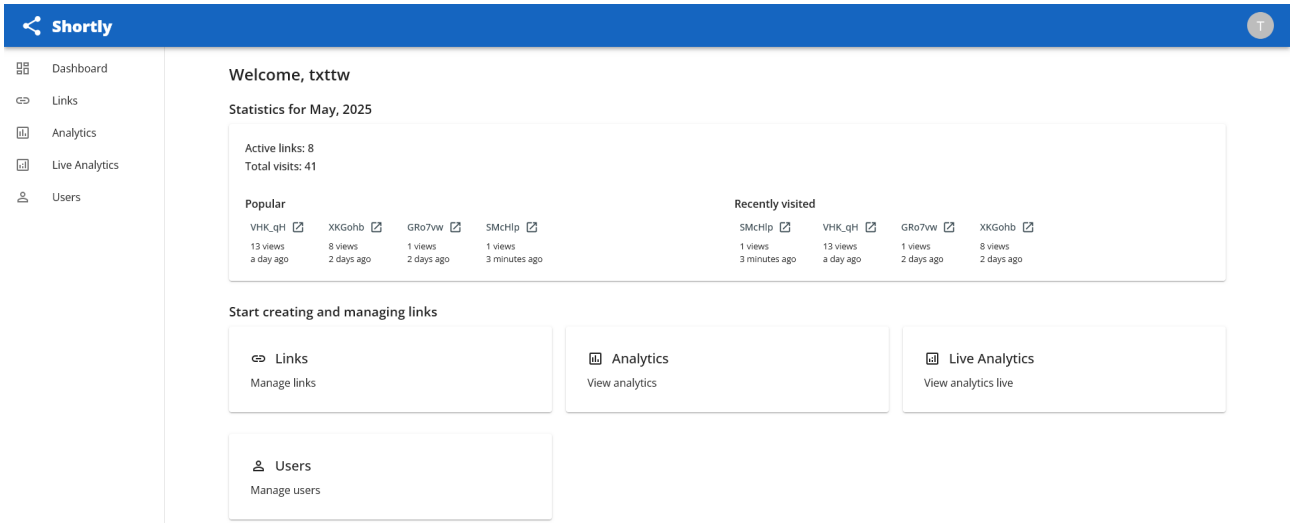
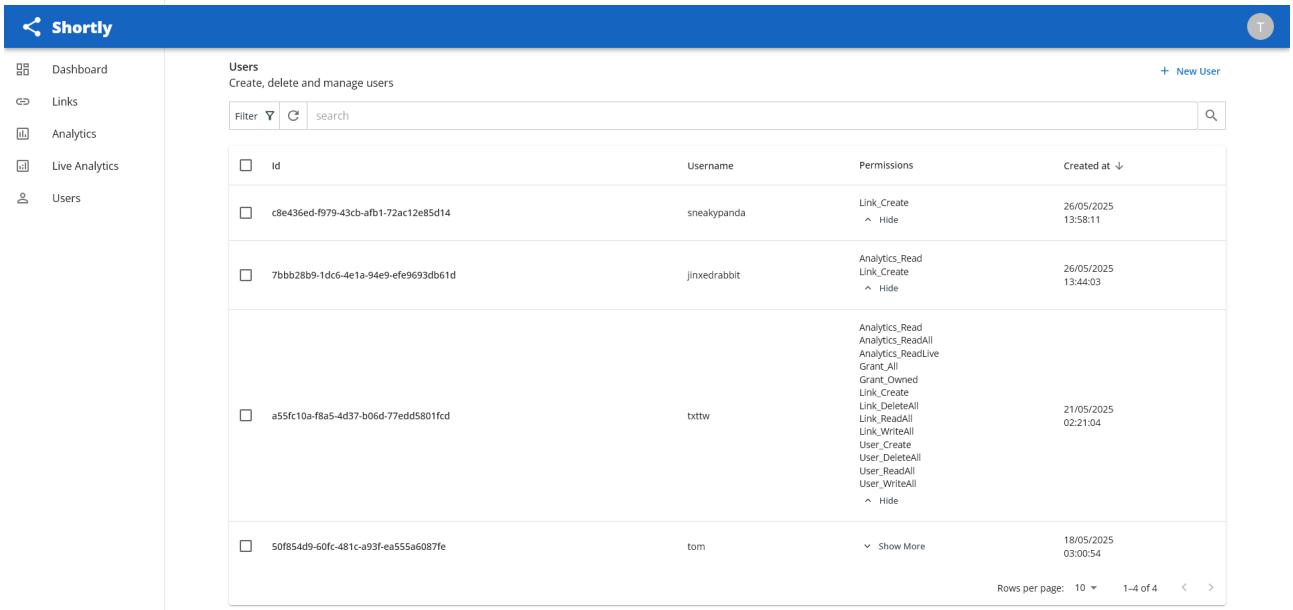
Shortly

