

C868 – Software Capstone

Microservices in a distributed environment



Capstone Proposal Project Name: Microservices in a distributed environment

Student Name: Michael West

Program Mentor Name: Dave Huff

Table of Contents

| | |
|--|----|
| Business Problem or Opportunity | 5 |
| Gap Analysis | 6 |
| Software Development Methodology and Project Outcomes..... | 9 |
| Project Deliverables | 12 |
| Validation and Verification of Outcomes | 16 |
| Programming/Development Environment | 17 |
| Development Architecture | 19 |
| SOLID..... | 20 |
| Single Responsibility Principle | 20 |
| Open/Closed Principle..... | 22 |
| Liskov Substitution Principle | 24 |
| Interface Segregation Principle | 25 |
| Dependency inversion principle..... | 25 |
| Domain Driven Design | 25 |
| Microservices Approach | 27 |
| Distributed Patterns..... | 30 |
| Saga Pattern..... | 31 |
| CAP Theorem..... | 33 |
| Messaging and transports | 35 |
| Class and Database Diagrams | 38 |
| Costs and Resources | 43 |

| | |
|--|----|
| Implementation Plan and Timeline | 47 |
| Test Plan..... | 50 |
| Unit Testing | 56 |
| Developer Maintenance Guide | 58 |
| Introduction | 58 |
| Required installations..... | 59 |
| Optional installations..... | 59 |
| Azure Components..... | 61 |
| Local environment docker packages | 62 |
| Debugging / Running the application..... | 63 |
| Deployment..... | 64 |
| Azure Virtual Machine Deployment | 64 |
| Kubernetes Deployment..... | 65 |
| User Guide | 67 |
| Home | 68 |
| About..... | 68 |
| Search | 68 |
| Products | 68 |
| Shopping Cart..... | 69 |
| Register..... | 69 |
| Login | 70 |
| Secured Areas..... | 70 |
| Purchase | 71 |

| | |
|------------------|----|
| Orders | 71 |
| References | 72 |

Business Problem or Opportunity

Taxology is an online company provides business services to consumers. Taxology groups the services provided into the tax filing, tax advice, audit defense, and appraisal product lines. Each of the products is independent of each other but are maintained together within Taxology's portfolio. For every product, there are accounting rules and tax laws that vary at the municipality, county, and state level. These rules are codified into specifications, driving the functionality and feature set for the Taxology system. Specific areas around the country, however, have varying rates of change regarding rules and laws. The variance of regulations has an impact on the time to market for Taxology to create and deploy new features for a product. Desired features cannot currently be built without planning for and minimizing bearing on every portion of the Taxology solution, as they are all deployed as a single unit.

Currently, Taxology's system comprises only one monolithic website. This website is used as the user interface and is sectioned up into different areas for each product. These areas, however, are developed in the same code base, packaged up and deployed together as a unit. The code and deployments have become more extensive and intertwined, much like a big ball of string. Hence the term monolithic. Each product area contains all the code necessary for a product to create, order, fulfill, and function. This way of development and deployment has worked well in the past, but as the company has matured, it has found that the time to market for new concepts is becoming slower and more difficult. Products that are in regions incurring more statutory change need to be updated more frequently so that they follow applicable laws. Development of any feature requests or refactoring needed for a product line must be cognizant of the whole system and take great care not to break any existing functionality. If a failure does occur for one part, it can bring down the entire system. An additional concern is that of scale -

some products have a higher rate of consumption than others and require more capacity for delivery. Planning for this capacity affects the whole system since it is all contained within one website. Taxology would like to resolve this problem by segmenting each product into an independently deployable service. This deployment type would create a distributed environment, allowing for product features to be built and deployed without an undue amount of concern for other unrelated parts of the system. It would also allow for each product to scale independently of others.

Recent successful attempts using microservices have demonstrated the capabilities of a microservice architecture to build highly responsive, reliable, and fault-tolerant systems. These examples can be observed in enterprises such as Microsoft's Halo; running on the Orleans Actor Service (now integrated into Service Fabric), it was able to face the technical challenges of distributing a gaming network at worldwide scale. Other phenomenal successes employing the microservices architectures include Netflix, whose groundbreaking endeavors remain nimble and quick to market is in part due to their ability to develop services with a microservice framework, all while consuming nearly forty percent of the world's available bandwidth. Additional examples within the online consumer shopping realm are that of Amazon, Gilt, Nordstrom. Taxology has noted that not only have these companies launched successful microservice frameworks, but that they are all industry leaders, and handle scale well with large customer bases.

Gap Analysis

The current environment for Taxology impedes rapid development of features as described. There are four main areas in which are culpable for the most significant lack of

productivity. The first is the planning and coordination of various teams within Taxology that are responsible for the development and maintenance of the site. Even small innocuous changes need to be accounted for and agreed upon by all groups. This portion of change control process, while beneficial in keeping all parties informed, also takes up a considerable amount of time due to scheduling and attendance of meetings. Breaking out the monolithic site into smaller services help to reduce the number of meetings and coordination necessary. In segmenting the application in this way, each product team owns a service. The team then leverages their specific subject matter knowledge to good use. The team will be responsible in a “soup to nuts” manner for all things within their vertical slice of the business realm. This team can then plan for feature sets internally as needed and then only have meetings with other product teams regarding integration points between services.

The next area that has become an impediment is the size and complexity of the Taxology system. This complexity slows down the ability for developers to create changes. Changes, when made, are then subject to more defects because of the difficulty in code integration. By separating out the concerns of each product into its own service, it will significantly reduce the complexity of each particular service. The total complexity will not be completely removed, as a distributed environment will bring with it other factors that have to be considered such as network communication failures and queue management. However, development will code for these from the beginning of the process. Assuming that it is a near guarantee that there will be network failures, the code will be created to be more resilient to handle these failures.

A third area of concern deals with the impact of regression testing. Once a feature set has been planned for and developed, full site regression must take place regardless of the change. In many cases the code produced should have little impact on the greater system; regression tests

must still be executed, however, to ensure that the system is sound. These tests come with significant incurrence of time and cost. The microservices approach will still enact a full test suite. This testing suite will, however, be tailored for the service being modified, and only the integration points with the system as a whole. The segmentation of services will allow for both a pinpointed test targeted at the product level, as well as more flexibility and overall more tests to take place.

Finally, different areas of the monolithic site receive more traffic and consume more resources than other areas. If one product or area needs to be given more resources the only feasible way is to either scale up the entire site by giving it more processing, memory, or disk throughput, or by scaling out the whole site by creating a new instance. By approaching the system as a connected set of services, it enables each area to be scaled up or down according to its own need.

The plan for action on these items is to section up and model each area of Taxology into its own service. The monolithic site will then be carved up into a set of smaller services. Teasing apart the current site does not have to happen in one fall swoop. Identifying the business rules and concerns for an area allow a service to be extracted. Subsequent product or area rollouts will be phased in as appropriate, all while participating with any remaining functionality from the original site. Requests to areas that have not been extracted will just route to the original site.

To begin with, Taxology has defined the following domains for the initial rollout, concerns dealing with the creation of a customer, concerns dealing with the creation of a product catalog for listing products, concerns dealing with a customer's shopping cart, and finally the concerns dealing with placing an order. Future releases will handle the other domains of product fulfillment, billing, and customer service.

Software Development Methodology and Project Outcomes

This project is governed by following basic waterfall methodology. This process was chosen to guarantee that appropriate requirement and system design documents are generated. Since the current initiative is a migration from an existing monolithic site into smaller microservices, many of the functional requirements are known, at least implicitly. In accord with Conway's law (Conway, 1968), most of the teams have naturally organized in a way that reflects respective product lines. Due to this, each team already has a high awareness of the necessary business goals, technical constraints, and understands the conceptual domain that their microservice will be responsible for providing.

Major phases included will be that of requirements gathering, design, implementation, verification, and deployment. The requirements phase lists the basic business goals and objectives. It will document all known functionality of the existing system. Once the artifacts from requirements phase are obtained, work will then begin for the design of each service. Design of each service will include the diagrams and documents necessary to express the architectural goals and communication interfaces needed for interoperability between services. Implementation will involve all tasks essential to build and unit test a team's service. Once the executable is created, it will be verified to ensure that the service exhibits the required features and performs to stated quality metrics. Finally, a deployment plan will be created as a guide to release management of the initial rollout. Teams may choose to modify the release methodology in future releases if their needs determine a more iterative or agile approach. Some agile practices will be included within this scheme, such as daily standups at a product team level, and weekly standups that are inclusive of all product teams. The purpose of this is to help keep meetings relevant to the tasks at hand, while still providing information to be disseminated

between all groups. The waterfall approach need not apply to future endeavors – if a project team decides that there is another methodology that better suits their needs they will be free to make changes as they sit fit, as long as the project can deliver the appropriate assets according to schedule and work in accordance with the PMO's (project management office) guidance.

Enterprise Architecture will follow an adapted pattern of the governance patterns provided by TOGAF (The Open Group Architecture Framework) to ensure that architectural assets, strategies, and deliverables can be integrated into the Taxology corporation in a way congruent with a best practices approach. Enterprise architecture is defined as the organization of a system, its components, and their relationships to each other regarding the fundamental principles of an entity (such as a corporation). This provides a formal description of a system to help with the execution of essential strategies, design, and implementation of enterprise goals.

This methodology relies heavily upon a process known as the ADM (Architecture Development Method), which is a means to manage the lifecycle of projects and software initiatives.

Figure 1. The ADM Cycle

The ADM is a cyclic and iterative process which develops a system of enterprise architecture. It provides a recommended sequence to move between the phases in creating architecture. The ADM works within and is scoped to four different Architecture domains:

- Business – governance and strategies related to business goals and processes
- Data – management and structure of an organizations data resources
- Applications – Designs for individual applications and their interactions with other applications and business procedures

- Technology – System capabilities for physical and logical devices that are used for the deployment of business processes and applications, including infrastructure, networks, and standards governance.

Project Deliverables

The primary deliverables for this project are the services that compose the Taxology solution. These deliverables will be the outputs of the Implementation phase and provide all of the functionality and features for customer consumption. The complete solution will be comprised of seven independent services. Each service will have executables that will be packaged up in a Docker image. This image will be used to create runtime container instances. Four of the services will be .Net Core 2.0 console applications. Also, there will be a ASP .Net Core 2.0 website to serve as the user interface, a backend ASP .Net Core 2.0 web API that will be used as an application gateway. Data and communications will be provided by an Azure SQL Server database and a RabbitMQ messaging queue.

Deliverable listing:

1. Console services
 - a. Customer
 - b. Product
 - c. Shopping Cart
 - d. Order
2. Website
 - a. Taxology site

3. Web API
 - a. Taxology web API
4. Azure SQL Server database
5. Service Bus/Message Queue
 - a. RabbitMQ message queue

Additional deliverables are generated from each of the project phases. Each phase will produce a set of artifacts that serve as the output of the project phase and the input criteria for the succeeding phase. The deliverables defined below will be generated for each service where applicable. These deliverables and other associated artifacts are maintained by a centralized repository of knowledge for the enterprise architecture.

For the Requirements phase, the following will be the deliverable artifacts:

1. Business Goals Document
 - a. This document will state the high-level opportunity or challenge that the project is purposed with solving.
“Shopping cart information needs to be kept for analytic purposes.”
2. Functional Requirements Document
 - a. This document will contain the specific criteria for which operations and behavior of the system are defined. The standard style for writing these requirements will follow specific rules. Well written requirements include:

Condition

Subject

Imperative

Action Verb

Object

Business rules (optional in some cases)

Outcome (optional in some cases)

“When the user presses the “Register” button, the system will convert the shopping cart from an anonymous cart to a customer cart, allowing the user to retain any product selections they have made.

3. Non-Functional Requirements Document

- a. This document will list the requirements and metrics that explain how the system should perform its functionality. A specific set of metrics should be identified and quantified, such as response time MTTF (mean time to failure), MTBF (mean time between failures), MTTR (mean time to repair), and any others that are important to the service.

“Retrieving a shopping cart for a customer will have a response time under 2 seconds.”

4. User analysis Document

- a. This document will have the results from focus groups, surveys, and interviews.

5. Design

- a. Architectural documents
 - i. The logical view will be comprised of documents that identify system boundaries, processes, and communications between them. These may include activity, sequence, database, and class diagrams among others.

- ii. The Use case view will contain the specific use cases that specify the scenarios in which a user will interact with the system.
 - b. Wireframes and prototypes
 - i. These will be specific and only relevant to the Taxology website. They will include the basic conceptual design mockups as well as higher fidelity prototypes and style guides that will display the desired user interface as close as possible.
6. Verification
- a. A test plan will be provided that documents the system features to be tested and the strategies that will be employed to achieve the test result.
 - b. Traceability matrix that maps each test case back to a specific requirement.
 - c. Unit test results and code coverage reports.
7. Deployment
- a. Each deployment will provide an implementation plan that lists each deployable item, the user or group responsible for the action(s) necessary for deployment. The implementation plan will include all necessary steps to deploy each item and a master sequencing of items that will serve as a schedule. Each implementation plan will also include rollback steps in case of a failure within the deployment process.

Validation and Verification of Outcomes

The first quality gate that the project employed by the project will be the validation of requirements. This will be performed by the business analysis team in conjunction with the team responsible for the service in question. This initial validation of requirements is to ensure that the business need and purpose is captured by the requirements.

