🖥️ PrivacySafe 3NWeb Platform: API Spec

This document provides a human-readable reference for the APIs of the 3NWeb `spec-server`, as implemented in the PrivacySafe ecosystem.

🧭 Purpose and Scope

3NWeb is a specification and platform for building federated, privacy-first applications that operate securely across devices and services. This documentation is intended for developers building server-side extensions, service endpoints, and core libraries.

🔍 API Organization

The documentation is grouped by subsystem or domain, including:

- **ASMail** — Encrypted messaging infrastructure (delivery, retrieval, sessions)
- **3NStorage** — Decentralized, versioned object storage (sync, archiving, session-based APIs)
- **Admin** — Tools for managing users, signup tokens, and domain provisioning
- **Config & CLI** — Runtime and environment configuration, command-line parsing
- **Common Libraries** — Core utilities used by all subsystems (assertions, encoding, crypto, etc.)

⚙️ Core Service Initializers

These entry points are responsible for bootstrapping core app modules based on the enabled features:

`services.ts`

- **Function:** `servicesApp(conf, errLogger, logSetup)`

  Starts services like ASMail, 3NStorage, and MailerId, based on `Configurations`. This is the main orchestrator for service processes.

- **Function:** `adminApp(conf, errLogger, logSetup)`

  Sets up administrative interfaces for managing users, domains, and mailer identities.

🗃️ 3NStorage: Versioned Object Storage

This module implements end-user file and blob storage, using version control and object-chunking with sessioned APIs.

`3nstorage/3nstorage-app.ts`

- **Function:** `makeApp(dataFolder, domain, midAuthorizer, errLogger)`

  Creates the main 3NStorage service app, which handles routing and object persistence for user-scoped encrypted files.

`3nstorage/owner.ts`

- **Function:** `makeApp(domain, sessions, users, midAuthorizer)`

  Constructs the storage handler for a domain owner. Manages session authentication, object ownership, and version reconciliation.

`3nstorage/shared.ts`

- **Function:** `makeApp(sessions, users)`

  Provides access-control–governed file sharing for secondary users.

`3nstorage/resources/sessions.ts`

- **Function:** `makeSessionFactory(timeout)`

  Generates session instances for storage operations, expiring based on timeout values to protect state.

`3nstorage/resources/store.ts`

- **Interfaces:** `MismatchedObjVerException`, `ObjStatusInfo`, `TransactionParams`

Core models for tracking version errors, object lifecycle state, and transactional write parameters.

- **Function:** `makeMismatchedObjVerException(current_version)`

  Generates a structured error object when client and server versions diverge.

📬 ASMail: Encrypted Messaging System

ASMail provides secure end-to-end encrypted communication for 3NWeb users. Messages are delivered via temporary delivery sessions and retrieved via inbox-style APIs.

`asmail/asmail-app.ts`

- **Function:** `makeApp(rootFolder, domain, midAuthorizer, errLogger)`

  Initializes the full ASMail delivery and retrieval system, including message queues, inbox resolution, and storage-backed blob handling.

`asmail/config.ts`

- **Function:** `makeApp(domain, sessions, recipients, midAuthorizer)`

  Loads or prepares ASMail domain-specific settings and session context for message flows.

`asmail/delivery.ts`

- **Function:** `makeApp(domain, sessions, recipients, midAuthorizer)`

  Bootstraps delivery-side logic for ASMail — validates senders, segments messages, and handles chunked uploads.

`asmail/retrieval.ts`

- **Function:** `makeApp(domain, sessions, recipients, midAuthorizer)`

  Sets up message inbox resolution and message object retrieval. Also provides support for message listing and metadata access.

`asmail/resources/recipients.ts`

- **Interface:** `Factory` — Provides methods for validating users, storing public keys, accepting uploads, and fetching inbox data.
- **Function:** `makeFactory(rootFolder)`

  Initializes recipient-based message storage and retrieval. Connects encrypted blobs with public key encryption and inbox lifecycles.

`asmail/resources/delivery-sessions.ts`

- **Interface:** `SessionParams` — Defines structure for sender, recipient, message ID, and remaining chunk space.

`asmail/resources/sessions.ts`

- **Function:** `makeSessionFactory(timeout)` — Session generator for delivery tracking and per-message encryption state.

🛠️ Admin: User, Domain & Identity Provisioning

The Admin subsystem handles the setup and governance of user identities, available domains, and signup flows. It's typically accessed by privileged system operators.

`admin/admin-app.ts`

- **Function:** `makeApp(conf, errLogger, logSetup)`

  Constructs the Admin service responsible for user provisioning, domain setup, and user-address validation.

`admin/signup-tokens.ts`

- **Interfaces:** `SingleUserSignupCtx`, `MultiDomainSignupCtx`

  Token-based signup contexts for individual users or multi-domain configurations.

- **Functions:** `canCreateUser`, `addressesForName`, `makeSingleUserSignupCtx`, `makeMultiDomainSignupCtx`, `parseToken`, `tokenPath`, `areTokensSame`

  Create and validate signup tokens. Parse and compare token values, bind them to specific user addresses or domains.

`admin/resources/users.ts`

- **Interface:** `Factory` — Operations for address/domain validation and adding new users.
- **Function:** `makeFactory(rootFolder, noTokenFile)` — Sets up persistent user provisioning backend.
- **Functions:** `validateUserMidParams`, `validateUserStorageParams`

  Ensures parameter sets for MailerId and Storage meet security and structural constraints.

`admin/routes/`

- `add.ts` → `addUser()` — Adds a new user to the platform via token or admin override.
- `get-available-addresses.ts` → `availableAddresses()` — Returns valid addresses for a given name.
- `get-available-domains.ts` → `availableDomains()` — Lists domains where user accounts may be created.

⚙️ Configuration & CLI Interfaces

These files define runtime configuration loading, command-line parsing, and service startup controls. Useful for operational and deploy-time management.

`config/from-yaml.ts`

- **Function:** `readYamlConfFile(path)` — Reads YAML-formatted configuration from disk and returns it as structured config objects.

`config/letsencrypt.ts`

- **Function:** `sslOptsFromConfig(letsencrypt, sslOpts, skipReload)` — Constructs SSL/TLS configuration from either Let's Encrypt folders or pre-loaded certs.

`config/cli/index.ts`

- **Function:** `parseProcessArgv()` — Parses process command-line arguments and flags into CLI config structure.

`config/cli/run-cmd.ts`

- **Function:** `parseRunArgs(args, execName)` — Handles CLI inputs for starting 3NWeb services.
- **Interface:** `ParsedRunArgs` — Command-line option schema.

`config/cli/signup-cmd.ts`

- **Function:** `parseSignupArgs(args, execName)` — CLI for managing user signup tokens and accounts.
- **Interface:** `ParsedSignupArgs` — Signup-related CLI args schema.

`config/cli/user-cmd.ts`

- **Function:** `parseSignupArgs(args, execName)` — Special user-centric flag parsing.
- **Interface:** `ParsedUserArgs` — Args for listing users or showing help.

`config/cli/utils.ts`

- **Interfaces:** `OptionDef`, `UsageSection`, `HelpArg`, `CliUsageDisplay`
- **Functions:** `toUsageDisplay(exitStatus, usage)`, `toErrorUsage(usage, errTxt)`

  Display help and error output for CLI usage. Also defines custom flag types and default settings.

🧱 lib-common: Core Utilities and Primitives

This package includes foundational building blocks used across 3NWeb modules: encoding, assertions, file system wrappers, public key structures, etc.

`assert.ts`

- **Function:** `assert(ok, errMsg?)` — Throws a runtime error if the condition is not met. Basic internal assertion utility.

`async-fs-node.ts`

- Async wrappers around Node.js `fs` module for reading, writing, opening, and managing files and folders. Supports buffering and custom options.

- Notable functions: `readFile`, `writeFile`, `mkdir`, `symlink`, `readlink`, `stat`, `unlink`, `truncate`, `copyFile`

`buffer-utils.ts`

- Encoding helpers for converting JSON and byte arrays to/from buffers.
- Functions: `toBuffer`, `bufFromJson`, `joinByteArrs`, `allCharsFromAlphabet`

`canonical-address.ts`

- Tools for address normalization and comparison.
- Functions: `toCanonicalAddress`, `areAddressesEqual`, `checkAndTransformAddress`

`json-equal.ts`, `json-utils.ts`

- **Function:** `deepEqual(a, b)` — Performs deep equality check between JSON-like objects.

`bytes-equal.ts`

- **Function:** `bytesEqual(a, b)` — Byte-level equality for cryptographic comparison.

`jwkeys.ts`

- Interfaces for public/private key JSON representations used in MailerId and message signing.
- (Described in detail in the full OpenAPI schema.)

📦 Project Structure

Folder/File Path	Description
lib.ts	Foundational utilities and shared constants for general reuse.
main.ts	Entry point for the application or module.

services.ts	Launches `adminApp` and `servicesApp` with shared config.
3nstorage/3nstorage-app.ts	Initializes the 3nstorage application with auth and logging.
3nstorage/owner.ts	Main owner-facing API for encrypted object storage.
3nstorage/shared.ts	Shared logic used across 3nstorage components.
3nstorage/resources/sessions.ts	Manages session creation and timeouts.
3nstorage/resources/store.ts	Metadata types and tools for versioned object storage.
3nstorage/resources/users.ts	User-facing operations for object version access.
3nstorage/routes/owner/archive-obj-version.ts	Route handler to archive an object's current version.
3nstorage/routes/owner/cancel-trans.ts	Cancels in-progress object transactions.
3nstorage/routes/owner/delete-obj.ts	Deletes the current version of an object.
3nstorage/routes/owner/put-current-obj.ts	Saves current version of an object with chunking logic.
3nstorage/routes/owner/session-params.ts	Injects session-specific parameters like max chunk size.
admin/admin-app.ts	Bootstraps the admin backend app with config and logging.
admin/signup-tokens.ts	Manages user/domain signup tokens and validation logic.
admin/resources/users.ts	Admin utilities for user creation, validation, and factories.

admin/routes/add.ts	Endpoint to add new users.
admin/routes/get-available-addresses.ts	Lists valid email addresses for signup.
admin/routes/get-available-domains.ts	Lists domains available for user signup.
asmail/asmail-app.ts	Bootstraps the ASMail app for secure messaging.
asmail/config.ts	Sets up ASMail component configuration.
asmail/delivery.ts	Handles message delivery features of ASMail.
asmail/retrieval.ts	Message fetching and inbox logic for ASMail.
asmail/resources/delivery-sessions.ts	Metadata structure for delivery session tracking.
asmail/routes/delivery/restart-session.ts	Restarts a partial or interrupted message delivery.
asmail/routes/delivery/sender-authorization.ts	Authenticates senders via domain credentials.
asmail/routes/delivery/start-session.ts	Initializes new message delivery sessions.
asmail/routes/retrieval/get-message-meta.ts	Provides metadata about a message.
asmail/routes/retrieval/get-message-obj.ts	Retrieves full message content by ID.
asmail/routes/retrieval/list-messages.ts	Lists all message IDs for a recipient.
asmail/routes/retrieval/remove-message.ts	Deletes a stored message securely.

config/from-yaml.ts	Loads YAML config files for runtime setup.
config/letsencrypt.ts	Manages TLS settings using Let's Encrypt certs.
config/signup.ts	Defines signup-related config logic.
config/cli/index.ts	CLI entry point for argument parsing and dispatch.
config/cli/run-cmd.ts	Handles CLI arguments for general app execution.
config/cli/signup-cmd.ts	CLI tools for managing signup tokens and domains.
config/cli/user-cmd.ts	CLI interface for listing or managing users.
config/cli/utils.ts	Defines shared CLI types and rendering helpers.
lib-common/assert.ts	Runtime assertion utility.
lib-common/async-fs-node.ts	Async wrappers for file system methods in Node.

