readme.md for @push.rocks/smartstate

A TypeScript-first reactive state management library with processes, middleware, computed state, batching, persistence, and Web Component Context Protocol support □□

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.

Install

```
pnpm install @push.rocks/smartstate --save
```

Or with npm:

```
npm install @push.rocks/smartstate --save
```

Usage

Quick Start

```
import { Smartstate } from '@push.rocks/smartstate';

// 1. Define your state part names
type AppParts = 'user' | 'settings';
```

```
// 2. Create the root instance
const state = new Smartstate<AppParts>();

// 3. Create state parts with initial values
const userState = await state.getStatePart<{ name: string; loggedIn: boolean }>('user', {
  name: '',
  loggedIn: false,
});

// 4. Subscribe to changes
userState.select((s) => s.name).subscribe((name) => {
  console.log('Name changed:', name);
});

// 5. Update state
await userState.setState({ name: 'Alice', loggedIn: true });
```

□□ State Parts & Init Modes

State parts are isolated, typed units of state — the building blocks of your application's state tree. Create them via `getStatePart()`:

```
const part = await state.getStatePart<IMyState>(name, initialState, initMode);
```

Init Mode	Behavior
`'soft'` (default)	Returns existing if found, creates new otherwise
`'mandatory'`	Throws if state part already exists — useful for ensuring single-initialization
`'force'`	Always creates a new state part, disposing and overwriting any existing one
`'persistent'`	Like `'soft'` but automatically persists state to IndexedDB via WebStore

You can use either string literal union types or enums for state part names:

```
// String literal types (simpler)
type AppParts = 'user' | 'settings' | 'cart';
```

```
// Enums (more explicit)
enum AppParts {
  User = 'user',
  Settings = 'settings',
  Cart = 'cart',
}
```

□□ Persistent State

```
const settings = await state.getStatePart('settings', { theme: 'dark', fontSize: 14 },
'persistent');

// □ Automatically saved to IndexedDB on every setState()
// □ On next app load, persisted values override defaults
// □ Persistence writes complete before in-memory updates
```

□□ Selecting State

`select()` returns an RxJS Observable that emits the current value immediately (via `BehaviorSubject`) and on every subsequent change:

```
// Full state
userState.select().subscribe((state) => console.log(state));

// Derived value via selector function
userState.select((s) => s.name).subscribe((name) => console.log(name));
```

Selectors are **memoized** — calling `select(fn)` with the same function reference returns the same cached Observable, shared across all subscribers via `shareReplay`. This means you can call `select(mySelector)` in multiple places without creating duplicate subscriptions.

Change detection is built in: `select()` uses `distinctUntilChanged` with deep JSON comparison, so subscribers only fire when the selected value actually changes. Selecting `s => s.name` won't re-emit when only `s.count` changes.

✂ AbortSignal Support

Clean up subscriptions without manual `.unsubscribe()` — the modern way:

```
const controller = new AbortController();

userState.select((s) => s.name, { signal: controller.signal }).subscribe((name) => {
  console.log(name); // automatically stops receiving when aborted
});

// Later: clean up all subscriptions tied to this signal
controller.abort();
```

⚡ Actions

Actions provide controlled, named state mutations with full async support:

```
interface ILoginPayload {
  username: string;
  email: string;
}

const loginAction = userState.createAction<ILoginPayload>(async (statePart, payload) => {
  const current = statePart.getState();
  return { ...current, name: payload.username, loggedIn: true };
});

// Two equivalent ways to dispatch:
await loginAction.trigger({ username: 'Alice', email: 'alice@example.com' });
// or
await userState.dispatchAction(loginAction, { username: 'Alice', email: 'alice@example.com'
});
```

Both `trigger()` and `dispatchAction()` return a Promise with the new state. All dispatches are serialized through a mutation queue, so concurrent dispatches never cause lost updates.

□□ Nested Actions (Action Context)

When you need to dispatch sub-actions from within an action, use the `context` parameter. This is critical because calling `dispatchAction()` directly from inside an action would deadlock (it tries to acquire the mutation queue that's already held). The context's `dispatch()` bypasses the queue and executes inline:

```
const incrementAction = userState.createAction<number>(async (statePart, amount) => {
  const current = statePart.getState();
  return { ...current, count: current.count + amount };
});

const doubleIncrementAction = userState.createAction<number>(async (statePart, amount,
context) => {
  // ✅ Safe: uses context.dispatch() which bypasses the mutation queue
  await context.dispatch(incrementAction, amount);
  const current = statePart.getState();
  return { ...current, count: current.count + amount };
});

// ❌ DON'T do this inside an action — it will deadlock:
// await statePart.dispatchAction(someAction, payload);
```

A built-in depth limit (10 levels) prevents infinite circular dispatch chains, throwing a clear error if exceeded.

