Sailplane Documentation

Release 1.0.0

Onica Group

Sep 05, 2019

Contents

CHAPTER 1

What is this?

While developing serverless applications at Onica, we found certain patterns being used repeatedly, and code being copied from one project to the next. These commonalities have been extracted, matured, and gathered into a reusable collection.

Sailplane is the result: a collection of useful packages for use in developing code that runs in AWS. They are primarily used in Lambda functions, but most are useful in other services that use the Node.js 8+ runtime as well.

The Typescript source is compiled to ES6 Javascript for portability, along with Typescript type definition files. While the typing provides the expected benefit, these utilities may be used in plain Javascript as well.

Every tool is the genesis of real world needs, and they continue to evolve. This collection is part of Onica's commitment to give back to the open source community. Find this and other Onica open source repositories on GitHub.

CHAPTER 2

Content

Each utility is described on its own page:

2.1 AwsHttps

HTTPS client with AWS Signature v4.

2.1.1 Overview

The AwsHttps class is an HTTPS (notice, *not* HTTP) client purpose made for use in and with AWS environments.

- Simple Promise or async syntax
- Optionally authenticates to AWS via AWS Signature v4 using aws4
- Familiar options.
- Helper to build request options from URL object
- Light-weight

- Easily extended for unit testing

AwsHttps is dependent on *Logger* and AWS4 for signing.

2.1.2 Install

```
npm install @sailplane/aws-https @sailplane/logger
```

2.1.3 Examples

Simple example to GET from URL:

```
const url = new URL('https://www.onica.com/ping.json');
const http = new AwsHttps();

// Build request options from a method and URL
const options = http.buildOptions('GET' url);

// Make request and parse JSON response.
const ping = await http.request(options);
```

Example hitting API with IAM credentials:

```
const awsHttp = new AwsHttps();
const options: AwsHttpsOptions = {
    // Same options as https://nodejs.org/api/http.html#http_http_request_options_
↪callback
    method: 'GET',
    hostname: apiEndpoint,
    path: '/cloud-help',
    headers: {
        'accept': 'application/json; charset=utf-8',
        'content-type': 'application/json; charset=utf-8'
    },
    timeout: 10000,

    // Additional option for POST, PUT, or PATCH:
    body: JSON.stringify({ website: "https://www.onica.com" }),

    // Additional option to apply AWS Signature v4
    awsSign: true
};

try {
    const responseObj = await awsHttp.request(options);
    process(responseObj);
}
catch (err) {
    // HTTP status response is in statusCode field
    if (err.statusCode === 404) {
        process(undefined);
    }
    else {
        throw err;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The *ElasticsearchClient* package is a simple example using AwsHttps.

2.1.4 Unit testing your services

- Have your service receive `AwsHttps` in the constructor. Consider using *Injector*.
- In your service unit tests, create a new class that extends AwsHttps and returns your canned response.
- Pass your fake AwsHttps class into the constructor of your service under test.

```
export class AwsHttpsFake extends AwsHttps {
    constructor() {
        super();
    }

    async request(options: AwsHttpsOptions): Promise<any | null> {
        // Check for expected options. Example:
        expect(options.path).toEqual('/expected-path');

        // Return canned response
        return Promise.resolve({ success: true });
    }
}
```

2.1.5 Type Declarations

```
/// <reference types="node" />
import * as https from "https";
import { URL } from "url";
/**
 * Same options as https://nodejs.org/api/http.html#http_http_request_options_callback
 * with the addition of optional body to send with POST, PUT, or PATCH
 * and option to AWS Sig4 sign the request.
 */
export declare type AwsHttpsOptions = https.RequestOptions & {
    /** Body content of HTTP POST, PUT or PATCH */
    body?: string;
    /** If true, apply AWS Signature v4 to the request */
    awsSign?: boolean;
};
/**
 * Light-weight utility for making HTTPS requests in AWS environments.
 */
export declare class AwsHttps {
    private readonly verbose?;
    /** Resolves when credentials are available - shared by all instances */
    private static credentialsPromise;
    /**
     * Constructor.
     * @param verbose true to log everything, false for silence,
     *                undefined (default) for normal logging.
```

(continues on next page)

(continued from previous page)

```
     */
    constructor(verbose?: boolean | undefined);
    /**
     * Perform an HTTPS request and return the JSON body of the result.
     *
     * @params options https request options, with optional body and awsSign
     * @returns parsed JSON content, or null if none.
     * @throws {Error{message,status,statusCode}} error if HTTP result is not 2xx or␣
↪unable
     *              to parse response. Compatible with http-errors package.
     */
    request(options: AwsHttpsOptions): Promise<any | null>;
    /**
     * Helper to build a starter AwsHttpsOptions object from a URL.
     *
     * @param method an HTTP method/verb
     * @param url the URL to request from
     * @param connectTimeout (default 5000) milliseconds to wait for connection to␣
↪establish
     * @returns an AwsHttpsOptions object, which may be further modified before use.
     */
    buildOptions(method: 'DELETE' | 'GET' | 'HEAD' | 'OPTIONS' | 'POST' | 'PUT' |
↪'PATCH', url: URL, connectTimeout?: number): AwsHttpsOptions;
    /**
     * Helper for signing AWS requests
     * @param request to make
     * @return signed version of the request.
     */
    private awsSign;
}
```

2.2 ElasticsearchClient

Communicate with AWS Elasticsearch.

2.2.1 Overview

Other solutions for communicating with Elasticsearch are either incompatible with AWS, very heavy weight, or both. This client gets the job done and is as simple as it gets!

- Simple Promise or async syntax
- Authenticates to AWS via AWS Signature v4
- Light-weight

Use it with Elasticsearch's Document API.

`ElasticsearchClient` depends on two other utilities to work:

- *AwsHttps*
- *Logger*

2.2.2 Install