📄 lib.ts

Purpose: Defines foundational utilities, shared constants, or core logic intended to be reused across multiple parts of the project.

📄 main.ts

Purpose: Acts as the primary entry point for the application or module.

📄 services.ts

Purpose: Defines application entry points for launching the `adminApp` and `servicesApp`, both based on a shared configuration.

🧩 Interface: `Configurations`

interface Configurations {
enabledServices: {
ASMail?: boolean;
storage?: boolean;
mailerId?: boolean;
}
}

Describes the available services enabled in the application configuration.

⚙️ Function: `servicesApp`

function servicesApp(conf: Configurations, errLogger?: ErrLogger | 'console', logSetup?: 'console')

Initializes and returns the services application, using the provided configuration and optional error logging.

⚙️ Function: `adminApp`

function adminApp(conf: Configurations, errLogger?: ErrLogger | 'console', logSetup?: 'console')

Initializes and returns the admin application, similarly configured with logging and error handling.

📄 `3nstorage/3nstorage-app.ts`

Purpose: Bootstraps the `3nstorage` application, handling mid authorization and optional error logging.

⚙️ Function: `makeApp`

function makeApp(dataFolder: string, domain: string, midAuthorizer: MidAuthorizer, errLogger?: ErrLogger)

Constructs the encrypted storage application given storage path, domain, and authorization logic.

📄 3nstorage/owner.ts

Purpose: Serves as the main app interface for owner-based access to encrypted object storage.

⚙️ Function: `makeApp`

function makeApp(domain: string, sessions: SessionsFactory, users: usersFactory, midAuthorizer: MidAuthorizer)

Builds the owner-facing storage app with user/session handling and authorization mechanisms.

📄 3nstorage/shared.ts

Purpose: Shared app logic for use between components in the `3nstorage` module.

⚙️ Function: `makeApp`

function makeApp(sessions: SessionsFactory, users: usersFactory)

Initializes shared application logic using the provided session and user factories.

📄 3nstorage/resources/sessions.ts

Purpose: Defines session management tools for object interaction workflows.

⚙️ Function: `makeSessionFactory`

function makeSessionFactory(timeout: number)

Creates a factory that produces session objects with the specified timeout period.

📄 3nstorage/resources/store.ts

Purpose: Contains types and utilities for object version tracking and storage metadata.

🧩 Interface: `MismatchedObjVerException`

```
interface MismatchedObjVerException {
type: 'mismatched-obj-ver';
current_version: number;
}
```

Describes a version mismatch error when working with stored objects.

🧩 Interface: `ObjStatusInfo`

```
interface ObjStatusInfo {
state: 'new' | 'current' | 'archived';
currentVersion?: number;
archivedVersions?: number[];
}
```

Provides status metadata about the current and archived versions of an object.

🧩 Interface: `TransactionParams`

```
interface TransactionParams {
isNewObj?: boolean;
version: number;
baseVersion?: number;
}
```

Describes parameters required for storing or modifying object versions.

⚙️ Function: `makeMismatchedObjVerException`

```
function makeMismatchedObjVerException(current_version: number)
```

Constructs a structured exception object for version mismatches.

📄 `3nstorage/resources/users.ts`

Purpose: Provides the user-facing factory interface to interact with stored objects and their metadata.

🧩 Interface: `Factory`

```
interface Factory {
exists: UserExists;
```

getKeyDerivParams: GetKeyDerivParams;
setKeyDerivParams: SetKeyDerivParams;
getSpaceQuota: GetSpaceQuota;
cancelTransaction: CancelTransaction;
getCurrentObj: GetCurrentObj;
saveNewObjVersion: SaveNewObjVersion;
saveNewRootVersion: SaveNewObjVersion;
getCurrentRootObj: GetCurrentObj;
getObjStatus: GetObjStatus;
archiveObjVersion: ArchiveObjCurrentVersion;
getArchivedRootVersion: GetArchivedObjVersion;
getArchivedObjVersion: GetArchivedObjVersion;
deleteCurrentObjVersion: DeleteCurrentObjVersion;
deleteArchivedObjVersion: DeleteArchivedObjVersion;
setStorageEventsSink(sink: EventsSink): void;
}

Defines user-level storage operations for creating, retrieving, updating, deleting, and archiving encrypted objects.

⚙️ Function: `makeFactory`

function makeFactory(rootFolder: string,
writeBufferSize?: string | number, readBufferSize?: string | number)

Creates a user-facing storage factory instance, parameterized by folder path and buffer sizes.

3nstorage/routes/owner/archive-obj-version.ts

Purpose: Route handler to archive an object's current version in owner-scoped storage.

⚙️ Function: `archiveCurrentObjVersion`

function archiveCurrentObjVersion(root: boolean, archiveObjVerFunc:
ArchiveObjCurrentVersion)

Wraps the archiving logic into a route-ready function for secure access by the storage owner.

3nstorage/routes/owner/cancel-trans.ts

Purpose: Route handler to cancel an in-progress object transaction.

⚙️ Function: `cancelTransaction`

function cancelTransaction(root: boolean, cancelTransFunc: CancelTransaction)

Implements cancellation logic for object version transactions via owner route.

📄 3nstorage/routes/owner/delete-obj.ts

Purpose: Route handler for removing the current version of a stored object.

⚙️ Function: `deleteCurrentObjVer`

function deleteCurrentObjVer(deleteObjFunc: DeleteCurrentObjVersion)

Removes the current version of an object using the configured deletion strategy.

📄 3nstorage/routes/owner/put-current-obj.ts

Purpose: Handles the saving of the current encrypted object version. Part of the object lifecycle management, this function coordinates storing and versioning of user-owned objects using segmentation and defined limits.

⚙️ Function: `saveCurrentObj`

function saveCurrentObj(root: boolean, saveObjFunc: SaveNewObjVersion, chunkLimit: string|number)

Stores a new version of an object, chunked according to a specified limit. Accepts a function that handles the actual object storage process (`saveObjFunc`), a `root` flag to determine if this is a root context, and `chunkLimit` which sets the max size for each object segment.

📄 3nstorage/routes/owner/session-params.ts

Purpose: Manages session-specific parameters like maximum chunk size for object storage. Used to customize or restrict upload behavior for a given session.

⚙️ Function: `sessionParams`

function sessionParams(maxChunk: number|string)

Returns a middleware or handler that injects session-related configuration, particularly the `maxChunk` limit which governs how large each object segment can be during upload or processing.

📄 admin/admin-app.ts

Purpose: Main entry point for the admin server-side application. This module sets up configuration, error handling, and optional logging behavior for administrative services.

⚙️ Function: `makeApp`

function makeApp(conf: Configurations, errLogger?: ErrLogger, logSetup?: 'console')

Initializes and returns the admin application. Accepts `Configurations`, an optional `ErrLogger`, and an optional `logSetup` to define how logs should be output (e.g., to console).

📄 admin/signup-tokens.ts

Purpose: Defines logic and data structures for managing user signup tokens, including both single-user and multi-domain contexts.

🧩 Interface: `SingleUserSignupCtx`

interface SingleUserSignupCtx {
token: string;
type: 'single-user';
userId: string;
validTill?: number;
}

Represents a signup context specifically for a single user. Includes a unique token, user ID, and optional expiration.

🧩 Interface: `MultiDomainSignupCtx`

interface MultiDomainSignupCtx {
token?: string;

type: 'multi-domain';
domains: string[];
validTill?: number;
}

Describes a signup context where access is granted based on permitted domains. Useful for multi-tenant or multi-org platforms.

⚙️ Function: `canCreateUser`

function canCreateUser(userId: string, ctx: SignupContext)

Checks whether the provided `userId` is authorized to be created under the given signup context.

⚙️ Function: `addressesForName`

function addressesForName(name: string, ctx: SignupContext)

Determines valid email addresses or aliases for a new user name based on signup policy.

⚙️ Function: `makeSingleUserSignupCtx`

function makeSingleUserSignupCtx(token: string, userId: string, validitySecs?: number)

Creates a `SingleUserSignupCtx` with an optional expiration time.

⚙️ Function: `makeMultiDomainSignupCtx`

function makeMultiDomainSignupCtx(signupDomains: string[], token?: string, validitySecs?: number)

Builds a `MultiDomainSignupCtx` that allows users from specific domains to sign up using an optional token.

⚙️ Function: `parseToken`

function parseToken(token: string)

Parses and decodes a signup token string into a usable object structure.

⚙️ Function: `tokenPath`

function tokenPath(rootFolder: string, token: string)

Determines the filesystem path for storing or retrieving token metadata based on a token ID.

⚙️ Function: `areTokensSame`

function areTokensSame(t1: Buffer, t2: string)

Performs a secure comparison between two tokens to prevent timing attacks or mismatches.

📄 `admin/resources/users.ts`

Purpose: Provides administrative utilities for managing users, including validation and user factory setup.

🧩 Interface: `Factory`

interface Factory {
getAvailableDomains: IAvailableDomains;
getAvailableAddresses: IAvailableAddressesForName;
add: IAdd;
}

Represents a user management factory. It provides methods to retrieve domain/address options and add new users.

⚙️ Function: `validateUserMidParams`

function validateUserMidParams(params: UserMidParams)

Validates mid-layer user parameters. Ensures required values for middleware operations are present and well-formed.

⚙️ Function: `validateUserStorageParams`

function validateUserStorageParams(params: UserStorageParams)

Validates the user's storage configuration. Ensures data-related parameters are structured properly before user creation.

⚙️ Function: `makeFactory`

function makeFactory(rootFolder: string, noTokenFile: string|undefined)

Creates a new user management factory using the specified root directory and optional token-based access override.

📄 admin/routes/add.ts

Purpose: Defines the endpoint for adding a new user to the system.

⚙️ Function: `addUser`

function addUser(addUserFunc: IAdd)

Registers an HTTP handler that creates users via the provided `addUserFunc`, which performs user insertion logic.

📄 admin/routes/get-available-addresses.ts

Purpose: Provides an API handler to return available email addresses for a given name.

⚙️ Function: `availableAddresses`

function availableAddresses(availableAddressesFunc: IAvailableAddressesForName)

Registers an HTTP endpoint to retrieve available addresses based on the provided name handler.

📄 admin/routes/get-available-domains.ts

Purpose: Provides an API handler to return a list of available domains for signup or usage.

⚙️ Function: `availableDomains`

function availableDomains(availableDomainsFunc: IAvailableDomains)

Registers an HTTP endpoint that returns domain options available in the system.

