@push.rocks/smartcache

A module for caching asynchronous function results with smart time and hash-based invalidation strategies.

- readme.md for @push.rocks/smartcache
- changelog.md for @push.rocks/smartcache

readme.md for @push.rocks/smartcache

Smart time-based caching for async functions - Because waiting is overrated!

□□ What is SmartCache?

`@push.rocks/smartcache` is a powerful TypeScript caching library that intelligently caches the results of asynchronous functions. It automatically detects identical function calls based on the call stack, preventing redundant computations and dramatically improving performance. Perfect for expensive API calls, complex calculations, or any async operation you don't want to repeat unnecessarily!

□ Key Features

- □□ **Automatic cache identification** - Uses smart hashing to detect identical function calls
- □ **Time-based expiration** - Set custom cache durations for each cached operation
- □□ **Concurrent request handling** - Multiple simultaneous requests for the same data only trigger one actual call
- □□ **Zero configuration** - Works out of the box with sensible defaults
- □□ **Lightweight** - Minimal dependencies, maximum performance
- □□ **Type-safe** - Full TypeScript support with proper typing

□□ Installation

Get started in seconds:

```
# Using npm
npm install @push.rocks/smartcache --save


# Using pnpm (recommended)
```

```
pnpm add @push.rocks/smartcache

# Using yarn
yarn add @push.rocks/smartcache
```

□□ Quick Start

```
import { SmartCache } from '@push.rocks/smartcache';

// Create a cache instance
const cache = new SmartCache();

// Your expensive async function
async function fetchUserData(userId: string) {
  console.log(`Fetching user ${userId} from API...`);
  const response = await fetch(`/api/users/${userId}`);
  return response.json();
}

// Wrap it with caching - subsequent calls within 5 seconds return cached result
async function getCachedUserData(userId: string) {
  return cache.cacheReturn(
    () => fetchUserData(userId),
    5000 // Cache for 5 seconds
  );
}

// First call - hits the API
const user1 = await getCachedUserData('123'); // Logs: "Fetching user 123 from API..."

// Second call within 5 seconds - returns cached result instantly!
const user2 = await getCachedUserData('123'); // No log, returns cached data
```

□□ Advanced Usage

Automatic Call Stack Detection

SmartCache uses intelligent call stack analysis to automatically identify unique function calls. This means you don't need to manually specify cache keys!

```
const cache = new SmartCache();

// These will be automatically cached separately based on their call location
async function getData() {
  // First location in code
  const result1 = await cache.cacheReturn(async () => {
    return await fetch('/api/data1').then(r => r.json());
  }, 10000);

  // Second location in code - different cache entry!
  const result2 = await cache.cacheReturn(async () => {
    return await fetch('/api/data2').then(r => r.json());
  }, 10000);

  return { result1, result2 };
}
```

Concurrent Request Handling

SmartCache brilliantly handles concurrent requests - if multiple requests come in for the same cached operation before the first one completes, they all wait for and share the same result:

```
const cache = new SmartCache();

async function expensiveOperation() {
  console.log('Starting expensive operation...');
  await new Promise(resolve => setTimeout(resolve, 2000));
  return { data: 'valuable result' };
}

// Fire off 5 concurrent requests
const promises = Array(5).fill(null).map(() =>
  cache.cacheReturn(expensiveOperation, 60000)
```

```
);

// Only ONE "Starting expensive operation..." log appears!
// All 5 requests get the same result
const results = await Promise.all(promises);
```

Dynamic Cache Durations

Adjust cache duration based on your data's characteristics:

```
const cache = new SmartCache();

async function getDataWithVariableCache(dataType: string) {
  // Critical data - short cache
  if (dataType === 'critical') {
    return cache.cacheReturn(fetchCriticalData, 1000); // 1 second
  }

  // Static data - long cache
  if (dataType === 'static') {
    return cache.cacheReturn(fetchStaticData, 3600000); // 1 hour
  }

  // Default - moderate cache
  return cache.cacheReturn(fetchNormalData, 60000); // 1 minute
}
```

