After reading this guide, you'll know:

1. What publications and subscriptions are in the Meteor platform.
2. How to define a publication on the server.
3. Where to subscribe on the client and in which templates.
4. Useful patterns for managing subscriptions.
5. How to reactively publish related data.
6. How to ensure your publication is secure in the face of reactive changes.
7. How to use the low-level publish API to publish anything.
8. What happens when you subscribe to a publication.
9. How to turn a 3rd-party REST endpoint into a publication.
10. How to turn a publication in your app into a REST endpoint.

Publications and subscriptions

In a traditional, HTTP-based web application, the client and server communicate in a "request-response" fashion. Typically the client makes RESTful HTTP requests to the server and receives HTML or JSON data in response, and there's no way for the server to "push" data to the client when changes happen at the backend.

Meteor is built from the ground up on the Distributed Data Protocol (DDP) to allow data transfer in both directions. Building a Meteor app doesn't require you to set up REST endpoints to serialize and send data. Instead you create *publication* endpoints that can push data from server to client.

In Meteor a **publication** is a named API on the server that constructs a set of data to send to a client. A client initiates a **subscription** which connects to a publication, and receives that data. That data consists of a first batch sent when the subscription is initialized and then incremental updates as the published data changes.

So a subscription can be thought of as a set of data that changes over time. Typically, the result of this is that a subscription "bridges" a server-side MongoDB collection, and the client side Minimongo cache of that collection. You can think of a subscription as a pipe that connects a subset of the "real" collection with the client's version, and constantly keeps it up to date with the latest information on the server.

Defining a publication

A publication should be defined in a server-only file. For instance, in the Todos example app, we want to publish the set of public lists to all users:

```
Meteor.publish('lists.public', function() {
  return Lists.find({
    userId: {$exists: false}
  }, {
    fields: Lists.publicFields
  });
});
```

There are a few things to understand about this code block. First, we've named the publication with the unique string `lists.public`, and that will be how we access it from the client. Second, we are returning a Mongo *cursor* from the publication function. Note that the cursor is filtered to only return certain fields from the collection, as detailed in the Security article.

What that means is that the publication will ensure the set of data matching that query is available to any client that subscribes to it. In this case, all lists that do not have a `userId` setting. So the collection named `Lists` on the

client will have all of the public lists that are available in the server collection named `Lists` while that subscription is open. In this particular example in the Todos application, the subscription is initialized when the app starts and never stopped, but a later section will talk about subscription life cycle.

Every publication takes two types of parameters:

1. The `this` context, which has information about the current DDP connection. For example, you can access the current user's `_id` with `this.userId`.
2. The arguments to the publication, which can be passed in when calling `Meteor.subscribe`.

> *Note: Since we need to access context on `this` we need to use the `function() {}` form for publications rather than the ES2015 `() => {}`. You can disable the arrow function linting rule for publication files with `eslint-disable prefer-arrow-callback`. A future version of the publication API will work more nicely with ES2015.*

In this publication, which loads private lists, we need to use `this.userId` to get only the todo lists that belong to a specific user.

```
Meteor.publish('lists.private', function() {
  if (!this.userId) {
    return this.ready();
  }

  return Lists.find({
    userId: this.userId
  }, {
    fields: Lists.publicFields
  });
});
```

Thanks to the guarantees provided by DDP and Meteor's accounts system, the above publication can be confident that it will only ever publish private lists to the user that they belong to. Note that the publication will re-run if the user logs out (or back in again), which means that the published set of private lists will change as the active user changes.

In the case of a logged-out user, we explicitly call `this.ready()`, which indicates to the subscription that we've sent all the data we are initially going to send (in this case none). It's important to know that if you don't return a cursor from the publication or call `this.ready()`, the user's subscription will never become ready, and they will likely see a loading state forever.

