Django send html email with template

Prior to Django 1.7 (currently in dev), we don't have any option to pass html version of an email to "send_mail()" from django.core.mail module. Even in the newer version, you can pass html template as an alternative but you have to pass a plain text version as well. What if you want to send html version, only? This is what I did – from django.core.mail import EmailMessage def send_email(to_list, subject, message, sender="Aircourts "): msg = EmailMessage(subject, message, sender, to_list) msg.content_subtype = "html" # Main content is now text/html return msg.send() from django.core.mail import EmailMessage   msg = EmailMessage(subject, message, sender, to_list)   msg.content_subtype = "html" # Main content is now text/html Now you would pass html as the message parameter. I use django templates with "render_to_string()" function from django.template.loader module. Do you know any other, perhaps smarter way to accomplish this? Please share in the comments! ☺ In this article we will continue with Python Django user registration emails – showing you how to send system/app emails from you Django app. At this stage – you should have a running Django Application, with a user profile registration process already designed. Check out our previous articled on this: Create Django Project from scratchDjango user registration process Typically when a user creates an account on your web application – you want to send them system emails. There are many reasons why you would want to send a user emails, including: Letting them know that you have registered their account successfullyGiving them useful information about your web applications, telling them the next steps etcLetting your users know if any significant changes have been made on their accountHelping a user reset their password, if they forget itSend users regular system updatesSubscription information python django user registration emails Creating a HTML Email Template The first step in sending HTML emails from Django – is to find the ideal email template to work with. The main requirements of a HTML email template is: inline CSS. You need to include all the CSS code inside of the HTML file. You can get some HTML template codes here to start with: Download the file, save it inside of your templates folder inside of your Django project. HTML Template Context – Sending data to the template You can send data to the template using standard templating methods, like so: Inside your HTML template {{object.title}} {{object.subtitle}} Data that you will send to the template in JSON format: { "object": { "title": "this is a title", "subtitle": "this is a subtitle" } } You will pass this entire object as the context, we will get to that later. Set up the email settings inside of the settings.html file We need an SMTP mail server that we will use to send our emails. Once the mail service is set-up, get the credentials and add the following to the settings.py file at the bottom of the file: . . . . . . #Email settings details EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend' EMAIL_HOST = 'mail.careers-portal.com' EMAIL_USE_TLS = True EMAIL_PORT = 587 EMAIL_HOST_USER = 'no-reply@careers-portal.com' EMAIL_HOST_PASSWORD = '************' . . . . . . Building the Send Email Function send django html emails Read the Django documentation on sending emails: Create a new file called functions.py File imports: from django.conf import settings from django.core.mail import EmailMultiAlternatives from django.template.loader import get_template We are going to be using the EmailMultiAlternatives class from django.core.mail. This class allows us to send multiple-alternative type emails. In our case, we will be sending a plain text mail and HTML text mail. Some mail servers only display plain text – in this case, they will display the plain-text alternatives. Most servers display HTML emails, including gmail. In this case, they will have the option to display the HTML email to the client. HTML is rich in formatting, it allows us to include styling, images, font and all sorts of styles to make our emails look interesting and customised for the user. Send Email function Creating the email data: email = { "title": "Thank your for registering with Careers Portal", "shortDescription": "Welcome to Careers Portal, South Africa's Leading Job Search Engine. These are the next steps.", "subtitle": "Careers Portal - Start your job search process today", "message": "You have successfully registered with Careers Portal. You can now login in to your profile and start creating a profile. We have thousands of jobs just waiting for you to apply. If you experience any issues feel free to contact our support at support@careers-portal.co.za.