
Sailplane Documentation

Release 1.0.0

Onica Group

Feb 28, 2023

Contents

1	What is this?	3
2	Content	5
2.1	AwsHttps	5
2.2	ElasticsearchClient	9
2.3	ExpiringValue	11
2.4	Injector	12
2.5	LambdaUtils	22
2.6	Logger	27
2.7	StateStorage	33
2.8	More Examples	36
2.9	Apache License	41
3	Why Sailplane?	45



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 12+ 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](#).

Each utility is described on its own page:

- *AwsHttps* - *HTTPS client with AWS Signature v4*
- *ElasticsearchClient* - *Communicate with AWS Elasticsearch*
- *ExpiringValue* - *Value that is instantiated on-demand and cached for a limited time*
- *Injector* - *Light-weight and type-safe Dependency Injection*
- *LambdaUtils* - *Lambda handler middleware*
- *Logger* - *CloudWatch and serverless-offline friendly logger*
- *StateStorage* - *Serverless state and configuration storage*
- *More Examples*
- *License*

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 the container's AWS 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)

```

    }
}

```

Example hitting API with the custom AWS credentials:

```

// Call my helper function to get credentials with AWS.STS
const roleCredentials = await this.getAssumeRoleCredentials();

const awsCredentials = {
  accessKey: roleCredentials.AccessKeyId,
  secretKey: roleCredentials.SecretAccessKey,
  sessionToken: roleCredentials.SessionToken,
};
const http = new AwsHttps(false, awsCredentials);

// Build request options from a method and URL
const url = new URL('https://www.onica.com/ping.json');
const options = http.buildOptions('GET' url);

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

```

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 { Credentials, CredentialsOptions } from "aws-sdk/lib/credentials";
import * as https from "https";
import { URL } from "url";
/**
 * Same options as https://nodejs.org/api/http.html#http_request_options_callback

```

(continues on next page)

(continued from previous page)

```

* 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 credentialsInitializedPromise;
  /** Credentials to use in this instance */
  private awsCredentials?;
  /**
   * Constructor.
   * @param verbose true to log everything, false for silence,
   *          undefined (default) for normal logging.
   * @param credentials
   *          If not defined, credentials will be obtained by default SDK behavior for
↳the runtime environment.
   *          This happens once and then is cached; good for Lambdas.
   *          If `true`, clear cached to obtain fresh credentials from SDK.
   *          Good for longer running containers that rotate credentials.
   *          If an object with accessKeyId, secretAccessKey, and sessionToken,
   *          use these credentials for this instance.
   */
  constructor(verbose?: boolean | undefined, credentials?: boolean | Credentials |
↳CredentialsOptions);
  /**
   * 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

```

(continues on next page)

(continued from previous page)

```

    * @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 {

```

(continues on next page)

```

    _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.
 *
 * 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

```

(continues on next page)

(continued from previous page)

```

    * @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));

```

(continues on next page)

```
    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.
   */
  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.

As of v3, Injector also supports a Typescript decorator for registering classes.

Injector depends on one other utility to work:

- *Logger*

2.4.2 Install

```
npm install @sailplane/injector @sailplane/logger bottlejs@1.7
```

2.4.3 Configuration

To use the Typescript decorator, these options must be enabled in `tsconfig.json`:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

If using `esbuild`, a plugin such as `esbuild-decorators` is necessary.

