

Table of Contents

1 – Introduction	3
2 – Project Requirements Recap	3
2.1 – User Requirements.....	3
2.2 – Technical Requirements	3
3 – Development Environment Setup Recap	4
3.1 – Tools and Technologies.....	4
3.2 – Installation Instructions	4
4 – Implementation of API Integration	5
4.1 – HTTP Request Implementation	5
4.1.1 – Description of HTTP Requests.....	5
4.1.2 – Code Snippets.....	5
4.1.3 – Error Handling	7
4.2 – Response Reception and Handling.....	8
4.2.1 – Description of Response Handling.....	8
4.2.2 – Code Snippet	9
4.2.3 – Data Parsing.....	9
4.3 – Display of Response	9
4.3.1 – User Interface Design.....	10
4.3.2 – Code Snippet	11
4.4 – Code Quality and Documentation	12
4.4.1 – Structure of the Code.....	12
4.4.2 – Comments and Documentation	12
5 – Testing and Validation	13
5.1 – Testing Approach	13
5.2 – Test Cases	13
5.3 – Results	14
5.4 – Bugs Fixes and Iterations	14
6 – Reflections and Lesson Learned	14
6.1 – Quality and Maintenance.....	14
6.2 – Key Learning Points	15
6.3 – What could we have done differently	15
7 – Conclusion.....	15
8 – References.....	16

List of Table

Table 1 - Test Cases	14
Table 2 - Contribution to the Report	Error! Bookmark not defined.

List of Figures

Figure 1 - Finding Hotel Reviews.....	6
Figure 2 - Fetching Hotel Photos.....	7
Figure 3 - Error Handling.....	8
Figure 4 - Search Hotels	9
Figure 5 - User Interface Design	10
Figure 6 - User Interface Coding Design for Search Hotel.....	11
Figure 7 - Program.cs – Comments.....	12
Figure 8 - HotelSearch Comments	12

1 – Introduction

Our API integration project is fully detailed in the included report. A key initial step in providing dynamic, real-time information inside an application, this report will go over our strategy of including the specified web API into a C# program. This is a vital interface since it allows the program to access live external feeds, hence offering functionality including hotel searches and booking.

The report details each tool and framework used to build up the development environment and the particular demands that drove us throughout the project, taking user and technical factors into account. Additionally explored will be the error handling and data parsing carried out to ensure the dependability of the solution and the HTTP request and response processing. We will stress the results of the tests that demonstrate the validity of the implementation. Finally, we will have a conclusion and references to the materials utilised during the project.

2 – Project Requirements Recap

This section will outline the fundamental prerequisites for constructing our console-based application. First, we must recall user requirements from Task 2 as we focus on characteristics required for a seamless and productive user experience. Following that, we will discuss the technical needs that enable appropriate operation, data management, and interaction with the Booking.com API using RapidAPI.

2.1 – User Requirements

Task 2 entailed establishing the major user demands that would guide us through the building of our application employing the .NET Framework console. The user primarily wants the ability to search for rooms based on location, with the possibility of applying filters based on price, rating, and availability. The application should also provide data about the preferred lodgings, including transportation options. They also want it to be easy to book flights and taxis using the app, with speedy results and simple input of trip data.

2.2 – Technical Requirements

Task 2's technical requirements must provide these user capabilities while also integrating a robust API, especially the Booking.com API via RapidAPI, into our .NET Framework console application. It requires the safe processing of HTTP requests and answers, as well as the parsing and presentation of JSON data, all while preserving a smooth user experience via data organisation and proper error management. This requires having a systematised directory for capturing and storing sensitive information like as payment details, as well as measures to keep users' data secure inside the console itself.

3 – Development Environment Setup Recap

In this section, we offer an overview of the tools, technologies, and settings essential to develop this console-based .NET Framework application. This, we feel, will be crucial to familiarise readers with what they will want for the correct implementation and integration of the Booking.com API via RapidAPI.

3.1 – Tools and Technologies

To facilitate the development of this application, we utilised the following tools and frameworks.

- **C#:** The major programming language applied in our application to appropriately handle APIs and edit data.
- **.Net Framework:** It was picked because it gives a stable and high-performance environment in which to construct the console application.
- **Visual Studio:** This refers to the integrated development environment (IDE) used for code generation, debugging and testing.
- **RapidAPI:** This offers as a one-stop shop for simple integration of Booking.com APIs, including important functionality such as hotel search and room availability, as well as transportation possibilities.
- **JSON:** This format is used to read and process API replies, as well as to represent structured data.
- **README.md:** A markdown file may be used to offer the project's final documentation.
- **appsettings.json:** This is a configuration file that maintains application settings such as API keys, among others.

3.2 – Installation Instructions

The following commands are required to set up the development environment and setup the project:

1. Install Visual Studio: It may be obtained on the official website. During the setup, check the .NET Desktop Development workload since it will be required later to work on the framework.

2. Create the project:

- Open Visual Studio and then click Create a new project. In the list of project types, pick Console App (.NET Framework).
- Name your project, specify a location, and then click Create.

3. Installation Packages Required:

- Open Visual Studio and open the NuGet Package Manager to add any packages required for JSON processing. You may hire a library such as Newtonsoft.Json, which processes JSON data exceptionally rapidly.

4. Add project files and structure:

- Create two major folders in the project's root: Models and Services.
- Include the following files in the Models: HotelData.cs, HotelDetails.cs, HotelPhotos.cs, HotelReview.cs, HotelSearch.cs, RoomAvailability.cs, SearchFlights.cs, and SearchTaxi.cs. Create the BookingService.cs file in the Services directory.
- The program.cs, README.md, and appsettings.json files are created in the project's root.

5. API Access Setup:

- In appsettings.json, insert your RapidAPI key and other API settings to allow secure access to the Booking.com API.