```
npm install @sailplane/elasticsearch-client @sailplane/aws-https @sailplane/logger
```

2.2.3 Examples

Simple example:

```
get(id: string): Promise<Ticket> {
    return this.es.request('GET', '/ticket/local/' + id)
        .then((esDoc: ElasticsearchResult) => esDoc._source as Ticket);
}
```

See *More Examples* for a comprehensive example.

2.2.4 Type Declarations

```
import { AwsHttps } from "@sailplane/aws-https";
/**
 * All-inclusive possible properties of returned results from ElasticsearchClient
 */
export interface ElasticsearchResult {
    _shards?: {
        total: number;
        successful: number;
        failed: number;
        skipped?: number;
    };
    _index?: string;
    _type?: string;
    _id?: string;
    _version?: number;
    result?: "created" | "deleted" | "noop";
    found?: boolean;
    _source?: any;
    took?: number;
    timed_out?: boolean;
    hits?: {
        total: number;
        max_score: number | null;
        hits?: [{
            _index: string;
            _type: string;
            _id: string;
            _score: number;
            _source?: any;
        }];
    };
    deleted?: number;
    failures?: any[];
}
/**
 * Lightweight Elasticsearch client for AWS.
 *
```

(continues on next page)

(continued from previous page)

```
 * Suggested use with Injector:
 *   Injector.register(ElasticsearchClient, () => {
 *     const endpoint: string = process.env.ES_ENDPOINT!;
 *     logger.info('Connecting to Elasticsearch @ ' + endpoint);
 *     return new ElasticsearchClient(new AwsHttps(), endpoint);
 *   });
 */
export declare class ElasticsearchClient {
    private readonly awsHttps;
    private readonly endpoint;
    /**
     * Construct.
     * @param awsHttps injection of AwsHttps object to use.
     * @param {string} endpoint Elasticsearch endpoint host name
     */
    constructor(awsHttps: AwsHttps, endpoint: string);
    /**
     * Send a request to Elasticsearch.
     * @param {"GET" | "DELETE" | "PUT" | "POST"} method
     * @param {string} path per Elasticsearch Document API
     * @param {any?} body request content as object, if any for the API
     * @returns {Promise<ElasticsearchResult>} response from Elasticsearch. An HTTP␣
↪404
     *              response is translated into an ElasticsearchResult with␣
↪found=false
     * @throws {Error{message,status,statusCode}} error if HTTP result is not 2xx or␣
↪404
     *              or unable to parse response. Compatible with http-errors package.
     */
    request(method: 'DELETE' | 'GET' | 'PUT' | 'POST', path: string, body?: any):␣
↪Promise<ElasticsearchResult>;
}
```

2.3 ExpiringValue

Value that is instantiated on-demand and cached for a limited time.

2.3.1 Overview

`ExpiringValue` is a container for a value that is instantiated on-demand (lazy-loaded via factory) and cached for a limited time. Once the cache expires, the factory is used again on next demand to get a fresh version.

In Lambda functions, it is useful to cache some data in instance memory to avoid recomputing or fetching data on every invocation. In the early days of Lambda, instances only lasted 15 minutes and thus set an upper-limit on how stale the cached data could be. With instances now seen to last for many hours, a mechanism is needed to deal with refreshing stale content - thus `ExpiringValue` was born.

`ExpiringValue` is not limited to Lambdas, though. Use it anywhere you want to cache a value for a limited time. It even works in the browser for client code.

A good use is with *StateStorage*, to load configuration and cache it, but force the refresh of that configuration periodically.

2.3.2 Install

```
npm install @sailplane/expiring-value
```

2.3.3 Example

Simplistic example of using ExpiringValue to build an HTTP cache:

```
const CACHE_PERIOD = 90_000; // 90 seconds
const https = new AwsHttps();
const cache = {};

export function fetchWithCache(url: string): Promise<any> {
    if (!cache[url]) {
        cache[url] = new ExpiringValue<any>(() => loadData(url), CACHE_PERIOD);
    }

    return cache[url].get();
}

function loadData(url: string): any {
    const req = https.buildRequest('GET', new URL(url));
    return https.request(req);
}
```

See *More Examples* for another example.

2.3.4 Type Declarations

```
/**
 * Container for a value that is lazy-loaded whenever needed.
 * Further, it expires after a given time to avoid overly-stale data.
 */
export declare class ExpiringValue<T> {
    private factoryFn;
    private ttl;
    /** Cached value */
    private value;
    /** Epoch millisecond time of when the current value expires */
    private expiration;
    /**
     * Construct a new expiring value.
     *
     * @param factoryFn factory to lazy-load the value
     * @param ttl milliseconds the value is good for, after which it is reloaded.
     */
    constructor(factoryFn: (() => Promise<T>), ttl: number);
    /**
     * Get value; lazy-load from factory if not yet loaded or if expired.
     */
    get(): Promise<T>;
    /**
     * Clear/expire the value now.
     * Following this with a get() will reload the data from the factory.
```

(continues on next page)

(continued from previous page)

```
     */
    clear(): void;
    /**
     * Is the value expired (or not set)
     */
    isExpired(): boolean;
}
```

2.4 Injector

Light-weight and type-safe Dependency Injection.

2.4.1 Overview

Simple, light-weight, lazy-instantiating, and type-safe dependency injection in Typescript! Perfect for use in Lambdas and unit test friendly.

It is built on top of BottleJS, with a simple type-safe wrapper. The original *bottle* is available for more advanced use, though. Even if you are not using Typescript, you may still prefer this simplified interface over using BottleJS directly.

`Injector` depends on one other utility to work:

- *Logger*

2.4.2 Install

```
npm install @sailplane/injector @sailplane/logger
```

2.4.3 Usage with Examples

Register a class with no dependencies and retrieve it

Use `Injector.register(className)` to register a class with the Injector. Upon the first call to `Injector.get(className)`, the singleton instance will be created and returned.

Example:

```
import {Injector} from "@sailplane/injector";

export class MyService {
}
Injector.register(MyService);

// Later...
const myService = Injector.get(MyService)!;
```

See the next section for another example of receiving a registered class as a dependency.

Register a class with an array of dependencies

Use `Injector.register(className, dependencies: [])` to register a class with constructor dependencies with the Injector. Upon the first call to `Injector.get(className)`, the singleton instance will be created and returned.

`dependencies` is an array of either class names or strings with the names of things.

Example:

```
import {Injector} from "@sailplane/injector";

export class MyHelper {
}
Injector.register(MyHelper);
Injector.registerConstant('stage', 'dev');

export class MyService {
    constructor(private readonly helper: MyHelper,
                private readonly stage: string) {
    }
}
Injector.register(MyService, [MyHelper, 'stage']);
```

Register a class with static $inject array

Define your class with a `static $inject` member as an array of either class names or strings with the names of registered dependencies, and a matching constructor that accepts the dependencies in the same order. Then use `Injector.register(className)` to register a class. Upon the first call to `Injector.get(className)`, the singleton instance will be created with the specified dependencies.

This functions just like the previous use, but allows you to specify the dependencies right next to the constructor instead of after the class definition; thus making it easier to keep the two lists synchronized.

Example:

```
import {Injector} from "@sailplane/injector";

export class MyHelper {
}
Injector.register(MyHelper);
Injector.registerConstant('stage', 'dev');

export class MyService {
    static readonly $inject = [MyHelper, 'stage'];
    constructor(private readonly helper: MyHelper,
                private readonly stage: string) {
    }
}
Injector.register(MyService);
```

Register a class with a factory

If your class takes constructor parameters that are not in the dependency system, then you can register a factory function.

Use `Injector.register<T>(className, factory: ()=>T)` to register a class with your own factory function for instantiating the singleton instance.

Example:

```
import {Injector} from "@sailplane/injector";

export class MyHelper {
}
Injector.register(MyHelper);

export class MyService {
    constructor(private readonly helper: MyHelper,
                private readonly stage: string) {
    }
}
Injector.register(MyService,
                  () => new MyService(Injector.get(MyHelper)!, process.env.STAGE!));
```

Register anything with a factory and fetch it by name

If you need to inject something other than a class, you can register a factory to create anything and give it a name. This is particularly useful if you have multiple implementations of an interface, and one to register one of them by the interface name at runtime. Since interfaces don't exist at runtime (they don't exist in Javascript), you must define the name yourself.

Use `Injector.registerFactory<T>(name: string, factory: ()=>T)` to register any object with your own factory function for returning the singleton instance.

Example: Inject a configuration

```
import {Injector} from "@sailplane/injector";

Injector.registerFactory('config', () => {
    // Note that this returns a Promise
    return Injector.get(StateStorage)!.get('MyService', 'config');
});

// Later...

const config = await Injector.getByName('config');
```

Example: Inject an interface implementation

```
import {Injector} from "@sailplane/injector";

export interface FoobarService {
    doSomething(): void;
}

export class FoobarServiceImpl implements FoobarService {
    constructor(private readonly stateStorage: StateStorage); {}

    doSomething(): void {
        this.stateStorage.set('foobar', 'did-it', 'true');
    }
}
```

(continues on next page)

(continued from previous page)

```
export class FoobarServiceDemo implements FoobarService {
    doSomething(): void {
        console.log("Nothing really");
    }
}

Injector.registerFactory('FoobarService', () => {
    if (process.env.DEMO! === 'true') {
        return new FoobarServiceDemo();
    }
    else {
        return new FoobarServiceImpl(Injector.get(StateStorage)!);
    }
});

// Elsewhere...

export class MyService {
    static readonly $inject = ['FoobarService']; // Note: This is a string!
    constructor(private readonly foobareSvc: FoobarService) {
    }
}
Injector.register(MyService);
```

Register a constant value and fetch it by name

Use `Injector.registerConstant<T>(name: string, value: T)` to register any object as a defined value. Unlike all other registrations, the value is passed in rather than being lazy-created.

Example:

```
import {Injector} from "@sailplane/injector";
import {environment} from "environment";

Injector.registerConstant('environment-config', environment);

// Later...

const myEnv = Injector.getByName('environment-config');
```

More Examples

See *More Examples* for another example.

Dependency Evaluation

Dependencies are not evaluated until the class is instantiated, thus the order of registration does not matter.

Cyclic constructor dependencies will fail. This probably indicates a design problem, but you can break the cycle by calling `Injector.get(className)` when needed (outside of the constructor), instead of injecting into the constructor.

This is a perfectly valid way of using Injector (on demand rather than upon construction). It does require that unit test mocks be registered with the Injector rather than passed into the constructor.

2.4.4 Type Declarations

```
/**
 * Dependency Injection (DI) injector.
 *
 * Using BottleJs because it is _extremely_ light weight and lazy-instantiate,
 * unlike any Typescript-specific solutions.
 *
 * @see https://github.com/young-steveo/bottlejs
 */
import * as Bottle from 'bottlejs';
declare type DependencyList = ({
    new (...args: any[]): any;
} | string)[];
/**
 * Wraps up type-safe version of BottleJs for common uses.
 *
 * The raw bottle is also available.
 */
export declare class Injector {
    static readonly bottle: Bottle<string>;
    /**
     * This may be called at beginning of process.
     * Not required at this point, but may help catch some mistakes.
     */
    static initialize(): void;
    /**
     * Register a class.
     *
     * Example service that lazy instantiates with no arguments:
     * - Injector.register(MyServiceClass);
     *
     * Example service that lazy instantiates with other registered items as␣
↪arguments to
     * the constructor:
     * - Injector.register(MyServiceClass, [DependentClass, OtherClass, 'constant-name
↪']);
     *
     * Example service that lazy instantiates with other registered items specified
     * as $inject:
     * - class MyServiceClass {
     * -   static readonly $inject = [OtherClass, 'constant-name'];
     * -   constructor(other, constValue) {};
     * - }
     * - Injector.register(MyServiceClass);

