Notable changes to the report are either directly highlighted or section headers are highlighted to indicate the whole section is novel.

Final Report

Computer Science 4471B
Software Design and Architecture

Submitted to
Professor Nazim Madhavji
April 2nd, 2020

Written by Group 4

Gavin Lu | Paul Li | Daniel Le | Yang Guo | Ryan Chen

Table of Contents

1. Documentation Roadmap

1.1 Scope and Summary

This document provides an overview of the system architecture. A discussion of changes made since the previous report is noted. Then, five views of the system are documented in detail that cover the service-oriented architecture of the system and the additional models implemented.

The views covered in this document are the module, component and connectors, allocation, security, and exception handling. The first three views detail the overall architecture structure of the system that is designed to satisfy the quality requirements of the system. Security and exception-handling quality views were carefully designed to ensure the system security is highly controllable.

Rationale is provided to justify the software design in terms of each individual view as well as in terms of the holistic design. The relationship between views is discussed and details on how stakeholders can use the information in the document is also outlined.

1.2 How the documentation is organized

The document is divided into seven distinct sections:

- **Section 1** opens with a summary of the five views of the system analyzed along with how stakeholders can use the information.
- **Section 2** discusses changes made to the project since the previous report, which importantly includes the shift from IBM cloud to Microsoft Azure.
- **Section 3** summarizes the system in terms of its functionality as well as its planned architecture.
- **Section 4** describes how each view is documented before detailing the four views presented in the document: the module, components and connectors, allocation, and error-handling views.
- **Section 5** describes the mapping of the four system views together and how elements across all views are connected.
- **Section 6** explains the rationale behind the design decisions of the system architecture.
- **Section 7** details the system implementation and how the system achieves the planned architecture, covers the system testing and provides test cases, and provides instructions on how to run the system locally.
- **Section 8** discusses the limitations of the project and setbacks that occured.
- **Section 9** contains supplementary information such a list of acronyms used and references to other work

1.3 View overview

Five views are presented in the document that describes the system. In the module view, the system is broken down into its modular components to show the decomposition of the system. In the components and connectors view, the system components are characterized with how communication occurs between modules, highlighting the purpose of the message bus and the publish-subscribe model. In the allocation view, a deployment view is described to document the quality attributes of the system. In the security view, user credential handling is documented.

The error-handling view discusses how errors are handled in the system. Finally, a combined view presents the total system that the system views derive from.

1.4 How stakeholders can use the documentation

Stakeholder interest and concerns with the software architecture document varies based on their interests. The documentation is intended to communicate key details to all stakeholders with various levels of detail.

Stakeholder	Views of interest
Service Users	**Component-and-Connector view:** Users will want to be familiar on how to use the available services. Since the system revolves around a pub/sub model, the users should be familiar with how to utilize the available services.
Acquirers	**Allocation View:** Anyone acquiring the system might be curious on the environments the software will be running on because this will correlate to the cost of running the system. For example, noticing an emailing service costs 5,000 dollars a month to run might seem unfeasible.
Developers	**Module View:** A developer would be interested in how modules within a system communicate with each other especially if they are looking to add a new module they would need to know the form of communication is some sort of middleware. **Component-and-Connector view:** Depending on the development work the developer might be concerned with how services are connected to a database. **Allocation View:** Developers who are developing new modules might be concerned with the existing environments the software will run on or utilize. For example, knowing you are using an SQL Database on Azure will help dictate how a service will be implemented.
Maintainers	**Module View:** Maintainers might be concerned with how each module will work together. For example, modifying a module might have a rippling effect. This is very good to know as a maintainer. **Allocation View:** With a deployment view maintainers will need to be familiarized with current limitations, performance issues and scalability concerns that the software is running on.

2. Changes from Progress Report 2

2.1 Services Offered and Service Requirements

Changes to the services offered were made. In particular, several reasons caused the ‘portfolio feature’ to be cut. First, since many features were planned for the project, the priority of features were set to be implemented in order and portfolios were low on the list. Second, the feature was vastly more complex than the other features (requiring subfeatures such as logging in, tracking portfolios in a database, adding and removing stocks, and displaying charts).

3. System Overview

The goal of the web application is to provide services to individuals that present informative metrics about company stocks. Users request a specific service from the application which then queries the application database, returns queried data, and displays the data in meaningful graphics.

To achieve this goal, a service-oriented architecture (SOA) will be adopted. SOA helps keep the services provided by the system modifiable and reliable. The design facilitates modifiability by reducing dependencies between services and maintaining loose coupling; both help prevent ripple effects caused by changes in service functionality. The design also grants encapsulation to our code by hiding the details of service implementation from users. Thus, the service is highly controllable from its execution environment in order to foster the reliability of the system.

4. System Views

4.1 How a View is Documented

The document presents each view with five subsections:

a) **Primary presentation.** This is a diagram that visualizes the components of the view as entities with relations.

b) **Element catalog.** This is a discussion of the elements in the primary presentation that elaborates what each entity represents and what the relationships mean.