6. Compile and run:

- Make sure that all parameters are adequate, then build and run the program from Program.cs to validate if the basic setup works as planned.

4 – Implementation of API Integration

The following section covers in detail how our application's API was developed. We'll look at the different HTTP queries accessible when dealing with the Booking.com API, identifying the request type, which is a POST request, and why it was utilised. A code sample for each of these queries will also be shown to explain how the input parameters are handled and how error handling mechanisms are implemented to interact robustly and reliably with the API.

We next go over how the app receives and processes API replies, including data processing and the structures that handle the provided data. It will also cover how the answer is shown to the customers; thus, focus will be made on the design of the user interface as well as user experience considerations for better app working. Finally, code structure and documentation will be examined, with the key emphasis being on clarity and maintainability during development.

4.1 – HTTP Request Implementation

This section illustrates how the Booking Console Application handles HTTP requests. An explanation of the types of requests used would be offered, along with how they were chosen to help fulfil the application's demands. Code snippets for the request methods that handle the arguments and retrieve the data will soon be given. Finally, we analyse the error-handling mechanisms that have been implemented in these HTTP exchanges to boost their durability.

4.1.1 – Description of HTTP Requests

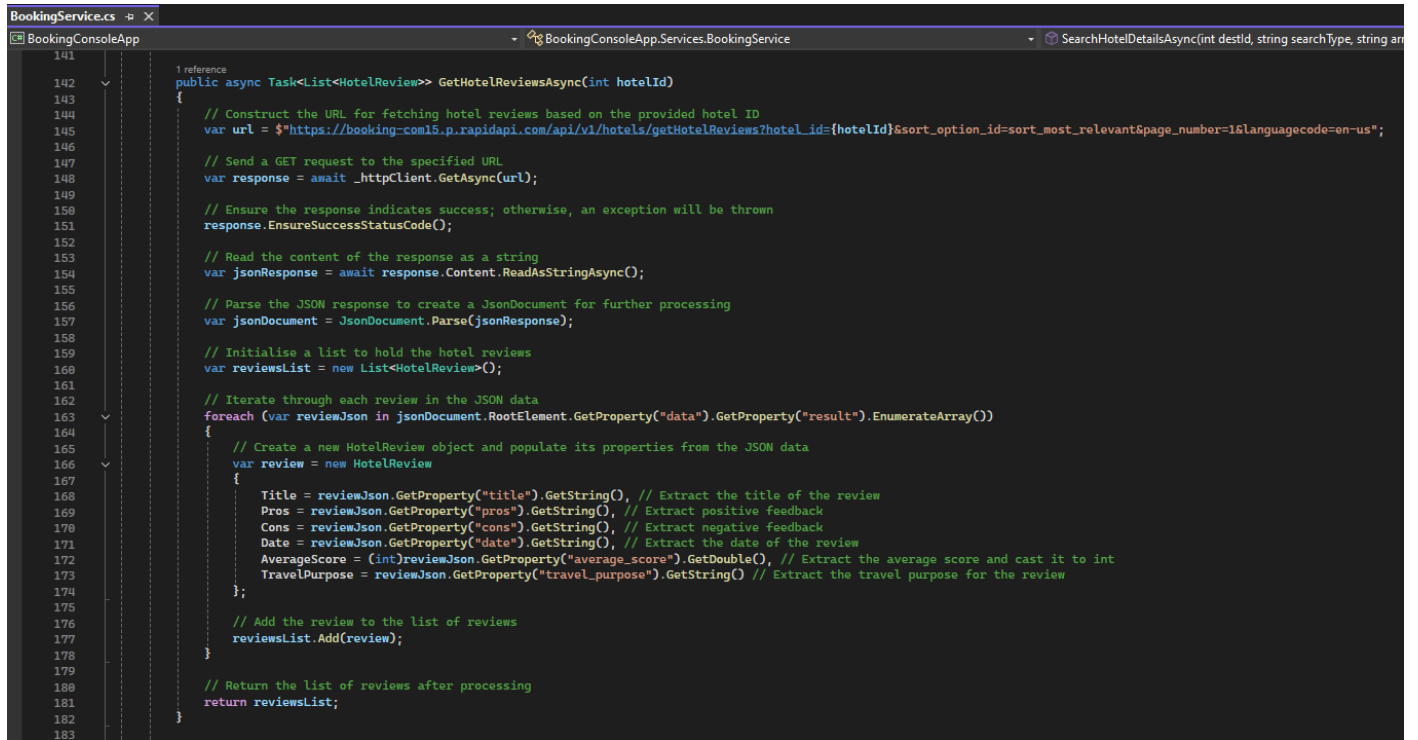
The most popular kind of HTTP request performed in this project is GET. This is most fit for the nature of the Booking Console Application, since the apps often acquire data such as hotel ratings, photographs of destinations, and transportation alternatives via the Booking.com API. The program is supposed to deliver information to the user; consequently, GET requests are allowed in this context since they collect data without causing any alterations on the server. Each GET request helps the API's architecture in some fashion by supplying specified data upon request, allowing for clear and speedy retrieval that fulfils users' search and selection criteria.

