**Содержание**

[Введение 3](#_Toc7210521)

[1 Позиционирование проекта 4](#_Toc7210522)

[1.1 Описание предметной области 4](#_Toc7210523)

[1.2 Обзор альтернативных решений 5](#_Toc7210524)

[1.3 Постановка задачи 7](#_Toc7210525)

[2 Разработка проекта 8](#_Toc7210526)

[2.1 Функциональные возможности 8](#_Toc7210527)

[2.2 Описание сценариев 9](#_Toc7210528)

[2.3 Информационно-логическая модель данных 13](#_Toc7210529)

[2.4 Обоснование выбора инструментария разработки 14](#_Toc7210530)

[2.5 Архитектура проекта 17](#_Toc7210531)

[3 Реализация приложения 19](#_Toc7210532)

[3.1 Реализация серверной части 19](#_Toc7210533)

[3.2 Реализация подсистемы конвертации 22](#_Toc7210534)

[Заключение 24](#_Toc7210535)

[Cписок использованных источников 25](#_Toc7210536)

# Введение

В рамках преддипломной практики была предоставлена тема «Разработка приложения для конвертации текста в речь средствами нейросетевых технологий».

Главной задачей курсового проекта является ознакомление с предметной областью и разработка её программной реализации.

Разрабатываемая приложение будет размещено в глобальной сети интернет и позволит озвучивать текста или интегрировать его в сторонние приложения.

Аналогом данного приложения является Google Text-to-Speech и Amazon Polly, которые предоставляют возможность интеграции в сторонние приложения.

В функциональные возможности входит перевод текста в речь из нескольких форматов, таких как прямой ввод, текстовый документ, также сохранение, удаление, прослушивание и скачивание голосовой дорожки, регистрация, оплата с посимвольным учетом и возможность пробного конвертирования.

Пояснительная записка курсового проекта включает в себя следующие разделы:

* введение;
* позиционирование проекта;
* разработка проекта;
* реализация приложения;
* тестирование приложения;
* заключение;
* список использованных источников.

Введение к курсовому проекту содержит краткую характеристику области применения программы и наименование объекта, в котором используют программу.

В первом разделе «Позиционирование проекта» описываются требования к разрабатываемому приложению, которые должны быть автоматизированы, перечисляются основные функции программы.

Во втором разделе «Разработка проекта» описываются сведения о программе, структура программы, приводятся диаграммы взаимодействия клиента и программного обеспечения, входные и выходные данные.

В третьем разделе описываются модули проекта, подробная реализация каждой функции модуля, а также взаимодействие функций и модулей между собой.

В заключении к преддипломной практике содержатся: краткие выводы по результатам выполнения преддипломной практики. Список использованных источников содержит сведения о литературных источниках, используемых при написании приложения.

# Позиционирование проекта

## Описание предметной области

Стремление к синтезу голоса, или искусственному производству человеческой речи, начало свою историю еще в 1789 году, задолго до электронной обработки сигналов. Тем не менее, первые успехи появились в 1930-х годах, когда Bell Labs разработала вокодер. С тех пор вычислительная мощность стала дешевле и мощнее, а качество технологии синтеза голоса неуклонно улучшалось. При этом выход из современных систем синтеза голоса по-прежнему остается четко отличимым от реальной человеческой речи.

Генерация речи на сегодняшний день в основном базируется на двух методах преобразования текста в речь: параметрические и конкатенативные. Первый, как следует из названия, основан на высококачественных аудиоклипах (или единицах) записи, которые затем объединяются вместе, чтобы сформировать речь. Хотя сгенерированный речевой звук очень чистый и ясный, но он звучит без эмоций. Понятно, но не естественно. Это связано с тем, что трудно получить аудиозаписи всех возможных слов, произнесенных во всем возможном сочетании эмоций, стресса и т. д. Естественно, эти системы требуют огромных баз данных и жесткого кодирования комбинации для формирования этих слов. Разработка надежной системы занимает от семи до восьми месяцев.

Параметрический метод генерирует речь путем объединения параметров, таких как фундаментальная частота, спектр величины и т. д. и обрабатывает их для генерации речи. Параметрическая система TTS будет иметь два этапа. Сначала нужно извлечь лингвистические особенности после обработки текста. Эти особенности могут быть фонемами, продолжительностью и т. д. Далее, необходимо извлечь особенности вокодера, которые представляют соответствующий речевой сигнал. Эти характеристики могут быть, спектрограммой, основной частотой и т.д. Они представляют собой некоторую неотъемлемую характеристику человеческой речи, и используются в обработке звука.

Эти запрограммированные вручную параметры, наряду с лингвистическими особенностями, подаются в математическую модель, называемую вокодером. Вокодер принимает эти функции и выполняет несколько сложных преобразований над этими функциями для генерации звуковой волны. При генерации сигнала вокодер оценивает такие параметры речи, как фаза, ритм и напряжение, интонация и др.

Теоретически это должно сработать, но практически есть много артефактов, приводящих к приглушенной речи, с постоянно присутствующим жужжащим звуком. Порожденная речь не является ни вразумительной, ни естественной.

Модели глубокого обучения доказали свою исключительную эффективность в изучении внутренних особенностей данных. Эти особенности на самом деле не читаются человеком, но они читаются компьютером, и представляют данные намного лучше для модели. Модель не должна использовать какие-либо ручные инженерные функции, а скорее изучать новые высокомерные функции, чтобы представлять то, что делает речь человеческой.

## Обзор альтернативных решений

Одной из альтернатив является Google Cloud Text-to-Speech сервис. В нем представлена возможность ввода текста, выбор языка, голоса, скорости, и тембра, а также RestAPI, RPC и библиотеки для языков Go, Java, Python, C++, Ruby, Node.js, PHP, C#. Профиль сервиса представлен на рисунке 1.

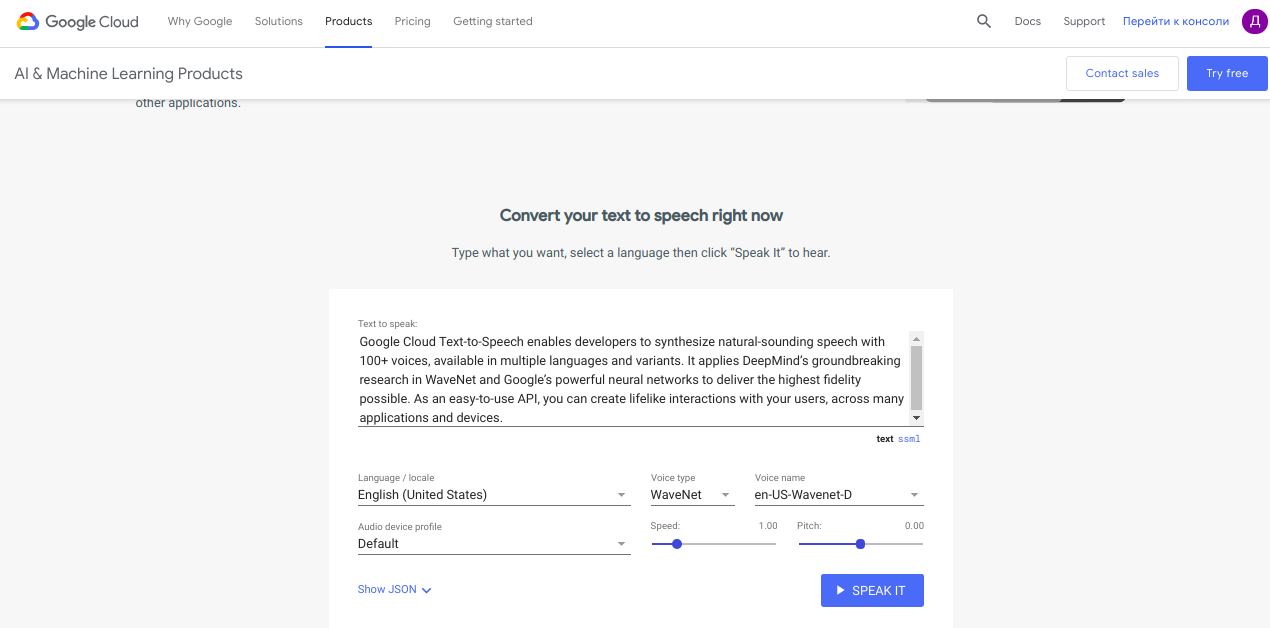


Рисунок 1 – Google сервис

Хотя сервис обладает хорошей документацией, и неплохим качеством речи генерация занимает приличное время.

Второй альтернативой является IBM Watson. Сервис предоставляет всего 8 языков на выбор, RestAPI, а также библиотеки для самых популярных языков. Сервис представлен на рисунке 2.

