**CS673 Software Engineering**

**Team 2 - Communication Tool**

**Software Design Document**

****

|  |  |  |  |
| --- | --- | --- | --- |
| Team Member | Role(s) | Signature | Date |
| Laura Kocubinski | Team Leader | *Laura Kocubinski* | 10/17/2019 |
| Hang Shi | Backup Team Leader  Requirement Leader | *Hang Shi* | 10/17/2019 |
| Xi You | QA Leader |  | 10/17/2019 |
| Jhuanderson Macias | Design and Implementation Leader | *Jhuanderson Macias* | 10/17/2019 |
| Mikhail (Misha) Chertushkin | Security Leader |  | 10/17/2019 |
| Bofeng (Beven) Liu | Configuration Leader |  | 10/17/2019 |
|  |  |  |  |
|  |  |  |  |

**Revision history**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Author** | **Date** | **Change** |
| 0.1 | Laura Kocubinski | 10/3/2019 | First Draft |
| 0.2 | Laura Kocubinski | 10/17/2019 | Second Draft |
| 0.3 | Behdad Shahossini | 10/27/2019 | Third Draft |

# 

# Table of Contents

[Introduction](#_gjdgxs)

[Software Architecture](#_30j0zll)

[Design Patterns](#_1fob9te)

[Key Algorithms](#_3znysh7)

[Classes and Methods](#_2et92p0)

[References](#_tyjcwt)

[Glossary](#_3dy6vkm)

# 

# Introduction

Software design is a representation, or model, of the software to be built [1]. Software design creates the “blueprints” of the software application which allow developers to know exactly how to build the application. As such, software design is an imperative step in the development of a software application.

The main goals of software design for our application are:

* **Sufficiency** - handles the user requirements. This is a top priority.
* **Understandability** - can be understood by the intended audience.
  + The architecture is clearly documented.
  + The code is well commented and easy-to-read. Variables, functions, classes, etc. are given meaningful names.
* **Modularity** - divided into well-defined parts. This facilitates code re-use for other Django projects.
* **Flexibility** - can be readily modified to handle changes in requirements or changes in features. This is imperative as we add new features in the development process.

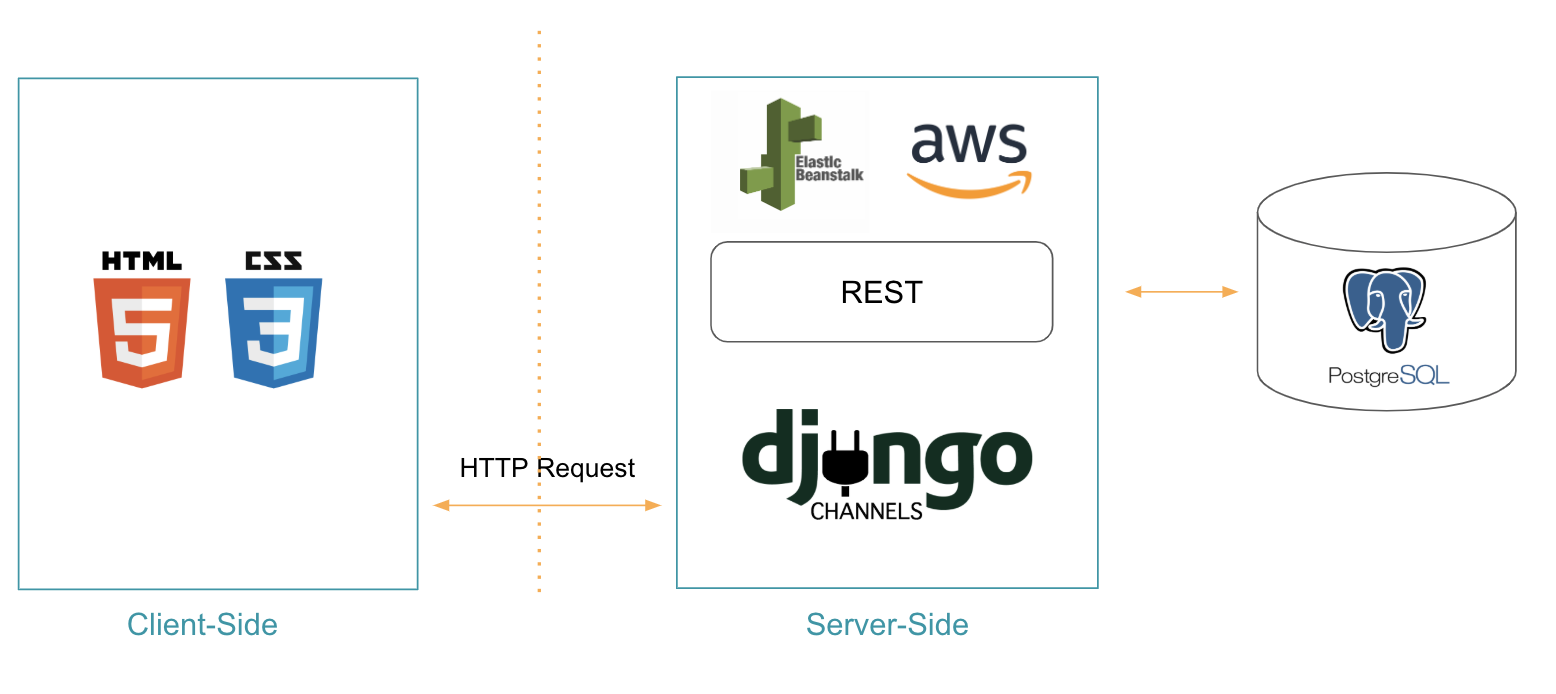
This document will cover the proposed software architecture, design patterns, key algorithms, and classes/methods used for the design of the real-time chat application as described in the [SPPP](https://drive.google.com/open?id=1lnU7U3JHg1hT1AW_P0c9C2IKjOqzdt5v6EGw5lJ14Ro).

# Software Architecture

## System Architecture

### Block Diagram

A crude, high-level block diagram of the web application can be found below.

Figure 1 - System block diagram.

The technologies which are used for the web application can be seen above. These technologies are also described below.

* Front-End
  + HTML/CSS, JavaScript
  + Bootstrap front architecture
* Back-End
  + Django 2.2.5, Django Channels 2.3.0, Python 3.7.4
  + Amazon
    - AWS Elastic Beanstalk
    - Relational Database Service (RDS)
      * PostgreSQL 11.5

This will be elaborated on in the following sections (2.2 and 2.3).

## Client-Side: Bootstrap (HTML, CSS, JavaScript)

### Front-End Framework:Bootstrap

We have chosen to use Bootstrap as the front-end framework. Bootstrap, from Twitter, is a popular front-end framework. Bootstrap is based on HTML, CSS, and Javascript. It is simple and flexible, making web development faster. Bootstrap provides elegant HTML and CSS specifications, which are written by the dynamic CSS language. Bootstrap has been popular since its launch and has been a popular open source project on GitHub, including NASA's MSNBC (Microsoft National Broadcasting Corporation) Breaking News. Some familiar mobile developers in China, such as the WeX5 front-end open source framework, are also based on Bootstrap source code for performance optimization. It includes several specifications:

1. Cross-device browser
   1. Even compatible with IE7/8; generally do not consider browsers below IE9.
2. Responsive layout
   1. It can not only support the display of various resolutions on the PC but also supports the mobile screen display.
3. Provide comprehensive components
   1. Provides a wide range of useful components, including navigation, labels, toolbars, buttons, and more.
4. Built-in jQuery plugin
   1. Many built-in useful jQuery plugins.
5. Support HTML5, CSS3
   1. HTML5 semantic tags and CSS3 properties are well supported.
6. Support LESS dynamic style
   1. LESS utility variables, nesting, operation hybrid coding, writing faster and more flexible CSS, can be well developed with Bootstrap.

#### File Structure



Figure 2 - Bootstrap library structure.

Bootstrap quick tips: Introduced at the bottom of the body, so that the web page is opened without loading the js library, which can improve the page loading speed.

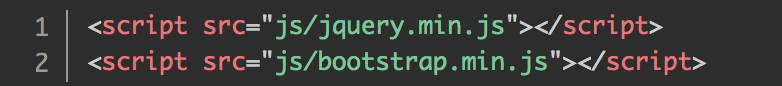


Figure 3 - Bootstrap quick tips.

## Server-Side

### Server: AWS Elastic Beanstalk

For this project we have chosen to use Amazon Web Services (AWS) Elastic Beanstalk to deploy and manage the web application in the AWS cloud. Elastic Beanstalk is an orchestration service which orchestrates various AWS services, such as Amazon Elastic Compute Cloud (EC2), capacity provisioning, load balancing, scaling, and application health monitoring. Elastic Beanstalk provides an additional layer of abstraction over the bare server and OS and can support an application developed in Python. As such, for our project, we see a pre-built combination of OS and platform (i.e. Python 3.6 running on 64-bit Amazon Linux/2.9.2).

### Back-End Framework: Django/Django Channels

#### **Django**

We have chosen to use Django, a Python-based open-source web framework for the back-end. Django follows the Model-Template-View (MTV) architecture, as can be seen below.

