Methods

After reading this article, you'll know:

1. What Methods are in Meteor and how they work in detail.
2. Best practices for defining and calling Methods.
3. How to throw and handle errors with Methods.
4. How to call a Method from a form.

What is a Method?

Methods are Meteor's remote procedure call (RPC) system, used to save user input events and data that come from the client. If you're familiar with REST APIs or HTTP, you can think of them like POST requests to your server, but with many nice features optimized for building a modern web application. Later on in this article, we'll go into detail about some of the benefits you get from Methods that you wouldn't get from an HTTP endpoint.

At its core, a Method is an API endpoint for your server; you can define a Method on the server and its counterpart on the client, then call it with some data, write to the database, and get the return value in a callback. Meteor Methods are also tightly integrated with the pub/sub and data loading systems of Meteor to allow for Optimistic UI—the ability to simulate server-side actions on the client to make your app feel faster than it actually is.

We'll be referring to Meteor Methods with a capital M to differentiate them from class methods in JavaScript.

Defining and calling Methods

Basic Method

In a basic app, defining a Meteor Method is as simple as defining a function. In a complex app, you want a few extra features to make Methods more powerful and testable. First, we're going to go over how to define a Method using the Meteor core API, and in a later section we'll go over how to use a helpful wrapper package we've created to enable a more powerful Method workflow.

Defining

Here's how you can use the built-in `Meteor.methods` API to define a Method. Note that Methods should always be defined in common code loaded on the

client and the server to enable Optimistic UI. If you have some secret code in your Method, consult the Security article for how to hide it from the client.

This example uses the simpl-schema npm package, which is recommended in several other articles, to validate the Method arguments.

```
import SimpleSchema from 'simpl-schema';

Meteor.methods({
  'todos.updateText'({ todoId, newText }) {
    new SimpleSchema({
      todoId: { type: String },
      newText: { type: String }
    }).validate({ todoId, newText });

    const todo = Todos.findOne(todoId);

    if (!todo.editableBy(this.userId)) {
      throw new Meteor.Error('todos.updateText.unauthorized',
        'Cannot edit todos in a private list that is not yours');
    }

    Todos.update(todoId, {
      $set: { text: newText }
    });
  }
});
```

Calling

This Method is callable from the client and server using `Meteor.call`. Note that you should only use a Method in the case where some code needs to be callable from the client; if you just want to modularize code that is only going to be called from the server, use a regular JavaScript function, not a Method.

Here's how you can call this Method from the client:

```
Meteor.call('todos.updateText', {
  todoId: '12345',
  newText: 'This is a todo item.'
}, (err, res) => {
  if (err) {
    alert(err);
  } else {
    // success!
  }
});
```

If the Method throws an error, you get that in the first argument of the callback.

If the Method succeeds, you get the result in the second argument and the first argument `err` will be `undefined`. For more information about errors, see the section below about error handling.

Advanced Method boilerplate

Meteor Methods have several features which aren't immediately obvious, but every complex app will need them at some point. These features were added incrementally over several years in a backwards-compatible fashion, so unlocking the full capabilities of Methods requires a good amount of boilerplate. In this article we will first show you all of the code you need to write for each feature, then the next section will talk about a Method wrapper package we have developed to make it easier.

Here's some of the functionality an ideal Method would have:

1. Run validation code by itself without running the Method body.
2. Override the Method for testing.
3. Call the Method with a custom user ID, especially in tests (as recommended by the Discover Meteor two-tiered methods pattern).
4. Refer to the Method via JS module rather than a magic string.
5. Get the Method simulation return value to get IDs of inserted documents.
6. Avoid calling the server-side Method if the client-side validation failed, so we don't waste server resources.

Defining

```
export const updateText = {
  name: 'todos.updateText',

  // Factor out validation so that it can be run independently (1)
  validate(args) {
    new SimpleSchema({
      todoId: { type: String },
      newText: { type: String }
    }).validate(args)
  },

  // Factor out Method body so that it can be called independently (3)
  run({ todoId, newText }) {
    const todo = Todos.findOne(todoId);

    if (!todo.editableBy(this.userId)) {
      throw new Meteor.Error('todos.updateText.unauthorized',
        'Cannot edit todos in a private list that is not yours');
    }

    Todos.update(todoId, {
      $set: { text: newText }
```

```
    });
  },

  // Call Method by referencing the JS object (4)
  // Also, this lets us specify Meteor.apply options once in
  // the Method implementation, rather than requiring the caller
  // to specify it at the call site.
  call(args, callback) {
    const options = {
      returnStubValue: true,     // (5)
      throwStubExceptions: true  // (6)
    }

    Meteor.apply(this.name, [args], options, callback);
  }
};

// Actually register the method with Meteor's DDP system
Meteor.methods({
  [updateText.name]: function (args) {
    updateText.validate.call(this, args);
    updateText.run.call(this, args);
  }
})
```

