CI615 Report

By Jake S. T. Ward

23/12/2022

## Section 1: Potential Design Patterns

I first identified five potential design patterns that could be used for the web portal. These designs were documented in UML and are presented in the screenshots below alongside the consequences of their use and their applicability.

### The Builder Pattern

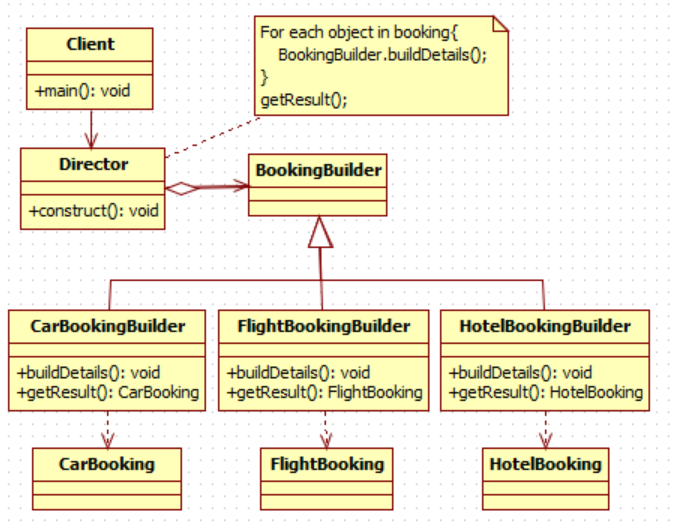
## This pattern could be used for creating bookings.

The Builder Pattern is a creational pattern that could be used for the design of the booking system that would be used for creating potential bookings. The Builder pattern creates complex objects separately from their representations. This way the same construction process can create multiple different representations with high degrees of control.

This pattern is applicable if the system should be building many objects from the same input. This could be applied as an array and could be used with the booking builder or director class to create different objects using its subclasses. The pattern separates what is used to build objects from how they are built. It is a simple pattern to implement into a pre-existing system so long as the builder pattern has the same external behaviour as whatever it is replacing. The builder pattern is used to reinforce polymorphism, which allows the reuse of code making the system more efficient.

The consequences of using this pattern for the design of the system are as follows:

* The use of the pattern isolates both the code for object construction and their representation
* A given object’s internal construction and internal representation can vary
* There is a large amount of control given when using this pattern.
* The director-class would be responsible for the structure of all other classes while the builder classes would be used to build individual components.



*Figure 1: Builder pattern example*

### The Façade pattern

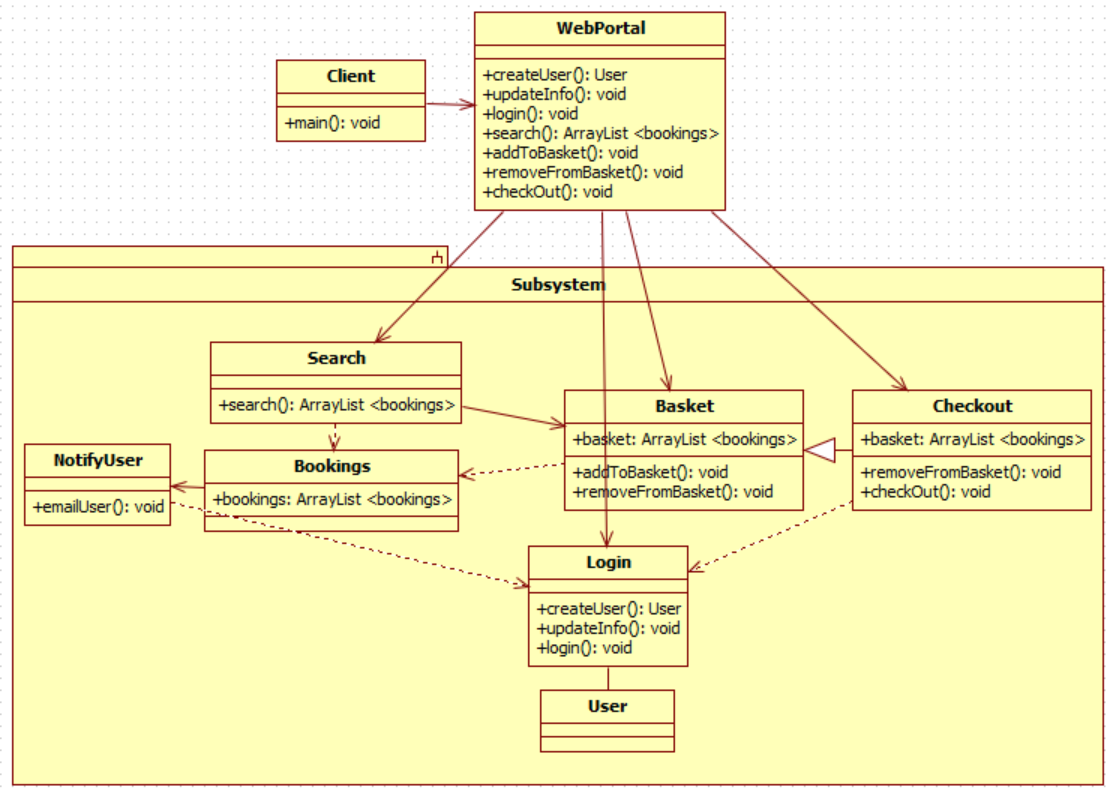
This pattern could easily apply to the login system.

The Façade pattern is a structural pattern used to provide an interface to be used with a set of other interfaces in a sub-system to hide complexity. When this pattern is used a façade object provides a singular simplified interface representing functions of other sub-system interfaces. It communicates with the other interfaces to minimize any communication between the other subsystems. The Façade object must know which sub-systems are responsible for requests and delegate them accordingly. The sub-systems carry out the functionality and the Façade acts as a controller that the sub-systems are unaware of.

The Façade Pattern is utilized when developers want to provide a simple interface for a complex sub-system. It can also be used if developers wish to add layered sub-systems or if there are an overwhelming number of dependencies between classes and clients.

The consequences of the patterns use are as follows:

* There is often a lack of coupling between subsystem components.
* Subsystem classes are still able to be accessed outside of the use of a Façade object, which is positive or negative depending on the situation. In this scenario, I see no negative effect with this consequence in the system.
* Clients are disconnected from subsystem components as clients will often only interact with Façade objects, which simplifies functionality.



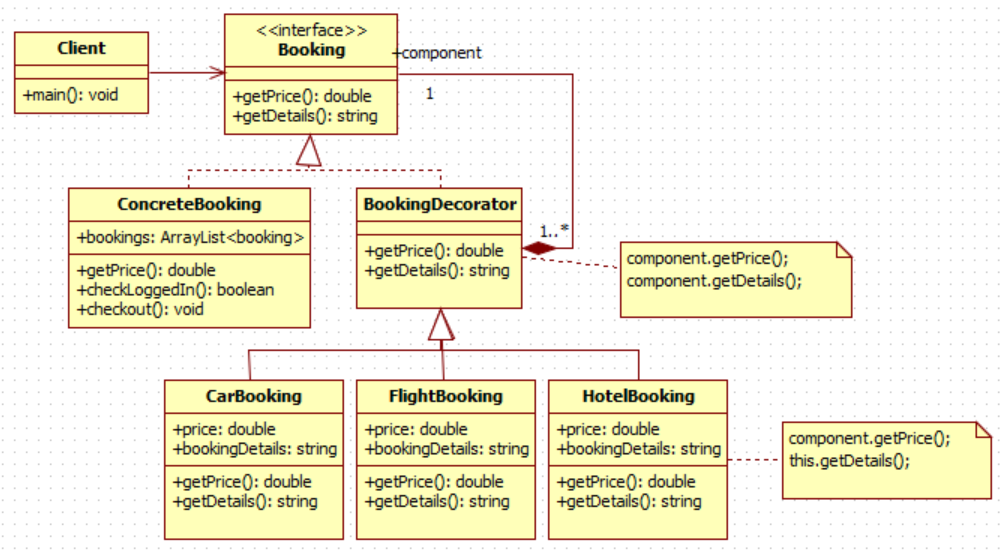
*Figure 2: Façade pattern example*

The Decorator pattern

The structural pattern known as the Decorator pattern is used to attach additional elements to an object dynamically. The pattern’s use provides a substitute for subclassing that allows greater flexibility and the extension of functionality via composition. While this pattern is commonly used with the addition of GUI elements to a window it is not exclusive to this. By using the Decorator Pattern for booking we can prevent class explosion when handling the different combinations of bookings.

