abstract-things Documentation

Release

AH

Jan 23, 2018

Getting started

abstract-things is a JavaScript library that provides a simple base for building libraries that interact with physical things, such as IoT-devices, and virtual things.

This library provides a base class named *Thing* that supports mixins of various types. Things are described using two types of tags, one describing the type of the thing and one describing its capabilities. Things are also expected to describe their public API, to make remote use easier.

Types and capabilities are designed to be stable and to be combined. When combined they describe a thing and what it can do.

Note: This documentation is a work in progress. Things are missing and may sometimes be inaccurate. Please open issues on Github if you find something that seems wrong.

CHAPTER 1

Using things

Things provide a basic shared API no matter their types and capabilities. The `matches` method can be used to match tags and to figure out what a thing is and what it can do:

```
if(thing.matches('cap:colorable')) {
  console.log('Current color:', thing.color());
}
```

Events are one of the most important parts of things and listeners can be added via the `on` method:

```
thing.on('colorChanged', color => console.log('The color has changed'));

// Listeners receive the thing as the second argument
const handler = (color, thing) => console.log('Color is now', color, 'for thing',␣
↪thing);
thing1.on('colorChanged', handler);
thing2.on('colorChanged', handler);
```

1.1 Thing API

id

The unique identifier of the thing as a string. The identifier should be globally unique and contain a namespace.

Example:

```
console.log(thing.id);
```

Example of identifiers:

- `hue:000b57fffe0eee95-01`
- `miio:55409498`
- `uuid:8125606b-7b57-405b-94d6-e5720c44aa6a`

- `space:global`

See *Naming of identifiers, types and capabilities* for more details.

`metadata`
Metadata associated with the thing. Contains information about types and capabilities.

Example:

```
console.log(thing.metadata);
console.log(thing.metadata.tags);
console.log(thing.metadata.types);
console.log(thing.metadata.capabilities);
```

`matches`(*...tags*)
Check if a thing matches a set of tags. Tags are created by the types and capabilities of the thing.

Arguments

- **`...tags`** – Set of tags that the thing should have.

Returns Boolean indicating if the thing has the given tags.

Example:

```
if(thing.matches('type:light', 'cap:switchable-power')) {
  // Thing is of type light and has the switchable-power capability
}
```

`on`(*eventName*, *listener*)
Register a listener for the given event. The listener will be invoked when the thing emits the event. The listener will receive two arguments, the first being the value of the event (or null) and the second being a reference to the Thing that emitted the event.

Arguments

- **`eventName`** (*`string`*) – The name of the event to listen for.
- **`listener`** (*`function`*) – Function that will be invoked when the event is emitted.

Example:

```
thing.on('stateChanged', (change, thing) =>
  console.log(thing, 'changed state:', change)
);
```

`off`(*eventName*, *listener*)
Remove a listener for the given event. The listener must have been previously registered via *`on()`*.

Arguments

- **`eventName`** (*`string`*) – The name of the event that the listener was registered for.
- **`listener`** (*`function`*) – Function that was used when registering the listener.

`init`()
Initialize the thing. Most commonly used when creating a new thing. Many libraries provide already initalized things via their main discovery or creation function.

Returns Promise that resolves to the instance being initalized.

```
thing.init()
  .then(thing => /* do something with the thing */)
  .catch(/* handle error */);
```

destroy()

Destroy the thing. Should be called whenever the thing is no longer needed.

Returns Promise that resolves to the instance being destroyed.

```
thing.destroy()
  .then(thing => /* do something with the thing */)
  .catch(/* handle error */);
```

1.2 Remote API

When a thing is exposed via a remote API, such as in Tinkerhub, it extends the above API with the addition that actions (and properties) return promises.

Example:

```
// Properties are now functions that return promises:
thing.state()
  .then(result => console.log('Invoked state and got', state))
  .catch(err => console.log('Error occurred:', err);

// async/await can be used with actions:
const power = await thing.power(false);

// The base API still works as before:
console.log(thing.id);
thing.on('stateChanged', change => console.log(change));
```

CHAPTER 2

Building things

Things are built by extending `Thing` with a combination of types and capabilities. The first step is to make sure that the project has acccess to `abstract-things`:

```
$ npm install abstract-things
```

It is recommended to target at least Node 8 to make use of `async` and `await`. It will make handling the asynchronous nature of API calls easier.

The smallest possible thing simply extends `Thing`:

```
const { Thing } = require('abstract-things');

class ExampleThing extends Thing {
  constructor(id) {
    super();

    // Identifier is required to be set
    this.id = 'example:' + id;
  }
}
```

The following example provides a class named `Timer` that declares its type and available API. It will emit the `timer` event when an added timer is fired.

```
const { Thing } = require('abstract-things');
const { duration } = require('abstract-things/values');

/**
* Timer that calls itself `timer:global` and that allows timers to be set
* and listened for in the network.
*/
class Timer extends Thing {
  static get type() {
    return 'timer';
  }
  static availableAPI(builder) {
    builder.event('timer')
      .description('A timer has been fired')
      .type('string')
      .done();

    builder.action('addTimer')
      .description('Add a timer to be fired')
      .argument('string', false, 'Name of timer')
      .argument('duration', false, 'Amount of time to delay the firing of the timer')
      .done();
  }

  constructor() {
    super();

    this.id = 'timer:global';
  }

  addTimer(name, delay) {
    if(! name) throw new Error('Timer needs a name');
    if(! delay) throw new Error('Timer needs a delay');

    delay = duration(delay);

    setTimeout(() => {
      this.emitEvent('timer', name);
    }, delay.ms)
  }
}
```

2.1 Naming of identifiers, types and capabilities

Naming is one of the most important aspects when both building and using things. Libraries that use `abstract-things` are expected to follow a few conventions to simplify use of the things they expose.

2.1.1 Namespaces

Libraries are expected to use a short and understandable namespace. Namespaces are used for things such as identifiers and to mark things with custom types.

The namespace should be connected to what the library interacts with. This can be something like `hue` for Philips Hue or `bravia` for Sony Bravia TVs.

2.1.2 Identifiers

Every Thing is required to have an identifer. Identifiers should be stable and globally unique. An identifier needs a prefix, which is usually the namespace of the library.

For most implementations an identifier will usually be provided with the thing being interacted with. In those case it can simply be prefixed with the namespace to create a suitable identifier.

Example of identifiers:

- `hue:000b57fffe0eee95-01`
- `miio:55409498`
- `uuid:8125606b-7b57-405b-94d6-e5720c44aa6a`
- `space:global`

As a convention things that bridge other networks such as Zigbee or Z-wave include the keyword `bridge` in their identifier, such as `hue:bridge:000b57fffe0eee95`.

2.1.3 Types

The types defined by `abstract-things` try to be short and descriptive. Libraries may mark things with custom types, but those types are expected to be namespaced or unique. Those custom types can be used to identify the specific type of thing.

Example of custom types:

- `hue:light`
- `miio:air-purifier`
- `zwave`

2.1.4 Capabilities

Capabilities follow the same rules as types, see the previous section.

2.2 Metadata

Metadata for a thing is provided either via `static` getters and methods on the defining class or during creation and initialization.

```
const { Thing, State } = require('thing');

// Calling with(State) will automatically add the state capability
class CustomThing extends Thing.with(State) {
  // This marks the thing as a custom:thing
  static get type() {
    return 'custom:thing';
  }

  constructor() {
    super();

    // Identifier is always required - set it
    this.id = 'custom:idOfThing';

    // Set the name of this thing, optional but recommended
    this.metadata.name = 'Optional name of thing';

    // Dynamically add a custom capability
    this.metadata.addCapabilities('custom:cap');
  }
}
```

2.2.1 Identifiers and name

The identifier of the thing could be considered metadata, but is actually set directly on the thing. This should be done either in the constructor or during initialization. See *Naming of identifiers, types and capabilities* for details about the identifier structure.

The name of the thing can be set on the metadata:

```
this.metadata.name = 'Custom Thing';
```

It is recommended to implement *nameable* if either the thing being interacted with does not provide a default name or it supports changing the name via its API.

2.2.2 Static getters for types and capabilities

static get type()

Set a single extra type. Usually used by type-definitions to declare their type.

Example:

```
static get type() {
  return 'namespace:custom-type';
}
```

static get types()

Set several extra types.

Example:

```
static get types() {
  return [ 'namespace:custom-type' ];
}
```

static get capability()

Set a single extra capability. Usually used by full capabilities that are mixed in with `Thing`.

Example:

```
static get capability() {
  return 'namespace:custom-cap';
}
```

static get capabilities()

Set serveral extra capabilities.

Example:

```
static get capabilities() {
  return [ 'namespace:custom-cap' ];
}
```

2.2.3 Dynamically adding

Types can be added at any time and so can capabilities. Capabilities can also be removed.

`metadata.`**addTypes**(*...types*)

Add one or more types to the metadata.

Arguments

- **...types** – Types as strings that should be added.

Returns The metadata object for chaining.

Example:

```
this.metadata.addTypes('custom:type', 'custom:type-2');
```

metadata.**addCapabilities**(*...caps*)

Arguments

- **...caps** – Capabilities as strings that should be added.

Returns The metadata object for chaining.

Example:

```
this.metadata.addCapabilities('custom:cap', 'color:temperature');
```

metadata.**removeCapabilities**(*...caps*)

Arguments

- **...caps** – Capabilities as strings that should be removed.

Returns The metadata object for chaining.

Example:

```
this.metadata.removeCapabilities('custom:cap', 'custom:connected');
```

2.3 Mixins and `with`

As things are just a combination of types and capabilities in there is support for combining them built in to the core library. `Thing` provides a method `with` that mixes several types and capabilities together:

```
class CustomThing extends Thing.with(Mixin1, Mixin2) {
  ...
}
```

2.3.1 Defining a mixin

Mixins are defined via `Thing.mixin` and they work the same as a normal `Thing`-class such as with *metadata*. Mixins are functions that create a JavaScript class with a specific parent:

```
const { Thing } = require('abstract-things');

const CustomMixin = Thing.mixin(Parent => class extends Parent {

  static get capability() {
    return 'custom:cap';
  }

  constructor(...args) {
    // Most mixins should call super with all arguments
    super(...args);

    // Set properties, initialize event listeners as normal
    this.custom = true;
  }

  customMethod() {
    return this.custom;
  }

});
```

2.3.2 Internal capabilities

In some cases when building a library things will be very straight-forward, just extend `Thing` with whatever is needed, implement the behavior and abstract methods and you're done. In other cases such as when working against a IoT-bridge for things such as lights or sensors you might find that its useful to package the API used to talk to the thing as an internal capability.

Example:

```
const { Thing } = require('abstract-things');
const { Light, SwitchablePower } = require('abstract-things/light');

// This mixin provides access to the external API for custom capabilities
const CustomAPI = Thing.mixin(Parent => class extends Parent {

  constructor(api) {
    super();

    this.api = api;
  }

  initCallback() {
    return super.initCallback()
      // Ask the fake API to initialize itself
      .then(() => this.api.init());
  }

});

/*
 * Create the custom capability that provides an implementation of
 * SwitchablePower on top of CustomAPI.
 */
const CustomPower = Thing.mixin(Parent => class extends Parent
  .with(CustomAPI, SwitchablePower) {

  initCallback() {
    return super.initCallback()
      .then(() => {
        // During init this connects to the powerChanged event of our fake API
        this.api.on('powerChanged', power => this.updatePower(power))

        // Set the power as well
        this.updatePower(this.api.hasPower());
      });
  }

  updatePower(power) {
    return this.api.setPower(power);
  }

});

const CustomDimmable = ...;

// Define the specific combinations that can exist
const PoweredThing = Light.with(CustomPower);
const PoweredAndDimambleThing = Light.with(CustomPower, CustomDimmable);

// Create them and pass the API-instance
new PoweredThing(getApiSomehow());
```

2.4 Initalization and destruction

Managing the lifecycle of a thing can be done via callbacks for initialization and destruction. Both callbacks are asynchronous using promises. Any initalization that can not be done synchronous in the constructor should be done via `initCallback`. The callback will be called when `init()` is called on the thing.

`destroyCallback` can be used for anything that needs to be done when the thing is destroyed, such as releasing socket connections and other resources. The callback is also asynchronous and will be called when `destroy()` is called on the thing.

```
class Example extends Thing {

  initCallback() {
    return super.initCallback()
      .then(() => console.log('initCallback run'));
  }

  destroyCallback() {
    return super.destroyCallback()
      .then(() => console.log('destroyCallback run'));
  }
}

new Example()
  // Initialize the thing
  .init()
  .then(thing => {
    // Then directly destroy it
    return thing.destroy();
  })
  .then(() => console.log('init() and destroy() finished'))
  .catch(err => console.log('Error occurred', err));
```

2.4.1 Protected methods

initCallback()

Callback to run when a thing is being initalized via `init()`. Implementation should return a promise and must call super.