> } subject = '[Careers Portal] Welcome to Careers Portal' to_email = 'anyemail@gmail.com" This is all the data we will need for now, we then need to create the email function as seen below. The function takes in the data as specified above. def sendEmail(email, subject, to_email): from_email = settings.EMAIL_HOST_USER text_content = """ {} {} {} regards, Careers Portal Support """. format(email['shortDescription'], email['subtitle'], email['message']) html_c = get_template('basic-email.html') d = { 'email': email } html_content = html_c.render(d) msg = EmailMultiAlternatives(subject, text_content, from_email, to_email) msg.attach_alternative(html_content, 'text/html') msg.send() We have to specify the following: from_email: The sender of the email. This is us, so we refer to the variable inside of our settings.py filetext_content: This is the body of the email in plain-text formathtml_c: The loaded html file, using get_template function, which was imported at the top of the file. The name 'basic-email.html' must match the name we gave the html file – that we saved inside of the templates folder of our django app. d: This is the context. We basically just add the email object with the data that needs to go in to the HTML file. html_content: we create this variable by rendering the html_c file with the context d. Finally we create the mgs – mail object, attach the HTML and send it off. This code is suitable for sending one mail, if you are coding it in the view function for example. For multiple mails – like sending a group email, we need more robust code. Python Django User Registration Emails You can now apply this function anywhere in your Django code to integrate sending emails inside of you registration process. Check out our full video tutorial here: Python Django User Registration Emails Transactional emails are an important way to keep users engaged when you're building a community site like Facteroid. It's importation to pay attention to details to keep users clicking on them and coming back. Unfortunately, development work on email is a bit of a chore. You have to either clog your inbox with test emails to test every little change or at best configure another awkward solution to open them in a browser. The restrictive HTML requirements are painful to work with, especially the need to use inline styles. I built some simple infrastructure to make email development workflow so much easier. I've built Facteroid in Django, but the basic ideas should apply to any platform. The Basic Setup Sending a formatted email means generating three strings: the subject, the HTML body, and the plain text body. The most obvious way to do this is to out each part in its own template (say welcome_email_subject.txt, welcome_email_plain.txt, welcome_email_html.txt) and using render_to_string. This can get a bit annoying to manage because you have three separate files to edit and keep in sync. Why not leverage blocks? The django-render-block package is perfect for this. It gives you a render_block_to_string function that works just like render_to_string, but gives you the contents of a specific block. All you have to do now is define a single template with separate blocks for the three parts. Let's encapsulate that in a class class Email: template = "emails/base.html" from_email = settings.DEFAULT_FROM_EMAIL def __init__(self, ctx, to_email): self.to_email = to_email self.subject = render_block_to_string(self.template, 'subject', ctx) self.plain = render_block_to_string(self.template, 'plain', ctx) self.body = render_block_to_string(self.template, 'html', ctx) def send(self): send_mail(self.subject, self.plain, self.from_email, [self.to_email], html_message=self.html) {% block subject %}Welcome to Facteroid{% endblock %} {% block plain %}Hi, This is an example email. Thanks, - The Facteroid Team {% endblock %} {% block html %} Hi, This is an example email. Thanks, The Facteroid Team {% endblock %} All we have to do now is create a template that extends base.html, and override template in a child class of Email and we're on our way. Of course, the HTML and body for this is likely to be a lot more involved than that, with a lot of boilerplace. In fact, the standard way to structure HTML emails involves two nested tables with all the content going into a single cell of the innermost table. It would be pretty wasteful to have to keep repeating that. Of course, we can leverage template inheritance nicely to avoid that. {% block subject %}{% endblock %} {% block plain %}{% endblock %} {% block html %} {% block html_main %} {% endblock %} You received this email because you signed up for an Facteroid account with this email. If you do not want to to hear from us, please reply to let us know. {% endblock %} You can still inherit from base.html, but you don't need to override all of html most of the time, you can just override the html_main block and write a much simpler document. While we're DRYing things up, why not generate the plain body from the HTML one? We can always specify one manually for the most important emails, but the generated body should be good enough in most cases. HTML2Text