**📚 TVO Bot Architecture Document (v3.0)**

**1. Overview**

TVO Bot is designed with modularity, scalability, maintainability, observability, security, and resilience at its core. The architecture centers around clean separation of concerns with a service layer, centralized handlers, standardized interaction flow, and advanced operational best practices. This approach ensures rapid feature development while maintaining high quality, robustness, and readiness for growth.

**2. Core Principles**

**2.1 Single Responsibility**

Each module/file has a clearly defined role, avoiding overlap and making code easier to maintain, test, and extend.

**2.2 Centralized Command Handler**

* Dynamically loads all slash commands at startup.
* Command files export:
  + .data — a SlashCommandBuilder instance describing the command and options.
  + .execute() — async function handling the interaction logic.
* Prevents duplicate command registrations to avoid Discord API conflicts.
* Handles command-level errors gracefully with user-friendly feedback.

**2.3 Centralized Interaction Routers**

* A single interactionCreate event listener routes interactions to appropriate handlers:
  + Slash commands → command handler
  + Buttons → button handler
  + Modals → modal handler
  + Select menus → select menu handler
  + Autocomplete → autocomplete handler
* Each router dynamically loads handlers by custom ID or command name, enabling easy feature expansion.

**2.4 MongoDB Integration via Mongoose**

* Persistent data stored in MongoDB collections accessed via Mongoose schemas/models.
* Models represent core domain entities like Users, Events, Profiles, and Scheduled Events.
* Services abstract all DB interactions, ensuring a single source of truth and simplifying testing.

**2.5 Internationalization (i18n) and Localization**

* All user-facing text passes through a centralized translation utility (translator.js) supporting locale fallbacks.
* Translation keys are separated from code, facilitating maintenance and future language additions.
* Embed messages, button labels, error messages, and commands are fully localized.

**2.6 Centralized Embed Handlers / Creators**

* Embed message construction factored into reusable service or utility functions/templates.
* Ensures consistent visual style and messaging across dashboards, event posts, and alerts.
* Supports dynamic content insertion and localization.

**2.7 Dashboard Channel & Message Management**

* Dedicated dashboards per bot-managed channel representing commands or info summaries.
* Dashboards posted as embeds with action buttons; message IDs tracked for refreshing and cleanup.

**2.8 Scheduler & Maintenance Tasks**

* Modular scheduled jobs maintain bot/server health by:
  + Cleaning stale messages or expired dashboards.
  + Syncing database events with Discord events.
  + Sending reminders or alerts.
* Scheduler is configurable and extensible.

**2.9 Time Handling & Consistency**

* User time inputs interpreted as local time.
* All times stored internally in UTC for global consistency.
* When displaying times, convert UTC back to the user’s local timezone.
* Utilities (dateUtils.js) handle safe parsing, conversion, and formatting.
* Prevents bugs related to daylight savings and multi-region users.

**2.10 Centralized Configuration**

* Environment variables, constants, and IDs (roles, channels, languages) centralized in config files.
* Reduces duplication and supports multi-environment deployment.

**2.11 Unified Error & Success Handling**

* Services return standardized response objects: { success, message, followup?, error? }.
* Handlers use centralized reply helpers (replySuccess(), replyError()) to maintain consistent UX.
* Decouples business logic from Discord-specific reply code, facilitating testing.

**2.12 Comprehensive Logging & Debugging**

* All significant actions logged with contextual tags (e.g., command:onboarding).
* Logs include timestamps, user IDs, interaction types, and error stack traces for rapid root cause analysis.
* Log verbosity configurable based on environment.
* **Integrate observability tools** (e.g., Sentry, Datadog, Prometheus) for centralized metrics, error tracking, and performance monitoring.

**2.13 Modular & Scalable Handler Loading**

* Use dynamic module loaders (e.g., routerLoader.js) to auto-discover and register handlers.
* Facilitates adding new features without modifying core routing logic.
* Encourages clean separation and independent development.

**2.14 Full Localization Coverage**

* Embed fields, button labels, modal titles, error texts, and all UI elements localized.
* Translation keys version controlled and periodically reviewed.
* Supports future multi-language expansions with minimal code changes.

