Pizza Empire

# Overview

Pizza Empire is a casual building game targeting mobile devices. It’s similar to Hay Day but with the setting of a chain of pizza restaurants instead of a farm. The player starts with one small pizza restaurant and expands to multiple locations and delivery services to cover as much of the country. Players can also connect with other players.

# Technology

This will be an online game using a traditional client / server model.

## Goals

* Performant
* Highly scalable
* Secure
* Low possibility for cheating

## Tools

### Client

* Xamarin development tools.
* OR
* Native
  + iOS
  + Android
  + Windows mobile
  + Mac app store
  + Google app store

### Server

* .NET WEB API RESTful interface hosted on Azure.
* Azure cloud storage.
* Azure cloud caching.
* Azure MS SQL Database

## Design

* Client heavy to enable easier scalability.
* RESTful API for requests from client.
  + Given the nature of the game, the server will of course have to keep track of all of the client state, so I’m not sure if this is strictly RESTful.
* Use JSON and encryption to keep payload size to a minimum.
* Client storage to hold all of the known information in the phone.
* Data verification system built into API so that when client is making a request for data (for example when the client application is shut down and re launched), only a small amount of information is required to be transferred to verify if the data on the client side is correct.
  + Every single piece of player data would be need to be transferred from the server to the client in the case where the client side data verification fails for any reason other than network issues.
    - Client data corruption.
    - Problem with data synchronization between server and client.
    - Uninstall / reinstall of app.
  + The logic of whether the client needs to request all of the data will be left up to the client.
* Whenever a client makes a request to the API that obviously requires new data to be sent that the client couldn’t have had before, the local storage / server storage check doesn’t apply.
  + For example, when the client sends a request to order raw ingredients, new data will obviously be required from the server.
* Use some industry standard RESTful authorization.
  + Basic authentication w/SSL.
  + OAuth1.
  + OAuth2.
* The RESTful API will be versioned so that there is no problem with upgrades / older versions of the client still working.
* Multiple data caching strategies to deal with the different data access patterns for different types of data.
  + Requests from the client are likely to be extremely “write heavy” in that almost every request will change the state of the data associated with that player. This will make a traditional read data cache useless for this type of data.
    - In order to achieve acceptable performance, it’s likely that we will have to think of the storage as strictly long term storage for player specific state data.
    - It’s likely that writing to the storage every time a client makes a request will make it impossible to have acceptable performance.
    - One idea to explore is to load all of a client’s data from the storage into a cache when they log in and write it back to the storage and remove it from the cache in one chunk when they log out / become inactive for a period of time.
      * This idea will cause issues with data integrity and scalability.
        + What happens if the server crashes and all of the in memory data is lost?
        + How easy is it to implement scaling caches?
      * Investigate Azure cloud caching services to see if using this service can address both the data integrity and scalability issues.
      * This might mean that all client data is actually stored in a blob of binary data in the storage instead of in a relational mapping.
      * This idea could be modified so that client specific data is only loaded from the storage into the cache as it is requested by the client instead of all at once.
        + If most of the client data is required in order to play the game then this might be less efficient because it will require more RESTful API requests.
        + This will require performance testing.
    - Need to investigate other possible designs to handle the described scenario.
  + Data that relates to the players but can’t be strictly associated with only one player.
    - For example, items that a player has made available for trade or leaderboard stats.
    - Need to think about / investigate best data access design for this type of data.
  + Data that is not player specific and likely to be accessed very frequently.
    - Should reside in an in-memory cache and only be read from the storage whenever absolutely necessary (when the server application is launched on a server for any reason).
    - For this data we can think of the data in the database as the source of truth, which will be used when a server instance is launched.
    - This data should never change except when a new version of the game is released.
    - For example, the data about the raw ingredients in the game would fall into this category including wait times for delivery, names, prices, etc. etc.
* The data access layer should be built so that any logic layers don’t care whether they are interacting with the storage or the cache.
* Respecting the above, there also needs to be a way to specify and manage the different caching strategies mentioned above: always keep in memory, pull from storage / push to storage when player logs in / logs out, shared player information.
* Need to think a lot more about the release /upgrade process for both client and server.

## Data classes

We will need separate classes for data that is:

* Currently in use in the game
* Stored in table storage
* Stored in database storage
* Stored in the cache

We will need a way to map data to / from the storage classes to / from the in use classes.

## Logging / Exception handling

We need to implement logging of problems and exceptions at the service level. Exceptions should then be re-thrown to be handled by the application level.

## Service availability

We need to

* Implement retry policies in the code for all of the cloud services. The number of retries should be configurable.
  + Table storage service
  + Cache service
  + DB storage service
* Design the ability to scale back / not use services that we can configure on the fly.
  + E.g. if the cache service is down, the server should still work if we tell it to not use the caching service. Data would then flow directly to table storage or db storage.
  + If the table storage server or db server is down, some parts of the game should still work.
  + We need to investigate how to do the scaling back / throttling.

# RESTFul API

The RESTful API should be based around the actions that the players will take.

## List of player actions

* Login / Start Game
* Order raw ingredient
* Prepare processed ingredient
* Assemble / cook pizza
* Take phone calls
* Perform delivery
* Seat customer
* Sell product to customer
* Upgrade equipment
* Upgrade restaurant
* Hire employee
* Check leaderboard
* View trophies
* Communicate with other players
  + Trade
  + Chat