Calling

Now calling the Method is as simple as calling a JavaScript function:

```
import { updateText } from './path/to/methods.js';

// Call the Method
updateText.call({
  todoId: '12345',
  newText: 'This is a todo item.'
}, (err, res) => {
  if (err) {
    alert(err);
  } else {
    // success!
  }
});

// Call the validation only
updateText.validate({ wrong: 'args'});

// Call the Method with custom userId in a test
```

```
updateText.run.call({ userId: 'abcd' }, {
  todoId: '12345',
  newText: 'This is a todo item.'
});
```

As you can see, this approach to calling Methods results in a better development workflow - you can more easily deal with the different parts of the Method separately and test your code without having to deal with Meteor internals. But this approach requires you to write a lot of boilerplate on the Method definition side.

Advanced Methods with mdg:validated-method

To alleviate some of the boilerplate that's involved in correct Method definitions, we've published a wrapper package called `mdg:validated-method` that does most of this for you. Here's the same Method as above, but defined with the package:

```
import { ValidatedMethod } from 'meteor/mdg:validated-method';

export const updateText = new ValidatedMethod({
  name: 'todos.updateText',
  validate: new SimpleSchema({
    todoId: { type: String },
    newText: { type: String }
  }).validator(),
  run({ todoId, newText }) {
    const todo = Todos.findOne(todoId);

    if (!todo.editableBy(this.userId)) {
      throw new Meteor.Error('todos.updateText.unauthorized',
        'Cannot edit todos in a private list that is not yours');
    }

    Todos.update(todoId, {
      $set: { text: newText }
    });
  }
});
```

You call it the same way you call the advanced Method above, but the Method definition is significantly simpler. We believe this style of Method lets you clearly see the important parts - the name of the Method sent over the wire, the format of the expected arguments, and the JavaScript namespace by which the Method can be referenced. Validated methods only accept a single argument and a callback function.

Error handling

In regular JavaScript functions, you indicate errors by throwing an `Error` object. Throwing errors from Meteor Methods works almost the same way, but a bit of complexity is introduced by the fact that in some cases the error object will be sent over a websocket back to the client.

Throwing errors from a Method

Meteor introduces two new types of JavaScript errors: `Meteor.Error` and `ValidationError`. These and the regular JavaScript `Error` type should be used in different situations:

Regular `Error` for internal server errors

When you have an error that doesn't need to be reported to the client, but is internal to the server, throw a regular JavaScript error object. This will be reported to the client as a totally opaque internal server error with no details.

Meteor.Error for general runtime errors

When the server was not able to complete the user's desired action because of a known condition, you should throw a descriptive `Meteor.Error` object to the client. In the Todos example app, we use these to report situations where the current user is not authorized to complete a certain action, or where the action is not allowed within the app - for example, deleting the last public list.

`Meteor.Error` takes three arguments: `error`, `reason`, and `details`.

1. `error` should be a short, unique, machine-readable error code string that the client can interpret to understand what happened. It's good to prefix this with the name of the Method for easy internationalization, for example: `'todos.updateText.unauthorized'`.
2. `reason` should be a short description of the error for the developer. It should give your coworker enough information to be able to debug the error. The `reason` parameter should not be printed to the end user directly, since this means you now have to do internationalization on the server before sending the error message, and the UI developer has to worry about the Method implementation when thinking about what will be displayed in the UI.
3. `details` is optional, and can be used where extra data will help the client understand what is wrong. In particular, it can be combined with the `error` field to print a more helpful error message to the end user.

ValidationError for argument validation errors

When a Method call fails because the arguments are of the wrong type, it's good to throw a `ValidationError`. This works like `Meteor.Error`, but is a custom constructor that enforces a standard error format that can be read by different form and validation libraries. In particular, if you are calling this Method from a form, throwing a `ValidationError` will make it possible to display nice error messages next to particular fields in the form.

When you use `mdg:validated-method` with `simpl-schema` as demonstrated above, this type of error is thrown for you.

Read more about the error format in the `mdg:validation-error` docs.

