Manner Core Documentation

Release 0.0.1

Jonathan Jacobs

April 27, 2015

Contents

Manner Core is a JavaScript library for managing the validity and visibility aspects of a schema. It is heavily geared towards complex web forms but uses standard data structures and is not reliant on the DOM at all.

As such Manner Core does not actually validate web forms or provide any web components to do so, this is left to the various framework integrations such as react-manner.

CHAPTER 1

Models

XXX: Subject to change.

Models are simply an `Immutable.Map` of field names to field values.

CHAPTER 2

Asynchronicity

The documentation about the results of most aspects of Manner will talk about them as though they're synchronous in an attempt to simplify the explanation of Manner's operations. Under the hood, however, Manner will assume that any predicate may potentially return an asynchronous result. The nature of asynchronous results means that this assumption has to propagate throughout the library, the results of both *Validators* and *Conditions* are `Promises` that only resolve when all their predicates have resolved.

XXX: Talk about pending result state that doesn't exist yet.

CHAPTER 3

Predicates

3.1 Overview

At the heart of Manner is the concept of a *predicate*: A function taking some inputs and returning a result indicating whether it considers those inputs as valid or invalid. For example, `equalTo(42)` constructs a predicate that only considers input equal to 42 valid.

A predicate does not return `true` or `false` but a *status* value that more richly describes the result by providing the reason for some input not passing the predicate. However, predicates are usually constructed from *boolean predicates* which do return `true` or `false`. It is strongly encouraged to make your own and combine existing predicates to better suit your domain.

Finally, it is important to note that Manner assumes that predicates—and, by extension, boolean predicates—are pure: Given the same inputs they always have the same output.

3.2 Boolean predicates

A boolean predicate is a simple factory that takes some arguments, used to set up any conditions, and returns a new function that may accept more parameters and finally returns `true` or `false`. As an example, here is the implementation of the `between` boolean predicate:

```
/** Between `a` and `b`, inclusively. */
function between(a, b) {
  return function (v) { return v >= a && v <= b; };
}
```

Often predicates are implemented in terms of boolean predicates, while not strictly required this tends to result in a collection of small composable functions which means easier testing and greater implementation flexibility.

Creating a predicate from a boolean predicate is so common that there is a public helper, `predicate`, in Manner to do this. Here is the implementation of the `between` predicate:

```
import * as P from "manner/predicates";
import * as PB from "manner/predicates/boolean";
/** Between `a` and `b`, inclusively. */
let between = P.predicate(PB.between, a_message);
```

3.3 Bound predicates

A predicate only takes input as parameters and returns an output, it has no knowledge of fields and their values. As one might imagine, invoking a predicate with values from your model is a very common operation and for this reason there exist *bound predicates*. Bound predicates enable you to describe the relationship between field names in a model and a predicate, in effect binding them.

For example, `is` binds a single field to a predicate: `is('one', equalTo(42))` produces a bound predicate that, when invoked with an `Immutable.Map` of field names to field values, will extract the value for the field `one`, pass it to the `equalTo(42)` predicate and return the result.

3.4 Combining predicates

Predicates are generally small, simple functions with a single purpose, meaning they can easily be combined to form more complex predicates. Some built-in predicates are combinations of others, such as `numeric`.

Requiring all combined predicates to pass can be done with `and` (as in logical AND), while requiring at least one combined predicate to pass can be done with `or` (as in logical OR); both return a single new predicate. For example:

```
import * as P from "manner/predicates";
let positiveAndNot42 = P.and(P.greaterThan(0),
                             P.notEqualTo(42));
```

3.5 Predicate status

While a predicate essentially returns only one of two values—valid or invalid—the result needs to be richer than a simple boolean value. If nothing else, there needs to be a reason indicating why the input failed to validate; which is where `Status` comes in.

A status is intended to be constructed only via its static methods and in the case of predicates there are only two such methods: `valid()` and `invalid(reason)`.

In the event that there is more than one status for a field—imagine that a field is involved in multiple predicates—the statuses are combined to form a new `Status` with invalid statuses trumping valid statuses.

3.6 Custom messages

In the event that a custom message for a predicate is necessary, it's possible to use `message` to wrap an existing predicate with a customized message:

Note: `message` always returns an asynchronous result, see *Asynchronicity*.

```
import * as P from "manner/predicates";
let myEqualTo = P.message("Nope", P.equalTo);
myEqualTo(42)(21).call('message');  // => "Nope"
```

Or provide a message function to access input arguments or perform *Internationalization*:

```
import * as P from "manner/predicates";
function myEqualToMsg(_, args, rest) {
  return args[0] + ' !== ' + rest[0];
```

```
}
let myEqualTo2 = P.message(myEqualToMsg, P.equalTo);
myEqualTo(42)(21).call('message')  // => "42 !== 21"
```

3.7 Long-running predicates

There may be some cases where it is undesirable to run a predicate too frequently, for example predicates that make an HTTP request. Usually these situations are resolved via a technique commonly referred to as "debouncing", only calling the function at most within some user-specified time frame, which may be achieved with the `debounce` function.

If the predicate is run again before the debounce interval elapses, the pending predicate is cancelled and a new one, with a fresh interval, started in its place.

CHAPTER 4

Validators

4.1 Overview

If a bound predicate is a way to check a single predicate against some fields, then a *validator* is a way to check many bound predicates against a model.

Note: Internally validators keep track of pending predicate results and a cache of results for the previous inputs, which means they need an extra layer of indirection if they are to be used with multiple models.

