Contents

[Chapter 1. Introduction 3](#_Toc116242532)

[1.1. Scope and context of the theme approach 3](#_Toc116242533)

[1.2. Proposed subject 3](#_Toc116242534)

[Chapter 2. Theoretical Background 4](#_Toc116242535)

[2.1. Alternative solutions 4](#_Toc116242536)

[2.2. Web Services 5](#_Toc116242537)

[2.2.1. RESTful services 5](#_Toc116242538)

[2.2.2. Web API Services 6](#_Toc116242539)

[2.3. Technologies 7](#_Toc116242540)

[2.3.1. MAS 7](#_Toc116242541)

[2.3.2. .NET 8](#_Toc116242542)

[2.3.3. ASP.NET Core 8](#_Toc116242543)

[2.3.4. Fluent Migrations Framework 9](#_Toc116242544)

[2.3.5. OpenTripMap API 10](#_Toc116242545)

[2.3.6. POSTMAN 10](#_Toc116242546)

[2.3.7. AJAX 11](#_Toc116242547)

[2.3.8. SQL and PostgreSQL 11](#_Toc116242548)

[Chapter 3. Application design 12](#_Toc116242549)

[3.1. Application architecture 12](#_Toc116242550)

[3.2. Application levels 14](#_Toc116242551)

[3.2.1. Presentation tier 14](#_Toc116242552)

[3.2.2. Data tier 15](#_Toc116242553)

[3.3 Application block diagram 16](#_Toc116242554)

[3.3.1. Web application block diagram 17](#_Toc116242555)

[3.3.2. Web API block diagram 18](#_Toc116242556)

[3.3.3. Database 23](#_Toc116242557)

[Chapter 4. The application development 24](#_Toc116242558)

[4.1. Database 24](#_Toc116242559)

[4.2. Data tier 27](#_Toc116242560)

[4.3. Presentation tier 30](#_Toc116242561)

[Chapter 5. Application testing 34](#_Toc116242562)

[Chapter 6. Conclusions and future directions of development 37](#_Toc116242563)

***Search for a complete vacation package using multi-agent negotiation***

Mihailov Emilian

**Summary**

This paper describes a recommender system that has a multi-agent system as its processing core. When the client, using the graphical interface, requests a complete holiday package, a call is sent to the server, which is interpreted as an API. The user's preferences are sent to the multi-agent system, which aims to provide the best possible recommendation. Agents will be able to communicate with each other as needed if they do not have enough knowledge to solve the proposed task.

A particularity of this system is that there are two types of agents: the *coordinator* and the *tourism agents*.

*The coordinator* is the one who informs the travel agents that a request has just arrived and is asked to provide recommendations for four services: finding a departure flight, a return flight, a place to stay during the holiday and some touristic attractions. This agent will also monitor the status of the tasks the agents have to complete and keep track of which services have been completed in order to be provided to the customer. When all services are collected into a complete package, the coordinator will notify the API services that the recommendation is ready and can be sent to the client.

*Tourism agents* are the ones who do the actual work. This means that they are responsible for providing a solution to the user in accordance with their preferences. Agents have their own knowledge, which can actually be interpreted as a personal database in which agents have stored data about tourist services (flights, properties, attractions). However, this can be incomplete, which is why communication between agents is vital.

Teamwork is a defining trait of a multi-agent system. Agents choose their tasks individually, without talking to others, but when they need help, they can ask others. In the proposed architecture agents become experts in certain types of holiday services, and this makes the recommendation process more efficient. The degree of expert is determined by the ratings the user provides. The more positive the rating, the better the agent is considered an expert, deducing that the local database the agent contains best fits the user's wishes.

The whole purpose of this process is to make it as easy as possible for the user to find what best matches them. It is no longer necessary to search on different websites for generic holiday packages or to plan a vacation by searching on flight websites then renting a space. This project aims to provide all the necessary vacation services to suit the exact needs of the user.

In addition to development ideas, this project also offers the possibility of including algorithms to analyze the user's profile and offer holiday packages based not only on the preferences displayed in the graphical interface, but also from the profile analysis.

The project architecture will apply two important actors: the client and the server. The first one will represent the graphical interface from where the preference calls will be made, and the server will be another resource that will be in charge of calculating all the data and providing the response to the user. The multi-agent system will obviously be stored on the server side.

ActressMAS [14], which is inspired by the actor model, was used as the technology for the multi-agent system. It is quite easy for a programmer to use it, plus all the necessary documentation can be found from official resources.

.NET was used as programming platform. It offered the possibility to create the web application using ASP.NET Core MVC, and for the server ASP.NET Core Web API.

It can also be mentioned that a separate project has been created for the database, which will generate it using migrations. From the same project the populating of the database with information about holiday services (in particular flights, properties, and tourist attractions) will also be performed. As database environment was used PostgreSQL.

# Chapter 1. Introduction

## 1.1. Scope and context of the theme approach

After most recent global disease outbreak Covid-19, the economic crisis that is felt in Europe, but also on other continents caused massive travel demands. The prices on flights or stays are rising as well, particularly in the high season when the demand is increased. The travel agencies are overloaded with requests, and it seems that a real travel boom is looming, but the industry may not be ready.

The UNWTO's latest World Tourism Barometer shows an increase of 182% for international tourism in the first three months of 2022 compared to the previous year [1]. As long as physical travel agencies are overloaded, a future perspective would be the creation of virtual agents that are able to perform the same functions, or even more.

When it comes to planning a vacation, people start browsing dozens of travel sites. Each service is searched separately, whether talking about flight tickets, hotels, or attractions. This search often takes quite a long time, but their efficiency is also low if people are not experienced travelers. Obviously, there are travel agencies that offer full services, but it must be taken into consideration that the agencies' services are included in the final price and are often not cheap.

The aim of the project is to help customers find the most suitable trips according to their preferences by accessing a website. An individuality of the proposed system is the fact that the user can indicate a necessary minimum of information to receive what he needs. The more explicit the preferences are, the closer the result will be to his requirements.

## 1.2. Proposed subject

The technology of agent-based systems has increased popularity in recent years due to its characteristic of new paradigm for creating a concept, designing and implementation of a software system. That’s why a best fit for solving a problem was developing of a multiagent system (MAS) that will compose a recommender system[[1]](#footnote-2). This type of system not only offers the possibility to find a solution to a problem, but also to allow the extensibility of services. In our case, is made reference to the possibility of integrating a system for analyzing the client's profile and providing recommendations based on this factor.

The project consists in realization of an application that will provide to the user a complete vacation package for each request. In this context, a full vacation package includes departure and return flight, a property to stay and some attractions. The required information from the customer, to send the request, are minimal: departure and destination city, the date of check-in and holidays period. Of course, here are more optional customized filters and preferences as flight companies, part of the day most preferred for flight (e.g., morning or night), type of place to stay, some attraction preferences and others. The more preferences are set, the better result customer will get.

The main component that will process the user request and bring a response back to the customer is a multi-agent system. The principle is simple: “*Divide et impera*”. Each agent will be responsible for a single vacation service (flight, property, or attractions). When all the agents are done with their task, a full vacation package is constructed based on their results. They act like real life tourism agents. Each have his own responsibilities, knowledges, rating, and the ability to communicate with other agents to complete his task. Last one is essential in a multi-agent system. Agents should be able to ask for some help from the others, to complete their recommendation, or in turn to offer some.

# Chapter 2. Theoretical Background

## 2.1. Alternative solutions

Multi-agent systems are new in software industry, but still, several projects have already been created related to the topic of vacation packages using a multi-agent system. One of them is the MAPWEB [2].

The system of MAPWEB consists of 4 agents: the agent that is responsible for communication between system and the user, another one plans the travel, only one that is searching for information (in this case only on Internet) and the coach agent that will manage the entire system of agents. As in a usual MAS system, in MAPWEB the agents register each recommendation as a case and on next request he will use it to improve his results. The biggest disadvantage of this system is the existence of the “coach agent”, because the communication between agents is strongly managed by him. He tells agents to who they should ask for help. That is a big waste of resources, and a better idea would be to make agents independent when they communicate with each other. A good practice of this system is that agents split the tasks in common agreement based on their service expert rate.

Compared to MAPWEB, our system has two types of agents: a *coordinator* agent that maintain the relation between customer and other agents and the *tourism agents* that are responsible for bringing solutions as the result of customer request.

Another multi-agent system related to recommendation system is the CASIS [3]. This application study the user preferences with each request and improves its suggestions over time. In this case, the multi-agent system is seen as swarm of agents that negotiate to release a result. As in MAPWEB, here is also used the recommendation case-based, to bring to the user the best travel results.

The recommendation process works as follows: the user elicits her preferences; the bees visit all cases in the case base and when they find the best case (according to the user’s preferences) they dance to that case, recruiting other bees to that case; and the case with the greatest number of bees dancing for it is the one recommended to the user.[4]

The advantage of CASIS is that the system has always something to offer to the customer, even the rate of match-cases is low. That means the user will get a recommendation regardless of his preferences or the knowledge that the agents possess. The main problem of this system is the lack of communication between agents if they need to check the cases that the other agents hold.

The information exchange in the MAS should be predominant as it is presented in the project of current document. All services datasets as flights, hotels or attractions are split between agents. This will increase the processing speed of a request. Also, that means each agent will be an expert on data he owns. But also here is a probability that any agent will miss some information for completion of his task. In that case the communication is vital.

The MATRES [5] is an assumption-bases multiagent system that bring to the user personalized travel services. A particularity of this application is the trust degree of the agent in other agents and in oneself. The first one will help the agent to choose who to send the request for help in case of inability to solve the proposed task. That will exclude unnecessary communication between agents. The second one, self-trust degree will help agent to choose which task type to perform.

The advantage of this system is that agents have three sources to find a result for the customer: his own knowledge database, community search (communication with other agents) and Internet search. Another advantage is the one that was not found in previous mentioned systems. It’s the agents’ assumptions that allow agent to reason with incomplete information by making guesses.

The disadvantage on the system is that it doesn’t have a stable recommendation quality. In the testing stage, were used three kinds of recommendation sources: cooperations among agents, decision assumption based on most popular option in the community of users, similar cases. After some simulations, was clear that a compromise should be made when choosing the recommendation strategy. For example, the *Similar cases* method produces better results for the hotel tasks only, compared with *Most popular option in the community*. Using *Similar cases* is a good strategy to make assumptions for new users when necessary [5].

Another disadvantage is the trust degree on other agents that was too severe. Even if other agent was expert in some sort of service, the current agent refused to communicate with the first one because of low trust degree. Or another scenario is that an agent decreases the trust degree in the other one and in the future requests he will not consider the communication with him.

Our project had also implemented the trust degree between agents, but the main differences is the usage. In MATRES, an agent asks for help only from one most trusted agent, but in our system, the agent will send a request to all agents one at time in descending order of trust.

## 2.2. Web Services

A web service is a software system whose services are provided on the Internet by special programs and are identified by a URL string. This software system is located by other software systems. Web services provide interaction of software systems regardless of platform and are based on open standards and protocols.

The universality of the technologies is the basis for understanding web services. They run on standard technologies, independent of application providers and other network resources. They can be used in any operating system, application servers, programming languages, etc.

The main advantages are:

* Creating the necessary conditions for the interaction of software components regardless of the platform.
* Web services are based on open standard protocols. By implementing XML, web services can be easily formed and configured.
* The use of HTTP guarantees the interaction of systems through inter-network access.

A great disadvantage is the security level. All modern web services must implement encryption and require user authentication. Whether HTTPS is enough here or more secure protocols like XML Encryption, SAML, etc. are needed is decided during development.

### 2.2.1. RESTful services

REST is the abbreviation of Representational State Transfer. It is a structural design approach for crafting loosely attached applications using HTTP, often implemented in the growth of web services. REST web services do not impose any rules concerning how it needs to be applied in practice at a low level; it only holds the high-level design guiding principles and leaves it to the developer to think about the implementation [6].

In theory, REST is not tied to the network, but is almost always implemented as such and was inspired by HTTP. As a result, REST can be used wherever HTTP is possible.

The standard database verbs in a RESTful architecture are CRUD (C – create, R – read, U – update, D - delete). Those are most common actions used to manipulate the database records. To not make any confusion, should be enounced that explicitly CRUD operations in the REST context have an equivalent (e.g., create = post/put, read = get, update = put/post, delete = delete). HTTP verbs tell the server what to do with the data specified by the URL. The request can still contain additional information in its body that may be needed to perform the operation - for example, the data user want to save with the resource.

Each REST API request reports results by numerical codes - HTTP statuses.

For example, editing a record on the server may work successfully (code 200), may be blocked for security reasons (code 401 or 403), or even crash in the process due to a server error (code 500).

The REST API also allows customer to exchange more than just text information. User can use this tool to transfer files and data in special formats: XML or JSON.

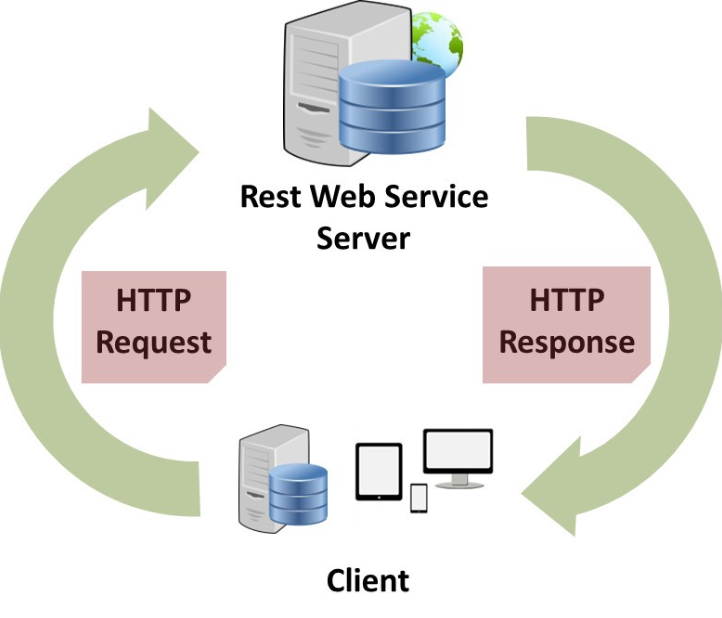


Figure 2.1 RESTful Client-Server Communication Cycle [image source [7]]

As it’s shown in the Figure 2.1, it is necessary to separate the needs of the client interface from the needs of the server storing the data. This constraint increases the portability of client code to other platforms and simplifying the server part improves the scalability of the system. The very delimitation into "client" and "server" allows them to develop independently of each other.

