Simpply: Simple Supply

Software Design Specification

Version 1.5

3-4-2015

Contributors:

Michael Yeaple

Jeremy Shulman

Curtis Burtner

Mustafa Al-Salihi

**Document History**

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Revision # | Revision Date | Description of Change | Author |
| 1 | 11/20/2014 | First Draft | Jeremy |
| 1.1 | 12/16/2014 | Addition of iteration 2 designs | Jeremy |
| 1.2 | 02/14/2015 | Added Plan-o-gram Explorer | Michael |
| 1.3 | 02/16/2015 | Added Modify and Save Item Distribution | Michael |
| 1.4 | 02/16/2015 | Added JSON formatting and threading | Jeremy |
| 1.5 | 03/04/2015 | Added logging standards | Jeremy |
| 2.0 | 5/18/2015 | Revision of many of the designs | Michael |

[1. Introduction](#h.bppo2iroxyqw)

[1.1 Purpose](#h.kg6doymlylkh)

[1.2 Overview](#h.rpgzwzi36v8d)

[2. System Architecture](#h.kuuiowlm4dv)

[2.1 Implementation Design](#h.dz5hgbw17l82)

[2.2 Rationale and Breakdown](#h.31uq0c8nfob2)

[3. Engine](#h.fbkd6ohudru3)

[3.1 Implementation Design](#h.klyaw42tffwg)

[3.2 Rationale and Breakdown](#h.nap78e58q74)

[4. Algorithms](#h.heqfmplvd1c7)

[4.1 Sequence Diagram](#h.cab9t2byaedz)

[4.2 Rationale and Breakdown](#h.ayf3dvz2lxr0)

[5. Wegmans Import Job](#h.d07dtb3miuqv)

[5.1 Implementation Design](#h.5l11h9b1jqau)

[5.2 Rationale and Breakdown](#h.wjad2ttlh9c5)

[6. Plan-o-gram Explorer](#h.4alnp8r7awjt)

[6.1 Implementation Design](#h.ks99t3n53kq)

[6.2 Rationale and Breakdown](#h.h268qrxmit2l)

[7. Customize Explorer Views](#h.wf5wyhkggj6i)

[7.1 Implementation Design](#h.kb9t3wwovnge)

[7.2 Rationale and Breakdown](#h.iikkj1gy94ly)

[8. Modify Algorithm Parameters](#h.u5juuvxymdtb)

[8.1 Sequence Diagram](#h.z23mptbca6wj)

[8.2 Rationale and Breakdown](#h.ncrkppwvluwn)

[9. Modify and Save Item Distribution](#h.8rt1fgiaxrpj)

[9.1 Implementation Design](#h.5qdqtxyswts1)

[9.2 Rationale and Breakdown](#h.teikh9uhd9d6)

[10. Work from Item Explorer](#h.fsvddiijq9jh)

[10.1 Implementation Design](#h.68jw3136ysx3)

[10.2 Rationale and Breakdown](#h.l7vghmmnzmbp)

[10.2.a Selecting Rows](#h.nsn4ozque5uu)

[10.2.b Finalizing Items](#h.rey56l9aeobp)

[10.2.c Run Algorithms](#h.omgbqpazog1q)

[11. Database](#h.3vb65qhlde6i)

[11.1 Implementation Design](#h.ic8wymp0m8a0)

[11.2 Rationale and Breakdown](#h.dvizdoms8c4v)

[12. JSON Payload Structure](#h.2j8qaboe3578)

[12.1 JSON Request](#h.pkp3ytq6fx96)

[12.2 JSON Response](#h.pexcdkwwkld0)

[12.3 Rationale and Breakdown](#h.nqbp2mi31mo5)

[13. Threading](#h.osenzkd57i89)

[13.1 Sequence Diagram](#h.mupokk6xh3pw)

[13.2 Breakdown and Rationale](#h.6o7nr7m61jdy)

[14. Breadcrumbs](#h.ev2pcup6vyxj)

[14.1 Design:](#h.fipac85noajk)

[14.1.1 Breadcrumbs Controller](#h.g11npx80il9d)

[14.1.2 Partial View](#h.v3cn54conhgp)

[14.1.3 Layout Page](#h.vbrmml6gnhzg)

[15. Logging Standards](#h.hqyrdeat8kd7)

[15.1 Current Logging Status](#h.ujezz4ioy01u)

[15.2 Adding Logging to Different Projects in the Solution](#h.i1jicqt0jgtk)

[15.3 Standard Usage](#h.vcfigeqw7whz)

# 1. Introduction

## 1.1 Purpose

This document will have all of the implementation designs and rationales for why the system was designed in the manner it was designed. This document will help new developers spin up on the project and remind current developers why certain design decisions were made.

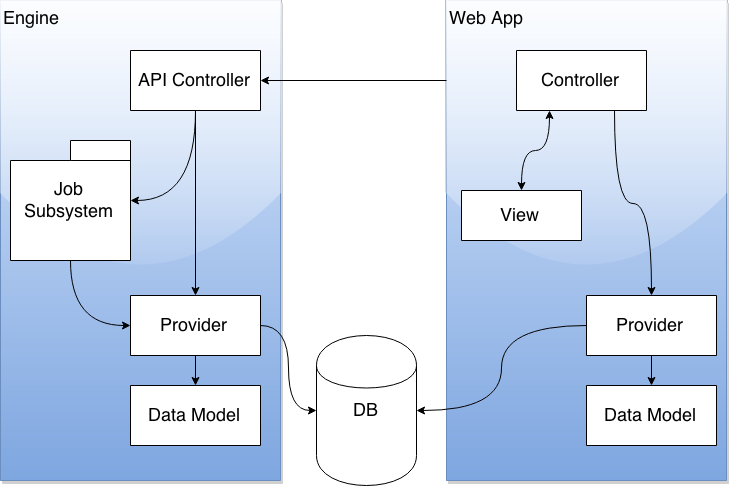
## 1.2 Overview

The system architecture section of this document will contain the bird’s eye view of the architecture for the project. This was a simple architecture that was done up front to guide the design for all features of the application. This follows our methodology’s core concepts for simplicity and to not have big design up front.

