# 2.4. Architecture

Picking the best architecture that would fit our needs, was a long process, since the requirements of the library written in problem statement were not quite clear in the begging and they could be interpreted in various ways. Furthermore, information such as how the application would be used and what different types of librarians should be able to do was not provided, leaving us to interpret it.

## 2.4.1. Type of Architecture

When deciding on the type of system, we would be making, we first had to review the requirements given to us by the library. After reading the request and thinking of possible solutions we came to two possible types of systems we could make, to achieve the what was required:

### Client-Server

Single application which would be given to the client and allow them to manage the database (depicted in Figure 1).

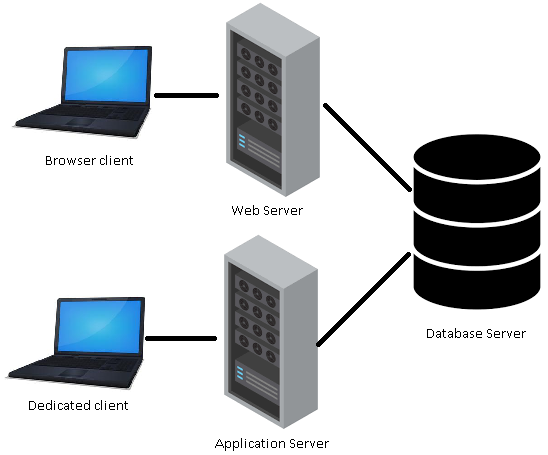


Figure 1

Pros

* Easy/fast to make
* Great performance could achievable

Cons

* If there is a new update, client could be required to get the newest version (without updating the client would not be able to experience newest features or the user’s applications might not work)
* Client system dependent
* Hard to maintain because of code duplication if multiple different types of clients
* A lot of work to add new client

Client-server Architecture can be classified into two models based on the functionality–

* **Thin-client model**- all the application processing and data management is carried by the server. The client is simply responsible for running the presentation software, it places a heavy processing load on both the server and the network.
* **Thick/Fat-client model**-the server is only in charge for data management. The software on the client implements the application logic and the interactions with the system user, new versions of the application must be installed on all clients.

### Multi-Tier (N-Tier)

Simple solution which has its business logic in separate project that could be shared by many clients, limiting code duplication (depicted in Figure 2).

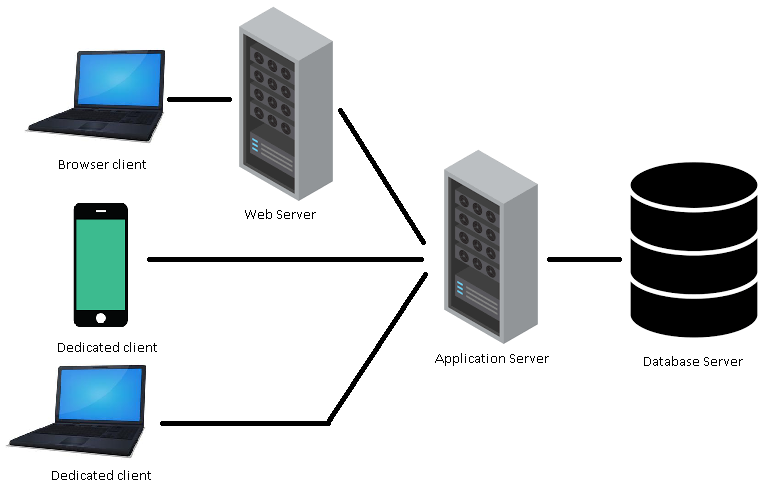


Figure 2

Pros

* If there is update to the product all the functionality is automatically accessible to the user
* Relatively easy maintenance
* Easy to expand with more clients
* Not dependent on the user’s hardware capabilities

Cons

* Harder to develop than the dedicated client
* Worse performance due to possible extra steps

After reviewing the pros and the cons, we determined that the N-Tier approach is better for our needs, as it would be easy to work with (for the user), has possibility to be simply expanded to new types of clients (such as mobile) and functionality updates automatically effect all consumers of service. Furthermore, it is required to have a service available to exchange information between the libraries and this would easily accommodate it. Meanwhile the cons of having a distributed application are rather minor, for example sacrificing a bit more time in development of the application for easier expansion in future and minor performance decreases, because now there would be a proxy between the client and the database.

