Publish and subscribe

These functions control how Meteor servers publish sets of records and how clients can subscribe to those sets.

If you prefer to watch the video, click below.

{% youtube RH2RxKgkPJY %}

{% apibox "Meteor.publish" %}

To publish records to clients, call `Meteor.publish` on the server with two parameters: the name of the record set, and a *publish function* that Meteor will call each time a client subscribes to the name.

Publish functions can return a `Collection.Cursor`, in which case Meteor will publish that cursor's documents to each subscribed client. You can also return an array of `Collection.Cursor`s, in which case Meteor will publish all of the cursors.

{% pullquote 'warning' %} If you return multiple cursors in an array, they currently must all be from different collections. We hope to lift this restriction in a future release. {% endpullquote %}

A client will see a document if the document is currently in the published record set of any of its subscriptions. If multiple publications publish a document with the same `_id` to the same collection the documents will be merged for the client. If the values of any of the top level fields conflict, the resulting value will be one of the published values, chosen arbitrarily.

```
// Server: Publish the `Rooms` collection, minus secret info...
Meteor.publish('rooms', function () {
  return Rooms.find({}, {
    fields: { secretInfo: 0 }
  });
});

// ...and publish secret info for rooms where the logged-in user is an admin. If
// the client subscribes to both publications, the records are merged together
// into the same documents in the `Rooms` collection. Note that currently object
// values are not recursively merged, so the fields that differ must be top
// level fields.
```

```
Meteor.publish('adminSecretInfo', function () {
  return Rooms.find({ admin: this.userId }, {
    fields: { secretInfo: 1 }
  });
});

// Publish dependent documents and simulate joins.
Meteor.publish('roomAndMessages', function (roomId) {
  check(roomId, String);

  return [
    Rooms.find({ _id: roomId }, {
      fields: { secretInfo: 0 }
    }),
    Messages.find({ roomId })
  ];
});
```

Alternatively, a publish function can directly control its published record set by calling the functions `added` (to add a new document to the published record set), `changed` (to change or clear some fields on a document already in the published record set), and `removed` (to remove documents from the published record set). These methods are provided by `this` in your publish function.

If a publish function does not return a cursor or array of cursors, it is assumed to be using the low-level `added`/`changed`/`removed` interface, and it **must also call `ready` once the initial record set is complete**.

Example (server):

```
// Publish the current size of a collection.
Meteor.publish('countsByRoom', function (roomId) {
  check(roomId, String);

  let count = 0;
  let initializing = true;

  // `observeChanges` only returns after the initial `added` callbacks have run.
  // Until then, we don't want to send a lot of `changed` messages-hence
  // tracking the `initializing` state.
  const handle = Messages.find({ roomId }).observeChanges({
    added: (id) => {
      count += 1;

      if (!initializing) {
        this.changed('counts', roomId, { count });
      }
    },
    removed: (id) => {
      count -= 1;
      this.changed('counts', roomId, { count });
    }

    // We don't care about `changed` events.
  });

  // Instead, we'll send one `added` message right after `observeChanges` has
  // returned, and mark the subscription as ready.
  initializing = false;
  this.added('counts', roomId, { count });
  this.ready();

  // Stop observing the cursor when the client unsubscribes. Stopping a
  // subscription automatically takes care of sending the client any `removed`
  // messages.
  this.onStop(() => handle.stop());
});

// Sometimes publish a query, sometimes publish nothing.
Meteor.publish('secretData', function () {
  if (this.userId === 'superuser') {
    return SecretData.find();
  } else {
    // Declare that no data is being published. If you leave this line out,
    // Meteor will never consider the subscription ready because it thinks
    // you're using the `added/changed/removed` interface where you have to
    // explicitly call `this.ready`.
    return [];
  }
});
```

Example (client):

```
// Declare a collection to hold the count object.
const Counts = new Mongo.Collection('counts');

// Subscribe to the count for the current room.
Tracker.autorun(() => {
  Meteor.subscribe('countsByRoom', Session.get('roomId'));
});

// Use the new collection.
const roomCount = Counts.findOne(Session.get('roomId')).count;
console.log(`Current room has ${roomCount} messages.`);
```

{% pullquote 'warning' %} Meteor will emit a warning message if you call `Meteor.publish` in a project that includes the `autopublish` package. Your publish function will still work. {% endpullquote %}

Read more about publications and how to use them in the Data Loading article in the Meteor Guide.

{% apibox "Subscription#userId" %}

This is constant. However, if the logged-in user changes, the publish function is rerun with the new value, assuming it didn't throw an error at the previous run.

{% apibox "Subscription#added" %} {% apibox "Subscription#changed" %} {% apibox "Subscription#removed" %} {% apibox "Subscription#ready" %} {% apibox "Subscription#onStop" %}

If you call `observe` or `observeChanges` in your publish handler, this is the place to stop the observes.