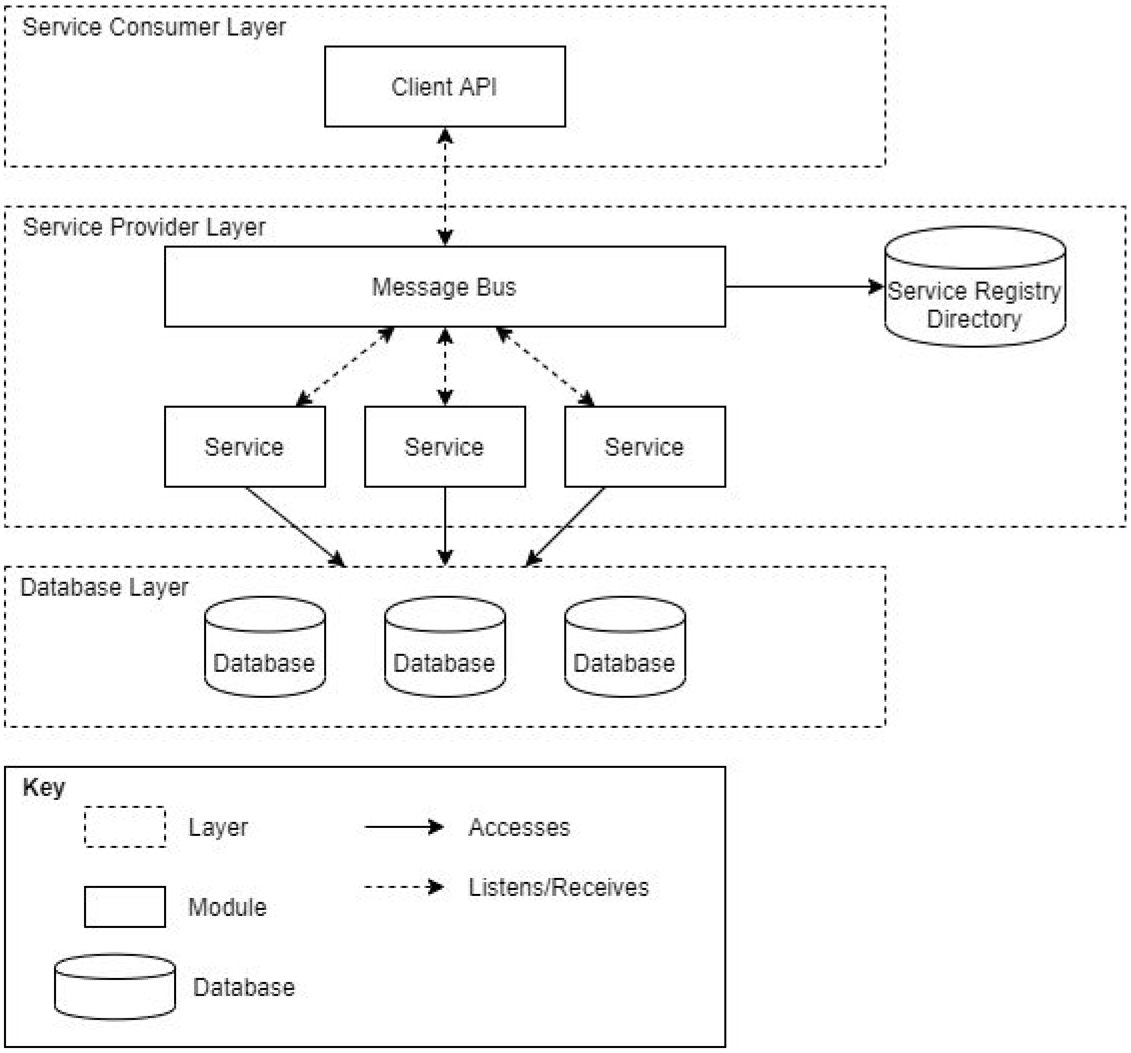
c) **Context diagram.** This diagram depicts how the view interacts with the system's environment.

d) **Variability guide.** If a view has aspects of the primary presentation that can vary, details on how the variability are described here.

e) **Rationale**. This section describes the thought processes that went into the design of the view.

4.2 Module View

4.2.1 Primary Presentation



4.2.2 Element Catalog

4.2.2.A Elements and their Properties

Client API - The Client API is the only component that consumes services from the application. Clients access the API through a web browser, which gives clients an interface to the various services offered by the application. The client API will be using the Model View Controller design pattern for interfacing between the client and the message bus.

Message Bus - The message bus handles message transfer between the client and the services.

Service - Multiple services are offered by the application that completes a specified request.

Database - Database that holds stock data.

Service Registry Directory - Database that holds provided services. The registry allows services to be discoverable on runtime.

4.2.2.B Relations and their Properties

Listen/Receive Relations - These relations represent asynchronous, bidirectional calls.

Accesses - Access relations represent components that have unidirectional access to a database.

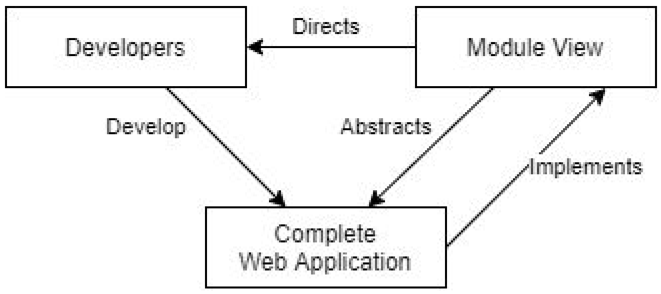
4.2.2.C Element Interfaces

Message Bus - The message bus provides an interface to the client API that gives clients access to various services.

4.2.2.D Element Behaviour

The message bus acts as the intermediary between the client and service providers. This architecture improves modifiability and interoperability by allowing different services to be swapped and provided to the users. While the backend services can be changed, the client interface remains unchanged.

