Extensibility

Faketastic strives to be as flexible and exstensible as possible, in order to be a great tool for any modelling use case you encounter. That's why you can always write your custom tools, in addition to the already built-in functionality.

Custom BuilderFns

Builder functions can have any signature in terms of input parameters, but must always return a `Buildable`. Thus their signature must match:

```
builderFn(...params: any[]): Buildable<any | ValueFn>
```

The value of the returned Buildable can be anything (static value for example) or a ValueFn. This example will be more complex than most of the others, that's why we explain the aim of our builder function first.

We want to implement an `eitherOr` builder, which takes two options from which it chooses one by chance. We must keep in mind, that the option that was chosen, ***might*** be a model again, so we also want to make sure, that the potential model gets built as well.

We start with the a rough view about the code first and go into detail afterwards:

```
// either-or.ts

import /* ... */ 'faketastic';

export function eitherOr(valueA: any, valueB: any, ...attachedFns:
AttachedFn[]): Buildable {
  // (1)
  function eitherOrImpl() {
    /* actual implementation */
  }

  // (2)
  const eitherOrInit = createProcessorFn(eitherOrImpl, 'initializer');

  // (3)
  return createBuildable(UnsetValue, [eitherOrInit, ...attachedFns]);
}
```

The first thing we do is creating an inner function named `eitherOrImpl`, which will run the actual implementation. As it is an inner function, it shares the parental scope, so we have direct access to the parent's parameters.

We will have a closer look on the implementation in a minute, but before that let's stay in the parental function. As next step, we convert the inner function into a processor function of type `initializer`, as this build cycle is a good choice for processors altering the build-tree.

Then we create a `Buildable` and add both, our internally created attached function `eitherOrInit` and the user-given attached functions of parameter `attachedFns` of the parental scope to it. As last step, we return the buildable and are done.

As promised, let's have a closer look on the actual implementation now: The body of `eitherOrImpl` have to be something like that:

```
function eitherOrImpl() {
  // (1)
  const valueToUse = probability() ? valueA : valueB;

  // (2)
  const valueAsBuildable = asBuildable(valueToUse, attachedFns);

  // (3)
  node.value = valueAsBuildable;
  node.children = [];

  // (4)
  markFnCalled(initOneOf, node);
  rebuild(node, 'initializer');
}
```

- `(1)`: At first, it chooses one of the options by chance. Faketastic's built-in `probability` function helps us here, as it returns per default `true` by a chance of 50%.
- `(2)`: Then, the chosen option gets converted into a buildable. This is necessary because we want to add the `attachedFns` argument to the randomly chosen item. Why? Consider the following situation:

  ```
  model({
    cart: eitherOr($Product, null, quantity(2)),
  });
  ```

  This model basically says: "a cart has always to positions, that may be two products or two times `null`. Of course, this model is conceptually not that useful, but if we would not attach the given `quantity` function to the randomly chosen item, the quantity-information would get lost.
- `(3)`: After converting the chosen item into the correct form, we set it as new value on the node that the `eitherOrImpl`-processor is attached to and remove all its children (as the content of the node changed, the children have to be reevaluated, which happens in the next step).
- `(4)`: Then, we mark the `eitherOrImpl`-processor function as "called", which is important when rebuilding the node. As last step, we let faketastic rebuild the node as we changed its content. This ensures, that the content gets built if it was a model.

Now we can use our builder function like that:

```
import { eitherOr } from './either-or.builder';

const $Color = model({
  r: range(0, 255),
  g: range(0, 255),
  b: range(0, 255),
});

const $Person = model({
  name: eitherOr('Peter', 'Sara'),
  favoriteColor: eitherOr($Color, null),
});

const sample = build($Person);
// => { name: 'Peter', favoriteColor: { r: 124, g: 81, b: 213 } }
// or { name: 'Sara', favoriteColor: null }
```

Custom ValueFns

Value functions converts user input into a randomized value, such as range or time. Of course, you can write your custom value functions by using the `createValueFn` function:

```
// bool.value-fn.ts

import { createBuildable, createValueFn, probability, AttachedFn,
Buildable } from 'faketastic';

export function bool(likelihoodToBeTrue = 0.5, ...attachedFns:
AttachedFn[]): Buildable<ValueFn> {
  // implementation for creating a random bool, based on a probability.
  // this is what gets called under the hood, later on.
  function boolImpl() {
    return probability(likelihoodToBeTrue);
  }

  const boolValueFn = createValueFn(boolImpl);

  return createBuildable(boolValueFn, attachedFns);
}

// use it:
import { model, canBe } from 'faketastic';
import { bool } from './bool.value-fn';

const $Model = model({
  isCool: bool(0.7, canBe(null)),
});
```

Custom AttachedFns

Attached functions have to have the following signature:

```
(node: ObjectTreeNode) => void
```

The passed in node represents the property they are attached to. In fact, while building, there exists an build-tree, and attached functions are then attached to those build-tree nodes.

While processor functions only manipulate the values on the node they are directly attached to, architect functions most likely do more, like manipulating the build-tree structure, by adding / removing children to their node.

Each attached function type has a factory method:

TreeReaders

```
createTreeReaderFn(fn: Function, runAt: BuildCycle)
```

```
import { ObjectTreeNode } from 'treelike';
import { createTreeReaderFn } from 'faketastic';

// count-tree-level.tree-reader.ts
export function countTreeLevel() {
  let levelCount = 0;

  return createTreeReaderFn((node: ObjectTreeNode) => {
    let currentNode = node.parent;

    while (currentNode) {
      levelCount++;
      currentNode = currentNode.parent;
    }
  });
}
```

Note: By this example we can perfectly see that tree-readers are only useful when not standing alone, but used by builder functions to gather information that they can use for building things then. The above-shown tree-reader is quite pointless, because nobody but the function itself can use the `levelCount` information. That's why you most likely never see a tree-reader function directly on a builder function's arguments list (`range(1, 2, someTreeReader())`), but only internally used by builder functions (like e.g. `range`) as explained earlier.

Architects

```
createArchitectFn(fn: Function, runAt: BuildCycle)
```

```
import { remove, ObjectTreeNode } from 'treelike';
import { createArchitectFn } from 'faketastic';

// remove-empty-entries-from-array.architect.ts
```

```
export function removeEmptyEntriesFromArray() {
  return createArchitectFn((node: ObjectTreeNode) => {
    if (node.type !== 'array') {
      return;
    }

    for (const child of node.children) {
      if (child.value === null || child.value === undefined) {
        remove(child, node);
      }
    }
  }, 'postprocessor');
}

// then use it:
import { removeEmptyEntriesFromArray } from './remove-empty-entries-
from-array.architect';

const $MyModel = model({
  items: someOf([null, undefined, 'str', 42],
removeEmptyEntriesFromArray()),
});
```

This architect runs only on arrays or returns immediately otherwise. It traverses the children of the target array and finds those having the value `null` or `undefined` and removes them from the children-collection.

Processors

`createProcessorFn(fn: Function, runAt: BuildCycle, stickiness: Stickiness)`

```
// to-number.processor.ts
import { ObjectTreeNode } from 'treelike';
import { createProcessorFn } from 'faketastic';

export function toNumber() {
  return createProcessorFn((node: ObjectTreeNode) => {
    if (typeof node.value === 'number') {
      return;
    }

    node.value = parseFloat(node.value);
  }, 'postprocessor');
}

// then use it:
import { toNumber } from './to-number.processor';

const $MyModel = model({
  items: oneOf(['99', 12, '4711', 42], toNumber()),
});
```

> This processor converts the value of the node it is attached to into a number, or – if it is already a number – returns immediately.