Verification of the specifications listed in the requirements will be performed on multiple levels, starting at the most granular level of code and progressing to the system level. The developers responsible for the service will complete unit testing. This will produce a pass/fail list of test cases the all the significant code in place. A code coverage report will be associated to the list of unit tests to call attention to any gaps that may be present in test coverage. This acknowledges the fact that some areas of the code base are either not feasible to test through unit tests, or that tests created for the gap would produce brittle and time-consuming tests that would cause instability in the development process.

After unit testing has achieved a positive result and all required features are accounted for, the service will be deployed to a staging environment. One environment will be dedicated to the Quality Analysis group, allowing them to execute the test cases defined in the test plan. Test harnesses and scripts will be created so that all testing is able to be repeatable where possible and will be combined with manual execution of the process from the front-end website to where applicable. It is important to capture both the results that pertain to a single service as a system, as well as an integration that the service has with other services.

After the QA group has approved the service, it will be deployed into a staging environment designed for user acceptance testing. Here the focus will be on the ability for users

to perform necessary system actions. Both a quantitative and qualitative result will be obtained from UAT testing. A short listing of the actions would include:

1. The user can easily navigate the website
2. The user feels that the design of the site is
3. The user can view a listing of products
4. The user can view details of each product
5. The user can add a product to a shopping cart
6. The user can remove a product from the shopping cart
7. The user can view the shopping cart and their chosen products and a subtotal of items
8. The user can register with the website
9. User can login
10. The user can make a purchase, which will create an order and clear out the shopping cart

Programming/Development Environment

The programming environment will consist of the following technologies:

1. Visual Studio 2017
2. .Net Core 2.0 framework
3. Docker Containers running Windows Nano server
4. Microsoft Azure
 - a. Hosting environment

- b. Azure SQL Server

5. RabbitMQ

The languages used in this system are:

1. HTML
2. JavaScript
3. CSS
4. C#
5. SQL

Framework packages included:

1. AutoFac -provides an IOC for dependency injection.
2. AutoMapper – provides a utility to create instances of a class and map properties from one instance of a class type to an instance of a different class type.
3. AutoFixture – provides the ability to automatically generated mock data and instances of interfaces and abstract classes for unit tests.
4. Entity Framework – EFCore – provides a SQL object relational mapper ORM.
5. MassTransit – provides a message bus for publish/subscribe, send/receive messaging concerns (utilizing, in this case, RabbitMQ as the message queue).
6. MediatR – provides the Mediator pattern for publishing Domain events.
7. XUnit – provides a suite of unit testing capabilities for a unit test project.

Development Architecture

When approaching a migration into a microservices environment, most get caught up in which orchestrator they are going to use. There are a few good options out there, Docker Swarm, Kubernetes, and Azure Service Fabric are three top contenders. These orchestrators have an important utility and ease management of many small services. They provide health monitoring, scaling, service name resolution, and a wide variety of other tasks. While they do these things well, more is needed to develop microservices. One is quite capable of creating a monolithic service or website and managing it with an orchestrator. Unfortunately, this does not somehow transform the monolithic service into a microservice. A further review of what a microservice is will come later, but it is worth noting that it is not defined strictly by size. The aim is not to make arbitrarily small or “nano” services. While IFTT (if this then that) operations (such as hosted functions, webjobs, etc.) can be employed to great use, they are not considered a service. A microservice is as big as it needs to be to maintain its business purpose. The lion’s share of complexity lies in modeling up the system so that each service has the right separation of concerns. Splitting apart a monolithic environment into this is no small task. A considerable amount of thought should be placed into what goes into a service and what does not. System boundaries, once defined, are much harder to refactor if you get the initial categorization wrong. For instance, in a monolithic environment if you find that you need to move a class from one module to another, it is a relatively easy process. Create the move refactoring, build, test, and deploy. In a microservices approach however, this will involve much more work if it is found out that the class in question really belongs to a different service altogether. Not only does basic code refactoring take place, but also added to the list is communication endpoints, data

transmission objects, and a general restructuring of purpose/concern for a service (Newman,2015).

The following sections define some of the vital principles and practices that are used in creating a solid foundation for productive, manageable, and clean code.

SOLID

Practitioners of software design have accumulated various lessons over the years. These lessons are sometimes learned during successful projects but are often gained by slogging around in the muck and mire and regretting the choices that brought them to the point they find themselves in. Eventually, these lessons become codified, virtually canonized for future endeavors. The benefit of foundational principles can then assist in reducing the complexity and frustrating experiences. While in truth, the Nirvanic experience might rarely be obtained – the principles at least stave some of the frustration off in the attempt to get there.

One set of principles goes by the mnemonic SOLID, which stands for Single responsibility, Open/closed, Liskov substitution, Interface segregation, and Dependency inversion. These principles are one of the bedrocks of which the rest of the architecture is based on.

Single Responsibility Principle

The SRP is a key principle in designing and developing organized and useful code. Its fundamental idea is that every class should only have one purpose. All the members and functions contained within it should be devoted in some way to achieving that purpose. There is no precise amount of lines of code that a class or method needs to have to maintain the SRP. Visual scans usually identify code smells, crying out to be changed. A good rule of thumb is to keep methods within ten lines of code, and classes under a few hundred– but again this is an

arbitrary and ideal number only. Taking metrics such as the Cyclomatic Complexity of a method (branching logic) might serve to pinpoint areas of interest better.

The SRP helps in organizing the basic code structure – when searching through the solution it is easier to find related functionality. Things are grouped together that change together. When a new feature is added that causes change to existing code, ideally only one thing must change. As an analogy, consider silverware in a kitchen. Knives, forks, and spoons are all kept in the same drawer. The drawer would be considered the class, while the utensils would be the class members. Grabbing a fork is an easy process because the location is always known and easy to remember. If the utensils were spread out in different drawers, however, it would be much more difficult to remember which drawer it was in. Likewise, if the silverware is at some point replaced (a change occurs), it becomes a much easier process and it can be confidently known that all of silverware was changed. In practice, it is not always as clean as in theory – but it is an excellent goal and worthy of attempt. Continuing the example, maybe potato peelers find a home in the same drawer as the silverware. Possibly even the ice cream scoop. But surely all can agree that spatulas have no home there. A line must be drawn somewhere.

This principle also extends throughout the software paradigm. As a class should only have one responsibility, the same can be said for a method. Not only does this aid in understanding the code, but it is a primary driver of being able to write unit tests that are not brittle and confusing. It also has an influence on ideas such as bounded contexts (discussed below) and microservices. A service exists as a set of related functionality that serves a common goal, no more and no less.

Open/Closed Principle

This principle states that things within the system should be created in such a way that they are closed to modification, but open for extension. The goal here is to write code in such a way that plans for future changes and can modify the system without changing (presumably) good stable code that has been well tested. An example here might be a class which takes in an interface implementing the Strategy pattern as a parameter. (Vlissides, Helm, Johnson, & Gamma, 1995). The consuming class might be designed to calculate an employee's payroll. The class itself might perform some simple operation, such as getting the employee's weekly pay amount, and then allocate amounts for 401k contributions. Perhaps, in the beginning, the only type of employee pay was based off an hourly rate. As the company grows, it adds a sales department, whose employees pay is a combination of hourly and commissions. By injecting the strategy interface to the class, a strategy can be created to handle hourly pay, and another one created to handle hourly and commissions. New strategies can be implemented in the future by extending the interface, without having to modify the consuming class.

Figure 2. Displaying the strategy pattern

```
public interface IGetPayAmountStrategy
{
    decimal GetPay();
}
public class HourlyStrategy : IGetPayAmountStrategy
{
    private decimal PayRate { get; set; }
    private double HoursWorked { get; set; }
    public decimal GetPay()
    {
        return PayRate * (decimal) HoursWorked;
    }
}
public class SalesStrategy : IGetPayAmountStrategy
{
    private decimal PayRate { get; set; }
    private double HoursWorked { get; set; }
    private decimal CommissionAmount { get; set; }
    public decimal GetPay()
    {
        return (PayRate * (decimal)HoursWorked) + CommissionAmount;
    }
}
public class PayrollService
{
    private readonly IGetPayAmountStrategy strategy;
    public PayrollService(IGetPayAmountStrategy strategy)
    {
        this.strategy = strategy;
    }
    public decimal GetTotalAfterDeductions()
    {
        decimal deductions = 1000;
        decimal amount = strategy.GetPay();
        return amount - deductions;
    }
}
```

Different techniques can be employed to achieve this such as inheritance or composition.

In many cases, however, it is better to use composition instead of inheritance. This reasoning behind this is that deep inheritance often serves as an impediment to change. A change to a base class such could have a broad impact on all inheritors, while composition tends to affect fewer components.

Liskov Substitution Principle

This conveys that when objects are used in a polymorphic fashion, the functionality of a consuming class should not have to alter itself according to the type. A simple way of understanding this is that when an object is passed in as a parameter, the consumer should only have to rely on the public contract of the object's interface. The consumer should not have to know precisely which type of interface implementation is passed in. Consumers should be able to provide its functionality without having to downcast the object to a specific type. The reasoning behind this is that it in some way defeats a purpose of polymorphism. If for example, the consumer has branching logic based on the type of object passed in, then future changes might be more likely to change the consumer. A new implementation of the interface is unusable in this case without also changing the consumer class and adding to its branched logic.

Figure 3. Breaking bad

```
public class BadConsumerPayrollService
{
    private readonly IGetPayAmountStrategy strategy;

    public BadConsumerPayrollService(IGetPayAmountStrategy strategy)
    {
        this.strategy = strategy;
    }
    public decimal GetTotal()
    {
        decimal amount = strategy.GetPay();
        decimal deductions = 1000;

        //intimate knowledge of the type has to be known
        //this downcast would make Barbara Liskov frown
        if (strategy is SalesStrategy)
        {
            decimal bonus = 1000;
            amount = amount + bonus;
            return amount - deductions;
        }
        else if (strategy is HourlyStrategy)
        {
            return amount - deductions;
        }

        return 0; //oops!
    }
}
```


Interface Segregation Principle

Instead of having one interface that serves as a general purpose for many different things, interfaces should be defined so that they perform a smaller set of functionalities. The definition of the interface is then targeted to grouped behavior, which allows for better understanding of code, easier composition, and implementing classes that follow the single responsibility principle.

Dependency inversion principle

Dependency injection is made possible due in part to this (along with inversion of control containers). When possible, it is better to use interfaces and abstractions instead of concretions. By doing this, a system can change its behavior and functionality with less refactoring. New implementations of an interface can provide different behavior while conforming to a generalized contract. Coding with this in mind is also a great benefit to unit testing. Interfaces can be mocked or faked out and used in a unit test. In some instances, unit testing is not even possible without this. Examples of candidate interfaces might be those for repository or web API calls. For instance, a web API call to a third-party system might incur a fee. Running thousands of iterations of a unit test might very well break the bank!

Domain Driven Design

The current solution for Taxology has been built in accordance with DDD. While an in-depth discussion of DDD is outside of the current scope, it is worth noting there is a natural and cohesive relationship between the paradigms of DDD and microservices. The concept of a bounded context from is a rudimentary principle that aims to break down a domain problem

space into different sections. As a system grows, its domain model (classes that model the behavior of a system) becomes more complicated. Some classes exist to provide one set of goals, such as maintaining customer information, while other classes exist to provide a different set of functionalities, such as maintaining orders placed for products. One approach is to create a unified model for a company, with domain classes that are incorporated into both sets. For some systems, this is an entirely appropriate design choice. As the system grows another approach is to look for all the core uses cases of a subject and group those into a bounded context. As noted within the discussion on the Single Responsibility Principle, a bounded context is a related concept in that the bounded context exists for a specific responsibility. The grouped concerns of a bounded context also fit well with the microservices ideology for system boundaries.

Elements of DDD are exhibited throughout the Taxology solution. Each bounded context is modeled and deployed as an independent microservice. Within the bounded context, the behavior is modeled by using Aggregate roots. These root objects encapsulate behavior and data access. For example, the shopping cart service contains a ShoppingCart class. This class provides the operations for interacting with a shopping cart such as adding and removing products. A shopping cart product class is modeled as well. However, all interactions with a product must go through the shopping cart. In other words, the shopping cart owns the products it contains. An important element of an aggregate root is that all traversal of its graph depends on interaction with the root instance. Children within contained within the aggregate are managed by the root, and no other object is allowed to maintain a direct reference to any child. References to children must be only be used as transient, falling out of scope if they are passed to an object outside of the aggregate. By maintaining this rule, the aggregate root is able to maintain self-consistency and ensure that all object instances within the system are in a valid state. Aggregate roots are

therefore tasked with ensuring that they are always valid within the system, from creation to deletion. At creation the immutable class pattern might be employed, only allowing property data to be changed via constructors (and possibly methods). In this way, the aggregate can ensure that all its invariant rules are always satisfied. Invariant checking is separate from validation that occurs later in the aggregate's lifetime. Take for instance a class representing a customer. A customer must always have a name and an email, from its initial creation onwards. However, the customer does not always have a phone number. If given, the phone number must be in a correct format, but some people do not have, or do not want to give, their phone numbers. In this instance, at construction, the customer would make sure that its invariant data, the name and email, are always present and throw an exception if missing. Later, through some other process, the phone number can be added and validated for accuracy.

