Reddit Clone

Complete Implementation with REST API

Team Members	UFID
Dinesh Reddy Ande	58723541
Saaketh Balachendil	86294284

Date: December 03, 2025
Course: COP5615 - Distributed Operating Systems Principles

Demo Video: https://youtu.be/Ua3XxFmMgmM

Table of Contents

1. Executive Summary

This project implements a fully functional Reddit-like social media platform in two parts: an actor-based engine using Gleam's OTP (Part 1) and a REST API interface (Part 2). The system demonstrates advanced distributed systems concepts including concurrent actor models, RESTful API design, and high-performance message passing. The implementation successfully handles up to 100,000 concurrent users while maintaining data consistency and system responsiveness.

Key Features

Component	Description
REST API Endpoints	18 endpoints covering all Reddit functionality
Concurrent Users	Tested up to 100,000 simultaneous users
Peak Throughput	746.76 operations/second at 50K users
Architecture	Actor-based with OTP supervision trees
API Design	RESTful with JSON responses
Command-line Client	15+ commands for complete API coverage

2. System Architecture

2.1 High-Level Architecture

The system follows a layered client-server architecture with three distinct layers:

Layer	Technology	Responsibility
Client Layer	HTTP/REST clients	User interaction, HTTP requests
API Layer	Wisp + Mist	HTTP handling, JSON serialization, routing
Engine Layer	Gleam OTP Actors	Business logic, data management, concurrency

2.2 Communication Flow

Request Flow: Client → HTTP Request → REST API Server → Message Passing → Engine Actor → Process → Response → JSON → Client

Response Flow: Engine completes operation → Sends reply to API handler → JSON serialization → HTTP response → Client receives data

2.3 Actor Model Architecture

The engine uses the Actor Model for concurrency, where each component is an independent actor that communicates via asynchronous message passing. This design provides:

• **Isolation:** Each actor has its own state, preventing race conditions

• **Concurrency:** Actors process messages independently and in parallel

• **Fault Tolerance:** Actor failures don't cascade to other components

• **Scalability:** Easy to spawn thousands of concurrent actors

• **Message Passing:** Asynchronous, non-blocking communication

3. Part 1: Actor-Based Engine

3.1 Core Components

Component	Role	Key Features
Engine Actor	Central coordinator	Manages all entities, routes messages, enforces business rules
User Actors	User processes	Short-lived, handle user-specific operations, lifecycle management
Data Structures	State management	Dict-based storage for O(log N) lookups
Message System	Communication	Asynchronous message passing with reply subjects

3.2 Functional Features

User Management: Registration, karma tracking, online/offline status

Subreddits: Creation, joining/leaving, member tracking, Zipf distribution

Content: Posts, hierarchical comments, threaded discussions

Voting: Upvote/downvote with real-time karma updates

Feeds: Personalized feeds based on subscribed subreddits

Direct Messaging: User-to-user messages with threading

3.3 Zipf Distribution

The system implements Zipf's law (s=1.0) to simulate realistic social network behavior. This power law distribution ensures that a small number of users and subreddits account for the majority of activity, mimicking real-world platforms where top users generate most content and popular subreddits dominate traffic. This is achieved using Erlang's rand:uniform() for randomness and math:pow() for power calculations.

4. Part 2: REST API Implementation

4.1 Technology Stack

Component	Technology	Version	Purpose
Web Framework	Wisp	2.1.0	Request routing, middleware, response handling
HTTP Server	Mist	5.0.3	High-performance Erlang-based HTTP server
JSON	gleam_json	3.1.0	Serialization and deserialization
HTTP Client	gleam_httpc	5.0.0	Making HTTP requests (client simulator)

4.2 API Server Architecture

The REST API server acts as a bridge between HTTP clients and the actor-based engine. Key architectural decisions include:

• **Stateless Design:** Server doesn't maintain session state, all state in engine

• **Synchronous Communication:** API handlers wait for engine responses (2s timeout)

