Developing an E-Commerce Website Using ASP.NET Core 6

James Clabo
Dept. of Computer Science
University of North Carolina at Asheville
Asheville, North Carolina
jcclabo5@gmail.com

Abstract*—E-commerce websites are essential for businesses to market and sell their products or services online. Developing an e-commerce website requires a comprehensive approach to ensure a seamless and secure shopping experience for customers. The web application developed for this project uses modern web technologies and design practices. The application offers functionality for guests, customers with accounts, and admins, with customized home pages for returning customers based on their purchase history. Admin's can access reports on orders, products, and customers, with the ability to search customer reports. Admin reports contain selectable rows to view details, and the data presented can be sorted by column. The admin product report can be used to select a product to open in the product editor, and the product editor can also be used to add new products. Additionally, a number of security measures are implemented to protect the application and enforce least privilege. Payment processing is implemented via a Braintree Direct integration with the Braintree payment gateway. The request pipeline adds global error handling, ensures web pages are only served over an encrypted connection and cookies are transmitted securely, restricts access to pages through authentication, and redirects requests based on a lowercase rule. Parameterized SQL queries are used to prevent SQL injection attacks, and passwords are hashed using a cryptographically strong hashing algorithm. Cross-site scripting is prevented, and the View's access to data is limited to the minimum necessary.

Keywords— Braintree Direct, software development, Vue.js

I. Introduction

An e-commerce website is an online platform that allows businesses to market and sell their products or services to customers through digital transactions. They typically include features such as product listings, shopping carts, checkout functionality, and secure payment gateways. With the increasing popularity of online shopping, e-commerce websites have become an essential component of modern business strategy. Developing an e-commerce website involves a comprehensive approach to creating a seamless and secure online shopping experience. With the right platform, design, and marketing strategies, businesses can tap into a global customer base and significantly increase their revenue.

The web application was built using modern web technologies and design practices. The building blocks of the web application are HTML, CSS (Material Design Bootstrap 5), Javascript (Vue and jQuery), C#, SQL Server, and ASP.NET Core 6. The application uses both server-side rendering via Razor Pages and client-side rendering with Vue. However, each page either uses server-side rendering or client-side rendering depending on whether or not the page needs to be indexed by search engines. For example, server-side rendering is employed on the product listing page. However, when indexing is not essential the Vue Framework is used to allow for reactivity to be integrated with ease. The user interface of the application relies on the Material Design Bootstrap 5 UI kit [1] for styling and implementing responsive design, jQuery for making asynchronous requests, and Vue for conditional rendering, list rendering, event handling, and more.

The website and database are hosted at a hosting company called SmarterASP.NET, and the application is publicly accessible at shopclabo.com. The application offers functionality for guests, customers with accounts, and admins. Signed-in customers who have made a purchase in the past are provided with a customized home page based on their purchase history. Admins have access to reports, which include orders, products, and customers. Admins can interact with the reports by sorting them and viewing additional details. Viewing an order from the order report will show the order information along with the products purchased, and viewing a customer from the customer report will display an order report specific to the customer. Admin's can add new products in the product editor, and selecting edit on a product in the product report will display the product editor with the selected product autofilled. Any changes they save will be reflected in a matter of seconds. When adding new products or editing products, all supplied information is validated before it is sent to SQL Server.

All web pages included in this application are designed responsively with the intention of providing a comfortable experience on any device. The pages include breakpoints, as determined by Bootstrap 5, for phones, tablets, laptops, and larger screens, with the exception of admin reports. These breakpoints are used to make changes to the page layout as necessary to maintain a user-friendly interface and an inviting design. Admin reports are still viewable on phones, however, the admin must pan across the report to view all of the information included. The admin product editor is also responsive, and can be used to edit or add new products comfortably from an admin's phone.

In order to keep the application secure, a number of security measures are implemented. Some of which include, enforcing secure HTTPS requests via redirect, secure cookie transport, custom middlewares, storage of app secrets in appsettings.json, parameterized SQL queries, cross-site scripting prevention, cryptographically strong password hashing, and enforcing least privilege. Moreover, the app integrates with Braintree Direct to ensure transactions are secure and payment methods are kept private. Braintree Direct acts as both a payment gateway and a merchant account provider streamlining the process of integration.

