readme.md for @push.rocks/lik

⚡ A lean, fully-typed collection of utility classes for TypeScript — efficient data structures, async execution control, and reactive helpers that work seamlessly in both Node.js and the browser.

Install

```
# pnpm (recommended)
pnpm install @push.rocks/lik

# npm
npm install @push.rocks/lik
```

This is a pure ESM package. Use `import` syntax in your TypeScript/JavaScript projects.

Issue Reporting and Security

For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull Requests directly.

Usage

`@push.rocks/lik` gives you **10 focused utility classes** — each solving one problem well. No bloat, no deep dependency trees, just clean tools for everyday TypeScript.

□□ AsyncExecutionStack

Control async task execution with **exclusive** (sequential, mutex-like) and **non-exclusive** (parallel with concurrency limits) modes. Think of it as a lightweight task scheduler.

```
import { AsyncExecutionStack } from '@push.rocks/lik';

const stack = new AsyncExecutionStack();

// □□Exclusive: only one task runs at a time (mutex-style)
const result = await stack.getExclusiveExecutionSlot(async () => {
  // critical section — no other task runs until this resolves
  return await doSomethingImportant();
}, 5000); // optional timeout in ms

// □□Non-exclusive: tasks run in parallel
const p1 = stack.getNonExclusiveExecutionSlot(async () => fetchUser(1));
const p2 = stack.getNonExclusiveExecutionSlot(async () => fetchUser(2));
const p3 = stack.getNonExclusiveExecutionSlot(async () => fetchUser(3));
await Promise.all([p1, p2, p3]);

// □□□Concurrency control for non-exclusive tasks
stack.setNonExclusiveMaxConcurrency(3); // max 3 in parallel
stack.getNonExclusiveMaxConcurrency();  // 3
stack.getActiveNonExclusiveCount();     // how many are running right now
stack.getPendingNonExclusiveCount();    // how many are waiting for a slot
```

Key behavior: Exclusive and non-exclusive slots are processed in order. When an exclusive slot comes up, it waits for all running non-exclusive tasks to finish, then runs alone. Non-exclusive tasks arriving together are batched and run in parallel (respecting the concurrency limit).

□□ BackpressuredArray

A bounded buffer with **backpressure** — perfect for producer/consumer patterns where you need to throttle the producer when the consumer falls behind. Uses RxJS subjects under the hood.

```
import { BackpressuredArray } from '@push.rocks/lik';

const buffer = new BackpressuredArray<string>(16); // high water mark

// Producer side
```

```
const hasSpace = buffer.push('item1');
if (!hasSpace) {
  await buffer.waitForSpace(); // blocks until consumer drains enough
}

// Consumer side
await buffer.waitForItems();       // blocks until something is available
const item = buffer.shift();       // grab the oldest item

// Introspect
buffer.checkSpaceAvailable();      // true if below high water mark
buffer.checkHasItems();            // true if items exist

// Teardown
buffer.destroy();                  // completes all internal subjects
```

⚡ FastMap

A high-performance string-keyed map. Faster than native `Map` for simple key-value lookups thanks to plain-object backing. Supports merging, concatenation, and async search.

```
import { FastMap } from '@push.rocks/lik';

const map = new FastMap<{ name: string; score: number }>();

// CRUD
map.addToMap('player1', { name: 'Alice', score: 100 });
map.addToMap('player2', { name: 'Bob', score: 85 });
map.getByKey('player1');           // { name: 'Alice', score: 100 }
map.isUniqueKey('player1');        // false (already exists)
map.removeFromMap('player2');      // returns the removed object
map.getKeys();                     // ['player1']

// Force overwrite existing key
map.addToMap('player1', { name: 'Alice', score: 200 }, { force: true });

// Merge two maps
const otherMap = new FastMap<{ name: string; score: number }>();
otherMap.addToMap('player3', { name: 'Carol', score: 90 });
```

```
const merged = map.concat(otherMap); // new FastMap with all entries
map.addAllFromOther(otherMap);       // merge in-place

// Async find
const found = await map.find(async (item) => item.score > 95);

// Reset
map.clean();
```