• **JSON-First:** All requests and responses use JSON for consistency

• **RESTful Principles:** Resource-based URLs, proper HTTP methods and status codes

• **Error Handling:** Graceful error responses with consistent format

4.3 Request Processing Pipeline

Step	Component	Action
1	Wisp Router	Match URL path and HTTP method
2	Handler Function	Parse form data from request body
3	Message Creation	Build engine message with reply subject
4	Engine Send	Send message to engine actor via process.send()
5	Wait	Block on process.receive() with 2s timeout
6	JSON Encoding	Serialize engine response to JSON
7	HTTP Response	Return JSON with appropriate status code

5. API Design and Endpoints

5.1 RESTful Design Principles

The API follows REST architectural constraints and best practices:

• **Resource-Based URLs:** /users, /posts, /subreddits represent resources

• **HTTP Methods:** GET (read), POST (create), PUT (update), DELETE (remove)

• **Stateless:** Each request contains all necessary information

• **Hierarchical Structure:** /subreddits/{id}/posts, /posts/{id}/comments

• **Consistent Responses:** All responses follow {success, data/error} format

• **Query Parameters:** Used for search operations (?q=query)

5.2 Complete Endpoint List

User Management (3 endpoints):

Method	Endpoint	Description
POST	/users	Register new user
GET	/users/{id}/karma	Get user karma score
GET	/users/{id}/feed	Get personalized feed

Subreddit Management (4 endpoints):

Method	Endpoint	Description
POST	/subreddits	Create new subreddit
PUT	/users/{id}/subscriptions/{sub_id}	Join subreddit
DELETE	/users/{id}/subscriptions/{sub_id}	Leave subreddit
GET	/subreddits/{id}/members	Get member count

Content & Voting (5 endpoints):

Method	Endpoint	Description

POST	/subreddits/{id}/posts	Create post
POST	/posts/{id}/votes	Vote on post
POST	/posts/{id}/comments	Create comment
POST	/comments/{id}/replies	Reply to comment
POST	/comments/{id}/votes	Vote on comment

Messaging, Search & System (6 endpoints):

Method	Endpoint	Description
POST	/dms	Send direct message
GET	/users/{id}/dms	Get messages
GET	/search/usernames?q={query}	Search users
GET	/search/subreddits?q={query}	Search subreddits
GET	/metrics	System statistics
GET	/health	Health check

6. Concurrency and Scalability

6.1 Concurrency Model

The system leverages Erlang VM's lightweight process model for massive concurrency:

• **Lightweight Processes:** Each user actor is ~40-50KB, enabling 100K+ concurrent processes

• **Preemptive Scheduling:** BEAM VM ensures fair CPU time distribution

• **Asynchronous Message Passing:** Non-blocking communication between actors

• **Isolated State:** Each actor has independent state, no shared memory

• **Short-Lived Processes:** User actors terminate after completing operations

6.2 Scalability Architecture

Vertical Scalability: Single node tested with 100,000 concurrent users. Memory usage scales linearly at ~40-50KB per user process. Total memory: ~4GB for 100K users.

Horizontal Scalability: While current implementation is single-node, the actor model naturally supports distribution. Future enhancements could include engine sharding and distributed Erlang nodes for multi-server deployment.

6.3 Bottleneck Analysis

Component	Bottleneck	Impact	Mitigation
Engine Actor	Sequential message processing	Peak at 750 ops/sec	Sharding or multiple engines
Message Queue	Queue depth at high load	Increased latency	Batch processing
Subreddit Operations	Member list updates	O(N) complexity	Incremental counters
JSON Serialization	Large feed responses	Network overhead	Pagination

7. Performance Analysis

7.1 Scalability Test Results

Users	Subreddits	Posts	Comments	Duration	Ops/sec
1,000	5	1,732	828	44s	58.20
5,000	10	6,236	3,464	44s	234.00
10,000	10	12,117	7,229	45s	493.32
50,000	20	34,610	21,223	86s	746.76
100,000	30	61,230	38,013	189s	605.58

7.2 Key Performance Insights

Peak Throughput: 746.76 ops/sec achieved at 50K users, demonstrating optimal balance between concurrency and resource utilization.