For a Decorator pattern to be applicable the system must either need dynamic addition of responsibilities to objects, or extension via subclassing must be impractical. In a situation where many independent extensions are possible, subclassing would be impractical because it would cause a class explosion. In the same way, a class definition being unavailable for whatever reason would make subclassing equally unavailable.

A consequence of the Decorator Pattern’s use is that it is wildly more flexible than inheritance. Decorators can be nested within other Decorators, and all can be removed or added at runtime without error. However, the decorator pattern can develop a fragmented system and make code hard to analyze.

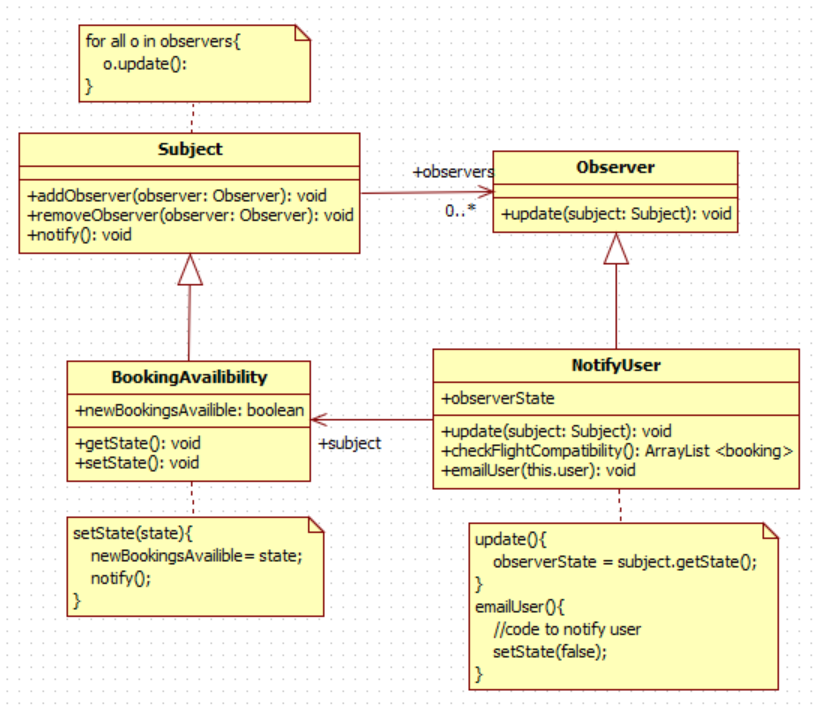
*Figure 3: Decorator Pattern example*

### The Observer pattern

This pattern could be used for sending emails based on profile changes or booking additions.

The behavioural pattern known as the Observer pattern is used to define one-to-many dependencies between objects so the change of an object will affect the other objects. Objects are notified of changes to one another automatically and are changed to correspond with the objects' new state.

A consequence of the Observer pattern is that it supports abstract coupling between Objects and Observers. This means that Objects are aware of their list of observers, all of which are interfaces of the observer class. This abstract coupling may cause indirect notifications which may have an inefficient effect. The Observer pattern doesn’t require observers to query objects for up-to-date information.

*Figure 4: Observer pattern example*

### The Abstract Factory Pattern

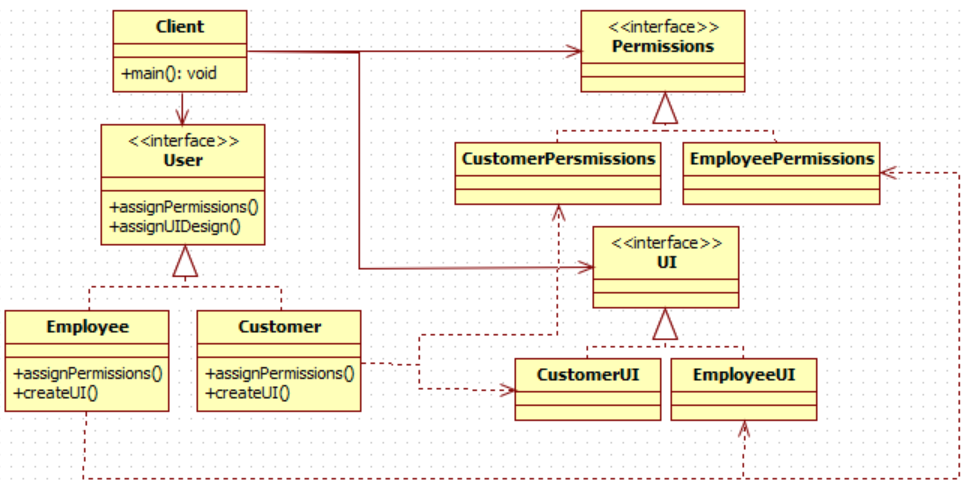
This method could be used to build different UIs and other elements depending on the type of user.

The Abstract Factory design pattern is a creational pattern that could potentially be used for the design of this system. The pattern provides interfaces for creating families of related objects. It differs from the Factory pattern as there are no concrete classes. A Factory pattern uses inheritance and subclasses to handle object instantiation while an Abstract Factory delegates object instantiation to other objects through composition.

The Abstract Factory pattern is useful for situations where a class cannot anticipate the class of objects it must create. It is also applicable when a system should be independent of an object’s creation or needs to use a set of families of objects. Finally, it can be applied in a situation where a family of related objects are designed to be used together and should be constrained to function this way.

The consequences of using this pattern for the system are as follows:

* The client is beneficially isolated from concrete class implementations.
* It enforces the use of products from only one family. This can be beneficial for the system for separating objects designed for customers and objects designed for employees.
* It makes the exchange of product families simple because a concrete factory can support a complete family of objects. However, the support of any new kinds of products requires changes to the Abstract Factory interface (in this case the User interface).

