REPORT ENTERPRISE COMPUTING - Share Trader Viewer

40212064 Alessio Williams Gava- 2019/2020

Introduction

The approach attempted is by utilizing combined techniques of Component Based Software Engineering, applied to a Service Oriented Architecture to build a Shares Tracker application.

Scope

The aim of this report is to describe the share trader application built by analyzing the component utilized and the Service Oriented Architecture applied. An overview of the technologies used, and the architecture utilized is offered. Three components utilized are presented along with a detailed description and the reasoning behind why they were chosen over other ones. Adaptations to external components are listed and discussed. A final evaluation of the benefits gained from utilizing component based software engineering and SOA is presented.

The application built is intended as a tracker service as requested in the coursework specification, not actual share trading functionalities are happening. Shares can be only viewed and tracked.

Technologies chosen

Backend: Asp .Net Web Api utilized to create the backbone for the Services where Api nodes controllers are set up to connect to services. SQL databases for each service, but since the project has been only created in a local environment, a single server (localhost) was utilized. Even though the server is shared for all the services, it has been taken care to not create dependencies from database-services. In fact, when designing the architecture, it was pretended that each database was not accessible from other services, so MSOA techniques has been required to request data that is present in other databases.

Frontend: Either desktop graphical interface as Java Beans or Windows Presentation Foundation have been taken into account, but after a research it was decided to utilize a web interface to provide graphical capabilities to the application because it would be providing the application to more types of devices, since only a browser would be needed to access the application. Instead, if it were focusing on creating a desktop application, a smaller range of users would be reached because for example mobile user would not be able to access it without a mobile application. It must be noted that creating a server oriented architecture, it is possible to extend the project to provide services to more types of devices just by creating new client side interfaces and keeping the same server side application running.

MSOA Design & Architecture

Following the project specification requirements, an architecture of separated services was drawn and the project was set up to depict the following figure.

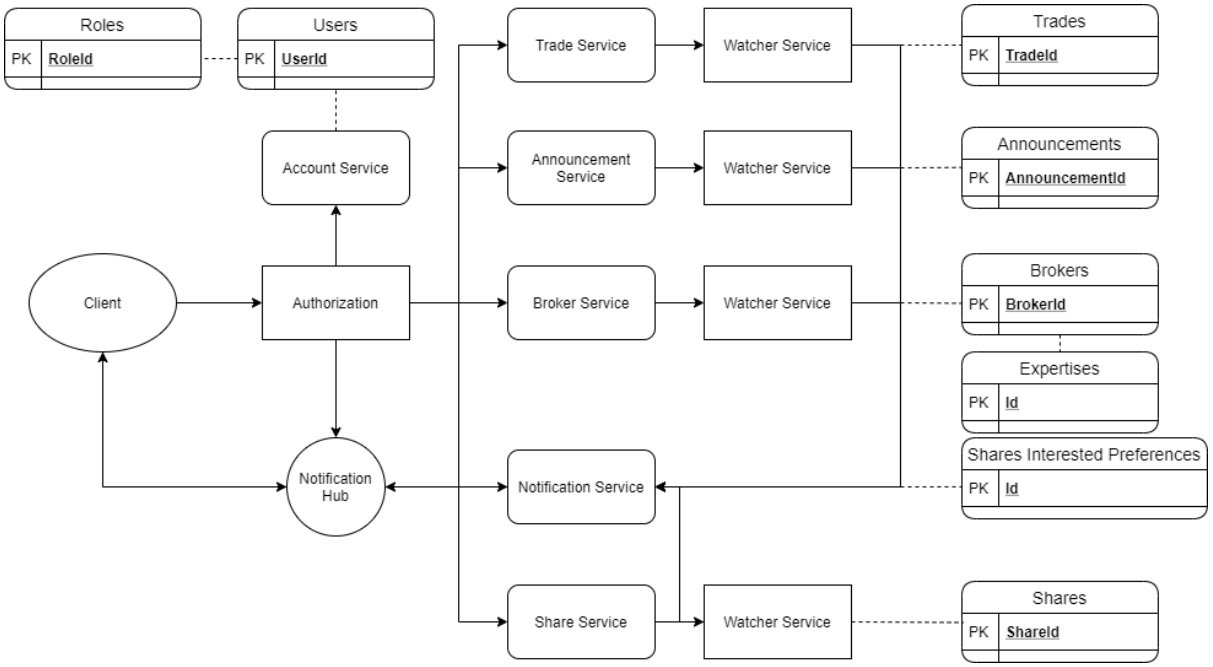

Figure 1 Project SOAchitecture

When a client connects to the application server, RESTFUL Api communications are utilized to contact services. JSON format is employed to transfer data to and from services, and HTTPS protocol is applied to specify the type of request from the clients. It has been mainly utilized GET, POST, PUT and DELETE requests.