Here's an example of a publication which takes a named argument. Note that it's important to check the types of arguments that come in over the network.

```
Meteor.publish('todos.inList', function(listId) {
  // We need to check the `listId` is the type we expect
  new SimpleSchema({
    listId: {type: String}
  }).validate({ listId });

  // ...
});
```

When we subscribe to this publication on the client, we can provide this argument via the `Meteor.subscribe()` call:

```
Meteor.subscribe('todos.inList', list._id);
```

Organizing publications

It makes sense to place a publication alongside the feature that it's targeted for. For instance, sometimes publications provide very specific data that's only really useful for the view for which they were developed. In that case, placing the publication in the same module or directory as the view code makes perfect sense.

Often, however, a publication is more general. For example in the Todos example application, we create a `todos.inList` publication, which publishes all the todos in a list. Although in the application we only use this in one place (in the `Lists_show` template), in a larger app, there's a good chance we might need to access all the todos for a list in other places. So putting the publication in the `todos` package is a sensible approach.

Subscribing to data

To use publications, you need to create a subscription to it on the client. To do so, you call `Meteor.subscribe()` with the name of the publication. When you do this, it opens up a subscription to that publication, and the server starts sending data down the wire to ensure that your client collections contain up to date copies of the data specified by the publication.

`Meteor.subscribe()` also returns a "subscription handle" with a property called `.ready()`. This is a reactive function that returns `true` when the publication is marked ready (either you call `this.ready()` explicitly, or the initial contents of a returned cursor are sent over).

```
const handle = Meteor.subscribe('lists.public');
```

Stopping Subscriptions

The subscription handle also has another important property, the `.stop()` method. When you are subscribing, it is very important to ensure that you always call `.stop()` on the subscription when you are done with it. This ensures that the documents sent by the subscription are cleared from your local Minimongo cache and the server stops doing the work required to service your subscription. If you forget to call stop, you'll consume unnecessary resources both on the client and the server.

However, if you call `Meteor.subscribe()` conditionally inside a reactive context (such as an `autorun`, or `getMeteorData` in React) or via `this.subscribe()` in a Blaze component, then Meteor's reactive system will automatically call `this.stop()` for you at the appropriate time.

Subscribe in UI components

It is best to place the subscription as close as possible to the place where the data from the subscription is needed. This reduces "action at a distance" and makes it easier to understand the flow of data through your application. If the subscription and fetch are separated, then it's not always clear how and why changes to the subscriptions (such as changing arguments), will affect the contents of the cursor.

What this means in practice is that you should place your subscription calls in *components*. In Blaze, it's best to do this in the `onCreated()` callback:

```
Template.Lists_show_page.onCreated(function() {
  this.getListId = () => FlowRouter.getParam('_id');

  this.autorun(() => {
    this.subscribe('todos.inList', this.getListId());
  });
});
```

In this code snippet we can see two important techniques for subscribing in Blaze templates:

1. Calling `this.subscribe()` (rather than `Meteor.subscribe`), which attaches a special `subscriptionsReady()` function to the template instance, which is true when all subscriptions made inside this template are ready.
2. Calling `this.autorun` sets up a reactive context which will re-initialize the subscription whenever the reactive function `this.getListId()` changes.

Read more about Blaze subscriptions in the Blaze article, and about tracking loading state inside UI components in the UI article.

Fetching data

Subscribing to data puts it in your client-side collection. To use the data in your user interface, you need to query your client-side collection. There are a couple of important rules to follow when doing this.

Always use specific queries to fetch data

If you're publishing a subset of your data, it might be tempting to query for all data available in a collection (i.e. `Lists.find()`) in order to get that subset on the client, without re-specifying the Mongo selector you used to publish that data in the first place.

But if you do this, then you open yourself up to problems if another subscription pushes data into the same collection, since the data returned by `Lists.find()` might not be what you expected anymore. In an actively developed application, it's often hard to anticipate what may change in the future and this can be a source of hard to understand bugs.