Returns Promise that resolves when initalization is done.

Example implementation:

```
initCallback() {
  return super.initCallback()
    .then(() => {
      // Custom initalization code
    });
}
```

Example using async/await:

```
async initCallback() {
  await super.initCallback();

  // Custom initalization code
}
```

destroyCallback()

Callback to run when a thing is being destroyed via ‘‘destroy()‘. Implementation should return a promise and must call super.

Returns Promise that resolves when initalization is done.

Example implementation:

```
destroyCallback() {
  return super.destroyCallback()
    .then(() => {
      // Custom destruction code
    });
}
```

Example using async/await:

```
async destroyCallback() {
  await super.destroyCallback();

  // Custom destruction code
}
```

2.5 Handling events

Events are emitted quite often by capabilities. In most cases the capability will automatically emit events as needed, but when implementing custom capabilities simply call `emitEvent`.

2.5.1 API

emitEvent (*name*[, *payload*[, *options*]])

Emit an event with the given name. By default only a single event will be emitted during a tick. So doing `emitEvent('test')` twice in a row will only emit a single event, see the options to change the behavior.

Arguments

- **name** (*string*) – The name of the event.
- **payload** – Optional payload of the event. Can be any object that can be converted to JSON. If omitted will be equal to `null`.
- **options** – Optional object containing options for event emittal. The only option available is `multiple` which can be set to allow multiple events to be emitted during the same tick.

Example:

```
this.emitEvent('test');
this.emitEvent('rotation', angle(102));
this.emitEvent('action', { name: 'test' });
this.emitEvent('test', null, { multiple: true });
```

For information about how to listen for events see *Using things*.

2.5.2 Common patterns

It is recommended to emit as few events as possible, such as only emitting an event when something changes.

As many capabilities extend *state* a common pattern for event emittal looks something like this:

```
updatePower(newPowerValue) {
  if(this.updateState('power', newPowerValue)) {
    // Emit event if new value was different from the previous value
    this.emitEvent('power', newPowerValue);
  }
}
```

CHAPTER 3

Values

abstract-things provides implementations of many commonly used value types, including conversions from strings and to and from JSON.

3.1 Angle

Representation of an angle. Returns objects created by amounts.

```
const { angle } = require('abstract-things/values');

// With no unit - degrees are the default unit
const v = angle(200);
console.log(v.value);
console.log(v.rad); // number converted to radians

// With a unit
console.log(angle(5, 'rad'));

// String (with our without unit)
console.log(angle('5 rad'));
```

3.1.1 Units

Unit	SI	Names
Degree	No	deg, degree, degrees
Radian	Yes	rad, radian, radians

3.1.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is degrees.

Examples: `200`, `200 deg`, `5 rad`, `5 radians`

3.2 Area

Representation of an area. Returns objects created by amounts.

```
const { area } = require('abstract-things/values');

// With no unit - m² are the default unit
const v = area(1);
console.log(v.value);
console.log(v.cm2); // number converted to cm²

// With a unit
console.log(angle(50, 'cm2'));

// String (with our without unit)
console.log(angle('1 m²'));
```

3.2.1 Units

Unit	SI	Names
Square Meter	Yes	`m²`, `m^2`, `m2`, `square metre`, `square metres`, `square meter`, `square meters`
Square Inch	No	`sq in`, `square inch`, `square inches`
Square Foot	No	`sq ft`, `square foot`, `square feet`
Square Yard	No	`sq yd`, `square yard`, `square yards`
Square Mile	No	`sq mi`, `square mile`, `square miles`
Hectare	No	`ha`, `hectare`, `hectares`
Acre	No	`acre`, `acres`

3.2.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is degrees.

Examples: `200`, `200 deg`, `5 rad`, `5 radians`

3.3 Array

Value type for representing an array. Mostly used when converting to and from JSON.

```
const values = require('abstract-things/values');

const json = values.toJSON('array', [ 'one', 'two' ]);
const array = values.fromJSON('array', json);
```

3.4 Boolean

Boolean value type. Supports conversion from many common string values.

```
const { boolean } = require('abstract-things/values');

console.log(boolean('true'));
console.log(boolean(false));
console.log(boolean(1));
console.log(boolean('no'));
```

3.4.1 String conversion

`true`, `yes`, `on`, `1` will be treated as `true`. `false`, `no`, `off`, `0` represent a `false` value. Any other string values will be treated as an error.

3.5 Buffer

Buffer value type, for representing binary values.

```
const { buffer } = require('abstract-things/values');

console.log(buffer(nodeBuffer));
console.log(buffer('base64-encoded-string-here'));
console.log(buffer([ 100, 20, 240 ]));
```

3.6 Code

Value type for representing a code with a description. Codes are commonly used for things like errors, actions and modes that need to be identifiable but also a human readable description.

```
const { code } = require('abstract-things/values');

const testCode = code('test');
console.log(testCode.id);
console.log(testCode.description);

const testCode2 = code({ id: 'test', description: 'Description for code' });
const testCode3 = code('test: Description for code');
```

3.7 Color

Colors are available the `color` type and supports conversions between many common color spaces.

```
const { color } = require('abstract-things/values');

console.log(color('red'));
console.log(color('5500K'));
console.log(color('#ff0000'));
console.log(color('hsl(300, 80%, 100%)'));
```

3.7.1 RGB

RGB colors are supported and are commonly created via either named colors, such as `red` and `purple` or via Hex-notation such as `#ff0000`.

RGB colors can be created via the `rgb`-function:

```
const colorPicked = color.rgb(255, 0, 0);
```

Colors can be converted to RGB via the `rgb`-accessor and their individual components accessed:

```
const rgb = colorPicked.rgb;

console.log('Red:', rgb.red);
console.log('Green:', rgb.green);
console.log('Blue:', rgb.blue);
```

3.7.2 Temperatures

Color temperatures can be created from a string on the form `[number]K`, such as `4000K` or `5500K`: `color('4000K')`. Temperatures can also be created via the `temperature` function: `color.temperature(4000)`.

The following temperatures are available via name:

- `overcast` - 6500 Kelvins
- `daylight` - 5500 Kelvins
- `sunrise` - 2400 Kelvins
- `sunset` - 2400 Kelvins
- `candle` - 2000 Kelvins
- `moonlight` - 4100 Kelvins

Example:

```
color('4000K');
color.temperature(5500);
color('overcast');
```

Any color can be converted to its nearest temperature via the getter `temperature`:

```
console.log(color('red').temperature);
console.log(color('white').temperature);
```

The actual Kelvin-value is available via the `kelvins` accessor:

```
console.log(color.kelvins);
```

It's also possible to get a mired-version of the temperature which is used by Zigbee-lights: `color('4000K').mired.value`

3.8 Duration

Representation of a duration of time. Returns objects created by amounts.

```
const { duration } = require('abstract-things/values');

// With no unit - milliseconds are the default unit
const v = duration(2000);
console.log(v.value);
console.log(v.seconds); // number converted to seconds

// With a unit
console.log(duration(2, 's'));

// String (with our without unit)
console.log(duration('2 s'));
console.log(duration('1m 10s'));
console.log(duration('2 hours 5 m'));
```

3.8.1 Units

Unit	SI	Names
Milliseconds	No	`ms`, `millisecond`, `milliseconds`
Seconds	No	`s`, `second`, `seconds`
Minutes	No	`m`, `minute`, `minutes`
Hours	No	`h`, `hour`, `hours`
Days	No	`d`, `day`, `days`

3.8.2 String conversion

Values in the string are parsed the same as for *numbers* with multiple values with units supported.

Examples: `2000`, `2000 ms`, `5 s`, `5 seconds`, `1 hour, 10 minutes`, `1d 5m`

3.9 Energy

Representation of an energy amount. Returns objects created by amounts.

```
const { energy } = require('abstract-things/values');

// With no unit - joules are assumed
const v = energy(200);
console.log(v.value);
console.log(v.wh); // number converted to watt hours

// With a unit
console.log(energy(3.5, 'Wh'));

// String (with our without unit)
console.log(energy('5 J'));
```

3.9.1 Units

Unit	SI	Names
Joules	Yes	`J`, `j`, `joule`, `joules`
Watt hours	True	`Wh`, `wh`, `watt hour`, `watt hours`

3.9.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is joules.

Examples: `200`, `200 J`, `3.5 Wh`, `40 kJ`

3.10 Illuminance

Representation of an illuminance level. Returns objects created by amounts.

```
const { illuminance } = require('abstract-things/values');

// With no unit - lux are the default unit
const v = illuminance(200);
console.log(v.value);
console.log(v.fc); // convert to foot-candle
console.log(v.lux); // convert to lux

// With a unit
console.log(illuminance(5, 'lx'));

// String (with our without unit)
console.log(illuminance('200 lx'));
```

3.10.1 Units

Unit	SI	Names
Lux	Yes	`lx`, `lux`
Phot	No	`ph`, `phot`
Nox	No	`nx`, `nox`
Foot-candle	No	`fc`, `lm/ft`^{2}, `ft-c`, `foot-candle`, `foot-candles`, `foot candle`, `foot candles`

3.10.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is lux.

Examples: `200`, `200 lx`, `5 fc`, `5 phot`

3.11 Length

Representation of a length. Returns objects created by amounts.

```
const { length } = require('abstract-things/values');

// With no unit - metre is the default unit
const v = length(2);
console.log(v.value);
console.log(v.cm); // convert to centimetres
console.log(v.ft); // convert to feet

// With a unit
console.log(length(5, 'in'));

// String (with our without unit)
console.log(length('200 cm'));
```

3.11.1 Units

Unit	SI	Names
Metre	Yes	m, meter, meters, metre, metres
Inch	No	in, inch, inches
Feet	No	ft, foot, feet
Yard	No	yd, yard, yards
Mile	No	mi, mile, miles

3.11.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is metre.

Examples: `200`, `200 cm`, `5 ft`, `20 inches`

3.12 Mass

Representation of a mass. Returns objects created by amounts.

```
const { mass } = require('abstract-things/values');

// With no unit - grams is the default unit
const v = mass(200);
console.log(v.value);
console.log(v.kg); // convert to kilograms
console.log(v.lb); // convert to pounds

// With a unit
console.log(mass(5, 'lbs'));

// String (with our without unit)
console.log(mass('20 oz'));
```

3.12.1 Units

Unit	SI	Names
Gram	Yes	`g`, `gram`, `grams`, `gramme`, `grammes`
Pound	No	`lb`, `lbs`, `pound`, `pounds`, `#`
Ounce	No	`oz`, `ounce`, `ounces`
Stone	No	`st`, `stone`, `stones`

3.12.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is grams.

Examples: `200`, `200 g`, `5 kg`, `20 lbs`

3.13 Mixed

Value type representing mixed values. Mostly used for converting to and from JSON. A mixed value can be any other value supported.

```
const values = require('abstract-things/values');

const json = values.toJSON('mixed', somethingToConvert);
const array = values.fromJSON('mixed', json);
```

3.14 Number

Number value type.

```
const { number } = require('abstract-things/values');

console.log(number(1234));
console.log(number('1234'));
console.log(number(12.34));
console.log(number('12.34'));
```

3.14.1 String conversion

The input string will be parsed into a number. Parsing supports integers such as `1` and `545`. Decimal points are also supported: `1.2` and `4.51`.

3.14.2 SI-prefixes

Units in the SI system can be combined with SI-prefixes to create a new unit. SI-prefixes are supported both by their short names and their long names. Examples: *cm*, *milliliters*, *hPa*, *MW*, *kilowatt*

Long Name	Short name	Factor	Factor (expanded)
`yocto`	`y`	10^{-24}	0.000 000 000 000 000 000 000 001
`zepto`	`z`	10^{-21}	0.000 000 000 000 000 000 001
`atto`	`a`	10^{-18}	0.000 000 000 000 000 001
`femto`	`f`	10^{-15}	0.000 000 000 000 001
`pico`	`p`	10^{-12}	0.000 000 000 001
`nano`	`n`	10^{-9}	0.000 000 001
`micro`	`u`, `mc`, `µ`	10^{-6}	0.000 001
`milli`	`m`	10^{-3}	0.001
`centi`	`c`	10^{-2}	0.01
`deci`	`d`	10^{-1}	0.1
`deca`, `deka`	`da`	10^{1}	10
`hecto`	`h`	10^{2}	100
`kilo`	`k`	10^{3}	1 000
`mega`	`M`	10^{6}	1 000 000
`giga`	`G`	10^{9}	1 000 000 000
`tera`	`T`	10^{12}	1 000 000 000 000
`peta`	`P`	10^{15}	1 000 000 000 000 000
`exa`	`E`	10^{18}	1 000 000 000 000 000 000
`zetta`	`Z`	10^{21}	1 000 000 000 000 000 000 000
`yotta`	`Y`	10^{24}	1 000 000 000 000 000 000 000 000

3.15 Object

Value type for representing an object. Mostly used when converting to and from JSON.

```
const values = require('abstract-things/values');

const json = values.toJSON('object', { key: 'value' });
const array = values.fromJSON('object', json);
```

3.16 Percentage

Number representing a percentage, forces the number to be between 0 and 100.

```
const { percentage } = require('abstract-things/values');

console.log(percentage(80.2));
console.log(percentage('80.2'));
console.log(percentage('80%'));
```

3.16.1 String conversion

String conversion uses `parseFloat`.