*Figure 5: Abstract Factory pattern example*

Section 2: Architectural Design

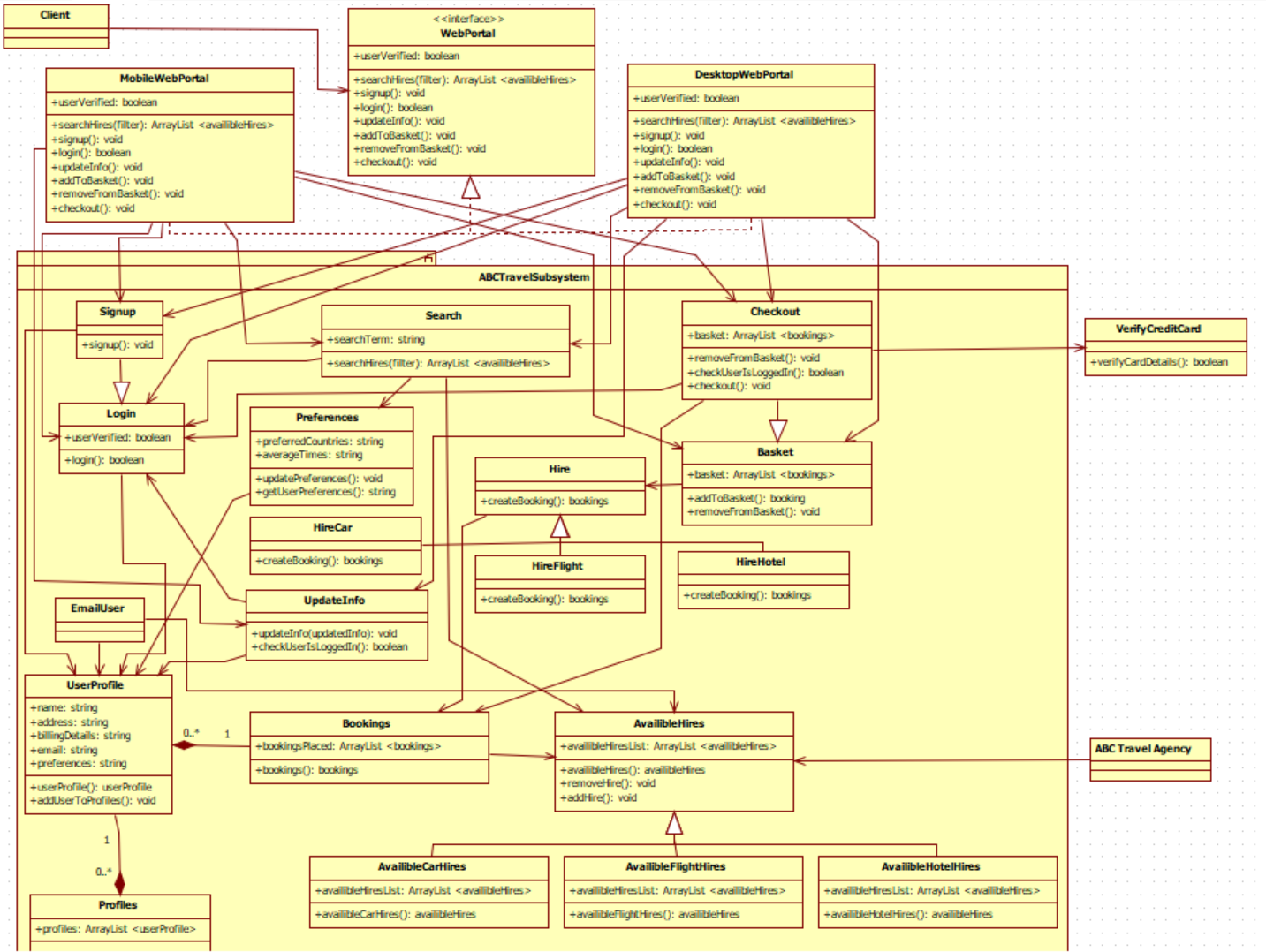
This section documents the design of the described ABC travel agency web portal and system. Architectural elements were identified in the system and various UML diagrams were created to lay out these elements and their relationships. The interfaces were also identified along with their relations and dependencies as well.

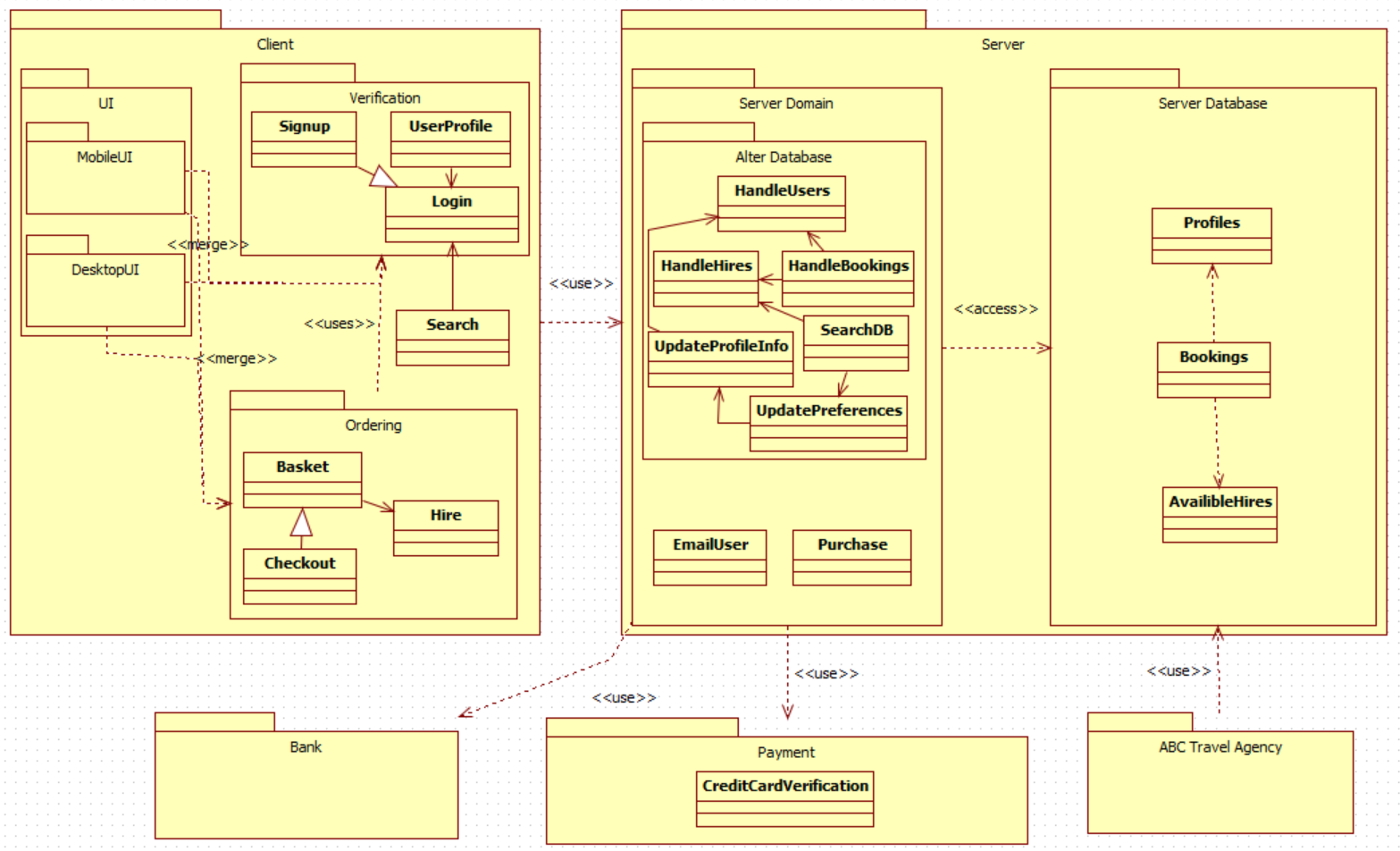
Below are the various use cases in a table used to identify key features of the system to be designed.