Microservices Approach

There is no exact definition of what a microservice is, although a consensus seems to be that a microservice is a subset of SOA (service-oriented architecture) and that it exhibits a some of the following key attributes:

- Distributed System
- Organized around business capabilities
- Componentized
- Have smart endpoints and dumb pipes
 - As opposed to an enterprise service bus
- Managed with infrastructure automation

- Name resolution, failover management, cluster management, image stores, health monitoring, and rolling upgrades. Usually carried out by Kubernetes, Docker Swarm, or Service Fabric.
- Designed for failure
 - Networks are not reliable. Servers and application code fail
- Have test and deployment strategies
 - Blue – Green deployments (Fowler, 2010), A/B testing, Canary releases
- Have decentralized data management

The first three of these items factor into a golden aim of software, to be loosely coupled but maintain high cohesion. When services are loosely coupled, a change to one service should not require a change to another. A primary goal of a microservice is being able to make a change to one service and deploy it, without needing to change any other part of the system. It is componentized in the sense that services can be plugged in, switched out, and added. Much like a home theatre system, one could add a subwoofer component, a Blue-ray player component, or an amplifier. Services are enlisted in the same way, each offering additional capabilities.

Other regular themes include separating code and data stores in such a way that is not shared between services. While this is admittedly a stumbling block for many traditional developers, it is helpful in creating atomic services into small manageable chunks. Once code frameworks become both pervasive throughout an enterprise, it encroaches on the ability of developers to change the system without affecting other projects. The same goes for a shared data store. A solution that allows other projects to create queries and reports that directly access their database, real-world concerns begin to stymie change. There are of course benefits to both shared code and data access, and one does not have to throw the baby out with the bathwater in

every instance. Some teams might find that there is framework code that handles the infrastructure plumbing and can be reused by all teams. This is seen within DDD as well – bounded contexts will usually have a separate code base, but a shared kernel might be common to all. It might serve a company well to use one single database for licensing concerns while adhering to separate schemas and strict boundaries dealing with access.

One benefit of microservices is flexibility and speed of development. Another is concerned more with the meaning and purpose of the code. As an example, the Taxology solution, being a shopping cart site, has two different services that have the idea of a product model. The product service is one which serves as a product catalog. Its product model contains detailed information about a product, its price, when it expires, and so on. The order service also has product model, but the information it needs is different. Orders only care about the name and the price of a product for tallying sums and recordkeeping. As services grow in disparate ways, the context of what a domain model means might begin to vary. At some point, it becomes a cleaner solution to simply have two different product models, each within their domain. All the benefits of microservices come with a cost though, much to the chagrin of DRY principle aficionados. One critical area that pays a penalty for this separation is data transactions, and this will be looked at in the next section with distributed patterns.

Distributed Patterns

One of the upsides of microservices is that are distributed. Of course, one of the downsides to microservices are that they are distributed. How to go about handling data transactions is a point of concern and consternation. Data transactions within a distributed system are problematic and must be accounted for differently than the standard ACID model. To review what an ACID transaction is:

- **A**tomic – it is an all or nothing proposition. All the statements that are part of the transaction must be successful as a unit.
- **C**onsistent – the data store must be in a consistent state both before and after the transaction. In a database table rules such as primary keys, foreign keys, unique constraints, and similiar rules, must all be able to be enforced.
- **I**solated – concurrent operations must be able to be isolated from each other – although the level of serialization differs in different database systems and can generally be a configurable option.
- **D**urable – if a transaction is going to be committed, then it needs to guarantee that the commitment is a lasting one (until another change), regardless of system crashes and shutdowns.

In a non-distributed environment, this scheme works out very well. Data is input into a system, and it can be relied upon that the data operation either succeeded or failed. Once a few more data systems are added to the picture it becomes problematic to keep up this rigor. Some strategies exist for this, such as 2PC (two-phase commit), which manage distributed transactions by having a managing process ask different data stores to take actions, and then afterward poll

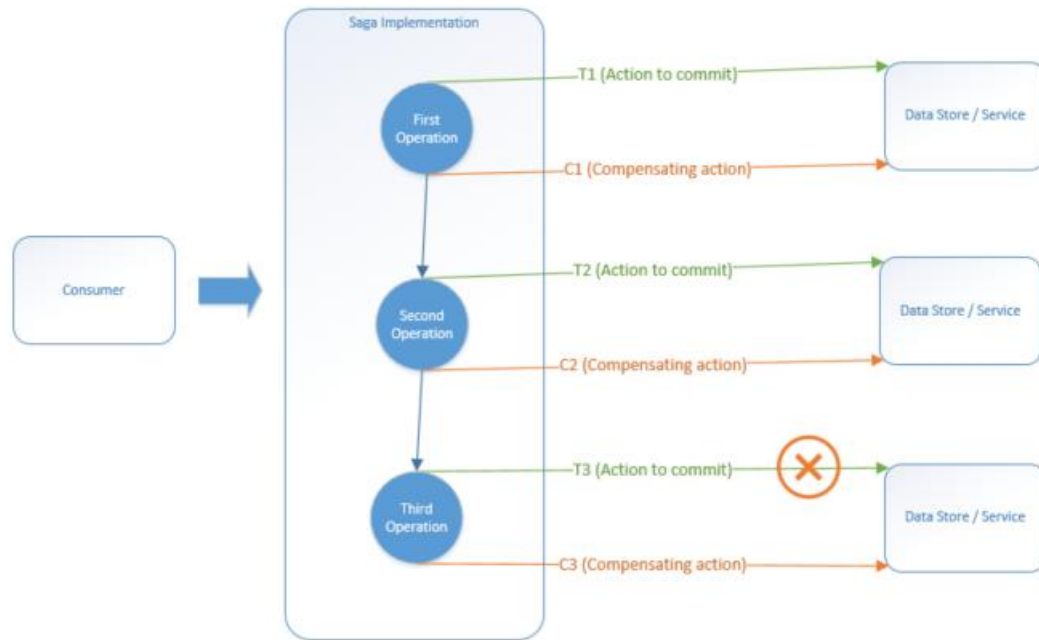
each data store for its status. Each data store then votes for the operation to be successful, the manager tallies the votes and if all systems respond with a go-ahead message the transaction is finally committed. Technical challenges arise however due to the expense of managing these transactions over a network which can fail. This could be potentially troublesome when locking resources that block further access to other consumers for some indefinite period.

Saga Pattern

An alternative to this is a system that relies upon another type of transaction management through Sagas or routing slips. The idea of a Saga as a pattern for handling transactions was developed by (Garcia-Molina & Salem, 1987). In it, the authors proposed a scheme for transactions to follow the ACID model in each data store locally but have a managing process that can undo any operations if a failure occurs with any of the contributing stores. The Saga manager controls some desired process and sends a transaction (denoted as T_i) to a data store (database, service, API, etc.). That transaction becomes a standard ACID transaction within the local system that T_i corresponds to. If that transaction succeeds, it moves on to the next operation, and then sends T_2 to the next data store. This process continues either till all operations have been completed, or till a failure occurs. In the case of a failure, the Saga manager begins to roll back the operations by sending compensating actions to each service. These compensating actions (denoted as C_i) then instruct the data store how to undo the previously committed transaction. This can be visualized with the next figure, in which the Saga begins by executing the first operation, which in turn sends T_1 to the first data store. After this completes successfully, it begins the second operation, which likewise sends T_2 to the second data store. Finally, in the third operation T_3 is sent, but a failure occurs. The Saga manager recognizes this

failure and then begins to unwind the whole operation in reverse order, by sending the compensating action C_2 to the second data store, and then sending C_1 to the first data store.

Figure 4. Saga pattern illustration



Compensating actions can either recover backward (as displayed) or they can recover forward. A backward recovery method might be a simple undo operation – if an insert was committed, then a delete command is issued as the compensating action. They can also be forward, in which case the Saga moves to another step that ensures that the whole process is still valid, and no work is wasted. This can be useful if operations within the Saga incur monetary costs or performance penalties. In these cases, the Saga might be implemented as a State Machine or in conjunction with the Byzantine or Fail stop approach (Schneider, 1990). With the Saga, the state machine provides the ability to have its state rehydrated and have save-points.

These save-points are then enlisted so that compensating actions can rollback to them. It is also possible for the compensating actions to fail, in which case the Saga would need to continue retrying them for success. Due to this, compensating actions should be created as idempotent operations, so that repeated occurrences would leave the system they act upon in the same state.

In an ideal world, all the operations of a Saga would execute successfully, and no outside processes would perform any intermediary requests. This might not always be the case, however, and system architects should plan for different consistency modes. Strong consistency occurs when an operation occurs, and every subsequent call, from every location, is guaranteed to receive the same result from the previous operation. While this might be the desired goal, in distributed systems a choice must be made. Does the system need to favor consistency, or does the system need to favor availability?

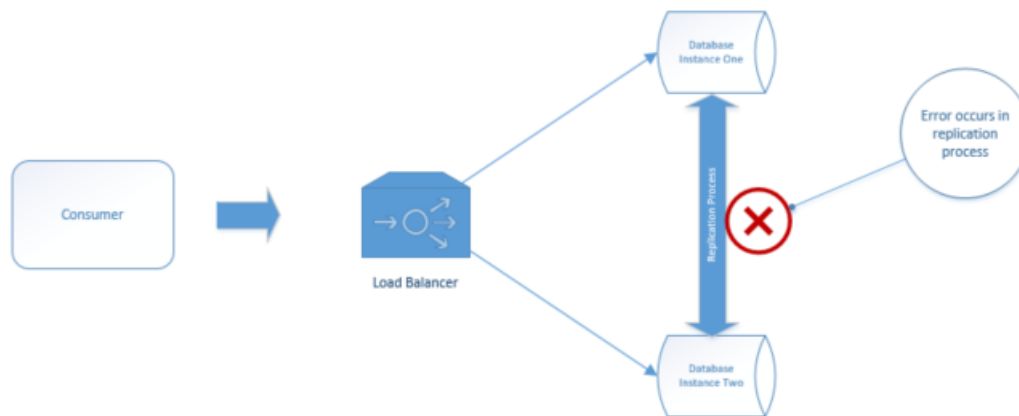
CAP Theorem

This choice is due to the CAP theorem (Brewer, 2000), which demonstrates that a logical choice must be made between consistency, availability, and partition tolerance. Consistency in this context means that all the data present exists in one system is present in another. Being distributed is a transparent process – if a query is ran against two data stores at the same instant the results will be identical. Availability denotes the uptime of a system, with the idea that if an operation such as an insert occurs, then both systems remain available – one of them does not get pulled offline until it also updates its data. Partition tolerance means merely that network failures, while hopefully rare, do occur.

A system can guarantee two of these choices are present at any moment, but not all three. For example, a system can decide that it will always be consistent, and it will always be available, but it can only do so if it does not cross any network partitions. (Brewer did note that

systems that run on LANS or have guaranteed network failover routes can be considered non-partitioned. However, if the network resources are located outside of this, such as in separate datacenters, cloud environments, or can encounter any network failure it is partitioned). Systems might also choose to have resources in various network locations and always be consistent – but at the cost of availability. A choice for many systems is to be deployed across a network partition and always be available, but in this case, there might be times in which it is inconsistent. This implies eventual consistency instead of strong consistency. Eventual consistency is a form of weak consistency, where the systems involved can be expected to become consistent eventually. A widely used system that relies upon eventual consistency is DNS (domain name system), where updates are not expected to be simultaneously present across the world.

A simple display of the CAP theorem can be seen with load balanced services with databases that have high availability and rely on replication. When a client executes a command to insert data through an instance of a service, that service then communicates with its data store, that data store then commits its transaction, and a replication process assists in moving the data from one database to another. It is entirely possible though that the replication process fails. From the moment that the first transaction committed until the replication process succeeds, the system is in an inconsistent state.