Cache Different Function Results

Each function call location gets its own cache entry:

```
const cache = new SmartCache();

class DataService {
  async getUserPosts(userId: string) {
    // Cached separately from getUserProfile
    return cache.cacheReturn(
      async () => {
```

```
        const response = await fetch(`/api/users/${userId}/posts`);
        return response.json();
      },
      30000 // Cache for 30 seconds
    );
  }

  async getUserProfile(userId: string) {
    // Different cache entry than getUserPosts
    return cache.cacheReturn(
      async () => {
        const response = await fetch(`/api/users/${userId}/profile`);
        return response.json();
      },
      60000 // Cache for 60 seconds
    );
  }
}
```

□□ Real-World Examples

API Rate Limiting Helper

Prevent hitting rate limits by caching API responses:

```
const cache = new SmartCache();

class GitHubService {
  async getRepoInfo(owner: string, repo: string) {
    return cache.cacheReturn(
      async () => {
        const response = await fetch(
          `https://api.github.com/repos/${owner}/${repo}`,
          { headers: { 'Authorization': `token ${process.env.GITHUB_TOKEN}` } }
        );
        return response.json();
      },
```

```
      300000 // Cache for 5 minutes - respect API limits
    );
  }
}
```

Database Query Optimization

Cache expensive database queries:

```
const cache = new SmartCache();

class DatabaseService {
  async getTopProducts(category: string) {
    return cache.cacheReturn(
      async () => {
        // Expensive aggregation query
        const products = await db.products.aggregate([
          { $match: { category } },
          { $group: { _id: '$productId', totalSales: { $sum: '$sales' } } },
          { $sort: { totalSales: -1 } },
          { $limit: 10 }
        ]);
        return products;
      },
      120000 // Cache for 2 minutes - data doesn't change often
    );
  }
}
```

Computed Values Cache

Cache expensive computations:

```
const cache = new SmartCache();

class AnalyticsService {
  async calculateMetrics(data: number[]) {
    return cache.cacheReturn(
```

```
      async () => {
        console.log('Running expensive calculation...');
        // Simulate expensive computation
        const result = {
          mean: data.reduce((a, b) => a + b, 0) / data.length,
          median: this.calculateMedian(data),
          standardDeviation: this.calculateStdDev(data),
          percentiles: this.calculatePercentiles(data)
        };
        return result;
      },
      600000 // Cache for 10 minutes
    );
  }

  private calculateMedian(data: number[]) { /* ... */ }
  private calculateStdDev(data: number[]) { /* ... */ }
  private calculatePercentiles(data: number[]) { /* ... */ }
}
```

□□ Performance Benefits

SmartCache can dramatically improve your application's performance:

- □□ **Reduce API calls** by up to 90% for frequently accessed data
- ⚡ **Instant responses** for cached data (sub-millisecond)
- □□ **Lower server costs** by reducing redundant computations
- □□ **Protect against thundering herd** problems
- □□ **Automatic cleanup** - expired cache entries are removed automatically

□□ Architecture & How It Works

SmartCache uses a sophisticated approach to caching:

1. **Call Stack Hashing**: When you call `cacheReturn`, SmartCache captures the call stack and generates a SHA-256 hash
2. **Cache Lookup**: It checks if a valid cached result exists for this hash
3. **Concurrent Protection**: If a request is in-flight, new requests wait for the same result

4. **Automatic Expiration**: Each cache entry has a timer that automatically removes it when expired
5. **Memory Efficient**: Only stores what's actively being used, expired entries are cleaned up

□□ Best Practices

□ Do's

- **Cache expensive operations** - API calls, database queries, complex calculations
- **Use appropriate durations** - Match cache time to your data's update frequency
- **Cache at the right level** - Cache complete results, not partial data
- **Monitor memory usage** - Be mindful when caching large objects

□ Don'ts

- **Don't cache user-specific sensitive data** without careful consideration
- **Don't use excessive cache durations** for frequently changing data
- **Don't cache massive objects** that could cause memory issues
- **Don't rely on cache** for critical data consistency