3.17 Power

Representation of power. Returns objects created by amounts.

```
const { power } = require('abstract-things/values');

// With no unit - watt is the default unit
const v = power(200);
console.log(v.value);
console.log(v.hp); // convert to horsepower
console.log(v.watt); // convert to watts

// With a unit
console.log(power(1, 'hp'));

// String (with our without unit)
console.log(power('200 W'));
```

3.17.1 Units

Unit	SI	Names
Watt	Yes	`w`, `W`, `watt`
Horsepower	No	`hp`, `horsepower`

3.17.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is watt.

Examples: `200`, `200 W`, `1 hp`, `200 horsepower`

3.18 Pressure

Representation of pressure. Returns objects created by amounts.

```
const { pressure } = require('abstract-things/values');

// With no unit - pascal is the default unit
const v = pressure(101325);
console.log(v.value);
console.log(v.atm); // convert to atmospheres

// With a unit
console.log(pressure(1, 'atm'));

// String (with our without unit)
console.log(pressure('2000 hPa'));
```

3.18.1 Units

Unit	SI	Names
Pascal	Yes	pa, Pa, pascal, pascals
Atmosphere	No	atm, atmosphere, atmospheres
Bar	No	bar, bars
PSI	No	psi, pounds per square inch, pound per square inch
Torr	No	torr
mmHg	No	mmHg, 'millimetre of mercury', millimetres of mercury, millimeter of mercury, millimetres of mercury

3.18.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is pascal.

Examples: `200`, `200 Pa`, `1 atm`, `200 hPa`, `1013.25 hPa`

3.19 Sound Pressure Level

Representation of a sound pressure level. Returns objects created by amounts.

```
const { soundPressureLevel } = require('abstract-things/values');

// With no unit - decibel is the default unit
const v = soundPressureLevel(40.2);
console.log(v.value);
console.log(v.db); // convert to decibel

// With a unit
console.log(soundPressureLevel(50, 'dB'));

// String (with our without unit)
console.log(soundPressureLevel('20 decibels'));
```

3.19.1 Units

Unit	SI	Names
Decibels	No	dB, db, dbs, decibel, decibels

3.19.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is decibel.

Examples: `20`, `45.5 dB`, `100 decibels`

3.20 Speed

Representation of a speed. Returns objects created by amounts.

```
const { speed } = require('abstract-things/values');

// With no unit - metres/second is the default unit
const v = speed(20);
console.log(v.value);
console.log(v.kph); // convert to kilometers per hour
console.log(v.mps); // convert to metres per second

// With a unit
console.log(speed(50, 'km/h'));

// String (with our without unit)
console.log(speed('20 knots'));
```

3.20.1 Units

Unit	SI	Names
Metres/Second	Yes	`m/s`, `mps`, `metre per second`, `metres per second`, `meter per second`, `meters per second`, `metre/second`, `metres/second`, `meter/second`, `meters/second`
Kilometre/Hour	No	`km/h`, `kph`, `kilometre per hour`, `kilometres per hour`, `kilometer per hour` `kilometers per hour`, `kilometers/hour`, `kilometre/hour`
Miles/Hour	No	`mph`, `mile per hour`, `miles per hour`, `mile/hour`, `miles/hour`
Feet/Second	No	`ft/s`, `fps`, `foot per second`, `feet per second`, `foot/second`, `feet/second`
Knot	No	`kt`, `knot`, `knots`

3.20.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is metres per second.

Examples: `20`, `20 m/s`, `100 km/h`, `30 mph`, `20 knots`

3.21 String

String value type.

```
const { string } = require('abstract-things/values');

console.log(string('Hello world'));
console.log(string(12));
```

3.22 Temperature

Representation of a temperature. Returns objects created by amounts.

```
const { temperature } = require('abstract-things/values');

// With no unit - celsius is the default unit
const v = temperature(20);
console.log(v.value);
console.log(v.F); // convert to fahrenheit
console.log(v.celsius); // convert to celsius

// With a unit
console.log(temperature(50, 'F'));

// String (with our without unit)
console.log(temperature('220 K'));
```

3.22.1 Units

Unit	SI	Names
Celsius	No	`C`, `c`, `celsius`
Kelvin	Yes	`K`, `kelvin`, `kelvins`
Fahrenheit	No	`F`, `f`, `fahrenheit`, `fahrenheits`

3.22.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is Celsius.

Examples: `20`, `20 C`, `100 kelvins`, `30 F`

3.23 Voltage

Representation of a voltage. Returns objects created by amounts.

```
const { voltage } = require('abstract-things/values');

// With no unit - volts is the default unit
const v = voltage(20);
console.log(v.value);
console.log(v.volts); // convert to volts

// With a unit
console.log(voltage(50, 'V'));

// String (with our without unit)
console.log(voltage('220 volts'));
```

3.23.1 Units

Unit	SI	Names
Volt	Yes	`V`, `v`, `volt`, `volts`

3.23.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is volts.

Examples: `20`, `20 V`, `100 volts`

3.24 Volume

Representation of a volume. Returns objects created by amounts.

```
const { volume } = require('abstract-things/values');

// With no unit - litres is the default unit
const v = volume(20);
console.log(v.value);
console.log(v.gallon); // convert to gallons
console.log(v.L); // convert to litres
console.log(v.ml); // convert to millilitres

// With a unit
console.log(volume(50, 'cl'));

// String (with our without unit)
console.log(voltage('220 ml'));
```

3.24.1 Units

Unit	SI	Names
Liter	Yes	`l`, `L`, `liter`, `litre`, `litre`, `litres`
Gallon	No	`gal`, `gallon`, `gallons`
Quart	No	`qt`, `quart`, `quarts`
Pint	No	`pt`, `pint`, `pints`
Cup	No	`cu`, `cup`, `cups`
Fluid ounce	No	`floz`, `oz`, `fluid ounce`, `ounce`, `fluid ounces`, `ounces`
Tablespoon	No	`tb`, `tbsp`, `tbs`, `tablesppon`, `tablespoons`
Teaspoon	No	`tsp`, `teaspoon`, `teaspoons`

3.24.2 String conversion

Strings are parsed the same as for *numbers* with the addition of units being parsed. The default unit is litres.

Examples: `1`, `1 l`, `100 cl`, `5 tbps`

CHAPTER 4

Common capabilities

4.1 `cap:children` - access child things

This capability is used when a thing has children. Children are used to map when a thing is a bridge or when a physical thing has several virtual children. An example of such use is for *power strips* that support control or monitoring of their indivudal outlets.

```
if(thing.matches('cap:children')) {
  // Get all children
  const children = thing.children();

  // Get a single child
  const child = thing.child('usb');
}
```

4.1.1 API

`children`()

Get the children of the thing as an iterable.

Example:

```
for(const child of thing.children) {
  console.log('Child:', child);
}
```

`child`(*id*)

Get a child based on its identifier. The identifier can either be a full identifier or a partial one.

Arguments

- **`id`** (*`string`*) – The identifier to get thing for.

Returns The thing if found or `null`.

Example:

```
const child = thing.child(fullIdOrPartial);
```

4.1.2 Partial identifiers

Partial identifiers are identifiers that make it easier to find a child. The are constructed in such a way that the full identifier is a combination of the parent id with a short logical id for the child.

For a thing with id `example:thing` a child with the partial identifier `usb` would have the full id `example:thing:usb`.

4.1.3 Events

`thing:available`
A new child is available for this thing. Emitted whenver a child is added.

Example:

```
thing.on('thing:available', child => console.log('Added child:', child));
```

`thing:unavailable`
A child is no longer available. Emitted when a child is no longer available.

Example:

```
thing.on('thing:unavailable', child => console.log('Removed child:', child));
```

4.1.4 Protected methods

`addChild`(*thing*)
Add a child to this thing. This will add the thing and emit the `thing:available` event.

Arguments

- **`thing`** – The thing to add as a child.

Example:

```
this.addChild(new ChildThing(...));
```

`removeChild`(*thingOrId*)
Remove a child from this thing. This will remove the thing and emit the `thing:unavailable` event.

Arguments

- **`thingOrId`** – The thing instance or identifier that should be removed.

Example:

```
this.removeChild(existingChild);
this.removeChild('id-of-thing');
```

`findChild`(*filter*)
Find the first child that matches the given filter function.

Arguments

- **filter** (*function*) – Filter function to apply, should return `true` when a thing matches.

Returns Thing if found, otherwise `null`.

Example:

```
// Get the first power outlet
this.findChild(thing => thing.matches('type:power-outlet'));
```

4.1.5 Implementing capability

When implementing this capability children need to be managed. This can either be done manually or via a method such as `ChildSyncer`.

Manual management is recommended if only a few known children exist:

```
const { Thing, Children } = require('abstract-things');

class Example extends Thing.with(Children) {

  constructor() {
    super();

    this.addChild(new ChildThing(this, ...));
  }

}
```

Using `ChildSyncer`, commonly for things such as bridges:

```
const { ChildSyncer } = require('abstract-things/children');

class Example extends Thing.with(Children) {

  constructor() {
    super();

    this.syncer = new ChildSyncer(this, (def, thing) => {

    });
  }

  async initCallback() {
    await super.initCallback();

    await this.loadChildren();
  }

  async loadChildren() {
    /*
     * Load the children, should be an array with objects that contain
     * at least an `id` property.
     */
    const defs = await loadChildrenSomehow();

    await syncer.update(defs);
  }
}
```

4.2 `cap:state` - state tracking

The `state`-capability provides a way to get and update the state of a thing. State is split into several state keys that are updated separately.

```
if(thing.matches('cap:state')) {
  console.log('Current state:', this.state);
}
```

4.2.1 API

state()

Get the current overall state.

Returns Promise that resolves to an object representing the current state. Keys represent names of the state key.

Usage:

```
const state = await thing.state();
console.log('State is', state);
console.log('Value of power is', state.power);
```

4.2.2 Events

stateChanged

State has changed for the thing.

```
thing.on('stateChanged', change =>
  console.log('Key', change.key, 'changed to', change.value)
);
```

4.2.3 Protected methods

getState(*key*[, *defaultValue*])

Get the current value of the given state key.

Arguments

- **power** (*string*) – The state key to get value for.
- **defaultValue** – Fallback to return if the state key is not set.

Returns The value for the state key, the default value or `null`.

updateState(*key*, *value*)

Update the state of the given key. This will update the state key and emit the event `stateChanged`.

Arguments

- **key** (*string*) – The state key to update.
- **value** – The new value of the state key.

Returns Boolean indicating if the state value has changed.

removeState(*key*)

Remove state stored for the given key. Will emit a `stateChanged` event.

Arguments

- **key** (*string*) – The state key to remove.

4.2.4 Implementing capability

The `state`-capability has no functions that need to be implemented. `updateState` can be called at any time to update a state key.

```
const { Thing, State } = require('abstract-things');

class Example extends Thing.with(State) {
  constructor() {
    super();

    this.updateState('key', true);
  }
}
```

4.3 `cap:restorable-state` - capture and restore state

`restorable-state` provides an extension to *state* that supports capturing and setting state.

```
if(thing.matches('cap:restorable-state')) {
  console.log('Keys that can be restored:' , thing.restorableState);

  // Capture the state
  const state = await thing.captureState();

  // A bit later the state can be restored
  await thing.setState(state);
}
```

4.3.1 API

restorableState

Get an array of the state-keys that are restorable.

Example:

```
console.log(thing.restorableState);
console.log(thing.restorableState[0]);
```

captureState()

Capture all the state that can be restored.

Returns Promise that resolves to the object representing the state.

Example:

```
thing.setState(state)
  .then(...)
  .catch(...);

const state = await thing.captureState();
```

setState(*state*)

Set the state of the thing. Can be used together with result captured via `captureState()`.

Arguments

- **state** (*object*) – State to set.

Returns Promise that will resolve when state has been set.

Example:

```
thing.setState(state)
  .then(...)
  .catch(...);

await thing.setState(state);
```

4.3.2 Protected methods

changeState(*state*)

Abstract. Change the state of the thing. Implementations should call `super` and restore custom state-keys when that promise resolves.

Example:

```
changeState(state) {
  return super.changeState(state)
    .then(() => {
      if(typeof state.color !== 'undefined') {
        return changeColorSomehow(state.color);
      }
    });
}
```

4.3.3 Implementing capability

Most implementations of this capability are by other capabilities. Implementations need to override both `get restorableState` and `changeState`.