4.1.2 – Code Snippets

The following major code snippets describe the implementation of GET requests inside BookingService.cs of the Booking Console Application:

1. Finding Hotel Reviews

The following code snippet seeks hotel reviews. It sets a URL with the ID of a hotel and seeks reviews. The URL request is dynamically constructed and includes the required query parameters. The JSON response is then evaluated to obtain individual review data.



```

141
142 1 reference
143 public async Task<List<HotelReview>> GetHotelReviewsAsync(int hotelId)
144 {
145     // Construct the URL for fetching hotel reviews based on the provided hotel ID
146     var url = $"https://booking-com15.p.rapidapi.com/api/v1/hotels/getHotelReviews?hotel_id={hotelId}&sort_option_id=sort_most_relevant&page_number=1&languagecode=en-us";
147
148     // Send a GET request to the specified URL
149     var response = await _httpClient.GetAsync(url);
150
151     // Ensure the response indicates success; otherwise, an exception will be thrown
152     response.EnsureSuccessStatusCode();
153
154     // Read the content of the response as a string
155     var jsonResponse = await response.Content.ReadAsStringAsync();
156
157     // Parse the JSON response to create a JsonDocument for further processing
158     var jsonDocument = JsonDocument.Parse(jsonResponse);
159
160     // Initialise a list to hold the hotel reviews
161     var reviewsList = new List<HotelReview>();
162
163     // Iterate through each review in the JSON data
164     foreach (var reviewJson in jsonDocument.RootElement.GetProperty("data").GetProperty("result").EnumerateArray())
165     {
166         // Create a new HotelReview object and populate its properties from the JSON data
167         var review = new HotelReview
168         {
169             Title = reviewJson.GetProperty("title").GetString(), // Extract the title of the review
170             Pros = reviewJson.GetProperty("pros").GetString(), // Extract positive feedback
171             Cons = reviewJson.GetProperty("cons").GetString(), // Extract negative feedback
172             Date = reviewJson.GetProperty("date").GetString(), // Extract the date of the review
173             AverageScore = (int)reviewJson.GetProperty("average_score").GetDouble(), // Extract the average score and cast it to int
174             TravelPurpose = reviewJson.GetProperty("travel_purpose").GetString() // Extract the travel purpose for the review
175         };
176
177         // Add the review to the list of reviews
178         reviewsList.Add(review);
179     }
180
181     // Return the list of reviews after processing
182     return reviewsList;
183 }

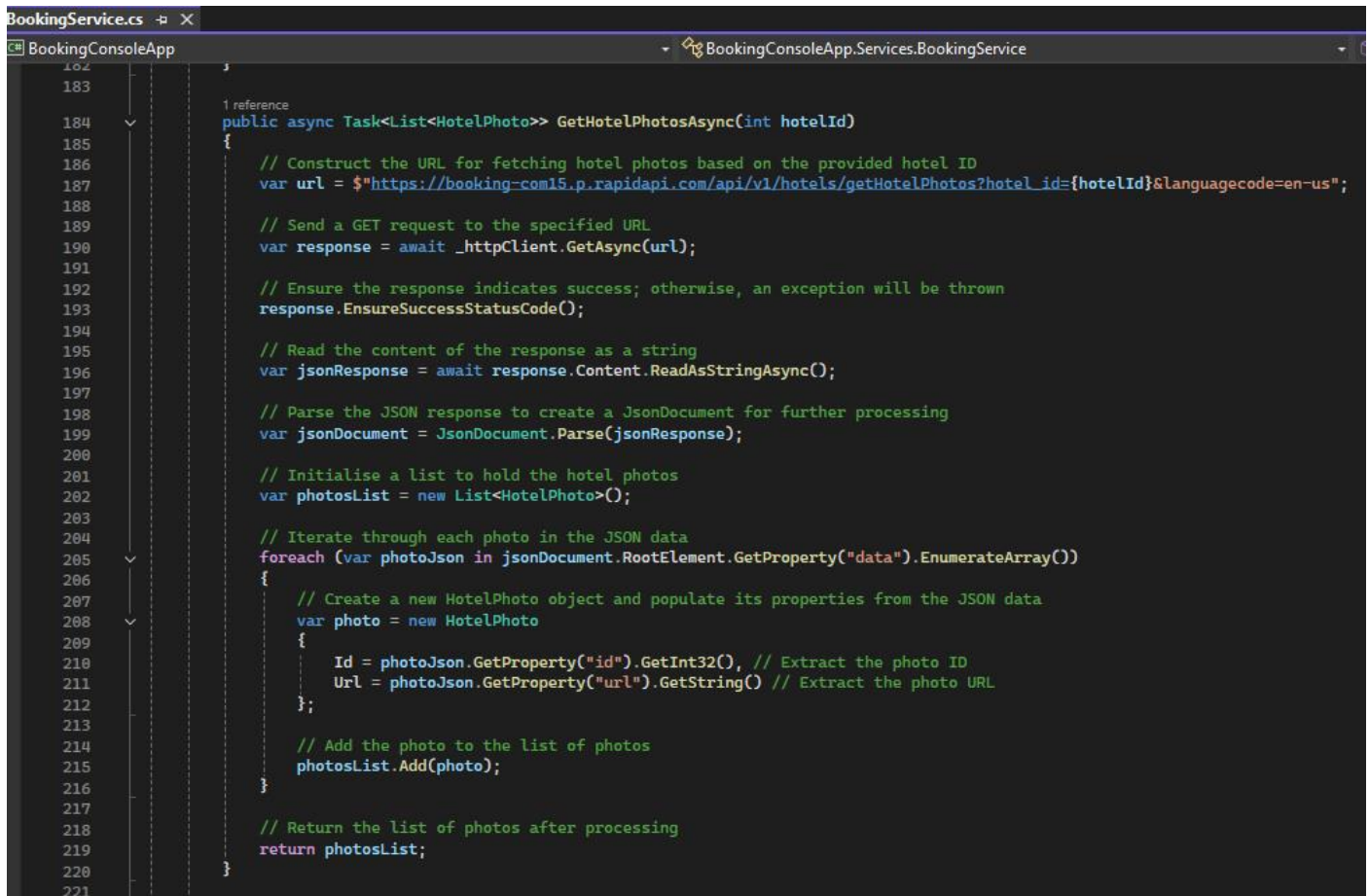
```

Figure 1 - Finding Hotel Reviews

As you can see above, the hotelId parameter is dynamically injected into the URL to receive precise reviews for the hotel specified by the user. Each review attribute is collected from the JSON format, allowing the app to accurately display the data.

