DOI: 10.2478/ijasitels-2024-0006

# Vacation Planning Application using API

*Gulean Paula-Florina*
*Computer Science and Electrical and Electronics Engineering Department, Faculty of Engineering, "Lucian Blaga" University of Sibiu, Romania*
*paula.gulean@ulbsibiu.ro*

**Abstract**
Choosing and booking a vacation can be difficult when the application where these things are done is not user-friendly. Therefore, the application that I have created will have to make the user experience much more pleasant. The purpose of the application is to be able to choose a holiday according to the desired filters, with the option to search for a flight in the same period. After booking it, the user who created an account will have stored in his account all the reservations made. Using the logic behind this application, any travel agency can create a new interface that they can customize according to their wishes, but the functionalities remain the same. They can also change what data they want to receive from the API, what data they want to store in their database and in addition modify the offers according to their wishes.

**Keywords**: application, API, MVC

## 1. Introduction

API or Application Programming Interface is a concept to develop applications. With its help, two or more applications can communicate bidirectionally. For example, using the logic behind this vacation planning application and changing only the interface, it can be used by several travel agencies. This greatly facilitates the work of programmers. The implemented API architecture was REST which was introduced in 2004 [1]. The calls that make up the API of the REST architecture are also known as endpoints. In the REST architecture, at an endpoint it is allowed to find only one resource (a user, a schedule list, a hashmap of bank accounts, etc.). An endpoint is characterized by a URI type identifier.[2]
For example : /api/users/ -- this endpoint descriptively specifies that I expect the resource provided by the API to be a list of users
/api/users/1 // -- fetch the user with id 1
Among the purposes of an API, there is also the hiding of the internal information of the way a system works.
Web APIs are a service accessed from the client to a web server via the optional Hypertext Transfer Protocol (HTTP). Clients send a request in the form of an HTTP request (REQUEST) and receive a response (RESPONSE), JavaScript Object Notation (JSON) or Extensible Markup Language (XML – no more practiced) format. Developers typically use APIs in querying a server for a specific set of data on that server.

The word API is often used in reference to web APIs, approving the connection between computers. There are also APIs for programming languages, software libraries, operating systems, and computer hardware. The origins of APIs date back to the 1940s, but the use of the term only appeared in the 1960s and 1970s. Recent developments in the use of APIs have led to the rise in popularity of microservices, which are ultimately loosely coupled services accessed through public APIs.[3]

For this project, I chose the Amadeus API for bringing information. Amadeus contains a selection of APIs for searching, booking and inspiring a trip. By calling endpoints from Amadeus, I was able to load hotel data into the database. To get a key and make an API call, you need to create an account. This API works via a key and secret pair. This data pair is regularly used in API application to obtain an application access token. This principle stops any bot (automated program) from "overloading" the API with unnecessary requests, because each access token must be requested again after a period of time (it has an expiration time usually 30-60 min).[4]

# 2 Theoretical considerations

## 2.1 Development environments

The application was developed in the Visual Studio 2019 programming environment using the C# programming language. This is an integrated programming environment that can be used to create new programs, web or mobile applications.
.Net Framework is a software development environment made by Microsoft. It contains a library of classes called the Framework Class Library (FCL) and provides increased flexibility because each language can use code written in another language. [5] The Framework Class libraries provide the interface part, data access, database connectivity, cryptography, building WEB applications, numerical algorithms and network communications.
When creating an application, a developer combines their source code with the one from the .Net Framework. Entity framework is a framework for Microsoft .NET applications. It allows developers to work with class-specific objects without focusing on the tables where the data is stored.

## 2.2 Programming languages

The programming language used, called C#, is a very well-known and used language by most programmers. It is based on the C language.
HTML or Hyper Text Markup Language is the standard language for creating and structuring web pages. This language comprises a string of elements that tell the browser how the content needs to be designed. The elements are set with the help of tags, which consist in writing them between „<" „>". The name of the element inside a tag is case sensitive, that is, it cannot be written in uppercase letters.[6]
CSS or Cascading Style Sheets describes how HTML elements will be displayed on screen. It saves a lot of time because once created, a CSS file can be used by several web pages at the same time. CSS describes page layout, design and screen size variations.