□□ InterestMap & Interest

A deduplicating interest/subscription tracker. Multiple callers can express interest in the same thing — the `InterestMap` deduplicates them and fulfills all waiters at once. Great for caching layers, resource pooling, or subscription fan-out.

```
import { InterestMap } from '@push.rocks/lik';

const interestMap = new InterestMap<string, Response>(
  (id) => id,                        // comparison function for deduplication
  { markLostAfterDefault: 30000 }    // auto-destroy unfulfilled interests after 30s
);

// Express interest (returns existing Interest if one matches)
const interest = await interestMap.addInterest('user:42');

// The interest is a promise-like object
interest.interestFullfilled.then((response) => {
  console.log('Got it!', response);
});

// Somewhere else — fulfill the interest for everyone waiting
const found = interestMap.findInterest('user:42');
found.fullfillInterest(apiResponse);

// Provide a default fulfillment value (returned if interest is destroyed unfulfilled)
const interest2 = await interestMap.addInterest('user:99', fallbackResponse);

// Check / manage
interestMap.checkInterest('user:42');        // true if active
```

```
interestMap.informLostInterest('user:42');   // starts the destruction timer

// Observable stream of all new interests as they arrive
interestMap.interestObservable.subscribe((interest) => {
  console.log('New interest:', interest.comparisonString);
});

// Clean up everything
interestMap.destroy();
```

Interest lifecycle: Created → (optionally) `markLost()` starts a 10s destruction timer → `renew()` resets the timer → `fullfillInterest(value)` resolves all waiters → `destroy()` cleans up.

LimitedArray

A fixed-capacity array that automatically discards the oldest items when the limit is exceeded. Ideal for rolling logs, recent-history buffers, or sliding-window metrics.

```
import { LimitedArray } from '@push.rocks/lik';

const recentLogs = new LimitedArray<string>(100);

recentLogs.addOne('request received');
recentLogs.addMany(['processed', 'response sent']);

console.log(recentLogs.array.length); // never exceeds 100

// Dynamically adjust the limit
recentLogs.setLimit(50); // trims immediately if over

// Built-in average for numeric arrays
const latencies = new LimitedArray<number>(1000);
latencies.addMany([12, 15, 9, 22, 18]);
console.log(latencies.getAverage()); // 15.2
```

LoopTracker

Detects infinite loops by tracking which objects have already been visited during a traversal. Lightweight guard for recursive algorithms.

```
import { LoopTracker } from '@push.rocks/lik';

const tracker = new LoopTracker<object>();

function traverse(node: any) {
  if (!tracker.checkAndTrack(node)) {
    console.warn('Cycle detected — skipping');
    return;
  }
  // safe to process this node
  for (const child of node.children) {
    traverse(child);
  }
}

tracker.reset();   // clear for reuse
tracker.destroy(); // free resources
```

□□ ObjectMap

A managed, observable object collection. Add, remove, find, and iterate objects — with **RxJS event notifications** for every mutation. Supports both auto-keyed and manually-keyed entries.

```
import { ObjectMap } from '@push.rocks/lik';

interface IUser { id: number; name: string; }

const users = new ObjectMap<IUser>();

// Auto-keyed add (returns generated key)
const key = users.add({ id: 1, name: 'Alice' });
users.addArray([{ id: 2, name: 'Bob' }, { id: 3, name: 'Carol' }]);

// Manually-keyed add/get/remove
users.addMappedUnique('admin', { id: 99, name: 'Admin' });
users.getMappedUnique('admin');
```

```
users.removeMappedUnique('admin');

// Find (sync and async)
const alice = users.findSync((u) => u.id === 1);
const bob = await users.find(async (u) => u.id === 2);

// Find and remove in one step
const removed = await users.findOneAndRemove(async (u) => u.id === 3);
const removedSync = users.findOneAndRemoveSync((u) => u.id === 2);

// FIFO-style pop
const first = users.getOneAndRemove();

// Iterate and inspect
await users.forEach((u) => console.log(u.name));
users.checkForObject(alice);  // true/false
users.getKeyForObject(alice); // the internal key string
users.isEmpty();              // true/false
users.getArray();             // cloned array of all objects

// □□Observe mutations
users.eventSubject.subscribe((event) => {
  // event.operation: 'add' | 'remove'
  // event.payload: the object
});

// Merge
const merged = users.concat(otherObjectMap); // new ObjectMap
users.addAllFromOther(otherObjectMap);        // merge in-place

// Teardown
users.wipe();    // remove all entries (fires 'remove' events)
users.destroy(); // wipe + complete eventSubject
```

□□ Stringmap

A string collection with add/remove/query operations and **glob pattern matching** (via minimatch). Supports reactive triggers that fire when a condition becomes true.

```
import { Stringmap } from '@push.rocks/lik';

const tags = new Stringmap();

tags.addString('feature:dark-mode');
tags.addStringArray(['feature:i18n', 'bug:login']);

tags.checkString('feature:dark-mode');   // true
tags.checkMinimatch('feature:*');        // true (glob matching!)
tags.checkIsEmpty();                     // false

tags.removeString('bug:login');
tags.getStringArray();                   // cloned array of current strings

// □□Trigger: resolves when condition is met
const waitForEmpty = tags.registerUntilTrue((arr) => arr.length === 0);
tags.wipe(); // triggers the above → waitForEmpty resolves

// Clean up
tags.destroy();
```