Handling errors

When you call a Method, any errors thrown by it will be returned in the callback. At this point, you should identify which error type it is and display the appropriate message to the user. In this case, it is unlikely that the Method will throw a `ValidationError` or an internal server error, so we will only handle the unauthorized error:

```
// Call the Method
updateText.call({
  todoId: '12345',
  newText: 'This is a todo item.'
}, (err, res) => {
  if (err) {
    if (err.error === 'todos.updateText.unauthorized') {
      // Displaying an alert is probably not what you would do in
      // a real app; you should have some nice UI to display this
      // error, and probably use an i18n library to generate the
      // message from the error code.
      alert('You aren\'t allowed to edit this todo item');
    } else {
      // Unexpected error, handle it in the UI somehow
    }
  } else {
    // success!
  }
});
```

We'll talk about how to handle the `ValidationError` in the section on forms below.

Errors in Method simulation

When a Method is called, it usually runs twice—once on the client to simulate the result for Optimistic UI, and again on the server to make the actual change to the database. This means that if your Method throws an error, it will likely fail on the client *and* the server. For this reason, `ValidatedMethod` turns on undocumented option in Meteor to avoid calling the server-side implementation if the simulation throws an error.

While this behavior is good for saving server resources in cases where a Method will certainly fail, it's important to make sure that the simulation doesn't throw an error in cases where the server Method would have succeeded (for example, if you didn't load some data on the client that the Method needs to do the

simulation properly). In this case, you can wrap server-side-only logic in a block that checks for a method simulation:

```
if (!this.isSimulation) {
  // Logic that depends on server environment here
}
```

Calling a Method from a form

The main thing enabled by the `ValidationError` convention is integration between Methods and the forms that call them. In general, your app is likely to have a one-to-one mapping of forms in the UI to Methods. First, let's define a Method for our business logic:

```
// This Method encodes the form validation requirements.
// By defining them in the Method, we do client and server-side
// validation in one place.
export const insert = new ValidatedMethod({
  name: 'Invoices.methods.insert',
  validate: new SimpleSchema({
    email: { type: String, regEx: SimpleSchema.RegEx.Email },
    description: { type: String, min: 5 },
    amount: { type: String, regEx: /^\d*\.(\d\d)?$/ }
  }).validator(),
  run(newInvoice) {
    // In here, we can be sure that the newInvoice argument is
    // validated.

    if (!this.userId) {
      throw new Meteor.Error('Invoices.methods.insert.not-logged-in',
        'Must be logged in to create an invoice.');
    }

    Invoices.insert(newInvoice)
  }
});
```

Let's define an HTML form:

```
<template name="Invoices_newInvoice">
  <form class="Invoices_newInvoice">
    <label for="email">Recipient email</label>
    <input type="email" name="email" />
    {{#each error in errors "email"}}
      <div class="form-error">{{error}}</div>
    {{/each}}

    <label for="description">Item description</label>
    <input type="text" name="description" />
```

```
    {{#each error in errors "description"}}
      <div class="form-error">{{error}}</div>
    {{/each}}

    <label for="amount">Amount owed</label>
    <input type="text" name="amount" />
    {{#each error in errors "amount"}}
      <div class="form-error">{{error}}</div>
    {{/each}}
  </form>
</template>
```

Now, let's write some JavaScript to handle this form nicely:

```
import { insert } from '../api/invoices/methods.js';

Template.Invoices_newInvoice.onCreated(function() {
  this.errors = new ReactiveDict();
});

Template.Invoices_newInvoice.helpers({
  errors(fieldName) {
    return Template.instance().errors.get(fieldName);
  }
});

Template.Invoices_newInvoice.events({
  'submit .Invoices_newInvoice'(event, instance) {
    const data = {
      email: event.target.email.value,
      description: event.target.description.value,
      amount: event.target.amount.value
    };

    insert.call(data, (err, res) => {
      if (err) {
        if (err.error === 'validation-error') {
          // Initialize error object
          const errors = {
            email: [],
            description: [],
            amount: []
          };

          // Go through validation errors returned from Method
          err.details.forEach((fieldError) => {
            // XXX i18n
```

```
            errors[fieldError.name].push(fieldError.type);
          });

          // Update ReactiveDict, errors will show up in the UI
          instance.errors.set(errors);
        }
      }
    });
  }
});
```

As you can see, there is a fair amount of boilerplate to handle errors nicely in a form, but most of it can be abstracted by an off-the-shelf form framework or an application-specific wrapper of your own design.

Loading data with Methods

Since Methods can work as general purpose RPCs, they can also be used to fetch data instead of publications. There are some advantages and some disadvantages to this approach compared with loading data through publications, and at the end of the day we recommend always using publications to load data.