**2.15 Consistent State & Data Flow**

* Services maintain clear data flow: validate input → perform logic → return clean result object.
* Avoid side effects in services like direct Discord replies.
* Commands and handlers handle Discord interaction lifecycles using service responses.

**2.16 State Management for Complex Flows**

* For multi-step interactions (e.g., onboarding, multi-page forms), implement **state machines or flow managers** to track user progress.
* Ensures conversational consistency, prevents duplication or conflicts.
* Facilitates clear recovery points and branching logic.
* Enables cleaner, testable, and maintainable interaction flows.

**2.17 Feature Flags / Toggle System**

* Support dynamic enabling/disabling of features without redeployment.
* Facilitates gradual rollouts, A/B testing, and rapid disablement of problematic features.

**2.18 Rate Limiting & Throttling**

* Enforce per-user and per-guild rate limits to prevent abuse and spamming.
* Maintains bot responsiveness and compliance with Discord API limits.

**2.19 Role & Permission Management**

* Centralize permission checks in dedicated services or middleware.
* Support complex rules beyond simple role presence, including hierarchical and contextual permissions.

**2.20 Bot Metrics & Telemetry**

* Integrate telemetry systems to collect detailed metrics like:
  + Command usage frequency
  + User engagement patterns
  + Interaction latencies
  + Error rates
* Use these insights to prioritize features and optimize performance.
* Tools like **Prometheus**, **Datadog**, or **Sentry Performance Monitoring** are recommended.

**2.21 Data Migration & Backup Strategy**

* Automate database schema migrations with tools like migrate-mongo or custom scripts.
* Schedule regular backups and test restore procedures for production data safety.

**2.22 Caching Layer**

* Use in-memory or Redis caching for frequently accessed data such as configs or user profiles.
* Improves performance and reduces database load.

**2.23 Security Best Practices**

* Rigorously validate and sanitize all user inputs.
* Encrypt secrets and use environment variables securely.
* Regularly audit dependencies and update to patch vulnerabilities.

**2.24 Documentation Automation**

* Generate developer and API documentation automatically using tools like **JSDoc** or **TypeDoc**.
* Embed documentation in code comments to keep docs synchronized with implementation.
* Facilitates onboarding and maintenance with accurate, current documentation.

**2.25 Version Control & Code Quality Standards**

**Git Workflow**

* Use feature branches off main branches for new work.
* Write descriptive commit messages.
* Use Pull Requests for review before merging.
* Tag releases for versioning and rollback.
* Consider Git Flow or a simplified branching strategy.

**Code Linting & Formatting**

* Use ESLint with a chosen style guide (Airbnb, Standard, or custom).
* Use Prettier for automatic formatting.
* Enforce linting and formatting via pre-commit hooks (e.g., Husky).
* Run linting and tests on CI/CD pipelines.

**Continuous Integration / Deployment (Optional)**

* Automate testing, linting, and deployment with tools like GitHub Actions or CircleCI.
* Run tests on pull requests and merges.

**2.26 Graceful Shutdown & Restart**

* Implement lifecycle hooks to cleanly close DB connections, flush caches, and shut down gracefully.
* Enables zero-downtime deployments and safe restarts.
* Critical for production stability and data integrity.

**2.27 Backpressure Handling & Queueing**

* Use task queues (e.g., BullMQ, RabbitMQ) for heavy workloads or burst traffic.
* Supports rate limiting, retries, and smoothing traffic spikes.
* Useful for integrations or async-heavy features.

**2.28 Multi-Instance Scaling Support**

* Design for horizontal scaling by externalizing state and using distributed locks if needed.
* Supports high availability and load balancing.

**3. Folder Structure**

bash

Copy

src/

├── commands/ # Slash commands (export .data & .execute)

│ ├── onboarding.js

│ ├── profile.js

│ ├── dashboard.js

│ └── ...

