ENTERPRISE COMPUTING

Krzysztof Dworczyk

[40340711] SET11509]

Abstract

The paper describes the process taken in developing a Store Management System. A CBSE methods are applied throughout the paper and a microservice oriented architecture patter is also followed. The use cases are presented in a diagram and the relationship are presented in an ER diagram. The component-based methods are explained and how they are applied throughout the development. An evaluation is given for the system as well the methods used. We finish on a conclusion about the whole project.

Table of Contents

1 INTRODUCTION

The purpose of this report is to document the development of a component-based web store management tool. This will allow the admins will be able to adjust inventory, prices, delivery prices and managing their staffs shift.

The focus of the project was to make sure we are implementing the component-based approach as well as the whole store application in a micro-service architecture (MSOA). This will allow for a more modular approach where each service and component can be independently worked on.

In this report I will go over the technologies used to achieve the application as well as the processes taken to mine, select, adapt, and integrate the components to create the store management page. An evaluation is provided at the end of the whole app and the methods used to finish the project.

2 SYSTEM SPECIFICATION

The specification file for this project stated the functionality which had to be implemented for the Store Management System. The document did not explicitly specify how the app needs to work but the overall function like price control, inventory stocks control, staff shift management as well as a login authorizations system which allows staff and Admins to log in with different access privileges. So, an admin should be able to change staff shift, inventory prices and stock numbers while the staff can only view the inventory and shifts. A USE-CASE diagram has been provided and can be seen in Appendix A; it lays out the functionality a user can expect when interacting with the system.

3 TOOLS AND TECHNOLOGIES

The following sections will discuss the tool and technologies used to implement the store management application. Picking the right tools can mean the difference in something taking a long time or making things easier to develop. This project will utilise the technologies listed in Table 1 as well as Table 2 shows the software used in development.

Technology	**Purpose**
C# language	User interface and logic
ASP.NET CORE	Web framework
Microsoft SQL Server	Backend Database
Swagger API(Swashbuckle)	API Gateway & Service Discovery

Table 1 - Store Management Technologies

Tool	**Purpose**
Microsoft Visual Studio 2019	Development IDE
*SQL Server Management Studio (**SSMS**)*	Database configuration tool

Table 2 - Project Development Tools

3.1 Microsoft Visual Studio, C#, ASP.NET CORE, API GATEWAY

The language of choice for this project was C#, this was due to my familiarity with the languages as well as the other chosen technologies go hand in hand with each other. The Microsoft visual studio is a powerful IDE that can be tuned to suit an array of languages if the correct add-ons and libraries are installed. For our project, we will be utilising the Web Application addons which provides the ASP.NET framework. The IDE supplies us with lots of debugging and testing tools which will come in handy in the evaluation stages of the software.

Using ASP.NET as our framework allows us to implement a web interface using a Model-View-Controller (MVC) architecture, along with the required interfaces for all the services to form a fully functional web application that communicates with the individual services. The framework automatically links the Razor View pages with the controllers and models to create pages on the fly, allowing us to reuse some of them to save time on creating them.

As we are going to be implementing the Store Management System using a microservice architecture (MSOA), each of the specified functionalities will be grouped into individual services that make sense. For inventory control like prices and stock number, we will create an InventoryService which will take care of the logic behind that. We will then need to have a way for our services to be discovered and communicate with each other. For this, we will implement an API gateway in the form of URLs which the web interface can call to retrieve data from the individual services like stock list, price or creating new items. The service discovery happens automatically due to the files generated by Swashbuckle technology, they are linked using internal URLs and start on their ports.

3.2 Microsoft SQL Server and SSMS

The E-Store Management System will require data persistence which can be done using a database. For this reason, I have gone with more Microsoft-based development suits like Microsoft SQL Server for the installation and setup of the database schemas.

The SQL Server Management Studio is a database configuration tool that was used in setting up a connection to the database as well as debugging. It allowed seeing what tables have been created along with their columns and any data that is stored in them.

Each microservice had to have its independent database to present the fact that the individual microservices can be developed on any platform or language if they are all discoverable and have a communication interface.

4 Component Mining

When undertaking Component-based Software-Engineering (**CBSE**) projects it is crucial to perform component mining to determine if any outside legacy systems, projects, or libraries are available to be used in the new project. CBSE allows developers to pull in code and functionality from other projects to save on time and even provide more functionality if the right components are found. Having access to lots of older codebases and open-source libraries can speed up the development process and allow for faster prototyping of the application.