The other sections in this document will be different components or modules that are used in our project. This document will act as the one stop shop for design rationale and any potential future work for these components.

# 2. System Architecture

## 2.1 Implementation Design



## 2.2 Rationale and Breakdown

Above is a genericised diagram of the architecture. The three main pieces of our architecture are the web application, the engine, and our database. We split it up this way because the web application should be able to be used independently of the engine. This also allows for the application to be more performant and for it to scale easily.

The engine is responsible for running the computations required for determining each item’s store distribution via the job subsystem--which is described in further detail in the Engine section. We chose to design a job subsystem because we wanted to be able to kick off events with a “fire and forget” type pattern. Each job asynchronously completes one action which is the responsibility of the engine. These jobs are each treated as a transaction. This way we do not have to worry about the web application hanging while the engine calculates. It is not expected that anyone should be accessing the engine via the web browser; hence, the lack of other controllers and views. The API will provide an access point for the Wegmans scheduler and the web app to easily kick off jobs.

The web application is responsible for allowing users to view detailed data, distributions calculated by the engine--which have been saved in the database by the engine--and saving finalized distributions after they have been modified and approved by a merchant analyst. It follows a standard MVC model. MVC is a standard, tried and true approach to building stable web applications within a framework. One of the main reasons for this is that Wegmans commonly uses MVC for their applications so designing it in this way will allow them to more easily integrate our application into their code base as we’ll be following similar conventions. Additionally, ASP.NET comes pre-configured with the MVC framework which is very easy to get up and running with.

Providers are responsible for all interaction with the database. They query the database via stored procedures and use Dapper to map the results into C# objects (data models) for us to use within the application. Data models simply hold data and have very little to no functionality. Our decision to separate database operations into the providers allows us to keep the data conceptually separate from the population of data objects. It also isolates our database interactions to one area of the system, creating a better separation of concerns. It should be noted that providers and data models are a shared class library between the web app and the engine. The separation of the providers and data models into separate class libraries is done to adhere to the DRY principle, because the web application and engine use a lot of the same data and database interactions. That being said, at runtime, the web app and engine have separate instances of these class libraries, because the web app and the engine are deployed as completely separate web applications.

The decision to use an ORM was for ease of use and speed of development, without sacrificing much control, if any, over the database interactions since we chose to use stored procedures. Dapper was chosen as our ORM because it was very easy to get up and running and to work with. It was also approximately twice as fast as other comparable ORMs such as NHibernate--which was considered, but had too much setup overhead and would have added unnecessary complexity to the project. The only downside to using Dapper is that complexly nested data requires some manual effort to map into C# objects.

Stored procedures were chosen for all database interaction for a few reasons. The first is for security. They reduce the possibility of SQL injection. Additionally, they are faster, cacheable, reduce network traffic, and make it easier for the developers to work with data in the database. Finally, Wegmans recommended the use of stored procedures since they typically use them in their applications as well.

# 3. Engine

## 3.1 Implementation Design

## 3.2 Rationale and Breakdown

Within the engine, there is a web API--the JobController--so that both the web application and the Wegmans scheduler can kick off jobs. A command pattern used for the scheduling and execution of jobs. This was chosen to encapsulate the functionality of each job so the invoker could determine how to schedule the job for execution (this could be done with the use of a message queue, threaded requests, etc.). The data access providers are reused from the web application.

We wanted to decouple the computation from the presentation of the distribution information which is why the engine is separate from the web application, but we also wanted to have the web application be able to ensure it had up-to-date information. The solution we came up with was to allow the web application to start jobs. The jobs would log their progress in a job table in the database via a job provider, which would allow the web application to poll for the progress but still keep our separation of concerns. There is no communication from the engine to the web application other than to give a success response when a job starts, meaning that jobs can run asynchronously when kicked off from the web application and the end-user can continue to work while the job is running.

# 4. Algorithms

## 4.1 Sequence Diagram

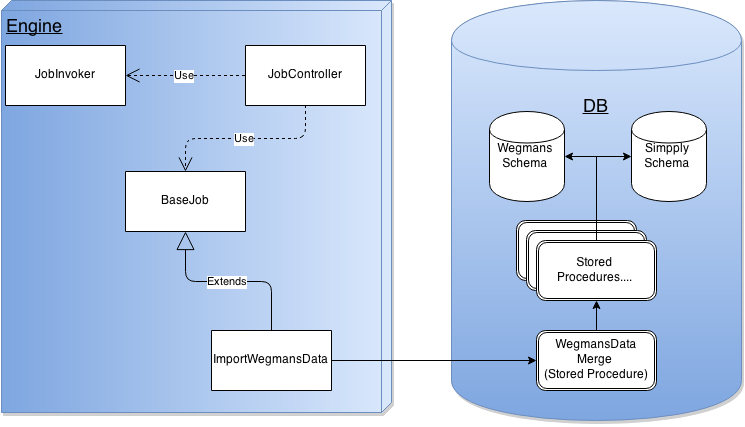
## 4.2 Rationale and Breakdown

We wanted an outline of exactly how a job would execute an algorithm and log its progress in the database specifically. The resulting sequence diagram follows the design of the engine as expected.