The getter for `restorableState` must also take care to include the state-keys defined as restorable by its parent:

```
get restorableState() {
  return [ ...super.restorableState, 'own-key' ];
}
```

It is recommended to provide a method that defines a default restore behavior, so that its easy to override the default behavior if needed.

Example:

```
const { Thing, RestorableState } = require('abstract-things');

const Custom = Thing.capability(Parent => class extends Parent.with(RestorableState) {

  get restorableState() {
    // Must call super.restorableState and make it part of the result
    return [ ...super.restorableState, 'color' ];
  }

  changeState(state) {
    return super.changeState(state)
      .then(() => {
        if(typeof state.color !== 'undefined') {
          return this.setColorState(state.color);
        }
      });
  }

  setColorState(color) {
    // The default restore behavior is to call setColor
    return this.setColor(color);
  }

  setColor(color) {
    ...
  }
});
```

4.4 `cap:nameable` - renameable things

`nameable` is used by things that have a name that can be updated.

```
if(thing.matches('cap:nameable')) {
  thing.setName('New Name')
    .then(() => console.log('Name updated'))
    .catch(err => console.log('Error occurred during update:', err));
}
```

4.4.1 API

`setName`(*name*)
Update the name of this thing.

Arguments

- **name** (*`string`*) – Name for thing.

Returns Promise that resolves to the name set.

4.4.2 Protected methods

`changeName`(*name*)
Abstract. Change and store the name of the thing. This is called when the user calls `setName`. This method should update the `name` property of the metadata when the new name has been stored.

Arguments

- **name** (*string*) – The name to set.

Returns Promise that resolves after name has been updated.

Example:

```
changeName(name) {
  return setNameSomehow(name)
    .then(() => this.metadata.name = name);
}
```

4.4.3 Implementing capability

`changeName` needs to be implemented to actually set the name. The name should be loaded and set either in the constructor or `initCallback` of the thing.

```
const { Thing, Nameable } = require('abstract-things');

class Example extends Thing.with(Nameable) {
  initCallback() {
    return super.initCallback()
      .then(() => loadNameSomehow())
      .then(name => this.metadata.name = name);
  }

  changeName(name) {
    return setNameSomehow(name)
      .then(() => this.metadata.name = name);
  }
}
```

For things that just need to be nameable a special capability is provided that stores the name locally:

```
const { Thing, EasyNameable } = require('abstract-things');

class Example extends Thing.with(EasyNameable) {
}
```

4.5 `cap:power` - monitor power state

The `power`-capability is used for any thing that can monitor its power state.

```
if(thing.matches('cap:power')) {
  console.log('Power is', await thing.power());

  thing.on('powerChanged', power => console.log('Power is now', power));
}
```

Related capabilities: *switchable-power*, *state*

4.5.1 API

power()

Get the current power state.

Returns Promise that resolves to a *boolean* representing the current power state.

Example:

```
thing.power()
  .then(power => ...)
  .catch(...);

const powerIsOn = await thing.power();
```

4.5.2 Events

powerChanged

The current power state has changed. Payload will be current power state as a *boolean*.

```
thing.on('powerChanged', power => console.log('power is now:', power));
```

4.5.3 Protected methods

updatePower(*power*)

Update the current power state of the thing. Will change the state key `power` and emit the `power` event.

Arguments

- **power** (*boolean*) – The current power state.

4.5.4 Implementing capability

The `power`-capability has no functions that need to be implemented. Call `updatePower` whenever the monitored power state changes.

Example:

```
const { Thing, Power } = require('abstract-things');

class Example extends Thing.with(Power) {
  constructor() {
    super();

    // Indicate that power has been switched every second
    setInterval(() => {
      this.updatePower(! this.getState('power'));
    }, 1000);
  }
}
```

4.6 `cap:switchable-power` - switch power state

The `switchable-power`-capability is an extension to the *power-capability* for things that can also switch their power state.

```
if(thing.matches('cap:switchable-power')) {
  console.log('Power is', await thing.power());

  // Switch the thing on
  await thing.power(true);
}
```

Related capabilities: *power*, *state*

4.6.1 API

`power` ([*powerState*])

Get or set the current power state.

Arguments

- **`powerState`** (*`boolean`*) – Optional *boolean* to change power state to.

Returns Promise when switching state, *boolean* if getting.

Example:

```
// Getting returns a boolean
const powerIsOn = await thing.power();

// Switching returns a promise
thing.power(false)
  .then(result => console.log('Power is now', result))
  .catch(err => console.log('Error occurred', err);
```

`setPower` (*powerState*)

Set the power of the thing.

Arguments

- **`powerState`** (*`boolean`*) – The new power state as a *boolean*.

Returns Promise that will resolve to the new power state.

Example:

```
thing.setPower(true)
  .then(result => console.log('Power is now', result))
  .catch(err => console.log('Error occurred', err);

await thing.setPower(true);
```

`togglePower` ()

Toggle the power of the thing. Will use the currently detected power state and switch to the opposite.

Returns Promise that will resolve to the new power state.

Example:

```
thing.togglePower()
  .then(result => console.log('Power is now', result))
  .catch(err => console.log('Error occurred', err);
```

turnOn()

Turn the thing on.

Returns Promise that will resolve to the new power state.

Example:

```
thing.turnOn()
  .then(result => console.log('Power is now', result))
  .catch(err => console.log('Error occurred', err);
```

turnOff()

Turn the thing off.

Returns Promise that will resolve to the new power state.

Example:

```
thing.turnOff()
  .then(result => console.log('Power is now', result))
  .catch(err => console.log('Error occurred', err);
```

4.6.2 Protected methods

changePower(*power*)

Abstract. Change the power of this thing. Called on the thing when of the power methods request a change. Implementations should call `updatePower` before resolving to indicate that a change has occurred.

Can be called with the same power state as is currently set.

Arguments

- **power** (*boolean*) – The new power of the thing as a *boolean*.

Returns Promise if asynchronous.

4.6.3 Implementing capability

The `switchable-power`-capability requires that the function `changePower` is implemented.

Example:

```
const { Thing, SwitchablePower } = require('abstract-things');

class Example extends Thing.with(SwitchablePower) {
  constructor() {
    super();

    // Make sure to initialize the power state via updatePower
  }

  changePower(power) {
    /*
     * This method is called whenever a power change is requested.
     *
     * Change the power here and return a Promise if the method is
     * asynchronous. Also call updatePower to indicate the new state
     * if not done by switching.
     */
    return switchWithPromise(power)
     .then(() => this.updatePower(power));
  }
}
```

4.7 `cap:mode` - monitor mode

`mode` is used for things that have a mode that can be monitored.

```
if(thing.matches('cap:mode')) {
  console.log('Mode is', await thing.mode());

  thing.on('modeChanged', mode => console.log('Mode is now', mode));
}
```

4.7.1 API

mode()

Get the current mode of the thing.

Returns Promises that resolves to a *string* indicating the identifier of the mode.

Example:

```
thing.mode()
  .then(mode => ...)
  .catch(...);

const mode = await thing.mode();
```

modes()

Get the modes that this thing supports.

Returns Promise that will resolve to the modes as an array containing *codes*.

Example:

```
const modes = await thing.modes();

const firstMode = modes[0];
console.log('Id:', firstMode.id);
console.log('Description:', firstMode.description);
```

4.7.2 Events

modeChanged

The current mode has changed. Payload of the event is the current mode as a *string*.

```
thing.on('modeChanged', mode => console.log('Mode is now', mode));
```

modesChanged
The available modes have changed.

4.7.3 Protected methods

updateMode (*mode*)
Update the currently detected mode. Calling this method with a new mode will change the mode and trigger the `mode` event.

Arguments

- **mode** (*string*) – The id of the current mode.

Example:

```
this.updateMode('silent');
```

updateModes (*modes*)
Update the modes that are available for the thing.

Arguments

- **modes** (*array*) – Array of modes as *codes*. Entries in the array will be automatically converted to codes if possible.

Example:

```
this.updateModes([
  'idle',
  'silent: Silent speed',
  { id: 'auto', description: 'Autoselect speed' }
]);
```

4.7.4 Implementing capability

When implementing this capability call `updateModes` in the constructor or `initCallback` of the thing. `updateMode` should be used whenever the mode is changed.

Example:

```
const { Thing, Mode } = require('abstract-things');

class Example extends Thing.with(Mode) {
  initCallback() {
    return super.initCallback()
      .then(() => this.updateModes(modesDetected));
  }
}
```

4.8 `cap:switchable-mode` - switch mode

Capability used for things that can switch their mode.

```
if(thing.matches('cap:switchable-mode')) {
  console.log('Mode is', await thing.mode());

  // Switch the mode
  await thing.mode('new-mode');
}
```

4.8.1 API

mode([*newMode*])

Get or set the mode of the thing. Will return the mode as a string if no mode is specified. Will return a promise if a mode is specified.

Arguments

- **newMode** (*string*) – Optional mode to change to.

Returns Promise when switching mode, string if getting.

Example:

```
// Getting returns a string
const currentMode = await thing.mode();

// Switching returns a promise
thing.mode('new-mode')
  .then(result => console.log('Mode is now', result))
  .catch(err => console.log('Error occurred', err);
```

4.8.2 Protected methods

changeMode(*newMode*)

Abstract. Change to a new mode. Will be called whenever a change to the mode is requested. Implementations should call `updateMode(newMode)` before resolving to indicate that the mode has changed.

Arguments

- **newMode** (*string*) – The new mode of the thing.

Returns Promise if asynchronous.

4.8.3 Implementing capability

Implementations require that the method `changeMode` is implemented.

```
const { Thing, SwitchableMode } = require('abstract-things');

class Example extends Thing.with(SwitchableMode) {

  changeMode(newMode) {
    return swithcWithPromise(newMode)
      .then(() => this.updateMode(newMode));
  }

}
```

4.9 `cap:error-state` - error reporting

The `error-state` capability is used when a thing can report an error, such as a humidifier running out of water or a autonomous vacuum getting stuck.

```
if(thing.matches('cap:error-state')) {
  if(thing.error) {
    console.log('Error is:', thing.error);
  }
}
```

4.9.1 API

error()

Get the current error or `null` if no error.

Returns Promise that resolves to a *code* if the thing is currently in an error state, or `null` if no error state.

Example:

```
thing.error()
  .then(err => ...)
  .catch(...);

const error = await thing.error();
```

4.9.2 Events

errorChanged

The current error has changed. The payload will be the current error state as a *code* or `null`.

Example:

```
thing.on('errorChanged', error => console.log('Error state:', error));
```

error

Emitted when an error occurs. The payload will be the error.

Example:

```
thing.on('error', error => console.log('Error occured:', error));
```

errorCleared

Emitted when the thing no longer has an error.

Example:

```
thing.on('errorCleared', () => console.log('Thing no longer has an error'));
```

4.9.3 Protected methods

updateError(*batteryLevel*)

Update the current error state.

Arguments

- **error** (*code*) – The new error state as a *code* or `null` if no error.

Example:

```
this.updateError('some-error');
this.updateError(null);
```

4.9.4 Implementing capability

When implementing this capability the implementor needs to call `updateError` whenever an error state is entered or left.

```
const { Thing, ErrorState } = require('abstract-things');

class Example extends Thing.with(ErrorState) {

}
```

4.10 `cap:battery-level` - monitor battery level

The `battery-level` capability is used for things that have a battery that can be monitored. Sometimes this capability is combined with *charging-state* if the thing also can report when it is being charged.

```
if(thing.matches('cap:battery-level')) {
  console.log('Current battery level:', await thing.batteryLevel());
}
```

4.10.1 API

batteryLevel()

Get the current battery level as a *percentage* between 0 and 100.

Returns Promise that resolves to the battery level in percent.

Example:

```
thing.batteryLevel()
  .then(level => ...)
  .catch(...);

const level = await thing.batteryLevel();
```

4.10.2 Events

batteryLevelChanged

The current battery level has changed. Payload will be the new battery level as a *percentage*.

```
thing.on('batteryLevelChanged', batteryLevel => console.log('Battery level is now:
↪', batteryLevel));
```

4.10.3 Protected methods

updateBatteryLevel (*batteryLevel*)

Update the current battery level. Should be called whenever a change in battery level is detected.

Arguments

- **batteryLevel** (*percentage*) – The new battery level. Will be converted to a *percentage*.