Also, when changing between subscriptions, there is a brief period where both subscriptions are loaded (see Publication behavior when changing arguments below), so when doing things like pagination, it's exceedingly likely that this will be the case.

Fetch the data nearby where you subscribed to it

We do this for the same reason we subscribe in the component in the first place---to avoid action at a distance and to make it easier to understand where data comes from. A common pattern is to fetch the data in a parent template, and then pass it into a "pure" child component, as we'll see it in the UI Article.

Note that there are some exceptions to this second rule. A common one is `Meteor.user()` ---although this is strictly speaking subscribed to (automatically usually), it's typically over-complicated to pass it through the component hierarchy as an argument to each component. However keep in mind it's best not to use it in too many places as it makes components harder to test.

Global subscriptions

One place where you might be tempted to not subscribe inside a component is when it accesses data that you know you *always* need. For instance, a subscription to extra fields on the user object (see the Accounts Article) that you need on every screen of your app.

However, it's generally a good idea to use a layout component (which you wrap all your components in) to subscribe to this subscription anyway. It's better to be consistent about such things, and it makes for a more flexible system if you ever decide you have a screen that *doesn't* need that data.

Patterns for data loading

Across Meteor applications, there are some common patterns of data loading and management on the client side that are worth knowing. We'll go into more detail about some of these in the UI/UX Article.

Subscription readiness

It is key to understand that a subscription will not instantly provide its data. There will be a latency between subscribing to the data on the client and it arriving from the publication on the server. You should also be aware that this delay may be a lot longer for your users in production than for you locally in development!

Although the Tracker system means you often don't *need* to think too much about this in building your apps, usually if you want to get the user experience right, you'll need to know when the data is ready.

To find that out, `Meteor.subscribe()` and (`this.subscribe()` in Blaze components) returns a "subscription handle", which contains a reactive data source called `.ready()`:

```
const handle = Meteor.subscribe('lists.public');
Tracker.autorun(() => {
  const isReady = handle.ready();
  console.log(`Handle is ${isReady ? 'ready' : 'not ready'}`);
});
```

If you're subscribing to multiple publications, you can create an array of handles and use `every` to determine if all are ready:

```
const handles = [
  Meteor.subscribe('lists.public'),
  Meteor.subscribe('todos.inList'),
];

Tracker.autorun(() => {
  const areReady = handles.every(handle => handle.ready());
  console.log(`Handles are ${areReady ? 'ready' : 'not ready'}`);
});
```

We can use this information to be more subtle about when we try and show data to users, and when we show a loading screen.

Reactively changing subscription arguments

We've already seen an example of using an `autorun` to re-subscribe when the (reactive) arguments to a subscription change. It's worth digging in a little more detail to understand what happens in this scenario.

```
Template.Lists_show_page.onCreated(function() {
  this.getListId = () => FlowRouter.getParam('_id');

  this.autorun(() => {
    this.subscribe('todos.inList', this.getListId());
  });
});
```

In our example, the `autorun` will re-run whenever `this.getListId()` changes, (ultimately because `FlowRouter.getParam('_id')` changes), although other common reactive data sources are:

1. Template data contexts (which you can access reactively with `Template.currentData()`).
2. The current user status (`Meteor.user()` and `Meteor.loggingIn()`).
3. The contents of other application specific client data stores.

Technically, what happens when one of these reactive sources changes is the following:

1. The reactive data source *invalidates* the autorun computation (marks it so that it re-runs in the next Tracker flush cycle).
2. The subscription detects this, and given that anything is possible in next computation run, marks itself for destruction.
3. The computation re-runs, with `.subscribe()` being re-called either with the same or different arguments.
4. If the subscription is run with the *same arguments* then the "new" subscription discovers the old "marked for destruction" subscription that's sitting around, with the same data already ready, and reuses that.
5. If the subscription is run with *different arguments*, then a new subscription is created, which connects to the publication on the server.
6. At the end of the flush cycle (i.e. after the computation is done re-running), the old subscription checks to see if it was re-used, and if not, sends a message to the server to tell the server to shut it down.

