Stonehenge StockTrader Sample Application

Scenario

Stonehenge StockTrader is a web application that allows the user to buy and sell stocks, manage their portfolio, view market data, and manage their account. The Web Application itself serves as a client for services that process buy and sell orders, and provide market and configuration information. The implementation does not consider all of the dynamics involved in trading stocks, but instead simplifies the scenario so that developers from any industry might benefit without a strong background in investment banking and finance.

StockTrader is the first undertaking of the larger Stonehenge initiative which will include a collection of sample applications built to demonstrate best practices for interoperability across multiple platforms and frameworks by conforming to the defined OASIS and W3C standards.

Table of Contents

Live Demonstrations

In December of 2009, the PHP and .NET implementations of the StockTrader Client were ported to run on Windows Azure. These live cloud-based versions of the StockTrader Client can be accessed using the URLs in the table below:

Implementation	URL
PHP	http://wso2wsastest1.cloudapp.net/php_stocktrader/trader_client/
.NET	http://stocktraderazure.cloudapp.net/

Architecture

The Stonehenge StockTrader Sample Application was designed in a loosely coupled, modularized, service-oriented fashion. Its design maximizes points of integration, and thus interoperability between systems and/or platforms. This section briefly defines the various components that compose the Stonehenge StockTrader Sample Application, describes their relationships, communication, and security requirements.

Overview and Components

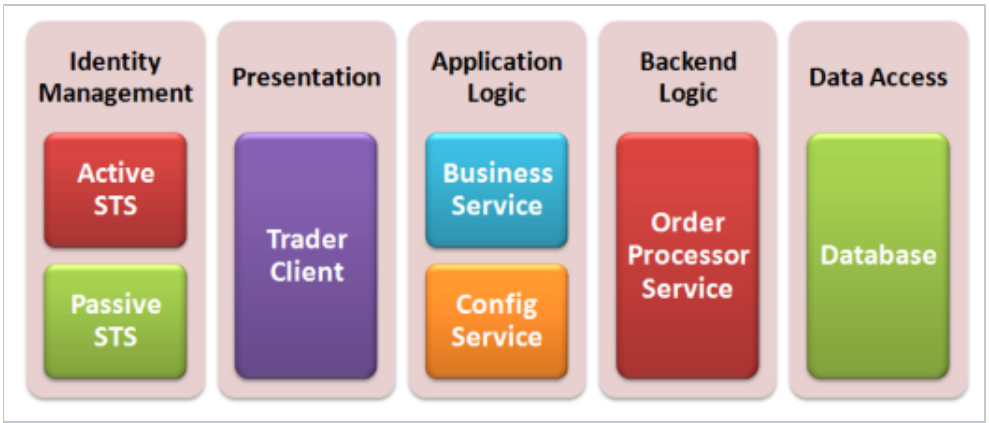

Figure 1.
StockTrader Sample Application Architecture

The Stonehenge StockTrader Sample Application components can be broken down into 5 categories. These categories are **Identity Management**, **Presentation**, **Application Logic**, **Backend Logic**, and **Data Access**. The Backend Logic category is more or less artificial inasmuch as the StockTrader Order Processor Service could well be grouped into the Application Logic category. Other components could fall under more than one category as well (e.g., the Passive STS component could be grouped under the Presentation heading). With such considerations in mind, this categorization will be used only for examination of the components here, and was not necessarily intended when the solution was designed.

Each category contains one or more components that are self-contained units that have no dependencies on a specific version of any other component. That is not to say that each component has no dependencies at all, but rather it can communicate with multiple different implementations of every other component without regard for the framework and/or platform on top of which the dependent components were designed. That being said, there are multiple different implementations of each component. Not all implementations of the StockTrader Sample Application offer all of the components mentioned (e.g., some choose only to implement the web service based components).

