Interaction Specifications *for* "Capital Games"

Report 2: Part 1
Software Engineering
14:332:452

Team 2:

Jeff Adler
Eric Cuiffo
Nick Palumbo
Jeff Rabinowitz
Val Red
Dario Rethage

March 3, 2013
Version: 1

Contents

1 Class Diagrams and Interface Specifications

2 System Architecture and System Design

2.1 Architectural Styles

Capital Games was designed to conform with several well-established software design principles. Some were chosen because of the software technologies employed (ie an MVC-based web framework), others represent a natural evolution of the needs of the system.

Model-View-Controller

Our philosophy in designing our website is to maintain a separation between the subsystems responsible for maintaining user information and those responsible for presenting it, in comformation with modern software engineering practice.

Therefore, we employ the Model-View-Controller (MVC) architecture pattern. In MVC, a View requests from the model the information it needs to generate an output; the Model contains user information; and the Controller can send commands to both the views and the models. [1]

This approach has made site design easier, by abstracting the interface specifications from the system responsibilities. The Views and Models each know only what they need, while the Controller and associated subsystems perform all the "business logic". The only complexity added by the decision to employ MVC is that updates to system components often have a ripple effect and require numerous modifications elsewhere in the system.

Representational State Transfer

As a well-designed web application, Capital Games conforms with the universal practice of employing RESTful design principles. RESTful design dictates, amongst other constraints, that a platform have a client-server relationship with the user (see below), that the interface is uniform, and that all information necessary for a request can be understood from the request sent to the server. [2]

We strive to keep the interface as uniform as possible so that it is clear to the user how he is interacting with Capital Games, on a multitude of levels. For example, when purchasing a group of stocks, a user may graphically "click on" a submit button for a certain order, but in effect he is also submitting an HTTP POST request with appropriate form data to the Orders resource.

This identificaiton of resources creates a tradeoff. On the one hand, all RESTful architecture must be designed at once, so that all resources are identified simultaneously, and the state transfers are possible to each of them. On the other hand, once resources are properly identified, the distribution of responsibilities is trivial for every possible interaction.

Data-centric

As a financial trading platform, Capital Games revolves around user data. To simplify access to that information from a variety of systems and to organize the data coherently and with the possibility of rapid retrieval, we eventually store all user data in a relational database. In this way, advanced queries can be performed on sets of data, both in application layer logic as well as by database administrators. Additionally, storing user data outside of a particular program's memory space enables subsystems which exist outside of the current application layer to also have access to the data. This additionally presents greater flexibility in terms of scaling site infrastructure.

Client-Server

By its definition as a web application, Capital Games follows a client-server model. The client, a user, interacts with the server, the various systems encapsulated by Capital Games.

2.2 Identifying Subsystems

As Capital Games exists as a website, a natural divison of subsystems arises: front end and back end. Front end essentially describes all the computations and objects that exist on the user's side of interaction with our application, and back end describes all the computations and objects that exist on the server's side. It is exceedingly simple to determine which parts of our system belong in the front end in the back end. We will also define another subsystem called "External" which will contain all the pieces necessary to our application but not technically a part of it. A high-level view of our system in the form of "packages" or subsystems follows on one of the next few pages.

As it turns out, we can go deeper into our system to define subsystems within the back end. Though the front end is relatively simple, the back end of our system is where most of the computation and interesting events occur. There are two major subsystems as have been described in previous sections of this report: financial data retrieval and the queueing subsystems. In addition to these two subsystems, the database and the controller exist within the back end, but it does not seem appropriate to further include them in another subsystem, as they are essentially separate, stand-alone packages that interact with or call upon the other packages within the system.

The financial data retrievial subsystem is the simpler of our two subsystems. It only requires the ability to handle requests given to it by the controller (requests ultimately generated by a user) and the ability to fetch data from Yahoo! Finance in response to a valid request. The queuing system is only somewhat more complicated, needing background processes to monitor outstanding tasks, an Action Mailer object to handle sending e-mails to users, and an order handler that can understand and process orders. Though the controller facilitates all their interactions with the rest of the system, these two packages dominate most of our application design and are the backbone of its functionality.