- Dynamic short link redirect
- QR code support
- Live analytics
- Statistics
- Management dashboard with filter, sort and search options

Dashboard with statistics

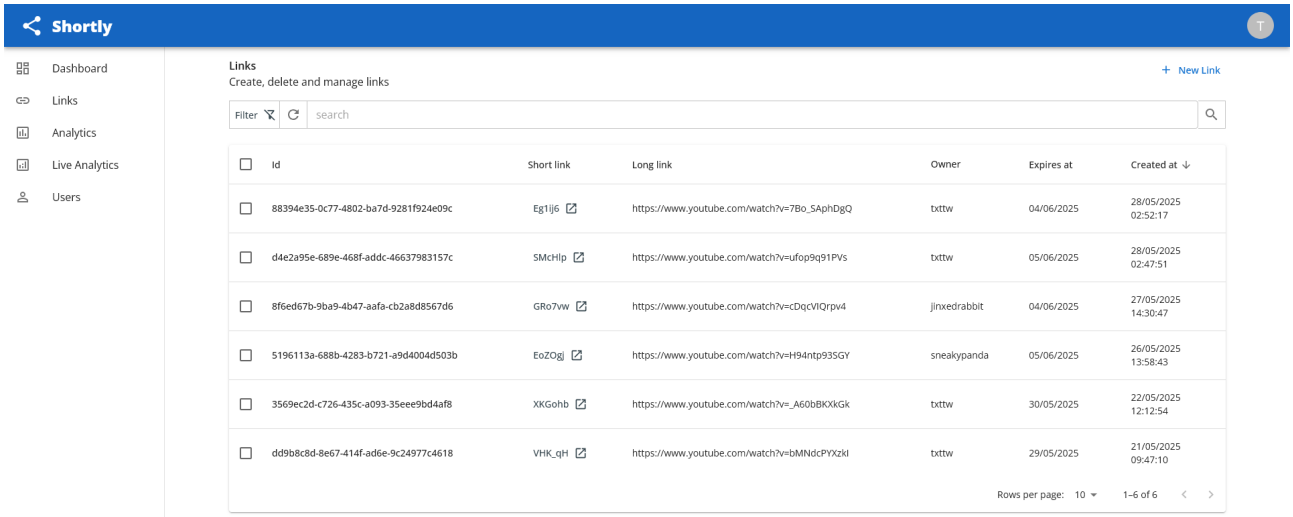


With granular permissions. Filter, search, pagination, multiple selection.