JSON, i.e. JAVASCRIPT OBJECT NOTATION, is a format that makes it much easier to represent data structures. Json is a much simpler variant than XML. It is based on a subset of the ECMA-262 JavaScript programming language standard. Json is a format that is language-independent, but uses conventions familiar to programmers in the C family of languages. An object is an unordered set of name/value pairs. An object starts with a left brace and ends with a right brace. Each name is followed by a "colon" then the value, and name/value pairs are separated by a comma.[7]

JavaScript (JS) is an object-oriented programming language based on the concept of prototypes. It is mainly used to introduce functionality into web pages, the JavaScript code in these pages being run by the browser. The language is well known for its use in building websites, but it is also used for accessing embedded objects in other applications. It was originally developed by Brendan Eich of Netscape Communications Corporation under the name Mocha, then LiveScript, and finally called JavaScript.[8]

## 2.3 Used during implementation

MVC (Model-View-Controller) is a software architecture model used to implement large-scale projects that can be scaled vertically (improving the current product) and horizontally (generating replicas with the same specifications). As its name suggests, the logic of the program is divided into three interconnected elements.[9]
User requests are directed to the controller. It must communicate with the model to execute and send responses to user requests. The controller orders to display on the view for the user the data he needs.
Components:

- Model- represents the logical scheme of a resource o Image of a table in the database
  - image of a response between client server etc.

- View- has the role of characterizing how the data should be displayed on the page (it doesn't care what the data is)
- Controller- is the one that makes the connection between the model and the view, also having the role of request validator.

**Data Transfer Objects**

DTOs or Data Transfer Objects are objects that transport data between processes in order to reduce the number of method calls. Martin Fowler is the one who implemented this kind of concept. He explained that the purpose of the pattern is to reduce the number of calls to the server by joining multiple parameters into a single call. Also, this principle was introduced to standardize the role of the model as a single SOT (Source of Truth) for the database, thus standardizing the fact that the model represents the outline of a table in the database.[10]

**Readers**

This principle is characterized not so much by design patterns as by the observance of SOLID principles[11]:

- The Single-responsibility principle: “There should never be more than one reason for a class to change.”

● The Open–closed principle : “Software entities ... should be open for extension, but closed for modification.”
● The Liskov substitution principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."
● The Interface segregation principle: "Clients should not be forced to depend upon interfaces that they do not use."
● The Dependency inversion principle: "Depend upon abstractions, [not] concretes."

With the implementation of this principle, all the logic necessary for reading from files was encapsulated in a class, further using an own templating mechanism for generating offers for hotels

**Services**

Classes of this type have the role of providing an encapsulation layer of access to database operations. These classes are the only classes that can provide access to read\write operations thus allowing us to have a single source of truth (single SOT) for the data provided by our database. In Fig. 1 you can see the application of this principle, the actors involved in the communication with the database actually use a service to always get the data, without taking into account exactly the logic of communication with the database. This service actually uses inside the principle called Facade.

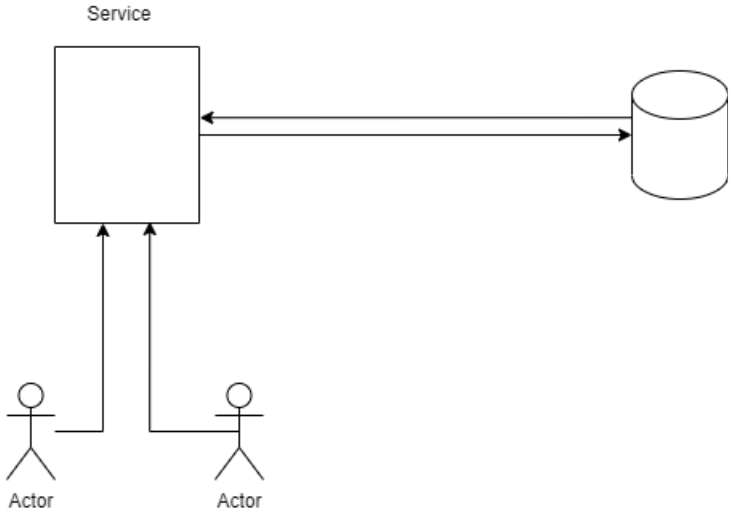


Figure 1 - Services

**Templating**

This is not necessarily a principle so much as a reference to an idea of structuring structural similarities and generating concrete examples based on them.
In the case of my application, I developed an algorithm that, based on some templates, will generate the offers dynamically. Illustrating this facility, I have generated JSON templates for the key features of an offer and JSON templates for the facilities of a location, these templates can be further extended for easy application development.

