AT Protocol Spaces

A Field Guide

Nick Gerakines

Edition updated 2026-11-23 · atproto.how/atproto-spaces-field-guide
Text CC BY-SA 4.0 · Code MIT

AT Protocol Spaces: A Field Guide
First edition, 2026

Copyright © 2026 Nick Gerakines. Text licensed under CC BY-SA 4.0. Code samples licensed under MIT.

Published by Nick Gerakines
https://atproto.how/

ISBN 979-8-9828034-0-5 (EPUB)
ISBN 979-8-9828034-1-2 (PDF)

AT Protocol, atproto, and atprotocol are trademarks of Bluesky Social PBC, used under license.

This book describes atproto spaces as the specification stood at the time of writing. The specification may change, and later versions of this guide may differ.

Contents

Preface 1
If you're not a developer 1
How this book works 2
The family in the examples 3
For developers 3
Chapter 1 · Orientation 5
1.1 Why public atproto isn't enough 6
1.2 Roles 7
1.3 Where records live 9
1.4 Access control, not confidentiality 10
1.5 The shape of atproto spaces 11
1.6 Common questions 12
Chapter 2 · Identity and keys 13
2.1 A DID document with spaces support 14
2.2 #atproto: the signing key 15
2.3 #atproto_kem: the key-encapsulation key 15
2.4 #atproto_pds: the space host 16
2.5 Client keys 16
2.6 Ephemeral keys 16
2.7 Rotation 17
2.8 Common questions 18
Chapter 3 · Spaces and addressing 20
3.1 The URI 21
3.2 The authority 23
3.3 The space type 23
3.4 The space key 25
3.5 Spaces used together 26
3.6 Common shapes 26
3.7 Common questions 27
Chapter 4 · Space repos and commits 28
4.1 What a space repo is 28
4.2 Lifecycle 32
4.3 The commit 34
4.4 Why a set hash instead of a tree 35
4.5 Common questions 35
Chapter 5 · Cryptography in depth 37
5.1 LtHash, the set hash 38
5.2 Why MACs and not signatures 39

5.3 The shared secret 40
5.4 MACs as HTTP Message Signatures 41
5.5 Credential binding signatures 44
5.6 Verification order 45
5.7 Common questions 46
Chapter 6 · Access control and credentials 48
6.1 Space credential token exchange 48
6.2 The delegation token 50
6.3 The client attestation 51
6.4 Authorizing space sync 52
6.5 Refusals 54
6.6 Presenting the credential 55
6.7 Expiry and revocation 56
6.8 Writing without read access 57
6.9 Identities without an account 57
6.10 Common questions 61
Chapter 7 · Sync 63
7.1 Getting a copy of a space 63
7.2 Staying current 71
7.3 Blobs 79
7.4 Why this scales 82
7.5 Common questions 82
Chapter 8 · Policies 84
8.1 The lifecycle methods 85
8.2 personalspace 89
8.3 memberspace 89
8.4 managedspace 97
8.5 Publishing configuration to readers 99
8.6 Other policies 101
8.7 Common questions 102
Chapter 9 · OAuth and consent 104
9.1 Grammar 104
9.2 Read is all-or-nothing 105
9.3 Writes and management 106
9.4 Worked scopes 107
9.5 Permission sets 108
9.6 The OAuth consent screen 109
9.7 Common questions 109
Chapter 10 · Operations 110
10.1 Deleting a space 111
10.2 Migrating an account 112

10.3 Moderation 119
10.4 Account status and identity events 121
10.5 Access control in applications 121
10.6 Common questions 122
Chapter 11 · Implementation notes 123
11.1 Where the spec leaves a choice 123
11.2 Passages that need careful reading 125
11.3 Design trade-offs worth knowing 125
11.4 Conformance checklist 126
Appendix A · XRPC methods 128
Appendix B · Headers 132
Appendix C · Token types 133
Appendix D · Errors 134
Appendix E · Test vectors 136
Appendix F · Example identities and keys 137
F.1 Marcus 138
F.2 Elena 139
F.3 Ruth 141
F.4 Nora 142
F.5 Jake 144
F.6 Erin 145
F.7 The managing app 146
F.8 The family app 147
F.9 The repo hosts 150
Appendix G · Worked values 151
G.1 Records 151
G.2 Ruth's commits 152
G.3 Tokens 152
G.4 OAuth requests 154
G.5 Key exchange and MACs 157
G.6 Credential signatures 157
Appendix H · Glossary 159
H.1 Token fields 162
Appendix I · AI use and disclosure 163
Sources 164

Preface

The atproto spaces field guide jumps into the world of permissioned data, going into the use cases and implementation details to explain how it works and why. This guide is for developers and technology enthusiasts who are building a PDS, a syncer, an AppView, or an app that reads and writes spaces, and who need to interoperate with other implementations. There are a lot of new terms and acronyms in this book, and it may feel overwhelming at first. I list them all in Appendix H, each with a short explanation.

📖 No code ahead

If you're not a developer

Most of this book is written for people who build atproto software. If you use apps built on atproto, such as Bluesky, and want to know what spaces mean for you, read this section, and then feel free to put the book down. Everything after it explains how the parts work underneath.

Outside of spaces, everything you post on atproto is public. Anyone can see it, copy it, and prove you wrote it. Spaces add places that aren't public: a group's private forum, a family's updates and photos, posts for paying subscribers, or your own bookmarks and drafts. A space works like a room with a guest list. Only the people and apps on the list can see what's inside.

What you write in a space stays with you. It's stored on the same server that holds your account and your public posts, and it moves with you if you switch servers. Each space has an owner, either a person or an account run for a community, and the owner decides who can read and who can post. Some spaces keep a member list, some open reading to everyone, and some let a service decide, the way a paid newsletter would.

When an app wants to use your spaces, it asks first. The sign-in screen names the kinds of spaces the app wants, such as "Family updates", and what it wants to do with them. A careful app asks only for what it needs, so an app that shows your RSVPs asks to read them and nothing more.

Private here means controlled rather than secret. The servers and apps that carry a space can read what's in it, the way most group services can, and that is what lets them offer search and notifications. Anyone you let in can also copy what they see somewhere else. What spaces add is that a copied post carries no proof that you wrote it. A public post does carry that proof.

If a space is deleted, the apps that kept a copy delete it too. Your own posts stay on your server, where only you can see them, and they come back if the owner recreates the space. If the owner removes you from a space, the apps that follow it stop showing what you posted there.

How this book works

The book follows the atproto spaces specification and its lexicons. Every behavior it describes is checked against atproto-crates, which implements that specification. Where the book mentions the reference PDS, it means Bluesky's official PDS, built from the libraries in bluesky-social/atproto.

Every claim that comes from a source carries a marker, so you know how much weight it bears.

REFERENCE

Source markers

Marker	Meaning
[spec]	Stated in the atproto spaces specification or its lexicons, so it holds for every implementation.
[crates]	Behavior of `atproto-space`, `atproto-space-client`, and `atproto-pds` in atproto-crates, where the spec leaves a choice to the implementation. Another implementation may choose differently.

Unmarked prose is explanation.

If you're implementing spaces yourself, read the book in order the first time. Each chapter assumes only what came before it: why spaces exist, then the keys in a DID document, then addressing, repos, the cryptography, credentials, sync, policies, OAuth, and finally operations and implementation notes. The appendices are for lookup once the model is in your head. Every DID, token, signature, MAC, CID, and

digest in the examples is real and checks out against the keys in Appendix F; long tokens are shortened in the chapters and given in full in Appendix G.

The family in the examples

IN THE FAMILY SPACE

> Marcus creates a private space where his family shares updates, and they read and post through a family app. Elena, the mom, posts pictures of nine-week-old Theo. Ruth, the grandmother, posts news about her health. Nora, the adult daughter, answers the invitation to the summer reunion, and her ex-boyfriend Jake was a member until they broke up.

To help make sense of it all, many example identities, data structures, and XRPC calls follow this family. Marcus's identity is the space's authority. Each record lives in its author's own space repo on that person's PDS, and the family app keeps a copy of the space so it can show everyone's updates. The chapters come back to the family as each part of the protocol is introduced.

Chapters 1 through 11 explain how spaces work. Practicals build on them. Each practical is a short companion guide that builds one complete application, as configuration, requests, responses, and records, and links back to the chapters it relies on. The first, A Forum Space, builds a forum for a small group. If you'd rather see everything working together before the detail, start there and follow its references back into this book.

For developers

This book doesn't assume you read specifications for fun. Each chapter mixes a few kinds of material, each with a different look on the page. Passages marked "In the family space" follow Marcus's family and show what a rule means for the people using it. The plain text around them explains how spaces work and why. Blocks marked "Reference" hold the exact detail: requests, fields, and rules. Where steps happen in an order, one numbered figure shows it. Every chapter also has a passage called "If you're building an app" that says what the chapter means for someone writing an application, and most chapters end with "Common questions" on the things developers tend to ask first. The reference detail is complete, because two

implementations work together only when they agree on every field, but you can skim it until you need it.

There are shorter paths through the book, depending on what you're building.

REFERENCE

Reading paths

If you're building	Read
An app that reads and writes spaces	Chapter 1 for the model, Chapter 3 for how spaces are named, Chapter 4 for writing records, Chapter 6 for credentials, Chapter 8 for policies, Chapter 9 for the OAuth scopes your users will approve, and Chapter 10 for what your app does when spaces are deleted, accounts move, and content is moderated
An app that keeps its own copy of spaces, as a syncer or an AppView does	The same chapters, plus Chapter 7 for sync
A PDS, a space host, or a library	All of it, including the cryptography in Chapter 5 and the implementation notes in Chapter 11

Chapter 1 · Orientation

A space is a named boundary that decides who can read a set of records, while each author keeps their own records on their own PDS.

IN THE FAMILY SPACE

> Marcus wants one place where his family can share updates that only they can see. Elena has pictures of Theo to post, and Ruth has news about her health.

I'd like to use the family from the Preface to paint a picture. Marcus creates a space for his family, which makes his identity the space's authority. The space has a space type, `com.example.familyUpdates`, that says what kind of space it is, and a space key, `our-family`, that tells it apart from any other space of that type he runs.

When Elena posts baby pictures, her PDS stores the post in her space repo for the family, a repo that holds only her records in that space. Ruth's health update sits in her own space repo on her own PDS.

The family app that shows everyone's updates is a syncer. It holds a space credential that lets it read every writer's space repo, and it keeps a copy of the space for itself.

If you're building an app

A few simple patterns cover what most apps do with spaces. They boil down to who can read or write to a space, and who decides.

To write, your app asks the user's PDS to create a record in a space, within an OAuth scope the user approved. The user's PDS accepts the write without checking with the space host, the service that answers for the space as a whole. Whether that write counts as part of the space is decided afterward. The space host tracks a repo only when the space's policy lets its account write. Chapters 4 and 9 explain why.

To read other members' records, your app gets a short-lived delegation token from the user's PDS, trades it with the space host for a credential, and presents that credential to each writer's PDS. Chapter 6 covers it. The other members' records

sit on their own PDSes, where this user's session has no standing.

To stay current, your app registers for notifications when it gets its credential, and pulls a space repo's changes when one arrives. Chapter 7 covers it. A notification carries no records, because atproto spaces do not broadcast data.

To run a space, your app creates it and changes its policy or member list through the authority's PDS, with a session whose scope allows managing the space. Chapter 8 covers it.

1.1 Why public atproto isn't enough

IN THE FAMILY SPACE

> Elena has pictures of nine-week-old Theo, and she wants the family to see them. On public atproto she has one way to post them: to everyone, for good.

Public atproto is built for open, broad distribution. Each account keeps its public records in one repository, or repo, on its PDS. The repo is a Merkle Search Tree whose root is signed by the identity's `#atproto` key, so any record taken from it can prove who wrote it. Relays collect every change from every PDS into one stream, the firehose, and anyone can copy what flows through it. That is what makes public data reliable. It is also why a family photo album or a company's internal forum can't run on it, since every post is copied everywhere, can't be taken back, and proves who wrote it.

Permissioned data needs three things public data lacks. First, it needs a perimeter that a server enforces before handing over data. Second, it needs a sync mechanism. Sync is a simple way of referring to the processes and credentials needed to read data from all of the space repos of accounts in the space. Without it, the application isn't able to stay up to date or include all of the data that is created by accounts in the space. Third, it needs to be verifiable by the reader it was sent to, yet deniable to anyone else, so a record that leaks or is intercepted does not betray who wrote it.

The spec names four modalities to describe what spaces make possible [spec]. The list isn't complete. As we follow Marcus and his family, we'll touch three of them.

REFERENCE

The four modalities [spec]

Modality	Examples from the spec	In this book
Personal data	Bookmarks, mutes, drafts	Jake's own bookmarks space (Chapter 8)
Gated content	Paid newsletters, subscriber-only posts	The paid newsletter (Chapter 8)
Socially shared data	Private posts, stories	Not followed here
Groups	Private forums, communities	The family space

1.2 Roles

There are four roles in atproto spaces that make all of this possible. A single PDS usually plays repo host and space host at once, but any service that implements the required methods can fill either [spec].

IN THE FAMILY SPACE

> Nora answers the reunion invitation in the family app. Her answer has to get from her phone to everyone else in the family, and four roles are involved on the way.

The family app writes her RSVP to her own PDS, which stores it in her space repo for the family space. For this record her PDS is the repo host. Her PDS then tells Marcus's PDS that her repo changed. This change notification doesn't include the changed content, because atproto spaces do not broadcast data. Marcus's PDS is the space host. It keeps the list of repos that have written to the space, called the writer set, and passes the news to the apps that are registered for it. Marcus's identity is the space authority. His key signed the credential the family app uses, and his member list is why Nora's repo counts as part of the space. The family app is the syncer. It hears that Nora's repo moved, asks her PDS for the change with its credential, and adds the RSVP to its copy of the space.

The space host never stores or forwards record bytes. It holds metadata only: each writer's latest `repoRev` and digest, sequenced under a space-wide revision. Record bytes travel only from a repo host to a credentialed syncer.

REFERENCE

The four roles [spec]

Role	What it does	Usually played by	In the family
Repo host	Stores an account's space repos and answers sync requests for them.	The PDS that hosts the account	Nora's PDS, for her RSVP
Space host	Answers for the space as a whole. Authorizes syncers and issues their credentials, tracks the writer set, and sends update notifications. Holds metadata only.	The authority's PDS, named by its `#atproto_pds` entry	Marcus's PDS
Space authority	The DID at the root of the space. Its `#atproto` key signs credentials.	A user's own identity, or a dedicated identity	Marcus
Syncer	Keeps its own copy of a space and serves views from it.	An application	The family app

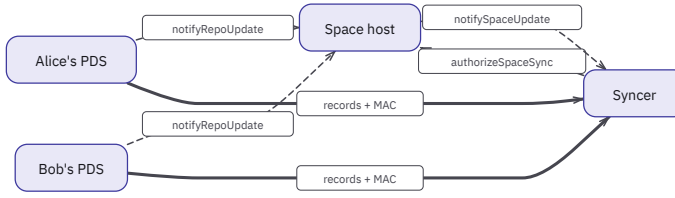

Figure 1.1 · Record bytes flow only from repo hosts to syncers

Each repo host tells the space host only that a repo advanced; the space host relays that to registered syncers, and they pull the records from the repo host directly.

1.3 Where records live

IN THE FAMILY SPACE

> Jake keeps his bookmarks in a space of his own. One of them is a pie recipe the family would like, so he shares it with them. Does the bookmark move, or are there now two?

A space record's address includes its space, so each record belongs to the space it was written into. The content isn't tied to that space, though. Anyone with permission to write in a second space can copy a record's content there, and the copy becomes a new record in that space, kept in the copier's own space repo. When Jake shares a bookmark from his own bookmarks space with the family, he writes a copy into the family space, and each space decides who can read its copy. If Nora copies Elena's post into another space, the copy is Nora's record, written under Nora's identity.

REFERENCE

One bookmark, two records

Jake's bookmark in his own bookmarks space, then his copy in the family space. The content is identical, so both records have the same CID, shown last.

TEXT

```
at:space/did:plc:brthxhhoomvrib2avauses35/com.example.bookmarks/
self/did:plc:brthxhhoomvrib2avauses35/
community.lexicon.bookmarks.bookmark/3n2gtr3c2w22p

at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family/
did:plc:brthxhhoomvrib2avauses35/
community.lexicon.bookmarks.bookmark/3n2gtrxvls22p

bafyreibeecgpzsoedv7jcxr5tf6ggkmpdpolwvq65ehewgirojdghv3hka
```

Each account keeps a separate space repo for every space it writes to. When Elena posts in the family space and also posts in a book club, her PDS holds two space repos for her, one for each space. A space's contents are the records in all of its writers' space repos taken together.

Daniel Holmgren's Reintroducing Spaces argues for grouping records this way over "circles", where each record carries its own audience list. I agree with that argument. Per-record access control lists (ACLs),

dynamic group ACLs, and other fine-grained permission systems are often not worth it. Spaces as designed are a reasonable compromise. A space gives readers a clear boundary. Everything in the family's space is visible to its members, and changing the member list changes access to all of it at once.

1.4 Access control, not confidentiality

IN THE FAMILY SPACE

> Ruth posts that her new medication is working. She means it for the family. Who else is able to read it?

Spaces are not end-to-end encrypted [spec]. The repo host and every credentialed syncer can read the plaintext they contain. The space host holds no records, but it issues the credentials, so it could read them by issuing one to itself. That is what makes search, notifications, moderation, and server-side feeds possible. The spec leaves end-to-end encryption to applications, so an application that needs confidentiality adds its own encryption on top [spec]. The protocol carries an encrypted record like any other record. Removing a reader stops that reader's future reads. It does not recall what they already fetched. A space narrows the "can't be taken back" problem of public data, and does not end it.

REFERENCE

Who can read Ruth's update

Party	In the family	Can it read the update?	Why
The author's repo host	Ruth's PDS	Yes	It stores the record.
A syncer with a credential	The family app	Yes	It fetches the record and keeps its own copy.
The space host	Marcus's PDS	Yes, indirectly	It never stores or forwards record bytes. It issues the credentials, so it could issue one to itself and fetch the record from Ruth's PDS.

Party	In the family	Can it read the update?	Why
A member who was removed	Jake	Only what he already fetched	Nothing recalls a copy.
Anyone else		No	A repo host refuses a read that carries no valid credential.

1.5 The shape of atproto spaces

As we've walked through atproto spaces, it should feel very familiar. Spaces keep the shape of public AT Protocol: DID-based authority, repos that stay with each account, lexicon-typed records, and applications that engage PDSes to build views. What changes is how data is synced, addressed, and committed, and the trust model applied to the data's perimeter. Public data relies on signatures anyone can check, and space data relies on checks that only the reader who asked can make.

REFERENCE

Public atproto and spaces compared

Aspect	Public atproto	atproto spaces
Unit of data	Record in a repo	Record in a space repo
Repo scope	One repo per account	One repo per (account, space)
Record authority	Author DID	Author DID
URI authority	Author DID	Space authority DID
Commit	Signed tree root (MST)	Unsigned digest of the record set (LtHash)
How a commit is proved	Signature, anyone can verify	A MAC made for one request, which only the requester can verify
Archive format	CAR	STAR
How apps collect data	Relays and firehose	The space host's writer set and notifications, then a pull from each repo host
Access	Public	Space credential

The LtHash digest and the per-request MAC are in Chapter 5, the space credential is in Chapter 6, and STAR archives and the writer set are in Chapter 7.

1.6 Common questions

Q Can a space be public?

Yes. A space can let anyone read it while still controlling who writes. A memberspace with `publicRead` set gives a credential to any app that asks, and a managedspace can do the same through its managing app. Chapter 8 builds an announcement space and an open guestbook this way. Readers still get their copy through a credential, so the authority can close the space later by changing its policy.

Q If the servers can read a space, what does it protect?

It controls who gets a copy. A space host hands out credentials only to the readers its policy admits, and repo hosts serve records only to holders of a credential. The repo hosts and the apps that sync a space can read the records, which is what lets them offer search and moderation. The space host could too, by issuing itself a credential. An app that needs secrecy from servers as well encrypts records itself.

Q Do users need new accounts for spaces?

Their existing identity and account work as they are. Spaces use the same DID, the same PDS, and the same OAuth sign-in. The one requirement is a PDS that supports spaces.

Q Does every member get a repo in every space?

An account gets a space repo the first time it writes to a space. Readers who never write have none, so a space with many readers and a few writers has only a few repos.

Chapter 2 · Identity and keys

Every identity in atproto has a permanent identifier in the form of a DID, and a DID document that lists the identity's keys and the services it uses. Anyone can look up a DID's document. Spaces need a handful of keys to save data into a space and read it back, and most of them live in these documents.

IN THE FAMILY SPACE

> The family app is about to read Ruth's health update for Elena. A few things have to be settled first: who vouches that Elena asked, which app is asking, and how the app will know the answer came from Ruth's PDS.

Spaces rely on cryptographic material in DID documents and OAuth client metadata documents. This material is used to verify authorization and to facilitate the exchange of data.

- `#atproto` signs tokens. Elena's PDS uses it to vouch that she asked.
- `#atproto_kem` lets a reader derive a secret for one request, which is how the app knows an answer came from Ruth's PDS.
- An application's own key, published in its JSON Web Key Set (JWKS), signs its client attestation, which tells a space host which app is asking.

If you're building an app

Your app rarely touches these keys directly. The user's PDS holds `#atproto_kem` and `#atproto` and uses them for the user.

Your app already works with keys like these. A confidential client publishes a key in its OAuth client metadata, and spaces use that same key to sign a client attestation. Every app also generates a key for DPoP in each session. Spaces add two more: a key for each credential, and a key for each request that asks for a MAC. They are the same kind of key you already handle, P-256 key pairs, put to new uses. Chapter 6 covers the attestation and the credential key, and Chapter 5 covers the MAC.

If your app reads spaces, it resolves the authority's DID document to find the space host, and each writer's DID

document to find their PDS and KEM key. It refreshes a document when a check fails. Chapter 5 covers the checks.

2.1 A DID document with spaces support

Elena's DID document is an ordinary atproto DID document with one addition. The `#atproto_kem` verification method is introduced to support spaces. The `#atproto` verification method item and `#atproto_pds` service item are already present and remain untouched.

REFERENCE

Elena's DID document

Her identity's two keys, and the server that acts for her.

JSON

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/multikey/v1",
    "https://w3id.org/security/suites/secp256k1-2019/v1",
    "https://w3id.org/security/suites/ecdsa-2019/v1"
  ],
  "id": "did:plc:53ni6w7okqrxvjxezmopejoo",
  "alsoKnownAs": [
    "at://elena.example.com"
  ],
  "verificationMethod": [
    {
      "id": "did:plc:53ni6w7okqrxvjxezmopejoo#atproto",
      "type": "Multikey",
      "controller": "did:plc:53ni6w7okqrxvjxezmopejoo",
      "publicKeyMultibase":
"zQ3shPWwfJKjriJqb4CsR7KBBHkaffnpZin5NCyAYJzztnuRV"
    },
    {
      "id": "did:plc:53ni6w7okqrxvjxezmopejoo#atproto_kem",
      "type": "Multikey",
      "controller": "did:plc:53ni6w7okqrxvjxezmopejoo",
      "publicKeyMultibase":
"zDnaeueFwjSjPA49dvab7bN4VjYDvv7eBVsDSjncdQHsud125"
    }
  ],
  "service": [
    {
      "id": "#atproto_pds",
      "type": "AtprotoPersonalDataServer",
      "serviceEndpoint": "https://pds-a.example.com"
    }
  ]
}
```

Field	What it holds	In Elena's document
`@context`	The standards the document follows	DID Core, Multikey, and two signature suites
`id`	The DID itself	`did:plc:53ni6w7okqrxvjxezmopejoo`
`alsoKnownAs`	The identity's handle	`at://elena.example.com`
`verificationMethod`	The identity's public keys, each named by a fragment	`#atproto` and `#atproto_kem`
`service`	The servers that act for the identity	Her PDS, as `#atproto_pds`

These fields come from specifications outside atproto. DID Core defines the document's structure, and the did:plc and did:web method specifications define how each kind of DID is created, updated, and resolved.

2.2 #atproto: the signing key

This is the identity's existing signing key (K-256 or P-256). It belongs to the core AT Protocol, and spaces reuse it. The PDS almost always manages it entirely, because the PDS is the service that signs data with it. As a *reader*, it signs delegation tokens. As a *space authority*, it signs space credentials and the service auth the authority sends (`notifyCredentialsRevoked`, `checkAccess`). When its account writes to a space, the PDS uses it to sign the service auth on `notifyRepoUpdate`. Every space token header carries `kid: "#atproto"` exactly [spec].

2.3 #atproto_kem: the key-encapsulation key

Each identity whose account holds space repos publishes a P-256 key with `id {did}#atproto_kem`, its DID as controller, `type Multikey`, and the key in atproto multikey encoding (the `zDna…` prefix) [spec]. The repo host holds the private half and uses it only to derive per-request shared secrets for MACs (Chapter 5). It should be a dedicated keypair rather than a reuse of `#atproto`, so rotating one leaves the other alone.

A requester needs this key before it can ask for a MAC, and it may cache it. `atproto-space-client` resolves an author's DID document once, keeps the key, and resolves again after a failed MAC check or an `#identity` event [crates]. An account whose identity publishes no KEM key can't be read with a space credential, because no requester can derive a secret for it.

2.4 #atproto_pds: the space host

A space authority is resolved like any other identity. Its `#atproto_pds` service endpoint is the space host, and its `#atproto` key signs the space's credentials [spec]. Tokens addressed to the space host use the service-id form `did:plc:ejjl2znzqp3ci25dqaslpkk2#atproto_pds` as their `aud`, the claim that names the service a token is meant for.

2.5 Client keys

A space can limit which apps may read it. To enforce that, the space host has to know which app is asking.

An application proves who it is with a client attestation signed by a key from its own `client-metadata.json` (`jwks` or `jwks_uri`). The space host resolves `iss` to that document and picks the key by the attestation's `kid`, the same machinery an atproto confidential OAuth client already uses for `private_key_jwt` [spec]. Public clients have no such key, so they can use only spaces that admit any app.

REFERENCE

The family app's published key

The `jwks` entry in its client metadata, at `https://family.example/client-metadata.json`.

JSON

```
{
  "keys": [
    {
      "kty": "EC",
      "crv": "P-256",
      "x": "COmaVfdUzj6QHwm3ZTZgY8KsbNhtJN12EgFWLsm7_pE",
      "y": "G36xIW2qad2tMbOE8zj-pSq39311l313hbZbcAPTxKw",
      "kid": "key-1",
      "alg": "ES256",
      "use": "sig"
    }
  ]
}
```

2.6 Ephemeral keys

Two more P-256 keys exist only briefly on the syncer. The credential key is generated for each credential, named in `authorizeSpaceSync` as `credentialKey`, and used to sign every call that presents that credential.

That signature is what stops a repo host that receives the credential from replaying it to another host (Chapter 5).

The KEM ephemeral key is generated for each read that asks a repo host for a MAC: a call to `getLatestCommit`, `getRepo`, `listRepoOps`, or `getRecord` made with a space credential. Every page and every retry counts as its own read. The key is sent as the `keyid` of that read's MAC request, and discarded once the response's MAC is checked. Neither is published anywhere.

2.7 Rotation

Rotating different keys and key references will have side effects that developers should be aware of.

The `#atproto` key signs every space credential an authority issues, and any service auth it sends as the authority [spec]. A repo host has to validate a credential's signature before it serves a read [spec]. `atproto-pds` checks it against the key in the authority's current DID document [crates], so a credential issued before a rotation stops verifying once the host sees the new document. The spec gives no rule for that case. Its key rotation entry covers only `#atproto_kem`. What bounds the damage is the credential's lifetime: ten minutes by default and sixty at most [spec].

Rotating `#atproto_kem` leaves every repo's digest and revision as they were, because the key is not part of repo operations that use cryptographic material. Requests made against the old key may fail MAC verification. A syncer that sees a MAC failure refreshes the author's DID document and retries with a new ephemeral key [spec]. Rotation, like every other identity change, arrives as an `#identity` event on the public firehose (Chapter 10).

A space host fetches an app's client metadata and JWKS each time it verifies a client attestation, and picks the key by `kid` [crates]. Rotating that key takes effect on the next attestation. One signed with a key that isn't published yet is refused with `InvalidClientAttestation`, so publish the new key before the app signs with it. Credentials already issued keep working, because they are bound to the credential key.

2.7.1 What should an app do when a user's signing key rotates?

For an app, which user rotated matters. When a member's key rotates, the app holds nothing signed by the old key. Delegation tokens are minted for each exchange and live 60 seconds [spec], so the next one

carries the new key's signature, and space repos hold no signatures at all [spec].

When the authority's key rotates, the app's credential is refused. It calls `authorizeSpaceSync` again, which the spec calls reauthorizing [spec], and retries the read. `atproto-space-client` does that once after an authorization failure [crates]. Until the new credential arrives the app still has its own copy of the space, so its users see data that is a little behind.

An app can also act before it is refused, because a signing key change arrives as an `#identity` event on the firehose [spec]. The spec asks for reauthorizing when the authority's `#atproto_pds` endpoint changes [spec]. `atproto-space-client` gives an app a hook to drop a cached `#atproto_kem` key when an `#identity` event arrives, and it renews a credential when a read is refused [crates].

2.8 Common questions

Q Are app developers responsible for generating or updating an identity's `#atproto` or `#atproto_kem` keys?

No. The user's PDS manages both keys and uses them for the user. Your app reads the public halves from the identity's DID document.

Q Does my app need a key of its own?

A space that lists its allowed apps asks for a client attestation, which your app signs with the key in its client metadata. A public client has no key there, so it is limited to spaces that admit any app. An app that reads spaces also makes short-lived keys as it goes, one for each credential and one for each read that asks for a MAC.

Q Why does an identity need a second key for spaces?

`#atproto` signs tokens. `#atproto_kem` lets a reader and a repo host agree on a secret for one read, which is how the reader knows who answered (Chapter 5). They are separate keys so that either can be rotated by itself.

Q What if an identity has no `#atproto_kem` key?

Then no reader can derive a secret for its space repos, and reads of them with a space credential fail. Erin reads the family space

and has no account, so she has no space repos and needs no such key (Chapter 6).

Q How does an app find the server for a space?

It resolves the space authority's DID document and reads the `#atproto_pds` entry, which names the space host. It finds each writer's repo host the same way, from that writer's document.

Q What happens when a key is rotated?

For an app, a rotation shows up as a failed check or a refused credential. It refreshes the DID document or asks for a new credential, and tries again. The Rotation section covers each key.

Chapter 3 · Spaces and addressing

Spaces introduce a new way to reference both spaces and the records they contain. Spaces only exist in the context of a space owner, referred to as the space authority, a modality, referred to as the space type, and an instance of that modality in the context of the authority, referred to as the space key. These three pieces of information are how you can refer to the data boundary that is a space.

Public atproto records also contain three pieces of information including the record authority, collection, and record key. Combined, a fully qualified AT-URI for a space record includes these six pieces of information that can tell you what you need to access record data. One small but important piece of information is that an AT-URI for a space or a space record begins with "at:space/", where a public AT-URI begins with "at://", to assist with disambiguation.



If you’re building an app

Your app will build and parse AT-URIs constantly, for public records and for space records, so get used to both forms. A space record’s URI starts with the space authority rather than the author, so code that pulls "the DID" out of an AT-URI has to know which kind it holds. The short function in this chapter handles that.

Where a URI came from decides how much work it deserves. One that arrives from outside, in a record your app has just ingested, is worth checking more thoroughly: validate its syntax, resolve the space authority and the author, and look up the lexicons for the space type and the collection, caching what you find. One that your app reads back from its own database has usually been through those checks already and needs less.

You’ll also choose a space type and a space key. The type is a public lexicon that names your kind of space, lists the collections it expects, and gives the title users see when they sign in. The key tells one space apart from another of the same type.

3.1 The URI

IN THE FAMILY SPACE

> Nora answers the reunion invitation in the family app, and her PDS keeps the RSVP in her space repo. The record is hers, and the space it belongs to is Marcus's. Its address has to name both.

Space references and space record references take the form of an AT-URI that contains the space literal, authority, type, and key, followed, in a record reference, by the record authority, collection, and key. These values have an "at:space/" prefix and use "/" characters for separators.

TEXT

```
at:space/[space-authority]/[space-type]/[space-key]/[record-
authority]/[record-collection]/[record-key]
```

REFERENCE

The URI of Nora's RSVP in the family space

TEXT