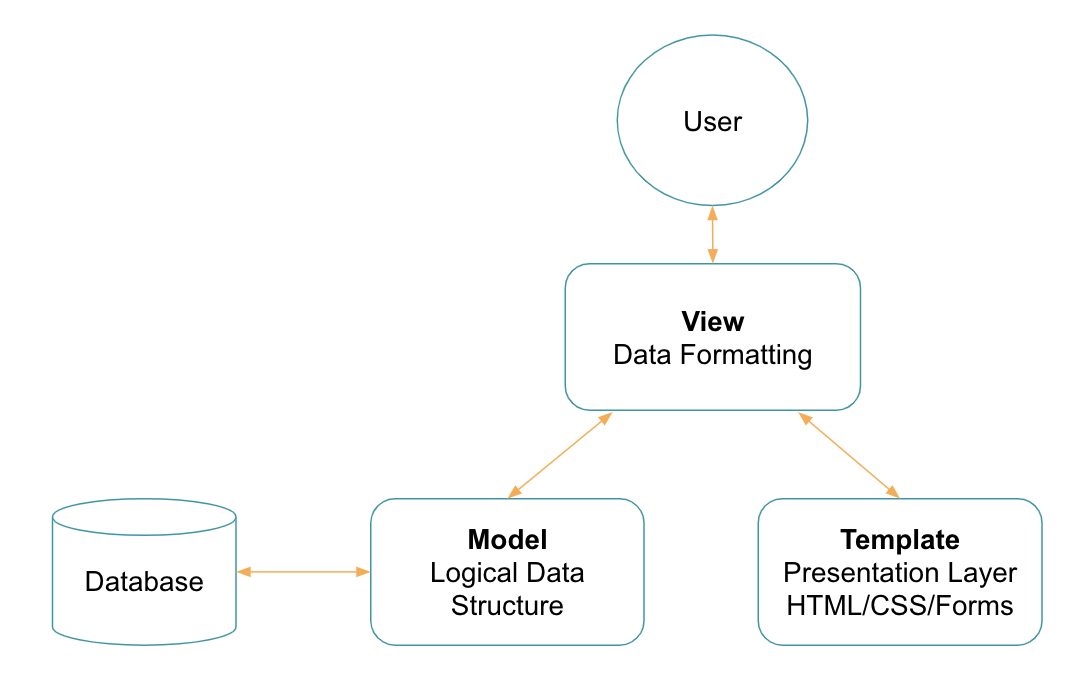


Figure 4 - Django MTV architecture.

The following figure describes the high-level data flow between Django components when handling HTTP requests/responses.

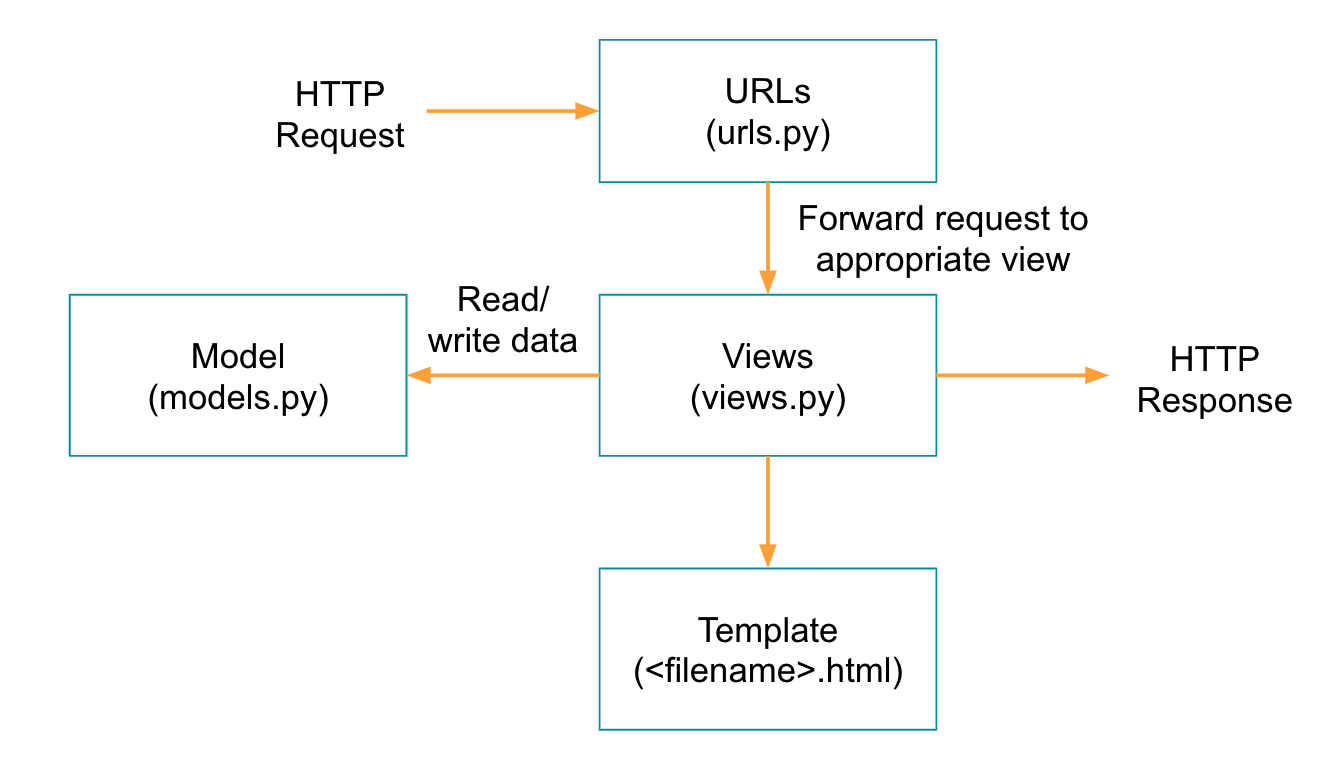


Figure 5 - Django data flow.

The main components of a Django application are:

* URL mappers (urls.py)
  + Forward the matching URLs to the appropriate view functions.
* View functions (views.py)
  + Describes the data that gets presented to the user. The view describes which data you see, not how you see it. Processes an HTTP request, fetches the required data from the database, renders the data in an HTML page using an HTML template, and then returns the generated HTML in an HTTP response to display the page to the user.
* Model (models.py)
  + Source of information about the data; it contains the essential fields and behaviors of the data. In general, each model maps to a database.
* Templates
  + Describes how the data is presented. Used to render data in the views.

Traditional Django is suited for rendering static web pages, such as login, and registration.

#### Django Channels

While vanilla Django is not well-suited for a real-time chat application, Django Channels is.

Django Channels is a project that takes vanilla Django and extends its abilities beyond HTTP to handle WebSockets, chat protocols, IoT protocols, and more. It’s built on a Python specification called [ASGI](http://asgi.readthedocs.io/) (Asynchronous Server Gateway Interface). ASGI allows us to use WebSocket receivers. This gives us the capability to send, listen and receive messages on specific web sockets.  
  
Django channels allows communication between client/server to be synchronous or asynchronous, or a mixture of both. A chat application must be event-driven and should be asynchronous. Meanwhile we can use traditional, synchronous Django for other apps (such as registration, and login mentioned above).

In the context of Django channels, a channel layer is a kind of communication system. It allows multiple consumer instances to talk with each other, and with other parts of Django.

A consumer is the basic unit of Channels code. When a request of a new socket comes in, Channels follows it routing table to find the right consumer for the incoming connection and starts a copy of it. In Django Channels, a consumer is the equivalent of views. As such, a chat consumer handles chat messaging and notification.

##### File Structure

The high-level file structure of Django can be seen below. We have about, chat, users, and direct\_messaging apps.  
  
/communication\_app  
 /communication\_app  
 /about  
 /chat  
 /users  
 /direct\_messaging

The division of apps allows Django code to be modular. Each app has a very similar file structure and must include:

* admin.py
* apps.py
* models.py
* tests.py
* urls.py
* views.py

Below we can see some of the files inside the chat app. In addition to the normal files, we need to include a consumers.py which handles the WebSocket on the back-end.

/communication\_app  
 /communication\_app  
 routing.py  
 settings.py   
 urls.py   
 /about  
 /chat  
 admin.py  
 apps.py  
 consumers.py  
 routing.py  
 tests.py  
 urls.py  
 models.py  
 views.py  
 /templates

/chat

some\_file.html

/users  
 /direct\_messaging

### Database: Amazon RDS (PostgreSQL)

Our group also decided to use Amazon Relational Database Service (RDS) to setup and operate our database. Free databases offered by Amazon Relational Database Services (RDS), include PostgreSQL, MySQL, MariaDB, Oracle BYOL, and SQL Server.

Django is intended to work with a relational database. Out of the box, it automatically creates a SQLite database for a project and is setup to communicate with it. In addition to SQLite, Django officially supports three other popular relational databases: PostgreSQL, MySQL, and Oracle SQL.

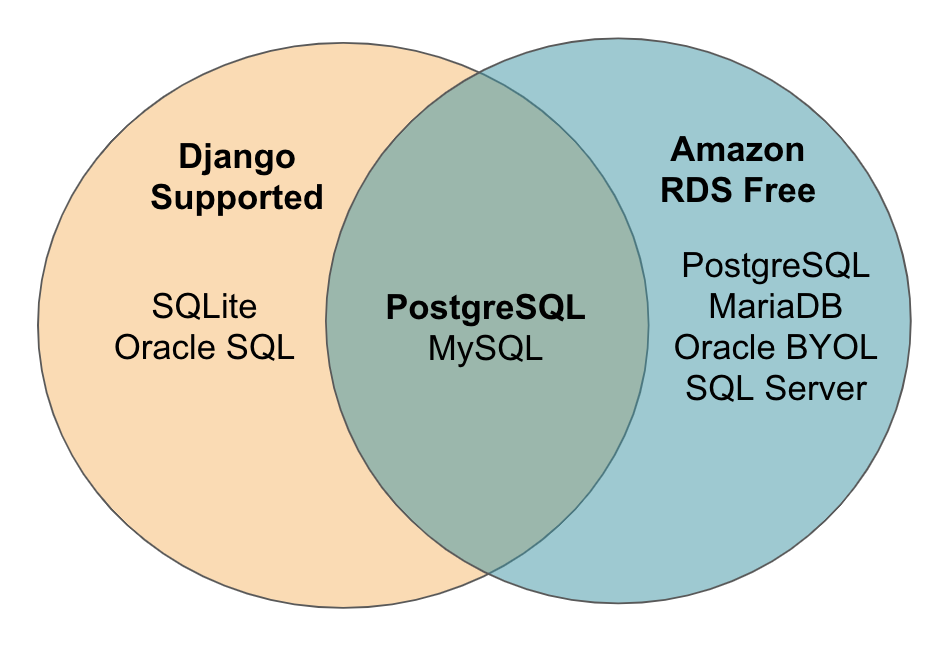


Figure 6 - Venn diagram of database selection.