2.4.4 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 without decorator:

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

export class MyService {
}

Injector.register(MyService);

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

Example with decorator:

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

@Injectable()
export class 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 without decorator:

```
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']);
```

Example with decorator:

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

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

@Injectable({dependencies: [MyHelper, 'stage']})
export class MyService {
  constructor(private readonly helper: MyHelper,
              private readonly stage: string) {
  }
}
```

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);
```

(continues on next page)

(continued from previous page)

```

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 without decorator:

```

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!));

```

Example with decorator:

```

import {Injector, Injectable} from "@sailplane/injector";

@Injectable()
export class MyHelper {
}

@Injectable({factory: () => new MyService(Injector.get(MyHelper)!, process.env.STAGE!
↪)})
export class MyService {
  constructor(private readonly helper: MyHelper,
              private readonly stage: string) {
  }
}

```

Register a child class that implements a parent

A common dependency injection pattern is to invent code dependencies by having business logic define an interface it needs to talk to via an abstract class, and then elsewhere have a child class define the actual behavior. Runtime options could even choose between implementations.

Here's the business logic code:

```
abstract class SpecialDataRepository {
  abstract get(id: string): Promise<SpecialData>;
}

class SpecialBizLogicService {
  constructor(dataRepo: SpecialDataRepository) {}
  public async calculate(id: string) {
    const data = await this.dataRepo.get(id);
    // do stuff
  }
}
Injector.register(SpecialBizLogicService); // Could use @Injectable() instead
```

Without decorators, we use `Injector.register<T>(className, factory: ()=>T)` to register the implementing repository, which could be done conditionally:

Example without decorator:

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

export class LocalDataRepository extends SpecialDataRepository {
  async get(id: string): Promise<SpecialData> {
    // implementation ....
  }
}

export class RemoteDataRepository extends SpecialDataRepository {
  async get(id: string): Promise<SpecialData> {
    // implementation ....
  }
}

const isLocal = !!process.env.SHELL;
Injector.register(
  MyService,
  () => isLocal ? new LocalDataRepository() : new RemoteDataRepository()
);
```

Example with decorator (can't be conditional):

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

@Injectable({as: SpecialDataRepository})
export class RemoteDataRepository extends SpecialDataRepository {
  async get(id: string): Promise<SpecialData> {
    // implementation ....
  }
}
```

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 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. (See previous example using an abstract base class for a more type-safe approach.)

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, conditionally and no decorator

```
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');
  }
}

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 foobarSvc: FoobarService) {}
}

Injector.register(MyService);
```

Example: Inject an interface implementation with the decorator

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

export interface FoobarService {
  doSomething(): void;
}

@Injectable({as: "FoobarService"})
export class FoobarServiceImpl implements FoobarService {
  constructor(private readonly stateStorage: StateStorage) {}

  doSomething(): void {
    // code
  }
}

// Elsewhere...

@Injectable({dependencies: ['FoobarService']}) // Note: This is a string!
export class MyService {
  constructor(private readonly foobarSvc: 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.

2.4.5 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 "reflect-metadata";
import * as Bottle from 'bottlejs';
declare type InjectableClass<T> = {
  new (...args: any[]): T;
  $inject?: DependencyList;
  name: string;
};
declare type GettableClass<T> = Function & {
  prototype: T;
  name: string;
};
declare type DependencyList = (InjectableClass<unknown> | 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.
   * @deprecated
   */
  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));

```

(continues on next page)

(continued from previous page)

```

*
* @param clazz the class to register. Ex: MyClass. NOT an instance, the class.
* @param factoryOrDependencies see above examples. Optional.
* @param asName by default is clazz.name, but may specify another name to
↳register as
* @throws TypeError if duplicate or bad request
*/
static register<T>(clazz: InjectableClass<T>, factoryOrDependencies?: (() => T) |
↳DependencyList, asName?: string): void;
/**
* 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.
* @throws TypeError if duplicate or bad request
*/
static registerFactory<T>(name: string, factory: (() => T)): void;
/**
* 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.
* @throws TypeError if duplicate or bad request
*/
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: GettableClass<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;
/**
* Is a class, factory, or constant registered?
* Unlike #getByName, will not instantiate if registered but not yet lazy created.
*
* @param clazzOrName class or name of factory or constant
*/
static isRegistered(clazzOrName: InjectableClass<unknown> | GettableClass<unknown>
↳ | string): boolean;

```

(continues on next page)