**Mappers**

Classes of this type allow the extraction of concrete attributes from a data structure with a very large variety of attributes. These classes are meant to allow simplistic implementation at a small development level, allowing the application implementation to be extended by adding new attribute support to the resulting class.
An example can be described by mapping objects in the database to objects of the type of interface that the user sees.

**Security**

To maintain security, I have implemented a security method by which the password is encrypted in the database with an irreversible algorithm (PassWord Bcrypt), using dynamic padding (move the entered password with n random characters to the left or right) and a single jump (a character string that is added to the password to ensure greater dynamism).
Thus, in order to check the integrity of the password saved in the database, the hashes of these passwords will be checked to match.

# 3 Implementation

For the implementation I will present the most important things that were implemented using the theoretical information from previous chapter.

## 3.1 Database

During the development of the application, I decided to use a Mysql database, because it facilitates the relationships between the tables, allows a quick access to the data and accurate filters. In the figure below you can see the structural diagram of the application’s database.

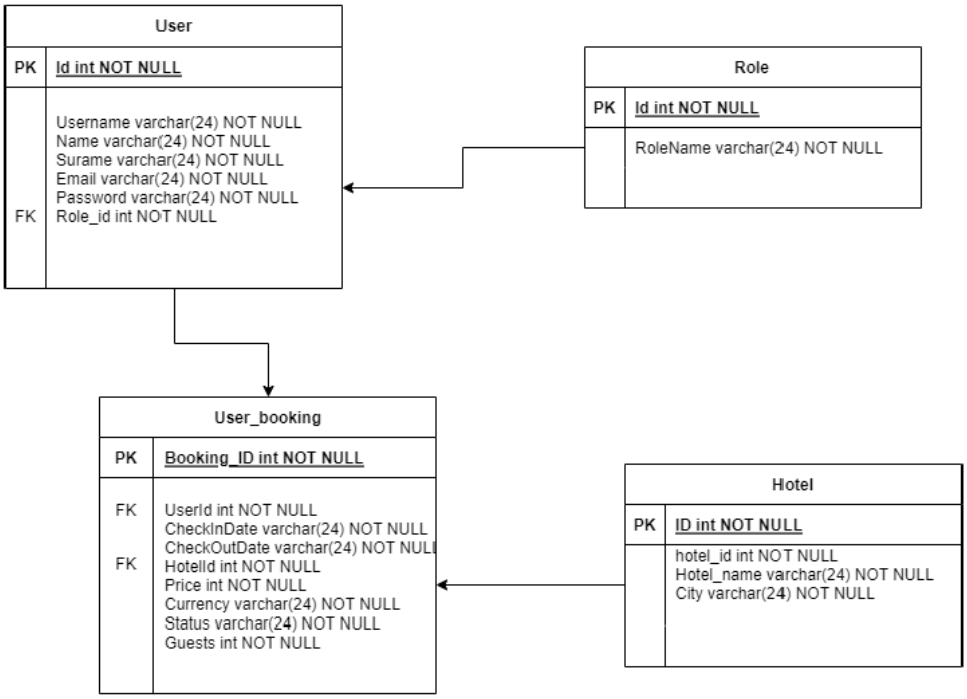


Figure 2 - Database

## 3.2 Architecture of services

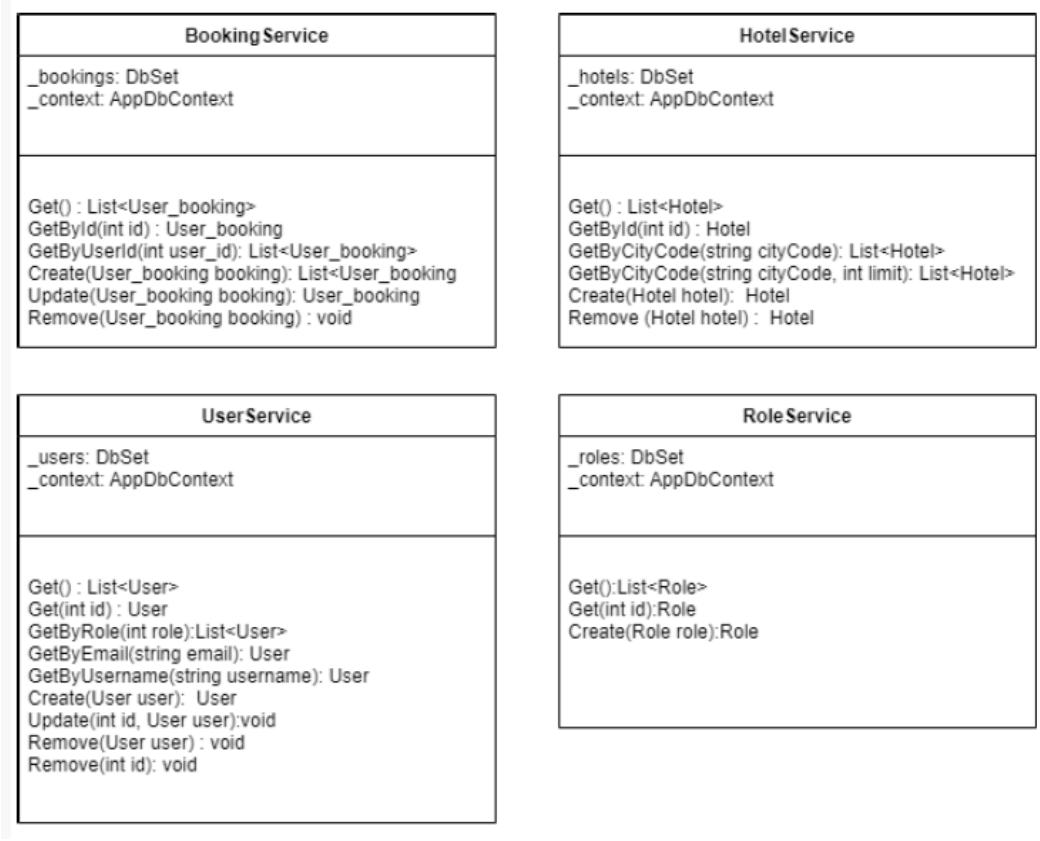


Figure 3 - Services

For example, in Fig. 4 is presented the UserService. It can be seen that it holds a member called users which actually represents the table rows in the database and a context, which is used to update the information in real time. This context will be saved for the safety of keeping the information at its most recent version.