4.1 Previous Projects

A previous project completed by me for a university module on software architectural design can provide us with lots of functionality. The project was created for a similar reason which was for an E-Store Management System where a user could log in as either staff or admins to be able to perform different tasks on the web application. The web application was created using ASP.NET and implemented an MVC architecture. Persistent data was implemented by providing a database that can be communicated with using the database connection interface as well as the CRUD functions that allow to read and write data from them.

4.1.1 User Interface

The old project was written in the same framework and language; therefore, I was able to take a lot of the already existing views of pages. These included layouts for login, Home, Order page, Products page, customer page for loyalty cards, a report page where a summarization of the products in stocks is displayed along with customer details also. This is all tied in with a shared layout of a navigation bar that allows the user to jump between any of the pages. Some pages will need to be heavily adapted or won't need at all like the report page which displayed statistics.

4.1.2 Database Connector

The project had to perform read/write operations to the database for storing customer orders as well as the details of inventory. Each time a connection to the database had to be established. As we are using the same database (SQL Server) we can also make the connection interface from the previous project and adapt it into our new application.

4.1.3 Email Alert System

An email notification system was also implemented which alerted the admin of low stocks. This was an automated call to the database which queried any items under a certain stock number and formatted it into an email. The email was implemented using a pre-existing library and sent an email to an address. This file provides us with the SMPT protocols and ports needed to create the email, using this pre-existing system will save time and the need for extensive testing.

4.2 Open-source Libraries

4.2.1 JwtAuthenticationService

An open-source authentication library was found on GitHub which implements a simple yet effective way of authentication and authorizing the users, implemented in a MSOA which perfectly fits our case. The library was written for ASP.NET so we only must adjust the interface connection and begin using the generated token to verify users' actions around the website. This is a great find as this will also save time on trying to fully test the authentication as this library already came with testing documents available.

5 Component Selection

From the previous section, I have identified the components during the mining process which will be adapted to be integrated into the final prototype of the system.

- SQL Server connection interface

- Web Pages (UI)
- Email Alert System (Low Stock)
- JwtAuthenticationService (Login System)

5.1 SQL Server Connection Interface

The database connection interface provided us with sensible commands and function that create a connection string which can be used by any of our services to access their databases. In both cases of the previous project and current project being development, they both used SQL server which will allow for easier adaptation and integration into our new application.

5.2 Web Pages(UI)

A huge portion of the previous coursework project can be salvaged. Its design facilitating a similar service of providing menus to different users to perform tasks around store management. The project had a streamlined navigation bar which allowed for quick and easy access to any part of the web application. The login and home page are the same whereas the remaining pages will have their specifying functionality. They all however perform similar actions like creating/deleting/editing of records. This means that with enough adaptation we can implement this as our UI for store management. By using this existing project interface, we can save a phenomenal amount of time as not only do we have a foundation to state on, but the structure of the application can also be based on the previous project, saving us even more development time.

5.3 Email Alert System

The previous projects specification also included a similar system in which the manager of the store receives an email notification if any of the current stocks are below a certain number. Adapting this code will be easy as we are again on the same frameworks, it provides us with a file that sets up the whole necessary connection to send an email. This includes the email credentials, as well as the port and sets the security up.

5.4 JwtAuthenticationService

As one of the specifications asked for a registration and login system, I knew I had to find a good implementation for it as it will be the first thing the user attempts. I have successfully found a library above on GitHub implementing such a system in ASP.NET as well as allowing for an MSOA patter to be followed. This made it very easy to select this and put it through an adaption process.

6 Component Adaptation

The specification states that the CBSE approaches must be implemented so in this section I will discuss how the mined and selected components are adapted to comply with the rules. As we also implemented the Micro Service-oriented Architecture, I will also discuss how the project follows those paradigms.

6.1 Component Based Software Engineering

A component-based software engineering approach focuses on the use of reusable code and libraries from legacy projects or any type of project the developer maybe has access to. If chosen

right they can provide a quick start to a project or even supply the final prototype with extra features that were adapted from the found components.

A few characteristics of components are:

- Reusability – The components should be made so that they can be reused in different areas of the program if they suit the problem
- Replaceable – components should allow being replaced with similar functioning components like a different authentication library which you want to replace your current one with.
- Not context-specific – components should not be specific to any domain, they should allow for environmental changes
- Extensible – A component should allow being extended and turned into a further different version of itself – still in component form
- Encapsulated – A component should only show its connections and ports top all communication but never actually allow interaction with its internal functions or methods
- Independent – A component should rely on no other component and be as dependencies free as possible

As I had access to the full source code of the previous project as well as the open-source libraries, I was able to make the necessary changes to my selected components to suit them to the project and comply with the CBSE standard.