```
at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family/
did:plc:ashwgt3d5ntfxmlcfyizpu2g/
community.lexicon.calendar.rsvp/3n2gtdo55s22p
```

#	Segment	In Nora's RSVP	Meaning
1	Space literal	`space`	Fixed marker for a space URI
2	Space authority	`did:plc:ejjl2znzqp3ci25dqaslpkk2`	Marcus's identity
3	Space type	`com.example.familyUpdates`	What kind of space it is
4	Space key	`our-family`	Which space of that type
5	Record authority	`did:plc:ashwgt3d5ntfxmlcfyizpu2g`	Nora, who wrote the record
6	Record collection	`community.lexicon.calendar.rsvp`	What kind of record it is
7	Record key	`3n2gtdo55s22p`	Which record of that kind

The literal `space` comes directly after the `at:` scheme, where a public URI has `//`, so a parser tells the two forms apart from the prefix alone [spec]. Truncating after the space key gives a space reference, which lexicons type with the `space-ref` string format [spec].

JAVASCRIPT

```
function uriKind(uri) {
  if (uri.startsWith("at:space/")) {
    const parts = uri.slice("at:space/".length).split("/")
    if (parts.length === 3) return "space reference"
    if (parts.length === 6) return "space record"
    return "other"
  }
  if (uri.startsWith("at://")) {
    const parts = uri.slice("at://".length).split("/")
    if (parts.length === 3) return "public record"
  }
  return "other"
}
```

In a public URI the first DID is the author. In a space URI the first DID is the authority, and the author is the second DID.

Take a URI apart below, then look up the space it names. The console replays a recorded response, so no real space is called.

> Try it 3.1 · Take a URI apart. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/3-addressing#take-a-uri-apart

Responses name spaces by reference in a field called `space`.

REFERENCE

Looking up the family space

The family app calls `getSpace` on Marcus's PDS with his session.

> Try it 3.2 · Look up a space. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/3-addressing#try-get-space
>
> REQUEST
>
> ```
> GET /xrpc/com.atproto.space.getSpace
>   ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
> com.example.familyUpdates/our-family
> Host: pds-a.example.com
> Authorization: DPoP <access token>
> DPoP: <proof JWT>
> ```
>
> HEADERS
>
> `Authorization` The session's access token, bound to a key only the client holds. Appendix G, OAuth requests

DPoP A proof signed with that key for this method and URL, so a copied token can't be replayed. Appendix G, OAuth requests

SAMPLE RESPONSE

```
{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "policy": {
    "$type": "com.atproto.memberspace.policy",
    "publicRead": false,
    "publicWrite": false,
    "publishPolicy": true,
    "publishMembers": true
  }
}
```

3.2 The authority

IN THE FAMILY SPACE

> Marcus created the space under his own identity, the one he already had. It was the quick choice. It also means the family's space goes wherever his account goes, and if he deletes that account, the space is deleted with it.

The authority is the root of trust for the space. It may be a user's own identity, as for bookmarks or drafts, or a dedicated identity for a shared space, which lets the space pass between people without depending on any one of them [spec]. The authority always reads and writes its own spaces, under every policy, without appearing on any member list [spec].

The authority comes first after the literal because it is the root, and every part after it is scoped by the parts before it. Marcus is the authority. Under his identity is a kind of space, named by the space type, and under that type the space key picks out one space. The record's parts continue the pattern, with an author in that space, a collection of that author's, and a record key in that collection.

3.3 The space type

IN THE FAMILY SPACE

When Elena signs in to the family app, her PDS asks her to approve access to "Family updates". That name comes from the space type. If the type later gains a new kind of record, the app can write it for Elena without asking her to sign in again.

The type is an NSID naming the modality. It resolves to a space type declaration, a lexicon whose `main` definition has `"type": "space"`:

REFERENCE

The family space's type declaration

Marcus's identity publishes it as the record `at://did:plc:ejjl2znzqp3ci25dqaslpkk2/com.atproto.lexicon.schema/com.example.familyUpdates`, and the DNS record `_lexicon.example.com` names that identity.

JSON

```
{
  "$type": "com.atproto.lexicon.schema",
  "lexicon": 1,
  "id": "com.example.familyUpdates",
  "defs": {
    "main": {
      "type": "space",
      "description": "A family's private space for updates,
photos, and events",
      "key": "any",
      "title": "Family updates",
      "title:lang": {
        "es": "Novedades familiares"
      },
      "collections": [
        "com.example.familyUpdates.post",
        "community.lexicon.calendar.event",
        "community.lexicon.calendar.rsvp",
        "community.lexicon.bookmarks.bookmark",
        "com.atproto.label.record"
      ]
    }
  }
}
```

REFERENCE

Fields of a space type declaration [spec]

Field	Required	Meaning
`type`	yes	`"space"`, on the `main` definition

Field	Required	Meaning
`description`	no	For developers; not shown to users
`key`	yes	Recommended space key type, like a record key type
`title`	no	Short name shown on consent screens
`title:lang`	no	Localized titles by language code
`collections`	yes	Collections clients should expect; no wildcard

A space type declaration is a public lexicon, published like any other: a `com.atproto.lexicon.schema` record in the repo of the identity that controls the NSID's authority, found through its `_lexicon` DNS record. Anyone can resolve it. A PDS has to, because it shows the `title` on the consent screen and reads the `collections` when it checks a bare scope [spec].

A space's records are permissioned, but its type is always public; there is no private space type declaration. The reason is the way lexicon resolution works today. The `_lexicon` DNS record says which repository holds the schemas, and it has no way to carry a space's authority, type, and key as well. Resolution ends at a public repo, so the declaration has to be published there. The side effect is that the kinds of records a space of that type is expected to hold are public. Someone who also knows who is in a space can start to piece together what it contains.

A lexicon author can add a record type to a space type later, and existing sessions pick it up without the user consenting again. That works because the declaration's `collections` are the default write set for a bare `space:` scope (Chapter 9), and the default is read from the declaration each time a grant is checked [spec]. An app that wants a fixed write set lists the collections in its scope. The list may name NSIDs from any namespace. A space may still hold records of any collection. The type also tells a reader what kind of data to expect before any network call, and its `title` is what users see on the OAuth consent screen, with the NSID shown when there is no title [spec].

3.4 The space key

The space key, written `skey` in the spec and the lexicons, distinguishes spaces of the same type under one authority. It follows record-key syntax, up to 512 bytes [spec]. The declaration's `key` field recommends a key style the way a record schema does: `self` for a singleton like a user's bookmarks, a slug like `our-family`, a TID for spaces created on demand, or a cryptographic identifier.

REFERENCE

Space key syntax

A space key is 1 to 512 bytes, drawn from `A-Z`, `a-z`, `0-9`, and `. _ : ~ -`, and it is never `.` or `..` [spec]. `atproto-space` checks this when it builds a `SpaceKey` [crates].

3.5 Spaces used together

An application often treats several spaces under one space authority as one whole. This guide calls that set a hyperspace. A hyperspace has a primary space, which holds the main content or community and which the other spaces are organized around. A hyperspace can hold a single space, with the other positions unfilled.

Two or more spaces are in syzygy (pronounced "SIZ-ih-jee") when they share the same space authority and the same space key but have different space types, as `at:space/{spaceDid}/{typeA}/foo` and `at:space/{spaceDid}/{typeB}/foo` do. Each is the other's syzygy space. The standard example is a community space and its moderation space.

An inner space sits beneath another space in the same hyperspace, such as a subgroup focused on specific content for members of the space above it. It has the same space authority and its own space key, and it can have a syzygy of its own.

That gives a hyperspace two directions. Along the type dimension the key stays and the type changes, which is where syzygy lives. Along the key dimension the key changes, which is where inner spaces live.

These are conventions an application adopts. Every space in a hyperspace is a full space with its own policy and member list, and the application arranges how they relate.

3.6 Common shapes

Table 3.1 · Common space shapes

Use	Authority	Type	Space key
Bookmarks	the user	`com.example.bookmarks`	`self`
A family's updates	a family member	`com.example.familyUpdates`	`our-family`
A shared photo album	the creator	`com.example.album`	a TID

Use	Authority	Type	Space key
A community and its moderation space	the community's identity	one type for each space	the same key for both
A subgroup inside a community	the community's identity	the community's type or one of its own	a key of its own

3.7 Common questions

Q How can I tell a space record's URI from a public one?

Look at the prefix. A space URI begins with `at:space/`, and a public URI begins with `at://`. The short function earlier in this chapter does exactly that.

Q Can I create my own space type?

Yes. A space type is a lexicon, so anyone who controls an NSID's domain can publish one. Publish it like any other lexicon, give it a `title` people will recognize on the consent screen, and list the collections your app writes.

Q Can a space hold records outside its type's collections?

Yes. The `collections` list sets which collections a bare `space:` scope lets an app write by default. A space can still hold any record type, so an app with a broader scope can write other kinds of records into it.

Chapter 4 · Space repos and commits

A space repo is where one identity's records live inside the boundary of a space. The first time an identity writes a record into a space, either someone else's or their own, its account gets a space repo that belongs to that one space. In practice a space repo is an extension to the space authority's atproto space data boundary. Underneath, it is a set of records summarized by a commit, and the commit's `data` is a 32-byte digest of that set. A syncer computes the same digest from its own copy. If the two match, the syncer holds exactly the records the host holds.

> If you're building an app
>
> Writing to a space looks almost exactly like writing a public record. `com.atproto.space.createRecord`, `putRecord`, `deleteRecord`, and `applyWrites` take the same inputs as their `com.atproto.repo.*` counterparts, plus a `space`. They leave out the `swapCommit` and `swapRecord` inputs [spec]. The record lands in the user's space repo on their own PDS. The commit and its digest matter once your app reads spaces, since they're how it confirms its copy is complete, and a library will usually handle that part for you.

4.1 What a space repo is

IN THE FAMILY SPACE

> Marcus has invited everyone to the summer reunion. Nora wants to answer maybe, because she has work commitments that week and can't promise yet. Her answer should reach the family and nobody else. Where does that answer live?

Records in a space repo are scoped the same way public records are, by collection and record key. The keys are `{collection}/{rkey}` and the values are DRISL-encoded records [spec]. Nora has one space repo per space she has written to, each on her own PDS, each with its own revision.

The PDS offers its write methods under `com.atproto.space.*`. They mirror `com.atproto.repo.*`, with `space` added and `repo` naming the caller's own DID [spec]. The separate namespace is a signal to developers.

Authentication behaves differently for these calls than for public records and carries different constraints, and a PDS stores a space repo differently inside. With that much different underneath, separate XRPC methods were the practical choice.

REFERENCE

The write methods [spec]

Method	Input	Output	Errors
`createRecord`	`space`, `repo`, `collection`, `rkey?`, `validate?`, `record`	`uri`, `cid`, `validationStatus?`	`SpaceNotFound`, `RecordAlreadyExists`
`putRecord`	`space`, `repo`, `collection`, `rkey`, `validate?`, `record`	`uri`, `cid`, `validationStatus?`	`SpaceNotFound`
`deleteRecord`	`space`, `repo`, `collection`, `rkey`	empty	`SpaceNotFound`
`applyWrites`	`space`, `repo`, `validate?`, `writes`	`results?`	`SpaceNotFound`, `RecordNotFound`, `RecordAlreadyExists`
`clearRecords`	`space`, `repo`	empty	none declared

`putRecord` creates or updates as needed, and `deleteRecord` also succeeds when the record is already absent. A delete is about the end state. The record should be gone when the call returns, so removing data is always safe, and calling it a second time does no harm. Reads go through `getRecord` and `listRecords`, covered with the other repo-host reads in Chapter 7.

REFERENCE

`validate` and `validationStatus`

Setting `validate` to `false` skips Lexicon validation, `true` requires it, and leaving it unset validates records whose Lexicon the PDS knows [spec]. `atproto-pds` answers `validationStatus` `"valid"` for a record it checked and `"unknown"` for one it had no Lexicon for, and leaves the field out when validation was skipped [crates].

A record write names its space explicitly:

Try it 4.1 · Create an RSVP in the family space. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/4-space-repos#try-create-record

REQUEST

```
POST /xrpc/com.atproto.space.createRecord
Host: pds-b.example.com
Authorization: DPoP <access token>
DPoP: <proof JWT>
Content-Type: application/json

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "repo": "did:plc:ashwgt3d5ntfxmlcfyizpu2g",
  "collection": "community.lexicon.calendar.rsvp",
  "record": {
    "$type": "community.lexicon.calendar.rsvp",
    "subject": {
      "uri": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family/
did:plc:ejjl2znzqp3ci25dqaslpkk2/
community.lexicon.calendar.event/3n2gsywstc22e",
      "cid":
"bafyreigjlo4wnf5e6inmlkp7ib7pcym5o34vblhnbrf6tij6olzq4hyliu"
    },
    "status": "community.lexicon.calendar.rsvp#interested"
  }
}
```

HEADERS

`Authorization` The session's access token, bound to a key only the client holds. Appendix G, OAuth requests

`DPoP` A proof signed with that key for this method and URL, so a copied token can't be replayed. Appendix G, OAuth requests

SAMPLE RESPONSE

```
{
  "uri": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family/
did:plc:ashwgt3d5ntfxmlcfyizpu2g/
community.lexicon.calendar.rsvp/3n2gtdo55s22p",
  "cid":
"bafyreiautip3hgihuw4ypueftfdiv3iyliieamu5mdv4d4duqhhyetxs7a"
}
```

IN THE FAMILY SPACE

Ruth answered "interested" while she waited to hear from her doctor. Now she can travel, and she wants to change her answer to "going". The family's list should never show two answers from her, or none.

`applyWrites` applies a batch in one commit. Its `writes` are a closed union of three objects defined in its own lexicon, and its `results` come back in the same order. Here Ruth replaces her RSVP in one commit, creating a new one marked going and deleting the first:

Try it 4.2 · Replace an RSVP in one commit. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/4-space-repos#try-apply-writes

REQUEST

```
POST /xrpc/com.atproto.space.applyWrites
Host: pds-b.example.com
Authorization: DPoP <access token>
DPoP: <proof JWT>
Content-Type: application/json

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "repo": "did:plc:p4256dosynj6j6tkypz2q2l3",
  "writes": [
    {
      "$type": "com.atproto.space.applyWrites#create",
      "collection": "community.lexicon.calendar.rsvp",
      "rkey": "3n2gtxdkr222p",
      "value": {
        "$type": "community.lexicon.calendar.rsvp",
        "subject": {
          "uri": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family/
did:plc:ejjl2znzqp3ci25dqaslpkk2/
community.lexicon.calendar.event/3n2gsywstc22e",
          "cid":
"bafyreigjlo4wnf5e6inmlkp7ib7pcym5o34vblhnbrf6tij6olzq4hyliu"
        },
        "status": "community.lexicon.calendar.rsvp#going"
      }
    },
    {
      "$type": "com.atproto.space.applyWrites#delete",
      "collection": "community.lexicon.calendar.rsvp",
      "rkey": "3n2gtkszes22p"
    }
  ]
}
```

HEADERS

`Authorization` The session's access token, bound to a key only the client holds. Appendix G, OAuth requests

`DPoP` A proof signed with that key for this method and URL, so a copied token can't be replayed. Appendix G, OAuth requests

SAMPLE RESPONSE

```
{
  "results": [
    {
      "$type": "com.atproto.space.applyWrites#createResult",
      "uri": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family/
did:plc:p4256dosynj6j6tkypz2q2l3/
community.lexicon.calendar.rsvp/3n2gtxdkr222p",
      "cid":
"bafyreiblr6f6oy3zi426nksahddwp4tgsfgedtqk27fyxqtav66kjvwkgu"
    },
    {
      "$type": "com.atproto.space.applyWrites#deleteResult"
    }
  ]
}
```

REFERENCE

`applyWrites` operations and results [spec]

Operation	Fields	Result
`#create`	`collection`, `value`, and optionally `rkey`	`#createResult`: `uri`, `cid`, and optionally `validationStatus`
`#update`	`collection`, `rkey`, and `value`	`#updateResult`: `uri`, `cid`, and optionally `validationStatus`
`#delete`	`collection` and `rkey`	`#deleteResult`, which is empty

4.2 Lifecycle

IN THE FAMILY SPACE

> Jake shared one bookmark with the family before he and Nora broke up. Marcus has since taken him off the member list. The bookmark is still in Jake's space repo on Jake's PDS, where Marcus has no way to delete it.

A space repo doesn't exist without data. It comes into being on its first write. This lazy creation is different from a public repo, which exists as soon as the account does. It was a deliberate design choice, and I agree with it. I'd like to see public repos work the same way. When the authority is someone else, Nora's PDS accepts the write without asking whether the space exists; the space host decides later whether to track

her. When the writer is the authority, as Marcus is, the space must already exist [spec].

That means Nora can write records into a space Marcus hasn't created yet. The records sit in her space repo on her PDS, and her PDS notifies Marcus's space host of the write as it always does [spec]. In `atproto-pds`, the space host answers `SpaceNotFound` for a space it doesn't have, and her PDS stops there [crates]. Once Marcus creates the space and admits her, her next write notification puts her repo in the writer set [spec]. Syncers then pull her whole repo, and the earlier records come with it, so a syncer sees them as soon as it sees the repo.

An empty repo and a missing repo are the same thing, so an account deletes its space repo by clearing it. The cleared repo stays in the writer set, and its host keeps answering sync requests with a commit showing that the records are gone [spec]. `clearRecords` deletes every record (apart from any published configuration records in the authority's repo, Chapter 8), advances the repo to a new `rev` with the empty digest, clears the operation log, and sends a write notification to the space host [spec]. The space host and syncers learn the records are gone. Clearing a repo never deletes the space.

Removing a writer leaves that writer's repo alone. The authority drops the account from the writer set, syncers stop treating the repo as part of the space, and the records stay on the writer's PDS until the writer clears them [spec].

A host can throw an empty space repo away once the clear has had time to spread, and the spec asks it to wait at least 72 hours [spec]. That number is a commonly used safe margin. It is also the recovery window `did:plc` gives an identity to undo a bad operation. The spec doesn't mention `did:plc` here, but I think the two are related.

REFERENCE

Garbage collection

A repo host may discard an empty space repo, its commit, and any remaining state. It should wait at least 72 hours after the clear, so the clear reaches the space host and syncers first. Once collected, the host answers as if the repo never existed, syncers treat that the same as an empty repo, and no notification is sent [spec]. Account migration may also leave empty repos behind.

4.3 The commit

IN THE FAMILY SPACE

> The family app has pulled Ruth's records from her PDS. Before it shows who is coming to the reunion, it needs to know that it holds every one of them, and that they came from her PDS.

A space repo's commit carries no signature, and that is a deliberate design decision. A signed commit would prove who wrote the records to anyone who ever saw a copy, including someone outside the space [spec]. Spaces offer a measure of deniability instead (Chapter 5). No one can take one of your records and link it back to you in a way that can be checked cryptographically. That has costs as well as benefits, and the protocol's judgment is that the benefits are far greater.

REFERENCE

The commit [spec]

Ruth's space repo after she changes her RSVP.

JSON

```
{
  "ver": 1,
  "repo": "did:plc:p4256dosynj6j6tkypz2q2l3",
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "rev": "3n2gtxdkr232p",
  "data": {
    "$bytes": "eVD7vKgDQrLKfOvN3VZJHYX0HGz158ocKa5oRd3kaYw"
  }
}
```

Field	Type	Meaning
`ver`	integer	Commit format version, `1`
`repo`	DID	Whose repo this is. A syncer checks it against the repo it asked for.
`space`	space-ref	Which space the repo is in. A syncer checks it the same way.
`rev`	TID	The revision of this commit. A syncer passes it to `listRepoOps` to ask what changed since.
`data`	32 bytes	`sha256` of the 2048-byte LtHash state. A syncer compares it with the digest of its own copy.

The lexicon definition is `com.atproto.space.defs#commit` [spec]. In JSON, `data` is written as `{"$bytes": "<base64>"}`, the lexicon form for raw bytes. In a STAR archive, the binary format that `getRepo` returns (Chapter 7), the

commit is encoded in DRISL instead. DRISL is a strict form of CBOR, a binary format much like JSON, and there `data` is a plain byte string.

`rev` is a TID, the sortable, timestamp-based identifier that atproto uses for revisions [spec].

The commit also has no `prev` and no tree root. It binds the repo to its space and revision, and `data` summarizes exactly which records exist. The host proves the commit to each requester with an HTTP Message Signature carried in the response headers (Chapter 5). A copy of the commit on its own proves nothing to anyone.

The empty repo's `data` is always:

TEXT

```
e5a00aa9991ac8a5ee3109844d84a555
83bd20572ad3ffcd42792f3c36b183ad
```

4.4 Why a set hash instead of a tree

Public atproto stores each repo as a Merkle Search Tree, so anyone can prove a record is in the repo or walk the changes between two versions. Space repos use something simpler. A syncer only needs to know whether its copy equals the host's, and it learns what changed from the op log (Chapter 7). A set hash answers the equality question with 32 bytes. It costs one vector addition per write, and depends only on which records exist. There is no dependency on the order they were written. The host keeps the full 2048-byte state and the commit carries a digest. A set hash gives up both of the tree's abilities. It can't prove that one record is in a repo, and it can't say what changed between two versions, so a syncer that loses its place in the op log fetches the whole repo again (Chapter 7) [spec].

The first of those losses is deliberate. A Merkle Search Tree gives a chain of validation, from one record up to the root and from the root into a signed commit. A space leaves that chain out on purpose, so a record that leaks stays deniable (Chapter 5).

4.5 Common questions

Q Can a user write to a space before it exists?

When someone else is the authority, yes. The user's PDS accepts the write without checking, and the space host decides later

whether to track it. When the user is the authority, the space has to be created first.

Q What happens to a user's records when they're removed from a space?

They stay in the user's space repo on the user's PDS. Removing a writer tells syncers to stop treating that repo as part of the space. To delete the records as well, the user's app calls `clearRecords`, which empties the repo in one step.

Q Why doesn't the commit carry a signature?

A signature would prove who wrote the records to anyone who ever saw a copy, including someone outside the space. Chapter 5 explains how spaces prove a commit to the one reader who asked, and to no one else.

Chapter 5 · Cryptography in depth

NOTE

Who this chapter is for. This chapter is dense because its subject is dense. Most readers, and most developers, never need the internals of these algorithms and data structures, and can probably skip it. It is written mostly for people building storage and sync applications and tools.

Commits in a space repo aren't signed with the account's signing key, because permissioned data shouldn't carry proof of authorship that survives a leak outside the space [spec]. A syncer still needs to know that its copy is complete and that it came from the repo host. Spaces use four cryptographic constructions: LtHash to summarize a repo, HPKE secret export to agree a per-request secret, RFC 9421 `hmac-sha256` signatures to authenticate a commit or record under that secret, and RFC 9421 `ecdsa-p256-sha256` signatures to bind a credential to its holder.

If cryptography is new to you, each one does one job:

- LtHash makes a short fingerprint of every record in a repo, so a reader can check that its copy is complete.
- HPKE secret export lets the reader and the repo host agree on a secret for one request, without ever sending the secret itself.
- The HMAC signature uses that secret to show that a response came from the repo host and wasn't changed on the way.
- The ECDSA signature shows that a request came from the app the credential was issued to.

In atproto-crates, `atproto-space` implements all four: `set_hash` (LtHash), `kem` (the shared secret), `mac` (commit and record MACs) and `http_signature` (credential binding). `atproto-space-client` signs and verifies through them [crates].



If you're building an app

If you use a library such as `@atproto/space` or `atproto-space-client`, you won't implement any of this, but it helps to know what a failed check means. `@atproto/space` is the official TypeScript package, maintained by Bluesky Social PBC, and `atproto-space-client` is the Rust client in atproto-crates. A MAC failure usually

means the author rotated a key, so refresh their DID document and retry. A digest mismatch means your copy has drifted, so recover the repo with `getRepo`. If you're writing the library, this chapter is your specification, and Appendix G holds worked values to test against.

5.1 LtHash, the set hash

LtHash keeps a running total. Each record is turned into a long list of numbers. Writing the record adds those numbers to the total, and deleting it subtracts them. Addition doesn't care about order, so two copies that hold the same records always reach the same total, however the records arrived. The commit carries a short fingerprint of that total. A syncer that computes the same fingerprint from its own copy holds the same records as the host. The numbers come from a one-way hash, so neither the total nor the fingerprint can be turned back into records. Two copies can be compared without revealing what either one holds.

LtHash comes from Lewi, Kim, Maykov, and Weis, Securing Update Propagation with Homomorphic Hashing. Spaces use the 16-bit variant: a 2048-byte state read as 1024 little-endian `uint16` lanes [spec].

Each record contributes one element, the UTF-8 bytes of `{collection}/{rkey}/{record_cid}`, where `record_cid` is the DASL CID of the record's DRISL encoding, rendered as a string [spec]. For example:

TEXT

```
community.lexicon.calendar.rsvp/3n2gtxdkr222p/
bafyreiblr6f6oy3zi426nksahddwp4tgsfgedtqk27fyxqtav66kjvwkgu
```

Both sides must produce identical bytes. `atproto-space` exposes `record_element_bytes(collection, rkey, cid)` so callers don't hand-format it [crates].

Adding means expand the element to 2048 bytes with BLAKE3 in XOF mode, read those bytes as 1024 little-endian `uint16` lanes, and add each lane into the state modulo 65536. Removing subtracts instead [spec]. An update is a remove of the old element and an add of the new one.

PYTHON

```
from blake3 import blake3
import hashlib, struct
```

```
def lanes(elem: bytes):
    raw = blake3(elem).digest(length=2048)
    return struct.unpack("<1024H", raw)

def add(state, elem):
    return [(a + b) & 0xFFFF
            for a, b in zip(state, lanes(elem))]

def sub(state, elem):
    return [(a - b) & 0xFFFF
            for a, b in zip(state, lanes(elem))]

def digest(state):
    raw = struct.pack("<1024H", *state)
    return hashlib.sha256(raw).digest()

EMPTY = [0] * 1024
```

REFERENCE

Properties

Both operations are commutative, so the state depends only on the current set of records, never on the order of writes [spec]. The empty state is all zeroes, and the commit's `data` is `sha256(state)`. The host keeps the full state, and the commit carries this digest.

REFERENCE

Security

Collision resistance rests on a lattice problem, which is believed to resist quantum attack [spec].

Lanes are little-endian. The signature halves and the HPKE labels in this chapter are big-endian.

5.2 Why MACs and not signatures

IN THE FAMILY SPACE

> Before Marcus removed him, Jake could see everyone's reunion RSVPs in the family app, and he kept a copy of Ruth's. If he shows it to someone outside the family, can that person check that Ruth wrote it?

A public record is signed so anyone, anywhere, forever, can confirm who wrote it. If someone had a PLC audit log for an identity, and a MST

(or slice of one) they effectively have offline content that is self verifying. Permissioned data should not carry proof of authorship that survives a leak outside its access boundary, so space repos are not signed [spec]. Holmgren's permissioned data diary makes the full case.

Instead the syncer and the repo host agree an ephemeral secret for one request, and the host returns a MAC (message authentication code) under it. A MAC is a short tag computed from a message and a secret, and only someone who holds the secret can make one or check one. The syncer can check the MAC, so it trusts the response. But the syncer could have computed the MAC itself, so the MAC proves nothing to anyone else. This property is called deniability.

5.3 The shared secret

The secret comes from the secret export feature of HPKE (RFC 9180). Only export is used, so nothing is encrypted [spec].

REFERENCE

The HPKE suite [spec]

Parameter	Value
Mode	Base (`0x00`)
KEM	DHKEM(P-256, HKDF-SHA256), `0x0010`
KDF	HKDF-SHA256, `0x0001`
AEAD	Export-only, `0xFFFF`
`info`	`"atproto-mac"`
`exporter_context`	empty
`L`	32

TEXT

```
// on the syncer
enc, shared_secret = SendExportBase(
    author_kem_public_key,
    info="atproto-mac",
    exporter_context="", L=32)

// on the repo host
shared_secret = ReceiveExportBase(
    enc, author_kem_private_key,
    info="atproto-mac",
    exporter_context="", L=32)
```

With this suite, `enc` is the syncer's ephemeral P-256 public key. The syncer sends it as the `keyid` of its MAC request, encoded as a `did:key`.

HPKE encodes `enc` as an uncompressed point, while `did:key` uses the compressed form, so convert at that boundary [spec]. The syncer should derive a new secret for every request, including retries and each page of a paginated response; the repo host derives its copy after authorizing the request [spec].

Where no HPKE library with secret export is available, the spec's non-normative appendix spells the same result in plain ECDH and HKDF-SHA256 calls:

TEXT

```
kem_id   = "KEM" || uint16be(0x0010)
suite_id = "HPKE" || uint16be(0x0010)
        || uint16be(0x0001) || uint16be(0xFFFF)

LabeledExtract(id, salt, label, ikm) =
  HKDF-Extract(salt, "HPKE-v1" || id || label || ikm)
LabeledExpand(id, prk, label, info, L) =
  HKDF-Expand(prk, uint16be(L) || "HPKE-v1"
              || id || label || info, L)

// 1. ECDH; keys uncompressed (0x04 || x || y)
dh = ECDH(ephemeral_priv, author_pub)  // syncer
   = ECDH(author_priv, ephemeral_pub)  // host
eae_prk    = LabeledExtract(kem_id, "", "eae_prk", dh)
kem_secret = LabeledExpand(kem_id, eae_prk,
               "shared_secret",
               ephemeral_pub || author_pub, 32)

// 2. Key schedule, base mode, no PSK
psk_id_hash = LabeledExtract(suite_id, "",
                "psk_id_hash", "")
info_hash   = LabeledExtract(suite_id, "",
                "info_hash", "atproto-mac")
context     = 0x00 || psk_id_hash || info_hash
secret      = LabeledExtract(suite_id, kem_secret,
                "secret", "")
exporter_secret = LabeledExpand(suite_id, secret,
                "exp", context, 32)

// 3. Export, empty exporter context
shared_secret = LabeledExpand(suite_id,
                exporter_secret, "sec", "", 32)
```

Check either path against the export-only P-256 entry in the HPKE test vectors. This KEM is not post-quantum secure; a future version may swap in one that is, such as ML-KEM [spec].

5.4 MACs as HTTP Message Signatures

A MAC is an RFC 9421 HTTP Message Signature using `hmac-sha256` (§3.3.3), keyed with `shared_secret`. If an `alg` parameter is present, it must

be `hmac-sha256` [spec]. Methods that return a commit or a single record require one when called with a space credential.

The syncer asks for a MAC with `Accept-Signature`. The `tag` names the profile, which fixes the covered header:

REFERENCE

MAC profiles [spec]

Profile (`tag`)	Covered header	Methods
`atproto-space-commit`	`Atproto-Space-Commit`	`getLatestCommit`, `getRepo`, `listRepoOps`
`atproto-space-record`	`Atproto-Space-Record`	`getRecord`

REFERENCE

A commit MAC [spec]

The family app asks Ruth's PDS for her latest commit, after she changed her RSVP.

HTTP

```
Accept-Signature: mac=("atproto-space-commit";sf);
keyid="did:key:zDnaenqFhbY4sffBMz5ZatBohkr8LNNAe1pWTmG6C88rMc477";
  tag="atproto-space-commit"
```

The value is wrapped here for width; on the wire it is one line. The label (`mac`) is the syncer's choice. `keyid` is the ephemeral key. The request may also carry `alg`, and nothing else [spec].

The host responds with the covered header, a signature under the requested label and parameters, and `Accept-Signature` in `Vary`:

HTTP

```
Atproto-Space-Commit: ver=1,
  repo="did:plc:p4256dosynj6j6tkypz2q2l3",
  space="at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  rev="3n2gtxdkr232p",
  data=:eVD7vKgDQrLKfOvN3VZJHYX0HGz158ocKa5oRd3kaYw=:
Signature-Input: mac=("atproto-space-commit";sf);
  keyid="did:key:zDnaenqFhbY4sffBMz5ZatBohkr8LNNAe1pWTmG6C88rMc477";
  tag="atproto-space-commit"
Signature: mac=:qcuN1a76R+6Aq9Drm6C0aSYroDZp08eeMcArYT+vd/M=:
Vary: Accept-Signature
```

The covered headers are Structured Field Dictionaries, signed with the `sf` flag, so both sides canonicalize them before signing [spec]. The signature base the HMAC runs over is therefore:

TEXT

```
"atproto-space-commit";sf: ver=1,
repo="did:plc:p4256dosynj6j6tkypz2q2l3",
  space="at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  rev="3n2gtxdkr232p",
data=:eVD7vKgDQrLKfOvN3VZJHYX0HGz158ocKa5oRd3kaYw=:
"@signature-params": ("atproto-space-commit";sf);
  keyid="did:key:zDnaenqFhbY4sffBMz5ZatBohkr8LNNAe1pWTmG6C88rMc477";
  tag="atproto-space-commit"
```

Each entry is one line in the real base; the breaks are for width.

For `getRecord`, the profile is `atproto-space-record` and the covered header carries the full record URI and CID:

REFERENCE

A record MAC [spec]

Ruth's changed RSVP, read with `getRecord`.

HTTP