4.2.3 Context Diagram



The module view serves as the web application plan. The web application implements details of the module view as developed by the application developers. The module view directs developers while developing the web application on how to implement the system with the module view's designed patterns.

4.2.4 Variability Guide

The module view features the variability of services offered that is designed into the architecture. Different services can be implemented and interfaced to the message bus that does not affect the client-message bus interface.

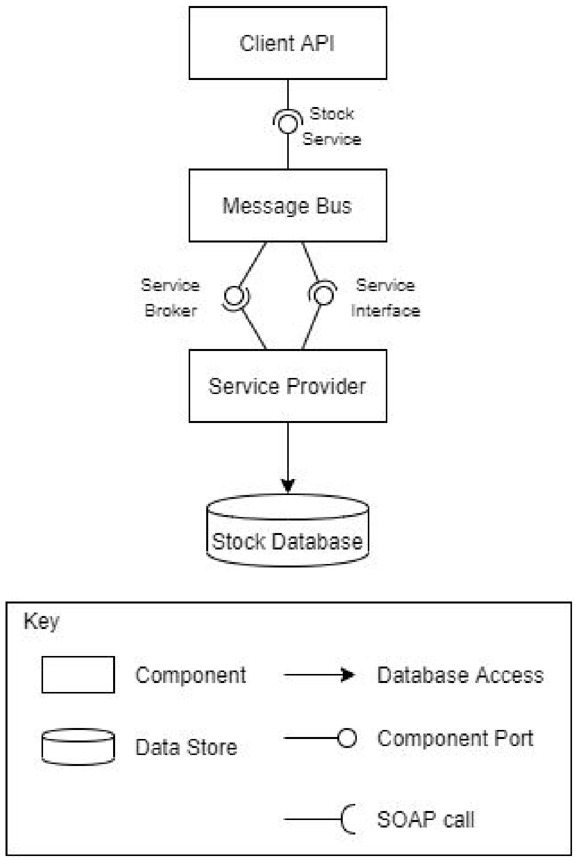
4.2.5 Rationale

The module view showcases the design of the application that features a service-oriented architecture using a publisher-subscriber model. The client API is the sole component that asks as a service consumer, while many services are offered as service providers. The message bus separates the services from the client interface to promote greater runtime modifiability and implements the publisher-subscriber pattern.

A useful feature available to service-oriented architectures is the service registry directory. This database holds various services offered by the application that allows for runtime discovery of various services.

4.3 Component-and-Connector View

4.3.1 Primary Presentation



4.3.2 Element Catalog

4.3.2.A Elements and their Properties

Client API - The client API is the component in the application that clients access through their web browser. Many clients can connect to the application at one time and access one of many service providers.

Service Provider - Service providers represent the system components that provide the services offered by the system.

Stock Database - This represents the database that will hold the stock data available to the application.

Message Bus - The message bus implements a publisher-subscriber model that acts as an intermediary between the client API and service provider.

4.3.2.B Relations and their Properties

Message Bus Relations - The relations the message bus forms with the service providers and client API implement the publisher-subscriber models. The message bus listens for incoming events and forwards the events to their respective destinations.

Database Access - Access relations represent components that have unidirectional access to a database.

4.3.2.C Element Interfaces

Message Bus - The message bus provides an interface to the client API that gives clients access to various services provided by many service providers.

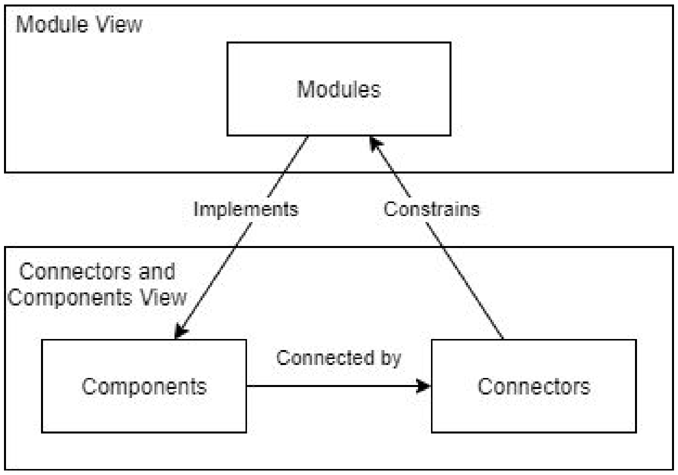
Service Provider - Each service provider provides an interface to the message bus. Service providers may provide different interfaces since they may vary in programming language.

Database - Service providers query the database for results through the database interface.

4.3.2.D Element Behaviour

The message bus acts as the intermediary between the client and service providers as prescribed by the service registry directory (not depicted in diagram).

4.3.3 Context Diagram



The components and connectors view directly relates to the module view. Modules implement components. Connectors constrain modules by dictating what modules can communicate and how the communications take place.

