Storage Canister Documentation

This document provides a detailed overview of the storage Motoko backend canister. This canister is designed to manage various aspects of data storage, versioning, real-time collaboration, and crucially, **automatic saving policies**.

Core Type Definitions

These types are fundamental across the different modules within the storage system.

FileId

- **Description**: A unique identifier for a file.
- **Type**: Nat32

Version

- **Description**: A sequential number representing a specific version of a file.
- **Type**: Nat

ClientId

- **Description**: A unique identifier for a client session in real-time collaboration.
- **Type**: Text

Seq

- **Description**: A sequence number for real-time events.
- **Type**: Nat

ChunkIndex

- **Description**: The index of a specific content chunk within a file.
- **Type**: Nat

ClientOpId

- **Description**: A unique identifier for an operation originating from a client, used for idempotency. Format: "clientId:clientSeq".
- **Type**: Text

Result<T, E>

- **Description**: A standard result type for operations that can either succeed (#ok) or fail (#err).
- **Type**: { #ok : T } | { #err : E }

Role

- **Description**: Defines user roles for access control on files.
- **Type**:
  - #Owner: Full control, can manage access.
  - #Editor: Can modify file content.
  - #Viewer: Can only read file content.

ContentChunk

- **Description**: Represents a segment of a file's binary data. Files are stored in chunks.
- **Type**: { index : ChunkIndex; data : Blob; size : Nat; }

FileMeta

- **Description**: Metadata associated with a file.
- **Type**: { id : FileId; tableId : Nat; var name : Text; var mime : Text; var size : Nat; var chunkCount : Nat; headVersion : Version; createdAt : Time.Time; var updatedAt : Time.Time; owner : Principal; var isDeleted : Bool; }

Access

- **Description**: Defines the access control rules for a file.
- **Type**: { owner : Principal; var sharedWith : [(Principal, Role)]; var isPublic : Bool; }

EditOp

- **Description**: A single text editing operation used in real-time collaboration.
- **Type**:
  - #Insert : { pos : Nat; text : Text }
  - #Delete : { pos : Nat; len : Nat }
  - #Replace : { pos : Nat; len : Nat; text : Text }

Patch

- **Description**: A collection of EditOps applied from a specific base version.
- **Type**: { base : Version; ops : [EditOp]; clientId : ClientId; clientOpId : ClientOpId; timestamp : Time.Time; }

Commit

- **Description**: A record of a version change, including the patch applied and authorship.
- **Type**: { version : Version; parent : Version; patch : Patch; author : Principal; message : ?Text; time : Time.Time; size : Nat; chunkCount : Nat; }

Cursor

- **Description**: Represents a user's cursor position and selection in a collaborative editing session.
- **Type**: { clientId : ClientId; pos : Nat; selection : ?{ from : Nat; to : Nat }; color : Text; lastSeen : Time.Time; }

PresenceEvent

- **Description**: Events indicating a client's presence status in a collaboration session.
- **Type**:
    - #Join : { clientId : ClientId; user : Principal }
    - #Leave : { clientId : ClientId }
    - #Heartbeat : { clientId : ClientId }

Event

- **Description**: A real-time update event for a file, covering various activities.
- **Type**: { seq : Seq; fileId : FileId; kind : { #PatchApplied : Commit; #CursorUpdate : Cursor; #Presence : PresenceEvent; #Snapshot : Version; #FileDeleted : { by : Principal }; #FileRestored : { by : Principal }; }; time : Time.Time; }

Subscription

- **Description**: Details of a client's subscription to real-time events for a file.
- **Type**: { clientId : ClientId; since : Seq; lastPolled : Time.Time; user : Principal; }

AutosavePolicy

- **Description**: Configuration for automatic saving for a file.
- **Type**: { intervalNanos : Nat; idleNanos : Nat; enabled : Bool; maxVersions : Nat; }

Paginated<T>

- **Description**: A generic structure for returning paginated lists of items.
- **Type**: { items : [T]; next : ?Nat; total : Nat; }

Error

- **Description**: Enumerated error types returned by canister functions.
- **Type**:
    - #NotFound
    - #AccessDenied
    - #InvalidOperation
    - #Conflict

   - #DuplicateOperation
   - #FileTooLarge
   - #InvalidChunk
   - #QuotaExceeded
   - #InternalError

FileMetaView

- **Description**: An immutable view of FileMeta for safe data transfer.
- **Type**: { id : FileId; tableId : Nat; name : Text; mime : Text; size : Nat; chunkCount : Nat; headVersion : Version; createdAt : Time.Time; updatedAt : Time.Time; owner : Principal; isDeleted : Bool; }

AccessView

- **Description**: An immutable view of Access for safe data transfer.
- **Type**: { owner : Principal; sharedWith : [(Principal, Role)]; isPublic : Bool; }

Public Functions (Autosave Management)

setAutosavePolicy(fileId : FileId, policy : AutosavePolicy)

- **Description**: Sets a custom autosave policy for a specific file. This allows per-file configuration of autosave behavior.
- **Arguments**:
   - fileId: FileId - The ID of the file to configure.
   - policy: AutosavePolicy - The desired autosave policy, including interval, idle time, enabled status, and max versions.
- **Return Value**: async Result<(), Error>
   - ok(()): Returns unit on success.
   - err(#NotFound): If the file is not found (though the current implementation might implicitly create a default if none exists, this is the expected error if file metadata isn't set up yet).
- **Dependent/Related Functions**: None directly, interacts with internal autosave state.

getAutosavePolicy(fileId : FileId)

- **Description**: Retrieves the currently active autosave policy for a given file.
- **Arguments**:
   - fileId: FileId - The ID of the file whose policy is to be retrieved.
- **Return Value**: async Result<AutosavePolicy, Error>

    - ok(AutosavePolicy): Returns the AutosavePolicy for the file.
    - err(#NotFound): If no autosave policy is found for the specified fileId.
- **Dependent/Related Functions**: None directly.

recordActivity(fileId : FileId)

- **Description**: Notifies the canister that activity has occurred on a specific file. This updates the lastActivity timestamp and sets pendingChanges to true, potentially triggering an autosave check. If no policy exists for the file, a default one is applied.
- **Arguments**:
    - fileId: FileId - The ID of the file on which activity occurred.
- **Return Value**: async Result<(), Error>
    - ok(()): Returns unit on success.
    - err(#NotFound): If the file is genuinely not recognized (after attempting to set a default policy).
- **Dependent/Related Functions**: setAutosavePolicy (if no existing policy is found).

hasPendingChanges(fileId : FileId)

- **Description**: Checks whether a specific file currently has unsaved changes according to its autosave state.
- **Arguments**:
    - fileId: FileId - The ID of the file to check.
- **Return Value**: async Result<Bool, Error>
    - ok(Bool): Returns true if there are pending changes, false otherwise.
    - err(#NotFound): If no autosave policy is found for the specified fileId.
- **Dependent/Related Functions**: None directly.

getFilesNeedingAutosave()

