Architectural design

# Overview

Architectural design is of crucial importance in software engineering because it will have to take account of functional and non-functional requirements, to meet the stakeholders needs and requests and to help not to focus only on standalone elements losing the so called big picture of the system, always adhering to general principles of good quality. An important aspect is in fact to find a good trade-off between the high-level description near to the analysis and the low-level one near to the implementation.

Coming up with good quality design and architecture is mostly a matter of experience and in our field, is also known the importance of the reusability of other’s people work. So, we tried to build our system with various kind of this patterns and known architectural styles.

# Selected architectural styles

Architectural style: Client-Server

It’s the best known and most used architectural style for distributed applications.

We have a well-defined distinction between client and server, they play different roles and also, they are both accessible with a precise interface.

Travlendar+ is a mobile application and will have multiple mobile users, but still the computations will be located in some point where the global view of the system can be seen.

The system must guarantee scalability so, resources in the form of network segments, computers and servers must be added to the network without major interruptions of it.

Said that, we agreed that the best solution for our needs could be the Client-Server style.

* Server is invoked to provide one or a set of services, in our case, for instance, the computation of the best mobility option for the user according to all of his characteristics.
* Clients uses the provided services and initiate the communication through messages (JMS) or remote invocations. they interact directly with end-users using any user interface such as graphical user interface. These are the user logged in to the application.

In the end, we will talk about web clients, which means that in our scenario, they will ask the server to provide services for them and they will not store data locally. Furthermore, the architecture is OS independent, it relies on a central server that will avoid consistency issues between different devices of the same client.

Three-tier architecture

The Client-Server model does not impose any constraint neither about how logical layers (presentation, application/business logic, data) have to be distributed among the deployment units nor about the physical tiers have to be designed.

We taught that the best decision for us could have been the three-tier architecture where each tier is elastically scaled independently.

That’s composed of:

* First tier: Presentation layer located into the mobile application. Handles the interaction with the user. It’s also in charge of some simple validations of the data and the interaction with external systems.
* Second tier: Application/business layer. Process and executes both new requests of the clients and possible variations on older and not yet completed requests. It will collect information from the users and store them into the databases, but also it will provide to them answers for their requests. This tier will do both the visualisation (web) and the processing (business) portion.
* Third tier: Data layer. Dedicated to the storage of information in persistent memory. Data is stored at different locations (replicas) to improve response time and to avoid data loss in case of failures while consistency of replicas is ensured at all times.

This architecture permits us to achieve one of the design principles, that says to decouple where possible, in fact we have a solid distinction between logic and data, and also between logic and presentation.

Basic architectural patterns:

Stateless components

The state is handled external of application components to ease their scaling-out and to make the application more tolerant to component failures.

In this system cloud resources can display low availability and also component instances can be added and removed regularly when the demand changes.

So, components will be implemented in a way that the do not have any internal state. Instead it will be provided to the component with each request form external persistent storage.

User interface component

An interactive synchronous access to the applications is provided to users, instead the interactions in the application are asynchronous if possible, to ensure loose coupling (Principle 3 of the design principles).

In other words, the user interface component is like a bridge between the synchronous access of the user and the asynchronous communication with other components. Like for stateless components also these ones are obtained from external storage.

The number of components are scaled based on the number of user requests through the elastic load balancer pattern.

Elastic load balancer

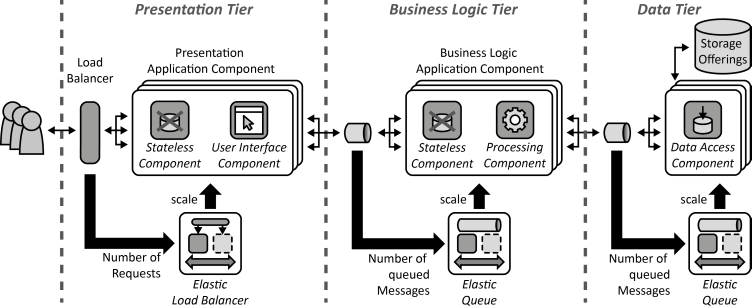
The number of required components instances is determined by the quantity of synchronous request coming from the users.

Processing component

Processing functionalities are divided into separate blocks and assigned to independent processing components each one implemented in a stateless way as explained above.

Elastic queue

Queues are used to distribute asynchronous requests to multiple application component instances. From the number of messages queued will be decided the number of component instances handling that requests.



# Architectural pattern

Model View Controller

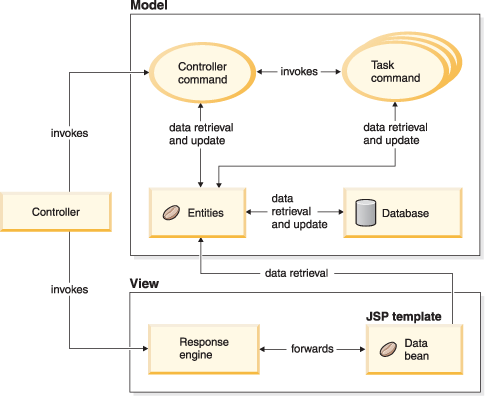
The model-view-controller (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

* The model contains only the pure application data, it contains no logic describing how to present the data to a user, but only data and operations associated to that.
* The view presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.
* The controller exists between the view and the model. It listens to events triggered by the view (so by the user) and executes the appropriate response to these events, usually calling a method on the model and passing the results both to the view and the model.

A key aspect is to separate data and its presentation. This not only makes the structure of an application simpler, it also enables code reuse and ensures loose coupling.

It also guarantees the divide and conquer principle allowing parallel development by separated teams in charge of different parts of the application.

An example of how can be represented the MVC pattern is shown in the figure below.



Commercial architectural system

To support the development and the execution on enterprise application with a lot of users and lots of requirements we suggest to use the Java Enterprise Edition (JEE).

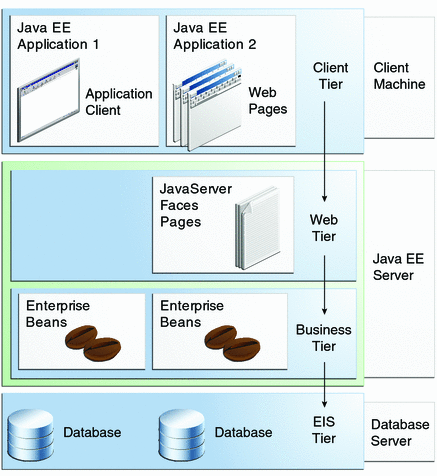
That’s a multitier architectural model composed by the client-tier running on the client machine (as we said in the overview of this section of the document, the clients involved in Travlendar+ will be web clients), both the web-tier and the business-tier running on the Java EE Server and the enterprise information system (EIS) consisting of databases or other external applications.

This approach will help us also to satisfy both functional and non-functional requirements like reliability, performance, security, scalability, availability, extensibility and interoperability.

The application tier will be divided into:

* web tier: managing the input insertion and the output visualization, written more likely in JSP or with Java Server Faces
* business tier: managing the computing and the execution of the business logic using components called Enterprise Java Beans (EJB) and also interacting with the database through Java Persistence API (JPA).

In the end the Enterprise Information System (EIS) is devoted to the data management and will operate like a DBMS.



# Deployment view