{% apibox "Subscription#error" %} {% apibox "Subscription#stop" %} {% apibox "Subscription#connection" %}

{% apibox "Meteor.subscribe" %}

When you subscribe to a record set, it tells the server to send records to the client. The client stores these records in local Minimongo collections, with the same name as the `collection` argument used in the publish handler's `added`, `changed`, and `removed` callbacks. Meteor will queue incoming records until you declare the `Mongo.Collection` on the client with the matching collection name.

```
// It's okay to subscribe (and possibly receive data) before declaring the
// client collection that will hold it. Assume 'allPlayers' publishes data from
// the server's 'players' collection.
Meteor.subscribe('allPlayers');
...

// The client queues incoming 'players' records until the collection is created:
const Players = new Mongo.Collection('players');
```

The client will see a document if the document is currently in the published record set of any of its subscriptions. If multiple publications publish a document with the same `_id` for the same collection the documents are merged for the client. If the values of any of the top level fields conflict, the resulting value will be one of the published values, chosen arbitrarily.

{% pullquote 'warning' %} Currently, when multiple subscriptions publish the same document *only the top level fields* are compared during the merge. This means that if the documents include different sub-fields of the same top level field, not all of them will be available on the client. We hope to lift this restriction in a future release. {% endpullquote %}

The `onReady` callback is called with no arguments when the server marks the subscription as ready. The `onStop` callback is called with a `Meteor.Error` if the subscription fails or is terminated by the server. If the subscription is stopped by calling `stop` on the subscription handle or inside the publication, `onStop` is called with no arguments.

`Meteor.subscribe` returns a subscription handle, which is an object with the following properties:

{% dtdd name:"stop()" %} Cancel the subscription. This will typically result in the server directing the client to remove the subscription's data from the client's cache. {% enddtdd %}

{% dtdd name:"ready()" %} True if the server has marked the subscription as ready. A reactive data source. {% enddtdd %}

{% dtdd name:"subscriptionId" %} The `id` of the subscription this handle is for. When you run `Meteor.subscribe` inside of `Tracker.autorun`, the handles you get will always have the same `subscriptionId` field. You can use this to deduplicate subscription handles if you are storing them in some data structure. {% enddtdd %}

If you call `Meteor.subscribe` within a reactive computation, for example using `Tracker.autorun`, the subscription will automatically be cancelled when the computation is invalidated or stopped; it is not necessary to call `stop` on subscriptions made from inside `autorun`. However, if the next iteration of your run function subscribes to the same record set (same name and parameters), Meteor is smart enough to skip a wasteful unsubscribe/resubscribe. For example:

```
Tracker.autorun(() => {
  Meteor.subscribe('chat', { room: Session.get('currentRoom') });
  Meteor.subscribe('privateMessages');
});
```

This subscribes you to the chat messages in the current room and to your private messages. When you change rooms by calling `Session.set('currentRoom', 'newRoom')`, Meteor will subscribe to the new room's chat messages, unsubscribe from the original room's chat messages, and continue to stay subscribed to your private messages.

Publication strategies

> The following features are available from Meteor 2.4 or `ddp-server@2.5.0`

Once you start scaling your application you might want to have more control on how the data from publications is being handled on the client. There are three publications strategies:

SERVER_MERGE `SERVER_MERGE` is the default strategy. When using this strategy, the server maintains a copy of all data a connection is subscribed to. This allows us to only send deltas over multiple publications.

NO_MERGE_NO_HISTORY The `NO_MERGE_NO_HISTORY` strategy results in the server sending all publication data directly to the client. It does not remember what it has previously sent to client and will not trigger removed messages when a subscription is stopped. This should only be chosen for special use cases like send-and-forget queues.

NO_MERGE `NO_MERGE` is similar to `NO_MERGE_NO_HISTORY` but the server will remember the IDs it has sent to the client so it can remove them when a subscription is stopped. This strategy can be used when a collection is only used in a single publication.

When `NO_MERGE` is selected the client will be handling gracefully duplicate events without throwing an exception. Specifically:

- When we receive an added message for a document that is already present in the client's collection, it will be changed.
- When we receive a change message for a document that is not in the client's collection, it will be added.
- When we receive a removed message for a document that is not in the client's collection, nothing will happen.

You can import the publication strategies from `DDPServer`.

```
import { DDPServer } from 'meteor/ddp-server'

const { SERVER_MERGE, NO_MERGE_NO_HISTORY, NO_MERGE } = DDPServer.publicationStrategies
```

You can use the following methods to set or get the publication strategy for publications:

{% apibox "setPublicationStrategy" %}

For publication `foo`, you can set `NO_MERGE` strategy as shown:

```
import { DDPServer } from "meteor/ddp-server";
Meteor.server.setPublicationStrategy('foo', DDPServer.publicationStrategies.NO_MERGE);
```

{% apibox "getPublicationStrategy" %}