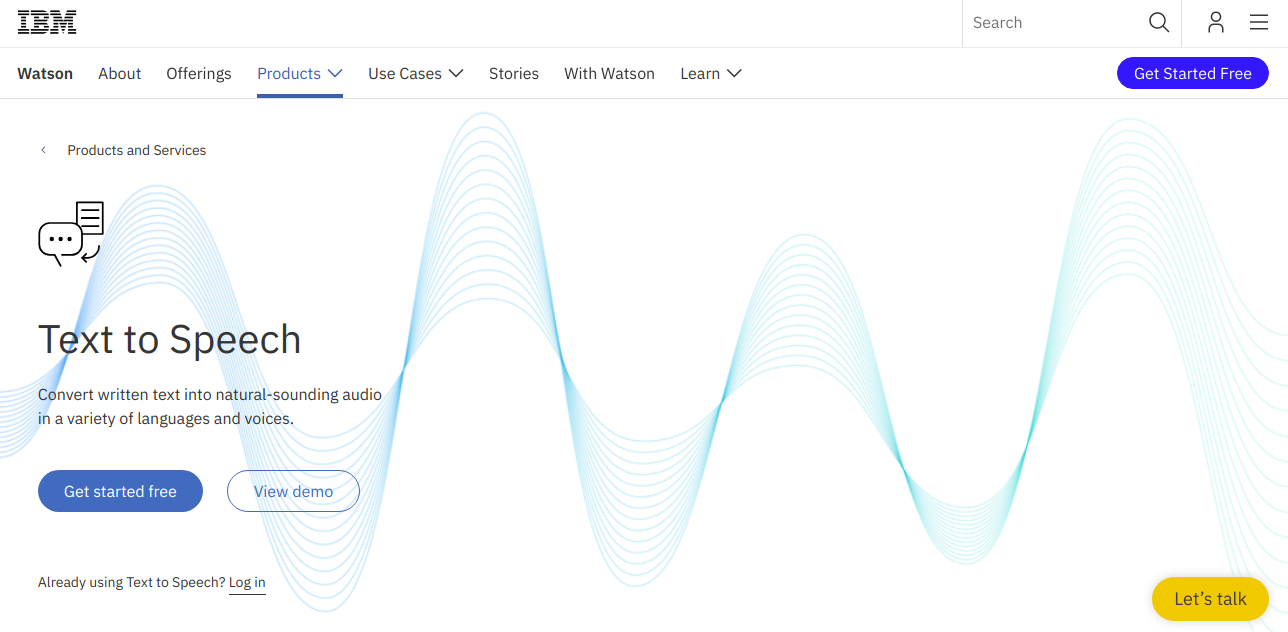


Рисунок 2 – IBM Watson

Однако проблема все так же, речь не похожа на человеческую.

Третьей альтернативой является Amazon Polly. SDK для самых популярных языков, несколько языков на выбор, настройка скорости речи и громкости. Сервис представлен на рисунке 3.

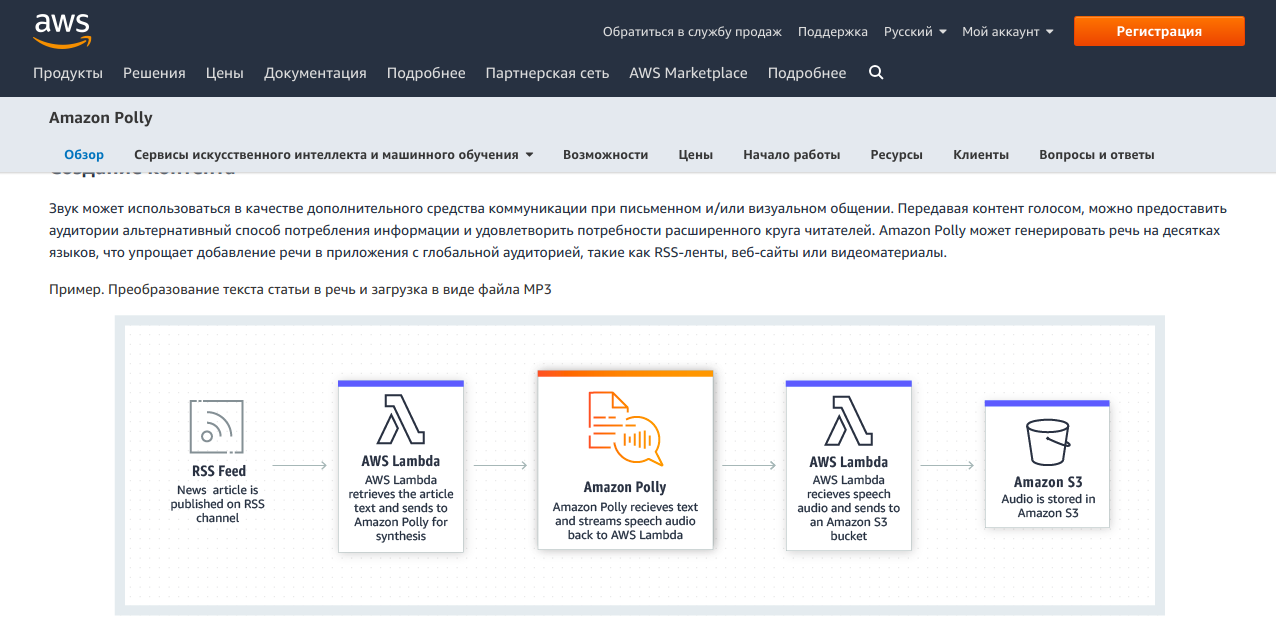


Рисунок 3 – Amazon Polly

Таблица 1 – Сравнение альтернативных решений

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Конвертация документов | RestAPI | Несколько голосов | Несколько языков | Клиентские SDK | Чистая речь |
| Проект | да | да | да | нет | нет | да |
| Google Text-to-Speech | да | да | да | да | да | нет |
| IBM-Watson | да | да | да | да | да | нет |
| Amazon Polly | да | да | да | да | да | нет |

В свою очередь представленный проект обладает всеми важными функциями, которые нужны большинству клиентов, за исключением SDK и поддержки только английского языка, которые можно использовать как в десктопной версии, так и в мобильной или интегрировать в свои приложения.

## Постановка задачи

Постановка задачи включает в себя следующие требования, которые необходимо реализовать в данном проекте:

* веб клиент;
* конвертация введенного текста через клиентское приложение;
* конвертация текстового документа через клиентское приложение;
* хранение ранее созданных звуковых дорожек;
* основные CRUD операции на ранее записанных дорожках;
* возможность скачивать и прослушивать созданные звуковые дорожки;
* возможность доступа к сервису по средствам RestAPI;
* посимвольный учет и оплата.
* возможность пробного конвертирования.

# Разработка проекта

## Функциональные возможности

Сценарий – это описание поведения системы, когда она взаимодействует с кем-то (или чем-то) из внешней среды. Система может отвечать на внешние запросы Актёра (actor) (может применяться термин Актант), может сама выступать инициатором взаимодействия. Другими словами, сценарий использования описывает, «кто» и «что» может сделать с рассматриваемой системой, или что система может сделать с «кем» или «чем». Методика сценариев использования применяется для выявления требований к поведению системы, известных также как пользовательские и функциональные требования. Сценарий нужен, чтобы сформулировать тот стержень, который будет в приложении. В системном проектировании сценарии использования применяются на более высоком уровне, чем при разработке программного обеспечения, часто представляя цели заинтересованных лиц или миссии.

Также при описании сценариев используется диаграмма вариантов использования или Use Case диаграмма.

Прецедент – возможность моделируемой системы (часть её функциональности), благодаря которой пользователь может получить конкретный, измеримый и нужный ему результат. Прецедент соответствует отдельному сервису системы, определяет один из вариантов её использования и описывает типичный способ взаимодействия пользователя с системой. Варианты использования обычно применяются для спецификации внешних требований к системе.

Отношение расширения определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства которого определяются на основе способа совместного объединения данных экземпляров. В метамодели отношение расширения является направленным и указывает, что применительно к отдельным примерам некоторого варианта использования должны быть выполнены конкретные условия, определенные для расширения данного варианта использования.

Исполнители, фигурирующие при работе с приложением:

* клиент;
* бэк-энд.
* подсистема конвертации.

Базовые прецеденты, представленные в системе (Рисунок 4):

* регистрация;
* вход в учетную запись;
* отправка запроса на конвертацию через форму;
* отправка Rest запроса с данными для конвертации;
* пополнения баланса;
* менеджмент ранее созданных голосовых записей.

После исследования предметной области были выявлены следующие роли: клиент, приложение; а также были выявлены следующие прецеденты: регистрация, вход в учетную запись, Отправка запроса на конвертацию через форму, Отправка Rest запроса с данными для конвертации, Пополнения баланса, менеджмент ранее созданных голосовых записей. На рисунке 4 представлена диаграмма вариантов использования приложения.

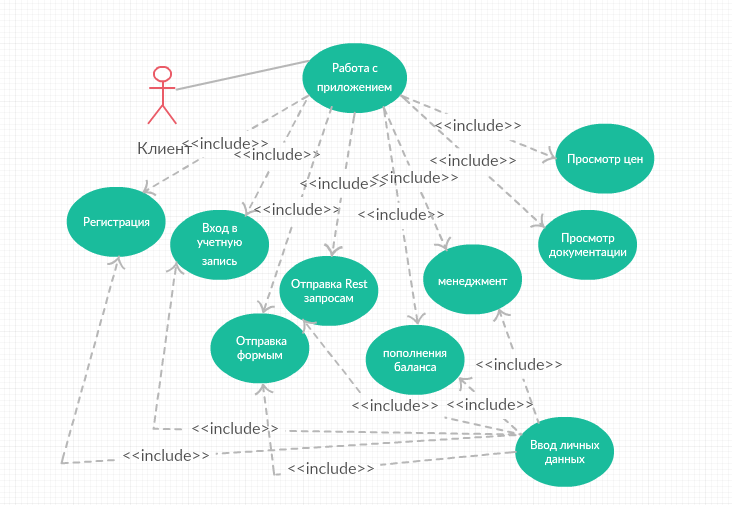


Рисунок 4 – Диаграмма вариантов использования

## Описание сценариев

Django предоставляет систему аутентификации и авторизации ("permission") пользователя, реализованную на основе фреймворка работы с сессиями. Система аутентификации и авторизации позволяет проверять полномочия пользователей и определять какие кому соответствуют. Данный фреймворк включает в себя встроенные модели для Пользователей и Групп (основной способ применения авторизации для более чем одного пользователя), непосредственно саму систему разграничения доступа (permissions)/флаги, которые определяют какую задачу, с какой формой и отображением, инструментом отображения, может работать тот, или иной пользователь.

