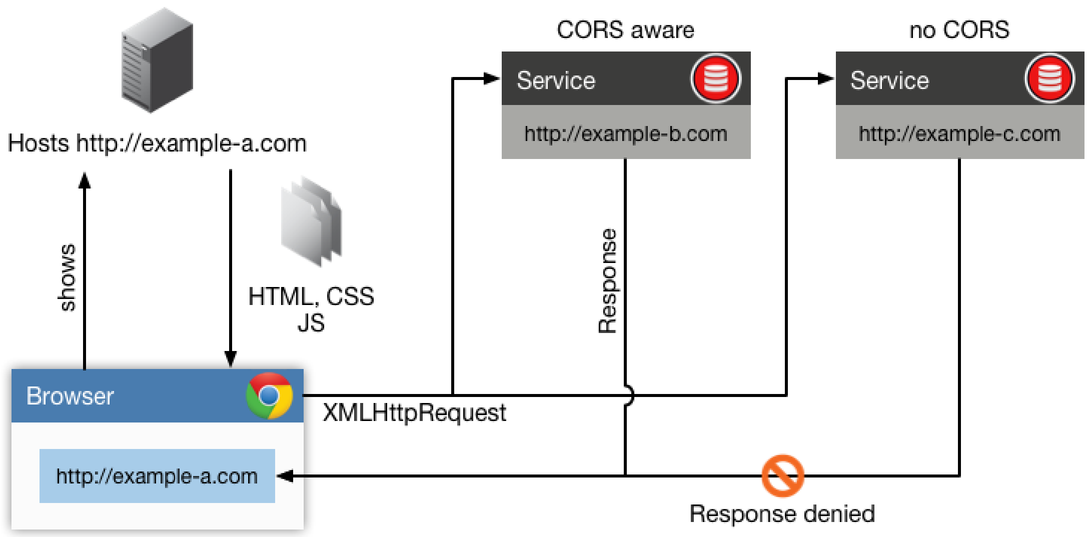


Figure 12 - CORS

3 Node.JS & Express

3.1 Introduction to Node.js

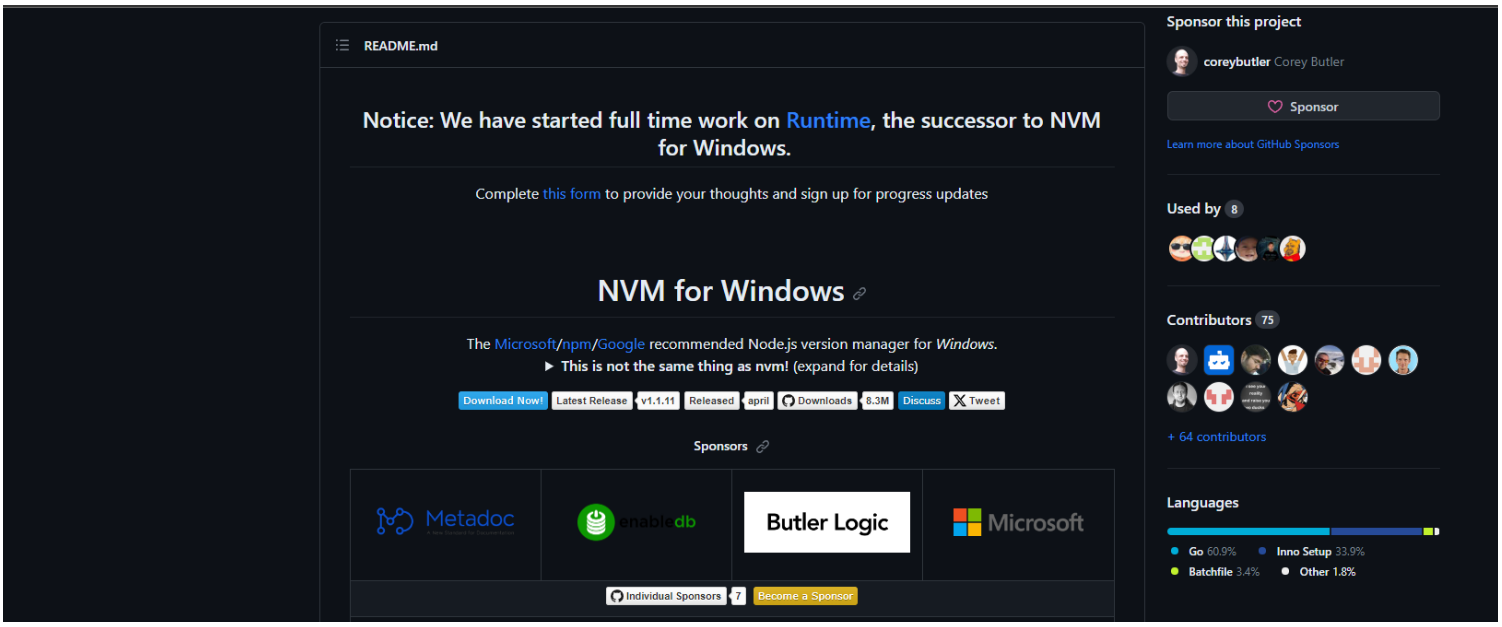
Node.js is an open-source, server-side runtime environment that allows developers to execute JavaScript code outside of a web browser. Built on Google's V8 JavaScript engine, it provides a non-blocking, event-driven architecture, making it efficient for handling I/O operations. Node.js is especially valuable for developing web applications and RESTful APIs.

It operates on a single process and is not creating a new thread for every incoming request. It features a standard library with asynchronous I/O primitives that prevent JavaScript code from blocking. When Node.js performs I/O operations such as reading from the network, accessing a database, or interacting with the filesystem, it does not block the thread or waste CPU cycles while waiting. Instead of that, Node.js continues to handle other tasks and resumes the operation once the response is received. This approach enables Node.js to manage thousands of concurrent connections with a single server efficiently, eliminating the complexities and potential bugs associated with managing thread concurrency.

3.2 Setting Up Node.js Environment

To set up a Node.js environment, both Node.js and npm (Node Package Manager) are required [9]. npm is the default package manager for Node.js and plays a crucial role in its ecosystem by managing and distributing JavaScript libraries and tools. It allows developers to install and manage packages required for Node.js applications and define project dependencies through a package.json file. Additionally, npm enables developers to define and execute scripts for common tasks, such as testing, building and deploying applications. These scripts, specified in the package.json file, can be run using npm commands.

Node.js and npm can be installed using various methods, but the recommended approach is through nvm (Node Version Manager). Nvm is a tool specifically designed to manage multiple versions of Node.js on a single machine, making it easy to switch between different versions. It can be installed from the official GitHub repository and it is compatible with all major operating systems, including Windows and macOS.



README.md

Notice: We have started full time work on Runtime, the successor to NVM for Windows.

Complete this form to provide your thoughts and sign up for progress updates

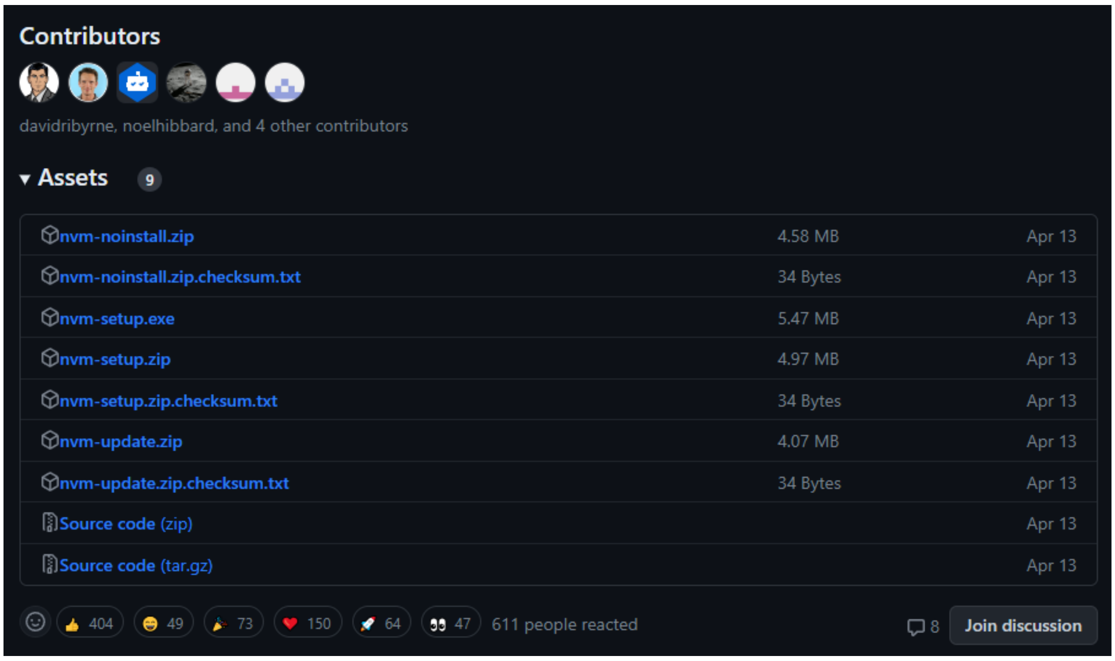
NVM for Windows

The Microsoft/npm/Google recommended Node.js version manager for *Windows*.

▸ **This is not the same thing as nvm!** (expand for details)

Figure 13 - Install NVM

After moving to the repositroy, the installer should be selected (nvm-setup.exe) for Windows:



Contributors

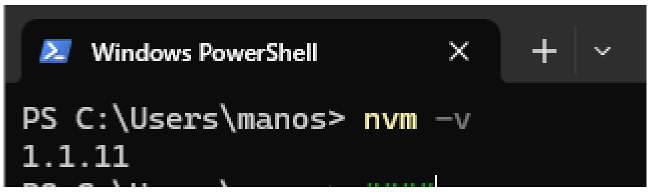
davidribyrne, noelhibbard, and 4 other contributors

▾ Assets 9

nvm-noinstall.zip	4.58 MB	Apr 13
nvm-noinstall.zip.checksum.txt	34 Bytes	Apr 13
nvm-setup.exe	5.47 MB	Apr 13
nvm-setup.zip	4.97 MB	Apr 13
nvm-setup.zip.checksum.txt	34 Bytes	Apr 13
nvm-update.zip	4.07 MB	Apr 13
nvm-update.zip.checksum.txt	34 Bytes	Apr 13
Source code (zip)		Apr 13
Source code (tar.gz)		Apr 13

Figure 14 - NVM Installer

After following the steps from the installer, the user can verify if the installation was succesfull by running the command “nvm -v”:



```
PS C:\Users\manos> nvm -v
1.1.11
```

Figure 15 - NVM version

The following NVM commands are the most useful and commonly used:

- nvm install <node version>: Install a specific version of Node.js
- nvm use <node version>: Use a specific version of Node.js
- nvm ls: See all the installed versions of Node.js
- nvm current: Check the current Node.js version

3.3 Core Concepts

- Event-Driven Architecture: Node.js is built on an event-driven architecture, allowing it to efficiently manage asynchronous operations. It continuously monitors the message queue for events and executes the associated callback functions when an event is detected.
- Non-blocking I/O: Node.js uses non-blocking I/O operations to handle multiple operations concurrently without using multiple threads. This makes it highly efficient for I/O-heavy tasks, such as serving HTTP requests or interacting with databases.
- Modules: Node.js uses a module system (CommonJS) to organize code into reusable components. Modules can be built-in (like http or fs), third-party (installed via npm) or custom. The require function is used to import modules.
- Promises and Async/Await: To manage asynchronous operations more effectively, Node.js supports Promises and the async/await syntax, providing a cleaner way to handle asynchronous operations.
- Streams: Streams are objects that enable users to read data from a source or write data to a destination continuously. Node.js offers four types of streams: Readable, Writable, Duplex and Transform. Streams are particularly efficient for managing large amounts of data.

3.4 Express

Express [10] is a flexible framework for Node.js, released as free open-source software under the MIT license. It is designed for creating web applications and APIs. Express has become one of the most popular choices for building web servers and APIs in the Node.js ecosystem. Its lightweight nature allows developers to quickly create RESTful APIs or web applications, making it an excellent tool for both beginners and experienced developers. A user can download the library by running the command "npm install express".

3.4.1 Middleware

Middlewares are software components that act as intermediaries between clients and data, implementing business logic rules. In the context of web services, middlewares implement and expose API methods that are used in HTTP requests, providing access to the server's request (req) and response (res). Middlewares can perform the following actions:

- Execute code
- Modify requests and responses
- End the request-response cycle
- Call the next middleware

To call the next middleware, the “next()” function must be invoked.

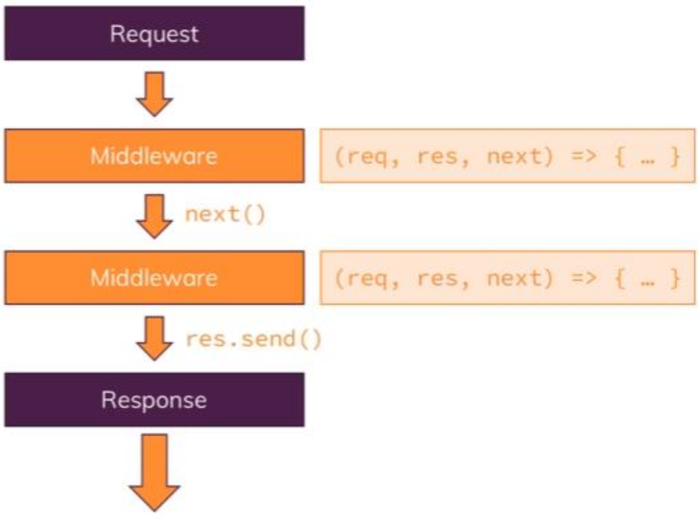


Figure 16 - Middleware

3.4.2 Routing

Routing refers to how an application decides to respond to an incoming request by defining an endpoint (PATH) and an HTTP method. Each route also includes a handler that manages the incoming request.

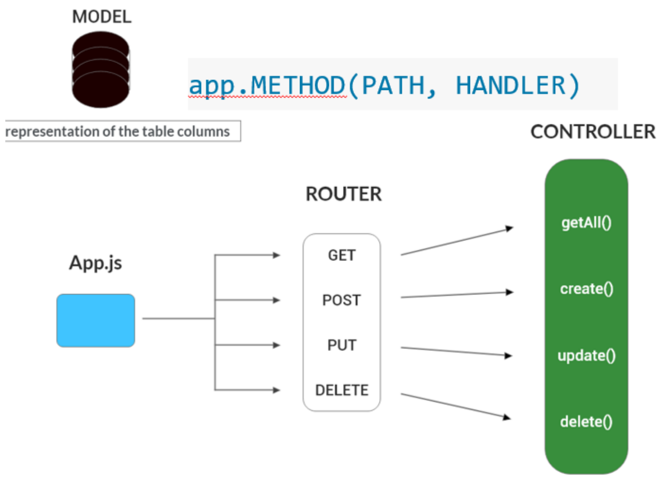


Figure 17 - Routing

3.4.3 Express Example

The process begins with importing the library into the node application, assuming it is already installed on the local machine. After that an instance of Express is created, which will act as the node server. A route is defined at the path ‘/’ using the GET method. When a request reaches this path, the respective middleware handles the request. The middleware receives information about the incoming request, such as headers and body through the req object, while the server's response corresponds to the res object.

Using the res.send() method, the server's response is sent back to the client. Finally, the application listens on port 3000 to accept incoming requests:

```
const express = require('express')
const app = express()
app.get('/', (req, res) => {
    res.send('hello world')
})
app.listen(3000)
```

3.5 Security and Authentication

In today's digital world, data privacy and security are vital. Protecting Node.js applications from vulnerabilities is crucial. Key security measures include input validation, secure data storage and defenses against common threats like SQL injection and cross-site scripting (XSS). Whether someone is developing RESTful APIs, building web applications, or managing microservices, maintaining data integrity and confidentiality through strong security practices and regular audits is essential [11].

3.5.1 Error Handling

Handling errors in REST APIs with Node.js and specially with the Express library involves implementing middleware to capture and manage exceptions. Express allows for centralized error handling by defining an error-handling middleware function that takes four parameters:

1. err
2. req
3. res
4. next

This middleware should log the error details for debugging purposes and send a standardized error response to the client, often including an appropriate HTTP status code (like 400 for bad requests or 500 for server errors) and a well described error message. Additionally, it's good practice to handle specific error types differently, such as validation errors, authentication issues, or database connectivity problems, to provide more better feedback. By effectively managing errors, developers can ensure their API is robust, user-friendly and easier to maintain:

```
app.use((err, req, res, next) => {
    res.status(err.status).send({
      error: true,
      status: err.status,
      statusDetail: err.statusDetail,
      type: err.name,
      message: err.message,
    });
});
```

3.5.2 Environment Variables

Environment variables are important for securing an application's codebase. They are specified in .env files located in the root directory of the application, with the actual values accessible only on the hosting platform. For local development, .env.sample or env.local files are typically provided in the repository, allowing developers to modify these files for testing purposes. Additionally, these variables are not exposed to the front end.

In Node.js, the dotenv package is commonly used for managing environment variables and it can be installed using the command “npm install dotenv”. After installation, the user should declare the environment variables in the .env file using the format VARIABLE_NAME='variable_value'. The following code snippet can be used to introduce one environemt variable in the code:

```
require("dotenv").config();
const variable = process.env.VARIABLE_NAME;
```

3.5.3 HTTP Headers

The default HTTP headers in Express are not very secure and usually include information that should not be exposed, like X-Powered-By. Additionally some important headers are missing and should be added to address various security aspects, including the prevention of cross-site scripting (XSS) attacks.
To enhance security, the Helmet library can be installed via npm using the command npm install helmet. Helmet secures the application by setting the most important security headers. The following code snippet can be used to introduce the helmet library:

```
const express = require('express');
const helmet = require('helmet');

const app = express();

app.use(helmet());
```

3.5.4 Rate Limiting

Rate limiting is a technique for securing backend APIs and managing traffic flow between client and server. By controlling the rate at witch user’s requests are processed, rate limiting helps prevent malicious attacks like DDoS and brute force, enusres that servers are not overloaded and maintains a smooth flow of data. This method allows developers to set specific constraints on user activity such as limiting an unsubsribed user to three password attempts during login within a defined time frame, often referred to as a “window”. Once this limit is exceeded, then any additional request is blocked, enhancing both security and performance by preventing excessive strain on the server.

A popular rate limiting package for Node.js is the “express-rate-limit”, used to limit repeated requests to public APIs and/or endpoints. It can be used globally for every request:

```
import { rateLimit } from 'express-rate-limit'
const limiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    limit: 3, // Limit each IP to 3 requests per `window`
    standardHeaders: 'draft-7',
    legacyHeaders: false,
})
// Apply the rate limiting middleware to all requests.
app.use(limiter)
```

Or explicity as a middleware for speciffic endpoints:

```
app.post('/reset_password', limiter, (req, res) => {
    // ...
})
```

3.5.5 Limit Request Size

In Node.js, the default request body size is set to 100 KB. In some cases, the server may need to accept larger files like PDF,CSV, etc. But it is important to balance this requirement with security considerations. To reduce the risk of DDoS attacks, where attackers might attempt to flood the server with excessive data, the request body size limit should be increased cautiously. This can be managed by adjusting the body-parser middleware configuration, as demonstrated below:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

// set the request size limit to 2 MB
app.use(bodyParser.json({ limit: '2mb' }));
```

3.5.6 Basic Authentication

The Basic Authentication scheme that has been discussed in the section *2.15.2 Username & Password* can be easily implemented using the Express library. This can be achieved by using middleware that checks for an authorization header in incoming requests. In the provided code snippet “require(“dotenv”).config()” is used to load environment variables from a .env file, allowing for the secure storage of sensitive information like usernames and passwords without exposing these values in the codebase.
The middleware function then checks if the Authorization header is present in the request. If the header is missing, it responds with a 401 Unauthorized

status, indicating that the client must authenticate itself to get the requested response. The WWW-Authenticate header is set to indicate that Basic Authentication is required.
If the authorization header is present, the middleware decodes the Base64 encoded username and password using Buffer.from() and splits the decoded string into its respective username and password. These credentials are then compared to the values stored in the environment variables BASIC_USERNAME and BASIC_PASSWORD.
If the credentials match, the middleware calls the next() function, allowing the request to proceed to the next middleware or route handler. If the credentials do not match, the response is once again set to 401 Unauthorized and the WWW-Authenticate header is set to indicate the requirement for Basic Authentication.

```
require("dotenv").config();

module.exports = (req, res, next) => {
  const authheader = req.headers.authorization;

  if (!authheader) {
    let err = new Error("Not authenticated!");
    res.setHeader("WWW-Authenticate", "Basic");
    err.status = 401;
    return next(err);
  }

  const auth = new Buffer.from(authheader.split(" ")[1], "base64")
    .toString()
    .split(":");
  const username = auth[0];
  const password = auth[1];

  if (
    username == process.env.BASIC_USERNAME &&
    password == process.env.BASIC_PASSWORD
  ) {
    // If Authorized user
    next();
  } else {
    let err = new Error("Not authenticated!");
    res.setHeader("WWW-Authenticate", "Basic");
    err.status = 401;
    return next(err);
  }
};
```

3.5.7 JWT

The JWT, or JSON Web Token [12], is a self-contained method for securely transmitting information between different parties in the form of a JSON object. It is commonly used for authentication and authorization on websites and API services. JWTs consist of three parts:

- Header: Specifies the algorithm used for signing (typically HS256 or RS256).
- Payload: Contains the actual data.
- Digital Signature: Ensures the integrity and authenticity of the token.

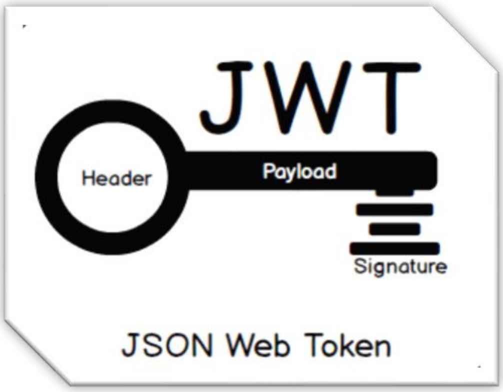


Figure 18 - JWT

JSON Web Tokens (JWTs) are a widely used method of authentication. When a user logs in, if the provided credentials are valid, a JWT is generated on the server and returned to the client. The client then presents the JWT in subsequent requests to the server to prove their previous authentication. This is usually done through a header in the format *Authorization: Bearer <JWT-token>*. As shown in the following code snippet, the data includes the username and user ID, while the digital signature is a base64-encoded value. The token's validity is set to one hour and the default algorithm is HS256 unless otherwise specified.

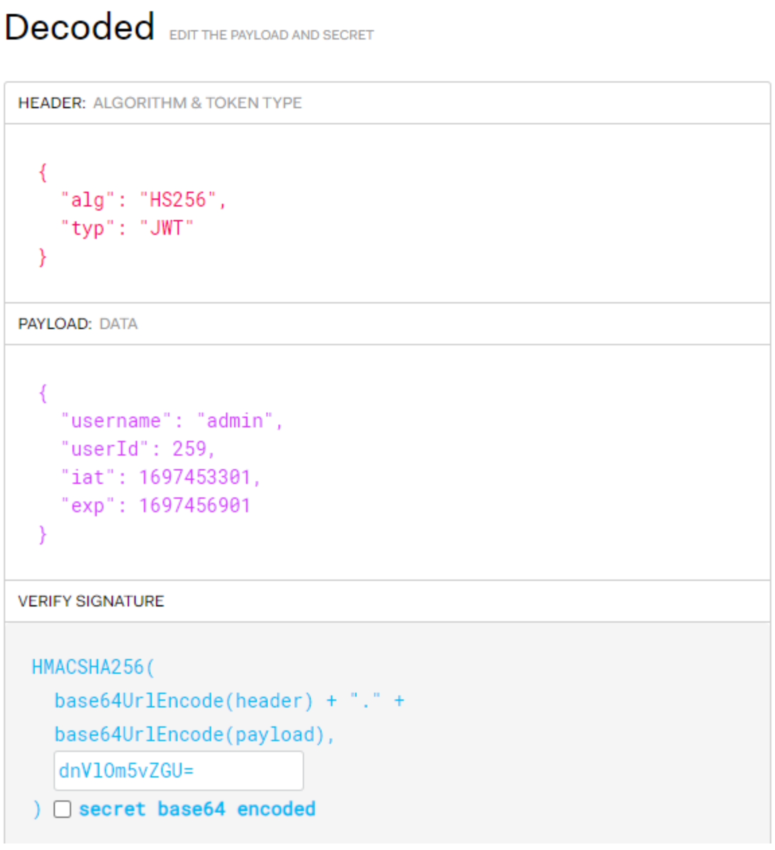
```
const token = jwt.sign(
  {
    username: username,
    userId: queryResult.rows[0].id,
  },
  process.env.JWT_PRIVATE_KEY,
  { expiresIn: "1h" }
);
```

As a result of the code above, a token in the following format is generated: *eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwid XNlcklkIjoyNTksImlhdCI6MTY5NzQ1MzMwMSwiZXhwIjoxNjk3NDU2OTAxf Q.INFr4GDoa5BpzDTFE8KBlfHp8MB1kbGSiL2wkd21E3o*

Upon successful decoding of the token [13], the following information is retrieved:

- username: The username used when the token was signed.
- userID: The ID used when the token was signed.
- iat (issued at): The timestamp of when the token was signed, in Epoch Unix Timestamp format, representing the number of seconds since January 1, 1970.

- exp (expires): The timestamp indicating when the token will expire, using the same logic as the iat field.


Decoded EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "username": "admin",
  "userId": 259,
  "iat": 1697453301,
  "exp": 1697456901
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  dnV1Om5vZGU=
) ☐ secret base64 encoded
```

Figure 19 - Decoded JWT

After a succesfull login, the user will receive a token, which will be used in subsequent requests that require authorization by adding it to the request headers as shown in the image below:

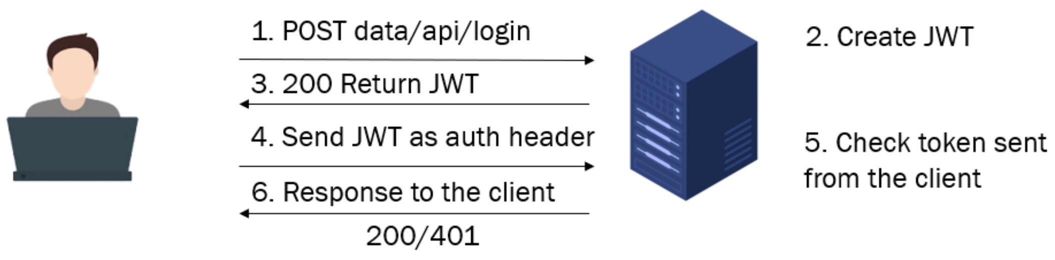

1. POST data/api/login
3. 200 Return JWT
4. Send JWT as auth header
6. Response to the client
200/401
2. Create JWT
5. Check token sent from the client

Figure 20 - JWT Use

To implement this check programmatically using the Express library, an additional middleware is required. This middleware is called before the final controller and is responsible for verifying whether the JWT received in the request matches the one that was generated when the user logged in:

```
const jwt = require("jsonwebtoken");

module.exports = (req, res, next) => {
  let token;
  if (!!req.get("Authorization")) {
    token = req.get("Authorization").split(" ")[1];
  }

  let decodedToken;
  try {
    decodedToken = jwt.verify(token, process.env.JWT_PRIVATE_KEY);
  } catch (err) {
    err.statusCode = 500;
    res.status(500).json({
      error: "Internal Server Error",
    });
  }

  if (!decodedToken) {
    const error = new Error("Not authenticated");
    error.statusCode = 401;
    throw error;
  }
  next();
};
```

3.5.8 Monitoring and Logging

Monitoring and logging are very important for maintaining the health and performance of a Node.js application built with Express. To effectively log HTTP requests and errors, the morgan logging library can be integrated into the Express app. Morgan provides real-time insights by capturing details about incoming requests, such as method, URL, status code and response time, which can be outputted to the console or saved to a log file for further analysis. After the user installs the package by running the command "npm install morgan", it can be integrated into the application as demonstrated in the code snippet below:

```
const express = require('express')
const morgan = require('morgan')

const app = express()

app.use(morgan('combined'))

app.get('/', function (req, res) {
  res.send('hello, world!')
})
```

For monitoring the server's health, the *express-status-monitor* package offers a lightweight solution. This package provides a real-time, interactive dashboard that tracks metrics like CPU usage, memory consumption, response time and throughput. After the user installs the package by running the command “npm install express-status-monitor”, it can be integrated into the application as demonstrated in the code snippet below:

```javascript
const express = require('express')
const app = express()

app.use(require("express-status-monitor")())
```

By incorporating both morgan for detailed logging and express-status-monitor for real-time monitoring, developers can ensure that their Node.js application remains stable, responsive and easy to debug:

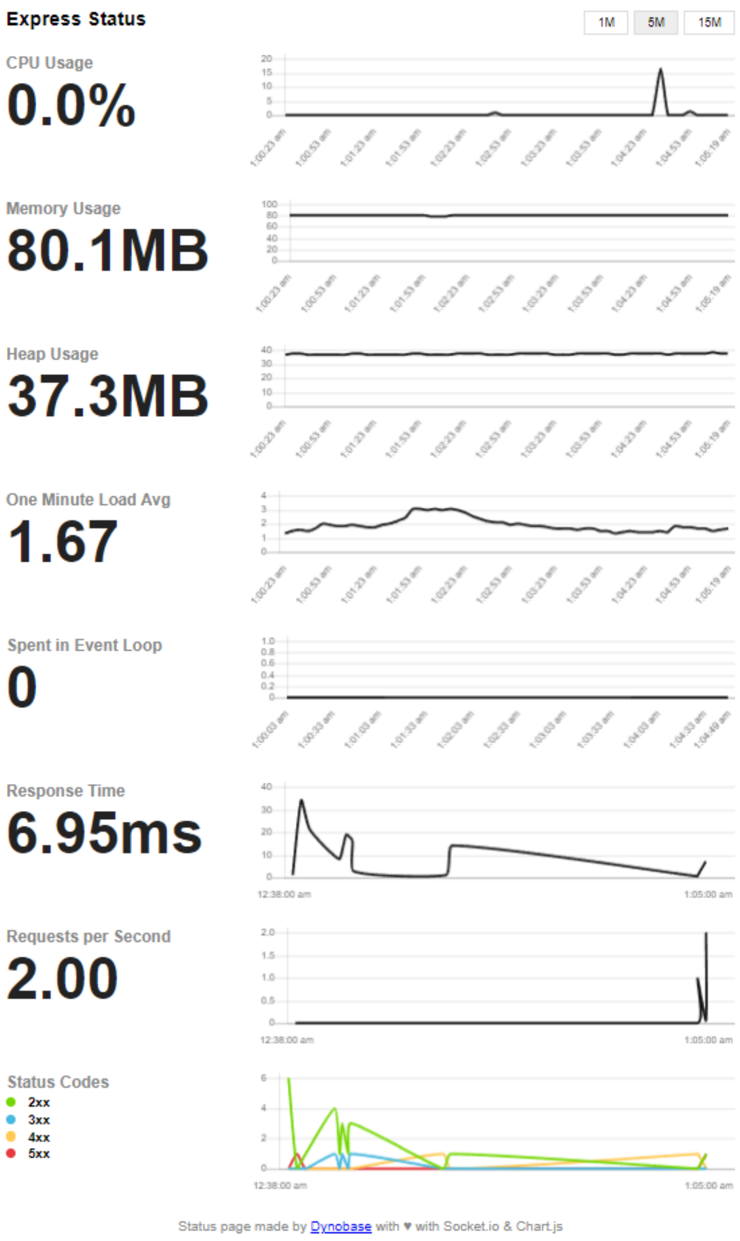


Figure 21 - Express-status-monitor

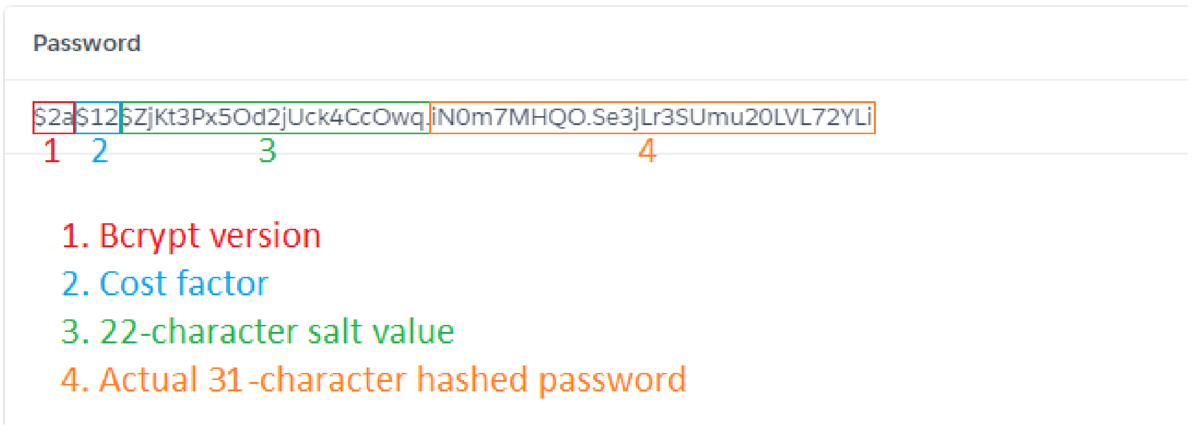
3.5.9 Password Management

Securely storing passwords and credentials in Node.js applications is essential for protecting sensitive user data. Passwords should never be stored in plain text on the server. Instead of that approach, a secure hashing algorithm should be used. These algorithms are intentionally designed to be slow and computationally intensive, making it difficult for attackers to crack passwords through brute-force or rainbow table attacks. A widely used and highly recommended algorithm for this purpose is bcrypt. For a node application, bcrypt library [14] can be installed by running the command "npm install bcrypt". Some key features of this algorithm are outlined below:

- In addition to incorporating a salt for protection against rainbow table attacks, bcrypt is an adaptive function: over time, the number of iterations can be increased to make it slower, ensuring it remains resistant to brute-force attacks
- While bcrypt.js is compatible with bcrypt's C++ bindings, it is written in pure JavaScript and is therefore approximately 30% slower. This reduction in speed effectively limits the number of iterations that can be processed in a given time frame.
- The maximum input length is 72 bytes and the generated hash values are 60 characters long.

The input to a bcrypt function consists of the password (up to 72 bytes), a cost factor and a randomly generated 16-byte salt value. Bcrypt uses these inputs to compute a 24-byte hash. The final output of the function is a string in the following format:

$2<a/b/x/y>$[cost]$[22-character salt][31-character hash]



Password

$2a$12$ZjKt3Px5Od2jUck4CcOwq.iN0m7MHQO.Se3jLr3SUmu20LVL72YLi

1. Bcrypt version
2. Cost factor
3. 22-character salt value
4. Actual 31-character hashed password

Figure 22 - Bcrypt example

Bcrypt can be easily integrated into a node.js application as demonstrated in the code snippet below:

```
const bcrypt = require("bcryptjs");

async function createUser(password) {
  return await bcrypt.hash(password, 12);
}

async function login(plainPassword, hashedPassword) {
  let match = await bcrypt.compare(plainPassword, hashedPassword);
  return match;
}

async function main() {
  let hashedPassword = await createUser("admin");
  login("password", hashedPassword); //false
  login("admin", hashedPassword); //true
}

main();
```

3.6 API Documentation & Validation

API Documentation refers to a collection that includes all the necessary information about the APIs available on the server, offering features such as:

- A list of available requests
- Request descriptions
- Execution examples (e.g., example body, example responses)
- The ability to execute requests
- Support for multiple servers (Development, UAT, Production)
- Authorization configuration where needed

An example of an API documentation tool is Swagger [15], a suite of tools used for developing and describing RESTful APIs.

3.6.1 YAML

YAML [16] is neither a programming language nor a markup language, despite its acronym standing for "Yet Another Markup Language". Instead, it is a simple format used to represent structured data, often found in configuration files:

- openapi: The current version
- info: Information about the documentation, such as title and version
- servers: The available servers
- tags: Categories used to organize requests for better management
- paths: The available requests

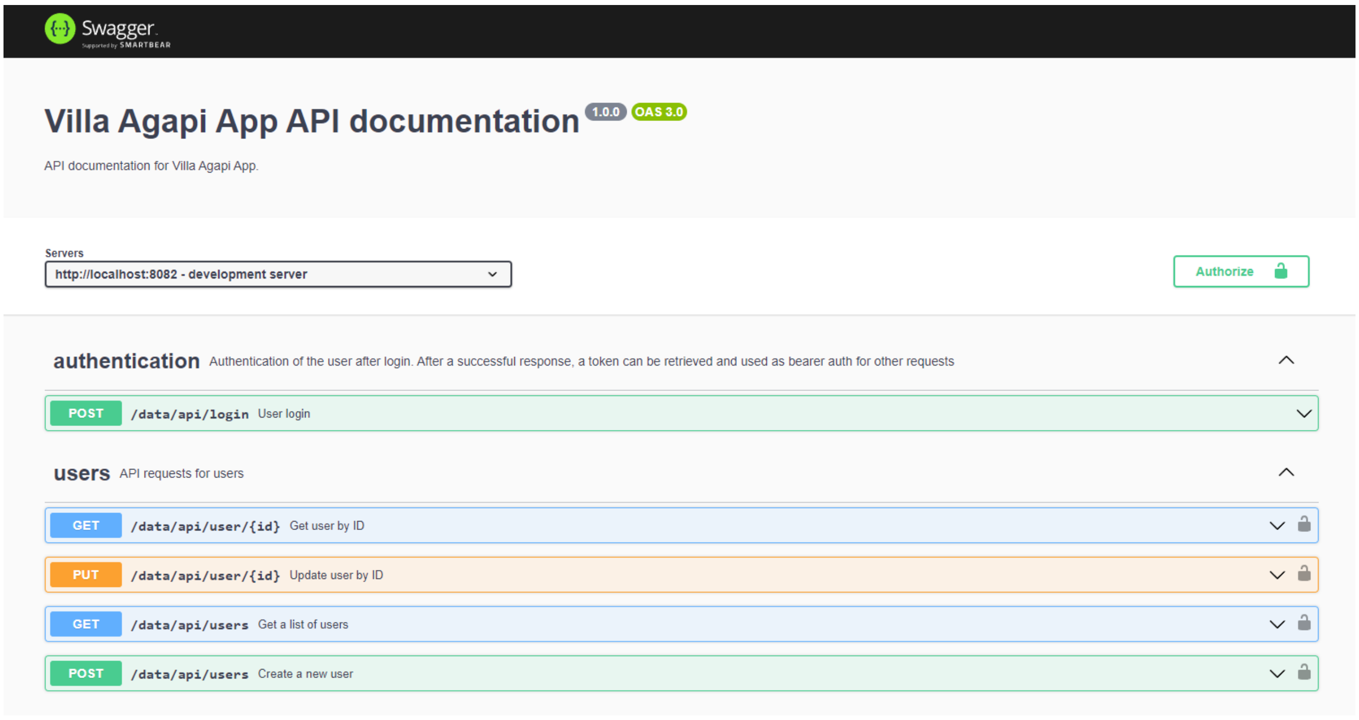
```yaml
openapi: 3.0.1
info:
  title: Villa Agapi App API documentation
  description: API documentation for Villa Agapi App.
  version: 1.0.0
servers:
  - url: http://localhost:8082
    description: development server
  - url: https://online-server.com/
    description: production server
tags:
  - name: users
    description: API requests for users
paths:
  /data/api/user/{id}:
    get:
      tags:
        - users
      summary: Get user by ID
      description: Retrieve user information by their ID.
      parameters:
        - name: id
          in: path
          description: ID of the user to retrieve.
          required: true
          schema:
            type: integer
            example: 1
      responses:
        "200":
          description: Successful response
          content:
            application/json:
              example:
                user:
                - id: 1
                  name: user
                  email: user@user.gr
                  created_date: 2023-09-01 07:06:00
                  arrival: 2023-11-08 07:40:27
                  departure: 2023-11-08 07:40:27
                  firstname: Manos
                  lastname: Karapiperakis
                  cleaningprogram: [2023-11-09, 2023-11-09]
                  phone: 90000000
                  country: Greece,
                  role: admin
```

3.6.2 Swagger-ui-express

Swagger-ui-express is a middleware that allows developers to easily integrate interactive API documentation into their Node.js applications. By using a swagger.json or swagger.yaml file, it generates a visually appealing and user-friendly interface for exploring and testing API endpoints. This live documentation is hosted on the server and can be accessed via a designated route, offering real-time interaction with the API. It supports features like request validation, response visualization and multiple authentication schemes, making it an essential tool for both development and API consumer communication. It can be easily integrated into a Node.js application, as demonstrated in the code snippet below:

```
const swaggerUi = require("swagger-ui-express");
const yaml = require("yamljs");
const swaggerDocument = yaml.load("./swagger.yaml");
const app = express().use("/data/api/doc", swaggerUi.serve, swag-
gerUi.setup(swaggerDocument));
```

If the user navigates to the specified path, they will see the result of the file displayed within a graphical interface.



Villa Agapi App API documentation 1.0.0 OAS 3.0

API documentation for Villa Agapi App.

Servers
http://localhost:8082 - development server

Authorize

authentication Authentication of the user after login. After a successful response, a token can be retrieved and used as bearer auth for other requests

POST /data/api/login User login

users API requests for users

GET /data/api/user/{id} Get user by ID

PUT /data/api/user/{id} Update user by ID

GET /data/api/users Get a list of users

POST /data/api/users Create a new user

Figure 23 - Swagger UI

3.6.3 Express-openapi-validator

Express-openapi-validator is a middleware for Express.js that helps validate API requests and responses against specific standards, typically defined in a configuration file. OpenAPI is a specification for building APIs, providing a standardized way to describe the structure and behavior of RESTful APIs.

By integrating express-openapi-validator into an Express.js application, developers can enforce validation rules defined in an OpenAPI specification (e.g. Swagger.yaml). This ensures that incoming requests conform to the specified schema and that outgoing responses match the expected format. Additionally, authentication handlers can be integrated, allowing developers to specify which requests require authentication directly within the Swagger.yaml file.
As a result, any request or response that doesn't meet the defined criteria in the documentation file will be blocked, returning an error message. This adds an extra layer of security to the server by ensuring that only compliant data is processed.

```
module.exports = async () => {
  await new OpenApiValidator({
    apiSpec: "./swagger.yaml",
    validateSecurity: {
      handlers: {
        BearerAuth: bearerAuthenticator,
        BasicAuth: basicAuthenticator
      },
    },
  }).install(app);
```

Handlers are simply functions written in JavaScript that take the request and check whether the headers are valid. The implementation of Bearer Authentication is demonstrated in the code snippet below:

```
const jwt = require("jsonwebtoken");
const { AuthError } = require("../lib/errors");

const bearerAuthenticator = (req, scopes, schema) => {
  try {
    const token = req.header("Authorization").replace("Bearer ",
"");
    req.jwtPayload = jwt.verify(token, process.env.JWT_PRIVATE_KEY);
    if (req.openapi.schema["x-acl"]) {
      const jwtScopes = new Set(req.jwtPayload.scopes);
      const reqScopes = new Set(req.openapi.schema["x-acl"]);
      const intersection = new Set(
        [...reqScopes].filter((x) => jwtScopes.has(x))
      );
      if (intersection.size > 0) return true;
      else throw new AuthError("Invalid Scopes");
    }
    return true;
  } catch (e) {
    throw new AuthError("Invalid Token");
  }
};
```

4 Socket.io

4.1 Introduction to Socket.io

Socket.IO [17] is a library that enables real-time, bidirectional communication between clients and servers. It simplifies the development of applications that require instant updates or live interaction, such as chat applications and online games.

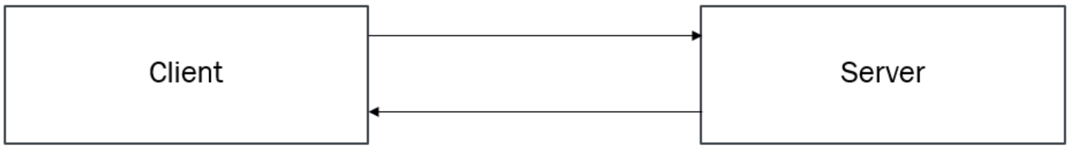


Figure 24 - Socket.io

The Socket.IO connection can be established with different low-level transports:

- HTTP long-polling
- WebSocket
- WebTransport

Socket.IO will automatically pick the best available option, depending on:

- The capabilities of the browser
- The network

4.2 Socket.io Features

Some of the key features that make Socket.IO stand out include the following:

- HTTP long-polling fallback: If the WebSocket connection fails or is interrupted for any reason, the connection will fall back to HTTP long-polling.
- Automatic reconnection: Under certain conditions, the WebSocket connection between the server and client may be lost without either side being aware. Socket.IO uses a “heartbeat” mechanism that regularly checks the connection, detects potential disconnections and automatically reconnects the user if necessary.
- Packet buffering: When the client disconnects, packets are automatically stored in a temporary cache and sent once the client reconnects.
- Acknowledgements: Socket.IO provides a flexible way to send and receive requests and events.
- Broadcasting: On the server side, an event can be broadcasted to all users or only to specific users within the same “room”.
- Multiplexing: Namespaces allow for the organization of application logic over a single shared connection.

4.3 Socket.io Architecture

The bidirectional communication channel between the Socket.IO server and the Socket.IO client is established using a WebSocket connection whenever possible, with HTTP long-polling used as a fallback.

The Socket.IO codebase is divided into two distinct layers:

- The low-level plumbing: Known as Engine.IO, it is responsible for establishing the low-level connection between the server and the client. Its responsibilities include:
    - Handling various transports (such as WebSocket and HTTP long-polling) and managing the upgrade mechanism.
    - Detecting disconnections.
- Socket.IO itself: This is the higher-level API that builds on top of Engine.IO to manage the application's communication logic.

4.3.1 Transports

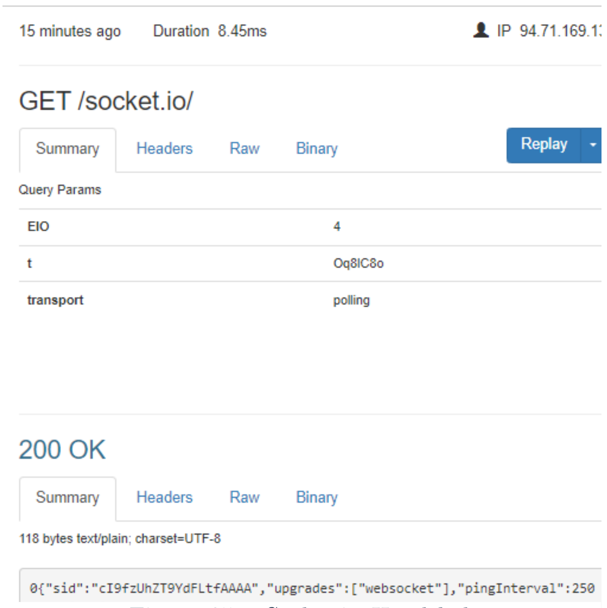
Two transport methods are implemented in Socket.IO:

- HTTP long-polling: This consists of multiple successful HTTP requests:
    - Long-running GET requests to receive data from the server.
    - Short-running POST requests to send data to the server.
- WebSocket: The WebSocket transport establishes a WebSocket connection, providing a bidirectional, low-latency communication channel between the server and the client.

4.3.2 Handshake

During the initiation of the connection, the server sends certain information as outlined below:

- sid: It represents the Session ID and must be included in the sid query parameter in all future requests.
- upgrades: This is an array containing a list of the best connection options supported by the server (e.g., websocket).
- pingInterval: Used in the heartbeat mechanism for automatic reconnection.
- pingTimeout: Also used in the heartbeat mechanism for automatic reconnection.



15 minutes ago Duration 8.45ms IP 94.71.169.1

GET /socket.io/

Query Params

EIO	4
t	Oq8IC8o
transport	polling

200 OK

118 bytes text/plain; charset=UTF-8

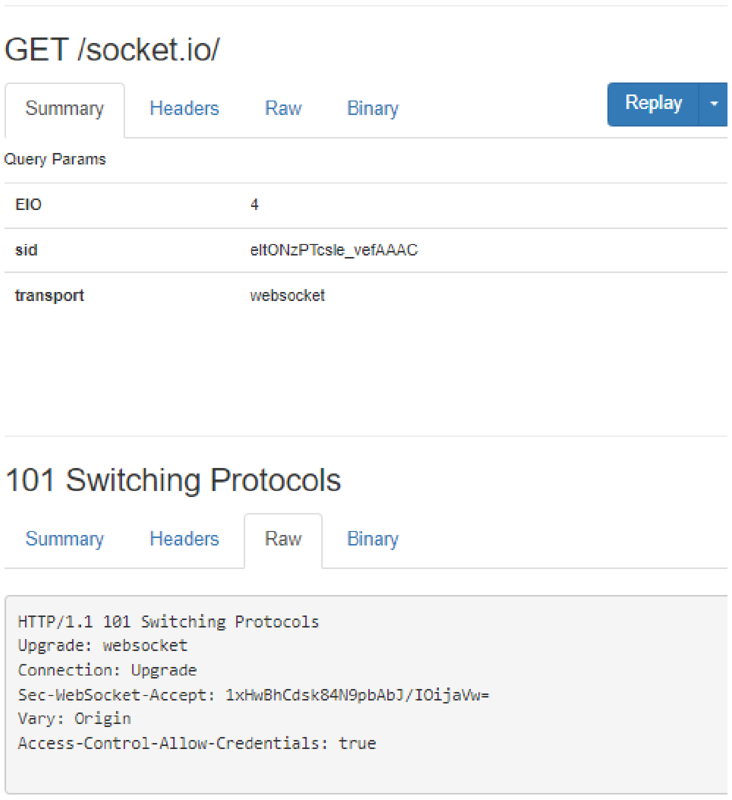
```
0{"sid":"cI9fzUhZT9YdFLtfAAAA","upgrades":["websocket"],"pingInterval":250
```

Figure 25 - Socket.io Handshake

4.3.3 Upgrade Mechanism

By default, the client establishes the connection using HTTP long-polling, as demonstrated earlier during the handshake. The upgrades array also included WebSocket as the optimal option supported by the server.
This happens because a WebSocket-based connection is not always possible. For example, it may be blocked by a firewall, antivirus software, or even not supported by the browser itself, although WebSocket is now supported by about 97% of browsers.
From the user's perspective, an unsuccessful WebSocket connection can result in up to a 10-second delay, which is undesirable. Therefore, the priority is to first create a reliable connection via HTTP long-polling, followed by upgrading to WebSocket when possible.
During the upgrade process, the client performs the following steps:

- Confirms that the outgoing buffer is empty.
- Switches the state of the current transport to read-only.
- Attempts to establish a connection using a different transport method (WebSocket).
- If successful, terminates the previous transport method (HTTP polling).



EIO	4
sid	eltONzPTcsle_vefAAAC
transport	websocket

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1xHwBhCdsk84N9pbAbJ/IOijaVw=
Vary: Origin
Access-Control-Allow-Credentials: true
```

Figure 26 - Upgrade Mechanism

4.3.4 Disconnection Detection

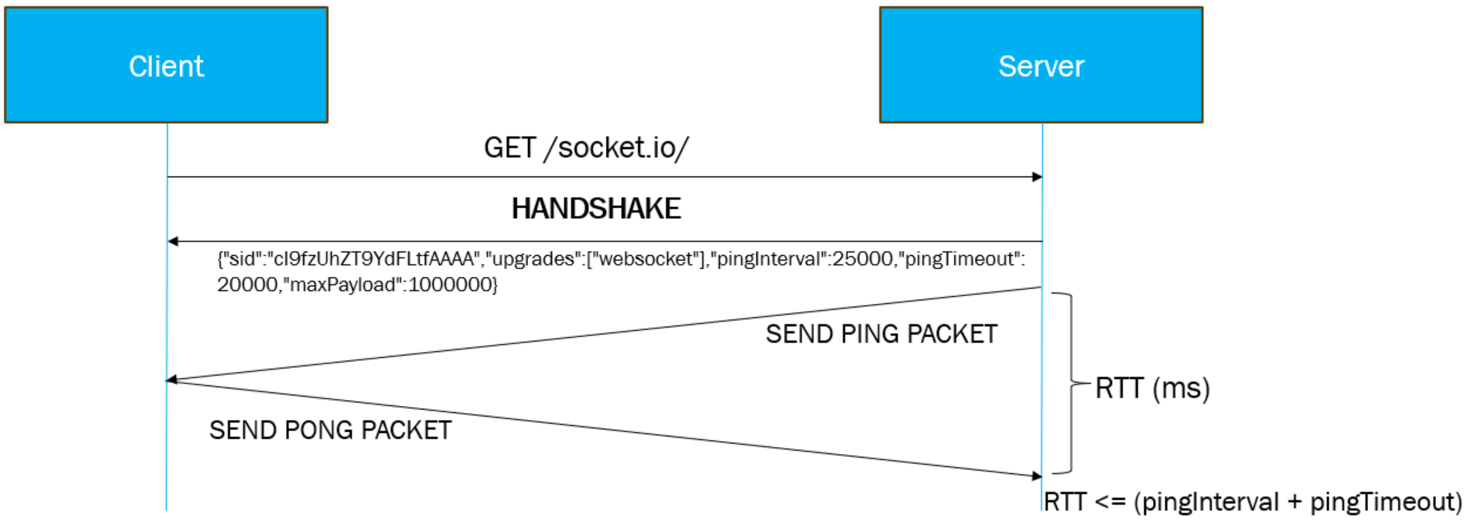
The connection is considered closed in the following scenarios:

1. An HTTP request (either GET or POST) fails.
2. The WebSocket connection is terminated, such as when the user closes the browser tab or the mobile app.
3. The socket.disconnect() method is called, either from the server or the client side.

Additionally, there is a heartbeat mechanism in place to ensure the connection between the server and client is still active:

- At regular intervals (defined by pingInterval), a value returned during the initial handshake, the server sends a PING packet to the client.
- The client then has a limited period, set by the pingTimeout value (also provided during the handshake), to respond by sending a PONG packet back to the server.
- If the server does not receive a response within the combined time frame of pingInterval + pingTimeout, it assumes the connection has been lost.

This mechanism is essential for detecting issues such as connection drops or network instability and it allows the server to handle disconnections more efficiently. For example, if a client becomes unavailable due to network issues or the user suddenly closes their browser or app, the heartbeat mechanism will detect this, ensuring that stale connections don't persist unnecessarily. This approach helps optimize resources and maintain an efficient communication flow between the client and server:



```
{"sid":"cI9fzUhZT9YdFLtfAAAA","upgrades":["websocket"],"pingInterval":25000,"pingTimeout":
20000,"maxPayload":1000000}
```

Figure 27 - Disconnection Detection

4.4 Socket.io Events

Socket.IO events are a core feature of the Socket.IO, which is used for real-time, bidirectional communication between clients (typically browsers or mobile apps) and servers. These events are used to handle and trigger specific actions or messages during a connection.

4.4.1 Built-in Events

Socket.IO provides plenty of predefined events that are useful for handling core functionality, including connection, disconnection and errors. Below are some of the most important and commonly encountered events:

- connect: Fired when a client successfully connects to the server.
- disconnect: Triggered when a client disconnects from the server.
- error: Occurs when there is an issue with the connection.
- reconnect: Triggered when the client successfully reconnects after a disconnection.

4.4.2 Custom Events

In addition to built-in events, custom events can be defined to address specific use cases. For instance, if a messaging service is needed for communication between two endpoints, the following two custom events can be implemented. This enables the creation of customized communication protocols to suit application's specific needs:

- Server-side: socket.emit('customEvent', data);
- Client-side: socket.on('customEvent', function(data) {});

Socket.IO works by emitting events and listening for them. For example, when the server or client emits an event, the other side can listen for it and perform a specific action. This follows a publisher/subscriber pattern:

- Emitting an event: socket.emit(eventName, data)
- Listening for an event: socket.on(eventName, callback)

4.5 Namespace Support

Socket.IO supports namespaces, allowing to define separate communication channels within the same application. Each namespace can have its own events.

4.6 Rooms

A room is a channel where sockets can connect and disconnect. It can be used to broadcast events to a subset of clients. To join a room, the method socket.join(room) is used, with the room name passed as an argument. This functionality is available only on the server side, but clients can also actively participate.
The process of joining a room is initiated by the client and the server can notify clients of any changes, such as when someone enters or leaves a room through an emitter.

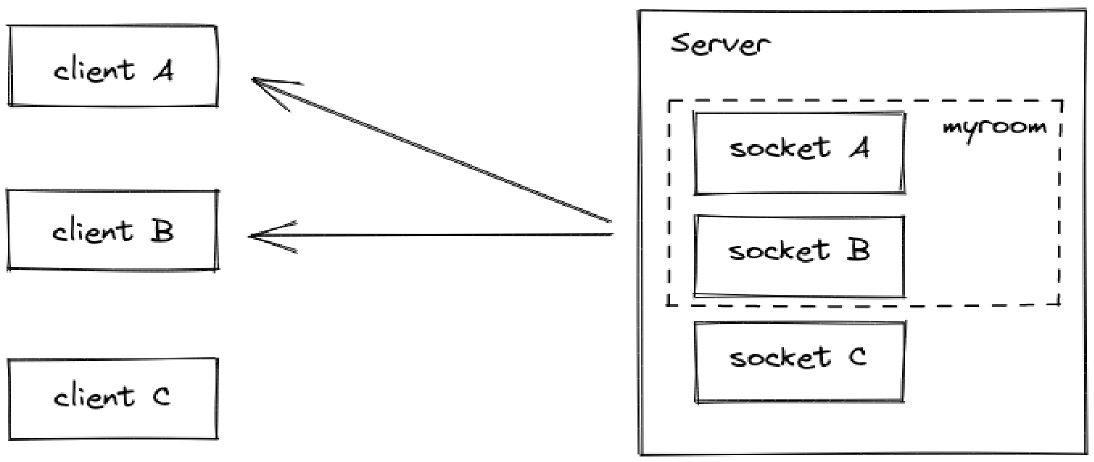
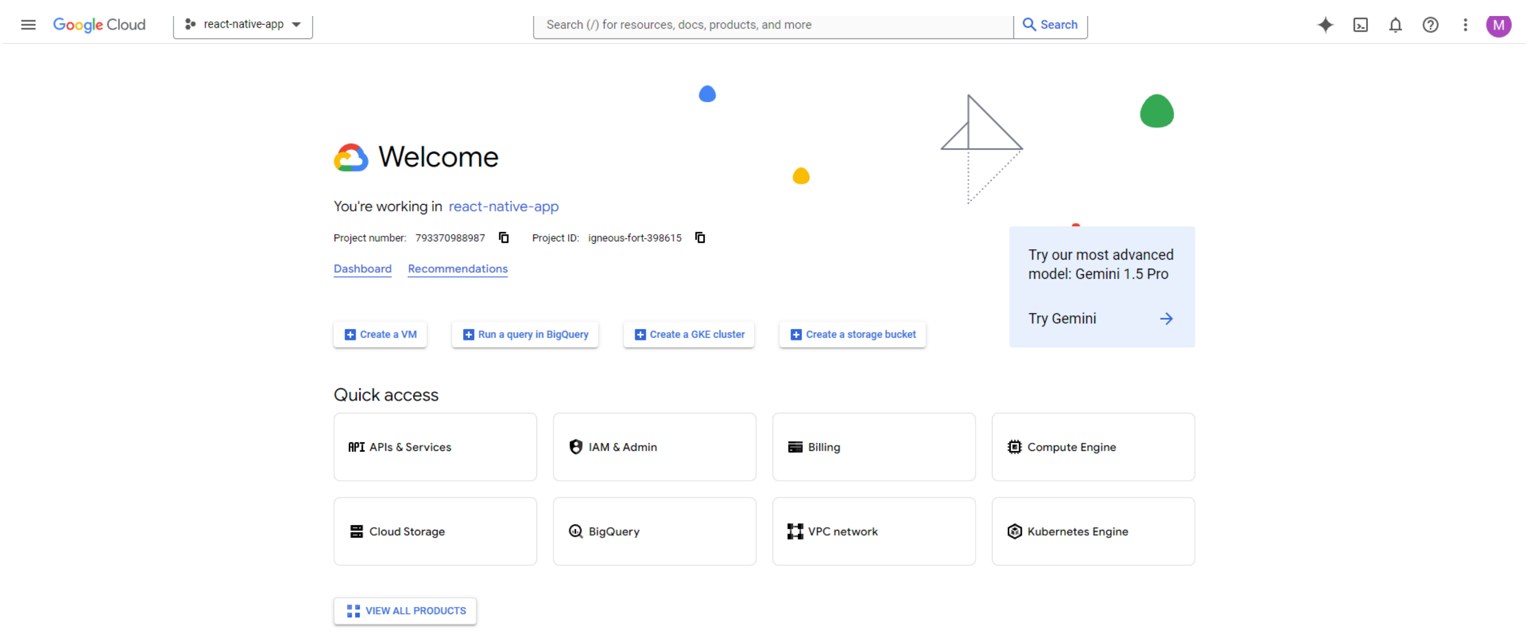


Figure 28 - Socket.io Rooms

5 Google Cloud Platform

5.1 GCP Introduction

Google Cloud Platform (GCP) is a suite of cloud computing services offered by Google that provides scalable and flexible solutions for businesses and independent developers. It includes a wide range of tools for computing, storage, data analytics, machine learning and networking. GCP enables developers to build and run applications on Google's global infrastructure, leveraging services like Google Kubernetes Engine, BigQuery and Cloud AI for enhanced performance. It also supports multi-cloud and hybrid cloud strategies, offering powerful solutions for businesses of all sizes.
After a user accesses Google Cloud Platform (GCP) through their browser, the first step is to create a project, which will serve as the container for all the services they require. Once the project is created, the user can explore the available services and easily navigate through them based on their needs:



You're working in react-native-app

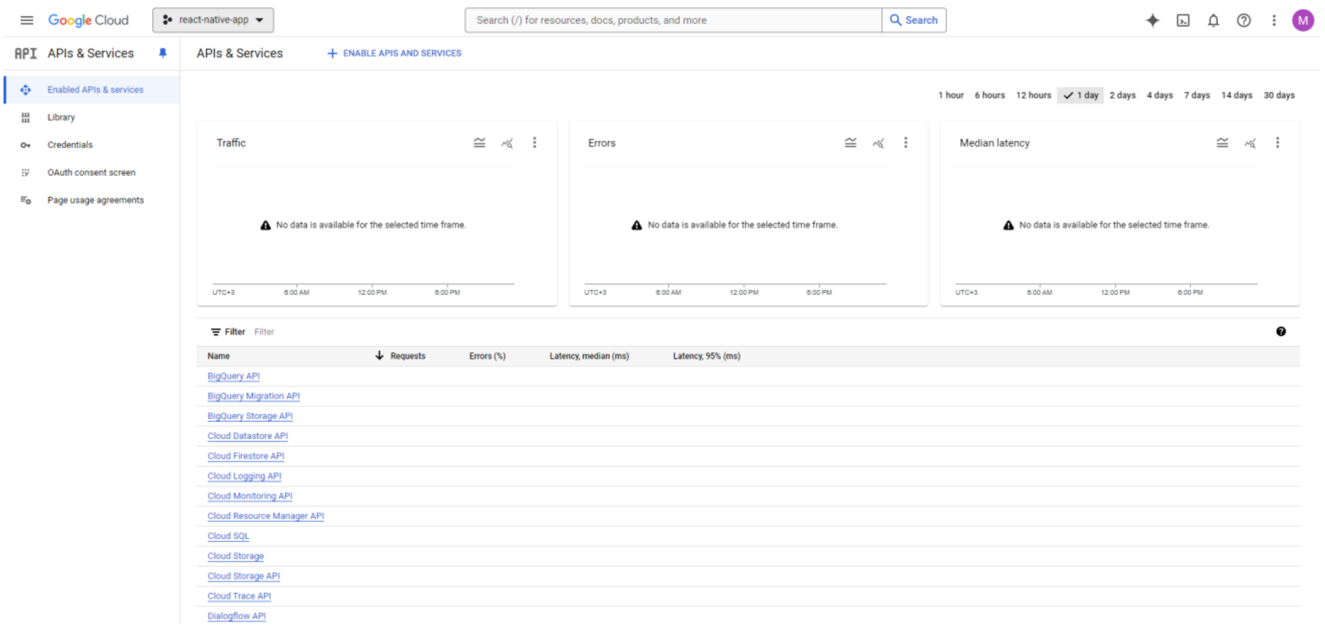
Project number: 793370988987 Project ID: igneous-fort-398615

Figure 29 - GCP

5.2 Google Maps API

Google Maps API provides developers with powerful tools to integrate location-based services into their applications. These APIs enable features like interactive maps, geolocation, route planning, place search and street view. Developers can customize maps with markers, layers and overlays, or retrieve geographic information like addresses and coordinates. Popular APIs include the Maps JavaScript API for embedding maps on websites, the Geocoding API for converting addresses into coordinates and the Directions API for navigation. These tools make it easier to build apps that rely on geographic data, enhancing user experience with real-time mapping capabilities.

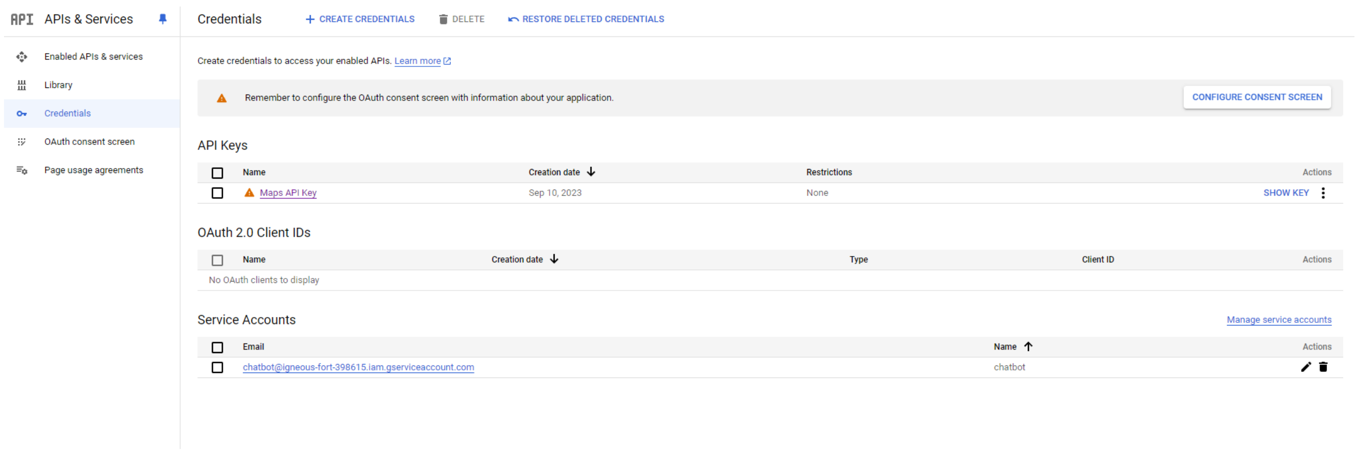
From the Home screen, if the user navigates to “APIs & Services”, they can view detailed information about the APIs offered by Google, including their features and availability:



Name	Requests	Errors (%)	Latency, median (ms)	Latency, 95% (ms)
BigQuery API				
BigQuery Migration API				
BigQuery Storage API				
Cloud Datastore API				
Cloud Firestore API				
Cloud Logging API				
Cloud Monitoring API				
Cloud Resource Manager API				
Cloud SQL				
Cloud Storage				
Cloud Storage API				
Cloud Trace API				
Dialogflow API				

Figure 30 - APIs & Services

Finally, the user needs to navigate to the “Credentials” page to generate a new API key. This key can be used by the developer in the application to grant access to the available resources and services:



Create credentials to access your enabled APIs. Learn more

Remember to configure the OAuth consent screen with information about your application.

API Keys

Name	Creation date	Restrictions	Actions
Maps API Key	Sep 10, 2023	None	SHOW KEY

OAuth 2.0 Client IDs

Name	Creation date	Type	Client ID	Actions
No OAuth clients to display				

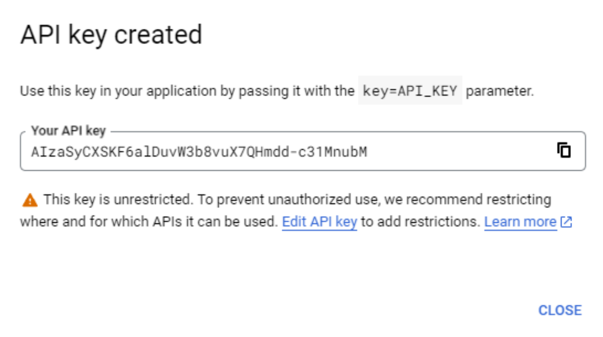
Service Accounts

Manage service accounts

Email	Name	Actions
chatbot@igneous-fort-398615.iam.gserviceaccount.com	chatbot	

Figure 31 - API Credentials

Then, by clicking “Create Credentials”, a new window will appear, displaying details about the newly generated API key:


API key created

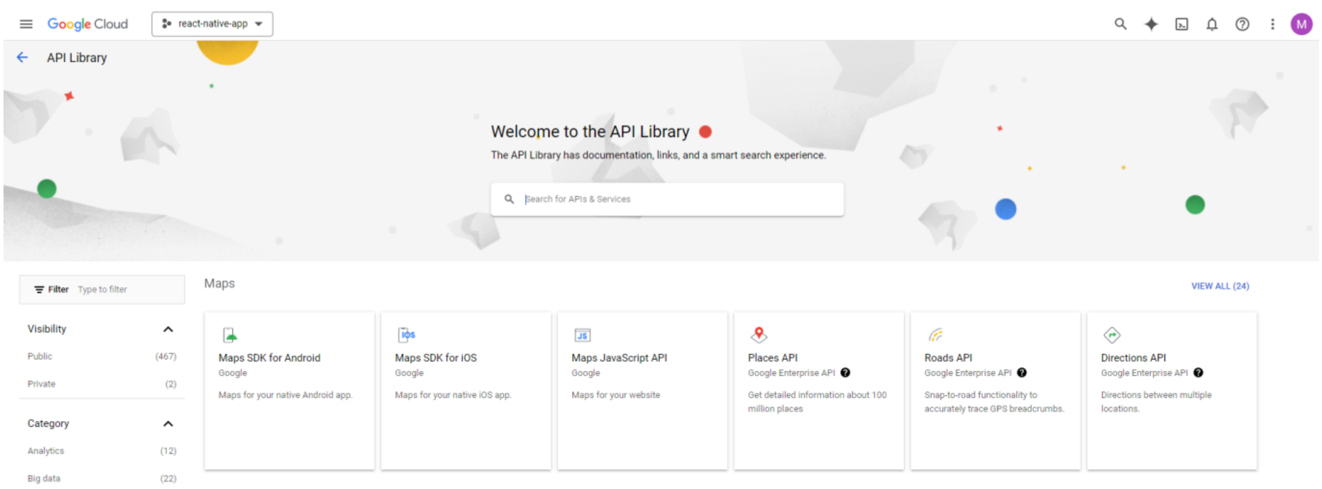
Use this key in your application by passing it with the `key=API_KEY` parameter.

Your API key
AIzaSyCXSKF6alDuvW3b8vuX7QHmdd-c31MnubM

This key is unrestricted. To prevent unauthorized use, we recommend restricting where and for which APIs it can be used. Edit API key to add restrictions. Learn more

Figure 32 - API Key

Additionally, the user can navigate to the "Library" section, where detailed information about the available APIs is provided:



Welcome to the API Library

The API Library has documentation, links, and a smart search experience.

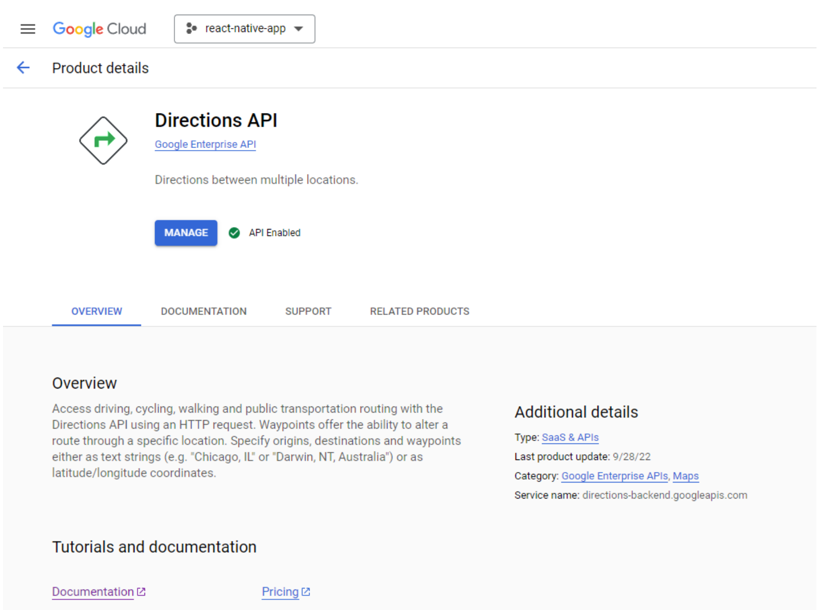
Visibility	
Public	(467)
Private	(2)
Category	
Analytics	(12)
Big data	(22)

Maps — VIEW ALL (24)

API	Provider	Description
Maps SDK for Android	Google	Maps for your native Android app.
Maps SDK for iOS	Google	Maps for your native iOS app.
Maps JavaScript API	Google	Maps for your website
Places API	Google Enterprise API	Get detailed information about 100 million places
Roads API	Google Enterprise API	Snap-to-road functionality to accurately trace GPS breadcrumbs.
Directions API	Google Enterprise API	Directions between multiple locations.

Figure 33 - APIs Library

From there, the user should enable the APIs they want to be supported by the generated API key:



Directions API

Google Enterprise API

Directions between multiple locations.

Overview

Access driving, cycling, walking and public transportation routing with the Directions API using an HTTP request. Waypoints offer the ability to alter a route through a specific location. Specify origins, destinations and waypoints either as text strings (e.g. "Chicago, IL" or "Darwin, NT, Australia") or as latitude/longitude coordinates.

Additional details

Type: SaaS & APIs
Last product update: 9/28/22
Category: Google Enterprise APIs, Maps
Service name: directions-backend.googleapis.com

Tutorials and documentation

Figure 34 - Enable API

5.3 Dialogflow

Dialogflow [18] is a natural language processing (NLP) platform developed by Google. It enables users to create engaging and interactive conversational experiences by understanding and analyzing natural language data.

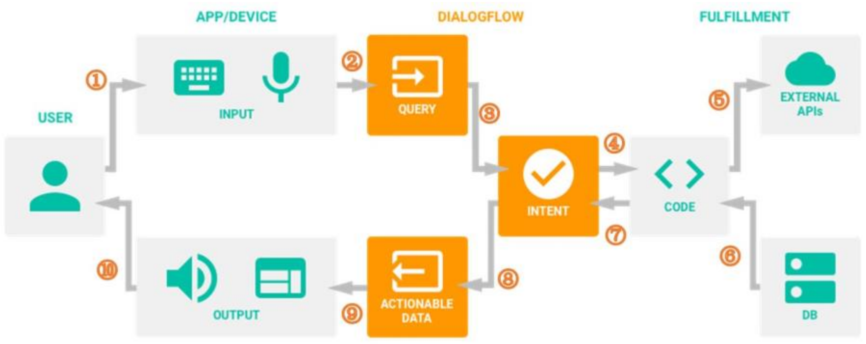


Figure 35 - Dialogflow

Dialogflow is available through Google Cloud Platform (GCP) and the following steps must be followed to use it:

1. Create a service account for the selected project.
2. Generate a private key (API key), which will be used for communication between the server and the platform.
3. Export the key as a JSON file.

5.3.1 Create Service Account

To begin working with Dialogflow, a service account is required. This can be quickly and easily created from Google Cloud Platform (GCP) by selecting the "Service Accounts" option from the navigation menu.

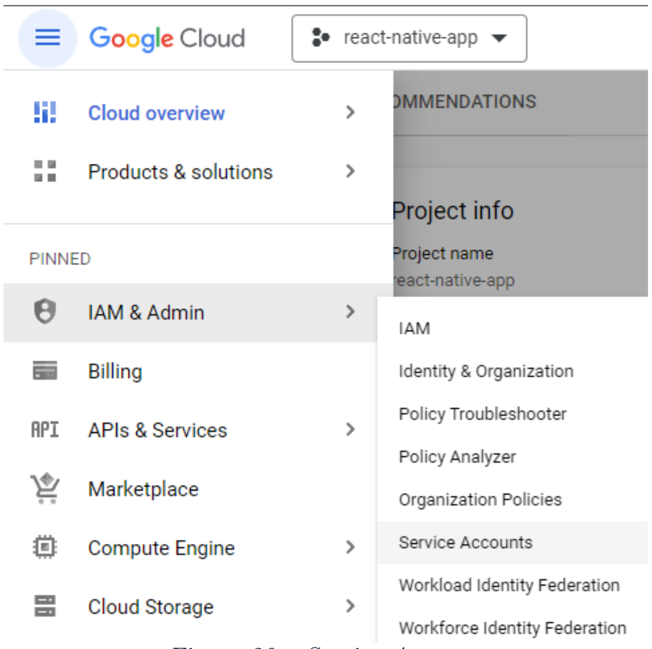
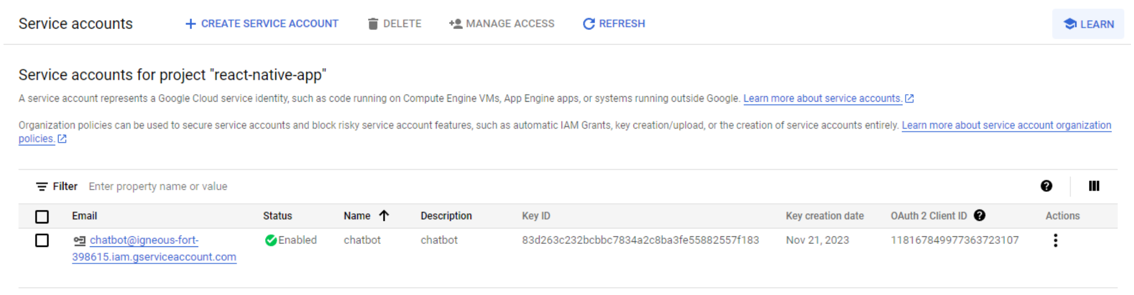


Figure 36 - Service Accounts

From there, the user can manage existing service accounts or add a new one by simply clicking the “CREATE SERVICE ACCOUNT” button:



Service accounts for project "react-native-app"

A service account represents a Google Cloud service identity, such as code running on Compute Engine VMs, App Engine apps, or systems running outside Google. Learn more about service accounts.

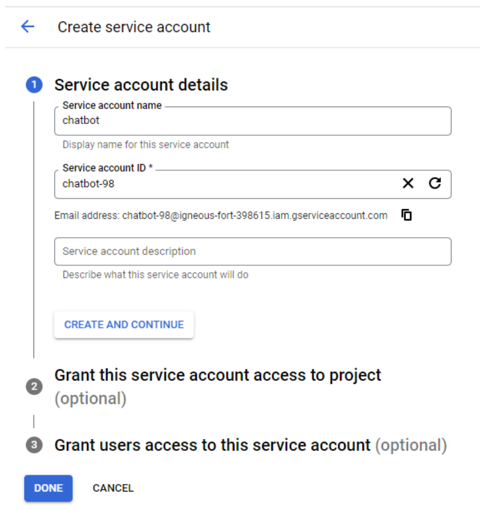
Organization policies can be used to secure service accounts and block risky service account features, such as automatic IAM Grants, key creation/upload, or the creation of service accounts entirely. Learn more about service account organization policies.

Filter Enter property name or value

Email	Status	Name ↑	Description	Key ID	Key creation date	OAuth 2 Client ID	Actions
chatbot@igneous-fort-398615.iam.gserviceaccount.com	Enabled	chatbot	chatbot	83d263c232bcbbc7834a2c8ba3fe55882557f183	Nov 21, 2023	118167849977363723107	⋮

Figure 37 - Service Accounts List

When creatig a new serivce account, a name shall be given as a first step:



Service account name: chatbot
Display name for this service account

Service account ID *: chatbot-98

Email address: chatbot-98@igneous-fort-398615.iam.gserviceaccount.com

Service account description
Describe what this service account will do

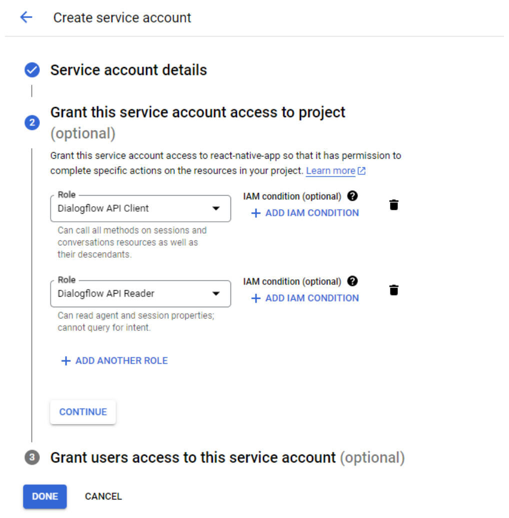
Grant this service account access to project (optional)

Grant users access to this service account (optional)

Figure 38 - Create Service Account 1st step

Next, the appropriate roles should be assigned. To access all the features of Dialogflow, two roles are required:

- Dialogflow API Client
- Dialogflow API Reader



Grant this service account access to project (optional)

Grant this service account access to react-native-app so that it has permission to complete specific actions on the resources in your project. Learn more

Role: Dialogflow API Client
IAM condition (optional)
Can call all methods on sessions and conversations resources as well as their descendants.

Role: Dialogflow API Reader
IAM condition (optional)
Can read agent and session properties; cannot query for intent.

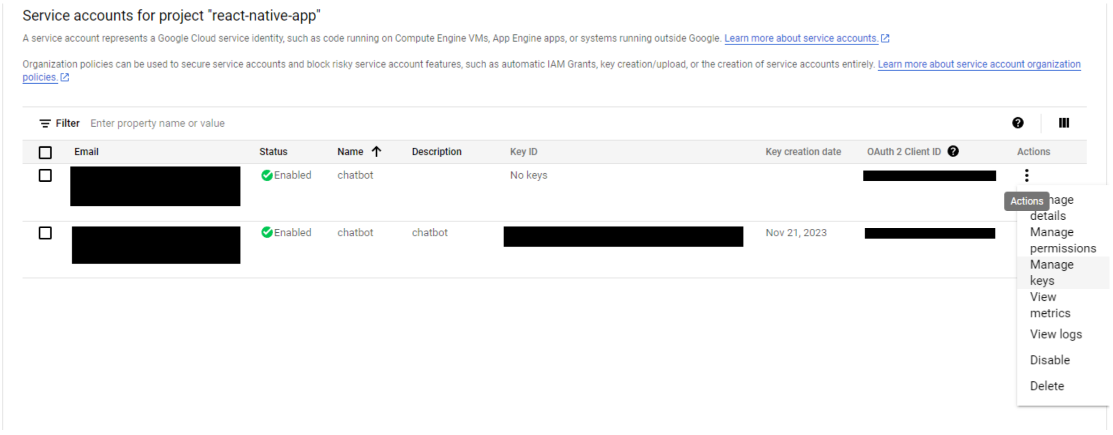
Grant users access to this service account (optional)

Figure 39 - Create Service Account 2nd step

The third and final step is optional and no input is required

5.3.2 Generate Private Key

A private key can be generated from the “Actions” column of the targeted service account, by clicking the “three dots” and selecting the “Manage Keys” option



Service accounts for project "react-native-app"

A service account represents a Google Cloud service identity, such as code running on Compute Engine VMs, App Engine apps, or systems running outside Google. Learn more about service accounts.

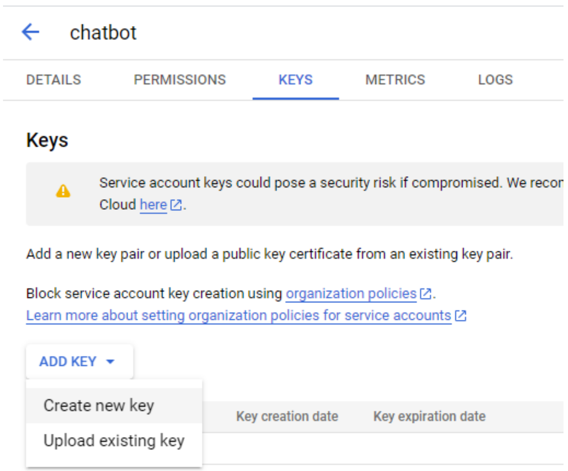
Organization policies can be used to secure service accounts and block risky service account features, such as automatic IAM Grants, key creation/upload, or the creation of service accounts entirely. Learn more about service account organization policies.

Filter Enter property name or value

	Email	Status	Name ↑	Description	Key ID	Key creation date	OAuth 2 Client ID	Actions
☐	[redacted]	✓ Enabled	chatbot		No keys		[redacted]	⋮
☐	[redacted]	✓ Enabled	chatbot	chatbot	[redacted]	Nov 21, 2023	[redacted]	

Figure 40 - Generate Private Key

From there, select the “KEYS” tab and click the “Create new key” option from the “ADD KEY” dropdown menu:



Keys

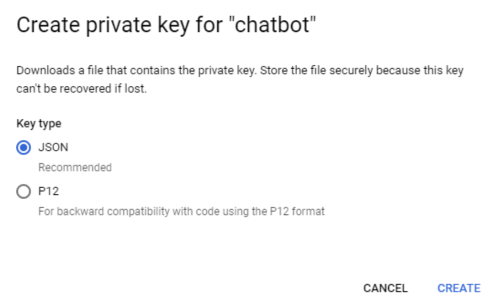
Service account keys could pose a security risk if compromised. We recor
Cloud here.

Add a new key pair or upload a public key certificate from an existing key pair.

Block service account key creation using organization policies.
Learn more about setting organization policies for service accounts

Figure 41 - Create New Key

Then the JSON format should be selected as the key type:



Create private key for "chatbot"

Downloads a file that contains the private key. Store the file securely because this key can't be recovered if lost.

Key type

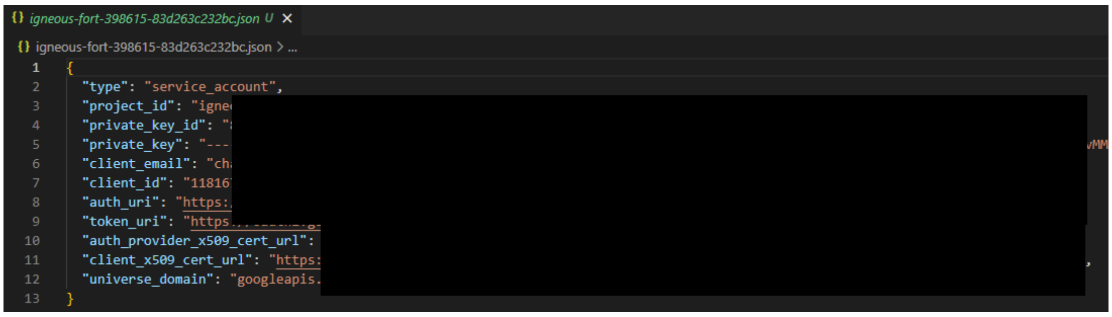
[✓ JSON]
Recommended

P12
For backward compatibility with code using the P12 format

Figure 42 - Key Type

5.3.3 Export Private Key

This key can be exported and used in the application to authenticate the user and grant access to the resources associated with that key:



igneous-fort-398615-83d263c232bc.json U ×

igneous-fort-398615-83d263c232bc.json > ...