Example:

```
this.updateBatteryLevel(20);
this.updateBatteryLevel('10');
```

4.10.4 Implementing capability

When implementing this capability the implementor needs to call `updateBatteryLevel` whenever the battery level changes.

```
const { Thing, BatteryLevel } = require('abstract-things');

class Example extends Thing.with(BatteryLevel) {

  initCallback() {
    return super.initCallback()
      .then(readBatteryLevelSomehow)
      .then(batteryLevel => {
        this.updateBatteryLevel(batteryLevel);
      });
  }

}
```

4.11 `cap:charging-state` - monitor if charging

The `charging-state` capability is used for things that have a battery and can report if they are being charged or not. Some of these things will also have the *battery-level* capability.

```
if(thing.matches('cap:charging-state')) {
  if(await thing.charging()) {
    // This thing is charging
  }
}
```

4.11.1 API

charging ()

Get the current charging state as a *boolean*. `true` indicates that the thing is charging.

Returns Promise that resolves to the current charging state.

Example:

```
thing.charging()
  .then(isCharging => ...)
  .catch(...);

const isCharging = await thing.charging();
```

4.11.2 Events

chargingChanged
The current charging state has changed. Payload will be the new state a *boolean*.

```
thing.on('chargingChanged', v => console.log('Charging:', v));
```

chargingStarted
The thing is now being charged.

```
thing.on('chargingStarted', () => console.log('Charging started'));
```

chargingStopped
The thing is no longer being charged.

```
thing.on('chargingStopped', () => console.log('Charging stopped'));
```

4.11.3 Protected methods

updateCharging(*chargingState*)
Update the current charging state. Should be called whenever a change in charging state is detected.

Arguments

- **chargingState** (*boolean*) – The new charging state.

Example:

```
this.updateCharging(true);
```

4.11.4 Implementing capability

When implementing this capability the implementor needs to call `updateCharging` whenever the charging state changes.

```
const { Thing, ChargingState } = require('abstract-things');

class Example extends Thing.with(ChargingState) {

  initCallback() {
    return super.initCallback()
      .then(readChargingStateSomehow)
      .then(chargingState => {
        this.updateCharging(chargingState);
      });
  }

}
```

4.12 `cap:autonomous-charging` - request charging

The `autonomous-charging` capability is used for things that have a battery and can charge it on request. This is commonly things such as vacuum robots that can head to a charging station to recharge.

```
if(thing.matches('cap:autonomous-charging')) {
  thing.charge()
    .then(() => console.log('Charging has been requested'))
    .catch(...);
}
```

4.12.1 API

charge()

Request that the thing charges.

Returns Promise that resolves to `null`

Example:

```
thing.charge()
  .then(...)
  .catch(...);

await thing.charge();
```

4.12.2 Protected methods

activateCharging()

Activate charging of the thing. Called by `charge()`.

Returns Promise that resolves when activation is performed.

Example:

```
activateCharging() {
  return activateChargingSomehow();
}
```

4.12.3 Implementing capability

When implementing this capability the implementor needs to implement the method `activateCharging`.

```
const { Thing, AutonomousCharging } = require('abstract-things');

class Example extends Thing.with(AutonomousCharging) {

  activateCharging() {
    // Create a promise that resolves when charging has been activated
    return activateChargingSomehow();
    }

}
```

CHAPTER 5

Controllers

Controllers are things that control other things, such as remotes and buttons. If a thing implements the *actions*-capability it will emit events when an action occurs such as a button being pressed. The actual actions available vary from thing to thing.

5.1 `cap:actions` - emit events on actions

This capability is used when a thing support emitting events when an action such a button press occurs.

```
if(thing.matches('cap:actions')) {
  // This thing supports actions
  thing.on('action', action => console.log('Action occurred:', action);

  // Listen for a specific action
  thing.on('action:test', () => console.log('Test action occurred');
}
```

5.1.1 API

actions();()

Get the actions that the thing supports.

Returns Promise that resolves to any array containing the actions as *codes*.

Example:

```
const actions = await thing.actions();

const action = actions[0];
console.log('First action id:', action.id);
```

5.1.2 Events

`actionsChanged`

The available actions have changed. Payload will be the same value that will be returned by the `values` attribute.

Example:

```
thing.on('actionsChanged', actions => console.log('Actions are now:', actions);
```

`action`

An action has occurred. The payload is an object with the keys:

- `action` - the identifier of the action
- `data` - optional data of the action

Example:

```
thing.on('action', e => console.log('Action', e.action, 'with data', e.data));
```

`action:<id>`

An action of type `<id>` has occurred. `<id>` will be a supported action, see the `actions` attribute for supported actions.

```
thing.on('action:test', () => console.log('Test action occurred'));
```

5.1.3 Protected methods

`updateActions`(*actions*)

Update the available actions.

Arguments

- **`actions`** (`array`) – The actions that this thing supports. Each item in the array will be converted to *code*.

Example:

```
this.updateActions([
  'button1',
  { id: 'button2', description: 'Optional description' },
  'button3: Description for button 3'
]);
```

`emitAction`(*action*[, *data*])

Emit an action with the given identifier. Optionally provide some extra data.

Arguments

- **`action`** (`string`) – The action that should be emitted.
- **`data`** (`mixed`) – The optional data to include with the action event.

Example:

```
this.emitAction('button1');
this.emitAction('rotated', { amount: 45 });
```

5.1.4 Implementing capability

When implementing this capability `updateActions` need to be called with the available actions. When an action occurs the method `emitAction` needs to be called.

Example:

```
const { Thing } = require('abstract-things');
const { Actions } = require('abstract-things/contollers');

class Example extends Thing.with(Actions) {
  initCallback() {
    return super.initCallback()
      .then(() => this.updateActions(actionsDetected));
  }
}
```

5.2 `type:controller` - Generic controller

The `controller` type is used for things that are controllers and can be combined with more specific types.

Controllers commonly emit events and implement the *actions-capability*.

```
if(thing.matches('type:controller')) {
  // This is a wall controller

  if(thing.matches('cap:actions')) {
    // Controller supports listening for actions
  }
}
```

5.2.1 Implementing type

```
const { Controller, Actions } = require('abstract-things/controllers');

class Example extends Controller.with(Actions, ...) {

}
```

5.3 `type:button` - Single button

If a thing is a single button the type `button` is commonly used. Buttons may emit events when buttons are pressed while implementing the *actions-capability*. Buttons are automatically marked as controllers.

```
if(thing.matches('type:button')) {
  // This is a button

  if(thing.matches('cap:actions')) {
    // Button supports listening for actions
  }
}
```

5.3.1 Implementing type

```
const { Button, Actions } = require('abstract-things/controllers');

class Example extends Button.with(Actions, ...) {

}
```

5.4 `type:remote-control` - Remote controls

Remote controls are marked with the type `remote-control`. Many remote controls are capable of emitting events when buttons are pressed and implement the *actions-capability*. Remote controls are automatically marked as controllers.

```
if(thing.matches('type:remote-control')) {
  // This is a remote control

  if(thing.matches('cap:actions')) {
    // Remote control supports listening for actions
  }
}
```

5.4.1 Implementing type

```
const { RemoteControl, Actions } = require('abstract-things/controllers');

class Example extends RemoteControl.with(Actions, ...) {

}
```

5.5 `type:wall-controller` - Controllers mounted on a wall

`wall-controller` is used for controllers that are commonly mounted on a wall, such as switches and scene controllers. Wall controllers are automatically marked as controllers.

Wall controllers may emit events when buttons are pressed while implementing the *actions-capability*.

```
if(thing.matches('type:wall-controller')) {
  // This is a wall controller

  if(thing.matches('cap:actions')) {
    // Controller supports listening for actions
  }
}
```

5.5.1 Implementing type

```
const { WallController, Actions } = require('abstract-things/controllers');

class Example extends WallController.with(Actions, ...) {

}
```

CHAPTER 6

Lights

The main type for lights is `light`. Lights commonly use at least the *switchable-power* capability.

```
if(thing.matches('type:light', 'cap:switchable-power')) {
  thing.power(true)
    .then(() => console.log('powered on'))
    .catch(err => console.log('error occurred', err));
}
```

6.1 Implementing lights

6.1.1 Protected methods

`setLightState`(*state*)

Set the state of the light. Light capabilities use this as a hook for restoring state. If this is not overriden capabilities implement a default behavior.

Arguments

- **`state`** (*`object`*) – The state to set.

Returns Promise that resolves when the state is set.

Example:

6.1.2 Power switching

To support proper restoring of power the implementors of lights should use a custom `SwitchablePower`:

```
const { Light, SwitchablePower } = require('abstract-things/lights');

class Example extends Light.with(SwitchablePower) {
  changePower(power) {
    return changePowerOfLight(power);
  }

}
```

6.2 `type:light-bulb` - Light bulbs

The type `light-bulb` is a marker used to mark lights that are of the bulb type.

```
if(thing.matches('cap:light-bulb')) {
  // The thing is a light bulb
}
```

6.2.1 Implementing capability

Light bulbs are an extension to *lights* and need to follow the same implementation guidelines.

```
const { LightBulb, SwitchablePower } = require('abstract-things/lights');

class Example extends LightBulb.with(SwitchablePower) {

  changePower(power) {
    return changePowerOfLight(power);
  }

}
```

6.3 `type:light-strip` - Light strips

The type `light-bulb` is a marker used to mark lights that are of the strip type.

```
if(thing.matches('cap:light-strip')) {
  // The thing is a light strip
}
```

6.3.1 Implementing capability

Light strips are an extension to *lights* and need to follow the same implementation guidelines.

```
const { LightStrip, SwitchablePower } = require('abstract-things/lights');

class Example extends LightStrip.with(SwitchablePower) {

  changePower(power) {
    return changePowerOfLight(power);
  }

}
```

6.4 `cap:fading` - support for fading changes

Capability used to mark lights that support fading of changes. When this capability is present the `duration` argument for other methods is available.

```
if(thing.matches('type:light', 'cap:fading')) {
  // This light supports fading
  const time = await this.maxChangeTime();
  console.log('Maximum fading time in milliseconds:', time.ms);
}
```

6.4.1 API

`maxChangeTime`

The maximum *duration* of time a change can be.

6.4.2 Protected methods

`updateMaxChangeTime`(*time*)

Arguments

- **`time`** (*`duration`*) – The maximum time the light can fade as a *duration*.

Example:

```
this.updateMaxChangeTime('20s');
```

6.4.3 Implementing capability

Implementing this capability requires that the maximum change time is set either in the constructor or in the `initCallback()`.

Example:

```
const { Light, Fading } = require('abstract-things/lights');

class Example extends Light.with(Fading) {

  initCallback() {
    return super.initCallback()
      // Set the maximum change time to 5 seconds
      .then(() => this.updateMaxChangeTime('5s'));
  }

}
```

6.5 `cap:brightness` - read brightness

Capability used when a light supports reading the brightness. Usually this is combined with *dimmable* for lights that can actually change their brightness.

```
if(thing.matches('cap:brightness')) {
  console.log(await thing.brightness());
}
```

6.5.1 API

brightness()

Get the brightness of the light.

Returns Promise that resolves to a *percentage* between 0 and 100, representing the brightness.

Example:

```
console.log(await thing.brightness());
```

6.5.2 Events

brightnessChanged

Brightness has changed. The payload of the event will be the brightness as a *percentage*.

```
thing.on('brightnessChanged', bri => console.log('Brightness is now', bri));
```

6.5.3 Protected functions

updateBrightness(*brightness*)

Update the current brightness. Should be called whenever the brightness has been detected to have changed.

Arguments

- **brightness** (*number*) – The new brightness as a *percentage*.

6.5.4 Implementing capability

This capability has no functions that need to be implemented. Things using the capability should call `updateBrightness` whenever the brightness changes.

```
const { Light, Brightness } = require('abstract-things/lights');

class Example extends Light.with(Brightness) {

  initCallback() {
    return super.initCallback()
      .then(() => this.updateBrightness(initialBrightnessHere));
  }

}
```

6.6 `cap:dimmable` - change brightness

Capability used when a light supports changing the brightness, extends *brightness*-capability.

```
if(thing.matches('cap:dimmable')) {
  // Get the current brightness
  console.log(await thing.brightness());

  // Set the current brightness
  const newBrightness = await thing.brightness(10);
}
```

6.6.1 API

`brightness`([*brightnessChange*[, *duration*]])

Get or change the brightness of the light. Setting the brightness to zero will power off the light. Setting the brightness to non-zero such as when increasing the brightness will turn it on.

Arguments

- **`brightnessChange`** (*`percentage`*) – Optional brightness *percentage* to set as a number or a change in brightness as a string. `20` would be 20% brightness, `'+10'` would be an increase of 10%.
- **`duration`** (*`Duration`*) – Optional *duration* to perform change in brightness over. Supported when the light has the *fading*-capability.

Returns Promise that resolves to the current or the set brightness.

Example:

```
// Get the current brightness
const currentBrightness = thing.brightness();

// Set a specific brightness
thing.brightness(20)
  .then(bri => console.log('Brightness is now', bri))
  .catch(err => console.log('Error while setting', err));

// Increase the brightness
thing.brightness('+10')
  .then(...)
  .catch(...);

// Set the brightness over 2 seconds (if cap:fading)
thing.brightness(70, '2s')
  .then(...)
  .catch(...);
```

`setBrightness`(*brightness*[, *duration*])

Set the brightness of the light. Setting the brightness to zero will power off the light. Setting the brightness to non-zero such as when increasing the brightness will turn it on.