2. Fetching Hotel Photos

When you need to download images for a certain hotel, the URL is created with the hotelId so that the API request is sent to the data set of this hotel to which it belongs. Then parse the JSON response to get and present a URL source for each one.



```
183
184
185 1 reference
186 public async Task<List<HotelPhoto>> GetHotelPhotosAsync(int hotelId)
187 {
188     // Construct the URL for fetching hotel photos based on the provided hotel ID
189     var url = $"https://booking-com15.p.rapidapi.com/api/v1/hotels/getHotelPhotos?hotel_id={hotelId}&languagecode=en-us";
190
191     // Send a GET request to the specified URL
192     var response = await _httpClient.GetAsync(url);
193
194     // Ensure the response indicates success; otherwise, an exception will be thrown
195     response.EnsureSuccessStatusCode();
196
197     // Read the content of the response as a string
198     var jsonResponse = await response.Content.ReadAsStringAsync();
199
200     // Parse the JSON response to create a JsonDocument for further processing
201     var jsonDocument = JsonDocument.Parse(jsonResponse);
202
203     // Initialise a list to hold the hotel photos
204     var photosList = new List<HotelPhoto>();
205
206     // Iterate through each photo in the JSON data
207     foreach (var photoJson in jsonDocument.RootElement.GetProperty("data").EnumerateArray())
208     {
209         // Create a new HotelPhoto object and populate its properties from the JSON data
210         var photo = new HotelPhoto
211         {
212             Id = photoJson.GetProperty("id").GetInt32(), // Extract the photo ID
213             Url = photoJson.GetProperty("url").GetString() // Extract the photo URL
214         };
215
216         // Add the photo to the list of photos
217         photosList.Add(photo);
218     }
219
220     // Return the list of photos after processing
221     return photosList;
222 }
```

Figure 2 - Fetching Hotel Photos

The `hotelId` argument is used here to verify that only relevant photographs are shown, with the degree of precision given by the user's input.

4.1.3 – Error Handling

Error management is commonly handled with `EnsureSuccessStatusCode` statuses, which assess whether the HTTP status code from each request was successful. If the response status is suggestive of a failure (i.e., outside the 200-299 range), it will quickly throw an exception and inform the application. The error handling will be saved to a `error_handling.txt` code file so whatever the error is showed on the CMD, it will then appear inside the `error_handling.txt` file.

Furthermore, JSON attributes remind one of continuations with conditional checks in place to prevent missing data or improper structure, which might generate runtime complications. Optional properties, such as the price, are acquired using the `TryGetProperty` method, as seen in the code sample below:

```

311 // Get price details with default value
312 PriceAmount = result.TryGetProperty("price", out var priceElement)
313 ? decimal.Parse(priceElement.GetProperty("amount").GetString())
314 : 0, // Default value if price is not found

```

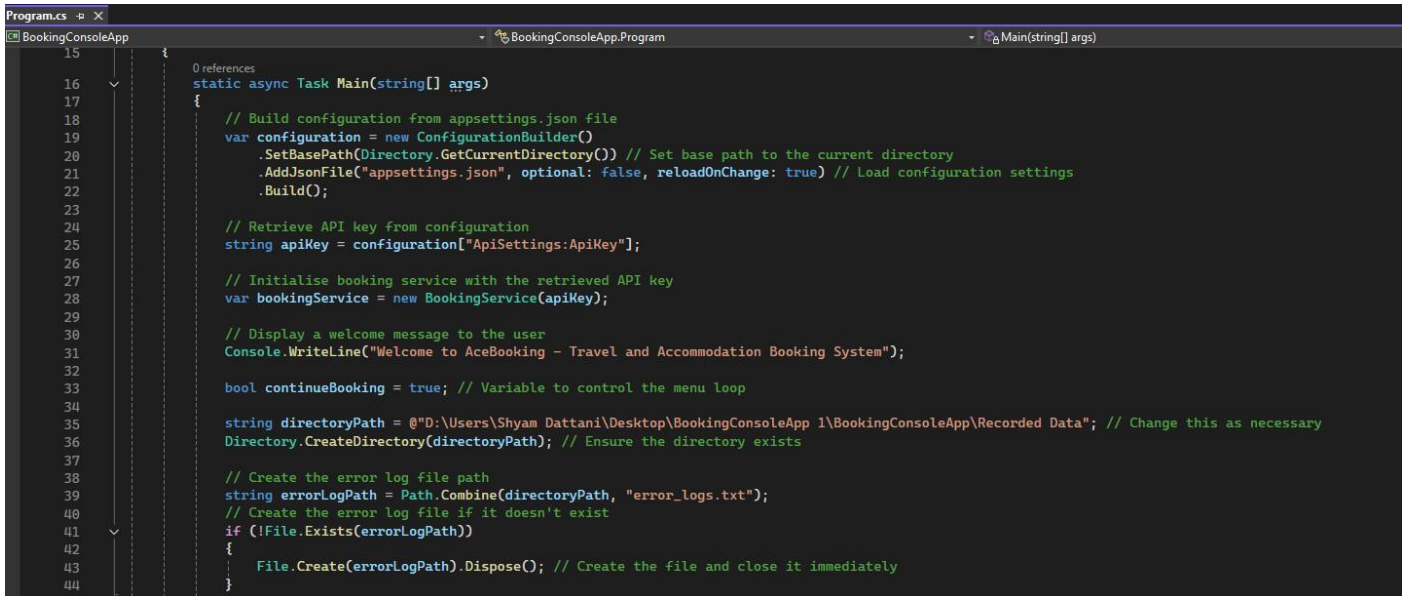


Figure 3 - Error Handling

That approach gives a default value if it does not exist, eliminating a possible null reference problem and enhancing program stability owing to varying API returns.