Step 4 above is an important detail---that the system cleverly knows not to re-subscribe if the autorun re-runs and subscribes with the exact same arguments. This holds true even if the new subscription is set up somewhere else in the template hierarchy. For example, if a user navigates between two pages that both subscribe to the exact same subscription, the same mechanism will kick in and no unnecessary subscribing will happen.

Publication behavior when arguments change

It's also worth knowing a little about what happens on the server when the new subscription is started and the old one is stopped.

The server *explicitly* waits until all the data is sent down (the new subscription is ready) for the new subscription before removing the data from the old subscription. The idea here is to avoid flicker---you can, if desired, continue to show the old subscription's data until the new data is ready, then instantly switch over to the new subscription's complete data set.

What this means is in general, when changing subscriptions, there'll be a period where you are *over-subscribed* and there is more data on the client than you strictly asked for. This is one very important reason why you should always fetch the same data that you have subscribed to (don't "over-fetch").

Paginating subscriptions

A very common pattern of data access is pagination. This refers to the practice of fetching an ordered list of data one "page" at a time---typically some number of items, say twenty.

There are two styles of pagination that are commonly used, a "page-by-page" style---where you show only one page of results at a time, starting at some offset (which the user can control), and "infinite-scroll" style, where you show an increasing number of pages of items, as the user moves through the list (this is the typical "feed" style user interface).

In this section, we'll consider a publication/subscription technique for the second, infinite-scroll style pagination. The page-by-page technique is a little tricker to handle in Meteor, due to it being difficult to calculate the offset on the client. If you need to do so, you can follow many of the same techniques that we use here and use the `percolate:find-from-publication` package to keep track of which records have come from your publication.

In an infinite scroll publication, we need to add a new argument to our publication controlling how many items to load. Suppose we wanted to paginate the todo items in our Todos example app:

```
const MAX_TODOS = 1000;

Meteor.publish('todos.inList', function(listId, limit) {
  new SimpleSchema({
    listId: { type: String },
    limit: { type: Number }
  }).validate({ listId, limit });

  const options = {
    sort: {createdAt: -1},
    limit: Math.min(limit, MAX_TODOS)
  };

  // ...
});
```

It's important that we set a `sort` parameter on our query (to ensure a repeatable order of list items as more pages are requested), and that we set an absolute maximum on the number of items a user can request (at least in the case where lists can grow without bound).

Then on the client side, we'd set some kind of reactive state variable to control how many items to request:

```
Template.Lists_show_page.onCreated(function() {
  this.getListId = () => FlowRouter.getParam('_id');

  this.autorun(() => {
    this.subscribe('todos.inList',
      this.getListId(), this.state.get('requestedTodos'));
  });
});
```

We'd increment that `requestedTodos` variable when the user clicks "load more" (or perhaps when they scroll to the bottom of the page).

One piece of information that's very useful to know when paginating data is the *total number of items* that you could see. The `tmeasday:publish-counts` package can be useful to publish this. We could add a `Lists.todoCount` publication like so

```
Meteor.publish('Lists.todoCount', function({ listId }) {
  new SimpleSchema({
    listId: {type: String}
  }).validate({ listId });

  Counts.publish(this, `Lists.todoCount.${listId}`, Todos.find({listId}));
});
```

Then on the client, after subscribing to that publication, we can access the count with

```
Counts.get(`Lists.todoCount.${listId}`)
```

Client-side data with reactive stores

In Meteor, persistent or shared data comes over the wire on publications. However, there are some types of data which doesn't need to be persistent or shared between users. For instance, the "logged-in-ness" of the current user, or the route they are currently viewing.

