Email

The `email` package allows sending email from a Meteor app. To use it, add the package to your project by running in your terminal:

```
meteor add email
```

There are two ways on how to setup the package for sending e-mail.

First is to set `MAIL_URL`. The server reads from the `MAIL_URL` environment variable to determine how to send mail. The `MAIL_URL` should reference an SMTP server and use the form `smtp://USERNAME:PASSWORD@HOST:PORT` or `smtps://USERNAME:PASSWORD@HOST:PORT`. The `smtps://` form (the `s` is for "secure") should be used if the mail server requires TLS/SSL (and does not use `STARTTLS`) and is most common on port 465. Connections which start unencrypted prior to being upgraded to TLS/SSL (using `STARTTLS`) typically use port 587 (and *sometimes* 25) and should use `smtp://`. For more information see the Nodemailer docs

Second, if you are using a one of the supported services you can setup the sending options in your app settings like this:

```
{
  "packages": {
    "email": {
      "service": "Mailgun",
      "user": "postmaster@meteor.com",
      "password": "superDuperPassword"
    }
  }
}
```

The package will take care of the rest.

> If you use a supported service the package will try to match to supported service and use the stored settings instead. You can force this by switching protocol like `smtp` to the name of the service. Though you should only use this as a stop-gap measure and instead set the settings properly.

If neither option is set, `Email.send` outputs the message to standard output instead.

Package setting is only available since Email v2.2

{% apibox "Email.send" %}

You must provide the `from` option and at least one of `to`, `cc`, and `bcc`; all other options are optional.

`Email.send` only works on the server. Here is an example of how a client could use a server method call to send an email. (In an actual application, you'd need to be careful to limit the emails that a client could send, to prevent your server from being used as a relay by spammers.)

```
// Server: Define a method that the client can call.
Meteor.methods({
  sendEmail(to, from, subject, text) {
    // Make sure that all arguments are strings.
    check([to, from, subject, text], [String]);

    // Let other method calls from the same client start running, without
    // waiting for the email sending to complete.
    this.unblock();

    Email.send({ to, from, subject, text });
  }
});

// Client: Asynchronously send an email.
Meteor.call(
  'sendEmail',
  'Alice <alice@example.com>',
  'bob@example.com',
  'Hello from Meteor!',
  'This is a test of Email.send.'
);
```

{% apibox "Email.hookSend" %}

`hookSend` is a convenient hook if you want to: prevent sending certain emails, send emails via your own integration instead of the default one provided by Meteor, or do something else with the data. This is especially useful if you want to intercept emails sent by core packages like accounts-password or other packages where you can't modify the email code.

The hook function will receive an object with the options for Nodemailer.

{% apibox "Email.customTransport" %}

`Email.customTransport` is only available since Email v2.2

There are scenarios when you have your own transport set up, be it an SDK for your mailing service or something else. This is where `customTransport` comes

in. If you set this function all sending events will be passed to it (after `hookSend` is run) with an object of the options passed into `send` function with addition of `packageSettings` key which will pass in package settings set in your app settings (if any). It is up to you what you do in that function as it will override the original sending function.

Here is a simple example with Mailgun:

```
import { Email } from 'meteor/email'
import { Log } from 'meteor/logging'
import Mailgun from 'mailgun-js'

Email.customTransport = (data) => {
  // `options.packageSettings` are settings from `Meteor.settings.packages.email`
  // The rest of the options are from Email.send options
  const mailgun = Mailgun({ apiKey: data.packageSettings.mailgun.privateKey, domain: 'mg.myg

  // Since the data object that we receive already includes the correct key names for sendi
  // we can just pass it to the mailgun sending message.
  mailgun.messages().send(data, (error, body) => {
    if (error) Log.error(error)
    if (body) Log.info(body)
  })
}
```

> Note that this also overrides the development display of messages in console so you might want to differentiate between production and development for setting this function.