II. Application Design

The web app built for this project is based on a multi-tiered architecture. The web browser is a client to the web server (IIS), and the web server acts as a client to SQL Server. The web browser makes web requests, and receives the necessary information to create interactive web pages. .NET Core 6 runs on IIS, and ASP.NET MVC services web requests. IIS communicates with SQL Server to provide back-end code access to data from queries and the privilege to initiate commands against the database.

Client-side code communicates with server-side code to retrieve data and perform various operations. The server-side code is written in C# and is responsible for handling the application's logic and managing data. The server side can validate, modify, and store data structures based on the client's needs. The client-server model separates layers of the application to service requests, display interactive web pages, and work with data.

III. Session State

The web application uses session state to identify and retain information from each user's requests. By nature HTTP is a stateless protocol meaning all requests are treated independently, such that each request is unaware of previous requests. This means that without configuring a session and storing the state, information like which items a user has added to cart, would not be available to the server on a request for the shopping cart page.

To reconcile with this, the web app relies on a .NET Core session state, which assigns a cookie to each user on their initial request and is used to identify the user on all subsequent requests. This solution relies on the nature of cookies and the role of the web browser in communicating requests. Cookies are transmitted with each request, so a cookie containing a session id allows the server to identify and locate the session state information on the server for the corresponding session. Session state information is only stored temporarily, and is discarded after a period of inactivity, which is generally set to 20 minutes. By using this approach the server is able to service many sessions in concurrency and associate the correct information with each request without involving the database. Furthermore, given that cookies are stored by the client, a user may exit out of the tab and come back before the session has ended to see that the website still has their items in the cart. However, if the browser itself is closed the cookie will be discarded whether the period of inactivity is exceeded or not. In the future, the use of cookies can be modified to overcome this behavior.

When using cookies it is essential to only transmit them over a secure connection, and to make them inaccessible to client-side scripts. In using this approach, one ensures that the data stored by the cookie is protected from malicious actors. When a web application does not configure their use of cookies this way they open up the application to attacks and leave user data vulnerable. By intercepting an unprotected session cookie a hacker could steal a user's session, by using the cookie to impersonate the user, and could gain access to their account and personal information. To prevent this, the application forces all requests and cookies to be transmitted over a secure connection, and cookies are not made accessible to client-side scripts.[2]

IV. ASP.NET MVC

The MVC architectural pattern separates application code, responsible for presentation, into three main sections: Models, Views, and Controllers. This architecture pattern follows the design principle: separation of concerns, which asserts that software should be separated based on the kinds of work it performs.[3]

A. Controller

When a request arrives to the server it is routed to a controller, which is responsible for determining the way a user can interact with the application. Controller code resides on the web server and is able to make changes to a session state based on requests. The controller receives data from the View via a request, and works with Models to satisfy those requests. Ultimately, controllers choose the View to display providing it with any data it requires via the Model.

B. View

Views are used to generate and display the user interface for the client. The logic contained on the view should only include what is necessary for display. The View contains front-end code, such as HTML, CSS, and JS. It also has access to data passed to it by the controller. In this application the View relies on two JavaScript frameworks, jQuery and Vue, as well as a CSS library, Material Design Bootstrap 5 (MDB5). JQuery is used to make asynchronous AJAX requests to the controllers, and allows the user to remain on the same page without reloading. Vue is used for conditional rendering, list rendering, event handling, and more. MDB5 provides a UI kit for Bootstrap 5 based on the principles of Material Design, and is utilized on all pages to create an inviting user interface.

C. Model

The Models used in this application are ViewModel types designed to transfer data to their corresponding Views. The ViewModels are section based, and their names correspond to the section of Views that receive data from them. An example of this is the OrderViewModel. Business logic, database access logic, validation logic, as well as any third party API logic, are encapsulated in Business Objects. Controllers can use Business Object instances to populate ViewModel instances with data to send to the View. The ViewModels used in this application contain a string property for storing JSON serialized objects to be deserialized and used securely on the View. ViewModels follow the cybersecurity principle of least privilege by only providing the View with necessary data for it to function.

D. Not included in .NET MVC

Business Objects are backend components that contain database access and validation logic as well as business logic, and they are responsible for making API requests to provide additional services. They are also designed to be portable, ideally allowing them to be transferred from one application to another without modification. For example, the Business Objects used in this web application could be

used to create a mobile application without making any major changes to the Business Objects.