Advantages of RESTful web services [8]:

* RESTful web services are platform independent.
* It can be written in any programming language and can be executed on any platform.
* It provides different data format like JSON, text, HTML, and XML.
* It is fast in comparison to SOAP because there is no strict specification like SOAP.
* These are reusable.
* They are language neutral.

### 2.2.2. Web API Services

An API or Application Programming Interface can be defined as a set of procedures that programmers use to retrieve concrete data or features for an application or service. A Web API on its turn, is an API over the web that communicates with other environments through HTTP calls (see Figure 2.2.).

An API includes functions, classes, methods, and structures that help one application to interact with another. The API contains some "bridges" that allow program A to access data from program B or some of its features. In this way, programmers can extend the functionality of their product and link it to the developments of others.

API can be considered as an interface, because in simple terms, an interface is a layer between application A and application B. It has processes that allow the two programs to exchange information and perform functions related to both sides, hiding the "inner workings" of the programs.

This approach allows for interaction between multiple utilities without having to think about how they are structured, what program logic drives them, or how the data being transmitted is handled. Interfaces simplify the work both for ordinary users and for programmers. The former does not have to think about what is behind the usual functions in their gadgets, and developers do not need to study the code of other programmers to connect someone else's product to their own.

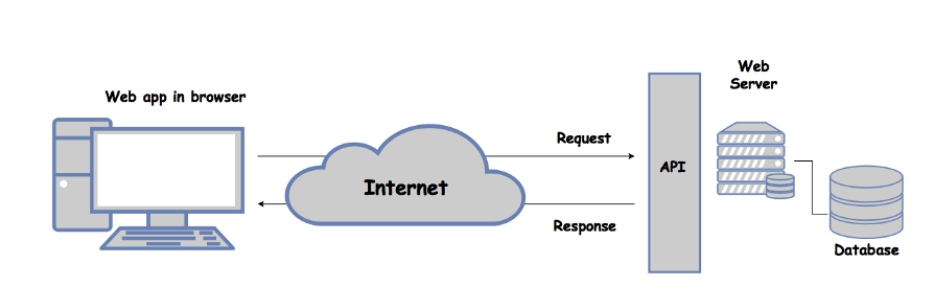


Figure 2.2 API basic architecture diagram [image source [10]]

This is called encapsulation. Hiding part of the functions for the sake of simplifying the work as a whole and minimizing the areas of the software where one of the developers could make a mistake.

Many companies offer APIs as a finished product. For example, American Express[[2]](#footnote-3) offers the Account and Transaction API. It provides customer-authorized, account-specific data for all American Express proprietary card products including Personal, Small Business and Corporate Cards for parties certified under the Payment Service Directive 2 (PSD2) legislation. [9]

## 2.3. Technologies

### 2.3.1. MAS

The core of the current application was built using a multi-agent system. In simple terms, MAS is a system that is formed by multiple agents that are able to communicate and interact with each other. Their main reason is to solve a concrete problem, as it is in our case – finding a full vacation package. Each agent has his own knowledge, capabilities, and intelligence that they demonstrate by solving tasks and interacting with others.

Multi-agent systems have various real-world applications. For example, they can be used to set up robots to perform certain operations, investigate different environments with a high degree of danger, as well as exploring disaster scenarios, investigating transport systems.

Multi-agent systems in robotics, are going through specific and considerable issues. As example will be approached the RoboCup project [11].

The main challenge of this work is to develop a team of robots having an appearance or character resembling that of a real football team. The reason is not only to play using some strategies, but to beat a real football team, more explicitly, the world champion of 2050. Football being a complex game with multiple leagues, and teams (each with own strategy and characters) will give the opportunity to the RoboCup system to learn and study different approaches on solving the problem - winning the football world cup.

The simulation league looks like a standard computer game, but the key difference is that each player is an individual robot, driven by their own program. Each agent must decide on the next move. Because simulation frees researchers from the physical limitations’ inherent limitations, these screen players can perform at a much more advanced level [12].

Another agent-based system in ADEPT. It is a project used in telecommunications domain for estimating an offer for installing a network by the network provider.

In the multi-agent system, each department is represented by an agent, and all the interactions between them take the form of negotiations. All negotiations are centered on a multi-attribute object, where attributes are, for instance, price, quality, duration of a service [13].

#### 2.3.1.1. ActressMAS Framework

In the presented project, as multiagent framework was used the ActressMAS. It is inspired by the actor model. The aim is to make it as easy to use as possible and was predestined for students learning in the domain of multi-agent systems [14].

As alternative framework can be counted MASON [15] or JADE [16], but they both are made in Java. The one that was created for .NET is Orleans [17] that also introduce the abstraction of virtual actors.

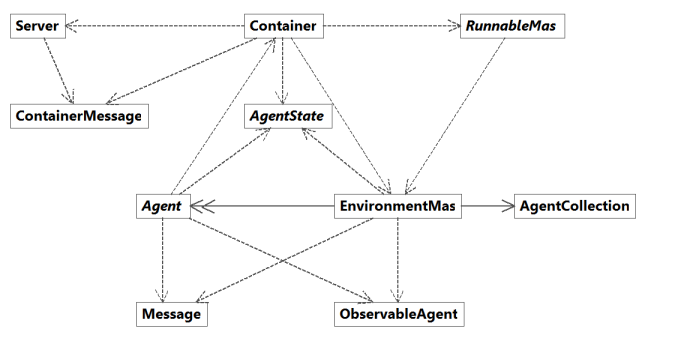


Figure 2.3. The general architecture of the ActressMAS framework. [image source [14]]

The reason of using the ActressMAS framework is the simplicity of use, light architecture (see Figure 2.3.) and compatibility with .NET. The communication between agents can be made by messaging an also through shared memory. The parallel execution allows agents to work on their task asynchronously. This brings high efficiency to the application and increase the execution speed. An interesting fact is that agents have a reactive behavior. That means they are triggered only when they got a message from others. Of course, agents can be customized to act even there are no incoming messages by tracking their state. ActressMAS proposes a lot of customized features that help to build an intelligent MAS environment. Some of these features were used as well in the presented project.

### 2.3.2. .NET

.NET is a framework from Microsoft that allows the use of the same namespaces, libraries, and APIs for different programming languages. Most commonly, these are four languages in the .NET family:

-C#.

-Visual Basic.

-Visual C++.

-F#.

If .NET didn’t exist, users would have to install a runtime environment for programs in each language. That is, to run a Visual Basic application, you must download the runtime for Visual Basic. If the program is written in C#, you would have to download the runtime for it as well.

This is also important for programmers, because it allows them to develop one environment that is used for four languages at once. Otherwise, regular developers would have to wait until a new version of the libraries for their language is released. Less popular languages, like F#, would get updated much later than C#.

The best part of using .NET are the wide documentation libraries created by Microsoft. It’s straight-forward for a beginner programmer to find required materials to learn basics, but also the professionals can be delighted by the newest features of .NET updates.

In the current project was used .NET 6 that was released in Nov 2021. Compared to previous version .NET 5, it brings more optimizations and refactoring.

### 2.3.3. ASP.NET Core

ASP.NET Core is a cross-platform framework for building a modern cloud of Internet-connected applications such as web, Internet of Things, and mobile server applications. ASP.NET Core applications can run on .NET Core or .NET Framework full platform.

ASP.NET Core has several architectural changes that result in a more compact and modular structure. It is no longer based on the System.Web.dll file. It is based on a set of detailed and well-structured NuGet[[3]](#footnote-4) packages. This allows you to optimize your application with the NuGet packages you need. The benefits of a smaller application footprint include tighter security, reduced maintenance, improved performance, and lower costs in a "pay for what you use" model.

This are the advantages of the ASP.NET Core:

* Common release history for Web UI and Web APIs
* Integration of modern client frameworks and development schemes
* Cloud-ready, environment-based configuration
* Native support for dependency deployment
* New lightweight and modular HTTP request
* Ability to host in IIS, or in your own application
* Delivered as complete NuGet packages
* New toolset that simplifies the development of modern web applications
* Build and run cross-platform ASP.NET applications on Windows, Linux, and Mac
* Publicly available and socially oriented framework

### 2.3.4. Fluent Migrations Framework

One of the best ways to achieve flexibility in the software development lifecycle is through automated build and deployment. More often than not, the bottleneck is database deployment. For many years, code deployment and database deployment were separate. The database was always deployed by database administrators, whereas code was usually deployed using automated and/or semi-automated systems.

Now the reliance on database administrators to deploy the deployment process is slower and there is more emphasis on coordination with multiple teams during deployment.

Fluent Migrator was developed for .NET users. It is a database migrator that uses fluent interface to work with database. It’s easy to use this framework because the migration classes are some C# classes that inherit the “Migration” base class and just implement the “Up” method that will upgrade the database or “Down” for downgrade. The framework also maintains a “Version” read-only table (See Figure 2.4.) in database for tracking all migrations that were applied to database.

O imagine care conține text

Descriere generată automat

Figure 2.4. Example of VersionInfo table created by Fluent Migrator

The biggest question when choosing Fluent Migrator would be “Why not choosing the most popular migration framework for .NET – Entity Framework?”. And the response will be simple. Entity Framework is an Object-Relational Mapping. That means you should have first to create each database entity as a class and then Entity Framework will abstract the code and then virtually will map to tables in the database. Fluent Migrator is commonly used with non-ORM frameworks. Besides the schema migrations, the developer can also migrate data using NuGet packages (e.g., CsvHelper for loading rows from a .csv file) and any data access framework as Entity Framework, Dapper or RepoDB.

### 2.3.5. OpenTripMap API

An API or application programming interface can be considered as a set of features and functionalities that allows an application to access some concrete data and interact with different operation system or software components.

OpenTripMap API allows users to retrieve data from OpenTripMap [18] website’s database through HTTP calls.

This are the features this API offers [19]:

* Place list - returns a list of places based on a location, type, rate, and other parameters
* Place Details - returns detailed information about a specific place, such as address, description, URL, image, and others
* Place Autosuggest - provides a places query by given (partial) search term and location context
* Geographic coordinates request - returns coordinates for the given placename (city, village, etc.)

The API is free for non-commercial use and its limitation are 5000 request/day and 10 requests/second.

### 2.3.6. POSTMAN

Postman is a service used for manual and automated testing of HTTP API. It can be used to execute any queries through a user-friendly web interface, create tests to check the API's performance automatically, and much more. Postman is not the only service of its kind, but it's the most popular.

O imagine care conține text

Descriere generată automat

Figure 2.5. Postman http Get request to OpenTripMap API

Today, Postman is a super-popular tool. More than 8 million developers and testers use it. And here's why:

* It's free.
* Very easy to start using
* Postman is intuitive. In just a few minutes after downloading and installing it, user will be able to send his first request.
* Supports multiple APIs. Developer can use Postman to make all kinds of requests to any API (REST, SOAP, GraphQL).
* Postman can be tailored to user’s specific needs using the Postman API.
* Easily integrate test suites into favorite CI/CD tool with Newman (the CLI collection runner - lets run Postman collections from the command line).
* Has a large community. Postman is very popular and, as a result, has a large community that will answer most questions.

### 2.3.7. AJAX

AJAX is an acronym that stands for Asynchronous JavaScript and XML. In fact, AJAX is not a new technology, as both JavaScript and XML have been around for quite some time, and AJAX is a synthesis of these technologies.

When using AJAX, there is no need to refresh the whole page every time, as only a specific part of it is refreshed. This is much more useful, because client don't have to wait too long, and is more economical, since not everyone has unlimited Internet access. In that case, the developer needs to make sure the user is aware of what is happening on the page. This can be done with the use of loading indicators, text messages indicating that the exchange of data with the server. Developers should also be aware that not all browsers support AJAX (older versions of browsers and text browsers). Plus, JavaScript can be disabled by the user.

The work principle is simple. On event trigger by user’s manipulations (a button click or page load) XMLHttpRequest object is created by JavaScript. A payload will be sent to the web server and a response will be awaited. Being an asynchronous operation, the web browser will allow customer to keep navigate and use the page without any problems.

### 2.3.8. SQL and PostgreSQL

Structured Query Language or SQL is a non-procedural programming language designed primarily to describe data, retrieve them from a relational database and then to process. Therefore, SQL operates exclusively with databases and cannot be used alone to create a full-fledged application.

In this case, the tools of other languages that support embedding SQL commands will be needed. It is because of its specificity that SQL is considered an auxiliary tool for data processing. In practice, this language is only used in conjunction with other languages.

In general, application programming tools involve the creation of procedures. SQL does not have this capability. It is not possible to specify ways of solving tasks - only the meaning of each particular task is specified. In other words, in SQL databases, it is the results, not the procedures that lead to those results, that are important.

Advantages of SQL [20]:

* Faster Query Processing –
* Large amount of data is retrieved quickly and efficiently. Operations like Insertion, deletion, manipulation of data is also done in almost no time.
* No coding skills required
* For data retrieval, large number of lines of code is not required. All basic keywords such as SELECT, INSERT INTO, UPDATE, etc. are used and the syntactical rules are not complex in SQL, which makes it a user-friendly language.

You cannot make full use of databases without database management systems (DBMS). There are dozens of such systems and some of most popular are: Oracle, MySQL, Microsoft SQL Server, and PostgreSQL.

PostgreSQL is an open-source object-relational database management system. In PostgreSQL all tables are represented as objects which can be inherited and all operations on them are performed using object-oriented functions. However, the structure of stored files (and even records in them) may be very different. The main difference between PostgreSQL and other RDBMSs is the object-oriented functionality, including support for the ACID (Atomicity, Consistency, Isolation, Durability) concept.

PostgreSQL advantages are [21]:

* Easy to use.
* Has a user-defined data type.
* Open source.
* A lot of community support.
* Make use of Stored procedures.
* It supports ACID, i.e., Atomicity, Consistency, Isolation, Durability.

# Chapter 3. Application design

Before start coding, a developer should clearly define the problem his future program needs to solve. Because without a good definition of the problem, can be spent a lot of effort and time on solving the wrong problem. At this stage a simple formulation of the essence of the problem without any hints to its possible solutions is carried out, and it should be formulated in a language understandable to the user, i.e., it should be described from the user's point of view.

Another step is setting program requirements. They are a detailed description of all the features of a program and the actions the program must perform. Such requirements are sometimes also called "Functional Specification" or simply "Specification". Requirements are developed to minimize changes to the system after the start of direct development. Such requirements must necessarily be formal, i.e., documented. Working out the requirements is very important, because it allows developer to define the functionality of the program before the start of programming.

On creating a development plan stage, developer should already be formally making a plan for software development, taking into account the problems at hand and the requirements that have been worked out. In other words, developer should make a plan of how he will proceed.