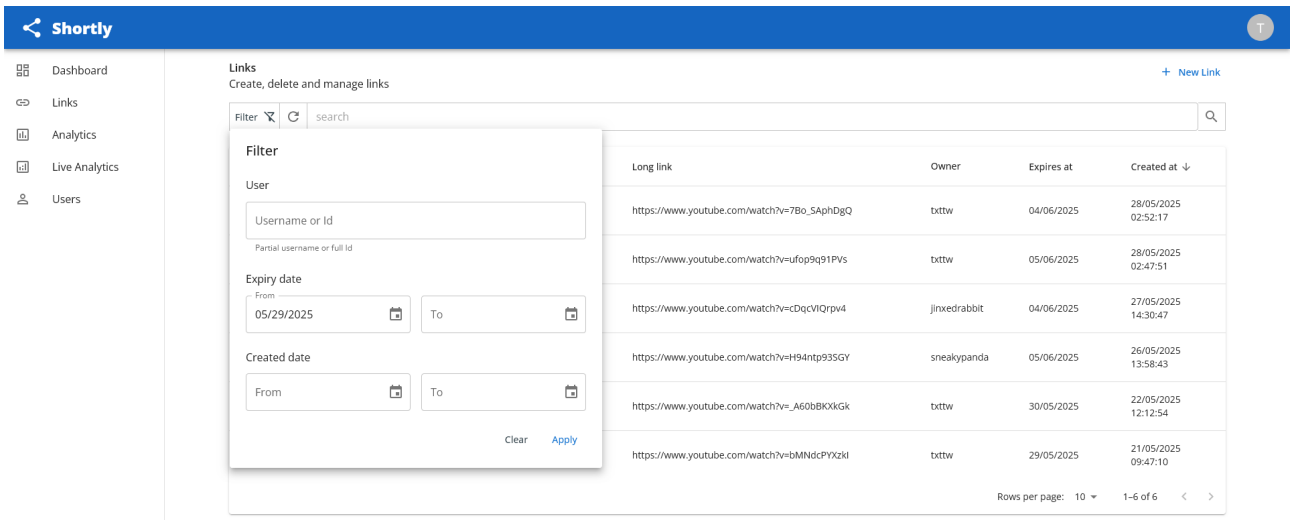


Link overview

With date range filter, search, pagination, multiple selection

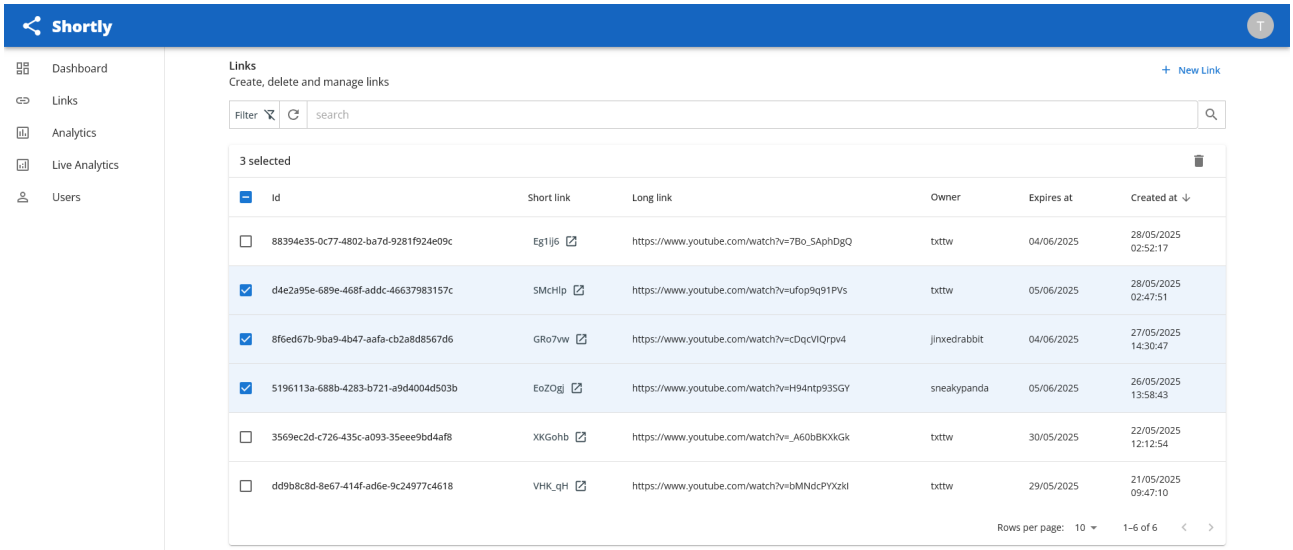


User filter only for Admins

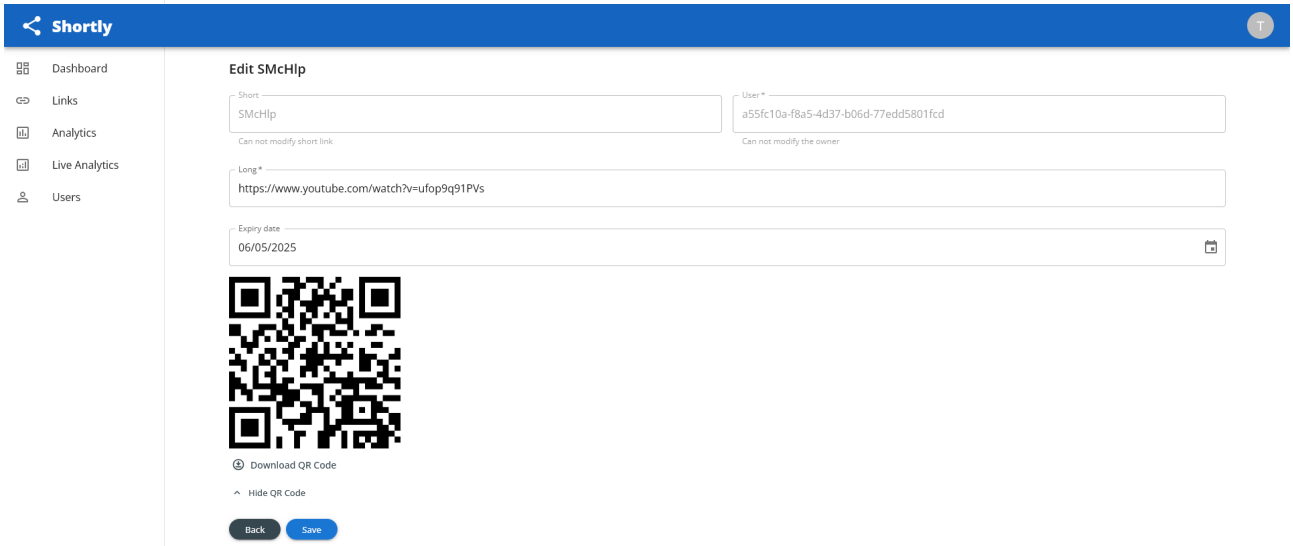


Link management

Create, edit, delete