4.3.4 Variability Guide

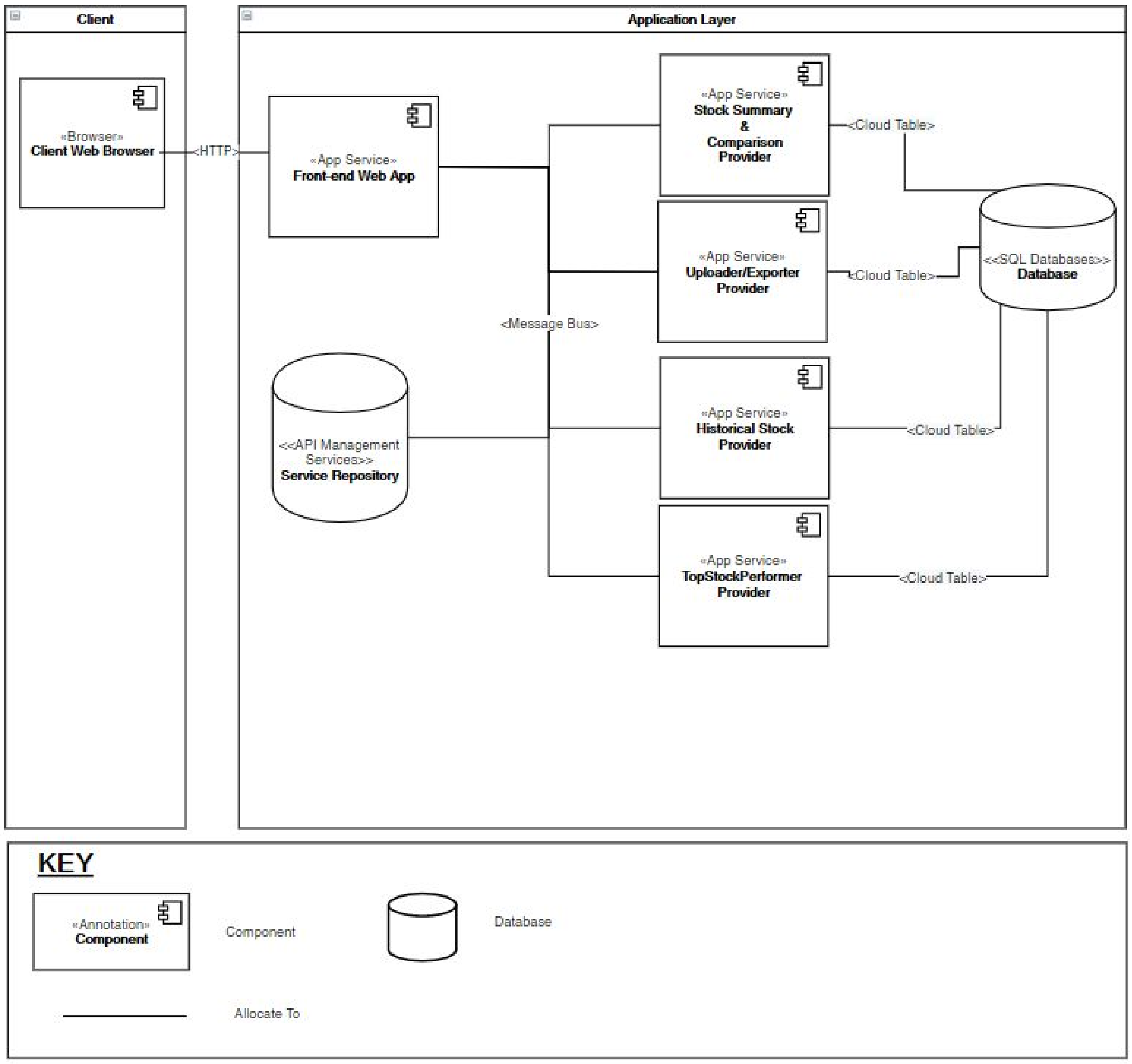
The components and connectors view abstract away one of the key variabilities of the system, which is the variability of the service provider. While service providers may vary in services offered and programming language developed in, the connection between the varying components remain the same to the message bus.

4.3.5 Rationale

The connectors and components view highlight the publisher-subscriber model built into the system's software-oriented architecture. The message bus handles the communication between the client API and the various service providers through subscriptions to messages and publishing back messages.

4.4 Allocation (Deployment) View

4.4.1 Primary Presentation


Client
Application Layer
«Browser» Client Web Browser
<HTTP>
«App Service» Front-end Web App
«App Service» Stock Summary & Comparison Provider
«App Service» Uploader/Exporter Provider
«App Service» Historical Stock Provider
«App Service» TopStockPerformer Provider
<Message Bus>
<Cloud Table>
<Cloud Table>
<Cloud Table>
<Cloud Table>
<<SQL Databases>> Database
<<API Management Services>> Service Repository
KEY
«Annotation» Component
Component
Database
Allocate To

4.4.2 Element Catalog

4.4.2.A Elements and their Properties

App Service - This would be an Azure App Service which will host a component that will be mainly web apps.
SQL Databases - Azure's implementation of cloud database. Used to store data.
Browser - How clients will access the front-end app that will utilize the services available.

4.4.2.B Relations and their Properties

Message Bus - These relations represent asynchronous, bidirectional calls to communicate using a message.
EF - Opens a connection to create an unidirectional access to a database.

HTTP - A communication protocol to transfer data over the network.

4.4.2.C Element Interfaces

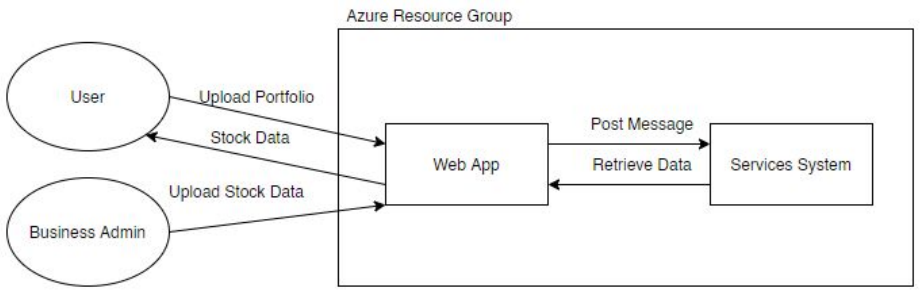
Message Bus - The message bus provides an interface to the client API that gives clients access to various services provided by many service providers.
Database - The database will have specific database schemes as an interface to the data.
HTTP - Requests will have to follow http request standards and the requests will have to adhere to the implementation of the application.