В соответствии с идеологией Django система аутентификации является очень общей и, таким образом, не предоставляет некоторые возможности, которые присутствуют в других системах веб-аутентификации. Решениями некоторых общих задач занимаются пакеты сторонних разработчиков, например, защита от подбора пароля (через стороннюю библиотеку OAuth).

Система аутентификации является очень гибкой  и позволяет вам формировать свои собственные URL-адреса, формы, отображения, а также шаблоны страниц, если вы пожелаете, с нуля,  через простой вызов функций соответствующего API для логирования пользователя.

Для того что бы начать работу с приложением нужно зарегистрироваться с помощью клиентского приложения (рисунок 5).

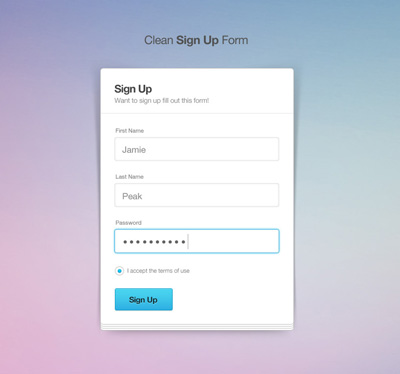


Рисунок 5 – Регистрация в приложении

Сценарий работы регистрации продемонстрирован диаграммой использования на рисунке 6.

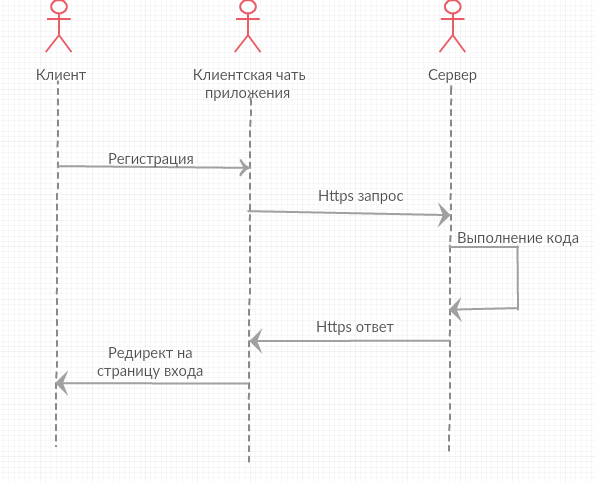


Рисунок 6 – Диаграмма регистрации

После регистрации нужно войти в учетную запись. Сценарий работы продемонстрирован диаграммой использования на рисунке 7.

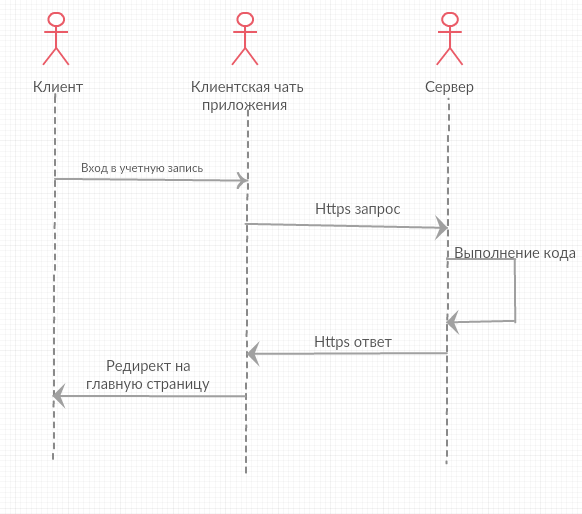


Рисунок 7 – Вход в учетную запись

После входа в учетную запись пользователь может отправить форму для конвертации текста. Для этого он может воспользоваться формой, размещенной на главную страницу. Сценарий работы данной команды продемонстрирован диаграммой использования на рисунке 8.

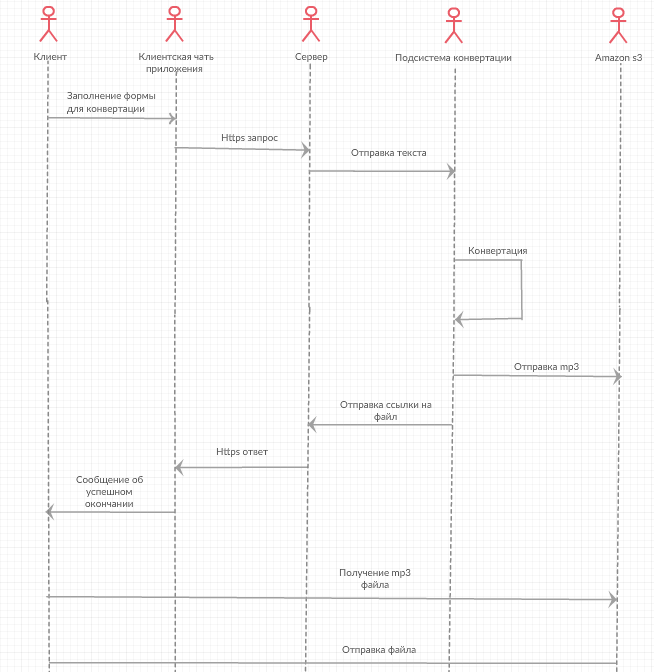


Рисунок 8 – Диаграмма отправки формы для конвертации

Отправка Rest запроса с данными для конвертации продемонстрирована на диаграмме 9.

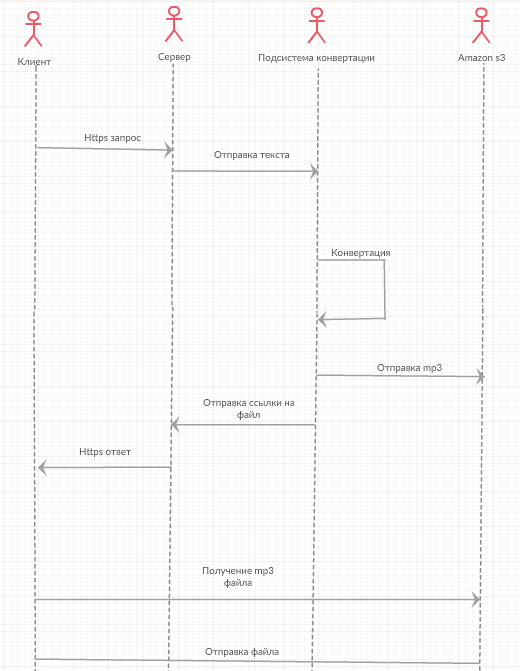


Рисунок 9 – Диаграмма отправки формы для конвертации

Пополнение баланса выполняется по средством платежной системы PayPal. Диаграмма пополнения баланса представлена на диаграмме 10.

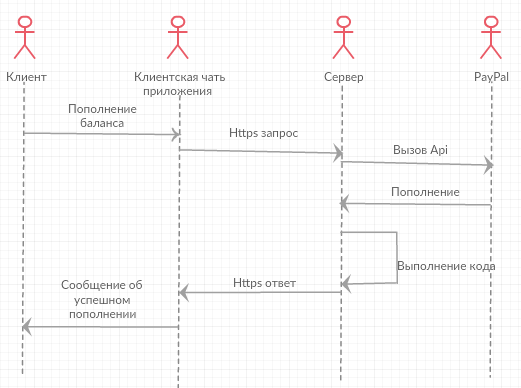


Рисунок 10 – Диаграмма пополнения баланса

Менеджмент ранее созданных голосовых записей, продиманстрирован на диаграмме 11.

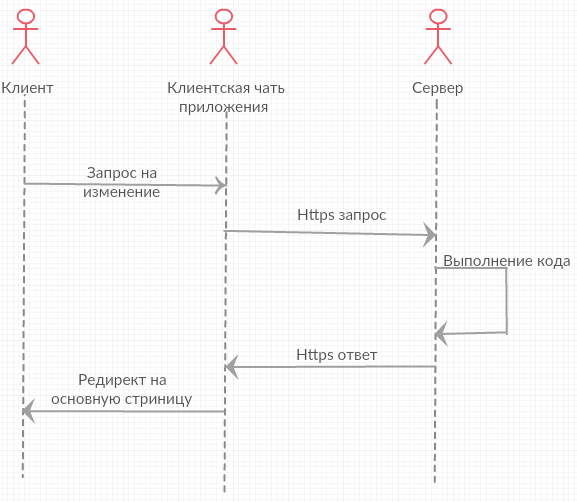


Рисунок 11 – Диаграмма пополнения баланса

## Информационно-логическая модель данных

Клиентская часть представляет собой веб сайт и позволяет выполнять конвертацию текста по средством заполнения формы. Которая отправляется на сервер, где текст с помощью брокера сообщений RabbitMQ попадает в отдельный процесс заранее созданный с помощью Сcelery в контексте которой работает подсистема конвертации.

Далее из подсистемы конвертации аудиофайл выгружается в облачное хранилище Amazon S3, а на сервер передается ссылка на него. Сервер в свою очередь передает клиенту ссылку на созданный аудиофайл.