6.2 Microservice Oriented Architecture (MSOA)

To further improve the overall quality and robustness of the project, a Microservice Oriented Architecture was applied throughout the development to ensure every component complied with the pattern.

The MSOA goes hand-in-hand with the already applied CBSE approach, as these both make similar use of the "reuse" components principles. Where in the MSOA case we are trying to create encapsulate each business need into a single processing unit that has access to a database. For example, a single microprocessor might take care of authenticating a user, checking customers credit card details, or processing a custom-made order. Below in Figure 1, we can see a simplified diagram of how the architecture looks. In this case, we only have one access point from the UI, and we access each processor independently depending on what we are needing to do. Once processed we will receive the response in an HTTP request form. We can also see that three are some microprocessor internal communication. These may be achieved using a database or a request queue, in the instance that a new user account is being created, then we will need to tell the User-Microservice that a new user needs to be added to the database. This however could be done from the Login-Microservice which sends out a message to other microservices with just enough data to create their user.

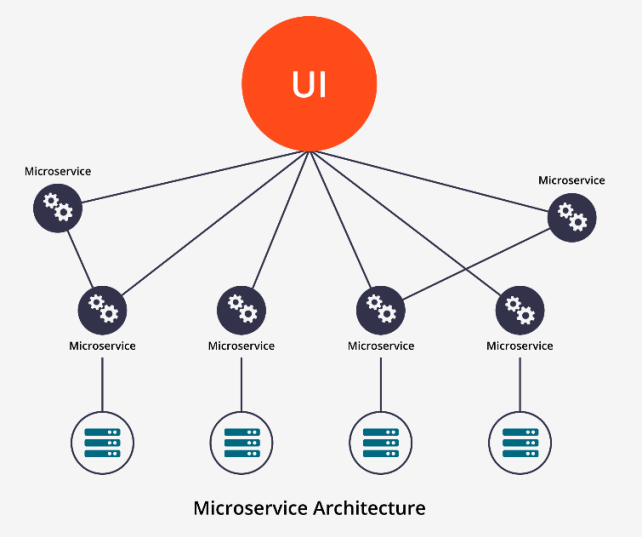

Figure 1 - Micro service Oriented Architecture example

Microservice architecture brings with it a lot of benefits mainly improved software maintainability, testability and deployability as well as a huge productivity boost to teams working on these architectures. The decoupled nature of the paradigm allows for each team to only focus on a single microservices and work on them individually, this speeds up prototyping and developers only must know a few functions and methods that encompass the service. The speed at which they can work on a small application also improves overall productivity. Improved fault isolation, once a part of the software begins t cause problems, the developers can quickly work on only that part of the program, as well as the individual services only slow themselves down as opposed to a monolithic system where if one method begins to slow down so does the entire application.

6.3 ADAPTATION PROCESS

6.3.1 SQL SERVER CONNECTION INTERFACE

The project in which the SQL server connection interface was found was old coursework which purpose was to implement the architecture of your choosing. The app was written in ASP.NET and implemented MVC for the UI and a Three-Tier Architecture for the whole software side. This meant that the connection was being made only at the Data Access layer of the application, whereas in our system we will have every service access their database. The old project provided the necessary dependencies and logic to access a SQL Database, all we had to do is take the code and refactor it to make it compatible with all the services to access and use.

6.3.2 WEB PAGES(UI)

As states previously about the old project, its form design was very similar as it implent4ed itself on the ASP.NET Core framework. The old project was implemented on an MVC architecture which is very handy in our case as we can take the views, strip them of the domain context like solution and class names, and adapt them into our Store Management software where we also use a lot of form filling. Certain logic of the controllers was also reused, the purpose of them in the old project was to access the business layers which pulled any data from the database and passed it back to the controller. I was able to again strip the files of any domain-specific context and keep the CRUD logic for retrieving data.

6.3.3 EMAIL ALERT SYSTEM

The email alert was the last component which we mined and selected for our old coursework project. The store management software also requires the implementation of an alert system that informs the store manager about low stock via email. Since our project is on the same frameworks, I was able to almost reuse the system as is expect from a few minor changes to the SMPT ports and credentials to log into an email to send out the alert.