System architecture development or high-level design is very important, because without a good architecture, developer can solve the right problem, but come to the wrong solution. A good program architecture makes programming easier, while a bad architecture makes it harder.

After system architecture is done, developer will work on detailed design at a low level. In other words, classes and methods are designed here, different options and reasons for choosing the final approaches and implementation methods are considered, evaluated, and compared.

When developing small programs, programmers usually design the program themselves at this level. It looks like writing pseudocode or drawing schemes, that is why this stage is often considered as a part of direct coding.

## 3.1. Application architecture

When a developer would undertake to write a small, but real and growing project, he will firsthand how important it is for a program not only to work well, but also to be well organized. Don't believe that well-designed architecture is necessary only for large projects. Complexity tends to grow much faster than program size. And if developer of the application doesn’t take care of it beforehand, there comes a point where he stops controlling it pretty quickly. The right architecture saves a lot of effort, time, and money. And often determines whether the project will survive. And even if it's just a matter of "building a stool," it's still very helpful to design it first.

There is no generally accepted term for “*software architecture*”. However, when it comes to practice, it is clear to most developers which code is good and which is bad. Good architecture is, first of all, an advantageous architecture making development and maintenance of the program simpler and more efficient. A program with a good architecture is easier to extend and change as well as to test, debug and understand. So, in fact, it is possible to formulate a list of quite reasonable and universal criteria:

* System efficiency.
* System flexibility.
* Expandability of the system.

The requirement for the system architecture to be flexible and extensible (that is, capable of change and evolution) is so important that it has even been formulated as a separate principle - the Open-Closed Principle (the second of the five SOLID principles): Software entities (classes, modules, functions, etc.) should be open to expansion but closed to modification [22].

The Client-Server architecture involves the separation of service provisioning and requesting on different computers in the network, each of which performs its tasks independently of the others.

In this architecture several client computers (remote systems) send requests and receive services from a centralized service machine (server), which may also be called a host system.

The client machine provides the user with a so-called "user-friendly interface" to make it easier for the user to interact with the server.

In the current project, was applied two-tier architecture (Figure 3.1.). Two-tire it is called because of the need to distribute the three basic components between the two nodes (client and server). This type of architecture is used in client-server systems, where the server responds to client requests directly and complete, while using only its own resources. That is, the server does not call third-party network applications and does not access third-party resources to fulfill any part of the request.

This architecture seems to be the most logical for a client-server architecture. In it, however, two variants can be distinguished. When general data are stored on the server and their processing logic and business data are stored on the client machine, this architecture is called "fat client thin server". When not only the data but also the logic and business data are stored on the server, it is called "thin client fat server". This architecture is the prototype of cloud computing.

The applied approach was the “fat server”. The client should maintain a lightweight structure and to not be aware of any business logics. Would be way easier for the system that encapsulates the business logic to communicate directly with server.

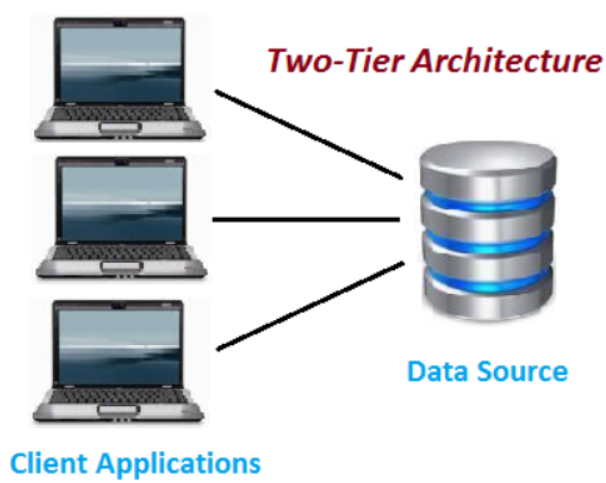


Figure 3.1. Two-Tier Architecture [image source [23]]

This type of architecture is easy to maintain and modify, and also should be considered that communication is way faster than others.

## 3.2. Application levels

As was mentioned, the application uses two-tier architecture. As presented in Figure 3.2., the client’s request is sent from *Presentation* to *Data layer.* The last one incapsulates both, *Application* and *Server* layers. The *Application layer* stores what is called business logic. Here the request is processed. This layer has direct access to the database using *Server layer*. By “*direct*” is meant that application does not apply a separate layer for each of the *Data layer.*

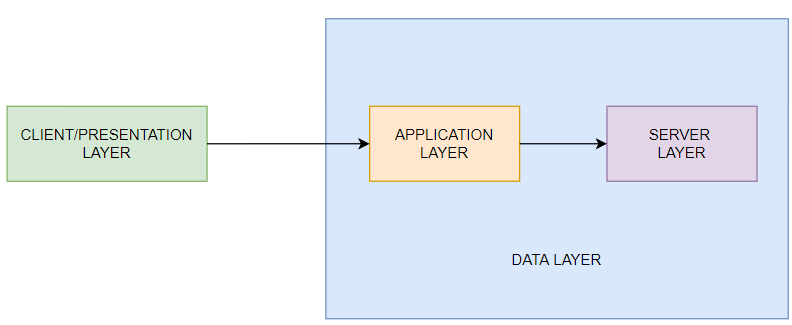


Figure 3.2. Application layers

### 3.2.1. Presentation tier

This is the first and uppermost layer that is present in the application. This layer represents the user interface, that is, the presentation of content to the end user through a graphical interface. This layer can be accessed through any type of client device, such as desktop, laptop, tablet, cell phone, etc. For content displayed to the user, the relevant web pages must be retrieved by a web browser or other presentation component running on the client device.

In more technical terms, presentation tier’s front-end is built with HTML, cascading style sheets – CSS and JavaScript. It is deployed to a web application that is able to communicate with other layers through HTTP calls. This tier should not be aware about any business logic. It just sends a request to an API and waiting for a response. Where the API get the results from is not such important from this layer. If the data provider ever changes, the customer doesn't even have to notice.

The user is considered the trigger of events. Through some manipulations on the web browser, for example a button click, JavaScript will intercept the event and using AJAX will send a HTTP call to the backend of web application. From this point the data will be validated and sent to data layer through a HTTP request as well. The response will be interpreted and sent to the user’s interface also by the web application.

To build the web application was used ASP.NET Core MVC Web Application.

The concept of MVC pattern implies the division of the application into three components:

* Model: describes the data used in the application as well as the logic that is directly related to the data, such as validation logic. Typically, model objects are stored in a database. In MVC, models are represented by two main types: view models, which are used by views to display and communicate data, and domain models, which describe the data management logic.
* A view: is responsible for the visual part or the user interface, often an HTML page, through which the user interacts with the application. A view may also contain logic related to the display of the data. A view must not contain any logic, however, about user request handling or data management.
* Controller: represents the central component of MVC, which provides the link between the user and the application, the view, and the data repository. This contains the logic for processing the user's request. The controller receives the user's input and processes it. And depending on the results of the processing, it sends a certain output to the user, e.g., in the form of a view filled with model data.

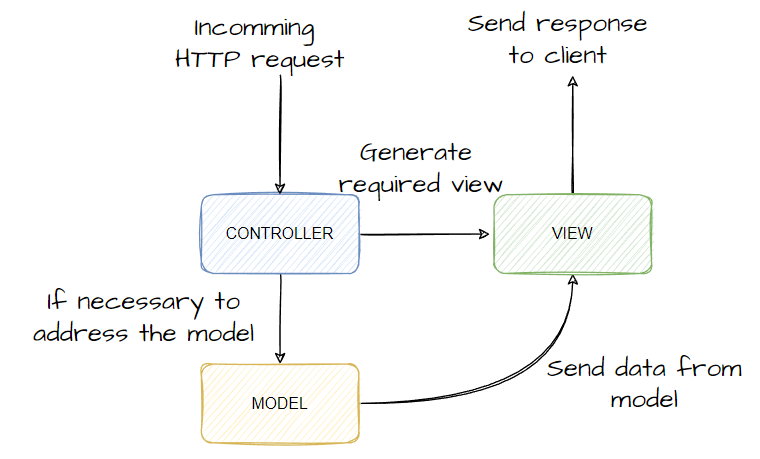


Figure 3.3. MVC request flow

In Figure 3.3., the model is an independent component - any changes to the controller or view have no effect on the model. The controller and view are relatively independent components. For example, you can access a particular controller from a view and generate views from a controller, but they can often be changed independently of each other.

This delineation of application components enables the concept of responsibility separation, where each component is responsible for its own strictly delineated scope. This makes it easier to build work on the individual components. And it makes it easier to develop, maintain and test the individual components. For example, should be cared about the visual part or the frontend, can be tested the view independently of the controller. Or can be focused on the backend and test the controller.

### 3.2.2. Data tier

This tier can be considered as a server that is responsible for request processing and transactions management. The server works on clients' jobs and manages the execution of their jobs. After each job is completed, the server sends the results to the client who sent the job. A “*job*” can be interpreted as a HTTP request.

The framework used for this tier to build HTTP services was ASP.NET Core Web API. For structuring the data tier, was used the “*Onion Architecture*”. It is the division of the application into layers. There is one independent layer which is at the center of the architecture. The second level depends on this level, the second level depends on the third level, and so on. So, it turns out that the second-dependent one overlaps around the first independent level. Around the second, the third, which may also depend on the first, is superimposed. Figuratively, this can be expressed as an onion, which also has a core, around which all the other layers, up to the husk, are layered.

The number of levels may vary, but in the center is always the Domain, that is, those model classes that are used in the application and whose objects are stored in the database. Is also shaped by business logic. The application layer’s services use this one for processing the user request data and then to receive a response for sending it to the next layer.

The first layer around the domain model consists of the interfaces that control the operation of the domain model. These are usually the repository interfaces through which developer interact with the database.

The external layer represents those components that change very frequently. Typically, the Application layer contains the lightweight services that bring data from database using repository interfaces and send them to API layer.

The last layer that has access to others is the API. It can be interpreted as API of the user interface. The controllers of API layer intercept the HTTP requests sent from web application and send them to the next layer for processing.

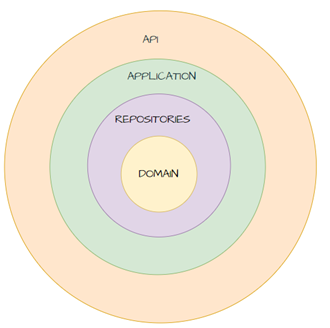


Figure 3.4. Data tier architecture

This architecture is ideal for applications with long lifecycles and complex business logic. Its use in such projects leads to excellent results because of the architecture's inherent emphasis on separating different aspects of the application. Onion Architecture pays special attention to describing system behavior in terms of contract programming and putting infrastructure code into external modules.

## 3.3 Application block diagram

As was presented previously, the application is formed by two layers. They are totally separated and can communicate with each other through HTTP calls.

In the Figure 3.5. is presented the simplified diagram of the application. The *User interface* is the graphical interface used by customer to send a request. There is no other type of users in this application. To access all services the app offers, customer should create an account, to login and that’s all. The registration is required for tracking individually user’s preferences and in that way to create a customer’s profile.

The web application *MVC Controller* provides communication between the user and the system and uses the model and representation to implement the necessary reaction to user actions. As a rule, at the controller level, filtering of received data and authorization is performed - the user's rights to perform actions or receive information are checked.

The *Presentation level* does not contain any business logic. The processing of data is done on the *Business level. Web API Controller* is similar to ASP.NET MVC controller. It receives a HTTP request and sends back a response to the caller. Based on the incoming request URL and HTTP verb (Get, Post or others), Web API finds the route and decides which endpoint to execute. These controllers are using the *Application Services*. Here can be considered each service as a manager that uses required repositories (from *Repository*) or even other services from *Domain.* Using *Repository,* the application services are saving the request data (e.g., user preferences over vacation package).

The business logic from *Domain* has only one purpose – to process the user’s request considered as a set of preferences. The whole business logic is based only on multiagent system. The purpose of the *MAS Environment* is to give a recommendation response based on user preferences.

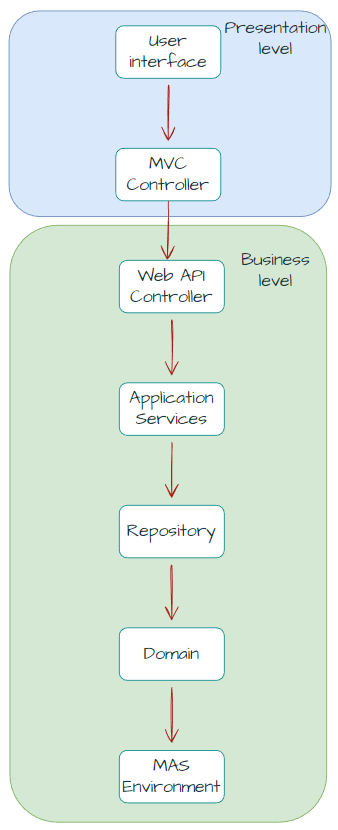


Figure 3.5. Application high level diagram

### 3.3.1. Web application block diagram

The web application has been designed to accommodate three actions: ask, receive, show. In other words, send a request to the server, receive the response, and show it to the user. In the application flow there is no data processing, the server takes care of everything. Respectively, the structure of the application is not very sophisticated. Figure 3.6 shows the flow behind a user request.

The first condition or first step is the user authentication. It was only necessary because of the need to create the user profile. This will help in the training of custom agents that will be trained based on user feedback.

After logging in, the user will be redirected to the services homepage. This page has one purpose: to search for a complete holiday package as close as possible to the user's preferences.

After entering the preferences, the user will click on the search button. This will trigger an event in JavaScript that was waiting for the button to be clicked. This function will use AJAX to trigger the MVC controller that is responsible for sending the user's preferences to the server.

From inside the MVC controller an HTTP request will be sent to the server. When a response is received, it will be returned to AJAX, which in turn will use JavaScript functionality to refresh the view and display the result.

