FactoryFns

Factory functions take user-input and create a random value, that gets directly returned as return-value.

```
factory<T>(...inputParams: any[]) => T
```

Skip forward to FactoryFn overview

FactoryFn vs. ValueFn?

Both, `FactoryFns` and `ValueFns` take arguments and turn it into a randomized value, with a specified type, that is restricted to given user-input. So what is the point in distinguishing it?

The answer is: The way they are used. That's why the signature of both differs:

- ValueFn: `(...args: any[]) => Buildable<ValueFn>`
- FactoryFn: `(...args: any[]) => any`

Factory functions create a randomized value and immediately returns it, so that it can be used within a processor function for example. It won't return a `Buildable`, that continiously delivers random data, but it directly returns *a single random value*, that is based on the input parameters given. This behavior is often important in situations where you need a randomized value **that is dependent on another value**.

So the main difference between value functions and factories is the place where you use them. Value functions are used on model's properties, in order to generate randomized values over and over, while factory functions are used within processors in order to generate a single random value to be set once.

This might sound really theoretical, so let's see it by example!

Understanding by Example

Consider modelling an appointment having a `startTime` and `endTime`. When defining the model, we have to ensure that the `endTime` is later than the `startTime`. In other words, the `endTime` is (of course) *dependent* on `startTime`.

There is no (good) way to express that via a value function, since it just generates random `Date` objects. How could we actually know if the generated time was after the `startTime`? In this case a factory can help you out, *creating a valid `endTime` based on a given `startTime`*.

Here is the proof:

The Bad Way: Modelling an appointment's time-bounds with ValueFns

What about that model:

```
const $BadAppointment = model({
  // random time between 17:55 and 20:00.
  startTime: time(['17:55'], ['20:00']),
  // random time between 20:01 and 23:59:59
```

```
  endTime: time(['20:01'], ['23:59:59']),
});
```

We used time (which is a ValueFn) to model out both, start- and endtime.

But why is it bad?

The first thing is, that touching either `startTime` or `endTime` will possibly break the other one, leading to invalid time bounds. In this example, the model is fairly simple, but what if the model grows and as worst case, the `startTime` and `endTime` is located in different submodels? Besides that, splitting the times as seen above will not work anymore, if appointments are allowed to be *at any time*:

```
model({
  startTime: time(), // 00:00:00:000 to 23:59:59:999
  endTime: ???,
})
```

Next, we have no control about the length of an appointment. How could we ever say, that the maximal lenght of an appointment is two hours? Or an appointment is not allowed to have a length of 30 seconds? In simple words: there is no way.

And finally, there is no explicit dependency drawn from `endTime` to `startTime`. Another developer would have no chance to see, that the `endTime` must be aligned to the `startTime`. Again, in more complex models this can lead to bugs that are really hard to find.

Fixing it: Modelling an appointment's time-bounds with FactoryFns

To fix it we use the duration-factory, which accepts a base-date value and a minimum and maximum parameter. Since it returns the created random value directly, we can use this function within a map-processor, attached to a ref-builder:

```
const $Appointment = model({
  startTime: time(), // 00:00:00:000 to 23:59:59:999
  endTime: ref(
    'startTime',
    map(time => duration(time, [5, 'minutes'], [1, 'hours'])),
  ),
});
```

This way, we explicitly let `endTime` be dependent of `startTime` and calculate the `endTime` based on our time-bound-restrictions. Here an appointment's length is at least `5 minutes` and at most `1 hour`.

Overview

These factory functions are built into faketastic:

duration

`duration(base: Date, min: TimeInput, max: TimeInput): Date`

Takes a base date instance and adds a random duration to it then returns the result.

Note: The duration summand can also be negative as in:

`duration(someDate, [-2, 'hours'], [-1, 'minutes']);`

```
import { model, time, ref, map, duration, build } from 'faketastic';

const $Appointment = model({
  startTime: time(),
  endTime: ref(
    'startTime',
    map(time => duration(time, [5, 'minutes'], [1, 'hours'])),
  ),
});

const appointment = build($Appointment);
// => {
//  startTime: [object Date; Today, 15:42:19],
//  endTime: [object Date; Today, 16:03:52]
// }
```

Example: `ageRef` will resolve the value of `age`-property at "finalizer" build cycle.

probability

`probability(n = 0.5): boolean`

Takes a number between `0` and `1` representing the percental likelihood that the return value will be `true`, i. e. `probability(0.73)` will return `true` with a chance of 73%, whereas `probability(0)` means, that it will never return `true`; `probability(1)`means; that it will always return `true`.

```
import { probability } from 'faketastic';

const bool = probability();
// => false
```

Example: `bool` will be `true` with a chance of 50%.

randomDate

`randomDate(min: Date, max: Date): Date`

Takes two parameters defining a date-range, in which the returned, randomly created date will be.

```
import { randomDate } from 'faketastic';
// 01.01.2020 04:35:12
const min = new Date(2020, 0, 1, 4, 35, 12);
// 26.02.2020 00:00:00
const max = new Date(2020, 1, 26);

const date = randomDate(min, max);
// => 03.02.2020 14:31:22
```

Example: `date` will be a random date being between `01.01.2020 04:35:12` and `26.02.2020`.

random

```
random(min: number, max: number): number
```

Takes two parameters defining a range, in which the generated float number will be.

```
import { random } from 'faketastic';

const min = 12;
const max = 42;

const value = random(min, max);
// => 32.29183
```

Example: `index` will be a valid index, i.e. in this example a number between `0` and `4` as the `values`-array has 5 items.

randomInt

```
randomInt(min: number, max: number): number
```

Takes two parameters defining a range, in which the randomly generated number will be. Same as random but returning whole numbers, instead of floats.

```
import { randomInt } from 'faketastic';

const values = ['a', 42, null, true, () => {}];

const index = randomIndex(values);
// => 2
```

Example: `index` will be a valid index, i.e. in this example a number between `0` and `4` as the `values`-array has 5 items.

randomIndex

`randomIndex(array: any[]): number`

Takes an array and returns a random index within that array. Similiar to randomItem which returns a random item from a given array.

```
import { randomIndex } from 'faketastic';

const values = ['a', 42, null, true, () => {}];

const index = randomIndex(values);
// => 2
```

Example: `index` will be a valid index, i.e. in this example a number between `0` and `4` as the `values`-array has 5 items.

randomItem

`randomItem(array: any[]): any`

Takes an array and returns a random item from that array. Similiar to randomIndex which returns a random index from a given array.

```
import { randomItem } from 'faketastic';

const values = ['a', 42, null, true, () => {}];

const item = randomItem(values);
// => 42
```

Example: `item` will be one of the entries within the `values`-array.

Related Topics

- ValueFns
- Overview
- Getting Started