4.2 – Response Reception and Handling

In this part of the project, the techniques taken to handle replies obtained from the Booking.com API will be discussed. In this respect, we will offer an explanation of how API answers are handled in a step-by-step event, code snippets exhibiting how the application receives and processes such API replies, and data processing, as well as the proper data structure utilised.

4.2.1 – Description of Response Handling

Each handler's activity starts when the application makes an acquire request to the Booking.com API to gather hotel information. As with any standard API, the response body may include extra information such as hotel IDs, names, rates, and ratings, as well as other valuable data.

When the answer comes, the application determines whether the request was successful. It does this via the `EnsureSuccessStatusCode` function, which raises an exception when the response indicates an error. The computer then evaluates the content of the response and parses the JSON into structured objects.

4.2.2 – Code Snippet

The code sample below demonstrates how an application assesses the response supplied by the API after searching for hotel details:

```
public async Task<List<SearchHotel>> SearchHotelDetailsAsync(int destId, string searchType, string arrivalDate, string departureDate)
{
    // Construct the GET request to search for hotel details with the specified parameters
    var response = await _httpClient.GetAsync($"https://booking-coali.p.rapidapi.com/api/v1/hotels/searchHotels?dest_id={destId}&search_type={searchType}&arrival_date={arrivalDate}&departure_date={departureDate}&page_number=1&units=metric&temperature_unit=c&languagecode=ru-us&currency_code=USD");

    // Ensure the response indicates success; otherwise, throw an exception
    response.EnsureSuccessStatusCode();

    // Read the response content as a string
    var jsonResponse = await response.Content.ReadAsStringAsync();

    // Parse the JSON response to create a JsonDocument for processing
    var jsonDocument = JsonDocument.Parse(jsonResponse);

    // Initialize a list to hold hotel details
    var hotelDetailsList = new List<SearchHotel>();

    // Check if the 'data' property exists in the JSON response
    if (jsonDocument.RootElement.TryGetProperty("data", out JsonElement dataElement) && dataElement.TryGetProperty("hotels", out JsonElement hotelsArray))
    {
        // Iterate through each hotel in the 'hotels' array
        foreach (var hotelItem in hotelsArray.EnumerateArray())
        {
            // Create a new SearchHotel object and populate its properties from the JSON data
            var hotelDetails = new SearchHotel
            {
                HotelId = hotelItem.GetProperty("hotel_id").GetInt32(), // Extract hotel ID
                Name = CleanUpString(hotelItem.GetProperty("property").TryGetProperty("name", out JsonElement nameElement) ? nameElement.GetString() : string.Empty), // Extract and clean up hotel name
                AccessibilityLabel = CleanUpString(hotelItem.GetProperty("accessibility_label", out JsonElement accessibilityLabelElement) ? accessibilityLabelElement.GetString() : string.Empty), // Extract and clean up accessibility label
                Price = hotelItem.GetProperty("property").TryGetProperty("price_breakdown", out JsonElement priceBreakdownElement) && priceBreakdownElement.TryGetProperty("gross_price", out JsonElement grossPriceElement) ? grossPriceElement.GetProperty("value").GetDouble() : 0, // Extract gross price
                Currency = hotelItem.GetProperty("property").TryGetProperty("price_currency", out JsonElement priceCurrencyElement) && priceCurrencyElement.TryGetProperty("gross_price", out JsonElement grossPriceElement) ? grossPriceElement.GetProperty("currency").GetString() : string.Empty, // Extract currency code
                Rating = hotelItem.GetProperty("property").TryGetProperty("review_score", out JsonElement reviewScoreElement) ? reviewScoreElement.GetDouble() : 0, // Extract review score
                ImageUrl = hotelItem.GetProperty("property").TryGetProperty("photo_urls", out JsonElement photoUrlsElement) && photoUrlsElement.GetArrayLength() > 0 ? photoUrlsElement[0].GetString() : string.Empty, // Extract image URL if available
                Status = CleanUpString(hotelItem.GetProperty("status", out JsonElement statusElement) ? statusElement.GetString() : string.Empty), // Extract and clean up hotel status
            };

            // Add the hotel details to the list
            hotelDetailsList.Add(hotelDetails);
        }
    }

    // Return the list of hotel details after processing
    return hotelDetailsList;
}

// Utility method to clean up unwanted characters from strings
private string CleanUpString(string input)
{
    return input.Replace("?", "").Trim(); // Remove '?' and trim whitespace
}
```

Figure 4 - Search Hotels

SearchHotelDetailsAsync starts a GET call to the API, supplying the DestinationId, SearchType, ArrivalDate, and DepartureDate. Then it assesses if the answer was successful. Following that, it obtains the json content. Finally, it generates a JsonDocument object that can be parsed.

4.2.3 – Data Parsing

This means that such information would be handled and preserved in a SearchHotels model instance. This is achieved by scanning the JSON structure for appropriate qualities, after which the values identified must be assigned to the corresponding attributes in the SearchHotels object.

For example, to acquire the hotel's ID, one may use the following line:

hotelJson.GetProperty("hotel_id").GetInt32(); similarly, we employ the helper function CleanUpString to clean up the hotel's name and give it to the Name property. This procedure is repeated for attributes such as price, currency, rating, and picture URL.

Using structured models, such as the one used for SearchHotels in this code, enables the application to keep a clear representation of the data, making it simpler to access and update information about hotels obtained from the API.

4.3 – Display of Response

This session will explain how the Hotel Search API answer is communicated to the user using the console application's specified interface. It should be built such that the end user can swiftly enter his or her search parameters while also offering organised and clear information about the available hotels.

