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 System Interaction Diagrams

1.1 Introduction

Following is an analysis of the interactions of the two most important internal subsystems in our system as identified in our domain model, the financial data retrieval subsystem and the asynchronous processing subsystem. The interaction diagrams included clearly describe the interactions that occur within each of these subsystems. They elaborate upon the mechanics behind our use cases, but do not necessarily correspond to them one-to-one. This is because several of our use cases are completely facilitated through the browser and controller to generate views for the users, and as such it would not be interesting or worthwhile to explore the internal interactions. The following analysis clearly describes how market orders are placed and processes, how information is retrieved from Yahoo! Finance, and how we manage asynchronous processes (i.e. a queue) in order to process market orders and enact our mailer system.

1.2 Financial Data Retrieval Subsystem

Enter the Capital Games Financial Adaptor

For the querying and retrieval of real time and historical financial data and stock quotes in a form that is both familiar and friendly to players of Capital Games, we will utilize the ***Yahoo! Finance*** Application Programming Interface (API), which allows for easy access of ***Yahoo! Finance*** stock data via data served via URLs that our system can retrieve, parse, and then translate for the use of Capital Games fantasy leagues platform. Since we will be drawing data from ***Yahoo! Finance***, it will be represented as external to the system of Capital Games. Internal to our system, however, will be the financial adaptor module that will automatically handle data retrieval from ***Yahoo! Finance*** based on user queries.

We chose this route over either option of having financial data querying and retrieval built-in to our system or taken from any other API because attempting to construct a built-in, live stock-querying system within Capital Games itself would have been both expensive and impractical — much akin to reinventing the wheel — and because ***Yahoo! Finance*** has proven itself as stable and reliable versus other available APIs. Thus, this section will explain our intended financial adaptor module for seamlessly delivering ***Yahoo! Finance*** data for use within Capital Games.

Essentially, by us deploying the a financial adaptor module into Capital Games, users will be able to easily search for stock data within our website and have it near-instantly displayed on the web page they are viewing without the user even being cognizant of all the work being done in the background via our financial adaptor module existing in our server. The financial adaptor module will have all the functionality for making requests for data from ***Yahoo! Finance*** based on user

input and will actively draw and translate the raw data from ***Yahoo! Finance*** into a form that can be delivered within our own views ergo the data will be displayed on our web pages.

One consideration we need to take from our end for the building of our financial data is validating user queries for stock symbols. In other words, what would happen in the case that a user attempts to query a stock symbol, company name, industry, or sector that does not exist? To resolve such issues, our adaptor will also draw from our own database built into the website that keeps an updated list of valid stock symbols and names that is drawn from a source similar to ***Yahoo! Finance***, *EODData*. We are using *EODData* to supplement our use of the ***Yahoo! Finance*** API as *EODData* offers easy retrieval of all stock symbols and names in a method that is similar to ***Yahoo! Finance***. ***Yahoo! Finance*** unfortunately does not offer that particular feature, so we will be using *EODData* as a supplement to that, in that respect. We will essentially update our database via *EODData* and our financial adaptor module at each market opening and closure to account for any mergers, acquisitions, or any other major changes involving companies in the stock market.

Once user queries are validated by our financial adaptor module, our financial adaptor module will then parse the user query into a URL format that will allow for the retrieval of data via ***Yahoo! Finance***. Upon completing this, the URL will then be passed through our financial adaptor to ***Yahoo! Finance***, from which data will be returned to our financial adaptor module via a comma-separated values format (.csv, a container for easily passing volumes of data), which our financial adaptor will then translate into an arrangement that our views can utilize to deliver to the content to the webpage the user made the query from. From there, the user can then view the data and choose whether they would like to interact with the queried stock within Capital Games.

To elaborate on the technical specifications of our financial adaptor, the rest of this section will incorporate and explain interaction diagrams of methods used by our financial adaptor, illustrating the process I summarized regarding how our financial adaptor will go through interacting with ***Yahoo! Finance***, *EODData*, and the Capital Games platform.

All interaction diagrams will begin in the following page.