```
Atproto-Space-Record:
  uri="at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family/
did:plc:p4256dosynj6j6tkypz2q2l3/
community.lexicon.calendar.rsvp/3n2gtxdkr222p",
cid="bafyreiblr6f6oy3zi426nksahddwp4tgsfgedtqk27fyxqtav66kjvwkgu"
Signature-Input: mac=("atproto-space-record";sf);
keyid="did:key:zDnaenqFhbY4sffBMz5ZatBohkr8LNNAe1pWTmG6C88rMc477";
  tag="atproto-space-record"
Signature: mac=:bR41y+7bxRG0QywwNj6jb+fsh70sdOMUi0tmN9OsRRw=:
Vary: Accept-Signature
```

REFERENCE

Host rules

The host ignores requested signatures with other tags. It answers `InvalidSpaceMac` when `Accept-Signature` is missing or malformed, contains more than one request with the profile's tag, requests other parameters, or has an invalid key [spec].

REFERENCE

Syncer rules

The syncer requires exactly one signature with the profile's tag, covering exactly the profile's header. If the signature has a `keyid`, it must match the request. Then [spec]:

- *Commits:* the `Atproto-Space-Commit` fields match the commit in the body (or the STAR metadata from `getRepo`), and `repo` and `space` match the requested repo.
- *Records:* the `uri` matches the request, and the CID computed from the returned record matches the header's `cid`.
- *On failure:* refresh the author's DID document and retry with a new keypair.

5.5 Credential binding signatures

IN THE FAMILY SPACE

> The family app reads Ruth's, Nora's, and Jake's space repos from one PDS, and Marcus's and Elena's from another. Each PDS sees the app's credential when it answers. What stops one PDS from using that credential to read the repos on the other?

A space credential is good across every repo host in the space. As a bearer token, any repo host that received it could replay it to read repos it should not see. So each credential names a key (`cnf.kid`, per RFC 7800 §3.4), and every request presenting it carries an RFC 9421 `ecdsa-p256-sha256` signature by that key over exactly two components [spec]:

REFERENCE

The credential signature base [spec]

The family app presents its credential to Ruth's PDS.

TEXT

```
"authorization": Atproto-Space-Credential
eyJhbGciOiJFUzI1NksiLCJ0eXAiOi...
"atproto-space-audience": did:plc:p4256dosynj6j6tkypz2q2l3
"@signature-params": ("authorization"
  "atproto-space-audience");tag="atproto-space-credential"
```

The signature value is 64 bytes, `r || s`, each half a zero-padded 32-byte big-endian integer, the same encoding as atproto commit signatures. Unlike commit signatures, low-S normalization is not required, and

verifiers must accept both low-S and high-S [spec]. Libraries tuned for public atproto often reject high-S by default; turn that off here.

The signature covers neither the HTTP method nor the request path (the `@method` and `@path` components in RFC 9421), so it may be reused with the same credential and audience until the credential expires. Chapter 10 explains why that is deliberate. The host-side checks are listed in Chapter 6, under presenting the credential.

5.6 Verification order

A syncer reading a repo runs these checks in order:

1. The host accepts the credential and its signature. If it doesn't, the host returns an error.
2. The commit's `repo` and `space` match the request. If they don't, the syncer rejects the response.
3. The MAC verifies, and the covered header matches the commit. If not, the syncer rejects the response.
4. The running LtHash over the records the syncer holds equals the commit's `data`. If it doesn't, the copies have diverged, and the syncer recovers with `getRepo`.

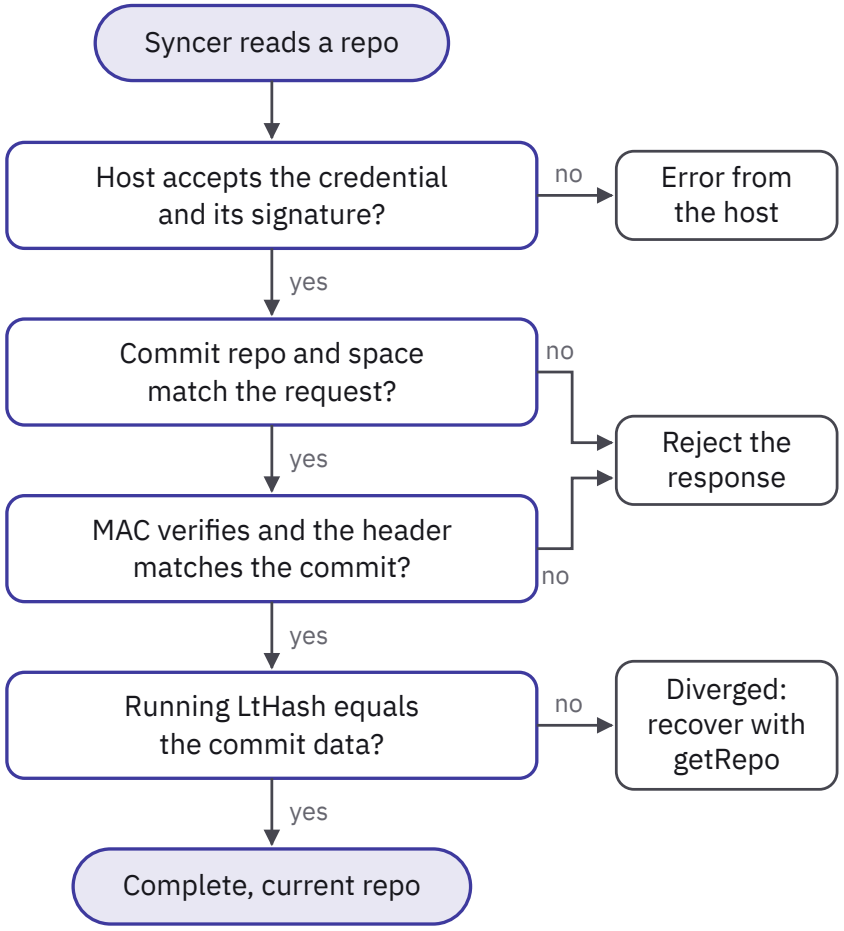

Figure 5.1 · Verification order

Only the last comparison establishes that the syncer holds the complete, current repo.

5.7 Common questions

Q Do I have to implement any of this?

If you use a library such as `@atproto/space` or `atproto-space-client`, it signs your requests and checks the responses for you. This chapter is for people writing a library, a PDS, or a syncer of their own.

Q Is the data in a space encrypted?

No. Spaces control who gets a copy, and the servers that handle a space can read its records. HPKE is here for its secret export, which gives the reader and the repo host a shared secret. Nothing is encrypted with it.

Q Can I prove to someone else who wrote a record?

No, and that is on purpose. A MAC convinces the reader who asked for it. That reader could have made the same MAC, so it proves nothing to anyone else.

Q What should my app do when a check fails?

A failed MAC usually means the author's `#atproto_kem` key changed, so refresh their DID document and retry with a new ephemeral key. A digest mismatch means your copy and the host's hold different records, so fetch the repo again with `getRepo`.

Q Is this safe against quantum computers?

Partly. The set hash rests on a lattice problem that is believed to resist quantum attack. The key agreement uses P-256, which is not post-quantum secure, and the spec says a future version may swap in a KEM that is, such as ML-KEM.

Chapter 6 · Access control and credentials

Because a space is a data perimeter, something has to enforce it, and that is where space credentials come in. A space's records are spread across the PDSes of the accounts that wrote them, so there is no single door to guard. The space authority decides who can read the data and puts that decision in a short-lived space credential. The space host mints it in `authorizeSpaceSync` in exchange for up to two signed assertions: a delegation token naming the user's identity, and a client attestation naming the app. Each repo host checks the credential before it hands a record to a syncer, and it can do so without asking the authority [spec].



If you're building an app

This is the chapter where your app gets permission to read. The flow is a token exchange. The user's PDS gives your app a 60-second delegation token, your app trades it with the space host for a credential that lasts ten minutes by default, and that credential reads every repo in the space. One credential serves all of your users of that space, so you need it through only one user's session. Plan for credentials to expire, because short expiry is how access is revoked.

6.1 Space credential token exchange

IN THE FAMILY SPACE

> Elena opens the family app to see who is coming to the reunion. The event record and the RSVP records are in space repos on two different PDSes. The app needs one thing that both PDSes will accept as proof that Marcus's space lets it read.

The app gets a space credential by trading a short-lived token from Elena's PDS with Marcus's space host. This is what that process looks like:

1. Elena signs in to the family app with OAuth, granting `space:com.example.familyUpdates?authority=*`.

2. The app calls `com.atproto.space.getDelegationToken` on Elena's PDS and receives a 60-second token naming Elena and the family space.
3. The app generates a fresh P-256 keypair, signs a client attestation with its own JWKS key, and calls `com.atproto.space.authorizeSpaceSync` on Marcus's space host with the token, the attestation, and the new public key.
4. The space host verifies what it received, asks the space's policy whether Elena and the app may read, and returns a credential bound to the new key.
5. The app reads every writer's repo, from each writer's own PDS, with that credential and a signature by the bound key.

The PDS hosting the user's account and space repo follows the space's policy on who gets the records it holds, and that typically involves confirming who the user is. The space host decides who's a member and signs the credential that repo hosts check.

A space may require either, neither, or both the delegation token and client attestation. The host must not reject a request just because it carries a valid assertion that the space doesn't require [spec]. A space with public read access skips steps 1 and 2, because a delegation token isn't necessary. A space that admits any app skips the attestation in step 3.

Which of the two assertions a space asks for is set by its policy (Chapter 8). With the member list policy, `com.atproto.memberspace.policy`, the four combinations look like this:

REFERENCE

What a member list policy asks for [spec]

Delegation token	Client attestation	Who can read	Policy settings
Not needed	Not needed	Anyone, with any app	`publicRead`, no `allowedApps`
Needed	Not needed	Members only, with any app	The defaults
Not needed	Needed	Anyone, but only through listed apps	`publicRead` and `allowedApps`
Needed	Needed	Members only, and only through listed apps	`allowedApps`

One credential is used for the whole space, so an app serving many users only needs one credential per space, obtained through any one

user's session [spec]. Calling `authorizeSpaceSync` again before the credential expires is reauthorizing. If the space requires a delegation token and the app loses every session for it, the app can no longer reauthorize. Which of its users may see the space is the app's own decision (Chapter 10).

6.2 The delegation token

This token is minted by the PDS hosting the user's account. It is single-use, valid 60 seconds by default, and addressed to the space host [spec]. The short life fits how the token is used. The app trades it for a space credential right away, as the first of two calls made in succession, so it never needs to last longer than that exchange. The app's session must hold a `space:` scope with `read` covering the space.

REFERENCE

`getDelegationToken` [spec]

The family app asks Elena's PDS for a token.

Try it 6.1 · Get a delegation token. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/6-access-control#try-get-delegation-token

REQUEST

```
GET /xrpc/com.atproto.space.getDelegationToken
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
Host: pds-a.example.com
Authorization: DPoP <access token>
DPoP: <proof JWT>
```

HEADERS

`Authorization` The session's access token, bound to a key only the client holds. Appendix G, OAuth requests

`DPoP` A proof signed with that key for this method and URL, so a copied token can't be replayed. Appendix G, OAuth requests

SAMPLE RESPONSE

```
{
  "token":
"eyJhbGciOiJFUzI1NksiLCJ0eXAiOiJhdHByb3RvLXNwYWNlLWRlbGVnYXRpb24rand0Iiwia2lkIjo
hlQvyJPVdoOADyuptkdq1VkYmA"
}
```

The response holds only the token. Decoded:

JSON

```
{
  "alg": "ES256K",
  "typ": "atproto-space-delegation+jwt",
  "kid": "#atproto"
}
{
  "iss": "did:plc:53ni6w7okqrxvjxezmopejoo",
  "sub": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "aud": "did:plc:ejjl2znzqp3ci25dqaslpkk2#atproto_pds",
  "iat": 1794843000,
  "exp": 1794843060,
  "jti": "16d985e81a23c572196a5b487d95ad98"
}
```

It looks like service auth, but it's a distinct credential type [spec]:

- Its `typ` is `atproto-space-delegation+jwt`, where service auth uses `JWT`.
- It carries no `lxm`, because it authorizes an exchange rather than one XRPC method.
- Its `sub` is the space, so the token is good for that space alone.

It asserts only that Elena delegates to the app. Whether Elena is a member is the authority's call.

6.3 The client attestation

A client attestation proves which OAuth client is asking for a space credential. It names only the client, never the user. The delegation token names the user, and the space host checks the two independently. The space host can verify it because the app's OAuth `client_id` is a URL pointing to its client metadata document, which identifies the signing key. The attestation is signed by the app with a key from its published JWKS, in the shape of an atproto `private_key_jwt` client assertion but addressed to the space host [spec].

REFERENCE

Client attestation [spec]

The family app's attestation, addressed to Marcus's space host, decoded.

JSON

```
{
  "alg": "ES256",
  "typ": "atproto-client-attestation+jwt",
  "kid": "key-1"
}
{
  "iss": "https://family.example/client-metadata.json",
  "sub": "https://family.example/client-metadata.json",
  "aud": "did:plc:ejjl2znzqp3ci25dqaslpkk2#atproto_pds",
  "iat": 1794843001,
  "exp": 1794843061,
  "jti": "6998f590a0858fa82833a1ac0f109db8"
}
```

The host resolves `iss` to the client metadata, fetches its JWKS, and verifies with the key named by `kid`. The verified `iss` is the attested `client_id` that `allowedApps` lists compare against and that `checkAccess` receives as `clientId`.

6.4 Authorizing space sync

REFERENCE

`authorizeSpaceSync` [spec]

The family app trades Elena's token and its attestation for a credential.

Try it 6.2 · Authorize space sync. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/6-access-control#try-authorize-space-sync

REQUEST

```
POST /xrpc/com.atproto.space.authorizeSpaceSync
Host: pds-a.example.com
Content-Type: application/json

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "credentialKey":
"did:key:zDnaeyJrXggfgfsMLx91wQDoHqk5rMmGtE4nkXGG3x5Xzs1rn",
  "delegationToken": "eyJhbGciOiJFUzI1NksiLCJ0eXAiOi...",
  "clientAttestation": "eyJhbGciOiJFUzI1NiIsInR5cCI6Im...",
  "notify": {
    "endpoint": "https://family.example",
    "token": "xIFHpilmY0Wlfxq62sqovyfp66i08zazz2dgpsT_38E"
```

```
  }
}
```

SAMPLE RESPONSE

```
{
  "credential":
"eyJhbGciOiJFUzI1NksiLCJ0eXAiOiJhdHByb3RvLXNwYWNlLWNyZWRlbnRpYWwrand0Iiwia2lkIjo
ELKxFgo3kjlFWH6frlUd38sRbm45zxJGhjLg",
  "spaceRev": "3n2gtrxvls22e"
}
```

The request doesn't cary transport authentication, such as an OAuth session or service auth, and doesn't need any. The delegation token and client attestation in the body are signed JWTs that say who is asking, and the credential the host returns is bound to `credentialKey`, so only the holder of that key can use it [spec]. `credentialKey` is required and must be a P-256 `did:key`; the host puts it in the credential as `cnf.kid`. `notify` is optional and registers the caller for update notifications (Chapter 7). The response's `spaceRev` is the space's current revision, omitted before the space's first update.

The credential, decoded:

JSON

```
{
  "alg": "ES256K",
  "typ": "atproto-space-credential+jwt",
  "kid": "#atproto"
}
{
  "iss": "did:plc:ejjl2znzqp3ci25dqaslpkk2",
  "sub": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "cnf": {
    "kid": "did:key:zDnaeyJrXggfgfsMLx91wQDoHqk5rMmGtE4nkXGG3x5Xzs1rn"
  },
  "iat": 1794843001,
  "exp": 1794843601,
  "jti": "6d3dabf08639dcf98ee9fe28cf4314a3"
}
```

The credential is signed by the authority's `#atproto` key. It lasts ten minutes by default and sixty at most. Repo hosts don't check back with the authority each time a credential is used, so its lifetime is a trade-off. A shorter one makes syncers mint credentials more often. A longer one stretches the window in which a removed member, a

removed app, or a leaked credential can still read. It works with every repo host in the space, so it has no `aud` [spec]. Generate a new credential key for each credential and discard it when the credential expires.

`atproto-pds` runs its checks in this order, which balances security against the load on the host [crates]:

1. Checks that `credentialKey` is a P-256 `did:key`, and that `notify` is well formed if present.
2. Confirms it hosts the space's authority.
3. Confirms the authority's account is active.
4. Answers `SpaceDeleted` if the space was deleted.
5. Verifies the delegation token.
6. Verifies the client attestation.
7. Asks the space's policy whether the user and the app may read.

6.5 Refusals

REFERENCE

`authorizeSpaceSync` errors [spec]

Error	When
`SpaceDeleted`	The space was deleted. Answered before any authorization check.
`SpaceNotFound`	No such space, or in place of `AccountNotAuthorized` to avoid revealing that the space exists
`AccountNotAuthorized`	The policy refuses the identity
`AppNotAuthorized`	The policy refuses the app, or a required attestation is missing
`NotAuthorized`	Generic refusal, which may replace either of the two above
`InvalidDelegationToken`	Token malformed, expired, wrong `aud`, `sub`, `typ`, or signature
`InvalidClientAttestation`	The attestation, or the client metadata it points to, can't be verified
`RegistrationNotConfirmed`	The `notify` endpoint did not confirm the registration
`RepoTakendown`, `RepoDeactivated`	The authority's account is in that state

These are the errors the `authorizeSpaceSync` lexicon declares [spec]. Because a host may answer a refused identity with `SpaceNotFound` or the generic `NotAuthorized`, don't infer membership from the error name. `atproto-pds` answers `AppNotAuthorized` and `InvalidClientAttestation` with status 400 and a suspended authority with `RepoSuspended`. When a managed

space's managing app can't answer, it returns `502 UpstreamFailure`, which the lexicon does not list [crates]. Treat `UpstreamFailure` as retryable, never as a refusal.

6.6 Presenting the credential

Every read sends the credential, an audience, and a signature by the bound key:

REFERENCE

A read with a credential [spec]

The family app asks Ruth's PDS for her latest commit.

HTTP

```
GET /xrpc/com.atproto.space.getLatestCommit
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
  &repo=did:plc:p4256dosynj6j6tkypz2q2l3
Authorization: Atproto-Space-Credential
eyJhbGciOiJFUzI1NksiLCJ0eXAiOi...
Atproto-Space-Audience: did:plc:p4256dosynj6j6tkypz2q2l3
Signature-Input: sig=("authorization"
  "atproto-space-audience");tag="atproto-space-credential"
Signature: sig=:aJSLd/
+RWTwSPbsM8ZKMIWXqkzDTt67Lnbi0AaFcax8M0owwTdDkpiRPA+I84/7BkhcI7Ls/5YbZmRHgd0CZTg=
Accept-Signature: mac=("atproto-space-commit";sf);
keyid="did:key:zDnaenqFhbY4sffBMz5ZatBohkr8LNNAe1pWTmG6C88rMc477";
  tag="atproto-space-commit"
```

The audience is the repo owner's DID for repo methods and the authority's DID for host methods such as `listWriters` [spec].

A host receiving a credential runs through the following checklist to validate it [spec]:

1. requires an `Authorization` header with the `Atproto-Space-Credential` scheme (recognized case-insensitively) and an `Atproto-Space-Audience` header;
2. validates the credential's type, issuer, signature, subject space, and expiration;
3. requires exactly one signature with `tag="atproto-space-credential"`, found by its tag under any label, since intermediaries may relabel;
4. requires its covered components to be exactly `("authorization" "atproto-space-audience")`, in that order;
5. if the signature has a `keyid`, requires it to equal `cnf.kid`; if it has an `alg`, requires `ecdsa-p256-sha256`; ignores any other parameters;

6. requires `cnf.kid` to be a P-256 `did:key` and verifies the signature against it;
7. checks that the audience equals the one the request implies.

The signature base uses the `Authorization` value exactly as received, with no case normalization.

6.7 Expiry and revocation

IN THE FAMILY SPACE

> The family app got a credential through Jake's session one minute before Marcus removed him. That credential reads the whole space, and it has nine minutes left. Marcus doesn't want to wait.

When an authority stops authorizing an app or the app has no authorized user's session left for a space that needs one, then the app can no longer reauthorize, and its credential lapses within its remaining lifetime [spec]. That makes short expiry the main revocation mechanism.

For faster cutoff the space host may send `com.atproto.space.notifyCredentialsRevoked` to repo hosts in the space, with service auth from the authority. Its input names the space and 1 to 100 revoked credential IDs (`jti`) [spec]:

REFERENCE

`notifyCredentialsRevoked` [spec]

Marcus's PDS tells each repo host to refuse that credential.

JSON

```
{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "credentials": [
    "8e78341f7da46cf37e3cbd8611ff6bbc"
  ]
}
```

A repo host must reject a revoked `jti` and remember it for at least 60 minutes, the longest a credential can live [spec]. In `atproto-pds`, revocation is idempotent, so delivery can be retried [crates]. The space host should also drop a notification registration whose latest credential it revokes [spec].

6.8 Writing without read access

With the memberspace policy, each member list entry carries both `read` and `write` flags, and they are checked at different moments. `read` is checked when an app asks for a credential. `write` is checked when a write notification arrives [spec]. A managedspace has no member list, and its managing app answers `read` and `write` separately through `checkAccess` [spec]. A member can hold either one alone, so a space can have members who put records in and never see what anyone else wrote. A suggestion box works this way, and so does a submissions inbox.

Writing records to your own space repository doesn't use space credentials. The member's app writes to the member's own space repo through its OAuth session, and the member's PDS notifies the space host. The space host finds `write: true`, adds the repo to the writer set, and tells syncers [spec]. Readers of the space see those records like any others.

Reading is where the member is turned away. The member's PDS will still hand the app a delegation token, because it checks the app's OAuth scope and knows nothing about the member list. The space host then finds `read: false` and refuses to issue a credential [spec]. `atproto-pds` answers `SpaceNotFound`, in the same words it uses for a space that doesn't exist [crates]. The member can still read their own space repo through the OAuth session, which needs no credential [spec].

One gap is left to the app. A credential covers the whole space and names no user, so an app that serves both this member and a member who can read already holds the space's records. The protocol can't stop that app from showing them to the wrong person. The app has to check each user's `read` flag itself, from the published member records when the space sets `publishMembers`, and show the space to the users who have it (Chapter 10).

6.9 Identities without an account

IN THE FAMILY SPACE

> Erin has never had an account on a PDS. Her identity lives at her own domain, `erin.example`, and Marcus has put her on the family's member list. With no PDS to vouch for her, how does she get a credential?

A member of a space doesn't need an account. A member list is a list of DIDs, and the space host checks a delegation token by resolving the issuer's DID document and verifying the signature against the `#atproto` key it finds there [spec]. Nothing in that check asks which server signed the token. An identity that holds its own `#atproto` private key can sign a delegation token itself and trade it for a credential, the same way an app does on a user's behalf.

Erin's identity is `did:web:erin.example`. Its DID document lists a `#atproto` key but doesn't have any services because there is no PDS:

JSON

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/multikey/v1",
    "https://w3id.org/security/suites/secp256k1-2019/v1"
  ],
  "id": "did:web:erin.example",
  "alsoKnownAs": [
    "at://erin.example"
  ],
  "verificationMethod": [
    {
      "id": "did:web:erin.example#atproto",
      "type": "Multikey",
      "controller": "did:web:erin.example",
      "publicKeyMultibase":
"zQ3shvWu4gufp9Qp4fCXSRtKdvBeX6nGVEjkvYuE2s3pYgAtz"
    }
  ]
}
```

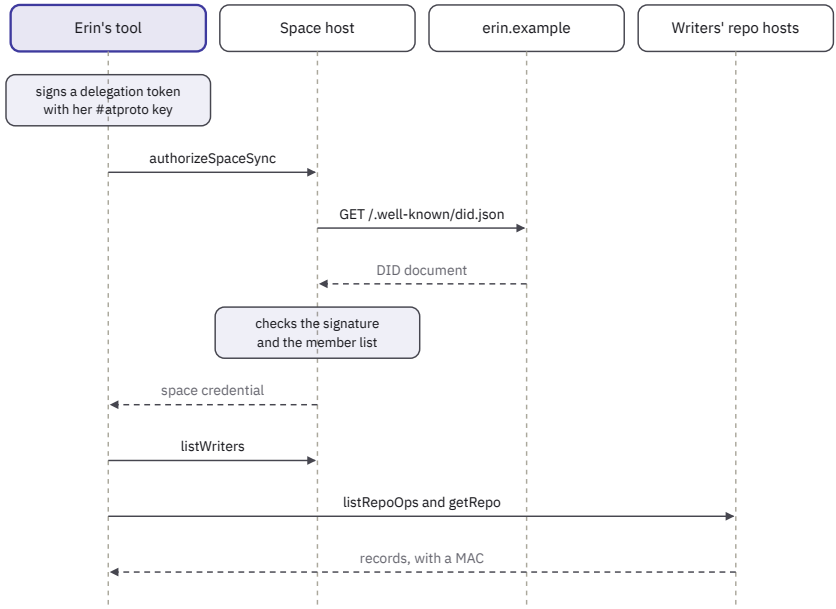

Figure 6.1 · Erin signs her own delegation token and reads like any syncer

The important thing to reiterate is that Erin signs tokens with the key she owns, as published in her DID document.

Marcus adds Erin to the family space with the `putMembers` call from Chapter 8. Erin has no space repo to write from, so her entry grants read access only:

JSON

```
{
  "did": "did:web:erin.example",
  "read": true,
  "write": false
}
```

Erin's tool signs a delegation token with her `#atproto` key, carrying the same claims a PDS would put in one. Decoded:

JSON

```
{
  "alg": "ES256K",
  "typ": "atproto-space-delegation+jwt",
  "kid": "#atproto"
}
{
  "iss": "did:web:erin.example",
```

```
  "sub": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "aud": "did:plc:ejjl2znzqp3ci25dqaslpkk2#atproto_pds",
  "iat": 1794843420,
  "exp": 1794843480,
  "jti": "c6b0c3082eeaf51efea6a49167ac8a28"
}
```

It then generates a P-256 credential key and calls `authorizeSpaceSync`. The family space admits any app, so the request carries no client attestation:

HTTP

```
POST /xrpc/com.atproto.space.authorizeSpaceSync
Content-Type: application/json

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "credentialKey":
"did:key:zDnaerESMGLw8NRXkwfGkQvpQ9u2R26RMXChzjkUbp9M1K7iw",
  "delegationToken": "eyJhbGciOiJFUzI1NksiLCJ0eXAiOi..."
}
```

The space host fetches `https://erin.example/.well-known/did.json`, verifies the token against the `#atproto` key there, finds Erin on the member list with `read: true`, and answers with a credential bound to her key. Decoded:

JSON

```
{
  "alg": "ES256K",
  "typ": "atproto-space-credential+jwt",
  "kid": "#atproto"
}
{
  "iss": "did:plc:ejjl2znzqp3ci25dqaslpkk2",
  "sub": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "cnf": {
    "kid": "did:key:zDnaerESMGLw8NRXkwfGkQvpQ9u2R26RMXChzjkUbp9M1K7iw"
  },
  "iat": 1794843421,
  "exp": 1794844021,
  "jti": "a9ede65fb181dd467a2bb7ef53de3dd5"
}
```

From there Erin reads the space like any syncer, starting with the writer set:

HTTP

```
GET /xrpc/com.atproto.space.listWriters
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
  &limit=100
Authorization: Atproto-Space-Credential
eyJhbGciOiJFUzI1NksiLCJ0eXAiOi...
Atproto-Space-Audience: did:plc:ejjl2znzqp3ci25dqaslpkk2
Signature-Input: sig=("authorization"
  "atproto-space-audience");tag="atproto-space-credential"
Signature: sig=:QVs7ToEXTf7SUsuiF0mE9umJtzgctST1uId3TajXZyEqG7/
mccBw3Ys04UEJuH2ofMCzsSKTxpicDm91/x3uFA==:
```

The full delegation token and credential are in Appendix G, and Erin's keys are in Appendix F.

Erin can read, and only read. Writing to a space means writing records into a space repo, and space repos live on repo hosts, whose write methods accept only an OAuth session on an account there [spec]. Giving Erin `write: true` has no effect until she has an account somewhere. Her DID document needs no `#atproto_kem` key either, since that key serves readers of an account's space repos, and Erin has none.

A space that sets `allowedApps` still asks for a client attestation from a listed app. Erin can act as her own client by publishing client metadata with a JWKS, as long as the authority lists her `client_id`.

The spec describes the delegation token as something the user's PDS mints, and that is the usual path. An identity with an account keeps its `#atproto` signing key on its PDS, and the PDS signs the token on the identity's behalf. Erin's token is self-signed instead. Her `did:web` document publishes her `#atproto` key, she holds the private half herself, and her own tool signs the delegation token with it. The space host verifies both kinds the same way, by resolving the issuer's DID document and checking the signature against its `#atproto` key, so it never needs to know who did the signing. A self-signed token should follow the same habits a PDS does, with a 60-second lifetime and a fresh `jti` every time. A space host caps how long a credential can live but not a delegation token, so Erin's tool keeps the token's lifetime short itself.

6.10 Common questions

Q Does my app need a credential for each user?

One per space is enough. A credential reads the whole space, and your app can get it through any one user's session. Which of your users may see that space is your app's decision (Chapter 10).

Q Can an app without a backend read spaces?

It can read spaces that admit any app. A space that lists its allowed apps asks for a client attestation, which takes a private key kept on a server. A public client, such as a single-page app or a mobile app without its own backend, has no such key.

Q What happens when a member is removed?

Their sessions stop producing credentials, because the policy now refuses them. Credentials already issued keep working until they expire, ten minutes by default, unless the space host revokes them sooner. Your app may still hold a valid credential through other members, so it also stops showing the space to the removed user.

Chapter 7 · Sync

The data perimeter of a space is dynamic and may extend past a single PDS to many. Each writer's records sit in a space repo on that writer's own PDS, so the perimeter reaches a new PDS whenever an account hosted there writes for the first time. The space host keeps a list of who has written, without the records themselves [spec]. The space comes together in one place, the copy a syncer builds, and every view your users see is drawn from that copy. Public atproto has relays that gather everyone's records into one firehose. Spaces have no relay [spec], so each app that reads a space does the gathering itself. A syncer uses everything the earlier chapters set up: the credential that lets it in (Chapter 6), the MAC that tells it who answered (Chapter 5), and the commit digest that tells it its copy is complete (Chapter 4).

A syncer learns *who* has written to the space from the space host and fetches *what* they wrote from each writer's own repo host. A syncer's copy is correct when its set hash matches the one in the repo host's commit. Notifications only tell the syncer when to check. Missing one delays an update, but the next check still catches up.



If you're building an app

Building an app on atproto spaces means understanding a space's data perimeter and copying some or all of the data inside it, so your app can index and store it. Your app asks the space host who has written, pulls each writer's repo from that writer's own PDS, and then stays current through notifications and the op log. It confirms every change with the repo host before trusting it, and deletes data only on the space host's own word. An app that can live with some delay can poll on a schedule instead of registering for notifications.

7.1 Getting a copy of a space

Your app can't show anything from a space until it holds its own copy of the records. There's no relay to subscribe to, so the syncer builds that copy one writer at a time. It asks the space host who has written, then goes to each writer's PDS for their records. The copy is ready to trust once it matches the commit each repo host hands back, and the sections below cover how a syncer gets there and how it checks.

7.1.1 First contact: the writer set

IN THE FAMILY SPACE

> The family app starts with a clean slate. It may know a few things about the space Marcus set up, but doesn't have any records or blobs yet. Before it can read any, it has to learn which accounts have written to the space and where their space repos are.

The writer set is the list of accounts that have written at least one record into the space under the policy [spec]. An account the policy allows to write joins it only after its first write. A syncer starting calls `listWriters` on the space host, with a credential whose audience is the authority:

REFERENCE

`listWriters` [spec]

The family app's call to Marcus's space host at 15:35, after Ruth's change and Jake's removal, so the response shows what a syncer starting at that moment would see.