2.3 Mapping To Hardware

When it comes to web design, there is a standard on how hardware is mapped. All front end parts of the system run on the user's machine (be it a computer, tablet, or smart phone), and all back end parts of the system will run on a server owned by the developer or the developer's company. This follows from the architecture of the web, and there is really no way to deviate from it. To clarify the hardware mapping of our system, a diagram is included within the next few pages.

2.4 Persistent Data Storage

As described previously, Capital Games is data-centric and therefore cannot exist without a robust mechanism for persisting user data between "uses" of the system.

Various methods exist for the persisting of data. They range from serialization of system state variables into files to storing all data in the form of elaborate relational database systems, and anything in between. Capital Games primarily makes use of relational and non-relational databases.

In a relational database, stand-alone sets of data are placed into indexed tables stored in computer memory, with each row in a table representing a single set and each column representing a single attribute. A table can have a very large (sometimes even infinite) number of columns and rows. A database possesses many interrelated tables which are cross referenced by their indices (called primary and foreign keys). These relations allow complex queries against tabulated data. [3]

For example, consider the figure shown previously, repeated here. A league is the most fundamental data structure of Capital games, yet it is not aware of its users, their portfolios, or of any orders associated with them. To retrieve these data, a query can be performed which pivots around the indices relating the tables. To find out a list of users participating in a league, one could take that league's index (not shown in figure for brevity) and search for all investors with that index; then take the user indices contained in the resulting query and dereference them to identify the original users.

Additionally, Capital Games makes use of a so-called "NoSQL", or non-relational database. The format of such a database is practically unrestricted, and data need not belong in tabular format. This approach exchanges speed and scalability for querying power. [4] Capital Games utilizes the Redis NoSQL database to store outstanding queued jobs. This structure was chosen so that the database store can be seamlessly scaled across many machines if need be, and because of its light weight.

2.5 Network Protocol

Though it is not necessarily an interesting topic to discuss for our project, it is none the less important to take note of. Because Hypertext Transfer Protocol (HTTP) is the predominate communication protocol distributed throughout the internet, it is critical that our website relies on it to make requests and send information between our user and system. Really, there is no other option if we desire Capital Games to be successful. HTTP is already a strictly and well defined

protocol; for a description, see this reference.

2.6 Global Control Flow

Execution Order

In general, our system is event-driven in terms of execution. As far as the user is concerned, our server sits and waits for a request to be made by a user accessing some part of our website. Though this is a simplification of the actual model, it is a good description of the general order of events within our system. The users can, nearly in any order, access different parts of our websites, search different companies, place different orders, etc., at their will. Any of these actions generate a request to our server, which then creates the necessary views, enacts the necessary computations, and takes any other necessary actions to facilitate the request.

To some degree, however, there are some procedures that drive our system as well, which force users to experience certain things in a predefined order. I will identify a few of these procedures hence:

- Registration: Before any user can begin browsing our site and joining leagues, they need to make an account.
- Order placement: Before a user can place an order, they need to join a league.
- Tutorials: When a tutorial is initiated, each user will experience the tutorial in the same order as all other users, excepting them terminating the tutorial prematurely.

However, on the whole, our system is still definitively an event-driven one.

Time Dependency

Real-time is very important to our system, though it does not entirely define it. While the user browsing our website is a real-time experience, there are a lot of back-end computation and processing that occur on our server based on real-time timers. In addition, as our system is strongly reliant on the stock market, which has certain times of operation, real-time matters quite a bit. I shall identify the timers present in our system:

- E-mail Timer: Based on the user's set preferences, they can receive periodic e-mails from our system describing their portfolios' progress over the last period, which can be set to daily or weekly.
- Market Open and Close: The stock market is only open and closed during certain times of the day, so our system must rely on these times to limit the placement of orders by users.
- Resque Process Check: As described earlier in our report, many of our system's tasks are carried out by a queueing subsystem. In short periods, this queueing process must check if there are any outstanding tasks to operate upon. The period is as yet defined, but will be chosen for a balance between ensuring quick execution and reasonable server load.