```
{
  "type": "service_account",
  "project_id": "igne
  "private_key_id": "
  "private_key": "---
  "client_email": "ch
  "client_id": "11816
  "auth_uri": "https:
  "token_uri": "https
  "auth_provider_x509_cert_url":
  "client_x509_cert_url": "https:
  "universe_domain": "googleapis.
}
```
[redacted]

Figure 43 - Export Key

5.4 Dialogflow Architecture

Through GCP, two versions of Dialogflow are offered:

1. Dialogflow CX: This is an out-of-the-box solution that uses flowcharts and is ideal for simple scenarios where users can choose from a predefined list of options. It is not well-suited for managing free-form text.
2. Dialogflow ES (Essentials): This is a more advanced option, designed for handling free-form text with greater capabilities than CX.

Dialogflow ES is the most commonly used option, offering more features and better performance. Therefore, will focus on its architecture:

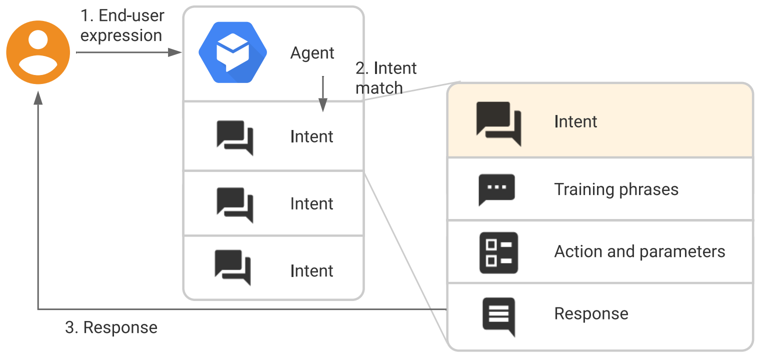


Figure 44 - Dialogflow ES

5.4.1 Agent

Agent is the first entity of this model. It is a natural language understanding unit designed and developed to manage conversations with end-users. It functions as a virtual assistant or chatbot, interacting with users and processing their requests. A custom agent can be created from scratch, use an existing prebuilt agent, or combine elements of both approaches. Typically prebuilt agents are designed to interact with specific entities, such as external systems or third-party APIs, making them suitable for more experienced users of the platform.

5.4.2 Intents

An intent represents what the user wants to achieve or communicate. Each new agent comes with two default intents (Welcome and Fallback) and also provides the option to create additional custom intents.

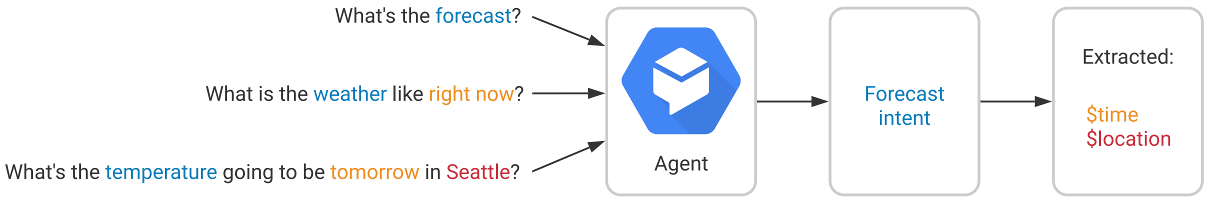

What's the forecast?
What is the weather like right now?
What's the temperature going to be tomorrow in Seattle?
Agent
Forecast intent
Extracted:
$time
$location

Figure 45 - Intents

Each intent offers the following options:

- Context: Maintains the state of a conversation, allowing intents to be connected in a meaningful way. This helps the chatbot better understand the conversation's content, improving intent matching accuracy.
- Events: A method for recognizing user input using predefined events from the platform, eliminating the need for text analysis and matching.
- Training Phrases: Specific phrases or sentences that users might say to trigger an intent. This functionality helps the chatbot identify and understand various ways of expressing the same intent.
- Actions and Parameters: Parameters are used to specify details within an intent. For example, in the phrase "What is the weather in Tripoli?" the word "Tripoli" can be labeled as a parameter named "city", which can take various values without altering the rest of the sentence. Similarly, an action represents the task or function that should be executed when an intent is triggered.
- Responses: The messages or actions that the chatbot should return to the user once the intent is matched. These can include simple text, audio messages, or other actions.
- Fulfillment: Code available in an external system responsible for handling the intent and providing data to the user through a network service.

5.4.3 Entities

An entity represents a concept or object related to user input. Entities are used to extract specific pieces of information from the user's text, aiding in a more precise understanding of their intent. They are categorized into two types:

- System Entities: Predefined entities useful for common and well-known tasks, such as extracting a date or color from the input text. Examples of system entities include @sys.date and @sys.number.

- Custom Entities: Entities created by the user to meet the specific needs of their implementation.

5.4.4 Integrations

Integrations are a fundamental component of chatbot architecture, allowing them to be embedded into existing systems such as Facebook, Messenger and Slack:

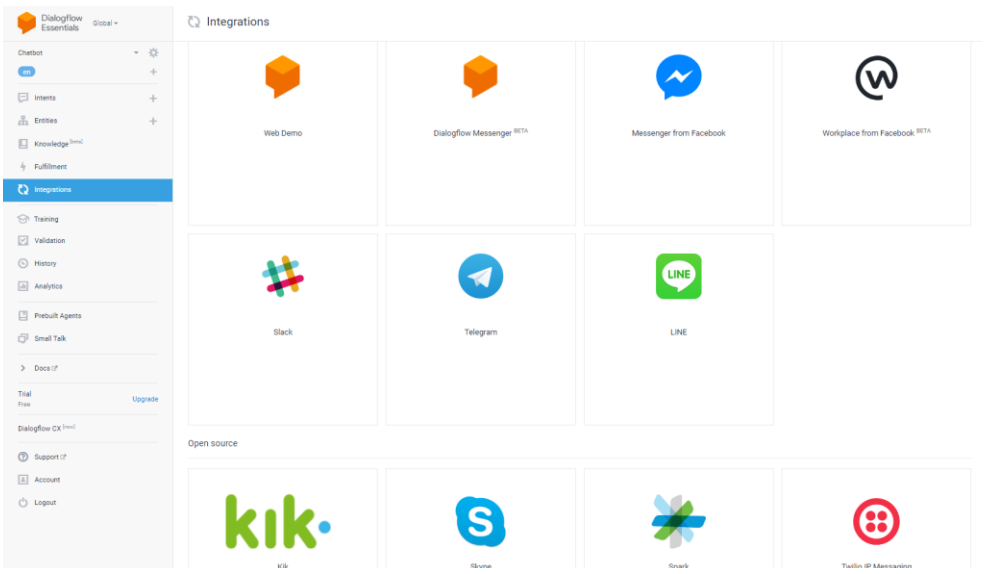
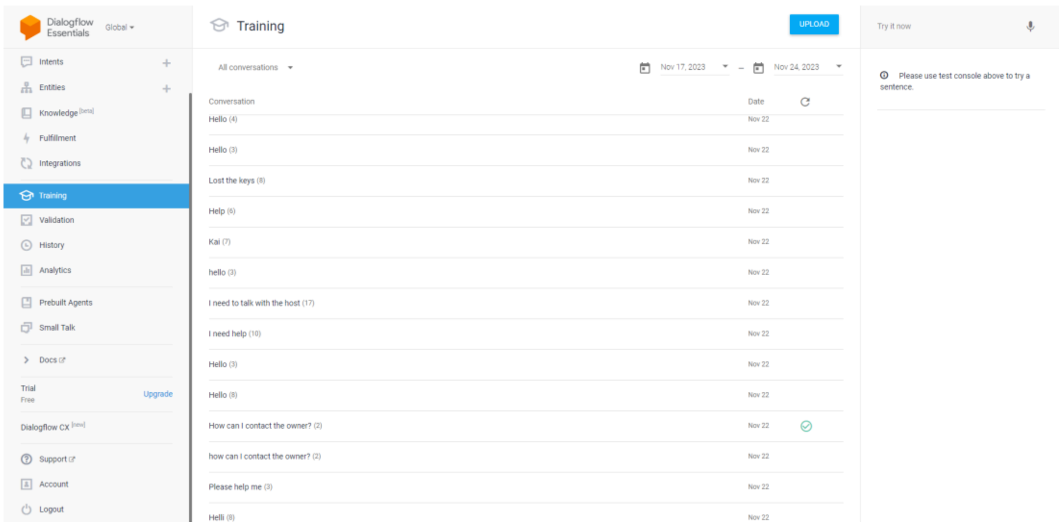


Figure 46 - Integrations

5.4.5 Training

In the Training tab, the agent can be trained either by uploading a file or by analyzing previous conversations. The file-based method involves automatic training, where the chatbot learns from a text file containing expected questions and answers. Although this method can be time-consuming, it often yields effective results. Alternatively, manual training is required when the chatbot fails to respond to certain user inputs. In such cases, intervention from the administrator is needed to either map the text to an existing intent or create a new intent to handle the situation:



Please use test console above to try a sentence.

Conversation	Date	
Hello (4)	Nov 22	
Hello (3)	Nov 22	
Lost the keys (8)	Nov 22	
Help (6)	Nov 22	
Kai (7)	Nov 22	
hello (3)	Nov 22	
I need to talk with the host (17)	Nov 22	
I need help (10)	Nov 22	
Hello (3)	Nov 22	
Hello (8)	Nov 22	
How can I contact the owner? (2)	Nov 22	✓
how can I contact the owner? (2)	Nov 22	
Please help me (3)	Nov 22	
Helli (8)	Nov 22	

Figure 47 - Training

6 React Native

6.1 Introduction

React Native [19] is an open-source JavaScript framework designed for building hybrid (cross-platform) applications. It was developed and is maintained by Facebook, with significant contributions from the programming community through open-source packages that can be easily integrated into the codebase. One of the main advantages of React Native is that it allows developers to write a single codebase for both Android and iOS devices, significantly reducing the time and effort required compared to writing separate code for each operating system.
This cross-platform capability offers a more efficient development process, allowing for faster deployment of apps across multiple platforms without sacrificing performance or user experience. React Native combines the best parts of native development with the flexibility of React, a popular JavaScript library for building user interfaces, making it easier to create dynamic and high-performing mobile applications. Additionally, the framework leverages native components instead of web-based ones, providing a more fluid and responsive app experience. React Native's wide ecosystem offers various pre-built components, plugins and libraries, making it a powerful tool for mobile app development.
Several key features distinguish React Native from other frameworks, enhancing its appeal for mobile app development:

- Native Performance: React Native provides near-native performance by avoiding the use of web views for rendering components. Instead, it leverages native modules and components, ensuring a smooth and optimized user experience.
- Reusable Code: Developers can write code once and apply it to both Android and iOS platforms. Through conditional logic, specific functionalities can be tailored to each operating system, reducing redundancy and speeding up development.
- Hot Reloading: This feature allows developers to see real-time updates in the app as they modify the code, without the need for rebuilding the app from scratch, making the development process faster and more efficient.
- Expo: An open-source framework that simplifies the React Native development process. Expo provides a set of tools, including Expo Go, which allows developers to test their app directly on physical devices without needing to install additional software.
  These features make React Native a highly efficient and versatile tool for building cross-platform mobile applications.

6.2 Local Development

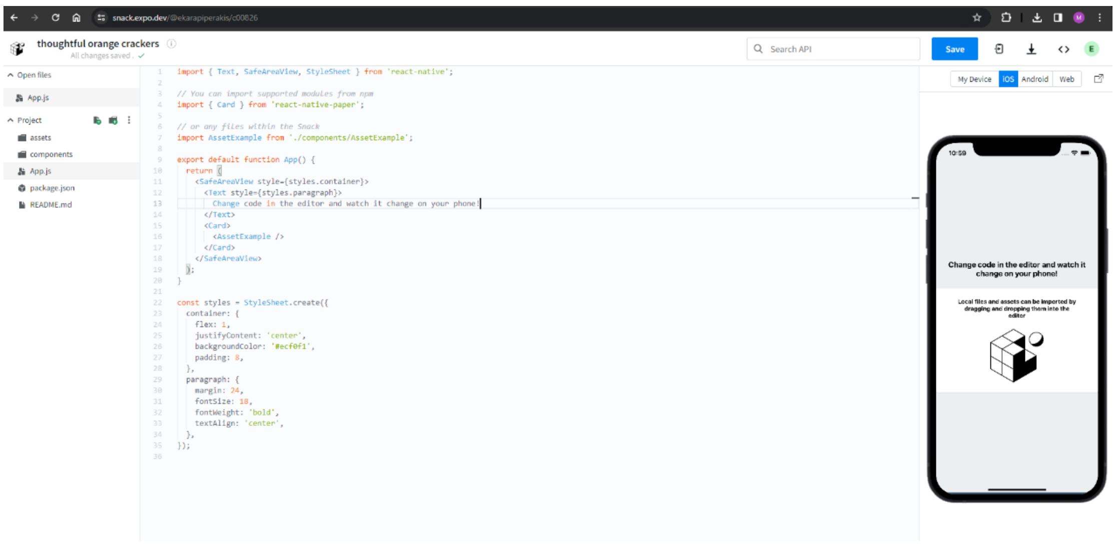
There are two main ways to develop a React Native application:

- Expo Go: This is the easiest way to get started with React Native, requiring only a recent version of Node (16+) and a physical device for testing.

- React Native CLI: This method requires Xcode for iOS devices and Android Studio for Android devices.

If the local machine runs macOS, both Xcode and Android Studio can be used, making the second option (React Native CLI) preferable. However, if the user is on Windows, Xcode is not available, meaning development can only be done for Android. In this case, the first option (Expo Go) is recommended, as it allows real-time testing of the app on a physical device. Since the local machine in this scenario is running Windows, iOS testing wouldn't be possible, making Expo Go the most suitable solution for app testing and development.

An even faster method, recommended for quickly testing specific features, is Snack. This is a web-based platform that allows users to write code and test their React Native app either through an embedded simulator on the website or directly on the user's physical device. It's a convenient option for rapid testing without needing a full development setup.


snack.expo.dev/@ekarapiperakis/c00826
thoughtful orange crackers
All changes saved
Open files
App.js
Project
assets
components
App.js
package.json
README.md
Search API
Save
My Device
iOS
Android
Web
10:59
Change code in the editor and watch it change on your phone!
Local files and assets can be imported by dragging and dropping them into the editor

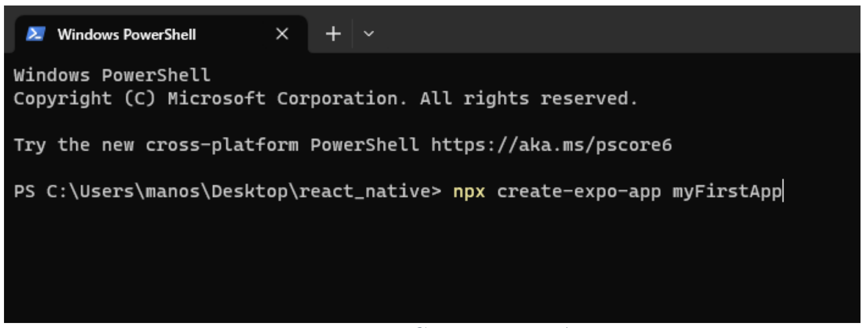
```
import { Text, SafeAreaView, StyleSheet } from 'react-native';

// You can import supported modules from npm
import { Card } from 'react-native-paper';

// or any files within the Snack
import AssetExample from './components/AssetExample';

export default function App() {
  return (
    <SafeAreaView style={styles.container}>
      <Text style={styles.paragraph}>
        Change code in the editor and watch it change on your phone!
      </Text>
      <Card>
        <AssetExample />
      </Card>
    </SafeAreaView>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    backgroundColor: '#ecf0f1',
    padding: 8,
  },
  paragraph: {
    margin: 24,
    fontSize: 18,
    fontWeight: 'bold',
    textAlign: 'center',
  },
});
```

Figure 48 - Snack

6.3 Expo Go

A new Expo project can be created by running the command: "npx create-expo-app <app name>".

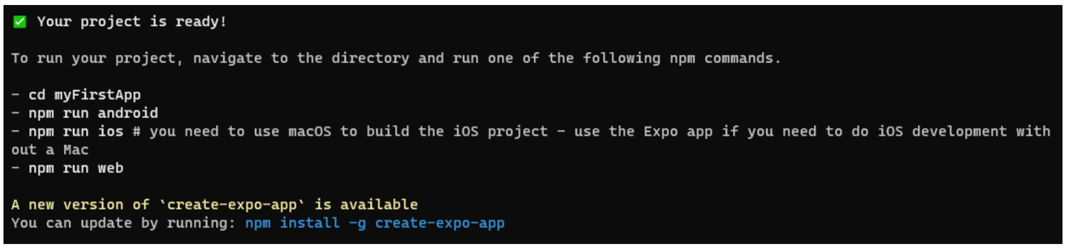
This initializes a new React Native project using Expo with the specified app name, simplifying the setup process:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\manos\Desktop\react_native> npx create-expo-app myFirstApp
```

Figure 49 - Create Expo App

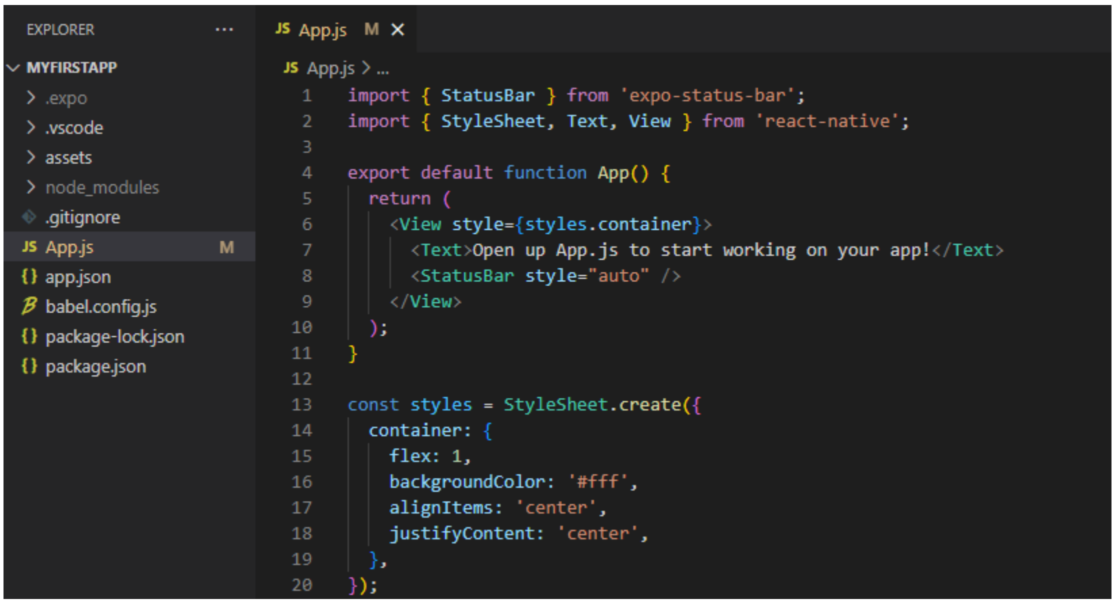
Afterward, additional information about the next steps will be displayed to guide the user through the setup and development process:



```
✅ Your project is ready!

To run your project, navigate to the directory and run one of the following npm commands.

- cd myFirstApp
- npm run android
- npm run ios # you need to use macOS to build the iOS project - use the Expo app if you need to do iOS development with
out a Mac
- npm run web

A new version of `create-expo-app` is available
You can update by running: npm install -g create-expo-app
```

Figure 50 - Expo App Details

By opening the newly created folder in some editor like VSCode, the user will find a default file called App.js. This file serves as the starting point for your React Native app:



```
import { StatusBar } from 'expo-status-bar';
import { StyleSheet, Text, View } from 'react-native';

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Open up App.js to start working on your app!</Text>
      <StatusBar style="auto" />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Figure 51 - Root File

To run the app and test it on a physical device, the user needs to first install the Expo Go app from the Play Store for Android devices or from the App Store for iOS devices, depending on the platform:

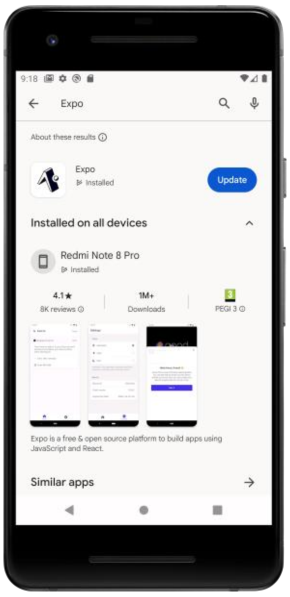
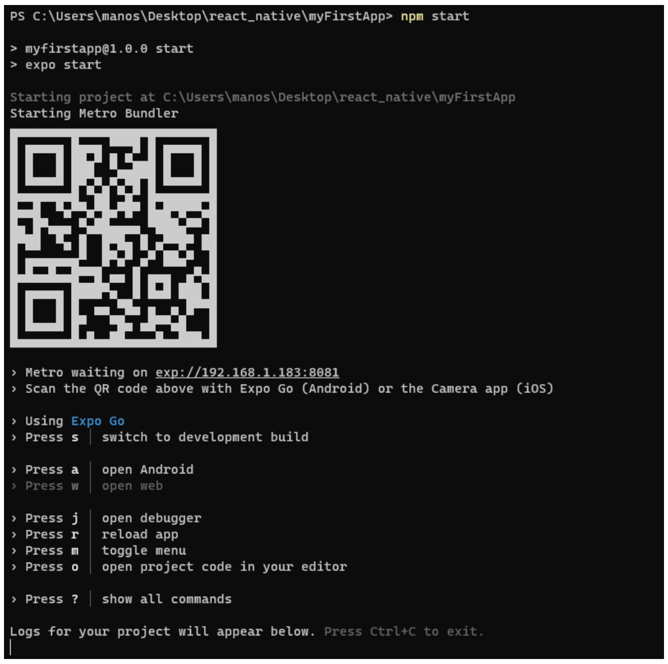


Figure 52 - Install Expo App

After running the command "npm start" in the local machine, the app will launch and several options will be available:

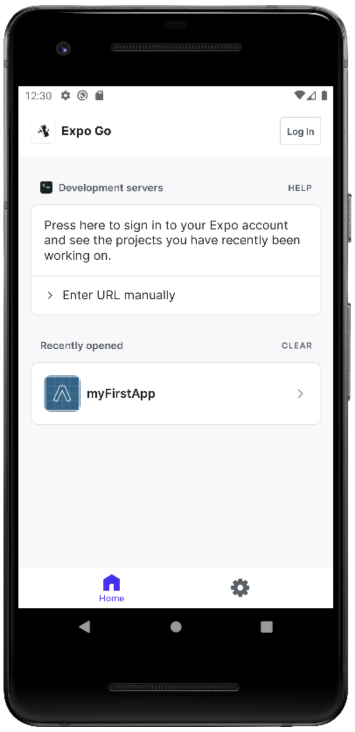
- QR Code: The user can scan this code and the app will open on their device, provided that both the device and the local machine are on the same network.
- Press 'a': If Android Studio is installed, the selected Android emulator will launch.
- Press 'j': A window will open, allowing the user to debug the app through an inspector, like browser developer tools.
- Press 'r': This will refresh the app.
- Press 'm': A menu will appear on the mobile device with options such as debug, refresh, performance monitor and more.
- Press 'o': This opens the app in the default text editor.



```
PS C:\Users\manos\Desktop\react_native\myFirstApp> npm start

> myfirstapp@1.0.0 start
> expo start

Starting project at C:\Users\manos\Desktop\react_native\myFirstApp
Starting Metro Bundler

› Metro waiting on exp://192.168.1.183:8081
› Scan the QR code above with Expo Go (Android) or the Camera app (iOS)

› Using Expo Go
› Press s │ switch to development build

› Press a │ open Android
› Press w │ open web

› Press j │ open debugger
› Press r │ reload app
› Press m │ toggle menu
› Press o │ open project code in your editor

› Press ? │ show all commands

Logs for your project will appear below. Press Ctrl+C to exit.
```

Figure 53 - Start App

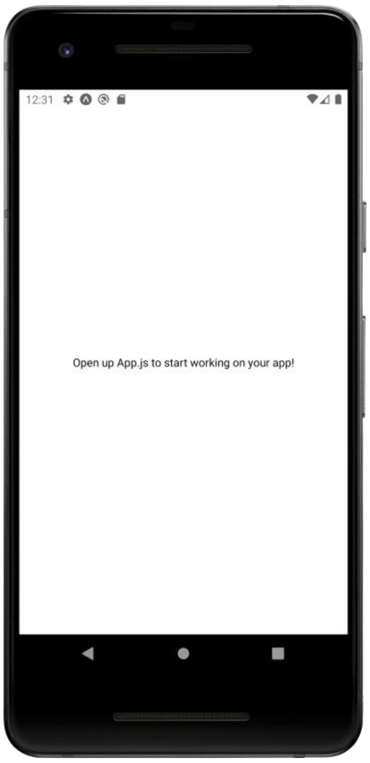
Upon opening the application, the user will see the following result:



12:30
Expo Go
Log In
Development servers
HELP
Press here to sign in to your Expo account and see the projects you have recently been working on.
Enter URL manually
Recently opened
CLEAR
myFirstApp
Home

Figure 54 - Available Apps

By pressing "myFirstApp", the project will build and the output of the code previously reviewed in the App.js file will be displayed:



```
Open up App.js to start working on your app!
```

Figure 55 - Build App

6.4 React Fundamentals

React Native runs on React, a popular open-source library for building user interfaces with JavaScript. React consists of several fundamental concepts [20], outlined below:

- Components
- JSX
- Props
- State

6.4.1 Components

In React, components serve as the building blocks of an application's UI. Complex interfaces can be broken down into smaller, reusable pieces that can be managed independently. In React Native, components are similar to those in React on the web, but they render native UI elements like View, Text, or Image instead of HTML elements like div or span. A number of built-in Core Components are provided by React Native, ready to be used in an app. Some of the most important and commonly used are mentioned below:

- View: The View component is one of the most fundamental building blocks in React Native, serving as a flexible container for layout using flexbox, styling, touch handling and accessibility features. It maps directly to the native view on the platform, such as UIView on iOS or android.view on Android. Designed to nest inside other views, a View can contain zero or more child components. For instance, a View can wrap two colored boxes and a text component in a row, with padding for spacing.
- Text: The Text component in React Native is used for displaying text and supports features like nesting, styling and touch handling. The code snippet below demonstrates an example of how the View and Text components can be combined to showcase the use of core components in React Native. The code snippet begins by importing the View and Text components from the react-native library. It then defines a functional component named *ViewBoxWithText*. Within this component, the return statement renders a View component that acts as a container for the Text component. The Text component displays the string "Hello World!" inside the View. Finally, the component is exported as the default export, making it available for use in other parts of the application:

```
import { View, Text } from "react-native";

const ViewBoxWithText = () => {
  return (
    <View>
      <Text>Hello World!</Text>
    </View>
  );
};

export default ViewBoxWithText;
```

- TextInput: The TextInput component is a fundamental element for entering text into an app via the keyboard. It offers various configurable features through props, including auto-correction, auto-capitalization, placeholder text and different keyboard types like a numeric keypad. The most basic use case involves placing a TextInput on the screen and subscribing to the onChangeText event to capture user input. Additionally, other events such as onSubmitEditing and onFocus can also be utilized to handle specific interactions.
- ScrollView: A component that wraps the platform's native scroll view and integrates with the touch locking "responder" system. It is crucial for ScrollView to have a bounded height to function properly, as it handles unbounded-height children within a constrained container through scrolling.

Additionally to the core components, the user can create their own custom components that are similar to a javascript function and can contain any core component or a custom one, like it was demonstrated in the previous code snapshot, where the *ViewBoxWithText* is a custom component.

6.4.2 JSX

JSX (JavaScript XML) is a syntax extension for JavaScript that resembles HTML. It is used to define the structure of the UI by combining JavaScript logic with declarative syntax, simplifying the development process. In React Native, JSX is employed to outline the layout of a mobile app's interface.

6.4.3 Props

Props (short for properties) are a way of passing data from parent components to child components. They make components dynamic and allow for reusability. As shown in the following example, the welcome function receives the name 'Manos' as a prop and displays it within the paragraph tags.

```
function Welcome(props) {
  return <p>Hello, {props.name}</p>;
}

function App() {
  return (
    <div>
      <Welcome name="Manos" />
    </div>
  );
}
```

6.4.4 State

While props are passed down from parent components, state is managed internally within the component. State holds data that can change over time and directly influences how the component behaves or renders. In React Native, updating the state triggers a re-render of the component, ensuring that the UI reflects the latest state changes. This distinction between props and state allows components to remain both interactive and responsive to user input or other dynamic events.
To manage state within a component, React provides a hook called useState. A hook is a special function that allows developers to "hook into" React's core features. For instance, useState enables adding state to functional components. Since useState is a part of the React library, it needs to be imported separately, as it is not a built-in feature of React Native like the core components imported from the react-native library. The syntax for using useState looks like this:

```
const [<variableName>, <setFunctionName>] = useState(<initialValue>);
```

The set function can be invoked at any point in the code to update the value of a variable. Similarly, the variable can be accessed and used directly in the code wherever needed.

6.5 React Hooks

React Hooks are a powerful feature of the React library, first introduced in version 16.8. Hooks provide a more flexible and streamlined way to manage state and handle various effects within components. Some of the most commonly used hooks include:

- State Hooks: State, as mentioned earlier, allows a component to retain information over time. This capability is crucial in scenarios like handling user input, where the component must dynamically track and respond to changes. By locally storing data within the component, state ensures that interactions, such as form entries or button clicks, are remembered and processed correctly, resulting in a smoother and more engaging user experience. The built-in state hooks include:
    - useState: Declares a state variable that can be dynamically updated.

    - useReduce: Declares a strate variable with the update logic inside a reducer function.
- Context Hooks: Context allows a component to access information from distant parent components, without passing it each time as props. This is especially helpful for managing global data that needs to be shared across multiple layers of the component tree. The built-in context hooks include:
    - useContext: Reads and subscribes to a context.
- Ref Hooks: Refs allow a component to store information that doesn't affect rendering, such as a DOM node or a timeout ID. Unlike state, updating a ref does not trigger a re-render of the components. Refs act as an "escape hatch" from the React paradigm, offering a way to handle certain tasks outside React's typical data flow. They are especially useful when working with non-React systems, like interacting with built-in browser APIs, where you need direct access to elements or other external resources without impacting the component's render behavior. The built-in ref hooks include:
    - useRef: Declares a ref capable of holding any value, though it is most commonly used to store references to DOM nodes.
- Effect Hooks: These hooks allow a component to interact with external systems. This includes handling tasks such as network requests, manipulating the browser DOM, managing animations, integrating widgets from other UI libraries and interacting with other non-React code. The built-in effect hooks include:
    - useEffect: connects a component to an external system.
- Performance Hooks: A common approach to optimizing re-rendering performance is to avoid unnecessary work. For instance, the user can instruct React to reuse a cached calculation or skip a re-render entirely if the data remains unchanged from the previous render. The built-in perfomance hooks include:
    - useMemo: Caches the result of an expensive calculation to prevent unnecessary recalculations.
    - useCallback: Caches a function definition, allowing it to be passed down to optimized components without triggering re-renders.

6.6 State Management

State management in React Native refers to how the app handles and shares data across components, ensuring consistent behavior and UI updates. In smaller applications, local component state, managed with hooks like useState, is often sufficient. However, as the app grows, managing shared or global state becomes more complex. To address this, React Native developers often use external libraries like Redux or React's Context API to centralize state, making it easier to manage data flow, synchronize updates and reduce prop drilling.

6.6.1 Context API

The Context API [21] is a powerful feature in React that enables components to access shared data without the need to pass props through every level of the component tree. This API simplifies the management of global or shared state by providing a way to create and consume context, making it ideal for scenarios like theming, user authentication or language settings:

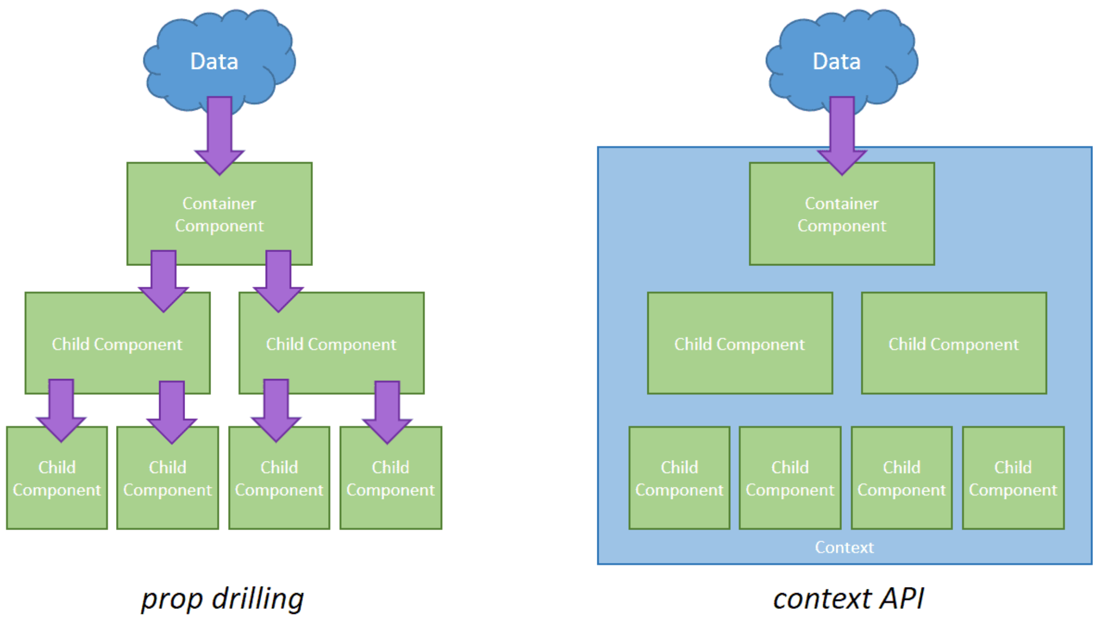


Figure 56 - Context API

Some of the key features of Context API include:

- Centralized State Management: State is available in a central container, accessible by all components in the application. This centralization simplifies the management of shared data.
- Provider Component: A provider component is defined to wrap the parts of the application where the state should be available. This ensures that any components within this provider have access to the shared state.
- Consumer Components: Consumer components are those that can access the state and invoke functions to modify it. These components interact with the state by reading from and updating it as needed.Test
- Reducers: Using reducers with the Context API is common for state management, as they provide a predictable and organized way to modify data. Reducers are functions that specify how the state changes in response to various actions.
- Dispatch Actions: Consumer components dispatch actions that describe changes to the state. These actions are then handled by reducers to update the state accordingly.

6.7 Navigation

Navigation is a fundamental aspect of any mobile app, providing users with the ability to move between different screens and views seamlessly. In React Native, managing navigation involves using libraries and tools designed to handle different navigation patterns and transitions. The most popular library for navigation in React Native is react-navigation.

6.7.1 Install Dependencies

The first step is to install the basic dependencies for react navigation:

- @react-navigation/native: The core library for navigation.
- @react-navigation/stack: For stack-based navigation.
- @react-navigation/bottom-tabs: For tab-based navigation.
- @react-navigation/drawer: For drawer-based navigation.

6.7.2 Stack Navigator

The following code snippet demonstrates the setup of a stack-based navigation system. First NavigationContainer is imported from @react-navigation/native and createStackNavigator is imported from @react-navigation/stack. An instance of the stack navigator is then created using createStackNavigator(). This Stack object will be used to configure the screens and manage navigation behavior, allowing users to push and pop screens in a stack, which is ideal for hierarchical navigation.
The App function defines a functional React component named App, which serves as the main component of the application. Within this component, NavigationContainer wraps the entire navigation setup, providing the necessary context for navigation throughout the app.
Inside NavigationContainer, the Stack.Navigator component is used to define the stack navigator. It manages the navigation between screens. Users can add screens to the stack navigator by specifying a name for each screen and providing a component prop that indicates which React component should be rendered for that screen.

```
import * as React from "react";
import { NavigationContainer } from "@react-navigation/native";
import { createStackNavigator } from "@react-navigation/stack";

const Stack = createStackNavigator();

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}

export default App;
```

6.7.3 Tab Navigator

The Tab Navigator offers a tab-based navigation interface, allowing users to switch between different screens by tapping on tabs. It is particularly useful for applications with multiple distinct sections that users need to navigate quickly. The implementation of a Tab Navigator follows a similar pattern to that of a stack navigator and the process is straightforward. The example below demonstrates how to set up a Tab Navigator:

```
import { createBottomTabNavigator } from '@react-navigation/bottom-
tabs';

const Tab = createBottomTabNavigator();

function TabNavigator() {
  return (
    <Tab.Navigator>
      <Tab.Screen name="Home" component={HomeScreen} />
      <Tab.Screen name="Settings" component={SettingsScreen} />
    </Tab.Navigator>
  );
}
```

After adding some test icons to the bottom tabs, the result of the above example will be:

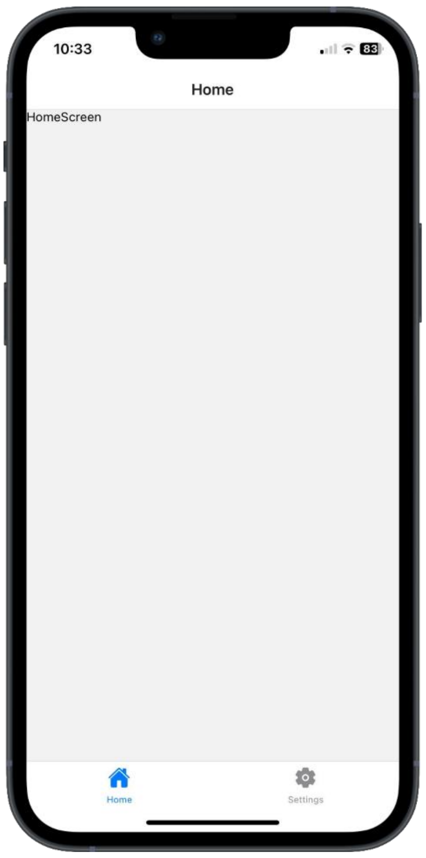


Figure 57 - Stack Navigator

6.7.4 Drawer Navigator

Drawer Navigator is a versatile navigation pattern in React Native that provides a side menu, often referred to as a drawer, allowing users to navigate between different sections or screens of an app. This type of navigation is especially useful for apps with multiple top-level screens. The example below demonstrates how to set up a Drawer Navigator:

```
import { createDrawerNavigator } from "@react-navigation/drawer";

const Drawer = createDrawerNavigator();

function DrawerNavigator() {
  return (
    <Drawer.Navigator>
      <Drawer.Screen name="Home" component={HomeScreen} />
      <Drawer.Screen name="Settings" component={SettingsScreen} />
    </Drawer.Navigator>
  );
}
```

After adding some test icons to the bottom tabs, the result of the following example will be:

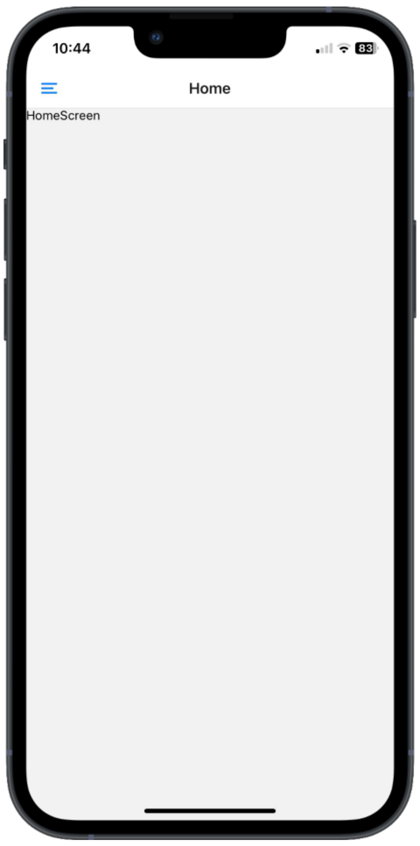


Figure 58 - Drawer Navigator Inactive

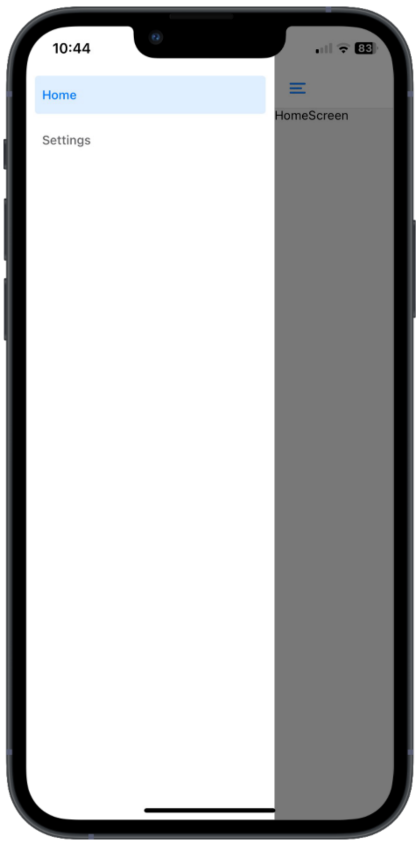


Figure 59 - Drawer Navigator Active

6.8 Styles

Styling in React Native is a key aspect of building user-friendly mobile applications. Unlike web development with CSS, React Native uses a styling system similar to CSS but with some differences tailored for mobile development. React Native provides the StyleSheet API to define styles in a way that optimizes performance and readability. Styles are defined as objects and then these style objects are applied to components using the style prop as it is demonstrated in the example below:

```
import { StyleSheet, Text, View } from 'react-native';

const App = () => {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Hello World!</Text>
      <Text style={styles.subtitle}>Welcome</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#f0f0f0',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    color: '#333',
  },
  subtitle: {
    fontSize: 18,
    color: '#666',
  },
});

export default App;
```

The following result demonstrates the application of basic styles to text elements using the provided code snippet:

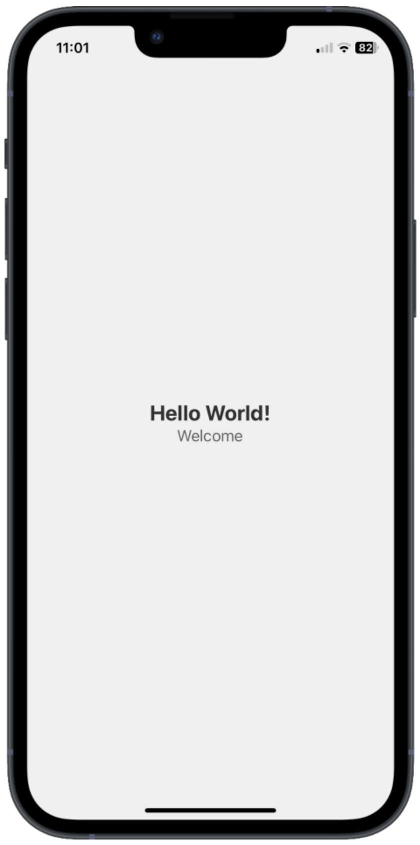


Figure 60 - Styles

6.9 API Integration

API integration involves connecting a React Native app to external services or backend servers to fetch or send data. This allows the app to display dynamic content, interact with databases and perform various operations that depend on external data.
React Native does not include built-in HTTP request functionality, therefore libraries like fetch (native) or third-party libraries like axios are required for making network requests [22].

6.9.1 Fetch

The Fetch API is a built-in JavaScript function for making network requests. It is straightforward and user-friendly for handling basic tasks. The code snippet below demonstrates how to use it to make a simple request to a specific endpoint to retrieve information about a map marker based on an ID. In this example, the useEffect hook initiates the HTTP request using the fetch method when the component mounts. The loading variable displays a loading indicator while the request is in progress. Upon receiving a successful response, the data is displayed on the screen using the data variable. If an error occurs, it is handled and displayed using the information available in the catch block.

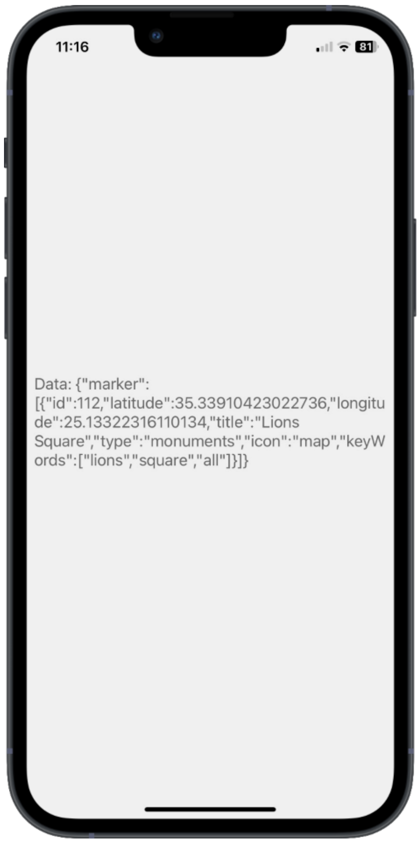
```
import React, { useState, useEffect } from 'react';
import { View, Text, ActivityIndicator, StyleSheet } from 'react-na-
tive';

const App = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch('https://villa-agapi-344fd44fcd28.hero-
kuapp.com/data/api/marker/112')
      .then((response) => response.json())
      .then((json) => {
        setData(json);
        setLoading(false);
      })
      .catch((err) => {
        setError(err);
        setLoading(false);
      });
  }, []);

  if (loading) return <ActivityIndicator />;
  if (error) return <Text>Error: {error.message}</Text>;

  return (
    <View style={styles.container}>
      <Text style={styles.response}>Data:
{JSON.stringify(data)}</Text>
    </View>
  );
};

export default App;
```

6.9.2 Axios

Axios is a popular JavaScript library used for making HTTP requests. It simplifies the process of interacting with APIs by providing a clean API for handling requests and responses. Axios supports promises and async/await syntax, making it easy to work with asynchronous data.
The previous example can be easily implemented using the Axios library this time as shown in the code snippet below:

```
import React, { useState, useEffect } from 'react';
import { View, Text, ActivityIndicator, StyleSheet } from 'react-na-
tive';
import axios from 'axios';

const App = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    axios.get('https://villa-agapi-344fd44fcd28.hero-
kuapp.com/data/api/marker/112')
      .then((response) => {
        setData(response.data);
        setLoading(false);
      })
      .catch((err) => {
        setError(err);
        setLoading(false);
      });
  }, []);

  if (loading) return <ActivityIndicator />;
  if (error) return <Text>Error: {error.message}</Text>;

  return (
    <View style={styles.container}>
      <Text style={styles.response}>Data:
{JSON.stringify(data)}</Text>
    </View>
  );
};

export default App;
```

In both examples, the outcome is identical, as illustrated in the image below.



```
Data: {"marker":
[{"id":112,"latitude":35.33910423022736,"longitu
de":25.13322316110134,"title":"Lions
Square","type":"monuments","icon":"map","keyW
ords":["lions","square","all"]}]}
```

Figure 61 - API Integration

7 Villa Agapi Mobile App

7.1 Data Model

The application's data model is structured around three distinct types of users, each with specific roles and access levels:

1) Visitor: A potential user who can interact with a limited set of features within the application. Visitors typically use the app to learn more about the property. Their access is restricted and they can:

- View information about the property, including basic details about the house and surrounding area.
- Explore local area and find recommended places via dynamic maps.
- Submit a booking request to express their interest in staying at the property.
- Manage their preferences (language, theme mode).
- Login to the app.

2) Guest: A user who has successfully made a booking and has arrived at the property. Upon arrival, the host (admin) provides the Guest with login credentials, that unlock advanced features within the app. These features are designed to improve the Guest's stay by providing more detailed and interactive options, including:

- Access to detailed property information through 2D or 3D map of the house.
- A report/question form for submitting feedback, asking questions or reporting issues.
- A virtual assistant to answer frequently asked questions about the property and the local area.
- Contact information for communicating directly with the host.
- 24/7 live chat support with the host for real-time assistance during their stay.

A guest is considered an active user only during their stay. Starting one day after the departure date, the user is automatically set to inactive and can no longer log in.

3) Admin: The Admin role is assigned to the host, who has complete control over the property and its users. Admins log in using the same interface as Guests but are directed to an exclusive admin dashboard, where they can manage various aspects of the property and its users. Key features available to the Admin include:

- User management, allowing the host to search, create or edit users.
- Booking management, where they can view and handle incoming booking requests.
- Editing property availability to control when the property is bookable.
- Dynamic map management, where the Admin can add or delete markers on interactive maps.
- Statistics dashboard, providing insights into the property's performance, including bookings, user devices and guests' country of origin.

Additionally, Admins can use the live chat feature to communicate with Guests in real-time, offering support and ensuring a better user experience throughout the stay.
This data model ensures a clear distinction between the different user roles, with each group having access to the tools and information they need. In conclusion, Visitors are prospective clients, Guests are active users during their stay and Admins oversee and manage the entire system.

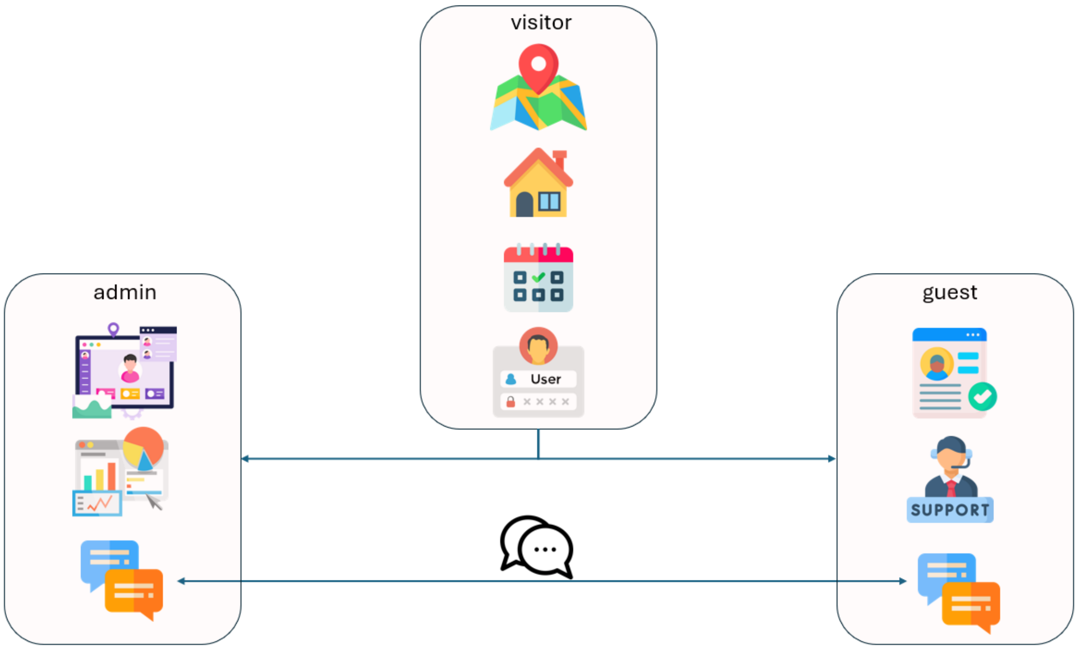


Figure 62 - Data Model

7.2 Tech Stack

The mobile application should be available on both Android and iOS platforms. For this purpose there are several implementation approaches, which can be broadly classified into two main categories:

- Native apps: Designed exclusively for a single platform (iOS or Android), these apps are built using platform-specific languages (Swift for iOS and Kotlin/Java for Android). This approach ensures optimal performance and seamless access to native device features. While such apps provide better performance and an improved user interface, they come with higher development costs. This is due to the additional time, effort and expertise required to develop and maintain two separate versions of the same app, each using different technologies, making the process more complex.
- Hybrid apps: Built using web technologies like HTML, CSS and JavaScript, hybrid apps are wrapped in a native container, enabling them to run across multiple platforms. While they may exhibit slightly reduced performance compared to native apps, the key advantage is the use of a single codebase for both iOS and Android, simplifying development and maintenance.

7.2.1 React Native

For this specific application, the hybrid approach was chosen due to its alignment with the project's priorities. While native performance is not a critical factor that would impact the user experience, the ability to accelerate development and simplify maintenance is important.

Several frameworks can be used to build hybrid apps, including Flutter, React Native and Ionic. In this case, React Native was chosen due to its popularity and the fact that it is more accessible for developers with a web development background, making it a smoother transition and easier to implement.

7.2.2 Node.js

For the backend of the application, Node.js along with the Express library was selected, due to their efficiency and flexibility. Node.js allows for fast, scalable server-side applications by using non-blocking, event-driven architecture, which is ideal for handling multiple requests simultaneously. Express, built on top of Node.js, simplifies the development process with its lightweight framework, offering powerful tools for routing, middleware and handling HTTP requests. This combination enabled easier development, easy maintenance and seamless integration with the front-end, especially given the full-stack JavaScript environment.

7.2.3 PostgreSQL

PostgreSQL was chosen for the database due to its status as an open-source relational database known for advanced features like ACID compliance, support for JSON data and extensibility, making it ideal for handling both structured and unstructured data. It also offers strong concurrency control, ensuring reliable performance even under heavy loads, which makes it a perfect fit for the application where data consistency, flexibility and scalability are critical. Additionally, PostgreSQL integrates seamlessly with a Node.js server, leveraging Node.js's capability to handle multiple connections efficiently, providing a solid foundation for high-performance, scalable applications.

7.2.4 Socket.io

Socket.IO was chosen for its ability to facilitate real-time, bidirectional communication between clients and servers. This makes it especially well-suited for the live chat feature of the application, ensuring instant messaging and seamless interactions between users. With its ability to maintain persistent connections and fall back on other transport methods, Socket.IO ensures reliable performance even in varying network conditions.

7.2.5 Heroku

Heroku was chosen to host the Node.js server and PostgreSQL database because of its easy-to-use platform and smooth deployment process. It scales well, allowing the application to handle different levels of traffic without downtime. Heroku also has built-in support for PostgreSQL, making database management simpler. Its wide range of add-ons and tools helps speed up development and deployment, which is important for staying flexible. Overall, Heroku offers the reliability and adaptability needed for the application's growth and performance. Two applications were created on Heroku: one for the Node.js server, which hosts the REST APIs for the mobile application and uses the PostgreSQL database and another server dedicated to the Socket.IO connection. This approach was chosen to better manage the application's features, as the two servers operate independently. This means that changes made to one application won't affect the other, allowing for more flexibility and easier maintenance.

7.2.6 GitHub

GitHub was utilized as the code repository for the two applications hosted on Heroku, the Node.js server and the Socket.IO app. Both repositories are private and linked to Heroku, enabling continuous integration and development. This setup allows for automatic building of the apps with each push to a selected branch, ensuring that new changes are applied seamlessly. More advanced options include creating separate branches for development, user acceptance testing (UAT) and production, which can be linked to pipelines in Heroku. This setup enhances workflow management and allows for streamlined deployment processes across different stages of development.

7.2.7 Git

Git is a distributed version control system that enables developers to track changes in their code and manage projects efficiently. By creating snapshots of a project's files, Git gives the opportunity to review, revert, or merge changes, making it easy to experiment without losing previous work. In this project, Git was used to manage both the mobile application and server codebases. For the mobile application, the main branch served as the default branch, with new feature branches created for each update. These feature branches were then merged into main upon completion. For the server applications, a three-branch strategy was implemented:

1. Main: Used for local development. A separate branch was created for each new feature, which was then merged into main.

2. Uat: This branch acted as a staging environment. After changes were merged into main, they were pushed to the UAT branch for further testing before deployment.
3. Prod: The production branch hosted the final version of the application. Once changes passed testing in the UAT branch, they were pushed to prod for live use.

7.2.8 Google Cloud Platform

The application was further enhanced by integrating Google Cloud Platform (GCP) to leverage several of its advanced services. A dedicated API key was generated to securely access Google Maps Services, which power the dynamic, interactive maps within the app.
In addition to the mapping functionality, Google Dialogflow was selected to implement an intelligent virtual assistant. This conversational AI is designed to respond to frequently asked questions, making the app more user-friendly and interactive. The virtual assistant uses natural language processing (NLP) capabilities to understand user queries and deliver accurate, real-time responses, improving user engagement and reducing the need for manual customer support.

7.2.9 Expo Go

Expo played an important role in the development and deployment of the mobile application, making both processes more efficient. During the development phase, it provided tools that enabled real-time testing directly on physical devices, which was extremely helpful in catching and fixing issues quickly. It was also very helpfull in the build phase, where it generated platform-specific executable files like APKs (for Android), AABs (Android App Bundles) and IPAs (for iOS). These files were necessary to ensure the app would run smoothly on different devices. After the builds were completed, Expo streamlined the deployment process, making it easier to publish the app to the App Store for iOS users and the Google Play Store for Android users.

7.2.10 Playwright

Playwright is an open-source framework for end-to-end testing, enabling automated tests across browsers like Chrome, Firefox and Safari. Created by Microsoft, it offers features like automated interactions, visual comparisons and network request interception, making it versatile for complex web app testing. In this project, Playwright was used on the backend to test the server's REST APIs. These tests can be run easily from the command line as needed, though they are primarily executed within Heroku Pipelines before merging changes to a higher environment. If any test fails, the process stops, preventing unapproved changes from being merged.

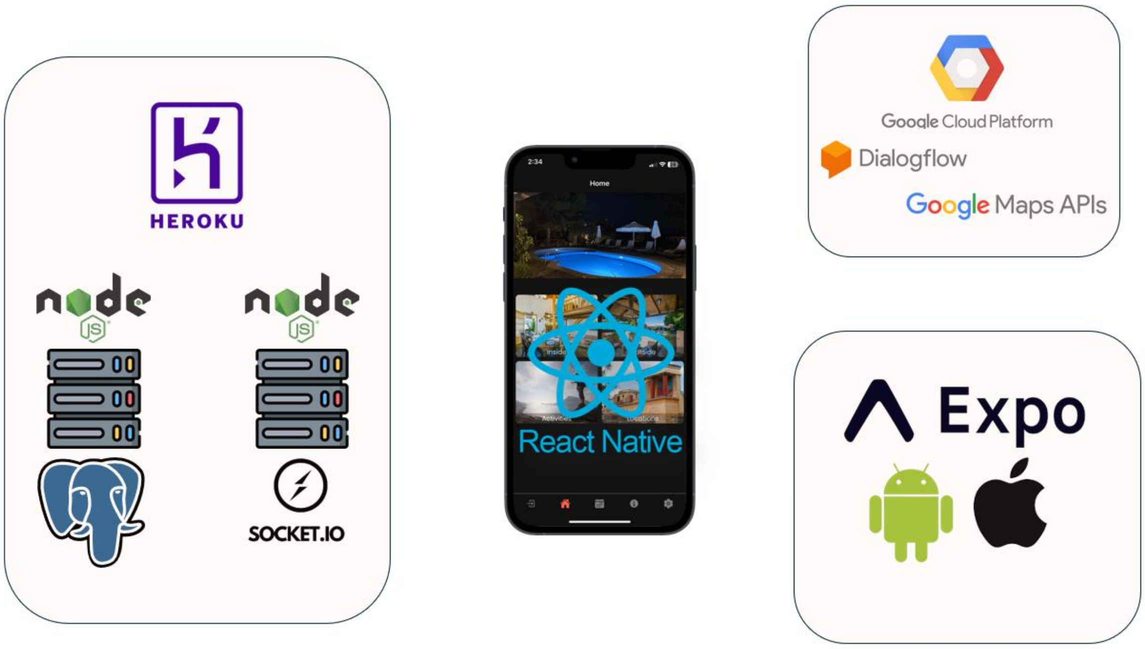
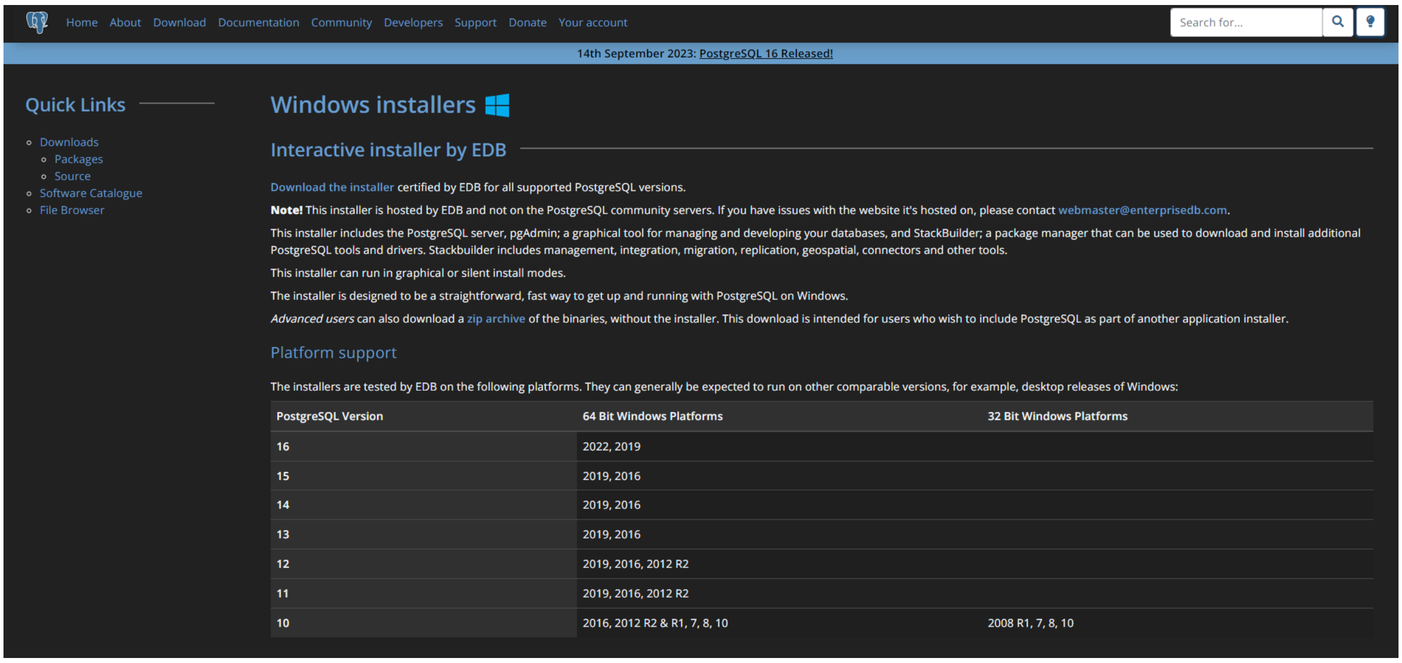


Figure 63 - Tech Stack

8 Application back-end

8.1 Database

For this project, PostgreSQL was chosen as the database system. Version 15 was used, which was installed locally by following the official website's instructions and downloading the installer


Home About Download Documentation Community Developers Support Donate Your account

14th September 2023: PostgreSQL 16 Released!

Quick Links

- Downloads
  - Packages
  - Source
- Software Catalogue
- File Browser

Windows installers

Interactive installer by EDB

Download the installer certified by EDB for all supported PostgreSQL versions.

Note! This installer is hosted by EDB and not on the PostgreSQL community servers. If you have issues with the website it's hosted on, please contact webmaster@enterprisedb.com.

This installer includes the PostgreSQL server, pgAdmin; a graphical tool for managing and developing your databases, and StackBuilder; a package manager that can be used to download and install additional PostgreSQL tools and drivers. Stackbuilder includes management, integration, migration, replication, geospatial, connectors and other tools.

This installer can run in graphical or silent install modes.

The installer is designed to be a straightforward, fast way to get up and running with PostgreSQL on Windows.

Advanced users can also download a zip archive of the binaries, without the installer. This download is intended for users who wish to include PostgreSQL as part of another application installer.

Platform support

The installers are tested by EDB on the following platforms. They can generally be expected to run on other comparable versions, for example, desktop releases of Windows:

PostgreSQL Version	64 Bit Windows Platforms	32 Bit Windows Platforms
16	2022, 2019	
15	2019, 2016	
14	2019, 2016	
13	2019, 2016	
12	2019, 2016, 2012 R2	
11	2019, 2016, 2012 R2	
10	2016, 2012 R2 & R1, 7, 8, 10	2008 R1, 7, 8, 10

Figure 64 - Install PostgreSQL

By opening a terminal and running the command "postgres --version", the installed version can be verified:

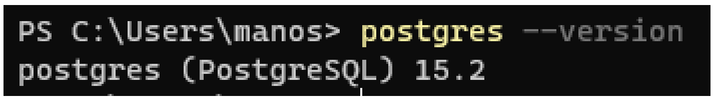

```
PS C:\Users\manos> postgres --version
postgres (PostgreSQL) 15.2
```

Figure 65 - Postgres Version

A prerequisite for accessing the database is starting the server. This can be done by opening a terminal and running the command: pg_ctl start -D "C:\Program Files\PostgreSQL\15\data".

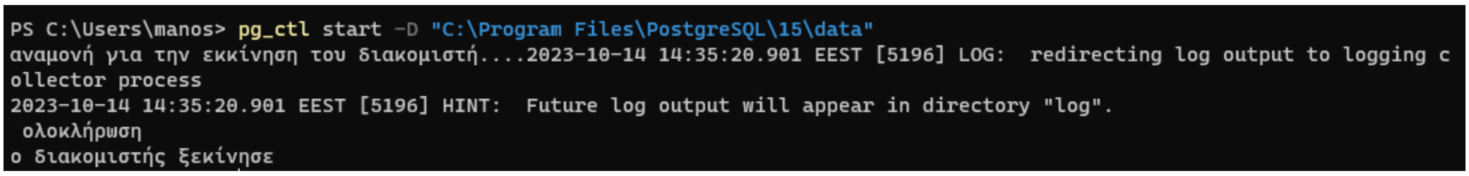
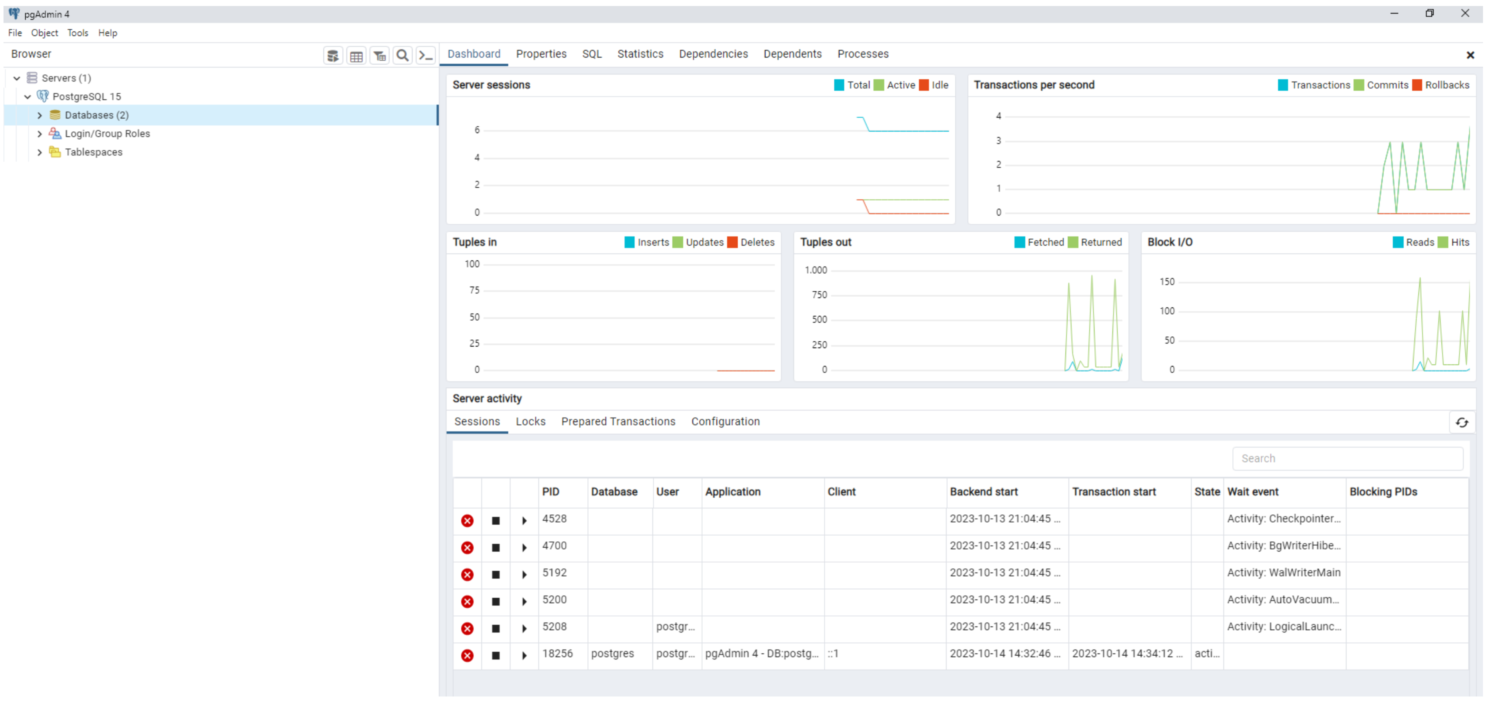

```
PS C:\Users\manos> pg_ctl start -D "C:\Program Files\PostgreSQL\15\data"
αναμονή για την εκκίνηση του διακομιστή....2023-10-14 14:35:20.901 EEST [5196] LOG:  redirecting log output to logging c
ollector process
2023-10-14 14:35:20.901 EEST [5196] HINT:  Future log output will appear in directory "log".
 ολοκλήρωση
ο διακομιστής ξεκίνησε
```

Figure 66 - Start DB

8.1.1 Pgadmin4

PgAdmin4 is the graphical interface that simplifies the creation, maintenance and the overall management of database objects [23]. It is installed alongside

PostgreSQL and access to it is only possible after the server has been started:



			PID	Database	User	Application	Client	Backend start	Transaction start	State	Wait event	Blocking PIDs
⊗	■	▸	4528					2023-10-13 21:04:45 ...			Activity: Checkpointer...	
⊗	■	▸	4700					2023-10-13 21:04:45 ...			Activity: BgWriterHibe...	
⊗	■	▸	5192					2023-10-13 21:04:45 ...			Activity: WalWriterMain	
⊗	■	▸	5200					2023-10-13 21:04:45 ...			Activity: AutoVacuum...	
⊗	■	▸	5208		postgr...			2023-10-13 21:04:45 ...			Activity: LogicalLaunc...	
⊗	■	▸	18256	postgres	postgr...	pgAdmin 4 - DB:postg...	::1	2023-10-14 14:32:46 ...	2023-10-14 14:34:12 ...	acti...		

Figure 67 - PgAdmin4

On the left side, the available servers and databases for each installed version of PostgreSQL are displayed. By default, only the Postgres database is available:

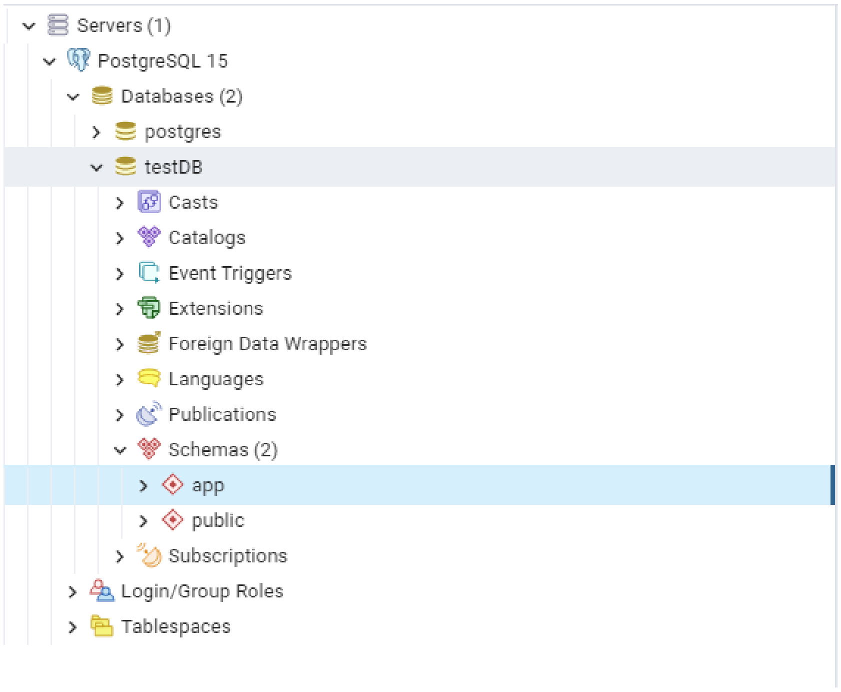


Figure 68 - Available Servers

8.1.2 Create Database

A new database can be created as shown: Databases → Create → Database.

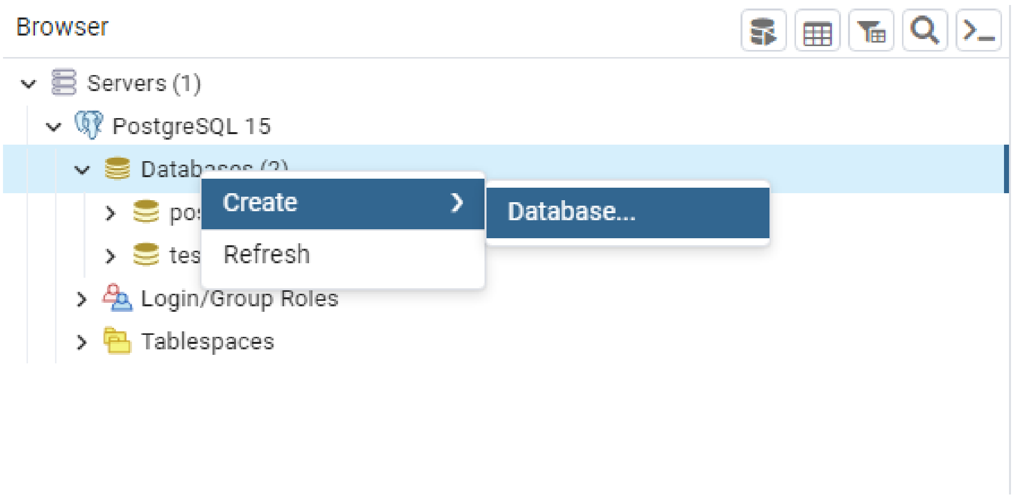
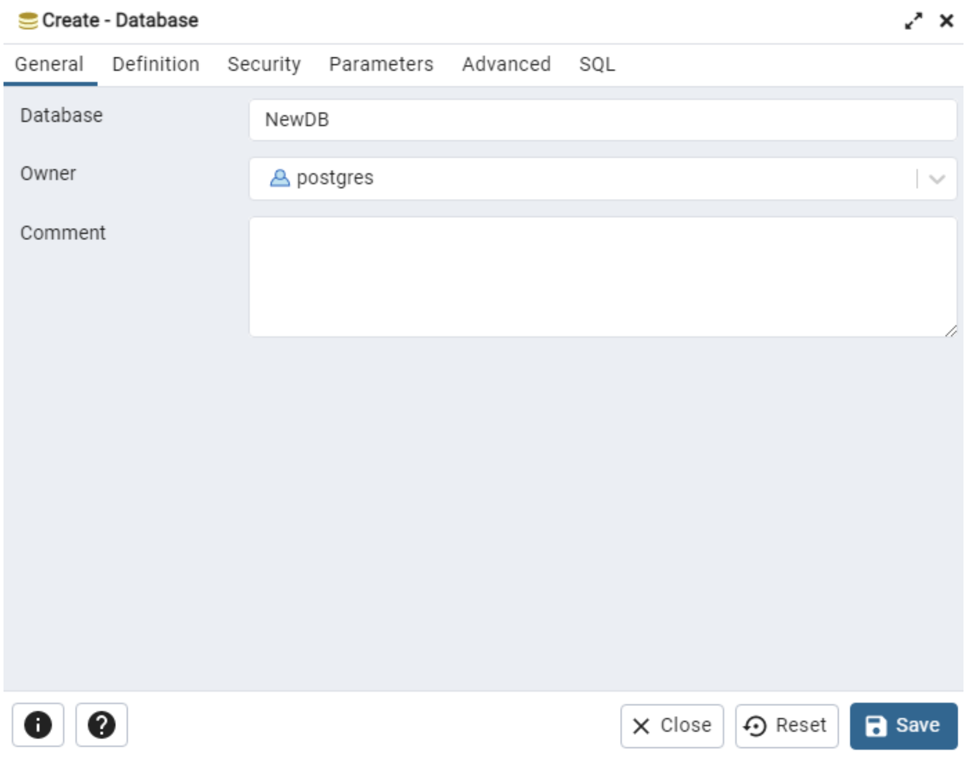


Figure 69 - Create Database 1st step

A new window will appear, prompting the user to choose the database name and select the user (with postgres being the default):

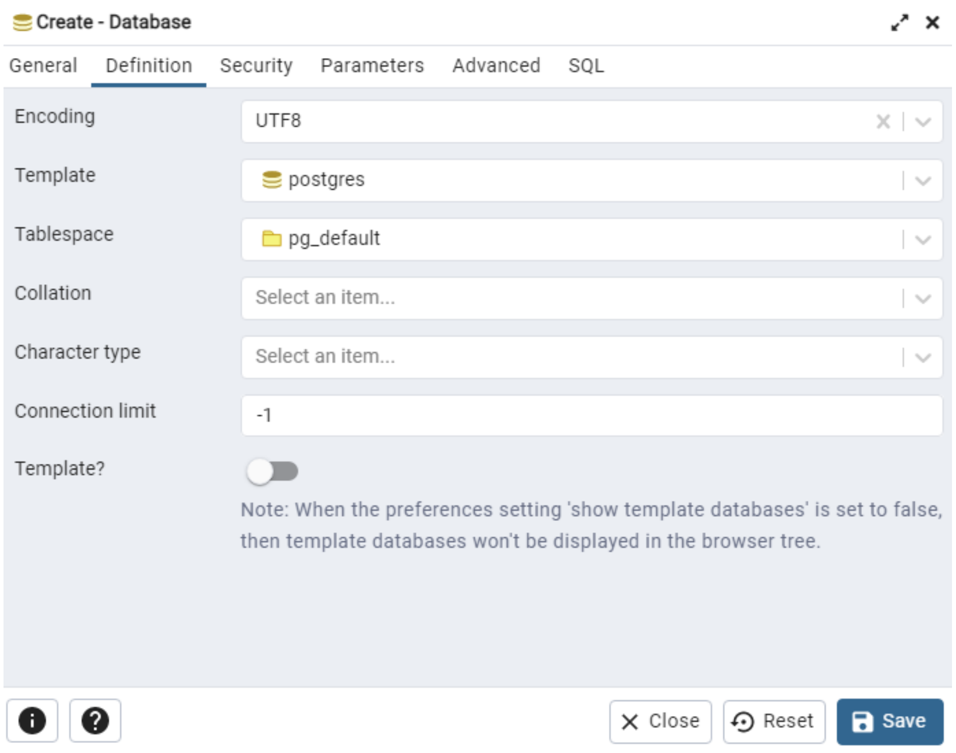


Database: NewDB
Owner: postgres
Comment:

Figure 70 - Create Database 2nd step

By switching tabs, additional options become available, such as:

- Encoding
- Template
- Tablespace



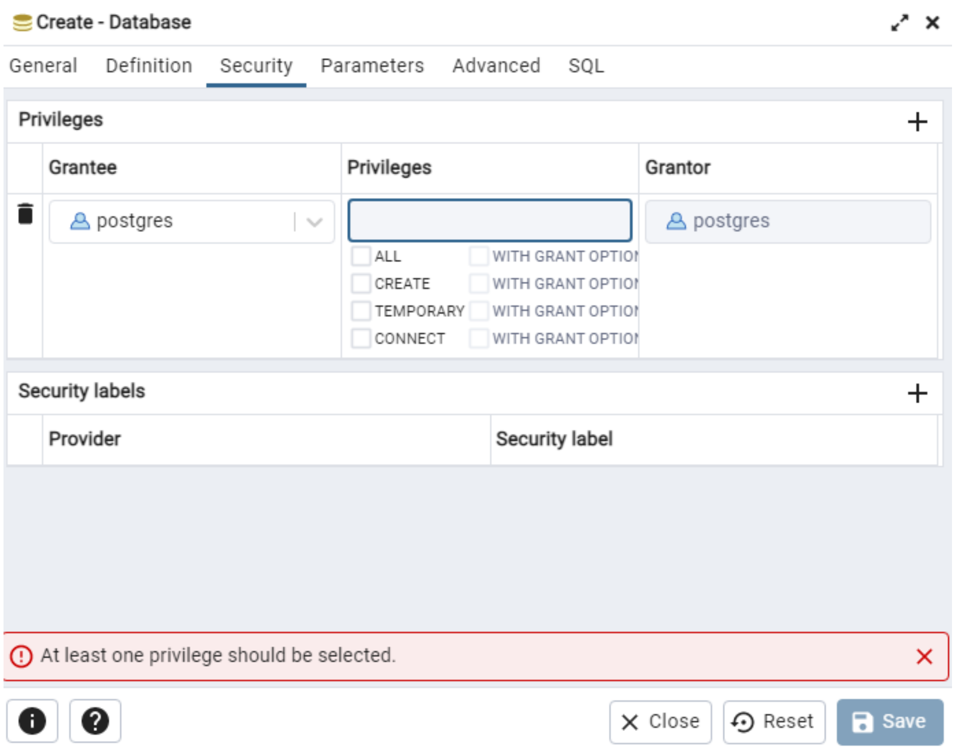
Create - Database

Encoding	UTF8
Template	postgres
Tablespace	pg_default
Collation	Select an item...
Character type	Select an item...
Connection limit	-1
Template?	

Note: When the preferences setting 'show template databases' is set to false, then template databases won't be displayed in the browser tree.

Figure 71 - Create Database Definition

In the “Security” tab, permissions can be set for database users as well as assign security labels.



Create - Database

Privileges

Grantee	Privileges	Grantor
postgres	☐ ALL ☐ WITH GRANT OPTION ☐ CREATE ☐ WITH GRANT OPTION ☐ TEMPORARY ☐ WITH GRANT OPTION ☐ CONNECT ☐ WITH GRANT OPTION	postgres

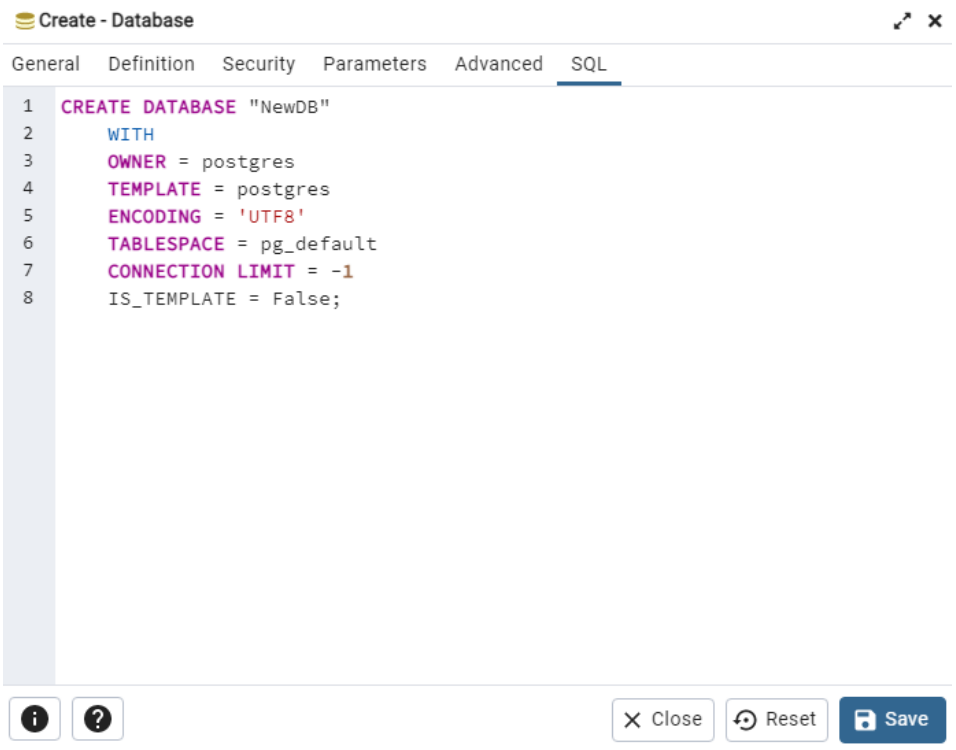
Security labels

Provider	Security label

At least one privilege should be selected.

Figure 72 - Create Database Security

Finally, in the SQL tab, the command that will be executed upon pressing the Save button is displayed. This command will create the database, reflecting the user's selections from the previous tabs:



