UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS301 - Software Engineering

MIDISENSE

Architectural Design Specifications Demo 3

Team Name: NoXception

Team Members:

Name	Surname	Student Number
Claudio	Teixeira	u19028581
Adrian	Rae	u19004029
Hendro	Smit	u17004609
Mbuso	Shakoane	u18094024
Rearabetswe	Maeko	u18179802

Contents

1 Architectural Design Strategy

1.1 Design based on quality requirements

This is the strategy we have chosen while designing the architecture for our system.

Since quality requirements are the capabilities that must be present in the system after its completion, they represent a metric of which we can judge successful product after the completion of this product. Having a vision is important for any successful project, thus we have chosen to base our design strategy on this. That way as we progress with the project we can verify our progress and completion with a visible metric to determine how we are preforming.

Additionally, as the project grows outside of the given scope, new features can be addressed based on the quality requirements as to not waste time and resources. With this strategy in place, NoXception is able to continuously work within our vision. In the event that a quality requirement changes, we should use that information to adapt the system to be in line with the updated requirements of the system. For example, we anticipate our system to be reliant on some external libraries that may go down for their own maintenance once a week/month. If we update the requirements to have our system constantly online, then we would need to limit our use of libraries that may need maintenance in the life cycle of the system.

2 System Quality Requirements

2.1 Availability

We expect the system to be able to consistently cater to requests, while experiencing the least amount of downtime due to maintenance and system errors. Ideally, there should not be any distinction made between the availability of the interface and that of the functions it provides, as the usability of the service depends heavily on the content delivered by the functional services. Thus, at an arbitrary point in time after deployment, we expect the likelihood of a particular system being operational to be in excess of 99%

2.2 Auditability

The system will log all requests and all responses for all services provided by the system, for the sake of knowing:

- When a particular request was made.
- What the context of a request is.
- The service being utilised.
- The success or failure rate of services.

2.3 Extensibility

The ability to introduce aspects to subsystems, such that existing structure and dataflow are minimally affected, is a crucial consideration in the design of the system. The potential growth of services related to analysis and classification necessitate a design that is modular and promotes integratability.

Subsystem design should prioritise interchangability, such that services can easily integrate with one another and are loosely coupled, should they need to be completely replaced at any moment. This requirement can be quantified by the *mean time to integrate* measure, the time required to add a new service or function, which should be monotone decreasing over the course of development.

2.4 Integrity

A key assumption of the midi file interpretation process is that user files which are of a valid type remain of a valid type. Files that are consumed by the system must retain their 'validity' and not be corrupted or truncated while being sent to the server or being stored.

The same is true of any data objects that are persisted or transferred. There is a need for frameworks that provide error-correction and validation mechanisms for transfer and persistence, embedded within their methods.

2.5 Maintainability

There should be a sense of ease in development, when new requirements should be taken into consideration. The design of the system is one that should minimise the intervention between system and developer. Thus, we aim to implement a self-sustaining system which minimised the frequency of errors or other situations which require correction or intervention.

The expected frequency of errors requiring intervention is set at least 7 days between major errors and at least a day between minor errors. The degree of maintainability is affected greatly by the availability of 3rd party technology and thus stable libraries should be used.

2.6 Performance

The system should have defined criteria against which the capabilities of system services can be measured.

- Services related to the uploading of files should conclude within 3 seconds.
- Services related to the parsing of files should conclude within 15 seconds.
- Services related to the interpretation of pre-parsed files should conclude within 3 seconds.
- Services related to the analysis of pre-parsed files should conclude within 2 seconds.

- Services related to the exchange and display of parsed data should conclude within 2 seconds.

It should be noted that such metrics do not Factor in round-trip request and response times and latency due to network performance, but should factor in the capabilities of the client making requests to the system.

2.7 Reliability

The reliability of the system shall be treated as the consistent ability to perform requested operations without exhibiting a point of failure. Failure in the system's ability to perform operations may occur due to faulty business logic, deprecated technologies or environmental factors.

Systems should have built in safety mechansims for detecting errors and producing meaningful responses if errors do occur. The system itself should be deployable without point of failure.

The reliability metric for the system is set at 95%, that being, at an arbitrary point in time, the likelihood of the system experiencing a signifcant system error is 5%.

2.8 Scalability

The system should meet the demands of a growing user set. The software is specialised and we expect the cumulative market roll out to extend to 1000 users. Thus, the system should make use of vertical scaling and, if appropriate, horizontal scaling.

2.9 Testability

Rigorous tests should be conducted to determine whether subsystems (and entire systems) are performing as intended. These should also be used to quantify concerns pertaining to availability, performance and reliability. Such test should be automated for the sake of repeatability and convenience, provided by a framework that provides consistent testing metrics.