📄 asmail/asmail-app.ts

Purpose: Main application initializer for the ASMail service, handling configuration and route registration.

⚙️ Function: `makeApp`

function makeApp(rootFolder: string, domain: string, midAuthorizer: MidAuthorizer,
errLogger?: ErrLogger)

Initializes the ASMail application using the root directory and domain, along with optional middleware authorization and error logging.

📄 `asmail/config.ts`

Purpose: Sets up application-level configuration used by the ASMail components.

⚙️ Function: `makeApp`

function makeApp(domain: string, sessions: SessionsFactory, recipients:
usersFactory,midAuthorizer: MidAuthorizer)

Constructs an application instance configured with session and recipient logic, along with domain and authorization behavior.

📄 `asmail/delivery.ts`

Purpose: Initializes the delivery-side functionality of ASMail, including message transmission setup.

⚙️ Function: `makeApp`

function makeApp(domain: string, sessions: SessionsFactory, recipients: recipFactory,
midAuthorizer: IMidAuthorizer)

Bootstraps the delivery component for ASMail with handlers for session control, recipient access, and authorization.

📄 `asmail/retrieval.ts`

Purpose: Handles retrieval-side logic of ASMail, providing routes for users to fetch stored messages.

⚙️ Function: `makeApp`

function makeApp(domain: string, sessions: SessionsFactory, recipients: recipFactory,
midAuthorizer: MidAuthorizer)

Bootstraps the retrieval-side application instance, giving access to inbox message fetching and validation routes.

📄 asmail/resources/delivery-sessions.ts

Purpose: Manages the session metadata used in message delivery operations.

🧩 Interface: `SessionParams`

interface SessionParams {
recipient: string;
sender: string|undefined;
invite: string|undefined;
maxMsgLength: number;
currentMsgLength: number;
msgId: string;
}

Defines the parameters required to track an active delivery session, including sender/recipient details and message limits.

📄 asmail/routes/delivery/restart-session.ts

Purpose: Defines the API route used to restart an in-progress message delivery session. This is used when a message object transfer is interrupted or needs to resume. The function ensures session continuity by retrieving prior session details and setting up the environment again.

⚙️ Function: `restartSession`

function restartSession(
sessionGenFunc: GenerateSession,
sessionForMsgFunc: GetSessionForMsg,
incompleteMsgFunc: IncompleteMsgDeliveryParams,
maxChunk: string|number,
redirectFunc?: Redirect

)

Reestablishes a message delivery session by using previously stored session and message metadata. It combines session creation with logic to resume transmission from a partial state. Optionally supports HTTP redirection on completion.

📄 asmail/routes/delivery/sender-authorization.ts

Purpose: Handles authentication and authorization of senders during message delivery. Ensures that the domain initiating the message has the correct credentials and authorization to proceed with delivery.

⚙️ Function: `authorize`

function authorize(
relyingPartyDomain: string,
midAuthorizingFunc: IMidAuthorizer
)

Validates a sender's domain against a middleware-based authorizer function. Intended for use in validating cross-domain or federated message deliveries.

📄 asmail/routes/delivery/start-session.ts

Purpose: Defines the endpoint for initiating a new delivery session. Validates message size limits and sets up the foundational structure required to begin transmitting message content.

⚙️ Function: `startSession`

function startSession(
allowedMsgSizeFunc: AllowedMaxMsgSize,
sessionGenFunc: GenerateSession,
redirectFunc?: Redirect
)

Initializes a new message delivery session by checking max allowed sizes, creating a session ID, and optionally redirecting clients as part of a federated message flow.

📄 asmail/routes/retrieval/get-message-meta.ts

Purpose: Provides metadata about a stored message, such as sender, size, timestamp, or related information. Used for UI display or verification purposes before downloading content.

⚙️ Function: `getMsgMeta`

function getMsgMeta(getMsgMetaFunc: GetMsgMeta)

Retrieves and returns structured metadata for a given message, identified by its message ID. Helps clients prepare for or understand the context of message content.

📄 asmail/routes/retrieval/get-message-obj.ts

Purpose: Enables the actual retrieval of a message object — including its encrypted content, attachments, or associated chunks — by ID.

⚙️ Function: `getMsgObj`

function getMsgObj(getMsgObjFunc: GetObj)

Fetches the full message object from storage for secure client retrieval. Complements `getMsgMeta` by delivering the actual data payload.

📄 asmail/routes/retrieval/list-messages.ts

Purpose: Lists all message IDs associated with a recipient. This is commonly used to display an inbox view or enumerate messages for batch operations.

⚙️ Function: `listMsgIds`

function listMsgIds(listMsgIdsFunc: GetMsgIds)

Returns a list of message identifiers belonging to a recipient, typically used by clients to present message overviews or inboxes.

📄 asmail/routes/retrieval/remove-message.ts

Purpose: Implements secure deletion of a stored message. Ensures that message objects and their metadata are purged from storage when no longer needed.

⚙️ Function: deleteMsg

function deleteMsg(delMsgFunc: DeleteMsg)

Deletes a message object by invoking the provided deletion logic. This function ensures clean removal of both content and associated metadata.

📄 config/from-yaml.ts

Purpose: Loads configuration settings from a YAML file. Used during server startup or runtime configuration parsing.

⚙️ Function: readYamlConfFile

function readYamlConfFile(path: string)

Reads a YAML file at the given path and parses its contents into an in-memory configuration object.

📄 config/letsencrypt.ts

Purpose: Generates TLS/SSL options from Let's Encrypt configuration, supporting dynamic certificate reloading and HTTPS setup.

⚙️ Function: sslOptsFromConfig

function sslOptsFromConfig(
letsencrypt: string|undefined,
sslOpts: ServerOptions|undefined,
skipReloadOnCertsChange: boolean|undefined
)

Creates a TLS configuration object based on Let's Encrypt certs or fallback options. Supports live reload of certificates and optional skipping of reload triggers.

📄 `config/signup.ts`

Purpose: Defines configuration schemas and logic related to the signup process for the system. This may include handling user registration, token validation, and domain restrictions.

📄 `config/cli/index.ts`

Purpose: Acts as the main entry point for CLI-based configuration processing. Typically responsible for parsing and routing command-line arguments to appropriate subcommands.

⚙️ Function: `parseProcessArgv`

function parseProcessArgv()

Parses arguments passed to the CLI tool and dispatches them for processing. This serves as the main initializer for CLI command interpretation.

📄 `config/cli/run-cmd.ts`

Purpose: Processes and interprets CLI arguments specific to general application execution.

🧩 Interface: `ParsedRunArgs`

interface ParsedRunArgs {
config?: string;
help?: boolean;
}

Defines the expected structure of parsed arguments for the main execution command, including support for configuration path and help toggling.

⚙️ Function: `parseRunArgs`

function parseRunArgs(args: string[], execName: string)

Extracts and validates run-specific arguments from the CLI invocation, customizing behavior based on the provided executable name.

📄 `config/cli/signup-cmd.ts`

Purpose: Handles CLI arguments related to user or domain signup operations.

🧩 Interface: `ParsedSignupArgs`

```
interface ParsedSignupArgs {
info?: boolean;
config?: string;
help?: boolean;
listTokens?: boolean;
showToken?: boolean;
token?: string;
createToken?: boolean;
domain?: string[];
user?: string;
}
```

Defines a structured representation of all arguments related to signup management, such as token operations, user assignment, and domain setup.

⚙️ Function: `parseSignupArgs`

```
function parseSignupArgs(args: string[], execName: string)
```

Parses and validates CLI arguments for the signup subcommand, customizing logic based on command context.

📄 `config/cli/user-cmd.ts`

Purpose: CLI interface for user-related administrative actions.

🧩 Interface: `ParsedUserArgs`

```
interface ParsedUserArgs {
listAll?: true;
}
```

Specifies argument options for listing users from the CLI.

⚙️ Function: `parseSignupArgs`

function parseSignupArgs(args: string[], execName: string)

Used here for argument parsing but reused from signup commands, highlighting shared CLI structure.

📄 `config/cli/utils.ts`

Purpose: Provides utility types and rendering logic for displaying CLI usage help, validating arguments, and building consistent argument schemas.

🧩 Interface: `OptionDef`

interface OptionDef {
name: string;
type: (typeof Boolean) | (typeof String) | (typeof Number) | ((value: string) => any);
alias?: string;
description?: string;
typeLabel?: string;
multiple?: boolean;
defaultOption?: boolean;
}

Defines the structure of a single CLI option, including type validation and formatting for help output.

🧩 Interface: `UsageSection`

interface UsageSection {
header?: string;
summary?: string;
optionList?: OptionDef[];
content?: string | UsageSectionContent | (string | UsageSection)[];
}

Represents a section of help documentation shown to the user in CLI tools.

🧩 Interface: `UsageSectionContent`

interface UsageSectionContent {
options: any;
data: any;
}

Provides structured content for `UsageSection`, especially useful for custom rendering engines.

🧩 Interface: `ParseOptions`

```
interface ParseOptions {
argv?: string[];
partial?: boolean;
stopAtFirstUnknown?: boolean;
camelCase?: boolean;
caseInsensitive?: boolean;
}
```

Controls the behavior of the argument parser, such as case handling and partial matching support.

🧩 Interface: `CliUsageDisplay`

```
interface CliUsageDisplay {
txtToDisplay: string;
exitStatus: number;
}
```

Defines how usage or help text should be displayed to the user, including exit codes.

🧩 Interface: `HelpArg`

```
interface HelpArg {
help?: boolean;
}
```

A minimal interface indicating whether help text should be displayed.

⚙️ Function: `toUsageDisplay`

```
function toUsageDisplay(exitStatus: number, usage: UsageSection[])
```

Generates a full usage message for the CLI from structured documentation sections.

⚙️ Function: `toErrorUsage`

```
function toErrorUsage(usage: UsageSection[], errTxt: string)
```

Renders a usage message that includes an error explanation for user feedback.

📄 lib-common/assert.ts

Purpose: Lightweight utility to enforce invariants at runtime.

⚙️ Function: `assert`

function assert(ok: boolean, errMsg?: string)

Throws an error if `ok` is false. Commonly used to enforce assumptions in code during execution. If `errMsg` is provided, it will be used as the error message.

📄 lib-common/async-fs-node.ts

Purpose: A promise-based wrapper around Node.js filesystem methods, supporting both low-level and high-level file operations.

⚙️ File Operations

`readFile`

function readFile(filename: string, options: { encoding: string; flag?: string; })
function readFile(filename: string, options?: { flag?: string; })
function readFile(path: string, options?: { flag?: string; encoding?: string; })

Reads the contents of a file. Supports multiple overloads to specify encoding, flags, or file path types.

`writeFile`

function writeFile(path: string, data: any, options: {
encoding?: string;
mode?: number|string;
flag?: string;
})

Writes data to a file with customizable options like encoding, permission mode, and open flag.

`mkdir`

function mkdir(path: string)

Creates a new directory at the specified path.

`open`

function open(path: string, flags: string)

Opens a file descriptor with specified flags.

`close`

function close(fd: number)

Closes an open file descriptor.

⚙️ Symlinks and Path Resolution

`symlink`

function symlink(target: string, path: string, type?: 'dir'|'file'|'junction')

Creates a symbolic link pointing to `target` from `path`, with an optional link type.