Модель отвечающая за проводимые конвертации хранится в базе данных на сервере и предоставляет возможность скачивать, удалять, проигрывать ранее созданные аудиофайлы.

Модель представляет собой:

* Имя файла;
* Длительность;
* Дата создания;
* Пользователь;
* Ссылка на S3;

Модель пользователя в базе данных представленна в виде:

* Электронная почта;
* Уникальное имя;
* Баланс в символах;

## Обоснование выбора инструментария разработки

В качестве языка разработки данного программного обеспечения был выбран Python. Python – высокоуровневый язык программирования, ориентированный на повышение производительности разработчика и читаемости кода. Python – это язык программирования, который позволяет работать быстро и осуществлять интеграцию систем более эффективно. Так же данный язык программирования является кроссплатформенным, что позволяет не привязываться к какой-либо операционной системе для создания программного обеспечения.

Разработка языка Python была начата в конце [1980-х годов](https://ru.wikipedia.org/wiki/1980-е) сотрудником голландского института CWI [Гвидо ван Россумом](https://ru.wikipedia.org/wiki/Гвидо_ван_Россум). Для распределённой ОС [Amoeba](https://ru.wikipedia.org/wiki/Amoeba_(операционная_система)) требовался расширяемый [скриптовый язык](https://ru.wikipedia.org/wiki/Сценарный_язык), и Гвидо начал писать Python на досуге, позаимствовав некоторые наработки для языка [ABC](https://ru.wikipedia.org/wiki/ABC_(язык_программирования)) (Гвидо участвовал в разработке этого языка, ориентированного на обучение программированию). В феврале [1991 года](https://ru.wikipedia.org/wiki/1991_год) Гвидо опубликовал исходный текст в [группе новостей](https://ru.wikipedia.org/wiki/Группа_новостей) alt.sources. С самого начала Python проектировался как [объектно-ориентированный язык](https://ru.wikipedia.org/wiki/Объектно-ориентированный_язык_программирования).

Python поддерживает [динамическую типизацию](https://ru.wikipedia.org/wiki/Динамическая_типизация), то есть тип переменной определяется только во время исполнения. Поэтому вместо «присваивания значения переменной» лучше говорить о «связывании значения с некоторым именем». В Python имеются встроенные типы: [булевый](https://ru.wikipedia.org/wiki/Булевский_тип), [строка](https://ru.wikipedia.org/wiki/Строковый_тип), [Unicode](https://ru.wikipedia.org/wiki/Unicode)-строка, целое число произвольной точности, число [с плавающей запятой](https://ru.wikipedia.org/wiki/Плавающая_запятая), [комплексное число](https://ru.wikipedia.org/wiki/Комплексное_число) и некоторые другие. Из [коллекций](https://ru.wikipedia.org/wiki/Коллекция_(программирование)) в Python встроены: [список](https://ru.wikipedia.org/wiki/Список_(информатика)), [кортеж](https://ru.wikipedia.org/wiki/Кортеж_(информатика)) (неизменяемый список), [словарь](https://ru.wikipedia.org/wiki/Ассоциативный_массив), [множество](https://ru.wikipedia.org/wiki/Множество) и другие. Все значения являются объектами, в том числе функции, методы, модули, классы.

Добавить новый тип можно либо написав [класс](https://ru.wikipedia.org/wiki/Класс_(программирование)) (class), либо определив новый тип в модуле расширения (например, написанном на языке C). Система классов поддерживает [наследование](https://ru.wikipedia.org/wiki/Наследование_(программирование)) (одиночное и [множественное](https://ru.wikipedia.org/wiki/Множественное_наследование)) и [метапрограммирование](https://ru.wikipedia.org/wiki/Метапрограммирование). Возможно наследование от большинства встроенных типов и типов расширений.

Все объекты делятся на ссылочные и атомарные. К атомарным относятся int, long (в версии 3 любое число int, так как в версии 3 нет ограничения на размер), complex и некоторые другие. При присваивании атомарных объектов копируется их значение, в то время как для ссылочных копируется только указатель на объект, таким образом, обе переменные после присваивания используют одно и то же значение. Ссылочные объекты бывают изменяемые и неизменяемые. Например, строки и кортежи являются неизменяемыми, а списки, словари и многие другие объекты — изменяемыми. Кортеж в Python является, по сути, неизменяемым списком. Во многих случаях кортежи работают быстрее списков, поэтому если вы не планируете изменять последовательность, то лучше использовать именно их. На рисунке 12 показана структура типов языка Python.

Дизайн языка Python построен вокруг объектно-ориентированной модели программирования. Реализация [ООП](https://ru.wikipedia.org/wiki/Объектно-ориентированное_программирование) в Python является элегантной, мощной и хорошо продуманной, но вместе с тем достаточно специфической по сравнению с другими объектно-ориентированными языками.

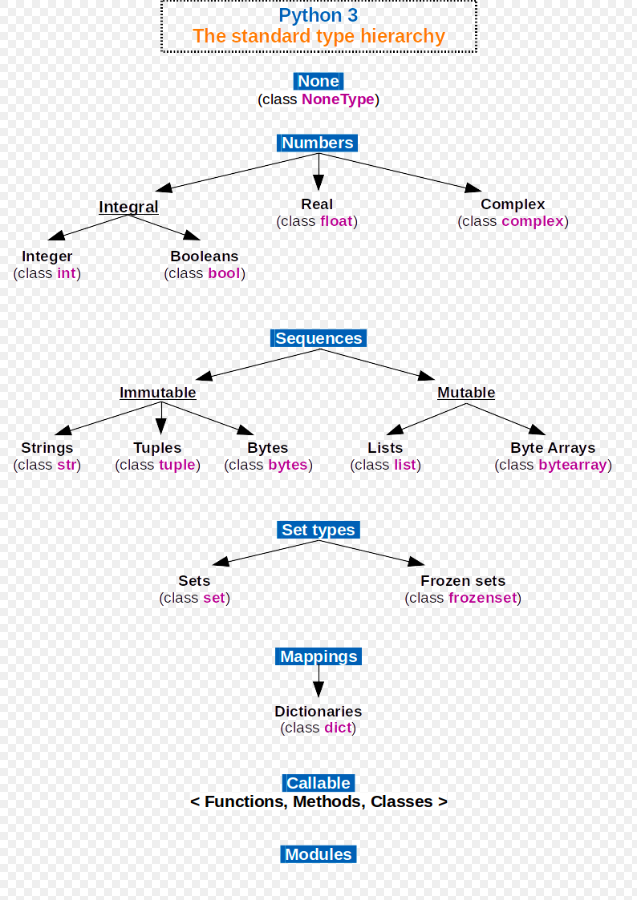


Рисунок 12 – Типы данных языка Python

В данном языке программирования представлены следующие возможности и особенности:

1. Классы являются одновременно объектами со всеми нижеприведёнными возможностями.
2. Наследование, в том числе множественное.
3. [Полиморфизм](https://ru.wikipedia.org/wiki/Полиморфизм_(информатика)) (все функции виртуальные).
4. [Инкапсуляция](https://ru.wikipedia.org/wiki/Инкапсуляция_(программирование)) (два уровня — общедоступные и скрытые методы и поля). Особенность — скрытые члены доступны для использования и помечены как скрытые лишь особыми именами.
5. Специальные методы, управляющие жизненным циклом объекта: конструкторы, деструкторы, распределители памяти.
6. Перегрузка операторов (всех, кроме is, '.', '=' и символьных логических).
7. Свойства (имитация поля с помощью функций).
8. Управление доступом к полям (эмуляция полей и методов, частичный доступ, и т. П.).
9. Методы для управления наиболее распространёнными операциями (истинностное значение, len(), глубокое копирование, [сериализация](https://ru.wikipedia.org/wiki/Сериализация), итерация по объекту, …).
10. Метапрограммирование (управление созданием классов, триггеры на создание классов, и др.).
11. Полная [интроспекция](https://ru.wikipedia.org/wiki/Интроспекция_(программирование)).
12. Классовые и статические методы, классов.

Так же при проектировании данного приложения были использованы фреймворки Django, Django Rest Framework, Tensorflow.

Django свободный фреймворк для веб-приложений на языке Python, использующий шаблон проектирования MVC. Проект поддерживается организацией Django Software Foundation. Сайт на Django строится из одного или нескольких приложений, которые рекомендуется делать отчуждаемыми и подключаемыми. Это одно из существенных архитектурных отличий этого фреймворка от некоторых других. Один из основных принципов фреймворка — DRY(англ. Don't repeat yourself). Также, в отличие от других фреймворков, обработчики URL в Django конфигурируются явно при помощи регулярных выражений. Для работы с базой данных Django использует собственный ORM, в котором модель данных описывается классами Python, и по ней генерируется схема базы данных.

Архитектура Django похожа на «Модель-Представление-Контроллер» (MVC). Контроллер классической модели MVC примерно соответствует уровню, который в Django называется Представление, а презентационная логика Представления реализуется в Django уровнем Шаблонов. Из-за этого уровневую архитектуру Django часто называют «Модель-Шаблон-Представление» (MTV). Первоначальная разработка Django как средства для работы новостных ресурсов достаточно сильно отразилась на его архитектуре: он предоставляет ряд средств, которые помогают в быстрой разработке веб-сайтов информационного характера. Так, например, разработчику не требуется создавать контроллеры и страницы для административной части сайта, в Django есть встроенное приложение для управления содержимым, которое можно включить в любой сайт, сделанный на Django, и которое может управлять сразу несколькими сайтами на одном сервере. Административное приложение позволяет создавать, изменять и удалять любые объекты наполнения сайта, протоколируя все совершённые действия, и предоставляет интерфейс для управления пользователями и группами (с пообъектным назначением прав).