- **Description**: Returns a list of FileIds for files that currently meet their autosave criteria (enabled policy, pending changes, sufficient idle time, and interval passed). This function is typically called by an internal timer or a management agent to trigger actual autosaves.
- **Arguments**: None.
- **Return Value**: async [FileId] - An array of FileIds.
- **Dependent/Related Functions**: None directly.

cleanupOldPolicies(maxAgeNanos : Nat)

- **Description**: Removes autosave policies for files that haven't had any activity for a specified duration (maxAgeNanos). This helps in garbage collection of inactive file states.
- **Arguments**:

    - maxAgeNanos: Nat - The maximum age (in nanoseconds) of inactivity for a policy to be retained.
- **Return Value**: async Result<Nat, Error>
    - ok(Nat): Returns the count of policies that were removed.
    - err(Error): Could return an internal error if something goes wrong during cleanup.
- **Dependent/Related Functions**: None directly.

setGlobalAutosaveEnabled(enabled : Bool)

- **Description**: Enables or disables the autosave feature globally across all files managed by this canister.
- **Arguments**:
    - enabled: Bool - true to enable, false to disable.
- **Return Value**: () - Returns unit.
- **Dependent/Related Functions**: None directly.

getGlobalAutosaveEnabled()

- **Description**: Retrieves the current global autosave enablement status.
- **Arguments**: None.
- **Return Value**: async Bool - true if globally enabled, false otherwise.
- **Dependent/Related Functions**: None directly.

setGlobalAutosaveInterval(intervalNanos : Nat)

- **Description**: Sets the default autosave interval (in nanoseconds) that applies to files unless a specific policy overrides it.
- **Arguments**:
    - intervalNanos: Nat - The new default interval in nanoseconds.
- **Return Value**: () - Returns unit.
- **Dependent/Related Functions**: None directly.

getGlobalAutosaveInterval()

- **Description**: Retrieves the current global default autosave interval.
- **Arguments**: None.
- **Return Value**: async Nat - The interval in nanoseconds.
- **Dependent/Related Functions**: None directly.

setMaxConcurrentAutosaves(max : Nat)

- **Description**: Configures the maximum number of autosave tasks that can be processed concurrently.

- **Arguments**:
    - max: Nat - The maximum number of concurrent autosaves.
- **Return Value**: () - Returns unit.
- **Dependent/Related Functions**: None directly.

getMaxConcurrentAutosaves()

- **Description**: Retrieves the configured maximum number of concurrent autosave tasks.
- **Arguments**: None.
- **Return Value**: async Nat - The maximum concurrent autosaves.
- **Dependent/Related Functions**: None directly.

getAutosaveStats(fileId : FileId)

- **Description**: Provides statistics about the autosave status of a particular file, including how many times it has been autosaved, its last autosave time, and if it has pending changes.
- **Arguments**:
    - fileId: FileId - The ID of the file to get statistics for.
- **Return Value**: async Result<{ autosaveCount : Nat; lastAutosave : Time.Time; pendingChanges : Bool }, Error>
    - ok(...): Returns an object with the autosave count, last autosave timestamp, and pending changes status.
    - err(#NotFound): If no autosave policy (and thus no stats) are found for the specified fileId.
- **Dependent/Related Functions**: None directly.

getGlobalAutosaveStats()

- **Description**: Provides aggregated statistics about the overall autosave system, including the total number of files with policies, files with pending autosaves, and files with enabled policies.
- **Arguments**: None.
- **Return Value**: async { totalFiles : Nat; pendingAutosaves : Nat; enabledFiles : Nat } - Returns an object with global autosave statistics.
- **Dependent/Related Functions**: None directly.

isAutosaveDue(fileId : FileId)

- **Description**: Checks if a specific file is currently "due" for an autosave based on its policy, pending changes, and activity/interval timings.
- **Arguments**:
    - fileId: FileId - The ID of the file to check.
- **Return Value**: async Result<Bool, Error>
    - ok(Bool): Returns true if autosave is due, false otherwise.

    - err(#NotFound): If no autosave policy is found for the specified fileId.
- **Dependent/Related Functions**: None directly.

getTimeUntilAutosave(fileId : FileId)

- **Description**: Calculates the remaining time (in nanoseconds) until the next autosave is scheduled for a given file. Returns 0 if autosave is not enabled, has no pending changes, or is already due.
- **Arguments**:
    - fileId: FileId - The ID of the file.
- **Return Value**: async Result<Int, Error>
    - ok(Int): Returns the remaining time in nanoseconds.
    - err(#NotFound): If no autosave policy is found for the specified fileId.
- **Dependent/Related Functions**: None directly.

removeAutosavePolicy(fileId : FileId)

- **Description**: Removes the autosave policy associated with a specific file. This effectively disables autosave for that file and removes it from the autosave management system.
- **Arguments**:
    - fileId: FileId - The ID of the file whose policy is to be removed.
- **Return Value**: async Result<(), Error>
    - ok(()): Returns unit on success.
    - err(Error): Returns an error if the policy wasn't found or an internal issue occurs.
- **Dependent/Related Functions**: None directly.

getAllAutosaveFiles()

- **Description**: Retrieves a list of all FileIds for which an autosave policy is currently being tracked by the canister.
- **Arguments**: None.
- **Return Value**: async [FileId] - An array of FileIds.
- **Dependent/Related Functions**: None directly.

File Management Functions

This section details the functions responsible for core file operations: creation, reading, updating, and deletion.

Internal State Variables

These are internal variables used to manage the file system's state:

- filesById: A HashMap storing FileStorage objects, indexed by FileId.
- filesByTableId: A HashMap mapping TableId to an array of FileIds, providing quick lookup of files within a specific table.
- fileNamesByOwner: A nested HashMap that maps a Principal (user owner) to another HashMap which stores FileName to FileId, ensuring unique file names per owner.

Internal Helper Functions

The following functions are used internally by the canister's public methods to perform file operations. Frontend developers will typically interact with higher-level public functions that orchestrate calls to these.

createFile(name : Text, tableId: Nat, mime : Text, owner : Principal, initialContent : ?Blob)

- **Description**: Creates a new file entry in the storage system. This function handles validation of the file name, checks for existing files with the same name for the given owner, generates unique file and version IDs, and chunks the initial content if provided. It then populates the FileMeta, Access, and FileStorage structures and updates the internal maps (filesById, filesByTableId, fileNamesByOwner).
- **Arguments**:
  - name: Text - The name of the file. Must not be empty and must be unique for the owner.
  - tableId: Nat - The ID of the table this file belongs to.
  - mime: Text - The MIME type of the file (e.g., "text/plain", "application/pdf").
  - owner: Principal - The principal ID of the file's owner.
  - initialContent: ?Blob - Optional initial binary content of the file.
- **Return Value**: Result<FileId, Error>
  - ok(FileId): Returns the FileId of the newly created file.
  - err(#InvalidOperation): If the name is empty or already exists for the owner.
  - err(#FileTooLarge): If initialContent exceeds Types.MAX_FILE_SIZE.
  - err(#NotFound): If the provided tableId does not exist in filesByTableId (indicating an issue with table registration or pre-creation).
- **Dependent/Related Functions**: Types.MAX_FILE_SIZE, Types.MAX_CHUNK_SIZE, Types.now().

getFileMeta(fileId : FileId)

- **Description**: Retrieves the metadata (FileMetaView) for a specified file, provided it's not soft-deleted.