`readlink`

function readlink(path: string)

Resolves and returns the target path of a symbolic link.

⚙️ File and Directory Stats

`lstat`

function lstat(path: string)

Returns information about a symbolic link or file without following the link.

`stat`

function stat(path: string)

Returns information about a file, following symlinks.

`fstat`

function fstat(fd: number)

Returns information about an open file descriptor.

⚙️ Directory Operations

`readdir`

function readdir(path: string)

Reads the contents of a directory, returning a list of file names.

`rmdir`

function rmdir(path: string)

Removes a directory.

`unlink`

function unlink(path: string)

Deletes a file.

`rename`

function rename(oldPath: string, newPath: string)

Renames or moves a file or directory.

⚙️ Truncation and File Size

`truncate`

function truncate(path: string, size: number)

Truncates a file to the specified size.

`ftruncate`

function ftruncate(fd: number, size: number)

Truncates an open file referenced by a file descriptor.

⚙️ Utility

`existsFolderSync`

function existsFolderSync(path: string)

Synchronously checks whether a folder exists.

`copyFile`

function copyFile(src: string, dst: string, overwrite = false, dstMode = '660')

Copies a file from `src` to `dst`. Overwrites only if specified, and sets permissions using `dstMode`.

📄 `lib-common/binding.ts`

Purpose: Placeholder module or stub for future functionality, potentially related to binding native code or system-level operations.

📄 `lib-common/buffer-utils.ts`

Purpose: Provides utilities for handling byte arrays (`Uint8Array`) including encoding, decoding, packing, joining, and converting between buffer representations and JSON.

⚙️ Function: `toBuffer`

function toBuffer(bytes: Uint8Array)

Wraps or clones a `Uint8Array` into a standardized buffer format.

⚙️ Function: `bufFromJson`

function bufFromJson(json: any)

Converts JSON-encoded data (typically base64 or byte-represented arrays) into a `Uint8Array`.

⚙️ Function: `pack`

function pack(bytes: Uint8Array)
function pack(str: string)

Overloaded to accept either a string or a `Uint8Array`. Converts the input into a compact string-encoded representation (typically base64 or URL-safe).

⚙️ Function: `open`

function open(str: string)

function open(bytes: Uint8Array)

Overloaded function that decodes a packed string (or byte buffer) back into its original byte form.

⚙️ Function: `allCharsFromAlphabet`

function allCharsFromAlphabet(str: string)

Checks whether all characters in a string are part of a predefined encoding alphabet (e.g., base64, custom safe charsets).

⚙️ Function: `joinByteArrs`

function joinByteArrs(arrs: Uint8Array[])

Merges multiple `Uint8Array` buffers into one contiguous byte array.

📄 `lib-common/bytes-equal.ts`

Purpose: Fast equality comparison of two `Uint8Array` byte sequences.

⚙️ Function: `bytesEqual`

function bytesEqual(a: Uint8Array, b: Uint8Array)

Performs a byte-by-byte comparison between two buffers to check for strict equality.

📄 `lib-common/canonical-address.ts`

Purpose: Normalizes and verifies user addresses or identifiers to ensure consistent formatting and comparison across the system.

⚙️ Function: `toCanonicalAddress`

function toCanonicalAddress(address: string)

Normalizes the format of an address (e.g., lowercase conversion, character trimming).

⚙️ Function: `areAddressesEqual`

function areAddressesEqual(a: string, b: string)

Determines if two user addresses are equivalent, accounting for canonical formatting.

⚙️ Function: `checkAndTransformAddress`

function checkAndTransformAddress(address: string)

Validates and normalizes an address, throwing an error or rejecting malformed inputs.

📄 `lib-common/json-equal.ts`

Purpose: Provides deep comparison functionality for structured JavaScript objects.

⚙️ Function: `deepEqual`

function deepEqual(a: any, b: any)

Recursively checks whether two JavaScript objects or arrays are deeply equal in content and structure.

📄 `lib-common/json-utils.ts`

Purpose: Additional JSON-related tools, likely extending or aliasing equality checking.

⚙️ Function: `deepEqual`

function deepEqual(a: any, b: any)

Re-exports or mirrors deep equality checking for structured JSON.

📄 `lib-common/jwkeys.ts`

Purpose: Defines types and utility functions for managing cryptographic keys and signatures, including validation, conversion, and access to certified public key structures. This module is critical for safely handling keys used in secure communication, especially in JSON Web Key (JWK)-like format.

🧩 Interface: `JsonKeyShort`

interface JsonKeyShort {
k: string;
kid: string;
}

Represents a minimal JSON-style public key. The `k` field holds the base64-encoded key bytes, and `kid` is a string identifier.

🧩 Interface: `Key`

interface Key {
k: Uint8Array;
kid: string;
use: string;
alg: string;
}

Defines a full cryptographic key object with byte-level data (`k`), a key ID (`kid`), intended application use (`use`), and the crypto box function or suite used (`alg`). Emphasizes complete cryptographic functionality rather than primitives.

🧩 Interface: `SignedLoad`

interface SignedLoad {
alg: string;
kid: string;
sig: string;
load: string;
}

Describes a signed payload structure, including the algorithm used, key identifier, base64-encoded signature, and the signed content in base64.

🧩 Interface: `KeyCert`

interface KeyCert {

cert: {
publicKey: JsonKey;
principal: {
address: string;
};
};
}

Represents a certificate that ties a JSON-formatted public key to an identity (such as an email address or user address).

🧩 Interface: `MailerIdAssertionLoad`

interface MailerIdAssertionLoad {
user: string;
rpDomain: string;
sessionId: string;
issuedAt: number;
expiresAt: number;
}

Defines a signed data structure used for asserting identity in a mailer session. Includes timestamps for issuance and expiration.

⚙️ Function: `isLikeJsonKey`

function isLikeJsonKey(jkey: JsonKey)

Type guard that verifies whether a given object conforms to the `JsonKey` structure.

⚙️ Function: `isLikeSignedLoad`

function isLikeSignedLoad(load: SignedLoad)

Type guard to ensure that an object looks like a valid `SignedLoad` payload.

⚙️ Function: `isLikeKeyCert`

function isLikeKeyCert(cert: KeyCert)

Type guard for validating whether an object matches the `KeyCert` structure.

⚙️ Function: `isLikeSignedKeyCert`

function isLikeSignedKeyCert(load: SignedLoad)

Checks if a `SignedLoad` object can be interpreted as a signed key certificate.

⚙️ Function: `keyFromJson`

function keyFromJson(key: JsonKey, use: string, alg: string, klen: number)

Creates a `Key` object from its JSON representation, specifying usage context and expected algorithm.

⚙️ Function: `keyToJson`

function keyToJson(key: Key)

Converts a `Key` object into a `JsonKeyShort` JSON-serializable format.

⚙️ Function: `getKeyCert`

function getKeyCert(signedCert: SignedLoad)

Extracts a verified `KeyCert` from a signed payload.

⚙️ Function: `getPubKey`

function getPubKey(signedCert: SignedLoad)

Parses and returns the public key from a signed key certificate payload.

⚙️ Function: `getPrincipalAddress`

function getPrincipalAddress(signedCert: SignedLoad)

Extracts the principal's address (identity info) from a signed certificate.

⚙️ Function: `isLikeMailerIdAssertion`

function isLikeMailerIdAssertion(assertLoad: MailerIdAssertionLoad)

Type guard that checks if a payload matches the structure of a MailerId assertion.

⚙️ Function: `isLikeSignedMailerIdAssertion`

function isLikeSignedMailerIdAssertion(load: SignedLoad)

Confirms whether a `SignedLoad` matches the expected structure of a signed MailerIdentity assertion.

📄 `lib-common/mid-sigs-NaCl-Ed.ts`

Purpose: Provides utility functions and interfaces for signing, verifying, and managing certificate chains and assertions using NaCl (Ed25519) cryptography. This is used to secure identity assertions and public key certificates within the system.

🧩 Interface: `Keypair`

interface Keypair {
pkey: JsonKey;
skey: Key;
}

Represents a public/private keypair, with the public key (`pkey`) in a JSON-safe format and the secret key (`skey`) in binary.

🧩 Interface: `IdProviderCertifier`

interface IdProviderCertifier {
certify(publicKey: JsonKey, address: string, validFor?: number): SignedLoad;
destroy(): void;

}

Describes an entity that can generate certificates for given public keys and destroy its own internal signing key when no longer needed.

🧩 Interface: `AssertionLoad`

interface AssertionLoad {
user: string;
rpDomain: string;
sessionId: string;
issuedAt: number;
expiresAt: number;
}

Represents the content of an identity assertion used in secure communication or authentication flows.

🧩 Interface: `CertsChain`

interface CertsChain {
user: SignedLoad;
prov: SignedLoad;
root: SignedLoad;
}

Represents the complete certificate chain for a user, including the user's certificate, provider's certificate, and the root authority's certificate.

🧩 Interface: `AssertionInfo`

interface AssertionInfo {
relyingPartyDomain: string;
sessionId: string;
user: string;
}

Contains identifying information extracted from an identity assertion, typically used for authorization or session validation.

🧩 Interface: `MailerIdSigner`

```
interface MailerIdSigner {
address: string;
userCert: SignedLoad;
providerCert: SignedLoad;
issuer: string;
certExpiresAt: number;
validityPeriod: number;
generateAssertionFor(rpDomain: string, sessionId: string, validFor?: number): SignedLoad;
certifyPublicKey(pkey: JsonKey, validFor: number): SignedLoad;
destroy(): void;
}
```

Manages signing responsibilities for a mailer's identity. It can generate signed assertions and sign public keys for recipients, and revoke its signing capability.

⚙️ Function: `makeMalformedCertsException`

```
function makeMalformedCertsException(msg: string, cause?: any)
```

Creates a structured error when certificates are improperly formatted or incomplete.

⚙️ Function: `makeSelfSignedCert`

```
function makeSelfSignedCert(address: string, validityPeriod: number,
                            sjkey: JsonKey, arrFactory?: arrays.Factory)
```

Generates a certificate signed with its own key—used for root authority or local development.

⚙️ Function: `generateRootKey`

```
function generateRootKey(address: string, validityPeriod: number,
                         random: GetRandom, arrFactory?: arrays.Factory)
```

Creates a keypair and self-signed root certificate to serve as the trust anchor for a certificate chain.

⚙️ Function: `generateProviderKey`

function generateProviderKey(address: string, validityPeriod: number,
rootJKey: JsonKey, random: GetRandom,
arrFactory?: arrays.Factory)

Generates a provider's certificate signed by the root key.

⚙️ Function: `makeIdProviderCertifier`

function makeIdProviderCertifier(issuer: string,
validityPeriod: number, signJKey: JsonKey,
arrFactory?: arrays.Factory)

Creates a certifier object capable of signing public keys with a provided signing key.

⚙️ Function: `verifyChainAndGetUserKey`

function verifyChainAndGetUserKey(certs: CertsChain,
rootAddr: string, validAt: number, arrFactory?: arrays.Factory)

Verifies a full certificate chain and returns the user’s public key if valid.

