LogikSutraAI Case Study

Udit Srivastava

Contents

1 1. Mobile App (React Native)

1.1 Problems

- Slow app while opening a list of the content.
- Images becoming a burden to load.
- Users drop off during onboarding.

1.2 Proposed Improvements

- Optimize lists using FlatList caching and skeleton loaders.
  - The problem with FlatList is that while scrolling, it **keeps re-rendering** the items unnecessarily, which **causes lag and high memory usage**.
  - Using **FlatList Caching** can help optimize the performance.
  - **Advantages of using Flatlist caching**
    - It **reuses** already rendered items
    - keeps **some items in memory**, so we don't lose them while scrolling back.
  - **Skeleton Loaders** are placeholder UI that is shown while real content loads.

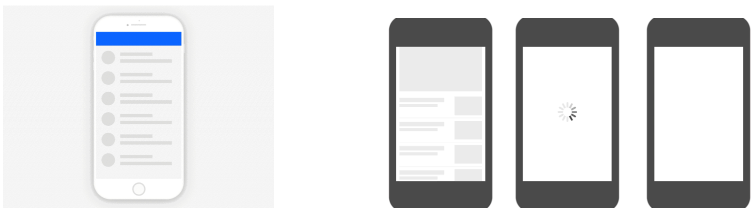

Figure 1: above two Images show some standard skeleton loaders

  - Instead of showing a blank screen, it is better to show skeleton of UI which can also keep user engaging for sometime.
  - **Advantages of using Skeleton Loaders**
    - Users feel the app is **responsive even when the data is not ready**.
    - Even if the API takes **longer time**, the user feels **content is coming quickly**.
    - Final content replaces skeleton smoothly.
- Compress and use faster image formats (**WebP**).
  - WebP can compress images **25-35% smaller** than JPEG/PNG with same quality, which results in faster loading and lesser data usage(that is **lower latency and reduced bundle size**)
  - Using WebP, results in faster rendering in app, which provides smoother scrolling and reduces memory usage.
  - it also supports Transparency and Animation(like PNG)

- Shorten onboarding with social login and fewer steps, also allow for ”Guest” or ”Skip for now” options.
  - Instead of asking users to fill in multiple fields (name, email, password), let them sign up with one tap.(like ”continue with google”)
  - Allow users to explore the app without full sign-up, Once they find value (like a course, chat, or content), then Aask them to create an account.

1.3 Why it Matters(Impact)

- **Users’ first time engagement** with the app **matters** because that is the first impression of the app on the user.
- It can decide whether user gonna keep the app or uninstall it.
- Also a **faster and smoother app** keeps the user **engaged with the content**, eventually making **user habitual with the app**.

2 Backend (Node.js, Express, MongoDB, Kafka-based Microservices)

2.1 Problems

- If the dataset has large number of records every seacrh becomes slower, MongoDB queries slows down as data grows.
- Services are tightly connected, making it difficult to scale and debug.

2.2 Proposed Improvements

- Add Indexes in MongoDB
  - Indexes are used as shortcuts for searches, Instead of checking every row, MongoDB quickly jumps to the right data, results in much faster queries.
- Separate tightly connected services into microservices
  - we can split backend into smaller independent services
    - **Auth Service:** handles login or signup
    - **User Service:** manages User profiles
    - **Courses service:** handles courses
    - **Notification service:** sends alerts
  - Each service can be individually scaled and debugged.

2.3 Why it Matters(Impact)

- Queries don't lag**(faster Backend)**
- If one service fails, it doesn't collapse whole system**(more reliable)**
- We can handle more users by scaling individual services that need to be scaled for handling more users.

3 Infrastructure (AWS + Kubernetes)

3.1 Problems

- Running everything on one server is risky and costly with high traffic.

3.2 Proposed Improvements

- **Use Kubernetes (EKS) for Auto-scaling**
  - instead of running everything on one server, we can **run App in containers.**
  - Kubernetes **automatically adds/removes containers when traffic goes up or down**.
- **Backups in Another AWS Region**
  - Regularly back up your database/files.
  - Keep a copy in another AWS region.
  - so that If one region fails, you can quickly restore from backup.

3.3 Why it Matters(Impact)

- **Scalability**(App can handle more users)
- **Cost Efficient**
- data remains **safe** even if a server crashes.

4 Security

4.1 Problems

- APIs may be open to get used by anyone
- Secrets (like passwords or keys) could leak

4.2 Proposed Improvements

- **JWT tokens and Rate Limiting**
  - **JWT(JSON Web Tokens)** are secure way to check if users are authenticated.
  - **Rate limiting** restrict how many times an API can be called per minute and hence blocks abuse.
- **AWS Secret Manager**
  - Instead of hardcoding keys in code, store them securely in **AWS Secrets Manager**.
  - App fetches them safely at runtime.
- **Security Checks in CI/CD**
  - **Continuous Integration(CI)** ensures the new code snippet which is going to be merged with the old one will not occur any bugs and will not make the app unstable.
  - After CI, **Continuous Deployment(CD)** automatically deploys the app to production, ensuring the latest version of app is always running.
  - So everytime the new code is pushed, **automatic checks** occur for **dependencies vulnerabilities, secret failures, test failures**.

4.3 Why it Matters(Impact)

- It **protects** User Data(like passwords, personal info.)
- **Prevents attacks** like brute force or API spamming.
- Keeps the system **safe**

5 System Design and Scaling

5.1 Problems

- Traffic will increase significantly if app expands
- Without preparation, this will cause slow response times, crashes, or downtime.

5.2 Proposed Improvements

- **Add Cachings(Redis, CDN)**
  - **Redis stores frequently used data** in memoery, from where we can retrive data very fast.
  - **CDN (like CloudFront) serves images and videos** very quickly to users worldwide.
- **Using Monitoring Tools**

    - Use **Prometheus, Grafana, CloudWatch** to **track CPU, memory, errors and latency**.
    - Prometheus **collects metrics**(like CPU usage, Memory request, latency) from our app servers.
    - Grafana **displays data collected from Prometheus** in a nice daashboard(like a graph showing latency spikes when traffic increases)
    - AWS CloudWatch **can trigger alarm, auto-scalling and notifications if something goes wrong**.
    - All of them combinely helps in detecting problems before users complain.
- **Real Time Features**
    - Use **AWS appSync** for **real-time chat and notifications.**
    - AWS AppSync manages service that **uses GraphQL** subscriptions and also **WebSockets** can be used to **handle real-time updates** easily.
    - Instead of 10,000 users sending requests every 5 seconds, **WebSockets ensures each user keeps one open connection** and only actual updates are sent.
    - Server only sends data when something changes.
    - Even with large number of users, System runs smoothly.

5.3 Why It Matters(Impact)

- We can see high CPU or memory usage and can improve it before users feel slow performance**(Catches Issues Early)**
- CloudWatch can auto-scale infrastructure when traffic increases.**(Automated actions when needed)**

6 Summary

- **Mobile App:** Optimize list with Flatlist Caching and skeleton loaders,use compressed images or better formats(WebP), and onboarding with fewer steps.
- **Backend:** Use microservices, optimize MongoDB,
- **Infrastructure:** Use AWS services, scaling, and backups
- **Security:** Secure APIs(using JWT tokens and rate limitings), safe secrets(using AWS secret manageer), dependency scanning(CI/CD checks)
- **Scaling:** Add caching(Use Redis,CDN), monitoring, and real-time communication strategies