Figure 5. CAP theorem view with database replication

Messaging and transports

While distributed systems can rely on any communication mode that they prefer, in many cases some brand of message queue is employed. Examples include MSMQ, Azure Service Bus, Kafka, RabbitMQ, and many others which generally implement the AMQP (advanced messaging queuing protocol) as a transport standard (MSMQ does not). By enabling a queue to operate as a message broker, applications can send commands, notifications/events, and messages to other application recipients in an asynchronous manner. This achieves both an added layer of durability and flexibility between applications. When an application/producer sends a message to a queue, this message is available to be received and processed accordingly by recipients/consumers. Once a recipient of a message has concluded its business, it notifies the queue that the message has been acknowledged, and then have the queue remove the message. If for some reason a consumer fails, then that message can be available for consumption again (details on this vary, with messages sometimes becoming poison letters, being abandoned, other

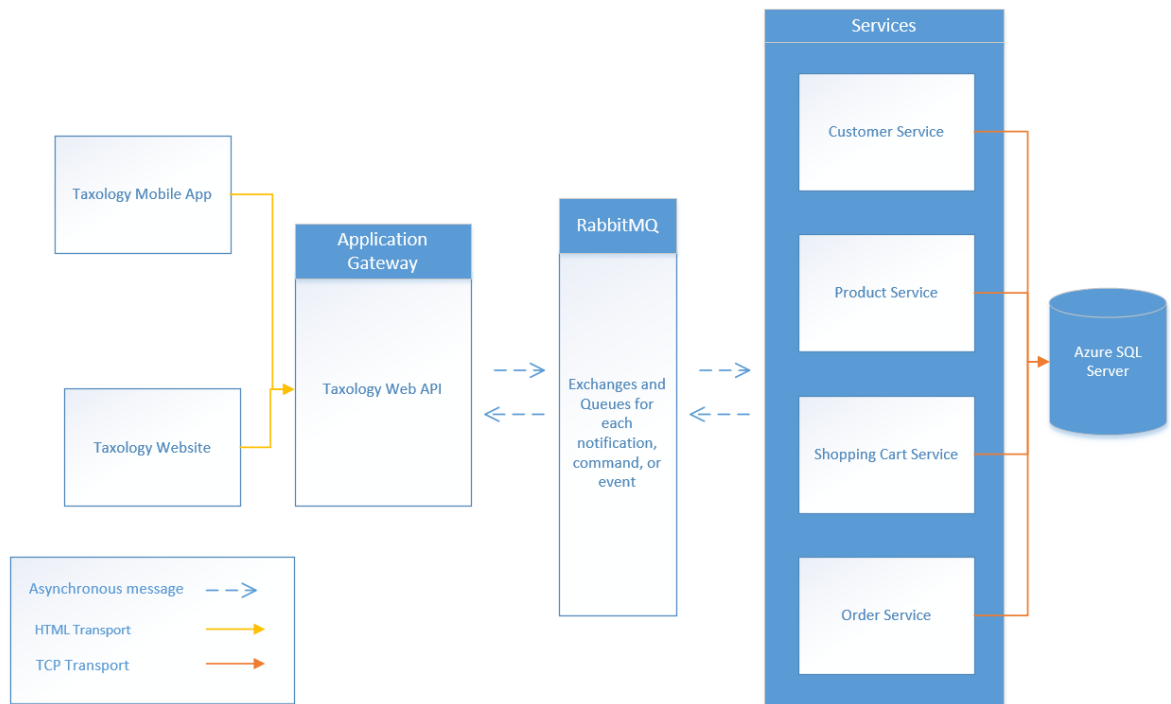
scenario factors). Compare this to a standard REST call, in which an HTTP request is sent to a server, and if for some reason that server fails, the request is effectively lost. A client can, of course, retry the REST call again, but it must build the context message back up.

Flexibility is achieved because producers of messages do not have to be overly concerned with who is consuming a message or how they do it. In contrast again to a REST interface, changes to either the interface or the server can be problematic for client applications. If the server moves locations, it can be troublesome for clients. While these problems are not always removed with queues, it is generally much easier for a team to do whatever they need to with a server. Clients do not have to be concerned with any of these changes; the only reliant factor is that a message is produced (usually in some managed format such as JSON or XML) and that the message will be delivered to the queue that they are listening to. In meetings for process control the phrases “don’t worry about it”, “all you have to know”, and “the same message will end up in your queue” can be heard and they are a relief to hear!

Other patterns exist for different messaging intentions such as competing-consumers, publish/subscribe, fanout, and other patterns. The Taxology solution includes a RabbitMQ server and uses an open source framework called MassTransit to provide an administrative overlay on top of code. While writing code to interact with queues is not exceedingly complex, many different scenarios must be accounted for, and MassTransit provides an excellent and opinionated patterned framework and plumbing code to ensure that this code is accurate for a wide range of possibilities. In the Taxology solution, a front-end website is accessible by clients and communicates directly with a backend web API through a REST interface. At this point, the web API then sends a message to whatever service needs to be communicated with and then awaits a for a responding message to be sent back through the queue. Once the API has received the

response, it then responds back to the frontend website as appropriate. The web API in this instance serves as an Application Gateway; a pattern used to route messages, provide authentication, and provides a single point of entry for all clients. This enables future clients to be added (such as a mobile application) and shields clients from many concerns of the backend system.

Figure 6. Taxology system topology



Class and Database Diagrams

Each microservice in the Taxology solution space is purposefully kept small. As previously discussed, only obtaining the requirements necessary for its bounded context to be fulfilled. This contrasts with a monolithic system, which over time grows in complexity, with different developers creating code in their style, and ending up with a system that struggles to maintain its own architecture (or at times under the weight of it). These systems are affectionally labeled as Big Balls of Mud.

A portion of the Customer service is displayed here, with class diagrams for its domain project, and its service project, as well as a project dependency diagram. The Customer domain project contains the Customer class, as well as domain event classes that track an instance of the Customer domain object's lifetime. Domain events are created when a Customer instance is created and added to the Customer's list of events. When a Customer instance is saved, it then publishes these events to anyone who cares to listen for them to occur. For example, when a customer is created, a CustomerCreatedEvent is added to the list, later when this customer is saved, this creation event is published. The Customer Service listens for this event, and then publishes an Application Event to the message queue. From there any other service can hook into this notification. In the present example, the Shopping Cart service listens for a customer creation event. When the shopping cart is created, it might have been created by an anonymous user based off an Id saved within the browser's cookie. When a user decides to register, the user's anonymous id (if present) is sent to the Customer service, who then creates and publishes

the event as noted earlier. The cart then ties the event data to its data store, converting an anonymous cart into a customer cart.

Figure 7. Event processing sample

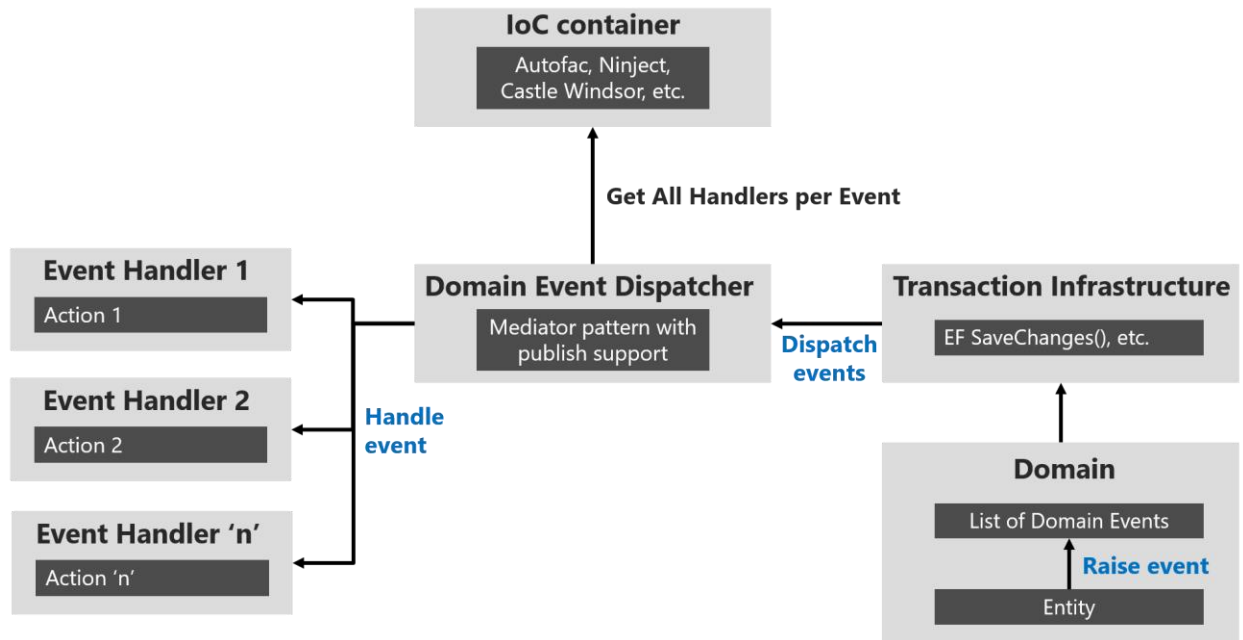


Figure 8. Customer Domain class diagram

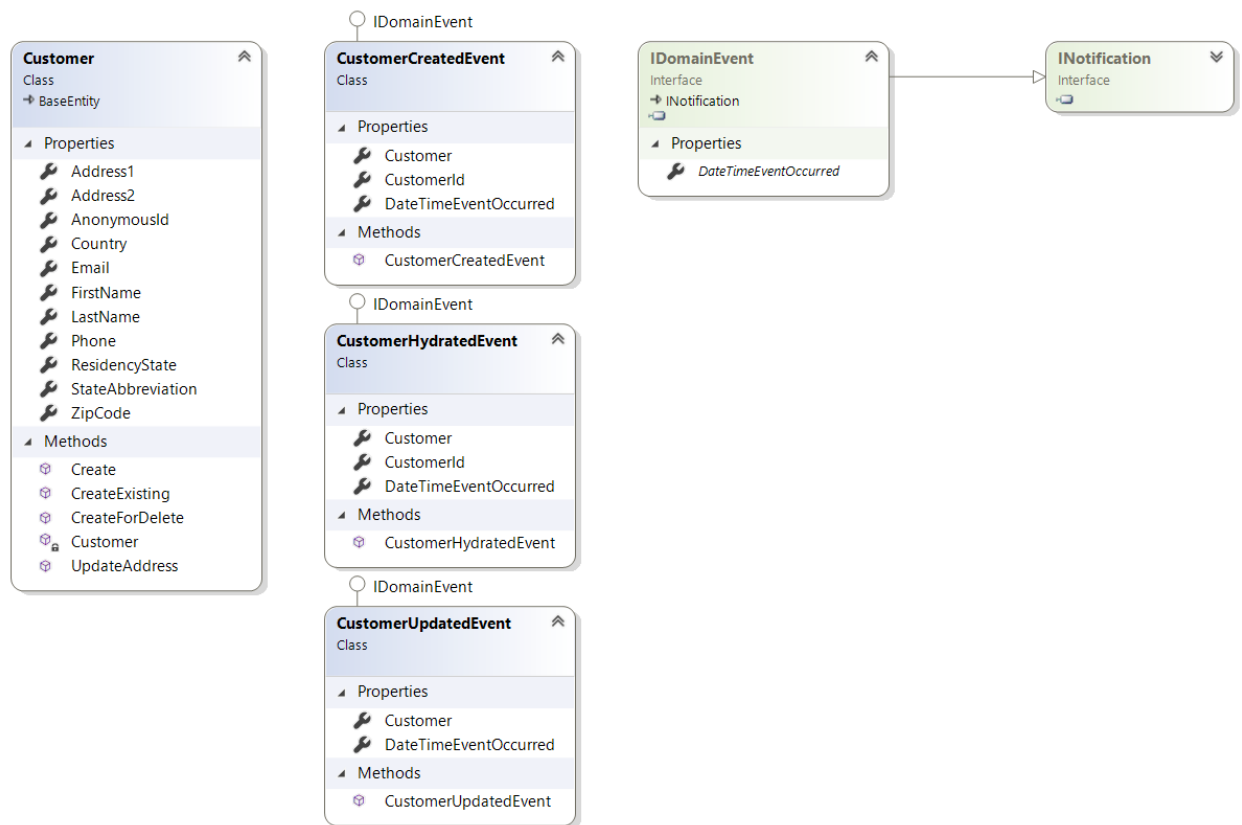
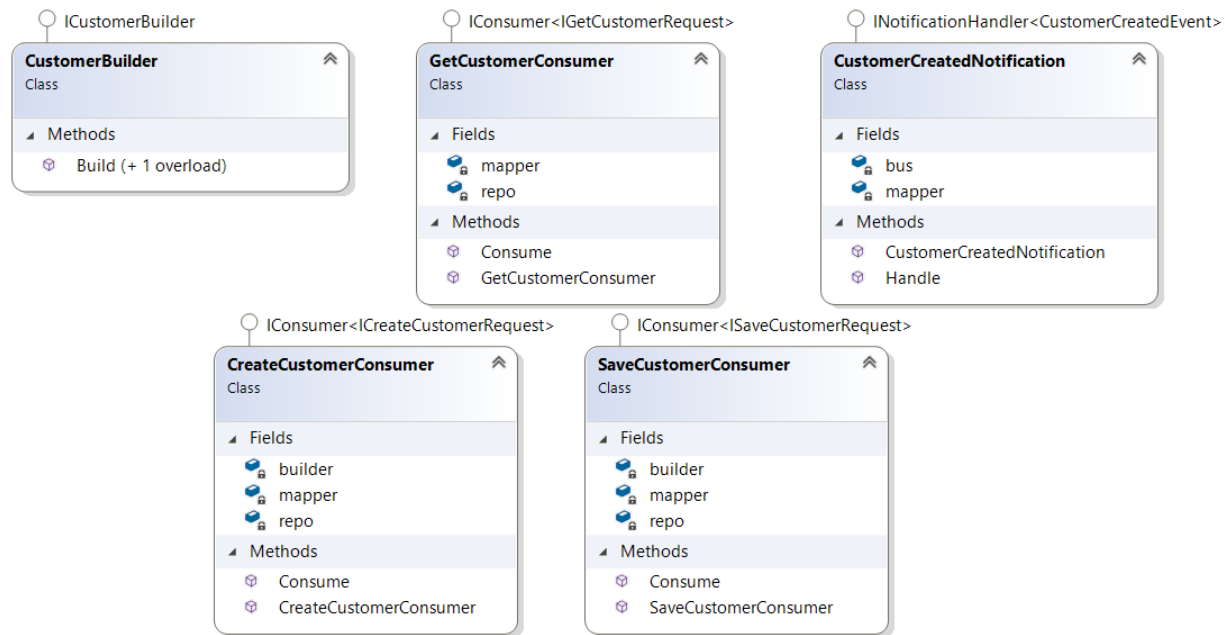


