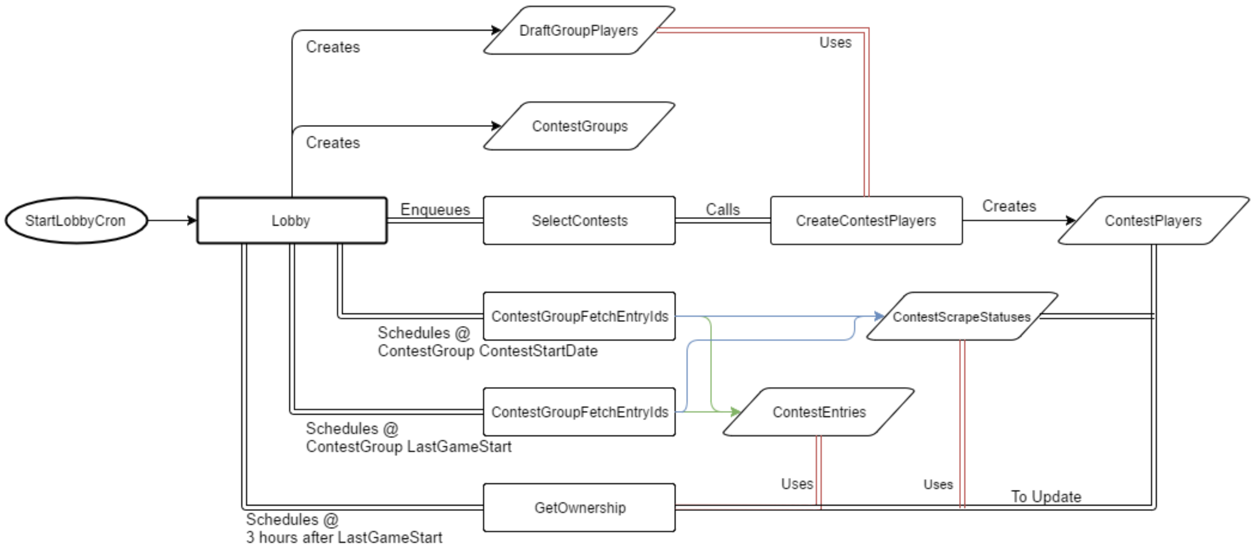

Starting the program:

Upon first start of the program, you will need to run **StartLobbyCron()** and **StartLoginCreate()**. **StartLobbyCron**() starts the recurring job to run the Lobby controller's **Index()** function. You only need to run this function once per machine. Even if the program stops, the job stays stored in the server. **StartLoginCreate**() starts the daily recurring job to run the **SendLoginCreate**(), which creates a DraftKings account. On first start, you should also trigger the **SendLoginCreate**() function (either manually via the URL or through Hangfire) once or twice to get a few accounts up and running.

The Lobby Controller:

Index(): This controller's Index function scrapes DraftKings lobby JSON for new *ContestGroups*, which DK calls "draft groups." If the function finds a ContestGroup that we have not stored yet, it does a few things:

1. Stores the player salary info for the ContestGroup as *DraftGroupPlayers* by accessing the JSON salary info on DK.
2. Stores the ContestGroup.
3. Schedules 4 different jobs:
    a. Immediately enqueues **SelectContests(int DraftGroupId, string SportAbbr, bool FirstRun)** with *FirstRun* as true.
    b. Schedules **ContestGroupFetchEntryIds(int ContestGroupId)** to run 3 minutes after the ContestGroup's games are schedule to start.
    c. Schedules **ContestGroupFetchEntryIds(int ContestGroupId)** to run when the ContestGroup's final games are scheduled to start.
    d. Schedules **GetOwnership(int ContestGroupId)** to run 3 hours after the ContestGroup's final games are scheduled to start.

If we find a ContestGroup that has already been stored, we enqueue **SelectContests(. . .)** with the *FirstRun* attribute as false.

SelectContests(int DraftGroupId, string SportAbbr, bool FirstRun): This function stores the available contests for the *ContestGroup* (marked by int *DraftGroupId*), and uses a simple algorithm to decide which 4 contests to scrape ownership from.
If *FirstRun* is true, it means we have yet to select these 4 contests. The algorithm selects the two largest contests that are either 50/50 or a Double-Up, and the two largest contests that are any other type of tournament (which will almost always be a GPP). For each selected contest, we store a *ContestScrapeStatus*. We also enqueue **CreateContestPlayers(int ContestId)**.

CreateContestPlayers(int ContestId): For the given ContestId, we find its ContestGroupId and use the *DraftGroupPlayers* stored in the **Index()** function to create *ContestPlayers*. These *ContestPlayers* will hold the ownership information.

CalculateFirstGameStart(SalaryRoot Salaries) & CalculateLastGameStart(SalaryRoot Salaries): Uses the given ContestGroup's salary info to calculate when the first and last game of the slate will start.

CheckAccess(HttpStatusCode StatusCode): Simple check to make sure we are not getting 403'ed. If we are, an email is sent an all tasks are paused.

StartLobbyCron(): Sends CRON job to scrape lobby for contests at 5am, 11am, 4pm, and 8pm.

DisposeHangfire(): Pauses all Hangfire tasks by setting a flag in the AppSettings to "true".

The Scrape Controller:

ContestGroupFetchEntryIds(int ContestGroupId): For the given ContestGroupId, we find the contests that we want to scrape (decided on earlier in **SelectContests()**) and send them out to collect their *ContestEntryIds* using **FetchContestEntryIds(. . .)**

FetchContestEntryIds(RestRequest LoginRequestTwo, RestClient DkClient, int ContestId, bool manual): Navigates to a contest's GameCenter and collects the IDs of all the shown entries. We scrape the page content and use regex to isolate the JSON containing all of the IDs. Then we send them over to **StoreEntryIds(. . .)**.