Try it 7.1 · List the writer set. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/7-sync#try-list-writers

REQUEST

```
GET /xrpc/com.atproto.space.listWriters
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
  &limit=100
Host: pds-a.example.com
Authorization: Atproto-Space-Credential <credential>
Atproto-Space-Audience: did:plc:ejjl2znzqp3ci25dqaslpkk2
Signature-Input: sig=("authorization" "atproto-space-
audience");tag="atproto-space-credential"
Signature: sig=:<signature by the credential key>:
```

HEADERS

`Authorization` The space credential the authority signed, naming the space and the key it is bound to. Appendix C, Token types

`Atproto-Space-Audience` The host the request is for: the repo's DID for repo methods, and otherwise the space's authority. Appendix B, Headers

`Signature-Input` The two headers the binding signature covers. Appendix B, Headers

`Signature` The credential key's signature over those headers, which shows the caller holds the key the credential is bound to. Appendix B, Headers

SAMPLE RESPONSE


```
{
  "repos": [
    {
      "$type": "com.atproto.space.defs#repoWrite",
      "repo": "did:plc:53ni6w7okqrxvjxezmopejoo",
      "repoRev": "3n2gsittdk32e",
      "data": {
        "$bytes":
"LQ8ty3iUor9cbpB7YIHwOzjr7V9P+yOiOatCVXE2rdY"
      },
      "spaceRev": "3n2gsittdk42e"
    },
    {
      "$type": "com.atproto.space.defs#repoWrite",
      "repo": "did:plc:ashwgt3d5ntfxmlcfyizpu2g",
      "repoRev": "3n2gtdo55s32p",
      "data": {
        "$bytes":
"INjMYdWaUBlDhmPVrmtaDralYgQLu+GffB4QmvGD2SE"
      },
      "spaceRev": "3n2gtdo55s22e"
    },
    {
      "$type": "com.atproto.space.defs#repoWrite",
      "repo": "did:plc:p4256dosynj6j6tkypz2q2l3",
      "repoRev": "3n2gtxdkr232p",
      "data": {
        "$bytes":
"eVD7vKgDQrLKfOvN3VZJHYX0HGz158ocKa5oRd3kaYw"
      },
      "spaceRev": "3n2gtxdkr222e"
    },
    {
      "$type": "com.atproto.space.defs#repoRemoved",
      "repo": "did:plc:brthxhhoomvrib2avauses35",
      "spaceRev": "3n2gtz4rss22e"
    },
    {
      "$type": "com.atproto.space.defs#repoWrite",
      "repo": "did:plc:ejjl2znzqp3ci25dqaslpkk2",
      "repoRev": "3n2gu2vyuk32e",
      "data": {
        "$bytes":
"x2sUmOzCG8t26JimHB15JxsbE9nBw3TZODEvXO+czuQ"
      },
```

```
      "spaceRev": "3n2gu2vyuk42e"
    }
  ],
  "cursor": "3n2gu2vyuk42e"
}
```

The response lists the latest update to each repo, ordered by `spaceRev` ascending. Each entry in `repos` is a union member carrying a `$type`, one of two shared definitions in `com.atproto.space.defs` [spec].

Clients should branch on `$type` for every entry. The response leaves `prevSpaceRev` unset. A `#repoRemoved` entry means the authority took that repo out of the writer set (see Removing a writer, below), so stop treating it as part of the space. Pass `cursor` to resume, and the host returns repos updated after that space revision. `limit` runs from 1 to 1000 and defaults to 100. The declared errors are `SpaceNotFound`, `RepoTakendown`, and `RepoDeactivated`. The last two mean the authority's account is taken down or deactivated [spec].

Treat the list as the authority's claim, current as far as its notifications have kept it. Inclusion is necessary but not sufficient: an app may still exclude a listed repo on its own grounds. Each repo's host is the source of truth for its contents [spec]. For each listed writer, the syncer resolves the DID document, finds `#atproto_pds` and `#atproto_kem`, and pulls the repo.

7.1.2 Incremental sync with listRepoOps

IN THE FAMILY SPACE

Ruth changes her RSVP from interested to going. The family app already holds her space repo as it stood after her first reply. It needs the change, and proof that its copy now matches hers.

REFERENCE

`listRepoOps` [spec]

The family app asks Ruth's PDS what changed since her first RSVP.

Try it 7.2 · Pull Ruth's ops. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/7-sync#try-list-repo-ops

REQUEST

```
GET /xrpc/com.atproto.space.listRepoOps
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
  &repo=did:plc:p4256dosynj6j6tkypz2q2l3
  &since=3n2gtkszes32p
Host: pds-b.example.com
Authorization: Atproto-Space-Credential <credential>
Atproto-Space-Audience: did:plc:p4256dosynj6j6tkypz2q2l3
Signature-Input: sig=("authorization" "atproto-space-
audience");tag="atproto-space-credential"
Signature: sig=:<signature by the credential key>:
Accept-Signature: mac=("atproto-space-
commit";sf);keyid="<ephemeral KEM key>";tag="atproto-space-
commit"
```

HEADERS

`Authorization` The space credential the authority signed, naming the space and the key it is bound to. Appendix C, Token types

`Atproto-Space-Audience` The host the request is for: the repo's DID for repo methods, and otherwise the space's authority. Appendix B, Headers

`Signature-Input` The two headers the binding signature covers. Appendix B, Headers

`Signature` The credential key's signature over those headers, which shows the caller holds the key the credential is bound to. Appendix B, Headers

`Accept-Signature` Asks the host to sign its answer with a MAC, using a key agreed with this ephemeral key. Appendix B, Headers

SAMPLE RESPONSE

```
{
  "ops": [
    {
      "rev": "3n2gtxdkr232p",
      "collection": "community.lexicon.calendar.rsvp",
      "rkey": "3n2gtxdkr222p",
      "cid":
"bafyreiblr6f6oy3zi426nksahddwp4tgsfgedtqk27fyxqtav66kjvwkgu",
      "prev": null,
      "value": {
        "$type": "community.lexicon.calendar.rsvp",
        "subject": {
```

```
          "uri": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family/
did:plc:ejjl2znzqp3ci25dqaslpkk2/
community.lexicon.calendar.event/3n2gsywstc22e",
          "cid":
"bafyreigjlo4wnf5e6inmlkp7ib7pcym5o34vblhnbrf6tij6olzq4hyliu"
        },
        "status": "community.lexicon.calendar.rsvp#going"
      }
    },
    {
      "rev": "3n2gtxdkr232p",
      "collection": "community.lexicon.calendar.rsvp",
      "rkey": "3n2gtkszes22p",
      "cid": null,
      "prev":
"bafyreiautip3hgihuw4ypueftfdiv3iyliieamu5mdv4d4duqhhyetxs7a"
    }
  ],
  "commit": {
    "ver": 1,
    "repo": "did:plc:p4256dosynj6j6tkypz2q2l3",
    "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
    "rev": "3n2gtxdkr232p",
    "data": {
      "$bytes": "eVD7vKgDQrLKfOvN3VZJHYX0HGz158ocKa5oRd3kaYw"
    }
  }
}
```


Each entry is a `com.atproto.space.listRepoOps#opEntry` [spec].

- `prev: null` is a create.
- `cid: null` is a delete.
- With both set, it's an update.
- Entries that share a `rev` came from one atomic batch.
- `value` holds the record. It's left out for deletes, when `excludeValues=true`, and when the record has changed since, so any value you get is current for its path.

The request's `since` is a TID, and `limit` runs from 1 to 1000 with a default of 100 [spec].

Every page carries the repo's current commit and its MAC [spec]. The syncer works through the pages like this:

1. For each op, it updates its copy of the repo and its running LtHash, subtracting the `prev` element and adding the `cid` element.

2. It keeps requesting pages until one comes back without a `cursor`. A full page always has a `cursor`, so the last page may be empty.
3. It compares `sha256(state)` with the last page's commit `data`. If they match, it's in sync. If not, the repo has diverged and the syncer recovers with `getRepo`.

The op log speeds up transport, but it isn't committed data. A host may compact it to a backfill window, and it starts fresh after account migration [spec]. `listRepoOps` answers `RepoNotFound` for a repo that has never been written.

In `atproto-pds`, a request can send both `since` and `cursor`. Both apply, and whichever points later in the log decides where the page starts. A resume point older than the retained log returns the remaining ops and the commit. The fold then won't match, so the syncer falls back to recovery [crates].

7.1.3 Full-state recovery with getRepo

`getRepo` streams the whole repo as a STAR archive with `Content-Type: application/vnd.dasl.star`:

REFERENCE

The STAR archive layout [spec]

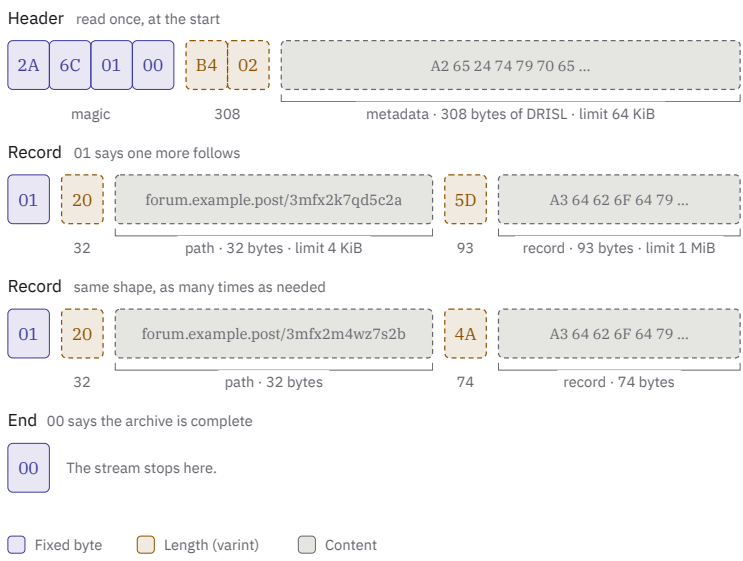

Figure 7.1 · Anatomy of a STAR archive

Anatomy of a STAR archive. Read from top to bottom. This is the order the bytes arrive in.

Header, read once at the start.

1. `2A 6C 01 00` is the magic: four fixed bytes that open every STAR archive.
2. `B4 02` is a length, written as a varint. Here it says 308.
3. `A2 65 24 74 79 70 65 …` is the metadata: 308 bytes of DRISL holding `$type` and the commit. The spec caps it at 64 KiB.

Record, opened by a marker byte.

1. `01` says a record follows.
2. `20` is a length. Here it says 32.
3. `forum.example.post/3mfx2k7qd5c2a` is the path: 32 bytes of UTF-8 in the form `{collection}/{rkey}`. The spec caps it at 4 KiB.
4. `5D` is a length. Here it says 93.
5. `A3 64 62 6F 64 79 …` is the record: 93 bytes of canonical DRISL. The spec caps it at 1 MiB.

Record, the same shape again, as many times as needed.

1. `01` says a record follows.
2. `20` is a length of 32.
3. `forum.example.post/3mfx2m4wz7s2b` is the path.
4. `4A` is a length of 74.
5. `A3 64 62 6F 64 79 …` is the record.

End

1. `00` says the archive is complete. The stream stops here.

The metadata, every path, and every record are preceded by their length in bytes, so the reader always knows how far to read next.

An empty repo has no records, so the end marker follows the metadata directly.

The metadata is a DRISL map with `$type: "atproto-space-repo"` and every commit field. Records come in strict bytewise order of path [spec]. The commit's MAC rides in the response's HTTP signature, not in the metadata.

REFERENCE

Verifying a STAR archive as it streams [spec]

1. Check the metadata's `$type`, `ver`, `repo`, and `space`.
2. Verify the commit MAC, and check that `Atproto-Space-Commit` matches the metadata.
3. Fold each record's `{collection}/{rkey}/{cid}` into a fresh LtHash, refusing any out-of-order or duplicate key.
4. After the `0x00` end byte, compare the digest to `data`.
5. On a match, replace the local copy, deleting records absent from the archive.

Refuse the whole archive on any parse error, non-canonical metadata, ordering problem, MAC mismatch, or digest mismatch.

For a copy that has drifted only slightly, `getLatestCommit` plus `listRecords?excludeValues=true` gives the path-to-CID map. Diff it against the local copy, then `getRecord` just the differing paths [spec]. This trades one large transfer for several small ones.

7.1.4 The read surface

Every read a repo host offers takes `space` and `repo`, and accepts either the repo owner's OAuth session or a space credential [spec]. Besides `getLatestCommit`, `listRepoOps`, and `getRepo`, a syncer can fetch one record with `getRecord`, list records with `listRecords`, and find and fetch blobs with `listBlobs` and `getBlob`. Only the reads that return a commit or a single record carry a MAC: `getLatestCommit`, `getRecord`, `listRepoOps`, and `getRepo`. Those are the answers a syncer can tie to the host that gave them. Parameters, limits, and errors for each read are in Appendix A.

7.2 Staying current

Once your app has a copy, the question becomes how soon it hears about changes. It can poll on a schedule and live with some delay, or register for notifications and learn about a write moments after it lands. Notifications ask more of the app in return. It needs an endpoint the space host can reach, it has to answer a challenge to confirm the registration, and it has to renew the registration before it lapses.

7.2.1 Write notifications

Notifications carry no record data [spec].

When Ruth writes, her PDS sends `com.atproto.space.notifyRepoUpdate` to the space host, with service auth (`lxm=com.atproto.space.notifyRepoUpdate`). It is skipped when the repo belongs to the authority [spec].

REFERENCE

`notifyRepoUpdate` [spec]

Ruth's PDS tells Marcus's space host that her repo advanced.

HTTP

```
POST /xrpc/com.atproto.space.notifyRepoUpdate
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJFUz...
Content-Type: application/json

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "repo": "did:plc:p4256dosynj6j6tkypz2q2l3",
  "repoRev": "3n2gtxdkr232p",
  "data": {
    "$bytes": "eVD7vKgDQrLKfOvN3VZJHYX0HGz158ocKa5oRd3kaYw"
  }
}
```

The repo host retries network failures and transient statuses with exponential backoff and stops on permanent errors. The space host rejects notifications for repos not authorized to write, ignores a `repoRev` that doesn't increase, and never verifies `data`. It answers a `repoRev` too far in the future with `FutureRev`, and an unknown space with `SpaceNotFound` [spec].

The space host records the repo's new state, assigns a new space-wide `spaceRev`, and sends `com.atproto.space.notifySpaceUpdate` to every registered syncer, on a best-effort basis [spec].

7.2.2 Registering for notifications

A syncer registers by including `notify` in `authorizeSpaceSync` [spec]:

REFERENCE

The `notify` object [spec]

Field	Meaning
`endpoint`	A base HTTPS URL the space host calls XRPC methods against
`token`	An opaque secret of at most 512 characters, such as 32 random bytes in base64

- Use a different token for each space and keep it stable across authorizations. A registration is identified by endpoint and token together; the same pair renews it, a different token creates another registration.
- Before creating a new registration, the space host calls `com.atproto.space.confirmRegistration` on the endpoint with `{space, token, challenge}`. The syncer checks that it issued that token for that space and answers `{challenge}`. Otherwise `authorizeSpaceSync` fails with `RegistrationNotConfirmed`.

  Each half stops something different. The token check protects a real syncer: if someone registers its endpoint with a token it never issued, it declines. The echo protects every other service on the internet. Plenty of URLs answer any POST with a 200, but only an endpoint that read the request can send back a fresh challenge, so a space host only delivers notifications to a service that asked for them.
  * A registration stays active while the credential from its latest authorization is valid. To receive notifications continuously, keep reauthorizing with the same `notify`.

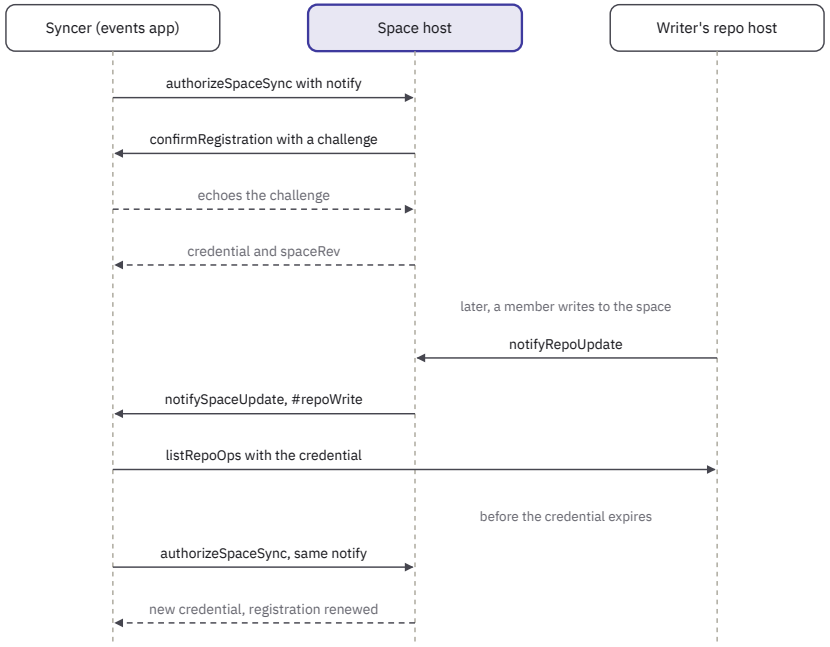

Figure 7.2 · A registration is confirmed once, then renewed by each authorization

Only a new registration triggers `confirmRegistration`. Reauthorizing with the same endpoint and token renews the registration without another challenge.

REFERENCE

`confirmRegistration` [spec]

Marcus's space host checks the family app's endpoint the first time the app registers.

Try it 7.3 · Confirm a registration. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/7-sync#try-confirm-registration

REQUEST

```
POST /xrpc/com.atproto.space.confirmRegistration
Host: family.example
Content-Type: application/json

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "token": "xIFHpilmY0Wlfxq62sqovyfp66i08zazz2dgpsT_38E",
  "challenge": "ij6vvo5y7rblrkvq2lxyyu63su"
}
```

SAMPLE RESPONSE

```
{
  "challenge": "ij6vvo5y7rblrkvq2lxyyu63su"
}
```

The family app recognizes the token as the one it issued for this space and echoes the challenge.

The reference PDS sends 16 random bytes in base32 as the challenge and gives the endpoint 10 seconds to answer. `atproto-pds` sends 32 random bytes in hex and allows the same 10 seconds [crates].

7.2.3 Handling an update

The space host calls the syncer at `POST {endpoint}/xrpc/com.atproto.space.notifySpaceUpdate` with `{space, token, update}`. These calls carry no service auth. The token in the body is the only thing that authenticates them [spec]. A token can leak, so the syncer treats each notification as a reason to go and check, and never deletes anything because of a notification alone.

REFERENCE

`notifySpaceUpdate` [spec]

Marcus's space host tells the family app that Ruth's repo advanced.

HTTP

```
POST /xrpc/com.atproto.space.notifySpaceUpdate
Host: family.example
Content-Type: application/json

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "token": "xIFHpilmY0Wlfxq62sqovyfp66i08zazz2dgpsT_38E",
  "update": {
    "$type": "com.atproto.space.defs#repoWrite",
    "repo": "did:plc:p4256dosynj6j6tkypz2q2l3",
    "repoRev": "3n2gtxdkr232p",
    "data": {
      "$bytes": "eVD7vKgDQrLKfOvN3VZJHYX0HGz158ocKa5oRd3kaYw"
    },
    "spaceRev": "3n2gtxdkr222e",
    "prevSpaceRev": "3n2gtrxvls22e"
  }
}
```

The family app answers with an empty `200` and pulls Ruth's repo afterward.

REFERENCE

Update types [spec]

`update.$type`	Meaning	Syncer action
`com.atproto.space.defs#repoWrite`	A repo advanced: `repo`, `repoRev`, `data`, `spaceRev`, `prevSpaceRev?`	Pull that repo from its host
`com.atproto.space.defs#repoRemoved`	A repo left the writer set: `repo`, `spaceRev`, `prevSpaceRev?`	Stop treating it as part of the space
`com.atproto.space.notifySpaceUpdate#spaceDeleted`	The space was deleted	Reauthorize; drop data only on a `SpaceDeleted` answer
`com.atproto.space.notifySpaceUpdate#registrationExpired`	The registration lapsed: `spaceRev`	Reauthorize with `notify` and catch up

The syncer's `notifySpaceUpdate` handler works in this order:

1. Check the `token` and reject one the syncer didn't issue for that space.

2. Match the update's `$type` in full, and ignore any type it doesn't recognize.
3. Answer quickly, and do the work afterward.

Delivery of `#repoWrite` and `#repoRemoved` is best effort [spec]. The reference PDS gives each call 10 seconds and never retries it, so a slow handler loses updates. `atproto-pds` queues each update in its notify outbox and retries a delivery that fails in a way that might clear, which covers a transport error, a timeout, and the statuses 408, 425, 429, and 5xx other than 501 [crates].

A `#repoWrite` says a repo advanced, but the space host never verifies `data`. If its `repoRev` is no newer than the syncer's copy, there is nothing to do. Otherwise the syncer pulls the repo from its own host with `listRepoOps` from its last `repoRev`, checks the commit MAC, and folds the ops into its LtHash, as in incremental sync above. The new state counts only once that digest matches the host's commit. After a successful pull, the syncer saves the notification's `spaceRev` as its checkpoint. If `prevSpaceRev` is present and differs from the last `spaceRev` it saw, it missed an update and catches up as described below.

A `#repoRemoved` says the authority took a repo out of the writer set. Removal is destructive, so the syncer confirms it with `listWriters` before acting, then stops pulling that repo and stops serving its records as part of the space. Nothing changes on the repo's own host. If its author writes again under the policy, the repo rejoins the writer set and arrives as a `#repoWrite`.

A `#spaceDeleted` says the space is gone. The syncer reauthorizes, and deletes its copy only if `authorizeSpaceSync` answers `SpaceDeleted`. Deletion notices are retried, so the same one can arrive more than once, and handling it twice must be harmless.

A `#registrationExpired` arrives in place of the first update after the syncer's registration lapsed, and carries that update's `spaceRev`. Until the syncer reauthorizes, the space host sends it nothing except a deletion notice [spec]. The syncer reauthorizes with the same `notify`, then catches up from its checkpoint with `listWriters`. The host retries expiry and deletion notices with exponential backoff for at least 72 hours, and may drop a registration it still can't reach [spec]. In the reference PDS, an expiry notice is skipped if the syncer renewed in the meantime, and a pending deletion notice replaces a pending expiry notice.

A syncer that wants to stop receiving notifications stops reauthorizing and answers the expiry notice with a non-retryable error [spec]. An authorization that denies the syncer access also withdraws the registration it carries.

`atproto-pds` keeps one registration per (space, endpoint, token) until its credential expires. It retries expiry and deletion notices after 1 minute, 10 minutes, 1 hour, and 6 hours, then every 12 hours with 50 to 100 percent jitter, for at least 72 hours. It unregisters a syncer that refuses its expiry notice, and keeps a registration in place while the managing app is undecided or the authority is suspended [crates].

On the syncer side, `atproto-space-client` makes one design choice worth copying [crates]. Rather than generating a random notify token for each space and storing it, `notify_token(app_secret, &space)` derives the token as an HMAC-SHA256 of the space URI under a single app secret. That gives every space its own token, keeps each token stable across reauthorizations, and lets `verify_notify_token` recognize an incoming token by recomputing it, so the webhook handler needs no token table. The rest of the client maps directly onto the protocol. `SpaceSubscription` takes a `NotifyRegistration { endpoint, token }`, `ConfirmRegistrationInput::confirm()` answers the registration challenge, and `SpaceUpdate` models the update union with an `Unknown` arm, so new update types are ignored rather than rejected. The crate also recommends letting a registration lapse and reauthorizing on `#registrationExpired`, which costs less than reauthorizing every few minutes to keep it alive.

7.2.4 Catching up

A syncer detects a missed notification from a gap between `prevSpaceRev` and the last `spaceRev` it saw, or by comparing the `spaceRev` returned by `authorizeSpaceSync` with its checkpoint. It catches up by calling `listWriters` with `cursor` set to that checkpoint [spec]. If its last seen `spaceRev` is more than 72 hours old, removed repos it missed may have been collected, so it lists the whole writer set instead and drops any repo that is missing or removed [spec].

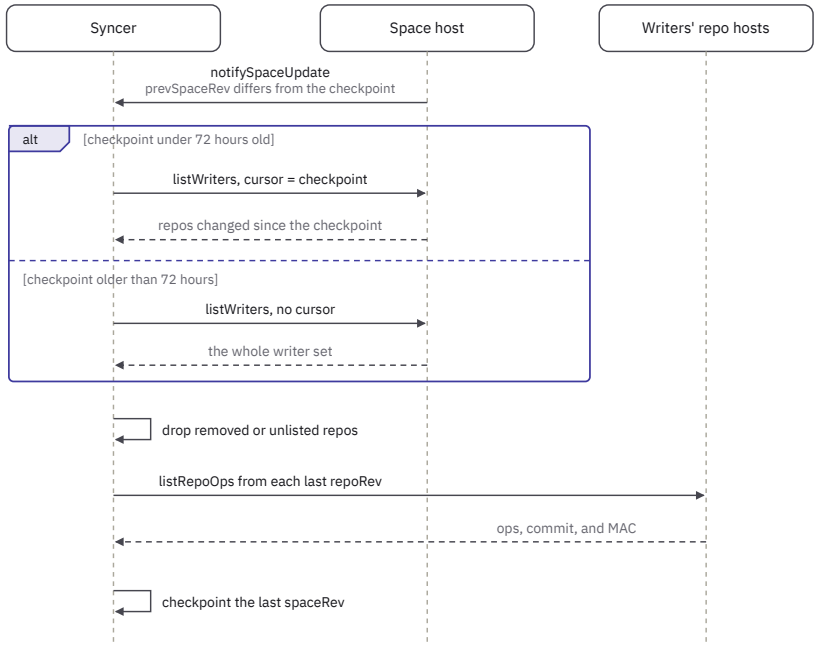

Figure 7.3 · A gap sends the syncer to listWriters, from its checkpoint or from scratch

In `atproto-space-client`, `SpaceCheckpoint::observe` tracks revisions, `catch_up` returns `CatchUp { entries, complete }` (with `complete` meaning drop every repo not named), and `REMOVAL_RETENTION` is 72 hours. `pull_repo` brings one repo up to date, applying the op log and checking the last page's commit, falling back to the archive on a mismatch, and treating `RepoNotFound` as empty. It returns `RepoPull::{Changes, Snapshot, Empty}` [crates].

7.2.5 Removing a writer

IN THE FAMILY SPACE

> Marcus changes Jake's access to read-only. Jake's bookmark is still in his space repo on his PDS, and the family app still has a copy of it. What should the app do with that copy?

The authority drops an account from the writer set with `removeWriter` (`manage=update`, input `repo`). A removal also happens when `removeMembers` removes a member or `putMembers` revokes write access. Removals are sequenced like writes: the space host advances `spaceRev`, sends `#repoRemoved`, and lists the repo as `#repoRemoved` until it writes again. A later authorized write re-adds it with a `#repoWrite` [spec]. The repo's records

stay on its host. The space host may drop empty and removed repos from the writer set after at least 72 hours, with no notification [spec].

What an app does with its copy of a removed writer's records is its own decision, and it shapes how the app behaves for its users. The spec asks only that the app stop treating the repo as part of the space. The safest choice is to hard delete that writer's records once the removal is confirmed. A better alternative is to soft delete them, so they disappear from view at once, and garbage collect them after a set period. While they wait, the app can show a tombstone in their place, so readers can see that something was there even though it is no longer available.

7.3 Blobs

IN THE FAMILY SPACE

> Elena's post about Theo has two photos. The post record arrived with her space repo, and the photos did not.

Blobs stay on the author's repo host, next to the records that reference them, and are fetched separately by CID [spec].

STAR archives don't include blobs, so a syncer that recovers a repo with `getRepo` still fetches its blobs one at a time [spec].

7.3.1 Uploading

The reference PDS takes space blobs through the same `com.atproto.repo.uploadBlob` as public ones. An uploaded blob starts out untethered, which means the host has stored it but no record refers to it yet. It becomes permanent once a space record references it. When an update or delete drops a record's reference, the blob is released from that record.

In `atproto-pds`, a blob that no public or space record references is deleted after a grace period, 24 hours by default [crates].

7.3.2 Listing blobs

`listBlobs` returns the CIDs that a repo's records in the space reference, each once [spec].

REFERENCE

`listBlobs` [spec]

The family app asks Elena's PDS which blobs her records in the family space reference, with Elena as the credential's audience.

Try it 7.4 · List Elena's blobs. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/7-sync#try-list-blobs

REQUEST

```
GET /xrpc/com.atproto.space.listBlobs
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
  &repo=did:plc:53ni6w7okqrxvjxezmopejoo
  &limit=500
Host: pds-a.example.com
Authorization: Atproto-Space-Credential <credential>
Atproto-Space-Audience: did:plc:53ni6w7okqrxvjxezmopejoo
Signature-Input: sig=("authorization" "atproto-space-
audience");tag="atproto-space-credential"
Signature: sig=:<signature by the credential key>:
```

HEADERS

`Authorization` The space credential the authority signed, naming the space and the key it is bound to. Appendix C, Token types

`Atproto-Space-Audience` The host the request is for: the repo's DID for repo methods, and otherwise the space's authority. Appendix B, Headers

`Signature-Input` The two headers the binding signature covers. Appendix B, Headers

`Signature` The credential key's signature over those headers, which shows the caller holds the key the credential is bound to. Appendix B, Headers

SAMPLE RESPONSE

```
{
  "cids": [
"bafkreicy6rib4mitvvzbsmj3rktg2fpihkfhfomngc4kynwpvjp45bioya",
"bafkreigne4u5q3h5hdcbhusc5osdwybrlkdpdl7222ufbklgo7swix4ore"
  ]
}
```

`limit` runs from 1 to 1000 and defaults to 500. The reference PDS sorts the CIDs in ascending order and returns a `cursor`, the last CID on the

page, only when the page is full. Instead of listing, a syncer can collect the blob refs from the records it already holds [spec].

7.3.3 Fetching a blob

`getBlob` takes `space`, `repo`, and `cid`, and returns the blob as originally uploaded [spec]. The credential signature from the listing works here too, since it covers only the credential and the audience.

REFERENCE

`getBlob` [spec]

The family app fetches one of Elena's two photos.

HTTP

```
GET /xrpc/com.atproto.space.getBlob
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
  &repo=did:plc:53ni6w7okqrxvjxezmopejoo
  &cid=bafkreigne4u5q3h5hdcbhusc5osdwybrlkdpdl7222ufbklgo7swix4ore
Authorization: Atproto-Space-Credential
eyJhbGciOiJFUzI1NksiLCJ0eXAiOi...
Atproto-Space-Audience: did:plc:53ni6w7okqrxvjxezmopejoo
Signature-Input: sig=("authorization"
  "atproto-space-audience");tag="atproto-space-credential"
Signature: sig=:Q/
Ya6HSYi45IB5C1Of0GGLVVUAflbUmE033JyiRxNZgn0nf0rEC5IJe30snJsRUSa85igHDoeBdLhJutfXL
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 1702
Content-Disposition: attachment;
filename="bafkreigne4u5q3h5hdcbhusc5osdwybrlkdpdl7222ufbklgo7swix4ore"
Content-Security-Policy: default-src 'none'; sandbox
X-Content-Type-Options: nosniff
```

A blob response carries no MAC. Verify it by hashing the body with SHA-256 and comparing the digest with the CID, which uses the raw codec. The reference PDS serves the body as untrusted content, as the headers show, and an app that re-serves a blob should do the same. It answers `BlobNotFound` for a missing blob, and also for one the account stores but no record in that space references, so a credential for one space can't probe for blobs from another.

7.4 Why this scales

Sync is lighter than public sync. The public event stream carries every published record along with the MST nodes and signatures that prove cryptographically that its author wrote it. A space repo has none of that proof material to move. Load grows with the number of syncing applications, not the number of users, since each app pulls each repo once and fans out from its own copy. And because every sync call requires a credential, a host can rate-limit by credential holder [spec].

The cost lands on the apps. With no relay to share, every app runs its own syncer, and a first sync touches every writer's PDS.

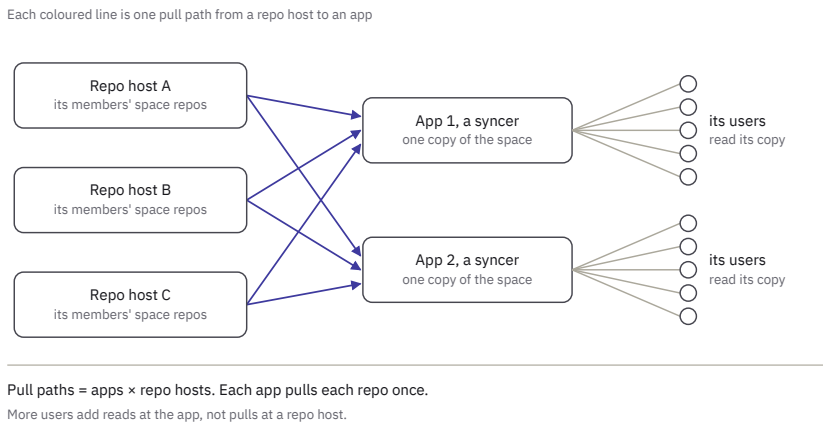

Figure 7.4 · Sync load grows with apps and repo hosts, not with users

A third app adds three pull paths, one per repo host, and a fourth repo host adds two, one per app. A thousand more users of App 1 add none.