Arguments

- **`brightness`** (*`percentage`*) – The brightness as a *percentage* the light should try to set.
- **`duration`** (*`Duration`*) – Optional *duration* to perform change in brightness over. Supported when the light has the *fading*-capability.

Returns Promise resolving to the new brightness.

Example:

```
thing.setBrightness(20)
  .then(bri => console.log('Brightness is now', bri))
  .catch(err => console.log('Error while setting', err));
```

increaseBrightness(*amount*[, *duration*])
Increase the brightness of the light. This will turn on the light.

Arguments

- **amount** (*percentage*) – The amount as a *percentage* to increase the brightness.
- **duration** (*Duration*) – Optional *duration* to perform change in brightness over. Supported when the light has the *fading*-capability.

Returns Promise that resolves to the new brightness.

Example:

```
thing.increaseBrightness(15)
  .then(bri => console.log('Brightness is now', bri))
  .catch(err => console.log('Error while setting', err));
```

decreaseBrightness(*amount*[, *duration*])
Decrease the brightness of the light. Decreasing to zero will power off the light.

Arguments

- **amount** (*percentage*) – The amount as a *percentage* to decrease the brightness.
- **duration** (*Duration*) – Optional *duration* to perform change in brightness over. Supported when the light has the *fading*-capability.

Returns Promise that resolves to the new brightness.

Example:

```
thing.decreaseBrightness(15)
  .then(bri => console.log('Brightness is now', bri))
  .catch(err => console.log('Error while setting', err));
```

6.6.2 Protected methods

changeBrightness(*targetBrightness*, *options*)
Abstract. Change the brightness of the light. Implementations need to supports the following:

- If `targetBrightness` is zero the light should be turned off.
- If `options.powerOn` is `true` the light should be powered on.
- `options.duration` should be respected if the light supports fading.

Arguments

- **targetBrightness** (*number*) – The *percentage* the brightness should be.
- **options** – Options for changing the brightness. Two options are available, `duration` (of type *duration*) which is the requested duration of the change and `powerOn` (of type *boolean*) which indicates if the power should be switched on if the thing is off.

Returns Promise if change is asynchronous.

Example:

```
changeBrightness(targetBrightness, options) {
  const duration = options.duration.ms;
  const shouldPowerOn = options.powerOn;

  return ...
}
```

6.6.3 Implementing capability

In addition to updating the brightness whenever it changes externally as outlined in the *brightness*-capability. The method `changeBrightness` needs to be implemented.

```
const { Light, Dimmable } = require('abstract-things/lights');

class Example extends Light.with(Dimmable) {

  changeBrightness(targetBrightness, options) {
    // Duration to use if this light supports fading
    const duration = options.duration.ms;

    // If the light should be powered on if it is off
    const shouldPowerOn = options.powerOn;

    // Lazy way to handle turning the light on if is switchable
    let promise;
    if(shouldPowerOn && ! this.state.power) {
      promise = this.turnOn();
    } else if(brightness <= 0) {
      promise = this.turnOff();
    } else {
      promise = Promise.resolve();
    }

    // Then actually change the brightness
    return promise
      .then(() => actuallyChangeBrightness(...))
      .then(() => this.updateBrightness(targetBrightness));
  }

}
```

6.7 `cap:colorable` - coloring of lights

Capability used for lights that can be colored.

```
if(thing.matches('type:light', 'cap:colorable')) {
  console.log('Current color', await thing.color());

  // Set the color
  await thing.color('red');
}
```

6.7.1 API

`color` ([*color*[, *duration*]])

Get the current color or change the color of the light.

Arguments

- **`color`** – Optional *color* to set. The color can be specified in many formats, hex values such as `#00ff00`, color names such as `red` and `blue`, and color temperatures such as `4000K` or `overcast`.
- **`duration`** (*`Duration`*) – Optional *duration* to perform change in brightness over. Supported when the light has the *fading*-capability.

Returns Promise that resolves to the current or set color.

Example:

```
// Get the current color
const currentColor = await thing.color();

// Change color
const newColor = await thing.color('4000K');

// Change color over 2 seconds
await thing.color('#00ffff', '2s');
```

6.7.2 Events

`colorChanged`

Color has changed. Payload will be the new *color*.

```
thing.on('colorChanged', color => console.log('Color is now', color));
```

6.7.3 Protected methods

`updateColor` (*color*)

Update the current color of the light. Should be called whenever a change in color occurs for the light. If the color set has changed this will emit the `color` event.

Arguments

- **`color`** – The *color* of the light.

```
this.updateColor('#ff00aa');

const { color } = require('abstract-things/values');
this.updateColor(color.rgb(255, 0, 170));
```

`changeColor` (*color*, *options*)

Abstract. Change the *color* of the light. Implementation should support the following:

- `color` should be converted to something supported by the light.
- `options.duration` should be respected if the light supports fading.

Arguments

- **`color`** – The new *color* of the light. The colorspace of the light can be be anything, but is most commonly temperatures or rgb-values.
- **`options`** – Options for changing the color. The only option available is `duration` which indicates amount of time the change should occur over.

Returns Promise if change is asynchronous.

6.7.4 Implementing capability

Implementations should call `updateColor` whenever the color of the light changes. `changeColor` needs to be implemented and will be called whenever a color change is requested. *color:temperature* and *color:full* should be implemented to indicate the type of color supported.

```
const { Light, Colorable, ColorFull } = require('abstract-things/lights');
const { color } = require('abstract-things/values');

class Example extends Light.with(Colorable, ColorFull) {

  initCallback() {
    return super.initCallback()
      .then(() => this.updateColor(color.rgb(0, 0, 0));
  }

  changeColor(color, options) {
    // Convert color to RGB colorspace
    const rgb = color.rgb;

    return setColorSomehow(rgb, options.duration);
  }
}
```

6.8 `cap:color:temperature` - light supports temperature

Capability used to mark lights that support setting color temperature natively.

```
if(thing.matches('cap:color:temperature')) {
  console.log('Range is', thing.colorTemperatureRange);
}
```

6.8.1 API

`colorTemperatureRange`

Get the range of temperatures this color supports.

Returns Object with `min` and `max` in Kelvin.

Example:

```
console.log('Min temperature:', thing.colorTemperatureRange.min);
console.log('Max temperature:', thing.colorTemperatureRange.max);
```

6.8.2 Events

`colorTemperatureRangeChanged`
The range of temperature the light supports has changed.

```
thing.on('colorTemperatureRangeChanged', range => console.log('Range is now',␣
↪range));
```

6.8.3 Protected methods

`updateColorTemperatureRange` (*min*, *max*)
Set the color temperature range the light support.

Arguments

- **min** (*number*) – The minimum color temperature in Kelvin.
- **max** (*number*) – The maximum color temperature in Kelvin.

6.8.4 Implementing capability

Implementors of this capability should call `setColorTemperatureRange` either in the constructor or `initCallback`.

Example:

```
const { Light, ColorTemperature } = require('abstract-things/lights');

class Example extends Light.with(ColorTemperature) {

  constructor() {
    super();

    this.updateColorTemperatureRange(2000, 5000);
  }

}
```

6.9 `cap:color:full` - light supports full range of color

Capability used to mark lights that support setting any color.

```
if(thing.matches('type:light', 'cap:color:full')) {
  // This light supports any color
}
```

6.9.1 Implementing capability

Implementors of this capability have no special requirements placed upon them.

Example:

```
const { Light, ColorFull } = require('abstract-things/lights');

class Example extends Light.with(ColorFull) {

  constructor() {
    super();
  }

}
```

CHAPTER 7

Sensors

The type `sensor` is used to mark things that read one or more values.

```
if(thing.matches('type:sensor') {
  console.log('Sensor values:', thing.values());
}

if(thing.matches('type:sensor', 'cap:temperature')) {
  console.log('Temperature:', thing.temperature());
}
```

7.1 `cap:atmospheric-pressure` - read atmospheric pressure

This capability is used to mark sensors that report the atmospheric pressure.

```
if(thing.matches('cap:atmospheric-pressure')) {
  console.log('Atmospheric pressure:', await thing.atmosphericPressure());
}
```

7.1.1 API

atmosphericPressure()

Get the current atmospheric pressure.

Returns Promise that resolves to the atmospheric pressure as a *pressure*.

Example:

```
console.log('Atmospheric pressure:', await thing.atmosphericPressure());
```

7.1.2 Events

`atmosphericPressureChanged`
The atmospheric pressure has changed.

Example:

```
thing.on('atmosphericPressureChanged', value => console.log('Pressure changed to:
↪', value));
```

7.1.3 Protected methods

`updateAtmosphericPressure`(*value*)
Update the current atmospheric pressure. Should be called whenever a change in atmospheric pressure is detected.

Arguments

- **`value`** – The new atmospheric pressure.

Example:

```
// Defaults to pascals
this.updateAtmosphericPressure(101325);

// pressure value can be used to use hPa (= millibar), bar, psi or mmHg
const { pressure } = require('abstract-things/values');
this.updateAtmosphericPressure(pressure(1, 'atm'));
this.updateAtmosphericPressure(pressure(1013, 'hPa'));
```

7.1.4 Implementing capability

Implementors of this capability should call `updateAtmosphericPressure` whenever the atmospheric pressure changes.

```
const { Sensor, AtmosphericPressure } = require('abstract-things/sensors');

class Example extends Sensor.with(AtmosphericPressure) {

  constructor() {
    super();

    this.updateAtmosphericPressure(101325);
  }

}
```

7.2 `cap:carbon-dioxide` - read carbon dioxide level

This capability is used to mark sensors that report their carbon dioxide level as PPM (parts per million). The value is reported as a *number*.

```
if(thing.matches('cap:carbon-dioxide')) {
  console.log('Carbon dioxide:', await thing.carbonDioxide());
}
```

7.2.1 API

carbonDioxide()

Get the current carbon dioxide levels as PPM.

Returns Promise that resolves to the current value as a *number*.

Example:

```
console.log('CO2 is:', await thing.carbonDioxide());
```

co2()

Get the current carbon dioxide levels as PPM. Reported as a *number*.

Returns Promise that resolves to the current value as a *number*.

Example:

```
console.log('CO2 is:', await thing.co2());
```

7.2.2 Events

carbonDioxideChanged

The carbon dioxide level has changed. Payload is the new PPM as a *number*.

Example:

```
thing.on('carbonDioxideChanged', v => console.log('Changed to:', v));
```

7.2.3 Protected methods

updateCarbonDioxide(*value*)

Update the current carbon dioxide level. Should be called whenever a change in PPM is detected.

Arguments

- **value** – The new PPM value. Will be converted to a *number*.

Example:

```
this.updateCarbonDioxide(389);
```

7.2.4 Implementing capability

Implementors of this capability should call `updateCarbonDioxide` whenever the PPM of carbon dioxide changes.

```
const { Sensor, CarbonDioxide } = require('abstract-things/sensors');

class Example extends Sensor.with(CarbonDioxide) {
  constructor() {
    super();

    this.updateCarbonDioxide(390);
  }

}
```

7.3 `cap:carbon-monoxide` - read carbon monoxide level

This capability is used to mark sensors that report their carbon monoxide level as PPM (parts per million). The value is reported as a *number*.

```
if(thing.matches('cap:carbon-monoxide')) {
  console.log('Carbon monoxide:', thing.carbonMonoxide);
}
```

7.3.1 API

`carbonMonoxide``()`

Get the current carbon monoxide levels as PPM.

Returns Promise that resolves to the current value as a *number*.

```
console.log('CO is:', await thing.carbonMonoxide());
```

`co``()`

Get the current carbon monoxide levels as PPM.

Returns Promise that resolves to the current value as a *number*.

```
console.log('CO is:', await thing.co());
```

7.3.2 Events

`carbonMonoxideChanged`

The carbon monoxide level has changed. Payload is the new PPM as a *number*.

Example:

```
thing.on('carbonMonoxideChanged', v => console.log('Changed to:', v));
```

7.3.3 Protected methods

`updateCarbonMonoxide``(`*value*`)`

Update the current carbon monoxide level. Should be called whenever a change in PPM is detected.

Arguments

- **`value`** – The new PPM value. Will be converted to a *number*.

Example:

```
this.updateCarbonMonoxide(0);
```

7.3.4 Implementing capability

Implementors of this capability should call `updateCarbonMonoxide` whenever the PPM of carbon monoxide changes.

```
const { Sensor, CarbonMonoxide } = require('abstract-things/sensors');

class Example extends Sensor.with(CarbonMonoxide) {

  constructor() {
    super();

    this.updateCarbonMonoxide(0);
  }

}
```

7.4 `cap:contact` - contact sensing

This capability is used to mark sensors that report a wether contact is detected, such as for door and window sensors that detect if the door or window is open.

```
if(thing.matches('cap:contact')) {
  console.log('Has contact:', await thing.contact());
}
```

7.4.1 API

`contact`()

Boolean representing if the sensor is currently detecting contact.