⚙️ Function: `verifyAssertion`

function verifyAssertion(midAssertion: SignedLoad,
certChain: CertsChain, rootAddr: string,
validAt: number, arrFactory?: arrays.Factory)

Validates a signed identity assertion against a certificate chain and a trusted root authority.

⚙️ Function: `verifyKeyCert`

function verifyKeyCert(keyCert: SignedLoad,
principalAddress: string, signingKey: Key, validAt: number,
arrFactory?: arrays.Factory)

Confirms that a key certificate was properly signed by the expected key and is currently valid.

⚙️ Function: `verifyPubKey`

```
function verifyPubKey(pubKeyCert: SignedLoad,
                      principalAddress: string, certChain: CertsChain, rootAddr: string,
                      validAt: number, arrFactory?: arrays.Factory)
```

Verifies the authenticity of a public key against a valid certificate chain and root authority.

⚙️ Function: `generateSigningKeyPair`

```
function generateSigningKeyPair(random: GetRandom,
                      arrFactory?: arrays.Factory)
```

Generates a new signing keypair used in identity and certificate signing processes.

⚙️ Function: `makeMailerIdSigner`

```
function makeMailerIdSigner(signKey: Key,
                      userCert: SignedLoad, provCert: SignedLoad,
                      assertionValidity = user.MAX_SIG_VALIDITY,
                      arrFactory?: arrays.Factory)
```

Creates a signer that can issue identity assertions and certificates based on user and provider credentials.

📄 `lib-common/processes.ts`

Purpose: Provides utility functions for asynchronous locking and scheduling. Enables read/write locking semantics for concurrency control in environments such as multi-threaded or async workflows.

🧩 Interface: `ReadWriteLock`

```
interface ReadWriteLock {
lockForRead(): Promise<() => void>;
lockForWrite(): Promise<() => void>;
}
```

Describes a concurrency lock that distinguishes between read and write access, ensuring fair scheduling and mutual exclusion where needed.

⚙️ Function: `sleep`

function sleep(millis: number)

Delays execution for a specified number of milliseconds. Returns a promise.

⚙️ Function: `makeLock`

function makeLock()

Creates a basic asynchronous lock. Only one caller can hold the lock at a time.

⚙️ Function: `makeReadWriteLock`

function makeReadWriteLock()

Generates a `ReadWriteLock` allowing concurrent reads and exclusive writes, with fair queuing logic.

📄 `lib-common/random-node.ts`

Purpose: Cryptographic randomness utilities, built on Node.js APIs. Used for generating secure random values and encoded representations.

⚙️ Function: `bytesSync`

function bytesSync(numOfBytes: number)

Synchronously generates a random byte array of the specified length.

⚙️ Function: `bytes`

function bytes(numOfBytes: number)

Asynchronously generates a random byte array.

⚙️ Function: `uint8Sync`

function uint8Sync()

Generates a single random byte.

⚙️ Function: `uint48Sync`

function uint48Sync()

Generates a 48-bit random integer.

⚙️ Function: `stringOfB64UrlSafeCharsSync`

function stringOfB64UrlSafeCharsSync(numOfChars: number)

Generates a base64 URL-safe random string.

⚙️ Function: `stringOfB64CharsSync`

function stringOfB64CharsSync(numOfChars: number)

Generates a standard base64-encoded random string.

📄 `lib-common/session-encryptor.ts`

Purpose: Provides encryption/decryption utilities for handling secure sessions using symmetric cryptography.

⚙️ Function: `makeSessionEncryptor`

function makeSessionEncryptor(key: Uint8Array, nextNonce: Uint8Array)

Creates an encryptor instance initialized with a shared secret and nonce seed. Enables encoding and decoding of session content.

📄 `lib-common/admin-api/signup.ts`

Purpose: Defines administrative API types related to the signup process for users and domains.

🧩 Interface: `Request`

interface Request {
name: string;
signupToken?: string;
}

Used for signup requests that include an optional token and a display name.

interface Request {
signupToken?: string;

}

A minimal form of the signup request used in other admin contexts.

interface Request {
userId: string;
mailerId: UserMidParams;
storage: UserStorageParams;
signupToken?: string;
}

Signup request with full mid and storage initialization parameters for creating new users.

interface Request {
userId: string;
}

Simplified version used for user-specific actions where only a user ID is required.

🧩 Interface: `UserMidParams`

interface UserMidParams {
defaultPKey: {
pkey: JsonKey;
kdParams: any;
}
}

Represents mid-tier cryptographic parameters used during user registration.

🧩 Interface: `UserStorageParams`

interface UserStorageParams {
kdParams: any;
}

Specifies the encryption parameters for the user's storage configuration.

🧩 Interface: `ErrorReply`

interface ErrorReply {
error: string;
}

Standard format for reporting API errors.

📄 lib-common/byte-streaming/common.ts

Purpose: Provides utility wrappers for byte source and sink implementations used in stream processing.

⚙️ Function: `wrapByteSourceImplementation`

function wrapByteSourceImplementation(src: ByteSource)

Wraps a low-level byte source implementation to ensure it conforms to a standardized streaming interface.

⚙️ Function: `wrapByteSinkImplementation`

function wrapByteSinkImplementation(sink: ByteSink)

Wraps a byte sink to standardize write operations for stream data consumers.

⚙️ Function: `sourceFromArray`

function sourceFromArray(bytes: Uint8Array)

Creates a byte source from a static byte array, allowing it to be consumed as a stream.

📄 lib-common/exceptions/error.ts

Purpose: Error composition utilities for attaching cause chains and readable string formatting.

⚙️ Function: `errWithCause`

function errWithCause(cause: any, message: string)

Attaches a causal error to a primary message, producing a richer exception object.

⚙️ Function: `stringifyErr`

function stringifyErr(err: any)

Converts an error object (including nested causes) to a readable string.

📄 lib-common/exceptions/file.ts

Purpose: Specialized error handling for filesystem-related operations.

⚙️ Function: `makeFileException`

function makeFileException(code: string|undefined, path: string, cause?: any)

Creates an exception tied to a file operation, including error code and root cause.

⚙️ Function: `maskPathInExc`

function maskPathInExc(pathPrefixMaskLen: number, exc: any)

Masks sensitive or absolute path information in exception messages.

⚙️ Function: `ensureCorrectFS`

function ensureCorrectFS(fs: web3n.files.FS, type: web3n.files.FSType, writable: boolean)

Validates that a given filesystem instance matches the expected type and writability.

📄 `lib-common/exceptions/http.ts`

Purpose: Provides exception factories for HTTP communication errors.

⚙️ Function: `makeConnectionException`

function makeConnectionException(url: string|undefined, method: string|undefined, msg?: string, cause?: any)

Builds a detailed exception for failures in establishing or maintaining HTTP connections.

⚙️ Function: `makeHTTPException`

function makeHTTPException(url: string, method: string, status: number, msg?: string, cause?: any)

Creates a structured error from an HTTP response failure, including status code and diagnostics.

📄 `lib-common/ipc/generic-ipc.ts`

Purpose: Implements IPC communication primitives over generic duplex channels.

⚙️ Function: `makeRequestingClient`

function makeRequestingClient(channel: string|undefined, comm: RawDuplex<Envelope>)

Creates an IPC client for making one-off requests over a bidirectional channel.

⚙️ Function: `makeRequestServer`

function makeRequestServer(channel: string|undefined, comm: RawDuplex<Envelope>)

Implements an IPC server that handles request/response message pairs.

⚙️ Function: `makeEventfulServer`

function makeEventfulServer(channel: string|undefined, comm: RawDuplex<Envelope>)

Enhances an IPC server with the ability to publish unsolicited events.

⚙️ Function: `makeSubscribingClient`

function makeSubscribingClient(channel: string|undefined, comm: RawDuplex<Envelope>)

Builds an IPC client that subscribes to and processes event streams.

📄 `lib-common/ipc/ws-ipc.ts`

Purpose: Provides WebSocket-based transport for JSON-based IPC communication.

⚙️ Function: `makeJsonCommPoint`

function makeJsonCommPoint(ws: WebSocket)

Turns a WebSocket into a JSON message communication point.

⚙️ Function: `makeSubscriber`

function makeSubscriber(ws: WebSocket, ipcChannel: string|undefined)

Creates a subscriber client over WebSocket for receiving event messages.

📄 `lib-common/objs-on-disk/file-layout.ts`

Purpose: Manages low-level structure of encrypted, versioned objects stored on disk.

🧩 Interface: `HeaderChunkInfo`

```
interface HeaderChunkInfo {
len: number;
fileOfs: number;
}
```

Describes the position and length of an object's header in its file representation.

🧩 Interface: `FiniteChunk`

```
interface FiniteChunk {
thisVerOfs: number;
len: number;
}
```

Represents a bounded segment of an object version stored in a file.

🧩 Interface: `NewEndlessSegsChunk`

```
interface NewEndlessSegsChunk {
type: 'new-endless';
thisVerOfs: number;
}
```

Describes a growing object segment—useful for append-only storage formats.

📄 `lib-common/objs-on-disk/utils.ts`

Purpose: Provides utility types and functions to process and construct chunked file layouts for object versioning on disk. These tools are crucial for interpreting storage structure and facilitating stream-based read/write operations.

🧩 Interface: `ChunkWithHeader`

```
interface ChunkWithHeader {
type: 'header';
len: number;
}
```

Represents a chunk containing the header portion of an object. Includes its length for processing in streaming layouts.

🧩 Interface: `ChunkWithSegs`

```
interface ChunkWithSegs {
type: 'segs';
len: number;
segsOfs: number;
}
```

Represents a chunk containing the segments portion of an object. Includes length and offset to locate segment data.

⚙️ Function: `diffToLayout`

```
function diffToLayout(diff: DiffInfo)
```

Computes a proposed layout of file chunks based on the given diff. Used to calculate how object changes will appear when written to disk.

⚙️ Function: `chunksInOrderedStream`

```
function chunksInOrderedStream(len: number, headerLen?: number, segsOfs: number)
```

Generates an ordered list of stream chunks based on total length, optional header length, and segment offset. Ensures a correct sequence of reading/writing structured data.

⚙️ Function: `makeObjPipe`

```
function makeObjPipe(file: ObjVersionFile, header: boolean, segsOfs: number, segsLen:
number,
        objId: string | null, getObjFile: GetObjFile)
```

Creates a stream-based pipeline to read or write an object version from disk, including header and segment processing logic. Can optionally reference a base object by ID.

⚙️ Function: `makeNoBaseObjPipe`

function makeNoBaseObjPipe(file: ObjVersionFile, header: boolean, segsOfs: number, segsLen: number)

Creates a simplified object I/O pipeline that excludes base object handling. Used for writing self-contained object versions.

📄 `lib-common/objs-on-disk/utils.ts`

Purpose: Provides utility types and functions to process and construct chunked file layouts for object versioning on disk. These tools are crucial for interpreting storage structure and facilitating stream-based read/write operations.

🧩 Interface: `ChunkWithHeader`

interface ChunkWithHeader {
type: 'header';
len: number;
}

Represents a chunk containing the header portion of an object. Includes its length for processing in streaming layouts.