Financial Adaptor Interaction Diagrams

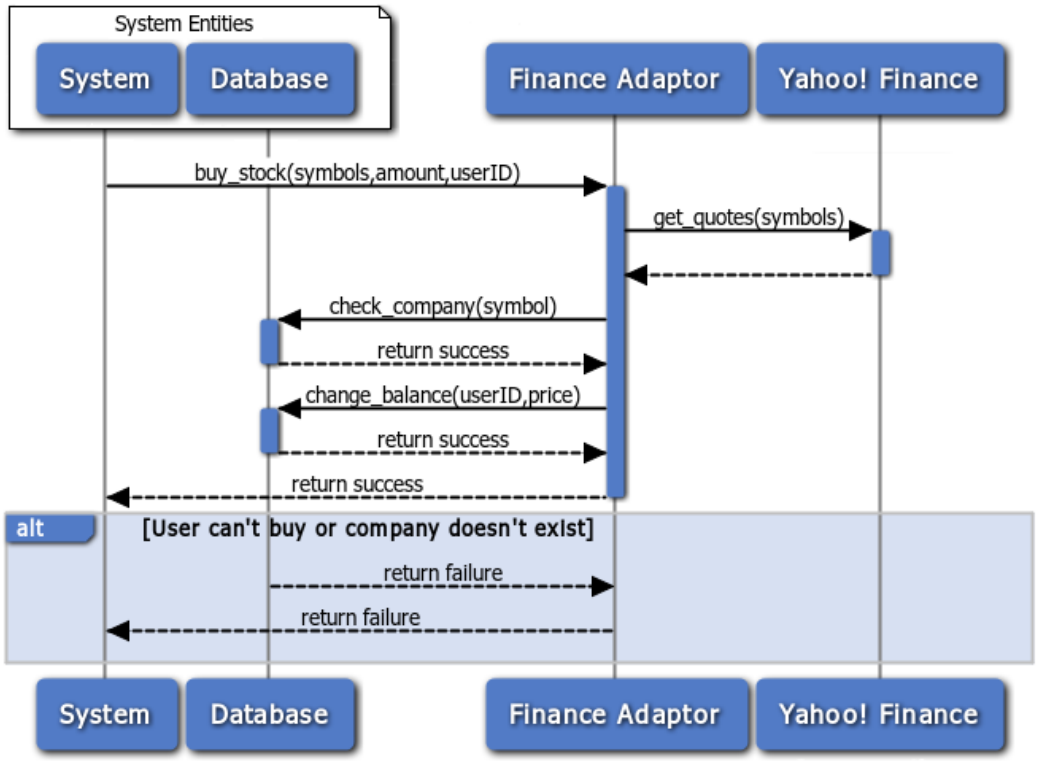

Figure 1.1: When a user buys a stock, the browser will inform the system of the transaction so that it can be approved. The system passes over the process to the finance adaptor who will check if the company is accepting trades, the current price from Yahoo! Finance, and if the user is able to afford the purchase from the database. If all goes well, the transaction will be recorded in the database and the balance will be changed. After all that is complete, the transaction will marked as a success and the system will be notified. (Related use case: UC-4)