- **Arguments**:
  - fileId: FileId - The ID of the file.
- **Return Value**: Result<FileMetaView, Error>
  - ok(FileMetaView): Returns an immutable view of the file's metadata.
  - err(#NotFound): If the file does not exist or is soft-deleted.
- **Dependent/Related Functions**: toFileMetaView.

replaceFileContent(fileId : FileId, newChunks : [ContentChunk], user : Principal)

- **Description**: Replaces the entire content of a file with a new set of ContentChunks. A new version of the file is created. This operation requires editor permissions.
- **Arguments**:
  - fileId: FileId - The ID of the file to modify.
  - newChunks: [ContentChunk] - An array of new content chunks.
  - user: Principal - The principal ID of the user performing the operation.
- **Return Value**: Result<Version, Error>
  - ok(Version): Returns the Version of the new file state.
  - err(#NotFound): If the file does not exist or is soft-deleted.
  - err(#AccessDenied): If the user does not have Editor role or higher.
  - err(#InvalidChunk): If any provided chunk is invalid (e.g., exceeds MAX_CHUNK_SIZE).
  - err(#FileTooLarge): If the total size of newChunks exceeds Types.MAX_FILE_SIZE.
- **Dependent/Related Functions**: Types.canEdit, Types.validateChunk, Types.MAX_FILE_SIZE, Types.now().

updateChunk(fileId : FileId, chunkIndex : Nat, data : Blob, user : Principal)

- **Description**: Updates a specific content chunk of a file. This is useful for incremental updates. A new version of the file is created. This operation requires editor permissions.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - chunkIndex: Nat - The index of the chunk to update.
  - data: Blob - The new binary data for the chunk.
  - user: Principal - The principal ID of the user performing the operation.
- **Return Value**: Result<Version, Error>
  - ok(Version): Returns the Version of the new file state.
  - err(#NotFound): If the file does not exist or is soft-deleted.
  - err(#AccessDenied): If the user does not have Editor role or higher.
  - err(#InvalidChunk): If the provided data exceeds Types.MAX_CHUNK_SIZE.

    - err(#FileTooLarge): If updating this chunk would make the total file size exceed Types.MAX_FILE_SIZE.
- **Dependent/Related Functions**: Types.canEdit, Types.MAX_CHUNK_SIZE, Types.MAX_FILE_SIZE, Types.now().

deleteFile(fileId : FileId, user : Principal)

- **Description**: Performs a soft-delete on a file, marking it as deleted without permanently removing its data. Only the file owner can perform this. The file's name is removed from the owner's index to allow for new files with the same name.
- **Arguments**:
    - fileId: FileId - The ID of the file to soft-delete.
    - user: Principal - The principal ID of the user performing the operation.
- **Return Value**: Result<(), Error>
    - ok(()): Returns unit on success.
    - err(#NotFound): If the file does not exist or is already soft-deleted.
    - err(#AccessDenied): If the user is not the file owner.
- **Dependent/Related Functions**: Types.isOwner, Types.now().

restoreFile(fileId : FileId, user : Principal)

- **Description**: Restores a previously soft-deleted file, making it accessible again. Only the file owner can perform this. The file's name is added back to the owner's index.
- **Arguments**:
    - fileId: FileId - The ID of the file to restore.
    - user: Principal - The principal ID of the user performing the operation.
- **Return Value**: Result<(), Error>
    - ok(()): Returns unit on success.
    - err(#NotFound): If the file does not exist.
    - err(#InvalidOperation): If the file is not currently soft-deleted.
    - err(#AccessDenied): If the user is not the file owner.
- **Dependent/Related Functions**: Types.isOwner, Types.now().

Publicly Exposed Functions for File Management

These are the functions frontend developers will directly call for file operations:

getFileAccess(fileId : FileId)

- **Description**: Retrieves the access control information (AccessView) for a specified file, provided it's not soft-deleted. This is a public query function.

- **Arguments**:
    - fileId: FileId - The ID of the file.
- **Return Value**: async Result<AccessView, Error>
    - ok(AccessView): Returns an immutable view of the file's access control.
    - err(#NotFound): If the file does not exist or is soft-deleted.
- **Dependent/Related Functions**: toAccessView.

listFiles(tableId: Nat)

- **Description**: Lists the FileIds of all files associated with a given tableId. This function ensures that the calling user is a collaborator of the specified table before returning the list.
- **Arguments**:
    - tableId: Nat - The ID of the table whose files are to be listed.
- **Return Value**: async Result<[FileId], Error>
    - ok([FileId]): Returns an array of FileIds belonging to the table.
    - err(#NotFound): If the table does not exist or no files are associated with it.
    - err(#AccessDenied): If the caller is not a collaborator of the table.
- **Dependent/Related Functions**: TableManagement.get_table_collaborators, arrayContains.

getChunk(fileId : FileId, chunkIndex : Nat)

- **Description**: Retrieves a specific ContentChunk of a file. Access is restricted: the caller must have at least Viewer permissions or the file must be public.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - chunkIndex: Nat - The index of the chunk to retrieve.
- **Return Value**: async Result<ContentChunk, Error>
    - ok(ContentChunk): Returns the requested content chunk.
    - err(#NotFound): If the file or chunk does not exist, or the file is soft-deleted.
    - err(#AccessDenied): If the caller does not have sufficient access to the file.
- **Dependent/Related Functions**: hasAccess (internal helper to check permissions).

getAllChunks(fileId : FileId)

- **Description**: Retrieves all ContentChunks for a specific file, ordered by their index. Access is restricted.
- **Arguments**:
    - fileId: FileId - The ID of the file.
- **Return Value**: async Result<[ContentChunk], Error>
    - ok([ContentChunk]): Returns an array of all content chunks, sorted.
    - err(#NotFound): If the file does not exist or is soft-deleted.

    - err(#AccessDenied): If the caller does not have sufficient access.
- **Dependent/Related Functions**: hasAccess (internal helper), Array.sort.

getFileContent(fileId : FileId)

- **Description**: Reconstructs the complete binary content of a file from its chunks and returns it as a Blob. Access is restricted.
- **Arguments**:
    - fileId: FileId - The ID of the file.
- **Return Value**: async Result<Blob, Error>
    - ok(Blob): Returns the complete file content.
    - err(#NotFound): If the file does not exist or is soft-deleted.
    - err(#AccessDenied): If the caller does not have sufficient access.
- **Dependent/Related Functions**: hasAccess (internal helper), getAllChunks, Types.calculateTotalSize.

updateFileMeta(fileId : FileId, name : ?Text, mime : ?Text)

- **Description**: Allows an authorized user to update the name or MIME type of a file. Requires Editor role or higher. Handles name conflict checks to ensure uniqueness per owner.
- **Arguments**:
    - fileId: FileId - The ID of the file to update.
    - name: ?Text - An optional new name for the file. If provided, must not be empty and must be unique for the owner.
    - mime: ?Text - An optional new MIME type for the file.
- **Return Value**: async Result<(), Error>
    - ok(()): Returns unit on successful update.
    - err(#NotFound): If the file does not exist or is soft-deleted.
    - err(#AccessDenied): If the caller does not have Editor role or higher.
    - err(#InvalidOperation): If the new name is empty or conflicts with an existing file name for the owner.
- **Dependent/Related Functions**: hasAccess (internal helper), Types.canEdit, Types.now().