4.4.2.D Element Behaviour

The message bus acts as the intermediary between the client and service providers as prescribed by the service registry directory (not depicted in diagram). The message bus will be implemented using Azure's Service bus where the other components will be subscribed and listening for messages that adhere to their service.

4.4.3 Context Diagram

In terms of external uses on the system they might involve users such as the regular users and maybe business admins who want to add more stock data into the system. Specifically these users communicate to the Azure Resource Group



4.4.4 Variability Guide

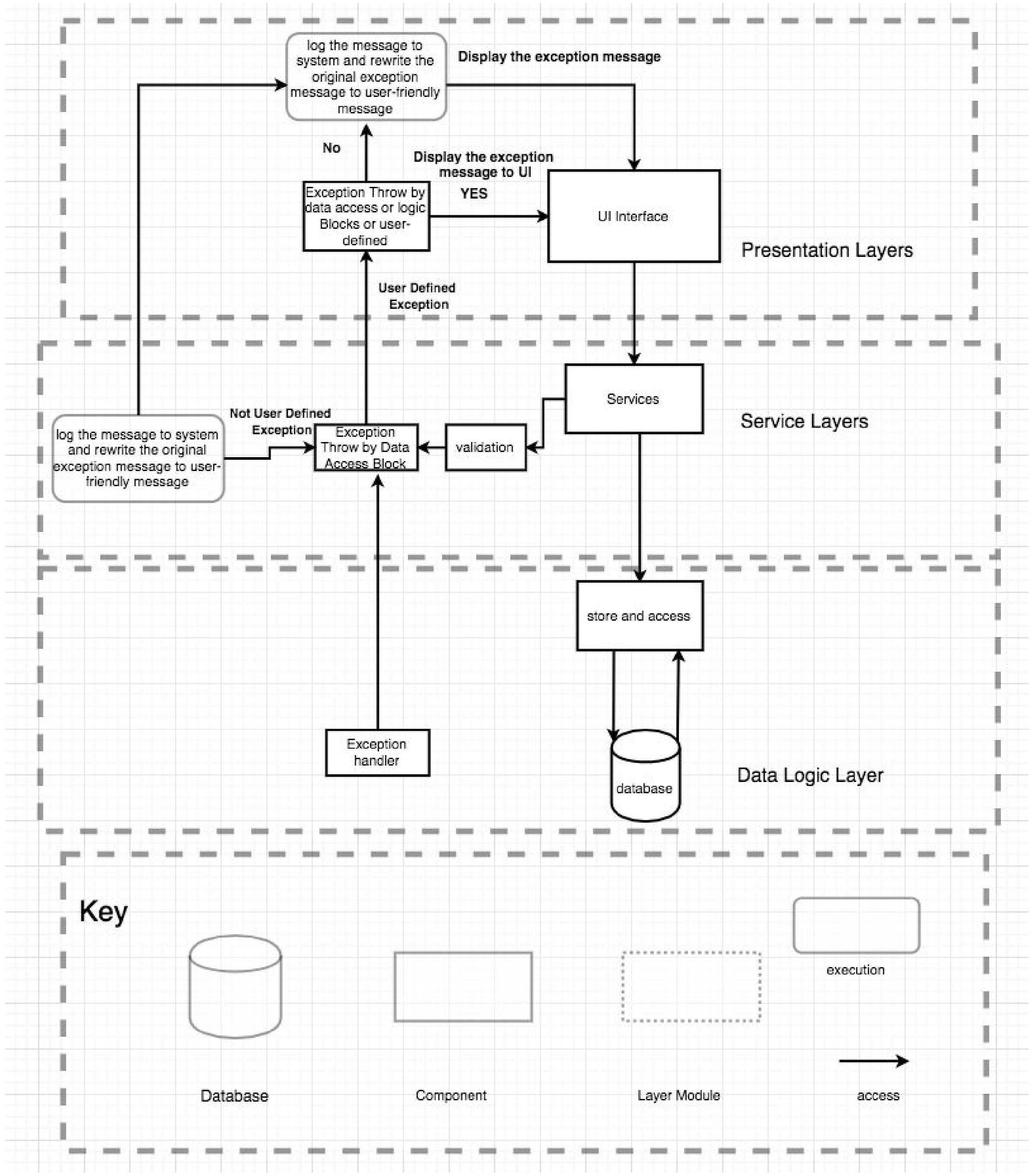
The deployment view is ideally to help document properties such as performance, availability, security, and safety within a system. Looking to see what environment each component belongs to can help clarify and rule out some pros and cons with the given choices such as possible bottlenecks. Following some benefits of an SOA is that each service can be implemented however, they wanted as long as it still complies with how the system operates. So this gives variability in what environment the components are built in.

4.4.5 Rationale

The deployment view as mentioned above is used to help document properties such as performance, availability, security, and safety within a system. For example, performance of the Azure's SQL Database might be less practical and causing bottlenecks when it scales. This can be subjected to using a different environment such as Azure's Cosmos DB which is more suited for global distribution. This provides us a visualization of the physical hardware and the software of the system to help further understand some of properties or constraints that are available.