A main Authorization component has been adopted to check for the validity of the information inputted from the user. This entity is wrapping the whole application, but it can be specified where not to use it, so specific controller end points could be set as being reachable for not logged users. In fact, the login and register services (contained in the Authorization) were set like that.

The project has been designed as having each service theoretically distributed, so is not possible to access directly the database of another service. This was reflected on the creation of services that needed to access more data properties of a certain entity that were stored in a different service. For example, a join table is created between the User Service and the interest share service, where the “Shares Interested” are stored with the “userID” that is interested in them. The relation between the two table is not set in SQL terms using Foreign Keys between the two services because they might be stored in different databases. Instead the link is provided as Api requests. The “Interest” table will query the User “GetUserID” api to retrieve the current “UserID” that is trying to add a new interest in the share.

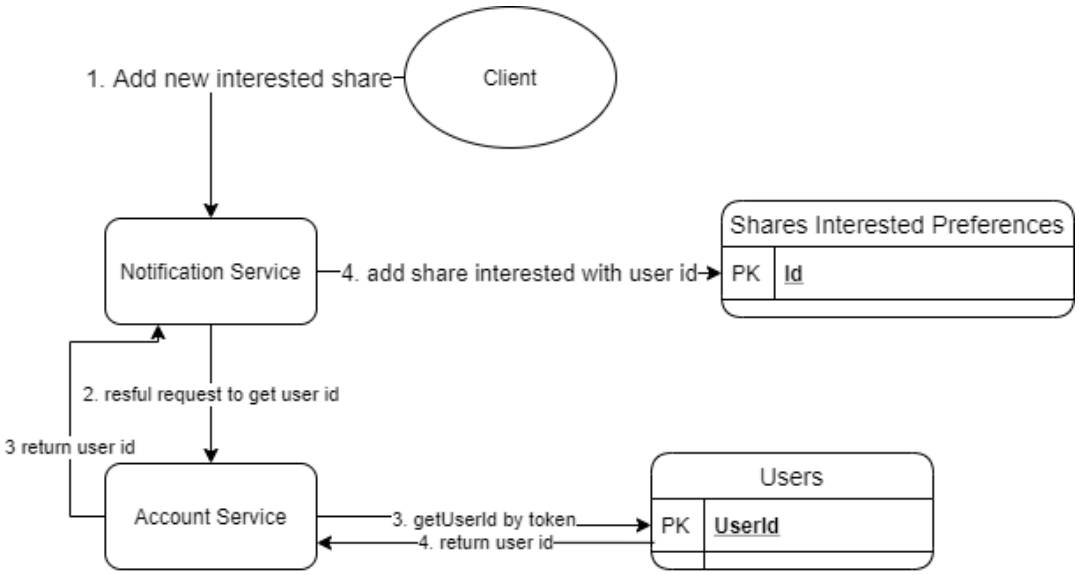

Figure 2 Example retrieve data from other services

A HTTP Client is created in the Share service and it utilized during the whole service life as pointed out from the Microsoft documentation (Microsoft), where is highly recommended to reutilize the client for many Api calls, instead of creating a new one because it could exhaust the number of available Sockets.

Each service has its own tables that contains only data that is useful for their parent service. All the services follow a similar base pattern in which they are accessible by a controller end point. Based on the path of the end point connection string and the HTTP request type, a specific function is called on the service. Data can also be passed from the client side to the controller. The default type of data is XML format, but it was preferred to switch it to JSON format because it is more performant, easy to create and manipulate and is very well handled from JavaScript (w3schools)(guru99) and ASP NET on the backend. Below is presented a figure showing more in detail the links between different entities in the project.

