The `check` package includes pattern checking functions useful for checking the types and structure of variables and an extensible library of patterns to specify which types you are expecting.

To add `check` (or `Match`) to your application, run this command in your terminal:

```
meteor add check
```

{% apibox "check" %}

Meteor methods and publish functions can take arbitrary EJSON types as arguments, but most functions expect their arguments to be of a particular type. `check` is a lightweight function for checking that arguments and other values are of the expected type. For example:

```
Meteor.publish('chatsInRoom', function (roomId) {
  // Make sure `roomId` is a string, not an arbitrary Mongo selector object.
  check(roomId, String);
  return Chats.find({ room: roomId });
});

Meteor.methods({
  addChat(roomId, message) {
    check(roomId, String);
    check(message, {
      text: String,
      timestamp: Date,
      // Optional, but if present must be an array of strings.
      tags: Match.Maybe([String])
    });

    // Do something with the message...
  }
});
```

If the match fails, `check` throws a `Match.Error` describing how it failed. If this error gets sent over the wire to the client, it will appear only as `Meteor.Error(400, 'Match Failed')`. The failure details will be written to the server logs but not revealed to the client.

{% apibox "Match.test" %}

`Match.test` can be used to identify if a variable has a certain structure.

```
// Will return true for `{ foo: 1, bar: 'hello' }` or similar.
Match.test(value, { foo: Match.Integer, bar: String });

// Will return true if `value` is a string.
Match.test(value, String);

// Will return true if `value` is a string or an array of numbers.
Match.test(value, Match.OneOf(String, [Number]));
```

This can be useful if you have a function that accepts several different kinds of objects, and you want to determine which was passed in.

Match Patterns

The following patterns can be used as pattern arguments to `check` and `Match.test`:

{% dtdd name:"`Match.Any`" %} Matches any value. {% enddtdd %}

{% dtdd name:" `String`, `Number`, `Boolean`, `undefined`, `null` " %} Matches a primitive of the given type. {% enddtdd %}

{% dtdd name:" `Match.Integer` " %} Matches a signed 32-bit integer. Doesn't match `Infinity`, `-Infinity`, or `NaN`. {% enddtdd %}

{% dtdd name:" `[pattern]` " %} A one-element array matches an array of elements, each of which match *pattern*. For example, `[Number]` matches a (possibly empty) array of numbers; `[Match.Any]` matches any array. {% enddtdd %}

`{ key1: pattern1, key2: pattern2, ... }`

Matches an Object with the given keys, with values matching the given patterns. If any *pattern* is a `Match.Maybe` or `Match.Optional`, that key does not need to exist in the object. The value may not contain any keys not listed in the pattern. The value must be a plain Object with no special prototype.

`Match.ObjectIncluding({ key1: pattern1, key2: pattern2, ... })`

Matches an Object with the given keys; the value may also have other keys with arbitrary values.

{% dtdd name:" `Object` " %} Matches any plain Object with any keys; equivalent to `Match.ObjectIncluding({})`. {% enddtdd %}

{% dtdd name:" `Match.Maybe(pattern)` " %}

Matches either `undefined`, `null`, or *pattern*. If used in an object, matches only if the key is not set as opposed to the value being set to `undefined` or `null`. This set of conditions was chosen because `undefined` arguments to Meteor Methods are converted to `null` when sent over the wire.

{% codeblock lang:js %} // In an object const pattern = { name: Match.Maybe(String) };

check({ name: 'something' }, pattern); // OK check({}, pattern); // OK check({ name: undefined }, pattern); // Throws an exception check({ name: null }, pattern); // Throws an exception

// Outside an object check(null, Match.Maybe(String)); // OK check(undefined, Match.Maybe(String)); // OK {% endcodeblock %} {% enddtdd %}

{% dtdd name:" `Match.Optional(pattern)` " %}

Behaves like `Match.Maybe` except it doesn't accept `null`. If used in an object, the behavior is identical to `Match.Maybe`.

{% enddtdd %}

{% dtdd name:" `Match.OneOf(pattern1, pattern2, ...)` " %} Matches any value that matches at least one of the provided patterns. {% enddtdd %}

{% dtdd name:"Any constructor function (eg, `Date`)" %} Matches any element that is an instance of that type. {% enddtdd %}

{% dtdd name:"`Match.Where(condition)`" %} Calls the function *condition* with the value as the argument. If *condition* returns true, this matches. If *condition* throws a `Match.Error` or returns false, this fails. If *condition* throws any other error, that error is thrown from the call to `check` or `Match.test`. Examples:

{% codeblock lang:js %} check(buffer, Match.Where(EJSON.isBinary));

const NonEmptyString = Match.Where((x) => { check(x, String); return x.length > 0; });

check(arg, NonEmptyString); {% endcodeblock %} {% enddtdd %}