# Purpose

This campaign manager is a purpose-built narrative operating system for tabletop role-playing games, designed to replace the need for DMs to force general-purpose note tools like Obsidian into workflows they were never designed for. Instead of relying on loosely structured markdown files and manually maintained templates, it provides typed campaign objects (such as NPCs and locations), dynamic views, and a fast inbox for capturing raw session notes without friction. DMs can freely write in markdown while selectively extracting people, places, and ideas into structured entities, enabling reliable queries, relationships, and future automation. The system prioritizes long-term campaign memory, non-destructive evolution, and DM control—supporting improvisation first, structure second—so preparation enhances play rather than becoming a maintenance burden.

## Core Principles

- Improvisation First, Structure Second.
    - Raw notes can be captured quickly without forced categorization
- Typed Objects, Not Templates.
    - Campaign elements are structured entities rendered through views. Not copied markdown templates.
- Non-Destructive Evolution.
    - Changing schemas, fields, or views updates all entities automatically without manual rewrites.
- Markdown Where It Matters.
    - Freeform writing remains markdown-based.
- DM Control and Trust.
    - The DM decides what becomes canon and/or visible.
- Long-Term Campaign Memory
    - The system should remain usable across multiple multi-year campaigns without decay.

# User-Facing Features

## Quick Notes

- Markdown. Allows for fast, low-friction capture of unstructured ideas or notes
- Open / Closed / Archived / Pinned states
- Optional Note Titles

## Quick Notes Inbox View

- Centralized inbox for all Quick Notes
- Designed for review, triage, and gradual organization rather than immediate structuring
- Notes can be filtered and sorted by state, creation time, or last activity
- Supports extracting people, places, and ideas directly from note content without altering the original note

## Quick Notes Extraction

- Highlight text within Quick Note and extract to Typed Object
- Non-destructive to original Quick Note
- Extracted entity links back to source note for context
- Extraction workflow:
    - User highlights text (e.g., "John the Blacksmith, a human fighter")
    - System suggests entity type (NPC)
    - System pre-populates fields from highlighted text
    - User reviews and completes additional fields
    - Entity created with backref to source note
- Batch extraction:
    - Extract multiple entities from single note
    - System detects wikilink patterns to suggest entities
    - Preview all extractions before confirming

## Typed Objects

- Campaign elements are structured entities with:
    - System-defined core fields - Required fields for consistency and querying
    - User-defined custom fields - DMs can add their own fields specific to their world
    - Markdown body - Long-form narrative content, notes, and improvisation

## Views, Not Templates

- Each Typed Object is rendered using configurable views.
- Views control layout and emphasis without modifying stored data
- Future support for multiple views per object (compact vs detailed)

## Markdown with First-Class Linking

- Wiki-style links between entities
- Mentions tracked separately from semantic relationships
- Notes remain readable outside the system

## Deterministic Queries

- Queries are code-driven and deterministic to ensure reliability and debuggability.
    - Show me locations related to “npc_name”
    - Name all NPCs linked to “faction_name”
    - List all currently active Plots
- Queries operate only on structured data and explicit relationships

## Generators

- Various Generators for things like:
    - Magic Shop Inventories
    - Tavern Generators
    - NPC Generators

- Allow for contextual World State Inputs (City, location, NPCs)
- AI utilized for flavor text

## Note Sharing

- Allow Players to view notes depending on visibility settings in individual Typed Objects or Notes
- Allow Players or Fellow DMs to add or edit notes
    - Settings to require validation from Campaign Owner, Any DM, or None.

- Exportable data - allow for saving locally.

## Timeline System

- Entities Valid from and to dates
    - Archived Entities listed separately

## Search

- Ability to do a full-text search across all notes and objects

## Import/Export

- Pluggable importer architecture - each source (Obsidian, World Anvil, etc.) gets its own import adapter
- Obsidian Migration
    - Phase 1: Content Import
        - Import all .md files to Quick Notes. Potentially Create an Import Folder in Inbox.
        - Preserve wikilinks as plain text initially
    - Phase 2: Assisted Extraction
        - UI to review imported notes
        - Suggest entities based on wikilink patterns
        - Batch Extraction with preview
        - DM confirms/tweaks field mappings
    - Phase 3: Template Mapping
        - If you detect common templates
        - Offer to map lines to core or user-defined fields
        - Show Preview
- World Anvil
    - Similar Phased Approach
    - Map their types: Article -> Lore, Character -> NPC, Organization -> Faction

## Mobile Access

- Access and Create Notes from multiple devices.