(continued from previous page)

```

}
/** Options for Injectable decorator */
export interface InjectableOptions<T> {
  /**
   * Register "as" this parent class or name.
   * A class *must* be a parent class.
   * The name string works for interfaces, but lacks type safety.
   */
  as?: GettableClass<unknown> | string;
  /** Don't auto-detect constructor dependencies - use this factory function,
  ↪instead */
  factory?: () => T;
  /** Don't auto-detect constructor dependencies - use these instead */
  dependencies?: DependencyList;
}
/**
 * Typescript Decorator for registering classes for injection.
 *
 * Must enable options in tsconfig.json:
 * {
 *   "compilerOptions": {
 *     "experimentalDecorators": true,
 *     "emitDecoratorMetadata": true
 *   }
 * }
 *
 * Usage:
 *
 * Like Injector.register(MyServiceClass, [Dependency1, Dependency2])
 * @Injectable()
 * class MyServiceClass {
 *   constructor(one: Dependency1, two: Dependency2) {}
 * }
 *
 * Like Injector.register(MyServiceClass, [Dependency1, "registered-constant"])
 * @Injectable({dependencies:[Dependency1, "registered-constant"]})
 * class MyServiceClass {
 *   constructor(one: Dependency1, two: string) {}
 * }
 *
 * Like Injector.register(MyServiceClass, () = new MyServiceClass())
 * @Injectable({factory: () = new MyServiceClass()})
 * class MyServiceClass {
 * }
 *
 * Like Injector.register(HexagonalPort, () => new HexagonalAdaptor())
 * abstract class HexagonalPort {
 *   abstract getThing(): string;
 * }
 * @Injectable({as: HexagonalPort })
 * class HexagonalAdaptor extends HexagonalPort {
 *   getThing() { return "thing"; }
 * }
 */
export declare function Injectable<T>(options?: InjectableOptions<T>): (target: ↪
↪InjectableClass<unknown>) => void;
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, its Typescript declarations are overly permissive, and you still have to format your response in the shape required by API Gateway.

`LambdaUtils` takes Middy further and is extendable so that you can add your own middleware (ex: authentication & authorization) on top of it.

Used with API Gateway v1 (REST API) and v2 (HTTP API), the included middlewares are:

- 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 `TypeError`s.
- **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`!
- Registers Lambda context with Sailplane's *logger* `<logger>` for structured logging. (Detail below.)
- Fully leverages Typescript and `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!

`LambdaUtils` depends on two other utilities to work:

- [Logger](#)
- [Middy](#)

2.5.2 Install

To use `LambdaUtils` v4.x with Middy v2.x.x (latest):

```
npm install @sailplane/lambda-utils@4 @sailplane/logger @middy/core@2 @middy/http-  
→cors@2 @middy/http-event-normalizer@2 @middy/http-header-normalizer@2 @middy/http-  
→json-body-parser@2
```

The extra `@middy/` middleware packages are optional if you write your own wrapper function that does not use them. See below.

To use LambdaUtils v3.x with Middy v1.x.x:

```
npm install @sailplane/lambda-utils@3 @sailplane/logger @middy/core@1 @middy/http-
↪cors@1 @middy/http-event-normalizer@1 @middy/http-header-normalizer@1 @middy/http-
↪json-body-parser@1
```

The extra `@middy/` middleware packages are optional if you write your own wrapper function that does not use them. See below.

To use LambdaUtils v2.x with Middy v0.x.x:

```
npm install @sailplane/lambda-utils@2 @sailplane/logger middy@0
```

2.5.3 Upgrading

To upgrade from older versions of `lambda-utils`, remove the old `lambda-utils` and `middy` dependencies and then follow the install instructions above to install the latest. See also the [Middy upgrade instructions](#).

2.5.4 Structured Logging Attributes

When *structured logging* is enabled, `LambdaUtils`'s `wrapApiHandlerV1` and `wrapApiHandlerV2` include the `loggerContextMiddleware`, which calls `Logger.setLambdaContext` for you and also adds the following properties:

- `api_request_id` - the request ID from AWS API Gateway
- `jwt_sub` - JWT (including Cognito) authenticated subject of the request

2.5.5 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.wrapApiHandlerV2(async (event: LambdaUtils.
↪APIGatewayProxyEvent) => {
  // These event objects are now always defined, so don't need to check for_
↪undefined.
  const who = event.pathParameters.who;
  let points = Number(event.queryStringParameters.points || '0');

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

    return {message};
  }
  else {
```

(continues on next page)

(continued from previous page)

```

    // 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 middy from "@middy/core";
import * as createError from "http-errors";
import * as LambdaUtils from "@sailplane/lambda-utils";

/** ID user user authenticated in running Lambda */
let authenticatedUserId: string|undefined;

export getAuthenticatedUserId(): string|undefined {
  return authenticatedUserId;
}

/**
 * Middleware for LambdaUtils to automatically manage AuthService context.
 */
const authMiddleware = (requiredRole?: string): Required<middy.MiddlewareObj> => {
  return {
    before: async (request) => {
      const claims = request.event.requestContext.authorizer?.claims;

      const role = claims['custom:role'];
      if (requiredRole && role !== requiredRole) {
        throw new createError.Forbidden();
      }

      authenticatedUserId = claims?.sub;
      if (!authenticatedUserId) {
        throw new createError.Unauthorized("No user authorized");
      }
    },
    after: async (_) => {
      authenticatedUserId = undefined;
    },
    onError: async (_) => {
      authenticatedUserId = undefined;
    }
  };
}

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

export function wrapApiHandlerWithAuth(
  options: WrapApiHandlerOptions,

```

(continues on next page)

(continued from previous page)

```

    handler: LambdaUtils.AsyncProxyHandlerV2
  ): LambdaUtils.AsyncMiddyifiedHandlerV2 {
    const wrap = LambdaUtils.wrapApiHandlerV2(handler);

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

    return wrap;
  }

```

2.5.6 Type Declarations

```

import { APIGatewayProxyResult } from "aws-lambda";
import { AsyncMiddyifiedHandlerV1, AsyncMiddyifiedHandlerV2, AsyncProxyHandlerV1, ↳
  ↳AsyncProxyHandlerV2 } from "./types";
/**
 * Wrap an API Gateway V1 format proxy lambda function handler to add features:
 * - Set CORS headers.
 * - Normalize incoming headers to lowercase
 * - 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.
 * - Catch http-errors exceptions into proper HTTP responses.
 * - Catch other exceptions and return as HTTP 500
 * - Set Lambda invocation and API request context in @sailplane/logger
 *
 * 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/#:~:text=available%20middlewares
 * @see https://www.npmjs.com/package/http-errors
 */
export declare function wrapApiHandler(handler: AsyncProxyHandlerV1): ↳
  ↳AsyncMiddyifiedHandlerV1;
export declare const wrapApiHandlerV1: typeof wrapApiHandler;
/**
 * Wrap an API Gateway V2 format proxy lambda function handler to add features:
 * - Set CORS headers.
 * - Normalize incoming headers to lowercase
 * - 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.
 * - Catch http-errors exceptions into proper HTTP responses.
 * - Catch other exceptions and return as HTTP 500
 * - Set Lambda invocation and API request context in @sailplane/logger
 *
 * This wrapper includes commonly useful middleware. You may further wrap it
 * with your own function that adds additional middleware, or just use it as

```

(continues on next page)