V. Dependencies

This application relies on a number of dependencies to operate. The list of dependencies is included below:

- ASP.NET Core 6
- SQL Server
- Vue JS 2.x
- JQuery (AJAX)
- MD Bootstrap 5
- .NET Core Braintree package
- Braintree client SDK

VI. Middleware

HTTPS interactions consist of a request and a response. Requestors, usually browsers, send a request and wait for a response from the target. Middleware is positioned between the requestor and the target, and can directly modify the response by rewriting, redirecting, modifying the headers, or responding directly. The request pipeline processes every request to the server, and is composed of a series of request delegates, within middleware, that can perform changes to the request before and after the next delegate.[4] A visual representation is provided in the ASP.NET Core documentation and included below:

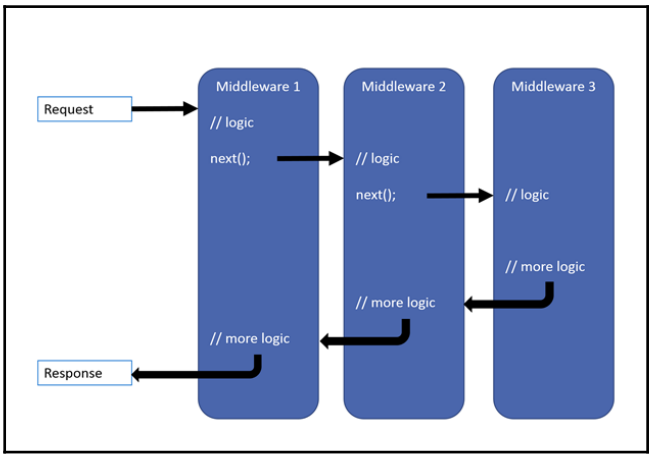

Figure 1: Request Pipeline

Source: Adapted from [4]

Some of the functionality provided by the middleware implemented in this application includes global error handling, redirecting requests to lowercase, enforcing HTTPS only, enforcing secure cookies (no client-side access, HTTPS only), redirecting unauthorized requests for admin pages back to home, and redirecting unauthorized requests for the MyAccount page to the customer login page.

As indicated above [4, Fig. 1] the ordering of middlewares determines how soon they can modify the request. An important consideration when implementing global error handling is the position in which it is added into the request pipeline. Exception handling delegates should be called early in the pipeline to identify exceptions before modification. Additionally, the .NET Core framework will catch some exceptions, so in order to ensure the custom global error handler can catch and handle the exception it must be called before system error handling. The best practice is to have the global error handler be the first request delegate in the pipeline so that it will see every request first.

VII. Security Measures

A number of security measures are implemented to protect the application and users from malicious actors.

A. Middleware and the request pipeline

1) HTTPS only via redirect: The request pipeline enforces HTTPS via redirect with a middleware built into ASP.NET. Additionally, Cookies are forced to be transmitted over HTTPS and client-side access is disallowed, protecting sessions from being stolen. However, when a computer is shared by multiple users and a customer does not log out their session could still be stolen, but not in the same way an attacker would usually steal a session.

2) Redirecting unauthorized requests: The request pipeline determines if the requested page is restricted to admins or logged in customers, and if the page is restricted validation is done before allowing access to the page. In the event a user is not authorized to access a page, they will be redirected to a different page and will not receive any information regarding the page they were attempting to access.

3) Global error handling: Through the use of global error handling, the behavior of the application in response to a malicious request is predetermined. By using certain software, any request an individual may want to send can be sent to a server without using a web browser or interacting with the user interface, and these requests can be contrived to attempt to cause the system to work in ways it was not meant to. However, this application was designed to throw exceptions on bad requests to prevent malicious actors from abusing the web apps API. By implementing global error handling these exceptions won't cause the site to stop functioning or act in ways it is not supposed to, but instead it will result in a predetermined response which will result in redirection to an error page. It is important to note that regular users would never go to the lengths of using software other than a web browser to make requests to the application or its AJAX endpoints, and the controllers are designed in a way that these errors will only be thrown by someone bypassing the user interface to make requests.

In order to keep the application running fast as it scales, not all the data sent to the AJAX endpoints is validated before being used for display, so there is a possibility that someone would send an add to cart request for a product which they altered the price and it will be added to session without the price being validated. This may seem problematic, however, no order is ever submitted without every product being validated against the database, and orders with totals that do not match the total calculated from the database will cause an exception. In the future, the application will need to handle the edge case of an admin updating a product price that a user had previously added to cart, so that these non malicious exceptions do not cause the transaction to fail. Ideally, the user will be notified that the product price has changed.