Linear Scaling: System shows near-linear scaling from 1K to 50K users with 105% efficiency at 10K users.

Content Patterns: Comment-to-post ratio stabilizes at ~60% across all scales, validating Zipf distribution effectiveness.

Memory Efficiency: Consistent 40-50KB per user process allows 100K+ users within 4GB memory constraint.

Stability: Zero crashes across all test configurations, demonstrating system reliability and fault tolerance.

8. Multiple Clients Demonstration

8.1 Command-Line Client

A full-featured CLI client supporting all 18 endpoints with 15+ commands. Demonstrates individual API operations with clear request/response flow.

8.2 HTTP Client Simulator

Spawns hundreds of concurrent HTTP client processes, each simulating a real Reddit user performing registration, posting, commenting, voting, and messaging. Uses Zipf distribution for realistic behavior patterns. Successfully tested with 500+ concurrent clients.

8.3 Multi-Terminal Demo

Three automated scripts (demo_terminal2.sh, demo_terminal3.sh, demo_terminal4.sh) run from separate terminals to demonstrate concurrent client operations:

Terminal	Operations	Endpoints	API Calls
Client 1	User registration, subreddit management, search	8	11
Client 2	Posts, comments, replies, voting	5	14
Client 3	Direct messaging, karma, system stats	5	10
Total	**Complete Reddit functionality**	**18**	**35+**

Each demo script shows formatted output with command, request body, and JSON response for every operation, providing clear visibility into the REST API interaction.

9. Conclusion

9.1 Project Achievements

✓ **Complete Reddit Implementation:** All core features including user management, subreddits, posts, comments, voting, karma, feeds, and direct messaging

✓ **RESTful API:** 18 well-designed endpoints following REST principles with consistent JSON responses

✓ **Massive Scalability:** Successfully tested with 100,000 concurrent users, demonstrating enterprise-grade scalability

✓ **High Performance:** Peak throughput of 746.76 ops/sec with efficient resource utilization

✓ **Actor-Based Concurrency:** Leveraging Gleam's OTP for fault-tolerant, concurrent processing

✓ **Multiple Client Support:** Demonstrated with CLI client, HTTP simulator, and multi-terminal concurrent operations

✓ **Production Quality:** Zero compilation warnings, comprehensive error handling, graceful failure recovery

9.2 Technical Innovations

Zipf Distribution: Implemented realistic social network behavior using power law distribution for user activity and subreddit popularity.

Short-Lived Actors: Memory-efficient design where user actors terminate after operations, enabling massive concurrency.

Asynchronous Architecture: Non-blocking message passing throughout the system for optimal performance.

RESTful Best Practices: Resource-based URLs, proper HTTP methods, hierarchical structure, and consistent error handling.

9.3 Future Enhancements

• **Distributed Architecture:** Multi-node deployment with engine sharding

• **Persistent Storage:** Database backend for data persistence across restarts

• **WebSocket Support:** Real-time updates for online users

• **Caching Layer:** Redis/Memcached for frequently accessed data

• **Authentication:** JWT-based user authentication and authorization

• **Rate Limiting:** API throttling to prevent abuse

• **Media Upload:** Support for images and video content

• **Full-Text Search:** Elasticsearch integration for advanced search

9.4 Final Summary

This project successfully demonstrates a production-ready Reddit clone implementation using modern functional programming principles, actor-based concurrency, and RESTful API design. The system achieves high performance, massive scalability, and fault tolerance while maintaining clean architecture and code quality. All project requirements have been met and exceeded, with comprehensive testing demonstrating the system's capability to handle real-world social media workloads.