🧩 Interface: `ChunkWithSegs`

interface ChunkWithSegs {
type: 'segs';
len: number;
segsOfs: number;
}

Represents a chunk containing the segments portion of an object. Includes length and offset to locate segment data.

⚙️ Function: `diffToLayout`

function diffToLayout(diff: DiffInfo)

Computes a proposed layout of file chunks based on the given diff. Used to calculate how object changes will appear when written to disk.

⚙️ Function: `chunksInOrderedStream`

function chunksInOrderedStream(len: number, headerLen?: number, segsOfs: number)

Generates an ordered list of stream chunks based on total length, optional header length, and segment offset. Ensures a correct sequence of reading/writing structured data.

⚙️ Function: `makeObjPipe`

function makeObjPipe(file: ObjVersionFile, header: boolean, segsOfs: number, segsLen: number,
    objId: string | null, getObjFile: GetObjFile)

Creates a stream-based pipeline to read or write an object version from disk, including header and segment processing logic. Can optionally reference a base object by ID.

⚙️ Function: `makeNoBaseObjPipe`

function makeNoBaseObjPipe(file: ObjVersionFile, header: boolean, segsOfs: number, segsLen: number)

Creates a simplified object I/O pipeline that excludes base object handling. Used for writing self-contained object versions.

`lib-common/objs-on-disk/v1-obj-file-format.ts`

Purpose: Implements the serialization logic for a versioned object file format. This module manages the conversion of storage metadata and file segment data into and from byte representations for encrypted, version-controlled disk storage.

🧩 Interface: `Attrs`

interface Attrs {
fileComplete: boolean;
segsChunks: SegsChunk[];
headerChunk?: HeaderChunkInfo;
segsLayoutFrozen: boolean;
baseVersion?: number;
sizeUnknown: boolean;
allBaseBytesInFile: boolean;
}

Describes the overall attributes and metadata of a stored object version. Tracks completeness, segmentation, header info, and layout states.

⚙️ Function: `uintTo8Bytes`

function uintTo8Bytes(u: number)

Converts a 64-bit unsigned integer into an 8-byte `Uint8Array`, used for serialization of numeric values.

⚙️ Function: `uintFrom8Bytes`

function uintFrom8Bytes(x: Uint8Array, i = 0)

Parses an 8-byte sequence from a `Uint8Array` to reconstruct a 64-bit unsigned integer. Defaults to starting at index `0`.

⚙️ Function: `toBytes` (Header)

function toBytes(hInfo: HeaderChunkInfo)

Serializes a `HeaderChunkInfo` object into bytes, used to persist header chunk metadata.

⚙️ Function: `fromBytes` (Header)

function fromBytes(b: Uint8Array, i: number)

Deserializes a header structure from bytes, reconstructing `HeaderChunkInfo` from an offset in the byte array.

⚙️ Function: `toBytes` (Segment)

function toBytes(sInfo: SegsChunk)

Serializes a `SegsChunk` (file segment descriptor) into bytes for on-disk storage.

⚙️ Function: `fromBytes` (Segment)

function fromBytes(b: Uint8Array, i: number)

Deserializes segment chunk information from a byte buffer, returning a `SegsChunk`.

⚙️ Function: `toBytes` (Attrs)

function toBytes(a: Attrs)

Serializes the full `Attrs` object (describing a file version) into a binary format for storage.

⚙️ Function: `fromBytes` (Attrs)

function fromBytes(b: Uint8Array, i: number)

Reconstructs an `Attrs` structure from a byte array, parsing file metadata and chunk layout info.

📄 `lib-common/service-api/pub-key-login.ts`

Purpose: Defines TypeScript interfaces used for secure authentication using public-key login over an API. These structures represent request and response contracts in the session-establishment process.

🧩 Interface: `Request`

interface Request {
userId: string;

kid?: string;
}

Represents a request to authenticate a user by public key. The `userId` is mandatory, while `kid` (Key ID) is optional and may help identify which specific key the server should verify.

🧩 Interface: `Reply`

interface Reply {
sessionId: string;
sessionKey: string;
serverPubKey: string;
keyDerivParams: any;
}

Response returned upon successful authentication. It includes session identifiers and key derivation parameters for the client to initialize a secure session.

🧩 Interface: `RedirectReply`

interface RedirectReply {
redirect: string;
}

Used to instruct the client to redirect (e.g., to a login flow or alternate service location) instead of directly receiving a session.

🧩 Interface: `ErrorReply`

interface ErrorReply {
error: string;
}

Conveys a failure message if the authentication or session establishment fails.

📄 `lib-common/service-api/web-socket-base.ts`

Purpose: Defines utility functions for formatting messages in a WebSocket-based subscription system. These functions handle request and response formatting as well as error reporting for channel-based interactions.

⚙️ Function: `subscriptionRequest`

function subscriptionRequest(act: 'subscribe'|'unsubscribe',
channel: string, reqCount: number)

Creates a structured WebSocket message to either subscribe to or unsubscribe from a specific data channel. The `reqCount` is a request ID for tracking responses.

⚙️ Function: `subscriptionReply`

function subscriptionReply(reqCount: number, status: string)

Generates a reply message for a subscription request, indicating success or failure. Includes the original request count for correlation.

⚙️ Function: `errorReplyTo`

function errorReplyTo(req: RequestEnvelope<any>, err: any)

Forms an error reply in response to a previously received request. It captures the error and ties it back to the original request envelope.

📄 `lib-common/service-api/3nstorage/owner.ts`

Purpose: Defines types and helper functions for interacting with the 3NStorage API on the owner side. This includes object versioning, segmentation, diff handling, and structured request/response formats.

🧩 Interface: `Reply`

interface Reply {
maxChunkSize: number;

}

Represents the maximum allowed chunk size for data transfers.

🧩 Interface: `GetObjQueryOpts`

```
interface GetObjQueryOpts {
header?: boolean;
ofs?: number;
limit?: number;
ver?: number;
}
```

Options for fetching objects, including whether to include headers, segment offset and limit, and version constraints.

🧩 Interface: `PutObjFirstQueryOpts`

```
interface PutObjFirstQueryOpts {
ver: number;
diff?: number;
header: number;
last?: boolean;
}
```

Query parameters for the first object PUT request, specifying the new version, optional diff/header lengths, and transaction termination.

🧩 Interface: `PutObjSecondQueryOpts`

```
interface PutObjSecondQueryOpts {
trans: string;
ofs: number;
last?: boolean;
}
```

Query parameters for subsequent object PUT requests, including transaction ID, segment offset, and optional closing flag.

🧩 Interface: `ReplyToPut`

```
interface ReplyToPut {
transactionId?: string;
}
```

Response payload for PUT operations, includes a transaction ID if further segments are expected.

🧩 Interface: `DiffInfo`

```
interface DiffInfo {
baseVersion: number;
segsSize: number;
sections: [0 | 1, number, number][];
}
```

Describes a binary diff between object versions, including segment size and byte-segment modifications.

🧩 Interface: `ObjStatus`

```
interface ObjStatus {
current?: number;
archived?: number[];
}
```

Reports the current and archived versions of an object.

🧩 Interface: `ErrorReply`

```
interface ErrorReply {
error: string;
}
```

Standard structure for error replies from the API.

🧩 Interface: `Event` (Variants)

Different `Event` interfaces reflect specific object lifecycle events:

1. **New version created**

```
interface Event {
objId: string | null;
newVer: number;
}
```

2. **Object removed**

```
interface Event {
objId: string;
}
```

3. **Version archived**

```
interface Event {
objId: string | null;
archivedVer: number;
}
```

4. **Archived version removed**

```
interface Event {
objId: string | null;
archivedVer: number;
}
```

⚙️ Function: `getReqUrlEnd`

```
function getReqUrlEnd(objId: string, opts?: GetObjQueryOpts)
function getReqUrlEnd(transactionId?: string)
function getReqUrlEnd(objId: string)
function getReqUrlEnd()
```

Generates a GET URL suffix for object or transaction-based requests, overloaded to support flexible combinations.

⚙️ Function: `delReqUrlEnd`

```
function delReqUrlEnd(objId: string, ver?: number)
```

Constructs a DELETE URL endpoint for object version deletion.

⚙️ Function: `firstPutReqUrlEnd`

function firstPutReqUrlEnd(objId: string, opts: PutObjFirstQueryOpts)
function firstPutReqUrlEnd(opts: PutObjFirstQueryOpts)

Generates a PUT request endpoint for the first part of a segmented object upload.

⚙️ Function: `secondPutReqUrlEnd`

function secondPutReqUrlEnd(objId: string, opts: PutObjSecondQueryOpts)
function secondPutReqUrlEnd(opts: PutObjSecondQueryOpts)

Forms the URL for uploading subsequent segments of an object in a transaction.

⚙️ Function: `sanitizedDiff`

function sanitizedDiff(diff: DiffInfo, version: number)

Processes a diff object to produce a cleaned, structured representation for serialization or transmission.

⚙️ Function: `addDiffSectionTo`

function addDiffSectionTo(sections: number[][], newBytes: boolean, srcPos: number, len: number)

Appends a diff segment to the `sections` array, describing whether it's new or based on prior content.

⚙️ Function: `postAndDelReqUrlEnd`

function postAndDelReqUrlEnd(objId: string, version: number)
function postAndDelReqUrlEnd(version: number)

Generates POST or DELETE endpoints targeting a specific object version. Flexible overloads support use with or without object ID.

📄 lib-common/service-api/asmail/config.ts

Purpose: Defines configuration structures for recipient-side policies, cryptographic credentials, and invitation management used during secure mail exchange.

🧩 Interface: `InvitesList`

```
interface InvitesList {
[invite: string]: number;
}
```

Maps invite tokens to expiration times or usage counts.

🧩 Interface: `Certs`

```
interface Certs {
pkeyCert: jwk.SignedLoad;
userCert: jwk.SignedLoad;
provCert: jwk.SignedLoad;
}
```

Represents a bundle of signed certificates used for verification and identity in secure messaging.

🧩 Interface: `Policy` (variant 1)

```
interface Policy {
acceptWithInvitesOnly: boolean;
acceptFromWhiteListOnly: boolean;
applyBlackList: boolean;
defaultMsgSize: number;
}
```

Defines message acceptance rules and size limits on a recipient.

🧩 Interface: `Policy` (variant 2)

```
interface Policy {
accept: boolean;
acceptWithInvitesOnly: boolean;
defaultMsgSize: number;
}
```

Alternate shape of the same concept, used in a different configuration context.

🧩 Interface: `List`

interface List {
[address: string]: number;
}

Used for whitelist/blacklist address mapping, with optional numeric flags.

🧩 Interface: `ErrorReply`

interface ErrorReply {
error: string;
}

Represents an error response from the service.

📄 `lib-common/service-api/asmail/delivery.ts`

Purpose: Defines request and reply formats, including encryption metadata, used during message delivery.

🧩 Interfaces

- `Reply`: Describes max message or chunk sizes in response.
- `RedirectReply`: Contains redirect instruction.
- `Request` (3 variants):
- Initial delivery metadata (recipient, sender, invitation).
- Delivery content query (recipient, msgId).
- Delivery authentication (assertion and certs).
- `CryptoInfo`: Holds optional cryptographic identifiers (e.g., sender key).
- `PutObjFirstQueryOpts` / `PutObjSecondQueryOpts`: Streaming metadata used for object chunking.
- `ErrorReply`: Generic error wrapper.

