Technical Solution Document - 4

Author- Alexander Adu-Sarkodie

MSc Telematics MSc Engineering

LinkedIn - <https://www.linkedin.com/in/alexander-adu-sarkodie-063b4b4a/> GitHub - https://github.com/kukuu?tab=repositories

Addendum

These are further steps with capability of enhancing the application’s usability and performance.

Performance

*Progressive Web App Performance*

Progressive Web Apps are judged by several performance standards, including Time to Interactive and First Meaningful Paint.

Time to Interactive - This audit identifies the time at which a page appears to be ready enough that a user can interact with it.

First Meaningful Paint - First Meaningful Paint is essentially the paint after which the biggest above-the-fold layout change has happened, and web fonts have loaded.

*Tools*

Capacitor: Native Progressive Web Apps

Capacitor is a cross-platform app runtime that makes it easy to build web apps that run natively on iOS, Android, Electron, and the web. We call these apps "Native Progressive Web Apps" and they represent the next evolution beyond Hybrid apps.

Capacitor provides a consistent, web-focused set of APIs that enable an app to stay as close to web-standards as possible, while accessing rich native device features on platforms that support them. Adding native functionality is easy with a simple Plugin API for Swift on iOS, Java on Android, and JavaScript for the web.

*Running Natively and on the Web*

One of the key features of Capacitor is the ability to build one app that runs both natively (in the app stores), and on the web. Capacitor does this by providing a layer between the underlying platform and the APIs/Plugins you'd like to use.

If your app makes native plugin calls that don't have a web substitute, such as SplashScreen.show(), the app will allow those calls without crashing. Calls that return a promise will return a rejected promise, which you should be handling in your app anyways.

Additionally, Capacitor's JavaScript API has a number of utilities that make it possible to programmatically check whether certain APIs are available. It will then handle gracefully negative responses (promises) without the application breaking. This will be done programmatically.

For example, if your app would normally rely on the Camera app being used to take a photo, you could check if the Camera is available, and if not, ask the user to upload a file instead:

Data Normalisation

**Data Normalisation**

Reading through the task statement, there is substantial redundancy of data handling between the individual cookery books, and for example the current rights management system. This calls for a strategy to optimise and normalise the database tables and the relationship between them.

Database normalisation is the process of organising the fields and tables of a relational database to minimise redundancy and dependency. Normalisation usually involves dividing large tables into smaller (and less redundant) tables and defining relationships between them. The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships.

The logical design of the database, including the tables and the relationships between them, is the core of an optimised relational database. A good logical database design can lay the foundation for optimal database and application performance. A poor logical database design can hinder the performance of the whole system.

Benefits of Normalisation include:

* Reasonable normalisation frequently improves performance. When useful indexes are available, the DB Server query optimiser is efficient at selecting rapid, efficient joins between tables.
* Faster sorting and index creation.
* Narrower and more compact indexes.
* Fewer indexes per table. This improves the performance of the INSERT, UPDATE, and DELETE statements.
* Fewer null values and less opportunity for inconsistency. This increases database compactness.

Note as normalisation increases, the number and complexity of joins required to retrieve data also increases. Too many complex relational joins between too many tables can hinder performance. Reasonable normalisation frequently includes few regularly executed queries that use joins involving more than four tables

We can optimise and normalise the data by creating foreign keys in the rights management system, and removing the corresponding images from the rights system. The foreign keys from the rights management system will be used to index the images in the cookery books.

Modeling Recipe Data

To design an efficient and structured database management system, it is necessary to have the following measures and procedures in place. The existing model will be crossed checked against these bench parameters and others.

* Descriptive name for your database
* Proper Identification of unique Objects – An object is an entity and a representative of the table in the database, and binds attributes of the table to itself. Here, we begin abstraction for the table. High level aggregation for the table.

The object list could be

* + recipe id
  + recipe name
  + A picture
  + cost
  + category
* Define and name the table for each object. For example Avocado subscribers
* Identify the attributes for each object in the table For example,
  + recipe id
  + recipe name
  + picture
  + cost
  + category
* Define and name columns. For example,
  + recipe\_id
  + picture
  + price
  + category
* Identify the Primary Key. Say, recipe\_id for cookery files table and book\_ID for the book table. These keys will make each record unique as they persist.
* Define the defaults
* Identify columns with required data. These columns should never be allowed to be empty. For example picture and price.

Health Checks

* Keep database trim.
* Archive old data – to remove excessive row returns or searches on queries.
* Put indexes on your data.
* Do not overuse indexes, compare with your queries.
* Compress text and blob data types – to save space and reduce number of disk reads.
* UTF 8 and UTF16 is slower than latin1.
* Use Triggers sparingly.
* Keep redundant data to a minimum – do not duplicate data unnecessarily.
* Use linking tables rather than extending rows.
* Pay attention to your data types, use the smallest one possible for your real data.
* Separate blob/text data from other data if other data is often used for queries when blob/text are not.
* Check and optimise tables often.
* Sometimes, it is faster to drop indexes when adding columns and then add indexes back.
* Use different storage engines for different needs.
* Use ARCHIVE storage engine for Logging tables or Auditing tables – this is much more efficient for writes.
* Store session data in memcache rather than MySQL – memcache allows for auto-expiring values and prevents you from having to create costly reads and writes to MySQL for temporal data.
* Use VARCHAR instead CHAR when storing variable length strings – to save space since CHAR is fixed length and VARCHAR is not (utf8 is not affected by this).
* Make schema changes incrementally – a small change can have drastic effects.
* Test all schema changes in a development environment that mirrors production.
* Do NOT arbitrarily change values in your config file, it can have disastrous affects.
* When in doubt use a generic config file.

Query Optimisation

* Use the slow query log to find slow queries.
* Use EXPLAIN to determine queries are functioning appropriately.
* Test your queries often to see if they are performing optimally – performance will change over time.
* Avoid count(\*) on entire tables, it can lock the entire table.
* Make queries uniform so subsequent similar queries will use query cache.
* Use GROUP BY instead of DISTINCT when appropriate.
* Use indexed columns in WHERE, GROUP BY, and ORDER BY clauses.
* Keep indexes simple, do not reuse a column in multiple indexes.
* Use a LIMIT on UNION instead of OR for less than 5 indexed fields.
* Use INSERT ON DUPLICATE KEY or INSERT IGNORE instead of UPDATE to avoid the SELECT prior to update.
* Use a indexed field and ORDER BY instead of MAX.
* Avoid using ORDER BY RAND().
* LIMIT M,N can actually slow down queries in certain circumstances, use sparingly.
* Use UNION instead of sub-queries in WHERE clauses.
* For UPDATES, use SHARE MODE to prevent exclusive locks.
* On restarts of MySQL, remember to warm your database, to ensure that your data is in memory and queries are fast.
* Use DROP TABLE then CREATE TABLE instead of DELETE FROM to remove all data from a table.
* Minimize the data in your query to only the data you need, using \* is overkill most of the time.
* Consider persistent connections instead of multiple connections to reduce overhead.
* Benchmark queries, including using load on the server, sometimes a simple query can have affects on other queries.
* When load increases on your server, use SHOW PROCESSLIST to view slow/problematic queries.
* Test all suspect queries in a development environment where you have mirrored production data.