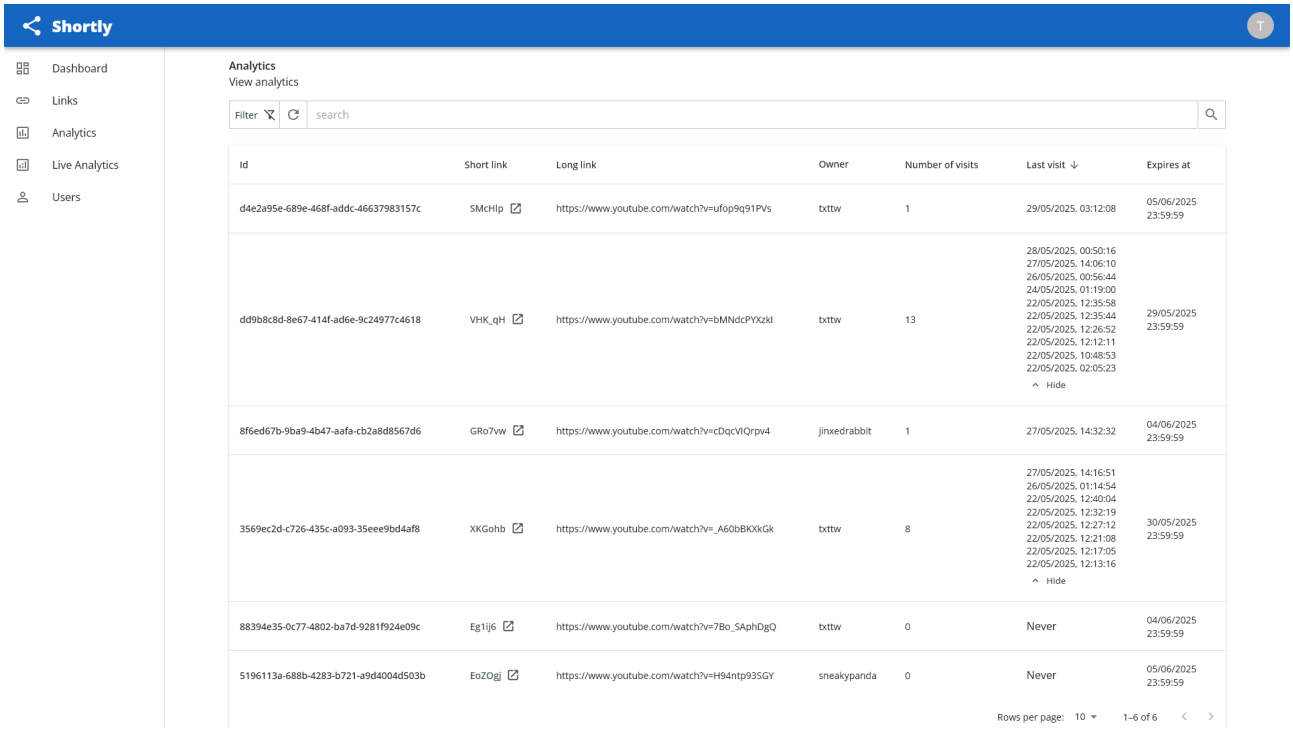


QR code support



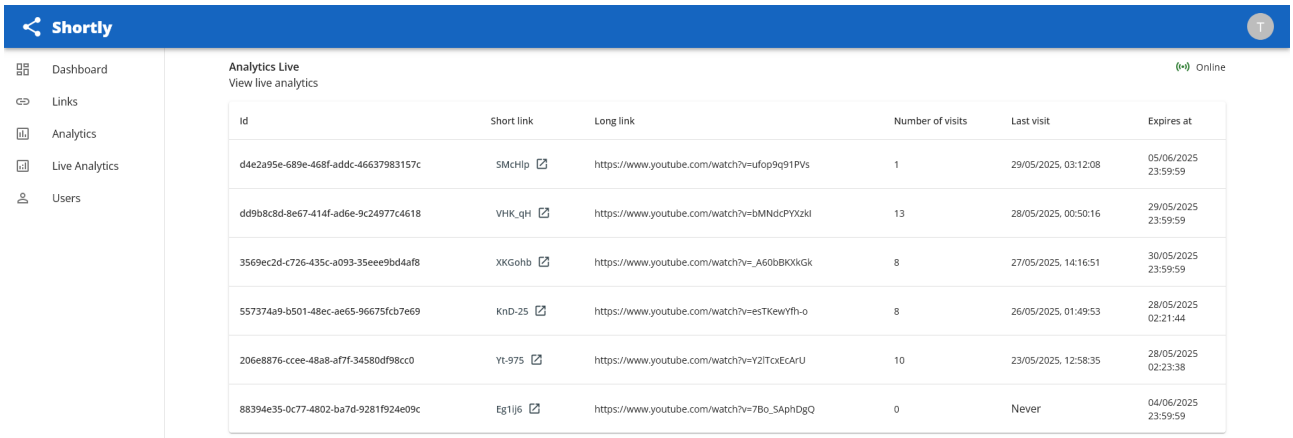
Analytics

Number of visits and last 10 visits date and time



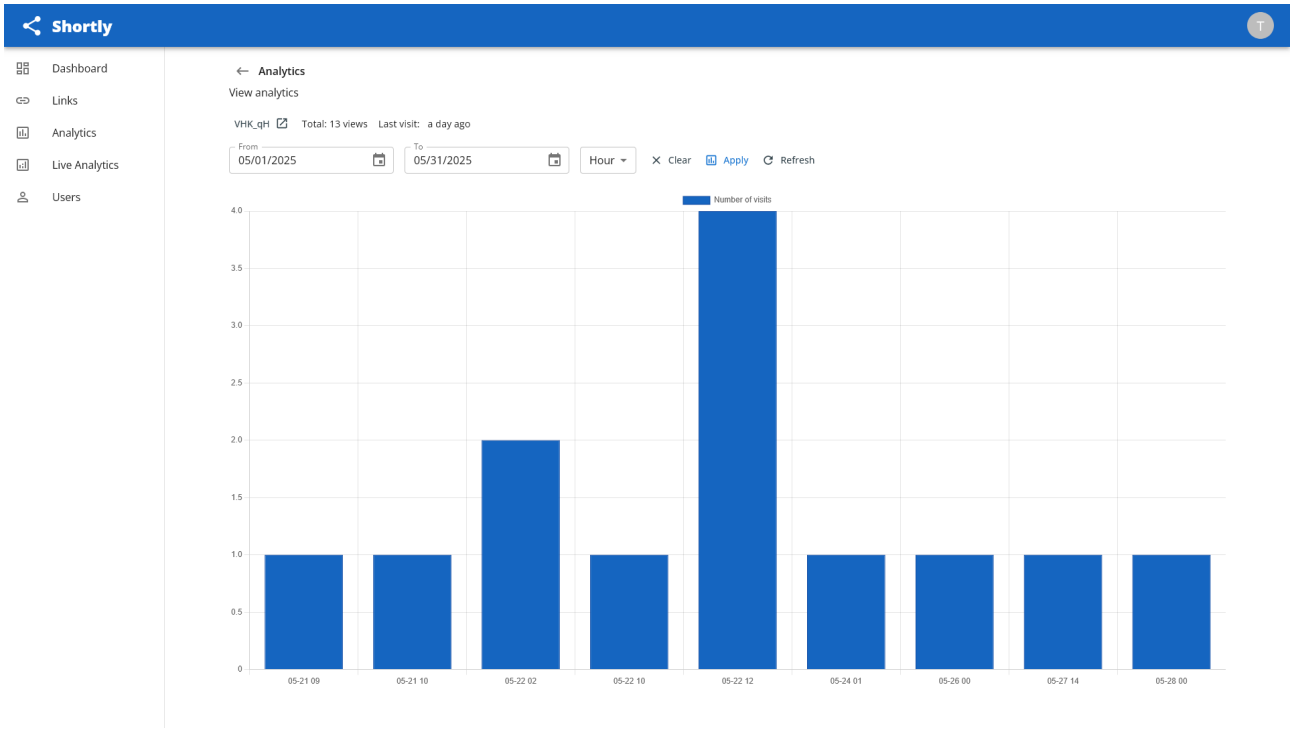
Live analytics

Account (user) based, updates status live for all links associated with a user



Configurable link statistics

Configurable date range and statistics resolution (e.g. monthly, daily, hourly)



How to try?

https://app.shortly.txttw.online

User accounts and permissions:

ask for a test account from the contact person who sent you the link to this document.

Recommended to try with most permissions to get a general overview of the application.

To try with less permissions sign in with an admin user and check other users permissions or create a new user or modify an existing one.