## 2.4.2. Server side

Deciding the best technology, to use for the server is a hard and a time-consuming process, that is why we decided to do more research on this subject and came to a few possible solutions. The options considered will be explained and our final choice will be supported by arguments in the following text.

Here are some definitions worth knowing, before diving deeper into the rest of this subject:

* **Application program interface (API)** is a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a graphical user interface (GUI) by providing all the necessary building blocks.
* **Web service** is a generic term for an interoperable machine-to-machine software function that is hosted at a network addressable location (All Web Services are APIs but not all APIs are Web Services).
* **Microservices** is an architectural style that structures an application as a collection of services that are highly maintainable and testable, loosely coupled, independently deployable and organized around business capabilities.

**Accessing the web services can be done through two different ways:**

* **REST** (Representational State Transfer) - Is a relative newcomer and as an architecture style which does not require processing, is lightweight and is naturally more flexible than SOAP, requesting the user to just use URLs and can get a response in multiple different formats, while SOAP is restricted to only using XML.
* **SOAP** (Simple Object Access Protocol) - Old but still relevant solutions to the data exchange problem, SOAP uses a more rigid set of messaging patterns than REST. SOAP relies exclusively on XML to provide messaging services, the XML can become extremely complex and in some programming languages, programmers need to build those requests manually, which becomes problematic because SOAP is intolerant of errors, but when working with .NET languages, developers don’t have to even see the XML message.

When we decided on a framework to follow for creating our API we came to the conclusion, that there are only two frameworks worth considering for the job. Both of the considered technologies are APIs in the .NET Framework meaning, that they are easy to use in Visual Studio and with other .NET technologies. Besides that, as it just so happens all of us have had previous work experience in both of them:

### ASP.NET Web API

Is a framework that makes it easy to build HTTP services that reach a broad range of clients using REST. (Resource oriented)

Pros

* Easy to make
* Speed (Data is kept to the minimum)
* Many content formats (XML, JSON, CSV and more)
* Supports a wide variety of clients (mobiles, browsers, PC and many more)
* Uses the full feature of HTTP (like URIs, request/response headers, caching, versioning, various content formats)
* Open-source
* Allows for caching, compression, versioning

Cons

* Only 1 transport protocol (HTTP)
* No support for higher level protocols such as Reliable Messaging or Transactions

### WCF (Windows Communication Foundation)

Is a set of APIs in the .NET Framework for building connected service-oriented applications, mostly implements SOAP.

Pros

* Many transport protocols can be used (TCP, UDP, Named pipes and many more).
* Multiple encoding types (Text, MTOM, and Binary).
* Support for higher level protocols (Reliable Messaging or Transactions).
* Improved security.
* Allows also for One way and duplex communication, not limited to request reply.

Cons

* A lot of time is spent in configuring WCF services.

After reviewing the Web API and WCF we came to conclusion, that we wanted to have a service which would be easily maintainable and extendable, that is why we came to conclusion that we should use WCF instead of Web API. However, Web API could be a better solution for exchanging information with other libraries, as it offers more content formats and higher speed, but it was determined that both projects would be mostly used by computers connected though LAN, where TCP protocol would be a better choice than HTTP. Furthermore, WCF has better security, which although may not be the main concern is still something to consider. We determined, that due to time constraints WCF should be our main choice, as working in WCF is quite automated process and does not require too much code, since a lot of the work is already done behind the scenes. Besides the aforementioned points, during our research and previous experience, we’ve noticed that it is easier to make simple mistakes in Web API when developing solution (example: entering wrong URL).