Although client-side state is often best contained as state of an individual template (and passed down the template hierarchy as arguments where necessary), sometimes you have a need for "global" state that is shared between unrelated sections of the template hierarchy.

Usually such state is stored in a *global singleton* object which we can call a store. A singleton is a data structure of which only a single copy logically exists. The current user and the router from above are typical examples of such global singletons.

Types of stores

In Meteor, it's best to make stores *reactive data* sources, as that way they tie most naturally into the rest of the ecosystem. There are a few different packages you can use for stores.

If the store is single-dimensional, you can probably use a `ReactiveVar` to store it (provided by the `reactive-var` package). A `ReactiveVar` has two properties, `get()` and `set()`:

```
DocumentHidden = new ReactiveVar(document.hidden);
$(window).on('visibilitychange', (event) => {
  DocumentHidden.set(document.hidden);
});
```

If the store is multi-dimensional, you may want to use a `ReactiveDict` (from the `reactive-dict` package):

```
const $window = $(window);
function getDimensions() {
  return {
    width: $window.width(),
    height: $window.height()
  };
};

WindowSize = new ReactiveDict();
```

```
WindowSize.set(getDimensions());
$window.on('resize', () => {
  WindowSize.set(getDimensions());
});
```

The advantage of a `ReactiveDict` is you can access each property individually (`WindowSize.get('width')`), and the dict will diff the field and track changes on it individually (so your template will re-render less often for instance).

If you need to query the store, or store many related items, it's probably a good idea to use a Local Collection (see the Collections Article).

Accessing stores

You should access stores in the same way you'd access other reactive data in your templates---that means centralizing your store access, much like you centralize your subscribing and data fetch. For a Blaze template, that's either in a helper, or from within a `this.autorun()` inside an `onCreated()` callback.

This way you get the full reactive power of the store.

Updating stores

If you need to update a store as a result of user action, you'd update the store from an event handler, just like you call Methods.

If you need to perform complex logic in the update (e.g. not just call `.set()` etc), it's a good idea to define a mutator on the store. As the store is a singleton, you can just attach a function to the object directly:

```
WindowSize.simulateMobile = (device) => {
  if (device === 'iphone6s') {
    this.set({width: 750, height: 1334});
  }
}
```

Advanced publications

Sometimes, the mechanism of returning a query from a publication function won't cover your needs. In those situations, there are some more powerful publication patterns that you can use.

Publishing relational data

It's common to need related sets of data from multiple collections on a given page. For instance, in the Todos app, when we render a todo list, we want the list itself, as well as the set of todos that belong to that list.

One way you might do this is to return more than one cursor from your publication function:

```
Meteor.publish('todos.inList', function(listId) {
  new SimpleSchema({
    listId: {type: String}
  }).validate({ listId });

  const list = Lists.findOne(listId);
```

```
  if (list && (!list.userId || list.userId === this.userId)) {
    return [
      Lists.find(listId),
      Todos.find({listId})
    ];
  } else {
    // The list doesn't exist, or the user isn't allowed to see it.
    // In either case, make it appear like there is no list.
    return this.ready();
  }
});
```

However, this example will not work as you might expect. The reason is that reactivity doesn't work in the same way on the server as it does on the client. On the client, if *anything* in a reactive function changes, the whole function will re-run, and the results are fairly intuitive.

On the server however, the reactivity is limited to the behavior of the cursors you return from your publish functions. You'll see any changes to the data that matches their queries, but *their queries will never change*.

So in the case above, if a user subscribes to a list that is later made private by another user, although the `list.userId` will change to a value that no longer passes the condition, the body of the publication will not re-run, and so the query to the `Todos` collection (`{listId}`) will not change. So the first user will continue to see items they shouldn't.

However, we can write publications that are properly reactive to changes across collections. To do this, we use the `reywood:publish-composite` package.