—---

Access Control & Utility Functions

This section details functions related to managing file access permissions, and providing general utility statistics.

hasAccess(fileId : FileId, user : Principal)

- **Description**: Checks if a given user has access to a specific file. This is a crucial security utility that verifies if the file exists, is not deleted, and if the user is a registered collaborator of the associated table.
- **Arguments**:
  - fileId: FileId - The ID of the file to check.
  - user: Principal - The principal ID of the user whose access is being verified.
- **Return Value**: async Result<Bool, Error>
  - ok(true): If the user has access.
  - ok(false): If the user does not have access but the file exists.
  - err(#NotFound): If the file does not exist or is soft-deleted, or if the associated table is not found.
- **Dependent/Related Functions**: TableManagement.get_table_collaborators, arrayContains.

getUserFileCount(user : Principal)

- **Description**: Counts the number of non-deleted files that a given user has Viewer access (or higher) to.
- **Arguments**:
  - user: Principal - The principal ID of the user.
- **Return Value**: async Nat - The count of accessible files.
- **Dependent/Related Functions**: Types.canView.

getUserStorageUsed(user : Principal)

- **Description**: Calculates the total storage space (in bytes) consumed by files where the given user is the Owner. This helps in managing user quotas.
- **Arguments**:
  - user: Principal - The principal ID of the user.
- **Return Value**: async Nat - The total storage used by the user's owned files.
- **Dependent/Related Functions**: Types.isOwner.

cleanupDeletedFiles()

- **Description**: Permanently removes all files that have been marked as isDeleted: true. This function is typically called by a maintenance job to free up storage space.
- **Arguments**: None.
- **Return Value**: Nat - The count of files that were permanently deleted.
- **Dependent/Related Functions**: None directly, interacts with internal file storage.

Event Management

These functions manage the real-time event stream for collaborative editing, providing a history of changes and updates.

Internal State Variables

- eventsByFile: A HashMap storing EventRingBuffers, indexed by FileId. Each buffer holds a chronological sequence of events for a specific file.
- nextSeq: A global counter for event sequence numbers, ensuring unique event ordering.

Internal Helper Functions

createEventBuffer(fileId : FileId)

- **Description**: Initializes a new EventRingBuffer for a given file. This buffer has a fixed maxSize (defined by Types.MAX_EVENTS_RETENTION) to prevent unbounded growth.
- **Arguments**:
  - fileId: FileId - The ID of the file.
- **Return Value**: EventRingBuffer - The newly created event buffer.
- **Dependent/Related Functions**: Types.MAX_EVENTS_RETENTION, nextSeq.

addEvent(fileId : FileId, event : Event)

- **Description**: Adds a new Event to the event RingBuffer for the specified file. It automatically manages the buffer size, removing older events if the maxSize is exceeded.
- **Arguments**:
  - fileId: FileId - The ID of the file the event pertains to.
  - event: Event - The event to add.
- **Return Value**: () - Returns unit.
- **Dependent/Related Functions**: createEventBuffer.

Publicly Exposed Functions for Event Management

getEvents(fileId : FileId, since : Seq, maxEvents : Nat)

- **Description**: Retrieves a batch of events for a specific file, starting from a given since sequence number. This allows clients to efficiently poll for new events without fetching the entire history. Requires user access to the file.
- **Arguments**:
  - fileId: FileId - The ID of the file to get events for.

    - since: Seq - The sequence number from which to retrieve events (exclusive).
    - maxEvents: Nat - The maximum number of events to return.
- **Return Value**: async Result<{ events : [Event]; nextSince : Seq }, Error>
    - ok(...): Returns an object containing an array of Events and the nextSince value for subsequent polls.
    - err(#NotFound): If the file or its event buffer does not exist.
    - err(#AccessDenied): If the caller does not have access to the file.
- **Dependent/Related Functions**: hasAccess.

Presence Management

These functions handle real-time user presence and cursor information within collaborative documents.

Internal State Variables

- presenceByFile: A nested HashMap mapping FileIds to another HashMap of ClientIds to ClientPresence objects. Stores the live presence data for each client on each file.
- subscriptionsByFile: A nested HashMap mapping FileIds to ClientIds to Subscription objects. (Though Subscription functions are not fully detailed in the provided code, this variable indicates the intent for subscription management).

Internal Helper Functions

joinFile(fileId : FileId, clientId : ClientId, user : Principal)

- **Description**: Registers a client's presence on a file. This creates a ClientPresence entry for the client, records their user and lastSeen timestamp, and generates a #Join PresenceEvent.
- **Arguments**:
    - fileId: FileId - The ID of the file the client is joining.
    - clientId: ClientId - The unique ID of the client session.
    - user: Principal - The principal ID of the user associated with the client.
- **Return Value**: Result<Seq, Error> - Returns the Seq of the generated Join event on success.
- **Dependent/Related Functions**: Types.now(), addEvent, nextSeq.

updatePresence(fileId : FileId, clientId : ClientId, cursor : ?Cursor)

- **Description**: Updates a client's presence and optionally their cursor information on a file. It refreshes the lastSeen timestamp and, if a cursor is provided, generates a #CursorUpdate event.

- **Arguments**:
    - fileId: FileId - The ID of the file.
    - clientId: ClientId - The unique ID of the client session.
    - cursor: ?Cursor - An optional Cursor object representing the client's current cursor position/selection.
- **Return Value**: Result<(), Error>
    - ok(()): Returns unit on success.
    - err(#NotFound): If the file or client presence entry does not exist.
- **Dependent/Related Functions**: Types.now(), addEvent, nextSeq.

cleanupStaleClients(fileId : FileId)

- **Description**: Iterates through active client presences for a file and removes those that haven't sent a heartbeat (lastSeen) within Types.CURSOR_TIMEOUT_NANOS. It also generates #Leave events for removed clients.
- **Arguments**:
    - fileId: FileId - The ID of the file to clean up clients for.
- **Return Value**: Result<Nat, Error>
    - ok(Nat): Returns the number of stale clients removed.
- **Dependent/Related Functions**: Types.now(), Types.CURSOR_TIMEOUT_NANOS, addEvent, nextSeq.

Publicly Exposed Functions for Presence Management

leaveFile(fileId : FileId, clientId : ClientId)

- **Description**: Explicitly removes a client's presence from a file and generates a #Leave PresenceEvent.
- **Arguments**:
    - fileId: FileId - The ID of the file the client is leaving.
    - clientId: ClientId - The unique ID of the client session.
- **Return Value**: async Result<(), Error>
    - ok(()): Returns unit on success.
- **Dependent/Related Functions**: addEvent, nextSeq.

getActiveClients(fileId : FileId)

- **Description**: Retrieves a list of all currently active ClientPresence records for a specified file, filtering out stale clients based on Types.CURSOR_TIMEOUT_NANOS.