Figure 9. Customer Service class diagram

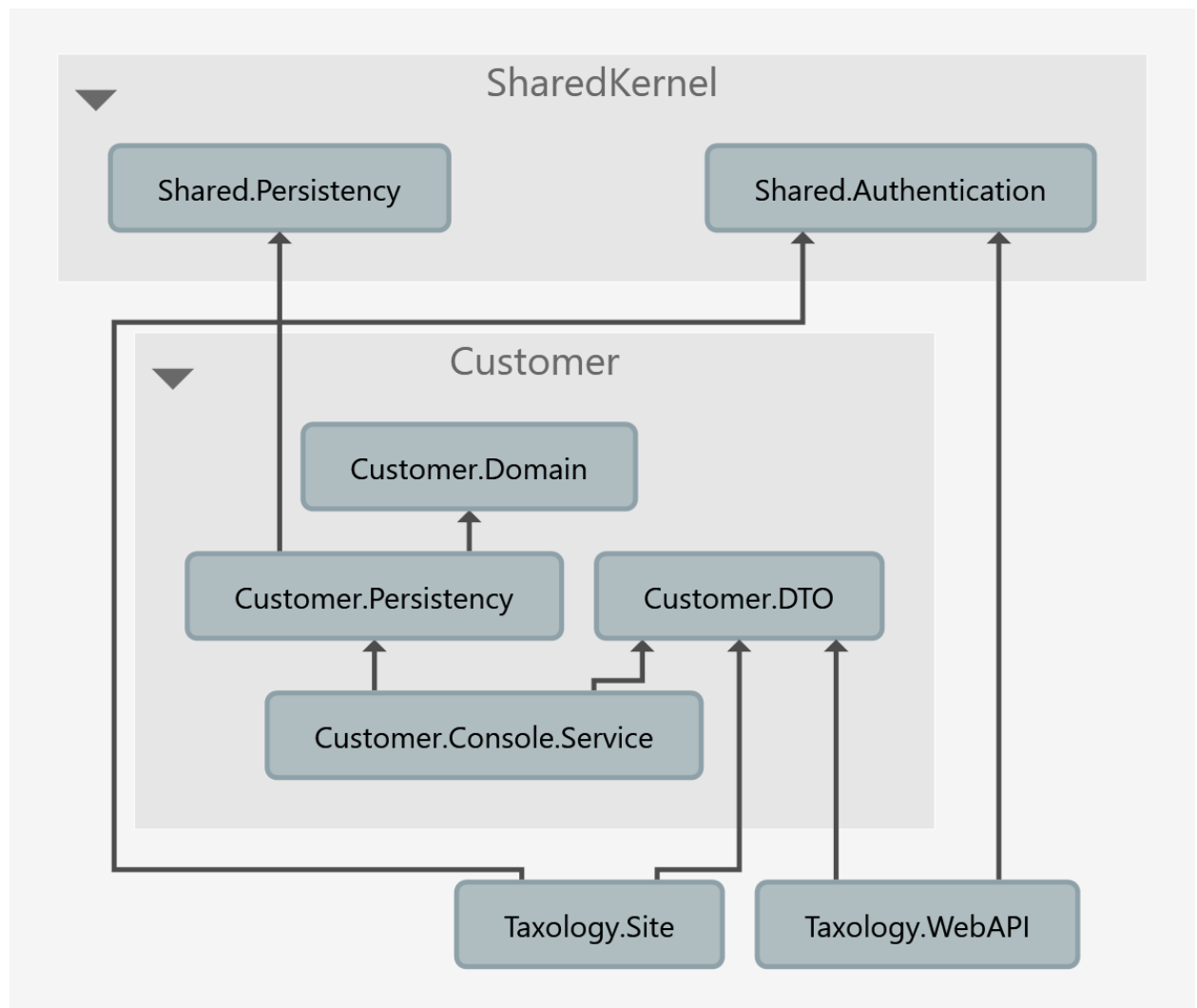


The general project structure for the Taxology services contains four similar projects

1. Domain – this project contains the domain models and business logic behavior
2. DTO – this project provides the contract for other projects outside of the service to use. The types of Request objects that a service responds to, the Response object, and data transmission objects that are used in conjunction with an anti-corruption layer. Domain objects stay within the service, outside services can only reference DTO objects if they chose to work with a service.
3. Persistency – manages all the data concerns utilizing Entity Framework and repository patterns. Also, is responsible for publishing domain events after a transaction is successful.

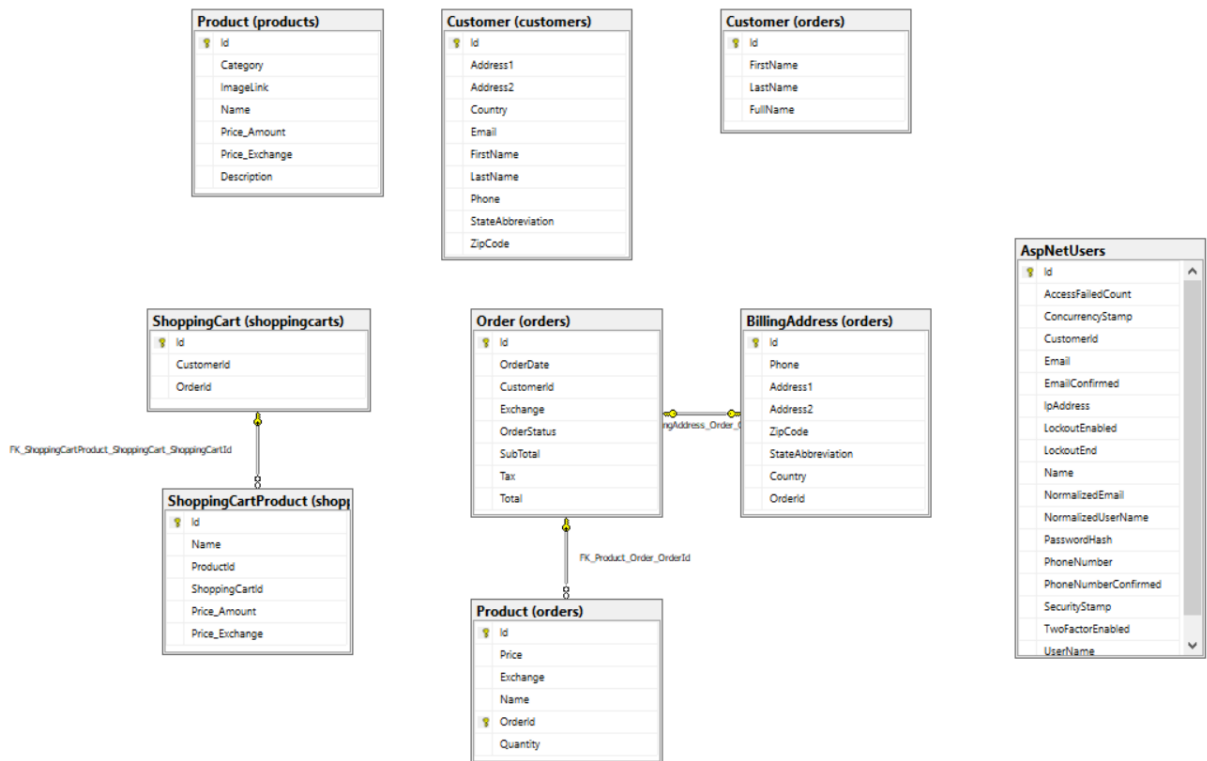
4. Service – serves as the basic executable and orchestrator for a service containing references to all other service level projects, handling any authentication necessary, providing an anti-corruption layer, etc.

Figure 10. Customer Microservice dependency diagram



The database for Taxology is all contained within one Azure SQL Server and one database within this server. The reason for this is simplicity and cost savings for this project. However, each service only accesses data from its schema, and foreign keys do not cross schema boundaries.

Figure 11. Taxology database diagram



Costs and Resources

In total there will be six separate services created. Each team is comprised of a project manager, a business analyst, three developers, and two quality assurance representatives. Most of costs for this project is generated by human resources. While the costs associated with each

resource will have some variance, a generalized projection for personnel costs is shown in the following figure.

| Resource | Cost | Duration (in days) | Quantity | Total |
|--------------------------------------|---------|--------------------|----------|--------------|
| Project Manager (PayScale, 2018) | \$66.34 | 84 | 1 | \$33,966.08 |
| Business Analyst (PayScale, 2018) | \$49.83 | 21 | 1 | \$8,371.44 |
| API Developers | \$49.62 | 63 | 3 | \$75,025.44 |
| Quality Assurance | 32.40 | 63 | 2 | \$32,659.20 |
| Total for Service | | | | \$150,022.16 |
| Total for all Services | | | | \$900,132.96 |


Taxology services are deployed to the Azure Cloud provider. A normal solution would be spread across many different virtual machines, but due to cost aversion, the complete application has been deployed to one virtual machine. For accurate and up to date pricing models for Virtual Machines and Azure SQL Server refer to <https://azure.microsoft.com/en-us/pricing/calculator/>.

In addition, costs will be accrued for the Azure SQL database, any domain name registrations, and SSL certificates.

SSL cost quote was taken from DigiCert at: <https://www.digicert.com/ssl-certificate/>

Domain Name quote taken from Domain.com at: <https://www.domain.com/domains/>

*Figure 12. Current Cost captures (3 images)**Azure VM*

 Virtual Machines

REGION:
West US

OPERATING SYSTEM:
Windows

TYPE:
(OS Only)

TIER:
Standard

INSTANCE:
B2MS: 2 Cores(s), 8 GB RAM, 16 GB Temporary storage, \$0.135/hour

Billing Option

Save up to 72% on pay as you go prices with 1 year or 3 year reserved options. [Learn more about Reserved VM Instances pricing.](#)

☒ Pay as you go
☐ 1 year reserved (~39% savings)
☐ 3 year reserved (~59% savings)

Save up to 40% with Windows Server Licenses you already own. [Learn more about Azure Hybrid Benefit to save compute costs.](#) ☐

1
Virtual machines

×

730
Hours

= \$98.55
Per month

SSL Certificate

| Product Features | | | | |
|------------------|--------------|-----------|------------------|--------------|
| | Standard SSL | EV SSL | Multi-Domain SSL | Wildcard SSL |
| 1-year Price | \$175 USD | \$295 USD | \$299 USD | \$595 USD |

Domain Name

Domain Registration

SMART DOMAIN SEARCH

Find A Great DOMAIN

Just \$9.99

→ SEARCH

Domain Registration Overview

Domain Pricing

Country Codes

Additional Top Level Domains

| Domain | Pricing | Description / Restrictions |
|--------|------------------|---|
| .aero | \$99.00 per year | .AERO is restricted to companies, organizations, and individuals recognized as members of the aviation community, holding a .AERO Membership ID, or an Aviation Community Membership ID issued by SITA. |
| .biz | \$11.99 per year | |
| .com | \$9.99 per year | |

Table – Service Costs

| Type | Rate | Period | Total Yearly Cost |
|-----------------|----------|---------|-------------------|
| Virtual Machine | \$98.55 | Monthly | \$1,182.60 |
| Azure SQL | \$4.90 | Monthly | \$58.68 |
| SSL Certificate | \$595.00 | Yearly | \$595.00 |
| Domain Name | \$9.99 | Yearly | \$9.99 |
| Total | | | \$1,836.28 |

Implementation Plan and Timeline

Taxology has a medium sized development group which includes resources from business analysis, development, project management, and quality assurance. Each product or area has its own team. Assigned resources are not shared between teams. The effort for this engagement will rely on concurrent processing of each team. While this does not shorten the magnitude of the work, it does enable the overall schedule to be crashed. For brevity, only two services are listed in the timeline. Apart from the website, tasks shown would be duplicated for each service, all following the same pattern. The website has an additional task related to user analysis as it will incorporate participatory design sessions with users regarding the interface. The timeline for the website is also greater than all the other services, as it will serve as the initiating choreographer for the entire system. Once all tasks have been completed, the system should be able to exhibit all functionality specified in the requirement documentation.