4.2 Creating a validator

Since validators are simply a list of bound predicates, creating one is a short two-step process:

1. Create a validators definition that can be reused:

```
import * as P from "manner/predicates";
import * as V from "manner/validators";
let someValidators = V.validators(
  P.is('one', P.equalTo(42)),
  P.is('two', P.notNull()));
```

2. Instantiate a validators definition to create an instance, a *validator*, with its own state suitable for repeated use with one particular model:

```
let formValidator = V.instantiate(someValidators);
formValidator(my_model);  // => Validation results
```

The ultimate result of invoking an validator is an `Immutable.Map` of field names to *predicate status*, which can then be used to update the state of a form, perhaps indicating which fields failed to validate.

4.3 Long-running predicates

A validator takes an optional second argument: a callback function, passed the result of a bound predicate, that is called as soon as the result is resolved.

Asynchronous predicates may prevent a validator from resolving for an extended period of time thus delaying any important user interface updates, in this case the callback function can be used to update the user interface as predicate results are resolved.

CHAPTER 5

Conditions

5.1 Overview

In complex forms it is often the case that some inputs need to be hidden or disabled under certain conditions, such as an earlier field having a particular value, this is what *conditions* in Manner provide.

Conditions in Manner, which are separate from *Validators*, build on bound predicates and additionally specify resulting *actions*. An action might be something like "hide fields X, Y and Z" or "enable fields A, B and C" or possibly even both.

Note: Internally conditions keep track of pending predicate results and a cache of results for the previous inputs, which means they need an extra layer of indirection if they are to be used with multiple models.

5.2 Condition actions

Condition actions are the way a condition acts on the result of a bound predicate, multiple condition actions may occur for a single condition. Available actions are: `hide`, `show`, `disable` and `enable`.

Note: Conditions will always emit a resulting action, the result of the bound predicate will dictate what the result of an action will be: The `hide` action will suggest hiding the bound fields on success and showing them on failure, and vice versa for `show`; likewise the `disable` action will suggest disabling the bound fields on success and enabling them on failure, and vice versa for `enable`.

The result of conflicting actions for a single field is not well defined.

5.3 Creating a condition

Conditions are only marginally more complex than validators to construct, in addition to containing a bound predicate they must also specify actions; the `when` function assists in this regard by creating a *condition checker*. Still, this is only a short two-step process:

1. Create a conditions definition that can be reused, read as: When `one` is equal to 42 then hide `x`, `y` and `z`, and enable `a`, `b` and `c`:

```
import * as P from "manner/predicates";
import * as C from "manner/conditions";
let someConditions = C.conditions(
```

```
C.when(P.is('one', P.equalTo(42)),
       // Hide "x", "y" and "z" fields on valid input.
       C.hide('x', 'y', 'z'),
       // Enable "a", "b" and "c" fields on valid input.
       C.enable('a', 'b', 'c')));
```

2. Instantiate a conditions definition to create an instance, a *condition*, with its own state suitable for repeated use with one particular model:

```
let formConditions = C.instantiate(someConditions);
formConditions(my_model);  // => Condition results
```

The ultimate result of invoking a condition is an `Immutable.Map` of field names to *condition status*, which can then be used to update the state of a form.

5.4 Long-running predicates

A condition takes an optional second argument: a callback function, passed the result of a condition checker, that is called as soon as the result is resolved.

Asynchronous predicates may prevent a validator from resolving for an extended period of time thus delaying any important user interface updates, in this case the callback function can be used to update the user interface as predicate results are resolved.

5.5 Condition status

The results of a condition are more complex than those of a predicate because there are three possible states: `hidden`, `disabled` and `normal`. Conditions have a separate `Status` to predicates for two main reasons:

1. There is an additional state;
2. Every state constructor accepts an optional message, which may be useful when describing why something is available or unavailable in a user interface

CHAPTER 6

Internationalization

Internationalization is an essential part of being able to effectively communicate schema requirements and reasons to users; it's also a problem that is best considered at an early stage in software.

In every case where Manner specifies that a function requires a message input, the value can either be a plain string—and thus no internationalization takes place—or a message function that accepts a mapping of keywords to a function taking some number of arguments and returning an internationalized message.

In every, but the final, case where Manner specifies that it returns a message it will return a function that needs to be called with an internationalization map for the desired output language and any arguments that may need to be formatted into the result.

We can better illustrate the concept with an example:

```
import {Status} from "manner/predicates";
import {i18nMessage} from "manner/i18n";
// A message with no internationalization.
let plain = Status.invalid("Hello Bob");
plain.message()  // => "Hello Bob"
// Define some internationalization maps for our internationalized message.
let i18n_en = {'greetings': {'hello': args => "Hello " + args.value}};
let i18n_se = {'greetings': {'hello': args => "Hej " + args.value}};
let enhanced = Status.invalid(i18nMessage('greetings', 'hello'));
enhanced.message(i18n_en, ['Bob']);  // => "Hello Bob"
enhanced.message(i18n_se, ['Bob']);  // => "Hej Bob"
```

6.1 Predicates

Predicate messages in Manner are all defined in a way that all the input arguments are closed over by the returned function and only the internationalization map is needed to yield a message, thanks to the `predicate` function. For example:

```
import * as P from "manner/predicates";
import en from "manner/i18n/en";
P.notEqual(42)(21).message(en)  // => 'Must be "42"'
P.empty()(21).message(en)  // => 'Must be empty not "21"'
```

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*