4.5 Error-handling View (Quality)

4.5.1 Primary Presentation


log the message to system and rewrite the original exception message to user-friendly message
Display the exception message
No
Display the exception message to UI
YES
Exception Throw by data access or logic Blocks or user-defined
UI Interface
Presentation Layers
User Defined Exception
Services
Service Layers
Not User Defined Exception
log the message to system and rewrite the original exception message to user-friendly message
Exception Throw by Data Access Block
validation
store and access
Exception handler
database
Data Logic Layer
Key
execution
Database
Component
Layer Module
access

4.5.2 Element Catalog

4.5.2.A Elements and their Properties

Execution Elements – These represent functions that are offered by the application to complete a specified task.
Database - Database that holds stock data.
Exception Handler – The component that handles exceptions at the data level.
Services - Services that are offered by the application.
Validation - This component validates the function provided by the service.
Store and Access - Component that store and retrieve data from database

4.5.2.B Relations and their Properties

Accesses - Access relations represent a component that has access to another component for function.

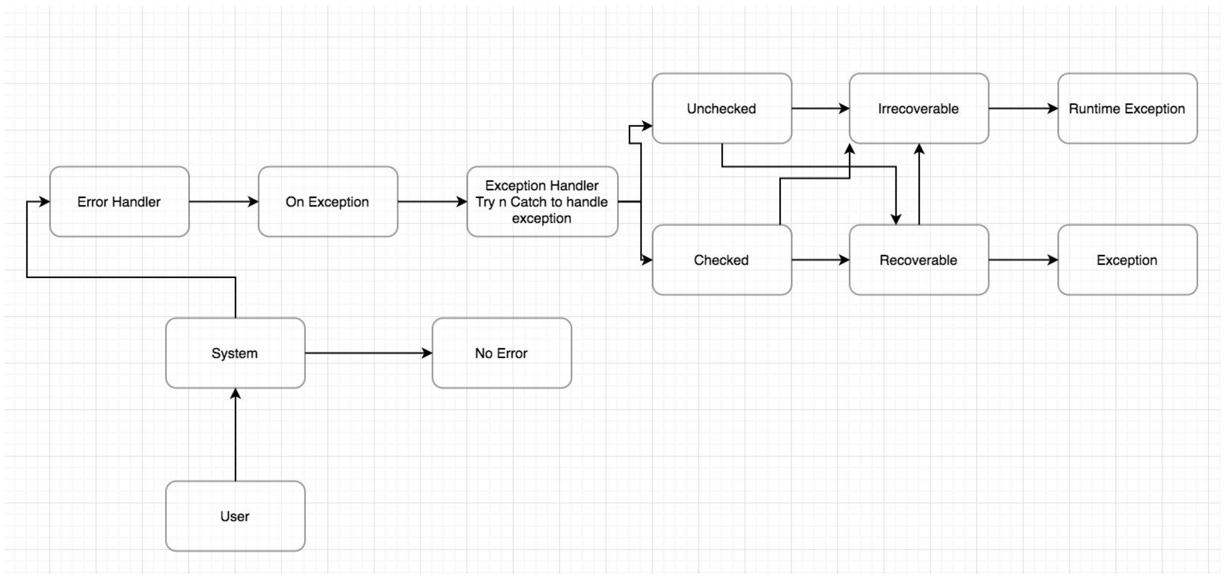
4.5.2.C Element Interfaces

UI - The interface provides client the view of data and options to choose the service

4.5.2.D Element Behaviour

Validation - A lot of validation steps will be executed during this process such as parameter checking, model state checks, etc.
Exception Handler: This component will raise an exception with a prompt at different layers. For example, an exception will be thrown from the database management system when there is a primary key violation.

4.5.3 Context Diagram



4.5.4 Variability Guide

There is no variability to discuss in this view.

