1.2.1- How would you gather, store, and update information from the EHP API?

Data Retrieval:

- The first step in dealing with any API on a programmatic basis is to handle the authentication. In the case of EHP, it appears to be a simple API Key, which I believe will be a constant value. For this, we can store this in a secure yet available location. I would typically store this in some sort of cloud-based secret management system (AWS Parameter Store for instance), that way the token is not stored in source control or tied directly to an instance or serverless function.

- Next, I would identify important endpoints for which we want to ingest data, and determine a cadence for how frequently we want to ingest it. For instance, the following datapoints might be retrieved at these timelines:
    - `/players`
        - Pulled daily
    - `/player-stats`
        - Pulled hourly
    - `/teams`
        - Pulled weekly
- Most of these endpoints appear to provide a filter on update datetime and use a limit and offset to manage throughput. In order to help manage our ingestion processes, I would leverage a database table (**EHPContext**) to track our access history on each endpoint, something like:

path	lastUpdatedUTC	offset
players	2021-11-03 04:00:00.000	6000
player-stats	2021-11-03 20:00:00.000	17000
teams	2021-11-03 00:00:00.000	400

- I would then write individual processes for pulling from each endpoint by supplying the **`lastUpdatedUTC`** for the respective **`path`**, getting us the most recent data for each path. After successfully handling the data (database write, manipulation, etc.) I'd update the **lastUpdatedUTC** to the current datetime and the **offset** to the latest offset/limit provided by the API response. This way if future calls fail for whatever reason, we'll be able to restart the pull where we left off.
- The processes themselves would be run on a cron job schedule. I would prefer to set these up as serverless cloud functions (AWS Lambda) but even an on-prem Windows scheduled job would suffice.

Data Storage:

- I'd initially store the results of each endpoint retrieval as raw (un-manipulated) and complete as possible. The API gives the ability to specify field arguments, for which I would typically be as greedy as possible and try to model our data models as close as possible to the API models. I would probably ignore some of the result fields that would otherwise be extracted from other endpoints (like **`nhl-rights`** or **`latestStats`** under `/player`) since they would be collected in another process. Most of the API models seem to have unique keys, which would also act as the primary keys on our

raw database tables. Following the endpoints listed above, this means a table for **`Players`**, **`PlayerStats`**, and **`Teams`** where columns are a subset of the result set available from the API. I'd prefer a relational DB for all of this like SQL Server.

- There may also be value in storing the raw JSON responses in a NoSQL data store like Mongo or SimpleDB, for which I'd write a separate process upstream from the relational solution described above. Storing raw JSON can be useful in testing, troubleshooting and future development.
- Downstream processes would be responsible for constructing and updating fact and dimension tables off of the raw data. This means constructing tables that are more useful and friendly to common database users, analysts, and web applications.

Data Updates:

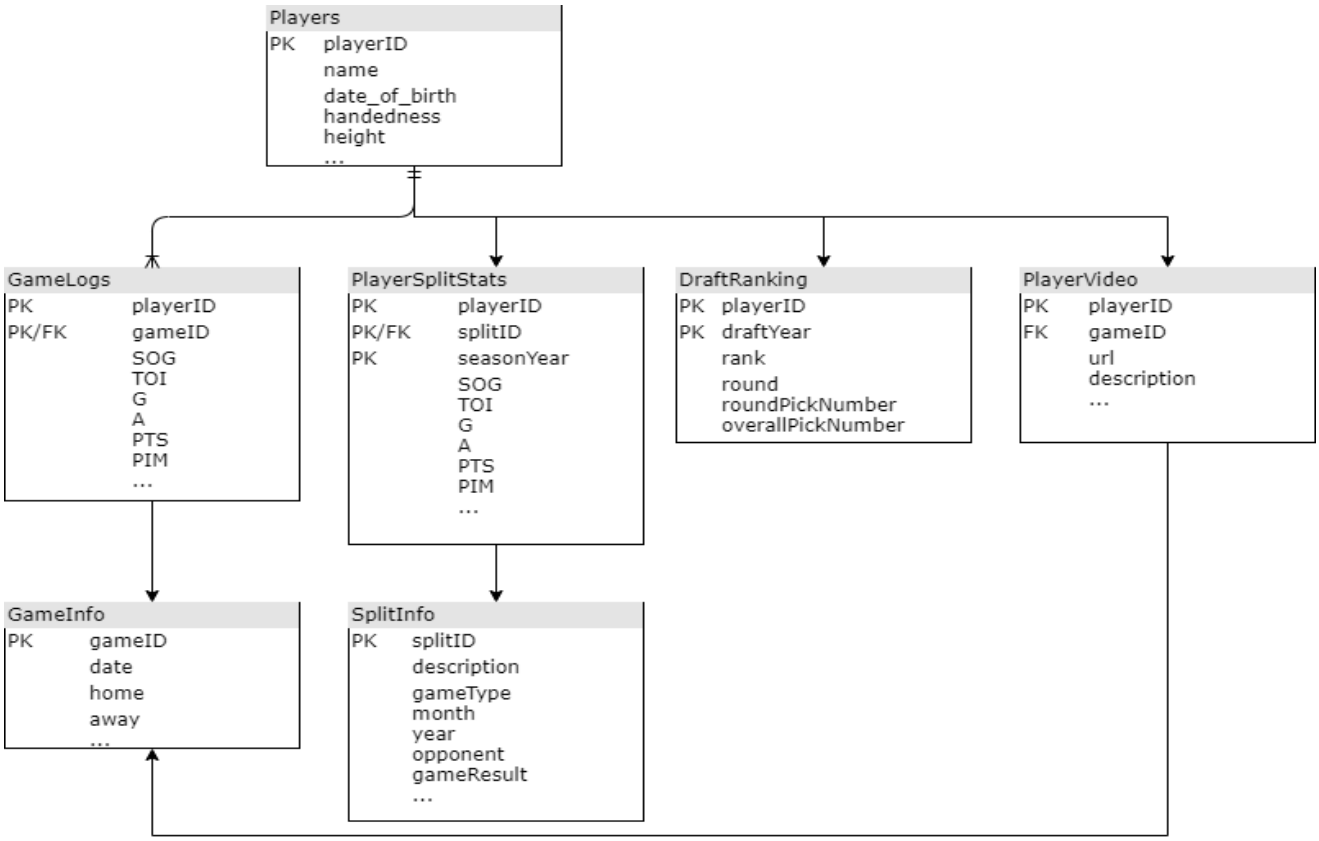
- The raw data from the API would be updated by means of the managed last-updated datetimes and primary keys in the processes described above.
- In order to manage updates to data that exists downstream of the raw data, I would employ either a Producer-Consumer Pattern using a message pipeline like rabbitmq, Apache Kafka, or AWS SQS to trigger those downstream ETL processes (more on this below in the **System Design** question), or use a Publish-Subscribe Pattern using AWS SNS.

1.2.2- What data tables do you think would be useful for hockey analytics purposes?...

- I think at least the following datasets from the Prospects API would be especially useful for analytics purposes:
    - **Players**: bio information such as name, position, age, nationality, handedness, cap hit, measurables at all useful for context. The player ID is necessary for putting a name to the other data sets.
    - **GameLogs:** Provides similar datapoints to the **PlayerStats** list, but on a game level for each player, so in theory we should be able to aggregate and build everything that's in the **PlayerStats** endpoint with this data. Also provides contextual information for the game: opponent, score, date, league, etc. Additional datapoints include:
        - Time on Ice
        - Shots on Goal
        - Shots Against
        - Shorthanded Goals
        - Saves
        - Goals Against
        - Games Played
        - Goals
        - Assists
        - Points
        - Penalty Mins
        - Power Play Goals
        - Save %
        - Goals Against Avg
    - **PlayerSplitStats (EHP PlayerStats but hopefully better):** This would act as a splits and aggregation table based off of **Gamelogs** metrics and the contextual information of the games. EHP has Regular/Postseason/Total stats. I think we could replicate that out of GameLogs, but

also include splits for home/away, month, win/loss, opponent, competition level and league splits.

- **DraftRanking:** I believe this is a prospect ranking list for a given year. This could be useful for internal draft ranking comparisons, building draft projections, and reflecting on past rankings vs. actual draft order.
- **PlayerVideo:** Storing a table full of in-game and workout video URLs could be a valuable resource for scouting analysts or for setting up a more robust in-house video-tagging solution



1.2.3- How would you deal with the name-standardization?