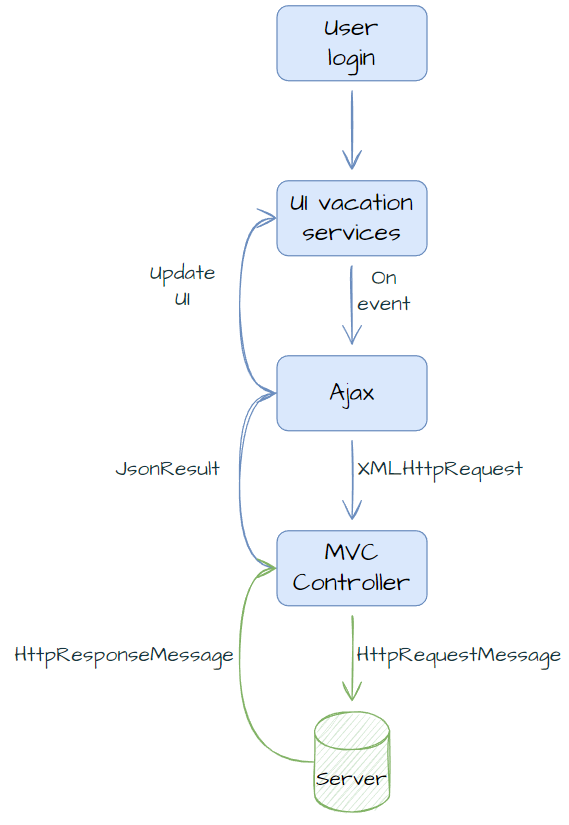


Figure 3.6. User sending a request for a full vacation package recommendation

### 3.3.2. Web API block diagram

The API application or rather the server does not only contain a part of the business logic but effectively every operation that needs to be calculated, operated takes place here. Therefore, the analysis of the user's preferences as well as the provision of an express response takes place here.

**Design of MAS startup**

The core of the API is considered the multi-agent system. Its purpose is to solve problems at an asynchronous pace. Breaking up tasks and solving them at the same time is the best strategy.

The biggest problem in designing a project to include a multi-agent system was: How will the MAS system be integrated into an API and how will they communicate?

A decision was made that the MAS environment should be created and initialized with the server "wake-up". The system instance can be invoked via a class that applies the Singleton[[4]](#footnote-5) pattern design. This was necessary to ensure the uniqueness of the MAS system instance, but also a better balancing of multi-threading access.

Access to the multi-agent system instance can theoretically be achieved from anywhere in the project, which is a disadvantage. Ideally, the system should be "*black box*". That is, the API service sends a request to the MAS and receives a response. This can be achieved by separating the MAS system from the server and creating yet another separate API. This would transform the application into a 3-tier architecture. This approach was not chosen because of the need to keep the architecture as simple as possible and to eliminate dependencies. But still, it was a good alternative.

After initialization of the multi-agent system instance, as shown in Figure 3.8., when the API is started, agent data is read from the database.

Also, from the database all information about flights, hotels or attractions is taken and shared equally between agents. A very important feature is the fact that the degree of expertness of an agent depends directly on his personal database. If the data the agent keeps changes, his rating will be meaningless.

Next, agents will be added to the MAS environment. "*Agent Coordinator*" is needed to coordinate the completion of all tasks. When this time comes, this agent will notify the “*Recommendation Service*” in an indirect way that the recommendation package has been completed. More information about that you can read in next chapters.

All agents except "Coordinator" will be marked as available. The last step in this flow is to start the MAS environment. Agents will now be able to receive requests and solve problems.

O imagine care conține text, persoană, captură de ecran

Descriere generată automat

Figure 3.7. MAS environment configuration and start

**Design of received vacation package request**

Figure 3.7. shows the complete path that a request for the delivery of a holiday package from the user takes.

Everything starts from the API controller. It receives a request from the client interface, specifically the Web Application, the body of which contains the user's preferences. This data is stored in the database using repositories. Along with the preferences, a record of the client request itself is created in the database, and the preferences, recommendations and evaluations that will be performed on this holiday request will all be linked in the database to the so-called "*client request*".

Next, information about the degree of trust of an agent in other agents is taken from the database. This data will be sent to MAS Environment.

The process of sending, checking, and receiving the holiday package is managed by the *Recommendation Service*. First the MAS system is asked to provide the list of available agents. Afterwards, the preference package is sent to be saved in the MAS memory, and the available agents are informed that a new request has just arrived.

From this point on, the multi-agent system will work to solve the proposed problem. In the meantime, the *Recommendation Service* will start a timer with an arbitrary value assumed to be 10 seconds (*10 seconds is also the timeout time of the HTTP request. If the response is slower, the Web Application will close the HTTP request and will not wait for a response*). Every few milliseconds, this service will query the MAS system trying to find out if the agents have already found a solution. If the answer is yes, the service will query the MAS to provide the recommendation package. If the agents take too long to solve the dilemma, the service will stop the timer and send a blank response to the Web Application. This scenario is not very probable, but still needed to be considered.

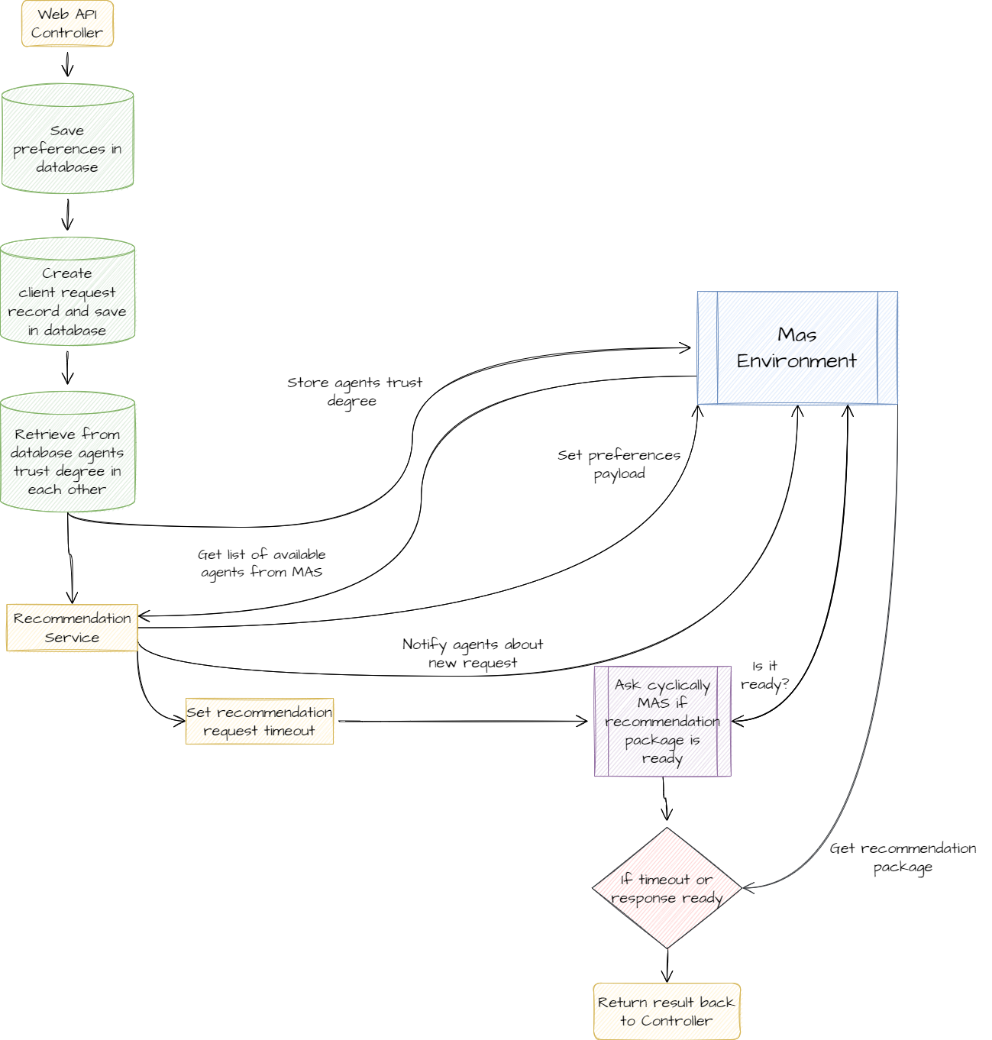


Figure 3.8. Web API receives a vacation package recommendation request

**Design of MAS processing a request**

As stated, MAS agents have a responsibility to solve a problem, in our case to find a solution for the service they are responsible for. The way agents start solving tasks is simple. The Agent Coordinator notifies the available agents about a new request.

Figure 3.9. shows the logical scheme that an agent follows when it receives a message.

First, the agent chooses the request type:

- a new request from the user

- a help request from an agent

In the first case, the user's preferences for the holiday package are read from the MAS memory. Then the agent subtracts its expert rate for the current user.

The tasks are stored in the MAS memory as a shared resource. Respectively when an agent wants to take a task from there, it must block access to the resource for other agents. The agent will choose a task according to its expert rank and available tasks.

Before finding the optimal service option, if the agent has to handle flights, it will fill in the missing preferences so that the algorithm gives the cheapest result (e.g., flight class to be *Economy*, flight type to be *Direct*).

In the case of *Property* or *Attractions* there is no need to supplement anything in the recommendation package. The agent immediately starts searching his database for an optimal offer. If one has been found, the agent stores the result in the shared memory and notifies the Coordinator that the service type has been completed (e.g., "Dear Coordinator, the result for Property is done."). If no results were found, the agent reads the trust level of other agents and sends a help message to each agent starting with the most trusted.

If the message the agent receives does not refer to a new request, it will most likely refer to a request from one of the agents for help. The message received will only specify the type of service that the other agent was unable to handle. The current agent will check if the task has not already been completed by other agents. If not, it will look for the most optimal recommendation in its database and store it in memory. After completion, the Coordinator will be notified about the completion of a particular task.

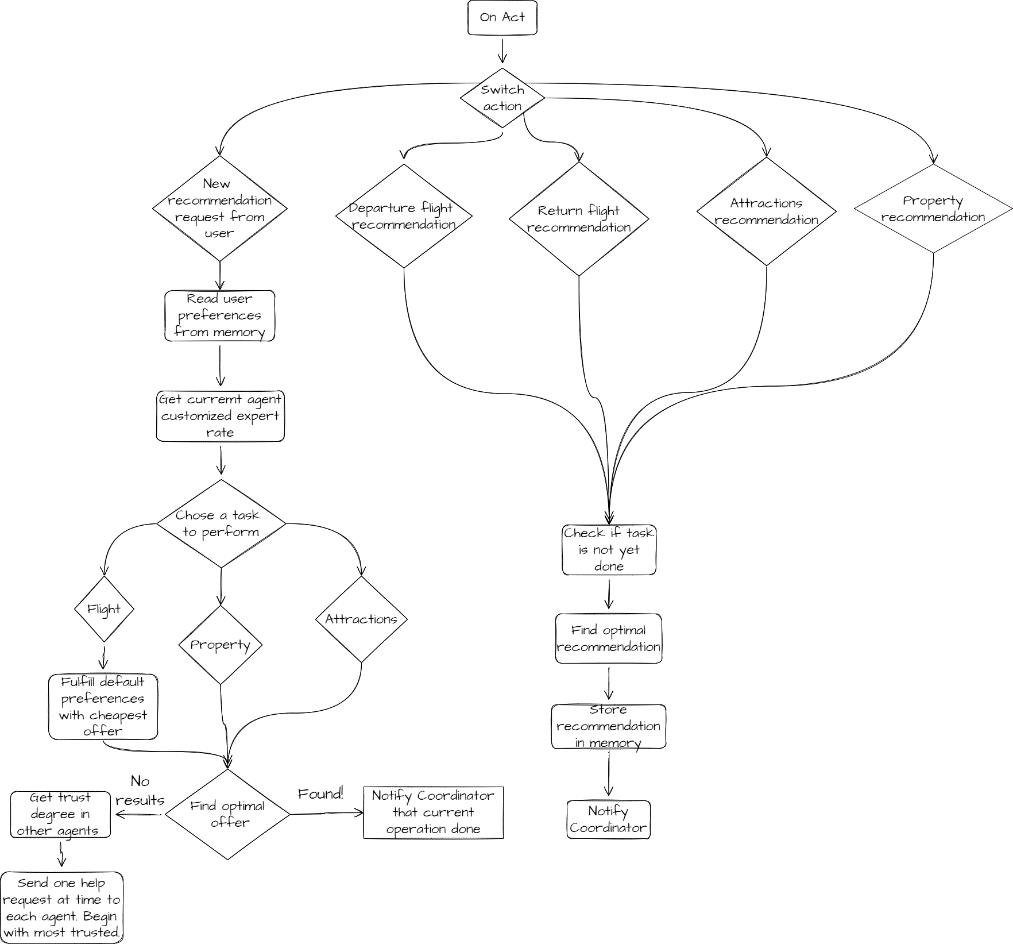


Figure 3.9. MAS agent receives a request

**Save evaluation and update agents’ rating**

An important step that shapes the choice of agents to be experts in a certain service is user feedback. This is the final process that completes the flow of a holiday package request. Every time the user requests a complete holiday package, after receiving it, the user will have to give a rating to the respective service. Obviously, in real life this option would appear after the customer has accepted the option presented and is waiting for the customer to return from the holiday. This way he will be able to give a mark to the services offered.

Figure 3.10. shows how the Web API receives a request to submit a rating of the holiday services delivered by the agents. The request is sent from the GUI. The user fills in some forms and states his liking or disliking of some services.

The API controller receives the request, and the first step is to calculate the final rating of each service, based on the user's ratings. Basically, the final rating of a service is the arithmetic average of all the ratings of all the subservices of that service (*e.g., for property will be considered the rating for Amenities, or for Property Type and others will be considered*).

The next step is to calculate and update the expert rating of each agent. The ratings given to that agent in the last 30 days will be considered. The older the rating, the more its relevance will decrease and vice versa. By definition, the expert level of an agent is determined by the tasks it has completed and is directly dependent on the ratings it has received from all users.

Another step is the updating of an agent's level of trust in other agents. A recommendation package contains a record for the agent who was first assigned to the task and the agent who completed it. With this data, as well as the customer's evaluation, the algorithm will be able to calculate how many times the help from a particular agent was successful and how many were not.

The personal agent rating is calculated in a similar way to the self-expert rating. The only difference is that only records that refer to data processed by the agent exclusively for concrete client will be considered.



Figure 3.10. Save evaluation process

### 3.3.3. Database

A database is an organized structure designed to store and process interconnected information. Its use allows you to reduce the load on the server, increasing the speed of data and content pages, as well as increasing site security, which is important.

In the current project was used relational database in PostgreSQL. This particular database system was chosen because it is quite easy to use and it was desired to try a new technology, different from the known ones such as Microsoft SQL Server.

Creating the database design involves knowing exactly what the project aims to achieve. It is quite difficult to create a database that does not require changes to its structure over time.

In the course of the project the database was modified several times until a relatively optimal version was reached. Existing technologies had to be taken into consideration, but also constraints that could affect the structure of the database. For example, the biggest problem was: "Where will agents get their information for recommendations? Will there be an external API that will provide the necessary services, and will agents just send a request to the API and get a response?".

