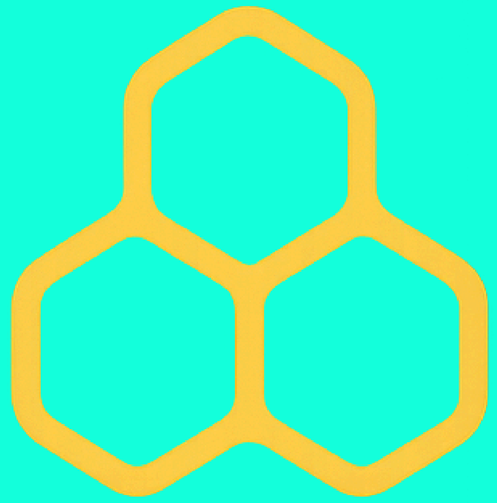


Pixstore

VANILLA TYPESCRIPT FULL STACK
IMAGE STORAGE & CACHING LIBRARY

What is Pixstore?

Pixstore is a modern, high-performance, and fully secure image storage and caching library for both Node.js backends and browser frontends.

Enables end-to-end encrypted, reliable, and scalable image serving for any web app, from simple JavaScript projects to full-stack production systems.

Written in vanilla TypeScript with zero external frameworks or heavy dependencies.

Pixstore Features

- Automatic browser caching
- End-to-end encryption
- Minimal wire protocol
- Stateless secure endpoints
- High performance backend
- Production ready
- Fully extensible
- Minimal dependencies
- TypeScript native
- Universal API

Why Pixstore?

Unified image solution

One library for both backend image storage and frontend caching.

Serious security

Images are always encrypted at rest and in transit; only the intended client can decrypt.

True stateless access

No sessions, cookies, or opaque tokens. All access is cryptographically verifiable.

Flexible integration

Use with any framework or tech stack. Example projects provided.

Minimal friction

Start serving and caching secure images in minutes.

Actively maintained

Frequent updates and new features based on real-world needs.

Getting Started

Pixstore is published on npm
and can be installed easily

```
npm install pixstore
```

Then, just import from the
entrypoints based on your needs

```
import { //export-name } from 'pixstore/backend'
import { //export-name } from 'pixstore/frontend'
import { //export-name } from 'pixstore/shared'
import type { //export-name } from 'pixstore/types'
```

Backend Initialization

You must call **initPixstoreBackend()** once before using any Pixstore backend features.

It should be the very first Pixstore call in your app (before saving or fetching any images).

```
import { initPixstoreBackend } from 'pixstore/backend'

// Minimal setup (uses default config)
initPixstoreBackend()

// Or with custom options
initPixstoreBackend({
  imageRootDir: 'my-images',
  databasePath: './data/pixstore.sqlite',
})
```

Frontend Initialization

Also, you must call **initPixstoreFrontend()** once before using any Pixstore functions on the frontend.

It should be the very first Pixstore call in your browser app, before attempting to cache or fetch images.

```
import { initPixstoreFrontend } from 'pixstore/frontend'

// Minimal setup (uses default config)
initPixstoreFrontend()

// Or with custom options
initPixstoreFrontend({
  frontendImageCacheLimit: 200,
  frontendDbName: 'custom-pixstore',
  defaultEndpointConnectHost: 'https://your-api-host',
})
```

Image CRUD on the Backend

Pixstore automatically creates and manages a **SQLite** database for image metadata, and stores all encrypted image files in your configured directory. You do not need to set up any storage or database manually.

When you save or update an image

- The image is encrypted using **AES-GCM** with a unique key for each image.
- All encryption details (key, IV, authentication tag, and the image token) are stored in Pixstore's internal SQLite table.
- The encrypted image file is saved in your configured image directory.

Image CRUD on the Frontend

Pixstore automatically creates and manages an **IndexedDB** table for image caching in the browser. You do not need to set up or initialize any storage manually.

Pixstore handles all frontend image cache management

- When you use **getImage,** Pixstore fetches and decrypts the image from the backend (if needed), then caches it in IndexedDB as decrypted **(without the key)** for fast future access.
- No encryption keys or sensitive data are ever written to IndexedDB. All decryption is performed **only in memory.**
- Cached images are validated with tokens to ensure you never use stale or out-of-date content.