7.5 Common questions

Q Is registering for notifications required?

Registering is optional. An app that can tolerate delay can call `listWriters` and `listRepoOps` on its own schedule. Notifications only make updates arrive sooner.

Q What if my app misses a notification?

It catches up. Each notification carries the previous space revision, so a gap is easy to spot, and `listWriters` with your last checkpoint returns everything that changed since. If the checkpoint is more than 72 hours old, list the whole writer set instead.

Q Does the space host send record data?

It sends only metadata, such as which repo advanced and its new revision. Records always come from each writer's own PDS, checked against that PDS's commit.

Chapter 8 · Policies

Policies are the core of how people interact with a space because policies are what enforces a space's data perimeter. Stored with the space as a typed object, a policy decides whether an identity and app may *read* (get a credential), and whether an account may *write* (be tracked in the writer set, with syncers notified). Adding or removing a member is a common operation and a relatively straightforward one. It's the policy that determines what adding a member means and what the side effects are.

A personal bookmark collection under personalspace has no members to add. It's your own space, and no one else will use it. With memberspace, adding and removing members has all kinds of side effects. A member can be given read access, write access, or both. The space can also publish its policy as a record, and a record for each member.

With managedspace you are effectively handing all of the decision making to a third party. The managing app determines what access any given account has. Every PDS must support these three baseline policies and may support others [spec]. They won't fit every community, and it's relatively easy to create a new one.



If you're building an app

The policy sets what a member can do once they are added to a space. If your app is a personal utility, where the user who created the space is the one person who will ever read and write it, use personalspace.

If your app spans multiple users, now or in the future, pick memberspace. It covers a lot of different scenarios and is a really good default. Managedspace offers some features, but it comes with a lot of caveats. Use it if you really want to take data perimeter decisions away from the space host, and proceed with caution.

The policy controls who gets a credential and whose writes are synced. What each of your users sees inside your app is still your decision, covered in Chapter 10.

8.1 The lifecycle methods

IN THE FAMILY SPACE

> Marcus sets up a space for his family in the family app. He is its authority, and the space lives on his PDS. He chooses a member list, and he turns on the two flags that let the family see the policy and who else belongs.

An authority creates, changes, reads back, and deletes its space with four methods under `com.atproto.space.*`. They work the same way under every policy [spec], and each one takes the authority's OAuth session with the matching `manage` grant from Chapter 9.

Creating a space happens once, when someone sets one up, and the caller becomes its authority. The call needs `manage=create` for the space type. `skey` is optional, and the host generates a TID when it's omitted. The response is the new space's URI. A call that names a deleted space's type and key recreates it under the same URI, with the new policy and no members [crates]. A second call with the same type and key fails with `SpaceAlreadyExists`, and a policy type the host doesn't support fails with `UnsupportedPolicy` [spec].

REFERENCE

`createSpace` [spec]

Marcus creates the family space on his PDS.

Try it 8.1 · Create the family space. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/8-policies#try-create-space

REQUEST

```
POST /xrpc/com.atproto.space.createSpace
Host: pds-a.example.com
Authorization: DPoP <access token>
DPoP: <proof JWT>
Content-Type: application/json

{
  "spaceType": "com.example.familyUpdates",
  "skey": "our-family",
  "policy": {
    "$type": "com.atproto.memberspace.policy",
    "publicRead": false,
```

```
    "publicWrite": false,
    "publishPolicy": true,
    "publishMembers": true
  }
}
```

HEADERS

`Authorization` The session's access token, bound to a key only the client holds. Appendix G, OAuth requests

`DPoP` A proof signed with that key for this method and URL, so a copied token can't be replayed. Appendix G, OAuth requests

SAMPLE RESPONSE

```
{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family"
}
```

Changing a space's rules is an `updateSpace` call, which needs `manage=update`. The new policy replaces the old one wholesale, so the request carries every field, not only the one that changes. Keeping the same `$type` is a configuration change. Switching to a different `$type` migrates the space, and the old policy's state, such as a member list, is discarded. It doesn't come back if the space later migrates back. A field the new policy leaves out takes its default, and the writer set isn't re-evaluated. `atproto-pds` revokes early any outstanding credential the new policy wouldn't issue, and brings the published records in line, so turning `publishMembers` off withdraws the member records [crates].

REFERENCE

`updateSpace` [spec]

Marcus stops publishing the family's member list.

HTTP

```
POST /xrpc/com.atproto.space.updateSpace
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "policy": {
```

```
    "$type": "com.atproto.memberspace.policy",
    "publicRead": false,
    "publicWrite": false,
    "publishPolicy": true,
    "publishMembers": false
  }
}
```

HTTP

```
HTTP/1.1 200 OK
Content-Length: 0
```

The lexicon declares no output, so success is a `200` with an empty body [spec].

Reading a space's current policy back is a `getSpace` call, such as when the family app opens its settings screen for Marcus. Only the authority can make it. Members who want to see the policy rely on the `publishPolicy` flag instead.

REFERENCE

`getSpace` [spec]

Marcus's family app loads the family space's settings.

Try it 8.2 · Load the family space's settings. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/8-policies#try-get-space-settings

REQUEST

```
GET /xrpc/com.atproto.space.getSpace
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
Host: pds-a.example.com
Authorization: DPoP <access token>
DPoP: <proof JWT>
```

HEADERS

`Authorization` The session's access token, bound to a key only the client holds. Appendix G, OAuth requests

`DPoP` A proof signed with that key for this method and URL, so a copied token can't be replayed. Appendix G, OAuth requests

SAMPLE RESPONSE

```
{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "policy": {
    "$type": "com.atproto.memberspace.policy",
    "publicRead": false,
    "publicWrite": false,
    "publishPolicy": true,
    "publishMembers": false
  }
}
```

`atproto-pds` writes out every default and leaves `allowedApps` out when it's unset, since its absence means any app may read [crates].

Removing a space is a `deleteSpace` call, which needs `manage=delete`. One call starts a longer sequence. The host stops issuing credentials, deletes the authority's own records in the space, and notifies every registered syncer, and each syncer then drops its copy. Chapter 10 walks through that sequence under Deleting a space. Members' own repo hosts aren't told, and their records in the space become unreadable to anyone but their own accounts. Deleting a space that's already deleted answers `200` and repeats none of this [crates].

REFERENCE

`deleteSpace` [spec]

Marcus deletes the family space.

HTTP

```
POST /xrpc/com.atproto.space.deleteSpace
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family"
}
```

HTTP

```
HTTP/1.1 200 OK
Content-Length: 0
```

As with `updateSpace`, the lexicon declares no output [spec].

Every method that works on an existing space refuses a caller who isn't the authority with `NotAuthorized`, and an unknown space with `SpaceNotFound`. That covers `updateSpace`, `deleteSpace`, `getSpace`, `removeWriter`, and `exportSpaceMeta` [spec]. `atproto-pds` sends `NotAuthorized` with status 400 [crates].

Three rules hold under every policy [spec]:

- The authority always reads and writes its own space.
- Read access decides whether a credential is minted.
- Write access decides only whether the space host *tracks* a writer and notifies syncers. It never blocks a write into an account's own repo, and applications still choose which records to show.

8.2 personalspace

`com.atproto.personalspace.policy` is for data that belongs to one person: private bookmarks, mutes, draft posts that aren't ready to publish, records scheduled to go out at a set time and not before, a personal journal, and snippets saved from the web. Only the authority's own account, through the apps it authorizes, may read, and no other writers are tracked. It has no fields and no methods [spec]. Choosing it over a memberspace with no members removes the risk of someone being added by accident, since there is no member list to add them to.

JSON

```
{
  "$type": "com.atproto.personalspace.policy"
}
```

IN JAKE'S BOOKMARKS SPACE

> Jake saves a recipe link in the family space. He keeps the same bookmark in a bookmarks space of his own, where he is the authority and no one else is tracked as a writer.

A bookmarks app asks for `space:com.example.bookmarks` (authority defaults to `self`), creates `at:space/did:plc:brthxhhoomvrib2avauses35/com.example.bookmarks/self` on first run, and writes records there. Each of Jake's apps needs its own `read` grant to reach it.

8.3 memberspace

IN THE FAMILY SPACE

> Marcus adds Elena, Ruth, Nora, and Jake, who can each read and post. He adds Erin as well. She can read what the family shares, and nothing she writes will be tracked.

`com.atproto.memberspace.policy` decides by a member list kept by the authority's PDS [spec].

REFERENCE

Memberspace policy fields [spec]

Field	Default	Effect
`publicRead`	false	Anyone may read; no delegation token needed
`publicWrite`	false	Track writes from accounts not on the member list
`allowedApps`	any	Mint only for these attested `client_id` values (URIs)
`publishPolicy`	false	Publish the policy as a record
`publishMembers`	false	Publish one record per member

Each member is a `com.atproto.memberspace.defs#member` with required `did`, `read`, and `write` [spec]. The list lives outside the policy object, so it survives configuration changes.

A member's `read` and `write` are independent permissions [spec]. Each works as the three rules above describe, so a member can hold either without the other. A read-only member follows the space without being synced as a writer. A write-only member, with `write: true` and `read: false`, has their records synced to everyone who can read, but their apps can't read anyone else's repo, which suits a submissions or tip-line space, where members post and only the authority and readers see everything. Either way, a member can always read their own space repo on their own PDS with OAuth.

Adding members, or changing what existing members can do, is a `putMembers` call, which needs `manage=update`. Each entry sets `read` and `write` together, so upgrading a read-only member to read and write means sending `true` for both. A call takes at most 100 entries and applies them in one commit, so larger changes go in batches of 100 [spec]. In `atproto-pds`, a DID listed twice takes its last entry, a member given `write: false` leaves the writer set, and a member who loses `read` loses their outstanding credentials too [crates].

REFERENCE

`putMembers` [spec]

Marcus adds the family in one call, with Erin as a read-only member.

HTTP

```
POST /xrpc/com.atproto.memberspace.putMembers
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "members": [
    {
      "did": "did:plc:53ni6w7okqrxvjxezmopejoo",
      "read": true,
      "write": true
    },
    {
      "did": "did:plc:p4256dosynj6j6tkypz2q2l3",
      "read": true,
      "write": true
    },
    {
      "did": "did:plc:ashwgt3d5ntfxmlcfyizpu2g",
      "read": true,
      "write": true
    },
    {
      "did": "did:plc:brthxhhoomvrib2avauses35",
      "read": true,
      "write": true
    },
    {
      "did": "did:web:erin.example",
      "read": true,
      "write": false
    }
  ]
}
```

HTTP

```
HTTP/1.1 200 OK
Content-Length: 0
```

IN THE FAMILY SPACE

Marcus changes Jake to read-only, and later takes him off the member list. Jake's bookmark stays in his own space repo on his PDS. The space host stops listing him as a writer.

REFERENCE

`putMembers` [spec]

Marcus changes Jake to read-only. The same call with `"write": true` would upgrade him back.

HTTP

```
POST /xrpc/com.atproto.memberspace.putMembers
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "members": [
    {
      "did": "did:plc:brthxhhoomvrib2avauses35",
      "read": true,
      "write": false
    }
  ]
}
```

HTTP

```
HTTP/1.1 200 OK
Content-Length: 0
```

Taking someone off the list is a `removeMembers` call, with the same `manage=update` grant and the same limit of 100. A DID that isn't a member is ignored [spec], writer-set entry and all. Each removed member leaves the list and the writer set together, and `atproto-pds` revokes their credentials unless the space is `publicRead` [crates]. Their records stay in their own space repo.

REFERENCE

`removeMembers` [spec]

Marcus removes Jake.

HTTP

```
POST /xrpc/com.atproto.memberspace.removeMembers
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...

{
  "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
  "dids": [
    "did:plc:brthxhhoomvrib2avauses35"
  ]
}
```

HTTP

```
HTTP/1.1 200 OK
Content-Length: 0
```

Reading the member list back is a `listMembers` call. Only the authority can make it, and the authority itself never appears in the list. It pages with `limit`, from 1 to 1000 with a default of 100, and `cursor` [spec]. `atproto-pds` sorts members by DID and uses the last DID on a page as the cursor [crates]. Members who want to see the list rely on the `publishMembers` flag instead.

REFERENCE

`listMembers` [spec]

Marcus's family app loads the member list after Jake's removal.

HTTP

```
GET /xrpc/com.atproto.memberspace.listMembers
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
  &limit=100
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...
```

JSON

```
{
  "members": [
    {
      "did": "did:plc:53ni6w7okqrxvjxezmopejoo",
      "read": true,
      "write": true
    },
    {
      "did": "did:plc:ashwgt3d5ntfxmlcfyizpu2g",
      "read": true,
```

```
      "write": true
    },
    {
      "did": "did:plc:p4256dosynj6j6tkypz2q2l3",
      "read": true,
      "write": true
    },
    {
      "did": "did:web:erin.example",
      "read": true,
      "write": false
    }
  ]
}
```

`atproto-pds` includes `cursor` only when more members remain, so this response has none. Its member store also returns at most 100 members a page, whatever `limit` asks for [crates].

All three methods declare `SpaceNotFound`, `NotAuthorized`, and `PolicyMismatch`, which means the space isn't a memberspace. `listMembers` adds `InvalidCursor` [spec].

REFERENCE

The memberspace access decision [spec]

TEXT

```
canRead(user, app):
  if allowedApps set and app not in allowedApps: deny
  if publicRead: allow
  if user is none: deny
  if user == authority: allow
  if members[user].read: allow
  deny

canWrite(user):
  if user == authority: allow
  if user in members: return members[user].write
  if publicWrite: allow
  deny
```

With `publicRead`, no delegation token is required, but `allowedApps` still applies, so a client attestation from a listed app is required when the list is set. Without `allowedApps`, any app may read and no attestation is needed, so public clients work. `allowedApps` applies only to reads, because a write notification authenticates the writer's repo host, not the app [spec].

When `removeMembers` removes a member, or `putMembers` revokes a member's write access, the authority should also remove that account from the writer set, and the removal is announced to syncers. Under `publicWrite`, a removed writer is tracked again as soon as it writes, unless it is on the member list with `write: false` [spec].

These three configurations cover a lot of use cases. The following examples show some of the ways the read and write flags can be used as common patterns. Each shows the body an organizer sends to create a space and manage a member list, and also describes what it means for readers and writers of those example spaces. They're illustrative, so the people in them don't appear in Appendix F.

8.3.1 A private group

Alice (`did:plc:ldianl3maiaomvmvyyt3xzhh`) runs a small planning committee. Bob (`did:plc:7gortyhgiwmpn47qdu6h2bhc`) helps run it, and Carol (`did:plc:5uf6dmcz5zmcw3evvgwl5jx4`) follows along. A private group leaves every flag at its default and lists its members:

JSON

```
{
  "spaceType": "com.example.group",
  "skey": "planning-committee",
  "policy": {
    "$type": "com.atproto.memberspace.policy"
  }
}
```

JSON

```
{
  "space": "at:space/did:plc:ldianl3maiaomvmvyyt3xzhh/
com.example.group/planning-committee",
  "members": [
    {
      "did": "did:plc:7gortyhgiwmpn47qdu6h2bhc",
      "read": true,
      "write": true
    },
    {
      "did": "did:plc:5uf6dmcz5zmcw3evvgwl5jx4",
      "read": true,
      "write": false
    }
  ]
}
```

Only Alice, Bob, and Carol get credentials, so only their apps can read the group. Bob's writes are tracked and reach syncers. Carol reads but

isn't synced as a writer. With `publishPolicy` and `publishMembers` off, nothing about the group shows up as a record in Alice's repo.

8.3.2 An announcement space

Alice also runs an announcements space, where Bob posts updates for anyone to read. An announcement space opens reads to everyone and keeps writing to its members:

JSON

```
{
  "spaceType": "com.example.group",
  "skey": "announcements",
  "policy": {
    "$type": "com.atproto.memberspace.policy",
    "publicRead": true,
    "publishPolicy": true
  }
}
```

JSON

```
{
  "space": "at:space/did:plc:ldianl3maiaomvmvyyt3xzhh/
com.example.group/announcements",
  "members": [
    {
      "did": "did:plc:7gortyhgiwmpn47qdu6h2bhc",
      "read": true,
      "write": true
    }
  ]
}
```

Any app gets a credential without a delegation token, so anyone can follow along. Only Alice and Bob are tracked as writers. Anyone else can still write records into their own repo for this space, but the space host never lists them, so syncers never pull them. Setting `allowedApps` would limit reading to attested apps, and publishing the policy lets clients see that the space is public before they ask.

8.3.3 An open guestbook

Finally, Alice opens a guestbook that anyone can sign, and Mallory (`did:plc:3fuyksp4k46rff4lht3ihb3y`) is one of the people who can. An open guestbook opens both reading and writing, and needs no member list:

JSON

```
{
  "spaceType": "com.example.group",
  "skey": "guestbook",
```

```
  "policy": {
    "$type": "com.atproto.memberspace.policy",
    "publicRead": true,
    "publicWrite": true
  }
}
```

Anyone can read, and any account that writes is tracked and synced. Moderation is handled after the fact. If Mallory were to post spam, Alice removes her ability to write with `removeWriter`:

JSON

```
{
  "space": "at:space/did:plc:ldianl3maiaomvmvyyt3xzhh/
com.example.group/guestbook",
  "repo": "did:plc:3fuyksp4k46rff4lht3ihb3y"
}
```

A removed writer is tracked again as soon as it writes, unless it is on the member list with `write` set to false [spec]. To keep Mallory out, Alice also adds her as a member who can read but not write:

JSON

```
{
  "space": "at:space/did:plc:ldianl3maiaomvmvyyt3xzhh/
com.example.group/guestbook",
  "members": [
    {
      "did": "did:plc:3fuyksp4k46rff4lht3ihb3y",
      "read": true,
      "write": false
    }
  ]
}
```

8.4 managedspace

`com.atproto.managedspace.policy` delegates access decisions to an external managing app, so access can depend on state the PDS doesn't hold: a paid subscription, a follow, a ticket [spec].

Choosing managedspace hands the space's access decisions, and with them its data perimeter, to a third party. That can be the right call for some situations. A managing app can be a specialized service with its own deployment and resource footprint, so the authority's PDS doesn't carry that work. It can also offer policies and features the PDS or space host doesn't support, which bridges the gap for apps that want them.

The trade-offs are real, though. If the managing app is slow or prone to outages, the space is unavailable to apps and users while it's down. The authority also gives up some control over important metadata about the space. Most important, the managing app can let people in without the authority's consent, including itself. Weigh these before delegating to a third party, and before building managing-app endpoints for others to use with your applications.

JSON

```
{
  "$type": "com.atproto.managedspace.policy",
  "managingApp": "did:web:newsletter.example#newsletter_app",
  "publishPolicy": true
}
```

`managingApp` is a `did#serviceId` reference. The managing app serves `com.atproto.managedspace.checkAccess`, and the authority calls it with service auth issued as the authority (`lxm=com.atproto.managedspace.checkAccess`) [spec].

REFERENCE

`checkAccess` [spec]

A newsletter author's PDS asks the managing app about one reader.

Try it 8.3 · Ask the managing app. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/8-policies#try-check-access

REQUEST

```
GET /xrpc/com.atproto.managedspace.checkAccess
  ?space=at:space/did:plc:ldianl3maiaomvmvyyt3xzhh/
com.example.newsletter/members
  &repo=did:plc:ybwcjf2zkz3yfyb6mo2z7cjf
  &clientId=https://newsletter.example/client-metadata.json
Host: newsletter.example
Authorization: Bearer <service auth JWT from the authority>
```

HEADERS

`Authorization` A service auth token the caller's PDS signed for this one method and audience. Appendix G, Tokens

SAMPLE RESPONSE

```
{
  "read": true,
```

```
  "write": false
}
```

The authority passes the space, the identity being checked as `repo`, and the attested `client_id` if the request presented one. A check prompted by a write carries no client, since `notifyRepoUpdate` doesn't identify the app [spec].

`read` and `write` come back as independent answers. A paid newsletter falls out naturally, since subscribers read and only the author writes. A follower-gated space checks the follow graph instead.

In `atproto-pds`, a request without a delegation token asks with no `repo`. A managing app should answer `read: true` to that check only for spaces meant to be publicly readable. If the managing app can't be reached, returns an error, or sends malformed output, the request fails with `502 UpstreamFailure` rather than a denial. The `managingApp` value must be an absolute `did:plc` or `did:web` reference with a fragment, and the server checks it on create, update, and import [crates].

8.5 Publishing configuration to readers

IN THE FAMILY SPACE

> Nora wants to know who else can see what she posts. Only Marcus can ask his PDS for the member list. The family app shows it to her anyway, because Marcus's space repo holds a record for each member.

By default a space's policy and member list are visible only to the authority, through `getSpace` and `listMembers`. `memberspace` and `managedspace` carry a `publishPolicy` flag, and `memberspace` also carries `publishMembers` [spec]. When set, the authority's PDS writes that state as records in the authority's own repo in the space.

Not every space needs that transparency, and leaving the flags off is a normal choice. In a memberspace, the published records are extra data the space may not want or need. A space can also be built so that members don't know who the other members are, such as a support group or a tip line.

REFERENCE

Published configuration records [spec]

Flag	Collection	Record key	Record
`publishPolicy`	`com.atproto.space.policy`	`self`	`policy`, holding the current policy object
`publishMembers`	`com.atproto.memberspace.member`	the member's DID	`read`, `write`

With both flags set, Marcus's repo in the family space holds a policy record and a record for each member, such as these two, at `com.atproto.space.policy/self` and `com.atproto.memberspace.member/did:plc:53ni6w7okqrxvjxezmopejoo` [spec]:

JSON

```
{
  "$type": "com.atproto.space.policy",
  "policy": {
    "$type": "com.atproto.memberspace.policy",
    "publicRead": false,
    "publicWrite": false,
    "publishPolicy": true,
    "publishMembers": true
  }
}
```

JSON

```
{
  "$type": "com.atproto.memberspace.member",
  "read": true,
  "write": true
}
```

The policy record wraps the policy in a `policy` field rather than being the policy object itself, so its `$type` stays `com.atproto.space.policy` whichever policy the space uses.

The stored configuration stays the source of truth, and the PDS keeps the records in step. It rejects direct writes to those collections through repo CRUD, and `clearRecords` leaves them in place. Readers trust these records only in the authority's repo, since any member could write look-alikes into their own [spec]. Members who have written can be enumerated through the writer set even when the member list is unpublished.

8.6 Other policies

The baseline policies are neither the only permitted ones nor privileged ones [spec]. Any policy defines a `policy` object with its own `$type`, its access rules, its configuration, and any additional XRPC methods, by convention in the policy's own namespace. A host that receives a `$type` it doesn't implement answers `UnsupportedPolicy`, so clients can feature-detect.

New policies are expected, and writing one is a modest amount of work. Communities come in different shapes, and a policy is where a community states what its data perimeter allows. A policy outside the baseline is a full participant in the protocol, and it can be implemented by a PDS or by a space host built for the purpose [spec].

Off the Clock, a members-only community application, is one example. Its community is a hyperspace, with a primary space that has its own members and inner spaces for smaller groups, and a member of an inner space has to be a member of the space above it first. Memberspace keeps each space's list separate, with no way to make one list depend on another. A managedspace whose managing app checks the space above could do the job, but it would trust that app to get the check right every time. A policy of its own keeps the check and its validation in the space host that enforces the policy. The protocol gives no meaning to one space sitting beneath another, so the policy is where the host learns that relationship. Off the Clock defines `us.off-the-clock.subspace.policy` for this, and `atproto-pds` implements it [crates].

An inner space names the space above it in its policy, in the `parent` field. A space that names none admits members directly. An inner space admits members of the space above it, and `parentAccess` sets what each of those members gets there without an entry of their own. When a member leaves or is removed from a space, the host removes their entries from every inner space beneath it, and from those spaces' writer sets, in the same change [crates].

The cost is portability. Every PDS implements the baseline policies [spec]. A space under a custom policy needs a host that implements that policy, which narrows where its authority's account can live. A practical answer is to make the community its own identity, as the space authority, and keep that identity's account on a host that supports the policy. The admins and members are then free to keep their accounts anywhere.

REFERENCE

`us.off-the-clock.subspace.policy` fields [crates]

Field	Default	Effect
`parent`	none, so no space is above this one	The space above this one, under the same authority and the same policy
`parentAccess`	`read`	What each member of the space above gets without an entry: `read`, `write` (which includes read), or `none`
`publicRead`	false	Anyone may read, with or without a delegation token
`allowedApps`	the list of the space above, or any app when there is none	Mint credentials for these attested `client_id` values
`publishPolicy`	false	Publish the policy as a `com.atproto.space.policy` record
`publishMembers`	not published	A space under the same authority, written `<spaceType>/<skey>`, that receives the member entries as records

A member entry carries `read`, `write`, and `admin`, and an admin manages the member list of that one space. The policy's methods live in its own namespace, `us.off-the-clock.subspace.*`. They are `putMembers`, `removeMembers`, `listMembers`, `listSubspaces`, and `leave` [crates].

8.7 Common questions

Q Is there a member list my app can read?

If the authority sets `publishMembers`, its repo in the space holds one record per member, and any reader of the space can see them. Otherwise only the authority can call `listMembers`. The writer set is a different list, the accounts that have written, and `listSpaces` is different again, the spaces where the caller's own account keeps a repo or is the authority.

Q Can someone write to a space without being able to read it?

Yes, under memberspace. A member with `write: true` and `read: false` is a write-only member, as described under memberspace. Under managedspace, the managing app can answer `checkAccess` the same way.

Q Can a space change policies?

Yes. `updateSpace` with a policy of a different `$type` migrates the space. The old policy's state, such as its member list, is discarded.

Chapter 9 · OAuth and consent

An app gets access to a person's spaces through OAuth, which makes OAuth a practical requirement for any app that works with spaces. On the consent screen, the person is asked which spaces the app may reach (by authority, type, and key), what it may do to their records (`action` and `collection`), and whether it may manage the spaces themselves (`manage`). All of that is carried by the `space:` scope. The app has to scope it carefully. Asking for too little cuts the app off from what it needs, and asking for too much puts both the app and the person at risk.



If you're building an app

Your app asks for space access in its OAuth scopes, and users see that request on the consent screen. Think critically about what your app needs, and be aware of accidental OAuth scope permission creep. The `authority` parameter defaults to the user's own identity, so an app that reads spaces run by someone else asks for `authority=*` or names the authority. Read access always covers the whole space, while writes can be narrowed to particular collections.

Consider a session upgrade checkpoint for the parts of your app that need more. A person signs in with a general set of `space:` permissions for everyday use. When they open an admin or moderation area, the app asks them to authenticate again with the expanded permissions those tools need. The clean way to do this is to treat the upgraded session as a separate app with its own `client_id`, so the person and the app can both tell the two sessions apart and track each one.

9.1 Grammar

IN THE FAMILY SPACE

Ruth signs in to the family app for the first time. Her PDS shows a consent screen asking whether the app may use her Family updates spaces. She approves it. What did she agree to?

REFERENCE

The `space:` scope [spec]

Ruth's session holds `space:com.example.familyUpdates?authority=*`.

TEXT

```
space:<spaceType>
  [?authority=<did>|self|*]
  [&skey=<skey>|*]
  [&collection=<nsid>|*]...
  [&action=read_self|read|create|update|delete]...
  [&manage=create|update|delete]...
```

Shown wrapped; a real scope is one token [spec].

Table 9.1 · Scope parameters and their defaults

Parameter	Multiple	Default
`spaceType`	no	required; an NSID or `*`
`authority`	no	`self`, the granting user's own DID
`skey`	no	`*`
`collection`	yes	the type declaration's `collections`
`action`	yes	`read`, `create`, `update`, `delete`
`manage`	yes	none

The type may also be given as a named parameter, `space?spaceType=com.example.familyUpdates&authority=*`, but not both ways in one scope. `atproto-oauth` accepts both forms [crates].

The `authority=self` default is the important one. A bare `space:com.example.bookmarks` covers only the user's own bookmarks space. The family app reads a space anchored on Marcus's identity, so for everyone but Marcus it must say `authority=*` or name the authority [spec].

9.2 Read is all-or-nothing

The `read` action is what lets an app read *other* members' repos, because it grants `getDelegationToken` along with the read and sync methods on the user's own PDS. It applies to every repo in the matching spaces and ignores `collection`. `read_self` is the narrower option, with the same read methods for the user's own repo only and no delegation token. Granting `read` also grants `read_self` [spec].

There is no per-collection or per-author read grant. A credential reads the whole space, so a narrower OAuth grant couldn't be enforced anyway.

9.3 Writes and management

`create`, `update`, and `delete` act on specific records and are constrained by `collection`. With `collection` omitted, they cover the type's declared collections, resolved when the grant is evaluated, not frozen at consent. When the lexicon author adds a collection, existing bare grants widen to include it once caches turn over. An app that wants a fixed set lists collections explicitly. With `spaceType=*`, a grant has no type declaration to fall back on, so its writes reach only the collections it lists [spec].

That widening is intentional. A space type that expects many collections can lean on the default, so its `space:` scopes stay short instead of naming every collection, and apps pick up new collections as the type grows without asking people to consent again. The trade-off is that the lexicon author, not the person who consented, decides how far a bare grant reaches.

The type's collection list is a default, not a limit. A scope with no `collection` gets exactly the listed collections, so a bare grant on a type that lists `com.example.foo` lets an app write `com.example.foo` records and nothing else. A scope that names `com.example.bar` explicitly lets the app write those records too, even though the type doesn't list them.

Applications can support space management features through the `manage` attribute of the `space:` scope, which covers the spaces themselves rather than the records in them [spec]. Each value authorizes a set of methods:

Table 9.2 · Space management grants

`manage`	Authorizes
`create`	`createSpace` for that type, usually with `skey=*`
`update`	`updateSpace`, `removeWriter` under every policy, and each policy's administrative methods, such as memberspace `putMembers` and `removeMembers`
`delete`	`deleteSpace`
`create` and `update`	`importSpaceMeta`

Migrating records is the one place the two kinds of grant meet: `importRepo` needs the record action `create` on every collection in the archive.

A `manage` grant covers the space type as a whole. No parameter narrows it to particular policies, so an app holding `manage=create` or `manage=update` can create or update a space under any policy [spec].

The power and scope of the `manage` field is intentional and by design. There are no finer-grained management permissions, and the risk is real. An app holding `manage=update` could let a person make a space public by accident, switch its policy type and lose its member list, or take other harmful and destructive actions. That's the case for the step-up sessions described under If you're building an app, so management grants aren't used casually. It's also a case for confirmation or two-factor steps in the app. A space owner's PDS or space host should be able to lock or disable some kinds of management action, or at least notify the owner when one happens.

Write methods accept only OAuth, since writes are attributed to the user. Read and sync methods accept either a covering OAuth scope or a space credential [spec].

9.4 Worked scopes

IN THE FAMILY SPACE

> The family app asks Marcus for more than it asks everyone else. Marcus creates the space and manages its members. Elena, Ruth, Nora, and Jake read the space and write their own posts, RSVPs, and bookmarks. Elena also uploads photos.

REFERENCE

The family app's session scopes

Session	Scopes	Grants
Marcus	`space:com.example.familyUpdates?manage=create&manage=update`	Create and update his own family updates spaces and manage their members, plus full read and writes to the declared collections
Elena	`space:com.example.familyUpdates?authority=*` and `blob:image/*`	The same reads and writes in every family updates space she's in, plus image uploads
Ruth, Nora, and Jake	`space:com.example.familyUpdates?authority=*`	Elena's grant without uploads

Each session also holds the base `atproto` scope. Marcus's scope leaves out `authority`, so it covers only spaces he's the authority of. Without `manage=delete`, this session can't delete them.

