Methods

Methods are remote functions that Meteor clients can invoke with `Meteor.call`.

If you prefer to watch the video, click below.

{% youtube 2uoeBq8SF9E %}

{% apibox "Meteor.methods" %}

Example:

```
Meteor.methods({
  foo(arg1, arg2) {
    check(arg1, String);
    check(arg2, [Number]);

    // Do stuff...

    if (/* you want to throw an error */) {
      throw new Meteor.Error('pants-not-found', "Can't find my pants");
    }

    return 'some return value';
  },

  bar() {
    // Do other stuff...
    return 'baz';
  }
});
```

Calling `methods` on the server defines functions that can be called remotely by clients. They should return an EJSON-able value or throw an exception. Inside your method invocation, `this` is bound to a method invocation object, which provides the following:

- `isSimulation`: a boolean value, true if this invocation is a stub.
- `unblock`: when called, allows the next method from this client to begin running.
- `userId`: the id of the current user.

- `setUserId`: a function that associates the current client with a user.
- `connection`: on the server, the connection this method call was received on.

Calling `methods` on the client defines *stub* functions associated with server methods of the same name. You don't have to define a stub for your method if you don't want to. In that case, method calls are just like remote procedure calls in other systems, and you'll have to wait for the results from the server.

If you do define a stub, when a client invokes a server method it will also run its stub in parallel. On the client, the return value of a stub is ignored. Stubs are run for their side-effects: they are intended to *simulate* the result of what the server's method will do, but without waiting for the round trip delay. If a stub throws an exception it will be logged to the console.

You use methods all the time, because the database mutators (`insert`, `update`, `remove`) are implemented as methods. When you call any of these functions on the client, you're invoking their stub version that update the local cache, and sending the same write request to the server. When the server responds, the client updates the local cache with the writes that actually occurred on the server.

You don't have to put all your method definitions into a single `Meteor.methods` call; you may call it multiple times, as long as each method has a unique name.

If a client calls a method and is disconnected before it receives a response, it will re-call the method when it reconnects. This means that a client may call a method multiple times when it only means to call it once. If this behavior is problematic for your method, consider attaching a unique ID to each method call on the client, and checking on the server whether a call with this ID has already been made. Alternatively, you can use `Meteor.apply` with the noRetry option set to true.

Read more about methods and how to use them in the Methods article in the Meteor Guide.

{% apibox "DDPCommon.MethodInvocation#userId" %}

The user id is an arbitrary string — typically the id of the user record in the database. You can set it with the `setUserId` function. If you're using the Meteor accounts system then this is handled for you.

{% apibox "DDPCommon.MethodInvocation#setUserId" %}

Call this function to change the currently logged-in user on the connection that made this method call. This simply sets the value of `userId` for future method calls received on this connection. Pass `null` to log out the connection.

If you are using the built-in Meteor accounts system then this should correspond to the `_id` field of a document in the `Meteor.users` collection.

`setUserId` is not retroactive. It affects the current method call and any future method calls on the connection. Any previous method calls on this connection will still see the value of `userId` that was in effect when they started.

If you also want to change the logged-in user on the client, then after calling `setUserId` on the server, call `Meteor.connection.setUserId(userId)` on the client.

{% apibox "DDPCommon.MethodInvocation#isSimulation" %}

{% apibox "DDPCommon.MethodInvocation#unblock" %}

On the server, methods from a given client run one at a time. The N+1th invocation from a client won't start until the Nth invocation returns. However, you can change this by calling `this.unblock`. This will allow the N+1th invocation to start running in a new fiber.

{% apibox "DDPCommon.MethodInvocation#connection" %}

{% apibox "Meteor.Error" %}

If you want to return an error from a method, throw an exception. Methods can throw any kind of exception. But `Meteor.Error` is the only kind of error that a server will send to the client. If a method function throws a different exception, then it will be mapped to a sanitized version on the wire. Specifically, if the `sanitizedError` field on the thrown error is set to a `Meteor.Error`, then that error will be sent to the client. Otherwise, if no sanitized version is available, the client gets `Meteor.Error(500, 'Internal server error')`.