When the web application is polling for job status, we wanted the result to be as close to a concrete percent as possible. Brainstorming how to do this, we found that it would be difficult to estimate that. Instead we came up with a solution that provides useful status information even though it isn’t necessarily exact. A job that is computing an algorithm will update its percent completion every time it processes an item. It will begin by logging that the job has started through the use of the job table. After getting any required information from the providers it will begin to iterate over each item being processed. After a distribution has been created for an item, the job’s status will be updated. Finally, the job will log when it is done saving any distribution results. Note that saving will be completed in one batch job so progress will not be updated while saving is taking place. Rather, it will occur after the database transaction is complete.

# 5. Wegmans Import Job

## 5.1 Implementation Design



## 5.2 Rationale and Breakdown

In order to move Wegmans’ data to our database, Wegmans is responsible for inserting the data they want into their schema in our DB. Then the WegmansParseData job is kicked off on the Engine through the Engine’s API. This job will call a stored procedure which is responsible for wiping our databases of their data, and then replacing it with the new data. The data wipe clears everything Wegmans is responsible for providing, from item data, to sales data, to historical data, to store data. We preserve distribution information, so as not to erase any work done by buyers. After the Simpply tables have been wiped and the new Wegmans data has been imported, the job then kicks off the other algorithm jobs with default settings to create default distributions for all items across all stores.

The advantage of this system is that it provides Wegmans with an easy point of integration. If they want data to appear, they need only to insert it into our staging tables. Bad data does not have to be traced back to a bug in a complex parser in our system, which means Wegmans maintains code which they’ve written.

This system has the added benefit of being simple. The SQL scripts are easily understood and easy to modify. Any schema changes will most certainly require an adjustment in any of the import scripts, but this cost is mitigated by the fact that the required change will certainly be easy to find and easy to make.

# 6. Plan-o-gram Explorer

## 6.1 Implementation Design

## 

## 6.2 Rationale and Breakdown

The plan-o-gram explorer is a fairly straightforward component. The design of this component is extremely similar to that of the item explorer and the season explorer. When a request is made to the PlanogramController’s PlanogramExplorer method, the planogram data and aggregate data is requested from the PlanogramProvider’s GetPlanogram method. The data is then sent to the PlanogramExplorerModel, which compiles the data into an easily usable format for the view. The GridModel is used for creating the consistent grid seen across each of the explorer views (and the grids seen in the item details page). Column preferences are also taken into account, which are discussed in another section of this document.

# 7. Customize Explorer Views

## 7.1 Implementation Design

Note: The \_\_\_\_ prefix in front of Controller and Model is meant to show that multiple different controllers and view models use these classes.

## 7.2 Rationale and Breakdown

Customizing explorers within the application entails allowing a user to change the order and visibility of columns on each explorer view. We wanted a user to not only be able to do this for the current session, but we wanted the changes to persist over many sessions. This required a structure that allowed us to store this configuration in the database. We also wanted to make the configuration generic enough that it could apply to many of the views within the application and not be limited to one single explorer.