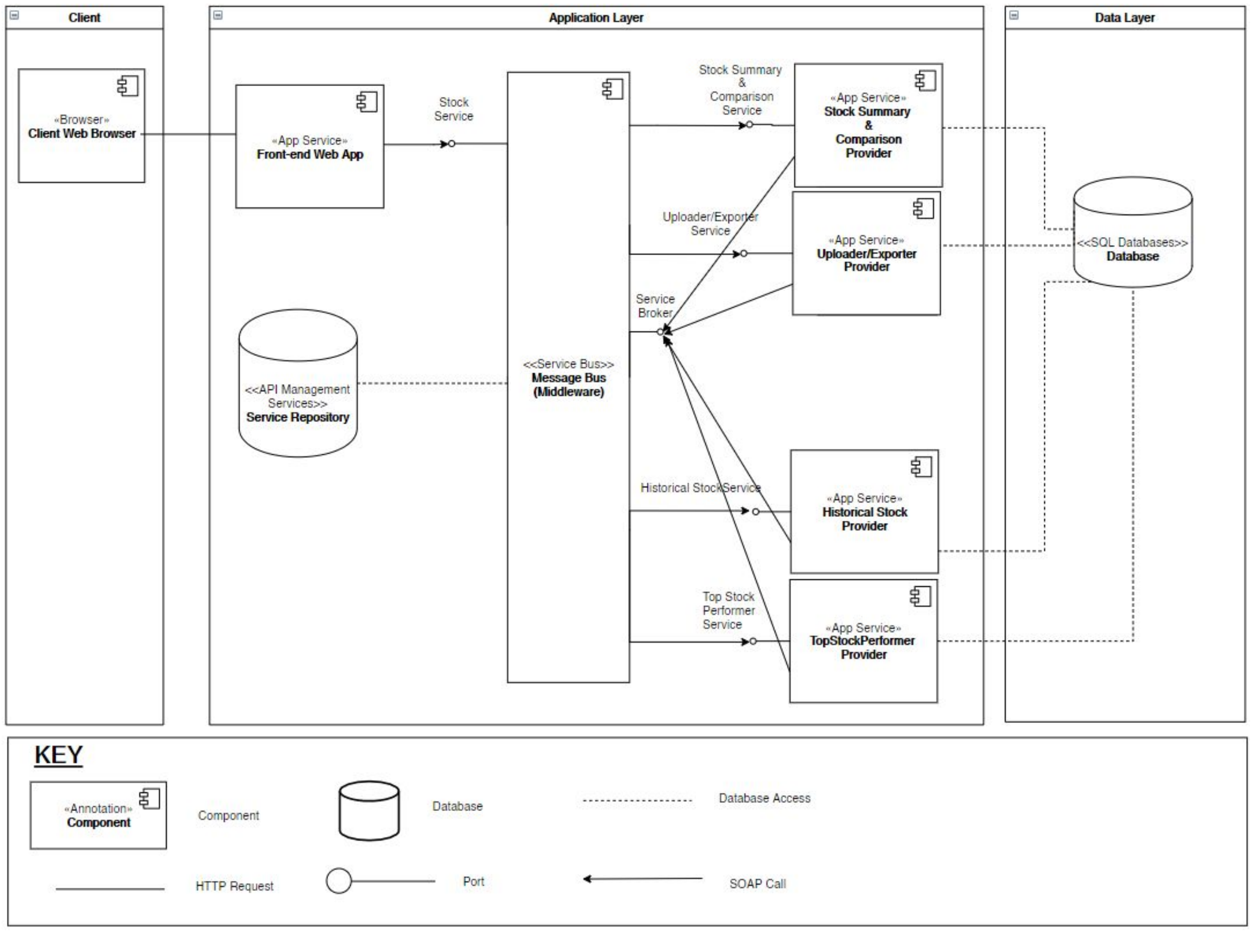
4.5.5 Rationale

This exception handling design help to handle the situation when an unexpected event happens and ensure the system won’t crash. Example will be an abnormal input file upload by users, the console will display a message that prompts the user the error and execute catch block.

This design provides fault tolerance of the application as well as provides systematic treatment of exceptions and helps balance quality requirements during the development stage. Distinguishing between checked exceptions (exceptions at compile time) and unchecked exceptions (exceptions at runtime) also enhance the robustness of the system.

4.6 Combined View

4.6.1 Primary Presentation


Client
«Browser» Client Web Browser
Application Layer
«App Service» Front-end Web App
Stock Service
<<Service Bus>> Message Bus (Middleware)
<<API Management Services>> Service Repository
Stock Summary & Comparison Service
«App Service» Stock Summary & Comparison Provider
Uploader/Exporter Service
«App Service» Uploader/Exporter Provider
Service Broker
Historical Stock Service
«App Service» Historical Stock Provider
Top Stock Performer Service
«App Service» TopStockPerformer Provider
Data Layer
<<SQL Databases>> Database
KEY
«Annotation» Component
Component
Database
Database Access
HTTP Request
Port
SOAP Call

4.6.2 Element Catalog

4.6.2.A Elements and Their Properties

User Interface Layer - Handles all actions that the user performs in the web application. Security concerns are mainly focused on the user's account and making sure they are authenticated when making actions that communicate with the API.

Application Layer - Back-end application logic that receives requests from the front-end. Also returns login tokens or data back to the user interface. For logging

Data Access Layer - Communicates with the databases to retrieve or store desired data.

Databases - Stores either data about user credentials or stock data.

App Services - These elements represent the services provided by the application.

Message Bus - The message bus handles communication between the client API and the service providers by routing client requests to available service providers.

4.6.2.B Relations and their Properties

Publish/Subscribe - Components will be publishing messages with certain topics it will fall under. Where other components will be listening/subscribed to specific topics to grab messages off the middleware.
Database Access - Access relations represent a component that has access to another component for function.
HTTP - Requests will have to follow http request standards and the requests will have to adhere to the implementation of the application.

4.6.2.C Element Interfaces

UI - The interface provides client with a view of the application, including its data and services

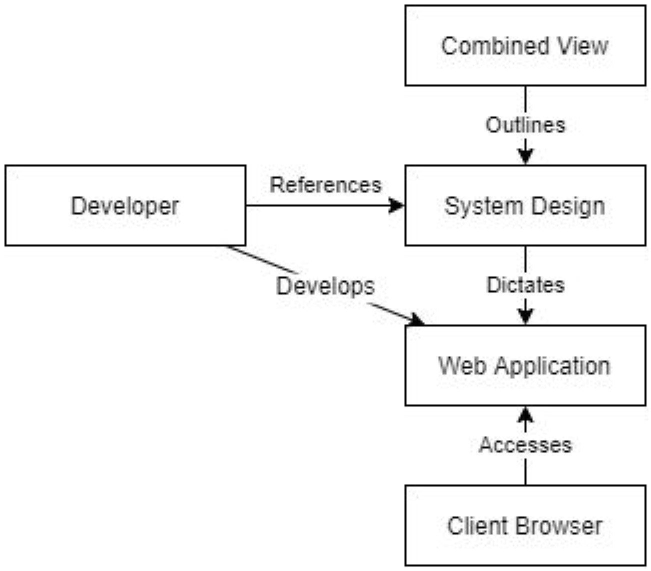
Service Broker - The service broker represents the interface between the services provided and the message bus. The purpose of the service broker is to unify the services in terms of implementation language.