В дистрибутив Django также включены приложения для системы комментариев, синдикации RSS и Atom, «статических страниц» (которыми можно управлять без необходимости писать контроллеры и представления), перенаправления URL и другое.

TensorFlow — открытая программная библиотека для машинного обучения, разработанная компанией Google для решения задач построения и тренировки нейронной сети с целью автоматического нахождения и классификации образов, достигая качества человеческого восприятия[4]. Применяется как для исследований, так и для разработки собственных продуктов Google.

## Архитектура проекта

Работа данного приложения будет заключаться в том, что пользователь отправляет форму или Rest HTTP запрос к серверу, который передает текст в подсистему конвертации и возврощает ссылку на файл в Amazon S3.

Архитектура данного приложения будет представлена сервером, и подсистемой конвертации запущенной в отдельном процессе. Сервер принимает сообщения от пользователя, пересылает текст с помощью RabbitMQ в очередь, которая считывается подсистемой конвертации.

Подсистемы в свою очередь загружает файл в облачное хранилище Amazon S3 и пересылает обратно на сервер ссылку на файл, а сервер в свою очередь эту ссылку пересылает клиенту. На рисунке 13 продемонстрирована схема взаимодействия сервера и подсистемы конвертации.

После запроса от пользователя вызывается функция synthesize(request). Она отвечает за прием формы от пользователя, функция в свою очередь вызывает функцию check\_balance(request.user, text) для проверки баланса. Далее если все хорошо вызывается channel.basic\_publish(text). Она помещает сообщение в очередь которую читает подсистема конвертации. А затем сама читает очередь с именем юзера для получения ссылки на аудио файл.

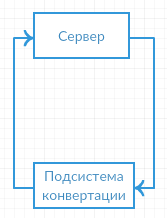


Рисунок 13 – Схема взаимодействия файлов сервера и подсистемы конвертации

В свою очередь подсистема читает очередь с помощью функции channel.basic\_consume(), передает сообщениы callback функции которая герерирует аудио и загружает в Amazon S3, а затем перадает в очередь сообщений уникальную для каждого юзера ссылку на файл.

# Реализация приложения

## Создание приложения

В качестве языка разработки данного программного обеспечения был выбран Python. Python – высокоуровневый язык программирования, ориентированный на повышение производительности разработчика и читаемости кода. Python – это язык программирования, который позволяет работать быстро и осуществлять интеграцию систем более эффективно. Так же данный язык программирования является кроссплатформенным, что позволяет не привязываться к какой-либо операционной системе для создания программного обеспечения. А также Django предоставляющий весь спектр необходимых инструментов. Сервер реализует регистрацию, вход в учетную запись, конвертацию, и менеджент ранее созданных аудиозаписей.

Для того чтобы создать Django проект в первую очередь нужно создать виртуальную среду. Для этого во время установки нужно использовать pipenv. Установим зависимости.

Django<2.1

psycopg<2.8

Далее создадим проект.

$ django-admin startproject config

В проекте используется PostgresDB, для настройки нужно изменить файл settings.py.

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'mymdb',

'USER': 'mymdb',

'PASSWORD': 'development',

'HOST': '127.0.0.1',

'PORT': '5432',

}

}

В то время как djnago можно создать и запустить миграцию для наших приложений в Django, он не может создавать сами базы данных и пользователе базы данных для нашего проекта django. Чтобы создать базу данных и пользователя, мы для подключения к серверу используя учетную запись администратора. Как только мы соединимся, мы сможем создать базу данных и пользователя, выполнив следующий SQL.

CREATE DATABASE voice;

CREATE USER voice\_user;

GRANT ALL ON DATABASE voice TO «voice\_user»;

ALTER USER voice\_user PASSWORD «DEVELOP»;

ALTER USER voice\_user CREATEDB;

Вышеуказанные операторы SQL создадут базу данных и пользователя для нашего проекта django. GRANT оператор гарантирует, что наш пользователь voice\_user будет иметь доступ к базе данных. Тогда мы установили пароль для пользователя voice\_user. Наконец, мы даем пользователю разрешение на создание новых баз данных, которые будут использоваться django для создания тестовой базы данных при выполнении тестов.

Далее нужно создать app. Нет ограничений на количество приложений, которые можно иметь в своем проекте Django. В идеале, каждое приложение должен иметь плотно ограниченную и автономную функциональность, как и любой другой модуль Python.

Нам понадобится 2 приложения users, в котором будет все что связано с пользователем, и core, где будет основная логика всего проекта. Для того чтобы создать app нужно воспользоватся manage.py.

$ python manage.py startapp core

$ python manage.py startapp users

Каждое приложение содержит в себе несколько файлов. \_\_init.py\_\_ говорит что это не только дериктория но еще и python модуль. Admin.py место где нужно регистрировать модели для встроеной админ бэкэнда. Apps.py место где можно запустить любой код при регистрации приложения, что полезно, если планируется создание многоразового приложения Django (например, пакет чтобы загрузить на PyPi). Папка mgirations, это модуль Python с миграциями баз данных. Миграции базы данных описывают, как перенести базу данных из одного состояния в другое. В Django, если вы добавляете модель, вы можете просто создать и запустить миграции используя manage.py. Models.py место для хранения моделй. Tests.py место для тестов. Views.py место для view. Теперь, когда основные приложения существует, нужно дать Django знать об этом, добавив их в список установленные приложения в файле settings.py. INSTALLED\_APPS это список путей Python к модулям Python, которые являются приложениями Django. Уже установлены приложения для решения распространенных проблем, таких как управление статическими файлами, сеансы и аутентификация и бэкэнд администратора. Нужно добавить core и users.

INSTALLED\_APPS = [

'users',

'core',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

## Реализация авторизации

Django предоставляет систему аутентификации и авторизации ("permission") пользователя, реализованную на основе фреймворка работы с сессиями. Система аутентификации и авторизации позволяет проверять полномочия пользователей и определять какие кому соответствуют. Данный фреймворк включает в себя встроенные модели для Пользователей и Групп(основной способ применения авторизации для более чем одного пользователя), непосредственно саму систему разграничения доступа (permissions)/флаги, которые определяют какую задачу, с какой формой и отображением, инструментом отображения, может работать тот, или иной пользователь. В соответствии с идеологией Django система аутентификации является очень общей и, таким образом, не предоставляет некоторые возможности, которые присутствуют в других системах веб-аутентификации. Решениями некоторых общих задач занимаются пакеты сторонних разработчиков, например, защита от подбора пароля (через стороннюю библиотеку Oauth).

Для начала нужно создать кастомную модель пользователя в котором будет содержатся поле баланса. Модель Django - это класс, производный от Model и имеет один или несколько Field. В терминах баз данных, класс Model соответствует таблице базы данных, а классы Field соответствуют столбцам и экземплярам. Используя ORM Django можно написать один класс вместо подхода на прямую использующего базу данных пишущих модели один раз в Python и снова в SQL. Ниже представлена модель кастомного пользователя.

class CustomUser(AbstractUser):

balance = models.BigIntegerField(blank=True, default=default\_balance)

# а также другие изменения необходимые для использования email вместо uesrname

Теперь, когда у нас есть модель, нам нужно будет создать таблицу в нашей базе данных, которая соответствует ей. Мы будем использовать Django для создания миграции для нас, а затем запустить миграцию для создания таблицы для нашей модели.

Необходимые настройки были выполнены для нас, когда мы создали приложение при помощи команды django-admin startproject. Таблицы базы данных для пользователей и модели авторизации были созданы, когда в первый раз выполнили команду python manage.py migrate. Соответствующие настройки сделаны в параметрах INSTALLED\_APPS и MIDDLEWARE файла проекта settings.py.

INSTALLED\_APPS = [

'django.contrib.auth',

# Фреймворк аутентификации и моделей по умолчанию.

'django.contrib.contenttypes',

# Django контент-типовая система (даёт разрешения, связанные с моделями).

]