PostgreSQL is included as a free database in Amazon Relational Database Services (RDS) and is also supported by Django, so this is the RDBMS that will be used.

Below is the ER-Diagram for the PostgreSQL database.

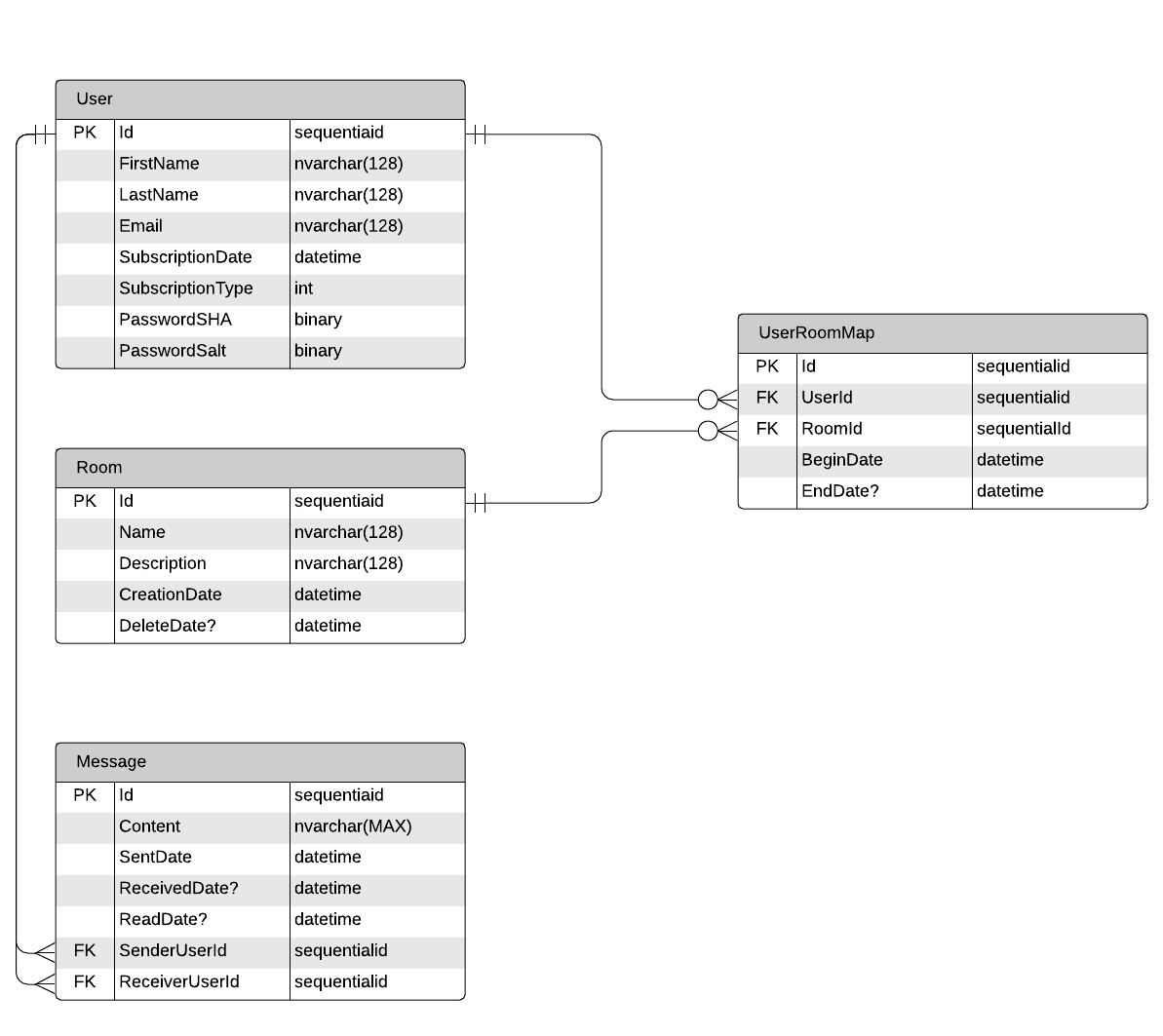


Figure 7 - Database ER-D.

A User may join several Rooms and a Room may have many Users. As such, the relationships between User and Room is Many-to-Many (M:N) and may be described with an associative entity.

Additionally a user may send many messages but each message can only have on sender, so this relationship is 1:M.  
  
A direct message is nothing more than a room limited to two users.

## Class Diagrams

### Component 1: Messaging

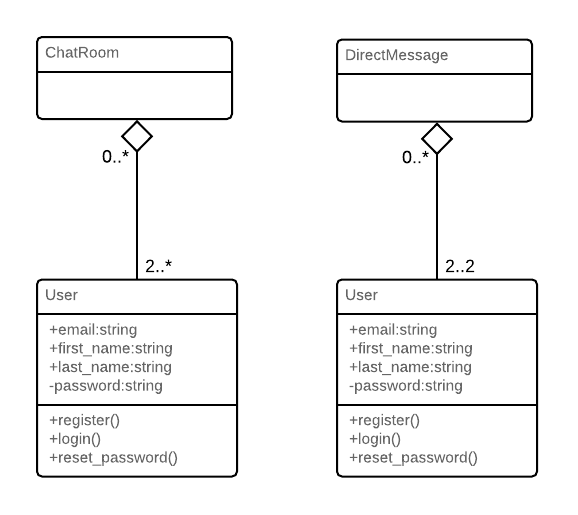


Figure 8 - Class diagram for users messaging.

### Component 2: boundary and control

# Design Patterns

The main design pattern that will use is dictated by the Django framework. We will use a Model View Template (MTV) design. Views are defined as what is presented to the user as described by the templates. Our Model will control the retrieval of data via API. The model will use to fetch the appropriate data needed to be fed into the template or send data to our server as presented by the view.

We have also made use of key features of the factory method pattern. This allows for one point on creation of entities. Currently, this is how we implemented the Chat Rooms**.** Each Chat Room inherits the *room.py* template and is through routing we are able to send the user to the appropriate room.

# Key Algorithms

## Chat Consumer

A key algorithm in our application revolves around allowing users to communicate with one another in designated rooms.

Currently, the main flow of a user includes registering/logging in, creating/joining a room, and then chatting. The following code denotes how we implement the algorithm in our ChatConsumer class, which is a child class of AsyncWebsocketConsumer. The consumer class handles the WebSocket connection and allows for real-time transactions to occur between users of the same “group”.

# create ChatConsumer from AsyncWebsocketConsumer

class ChatConsumer(AsyncWebsocketConsumer):

async def connect(self): # handle webSocket.connect

self.room\_name = self.scope['url\_route']['kwargs']['room\_name']

self.room\_group\_name = 'chat\_%s' % self.room\_name

self.user = self.scope["user"] # get username

# join room group

if rooms.objects.filter(name = self.room\_name).exists() is False:

room\_creation = rooms(name=self.room\_name, description = 'Test Desc',

creation\_date = datetime.utcnow())

room\_creation.save()

await self.channel\_layer.group\_add(

self.room\_group\_name,

self.channel\_name

)

self.rooms = set

await self.accept() # accept WebSocket connection

async def disconnect(self, close\_code): # handle webSocket.disconnect

# leave room group

await self.channel\_layer.group\_discard(

self.room\_group\_name,

self.channel\_name

)

# receive message from WebSocket

async def receive(self, text\_data): # handle webSocket.receive

text\_data\_json = json.loads(text\_data) # load in text\_data

message = text\_data\_json['message'] # grab message from text\_data

room\_id = rooms.objects.get(name=self.room\_name).id

message\_db = messages(room\_id=room\_id,

sender\_user\_id = self.scope["user"].username,

receiver\_user\_id = 'User2',

content = message,

sent\_date = datetime.utcnow(),

received\_date = datetime.utcnow()

)

message\_db.save()

# send message to corresponding room group

await self.channel\_layer.group\_send( # send message

self.room\_group\_name,

{

'type': 'chat\_message',

'message': message,

'username': self.scope["user"].username

}

)

# receive message from room group

# every chat consumer in group (room) will receive the message

async def chat\_message(self, event):

message = event['message']

username = event['username']

# send message back to WebSocket

await self.send(text\_data=json.dumps({

'message': message,

'username': username

}))

## Room.html

Of course, the WebSocket needs it client-side counterpart. The WebSocket is handled on the front-end in room.html.