A few other shapes cover cases the family app doesn't need:

Table 9.3 · Worked space scopes

Scope	Grants
`space:com.example.bookmarks`	Own bookmarks: read and write declared collections
`space:com.example.familyUpdates?authority=*&action=read`	Same, read only
`space:com.example.familyUpdates?authority=*&action=read_self`	Own records only, as for a backup tool
`space:com.example.familyUpdates?authority=*&collection=*`	Full read, write any collection
`space:com.example.familyUpdates?authority=did:plc:ejjl2znzqp3ci25dqaslpkk2&skey=our-family&collection=community.lexicon.calendar.rsvp&action=create&action=update`	Create and update RSVPs in the family space
`space:com.example.familyUpdates?authority=*&action=read_self&manage=update&manage=delete`	Administer own family updates spaces, no record writes
`space:*?authority=did:plc:ejjl2znzqp3ci25dqaslpkk2`	Read every space under one authority

9.5 Permission sets

Applications are encouraged to use permission sets whenever possible, because they make for a much better user experience. A permission set describes what the app is asking for in friendlier wording, which is often clearer and easier to read than the raw scopes. The family app publishes `com.example.authFamilyMember` for its members and requests it as `include:com.example.authFamilyMember`:

JSON

```
{
  "type": "permission-set",
  "title": "Family updates",
  "detail": "Read your family's updates and post your own",
  "permissions": [
    {
      "type": "permission",
      "resource": "space",
      "spaceType": "com.example.familyUpdates",
      "authority": "*",
      "action": [
        "read",
        "create",
        "update",
        "delete"
      ]
    }
  ]
}
```

Inside a permission set, `spaceType` must name a concrete type that follows the set's namespace-authority rules. `authority` and `collection` can still be wildcards, and `collection` can name NSIDs from other namespaces. A grant for every space type, written as `spaceType=*`, works only as a standalone scope [spec]. A `space:` scope names one space type, so an app that works with spaces in syzygy asks for one scope for each type, with the same `skey` in each, and a permission set can bundle them when the types fall under its namespace.

9.6 The OAuth consent screen

When Elena signs in to the family app, the OAuth consent screen leads with the permission set's own words: "Family updates: Read your family's updates and post your own" [crates]. Under that, each space permission names its type by the type declaration's `title` ("Family updates"), localized with `title:lang`, and falls back to the NSID when there's no title. A named authority appears as its handle when the handle verifies in both directions, and as its DID otherwise, and `self` needs no label [spec].

The screen saves its loudest warning for a scope with wildcards in both `authority` and `spaceType`, which reaches every space of every type [spec]. The family set names a concrete type, so even with `authority` set to `*`, Elena sees no warning.

9.7 Common questions

Q Why can't my app ask to read just one collection?

A credential reads the whole space, so a narrower read grant couldn't be enforced. Writes are different, and `collection` narrows them.

Q Why doesn't my scope cover other people's spaces?

The `authority` parameter defaults to the user's own identity. An app that reads spaces run by someone else asks for `authority=*` or names the authority.

Q Do users approve again when a space type adds a collection?

Existing sessions pick up the new collection on their own. A bare grant resolves the type's collections each time it's checked, so the change takes effect once caches refresh.

Chapter 10 · Operations

Running spaces is ongoing work, the normal operational work of any distributed system. Providers, applications, and data all have to stay healthy. Moving accounts and content should be worked through often enough to feel routine, and people should have the tooling and a plan for recovering from data loss before an incident happens. This chapter covers what happens to a space after it exists: deletion, migration, moderation, account status, and deciding what each user may see. Each reuses the mechanisms of earlier chapters, with a few space-specific rules.

Outgrowing a provider. The family space grows past what Marcus's PDS can comfortably host, so he moves his account, and with it the space's authority, to a new PDS. He exports the space's metadata and his space repo, imports them on the new host, and points his DID document's `#atproto_pds` service at it. Syncers see the change and reauthorize with the new host. The members' own repos don't move at all. Migrating an account, below, walks through the steps.

An emergency move. Marcus's PDS goes offline without warning. If he kept recent exports, he restores them on a new host and updates his DID document with a rotation key he holds himself, with no help from the old provider. Without backups, the policy, the member list, and his own records go with the old host. The members' records stay safe in their own repos.

Reaching the end. A space made for a family reunion has served its purpose. Marcus tells the family it's closing, then deletes it. The host stops issuing credentials, deletes his records in the space, and notifies every registered syncer, which drops its copy. The members' records stay in their own repos. Deleting a space, below, has the details.



If you're building an app

These are the things that happen after launch. Spaces get deleted, users move to other servers, content gets moderated, and accounts change status. Most of them reach your app as a notification or a firehose event, and each comes with a rule your app follows, such as deleting its copy of a deleted space or hiding an inactive author's records. This chapter is also where

> your app decides what each of its users may see, since a credential gives your app the whole space.

Think through the scenarios above and build them into your integration tests. At the very least, your app should handle a space that migrates to a new host and a space that is deleted. A space that moves to a new authority is harder, since its URI changes and records that point into it may need rewriting, but it happens rarely, if at all.

10.1 Deleting a space

The authority calls `deleteSpace` (`manage=delete`), and the deletion then runs in this order.

1. The host stops issuing credentials for the space and deletes the authority's own records in it [spec].
2. The host sends `notifySpaceUpdate` with `#spaceDeleted` to every registered syncer, including lapsed registrations. It delivers this notice as reliably as an expiry notice.
3. Each syncer that receives the notice reauthorizes. `authorizeSpaceSync` answers a deleted space with `SpaceDeleted` whatever the request's authorization, and only that answer confirms the deletion.

   The notice does carry the syncer's notify token, but that token is a shared secret the syncer chose, not a signature from the host. Anyone who learns it can send a convincing notice, and a deletion can't be undone, so the syncer checks with the host before it drops anything.
   . The syncer deletes every copy of the space's data it holds, including derived state.

A syncer that missed the notice learns of the deletion the next time it reauthorizes. Any other failure says nothing about the space, so the syncer keeps its copy [spec]. The authority should keep a record of the deletion until every syncer registered at the time has been notified or has had its registration dropped.

Members' repos are not touched. Members fully own and control their own data, and an authority has no power to alter their records. After a deletion, each member still holds a copy of their records on their own PDS, readable only by them. An authority may recreate a deleted space with the same type and `skey`, and surviving member records become readable again under the new policy [spec]. Deleting the authority's

account deletes all its spaces, and syncers learn of it through the account events on the firehose.

10.2 Migrating an account

IN THE FAMILY SPACE

> Marcus wants to move his account to another PDS. The reunion event and his other records travel in his space repo. The space's policy, member list, and writer set are kept by his PDS as the space host, so they have to be moved separately.

Migration follows the public-repo pattern, repeated per space, plus host-side state for spaces the account is the authority of [spec]. It runs in four phases: list what has to move, export and copy it, import it on the new host, and verify the copy before anything points at the new host.

10.2.1 1. List spaces

Start on the old host with `listSpaces`, which lists every space the account holds a repo in or is the authority of. That is exactly what needs to move. It takes optional `spaceDid` (authority) and `spaceType` filters, a `limit` from 1 to 100 (default 50), and a `cursor`. Each `spaceView` in the response describes the caller's repo in one space, and `spaceRev` appears only when the caller is the authority [spec]. Keep the response, since phase 4 checks the new host against it.

REFERENCE

`listSpaces` [spec]

Marcus asks his old PDS what he has to move.

HTTP

```
GET /xrpc/com.atproto.space.listSpaces?limit=50
Host: pds-a.example.com
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...
```

JSON

```
{
  "spaces": [
    {
      "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
```

```
      "repoRev": "3n2gudu45c22e",
      "data": {
        "$bytes": "5LoRPo+bR01M+qgTxu/8rCMKh65p+UbeiihTtnbQNKM"
      },
      "spaceRev": "3n2gudu45c32e"
    }
  ]
}
```

10.2.2 2. Export and copy

Phase 2 downloads everything the new host will need. For each space the account is the authority of, `exportSpaceMeta` returns a space meta archive holding the policy, member list, and writer set. For each space it keeps a repo in, `getRepo` returns the repo as a STAR archive, and `listBlobs` names the blobs its records reference, which `getBlob` downloads one at a time. A taken-down account can still make every one of these calls [spec], and `atproto-pds` also allows them for suspended and deactivated accounts [crates].

REFERENCE

`exportSpaceMeta` [spec]

Marcus exports the family space's state.

HTTP

```
GET /xrpc/com.atproto.space.exportSpaceMeta
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
Host: pds-a.example.com
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/vnd.dasl.star
Content-Length: 1195

(1,195-byte STAR archive)
```

The space meta archive is STAR with metadata `{$type: "atproto-space-meta", ver: 1, space}`, then three sections in order, each in strict key order [spec].

REFERENCE

Marcus's space meta archive [spec]

The body of the 1,195-byte archive his PDS exports for the family space, after Jake's removal.

TEXT

```
policy
  -> { "$type": "com.atproto.memberspace.policy",
       "publicRead": false, "publicWrite": false,
       "publishPolicy": true, "publishMembers": true }
com.atproto.memberspace.member/did:plc:53ni6w7okqrxvjxezmopejoo
  -> { "$type": "com.atproto.memberspace.member",
       "read": true, "write": true }
com.atproto.memberspace.member/did:plc:ashwgt3d5ntfxmlcfyizpu2g
  -> { "$type": "com.atproto.memberspace.member",
       "read": true, "write": true }
com.atproto.memberspace.member/did:plc:p4256dosynj6j6tkypz2q2l3
  -> { "$type": "com.atproto.memberspace.member",
       "read": true, "write": true }
com.atproto.memberspace.member/did:web:erin.example
  -> { "$type": "com.atproto.memberspace.member",
       "read": true, "write": false }
writer/did:plc:53ni6w7okqrxvjxezmopejoo
  -> { "repoRev": "3n2gsittdk32e",
       "data": { "$bytes":
"LQ8ty3iUor9cbpB7YIHwOzjr7V9P+yOiOatCVXE2rdY" },
       "spaceRev": "3n2gsittdk42e" }
writer/did:plc:ashwgt3d5ntfxmlcfyizpu2g
  -> { "repoRev": "3n2gtdo55s32p",
       "data": { "$bytes":
"INjMYdWaUBlDhmPVrmtaDralYgQLu+GffB4QmvGD2SE" },
       "spaceRev": "3n2gtdo55s22e" }
writer/did:plc:ejjl2znzqp3ci25dqaslpkk2
  -> { "repoRev": "3n2gudu45c22e",
       "data": { "$bytes":
"5LoRPo+bR01M+qgTxu/8rCMKh65p+UbeiihTtnbQNKM" },
       "spaceRev": "3n2gudu45c32e" }
writer/did:plc:p4256dosynj6j6tkypz2q2l3
  -> { "repoRev": "3n2gtxdkr232p",
       "data": { "$bytes":
"eVD7vKgDQrLKfOvN3VZJHYX0HGz158ocKa5oRd3kaYw" },
       "spaceRev": "3n2gtxdkr222e" }
```

Each writer value holds `repoRev`, `data`, and `spaceRev`. The importing host must reject a parse error, a missing, duplicate, or out-of-order key, a policy it doesn't implement (`UnsupportedPolicy`), and any policy state key it doesn't recognize, rather than silently dropping access state [spec].

REFERENCE

`getRepo` [spec]

Marcus downloads his own repo in the family space.

HTTP

```
GET /xrpc/com.atproto.space.getRepo
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
  &repo=did:plc:ejjl2znzqp3ci25dqaslpkk2
Host: pds-a.example.com
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/vnd.dasl.star

(STAR archive, laid out as in Chapter 7)
```

REFERENCE

`listBlobs` [spec]

Marcus checks which blobs his records reference.

Try it 10.1 · List Marcus's blobs. This example is interactive on the web: https://atproto.how/atproto-spaces-field-guide/10-operations#try-list-blobs-marcus

REQUEST

```
GET /xrpc/com.atproto.space.listBlobs
  ?space=at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family
  &repo=did:plc:ejjl2znzqp3ci25dqaslpkk2
Host: pds-a.example.com
Authorization: DPoP <access token>
DPoP: <proof JWT>
```

HEADERS

`Authorization` The session's access token, bound to a key only the client holds. Appendix G, OAuth requests

`DPoP` A proof signed with that key for this method and URL, so a copied token can't be replayed. Appendix G, OAuth requests

SAMPLE RESPONSE

```
{
  "cids": []
}
```

Marcus's records in the family space reference no blobs, so there is nothing more to copy. An account with photos, like Elena's, fetches each CID with `getBlob` and uploads it to the new host with `com.atproto.repo.uploadBlob`.

10.2.3 3. Import

Imports run on the new host, against the account created there by the standard account migration. Order matters. `importSpaceMeta` comes first for each space the account is the authority of, because `importRepo` refuses the authority's repo in a space the host doesn't know yet. Neither call takes a `space` parameter, since the archive itself says which space and repo it holds [spec].

REFERENCE

`importSpaceMeta` [spec]

Marcus loads the family space's state onto his new PDS.

HTTP

```
POST /xrpc/com.atproto.space.importSpaceMeta
Host: pds-c.example.com
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...
Content-Type: application/vnd.dasl.star
Content-Length: 1195

(the space meta archive from phase 2)
```

HTTP

```
HTTP/1.1 200 OK
Content-Length: 0
```

REFERENCE

`importRepo` [spec]

Then his own repo in the space.

HTTP

```
POST /xrpc/com.atproto.space.importRepo
Host: pds-c.example.com
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...
Content-Type: application/vnd.dasl.star

(the repo archive from phase 2)
```

HTTP

```
HTTP/1.1 200 OK
Content-Length: 0
```

`importSpaceMeta` answers `NotAuthorized`, `UnsupportedPolicy`, or `InvalidSpaceMeta`. `importRepo` answers `InvalidRepoFile` for a malformed archive, `RepoMismatch` for another account's repo, and `SpaceNotFound` when the caller is the authority and the space hasn't been imported yet [spec].

Each import needs a `space:` scope covering every write it makes, and an archive with any uncovered record is rejected whole [spec]. In practice that means `create` on every collection in each repo archive, plus `manage=create` and `manage=update` for each space the account is the authority of [spec]. The breadth is intentional. Migration is a deliberately destructive operation that also changes the DID document, which for a did:plc identity means signing a PLC operation. A migration tool should tell its user this plainly before asking for either.

10.2.4 4. Verify

Before anything points at the new host, check that it holds exactly what the old one did. Run `listSpaces` on the new host and compare each entry with phase 1. `data` is the LtHash digest of the repo's records, so it must match byte for byte, while `repoRev` may differ because the new host can assign its own [spec]. For each space the account is the authority of, export the space meta from the new host and compare its policy, members, and writer entries with the phase 2 archive. For each repo, compare the `listBlobs` CIDs on both hosts.

REFERENCE

`listSpaces` on the new host [spec]

The digest matches phase 1, and the new host has assigned its own `repoRev`.

HTTP

```
GET /xrpc/com.atproto.space.listSpaces?limit=50
Host: pds-c.example.com
Authorization: DPoP eyJ0eXAiOiJhdCtqd3QiLCJhbGciOi...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6Im...
```

JSON

```
{
  "spaces": [
    {
      "space": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family",
      "repoRev": "3n2gx4mfrk22e",
      "data": {
        "$bytes": "5LoRPo+bR01M+qgTxu/8rCMKh65p+UbeiihTtnbQNKM"
      },
      "spaceRev": "3n2gudu45c32e"
    }
  ]
}
```

With every digest and blob list matching, Marcus finishes the standard account migration. He points `#atproto_pds` in his DID document at the new host and deactivates the old account.

10.2.5 After the move

The new host generates its own `#atproto_kem` key and publishes it in the DID document. Its op log starts empty, which leaves every syncer where it would be after falling behind a compacted op log: it has to resync everything. A syncer that sees an authority's `#atproto_pds` change reauthorizes with the new host, which registers it again [spec].

Notification registrations, credential revocations, and deleted-space tombstones don't carry over [spec]. Lost revocations matter only briefly, since a credential lives at most 60 minutes and a repo host only has to keep a revocation for 60 minutes [spec]. Until then, the new host can accept a revoked credential for the migrated account's repos. If the migration also changed the authority's `#atproto` key, which signs its credentials [spec], a host that has refreshed the authority's DID document rejects credentials signed with the old key, while one still holding a cached copy may accept them.

For app developers. Build a way for your app to notice that a host has changed. When an authority's or a writer's `#atproto_pds` moves, clear the sessions and cached credentials tied to the old host, drop cached

endpoints and `#atproto_kem` keys, reauthorize with the new host, and resync from scratch once, since the new host's op log starts empty.

10.3 Moderation

Every service moderates what it hosts and serves. A moderation service is just another reader: it needs a credential, so the authority must admit it. That is a feature. Moderation needs access to the data and often its own data patterns, so a moderator, and the applications and tools they use, are admitted to the space the same way everyone else is. The protocol defines no universal reporting interface [spec].

IN THE FAMILY SPACE

> Marcus has made Jake read-only, and Jake's bookmark is still in the space. Marcus can't delete it, because it sits in Jake's space repo on Jake's PDS. He labels it instead.

Labels on space data are `com.atproto.label.record` records, written by the labeler into its own repo *in the same space* as the subject [spec].

Labels that reference a record in another space must be ignored [spec], but an application can make a deliberate exception for spaces it knows are in syzygy. An application can put a community space in syzygy with a space for its moderators (Chapter 3). The community space is the primary space and holds members' content. Its syzygy space, readable only by moderators and their tools, holds moderator notes, case records, and `com.atproto.label.record` labels pointing at content in the community space. Because the application knows the two spaces are in syzygy, it honors labels from the moderation space while still ignoring cross-space labels from anywhere else. A service that doesn't know about the syzygy follows the spec and ignores them, so this is a deliberate, app-level departure from the default.

REFERENCE

A label record [spec]

Marcus's `!takedown` label on Jake's bookmark, written to `com.atproto.label.record/3n2gu2vyuk22e` in his own space repo.

JSON

```
{
  "$type": "com.atproto.label.record",
```

```
  "uri": "at:space/did:plc:ejjl2znzqp3ci25dqaslpkk2/
com.example.familyUpdates/our-family/
did:plc:brthxhhoomvrib2avauses35/
community.lexicon.bookmarks.bookmark/3n2gtrxvls22p",
  "val": "!takedown",
  "cts": "2026-11-16T15:34:00.000Z"
}
```

REFERENCE

`com.atproto.label.record` fields [spec]

Field	Type	Required	Meaning
`uri`	`uri`	yes	Subject: a record, a repo (account) or another resource
`cid`	`cid`	no	Version of the subject
`val`	string, at most 128 characters	yes	Label value
`cts`	`datetime`	yes	Created-at
`exp`	`datetime`	no	Expires-at

Label records are keyed by TID and appear only in space repos. Public data uses signed `com.atproto.label.defs#label` objects, served through `com.atproto.label.subscribeLabels` and `queryLabels` [spec].

Per the spec, labels referencing records in another space are ignored. The other label rules keep labels cheap to index and private to the space [spec]:

- When one repo holds several labels with the same `(uri, cid, val)`, the one with the latest `cts` wins, so a labeler revises a label by writing a new one.
- A labeler's labels sync only if it is in the writer set. The authority is always in it, so it can always label.
- Labels on records in non-public spaces stay off the public `subscribeLabels` stream, where they would leak metadata about the data.
- Each space type decides what labels about whole accounts mean, since the spec leaves that open.

The authority can publish `!takedown` labels to remove content from its space, and syncers and apps are expected to respect them [spec]. Nothing enforces this. Applications honor takedowns by convention, and no protocol mechanism makes an application hide a labeled record. The authority's recourse is reactive. In a memberspace, it can

drop an application that ignores its labels from `allowedApps`, and it stops minting credentials for that application [spec]. That cuts off the application's reads, but not the writes its users make through it, because `allowedApps` doesn't apply to writes.

Infrastructure takedowns work at each layer [spec]. A PDS can take down an account (all its spaces and space repos) or a blob. A space host can take down an authority (all its spaces). An AppView can take down an account, a whole space, or individual records. Repo hosts use the public sync error names, such as `RepoTakendown` and `RepoDeactivated`.

`atproto-pds` answers `getBlob` for a taken-down space blob with `400 BlobNotFound` [crates].

10.4 Account status and identity events

Deactivation and deletion apply to space data the same way they apply to public data. When a space authority is taken down or deactivated, its spaces are suspended. `authorizeSpaceSync` and `listWriters` answer `RepoTakendown` or `RepoDeactivated`, and existing credentials run out. Member repos stay as they are, and the spaces resume once the authority is reinstated [spec]. Registrations lapse but are kept, so syncers get `#registrationExpired` on the first update after the space resumes. Syncers keep their data through a suspension, since only deletion tells them to drop it.

Account status, handle changes, and key rotations (including `#atproto_kem`) arrive as `#account` and `#identity` events on the public firehose, so a syncer that follows only spaces still subscribes to them [spec]. That holds for every atproto application, space-specific or not. An active Jetstream consumer for account and identity events is the cheapest and most effective way to know when an account your application cares about needs its cached handle, service endpoints, or verification keys refreshed.

10.5 Access control in applications

A space credential lets an app sync a space. The app decides which of its users may see the space [spec]. It bases that decision on the space type, its policy, its records, any supporting data the type depends on, and the other spaces in its hyperspace. For example, a `memberspace` may publish its member records, while a follower-gated `managedspace` needs the app to know the follow graph. The app also decides whether a space is fully public, and it watches for changes to access.

Some rules apply to all space data, and apps must respect them [spec]:

- an inactive author's space data is not displayed or distributed;
- an inactive space authority's account affects all data in all its spaces;
- deleted records are deleted, as in public repos;
- space deletion is honored when a syncer is told of it.

Space sync does not carry account status changes. Apps sync those separately, for example from the firehose.

10.6 Common questions

Q If a space is deleted, are my posts gone?

Your records stay in your space repo on your own PDS, readable only by you. The apps that kept copies delete them. If the authority recreates the space with the same type and key, your records become readable again under the new policy.

Q What happens to my spaces when I move my account?

They move with you, one space at a time. Each space repo is exported and imported, and spaces you're the authority of bring their policy and member list along in a space meta archive.

Q Who moderates a space?

Every service moderates what it hosts and serves. Inside the space, the authority publishes `!takedown` labels, and syncers and apps respect them. A moderation service that wants to see the space needs a credential like any other reader.

Chapter 11 · Implementation notes

The spec is firmly opinionated in some places and open-ended in others, so developers of space hosts, PDSes, applications, and tools have a lot of room in what they build and how. This chapter walks through the directions those choices can take and what each one implies, including its side effects, along with the passages that need careful reading and a checklist for your own conformance.

If you're building an app

If you build on an existing library, skim the trade-offs section. It explains behavior that can surprise app developers, such as why a captured request can be replayed to the same host until the credential expires. The rest of the chapter, including the conformance checklist, is for people building PDSes, space hosts, and libraries.

11.1 Where the spec leaves a choice

These are the choices atproto-crates makes. Another conforming implementation may differ, so clients should not depend on them.

REFERENCE

Authorization choices [spec] [crates]

What happens inside `authorizeSpaceSync` when the spec doesn't pin it down.

Choice	What the spec says	What atproto-crates does
Check order	Unstated	Key and notify, hosted authority, active authority, `SpaceDeleted`, delegation token, attestation, policy
HTTP status of refusals	Unstated	`NotAuthorized`, `AppNotAuthorized`, `InvalidClientAttestation`, and `BlobNotFound` all return 400
Managing app can't answer	No error declared	502 `UpstreamFailure`; the credential isn't issued,

Choice	What the spec says	What atproto-crates does
		and the registration is kept
Suspended authority	`RepoTakendown` or `RepoDeactivated`	Adds `RepoSuspended`

`RepoSuspended` is there because repo-host reads already declare it. A client gets the same error for the same account state whichever host it asks.

REFERENCE

Notification and sync choices [spec] [crates]

How the space host talks to syncers.

Choice	What the spec says	What atproto-crates does
Notify token	Opaque, at most 512 bytes, one per space and stable across authorizations	The client derives an HMAC-SHA256 of the space URI under an app secret and checks it statelessly
`confirmRegistration` challenge	A random challenge	32 random bytes in hex, with 10 seconds to answer
Expiry and deletion notice retries	Backoff for at least 72 h	1 m, 10 m, 1 h, 6 h, then every 12 h, with 50 to 100% jitter, for at least 72 h
Write and removal updates	Reasonable-effort delivery	Queued in the notify outbox; a failure that might clear is retried
`listRepoOps` with both `since` and `cursor`	Both parameters exist	Both apply; a stale `since` returns the remaining ops and the commit

The retry schedule is front-loaded so a syncer that was briefly down hears about an expiry or deletion within the hour. After that it settles to every 12 hours, which keeps the 72-hour window to a handful of attempts. The jitter spreads retries out, so a syncer coming back from an outage isn't hit by every space at once.

REFERENCE

Membership and migration choices [spec] [crates]

Choice	What the spec says	What atproto-crates does
Repeated DID in `putMembers`	Unstated	Last entry wins
`managingApp` validation	A `did#serviceId` string	An absolute `did:plc` or `did:web` with a fragment, checked on create, update, and import
Export by a restricted account	Taken-down accounts may export	Suspended and deactivated accounts may too

11.2 Passages that need careful reading

The delegation token section says the token is "addressed (`aud`) to the space authority", but the normative example uses the space host's service id, which for the family space is `did:plc:ejjl2znzqp3ci25dqaslpkk2#atproto_pds`. Use the service id, as client attestations also do.

The policy section says that after a policy migration "applications remain responsible for removing any writers", while memberspace says the authority should remove a removed member from the writer set. Hosts may prune differently on a policy change. A careful syncer re-reads `listWriters` after any policy change it learns of.

`clearRecords` clears the operation log along with the records. A syncer resuming from an old `since` gets no ops, and the commit it gets back has the empty digest, so its fold mismatches and it recovers with `getRepo` (or treats the repo as empty).

11.3 Design trade-offs worth knowing

11.3.1 No proof of possession at authorization

`authorizeSpaceSync` binds `credentialKey` without asking the caller to sign with it. Whoever holds a valid delegation token can bind a credential to a key of their choosing. Delegation tokens are single-use, last 60 seconds, and name one host, so the exposure is small, but treat one as a bearer secret for that minute and keep it out of logs.

11.3.2 Audience replay is allowed

A credential signature covers only the credential and the audience, not the HTTP method or request path, so a captured request can be replayed to the same audience until the credential expires. That is

deliberate. The design targets one threat, a repo host that received the credential replaying it to another host, and binding the signature to the audience is enough to stop it. Within one audience, TLS keeps a request from being captured in the first place, so covering the method and path would add nothing against that threat.

11.3.3 Notifications are unauthenticated beyond the token

`confirmRegistration` and `notifySpaceUpdate` carry no service auth. A leaked token can only make a syncer check for updates unnecessarily, provided the syncer confirms everything with the repo host or space host and never drops data on a notice alone.

11.3.4 Compressed versus uncompressed keys

DID documents and `did:key` carry compressed P-256 keys; HPKE's `enc` and the derivation use uncompressed 65-byte encodings.

11.4 Conformance checklist

- LtHash: empty digest matches `e5a00aa9...b183ad`; add then remove returns to empty; order of adds doesn't matter
- Element bytes are exactly `{collection}/{rkey}/{cid}` with the CID as a string
- HPKE secret export matches the export-only P-256 entry in the RFC 9180 test vectors
- A new ephemeral KEM key for every request, retry, and page
- `Accept-Signature` with the right profile tag on every credentialed `getLatestCommit`, `getRepo`, `listRepoOps`, and `getRecord`
- MAC verified as `hmac-sha256` over the `sf`-canonicalized covered header, and the header matched against the body
- MAC failure triggers a DID refresh and a retry with a new key
- Credential signature found by tag `atproto-space-credential` under any label, covering exactly `("authorization" "atproto-space-audience")`
- `keyid` checked against `cnf.kid` when present; extra parameters ignored
- Verifiers accept high-S P-256 signatures
- Audience is the repo DID for repo methods and the authority DID for host methods
- Credentials capped at 60 minutes; revoked `jti` values held 60 minutes
- `listWriters` entries branched on `$type`; `#repoRemoved` stops a repo being treated as part of the space

- `listRepoOps` applied until a page has no `cursor`, then compared with that page's commit
- STAR reader refuses bad control bytes, missing end byte, trailing bytes, out-of-order keys, and size-limit breaches
- `confirmRegistration` echoes the challenge only for a token issued for that space
- `notifySpaceUpdate` matched on full `$type`, with unknown types ignored
- `#spaceDeleted` triggers reauthorization; data is dropped only on a `SpaceDeleted` answer
- A checkpoint older than 72 hours lists the whole writer set
- `UpstreamFailure` treated as retryable, never as a refusal
- Space-only syncers still consume `#account` and `#identity` firehose events

Appendix A · XRPC methods

Every method of the spaces extension, under `com.atproto.space.*`, `com.atproto.memberspace.*` and `com.atproto.managedspace.*` [spec]. The lexicons in `bluesky-social/atproto` are the source of truth for parameters, outputs, and errors, and atproto-crates validates against copies of them [crates].

Table A.1 · Who calls whom

Method	From	To	Auth
`getDelegationToken`	App	PDS	OAuth read
`authorizeSpaceSync`	Syncer	Space host	tokens in body
`listWriters`	Syncer	Space host	credential
`getRecord`	Syncer	Repo host	credential · MAC
`listRepoOps`	Syncer	Repo host	credential · MAC
`notifyRepoUpdate`	Repo host	Space host	service auth

Method	Served by	Type	Auth	Purpose
`com.atproto.space.authorizeSpaceSync`	space host	procedure	none; tokens in body	Credential, spaceRev, optional notify registration
`com.atproto.space.listWriters`	space host	query	credential	Writer set: #repoWrite and #repoRemoved entries
`com.atproto.space.removeWriter`	space host	procedure	OAuth as the authority, manage=update	Remove one writer
`com.atproto.space.getSpace`	space host	query	OAuth as the authority	Space and its policy
`com.atproto.space.exportSpaceMeta`	space host	query	OAuth as the authority	Space meta STAR
`com.atproto.space.notifyRepoUpdate`	space host	procedure	service auth from repo host	Repo advanced: repoRev, data
`com.atproto.space.getRecord`	repo host	query	OAuth or credential; MAC	One record
`com.atproto.space.listRecords`	repo host	query	OAuth or credential	Records, or paths with excludeValues
`com.atproto.space.getBlob`	repo host	query	OAuth or credential	One blob
`com.atproto.space.listBlobs`	repo host	query	OAuth or credential	Referenced blob CIDs

Method	Served by	Type	Auth	Purpose
com.atproto.space.getLatestCommit	repo host	query	OAuth or credential; MAC	Current commit
com.atproto.space.getRepo	repo host	query	OAuth or credential; MAC	Whole repo as STAR
com.atproto.space.listRepoOps	repo host	query	OAuth or credential; MAC	Op log since a rev, commit on every page
com.atproto.space.notifyCredentialsRevoked	repo host	procedure	service auth from authority	Revoke credentials by jti
com.atproto.space.getDelegationToken	user's PDS	query	OAuth read	Mint a delegation token
com.atproto.space.createRecord	PDS	procedure	OAuth	Create in own repo
com.atproto.space.putRecord	PDS	procedure	OAuth	Create or update
com.atproto.space.deleteRecord	PDS	procedure	OAuth	Delete
com.atproto.space.applyWrites	PDS	procedure	OAuth	Atomic batch
com.atproto.space.clearRecords	PDS	procedure	OAuth	Empty own repo
com.atproto.space.importRepo	PDS	procedure	OAuth, create on every collection	Migration import
com.atproto.space.listSpaces	PDS	query	OAuth	Spaces held or owned, with repo state
com.atproto.space.createSpace	PDS	procedure	OAuth manage=create	Create with a policy
com.atproto.space.updateSpace	PDS	procedure	OAuth as the authority, manage=update	Replace the policy
com.atproto.space.deleteSpace	PDS	procedure	OAuth as the authority, manage=delete	Delete
com.atproto.space.importSpaceMeta	PDS	procedure	OAuth as the authority, manage=create and manage=update	Migration import
com.atproto.space.notifySpaceUpdate	syncer	procedure	notify token in body	A repo write or removal, deletion, or expired registration

Method	Served by	Type	Auth	Purpose
com.atproto.space.confirmRegistration	syncer	procedure	notify token in body	Echo a challenge to confirm a registration
com.atproto.memberspace.putMembers	space host	procedure	OAuth as the authority, manage=update	Add or replace members
com.atproto.memberspace.removeMembers	space host	procedure	OAuth as the authority, manage=update	Remove members
com.atproto.memberspace.listMembers	space host	query	OAuth as the authority	List members
com.atproto.managedspace.checkAccess	managing app	query	service auth from authority	read and write for one identity

REFERENCE

Repo host read methods [spec]

Method	Other parameters	Output	MAC
getLatestCommit	none	commit	yes
getRecord	collection, rkey	uri, cid, value	yes
listRecords	collection?, limit (1 to 1000, default 50), cursor?, excludeValues?	records of {collection, rkey, cid, value?}, cursor?	no
listRepoOps	since?, limit (1 to 1000, default 100), cursor?, excludeValues?	ops, commit, cursor?	yes
getRepo	none	STAR body	yes
listBlobs	limit (1 to 1000, default 500), cursor?	cids, cursor?	no
getBlob	cid	the blob as uploaded	no

All of them declare SpaceNotFound, RepoNotFound, RepoTakendown, RepoSuspended, and RepoDeactivated. The MAC'd methods add InvalidSpaceMac, getRecord adds RecordNotFound, getBlob adds BlobNotFound, and listRecords and listRepoOps add InvalidCursor [spec]. Without collection, listRecords lists every collection.