4.3.1 – User Interface Design

The console program allows users to input their destination ID, search type, and trip dates. Based on the user's input of the following data, the program receives and processes a list of hotels that satisfy the supplied criteria. Results are presented in a tidy and organised fashion, allowing visitors to easily view vital information for each hotel, including ID, name, accessibility label, price, currency, rating, and picture URL. This will assure that the interface is straightforward and easy to use. Here's a screenshot of how the user interface design is for the Search Hotel:

```
Enter your choice (1-9): 1
Enter a destination: Manchester
Hotels in Manchester:
Destination ID: -2602512
Name: Manchester
Region: Greater Manchester
Country: United Kingdom
Latitude: 53.4812, Longitude: -2.23615
Image URL: https://cf.bstatic.com/xdata/images/city/150x150/976977.jpg?k=8d13c94917fa00569d115c9123c7b5789ad41f7383b6fad32a1bda8e215e8936&o=
-----
Destination ID: 20079942
Name: Manchester
Region: New Hampshire
Country: United States
Latitude: 42.9956, Longitude: -71.4553
Image URL: https://cf.bstatic.com/xdata/images/city/150x150/980138.jpg?k=13118c0c36fd028243e14cd7fbd70f7e9b0364ef8dfc82e82791d3022aac8bc7&o=
-----
Destination ID: 1077
Name: Manchester City Centre
Region: Greater Manchester
Country: United Kingdom
Latitude: 53.47925, Longitude: -2.241755
Image URL: https://cf.bstatic.com/xdata/images/district/150x150/56351.jpg?k=81acc465a1c7b3a58b068901ae18dd157da72b472a9753515b9801dfa1c1f832&o=
-----
Destination ID: 47
Name: Manchester Airport
Region: Greater Manchester
Country: United Kingdom
Latitude: 53.362, Longitude: -2.27344
Image URL: https://cf.bstatic.com/static/img/plane-100.jpg
-----
Destination ID: 262863
Name: Country Inn & Suites by Radisson, Manchester Airport, NH
Region: New Hampshire
Country: United States
Latitude: 42.941082, Longitude: -71.468925
Image URL: https://cf.bstatic.com/xdata/images/hotel/150x150/521225131.jpg?k=0013d71225d985a171cc95ecaab18db50fa00f378e6041e1c0a962f31eedc88a&o=
-----
Press Enter to return to the main menu...
|
```

Figure 5 - User Interface Design

4.3.2 – Code Snippet

The following code snippets demonstrate how the application processes and displays the API response.

```
    }
    else if (choice == "2")
    {
        try
        {
            // Collecting the parameters from the user for the search
            Console.WriteLine("Enter destination ID: ");
            int destId = int.Parse(Console.ReadLine()); // Parse the destination ID from user input

            Console.WriteLine("Enter search type (e.g., City): ");
            string searchType = Console.ReadLine(); // Get the search type from user input

            Console.WriteLine("Enter arrival date (DD-MM-YYYY): ");
            string arrivalDateInput = Console.ReadLine(); // Get the arrival date input

            Console.WriteLine("Enter departure date (DD-MM-YYYY): ");
            string departureDateInput = Console.ReadLine(); // Get the departure date input

            // Convert dates from DD-MM-YYYY to YYYY-MM-DD
            DateTime arrivalDate = DateTime.ParseExact(arrivalDateInput, "dd-MM-yyyy", CultureInfo.InvariantCulture); // Parse the arrival date
            DateTime departureDate = DateTime.ParseExact(departureDateInput, "dd-MM-yyyy", CultureInfo.InvariantCulture); // Parse the departure date

            // Format dates as YYYY-MM-DD for API
            string formattedArrivalDate = arrivalDate.ToString("yyyy-MM-dd"); // Format arrival date for API
            string formattedDepartureDate = departureDate.ToString("yyyy-MM-dd"); // Format departure date for API

            // Call the updated method with the correctly formatted date strings
            List<SearchHotels> hotelDetails = await bookingService.SearchHotelDetailsAsync(destId, searchType, formattedArrivalDate, formattedDepartureDate);

            // Output the hotel details
            if (hotelDetails.Count > 0) // Check if any hotels were found
            {
                Console.WriteLine($"Hotels found:");
                foreach (var hotel in hotelDetails)
                {
                    // Display details for each hotel
                    Console.WriteLine($"Hotel ID: {hotel.HotelId}");
                    Console.WriteLine($"Name: {hotel.Name}");
                    Console.WriteLine($"Accessibility Label: {hotel.AccessibilityLabel}");
                    Console.WriteLine($"Price: {hotel.Price} {hotel.Currency}");
                    Console.WriteLine($"Rating: {hotel.Rating}");
                    Console.WriteLine($"Image URL: {hotel.ImageUrl}");
                    Console.WriteLine($"-----");
                }
            }
            else
            {
                // Notify user if no hotels were found
                Console.WriteLine($"No hotels found.");
            }
        }
        catch (FormatException ex)
        {
            Console.WriteLine("Input format is incorrect. Please ensure dates are in DD-MM-YYYY format.");
        }
        catch (ArgumentOutOfRangeException ex)
        {
            Console.WriteLine("The date entered is out of range. Please enter a valid date.");
        }
        catch (Exception ex)
        {
            // Handle any other errors that occur during hotel search
            Console.WriteLine($"An error occurred: {ex.Message}");
        }

        Console.WriteLine("Press Enter to return to the main menu...");
        Console.ReadLine(); // Wait for user input to return
    }
}
```

Figure 6 - User Interface Coding Design for Search Hotel

It depicts the procedure from retrieving hotel facts to displaying in a user-friendly way, with information easily gathered.

4.4 – Code Quality and Documentation

This section will assess code quality and documentation throughout development. How code is ordered and arranged, why comments are important, and how well-documented chunks of code appear.