- **Arguments**:
  - fileId: FileId - The ID of the file.
- **Return Value**: async Result<[ClientPresence], Error>
  - ok([ClientPresence]): Returns an array of active ClientPresence objects. Returns an empty array if no clients are active or the file isn't found in presence map.
- **Dependent/Related Functions**: Types.now(), Types.CURSOR_TIMEOUT_NANOS.

isClientActive(fileId : FileId, clientId : ClientId)

- **Description**: Checks if a specific client is currently considered active on a file, based on its lastSeen timestamp and Types.CURSOR_TIMEOUT_NANOS. Requires user access to the file.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - clientId: ClientId - The ID of the client.
- **Return Value**: async Bool - true if the client is active, false otherwise (including if the file doesn't exist or access is denied).
- **Dependent/Related Functions**: hasAccess, Types.now(), Types.CURSOR_TIMEOUT_NANOS.

getClientCursor(fileId : FileId, clientId : ClientId)

- **Description**: Retrieves the last known Cursor information for a specific client on a file. Requires user access to the file.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - clientId: ClientId - The ID of the client.
- **Return Value**: async Result<Cursor, Error>
  - ok(Cursor): Returns the Cursor object.
  - err(#NotFound): If the file, client presence, or cursor information is not found.
  - err(#AccessDenied): If the caller does not have access to the file.
- **Dependent/Related Functions**: hasAccess.

getAllCursors(fileId : FileId)

- **Description**: Retrieves a list of all active Cursors for a specific file. Useful for rendering multiple collaborators' cursors in a real-time editor.
- **Arguments**:
  - fileId: FileId - The ID of the file.
- **Return Value**: async Result<[Cursor], Error>
  - ok([Cursor]): Returns an array of active Cursor objects. Returns an empty array if no cursors are found or no clients are active.
- **Dependent/Related Functions**: Types.now(), Types.CURSOR_TIMEOUT_NANOS.

—---

Versioning & Patch Processing

These functions are responsible for maintaining file version history, applying real-time patches, and creating snapshots.

Internal State Variables

- commitsByFile: A nested HashMap storing VersionStorage objects, indexed by FileId and then by Version. Each VersionStorage holds a Commit record and a snapshot of the file's chunks at that version.
- headsByFile: A HashMap mapping FileIds to their current headVersion.
- dedupeOpIdsByFile: A nested HashMap used for deduplicating client operations (ClientOpId), mapping FileIds to a map of ClientOpIds to timestamps.

Internal Helper Functions

isDuplicateOperation(fileId : FileId, clientOpId : Text)

- **Description**: Checks if a clientOpId for a specific file has already been processed, preventing duplicate application of patches.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - clientOpId: Text - The client operation ID to check.
- **Return Value**: Bool - true if the operation is a duplicate, false otherwise.
- **Dependent/Related Functions**: None directly.

recordOperation(fileId : FileId, clientOpId : Text)

- **Description**: Records a clientOpId as processed for a specific file, along with its timestamp, to facilitate deduplication.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - clientOpId: Text - The client operation ID to record.
- **Return Value**: () - Returns unit.
- **Dependent/Related Functions**: Types.now().

createInitialVersion(fileId : FileId, chunks : [ContentChunk], owner : Principal)

- **Description**: Creates the very first version of a file. It sets the parent version to 0, records the initial content as chunks, and stores the Commit and VersionStorage for this base version. It also updates the headsByFile map.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - chunks: [ContentChunk] - The initial content chunks of the file.
    - owner: Principal - The owner of the file (author of this initial commit).
- **Return Value**: Result<Version, Error>
    - ok(Version): Returns the Version of the initial commit.
- **Dependent/Related Functions**: nextVersion, Types.now(), Types.calculateTotalSize.

createCommit(fileId : FileId, patch : Patch, author : Principal, message : ?Text, newChunks : [ContentChunk])

- **Description**: Creates a new version (commit) based on an applied Patch. It validates the patch.base version against the current headVersion to prevent conflicts, then generates a new Version and records the Patch, author, message, and the resulting newChunks as part of the Commit. The headVersion for the file is updated.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - patch: Patch - The patch containing the editing operations.
    - author: Principal - The principal ID of the user who applied the patch.
    - message: ?Text - An optional commit message.
    - newChunks: [ContentChunk] - The state of the file's chunks *after* the patch has been applied.
- **Return Value**: Result<Version, Error>
    - ok(Version): Returns the Version of the new commit.
    - err(#NotFound): If the file's head version is not found.
    - err(#Conflict): If patch.base does not match the current headVersion.
- **Dependent/Related Functions**: nextVersion, Types.now(), Types.calculateTotalSize.

createSnapshot(fileId : FileId, author : Principal, message : ?Text, chunks : [ContentChunk])

- **Description**: Creates a manual snapshot (save point) of a file's current state. This is useful for marking significant points in a file's history. It creates a new Commit with an empty patch, storing the current chunks as the version's content.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - author: Principal - The principal ID of the user creating the snapshot.

    - message: ?Text - An optional message for the snapshot.
    - chunks: [ContentChunk] - The current content chunks of the file at the time of the snapshot.
- **Return Value**: Result<Version, Error>
    - ok(Version): Returns the Version of the new snapshot.
    - err(#NotFound): If the file's head version is not found.
- **Dependent/Related Functions**: nextVersion, Types.now(), Types.calculateTotalSize.

Internal Helper Functions (Event Generation for File Actions)

These functions are called by other modules (e.g., File Management) to generate appropriate real-time events when significant file actions occur.

applyPatch(fileId : FileId, patch : Patch, commit : Types.Commit)

- **Description**: Orchestrates the application of a client Patch. It first checks for duplicate operations using dedupeOpIdsByFile, records the operation, then generates a #PatchApplied Event and adds it to the file's event stream.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - patch: Patch - The patch to apply.
    - commit: Types.Commit - The commit record resulting from the patch application.
- **Return Value**: Result<Seq, Error>
    - ok(Seq): Returns the sequence number of the generated event.
    - err(#DuplicateOperation): If the clientOpId has already been processed.
- **Dependent/Related Functions**: isDuplicateOperation, recordOperation, addEvent, nextSeq, Types.now().

createSnapshotEvent(fileId : FileId, version : Version)

- **Description**: Generates a #Snapshot Event for a file. This is typically called after a manual save or autosave.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - version: Version - The version number of the created snapshot.
- **Return Value**: Result<Seq, Error> - Returns the sequence number of the generated event.
- **Dependent/Related Functions**: addEvent, nextSeq, Types.now().

createFileDeletedEvent(fileId : FileId, by : Principal)

- **Description**: Generates a #FileDeleted Event when a file is soft-deleted.
- **Arguments**:

    - fileId: FileId - The ID of the deleted file.
    - by: Principal - The principal ID of the user who deleted the file.
- **Return Value**: Result<Seq, Error> - Returns the sequence number of the generated event.
- **Dependent/Related Functions**: addEvent, nextSeq, Types.now().

createFileRestoredEvent(fileId : FileId, by : Principal)

- **Description**: Generates a #FileRestored Event when a file is restored from soft-deletion.
- **Arguments**:
    - fileId: FileId - The ID of the restored file.
    - by: Principal - The principal ID of the user who restored the file.
- **Return Value**: Result<Seq, Error> - Returns the sequence number of the generated event.
- **Dependent/Related Functions**: addEvent, nextSeq, Types.now().