Resource assignments

| Phase | Task | Resource |
|----------------|---------------------------------------|-------------------|
| Requirements | Project Schedule | Project Manager |
| Requirements | Business Goals | Business Analysts |
| Requirements | Functional / Non-Functional Documents | Business Analysts |
| Design | All Tasks | API Developers |
| Implementation | All Tasks | API Developers |
| Verification | All Tasks except for Unit Testing | Quality Analysts |
| Verification | Unit Tests / Coverage reports | API Developers |

| | | |
|------------|----------------------------|-------------------------------------|
| Deployment | Create diagrams | API Developers |
| Deployment | Create implementation plan | Project Manager / API Developers |

Task Schedule

| Service | Phase | Task | Deliverable | Start Date | End Date |
|------------------------|----------------|------------------------------------|--------------------------------------|------------|-----------|
| | | | | | |
| Business Cross Cutting | Requirements | Project Schedule | Project WBS | 7/1/2018 | 7/6/2018 |
| | | | | | |
| Customer | Requirements | Business Goals | Business Goals Document | 7/7/2018 | 7/9/2018 |
| Customer | Requirements | Elicit Functional specification | Functional Requirements Document | 7/10/2018 | 7/15/2018 |
| Customer | Requirements | Create Non-functional metrics | Non-Functional Requirements Document | 7/16/2018 | 7/21/2018 |
| Customer | Design | Model system | Architecture Documents | 7/22/2018 | 8/4/2018 |
| Customer | Design | Create system diagrams | Architecture Documents | 8/5/2018 | 8/10/2018 |
| Customer | Implementation | Customer Console Service | Dockorized Executable | 8/11/2018 | 9/11/2018 |
| Customer | Verification | Create Traceability Matrix | Verification Documents | 7/22/2018 | 7/27/2018 |
| Customer | Verification | Create Test plan | Verification Documents | 7/28/2018 | 9/11/2018 |
| Customer | Verification | Execute Test plan | Verification Documents | 9/12/2018 | 9/22/2018 |
| Customer | Verification | Compile Unit test coverage reports | Verification Documents | 9/10/2018 | 9/11/2018 |
| Customer | Deployment | Create deployment diagrams | Deployment Documents | 9/12/2018 | 9/17/2018 |
| Customer | Deployment | Create implementation plan | Deployment Documents | 9/18/2018 | 9/23/2018 |

| | | | | | |
|---------------|----------------|------------------------------------|--------------------------------------|------------|------------|
| | | | | | |
| Taxology site | Requirements | Business Goals | Business Goals Document | 7/7/2018 | 7/9/2018 |
| Taxology site | Requirements | Elicit Functional specification | Functional Requirements Document | 7/10/2018 | 7/15/2018 |
| Taxology site | Requirements | Create Non-functional metrics | Non-Functional Requirements Document | 7/16/2018 | 7/21/2018 |
| Taxology site | Requirements | Create and compile user surveys | User Analysis Document | 7/10/2018 | 7/20/2018 |
| Taxology site | Design | Model system | Architecture Documents | 7/22/2018 | 8/4/2018 |
| Taxology site | Design | Create system diagrams | Architecture Documents | 8/5/2018 | 8/10/2018 |
| Taxology site | Implementation | Taxology Website | Dockorized Executable | 8/11/2018 | 10/11/2018 |
| Taxology site | Verification | Create Traceability Matrix | Verification Documents | 7/22/2018 | 7/27/2018 |
| Taxology site | Verification | Create Test plan | Verification Documents | 7/28/2018 | 10/11/2018 |
| Taxology site | Verification | Execute Test plan | Verification Documents | 10/12/2018 | 10/22/2018 |
| Taxology site | Verification | Compile Unit test coverage reports | Verification Documents | 10/10/2018 | 10/11/2018 |
| Taxology site | Deployment | Create deployment diagrams | Deployment Documents | 10/12/2018 | 10/17/2018 |
| Taxology site | Deployment | Create implementation plan | Deployment Documents | 10/18/2018 | 10/23/2018 |

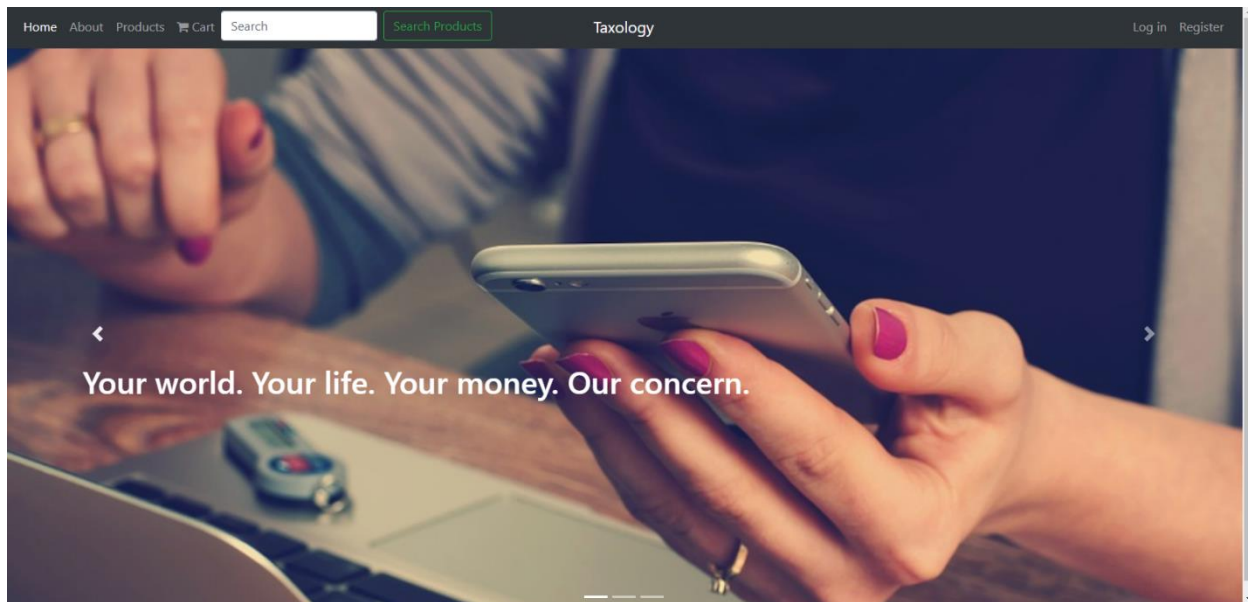
Test Plan

This test will align with one of the use cases defined in the requirements phase. The purpose of this test will be to ensure that a customer can add products to their cart and be able to view relevant information needed to make a purchase.

| Task # | Action | Expected Result | Verification means | Pass/Fail |
|--------|--|---|---------------------------|-----------|
| 1 | User navigates to Products page by clicking on "Products" in main navigation | Listing of 4 products shown | Visual assessment of page | Pass |
| 2 | User adds a product to shopping cart by clicking "Add to Cart" | Product added to cart | Visual assessment of page | Pass |
| | | | Database query | Pass |
| | | Page is redirected to Shopping Cart | Visual assessment of page | Pass |
| | | Shopping cart displays product name | Visual assessment of page | Pass |
| | | Shopping cart displays product price | Visual assessment of page | Pass |
| | | Shopping cart displays subtotal of all products | Visual assessment of page | Pass |
| 3 | User clicks "Continue Shopping" | User is redirected to Products page | Visual assessment of page | Pass |
| 4 | User adds a different product to shopping cart by clicking "Add to Cart" | Product added to cart | Visual assessment of page | Pass |
| | | | Database query | Pass |
| | | Page is redirected to Shopping Cart | Visual assessment of page | Pass |
| | | Shopping cart displays product name | Visual assessment of page | Pass |
| | | Shopping cart displays product price | Visual assessment of page | Pass |
| | | Shopping cart displays subtotal of all products | Visual assessment of page | Pass |

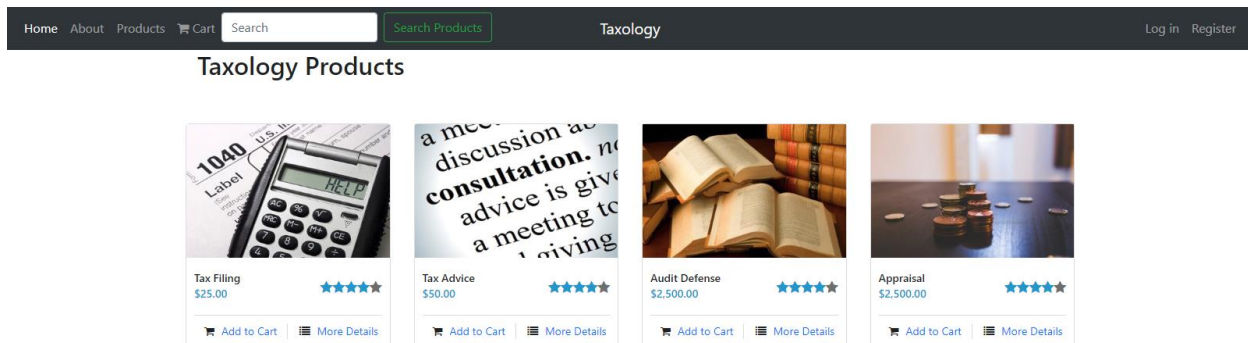
Task 1 Action: User selects product from main navigation

Figure 13. Task 1 action



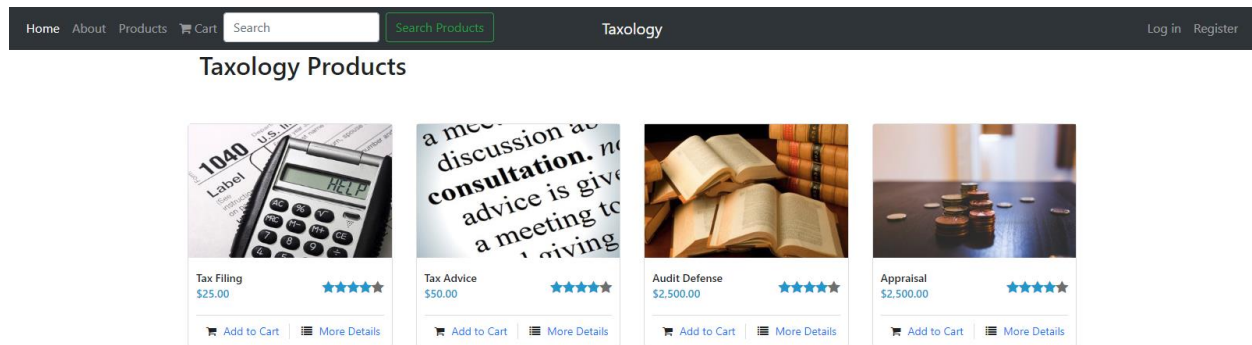
Result:

Figure 14. Task 1 result



Task 2 Action: User selects a product to add to the cart:

Figure 15. Task 2 action



Result:

Figure 16. Task 2 website result

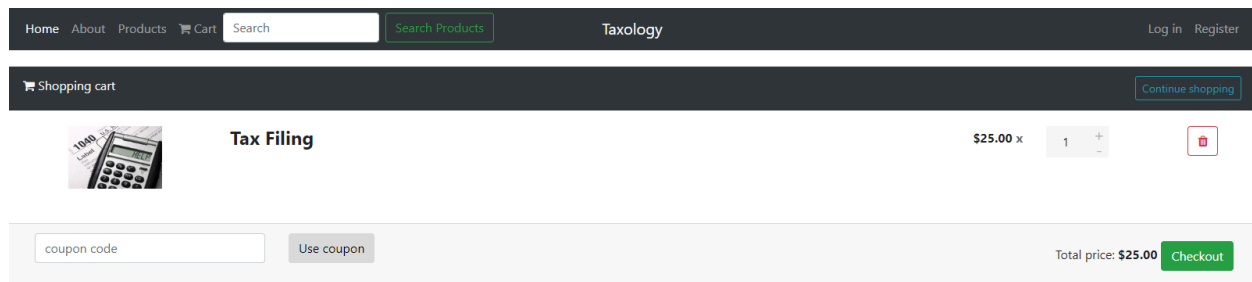
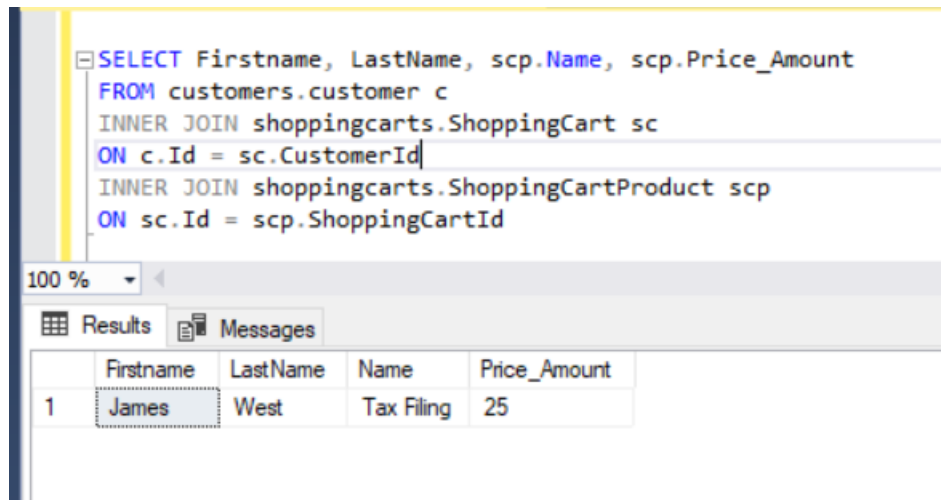


Figure 17. Task 2 database result

The screenshot shows a SQL query editor with the following query:

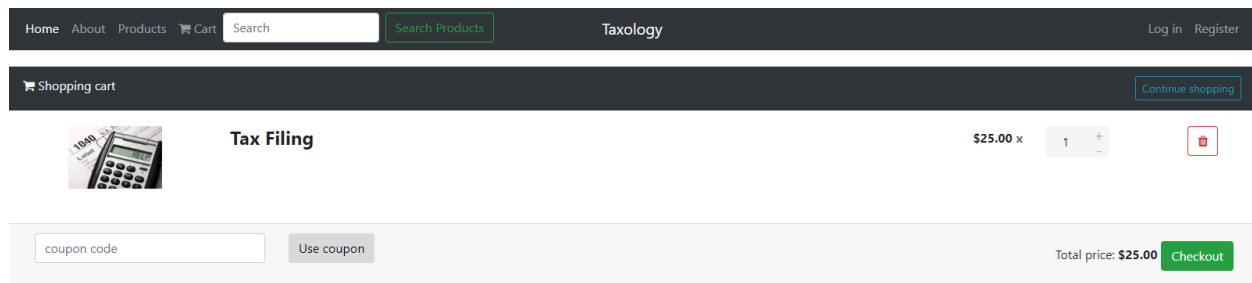
```
SELECT Firstname, LastName, scp.Name, scp.Price_Amount
FROM customers.customer c
INNER JOIN shoppingcards.ShoppingCart sc
ON c.Id = sc.CustomerId
INNER JOIN shoppingcards.ShoppingCartProduct scp
ON sc.Id = scp.ShoppingCartId
```