This is where resource constraints and limitations came into play. The simplest way to work with data would have been to find external APIs. For example, the Flight API [24], which provides data about flights and their prices in real time, or the Booking.com API [25] which provides data about apartments and hotels for booking. But APIs that work in real time in a travel domain cost money. Every API call is charged. For this reason, even large companies when choosing the technologies, they will work with, always calculate the price of using foreign services.

The project required the creation of a database table for each type of service: flight, stay properties and tourist attractions. The current structure of relational database can be seen in the ***Annexes***.

The design of the reports for flights or properties to stay were made according to an improvised model, which could store the most important information about the respective services (**See in Annexes** *Property relations, Flight relations*).

The request of the client for full vacation package was designed in three divisions: preferences, recommendations, and evaluations.

The simplest part was modelling the first section – preferences (**See in Annexes** *Preferences relations*). Here it is quite clear what the user's preferences on already known holiday services might be. Plus, there are already sites that offer flight or hotel booking services, such as *Cheapflights* [26]. It was quite simple to analyze all the preferences a user might exhibit due to sites offering similar travel services. Except that these are based on only one service, whether it's just flying or just staying properties.

The development of the tables for the recommendation process (**See in Annexes** *Recommendation relations*) involved more advanced knowledge of the system itself. At this point the agents in the MAS system come into play. The recommendations are made by them, i.e., in addition to the data about the recommendations, each table containing the recommendation of a type of service also indicates two additional fields: the initial agent responsible for performing the task and the agent who completed the task. Thus, it will be easy to calculate the degree of trust of the agent, who was initially supposed to solve the task but failed, in the other agent, who helped the first one and finished the task.

For the creation of evaluation reports, it is necessary to have the design ready for the recommendation or preference relationships. Because the user will evaluate exactly the services that the agents have proposed as a recommendation.

The realization of the database relationships for the agents of the MAS must be approached separately (**See in Annexes** *Agent relations*). Only after studying the necessary documentation and forming a concrete vision of how the multi-agent system would work, it was possible to create the agent tables. It was necessary to create the agent table itself, but also those of trust in other agents. The expert rate of agents for a specific client also refers to the multi-agent system (**See in Annexes** *Customer relations*).

# Chapter 4. The application development

The main embodiment of the final project is realized at coding stage. This is precisely the stage that everyone knows and probably thinks is the only stage in the software development process when developer direct writes code and debugs it. But, as it was observed, this is not the first and not the only stage of software development. The design stage of the project can also be considered a development stage. This is where schematics, pseudocode sketches and even some algorithm tests are made.

However, in addition to applying business logic, there is also scope in the development process to consider ways of optimizing and rectifying concepts developed at the design stage. The design and development phases complement each other. Sometimes the development part might find vulnerabilities in the design scheme or vice versa.

If it has not been done in the design phase, then on the development side there is also the research phase of external services such as APIs or technologies that would optimize the process. Thus, it can be concluded that application development does not only include code and no more.

## 4.1. Database

For database management PostgreSQL was chosen. It's a fairly easy to use tool, has a friendly interface, but most importantly it supports ACID concepts.

**Database generation**

For the design and realization of the database it was decided to apply an automatic method to generate the entities and the relationships between them. In order to achieve this, a separate application was created in .NET, which through migrations defines the database schema. *Fluent Migrations* was used as a framework for this. Obviously, the *Entity Framework* could also be used, which is a very popular and powerful tool, but it incorporates too much functionality that is not required at this stage. In addition, *Fluent Migrations* is perfect for frameworks that are not *Object-Relational Mapping*.

O imagine care conține text, masă

Descriere generată automat

Figure 4.1. Database migrations

Figure 4.1. shows all the migrations that were used to create the database tables, the relationships between them, and the definition for the indexes. As a pattern is used the keyword Migration and then should be inserted the date and time the migration was created, plus a suggestive name. The full name of the migrations is very important, because this way *Fluent Migrator* will know the loading order of all migrations. If the migrations are loaded incorrectly, there is an inevitable risk of creating some database conflicts.

O imagine care conține text

Descriere generată automat

Figure 4.2. Example of initial migration to database

In the Figure 4.2. you can see an example of a migration. It is a simple C# class that inherits the *Migration* base class or as in our case *AutoReversingMigration*. Both belong to *Fluent Migrator*, the only difference being that *AutoReversingMigration* will know how to revert the data included in the *Up* method, effectively performing the reverse of the actions in this method. The Figure 4.2. example shows the creation of the "*Country*" and "*City*" tables. *City* will have a foreign key pointing to the *id* of the table "*Country*".

After setting up the database connection and configuring the necessary communication services, the migrations are ready to be uploaded.

On running the application, in the database *Fluent Migrations* will create a read-only table that will take the evidence of all the migrations that were already merged. In the Figure 4.3. you will see the order the migrations were uploaded.

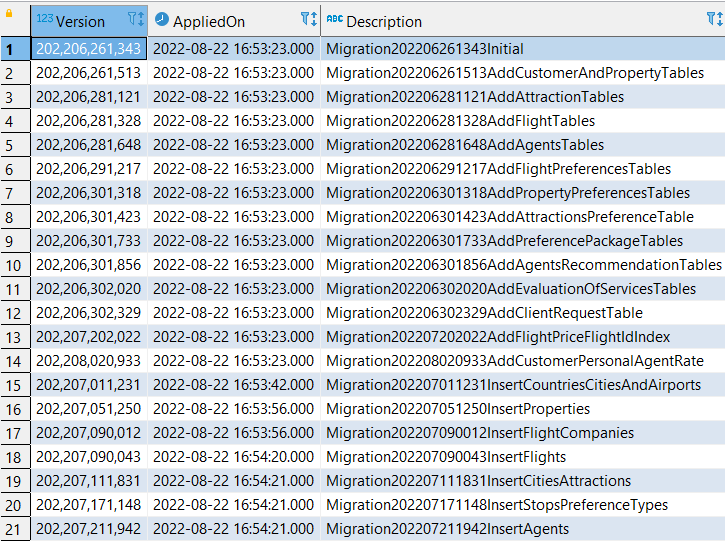


Figure 4.3. VersionInfo Table created automatically by Fluent Migrations

That’s why on running the database migration application, the *Fluent Migrations* will check the migrations and will pick for upload only those that have not yet been applied. In the Figure 4.4. is presented the situation when was runt the migration application, but all migrations were already applied successfully and there is no work to do.

O imagine care conține text

Descriere generată automat

Figure 4.4. Console log of database migration application

**Database population**

Assuming that is required to create an application that will provide users with vacation packages, one concern would be providing the multi-agent system with a dataset to work with. Preferably this data should be as large as possible, thus highlighting the advantages of a multi-agent system.

The same Fluent Migrations Framework was used to populate the tables. Only this time the migrations will be used for a different purpose - to create new records in the tables. But the first thing that was needed - finding the data or generating it.

For example, for properties to stay the database belonging to Airbnb [27] was chosen. From this database the names of some properties and their type were copied: hotel, house, apartment. All data were inserted into a .csv table. The other properties that are required to be indicated for this service in the database were randomly generated using *Kutools for Excel* [28]. It is an extension that increases the possibilities of Excel.

Creating flights and their schedules was more difficult. Here routes have to be created that are bi-directional, i.e., if the customer flies into a city, they have to have a flight company to fly back with and in addition diversification and multiple data is indispensable. It should also be taken into account that a route can fly several times a week and several times a day. *Kutools* was used to generate this data as well.

In order to have some data that is at least a little closer to reality, it was decided to find all passenger airports in Europe and create routes from these airports. Data from *Airportcodes.io* [29] was used as a resource for airport data. As in case of properties, data for the flights were stored in a .csv file.

The *OpenTripMap API* [18] was used to find tourist attractions in a concrete city. Obviously, if needed, agents could have called the API themselves and asked for the necessary attraction recommendations. But it was desired to keep the pattern of previous services, i.e., populating the databases with records. For this reason, a script was created to take the API data about the tourist attractions in the cities where our "multinational company" has airports and insert them into a .csv file.

After all the .csv data files for the three services: flight, properties, and attractions, were ready, the script for populating the database with values was implemented. For retrieving data from files and converting them to C# database models was very helpful the *CsvHelper* [30]. Thus, after reading from the .csv files, *Fluent Migrator* uploads the records to database using same approach of uploading migrations.

## 4.2. Data tier

The data level involves the implementation of business logic. As thought at the design stage, the basic implementation will be done on the server side.

*ASP.NET Core Web API* with platform on .NET 6 was used to create the server design. *.NET* was chosen as the application programming environment because it is modern and is in continuous development. Another plus is the presence of the documentation. Microsoft likes to detail in depth the services its platform offers.

The design phase offered the possibility to create logical schematics that the implementation phase will develop in code. The first step of the implementation was the creation of the application layouts that would follow the "Onion architecture". After that it was necessary to configure the processes that will take place at startup (adding services, configuring the database connection, and loading the multi-agent system).

An interesting feature of this project was the implementation of the Dependency Injection design pattern. It was used in the services so that they do not depend on some hard-coded objects but can always be replaced with others. Plus, Dependency Injection fits perfectly with the architecture of the project, making it easy to test and validate functionality.

The design phase offered the possibility to create logical schematics that the implementation phase will develop in code. The first step of the implementation was the creation of the application layouts that would follow the Onion Architecture. After that, it was necessary to configure the processes that will take place at startup (adding services, configuring the database connection, and loading the multi-agent system).

An interesting feature of this project was the implementation of the Dependency Injection design pattern. It was used in the services so that they do not depend on some hard-coded objects but can always be replaced with others. Plus, Dependency Injection fits perfectly with the architecture of the project, making it easy to test and validate functionality.

Figure 4.5 shows an example of dependency injection from the project. Repositories are used to handle database operations. Thus, if is required to use a database resource, should be created the repository service that is strictly responsible for that process, define its interface and inject it into the required service. In this way, the services in the *Application* layer will have access to those belonging to the *Repositories* layer (see description in Chapter 3.2.2. Data tier) through *Dependency injection*.

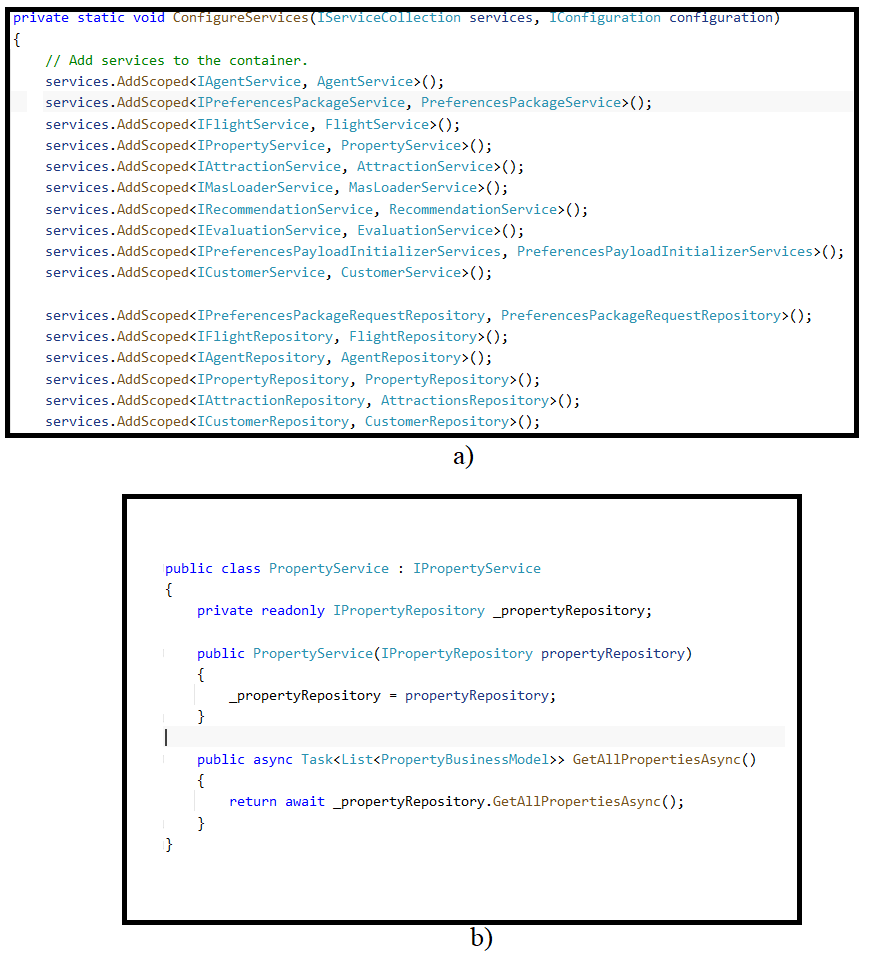


Figure 4.5. Dependency injection of the **Property Repository a)** in the **Property Service b)**

The core business logic implementation is focused on the multi-agent system. All requests for a complete holiday package arrive in MAS. The steps outlined at the design stage have been fulfilled exactly, but there were also some implementations that contributed to adjust the process of finding an optimal recommendation by the agents.

**Similarity match with user needs**

When agents receive a set of preferences on the holiday package, they should somehow check their knowledge base for the best match. But the idea of "best match" can be controversial. One thing is for sure, it should benefit the client as much as possible.

Ensuring a good result for the customer would mean knowing their general preferences. Somehow it would imply the need to create a profile of the customer, and to each call from the customer, the agents could give a response based on the preferences of the profile created. This implementation can be achieved using ML.NET [31] which involves the use of machine learning. This framework uses *matrix factorization* [32] for training the data and estimating a suitable recommendation.

The project presented in this paper tended to present a startup version of a project. Thus, ML.NET can be used in the future as an extension of the functionalities and increase the efficiency of the agents' results.

A simplified variant of finding a best match was the use of the method presented in the paper "*Improving recommendations through an assumption-based multiagent approach: An application in the tourism domain*", chapter 3.7.1. *LocalSearch*, equation 3 [5]. In this resource a very simple way of comparing the similarity between two entities is presented - the application of the *Euclidean Distance*. In simple terms it checks how many matches there are between user preferences and potential recommendation. The higher the number of matches compared to the number of unmatches, the more the recommendation tends to be accepted and sent to the user.

**Update agents’ ratings**

Within the multi-agent system, agents choose a task that most closely matches their expert grade for that service. As explained in the design chapter, the rating of agents depends solely on customer feedback.

After each request of a vacation package, the user will have to fill in a form with his evaluations about the services provided. From the UI application the payload will be sent to the API or actually server. This is where the final rating of each type of service, i.e., flight, property, and tourist attraction, is first calculated. The formula is simple, the final rating given by the client to a specific service is the arithmetic average of all the subservices in that service. For example, for tourism attractions, each attraction will receive a rating of "Liked" or "Didn't like". The server will read the data of all ratings for the attractions and calculate the average of these values. This will be the final rating of the tourist attraction service.