(continued from previous page)

```

* an example.
*
* @param handler async function to wrap
* @see https://middy.js.org/#:~:text=available%20middlewares
* @see https://www.npmjs.com/package/http-errors
*/
export declare function wrapApiHandlerV2(handler: AsyncProxyHandlerV2):
↳ AsyncMiddyifiedHandlerV2;
/**
* Construct the object that API Gateway payload format v1 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 {APIGatewayProxyResult}
*/
export declare function apiSuccess(result?: any): APIGatewayProxyResult;
/**
* Construct the object that API Gateway payload format v1 wants back upon a failed
↳ run.
*
* Often, it is simpler to throw a 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 {APIGatewayProxyResult}
*/
export declare function apiFailure(statusCode: number, message?: string):
↳ APIGatewayProxyResult;

```

```

import { APIGatewayProxyEvent as AWS_APIGatewayProxyEvent, APIGatewayProxyEventV2 as
↳ AWS_APIGatewayProxyEventV2, APIGatewayProxyResult,
↳ APIGatewayProxyStructuredResultV2, Callback, Context } from "aws-lambda";
import middy from "@middy/core";
/**
* 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
  */

```

(continues on next page)

(continued from previous page)

```

queryStringParameters: {
    [name: string]: string;
};
}
export declare type APIGatewayProxyEventV1 = APIGatewayProxyEvent;
/**
 * Casted interface for APIGatewayProxyEventsV2 as converted through the middleware
 */
export interface APIGatewayProxyEventV2 extends AWS_APIGatewayProxyEventV2 {
    /**
     * 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;
    };
}
export declare type APIGatewayProxyEventAnyVersion = AWS_APIGatewayProxyEvent |
↳APIGatewayProxyEvent | AWS_APIGatewayProxyEventV2 | APIGatewayProxyEventV2;
export declare type APIGatewayProxyResultAnyVersion = APIGatewayProxyResult |
↳APIGatewayProxyStructuredResultV2;
/** LambdaUtils version of ProxyHandler for API Gateway v1 payload format */
export declare type AsyncProxyHandlerV1 = (event: APIGatewayProxyEvent, context:
↳Context, callback?: Callback<APIGatewayProxyResult>) => Promise
↳<APIGatewayProxyResult | object | void>;
/** LambdaUtils version of an API Gateway v1 payload handler wrapped with middy */
export declare type AsyncMiddyifiedHandlerV1 = middy.MiddyifiedHandler<AWS_
↳APIGatewayProxyEvent, APIGatewayProxyResult | object | void>;
/** LambdaUtils version of ProxyHandler for API Gateway v2 payload format */
export declare type AsyncProxyHandlerV2 = (event: APIGatewayProxyEventV2, context:
↳Context, callback?: Callback<APIGatewayProxyStructuredResultV2>) => Promise
↳<APIGatewayProxyStructuredResultV2 | object | void>;
/** LambdaUtils version of an API Gateway v2 payload handler wrapped with middy */
export declare type AsyncMiddyifiedHandlerV2 = middy.MiddyifiedHandler<AWS_
↳APIGatewayProxyEventV2, APIGatewayProxyStructuredResultV2 | object | void>;

```

2.6 Logger

CloudWatch, web browser, and local development friendly logger with optional structured logging, while still small and easy to use.

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, object formatting, and optional structured logging. It's just a few incremental improvements, and yet together takes logging a leap forward.

If you are transpiling, be sure to enable source maps (in `Typescript`, `Babel`) and the enable use via the `source-map-support` library or `Node.js` `enable-source-maps` option so that you get meaningful stack traces.

2.6.2 Install

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

2.6.3 Examples

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

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

Logger.initialize({level: LogLevel.INFO});
logger.debug("DEBUG < INFO.");
// No output

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

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

Logger.initialize({format: "PRETTY"});
logger.error("Exception ", {message: "oops"});
// 2018-11-15T18:30:49 INFO name-of-module: Exception {
//   message: "oops"
// }