- Automated Unit Testing should be conducted at a subsystem level of granularity.

- Automated Integration Testing should be conducted when integrating sub-systems within the system environment.

In both circumstances, the tests should provide a mechanism for knowing whether or not services are provided, granted all preconditions are met and no exceptions are thrown and that post-conditions hold once the service has been provided.

2.10 Usability

The system should cater to the amateur user, with no prior experience in either the realms of music or computing. There should be descriptive error messages for services which do not provide the correct output. The interface and functional components should be minimal but also visually descriptive such that extra documentation would be helpful, but not essential. This can be quantified by relative index as filled out during alpha user feedback.

3 An Overview of Architectural Styles

We will be using a hybrid system for our MidiSense project. This entails using a combination of the MVC (Model View Controller) Architectural Pattern and a Multilayered Architectural Pattern and Service Orientated Architectural Pattern.

3.1 Component-based

This pattern is used in conjunction with multilayered architecture to allow for easy usability and testing. By separating concerns into different areas testing is also separated and becomes more efficient to manage. It also offers us a dynamic view of our front end which is very useful.

3.2 Database-centric

This pattern is used in conjunction with the multilayered style to allow for easy usability and testing. The style is used in our core functionality in order to manage uploaded files and access the AI algorithms. The integrity and reliability of the system should increase if a midi file can successfully be uploaded to the database.

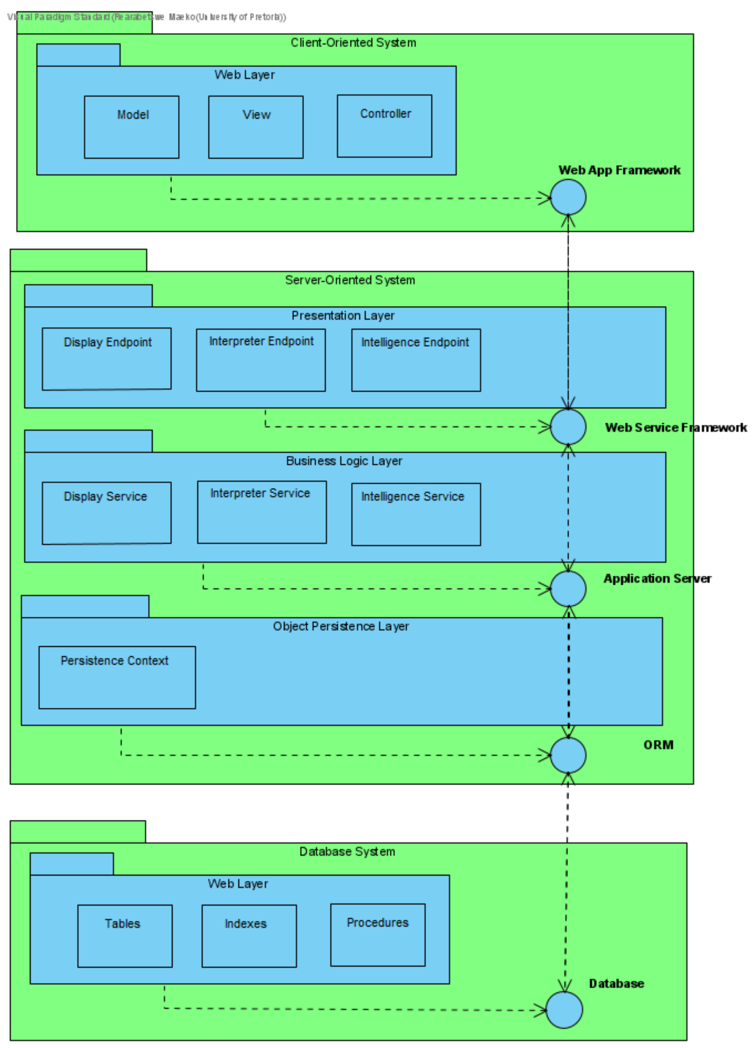
3.3 Multilayered Architecture

In this layer, Presentation, application processing, and data organising functionality is physically separated. It helps performance of the system by delegating different roles to different layers.

3.4 Service-oriented architecture

This architectural pattern supports service orientation in the project. It is self contained and logically represents a repeatable business activity with a specified outcome. This pattern integrates distributed, separately maintained and deployed software components in our project. This helps us in addressing scalability of the system we choose for the project

4 An Overview of Architectural design and pattern



This diagram below provides a high level overview of the software architecture of the system. It consists of the architecture components and the infrastructure between them. In our multi-layered architectural approach, the infrastructure provides the ability to limit access of components in one layer with components in the same layer or a lower one.