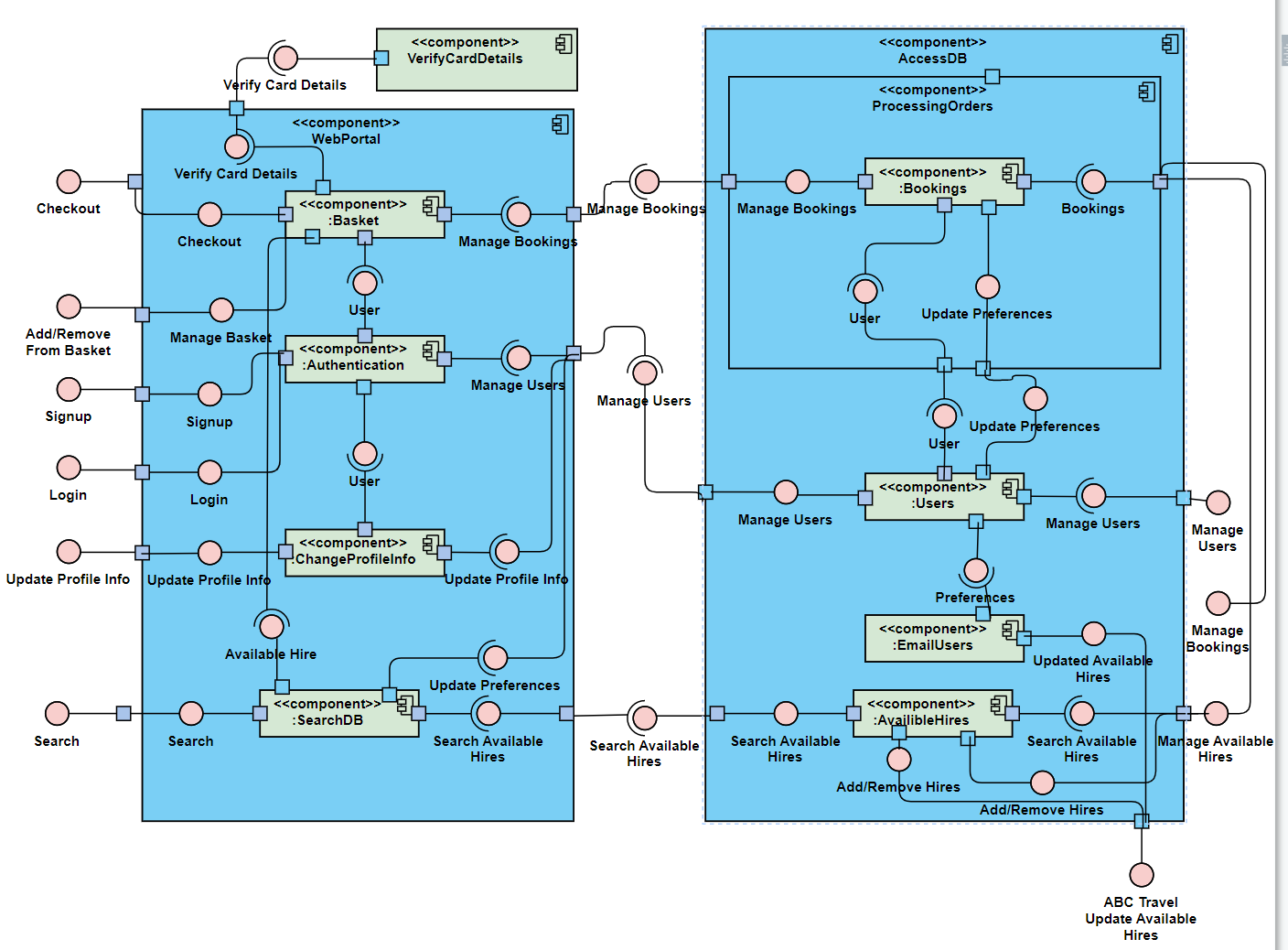
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Use Case Name** | **Use Case Overview** | **Actors** | **Pre-conditions** | **Stimulus** | **Basic Path** | **Post-conditions** | **Exception Path** |
| Open Web Portal | Allow users to use the Web Portal from multiple platforms including Desktop, IOS mobile and Android mobile devices. | Customer, Web Portal | Website is running on a server and the customer has internet access. | The user navigates to the DNS URL | 1. User navigates to DNS URL.  2. An IP address is found and navigated to | 1. Home/Index page is displayed | If an invalid URL is entered the User fails to navigate to site |
| Search for flights, hotels, or car hire | The customer can search for flights, hotels, or car hiring facilities individually or together (e.g., search for flight+hotel or hotel+car hiring). The customer has the option to log in or not when searching. | Customer, Web Portal, Search System, Database | The customer has access to the web portal. | The customer initiates a search on the web portal. | 1. The customer selects the type of search they want to perform (e.g., flights, hotels, car hire, or a combination). 2. The system displays the available options based on the search criteria.  3. The customer can browse the options and select the ones they are interested in. | 1. The search results are displayed to the customer. 2. The customer can choose to purchase the products they are interested in or continue browsing.  3. If logged in the customer's preferences are mildly updated according to the search filters | If the search results are not available, the system displays an error message. |
| Register on the Web Portal | The customer must register on the web portal by providing their name, address, billing details, email, and any preferences they have for travel (e.g., preferred countries, cities, or times of travel). They would set a password so they may log in later. | Customer, Web Portal, Login/Signup System, Database | The customer has access to the web portal. | The customer initiates the registration process on the web portal. | 1. The customer provides the required information (e.g., name, address, billing details, email, preferences and password).  2. The system verifies the provided information. 3. The system creates a profile for the customer. | 1. The customer's profile is created and saved in the system.  2. The customer is registered on the web portal. | If the provided information is invalid or incomplete, the system displays an error message. |
| Login on the Web Portal | The customer can log in on the web portal by providing their username/ email and password. | Customer, Web Portal, Login/Signup System, Database | The customer has access to the web portal and has a pre-existing profile/ account. | The customer initiates the login process on the web portal. | 1. The customer provides the email and password associated with their account.  2. The system verifies the provided information. | 1. The customer's profile is verified in the system.  2. The customer is authenticated on the web portal | If the provided information is invalid or incomplete, the system displays an error message. |
| Purchase from a Hiring Company/ Make Bookings | The customer can make bookings using their credit card. The credit card system, which is an external system, will verify the credit card details and debit the customer's account. | Customer, Web Portal, Search System, Basket/Checkout System, External Credit Card System, Database | The customer has access to the web portal and Is authenticated.  The customer has searched for hires. | The customer adds one or more hires to their basket on the web portal. | 1. The customer adds hires to their basket.  2. The customer initiates checkout.  3. The system confirms the customer is authenticated.4. The user inputs their payment info  5. The external credit card system verifies the payment information's validity. | 1. The customer's profile information and the selected hires are combined to make bookings.  2. The bookings are inserted into the database.  3. The hires are removed from the database as they are now booked.  4. The user's preferences are updated in accordance with the bookings | If the customer is not logged in, then they are redirected to the login/signup page. If the card details are not valid then the checkout process is cancelled, and the system responds with an error message. |
| Update profile information | The customer can update their profile information (e.g., name, address, billing details, email, preferences) at any time when they are logged in. | Customer, Web Portal, Database | The customer is registered on the web portal and is logged in. | The customer initiates the update of their profile information. | 1. The customer accesses their profile on the web portal.  2. The customer modifies the information they want to update.  3. The system verifies the updated information. 4. The system updates the customer's profile. | 1. The customer's profile is updated with the new information. 2. The updated information is saved in the system. | If the updated information is invalid or incomplete, the system displays an error message. If the customer is not logged in, then they are redirected to the login/signup page. |
| Receive personalized offers via email | If there is a flight, hotel, or car hiring offer that is compatible with a customer's profile, the system will send an email to the customer with the details. | Customer, Database, Database Observer | The customer is registered on the web portal and has provided their email address in their profile. The ABC Travel Agency has added new hires to the database. | The system detects a compatible offer for the customer. | 1. The system retrieves the customer's email address from their profile.  2. The system sends an email to compatible customers with the details of the offer. | 1. The customer receives an email with the details of the offer.  2. The customer can choose to take advantage of the offer or ignore it. | If the system is unable to retrieve the customer's email address, the email is not sent. |
| Manage list of flights, hotels, and car hire companies | The ABC travel agency can add or remove flights, car or hotel hires. | ABC Travel Agency, Database | An administrator has access to the database. | The travel agency has new hires to add to the database or needs to manually remove hires. | 1. The travel agency employee can add several new hires through a client separate from the public web portal.  2. The travel agency employee can remove several hires through a client separate from the public web portal. | 1. The database adds and removes the hires specified.  2. Any bookings containing the cancelled hires will be removed.  3. The customers with removed bookings are notified via email and refunded. | If the system is unable to identify the administrator no changes will be made, and the attempt will be flagged as attempted unauthorized access. |

Below are examples of Quality Attribute Scenarios for the system overall. This includes the use of REST APIs, Multitier client-server architecture and the possibility of the usage of the messaging pattern.