Logger.initialize({
  format: "STRUCT",
  attributes: { my_trace_id: request.id }
});
logger.error("Processing Failed", new Error("Unreachable"));
// {
//   "aws_region": "us-east-1",
//   "function_name": "myDataProcessor",
//   "function_version": "42",
//   "invocation_num": 1,
//   "my_trace_id": "ebfb6f2f-8f2f-4e2e-a0a9-4495e90a4316",
//   "stage": "prod",
//   "level": "ERROR",
//   "module": "name-of-module",
//   "timestamp": "2022-03-03T17:32:19.830Z",
//   "message": "Processing Failed",
```

(continues on next page)

(continued from previous page)

```
//  "value": {
//    "name": "Error",
//    "message": "Unreachable",
//    "stack": "Error: Unreachable\n  at /home/adam/my-project/src/service/processor.
→service.ts:83\n  at ..."
//    "source": "/home/adam/my-project/src/service/processor.service.ts:83"
//  }
// }
```

2.6.4 Configuration / Behavior

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

Default behavior should work for Lambdas. If you are using Logger in another container (EC2, Fargate, ...) you likely will want to adjust these settings.

CloudWatch detection

The default behaviors of some configuration change depending on whether log output is going to CloudWatch vs local console. This is because within the AWS Lambda service, logging to stdout is automatically prefixed with the log level and timestamp. Local console does not. So Logger adds these for you when a login shell (offline mode) is detected. You can force CloudWatch logging behavior via the environment variable `export LOG_TO_CLOUDWATCH=true` or `export LOG_TO_CLOUDWATCH=false`

Note: The above is ignored when using structured logging.

Recommendations

The best logging format often depends on the environment/stage. It may be selected via the `LOG_FORMAT` environment variable.

For local development, the default format is `PRETTY` and this is usually the most readable in a terminal window.

For deployment to AWS in lower stage environments (ex: dev), the default is `FLAT` and is recommended. When viewed in CloudWatch AWS Console, it will handle pretty printing JSON output when a line is expanded.

For higher environments (ex: production), the default is still `FLAT` but `STRUCT` is suggested to allow for analysis of massive log content, when using [Amazon CloudWatch Logs Insights](#) or an [ELK stack](#).

2.6.5 Structured Logging Attributes

When structured logging is used, the following properties are included:

- `aws_region` - the AWS region name
- `aws_request_id` - the AWS generated unique identifier of the request
- `xray_trace_id` - AWS X-Ray trace ID (if available)
- `function_name` - name of the logging AWS Lambda
- `function_memory_size` - the capacity configuration of the Lambda, in memory megabytes

- `invocation_num` - count of invocations of the Lambda handler since cold start (1 = first request since cold start)
- `level` - logging level
- `module` - source module of the Logger instance
- `timestamp` - ISO8601 date-time in UTC when the line was logged
- `message` - text message of the line (first parameter to log function)
- `value` - if only two parameters are given to the log function, this is the second one
- `params` - when more than two parameters are given, all after the message are in this array

Sailplane's *LambdaUtils* adds additional properties.

2.6.6 Type Declarations

```

export declare enum LogLevel {
  NONE = 1,
  ERROR = 2,
  WARN = 3,
  INFO = 4,
  DEBUG = 5
}
export declare enum LogFormat {
  FLAT = 1,
  PRETTY = 2,
  STRUCT = 3
}
export declare type LoggerAttributes = Record<string, string | number>;
/**
 * Signature of a Formatter function.
 * @param loggerConfig configuration of Logger instance
 * @param globalConfig global configuration
 * @param level logging level
 * @param message text to log
 * @param params A list of JavaScript objects to output.
 * @return array to pass to a console function
 */
export declare type FormatterFn = (loggerConfig: LoggerConfig, globalConfig:
↳LoggerConfig, level: LogLevel, message: string, params: any[]) => any[];
/**
 * Configuration of a Logger.
 * See individual properties for details.
 * The default behavior of some vary based on runtime environment.
 * Some properties may be initialized via environment variables.
 * Configuration for all loggers may be set via the Logger.initialize(config)
↳function.
 * Overrides for individual Loggers may be given in the constructor.
 */
export interface LoggerConfig {
  /**
   * Source module of the logger - prepended to each line.
   * Source file name or class name are good choices, but can be any label.
   */
  module: string;
}

```

(continues on next page)

(continued from previous page)