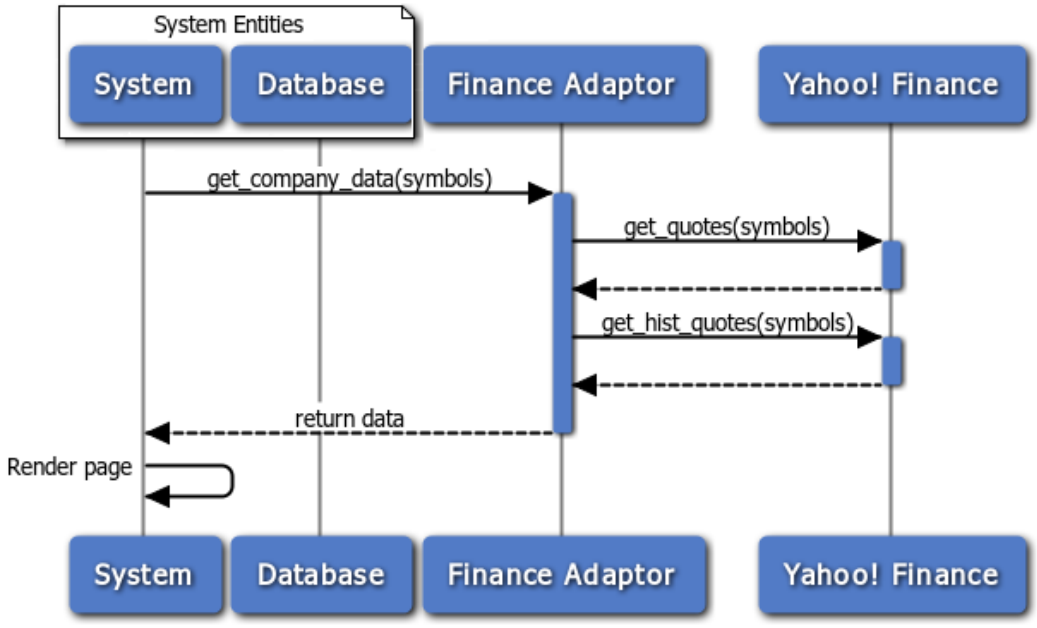

Figure 1.2: When a user wants to view a company page, the company data must be loaded from our finance API. Once the process is passed to the finance adaptor, the quotes and the historical quotes will be pulled from Yahoo! Finance and brought back to the system, who will prepare the page for the user. This is also the process by which user portfolios will be generated, via aggregating the value of all their stocks. (Related use cases: UC-3, UC-5

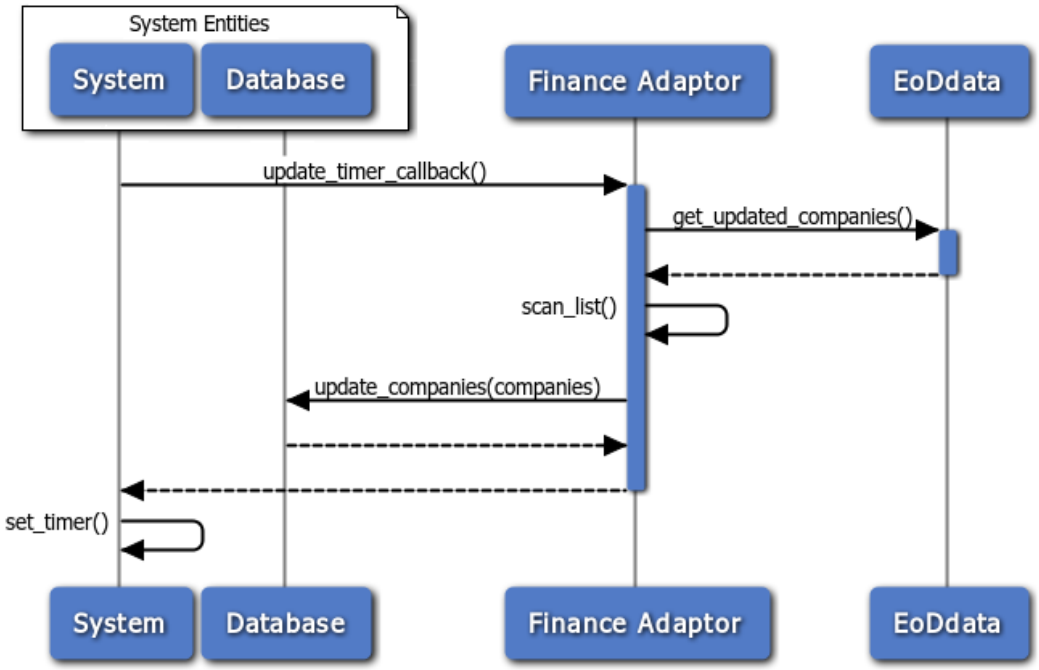

Figure 1.3: We need to keep a local copy of the current companies in our database so we can do rapid processes sing of all of the companies. In order to do this, there will be a timer that is set to update the database every once in a while. When the timer goes off, the system will pass the process onto the finance adaptor. The finance adaptor will then call data from EODData, who knows all of the current companies in the stock market. The finance adaptor will then scan the data for any new/deleted companies and change the database accordingly. After this is complete, the timer will start again so this process can loop.

1.3 Asynchronous Processing Subsystem

Introduction

One of Capital Games' primary requirements is to have an asynchronous processing subsystem. This requirement exists both due to the nature of our system, which involves events conditionally occurring at certain time intervals, and the pursuit to build a scalable product. In an attempt to build a system that most closely represents the real stock market, the decision was made to have a pending order queuing system which processes orders at 5 minute intervals. Many orders are processed directly, however some such as short sales and limit orders have conditions associated with them which determine when exactly they are processed. In addition, as the system involves sending summarized reports of player performance metrics at certain time intervals an asynchronous, non-event driven subsystem is highly necessary.