So logically, the agent's rating for a given service will be conditioned by the last task he completed, i.e., exactly that final calculated rating of that service. The formula would be simple, take all the ratings the agent has received for that service and calculate the average.

But here come two problems with this formula. The first is loading too much data, and the second is the irrelevance of old data. People change their preferences over time. Why should the agent's rating depend on all the preferences the customer has ever shown? The best solution would be to consider the freshest feedback from the customer.

Not in great detail, but this principle has been discussed in "*Improving recommendations through an assumption-based multiagent approach: An application in the tourism domain*", chapter 3.5. *Agent specialization and trust* [5]". Here a formula for calculating the trust degree is presented.

(1)

This formula has been readapted for use in the current project. Thus, in equation (1) is presented the formula for calculating the relevance rate of an evaluation where **q** **­-** is the final rating, **r** **-** is the relevance coefficient and **t** **-** is the number of days elapsed since the date of calculation of the final rating (i.e., since the date of the user evaluation).

*Figure 4.6. User evaluation relevance in 30 days cycle*

The coefficient of relevance is a constant, in this project the value 0.16 was considered. This value was not chosen randomly, but several simulations were run with the consideration that the maximum lifetime of the evaluations would be 30 days. This means that the current day's rating should have maximum relevance and the rating given should not be altered, and the 30-day old rating should be completely irrelevant, i.e., null. In Figure 4.6. the degree of relevance of customer feedback is demonstrated. This formula was applied to calculate the agent's expert level for each vacation service (this includes the general and personal expert rate).

**Optimization of API’s response time**

Since in the current project is used a REST architecture, comparable to the WebSocket architecture, must also be considerd the execution time of the whole recommendation process. HTTP calls have a set time to wait for a response. If the server does not respond in time, the request will be cancelled, and an error message will appear.

Because of this, it was also demonstrated at the design stage that there is a timeout for waiting for the recommendation imposed on the MAS system. That timeout was just an assurance that the server would be able to provide a response, but it was also necessary to make a decision to optimize the server response time itself. This was achieved by applying the "*fire and forget*" principle. That is, triggering some functionalities without having to wait for a response from them. For example, at the end of a request to provide a recommendation, information about that recommendation is stored in the database. This is not vital for providing a response to the client. Thus, in order not to take time away from the response time to wait for the database save, this process is invoked without waiting for something to result. Such an approach is quite risky because this process is uncontrollable and may create some conflicts. For this reason, when implementing "*fire and forget*" a good assurance must be given that the data always reaches its destination.

## 4.3. Presentation tier

**Front-end**

The implementation of the user interface involved finding a GUI template that would fit the main purpose of the project - data entry functionality for a complete holiday package. Thus, the Colorlib’s Travelix template [33] version was chosen...

After adjustments and redesigns, the search page for a holiday package looks like attached Figure 4.7. The recommendation pack is only five mandatory details that the user must indicate: departure and destination city, check in date and period of the holiday and how many adults will fly. The other details are absolutely optional. But the more preferences the user gives, the closer the recommendation will be to what customer wanted from the start.

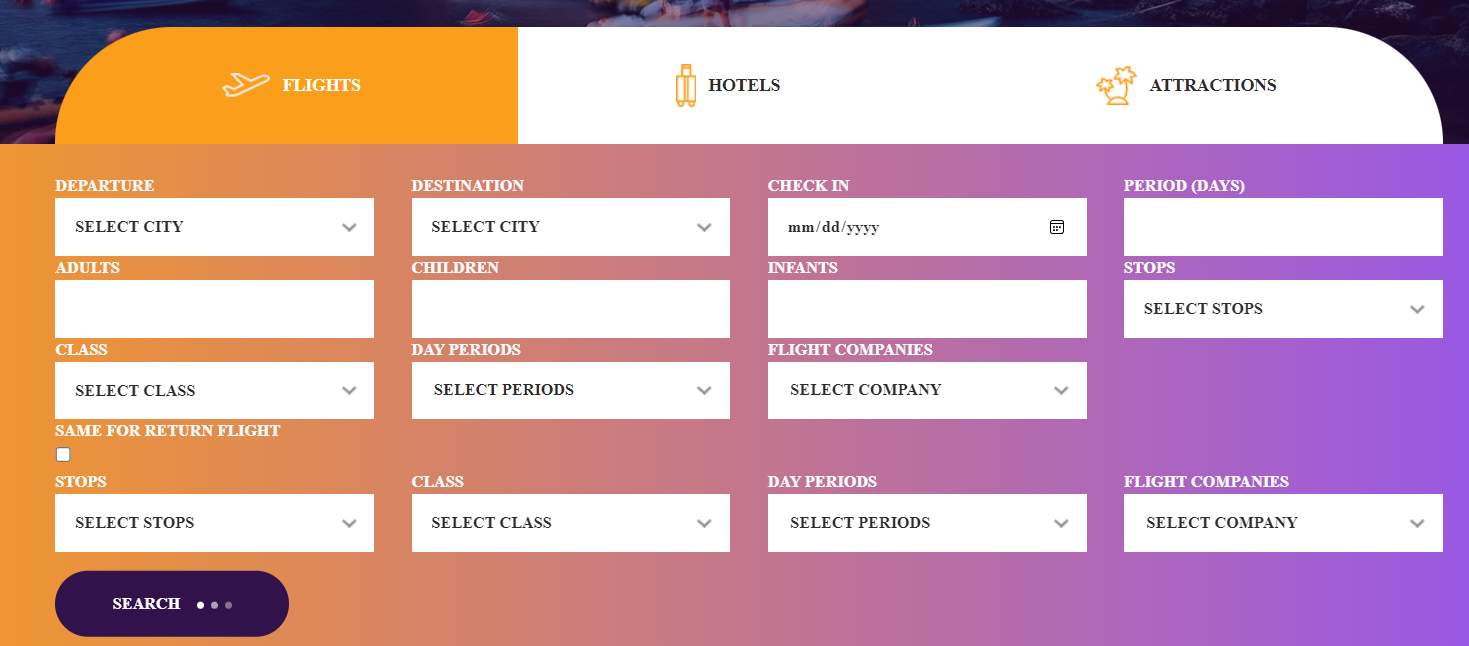


Figure 4.7. Graphical interface of vacation preferences

A special feature of the holiday package search GUI is the loading of certain data from the database before sending the request to receive a recommendation. So, it appears that yes, the user makes requests to the server, probably without being aware of it, even before clicking on the Search button.

The list of starting cities is built from the moment the page loads. An HTTP request is made to the server asking for the cities data. But only when the user selects a specific city from which he wants to leave, an HTTP call is sent to the server to load the available cities as destination. These are found by the available flights from the start city.

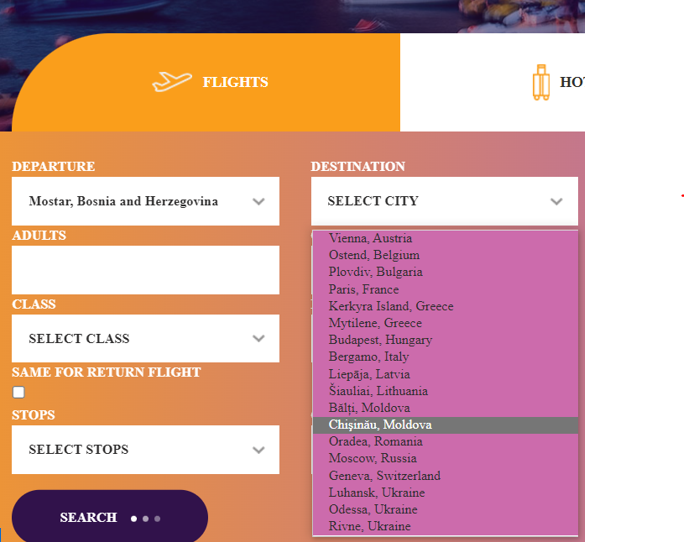


Figure 4.8. Selecting Departure and destination City

The result of a request for a complete holiday package is shown in figure 4.9. Data are presented for each type of service, the data being as close as possible to the user's preferences.

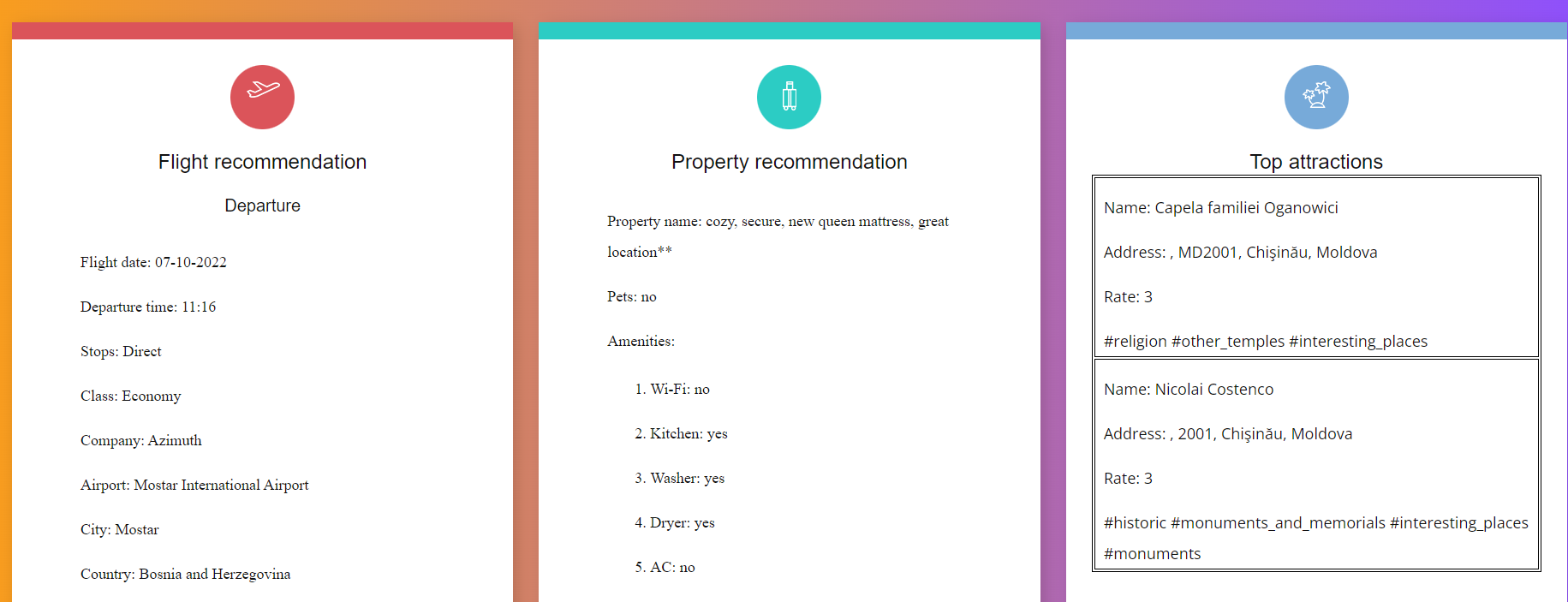


Figure 4.9. Recommendation result displayed on GUI

In Figure 4.10 is shown the form that the user should fill in when returning from holiday. This is the feedback that the user sends to the server to appreciate the work of the agents. This is where the calculation of the agents' ratings and their training as experts starts.

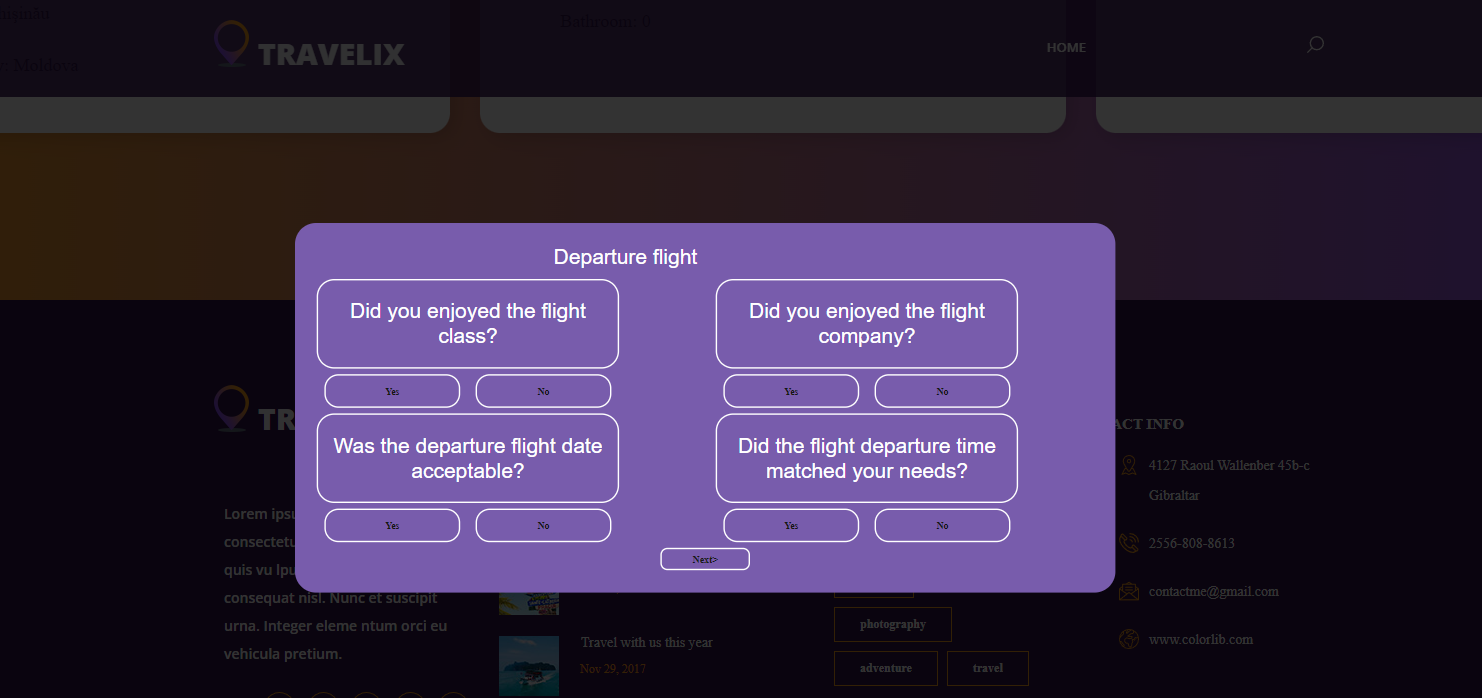


Figure 4.10. Evaluation of services formular

Another important interface is the login and registration page. This was necessary to create a user in the database, which each time would receive personalized results according to its evaluations. This is also directly related to the creation of agent ratings as "personalized agents", i.e., the same agents with only one rating for each type of service customized by user feedback.

