Logger Utility - Complete Documentation

Overview

The **Logger** class is a singleton-based logging utility designed for TypeScript applications. It provides environment-aware, formatted console logging with color-coded output, automatic timestamps, and type-safe error handling.

Key Features

- **Singleton Pattern**: Ensures a single Logger instance throughout the application lifecycle
- **Environment-Aware**: Different logging behaviors for `dev`, `prod`, and `test` environments
- **Color-Coded Output**: Distinct colors for different log levels (log, info, warn, error)
- **Automatic Timestamps**: ISO 8601 format timestamps on every log entry
- **Smart Formatting**: Intelligent handling of Errors, objects, and primitive values
- **Production-Safe**: Respects DEBUG flag and environment settings

---

Installation & Import

```typescript
import { Logger, Env } from './logger';
```

Type Definitions

`Env` Type

Supported environment types for the application:

```typescript
type Env = 'dev' | 'prod' | 'test';
```

`LogLevel` Enum

Internal enum for log level categorization:

```typescript
```

```
enum LogLevel {
  LOG = 'log',
  INFO = 'info',
  WARN = 'warn',
  ERROR = 'error',
}
```

API Reference

Static Methods

`Logger.log(...args: any[]): void`

Purpose: Logs general messages for debugging and informational purposes.

Behavior:

- Only outputs in non-production environments (`dev`, `test`)
- Requires `DEBUG` flag to be `true`
- Useful for tracing application flow

Example:

```typescript
Logger.log('Application started');
Logger.log('Processing user request', { userId: 123 });
```

Output:

```
[LOG]    2025-05-26T10:20:30.123Z  "Application started"
[LOG]    2025-05-26T10:20:30.456Z  "Processing user request" { "userId": 123 }
```

`Logger.info(...args: any[]): void`

Purpose: Logs informational messages with higher specificity than general logs.

Behavior:

- Only outputs in `dev` environment with `DEBUG` enabled
- Most restrictive logging level (except error)

- Ideal for detailed debugging information

Example:

```typescript
Logger.info('User logged in', { id: 123, name: 'Alice', email: 'alice@example.com' });
```

Output:

```
[INFO]   2025-05-26T10:20:30.456Z  "User logged in" {
  "id": 123,
  "name": "Alice",
  "email": "alice@example.com"
}
```

`Logger.warn(...args: any[]): void`

Purpose: Logs warning messages for potentially problematic situations.

Behavior:

- Outputs in non-production environments OR when `DEBUG` is enabled
- Useful for deprecated function calls, validation warnings, etc.
- More permissive than `info()` but less critical than `error()`

Example:

```typescript
Logger.warn('Deprecated function called', { alternative: 'newFunction()' });
Logger.warn('High memory usage detected', { memoryMB: 512 });
```

Output:

```
[WARN]   2025-05-26T10:20:30.789Z  "Deprecated function called" {
  "alternative": "newFunction()"
}
```

`Logger.error(...args: any[]): void`

Purpose: Logs error messages and exceptions.

Behavior:

- **Always outputs** regardless of environment or DEBUG setting
- Critical for error tracking and production debugging
- Automatically handles Error objects with stack traces

Example:

```typescript
try {
  // some operation
} catch (error) {
  Logger.error('Failed to process request', error);
}

Logger.error(new Error('Database connection failed'));
```

Output:

```
[ERROR]   2025-05-26T10:20:31.000Z  "Failed to process request" Error: Error: Database connection failed
at processRequest (app.ts:45:12)
  undefined
```

`Logger.getInstance(): Logger`

Purpose: Retrieves the singleton Logger instance.

Returns: The singleton `Logger` instance

Usage:

```typescript
const logger = Logger.getInstance();
console.log(logger.APP_ENV); // 'dev'
console.log(logger.DEBUG); // true
```

Private Methods

`formatArgs(args: any[]): any[]`

Purpose: Internally formats arguments for console output.