## Calendar System

- Users can define their own month names, days per month, etc.
- Calendar stored as configuration: JSON definition with month/day structure

# Data Model

## Typed Objects

- Core System Fields
    - id: GUID-based unique identifier
    - name: Display name
    - created_at: Timestamp of creation
    - updated_at: Timestamp of last modification
    - created_by_actor_id: Which user/DM created this
    - last_modified_by_actor_id: Which user/DM last edited this
    - body: Markdown content for long-form notes
- Campaign Scoping Fields
    - campaign-id : Primary campaign this entity belongs to
    - Is_global: Boolean - if true, visible to all campaigns in world
    - Linked_campaign_ids - List of specific campaigns with explicit access
- Temporal Fields
    - Valid_from and valid_to - nullable dates
- Visibility Fields
    - Visibility - Enum:
        - Private: Only campaign owner can see
        - DMOnly: All DMs in campaign can see
        - Public: Players can see based on campaign setting
- Custom Field Types. Support the following types:
    - Text
    - LongText
    - Number
    - Boolean
    - Date
    - InGameDate
    - EntityReference
    - EntityReferenceList
    - Currency
    - DiceFormula
    - Measurement

## Field Definitions

- Fields are defined per Typed Object not copied via Templates
- Adding or Changing a field automatically updates all existing entities
    - Optional request for a default value to be applied to older entities, if needed
- Entity Fields
    - campaign_id: The primary campaign it belongs to
    - is_global: Boolean - if true, visible to all campaigns
    - linked_campaign_ids: List of specific campaigns with access
    - Valid_from and valid_to - Nullable date

## Relationships vs Mentions

- Relationships - Structured, can be filtered by type, used for queries
- Mentions - Generated from markdown content of an entity.
    - Discovery tool "This entity is mentioned in…."

## Actor Roles & Permissions

- Three core actor roles:
    - World Owner: Creator the world, ultimate authority, can manage all campaigns
    - DM (Campaign-level): Runs specific campaign(s), full edit access within their campaign
    - Player: View-only by default, can be granted limited edit rights
- Players inherit view permissions from campaign visibility settings
- World Owner can override default permissions per campaign
- Every entity tracks created_by_actor_id and last_modified_by_actor_id
- Every data access requires both campaign_id and actor_id

## Relationship Types

- Relationships are typed and directional
- Core relationship types:
    - Location Relationships
        - ParentLocation (City->Nation, Business->City, Cave->Region, etc)
    - NPC Relationships
        - MemberOf (NPC -> Faction)
        - EmployedBy (NPC -> Location/NPC)
        - AlliedWith (NPC -> NPC/Faction)
        - EnemyOF (NPC -> NPC/Faction)
    - Plot Relationships
        - InvolvedIn (NPC -> Plot)
        - OccursIn (Plot -> Location)
    - Generic
        - Related (catch-all for undefined relationships)

# Technical Architecture

## Storage & Backup

- Storage abstraction layer: All backup operations go through an interface, not directly to filesystem or cloud provider
- Content-addressable media storage. Files named by their hash rather than original filename.
- Relative file paths only. Never store absolute paths
- Use GUIDs instead of auto-incrementing integers for all entity IDs - enables merging campaigns and cross-instance portability
- Backups are complete archives (database + media) that can be restored independently
- All backups encrypted at rest
- Each backup includes schema version number for future migrations

- Support for the Fresh Campaigns (no shared entities) and Linked Campaigns (Global Entities already linked)

## API Architecture

- All business logic lives in backend API, not in UI code
- Design auth to support both session cookies (web) and bearer tokens (mobile) from the start
- Each request contains all needed context (campaignId, userId) - no server-side session dependence
- API can return different detail levels (compact/full) based on client needs
- Store thumbnails, medium, and full-size versions of all images
- API versioning from day one (/api/v1/)
- Stateless design: Server stores no session state between requests
- Consistent error handling: Standard error response format across all endpoints