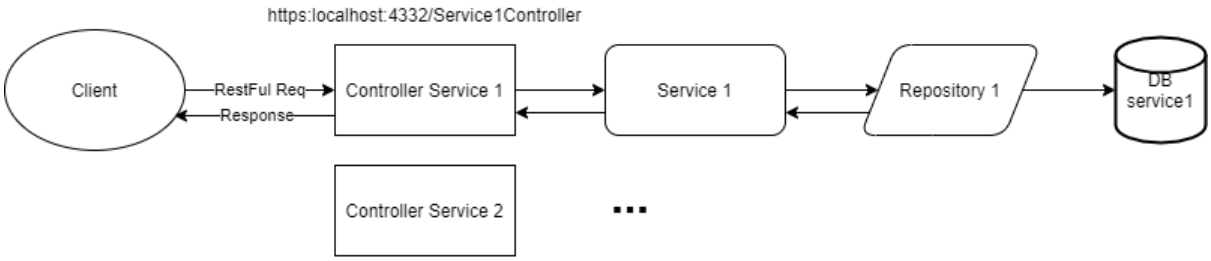

Figure 3 More in depth controller-services architecture

The service can then process the data received and then it calls a function in the repository which will query the related database residing in the service. If any data is to be sent back, it will be passed back up to the controller which will contact the calling client, returning any response data needed.

As it can be seen, a watcher service has been re-utilized throughout the application. Its goal is to check for any changes in its parent service and then report back to the notification service with the data modified. The notification service is in charge of analyzing the data received from the watcher services and then to decide which users should be contacted, utilizing the filters preferences set in the table “Shares Interested Preferences” that reside inside the Notification Service.

The notifications are being sent to clients in real time by utilizing an external component, SignalR (Microsoft), applying a design pattern in which a notification Hub is the central point between the Notification Service and the clients. This Hub is used to both receive and send data to the users, so it is possible to set up scenarios in which the notification can receive success or failure response messages from the notified users. This could lead to future complex situations of storing notifications if a user was not online or if the notification is wanted to be shown also on a different type of device.

User Interface

As mentioned in the introduction, the User Interface has been done as a web application.

The client side is composed by multiple pages connected by a top banner which lists links to other pages. Home page is the login as per requirement, if a user is not logged in, it will not be shown anything else than the login-page. Register functionality is contained together with the login page, so user can both register and login in the same page.

Connection to the web apis controllers are done by ajax requests tuned to use the correct HTTP request type and message body. Since the project has been written to be ran locally, all the requests points to the same base url: “https://localhost:<port>/”. In case the project is to be ported to a live server, the addresses should be modified accordingly, which would be very straightforward because the url variable used to store the address is declared in a single JavaScript file that is imported from any pages that need to contact the server. The same could be said in case a load balancer would be needed in case the application needs to scale up.

Shares are listed on a single page. A table is used to show all the available shares with their most important information. They are retrieved from the web api and dynamically loaded and they are also in real time visually updated in case of database update. On the same page is possible to add interest of specific share, setting the share, its minimum and maximum price to be notified of.

A single page is dedicated to the visualization of the interested share, so a user can easily view which shares have been set as interested. It is also possible to remove them by pressing a button which will send a web api delete request with the shareid as body.

Similarly, a page is created to view all the available brokers in the system. This page is also equipped with a real time updater for brokers information. A registered user can request to be recommended a broker that matches his shares types. The request will contact the Recommend Service residing in the server which takes into account also the brokers' stored quality of work.

Share trades are shown on a different page and can be filtered by user input values. Any of the values inputted are optional, so it is possible to filter by a single field or by any number of filters the user inputs. The client side will only pass to the backend only valid values (for example no empty fields are passed).

Finally, announcements have their own page in which are listed by share id.

As it can be seen there are present some similar front end pages, for example a recurring patter is the viewing of all elements of a certain type. These pages were created from a single component that has been reused and adapted to show a different type of data, making it easier and faster to create the web page.

The notifications sent to the user are based on end points set on the client-side application. As long as a user is logged in and with the browser open, he will get notified. Different types of notification are sent, but it was preferred to inform the user of the notification but not give any information about it. The user will have to follow the link to open the relative page that will show the entities that were modified (for example if an announcement is created, the user will be sent to the announcement page).

Components

Each service identified, is responsible for its own data and functionality. It can't modify or access data of other services directly. Applying a component-based SE, it was thought that each service could be viewed as a separate component that could be attached to the application. Research of components was then focused on finding reusable services to easily add to the project.