Solving this type of problem will almost always require some sort of user intervention to handle ID and name conflicts.

- I think the first step in handling this is to build a table that manages all of a player's various IDs from different sources in one location, and assign that player a universal ID:

```
SELECT [sabres_id] -- PK
      ,[EHP_id] -- rest of the ID cols are nullable
      ,[NHL_id]
      ,[Catapult_id]
      ,[TrackingSensorSystem_id]
      ,[KHL_id]
      ,[CHL_id]
      ,[first_name] -- not nullable
      ,[last_name] -- not nullable
      ,[date_of_birth]
FROM [PitchTracker].[dbo].[PlayerIDMaster]
```

esults | Messages

sabres_id	EHP_id	NHL_id	Catapult_id	TrackingSensorSystem_id	KHL_id	CHL_id	first_name	last_name	date_of_birth
1	A00001	109283	NULL	NULL	NULL	NULL	Pete	Ash	1986-04-03 0
2	A44444	101234	NULL	3223	13121	NULL	Maxim	Afinogenov	1979-09-04 0

- The sabres_id is the primary key and auto-incremented with each new record in the table. The other **_id* columns are from various data sources (EHP, NHL, KHL, wearables, tracking systems, etc.). The name and DOB columns are both contextual and could be helpful in mapping incoming IDs together.
- For every new data source that provides a different player ID, a new column will be needed in this table.
- This proposed table would allow queries that join EHP data with KHL data if available. Making a query like this possible:

```
SELECT player_master.EHP_id, player_master.KHL_id, ehp.*, khl.*
FROM raw.PlayerStats ehp
INNER JOIN dbo.PlayerIDMaster player_master ON ehp.prospect_id  = player_master.EHP_id
LEFT JOIN raw.KHLPlayerStats khl ON khl.player_id = player_master.KHL_id
```

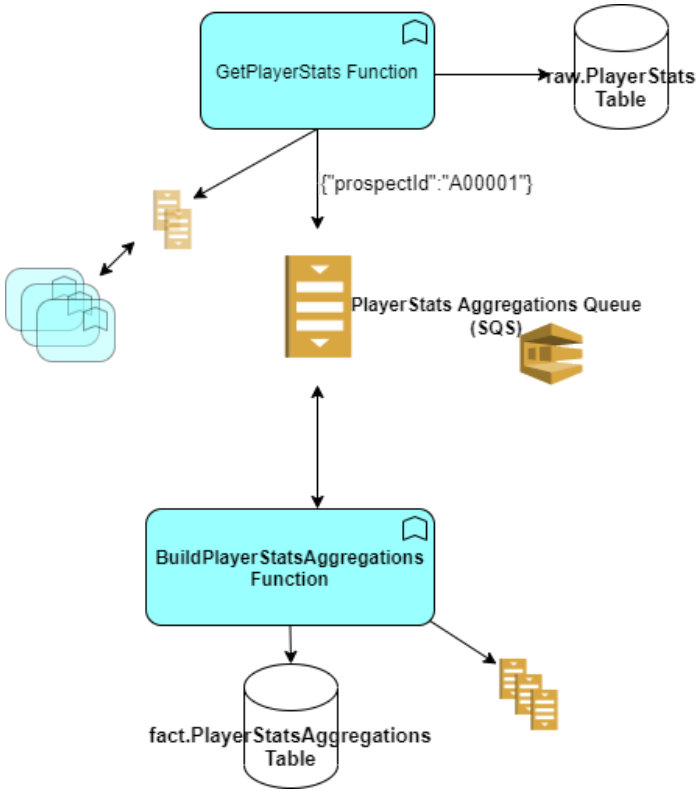
- Basic logic for updating the table:
  - When a new player and ID comes in from a source: try to use some fuzzy string matching on the names + the date of birth or other common bio metadata to get a confidence score on whether a current **PlayerIDMaster** record exists. Python has a library called fuzzywuzzy that is pretty good at generating a string matching score.
  - If a **PlayerIDMaster** record exists and the new ID column *does not* have a value, then update the master record. If a master record does not exist, create one with a new **sabres_id** (auto increment) and appropriate source ID. Again: whether or not a master record already exists is based on thresholds related to the string matching score.
  - If a master record exists and the source ID column *does* have a value, send an alert to the department (via slack message, email, web app, etc.) to review the incoming ID.
  - It may be useful to send similar alerts for fuzzy name/DOB matching scores that fall within a certain threshold range. “ Vyachslav vs. Slava” might generate a less-than-ideal confidence score for which would require a user to review. Eventually it may be useful to build out a dictionary of common outliers that can be used in the matching process.Maybe send alerts for fuzzy matching score that are in a certain range “just in case”
- I think it’s also important to keep a logging table for any updates that occur to the **PlayerIDMaster** table so that changes can be reviewed retroactively. Knowing about confidence scores used in creating new records or updating source ID columns would be important for troubleshooting and improving the process over time.