4.1 Layered Pattern

Components are distributed among a system of layers, whereby each layer encapsulates a set of related operations. In this manner, each layer makes use of the services of the layer directly beneath it, such that the implementation of behaviour is hidden from components not within the same layer.

This aligns with the quality requirements of extensibility and performance, as new components and functionality can be both introduced and optimised at a layer-wise level of granularity without massive changes being made across the entire subsystem.

4.1.1 Client layer

This layer contains calling applications that will consume the web services, allowing for integration with external services. Applications found here are the website of a RESTful web services.

4.1.2 Presentation layer

The presentation layer provides components for the user interface to interact with in order to receive the presented content. For the correct content to be presented the relevant requests need to be used by the client layer for easy interaction and consumption. In order to present the content, this layer must communicate with the preceding layers to come.

4.1.3 Business Logic Layer

The layer provides an aggregation point to allow for combining services to achieve the business goal of the project by executing a sequence of use cases.

4.1.4 Object Persistence Layer

An abstraction layer between how data is stored in the persistence provider and how the application processes said data. One can swap out the underlying provider without affecting the system as a whole. This is possible because the layer allows for decoupling the system from the storage provider.

4.1.5 Database Layer

This layer's primary objective is to physically store information used in the system for long term retrieval and process.
The communication between the external subsystem in the client layer and the back end application will take place using a request-response model.

4.2 Multi-tiered pattern

Components of the system are divided among disjoint subsets of hardware and software, where each is connected by a specific communication medium. This allows for separation of concerns across different software media and enables greater auditability of system processes.

4.2.1 Client-Oriented System

The goal of the client oriented system is to help the user to interact with the rest of the system in the most intuitive way possible. The MVC design pattern facilitates this interaction in 3 logical parts that integrate the back-end and front-end systems.

4.2.2 Server-Oriented System

A server oriented system highlights a constant communication of a number of services with each other in various ways such as sending data back and forth or processes coordinating an activity. The most defining factor that is present in a server-oriented system is interaction. This truly allows multiple platforms and subsystems to function independent of one another and communicate when necessary.

4.2.3 Database System

This layer makes use of a cloud service provider that is responsible for allocating storage space where the midi files will be located.

5 System Constraints

5.1 Imposed by Equipment and Technologies:

• Memory limitations – the efficiency of tasks is affected by the quantity of users using the system at any one time and is constrained by the working memory of the web server along with the efficiency of subsystem methods. Thus, the number of active users restricted and must be addressed when nearing its thresh hold.

• Storage limitations – the amount of data able to be stored is bounded by the database used to store it, and the file system which manages system data.

• Latency and Bottlenecks – the communication medium between the web server and database/file system is subject to a finite rate of transmission and thus requests are not necessarily reflected instantaneously. (especially when there are vastly many users making use of services concurrently)

• Third-party Interoperability and standards – the site will have to adhere to the standards imposed by web-services and their browsers along with the third party governing the database software.

• Miscellaneous Network configurations - such as the server-side topology, server configuration settings and firewall settings affect how different layers within the architecture interact with one another and how efficient such communication is.

6 Choice of Technology

6.1 Application Framework

In terms of our back-end Application framework two of the best choices to consider are ASP.Net and Spring Boot. These are all optimal choices for the system as each of these Frameworks fulfill multiple of out quality requirements. Both of them allows for fast, easy development and maintainability as well as extensibility.

ASP.Net is an extension of the .Net platform which means it has the ability to use tools and libraries for building apps provided by the .Net platform. ASP.Net also adds various features to the platform such as and authentication system for logins and libraries for common web patterns A huge advantage that can be gained from implementing a .Net framework is the modular design that it provides such that the application can be taken apart and worked on modularly. This provides a huge advantage when it comes to updating code and building the system. A feature such as this ties in heavily to our service oriented architectural style which states how we would like distributed, seperately maintained software components The disadvantages of ASP.Net however do take away from most of its advantages as Object-Relational support issues are present in the sense that the framework is not as flexible in supporting the new and emerging database designs. ASP.Net is a Microsoft product therefore any limitations that the company may impose also affect the framework and therefore the project aswell. Therefore the huge drawback that seems to present itself with a ASP.Net framework would be that it is not as flexible as other application frameworks