Records and shared object definitions [spec]:

Table A.2 · Records and shared definitions

NSID	Kind	Record key	Fields
com.atproto.space.defs#commit	object	none	ver, repo, space, rev, data
com.atproto.space.defs#repoWrite	object	none	repo, repoRev, data, spaceRev, prevSpaceRev?

NSID	Kind	Record key	Fields
com.atproto.space.defs#repoRemoved	object	none	repo, spaceRev, prevSpaceRev?
com.atproto.space.policy	record	self	policy
com.atproto.personalspace.policy	object	none	none
com.atproto.memberspace.policy	object	none	publicRead?, publicWrite?, allowedApps?, publishPolicy?, publishMembers?
com.atproto.memberspace.defs#member	object	none	did, read, write
com.atproto.memberspace.member	record	the member's DID	read, write
com.atproto.managedspace.policy	object	none	managingApp, publishPolicy?
com.atproto.label.record	record	TID	uri, cid?, val, cts, exp?

Spaces reuse com.atproto.sync.subscribeRepos unchanged for the #account and #identity events that carry account status and key changes [spec]. Records inside a space can still point at each other with a com.atproto.repo.strongRef, as the RSVP's subject does.

Appendix B · Headers

Table B.1 · Space headers

Header	Direction	Value
`Authorization`	request	`Atproto-Space-Credential <jwt>`
`Atproto-Space-Audience`	request	Repo DID, or authority DID for host methods
`Signature-Input` (credential)	request	`<label>=("authorization" "atproto-space-audience");tag="atproto-space-credential"`
`Signature` (credential)	request	`<label>=:<64-byte r and s, base64>:`
`Accept-Signature`	request	`<label>=("atproto-space-commit";sf);keyid="did:key:...";tag="atproto-space-commit"`, or the `atproto-space-record` profile
`Atproto-Space-Commit`	response	SF dictionary: `ver`, `repo`, `space`, `rev`, `data`
`Atproto-Space-Record`	response	SF dictionary: `uri`, `cid`
`Signature-Input` (MAC)	response	Echoes the requested label and parameters
`Signature` (MAC)	response	`<label>=:<32-byte HMAC, base64>:`
`Vary`	response	Includes `Accept-Signature`
`Content-Type`	both	`application/vnd.dasl.star` for `getRepo`, `exportSpaceMeta`, `importRepo`, `importSpaceMeta`

Appendix C · Token types

Table C.1 · Space token types

typ	Signed by	aud	Lifetime	Special claims
atproto-space-delegation+jwt	user #atproto	<authority> #atproto_pds	60 s, single use	sub = space; no lxm
atproto-client-attestation+jwt	app JWKS key	<authority> #atproto_pds	short, single use	iss = sub = client_id
atproto-space-credential+jwt	authority #atproto	none	10 min default, 60 max; multi-use	sub = space, cnf.kid, jti

Appendix D · Errors

Errors declared in the lexicons [spec], plus the few `atproto-pds` adds, which are marked.

Table D.1 · Space errors

Error	Declared by	Meaning
`SpaceNotFound`	most methods	No such space; `authorizeSpaceSync` may also use it in place of `AccountNotAuthorized`
`SpaceDeleted`	`authorizeSpaceSync`	Deleted; drop data
`SpaceAlreadyExists`	`createSpace`	The space already exists
`AccountNotAuthorized`	`authorizeSpaceSync`	The policy refuses the identity
`AppNotAuthorized`	`authorizeSpaceSync`	The policy refuses the app, or a required attestation is missing
`NotAuthorized`	`authorizeSpaceSync`, space management, member methods	Generic refusal, or the caller isn't the authority
`InvalidDelegationToken`	`authorizeSpaceSync`	Bad delegation token
`InvalidClientAttestation`	`authorizeSpaceSync`	Attestation or client metadata can't be verified
`RegistrationNotConfirmed`	`authorizeSpaceSync`	The notify endpoint didn't echo the challenge
`UnsupportedPolicy`	`createSpace`, `updateSpace`, `importSpaceMeta`	Host lacks the policy
`PolicyMismatch`	`putMembers`, `removeMembers`, `listMembers`	The space isn't a memberspace
`FutureRev`	`notifyRepoUpdate`	`repoRev` too far in the future
`InvalidSpaceMac`	`getLatestCommit`, `getRecord`, `getRepo`, `listRepoOps`	Missing or bad `Accept-Signature`
`InvalidCursor`	`listRepoOps`, `listRecords`, `listSpaces`, `listMembers`	Malformed cursor
`RepoNotFound`	repo-host reads	Repo never written
`RecordNotFound`	`getRecord`, `applyWrites`	No such record
`RecordAlreadyExists`	`createRecord`, `applyWrites`	The record key is taken
`BlobNotFound`	`getBlob`	No such blob, or taken down
`InvalidRepoFile`, `RepoMismatch`	`importRepo`	Malformed archive, or another account's repo
`InvalidSpaceMeta`	`importSpaceMeta`	Malformed archive

Error	Declared by	Meaning
`RepoTakendown`, `RepoSuspended`, `RepoDeactivated`	repo-host reads; `authorizeSpaceSync` and `listWriters` declare the first and last for the authority	Account status
`UpstreamFailure`	not declared; `atproto-pds` (502)	A managing app couldn't answer; retry

Appendix E · Test vectors

The empty LtHash digest is `e5a00aa9991ac8a5ee3109844d84a55583bd20572ad3ffcd42792f3c36b183ad`. The shared secret is checked against the export-only P-256 entry in the HPKE test vectors. Appendix G has worked MACs, populated LtHash states, tokens, and credential signatures, all computed from the Appendix F keys. For further cases and for STAR archives, generate values from an implementation: atproto-crates' `tools/space-vectors` and test suites, or the reference `@atproto/space` package. microcosm.blue's STAR interop fixtures are vendored in `atproto-dasl`.

Appendix F · Example identities and keys

These are the identities and keys behind the book's examples, published with their private keys so that readers and agents can check signatures, tokens, and other cryptographic structures against real material. Every private key here is public test material: use it only for examples and tests.

The five `did:plc` identities are valid. Each DID is derived from its signed genesis operation, which is included so you can verify it offline. They are not registered with the PLC directory, so resolve them from this appendix. `#atproto` and rotation keys are K-256; `#atproto_kem`, the app's client and DPoP keys, and the syncer's keys are P-256. Private keys are in the multibase form that `goat key generate` prints, beside the public key as a `did:key`.

Table F.1 · Example identities

Name in the book	DID	Handle	PDS	Role
Marcus	`did:plc:ejjl2znzqp3ci25dqaslpdk2`	`marcus.example.com`	`https://pds-a.example.com`	Space authority for the family space
Elena	`did:plc:53ni6w7okqrxvjxezmopejoo`	`elena.example.com`	`https://pds-a.example.com`	Member and writer
Ruth	`did:plc:p4256dosynj6j6tkypz2q2l3`	`ruth.example.com`	`https://pds-b.example.com`	Member and writer on a second repo host
Nora	`did:plc:ashwgt3d5ntfxmlcfyizpu2g`	`nora.example.com`	`https://pds-b.example.com`	Member and writer
Jake	`did:plc:brthxhhoomvrib2avauses35`	`jake.example.com`	`https://pds-b.example.com`	Removed member, author of the labeled bookmark
Erin	`did:web:erin.example`	`erin.example`	none	Read-only member with no account
Managing app	`did:web:newsletter.example`	none	none	Managing app for the managedspace example, service `#newsletter_app`

Name in the book	DID	Handle	PDS	Role
Family app	`https://family.example/client-metadata.json`	none	none	OAuth client and syncer

To check the material yourself:

1. Decode a private key from base58btc, drop its two-byte multicodec prefix (`0x86 0x26` for P-256, `0x81 0x26` for K-256), derive the public key, and compare its compressed `did:key` with the one listed.
2. Encode the signed genesis operation as DAG-CBOR and hash it with SHA-256. The DID is `did:plc:` followed by the first 24 characters of that hash in lowercase base32.
3. Remove `sig` from the genesis operation, encode the rest as DAG-CBOR, and verify `sig` (base64url) against the rotation key.
4. Compare the DID document with the genesis operation: the `#atproto` and `#atproto_kem` methods and the `#atproto_pds` service all come from it.

F.1 Marcus

Table F.2 · Marcus's keys

Key	Type	Private key	Public key
`#atproto`	K-256	`z3vLbPcG9hWAWGjzA3HGDjuaaggcNZdDbH44rWmbCWMS1DNJ`	`zQ3sha4dxQwxufYojiGeueDiyHcEFjYjPDVaNWoUD9UJbJRfG`
`#atproto_kem`	P-256	`z42tqfEFoheNLvyMjabEiYLfFJnwEn2MN12eybBWU9qwxtM6`	`zDnaethu4zDP2t6nmJuTrZGSnc4VoDUmaG3NfqziTmsExNC8f`
Rotation	K-256	`z3vLgGL5a3JegMMUxx8GCNCw5rWyhcFq86Rwtw2mQQTkfy4X`	`zQ3shbtBdtYs7b3NBfApyRebCCmg8nT1mL3E8A5KAyXREE3bS`

DID document for `did:plc:ejjl2znzqp3ci25dqaslpkk2`:

JSON

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/multikey/v1",
    "https://w3id.org/security/suites/secp256k1-2019/v1",
    "https://w3id.org/security/suites/ecdsa-2019/v1"
  ],
  "id": "did:plc:ejjl2znzqp3ci25dqaslpkk2",
  "alsoKnownAs": [
    "at://marcus.example.com"
  ],
  "verificationMethod": [
    {
      "id": "did:plc:ejjl2znzqp3ci25dqaslpkk2#atproto",
      "type": "Multikey",
```

```
      "controller": "did:plc:ejjl2znzqp3ci25dqaslpkk2",
      "publicKeyMultibase":
"zQ3sha4dxQwxufYojiGeueDiyHcEFjYjPDVaNWoUD9UJbJRfG"
    },
    {
      "id": "did:plc:ejjl2znzqp3ci25dqaslpkk2#atproto_kem",
      "type": "Multikey",
      "controller": "did:plc:ejjl2znzqp3ci25dqaslpkk2",
      "publicKeyMultibase":
"zDnaethu4zDP2t6nmJuTrZGSnc4VoDUmaG3NfqziTmsExNC8f"
    }
  ],
  "service": [
    {
      "id": "#atproto_pds",
      "type": "AtprotoPersonalDataServer",
      "serviceEndpoint": "https://pds-a.example.com"
    }
  ]
}
```

Signed genesis operation:

JSON

```
{
  "type": "plc_operation",
  "rotationKeys": [
    "did:key:zQ3shbtBdtYs7b3NBfApyRebCCmg8nT1mL3E8A5KAyXREE3bS"
  ],
  "verificationMethods": {
    "atproto":
"did:key:zQ3sha4dxQwxufYojiGeueDiyHcEFjYjPDVaNWoUD9UJbJRfG",
    "atproto_kem":
"did:key:zDnaethu4zDP2t6nmJuTrZGSnc4VoDUmaG3NfqziTmsExNC8f"
  },
  "alsoKnownAs": [
    "at://marcus.example.com"
  ],
  "services": {
    "atproto_pds": {
      "type": "AtprotoPersonalDataServer",
      "endpoint": "https://pds-a.example.com"
    }
  },
  "prev": null,
  "sig": "rX1R8ryG9hDFsdYzIJt15oQXGg1T4M1zQKUCCpLAQcNvlTWp6X6vrPpUNKx-
NbY6suTFznt8KGOkWR9aGRT-3Q"
}
```

F.2 Elena

Table F.3 · Elena's keys

Key	Type	Private key	Public key
#atproto	K-256	z3vLWq25XWqSdkR5bWYZMy2nmXijMLDxAy4L24QVA5pkAp9c	did:plc:ejjl2znzqp3ci25dqaslpkk2 zQ3shPWwfJKjriJqb4CsR7KBBHkaffnpZin5NCyAYJzztnuRV

Key	Type	Private key	Public key
#atproto_kem	P-256	z42ts5ZyDd6kG1fUTNtQJDjgyygVGigU2nwRqdHgVdMkeyK3	zDnaeueFwjSjPA49dvab7bN4VjYDvv7eBVsDSjncdQHsud125
Rotation	K-256	z3vLgyyN9LGHhkntCvaEArxvdfkEJYkAMyE2vsUZDdtXTFo6	zQ3shiEHng2LXQA12HNn54XzBTquzZprwYjCTKu1ySPh71N6r

DID document for `did:plc:53ni6w7okqrxvjxezmopejoo`:

JSON

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/multikey/v1",
    "https://w3id.org/security/suites/secp256k1-2019/v1",
    "https://w3id.org/security/suites/ecdsa-2019/v1"
  ],
  "id": "did:plc:53ni6w7okqrxvjxezmopejoo",
  "alsoKnownAs": [
    "at://elena.example.com"
  ],
  "verificationMethod": [
    {
      "id": "did:plc:53ni6w7okqrxvjxezmopejoo#atproto",
      "type": "Multikey",
      "controller": "did:plc:53ni6w7okqrxvjxezmopejoo",
      "publicKeyMultibase":
"zQ3shPWwfJKjriJqb4CsR7KBBHkaffnpZin5NCyAYJzztnuRV"
    },
    {
      "id": "did:plc:53ni6w7okqrxvjxezmopejoo#atproto_kem",
      "type": "Multikey",
      "controller": "did:plc:53ni6w7okqrxvjxezmopejoo",
      "publicKeyMultibase":
"zDnaeueFwjSjPA49dvab7bN4VjYDvv7eBVsDSjncdQHsud125"
    }
  ],
  "service": [
    {
      "id": "#atproto_pds",
      "type": "AtprotoPersonalDataServer",
      "serviceEndpoint": "https://pds-a.example.com"
    }
  ]
}
```

Signed genesis operation:

JSON

```
{
  "type": "plc_operation",
  "rotationKeys": [
    "did:key:zQ3shiEHng2LXQA12HNn54XzBTquzZprwYjCTKu1ySPh71N6r"
  ],
  "verificationMethods": {
    "atproto":
"did:key:zQ3shPWwfJKjriJqb4CsR7KBBHkaffnpZin5NCyAYJzztnuRV",
```

```
    "atproto_kem":
"did:key:zDnaeueFwjSjPA49dvab7bN4VjYDvv7eBVsDSjncdQHsud125"
  },
  "alsoKnownAs": [
    "at://elena.example.com"
  ],
  "services": {
    "atproto_pds": {
      "type": "AtprotoPersonalDataServer",
      "endpoint": "https://pds-a.example.com"
    }
  },
  "prev": null,
  "sig": "URh4sPB-AiJUI09Pax23GpK5PL-
o8Ei5FKwI_C0kV8EoQcRde0yUUYX_slPGw5bokzhSwqZ-QsDy8xkatgfLPg"
}
```


F.3 Ruth

Table F.4 · Ruth's keys

Key	Type	Private key	Public key
`#atproto`	K-256	`z3vLhSfej1aqiMWBvhjyjTuXM3L7QvFh2J4DKhAdPdtAmAd3`	`zQ3shtrHbFC5PwLhynynQQQftQcnZeUJ79kx2PvLacgJ5NpLq`
`#atproto_kem`	P-256	`z42tzGEBNL7iDdZLbfZuPyrUr8XFAsrdzhpZimyUidHZkyrZ`	`zDnaefddyQ4HYrC15kpiBjhAmUcmm6wVqDqNSFWatYRUiHaxy`
Rotation	K-256	`z3vLVMBwrNZXbuPf6ji4zUEZcHH7rFUk5ZwZqu2Ed7kJNgFt`	`zQ3shj6RvpC1UQdemwpNLbEUbUd1Bo7Nfm4gUGzyqpfpe27jt`

DID document for `did:plc:p4256dosynj6j6tkypz2q2l3`:

JSON


```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/multikey/v1",
    "https://w3id.org/security/suites/secp256k1-2019/v1",
    "https://w3id.org/security/suites/ecdsa-2019/v1"
  ],
  "id": "did:plc:p4256dosynj6j6tkypz2q2l3",
  "alsoKnownAs": [
    "at://ruth.example.com"
  ],
  "verificationMethod": [
    {
      "id": "did:plc:p4256dosynj6j6tkypz2q2l3#atproto",
      "type": "Multikey",
      "controller": "did:plc:p4256dosynj6j6tkypz2q2l3",
      "publicKeyMultibase":
"zQ3shtrHbFC5PwLhynynQQQftQcnZeUJ79kx2PvLacgJ5NpLq"
    },
    {
      "id": "did:plc:p4256dosynj6j6tkypz2q2l3#atproto_kem",
      "type": "Multikey",
      "controller": "did:plc:p4256dosynj6j6tkypz2q2l3",
```

```
      "publicKeyMultibase":
"zDnaefddyQ4HYrC15kpiBjhAmUcmm6wVqDqNSFWatYRUiHaxy"
    }
  ],
  "service": [
    {
      "id": "#atproto_pds",
      "type": "AtprotoPersonalDataServer",
      "serviceEndpoint": "https://pds-b.example.com"
    }
  ]
}
```

Signed genesis operation:

JSON

```
{
  "type": "plc_operation",
  "rotationKeys": [
    "did:key:zQ3shj6RvpC1UQdemwpNLbEUbUd1Bo7Nfm4gUGzyqpfpe27jt"
  ],
  "verificationMethods": {
    "atproto":
"did:key:zQ3shtrHbFC5PwLhynynQQQftQcnZeUJ79kx2PvLacgJ5NpLq",
    "atproto_kem":
"did:key:zDnaefddyQ4HYrC15kpiBjhAmUcmm6wVqDqNSFWatYRUiHaxy"
  },
  "alsoKnownAs": [
    "at://ruth.example.com"
  ],
  "services": {
    "atproto_pds": {
      "type": "AtprotoPersonalDataServer",
      "endpoint": "https://pds-b.example.com"
    }
  },
  "prev": null,
  "sig":
"AaPEuAAZ6K94a7QTnEnfeftCp15mfulaTYfAXFkPVtJA5jjp1_DJGL4IwomgNvx8DAVYeEDOYh5UnJfQa2w
}
```

F.4 Nora

Table F.5 · Nora's keys

Key	Type	Private key	Public key
#atproto	K-256	z3vLjrQAHLD4mz7Qp4huAh9zmcu9KnJsePE992XCpU3Kezw	did:key:zQ3shXqjG1uHZHmwtPA1sQzqVJB3J7r1FHJNTPPLHNdwkdMei
#atproto_kem	P-256	z42tkHewqpP6Uec4E4Y4ZqUcotouSc5LqCDYD6SDXd3Kej8MU	did:key:zDnaefZoqiT1a5Z5yQdogzNncMBp9VeNKybzCtHY5aDFZ1AAS
Rotation	K-256	z3vLWJ9Skm5WhA6ZQhMKmgSi5JBAEeSv6JzVxdHLidgkfC2T	did:key:zQ3shbAyeCwQBgG2T67tkwA4aifdPyA6BCdArGHm35eNeL3TH

DID document for `did:plc:ashwgt3d5ntfxmlcfyizpu2g`:

JSON

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/multikey/v1",
    "https://w3id.org/security/suites/secp256k1-2019/v1",
    "https://w3id.org/security/suites/ecdsa-2019/v1"
  ],
  "id": "did:plc:ashwgt3d5ntfxmlcfyizpu2g",
  "alsoKnownAs": [
    "at://nora.example.com"
  ],
  "verificationMethod": [
    {
      "id": "did:plc:ashwgt3d5ntfxmlcfyizpu2g#atproto",
      "type": "Multikey",
      "controller": "did:plc:ashwgt3d5ntfxmlcfyizpu2g",
      "publicKeyMultibase":
"zQ3shXqjG1uHZHmwtPA1sQzqVJB3J7r1FHJNTPPLHNdwkdMei"
    },
    {
      "id": "did:plc:ashwgt3d5ntfxmlcfyizpu2g#atproto_kem",
      "type": "Multikey",
      "controller": "did:plc:ashwgt3d5ntfxmlcfyizpu2g",
      "publicKeyMultibase":
"zDnaefZoqiT1a5Z5yQdogzNncMBp9VeNKybzCtHY5aDFZ1AAS"
    }
  ],
  "service": [
    {
      "id": "#atproto_pds",
      "type": "AtprotoPersonalDataServer",
      "serviceEndpoint": "https://pds-b.example.com"
    }
  ]
}
```

Signed genesis operation:

JSON

```
{
  "type": "plc_operation",
  "rotationKeys": [
    "did:key:zQ3shbAyeCwQBgG2T67tkwA4aifdPyA6BCdArGHm35eNeL3TH"
  ],
  "verificationMethods": {
    "atproto":
"did:key:zQ3shXqjG1uHZHmwtPA1sQzqVJB3J7r1FHJNTPPLHNdwkdMei",
    "atproto_kem":
"did:key:zDnaefZoqiT1a5Z5yQdogzNncMBp9VeNKybzCtHY5aDFZ1AAS"
  },
  "alsoKnownAs": [
    "at://nora.example.com"
  ],
  "services": {
    "atproto_pds": {
      "type": "AtprotoPersonalDataServer",
      "endpoint": "https://pds-b.example.com"
```

```
    }
  },
  "prev": null,
  "sig": "DhE2FTE6D99kuwN7eTdL23aNQJAUQc5PZO9A-
SJEWOcZyyxu8iUd533MBeKpNeS0v6pzrod8uGu6SathZWdKxg"
}
```


F.5 Jake

Table F.6 · Jake's keys

Key	Type	Private key	Public key
#atproto	K-256	z3vLaKimzb4aMSMGyQzFNLqQwmSHgme7z8gL4NJCidzkeyGm	zQ3shV9QNX7FEXYyRtMqgxHQ2LsqPU4NhJX6UZUZ6QMxy1FjX
#atproto_kem	P-256	z42tkUDumnF9QyqtjqXsoU7GtSQF1D5tWgSzFL1UUdjkeyGF	zDnaehVhdn1N89vzNfh2ez4QyqL6tuk6QY49jR85CLYnrUkAn
Rotation	K-256	z3vLhqMrdmALa6heueBHHKv2p2JcyYEaQrA7cdrydtwqFyFz	zQ3shQuSUsNpucKZdbVmTnw2fdnzj4M3PoNZ2LXEoj3hFsnVc

DID document for `did:plc:brthxhhoomvrib2avauses35`:

JSON


```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/multikey/v1",
    "https://w3id.org/security/suites/secp256k1-2019/v1",
    "https://w3id.org/security/suites/ecdsa-2019/v1"
  ],
  "id": "did:plc:brthxhhoomvrib2avauses35",
  "alsoKnownAs": [
    "at://jake.example.com"
  ],
  "verificationMethod": [
    {
      "id": "did:plc:brthxhhoomvrib2avauses35#atproto",
      "type": "Multikey",
      "controller": "did:plc:brthxhhoomvrib2avauses35",
      "publicKeyMultibase":
"zQ3shV9QNX7FEXYyRtMqgxHQ2LsqPU4NhJX6UZUZ6QMxy1FjX"
    },
    {
      "id": "did:plc:brthxhhoomvrib2avauses35#atproto_kem",
      "type": "Multikey",
      "controller": "did:plc:brthxhhoomvrib2avauses35",
      "publicKeyMultibase":
"zDnaehVhdn1N89vzNfh2ez4QyqL6tuk6QY49jR85CLYnrUkAn"
    }
  ],
  "service": [
    {
      "id": "#atproto_pds",
      "type": "AtprotoPersonalDataServer",
      "serviceEndpoint": "https://pds-b.example.com"
    }
```

```
  ]
}
```

Signed genesis operation:

JSON

```
{
  "type": "plc_operation",
  "rotationKeys": [
    "did:key:zQ3shQuSUsNpucKZdbVmTnw2fdnzj4M3PoNZ2LXEoj3hFsnVc"
  ],
  "verificationMethods": {
    "atproto":
"did:key:zQ3shV9QNX7FEXYyRtMqgxHQ2LsqPU4NhJX6UZUZ6QMxy1FjX",
    "atproto_kem":
"did:key:zDnaehVhdn1N89vzNfh2ez4QyqL6tuk6QY49jR85CLYnrUkAn"
  },
  "alsoKnownAs": [
    "at://jake.example.com"
  ],
  "services": {
    "atproto_pds": {
      "type": "AtprotoPersonalDataServer",
      "endpoint": "https://pds-b.example.com"
    }
  },
  "prev": null,
  "sig": "U1jzcI_AG-af2WivluvNF-
OWIAQqjgzVj8a57lr7g01kQ83hhhfXh_7-9Fe9T2SlRHpDX2E6pvn7qrIkS_8Z2A"
}
```

F.6 Erin

Erin's identity is a `did:web` with no PDS account, so her document is served at `https://erin.example/.well-known/did.json` and names no services. Chapter 6 uses her to show an identity reading a space with a delegation token it signs itself. Her credential key is the P-256 key her space credential is bound to.

Table F.7 · Erin's keys

Key	Type	Private key	Public key
`#atproto`	K-256	`z3vLm5xczeGnwY8V2rrp2owhBUCerinCkryK2GvA34Jn5AzV`	`zQ3shvWu4gufp9Qp4fCXSRtKdvBeX6nGVEjkvYuE2s3pYgAtz`
Credential key	P-256	`z42txN6maM9D6qsfrrFhxZ54pfAovoTRuEoq3kaPPHyz8AuN`	`zDnaerESMGLw8NRXkwfGkQvpQ9u2R26RMXChzjkUbp9M1K7iw`

JSON

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/multikey/v1",
```

```
    "https://w3id.org/security/suites/secp256k1-2019/v1"
  ],
  "id": "did:web:erin.example",
  "alsoKnownAs": [
    "at://erin.example"
  ],
  "verificationMethod": [
    {
      "id": "did:web:erin.example#atproto",
      "type": "Multikey",
      "controller": "did:web:erin.example",
      "publicKeyMultibase":
"zQ3shvWu4gufp9Qp4fCXSRtKdvBeX6nGVEjkvYuE2s3pYgAtz"
    }
  ]
}
```

F.7 The managing app

The managedspace example names `did:web:newsletter.example#newsletter_app` as its `managingApp`. It is a `did:web` identity, so its document is served at `https://newsletter.example/.well-known/did.json`. The spec fixes only the `did#serviceId` reference, so the service type here is illustrative.

Table F.8 · The managing app's key

Key	Type	Private key	Public key
`#atproto`	K-256	`z3vLiTruhHZscwSy1SS8WZ5ZV2GzAPkKCCzBvwfDEdYdBkT`	`zQ3shhtKcZp4cAVPUPynLP7qZf5wQhEMXbY13fEsfoT3JtZqE`

JSON

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/multikey/v1",
    "https://w3id.org/security/suites/secp256k1-2019/v1"
  ],
  "id": "did:web:newsletter.example",
  "verificationMethod": [
    {
      "id": "did:web:newsletter.example#atproto",
      "type": "Multikey",
      "controller": "did:web:newsletter.example",
      "publicKeyMultibase":
"zQ3shhtKcZp4cAVPUPynLP7qZf5wQhEMXbY13fEsfoT3JtZqE"
    }
  ],
  "service": [
    {
      "id": "#newsletter_app",
      "type": "AtprotoManagedSpaceApp",
      "serviceEndpoint": "https://newsletter.example"
    }
```

```
  ]
}
```

F.8 The family app

The family app signs its client attestations with an ES256 key published in its client metadata, under `kid key-1`.

Table F.9 · The family app's client key

Key	Type	Private key	Public key
Client attestation key, `key-1`	P-256	`z42tuwr2CJijDjeAGXJ8woV2kwPuJrMKi4bKeyGo3zKaivKC`	`zDnaeR2kuaWh2w7c4fhAzdDeeACMs2wpCPoV9hVj1thrm`

The same key as a private JWK:

JSON

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "COmaVfdUzj6QHwm3ZTZgY8KsbNhtJN12EgFWLsm7_pE",
  "y": "G36xIW2qad2tMbOE8zj-pSq39311l313hbZbcAPTxKw",
  "d": "szP7wc1M4WX3Fdb_md_ADuF5fUUsVLLO2Kv5-oQnsos",
  "kid": "key-1",
  "alg": "ES256",
  "use": "sig"
}
```

Its client metadata, at `https://family.example/client-metadata.json`, publishes the public half:

JSON

```
{
  "client_id": "https://family.example/client-metadata.json",
  "client_name": "Family",
  "client_uri": "https://family.example",
  "application_type": "web",
  "grant_types": [
    "authorization_code",
    "refresh_token"
  ],
  "response_types": [
    "code"
  ],
  "redirect_uris": [
    "https://family.example/oauth/callback"
  ],
  "scope": "atproto space:com.example.familyUpdates?authority=*
space:com.example.familyUpdates?manage=create&manage=update
blob:image/*",
  "token_endpoint_auth_method": "private_key_jwt",
  "token_endpoint_auth_signing_alg": "ES256",
  "dpop_bound_access_tokens": true,
```

```
  "jwks": {
    "keys": [
      {
        "kty": "EC",
        "crv": "P-256",
        "x": "COmaVfdUzj6QHwm3ZTZgY8KsbNhtJN12EgFWLsm7_pE",
        "y": "G36xIW2qad2tMbOE8zj-pSq39311l313hbZbcAPTxKw",
        "kid": "key-1",
        "alg": "ES256",
        "use": "sig"
      }
    ]
  }
}
```


As a syncer, the app also holds credential keys, which `authorizeSpaceSync` binds as `credentialKey` and which sign every request presenting the credential, and an ephemeral KEM key whose `did:key` is the `keyid` of a MAC request. Real syncers make a new ephemeral key for every request; the examples use this one so their values can be reproduced.

Table F.10 · The family app's syncer keys

Key	Type	Private key	Public key
Credential key, authorization through Elena's session	P-256	`z42trsj4FFwSja7KNi4gmPVajBmeAN…`	`…WX79bT5Bmk` `zDnaeyJrXggfgfsMLx91wQDoHqk5rMmGtE4nkXGG3x5Xz`
Credential key, authorization through Jake's session	P-256	`z42toDGPtYaHhofXZwdxCJuzr48Rfs…`	`…A1WN8JXqcGi` `zDnaepyARB3jVLncHov8c5rxMEBtkdQwE6xfaXmWUNzKH`
Ephemeral KEM key	P-256	`z42tj5gMgfCRAzEGTszobvjz8QaxKc…`	`…13FB6qBh3` `zDnaenqFhbY4sffBMz5ZatBohkr8LNNAe1pWTmG6C88rM`

The app registers for notifications at `https://family.example`. Its notify token for the family space is the base64url HMAC-SHA256 of the space URI, keyed by the 32 bytes of an app secret.

Table F.11 · The family app's notification settings

Setting	Value
Notify secret	`3514c7e82b3a2e90772a496a8fde3406a0b60279891ed3d2dbfdbb6194b75b80`
Notify token	`xIFHpilmY0Wlfxq62sqovyfp66i08zazz2dgpsT_38E`

For OAuth, the app binds each session's access tokens to a DPoP key and signs a DPoP proof with it on every request. Real apps make a new DPoP key for each session; these five are fixed so the examples can be reproduced. The `jkt` is the key's RFC 7638 thumbprint, which the access token carries as `cnf.jkt`.