Backend ↔ Frontend Integration

Pixstore securely connects backend and frontend for encrypted image storage and safe, up-to-date image access.

You should **never send static image URLs** from your backend to your frontend. Instead, always provide an up-to-date **imageRecord** object in your API responses.

On the frontend, always use **getImage(imageRecord)** to fetch, decrypt, and cache the image. Do not use hardcoded image paths or try to fetch images directly by URL.

This pattern ensures your app always displays the latest, authorized, and securely decrypted image. You do not have to manage storage, security, or cache validation yourself.

Backend ↔ Frontend Integration

```
import { getImageRecord } from 'pixstore/
backend'

app.get('/api/player/:id', async (req, res) => {
  const playerId = req.params.id
  const imageRecord =
getImageRecord(playerId)

  // ...fetch player data from DB
  res.json({ ...player, imageRecord })
})
```

Backend Example

```
import { getImage } from 'pixstore/frontend'

const response = await fetch('/api/player/42')

const { imageRecord } = await response.json()

const blob = await getImage(imageRecord)
if (blob) {
  const url = URL.createObjectURL(blob)
}
```

Frontend Example

Stateless Proof Mechanism

Pixstore uses a **stateless proof mechanism** to ensure that image requests are authorized and originate from valid frontend flows, not from direct access to public endpoints.

The stateless proof ensures

- Only your authenticated backend can generate image access metadata.
- Only a fresh, valid request from the frontend can successfully load a decrypted image.
- Without a valid proof, the backend responds with a wire payload in MissingProof or InvalidProof state, and no image data is served.

Default Endpoint & Fetcher

Pixstore provides a default backend endpoint and a **built-in fetcher** for secure and easy image retrieval with minimal configuration.

By default, encrypted images can be served and fetched automatically between backend and frontend. This works as long as you pass a valid **imageRecord** object and configure the backend host on the frontend.

You do not need to write custom routes or fetch logic for standard use cases.
All request validation, including stateless proof generation and validation, happens automatically behind the scenes.

Default Endpoint & Fetcher

How default endpoint and default fetcher works

- The backend exposes an HTTP endpoint for encrypted image requests, typically at /pixstore-image/:id.
- The frontend uses the default fetcher to call this endpoint using a valid **imageRecord.**
- Each request includes a short-lived **statelessProof** that authorizes access to the image.
- The backend validates the proof and responds with encrypted image data in **Pixstore wire format** (Pixstore's internal binary wire format).
- The frontend decodes, decrypts, and caches the image automatically using the metadata provided in the record.

Custom Endpoint & Fetcher

If you need additional protection, such as user-specific access control (for example **JWT tokens** or roles) or if you want to use a different transport (like **WebSocket or gRPC**), you can implement a custom image endpoint and register a custom fetcher on the frontend.

When you use a custom fetcher, Pixstore still enforces stateless proof validation, ensuring that only valid and time-bound image requests are processed. This means you can **combine Pixstore's built-in security with your own logic** for authentication, rate limiting, analytics and more, all while preserving encryption, token-based caching and metadata management.

Pixstore Wire Format

When you create a custom endpoint or a custom fetcher, all image data flows through Pixstore's internal wire format.

This format ensures encrypted data, tokens, and decryption metadata are transmitted **efficiently and securely,** without exposing sensitive internals.

Pixstore uses this layout internally to **encode and decode wire payloads.** You must not return plain JSON or raw image buffers, always use this binary format.

Minimal vs Full Metadata Responses

Pixstore is optimized to send the **smallest possible payload** when the image is already cached and valid.

This mechanism is **fully automated,** but only works correctly if the clientToken is passed to the backend.

Full metadata is only returned if the image was **updated recently,** and the frontend's token is stale.

- If the **token in your imageRecord** matches the **backend token**, Pixstore returns a **minimal payload** containing only the encrypted image.
- If the token is **outdated**, Pixstore returns a **full payload** with updated encryption metadata (meta) and the new token.

Automatic Cache Cleanup

Pixstore **automatically manages** image cache in the browser using IndexedDB. This includes cleanup of old or unused images to stay within your **configured** cache limits.

How automatic cache cleanup works

- Every cached image includes a **lastUsed** timestamp.
- When a **new image is saved,** Pixstore checks how many images exist in the cache.
- If the number exceeds the limit, it **automatically deletes** the least recently used images.
- The cleanup happens in the background and does not block image rendering.