Nature of the Subsystem

The asynchronous processing subsystem features three primary components. First, the ability to spawn multiple, independent processes to handle the different kinds of asynchronous tasks. Second, the ability to handle arbitrary object types. And finally, the ability to queue tasks that are waiting to be processed. This is why the Resque Background Process Library built in Ruby was an ideal pick. It allows for the creation of customizable background processes known as "workers". Each worker processes a unique queue. Moreover, each queue can have objects of vastly different types, as long as they implement the function "perform". This is very intuitive as it allows each object to posses the code which acts on it. Lastly, it implements a very smart technique of only storing references to objects in the queue as opposed to the objects themselves so that outdated objects are never processed. This forces the worker to request the most recent version of the object from the DB when it starts being processed. Of course this comes at the slight expense of higher load on the DB when a worker is not sleeping. It is possible that this subsystem will be expanded to incorporate caching techniques. However, they are currently not a requirement. Finally, the queues are stored in RAM for the fastest possible performance. Nevertheless, queues are persisted in JSON encoded flat files to ensure redundancy.

Structural Model

Interaction Diagrams

There are two interaction diagrams displayed below, each associated with one worker. Due to the inherent background nature of this subsystem, there are relatively few actors involved in this subsystem.

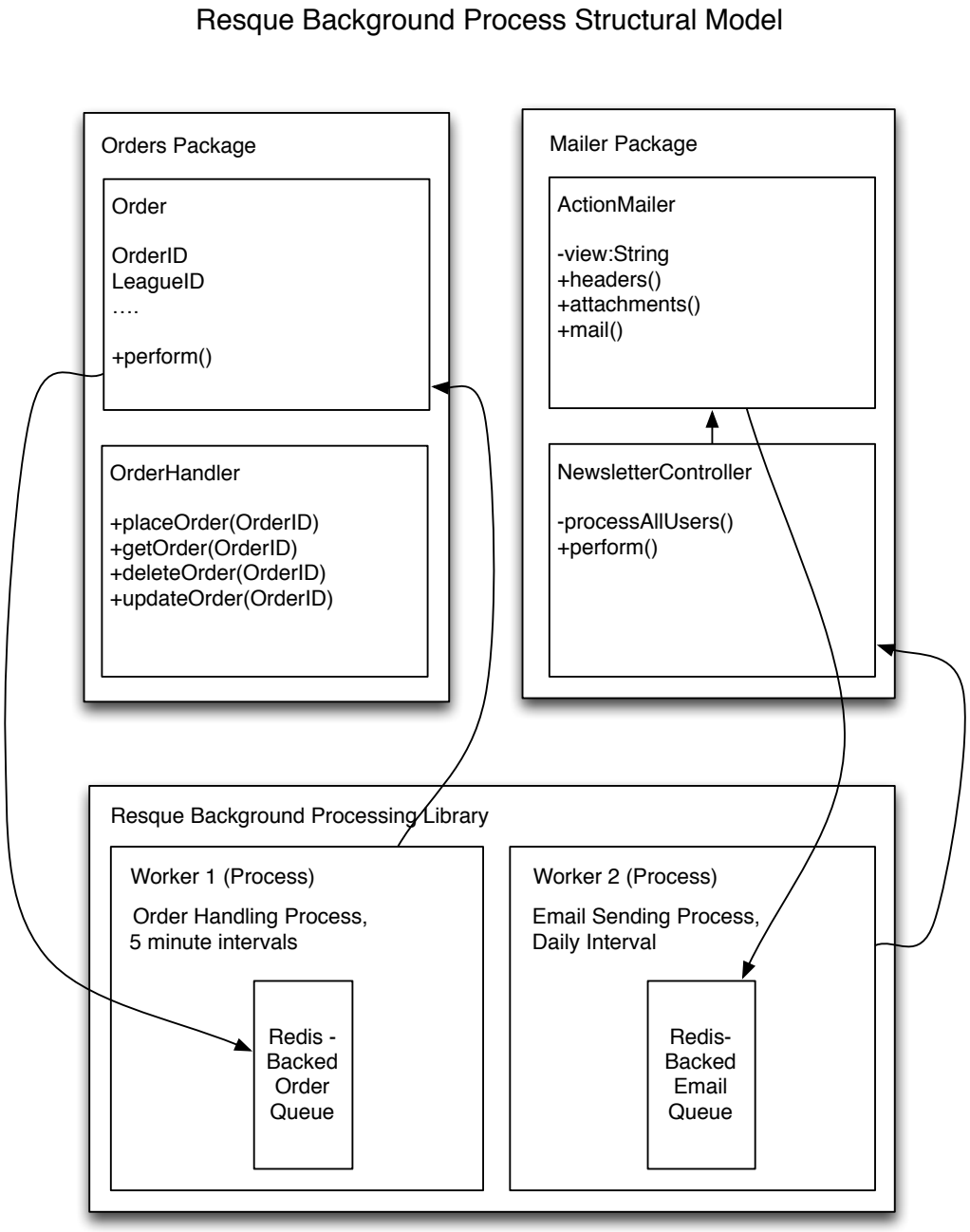

Figure 1.4: The structural model depicts the overall structure of this subsystem. Namely, the Resque Library and two packages or modules which each are responsible for one kind of task. On the left, the orders package displays a relevant subset of all classes that pertain to placing and processing orders. As previously mentioned, the Order object itself implements the perform method. Therefore, it knows how to process its data when it get gets placed in worker 1's queue. While the OrderHandler class isn't directly involved in the asynchronous processing of orders, it is still relevant in this scope and therefore included in the diagram. It is ultimately the class responsible for placing the order object on the queue when an order is placed. Similarly, the mailer package is depicted with a subset of classes which aggregate data about user performance and send out periodic summarizations of performance metrics to all users on the site. Worker 2 is dedicated to processing email related tasks daily. In this case, the architecture is slightly different as the worker doesn't directly call perform on each ActionMailer object, but instead on a NewsletterController which populates the worker's queue with customized ActionMailer Objects.