The way this package works is to first establish a cursor on one collection, and then explicitly set up a second level of cursors on a second collection with the results of the first cursor. The package uses a query observer behind the scenes to trigger the subscription to change and queries to re-run whenever the source data changes.

```
Meteor.publishComposite('todos.inList', function(listId) {
  new SimpleSchema({
    listId: {type: String}
  }).validate({ listId });

  const userId = this.userId;

  return {
    find() {
      const query = {
        _id: listId,
        $or: [{userId: {$exists: false}}, {userId}]
      };

      // We only need the _id field in this query, since it's only
      // used to drive the child queries to get the todos
      const options = {
        fields: { _id: 1 }
      };
```

```
      return Lists.find(query, options);
    },

    children: [{
      find(list) {
        return Todos.find({ listId: list._id }, { fields: Todos.publicFields });
      }
    }]
  };
});
```

In this example, we write a complicated query to make sure that we only ever find a list if we are allowed to see it, then, once per list we find (which can be one or zero times depending on access), we publish the todos for that list. Publish Composite takes care of stopping and starting the dependent cursors if the list stops matching the original query or otherwise.

Complex authorization

We can also use `publish-composite` to perform complex authorization in publications. For instance, consider if we had a `Todos.admin.inList` publication that allowed an admin to bypass default publication's security for users with an `admin` flag set.

We might want to write:

```
Meteor.publish('Todos.admin.inList', function({ listId }) {
  new SimpleSchema({
    listId: {type: String}
  }).validate({ listId });

  const user = Meteor.users.findOne(this.userId);

  if (user && user.admin) {
    // We don't need to worry about the list.userId changing this time
    return [
      Lists.find(listId),
      Todos.find({listId})
    ];
  } else {
    return this.ready();
  }
});
```

However, due to the same reasons discussed above, the publication *will not re-run* if the user's `admin` status changes. If this is something that is likely to happen and reactive changes are needed, then we'll need to make the publication reactive. We can do this via the same technique as above however:

```
Meteor.publishComposite('Todos.admin.inList', function(listId) {
  new SimpleSchema({
    listId: {type: String}
  }).validate({ listId });
```

```
  const userId = this.userId;
  return {
    find() {
      return Meteor.users.find({_id: userId, admin: true}, {fields: {admin: 1}});
    },
    children: [{
      find(user) {
        // We don't need to worry about the list.userId changing this time
        return Lists.find(listId);
      }
    },
    {
      find(user) {
        return Todos.find({listId});
      }
    }]
  };
});
```

Note that we explicitly set the `Meteor.users` query fields, as `publish-composite` publishes all of the returned cursors to the client and re-runs the child computations whenever the cursor changes.

Limiting the results serves a double purpose: it both prevents sensitive fields from being disclosed to the client and limits recomputation to the relevant fields only (namely, the `admin` field).

Custom publications with the low level API

In all of our examples so far (outside of using `Meteor.publishComposite()`) we've returned a cursor from our `Meteor.publish()` handlers. Doing this ensures Meteor takes care of the job of keeping the contents of that cursor in sync between the server and the client. However, there's another API you can use for publish functions which is closer to the way the underlying Distributed Data Protocol (DDP) works.

DDP uses three main messages to communicate changes in the data for a publication: the `added`, `changed` and `removed` messages. So, we can similarly do the same for a publication:

```
Meteor.publish('custom-publication', function() {
  // We can add documents one at a time
  this.added('collection-name', 'id', {field: 'values'});

  // We can call ready to indicate to the client that the initial document sent has
been sent
  this.ready();

  // We may respond to some 3rd party event and want to send notifications
  Meteor.setTimeout(() => {
    // If we want to modify a document that we've already added
    this.changed('collection-name', 'id', {field: 'new-value'});

    // Or if we don't want the client to see it any more
    this.removed('collection-name', 'id');
  });
```

```
  // It's very important to clean up things in the subscription's onStop handler
  this.onStop(() => {
    // Perhaps kill the connection with the 3rd party server
  });
});
```

From the client's perspective, data published like this doesn't look any different---there's actually no way for the client to know the difference as the DDP messages are the same. So even if you are connecting to, and mirroring, some esoteric data source, on the client it'll appear like any other Mongo collection.