⚙️ Processes (Polling, Streams & Scheduled Tasks)

Processes are managed, pausable observable-to-state bridges — the "side effects" layer. They tie an ongoing data source (polling, WebSockets, event streams) to state updates with full lifecycle control and optional auto-pause.

Basic Process: Polling an API

```
import { interval, switchMap, from } from 'rxjs';

const metricsPoller = dashboard.createProcess<{ cpu: number; memory: number }>({
  // Producer: an Observable factory — called on start and each resume
  producer: () => interval(5000).pipe(
    switchMap(() => from(fetch('/api/metrics').then(r => r.json()))),
  ),
  // Reducer: folds each produced value into state (runs through middleware & validation)
  reducer: (currentState, metrics) => ({
    ...currentState,
    metrics,
```

```
    lastUpdated: Date.now(),
  }),
  autoPause: 'visibility',  // ⏸ Stop polling when the tab is hidden
  autoStart: true,          // ▶ Start immediately
});

// Full lifecycle control
metricsPoller.pause();    // Unsubscribes from producer
metricsPoller.resume();   // Re-subscribes (fresh subscription)
metricsPoller.dispose();  // Permanent cleanup

// Observe status reactively
metricsPoller.status;     // 'idle' | 'running' | 'paused' | 'disposed'
metricsPoller.status$.subscribe(s => console.log('Process:', s));
```

Scheduled Actions

Dispatch an existing action on a recurring interval — syntactic sugar over `createProcess`:

```
const refreshAction = dashboard.createAction<void>(async (sp) => {
  const data = await fetch('/api/dashboard').then(r => r.json());
  return { ...sp.getState()!, ...data, lastUpdated: Date.now() };
});

// Dispatches refreshAction every 30 seconds, auto-pauses when tab is hidden
const scheduled = dashboard.createScheduledAction({
  action: refreshAction,
  payload: undefined,
  intervalMs: 30000,
  autoPause: 'visibility',
});

// It's a full StateProcess — pause, resume, dispose all work
scheduled.dispose();
```

Custom Auto-Pause Signals

Pass any `Observable<boolean>` as the auto-pause signal — `true` means active, `false` means pause:

```
import { fromEvent, map, startWith } from 'rxjs';

// Pause when offline, resume when online
const onlineSignal = fromEvent(window, 'online').pipe(
  startWith(null),
  map(() => navigator.onLine),
);

const syncProcess = userPart.createProcess<SyncPayload>({
  producer: () => interval(10000).pipe(
    switchMap(() => from(syncWithServer())),
  ),
  reducer: (state, result) => ({ ...state, ...result }),
  autoPause: onlineSignal,
});
syncProcess.start();
```

WebSocket / Live Streams

Pause disconnects; resume creates a fresh connection:

```
const liveProcess = tickerPart.createProcess<TradeEvent>({
  producer: () => new Observable<TradeEvent>(subscriber => {
    const ws = new WebSocket('wss://trades.example.com');
    ws.onmessage = (e) => subscriber.next(JSON.parse(e.data));
    ws.onerror = (e) => subscriber.error(e);
    ws.onclose = () => subscriber.complete();
    return () => ws.close();  // Teardown: close WebSocket on unsubscribe
  }),
  reducer: (state, trade) => ({
    ...state,
    lastPrice: trade.price,
    trades: [...state.trades.slice(-99), trade],
  }),
  autoPause: 'visibility',
});
liveProcess.start();
```

Error Recovery

If a producer errors, the process gracefully transitions to `'paused'` instead of dying. Call `resume()` to retry with a fresh subscription:

```
process.start();
// Producer errors → status becomes 'paused'
process.resume(); // Creates a fresh subscription — retry
```

Process Cleanup Cascades

Disposing a `StatePart` or `Smartstate` instance automatically disposes all attached processes:

```
const p1 = part.createProcess({ ... });
const p2 = part.createProcess({ ... });
p1.start();
p2.start();

part.dispose();
console.log(p1.status); // 'disposed'
console.log(p2.status); // 'disposed'
```

□□ Middleware

Intercept every `setState()` call to transform, validate, log, or reject state changes:

```
// Logging middleware
userState.addMiddleware((newState, oldState) => {
  console.log('State changing:', oldState, '→', newState);
  return newState;
});

// Validation middleware — throw to reject the change
userState.addMiddleware((newState) => {
  if (!newState.name) throw new Error('Name is required');
  return newState;
});

// Transform middleware
userState.addMiddleware((newState) => {
  return { ...newState, name: newState.name.trim() };
});
```

```
// Async middleware
userState.addMiddleware(async (newState, oldState) => {
  await auditLog('state-change', { from: oldState, to: newState });
  return newState;
});

// Removal — addMiddleware() returns a dispose function
const remove = userState.addMiddleware(myMiddleware);
remove(); // middleware no longer runs
```

Middleware runs **sequentially** in insertion order. If any middleware throws, the state remains unchanged — the operation is **atomic**. Process-driven state updates go through middleware too.