⚙️ Functions

- `firstPutReqUrlEnd(objId, opts)`: Constructs the URL suffix for the first object PUT.
- `secondPutReqUrlEnd(objId, opts)`: Constructs the URL suffix for a follow-up object PUT.

📄 `lib-common/service-api/asmail/retrieval.ts`

Purpose: Defines object retrieval structures and helpers for handling secure message access and metadata.

🧩 Interfaces

- `ObjSize`: Number of bytes in object header and segments.
- `ObjStatus`: Includes completeness state and associated size.
- `MsgMeta`: Metadata about a message, including delivery timeline, sender, and object statuses.
- `GetObjQueryOpts`: Optional query flags for object part retrieval.
- `ErrorReply`: Common error reply format.
- `Event`: Basic structure for identifying message-related events.

⚙️ Functions

- `genUrlEnd(msgId: string)`: Generates retrieval endpoint for a message.
- `genUrlEnd(msgId: string, objId: string, opts?)`: Extended endpoint generator for specific object content.
- `sanitizedMeta(meta: MsgMeta)`: Redacts or transforms metadata for public-facing responses.

📄 `lib-common/service-api/mailer-id/login.ts`

Purpose: Defines the request and response interfaces for secure login using cryptographic identity in the MailerId authentication flow.

🧩 Interface: `Request`

```
interface Request {
userId: string;
}
```

Represents a basic login request containing only a user identifier.

🧩 Interface: `Reply`

```
interface Reply {
```

sessionId: string;
}

Contains the resulting session ID after a successful login.

🧩 Interface: `RedirectReply`

interface RedirectReply {
redirect: string;
}

Indicates that the login response should redirect the client to a different endpoint (e.g., for completing auth).

🧩 Interface: `Request`

interface Request {
assertion: jwk.SignedLoad;
userCert: jwk.SignedLoad;
provCert: jwk.SignedLoad;
}

Describes a login attempt using cryptographic credentials. Includes a signed assertion and associated certificates for the user and provisioner.

🧩 Interface: `ErrorReply`

interface ErrorReply {
error: string;
}

Encapsulates error information returned during a failed login attempt.

📄 `lib-common/service-api/mailer-id/provisioning.ts`

Purpose: Defines the API interfaces used for provisioning and retrieving MailerId certificates.

🧩 Interface: `Request`

interface Request {
pkey: JsonKey;
duration: number;
}

Specifies the key and validity duration for which a new certificate is requested.

🧩 Interface: `Reply`

interface Reply {
userCert: SignedLoad;
provCert: SignedLoad;
}

Contains the newly provisioned user certificate along with the provisioner's certificate.

🧩 Interface: `Reply`

interface Reply {
"current-cert": SignedLoad;
"previous-certs": SignedLoad[];
provisioning: string;
}

Represents a provisioning history response, detailing the currently active certificate, any previous ones, and the provisioning source.

📄 `lib-server/conf-util.ts`

Purpose: Utility for parsing human-readable size strings into byte counts.

⚙️ Function: `stringToNumOfBytes`

function stringToNumOfBytes(size: string|number)

Converts string values like `"2MB"` or `"500kB"` into numerical byte counts for configuration parsing.

📄 `lib-server/middleware/body-parsers.ts`

Purpose: Defines HTTP request body parsers and utilities for safely handling body data in Express apps.

⚙️ Function: `attachByteDrainToRequest`

function attachByteDrainToRequest(req: express.Request)

Attaches logic to drain unused request body data, preventing incomplete stream reads.

⚙️ Function: `emptyBody`

function emptyBody()

Returns middleware that ensures incoming requests have no body content.

⚙️ Function: `binary`

function binary(maxSize: string|number)

Middleware that parses binary payloads up to a specified size limit.

⚙️ Function: `json`

function json(maxSize: string|number, allowNonObject?: boolean)

Parses JSON payloads with an optional flag to allow primitives like strings or numbers as top-level values.

⚙️ Function: `textPlain`

function textPlain(maxSize: string|number)

Parses plain text request bodies up to a defined maximum size.

📄 lib-server/middleware/error-handler.ts

Purpose: Provides centralized error handling and logging logic for Express servers.

⚙️ Function: makeErr

function makeErr(code: number, msg: string, cause?: any)

Constructs an HTTP error object with optional cause information.

⚙️ Function: makeErrHandler

function makeErrHandler(log?: ErrLogger)

Creates middleware to catch and process thrown errors using an optional logging function.

⚙️ Function: makeErrLoggerToConsole

function makeErrLoggerToConsole(srvName: string)

Produces a logger that outputs formatted error messages to the console for a given service.

📄 lib-server/resources/dns.ts

Purpose: Handles DNS-related API endpoints or utility logic for verifying or managing DNS records.

📄 lib-server/resources/mailerid-authorizer.ts

Purpose: Implements logic for verifying and validating MailerId-based authorization tokens or credentials.

⚙️ Function: `validator`

function validator()

Creates a validation handler, likely used to check the authenticity and structure of incoming authorization data.

📄 `lib-server/resources/mem-backed-sessions-factory.ts`

Purpose: Implements a session management factory that stores session data in memory.

📄 `lib-server/resources/server-data-folders.ts`

Purpose: Provides utilities and structures to define and manage folder hierarchies used for storing server data related to users, tokens, and defaults.

🧩 Interface: `FolderDef`

interface FolderDef {
path: string;
}

Represents a directory on disk by its path. Used as the base structure for all folder tree definitions.

🧩 Interface: `RootDataTree`

interface RootDataTree {
tokens: FolderDef;
users: FolderDef;
defaults: FolderDef;
}

Describes the top-level folder layout for server-side data, including paths to token storage, user data, and default configurations.

🧩 Interface: `DefaultsDataTree`

interface DefaultsDataTree {
info: FolderDef;
store: FolderDef & {
params: FolderDef;
};
}

Describes the folder structure for default system-wide configuration, including general info and storage-related parameters.

🧩 Interface: `UserDataTree`

interface UserDataTree {
mail: FolderDef & {
messages: FolderDef;
delivery: FolderDef;
params: FolderDef;
};
}

Represents the per-user folder layout, focused on mail-related storage including messages, delivery metadata, and parameters.

⚙️ Function: `addressToFName`

function addressToFName(address: string)

Generates a filename-safe string derived from a user's address.

⚙️ Function: `treeInRootFolder`

function treeInRootFolder(rootFolder: string)

Creates a `RootDataTree` structure within a specified root folder.

⚙️ Function: `tokensInRootFolder`

function tokensInRootFolder(rootFolder: string)

Returns the path to the tokens folder inside a root directory.

⚙️ Function: `userDataInRootFolder`

function userDataInRootFolder(root: string, userId: string)

Generates a `UserDataTree` structure for a given user ID based on a root directory.

⚙️ Function: `treeInDefaultsFolder`

function treeInDefaultsFolder(defaults: string)

Returns the folder layout for defaults stored within a specific defaults folder path.

⚙️ Function: `treeInUserFolder`

function treeInUserFolder(userFoder: string)

Constructs a `UserDataTree` based on a given path to a user's folder.

⚙️ Function: `createFolderTreeSync`

function createFolderTreeSync(structure: RootDataTree | UserDataTree | DefaultsDataTree | string)

Synchronously creates the full folder structure defined by one of the folder tree interfaces, or a raw string path.

📄 `lib-server/resources/sessions.ts`

Purpose: Defines the minimal session parameter structure needed for user-specific session contexts.

🧩 Interface: `SessionParams`

```
interface SessionParams {
userId: string;
}
```

Basic structure representing a user's session, scoped by their `userId`.

📄 `lib-server/resources/user-files.ts`

Purpose: Provides definitions and utilities for working with user-related files, such as reading objects or managing identity certificates.

🧩 Interface: `ObjReader`

```
interface ObjReader {
len: number;
pipe?: ObjPipe;
headerLen?: number;
segsLen: number;
}
```

Represents a readable object on disk, potentially piped in streaming form. Includes metadata about length and segment layout.

⚙️ Function: `addressToFName`

```
function addressToFName(address: string)
```

Converts a mail address to a safe filename format, used when mapping user identities to storage.

📄 `lib-server/routes/pub-key-login/complete-exchange.ts`

Purpose: Implements the server-side logic for completing a public key–based login exchange.

⚙️ Function: `completePKLogin`

function completePKLogin()

Handles the final step of a public key login flow, such as verifying credentials and establishing a session.

📄 `lib-server/routes/pub-key-login/start-exchange.ts`

Purpose: Defines the public key login initiation route for secure client authentication using a Diffie-Hellman exchange.

🧩 Interface: `UserPKeyAndKeyGenParams`

interface UserPKeyAndKeyGenParams {
pkey: Uint8Array;
params?: any;
}

Describes parameters associated with a user's public key and any additional key-generation-related data needed during login.

⚙️ Function: `startPKLogin`

function startPKLogin(
findUserParamsAndKeyFunc: GetUserPKeyAndKeyGenParams,
sessionGenFunc: GenerateSession<SessionParams>,
computeDHSharedKeyFunc: ComputeDHSharedKey
)

Defines a handler for initiating public key login. It looks up user key parameters, generates a session, and calculates a shared secret using Diffie-Hellman.

📄 `lib-server/routes/sessions/close.ts`

Purpose: Provides an endpoint for securely closing user sessions.

⚙️ Function: `closeSession`

function closeSession()

Defines a route handler that terminates an active session, cleaning up resources or credentials.

📄 `lib-server/routes/sessions/mid-auth.ts`

Purpose: Handles session creation using MID-based authorization for secure identity confirmation.

⚙️ Function: `midLogin`

function midLogin(relyingPartyDomain: string, midAuthorizingFunc: MidAuthorizer)

Sets up a login route using MID (Mobile Identity or Middleware Identity) verification with the specified domain and authorization strategy.

📄 `lib-server/routes/sessions/start.ts`

Purpose: Exposes an endpoint to initiate a new session, verifying the user and redirecting as necessary.

⚙️ Function: `startSession`

function startSession(
userExistsFunc: UserExists,
sessionGenFunc: GenerateSession<any>,
redirectFunc?: Redirect
)

Starts a new session if the user exists. Can optionally redirect after successful session creation.

📄 `lib-server/web-sockets/app.ts`

Purpose: Provides the interface for WebSocket-based inter-process communication with attached IPC channels.

🧩 Interface: `SocketIPCs`

```
interface SocketIPCs {
ipcChannel: string|undefined;
attachIPC(userId: string, ws: WebSocket): void;
}
```

Describes a WebSocket object capable of binding IPC channels to user IDs for communication.

📄 `mailerId/mailerId-app.ts`

Purpose: Application entry point for the MailerID service, initializing services and configuring error logging.

⚙️ Function: `makeApp`

```
function makeApp(
rootFolder: string,
domain: string,
certFile: string,
errLogger?: ErrLogger
)
```

Initializes the MailerID application with the given domain, certificate file, and optional error logger.

📄 `mailerId/resources/certifier.ts`

Purpose: Implements logic for digital certificate management, including current and previous root certificates.