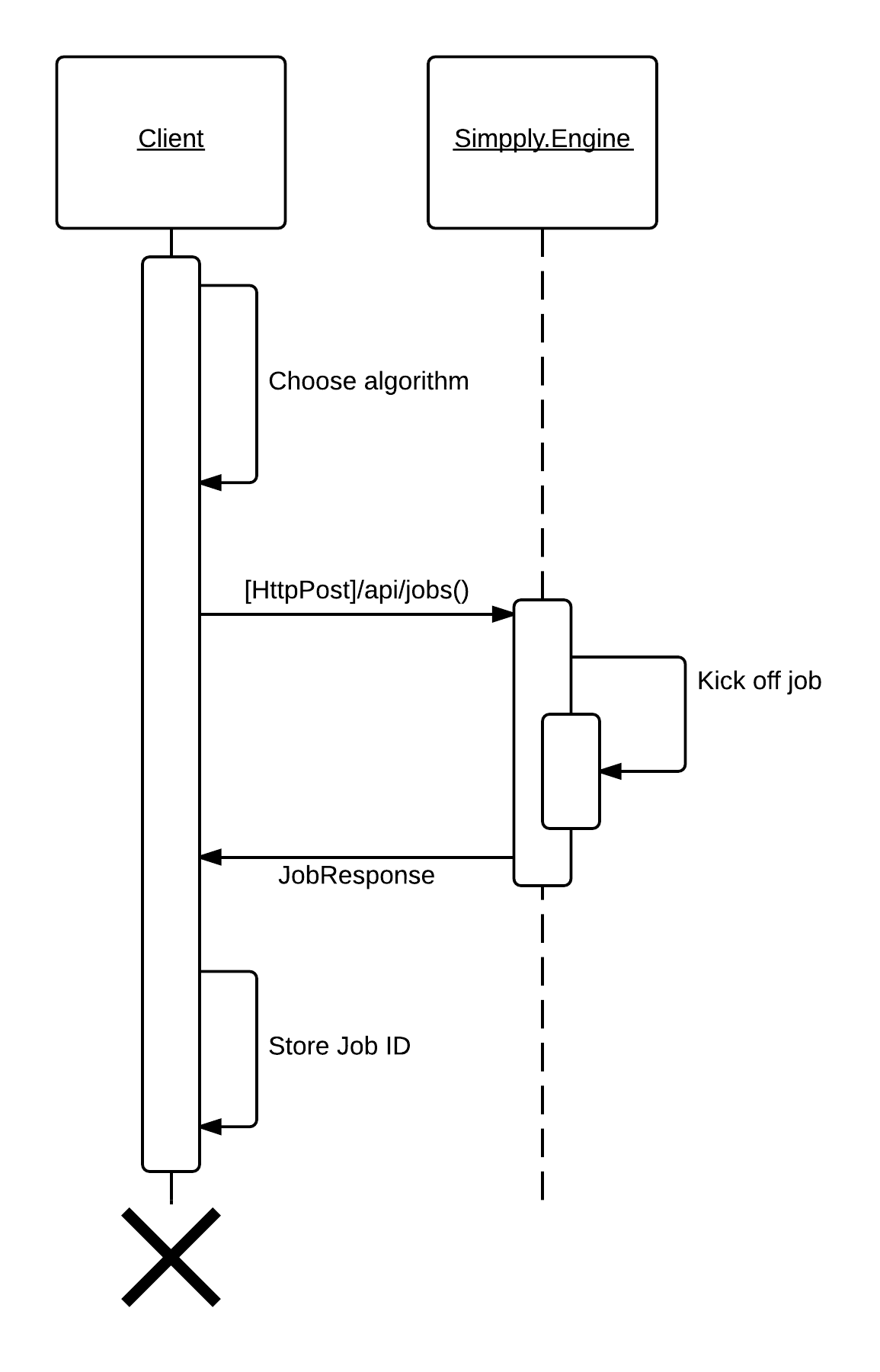
The general way in which the column preferences work is that a form is posted to the AccountController’s SaveColumnPreferences method, which then tasks the ColumnPreferenceProvider with saving these preferences into the database. Then, when a user makes a request the respective controller and action method associated with the column preferences, the ColumnPreferenceProvider’s GetColumnPreferences method is called to return a list of ColumnPreference objects determining how the view should be displayed. These are sorted into shown and hidden columns lists in the ColumnPreferenceModel within the view model associated with the controller and action method being called so that they can be used easily within the view for organizing the view and for populating the edit column preferences form.

It is important to note that user accounts are considered to be out of scope for this project. That being the case, the current implementation will result in global changes whenever someone changes the column preferences. In the future, ColumnPreference objects will have a reference to a specific user ID so that the changes are only visible to the user who made the changes.

The alternative to this was to use JavaScript to rearrange the columns on the fly within the page and store the settings for the duration of the user’s session; however, that did not offer the persistence that we wanted to offer to the users.

# 8. Modify Algorithm Parameters

## 8.1 Sequence Diagram



## 8.2 Rationale and Breakdown

The above sequence diagram depicts the process for modifying an algorithm’s calculation parameters. First, the user clicks a button on the page which calls a jQuery function to create a popup from a hidden form. On this form, the user chooses the algorithm he or she wishes to use, and then fills in the inputs for the calculation parameters he or she wishes to overwrite. The user then clicks the “Run Algorithm” button, which validates the form and then makes an AJAX POST request to the engine to kick off a job with the supplied calculation parameters. The engine kicks off a job and returns a JobResponse with the ID of the job which was just created. Finally, the Job ID is stored in JavaScript session storage for use elsewhere (e.g. getting the job status and creating a progress bar to display the job’s progress).