□□ Pro Tips

1. **Layer your caching**: Use SmartCache alongside CDN and database caching for maximum effect
2. **Cache warming**: Pre-populate cache with frequently accessed data on startup
3. **Metrics tracking**: Monitor cache hit rates to optimize cache durations
4. **Error handling**: Always handle potential errors in cached functions

```
const cache = new SmartCache();


// Good - with error handling
async function getResilientData() {
  return cache.cacheReturn(
    async () => {
      try {
        const response = await fetch('/api/data');
```

```
        if (!response.ok) throw new Error(`HTTP ${response.status}`);
        return response.json();
      } catch (error) {
        console.error('Failed to fetch data:', error);
        // Return fallback or rethrow based on your needs
        throw error;
      }
    },
    30000
  );
}
```

□□ Support & Community

- □□ **Found a bug?** Open an issue
- □□ **Have a feature request?** Start a discussion
- □□ **Need help?** Check our comprehensive documentation

License and Legal Information

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

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 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, and any usage must be approved in writing by Task Venture Capital GmbH.

Company Information

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

For any legal inquiries or if you require 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.

changelog.md for @push.rocks/smartcache

2025-08-28 - 1.0.18 - fix(config)

Add local .claude settings to configure assistant/tooling permissions

- Add .claude/settings.local.json to define allowed permissions for local assistant tooling
- Permissions include WebFetch, Bash command patterns, and multiple mcp__serena capabilities
- No changes to library source files or package API

2025-08-26 - 1.0.17 - fix(package)

Update dependency scopes/versions, fix imports and scripts, add pnpm workspace, remove GitLab CI

- Normalized package scopes to include dot notation (e.g. @push.rocks/*) and updated internal import paths in TypeScript files
- Updated runtime dependencies: @push.rocks/smartdelay -> ^3.0.5, @push.rocks/smartpromise -> ^4.2.3 (other @push.rocks/* deps kept/validated)
- Switched dev tooling packages to @git.zone/* namespace and bumped @git.zone/tstest version
- Adjusted npm scripts: more verbose test command and changed build target to use tsfolders
- Added pnpm-workspace.yaml and packageManager entry in package.json
- Removed .gitlab-ci.yml CI configuration

2024-05-29 - 1.0.16 - chore

Project metadata and TypeScript configuration updates.

- Update package description.

- Update tsconfig for TypeScript configuration.
- Update npmextra.json (githost) entries.
- Switch to new organization scheme (org restructuring).

2023-01-08 - 1.0.15 - fix(core)

Core fixes and maintenance.

- Apply core updates and fixes.

2022-06-26 - 1.0.14 - fix(core)

Core fixes and maintenance.

- Apply core updates and fixes.

2021-04-21 - 1.0.13 - fix(core)

Core fixes and maintenance.

- Apply core updates and fixes.

2020-02-03 - 1.0.12 - fix(core)

Core fixes and maintenance.

- Apply core updates and fixes.

2019-10-23 - 1.0.11 - fix(core)

Core fixes and maintenance.

- Apply core updates and fixes.

2018-12-02 - 1.0.10 - fix(core)

Core fixes and maintenance.

- Apply core updates and fixes.

2018-12-02 - 1.0.9 - test

Test updates.

- Update and improve tests.

2018-09-07 - 1.0.8 - fix(core)

Core fixes and maintenance.

- Apply core updates and fixes.

2018-08-21 - 1.0.7 - fix

Remove obsolete logging.

- Remove obsolete console.log statements.

2018-08-19 - 1.0.6 - ci

Continuous integration cleanup.

- Remove obsolete build dependencies from CI/build configuration.

2018-08-19 - 1.0.5 - dependencies

Dependency updates.

- Update project dependencies.

2018-07-13 - 1.0.4 - package

Package metadata fixes.

- Fix package name in metadata.

2018-07-12 - 1.0.3 - meta

Tag-only release.

- No code changes recorded for this tag (tagging/release bookkeeping only).

2018-07-12 - 1.0.2 - package

Project setup.

- Fix project/package setup issues.

2018-07-12 - 1.0.1 - init

Initial release.

- Initial project release and base setup.