MIDDLEWARE = [

'django.contrib.sessions.middleware.SessionMiddleware',

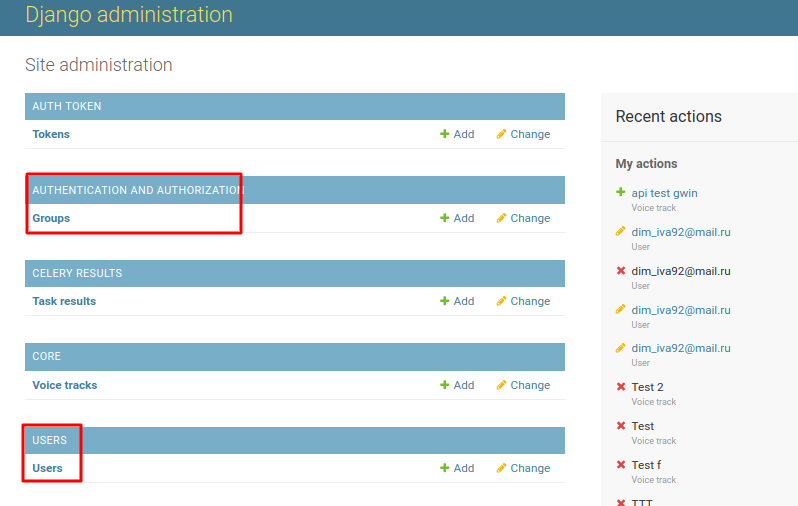
# Управление сессиями между запросами

'django.contrib.auth.middleware.AuthenticationMiddleware',

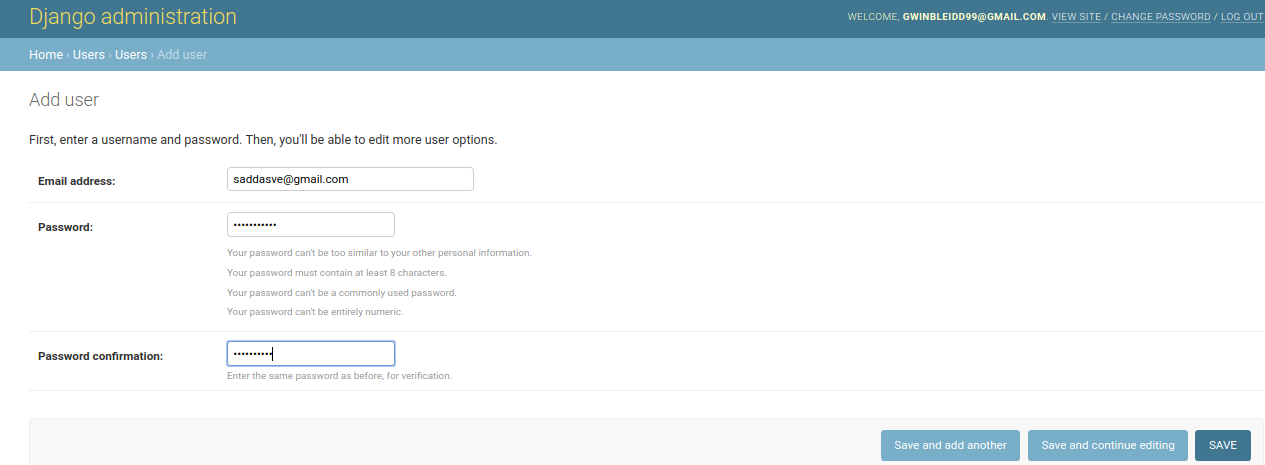
# Связывает пользователей, использующих сессии, запросами.

]

В админской панели появятся ссылки создания групп и пользователей. Залогиньтесь на сайте при помощи параметров (имя пользователя и пароля) аккаунта суперпользователя. Для того, чтобы увидеть записи в разделе Authentication and Authorisation вы можете нажать на ссылку Users, или Groups.



Для коздания пользователя нужно перейти по вкладке Users. Административная часть сайта создаст нового пользователя и немедленно перенаправит вас на страницу Change user (Изменение параметров пользователя) где вы можете, соответственно, изменить email, а кроме того добавить информацию для дополнительных полей модели User. Эти поля включают в себя имя пользователя, фамилию, адрес электронной почты, статус пользователя, а также соответствующие параметры доступа (может быть установлен только флаг  Active). Ниже вы можно определить группу для пользователя и необходимые параметры доступа, а кроме того, увидеть важные даты, относящиеся к пользователю (дату подключения к сайту и дату последнего входа/логирования).



Django предоставляет почти все, что нужно для создания страниц аутентификации входа, выхода из системы и управления паролями из коробки. Это включает в себя url-адреса, представления (views) и формы,но не включает шаблоны. Для настройки поддтверждения регистрации по электронной почты и смены пароля нежно настроить почтовый сервер.

EMAIL\_USE\_TLS = True

EMAIL\_HOST = 'smtp.gmail.com'

EMAIL\_HOST\_USER = 'gwinbleidd99@gmail.com'

EMAIL\_HOST\_PASSWORD = '\*\*\*\*\*\*\*\*\*\*\*\*'

EMAIL\_PORT = 587

Далее нужен токенгенератор для генерации уникальных url. Который будет использоватся для подтверждения электронной почты.

class TokenGenerator(PasswordResetTokenGenerator):

def \_make\_hash\_value(self, user, timestamp):

# Вернуть уникальный токен

Для работы с пользователем нужно создать django фрому пользователя.

class CustomUserCreationForm(UserCreationForm):

class Meta(UserCreationForm):

# Указание модели

# И полей используемых в форме

Для обработки регистрации нужно создать view который будет вызыватся по запросу.

def signup(request):

# Проверка на тип запроса если psot

# Создание формы

# Если форма валидна

# Создание пользователя со статсом неактивный

# И отправка сообщения со ссылкой на почтовый ящик указанный при реистрации

else:

# Создание формы

# Редирект на страницу регистрации

Где при регистрации пользователю присваевается статус user.is\_active = False. Так что он не может войти без подтверждения. Для этого на элестронный адрес пользователя отправляется сообщение

Hi {{ user.username }},

Please click on the link to confirm your registration,

http://{{ domain }}{% url 'users:activate' uidb64=uid token=token %}

Пользователю передана ссылка активации перейдя по которой он вызовет функцию активации которая присвоит ему статус active.

def activate(request, uidb64, token):

# попытка получения пользователя по pk в запросе

# проверка токена если он правильный

# присваение статуса активный

else:

# редирект на страницу регистрации с сообщением об ошибке.

Далее нужно добавить url паттерны в файл users/urls.py.

urlpatterns = [

url(r'^signup/$', users.views.signup, name='signup'),

path('login/', LoginView.as\_view(), name='login'),

path('logout/', LogoutView.as\_view(), name='logout'),

url(r'^activate/(?P<uidb64>[0-9A-Za-z\_\-]+)/(?P<token>[0-9A-Za-z]{1,13}-[0-9A-Za-z]{1,20})/$',users.views.activate, name='activate'),]

URL-адреса и неявные представления ожидают найти связанные с ними шаблоны в каталоге регистрации где-то в пути поиска шаблонов. Для этого сайта мы разместим наши HTML-страницы в каталоге templates / registration *.Этот каталог должен находиться в корневом каталоге проекта, то есть в том же каталоге.* Чтобы сделать эти директории видимыми для загрузчика шаблонов, помещать этот каталог в путь поиска шаблона нужно обнавить настройки проекта settings.py в секции TEMPLATES строку 'DIRS'.

TEMPLATES = [

{

...

'DIRS': [os.path.join(BASE\_DIR, ''), os.path.join(BASE\_DIR, 'templates')],

'APP\_DIRS': True,

...

},

]

Сам шаблон он расширяет наш базовый шаблон и переопределяет блок контента. Шаблон предостовляет показ формы, в которой вы можете ввести электронный адрес и пароль, а если вы введете недопустимые значения, вам будет предложено ввести правильные значения, когда страница обновится.

<form action="" method="post">

# токен защиты от csrf

# поля формы для входа в систему

# кнопка подтверждения формы

</form>

Если ваша попытка войти в систему будет успешной,  вы будете перенаправлены на главную страницу. Django ожидает, что после входа в систему вы захотите перейти на страницу профиля, а поскольку эта страница не определена будет ошибка. Для этого ннжно переопределить значение переменной LOGIN\_REDIRECT\_URL в файле настроек settings.py.

Для регистрации создан шаблон регистрации функционал которого похож на шаблон входа в систему (отображение формы).

<form action="" method="post">

# токен защиты от csrf

# поля формы для регистрации

# кнопка подтверждения формы

</form>

Система сброса пароля по умолчанию использует электронную почту, чтобы отправить пользователю ссылку на сброс. Необходимо создать формы, чтобы получить адрес электронной почты пользователя, отправить электронное письмо, разрешить им вводить новый пароль и отметить, когда весь процесс будет завершен. Для этого нужно добавить в патерны url несколько значений.

urlpatterns=[

url(r'password\_change/'),

url(r'password\_change\_done/'),

url(r'password\_reset/$'),

url(r'password\_reset\_done/'),

url(r'password\_reset\_confirm/(?P<uidb64>[0-9A-Za-z\_\-]+)/(?P<token>[0-9A-Za-z]{1,13}-[0-9A-Za-z]{1,20})/$'),

url(r'password\_reset\_complete/'),

]

#### Форма сброса пароля используемая для получения адреса электронной почты пользователя, для отправки уникального url для сброса пароля.

<form method="post">

# токен защиты от csrf

# поля формы для ввода электронной почты

# кнопка подтверждения формы

</form>

После чего на почту будет выслано письмо с уникальным url.

To initiate the password reset process for your {{ user.get\_username }} TestSite Account,

click the link below:

{{ protocol }}://{{ domain }}{% url 'users:password\_reset\_confirm' uidb64=uid token=token %}

#### На странице подтверждение сбросса пароля вводится новый пароль после нажатия ссылки в электронном письме с возвратом пароля. После которого происходит редирект на главную страницу.