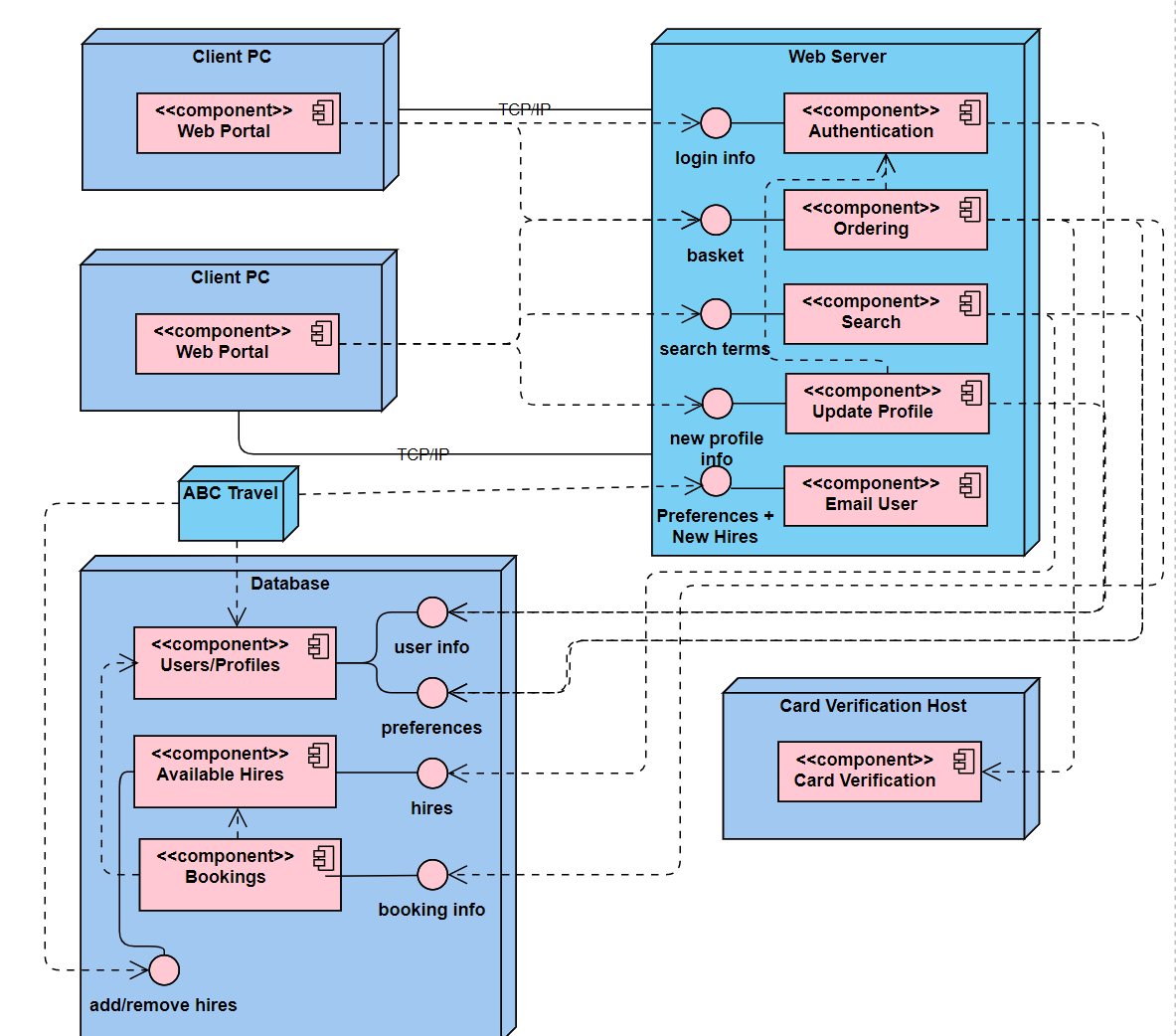
|  |  |  |
| --- | --- | --- |
| Quality Attribute | Stimulus | Response |
| Security | No requests have been received from a user session for an unnatural amount of time. | The session is flagged as insecure, and the security credentials provided to it are invalidated. The user will now have to re-authenticate to resume. |
| Scalability/Performance | Servers are frequently reaching capacity, which affects performance. | The systems servers and queues are replicated and used to scale outward for the increased demand. |
| Modifiability | The internal logic of the business tier needs replacing. | The associated layer can be changed without issue so long as the input and output formats are compatible with the other layers. |
| Performance/Security | Extra security can be added to the system to protect users' sensitive data, but it will negatively affect runtime. | Extra security would be added as security should take priority. |
| Availability/Failure Handling | The connection between the queue and business tier drops. | Any requests/messages are stored on the MOM server or REST API until a connection is reestablished unless the server itself has failed. |
| Failure Handling | The central server fails. | A replication of the server replaces the original until the original central server is online again. |
| Security | The central server is unlawfully accessed. | A potential area of risk due to centralized data. This can be remedied by splitting data amongst multiple servers to reduce the impact or the system can rely on encryption to keep data safe. The latter is preferable as only one location would need to be heavily secured and the advantages of centralised data could remain. |