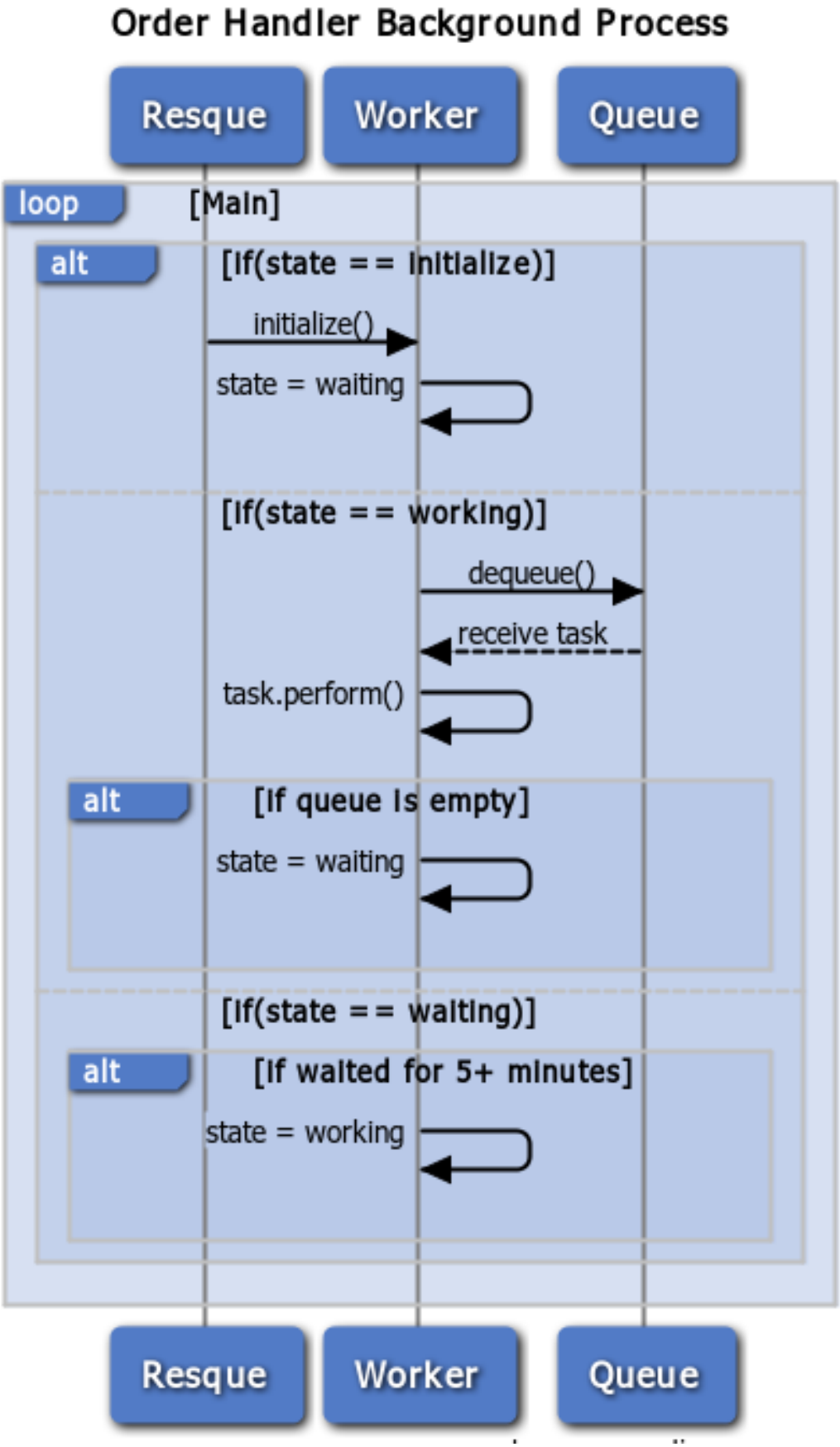

Figure 1.5: The interaction diagram above is roughly divided into two areas, when the process is working and when it is sleeping. This portrays the typical polling behavior of such a background running process. After initialization, when the worker wakes up it attempts to dequeue all objects