The constructor injects the database context into the current service. Access to the users table will be taken from this context and put into the Users variable, and the context will be saved into the context variable to keep the data up to date.

```
public class UserService
{
    private readonly DbSet<User> _users;
    private readonly AppDbContext _context;

    1 reference
    public UserService(AppDbContext context)
    {
        _users = context.Users;
        _context = context;
    }

    2 references
    public List<User> Get()=>
     _users.Include(user => user.Role).ToList();

    5 references
    public User Get(int id) =>
        _users.Where(user => user.Id == id).Include(user => user.Role).FirstOrDefault();
    1 reference
    public List<User> GetByRole(int role) =>
        _users.Where(user => user.Role_id == role).Include(user => user.Role).ToList();
    0 references
    public User GetByEmail(string email) =>
        _users.Where(user => user.Email == email).Include(user => user.Role).FirstOrDefault();
    1 reference
    public User GetByUsername(string username) =>
        _users.Where(user => user.Username == username).Include(user => user.Role).FirstOrDefault();
```

Figure 4 - UserService

## 3.3 MVC implementation

In the following figures (Fig.5,6,7) I will present the Model, Controller and View for User. All others MVC were represented also using this logic behind.

```
namespace Vacation.Models
{
    26 references
    public class User
    {
        [Column("ID")]
        [Key]
        5 references
        public int Id { get; set; }
        5 references
        public string Username { get; set; }

        3 references
        public string Name { get; set; }

        3 references
        public string Surname { get; set; }

        7 references
        public string Email{ get; set; }
        14 references
        public string Password { get; set; }

        [ForeignKey("Role")]
        2 references
        public int Role_id { get; set; }
        0 references
        public Role Role { get; set; }
    }
}
```