Spring Boot is an extension of the Spring application Framework for the java platform. This framework allows developers to build standalone applications and containers by providing an easy way to develop web applications.
Spring Boots provides us with several advantages. It allows for an easy implementation of dependency injection. It contains numerous trusted and flexible libraries that can easily be implemented. It allows for simple unit and integration testing. It also provides the ability to create multiple stand-alone containers to test functionality. However, the one drawback that is often experienced when navigating Spring Boot is a lack of control when it comes to installing dependencies that one would not have to use in a project. This occurs as Spring tries to make the development experience as simple

and easy to use as possible for the developer. This complies the to requirement of extensibility we have outlined in the requirements as an application server such as Spring Boot provides for a very interchangeable and modular project as well as the ability to add new aspects to subsystems. Spring Boot also fulfills the criteria of having a multilayered architecture as it helps the process of delegating different components to different layers It also allows for easily setting up database connections and developing RESTful API endpoints to allow for easy communication between the client side and service side of the system.
Therefore the technology we have decided to use as an application framework is Spring Boot. Where ASP.Net lacked in its ability to be flexible in terms of being accommodating to new and emerging technologies, Spring Boot thrived in allowing multiple way to separate components in the system as well providing a way to implement the multi tiered architectural design of out system by allowing easy communication between the client-side, server-side and database of our system.

6.2 Front-end Framework

Given the requirements that we found to be crucial for the system to operate efficiently, we came to the choice of two frameworks that could satisfy them, namely React and Angular. After weighing in the pros and cons of both frameworks, we came to the following conclusions.
The benefits of React include allowing the team to create a more personalized application solution which would be beneficial in separating MidiSense from competing products. Additionally React is more suited towards 3rd party integration's.In regards to usability, both frameworks should deliver an acceptable UI/UX for end user, however the learning curve is much more difficult for Angular and would therefore take the team longer to fully utilise the framework. This would also cause the maintainability to suffer as the team would struggle more to implement fixes compared to React.In terms of scalability and performance, the deficits of Angular created doubt within the team regarding future implementations of the front-end due to Angular's slower speeds and its difficulty to implement third party libraries and software. In regards to the other quality requirements both frameworks would otherwise perform similarly from the developers' and clients' perspectives. The choice of React was then chosen based on the benefits it provided over Angular.

To further clarify our choice, React is a JavaScript front-end library used for building user interfaces consisting of UI components. React makes building interactive user interfaces efficient and simple as it is components based which enables developers to build entire web interfaces comprising of different components. The key advantage that React JS provides is faster rendering. React uses a virtual DOM system that ensures that when changes are applied to the virtual DOM, the minimal scope of necessary DOM operations are calculated. Using these calculations, the real DOM tree is updated which therefore allows for minimum time consumption. Configurations like this allow for a better application performance and user experience. A disadvantage often found when using React are the complexities faced when using JSX. JSX is a syntax extension that allows HTML and JavaScript to be used together. React answers both out scalability and performance requirements. As has been established a React interface comprises of a number of different components all working together to bring the interface to the user in an efficient and timely manner.

6.3 Database

When referencing our quality requirements when coming to a decision on a database we ended up coming to a choice between PostgresSQL and MySQL.
Postgres is an open source Object-relational database that is well known for its reliability and performance. It is also a very flexible solution as it supports advanced data types and performance optimization features.
When it comes to the quality requirement of a system that preserves integrity Postgres is a great candidate as it has built a reputation for its feature robustness and performance. For example PostgreSQL supports most of the major features of SQL:2016. That is to say out of 179 features that are required for full Core conformance, Postgre has atleast 160. This is more than almost any other database engine. This makes it an excellent solution to having an integral and Reliable database. However, when considering databases, one must also take into account what the database will be used for in terms of the system. Our system only requires the base features and uses of a database as we only store midi-file components and identifiers. Thus the number of features provided by the database weighed less in our consideration than compatibility and speed. These are two areas that Postgres does fall short in. Firstly in terms of compatibility, the support for a Postgres engine database is found far less in open source software than more popular databases such as MySQL. Secondly when it comes

to speed it is often found to be slower in performance than other popular databases such as MySQL which is an important requirements listed.

MySQL is a relation database management system and our choice for the database of the system. This web database allows us to store our midi files so that they can be easily retrieved by the frontend when processing has been completed. Perhaps the greatest advantage of MySQL is how widely known and used it us. Multiple small or medium sized websites use MySQL which inly helps to boost its reputation as a reliable database for the architecture. MySQL has also built a reputation of being a database framework that is easy to understand and has a great performance which is ideal as speed is critical. However, the one disadvantage it does have is that it was not built for large data. This However, should not be a problem for our system as we are only storing midi files and not user information. MySQL as has already been established is an easily maintainable database which can easily be understood and extended upon. It also provides great availability to developers which would reflect in the system.