StoreEntryIds(List<ContestEntryJson> IdList, int ContestId): Stores each *ContestEntry* from the list of IDs for a given *ContestId.*

GetOwnership(int ContestGroupId): For the given ContestGroupId, we find the contests that we want to scrape (decided on earlier in **SelectContests()**) and send them out to collect ownership using **FetchOwnership(. . .)**.

FetchOwnership(List<double> IdList, int ContestId, int ContestGroupId, RestRequest LoginRequestTwo, RestClient DkClient): Given the list of IDs and the appropriate RestClient/RestRequest data, we send a post to DK to retrieve to player ownership for the entries. We send the response data to **StoreOwnership(. . .)** to be stored.

StoreOwnership(Dictionary<string, List<UserPlayerDataJson>> DkResponse, int ContestId): Stores player ownership, which is held in the *DkResponse* sent from **FetchOwnership(. . .)**, for the given *ContestId*.

FetchSingular(int ContestId): Used for manual Contest entry ID collection.
GetContestOwnership(int ContestId): Used for manual Contest ownership collection and for ownership collection retries in case of an error in one contest.

IRestResponse Login(RestClient DkClient): Called from any Fetch function in the scrape controller, this function returns a logged-in *IRestResponse*. The function starts by selecting a random login from our stored logins (created by the CRON job **SendLoginCreate()**). We attempt to login. If we get a 403, we pause everything and send out an email. Otherwise, we serialize the response (*LoginResponseObject*). If we get a statusId of -1, it means the account is restricted. We remove the login from the DB and recursively call the function. If we get a statusId of 0, then we've been captcha'ed. Tasks put on hold and an email is sent. Otherwise, the login is successful and we return the response.

SendMail(string Title, string Body): Sends an email to me from a Gmail account detailing the error. The mail recipient can be changed to one of you if you'd like.

The Create Controller:

StartLoginCreate() creates the CRON job to run **SendLoginCreate()** daily at midnight.
SendLoginCreate() creates a random string of numbers and letters to use as an account login.
CreateLogin(string id) sends the post to create the account.

CRON Jobs:

There is a recurring job to scrape DraftKing's lobby and its contests daily at 05:00 AM, 11:00 AM, 04:00 PM and 08:00 PM. This will cover all large contests and many of the small ones. We will most certainly miss a few small H2H and Multipliers that fill up within those timeslots. However, this will collect more than enough contests.
The other recurring job creates a DK login every day at midnight.

HangFire:

On startup, the program starts 3 HangFire servers that will process our tasks. While HangFire servers cannot be named, you can identify what each one is for by it's options. The first one

```
var options = new BackgroundJobServerOptions
            {
                Queues = new[] { "default", "contestload" },
                WorkerCount = 1
            };
```

Is for scraping the contests from their lobby JSON. To avoid IP/Account Banning errors, we leave the WorkerCount at 1, which means only 1 task will process at a time. While all tasks should be in the "contestload" queue, we leave a default queue, which has a higher priority, in case we want to run a manual task and not wait for other enqueued jobs.

```
var InternalOptions = new BackgroundJobServerOptions
            {
                WorkerCount = 25,
                Queues = new[] { "default", "playercreate" }
            };
```

This is used for internal operations and thus has a higher WorkerCount. As of now, it is only used for the **CreateContestPlayers(. . .)** function, which is in the playercreate queue.

```
var ExternalOptions = new BackgroundJobServerOptions
            {
                WorkerCount = 1,
                Queues = new[] { "default", “entryids”, “ownership” }
            };
```

The WorkerCount is low to avoid IP and account banning. The entryids queue holds the queue of functions that scrape the entry IDs, and the ownership queue holds ownership-scraping functions.

Navigating the Data:

From the homepage, select a date and click go to see the ContestGroups available on that date.

From the list of ContestGroups, click “View Contests” to see the Contests in that ContestGroup. For each contest, click on “Ownership” to see a list of players and their ownership. Click on “Send” to manually trigger ownership collection for that contest.

Or select “View Scraped Contests” to see the contests selected for scraping. From this page, you can also manually add contests to scrape.

FAQ:

1. Why are we not using the ownership .csv provided by DraftKings?
    a. By sending POSTs and using the DK response to collect ownership data, we receive each player’s actual player ID instead of a text name. With only a text name, we would have to match it using a dictionary of names and IDs. This has many problems (upkeeping the dictionary and players with the same name come to mind). We also avoid having to use RegEx to split up the player names in the .csv file

            i. "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems." - Jamie Zawinski

2. Okay, but why not get the entry IDs from the .csv instead of the GameCenter?
    a. DraftKings puts stricter location control on the .csv and I ran into numerous problems retrieving it via code. It's a possibility, but the GameCenter route was easier.
3. Why do we collect Entry IDs twice?
    a. DK's GameCenter only displays 500 IDs at a time. The concern is there may be a scenario in a large contest (think 20k+) where the entries we collect are all too similar. Imagine if we collect IDs after only one player has scored a point or only one game has started. Since DK only shows the top 500 scoring entry IDs, we would only get entries that rostered that 1 player or players from that 1 game.
    b. Additionally, there is a clause in the **FetchContestEntryIds(. . .)** function that makes sure to only scrape the entry IDs of contests under 500 people once.
4. How do I change this to use a MySQL server and not a LocalDb?
    a. Change the connection strings in Web.Config
5. Why all the **Thread.Sleep()** ?
    a. I've worked very hard to make sure the time in between requests is enough to keep our accounts from being restricted and our IP from getting banned.

Adding Old Data:

I have been collecting DK ownership on a LocalDb for a few months. Once we get this program up and running with MySql, I can create a .bak file from it and attach it to the MySQL database.