System Design Document

UMBC Market

**Client**

Abhay Kashyap

**Team 4**

Cory Ferrier

Sam Leung

Seth Jenkins

Wesley Chiou

Zachary Robinson

UMBC Market

System Design Document

#### **Table of Contents**

Page

## 1. Introduction

* 1. Purpose of This Document
  2. References

1. System Architecture

2.1 Architectural Design

2.2 Decomposition Description

3. Persistent Data Design

5.1 Database Descriptions

5.2 File Descriptions

1. Requirements Matrix

Appendix A – Agreement Between Customer and Contractor

Appendix B – Peer Review Sign-off

Appendix C – Document Contributions

1. **Introduction**

1.1 Purpose of This Document

The purpose of this document is to describe the design of the UMBC Market application. Key topics covered in this document include the high level system architecture, lower level class designs, and the persistent data design of UMBC Market.

1.2 References

1. UMBC Market Systems Requirements Specifications
2. UML Distilled, 2nd edition, Martin Fowler and Kendall Scott, 2000
3. **System Architecture**

2.1 Architectural Design

Overview.png

The UMBC Market application will be built using the Meteor Javascript web framework. The meteor framework is loosely broken down into collections, components, and Templates and follows a Model View Controller architecture (MVC).

The collections layer consists of the MongoDB Database and contains collections of items with class like-properties, this is exclusively on the server side.

The information from these collections is passed down to components which consist of a series of functions, some on the server some on the client side, which provide functionality such as editing, updating, deleting, and inserting new collection items. Components have functions and properties that can be run/accessed on the client, the server, or both. Meteor detects where the code is running and processes the file accordingly.

This functionality is accessed through the templates, which handle the client rendering of the information provided by the components.

2.2 Decomposition Description

System FLow.png

In the Meteor Framework, the client requests a series of objects from the server and provides a filter object that the server can check against, the server then checks it’s database against the filter and returns whatever it determines the user is authorized to see. The client then passes this information to the templates for rendering.

In our case, we will be primarily fetching message chains, sets of messages representing a ‘conversation’ between two users, and product/service listings.

An important feature of meteor is that only talks to the server when necessary. This should only be when an action requires the database be updated, data needs to be fetched, or an action requires authentication.

Data Updates.png

The above demonstrates the overall process when performing an action that will update the database. For our application, this means Add/Changes/Deletions in regards to Messages, Users, and Listings.

The change is requested by the client with the relevant information attached, the server determines if the action is valid and returns an error if it’s not, it then attempts the requested action on the database and returns an error if the action fails, otherwise it returns an updated set of information depending on what was changed. The client then updates the display based on the result.

1. **Persistent Data Design**
   1. Database Descriptions

All Database objects are kept in a MongoDB instance that comes packaged with Meteor. The schema for the Users database is modeled below:.

DB Objects.png

1. **Requirements Matrix**

|  |  |  |
| --- | --- | --- |
| **Use Case** | **Requirement** | **System Component** |
| 1 | Account Registration | Registration interface |
| 2 | Account Login | Login Interface |
| 3 | Listing a product/service | Product Listing button/interface |
| 4 | Edit/Remove listing | Editing/remove button |
| 5 | Listing complete | Complete button |
| 6 | Searching market | Market search button and backend database |
| 7 | Contacting seller | Messaging component |
| 8 | Managing messages | View, delete, reply buttons |

**Unified Modeling Language (UML)**

**Class Diagrams**

Reference: **UML Distilled**, 2nd edition, Martin Fowler and Kendall Scott, 2000

Note: The information below has been modified slightly to meet the purposes of CMSC 345

Symbols:

Class – Represented by a box as follows:



Navigability – Represented by a solid line with an arrowhead at one or both ends (unidirectional or bidirectional association, respectively). Indicates the direction(s) of an association between classes. (Below, A can navigate to B, but not the reverse.)



Generalization – Represented by a solid line with a hollow arrowhead at one end. Indicates that one class is a generalization of another. (Below, B is a generalization of A.)







Composition – Represented by a solid line with a solid diamond at one end. Indicates that an instance of one class is “owned” by a single instance of another. (Below, an instance of B is owned by a single instance of class A. Note: if an instance of A is deallocated, the associated instance(s) of B are also deallocated.)



Dependency – Represented by a dashed line with an arrowhead at one end. Indicates that one class depends on the interface to another class. (Below, an instance of A is dependent on the interface to an instance of B.)



Multiplicity – Indicates how many instances of one class type are associated with another class. (Below, 1 to 5 instances of class A are associated with class B.)







Note: We will not be using any other symbols in our class diagrams.

Format for class attributes:

*visibility name* : *type* = *defaultValue*

where *visibility* = + for public, # for protected, - for private

*name* = attribute name

*type* = data type

*defaultValue*  = default value

Format for class operations:

v*isibility name* (*parameter-list*) : *return-type*

where *visibility* = + for public, # for protected, - for private

*name* = operation name

*parameter-list*  = comma-separated parameters with the syntax

*direction name* : *type* = *defaultValue*

where *direction*  = in for input

out for output

inout for input/output

*name* = parameter name

*type* = parameter type

*defaultValue* = default value

*return-type* = return type