Publicly Exposed Functions for Versioning & Cleanup (For Frontend Integration)

cleanupOldOperations(fileId : FileId, maxAgeNanos : Nat)

- **Description**: Cleans up old clientOpId entries from the deduplication map for a specific file. This helps manage memory by removing records of operations older than maxAgeNanos.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - maxAgeNanos: Nat - The maximum age (in nanoseconds) to retain operation IDs.
- **Return Value**: async Result<Nat, Error>
    - ok(Nat): Returns the number of old operation IDs removed.
- **Dependent/Related Functions**: Types.now().

cleanupFileData(fileId : FileId)

- **Description**: Cleans up all real-time collaboration related data for a given file, including its event buffer, presence information, subscriptions (if implemented), and deduplication records. This is typically called when a file is permanently deleted or becomes inactive.
- **Arguments**:
    - fileId: FileId - The ID of the file to clean up.
- **Return Value**: async Result<{ events : Nat; presence : Nat; operations : Nat }, Error>
    - ok(...): Returns an object showing the count of events, presence entries, and operations that were removed.
- **Dependent/Related Functions**: getEventCount, getActiveClientCount.

—---

Version Reading & Management

This section provides functions for accessing file version history, comparing versions, and rolling back to previous states.

Internal Helper Functions

getCommit(fileId : FileId, version : Version)

- **Description**: Retrieves the Commit information for a specific version of a file.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - version: Version - The specific version number.
- **Return Value**: Result<Commit, Error>
  - ok(Commit): Returns the Commit object for the specified version.
  - err(#NotFound): If the file or version does not exist.
- **Dependent/Related Functions**: None directly.

getVersionChunks(fileId : FileId, version : Version)

- **Description**: Retrieves all ContentChunks that constitute the content of a file at a specific version, ordered by their index.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - version: Version - The specific version number.
- **Return Value**: Result<[ContentChunk], Error>
  - ok([ContentChunk]): Returns an array of ContentChunks.
  - err(#NotFound): If the file or version does not exist.
- **Dependent/Related Functions**: Array.sort.

getVersionChunk(fileId : FileId, version : Version, chunkIndex : Nat)

- **Description**: Retrieves a single ContentChunk from a specific version of a file.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - version: Version - The specific version number.
  - chunkIndex: Nat - The index of the chunk to retrieve.
- **Return Value**: Result<ContentChunk, Error>
  - ok(ContentChunk): Returns the requested content chunk.
  - err(#NotFound): If the file, version, or chunk does not exist.

- **Dependent/Related Functions**: None directly.

getHeadVersion(fileId : FileId)

- **Description**: Retrieves the Version number of the current (latest) head of a file.
- **Arguments**:
    - fileId: FileId - The ID of the file.
- **Return Value**: Result<Version, Error>
    - ok(Version): Returns the head version number.
    - err(#NotFound): If the file's head version is not found (e.g., file doesn't exist).
- **Dependent/Related Functions**: None directly.

rollbackToVersion(fileId : FileId, targetVersion : Version, author : Principal, chunks : [ContentChunk])

- **Description**: Reverts a file's content to a specified targetVersion. This creates a new Commit representing the rollback, but the patch operations are empty as it's a content replacement. The file's headVersion is updated to this new rollback commit.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - targetVersion: Version - The version to which the file content should be reverted.
    - author: Principal - The principal ID of the user performing the rollback.
    - chunks: [ContentChunk] - The content chunks that represent the state of the targetVersion. (Frontend should retrieve these first using getVersionChunks and then pass them here).
- **Return Value**: Result<Version, Error>
    - ok(Version): Returns the Version of the new rollback commit.
    - err(#NotFound): If the targetVersion or file's head version does not exist.
- **Dependent/Related Functions**: getCommit, nextVersion, Types.now(), Types.calculateTotalSize.

pruneVersions(fileId : FileId, keepCount : Nat)

- **Description**: Cleans up old versions of a file, keeping only the specified keepCount of the most recent versions, plus the current head version (if not among the keepCount). This helps manage storage space for version history.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - keepCount: Nat - The number of recent versions to retain.
- **Return Value**: Result<Nat, Error>
    - ok(Nat): Returns the number of versions that were removed.
    - err(#NotFound): If the file or its head version is not found.

- **Dependent/Related Functions**: Array.sort, getHeadVersion.

Publicly Exposed Functions for Version Management (For Frontend Integration)

listVersions(fileId : FileId, offset : Nat, limit : Nat)

- **Description**: Retrieves a paginated list of Commit objects for a specific file, ordered from newest to oldest. Requires user access to the file.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - offset: Nat - The starting offset for pagination.
  - limit: Nat - The maximum number of commits to return per page.
- **Return Value**: async Result<Paginated<Commit>, Error>
  - ok(Paginated<Commit>): Returns a Paginated object containing an array of Commits, the next offset, and total count.
  - err(#NotFound): If the file or its version history is not found.
  - err(#AccessDenied): If the caller does not have access to the file.
- **Dependent/Related Functions**: hasAccess, Array.sort.

getVersionHistory(fileId : FileId, fromVersion : Version)

- **Description**: Retrieves the linear parent history of a file starting from a specified fromVersion up to the initial version (version 0). This is useful for understanding the lineage of changes. Requires user access to the file.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - fromVersion: Version - The version from which to start tracing the history.
- **Return Value**: async Result<[Commit], Error>
  - ok([Commit]): Returns an array of Commits in reverse chronological order (from fromVersion backward).
  - err(#NotFound): If the file or the specified version does not exist.
  - err(#AccessDenied): If the caller does not have access to the file.
- **Dependent/Related Functions**: hasAccess, getCommit.

getVersionDiff(fileId : FileId, fromVersion : Version, toVersion : Version)

- **Description**: Retrieves the EditOps that were applied to transition from fromVersion to toVersion. Currently, it simply returns the patch operations of the toVersion's commit. For a full textual diff, the frontend would need to reconstruct content and compare. Requires user

access.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - fromVersion: Version - The base version for the comparison.
  - toVersion: Version - The target version for the comparison.
- **Return Value**: async Result<[EditOp], Error>
  - ok([EditOp]): Returns an array of EditOps associated with the toVersion's commit.
  - err(#NotFound): If either fromVersion or toVersion (or the file) does not exist.
  - err(#AccessDenied): If the caller does not have access to the file.
- **Dependent/Related Functions**: hasAccess, getCommit.

isAncestor(fileId : FileId, ancestor : Version, descendant : Version)

- **Description**: Checks if one version is an ancestor of another version in the file's history. This means ancestor logically precedes descendant in the version chain. Requires user access.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - ancestor: Version - The potential ancestor version.
  - descendant: Version - The potential descendant version.
- **Return Value**: async Result<Bool, Error>
  - ok(true): If ancestor is indeed an ancestor of descendant (or they are the same version).
  - ok(false): If ancestor is not an ancestor of descendant.
  - err(#NotFound): If the file or either version does not exist.
  - err(#AccessDenied): If the caller does not have access to the file.
- **Dependent/Related Functions**: hasAccess.

deleteAllVersions(fileId : FileId)

- **Description**: Permanently deletes all versions (commits and their associated content snapshots) for a specified file. This action cannot be undone. Requires user access to the file.
- **Arguments**:
  - fileId: FileId - The ID of the file whose versions are to be deleted.
- **Return Value**: async Result<Nat, Error>
  - ok(Nat): Returns the count of versions that were deleted.
  - err(#NotFound): If the file does not exist.
  - err(#AccessDenied): If the caller does not have access to the file.
- **Dependent/Related Functions**: hasAccess.

getVersionCount(fileId : FileId)

- **Description**: Retrieves the total number of versions (commits) available for a specific file.