Methods can be useful to fetch the result of a complex computation from the server that doesn't need to update when the server data changes. The biggest disadvantage of fetching data through Methods is that the data won't be automatically loaded into Minimongo, Meteor's client-side data cache, so you'll need to manage the lifecycle of that data manually. Another disadvantage is that database queries are not shared between clients like publication cursors often are—the Method (and any queries it contains) will run once for each client that calls it.

Using a local collection to store and display data fetched from a Method

Collections are a very convenient way of storing data on the client side. If you're fetching data using something other than subscriptions, you can put it in a collection manually. Let's look at an example where we have a complex algorithm for calculating average scores from a series of games for a number of players. We don't want to use a publication to load this data because we want to control exactly when it runs, and don't want the data to be cached automatically.

First, you need to create a *local collection* - this is a collection that exists only on the client side and is not tied to a database collection on the server. Read more in the Collections article.

```
// In client-side code, declare a local collection
// by passing `null` as the argument
ScoreAverages = new Mongo.Collection(null);
```

Now, if you fetch data using a Method, you can put into this collection:

```
import { calculateAverages } from '../api/games/methods.js';

function updateAverages() {
  // Clean out result cache
  ScoreAverages.remove({});

  // Call a Method that does an expensive computation
  calculateAverages.call((err, res) => {
    res.forEach((item) => {
      ScoreAverages.insert(item);
    });
  });
}
```

We can now use the data from the local collection `ScoreAverages` inside a UI component exactly the same way we would use a regular MongoDB collection. Instead of it updating automatically, we'll need to call `updateAverages` every time we need new results.

Advanced concepts

While you can use Methods in an app by following the Meteor introductory tutorial, it's important to understand exactly how they work to use them effectively in a production app. One of the downsides of using a framework like Meteor that does a lot for you under the hood is that you don't always understand what is going on, so it's good to learn some of the core concepts.

Method call lifecycle

Here's exactly what happens, in order, when a Method is called:

1. Method simulation runs on the client

If we defined this Method in client and server code, as all Methods should be, a Method simulation is executed in the client that called it.

The client enters a special mode where it tracks all changes made to client-side collections, so that they can be rolled back later. When this step is complete, the user of your app sees their UI update instantly with the new content of the client-side database, but the server hasn't received any data yet.

If an exception is thrown from the Method simulation, then by default Meteor ignores it and continues to step (2). If you are using `ValidatedMethod` or pass a special `throwStubExceptions` option to `Meteor.apply`, then an exception thrown from the simulation will stop the server-side Method from running at all.

The return value of the Method simulation is discarded, unless the `returnStubValue` option is passed when calling the Method, in which case it is returned to the Method caller. ValidatedMethod passes this option by default.

2. A `method` DDP message is sent to the server

The Meteor client constructs a DDP message to send to the server. This includes the Method name, arguments, and an automatically generated Method ID that represents this particular Method invocation.

3. Method runs on the server

When the server receives the message, it executes the Method code again on the server. The client side version was a simulation that will be rolled back later, but this time it's the real version that is writing to the actual database. Running the actual Method logic on the server is crucial because the server is a trusted environment where we know that security-critical code will run the way we expect.

4. Return value is sent to the client

Once the Method has finished running on the server, it sends a `result` message to the client with the Method ID generated in step 2, and the return value itself. The client stores this for later use, but *doesn't call the Method callback yet*. If you pass the `onResultReceived` option to `Meteor.apply`, that callback is fired.

5. Any DDP publications affected by the Method are updated

If we have any publications on the page that have been affected by the database writes from this Method, the server sends the appropriate updates to the client. Note that the client data system doesn't reveal these updates to the app UI until the next step.

6. `updated` message sent to the client, data replaced with server result, Method callback fires

After the relevant data updates have been sent to the correct client, the server sends back the last message in the Method life cycle - the DDP `updated` message with the relevant Method ID. The client rolls back any changes to client side data made in the Method simulation in step 1, and replaces them with the actual changes sent from the server in step 5.

Lastly, the callback passed to `Meteor.call` actually fires with the return value from step 4. It's important that the callback waits until the client is up to date, so that your Method callback can assume that the client state reflects any changes done inside the Method.

Error case

In the list above, we didn't cover the case when the Method execution on the server throws an error. In that case, there is no return value, and the client gets an error instead. The Method callback is fired instantly with the returned error as the first argument. Read more about error handling in the section about errors below.