- Lastly: a web app should be built to help cleanly manage conflicts that occur in the master table. This is where users will be directed to on alerts. Users will need the ability to inspect the master record itself, as well as the metadata behind the source IDs to make the best decision on determining “who is who?”.

1.2.4- How would you design this system to handle the streaming nature of the data?

I touched on this a bit in the **data updates** portion of question 1, which describes a Producer/Consumer pattern.

- In this pattern our Raw processes would produce a Player ID, Team ID, or Player Stats Context to a dedicated message queue. Then, we would have a consumer process poll the queue for new messages and when one arrives, process it accordingly:



- In the above example, GetPlayerStats is our API-accessing function that is on a cron schedule, running once every *X* minutes or so to get the latest data from the `/player-stats` endpoint. It writes updated data to the database, and then sends a message containing the prospectId to at least one queue. Meanwhile the queue (PlayerStats Aggregations) is being consumed by at least one function that is responsible for aggregating data, saving to the database, and potentially producing to yet another queue.
- The **BuildPlayerStatsAggregations** function will follow these steps:
    - Read the message from **PlayerStats Aggregations Queue**
    - Try to parse the “prospectId” value from the message (*“A00001”*)

    - Process the value: Collect the metric values upon which to aggregate (for simplicity let's assume it's just current year metrics), build aggregation record for player's current year, and atomically upsert record to **fact.PlayerStatsAggregations** table.
    - Delete message from **PlayerStats Aggregations Queue.**
    - Produce message(s) to any downstream queues that might be dependent on the completion of this process (for instance: percentiles)
- With **AWS Simple Queuing Service (SQS)**, the queues can act as a FIFO (first-in, first-out) queue, where the data is received from **PlayerStats Aggregations Queue** in the same order in which it was sent, if that's important. Increased performance can be had by using a standard queue, which does not guarantee ordering.
    - Not pictured: each queue can have an attached "Dead Letter Queue", which can store messages that result in an error or those that time out as a result of processing out of the queue. This is necessary for error handling and troubleshooting.

1.2.5- What other issues do you expect to encounter along the way…?

- EHP changes the format of their API results or sends unexpected values. Expected integers being sent as decimals, expected strings sent as numerics, enumerations with new or different values can all cause problems in the ETL process. We would need to know when these errors occur immediately and do anything we can to make sure they don't "fail silently". With the SQS design we can rely on "dead letter queues" to handle message inputs that result in errors so that we can retroactively adjust our code. We can set AWS Cloudwatch alarms and alerts to send us notifications when a dead letter queue has messages that need review.
- EHP API is unavailable when we try to retrieve new data. This should be handled thanks to the **EHPContext** table discussed in question 1 with regards to data retrieval. It contains a **path and lastUpdatedUTC** to help manage when we last accessed the API, so that when the API becomes available again we won't lose track of where we left off. We'll also want to set up alerts for when the API becomes unavailable for an extended period of time so that we can determine the root cause (perhaps we have a bad token, credentials have been changed, we didn't pay the subscription bill, etc.)

---

2 - Would you recommend housing our internal databases... in the cloud or on premise?