{% apibox "Meteor.call" %}

This is how to invoke a method. It will run the method on the server. If a stub is available, it will also run the stub on the client. (See also `Meteor.apply`, which is identical to `Meteor.call` except that you specify the parameters as an array instead of as separate arguments and you can specify a few options controlling how the method is executed.)

If you include a callback function as the last argument (which can't be an argument to the method, since functions aren't serializable), the method will run asynchronously: it will return nothing in particular and will not throw an exception. When the method is complete (which may or may not happen before `Meteor.call` returns), the callback will be called with two arguments: `error` and `result`. If an error was thrown, then `error` will be the exception object. Otherwise, `error` will be `undefined` and the return value (possibly `undefined`) will be in `result`.

```
// Asynchronous call
Meteor.call('foo', 1, 2, (error, result) => { ... });
```

If you do not pass a callback on the server, the method invocation will block until the method is complete. It will eventually return the return value of the method, or it will throw an exception if the method threw an exception. (Possibly mapped to 500 Server Error if the exception happened remotely and it was not a `Meteor.Error` exception.)

```
// Synchronous call
const result = Meteor.call('foo', 1, 2);
```

On the client, if you do not pass a callback and you are not inside a stub, `call` will return `undefined`, and you will have no way to get the return value of the method. That is because the client doesn't have fibers, so there is not actually any way it can block on the remote execution of a method.

Finally, if you are inside a stub on the client and call another method, the other method is not executed (no RPC is generated, nothing "real" happens). If that other method has a stub, that stub stands in for the method and is executed. The method call's return value is the return value of the stub function. The client has no problem executing a stub synchronously, and that is why it's okay for the client to use the synchronous `Meteor.call` form from inside a method body, as described earlier.

Meteor tracks the database writes performed by methods, both on the client and the server, and does not invoke `asyncCallback` until all of the server's writes replace the stub's writes in the local cache. In some cases, there can be a lag between the method's return value being available and the writes being visible: for example, if another method still outstanding wrote to the same document, the local cache may not be up to date until the other method finishes as well. If you want to process the method's result as soon as it arrives from the server, even if the method's writes are not available yet, you can specify an `onResultReceived` callback to `Meteor.apply`.

{% apibox "Meteor.apply" %}

`Meteor.apply` is just like `Meteor.call`, except that the method arguments are passed as an array rather than directly as arguments, and you can specify options about how the client executes the method.

DDPRateLimiter

Customize rate limiting for methods and subscriptions to avoid a high load of WebSocket messages in your app.

> Galaxy (Meteor hosting) offers additional App Protection, read more and try it with our free 30-day trial.

By default, `DDPRateLimiter` is configured with a single rule. This rule limits login attempts, new user creation, and password resets to 5 attempts every 10 seconds per connection. It can be removed by calling `Accounts.removeDefaultRateLimit()`.

To use `DDPRateLimiter` for modifying the default rate-limiting rules, add the `ddp-rate-limiter` package to your project in your terminal:

```
meteor add ddp-rate-limiter
```

{% apibox “DDPRateLimiter.addRule” nested:true instanceDelimiter:. %}

Custom rules can be added by calling `DDPRateLimiter.addRule`. The rate limiter is called on every method and subscription invocation.

A rate limit is reached when a bucket has surpassed the rule’s predefined capacity, at which point errors will be returned for that input until the buckets are reset. Buckets are regularly reset after the end of a time interval.

Here’s example of defining a rule and adding it into the `DDPRateLimiter`:

```
// Define a rule that matches login attempts by non-admin users.
const loginRule = {
  userId(userId) {
    const user = Meteor.users.findOne(userId);
    return user && user.type !== 'admin';
  },

  type: 'method',
  name: 'login'
};

// Add the rule, allowing up to 5 messages every 1000 milliseconds.
DDPRateLimiter.addRule(loginRule, 5, 1000);
```

{% apibox “DDPRateLimiter.removeRule” nested:true instanceDelimiter:. %}
{% apibox “DDPRateLimiter.setErrorMessage” nested:true instanceDelimiter:. %}