Figure 5 – Model

```
[HttpPost]
0 references
public ActionResult Create([FromForm]User user)
{
    user.Role_id = 2;
    List<User> users = _userService.Get();
    User existringUser = users.Find(i => i.Email == user.Email);
    if (existringUser != null)
        return Ok("Email already in use");
    user.Password = BCrypt.Net.BCrypt.HashPassword(user.Password);
    _userService.Create(user);

    return View("Login");
}

[HttpGet("users-by-role/{roleId}")]
0 references
public ActionResult<List<User>> GetUsersByRole(int roleId)
{
    return _userService.GetByRole(roleId);
}

[HttpPost("login", Name = "Login")]
0 references
public ActionResult<dynamic> Login([FromForm] User user)
{
    User userLogIn = _userService.GetByUsername(user.Username);
    if (userLogIn == null)
        return Ok("Username not found");
    bool verified = BCrypt.Net.BCrypt.Verify(user.Password, userLogIn.Password);
    if (!verified)
        return Ok("Incorect Password");

    HttpContext.Session.SetString("user_name", userLogIn.Username);
    HttpContext.Session.SetString("role", userLogIn.Role.RoleName);
    HttpContext.Session.SetInt32("user_id", userLogIn.Id);
    return RedirectToAction("UserDashBoard");
}
```

Figure 6 - Controller

```
@model User
@{
    ViewData["Title"] = "Login Page";
}

<h4 style="width:100%;text-align:center">Autentifica-te</h4>
<hr />
<div class="row">
    <div class="col-md-4"></div>
    <div class="col-md-4">
        <form asp-controller="User" asp-action="Login">
            <div class="form-group">
                <label asp-for="Username" class="control-label">Nume de utilizator</label>
                <input asp-for="Username" class="form-control" />
                <span asp-validation-for="Username" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password" class="control-label">Parola</label>
                <input asp-for="Password" class="form-control" />
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Autentifica-te" class="btn btn-primary" />
            </div>
        </form>
    </div>
    <div class="col-md-4"></div>
</div>
```

Figure 7 - View

## 3.4. API

This class is used as a communication medium between the developed application and the API implemented by Amadeus.In the constructor of this class it is observed that with the help of the key and secret mechanism Amadeus.builder ("AoWlJ6f5AMkZYcbBAxlrFTSt54uGGheK", "S1apMroKeQ8Bgnml")), a connection to Amadeus will be made. This key and secret mechanism works in the same way as an email and a password, only when authenticating through the client with a key and secret, you will receive back a string of characters that represents the "password" to access Amadeus. This access password is also accompanied by an expiration time of ~30 min. That is, in other words, when it is desired to use Amadeus, the client variable checks if it has a valid Amadeus access password, if it does not have that password or if it has expired, a new one will be requested. Then allowing other methods from the Amadeus API to be called.

```
4 references
public HotelDTOResponse[]? getHotelsByCity(string cityCode) {
    var request_params = Params.with("cityCode", cityCode);
    var path = "/v1/reference-data/locations/hotels/by-city";
    var resp = client.get(path, request_params);
    var jsonObj = resp.result;
    var data = jsonObj["data"].Children();
    List<HotelDTOResponse> hottel_list = new List<HotelDTOResponse>();
    foreach (JToken res in data) {
        hottel_list.Add(res.ToObject<HotelDTOResponse>());
    }
    return hottel_list.ToArray();
}
```

Figure 8 - getHotelsByCity method

The getHotelsByCity method (Fig.8 ) has the role to fetch hotels via the Amadeus API based on a city. In this method, it is set in the request_params variable, in which city

to search for hotels, then a request will be sent to "/v1/reference-data/locations/hotels/by-city" "to get this list.Once a response is received from Amadeus, it will take that response from the data variable and iterate over it to convert the data, which is passed in JSON format, into the format of the HotelDTOResponse class.

```
public class HotelDTOResponse
{
    public string chainCode;
    public string iataCode;
    public string dupeId;
    public string name;
    public string hotelId;
    public dynamic? geoCode = null;
    public dynamic? address = null;
    public dynamic? distance = null;
}
```

