* Microservice design pattern: *controller* calls *service*, *service* calls *dao*.   
    
  Usually a *service* file only calls its own *dao*, if it needs to call the functions from other *dao*'s, call their *service*'s instead. Otherwise, cache from *service* files cannot be fetched.
* Implement a **Result** class to encapsulate information from server side.  
    
  Implement a **Key** class to get the key for accessing database.
* For serialization, use **Fast.json** instead of Protocol Buffer for better human data readability.
* Apply [MD5](https://en.wikipedia.org/wiki/MD5) algorithm twice to the plaintext passwords at user login to protect data transfer: MD5\_server(MD5\_client(pass + salt) + random salt).
* To check the validity of user input at Login, use [JSR 303](https://beanvalidation.org/1.0/spec/) to implement a validation annotation; then, allocate a exception package to catch the exceptions thrown.
* **Distributed Session:** after the user logs in, generate a *session ID* for the user, write it to cookie and pass it to the server. The server then takes this specific ID to fetch data for the user from database. Therefore, each session does not directly store data to the server, but instead to our cache managed by Redis.  
    
  When the user visit the website before the corresponding token expires, the project extends the token's expiration time by adding a new one to the database.  
    
  This will help in future scaling and expansions to keep data consistency.
* To keep the databases easy to maintain and ensure database performance, separate tables for different sales events from the regular product table.
* After implementing the basic product sales functionalities, use JMeter to conduct load test and measure the system's performance; use custom variables to simulate real-world users. As for Redis, use [redis-benchmark](https://redis.io/topics/benchmarks) for testing.  
  JMeter Linux command:

$ sh jmeter.sh -n -t XXX.jmx -l result.jtl

Redis-benchmark Linux command: ```sh $ redis-benchmark -h HOST -p PORT -c CONNECTION -n REQUEST $ redis-benchmark -h HOST -p PORT -d DATA ``` 

**Redis Load Test Result:** about 65k QPS for 100 connections and 100,000 total requests.   
**JMeter Load Test Result:** only around 1.2k QPS for 5000 concurrent threads \* 10 iterations :(   
This shows that the bottleneck for the system is at the MySQL database.  *top* command: monitor system resource usage & storage.

* Now moving to optimizations: first, utilize caching methods to lower the pressure of database. More specifically, apply **Page Caching, URL Caching and Object Caching** (listed in order of granularity level)  
    
  For HTML template page caching, the caching period should be relatively short: page caching mainly aims to deal with large concurrent requests in an extreme short period, therefore storing the HTML templates in cache for a long time might result in data inconsistency.  
    
  On the other hand, Object Caching focuses on a single user, and its cache should exist permanently rather than expiring fast. The only case to make changes to the cache is when the user updates his/her password.  
    
  After applying these caching methods, the service's QPS increased to 30k.
* To further optimize the system's performance, make the **dynamic HTML pages static**. HTML pages can be stored in the user's browser, therefore decreasing the queries to the database. Modern language and tools, such as AngularJS and Vue.js are all implemented in this manner.   
    
  The client would ask the server whether data have been updated since a specific time, and upon receiving a 304 code indicating no update, it will directly fetch the cache from the browser.   
    
  However, this still requires some communication between the server and client. We can completely get rid of this by manually configure our program such that the browser will directly ask its cache without querying the server in a specific time period we allocated.
* During load test, products are being **oversold**. That is, inside the database, remaining stock in warehouse of the products became negative.   
    
  To deal with this problem, check the stock count in database to make sure it is larger than 0 everytime a purchase is made. (When the database is updated by a thread, it will be automatically locked, so race conditions won't happen and can be ignored)
* Caching improved the system's performance in some degree. However, for an extreme number of concurrent queries, this is not enough on its own. More methods are needed to further optimize the system by reducing queries to database. This can be done with **Redis** and **asynchronous programming**.
* **Nginx horizontal scaling：** when the scale of the system increases, expand the system with [Nginx](https://www.nginx.com).  
    
  With the help of previous optimizations, Nginx would provide a powerful tool to expand the system while ensuring performance.   
    
  As the system grows even larger, apply (LVS)[<http://www.linuxvirtualserver.org>] for further scaling.