## Performance Principles

- Pagination. When searching results only load N number of results at a time. (1-50 of 857)
- Lazy Loading. Entities are not loaded until requested by the user.
- Cache frequently-read, rarely-changed data (view definitions, custom field schemas)
- Cache invalidation: When definitions change, clear related cache
- Database Query Performance
    - Database Indexing
        - Composite indexes for common query combinations
    - Always include campaign_id in WHERE clause
    - Add database query timeouts
- Performance Testing:
    - Create seed scripts for realistic data volumes

## Version Control / Audit Trail

- Every update logged automatically, no user action required
- Track which specific fields changed, not just "entity updated"
- Before/after values for each field modification
- Rolling window of detailed history
- Extended audit trail accessible.
- For recent changes, single action to restore previous value
- Every change links to actor who made it
- Multiple field changes in one save operation grouped together
- Audit logs older than 1 year moved to archive table for performance
- Markdown body edits will just show the body is modified. And store snapshots of before and after

## Offline First Strategy

- Desktop app maintains local database
- All writes go to local database immediately
- Background sync to server when online
- Offline operations queued and synced when connection restored
- Conflict resolution strategy:

    - Last-write-wins based on timestamp (initial implementation)
    - Manual conflict resolution UI for concurrent edits
    - Only applies when multiple users edit same entity

### Technologies

- C#/.NET 10
- EF ORM
- Postgres
- Docker
- Next.js
- TypeScript
- Tailwind CSS

## Slice Roadmap

1. Quick Notes
2. Quick Note Inbox
3. Typed Entity Foundation
4. Extraction
5. Custom Fields
6. Entity Type Managmenet
7. Relationships
8. Mentions/Wikilinks
9. Search
10. Media
11. Views
12. Import/Export
13. Campaign Management
14. API Hardening
15. Authentication & User Accounts
16. Cloud Backup
17. Actor Roles & Permissions
18. Visibility & Sharing
19. Audit Trail UI
20. Collaborative Editing
21. Timeline System
22. Calendar System
23. Generators
24. CQL
25. Offline-First Architecture
26. Mobile App

## Future Features

- Second Window view. Or Player View. A way to open a second window viewable by players through a window capture on OBS, Discord, or a second monitor if in person.

- Ability to modify keybinds
- Start with just Entities being the folders. Eventually allow custom folder creations. Can I make it so the folder itself can be a note as well
- Pinnable and Zoomable maps. Change image if zoomed in enough to like a city.

## Slice Features

Slice 1: Quick Note Features

- Create a Quick Note Object with a simple text body
- Add Optional Title (Fallback is first N Characters of Note Body)
- Add Note States (Open, Closed, Pinned, Archived)
- Make Notes able to be deleted. Soft Delete (Trash Bin implemented later). Confirmation Box.
- Add CreatedAt/UpdatedAt Timestamps
- Auto Save Notes. Debounce (1s). Throttle (30s)
- Implement Markdown for Body
- Have an option for an edit mode and Preview mode for Markdown. Live Preview will come later
- Create Polished Frontend UI (Title, Body, Flags, Delete, Autosave Message)
- Auto Save on click away or window close
- Shortcuts (Bold, Italic, Strikethrough, Save, Find)
- Create a toolbar and Implement Functions (Bold, Italic, Strikethrough, Bullet List, Numbered List, Link, Quote, Checkbox, Code inline, Code Block, Headings, and Horizontal Line, word count)
- Add button to create new note
- Create a firstNote. Something to show when it's all empty.
- Error Handling. "Offline. Saving Locally"... "Save Failed. Trying again."
- Loading State UI
- Readable Timestamp Display (e.g. 2 Hours ago, a few seconds ago, etc)
- Export note. Exports as a .md file.
- Dark Mode Implemented

Slice 2: Inbox and Trash Bin

- Inbox view to display all non-deleted Quick Notes
  - Each note shows the title, first N characters preview, and a visual state indicator icon.
- Click a note to open the full Quick Note view
- Add a button to create a new Quick Note
- Add Filter opens to filter by Note State
- Add Sort options for Creation Date and Last Updated Date
- Add Pagination to Inbox View
- Configurable page size (5, 10, 20, 50, 100)
- Add ability to Search by title or body content. List is updated to include matching content.
- Add zero results found view for Searches
- Quick Actions on Hover in Inbox
  - Change Quick Note State
  - Delete
- Bulk Selection and actions in Inbox
  - Checkbox appears on hover
  - Selecting one note reveals checkboxes for all notes