One point to be aware of is that if you allow the user to *modify* data in the "pseudo-collection" you are publishing in this fashion, you'll want to be sure to re-publish the modifications to them via the publication, to achieve an optimistic user experience.

Subscription lifecycle

Although you can use publications and subscriptions in Meteor via an intuitive understanding, sometimes it's useful to know exactly what happens under the hood when you subscribe to data.

Suppose you have a publication of the following form:

```
Meteor.publish('Posts.all', function() {
  return Posts.find({}, {limit: 10});
});
```

Then when a client calls `Meteor.subscribe('Posts.all')` the following things happen inside Meteor:

1. The client sends a `sub` message with the name of the subscription over DDP.
2. The server starts up the subscription by running the publication handler function.
3. The publication handler identifies that the return value is a cursor. This enables a convenient mode for publishing cursors.
4. The server sets up a query observer on that cursor, unless such an observer already exists on the server (for any user), in which case that observer is re-used.
5. The observer fetches the current set of documents matching the cursor, and passes them back to the subscription (via the `this.added()` callback).
6. The subscription passes the added documents to the subscribing client's connection *mergebox*, which is an on-server cache of the documents that have been published to this particular client. Each document is merged with any existing version of the document that the client knows about, and an `added` (if the document is new to the client) or `changed` (if it is known but this subscription is adding or changing fields) DDP message is sent.

Note that the mergebox operates at the level of top-level fields, so if two subscriptions publish nested fields (e.g. sub1 publishes `doc.a.b = 7` and sub2 publishes `doc.a.c = 8`), then the "merged" document might not look as you expect (in this case `doc.a = {c: 8}`, if sub2 happens second).

7. The publication calls the `.ready()` callback, which sends the DDP `ready` message to the client. The subscription handle on the client is marked as ready.

8. The observer observes the query. Typically, it uses MongoDB's Oplog to notice changes that affect the query. If it sees a relevant change, like a new matching document or a change in a field on a matching document, it calls into the subscription (via `.added()`, `.changed()` or `.removed()`), which again sends the changes to the mergebox, and then to the client via DDP.

This continues until the client stops the subscription, triggering the following behavior:

1. The client sends the `unsub` DDP message.
2. The server stops its internal subscription object, triggering the following effects:
3. Any `this.onStop()` callbacks setup by the publish handler run. In this case, it is a single automatic callback setup when returning a cursor from the handler, which stops the query observer and cleans it up if necessary.
4. All documents tracked by this subscription are removed from the mergebox, which may or may not mean they are also removed from the client.
5. The `nosub` message is sent to the client to indicate that the subscription has stopped.

Working with REST APIs

Publications and subscriptions are the primary way of dealing with data in Meteor's DDP protocol, but lots of data sources use the popular REST protocol for their API. It's useful to be able to convert between the two.

Loading data from a REST endpoint with a publication

As a concrete example of using the low-level API, consider the situation where you have some 3rd party REST endpoint which provides a changing set of data that's valuable to your users. How do you make that data available?

One option would be to provide a Method that proxies through to the endpoint, for which it's the client's responsibility to poll and deal with the changing data as it comes in. So then it's the clients problem to deal with keeping a local data cache of the data, updating the UI when changes happen, etc. Although this is possible (you could use a Local Collection to store the polled data, for instance), it's simpler, and more natural to create a publication that does this polling for the client.