Below the query editor, the 'Results' tab is active, displaying a single row of data:

| | Firstname | LastName | Name | Price_Amount |
|---|-----------|----------|------------|--------------|
| 1 | James | West | Tax Filing | 25 |

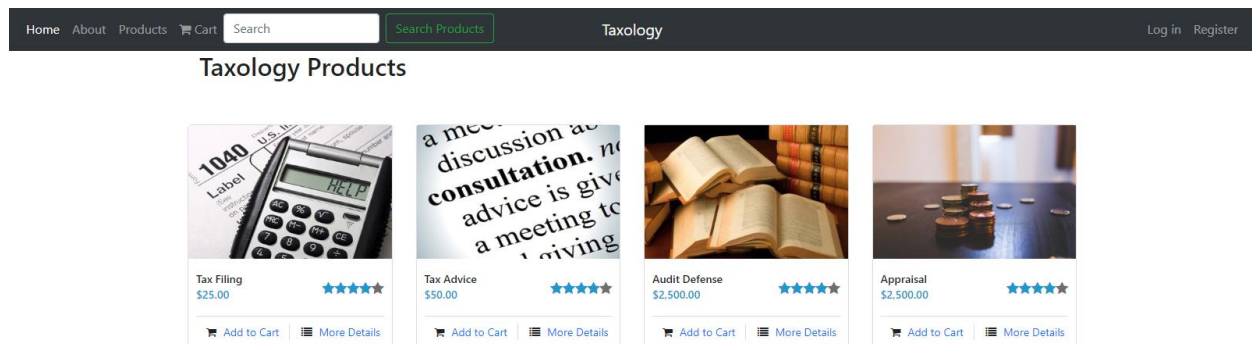
Task 3 Action User selects Continue Shopping:

Figure 18. Task 3 action



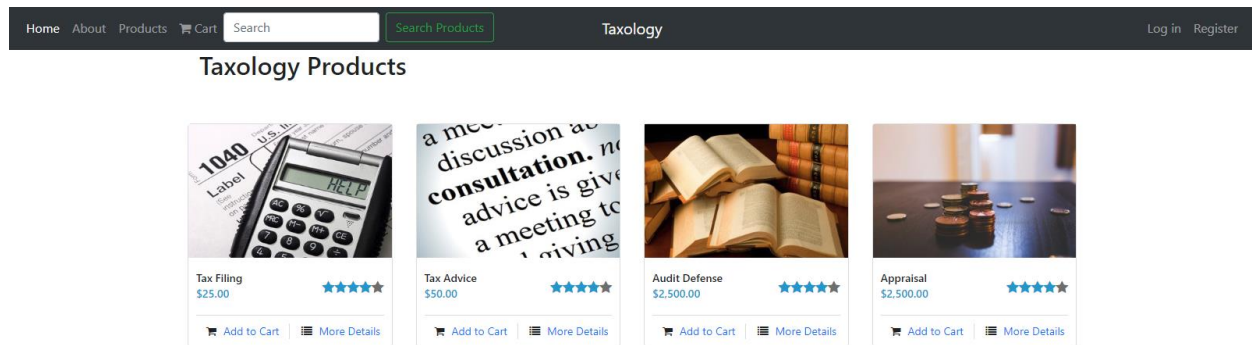
Result:

Figure 19. Task 3 result



Task 4 Action User selects an additional product to add to the cart:

Figure 20. Task 4 action



Result:

Figure 21. Task 4 website result

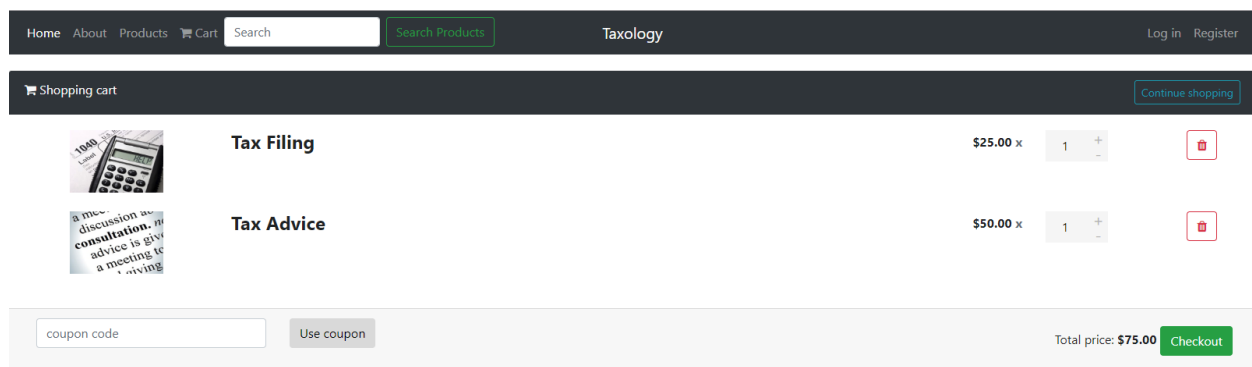


Figure 21. Task 4 database result

SQLQuery2.sql - ta...xology (mwest (73))* SQLQuery1.sql - ta...xology (mw

```

SELECT Firstname, LastName, scp.Name, scp.Price_Amount
FROM customers.customer c
INNER JOIN shoppingcarts.ShoppingCart sc
ON c.Id = sc.CustomerId
INNER JOIN shoppingcarts.ShoppingCartProduct scp
ON sc.Id = scp.ShoppingCartId

```

100 %

Results Messages

| | Firstname | LastName | Name | Price_Amount |
|---|-----------|----------|------------|--------------|
| 1 | James | West | Tax Filing | 25 |
| 2 | James | West | Tax Advice | 50 |

Unit Testing

Unit testing is performed with the help of two collaborating tools, XUnit and Resharper. XUnit provides the unit testing framework, and ReSharper provides the unit testing runtime and reporting tools. The language used for the term unit test is not always accurate. For this section, a unit test is defined as a structured, coded test that can be repeated. These types of tests are often integrated into the build process, so that results are immediately visible by a developer as he or she makes changes to the application code. A unit test proper also ideally tests the smallest unit or module available, i.e., a single method. If other classes are needed for this method to work, those classes are normally created as mocks so that their behavior is stipulated. These mocked objects just move the test along. The mocked classes are not the SUT (subject under test).

Another category within the unit testing umbrella is integration testing, which can be defined as *either* a test that crosses boundary lines of a unit (the test calls more than one class, and these

classes are not mocked up), or a test that crosses system boundaries (such as calling a database).

The latter category of tests is normally only ran infrequently due to the cost of time and possible side effects. Xunit defines two types of attributes that are helpful in creating unit tests. One is the **Fact** – by using this attribute the developer is stating that this test should always pass (unless the code is wrong). An would be:

[Fact]

```
public void TestOne(){  
    int x = 0;  
    int y = 1;  
    Assert.AreNotEqual(x, y);  
}
```

Xunit also defines the **Theory** attribute, which allows for a test to have data past into it. This data is usually auto generated and allows the test to monitor the code with various conditions. This helps identify scenarios that are better tested with a wide variety of data. Ideally the test will always pass, however, it remains a theory because there are unknown values being passed to it – Unlike the Fact, it cannot be rigorously shown that the test will pass in every condition. An example would be:

[Theory]

```
public void TestOne(int x, string yString){  
    var sut = new SomeClass();  
    int returnValue = sut.Add(x, yString);
```

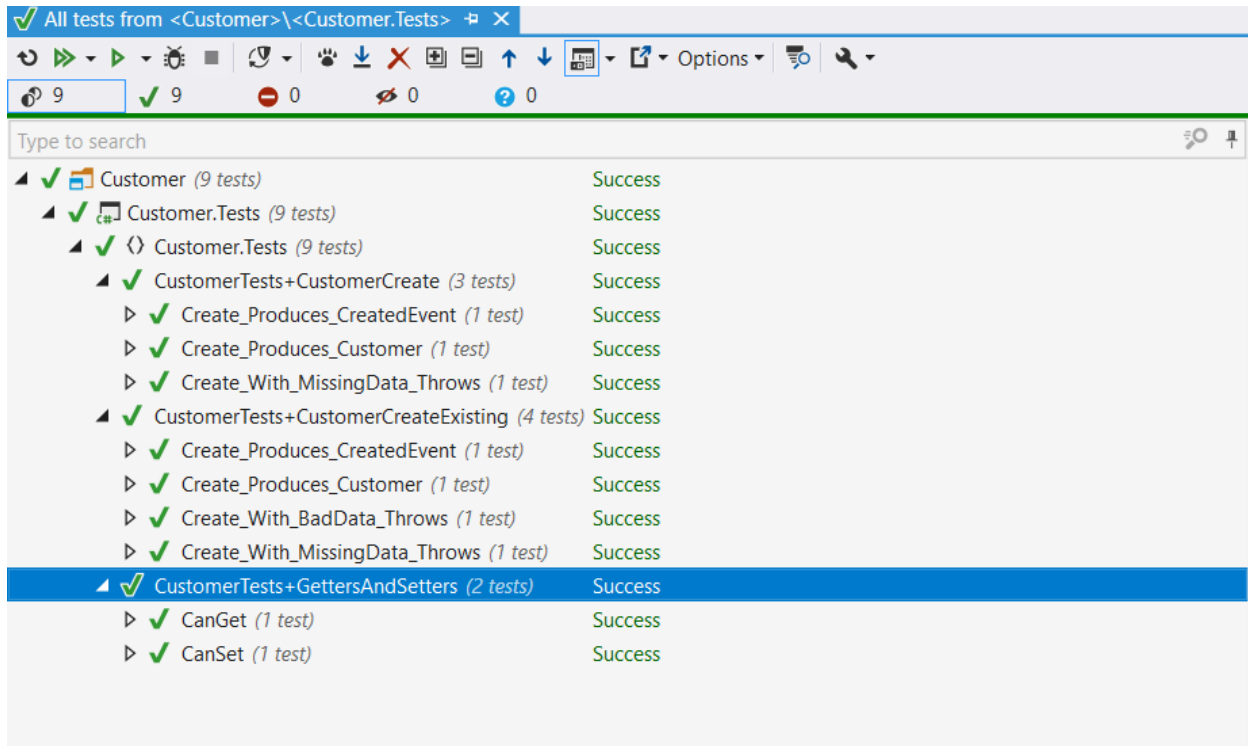
//maybe the y passed in wasn't able to be converted by the method

Assert.AreEqual(returnValue, (x + Convert.ToInt(yString)));

}

The following is a screenshot of the unit tests within the Customer.Tests project.

Figure 21. Unit test results



Developer Maintenance Guide

Introduction

This guide is to help technical teams understand what is needed for the development, maintenance, and deployment of the Taxology suite of services. Three main components are required for the solution to operate successfully. The application code, a SQL Server database, and a RabbitMQ service. The application code contains all the code necessary to develop the

website, web API, customer, order, product, and shopping cart services. As Entity Framework is used as an ORM all the SQL needed is also contained and generated by the application code. The next two necessary components are an instance of SQL Server and an instance of the RabbitMQ service. As will be detailed later in the document, various options exist in which platform these instances run on and how they are hosted.

Required installations

1. Visual Studio Professional 2017 (probable that Community edition works as well but this hasn't been verified).
 - a. Visual Studio is the IDE used to develop the application code and can be downloaded at <https://visualstudio.microsoft.com/downloads/>
2. Docker for Windows
 - a. Docker provides support for containerized infrastructure to run the application services
 - b. Additional information about Docker can be found at <https://docs.docker.com/engine/docker-overview/>
 - c. Docker for Windows can be downloaded at <https://docs.docker.com/docker-for-windows/install/>

Optional installations

1. RabbitMQ
 - a. RabbitMQ is a message broker used to provide notifications from microservices to each other.
 - b. Erlang – during the installation RabbitMQ will route the user to the Erlang website to install the Erlang runtime (if not already installed).

- c. RabbitMQ can be downloaded at

<http://www.rabbitmq.com/download.html>

2. SQL Server Management Studio (SSMS)

- a. SSMS is a tool used to access, manage, and query the database that persists the application code's state. While not strictly necessary to run the application, it is highly recommended for observational and verification purposes of by way of query database commands.

- b. SSMS can be downloaded at <https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017>

3. Azure Command Line Interface (Azure CLI)

- a. The Azure CLI is used to deploy and manage Azure resources. While most operations are available through Azure's website portal, a few are only possible via the CLI. If it is desired to deploy the application into a Kubernetes environment the CLI will be necessary.

- b. The Azure CLI can be downloaded at <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest>

4. Kubectl

- a. Kubectl is a command line tool that helps to manage nodes, pods, and clusters in a Kubernetes environment.
- b. Kubectl can be downloaded installed from a PowerShell command line, instructions are at <https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-with-powershell-from-psgallery>

Azure Components

A hosted cloud environment is used for the deployment of services, although there is no strict dependency on the Azure environment. The alternative exists to deploy the services can into another cloud provider, a data center, or local infrastructure.