□□ Computed / Derived State

Derive reactive values from one or more state parts using `combineLatest` under the hood:

```
import { computed } from '@push.rocks/smartstate';

const userState = await state.getStatePart('user', { firstName: 'Jane', lastName: 'Doe' });
const settingsState = await state.getStatePart('settings', { locale: 'en' });

// Standalone function
const greeting$ = computed(
  [userState, settingsState],
  (user, settings) => `Hello, ${user.firstName} (${settings.locale})`,
);

greeting$.subscribe((msg) => console.log(msg));
// => "Hello, Jane (en)"

// Also available as a convenience method on the Smartstate instance:
const greeting2$ = state.computed(
  [userState, settingsState],
  (user, settings) => `${user.firstName} - ${settings.locale}`,
);
```

Computed observables are **lazy** — they only subscribe to their sources when someone subscribes to them, and they automatically unsubscribe when all subscribers disconnect. They also use

`distinctUntilChanged` to avoid redundant emissions when the derived value hasn't actually changed.

□□ Batch Updates

Update multiple state parts at once while deferring all notifications until the entire batch completes:

```
const partA = await state.getStatePart('a', { value: 1 });
const partB = await state.getStatePart('b', { value: 2 });

await state.batch(async () => {
  await partA.setState({ value: 10 });
  await partB.setState({ value: 20 });
  // No notifications fire inside the batch
});
// Both subscribers now fire with their new values simultaneously

// Nested batches are supported — flush happens at the outermost level only
await state.batch(async () => {
  await partA.setState({ value: 100 });
  await state.batch(async () => {
    await partB.setState({ value: 200 });
  });
  // Still deferred — inner batch doesn't trigger flush
});
// Now both fire
```

□ Waiting for State

Wait for a specific state condition to be met before proceeding:

```
// Wait for any truthy state
const currentState = await userState.waitUntilPresent();

// Wait for a specific condition
const name = await userState.waitUntilPresent((s) => s.name || undefined);

// With timeout (milliseconds)
```

```
const name = await userState.waitUntilPresent((s) => s.name || undefined, 5000);

// With AbortSignal and/or timeout via options object
const controller = new AbortController();
try {
  const name = await userState.waitUntilPresent(
    (s) => s.name || undefined,
    { timeoutMs: 5000, signal: controller.signal },
  );
} catch (e) {
  // e.message is 'Aborted' or 'waitUntilPresent timed out after 5000ms'
}
```

□□ Context Protocol Bridge (Web Components)

Expose state parts to web components via the W3C Context Protocol. This lets any web component framework (Lit, FAST, Stencil, or vanilla) consume your state without coupling:

```
import { attachContextProvider } from '@push.rocks/smartstate';

// Define a context key (use Symbol for uniqueness)
const themeContext = Symbol('theme');

// Attach a provider to a DOM element — any descendant can consume it
const cleanup = attachContextProvider(document.body, {
  context: themeContext,
  statePart: settingsState,
  selectorFn: (s) => s.theme, // optional: provide a derived value instead of full state
});

// A consumer dispatches a context-request event:
myComponent.dispatchEvent(
  new CustomEvent('context-request', {
    bubbles: true,
    composed: true,
    detail: {
      context: themeContext,
```

```
      callback: (theme) => console.log('Got theme:', theme),
      subscribe: true, // receive updates whenever the state changes
    },
  }),
);

// Works seamlessly with Lit's @consume() decorator, FAST's context, etc.

// Cleanup when the provider is no longer needed
cleanup();
```

🛡️ State Validation

Built-in validation prevents `null` and `undefined` from being set as state. For custom validation, extend `StatePart`:

```
import { StatePart } from '@push.rocks/smartstate';

class ValidatedUserPart extends StatePart<string, IUserState> {
  protected validateState(stateArg: any): stateArg is IUserState {
    return (
      super.validateState(stateArg) &&
      typeof stateArg.name === 'string' &&
      typeof stateArg.loggedIn === 'boolean'
    );
  }
}
```

If validation fails, `setState()` throws and the state remains unchanged.

⚙️ Async State Setup

Initialize state with async operations while ensuring actions wait for setup to complete:

```
await userState.stateSetup(async (statePart) => {
  const userData = await fetchUserFromAPI();
  return { ...statePart.getState(), ...userData };
});
```

```
// Any dispatchAction() calls will automatically wait for stateSetup() to finish
```

□□ Disposal & Cleanup

Both `Smartstate` and individual `StatePart` instances support disposal for proper cleanup:

```
// Dispose a single state part — completes the BehaviorSubject, clears middleware, caches,
// and disposes all attached processes
userState.dispose();

// Dispose the entire Smartstate instance — disposes all state parts and clears internal maps
state.dispose();
```

After disposal, `setState()` and `dispatchAction()` will throw if called on a disposed `StatePart`. Calling `start()`, `pause()`, or `resume()` on a disposed `StateProcess` also throws.