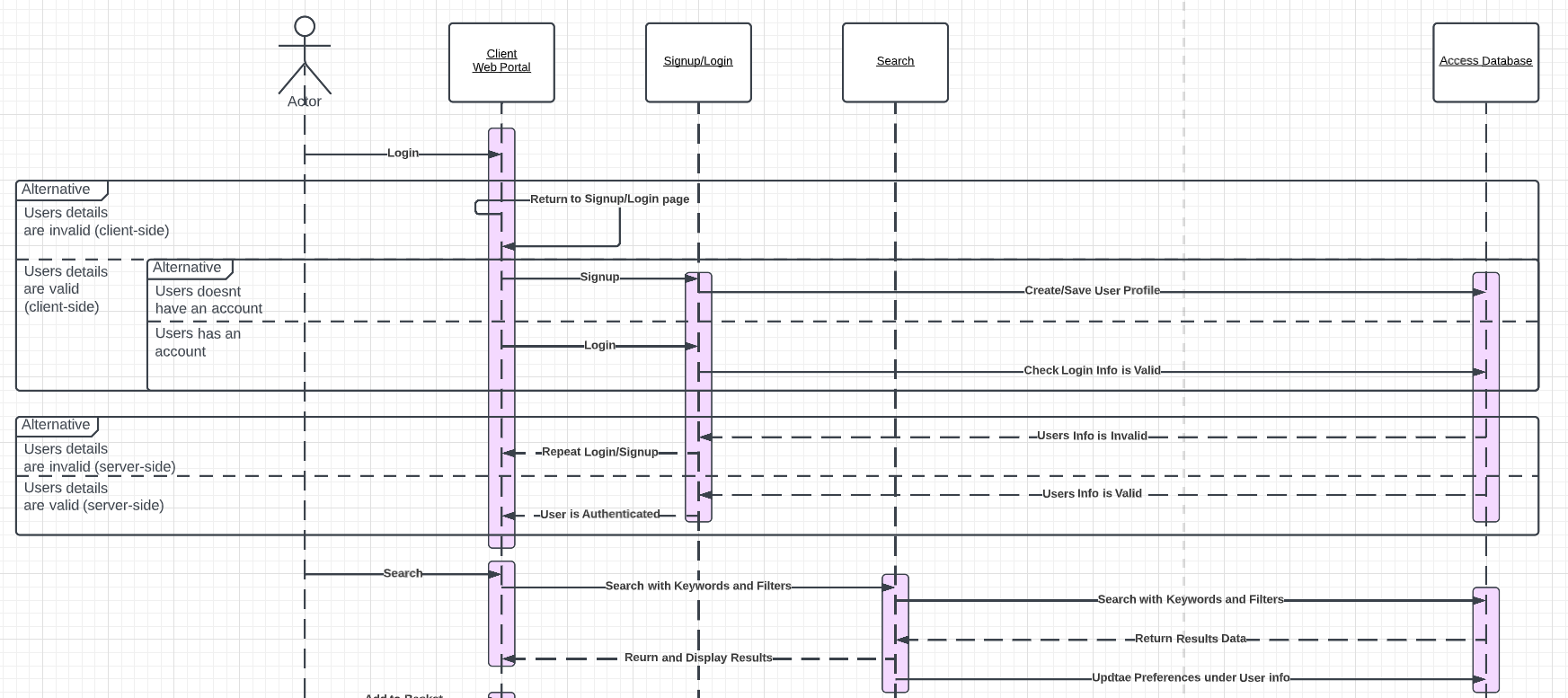
*Figure 6: Class diagram*

*Figure 7: Package diagram*

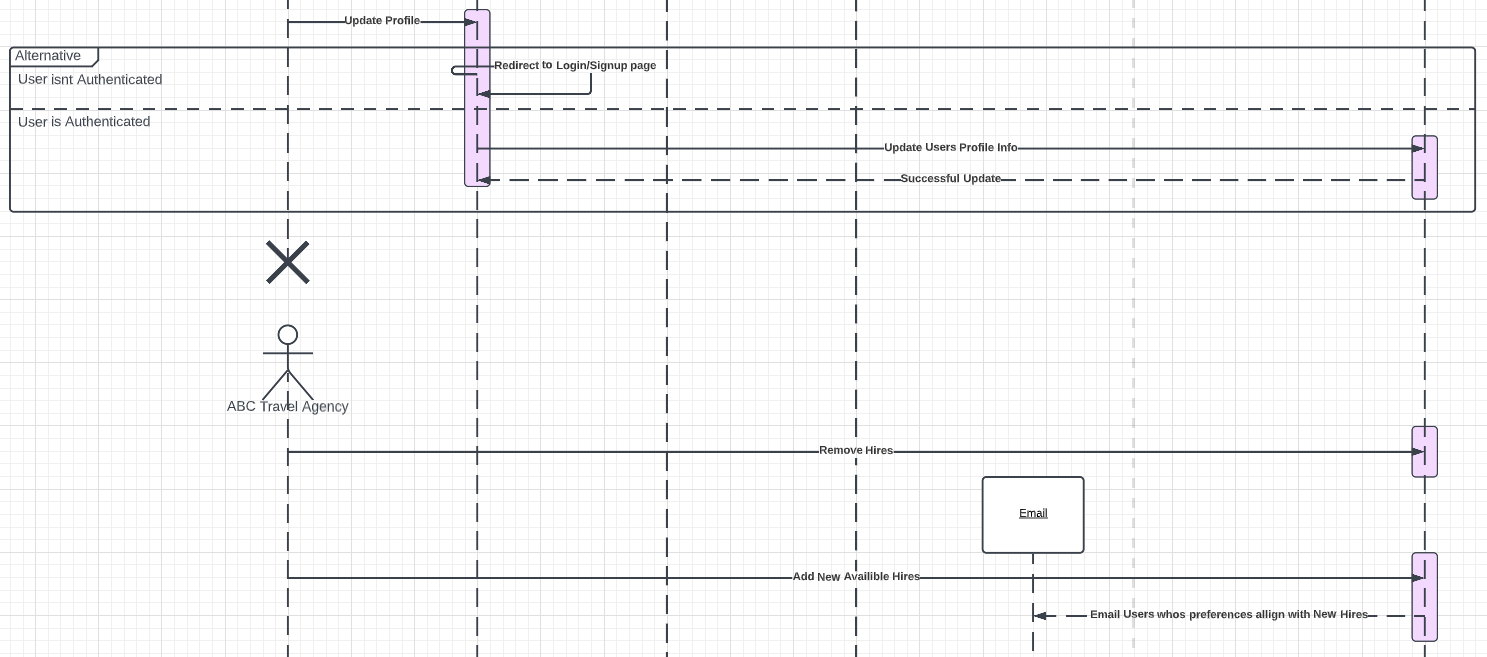


*Figure 8: Component diagram*

*Figure 9: Deployment diagram*







*Figures 10-12: Sequence/communication diagram*

### Quality Attributes

The best way to determine what architecture should be used for building the system and how to implement it is by analyzing quality attributes. Quality attributes are testable/measurable properties that indicate the efficiency and effectiveness of a system. The main quality attributes to analyze in this type of systems implementation and design are:

* Security
* Scalability
* Modifiability
* Performance
* Availability.

These attributes can apply both globally or locally and will only make sense when the context is provided. It must be acknowledged that some attributes conflict with one another, such as how extra security measures can affect performance and vice versa. Due to this certain quality attributes should be prioritized. Quality attribute scenarios have been displayed above in the use case table and quality scenarios table

In this type of system, security must be higher in priority than the other attributes even if it negatively affects them. This is because this system is designed to handle sensitive confidential information which could be targeted for theft, such as emails, passwords, payment information etc.

The modifiability should not be of especially high priority but should be considered in the design. Allowing the system to be adjusted cost efficiently will always be a positive addition and often will not negatively affect any other quality attributes.

Equally high in priority should be the availability of the system. When a web application is being created for the public, the hours it will be used are unpredictable, so the system should be online and functional. Downtime must be reduced at all costs and a cost-effective solution would have to be put in place to run the server(s) for extended periods of time.

Ideally, the system's performance should still be a priority to avoid sporadic load times and bad latency. However, without acknowledging specific hardware, locations of servers and software design, this quality attribute is unpredictable until the system is actually being implemented.

Usability should also be a factor considered when designing the clients as the system must be easy to understand but more importantly easy to use.

### Client-Server Architecture

I chose the most widely-used architecture for constructing the requested system, that being the client-server model, due to its conventional usage reinforcing its many advantages. By using this architectural structure, a system inherently has a higher level of security, easy maintenance and centralized data access due to the centralized server. However, the use of this type of architecture means that it is dependent on a centralized server. To remedy this there can be multiple replications of the server set to run in place of the central server should something go wrong. If only a singular server/device is used in a client-server model, then the scalability and modifiability of the system can suffer due to the business tier and data management tier being interwoven out of necessity.

While the consideration of a peer-to-peer network would reduce the security risks of having sensitive information centralized it would not be fit for purpose as the web portal is meant for the public and not designed as an internal system for employees.

### Multitier Client-Server Architecture

The client-server architecture can be applied to the ABC travel agency system but there are different sub-architectures within this model. While 1-Tier or 2-Tier architecture would be sufficient, multitier architecture (or N-tier) would be best suited due to its scalability and adaptability so long as communication formats between layers do not change. This means that over N number of layers the presentation tier, business tier and data management tiers' responsibilities are managed. In this case, the architecture is best to be split among three layers, one for each tier. This model supports synchronous communication, meaning each layer waits for a response from the proceeding layer. The aforementioned adaptability with multilayer architecture is due to its flexible deployment. This means that all tiers can be hosted across one or multiple devices.

By having the storage of data and handling of data managed by singularly dedicated layers (3-Tier architecture) it is easier to guarantee the integrity of the stored data as well as allow layers to easily be swapped with new versions without error.

There are many other quality attributes of a multitier client-server model.

Scalability is often high with this model. The servers in each tier can easily be replicated and run across multiple servers or one device. The architecture can easily be scaled outward and upward. However, the database will often act as a bottleneck for system capacity when considering scalability.

Availability is another quality attribute. Due to the easy replication of tiers, if one server fails to provide the responsibilities of the tier, another one can take its place. The only consequence of this is that the service provided by the affected layer would be of lower quality until the failed server/layer upheld its responsibilities again.

Modifiability is a quality attribute of the N-tier model because of the separation of responsibilities. The separation of the presentation tier, business tier and data management tiers ensures encapsulation. Meaning each can have its internal logic modified without affecting other tiers so long as the outgoing and incoming formats are still compatible.