Service Accounts necessary for an Azure deployment are

1. Azure Account
 - a. This is obtained by signing up with the Azure service and creating a subscription.
2. SQL Azure
 - a. Created by adding an Azure SQL Server database. The system admin account can be found by accessing the Azure Portal and navigating to the Azure SQL Server. The password is generated when creating the server and cannot be retrieved. However it can be reset if by an Azure tenant admin if lost.
3. Azure Container Service
 - a. Created by adding an Azure Container Service, the username and password are contained in the Access Keys section of the properties blade.

Optional Azure Services

1. Azure Container Registry
 - a. Used as a repository for published Docker containers. This can be used in conjunction with the Azure Kubernetes Service.
2. Azure Kubernetes Service

- a. This service can be used as a microservice orchestrator, monitoring health and managing the scale of the Taxology service.
3. Azure Virtual Machine
 - a. As an alternative to deploying in the Azure Kubernetes Service, Azure App services and WebJobs, the Taxology services can be deployed to a Virtual Machine. While this can be a cost-effective solution, it will not be as reliable without a dedicated operational plan and oversight. For cost purposes, this is how the current project is actively deployed (an instance of Kubernetes was created but dismantled afterward).

Local environment docker packages

1. Windows Nano Server 2016
 - a. This is used as the base image for all the Taxology service containers
2. RabbitMQ
 - a. A RabbitMQ container is available for local deployments, and can be retrieved by executing the following statement from a command line:
 - i. *docker run -d --hostname taxRabbit --name some-rabbit
micdenny/rabbitmq-windows*
3. SQL Server
 - a. A SQL Server container is available for local deployments, and can be retrieved by executing the following statement from a command line:
 - i. *docker run -d -p 1433:1433 -e sa_password=D0ckers! -e
ACCEPT_EULA=Y microsoft/mssql-server-windows-developer*

Debugging / Running the application

After SQL Server and RabbitMQ instances have been created and setup, the first step in running the application is to change the appsettings.json file (located in each project). This configuration file will need to be modified by entering in the appropriate connection string information for the database and message queue.

On the initial build, the solution will attempt to restore all necessary referenced assemblies via the NuGet package manager. There is no particular path in which the solution needs to be located at, but a network connection will be needed for the initial build process to download the packages.

When running the application for the first time, a network connection will also be needed as the Windows Nano container will need to be downloaded and registered with the local Docker service. No user interaction is required at this point; however, the initial run will be comparatively slower than subsequent runs. After this has taken place, each container will be generated for each service and registered with the local Docker service. Each of the solutions services will then be started. Every service creates its own database context with the Entity Framework, and on the startup of the application, checks to ensure that any database migrations are deployed to the configured database. This is how the table schema is generated for the SQL Server database. As changes are made to the domain model, migrations are created (a manual process of creating C# mapping files). This is how the SQL and code stays synchronized. Additionally, the creation of a database context will seed the database with any essential data if it does not already exist. RabbitMQ message queues and exchanges will also automatically be created during the runtime as needed without need for user interaction.

Deployment

Azure Virtual Machine Deployment

Deploying to a single VM can be achieved by publishing each project (right click on project and click Publish). By publishing to a local file profile, the executable and all dlls are created. The contents of this folder can then be transferred to the hosted VM. This file transfer can be set up by assigning a local resource to the local drive in which these files are in by way of a Remote Desktop Connection, or using the Azure Storage service to process the transfer. Once the files have been deployed, the Windows or Linux system will need to have the following ports opened via inbound firewall rules:

15672 – optional, this is used to manage the RabbitMQ service

5672 – required for the RabbitMQ transport

80 – required for the Http transport to reach the Taxology website. The web API does not need a port opened as it is not directly accessible from outside of the VM.

To run the applications, open a command line or PowerShell command line and run the following commands:

```
dotnet <path>/Customer.Console.Service.dll
```

```
dotnet <path>/Product.Console.Service.dll
```

```
dotnet <path>/ShoppingCart.Console.Service.dll
```

```
dotnet <path>/Order.Console.Service.dll
```

```
dotnet <path>/Taxology.WebApi.Console.Service.dll
```

```
dotnet <path>/Taxology.Site.Console.Service.dll
```


Kubernetes Deployment

Additional steps are listed if a more substantial and reliable hosting environment is desired. To run the following commands, open a PowerShell command window with administrative permissions.

1. Login

command:

```
az login
```

note:

follow the screen instructions to set up authentication for the current PowerShell instance by opening a web browser

2. Get the service principle for Azure Kubernetes Service

command:

```
az aks show --resource-group taxKubernetes --name taxologyKube --query "servicePrincipalProfile.clientId" --output tsv
```

output:

```
c1d57c7e-1a4b-4c64-b6f4-042789ae01e1
```

3. Get the id for Azure Container Service

command:

```
az acr show --name taxContainerService --resource-group taxContainer --query "id" --output tsv
```

output:

```
/subscriptions/0492fc37-6ea0-4dce-89d9-56a2d9ec27bd/resourceGroups/taxContainer/providers/Microsoft.ContainerRegistry/registries/taxContainerService
```

4. Create a role assignment for the AKS and ACS

command:

```
az role assignment create --assignee c1d57c7e-1a4b-4c64-b6f4-042789ae01e1 --role Reader --scope /subscriptions/0492fc37-6ea0-4dce-89d9-56a2d9ec27bd/resourceGroups/taxContainer/providers/Microsoft.ContainerRegistry/registries/taxContainerService
```

5. Push images to Azure Container registry

command: `docker tag taxcontainerservice.azurecr.io/taxologywebapi`

command: `docker push taxcontainerservice.azurecr.io/taxologywebapi`

command: `docker tag taxcontainerservice.azurecr.io/customerconsoleservice`

command: `docker push taxcontainerservice.azurecr.io/customerconsoleservice`

command: `docker tag taxcontainerservice.azurecr.io/orderconsoleservice`

command: `docker push taxcontainerservice.azurecr.io/orderconsoleservice`

command: `docker tag taxcontainerservice.azurecr.io/productconsoleservice`

command: `docker push taxcontainerservice.azurecr.io/productconsoleservice`

command: `docker tag s taxcontainerservice.azurecr.io/shoppingcartconsoleservice`

command: `docker push taxcontainerservice.azurecr.io/shoppingcartconsoleservice`

6. Create pods in Kubernetes

```
kubectrl create -f customer.console.service-deployment.yaml
```

```
kubectrl create -f order.console.service-deployment.yaml
```

```
kubectrl create -f product.console.service-deployment.yaml
```

```
kubectrl create -f shoppingcart.console.service-deployment.yaml
```

```
kubectrl create -f taxology.site-deployment.yaml
```

```
kubectrl create -f taxology.webapi-deployment.yaml
```

7. Get pods (to verify deployments)

```
kubectrl get pods
```

8. Browse Kubernetes Dashboard (to view system health and status)

```
az aks browse --name taxologyKube --resource-group taxKubernetes
```

User Guide

The Taxology user interface is devoted to the shopping cart experience. Users are free to navigate to most of the pages within without any credentials necessary. When a user does need credentials to access a page, we will prompt you for the information with a login screen.

The user can access the following pages without logging in by clicking on the main navigation links at the top of the page:

Home

This page is the main landing page for the site. As a user you can browse to this page and view any important news that Taxology is spreading and keep up to date with any current sales or offerings.

About

The about us page is here to tell you a little bit about the team that is working behind the scenes to deliver information and products to you. Feel free to stay in touch with them by following their blogs and social media outlets.

Search

If you need to find something, but not sure where to look, just locate the search field. Psst, it's at the top of every page. Enter your search criteria into the text box, click the "Search Products" button, and we will direct you to a page that lists out anything that we've found. You can either search for a product name, or for key information about products (it's short description).

Products

Users can browse through any of our listed products on this page. If a product interests you, or you have a need for one of our services, please click the "More Details" link to find out more information about the product. Current offerings will direct you to:

- A webpage telling you more about our premier tax filing service.
- A webpage letting you know how our accountants can help you with your day to day financial concerns and helpful hints to better prepare for your tax statements.
- A webpage dedicated to keeping you safe in case of an audit by the IRS.

- And a webpage for business owners who need appraisal services for valuation purposes.

If you determine that a product or service does suit your needs, be sure to add it to your cart by clicking on the “Add to Cart” link. This will save your product to the cart and transfer you to the Shopping Cart page.

Shopping Cart

In the Shopping Cart page, you can review all of your selected products, and get a rough idea of how much these products will cost (taxes not included). If for some reason you accidentally added a product by mistake no worries! Removing a product from the cart is a simple thing to do. Just click on the Red Trash Can button and it we will swiftly discard it from your cart. If you decide that you want to find another service, go ahead and click the “Continue Shopping” button. Clicking this will take you back to the Products page.

However, if you are ready to move ahead with your financial well-being, click on the Green “Checkout” button. If you have already registered and logged in, then you will be directed to the Purchase product page to finalize your order. If you haven’t logged in, then you will be temporarily redirected to the Login page where you can either log in or register as a new user. Once you’ve done this, we’ll save you a click and move you directly back to where you wanted to go so you can make that purchase.

Register

As a new user, we need to find out a little bit about you. Don’t worry your information is kept locked up safe and we won’t ask for much. However, we do need to know your first and last

name, as well as an email address and password. You will use this email and password to login to the site – so don't lose it!

Passwords will need to have the following attributes (so that no one can fake being you no matter how bad he or she want to).

- At least eight characters long
- At least one digit
- At least one lowercase letter
- At least one uppercase letter

Once all of this is filled out, click the “Register” button! Thanks for joining up with us!

Login

Here you will provide your registered email and password. The process is simple, enter the information and click the “Login” button. If you happen to get it wrong, we'll let you know. Sometimes people don't want to enter in a username and password all the time, and if that's you, we've got you covered. Just check the “Remember me?” box, and we'll save a cookie for you that will keep you from having to log in every time.

If you need to register first, click on the “Register as a new user?” link at the bottom of the page. This will take you to the Register page.

Secured Areas

After login, you can do a couple of extra things. Purchase any products from your cart and view any previous orders that you've made.

Purchase

Making an order is a simple One, Two, Three process. First, we need to get a bit of information from you so that we can mail you a bill (who uses credit cards these days? PCI compliance? That is so yesterday). Make sure to fill out your address, phone, city, state, zip, and country information. If you've made a previous order with us, we'll save you some typing and pre-fill this out based on your last order. However, if you've moved, changed names, or switched telephones make sure to update your record. After you've put this in, click the "Continue" button.

Once you've continued, we will show you a listing of your products one last time, and give you a subtotal, tax, and total amount. Check this over good to make sure that it is accurate and then click the "Confirm" button.

Confirming your purchase takes you to the final step, where we might display any information regarding fulfillment of your products. Click the Green "Purchase" button and you're all set!

Orders

As soon as you log in, a menu item will appear at the top of your browser. If you click on the Orders link, it will take you to a page that will show you all your past activity. We group these up by order, and you can find the Order #, date placed, and all the products for each order. Thanks for being a great customer!

References

Conway, M. E. (1968). How do committees invent.

Datamation, 28-31.

Newman, S. (2015). Building microservices: designing fine-grained systems. "

O'Reilly Media, Inc."

Vlissides, J., Helm, R., Johnson, R., & Gamma, E. (1995). Design patterns: Elements of reusable object-oriented software.

Reading: Addison-Wesley, 49(120), 11.

Fowler, M. (2010, March 1). Blue-Green Deployment.

Retrieved from <https://martinfowler.com/bliki/BlueGreenDeployment.html>

Garcia-Molina, H., & Salem, K. (1987). Sagas

(Vol. 16, No. 3, pp. 249-259). ACM.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial.

ACM Computing Surveys (CSUR), 22(4), 299-319.

Brewer, E. A. (2000, July). Towards robust distributed systems.

In PODC (Vol. 7).

Project Management Professional (PMP) Hourly Rate - Average Wages Per Hour -

PayScale. (n.d.).

Retrieved April 10, 2018, from

[https://www.payscale.com/research/US/Certification=Project_Management_Professional_\(PMP\)/Hourly_Rate](https://www.payscale.com/research/US/Certification=Project_Management_Professional_(PMP)/Hourly_Rate)

Business Analyst Hourly Rate – Average Wages Per Hour – PayScale.

Retrieved April 10, 2018, from

https://www.payscale.com/research/US/Job=Business_Analyst,_IT/Salary

Azure Pricing Calculator

Retrieved June 22, 2018, from

<https://azure.microsoft.com/en-us/pricing/calculator/>

DigiCert SSL Certificate

Retrieved June 24, 2018, from

<https://www.digicert.com/ssl-certificate/>

Domain.Com Domain Name Registration

Retrieved June 24, 2018, from

<https://www.domain.com/domains/>

Visual Studio Download

Retrieved June 22, 2018, from

<https://visualstudio.microsoft.com/downloads/>

Docker Overview

Retrieved June 22, 2018, from

<https://docs.docker.com/engine/docker-overview/>

Docker Install

Retrieved June 22, 2018, from

<https://docs.docker.com/docker-for-windows/install/>

RabbitMQ Install

Retrieved June 22, 2018, from

<http://www.rabbitmq.com/download.html>

SSMS Install

Retrieved June 22, 2018, from

<https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017>

Azure CLI Install

Retrieved June 22, 2018, from

<https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest>

Kubernetes Install

Retrieved June 22, 2018, from

<https://kubernetes.io/docs/tasks/tools/install-kubectrl/#install-with-powershell-from-psgallery>