     * Example service that lazy instantiates with a factory function:
     * - Injector.register(MyServiceClass,
     * -                   () => new MyServiceClass(Injector.get(OtherClass)!,␣
↪MyArg));
     *
     * @param clazz the class to register. Ex: MyClass. NOT an instance, the class.
     * @param factoryOrDependencies see above examples. Optional.
```

(continues on next page)

(continued from previous page)

```
     * @return true if registered, false if not (duplicate or bad)
     */
    static register<T>(clazz: {
        new (...args: any[]): T;
        $inject?: DependencyList;
        name: string;
    }, factoryOrDependencies?: (() => T) | DependencyList): boolean;
    /**
     * Register a factory by name.
     *
     * Example:
     * - Injector.registerFactory('MyThing', () => Things.getOne());
     *
     * @see #getByName(name)
     * @param name name to give the inject.
     * @param factory function that returns a class.
     * @return true if registered, false if not (duplicate or bad)
     */
    static registerFactory<T>(name: string, factory: (() => T)): boolean;
    /**
     * Register a named constant value.
     *
     * @see #getByName(name)
     * @param name name to give this constant.
     * @param value value to return when the name is requested.
     */
    static registerConstant<T>(name: string, value: T): void;
    /**
     * Get instance of a class.
     * Will instantiate on first request.
     *
     * @param clazz the class to fetch. Ex: MyClass. NOT an instance, the class.
     * @return the singleton instance of the requested class, undefined if not␣
↪registered.
     */
    static get<T>(clazz: {
        new (...args: any[]): T;
    }): T | undefined;
    /**
     * Get a registered constant or class by name.
     *
     * @see #registerFactory(name, factory, force)
     * @see #registerConstant(name, value, force)
     * @param name
     * @return the singleton instance registered under the given name,
     *         undefined if not registered.
     */
    static getByName<T>(name: string): T | undefined;
}
export {};
```

2.5 LambdaUtils

Lambda handler middleware.

2.5.1 Overview

There's a lot of boilerplate in Lambda handlers. This collection of utility functions leverages the great Middy library to add middleware functionality to Lambda handlers. You can extend it with your own middleware.

Middy gives you a great start as a solid middleware framework, but by itself you are still repeating the middleware registrations on each handler, its exception handler only works with errors created by the http-errors package, and you still have to format your response in the shape required by API Gateway.

`LambadUtils` takes Middy further and is extendable so that you can add your own middleware (authentication & authorization, maybe?) on top of it.

Used with API Gateway, the included middlewares:

- Set CORS headers.
- Normalize incoming headers to mixed-case
- If incoming content is JSON text, replaces event.body with parsed object.
- Ensures that event.queryStringParameters and event.pathParameters are defined, to avoid TypeErrors.
- **Ensures that handler response is formatted properly as a successful API Gateway result.**
  - Unique to LambdaUtils!
  - Simply return what you want as the body of the HTTP response.
- Catch http-errors exceptions into proper HTTP responses.
- **Catch other exceptions and return as HTTP 500.**
  - Unique to LambdaUtils!
  - Besides providing better feedback to the client, not throwing an exception out of your handler means that your instance will not be destroyed and suffer a cold start on the next invocation.
- Leverages async syntax.

See Middy middlewares for details on those. Not all Middy middlewares are in this implementation, only common ones that are generally useful in all APIs. You may extend LambdaUtils's `wrapApiHandler()` function in your projects, or use it as an example to write your own, to add more middleware!

WARNING: Middy has, historically, introduced some breaking changes in previous minor version bumps. LambdaUtils has been built and tested with Middy 0.29.0, and should continue to work with future versions, but bumping your Middy version beyond 0.29.0 may produce some weird errors.

`LambdaUtils` depends on two other utilities to work:

- *Logger*
- Middy

2.5.2 Install

```
npm install @sailplane/lambda-utils @sailplane/logger middy
```

2.5.3 Examples

General use

```
import {APIGatewayEvent} from 'aws-lambda';
import * as LambdaUtils from "@sailplane/lambda-utils";
import * as createError from "http-errors";

export const hello = LambdaUtils.wrapApiHandler(async (event: LambdaUtils.
↪APIGatewayProxyEvent) => {
    // These event objects are now always defined, so don't need to check for␣
↪undefined.
    const who = event.pathParameters.who;
    const points = parseInt(event.queryStringParameters.points || 0);

    if (points > 0) {
        let message = 'Hello ' + who;
        for (; points > 0; --points)
            message = message + '!';

        return {message};
    }
    else {
        // LambdaUtils will catch and return HTTP 400
        throw new createError.BadRequest('Missing points parameter');
    }
});
```

See *More Examples* for another example.

Extending LambdaUtils for your own app

