Design Defects and Restructuring

Engr. Abdul-Rahman Mahmood

 abdulrahman@nu.edu.pk

 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786

 http://alphapeeler.sf.net/me

 alphapeeler#9321

 *reddit.com/user/alphapeeler*

 www.flickr.com/alphapeeler

 http://alphapeeler.tumblr.com

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood_cubix  48660186

 alphapeeler@icloud.com

 pinterest.com/alphapeeler

 www.youtube.com/user/AlphaPeeler

10 Major mistakes in Architecture Design and Software Development

Credit: **Vijayananthan JC**

Enterprise Solution Architect at Tata Consultancy Services

Service Chattering

- API approach between two eco system that leads to port exhaustion and other bulk traffic related issues.

- We all align towards making API calls between multiple system to finish the job by considering the **loose coupling Architecture** pattern

- ***Best way to avoid this mistake*** *is to assemble all the employee details as a Json list or XML and send it to one of the API parameters for further processing (instead of calling an API for each employee) thereby reducing the overall pipeline traffic.*

Long polling into persistence storage

- Developers are in need to compare, update or insert list items in a loop to any persistence storage. (10 employees to be compared with 100 available in DB and process business logic on matching records.)

- ***Best way to avoid this mistake*** *is to do a onetime call and populate the in-memory list then compare with the incoming items for reducing the database traffic thereby increasing the performance as well.*

Heavy weighted API

- Prolonged API Processing.
- Situation where a majority of business logic gets embedded into the API's business layer in due course of time of project development without considering the overall API Architecture in mind thereby allowing the API to execute for quite some time (**more than 90 seconds**) before a response is generated to the calling system which would be one of the biggest mistakes in Application design.
- ***To avoid this mistake*** *is to isolate heavy lifting work to a background job or a similar Architecture so that the API's purpose is to just receive and acknowledge thereby handing over the rest of heavy lifting task to another component called "* ***The Job Processor****" This levels the load to the processor from API.*

Dedicated Read and Write channel

- It's a drawbacks of Not implementing Command and Query segregation pattern in a system.
- Most of our production projects lack in proper implementation of dedicated read and write channels for CRUD operations due to delivery timelines which in turn puts the primary data source under heavy load for all (Read, Insert, Update and Delete) operations and leaving the secondary data source idle thereby affecting the overall performance of the system. This limits the overall scalability of the system.

- ***Best way to avoid this mistake*** *is to route the traffic through dedicated designated channels. A best example would be routing all the read traffic through a separate read query pointing to secondary data source and all write (insert, update, delete) operations to primary source with auto seeding enabled in asynchronous mode thereby we divert all heavy read traffic to secondary data source.*

Justify your API's

- Unnecessary usage of API.
- In some cases, the requirement would be to interact with only one system and as per standards we will implement an API layer for such interaction and process the business logic which is actually an overkill.
- All logic check performed would now require an additional latency call to API then retrieve data, process business logic and display it which adds to performance strain in the overall system.
- ***To avoid this mistake*** *is to justify whether an API is required to be in place or not in a system design. Instead of placing the business logic and data retrieval logic inside the API, consider creating a separate layer inside the api hosting system.*

Maintain thin Cache strategy

- Dumping more data into Cache for better performance vs DB calls.
- Mistakes we tend to do is to load non-static data into cache for increasing the system performance thereby avoiding database calls that puts the entire system at risk.
- During system restarts the entire cache is rebuilt due to the missing reference of index into the system that takes the system downtime to a higher period thereby reducing the overall system stability.

- ***Best way to avoid this mistake*** *is to make a balance between cache and data source and do a distribution of data retrieval using Command and Query segregation implementation. Utilizing secondary source for cache re-build on important system data also mitigates the issue to some extent.*

Horizontal Partitioning of Data

- Let's assume there are 2 lakh employees available in the database and searching an employee information always puts an extra load in the querying source.

- ***Best way to avoid this mistake*** *is to make a logical partition of the data horizontally. Instead of loading all employee detail into a single source, categorize the employee detail into different sources. (Employees 1 – 50,000 in database 001, Employees 50,001 to 1,00,000 in database 002 etc.) and route the traffic based on the employee ID in the above example thereby configuring the data source to handle huge traffic and load.*

Call back Design a big NO

- Avoid call back response to ensure that processing is completed.
- While communicating with two systems for processing a bunch of data and getting the processing response from the target system might be a challenging task **due to multiple packet loss**.
- Assume we are sending a list of 100 employees for processing an address update in the target system. Let's assume we design a system call back intimation to source system as soon as the address is completed for each employee. We would end up receiving only 90 employees instead of 100 processed employees in the target system due to various issues like latency, system failure etc.

- ***Best way to avoid this mistake*** *is to make a check point call to the source system say at every 100th processing employee and intimate the processing completed status or perform a lazy polling to the target system with number of processed employees at a given time*

Overloading the input stream of an API

- Sending too much data in an API?
- Sending a list of 1000 employees to process an address update is a big NO in designing an API which involves in timeout exceptions and latency issues over the internet.
- ***Best way to avoid this mistake*** *is to update the list of employees into a flat file similar to CSV or XML and pass on the metadata information like employees count, information download URL and time to live information so that the target system downloads the necessary information and processes the same thereby reducing the overall latency charges*

Proper cleanup of data source

- **Compensating Transaction pattern** is a basic need.
- This pattern should be implemented in systems where operations must be undone upon failure.
- A best example would be a travel website lets customers to book itineraries. A single itinerary might comprise a series of flights and hotels. As a part of itinerary there might be many connecting flights booked before the final destination flight is confirmed, but if the final destination flight is not confirmed or cancelled then system needs to clean up all the dependent flights to maintain a clean system automatically.

- ***Best way to avoid this mistake*** *is to implement a Compensating transaction pattern to properly clean up the left-out records upon a transaction failure thereby unused records would be eliminated*

Readings

- A case study:
- https://dwheeler.com/essays/heartbleed.html

- In his essay "How to Prevent the next **HeartBleed**" David Wheeler said "OpenSSL uses unnecessarily complex structures, which makes it harder for both humans and machines to review." There should be a continuous effort to simplify the code. Otherwise, just adding capabilities will slowly increase software complexity. The code should be **refactored over time** to make it simple and clear while new features are being added. The goal should be code that is "obviously right," as opposed to code that is so complicated that "I can't see any problems."