Accounts (multi-server)

The `accounts-base` package exports two constructors, called `AccountsClient` and `AccountsServer`, which are used to create the `Accounts` object that is available on the client and the server, respectively.

This predefined `Accounts` object (along with similar convenience methods of `Meteor`, such as `Meteor.logout`) is sufficient to implement most accounts-related logic in Meteor apps. Nevertheless, these two constructors can be instantiated more than once, to create multiple independent connections between different accounts servers and their clients, in more complicated authentication situations.

{% apibox "AccountsCommon" %}

The `AccountsClient` and `AccountsServer` classes share a common superclass, `AccountsCommon`. Methods defined on `AccountsCommon.prototype` will be available on both the client and the server, via the predefined `Accounts` object (most common) or any custom `accountsClientOrServer` object created using the `AccountsClient` or `AccountsServer` constructors (less common).

Here are a few of those methods:

{% apibox "AccountsCommon#userId" %}

{% apibox "AccountsCommon#user" %}

{% apibox "AccountsCommon#config" %}

From Meteor 2.5 you can set these in your Meteor settings under `Meteor.settings.packages.accounts-base`. Note that due to the nature of settings file you won't be able to set parameters that require functions.

{% apibox "AccountsCommon#onLogin" %}

See description of AccountsCommon#onLoginFailure for details.

{% apibox "AccountsCommon#onLoginFailure" %}

Either the `onLogin` or the `onLoginFailure` callbacks will be called for each login attempt. The `onLogin` callbacks are called after the user has been successfully logged in. The `onLoginFailure` callbacks are called after a login attempt is denied.

These functions return an object with a single method, `stop`. Calling `stop()` unregisters the callback.

On the server, the callbacks get a single argument, the same attempt info object as `validateLoginAttempt`. On the client, the callback argument is an object containing a single `error` property set to the `Error`-object which was received from the failed login attempt.

{% apibox "AccountsCommon#onLogout" %}

On the server, the `func` callback receives a single argument with the object below. On the client, no arguments are passed.

{% dtdd name:"user" type:"Object" %} The Meteor user object of the user which just logged out. {% enddtdd %}

{% dtdd name:"connection" type:"Object" %} The `connection` object the request came in on. See `Meteor.onConnection` for details. {% enddtdd %}

{% apibox "AccountsClient" %}

At most one of `options.connection` and `options.ddpUrl` should be provided in any instantiation of `AccountsClient`. If neither is provided, `Meteor.connection` will be used as the `.connection` property of the `AccountsClient` instance.

Note that `AccountsClient` is currently available only on the client, due to its use of browser APIs such as `window.localStorage`. In principle, though, it might make sense to establish a client connection from one server to another remote accounts server. Please let us know if you find yourself needing this server-to-server functionality.

These methods are defined on `AccountsClient.prototype`, and are thus available only on the client:

{% apibox "AccountsClient#loggingIn" %}

{% apibox "AccountsClient#logout" %}

{% apibox "AccountsClient#logoutOtherClients" %}

{% apibox "AccountsServer" %}

These methods are defined on `AccountsServer.prototype`, and are thus available only on the server:

{% apibox "AccountsServer#validateNewUser" %}

This can be called multiple times. If any of the functions return `false` or throw an error, the new user creation is aborted. To set a specific error message (which will be displayed by `accounts-ui`), throw a new `Meteor.Error`.

Example:

```
// Validate username, sending a specific error message on failure.
Accounts.validateNewUser((user) => {
  if (user.username && user.username.length >= 3) {
    return true;
  } else {
    throw new Meteor.Error(403, 'Username must have at least 3 characters');
  }
});

// Validate username, without a specific error message.
Accounts.validateNewUser((user) => {
  return user.username !== 'root';
});
```

If the user is being created as part of a login attempt from a client (eg, calling `Accounts.createUser` from the client, or logging in for the first time with an external service), these callbacks are called *before* the `Accounts.validateLoginAttempt` callbacks. If these callbacks succeed but those fail, the user will still be created but the connection will not be logged in as that user.

{% apibox "AccountsServer#onCreateUser" %}

Use this when you need to do more than simply accept or reject new user creation. With this function you can programatically control the contents of new user documents.

The function you pass will be called with two arguments: `options` and `user`. The `options` argument comes from `Accounts.createUser` for password-based users or from an external service login flow. `options` may come from an untrusted client so make sure to validate any values you read from it. The `user` argument is created on the server and contains a proposed user object with all the automatically generated fields required for the user to log in, including the `_id`.

The function should return the user document (either the one passed in or a newly-created object) with whatever modifications are desired. The returned document is inserted directly into the `Meteor.users` collection.

The default create user function simply copies `options.profile` into the new user document. Calling `onCreateUser` overrides the default hook. This can only be called once.