□□ Performance

Smartstate is built with performance in mind:

- □□ **SHA256 Change Detection** — Uses content hashing to detect actual changes. Identical state values don't trigger notifications, even with different object references.
- □□ **distinctUntilChanged on Selectors** — Sub-selectors only fire when the selected slice actually changes. `select(s => s.name)` won't emit when `s.count` changes.
- ♻ **Selector Memoization** — `select(fn)` caches observables by function reference and shares them via `shareReplay({ refCount: true })`. Multiple subscribers share one upstream subscription.
- □□ **Cumulative Notifications** — `notifyChangeCumulative()` debounces rapid changes into a single notification at the end of the call stack.
- □□ **Concurrent Safety** — Simultaneous `getStatePart()` calls for the same name return the same promise, preventing duplicate creation. All `setState()` and `dispatchAction()` calls are serialized through a mutation queue. Process values are serialized through their own internal queue.
- □□ **Atomic Persistence** — WebStore writes complete before in-memory state updates, ensuring consistency.
- □ **Batch Deferred Notifications** — `batch()` suppresses all subscriber notifications until every update in the batch completes.

API Reference

`Smartstate<T>`

Method / Property	Description
`getStatePart(name, initial?, initMode?)`	Get or create a typed state part
`batch(fn)`	Batch state updates, defer all notifications until complete
`computed(sources, fn)`	Create a computed observable from multiple state parts
`dispose()`	Dispose all state parts and clear internal state
`isBatching`	`boolean` — whether a batch is currently active

`StatePart<TName, TPayload>`

Method	Description
`getState()`	Get current state synchronously (`TPayload | undefined`)
`setState(newState)`	Set state — runs middleware → validates → persists → notifies
`select(selectorFn?, options?)`	Observable of state or derived values. Options: `{ signal?: AbortSignal }`
`createAction(actionDef)`	Create a reusable, typed state action
`dispatchAction(action, payload)`	Dispatch an action and return the new state
`addMiddleware(fn)`	Add a middleware interceptor. Returns a removal function
`waitUntilPresent(selectorFn?, opts?)`	Wait for a state condition. Opts: `number` (timeout) or `{ timeoutMs?, signal? }`
`createProcess(options)`	Create a managed, pausable process tied to this state part
`createScheduledAction(options)`	Create a process that dispatches an action on a recurring interval
`notifyChange()`	Manually trigger a change notification (with hash dedup)
`notifyChangeCumulative()`	Debounced notification — fires at end of call stack
`stateSetup(fn)`	Async state initialization with action serialization
`dispose()`	Complete the BehaviorSubject, dispose processes, clear middleware and caches

`StateAction<TState, TPayload>`

Method	Description
`trigger(payload)`	Dispatch the action on its associated state part

`StateProcess<TName, TPayload, TProducerValue>`

Method / Property	Description
`start()`	Start the process (subscribes to producer, sets up auto-pause)
`pause()`	Pause the process (unsubscribes from producer)
`resume()`	Resume a paused process (fresh subscription to producer)
`dispose()`	Permanently stop the process and clean up
`status`	Current status: `'idle' | 'running' | 'paused' | 'disposed'`
`status$`	Observable of status transitions

`IActionContext<TState>`

Method	Description
`dispatch(action, payload)`	Dispatch a sub-action inline (bypasses mutation queue). Available as the third argument to action definitions

Standalone Functions

Function	Description
`computed(sources, fn)`	Create a computed observable from multiple state parts
`attachContextProvider(element, options)`	Bridge a state part to the W3C Context Protocol

Exported Types

Type	Description
`TInitMode`	`'soft' | 'mandatory' | 'force' | 'persistent'`
`TMiddleware<TPayload>`	`(newState, oldState) => TPayload | Promise<TPayload>`
`IActionDef<TState, TPayload>`	Action definition function signature (receives statePart, payload, context?)
`IActionContext<TState>`	Context for safe nested dispatch within actions
`IContextProviderOptions<TPayload>`	Options for `attachContextProvider`

Type	Description
`IProcessOptions<TPayload, TValue>`	Options for `createProcess` (producer, reducer, autoPause, autoStart)
`IScheduledActionOptions<TPayload, TActionPayload>`	Options for `createScheduledAction` (action, payload, intervalMs, autoPause)
`TProcessStatus`	`'idle' | 'running' | 'paused' | 'disposed'`
`TAutoPause`	`'visibility' | Observable<boolean> | false`

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 11:12:36 UTC by foss.global Team

Updated 2026-03-28 12:19:22 UTC by foss.global Team