```
CREATE DATABASE "NewDB"
    WITH
    OWNER = postgres
    TEMPLATE = postgres
    ENCODING = 'UTF8'
    TABLESPACE = pg_default
    CONNECTION LIMIT = -1
    IS_TEMPLATE = False;
```

Figure 73 - Create Database SQL

8.1.3 Schemas

In PostgreSQL, a schema is a logical container used to organize and manage database objects such as tables, views, indexes and functions. Schemas help structure the database by grouping related objects together, allowing for better organization and access control.
In the newly created database named NewDB, the schemas option displays the public schema, which is created automatically:

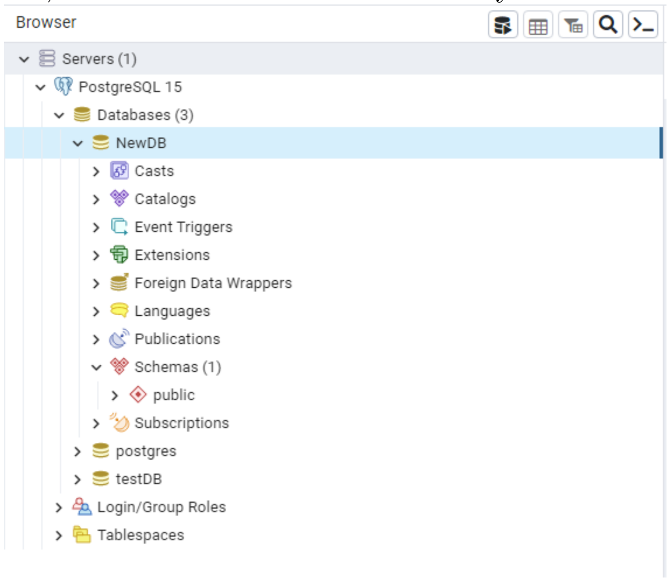


Figure 74 - Database Schema

Expanding the public schema reveals various components that make it up, including:

- Functions
- Sequences
- Tables
- Triggers

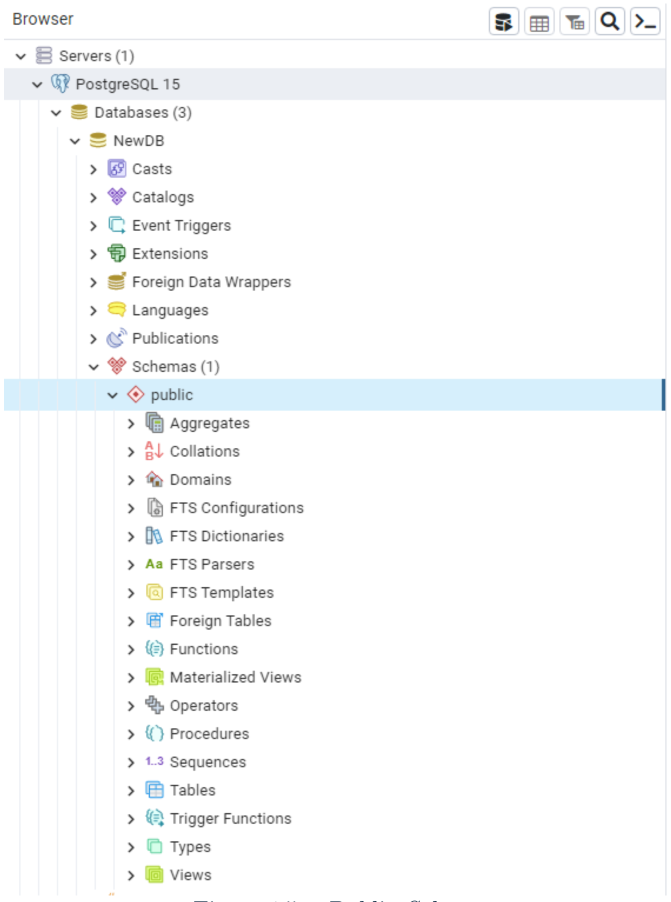


Figure 75 - Public Schema

For the app a new schema, named “app” was created. By right-clicking on the Schemas option, a new schema can be created:

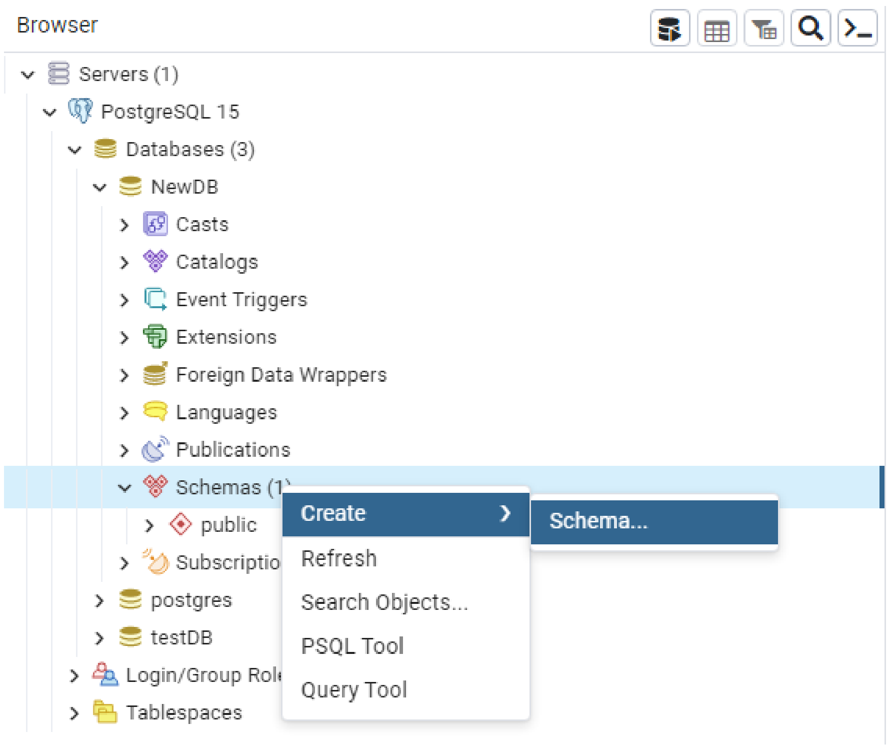
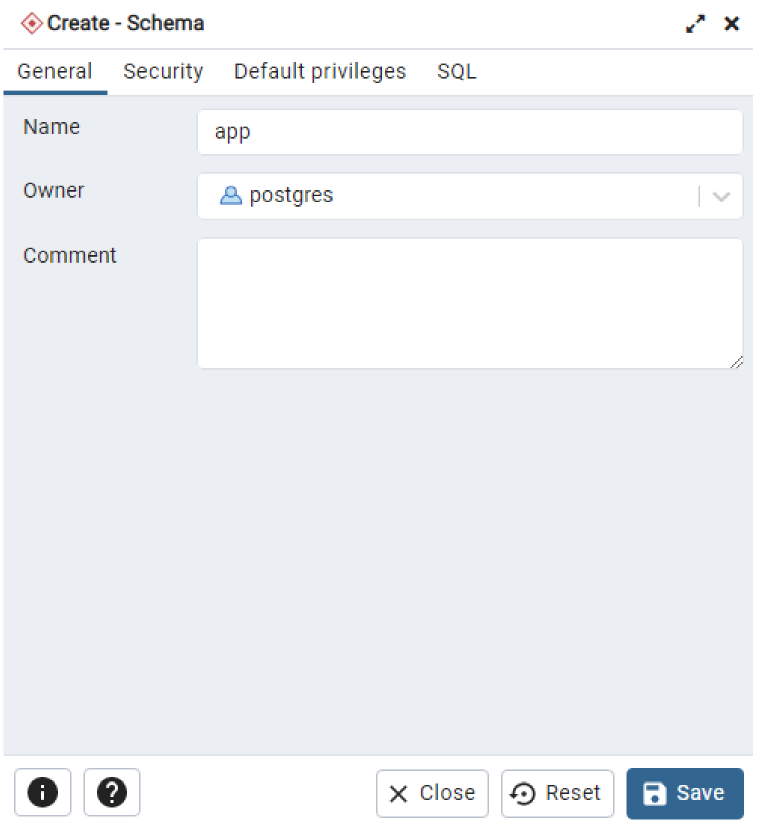


Figure 76 - Create new Schema 1st step

Similarly, to creating a new database, a window appears prompting the user to choose a name for the new schema:



Name: app

Owner: postgres

Comment:

Figure 77 - Create new Schema 2nd step

8.1.4 Tables

The application's database consists of the following tables:

- Users: This table stores information about the users. More specifically 16 columns were created for that purpose and they can split into 3 categories:
    - Front-end editable fields: These columns are visible and modifiable by guests, containing profile information such as first name, last name, email, country and phone number.
    - Front-end read-only fields: These columns are visible to guests but cannot be modified by them, only from an admin user. They include details related to their stay, such as login credentials (username and password), arrival and departure dates and the cleaning program.
    - Back-end fields: These columns are not visible to guests but play a crucial role in data management and business logic. They include the user ID (ensuring uniqueness), an incremental ID, the created date (defaulting to the current timestamp), user role (admin or visitor) for feature access control, device information for statistical purposes and an active status to restrict access to advanced features for users no longer staying at the property.
- Markers: This table holds details about the markers available in the application's dynamic interface. It includes an incremental ID, geographical coordinates (longitude and latitude), title, type, icon and an array of keywords for search purposes.
- Availability: This table tracks the availability of the house and includes only an incremental ID along with a date indicating when the house is not bookable.
- Booking_Request: This table records booking requests, including:
    - Information provided by the user in the form, such as the number of visitors, email address, comments and start & end dates.
    - Information generated by the server during request processing, including the requested date and an info_message, which is an ID created when sending the email with booking details to the host. This ID allows hosts to search for booking requests easily.
- Attack: This table records information about unusual traffic patterns. It typically tracks details of suspicious login activities, such as attempts to use nonexistent usernames, as well as actions that are not permitted within the app, indicating that someone is trying to access the server's resources. In such cases, the user's IP address, the timestamp of the attempted attack and comments related to the specific activity are saved in this table.

In the following diagram, all the tables and their corresponding columns, along with their data types, are displayed.

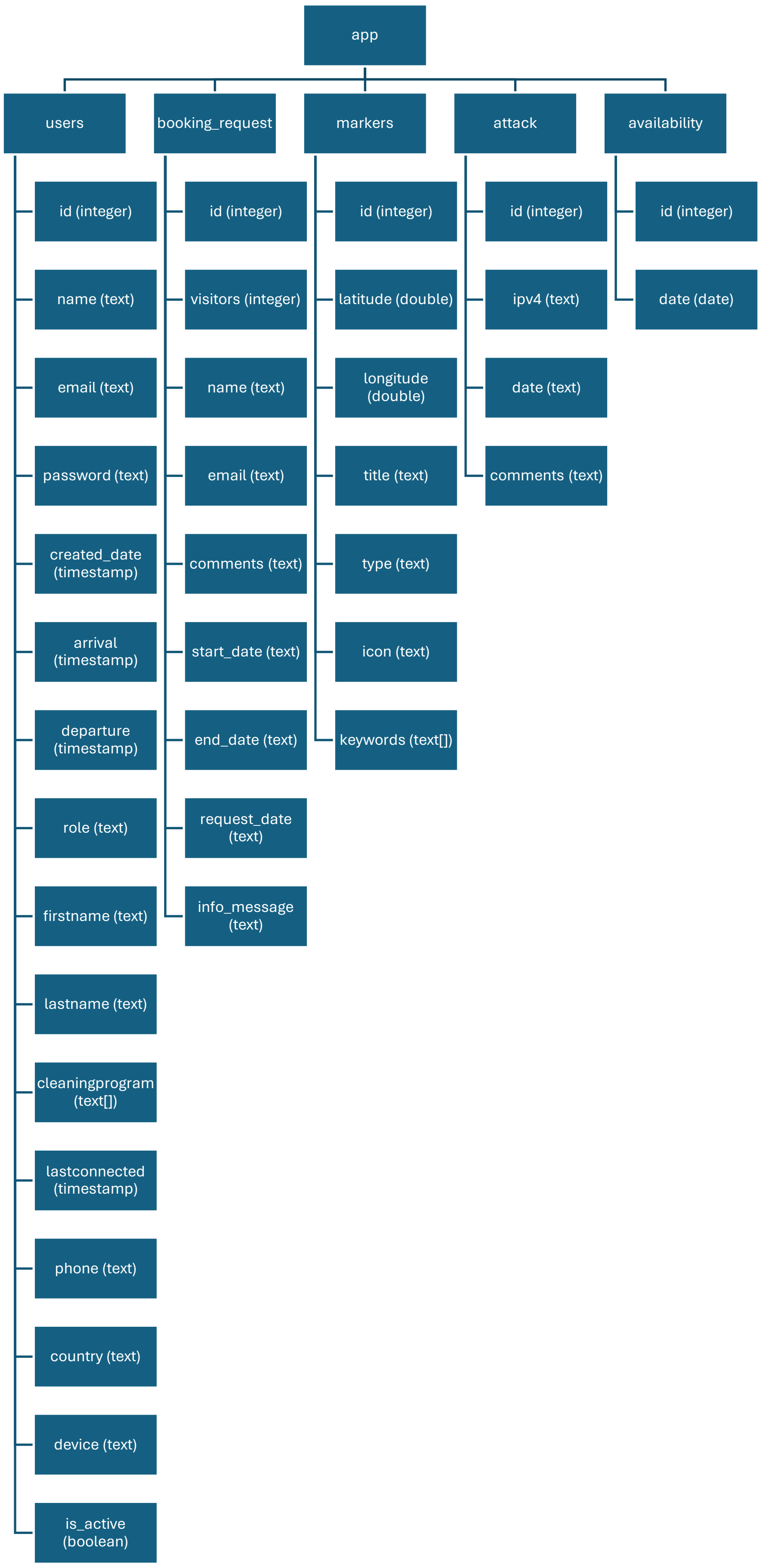


Figure 78 - DataBase Tables

The creation of a new table is possible from the pgadmin4 from the option Schemas → app → Create → Table

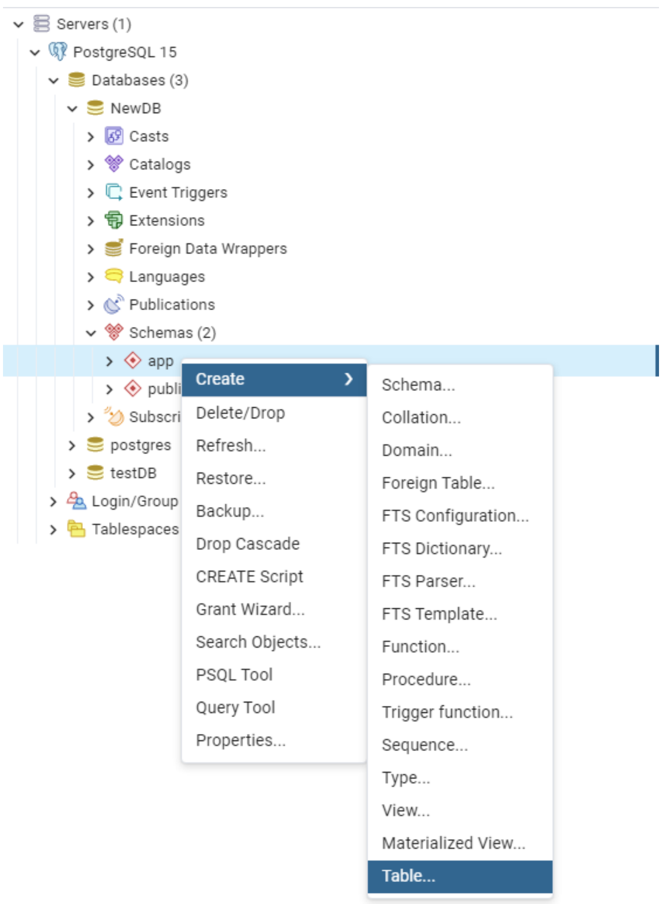
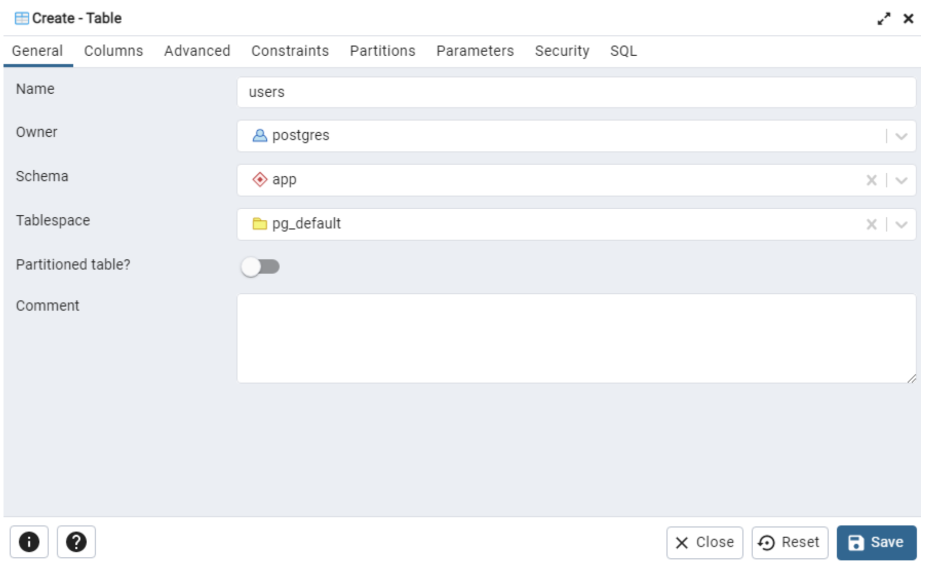


Figure 79 - Create Table 1st step

From there, only a name is required, as the Owner, Schema and Tablespace are predefined. However, these settings can be modified if needed.



Name	users
Owner	postgres
Schema	app
Tablespace	pg_default
Partitioned table?	
Comment	

Figure 80 - Create Table 2nd step

By expanding the Tables option of the current schema, the table that was just created will be displayed.

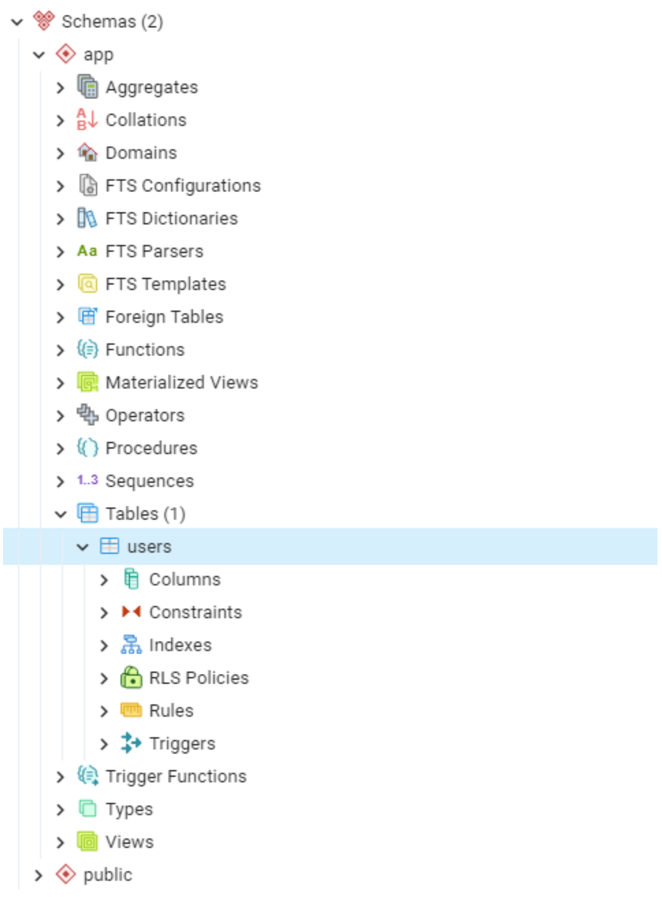


Figure 81 - Users Table

8.1.5 Columns

Columns can be easily added to a specific table by navigating to: Columns → Create → Column:

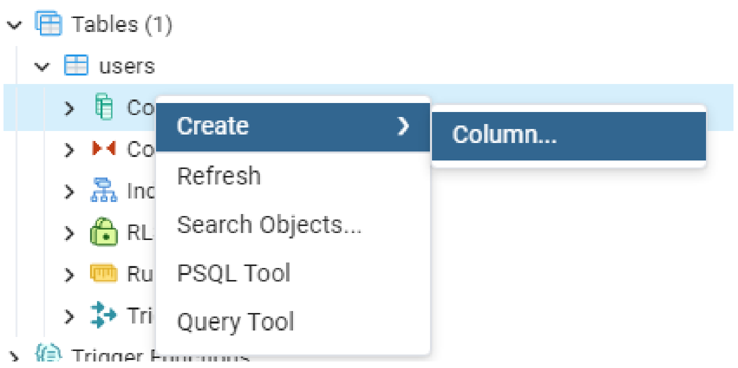


Figure 82 - Add Columns

8.1.6 Query Tool

The Query Tool is a useful feature in pgAdmin that allows for quick and easy execution of queries. If the user wishes to execute an SQL command, they can open the Query Tool after selecting one of the available databases. This can be done either by clicking the icon in the “Browser” tab or by using the shortcut “ALT + SHIFT + Q”.

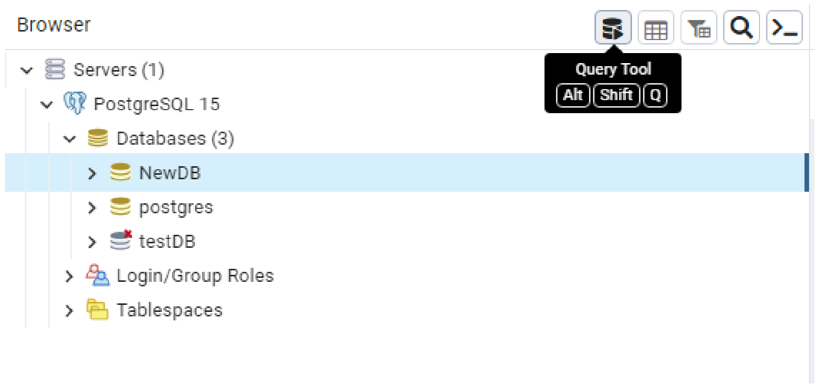
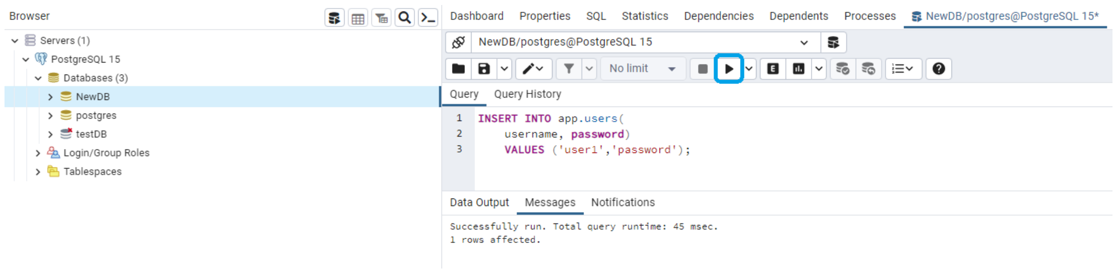


Figure 83 - Query Tool

A new window will appear on the right, where the user can write the desired command, such as an INSERT statement for the users table. Finally, they can click “Execute”. A message indicating either success or an error will be displayed accordingly:



```sql
INSERT INTO app.users(
    username, password)
    VALUES ('user1','password');
```

```
Successfully run. Total query runtime: 45 msec.
1 rows affected.
```

Figure 84 - Query Tool Example

An alternative method is to navigate through the table itself from the left menu: Tables → users → Scripts:

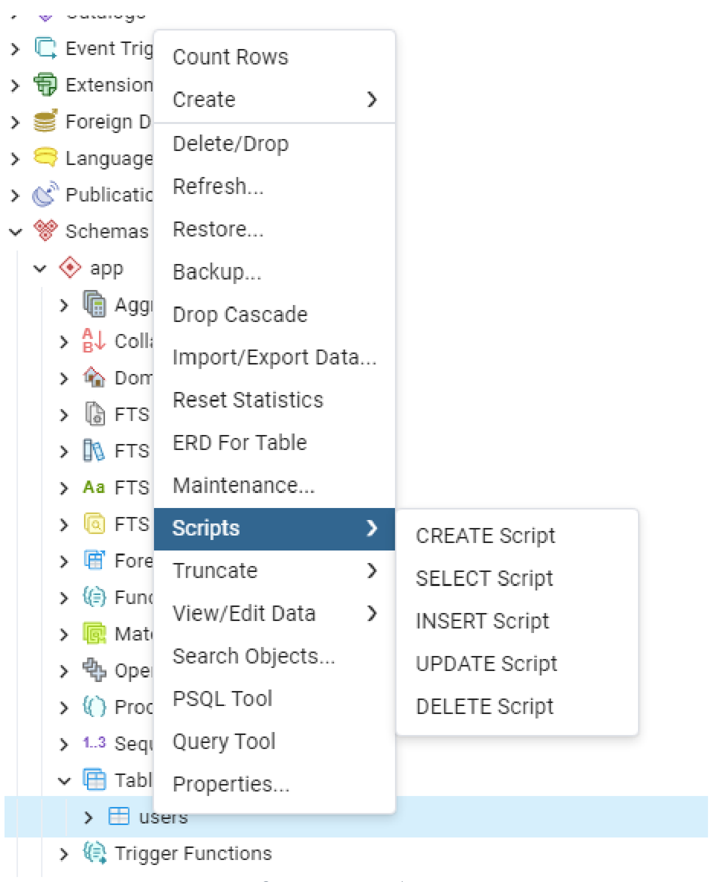
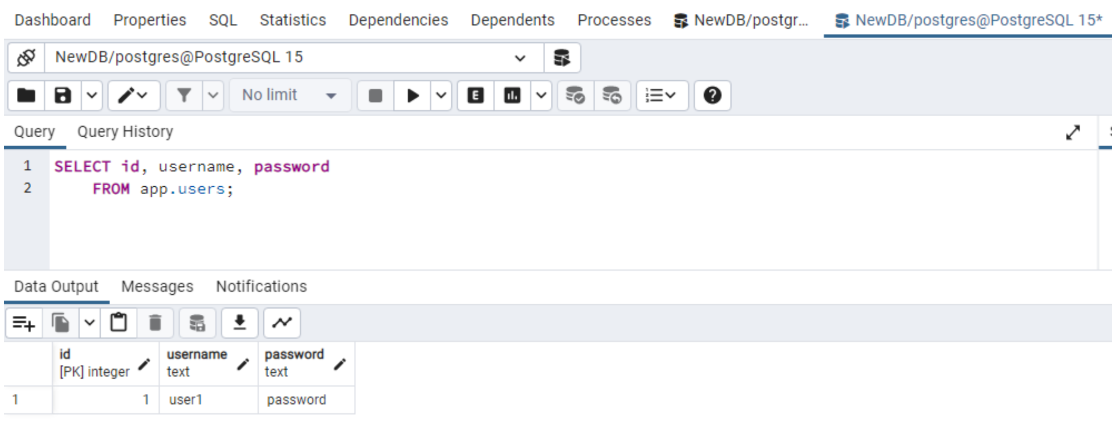


Figure 85 - Query Tool Alternative Method

By selecting SELECT, a new window will appear on the right, similar to the previous one. This time, it will feature a SELECT query for the users table, which can be executed as before, returning the values that were previously entered:



```
SELECT id, username, password
	FROM app.users;
```

	id [PK] integer	username text	password text
1	1	user1	password

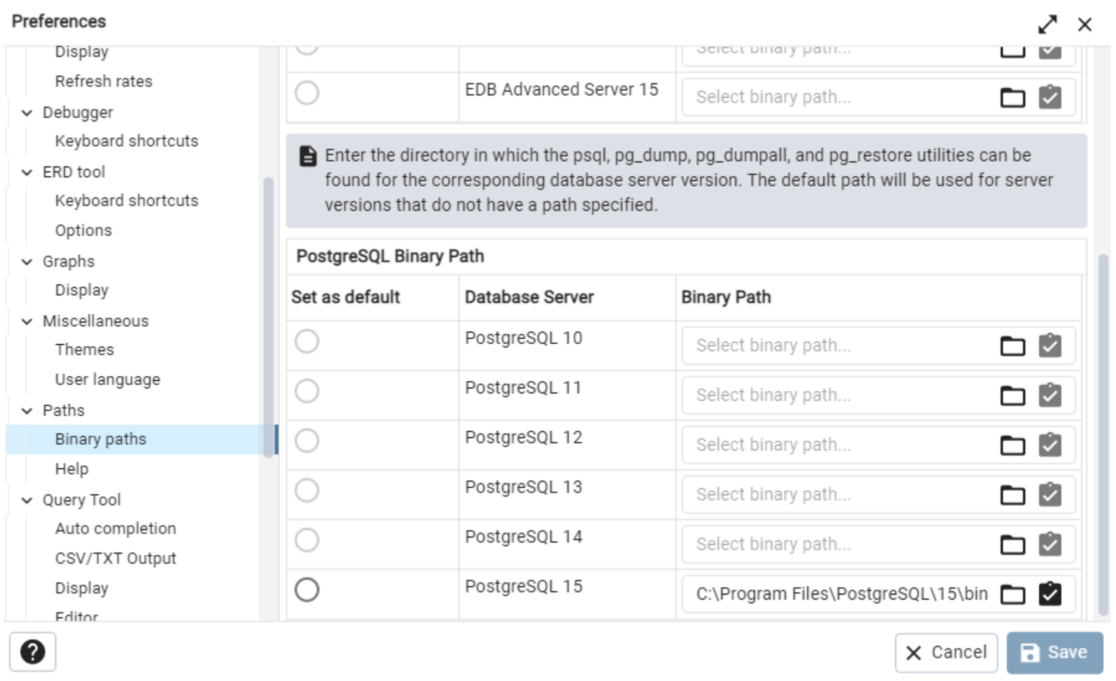
Figure 86 - Query Tool Select

8.1.7 Backup

Creating backups is crucial for any database system. Backups protect data against system failures, natural disasters, or human errors. Without regular backups, users risk losing valuable information, which can severely impact their business or application.

Moreover, in cases of attacks like ransomware, having a backup can be invaluable for data recovery. A good backup strategy should also include testing backups for completeness and maintaining multiple versions for added security.

If a user wants to create a backup of their database by exporting the commands needed for recreation elsewhere, they first need to add the binary path of the installed PSQL on their machine. This can be done via: File → Preferences → Paths → Binary Paths:



	EDB Advanced Server 15	Select binary path...

Enter the directory in which the psql, pg_dump, pg_dumpall, and pg_restore utilities can be found for the corresponding database server version. The default path will be used for server versions that do not have a path specified.

PostgreSQL Binary Path

Set as default	Database Server	Binary Path
○	PostgreSQL 10	Select binary path...
○	PostgreSQL 11	Select binary path...
○	PostgreSQL 12	Select binary path...
○	PostgreSQL 13	Select binary path...
○	PostgreSQL 14	Select binary path...
○	PostgreSQL 15	C:\Program Files\PostgreSQL\15\bin

Figure 87 - Pgadmin4 Backup

Next, locate desired Database to back up: NewDB → Backup:

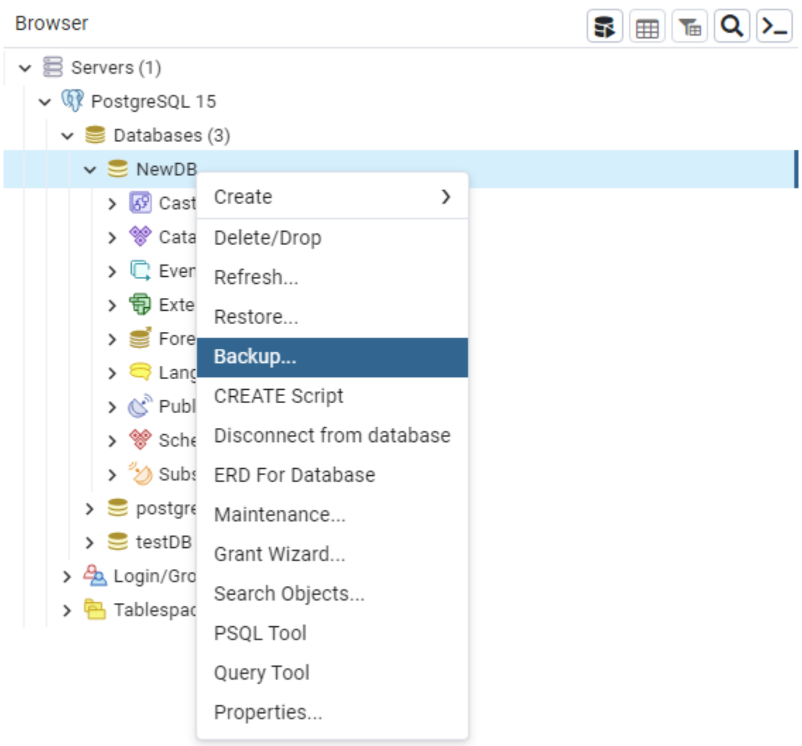
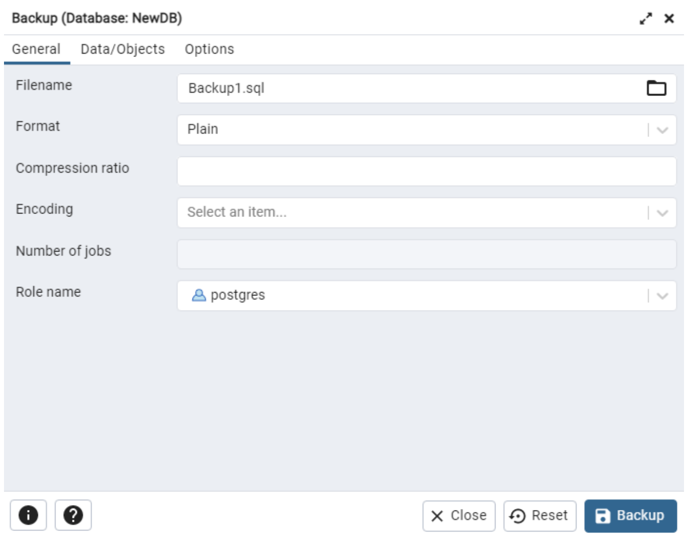


Figure 88 - Select Database to Backup

After following the following configuration, the backup button can be selected:

- First some general information is required, like the file name, that should be .sql with a plain format



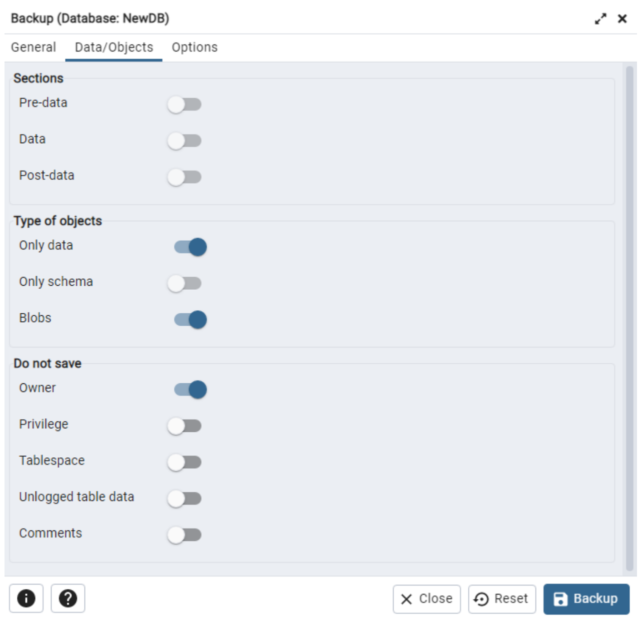
Backup (Database: NewDB)

General | Data/Objects | Options

Filename	Backup1.sql
Format	Plain
Compression ratio	
Encoding	Select an item...
Number of jobs	
Role name	postgres

Figure 89 - Backup General Settings

- After that from the Data/objects tab some additional configuration is required:


Backup (Database: NewDB)

Sections

Pre-data	○
Data	○
Post-data	○

Type of objects

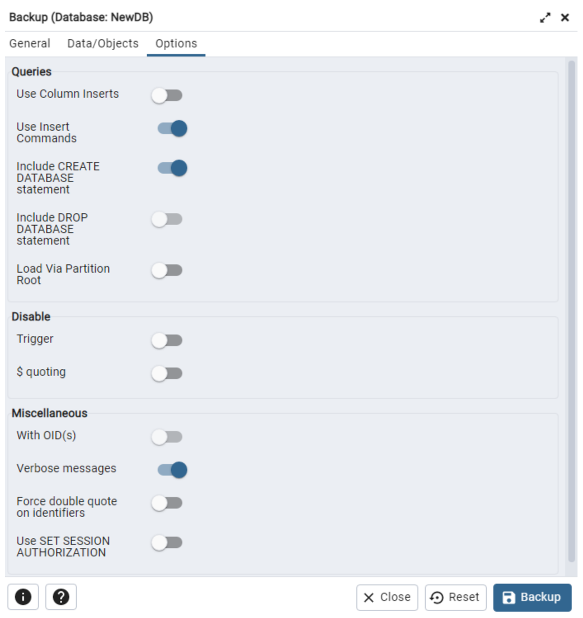
Only data	●
Only schema	○
Blobs	●

Do not save

Owner	●
Privilege	○
Tablespace	○
Unlogged table data	○
Comments	○

Figure 90 - Backup Data Configuration

- Finally the Options tab should be configured according to the settings shown below:


Backup (Database: NewDB)

Queries

Use Column Inserts	○
Use Insert Commands	●
Include CREATE DATABASE statement	●
Include DROP DATABASE statement	○
Load Via Partition Root	○

Disable

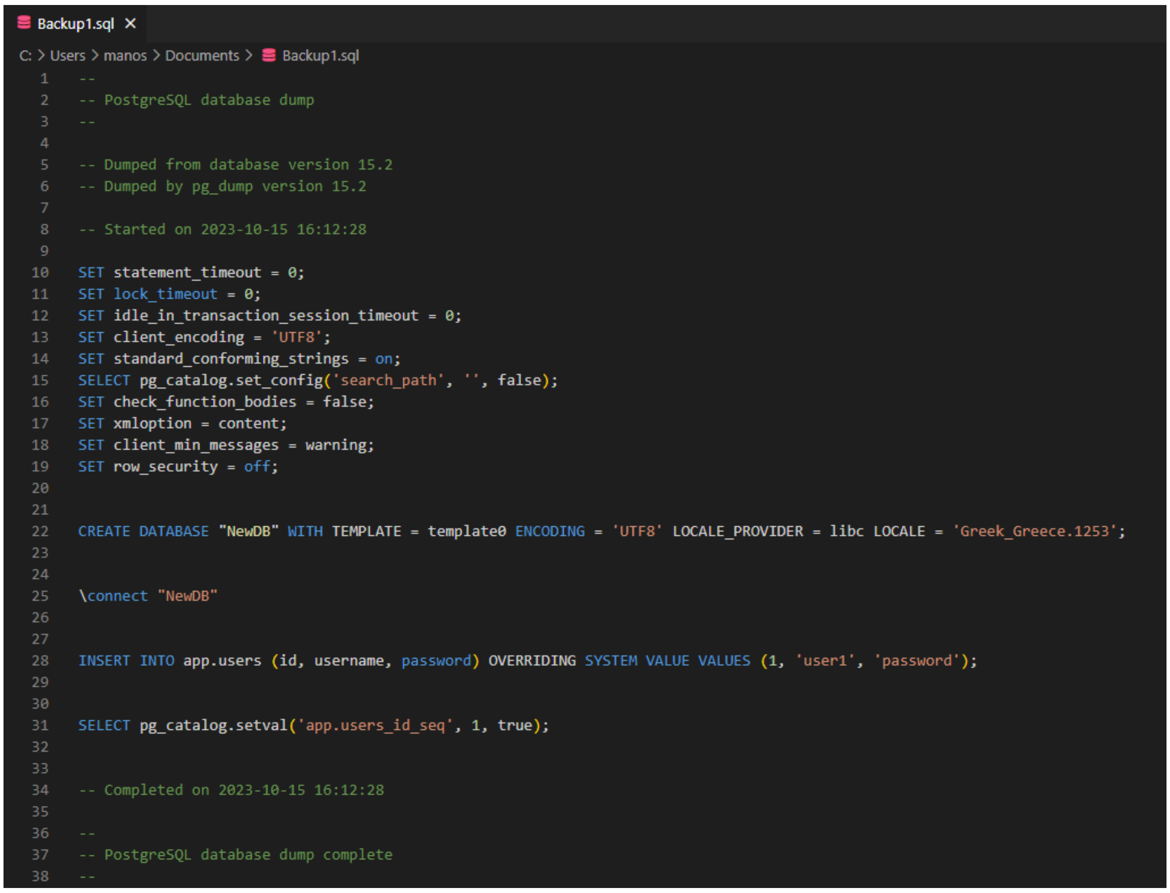
Trigger	○
$ quoting	○

Miscellaneous

With OID(s)	○
Verbose messages	●
Force double quote on identifiers	○
Use SET SESSION AUTHORIZATION	○

Figure 91 - Backup Options

After clicking Backup, a new .sql file will be generated, containing all the SQL commands necessary to recreate the database:


Backup1.sql

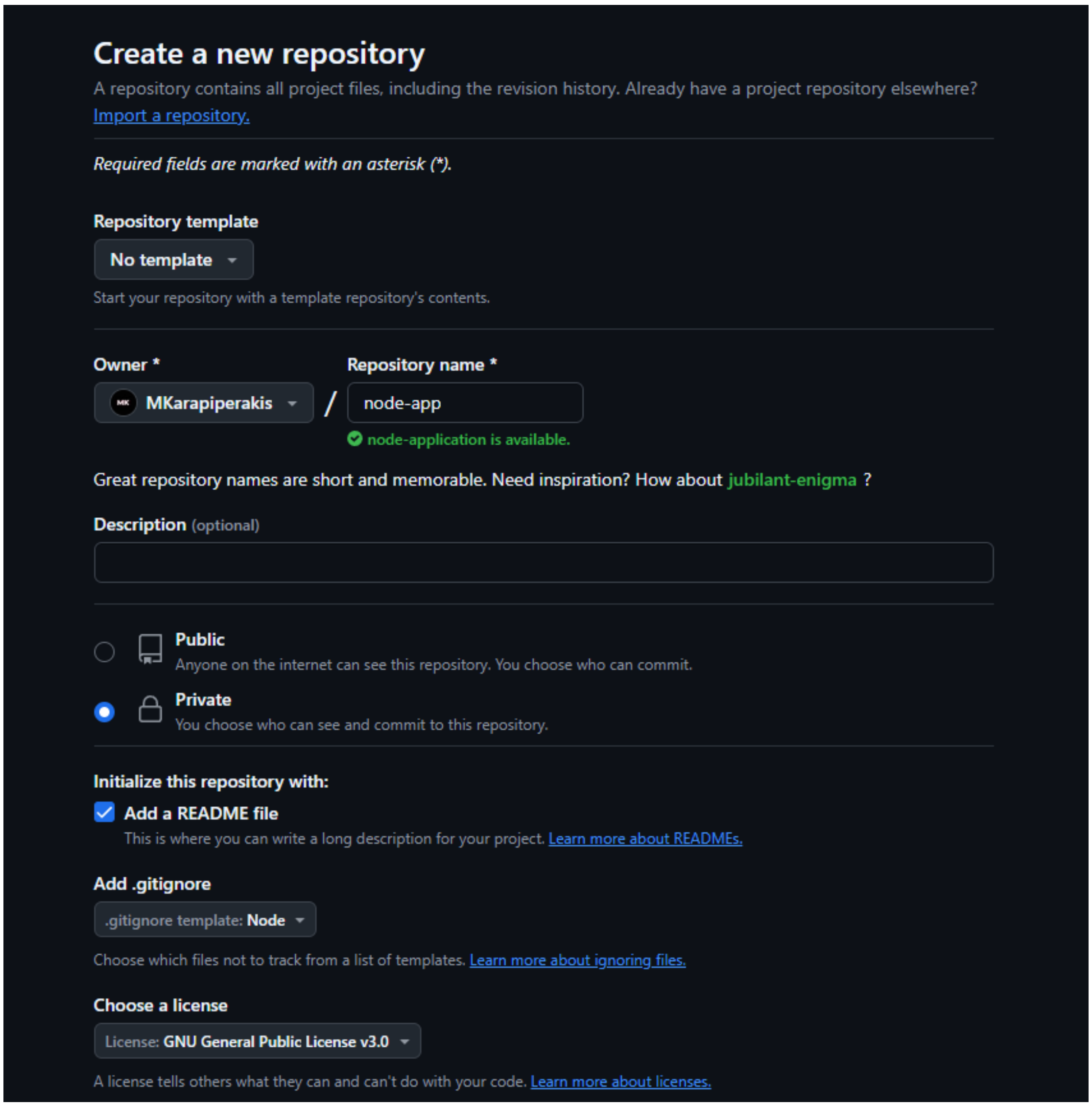
C: > Users > manos > Documents > Backup1.sql

```sql
--
-- PostgreSQL database dump
--

-- Dumped from database version 15.2
-- Dumped by pg_dump version 15.2

-- Started on 2023-10-15 16:12:28

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;


CREATE DATABASE "NewDB" WITH TEMPLATE = template0 ENCODING = 'UTF8' LOCALE_PROVIDER = libc LOCALE = 'Greek_Greece.1253';


\connect "NewDB"


INSERT INTO app.users (id, username, password) OVERRIDING SYSTEM VALUE VALUES (1, 'user1', 'password');


SELECT pg_catalog.setval('app.users_id_seq', 1, true);


-- Completed on 2023-10-15 16:12:28

--
-- PostgreSQL database dump complete
--
```

Figure 92 - Backup File

8.2 Node.js server

The first step was to create a new private GitHub repository to host the server, which will later be linked to Heroku.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Required fields are marked with an asterisk (*).

Repository template: No template

Start your repository with a template repository's contents.

Owner *: MKarapiperakis / Repository name *: node-app

node-application is available.

Great repository names are short and memorable. Need inspiration? How about jubilant-enigma ?

Description (optional)

Public — Anyone on the internet can see this repository. You choose who can commit.

[✓ Private] — You choose who can see and commit to this repository.

Initialize this repository with:

[✓] Add a README file — This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore: .gitignore template: Node

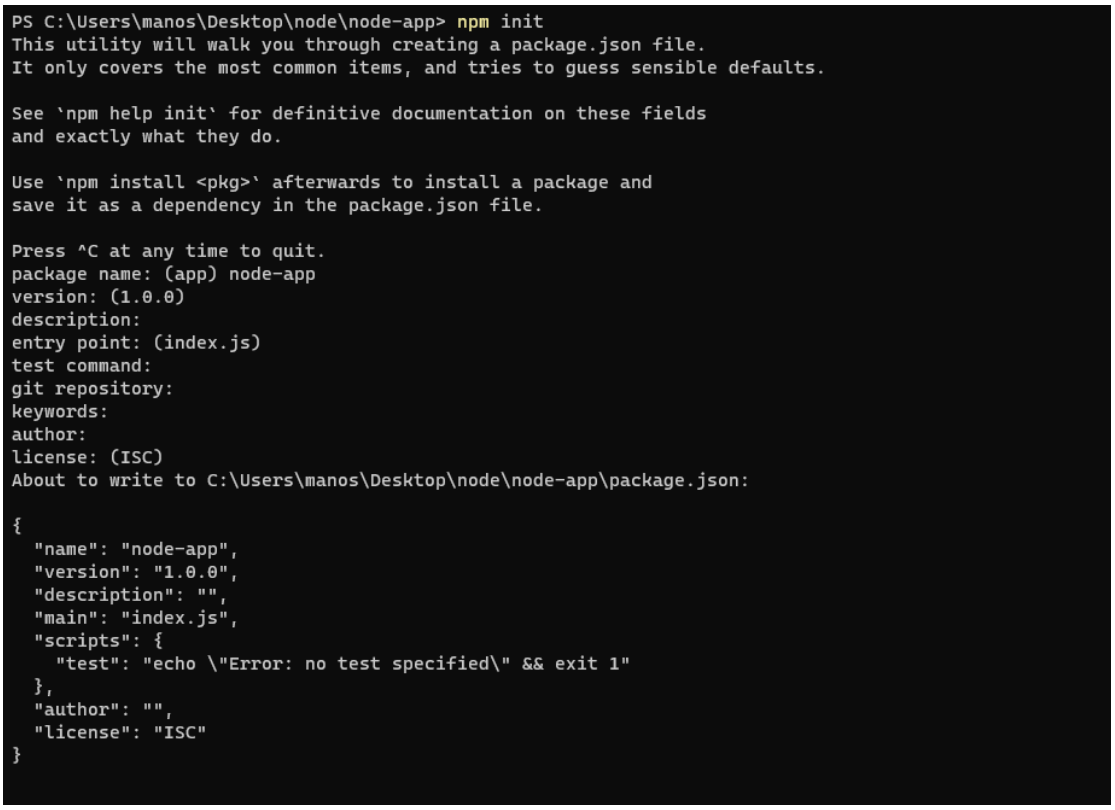
Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license: License: GNU General Public License v3.0

A license tells others what they can and can't do with your code. Learn more about licenses.

Figure 93 - Server Repository

After cloning the project locally, the command "npm init" should be given to initialize the application, where the node version that was used is 21.2.0:

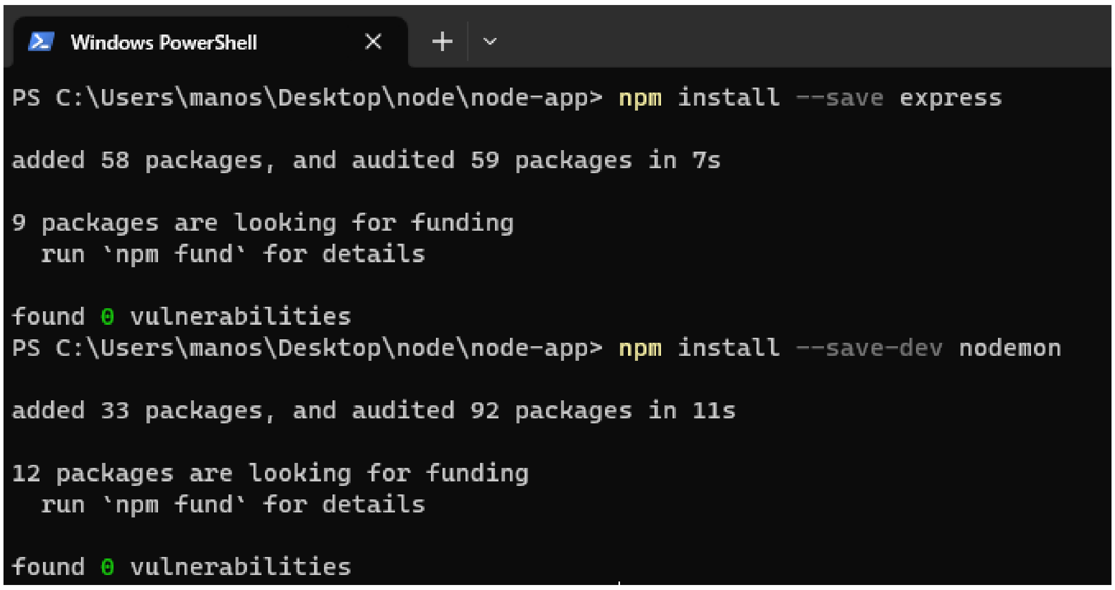


```
PS C:\Users\manos\Desktop\node\node-app> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (app) node-app
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\manos\Desktop\node\node-app\package.json:

{
  "name": "node-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Figure 94 - Initialize the Server

After setting up the repository, the next step is to install two important libraries that will serve as the foundation for the server:

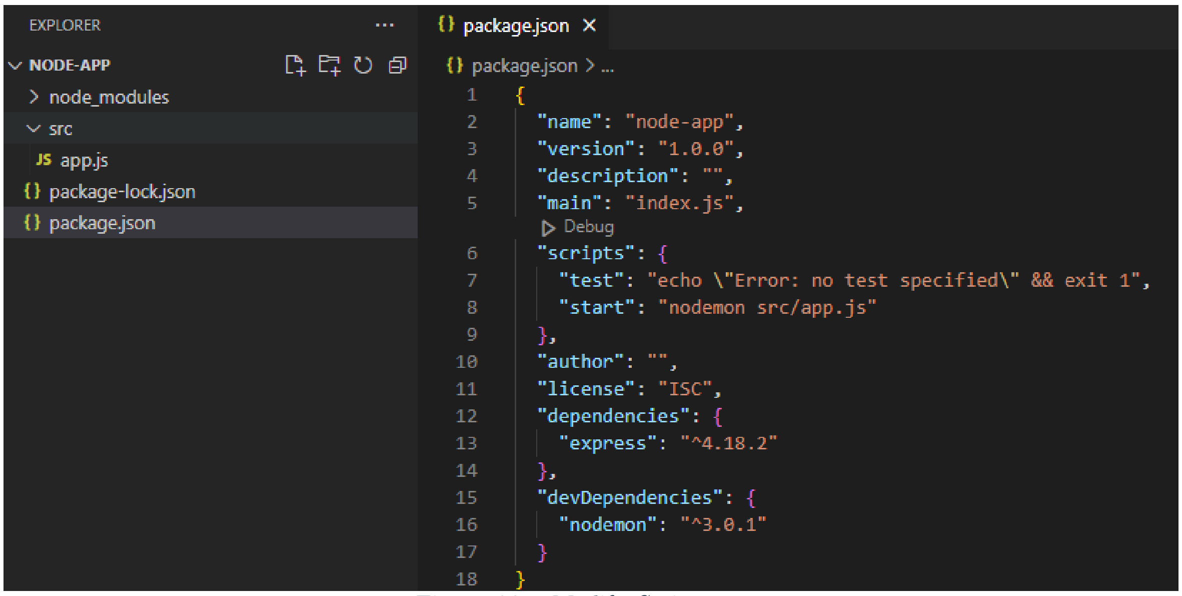
- Express: It provides all the tools needed for handling HTTP requests
- Nodemon: A development tool that automatically restarts the server whenever code changes are detected



```
PS C:\Users\manos\Desktop\node\node-app> npm install --save express

added 58 packages, and audited 59 packages in 7s

9 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\manos\Desktop\node\node-app> npm install --save-dev nodemon

added 33 packages, and audited 92 packages in 11s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Figure 95 - Install Basic Dependencies

Finally, the package.json should be modified by adding the script to start the server:



```json
{
  "name": "node-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon src/app.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

Figure 96 - Modify Scripts

Now the server is ready to accept incoming requests after running the command "npm start"

8.2.1 Application Dependencies

Besides Express and Nodemon, some additional packages are required to implement the business logic, protect the resources and improve the overall performance of the server:

- cors: Enables Cross-Origin Resource Sharing for secure API access.
- swagger-ui-express: Serves API documentation using Swagger in Express.
- yamljs: Parses YAML files into JavaScript objects for configuration.
- express-openapi-validator: Validates API requests and responses based on OpenAPI definitions.
- morgan: HTTP request logger for easier debugging and monitoring.
- dotenv: Loads environment variables from a .env file into process.env.
- express-status-monitor: Provides real-time monitoring of Express server metrics.
- chalk: Adds colored terminal output for better readability in logs.
- dialogflow: Integrates Google's Dialogflow for building conversational AI interfaces
- jsonwebtoken: Handles token-based authentication with JSON Web Tokens (JWT).
- nodemailer: Sends emails from Node.js applications via SMTP or other transport methods.
- pdfkit: Generates PDFs programmatically in Node.js.
- pg: PostgreSQL client for interacting with a PostgreSQL database.
- socket.io: Enables real-time, bidirectional communication between server and clients.
- @socket.io/admin/ui: Provides a user interface to monitor and manage Socket.io events.

8.2.2 Environment Variables

To protect the application's codebase, sensitive information such as API keys is stored securely within .env files. Specifically, the following values are included:

- DATABASE_URL: The database URL used to establish a connection between the server and the PostgreSQL database.
- BASIC_AUTH: A base64-encoded token for basic authentication.
- BASIC_USERNAME: The username used in the app's basic authentication schema to secure resources.
- BASIC_PASSWORD: The password used alongside the username in the app's basic authentication schema to protect resources.
- EMAIL_RECEIVER: The designated recipient for emails, relevant to a specific business process.
- EMAIL_SENDER: The sender's email address, part of a business process implementation.
- PASSWORD_SENDER: A unique password for the email sender, enabling email-sending functionality.
- JWT_PRIVATE_KEY: The private key used for JSON Web Token (JWT) bearer authentication.
- NODE_ENV: The environment setting (production or development), which can be used to adjust configurations for testing or deployment.
- PORT: The port the server operates on (8082).

8.2.3 Connecting the Database to the Node.js Server

To link the previously created PostgreSQL database with the Node.js application, the pg library was used for connecting and querying the database. In this example, the Pool class from the pg package is utilized to manage multiple connections efficiently. The connection is established using environment variables, specifically DATABASE_URL, which stores the PostgreSQL connection string:

```javascript
const { Pool } = require("pg");
require("dotenv").config();

let pool = new Pool({
    connectionString: process.env.DATABASE_URL,
  });

module.exports = pool;
```

8.2.4 Custom Errors

Some custom errors have been implemented to handle errors from the HTTP requests, as shown below:

```
class RestError extends Error {
    constructor(status, message, statusDetail) {
      super(message)
      this.status = status
      this.statusDetail = statusDetail
      this.name = this.constructor.name
      Error.captureStackTrace(this, this.constructor)
    }
  }

  class BadRequestError extends RestError {
    constructor(message, statusDetail = 'Bad Request') { super(400,
message, statusDetail) }
  }

  class AuthError extends RestError {
    constructor(message, statusDetail = 'Unauthorized') { super(401,
message, statusDetail) }
  }

  class ForbiddenError extends RestError {
    constructor(message, statusDetail = 'Forbidden') { super(403,
message, statusDetail) }
  }

  class NotFoundError extends RestError {
    constructor(message, statusDetail = 'Not Found') { super(404,
message, statusDetail) }
  }
```

8.2.5 Authentication Middlewares

The application contains three authentication middlewares:

- Bearer Authentication for APIs: The bearer token grants access to resources on behalf of the user who holds it. When a client requests access to a protected resource, it sends the bearer token in the authorization header of the HTTP request. The server validates the token to confirm the legitimacy of the request. Upon successful user login, after verifying their credentials, a JWT (JSON Web Token) is generated by the server using the JWT_PRIVATE_KEY environment variable to digitally sign the token. Certain requests in the application may require specific user permissions and the bearer token is used in these cases by including it in the request header as “Bearer <token>”. This middleware is responsible for verifying the token's integrity using the same environment variable that signed it. Since bearer tokens are stateless and can be used by anyone who possesses them, it is important to handle them securely, considering the risk of potential data breaches:

```
const jwt = require("jsonwebtoken");
const { AuthError } = require("../lib/errors");

const bearerAuthenticator = (req, scopes, schema) => {
  try {
    const token = req.header("Authorization").replace("Bearer ",
"");
    req.jwtPayload = jwt.verify(token, process.env.JWT_PRIVATE_KEY);
    if (req.openapi.schema["x-acl"]) {
      const jwtScopes = new Set(req.jwtPayload.scopes);
      const reqScopes = new Set(req.openapi.schema["x-acl"]);
      const intersection = new Set(
        [...reqScopes].filter((x) => jwtScopes.has(x))
      );
      if (intersection.size > 0) return true;
      else throw new AuthError("Invalid Scopes");
    }
    return true;
  } catch (e) {
    throw new AuthError("Invalid Token");
  }
};
```

- Basic Authentication for APIs: The bearer authentication that was introduced before, is typically used after a user logs in to grant access to various features of the application for authenticated users (guests). However, some of the application's resources are available to non-registered users (visitors) as well, requiring an additional layer of protection. To secure these resources, a basic authentication scheme is implemented. In this setup, an environment variable called BASIC_AUTH is used on the server, containing a base64-encoded value. For every incoming request that employs basic authentication middleware, the client must include a token in the request header in the format "Basic <token>". The server then verifies the token to ensure proper access control for the requested resource:

```
const jwt = require("jsonwebtoken");
const { AuthError } = require("../lib/errors");

const basicAuthenticator = (req, scopes, schema) => {
  try {
    const header_basic_auth = (req.headers.authorization ||
"").split(" ")[1] || "";
    const [headerUsername, headerPassword] =
Buffer.from(header_basic_auth, "base64")
      .toString()
      .split(":");

    const server_basic_auth = process.env.BASIC_AUTH;
    const [serverUsername, serverPassword] =
Buffer.from(server_basic_auth, "base64")
      .toString()
      .split(":");

    if (headerUsername && headerPassword && headerUsername === serv-
erUsername && headerPassword === serverPassword) {
      return true;
    } else throw new AuthError("Invalid Token");
  } catch (e) {
    throw new AuthError("Invalid Token");
  }
};
```

- Basic Authentication to get access to the API documentation [24]: A basic authentication scheme is also used to protect a specific endpoint where the API documentation is hosted. Since this endpoint contains sensitive information that should not be publicly accessible, additional security measures are required. To achieve this, two environment variables, BASIC_USERNAME and BASIC_PASSWORD, are used. When a user attempts to access the protected endpoint, a login prompt will appear (dialog box), requesting credentials:

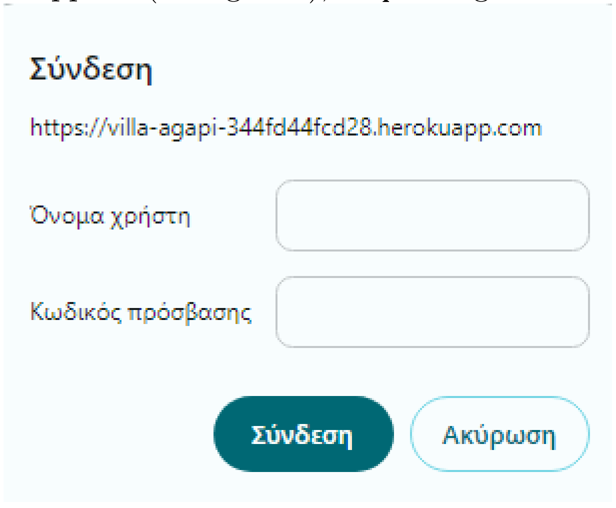


Figure 97 - Basic Authentication Dialog

The middleware will then compare the provided username and password with the values stored in the environment variables. If they match, the user is granted access to the endpoint:

```
const authheader = req.headers.authorization;


  if (!authheader) {
    let err = new Error("You are not authenticated!");
    res.setHeader("WWW-Authenticate", "Basic");
    err.status = 401;
    return next(err);
  }

  res.setHeader("Cache-Control", "no-cache, no-store, must-revali-
date");
  res.setHeader("Pragma", "no-cache");
  res.setHeader("Expires", "0");

  const auth = new Buffer.from(authheader.split(" ")[1], "base64")
    .toString()
    .split(":");
  const username = auth[0];
  const pass = auth[1];

  if (
    username == process.env.BASIC_USERNAME &&
    pass == process.env.BASIC_PASSWORD
  ) {
    next();
  } else {
    let err = new Error("You are not authenticated!");
    res.setHeader("WWW-Authenticate", "Basic");
    err.status = 401;
    return next(err);
  }
```

The following image provides a visual representation of the process:

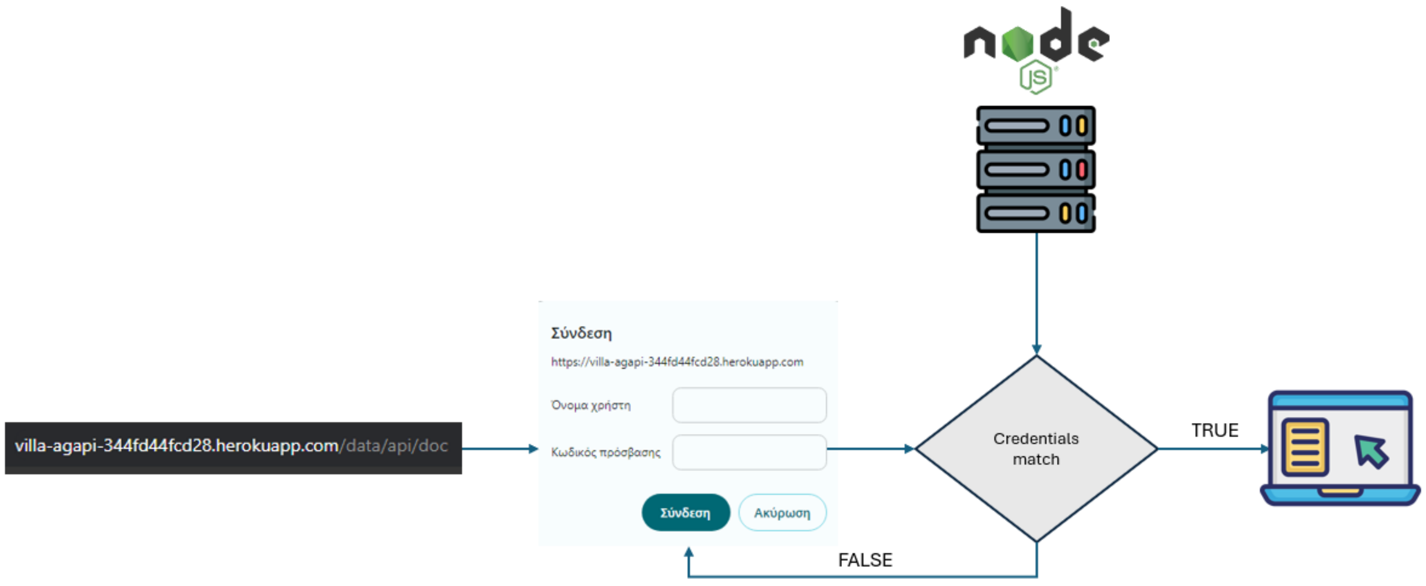
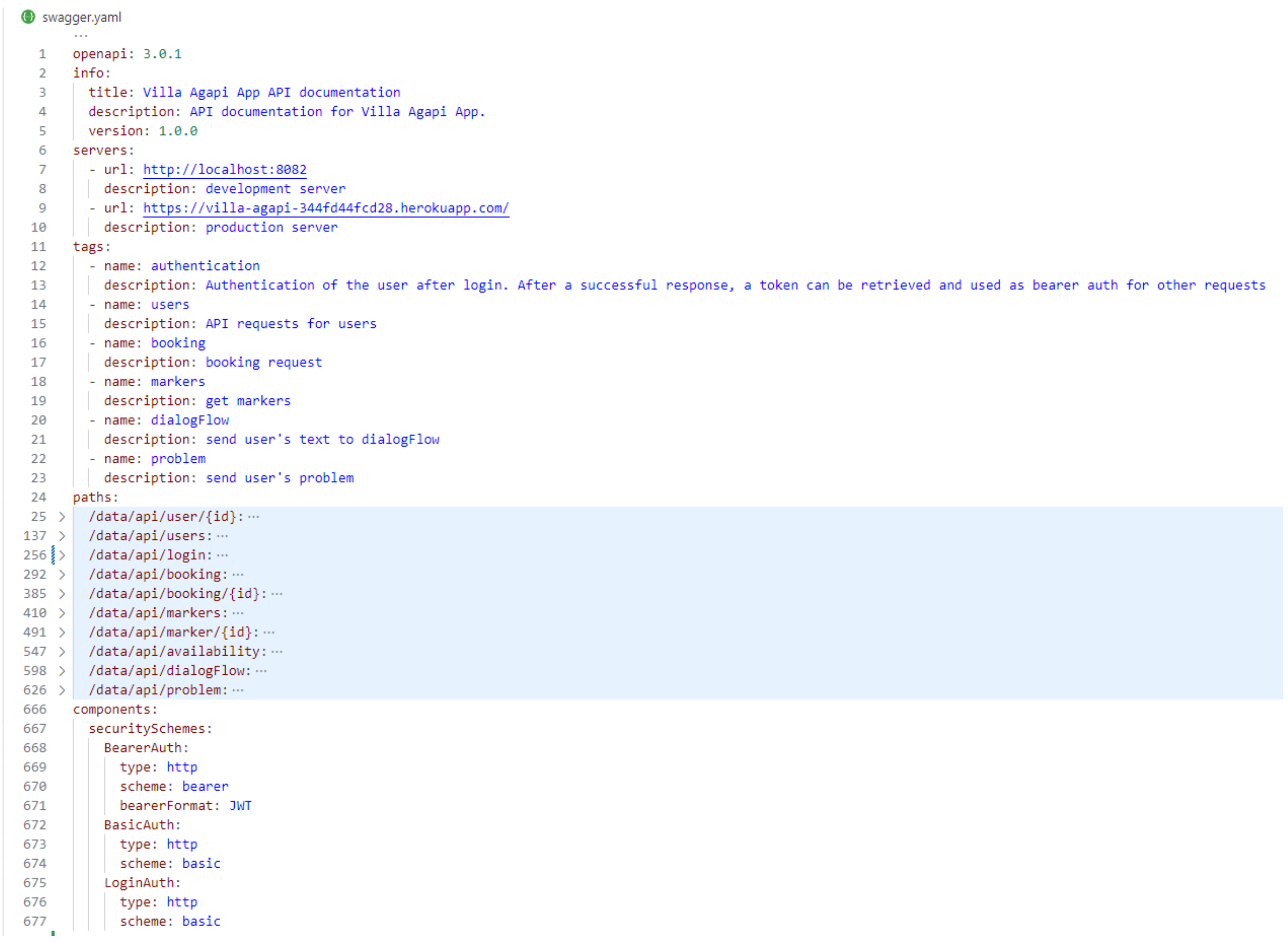

villa-agapi-344fd44fcd28.herokuapp.com/data/api/doc
Σύνδεση
https://villa-agapi-344fd44fcd28.herokuapp.com
Όνομα χρήστη
Κωδικός πρόσβασης
Σύνδεση
Ακύρωση
Credentials match
TRUE
FALSE

Figure 98 - Basic Authentication Middleware

8.2.6 API Documentation

The API documentation was introduced in the previous section, where a basic authentication scheme was implemented to restrict access to authorized users only. API documentation is crucial for defining the structure of available APIs, providing use cases, usage instructions, sample request bodies and expected response bodies. A swagger.yaml file is used to organize and present this information, covering various aspects of the API requests. The file consists of the following sections:

- OpenAPI version: Specifies the current OpenAPI version being used (3.0.1).
- Info: Contains metadata about the documentation, such as the title, description and version.
- Servers: Lists the available servers, including both development and production environments, along with their respective URLs.
- Paths: Includes the API endpoints with the available requests and methods.
- Tags: Helps categorize and organize requests into sections based on their purpose.
- Components: Defines reusable components, including security schemas.



swagger.yaml

```yaml
openapi: 3.0.1
info:
  title: Villa Agapi App API documentation
  description: API documentation for Villa Agapi App.
  version: 1.0.0
servers:
  - url: http://localhost:8082
    description: development server
  - url: https://villa-agapi-344fd44fcd28.herokuapp.com/
    description: production server
tags:
  - name: authentication
    description: Authentication of the user after login. After a successful response, a token can be retrieved and used as bearer auth for other requests
  - name: users
    description: API requests for users
  - name: booking
    description: booking request
  - name: markers
    description: get markers
  - name: dialogFlow
    description: send user's text to dialogFlow
  - name: problem
    description: send user's problem
paths:
  /data/api/user/{id}: ···
  /data/api/users: ···
  /data/api/login: ···
  /data/api/booking: ···
  /data/api/booking/{id}: ···
  /data/api/markers: ···
  /data/api/marker/{id}: ···
  /data/api/availability: ···
  /data/api/dialogFlow: ···
  /data/api/problem: ···
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
    BasicAuth:
      type: http
      scheme: basic
    LoginAuth:
      type: http
      scheme: basic
```

Figure 99 - Swagger.yaml

8.2.7 Routes

Routes specify the paths through which the server handles client requests, with each route linked to a particular HTTP method. In this application, routes are categorized into two types:

- Read routes: These paths are responsible for fetching data from the database or executing custom server logic without modifying any database tables. They typically correspond to safe methods, such as GET.
- Write routes: These paths handle writing data to the database and are generally associated with unsafe methods like PUT and POST, as they alter or create records in the database.

These routes are organized into two files: data-read.js and data-write.js, both located within the routes folder. The following diagram illustrates how the routes are structured within the application:

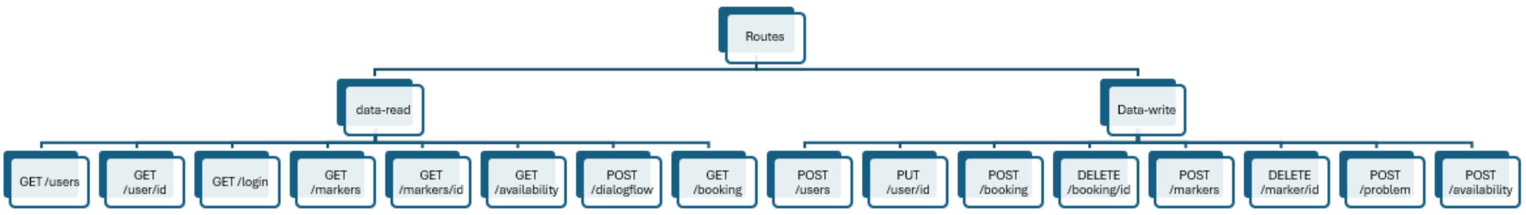


Figure 100 - Application Routes

8.2.7.1 Read Routes

The application offers the following read routes:

- GET /users: Retrieves a list of all users in the application. Requires Bearer authentication.
- GET /user/{id}: Retrieves user details by ID. Requires Bearer authentication.
- POST /login: Authenticates and logs in a user. Requires Basic authentication.
- GET /markers: Fetches a list of markers.
- GET /marker/{id}: Retrieves a marker by ID.
- GET /availability: Fetches property availability details.
- POST /dialogflow: Facilitates message exchanges with the Google Dialogflow virtual assistant. Requires Basic authentication.
- GET /booking: Retrieves booking requests. Requires Bearer authentication.

8.2.7.2 Write Routes

The application offers the following write routes:

- POST /users: Creates a new user. Requires Bearer authentication.
- PUT /user/{id}: Updates user details. Requires Bearer authentication.
- POST /booking: Submits a booking request. Requires Bearer authentication.
- DELETE /booking/{id}: Deletes a booking by ID. Requires Bearer authentication.
- POST /markers: Creates a new marker. Requires Bearer authentication.
- DELETE/marker/{id}: Deletes a marker by ID. Requires Bearer authentication.
- POST /problem: Reports a problem. Requires Bearer authentication.
- POST /availability: Updates property availability. Requires Bearer authentication.

8.2.8 Controllers

For each router, there is a controller responsible for executing the respective logic. The controllers are all organized in the same folder named “controllers” in different JavaScript files related to their functionality. The application offers the following controllers:

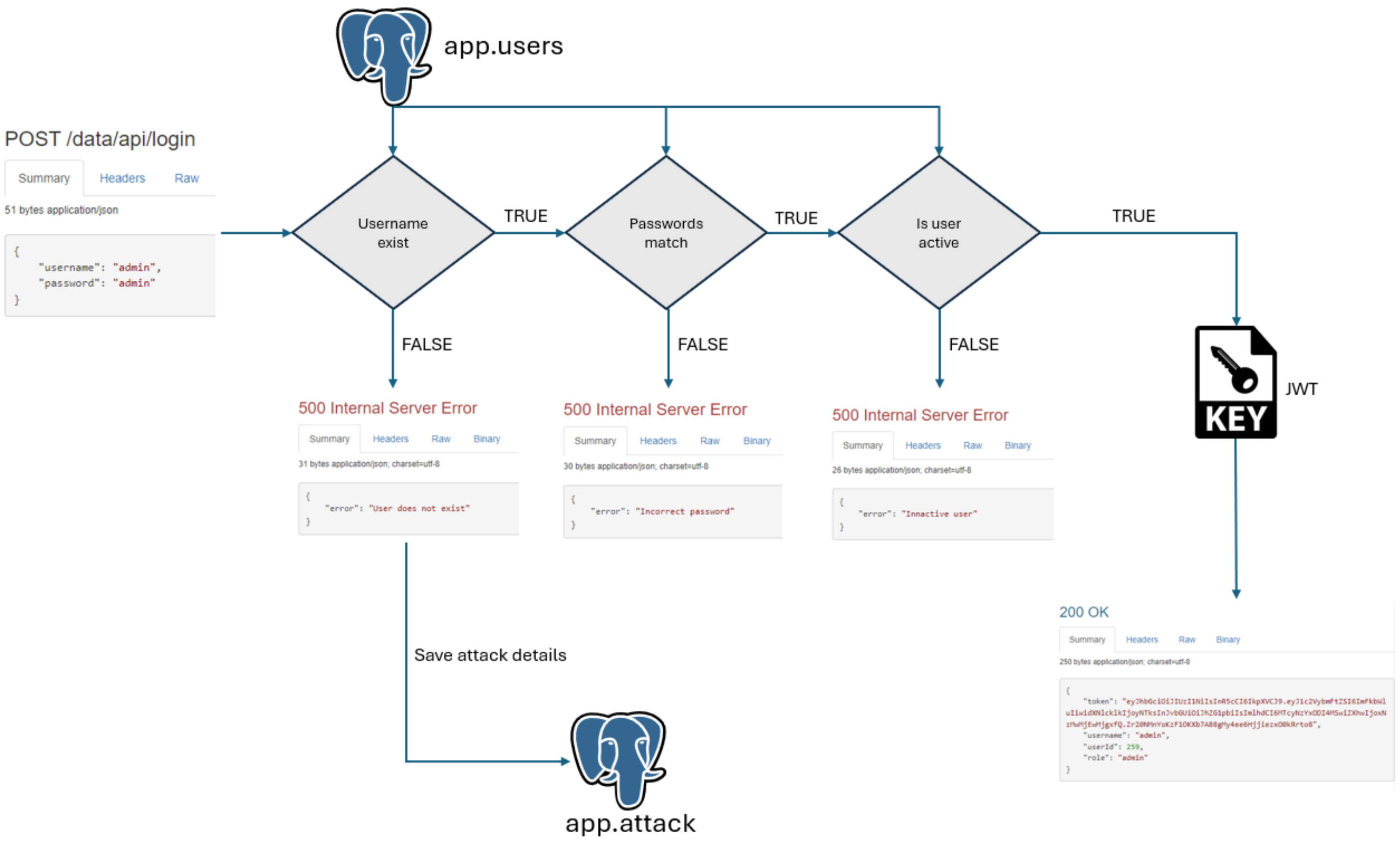
- login: The request body of this controller is
    - username: The username, the user filled in the login form

- password: The password, the user filled in the login form

These values are stored in two constant variables since they remain unchanged and are used in a query aiming the table app.users, to check if the username exists in the database. The query is executed using the previously implemented *pg* library from another file. If the query returns results, the provided password is compared with the stored hashed password using the bcrypt package. If the credentials match, the user's status is verified by checking the "is_active" column from the query results.
If the user is active, a new JWT token is generated using the *jwt* package, which includes the following information:

- username
- userId
- role

This token is signed using the JWT_PRIVATE_KEY from the environment variables and is set to expire in 30 days. The token is then returned to the user with a status code of 200. If the user is inactive, a response with a status code of 500 and the message "Inactive user" is returned.
If the username is correct but the password is incorrect, the client receives a response with a status code of 500 and the message "Incorrect password".
If the database query returns no results, it indicates that the username does not exist. This could suggest a potential attack, though it might simply be a typo. To handle this case, the user's IPv4 address is extracted from the request headers and a record of the attempted attack is logged in the app.attack database table, along with relevant comments and the current timestamp. The client receives a response with a status code of 500 and the message "User does not exist", without being informed of the attack logging, as this is a security measure handled in the background.
The entire asynchronous request is wrapped in a try-catch block. If any errors occur during processing, a server error response with a status code of 500 and the message "An error occurred while fetching users" is returned.



POST /data/api/login

51 bytes application/json

```
{
    "username": "admin",
    "password": "admin"
}
```

500 Internal Server Error

31 bytes application/json; charset=utf-8

```
{
    "error": "User does not exist"
}
```

500 Internal Server Error

30 bytes application/json; charset=utf-8

```
{
    "error": "Incorrect password"
}
```

500 Internal Server Error

26 bytes application/json; charset=utf-8

```
{
    "error": "Inactive user"
}
```

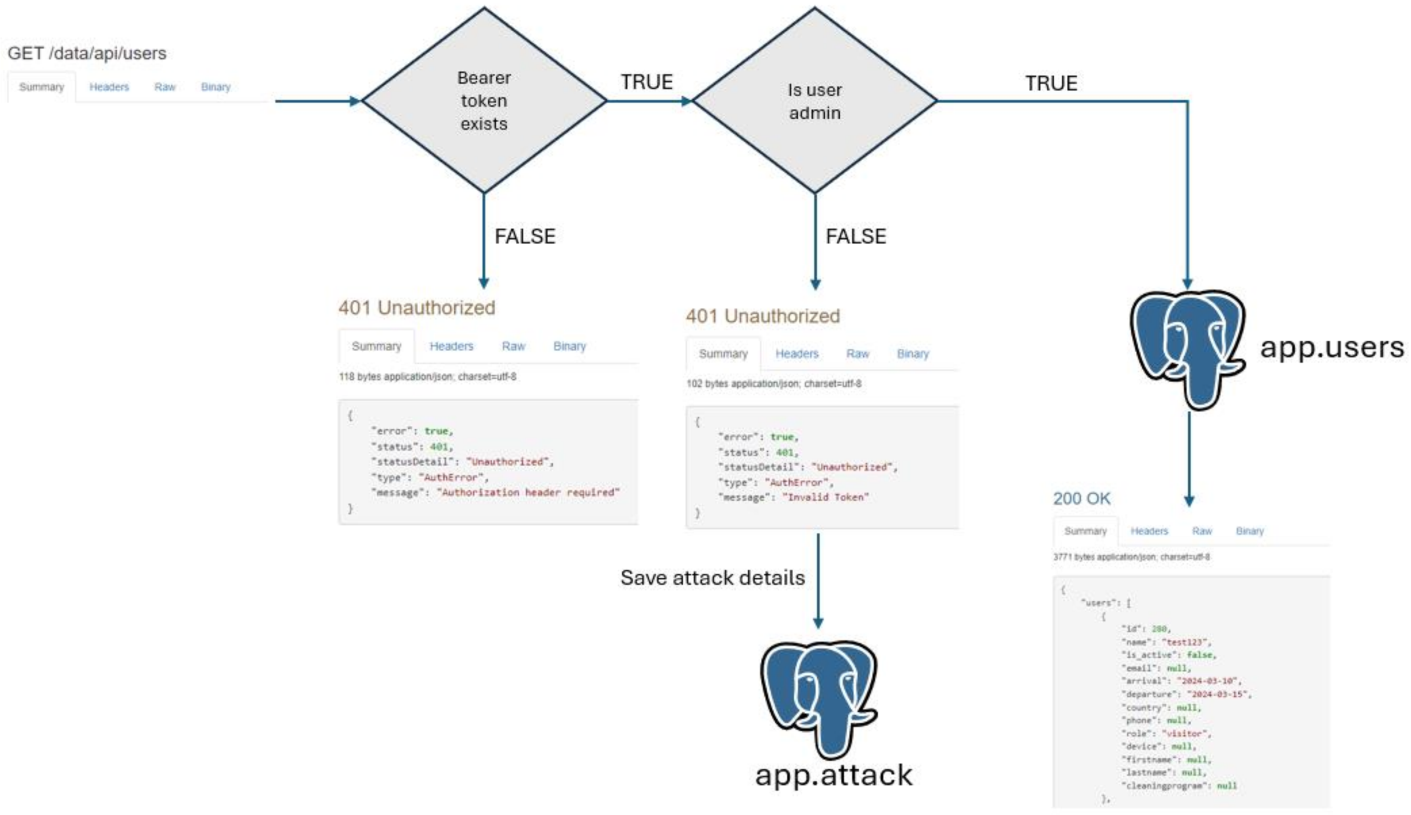
200 OK

250 bytes application/json; charset=utf-8

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWl
uIiwidXNlcklkIjoyNTksInJvbGUiOiJhZG1pbiIsImlhdCI6MTcyNzYxODI4MSwiZXhwIjoxN
zMwMjEwMjgxfQ.Zr20NMnYoKzF1OKXb7A88gMy4ee6HjjiezxO0kRrto8",
    "username": "admin",
    "userId": 259,
    "role": "admin"
}
```

Figure 101 - Login Controller

- getUsers: This controller is responsible for returning all available users. Since it provides sensitive information, access should be restricted to users with the "admin" role. The user's role is embedded in the JSON Web Token (JWT) found in the request headers. If the JWT is missing, a response with status code 401 and the error message "Authorization Header Required" is returned. If the token is present, the role validation process checks whether the user has the "admin" role. If the role is not "admin", this is considered a potential bearer authentication attack, where someone may be attempting to access sensitive data with an invalid token. In such cases, details of the attack, including the user's IPv4 address, relevant comments and the current timestamp, are logged in the app.attack database table. If the user holds the "admin" role, a query is executed to retrieve all users' information from the app.users database table. The data is then returned to the client with a status code of 200 and the user information in the response body:



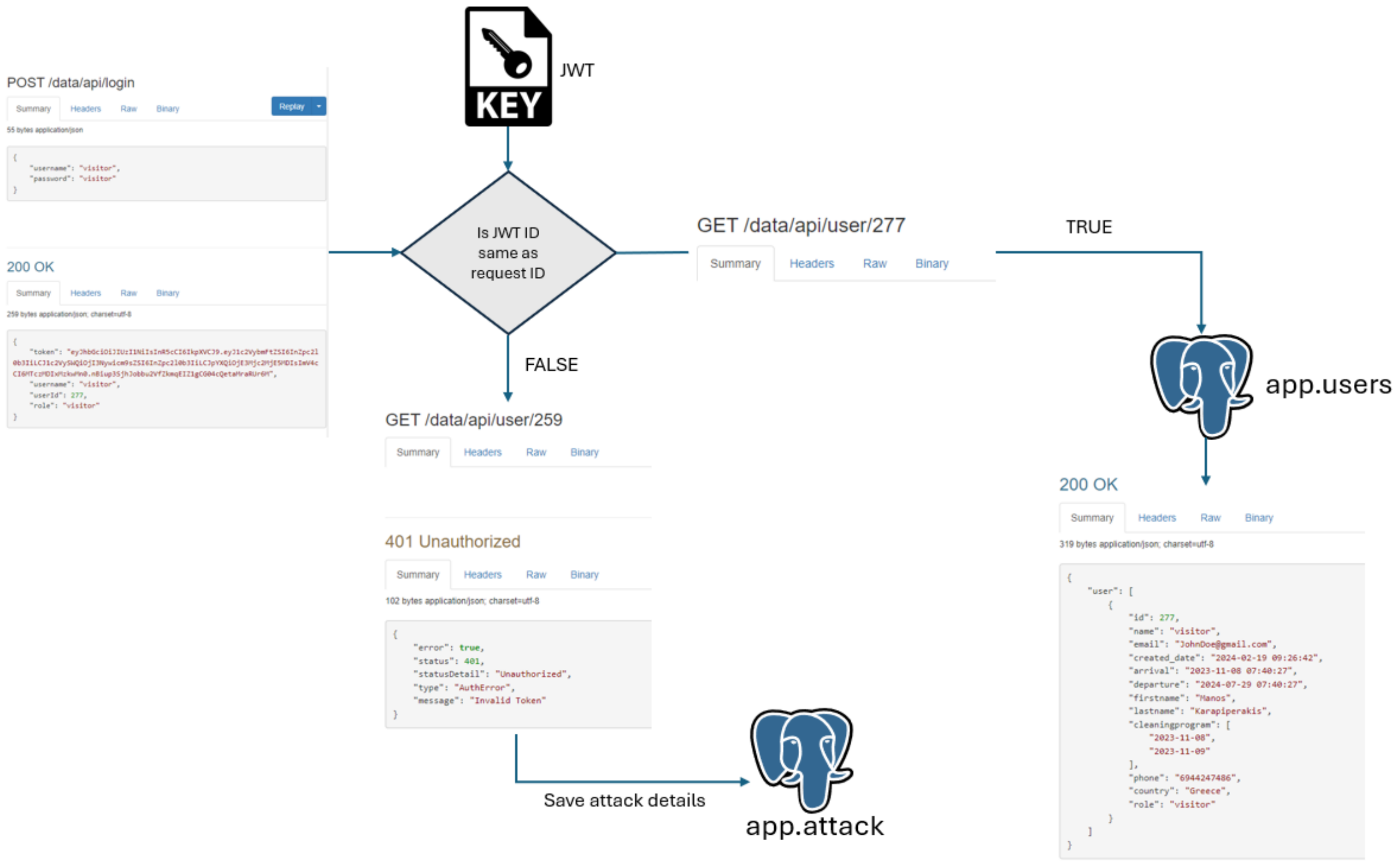
```
118 bytes application/json; charset=utf-8
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

```
102 bytes application/json; charset=utf-8
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Invalid Token"
}
```

```
3771 bytes application/json; charset=utf-8
{
    "users": [
        {
            "id": 280,
            "name": "test123",
            "is_active": false,
            "email": null,
            "arrival": "2024-03-10",
            "departure": "2024-03-15",
            "country": null,
            "phone": null,
            "role": "visitor",
            "device": null,
            "firstname": null,
            "lastname": null,
            "cleaningprogram": null
        },
```

Figure 102 - Get Users Controller

- getUser: This controller retrieves information about a specific user based on the user ID provided in the request parameters. Since this is private information, only the authenticated user should have access to their own data. Therefore, a bearer authentication token is required, along with an additional ID check. The user ID in the JWT payload must match the one in the request parameters. This ensures that users can only access their own information, preventing access to others' data. If a user attempts to access information belonging to another user using their token, it is treated as an attack. In such cases, details including the user's IPv4 address, relevant comments and the current timestamp are logged in the app.attack database table. If the request passes the authentication and ID check, a query is executed against the app.users table to retrieve the specified user's information. The data is returned in the response body with a status code of 200:



POST /data/api/login

55 bytes application/json

```
{
    "username": "visitor",
    "password": "visitor"
}
```

200 OK

259 bytes application/json; charset=utf-8

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InZpc2l
0b3IiLCJ1c2VySWQiOjI3Nywicm9sZSI6InZpc2l0b3IiLCJpYXQiOjE3Mjc2MjE5MDIsImV4c
CI6MTczMDIxMzkwMn0.nBiup35jhJobbu2VfZkmqEIZ1gCG04cQetaHraRUr6M",
    "username": "visitor",
    "userId": 277,
    "role": "visitor"
}
```

GET /data/api/user/277

GET /data/api/user/259

401 Unauthorized

102 bytes application/json; charset=utf-8

```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Invalid Token"
}
```

200 OK

319 bytes application/json; charset=utf-8

```
{
    "user": [
        {
            "id": 277,
            "name": "visitor",
            "email": "JohnDoe@gmail.com",
            "created_date": "2024-02-19 09:26:42",
            "arrival": "2023-11-08 07:40:27",
            "departure": "2024-07-29 07:40:27",
            "firstname": "Manos",
            "lastname": "Karapiperakis",
            "cleaningprogram": [
                "2023-11-08",
                "2023-11-09"
            ],
            "phone": "6944247486",
            "country": "Greece",
            "role": "visitor"
        }
    ]
}
```

Figure 103 - Get User controller

- createUser: This controller is used to create a new user and the following information should be included in the request body:
    - username
    - password
    - email
    - arrival
    - departure
    - role
    - firstname
    - lastname
    - cleaningprogram
    - phone
    - country

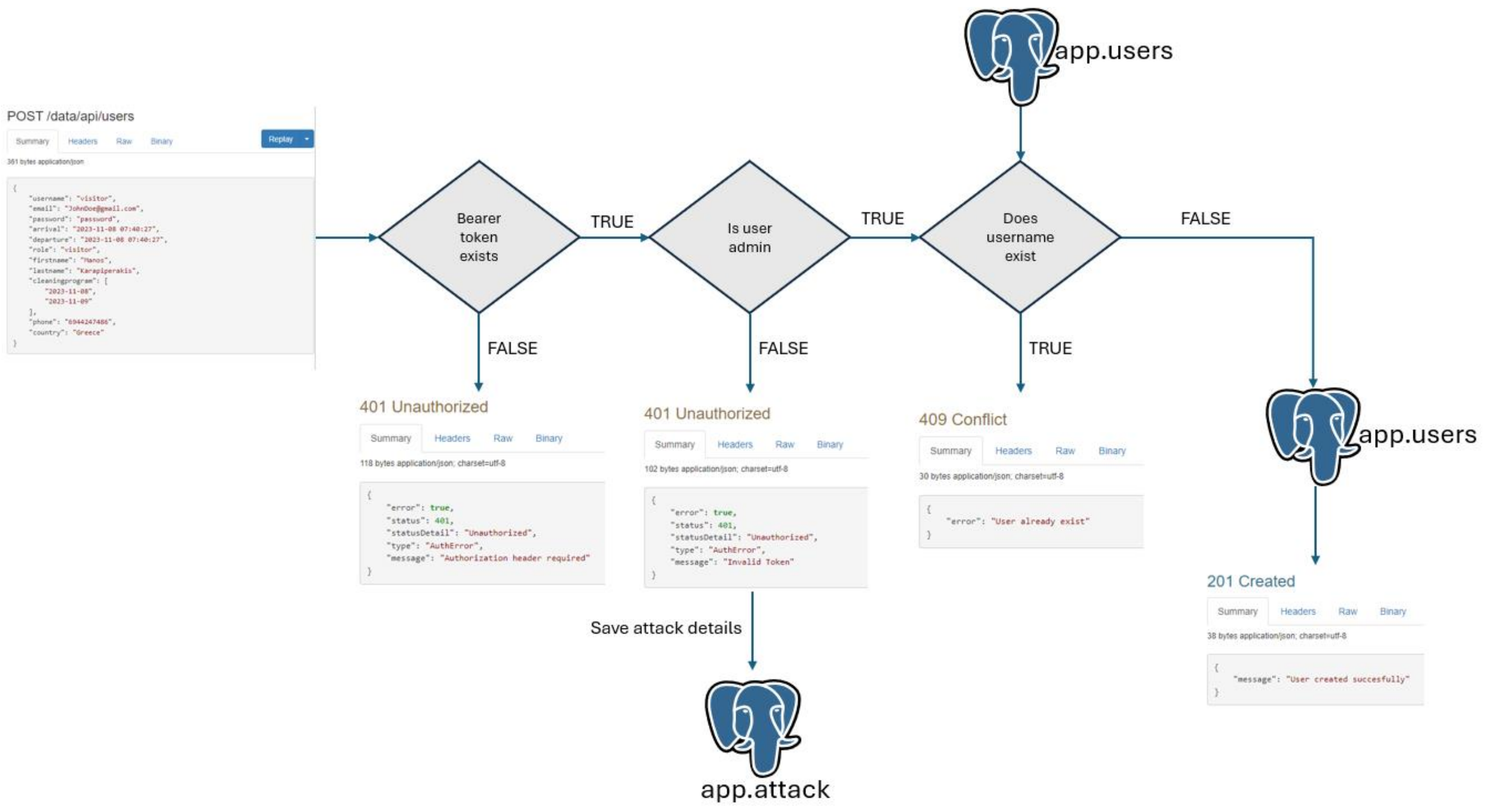
  This functionality is restricted to administrators, so a bearer authentication token is required and the user's role must be "admin" If the role is not "admin", the request is considered a potential attack. In such cases, details like the user's IPv4 address, relevant comments and the current timestamp are logged in the app.attack database table. If the token is valid and the user has admin privileges, the next step is to check if the provided username already exists in the app.users table, as usernames must be unique. If the username is already taken, the process stops, a response with status code 409

Conflict and the message "User already exists" is returned to the client. If the username is unique and meets the requirements, the details from the request body are saved to the app.users table and the response is 201 Created. Most fields are saved as-is, but some require preprocessing before being stored:

- password: The password cannot be stored in plain text. Instead, it is hashed using the *bcrypt* package.
- created_date: The creation date is included in the final INSERT statement and is generated using the built-in Date object.
- cleanningprogram: The cleaning program is stored as an array in the PostgreSQL database, so it must be converted to an array format, as demonstrated in the code snippet below:

```
const cleaningprogramArray = `ARRAY[${cleaningprogram.map(
          (timestamp) => `'${timestamp}'`
        )}]`;
```

The previously described process is illustrated in the following diagram with additional details:



POST /data/api/users

361 bytes application/json

```
{
    "username": "visitor",
    "email": "JohnDoe@gmail.com",
    "password": "password",
    "arrival": "2023-11-08 07:40:27",
    "departure": "2023-11-08 07:40:27",
    "role": "visitor",
    "firstname": "Manos",
    "lastname": "Karapiperakis",
    "cleaningprogram": [
        "2023-11-08",
        "2023-11-09"
    ],
    "phone": "6944247486",
    "country": "Greece"
}
```

401 Unauthorized

118 bytes application/json; charset=utf-8

```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

401 Unauthorized

102 bytes application/json; charset=utf-8

```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Invalid Token"
}
```

409 Conflict

30 bytes application/json; charset=utf-8

```
{
    "error": "User already exist"
}
```

201 Created

38 bytes application/json; charset=utf-8

```
{
    "message": "User created succesfully"
}
```

Figure 104 - Create User Controller

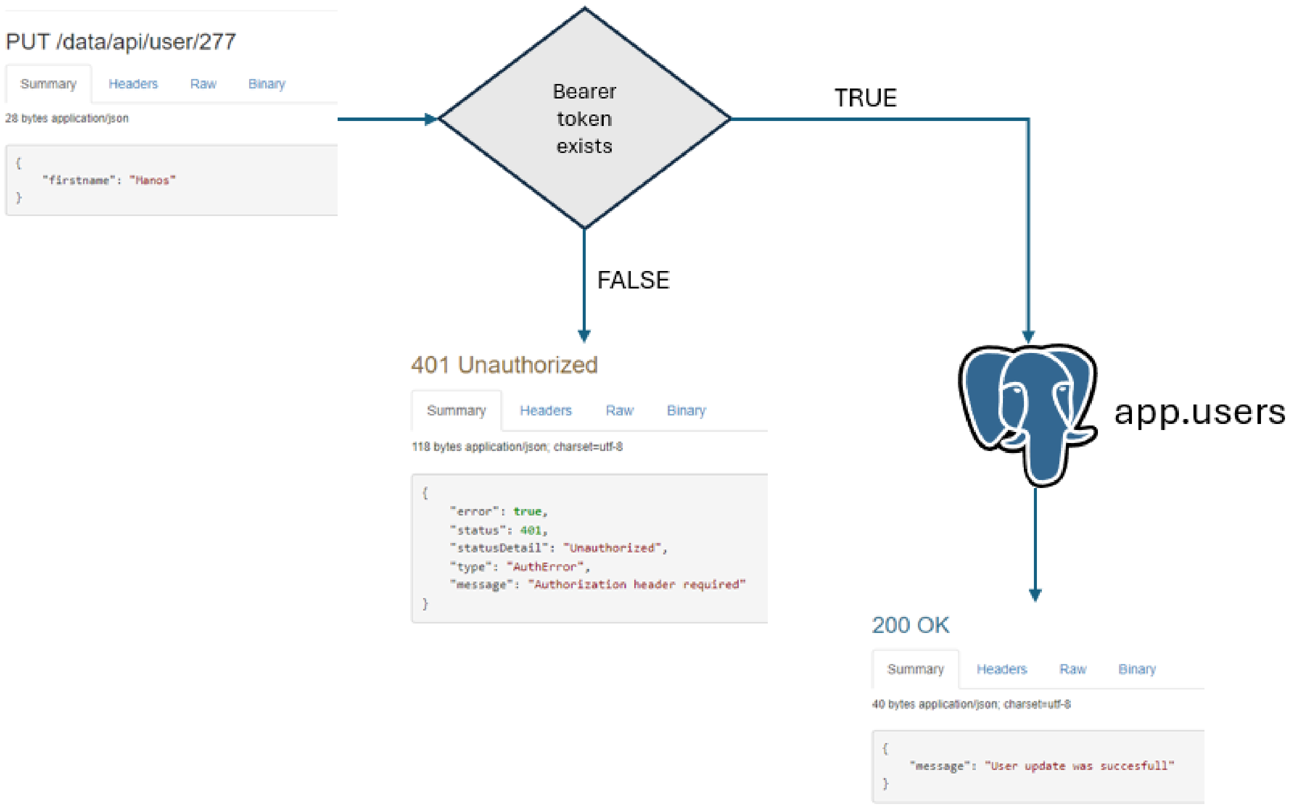
- updateUser: This controller handles the updating of user information. Within the application, there are two ways this can be executed: Either by users (guests) updating one or more front-end visible fields (user information) or by an admin updating both read-only and front-end visible fields (stay information and user information). To accommodate both scenarios, the UPDATE statement should be dynamic, allowing updates to any combination of fields, whether none, one, or all:

```
let userId = req.params.id;
const updateFields = [
    "email",
    "arrival",
    "departure",
    "cleaningprogram",
    "role",
    "firstname",
    "lastname",
    "phone",
    "country",
    "device",
  ];

  const setClause = updateFields
  .filter((field) => req.body[field] !== undefined)
  .map((field) => {
    if (field === "cleaningprogram") {
      return `${field} = '{"${req.body[field].join('","')}"}'`;
    } else {
      return `${field} = '${req.body[field]}'`;
    }
  });
```