Error Handling Modes

hybrid

Logs expected errors (e.g. image not found), returns null. Throws on unexpected internal errors.

Optionally use **getLastPixstoreError()**.

custom

Forwards all errors to your handler via **setCustomErrorHandler()**.

You must define a handler.

throw

Always throws exceptions.

Use try/catch.

warn

Logs all errors, always returns null.

Useful for debugging silently.

silent

Ignores all errors silently, always returns null.

Use in production fallback flows.

EXAMPLES

Pixstore comes with two fullstack examples to help you learn by doing. Each demonstrates different use cases: default image endpoint vs. custom secure access.

Example 1: Vue + NestJS + Default Endpoint

A minimal GraphQL + Vue 3 app using Pixstore's default image endpoint.

Example 2: React + Express + Custom JWT Endpoint

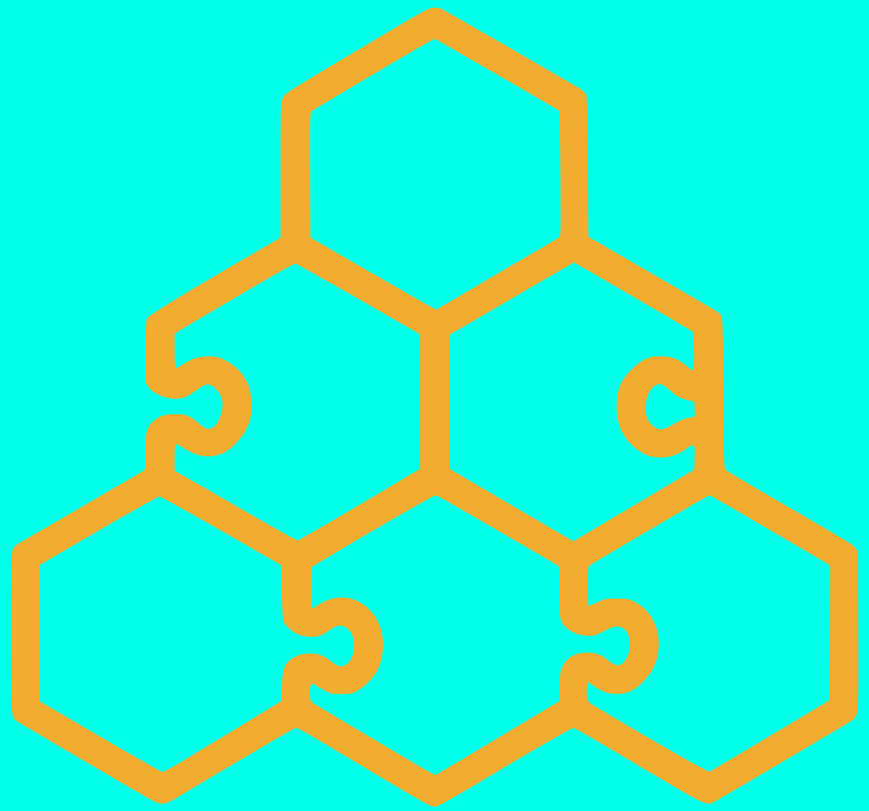
A secure, role-based app using custom image endpoint + fetcher.

Learn – Read the Docs

Pixstore is a secure and high-performance image storage library, designed to be simple yet powerful.
To get the most out of it, understanding how it handles encryption, tokens, and caching will help.

The **official documentation** covers all core concepts and guides you through setup, usage, and advanced features.

Use – Explore the Repository



The Pixstore **repository** is **open source** and production-ready, with full TypeScript code and real-world examples.

You can explore the source, try the examples, check out test coverage, or simply use it as a reliable base for secure image handling in your apps.

Contribute – Get Involved

Pixstore welcomes **contributions** from anyone who wants to improve the project.

Whether it’s fixing bugs, improving documentation, or suggesting features, the contributing guide shows how to get started and submit meaningful changes.

Links

DOCUMENTATION

https://sdenizozturk.github.io/pixstore

REPOSITORY

https://github.com/sDenizOzturk/pixstore

CONTRIBUTING GUIDE

https://sdenizozturk.github.io/pixstore/docs/contributing/

Thank you