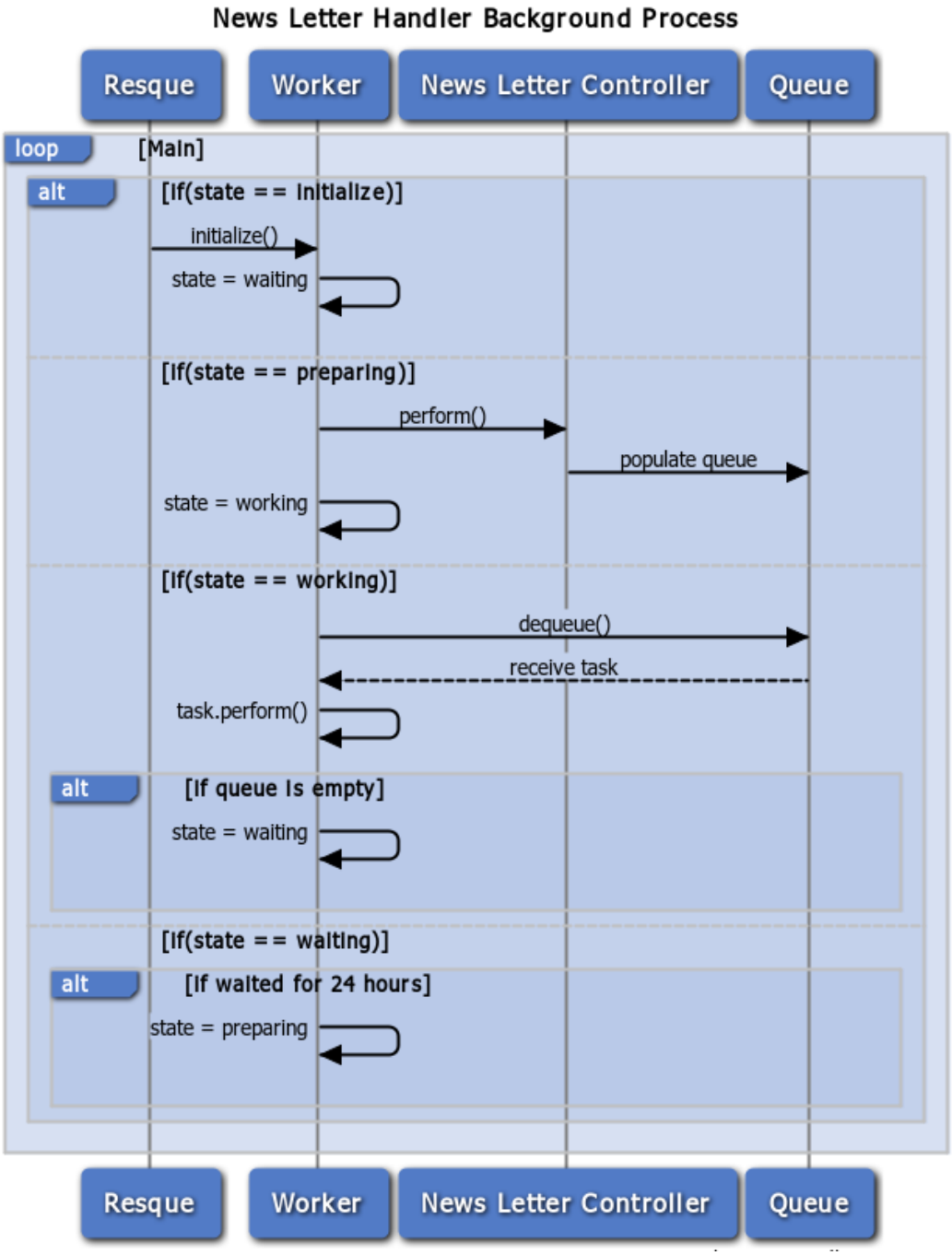

Figure 1.6: Worker 2 behaves a bit differently than worker 1 which results in having an additional state. This "prepare" state is when all the customizing of user-specific emails is done. Afterwards, the process enters the working state where it attempts to fire off all customized emails which were placed onto the queue during the "prepare" state. As in the previous diagram, the diagram incorporates the base case when the worker's queue has been emptied and when the process is sleeping.

2 Plan of Work

2.1 Development and Report Milestones

Illustrated on the next page is a gantt chart reflecting our goals relative to the project deadlines. It incorporates both core development and report items. For our initial stages we focus on environment and platform set-up (i.e. deploying a development webserver) and the initial, core code implementation. At the same time we will finalize the details of our final product via the report milestones.

Development milestones have been spread out following the completion of the first report on 22 February 2013, beginning with deploying our development environment and server through Heroku from which we continue to our next milestone of deploying Ruby on Rails as well as all the Gems and API packages we are incorporating into our project, most notably Yahoo! Finance.

Report milestones are also set concurrently. As we begin to initialize our development environment, we will also build on top of and expand on previous reports to expand upon and fully realize the details of *C*apital Games.

Core goals leading up to Demo 1 include establishing all core functionality for *C*apital Games. This includes the following:

- **Rails framework-deployed core functionality :** This includes a working system for navigating the website, registering a new user account, logging in, and creating as well as participating in leagues.
- **Setting a foundation for the database:** On top of having the aforementioned core functionalities, they also must be able to pass data through a routed database.
- **Implementing the Yahoo! Finance API:**
- **A functional user interface:** Our website should be usable, and having a functional user interface from the start will give us a lot of room to expand and optimize the UI.