Additionally, the user's status (active or inactive) should be updated correctly whenever the departure date is modified:

```
const today = new Date();
today.setHours(0, 0, 0, 0); // Set time to midnight for comparison
if (req.body.departure) {
  const departureDate = new Date(req.body.departure);
  if (departureDate > today) {
    setClause.push("is_active = true");
  } else {
    setClause.push("is_active = false");
  }
}

const query = `
  UPDATE app.users
  SET ${setClause.join(", ")}
  WHERE id = ${userId}
`;
```

Although a bearer authentication token is required, no additional role-based checks are needed, as this functionality is accessible to both guests and admins. Upon a successful update, the client receives a response with status code 200 and the message "User update was successful":



PUT /data/api/user/277

28 bytes application/json

```
{
    "firstname": "Manos"
}
```

401 Unauthorized

118 bytes application/json; charset=utf-8

```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

200 OK

40 bytes application/json; charset=utf-8

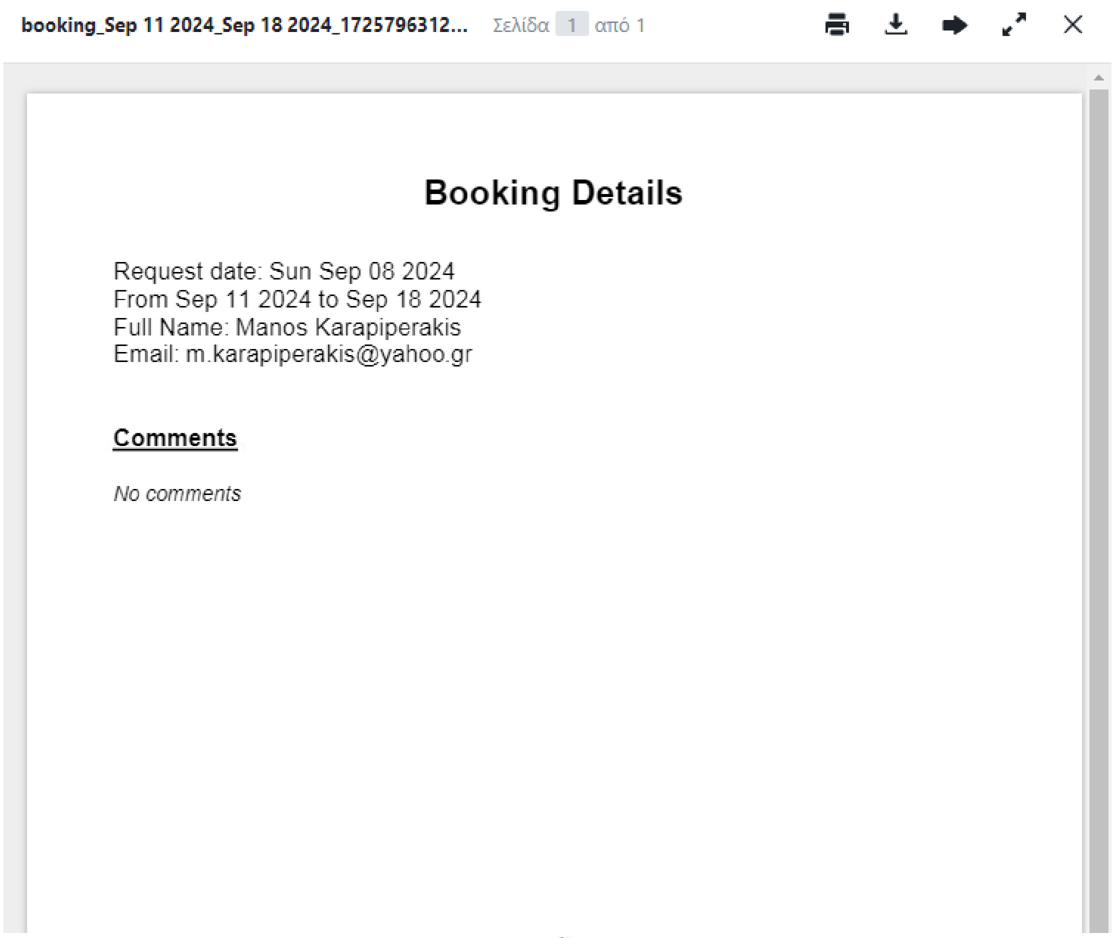
```
{
    "message": "User update was succesfull"
}
```

Figure 105 - Update User Controller

- Booking: This controller handles the creation of a new booking request. Since this functionality is available to non-registered users, a bearer token is not required. However, to protect the server, a basic token is necessary for this request. If no basic token is provided, or if the token is invalid, the server responds with a 401 status code and the error message "Authorization Header Required". When the token is valid, the following three steps are executed:
    - A dynamic pdf file is generated, including information given by the client. For this purpose, the *pdfkit*, *fs* and *path* packages are used as it shown in the following code snippet:

```
const PDFDocument = require("pdfkit");
const fs = require("fs");
const path = require("path");

const { startDate, endDate, fullName, email, comments, visitors } =
req.body;
let currentDate = new Date().toDateString();
const currentTimestamp = new Date().getTime();
const pdfFileName = `booking_${startDate}_${endDate}_${cur-
rentTimestamp}.pdf`;
const pdfPath = path.join(__dirname, "pdf_to_send", pdfFileName);
const pdfStream = fs.createWriteStream(pdfPath);

// Create a new PDF document
const doc = new PDFDocument({
  margin: 50,
});
// Add content to the PDF document with styling
doc
  .font("Helvetica-Bold")
  .fontSize(20)
  .text("Booking Details", { align: "center" });
doc.moveDown();

doc.font("Helvetica").fontSize(14);

doc.text(`Request date: ${currentDate}`);
doc.text(`From ${startDate} to ${endDate}`);
doc.text(`Full Name: ${fullName}`);
doc.text(`Email: ${email}`);

doc.moveDown(2);
doc.font("Helvetica-Bold").text("Comments", { underline: true });

doc.moveDown();
if (comments) doc.font("Helvetica-Oblique").fontSize(12).text(com-
ments);
else doc.font("Helvetica-Oblique").fontSize(12).text("No comments");

// Finalize the PDF document and end the response stream
doc.end();
```

As a result of the above code snippet, a new pdf will be generated and saved inside a folder named pdf_to_send:


booking_Sep 11 2024_Sep 18 2024_1725796312... Σελίδα 1 από 1

Booking Details

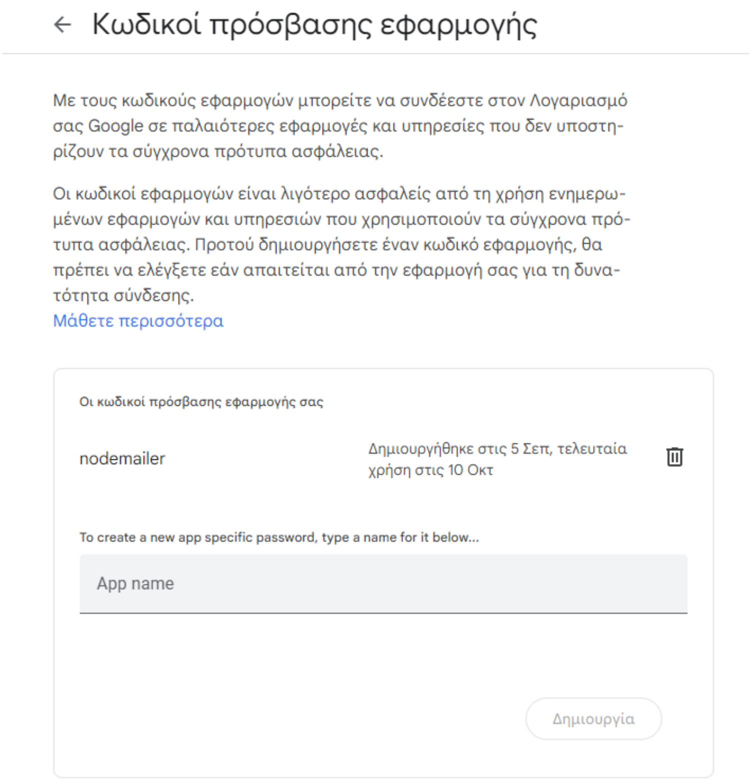
Request date: Sun Sep 08 2024
From Sep 11 2024 to Sep 18 2024
Full Name: Manos Karapiperakis
Email: m.karapiperakis@yahoo.gr

Comments

No comments

Figure 106 - Generated PDF

- An email notification should be sent to the host, informing them about the new booking request. This email must include details of the booking along with the previously generated PDF as an attachment. To achieve this, the *nodemailer* package was used. The email can have one sender and one or more recipients, managed via two environment variables: EMAIL_SENDER and EMAIL_RECEIVER. A Gmail address was chosen as the sender, while the recipients can use any email provider. For security purposes, credentials are required for the sender's email, ensuring that unauthorized users cannot send emails on someone else’s behalf. An application-specific password was created in Google account settings, which is only visible when first generated. This password is stored in the .env file under the variable PASSWORD_SENDER.


Κωδικοί πρόσβασης εφαρμογής

Με τους κωδικούς εφαρμογών μπορείτε να συνδέεστε στον Λογαριασμό σας Google σε παλαιότερες εφαρμογές και υπηρεσίες που δεν υποστηρίζουν τα σύγχρονα πρότυπα ασφάλειας.

Οι κωδικοί εφαρμογών είναι λιγότερο ασφαλείς από τη χρήση ενημερωμένων εφαρμογών και υπηρεσιών που χρησιμοποιούν τα σύγχρονα πρότυπα ασφάλειας. Προτού δημιουργήσετε έναν κωδικό εφαρμογής, θα πρέπει να ελέγξετε εάν απαιτείται από την εφαρμογή σας για τη δυνατότητα σύνδεσης.
Μάθετε περισσότερα

Οι κωδικοί πρόσβασης εφαρμογής σας

nodemailer — Δημιουργήθηκε στις 5 Σεπ, τελευταία χρήση στις 10 Οκτ

To create a new app specific password, type a name for it below...

App name

Δημιουργία

Figure 107 - Gmail App Password

The next step is to create a new transport instance using the provided method named "createTransport" from the *nodemailer* library as shown in the code snippet below:

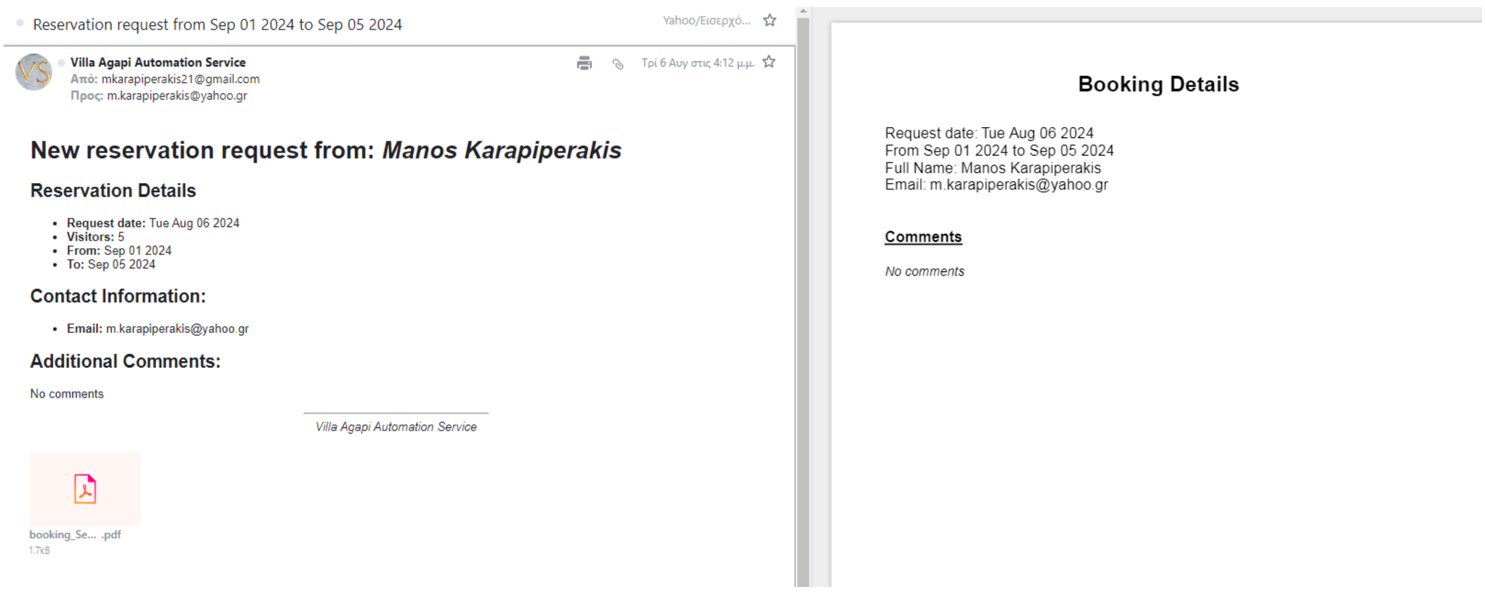
```javascript
let transporter = nodemailer.createTransport({
    host: "smtp.gmail.com", // SMTP server address
    port: 465, // Port for SMTP (usually 465)
    secure: true, // Usually true if connecting to port 465
    auth: {
      user: process.env.EMAIL_SENDER,
      pass: process.env.PASSWORD_SENDER,
    },
  });
```

After that, the email is sent using Nodemailer's "sendMail" method, which includes the following options:

1. from: Sender
2. to: Receiver
3. subject: Email Subject
4. html: Dynamic content
5. attachments: The PDF file to be attached to the email

```
let info = await transporter.sendMail({

    from: `"Villa Agapi Automation Service" <${pro-
cess.env.EMAIL_SENDER}>`,
    to: process.env.EMAIL_RECEIVER,
    subject: `Reservation request from ${startDate} to ${endDate}`,
    html: `
    <h1>New reservation request from: <i>${fullName}</i></h1>
    <h2>Reservation Details</h2>
    <ul>
        <li><strong>Request date: </strong> ${currentDate}</li>
        <li><strong>Visitors: </strong> ${visitors}</li>
        <li><strong>From:</strong> ${startDate}</li>
        <li><strong>To:</strong> ${endDate}</li>
    </ul>
    <h2>Contact Information:</h2>
    <ul>
        <li><strong>Email:</strong> ${email}</li>
    </ul>
    <h2>Additional Comments:</h2>
    <p>${!!comments ? comments : "No comments"}</p>
    <hr style = "width: 200px">
    <div style="text-align: center;">
        <i>Villa Agapi Automation Service</i>
    </div>
    `,
    attachments: [
      {
        filename: pdfFileName,
        path: pdfPath,
      },
    ],
  });
```

The final outcome of the email process is displayed in the image below:



Reservation request from Sep 01 2024 to Sep 05 2024

Yahoo/Εισερχό...

Villa Agapi Automation Service
Από: mkarapiperakis21@gmail.com
Προς: m.karapiperakis@yahoo.gr

Τρί 6 Αυγ στις 4:12 μ.μ.

New reservation request from: *Manos Karapiperakis*

Reservation Details

- **Request date:** Tue Aug 06 2024
- **Visitors:** 5
- **From:** Sep 01 2024
- **To:** Sep 05 2024

Contact Information:

- **Email:** m.karapiperakis@yahoo.gr

Additional Comments:

No comments

Villa Agapi Automation Service

booking_Se... .pdf
1.7kB

Booking Details

Request date: Tue Aug 06 2024
From Sep 01 2024 to Sep 05 2024
Full Name: Manos Karapiperakis
Email: m.karapiperakis@yahoo.gr

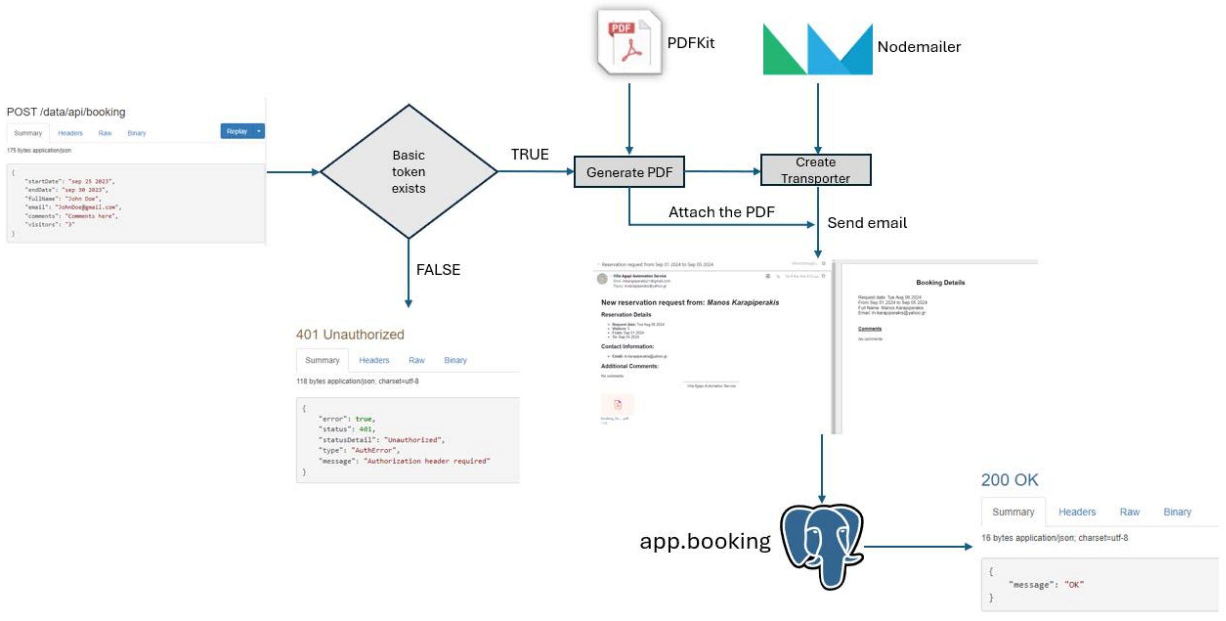
Comments

No comments

Figure 108 - Booking Request Email

- The final step of the booking controller, is to save the booking details in the respective database table named app.booking

The whole process is displayed in the below diagram:



POST /data/api/booking

175 bytes application/json

```
{
    "startDate": "sep 25 2023",
    "endDate": "sep 30 2023",
    "fullName": "John Doe",
    "email": "JohnDoe@gmail.com",
    "comments": "Comments here",
    "visitors": "3"
}
```

401 Unauthorized

118 bytes application/json; charset=utf-8

```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

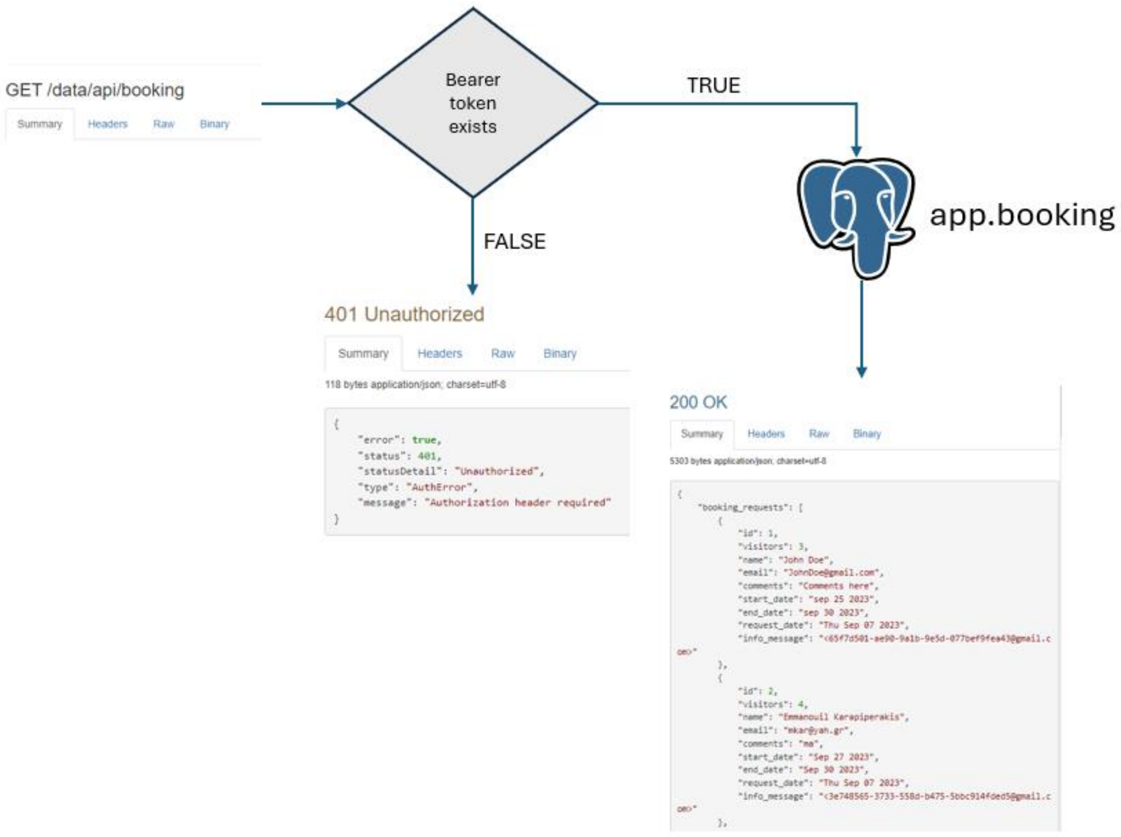
200 OK

16 bytes application/json; charset=utf-8

```
{
    "message": "OK"
}
```

Figure 109 - Create Booking Controller

- getBookings: This controller is responsible for returning all the bookings requests from the table app.booking. Since this functionality is available only through the admin dashboard, a bearer token is required. If the JWT is missing from the request headers, a response with status code 401 and the error message "Authorization Header Required" is returned. Since this is a safe method that does not modify the database content, no additional security measures are required. The process is illustrated in the following diagram:



118 bytes application/json; charset=utf-8

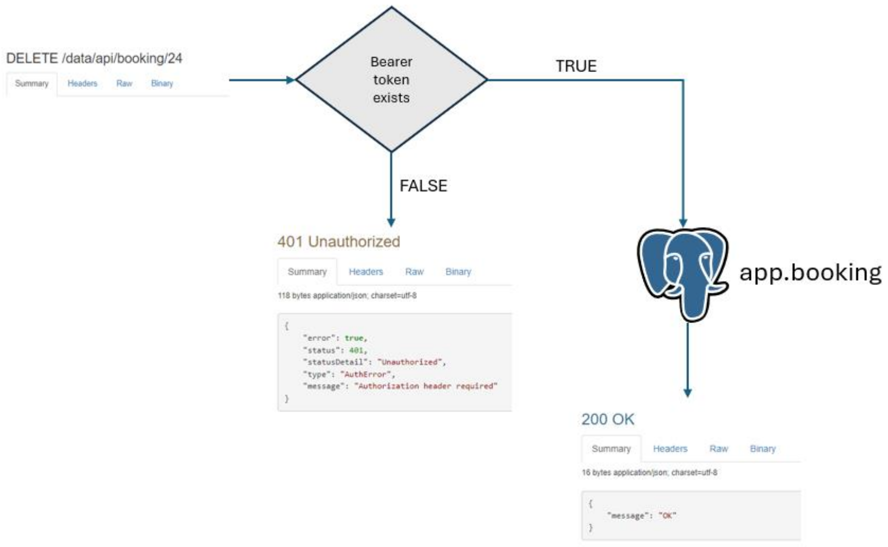
```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

5303 bytes application/json; charset=utf-8

```
{
    "booking_requests": [
        {
            "id": 1,
            "visitors": 3,
            "name": "John Doe",
            "email": "JohnDoe@gmail.com",
            "comments": "Comments here",
            "start_date": "sep 25 2023",
            "end_date": "sep 30 2023",
            "request_date": "Thu Sep 07 2023",
            "info_message": "<65f7d501-ae90-9a1b-9e5d-077bef9fea43@gmail.c
om>"
        },
        {
            "id": 2,
            "visitors": 4,
            "name": "Emmanouil Karapiperakis",
            "email": "mkar@yah.gr",
            "comments": "ma",
            "start_date": "Sep 27 2023",
            "end_date": "Sep 30 2023",
            "request_date": "Thu Sep 07 2023",
            "info_message": "<3e748565-3733-558d-b475-5bbc914fded5@gmail.c
om>"
        },
```

Figure 110 - Get Bookings Controller

- deleteBooking: This controller is responsible for deleting a specific record from the app.booking table based on the provided ID in the URL parameters. Since this functionality is available only through the admin dashboard, a bearer token is required. If the JWT is missing from the request headers, a response with status code 401 and the error message "Authorization Header Required" is returned. The process is illustrated in the following diagram:



118 bytes application/json; charset=utf-8

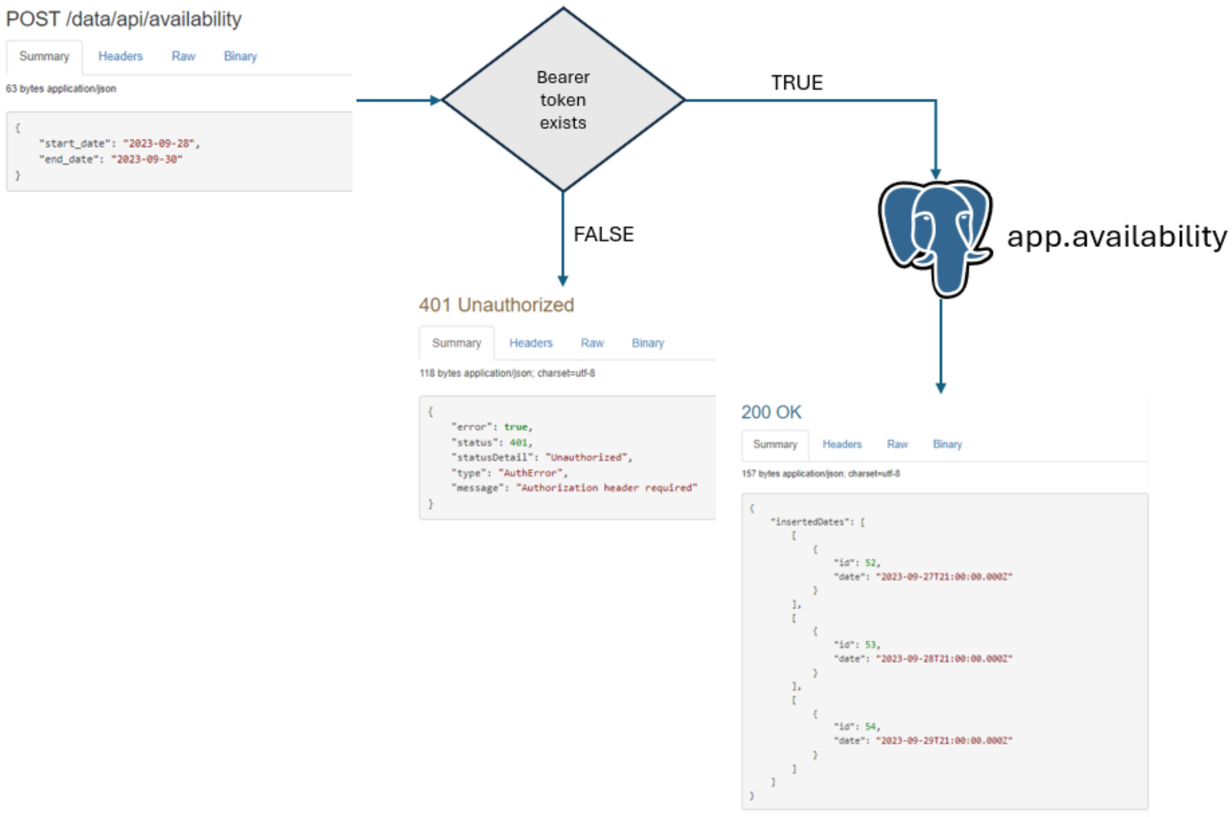
```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

16 bytes application/json; charset=utf-8

```
{
    "message": "OK"
}
```

Figure 111 - Delete Booking Controller

- addAvailability: This controller is responsible for updating the availability by adding dates during which the house is not available, by modifying the app.availability table. The request body includes the arrival date and departure date of a booking, covering all the dates in between. Since this functionality is available only through the admin dashboard, a bearer token is required. If the JWT is missing from the request headers, a response with status code 401 and the error message "Authorization Header Required" is returned. The process is illustrated in the following diagram:


POST /data/api/availability
Summary
Headers
Raw
Binary
63 bytes application/json
Bearer token exists
TRUE
FALSE
app.availability
401 Unauthorized
118 bytes application/json; charset=utf-8
200 OK
157 bytes application/json; charset=utf-8

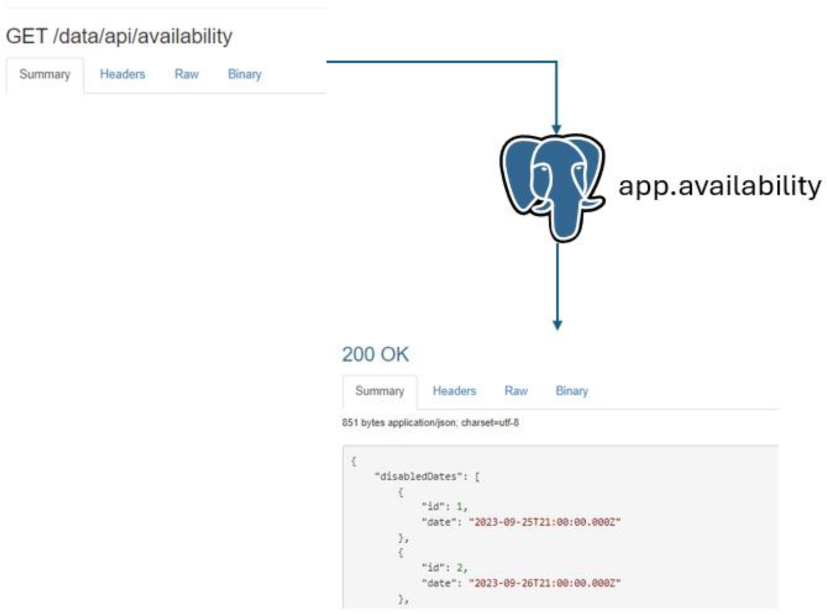
```
{
    "start_date": "2023-09-28",
    "end_date": "2023-09-30"
}
```

```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

```
{
    "insertedDates": [
        [
            {
                "id": 52,
                "date": "2023-09-27T21:00:00.000Z"
            }
        ],
        [
            {
                "id": 53,
                "date": "2023-09-28T21:00:00.000Z"
            }
        ],
        [
            {
                "id": 54,
                "date": "2023-09-29T21:00:00.000Z"
            }
        ]
    ]
}
```

Figure 112 - Add Availability Controller

- getAvailability: This controller is responsible for retrieving house availability from the app.availability table. This request is safe because it does not modify any resources and is accessible to unregistered users. Additionally, since it can be used to promote the property elsewhere, no security measures are implemented. The process is illustrated in the following diagram:

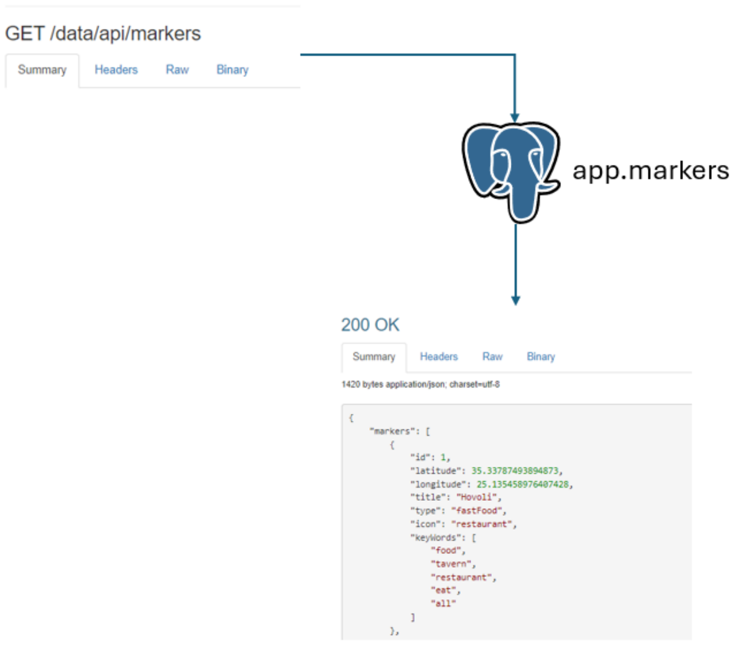


```
851 bytes application/json; charset=utf-8
```

```
{
    "disabledDates": [
        {
            "id": 1,
            "date": "2023-09-25T21:00:00.000Z"
        },
        {
            "id": 2,
            "date": "2023-09-26T21:00:00.000Z"
        },
```

Figure 113 - Get Availability Controller

- getMarkers: This controller fetches all available markers from the app.markers database table, to be displayed on the dynamic maps within the mobile application. The request is safe, as it doesn't alter any resources and is accessible to unregistered users. Moreover, since the markers can be shared externally, no security measures have been implemented. The process is outlined in the diagram below:

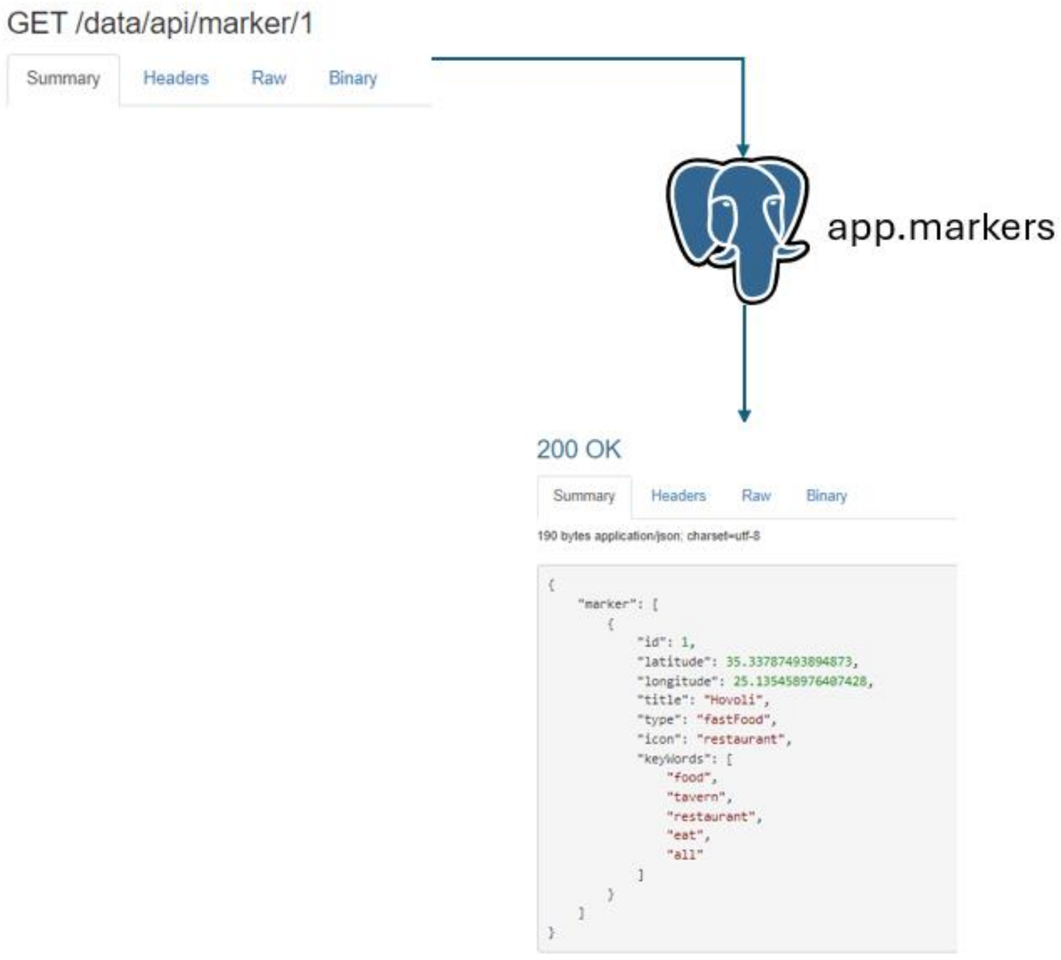


```
1420 bytes application/json; charset=utf-8
```

```
{
    "markers": [
        {
            "id": 1,
            "latitude": 35.33787493894873,
            "longitude": 25.135458976407428,
            "title": "Hovoli",
            "type": "fastFood",
            "icon": "restaurant",
            "keyWords": [
                "food",
                "tavern",
                "restaurant",
                "eat",
                "all"
            ]
        },
```

Figure 114 - Get Markers Controller

- getMarkerById: This controller retrieves a specific marker by its ID, passed through the URL parameters, from the app.markers database table to display on the dynamic maps within the mobile application. The request is safe, as it does not modify any resources and is accessible to unregistered users. Moreover, since the markers can be shared externally, no security measures have been implemented. The process is outlined in the diagram below:



190 bytes application/json; charset=utf-8

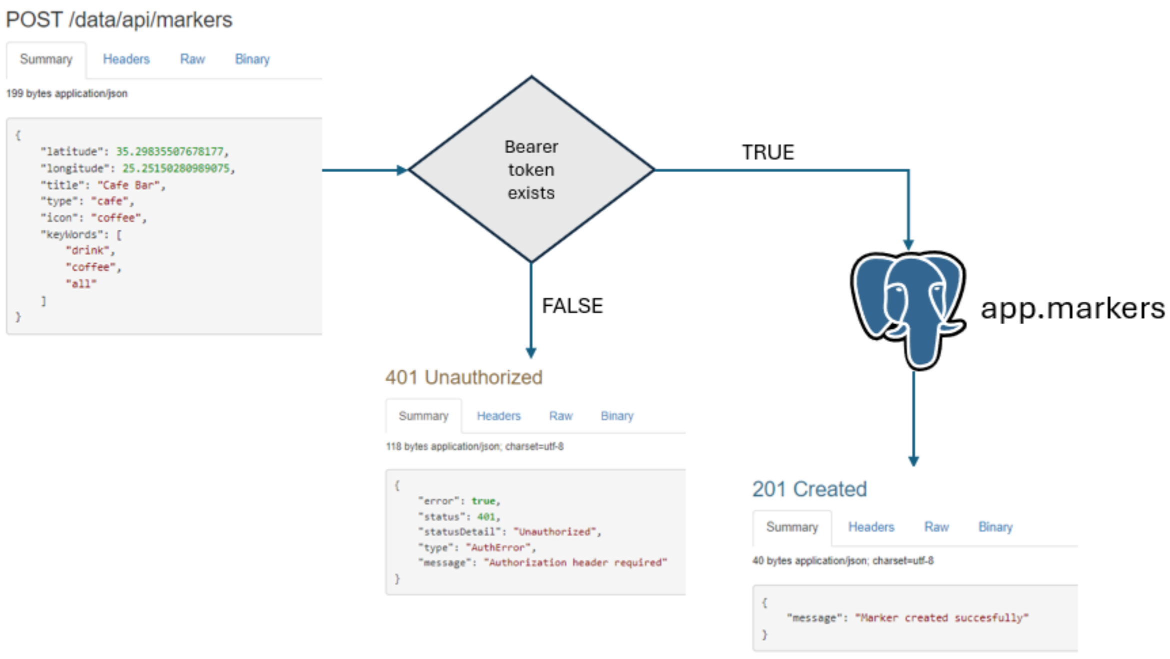
```
{
    "marker": [
        {
            "id": 1,
            "latitude": 35.33787493894873,
            "longitude": 25.135458976407428,
            "title": "Hovoli",
            "type": "fastFood",
            "icon": "restaurant",
            "keyWords": [
                "food",
                "tavern",
                "restaurant",
                "eat",
                "all"
            ]
        }
    ]
}
```

Figure 115 - Get Marker By Id Controller

- createMarker: This controller creates a new marker by saving information sent by the client to the app.markers database table. The information is available in the request body and includes:
    - latitude
    - longitude
    - title
    - type
    - icon
    - keywords

  Since this functionality is available only through the admin dashboard, a bearer token is required. If the JWT is missing from the request headers, a response with status code 401 and the error message

"Authorization Header Required" is returned. The process is outlined in the diagram below:



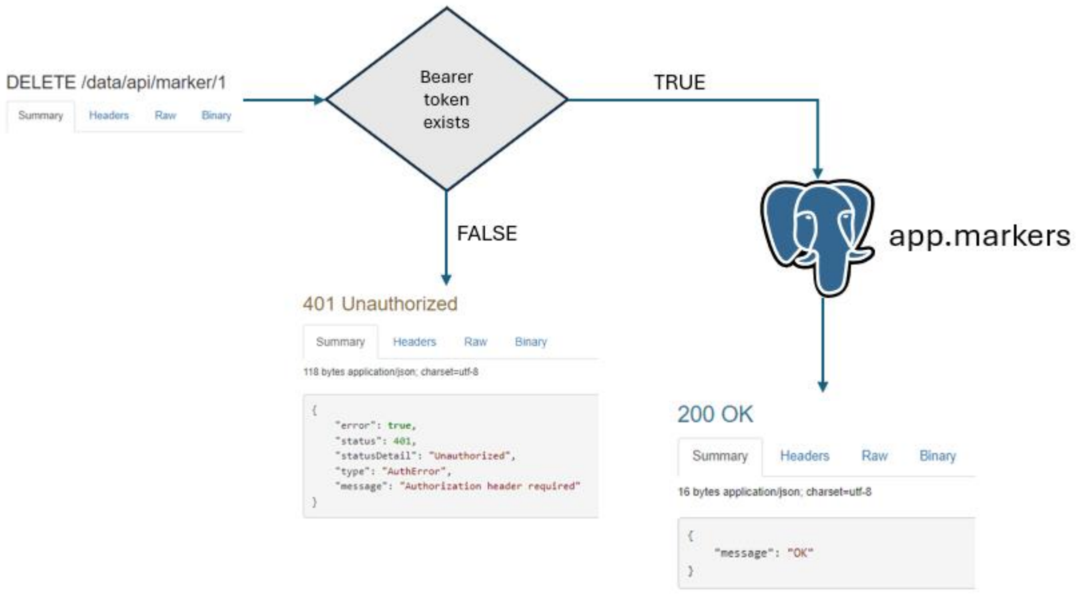
```
{
    "latitude": 35.29835507678177,
    "longitude": 25.25150280989075,
    "title": "Cafe Bar",
    "type": "cafe",
    "icon": "coffee",
    "keyWords": [
        "drink",
        "coffee",
        "all"
    ]
}
```

```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

```
{
    "message": "Marker created succesfully"
}
```

Figure 116 - Create Marker Controller

- deleteMarker: This controller deletes a specific marker by its ID, passed through the URL parameters, from the app.markers database table. Since this functionality is available only through the admin dashboard, a bearer token is required. If the JWT is missing from the request headers, a response with status code 401 and the error message "Authorization Header Required" is returned. The process is outlined in the diagram below:


DELETE /data/api/marker/1
Summary
Headers
Raw
Binary
Bearer token exists
TRUE
FALSE
app.markers
401 Unauthorized
118 bytes application/json; charset=utf-8
200 OK
16 bytes application/json; charset=utf-8

```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

```
{
    "message": "OK"
}
```

Figure 117 - Delete Marker Controller

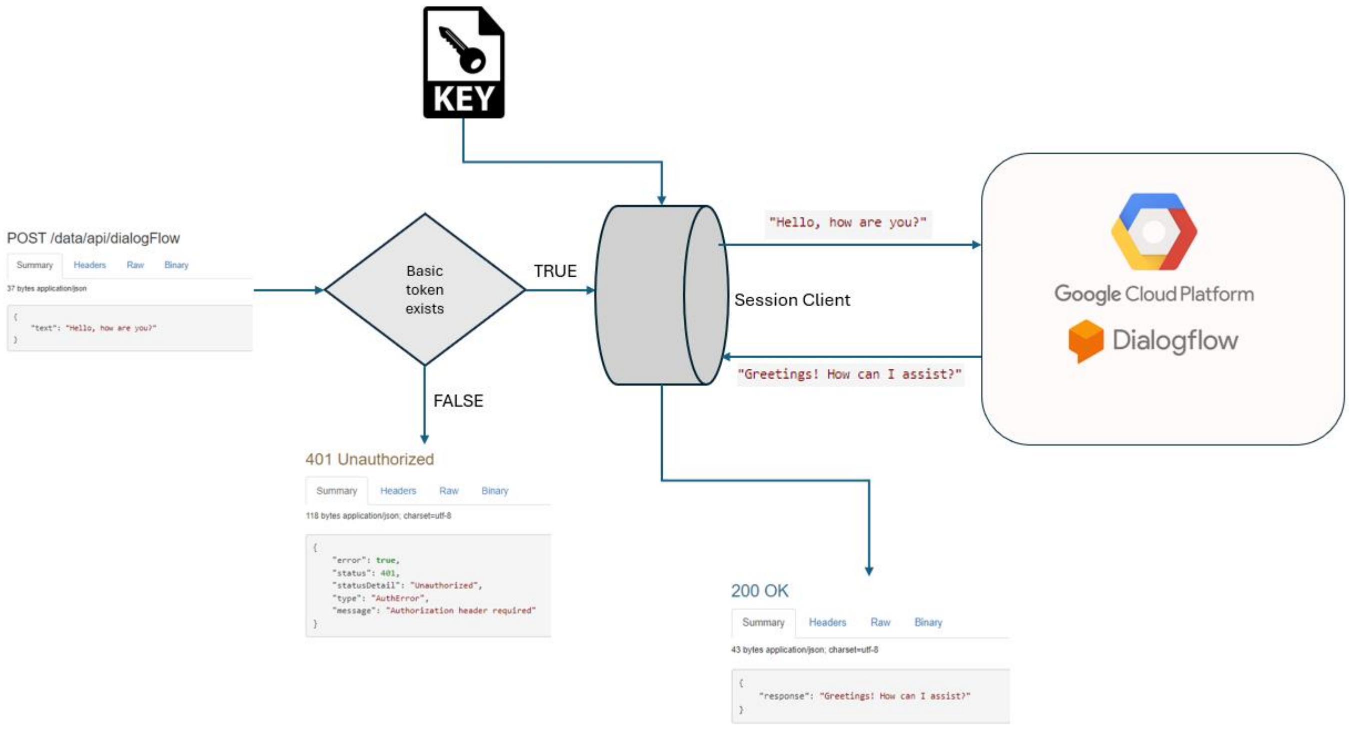
- getBotResponse: This controller facilitates communication with the Google Dialogflow Agent. The request body contains the message the client wants to send to Dialogflow and the response body returns the generated text (response) from Dialogflow. The first step is to import the dialogflow library and then create a new instance of a SessionClient using a .json file that has been exctracted from GCP as mentioned in the section *5.3.3 Export Private Key:*

```
const { SessionsClient } = require("dialogflow");
const path = require("path");

const sessionClient = new SessionsClient({
  keyFilename: path.join(__dirname, "auth.json"),
});
```

Next, a new session path should be created using the previously initialized session client. After that, a request object can be constructed, containing the text to be transmitted. By invoking the detectIntent method on the session client, the response will be returned to the server. The process described is illustrated in the following code snippet:

```
 const sessionPath = sessionClient.sessionPath("auth", "1");

  const request = {
    session: sessionPath,
    queryInput: {
      text: {
        text: req.body.text,
        languageCode: "en-US",
      },
    },
  };

  try {
    const response = await sessionClient.detectIntent(request);

    res.status(200).json({
      response: response[0].queryResult.fulfillmentText,
    });
  } catch (error) {
    console.error("Error while processing Dialogflow request:", er-
ror);
    res.status(500).json({
      error: "Internal Server Error",
    });
  }
```

Since this functionality is available for registered users (guests) but can also be expanded outside of the application, a Basic token is required. If the Basic token is missing from the request headers, a response with status code 401 and the error message "Authorization Header Required" is returned. The process is outlined in the diagram below:


KEY
POST /data/api/dialogFlow
Summary Headers Raw Binary
Basic token exists
TRUE
FALSE
Session Client
"Hello, how are you?"
"Greetings! How can I assist?"
Google Cloud Platform
Dialogflow
401 Unauthorized
Summary Headers Raw Binary
200 OK
Summary Headers Raw Binary

37 bytes application/json

```
{
    "text": "Hello, how are you?"
}
```

118 bytes application/json; charset=utf-8

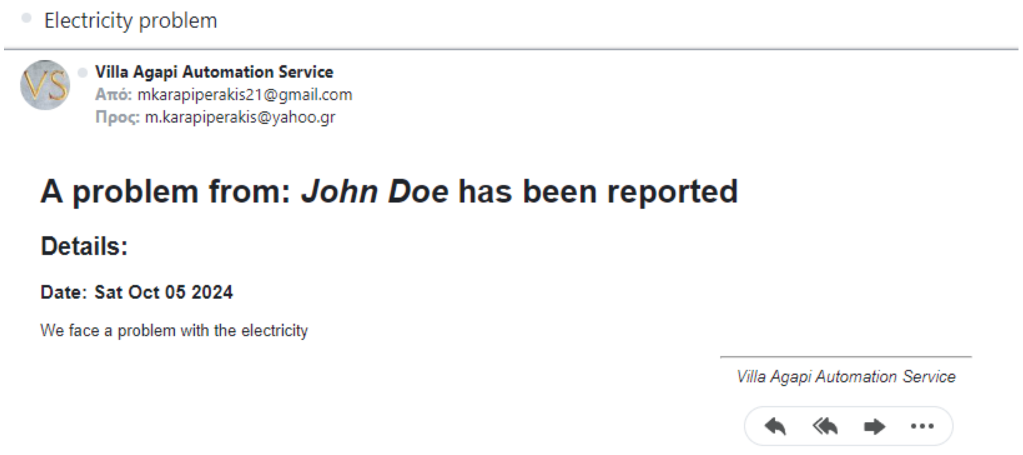
```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

43 bytes application/json; charset=utf-8

```
{
    "response": "Greetings! How can I assist?"
}
```

Figure 118 - DialogFlow Controller

- problem: The final controller provided by the server, is responsible for sending an email to the host regarding a complaint form submitted by the user within the mobile application. The request body includes information required for the email, such as:
    - full name: name of the user
    - subject: subject of the email
    - problem: detailed description of the problem



Electricity problem

Villa Agapi Automation Service
Από: mkarapiperakis21@gmail.com
Προς: m.karapiperakis@yahoo.gr

A problem from: *John Doe* has been reported

Details:

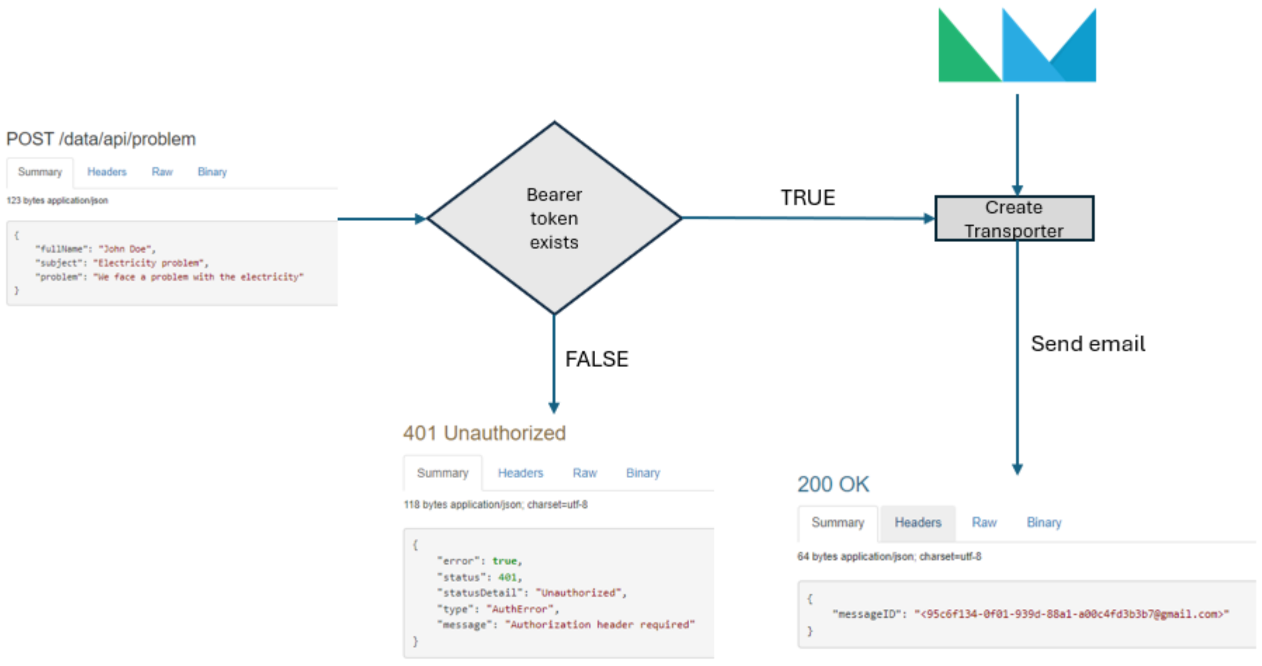
Date: Sat Oct 05 2024

We face a problem with the electricity

Villa Agapi Automation Service

Figure 119 - Problem Email

Since this functionality is applicable for registered users (guests) a Bearer token is required. If the Bearer token is missing from the request headers, a response with status code 401 and the error message "Authorization Header Required" is returned. The process is outlined in the diagram below:



```
{
    "fullName": "John Doe",
    "subject": "Electricity problem",
    "problem": "We face a problem with the electricity"
}
```

```
{
    "error": true,
    "status": 401,
    "statusDetail": "Unauthorized",
    "type": "AuthError",
    "message": "Authorization header required"
}
```

```
{
    "messageID": "<95c6f134-0f01-939d-88a1-a00c4fd3b3b7@gmail.com>"
}
```

Figure 120 - Problem Controller

8.2.9 Create Root File

All the information covered in the previous sections, such as application dependencies, custom errors, authentication middleware, API documentation, routes and controllers, should be consolidated into a single root file. This root file is named app.js and will be executed each time the command npm start is run, serving as the main entry point for the application. First of all, some important libraries are imported: dotenv allows the use of environment variables from a .env file, enabling easy configuration without hardcoding sensitive information. The chalk library is used to style and improve the readability of the command line output by adding colors and formatting. The http module is utilized to create the server instance later in the code. Next, the PORT is set, either from the environment variable process.env.PORT or defaulting to 3010 if not provided. This allows flexibility in specifying the port for the application during different environments (e.g. development, production). For example, Heroku, which will be used later to deploy the application, assigns a dynamic port that is unknown to the user at runtime. The serverInit function is required from another file and will return the application instance when invoked.

Error handling for the application is also set up here. The mainErrorHandler function logs any errors that occur in the application. The code listens for two specific types of errors:

- uncaughtException: errors that are not caught by any try-catch block
- unhandledRejection: errors from promises that are not handled properly.

Both types of errors are directed to mainErrorHandler to ensure they are logged and can be addressed. Once the server initialization (serverInit()) is successful, the code uses the http module to create an HTTP server using the app instance. The server is then set to listen on the specified PORT.
Finally, three log messages are printed to the console using chalk:

- The base URL where the server is running
- The URL for accessing the Swagger UI
- The URL for accessing application metrics

```
require("dotenv").config();
const chalk = require("chalk");
const PORT = process.env.PORT || 3010;
const serverInit = require("./server");

const mainErrorHandler = (err) => console.error(err);
process.on("uncaughtException", mainErrorHandler);
process.on("unhandledRejection", mainErrorHandler);

serverInit().then((app) => {
  const server = require("http").createServer(app);

  server.listen(PORT, () => {
    console.log(
      "Up & running on http://localhost:" + chalk.blue.under-
line.bold(PORT)
    );
    console.log(
      "Swagger UI is available on http://localhost:" +
        chalk.blue.underline.bold(`${PORT}/data/api/doc`)
    );
    console.log(
      "Metrics are available on http://localhost:" +
        chalk.blue.underline.bold(`${PORT}/status`)
    );
  });
});
```

The serverInit function, located in a separate file, is responsible for creating the Express app, as shown below:

```
const express = require("express");
const cors = require("cors");
const swaggerUi = require("swagger-ui-express");
const yaml = require("yamljs");
const drRouter = require("./routes/data-read");
const dwRouter = require("./routes/data-write");
const swaggerDocument = yaml.load("./swagger.yaml");
const { OpenApiValidator } = require("express-openapi-validator");
const {
  RestError,
  AuthError,
  BadRequestError,
  NotFoundError,
} = require("./lib/errors");
const {
  bearerAuthenticator,
  basicAuthenticator,
} = require("./middlewares/authenticator");
const login = require("./middlewares/login-screen");
const morgan = require('morgan')
require("dotenv").config();
```

After importing all the necessary libraries, along with custom functions and classes, the next step is to create a new Express app, as shown in the following code snippet:

```
const app = express()
  .use(require("express-status-monitor")())
  .use(cors())
  .use(express.json({ limit: "20MB" }))
  .use(morgan('combined'))
  .use("/data/api/doc",login, swaggerUi.serve, swaggerUi.setup(swag-
gerDocument)
 .use("/data/api", [drRouter, dwRouter]));
```

Some libraries, such as express-status-monitor and cors, are utilized in the app to enhance its functionality and security. The express-status-monitor library provides real-time monitoring of the server's performance. The cors library is used to enable Cross-Origin Resource Sharing, which allows the server to accept requests from different origins, thereby facilitating secure communication between the client and server. The .use(express.json({ limit: "20MB" })) middleware parses incoming JSON requests and makes the data available in req.body, with a maximum allowed size of 20 megabytes to prevent excessively large requests from straining server resources. The .use(morgan('combined')) middleware sets up logging of HTTP requests in the "combined" format, providing detailed information about each request, including the method, URL, response time and status code, which aids in monitoring and debugging the application.

After that, the line .use("/data/api/doc", login, swaggerUi.serve, swaggerUi.setup(swaggerDocument)); sets up Swagger UI for API documentation, protecting the route "/data/api/doc" with a login middleware to ensure that only authenticated users can access the documentation. The swaggerUi.serve function serves the Swagger UI static files, while swaggerUi.setup(swaggerDocument) initializes the UI with the provided API documentation. Finally the data-read and data-write routes are used in the application, both prefixed with the fixed path “/data/api”.
Additionaly the OpenApiValidator is configured using a specification file located at ./swagger.yaml, which outlines the API's endpoints, request parameters, responses and security definitions. By specifying validateSecurity, the validator is set to check the security of incoming requests against the defined authentication methods. In this case, two authentication handlers are provided: BearerAuth, which utilizes a bearer token for authentication and BasicAuth, which implements basic authentication. The install(app) method integrates the validator into the Express app, ensuring that all incoming requests are validated according to the specifications defined in the swagger.yaml file as illustrated in the following code snippet:

```
await new OpenApiValidator({
    apiSpec: "./swagger.yaml",
    validateSecurity: {
      handlers: {
        BearerAuth: bearerAuthenticator,
        BasicAuth: basicAuthenticator
      },
    },
  }).install(app);
```

8.3 Socket.io Server

To implement the Live Chat feature in the application, a new Node.js server was set up, as outlined in section "8.2 Create Node.js Server". This time, the version 16.20.1 of Node.js was used. A new root file named index.js was created and in the package.json file, the scripts object was updated with the following command: "start": "node src/index.js". This ensures that whenever the start command is executed, the server will be initiated automatically.

8.3.1 Application Dependencies

The following dependencies were installed in this node application:

- cors: Enables Cross-Origin Resource Sharing for secure API access.
- dotenv: Loads environment variables from a .env file into process.env.
- @socket.io/admin-ui: A socket.io admin dashboard, allowing for monitor the live chat.
- express: It provides all the tools needed for handling HTTP requests.
- http: This package was used to create the socket.io server.

- nodemailer: Sends emails from Node.js applications via SMTP or other transport methods
- socket.io: The socket.io server side library

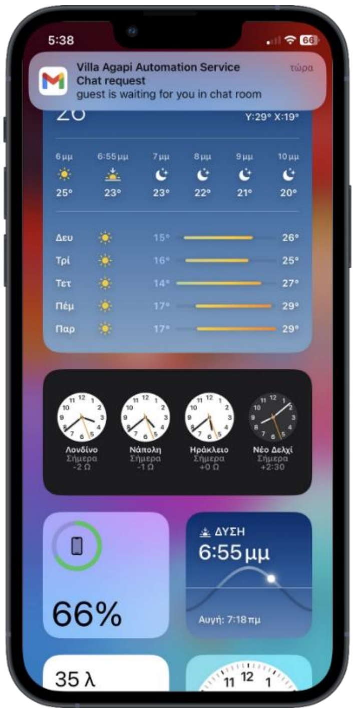
8.3.2 Environment Variables

To protect the application's codebase, sensitive information such as API keys is stored securely within .env files. Specifically, the following values are included:

- EMAIL_RECEIVER: The designated recipient for emails, relevant to a specific business process.
- EMAIL_SENDER: The sender's email address, part of a business process implementation.
- PASSWORD_SENDER: A unique password for the email sender, enabling email-sending functionality.
- INSTRUMENT_USERNAME: Part of credentials needed to access the socket.io admin dashboard.
- INSTRUMENT_PASSWORD: Part of credentials needed to access the socket.io admin dashboard.
- NOTIFY_ADMIN: A Boolean flag was introduced to control whether the admin user receives a notification when someone enters a chat room.

8.3.3 Middlewares

A new folder named middlewares was created, containing a JavaScript file called notify-admin.js. This file is responsible for notifying the administrator when a user joins the chatroom. To achieve this, the *nodemailer* package was utilized, sending an email notification to the designated EMAIL_RECEIVER.



Villa Agapi Automation Service
Chat request
guest is waiting for you in chat room
τώρα

Figure 121 - Email Notification

8.3.4 Initiate Server

To initiate the server, a new instance of the Server was created using the method provided by the socket.io package, as demonstrated in the code snippet below:

```
const { Server } = require("socket.io");

const io = new Server(server, {
  cors: {
    cors,
    origin: [
      "https://admin.socket.io",
      "https://sockets-48f4f0779b20.herokuapp.com",
    ],
    credentials: true,
  },
});
```

The CORS configuration in the above code, allows the WebSocket server to handle requests from specific origins. It includes an origin option that permits connections from two URLs:

- https://admin.socket.io: the socket.io admin panel.
- https://sockets-48f4f0779b20.herokuapp.com: The server hosted on Heroku.

The credentials setting enables the server to accept cookies and authentication headers with WebSocket requests, ensuring secure cross-origin communication.

8.3.5 Events

The server provides the following event listeners, enabling it to respond to incoming events from connected clients:

- join-room: This event listener is responsible for detecting when a user joins a chat room. It receives two arguments: the room name and the username. First, it checks if the environment variable NOTIFY_ADMIN is set to true. If so, an email notification is sent to the administrator using the notify-admin middleware. Next, the socket.join method is called to add the user to the specified room, passing the room name as the argument. Finally, an event emitter is triggered, to notify the client side that a user has joined the room by definining the event name in the first argument and the username in the second. The process is illustrated in the following code snippet:

```
const notifyAdmin = require("./middlewares/notify-admin");
socket.on("join-room", (room, username) => {
    if (process.env.NOTIFY_ADMIN) notifyAdmin(username);

    socket.join(room);
    socket.to(room).emit("joined", username);
  });
```

- message: This event listener handles messages sent from the client, accepting two arguments:
    - message: the message itself
    - room: the room name

  If no room is specified, the message is broadcast to all connected clients using the built-in *socket.broadcast* emitter, which sends the message to everyone except the sender. However, if a room is defined, the message is sent exclusively to that room using the socket.to(<room>) emitter, ensuring only users in the specified room receive it. The process is illustrated in the following code snippet:

```
socket.on("message", (message, room) => {
    if (room == "") {
      socket.broadcast.emit("receive-message", message);
    } else {
      socket.to(room).emit("receive-message", message);
    }
  });
```

- leave-room: This event listener manages the scenario when a user leaves a room they were previously connected to, ensuring that the other users in the room are informed. When triggered, it captures the room name and the username of the user who is leaving. The listener then emits an event to notify all other clients in the specified room:

```
socket.on("leave-room", (room, username) => {
    socket.to(room).emit("disconnected", username);
  });
```

- isTyping: This event listener captures the scenario when a user is actively typing a message in the chat interface. When a user begins typing, this event is triggered and sends an indication to other users in the same chat room, alerting them that someone is currently typing:

```
socket.on("isTyping", (username, room) => {
    socket.to(room).emit("isTyping", username);
  });
```

- isNotTyping: This event listener captures the scenario when a user stops typing a message in the chat interface. When a user stops typing, this event is triggered and sends an indication to other users in the same chat room:

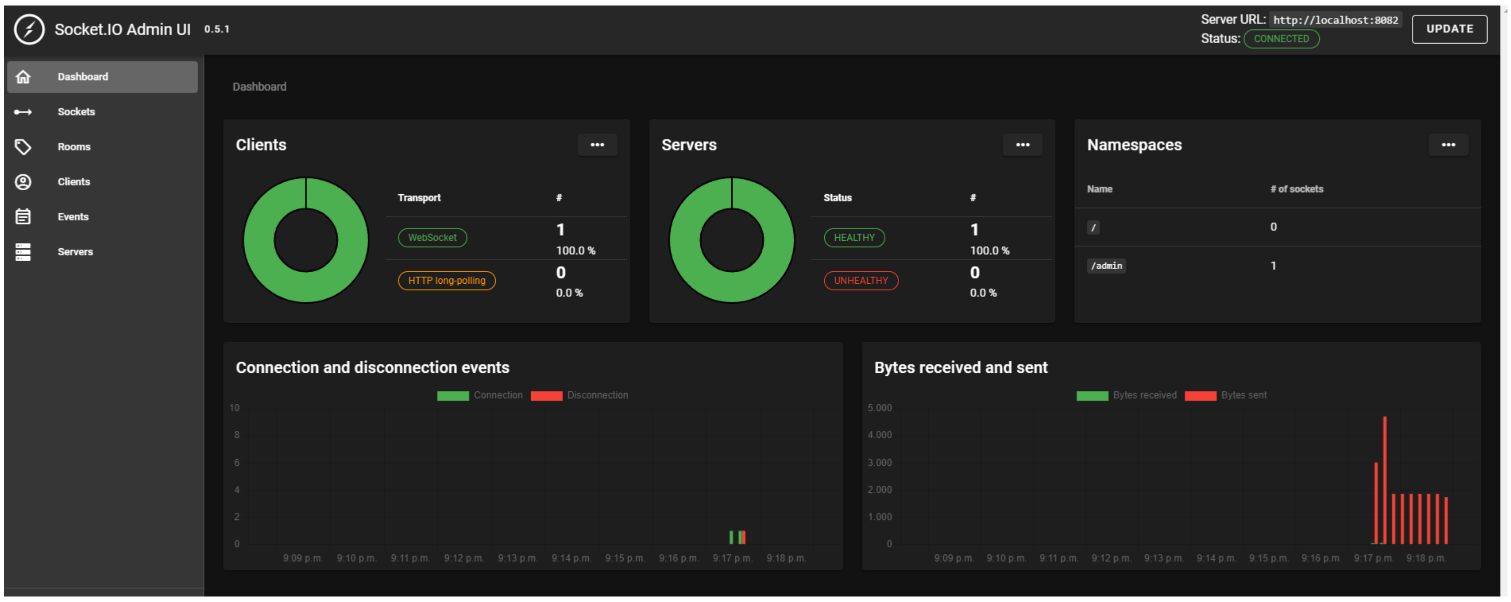
```
socket.on("isNotTyping", (username, room) => {
    socket.to(room).emit("isNotTyping", username);
  });
```

8.3.6 Admin Panel

The admin panel provides a convenient way to monitor socket connection details. It can be seamlessly integrated into a Node.js application, as demonstrated in the following code snippet:

```javascript
app.get("/", (req, res) => {
  res.redirect("https://admin.socket.io/#/");
});

instrument(io, {
  auth: {
    type: "basic",
    username: process.env.INSTRUMENT_USERNAME,
    password: process.env.INSTRUMENT_PASSWORD,
  },
  mode: "development",
});
```

The first step is to create a route that redirects to the official admin panel URL. From this page, the user can log in using the credentials provided in the instrument function. After successfully logging in, the user can access detailed socket information, including both client and server data:



Clients

Transport	#
WebSocket	1 100.0 %
HTTP long-polling	0 0.0 %

Servers

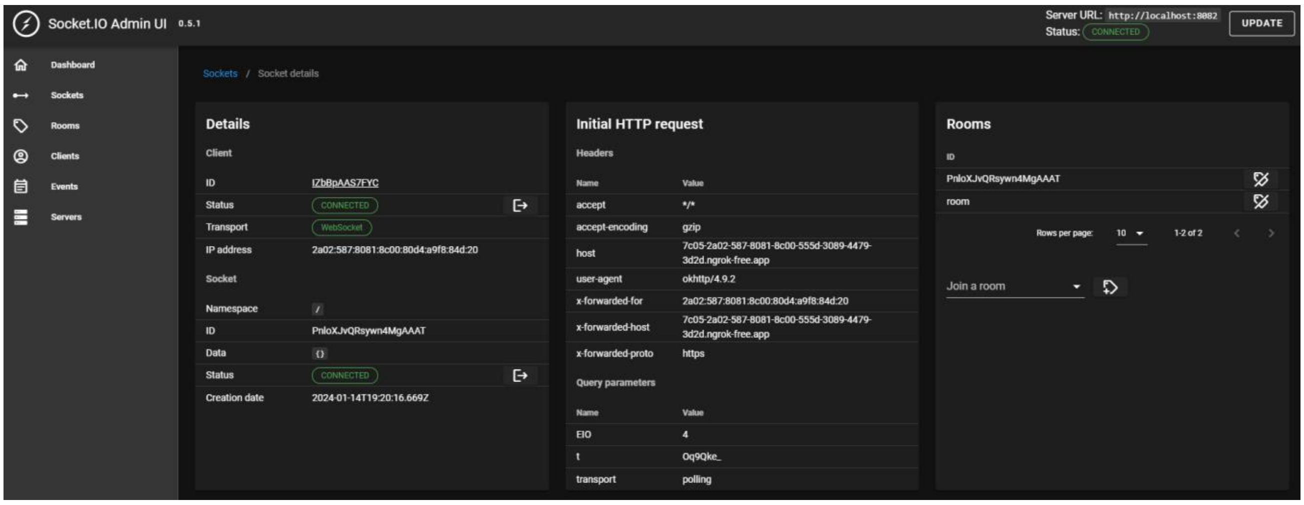
Status	#
HEALTHY	1 100.0 %
UNHEALTHY	0 0.0 %

Namespaces

Name	# of sockets
/	0
/admin	1

Figure 122 - Socket.io Admin Panel

By navigating to the "Sockets" tab, the user can view details about the active socket connections, including their status, transport method and general information related to the initial HTTP request:


Socket.IO Admin UI 0.5.1

Server URL: `http://localhost:8082`
Status: CONNECTED

Sockets / Socket details

Details

Client

ID	IZbBpAAS7FYC
Status	CONNECTED
Transport	WebSocket
IP address	2a02:587:8081:8c00:80d4:a9f8:84d:20

Socket

Namespace	/
ID	PnloXJvQRsywn4MgAAAT
Data	{}
Status	CONNECTED
Creation date	2024-01-14T19:20:16.669Z

Initial HTTP request

Headers

Name	Value
accept	*/*
accept-encoding	gzip
host	7c05-2a02-587-8081-8c00-555d-3089-4479-3d2d.ngrok-free.app
user-agent	okhttp/4.9.2
x-forwarded-for	2a02:587:8081:8c00:80d4:a9f8:84d:20
x-forwarded-host	7c05-2a02-587-8081-8c00-555d-3089-4479-3d2d.ngrok-free.app
x-forwarded-proto	https

Query parameters

Name	Value
EIO	4
t	Oq9Qke_
transport	polling

Rooms

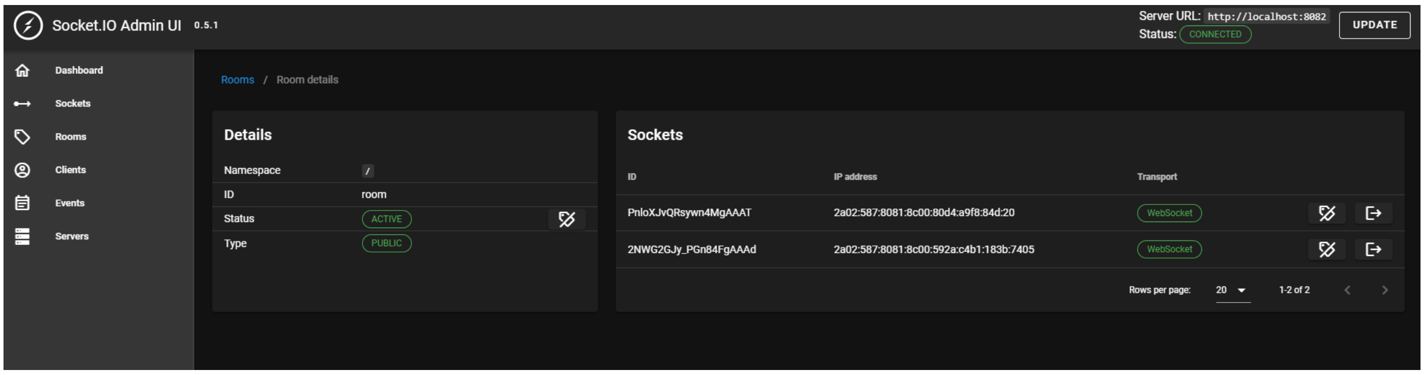
ID
PnloXJvQRsywn4MgAAAT
room

Rows per page: 10 1-2 of 2

Join a room

Figure 123 - Socket Details

Additionally, in the "Rooms" tab, the user can access information about the rooms and the active sockets within each room. This provides an overview of socket activity and room-specific details:


Socket.IO Admin UI 0.5.1

Server URL: `http://localhost:8082`
Status: CONNECTED

Rooms / Room details

Details

Namespace	/
ID	room
Status	ACTIVE
Type	PUBLIC

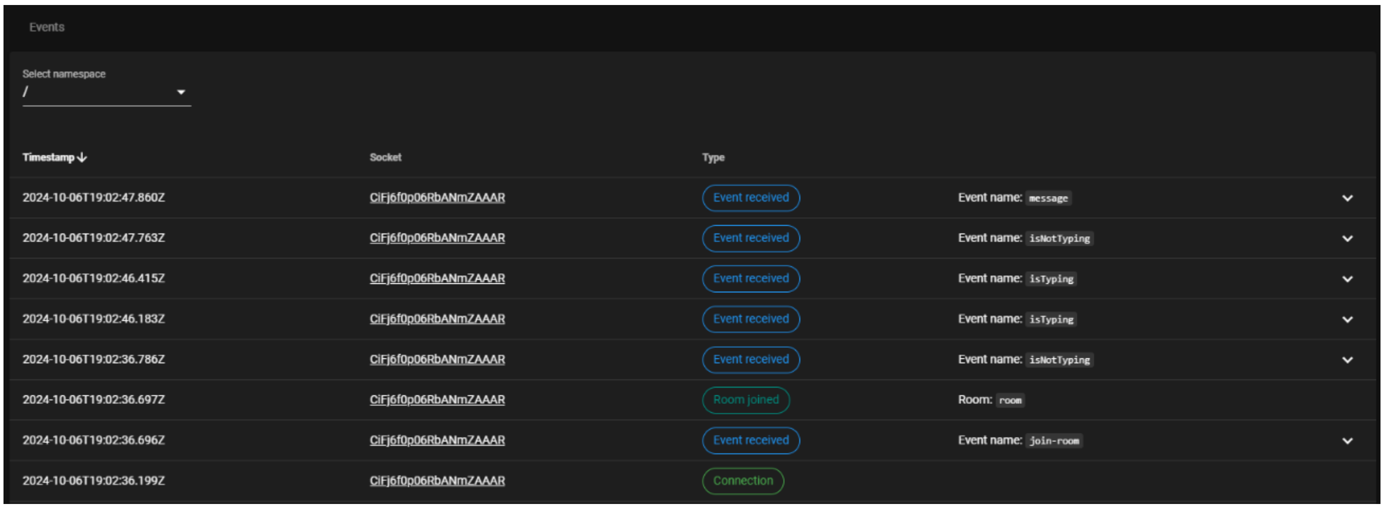
Sockets

ID	IP address	Transport
PnloXJvQRsywn4MgAAAT	2a02:587:8081:8c00:80d4:a9f8:84d:20	WebSocket
2NWG2GJy_PGn84FgAAAd	2a02:587:8081:8c00:592a:c4b1:183b:7405	WebSocket

Rows per page: 20 1-2 of 2

Figure 124 - Room Details

Useful information can also be found in the “Events” tab, showing information about the events:

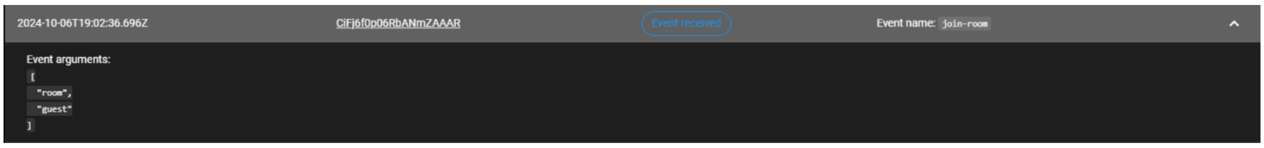

Events

Select namespace
/

Timestamp ↓	Socket	Type	
2024-10-06T19:02:47.860Z	CiFj6f0p06RbANmZAAAR	Event received	Event name: `message`
2024-10-06T19:02:47.763Z	CiFj6f0p06RbANmZAAAR	Event received	Event name: `isNotTyping`
2024-10-06T19:02:46.415Z	CiFj6f0p06RbANmZAAAR	Event received	Event name: `isTyping`
2024-10-06T19:02:46.183Z	CiFj6f0p06RbANmZAAAR	Event received	Event name: `isTyping`
2024-10-06T19:02:36.786Z	CiFj6f0p06RbANmZAAAR	Event received	Event name: `isNotTyping`
2024-10-06T19:02:36.697Z	CiFj6f0p06RbANmZAAAR	Room joined	Room: `room`
2024-10-06T19:02:36.696Z	CiFj6f0p06RbANmZAAAR	Event received	Event name: `join-room`
2024-10-06T19:02:36.199Z	CiFj6f0p06RbANmZAAAR	Connection	

Figure 125 - Events

By expanding an event, additional information will be displayed:


2024-10-06T19:02:36.696Z CiFj6f0p06RbANmZAAAR Event received Event name: `join-room`

Event arguments:

```
[
  "room",
  "guest"
]
```

Figure 126 - Event Details

8.4 Playwright Tests

Playwright's capabilities proved invaluable in developing the server application, offering an efficient way to automate testing for the REST APIs available on the server [25]. This allows each new addition or modification to requests to be automatically checked, effectively preventing potential side effects. Through the Swagger UI, manual tests could already be performed to verify that server responses matched expected outcomes as defined in the swagger.yaml file. To further automate this process, Playwright was used to run tests automatically based on the contents of the swagger.yaml file, where all API requests are defined.

Within the Node application, a Playwright project was initialized using the command "npm init playwright@latest" to install the latest version. This created a test folder, which was then customized to meet project needs by adding several subfolders and files:

- data: Contains test-data.js, a file for generating test data like user credentials and random IDs.
- helper: Holds helper functions organized as follows:
  - compareResponseBody: Compares the actual response from the test results with the expected response defined in the .yaml file.
  - getRequestBody: Retrieves the request body from the .yaml file for use in test requests.
  - screenshotError: Captures a screenshot for analysis when the response is unexpected.
- page-objects: Includes objects representing each test entity, such as HTTP requests and authentication methods:
  - Basic-authentication: Contains a class and methods to set up a basic authentication object.
  - Bearer-authentication: Contains a class and methods to set up a bearer authentication object.
  - Delete-request, Get-request, Put-request, Post-request: Files for handling respective HTTP request types.
- Swagger.spec: The main test file, responsible for opening the Swagger UI page and through DOM manipulation, testing REST APIs by leveraging test data, helper functions and page objects.

9 Application front-end

9.1 Create React Native app

To begin, a new React Native application was created using Node.js version 16.20.1 by running the command: npx create-expo-app villa-agapi
Next, the following folder structure was created:

- api: Contains API requests to interact with the server.
- assets: Stores styles, images and fonts.
- translations: Stores translation labels for multi-language support.
- store: Manages the application's global state.
- constants: Stores various constants such as authorization keys, URLs and styles
- models: Contains classes that assist with initializing data models.
- util: Contains reusable utility functions.
- app: Includes the root file responsible for starting the application and managing the navigation system.
- components: Houses reusable components for the app.
- screens: Holds all the available screens in the application.

9.1.1 Api

There are 15 available server requests, excluding the login request and for each of these, a separate JavaScript file was created in the "api" folder, with each handling a specific endpoint. All the requests follow a consistent structure: first, the host URL is imported from the constants file, along with the authentication token if required. Then, an asynchronous function is defined, taking any necessary URL parameters or payload as arguments. A const variable is created to store the full URL (combining the host and endpoint) and for requests requiring a body, another variable is defined to hold the payload. Next, an object called requestOptions is created with attributes for the HTTP method (POST, GET, PUT, DELETE), headers (e.g., authentication, content-type, cache control) and the request body if needed. Inside a try-catch block, the request is executed using the fetch method, passing the URL and requestOptions. The response is then stored in a variable named response and if no data is expected (e.g. for actions needing only a success or failure message), the response status is returned, otherwise, the relevant data from the request is returned. An example of the request, that was created to retrieve all the users of the application is displayed in the following code snippet.

```javascript
import { host } from "../constants/host";

export async function GetUsersRequest(token) {
  const url = `${host}/data/api/users`;

  const requestOptions = {
    method: "GET",
    headers: {
      Authorization: "Bearer " + token,
      "Content-Type": "application/json",
      "Cache-Control": "no-cache",
    },
  };

  try {
    const response = await fetch(url, requestOptions);
    const res = await response.json();

    if (response.status === 200) return res;
    else return res.status;
  } catch (error) {
    console.error("User request error: ", error);
    return null;
  }
}
```

9.1.2 Assets

The assets folder contains all the images and fonts used throughout the components of the application. Within this folder, there is a subfolder named fonts, where different fonts are stored, with "Poppins" serving as the primary font family for the app. Additionally, an images subfolder was created, organized into the following categories:

- aboutUs: Contains the image used in the About Us component.
- activities: Holds images used in the Activities component.
- categories: Includes images for the four navigation tiles on the home screen.
- home: Contains images used for the 360-degree view of the home.
- inside: Stores images of the interior of the home.
- locations: Includes images for the Locations component.
- markers: Contains images used in the Markers component.
- outside: Stores images of the exterior of the home, further divided into two subcategories:
    - day: Used in the carousel to showcase the house during the day.
    - night: Used in the carousel to showcase the house at night.

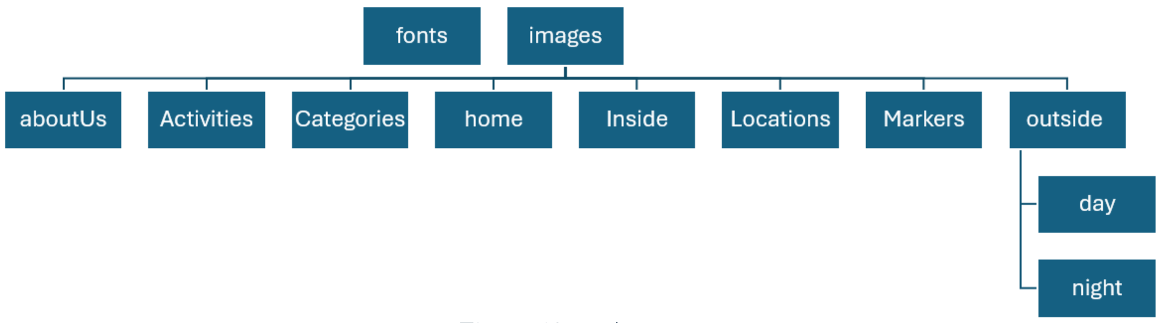
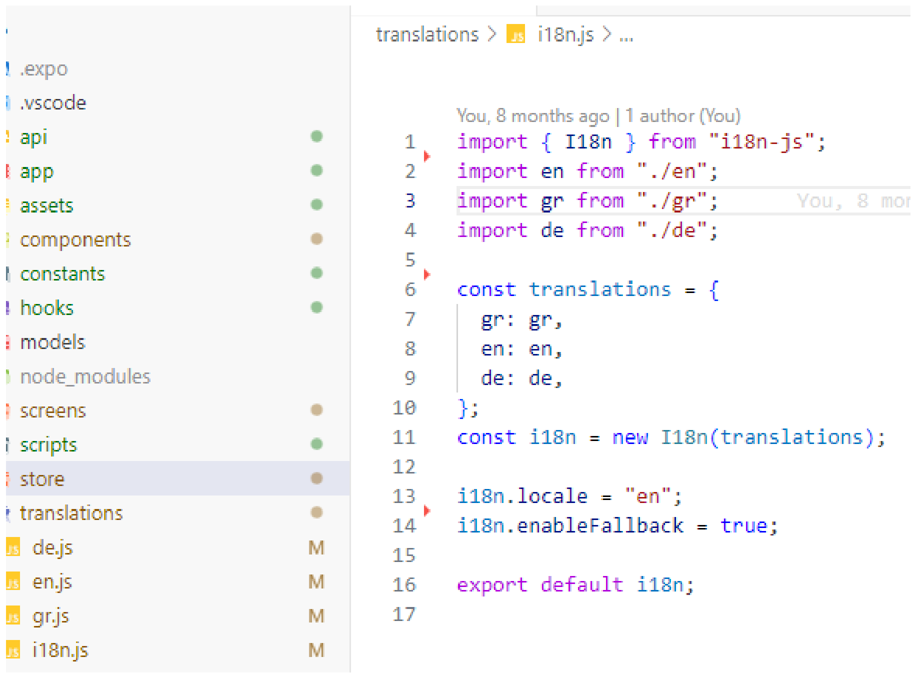


Figure 127 - Assets

9.1.3 Translations

The application's audience is consisted of people around the world, therefore it offers multilanguage support for three languages (English, Greek and German). Inside this folder there are four files, one for each language including the label for translation as a key and next to it the translated label. Finally, there is a file responsible to manage the translations, using the i18n package [26], by initializing the locale as default language the English with the ability to fallback to this language, if an invalid language is given:



```
import { I18n } from "i18n-js";
import en from "./en";
import gr from "./gr";
import de from "./de";

const translations = {
  gr: gr,
  en: en,
  de: de,
};
const i18n = new I18n(translations);

i18n.locale = "en";
i18n.enableFallback = true;

export default i18n;
```

Figure 128 - I18n

To make a label translatable within a component, these seven steps should be followed:

1. Import the i18n instance that was exported from the i18n.js file.
2. Import the application's store.
3. Create a state management variable using the useContext hook to handle the store's state.
4. Initialize a variable named locale with its setter function setLocale using the useState hook and set its initial value to the current locale from the store.