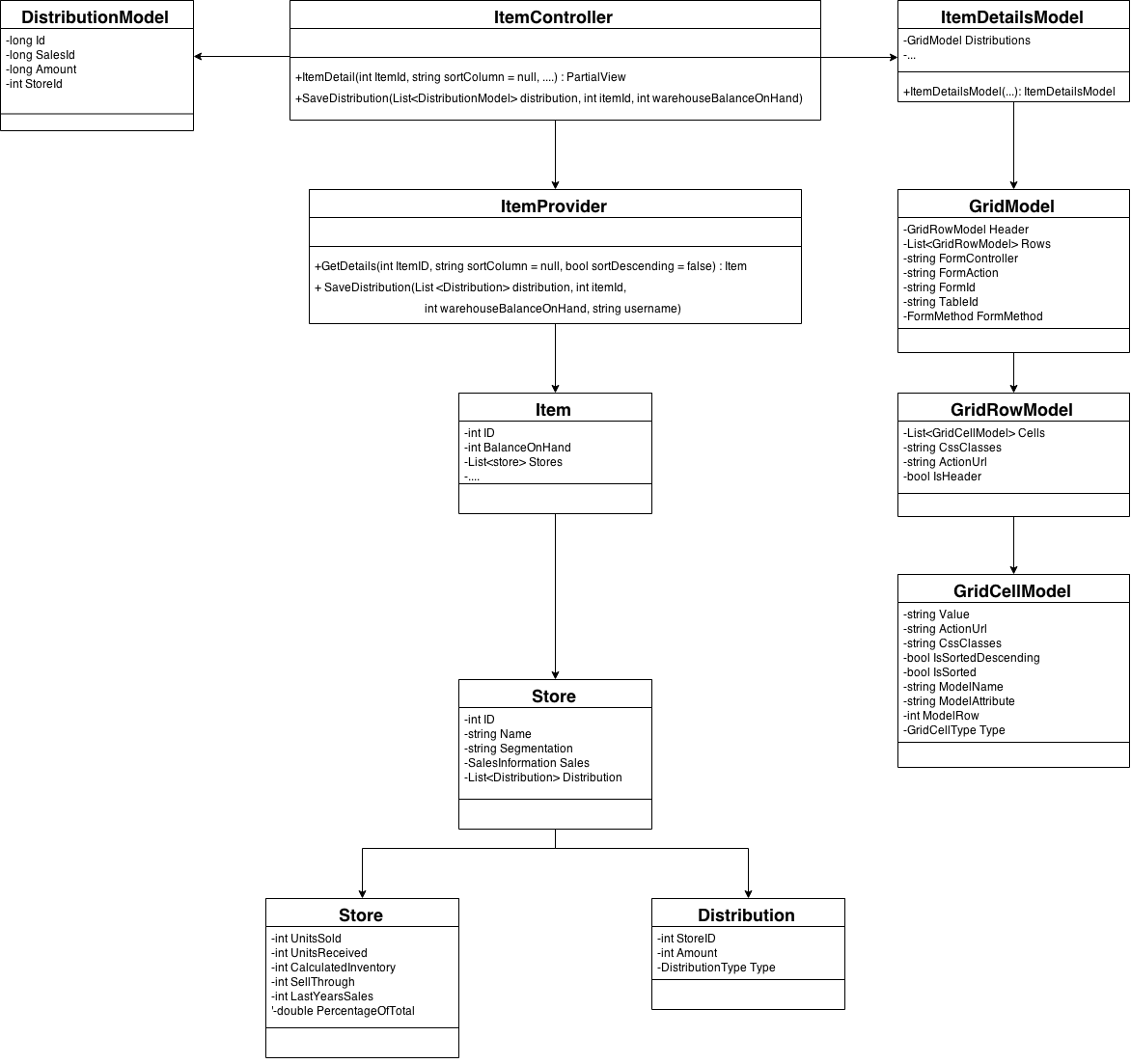
The code for this user story is all front-end (HTML, CSS, JavaScript) code. Reusable functions are all located in the app-function.js script file, whereas, one-off logic such as attaching these functions or managing form elements with JavaScript is done within the respective view.

It made a lot of sense for us to keep the logic for this all on the front-end in order to create a more responsive experience. We decouple the engine from the web application, and we also gain performance in the web application through this implementation. Jobs can run asynchronously while the user does other work in the system and then returns to the item(s) they ran an algorithm on.

The alternative to this design was to separate out the algorithm code into a reusable class library and run the algorithm through the web application upon a user request. That being said, there would be no performance gain or improvements from doing it this way. In fact, we could introduce concurrency issues that are not easily handled by doing it this way (e.g. a race condition resulting from a job running an algorithm an item and a user running the same algorithm on the same item with different parameters).

# 9. Modify and Save Item Distribution

## 9.1 Implementation Design



## 9.2 Rationale and Breakdown

Modifying and saving an item distribution is slightly tricky. To start with, we wanted to be able to modify a distribution within our GridModel. The reason for this is that each row in the GridModel is associated with a different store. This allowed us to keep our distribution amounts in the same rows as their store information, providing the user with a more cohesive and fluid experience. In order to accommodate this need, the GridModel had to be modified to be able to hold input fields and be placed within a form, if necessary. To make the GridModel into a form, three attributes were added to the GridModel class: FormController, FormAction, and FormMethod. If FormController and FormAction are both populated, the GridModel in the view will be wrapped in a form, using the FormController, FormAction, and FormMethod (GET, POST [This is a System.Web.Mvc enum]) attributes to populate the form HTML element appropriately. The other change that had to be made was to GridCellModel. ModelName, ModelRow, and GridCellType were all added to GridCellModel. ModelName, ModelAttribute, and ModelRow will be used to appropriately attach the correct name for the HTML input so that it posts to the server correctly as can be seen in the following article:

<http://www.binaryintellect.net/articles/b1e0b153-47f4-4b29-8583-958aa22d9284.aspx>

GridCellType is simply an enum which indicates whether a cell is a Text value, an Input, or a HiddenInput (the previous Pascal case words indicate the enum names). This allowed us to generate the correct HTML in the View when using the CellModel. Saving the item distribution is done by posting the GridModel form to the /Item/SaveDistribution method (POST only). This method takes a list of DistributionModel. The list of DistributionModel is then converted to a List of Distribution and saved to the database via the ItemProvider. The saved distribution is assigned a DistributionType of 4 (WorkInProgress) in the database, meaning that the distribution has not been finalized yet. Validation of the distribution in order to make sure that it is a valid distribution is done on the client side via jQuery and also on the server side.

# 10. Work from Item Explorer

## 10.1 Implementation Design

## 10.2 Rationale and Breakdown

The class diagram above depicts the back-end (C#) classes involved in working from an ItemExplorer level. There are several components to this user story. They have been broken down into the sections that follow.

### 10.2.a Selecting Rows

The ItemExplorerModel was the only class affected by this. Selecting rows was done by prepending checkbox inputs into the header and the item rows upon creating the GridModel for the Item Explorer. Hence, the reason there are no extra attributes or methods shown in the ItemExplorerModel. These checkboxes are disabled if the user is not filtered to a single plan-o-gram or the item has been finalized. A JavaScript function (Simpply\_GridModelSelectAll in app-function.js) is used to select all non-disabled checkboxes if the checkbox in the table header was clicked. Another JavaScript function (Simpply\_GridModelGetSelectedRows in app-function.js) is used to get all selected rows for use in the following sections.