In the scenario for the sample, the StockTrader Client application is an application managed by an online bank that wishes to provide their clients with access to an investment account with a third party stock broker. The StockTrader Client and the Passive STS are managed by the online bank, while the remaining components are managed by the stock broker. The StockTrader Configuration Service does not fit well into this scenario, but can be excused as being simply a mechanism through which one can easily test and demonstrate different runtime configurations, and thus a component divorced from the scenario at hand.

The **Presentation** category contains the StockTrader Client application. This is the web application that allows an end user to interact with his or her investment account. This web application communicates with the StockTrader Configuration Service to gather endpoint information at runtime, and the StockTrader Business Service to access market and account information.

The **Business Logic** category contains the StockTrader Business Service and the StockTrader Configuration Service. The StockTrader Business Service is the heart of the StockTrader sample application. Nearly all interactions with the Stonehenge StockTrader Sample Application will, at one point or another, result in data flowing through this service. It communicates directly with the StockTrader Database as well as with the StockTrader Order Processor Service. It also relies on the StockTrader Configuration Service to provide information about where each of these is located.

The **Backend Logic** category contains the StockTrader Order Processor Service. This service is responsible for final processing of all buy and sell orders that flow through the system. It communicates directly with the StockTrader Database, and resolves the location of the database at runtime with the help of the StockTrader Configuration Service.

The **Data Access** category simply contains the StockTrader Database. This represents the data storage location of all market, account, and configuration data for the Stonehenge StockTrader Sample Application.

The **Identity Management** category contains two components that were added as part of M2. These components are the Passive STS and the Active STS. The online bank's Passive STS is responsible for actually authenticating users, and passing claims about the users to the StockTrader Client. The stock broker's Active STS has a trust relationship with the bank's Passive STS and willingly translates the bank's claims about the user to claims that the stock broker's services will require for authorization purposes.

Communication and Process Flow

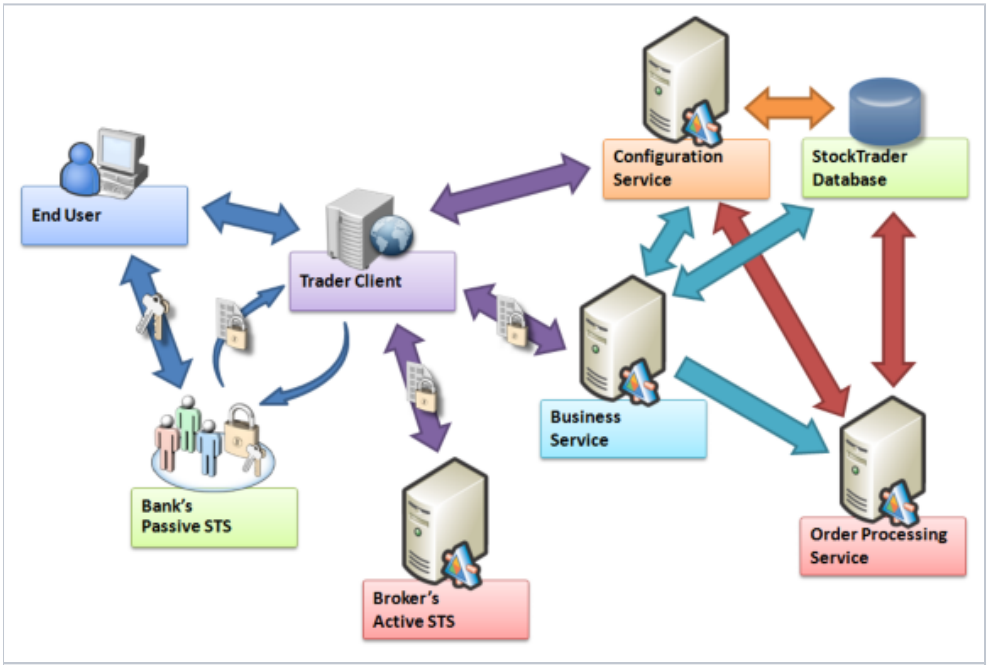

Figure 2.
Possible Communication within StockTrader Sample Application

This section describes a typical sequence of interactions that might be encountered while using the Stonehenge StockTrader Sample Application. It represents multiple requests by the user in a single session that interacts with all of the components within the solution.