        - Actions
            - Update state
            - Delete
- Add arrow key navigation through inbox list
- Preserve scroll position when opening and returning from a note
- Empty state display when:
    - Inbox has no notes
    - Filter return no results
- Display all deleted Quick Notes
- Each note shows title and body preview
- Quick Actions on Hover in Trash Bin
    - Delete Forever
    - Restore
- Bulk Actions similar to inbox
    - Delete Forever
    - Restore
- Delete All Forever Button
- Empty state display for Trash Bin
- Back and Forward navigation buttons added to Quick Note View
- Collapsible and scrollable side pane showing Inbox List or Trash Bin (Tab Selection at top of pane)
- If deleted note. Add buttons for
    - Delete Forever
    - Restore

Slice 3: Typed Entity Foundation

# Slice Implementation RoadMap

## Slice 1: Quick Notes (Single Note Experience)

**Goal:** Build a polished, production-ready markdown note editor for campaign session notes. Focus on making a single note experience perfect.

### Phase 1.1: Core Data & API

**Goal:** Database foundation and basic CRUD endpoints
**Backend:**

1. Create QuickNote entity
    - Id (Guid)
    - Title (string, nullable)
    - Body (string)
    - State (enum: Open, Closed, Pinned, Archived)
    - CreatedAt (DateTime)
    - UpdatedAt (DateTime)
    - DeletedAt(DateTime)

2. Create DbContext and initial migration
3. Create API endpoints:
    - POST /api/quicknotes (create new note)
    - GET /api/quicknotes/{id} (get single note)
    - PUT /api/quicknotes/{id} (update note)
    - DELETE /api/quicknotes/{id} (soft delete: set IsDeleted = true)
4. Test ( .\scripts\smoke-quicknotes.ps1 -BaseUrl "http://localhost:5185" -PageSize 5 )

**Done when:** Can create, retrieve, update, and delete notes via API

## Phase 1.2: Basic Frontend Structure

**Goal:** Minimal working UI to view/edit a single note
**Frontend:**

1. Set up Next.js with TypeScript + Tailwind
2. Create API client helper (`lib/api.ts`)
3. Create route: `/quicknotes/[id]` (note detail/edit page)
4. Build basic note component:
    - Optional title input (placeholder: "Untitled")
    - Large textarea for body
    - Display created/updated timestamps (raw format for now)
5. Manual "Save" button (auto-save comes later)
6. Fetch note on page load
7. Update note on save

**Done when:** Can open a note by ID, edit it, and manually save changes

## Phase 1.3: Auto-Save System

**Goal:** Seamless automatic saving with user feedback
**Backend:**

1. Ensure PUT endpoint handles partial updates efficiently
2. Update UpdatedAt timestamp automatically on save

**Frontend:**

3. Remove manual "Save" button
4. Implement debounced auto-save:
    - Debounce: 1 second after user stops typing
    - Throttle: Maximum 30 seconds between saves (even while typing)
5. Auto-save on blur (click away from textarea/title)
6. Auto-save on beforeunload (window close/navigation)
7. Add save status indicator:
    - "Saving..." (during request)
    - "Saved at 2:34 PM" (success)
    - "Save failed. Retrying..." (error, with retry logic)
    - "Offline. Saving locally" (no connection - queue for later)
8. Warn on navigate away with unsaved changes:
    - If debounce timer is active, show confirmation dialog
    - "You have unsaved changes. Leave anyway?"
9. Delete Save Button

**Done when:** Users never manually save, always know their data is safe

## Phase 1.4: Note States & Metadata

**Goal:** Manage note lifecycle and display metadata
**Frontend:**

1. Add state selector dropdown
   - Options: Open, Closed, Pinned, Archived
   - Visual badge showing current state
   - Auto-save when state changes
2. Display readable timestamps:
   - "Created 2 hours ago"
   - "Updated a few seconds ago"
   - Use library like `date-fns` or implement custom formatter
3. Add word count display:
   - Live count as user types
   - Display: "245 words"