```
import {ProxyHandler} from "aws-lambda";
import * as LambdaUtils from "@sailplane/lambda-utils";
import {userAuthMiddleware} from "./user-auth"; //you write this

export interface WrapApiHandlerOptions {
    noUserAuth?: boolean;
    requiredRole?: string;
}

export function wrapApiHandlerWithAuth(options: WrapApiHandlerOptions,
                                       handler: LambdaUtils.AsyncProxyHandler):␣
↪ProxyHandler {
    let wrap = LambdaUtils.wrapApiHandler(handler);

    if (!options.noUserAuth) {
        wrap.use(userAuthMiddleware(options.requiredRole));
    }

    return wrap;
}
```

2.5.4 Type Declarations

```
import { APIGatewayProxyEvent as AWS_APIGatewayProxyEvent, APIGatewayEvent, Context,␣
↪ProxyResult } from "aws-lambda";
import * as middy from "middy";
/** Define the async version of ProxyHandler */
export declare type AsyncProxyHandler = (event: APIGatewayEvent, context: Context) =>␣
↪Promise<any>;
/**
 * Casted interface for APIGatewayProxyEvents as converted through the middleware
 */
export interface APIGatewayProxyEvent extends AWS_APIGatewayProxyEvent {
    /**
     * HTTP Request body, parsed from a JSON string into an object.
     */
    body: any | null;
    /**
     * HTTP Path Parameters, always defined, never null
     */
    pathParameters: {
        [name: string]: string;
    };
    /**
     * HTTP URL query string parameters, always defined, never null
     */
    queryStringParameters: {
        [name: string]: string;
    };
}
/**
 * Middleware to handle any otherwise unhandled exception by logging it and generating
 * an HTTP 500 response.
 *
 * Fine tuned to work better than the Middy version.
 */
export declare const unhandledExceptionMiddleware: () => {
    onError: (handler: middy.HandlerLambda<AWS_APIGatewayProxyEvent, import("aws-
↪lambda").APIGatewayProxyResult>, next: middy.NextFunction) => void;
};
/**
 * Middleware to allow an async handler to return its exact response body.
 * This middleware will wrap it up as an APIGatewayProxyResult.
 * Must be registered as the last (thus first to run) "after" middleware.
 */
export declare const resolvedPromiseIsSuccessMiddleware: () => {
    after: (handler: middy.HandlerLambda<AWS_APIGatewayProxyEvent, import("aws-lambda
↪").APIGatewayProxyResult>, next: middy.NextFunction) => void;
};
/**
 * Wrap an API Gateway proxy lambda function handler to add features:
 * - Set CORS headers.
 * - Normalize incoming headers to mixed-case
 * - If incoming content is JSON text, replace event.body with parsed object.
 * - Ensures that event.queryStringParameters and event.pathParameters are defined,
 *   to avoid TypeErrors.
 * - Ensures that handler response is formatted properly as a successful
 *   API Gateway result.
```

(continues on next page)

(continued from previous page)

```
 * - Catch http-errors exceptions into proper HTTP responses.
 * - Catch other exceptions and return as HTTP 500
 *
 * This wrapper includes commonly useful middleware. You may further wrap it
 * with your own function that adds additional middleware, or just use it as
 * an example.
 *
 * @param handler async function to wrap
 * @see https://middy.js.org/docs/middlewares.html
 * @see https://www.npmjs.com/package/http-errors
 */
export declare function wrapApiHandler(handler: AsyncProxyHandler): middy.Middy
↪<APIGatewayEvent, ProxyResult>;
/**
 * Construct the object that API Gateway wants back upon a successful run. (HTTP 200␣
↪Ok)
 *
 * This normally is not needed. If the response is simply the content to return as the
 * body of the HTTP response, you may simply return it from the handler given to
 * #wrapApiHandler(handler). It will automatically transform the result.
 *
 * @param result object to serialize into JSON as the response body
 * @returns {ProxyResult}
 */
export declare function apiSuccess(result?: any): ProxyResult;
/**
 * Construct the object that API Gateway wants back upon a failed run.
 *
 * Often, it is simpler to throw an http-errors exception from your #wrapApiHandler
 * handler.
 *
 * @see https://www.npmjs.com/package/http-errors
 * @param statusCode HTTP status code, between 400 and 599.
 * @param message string to return in the response body
 * @returns {ProxyResult}
 */
export declare function apiFailure(statusCode: number, message?: string): ProxyResult;
```

2.6 Logger

CloudWatch and serverless-offline friendly logger.

2.6.1 Overview

Sadly, `console.log` is the #1 debugging tool when writing serverless code. Logger extends it with levels, timestamps, context/category names, and object formatting. It's just a few small incremental improvements, and yet together takes logging a leap forward.

If your transpiling, be sure to enable source maps (in Typescript, Babel) and use the source-map-support library so that you get meaningful stack traces.

2.6.2 Install

```
npm install @sailplane/logger
```

2.6.3 Examples

```
import {Logger, LogLevels} from "@sailplane/logger";
const logger = new Logger('name-of-module');

logger.info("Hello World!");
// INFO name-of-module: Hello World!

Logger.globalLogLevel = LogLevels.INFO;
logger.debug("DEBUG < INFO.");
// No output

Logger.logTimestamps = true;
logger.info("Useful local log");
// 2018-11-15T18:26:20 INFO name-of-module: Useful local log

logger.warnObject("Exception ", {message: "oops"});
// 2018-11-15T18:29:38 INFO name-of-module: Exception {message:"oops"}

Logger.formatObjects = true;
logger.errorObject("Exception ", {message: "oops"});
// 2018-11-15T18:30:49 INFO name-of-module: Exception {
//   message: "oops"
// }
```

2.6.4 Configuration / Behavior

The output of Logger varies based on some global settings, whether the Lambda is executing in AWS or local (serverless-offline, SAM offline), and whether the runtime is Node.js 10 vs earlier versions.

globalLogLevel

Only this level and higher will be output. Options are 'DEBUG', 'INFO', 'WARN', 'ERROR', 'NONE'. Default is 'DEBUG'.

Override programmatically: `Logger.globalLogLevel = 'WARN';`

Override from environment variable: `export LOG_LEVEL=INFO`

outputLevels

If true, output level text on each line.

Default is true when running outside of AWS Lambda environment, and in a Lambda runtime earlier than Node.js v10. Starting with Node.js v10, the Lambda runtime itself adds log levels to the output sent to CloudWatch. This is detected and outputLevels is set to false.

Override programmatically: `Logger.outputLevels = false;`

logTimestamps

Output date & time prefix on logged lines?

Timestamps are enabled outside of AWS Lambda, and disabled in AWS Lambda where CloudWatch provides time stamping.

Override programmatically: `Logger.logTimestamps = false;`

formatObjects

If true, objects logged by debugObject(), infoObject(), warnObject() and errorObject() will be formatted with proper indentation. If false, no formatting is performed.

Formatting is enabled outside of AWS Lambda, and disabled in AWS Lambda where CloudWatch provides formatting.

2.6.5 Type Declarations