Performance is generally high with this architectural model. Considerations that need to be made for the performance are the speed of connections between the various tiers, the number of concurrent threads supported on a server and the amount of data being transferred between tiers. To improve performance the calls between tiers should be minimized.

Failure Handling is optimal with the client-server model because if communication fails between tiers, then another replication of the server can take its place. Most web applications making use of this model also utilize this process known as transparent failover.

Overall, the client-server model with a multitier structure would be sufficient for the architecture of the described system because of its high positive reinforcement of Security, Scalability, Modifiability, Performance and Availability. Quality attribute scenarios for this design have been incorporated into the use case table displayed above in the second section of the report.

### Messaging Pattern

While the N-tier model would be chosen for the overall structure of the described system, complex architectures require the creative blending of multiple patterns. So, as well as the N-tier architecture being chosen and all previously listed design patterns being implemented, I would also consider making use of the messaging pattern/style as N-tier and messaging architectures are not mutually exclusive.

By blending the messaging pattern with the system it would allow for asynchronous communication as clients would be sending requests to a queue instead of directly to the server. It would also have the advantage of loose coupling between tiers as there would be no need for direct binding between the different tiers, as long as they know what format to send/receive data. By implementing this pattern with N-tier architecture I would also be able to reinforce scalability and modifiability/adaptability as the queue could be configured and reconfigured for slow-speed reliable connections or high-speed less reliable connections.

The quality attributes to be considered when applying this pattern in the system include:

Availability. As queues can be replicated and placed across multiple server instances; if a queue were to fail, the users could have their requests redirected to a replicant queue. Thus, the availability of the system would be much more likely to be maintained when combined with the possibility of multiple server adoptions seen in the N-tier client-server model.

Scalability can be mentioned as queues can have replications across many servers hosted across multiple devices. Due to this, queues can be scaled up or down with ease just like the servers in the N-tier architectural style.

Performance. Messaging queues’ inclusions generally allow for thousands of messages to be handled per second. Reliable messaging will always be a slower solution and vice versa, this will mostly be based on the specific messaging technologies used.

Failure Handling is simple with this pattern's implementation as previously mentioned. If a queue fails then the users' messages/requests can be redirected to a replica queue.

Modifiability In terms of Modifiabilty the messaging pattern excels. Its inclusion ensures an inherent loose coupling of layers and often promotes a high degree of modifiability. This is because both servers and clients are not bound through an interface. Message formats will need to remain the same for the system to function unless self-describing and discoverable message formats are used.

### REST APIs

A REST API would be the chosen approach for implementing the business tier of the system. The use of a REST API (while less flexible due to the client requiring the URI of the service) would provide a much simpler approach and reinforce the advantages of loose coupling. Unlike the SOAP approach, no special languages or formats would be needed.

Validation

Validation of these patterns/styles can be done in two ways:

1. manual testing of the architectural designs using quality attribute scenarios
2. the construction of prototypes that create an archetype of the real system.

Both should ideally be used to fix flaws in the design stage so that when creating the application there are fewer errors to fix and overall implementation becomes a cheaper process. I used a use-case table as well as a quality attribute scenarios table for this purpose so that they would coincide with the UML diagrams. Both serve the same purpose and prevent errors in the design stage and both can be found at the start of the section.

Quality attribute scenarios involve defining stimuli that would often negatively impact the architecture and displaying how the architecture would react to this scenario. If the response is desirable then it is labelled a success while if not or is difficult to define, then it is labelled as an area of risk.

Since scenarios cannot accurately account for all possible errors, prototyping should also be used. By building prototypes to test the architecture, the other design issues can realistically be identified and hopefully remedied or defined as areas of risk.

### Effect of Architectural Styles on Overall Architecture

In utilizing these architectural styles, there are many consequences both positive and negative. Below I have listed the Architectural styles used in the actual system designed and explained the reason for choosing these styles.

The use of a client-server structure means that the system has a high level of security helping to keep users' customers' sensitive information safe. This is because the different tiers are not closely intertwined and simply communicate with each other, helping to reduce possible access to the data management tiers. It also means that the customer's data as well as bookings and hires information would be centralized. This would significantly reduce data redundancy, which would minimise errors and simplify access to information. The central server could easily support quick maintenance and because the different tiers would be separated the server could be replicated and replaced until the maintenance of the original server had taken place.

By using the N-tier/Multitier architectural style the ABC travel agency would have great modifiability and could add new layers for increased functionality if they needed to at a later date. By separating the system into at least three layers (one for the presentation tier, one for the business tier and one for the data management tier), the integrity of the stored data is easier to guarantee and the various layers can be replaced for maintenance or updates without causing errors. Equally, because of the separation of the tiers via servers or physical devices, in this case, the scalability is very high as the tiers can easily be replicated and used to scale both upwards and outwards. This means even if the travel agency was experiencing higher demand, as long as they could afford more servers or space, then the system can expand to account for the higher demand. The centralized database, while still holding its advantages, would act as the bottleneck for request capacity.

The fact that the chosen multitier architecture allows for the replication of tiers if necessary, means if a layer like the database access layer were to fail, then a replica could take its place temporarily ensuring availability. If the entire server were to go down then the same process could take place on a larger scale. The architecture chosen could involve the Database and API being stored on a different device or the same device but to increase availability, security and modifiability it would be best to store each layer on a different device. This can also be considered optimal failure handling.

While performance is lower in priority than security and the two quality attributes often conflict, the use of an N-tier client-server model often supports high performance with modern technologies. By keeping the number of calls between layers at a minimum the performance is increased. This can be seen by updating the preferences alongside the user's search, booking or information update.

Hires, users and bookings are how the database/ data management tier would be divided. The hires would consist of the potential bookings to be listed on the web portal. The users would consist of the user's profile information as well as preferences such as name, address, billing details, email, and some preferences such as preferred countries, cities, or the usual times that he/she makes trips. Finally, the bookings would serve as a confirmation of payment from the web portal and would consist of a combination of information from the users and hires tables. This allows the system to run efficiently due to its simplicity while also providing the users with their proof of purchase and allowing orders to easily be identified and cancelled after they have been placed if necessary. By placing the preferences of users along with their other information the system is more efficient as it only needs to query the database tier once.

Clients would make use of the web portal as the presentation layer which would allow users to seemingly create profiles, log in, search for hires and make bookings. The decorator pattern could be used as the internal logic for building an order before checkout and also for searching with filters to avoid the need for a class for every combination. An abstract factory pattern could be used to build different web portals based on who was trying to use it or what type of device or browser. This would allow great compatibility as well as an easy solution for differentiating employees from customers. Because the web portal isn't actually carrying out all tasks, it is acting as a façade. The façade pattern was chosen for use because it hides the complexity of the system from users making it easier to use. This also provides greater security as the code is harder to reach and analyse seeing as how only requests to the other tiers would be viewable. The bookings themselves, while not part of an order, could be created using the builder design pattern because it reinforces polymorphism and could handle the creation of the varying types of bookings with ease.

A REST API on a separate web server would be responsible for the business tier because the separation of tiers and APIs independence increases security while not dramatically reducing performance. REST would be used specifically because it is a lightweight method due to its reliance on the HTTP standard. It also allows for great flexibility and scalability with little to no limitations. This API would handle most processing of information such as the basket and checkout processes and signup and login processes. It is easy to understand and design meaning less cost for the travel agency should maintenance or modification need to take place. It would also be able to handle automatic processes that shouldn’t be reliant on clients such as the use of an observer pattern to update users with personalized emails on new hires that correspond with their preferences. While only a singular API/web server has been shown in the UML diagrams to demonstrate the normal implementation of this style, the possibility would remain open to adding new REST APIs or Web servers. This is because they could still communicate with each other and would separate processes further allowing for easier modifying of code, without affecting other layers as well as increasing security through the separation of processes.