<!-- chat/templates/chat/room.html -->

{% extends "chat/base.html" %}

{% block content %}

<body>

<textarea id="chat-log" cols="100" rows="20">

{% for i in prev\_messages %}

{{ i.sender\_user\_id }} : {{ i.content}}

{% endfor %}

{{ messages.content }}

</textarea><br/>

<input id="chat-message-input" type="text" size="100"/><br/> <!-- text box -->

<input id="chat-message-submit" type="button" value="Send"/> <!-- send button -->

</body>

<script>

var roomName = {{ room\_name\_json }}; // store roomName as room\_name\_json

var username = '{{user.username}}';

var chatSocket = new WebSocket( // create new WebSocket object chatSocket when render room.html

'ws://' + window.location.host + // create url to connect to

'/ws/chat/' + roomName + '/');

chatSocket.onmessage = function(e) { // when chatSocket receives a message event

var data = JSON.parse(e.data); // get message event data

var message = data['message']; // from data get message

var userN = data['username']; // from data get username

console.log(message); // use for debug

document.querySelector('#chat-log').value += (userN + ": " + message + '\n'); // append chat log with new username and message

};

chatSocket.onclose = function(e) { // when chatSocket closes

console.error('Chat socket closed unexpectedly');

};

document.querySelector('#chat-message-input').focus();

document.querySelector('#chat-message-input').onkeyup = function(e) { // when key is pressed

if (e.keyCode === 13) { // if user presses enter key

document.querySelector('#chat-message-submit').click(); // force onclick event

}

};

document.querySelector('#chat-message-submit').onclick = function(e) { // onclick event (when user clicks send button)

var messageInputDom = document.querySelector('#chat-message-input'); // get value from chat-message-input text box

var message = messageInputDom.value;

var my\_message = JSON.stringify({'message': message, 'name': '{{user.username}}'}) // convert to JSON

chatSocket.send(my\_message); // send message (data) to server over webSocket

messageInputDom.value = ''; // reset value of input to empty string

};

</script>

{% endblock content %}

# Classes and Methods

Django provides an automatic admin interface. Django may use an admin documentation generator (“admindocs”). Django’s admindocs app pulls documentation from the docstrings of models, views, template tags, and template filters for any app in INSTALLED\_APPS and makes that documentation available from the Django admin.

See Appendix A for the automatically generated documentation.

# User Interface (UI)

The flow diagram of the UI can be seen below.

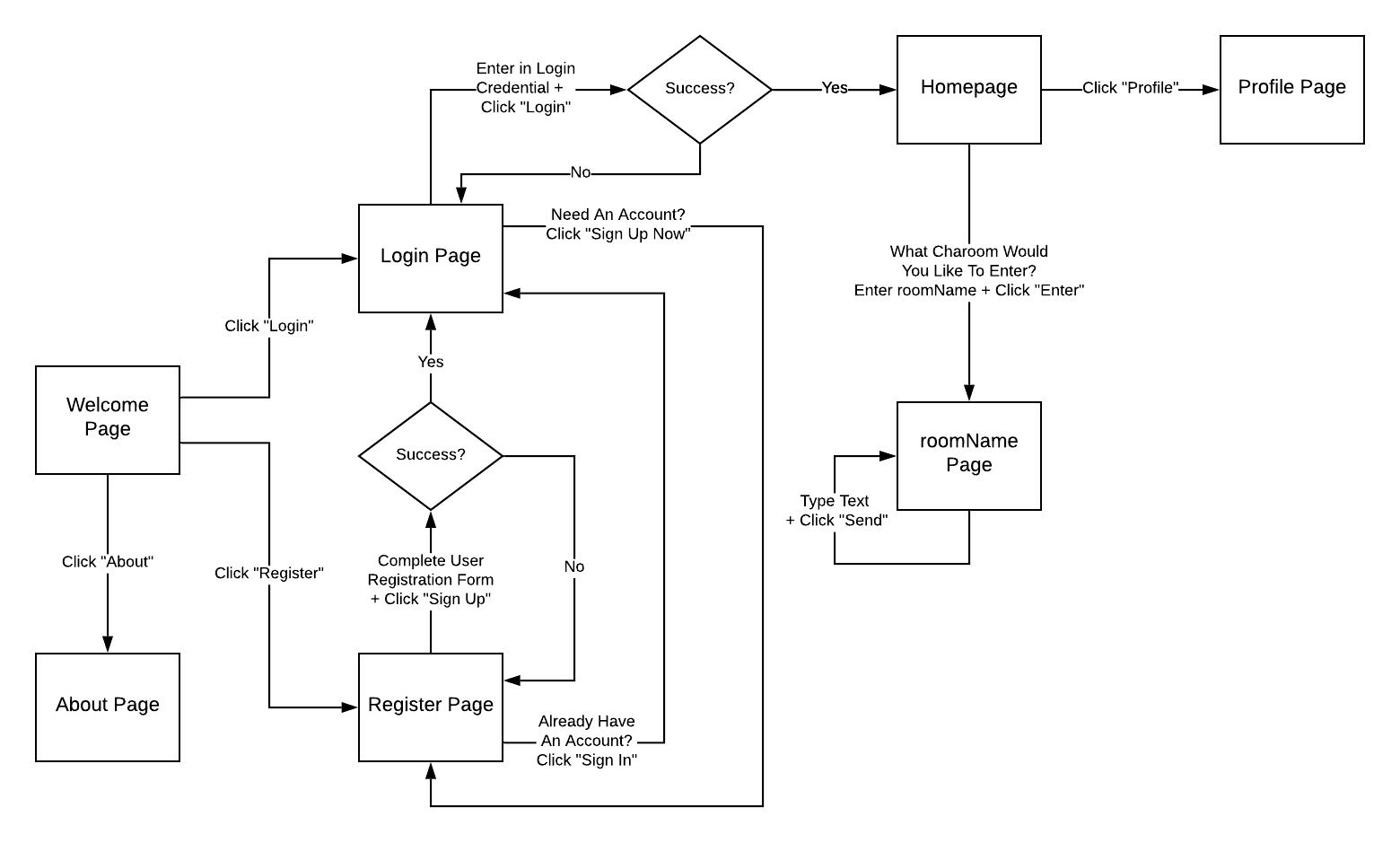
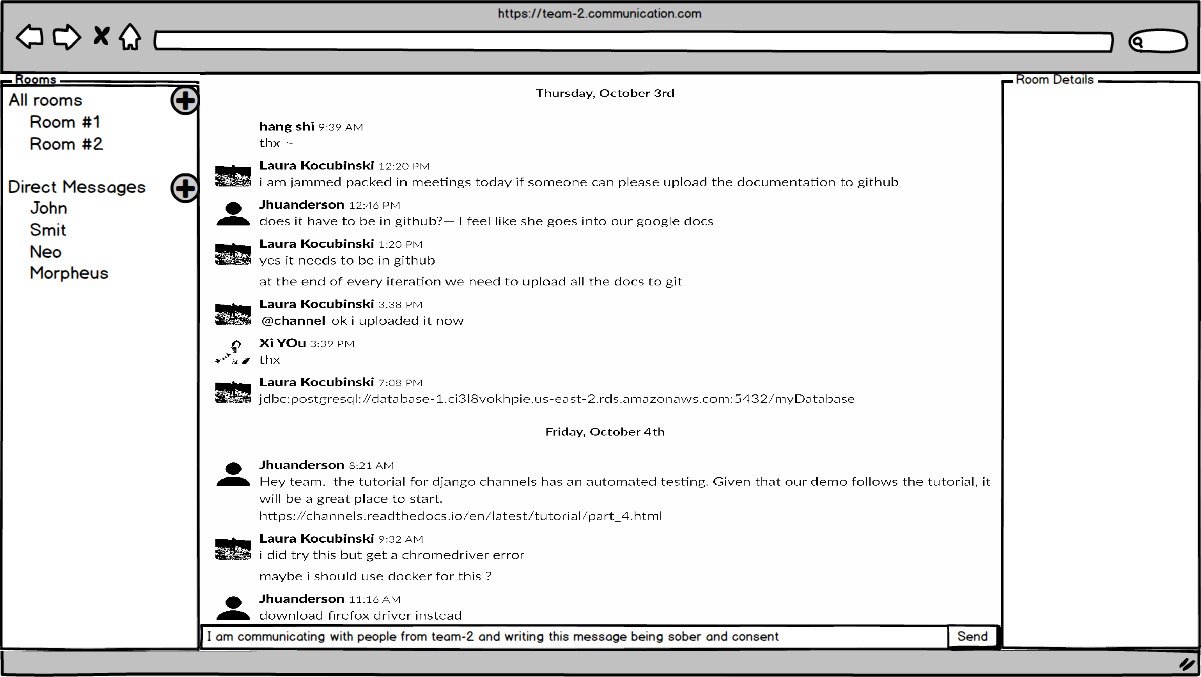
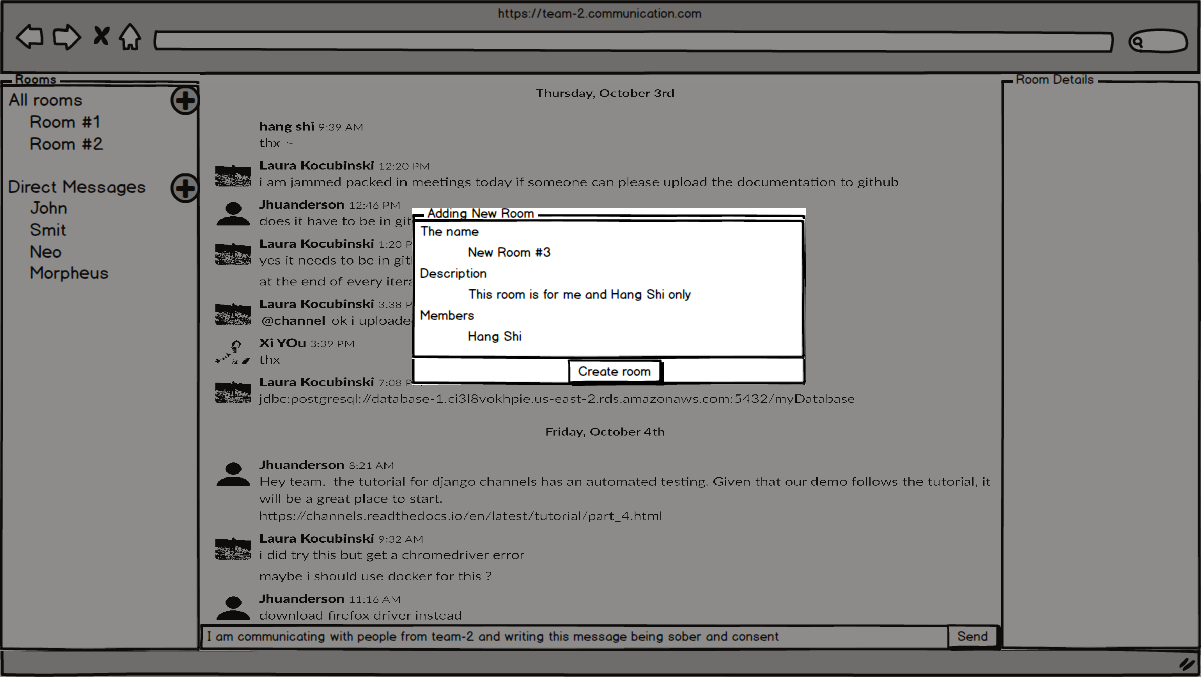


Figure 9 - UI flow diagram.