Components mining

Initially, it has been performed a requirement analysis listing all the required features that the project needs to satisfy. From the found requirements, it has been created a list of components that could be used to achieve the wanted features. Successively, a research has been performed to find the most common components in the list to be reused in the application. Some features of the project were classified as being too specific to be found in open source projects, so they have been created from scratch taking into account the time needed to code them.

Component 1

Authentication component

The first component being considered for reutilization was the log in and register user component. It was thought that since it is such a common functionality that is present in any website/webapplication in which a user will have an active role, it must be available to be easily deployed.

The search has been focused on the web api and C# language. Two complete and fully functional components have been discovered. The first is created directly from Microsoft and is provided under a package called Identity module. Additionally, is added as a framework in the Visual Studio IDE, so it is possible to select it and it will be automatically installed in the project, along with the relative Apis controllers and Database.

The second component is published on GitHub by Jason Watmore - https://github.com/cornflourblue/aspnet-core-3-registration-login-api

In which a clean registration and log in api has been provided as a service class. This component matches exactly the architecture of the project being created, so it would be easily integrated just by injecting the package as a User service. On the other hand, the component proposed by Microsoft will create a project slightly dissimilar which will require a few modifications on the architecture. It was chosen the Microsoft approach, even though is more cryptic to understand, it is thought to be more performant and secure. Additionally, it performs more functionalities, for example not only log in and register, but is possible to modify the password, assign roles to users, possibly utilize external login (FB, Twitter, etc..).

Nonetheless, both the approaches are very interesting and suitable for the current project, because they implement exactly the authentication functionalities as Api so they can easily be set as services, ensuring that the MSOA architecture pattern is respected.

The Microsoft Identity component, utilizes a token pattern in which once a user has been authenticated, a token is generated and passed back to the client. The token will need to be stored from the client and can be added to the RESTful requests as a authentication bearer token in the head of the request. Following this pattern, the authorization component can check that the user is in possess of a valid token and any requests that need to authenticate a user will be satisfied. This component provides the customizability of the token duration, it can be a session token, or it can be valid for a length of time.

Additionally, the component provided with a useful functionality of register different types of roles. It was utilized to create admin roles and normal users. An admin can access restricted controller end points, so the application can set limitations of access to possible disruptive functionalities (for example deleting trades, brokers).

An admin can only be added from another admin profile, and the first admin created cannot be done though the invocation of controller end point, but only by directly manipulating the Database.

Provides Interface:

1. LogIn
    [PreCond]
    User logging in, exists
    [PostCond]
    hash stored password == hash password provided AND
    New token created == hash (login session + user specific fields)

2. Register
    [PreCond]
    User name available
    [PostCond]
    New user created AND password hash stored == password hash provided AND
    No missing not optional fields in user table

3. Register Admin/Role
    [PreCond]
    User requesting role == 'Admin'
    [PostCond]
    User role registered == 'Admin'

4. Delete User
    [PreCond]
    User requesting role == 'Admin' AND
    User deleting exists
    [PostCond]
    User deleted doesn't exist

5. Check user token
    [PreCond]
    token is not empty
    [PostCond]
    token argument == token in db AND
    token is not expired

Safety condition: Admin can only be added from other admin user. HTTP DELETE end points only accessible from admin user
Time requirement: token is valid for developer specified time frame.
Space requirement: User table and User Roles are initialized with needed columns
Suitable domain: Authentication
Implementation language: .Net languages, C#
Platform requirement: OWIN middleware

Component 1 adaptation

Some modifications to the Identity model from Microsoft were needed to be implemented, but only on an high abstraction level by extending some of the classes provided. No

modification in the inner working code was needed to be performed, otherwise it would have been chosen the component from Jason Watmore because the modifications would have been easier to apply (simpler code) and it was exposing the whole code without being shipped as a single packaged component.

Roles were needed to be added to a table in which all the available were listed.

Controller end points were created so specific service functionalities were directly available to be executed from a client side.

Component 2

Watcher service

This component is part of a larger requirement which is built utilizing more components: notification of the user upon certain events. By analyzing this requirement, it was listed two different methods in achieving it. In both of them, a certain type of notification is sent to the user after an action triggered by some event. The notification could be either in real time or stored in the database and then shown to the user when a page is loaded. It was thought that this type of application is more incline to the adoption of real time notification because the system being monitored is very changeable, having share prices that can continuously change over the course of the day.