Table F.12 · The family app's DPoP keys

Key	Type	Private key	Public key	jkt
DPoP key, Marcus's session	P-256	z42tkZ7DxxksEvA3DDgeGNAHiMKeyMmQjur18KwiRZWEkD2a6wePvzKvQRC05Prf02qcTddiDPE156T5t4	zDnaeYN2ihXVw5D3PSpcogqGGdRihGCxrTViNP38Wk1dPoDD5	
DPoP key, Elena's session	P-256	z42tzeqcjvjekYFJtsrQ85odcAAbv8KMHTG3Xtgq7KucAbGStvuegh8iSNaDvxKEvVRWMmTfIB_rWiyR6X	zDnaegjsHj1GQBUmPuBcJPZjKPVDaeyPVVrbGP8fq1VEFor5w	
DPoP key, Ruth's session	P-256	z42tjHp7DDLVZJJTHHjC3u7KfFoKey5CFnqxPvsiX9NYZoPSaM04WcMq9e2H9REzOldNOnjksltWEMBSdn9	zDnaecShktUyk9u9sgVEvLQWH427TzfFSbjrCoL88SF9gBh2S	
DPoP key, Nora's session	P-256	z42tpNtJuehD64TtCqwydctdAFXhe2d4js9877LypLJBiqkRDp8F5NcGr6rK-	zDnaeVx9oDH8HWE2xiSBQajbF3p2PPrN5NgGRSg3qJOZA5ZDqBY	
DPoP key, Jake's session	P-256	z42txz64hK4ZR3TZk7Ne4KSdiABUWkb7HMfVTPSBJkbJG6NJ04uaJmFpHjcgOZGSfM05u3nOMM5ePlwdCri	zDnaecbB8WRUevzafNscSNjZbgmGGfSg8zPuJqCoE8BWQaUQE	

The same keys as private JWKs, in the same order:

JSON

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "ddtTbllFr3qROgtUOrIp7XI8Ix8RVFVx1Qf1JVpm2_w",
  "y": "aZhmESA3LSdjidIUmA1U6HcBP35RoWf3cvtHfkxtUSY",
  "d": "J6eh6rQ_DgnxOxSVbuDchycL1SyvrYUlPjNgoKigtrM"
}
```

JSON

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "8lDvl35BUkpVY4UdECy4rbbkkc8JCt-RC92LrX13fzI",
  "y": "02AL1e4xcocPvFZ9w1teKY8hKt1CpAtdACacmkd0mrg",
  "d": "-SMWSTOTZU66FTvX3IOhj7CgP2LmU6fw8Qh-vDsjChU"
}
```

JSON

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "snyJdgjOVLSQxH76MkMxskPpWIJPQcelRaj8mvfghPM",
  "y": "mrbJXlc9oNNUsemf51C2n6Tfy1a2U8VFlxgc90lQM3g",
  "d": "FODhOeYjtauf8aTgyh87XXqCEZUeQSOP_Y7AUr8e350"
}
```

JSON

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "UgYD-j0pgwtCZ9Y6ktVIfQaAOBLvyqElTcR-Vkuwgag",
  "y": "g7Wb-GnjAXDgsSwWHLqV7ojb5BbG4AeJ0NINkOd9_bI",
  "d": "YHf4JSSvDvYIjW7wbfXtB1EIXcxeFCnUOekgHxDWBWo"
}
```

JSON

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "tKgiR0Oy5eC1yim9_3o3ryhf9zzw1ROyexdYllL3o8c",
  "y": "Bin6n3KryPiMySG4WOUNaOmJStJFuMttCgo-ABR0ZDo",
  "d": "4FoVnBP9IppzVMzhU0pUBspXgkFGRR5Law6LOScMJ5Y"
}
```

F.9 The repo hosts

Marcus and Elena have their accounts on `pds-a.example.com`, and Ruth, Nora, and Jake have theirs on `pds-b.example.com`. Each host is also the OAuth authorization server for its accounts.

Table F.13 · The repo hosts' settings

Setting	pds-a	pds-b
Issuer	https://pds-a.example.com	https://pds-b.example.com
Service DID, the access token `aud`	did:web:pds-a.example.com	did:web:pds-b.example.com
JWT secret	ae7a49de5c93efd16a91304ed7bfb65d3a	352365130d5aba948d
DPoP nonce secret	f7fb075a969b2196ab50887ce2e826bd4f	bcf9c039dd5fa067c8
Nonce rotation	60 seconds	60 seconds

The host signs access tokens with HS256, keyed by the JWT secret's text as UTF-8 bytes, which is how the reference PDS uses its `PDS_JWT_SECRET`. Access tokens are opaque to clients, so another host may use a different format.

A DPoP nonce is the base64url HMAC-SHA256, keyed by the 32 bytes of the nonce secret, of the rotation counter as a big-endian 64-bit integer. The counter is the Unix time in milliseconds divided by 60,000, rounded down. The host sends the next counter's nonce in `DPoP-Nonce` and accepts the previous, current, and next ones.

Appendix G · Worked values

Every value in the examples was computed from the keys in Appendix F with the reference `@atproto/space` package, then checked independently. This appendix holds the values the chapters shorten, along with the inputs needed to reproduce them. The tokens were issued on 2026-11-16 UTC and have long expired, so check their signatures, not their expiry.

G.1 Records

Table G.1 · Example records

Record	Owner	rkey	CID
Post with two photos	Elena	`3n2gsittdk22e`	`bafyreifomtdsjliazvhlq5w25vj64mrqpo7osdmineav`
Post	Ruth	`3n2gsrrwmc22p`	`bafyreiexyeo5zjfppsf2qvf7wmkwy7lf6e2k5pnzr4l`
Family reunion event	Marcus	`3n2gsywstc22e`	`bafyreigjlo4wnf5e6inmlkp7ib7pcym5o34vblhnbrf`
RSVP, interested	Nora	`3n2gtdo55s22p`	`bafyreiautip3hgihuw4ypueftfdiv3iyliieamu5mdv`
RSVP, interested	Ruth	`3n2gtkszes22p`	`bafyreiautip3hgihuw4ypueftfdiv3iyliieamu5mdv`
RSVP, going	Ruth	`3n2gtxdkr222p`	`bafyreiblr6f6oy3zi426nksahddwp4tgsfgedtqk27f`
Bookmark	Jake	`3n2gtrxvls22p`	`bafyreibeecgpzsoedv7jcxr5tf6ggkmpdpolwvq65eh`
`!takedown` label	Marcus	`3n2gu2vyuk22e`	`bafyreihfatmz52lbyzkb2oqyjxtzf3v5cz4dgjqxz6k`
Policy record	Marcus	`self`	`bafyreifov3toaen6phrdukkj2hdtolxs7x4ucij2ish`
Member record for Elena	Marcus	`did:plc:53ni6w7okqrxvjxezmopejoo`	`bafyreid4u6fj5zhcje5vqnazohfm7ezbdrononwnfgv`

The RSVP values are in Chapter 4. The event record is:

JSON

```
{
  "$type": "community.lexicon.calendar.event",
  "name": "Family reunion",
  "description": "A weekend at the lake. Bring a dish and a folding
chair.",
  "startsAt": "2027-07-17T16:00:00.000Z",
  "endsAt": "2027-07-18T22:00:00.000Z",
  "mode": "community.lexicon.calendar.event#inperson",
  "status": "community.lexicon.calendar.event#scheduled",
  "createdAt": "2026-11-16T15:15:00.000Z"
}
```

The blobs in Chapter 7's `listBlobs` and `getBlob` examples are the two photos in Elena's post. Both are JPEG files from the fixture generator's `tools/fixtures/assets/` folder.

Table G.2 · The photos in Elena's post

File	CID	Size	SHA-256
theo-blanket.jpg	bafkreigne4u5q3h5hdcbhusc5osdwybrlkd7dl72b3ufbklgcrb2r2ddbcfd38c413d242eba43b60315a86f1affad6a850	1,702 bytes	
theo-bath.jpg	bafkreicy6rib4mitvvzbsmj3rktg2fpihkf4fomlgc4kvqwpvbjf45ole3	1,468 bytes	5f4501e3113ad7219313b8aa66d15e83a8a72b98d30b8ac

G.2 Ruth's commits

Ruth's first commit holds her post. Her second adds the interested RSVP. Her `applyWrites` call creates the going RSVP and deletes the first one, so the third commit holds the post and the new RSVP. Each `data` value is the LtHash digest of the set's element strings.

Table G.3 · Ruth's commits

rev	Element strings	`data` (base64)	`data` (hex)
3n2gsrrwmc32p	com.example.familyUpdates.post/3n2gsrrwmc22p/bafyreiexyeo5zjfppsf2qvf7wmkwy7lf6e2k5pnzr4ll2oxibmvizimpmu	11GKdFSf6OpfEshR8qwODBc4Tgnstd7518sVlUldo7w	d7518a74549fe8ea5f12c851f2ac0e0c17384e09ecb
3n2gtkszes32p	com.example.familyUpdates.post/3n2gsrrwmc22p/bafyreiexyeo5zjfppsf2qvf7wmkwy7lf6e2k5pnzr4ll2oxibmvizimpmu and community.lexicon.calendar.rsvp/3n2gtkszes22p/bafyreiautip3hgihuw4ypueftfdiv3iyliieamu5mdv4d4duqhhyetxs7a	/SA48RWigXRyODioVTDgAI4L/xhBodK94P7Cusxye9Q	fd2038f115a28174723838a85530e0008e0bff1841a
3n2gtxdkr232p	com.example.familyUpdates.post/3n2gsrrwmc22p/bafyreiexyeo5zjfppsf2qvf7wmkwy7lf6e2k5pnzr4ll2oxibmvizimpmu and community.lexicon.calendar.rsvp/3n2gtxdkr222p/bafyreiblr6f6oy3zi426nksahddwp4tgsfgedtqk27fyxqtav66kjvwkgu	eVD7vKgDQrLKfOvN3VZJHYX0HGz15795KfbbrBK3Ab2ca7cebcddd56491d85f41c6cf5e	

The empty digest is `e5a00aa9991ac8a5ee3109844d84a55583bd20572ad3ffcd42792f3c36b183ad`.

G.3 Tokens

The delegation token Elena's PDS signs with her `#atproto` key:

TEXT

```
eyJhbGciOiJFUzI1NksiLCJ0eXAiOiJhdHByb3RvLXNwYWNlLWRlbGVnYXRpb24rand0Iiwia2lkIjoiI2F0
hlQvyJPVdoOADyuptkdq1VkYmA
```

The client attestation the family app signs with its `key-1` client key:

TEXT

```
eyJhbGciOiJFUzI1NiIsInR5cCI6ImF0cHJvdG8tY2xpZW50LWF0dGVzdGF0aW9uK2p3dCIsImtpZCI6Imtl
vdJJp22zprnFla_vAu4QcshAiuwA
```

The space credential Marcus's PDS signs with Marcus's `#atproto` key, bound to the syncer's first credential key:

TEXT

```
eyJhbGciOiJFUzI1NksiLCJ0eXAiOiJhdHByb3RvLXNwYWNlLWNyZWRlbnRpYWwrand0Iiwia2lkIjoiI2F0
ELKxFgo3kjlFWH6frlUd38sRbm45zxJGhjLg
```

The service auth token Ruth's PDS sends with `notifyRepoUpdate` for her batch (`lxm` `com.atproto.space.notifyRepoUpdate`, `aud` `did:plc:ejjl2znzqp3ci25dqaslpkk2#atproto_pds`):

TEXT

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NksifQ.eyJpYXQiOjE3OTQ4NDMxMjAsImlzcyI6ImRpZDpwbGM6c
hbmHUVaIP7TffbGAR_f386xFe4PQxzkCGKXg_X8_3qgu284MEA-DsqE__ukhNJqUwtoKQ
```

The service auth token a newsletter authority's PDS sends with `checkAccess` (`lxm` `com.atproto.managedspace.checkAccess`, `aud` `did:web:newsletter.example#newsletter_app`). Its issuer, `did:plc:ldianl3maiaomvmvyyt3xzhh`, is outside the family, so its key is not in Appendix F:

TEXT

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NksifQ.eyJpYXQiOjE3OTQ4NDQyMDAsImlzcyI6ImRpZDpwbGM6b
ZSRlFcXf2jQ
```

The delegation token Erin signs herself with her `#atproto` key:

TEXT

```
eyJhbGciOiJFUzI1NksiLCJ0eXAiOiJhdHByb3RvLXNwYWNlLWRlbGVnYXRpb24rand0Iiwia2lkIjoiI2F0
cQ5I2w79sJztL3sEXCKtMb017bweNTqMiuZa6Ar_bpKhCGLb-eSju3R66ipUqjH-qg
```

The space credential Marcus's PDS issues to Erin, bound to her credential key:

TEXT

```
eyJhbGciOiJFUzI1NksiLCJ0eXAiOiJhdHByb3RvLXNwYWNlLWNyZWRlbnRpYWwrand0Iiwia2lkIjoiI2F0
Uv3sw
```

The space credential from the authorization through Jake's session, bound to the syncer's second credential key. Its `jti`, `8e78341f7da46cf37e3cbd8611ff6bbc`, is the one in Chapter 6's revocation notice:

TEXT

```
eyJhbGciOiJFUzI1NksiLCJ0eXAiOiJhdHByb3RvLXNwYWNlLWNyZWRlbnRpYWwrand0Iiwia2lkIjoiI2F0
qw67wYiTa0K2h6xosjFN_sILBtk4Bc_oPp280UcbU0jin29wbAz3LmrjzEKhyVPCVg
```

G.4 OAuth requests

Each request to an account's PDS carries `Authorization: DPoP` with the session's access token and a `DPoP` header with a fresh proof. The proof is signed with the session's DPoP key and binds the method (`htm`), the URL without its query (`htu`), the time (`iat`), the latest nonce from that host, and the access token's SHA-256 (`ath`). An independent verifier checked each of the 17 proofs.

Table G.4 · OAuth sessions

Session	Account	Granted scope	Access token `jti`	Issued (UTC)
Marcus	did:plc:ejjl2znzqp3ci25dqaslpkke	atproto space:com.example.familyUpdates?manage=create&manage=update	tok-bf17e9643686a1fd1a380505a1d99669	15:00:00, valid 1 hour
Elena	did:plc:53ni6w7okqrxvjxezmopejoa	atproto space:com.example.familyUpdates?authority=* blob:image/*	tok-c9cdda2f905e26004eada60aaeba39ad	15:05:00, valid 1 hour
Ruth	did:plc:p4256dosynj6j6tkypz2q21a	atproto space:com.example.familyUpdates?authority=*	tok-48f499280c76c76fe8d[illegible]	15:10:00, valid 1 hour
Nora	did:plc:ashwgt3d5ntfxmlcfyizpuzx	atproto space:com.example.familyUpdates?authority=*	tok-819f3e246cf872d8d10[illegible]	15:20:00, valid 1 hour
Jake	did:plc:brthxhhoomvrib2avauses35	atproto space:com.example.familyUpdates?authority=*	tok-64f632c615bb640f2f5[illegible]	15:25:00, valid 1 hour

The host keeps the granted scope with the session rather than in the token. Marcus's access token:

TEXT

```
eyJ0eXAiOiJhdCtqd3QiLCJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJ0b2stYmYxN2U5NjQzNjg2YTFmZDFhMzg
Huvkl01n8_ITEMiA8ZcPZA6s
```

Elena's access token:

TEXT

```
eyJ0eXAiOiJhdCtqd3QiLCJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJ0b2stYzljZGRhMmY5MDVlMjYwMDRlYWF
```

Ruth's access token:

TEXT

```
eyJ0eXAiOiJhdCtqd3QiLCJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJ0b2stNDhmNDk5MjgwYzc2Yzc2ZmU4ZDI
```

Nora's access token:

TEXT

```
eyJ0eXAiOiJhdCtqd3QiLCJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJ0b2stODE5ZjNlMjQ2Y2Y4NzJkOGQxMDI
ronZqFtE0lBQHOZISPe1ew
```

Jake's access token:

TEXT

```
eyJ0eXAiOiJhdCtqd3QiLCJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJ0b2stNjRmNjMyYzYxNWJiNjQwZjJmNWQ
```

Table G.5 · OAuth requests

Request	Session	Time (UTC)	htm	Nonce
createSpace	Marcus	15:00:40	POST	tVRqqxjMuJ_uASFjmWWo5tPpGU9XSI69rjm3LgX3eqE
putMembers, adding the family	Marcus	15:01:40	POST	9eyTxuk5FNibHOyr3pEEtZnSJuKn-oq2BcJsZKvadMI
getSpace	Marcus	15:02:40	GET	rB4frOk6N3yQiNllYO7FR2L_MbbuXRDS8-AnISgQI_s
uploadBlob, first photo	Elena	15:05:30	POST	T_70AQqmPSi58KCWt64O1Cmaig5H9ySqaZf8TWZmO9Y
uploadBlob, second photo	Elena	15:05:45	POST	T_70AQqmPSi58KCWt64O1Cmaig5H9ySqaZf8TWZmO9Y
createRecord, Elena's post	Elena	15:06:00	POST	xI7s-uJd54unXioKCbMEWiCpke2XkOwMEWS8vYsnbWg
createRecord, Ruth's post	Ruth	15:11:00	POST	ayVahjoDne9Rur2ihZoe5DCkNtbfAG2tglHP-iz_XCI
createRecord, the reunion event	Marcus	15:15:00	POST	5wyQkCcRI2aguhdd4VtnfpA4vGevV-IY5Tbp9VX-UQY
createRecord, Nora's RSVP	Nora	15:21:00	POST	bgQbUNVCgHs9DpKfe4m_SA9GT1LAPo87MD1hYe1odiQ
createRecord, Ruth's first RSVP	Ruth	15:25:00	POST	Xt5D881tUiPZseFi2CIkjyHmL5Jtfhj4UoCeKV4eZVE
createRecord, Jake's bookmark	Jake	15:29:00	POST	Wm3c4m4p4sxbalrDPmf_EqbIYLqWE9OSFSMWahWbqgw
getDelegationToken	Elena	15:30:00	GET	btWGu_BTmZDwsq3lMCXibcCq_jr4N6PvkeGqzgVdxsA
applyWrites, Ruth's batch	Ruth	15:32:00	POST	jb_mqXrUBqYrHoOQkGjqHMDzoCmylNe1SxR9zK_OK5E
putMembers, making Jake read-only	Marcus	15:33:00	POST	NUknXUENM0vKBCtaJqTHs8_4_ey3VIKE2UNh2C0f2SU
createRecord, the !takedown label	Marcus	15:34:00	POST	PCyiQAiFwcxfBoXgjbcjJXA5Y6qoKJOm0-AwlYxPld4
getDelegationToken, through Jake's session	Jake	15:38:00	GET	ajuYzgyp9q5_yibQI1K8iBthCklOSYwNVHPl7VYZDKA
removeMembers	Marcus	15:39:00	POST	UfCttoUUWJGtJLzKt8DGJive5EdroZTbjoE2zrUkZHg

The chapters show seven of these requests, and their proofs follow.

The DPoP proof for createSpace (htu https://pds-a.example.com/xrpc/com.atproto.space.createSpace):

TEXT

```
eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0IiwiandrIjp7Imt0eSI6IkVDIiwiY3J2IjoiUC0yNTYi
hp6wonqjNkwiXABpXkdIK8uLlM6_PN_8KwQqsYBMuwwVA
```

The DPoP proof for `putMembers`, adding the family (`htu https://pds-a.example.com/xrpc/com.atproto.memberspace.putMembers`):

TEXT

```
eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0IiwiandrIjp7Imt0eSI6IkVDIiwiY3J2IjoiUC0yNTYi
```

The DPoP proof for `getSpace` (`htu https://pds-a.example.com/xrpc/com.atproto.space.getSpace`):

TEXT

```
eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0IiwiandrIjp7Imt0eSI6IkVDIiwiY3J2IjoiUC0yNTYi
FnVwgDZCLMbPBP1g9KlAhrTYndOPWvvms86nKCPmEv05daN9qKuErS9L5m1MqMXpQntDa6zyaPMIg
```

The DPoP proof for `createRecord`, Nora's RSVP (`htu https://pds-b.example.com/xrpc/com.atproto.space.createRecord`):

TEXT

```
eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0IiwiandrIjp7Imt0eSI6IkVDIiwiY3J2IjoiUC0yNTYi
kebqF72-wWQH1U3TY4EJxLzmDDypJxKMpvJJNDFJccUU-
XaEWXIZDhVcOHkI0QzPqL2ZIAkRrO0r4A
```

The DPoP proof for `getDelegationToken` (`htu https://pds-a.example.com/xrpc/com.atproto.space.getDelegationToken`):

TEXT

```
eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0IiwiandrIjp7Imt0eSI6IkVDIiwiY3J2IjoiUC0yNTYi
kgIdt6-
ry0irbkvzQV2NThNRhLp94LfnF0DdIQkE1PqBRFqPoSXPLCjJVqnHxX0zTWMVgaFFHUvxmQyWw
```

The DPoP proof for `applyWrites`, Ruth's batch (`htu https://pds-b.example.com/xrpc/com.atproto.space.applyWrites`):

TEXT

```
eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0IiwiandrIjp7Imt0eSI6IkVDIiwiY3J2IjoiUC0yNTYi
fX1ykIGnj91L48Wp9lcFSNwzDqBghUh7zPTuipUnLH0xf_iRDhKe6W9E_1Cy7XV0rYzcbBdCA
```

The DPoP proof for `removeMembers` (`htu https://pds-a.example.com/xrpc/com.atproto.memberspace.removeMembers`):

TEXT

```
eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3Arand0IiwiandrIjp7Imt0eSI6IkVDIiwiY3J2IjoiUC0yNTYi
BIIpdAyMjTqUYdMNES1kWHQYr07zw
```

G.5 Key exchange and MACs

The syncer's MAC requests carry its ephemeral KEM key as `keyid`. Ruth's PDS decapsulates against her `#atproto_kem` key and exports the shared secret, which is the HMAC-SHA256 key for both MACs.

Table G.6 · Key exchange inputs

Input	Value
Syncer's key share (`keyid`)	`did:key:zDnaenqFhbY4sffBMz5ZatBohkr8LNNAe1pWTmG6C88rMc477`
Ruth's `#atproto_kem`	`did:key:zDnaefddyQ4HYrC15kpiBjhAmUcmm6wVqDqNSFWatYRUiHaxy`
Shared secret (hex)	`96523627ef6c8bc21ab2ba80b70ed2e4e119317353195d4a2934bb2acd3f2f10`

The commit MAC's signature base is two lines joined by a single LF, with no trailing newline:

TEXT

```
"atproto-space-commit";sf: ver=1,
repo="did:plc:p4256dosynj6j6tkypz2q2l3", space="at:space/
did:plc:ejjl2znzqp3ci25dqaslpkk2/com.example.familyUpdates/our-
family", rev="3n2gtxdkr232p",
data=:eVD7vKgDQrLKfOvN3VZJHYX0HGz158ocKa5oRd3kaYw=:
"@signature-params": ("atproto-space-
commit";sf);keyid="did:key:zDnaenqFhbY4sffBMz5ZatBohkr8LNNAe1pWTmG6C88rMc477";tag="a
space-commit"
```

Its MAC is `qcuN1a76R+6Aq9Drm6C0aSYroDZp08eeMcArYT+vd/M=`.

The record MAC's signature base:

TEXT

```
"atproto-space-record";sf: uri="at:space/
did:plc:ejjl2znzqp3ci25dqaslpkk2/com.example.familyUpdates/our-family/
did:plc:p4256dosynj6j6tkypz2q2l3/
community.lexicon.calendar.rsvp/3n2gtxdkr222p",
cid="bafyreiblr6f6oy3zi426nksahddwp4tgsfgedtqk27fyxqtav66kjvwkgu"
"@signature-params": ("atproto-space-
record";sf);keyid="did:key:zDnaenqFhbY4sffBMz5ZatBohkr8LNNAe1pWTmG6C88rMc477";tag="a
space-record"
```

Its MAC is `bR41y+7bxRG0QywwNj6jb+fsh70sdOMUi0tmN9OsRRw=`.

G.6 Credential signatures

Each request that presents the credential is signed with the syncer's credential key (ECDSA P-256 with SHA-256, raw `r` and `s`). The signature base has three lines joined by LF, where `<credential>` is the syncer's first credential from the tokens above:

TEXT

```
"authorization": Atproto-Space-Credential <credential>
"atproto-space-audience": did:plc:ejjl2znzqp3ci25dqaslpkk2
"@signature-params": ("authorization" "atproto-space-
audience");tag="atproto-space-credential"
```

The signature base depends on the credential and the audience alone. The examples reuse one signature for each audience:

Table G.7 · Credential signatures by audience

Requests	Audience	Signature
`listWriters` to the space host	`did:plc:ejjl2znzqp3ci25dqaslpkk2`	`LwMAnDlLz8tSqIYx4l95CnFiycN4FY8e+DhgHy55waJFVxtZ`
`getLatestCommit`, `listRepoOps`, and `getRecord` to Ruth's host	`did:plc:p4256dosynj6j6tkypz2q2l3`	`aJSLd/+RWTwSPbsM8ZKMIWXqkzDTt67Lnbi0AaFcax8M0owwTdDkpi7BkhcI7Ls/5YbZmRHgd0CZTg==`
`listBlobs` and `getBlob` to Elena's host	`did:plc:53ni6w7okqrxvjxezmopejoo`	`Q/Ya6HSYi45IB5C1Of0GGLVVUAflbUmE033JyiRxNZgn0nf0rE`

Erin's `listWriters` request in Chapter 6 is signed the same way, with her credential from the tokens above in place of the syncer's, and with her own credential key:

Table G.8 · Erin's credential signature

Request	Audience	Signature
`listWriters` from Erin to the space host	`did:plc:ejjl2znzqp3ci25dqaslpkk2`	`QVs7ToEXTf7SUsuiF0mE9umJtzgctST1uId3TajXZyEqG7/mccBw3Ys04UEJuH2ofMCzsSKTxpicDm91/x3uFA==`

Appendix H · Glossary

The terms and acronyms used in this book, in alphabetical order. Where a chapter explains a term in depth, its entry says which.

AppView A service that collects atproto data from many accounts and builds what an app shows, such as feeds and threads. An AppView that keeps a copy of a space is that space's syncer.

AT URI The address of a record or a space. Public records use `at://`, and spaces and the records in them use `at:space/`. See Chapter 3.

atproto The AT Protocol, an open protocol for social apps in which each account's data lives in repos the account controls.

BLAKE3 A fast hash function. LtHash uses it to turn each record into 2048 bytes.

blob A file, such as an image, stored by an account's PDS and referenced from records by its CID.

CBOR Concise Binary Object Representation, a binary data format much like JSON.

CID Content identifier, a short string computed from a hash of some data, so the data can be found and checked by its content. Spaces use DASL CIDs.

client The app a person uses, on the web or a phone, which reads and writes on their behalf.

client attestation A token an app signs with its own key to show which app is asking for a credential. See Chapter 6.

commit The five-field summary of a space repo at one revision. See Chapter 4.

DASL Data-Addressed Structures & Links, a small set of specifications for CIDs and DRISL that atproto uses.

delegation token A 60-second token from a user's PDS that names the user and a space, so an app can ask for a credential on the user's behalf. See Chapter 6.

deniability The property that a response convinces the requester who received it, and no one else. See Chapter 5.

DID Decentralized identifier, the permanent identifier of an account or service, such as `did:plc:ejjl2znzqp3ci25dqaslpkk2`.

DID document The JSON document a DID resolves to. It lists the identity's handle, public keys, and services. See Chapter 2.

DPoP Demonstrating Proof of Possession, an OAuth addition in which every request is signed with a key tied to the access token.

DRISL DASL's strict form of CBOR. A record has exactly one DRISL encoding, so everyone computes the same CID for it.

ECDSA Elliptic Curve Digital Signature Algorithm, the kind of signature atproto keys make.

firehose The stream of every public repo change that a relay publishes. Spaces have no firehose.

handle A readable name for an account, such as `elena.example.com`, that resolves to its DID.

HMAC A MAC built from a hash function. Spaces use HMAC-SHA256.

HPKE Hybrid Public Key Encryption, RFC 9180. Spaces use only its secret export, to agree a secret for one request. See Chapter 5.

HTTP Message Signature A signature over parts of an HTTP request or response, carried in headers and defined by RFC 9421.

hyperspace The set of spaces an application treats as one whole under a single space authority: a primary space, the spaces in syzygy with it, and any inner spaces beneath it. See Chapter 3.

inner space A space beneath another space in the same hyperspace, with the same space authority and its own space key. See Chapter 3.

JWKS JSON Web Key Set, a list of public keys in JSON. An OAuth client publishes one so servers can check what it signs.

JWT JSON Web Token, a compact signed token made of a header, a set of claims, and a signature.

K-256 An elliptic curve that atproto signing keys can use, also called secp256k1.

KEM Key encapsulation mechanism, the part of HPKE that lets two parties derive a shared secret from a public key. It gives `#atproto_kem` its name.

key dimension The direction in a hyperspace that changes the space key, which is where inner spaces live. See Chapter 3.

lexicon A schema that defines a record type or an API method in atproto, named by an NSID.

LtHash The set hash that summarizes which records a space repo holds. See Chapter 5.

MAC Message authentication code, a short tag computed from a message and a secret. Only someone who holds the secret can make one or check one.

MST Merkle Search Tree, the signed tree that holds a public atproto repo. Space repos don't use one.

NSID Namespaced identifier, the reverse-domain name of a lexicon, such as `com.atproto.space.getSpace`.

OAuth The protocol an app uses to get a user's permission to act for them at their PDS. See Chapter 9.

op log A repo host's list of recent changes to a space repo, read with `listRepoOps`. See Chapter 7.

P-256 An elliptic curve that atproto keys can use, and the one spaces use for `#atproto_kem` and credential keys.

PDS Personal Data Server, the server that hosts an account with its repos and blobs. In spaces it is usually the account's repo host.

primary space The space in a hyperspace that holds the main content or community, and that the other spaces are organized around. See Chapter 3.

record One piece of data in a repo, such as an RSVP, typed by a lexicon.

relay A service that collects changes from every public repo and publishes them as a firehose.

repo A collection of one account's records. Public atproto has one per account. Spaces add one per account per space.

repo host The service that stores an account's space repos and answers sync requests for them. See Chapter 1.

rev A repo's revision, a TID that grows with every commit.

rkey Record key, the last segment of a record's URI, unique within its collection.

skey Space key, the segment that tells apart spaces of the same type under one authority, such as `our-family`. See Chapter 3.

space A named boundary that decides who can read a set of records. See Chapter 1.

space authority The DID at the root of a space. Its `#atproto` key signs the space's credentials. See Chapter 1.

space credential A short-lived token from the space host that lets an app read every repo in a space. See Chapter 6.

space host The service that authorizes syncers, tracks the writer set, and sends update notifications for a space. See Chapter 1.

space type The lexicon that says what kind of space it is, such as `com.example.familyUpdates`. See Chapter 3.

STAR The archive format `getRepo` returns, with the commit in a DRISL header followed by each record. See Chapter 7.

syncer An app that keeps its own copy of a space and serves views from it. See Chapter 1.

syzygy The alignment of two or more spaces that share a space authority and a space key but have different space types. Each is the other's syzygy space. Pronounced "SIZ-ih-jee". See Chapter 3.

TID Timestamp identifier, a 13-character identifier built from the time, which sorts in time order.

type dimension The direction in a hyperspace that keeps the space key and changes the space type, which is where syzygy lives. See Chapter 3.

writer set The accounts that have written at least one record into a space. See Chapter 7.

XRPC atproto's HTTP API convention. Each method is a lexicon, called at `/xrpc/{nsid}`.

H.1 Token fields

The tokens in this book are JWTs, and their fields have short names:

`iss` issuer, who made the token.

`sub` subject, who the token is about.

`aud` audience, the service the token is meant for. Any other service must reject it.

`exp` when the token expires.

`iat` when the token was issued.

`jti` a unique ID for the token, so a server can refuse to accept it twice.

`cnf` confirmation, the key the token is bound to.

`kid` key ID, in the token's header, naming the key that signed it.

`lxm` in service auth, the XRPC method the token may call.

`htm` and `htu` in a DPoP proof, the HTTP method and URL of the request.

`ath` in a DPoP proof, a hash of the access token it goes with.

`nonce` in a DPoP proof, a value the server handed out, which keeps old proofs from being reused.

Appendix I · AI use and disclosure

This book is an idea I came up with while working in atproto spaces in 2026. The outline and the majority of the content came from my own writing, transcription, and development notes.

I did use AI, and tooling created with AI, for parts of the book, in several ways.

The transcription process corrected and bridged what I was saying, and what I meant, into book material. During that time, some corrections were made and used. This involved Apple Voice Memos, Whisper Transcription, and Claude Desktop.

I used Claude to create strong reference maps of the content and material, for the purpose of correcting, proving, and validating my statements and claims while writing.

I used Claude to call out claims according to the specification material, Bluesky Social's reference implementation, and my own atproto-crates implementation. These call-outs were blended into the book as the content annotations [spec] and [crates], and as mentions of the reference PDS by name.

I used Claude to generate, reference, and validate XRPC methods and lexicon schema references as stubs for some sections. For example, when writing about sync in Chapter 7, in the section on handling an update, I used Claude to fill in the XRPC methods used in that process and to list their method descriptions and the associated descriptive content from the specification. I used that filled-in data to further my own explanation and framing of the process.

I created tools with Claude to produce the data and cryptographic material for the family updates space example used throughout the book. This included generating the identity documents, key material, reference requests, reference responses, and cryptographic signatures.

I used Claude to review the prose against my writing guide and to flag passages for me.

I used Claude to make wording and consistency edits that I approved, such as the hyperspace terms.

I used Claude to apply mechanical changes across the book, such as the `at:space/` URI format and the regenerated example values.

Sources

- The atproto spaces specification and its discussion thread
- Lexicons in `bluesky-social/atproto`: space, memberspace, managedspace, personalspace, label, repo, and sync
- Permissioned data diary, including Reintroducing Spaces
- at:// pizza thoughts, by Bryan Newbold
- The atproto OAuth specification and permissions specification
- ATProtocol OAuth Masterclass, Fall 2025
- atproto-crates and its commit history
- LtHash paper, RFC 9180, RFC 9421, RFC 7800, RFC 9651, DASL, and STAR