Requires user access.

- **Arguments**:
  - fileId: FileId - The ID of the file.
- **Return Value**: async Nat - The total number of versions. Returns 0 if the file doesn't exist or access is denied.
- **Dependent/Related Functions**: hasAccess.

getVersionStorageUsed(fileId : FileId)

- **Description**: Calculates the total storage space (in bytes) consumed by all versions of a specific file. Requires user access.
- **Arguments**:
  - fileId: FileId - The ID of the file.
- **Return Value**: async Nat - The total storage used by the file's versions. Returns 0 if the file doesn't exist or access is denied.
- **Dependent/Related Functions**: hasAccess.

versionExists(fileId : FileId, version : Version)

- **Description**: Checks if a specific version exists for a given file. Requires user access.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - version: Version - The version number to check.
- **Return Value**: async Bool - true if the version exists, false otherwise (including if the file doesn't exist or access is denied).
- **Dependent/Related Functions**: hasAccess.

getLatestCommitMessage(fileId : FileId)

- **Description**: Retrieves the commit message of the current head version of a file. Requires user access.
- **Arguments**:
  - fileId: FileId - The ID of the file.
- **Return Value**: async Result<Text, Error>
  - ok(Text): Returns the commit message (empty string if no message).
  - err(#NotFound): If the file or its head version does not exist.
  - err(#AccessDenied): If the caller does not have access to the file.
- **Dependent/Related Functions**: hasAccess, getHeadVersion, getCommit.

getCommitAuthor(fileId : FileId, version : Version)

- **Description**: Retrieves the Principal ID of the author for a specific version of a file. Requires user access.

- **Arguments**:
  - fileId: FileId - The ID of the file.
  - version: Version - The version number.
- **Return Value**: async Result<Principal, Error>
  - ok(Principal): Returns the author's principal ID.
  - err(#NotFound): If the file or version does not exist.
  - err(#AccessDenied): If the caller does not have access to the file.
- **Dependent/Related Functions**: hasAccess, getCommit.

getCommitTime(fileId : FileId, version : Version)

- **Description**: Retrieves the timestamp (Time.Time) when a specific version of a file was committed.
- **Arguments**:
  - fileId: FileId - The ID of the file.
  - version: Version - The version number.
- **Return Value**: async Result<Time.Time, Error>
  - ok(Time.Time): Returns the commit timestamp.
  - err(#NotFound): If the file or version does not exist.
- **Dependent/Related Functions**: getCommit.

Public Interface Integration (Main Actor Functions)

These functions represent the primary public entry points for frontend interaction with the Storage canister, orchestrating calls to the internal modules (File Management, Versioning, Realtime, Autosave).

create_file(name : Text, tableId : Nat, mime : Text, initialContent : ?Blob)

- **Description**: Creates a new file in the system. This function integrates file creation with initial versioning and sets up a default autosave policy for the new file. It also performs access control checks to ensure the caller is a collaborator of the tableId.
- **Arguments**:
  - name: Text - The name of the file.
  - tableId: Nat - The ID of the table the file belongs to.
  - mime: Text - The MIME type of the file.
  - initialContent: ?Blob - Optional initial binary content.
- **Return Value**: async Result<FileId, Error>
  - ok(FileId): Returns the ID of the newly created file.
  - err(#AccessDenied): If the caller is not a collaborator of the table.

    - err(Error): Propagates errors from internal createFile, getAllChunks, createInitialVersion, or setAutosavePolicy.
- **Dependent/Related Functions**: hasAccess (internal helper), TableManagement.get_table_collaborators, createFile (internal), getAllChunks, createInitialVersion, Autosave.getDefaultPolicy, setAutosavePolicy.

delete_file(fileId : FileId)

- **Description**: Soft-deletes a file. Only the file owner can perform this. It marks the file as deleted and generates a FileDeleted event.
- **Arguments**:
    - fileId: FileId - The ID of the file to delete.
- **Return Value**: async Result<(), Error>
    - ok(()): Returns unit on success.
    - err(#AccessDenied): If the caller is not the file owner.
    - err(Error): Propagates errors from hasAccess or deleteFile.
- **Dependent/Related Functions**: hasAccess, deleteFile (internal), createFileDeletedEvent (internal).

restore_file(fileId : FileId)

- **Description**: Restores a previously soft-deleted file. Only the file owner can perform this. It marks the file as not deleted and generates a FileRestored event.
- **Arguments**:
    - fileId: FileId - The ID of the file to restore.
- **Return Value**: async Result<(), Error>
    - ok(()): Returns unit on success.
    - err(#AccessDenied): If the caller is not the file owner.
    - err(Error): Propagates errors from hasAccess or restoreFile.
- **Dependent/Related Functions**: hasAccess, restoreFile (internal), createFileRestoredEvent (internal).

update_chunk(fileId : FileId, chunkIndex : Nat, data : Blob)

- **Description**: Updates a specific chunk of a file. This generates a new version and records activity for autosave. Requires editor permissions.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - chunkIndex: Nat - The index of the chunk to update.
    - data: Blob - The new binary data for the chunk.
- **Return Value**: async Result<Version, Error>

    - ok(Version): Returns the new Version of the file.
    - err(#AccessDenied): If the caller does not have edit access.
    - err(Error): Propagates errors from hasAccess or updateChunk.
- **Dependent/Related Functions**: hasAccess, updateChunk (internal), recordActivity.

replace_file_content(fileId : FileId, newChunks : [ContentChunk])

- **Description**: Replaces the entire content of a file with new chunks. This generates a new version and records activity for autosave. Requires editor permissions.
- **Arguments**:
    - fileId: FileId - The ID of the file.
    - newChunks: [ContentChunk] - The new array of content chunks.
- **Return Value**: async Result<Version, Error>
    - ok(Version): Returns the new Version of the file.
    - err(#AccessDenied): If the caller does not have edit access.
    - err(Error): Propagates errors from hasAccess or replaceFileContent.
- **Dependent/Related Functions**: hasAccess, replaceFileContent (internal), recordActivity.