6.3.4 JWTAUTHENTICATIONSERVICE (Login System)

Our final piece to the puzzle is the authentication system which allows us to create a login system for users and apply roles to certain accounts for access controls. The code was made available on GitHub and provide all the source code, this allowed me to extract the necessary code to create a component that allows us to plug it into our system.

7 COMPONENT INTEGRATION

Once the components have been mined, selected, and adapted we can begin the integrating them into a whole store management system. To ensure we are following the micro service-oriented architecture each of the services like LoginService, InvoentoryService or UserService are create as sand alone solution in our projects. This means that these services can be developed on any machine and then be deployed together on a container site to allow for them to communicate. Each of the services was made up of a Controller class which had the function the service was able to perform, these were the HTTP request that encapsulate each function in GET or POST requests. Once a function has been triggered, the controller will then take a model class to use a template once the SQL query brings back the results. This model can then be passed into the user application and displayed in the usual MVC patter also. Even though all the services in the project were developed in C# and ASP.net that is not to say some of them, could have been implemnted in different languages, and provided with the correct interfaces to be able to communicate with each other. Lastly, we also include the Database Context file which implements the SQL Server Connection interface and create a connection string for us to connect to the database.

The login system has been made in a similar fashion; we created a LoginService which has access to a database with all the registered users. Using the JwtAuthenticationService we can utilise user accounts and provide them with roles that restrict access to certain features on the website.

The UI elements were created using the View mined from the previous project. The controller and models were f no use but the forms inside the Razor view files gave us some logic to display certain datatypes in a better way, like the datetime format in the staff shift menu. The rest of the UI elements were added as necessary and which suited our datatypes.

8 TESTING

Testing is one of the most crucial steps in any software project. It allows us to see if all the requirements specification have been met and most importantly, do those features work as intended. Several tests and debugging solutions were deployed to try and encapsulate it all and make sure the software behaves as it should.

8.1 Unit Testing

In the unit testing, we are checking each induvial component and ensuring that we are taking in the correct data as well as outputting the correct results. As if we ensure that the software behaves correctly on a unit-to-unit basis then we can be more confident that the system will perform correctly.

Since our Web UI is using an MVC pattern we can write tester classes for the controller which will call only one HTTP request like GET or POST, we can then see the response and if they match our specification. We are also able to test the APIs by using **Postman**. With Postman we can send a request to the individual services from our browsers which is in no way tied to the project. This will allow us to fully see if each component can perform correctly.

Another area of unit testing that had to be performed was on the SQL Server queries. These are premade functions that can call the database and review certain information for us like creating new users, updating them, deleting them, or just trying to filter them by ID. The best solution to this was to simply use the SQL Server Management Studio(SSMS) which allowed us to execute SQL commands and see the results, if the results are satisfactory, we then transform the quarrier into C# and encapsulate it in a GET function.

8.2 Integration Testing

Once we have completed unit testing and have confidence that the individual components work as they should, we can begin testing a few components at a time. This will ensure that the behaviour and results between working components also produce the correct results and we do not get any unexpected errors or incompatibility issues. In our Store management system, we can do this by adding URIs to the controller UI system which allows it to communicate with services. This would reveal if the system were ready to be integrated into a whole system. The initial trouble was with the authentication system, where even if inputted incorrect email ad password we were granted access to the system. This was due to a small oversight in the authentication class which was brought over from the JwtAuthenticationService library. The logic which governed giving access to users after they claimed their roles was simply not in any kind of check to see if the user was even granted any. This was fixed by a simple IF stamen where we checked the user's claims after they submit the form for authentication.

Once we resolved the login issues the rest of the system worked as normal. As apart from the authentication service, a lot of the system was brought over from a previous project which I worked on, this gave me the best insight there is and therefore allowed me to know from the beginning how best to integrate the components into this new project. Those components brought over from the previous project have also had Unit and Integration testing done to them as this was a requirement for the previous project.

8.3 Validation Testing

The final step of the testing was validation testing which ensures that the whole system aligns with the specifications. The initial specification did not go into details but rather left interpretation on how to implement features up to the developer. Here are the main features from the specification file:

- User Registration and login – Allow users to register and log in with accounts that can be assigned 3 different roles: System Administrator, Store Manager and Staff.