Example:

```
// Support for playing D&D: Roll 3d6 for dexterity.
Accounts.onCreateUser((options, user) => {
  const customizedUser = Object.assign({
    dexterity: _.random(1, 6) + _.random(1, 6) + _.random(1, 6),
  }, user);
```

```
  // We still want the default hook's 'profile' behavior.
  if (options.profile) {
    customizedUser.profile = options.profile;
  }

  return customizedUser;
});
```

{% apibox “AccountsServer#validateLoginAttempt” %}

Call `validateLoginAttempt` with a callback to be called on login attempts. It returns an object with a single method, `stop`. Calling `stop()` unregisters the callback.

When a login attempt is made, the registered validate login callbacks are called with a single argument, the attempt info object:

{% dtdd name:“type” type:“String” %} The service name, such as “password” or “twitter”. {% enddtdd %}

{% dtdd name:“allowed” type:“Boolean” %} Whether this login is allowed and will be successful (if not aborted by any of the validateLoginAttempt callbacks). False if the login will not succeed (for example, an invalid password or the login was aborted by a previous validateLoginAttempt callback). {% enddtdd %}

{% dtdd name:“error” type:“Exception” %} When `allowed` is false, the exception describing why the login failed. It will be a `Meteor.Error` for failures reported to the user (such as invalid password), and can be a another kind of exception for internal errors. {% enddtdd %}

{% dtdd name:“user” type:“Object” %} When it is known which user was attempting to login, the Meteor user object. This will always be present for successful logins. {% enddtdd %}

{% dtdd name:“connection” type:“Object” %} The `connection` object the request came in on. See `Meteor.onConnection` for details. {% enddtdd %}

{% dtdd name:“methodName” type:“String” %} The name of the Meteor method being used to login. {% enddtdd %}

{% dtdd name:“methodArguments” type:“Array” %} An array of the arguments passed to the login method. {% enddtdd %}

A validate login callback must return a truthy value for the login to proceed. If the callback returns a falsy value or throws an exception, the login is aborted. Throwing a `Meteor.Error` will report the error reason to the user.

All registered validate login callbacks are called, even if one of the callbacks aborts the login. The later callbacks will see the `allowed` field set to `false` since the login will now not be successful. This allows later callbacks to override an

error from a previous callback; for example, you could override the “Incorrect password” error with a different message.

Validate login callbacks that aren't explicitly trying to override a previous error generally have no need to run if the attempt has already been determined to fail, and should start with

```
if (!attempt.allowed) {
  return false;
}
```

{% apibox “AccountsServer#beforeExternalLogin” %}

Use this hook if you need to validate that user from an external service should be allowed to login or create account.

{% dtdd name:“type” type:“String” %} The service name, such as “google” or “twitter”. {% enddtdd %}

{% dtdd name:“data” type:“Object” %} Data retrieved from the service {% enddtdd %}

{% dtdd name:“user” type:“Object” %} If user was found in the database that matches the criteria from the service, their data will be provided here. {% enddtdd %}

You should return a `Boolean` value, `true` if the login/registration should proceed or `false` if it should terminate. In case of termination the login attempt will throw an error `403`, with the message: `Login forbidden`.

{% apibox “AccountsServer#setAdditionalFindUserOnExternalLogin” %}

When allowing your users to authenticate with an external service, the process will eventually call `Accounts.updateOrCreateUserFromExternalService`. By default, this will search for a user with the `service.<servicename>.id`, and if not found will create a new user. As that is not always desirable, you can use this hook as an escape hatch to look up a user with a different selector, probably by `emails.address` or `username`. Note the function will only be called if no user was found with the `service.<servicename>.id` selector.

The function will be called with a single argument, the info object:

{% dtdd name:“serviceName” type:“String” %} The external service name, such as “google” or “twitter”. {% enddtdd %}

{% dtdd name:“serviceData” type:“Object” %} The data returned by the service oauth request. {% enddtdd %}

{% dtdd name:“options” type:“Exception” %} An optional arugment passed down from the oauth service that may contain additional user profile information. As the data in `options` comes from an external source, make sure you validate any values you read from it. {% enddtdd %}

The function should return either a user document or `undefined`. Returning a user will result in the populating the `service.<servicename>` in your user document, while returning `undefined` will result in a new user account being created. If you would prefer that a new account not be created, you could throw an error instead of returning.

Example:

```
// If a user has already been created, and used their Google email, this will
// allow them to sign in with the Meteor.loginWithGoogle method later, without
// creating a new user.
Accounts.setAdditionalFindUserOnExternalLogin(({serviceName, serviceData}) => {
  if (serviceName === "google") {
     // Note: Consider security implications. If someone other than the owner
     // gains access to the account on the third-party service they could use
     // the e-mail set there to access the account on your app.
     // Most often this is not an issue, but as a developer you should be aware
     // of how bad actors could play.
    return Accounts.findUserByEmail(serviceData.email)
  }
})
```

Rate Limiting

By default, there are rules added to the `DDPRateLimiter` that rate limit logins, new user registration and password reset calls to a limit of 5 requests per 10 seconds per session. These are a basic solution to dictionary attacks where a malicious user attempts to guess the passwords of legitimate users by attempting all possible passwords.

These rate limiting rules can be removed by calling `Accounts.removeDefaultRateLimit()`. Please see the `DDPRateLimiter` docs for more information.