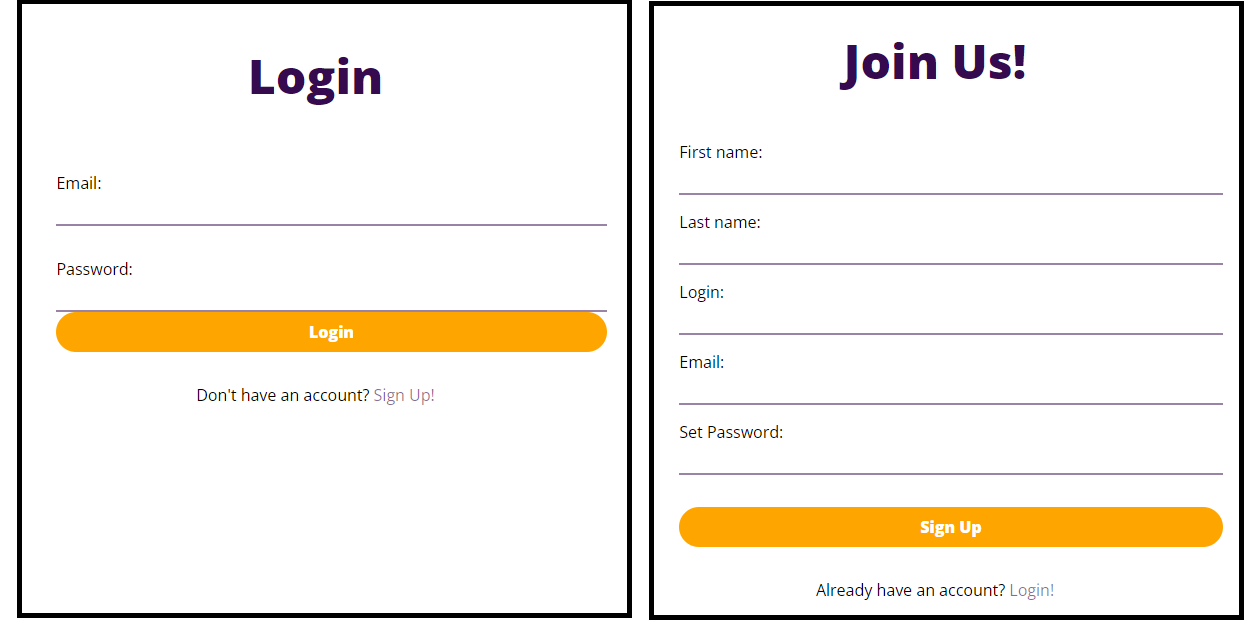


Figure 4.11. GUI for login and registration

**Back-end**

For the back end of the GUI, schematic diagram logic from the design section was used to implement the code. One feature that was not conceived at the design stage was finding the most efficient way to retrieve the UI data entered by the client.

If Ajax had sent all the data entered by the client in a single call to the Controller, it would have been much more difficult to parse that payload. Thus, the decision was taken to send separate calls containing certain properties of the vacation services the user wanted (e.g., selected preferred flight companies, part of the day to fly most preferred and others). 

Figure 4.12. On HTML interface a) the client pushes the Search button. The jQuery handles the button click event b). Here using Ajax is sent a POST request to the MVC Controller c)

In Figure 4.12. a small part of the processes that take place when pressing the "*Search*" button is shown. In jQuery the values of the data boxes that the user has filled in are read and then sent to the Controller. In each Controller, the *Singleton* instance of the preferences package will add the information received from the request to its own container. After sending all the necessary information from AJAX, the last call from here will be the one that will signal that the preferences package has been completed with the data from the UI and can be sent to the server.

# Chapter 5. Application testing

After the code is written, the programmer must debug his code to make sure there are no bugs in it.

**Component testing**

After the code is written and debugging is performed, it is necessary to test the implemented functionality. If the program consists of several components, each component is tested separately at first, since very large programs include huge functionality, which is often divided into separate components, which are developed separately. In smaller projects, this step may simply include testing of individual classes.

**API testing**

Component testing took place in various methods. As an example, for the testing of the server application it was necessary to create a *Console Application* that simulates the entire request flow of a complete holiday package, following these steps:

1. Create record of a new user in database directly (It’s done manually, not from Console App)
2. Get the list of departure cities for vacation
3. Select a random city of departure
4. Get the list of arrival cities, considering previous departure city
5. Select a random city of arrival
6. Get flight companies that can bring customer from departure to destination
7. Select some of them to store as preferred ones randomly
8. Get flight companies that can bring customer back home
9. Select some random to store as preferred for return flight
10. Fulfill preferences package with required data. This means that the fields user id, departure date, holiday period and number of people who want to travel must be filled in. Otherwise, the other data are not mandatory, but can also be entered to test the correctness of the interpretation of the data by the MAS system, which should be able to take into account every preference (e.g., preferred flight companies).
11. Send preferences package as a payload of a HTTP POST request to server
12. Read recommendation response from API
13. Fulfill evaluation of the API response
14. Send it to API through HTTP request

These steps are sufficient to follow the workings of the multi-agent system. But interpreting the data is more difficult. It is quite hard to interpret a JSON payload when it has a large amount of data. Because of this, a simplified system for logging the most important data has been created in the API.

With each customer call, a file is created on local disk in *Log* folder, that keeps the user's registered preferences, the recommendations offered, the agents assigned to a recommendation service and those who actually made the recommendation. And if this will not be the first call, information about the agent rating updates for each service, a table with how many recommendations each agent has completed for each service separately (for the flight being two recommendations, one for the outbound and one for the return flight will be considered as two) and another table with the personal agent rating of each agent for each service will also be visible.

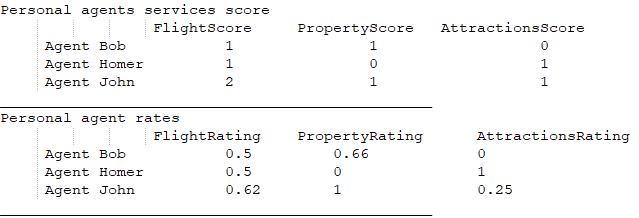


Figure 5.1. Example of a sequence from a log file

Also, for a better tracking of the interactions between agents, the messages the agents receive or send are recorded in a text document, as well as on the local disk. These messages are sent directly from the multi-agent system. An example is presented in the Figure 5.2.

O imagine care conține text

Descriere generată automat

Figure 5.2. Example of logging agents’ interaction

It is impossible to create integration tests for the entire preferences request flow of a complete holiday package. It is true that on each API project load, agents will have the same knowledge base, but it must be taken into consideration that the MAS algorithm, when it comes to no recommendation and has several choices that are equally good, takes one randomly. Thus, from an extremely large amount of data, cannot be concluded exactly which solution the agent will choose. In addition, when a new client appears in the system, it is logical that the rating of custom agents is zero, i.e., it is not known which agent will take a particular service. What remains is the analysis of the data in the log files and possibly, if necessary, the analysis in the database.

Figure 5.3. Time required to receive a full vacation package from API

Since a REST architecture is used, the request processing time for a recommendation has to be considered. In Figure 5.3. the resulting time for one hundred requests of a complete holiday package is shown.

The average waiting time for a result is approximately 181 milliseconds. Which is a pretty good result. If processing time adjustments were needed, then the place of focus would be the multi-agent system. The easiest example would be the application of search algorithms such as "*Binary Search*". Perhaps even implementing an agent-based algorithm to search more efficiently across a large collection of data.

**Web application testing**

Specific methods were used to test the web application. For the back end, was used POSTMAN which sent HTTP calls to the MVC Controller of the application. This way it was possible to check the correctness of the data transmission, the response code, and its body.

For the front end, it has been verified that all data entered by the user are correctly sent to the back end. Another way of testing and debugging was to use the development tools offered by browsers (e.g., Microsoft Edge or Google Chrome). Thanks to them a programmer can see the front-end errors of the web application, but also debug data from JavaScript.

However, there are also manual tests that do not require a setup or the use of other tools. For example, when logging in, if the user enters the wrong data, a warning message will be displayed.

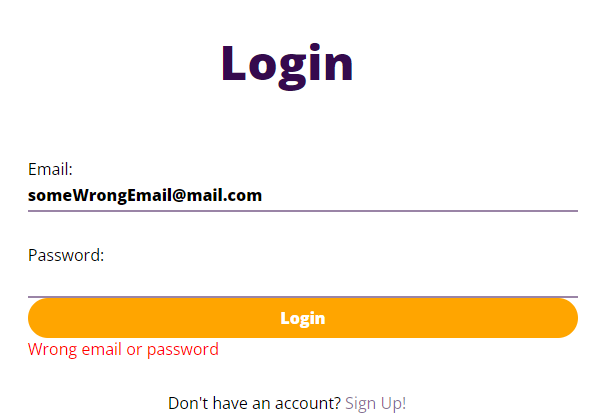


Figure 5.4. User enters wrong login or password and gets error

Or another example of testing would be trying to access the main page that offers vacation services to the user without being logged in. As shown in Figure 5.5, the front-end template used from *Codepen.io* [34] will display an error page.

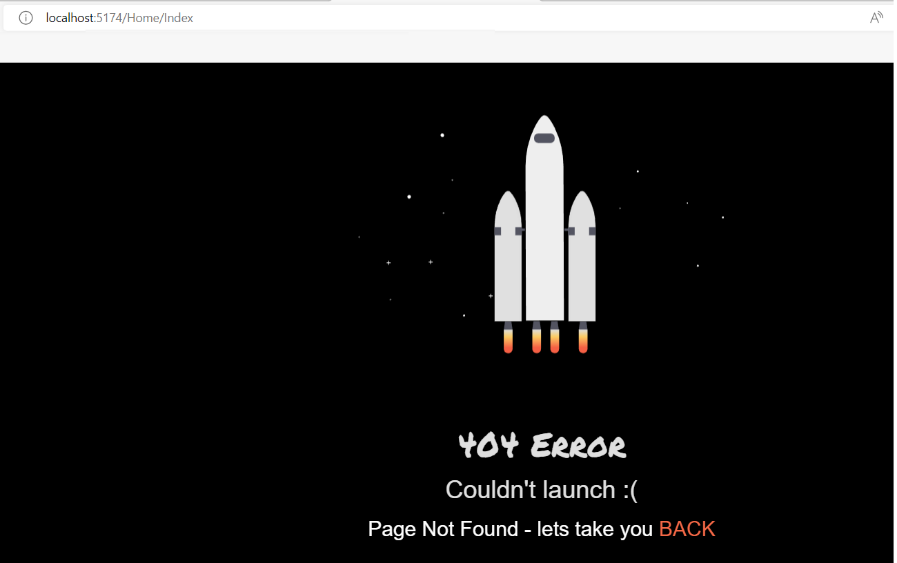


Figure 5.5. Access the vacation services page without being logged in

**Component integration**

When testing of all the components is complete, can be moved on to the integration of all the components into a single software package, this stage is exactly the process of integration, i.e., merging of all the components into a single system.

**Testing of the entire system**

At this stage, testing of the entire system is carried out, considering the integration of all components. At this stage, developer can identify problems with the interaction of components and fix them. Also at this stage, the main subject of testing is security, performance, resource leakage and other things that cannot be tested at lower levels of testing. The programmer must ensure that the data packet sent from the web application will arrive in a correct format to the API, and from there the data will be stored in a secure and correct way in the database.

# Chapter 6. Conclusions and future directions of development

This paper presents a web application based on the use of a multi-agent system to recommend a complete holiday package to the user. This system will be composed of 3 agents, also called "*tour agents*", each for one type of service: flight, property to stay and tourist attractions, and there is one more agent type "*coordinator*", who manages the completion of tasks and the announcement of the request completion. When the system receives a request, the coordinator will notify all agents that they have work to do. The travel agents will access the "gift basket" one by one and will choose one "gift" at a time, i.e., they will choose a service to complete. After searching for a suitable recommendation in its own database, the agent finding a solution will inform the coordinator, or if not found, will ask for help from other agents.

So, can be concluded that the multi-agent system aims not only at sharing tasks and executing them at the same time, but also at providing the opportunity for agents to talk to each other to solve a problem.

Over time, agents consolidate their level of expert in a particular service. Thus, each will have their own knowledge, and cooperation between agents will result in giving the client what they requested. Expert agents become experts exclusively because of the feedback that users provide. Because of this, when choosing a task, the agent will choose it based on its rating created exclusively by user evaluations.

Another feature of this app is that the user does not have to enter much data to receive a recommendation. Just with a minimum required, agents can find the necessary data, such as: departure and visit city, check-in date and holiday period plus the total number of adults.

When providing a response, agents do not search the user's request history to make an assumption of their preferences. Thus, as an improvement, a machine learning implementation could be added to study the user profile and make a much better offer based on his preference history.

Another way to increase the performance of the system is to add more powerful algorithms to search for potential recommendations. If it is assumed that a company's database stores millions of results, then processing all the data to find the best match is totally inefficient. This requires too many resources and time. One way would be Linear search, Binary search, Interpolation search, Hash table.

In principle, a search algorithm is not necessary if the multi-agent system would use an API to provide data about each holiday service. That is, there would be no need to store in its own database records about flights, or properties, or tourist attractions around the world. Agents could actually make an HTTP request to the API and receive the necessary responses. It would be a good practice, except that in this scenario it diminishes the usefulness of the multi-agent system. Because it should be the one that studies the user profile and based on the information it has to give a response.

One improvement that would certainly change the way the server communicates with the web application would be to move from *RESTful* to *Web Socket* architecture. The biggest advantage of this being the creation of a channel between client and server and maintaining continuous real-time communication.