**Done when:** Can change note states, timestamps are human-readable, word count is accurate

## Phase 1.5: Delete with Confirmation

**Goal:** Safe deletion with confirmation dialog
**Frontend:**

1. Add "Delete" button
2. Implement confirmation dialog:
   - "Are you sure you want to delete this note?"
   - Show note title (or "Untitled") in confirmation
   - "Delete" (red) / "Cancel" (gray) buttons
3. Call DELETE endpoint on confirm
4. Redirect after successful delete
   - For now: redirect to most recent not deleted note or home
   - In Slice 2: redirect to inbox

**Backend:**

5. Soft Delete Implementation:
   - Set IsDeleted = true (don't actually delete row)
   - Filter out deleted notes from GET queries
   - Trash bin / recovery implemented in later slice

**Done when:** Can safely delete notes with confirmation, deleted notes are hidden

## Phase 1.6: Markdown Foundation

**Goal:** Render markdown and toggle between edit/preview modes
**Frontend:**

1. Install markdown library:
   - `npm install react-markdown remark-gfm`
   - remark-gfm adds GitHub-flavored markdown (tables, strikethrough, task lists)
2. Add Edit/Preview mode toggle buttons
   - Toggle button group: [Edit] [Preview]
   - Default to Edit mode
3. In Edit mode:
   - Show textarea with monospace font

        - Syntax remains visible
4. In Preview mode:
    - Render body with `<ReactMarkdown>`
    - Apply Tailwind typography plugin (`@tailwindcss/typography`)
    - Test rendering: headings, lists, bold, italic, links, code blocks, task lists
5. Preview updates automatically (not live during typing, but on mode switch)

**Done when:** Can toggle between raw markdown and beautiful rendered preview

## Phase 1.7: Markdown Toolbar

**Goal:** Visual formatting buttons for common markdown syntax
**Frontend:**

1. Create toolbar component (fixed position below title, above body)
2. Implement button actions (insert syntax at cursor position):
    - **Bold** → Wrap selection in `**text**` (or insert `****` with cursor in middle)
    - *Italic* → Wrap in `*text*`
    - Strikethrough → Wrap in `~~text~~`
    - Heading dropdown → Insert `#`, `##`, `###`, etc. at line start
    - Bullet list → Insert `- ` at line start
    - Numbered list → Insert `1. ` at line start
    - Quote → Insert `> ` at line start
    - Checkbox → Insert `- [ ] ` (task list item)
    - Link → Insert `[text](url)` with cursor positioned in URL
    - Code inline → Wrap in `` `code` ``
    - Code block → Wrap in triple backticks with optional language
    - Horizontal rule → Insert `---` on new line
3. Handle cursor/selection intelligently:
    - If text selected: wrap it
    - If no selection: insert syntax and position cursor appropriately
4. Show word count in toolbar:
    - Live update as user types
    - "245 words"

**Done when:** All toolbar buttons work reliably, cursor positioning feels natural

## Phase 1.8: Keyboard Shortcuts

**Goal:** Power user efficiency with keyboard shortcuts
**Frontend:**

1. Implement shortcuts:
    - Cmd/Ctrl + B → Bold
    - Cmd/Ctrl + I → Italic
    - Cmd/Ctrl + Shift + X → Strikethrough
    - Cmd/Ctrl + K → Insert link
    - Cmd/Ctrl + S → Force save immediately
    - Cmd/Ctrl + F → Find in note (browser default, ensure it works)
    - Cmd/Ctrl + / → Toggle Edit/Preview mode
2. Add keyboard shortcut hints to toolbar:

    - Hover tooltips show shortcuts: "Bold (⌘B)"
3. Prevent browser conflicts:
    - Cmd/Ctrl + S should trigger note save, not browser save
    - Use `event.preventDefault()` appropriately

**Done when:** All shortcuts work, tooltips show them, no browser conflicts

## Phase 1.9: Error Handling & Edge Cases

**Goal:** Graceful handling of all failure scenarios
**Frontend:**

1. Network error handling:
    - Detect offline state
    - Show: "Offline. Saving locally"
    - Queue failed saves, retry when connection restored
    - Test: disconnect wifi, edit note, reconnect
2. Server error handling:
    - Show: "Save failed. Retrying..."
    - Automatic retry with exponential backoff
    - After 3 failures: "Save failed. Retry?" with manual retry button
3. Note not found (404):
    - Show: "This note doesn't exist or was deleted"
    - Button to create new note or go home
4. Loading states:
    - Skeleton loader while fetching note
    - Don't show empty textarea until data loads
5. Empty state (new note):
    - When no note ID provided, show blank note
    - Or implement "First note" experience (see Phase 1.11)
6. Handle very long notes:
    - Test with 10,000+ word note
    - Ensure auto-save doesn't lag
    - Consider throttling more aggressively for huge notes