Handles:

- **Error objects**: Extracts name, message, stack trace, and cause
- **Complex objects**: Serializes to formatted JSON with 2-space indentation
- **Primitive values**: Returns as-is
- **Unserializable objects**: Returns `[Unserializable object]` placeholder

`getPrefix(level: LogLevel): PrefixConfig`

Purpose: Generates color-coded console prefix with timestamp.

Returns:

```typescript
interface PrefixConfig {
  prefixString: string;  // Format string with %c placeholders
  styles: string[];      // CSS styles for console styling
}
```

Color Mapping:

Level	Color	Code
LOG	Dark Gray	`#333`
INFO	Blue	🟦 `#0066cc`
WARN	Orange	🟧 `#ff9900`
ERROR	Red	🟥 `#cc0000`

Configuration Properties

`APP_ENV: Env` (readonly)

Default: `'dev'`

Purpose: Specifies the current application environment.

Behavior Impact:

- `'dev'`: All log levels active
- `'prod'`: Only `warn()` and `error()` active
- `'test'`: All log levels active (like `dev`)

`DEBUG: boolean` (readonly)

Default: `true`

Purpose: Global debug flag to control logging verbosity.

Behavior Impact:

- `true`: Enables `log()`, `info()`, and `warn()` based on environment
- `false`: Only `error()` is output

Logging Behavior Matrix

Method	dev + DEBUG	dev + !DEBUG	prod + DEBUG	prod + !DEBUG	test
`log()`	✅	✗	✗	✗	✅
`info()`	✅	✗	✗	✗	✅
`warn()`	✅	✗	✅	✗	✅
`error()`	✅	✅	✅	✅	✅

Usage Examples

Basic Logging

```typescript
Logger.log('Server listening on port 3000');
Logger.info('Database connected successfully');
Logger.warn('Rate limit approaching: 950/1000 requests');
Logger.error('Failed to fetch user data');
```

With Objects

```typescript
const user = { id: 1, name: 'John', email: 'john@example.com' };
Logger.info('User data retrieved', user);
```

Error Handling

```typescript
```

```
try {
  await fetchData();
} catch (error) {
  Logger.error('Failed to fetch data', error);
}
```

Complex Scenarios

```typescript
const apiResponse = {
  status: 200,
  data: { users: [1, 2, 3] },
  meta: { timestamp: new Date().toISOString() }
};

Logger.log('API call completed', apiResponse);
```

Design Patterns Used

1. Singleton Pattern

Ensures only one Logger instance exists application-wide, promoting memory efficiency and consistent state.

```typescript
private static instance: Logger;

public static getInstance(): Logger {
  if (!Logger.instance) {
    Logger.instance = new Logger();
  }
  return Logger.instance;
}
```

2. Type Safety

Leverages TypeScript for compile-time type checking:

- Enum-based log levels prevent typos
- Interface definitions for complex types
- Type-safe argument handling

3. Environment Awareness

Adapts logging behavior based on deployment environment, reducing noise in production while preserving debugging capabilities during development.

Best Practices

✅ **Do**:

- Use `Logger.error()` for all exception handling
- Use `Logger.info()` for detailed debugging in development
- Use `Logger.warn()` for deprecated features or performance issues
- Handle Error objects properly in catch blocks

❌ **Don't**:

- Use `console.log()` directly; always use Logger
- Log sensitive information (passwords, API keys, PII)
- Over-log in production; rely on proper log levels
- Mix Logger with other logging libraries

Performance Considerations

- **Minimal Overhead**: Static method calls are optimized by JavaScript engines
- **Conditional Output**: Logging statements are short-circuited based on environment/DEBUG flag
- **Memory Efficient**: Singleton pattern prevents multiple instances
- **Stack Trace Handling**: Automatic stack trace capture for Error objects

---

Version Information

- **Version**: 1.0.0
- **Author**: AGBOKOUDJO Franck
- **License**: MIT
- **Company**: INTERNATIONALES WEB SERVICES
- **Contact**: +229 0167 25 18 86

Changelog

v1.0.0 (Initial Release)

- Singleton Logger implementation
- Support for dev, prod, test environments
- Color-coded console output with timestamps
- Smart error and object formatting
- Environment-aware logging levels

License