5. Use the useEffect hook with the current locale from the store as a dependency. This allows the component to react to store changes and update the locale variable by calling setLocale.
6. Assign the locale variable to i18n.locale.
7. Use i18n within a label like this: {i18n.t("<label key>.text")}.

```javascript
import { AuthContext } from "../store/auth-context";
import i18n from "../translations/i18n";
import React, { useState, useContext, useEffect } from "react";


function AboutUsScreen() {
  const authCtx = useContext(AuthContext);
  const [locale, setLocale] = useState(authCtx.currentLocale.toLow-
erCase());

  useEffect(() => {
    setLocale(authCtx.currentLocale.toLowerCase());
  }, [authCtx.currentLocale]);

  i18n.locale = locale;

  return (

    ...

    {i18n.t("about_us.text")}

    ...

    )

...

}
```

9.1.4 Store

This folder contains a file named "auth-context.js", which is responsible for managing the application's global state. This includes key information such as user details, the current theme (light or dark mode) and the selected language (English, Greek or German). The application's global state is managed using React's built-in Context API, ensuring that this information is available throughout the app's lifecycle while it is open. However, since the global state is cleared when the app is restarted, local storage is used to store user data. This is achieved using the react-native-async-storage package. The application's store is composed of five key parts:

1. Library Imports and Variable Initialization: First, the previously mentioned libraries are imported. A new context is created using

React's createContext function. Inside the exported function, several variables are initialized using the useState Hook, as shown in the following code snippet:

```
import AsyncStorage from "@react-native-async-storage/async-stor-
age";
import { createContext, useEffect, useState } from "react";

export const AuthContext = createContext({
  token: "",
  currentLocale: "en",
  userId: "",
  userName: "",
  role: "",
  currentMode: "",
  isAuthenticated: false,
  authenticate: (token) => {},
  logout: () => {},
  changeMode: (mode) => {},
  changeLocale: (locale) => {}
});

function AuthContextProvider({ children }) {
  const [authToken, setAuthToken] = useState(null);
  const [userId, setUserId] = useState(null);
  const [userName, setUserName] = useState(null);
  const [role, setRole] = useState(null);
  const [currentMode, setCurrentMode] = useState("light");
  const [currentLocale, setCurrentLocale] = useState("en");

...

}
```

2. Reducers: Functions that determine how the state changes in response to specific actions. This file contains four different consumers:
    1. changeMode: Changes the application's theme.
    2. changeLocale: Changes the application's locale.
    3. authenticate: Authenticates the user after login.
    4. logout: Handles user logout

```
function changeMode(mode) {
  setCurrentMode(mode);
}

function changeLocale(locale) {
  setCurrentLocale(locale === "GB" ? "EN" : locale);
}

function authenticate(token, userId, userName, role) {
  setAuthToken(token);
  setUserId(userId);
  setUserName(userName);
  setRole(role);

  const userInfo = `${token} ${userId} ${userName} ${role}`;
  AsyncStorage.setItem("token", userInfo);
}

function logout() {
  setAuthToken(null);
  setUserId(null);
  setUserName(null);
  setRole(null);
  AsyncStorage.removeItem("token");
}
```

3. Actions: An action is a function called by a consumer, as will be described later and is handled by the reducer.
4. Dispatching Reducer Functions: The reduce function for the respective action.
5. Provider: Provides this data to its child components, allowing them to access that data without the need for explicitly passing it through props at every level. It should be extracted from this file and imported to the root file, as displayed in the following code snippet:

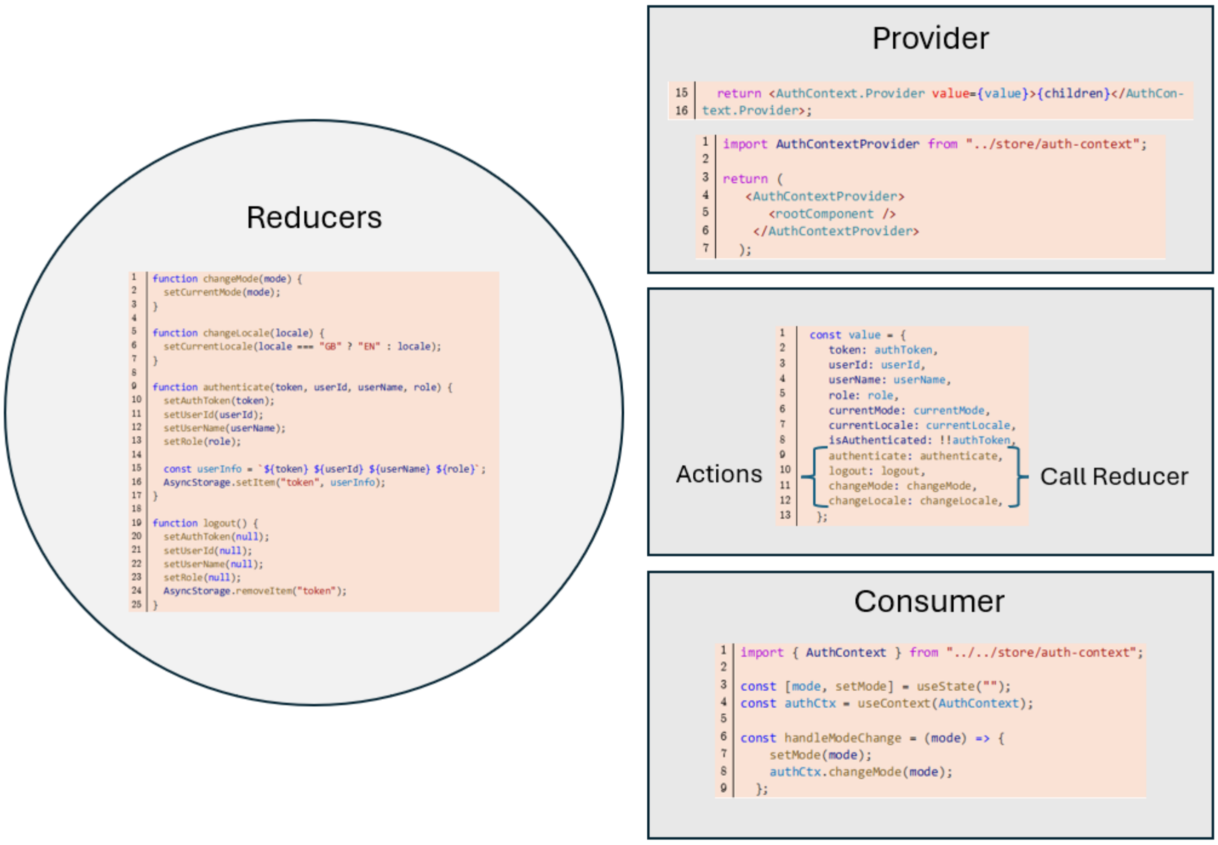
```
 const value = {
    token: authToken,
    userId: userId,
    userName: userName,
    role: role,
    currentMode: currentMode,
    currentLocale: currentLocale,
    isAuthenticated: !!authToken,
    authenticate: authenticate,
    logout: logout,
    changeMode: changeMode,
    changeLocale: changeLocale,
  };

  return <AuthContext.Provider value={value}>{children}</AuthCon-
text.Provider>;
```

```
import AuthContextProvider from "../store/auth-context";

return (
   <AuthContextProvider>
      <rootComponent />
    </AuthContextProvider>
  );
```

6. Consumer: A consumer utilizes these methods and data to interact with the store, allowing it to read and update the shared state:

```
import { AuthContext } from "../../store/auth-context";

const [mode, setMode] = useState("");
const authCtx = useContext(AuthContext);

const handleModeChange = (mode) => {
    setMode(mode);
    authCtx.changeMode(mode);
  };
```

The processes described before are shown in the following diagram:



```
function changeMode(mode) {
  setCurrentMode(mode);
}

function changeLocale(locale) {
  setCurrentLocale(locale === "GB" ? "EN" : locale);
}

function authenticate(token, userId, userName, role) {
  setAuthToken(token);
  setUserId(userId);
  setUserName(userName);
  setRole(role);

  const userInfo = `${token} ${userId} ${userName} ${role}`;
  AsyncStorage.setItem("token", userInfo);
}

function logout() {
  setAuthToken(null);
  setUserId(null);
  setUserName(null);
  setRole(null);
  AsyncStorage.removeItem("token");
}
```

```
  return <AuthContext.Provider value={value}>{children}</AuthCon-
text.Provider>;
```

```
import AuthContextProvider from "../store/auth-context";

return (
   <AuthContextProvider>
      <rootComponent />
    </AuthContextProvider>
  );
```

```
 const value = {
    token: authToken,
    userId: userId,
    userName: userName,
    role: role,
    currentMode: currentMode,
    currentLocale: currentLocale,
    isAuthenticated: !!authToken,
    authenticate: authenticate,
    logout: logout,
    changeMode: changeMode,
    changeLocale: changeLocale,
  };
```

```
import { AuthContext } from "../../store/auth-context";

const [mode, setMode] = useState("");
const authCtx = useContext(AuthContext);

const handleModeChange = (mode) => {
    setMode(mode);
    authCtx.changeMode(mode);
  };
```

Figure 129 - Global State Management

9.1.5 Constants

The Constants folder contains the following JavaScript files:

- auth.js: Stores the authentication token used for certain API requests.
- host.js: Defines the host URL used for making API calls.
- imageSources.js: Organizes arrays of image files from the assets folder, making them easy to manage and import into components for use.

9.1.6 Models

This folder contains information about the various models used within the application. Since each model typically contains similar, repeated information, a class has been created for each model. These classes include specific fields, which are initialized through a constructor. Whenever a new model is needed within a component, a new instance of the corresponding class is created and displayed within that component. The following models are available:

- Benefit: A Benefit the property provides, including the following fields:
    - Id
    - Title
- Category: A category (the four tiles that are available in the Home Page):
    - Id
    - Title
- Inside Image: Images of the house interior:
    - Id
    - Title
- Outside Image: Images of the house exterior:
    - Id
    - Title

9.1.7 Util

This folder contains reusable functions that can be used across the components for specific business processes. More specifically, the following files are available:

- Auth.js: A function named “login” is exported, which provides an HTTP request to log in a user. It accepts a username and password as arguments and sends the request to the server using the fetch method. If the response includes a token, it returns the full response body, otherwise it returns null.
- Dates.js: This file contains numerous methods related to dates such as:
    - calculateDuration: Accepts the arrival and departure date and returns the duration of the stay (on days).
    - getMonthFromDate: Extract the month from a date.
    - getYearFromDate: Extract the year from a date.
    - getDayFromDate: Extract the day from a date.
    - filterYearlyData: Takes in a dataset and a specific year, then filters and returns only the data corresponding to that year.

- getCountriesByYear: Accepts a list of users and returns the count of distinct countries per year, based on each user's arrival year and country.
- getDevicesByYear: Accepts a list of users and returns the count of devices used per year, based on each user's arrival year and device type.
- getMinDate: Receives a list of dates and returns the earliest date.
- getMaxDate: Receives a list of dates and returns the latest date.

- Location.js: This file contains helper functions related to the dynamic maps such as:
  - getAddress: Receives the latitude and longitude of a location as parameters and uses the Google Maps API to return the corresponding physical address:

```
const GOOGLE_API_KEY = "<API KEY>";

export async function getAddress(lat, lng) {
  const url = `https://maps.googleapis.com/maps/api/geo-
code/json?latlng=${lat},${lng}&key=${GOOGLE_API_KEY}`;

  const response = await fetch(url);

  if (!response.ok) {
    throw new Error("Failed to fetch address!");
  }

  const data = await response.json();
  const address = data.results[0].formatted_address;

  return address;
}
```

  - calculateDistance: Takes the latitude and longitude of two locations as parameters and calculates the distance between them using the Google Maps API:

```javascript
const GOOGLE_API_KEY = "<API KEY>";

export async function calculateDistance(lat1, lon1, lat2, lon2) {
  const url = `https://maps.googleapis.com/maps/api/direc-
tions/json?origin=${lat1},${lon1}&destina-
tion=${lat2},${lon2}&key=${GOOGLE_API_KEY}`;

  const response = await fetch(url);
  if (response.ok) {
    const data = await response.json();
    if (data.routes && data.routes.length > 0) {
      const distance = data.routes[0].legs[0].distance.text;
      return distance;
    } else {
      throw new Error("No routes found.");
    }
  }
}
```

- Socket.js: This file is responsible for initializing the socket connection using the socket.io-client package and connecting it to the Node.js application where the server-side socket events are implemented.

9.1.8 App

This file contains the root component responsible for initializing the application and managing its navigation system. Additionally, it wraps the entire application with the global state management provider introduced before, ensuring the store is accessible across all components. Outside of the root component, a separate component named *FlashMessage* (imported from the "react-native-flash-message" package) is included to display pop-up notifications for users. The Platform API is used within the *FlashMessage* component to adjust the position of these notifications, displaying them at the top of the screen on iOS devices and at the bottom on Android devices:

```jsx
 <AuthContextProvider>
      <Stack>
        <Stack.Screen name="(tabs)" options={{headerShown:false}} />
      </Stack>
        <FlashMessage position={Platform.OS === "ios" ? "top" :
"bottom"} />
    </AuthContextProvider>
  );
```

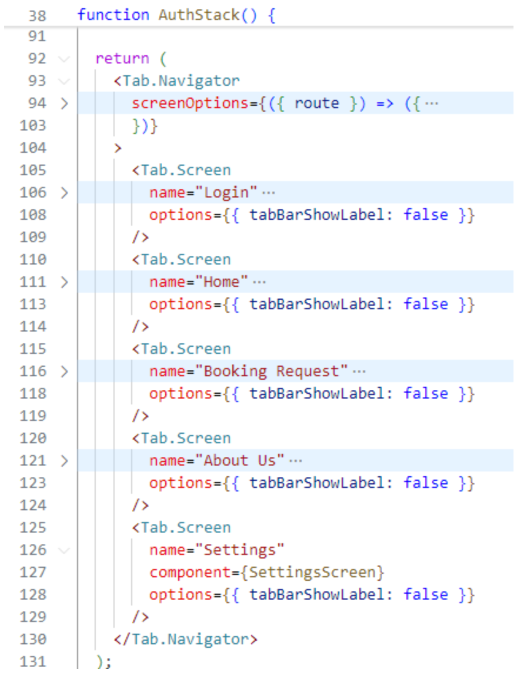
The navigation structure consists of a root stack navigator with the following screens:

- Auth: Screens accessible to non-registered users, organized under a tab navigator
- Authenticated: Screens available to registered users, also utilizing a tab navigator
- Inside: Displays interior views of the house

- Outside: Shows exterior views of the house
- Activities: Provides information on various activities
- Locations: Lists different locations
    - Information: Details about a selected location
- Map: A dynamic map displaying various markers
    - Details: Specific details about a selected marker
- Chat: Interaction with a chatbot
    - Live Chat: The live chat interface
- Add User: Screen for adding a new user
- Add Marker: Screen for adding a new marker
- Edit User: Allows editing of existing user profiles
- Booking Request: Preview of a booking request
- Availability: Interface for editing property availability

The Auth screen is defined within a function called AuthStack, which utilizes the Tab.Navigator component to create a navigation structure specifically for non-authenticated users. This tab navigator includes the following screens:

- Login: The login screen.
- Home: The home screen, which also includes the following stack screens previously described:
    - Inside
    - Outside
    - Activities
    - Locations
- Calendar: A screen that allows users to submit a direct booking request via a calendar.
- About Us: General information about the property.
- Settings: User settings, such as locale and theme preferences.

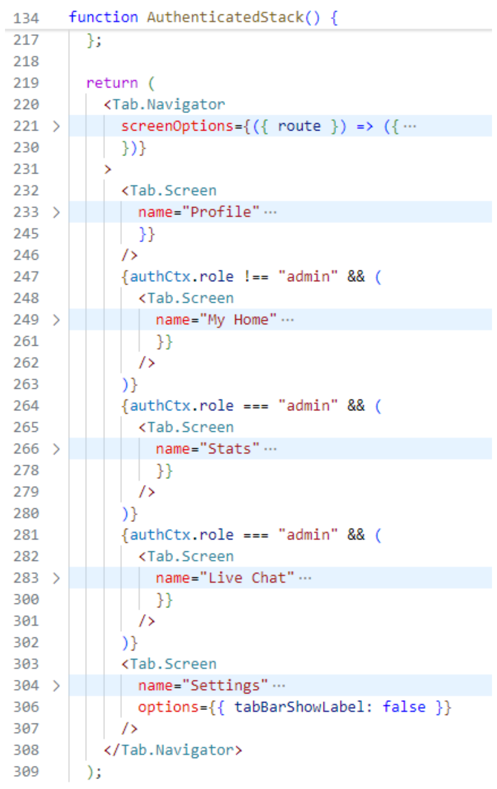


```
function AuthStack() {

  return (
    <Tab.Navigator
      screenOptions={({ route }) => ({⋯
      })}
    >
      <Tab.Screen
        name="Login"⋯
        options={{ tabBarShowLabel: false }}
      />
      <Tab.Screen
        name="Home"⋯
        options={{ tabBarShowLabel: false }}
      />
      <Tab.Screen
        name="Booking Request"⋯
        options={{ tabBarShowLabel: false }}
      />
      <Tab.Screen
        name="About Us"⋯
        options={{ tabBarShowLabel: false }}
      />
      <Tab.Screen
        name="Settings"
        component={SettingsScreen}
        options={{ tabBarShowLabel: false }}
      />
    </Tab.Navigator>
  );
```

Figure 130 - Auth Stack

The Authenticated screen is defined within a function called AuthenticatedStack, which uses the Tab.Navigator component to create a navigation structure specifically for authenticated users. Since there are two user roles (admin and guest), conditional rendering is applied based on the user's role in some cases. This tab navigator includes the following screens:

- Profile: A shared screen for both admins and guests, with different content based on the user type. Guests see information about their stay and personal profile, while admins have access to an admin dashboard. For admin users, the following stack screens are available within this section:
    - Add User
    - Edit User
    - Availability
    - Booking Request
    - Add Marker
- My Home: This screen provides information about the house interior, including 2D/3D property maps, a complaint/feedback form and host communication details. It is accessible only to guest users.
- Stats: Statistics about guests and property details, including booking information, user devices and countries of origin, presented on a yearly or total basis. Access is restricted to admin users only.
- Live Chat: This screen is directly accessible to admins. Guests can also access the live chat, but only through the Dialogflow screen.
- Settings: User settings, such as locale and theme preferences.
- Chat: The Dialogflow chatbot is accessible exclusively to guest users and is available across all screens that a guest can navigate.



```
function AuthenticatedStack() {
  };

  return (
    <Tab.Navigator
      screenOptions={({ route }) => ({···
      })}
    >
      <Tab.Screen
        name="Profile"···
        }}
      />
      {authCtx.role !== "admin" && (
        <Tab.Screen
          name="My Home"···
          }}
        />
      )}
      {authCtx.role === "admin" && (
        <Tab.Screen
          name="Stats"···
          }}
        />
      )}
      {authCtx.role === "admin" && (
        <Tab.Screen
          name="Live Chat"···
          }}
        />
      )}
      <Tab.Screen
        name="Settings"···
        options={{ tabBarShowLabel: false }}
      />
    </Tab.Navigator>
  );
```

Figure 131 - Authenticated Stack

The complete structure of the navigation system is illustrated in the following diagram:

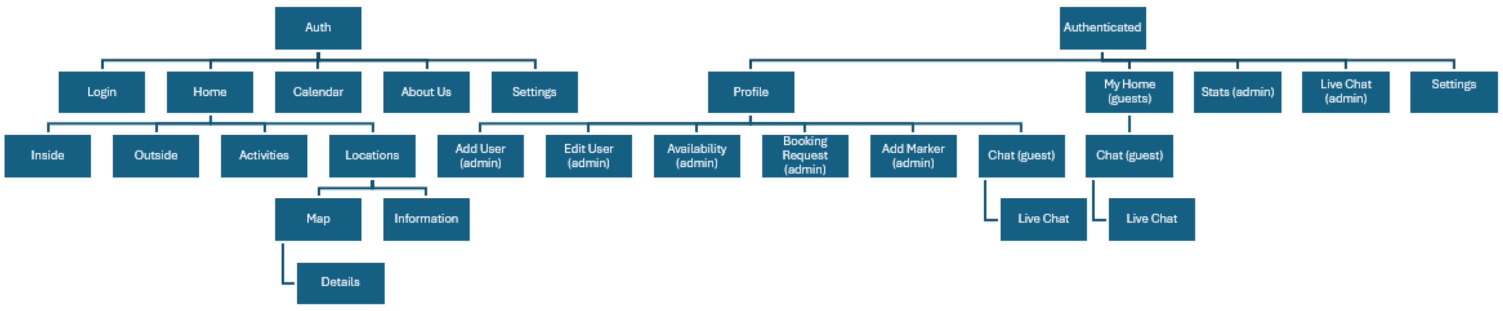


Figure 132 - Navigation System

To determine which of the available functions should be called (auth or authenticated), the application store relies on the value of authCtx.isAuthenticated. This value is set to true after a successful login and false after a user logs out. However, an additional scenario occurs when the user closes the app. In this case, the authentication token is automatically saved using the Async Storage package, which was set up in section "9.1.4 Store".
Each time the application is reopened, it checks whether the user is still authenticated, thereby implementing a "remember me" functionality. This ensures that users do not need to log in again after simply closing and reopening the app. Re-authentication is only required if the user manually logs out or if the token expires (after 30 days). The login schema is displayed in the following diagram:

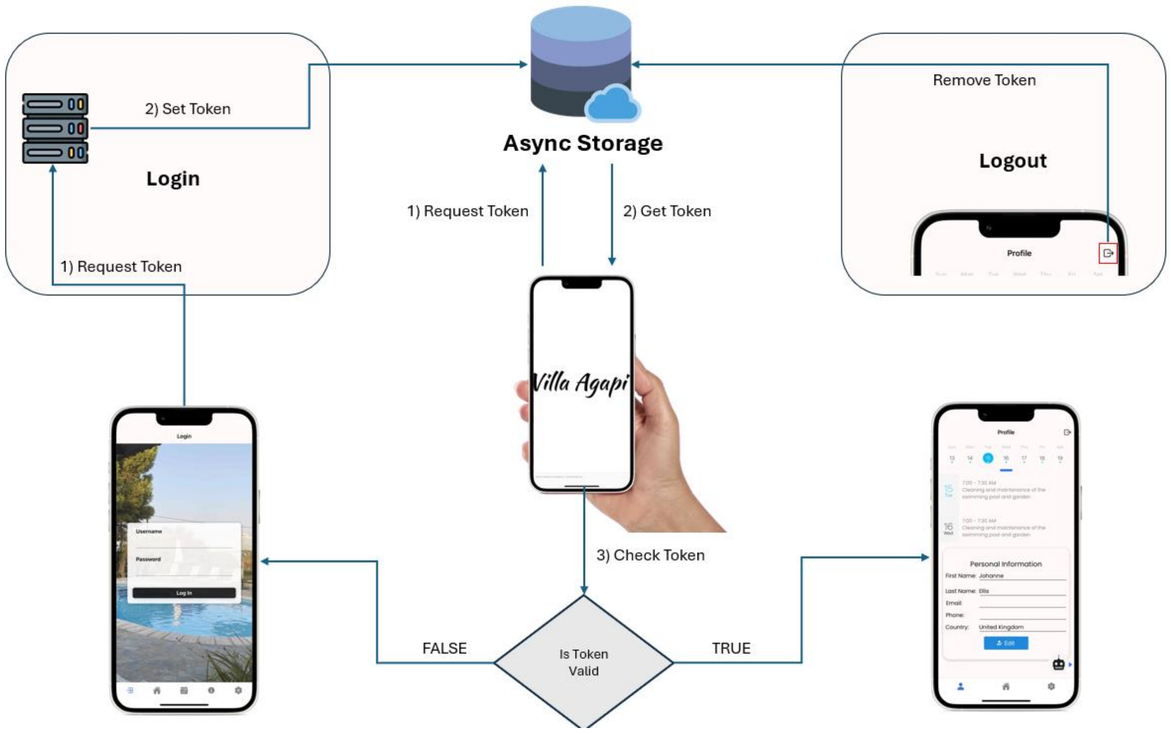


Figure 133 - Remember Me Functionality

9.1.9 Screens

The available screens and the users with access to each were outlined in the previous section on the navigation system structure. The only screen accessible to both authenticated and unauthenticated users is the settings screen, making it the first one to be checked. This screen contains global configuration settings that apply across all components of the application, managed through the context API to handle the global state efficiently. The application's theme can be controlled through a toggle button, allowing the user to see the changes immediately after pressing it. Similarly, a dropdown menu is provided to switch between the available languages offered by the application:

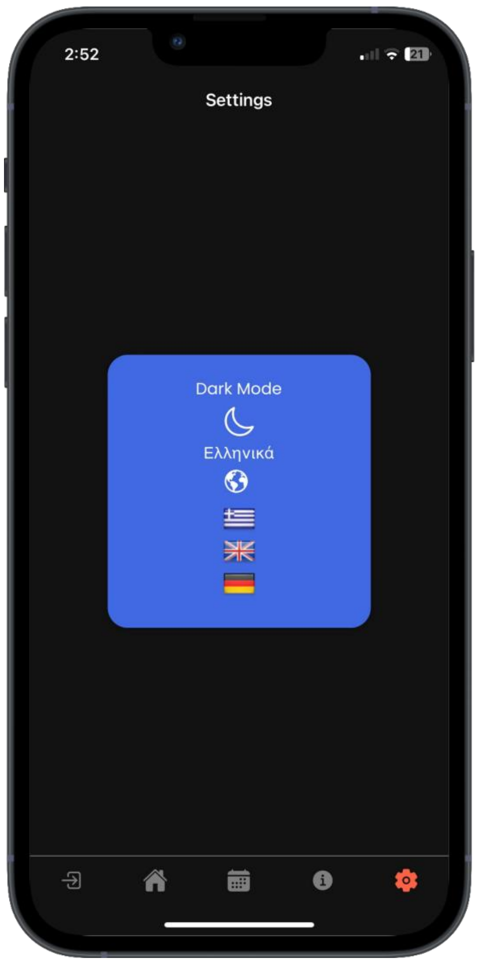


Figure 134 - Settings Screen

Next, the remaining screens for each user role will be described, providing screenshots of the application for both Android (left) and iOS (right). To highlight the differences between Light Mode and Dark Mode, the Android screenshots will display the Light Mode, while the iOS screenshots will showcase the Dark Mode.

9.2 Visitor

A visitor is an unauthenticated user of the application who has access to its basic features, including the tab screens available in the “auth” stack described in section "9.1.8 App". Visitors can also access the respective stack screens associated with these tab screens, providing them with limited functionality without requiring authentication.

9.2.1 Login Screen

The login form features a simple design. It consists of a white, semi-transparent container with rounded corners. Inside the container, there are two input fields: one for the username and one for the password. These input elements are imported from the "react-native-elements" library, which provides additional configuration options for each field:

Username Input Configuration:

- value: The current value of the input, managed with a useState hook.
- label: Displays the label "Username".
- labelStyle: Custom styles for the label, such as font, color, etc.
- autoCapitalize: Set to "none" to prevent automatic capitalization of the input.
- autocomplete: Set to "off" to disable text autocomplete.
- onChangeText: A function that updates the username value, using the same useState hook that controls the input value.
- enterKeyHint: Set to "next", enabling the user to move directly to the password field when pressing the button "next" on the keyboard.
- onSubmitEditing: A callback function that triggers when the user presses "next" automatically shifting the focus to the password field, enhancing the overall user experience by eliminating the need to manually tap into the next field.

Password Input Configuration

- value: The current value of the input, managed with a useState hook.
- label: Displays the label "Password".
- labelStyle: Custom styles for the label, such as font, color, etc.
- autoCapitalize: Set to "none" to prevent automatic capitalization of the input.
- secureTextEntry: Set to true, so the text input obscures the text entered
- onChangeText: A function that updates the username value, using the same useState hook that controls the input value.
- enterKeyHint: Set to "done", enabling the user to submit the login form.
- onSubmitEditing: A callback function that triggers when the user presses "done", by calling the form submission handler, ensuring the form is submitted without requiring the user to manually tap a submit button.

When the user fills in their login credentials and submits the form, the login function introduced in section "9.1.7" is triggered. If the provided credentials are valid, the following information is extracted from the server response:

- token
- userId
- username
- role

This data is then used to update the application's global state by calling the authenticate function, which receives these values as arguments. If the user provides invalid credentials or an error occurs (e.g., poor internet connection or server error), a modal will appear to inform the user.

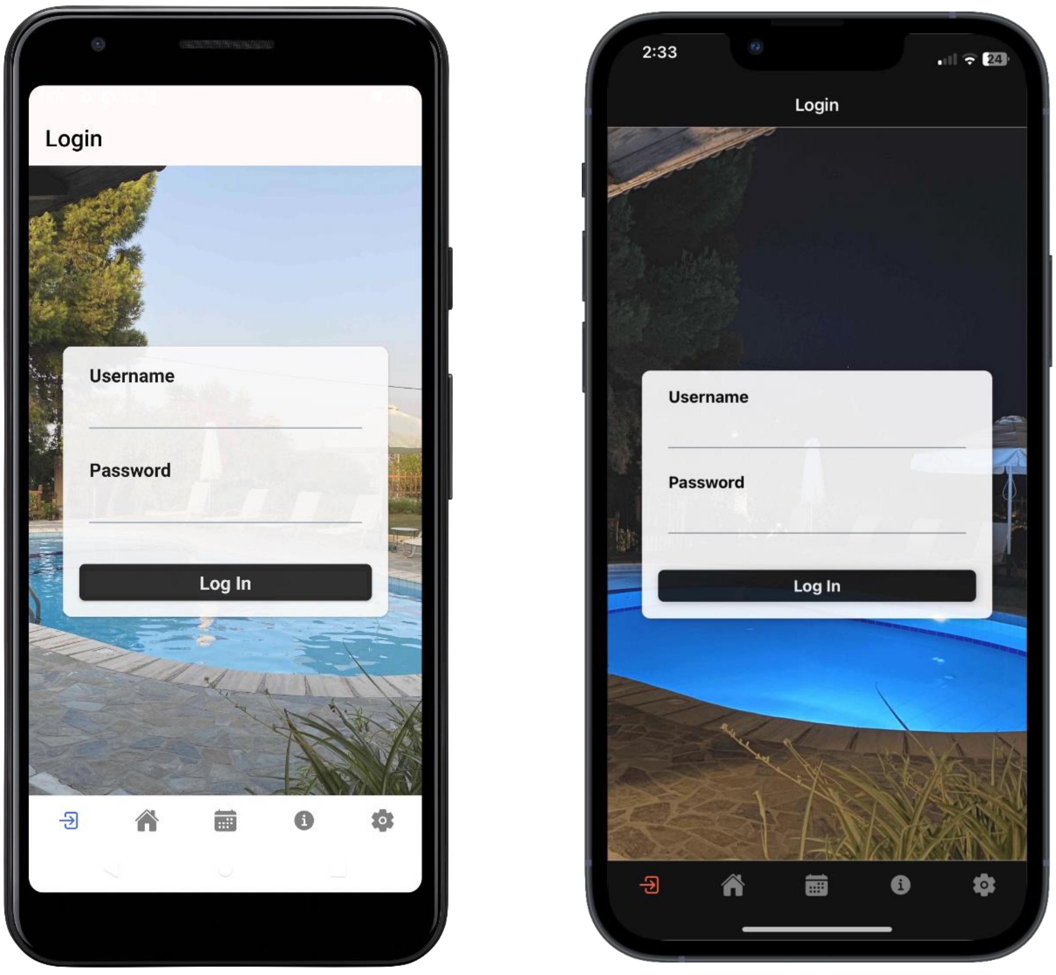


Figure 135 - Login Screen

9.2.2 Home Screen

The Home Screen is the second available screen for unauthorized users. At the top of the screen, a carousel displays images of the house using the "react-native-reanimated-carousel" package. The carousel automatically moves to the next image every five seconds, but users can manually scroll through the images, which resets the timer.

The images are imported from the constants folder using the following functions:

- getDayImageSources: retrieves the house's exterior images during the day.
- getNightImageSources: retrieves the house's exterior images at night.

Depending on the selected theme (light or dark mode) from the global state, either the day images are shown for the light theme or the night images for the dark mode.
Next, four tiles are displayed, each allowing for additional navigation when clicked. A new model, named "Category", is created to represent these tiles. For this purpose, the React Native FlatList component was used, enabling the display of multiple elements organized side by side, allowing control over the data to be displayed, the layout and the number of items shown per column.

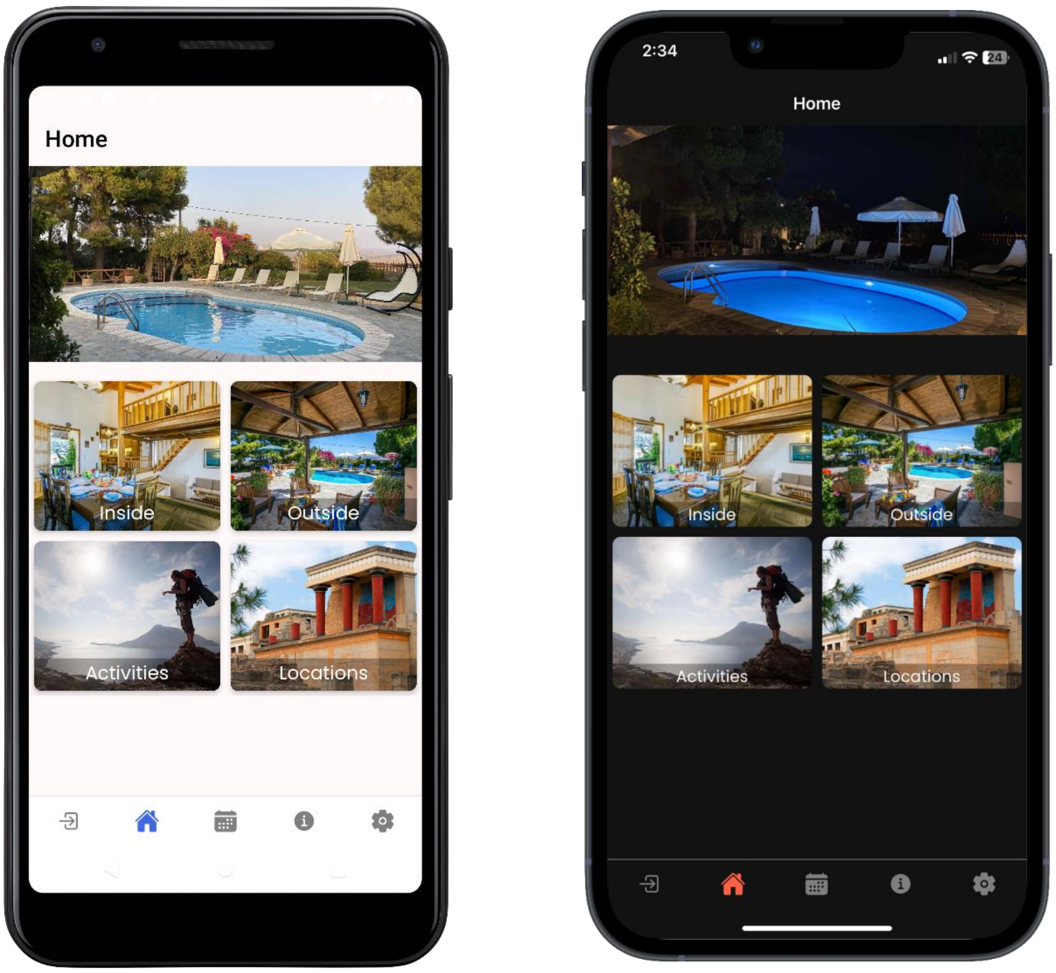


Figure 136 - Home Screen

9.2.3 Inside Screen

The Inside Screen is the first of the four tiles, providing information about the interior of the house.

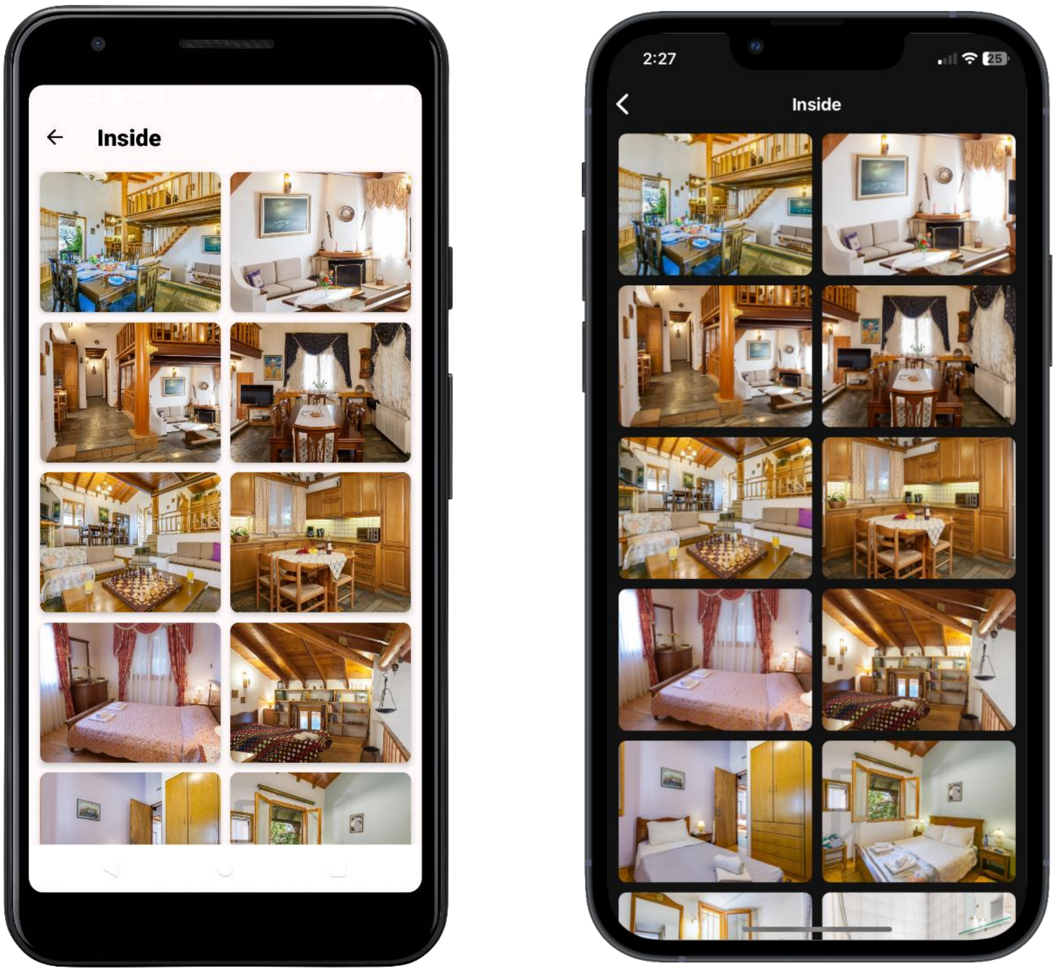
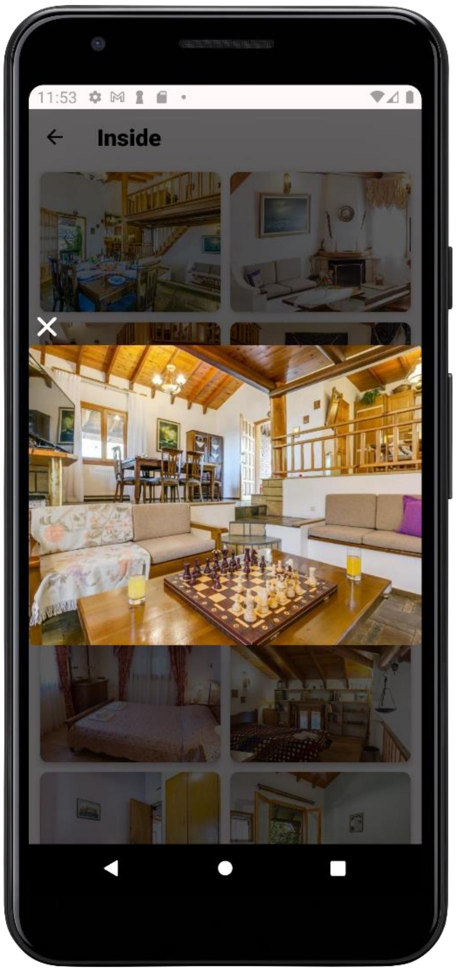


Figure 137 - Inside Screen

9.2.3.1 Image Viewer Component

When the user clicks on an image, a modal will pop up displaying the image in an enlarged view, with a darkened background. The user can close the modal either by clicking a "Close" button located in the top-left corner or by clicking the image again.



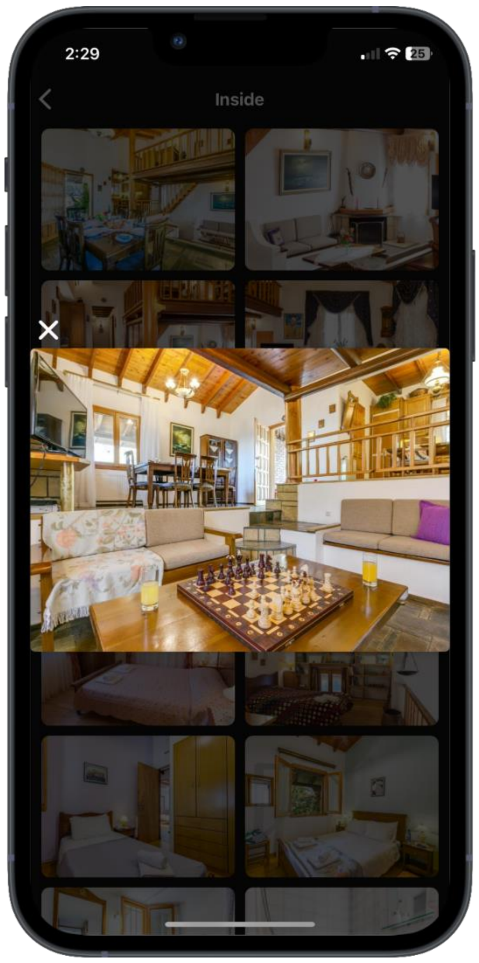
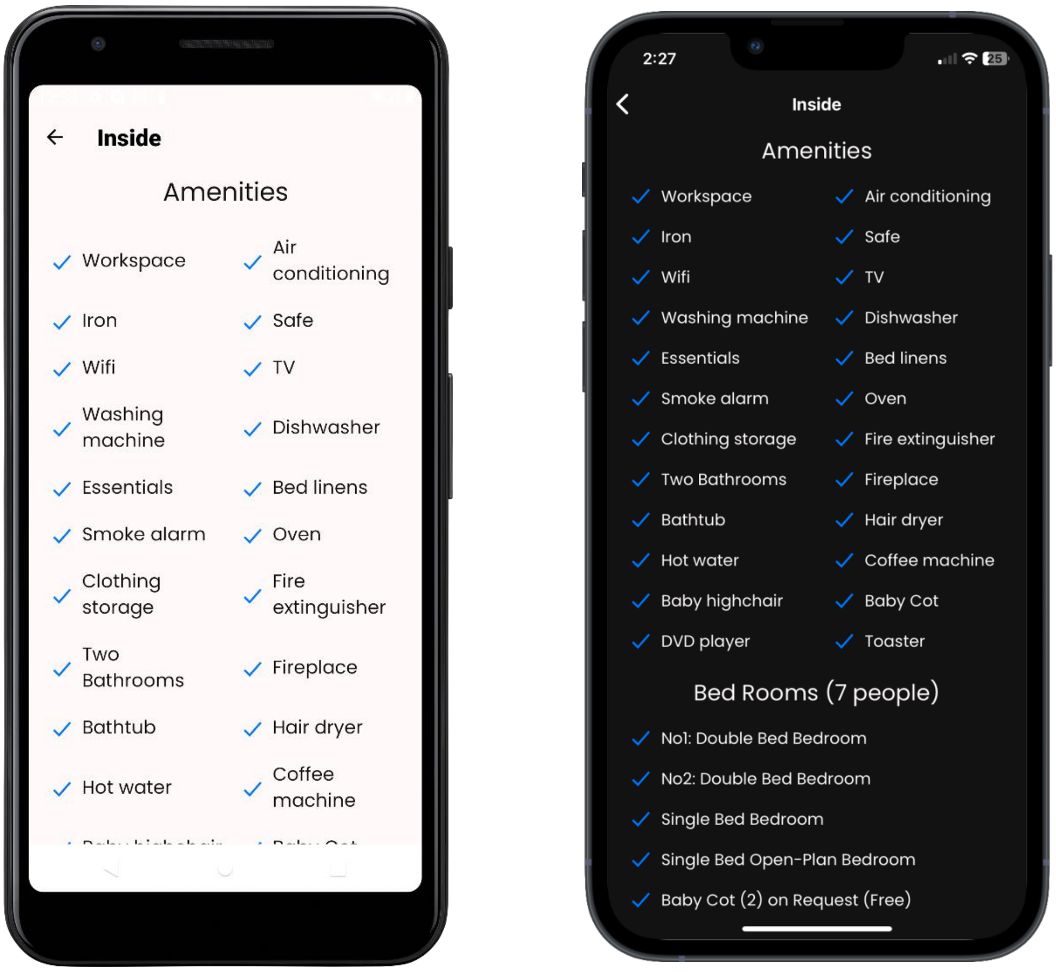


Figure 138 - Image Modal

9.2.3.2 List of Benefits

By scrolling down, additional information about the house appears, such as Amenities, Bedrooms and Cleaning. Each subcategory is displayed using respective “Benefit” models and FlatList components.



Inside

Amenities

- ✓ Workspace
- ✓ Air conditioning
- ✓ Iron
- ✓ Safe
- ✓ Wifi
- ✓ TV
- ✓ Washing machine
- ✓ Dishwasher
- ✓ Essentials
- ✓ Bed linens
- ✓ Smoke alarm
- ✓ Oven
- ✓ Clothing storage
- ✓ Fire extinguisher
- ✓ Two Bathrooms
- ✓ Fireplace
- ✓ Bathtub
- ✓ Hair dryer
- ✓ Hot water
- ✓ Coffee machine

2:27

Inside

Amenities

- ✓ Workspace
- ✓ Air conditioning
- ✓ Iron
- ✓ Safe
- ✓ Wifi
- ✓ TV
- ✓ Washing machine
- ✓ Dishwasher
- ✓ Essentials
- ✓ Bed linens
- ✓ Smoke alarm
- ✓ Oven
- ✓ Clothing storage
- ✓ Fire extinguisher
- ✓ Two Bathrooms
- ✓ Fireplace
- ✓ Bathtub
- ✓ Hair dryer
- ✓ Hot water
- ✓ Coffee machine
- ✓ Baby highchair
- ✓ Baby Cot
- ✓ DVD player
- ✓ Toaster

Bed Rooms (7 people)

- ✓ No1: Double Bed Bedroom
- ✓ No2: Double Bed Bedroom
- ✓ Single Bed Bedroom
- ✓ Single Bed Open-Plan Bedroom
- ✓ Baby Cot (2) on Request (Free)

Figure 139 - Benefits

9.2.4 Outside Screen

Similarly, there is another tile for the Outside, offering the same functionalities as the Inside Screen.

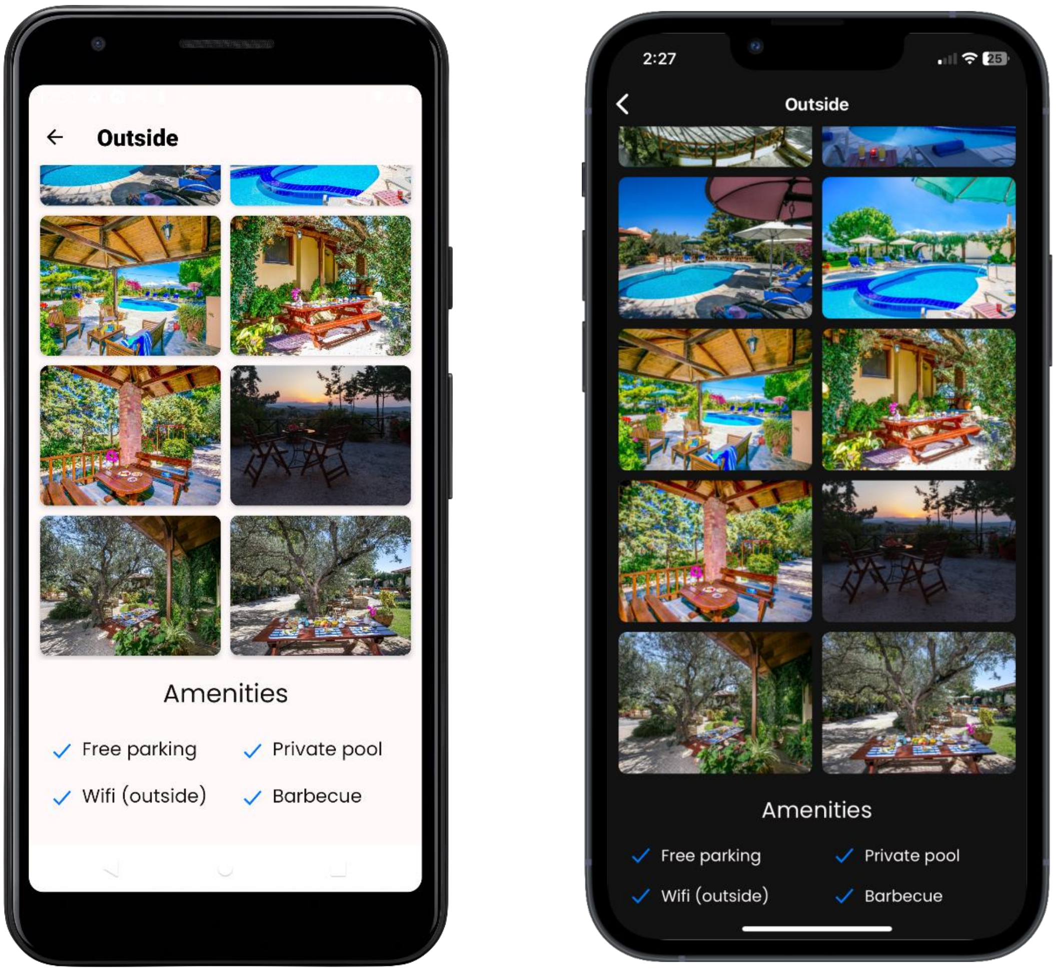
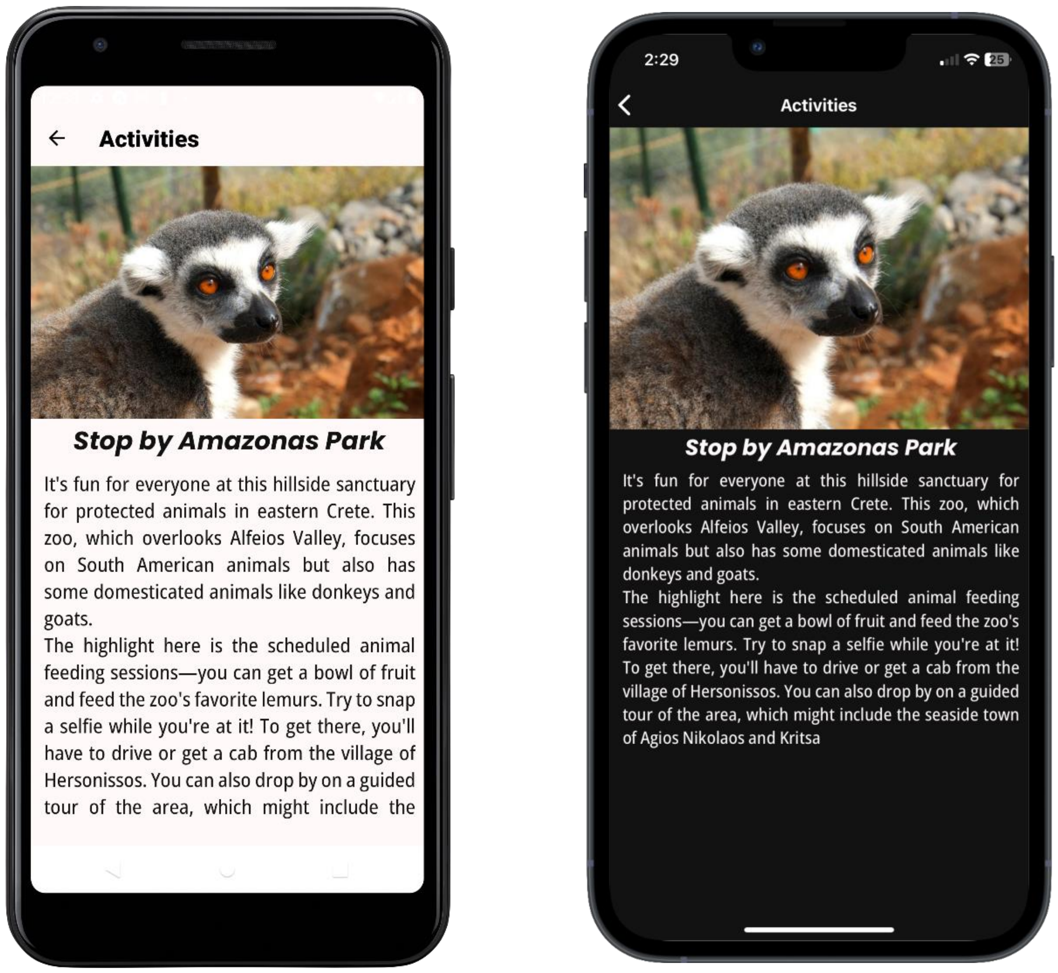


Figure 140 - Outside Screen

9.2.5 Activities Screen

The third tile available on the Home Screen is Activities. This screen utilizes a modified ScrollView component to create a horizontal layout with paging enabled, allowing users to feel as if they are turning pages by swiping to the right. Each "page" consists of an image, the name of the activity and additional details. Since the details text can be lengthy and users may wish to read it in their preferred language, this information is available in multiple languages.



Stop by Amazonas Park

It's fun for everyone at this hillside sanctuary for protected animals in eastern Crete. This zoo, which overlooks Alfeios Valley, focuses on South American animals but also has some domesticated animals like donkeys and goats.
The highlight here is the scheduled animal feeding sessions—you can get a bowl of fruit and feed the zoo's favorite lemurs. Try to snap a selfie while you're at it! To get there, you'll have to drive or get a cab from the village of Hersonissos. You can also drop by on a guided tour of the area, which might include the

Stop by Amazonas Park

It's fun for everyone at this hillside sanctuary for protected animals in eastern Crete. This zoo, which overlooks Alfeios Valley, focuses on South American animals but also has some domesticated animals like donkeys and goats.
The highlight here is the scheduled animal feeding sessions—you can get a bowl of fruit and feed the zoo's favorite lemurs. Try to snap a selfie while you're at it! To get there, you'll have to drive or get a cab from the village of Hersonissos. You can also drop by on a guided tour of the area, which might include the seaside town of Agios Nikolaos and Kritsa

Figure 141 - Activities Screen

9.2.6 Locations Screen

The Locations screen is the final navigation option accessible from the home page, allowing users to explore various recommended destinations. There are four main regions available, one for each prefecture of the island (Heraklion, Rethymno, Chania and Lasithi). For each prefecture, the four most popular spots have been selected, giving users a total of 16 locations (4x4) to explore. Users can scroll down to switch between prefectures and scroll horizontally to browse the locations within each one. As they scroll, an interactive map of the island updates in real-time to highlight the exact location of the selected destination. When a user clicks on a location, detailed information about that destination will be displayed. The text is fully translatable into the supported languages.
Additionally, users can switch between two views: the default list view or a dynamic map view for more detailed exploration. The map view is accessible via the map icon located at the top right of the screen.

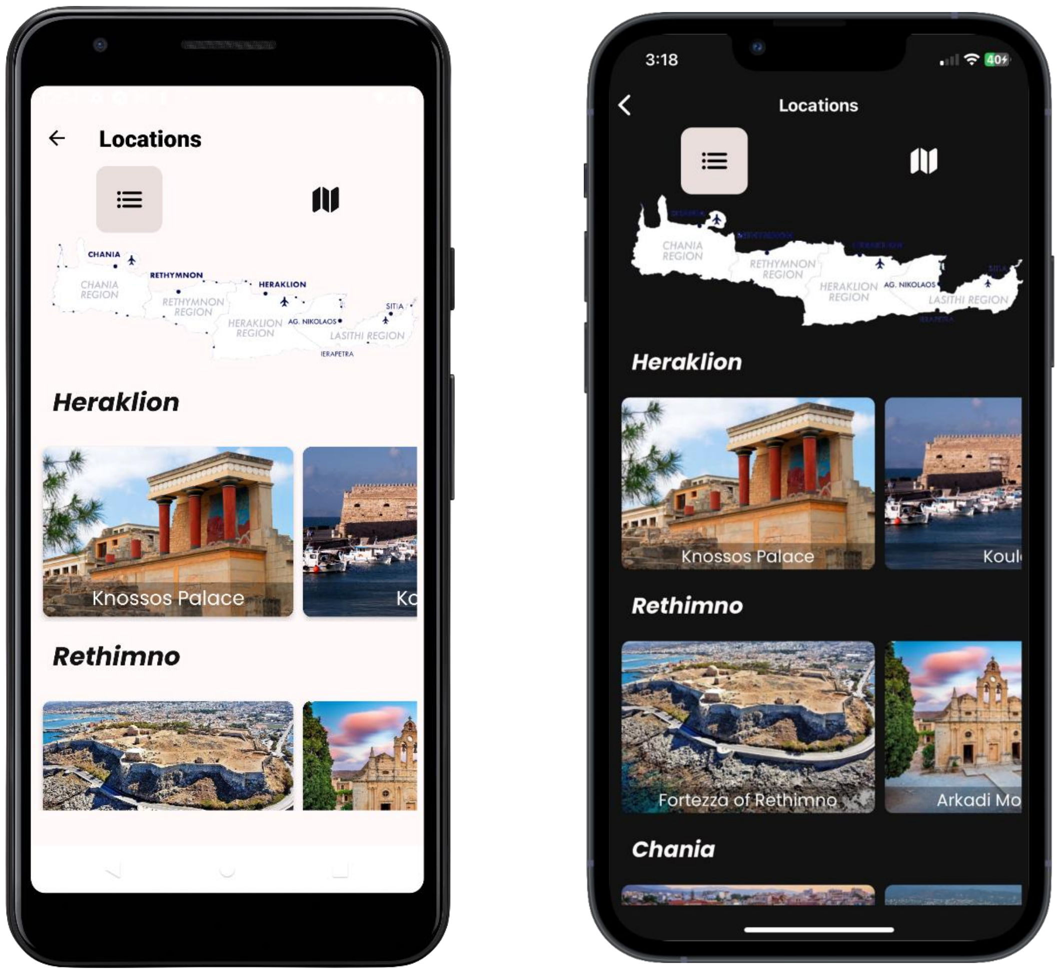
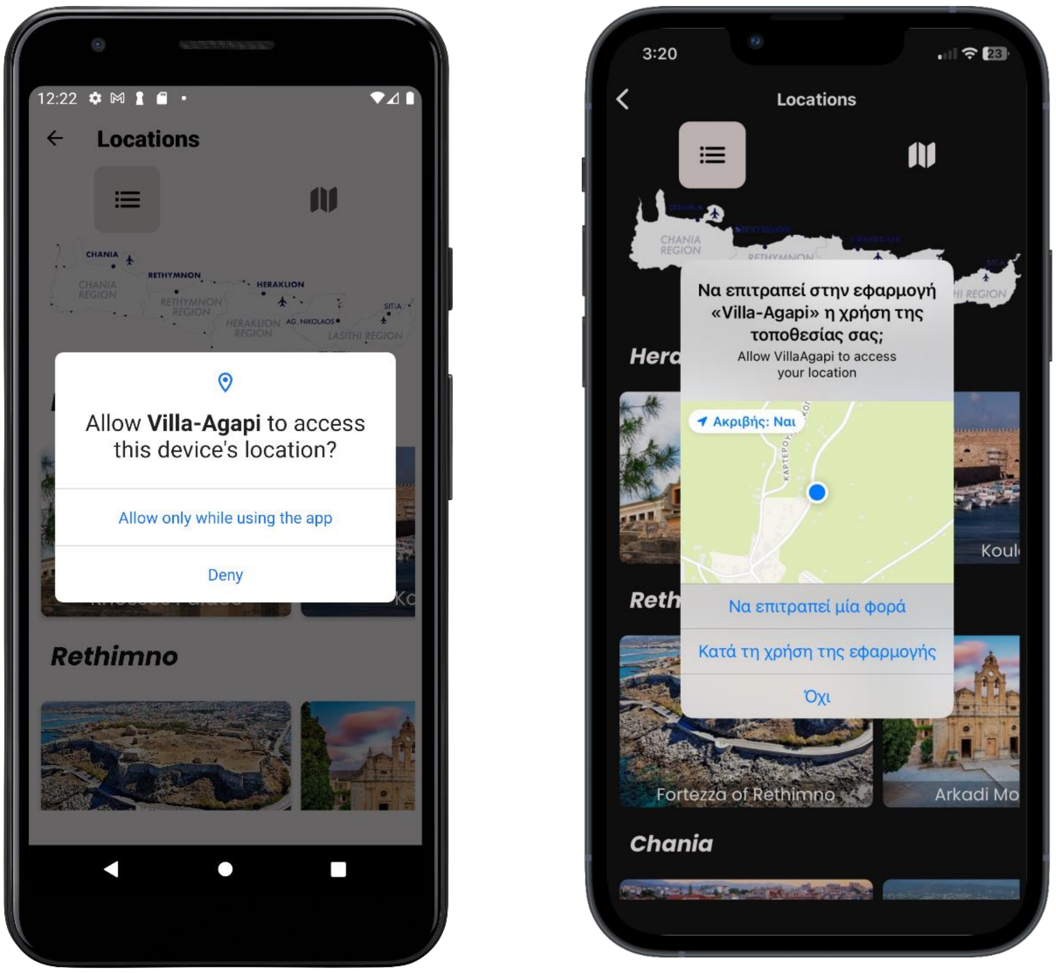


Figure 142 - Locations Screen

If the user wishes to switch to the map view, location access is required. A native modal will appear, requesting permission to access the user's location. The appearance of this modal will differ between iOS and Android devices (and may vary even between different Android devices), but the function remains the same: the user is asked to grant location access. Additionally, the message in the modal will be displayed in the device's selected language, as it is a native system prompt.
To implement location-based functionality in React Native, the expo-location package was used, specifically the "useForegroundPermissions" hook and the "PermissionStatus" enumeration. A variable is introduced to store the current permission status and a function to request permission if needed. When the user attempts to access the map, the app checks the current permission status: if it's undetermined, it prompts the user for permission and if previously denied, a modal is displayed asking the user to reconsider, followed by a new permission request. Once permission is granted, the user is navigated to the map screen. The described process is shown in the following code snippet:

```
import {
  getCurrentPositionAsync,
  useForegroundPermissions,
  PermissionStatus,
} from "expo-location";

export default function LocationsScreen() {

const [locationPermissionInformation, requestPermission] =
    useForegroundPermissions();

async function map() {
    const hasPermission = await requestGeolocationPermission();

    if (hasPermission) navigation.navigate("Map");
  }

async function requestGeolocationPermission() {
    if (
      locationPermissionInformation.status === PermissionSta-tus.UN-
DETERMINED
    ) {
      const permissionResponse = await requestPermission();

      return permissionResponse.granted;
    }

    if (locationPermissionInformation.status === PermissionSta-
tus.DENIED) {
      showModal();
      const permissionResponse = await requestPermission();
      return permissionResponse.granted;
    }

    return true;
  }

...

}
```


Allow **Villa-Agapi** to access this device's location?

Να επιτραπεί στην εφαρμογή «Villa-Agapi» η χρήση της τοποθεσίας σας;
Allow VillaAgapi to access your location

Figure 143 - Access Device's Location

9.2.7 Map Screen

Once the user grants access to their device's location, a map of the island with various location markers is displayed. For Android devices, Google Maps is used to render the map. However, due to a recent major update in Expo 51, Google Maps is no longer supported on iOS devices. As a result, Apple Maps is utilized for iOS, offering the same features and functionality as on Android just with different styles.
As shown in the screenshots below, markers are not visible individually. Instead, green circles with numbers inside are displayed, which represent clusters. This occurs because marker clustering is used to group markers that are located close to each other, replacing individual markers with a cluster that shows the total number of markers in that area. This approach was chosen to prevent overloading the map with too many markers, which could overwhelm the user. The clustering dynamically adjusts based on the map's zoom level: as the user zooms out, fewer clusters appear, with the minimum zoom showing a single cluster representing all markers. Conversely,

zooming in gradually reveals more individual markers as clusters break apart.

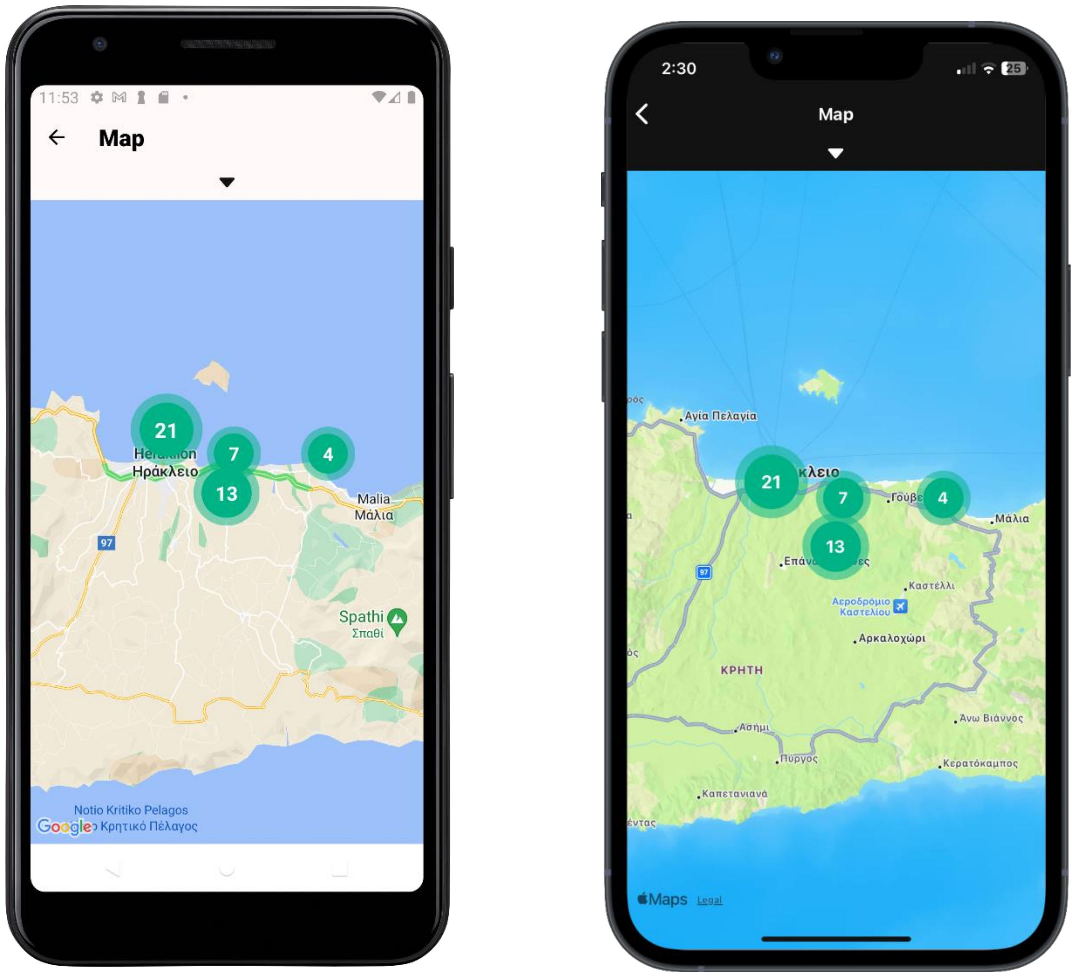


Figure 144 - Maps

9.2.7.1 Markers

As the user zooms in, individual markers start to appear in blue, each representing different types of locations based on their icons. These markers are categorized into the following main groups:

- Food
- Bank
- Drink
- Beach
- Market

- Health
- Fast-Food
- Activities
- Monuments
- Gast Station

These icons were imported by the expo/vector-icons package [27]

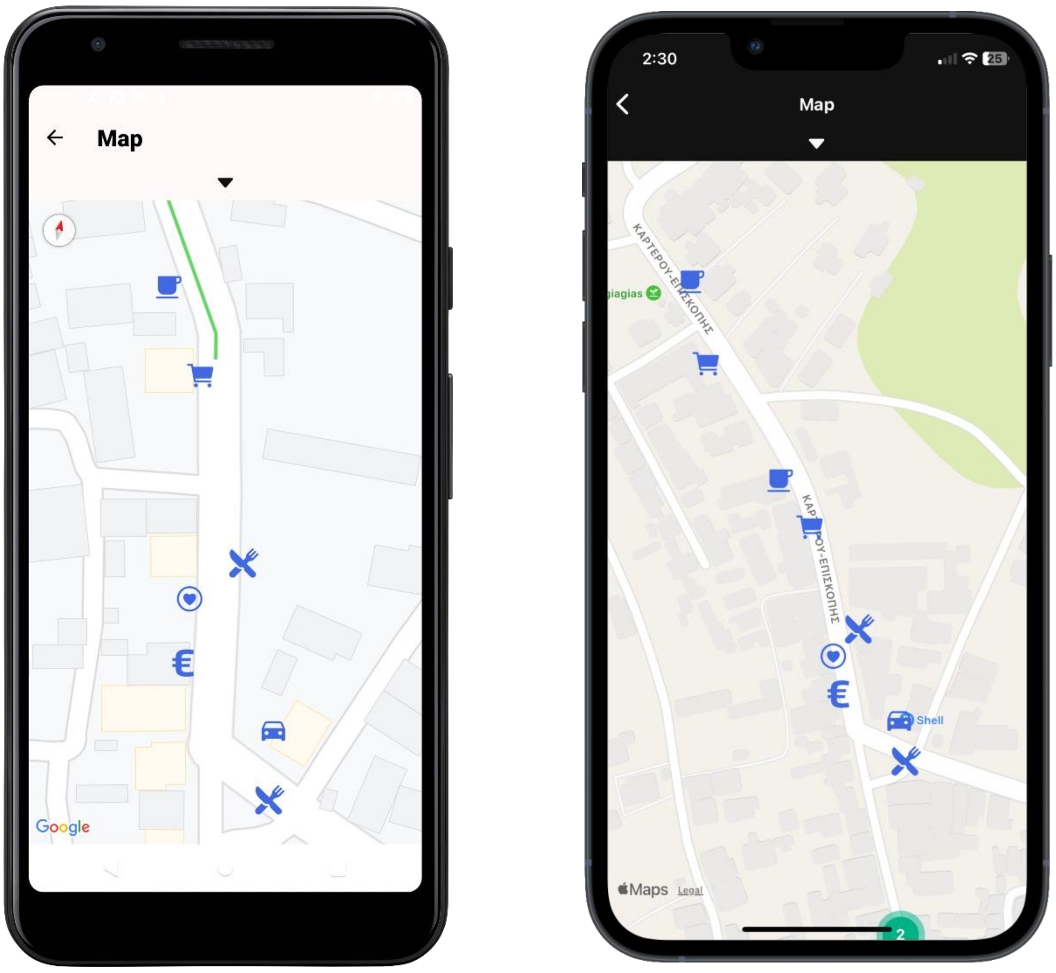


Figure 145 - Markers

9.2.7.2 Search filters

As expected, each type of marker is not strictly limited to the primary service it represents. For example, a gas station might also offer services like a café or quick snacks, allowing it to fall under multiple categories such as food, drink, or shopping. To handle this case, each marker contains additional information in the form of keywords that describe all the services available at that location. Users can apply search filters to easily find specific services or locations. These filters include a free-text search, where users can input any keyword and a dropdown menu offering predefined categories based on the marker types described earlier:

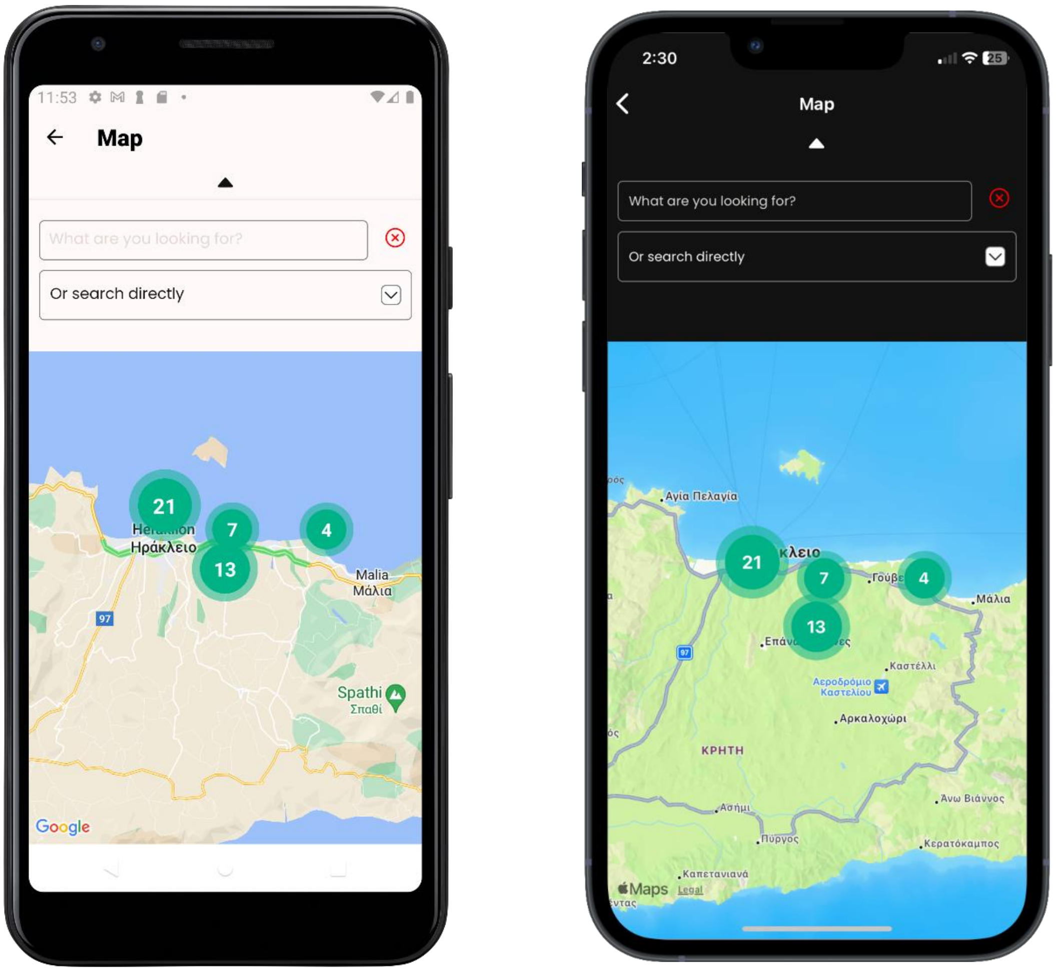


Figure 146 - Map Search Filters

The free-text search will start filtering results as soon as it matches a keyword from the array associated with each marker. For example, if the user begins typing "something to", all markers will appear because those words are not present in any marker's keywords. However, once the user types "eat", the system will immediately filter and display only the locations where eating is available, as "eat" matches one of the relevant keywords

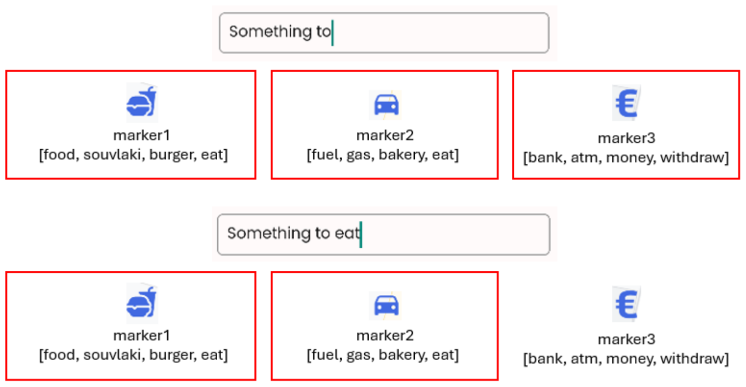


Figure 147 - Marker Search Example

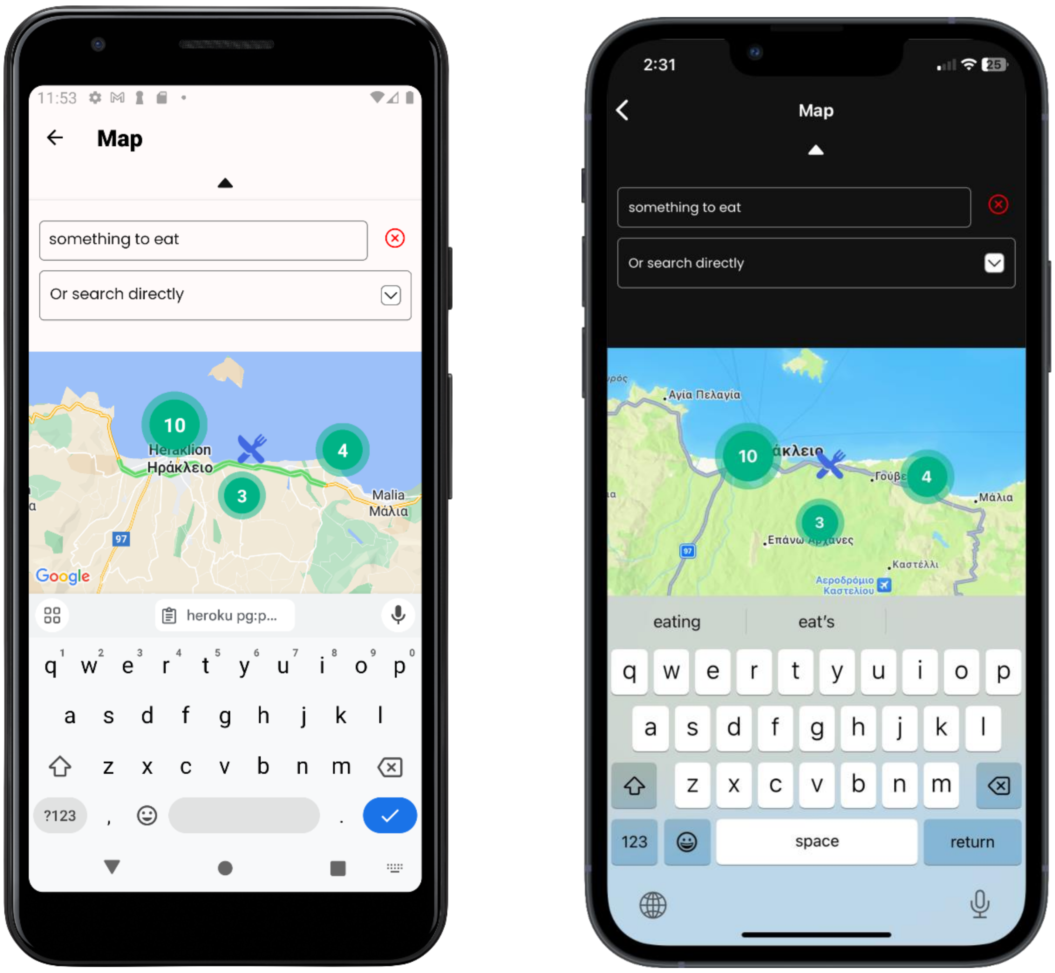


Figure 148 - Search Example

Similarly, the dropdown search filter displays all available types, allowing the user to select one. When a type is chosen from the dropdown, the free-text search input field is automatically populated with that option. This integration makes it easy for users to apply filters without needing to type. If the user wishes to remove the selected filter at any time, they can simply press the "close" icon next to the field, which will clear the filter and reset the search options.

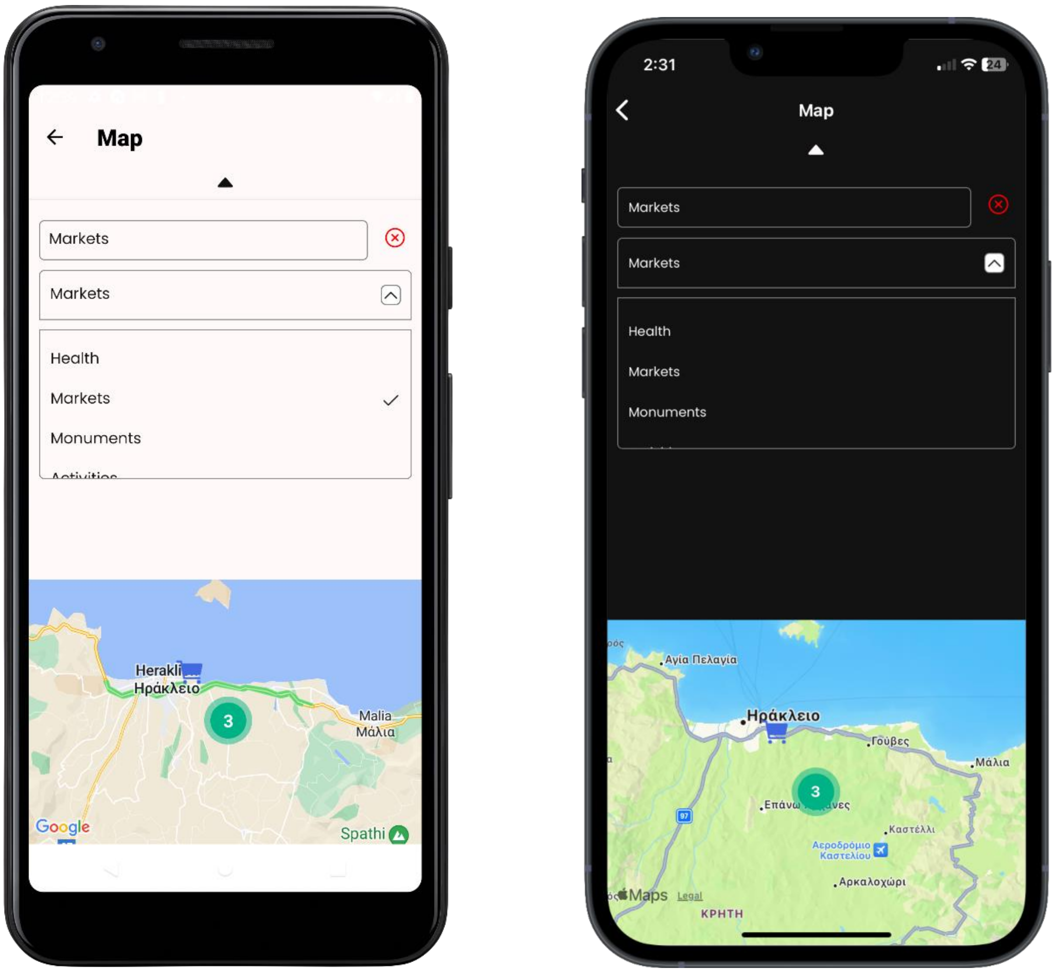


Figure 149 - Dropdown Search Filter

9.2.7.3 Marker Details

When the user clicks on a marker, a small text label will appear above the marker, displaying its name. Additionally, for Google Maps on Android devices, two icons will appear in the bottom right corner of the screen. The first icon allows users to open the Google Maps app directly, providing turn-by-turn directions and the route from the user's current location to the selected destination. The second icon shows the exact location of the marker on the map, also launching the Google Maps app for a more detailed view:

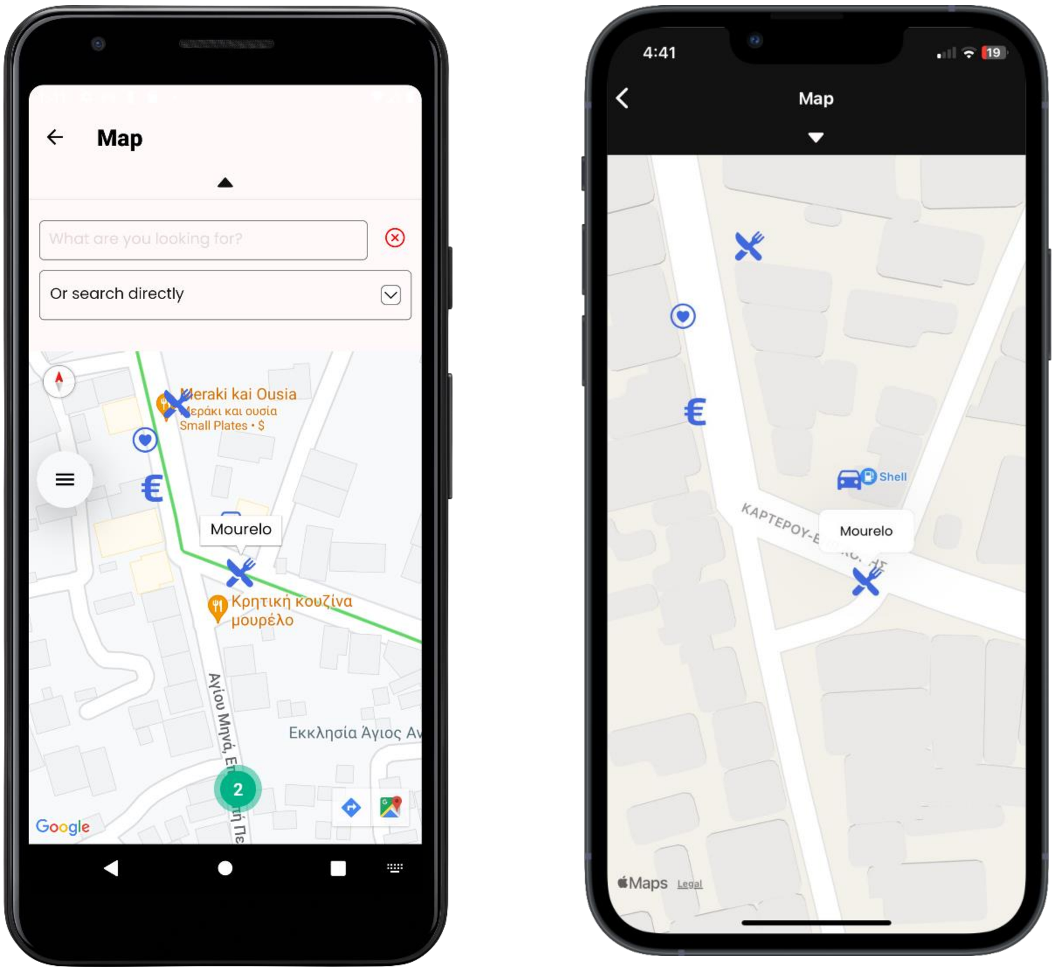
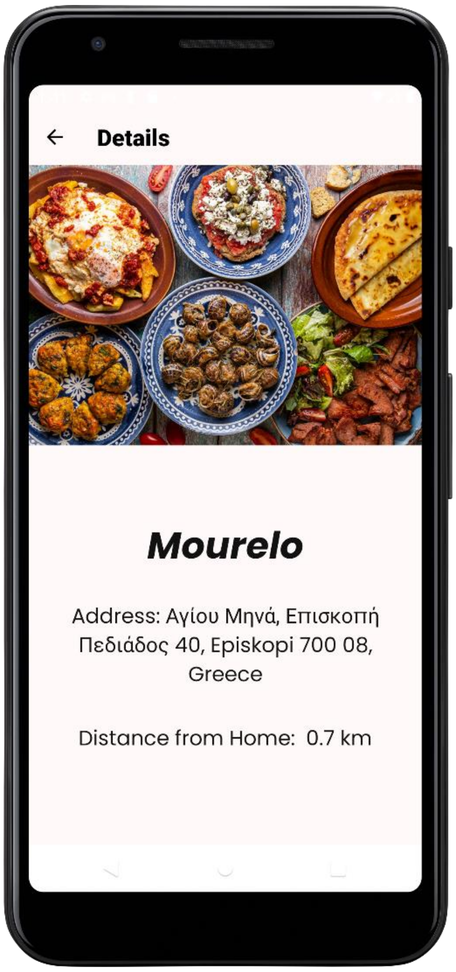


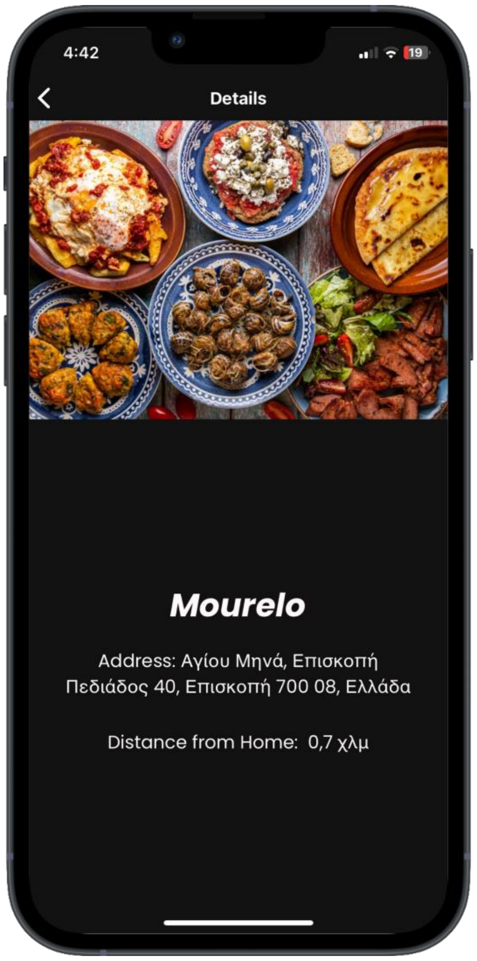
Figure 150 - Marker Name

When the user clicks on the label, they are taken to a new window that provides additional information about the selected marker. The first section of this window features an image related to the type of the marker. Below the image, useful information is displayed using the Google Maps API functions discussed in section “9.1.7.” This includes details such as the physical address of the location and the distance from the home.



Address: Αγίου Μηνά, Επισκοπή Πεδιάδος 40, Episkopi 700 08, Greece

Distance from Home: 0.7 km



Address: Αγίου Μηνά, Επισκοπή Πεδιάδος 40, Επισκοπή 700 08, Ελλάδα

Distance from Home: 0,7 χλμ

Figure 151 - Marker Details

9.2.8 Booking request screen

The Booking Request screen enables users to send direct booking requests to property owner, expressing their interest in reserving the house. Upon opening the component, users will see a calendar displaying the property's availability, along with a form for submitting their booking request. This form allows users to specify the number of visitors and enter personal information such as their email address, name and any additional comments. This information helps the host to contact the user for further details regarding their request.

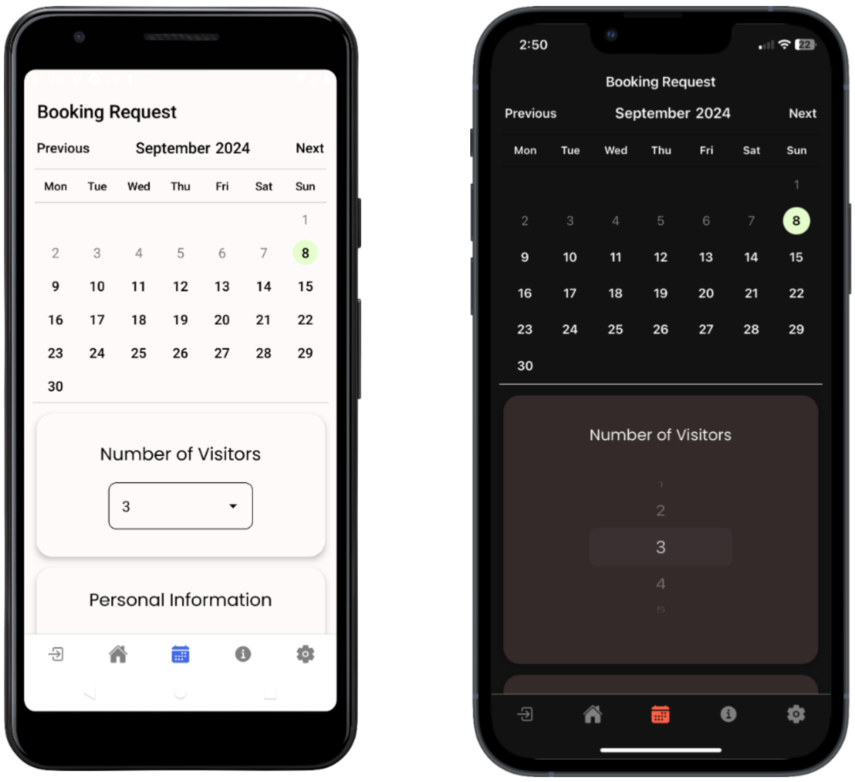


Figure 152 - Booking Request Screen

As a first step, the user needs to select the dates for their desired booking. This can be done by interacting with the calendar, where they can choose their check-in and check-out dates.