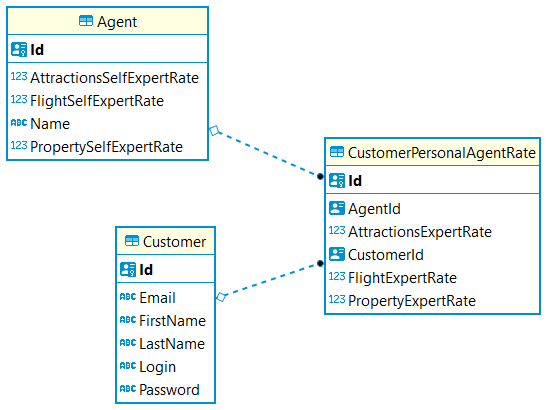


Figure A.1. Customer relations

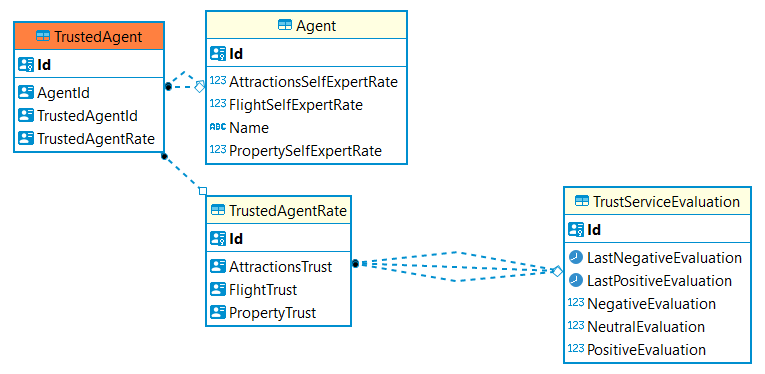


Figure A.2. Agent relations

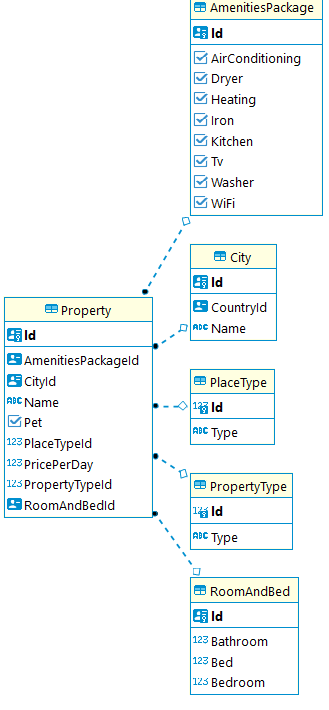


Figure A.3. Property relations

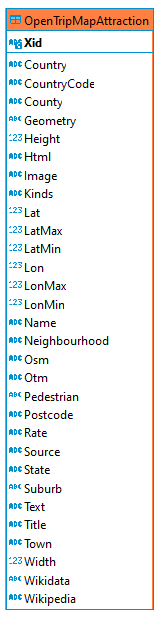


Figure A.4. Attractions table

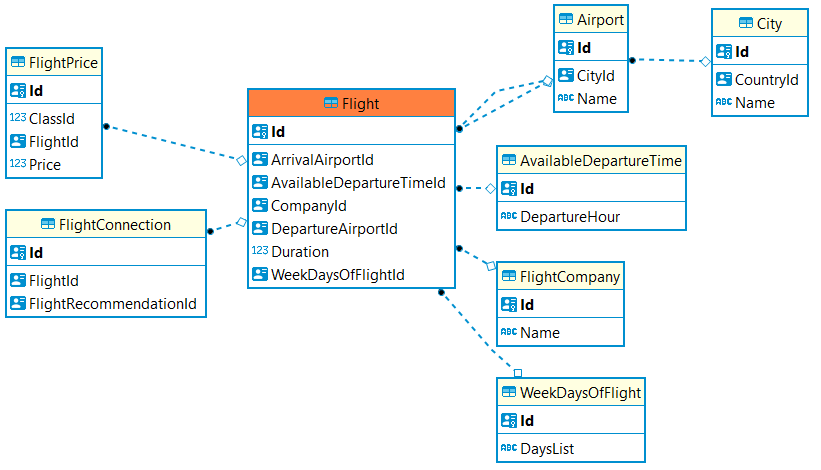


Figure A.5. Flight relations

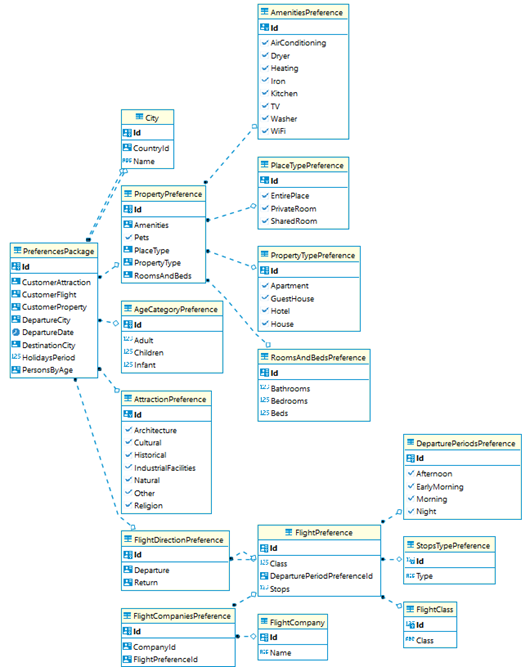


Figure A.6. Preferences relations

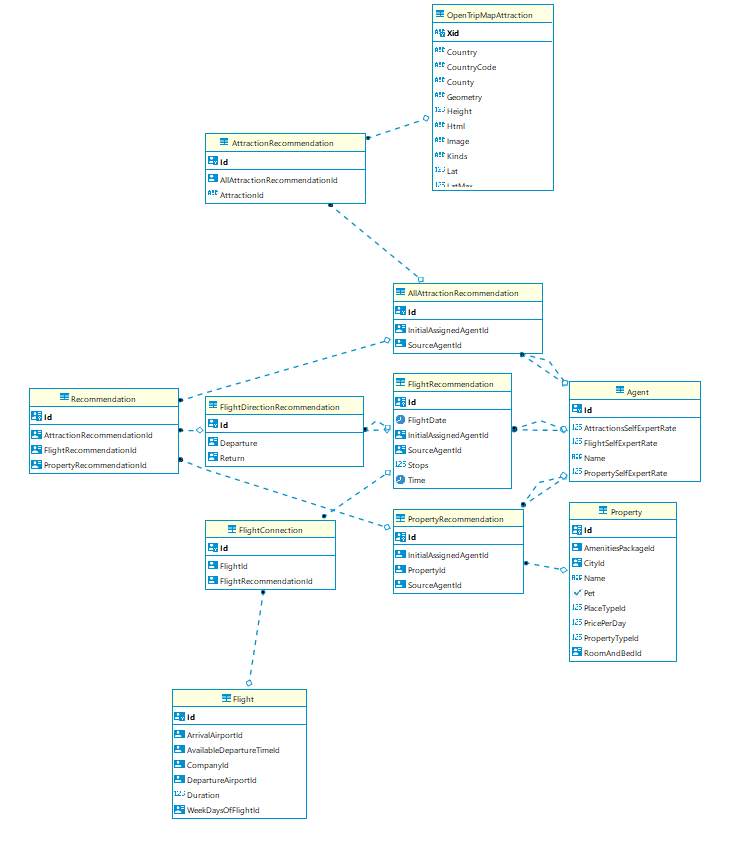


Figure A.7. Recommendation relations

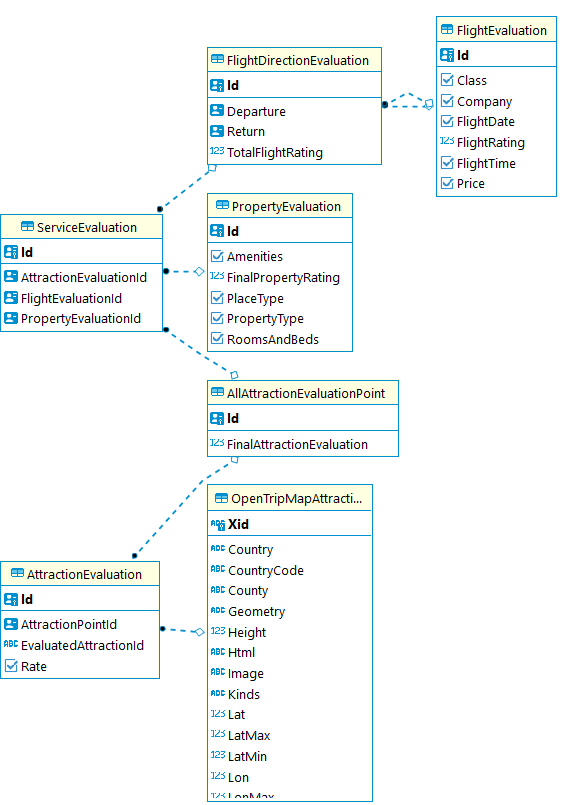


Figure A.8. Evaluation relations

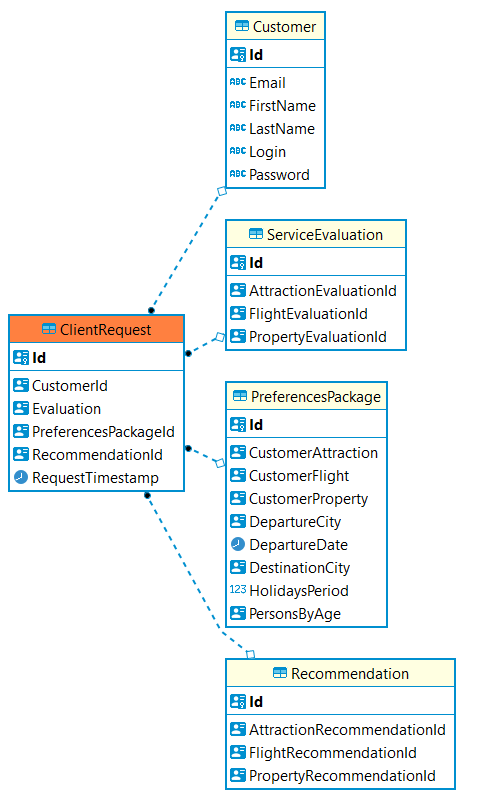


Figure A.9. Client’s request relations

[1]https://www.weforum.org/agenda/2022/06/international-travel-2022-covid19-tourism/#:~:text=The%20UNWTO's%20latest%20World%20Tourism,compared%20to%20the%20previous%20year.

[2] David Camacho, José M. Molina, Daniel Borrajo, Ricardo Aler, “MAPWEB: cooperation between planning agents and Web agents”, 2001

[3] F. Lorenzi, D. S. Santos, D. de Oliveira, and A. L. C. Bazzan. Task allocation in case-based recommender systems: a swarm intelligence approach. In H. Lin, editor, Architectural Design of Multi-Agent Systems. Information Science Reference, 2007

[4] Fabiana Lorenzi, Fabio Arreguy Camargo Correa, Ana L. C. Bazzan, Mara Abel, Francesco Ricci, “A Multiagent Recommender System with Task-Based Agent Specialization”, 2008

[5] Fabiana Lorenzi, Ana L. C. Bazzana, Mara Abel, Francesco Ricci, “Improving recommendations through an assumption-based multiagent approach: An application in the tourism domain”, 2011

[6] https://www.w3schools.in/restful-web-services/intro

[7]https://ahmetozlu.medium.com/mastering-rest-architecture-rest-architecture-details-e47ec659f6bc

[8]https://www.javatpoint.com/restful-web-services-spring-boot#:~:text=Advantages%20of%20RESTful%20web%20services%20RESTful%20web%20services,data%20format%20like%20JSON%2C%20text%2C%20HTML%2C%20and%20XML.

[9] https://developer.americanexpress.com/open-banking

[10] https://gabrymartinez.medium.com/how-to-create-your-own-little-restful-web-api-and-not-get-lost-in-the-process-part-2-473400256ce0

[11] https://www.robocup.org/

[12] Frans C.A. Groen, Matthijs T.J. Spaan, Jelle R. Kok, and Gregor Pavlin, “Real world multi-agent systems: information sharing, coordination and planning”, 2007

[13] Mihaela Oprea, “Applications of multi-agent systems”, 2004

[14] Florin Leon, “ActressMAS, a .NET Multi-Agent Framework Inspired by the Actor Model”, 2022

[15] Luke, S.; Balan, G.C.; Panait, L.; Cioffi-Revilla, C.; Paus, S. MASON: A Java Multi-Agent Simulation Library. In Proceedings of the Agent 2003 Conference on Challenges in Social Simulation, 2003.

[16] Bellifemine, F.L.; Caire, G.; Greenwood, D. Developing Multi-Agent Systems with JADE, 2007

[17] Bernstein, P.; Bykov, S.; Geller, A.; Kliot, G.; Thelin, J. Orleans: Distributed Virtual Actors for Programmability and Scalability; 2014.

[18] https://opentripmap.com/en/#0/

[19] https://opentripmap.io/product

[20] https://www.geeksforgeeks.org/advantages-and-disadvantages-of-sql/

[21] https://www.educba.com/what-is-postgresql/

[22] Robert C. Martin , “Design Principles and Design Patterns”, 2000, available online at http://staff.cs.utu.fi/~jounsmed/doos\_06/material/DesignPrinciplesAndPatterns.pdf

[23] https://www.eternalsoftsolutions.com/blog/how-to-design-aws-architecture-for-web-application-with-high-availability/

[24] https://www.flightapi.io/

[25] https://developers.booking.com/api/index.html

[26] https://www.cheapflights.com/book-flights-online/

[27] http://insideairbnb.com/get-the-data/

[28] https://www.extendoffice.com/product/kutools-for-excel.html

[29] https://airportcodes.io/en/continent/europe/

[30] https://joshclose.github.io/CsvHelper/

[31] https://dotnet.microsoft.com/en-us/apps/machinelearning-ai/ml-dotnet

[32]https://learn.microsoft.com/en-us/dotnet/api/microsoft.ml.trainers.matrixfactorizationtrainer?view=ml-dotnet-preview

[33] https://colorlib.com/wp/template/travelix/

[34] https://codepen.io/namratapdr/pen/yLOgREo

1. A recommender system, or a recommendation system is a subclass of information filtering system that provide suggestions for items that are most pertinent to a particular user. See Francesco Ricci, Lior Rokach & Bracha Shapira, “Recommender Systems: Techniques, Applications, and Challenges”, 2021. [↑](#footnote-ref-2)
2. American Express (AXP) is an American financial and travel services corporation with operations in more than 110 countries. To see <https://www.investopedia.com/articles/investing/032716/top-5-companies-owned-american-express-jblu-flws.asp#:~:text=American%20Express%20%28AXP%29%20is%20an%20American%20financial%20and,financial%20services%2C%20and%20travel-related%20services%20like%20traveler%27s%20checks>. Visited on 21.09.2022 [↑](#footnote-ref-3)
3. NuGet is the package manager for .NET. The NuGet client tools provide the ability to produce and consume packages. The NuGet Gallery is the central package repository used by all package authors and consumers. To see https://www.nuget.org/. Visited on 09.24.2022. [↑](#footnote-ref-4)
4. Singleton class allows to allocate and instantiate data single time. [↑](#footnote-ref-5)