### 10.2.b Finalizing Items

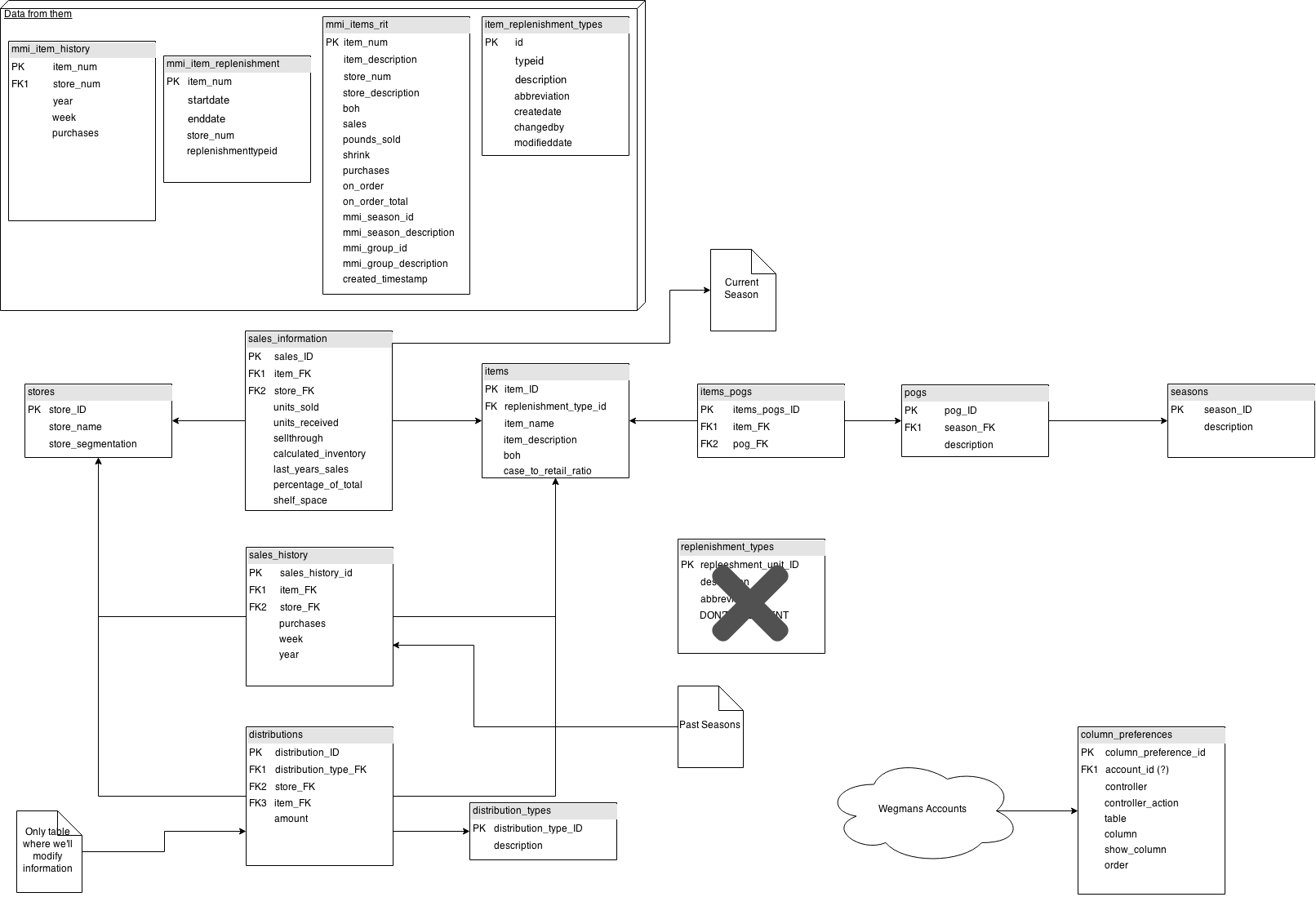
Finalizing items is fairly straightforward. A JavaScript function (Simpply\_FinalizeSelected in app-function.js) is used to first get the selected rows (using Simpply\_GridModelGetSelectedRows in app-function.js) and then to create a pop up prompting the user to choose a distribution algorithm to finalize for the selected items. When the user chooses an algorithm, an AJAX request is fired off (Simpply\_FinalizeItems in app-function.js) calling the ItemController’s FinalizeItems function. This, in turn, calls the ItemProvider’s FinalizeItems function, which calls the FinalizeItems stored procedure on the database.

### 10.2.c Run Algorithms

Running algorithms was developed from the ground-up with working from both the item explorer and the item detail page in mind. A JavaScript function (Simpply\_RunAlgorithmFromItemExplorer in app-function.js) gets the Item IDs from the selected rows and passes them to the run algorithm JavaScript function (Simpply\_RunAlgorithmPopup in app-function.js). This function renders a modal window from a hidden form for users to select an algorithm and change the inputs. When the submit button is clicked, an AJAX request is fired off to the engine’s JobController--which is an API controller--and then the JobController kicks off the respective job with the parameters passed to it via the job subsystem, described elsewhere in this document.