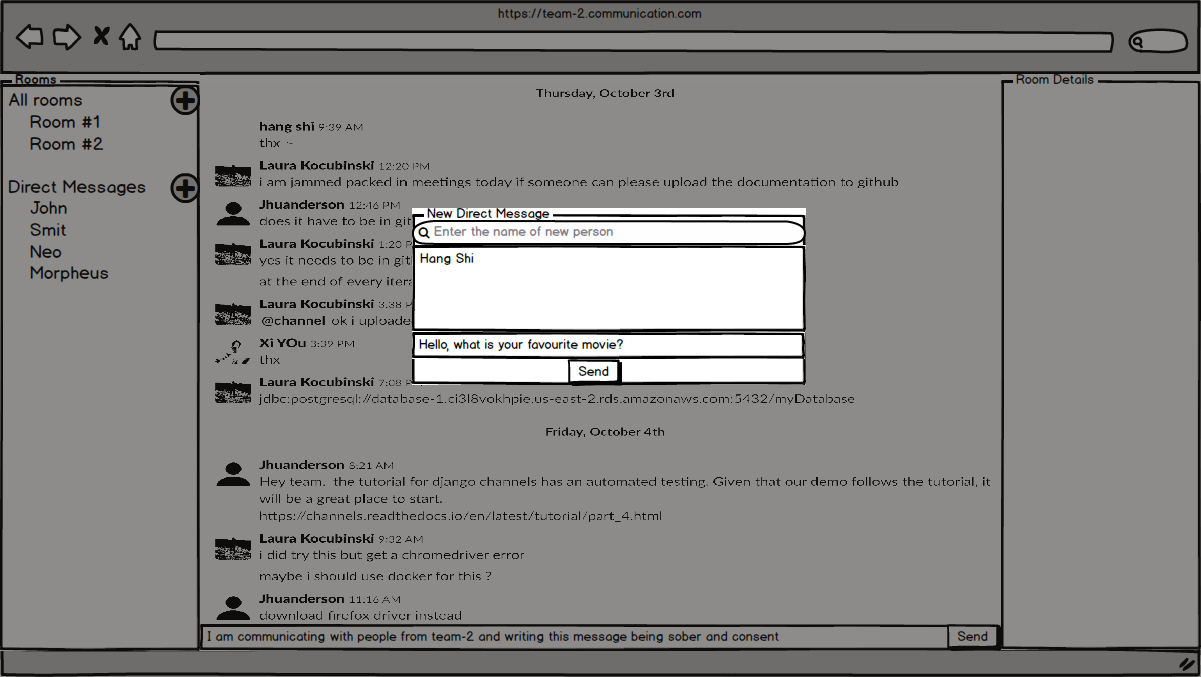
UI mockups of some of the main pages/functions may be seen below.



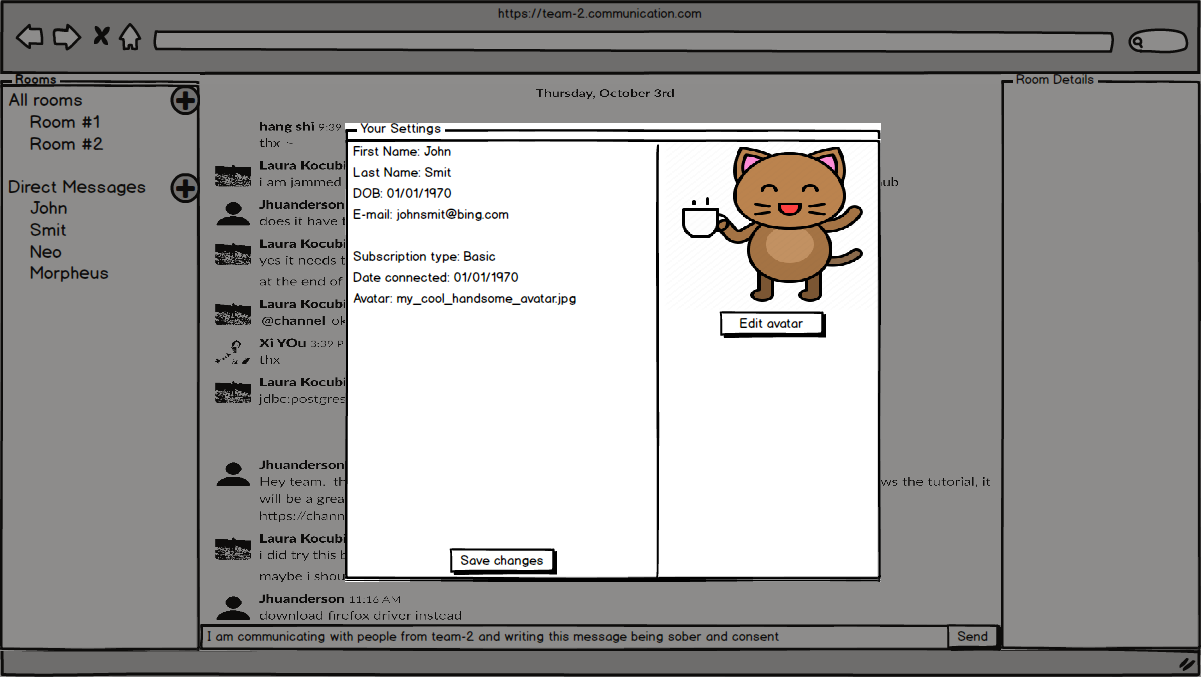
1. roomName page.



1. roomName page - add room.



(c) roomName page - create direct message.



(d) Edit profile.

Figure 10 - UI mockups.

# References

1. Braude, E. J., & Bernstein, M. E. (2016). Software Engineering: Modern Approaches. Long Grove: Waveland Press.
2. AWS Elastic Beanstalk. *Wikipedia*. <https://en.wikipedia.org/wiki/AWS_Elastic_Beanstalk>
3. AWS Elastic Beanstalk. *AWS Documentation.* <https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html>
4. ASGI Documentation. *Read The Docs.*  <http://asgi.readthedocs.io/>
5. Django Channel. *Read The Docs.* [https://channels.readthedocs.io/](https://channels.readthedocs.io/en/latest/)
6. Django (web framework). *Wikipedia.* <https://en.wikipedia.org/wiki/Django_(web_framework)>
7. Tutorial Part 2: Implement a Chat Server. *Django Channels.* <https://channels.readthedocs.io/en/latest/tutorial/part_2.html>
8. React (web framework). *Wikipedia*. <https://en.wikipedia.org/wiki/React_(web_framework)>
9. Django Tutorial: The Local Library Website. *MDN Web Docs*. <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Tutorial_local_library_website>
10. User Authentication in Django. *Django Documentation*. <https://docs.djangoproject.com/en/2.2/topics/auth/>
11. Starter Template. *Bootstrap.* <https://getbootstrap.com/docs/4.0/getting-started/introduction/#starter-template>

# The Django Admin Documentation Generator. *Django Documentation.* [*https://docs.djangoproject.com/en/2.2/ref/contrib/admin/admindocs/*](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/admindocs/)

# Glossary

# Appendix

## View Documentation

Views by empty namespace

[/about/](http://127.0.0.1:8000/admin/doc/views/about.views.index/)

View function: about.views.index. Name: about-index.

[/admin/doc/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admindocs.views.BaseAdminDocsView/)

View function: django.contrib.admindocs.views.BaseAdminDocsView. Name: django-admindocs-docroot.

[/admin/doc/bookmarklets/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admindocs.views.BookmarkletsView/)

View function: django.contrib.admindocs.views.BookmarkletsView. Name: django-admindocs-bookmarklets.

[/admin/doc/filters/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admindocs.views.TemplateFilterIndexView/)

View function: django.contrib.admindocs.views.TemplateFilterIndexView. Name: django-admindocs-filters.

[/admin/doc/models/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admindocs.views.ModelIndexView/)

View function: django.contrib.admindocs.views.ModelIndexView. Name: django-admindocs-models-index.

[/admin/doc/models/<app\_label>\.<model\_name>/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admindocs.views.ModelDetailView/)

View function: django.contrib.admindocs.views.ModelDetailView. Name: django-admindocs-models-detail.

[/admin/doc/tags/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admindocs.views.TemplateTagIndexView/)

View function: django.contrib.admindocs.views.TemplateTagIndexView. Name: django-admindocs-tags.