- Price Control – The system should allow managers to control many different aspects of their inventory like price, variety of sale offers and customers discounts(loyalty).
- Inventory Control – The system will allow for managing stocks and creating new ones. There should be an event that checks for low stock and alerts managers.
- Delivery Charges – A delivery multiplier can be adjusted per item to allow for bulk discounts when customers order
- Staff Shifting – The system will allow the manager to view all staff on a page and adjust their working days and hours.

Validation testing was simply done by running the system and clicking through it to ensure that each of the listed features was met to a working degree. Since we implemented the system using MVC and MOS Architecture it was very easy to adjust individual components to fully comply with the feature list. The component-based approach allowed us to get to the final stages quicker and more efficiently with fewer bugs to fix as a result.

9 EVALUATION

In this section I will discuss and evaluate the final system developed and the methods we used to reach it. This is an important part of the report as it will detail how effectively we used the patterns and what effect they had on our efficiency and quality of the final system.

9.1 SYSTEM EVALUATION

The system does everything the specification file wanted. The user can log into registered accounts which can also be created if the correct user is logged into the system. Once logged in the user can perform any of the predefined features, if the correct user type is logged in.

The system appearance was not considered until the very last end as it was not a priority, the default ASP.NET is used throughout the system with a simple navigation bar at the top which allows the user to log in and once authenticated an updated bar is created with all the links to different services to use. The UI is simple and sleek which allows ease of use and plenty of room for expandability.

For the API gateway and service discovery, we use the Swashbuckle NuGet package which generates Swagger API metadata that enables this. Each of the services communicates with their database and allows access to their functions using HTTP request which can be reached with an address and port set out in the file located in the UI application. Each service application only doses a small selection of functions that create the system.

Overall, the system does everything specified and implemented a microservice architecture. The component-based approach was explained throughout the whole project and applied. The next section will evaluate this.

9.2 METHOD EVALUATION

The development process of the Store management app applied numerous methods to achieve the final system. Tests were performed to ensure the components-based system was suited to the specification.

9.2.1 Component Mining

Component mining relies on us to crawl through any old projects, legacy systems or open-source libraries and figure out if we can reuse any of the code to save on time developing our new project. To start the process of mining I first got to grips with the current specification as this will be our domain, we are trying to map inputs from other domains. With the knowledge gathered I was able to q2uiockylm determine that we will be able to use my old coursework project as a foundation start to start developing our Store Management system.

Using code from other systems can cause some problems such as incompatibility issues between libraries or versions. However, due to the recent nature of the mined code, we were able to adapt them easily.

9.2.2 Component Adaptation

The adaption process came very naturally and easily as we were dealing with similar frameworks for our old project, we used components from. We also had full access to the code and therefore were able to fully extract the necessary components to perform adaptations on.

It can sometimes happen that the components we are trying to use are very incompatible and reusing them would cause more issues than simply finding a better implementation solution. An example might have been in the previous project was written in a very difretn4 framework and language like a python on a Django framework. Even though parts could still be mined it would become more effort to adapt them than just implement a new solution.

9.2.3 Component Integration

The final part of the component-based software engineering is the integration process which came very naturally in this domain. Not many errors were encountered as we were dealing with familiar libraries already used and tested. Due to the similar UI elements from the previous project were able to reuse a fair amount of the view from the old domain. Overall, the integration resulted in success.

10 FUTURE WORK

The purpose of the project was to develop an application with setting out features while also applying the component-based software engineering and optionally the microservice architecture which we decided to implement also.

The first critical improvements that could be made in a future version is to implement an API library to do the service discovery and communication. This would allow the system to get closer to being a deployable system. Due to the time constraints, there was just not enough time to really delve into the Ocelot library and set up all the necessary connection, as well as correctly debug and make sure the calls are being made. Although a very similar system was deployed using the Swashbuckle which automatically generated our API connections.

A final improvement to the system could have been a refactoring of the UI and possibly installing a package like Bootloader themes to allow for a more stylistic look to the system. This was a visual change and will not affect the performance and therefore was unfortunately left to the very end. Although the system still manages to look professional and simple to use.

11 CONCLUSION

Overall, the project was completed successfully and managed to implement all the features. The MSOA patter was followed which allowed for easier debugging of the system as well as the loosely coupled nature of the service means we can change them for other implementation or even expand the system with more services to perfume different actions. The CBSE methods were also followed and applied which results in some quicker development in some areas. The CBSE method may have been a little redundant with such a small system to develop however that is not to say that in the real world with an enterprise-sized application such methods can be very effective in keeping everything organised and up to date.

Appendix B – USE-CASE Diagram