# 11. Database

## 11.1 Implementation Design



## 11.2 Rationale and Breakdown

We knew the format of information that we were getting from the Wegmans team but we wanted to normalize it and have our own representation to suit the needs of our application. We did this to make data retrieval easier and to logically assist anyone who needed to look at the data in our database.

# 12. JSON Payload Structure

## 12.1 JSON Request

When there is any interaction with the engine using JSON payloads, the format of said **request** payload will be as follows:

The JSON object will contain two elements

* The first element will be the name of the job to be executed (type string)
* The second element will be the necessary data for that job (type json string)

Setup distribution example =

{

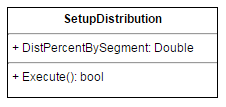
‘Job’ : ‘setup’,

‘Data’ : ‘{

‘DistPercentByStoreSegment’ : 25

}’

}



## 12.2 JSON Response

The **response** payload will be in the following format:

The JSON object will contain four elements

* The first element will be whether the request was successfully made (type bool)
* The second element will be an error code used for parsing failed requests. This value will be 0 in the event of a successful request.
* The third element will be a message with any additional information that the user needs to know
* The fourth element will be the job id if the job was successfully created. If it was not successfully created, this value will be 0 (type int)

Response example**s** =

{

‘Success’ : false,

‘ErrorCode’ : 2,

‘Message’ : ‘Incorrect parameters given to job’,

‘JobId’ : 0

}

{

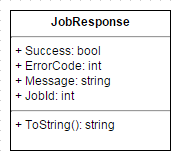
‘Success’ : true,

‘ErrorCode’ : 0,

‘Message’ : ‘Job successfully scheduled.’,

‘JobId’ : 1337

}



## 12.3 Rationale and Breakdown

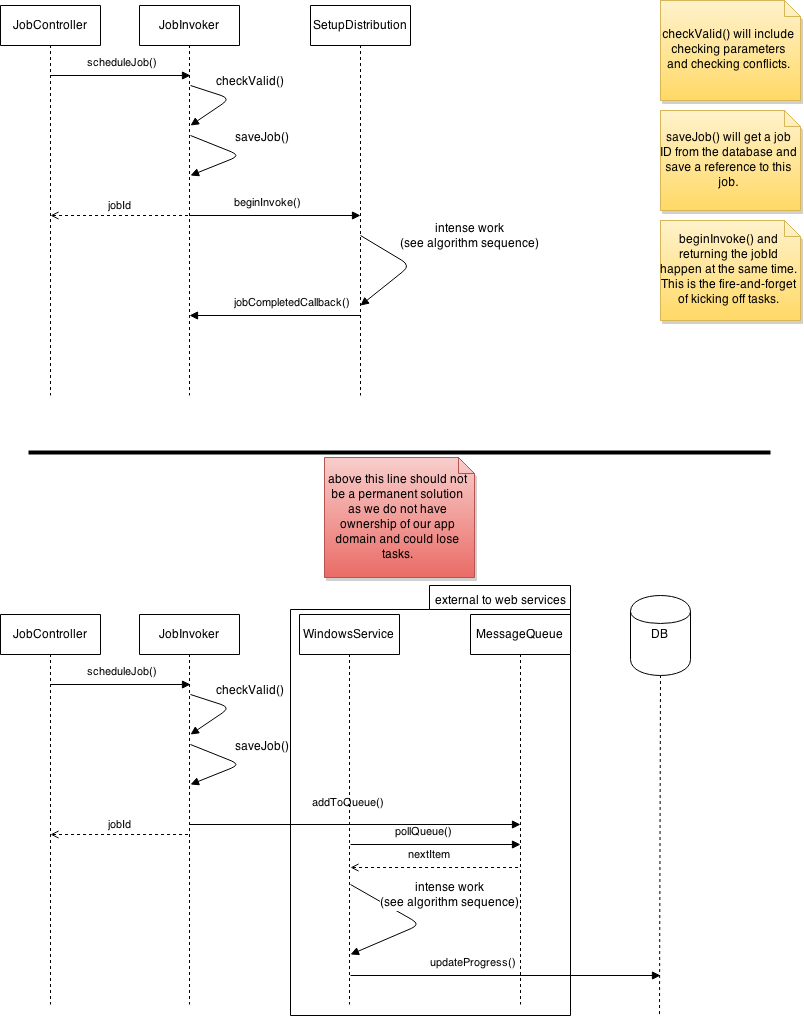
We wanted to normalize what was being sent back and forth to the engine. By formatting requests and responses, it is easy to parse on both sides of the transaction and it’s easy to understand for someone who is learning the system.

The request was formatted with the two fields because requests are only ever going to be for firing off jobs. However, each job has its own parameters. We’ll leave it to the jobs to determine if they have the correct parameters so that we can make sure we’re getting passed valid information formatted the way we want it before we actually create a job.

The response was formatted with the four fields for different amounts of information. You can easily check if the job was successfully scheduled or not. You can easily check what kind of failure was returned by the error code. A user-friendly response is contained in the message. Finally, the job id for future requests is returned.

# 13. Threading

## 13.1 Sequence Diagram



## 13.2 Breakdown and Rationale

Most of the jobs that the engine can execute have the potential to take a fair amount of time. For this reason we decided to set up a fire-and-forget model for queueing jobs. This way the client does not have to wait a long time to get a response from the engine while calculations are under way. Instead, the client will get a quick response that the job was queued.

This is not an optimal solution as outline below the horizontal rule in the diagram as we do not have ownership of our application domain. This can result in losing reference to jobs or even having jobs not finish at all. In the future we will have to set up some sort of messaging queue to handle these fire-and-forget tasks.