The watcher service is needed to perform the notification in real time. After any changes on a specific table/database, an event is raised and the watcher will be notified of the modification. At this point, it can notify whichever entity needs to be notified. The main component of the watcher service is then a form of table watcher applied to the SQL Database.

Component mining

The main reusable component found is called SqlDependency and is contained in the Net Framework under the SqlClient library and “represents a query notification dependency between an application and an instance of SQL Server” (stevestein). By analyzing the documentation, it was noticed that is requested some set up to make it work with the share trader project because is a general-purpose SQL dependency library. A few disadvantages were encountered in which it was not possible to easily achieve a watcher and return the modified data just by utilizing the library alone.

Based on it, two extended implementations were found that managed to provide services very similar to the needed watcher over a specific database table: ServiceBrokerListener (dyatchenko n.d.) and SqlTableDependency (Bianco [2017] 2020). Both of the presented components were able to provide with the needed functionality, but SqlTableDependency was found to be more comprehensive in the documentation and example of use cases.

Additionally, it was built in a way that facilitated the integration with other services that would notify users, following a similar pattern adopted in the application being built.

Provides Interface:

1. Receive table change
    [PreCond]
    table watching not empty AND
    table and Database connection is valid
    [PostCond]
    received change == last added/modified table row

2. Notify of change
    [PreCond]
    url and http client to notify are set and not empty AND
    modified row data is not null
    [PostCond]
    Success returned from http request to notification url

Safety Condition: Database referring to is running and valid.
Time requirement: components needs enough time to contact notifier after change is registered. Otherwise added to the queue
Space requirement: queue of changes is limited
Suitable domain: SQL, backend database
Implementation language: C#
Platform requirement: Microsoft SQL Server 2012, NET Framework 4.5.1

Component adaptation

No adaptation has been needed with this component, but it was needed customization of parameters to provide this service in a more general way to the whole share trader application. For this reason, a wrapper (Watcher service) has been created around this object library that would create a parameterizable reusable object which can be added to any existing service and provide them with a customizable watcher functionality.

```
public class Watcher<P>
{
        private string _connString;
        private string _tableName;
        private SqlTableDependency<P> _dependency
        public Watcher(string connectionString, string urlNotificationApi, string
        tableName) {}
        private void dependency_OnChange(object sender, RecordChangedEventArgs<P> e)
         {//contact api url specified in constructor}
}
```

Following this pattern, it was created a very flexible way to get informed when a specific table was modified and how to respond to these changes.

Component 3

DateRangePicker

A filter functionality has been provided in which the shares can be filtered based on multiple fields, one of which is date. This component is part of the client-side since the date has to be chosen from the user which must be validated and correctly formatted.

Component mining

The frontend, as mentioned earlier, was built as a web application, so it could have been utilized a date HTML input and CSS styling to make it look nice, which can be easily set it up. Being such a common component on a web page, many different components are available so it was decided to utilize a more complex component which would provide with more fine grained functionalities as format checker, possible time input and importantly a calendar-view that would help the user to choose the dates.

Amongst the wide variety of available graphical component, it was chosen DateRangePicker because of high customizability and particularly because it provides with intuitive range selector for minimum date and maximum date, which is needed to implement the date filter. Additionally, it is included as an optional library in the NET Framework Nuget package manager, so installation and maintenance are managed directly from the tool.

Interface:

1. Parse data to format specified
    [PreCond]
    specified format stored in the datapicker object (can be default)
    [PostCond]
    Parse(data, format) == outputted data

2. Show selected data
    [PreCond]
    Data range selected from user
    [PostCond]
    div showing the selected data == user selected data

3. Show calendar
    [PreCond]
    calendar current show state == not showing AND
    calendar property != hidden
    [PostCond]
    calendar current show state == showing

4. Validate data
    [PreCond]

Date inputted != null or empty
[PostCond]
date > min date AND date < min date AND
date number of characters == number characters in current format AND
separator character in date is amongst accepted separators AND
day in month < max number of days in month AND
month > 0 and month < 13

Safety Condition: date is conform to the format specified and returns true from validate date function
Time requirement: None
Space requirement: NA
Suitable domain: web application
Implementation language: HTML, JS, Bootstrap framework (CSS)
Platform requirements: Jquery.js, moment.js

Component adaptation

Graphically, the component fits perfectly on the client-side GUI, it is also automatically resizable. Related to the code logic, it was needed to set the calendar to output data format and data symbols that would match the one used in the filter service in the backend because the two environments utilizes different technologies to parse the dates. Additionally, it was discovered that it if a user would clear the datepicker date, internally was not cleared the date from the datepicker object, but only displaying an empty string on the GUI. This was problematic because the GUI filter functionality would contact the filter Service utilizing the datepicker object data which was never completely cleared. For this reason, an adaptation of the GUI has been applied in order to not pass to the server erroneous dates. This adaptation was performed on the client side and not on the component.