## Реализация серверной части

Как в случае с польователем нам понадобится модель для представления запроса на конвертацию. Модель представлена ниже.

class VoiceTrack(models.Model):

id = models.UUIDField()

created = models.DateTimeField()

text = models.CharField()

owner = models.ForeignKey(to=settings.AUTH\_USER\_MODEL,)

audio = models.FileField()

duration = models.IntegerField()

pitch = models.SmallIntegerField()

speed = models.FloatField()

language = models.CharField()

voice = models.CharField()

Где id представлена в виде UUID поля, присутствует поле для даты с автозаполнением, сам текст запроса, внешний ключ владелеца, поле для хранения звукового файла, длительность, и настройки такие как, тон, скорость, язык и голос. Также нужно добавить модель в админ бэкэнд. Для этого нужно изменить admin.py и зарегистрировать в нем нашу модель.

admin.site.register(VoiceTrack)

На гравной странице присутствует форма для конвертации текста, а также таблица со всеми предыдущими дорожками. Для отображения главной страницы в core/urls.py нужно добавить url паттерн для главной страницы.

path('', views.MainPage.as\_view(), name='MainPage'),

А также создать CBV (class based view) для обработки запроса.

class MainPage(TemplateView):

# имя шаблона

def get\_context\_data(self, \*\*kwargs):

# добавления контекста

# в виде баланса пользователя

# и предыдущих звуковых дорожек пользователя

Заполнив форму и настроив параметры и зажав на кнопку Speek, клиент посылает форму на сервер по средством Http запроса. На стороне сервера вызывается view создния формы. Для этого ну ниже будет приведен CBV отвечающей на запрос. Также нужно добавить url в core/urls.py.

path('create\_voice/', views.CreateMessageView.as\_view(), name='CreateVoice'),

class CreateMessageView(LoginRequiredMixin, UserBalanceTextMixin, CreateView):

# указание формы модели

def get\_success\_url(self):

# редирект на главную страницу

def get\_initial(self):

# получение инициализационных данных

Который проверяет с помощью mixins авторизован ли пользователь и есть ли у него достаточно баланса.

class UserBalanceTextMixin(object):

def form\_valid(self, form):

# Проверка достаточно ли баланса у польователя

else:

# Вызов исключения отказано в доступе

Также вызывавет создание модели и в момент создания помещение текста в очередь на конвертацию.

В клиенте отображаются ранее созданные аудиозаписи в виде таблици с элементами управления, такими как прослушать, удалить, скачать. Для прослушивания аудио использовалась JavaScript блиотека wavesurfer, позволяющая отрисовывать спектограмму аудио для большей наглядности (рисунок 16).

Для удаления аудио нужно нажать на крес в таблице предыдущих аудио. Который отправит Http запрос на сервер на который будет обрабатыватся CBV. В котором будет произведена проверка на авторство, чтобы только создатель мог удалить запись. После чего будет удалена запись которая соответствует pk в запросе.

class DeleteMassageView(LoginRequiredMixin, DeleteView):

# Указание модели

def get\_success\_url(self):

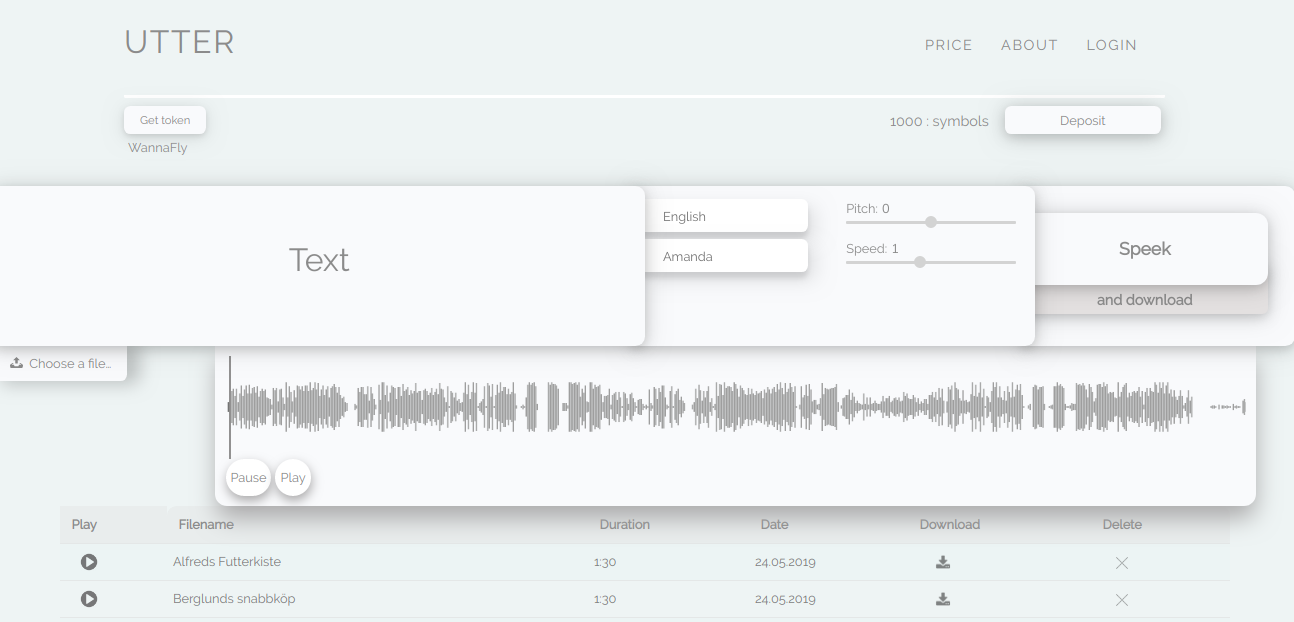
# Редирект на главную страницу

Также нужно добавить url в core/urls.py.

# Delete link

path('<uuid:pk>/delete/',views.DeleteMassageView.as\_view(), name='delete\_track'),

В клиенте отображаются ранее созданные аудиозаписи в виде таблици с элементами управления, такими как прослушать, удалить, скачать. Для прослушивания аудио использовалась JavaScript блиотека wavesurfer, позволяющая отрисовывать спектограмму аудио для большей наглядности (рисунок 16).

Рисунок 16 – Спектограмма при проигрывании аудио

Реализация очереди на конвертацию

Она представляет из себя асинхронные процессы запускаемые на старте приложения которые получают сообщения из очереди для балансирования нагрузки. В качестве асинхронных задач используется Celery. Celery - это асинхронная очередь задач на основе распределенной передачи сообщений. Он ориентирован на работу в режиме реального времени, но также поддерживает планирование. Задачи могут запускаться как сразу так и периодически.

Для начала нужно установить Celery в виртуальную среду.

Pipenv install celery==3.1.18

А также RabbitMQ ктоорый будет выступать в качестве «брокера» сообщений.

apt-get install -y erlang

apt-get install rabbitmq-server

Затем нужно запустить сервер RabbitMQ.

systemctl enable rabbitmq-server

systemctl start rabbitmq-server

Для начала нужно добавить url «брокера» сообщений в settings.py.

CELERY\_BROKER\_URL **=** 'amqp://localhost'

Затем нужно добавить файл celery.py, в котором созданно и настроенно приложение celery. Celery использует «брокер» для передачи сообщений между django поректом и celery «воркером». Наконец можно создать celery задачу в файле task.py.

@task()

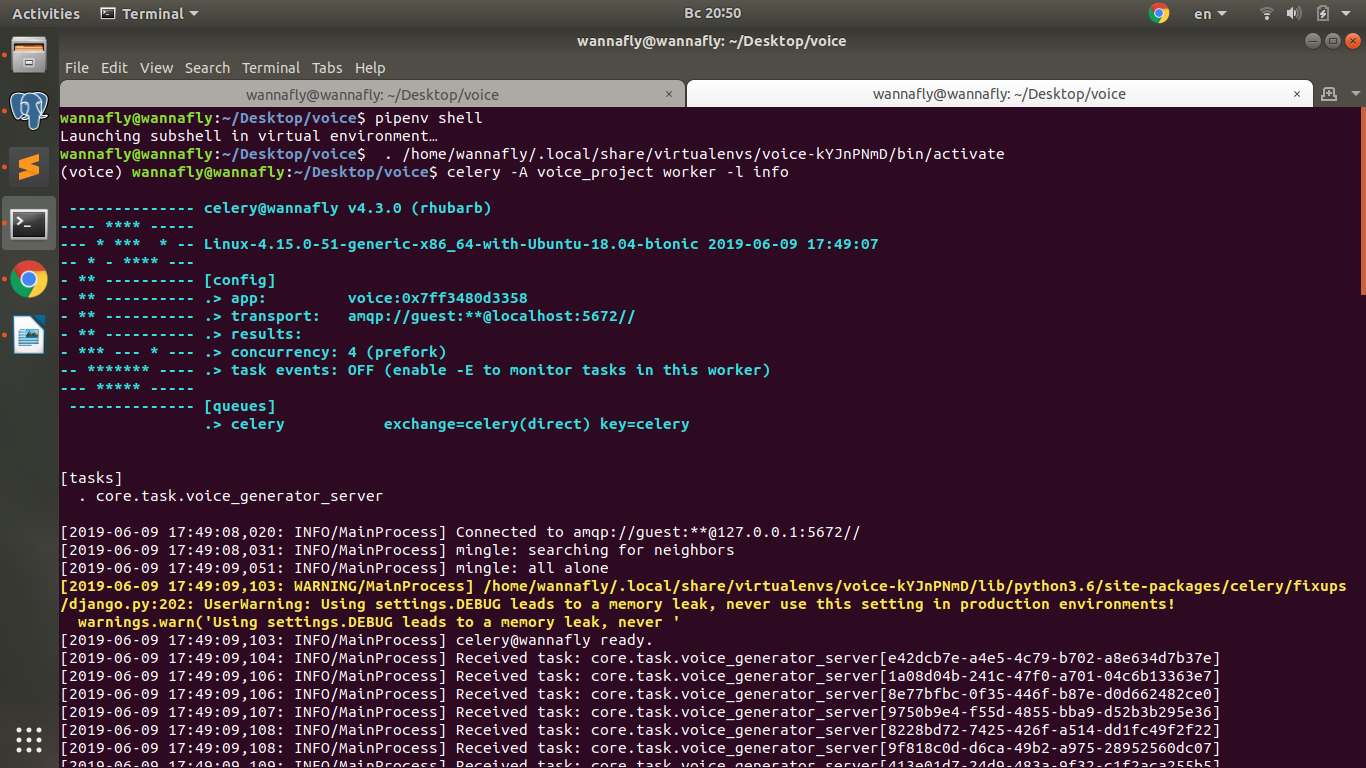
def voice\_generator\_server():