[/admin/doc/templates/<path:template>/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admindocs.views.TemplateDetailView/)

View function: django.contrib.admindocs.views.TemplateDetailView. Name: django-admindocs-templates.

[/admin/doc/views/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admindocs.views.ViewIndexView/)

View function: django.contrib.admindocs.views.ViewIndexView. Name: django-admindocs-views-index.

[/admin/doc/views/<view>/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admindocs.views.ViewDetailView/)

View function: django.contrib.admindocs.views.ViewDetailView. Name: django-admindocs-views-detail.

[/users/login/](http://127.0.0.1:8000/admin/doc/views/django.contrib.auth.views.LoginView/)

View function: django.contrib.auth.views.LoginView. Name: login.

[/users/logout/](http://127.0.0.1:8000/admin/doc/views/django.contrib.auth.views.LogoutView/)

View function: django.contrib.auth.views.LogoutView. Name: logout.

[/users/profile/](http://127.0.0.1:8000/admin/doc/views/users.views.profile/)

View function: users.views.profile. Name: profile.

[/users/register/](http://127.0.0.1:8000/admin/doc/views/users.views.register/)

View function: users.views.register. Name: register.

Views by namespace admin

[/admin/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.sites.AdminSite.index/)

View function: django.contrib.admin.sites.AdminSite.index. Name: admin:index.

[/admin/<app\_label>/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.sites.AdminSite.app_index/)

View function: django.contrib.admin.sites.AdminSite.app\_index. Name: admin:app\_list.

[/admin/auth/group/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.changelist_view/)

View function: django.contrib.admin.options.ModelAdmin.changelist\_view. Name: admin:auth\_group\_changelist.

[/admin/auth/group/<path:object\_id>/](http://127.0.0.1:8000/admin/doc/views/django.views.generic.base.RedirectView/)

View function: django.views.generic.base.RedirectView. Name: admin.

[/admin/auth/group/<path:object\_id>/change/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.change_view/)

View function: django.contrib.admin.options.ModelAdmin.change\_view. Name: admin:auth\_group\_change.

[/admin/auth/group/<path:object\_id>/delete/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.delete_view/)

View function: django.contrib.admin.options.ModelAdmin.delete\_view. Name: admin:auth\_group\_delete.

[/admin/auth/group/<path:object\_id>/history/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.history_view/)

View function: django.contrib.admin.options.ModelAdmin.history\_view. Name: admin:auth\_group\_history.

[/admin/auth/group/add/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.add_view/)

View function: django.contrib.admin.options.ModelAdmin.add\_view. Name: admin:auth\_group\_add.

[/admin/auth/group/autocomplete/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.autocomplete_view/)

View function: django.contrib.admin.options.ModelAdmin.autocomplete\_view. Name: admin:auth\_group\_autocomplete.

[/admin/auth/user/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.changelist_view/)

View function: django.contrib.admin.options.ModelAdmin.changelist\_view. Name: admin:auth\_user\_changelist.

[/admin/auth/user/<id>/password/](http://127.0.0.1:8000/admin/doc/views/django.contrib.auth.admin.UserAdmin.user_change_password/)

View function: django.contrib.auth.admin.UserAdmin.user\_change\_password. Name: admin:auth\_user\_password\_change.

[/admin/auth/user/<path:object\_id>/](http://127.0.0.1:8000/admin/doc/views/django.views.generic.base.RedirectView/)

View function: django.views.generic.base.RedirectView. Name: admin.

[/admin/auth/user/<path:object\_id>/change/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.change_view/)

View function: django.contrib.admin.options.ModelAdmin.change\_view. Name: admin:auth\_user\_change.

[/admin/auth/user/<path:object\_id>/delete/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.delete_view/)

View function: django.contrib.admin.options.ModelAdmin.delete\_view. Name: admin:auth\_user\_delete.

[/admin/auth/user/<path:object\_id>/history/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.history_view/)

View function: django.contrib.admin.options.ModelAdmin.history\_view. Name: admin:auth\_user\_history.

[/admin/auth/user/add/](http://127.0.0.1:8000/admin/doc/views/django.contrib.auth.admin.UserAdmin.add_view/)

View function: django.contrib.auth.admin.UserAdmin.add\_view. Name: admin:auth\_user\_add.

[/admin/auth/user/autocomplete/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.options.ModelAdmin.autocomplete_view/)

View function: django.contrib.admin.options.ModelAdmin.autocomplete\_view. Name: admin:auth\_user\_autocomplete.

[/admin/jsi18n/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.sites.AdminSite.i18n_javascript/)

View function: django.contrib.admin.sites.AdminSite.i18n\_javascript. Name: admin:jsi18n.

[/admin/login/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.sites.AdminSite.login/)

View function: django.contrib.admin.sites.AdminSite.login. Name: admin:login.

[/admin/logout/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.sites.AdminSite.logout/)

View function: django.contrib.admin.sites.AdminSite.logout. Name: admin:logout.

[/admin/password\_change/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.sites.AdminSite.password_change/)

View function: django.contrib.admin.sites.AdminSite.password\_change. Name: admin:password\_change.

[/admin/password\_change/done/](http://127.0.0.1:8000/admin/doc/views/django.contrib.admin.sites.AdminSite.password_change_done/)

View function: django.contrib.admin.sites.AdminSite.password\_change\_done. Name: admin:password\_change\_done.

[/admin/r/<int:content\_type\_id>/<path:object\_id>/](http://127.0.0.1:8000/admin/doc/views/django.contrib.contenttypes.views.shortcut/)

View function: django.contrib.contenttypes.views.shortcut. Name: admin:view\_on\_site.

Views by namespace chat

[/chat/](http://127.0.0.1:8000/admin/doc/views/chat.views.index/)

View function: chat.views.index. Name: chat:chat-index.

[/chat/<str:room\_name>/](http://127.0.0.1:8000/admin/doc/views/chat.views.room/)

View function: chat.views.room. Name: chat:room.

[/users/](http://127.0.0.1:8000/admin/doc/views/chat.views.index/)

View function: chat.views.index. Name: chat:chat-index.

[/users/<str:room\_name>/](http://127.0.0.1:8000/admin/doc/views/chat.views.room/)

View function: chat.views.room. Name: chat:room.

## Tag Template Documentation

Built-in tags

autoescape

Force autoescape behavior for this block.

block

Define a block that can be overridden by child templates.

comment

Ignore everything between {% comment %} and {% endcomment %}.

csrf\_token

cycle

Cycle among the given strings each time this tag is encountered.

Within a loop, cycles among the given strings each time through the loop:

{% for o in some\_list %}

<tr class="{% cycle 'row1' 'row2' %}">

...

</tr>

{% endfor %}

Outside of a loop, give the values a unique name the first time you call it, then use that name each successive time through:

<tr class="{% cycle 'row1' 'row2' 'row3' as rowcolors %}">...</tr>

<tr class="{% cycle rowcolors %}">...</tr>

<tr class="{% cycle rowcolors %}">...</tr>

You can use any number of values, separated by spaces. Commas can also be used to separate values; if a comma is used, the cycle values are interpreted as literal strings.

The optional flag "silent" can be used to prevent the cycle declaration from returning any value:

{% for o in some\_list %}

{% cycle 'row1' 'row2' as rowcolors silent %}

<tr class="{{ rowcolors }}">{% include "subtemplate.html " %}</tr>

{% endfor %}

debug

Output a whole load of debugging information, including the current context and imported modules.

Sample usage:

<pre>

{% debug %}

</pre>

extends

Signal that this template extends a parent template.

This tag may be used in two ways: {% extends "base" %} (with quotes) uses the literal value "base" as the name of the parent template to extend, or {% extends variable %} uses the value of variable as either the name of the parent template to extend (if it evaluates to a string) or as the parent template itself (if it evaluates to a Template object).

filter

Filter the contents of the block through variable filters.

Filters can also be piped through each other, and they can have arguments -- just like in variable syntax.

Sample usage:

{% filter force\_escape|lower %}

This text will be HTML-escaped, and will appear in lowercase.

{% endfilter %}

Note that the escape and safe filters are not acceptable arguments. Instead, use the autoescape tag to manage autoescaping for blocks of template code.

firstof

Output the first variable passed that is not False.

Output nothing if all the passed variables are False.

Sample usage:

{% firstof var1 var2 var3 as myvar %}

This is equivalent to:

{% if var1 %}

{{ var1 }}

{% elif var2 %}

{{ var2 }}

{% elif var3 %}

{{ var3 }}

{% endif %}

but obviously much cleaner!

You can also use a literal string as a fallback value in case all passed variables are False:

{% firstof var1 var2 var3 "fallback value" %}

If you want to disable auto-escaping of variables you can use:

{% autoescape off %}

{% firstof var1 var2 var3 "<strong>fallback value</strong>" %}

{% autoescape %}

Or if only some variables should be escaped, you can use:

{% firstof var1 var2|safe var3 "<strong>fallback value</strong>"|safe %}

for

Loop over each item in an array.

For example, to display a list of athletes given athlete\_list:

<ul>

{% for athlete in athlete\_list %}

<li>{{ athlete.name }}</li>

{% endfor %}

</ul>

You can loop over a list in reverse by using {% for obj in list reversed %}.