Additional reused components:

Pop up notification on User Interface: it has been used an external component to show user with customized notifications pop up, displaying success and failure.

User Service from the practical code: most of the services that provide query of database functionality, have been created following the MSOA tutorial in which is possible to add new entities, modify them, delete single, query all and query a single element. It was very straightforward because the architecture is the same used in the ShareTrader; it was needed to apply small changes, mostly were only related to the different name of the class.

Real time notification: it has been utilized external library Signalr to provide with the ability to notify single user in real time. It also possible to create groups of users to broadcast same messages. This component has been used to create the Notification Service. No adaptation

was needed, but it was required to set up the server side notification hub and the relative connection on the client side, so the right type of notification can be sent to the user. A slight modification was required because the shares interests are stored along with user id, instead the library would notify users by user-name, so it was needed to configure the component to notify them by user id.

Testing

Testing has been applied at each stages of the application construction. Initially, only node ends were created for the distinct services put in place, but it was not possible to directly test their validity because the Graphical User Interface was created as last element of the application. For this reason, it has been utilized the tool 'Postman' which can create HTTP and HTTPS requests to specific end points, making it possible to query the Web Api application with JSON communication.

The reused component functionalities were encapsulated in the services which are then contained in the controllers that have been coded specifically for this application. So, all the controllers were needed to be tested in order to check that they were reachable, behaving as expected and were communicating correctly with the inner services.

The reused components did not need to be unit tested, because they have been already tested from their respective authors. They were thought as being black-box components, so only integration testing was needed, where the unit models are added one by one and their interfaces between models are tested. This type of test should find any cases where the components are utilizing mismatching type of communications or data format. Additionally, it can be useful to understand any limitations that components might have. An example that was encountered during the development of the application was the integration between the datepicker component (an external component that displays a datepicker to choose a date utilizing a calendar in the GUI) and the GUI filter functionality to query specific shares. By testing multiple use cases, it was discovered a mismatch on the implementation of the external component.

Integration testing has been carried out for the external imported components by testing that the result of common use cases were as the expected ones. Additionally, it was tested many edge use cases in which the inputted data was conform to the services expected type, but it might have been an unusual value. For example, testing negative prices, empty dates. This is a type of white box testing because the it was known the expected type of data.

Due to time constraints no unit testing were create or automatic testing. Testing was done at each stages, but manually with the help of Postman.

In case of external component, they were tested as being black boxes, not knowing the exact type of data needed and their inner workings.

Utilizing a local server, it was not possible perform a load test on the application.

Evaluation System

The system is thought that achieved a good amount of theoretical MSOA benefits. For example, having built the system with the concept of keeping services divided and not dependent on each other, it will be possible to create a distributed system when live-deployed. In case is not wanted to build a distributed project, the whole system could be stored in the same server without any additional modifications.

Flexibility has been shown to be achieved, in fact by using RESTful communications between clients and services, it was created an environment that is propense to be extended with no major modifications to the existing code. Each service is proprietary of their own code and don't need to rely on other's code. Modifications on a service will not affect others, as long as the data outputted is of the same format required from other services or clients. For this reason, adding new services will be as easy as plugging a new component to the system; it will mostly be needed to configurate the correct controllers' addresses.

Having clear separation from front end and backend environment, it was achieved flexible future improvement of the platform. For example, adding new and different clients that would communicate to the services would only need the creation of the client itself. In fact, it would be possible to add new interfaces as: desktop application, mobile application. Since the connection to the backend is not managed by specific technologies, but just by datatype driven, using JSON messages, it was avoided high coupling between the environments.