Figure 9 - HotelDTOResponse class

For my application, I needed the available hotels in a city. For this I called the endpoint "/v1/reference-data/locations/hotels/by-city" which, depending on the city code provided, brings the required information. These will bring more information about each hotel in the city I searched for. For the implemented application, I chose to use only a part of this information stored with the help of the API and with the help of a mapper, I kept only the desired information.

```
1 reference
public static class HotelMapper
{
    1 reference
    public static Hotel map(HotelDTOResponse hotel_details)
    {
        var hotel = new Hotel();
        hotel.Hotel_id = hotel_details.hotelId;
        hotel.Hotel_name = hotel_details.name;
        hotel.City = hotel_details.iataCode;
        return hotel;
    }
}
```

Figure 10 - HotelMapper class

In Fig. 10 you can see the implementation for the HotelMapper class. This class contains the map method that receives as a parameter an object of type HotelDTOResponse. In the body of this method, an object of the Hotel type will be instantiated and the necessary attributes will be set, and then that object will be returned.

```
public static class OfferMapper
{
    public static Offer map(OfferTemplateDTO templateCity, List<AmenityTemplateDTO> templateAmenities, string guests,
                            string checkInDate, string checkOutDate, AmenitiesCombinations amenity)
    {
        var offer = new Offer();
        offer.CheckInDate = checkInDate;
        offer.CheckOutDate = checkOutDate;
        offer.Currency = templateCity.currency;
        offer.Price = float.Parse(templateCity.price);
        offer.Guests = Int32.Parse(guests);
        List<AmenityTemplateDTO> amenitiesFilter = null;
        if (amenity == AmenitiesCombinations.BASE)
        {
            amenitiesFilter = templateAmenities
                    .FindAll(amenity => amenity.Type == "base");
        }
        else if ((amenity == AmenitiesCombinations.ADVANCED)) {
            amenitiesFilter = templateAmenities
                    .FindAll(amenity => amenity.Type == "base" || amenity.Type == "advanced");
        }
        else if ((amenity == AmenitiesCombinations.PREMIUM))
        {
            amenitiesFilter = templateAmenities
                    .FindAll(amenity => amenity.Type == "base" || amenity.Type == "advanced" || amenity.Type == "premium" );
        }
        offer.Amenities = amenitiesFilter.Select(amenity => amenity.Name).ToList();
        return offer;
    }
}
```

Figure 11 - OfferMapper class

In the figure illustrated above (Fig. 11) you can see the implementation of the OfferMapper class. This class contains the map method that receives as parameters the details necessary to generate an offer. In the body of the method, an object of type Offer will be instantiated and members of this instance will be set. At the end, the instance of the generated offer will be returned.

# 4. Conclusions

The application allows choosing and booking a holiday, letting the user choose their favorite destination, the desired period and the number of accompanying people. In addition to this, after logging in, he is able to see all his created bookings, from his account.
The administrator can see the bookings created by any user at any time.
Another useful thing is the fact that hotel data is brought via an API.
The application can be developed in the future by adding an option (a check-box) that by ticking will also bring offers for air transport to the desired destination. Thus, at the end the user will receive a complete offer, which he can reserve.
Another thing that could be implemented is the addition of a possibility to pay by card, so the administrator will receive a confirmation of the reservation created, by paying the customer. Also, a useful thing could be receiving a holiday confirmation on the user's email.
Because the information is fetched with the API, we can always change what we want to display depending on what information we can get through it.

# References

[1] REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces, Mark Masse, O'Reilly Media, Inc., 2011
[2] https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
[3] https://en.wikipedia.org/wiki/API
[4] https://developers.amadeus.com/
[5] https://en.wikipedia.org/wiki/.NET_Framework
[6] https://developer.mozilla.org/en-US/docs/Web/HTML

[7] https://www.json.org/json-ro.html
[8] https://en.wikipedia.org/wiki/JavaScript
[9] ASP.NET MVC Framework Unleashed, Stephen Walther,Sams Publishing, 2009
[10] https://www.baeldung.com/java-dto-pattern
[11] https://en.wikipedia.org/wiki/SOLID