Notes:
The application is serverless, infinitely scalable including the database.
The execution environment keeps services initialized in memory only if there is continuous load, therefore rare invocations can be way slower. In addition, some system components use free plans resulting in lower performance.

The application has a distributed architecture. Data synchronization (including live analytics data) between services can take a few seconds to optimize performance by batching events.

System design

Architecture

Distributed Microservice architecture

Services

Services implemented as serverless functions (Cloudflare workers).

High-performance short link lookup service

High performance operation (short link lookup) is implemented as a simple function capable of running on edge and autoscale to provide maximum performance and low latency. It uses a simple key-value persistent store with built in in-memory cache. Analytics data is not stored in this service but enqueued to an auto-scalabe queue for further processing.

Resource management services (auth, users, links, analytics)

Resource management (users, links, analytics data) related services use a light weight framework Hono designed for serverless functions. As these services do not need complex routing they use the Linear router (fastest from hono/quick) to minimize cpu time and cost.

Live analytics (using WebSocket)

Implemented using Cloudflare's Durable Object ("hibernatable") and WebSocket.

During inactivity, the Durable Object can be evicted from memory, but the WebSocket connection will remain open. If at some later point the WebSocket receives a message, the runtime will recreate the Durable Object and deliver the message to the appropriate handler.

Hibernation can save cost as objects not in memory are not billed.

Authentication service

There is a dedicated auth service handling sign in, logout, token refresh and APIKey management and APIKey exchange for auth token. It is discussed later in this document.

Database

Each microservice has its own database and they only communicate via the MQ (Cloudflare Queue) with async events.

Application uses Prisma ORM, having a light weight generated (based on schema) client fit for serverless functions. It supports a variety of databases like Postgres, MySQL, Mongo and more including cloud native DBs from AWS. The ORM has a CLI that generates and executes migrations from schema changes and generates types for typescript use.

Note: Cloudflare workers currently are not compatible with MongoDB and Cloudflare D5 (Cloudflare's relational DB) does not support interactive transactions, so the application uses a serverless Postgres DB from Prisma Data Platform. It can be replaced by any Postgres compatible database including cloude native solutions without too much code change .

Data integrity considerations:

Resource (e.g. user, link, etc.) is saved in DB in transaction with events including the resource as event data. These stored events also serve as a complete incremental history of the associated resource. It can be used to fix data integrity errors (by ‘replaying’ them from the corrupted version or from scratch) in other services.

Because the events are stored, they allows multiple enqueue attempts if the MQ is down. Each event has a sentAt column with default NULL to identify the unsent messages. (current implementation uses a serverless queue with high availability so periodical triggers to check potential MQ failure and unsent events is not implemented. In a very critical system it could be easily)

When the event is enqueued the MQ tries to deliver based on configuration (currently, it uses small delay and batching, then 3x retry possibility with some retry delay). If delivery fails repeatedly the message enqueued to the resource associated DLQ. It delivers the event back to the resource owner that updates the failedAt column in the events table. There is no automatic process to resolve the multiple errors leading to DLQ as it highly likely indicates a miss-configuration or a development error in data synchronization.

Data versioning

Resource owners (services) increment the v (version) of the document in each update including delete (it is a soft delete). Resource updated events contain the updated v. Event consumers only store updated resource if its version is exactly +1 their own version stored in their own DB, and only in this case acknowledge the event. Not acknowledged events will be retried later. Versioning also handles when the MQ delivers an event multiple times. Soft delete is required to avoid data corruption in some edge cases.

If hard delete is needed the soft deleted records can be pruned after all related synchronization events are completed e.g. after the maximum message retention time of the queue.

Lookups in analytics service can be hard deleted because they are not synchronised with other services and the service stores an aggregated count and last visit timestamp so keeping old timestamps does not add enough business value to pay for DB volume.

Currently there is no automatic process (lack of business requirements when to delete) to prune expired links and corresponding lookup timestamps. A scheduled function could handle easily.