Component based software engineering has been exploited in order to create needed functionalities to satisfy the project requirements. When creating or importing a new component, it was analyzed the pros and cons of doing so. In case that the component mined were not fitting properly in the project architecture, it was evaluated the extent of a project architecture modification and all the services that would be affected by it. A second option that was taken into account, was the modification of the imported component. Not always this choice is available because the component could be not locally available and the source code is not adaptable. As a last resource, it is considered to build a component from scratch considering the predicted time needed to code it and the possible difficulties encountered. If the component is not complex, this was the preferred way.

It can be confidently said, that by reutilizing the chosen components, the cost and time taken to build the project was highly diminished. For example, the Authorization component utilized didn't need a high degree of adaptation and provided to the project a very complex feature. It can be easily seen the benefits that were achieved by introducing this component. Another example is the real time notification system utilized, which greatly helped in producing the user announcements.

Nonetheless, it must be noted that these components needed to be set up and configured which meant to be reading their documentation. This step can be quite time consuming, but fortunately it was not encountered much difficulty in importing them into the project. If

they will be reused in a future project, it will be even faster to work with them because now is known how to utilize them.

Another point to be mentioned is that when scouting for possible components for a single feature, it was considered the future maintenance. Components built from more reliable and presumably long-lasting companies/developers were preferred, because there are more chances they will be adapted to possible changes of the technologies being used.

A possible drawback of the chosen architecture is the high distribution of data between different services. In case many calls are needed to be done to a service, is possible that it would create a bottleneck because a HTTP request is definitely slower than direct access to the database. If this is the case, it would be better to join multiple services that needs to request much data, so it would be faster but more coupled.

Additionally, since the system was created only on a localhost, it is possible that some aspects of a distributed system were overlooked and it would require a few modifications to properly perform in a distributed manner.

Reflection

Hardest part was the clear separation in services that encompass single activities. Not always simple to work out a correct division, because it might become evident only later that a service is heavily dependent on another service and would be better as being a whole single service.

When building a SOA application, thoughts must be given to the future scalability, and it can be quite difficult to envision possible situations of bottlenecks. Additionally, building a system taking care of future performances is very time consuming compared to build it just to be performant at the current time.

The proposed architecture is not fully compliant with the MSOA standards because services should be discovered by utilizing a registry service, so it is simple to deploy a service in a different server container. In this project, the address to use to connect to the services is hard coded on the client code, which is a simplification of the MSOA standard. The project was not thought as being a fully complaint MSOA because it would have needed much more time and resources to be done.

By not having an API Gateway, the services outputs were needed to be modified slightly to be able to be displayed correctly on the client side. A service should not worry about these problems, in fact an Api Gateway could even call different services and bundle their result together in case of a functionality that requires data from different services. There can be seen a few examples in the application where an API Gateway would have been useful, so services didn’t have to contact each other to request data which is not worked upon, but is only clustered together for better client views (example: request the share Symbol from the share service because is not nice to show a share ID on the front end application).

Future work

As future work, it might be worth to invest time on the improvement of the notification system. At the moment a user will only be notified if is online and logged in. In case the browser is not currently displaying a page that loaded the shared script containing the proxy functions needed to be connected to the Notification Hub, it will not be notified. It could be interesting to check if a user is logged in and received the notification, otherwise store the notification to be shown next time logs in the website.

Secondly, the number of shares stored in the database is not very high, so it is clean and easy to view them all. Once their number will rise, there is only a single page showing them all, so it could get very messy. It would be better to find a page splitter component, where only a certain number of shares are stored per page.

An API Gateway could be provided by extending the Authentication service, since it is a very central and entry point of the backend application.

References

Bianco, Christian Del. [2017] 2020. *SqlTableDependency (https://github.com/christiandelbianco/monitor-table-change-with-mvc-signalR-jquery-sqltabledependency-example)*.

dyatchenko. ServiceBrokerListener https://github.com/dyatchenko/ServiceBrokerListener.

guru99. n.d. “JSON vs XML: What’s the Difference?” Retrieved April 8, 2020 (https://www.guru99.com/json-vs-xml-difference.html).

Microsoft. n.d. HttpClient Class (System.Net.Http) https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient.

Microsoft. n.d. SignalR | .NET https://dotnet.microsoft.com/apps/aspnet/signalr.

stevestein. n.d. “SqlDependency Class (System.Data.SqlClient).” Retrieved April 7, 2020 (https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldependency).

w3schools. n.d. “JSON vs XML.” Retrieved April 8, 2020 (https://www.w3schools.com/js/js_json_xml.asp).