When using any kind of architectural model that involves the use of a data management tier it is conventional to use a database for that purpose. It was used in this architecture not just because of its conventionality but because they are specifically designed for the purpose of handling data securely and efficiently. The database would centralise all data to allow for faster access but also allow the securing of data to be an efficient process in terms of both cost and processing power. It also allows the security of the system to be targeted at one element of the system as opposed to having securities attention split among a cluster of servers. This would increase the security of sensitive data, avoiding any unauthorised access.

While the Messaging pattern was not directly incorporated, if the system were in the development stages then elements of this architectural pattern would be expected to be blended with the N-tier architecture. By utilising queues or by similarly utilising the API, then asynchronous communication can be incorporated along with loose coupling. This increases security as there is no need for direct binding between elements/layers. Scalability and modifiability could also be reinforced through the reconfiguration of queues, either favouring reliability or fast speeds depending on the situation. It would also redirect requests that have reached a failed queue to a functional queue to uphold availability.

## Section 3: Reflection

This section is used to provide an argument as to how the provided architectural design could be implemented. This is described in terms of technology as opposed to theoretical design structures.

### Service Oriented Architecture (Alternate Implementation Solution)

Seeing as how services are mechanisms to enable access to many capabilities I thought it best to incorporate elements of service-oriented architecture into the system or consider the approach applicable for an alternate method of implementation. Due to SOA (Service Oriented Architecture) being a sub-sect of the client-server model, it has the advantage of loose coupling. It can be used to incorporate software components from dissimilar platforms and can reincorporate components in unexpected ways upon development. It is useful in outsourcing development by breaking systems down and can aid in leveraging latency investment.

In order for this type of implementation of architecture to be effective, the service interface would need to articulate the required input parameters, any known possible errors or exceptions, the operations they perform and the response details. It would need to implement/be based on endpoints, policies and services. It would need to have explicit boundaries, autonomous services and service compatibility based on policy and share schemas as opposed to utilising implementations. The use of schemas helps separate the system as the only dependencies are the service messages they exchange.

The implementation technologies that could be used to implement this system as a service-orientated architecture are any application integration technology, such as CORBA, DCOM, JEE or XML standard Web Services. UDDI would be used to find appropriate services, WSDL would be used to find information about services and SOAP would be used to request a service to take action.

Overall, this alternate implementation of the system reinforces open collaboration, flexible/dynamic changes and cost-efficient methods. This solution would reinforce loosely coupled reusable software over internet standard protocols.

Implementation

The Client Web Portal could be implemented in any number of ways. Seeing as how the web portal would be accessed through a browser it would have to use JavaScript, CSS and HTML code to be easily publically accessible. While the web application could be written in Java this would only realistically be useful if the system were to only be used locally or within a small work environment which is not what the requested system is. Progressive web application elements such as service workers and manifests could be used to allow the application to be installed as an application as well as queue requests like the messaging pattern when offline. By designing the web portal with these elements as a single-page application the web portal could retain an amount of functionality while offline, such as exploring the results of previous searches or viewing the bookings the users have made. I have previously created an online marketplace which supported searching through database results, a login system and purchases through the use of these languages for the front end, so am confident that it could be implemented in this fashion.

Frameworks could be used so that the development of the client web portal doesn’t need to begin from scratch. Many front-end frameworks allow for the fast production of these types of UIs. The React framework evidently functions for this purpose due to its frequent involvement in the development of Meta-owned web applications such as Instagram and Facebook. It is frequently updated and is a component-based library which doesn’t inhibit any functionality of the base languages of web applications, meaning that the API could easily be accessed. React could easily be combined with other Javascript libraries to support additional functionality and many tutorials exist on how to create search systems, login/signup systems and ordering systems, proving that it is possible to use the framework to implement this kind of system in this way. The React framework/library also supports easy migration between versions which would support high modifiability. It is also an easy framework to learn, meaning that it would be an easy framework to implement the web portals front end in. It has virtual DOM capabilities making it well-suited to create complex UIs with multiple blocks such as navigation panels, search results and more. Overall the React framework would be well suited to the implementation of the front-end and presentation tier. I have previously made a web application using React which supported search functionality through the contact of a REST API, so I am confident that I could build this systems presentation tier with the use of the React framework to save development time and thus save cost for the travel agency. Reacts website actually features examples of e-commerce sites built using React, this is an example of another already established software artefact very similar to this system, further proving that it could be implemented.

Webpack or Gulp are applications that can be used in tandem with the chosen framework to increase security. They can compress code and format it to increase performance but also to make the analysis of the code harder, effectively making it more difficult to find clues on how to access the rest of the layers of the system.

Angular is another example of a Node JavaScipt-based framework in which the client web portal could be made. It would again ease development and has been used to create many e-commerce sites in the past, proving that it could be used to implement the specified design. The Angular framework isn't ideal at the moment as the community is split between the old version and the new rewrite. However it is still highly efficient and is supported by Google. Angular is optimal for creating large-scale applications which would be appropriate for the travel agency’s professional web portal especially if the employee's section was hosted in the same domain. It is highly testable and reusable making it useful for testing the system.

I have personally written many RESTful APIs for various previous projects, both personal and academic. I know from this experience that the API could easily be written in server scripting languages such as PHP or JavaScript with Node and Express and hosted on any number of places such as Brighton Domains. They could be utilized in the same way as queues in the messaging pattern, by passing and formatting requests as well as temporarily storing information should the connection drop and also queuing requests. It is possible to change endpoints/routes to some degree with technologies like the Express framework.

Node applications support the incorporation of many different packages to increase functionality. In the past, I have used the Express package to accept incoming requests and build a JavaScript-based REST API. I have also made use of packages like the MySQL package to communicate with databases. Therefore I am aware that Express can be used to create the type of API designed in this system. I have previously used packages like body-parser installed through NPM to accept multiple types of formats in the API and have built search, login and ordering functionality in JavaScript through Node. Due to these past experiences, I believe the system could be implemented in the way I have designed through the UML diagrams. JavaScript also supports the creation of all previously mentioned design patterns such as the observer pattern and so it is possible to build the functionality as specified in the design.

PHP is mildly outdated due to newer technologies but would still function for the implementation of this system as it is a server scripting language that could be used to connect clients, APIs and the database. I've used this for the creation of an e-commerce site in the past and so am confident that this type of system could be implemented again in this fashion. Due to PHP being an older language; while it would function, JavaScript would be the optimal choice due to its adaptability with the use of external packages and libraries. Any API should have CORS enabled to prevent errors when communicating between layers.

I am certain that the Database tier/ data management tier I designed can easily be implemented through any relational database-supporting software. I have used applications like MySQL both on a device and via web portals to store data like the bookings with relations through the use of secondary keys. They are contactable via many ways but the MySQL Node package is one way I have previously successfully used as well as direct contact through PHP. Ideally, the contact information for the database should be kept confidential from the other layers to help reinforce security. There are other possibilities for storing information such as the use of SQL Server or NoSQL.

While the application could be hosted by myself using the universities server the ABC travel agency would either need to purchase and run their own server or could run the API on a free web application service like Heroku as it is considered one of the best platforms for hosting REST APIs. The static web portal could be placed on a domain through GitHub pages or cloud storage. The google cloud platform allows the purchase of databases and Amazon’s relational Database service allows the installation of MySQL, SQL Server, MariaDB, Oracle BYOL and PostgreSQL databases.

Overall, through the existing systems, previous implementations and past experiences mentioned I believe I have proven that the chosen architecture along with the added architectural styles could be implemented through the mentioned technologies.

### References

* Gamma et al: Design Patterns. Addison-Wesley 1995.
* Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture, Foundations, Theory, and Practice, Wiley Publisher, 2010
* Mary Shaw, David Garlan, Software architecture Perspectives on emerging discipline, Chapter 3, Prentice Hall.
* L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley 2013
* Gorton, I., Essential of Software Architecture. Springer 2006
* Paul Greenfield, Service-Oriented Architectures and Technologies, Essential Software Architecture, Springer, 2006.