```
export declare enum LogLevels {
    NONE = 0,
    ERROR = 1,
    WARN = 2,
    INFO = 3,
    DEBUG = 4
}
/**
 * Custom logger class.
 *
 * Works much like console's logging, but includes levels, date/time,
 * and category (file) on each line.
 *
 * Usage:
 *   import {Logger} from "@sailplane/logger";
 *   const logger = new Logger('name-of-module');
 *   logger.info("Hello World!");
 */
export declare class Logger {
    private category;
    /**
     * Global logging level.
     * May be initialized via LOG_LEVEL environment variable, or set by code.
     */
    static globalLogLevel: LogLevels;
    /**
     * Include the level in log output?
     * Defaults to true if not streaming to CloudWatch or running Node.js v9 or older.
     * May also be set directly by code.
     * Note: AWS behavior changed with nodejs10.x runtime - it now includes log␣
↪levels automatically.
     */
    static outputLevels: boolean;
    /**
     * Include timestamps in log output?
     * Defaults to false if streaming to CloudWatch, true otherwise.
     * (CloudWatch provides timestamps.)
     * May override by setting the LOG_TIMESTAMPS environment variable to 'true' or
↪'false',
```

(continues on next page)

(continued from previous page)

```
     * or set by code.
     */
    static logTimestamps: boolean;
    /**
     * Pretty format objects when stringified to JSON?
     * Defaults to false if streaming to CloudWatch, true otherwise.
     * (Best to let CloudWatch provide the formatting.)
     * May override by setting the LOG_FORMAT_OBJECTS environment variable to 'true'␣
↪or 'false',
     * or set by code.
     */
    static formatObjects: boolean;
    /**
     * Construct.
     * @param category logging category to include in each line.
     *                 Source file name or class name are good choices.
     */
    constructor(category: string);
    /**
     * Format a log line. Helper for #debug, #info, #warn, and #error.
     *
     * @param level logging level
     * @param message text to log
     * @param optionalParams A list of JavaScript objects to output.
     *                       The string representations of each of these objects are
     *                       appended together in the order listed and output.
     * @return array to pass to a console function, or null to output nothing.
     */
    private formatLog;
    /**
     * Format a log ine with a stringified object.
     * Helper for #debugObject, #infoObject, #warnObject, and #errorObject.
     *
     * @param level logging level
     * @param message text to log - may want to end with a space,
     *                as `{` will immediately follow.
     * @param object object to stringify to JSON and append to message
     * @return array to pass to a console function, or null to output nothing.
     */
    private formatLogStringified;
    /**
     * Log a line at DEBUG level.
     *
     * @param message text to log
     * @param optionalParams A list of JavaScript objects to output.
     *                       The string representations of each of these objects are
     *                       appended together in the order listed and output.
     */
    debug(message: any, ...optionalParams: any[]): void;
    /**
     * Log a line at INFO level.
     *
     * @param message text to log
     * @param optionalParams A list of JavaScript objects to output.
     *                       The string representations of each of these objects are
     *                       appended together in the order listed and output.
     */
```

(continues on next page)

(continued from previous page)

```
    info(message: any, ...optionalParams: any[]): void;
    /**
     * Log a line at WARN level.
     *
     * @param message text to log
     * @param optionalParams A list of JavaScript objects to output.
     *                       The string representations of each of these objects are
     *                       appended together in the order listed and output.
     */
    warn(message: any, ...optionalParams: any[]): void;
    /**
     * Log a line at ERROR level.
     *
     * @param message text to log
     * @param optionalParams A list of JavaScript objects to output.
     *                       The string representations of each of these objects are
     *                       appended together in the order listed and output.
     */
    error(message: any, ...optionalParams: any[]): void;
    /**
     * Log a line at DEBUG level with a stringified object.
     *
     * @param message text to log - may want to end with a space,
     *                as `{` will immediately follow.
     * @param object object to stringify to JSON and append to message
     */
    debugObject(message: string, object: any): void;
    /**
     * Log a line at INFO level with a stringified object.
     *
     * @param message text to log - may want to end with a space,
     *                as `{` will immediately follow.
     * @param object object to stringify to JSON and append to message
     */
    infoObject(message: string, object: any): void;
    /**
     * Log a line at WARN level with a stringified object.
     *
     * @param message text to log - may want to end with a space,
     *                as `{` will immediately follow.
     * @param object object to stringify to JSON and append to message
     */
    warnObject(message: string, object: any): void;
    /**
     * Log a line at ERROR level with a stringified object.
     *
     * @param message text to log - may want to end with a space,
     *                as `{` will immediately follow.
     * @param object object to stringify to JSON and append to message
     */
    errorObject(message: string, object: any): void;
}
```

2.7 StateStorage

Serverless state and configuration storage.

2.7.1 Overview

The AWS Parameter Store (SSM) was originally designed as a place to store configuration. It turns out that it is also a pretty handy place for storing small bits of state information in between serverless executions.

StateStorage is a simple wrapper for SSM *getParameter* and *putParameter* functions, abstracting it into a contextual storage of small JSON objects.

Why use this instead of AWS SSM API directly?

- Simple Promise or async syntax
- Automatic object serialization/deserialization
- Logging
- Consistent naming convention

`Injector` depends on one other utility to work:

- *Logger*

2.7.2 Install

