Presenter: Leijun Jiang

Rails Best Practices

Building Scalable, Secure, and Efficient Applications

1. Database Best Practices
2. Async Jobs & Sidekiq
3. Business Logic & Security
4. Caching Strategies
5. Code & Performance Optimization

Database Best Practices

- **Indexing:**
  - Avoid indexing low-cardinality fields (e.g., `status`, enums). < 100
  - Example: `status: ['pending', 'completed']` → Indexing is inefficient.
  - Index maintenance cost vs. benefit.

Database Best Practices

- **Migrations:**
  - Review migrations before deployment.
  - Handle large tables manually
    1. 1M rows: seconds of lock
    2. 10M rows: minutes of lock
    3. 100M+ rows: potential hours of lock
  - Zero-downtime migration strategies:
    1. Add column without default.
    2. Backfill data in batches.
    3. Add index concurrently (PostgreSQL).

Database Best Practices

- **Transactions:**
  - Avoid starting transactions too early (e.g., in `before_action`).
  - Example: Move HTTP calls outside transactions.

Database Best Practices

```ruby
class OrdersController < ApplicationController
  # Starts transaction here with user query
  before_action :authenticate_user!
  # Long HTTP call while transaction is open
  before_action :fetch_external_data

  def create
    # DB operation inside transaction
    @order = Order.create!(params)
  end
end
```

Database Best Practices

```ruby
class OrdersController < ApplicationController
  skip_before_action :authenticate_user!
  def create
    # Do HTTP calls first
    external_data = fetch_external_data
    # Then start transaction with auth
    authenticate_user!
    # DB operations last, keeping transaction short
    @order = Order.create!(params.merge(external_data: external_data))
  end
end
```

Database Best Practices

- **Unique Constraints:**
  - Always enforce uniqueness at the database level.
  - Example: `add_index :users, :email, unique: true`.
  - Model-level validation is not enough → Race conditions in high concurrency.
  - Data duplication is hard to fix.

Async Jobs & Sidekiq

- **Idempotency:**
  - Ensure database operations are idempotent.
  - Place non-idempotent operations (e.g., external API calls) at the end.
- **Error Handling:**
  - Avoid overwhelming error emails during retries.
  - Use defensive code to handle corrupted data (e.g., soft-deleted records).
- **Performance:**
  - Ensure Sidekiq jobs finish within 10 seconds (k8s OOMkill threshold).

Business Logic & Security

- **Authorization:**
  - Never trust IDs from the frontend.
  - Always verify ownership:
    - Use `current_user.orders.find(params[:id])` instead of `Order.find(params[:id])`.
- **Security:**
  - Use UUIDs for IDs to prevent guessing attacks.
  - Never hardcode secrets (e.g., SMS templates, JWT secrets).
  - Use `Rails.application.credentials` or environment variables.

Caching Strategies

- **Don’t abuse Rails.cache methods**
- **Two-Level Caching:**
  - Avoid storing large data (>1KB) directly in Redis.
  - Example:
    1. Store metadata/IDs in Redis.
    2. Fetch actual data in separate keys.
  - Benefits: Reduced memory usage, better performance.

Code & Performance Optimization

- **Development Process:**
  - Write design docs, get reviews, then code.
- **Performance Issues:**
  - Most issues come from database or external dependencies.
    i. Check for N+1 queries first.
    ii. Implement monitoring.
- **Memory-Intensive Tasks:**
  - QRCode, Image, Excel, PDF generation → Use serverless (e.g., AWS Lambda).
    i. memory consumption at runtime
       1. PDF >= 300MB
       2. Image >= 100MB
       3. Excel >= 100MB

Conclusion

1. **Database:** Optimize indexing, migrations, and transactions.
2. **Async Jobs:** Ensure idempotency and handle errors gracefully.
3. **Security:** Verify ownership, use UUIDs, and avoid hardcoding secrets.
4. **Caching:** Implement two-level caching for large data.
5. **Performance:** Monitor and optimize database queries, offload heavy tasks.