🧩 Interface: `RootCerts`

```
interface RootCerts {
current: SignedLoad;
previous: SignedLoad[];
}
```

Tracks the current and previous root certificates used by the system.

🧩 Interface: `Certifier`

```
interface Certifier {
certify: ICertify;
getRootCert(): SignedLoad;
getPrevCerts(): SignedLoad[];
}
```

Defines an object responsible for issuing, retrieving, and managing digital certificates.

⚙️ Function: `makeSingleProcCertifier`

```
function makeSingleProcCertifier(domain: string, certsPath: string)
```

Creates a single-process certifier tied to the given domain and certificate storage location.

📄 `mailerId/resources/compute-login-dhshared-key.ts`

Purpose: Provides the Diffie-Hellman key calculation logic for secure communication.

⚙️ Function: `calcNaClBoxSharedKey`

```
function calcNaClBoxSharedKey(userPubKey: Uint8Array)
```

Computes a shared secret key using NaCl's box key exchange, based on the user's public key.

📄 `mailerId/resources/sessions.ts`

Purpose: Manages MailerID session lifecycles, including creation and timeouts.

⚙️ Function: `makeSessionFactory`

```
function makeSessionFactory(timeout: number)
```

Initializes a factory that produces session objects with the specified timeout duration.

📄 `mailerId/resources/users.ts`

Purpose: Provides user-related functionality, including public key retrieval for login.

🧩 Interface: `Factory`

```
interface Factory {
getUserParamsAndKey: GetUserPKeyAndKeyGenParams;
}
```

Describes a factory interface for retrieving user parameters necessary for key-based login.

⚙️ Function: `makeFactory`

```
function makeFactory(rootFolder: string)
```

Creates a user data access factory rooted in the specified folder.

📄 `mailerId/routes/certify.ts`

Purpose: Defines the endpoint for issuing digital certificates.

⚙️ Function: `certify`

```
function certify(certifyingFunc: ICertify)
```

Sets up a certificate issuance route using the provided certification logic.

📄 `mock/dns.ts`

Purpose: Mock implementation of DNS TXT record resolution, useful for testing identity verification.

🧩 Interface: `DnsTxtRecords`

```
interface DnsTxtRecords {
[ domain: string ]: string[][];
}
```

Represents the structure of mocked DNS TXT records mapped by domain.

📄 `mock/mock-as-child-proc.ts`

Purpose: Defines utilities to mock modules or services that behave as child processes, useful for simulating system-level or background behaviors in tests.

📄 `mock/node-child-ipc.ts`

Purpose: Provides IPC (inter-process communication) helpers for communicating with mocked child processes in tests.

⚙️ Function: `commToChild`

function commToChild(channel: string, child: cProcs.ChildProcess)

Initializes a communication channel to a child process for structured data exchange using a named channel.

⚙️ Function: `commToParent`

function commToParent(channel: string)

Sets up the parent-side interface to receive messages from a child process over the specified channel.

📄 `mock/run-in-proc.ts`

Purpose: Provides test-specific helpers to simulate full service stack behavior on localhost using predefined DNS and TLS configurations.

⚙️ Function: `setTestCertsAndDNS`

function setTestCertsAndDNS(domains: string[], thisSrvLoc: string)

Configures the testing environment by stubbing certificates and DNS mappings to the given domains.

⚙️ Function: `startOnLocalhost`

function startOnLocalhost(dataDir: string, port: number,
domains: { noTokenSignup: string[]; other: string[]; })

Launches the test server on localhost with mock domain setup and a structured data directory for isolated testing.

📄 mock/tls-options.ts

Purpose: Provides mocked TLS configuration data and helpers for testing secure communication scenarios.

📄 tests/jasmine.ts

Purpose: Global setup or extensions for Jasmine test framework behavior across the test suite. Likely includes custom matchers, reporters, or hooks.

📄 tests/admin-proto/signup.ts

Purpose: Tests administrative logic related to the signup and token management flows for user onboarding.

📄 tests/libs-for-tests/3nstorage.ts

Purpose: Testing utilities and object definitions for working with the `3nstorage` module.

🧩 Interface: `BaseObj`

interface BaseObj {
objId: string,
header: Uint8Array,
segs: Uint8Array,
}

Represents the base form of a stored object for test operations, including its ID and binary segments.

📄 `tests/libs-for-tests/ASMail.ts`

Purpose: Test support for `ASMail` features, particularly message structure and delivery behavior.

🧩 Interface: `Msg`

interface Msg {
cryptoMeta: deliveryApi.msgMeta.CryptoInfo,
msgObjs: Obj[]
}

Defines a message payload for testing, containing cryptographic metadata and one or more message objects.

📄 `tests/libs-for-tests/assert.ts`

Purpose: Simplified assertion helpers for writing readable and reliable test expectations.

⚙️ Function: `assert`

function assert(ok: boolean, errMsg?: string)

Throws an error with a custom message if the given condition evaluates to false.

📄 `tests/libs-for-tests/async-jasmine.ts`

Purpose: Adds async/await-compatible test utilities to Jasmine to improve readability and async test control.

⚙️ Functions:

- `itAsync`, `xitAsync`, `fitAsync`: Define async-compatible test cases, skipped or focused as needed.

- `beforeAllAsync`, `afterAllAsync`, `beforeEachAsync`, `afterEachAsync`: Provide async lifecycle hooks for tests.

📄 tests/libs-for-tests/bytes-equal.ts

Purpose: Byte-level equality check utility for deep comparison of binary data.

⚙️ Function: `bytesEqual`

function bytesEqual(a: Uint8Array, b: Uint8Array)

Returns true if two binary arrays are identical in content and order.

📄 tests/libs-for-tests/json-copy.ts

Purpose: Likely provides deep cloning of JSON-compatible data structures for test isolation.

📄 tests/libs-for-tests/json-equal.ts

Purpose: Tools for deep equality checks between two JSON values in tests.

⚙️ Function: `deepEqual`

function deepEqual(a: any, b: any)

Recursively checks whether two JSON-compatible values are structurally identical.

📁 tests/libs-for-tests

📄 mailerid.ts

Purpose: Contains test utilities related to MailerId handling or generation.

📄 `pkl.ts`

Purpose: Test support for public key login flows, session encryption, and secure exchange mechanisms.

🧩 Interfaces:

- **`User`** Represents a user's cryptographic login information, including default and labeled secret keys.

```
interface User {
id: string;
loginDefaultSKey: Uint8Array;
loginLabeledSKey: {
kid: string;
k: Uint8Array;
}
}
```

- **`ExchangeCrypto`** Carries encrypted values and cryptographic structures used during login/session key exchange.

```
interface ExchangeCrypto {
sessEncr: SessionEncryptor;
serverVerificationBytes: Uint8Array;
encResponse: Uint8Array;
}
```

- **`ExchangeParams`** Bundles `ExchangeCrypto` with an active session ID for structured handling during login flows.

```
interface ExchangeParams {
crypto: ExchangeCrypto;
sessionId: string;
}
```

- **`SessionParams`** Minimal representation of encrypted session configuration.

```
interface SessionParams {
sessEncr: SessionEncryptor;
sessionId: string;
}
```

⚙️ Function:

- **`decryptSessionParamsForCurve25519`** Decrypts session parameters from a `start.Reply` object using the user's login secret key.

function decryptSessionParamsForCurve25519(r: pklApi.start.Reply, loginSKey: Uint8Array)

📄 server-runner.ts

Purpose: A utility to start and manage the test server environment for integration tests.

📄 spec-assembly.ts

Purpose: Responsible for assembling specific test scenarios or configuration sets.

📄 ws-utils.ts

Purpose: WebSocket utility used for integration testing of real-time APIs.

⚙️ **Function:**

- **`openSocket`** Opens a WebSocket connection to a specified URL and attaches a session ID.

function openSocket(url: string, sessionId: string)

📄 xhr-utils.ts

Purpose: Utility functions for working with XMLHttpRequest-like APIs in tests.

🧩 **Interfaces:**

- **`RequestOpts`** Describes options available for HTTP requests made via XHR during testing.

interface RequestOpts {
method: 'GET' | 'POST' | 'PUT' | 'DELETE';
url?: string;
path?: string;
responseType?: 'json' | 'arraybuffer';
sessionId?: string;

responseHeaders?: string[];
}

- `Headers` Wrapper for accessing HTTP headers from responses.

interface Headers {
get(name: string): string|undefined;
}

⚙️ **Functions:**

- `makeException` Constructs a standardized error object from a failed API reply.

function makeException(rep: Reply<any>, errMsg?: string)

- `extractIntHeader` Pulls an integer header from a response and parses it.

function extractIntHeader(rep: Reply<any>, headerName: string)

📄 server-components/3nstorage.ts

Purpose: Simulates encrypted object storage, including chunking, file I/O, and sync logic for testing backend behavior.

📄 server-components/admin.ts

Purpose: Provides test scaffolding and utilities related to administrative functions (e.g. user creation, token handling) in the system.

📄 tests/shared-checks/requests.ts

Purpose: Provides common helper functions used in tests for simulating client HTTP requests, including versioned object transactions and user-related operations.

⚙️ Function: `getCurrentObj`

function getCurrentObj(user: User, objId: string)

Fetches the latest committed version of an object for a given user. Useful in testing retrieval consistency or update propagation.

⚙️ Function: `getCurrentRoot`

function getCurrentRoot(user: User)

Retrieves the latest committed root object version for a user, typically representing the entry point of an object graph.

⚙️ Function: `putCurrentObj`

function putCurrentObj(user: User, objId: string, payload: any)

Saves a new version of a user object with updated content. Used in tests to validate object mutation and chunking logic.

⚙️ Function: `putCurrentRoot`

function putCurrentRoot(user: User, payload: any)

Commits an update to the root object for a given user. Primarily used in syncing metadata or bootstrapping storage state in test scenarios.

⚙️ Function: `archiveObjVer`

function archiveObjVer(user: User, objId: string, version: number)

Archives a specific version of an object for a user. This simulates lifecycle transitions in storage (e.g., archiving old states).

⚙️ Function: `archiveRootVer`

function archiveRootVer(user: User, version: number)

Archives a specific root version. Commonly tested to validate transition from active to archived states across root objects.

⚙️ Function: `deleteObj`

function deleteObj(user: User, objId: string)

Deletes an object from the user's namespace. Useful for testing deletion logic and its effects on downstream operations or versions.

⚙️ Function: `deleteArchiveRootVer`

function deleteArchiveRootVer(user: User, version: number)

Removes an archived version of a root object. Ensures archive lifecycle tests can simulate pruning and data hygiene.

⚙️ Function: `deleteArchiveObjVer`

function deleteArchiveObjVer(user: User, objId: string, version: number)

Deletes a specific archived version of an object. Test coverage ensures that partial deletion and cleanup routines are correct.

⚙️ Function: `transCancel`

function transCancel(user: User, objId: string, transactionId: string)

Cancels an in-progress object transaction for a user. Used in testing rollback scenarios or error recovery mid-transaction.

⚙️ Function: `transCancelRoot`

function transCancelRoot(user: User, transactionId: string)

Cancels a root transaction in progress. Often verified in scenarios involving sync interruptions or rollback validation.