Benefits of Methods over REST

We believe Methods provide a much better primitive for building modern applications than REST endpoints built on HTTP. Let's go over some of the things you get for free with Methods that you would have to worry about if using HTTP. The purpose of this section is not to convince you that REST is bad - it's just to remind you that you don't need to handle these things yourself in a Meteor app.

Methods use synchronous-style APIs, but are non-blocking

You may notice in the example Method above, we didn't need to write any callbacks when interacting with MongoDB, but the Method still has the non-blocking properties that people associate with Node.js and callback-style code. Meteor uses a coroutine library called Fibers to enable you to write code that uses return values and throws errors, and avoid dealing with lots of nested callbacks.

Methods always run and return in order

When accessing a REST API, you will sometimes run into a situation where you make two requests one after the other, but the results arrive out of order. Meteor's underlying machinery makes sure this never happens with Methods. When multiple Method calls are received *from the same client*, Meteor runs each Method to completion before starting the next one. If you need to disable this functionality for one particularly long-running Method, you can use `this.unblock()` to allow the next Method to run while the current one is still in progress. Also, since Meteor is based on Websockets instead of HTTP, all Method calls and results are guaranteed to arrive in the order they are sent. You can also pass a special option `wait: true` to `Meteor.apply` to wait to send a particular Method until all others have returned, and not send any other Methods until this one returns.

Change tracking for Optimistic UI

When Method simulations and server-side executions run, Meteor tracks any resulting changes to the database. This is what lets the Meteor data system roll back the changes from the Method simulation and replace them with the actual writes from the server. Without this automatic database tracking, it would be very difficult to implement a correct Optimistic UI system.

Calling a Method from another Method

Sometimes, you'll want to call a Method from another Method. Perhaps you already have some functionality implemented and you want to add a wrapper that fills in some of the arguments automatically. This is a totally fine pattern, and Meteor does some nice things for you:

1. Inside a client-side Method simulation, calling another Method doesn't fire off an extra request to the server - the assumption is that the server-side implementation of the Method will do it. However, it does run the *simulation* of the called Method, so that the simulation on the client closely matches what will happen on the server.

2. Inside a Method execution on the server, calling another Method runs that Method as if it were called by the same client. That means the Method runs as usual, and the context - `userId`, `connection`, etc - are taken from the original Method call.

Consistent ID generation and optimistic UI

When you insert documents into Minimongo from the client-side simulation of a Method, the `_id` field of each document is a random string. When the Method call is executed on the server, the IDs are generated again before being inserted into the database. If it were implemented naively, it could mean that the IDs generated on the server are different, which would cause undesirable flickering and re-renders in the UI when the Method simulation was rolled back and replaced with the server data. But this is not the case in Meteor!

Each Meteor Method invocation shares a random generator seed with the client that called the Method, so any IDs generated by the client and server Methods are guaranteed to be the same. This means you can safely use the IDs generated on the client to do things while the Method is being sent to the server, and be confident that the IDs will be the same when the Method finishes. One case where this is particularly useful is if you want to create a new document in the database, then immediately redirect to a URL that contains that new document's ID.

Method retries

If you call a Method from the client, and the user's Internet connection disconnects before the result is received, Meteor assumes that the Method didn't actually run. When the connection is re-established, the Method call will be sent again. This means that, in certain situations, Methods can be sent more than once. This should only happen very rarely, but in the case where an extra Method call could have negative consequences it is worth putting in extra effort to ensure that Methods are idempotent - that is, calling them multiple times doesn't result in additional changes to the database.

Many Method operations are idempotent by default. Inserts will throw an error if they happen twice because the generated ID will conflict. Removes on collections won't do anything the second time, and most update operators like `$set` will have the same result if run again. The only places you need to worry about code running twice are MongoDB update operators that stack, like `$inc` and `$push`, and calls to external APIs.

Historical comparison with allow/deny

The Meteor core API includes an alternative to Methods for manipulating data from the client. Instead of explicitly defining Methods with specific arguments, you can instead call `insert`, `update`, and `remove` directly from the client and specify security rules with `allow` and `deny`. In the Meteor Guide, we are taking a strong position that this feature should be avoided and Methods used instead. Read more about the problems with allow/deny in the Security article.

Historically, there have been some misconceptions about the features of Meteor Methods as compared with the allow/deny feature, including that it was more difficult to achieve Optimistic UI when using Methods. However, the client-side `insert`, `update`, and `remove` feature is actually implemented *on top of* Methods, so Methods are strictly more powerful. You get great default Optimistic UI by defining your Method code on the client and the server, as described in the Method lifecycle section above.