4.4.1 – Structure of the Code

Code organisation is critical for retaining readability, scalability, and simplicity of debugging. We employed an organised approach in our application, placing distinct parts into different folders. For example, Program.cs works as our application's entry point, where the user interface exists to receive user inputs and perform the core logic. It gives a user interface for entering parameters for hotel information searches and showing the results.

The model SearchHotels is provided in the file HotelSearch.cs and comprises properties relevant to the hotel search result: HotelId, Name, Price, and Rating. This division of duties makes code maintenance much simpler since developers no longer have to crawl around in other regions and spend less time looking for and updating individual components.

Additionally, the BookingService.cs class will be in charge of connecting with Booking.com's API. It fulfils API requests and answers, separates the logic for obtaining data from the user interface, and allows new features or services to be effortlessly added into this application without modifying the present code.

4.4.2 – Comments and Documentation

Comments and documentation are vital for code quality. This promotes understandability, promoting cooperation among engineers. Well-documented code assures that a new person entering the project may instantly comprehend what the project accomplishes and decreases the learning curve associated with challenging implementations.

A fantastic example can be found in Program.cs, where comments are used to clarify in detail what the block of code is designed for and to add context to the collection of user inputs:

```
// Collecting the parameters from the user for the search
Console.Write("Enter destination ID: ");
int destId = int.Parse(Console.ReadLine()); // Parse the destination ID from user input
```

Figure 7 - Program.cs – Comments

The commented code below demonstrates what is being done, making it more understandable for other developers who may arrive later, since they can readily determine from the remark what the code does.

In the file HotelSearch.cs, the properties are well-defined in the SearchHotels class; it helps to understand how data should search for a managed hotel:

```
// Property to store the unique identifier for the hotel
2 references
public int HotelId { get; set; }

// Property to store the name of the hotel
2 references
public string Name { get; set; }
```

Figure 8 - HotelSearch Comments

All of the properties have a statement indicating what the value does, which is useful for those unfamiliar with the code base.

The product is now manageable and will be simpler to extend in the future thanks to standard code comments and acceptable structure.

5 – Testing and Validation

This section outlines the testing and validation techniques used to validate the reliability and functioning of our application. Testing is a vital phase in the software development lifecycle because it enables us to detect and resolve problems prior to release. We go over the methodology utilised, discuss multiple test cases, provide results, and highlight problem solutions and iterations made throughout the testing process.

5.1 – Testing Approach

We applied a mix of unit testing and user acceptability testing to verify the application's resiliency.

Unit testing was undertaken to examine the application, with an emphasis on strategies for handling API requests and processing answers. Automated unit tests have been built, in which each function is tested against a given set of input and projected output, enabling us to discover errors early while also insuring that updates to one part of our code do not mistakenly effect others.

User Acceptance Testing includes real users testing the program to verify that it satisfied expectations and requirements. The substantial input supplied helped us to better the user interface and overall experience in terms of usability and functionality.

5.2 – Test Cases

Some test cases have been developed to analyse the application's implementation. Some of the primary test cases and outcomes might include:

Test Case ID	Description	Input Parameters	Expected Outcome
TC001	Validate successful hotel search	Enter a destination: Manchester	List of hotels is returned with valid details
TC002	Validate handling of invalid destination ID	Enter destination ID: 48275 Enter search type (e.g., City): City Enter arrival date (DD-MM-YYYY): 01-12-2024 Enter departure date (DD-MM-YYYY): 06-12-2024	Error message indicating invalid destination ID
TC003	Validate date format parsing	Enter arrival date (DD-MM-YYYY): 01-12-2024 Enter departure date (DD-MM-YYYY): 06-12-2024	Dates are correctly formatted for API request
TC004	Validate response for no hotels found	Enter destination ID: 63 Enter search type (e.g., City): City Enter arrival date (DD-MM-YYYY): 01-12-2024	Message indicating no hotels found

		Enter departure date (DD-MM-YYYY): 06-12-2024	
TC005	Validate proper error handling on API failure	Simulate API failure	Error message indicating API is unreachable

Table 1 - Test Cases

5.3 – Results

These tests frequently suggested that the application performed as it should. The majority of the unit tests passed without problem, suggesting that most of the components were operating appropriately. However, a number of faults were found during UAT, such as intuitiveness in the use of the user interface and clarity in error messages.

For example, users struggled to grasp the key date format. As a result, there were many entry mistakes in the date field. This was to alert to us that we needed to advise the user and do extra validation of the input.

5.4 – Bugs Fixes and Iterations

Several flaws were found throughout the testing method. Iterations to improve the application were subsequently required. One of the key difficulties noticed was in the date parsing process. The improper formats chosen prompted the program to crash. To address this issue, we had to perform an additional check to assure that the user supplied the date in the appropriate format before completing the search.

Other issues emerged while dealing with API failures. When the API was unavailable, users received generic error messages that left them bewildered. We changed an error handling system to offer more useful notifications that would help users in rectifying the problem or retrying their request.

All of these enhancements and bug fixes enhanced the program tremendously, making it more robust and user-friendly.

6 – Reflections and Lesson Learned

This part will reflect on the full process of application development, concentrating on code quality and maintainability, highlighting significant learning points, and investigating different options if the project were reopened. It seeks to produce insights that will be beneficial in future activities.

6.1 – Quality and Maintenance

The code generated for this application was generally qualitative, well-structured in terms of class design, and had a clear separation of concerns. The object-oriented programming approaches utilised made it accessible, allowing one to easily learn and alter the code as necessary. However, issues occurred in ensuring uniformity among varied components, notably in naming conventions and documentation processes.

The significant one was the problems in establishing the Booking.com API, which should be done with strong response validation and error handling, since robust error handling and response validation are depending on its implementation. This needs careful planning and testing without compromising high-quality code while avoiding runtime blunders. Regular code reviews and adherence to coding standards helped lessen these challenges, resulting in this viable codebase.