## 2.4.3. Client side

For the client we reviewed multiple options, in the beginning we thought of creating a dedicated client, which could be for desktop or phone, but after considering the request of the library we got to know that some of their requirements for the client were:

1. Application should work on multiple different platforms (pc, phone…)
2. Easily accessible, not requiring any installation, as that would make students less likely to use it
3. Should be easily changed and updated for future expansions

This forced use to make a web-based client, which would be interactive, understandable and would allow us to expose the solution to almost every conceivable type of client. Since we were relatively new to web development and user interface was not the main concern of the solution, we determined to review this subject and decide on what technology is best to use, so it would be easy/fast to develop and later use. We limited our choice to ASP.NET solutions, since we had decided to work in C#.

Some of our options which we reviewed were:

1. MVC
2. Web forms
3. Web pages

After reviewing these choices, we determined that web forms are not for us, even though they are easy to create, they are being depreciated and we would not want to make a solution which could be considered outdated from the beginning. After comparing the options which were left with, we determined that MVC is the best choice, as it is frequently being used in actual businesses, it is Ideal for developing complex, but lightweight applications and it was the technology which we had the most knowledge in, so less time would be spent learning a new way of doing things, instead focusing on developing a solution.

To improve the created client, we decided to use various technologies such as:

1. Ajax
2. JQuery
3. Bootstrap
4. ASPX
5. Session
6. Cookies

And languages associated to them:

1. JavaScript
2. CSS
3. Html
4. Razor

## 2.4.4. System architecture

When working on plan for architecture, we decided to create multi-tier solution, which was similar to one used in previous project as it tried to achieve already expressed goals and it was easy to work with, while also allowing for relatively easily achievable testability of the code. The solution provided several benefits, such as easy addition of new clients and low-cost maintainability. Not only that, but also, this architecture helped us achieve the goal we had set for ourselves, for this project, that being to pursue “high cohesion and low coupling”.



Figure 3

The previous solution (Figure 3) was N-Tier architecture with simple web client and separate project for tests and models. Even though the current architecture was inspired by one used previously, there were some improvements made. The changes furthered the code testability, simplified the solution and improved solution based on previous experience. Most notable changes to architecture, as shown in Figure 4, is that there is no longer project for model layer and documentation for service, as we determined that they were not required.

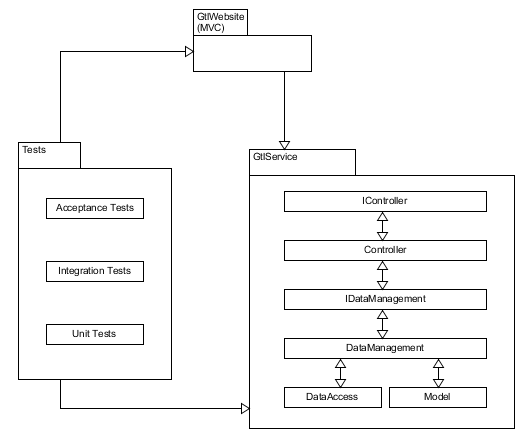


Figure 4

As seen in Figure 4, we have three projects, with each of them being used for generally different purpose.

* GtlService- The brains of the product, providing services who could be consumed by user and performing corresponding actions in the database.
* GtlWebsite- holds user interface, allowing consumer of the service to easily interact with it.
* Tests- project dedicated to improving and maintaining the quality of product.

GtlService service was further subdivided into layers, where each of them is responsible for different task (Controller-communication between projects, DataManagement- making calculations and changes to objects, Model- contained objects used, DataAccess- connecting and working with database). Some of the Layers had interface, as it helped with testing and maintenance of the software. Project called “Tests” contained folders for each type of dynamic test used in project, so they would be easy to keep track of.

## 2.4.5. Design class diagram

Diagram could be generated and talked about after the project is made

Used:

<https://slideplayer.com/slide/7640839/>

<https://www.tutorialspoint.com/software_architecture_design/distributed_architecture.htm>

# X.X.X. ER (mention What they are and why do we need them)

# X.X.X. EER (What they are and why do we need them)