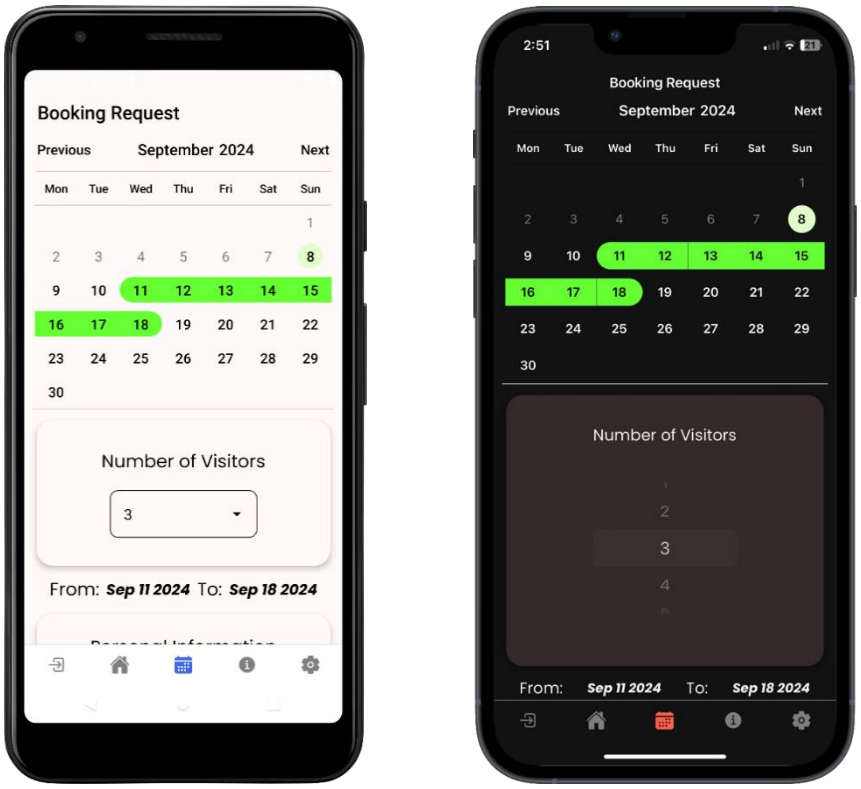
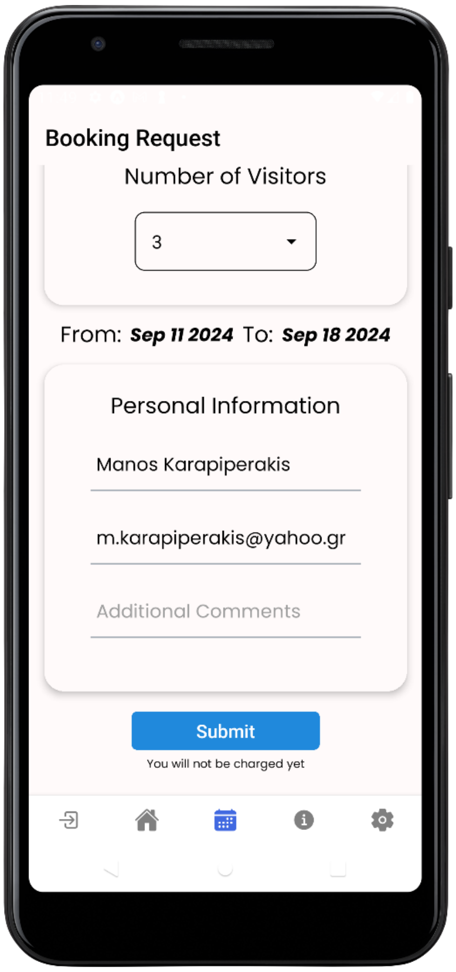

Booking Request
Previous
September 2024
Next
Mon Tue Wed Thu Fri Sat Sun
Number of Visitors
3
From: Sep 11 2024 To: Sep 18 2024

Figure 153 - Booking Request Select Dates

After selecting the dates, the user should fill out the card with their personal information, including their name and email address. Once the required fields are completed, the user can press the submit button to send their booking request. In this form, while the comments field is optional, both the name and email fields are mandatory. If either of these required fields is missing or if the email address provided is invalid, the submission process will trigger an error message displayed in a modal window:



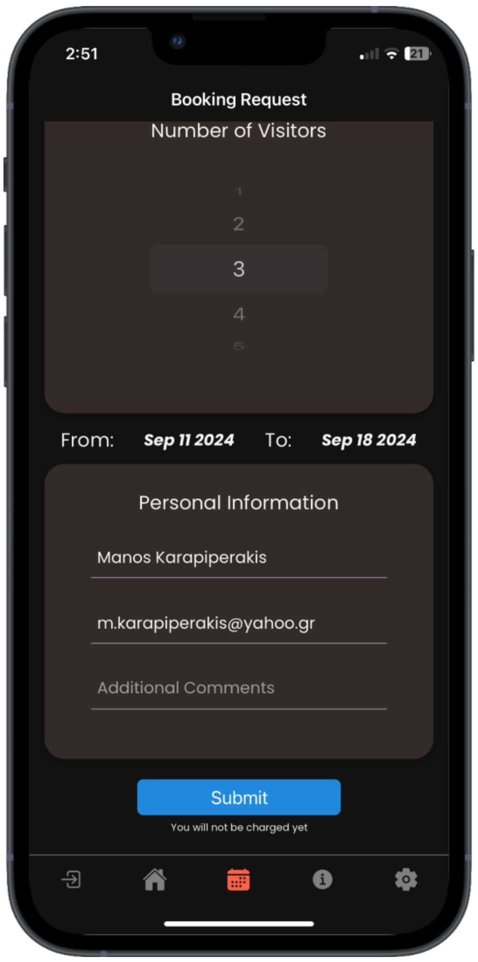


Figure 154 - Personal Information Card

If the form input is valid and all required information is correctly filled out, a success message will be displayed to the user, confirming that their booking request has been submitted successfully.

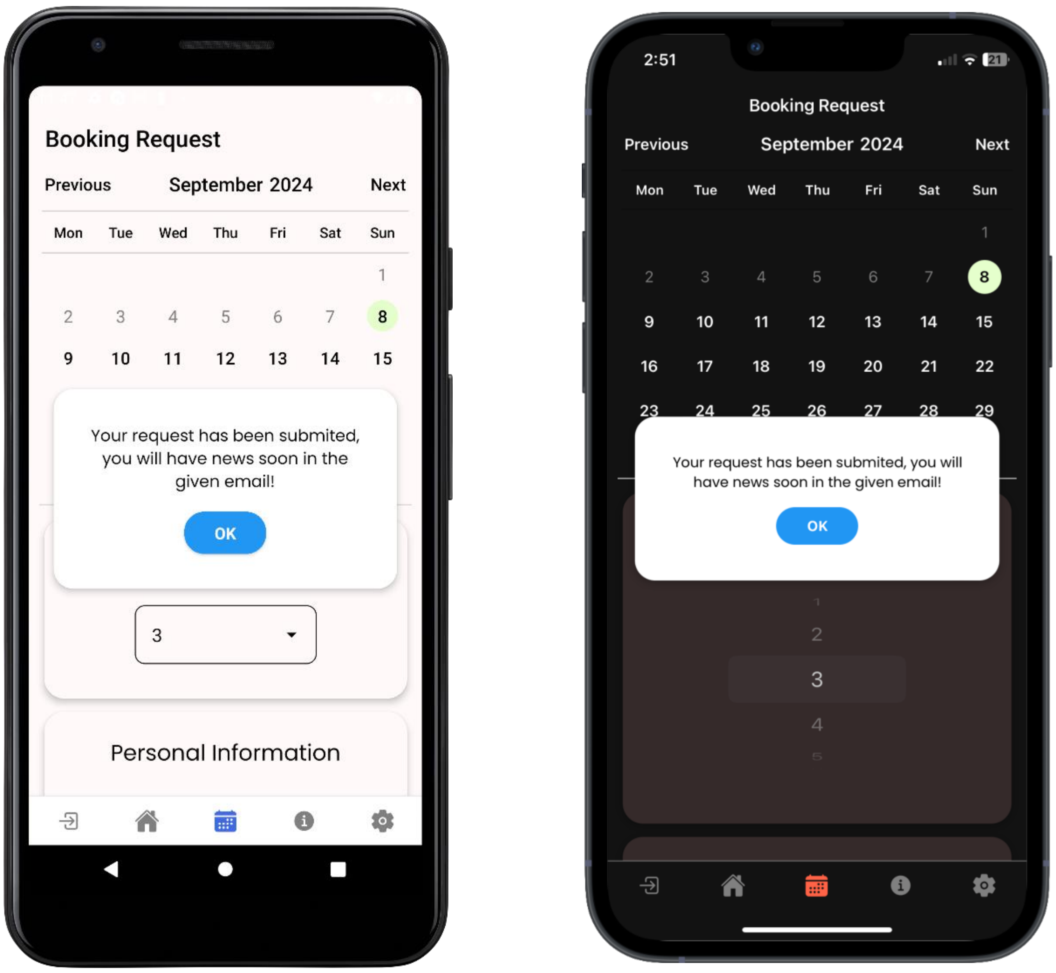
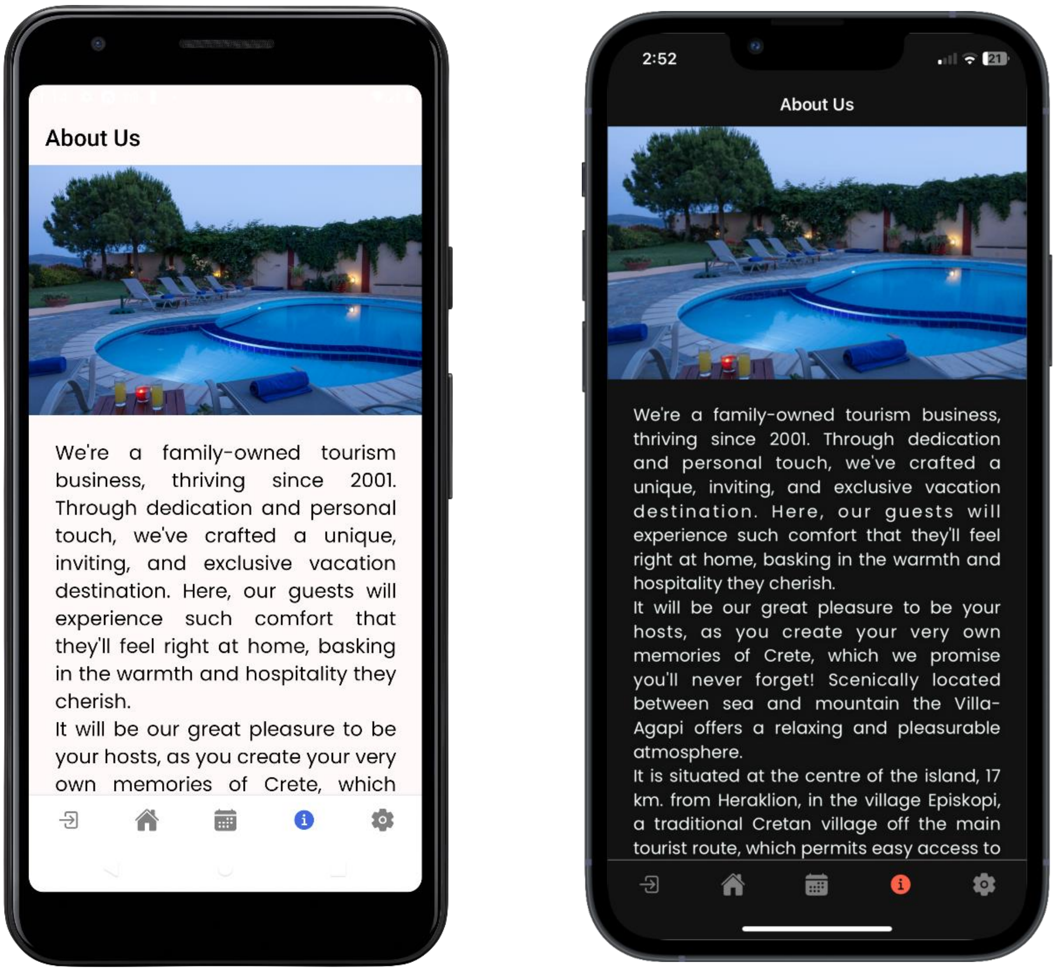

Booking Request
Previous
September 2024
Next
Mon Tue Wed Thu Fri Sat Sun
Your request has been submited, you will have news soon in the given email!
OK
3
Personal Information
2:51

Figure 155 - Confirmation Modal

9.2.9 About Us

This is the final screen available for unauthenticated users, providing general information about the host and the property. The content on this screen is designed to be translatable into all supported languages.


About Us

We're a family-owned tourism business, thriving since 2001. Through dedication and personal touch, we've crafted a unique, inviting, and exclusive vacation destination. Here, our guests will experience such comfort that they'll feel right at home, basking in the warmth and hospitality they cherish.
It will be our great pleasure to be your hosts, as you create your very own memories of Crete, which

2:52

About Us

We're a family-owned tourism business, thriving since 2001. Through dedication and personal touch, we've crafted a unique, inviting, and exclusive vacation destination. Here, our guests will experience such comfort that they'll feel right at home, basking in the warmth and hospitality they cherish.
It will be our great pleasure to be your hosts, as you create your very own memories of Crete, which we promise you'll never forget! Scenically located between sea and mountain the Villa-Agapi offers a relaxing and pleasurable atmosphere.
It is situated at the centre of the island, 17 km. from Heraklion, in the village Episkopi, a traditional Cretan village off the main tourist route, which permits easy access to

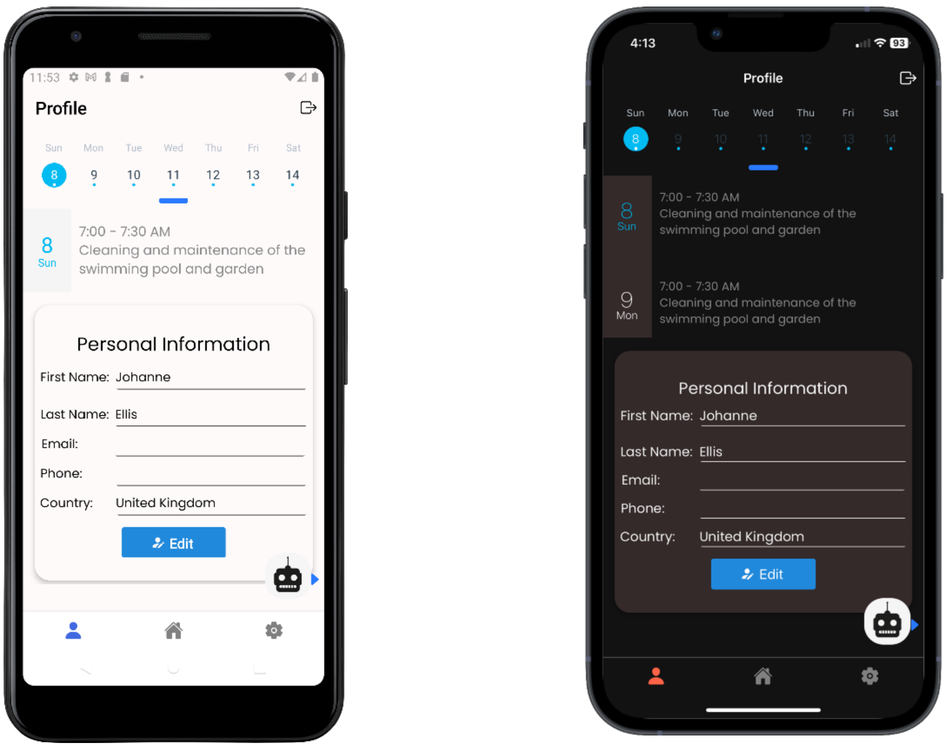
Figure 156 - About Us Screen

9.3 Guest

A guest is an authenticated user of the application who has made a booking. Upon arrival, the guest receives login credentials, granting access to additional advanced features of the application, including the tab screens available in the “authenticated” stack described in section "9.1.8 App".

9.3.1 Guest Profile

The first screen displayed to a guest user after logging in is the profile page, which includes details about their stay and some personal information that can be edited. At the top of the screen, there is a calendar-agenda showing all events scheduled during the user's stay, such as arrival, departure and house cleaning. The second section features a card with personal details that the user can modify. In the bottom right corner, a virtual assistant is available, clicking it redirects the user to another screen, where they can connect with an agent to address frequently asked questions.


11:53

Profile

Sun Mon Tue Wed Thu Fri Sat
8 9 10 11 12 13 14

8 Sun — 7:00 - 7:30 AM Cleaning and maintenance of the swimming pool and garden

Personal Information

First Name: Johanne
Last Name: Ellis
Email:
Phone:
Country: United Kingdom

Edit

4:13

Profile

Sun Mon Tue Wed Thu Fri Sat
8 9 10 11 12 13 14

8 Sun — 7:00 - 7:30 AM Cleaning and maintenance of the swimming pool and garden

9 Mon — 7:00 - 7:30 AM Cleaning and maintenance of the swimming pool and garden

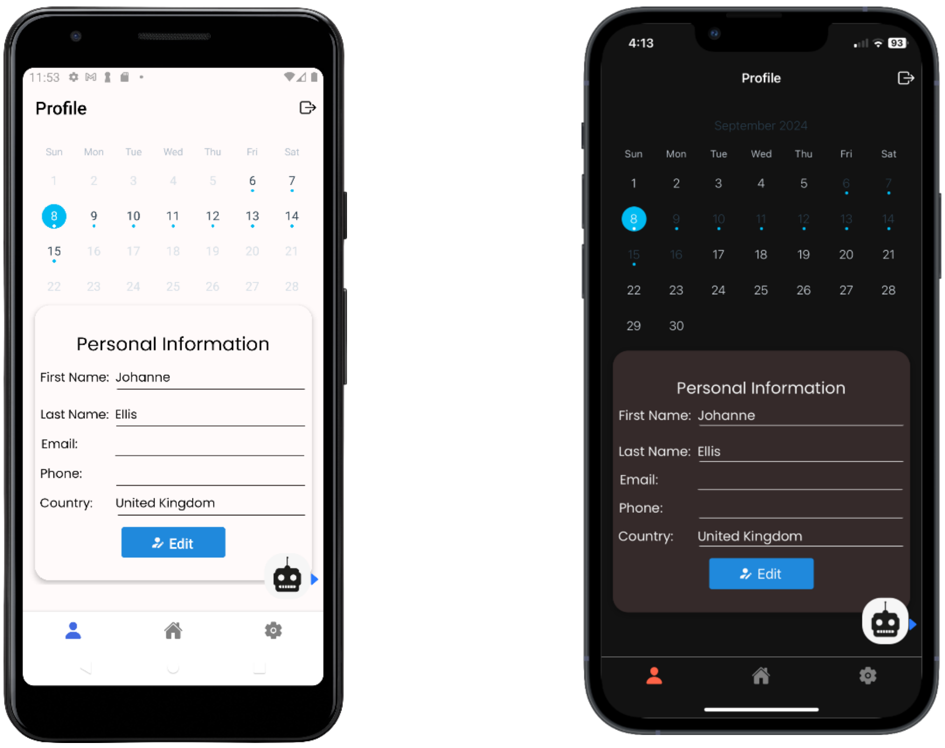
Personal Information

First Name: Johanne
Last Name: Ellis
Email:
Phone:
Country: United Kingdom

Edit

Figure 158 - Guest Profile

The user can expand the calendar to view only the days of their stay, without the agenda details. By clicking on a specific day, the agenda will reopen, displaying detailed information for that particular day.


11:53

Profile

Sun Mon Tue Wed Thu Fri Sat
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28

Personal Information

First Name: Johanne
Last Name: Ellis
Email:
Phone:
Country: United Kingdom

Edit

4:13

Profile

September 2024

Sun Mon Tue Wed Thu Fri Sat
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

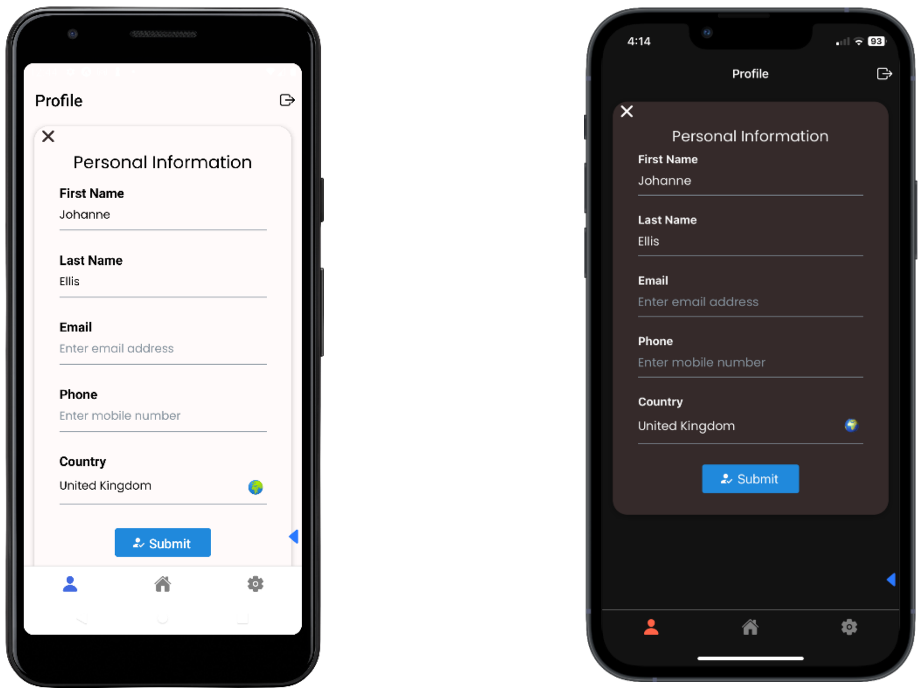
Personal Information

First Name: Johanne
Last Name: Ellis
Email:
Phone:
Country: United Kingdom

Edit

Figure 157 - Agenda

When the user presses the "Edit" button on the card, the calendar will be hidden and the card will expand to take up the full length of the screen, providing more space for the user to easily edit their personal information.


Profile

Personal Information

First Name
Johanne

Last Name
Ellis

Email
Enter email address

Phone
Enter mobile number

Country
United Kingdom

Submit

4:14

Profile

Personal Information

First Name
Johanne

Last Name
Ellis

Email
Enter email address

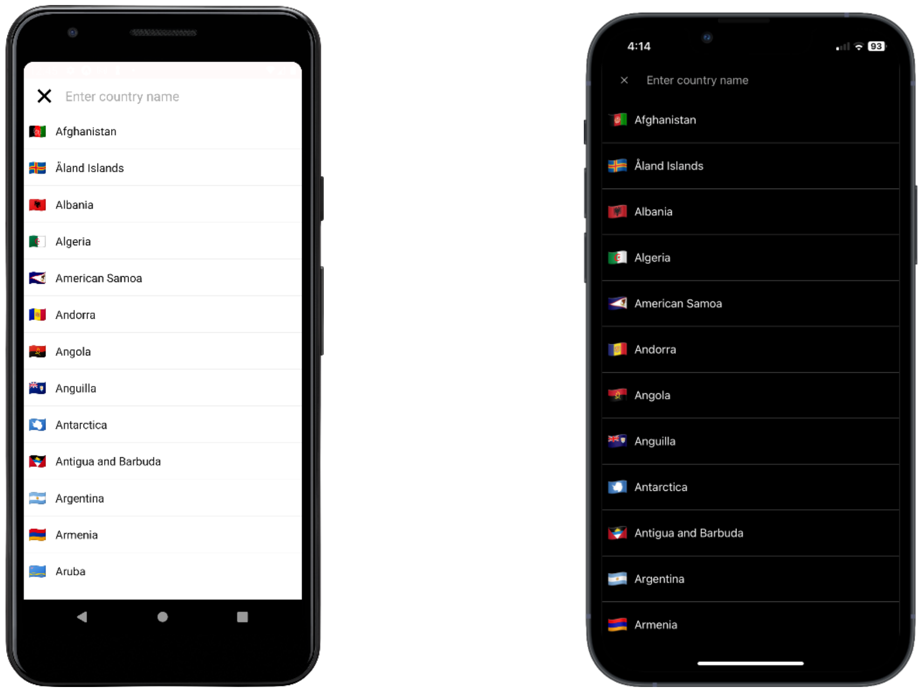
Phone
Enter mobile number

Country
United Kingdom

Submit

Figure 159 - Personal Information

For the "Country" field, the user should click the globe icon to the right of the field, which opens a new screen displaying a list of all available countries. This screen includes a search filter, allowing the user to quickly find and select a specific country. The "react-native-country-picker-modal" package was used to implement this feature.


Enter country name

Afghanistan
Åland Islands
Albania
Algeria
American Samoa
Andorra
Angola
Anguilla
Antarctica
Antigua and Barbuda
Argentina
Armenia
Aruba

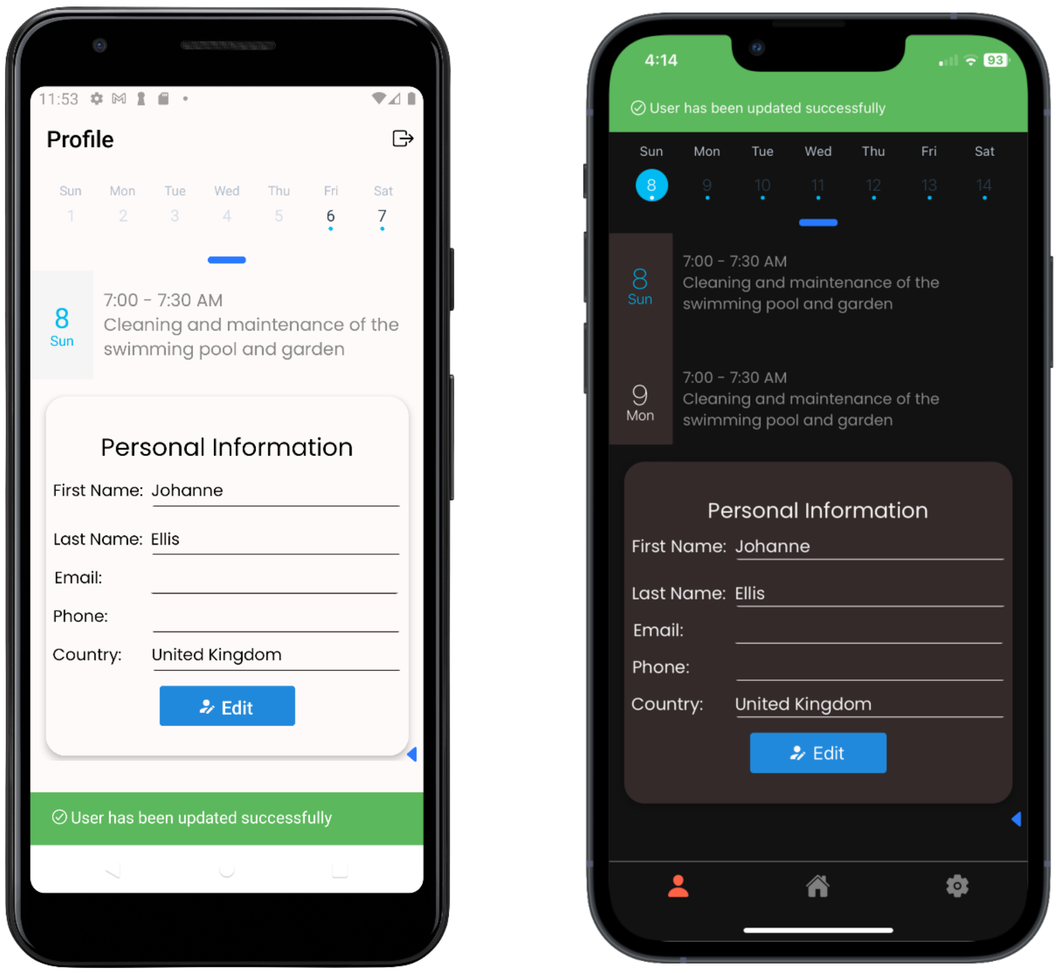
4:14

Enter country name

Afghanistan
Åland Islands
Albania
Algeria
American Samoa
Andorra
Angola
Anguilla
Antarctica
Antigua and Barbuda
Argentina
Armenia

Figure 160 - Country Picker

The user can either close the card by clicking the "Close" button in the top left corner or submit their changes by clicking the "Submit" button. Upon submission, a notification will appear informing the user of the outcome of their request, indicating whether it was successful or failed.


11:53

Profile

Sun 1, Mon 2, Tue 3, Wed 4, Thu 5, Fri 6, Sat 7

8 Sun — 7:00 - 7:30 AM Cleaning and maintenance of the swimming pool and garden

Personal Information

First Name: Johanne

Last Name: Ellis

Email:

Phone:

Country: United Kingdom

Edit

User has been updated successfully

4:14

User has been updated successfully

Sun 8, Mon 9, Tue 10, Wed 11, Thu 12, Fri 13, Sat 14

8 Sun — 7:00 - 7:30 AM Cleaning and maintenance of the swimming pool and garden

9 Mon — 7:00 - 7:30 AM Cleaning and maintenance of the swimming pool and garden

Personal Information

First Name: Johanne

Last Name: Ellis

Email:

Phone:

Country: United Kingdom

Edit

Figure 161 - Edit User

9.3.2 My Home

This is the second screen available to a guest, providing detailed information about the house that is not accessible to unauthenticated users. It includes a 2D/3D map of the house, allowing the user to switch between these two views. Additionally, there are detailed descriptions of each room and its contents. A report form is also available for the guest to report an issue or ask a question. Furthermore, a card displays the host's contact information, such as their email address and mobile phone number. As with the profile page, a virtual assistant is accessible from this screen as well.

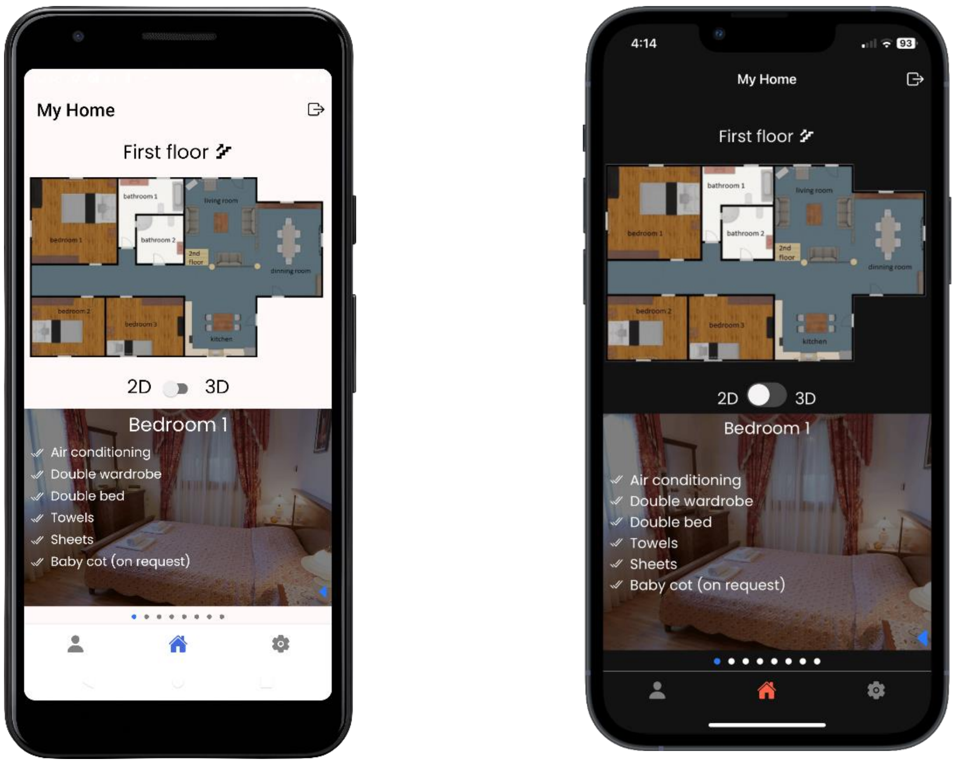


Figure 162 - My Home

By default, the 2D map of the property is selected and the user can switch to the 3D view by clicking the toggle button located beneath the map.

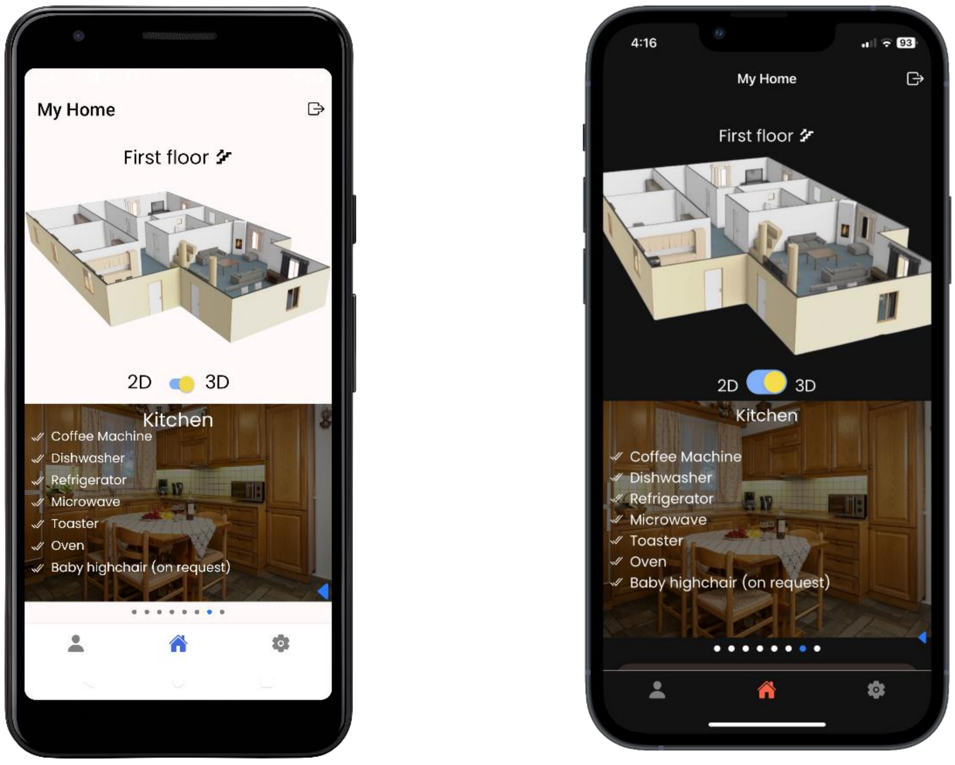
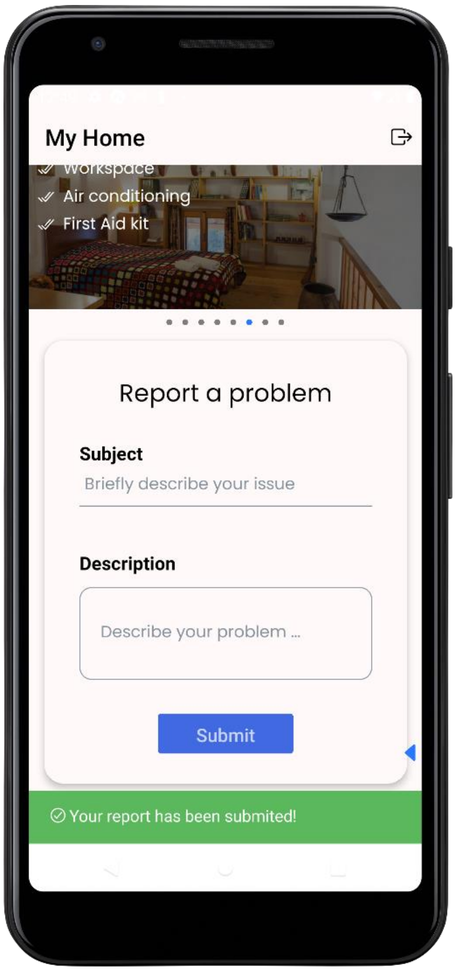


Figure 163 - 3D Map

The 3D model of the house is not an actual embedded 3D model within the app, but rather an illusion created using multiple screenshots of the house model built on the online platform "Floorplanner" [28]. After the model was created, several screenshots were taken and imported into the project as assets. A custom component called "image-360-viewer" is used to provide a 3D-like experience, allowing the user to scroll left or right to view the house from different angles, simulating a 3D effect.

Following on this screen, the report form is consisted of two fields:

- Subject: A single line text input field
- Details: A multi-line text area



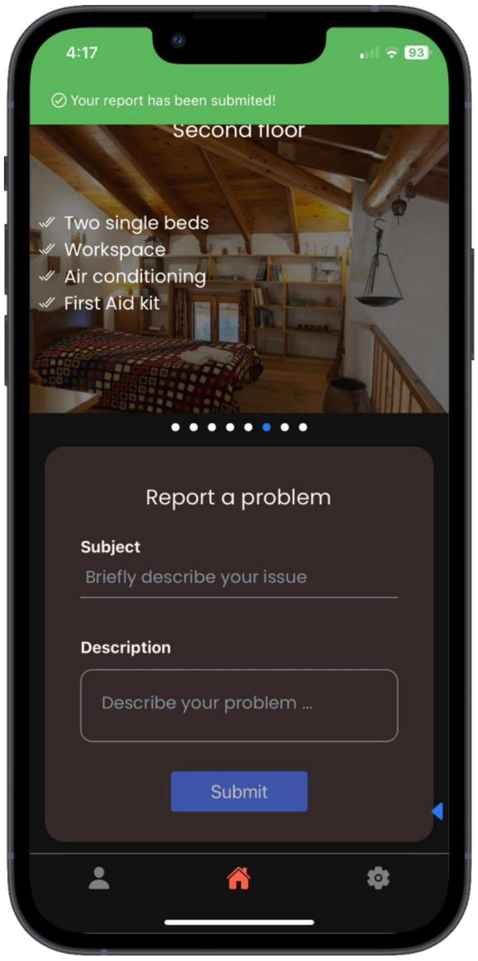
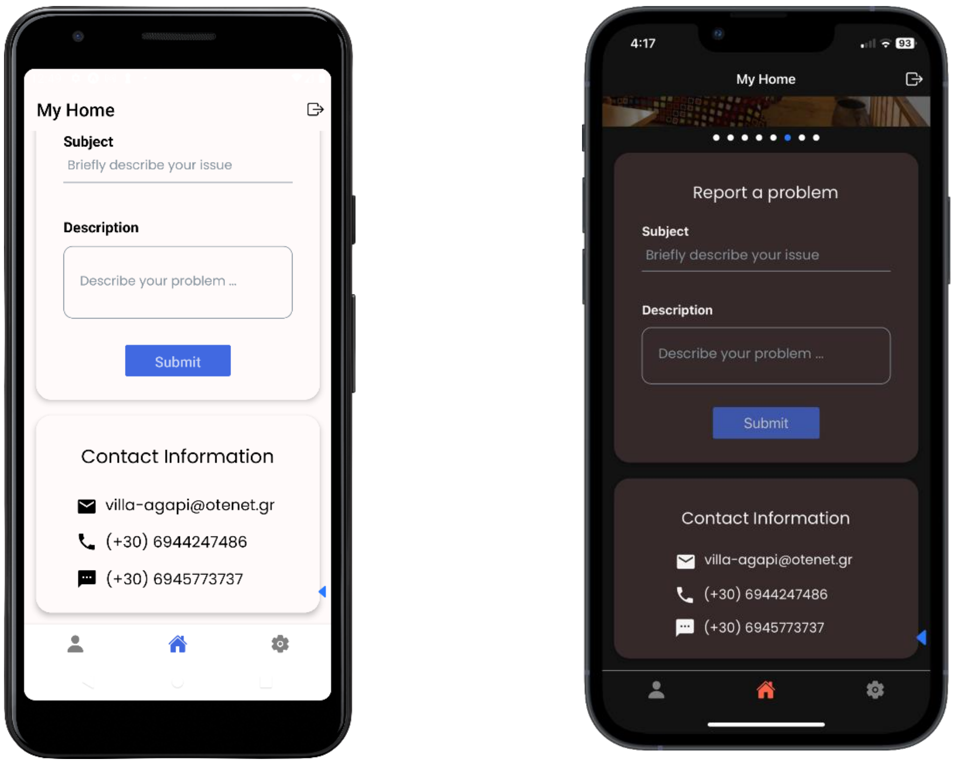


Figure 164 - Report Form

Both fields are mandatory and the "Submit" button remains disabled until both fields are filled out. If the form is valid and the user presses the "Submit" button, a notification will appear, confirming the submission of their report.

The final card of this screen is the Contact Information.



villa-agapi@otenet.gr
(+30) 6944247486
(+30) 6945773737

Figure 165 - Contact Information

The host's email address and mobile number are available in this section, with the following functionalities implemented:

- Copy to Clipboard: The user can click on either the mobile number or the email address and it will be automatically copied to the clipboard, allowing them to easily paste it elsewhere.
- Interactive Icons: The user can click each icon and the following activity will take place:
    - Email icon: Opens the user's default email application with the host's email address pre-filled in the recipient field

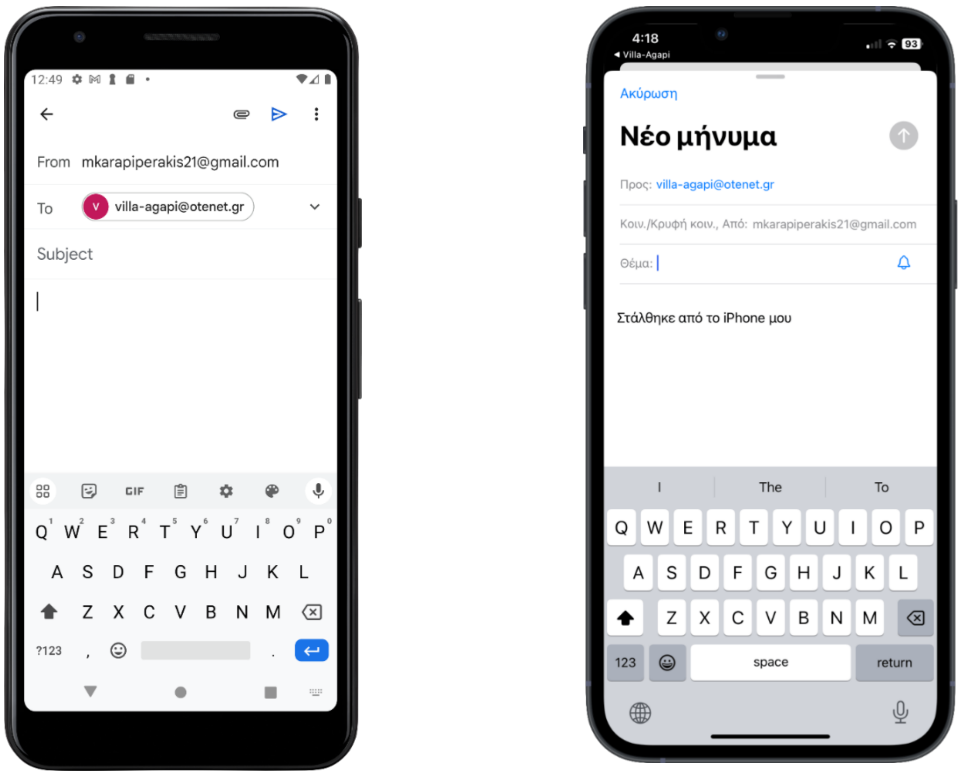

12:49
From mkarapiperakis21@gmail.com
To villa-agapi@otenet.gr
Subject
4:18
Villa-Agapi
Ακύρωση
Νέο μήνυμα
Προς: villa-agapi@otenet.gr
Κοιν./Κρυφή κοιν., Από: mkarapiperakis21@gmail.com
Θέμα:
Στάλθηκε από το iPhone μου

Figure 166 - Send Email

- Mobile Phone icon: For Android devices, this icon opens the user's phone interface with the host’s number pre-filled. On iOS, it triggers a pop-up window at the bottom of the screen within the app:

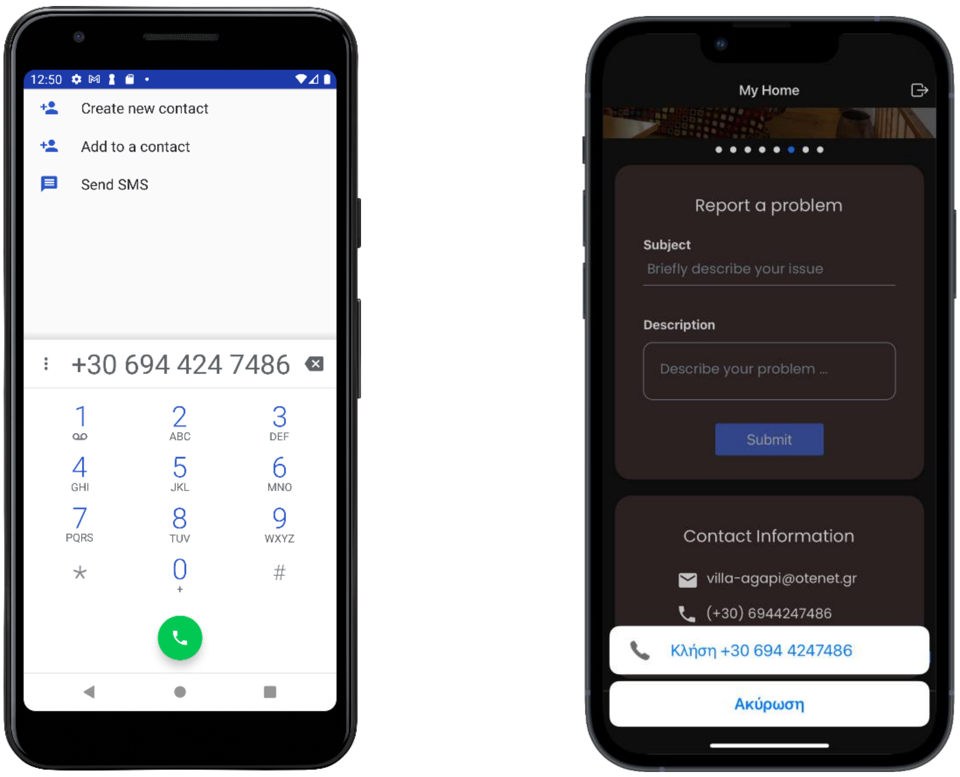
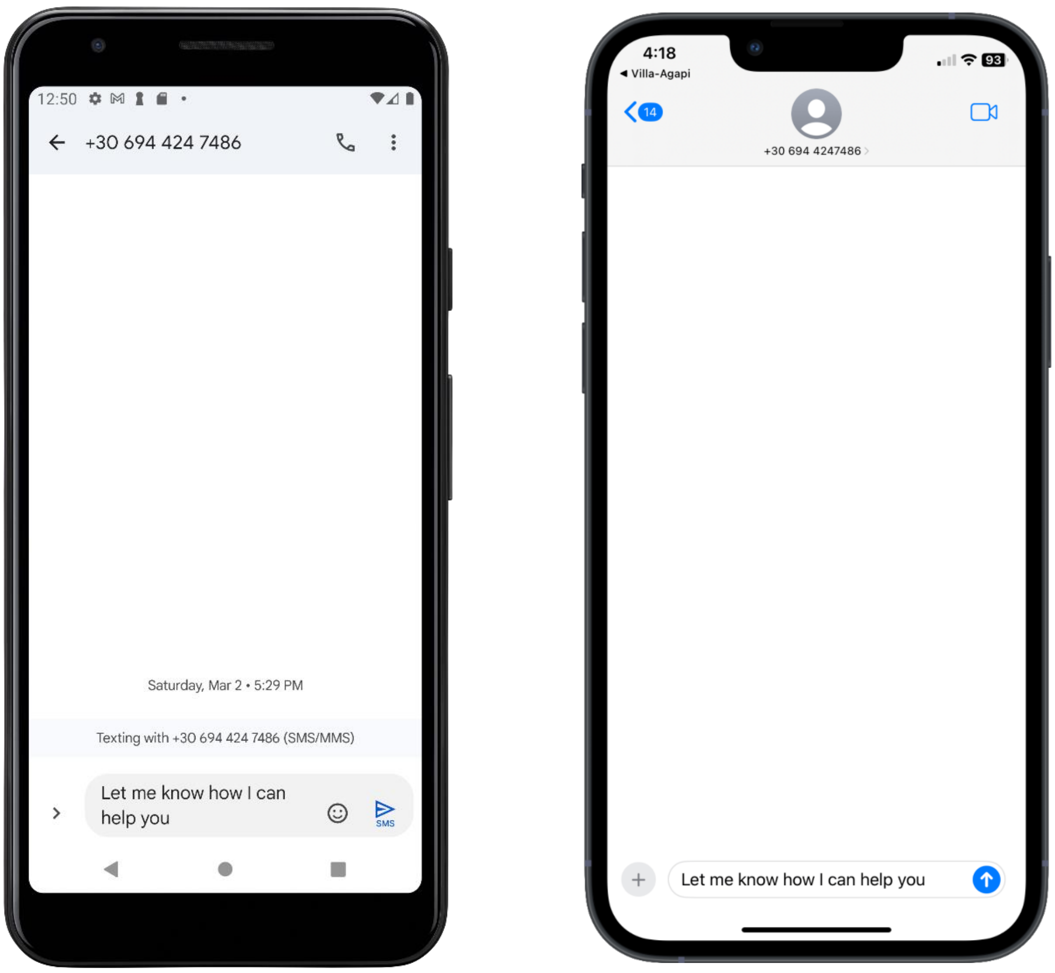


Figure 167 - Phone Call

- SMS icon: On both Android and iOS devices, this icon opens the default SMS service with the host's number as the recipient and a pre-populated message:

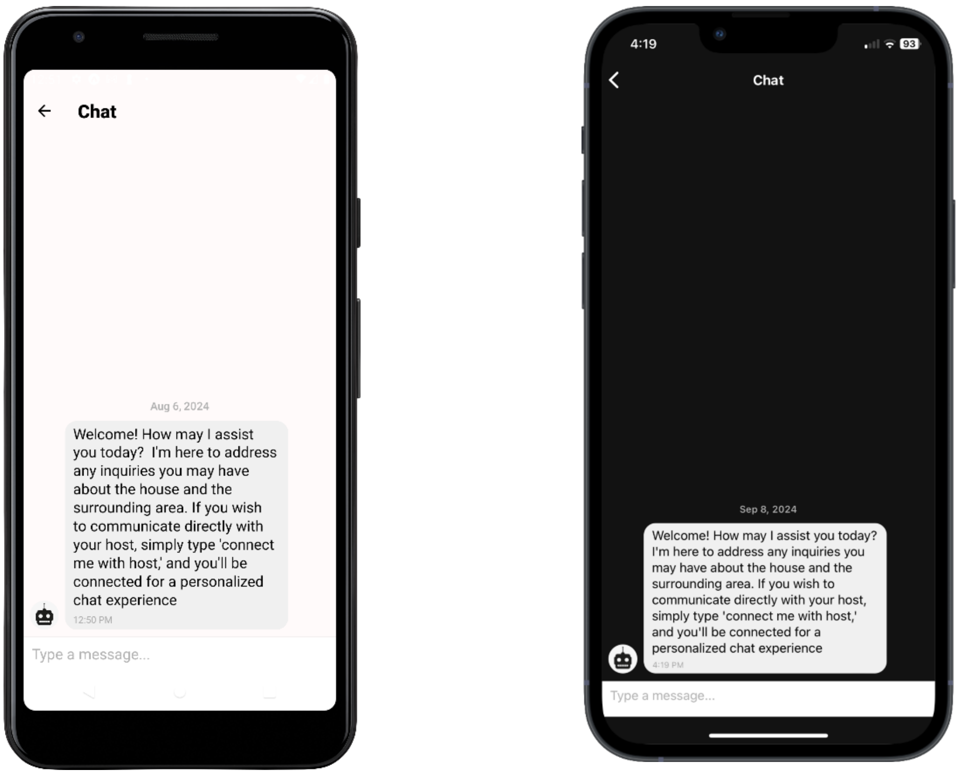

12:50
+30 694 424 7486
Saturday, Mar 2 • 5:29 PM
Texting with +30 694 424 7486 (SMS/MMS)
Let me know how I can help you

4:18
◂ Villa-Agapi
+30 694 4247486
Let me know how I can help you

Figure 168 - Send SMS

9.3.3 Chatbot

The chatbot screen is the final interface available to authenticated users, accessible from the two previously described screens. Upon entering this screen, the chatbot automatically sends a welcome message to greet the user and provide an overview of the available services


Chat

Aug 6, 2024

Welcome! How may I assist you today? I'm here to address any inquiries you may have about the house and the surrounding area. If you wish to communicate directly with your host, simply type 'connect me with host,' and you'll be connected for a personalized chat experience
12:50 PM

Type a message...

4:19

Chat

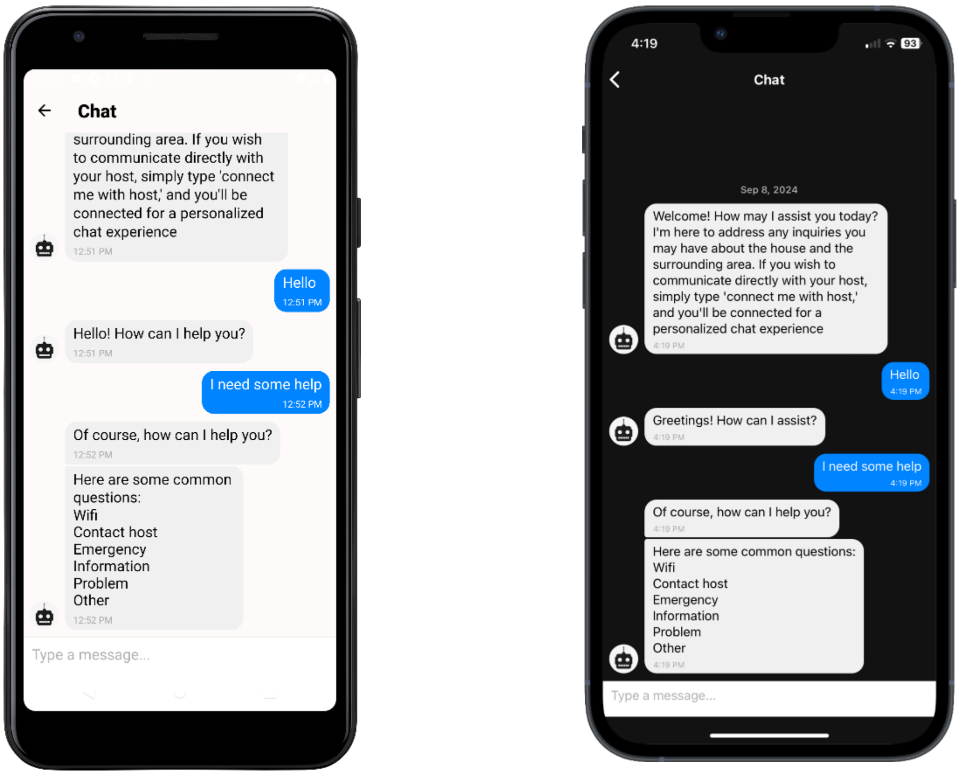
Sep 8, 2024

Welcome! How may I assist you today? I'm here to address any inquiries you may have about the house and the surrounding area. If you wish to communicate directly with your host, simply type 'connect me with host,' and you'll be connected for a personalized chat experience
4:19 PM

Type a message...

Figure 169 - Chatbot Welcome Message

The chatbot can respond to common phrases, but its primary function is to guide users toward 'standard' questions related to the property and the surrounding area:


Chat

surrounding area. If you wish to communicate directly with your host, simply type 'connect me with host,' and you'll be connected for a personalized chat experience
12:51 PM

Hello
12:51 PM

Hello! How can I help you?
12:51 PM

I need some help
12:52 PM

Of course, how can I help you?
12:52 PM

Here are some common questions:
Wifi
Contact host
Emergency
Information
Problem
Other
12:52 PM

Type a message...

4:19

Chat

Sep 8, 2024

Welcome! How may I assist you today? I'm here to address any inquiries you may have about the house and the surrounding area. If you wish to communicate directly with your host, simply type 'connect me with host,' and you'll be connected for a personalized chat experience
4:19 PM

Hello
4:19 PM

Greetings! How can I assist?
4:19 PM

I need some help
4:19 PM

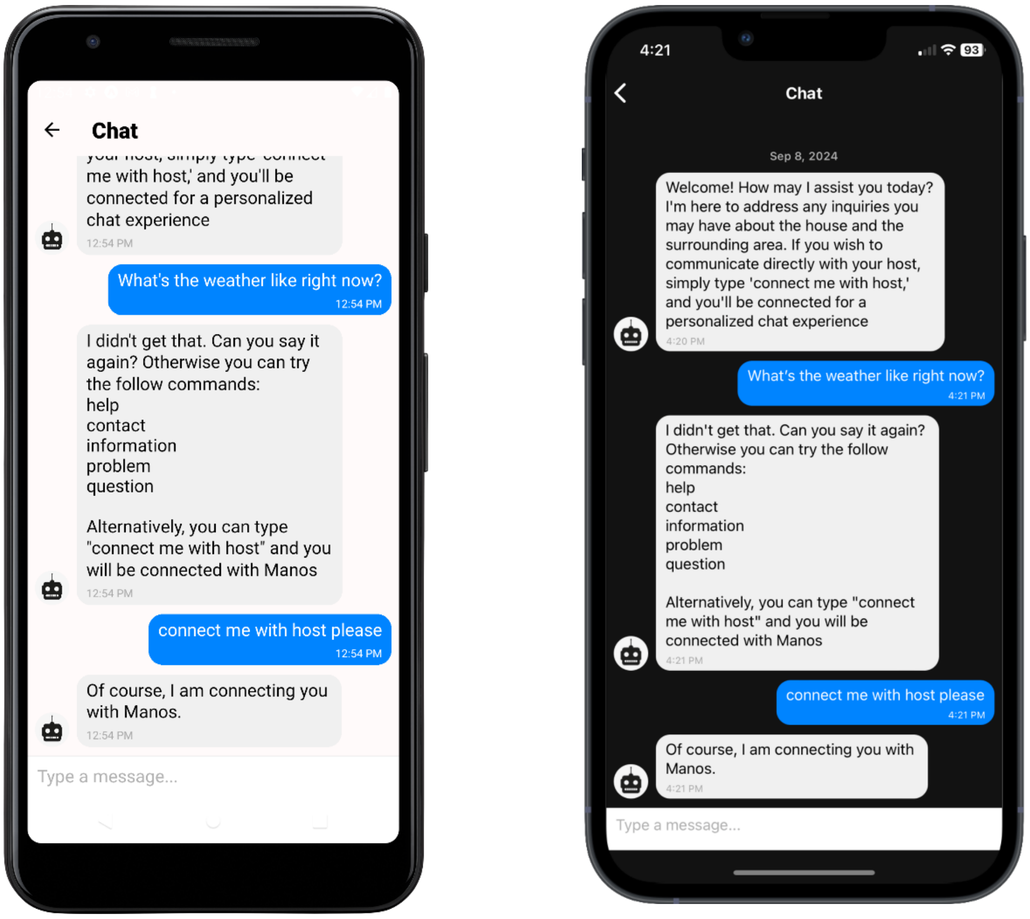
Of course, how can I help you?
4:19 PM

Here are some common questions:
Wifi
Contact host
Emergency
Information
Problem
Other
4:19 PM

Type a message...

Figure 170 - Chatbot Messages

If the user sends a message that the chatbot cannot respond to, a default message will be returned, prompting the user to try again. If they are not satisfied, the message will also suggest connecting with a live agent through the live chat feature:


Chat

me with host,' and you'll be connected for a personalized chat experience
12:54 PM

What's the weather like right now?
12:54 PM

I didn't get that. Can you say it again? Otherwise you can try the follow commands:
help
contact
information
problem
question

Alternatively, you can type "connect me with host" and you will be connected with Manos
12:54 PM

connect me with host please
12:54 PM

Of course, I am connecting you with Manos.
12:54 PM

Type a message...

4:21

Chat

Sep 8, 2024

Welcome! How may I assist you today? I'm here to address any inquiries you may have about the house and the surrounding area. If you wish to communicate directly with your host, simply type 'connect me with host,' and you'll be connected for a personalized chat experience
4:20 PM

What's the weather like right now?
4:21 PM

I didn't get that. Can you say it again? Otherwise you can try the follow commands:
help
contact
information
problem
question

Alternatively, you can type "connect me with host" and you will be connected with Manos
4:21 PM

connect me with host please
4:21 PM

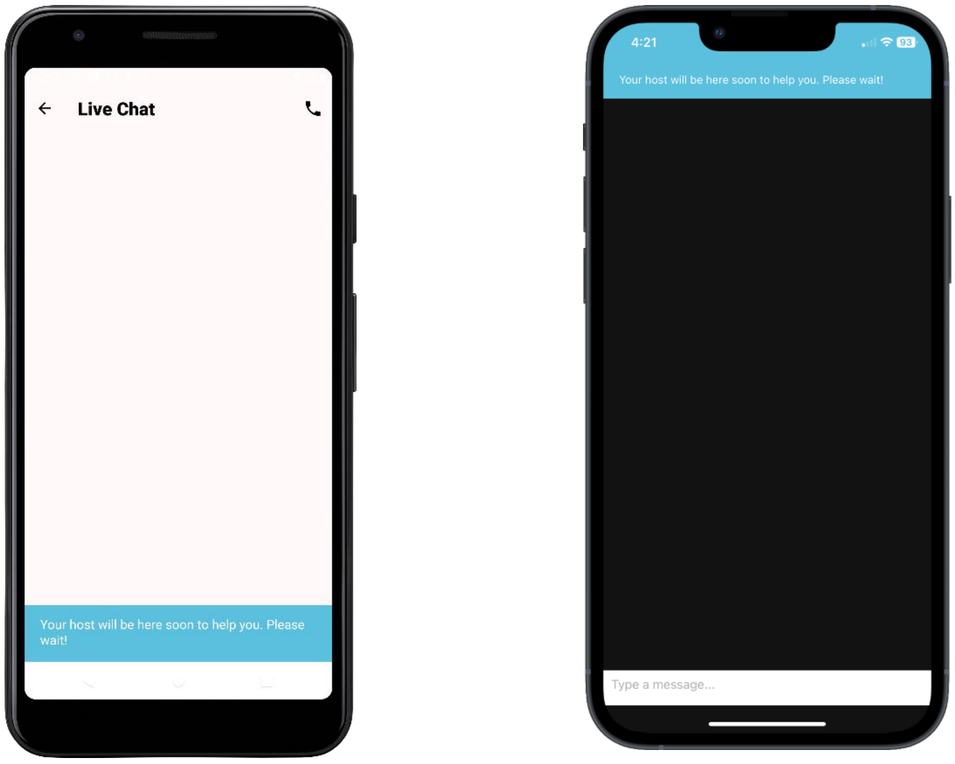
Of course, I am connecting you with Manos.
4:21 PM

Type a message...

Figure 171 - Connect With Host

9.3.4 Live Chat Guest

If the user types 'Connect me with host' or a similar phrase recognized by the chatbot, they will be redirected to the “Live Chat” screen to directly contact the host. The user will receive a notification informing them that the host has been notified and will join the chat shortly. Meanwhile, the host will receive an email notification, as outlined in section “8.3.3 Middlewares”.

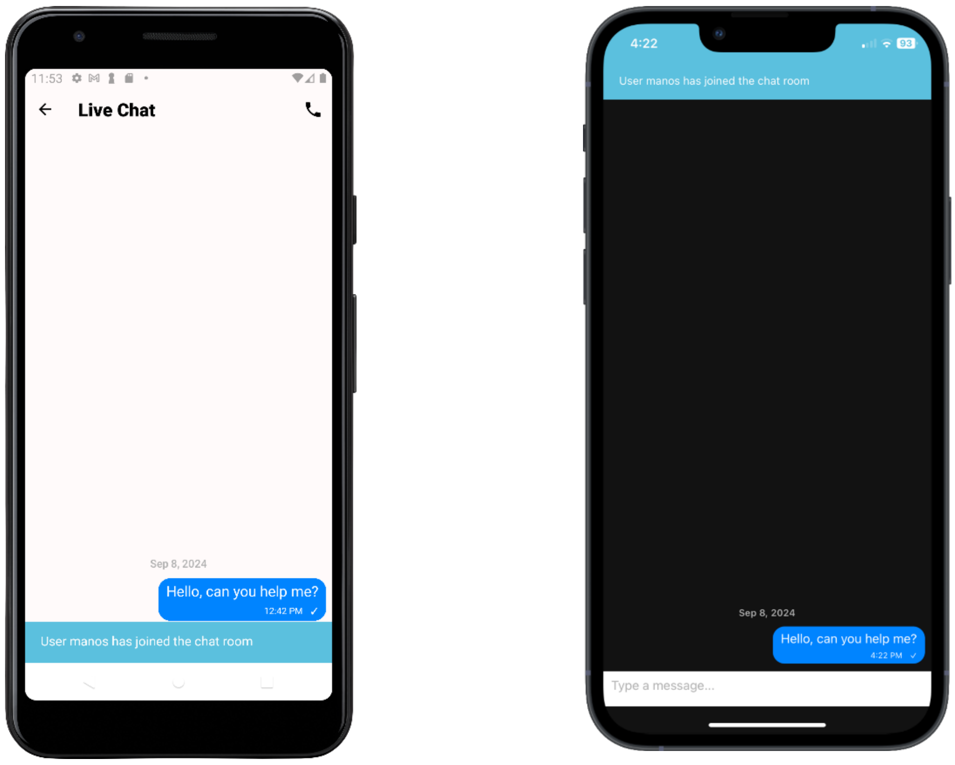


Your host will be here soon to help you. Please wait!

Your host will be here soon to help you. Please wait!

Figure 172 - Join Room

The user can either send a message immediately or wait for the host to join the chat room. A notification will be sent to the user when the host connects to the room:



Sep 8, 2024

Hello, can you help me?
12:42 PM ✓

User manos has joined the chat room

User manos has joined the chat room

Sep 8, 2024

Hello, can you help me?
4:22 PM ✓

Figure 173 - Live Chat Messages

When the host begins typing their message, the other user will be notified by the appearance of the characteristic 'three dots' indicator:

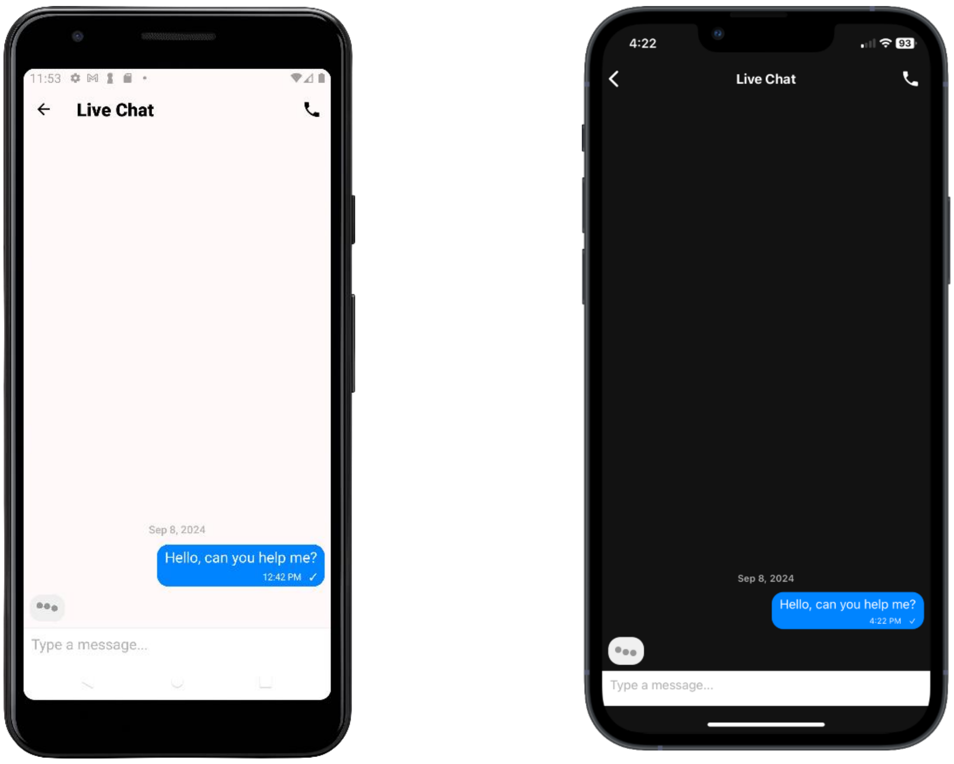


Figure 174 - Live Chat Typing

When the host responds, the other user will see the host's avatar alongside the message. Additionally, similar to the login notification, the user will receive a notification if the host leaves the room for any reason:

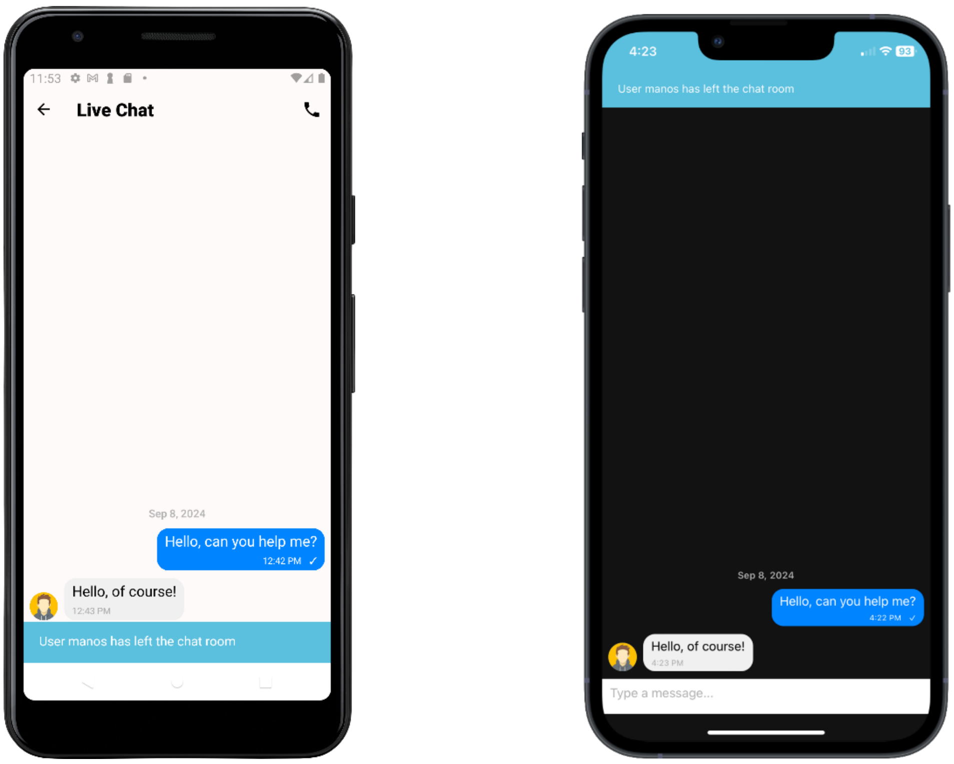


Figure 175 - Leave Room

9.4 Admin

Admin is an authenticated user of the application. This role is assigned to the property host, granting access to advanced features, including the admin-specific screens within the “authenticated” stack, as outlined in section “9.1.8 App”.

9.4.1 Admin Dashboard

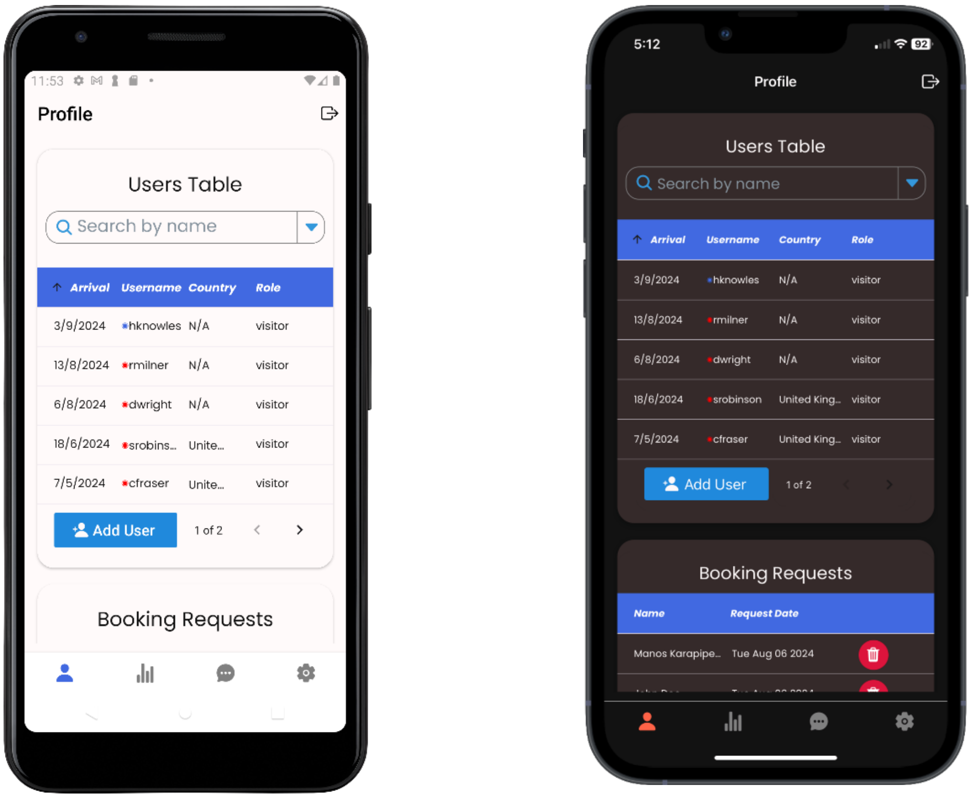
The Admin Dashboard is the first screen an admin sees after logging in. This panel enables the host to monitor user activity, manage users and booking requests, update property availability and control dynamic maps by adding or removing markers. The Admin Dashboard includes four cards, each for the following functionalities:

- User Management
- Booking Requests
- Availability
- Locations

9.4.1.1 User Management

The Users Table is the first card available in the Admin Dashboard and consists of the following components:

- Users Table: Displays a list of available users. Each row is clickable, navigating to a detailed screen for the selected user. The table includes pagination, showing five users per page.
- Search Bar: Allows filtering of users based on role, name or country.
- Add User Button: Enables the admin to add a new user.



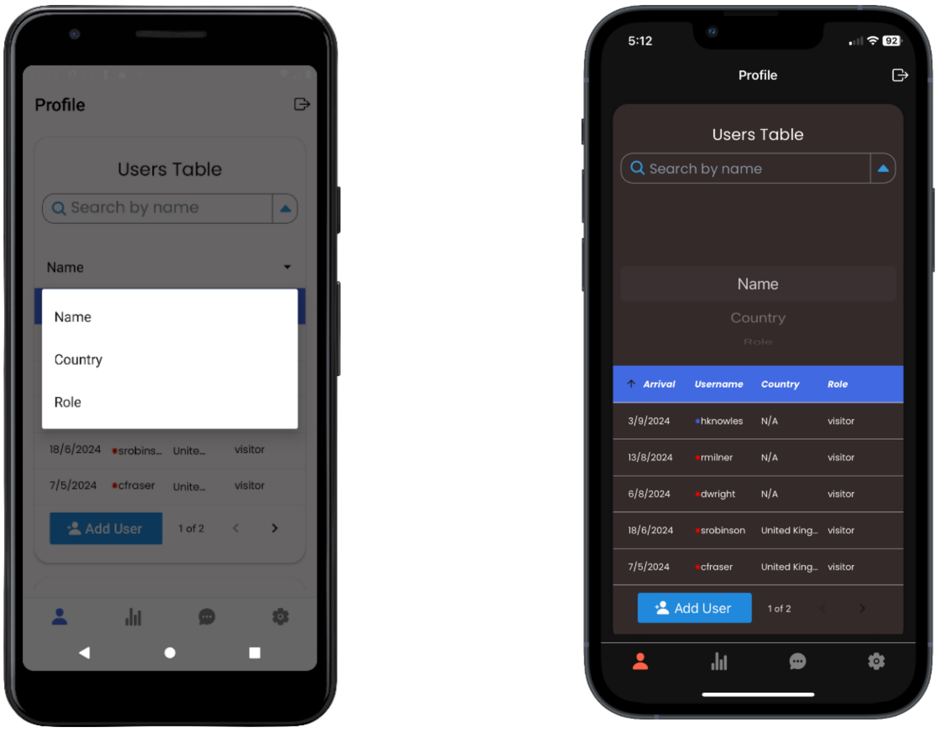
Arrival	Username	Country	Role
3/9/2024	hknowles	N/A	visitor
13/8/2024	rmilner	N/A	visitor
6/8/2024	dwright	N/A	visitor
18/6/2024	srobins...	Unite...	visitor
7/5/2024	cfraser	Unite...	visitor

Arrival	Username	Country	Role
3/9/2024	hknowles	N/A	visitor
13/8/2024	rmilner	N/A	visitor
6/8/2024	dwright	N/A	visitor
18/6/2024	srobinson	United King...	visitor
7/5/2024	cfraser	United King...	visitor

Name	Request Date
Manos Karapipe...	Tue Aug 06 2024

Figure 176 - Users Table

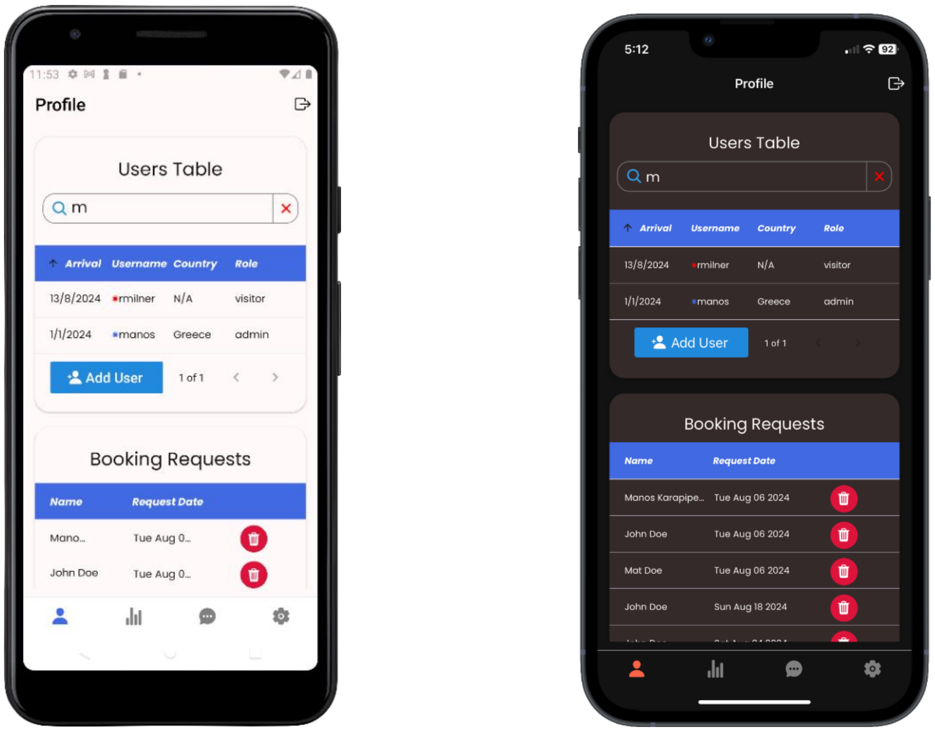
The user can click the down arrow at the end of the search input field to display a dropdown menu, allowing them to select the column they want to filter by.



Arrival	Username	Country	Role
3/9/2024	hknowles	N/A	visitor
13/8/2024	rmilner	N/A	visitor
6/8/2024	dwright	N/A	visitor
18/6/2024	srobinson	United King...	visitor
7/5/2024	cfraser	United King...	visitor

Figure 177 - Users Table Filters

Once the user selects a column and starts typing, the results will appear in real time, without the need to submit anything.

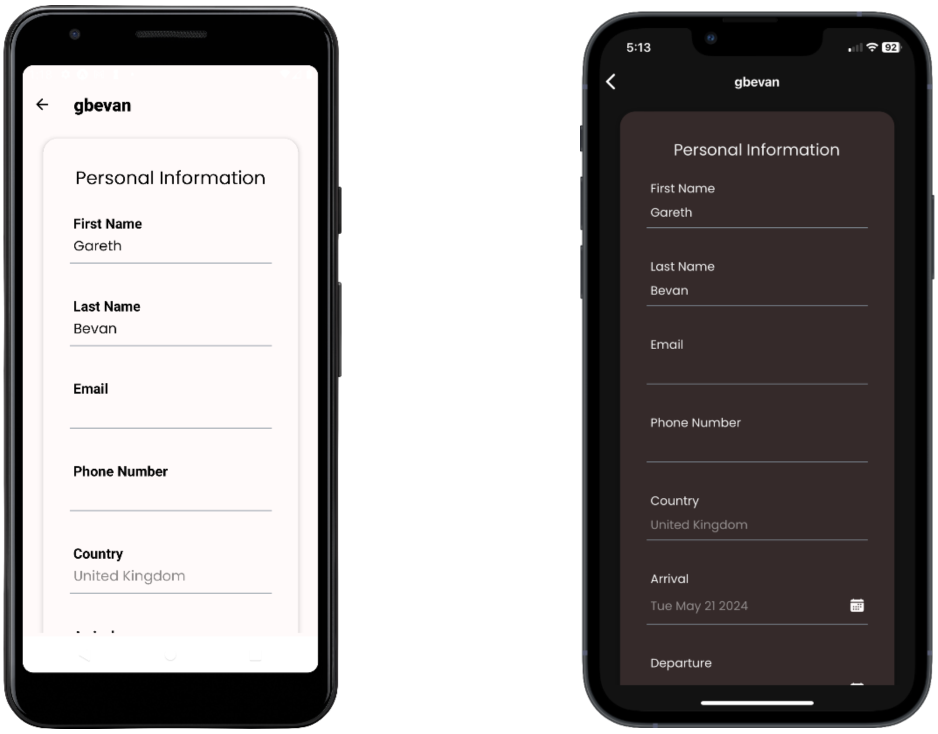


Arrival	Username	Country	Role
13/8/2024	rmilner	N/A	visitor
1/1/2024	manos	Greece	admin

Name	Request Date
Manos Karapipe...	Tue Aug 06 2024
John Doe	Tue Aug 06 2024
Mat Doe	Tue Aug 06 2024
John Doe	Sun Aug 18 2024

Figure 178 - Filter Results

When the admin clicks on a row, they will be taken to a new screen where they can view details about the selected user and make modifications.


gbevan

Personal Information

First Name
Gareth

Last Name
Bevan

Email

Phone Number

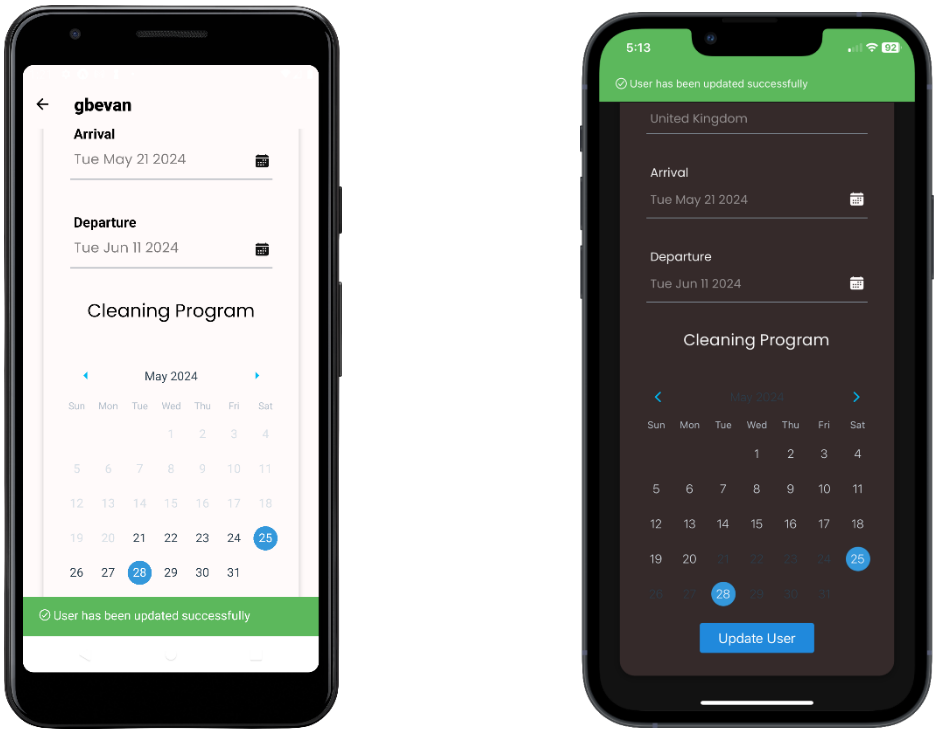
Country
United Kingdom

Arrival
Tue May 21 2024

Departure

Figure 179 - Edit User

Some fields, such as the user's phone number and country, are read-only since this information should be provided by the guest. When the admin edits any other fields and presses the "Update" button, a notification message will appear, informing them of the result of the update.


gbevan

Arrival
Tue May 21 2024

Departure
Tue Jun 11 2024

Cleaning Program

May 2024

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

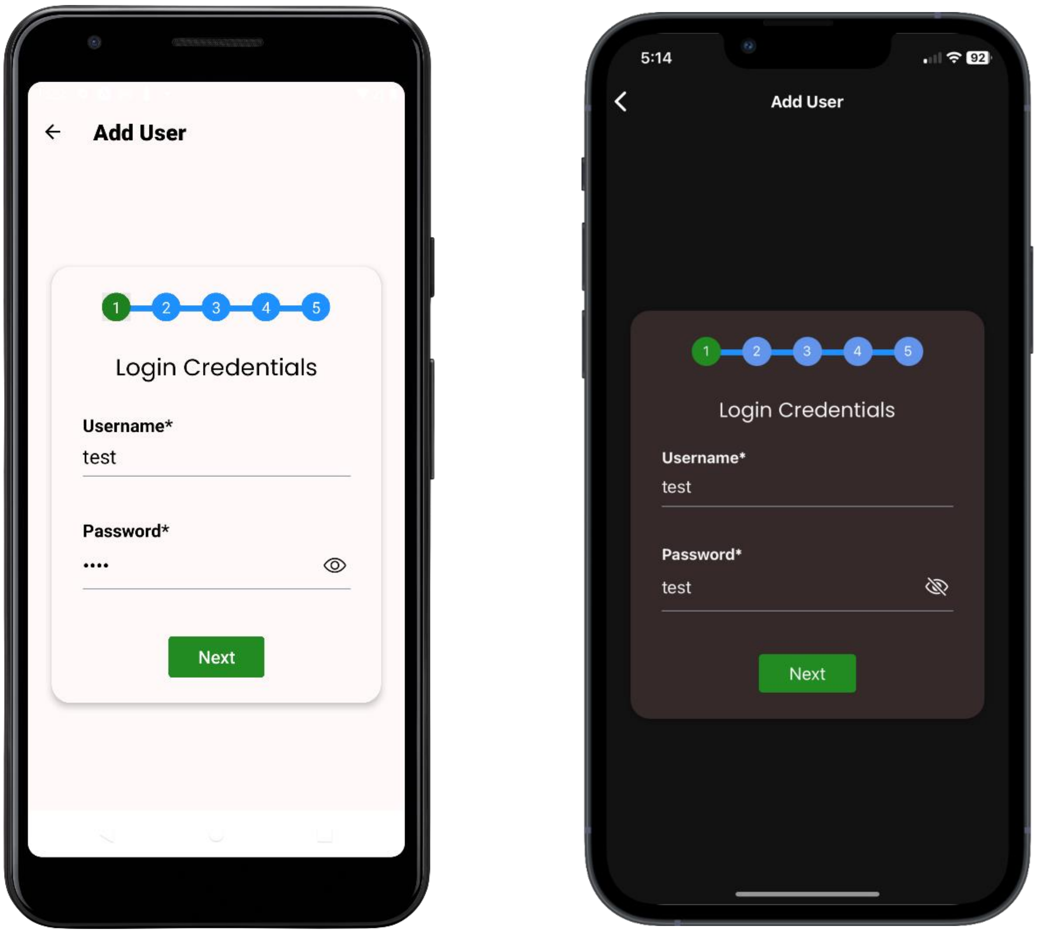
User has been updated successfully

Country
United Kingdom

Figure 180 - Update User

By clicking the option to add a new user, the admin will navigate to a new screen where the user can be added through five steps:

1. Login Credentials: Enter the credentials required for the user to log in.
2. Personal Information: Provide the first and last name.
3. Contact Information: Input the email address and phone number.
4. Arrival & Departure: Specify the arrival and departure dates.
5. Cleaning Program: Select the cleaning program for the guest's stay.



Login Credentials

Username*
test

Password*
••••

Login Credentials

Username*
test

Password*
test

Figure 181 - Add User 1st step

The login credentials fields (Username and Password) are mandatory, meaning the "Next" button to proceed to the next step is disabled if either field is empty. For the Password field, an icon is located on the right side, allowing the user to show or hide the typed password.

The next step is Personal Information, where the fields are optional, as this information may not always be available to the host from the booking. Therefore, the "Next" button will be enabled by default. The user can return to the previous step by simply clicking the number “1” in the progress bar.

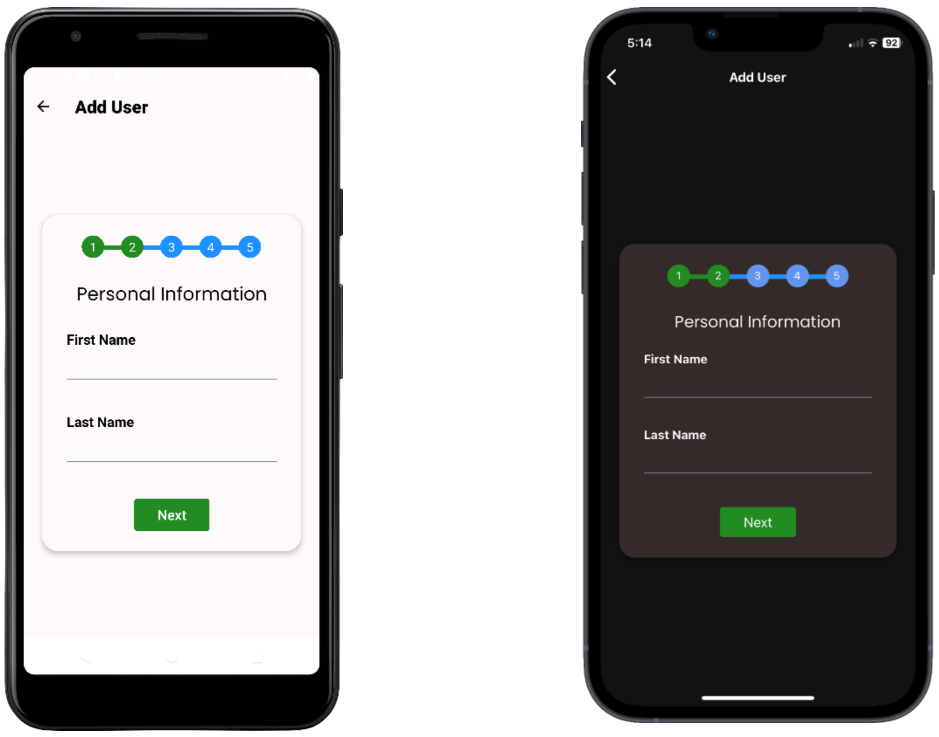


Figure 182 - Add User 2nd step

The following step includes Contact Information, which consists of the Email and Phone Number fields. These fields are optional and are only available in special cases when the guest wishes to communicate with the host prior to arrival.