```
npm install @sailplane/state-storage @sailplane/logger
```

2.7.3 Examples

Your Lambda will need permission to access the Parameter Store. Here's an example in `serverless.yml`:

```
provider:
  name: aws
  runtime: nodejs8.10

  environment:
    STATE_STORAGE_PREFIX: /${opt:stage}/myapp

  iamRoleStatements:
    - Effect: Allow
      Action:
        - ssm:GetParameter
        - ssm:PutParameter
      Resource: "arn:aws:ssm:${opt:region}:*:parameter${self:provider.environment.
↪STATE_STORAGE_PREFIX}/*"
```

Simple example storing state

```
import {StateStorage} from "@sailplane/state-storage";

const stateStore = new StateStorage(process.env.STATE_STORAGE_PREFIX!);
```

(continues on next page)

(continued from previous page)

```
export async function myHandler(event, context): Promise<any> {
  let state = await stateStore.get('thing', 'state');
  const result = await processRequest(state, event);
  await stateStore.set('thing', 'state', state);
  return result;
}
```

See *More Examples* for another example.

2.7.4 Unit testing your services

Use `StateStorageFake` to unit test your services that use `StateStorage`. The fake will store data in instance memory, instead of the AWS Parameter Store.

2.7.5 Type Declarations

```
/**
 * Service for storing state of other services.
 * Saved state can be fetched by any other execution of code in the AWS account,␣
↪region,
 * and environment (dev/prod).
 *
 * Suggested use with Injector:
 *   Injector.register(StateStorage, ()=>new StateStorage(process.env.STATE_STORAGE_
↪PREFIX));
 */
export declare class StateStorage {
    private readonly namePrefix;
    private readonly ssm;
    /**
     * Construct
     *
     * @param namePrefix prefix string to start all parameter names with.
     *                   Should at least include the environment (dev/prod).
     */
    constructor(namePrefix: string);
    /**
     * Save state for a later run.
     *
     * @param {string} service name of the service (class name?) that owns the state
     * @param {string} name name of the state variable to save
     * @param value content to save
     * @param quiet if true, don't log content
     * @returns {Promise<void>} completes upon success - failures shouldn't happen
     */
    set(service: string, name: string, value: any, quiet?: boolean): Promise<void>;
    /**
     * Fetch last state saved.
     *
     * @param {string} service name of the service (class name?) that owns the state
     * @param {string} name name of the state variable to fetch
     * @param quiet if true, don't log content
     * @returns {Promise<any>} completes with the saved value, or reject if not found
     */
```

(continues on next page)

(continued from previous page)

```
    get(service: string, name: string, quiet?: boolean): Promise<any>;
    protected generateName(service: string, name: string): string;
}
```

```
import { StateStorage } from "./state-storage";
/**
 * Version of StateStorage to use in unit testing.
 * This fake will store data in instance memory, instead of the AWS Parameter Store.
 */
export declare class StateStorageFake extends StateStorage {
    storage: {};
    constructor(namePrefix: string);
    set(service: string, name: string, value: any, quiet?: boolean): Promise<void>;
    get(service: string, name: string, quiet?: boolean): Promise<any>;
}
```

2.8 More Examples

This section includes some larger examples which use multiple packages.

2.8.1 Data Storage in Elasticsearch

Uses:

- *AwsHttps*
- *ElasticsearchClient*
- *Injector*
- *Logger*

```
import {AwsHttps} from "@sailplane/aws-https";
import {ElasticsearchClient} from "@sailplane/elasticsearch-client";
import {Injector} from "@sailplane/injector";
import {Logger} from "@sailplane/logger";
import {Ticket} from "./ticket";

const logger = new Logger('ticket-storage');
const ES_TICKET_PATH_PREFIX = "/ticket/local/";

// TODO: Ideally, put this in central place so it only runs once.
Injector.register(ElasticsearchClient, () => {
    const endpoint: string = process.env.ES_ENDPOINT!;
    logger.info('Connecting to Elasticsearch @ ' + endpoint);
    return new ElasticsearchClient(new AwsHttps(), endpoint);
});

/**
 * Storage of service tickets in Elasticsearch on AWS.
 */
export class TicketStorage {

    static readonly $inject = [ElasticsearchClient];
```

(continues on next page)

(continued from previous page)

```
    constructor(private readonly es: ElasticsearchClient) {
    }

    /**
     * Fetch a previously stored ticket by its ID
     * @param {string} id
     * @returns {Promise<Ticket>} if not found, returns undefined
     */
    get(id: string): Promise<Ticket> {
        return this.es.request('GET', ES_TICKET_PATH_PREFIX + id)
            .then((esDoc: ElasticsearchResult) => esDoc._source as Ticket);
    }

    /**
     * Store a ticket. Creates or replaces automatically.
     *
     * @param {Ticket} ticket
     * @returns {Promise<Ticket>} data stored (should match 'ticket')
     */
    put(ticket: Ticket): Promise<Ticket> {
        const path = ES_TICKET_PATH_PREFIX + ticket.id;
        return this.es.request('PUT', path, ticket)
            .then(() => ticket);
    }

    /**
     * Query for tickets that are not closed.
     *
     * @param {string} company
     * @param {number} maxResults Maximum number of results to return
     * @returns {Promise<Ticket[]>}
     * @throws Forbidden if no company value provided
     */
    queryOpen(company: string, maxResults: number): Promise<Ticket[]> {
        let query = {
            bool: {
                must_not: [
                    exists: {
                        field: "resolution"
                    }
                ]
            }
        };

        return this.es.request('GET', ES_TICKET_PATH_PREFIX + '_search', {
            size: maxResults,
            query: query
        })
        .then((esResults: ElasticsearchResult) => {
            if (esResults.timed_out) {
                throw new Error("Query of TicketStorage timed out");
            }
            else if (esResults.hits && esResults.hits.hits && esResults.hits.total) {
                return esResults.hits.hits.map(esDoc => esDoc._source as Ticket);
            }
            else {
                return [] as Ticket[];
```

(continues on next page)

(continued from previous page)

```
            }
        });
    }

}

Injector.register(TicketStorage);
```

2.8.2 Serverless Framework Lambda

This example shows how to:

- Configure Serverless Framework for use with *StateStorage*.
- Cache StateStorage result in *ExpiringValue*.
- Use *LambdaUtils* to simplify the lambda handler function.
- Do dependency injection with *Injector*.
- Make HTTPS request with *AwsHttps*. No SigV4 signature required on this use.
- Log status and objects via *Logger*.