2.2 Breakdown of Responsibilities Introduciton

Contributions leading up to the completion of this report are covered in the "Contributions" table on the page following the gantt chart. For the future division of labor, we all plan on subdividing aspects of both the next reports as well as the development of the *C*apital Games Alpha.

2.3 Breakdown of Responsibilities

Responsibilities for server/development environment deployment and set-up will be shared between Val and both Jeffs, as all three equally have great experience in the subject. Meanwhile Nick, and Eric will work of sequence diagrams.

While all other diagrams on the report will be covered by both Jeffs, the User Interface Design and Implementation will be worked on by Val, Dario, and Eric. Nick and Dario will work on both data types and operations while Val and Eric also work on the traceability matrix.

System architecture and design will be covered by Nick and Val. Jeff R. and Dario will begin on the database structure and site routing via the Rails framework. Nick and Jeff A. will work on the implementation of users in the meantime.

From there we will further subdivide work on the final aspects of the website, likely sticking with our initial idea of splitting predominant responsibilities following the model, view, and controller (MVC) design pattern. Jeff R. and Jeff A. will lead work relating to model design, Dario and Eric will lead work relating to views and interface, and Nick and Val will lead work involving controllers. That being said, while the MVC pattern will model sub-component ownership among the team. Individual implementation responsibilities will be distributed a bit more evenly based on the particular strengths of team members.

In summary, Project Ownership will be based on the MVC architecture. To reiterate, Jeff R. and Jeff A. will have ownership over the Model portion, Dario and Eric will have ownership over the Views (user interface, etc.) portion, and Nick and Val will have ownership over the controllers portion. Even beyond Project Ownership, however, responsibility for the whole project will be shared and the success of our MVC architecture requires close coordination between all aspects.

