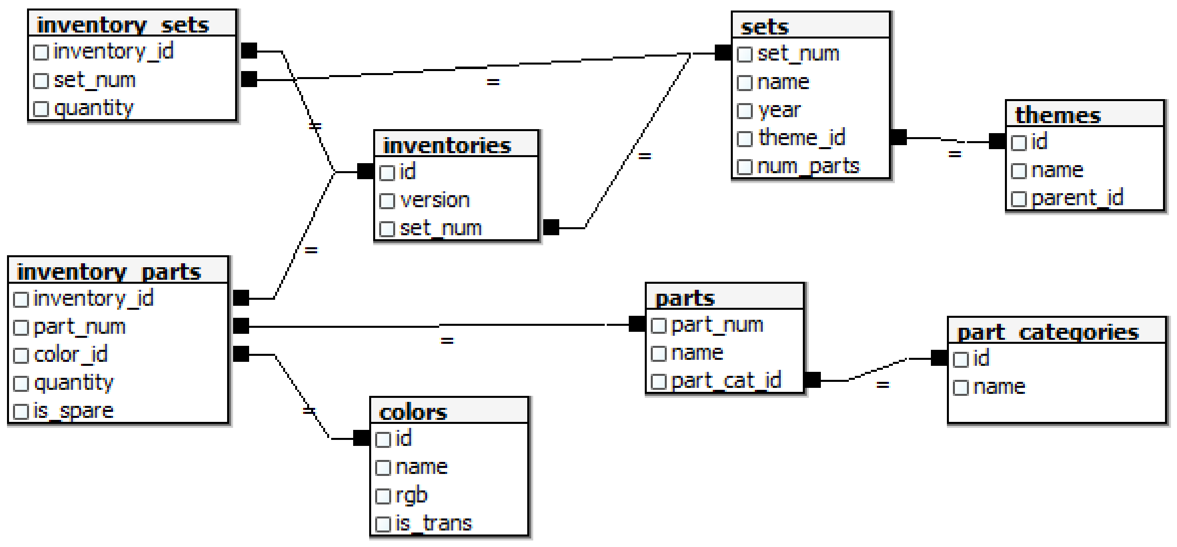
Advanced Distributed Systems Spring 2018 Project Description

- You will build an application that uses an eventually consistent, replicated, fault tolerant key-value store.
- That application will be implemented ONCE
- It will run in a JVM on your computer, and use the AWS SDK to make calls to an http accessible server which runs on an EC2 instance in AWS.
- The server component provides a single API to the client, but can be backed by 3 different eventually consistent Databases:
    - Amazon DynamoDB
    - Cassandra, deployed by you on a cluster of EC2 machines
    - **A new key-value store, implemented by you from scratch which runs on a cluster of EC2 machines and implements section 4.2 – 4.5 of the Amazon Dynamo paper, with the following modification:**
        - **R and W are always equal 1**
        - **N is configurable, but must be at least 2 and the system must work with arbitrarily large values of N**
- When the http accessible application server starts, it is configured via command line parameters to use one of the 3 stores listed above. The functionality of the application server must be the same regardless of which store it is using.
- The application server and client must both be built using Maven, and must include JUnit tests.
- When testing data store options #2 and #3, the unit tests must use the AWS SDK to kill some of the EC2 machines that the data store is running on and show that the system still works correctly and returns correct results
- The security of the system must be configured such that only the application server is accessible outside your VPC, but the data store EC2 machines are not.
- The two teams for this project (as chosen at random in class) are:
    - Team 1: Marty, Adam, Yehuda, Noah - https://github.com/Yeshiva-University-CS/ad-team-1
    - Team 2: Avi, Laivi, Mordechai, Aharon - https://github.com/Yeshiva-University-CS/ad-team-2
- **Each team must pick a team leader who will monitor the team's progress and submit weekly progress reports (starting April 12) and plans that spell out progress and tasks across the team (i.e. who is assigned to do what, and what progress have they made.)**
- **Each team must choose a time on May 14 or May 15 to demonstrate their whole project to me, including all aspects of their AWS configuration and deployment, tests, code, etc. We should allow at least an hour for that appointment/demonstration.**

Application Data



Download the data set from here: https://www.kaggle.com/rtatman/lego-database/data

Application Logic

- There are roughly 26,000 parts and 11,000 sets in the data set. You will load all that data into the **data store**.
- To keep things manageable and avoid the need for transactions and/or locking, we will make the connection between the application server and the data stores single threaded
- The **application server** accepts orders for lego sets from the client
  - When an order is received, the application server accesses the inventory_sets table to see if a set is available. If so, it reduces the inventory_sets.quantity of the set by one and replies to the customer with a message that the set has been shipped
  - If inventory_sets.quantity < the number of that set the customer ordered (e.g. the customer ordered 5 lego police cars and inventory_sets.quantity for the police car set is < 5), the application server checks the inventory_parts table to see if there is enough inventory of all the parts in the set to assemble enough sets to fulfill the order. If so, it "assembles" the sets by decrementing the inventory_ parts.quantity by the amount needed and replies to the customer that the sets have shipped.
  - If there are not enough parts, the application server:
    - Sends a message to the client that the set is "backordered"
    - creates a timer thread which counts 100 milliseconds for the required parts to be "manufactured." When the 100 milliseconds are up, the inventory_ parts.quantity for the part is incremented by 30.
    - Once all of a given order's manufacturing timers are done, the application server tries again to fill the order.
    - When an order is filled, the server will include an "order shipped" message to the client
- The **client** randomly picks a lego set to order and places a new order with the application server every 50 milliseconds.
- Many "request-response" connections can be created between client and server, but the client can not have more than 25 orders at a time that have not yet shipped.
- The application server and the data store you build will communicate with each other via serialized Java objects being sent over TCP connections. This can be done either using java.net.Socket and java.net.ServerSocket or using Netty.
- The client and application server can communicate either via TCP or via HTTP. You may use Netty or Jetty or HttpURLConnection.

Useful Information

- Amazon Dynamo SOSP 2007 Research paper
- Getting started on AWS
- Amazon SDK for Java
- DynamoDB Documentation, Overview of AWS SDK Support for DynamoDB
- Amazon EC2 User Guide for Linux Instances
- Amazon Virtual Private Cloud
- Cassandra:
  - Documentation
  - Getting started
  - Configuration
  - Operating Cassandra
  - Best Practices for Running Apache Cassandra on Amazon EC2
  - Apache Cassandra on AWS
  - Setting Up a Cassandra Cluster in AWS