□ TimedAggregator

Batches incoming items over a configurable time window, then processes the entire batch at once. Perfect for log aggregation, metric flushing, or debounced event processing.

```
import { TimedAggregator } from '@push.rocks/lik';
// Also available as: import { TimedAggregtor } from '@push.rocks/lik'; (legacy spelling)

const batcher = new TimedAggregator<string>({
  aggregationIntervalInMillis: 5000,
  functionForAggregation: (batch) => {
    console.log(`Processing ${batch.length} items:`, batch);
  },
});

batcher.add('event-a');
batcher.add('event-b');
```

```
batcher.add('event-c');
// After 5 seconds → "Processing 3 items: ['event-a', 'event-b', 'event-c']"

// Stop and flush any remaining items
batcher.stop(true);   // true = flush remaining immediately

// Restart after stopping
batcher.restart();
batcher.add('event-d'); // timer starts again
```

How it works: The timer starts when the first item arrives. When it fires, all accumulated items are passed to `functionForAggregation` and a new timer starts. If no items arrive, the timer doesn't restart — it's lazy.

□□ Tree

A typed wrapper around `symbol-tree` for managing hierarchical data. Full parent/child/sibling navigation, ordered iteration, and structural mutation.

```
import { Tree } from '@push.rocks/lik';

interface INode { name: string; }

const tree = new Tree<INode>();
const root: INode = { name: 'root' };
tree.initialize(root);

const child1: INode = { name: 'child1' };
const child2: INode = { name: 'child2' };
const grandchild: INode = { name: 'grandchild' };

tree.appendChild(root, child1);
tree.appendChild(root, child2);
tree.appendChild(child1, grandchild);

// □□Navigate
tree.parent(child1);           // root
tree.firstChild(root);         // child1
tree.lastChild(root);          // child2
```

```
tree.nextSibling(child1);     // child2
tree.previousSibling(child2); // child1
tree.hasChildren(child1);     // true

// □□Query
tree.childrenCount(root);               // 2
tree.index(child2);                     // 1
tree.childrenToArray(root, {});         // [child1, child2]
tree.ancestorsToArray(grandchild, {});  // [child1, root]
tree.treeToArray(root, {});             // entire tree as flat array

// □□Iterate
for (const node of tree.treeIterator(root, {})) {
  console.log(node.name);
}

// ✂□ Mutate
tree.insertBefore(child2, { name: 'between' });
tree.insertAfter(child1, { name: 'after1' });
tree.prependChild(root, { name: 'new-first' });
tree.remove(child2);
```

API at a Glance

Class	Purpose	Key Feature
`AsyncExecutionStack`	Async task scheduling	Exclusive/non-exclusive modes with concurrency limits
`BackpressuredArray`	Bounded buffer	Producer/consumer backpressure via RxJS
`FastMap`	Key-value store	O(1) lookups, merge/concat support
`InterestMap`	Deduplicated subscriptions	Multiple waiters, single fulfillment
`LimitedArray`	Rolling buffer	Auto-trim + built-in average
`LoopTracker`	Cycle detection	Track-and-check in one call
`ObjectMap`	Observable collection	RxJS event stream on every mutation
`Stringmap`	String set	Glob matching + reactive triggers

Class	Purpose	Key Feature
`TimedAggregator`	Batch processor	Time-windowed aggregation with flush
`Tree`	Hierarchical data	Full navigation, iteration, mutation

License and Legal Information

This repository contains open-source code licensed under the MIT License. A copy of the license can be found in the LICENSE file.

Please note: The MIT License does not grant permission to use the trade names, trademarks, service marks, or product names of the project, except as required for reasonable and customary use in describing the origin of the work and reproducing the content of the NOTICE file.

Trademarks

This project is owned and maintained by Task Venture Capital GmbH. The names and logos associated with Task Venture Capital GmbH and any related products or services are trademarks of Task Venture Capital GmbH or third parties, and are not included within the scope of the MIT license granted herein.

Use of these trademarks must comply with Task Venture Capital GmbH's Trademark Guidelines or the guidelines of the respective third-party owners, and any usage must be approved in writing. Third-party trademarks used herein are the property of their respective owners and used only in a descriptive manner, e.g. for an implementation of an API or similar.

Company Information

Task Venture Capital GmbH Registered at District Court Bremen HRB 35230 HB, Germany

For any legal inquiries or further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

Revision #3
Created 2026-03-28 11:09:54 UTC by foss.global Team
Updated 2026-03-28 12:16:30 UTC by foss.global Team