Returns Promise that resolves to if the sensor is detecting contact.

Example:

```
if(await thing.contact()) {
  console.log('Thing has detected contact');
}
```

`isOpen`()

Boolean representing if the sensor is currently open (not detecting contact).

Returns Promise that resolves to if the sensor is in an open state.

Example:

```
console.log('Is open:', await thing.isOpen());
```

`isClosed`()

Boolean representing if the sensor is currently closed (detecting contact).

Returns Promise that resolves to if the sensir is in a closed state.

Example:

```
console.log('Is closed:', await thing.isClosed());
```

7.4.2 Events

`contactChanged`
The contact value has changed. Payload is the new contact state as a *boolean*.

Example:

```
thing.on('contactChanged', v => console.log('Contact is now:', c));
```

`opened`
The sensor has detected it is does not have contact and is now opened.

Example:

```
thing.on('opened', v => console.log('Sensor is now open'));
```

7.4.3 Protected methods

`updateContact`(*value*)
Update if the sensor is currently detecting contact.

Arguments

- **`value`** – The new contact status as a *boolean*.

Example:

```
// Set the sensor to open
this.updateContact(false);
```

7.4.4 Implementing capability

Implementors of this capability should call `updateContact` whenever the contact state changes.

```
const { Sensor, Contact } = require('abstract-things/sensors');

class Example extends Sensor.with(Contact) {

  constructor() {
    super();

    this.updateContact(true);
  }

}
```

7.5 `cap:illuminance` - read illuminance

This capability is used to mark sensors that report *illuminance*. This is commonly used for sensors that read light levels.

```
if(thing.matches('cap:illuminance')) {
  console.log('Light level:', await thing.illuminance());
}
```

7.5.1 API

`illuminance`()
Get the current *illuminance*.

Returns Promise that resolves to the current *illuminance*.

Example:

```
const lightLevel = await thing.illuminance();
console.log('Light level:', lightLevel.lux);
```

7.5.2 Events

`illuminanceChanged`
The illuminance has changed. Payload is the new *illuminance*.

Example:

```
thing.on('illuminanceChanged', v => console.log('Changed to:', v));
```

7.5.3 Protected methods

`updateIlluminance`(*value*)
Update the current illuminance level. Should be called whenever a change in is detected.

Arguments

- **`value`** – The new illuminance. Will be converted to *illuminance*, the default conversion uses lux.

Example:

```
this.updateIlluminance(20);
```

7.5.4 Implementing capability

Implementors of this capability should call `updateIlluminance` whenever the detected light level changes.

```
const { Sensor, Illuminance } = require('abstract-things/sensors');

class Example extends Sensor.with(Illuminance) {

  constructor() {
    super();

    this.updateIlluminance(10);
  }

}
```

7.6 `cap:motion` - motion sensing

This capability is used to mark sensors that monitor movement.

```
if(thing.matches('cap:motion')) {
  console.log('Detected motion:', await thing.motion());

  thing.on('movement', () => console.log('Motion detected'));
  thing.on('inactivity', () => console.log('Inactivity detected'));
}
```

7.6.1 API

`motion`()

Get the motion status.

Returns Promise that resolves to a *boolean* indicating if movement is currently detected.

Example:

```
console.log('Motion is:', thing.motion);
```

7.6.2 Events

`motionChanged`

The current motion status has changed.

```
thing.on('motionChanged', value => console.log('Motion changed to:', value));
```

`movement`

Emitted when movement has been detected and `motion` changes to `true`.

```
thing.on('movement', () => console.log('Movement detected'));
```

`inactivity`

Emitted when movement is no longer detected and `motion` changes to `false`.

```
thing.on('inactivity', () => console.log('Movement no longer detected'));
```

7.6.3 Protected methods

`updateMotion`(*value*[, *autoIdleTimeout*])

Update the current motion status.

Arguments

- **value** (*boolean*) – The motion status, `true` if motion detected otherwise `false`.
- **autoIdleTimeout** (*duration*) – Optional duration to switch back the motion status to `false`.

Example:

```
this.updateMotion(false);

// Set motion to true and automatically switch back after 20 seconds
this.updateMotion(true, '20s');
```

7.6.4 Implementing capability

Implementors of this capability should call `updateMotion` if motion is detected. Implementations may choose between using automatic timeouts for switching motion back to `false` or managing the switchin on their own.

```
const { Sensor, Motion } = require('abstract-things/sensors');

class Example extends Sensor.with(Motion) {

  constructor() {
    super();

    this.updateMotion(true, '1m');
  }

}
```

7.7 `cap:pm2.5` - read PM2.5 density (air quality)

This capability is used to mark sensors that monitor fine particulate matter (PM) of up to 2.5 micrometers (μm).

```
if(thing.matches('cap:pm2.5')) {
  console.log('PM2.5:', await thing.pm2_5());
}
```

7.7.1 API

pm2_5()

Get the current PM2.5 as micrograms per cubic meter (μg/m^3). Value is a *number*.

Returns The current value as micrograms per cubic meter (μg/m^3). Value is a *number*.

Example:

```
console.log('PM2.5:', await thing.pm2_5());
```

'pm2.**5**'()

Get the current PM2.5 as micrograms per cubic meter (μg/m^3). Value is a *number*.

Returns The current value as micrograms per cubic meter (μg/m^3). Value is a *number*.

Example:

```
console.log('PM2.5:', await thing['pm2.5']());
```

7.7.2 Events

pm2.**5Changed**

The PM2.5 has changed. Payload is a *number* with the new PM2.5 as micrograms per cubic meter (μg/m^3).

Example:

```
thing.on('pm2.5Changed', v => console.log('Changed to:', v));
```

7.7.3 Protected methods

updatePM2_5(*value*)

Update the current PM2.5 as micrograms per cubic meter (μg/m^3). Should be called whenever a change is detected.

Arguments

- **value** – The new PM2.5 value. Will be converted to a *number*.

Example:

```
this.updatePM2_5(10);
```

7.7.4 Implementing capability

Implementors of this capability should call `updatePM2_5` whenever the detected PM2.5 changes.

```
const { Sensor, PM2_5 } = require('abstract-things/sensors');

class Example extends Sensor.with(PM2_5) {

  constructor() {
    super();

    this.updatePM2_5(10);
  }

}
```

7.8 `cap:pm10` - read PM10 density (air quality)

This capability is used to mark sensors that monitor particulate matter (PM) between 2.5 and 10 micrometers (μm).

```
if(thing.matches('cap:pm10')) {
  console.log('PM10:', await thing.pm10());
}
```

7.8.1 API

pm10()

Get the current PM10 as micrograms per cubic meter (μg/m^3).

Returns The current value as micrograms per cubic meter (μg/m^3). Value is a *number*.

Example:

```
console.log('PM10:', await thing.pm10());
```

7.8.2 Events

pm10Changed

The PM10 has changed. Payload is a *number* with the new PM10 as micrograms per cubic meter (μg/m^3).

Example:

```
thing.on('pm10Changed', v => console.log('Changed to:', v));
```

7.8.3 Protected methods

updatePM10(*value*)

Update the current PM10 as micrograms per cubic meter (μg/m^3). Should be called whenever a change is detected.

Arguments

- **value** – The new PM10 value. Will be converted to a *number*.

Example:

```
this.updatePM10(5);
```

7.8.4 Implementing capability

Implementors of this capability should call `updatePM10` whenever the detected PM10 changes.

```
const { Sensor, PM10 } = require('abstract-things/sensors');

class Example extends Sensor.with(PM10) {

  constructor() {
    super();

    this.updatePM10(5);
  }

}
```

7.9 `cap:power-consumed` - read power consumed

This capability is used to mark sensors that report power consumed by something.

```
if(thing.matches('cap:power-consumed')) {
  const powerConsumed = await thing.powerConsumed();
  console.log('Power consumed:', powerConsumed.wattHours);
}
```

7.9.1 API

powerConsumed()

Get the current amount of power consumed. .

Returns Promise that resolves to the amount of power consumed as *energy*.

Example:

```
const powerConsumed = await thing.powerConsumed();
console.log('Power consumed:', powerConsumed.wattHours);
```

7.9.2 Events

powerConsumedChanged

The amount of power consumed has changed. Payload is the power consumed as *energy*.

Example:

```
thing.on('powerConsumedChanged', v => console.log('Changed to:', v));
```

7.9.3 Protected methods

updatePowerConsumed(*value*)

Update the power consumed. Should be called whenever a change is detected.

Arguments

- **value** – The new amount of power consumed, as *energy*. The default unit is joules.

Example:

```
const { energy } = require('abstract-things/values');
this.updatePowerConsumed(energy(0.5, 'wh'));
```

7.9.4 Implementing capability

Implementors of this capability should call `updatePowerConsumed` whenever the power consumed changes.

```
const { Sensor, PowerConsumed } = require('abstract-things/sensors');

class Example extends Sensor.with(PowerConsumed) {

  constructor() {
    super();

    this.updatePowerConsumed(10); // Joules
  }
}
```

7.10 `cap:power-load` - read the current power load

This capability is used to mark sensors that report power load, that is the *power* currently being used.

```
if(thing.matches('cap:power-load')) {
  const powerLoad = await thing.powerLoad();
  console.log('Power load:', powerLoad.watts);
}
```

7.10.1 API

`powerLoad``()`

Get the current amount of power being used.

Returns Promise that resolves to the current amount of power used as a *power*.

Example:

```
const powerLoad = await thing.powerLoad();
console.log('Power load:', powerLoad.watts);
```

7.10.2 Events

`powerLoadChanged`

The amount of power being used has changed. Payload is the power load as *power*.

Example:

```
thing.on('powerLoadChanged', v => console.log('Changed to:', v));
```

7.10.3 Protected methods

`updatePowerLoad``(`*value*`)`

Update the power load. Should be called whenever a change is detected.

Arguments

- **`value`** – The new amount of power being used, as *power*. The default unit is watts.

Example:

```
this.updatePowerLoad(5);
```

7.10.4 Implementing capability

Implementors of this capability should call `updatePowerLoad` whenever the power load changes.

```
const { Sensor, PowerLoad } = require('abstract-things/sensors');

class Example extends Sensor.with(PowerLoad) {

  constructor() {
    super();

    this.updatePowerLoad(10);
  }

}
```

7.11 `cap:relative-humidity` - read humidity of air

This capability is used to mark sensors that report the relative humidity of the air.

```
if(thing.matches('cap:relative-humidity')) {
  console.log('RH:', await thing.relativeHumidity());
}
```

7.11.1 API

`relativeHumidity`()

Get the current relative humidity as a *percentage*.

Returns Promise that resolves to the current relative humidity as a *percentage*.

Example:

```
console.log('RH:', await thing.relativeHumidity());
```

7.11.2 Events

`relativeHumidityChanged`

The relative humidity has changed. Payload is the new humidity as a *percentage*.

Example:

```
thing.on('relativeHumidityChanged', v => console.log('Changed to:', v));
```

7.11.3 Protected methods

`updateRelativeHumidity`(*value*)

Update the relative humidity. Should be called whenever a change is detected.

Arguments

- **`value`** – The new relative humidity. Will be converted to a *percentage*.

Example:

```
this.updateRelativeHumidity(32);
```

7.11.4 Implementing capability

Implementors of this capability should call `updateRelativeHumidity` whenever the relative humidity changes.

```
const { Sensor, RelativeHumidity } = require('abstract-things/sensors');

class Example extends Sensor.with(RelativeHumidity) {

  constructor() {
    super();

    this.updateRelativeHumidity(56);
  }

}
```

7.12 `cap:temperature` - read temperature

This capability is used to mark sensors that report a *temperature*.

```
if(thing.matches('cap:temperature')) {
  const temperature = await thing.temperature();
  console.log('Temperature:', temperature.celsius);
}
```

7.12.1 API

`temperature`()

Get the current *temperature*.

Returns Promise that resolves to the current *temperature*.

Example:

```
console.log('Temperature is:', thing.temperature);
```

7.12.2 Events

`temperatureChanged`

The temperature has changed. Payload is the new *temperature*.

Example:

```
thing.on('temperatureChanged', temp => console.log('Temp changed to:', temp));
```

7.12.3 Protected methods

`updateTemperature`(*value*)

Update the current temperature. Should be called whenever a change in temperature was detected.

Arguments

- **`value`** – The new temperature. Will be converted to a *temperature*, the default conversion uses degrees Celsius.