# 14. Breadcrumbs

## 14.1 Design:

Three designs were created for this feature. We selected the first design and have kept the two alternative designs listed, and described.

1. From the layout page, make an AJAX request to a BreadCrumbs **controller** that will return the appropriate HTML
2. From the layout page, render a \_BreadCrumbs **partial view**
3. Put all breadcrumbs logic on the **layout page.**

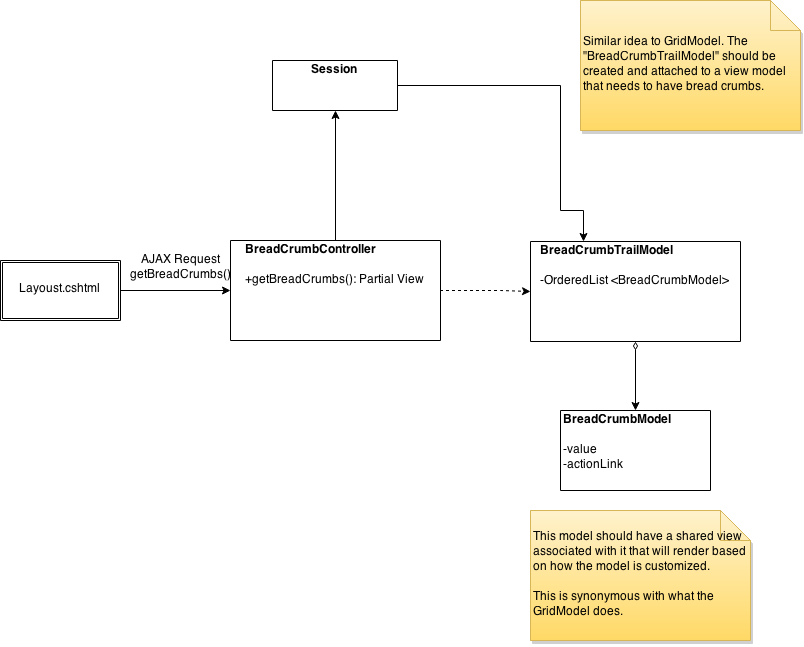
In all implementations, web-helper functions can be created to simplify complex code.

### 14.1.1 Breadcrumbs Controller

A breadcrumbs controller could be responsible for accessing session information as well as writing appropriate information to the session. This would definitely include writing the previous URL to the session, but it could also include saving a custom “Breadcrumbs” model object to the session (see <http://stackoverflow.com/questions/5644304/storing-custom-objects-in-sessions>) . Alternatively, it could simply save action links to “Season”, “PlanOGram” and “Item” values in the session.

This design would include the writing of a partial view, similar to what is done for the GridModel. The only disadvantage here is that it doesn’t follow **exactly** what our other controllers do in that it has no interaction with a “provider” class - which is something our other controllers tend to do. However, it is still responsible for interacting with a Breadcrumbs object. The advantage is that we get an increase in cohesion and separate our concerns (a .cshtml file is not responsible for handling breadcrumbs logic, only displaying a breadcrumbs model). Below is a rough diagram of how the feature would be implemented and interact with the system.

This is the design we chose to implement, as it was a good balance of advantages and disadvantages. The design sections below were included in this document for completeness.



### 14.1.2 Partial View

In a breadcrumbs partial view, we would incorporate the logic necessary to render a trail of breadcrumbs by utilizing CSharp code on the .cshtml page. This means the partial view will take on the responsibilities that the Breadcrumbs Controller would have had, this also includes potentially creating a Breadcrumbs model objects and saving those to the session.

The major disadvantages here would be maintenance and readability. CSharp code in the razor engine (.cshtml) can get quite finnicky. There will be a loss in cohesion, as the breadcrumbs partial view will be responsible for both displaying the breadcrumbs, and any logic that is unique to breadcrumbs

### 14.1.3 Layout Page

This is similar to the “Partial View” design, except the CSharp code now resides on the Layout Page. This *could* be appropriate, as we probably don’t want to reuse breadcrumbs anywhere else, but we would lose a lot of cohesion.

# 15. Logging Standards

## 15.1 Current Logging Status

Logging is currently configured for the following projects in the Simpply solution:

* Simpply
* Simpply.Engine

## 15.2 Adding Logging to Different Projects in the Solution

Log4net has been added to the package configuration for all of the projects in the Simpply solution. However, it must be configured if you want to use it. To do this, open the Web.config of the project you want to add logging for. You will need to add a section for log4net and then the specific log4net configurations. These can be found in the Web.config files in projects that already have logging set up. The configuration is based off of the examples on apache’s website(<http://logging.apache.org/log4net/release/config-examples.html>).

## 15.3 Standard Usage

In our solution there are two types of logging. There is logging that is part of the application which includes user actions including changes that were made, who made those changes, and when those changes took place. There is also debug logging which is used for development and debugging.

Apache has different log levels built in to log4net. The log levels from least important to most important are: DEBUG, INFO, WARN, ERROR, FATAL.

In order to use these log levels most effectively they will be used as follows:

DEBUG will be used for the finest-grained informational events. This level of logging will be disabled on production. This will be where most of the debug logging is done.

INFO will be used for the user actions in the system. In this way we do not need multiple database tables for logging. The different types of logging in the system will be separated by log level.

WARN, ERROR, and FATAL will be be up to the developer’s discretion but will not contain user action level logging.