to the rescue. class Email: template = "emails/base.html" from_email = settings.DEFAULT_FROM_EMAIL def __init__(self, ctx, to_email): self.to_email = to_email self.subject = render_block_to_string(self.template, 'subject', ctx) self.plain = render_block_to_string(self.template, 'plain', ctx) self.body = render_block_to_string(self.template, 'html', ctx) if self.plain == "": h = HTML2Text() h.ignore_images = True h.ignore_emphasis = True h.ignore_tables = True self.plain = h.handle(self.html) def send(self): send_mail(self.subject, self.plain, self.from_email, [self.to_email], html_message=self.html) It's pretty common for an email to take a User object and present some information from related objects. You also want to make it easy for users to unsubscribe from specific types of emails. Those are handled with boolean fields in the user Profile. Let's encapsulate that. class UserEmail(Email): unsubscribe_field = None def __init__(self, ctx, user): if self.unsubscribe_field is None: raise ProgrammingError("Derived class must set unsubscribe_field") self.user = user ctx = { 'user': user, 'unsubscribe_link': user.profile.unsubscribe_link(self.unsubscribe_field) **ctx } super().__init__(ctx, user.email) def send(self): if getattr(self.user.profile, self.unsubscribe_field): super().send() class NotificationEmail(UserEmail): template = 'emails/notification_email.html' unsubscribe_field = 'notification_emails' def __init__(self, user): ctx = { 'notifications': user.notifications.filter(seen=False) } super().__init__(ctx, user) Viewing emails in the browser Now it's pretty easy to write a view to preview emails in a browser with very little effort. # user_emails/views.py @staff_member_required def preview_email(request): email_types = { 'email_confirmation': ConfirmationEmail, 'notification': NotificationEmail, 'welcome': WelcomeEmail, } email_type = request.GET.get('type') if email_type not in email_types: return HttpResponseBadRequest("Invalid email type") if 'user_id' in request.GET: user = get_object_or_404(User, request.GET['user_id']) else: user = request.user email = email_types[email_type](user) if request.GET.get('plain'): text = "Subject: %s%s" % (email.subject, email.plain) return HttpResponse(text, content_type='text/plain') else: # Insert a table with metadata like Subject, To etc. to top of body extra = render_to_string('user_email/metadata.html', {'email': email}) soup = BeautifulSoup(email.html) soup.body.insert(0, BeautifulSoup(extra)) return HttpResponse(soup.encode()) Now I can easily preview emails in the browser. I also use django-live-reload in development, so I can edit template files and code and see the effect instantly in the browser window without having to reload the page. Keeping CSS Sane Another thing that makes developing HTML emails painful is the need to use inline styles. Unlike normal webpages, you can't just have a style block and rely on user agents to render them properly. You really do have to put style="..." attributes on every element you want to style, which makes the simplest thing like "make all my links light blue and remove the underline" rather painful. I made that easier with a little custom template tag that reads styles from an external file, and spits out a style attribute. I could have just defined styles in a dictionary, but with a little bit of help from cssutils can keep it in a .CSS file which makes it play nicely with code editors so I get the correct syntax highlighting, autocomplete etc. # user_email/templatetags/email_tags.py _styles = None @register.simple_tag() def style(names): global _styles if _styles is None or settings.DEBUG: _load_styles() style = ';'.join(_styles.get(name, '') for name in names.split()) return mark_safe('style="%s"' % style) def _load_styles(): global _styles _styles = {} sheet = cssutils.parseFile(finders.find('user_email/email_styles.css')) for rule in sheet.cssRules: for selector in rule.selectorList: _styles[selector.selectorText] = rule.style.cssText Now in my HTML files, all I need to do is this: {% extends 'email/base.html' %} {% load email_tags %} {% block html_main %} Hello {{user.profile.display_name}}, Thanks for signing up! Here's an article to help you get started. GET STARTED {% endblock %} Conclusion I stripped out some of the code from this example to make it simpler, but I have all kinds of additional functionality that can be easily added with this base. The Email base class parses the email and automatically prepends the base URL to any relative links. The send function records a log of sent emails in the database and calls an after_send method if it exists. The after_send function in some of the email classes do housekeeping tasks like record which notifications have already been sent to the user. My view function is a bit more complex so I can preview emails that take more than just a user (but it builds on the same idea). I'd welcome any comments, suggestions, or questions!