System Functions (Upgrades)

These functions are critical for maintaining the canister's state across upgrades.

system func preupgrade()

- **Description**: This function is automatically called by the Internet Computer replica before an upgrade. Its purpose is to serialize the current volatile state of the canister into stable variables, ensuring data persistence across the upgrade. It converts HashMap data structures into their stable Array representations.
- **Arguments**: None.
- **Return Value**: None.
- **Dependent/Related Functions**: fileStorageToStable, versionStorageToStable, stableFiles, stableFilesByTableId, stableVersions, stableEvents, stablePresence, stableSubscriptions, stableAutosavePolicies.

system func postupgrade()

- **Description**: This function is automatically called by the Internet Computer replica after an upgrade. Its purpose is to deserialize the data from stable variables back into the canister's active (volatile) state, restoring the application to its previous operational state. It also clears the stable variables.

- **Arguments**: None.
- **Return Value**: None.
- **Dependent/Related Functions**: stableToFileStorage, stableToVersionStorage, createEventBuffer. It uses the global stable counters (nextFileId, nextVersion, nextSeq) that implicitly retain their state across upgrades without explicit restoration here.

User Story: Collaborative Document Versioning and Real-time Editing (Full Flow)

1. **Alex Creates a New Document in a Shared Table**: Alex logs into the application and decides to create a new "Project Proposal" document within his "Team Project Alpha" table (Table ID: 5).
   - *Frontend Action*: Calls Storage.create_file("Project Proposal", 5, "text/plain", null).
   - *Canister Interaction*:
     - create_file first calls hasAccess to ensure Alex is a collaborator of Table 5.
     - Then, it internally calls createFile to set up basic file metadata and allocate fileId (e.g., 101).
     - It fetches initial (empty) chunks via getAllChunks.
     - createInitialVersion is called to record the first version of the file.
     - A default AutosavePolicy is set using setAutosavePolicy.
   - *Canister Response*: Returns ok(101) (the new file ID).
   - *Frontend Update*: The UI displays "Project Proposal" in Table 5's file list and opens a new, empty document editor for Alex.
2. **Alex Starts Editing and Mia Joins**: Alex begins typing. Mia, a collaborator on the same table, sees the new document and decides to join the editing session.
   - *Frontend Action (Alex)*: As Alex types, the frontend generates Patch objects. These are conceptually sent to the canister via internal functions which will call applyPatch. Simultaneously, recordActivity(101) is called periodically.
   - *Frontend Action (Mia)*: Generates a clientId for Mia (e.g., client_Mia_123) and calls Storage.join_file(101, "client_Mia_123"). It then polls for events: Storage.getEvents(101, 0, 100) and Storage.getActiveClients(101).
   - *Canister Interaction (apply_patch)*: apply_patch checks for duplicate operations (isDuplicateOperation), records them (recordOperation), createCommit creates a new version, and addEvent adds a #PatchApplied event to the eventsByFile buffer.
   - *Canister Interaction (join_file)*: Adds Mia to presenceByFile for fileId: 101 and adds a #Presence(#Join) event to eventsByFile.

    - *Frontend Update*: Alex's changes appear in his editor. Mia's editor loads, shows Alex's current text, and displays Alex's cursor/presence. Mia's clientId is also visible to Alex.
3. **Alex and Mia Collaborate with Cursors**: They both continue editing, and their cursors are visible to each other.
    - *Frontend Action (Both)*: Periodically calls Storage.update_cursor(101, theirCurrentCursorInfo).
    - *Canister Interaction*: updatePresence (called by update_cursor) updates the ClientPresence in presenceByFile and generates #CursorUpdate events, adding them to the stream.
    - *Frontend Action (Both, polling)*: Continuously calls Storage.getEvents(101, lastSeenSeq, 100).
    - *Canister Response (getEvents)*: Returns new #PatchApplied and #CursorUpdate events.
    - *Frontend Update*: Both editors reflect real-time changes and synchronized cursors.
4. **Alex Creates a Manual Snapshot**: Alex finishes the first draft and wants a save point.
    - *Frontend Action*: Calls Storage.create_snapshot(101, "First Draft Complete").
    - *Canister Interaction*: create_snapshot first gets all current chunks using getAllChunks, then calls the internal createSnapshot function to create the new version, and finally calls createSnapshotEvent to add a #Snapshot event to the event stream.
    - *Canister Response*: The version number of the new snapshot.
    - *Frontend Update*: A "Snapshot created" message appears, and the new version is visible in the document's version history interface. Storage.listVersions(101, 0, 10) would now show this new snapshot.
5. **Mia Leaves the Document, Then Restores a Previous Version**: Mia closes her browser. Later, she reopens the document and decides the content from Alex's "First Draft Complete" snapshot is better than the current messy state.
    - *Frontend Action (Mia Leaves)*: Calls Storage.leaveFile(101, Mia's ClientId).
    - *Canister Response*: Mia's presence is removed, and a #Presence(#Leave) event is added.
    - *Frontend Action (Mia Restores)*:
        - Mia finds the "First Draft Complete" snapshot in the version history using Storage.listVersions. She then calls Storage.rollback_to_version(101, snapshotVersionId).
    - *Canister Interaction (rollback_to_version)*: It first uses getVersionChunks to get the content of the targetVersion. Then, it calls the internal rollbackToVersion to create a new Commit representing the rollback and updates the headVersion. Finally, it calls replaceFileContent to update the actual file content to that of the target version.
    - *Frontend Update*: The document content reverts to the snapshot, and a new version is added to the history, indicating the rollback.
6. **Alex Reviews Version History and Compares**: Alex wants to see the changes between two

specific versions.

- *Frontend Action*: Calls Storage.listVersions(101, 0, 20) to get a list of versions. Then, selecting two versions (e.g., version A and version B), calls Storage.getVersionDiff(101, versionA, versionB). He might also use Storage.isAncestor(101, versionA, versionB) to verify their relationship.
- *Canister Response*: Returns Paginated<Commit> for listVersions, [EditOp] for getVersionDiff, and Bool for isAncestor.
- *Frontend Update*: Displays the version history and highlights the differences between the chosen versions in a diff viewer.

7. **Admin Manages Autosave and Performs Cleanup**: An administrator wants to configure the autosave policy for a specific file and later perform general data cleanup.
   - *Admin Frontend Action (Set Autosave Policy)*: Calls Storage.set_autosave_policy(101, { intervalNanos = 60_000_000_000; idleNanos = 10_000_000_000; enabled = true; maxVersions = 100; }).
   - *Canister Response*: The policy for fileId: 101 is updated.
   - *Admin Frontend Action (Trigger Manual Autosave)*: The admin (or an automated job) notices Storage.is_autosave_due(101) returns true and calls Storage.process_autosave(101).
   - *Canister Interaction*: process_autosave performs the autosave by calling saveFunction (which internally calls createSnapshot) and then markSaved.
   - *Canister Response*: Returns the version of the new autosaved snapshot.
   - *Admin Frontend Action (General Cleanup)*: Calls Storage.cleanup_old_data().
   - *Canister Response*: This function cleans up soft-deleted files (cleanupDeletedFiles), prunes old versions for all files (pruneVersions), and removes old events from event buffers.
   - *Frontend Update*: Admin dashboard reflects updated autosave settings and cleanup results.