Concurrency

There is a bit of concurrency within our system. Outside the main stream of execution with potentially parallel gets and posts from users' browsers, this concurrency occurs mostly within the queueing system earlier mentioned. It is relativey simple; there are persistent processes that handle order processing and e-mail updates. As these are entirely separate functions, there is no need for synchronization between these two threads of control. Synchronization between these threads and the rest of our system (i.e. the user interactions with the browser and the browser's interactions with the controller) to ensure that no data is being altered by separate entities at the same time is enacting through Ruby's including protection functions–mainly flock (file lock).

2.7 Hardware Requirements

The hardware requirements for Capital Games are minimal on the client side, and moderate on the server side.

Internet Connection

The server needs to have an internet connection. Because all data are transmitted as text, it is technically possible for the server to function on even a low-bandwidth connection. Obviously this is not ideal and low bandwidth can increase server latency during peak use hours.

Disk Space

Under the current configuration, Capital Games does not commit any additional resources to the server's disk storage during runtime. Rather, all data are stored to memory, and only backed up to the disk. Therefore, the disk requirements for Capital Games is simply the sum of the storage occupied by all program instructions for the system, or approximately 1GB at the time of this writing.

System Memory

Because all runtime data are stored to the server's memory, as well as the space in memory occupied by the actual system runtime, having a large amount of "headroom" is vital to the performance of the application. Although it is hard to analyze performance requirements of an application that is still in active development, empirical evidence from users of similar technology make a few key observations. First, the amount of memory consumed by an idle application can work out to be over 100MB. Next, the active application will load copies of its database-stored information into memory in order to operate over it, which can result in large spikes in memory usage. Finally, operating over the loaded data itself can consume a large amount of memory. This is in addition to any memory occupied by the databases and worker processes. [5] Therefore, having at least 200MB should be the minimum required for internal testing of our application. Obviously, increasing user base will exponentially increase the memory requirements of our application.

Client-side Hardware Requirements

The user needs to have an internet connection in order to interact with the server remotely. Although the intended use of the system entailing the use of a graphical web browser strongly encourages the use of a monitor (as mentioned previously, the responsive nature of the application means

that screen resolution is not a limiting factor), it is also possible for technically proficient users to interact with the server through its RESTful resources. At some future date, we may publish the official RESTful API for Capital Games, but at this point, interacting purely through a command line interface is discouraged.