1. The user accesses the StockTrader Client page for the first time, and navigates to their account information. Since the user has not yet been authenticated, the StockTrader Client redirects the user to the Passive STS. The Passive STS is a completely different web application that is only concerned with authenticating users, and nothing more.
2. The user enters his or her credentials into the web page presented by the Passive STS, and then clicks a button to continue what they were doing. The information is validated and issued an XML-based SAML security token that contains claims about the user. In the case of the StockTrader sample, the only claim is their userID. This token is POSTed back to the StockTrader Client site and temporarily stored for later use.
3. Now that the user has been authenticated, the StockTrader Client needs to determine where the StockTrader Business Service is located. In order to do this, it makes a request to the StockTrader Configuration Service to find the correct endpoint.
4. Next the StockTrader Client needs to actually gather account information to return to display to the user. Since the Passive STS is managed by the bank, while the investment account is managed by the stock broker, the StockTrader Client must get a new security token that makes sense to the stock broker's systems. To retrieve this new token, the client sends the user's stored token to the broker's Active STS.
5. The stock broker's Active STS verifies that the token is from a trusted source, and translates the claims to those claims that are required by the StockTrader Business Service, and creates and signs a new token with these claims. It returns this new token to the StockTrader Client.
6. The StockTrader Client now has all of the data required to call the StockTrader Business Service. It requests the user's account information from the Business Service, and includes the token issued by the broker's Active STS.

7. The StockTrader Business Service receives the request, and then queries the StockTrader Configuration Service for information about where the database is located.
8. Once the StockTrader Business Service has the location of the database, it looks up the account information and returns it to the StockTrader Client.
9. The StockTrader Client can now render the information to the user.
10. The user wishes to sell one of his or her stocks, and initiates a trade using the user interface presented by the StockTrader Client.
11. The StockTrader Client sends the request to the StockTrader Business Service. It uses all appropriate tokens and configuration already described. The call to the Passive STS is no longer required, as the client has temporarily stored the token.
12. The StockTrader Business Service receives the order and forwards it to the StockTrader Order Processor Service asynchronously. It immediately responds back to the StockTrader Client with details about the order. Just like before, the endpoint information for the StockTrader Order Processor Service was resolved at runtime through the StockTrader Configuration Service.
13. The StockTrader Order Processor Service receives the order, and then updates the appropriate tables in the StockTrader Database. Just like before, the database location was resolved at runtime through the StockTrader Configuration Service.
14. The user wishes to check on the status of the order, and accesses his or her Account page within the StockTrader Client.
15. The StockTrader Client calls the StockTrader Business Service for a list of "closed" orders. This assumes the same security and endpoint resolution steps already described.
16. Upon retrieving the "closed" orders from the StockTrader Database, the StockTrader Business Service marks them as "completed". These orders are sent back to the StockTrader Client.
17. The StockTrader Client can now render recently completed orders to the user in the form of "Trade Alerts".

> **Note**
>
> *These interactions may not be obvious within the code of some implementations, as they build on top of frameworks and platforms that automatically handle much of this communication.*

The swim lane diagram in **Figure 3** shows a subset of the interactions desribed in the process above. It includes only those steps up to the point that the order is successfully processed by the StockTrader Order Processor Service, but not to the point that the end user has received a "Trade Alert".