Overall project success will be decided with how well the MVC component teams communicate and work with each other, as *C*apital Games will rely on the interactivity between the Model, Views, and Controller portions of the architecture.

2.4 Gantt Chart of Projected Milestones

Best viewed at 100% or greater:

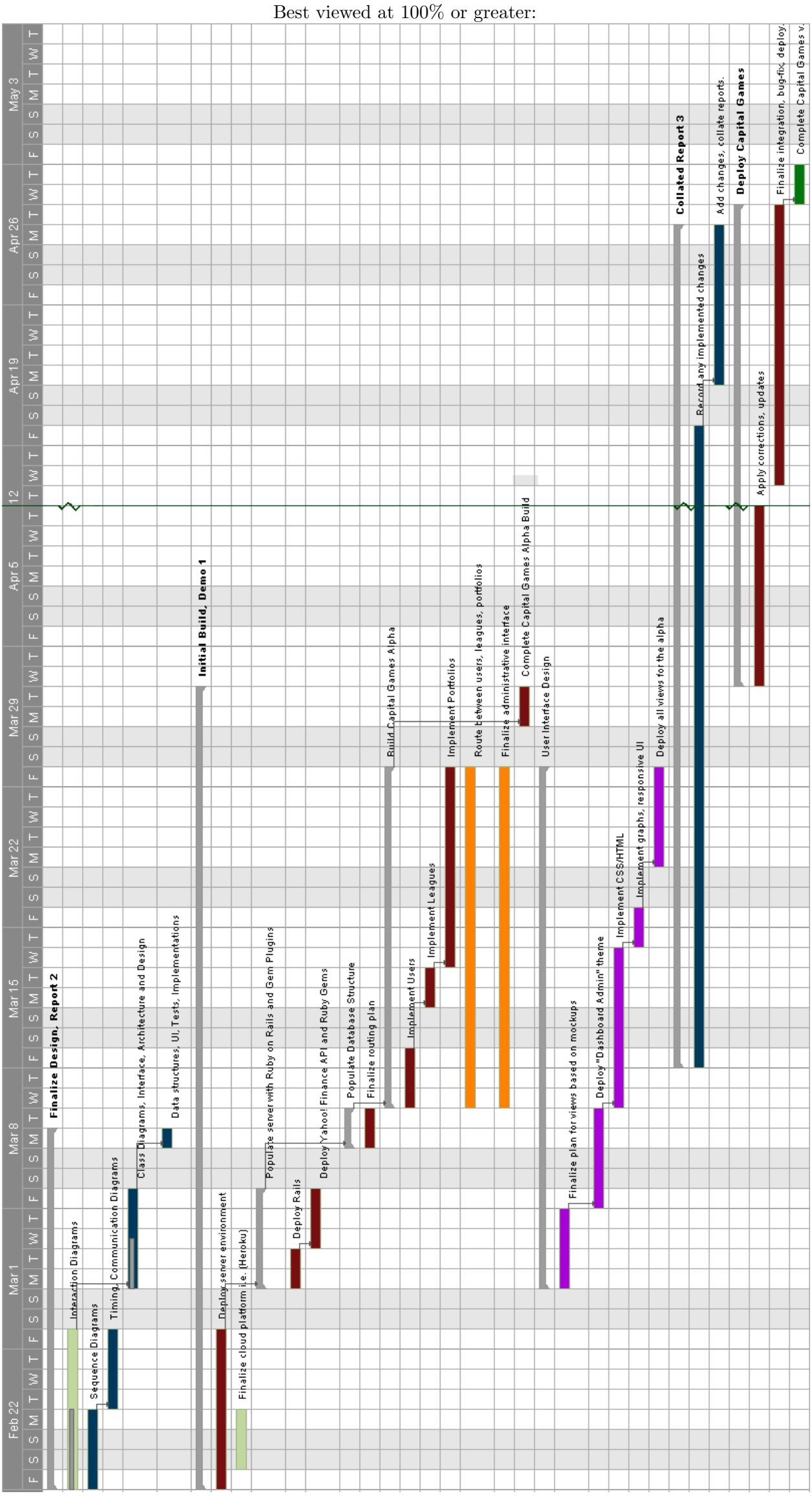

Figure 2.1: This gantt chart projects how we will concurrently work on the project. All blue items are report-related, red and orange relate to the core project development and purple illustrates UI milestones.