A pattern for turning a polled REST endpoint looks something like this:

```
const POLL_INTERVAL = 5000;

Meteor.publish('polled-publication', function() {
  const publishedKeys = {};

  const poll = () => {
    // Let's assume the data comes back as an array of JSON documents, with an _id
field
    const data = HTTP.get(REST_URL, REST_OPTIONS);

    data.forEach((doc) => {
      if (publishedKeys[doc._id]) {
        this.changed(COLLECTION_NAME, doc._id, doc);
      } else {
        publishedKeys[doc._id] = true;
```

```
        this.added(COLLECTION_NAME, doc._id, doc);
      }
    });
  };

  poll();
  this.ready();

  const interval = Meteor.setInterval(poll, POLL_INTERVAL);

  this.onStop(() => {
    Meteor.clearInterval(interval);
  });
});
```

Things can get more complicated; for instance you may want to deal with documents being removed, or share the work of polling between multiple users (in a case where the data being polled isn't private to that user), rather than doing the exact same poll for each interested user.

Accessing a publication as a REST endpoint

The opposite scenario occurs when you want to publish data to be consumed by a 3rd party, typically over REST. If the data we want to publish is the same as what we already publish via a publication, then we can use the simple:rest package to do this.

In the Todos example app, we have done this, and you can now access our publications over HTTP:

```
$ curl localhost:3000/publications/lists.public
{
  "Lists": [
    {
      "_id": "rBt5iZQnDpRxypu68",
      "name": "Meteor Principles",
      "incompleteCount": 7
    },
    {
      "_id": "Qzc2FjjcfzDy3GdsG",
      "name": "Languages",
      "incompleteCount": 9
    },
    {
      "_id": "TXfWkSkoMy6NByGNL",
      "name": "Favorite Scientists",
      "incompleteCount": 6
    }
  ]
}
```

You can also access authenticated publications (such as `lists.private`). Suppose we've signed up (via the web UI) as `user@example.com`, with the password `password`, and created a private list. Then we can access it as follows:

```
# First, we need to "login" on the commandline to get an access token
$ curl localhost:3000/users/login  -H "Content-Type: application/json" --data
'{"email": "user@example.com", "password": "password"}'
{
  "id": "wq5oLMLi2KMHy5rR6",
  "token": "6PN4EIlwxuVua9PFoaImEP9qzysY64zM6AfpBJCE6bs",
  "tokenExpires": "2016-02-21T02:27:19.425Z"
}

# Then, we can make an authenticated API call
$ curl localhost:3000/publications/lists.private -H "Authorization: Bearer
6PN4EIlwxuVua9PFoaImEP9qzysY64zM6AfpBJCE6bs"
{
  "Lists": [
    {
      "_id": "92XAn3rWhjmPEga4P",
      "name": "My Private List",
      "incompleteCount": 5,
      "userId": "wq5oLMLi2KMHy5rR6"
    }
  ]
}
```

Scaling updates

As previously mentioned, Meteor uses MongoDB's Oplog to identify which changes to apply to which publication. Each change to the database is processed by every Meteor server, so frequent changes can result in high CPU usage across the board. At the same time, your database will come under higher load as all your servers keep fetching data from the oplog.

To solve this issue, you can use the `cultofcoders:redis-oplog` package, which opts out completely from using MongoDB's Oplog and shifts the communication to Redis' Pub/Sub system.

The caveats:

1. You may not need it, if your application is smaller or your data doesn't change a lot.
2. You'll have to maintain another database Redis.
3. Changes that happen outside Meteor instance, must be manually submitted to Redis.

The benefits:

1. Lower load on server CPU and MongoDB.
2. Backwards compatible (no changes required to your publication/subscription code).
3. Better control over which updates should trigger reactivity.
4. You can work with a MongoDB database that does not have oplog enabled.
5. Full control over reactivity using Vent.

```
meteor add cultofcoders:redis-oplog
meteor add disable-oplog
```

In your `settings.json` file:

```
{
    "redisOplog": {}
}
```

```
# Start Redis, then run Meteor
meteor run --settings settings.json
```

Read more about `redis-oplog` here: https://github.com/cult-of-coders/redis-oplog