```
# serverless.yml
service:
name: serverless-demo

plugins:
  - serverless-webpack
  - serverless-offline
  - serverless-plugin-export-endpoints

provider:
  name: aws
  runtime: nodejs8.10

  environment:
    STATE_STORAGE_PREFIX: /${opt:stage}/myapp

  iamRoleStatements:
    - Effect: Allow
      Action:
        - ssm:GetParameter
        - ssm:PutParameter
      Resource: "arn:aws:ssm:${opt:region}:*:parameter${self:provider.environment.
↪STATE_STORAGE_PREFIX}/*"

functions:
  getChatHistory:
    description: Retrieve some (more) history of the user's chat channel.
    handler: src/handlers.getChatHistory
    events:
      - http:
          method: get
          path: chat/history
          cors: true
```

(continues on next page)

(continued from previous page)

```
          request:
            parameters:
              querystrings:
                channel: true
                cursor: false
```

```
import 'source-map-support/register';
import {APIGatewayEvent} from 'aws-lambda';
import {Injector} from "@sailplane/injector";
import * as LambdaUtils from "@sailplane/lambda-utils";
import {ChatService} from "./chat-service";
import * as createHttpError from "http-errors";

Injector.register(StateStorage, () => new StateStorage(process.env.STATE_STORAGE_
↪PREFIX));

/**
 * Fetch history of chat on the user's channel
 */
export const getChatHistory = LambdaUtils.wrapApiHandler(async (event: LambdaUtils.
↪APIGatewayProxyEvent) => {
    const channel = event.queryStringParameters.channel;
    const cursor = event.queryStringParameters.cursor;

    return Injector.get(ChatService)!.getHistory(channel, cursor);
});
```

```
// chat-service.ts
import {AwsHttps} from "@sailplane/aws-https";
import {ExpiringValue} from "@sailplane/expiring-value";
import {Injector} from "@sailplane/injector";
import {Logger} from "@sailplane/logger";
import {URL} from "url";
import * as createHttpError from "http-errors";

const logger = new Logger('chat-service');

const CONFIG_REFRESH_PERIOD = 15*60*1000; // 15 minutes

//// Define Data Structures
interface ChatConfig {
    url: string;
    authToken: string;
}

interface ChatMessage {
    from: string;
    when: number;
    text: string;
}

interface ChatHistory {
    messages: ChatMessage[];
    cursor: string;
}
```

(continues on next page)

(continued from previous page)

```
/**
 * Service to interface with the external chat provider.
 */
export class ChatService {
    private config = new ExpiringValue<ChatConfig>(
            () => this.stateStorage.get('ChatService', 'config') as ChatConfig,
            CONFIG_REFRESH_PERIOD);
    private readonly awsHttps = new AwsHttps();

    /** Construct */
    constructor(private readonly stateStorage: StateStorage) {
    }

    /**
     * Fetch history of a chat channel.
     */
    async getHistory(channelId: string, cursor?: string): Promise<ChatHistory> {
        logger.debug(`getHistory(${channelId}, ${cursor})`);
        const config = await this.config.get();

        // Fetch history from external chat provider
        let options = this.awsHttp.buildOptions('POST' new URL(config.url));
        options.headers = { authorization: 'TOKEN ' + config.authToken };
        options.body = JSON.stringify({
            channel: channelId
            cursor: cursor
        });

        const response = await this.awsHttp.request(options);

        // Check for error
        if (!response.ok) {
            logger.infoObject("External history request returned error: ", response);
            throw new createHttpError.InternalServerError(response.error);
        }

        // Prepare results
        const history: ChatHistory =  {
            messages: [],
            cursor: response.next_cursor
        };

        // Process each message
        for (let msg of response.messages) {
            history.messages.push({
                from: msg.username,
                when: msg.ts
                text: msg.text
            });
        }

        return history;
    }
}

Injector.register(ChatService, [StateStorage]);
```

2.9 Apache License

Version 2.0

Date January 2004

URL http://www.apache.org/licenses/

2.9.1 TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means *(i)* the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or *(ii)* ownership of fifty percent (50%) or more of the outstanding shares, or *(iii)* beneficial ownership of such entity.

"You" (or **"Your"**) shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution.

You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

- You must give any other recipients of the Work or Derivative Works a copy of this License; and
- You must cause any modified files to carry prominent notices stating that You changed the files; and
- You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
- If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions.

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks.

This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty.

Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an **"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND**, either express or implied, including, without limitation, any warranties or conditions of **TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE**. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability.

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability.

While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

> Copyright 2018 Onica Group LLC
>
> Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at
>
> > http://www.apache.org/licenses/LICENSE-2.0
>
> Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER 3

Why Sailplane?

Onica's early OSS releases have had aviation themed names; this may or may not have something to do with the CTO being a pilot. Nobody really knows.

Sailplane was selected for this *serverless* toolset by considering that serverless is to computing without the complexities of a server, as a sailplane is to flight without the complexities of an airplane.

And that's it. Also, the NPM scope was available.