Referential delete and synchronization

Resource owners can decide if they restrict a delete in case of existing references (e.g. user delete restricted if there are active links), but it is optimistic concurrency handling so services with synchronized data perform cascade delete on application level (soft delete in DB).

For example a link created for user A (by Links service) and user A was deleted (by Users service) in the synchronisation window (event delivery can take several seconds, worst case much more so there is a time window when the data is not in sync between services). Restrict (in Users service) couldn’t work because the new link had not yet been synchronized to users service. Therefore, when the Links service receives the user deleted event it will cascade delete the associated link so it won’t be orphaned. This behaviour as many other related to optimistic concurrency has to be (and was) analysed from business logic perspective, its not a general system design consideration.

Authentication, authorization

Application supports 2 types of authentication:

1. Sign in, username + password (for UI users)
2. ApiKey (for programmatical actions, e.g. by a script or other system)

Sign in generates a short lived (10 mins) auth token (JWT with payload containing userId and permissions) for authentication and a long lived (24h) refresh token (JWT) to refresh the short lived token. Then, auth token is sent with the request in the Authorization header as Bearer token.

The authentication token is stateless (can not be revoked) but the front end (UI) does not store it in any persistent (e.g. browser local or session storage, cookie) storage (only in react state). If it expires or the SPA reloads (therefore loosing the state) the UI app will request another auth token using the refresh token.

The refresh token is stateful (reference stored in database) and can be revoked by logout. On the front end it is stored in the browser's local storage. Local storage is vulnerable to XSS attacks so the application also requires a valid fingerprint stored in a HttpOnly, Secure Cookie. Cookies are vulnerable to CSRF attacks but using both at the same time is sufficient enough. Fingerprint is stored as raw text in the cookie and a signed hash in the refresh token. Guessing one from the other is not possible.

ApiKey exchanged for a longer lived (1h) auth token (JWT) otherwise identical to the above mentioned one. It is then used as a Bearer token in Authorization header.

As the authentication token is stateless, in this application (business domain/logic related decision not a general system design consideration) there is no need for centralised authentication. Each service can verify the token and extract the stored information (userId, permissions).

The routing for the application is not overly complex so Cloudflare routes replace the former API gateway service.

Services use a middleware to authenticate, and other middleware(s) or application logic to authorize certain actions based on user id and permissions.

Instead of user roles, the application uses permissions for more granular control.

Validation

Route parameters, body content and query string is validated by the Zod schema validation library.

To handle complex query strings for filtering, the application uses qs-esm npm package to convert objects to query string and string back to object.

Note: For more complex resource types and relational queries GraphQL could be a better option but in this case it is not required.

Service access

Auth service:

https://api.shortly.txttw.online/auth

Users service:
https://api.shortly.txttw.online/users

Links service:
https://api.shortly.txttw.online/links

Analytics service:
https://api.shortly.txttw.online/analytics

Live analytics:
wss://live.shortly.txttw.online/websocket?token=<AUTH_TOKEN>

Lookup service:
https://shortly.txttw.online/:short
where :short is a route param representing the short part

API schema can be viewed as openapi schema on the following endpoints

Auth service:
https://api.shortly.txttw.online/auth/schema/openapi

Users service:
https://api.shortly.txttw.online/users/schema/openapi

Links service:
https://api.shortly.txttw.online/links/schema/openapi

Analytics service:

https://api.shortly.txttw.online/analytics/schema/openapi

Source

In a large project services live independently. They can use different technologies and can be developed by different teams therefore version controlling them independently is the best choice.

But, in this small project it wouldn't add any value so services share one repository.

Note: Both repositories are public, but the project currently lacks documentation and deployment instructions. However, someone familiar with Cloudflare workers and Prisma ORM could deploy easily without instructions.

Backend

https://github.com/txttw/shortly.git

Admin UI

https://github.com/txttw/shortly-app.git