You can also unpack multiple values from a two-dimensional array:

{% for key,value in dict.items %}

{{ key }}: {{ value }}

{% endfor %}

The for tag can take an optional {% empty %} clause that will be displayed if the given array is empty or could not be found:

<ul>

{% for athlete in athlete\_list %}

<li>{{ athlete.name }}</li>

{% empty %}

<li>Sorry, no athletes in this list.</li>

{% endfor %}

<ul>

The above is equivalent to -- but shorter, cleaner, and possibly faster than -- the following:

<ul>

{% if athlete\_list %}

{% for athlete in athlete\_list %}

<li>{{ athlete.name }}</li>

{% endfor %}

{% else %}

<li>Sorry, no athletes in this list.</li>

{% endif %}

</ul>

The for loop sets a number of variables available within the loop:

|  |  |
| --- | --- |
| VARIABLE | DESCRIPTION |
| forloop.counter | The current iteration of the loop (1-indexed) |
| forloop.counter0 | The current iteration of the loop (0-indexed) |
| forloop.revcounter | The number of iterations from the end of the loop (1-indexed) |
| forloop.revcounter0 | The number of iterations from the end of the loop (0-indexed) |
| forloop.first | True if this is the first time through the loop |
| forloop.last | True if this is the last time through the loop |
| forloop.parentloop | For nested loops, this is the loop "above" the current one |

if

Evaluate a variable, and if that variable is "true" (i.e., exists, is not empty, and is not a false boolean value), output the contents of the block:

{% if athlete\_list %}

Number of athletes: {{ athlete\_list|count }}

{% elif athlete\_in\_locker\_room\_list %}

Athletes should be out of the locker room soon!

{% else %}

No athletes.

{% endif %}

In the above, if athlete\_list is not empty, the number of athletes will be displayed by the {{ athlete\_list|count }} variable.

The if tag may take one or several `` {% elif %}`` clauses, as well as an {% else %} clause that will be displayed if all previous conditions fail. These clauses are optional.

if tags may use or, and or not to test a number of variables or to negate a given variable:

{% if not athlete\_list %}

There are no athletes.

{% endif %}

{% if athlete\_list or coach\_list %}

There are some athletes or some coaches.

{% endif %}

{% if athlete\_list and coach\_list %}

Both athletes and coaches are available.

{% endif %}

{% if not athlete\_list or coach\_list %}

There are no athletes, or there are some coaches.

{% endif %}

{% if athlete\_list and not coach\_list %}

There are some athletes and absolutely no coaches.

{% endif %}

Comparison operators are also available, and the use of filters is also allowed, for example:

{% if articles|length >= 5 %}...{% endif %}

Arguments and operators \_must\_ have a space between them, so {% if 1>2 %} is not a valid if tag.

All supported operators are: or, and, in, not in ==, !=, >, >=, < and <=.

Operator precedence follows Python.

ifchanged

Check if a value has changed from the last iteration of a loop.

The {% ifchanged %} block tag is used within a loop. It has two possible uses.

Check its own rendered contents against its previous state and only displays the content if it has changed. For example, this displays a list of days, only displaying the month if it changes:  
<h1>Archive for {{ year }}</h1>

{% for date in days %}

{% ifchanged %}<h3>{{ date|date:"F" }}</h3>{% endifchanged %}

<a href="{{ date|date:"M/d"|lower }}/">{{ date|date:"j" }}</a>

{% endfor %}

If given one or more variables, check whether any variable has changed. For example, the following shows the date every time it changes, while showing the hour if either the hour or the date has changed:  
{% for date in days %}

{% ifchanged date.date %} {{ date.date }} {% endifchanged %}

{% ifchanged date.hour date.date %}

{{ date.hour }}

{% endifchanged %}

{% endfor %}

ifequal

Output the contents of the block if the two arguments equal each other.

Examples:

{% ifequal user.id comment.user\_id %}

...

{% endifequal %}

{% ifnotequal user.id comment.user\_id %}

...

{% else %}

...

{% endifnotequal %}

ifnotequal

Output the contents of the block if the two arguments are not equal. See ifequal.

include

Load a template and render it with the current context. You can pass additional context using keyword arguments.

Example:

{% include "foo/some\_include" %}

{% include "foo/some\_include" with bar="BAZZ!" baz="BING!" %}

Use the only argument to exclude the current context when rendering the included template:

{% include "foo/some\_include" only %}

{% include "foo/some\_include" with bar="1" only %}

load

Load a custom template tag library into the parser.

For example, to load the template tags in django/templatetags/news/photos.py:

{% load news.photos %}

Can also be used to load an individual tag/filter from a library:

{% load byline from news %}

lorem

Create random Latin text useful for providing test data in templates.

Usage format:

{% lorem [count] [method] [random] %}

count is a number (or variable) containing the number of paragraphs or words to generate (default is 1).

method is either w for words, p for HTML paragraphs, b for plain-text paragraph blocks (default is b).

random is the word random, which if given, does not use the common paragraph (starting "Lorem ipsum dolor sit amet, consectetuer...").

Examples:

* {% lorem %} outputs the common "lorem ipsum" paragraph
* {% lorem 3 p %} outputs the common "lorem ipsum" paragraph and two random paragraphs each wrapped in HTML <p> tags
* {% lorem 2 w random %} outputs two random latin words

now

Display the date, formatted according to the given string.

Use the same format as PHP's date() function; see <https://php.net/date> for all the possible values.

Sample usage:

It is {% now "jS F Y H:i" %}

regroup

Regroup a list of alike objects by a common attribute.

This complex tag is best illustrated by use of an example: say that musicians is a list of Musician objects that have name and instrument attributes, and you'd like to display a list that looks like:

* Guitar:
  + Django Reinhardt
  + Emily Remler
* Piano:
  + Lovie Austin
  + Bud Powell
* Trumpet:
  + Duke Ellington

The following snippet of template code would accomplish this dubious task:

{% regroup musicians by instrument as grouped %}

<ul>

{% for group in grouped %}

<li>{{ group.grouper }}

<ul>

{% for musician in group.list %}

<li>{{ musician.name }}</li>

{% endfor %}

</ul>

{% endfor %}

</ul>

As you can see, {% regroup %} populates a variable with a list of objects with grouper and list attributes. grouper contains the item that was grouped by; list contains the list of objects that share that grouper. In this case, grouper would be Guitar, Piano and Trumpet, and list is the list of musicians who play this instrument.

Note that {% regroup %} does not work when the list to be grouped is not sorted by the key you are grouping by! This means that if your list of musicians was not sorted by instrument, you'd need to make sure it is sorted before using it, i.e.:

{% regroup musicians|dictsort:"instrument" by instrument as grouped %}

resetcycle

Reset a cycle tag.

If an argument is given, reset the last rendered cycle tag whose name matches the argument, else reset the last rendered cycle tag (named or unnamed).

spaceless

Remove whitespace between HTML tags, including tab and newline characters.

Example usage:

{% spaceless %}

<p>

<a href="foo/">Foo</a>

</p>

{% endspaceless %}

This example returns this HTML:

<p><a href="foo/">Foo</a></p>

Only space between tags is normalized -- not space between tags and text. In this example, the space around Hello isn't stripped:

{% spaceless %}

<strong>

Hello

</strong>

{% endspaceless %}

templatetag

Output one of the bits used to compose template tags.

Since the template system has no concept of "escaping", to display one of the bits used in template tags, you must use the {% templatetag %} tag.

The argument tells which template bit to output:

|  |  |
| --- | --- |
| ARGUMENT | OUTPUTS |
| openblock | {% |
| closeblock | %} |
| openvariable | {{ |
| closevariable | }} |
| openbrace | { |
| closebrace | } |
| opencomment | {# |
| closecomment | #} |

url

Return an absolute URL matching the given view with its parameters.

This is a way to define links that aren't tied to a particular URL configuration:

{% url "url\_name" arg1 arg2 %}

or

{% url "url\_name" name1=value1 name2=value2 %}

The first argument is a URL pattern name. Other arguments are space-separated values that will be filled in place of positional and keyword arguments in the URL. Don't mix positional and keyword arguments. All arguments for the URL must be present.

For example, if you have a view app\_name.views.client\_details taking the client's id and the corresponding line in a URLconf looks like this:

path('client/<int:id>/', views.client\_details, name='client-detail-view')

and this app's URLconf is included into the project's URLconf under some path:

path('clients/', include('app\_name.urls'))

then in a template you can create a link for a certain client like this:

{% url "client-detail-view" client.id %}

The URL will look like /clients/client/123/.

The first argument may also be the name of a template variable that will be evaluated to obtain the view name or the URL name, e.g.:

{% with url\_name="client-detail-view" %}

{% url url\_name client.id %}

{% endwith %}

verbatim

Stop the template engine from rendering the contents of this block tag.

Usage:

{% verbatim %}

{% don't process this %}

{% endverbatim %}

You can also designate a specific closing tag block (allowing the unrendered use of {% endverbatim %}):

{% verbatim myblock %}

...

{% endverbatim myblock %}

widthratio

For creating bar charts and such. Calculate the ratio of a given value to a maximum value, and then apply that ratio to a constant.

For example:

<img src="bar.png" alt="Bar"

height="10" width="{% widthratio this\_value max\_value max\_width %}">

If this\_value is 175, max\_value is 200, and max\_width is 100, the image in the above example will be 88 pixels wide (because 175/200 = .875; .875 \* 100 = 87.5 which is rounded up to 88).

In some cases you might want to capture the result of widthratio in a variable. It can be useful for instance in a blocktrans like this:

{% widthratio this\_value max\_value max\_width as width %}

{% blocktrans %}The width is: {{ width }}{% endblocktrans %}

with

Add one or more values to the context (inside of this block) for caching and easy access.

For example:

{% with total=person.some\_sql\_method %}

{{ total }} object{{ total|pluralize }}

{% endwith %}

Multiple values can be added to the context:

{% with foo=1 bar=2 %}

...