6.2 – Key Learning Points

Several lessons were learnt along the development process. First and foremost, effective testing is important; it plays a significant role in finding and correcting issues early in the development life cycle. It showed the necessity of unit and user acceptance testing in building a trustworthy product.

Further knowledge of API integration, particularly the challenge of handling asynchronous requests and storing return data, was expanded. This expertise will undoubtedly be beneficial in future initiatives that demand the development of equivalent skills.

It also helped me build my abilities in user interface design by concentrated on providing user-friendly inputs and outputs that enhance the overall user experience. Actual users' comments underlined the necessity of simplicity of use and clear instructions, which I will continue to prioritise in future development.

6.3 – What could we have done differently

Looking back, there are undoubtedly a number of additional strategies I may have used, and may apply in the future, when discussing this issue. For example, we may have done more with the user interface, such as looking into web development frameworks to generate a more visually pleasing application. The console software functioned, but the online interface was more configurable and user-friendly.

Also, we would do substantially more detailed early user research to verify that the program fulfils all user expectations from the outset. The slightest bit of specific user input, if asked far earlier in the development cycle, may have saved significant adjustments in the latter phases.

Finally, architectural patterns such as MVC may have been utilised to minimise the amount of code required to construct the product and enhance scalability, making it much simpler to upgrade in the future with new features.

7 – Conclusion

To summarise, the creation of the Booking Console Application was a great feat since it entailed the connection of the Booking.com API to provide the user with basic hotel search capacity. The application produced herein fulfils with the requirement by allowing the user to enter destination, arrival, and departure dates and then get hotel data, pricing details, ratings, and accessibility information. It will increase the user experience by making the hotel search process easier while also presenting users with full information to make wise decisions.

Throughout the development process, thorough testing and validation approaches were applied to validate the dependability and performance of this application. With a heavy focus on code quality and maintainability, a solid foundation is established for future improvements.

The following are some probable future upgrades to the project:. It may also make ideas for future expansion, such as upgrading the application to encompass airline reservations and transportation possibilities in order to provide consumers with a more holistic vacation planning experience. The change to a Web-based interface should enhance access and user engagement by being more visually beautiful and engaging.

Continuous user input will be vital in paving the way for future modifications, allowing the application to adapt alongside its users.

8 – References

- Basit, A. (2024). A Conceptual Framework of Organizational Antecedents to Knowledge Sharing- With Empirical and Managerial Implications. *Journal of Accounting and Finance in Emerging Economies*, [online] 10(2). doi:<https://doi.org/10.26710/jafee.v10i2.2970>.
- Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10), pp.70–77. doi:<https://doi.org/10.1109/2.796139>.
- Beck, K. (2001). *Praise for Extreme Programming Explained, Second Edition*. [online] Available at: <https://ptgmedia.pearsoncmg.com/images/9780321278654/samplepages/9780321278654.pdf>.
- Dumas, M. (2013). *Fundamentals of Business Process Management*. [online] Available at: https://repository.dinus.ac.id/docs/ajar/Fundamentals_of_Business_Process_Management_1.pdf.
- Gemma, E. (1994). *Creational Patterns*. [online] Available at: <https://www.javier8a.com/itc/bd1/articulo.pdf>.
- Hahn, C. (2022). *Online Booking System - What It Is and How It Works*. [online] EasyPractice. Available at: <https://easypractice.net/what-is-online-booking/>.
- Kaufmann, M. (2024). *Usability Engineering*. [online] Google Books. Available at: https://books.google.com/books/about/Usability_Engineering.html?id=v4PkIjEJkmsC [Accessed 1 Nov. 2024].
- Kumar, V., Kumar, V., Singh, S., Singh, N. and Mr. Sreenu Banoth (2023). The Impact of User Experience Design on Customer Satisfaction in E-commerce Websites. *International Journal For Science Technology And Engineering*, 11(5), pp.4571–4575. doi:<https://doi.org/10.22214/ijraset.2023.52580>.
- Laanti, M., Similä, J. and Abrahamsson, P. (2013). Definitions of Agile Software Development and Agility. *Communications in Computer and Information Science*, pp.247–258. doi:https://doi.org/10.1007/978-3-642-39179-8_22.
- Larman, C. and Basili, V.R. (2003). Iterative and incremental developments. a brief history. *Computer*, [online] 36(6), pp.47–56. doi:<https://doi.org/10.1109/mc.2003.1204375>.
- Rajgopal, S. (2024). *Does Online Customer Experience Affect the Performance of E-commerce Firms?* [online] ResearchGate. Available at: https://www.researchgate.net/publication/2430230_Does_Online_Customer_Experience_Affect_the_Performance_of_E-commerce_Firms [Accessed 1 Nov. 2024].
- Saeed, S. (2023). A Customer-Centric View of E-Commerce Security and Privacy. *Applied Sciences*, [online] 13(2), p.1020. Available at: <https://www.mdpi.com/2076-3417/13/2/1020>.
- Schwaber, K. and Sutherland, J. (2019). *The Scrum Guide*. [online] Scrum.org. Available at: <https://www.scrum.org/resources/scrum-guide>.

Sommerville, I. (2011). *SOFTWARE ENGINEERING Ninth Edition*. [online] Available at: <https://engineering.futureuniversity.com/BOOKS%20FOR%20IT/Software-Engineering-9th-Edition-by-Ian-Sommerville.pdf>.

W3C (2023). *Web Content Accessibility Guidelines (WCAG) 2.1*. [online] W3.org. Available at: <https://www.w3.org/TR/WCAG21/>.

Xu, J. and Wei, W. (2023). A theoretical review on the role of knowledge sharing and intellectual capital in employees' innovative behaviors at work. *Heliyon*, 9(10), pp.e20256–e20256.
doi:<https://doi.org/10.1016/j.heliyon.2023.e20256>.