Database Interface - Services access databases through the database interface available to C#.

4.6.2.D Element Behaviour

The message bus acts as the intermediary between the client and service providers as prescribed by the service registry directory. The message bus will be implemented using Azure's Service bus where the other components will be subscribed and listening for messages that adhere to their service.

4.6.3 Context Diagram



The combined view outlines and abstracts details in the system design that is later used by developers while developing the web application.

4.6.4 Variability Guide

The database access interface allows variability in the type of database that is used. For example, if the database system were to change from SQL to MongoDB, the data layer can be changed without affecting the application layer. There can also be variability in the services used by the message bus as they can be added or removed at will without affecting the interface.

4.6.5 Rationale

The combined view highlights the service-oriented architecture of the system and its various components that play key roles in the system. In the application layer, multiple service providers are available to the front-end web app (client API) that is mediated by the message bus. Communications between many clients and many services must be mediated smoothly, so a publish/subscribe pattern is implemented into the message bus. To allow for dynamic addition and revisions of services, a service registry directory is designed to present the message bus with a list of available services and their locations.

5. Mapping Between Views

View	**Relationship**	**Associated View**	**Description**
Module	implements	Components and Connectors	Modules in the module view implement the components and connectors detailed in the components and connectors view.
Allocation	deploys	Module	Allocation view is used to help associate what each environment the software module will run on.
Error-handling	component-of	Module	The error-handling view of the system details the protocol for handling errors. The protocols are implemented in modules found in the module view.
Module, Allocation, Components and Connectors	component-of	Combined	The combined view pieces together the three system views to form a cohesive system design.

6. Rationale

The goal of the system is to provide clients with services that present informative stock data to users. The application is intended to be deployed through a web application. Services provided by the application should also be easily updated and iterated upon after release to better tailor the application to client needs. Service provided should also be highly uptime for clients.

With these requirements in consideration, the system adopts a service-oriented architecture that also uses a publish-subscribe pattern. The service-oriented architecture allows for services to be updated and added during runtime that minimizes disruptions to client activity. The architecture also implements an intermediary message bus between the client and service providers that allows for the system to dynamically provide clients with services from available service providers while hiding unavailable services, improving the availability of the system.

The publish-subscribe pattern is implemented into the service-oriented architecture through the message bus. The bus acts as a connector between the client API and the service providers by receiving the client's published messages and pushing the requests to the corresponding service provider.

7.Results

7.1 System Architecture and Implementation

Service-oriented Architecture

There are two major layers to the architecture: the web application and the functions. The web application represents the website presented to system users and is implemented with HTML and C# code and hosted either through Microsoft Azure's web application or hosted locally. The functions represent the services offered by the system and are implemented and hosted by Azure Function Apps. These functions are published under Azure's API Management Service (discussed belong) and are independent of the web application.

Service Repository

The service repository is implemented using Azure's API Management Service to manage all the published and unpublished (under development) services including their metadata. During this development, services were slowly added as they became finished causing our service repo to grow and display more services to the rest of the team. This gives the flexibility for people utilizing the service (such as our frontend web app) to view what's available and details regarding sending data. So now other systems/services have the capability of searching up all the available services.

There exists a service to subscribe to specific applications or services to see if there are any updates or changes. This could have been implemented into our web application to dynamically enable or disable new services if we so see fit. Unfortunately, we did not have the time to fully utilize this important feature in an SOA system.

7.2 System Testing and Test Cases

System test cases and how to perform the tests are listed in an attached excel file.

8.Limitations

8.1 Azure Function Quotas

With the given free service plan on Azure, there were some limitations on the available resources and a strict quota. Oftentimes the system would shut down because the team hit the quota set for free users, which severely hindered the team’s progress.

8.2 Database

The database was intended to be a SQL server hosted in Azure but during development of the project, Microsoft did not allow new SQL databases to be created due to resourcing issues. As a result, we had to compromise by alternatively using Azure Table Storage, a NoSQL database. Azure Tables came with its own limitations like not being able to perform group queries or skip queries to query the data in batches. This ends up hindering performance of querying and requiring some extra pre-processing on the data.

8.3 Service Bus

The service bus acted as our messaging middleware for communicating between services. They provide a handy feature called ‘topics’ to help services subscribe to specific topics. This helps streamline and create messages much more simpler for the pub-sub model in an SOA. However, this feature was only available for premium plans (we only have free student plan) so we were not able to make use of this handy feature.

9. Directory

9.1 References

Bass, L., Kazman, R., & Clements, P. (2012). *Software Architecture in Practice, Third Edition*. Addison-Wesley Professional.

Bass, L., Weber, I. M., & Zhu, L. (2015). *DevOps: a software architects perspective*. New York: Addison-Wesley.

Hohpe, Gregor. (2019). Enterprise Integration Pattern. Retrieved March 11, 2020, from https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html

Software Architecture in Practice. (n.d.). Retrieved from https://sites.google.com/site/softwarearchitectureinpractice/home

9.2 Acronyms

EF	Entity Framework
HTTP	HyperText Transfer Protocol
Pub/Sub	Publisher Subscriber
REST	Representational state transfer
SOA	Service-oriented Architecture