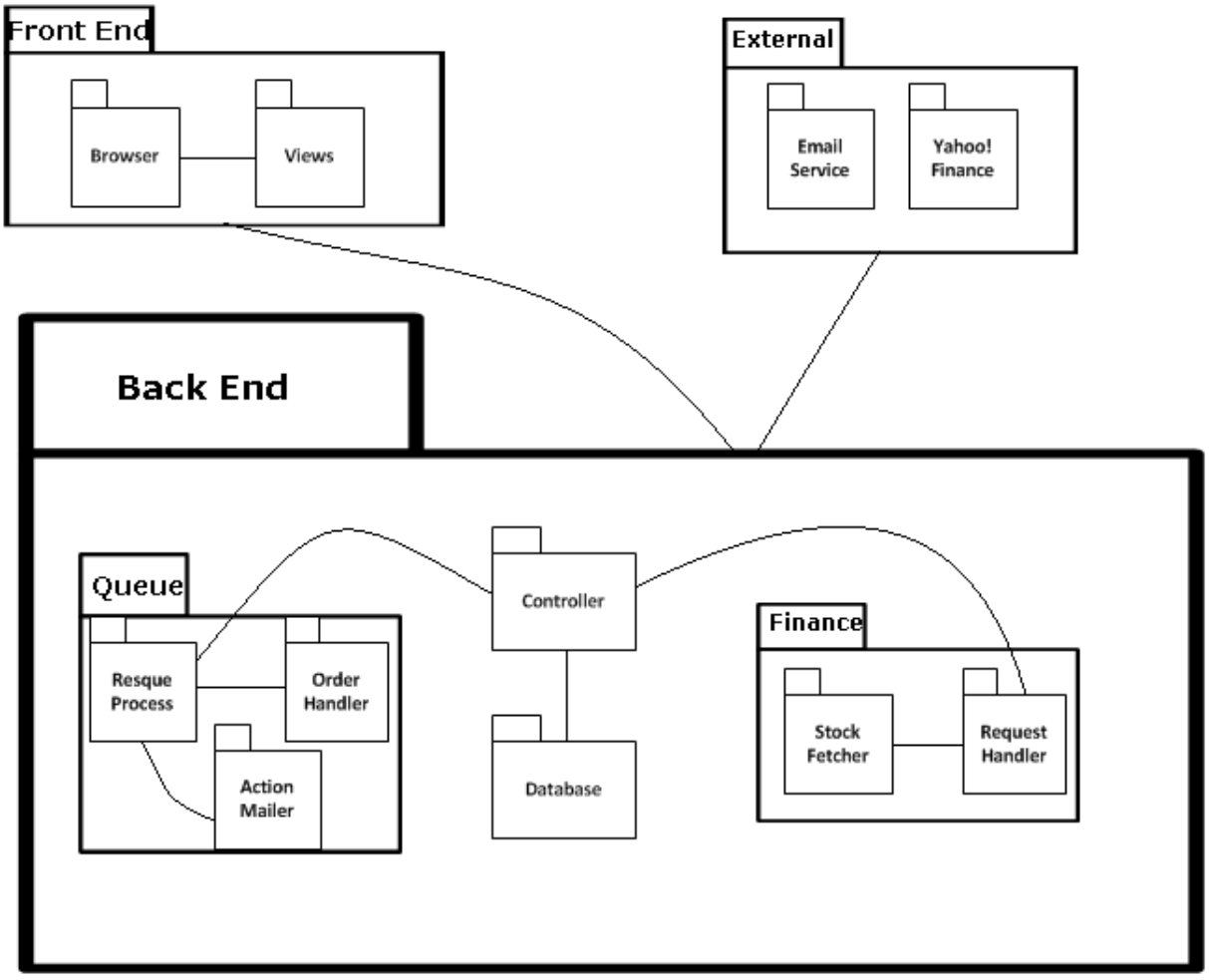

Figure 2.1: The UML package diagram for our system.