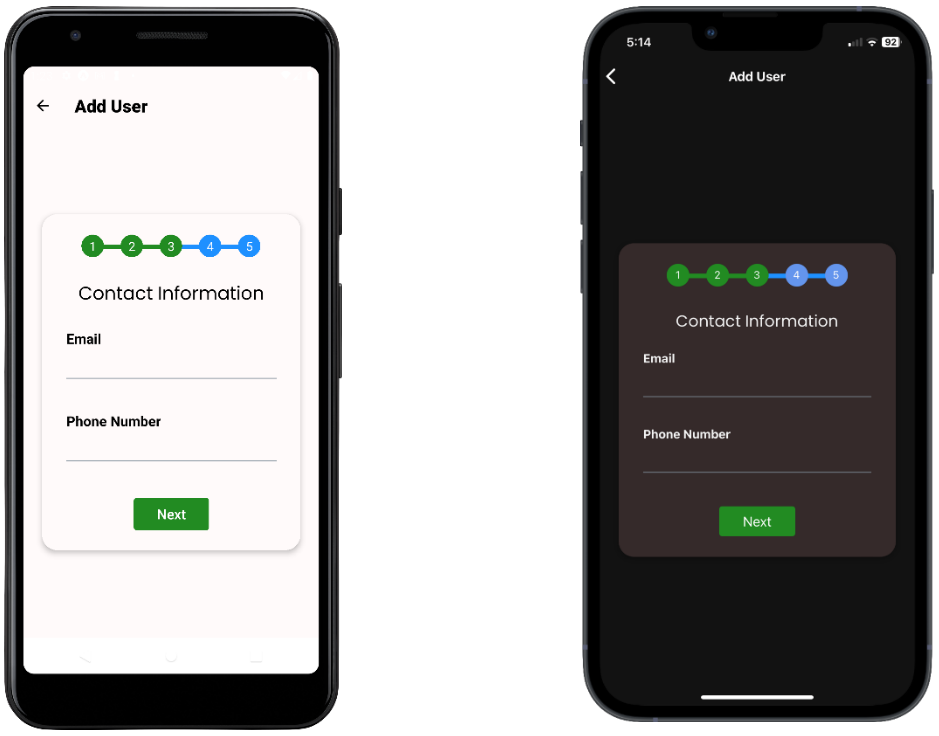
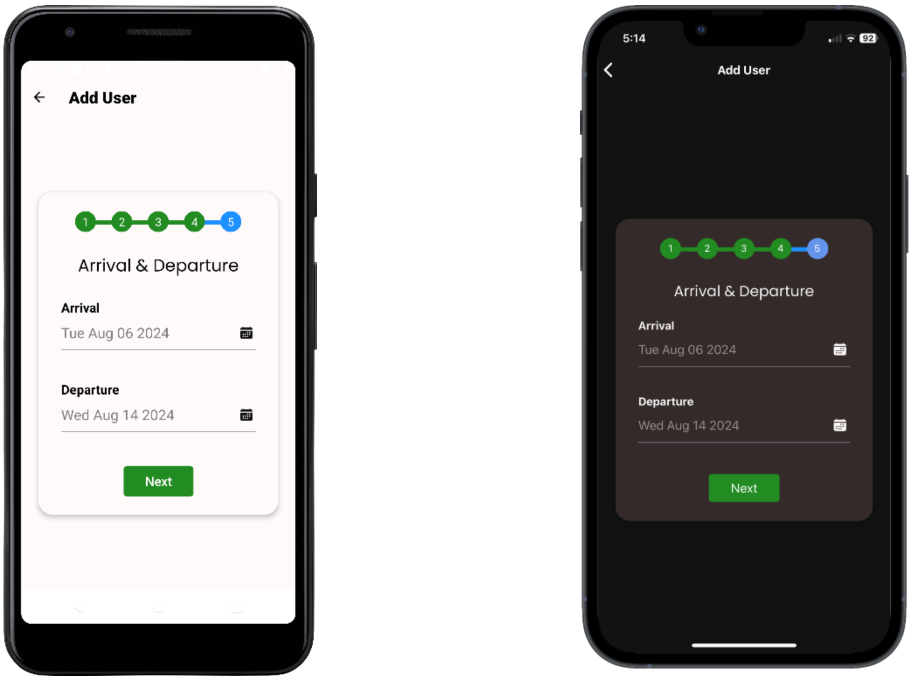


Figure 183 - Add User 3rd step

The fourth step includes Arrival and Departure Dates, which are required fields. As a result, the "Next" button will remain disabled until both dates are valid and filled in.



Arrival
Tue Aug 06 2024

Departure
Wed Aug 14 2024

Figure 184 - Add User 4th step

The final step involves selecting the Cleaning Program for the house. Typically, the host cleans the interior of the house twice a week and these dates are available from the guest's agenda. The host will see a calendar displaying the minimum and maximum dates based on the arrival and departure dates set in the previous step. Between these dates, the host will be able to select the cleaning dates for the house.

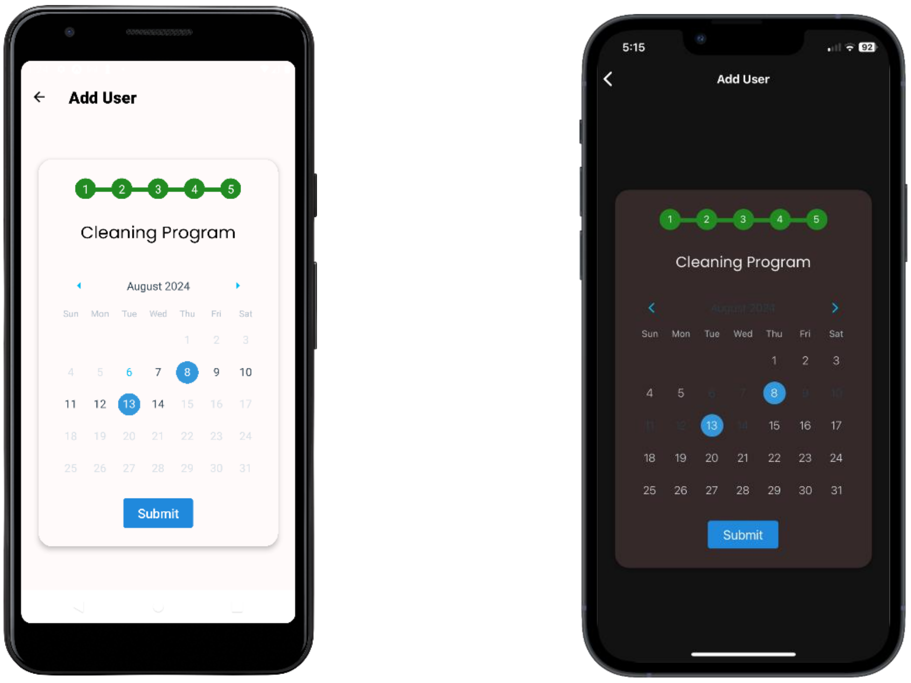
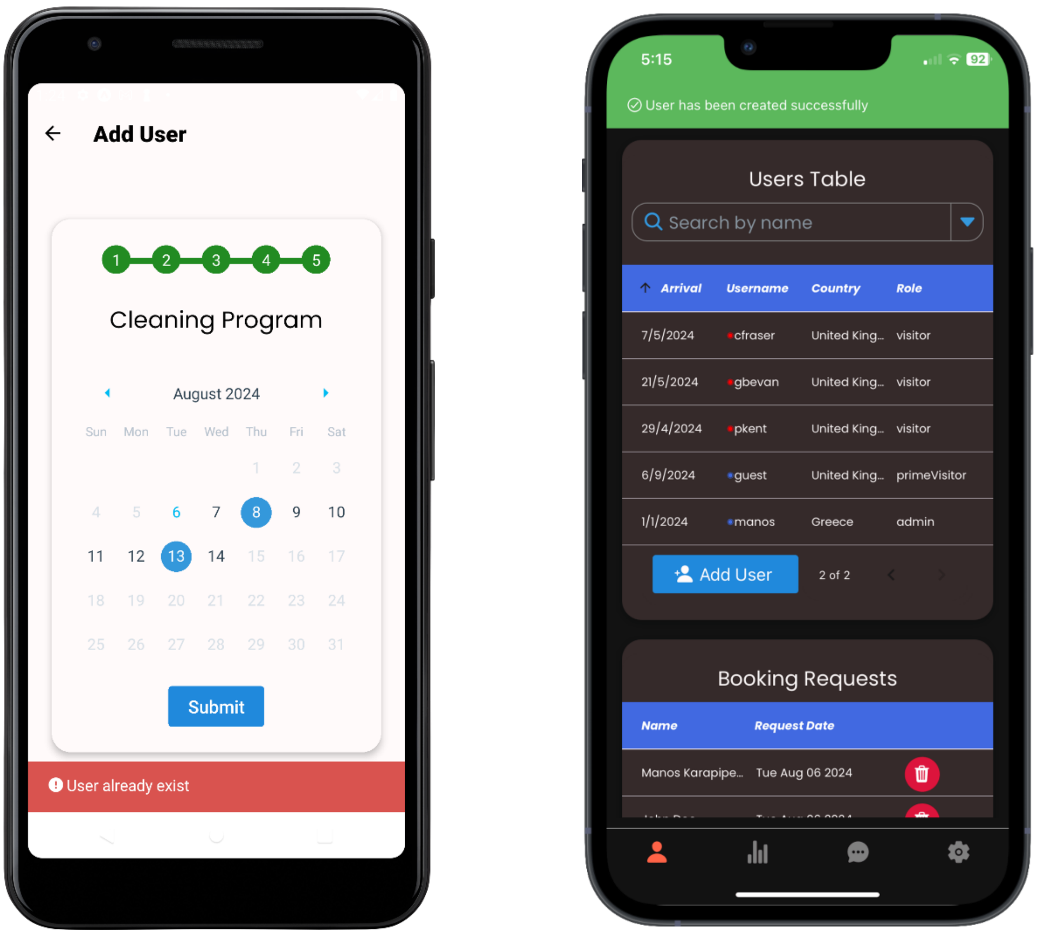


Figure 185 - Add User 5th step

If the addition of the new user is successful, the host will be redirected back to the Admin Dashboard and a success notification will pop up. If an error occurs, the host will remain on the final step and an error notification will appear, informing them of the issue.



Cleaning Program

August 2024

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

User already exist

User has been created successfully

Users Table

Search by name

Arrival	Username	Country	Role
7/5/2024	cfraser	United King...	visitor
21/5/2024	gbevan	United King...	visitor
29/4/2024	pkent	United King...	visitor
6/9/2024	guest	United King...	primeVisitor
1/1/2024	manos	Greece	admin

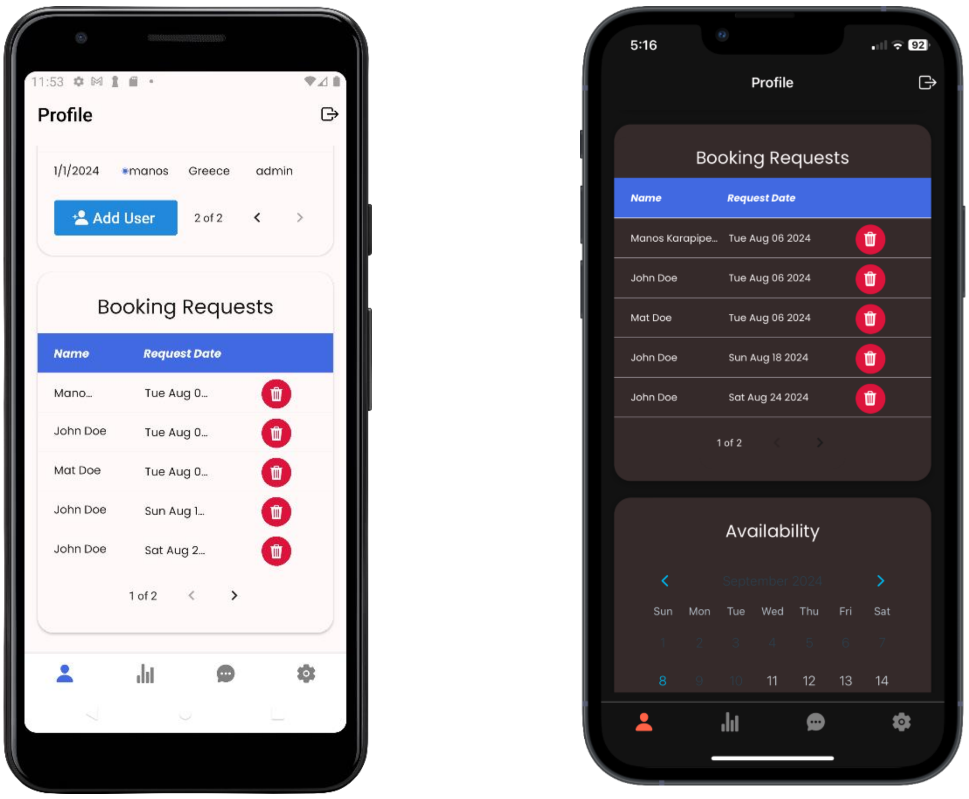
Booking Requests

Name	Request Date
Manos Karapipe...	Tue Aug 06 2024

Figure 186 - Add User Submission

9.4.1.2 Booking Requests

The next available card is the Booking Requests, where the host can navigate through requests using a table similar to the Users Table. The host can select a booking request to review its details or click the “bin” icon to permanently delete it.


Profile

1/1/2024	manos	Greece	admin

Add User 2 of 2

Booking Requests

Name	Request Date
Mano...	Tue Aug 0...
John Doe	Tue Aug 0...
Mat Doe	Tue Aug 0...
John Doe	Sun Aug 1...
John Doe	Sat Aug 2...

1 of 2

Profile

Booking Requests

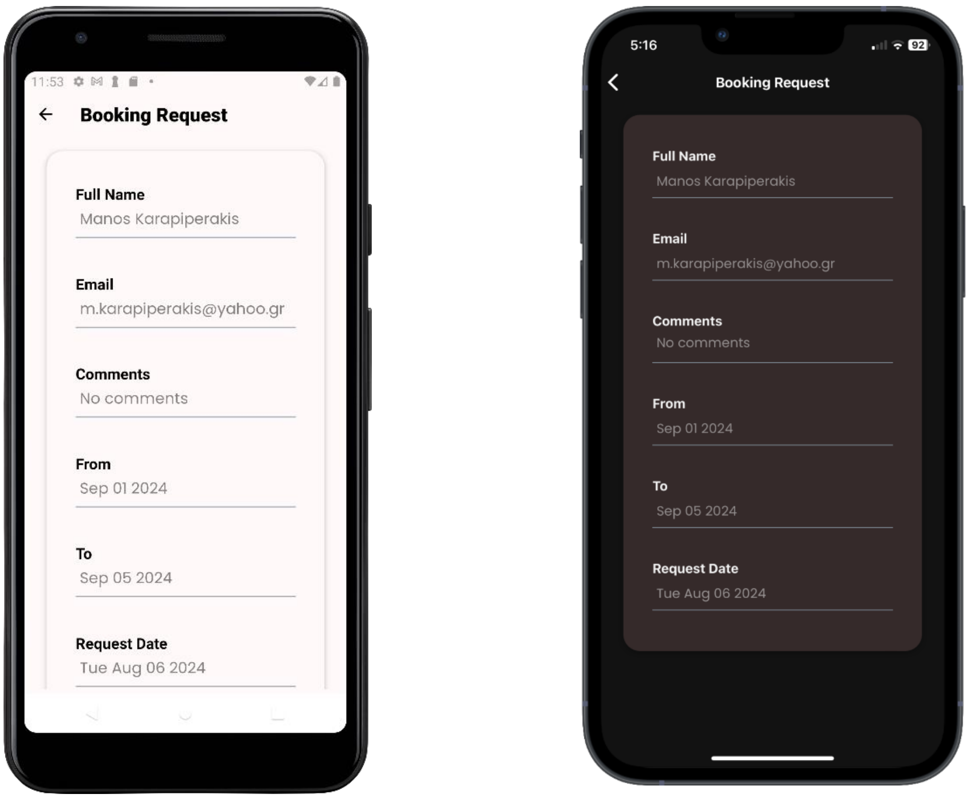
Name	Request Date
Manos Karapipe...	Tue Aug 06 2024
John Doe	Tue Aug 06 2024
Mat Doe	Tue Aug 06 2024
John Doe	Sun Aug 18 2024
John Doe	Sat Aug 24 2024

1 of 2

Availability

Figure 187 - Booking Requests

The user can click on a row, which will navigate them to a new screen where they can view all the details related to the booking request:


Booking Request

Full Name
Manos Karapiperakis

Email
m.karapiperakis@yahoo.gr

Comments
No comments

From
Sep 01 2024

To
Sep 05 2024

Request Date
Tue Aug 06 2024

Figure 188 - Booking Request Details

9.4.1.3 Availability

The next card available in the Admin Dashboard is Availability, where the user can modify the availability of the house that is presented during the booking request process for visitors.

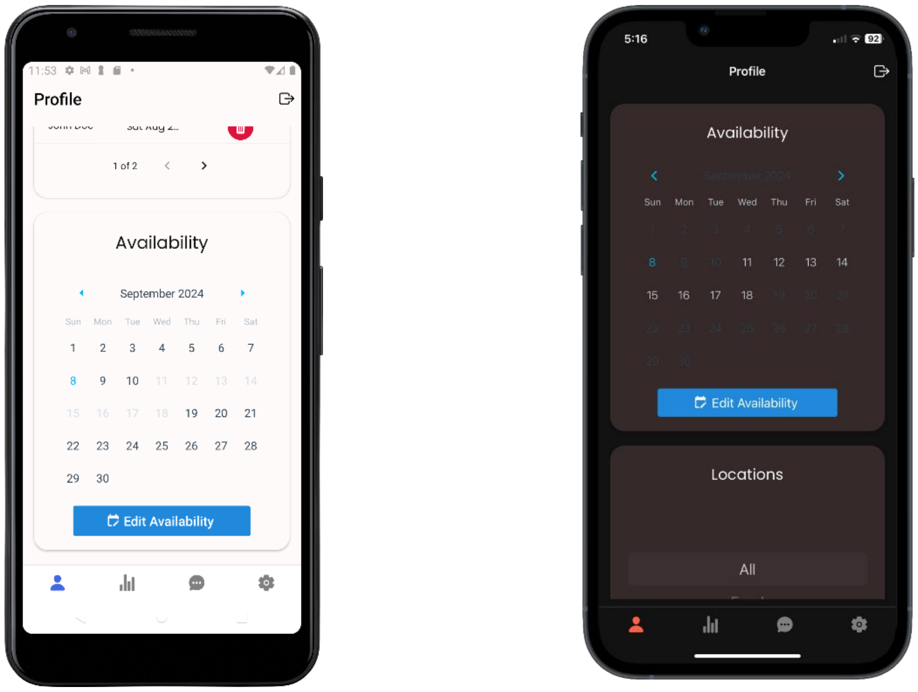


Figure 189 - Availability

By clicking the "Edit Availability" button, the user will be directed to a new screen where they can add a new booking by selecting the arrival and departure dates. This will mark the intervening days as unavailable:

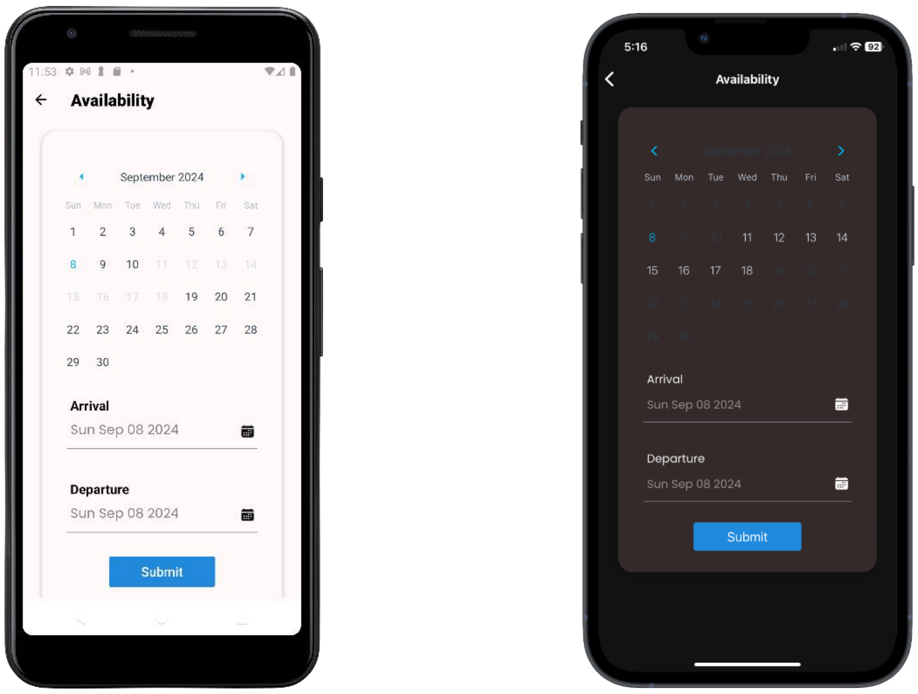
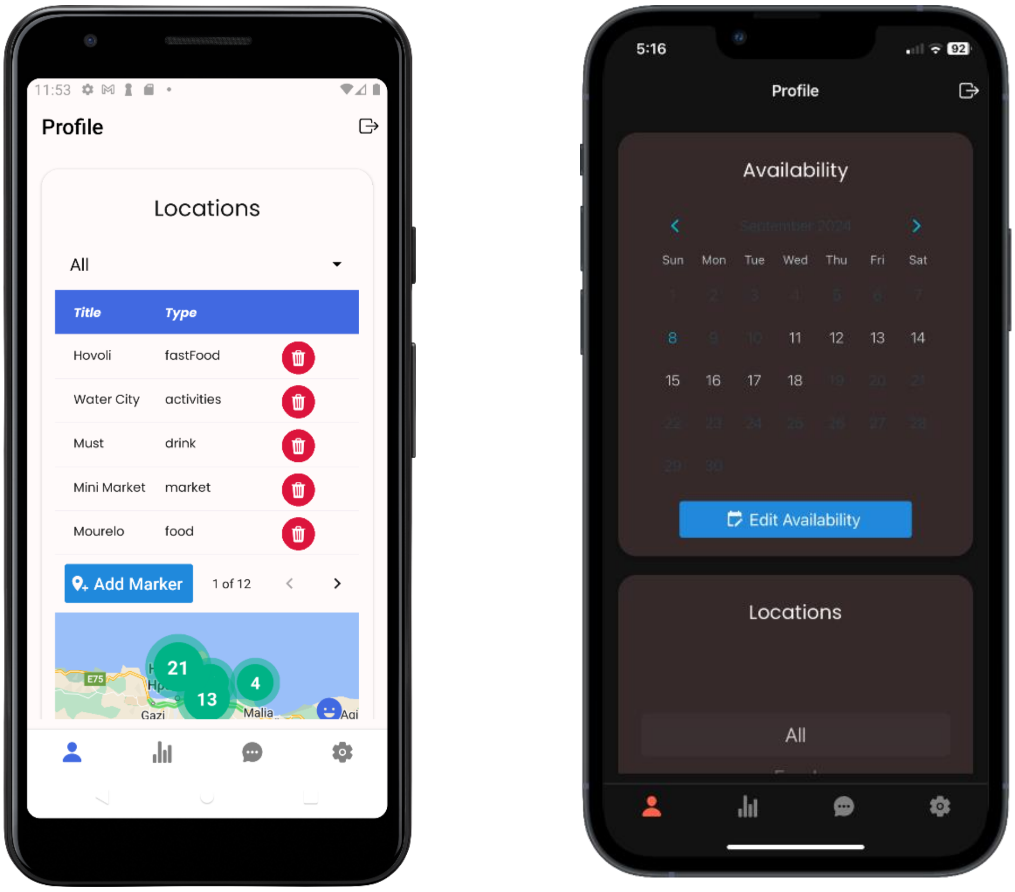


Figure 190 - Edit Availability

9.4.1.4 Locations

The "Locations" card is the final section where admins can view all available markers on dynamic maps for visitors. A filter by marker type is provided, allowing users to easily sort the markers. Additionally, users can add a new marker by clicking the "Add Marker" button or delete an existing one by selecting the "bin" icon next to the marker in the table.



Title	Type
Hovoli	fastFood
Water City	activities
Must	drink
Mini Market	market
Mourelo	food

Figure 191 - Locations

The addition of a new marker includes three steps:

1. Marker Information: In the first step, users provide details about the marker, including its title (a free text field) and its type, which is selected from a dropdown menu. The “Next” button allows users to proceed to the following steps, but it remains disabled until both of these mandatory fields are completed.

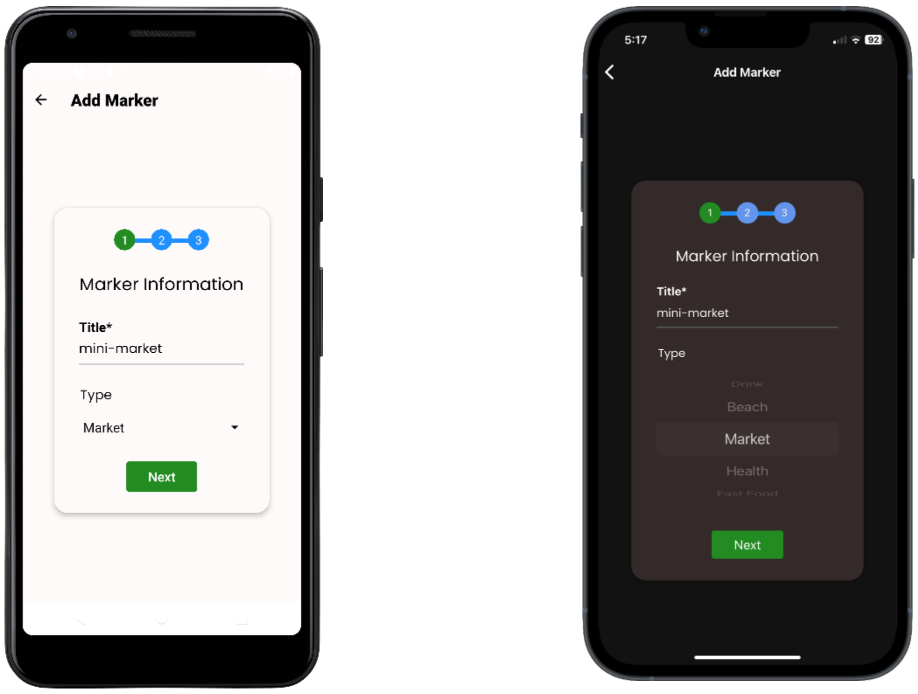


Figure 192 - Add Marker 1st step

2. Key Words: Key words are simple services associated with the selected marker. Users can type a keyword and click the "Add" button to include it. Each keyword will appear in a list, with a maximum of 12 keywords allowed. Keywords can be removed at any time by clicking the corresponding delete button.

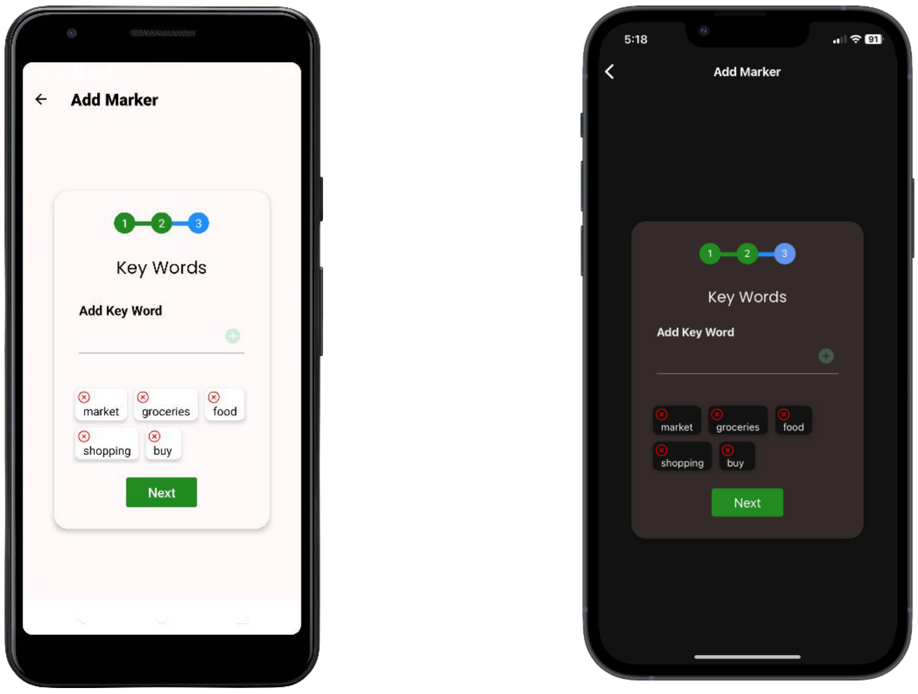
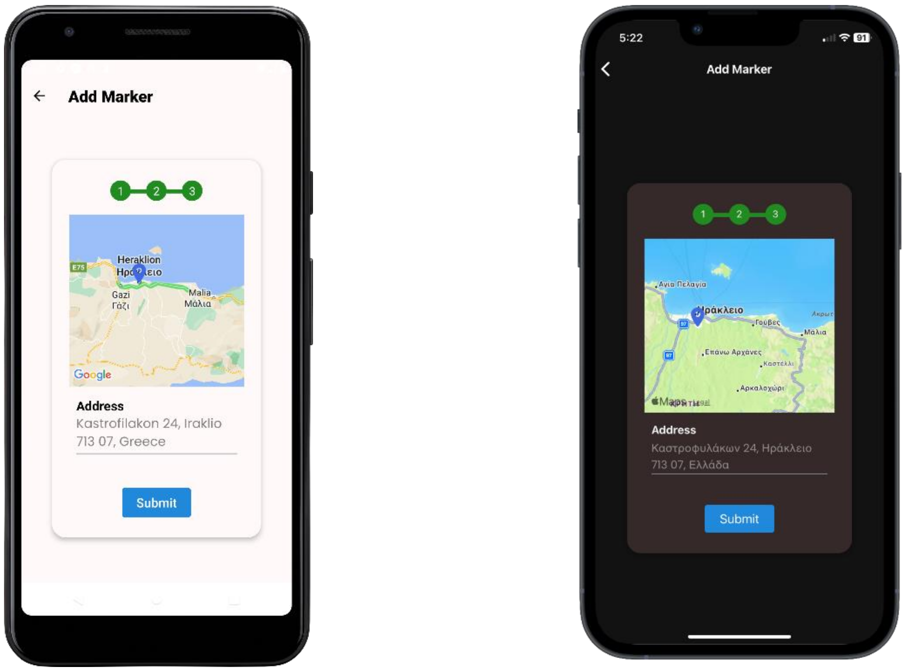


Figure 193 - Add Marker 2nd step

3. Location: In the final step, users select the precise location on the map by dragging the marker to the desired spot. Once placed, the physical address will be displayed and the user must confirm the selection by clicking the “Submit” button.



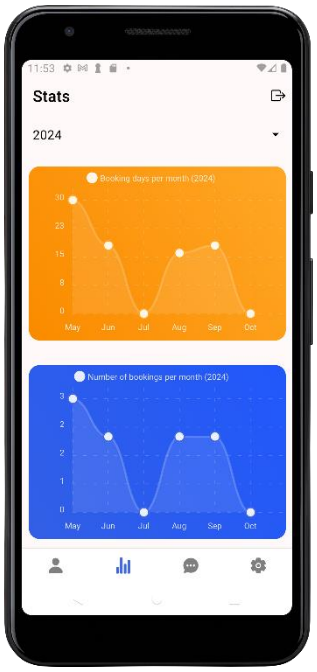
Address
Kastrofilakon 24, Iraklio
713 07, Greece

Address
Καστροφυλάκων 24, Ηράκλειο
713 07, Ελλάδα

Figure 194 - Add Marker 3rd step

9.4.2 Charts

This is the second screen available to admin users, offering various statistics related to the property and its guests. These include metrics such as the number of booking days per month and the total bookings per month for a selected year. Additionally, it provides insights into users' devices (iOS, Android) and their geographic origins, either for a specific year or across all time.


11:53
Stats
2024
Booking days per month (2024)
30
23
15
8
0
May
Jun
Jul
Aug
Sep
Oct
Number of bookings per month (2024)
3
2
2
1
0
May
Jun
Jul
Aug
Sep
Oct

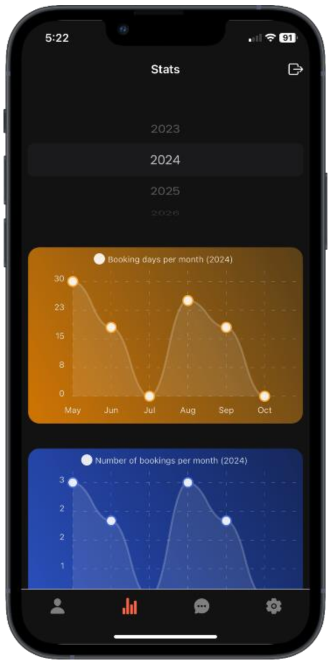

5:22
Stats
2023
2024
2025
2026
Booking days per month (2024)
30
23
15
8
0
May
Jun
Jul
Aug
Sep
Oct
Number of bookings per month (2024)
3
2
2
1

Figure 195 - Charts Screen

All the data displayed in these charts is retrieved from the user object. To visualize the information, two components (LineChart and PieChart) from the “react-native-chart-kit” package were used.

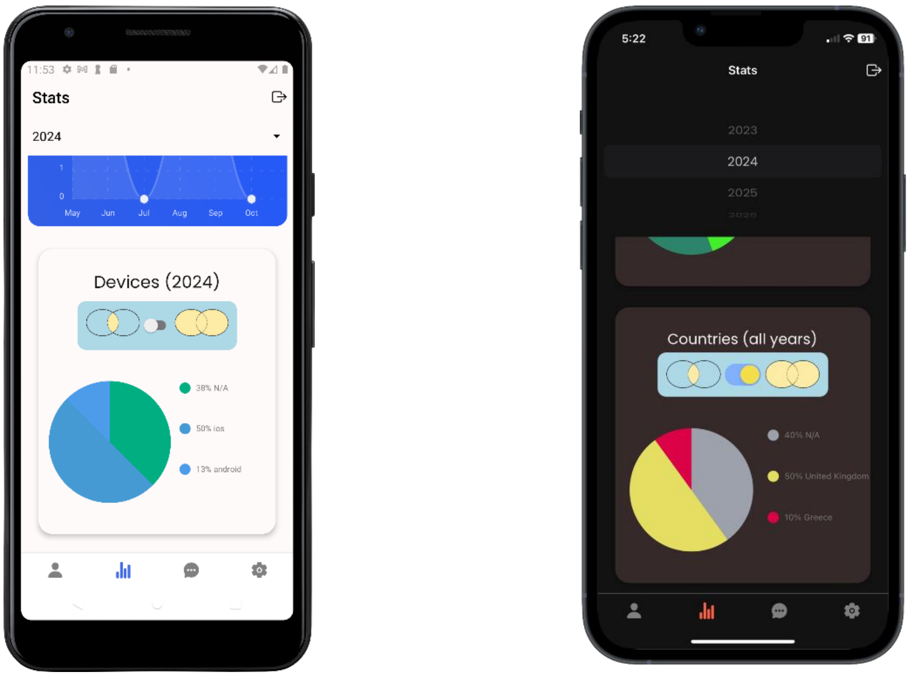
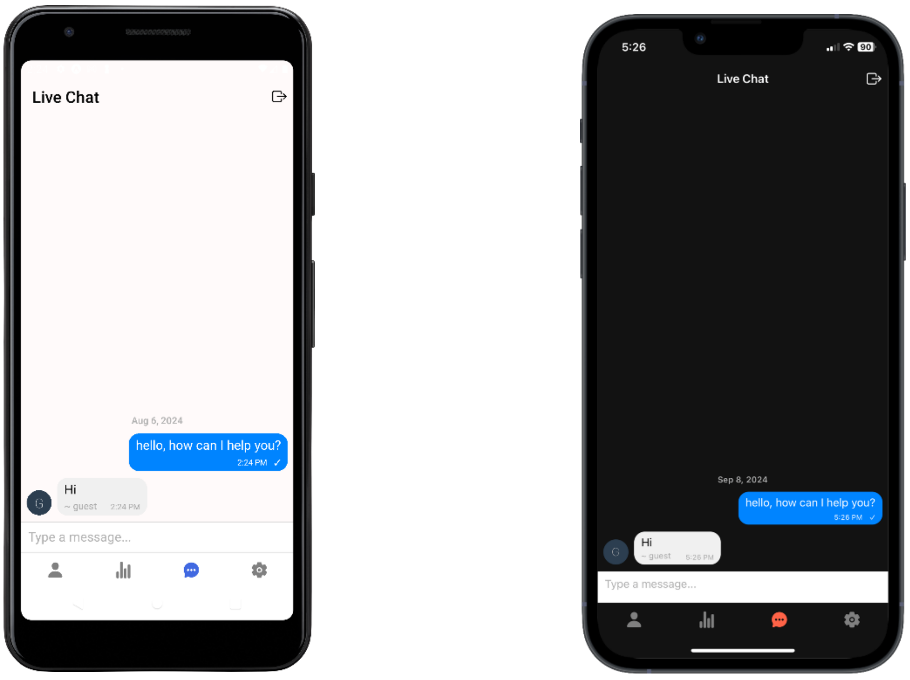


Figure 196 - Pie Charts

9.4.3 Live Chat

The final screen available to admin users is the live chat. Most functionalities are identical to those in the guest user's chat screen. However, the key difference is that the live chat is accessed via the bottom tab navigation, as the chatbot is not available for admin users. Additionally, the host can see the name of the user they are chatting with.


Live Chat

Aug 6, 2024

hello, how can I help you? 2:24 PM ✓

Hi ~ guest 2:24 PM

Type a message...

5:26

Live Chat

Sep 8, 2024

hello, how can I help you? 5:26 PM ✓

Hi ~ guest 5:26 PM

Type a message...

Figure 197 - Live Chat Admin

10 Heroku

10.1 Heroku Introduction

Heroku was chosen as the hosting platform for the application's two servers, each serving a distinct purpose. The first server hosts the back-end of the application, which includes RESTful APIs that expose the database system's resources to the mobile app. The second server is dedicated to managing the Socket.io connection, enabling real-time communication for the app's live chat feature.

10.2 Create Application

The first step is to log in to Heroku and create a new application [29], selecting the application's name and the hosting region. The available hosting regions are provided in a dropdown menu, primarily offering options between USA and Europe. Choosing the region impacts latency, with European servers providing lower latency for users in Europe compared to a U.S. based server.

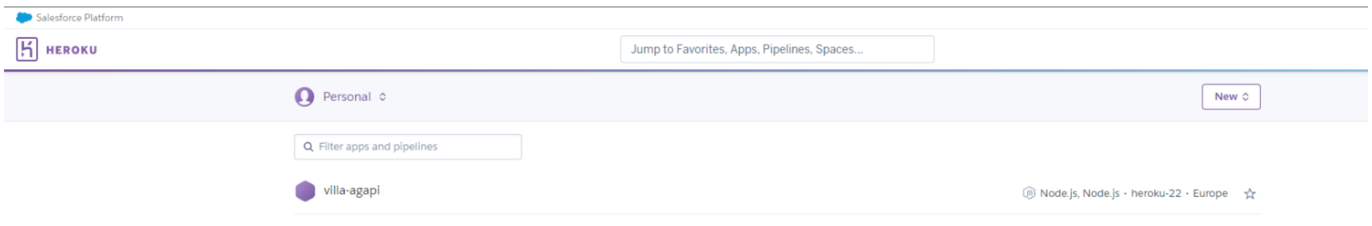

Salesforce Platform
HEROKU
Jump to Favorites, Apps, Pipelines, Spaces...
Personal
New
Filter apps and pipelines
villa-agapi
Node.js, Node.js · heroku-22 · Europe

Figure 198 - Heroku Create Application

Additionally, since this is a Node.js application, the Node version should be explicitly defined in the codebase. Otherwise, a default version, typically the latest stable release, will be used. To specify the desired version, a new attribute called "engines" should be added to the package.json file, which describes the target Node version, as shown below:

```
"engines": {
  "node": "21.2.0"
}
```

Furthermore, the scripts should be defined in the same file and can be executed using npm:

- start: Starts the server.
- start:dev: Starts the server in development mode using nodemon. By default, Heroku applications run in production mode, so nodemon cannot be used in the live app, as it is installed only in local dependencies.
- test: Runs the Playwright tests.

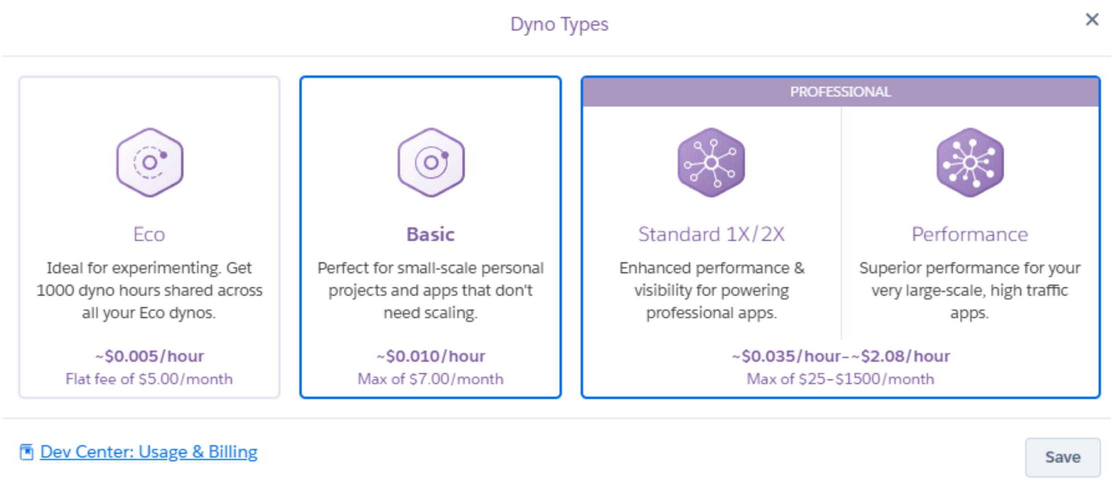
```
"scripts": {
    "start": "node src/app.js",
    "start:dev": "nodemon src/app.js",
    "tests": "npx playwright test --project=chromium"
  }
```

10.3 Dynos

After creating the application, the next step is to select the dyno that will be associated with it. The application's code runs on the Heroku platform within structures known as dynos [30]. These dynos are runtime containers that operate on a Linux operating system in the background, executing the processes necessary to run the application's custom code. Heroku offers three types of dynos:

- Eco
- Basic
- Professional

For this project, Eco dynos were chosen during the development phase. However, one disadvantage of Eco dynos is that they enter sleep mode after 30 minutes of inactivity. When a new request is made while the application is in sleep mode, the server can take up to 10 seconds to "wake up", which creates an inconvenience for end users. Therefore, after completing the basic development and making the application accessible to users, Basic dynos were implemented to improve performance and user experience.


Dyno Types

Eco
Ideal for experimenting. Get 1000 dyno hours shared across all your Eco dynos.
~$0.005/hour
Flat fee of $5.00/month

Basic
Perfect for small-scale personal projects and apps that don't need scaling.
~$0.010/hour
Max of $7.00/month

PROFESSIONAL

Standard 1X/2X
Enhanced performance & visibility for powering professional apps.

Performance
Superior performance for your very large-scale, high traffic apps.

~$0.035/hour–~$2.08/hour
Max of $25–$1500/month

Figure 199 - Dyno Types

10.4 Slugs & Buildpacks

Slugs are compressed and pre-packaged copies of applications optimized for distribution on the dyno manager [31]. When code is pushed to Heroku, it is processed by the slug compiler, which converts it into a slug. At the core of the slug compiler is a collection of scripts known as buildpacks, which are responsible for managing different programming languages. All applications written in Ruby, Python, Java, Clojure, Node.js, Scala, Go and PHP are built and compiled using buildpacks.

The desired buildpack can be selected from the "Settings" tab by clicking the "Add buildpack" button:

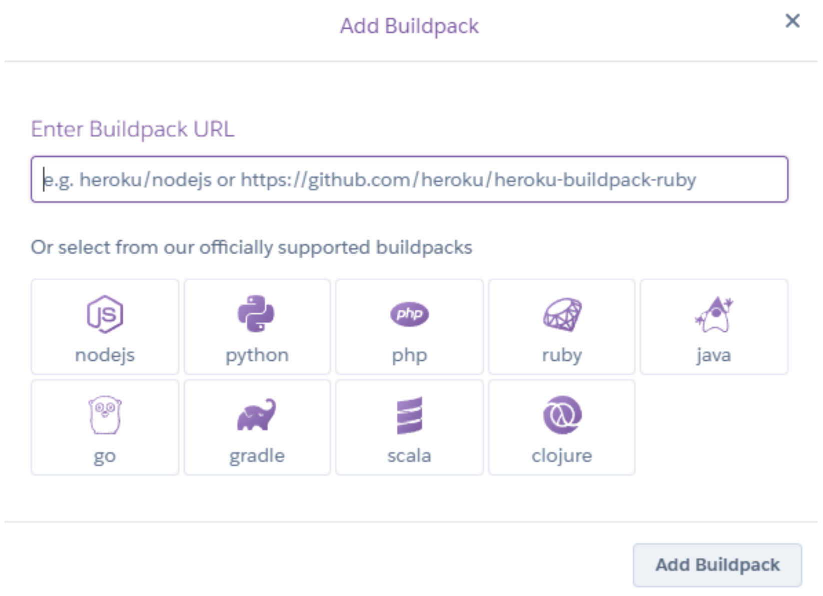

Add Buildpack
Enter Buildpack URL
e.g. heroku/nodejs or https://github.com/heroku/heroku-buildpack-ruby
Or select from our officially supported buildpacks
nodejs
python
php
ruby
java
go
gradle
scala
clojure
Add Buildpack

Figure 200 - Buildpacks

In addition to the standard buildpacks provided by Heroku, custom buildpacks can also be added. For the server application, the Playwright buildpack has been imported. To accomplish this, a new file named app.json has been created in the root directory of the project, containing a URL that points to the GitHub page hosting the Playwright buildpack. Finally, the same "test" command should be defined within the scripts, as it will be used later in the testing process.

```
{
    "environments": {
      "test": {
        "buildpacks": [
          {
            "url": "https://github.com/mxschmitt/heroku-playwright-
buildpack.git"
          },
          {
            "url": "https://github.com/heroku/heroku-buildpack-
nodejs"
          }
        ],
        "scripts": {
          "test": "npx playwright test --project=chromium"
        }
      }
    }
  }
```

10.5 Heroku CLI

The Heroku Command Line Interface (Heroku CLI) is a vital component of the Heroku platform. It allows users to create and manage applications directly from the terminal, providing a powerful toolset for developers. With the Heroku CLI, users can deploy code, scale applications and configure environment variables with ease. Additionally, it offers features for logging and viewing application logs, as well as connecting to databases. To use the Heroku CLI, the Heroku npm package should be installed globally, executing the command "npm install -g Heroku".

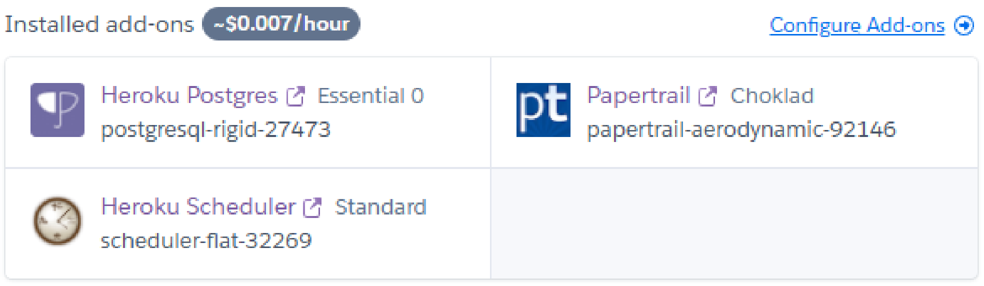
10.6 Add-ons

Add-ons are an essential part of the Heroku platform. They enable users to integrate complex features into their applications without needing to build and manage them from scratch. Examples of commonly used add-ons include:

- Data Stores: MySQL, Heroku Postgres, MariaDB, Apache Kafka.
- Logging: Papertrail, Coralogix.
- Email/SMS Services: Mailgun, Trustifi.
- Messaging and Queueing: RabbitMQ.
- Dynos and Scheduling: Heroku Scheduler.

In this application the following Add-ons were used:

- Heroku Postgres
- Heroku Scheduler
- Papertrail



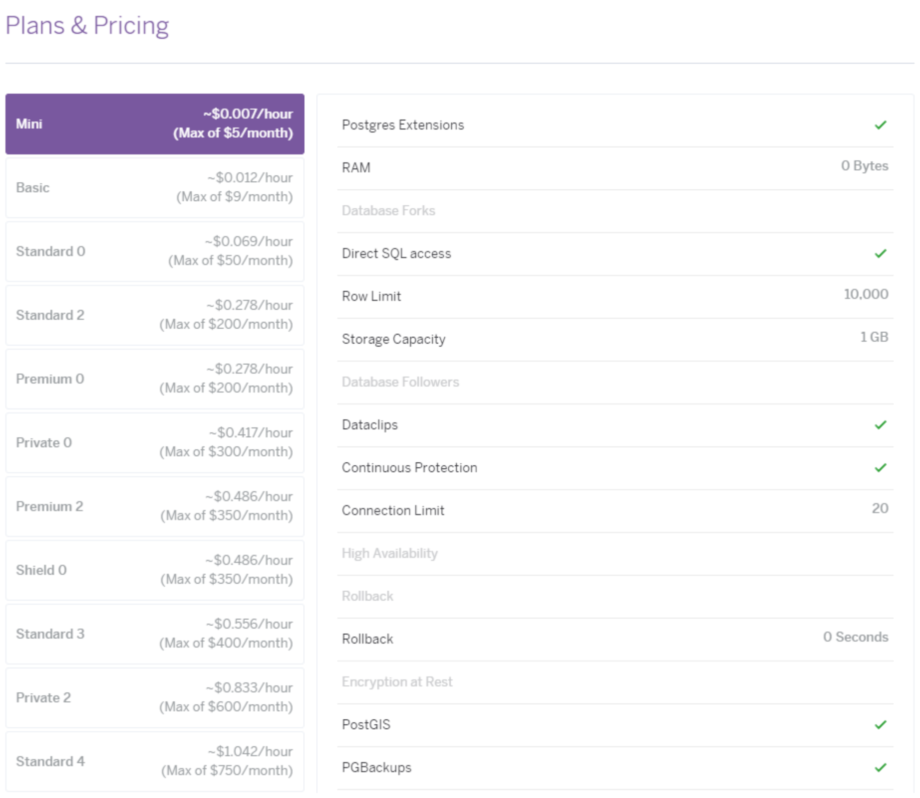
Installed add-ons ~$0.007/hour
Configure Add-ons

Heroku Postgres Essential 0 postgresql-rigid-27473	Papertrail Choklad papertrail-aerodynamic-92146
Heroku Scheduler Standard scheduler-flat-32269	

Figure 201 - Add ons

10.6.1 Heroku Postgres

Heroku Postgres is the first add-on that was added, allowing to include a Postgres database in the application. The first step is selecting the appropriate plan, as add-ons are not always free to use. Prices range from 5€ up to 34,000€ per month. Since the application's database has relatively low requirements and a modest level of complexity, the affordable plan ("Mini") was chosen:



Plans & Pricing

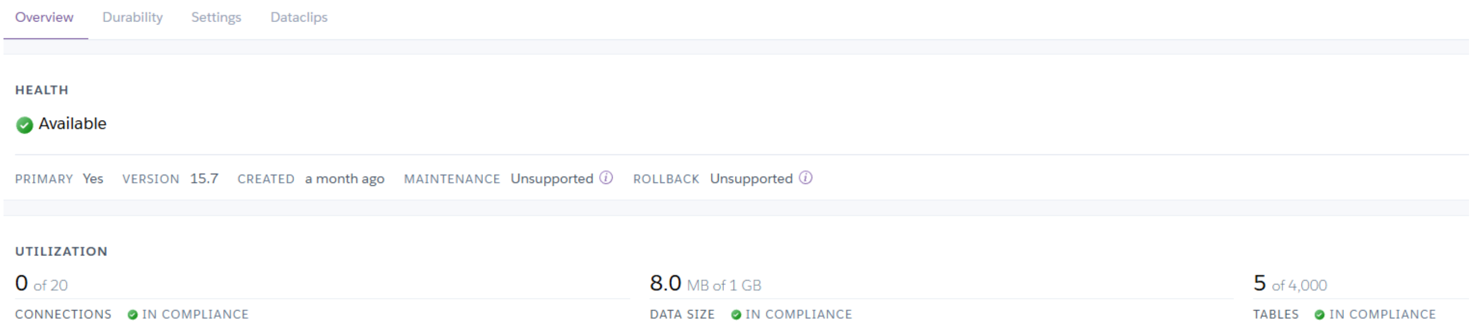
Mini	~$0.007/hour (Max of $5/month)
Basic	~$0.012/hour (Max of $9/month)
Standard 0	~$0.069/hour (Max of $50/month)
Standard 2	~$0.278/hour (Max of $200/month)
Premium 0	~$0.278/hour (Max of $200/month)
Private 0	~$0.417/hour (Max of $300/month)
Premium 2	~$0.486/hour (Max of $350/month)
Shield 0	~$0.486/hour (Max of $350/month)
Standard 3	~$0.556/hour (Max of $400/month)
Private 2	~$0.833/hour (Max of $600/month)
Standard 4	~$1.042/hour (Max of $750/month)

Postgres Extensions	✓
RAM	0 Bytes
Database Forks	
Direct SQL access	✓
Row Limit	10,000
Storage Capacity	1 GB
Database Followers	
Dataclips	✓
Continuous Protection	✓
Connection Limit	20
High Availability	
Rollback	
Rollback	0 Seconds
Encryption at Rest	
PostGIS	✓
PGBackups	✓

Figure 202 - Heroku Postgres Add on

Opening the newly added add-on displays a screen with four navigation options:

- Overview: Provides database details such as region, version, size and number of tables:



HEALTH
Available

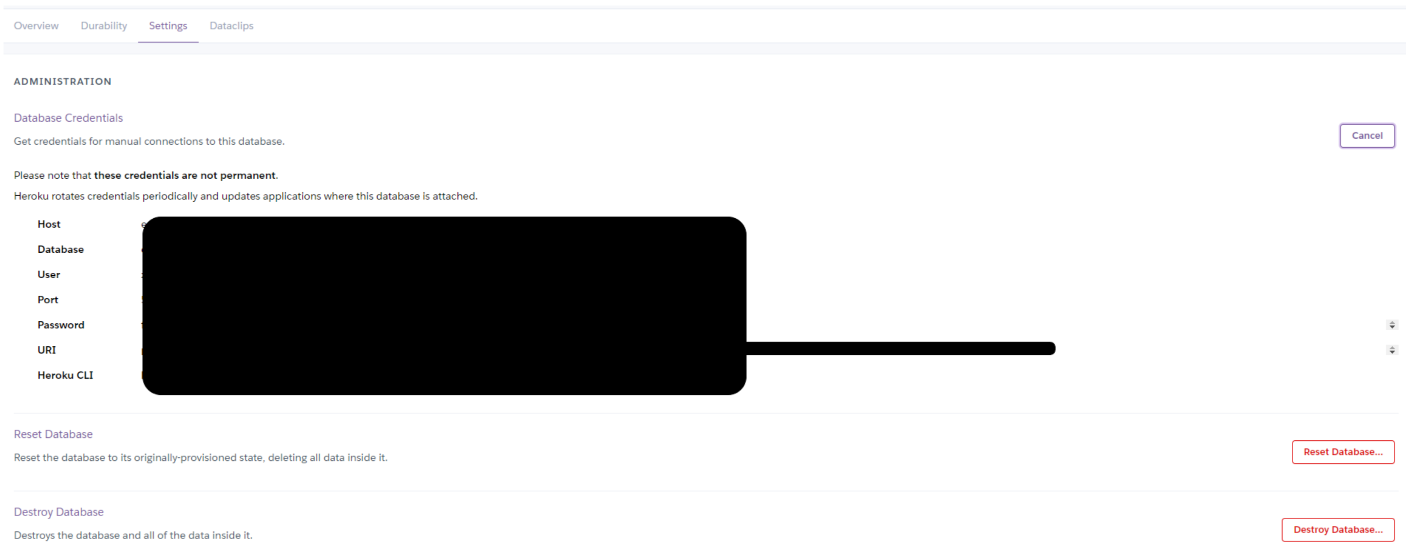
PRIMARY Yes VERSION 15.7 CREATED a month ago MAINTENANCE Unsupported ROLLBACK Unsupported

UTILIZATION

0 of 20 CONNECTIONS IN COMPLIANCE
8.0 MB of 1 GB DATA SIZE IN COMPLIANCE
5 of 4,000 TABLES IN COMPLIANCE

Figure 203 - Heroku Postgres Overview

- Durability: Offers backup functionality (manual only, due to the selected budget-friendly plan):
- Settings: Shows general database information, including name, host, URI and password.



Overview Durability Settings Dataclips

ADMINISTRATION

Database Credentials

Get credentials for manual connections to this database.

Please note that **these credentials are not permanent**.

Heroku rotates credentials periodically and updates applications where this database is attached.

Host	[redacted]
Database	
User	
Port	
Password	
URI	
Heroku CLI	

Reset Database

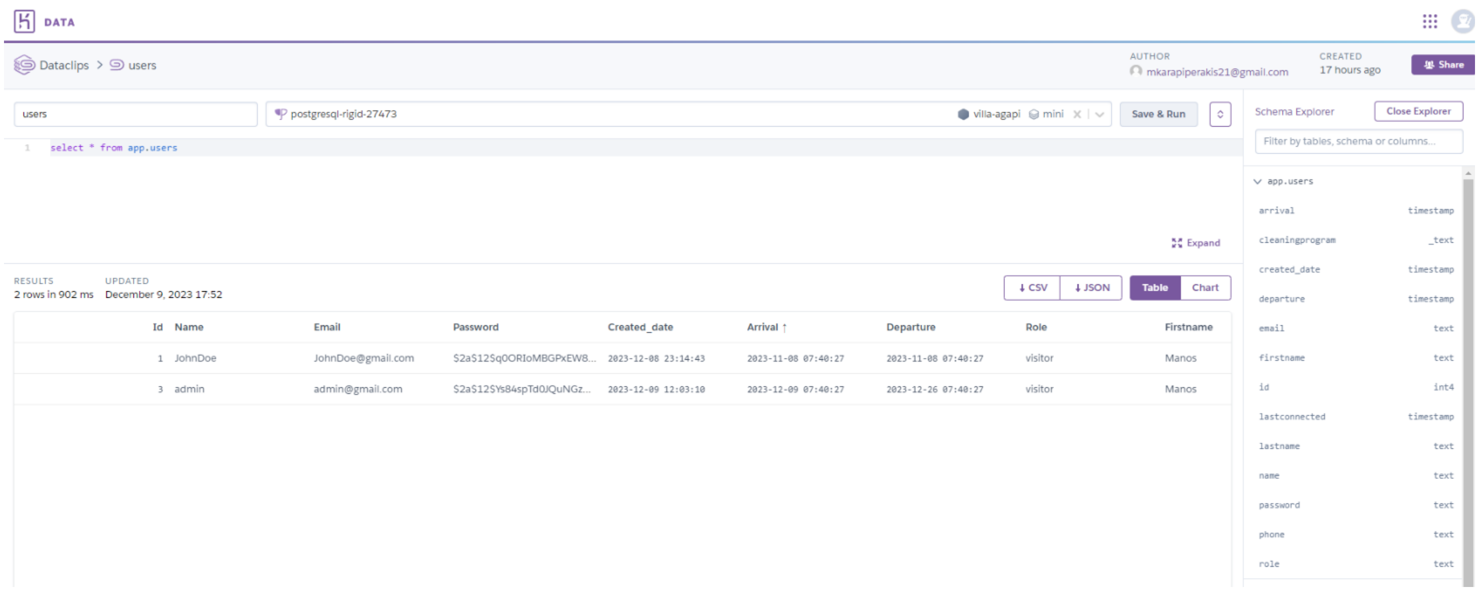
Reset the database to its originally-provisioned state, deleting all data inside it.

Destroy Database

Destroys the database and all of the data inside it.

Figure 204 - Heroku Postgres Settings

- Data Clips: Allows for SQL queries, though insert, update and delete commands are not permitted. This feature is intended for viewing and exporting data.



```
select * from app.users
```

RESULTS 2 rows in 902 ms UPDATED December 9, 2023 17:52

Id	Name	Email	Password	Created_date	Arrival ↑	Departure	Role	Firstname
1	JohnDoe	JohnDoe@gmail.com	$2a$12$q0ORIoMBGPxEW8...	2023-12-08 23:14:43	2023-11-08 07:40:27	2023-11-08 07:40:27	visitor	Manos
3	admin	admin@gmail.com	$2a$12$Ys84spTd0JQuNGz...	2023-12-09 12:03:10	2023-12-09 07:40:27	2023-12-26 07:40:27	visitor	Manos

app.users

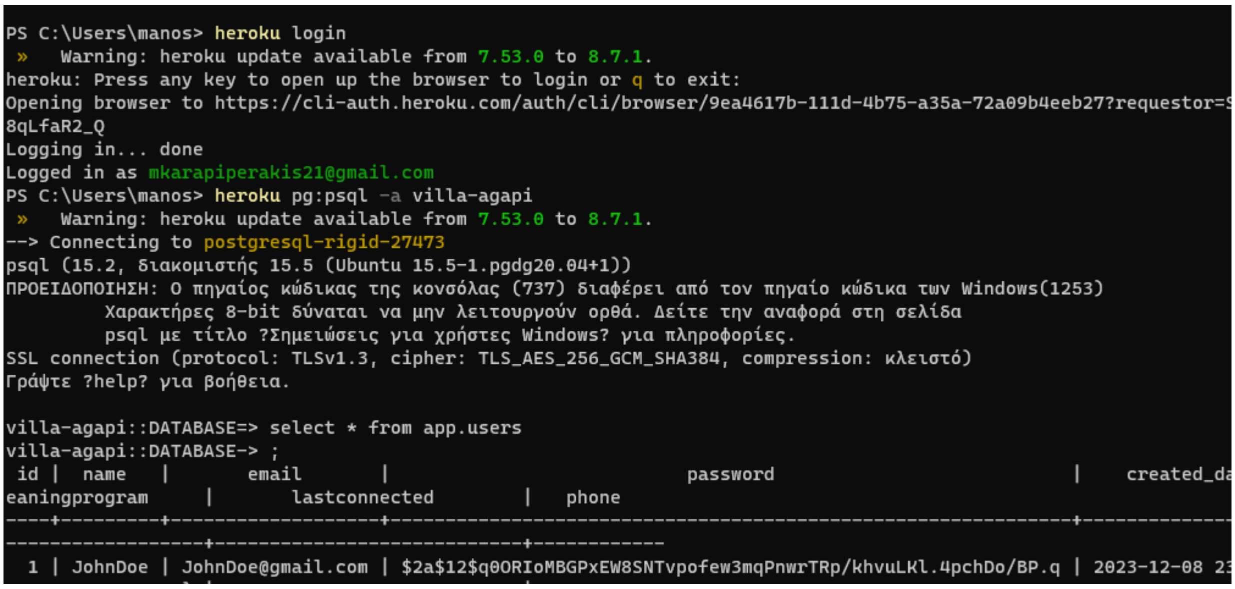
arrival	timestamp
cleaningprogram	_text
created_date	timestamp
departure	timestamp
email	text
firstname	text
id	int4
lastconnected	timestamp
lastname	text
name	text
password	text
phone	text
role	text

Figure 205 - Heroku Postgres Dataclips

Data Clips only allow select queries that do not modify the database content. If a user needs to interact with the database, there are two available options:

1. Using Database Credentials: Database credentials from the *Settings* tab can be used to connect through a database client like pgAdmin.
2. Using the Heroku CLI: The Heroku CLI can be used to connect to the database directly from the terminal. This requires the following two commands:
   - heroku login: Opens a new browser tab for user authentication.

- heroku pg:psql -a <app name>: Initiates a connection to the database server.

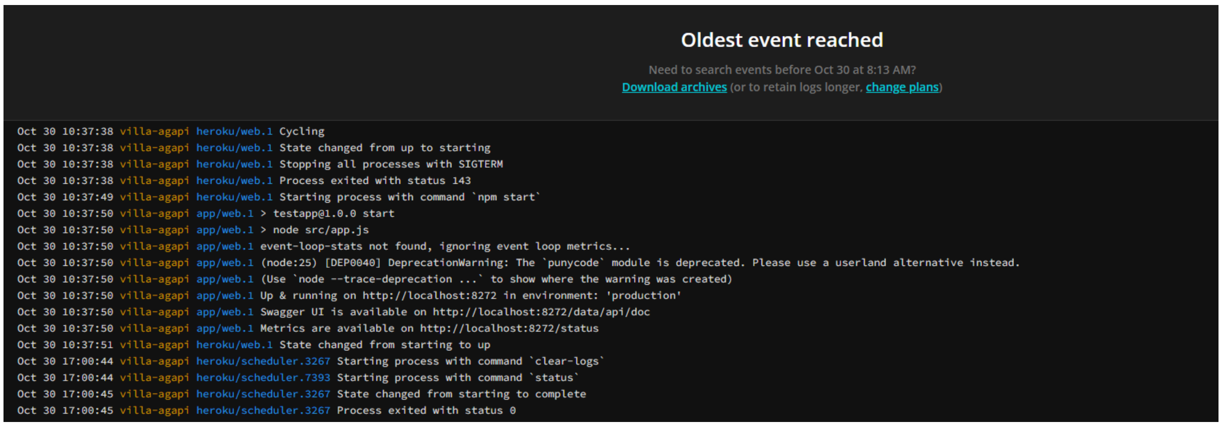


```
PS C:\Users\manos> heroku login
 »   Warning: heroku update available from 7.53.0 to 8.7.1.
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/9ea4617b-111d-4b75-a35a-72a09b4eeb27?requestor=S
8qLfaR2_Q
Logging in... done
Logged in as mkarapiperakis21@gmail.com
PS C:\Users\manos> heroku pg:psql -a villa-agapi
 »   Warning: heroku update available from 7.53.0 to 8.7.1.
--> Connecting to postgresql-rigid-27473
psql (15.2, διακομιστής 15.5 (Ubuntu 15.5-1.pgdg20.04+1))
ΠΡΟΕΙΔΟΠΟΙΗΣΗ: Ο πηγαίος κώδικας της κονσόλας (737) διαφέρει από τον πηγαίο κώδικα των Windows(1253)
         Χαρακτήρες 8-bit δύναται να μην λειτουργούν ορθά. Δείτε την αναφορά στη σελίδα
         psql με τίτλο ?Σημειώσεις για χρήστες Windows? για πληροφορίες.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: κλειστό)
Γράψτε ?help? για βοήθεια.

villa-agapi::DATABASE=> select * from app.users
villa-agapi::DATABASE-> ;
 id |  name    |       email       |                           password                           |    created_da
eaningprogram     |        lastconnected         |  phone
----+----------+-------------------+--------------------------------------------------------------+-------------
------------------+------------------------------+------------
  1 | JohnDoe  | JohnDoe@gmail.com | $2a$12$q0ORIoMBGPxEW8SNTvpofew3mqPnwrTRp/khvuLKl.4pchDo/BP.q | 2023-12-08 23
```

Figure 206 - Connect to Heroku Postgres

10.6.2 Papertrail

Papertrail is the second add-on used for the application, providing a convenient way to view and monitor application logs in real-time.



Oldest event reached

Need to search events before Oct 30 at 8:13 AM?
Download archives (or to retain logs longer, change plans)

```
Oct 30 10:37:38 villa-agapi heroku/web.1 Cycling
Oct 30 10:37:38 villa-agapi heroku/web.1 State changed from up to starting
Oct 30 10:37:38 villa-agapi heroku/web.1 Stopping all processes with SIGTERM
Oct 30 10:37:38 villa-agapi heroku/web.1 Process exited with status 143
Oct 30 10:37:49 villa-agapi heroku/web.1 Starting process with command `npm start`
Oct 30 10:37:50 villa-agapi app/web.1 > testapp@1.0.0 start
Oct 30 10:37:50 villa-agapi app/web.1 > node src/app.js
Oct 30 10:37:50 villa-agapi app/web.1 event-loop-stats not found, ignoring event loop metrics...
Oct 30 10:37:50 villa-agapi app/web.1 (node:25) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
Oct 30 10:37:50 villa-agapi app/web.1 (Use `node --trace-deprecation ...` to show where the warning was created)
Oct 30 10:37:50 villa-agapi app/web.1 Up & running on http://localhost:8272 in environment: 'production'
Oct 30 10:37:50 villa-agapi app/web.1 Swagger UI is available on http://localhost:8272/data/api/doc
Oct 30 10:37:50 villa-agapi app/web.1 Metrics are available on http://localhost:8272/status
Oct 30 10:37:51 villa-agapi heroku/web.1 State changed from starting to up
Oct 30 17:00:44 villa-agapi heroku/scheduler.3267 Starting process with command `clear-logs`
Oct 30 17:00:44 villa-agapi heroku/scheduler.7393 Starting process with command `status`
Oct 30 17:00:45 villa-agapi heroku/scheduler.3267 State changed from starting to complete
Oct 30 17:00:45 villa-agapi heroku/scheduler.3267 Process exited with status 0
```

Figure 207 - Papertrail

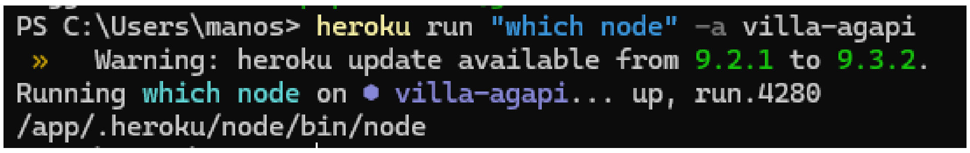
10.6.3 Heroku Scheduler

Heroku Scheduler [32] is the final add-on used, enabling scheduled tasks to run at specific intervals, every 10 minutes, every hour or once daily. This tool is usefull for automating repeative jobs, such as data backups, maintenance tasks or API calls. In this application, two scheduled jobs have been created:

1. Status: This job runs daily to update user statuses by comparing each user's departure date with the current date. If the current date is later than the departure date, the user's status is set to inactive (false). This process ensures that only active users have access to the application, fulfilling a business requirement.

```
#!/app/.heroku/node/bin/node

async function checkStatus() {
  try {
    const users = await getUsers();
    const expectedDate = new Date();
    expectedDate.setDate(expectedDate.getDate() - 1);

    for (const user of users) {
      const departureDate = new Date(user.departure);
      if (expectedDate > departureDate && user.is_active !== false)
{
        await updateUserStatus(user.id, false);
        counter++;
      }
    }
    console.log(`${counter} users out of ${users.length} have been
updated`);
  } catch (error) {
    console.error("Error fetching users:", error);
  } finally {
    process.exit();
  }
}
```

For Heroku scheduled jobs, the script file needs to be placed in a specific location within the project structure. Heroku expects the file to be inside a folder named bin at the root level of the project and the file name should exactly match the name of the job created in Heroku Scheduler. Moreover the first line inside this file should be the "shebang" also know as hashbang [33]. The shebang line specifies the interpreter that should be used to execute the script. The first two characters (#!) tell the system that this file should be executed by the interpreter specified in the path that follows. The remainder of the line indicates the exact location of the interpreter.
To find this specific path to be used, the command "heroku run "which node" -a <app name>" should be executed:



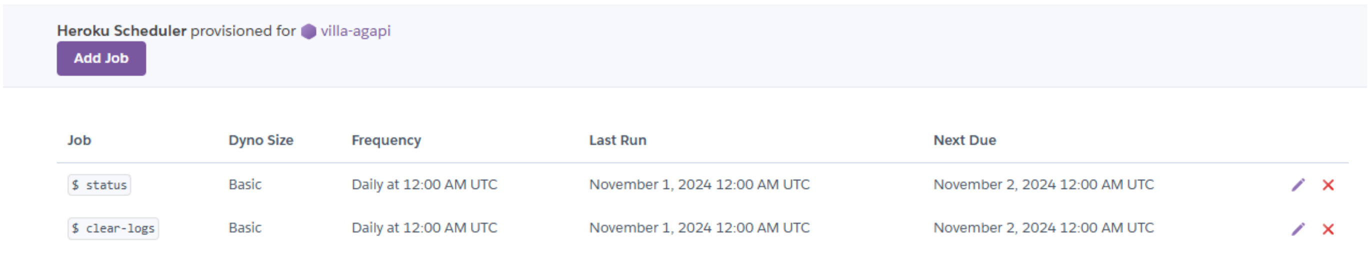
```
PS C:\Users\manos> heroku run "which node" -a villa-agapi
 »   Warning: heroku update available from 9.2.1 to 9.3.2.
Running which node on ● villa-agapi... up, run.4280
/app/.heroku/node/bin/node
```

Figure 208 - Which Node

2. clear-logs: This is the second scheduled job that has been created and it is responsible for clearing data from the app.attack table. This table is typically used to monitor suspicious activity from potential attackers. However, due to the limited database size associated with the mini plan, it was essential to implement an automated method for clearing logs periodically. Consequently, a job was created to clear these logs once a week, specifically every Sunday. Although the job can be set to run every 10 minutes, every hour or every day, it has

been configured to run daily. Each time it runs, the job checks the current day of the week. If it is Sunday, all records from the table are deleted and the job prints the number of records that were affected:

```
#!/app/.heroku/node/bin/node

async function clearDB() {
  const date = new Date();
  let rows = 0;
  //Every Sunday clear logs
  if (date.getDay() === 0) {
    try {
      let result = await deleteLogs();
      rows = result.rowCount;
    } catch (err) {
      console.log(`error deleting logs: ${err}`);
    }
  }

  console.log(`Process clear-logs has been completed. ${rows} ${rows
=== 1 ? "row has" : "rows have"} been affected `)

  process.exit();
}
```

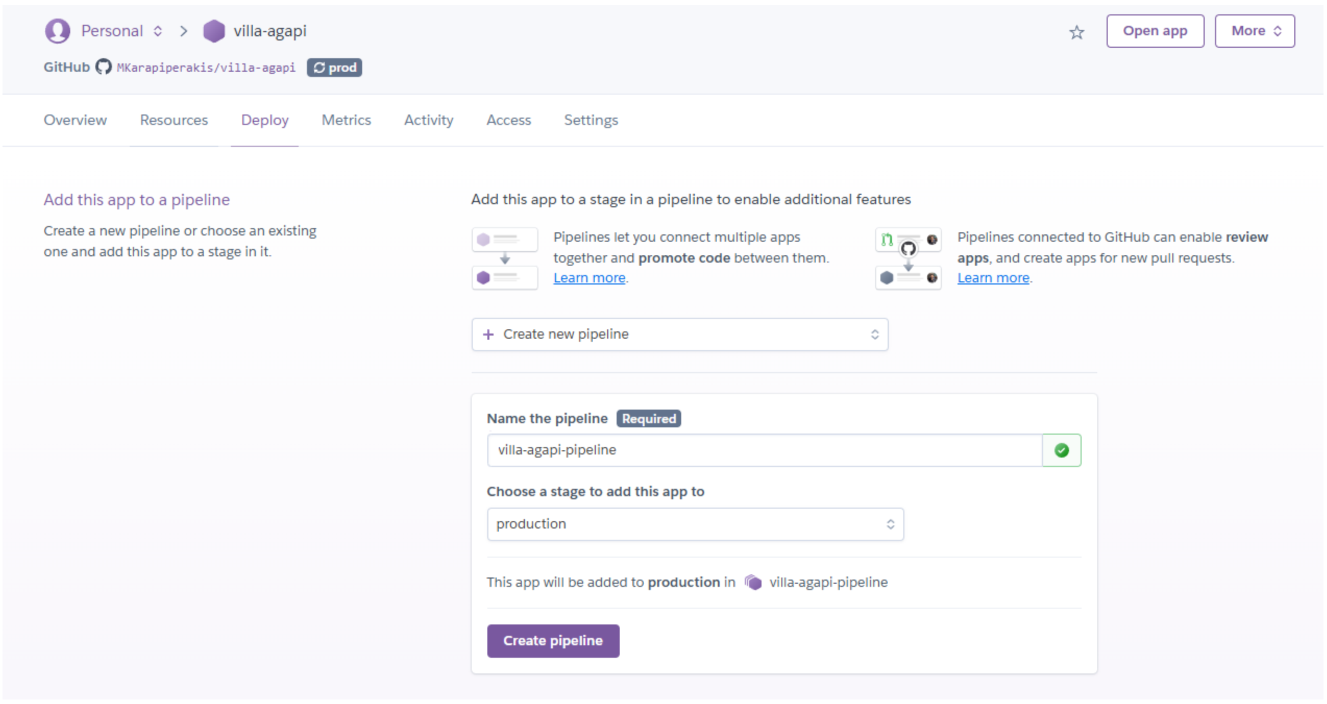

Heroku Scheduler provisioned for villa-agapi

Job	Dyno Size	Frequency	Last Run	Next Due
`$ status`	Basic	Daily at 12:00 AM UTC	November 1, 2024 12:00 AM UTC	November 2, 2024 12:00 AM UTC
`$ clear-logs`	Basic	Daily at 12:00 AM UTC	November 1, 2024 12:00 AM UTC	November 2, 2024 12:00 AM UTC

Figure 209 - Heroku Scheduler

10.7 CI/CD

Heroku offers continuous integration (CI) support through its Pipelines and Heroku CI [34]. With these features, developers can automate the testing process by configuring Heroku to run application tests automatically whenever changes are pushed to the GitHub repository associated with the app. To set up Heroku CI, the first step is to create a new pipeline, which serves as an organized workflow for managing and deploying different versions of the application across development, staging and production stages:



Add this app to a pipeline

Create a new pipeline or choose an existing one and add this app to a stage in it.

Add this app to a stage in a pipeline to enable additional features

Pipelines let you connect multiple apps together and **promote code** between them. Learn more.

Pipelines connected to GitHub can enable **review apps**, and create apps for new pull requests. Learn more.

Create new pipeline

Name the pipeline Required: villa-agapi-pipeline

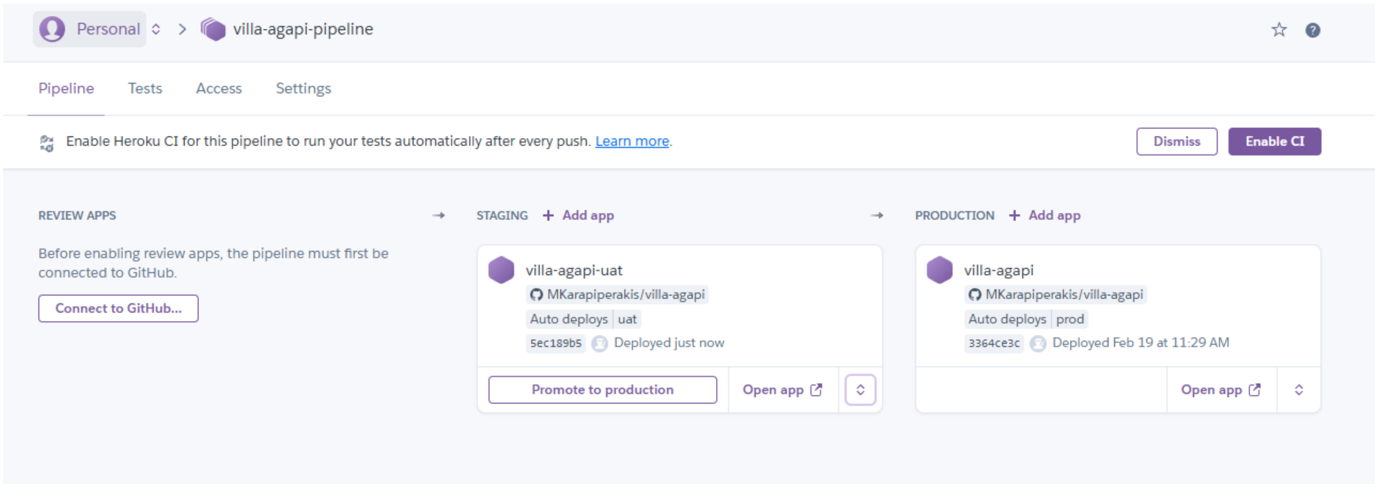
Choose a stage to add this app to: production

This app will be added to **production** in villa-agapi-pipeline

Create pipeline

Figure 210 - Heroku Create Pipeline

For the existing Node.js application, a new Heroku pipeline was created to facilitate a streamlined deployment process. The application was moved to the production stage within this pipeline and associated with the "prod" branch of the GitHub repository, ensuring that changes pushed to this branch are ready for live deployment. Additionally, a separate app named "villa-agapi-uat" was created as a staging environment. This staging app was added to the pipeline just before the production stage, allowing for pre-production testing and is connected with the “uat” branch respectively.



Enable Heroku CI for this pipeline to run your tests automatically after every push. Learn more.

REVIEW APPS

Before enabling review apps, the pipeline must first be connected to GitHub.

STAGING + Add app

villa-agapi-uat
MKarapiperakis/villa-agapi
Auto deploys uat
5ec189b5 Deployed just now

PRODUCTION + Add app

villa-agapi
MKarapiperakis/villa-agapi
Auto deploys prod
3364ce3c Deployed Feb 19 at 11:29 AM

Figure 211 - Create Staging App

The new staging app, "villa-agapi-uat", has its own environment variables, so it can function independently from production. It uses the same dynos as the production app, so there are no extra costs for running it.

To save on database fees though, the staging app is connected to the same PostgreSQL database as production. This allows testing with live data without needing a second database instance.
After that, this application should be configured, enabling automatic deploys:

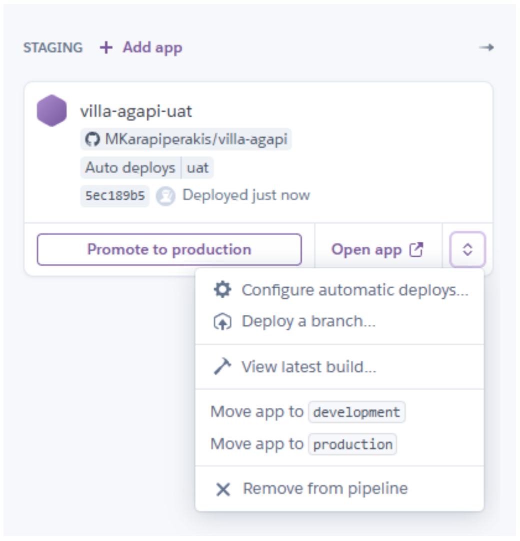
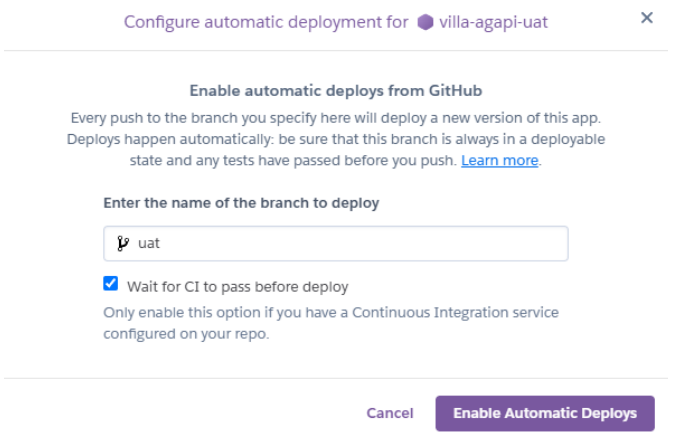


Figure 212 - Configure Automatic Deploys

From there the the desired branch (uat) should be selected, the "Wait for CI to pass before deploy" checkbox should be checked and the "Enable Automatic Deploys" button should be clicked:


Configure automatic deployment for villa-agapi-uat

Enable automatic deploys from GitHub

Every push to the branch you specify here will deploy a new version of this app. Deploys happen automatically: be sure that this branch is always in a deployable state and any tests have passed before you push. Learn more.

Enter the name of the branch to deploy

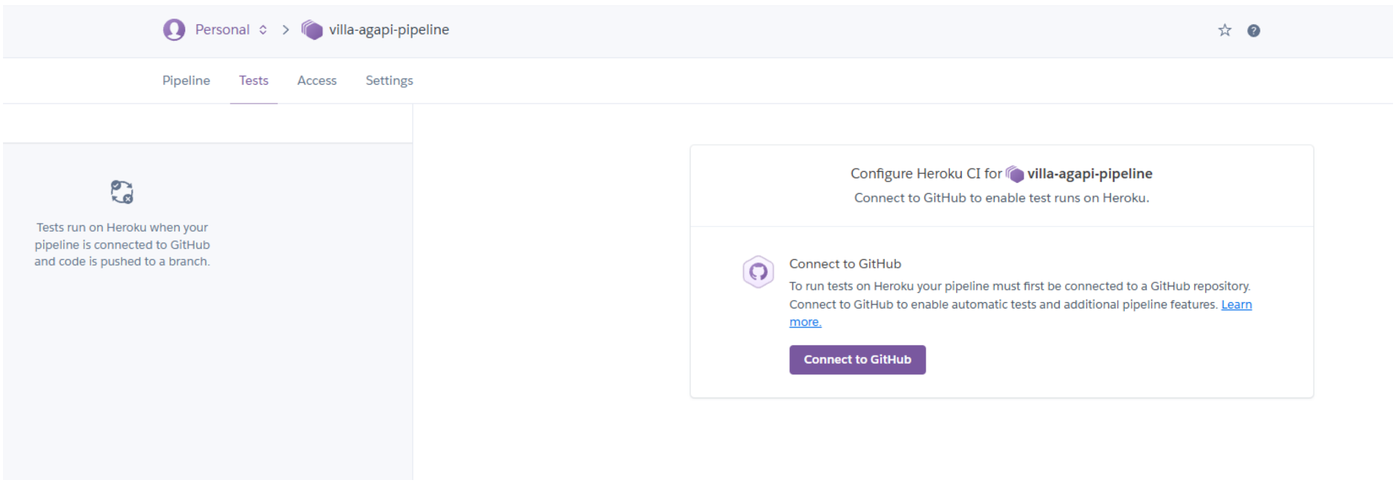
uat

[✓] Wait for CI to pass before deploy

Only enable this option if you have a Continuous Integration service configured on your repo.

Figure 213 - Enable Automatic Deploys

After that the tests should be enabled by navigating to the tab “Tests” in the pipeline and connecting with GitHub:


Personal > villa-agapi-pipeline

Pipeline Tests Access Settings

Tests run on Heroku when your pipeline is connected to GitHub and code is pushed to a branch.

Configure Heroku CI for villa-agapi-pipeline

Connect to GitHub to enable test runs on Heroku.

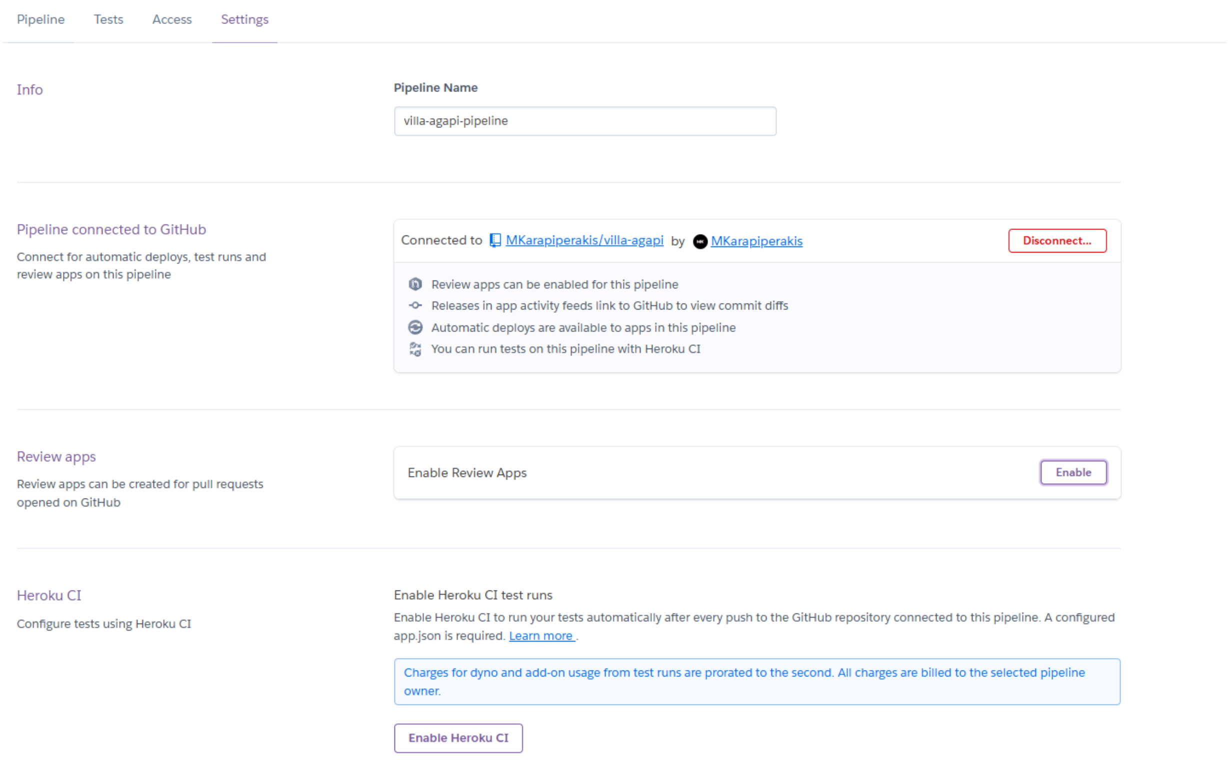
Connect to GitHub

To run tests on Heroku your pipeline must first be connected to a GitHub repository. Connect to GitHub to enable automatic tests and additional pipeline features. Learn more.

Connect to GitHub

Figure 214 - Configure Pipeline Tests

As a final configuration step, the "Review Apps" option should be enabled from the Settings tab along with Heroku CI, if it is not already activated:


Pipeline Tests Access Settings

Info

Pipeline Name: villa-agapi-pipeline

Pipeline connected to GitHub

Connect for automatic deploys, test runs and review apps on this pipeline

Connected to MKarapiperakis/villa-agapi by MKarapiperakis Disconnect...

- Review apps can be enabled for this pipeline
- Releases in app activity feeds link to GitHub to view commit diffs
- Automatic deploys are available to apps in this pipeline
- You can run tests on this pipeline with Heroku CI

Review apps

Review apps can be created for pull requests opened on GitHub

Enable Review Apps Enable

Heroku CI

Configure tests using Heroku CI

Enable Heroku CI test runs

Enable Heroku CI to run your tests automatically after every push to the GitHub repository connected to this pipeline. A configured app.json is required. Learn more .