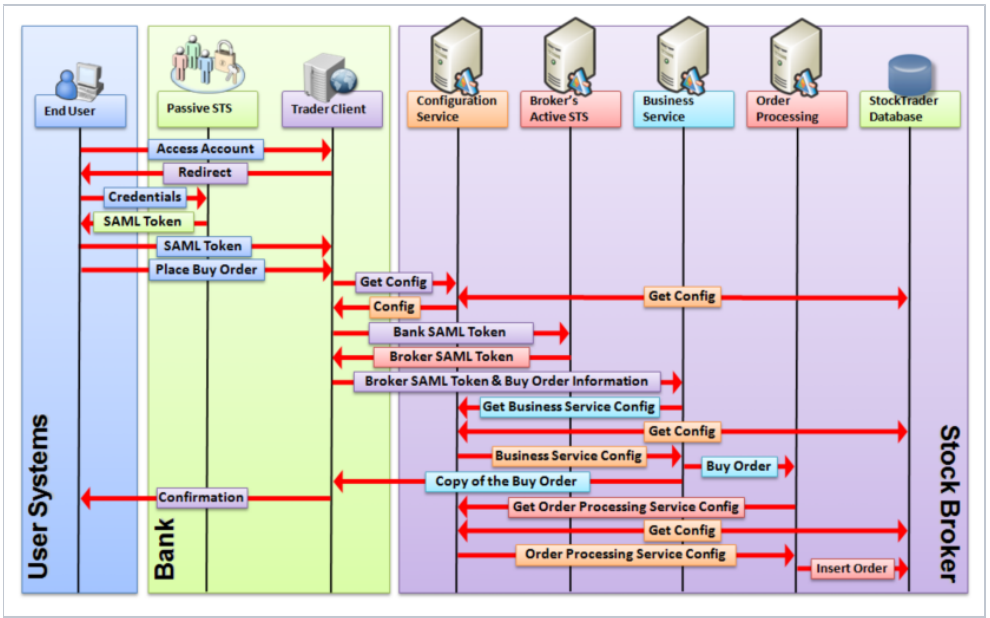

Figure 3.
Swimlane diagram depicting the process of placing a buy order within the StockTrader Sample Application

Security

In the M1 Release, the StockTrader Client handled authentication and authorization internally. Beginning with M2, users are authenticated through the bank's Passive STS, which issues claims to authenticated users which include their StockTrader userID. Before any additional service calls are made, the claims are passed by the StockTrader Client to the broker's Active STS which translates these claims into a new set of claims in a new token that the broker's services understand. This new token is used in subsequent service calls from the StockTrader Client. This approach effectively decouples authentication from any client-side logic, and allows a user's identity to be managed by an external entity, as opposed to the same entity that manages their investment accounts.

Beyond the client, communication between the StockTrader Business Service and StockTrader Order Processor Service is encrypted and signed using a shared certificate.

Security in M2 relies on frameworks that implement the WS-Security, WS-Trust, and WS-Federation standards.

History

Stonehenge StockTrader began its life as two separate projects by two industry vendors. Microsoft created the .NET StockTrader in response to an IBM sample application with a similar feature set. It was used primarily as a performance benchmark between the platforms each offered. Later, WSO2 developed a full suite of Stocktrader samples that offered full interoperability with the .NET StockTrader. Both Microsoft's .NET StockTrader and WSO2's StockTrader implementations continued life in the Stonehenge project as code donations to jumpstart the project.

Implementations

- **Metro StockTrader** (Sun Metro & JSF)
  - Installation Guide
  - Documentation
- **.NET StockTrader** (Microsoft WCF & ASP.NET)
  - Installation Guide
  - Documentation
- **PHP StockTrader** (WSO2 WSF/PHP)
  - Installation Guide
  - Documentation
- **WSAS StockTrader** (WSO2 WSAS)
  - Installation Guide
  - Documentation

Components

- **Identity Management**
  - Active STS
  - Passive STS
- **Presentation**
  - Trader Client
- **Application Logic**
  - Business Service
  - Configuration Service
- **Backend Logic**
  - Order Processor Service
- **Data Access**
  - Database

Learn & Explore

- StockTrader Interoperability Guides