B. Secure payment gateway

Integration with credit card processing and PayPal is provided by a secure payment gateway. The web application integrates with Braintree Direct, using Braintree's .NET Core SDK to configure the payment gateway and create a secure transaction. While the application currently runs in sandbox mode, sandbox testing uses the same API feature set as the live environment, and sandbox processes behave the same as they do on production servers.[5] In order to move the integration out of sandbox mode and into production, approval is needed from Braintree, and the integration needs to be improved to handle edge cases comprehensively.

By integrating with payment processors, the fields provided for customers to input their payment data are not accessible to the application it is integrated into. The payment section is generated by the payment processor usually through an iframe or a pop up window, which allows the payment processor to collect the data and validate the payment method without letting the host application have visibility to what a customer is entering.

In order for the iframe or popup window to be generated the payment processor needs to know that the application is authorized to use their system. In order to convey this, it is necessary to generate a token on the backend using the payment gateway. This token will then be passed to the client-side and provided to the client SDK to generate the iframe. These are the first two steps in the Braintree Direct integration process, as illustrated below. [5, Fig. 1]

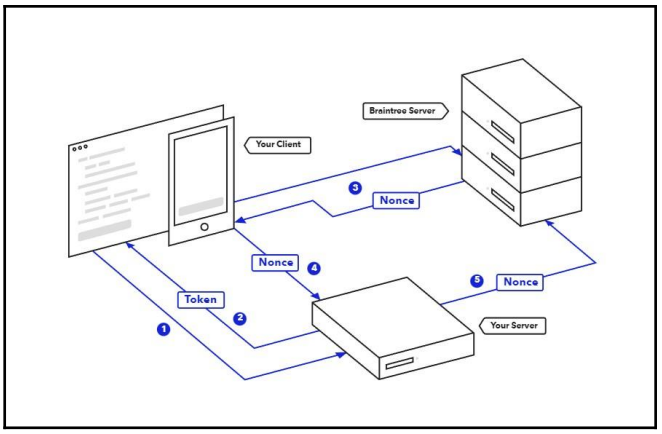

Figure 2: Payment Gateway

Source: Adapted from [5]

Steps one and two are necessary to generate the iframe the customer uses to enter their payment details securely. The client SDK is used to allow for secure and private collection of the payment method by the payment processor. Once the order is submitted with valid payment information, the client SDK communicates to the Braintree server which generates a payment method nonce and sends it to the client, as illustrated in [5, Fig. 1] number three. A payment method nonce is a one time representation of the payment method and authorized charge amount. The payment method nonce is then sent to the web server, and after being received it is used to create a transaction with the payment gateway using the server SDK, [5, Fig. 1] number four and five. The payment method nonce does not expose any of the information about the payment method, and can be stored in a database and later used to issue a refund.

It is important to mention that there are standards in place that regulate how the payment process must work, who is allowed to access and store data, and what needs to be done to meet the standard. This is referred to as the Payment Card Industry Data Security Standard. In order for this application to be used in a production environment, it must meet the requirements listed in the PCI DSS.

C. Storage of application secrets

Another security measure taken by this application is to store sensitive information inside the appsettings.json file. This file is not accessible from the web as it is located above the root. Thus, the only way a malicious actor could gain access to the application secrets is to breach the hosting companies local network to access the file. However, in the event of this happening, the admin account information is likely still protected as it is stored as a hash from a cryptographically strong hashing algorithm provided by ASP.NET Core. Best practice entails encrypting all of the application secrets which must be reversible, while cryptographically hashing other secrets which should not be reversible, such as passwords.

D. Parameterized SQL queries

In prevention of SQL injection, this application uses parameterized SQL queries also known as prepared statements. Parameterized SQL queries will pre-compile SQL code, so that attempted SQL injection strings are resolved to the input types of the columns they were assigned to on the web server and cannot be used to change the SQL query itself. Parameterized SQL queries also prevent less malicious scenarios where a user might add a single quote in the input.