```

    * Enabled logging level.
    * May be initialized via LOG_LEVEL environment variable.
    */
    level: LogLevel;
    /** Any additional context attributes to include with _structured_ format (only).
    ↪ */
    attributes?: LoggerAttributes;
    /**
    * Include the level in log output?
    * Defaults to true if not streaming to CloudWatch;
    * always included with _structured_ format.
    */
    outputLevels: boolean;
    /**
    * Include timestamps in log output?
    * Defaults to false if streaming to CloudWatch (CloudWatch provides timestamps.),
    * true otherwise; always included with _structured_ format.
    * May override by setting the LOG_TIMESTAMPS environment variable to 'true' or
    ↪ 'false'.
    */
    logTimestamps: boolean;
    /**
    * Output format to use.
    * Defaults to FLAT if streaming to CloudWatch, PRETTY otherwise.
    * (Best to let CloudWatch provide its own pretty formatting.)
    * May initialize by setting the LOG_FORMAT environment variable to
    * "FLAT", "PRETTY", or "STRUCT".
    */
    format: LogFormat;
    /**
    * Function use to format output. Set based on format property, but may
    * be programmatically set instead.
    */
    formatter: FormatterFn;
}

```

```

import { LoggerAttributes, LoggerConfig, LogLevel } from "../common";
import { Context } from "aws-lambda";
/**
 * Custom logger class.
 *
 * Works much like console's logging, but includes levels, date/time,
 * and module (file) on each line, or structured formatting if configured to do so.
 *
 * Usage:
 *   import {Logger} from "@sailplane/logger";
 *   const logger = new Logger('name-of-module');
 *   logger.info("Hello World!");
 */
export declare class Logger {
    /**
    * Configure global defaults. Individual Logger instances may override.
    * @param globalConfig configuration properties to changed - undefined properties
    * will retain existing value
    */
    static initialize(globalConfig: Partial<LoggerConfig>): void;
}

```

(continues on next page)

(continued from previous page)

```

/**
 * Set some context attributes to the existing collection of global attributes
 * Use initialize({attributes: {}}) to override/reset all attributes.
 */
static addAttributes(attributes: LoggerAttributes): void;
/**
 * Set structured logging global attributes based on Lambda Context:
 *
 * - aws_request_id: identifier of the invocation request
 * - invocation_num: number of invocations of this process (1 = cold start)
 *
 * Call this every time the Lambda handler begins.
 */
static setLambdaContext(context: Context): void;
private readonly config;
/**
 * Construct.
 * @param ops LoggerConfig, or just module name as string
 */
constructor(ops: string | Partial<LoggerConfig>);
/**
 * The Log Level of this Logger
 */
get level(): LogLevel;
/**
 * Change the log level of this Logger
 */
set level(level: LogLevel);
/**
 * Log an item at given level.
 * Usually better to use the specific function per log level instead.
 *
 * @param level log level
 * @param message text to log
 * @param params A list of JavaScript objects to output
 */
log(level: LogLevel, message: string, params: any[]): void;
/**
 * Log a line at DEBUG level.
 *
 * @param message text to log
 * @param optionalParams A list of JavaScript objects to output.
 */
debug(message: string, ...optionalParams: any[]): void;
/**
 * Log a line at INFO level.
 *
 * @param message text to log
 * @param optionalParams A list of JavaScript objects to output.
 */
info(message: string, ...optionalParams: any[]): void;
/**
 * Log a line at WARN level.
 *
 * @param message text to log
 * @param optionalParams A list of JavaScript objects to output.
 */

```

(continues on next page)

(continued from previous page)