I would recommend housing the internal systems in a cloud environment for a few reasons:

- **Reduced IT staff and infrastructure support:** Server maintenance and physical storage is delegated to the cloud service company reducing the dependency and costs involved in hardware upkeep and server-room expenses.
- **Scalability:** upgrading server performance and storage can be performed with the reconfiguration of a few settings or provisioning a new cloud instance/function.
- **Serverless functions:** The ability to develop data processing systems without the need for servers at all can't be matched with a strictly on-prem solution. AWS Lambda Functions or Google Cloud functions

can be scaled up instantly to handle increasing throughput as needed and quite possibly for a fraction of the cost of running dedicated servers.
- **Reduced cost:** The combination of spot instance servers, serverless functions, and cloud-based cron jobs means you can run processes without having a dedicated server consuming resources 24/7.
- **Availability of managed services:** Services to support our applications and database already exist in the cloud and can be set up relatively quickly. For instance: **AWS SQS,** a message queuing system, is already configured to the extent that we would just need to configure our queues. If we wanted to create a similar system on-prem, we'd likely use something like **RabbitMQ,** which would require us to dedicate at least a partition of a server to manage just the queue management system alone...with a cloud solution the queue management is handled entirely by the cloud service provider.

One of the pros involved with an on-premises server would be the reduced reliability on internet connection and a third party to manage our database, data and services. Also, if a capable, resourceful and experienced IT department is readily available to support on-premises systems, then it reduces the benefit of some aspects of cloud services.

3.1- Difference between "past performance and predictive future performance"

Past performance looks at metrics that have already occurred on an individual level. Predictive performance takes into account a population of similar individuals and/or situations and applies their average past performance to try and deduce likely future results.

Answering the question "who performed best in the past" first requires you to come up with a concrete definition of what "best performance" means by establishing some concrete metrics to measure. It also requires you to select a fair sample population to include in the "best performance" ranking. These two steps are important in reducing the amount of subjective noise that can come along with such a ranking.

The question "who will perform the best" is best answered after having come up with a conclusion to the former question and then determining what variables tightly correlate to "best performance". When these variables are known, then we can perform analysis to predict who will likely trend in a positive direction of high performance.

3.2 - Does a goalie's ability change over time? (20 yrs of stats)

Using the past 20 years of goaltender data for a given league, assuming it includes save percentage I think I would approach the problem with a regression analysis.
- First, for each available season, I would get the delta between each player's Year X save percentage and their Year X-1 save percentage. By getting the change in player's save percentage from year to year, we help put all-star goaltenders in the same realm as the replacement level and below-average goaltenders. We're just comparing how much better or worse each goalie gets from one year to the next
- Next I would probably remove goalie seasons that don't meet a minimum shots against criteria, just to throw out some outliers of goalies that make brief and limited appearances for whatever reason.
- At this point I should have a dataset that consists of X players over the past 20 years identified by their age at the start of the season and their change in save percentage from year to year. The assumption

I'm making is that variable X (player age) has an influence on variable Y (annual save percentage change)
- I would then plot the results and fit a line through the scatter plot. With a line available we can get a slope, which can tell us the amount, on average, a player's save percentage can be expected to change from one age to the next.

3.3 - Why are some of the league's most talented players among the worst in the league when it comes to giving the puck away? What are the issues with a metric like "giveaways per 60"? How would you design a better metric to capture how responsible a player is with the puck?

I think this can be explained by the fact that the players on this list, all centers and wingers, probably handle the puck more frequently than other positions on the ice on the same team, and that these players are asked to make passes within tighter defensive quarters than that of defensemen and sometimes forwards. Essentially "giveaways per 60" acts as a counting stat when a rate state might be more useful. Perhaps a better measure might be giveaway average, where the average is calculated as

- `giveaways / puck-possessions`

Using the player's number of possessions then accounts for the players who have the puck on their stick more often than others.

I think if we wanted to take this a step further we could weigh the types of giveaways differently, since I don't think the cost of a giveaway is as high in the offensive zone as it is in the neutral or defensive zone (although maybe the lost opportunity cost of not having a shot on goal balances this out a little bit). Giveaways in the offensive zones might lead to fewer goal opportunities for the opposition than those that occur in your own defensive zone. For this variation maybe the weighted average looks something like:

- `(.2 * OGA) + (.3 * NGA) + (.4 * DGA) / puck-possessions`
- Where `OGA` = offensive giveaways, `NGA` = neutral giveaways, `DFA` = defensive giveaways