{% endwith %}

The legacy format of {% with person.some\_sql\_method as total %} is still accepted.

admin\_list

To use these tags, put {% load admin\_list %} in your template before using the tag.

admin\_actions

admin\_list\_filter

change\_list\_object\_tools

Display the row of change list object tools.

date\_hierarchy

pagination

paginator\_number

Generate an individual page index link in a paginated list.

result\_list

search\_form

admin\_modify

To use these tags, put {% load admin\_modify %} in your template before using the tag.

change\_form\_object\_tools

Display the row of change form object tools.

prepopulated\_fields\_js

submit\_row

admin\_static

To use these tags, put {% load admin\_static %} in your template before using the tag.

static

admin\_urls

To use these tags, put {% load admin\_urls %} in your template before using the tag.

add\_preserved\_filters

cache

To use these tags, put {% load cache %} in your template before using the tag.

cache

This will cache the contents of a template fragment for a given amount of time.

Usage:

{% load cache %}

{% cache [expire\_time] [fragment\_name] %}

.. some expensive processing ..

{% endcache %}

This tag also supports varying by a list of arguments:

{% load cache %}

{% cache [expire\_time] [fragment\_name] [var1] [var2] .. %}

.. some expensive processing ..

{% endcache %}

Optionally the cache to use may be specified thus:

{% cache .... using="cachename" %}

Each unique set of arguments will result in a unique cache entry.

crispy\_forms\_field

To use these tags, put {% load crispy\_forms\_field %} in your template before using the tag.

crispy\_addon

Renders a form field using bootstrap's prepended or appended text: .. default-role::

System Message: ERROR/3 (<tag:crispy\_addon>, line 2)

Error in "default-role" directive: no content permitted.

.. default-role:: cmsreference

{% crispy\_addon form.my\_field prepend="$" append=".00" %}

You can also just prepend or append like so

{% crispy\_addon form.my\_field prepend="$" %} {% crispy\_addon form.my\_field append=".00" %}

crispy\_field

{% crispy\_field field attrs %}

crispy\_forms\_filters

To use these tags, put {% load crispy\_forms\_filters %} in your template before using the tag.

crispy

You need to pass in at least the form/formset object, and can also pass in the optional crispy\_forms.helpers.FormHelper object.

helper (optional): A [crispy\_forms.helper.FormHelper](http://127.0.0.1:8000/admin/doc/tags/#crispy_forms.helper.formhelper) object.

Usage:

{% load crispy\_tags %}

{% crispy form form.helper %}

You can also provide the template pack as the third argument:

{% crispy form form.helper 'bootstrap' %}

If the [FormHelper](http://127.0.0.1:8000/admin/doc/tags/#formhelper) attribute is named [helper](http://127.0.0.1:8000/admin/doc/tags/#helper) you can simply do:

{% crispy form %}

{% crispy form 'bootstrap' %}

crispy\_forms\_tags

To use these tags, put {% load crispy\_forms\_tags %} in your template before using the tag.

crispy

You need to pass in at least the form/formset object, and can also pass in the optional crispy\_forms.helpers.FormHelper object.

helper (optional): A [crispy\_forms.helper.FormHelper](http://127.0.0.1:8000/admin/doc/tags/#crispy_forms.helper.formhelper) object.

Usage:

{% load crispy\_tags %}

{% crispy form form.helper %}

You can also provide the template pack as the third argument:

{% crispy form form.helper 'bootstrap' %}

If the [FormHelper](http://127.0.0.1:8000/admin/doc/tags/#formhelper) attribute is named [helper](http://127.0.0.1:8000/admin/doc/tags/#helper) you can simply do:

{% crispy form %}

{% crispy form 'bootstrap' %}

crispy\_forms\_utils

To use these tags, put {% load crispy\_forms\_utils %} in your template before using the tag.

specialspaceless

Removes whitespace between HTML tags, and introduces a whitespace after buttons an inputs, necessary for Bootstrap to place them correctly in the layout.

i18n

To use these tags, put {% load i18n %} in your template before using the tag.

blocktrans

Translate a block of text with parameters.

Usage:

{% blocktrans with bar=foo|filter boo=baz|filter %}

This is {{ bar }} and {{ boo }}.

{% endblocktrans %}

Additionally, this supports pluralization:

{% blocktrans count count=var|length %}

There is {{ count }} object.

{% plural %}

There are {{ count }} objects.

{% endblocktrans %}

This is much like ngettext, only in template syntax.

The "var as value" legacy format is still supported:

{% blocktrans with foo|filter as bar and baz|filter as boo %}

{% blocktrans count var|length as count %}

The translated string can be stored in a variable using [asvar](http://127.0.0.1:8000/admin/doc/tags/#asvar):

{% blocktrans with bar=foo|filter boo=baz|filter asvar var %}

This is {{ bar }} and {{ boo }}.

{% endblocktrans %}

{{ var }}

Contextual translations are supported:

{% blocktrans with bar=foo|filter context "greeting" %}

This is {{ bar }}.

{% endblocktrans %}

This is equivalent to calling pgettext/npgettext instead of (u)gettext/(u)ngettext.

get\_available\_languages

Store a list of available languages in the context.

Usage:

{% get\_available\_languages as languages %}

{% for language in languages %}

...

{% endfor %}

This puts settings.LANGUAGES into the named variable.

get\_current\_language

Store the current language in the context.

Usage:

{% get\_current\_language as language %}

This fetches the currently active language and puts its value into the language context variable.

get\_current\_language\_bidi

Store the current language layout in the context.

Usage:

{% get\_current\_language\_bidi as bidi %}

This fetches the currently active language's layout and puts its value into the bidi context variable. True indicates right-to-left layout, otherwise left-to-right.

get\_language\_info

Store the language information dictionary for the given language code in a context variable.

Usage:

{% get\_language\_info for LANGUAGE\_CODE as l %}

{{ l.code }}

{{ l.name }}

{{ l.name\_translated }}

{{ l.name\_local }}

{{ l.bidi|yesno:"bi-directional,uni-directional" }}

get\_language\_info\_list

Store a list of language information dictionaries for the given language codes in a context variable. The language codes can be specified either as a list of strings or a settings.LANGUAGES style list (or any sequence of sequences whose first items are language codes).

Usage:

{% get\_language\_info\_list for LANGUAGES as langs %}

{% for l in langs %}

{{ l.code }}

{{ l.name }}

{{ l.name\_translated }}

{{ l.name\_local }}

{{ l.bidi|yesno:"bi-directional,uni-directional" }}

{% endfor %}

language

Enable the given language just for this block.

Usage:

{% language "de" %}

This is {{ bar }} and {{ boo }}.

{% endlanguage %}

trans

Mark a string for translation and translate the string for the current language.

Usage:

{% trans "this is a test" %}

This marks the string for translation so it will be pulled out by makemessages into the .po files and runs the string through the translation engine.

There is a second form:

{% trans "this is a test" noop %}

This marks the string for translation, but returns the string unchanged. Use it when you need to store values into forms that should be translated later on.

You can use variables instead of constant strings to translate stuff you marked somewhere else:

{% trans variable %}

This tries to translate the contents of the variable variable. Make sure that the string in there is something that is in the .po file.

It is possible to store the translated string into a variable:

{% trans "this is a test" as var %}

{{ var }}

Contextual translations are also supported:

{% trans "this is a test" context "greeting" %}

This is equivalent to calling pgettext instead of (u)gettext.

l10n

To use these tags, put {% load l10n %} in your template before using the tag.

localize

Force or prevents localization of values, regardless of the value of settings.USE\_L10N.

Sample usage:

{% localize off %}

var pi = {{ 3.1415 }};

{% endlocalize %}

log

To use these tags, put {% load log %} in your template before using the tag.

get\_admin\_log

Populate a template variable with the admin log for the given criteria.

Usage:

{% get\_admin\_log [limit] as [varname] for\_user [context\_var\_containing\_user\_obj] %}

Examples:

{% get\_admin\_log 10 as admin\_log for\_user 23 %}

{% get\_admin\_log 10 as admin\_log for\_user user %}

{% get\_admin\_log 10 as admin\_log %}

Note that context\_var\_containing\_user\_obj can be a hard-coded integer (user ID) or the name of a template context variable containing the user object whose ID you want.

static

To use these tags, put {% load static %} in your template before using the tag.

get\_media\_prefix

Populate a template variable with the media prefix, settings.MEDIA\_URL.

Usage:

{% get\_media\_prefix [as varname] %}

Examples:

{% get\_media\_prefix %}

{% get\_media\_prefix as media\_prefix %}

get\_static\_prefix

Populate a template variable with the static prefix, settings.STATIC\_URL.

Usage:

{% get\_static\_prefix [as varname] %}

Examples:

{% get\_static\_prefix %}

{% get\_static\_prefix as static\_prefix %}

static

Join the given path with the STATIC\_URL setting.

Usage:

{% static path [as varname] %}

Examples:

{% static "myapp/css/base.css" %}

{% static variable\_with\_path %}

{% static "myapp/css/base.css" as admin\_base\_css %}

{% static variable\_with\_path as varname %}

staticfiles

To use these tags, put {% load staticfiles %} in your template before using the tag.

static

tz

To use these tags, put {% load tz %} in your template before using the tag.

get\_current\_timezone

Store the name of the current time zone in the context.

Usage:

{% get\_current\_timezone as TIME\_ZONE %}

This will fetch the currently active time zone and put its name into the TIME\_ZONE context variable.

localtime

Force or prevent conversion of datetime objects to local time, regardless of the value of settings.USE\_TZ.

Sample usage:

{% localtime off %}{{ value\_in\_utc }}{% endlocaltime %}

timezone

Enable a given time zone just for this block.

The timezone argument must be an instance of a tzinfo subclass, a time zone name, or None. If it is None, the default time zone is used within the block.

Sample usage:

{% timezone "Europe/Paris" %}

It is {{ now }} in Paris.

{% endtimezone %}