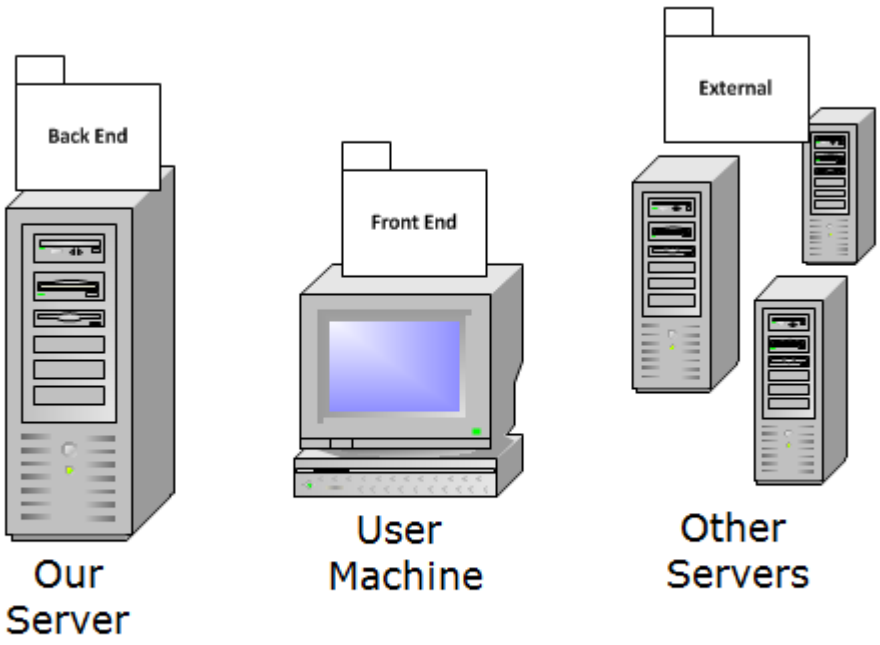

Figure 2.2: The hardware mapping for our system.

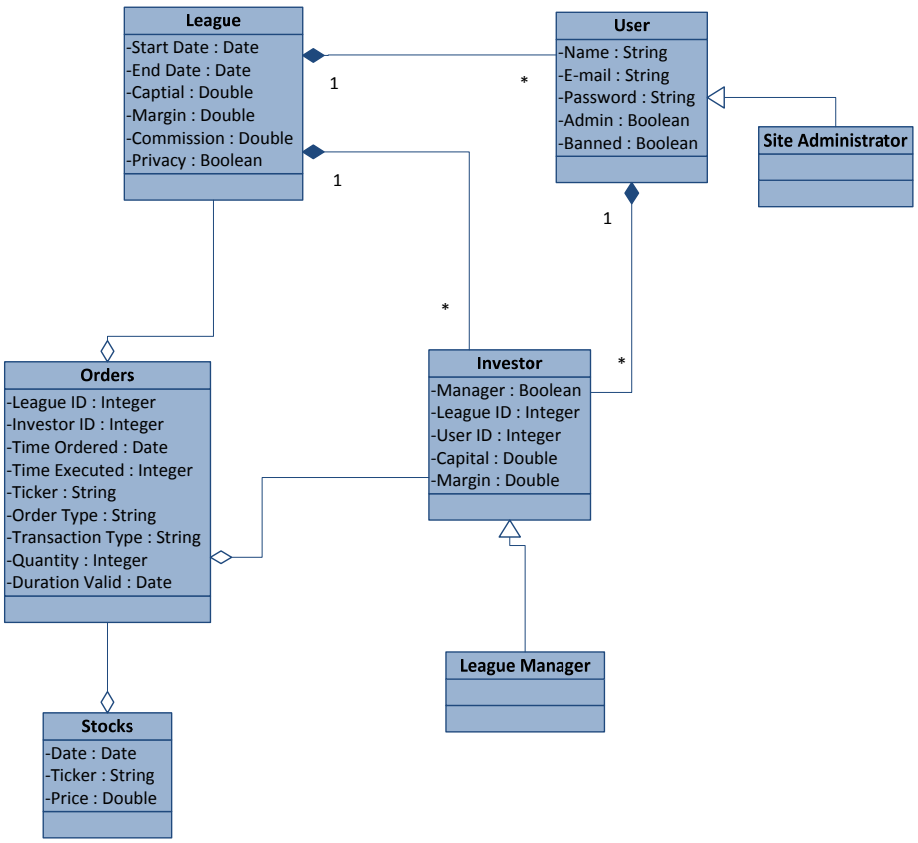

Figure 2.3: The format of the relational database schema implemented by Capital Games for its core features. (Not comprehensive.)

References

[1] Wikipedia, "Model-view-controller - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Model-view-controller. [Online; accessed 10 March 2013].

[2] Wikipedia, "Representational state transfer - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Representational_state_transfer. [Online; accessed 3 March 2013].

[3] Wikipedia, "Relational database - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Relational_database. [Online; accessed 10 March 2013].

[4] Wikipedia, "NoSQL - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/NoSQL. [Online; accessed 10 March 2013].

[5] D. Collective, "How much memory should a ruby on rails application consume? - Stack Overflow." http://stackoverflow.com/questions/2971812/how-much-memory-should-a-ruby-on-rails-application-consume. [Online; accessed 10 March 2013].