│

├── services/ # Business logic & core functions

│ ├── onboardingService.js

│ ├── profileService.js

│ ├── dashboardService.js

│ ├── eventService.js

│ ├── eventEmbedService.js

│ ├── permissionService.js # Role and permission logic

│ ├── featureFlagService.js # Feature toggling

│ ├── rateLimitService.js # Rate limiting and throttling

│ ├── cacheService.js # Caching layer

│ ├── queueService.js # Task queue management

│ └── ...

│

├── handlers/ # Interaction handlers

│ ├── buttonHandler.js

│ ├── modalHandler.js

│ ├── selectMenuHandler.js

│ ├── autocompleteHandler.js

│ ├── reactionHandler.js # Handles message reaction add/remove events

│ ├── interactionCreate.js # Central interaction router

│ └── ...

│

├── models/ # Mongoose schemas/models

│ ├── UserProfile.js

│ ├── Event.js

│ ├── ScheduledEvent.js

│ └── ...

│

├── config/ # Constants and environment config

│ ├── roleIDs.js

│ ├── channels.js

│ ├── languageOptions.js

│ ├── featureFlags.js # Feature toggle config

│ └── ...

│

├── utils/ # Utilities (date formatting, logging, translation)

│ ├── dateUtils.js

│ ├── logger.js

│ ├── replyHelpers.js

│ ├── translator.js

│ ├── routerLoader.js

│ └── ...

│

├── schedulers/ # Scheduled job definitions

│ ├── cleanupScheduler.js

│ ├── eventSyncScheduler.js

│ ├── reminderScheduler.js

│ └── ...

│

├── events/ # Discord event listeners (e.g., interactionCreate)

│ └── interactionCreate.js

│

├── tests/ # Unit and integration tests

│ ├── commands/

│ ├── services/

│ ├── handlers/

│ └── ...

│

└── index.js # Bot entry point (client setup & login)

**4. Interaction Flow (Example)**

1. User issues slash command /onboarding.
2. Command file (commands/onboarding.js) invokes service (services/onboardingService.js).
3. Service processes logic and returns { success, message, followup?, error? }.
4. Command uses replyHelpers to send localized, consistent response.
5. If interaction involves buttons or modals, handlers route these to their respective service functions similarly.

**5. Development Standards**

* Use async/await throughout for asynchronous code.
* Catch and handle errors with informative logs and user-friendly messages.
* Never hardcode sensitive data; use environment variables.
* Localize all user-facing text.
* Keep functions focused, small, and testable.
* Write unit and integration tests covering critical flows.

**6. Future Refinements**

**6.1 Consider a Lightweight Service Mesh or API Gateway**

* When extending to microservices or integrating multiple external APIs, use a service mesh or API gateway to manage routing, retries, circuit breaking, and security policies.
* This abstraction enhances reliability and observability across distributed systems.

**6.2 Standardize on a Schema Validation Library**

* Adopt a consistent validation library (e.g., Joi, Zod) for all service inputs and outputs.
* Enforces data consistency, reduces bugs, and simplifies debugging.

**6.3 Define Clear SLAs & SLOs for Bot Behavior**

* Establish service-level agreements (SLAs) and objectives (SLOs) around key metrics such as response times, uptime, and error rates.
* SLAs provide clear targets and help focus monitoring and incident response.

**6.4 Document Interaction Contracts**

* Use specifications like OpenAPI or equivalent to formally document interaction inputs, outputs, and protocols.
* Facilitates integration with other teams, API consumers, or future extensions.

**6.5 Security Reviews & Penetration Testing**

* Schedule regular security audits and penetration tests, especially when handling sensitive data or elevated permissions.
* Proactively identify and remediate vulnerabilities to maintain trust and compliance.

**6.6 User Feedback & Telemetry Loops**

* Implement mechanisms to collect user feedback via in-bot surveys or feedback commands.
* Link user feedback with telemetry data to continuously validate and improve user experience and feature prioritization.

**6.7 Fallback & Recovery Strategies**

* Design fallback behaviors for critical components when external dependencies or databases fail.
* Examples include serving cached stale data, exponential retry queues, or graceful degradation of features.
* Improves bot resilience and user experience during partial outages.