E. Cross-site scripting prevention

While the application does not prevent users from inserting JavaScript into the database directly, cross-site scripting is prevented through the way Model data is accessed by the View. Given the generous length of input available for users to add an email to their account, it is possible a malicious script could be inserted into the database. However, in this application it is not possible for the script to be executed as it is extracted from a json-data attribute on a hidden span in the dom, deserialized, and added into a Vue v-text attribute for display. By doing it this way, despite rendering the code on the screen, the code is interpreted as text and will be harmless to the system. This approach is taken across the platform for extracting data from a View Model to be displayed on the View.

F. Cryptographically strong password hashing

All passwords provided by users are stored as hashes generated by a cryptographically strong hashing algorithm provided by ASP.NET Core Identity Version 3, which uses the most recent format available in Identity. The application uses the default implementation for the password hasher, which runs PBKDF2 with HMAC-SHA256, 128-bit salt, 256-bit subkey, and 10000 iterations.[6]

G. Excluding fields from JSON serialization

The "[JsonIgnore]" flag is used to prevent the default JSON serializer from serializing certain fields on the Business Objects. An example of this is the customer's password hash. This web app uses JSON serialization for storing Business Object instances in session, which allows for quick access and the transmission of Business Object data to the View as needed. The View does not need access to certain information, and it best to exclude extraneous information from the View Models as necessary.

H. Server-side input validation

All user input collected from forms is validated by a Business Object before it is added to the database. Business Object instances can store database table rows as well as additional information as needed for business logic. For example, a Customer object contains fields that correspond to the columns in the customers table, as well as a list of Order objects, and an array of input error messages.

Validation methods add error messages into the input errors array as validation checks are failed. After all validation checks are evaluated, if the input errors array is not empty, the method returns false and the Controller returns a status code of 422 along with the JSON serialized errors. The View then receives the response in an AJAX callback function and updates its Vue variables accordingly. In the event of unsuccessful validation, error messages will be reactively displayed after updating Vue data variables.

VII. Customized Product Listings

A. Algorithm Description

Product listings are customized for signed-in customers based on their purchase history. The algorithm implemented takes into consideration each of the products a customer has purchased and compares their purchase history to every other customer with an account. The customer with the highest number of matching products in their purchase history will be selected to influence the products which appear first for the signed-in customer. Products that the selected customer has purchased that the signed-in customer has not purchased will be moved to the top, with the products appearing in descending order from the most recent purchase date. The algorithm compares both active and inactive products purchased to determine the most similar customer to select, and inactive products are removed from the resulting product list to display to the signed-in customer.

The database is only queried twice: once for the list of active products and once for the list of order lines, which includes data such as the product id and quantity purchased as well as the order id it is associated with. The list of order lines is ordered by descending customer id, which allows for easy comparisons between customer purchase history. The signed-in customer is not compared to themself, and the comparisons end when a customer is found who has purchased all of the products the signed-in customer has purchased or an order line with a customer id of zero is next to be compared, signifying that all of the customers with accounts have been compared against the signed-in customer.

B. Complexity

This algorithm is implemented with a worst case runtime of $O(N^2)$ and makes the minimum number of necessary queries to the database. Although the worst case is $O(N^2)$, it is worth noting that in order for the worst case to occur, the signed-in customer would need to have purchased as many products as there are order lines in the database and none of the same products as any other customer. The bottleneck of the algorithm is the comparison between the products purchased by the signed-in customer and that of every other customer with an account.

The current algorithm could be improved to consider more than one similar customer without changing the complexity. Additionally, other algorithms could easily be implemented in place of the current one.

References

[1] "Material Design for Bootstrap 5 & 4." *Material Design for Bootstrap*, https://mdbootstrap.com/.

[2] Anderson, Rick, et al. "Session in ASP.NET Core." Session in ASP.NET Core | Microsoft Learn, 13 Feb. 2023, https://learn.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-7.0.

[3] Smith, Steve. "Overview of ASP.NET Core MVC." Overview of ASP.NET Core MVC | Microsoft Learn, 27 June 2022, https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-7.0.

[4] Anderson, Rick, and Steve Smith. "ASP.NET Core Middleware." ASP.NET Core Middleware | Microsoft Learn, 18 Apr. 2023, https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-7.0.

[5] "Braintree Direct Integration Overview." Get Started - Braintree Developer, https://developer.paypal.com/braintree/docs/start/overview.

[6] Lock, Andrew. "Exploring the Asp.net Core Identity Password Hasher." *Andrew Lock | .NET Escapades*, 24 Oct. 2017, https://andrewlock.net/exploring-the-asp-net-core-identity-passwordhasher/.