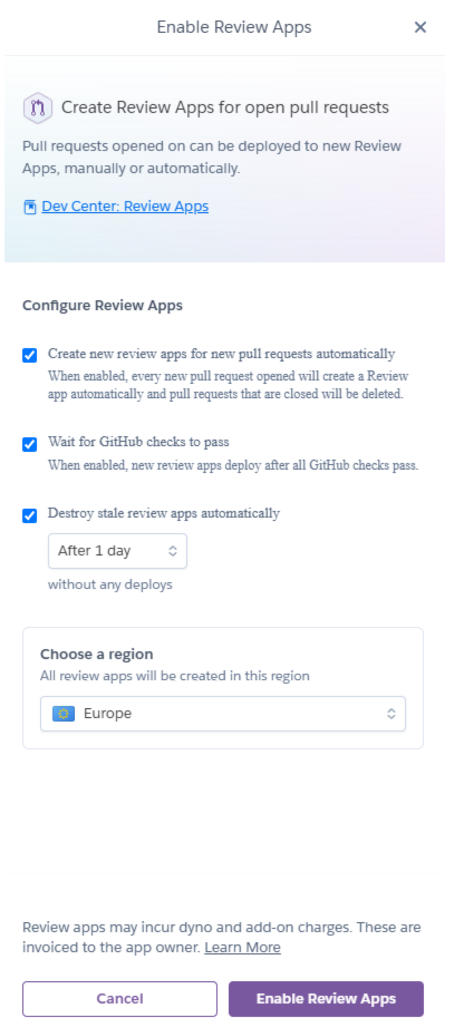
Charges for dyno and add-on usage from test runs are prorated to the second. All charges are billed to the selected pipeline owner.

Enable Heroku CI

Figure 215 - Enable Review Apps

For the Review Apps, the following configuration is important:

- Create new review apps for new pull requests automatically
- Wait for GitHub checks to pass
- Destroy state review apps automatically after 1 day without any deploys



Enable Review Apps

Create Review Apps for open pull requests

Pull requests opened on can be deployed to new Review Apps, manually or automatically.

Dev Center: Review Apps

Configure Review Apps

[✓] Create new review apps for new pull requests automatically
When enabled, every new pull request opened will create a Review app automatically and pull requests that are closed will be deleted.

[✓] Wait for GitHub checks to pass
When enabled, new review apps deploy after all GitHub checks pass.

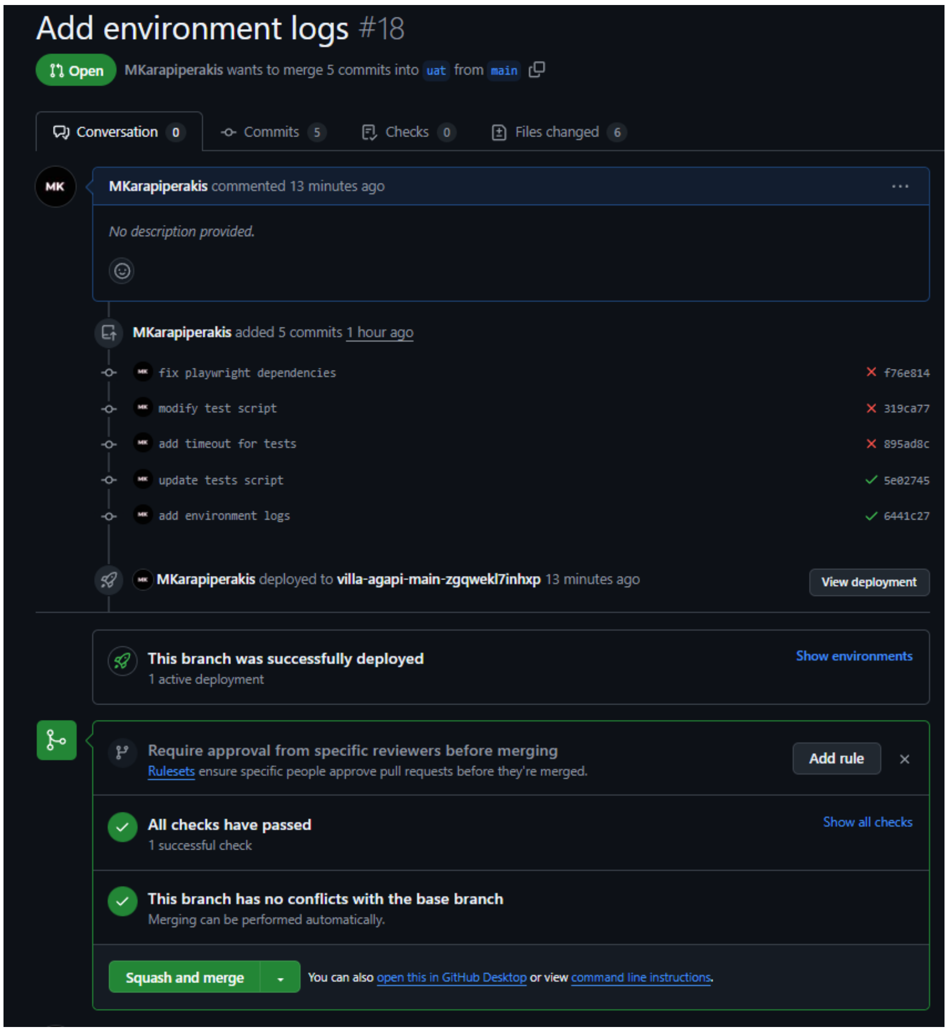
[✓] Destroy stale review apps automatically
After 1 day
without any deploys

Choose a region
All review apps will be created in this region
Europe

Review apps may incur dyno and add-on charges. These are invoiced to the app owner. Learn More

Figure 216 - Review Apps Configuration

Now that the pipelines are set up, a new review app will be created for every new pull request targeting the UAT branch and the Playwright tests available for the server will be executed.



Add environment logs #18

Open MKarapiperakis wants to merge 5 commits into uat from main

Conversation 0 · Commits 5 · Checks 0 · Files changed 6

MKarapiperakis commented 13 minutes ago

No description provided.

MKarapiperakis added 5 commits 1 hour ago

```
fix playwright dependencies        ✗ f76e814
modify test script                 ✗ 319ca77
add timeout for tests              ✗ 895ad8c
update tests script                ✓ 5e02745
add environment logs               ✓ 6441c27
```

MKarapiperakis deployed to villa-agapi-main-zgqwekl7inhxp 13 minutes ago · View deployment

This branch was successfully deployed
1 active deployment · Show environments

Require approval from specific reviewers before merging
Rulesets ensure specific people approve pull requests before they're merged. · Add rule

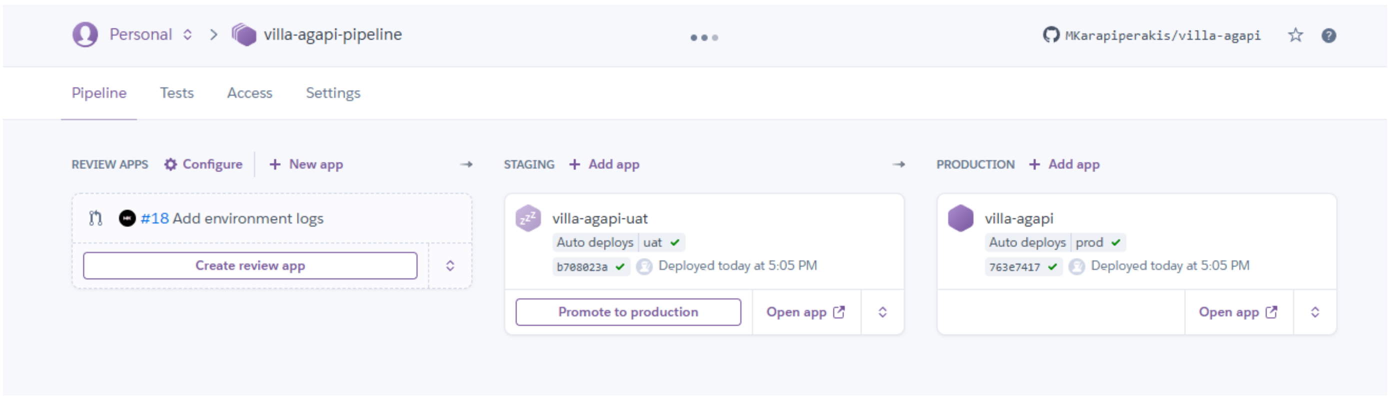
All checks have passed
1 successful check · Show all checks

This branch has no conflicts with the base branch
Merging can be performed automatically.

Squash and merge · You can also open this in GitHub Desktop or view command line instructions.

Figure 217 - Raise PR

Once the checks pass, a new slug will be created for the "villa-agapi-uat" app, which can then be promoted to production by clicking the respective button:



Personal > villa-agapi-pipeline · MKarapiperakis/villa-agapi

Pipeline · Tests · Access · Settings

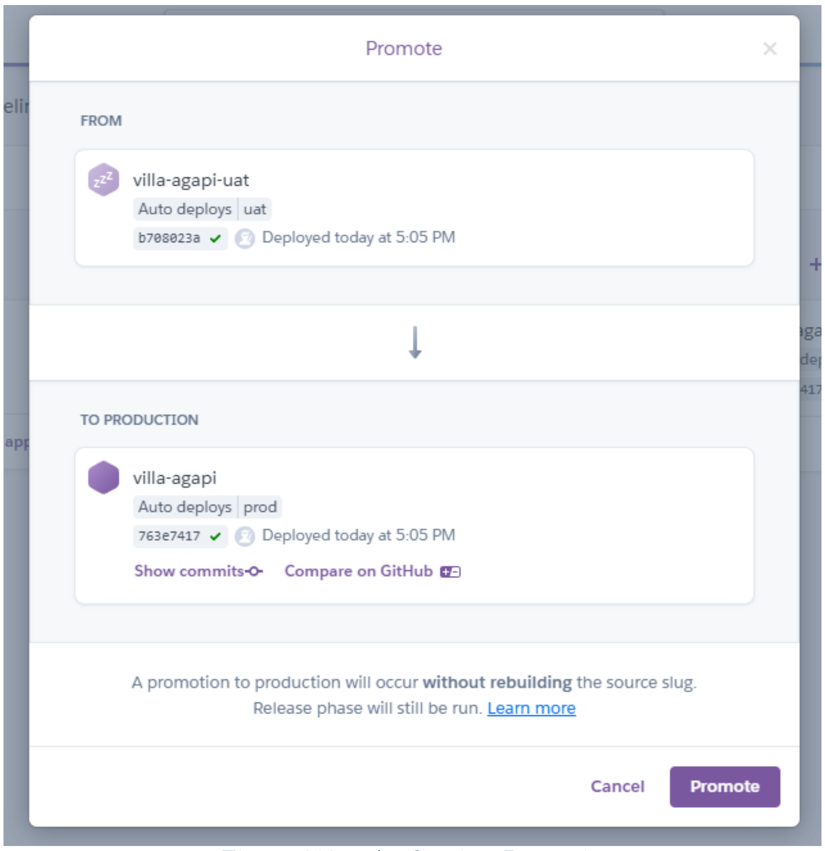
REVIEW APPS · Configure · New app
#18 Add environment logs · Create review app

STAGING · Add app
villa-agapi-uat · Auto deploys uat ✓
b708023a ✓ Deployed today at 5:05 PM
Promote to production · Open app

PRODUCTION · Add app
villa-agapi · Auto deploys prod ✓
763e7417 ✓ Deployed today at 5:05 PM
Open app

Figure 218 - Pipeline Overview

Promoting the app to production does not affect the "prod" repository, only the slug is deployed to this stage from UAT. However, it is good practice to remain aligned with UAT after the build is completed and the tests have passed.


Promote

FROM

villa-agapi-uat
Auto deploys | uat
b708023a ✓ Deployed today at 5:05 PM

TO PRODUCTION

villa-agapi
Auto deploys | prod
763e7417 ✓ Deployed today at 5:05 PM
Show commits Compare on GitHub

A promotion to production will occur **without rebuilding** the source slug.
Release phase will still be run. Learn more

Cancel Promote

Figure 219 - Application Promotion

10.8 Release Process

As a result of the previously described CI process, the release procedure consists of the following steps:

1. For each new feature to be added, a feature branch is created from the "main" branch, following the naming convention: "feat/<feature-short-description>"
2. This feature branch is then merged into the "main" branch.
3. A new Pull Request (PR) is raised from the "main" branch targeting the "uat" branch.
4. This action initiates a new review app process in Heroku, which runs all the tests before deploying to the "uat" branch.
5. After all tests pass, the changes are merged into the "uat" branch on GitHub.
6. The tests defined for the UAT are executed again, if they pass successfully, the build for the UAT pipeline begins.
7. Upon completion of the build, it can be promoted to production, where the respective tests are run once more. If these tests pass, the build process is initiated.

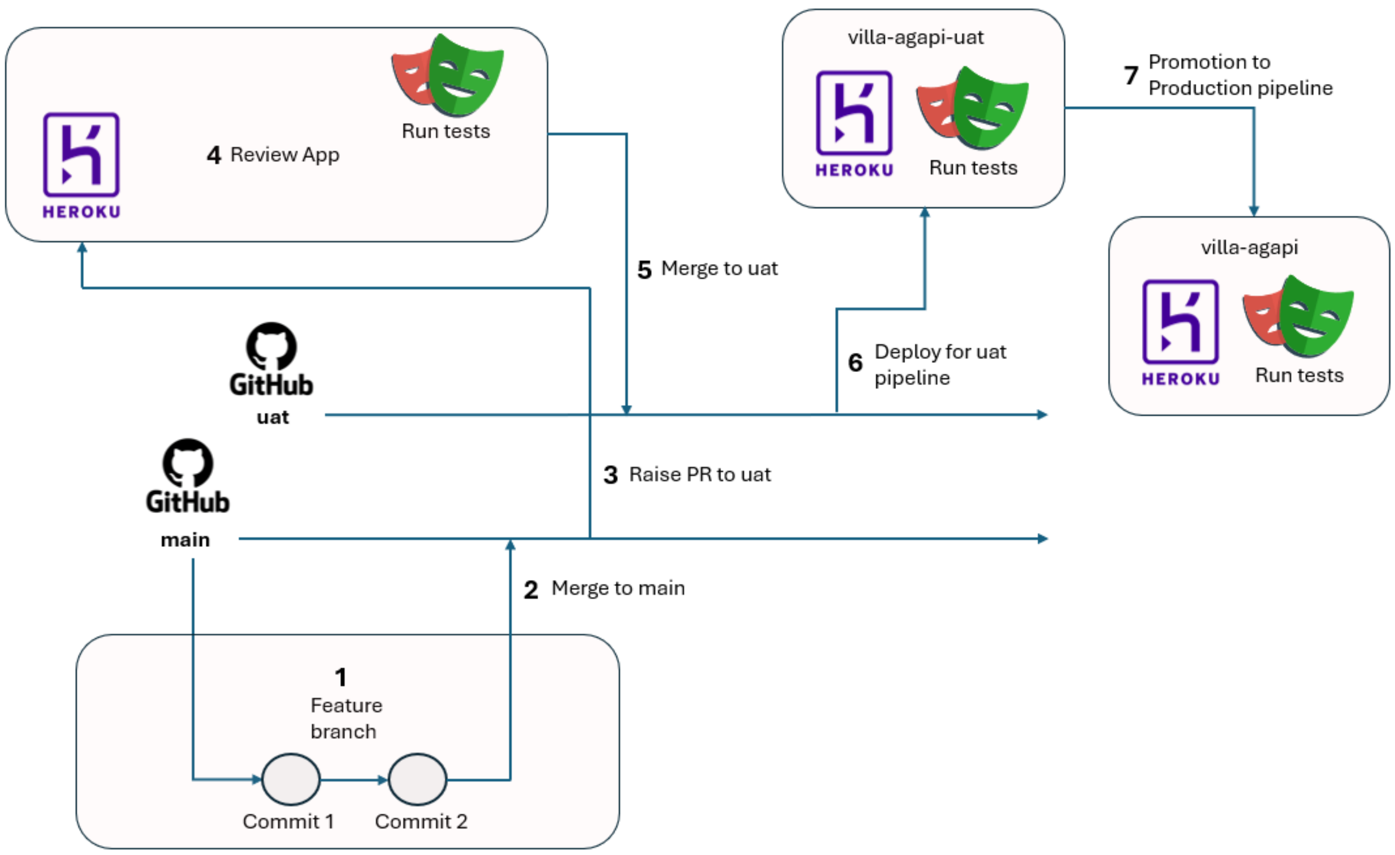
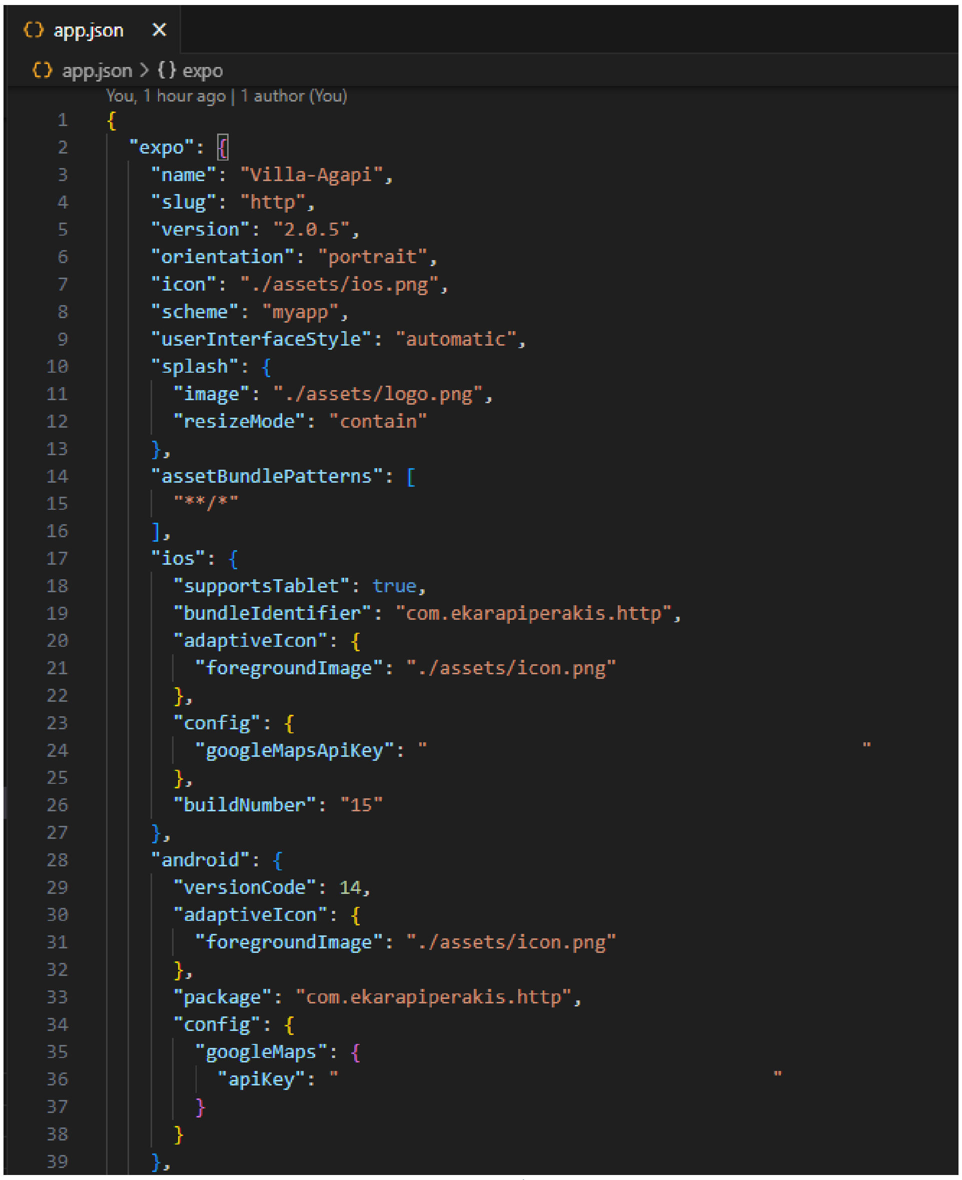


Figure 220 - Release Process

11 Deployment for Mobile Application

11.1 Metadata Configuration

Before deploying the mobile application online, the first step is to configure essential metadata required for building the app. During project setup, a file named *app.json* was generated. This file includes crucial metadata, such as the app logo, splash screen image (displayed while the app content loads), API keys (e.g. Google Maps API key), app version and more:



app.json > {} expo

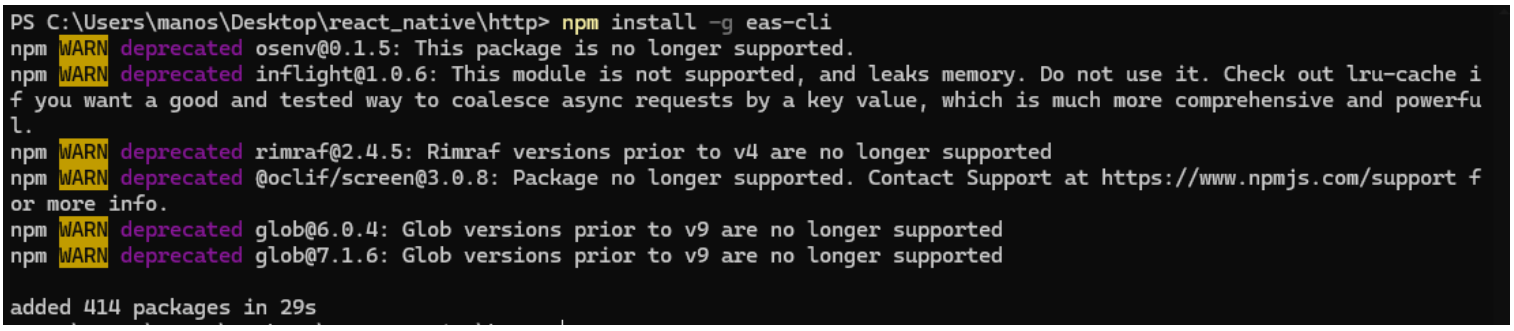
You, 1 hour ago | 1 author (You)

```json
{
  "expo": {
    "name": "Villa-Agapi",
    "slug": "http",
    "version": "2.0.5",
    "orientation": "portrait",
    "icon": "./assets/ios.png",
    "scheme": "myapp",
    "userInterfaceStyle": "automatic",
    "splash": {
      "image": "./assets/logo.png",
      "resizeMode": "contain"
    },
    "assetBundlePatterns": [
      "**/*"
    ],
    "ios": {
      "supportsTablet": true,
      "bundleIdentifier": "com.ekarapiperakis.http",
      "adaptiveIcon": {
        "foregroundImage": "./assets/icon.png"
      },
      "config": {
        "googleMapsApiKey": "[redacted]"
      },
      "buildNumber": "15"
    },
    "android": {
      "versionCode": 14,
      "adaptiveIcon": {
        "foregroundImage": "./assets/icon.png"
      },
      "package": "com.ekarapiperakis.http",
      "config": {
        "googleMaps": {
          "apiKey": "[redacted]"
        }
      }
    },
```

Figure 221 - App.json

11.2 Build Application

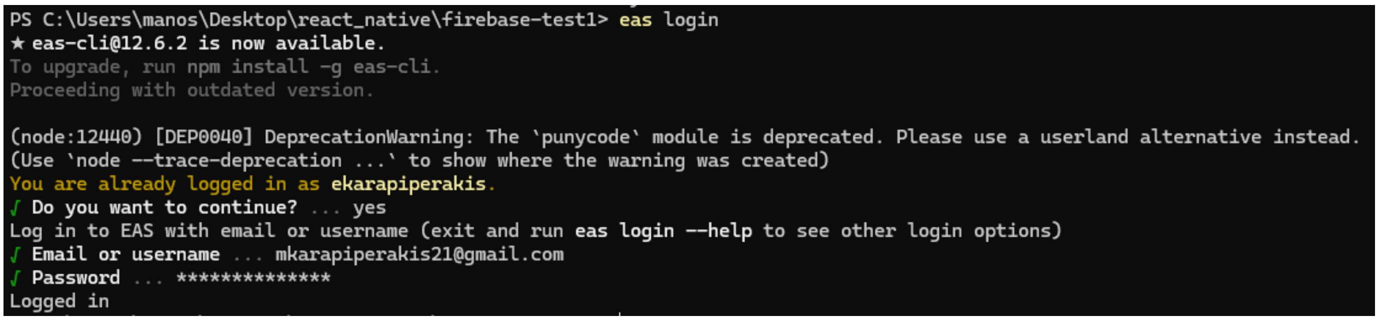
Since the application is built with Expo, the build process will take place on Expo's servers, supporting both iOS and Android. This allows a common build phase for both operating systems, with only the final distribution differing for each app store. As a first step, the command “npm install -g eas-cli" should be run to globally install the Expo Application Services (EAS) CLI:



```
PS C:\Users\manos\Desktop\react_native\http> npm install -g eas-cli
npm WARN deprecated osenv@0.1.5: This package is no longer supported.
npm WARN deprecated inflight@1.0.6: This module is not supported, and leaks memory. Do not use it. Check out lru-cache i
f you want a good and tested way to coalesce async requests by a key value, which is much more comprehensive and powerfu
l.
npm WARN deprecated rimraf@2.4.5: Rimraf versions prior to v4 are no longer supported
npm WARN deprecated @oclif/screen@3.0.8: Package no longer supported. Contact Support at https://www.npmjs.com/support f
or more info.
npm WARN deprecated glob@6.0.4: Glob versions prior to v9 are no longer supported
npm WARN deprecated glob@7.1.6: Glob versions prior to v9 are no longer supported

added 414 packages in 29s
```

Figure 222 - Install eas-cli

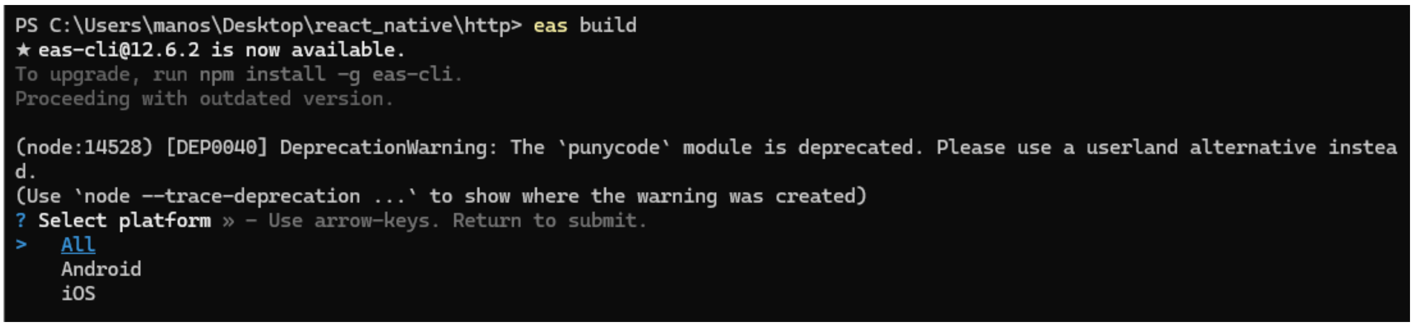
After that the command "eas login" should be given in a terminal and login credentials for the expo should be given:



```
PS C:\Users\manos\Desktop\react_native\firebase-test1> eas login
★ eas-cli@12.6.2 is now available.
To upgrade, run npm install -g eas-cli.
Proceeding with outdated version.

(node:12440) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)
You are already logged in as ekarapiperakis.
√ Do you want to continue? ... yes
Log in to EAS with email or username (exit and run eas login --help to see other login options)
√ Email or username ... mkarapiperakis21@gmail.com
√ Password ... **************
Logged in
```

Figure 223 - Eas login

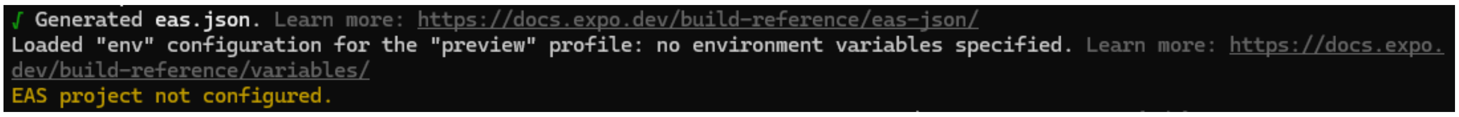
After logging in, the application can be built for the first time by running the command eas build in the terminal and the user should select the desired Platform:



```
PS C:\Users\manos\Desktop\react_native\http> eas build
★ eas-cli@12.6.2 is now available.
To upgrade, run npm install -g eas-cli.
Proceeding with outdated version.

(node:14528) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instea
d.
(Use `node --trace-deprecation ...` to show where the warning was created)
? Select platform » - Use arrow-keys. Return to submit.
>   All
    Android
    iOS
```

Figure 224 - Eas build

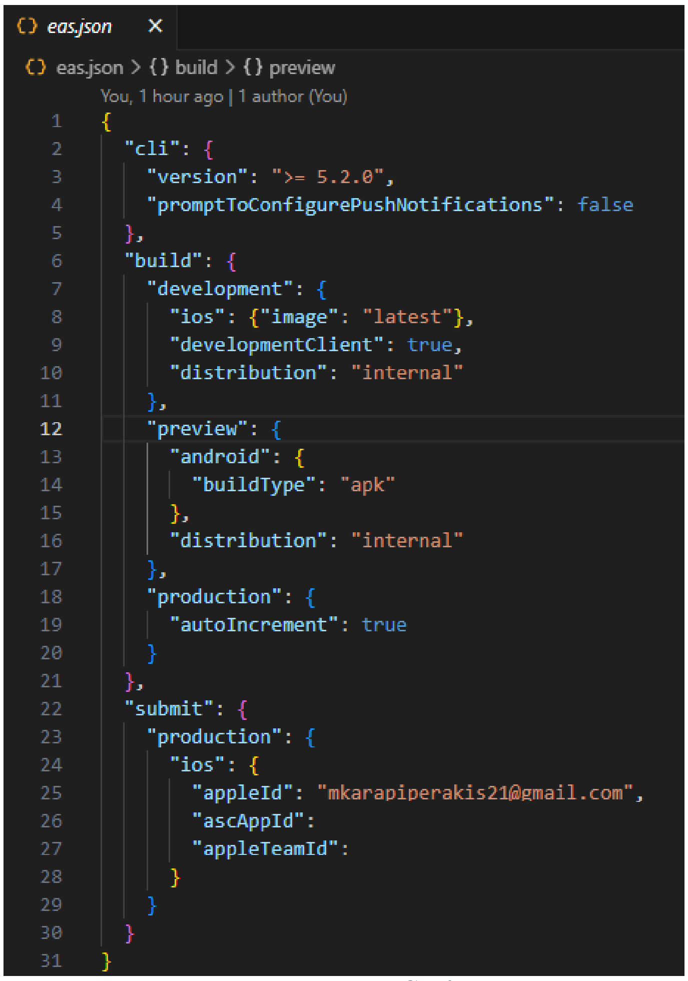
As a result of this process, a new file named *eas.json* will be generated in the project's root directory. This file contains important metadata for the build configuration:



```
√ Generated eas.json. Learn more: https://docs.expo.dev/build-reference/eas-json/
Loaded "env" configuration for the "preview" profile: no environment variables specified. Learn more: https://docs.expo.
dev/build-reference/variables/
EAS project not configured.
```

Figure 225 - Eas.json

This file was configured as shown in the following image:



eas.json ×
eas.json > {} build > {} preview
You, 1 hour ago | 1 author (You)

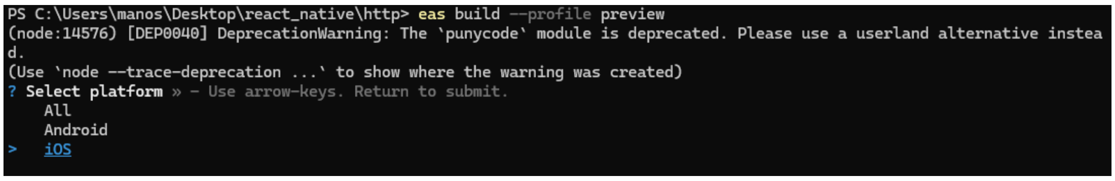
```json
{
  "cli": {
    "version": ">= 5.2.0",
    "promptToConfigurePushNotifications": false
  },
  "build": {
    "development": {
      "ios": {"image": "latest"},
      "developmentClient": true,
      "distribution": "internal"
    },
    "preview": {
      "android": {
        "buildType": "apk"
      },
      "distribution": "internal"
    },
    "production": {
      "autoIncrement": true
    }
  },
  "submit": {
    "production": {
      "ios": {
        "appleId": "mkarapiperakis21@gmail.com",
        "ascAppId":
        "appleTeamId":
      }
    }
  }
}
```

Figure 226 - Eas.json Configuration

The *eas.json* file includes three configured build profiles:

- Development: This profile generates a single executable file compatible with both iOS and Android devices. Users can install this version locally and start the development server by running "npm start" in the terminal while both the mobile device and local machine are connected to the same network. This setup allows for real-time debugging, with logs and errors displayed in the terminal. It is very useful for early testing and internal distribution among other developers. If the development server is not running (i.e. if npm start hasn't been executed), the app will not open.
- Preview: This profile also creates a single executable for both platforms, but it does not require the development server to be running. This version can be distributed to testers for functionality testing. It serves as a pre-production release to validate the app's business requirements rather than testing the code itself.
- Production: This is the final production build, intended for distribution to end users through app stores. For this version, an autoincrement attribute is enabled to automatically increase the app version with each build, ensuring version control for each release.

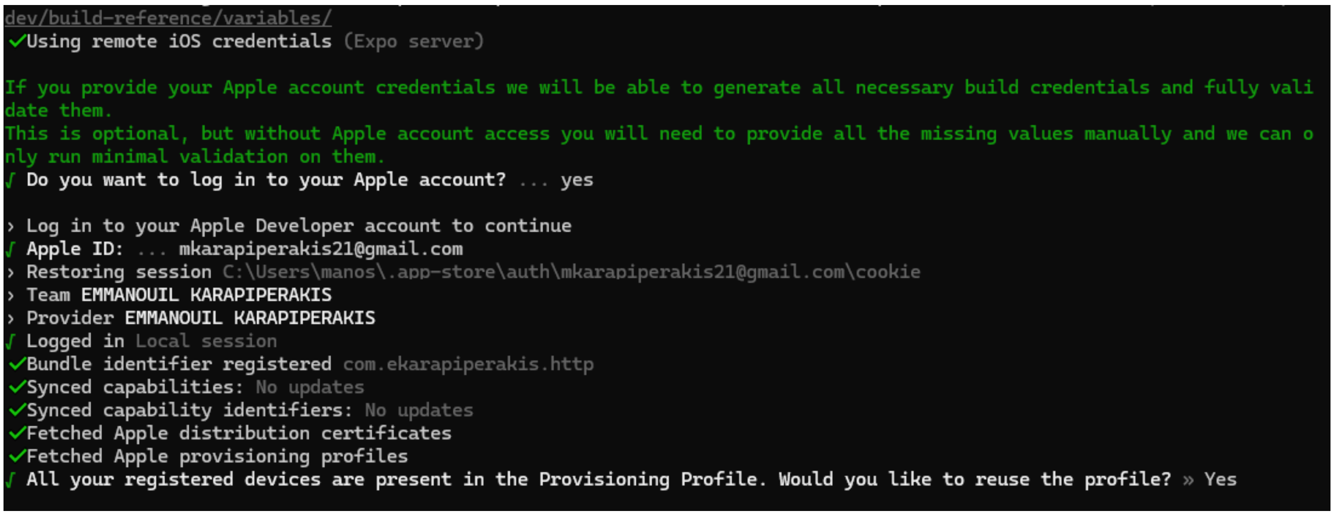
Finally, additional configuration is included in the production profile for iOS to support deployment, such as the required iOS app identifiers. Now the desired profile can be specified in the build command:



```
PS C:\Users\manos\Desktop\react_native\http> eas build --profile preview
(node:14576) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instea
d.
(Use `node --trace-deprecation ...` to show where the warning was created)
? Select platform » - Use arrow-keys. Return to submit.
    All
    Android
>   iOS
```

Figure 227 - Profile Build

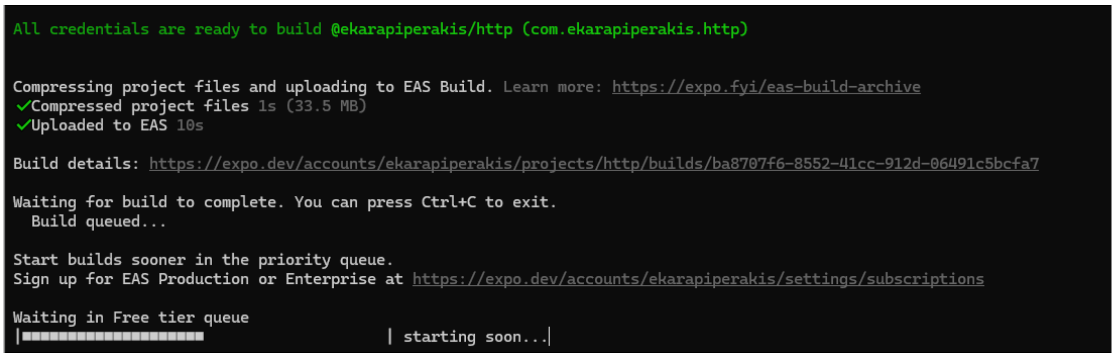
When selecting iOS as the preferred platform, logging into an Apple account is required, as a subscription to the Apple Developer Program is necessary to develop and distribute iOS applications. This program costs 100€ per year and provides a unique identifier linked to the Apple ID, enabling app development and deployment for iOS devices.



```
dev/build-reference/variables/
✓Using remote iOS credentials (Expo server)

If you provide your Apple account credentials we will be able to generate all necessary build credentials and fully vali
date them.
This is optional, but without Apple account access you will need to provide all the missing values manually and we can o
nly run minimal validation on them.
√ Do you want to log in to your Apple account? ... yes

› Log in to your Apple Developer account to continue
√ Apple ID: ... mkarapiperakis21@gmail.com
› Restoring session C:\Users\manos\.app-store\auth\mkarapiperakis21@gmail.com\cookie
› Team EMMANOUIL KARAPIPERAKIS
› Provider EMMANOUIL KARAPIPERAKIS
√ Logged in Local session
✓Bundle identifier registered com.ekarapiperakis.http
✓Synced capabilities: No updates
✓Synced capability identifiers: No updates
✓Fetched Apple distribution certificates
✓Fetched Apple provisioning profiles
√ All your registered devices are present in the Provisioning Profile. Would you like to reuse the profile? » Yes
```

Figure 228 - Build for iOS

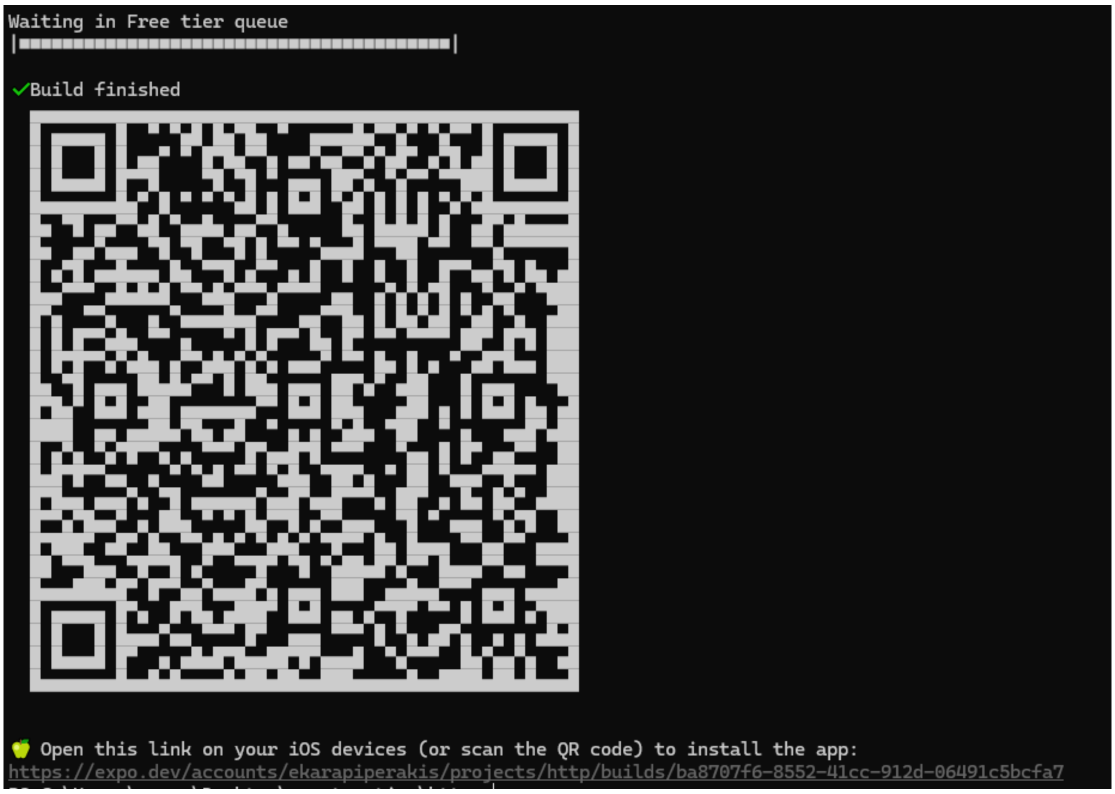
Building for Android is simpler, as there are no restrictions during the build process itself. However, for final distribution to the Google Play Store, a Google Play Console account is required, which involves a one-time registration fee of 15€.
After that, the application will be uploaded to the Expo server. This process is free under the basic package, but it does come with a limit of 30 builds per month and usually waiting in a queue for the build to begin is required.



```
All credentials are ready to build @ekarapiperakis/http (com.ekarapiperakis.http)


Compressing project files and uploading to EAS Build. Learn more: https://expo.fyi/eas-build-archive
 ✓Compressed project files 1s (33.5 MB)
 ✓Uploaded to EAS 10s

Build details: https://expo.dev/accounts/ekarapiperakis/projects/http/builds/ba8707f6-8552-41cc-912d-06491c5bcfa7

Waiting for build to complete. You can press Ctrl+C to exit.
  Build queued...

Start builds sooner in the priority queue.
Sign up for EAS Production or Enterprise at https://expo.dev/accounts/ekarapiperakis/settings/subscriptions

Waiting in Free tier queue
|■■■■■■■■■■■■■■■■■■■■                    | starting soon...
```

Figure 229 - Start Build

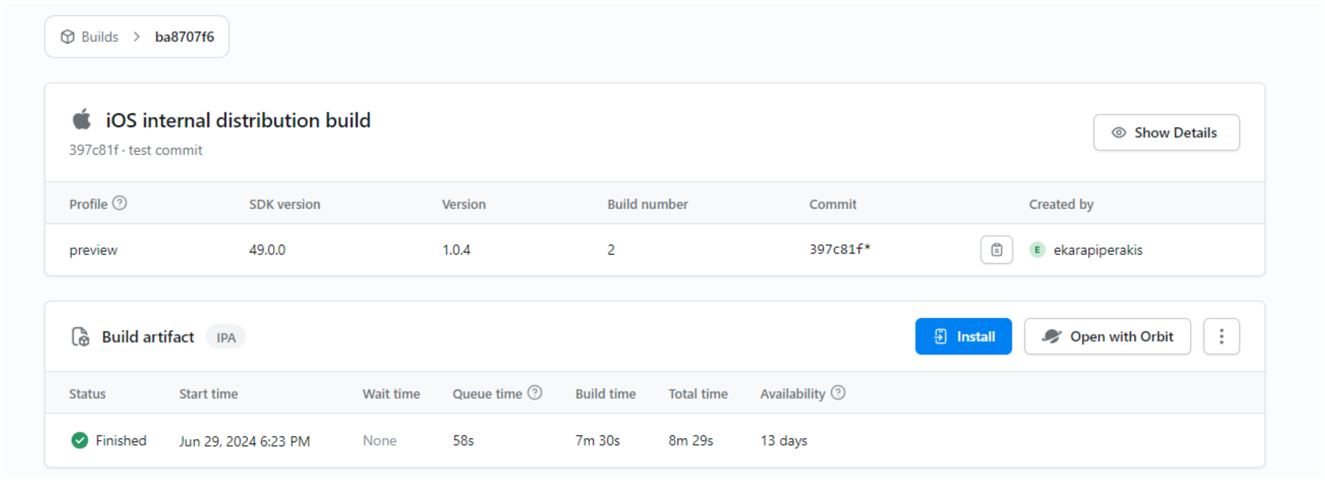
Once the build is complete, a message will appear in the terminal, accompanied by a QR code that can be used to install the built application:



```
Waiting in Free tier queue
|■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■|

✓Build finished

🍏 Open this link on your iOS devices (or scan the QR code) to install the app:
https://expo.dev/accounts/ekarapiperakis/projects/http/builds/ba8707f6-8552-41cc-912d-06491c5bcfa7
```

Figure 230 - Install the App

The user can also navigate to expo, to check more details about the build:



Builds > ba8707f6

iOS internal distribution build

397c81f · test commit

Profile	SDK version	Version	Build number	Commit	Created by
preview	49.0.0	1.0.4	2	397c81f*	ekarapiperakis

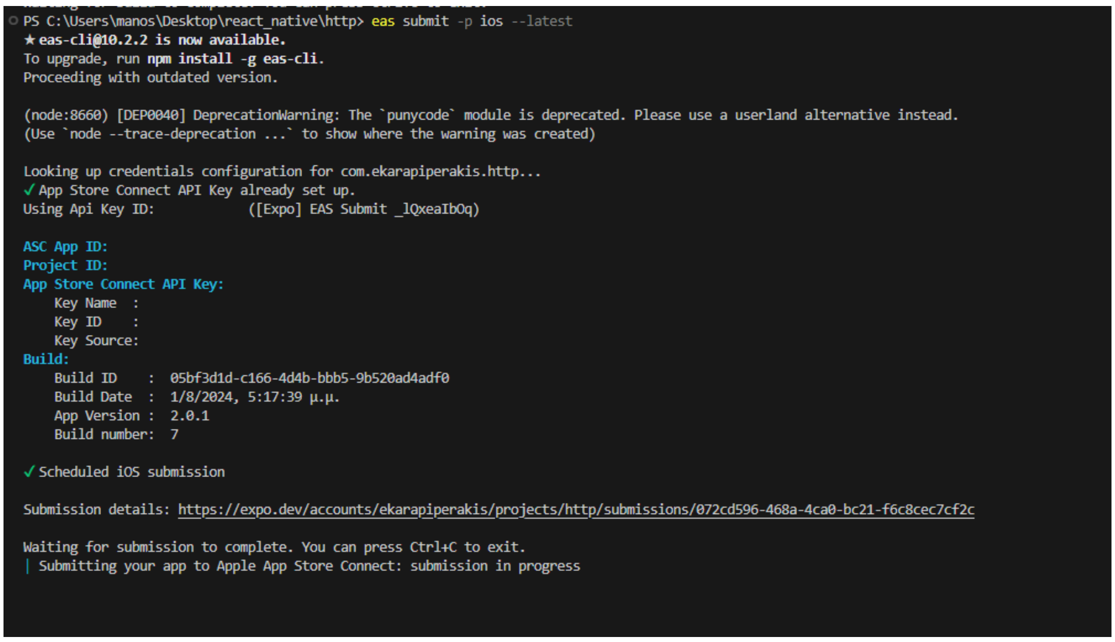
Build artifact IPA

Status	Start time	Wait time	Queue time	Build time	Total time	Availability
Finished	Jun 29, 2024 6:23 PM	None	58s	7m 30s	8m 29s	13 days

Figure 231 - Build Details

11.3 Deployment for iOS

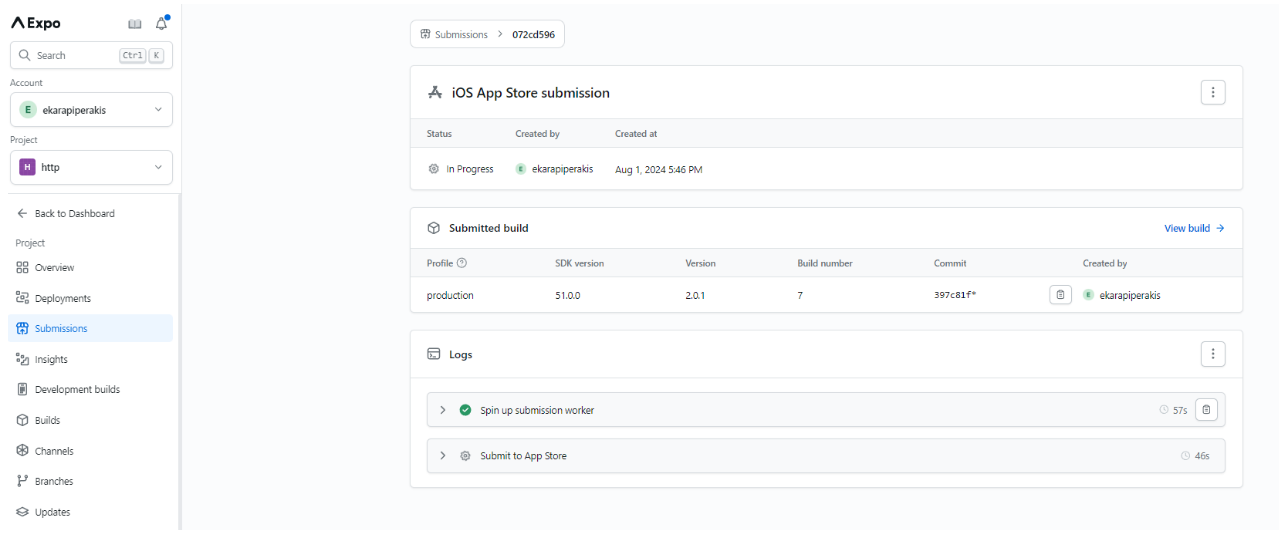
After a succesfull build through the expo, the application can be submited to the app store [35] by running the command “eas submit -p ios --latest”. This command will submit the most recent iOS build from expo:



```
PS C:\Users\manos\Desktop\react_native\http> eas submit -p ios --latest
★ eas-cli@10.2.2 is now available.
To upgrade, run npm install -g eas-cli.
Proceeding with outdated version.

(node:8660) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)

Looking up credentials configuration for com.ekarapiperakis.http...
✓ App Store Connect API Key already set up.
Using Api Key ID:           ([Expo] EAS Submit _lQxeaIbOq)

ASC App ID:
Project ID:
App Store Connect API Key:
    Key Name  :
    Key ID    :
    Key Source:
Build:
    Build ID    :  05bf3d1d-c166-4d4b-bbb5-9b520ad4adf0
    Build Date  :  1/8/2024, 5:17:39 μ.μ.
    App Version :  2.0.1
    Build number:  7

✓ Scheduled iOS submission

Submission details: https://expo.dev/accounts/ekarapiperakis/projects/http/submissions/072cd596-468a-4ca0-bc21-f6c8cec7cf2c

Waiting for submission to complete. You can press Ctrl+C to exit.
| Submitting your app to Apple App Store Connect: submission in progress
```

Figure 232 - Submit iOS Build

If the user returns back to the expo, he can see the execution of this process with more details available:



iOS App Store submission

Status	Created by	Created at
In Progress	ekarapiperakis	Aug 1, 2024 5:46 PM

Submitted build

Profile	SDK version	Version	Build number	Commit	Created by
production	51.0.0	2.0.1	7	397c81f*	ekarapiperakis

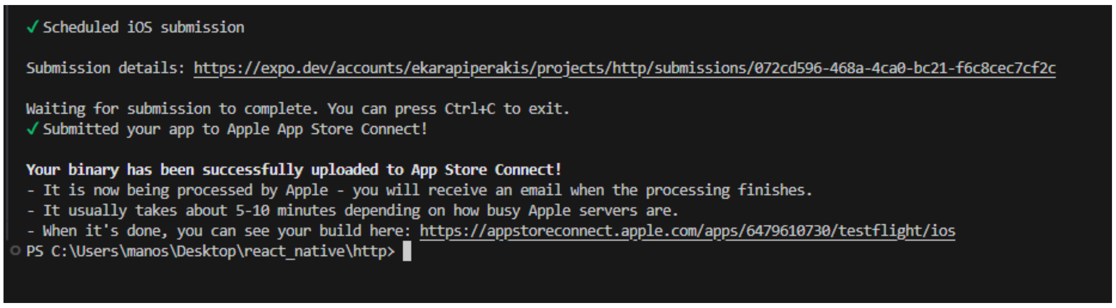
Logs

Spin up submission worker 57s

Submit to App Store 46s

Figure 233 - Expo Submission

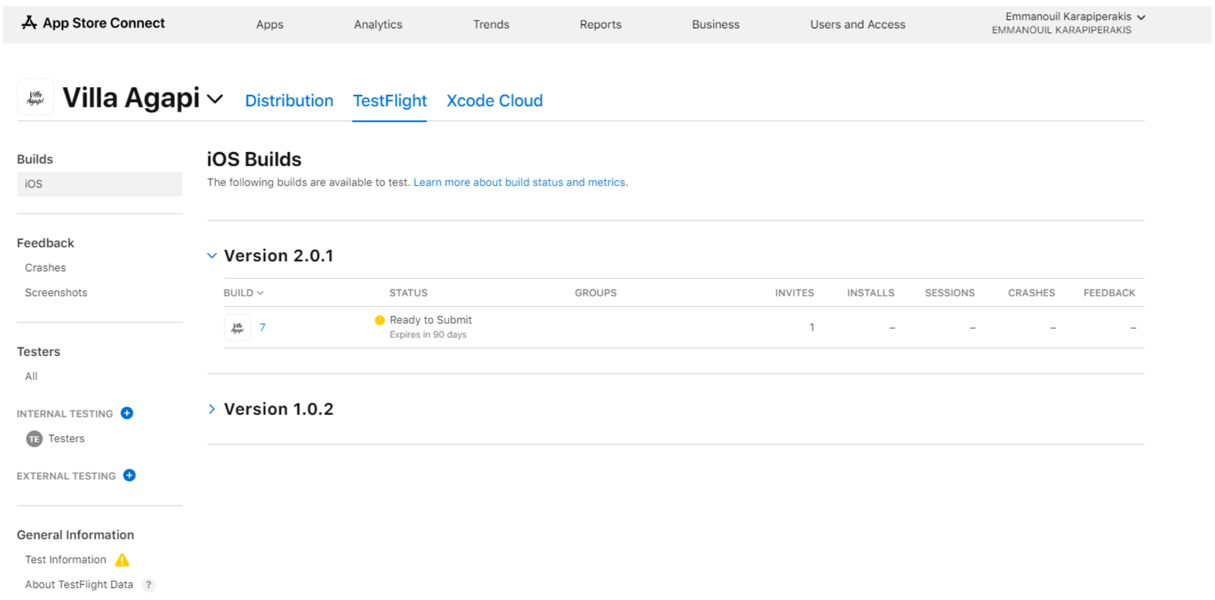
Once the process is complete, the terminal will display an update with the next steps:



```
✓ Scheduled iOS submission

Submission details: https://expo.dev/accounts/ekarapiperakis/projects/http/submissions/072cd596-468a-4ca0-bc21-f6c8cec7cf2c

Waiting for submission to complete. You can press Ctrl+C to exit.
✓ Submitted your app to Apple App Store Connect!

Your binary has been successfully uploaded to App Store Connect!
- It is now being processed by Apple - you will receive an email when the processing finishes.
- It usually takes about 5-10 minutes depending on how busy Apple servers are.
- When it's done, you can see your build here: https://appstoreconnect.apple.com/apps/6479610730/testflight/ios
PS C:\Users\manos\Desktop\react_native\http> █
```

Figure 234 - Build Upload to App Store Connect

Afterward, the user can visit the URL provided upon successful completion of the build upload. In the TestFlight tab, the status of the specific uploaded build version is available. Additionally, from the sidebar, options for internal or external testing are available. While optional for submission, enabling testing is considered a best practice.



iOS Builds

The following builds are available to test. Learn more about build status and metrics.

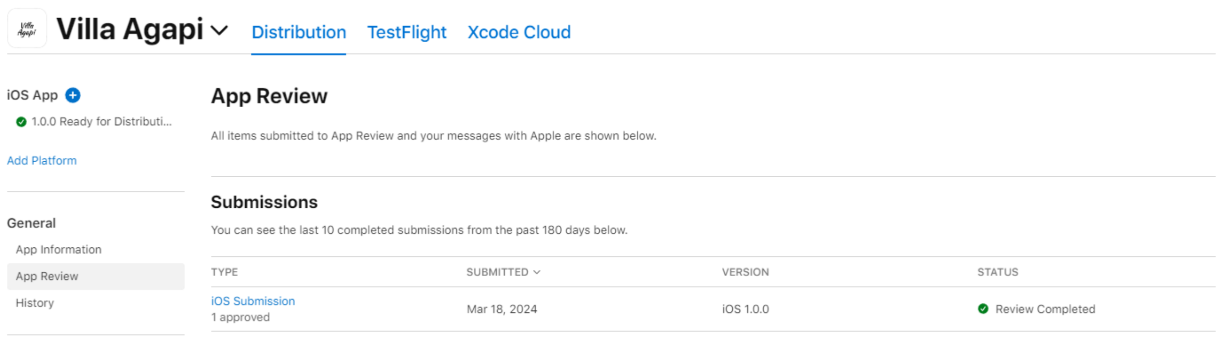
Version 2.0.1

BUILD	STATUS	GROUPS	INVITES	INSTALLS	SESSIONS	CRASHES	FEEDBACK
7	Ready to Submit Expires in 90 days		1	–	–	–	–

Version 1.0.2

Figure 235 - Build Details

Next, in the "Distribution" tab, users can view past submissions and import a new iOS app by clicking the "+" icon located in the top-left corner beneath the project name.



App Review

All items submitted to App Review and your messages with Apple are shown below.

Submissions

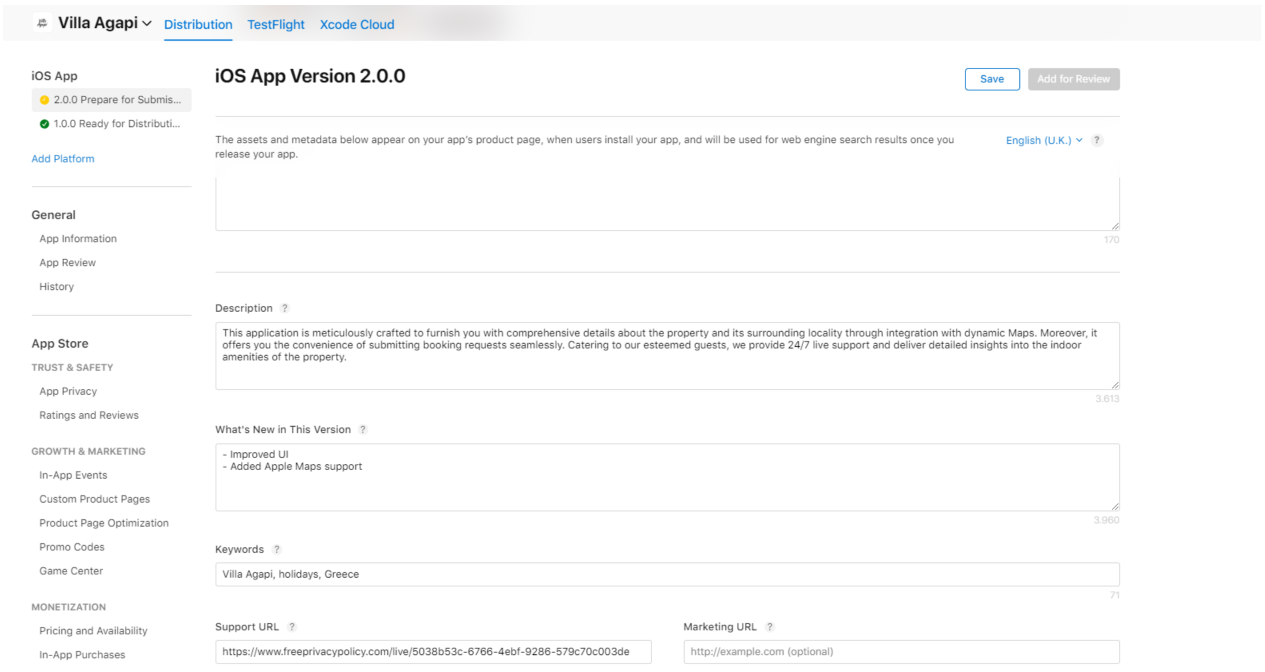
You can see the last 10 completed submissions from the past 180 days below.

TYPE	SUBMITTED	VERSION	STATUS
iOS Submission 1 approved	Mar 18, 2024	iOS 1.0.0	Review Completed

Figure 236 - Submissions Details

From there, additional details should be provided, including:

- Description: A brief overview of the app.
- What's New in This Version: Details about updates or new features in this version.
- Keywords: Keywords to improve search visibility in the App Store.
- Support URL: The URL for user support or privacy information for the application.



iOS App Version 2.0.0

The assets and metadata below appear on your app's product page, when users install your app, and will be used for web engine search results once you release your app.

Description

This application is meticulously crafted to furnish you with comprehensive details about the property and its surrounding locality through integration with dynamic Maps. Moreover, it offers you the convenience of submitting booking requests seamlessly. Catering to our esteemed guests, we provide 24/7 live support and deliver detailed insights into the indoor amenities of the property.

What's New in This Version

- Improved UI
- Added Apple Maps support

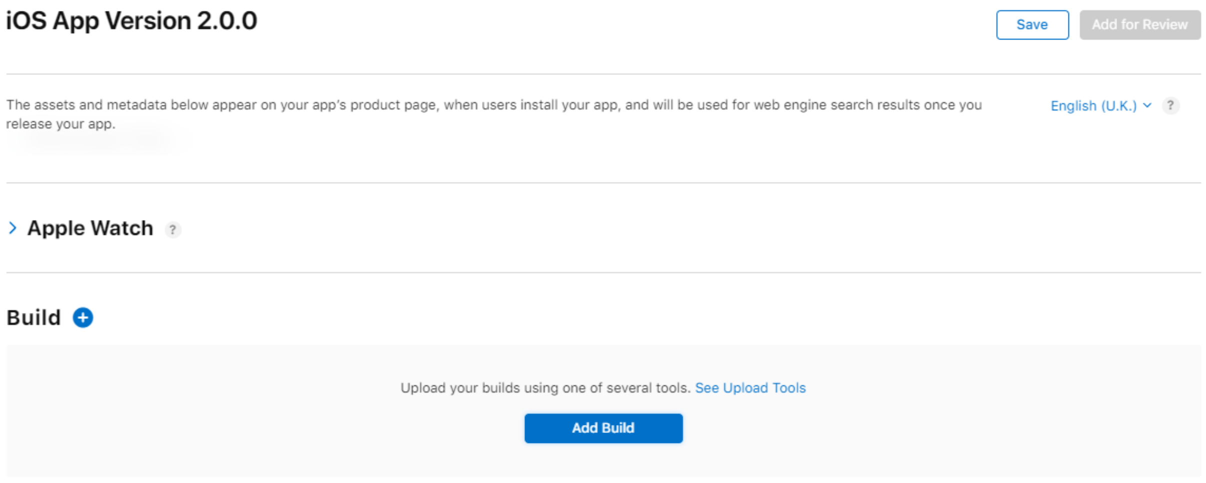
Keywords: Villa Agapi, holidays, Greece

Support URL: https://www.freeprivacypolicy.com/live/5038b53c-6766-4ebf-9286-579c70c003de

Marketing URL: http://example.com (optional)

Figure 237 - New Version Configuration

Finally, by scrolling to the end of this page, the new build should be imported:



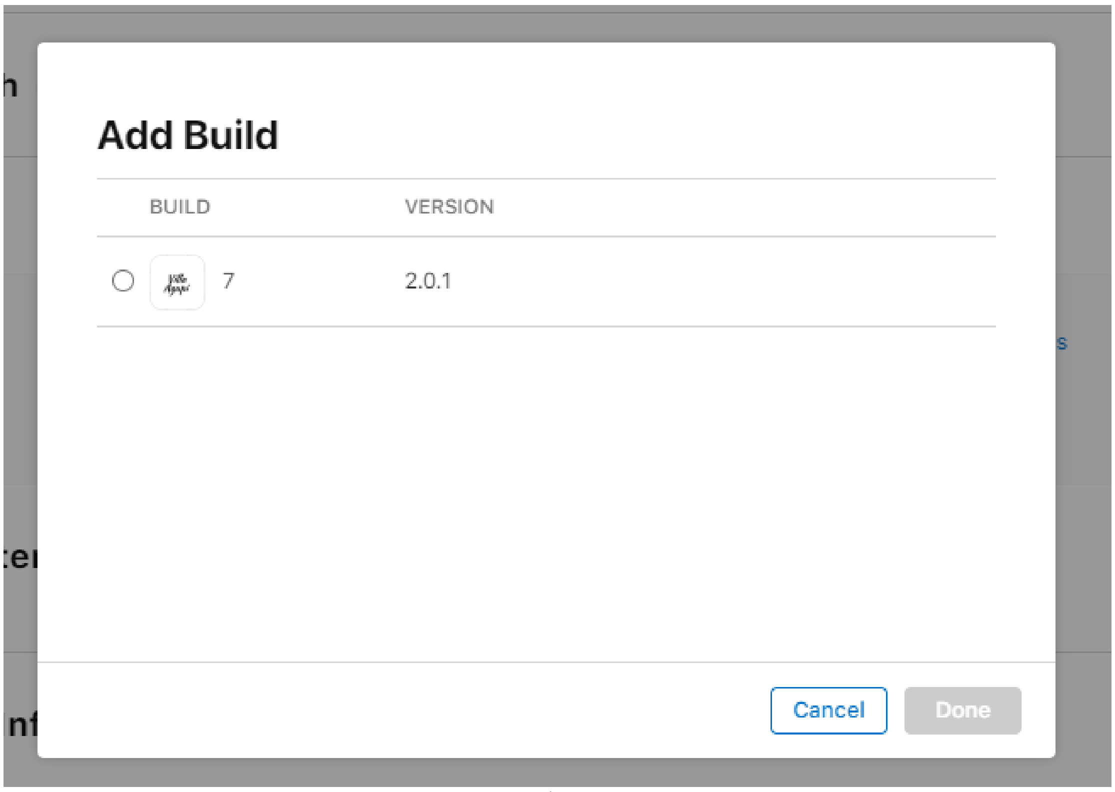
iOS App Version 2.0.0

The assets and metadata below appear on your app's product page, when users install your app, and will be used for web engine search results once you release your app.

Upload your builds using one of several tools. See Upload Tools

Figure 238 - Add Build

By clicking the "Add Build" button, a new modal will pop up, allowing the user to select the appropriate version of the build.



Add Build

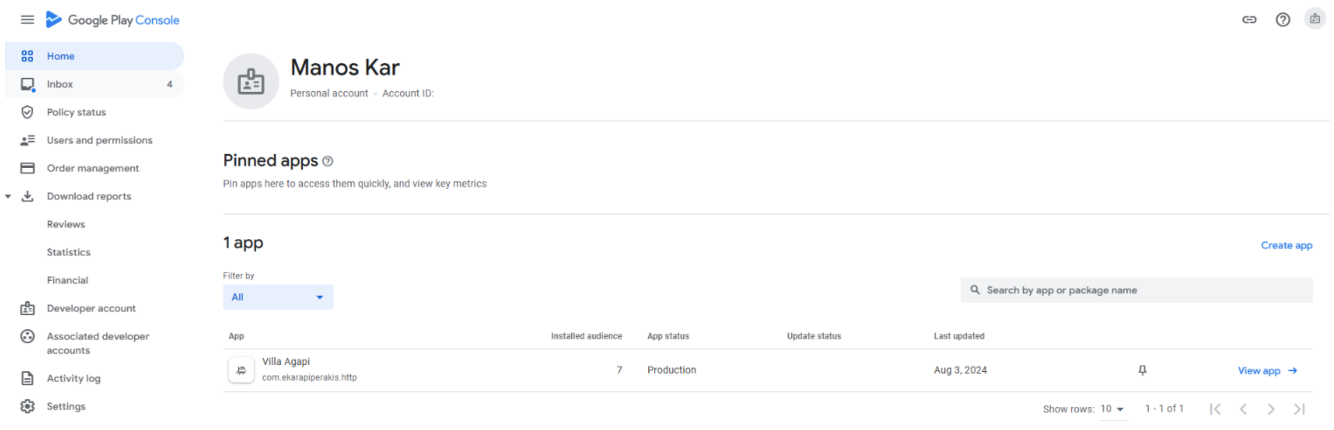
BUILD	VERSION
7	2.0.1

Cancel Done

Figure 239 - Add Build Modal

11.4 Deployment for Android

Deploying to Android differs significantly from iOS. First, a Google Play Console account is required. The app must be manually uploaded for the initial release, after which it can be updated in a similar manner to the iOS submission process [36]. After creating a new account to the Google Play Console, in the dashboard the user can see previouslly created applications:



Manos Kar

Personal account · Account ID:

Pinned apps

Pin apps here to access them quickly, and view key metrics

1 app

Create app

Filter by All

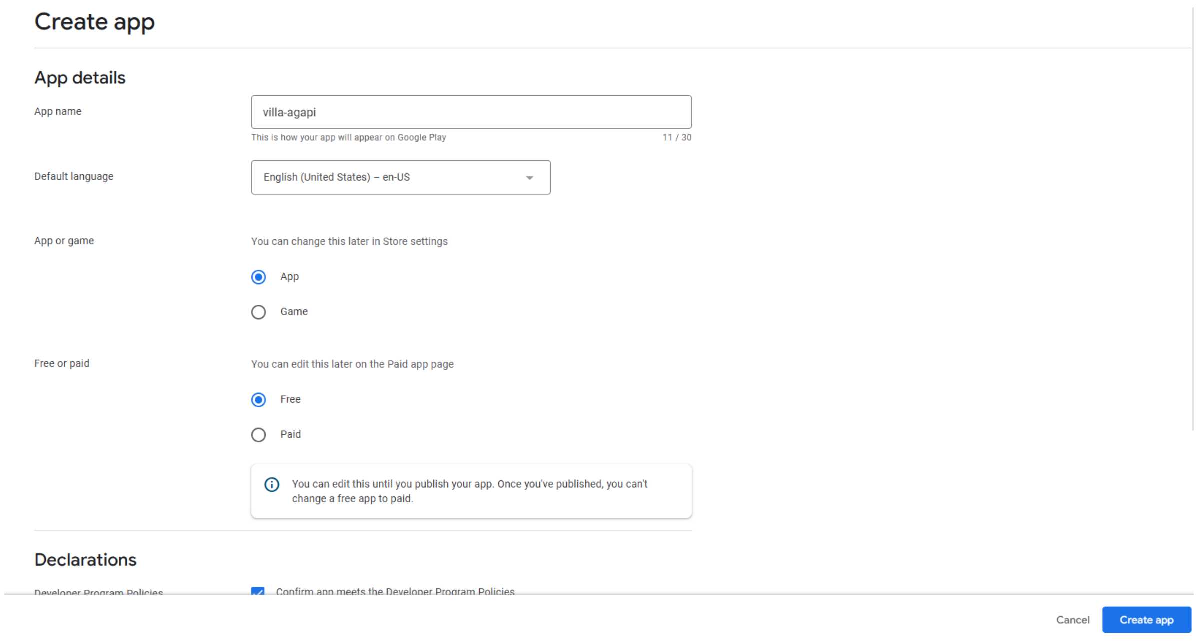
Search by app or package name

App	Installed audience	App status	Update status	Last updated		
Villa Agapi com.ekarapiperakis.http	7	Production		Aug 3, 2024		View app

Show rows: 10 1 - 1 of 1

Figure 240 - Google Play Console

Next, a new application can be created by simply clicking the "Create app" button:



Create app

App details

App name: villa-agapi

This is how your app will appear on Google Play 11 / 30

Default language: English (United States) – en-US

App or game: You can change this later in Store settings

(●) App
( ) Game

Free or paid: You can edit this later on the Paid app page

(●) Free
( ) Paid

You can edit this until you publish your app. Once you've published, you can't change a free app to paid.

Declarations

Developer Program Policies: ✓ Confirm app meets the Developer Program Policies

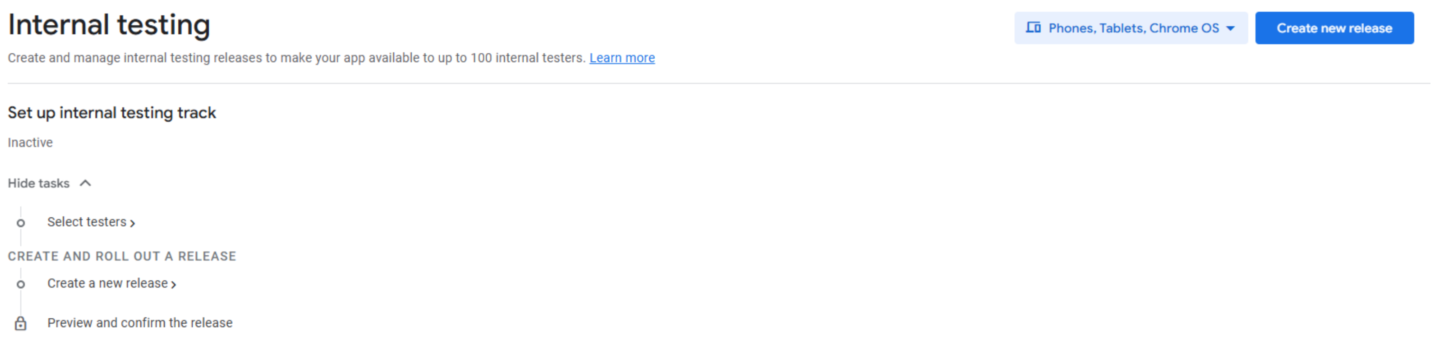
Figure 241 - App Details

In this section, key information is required, including the app name, default language, type (app or game) and pricing (free or paid). A paid app requires a purchase to download, whereas a free app can still offer in-app purchases. Once the application is published, the pricing status (free or paid) cannot be changed.
After creating the application, the following tasks should be completed before the final deployment to production:

- Internal Testing
- App Information
- Closed Testing
- Release to Production

11.4.1 Internal Testing

Before finalizing the app setup, a round of quick testing is required. This involves distributing builds to a small group of trusted testers to identify issues and gather early feedback. The email addresses of these testers should be added to a list and the production application executable (in the .aab format) previously exported from Expo will be uploaded as a new release. Once this is complete, testers will receive an email with a URL to install the app and a link to provide feedback after testing the application.


Internal testing

Create and manage internal testing releases to make your app available to up to 100 internal testers. Learn more

Phones, Tablets, Chrome OS

Create new release

Set up internal testing track

Inactive

Hide tasks

Select testers ›

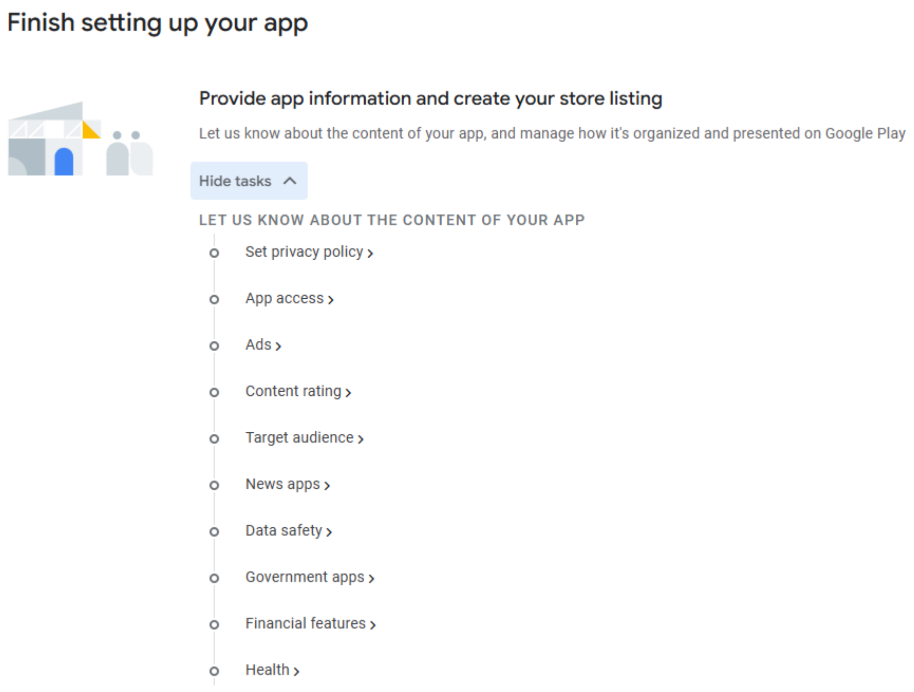
CREATE AND ROLL OUT A RELEASE

Create a new release ›

Preview and confirm the release

Figure 242 - Internal Testing

The next step is to enter essential app information, including the privacy policy, app access settings (available countries), content rating and target audience, among other details. For the privacy policy, a website called "FreePrivacyPolicy" [37] was used to generate a privacy policy document, which can then be shared as a URL.


Finish setting up your app

Provide app information and create your store listing

Let us know about the content of your app, and manage how it's organized and presented on Google Play

Hide tasks

LET US KNOW ABOUT THE CONTENT OF YOUR APP

Set privacy policy ›

App access ›

Ads ›

Content rating ›

Target audience ›

News apps ›

Data safety ›

Government apps ›

Financial features ›

Health ›

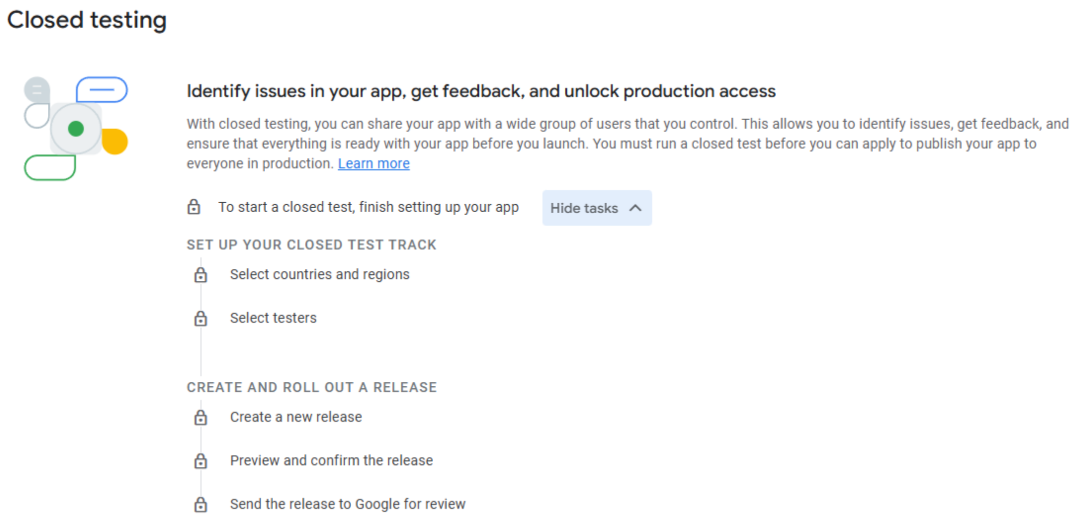
Figure 243 - App Information

The previous step usually takes up to two weeks, as Google reviews all the provided information. Feedback may be given if any part of the app violates their policies, requiring updates before proceeding. Once the review is complete, the app can move on to the next step which is the closed testing.

11.4.2Closed Testing

Closed testing is a critical step before deploying the application to production. It is a required phase where a new release of the app is submitted for review. Google's developers will examine the application, ensuring it complies with their guidelines and policies. As part of this

process, a 3-5 minute demo video of the app must be provided to assist in testing, along with login credentials for different user types.
The review process will verify that the app aligns with Google's data policies and the information provided during the app information setup in the second step. If the app passes the review, it can then be distributed to testers for further evaluation. At least 20 testers are required to install the app and they should remain opted in for a continuous 14-day period of testing. Once this testing period is complete and no major issues are found, the app is ready for deployment to production.


Closed testing

Identify issues in your app, get feedback, and unlock production access

With closed testing, you can share your app with a wide group of users that you control. This allows you to identify issues, get feedback, and ensure that everything is ready with your app before you launch. You must run a closed test before you can apply to publish your app to everyone in production. Learn more

To start a closed test, finish setting up your app   Hide tasks

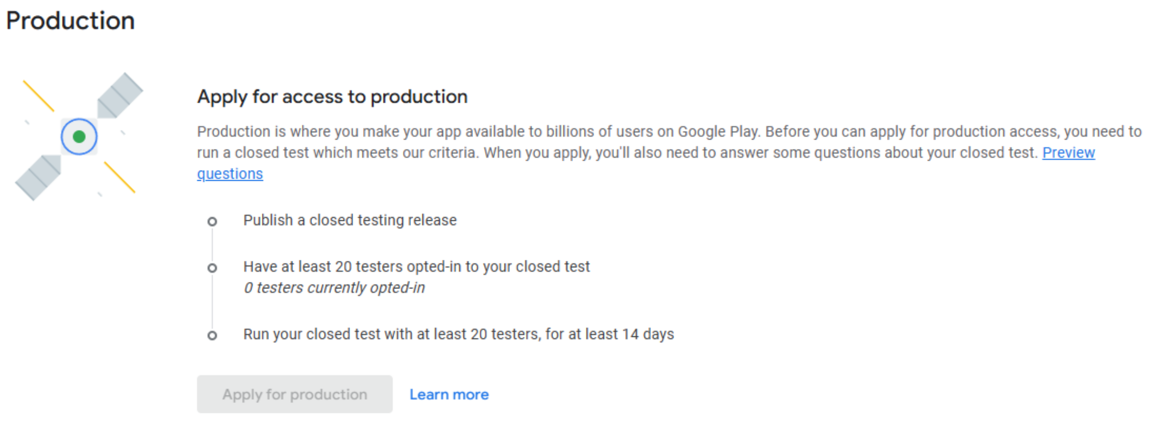
SET UP YOUR CLOSED TEST TRACK
- Select countries and regions
- Select testers

CREATE AND ROLL OUT A RELEASE
- Create a new release
- Preview and confirm the release
- Send the release to Google for review

Figure 244 - Closed Testing

11.4.3Production

To apply for production access, all the previously mentioned steps must be completed. Information about the users who have installed the application during the closed testing phase will be available. This data helps inform the user about the progress and success of the closed testing process.


Production

Apply for access to production

Production is where you make your app available to billions of users on Google Play. Before you can apply for production access, you need to run a closed test which meets our criteria. When you apply, you'll also need to answer some questions about your closed test. Preview questions

- Publish a closed testing release
- Have at least 20 testers opted-in to your closed test
  0 testers currently opted-in
- Run your closed test with at least 20 testers, for at least 14 days

Apply for production   Learn more

Figure 245 - Apply for Access to Production

After successfully passing closed testing and following a few hours of additional review by Google, the application will become available on the Play Store.

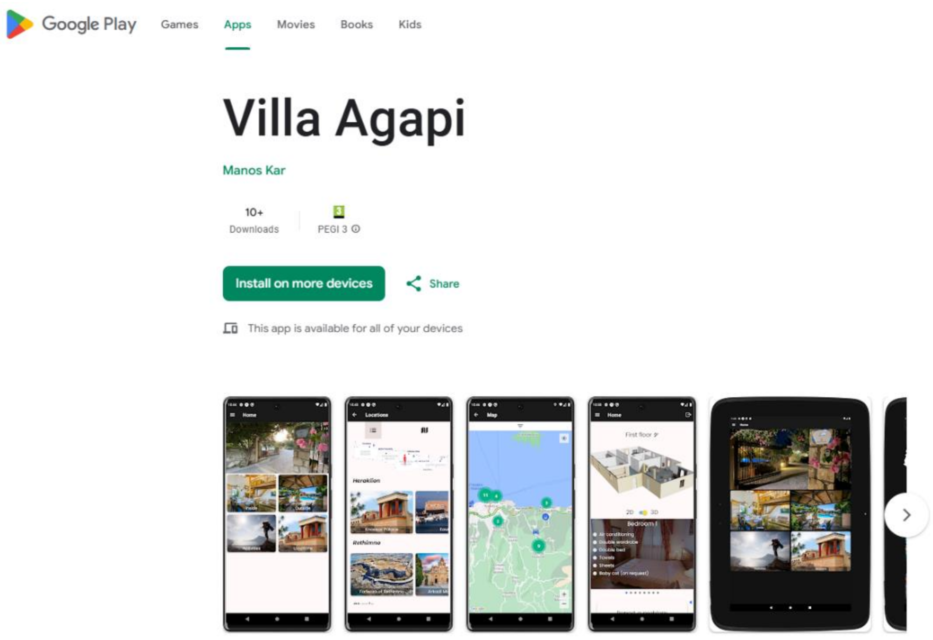


Figure 246 - Play Store

12 Conclusion & Further Development

In conclusion, the analysis of Villa Agapi's needs led to the development of a hybrid, multilingual application for iOS and Android devices, designed to accommodate various screen sizes. The primary objective was successfully achieved: prospective guests can now explore detailed information about the property and its surrounding area, as well as submit direct booking requests to express their interest. Furthermore, the application enhances the guest experience by offering a more personal and welcoming interaction through additional property details, profile management and a live chat feature, enabling direct communication with either a virtual assistant or the host. Finally, the host benefits from a powerful tool to promote the property, with the ability to update dynamic content within the app and effectively respond to client needs.

From a technical perspective, React Native proved to be an excellent choice for implementing the mobile application, enabling a single codebase for both iOS and Android platforms. Additionally, the decision to use JavaScript with Node.js and Express on the backend resulted in a high-performance, scalable and easily maintainable server that effectively supports the mobile app by sharing property resources with users. Socket.io was crucial in enabling real-time communication between guests and the host by handling event-driven messaging for seamless interactions. Finally, Heroku played a valuable role by hosting the Node.js server, Socket.io application and PostgreSQL database, which stores Villa Agapi’s resources. Its user-friendly interface, combined with an easy setup and streamlined release process through pipelines, provided complete control over the backend, significantly contributing to the project’s successful outcome.

Although the primary goal outlined earlier has been achieved, additional features can be implemented to further enhance the application. By analyzing user statistics, including the countries of origin, the host can determine whether translations for additional languages are needed to better serve users. Moreover, the application can benefit from more dynamic content. For instance, an announcement board could be introduced, allowing the host to share updates with users. These updates might include information about local events, file sharing options, or general notices that guests should be aware of during their stay. Lastly, push notifications could be integrated to keep users informed about key events. For example, notifications could remind users about upcoming house cleaning schedules or notify them as their departure day approaches, including a countdown of the remaining days.

13 References

[1] Brajesh De. API Management: An Architect's Guide to Developing and Managing APIs for Your Organization, 1st Edition. Apress, USA, 2017.
[2] Fernando Doglio. Pro REST API Development with Node.js, 1st Edition. Apress, USA, 2015.
[3] GitHub. Architectural Styles and the Design of Network-based Software Architectures,https://github.com/ggdaddy/ebooks/blob/master/Architectural%20Styles%20and%20the%20Design%20of%20Networkbased%20Software%20Architectures.pdf
[4] mdn web docs. HTTP. https://developer.mozilla.org/en-US/docs/Web/HTTP
[5] aws.amazon. What's the Difference Between JSON and XML. https://aws.amazon.com/compare/the-difference-between-json-xml
[6] stackoverflow. Best practices for REST API design. https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design
[7] restfulapi. REST API URI Naming Conventions and Best Practices. https://restfulapi.net/resource-naming
[8] mdn web docs. Cross-Origin Resource Sharing (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
[9] npm docs. Downloading and installing Node.js and npm. https://docs.npmjs.com/downloading-and-installing-node-js-and-npm
[10] Express. Express Fast, unopinionated, minimalist web framework for Node.js. https://expressjs.com/
[11] Semaphore. Best Practices for Securing Node.js Applications in Production. https://semaphoreci.com/blog/securing-nodejs
[12] Auth 0. JSON Web Tokens. https://auth0.com/docs/secure/tokens/json-web-tokens
[13] JWT. JWT Debugger. https://jwt.io/
[14] Specops. How tough is bcrypt to crack? And can it keep passwords safe? https://specopssoft.com/blog/hashing-algorithm-cracking-bcrypt-passwords
[15] Swagger. Why Does API Documentation Matter? https://swagger.io/blog/api-documentation/what-is-api-documentation-and-why-it-matters
[16] Cloudbees. YAML Tutorial: Everything You Need to Get Started in Minutes. https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started
[17] Socket.io. Socket.IO https://socket.io
[18] dev. Integrating a chatbot into your Nodejs API using Dialogflow. https://dev.to/realsteveig/integrating-a-chatbot-into-your-nodejs-api-using-dialogflow-1dpn
[19] React Native. React Native. https://reactnative.dev/
[20] FreeCodeCamp. React Fundamentals – JSX, Components, and Props Explained. https://www.freecodecamp.org/news/react-fundamentals/
[21] Medium. Top 7 State Management Libraries for React Native in 2023. https://medium.com/@thomassentre/top-7-state-management-libraries-for-react-native-in-2023-cd10f461a960
[22] React Native. Networking. https://reactnative.dev/docs/network

[23] W3schools. PostgreSQL - pgAdmin4.
https://www.w3schools.com/postgresql/postgresql_pgadmin4.php
[24] Geeksforgeeks. Basic Authentication in Node.js using HTTP Header.
https://www.geeksforgeeks.org/basic-authentication-in-node-js-using-http-header/
[25] LAMBDATEST. What Is Playwright: A Tutorial on How to Use Playwright. https://www.lambdatest.com/playwright
[26] Medium. How to use i18next, react-i18next in React Native.
https://medium.com/@raazthemystery273/how-to-use-i18next-react-i18next-in-react-native-f81ece184cd2
[27] Expo. @expo/vector-icons@14.0.4. https://icons.expo.fyi/Index
[28] Floorplanner. Space is important Make the most of your space!
https://floorplanner.com/
[29] Heroku Dev Center. Deploying Node.js Apps on Heroku.
https://devcenter.heroku.com/articles/deploying-nodejs
[30] Heroku Dev Center. Guidance for Choosing a Dyno.
https://devcenter.heroku.com/articles/guidance-for-choosing-a-dyno
[31] Liran Tal. Installing Playwright on Heroku for Programmatic Node.js Browser Automation. https://lirantal.com/blog/installing-playwright-on-heroku-for-programmatic-nodejs-browser-automation
[32] Stackoverflow. Using Heroku Scheduler with Node.js.
https://stackoverflow.com/questions/13345664/using-heroku-scheduler-with-node-js
[33] Stackoverflow. What is the preferred Bash shebang ("#!")?
https://stackoverflow.com/questions/10376206/what-is-the-preferred-bash-shebang
[34] Heroku. Continuous Delivery on Heroku. Fundamental concepts, best practices, and tools. https://www.heroku.com/continuous-delivery
[35] Pagepro. How To Publish Expo React Native App to iOS and Android. https://pagepro.co/blog/publishing-expo-react-native-app-to-ios-and-android/
[36] Play Console Help. App testing requirements for new personal developer accounts. https://support.google.com/googleplay/android-developer/answer/14151465
[37] Freeprivacypolicy. Privacy Policy for Villa Agapi
https://www.freeprivacypolicy.com/