**Done when:** App handles all error scenarios gracefully, never loses data

## Phase 1.10: Export Note

**Goal:** Download note as markdown file
**Frontend:**

1. Add "Export" button in toolbar or menu
2. Generate `.md` file:
    - Filename: sanitized title or "untitled-note.md"
    - Content: raw markdown from body
    - Optional: prepend frontmatter with metadata:
3. Trigger browser download:
    - Use Blob API and `URL.createObjectURL()`
    - `<a download>` element
4. Show success feedback:
    - Toast notification: "Note exported"

**Done when:** Can download any note as a properly formatted .md file

## Phase 1.11: First Note Experience

**Goal:** Friendly onboarding when no notes exist
**Frontend:**

1. Create "First Note" component:
   - Friendly illustration or icon
   - Message: "Welcome to Archivist! Create your first Quick Note."
   - "Start Writing" message
2. Pre-populate first note with helpful content
3. Create note automatically when start Clicking
3. Redirect to note editor

**Done when:** First-time users have clear, friendly entry point

## Phase 1.12: UI Polish with shadcn/ui

**Goal:** Professional, cohesive visual design
**Frontend:**

1. ?

**Done when:** UI feels polished and professional, accessibility basics covered

## Phase 1.13: Dark Mode

**Goal:** Full dark theme support
**Frontend:**

1. Configure Tailwind for dark mode:
2. Add dark mode variants to all components:
   - `bg-white dark:bg-gray-900`
   - `text-gray-900 dark:text-gray-100`
   - `border-gray-200 dark:border-gray-700`
3. Create theme toggle button:
   - Icon: Sun/Moon
   - Position: Top-right navigation
4. Store preference in localStorage
5. Apply theme class to `<html>` element
6. Update shadcn components for dark mode:
   - Most support dark mode automatically
   - Test each component in both themes
7. Ensure markdown preview looks good in dark mode:
   - Code blocks, links, headings all readable
8. Test thoroughly in both modes

**Done when:** Dark mode looks as polished as light mode, preference persists

## Phase 1.14: Create New Note Flow

**Goal:** Easy way to create additional notes
**Frontend:**

1. Add "New Note" button in navigation/header
   - Icon: Plus sign or "New Note"
   - Prominent position, always accessible

2. Click behavior:
    - If current note has unsaved changes, confirm first
    - Create new blank note via API
    - Redirect to new note
3. Alternative: Keyboard shortcut
    - Cmd/Ctrl + N → Create new note
4. New note defaults:
    - Title: empty (shows "Untitled" placeholder)
    - Body: empty
    - State: Open
    - CreatedAt: now

**Done when:** Can easily create new notes without confusion

## Phase 1.15: Final Testing & Polish

**Goal:** Ensure everything works, refine UX
**Testing:**

1. Create 10+ test notes with varied content
2. Test all toolbar buttons thoroughly
3. Test all keyboard shortcuts
4. Test auto-save (stop typing, wait 1s, verify save)
5. Test throttle (type continuously for 30s, verify saves)
6. Test save on blur (click away, verify)
7. Test save on window close (close tab, reopen, verify)
8. Test offline mode (disconnect wifi, edit, reconnect, verify)
9. Test error recovery (kill backend, edit, restart backend, verify retry)
10. Test export (download .md, open in external editor, verify format)
11. Test very long notes (5000+ words)
12. Test special characters, emojis in title/body
13. Test all states (Open, Closed, Pinned, Archived)
14. Test delete with confirmation
15. Test both light and dark modes

**Polish**

16. Review all visual spacing, ensure consistency
17. Check all text for clarity
18. Ensure all buttons have hover states
19. Verify all dialogs are centered and look good
20. Check load times, ensure fast
21. Get feedback from a friend or fellow DM

**Documentation:**

22. Update README with: - Screenshots (light and dark mode) - Feature list - Keyboard shortcuts reference
23. Document known issues/limitations
24. Add notes for future self on technical decisions

**Done when:** QuickNotes feels production-ready, you're proud to use it yourself