```

warn(message: string, ...optionalParams: any[]): void;
/**
 * Log a line at ERROR level.
 *
 * @param message text or Error instance
 * @param optionalParams A list of JavaScript objects to output.
 */
error(message: string | Error, ...optionalParams: any[]): void;
/**
 * Log a line at DEBUG level with a stringified object.
 *
 * @param message text to log
 * @param object a Javascript object to output
 * @deprecated #debug has the same result now
 */
debugObject(message: string, object: any): void;
/**
 * Log a line at INFO level with a stringified object.
 *
 * @param message text to log
 * @param object a Javascript object to output
 * @deprecated #info has the same result now
 */
infoObject(message: string, object: any): void;
/**
 * Log a line at WARN level with a stringified object.
 *
 * @param message text to log
 * @param object a Javascript object to output
 * @deprecated #warn has the same result now
 */
warnObject(message: string, object: any): void;
/**
 * Log a line at ERROR level with a stringified object.
 *
 * @param message text to log
 * @param object a Javascript object to output
 * @deprecated #error has the same result now
 */
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 @aws-sdk/client-ssm
```

2.7.3 Examples

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

```
provider:
  name: aws

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/*"

  - Effect: Allow
    Action:
      - kms:Decrypt
      - kms:Encrypt
    Resource: "arn:aws:kms:${opt:region}:*:alias/aws/ssm"
    Condition:
      StringEquals:
        "kms:EncryptionContext:PARAMETER_ARN": "arn:aws:ssm:${opt:region}
↪*:parameter:${self:provider.environment.STATE_STORAGE_PREFIX}/*"
```

Note that this is the complete set of possible permissions. Not all are needed if only reading parameters or if not using the `secure` option.

Simple example storing state

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

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

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

(continues on next page)

(continued from previous page)

```

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

```

import { SSMClient } from "@aws-sdk/client-ssm";
interface StateStorageOptions {
  /** If true, do not log values. */
  quiet?: boolean;
  /**
   * If true, store as encrypted or decrypt on get. Uses account default KMS key.
   * Implies quiet as well.
   */
  secure?: boolean;
  /**
   * If set, set and get the value as is, not JSON. (Only works for string values.)
   */
  isRaw?: boolean;
}
/**
 * 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).
   * @param ssm the SSMClient to use
   */
  constructor(namePrefix: string, ssm?: SSMClient);
  /**
   * Save state for a later run.
   *
   * @param {string} serviceName name of the service (class name?) that owns the state
   * @param {string} stateName name of the state variable to save

```

(continues on next page)

(continued from previous page)

```

    * @param value content to save
    * @param optionsOrQuiet a StateStorageOptions, or if true sets quiet option.
↳ (For backward compatibility.)
    * @returns {Promise<void>} completes upon success - rejects if lacking
↳ ssm:PutParameter permission
    */
    set(service: string, name: string, value: any, optionsOrQuiet?: boolean |
↳ StateStorageOptions): 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 optionsOrQuiet a StateStorageOptions, or if true sets quiet option.
↳ (For backward compatibility.)
    * @returns {Promise<any>} completes with the saved value, or reject if not found
↳ or lacking ssm:GetParameter permission
    */
    get(service: string, name: string, optionsOrQuiet?: boolean |
↳ StateStorageOptions): Promise<any>;
    protected generateName(service: string, name: string): string;
}
export {};

```

```

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, options: any): Promise<void>;
    get(service: string, name: string, options: any): 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";

```

(continues on next page)

(continued from previous page)

```

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];
  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: [

```

(continues on next page)

(continued from previous page)

```

                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[];
        }
    });
}

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

```

(continues on next page)

(continued from previous page)

```

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
          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');

```

(continues on next page)

(continued from previous page)

```

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;
}

/**
 * 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.info("External history request returned error: ", response);
      throw new createHttpError.InternalServerError(response.error);
    }

    // Prepare results

```

(continues on next page)

(continued from previous page)

```
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.