Example:

```
// Defaults to Celsius
this.updateTemperature(20);

// temperature value can be used to use Fahrenheit (or Kelvin)
const { temperature } = require('abstract-things/values');
this.updateTemperature(temperature(45, 'F'));
```

7.12.4 Implementing capability

Implementors of this capability should call `updateTemperature` whenever the temperature changes.

```
const { Sensor, Temperature } = require('abstract-things/sensors');

class Example extends Sensor.with(Temperature) {

  constructor() {
    super();

    this.updateTemperature(22);
  }

}
```

7.13 `cap:voltage` - read voltage of something

This capability is used to mark sensors that report the *voltage* of something.

```
if(thing.matches('cap:voltage')) {
  const voltage = await thing.voltage();
  console.log('Voltage:', voltage.volts);
}
```

7.13.1 API

`voltage`

Get the current *voltage*.

Returns Promise that resolves to the current *voltage*.

Example:

```
const voltage = await thing.voltage();
console.log('Voltage:', voltage.volts);
```

7.13.2 Events

`voltageChanged`
The voltage has changed. Payload is the new voltage as a *voltage*.

Example:

```
thing.on('voltageChanged', v => console.log('Changed to:', v));
```

7.13.3 Protected methods

`updateVoltage`(*value*)
Update the *voltage*. Should be called whenever a change is detected.

Arguments

- **`value`** – The new voltage. Will be converted to a *voltage* with the default unit being volts.

Example:

```
this.updateVoltage(12);
```

7.13.4 Implementing capability

Implementors of this capability should call `updateRelativeHumidity` whenever the relative humidity changes.

```
const { Sensor, Voltage } = require('abstract-things/sensors');

class Example extends Sensor.with(Voltage) {

  constructor() {
    super();

    this.updateVoltage(230);
  }

}
```

CHAPTER 8

Climate

Climate types and capabilities are provided for things that have to do with the climate of a space, such as air purifiers, humidifiers, fans and thermostats.

8.1 `cap:target-humidity` - read the target humidity

The `target-humidity` capability is used by things such as humidifers and *dehumidifiers* that support stopping when a certain target humidity is reached. Some things may also support setting the target humidity via *adjustable-target-humidity*.

```
if(thing.matches('cap:target-humidity')) {
  const humidity = await thing.targetHumidity();
  console.log('Target humidity:', humidity);
}
```

8.1.1 API

`targetHumidity``()`
Get the current target humidity.

Returns Promise that resolves to the current target humidity as a *percentage*.

Example:

```
const target = await thing.targetHumidity();
```

8.1.2 Events

`targetHumidityChanged`
The current target humidity has changed. Payload will be the new target humidity as a *percentage*.

Example:

```
thing.on('targetHumidityChanged', th => console.log('Target:', th));
```

8.1.3 Protected methods

`updateTargetHumidity`(*target*)

Update the current target humidity.

Arguments

- **`target`** (*`percentage`*) – The new target humidity as a *percentage*.

Example:

```
this.updateTargetHumidity(40);
this.updateTargetHumidity('55%');
```

8.1.4 Implementing capability

When implementing this capability the implementor needs to call `updateTargetHumidity` whenever a change in target humidity is detected.

```
const { Thing } = require('abstract-things');
const { TargetHumidity } = require('abstract-things/climate');

class Example extends Thing.with(TargetHumidity) {

}
```

8.2 `cap:adjustable-target-humidity` - change the target humidity

The `adjustable-target-humidity` capability is an extension to *target-humidity* that in addition to reporting the target humidity also supports setting it.

```
if(thing.matches('cap:changeable-target-humidity')) {
  const humidity = await thing.targetHumidity();
  console.log('Target humidity:', humidity);

  // Set the target humidity
  await thing.targetHumidity(20);
}
```

8.2.1 API

`targetHumidity`([*target*])

Get or set the current target humidity.

Arguments

- **target** (*percentage*) – Optional target humidity to set as a *percentage*. If specified the thing will update the target humidity.

Returns Promise that resolves to the current or set target humidity as a *percentage*.

Example:

```
const target = await thing.targetHumidity();

await thing.targetHumidity(55);
```

setTargetHumidity(*target*)

Set the target humidity.

Arguments

- **target** (*percentage*) – The target humidity as a *percentage*.

Returns Promise that resolves to the set target humidity.

Example:

```
await thing.setTargetHumidity(40);
```

8.2.2 Protected methods

changeTargetHumidity(*target*)

Abstract. Change the current target humidity.

Arguments

- **target** (*percentage*) – The new target humidity as a *percentage*.

Returns Promise if asynchronous.

Example:

```
changeTargetHumidity(target) {
  return actuallySetTargetHumidity(target);
}
```

8.2.3 Implementing capability

When implementing this capability the implementor needs to call `updateTargetHumidity` whenever a change in target humidity is detected. The `changeTargetHumidity` method must also be implemented.

```
const { Thing } = require('abstract-things');
const { AdjustableTargetHumidity } = require('abstract-things/climate');

class Example extends Thing.with(AdjustableTargetHumidity) {

  changeTargetHumidity(target) {
    return actuallySetTargetHumidity(target);
  }

}
```

8.3 `type:air-purifier` - Air purifiers

Air purifiers are appliances that filter and purify the air. Commonly used with the *switchable-power* and *switchable-mode* capabilities.

```
if(thing.matches('type:air-purifier')) {
  // The thing is an air purifier
}
```

8.3.1 Implementing type

```
const { AirPurifier } = require('abstract-things/climate');

class Example extends AirPurifier.with(...) {

}
```

8.4 `type:humidifer` - Humidifiers

Humidifiers are appliances that increase the humidity of the air. Many humidifers will support *switchable-power* so that they can be switched on or off. Some implement *switchable-mode* to support different modes, such as switching between automatic and manual modes.

```
if(thing.matches('type:humidifier')) {
  // The thing is a humidifier
}
```

8.4.1 Implementing type

```
const { Humidifier } = require('abstract-things/climate');

class Example extends Humidifier.with(...) {

}
```

8.5 `type:dehumidifier` - Dehumidifers

Dehumidifiers are appliances that decrease the humidity of the air. Many dehumidifers will support *switchable-power* so that they can be switched on or off. Some implement *switchable-mode* to support different modes, such as switching between automatic and manual modes.

```
if(thing.matches('type:dehumidifier')) {
  // The thing is a dehumidifier
}
```

8.5.1 Implementing type

```
const { Dehumidifier } = require('abstract-things/climate');

class Example extends Dehumidifier.with(...) {

}
```

8.6 `type:vacuum` - Vacuum cleaners

Vacuum cleaners are used as a type for both autonomous and non-autonomous cleaners.

```
if(thing.matches('type:vacuum')) {
  // The thing is a vacuum
}
```

8.6.1 Implementing type

```
const { Vacuum } = require('abstract-things/climate');

class Example extends Vacuum.with(...) {

}
```

CHAPTER 9

Electrical

Electrical types and capabilities for power plugs, electrical outlets and sockets and more. The most common type is *power-outlet* which is used to represent a single generic outlet/socket. Such a power outlet may be a child of other types such as the individual outlets in a *power strip* or a *wall outlet*.

9.1 `type:power-outlet` - Power outlets

Things marked with `power-outlet` represent a single outlet that can take a single plug. Outlets can be both standalone and children of another thing, such as a *power strip* or *wall outlet*.

The *power* and *switchable-power* capability is commonly used with outlets to switch the power of the outlet. Outlets can also be *sensors* if they report *power load* or *power consumption*.

```
if(thing.matches('type:power-outlet')) {
  // This is a power outlet

  if(thing.matches('cap:switchable-power')) {
    // And it also supports power switching
    thing.turnOn()
      .then(...)
      .catch(...);
  }
}
```

9.1.1 Implementing type

```
const { PowerOutlet } = require('abstract-things/electrical');

class Example extends PowerOutlet.with(...) {

}
```

9.2 `type:power-channel` - Power channels

Things marked with `power-channel` represent a single channel of power. Power channels are usually virtual, such as individual power lines in a *power switch*.

The *power* and *switchable-power* capability is commonly used with channels to support switch the power. Channels can also be *sensors* if they report *power load* or *power consumption*.

```
if(thing.matches('type:power-channel')) {
  // This is a power channel

  if(thing.matches('cap:switchable-power')) {
    // And it also supports power switching
    thing.turnOn()
      .then(...)
      .catch(...);
  }
}
```

9.2.1 Implementing type

```
const { PowerChannel } = require('abstract-things/electrical');

class Example extends PowerChannel.with(...) {

}
```

9.3 `type:power-strip` - Power strips

Things marked with `power-strip` represent a power strip with several outlets. Power strips can expose their individual outlets as children, in which case they implement the *children* capability.

```
if(thing.matches('type:power-strip')) {
  // This is a power strip

  if(thing.matches('cap:children')) {
    // Each outlet in the strip is available as a child
    const firstOutlet = thing.getChild('1'); // depends on the implementation
  }
}
```

9.3.1 Implementing type

Without any children:

```
const { PowerStrip } = require('abstract-things/electrical');

class Example extends PowerStrip.with(...) {

}
```

With outlets as children:

```
const { Children } = require('abstract-things');
const { PowerStrip, PowerOutlet } = require('abstract-things/electrical');

class Example extends PowerStrip.with(Children, ...) {

  constructor() {
    super();

    this.addChild(new ExampleOutlet(this, 1));
    this.addChild(new ExampleOutlet(this, 2));
  }

}

class ExampleOutlet extends PowerOutlet.with(...) {

  constructor(parent, idx) {
    this.parent = parent;
    this.id = parent.id + ':' + idx;
  }

}
```

9.4 `type:power-plug` - Power plugs

Things marked with `power-plug` are plugs that can be plugged in to an outlet. Most plugs are also *power outlets* in that appliances can be plugged in to them.

The *power* and *switchable-power* capability is commonly used with plugs to switch the power of the outlet of the plug. They can also be *sensors* if they report *power load* or *power consumption*.

```
if(thing.matches('type:power-plug')) {
  // This is a power plug

  if(thing.matches('cap:switchable-power')) {
    // And it also supports power switching
    thing.turnOn()
      .then(...)
      .catch(...);
  }
}
```

9.4.1 Implementing type

```
const { PowerPlug, PowerOutlet } = require('abstract-things/electrical');

class Example extends PowerPlug.with(PowerOutlet, ...) {

}
```

9.5 `type:wall-outlet` - Wall outlets

The `wall-outlet` type is used to mark things that represent a wall mounted power outlet. Wall outlets like *power strips* can expose their individual outlets as *children*.

```
if(thing.matches('type:wall-outlet')) {
  // This is a wall outlet

  if(thing.matches('cap:children')) {
    // Each outlet is available as a child
    const firstOutlet = thing.getChild('1'); // depends on the implementation
  }
}
```

9.5.1 Implementing type

Without any children:

```
const { WallOutlet } = require('abstract-things/electrical');

class Example extends WallOutlet.with(...) {

}
```

With outlets as children:

```
const { Children } = require('abstract-things');
const { WallOutlet, PowerOutlet } = require('abstract-things/electrical');

class Example extends WallOutlet.with(Children, ...) {

  constructor() {
    super();

    this.addChild(new ExampleOutlet(this, 1));
    this.addChild(new ExampleOutlet(this, 2));
  }

}

class ExampleOutlet extends PowerOutlet.with(...) {

  constructor(parent, idx) {
    this.parent = parent;
    this.id = parent.id + ':' + idx;
  }

}
```

9.6 `type:power-switch` - Power switches

Things marked with `power-switch` are switches that control something. Switches commonly control *power outlets*, *power channels* and *lights*.

```
if(thing.matches('type:power-switch')) {
  // This is a power switch
}
```

9.6.1 Implementing type

```
const { PowerSwitch } = require('abstract-things/electrical');

class Example extends PowerSwitch.with(...) {

}
```

9.7 `type:wall-switch` - Wall switches

The `wall-switch` type is used to mark things that represent a wall mounted power switch. A wall switch is commonly used to control *lights* or *power channels*.

```
if(thing.matches('type:wall-switch')) {
  // This is a wall switch

  if(thing.matches('cap:children')) {
    // Lights or power channels available as children
    const firstChild= thing.getChild('1'); // depends on the implementation
  }
}
```

9.7.1 Implementing type

Without any children:

```
const { WallSwitch } = require('abstract-things/electrical');

class Example extends WallOutlet.with(...) {

}
```

With power channels as children:

```
const { Children } = require('abstract-things');
const { WallSwitch, PowerChannel } = require('abstract-things/electrical');

class Example extends WallSwitch.with(Children, ...) {

  constructor() {
    super();

    this.addChild(new ExampleChild(this, 1));
    this.addChild(new ExampleChild(this, 2));
  }

}
```

```
class ExampleChild extends PowerChannel.with(...) {

  constructor(parent, idx) {
    this.parent = parent;
    this.id = parent.id + ':' + idx;
  }

}
```

Index