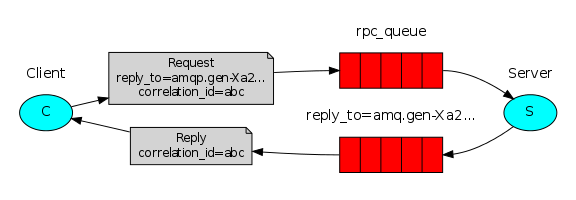
# Загрузка модели

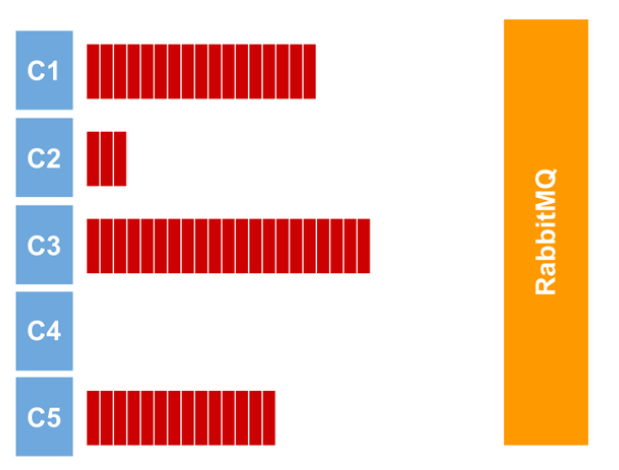
# Инициализация RPC сервера

Затем в apps.py в классе CoreConfig в методе ready(), мы вызываем задачу voice\_generator\_server.delay(). Которая запустится как только django пирложение закончит инициализацию. При запуске celery воркера в командной строке через celery --app= просто создается процесс. Воркер сам не обрабатывает никакие задачи. Он порождает дочерние процессы (или потоки). Дочерние процессы (или потоки) выполняют фактические задачи. Эти дочерние процессы (или потоки) также известны как пул выполнения. Размер пула выполнения определяет количество задач, которые может обрабатывать воркер. Чем больше процессов (или потоков) порождает воркер, тем больше задач он может обрабатывать одновременно. Если нужно обработать как можно больше задач как можно быстрее, нужен больший пул выполнения. В терминале, в новом окне нужно ввести команду чтобы запустить воркер результат на рисунке #########.

celery -A voice\_project worker -l info

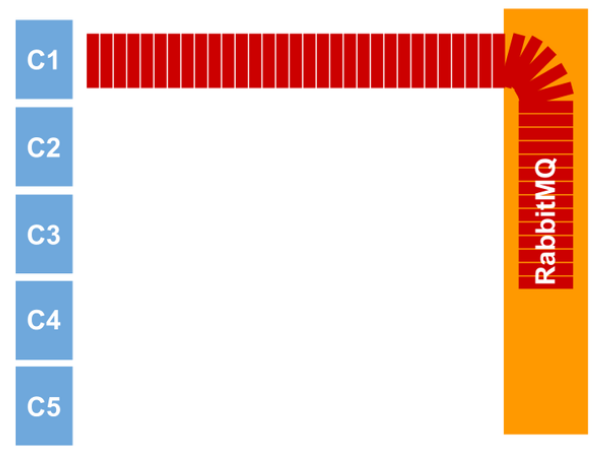
Для передачи сообщений в очередь задач и обратно нужно построить на основе RabbitMQ RPC систему, которая будет включать клиент и масштабируемый RPC сервер. Идея вызова удалённых процедур состоит в расширении хорошо известного и понятного механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. Средства удалённого вызова процедур предназначены для облегчения организации распределённых вычислений и создания распределенных клиент-серверных информационных систем. Наибольшая эффективность использования RPC достигается в тех приложениях, в которых существует интерактивная связь между удалёнными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Для использования RPC службы, создадим клиентский класс VoiceGeneratorRpcClient. Этот класс будет содержать метод *call*, который будет отправлять RPC запросы и блокироваться до получения ответа. Совершать RPC через RabbitMQ легко. Клиент отправляет запрос и сервер отвечает на запрос. Чтобы получить ответ, клиент должен передать очередь для размещения результатов вместе с запросом.  
Получив ответ из очереди не совсем ясно, какому запросу соответствует этот ответ. Тут нам пригодится свойство *correlation\_id*. Мы будем присваивать этому свойству уникальное значение при каждом запросе. Позднее, когда мы извлечем полученный ответ из очереди ответов, основываясь на значении этого свойства мы сможем однозначно сопоставить запрос с ответом. Если встретим неизвестное значение в свойстве *correlation\_id*, мы можем спокойно игнорировать это сообщение, так как оно не соответствует ни одному из наших запросов схема представлена на рисунке #########.

Наш RPC будет работать следующим образом:  
Когда Клиент стартует, он создает анонимную уникальную очередь результатов. Для совершения RPC запроса, Клиент отправляет сообщение с двумя свойствами: reply\_to, где в качестве значения указывается очередь результатов и correlation\_id, устанавливаемый в уникальное значение для каждого. Запрос отправляется в очередь rpc\_queue. Сервер ожидает запросы из этой очереди. Когда запрос получен, Сервер выполняет свою задачу и отправляет сообщение с результатом обратно Клиенту, используя очередь из свойства reply\_to. Клиент ожидает результат из очереди результатов. Когда сообщение получено, Клиент проверяет свойство correlation\_id. Если оно соответствует значение из запроса, то результат отправляется приложению. Также важно установить правильный пораметр prefetch\_count.

Рисунок ­ Схема работы консумеров с задачами при prefetch\_count = 0.

Задачи всегда будут отличаться по трудоемкости, хотя бы незначительно. Поэтому даже если они равны (очень близки), то картина будет приблизительно такой. Разница в количестве неподтвержденных сообщений у консумеров будет расти, если не все из них успевают обрабатывать свои сообщения. Если задачи могут значительно отличаться по трудоемкости, то при получении сообщения, обработка которого займет много времени, будет накапливаться количество неподтвержденных сообщений. Визуально кажется, что сообщения зависают, т.к. свободные консумеры обрабатывают новые сообщения мгновенно, а у занятого сообщения накапливаются. Эти сообщения не будут отданы на обработку другим консумерам.

Рисунок ­ После рестарта консумеров при prefetch\_count = 0.



Интереснее дела обстоят, когда в очереди есть сообщения, и происходит перезапуск консумеров (или просто первый запуск). Консумеры стартуют вместе, но все-равно с минимальным временным интервалом. И поэтому как только первый запускается, он сразу получает пачку сообщений (т.к. на этот момент других консумеров еще нет).

Рисунок ­ Схема работы консумеров при prefetch\_count = 1.

При использовании опции **prefetchCount = 1** консумер не получает следующее сообщение, пока не подтвердит предыдущее. Таким образом можно получить равномерную загруженность, не зависимо от равномерности трудоемкости задач. Не возникнет ситуации, когда в очереди есть сообщения, а какие-то консумеры простаивают. Псевдокод сервера представлен ниже.

# Устанавливаем соединение и объявляем очередь;

connection = pika.BlockingConnection(pika.ConnectionParameters(

host='localhost'))

# Обьявляем коллбэк функцию в которой будет происходить конвертация с помощью модели.

def on\_request(ch, method, props, body):

# Начинаем получать сообщения и устанавливаем prefetch\_count=1 в случае запуска более 1 сервера

channel.basic\_qos(prefetch\_count=1)

channel.basic\_consume(on\_message\_callback = on\_request, queue='rpc\_queue')

channel.start\_consuming()

Псевдокод клиента VoiceGeneratorRpcClient.

# client for rpc RabbitMQ

class VoiceGeneratorRpcClient(object):

def \_\_init\_\_(self):

# Устанавливаем соединение, канал и объявляем уникальную очередь результатов для полученных ответов;

self.connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))

self.channel = self.connection.channel()

result = self.channel.queue\_declare('', exclusive=True)

self.callback\_queue = result.method.queue

# Мы подписываемся на очередь результатов для получения ответов от RPC;

self.channel.basic\_consume()

def on\_response(self, ch, method, props, body):

# Функция обратного вызова 'on\_response', исполнямая при получении каждого ответа, выполняет тривиальную задачу — для каждого поступившего ответа она проверяет соответствует ли correlation\_id тому что мы ожидаем. Если это так, она сохраняет ответ в self.response и прерывает цикл;

def call(self, n):

# метод call, который и выполняет RPC запрос;

# В этом методе мы сначала генерируем уникальный correlation\_id и сохраняем его — функция обратного вызова 'on\_response' будет использовать это значение для отслеживания нужного ответа;

# помещаем запрос со свойствами reply\_to и correlation\_id в очередь; начинается процесс ожидания ответа; возвращаем результат обратно пользователю.

## Реализация RestAPI

Для начала нужно установить djangorestframework, для упращения создания API.

$ pipenv install djangorestframework

После чего нужно добавить его в INSTALLED\_APPS.

INSTALLED\_APPS = [

„rest\_framework“,

]

DRF предоставляет богатую коллекцию view которые расширяют APIView(который в свою очередь расширяет Django View класс). Цикл жизни DRF предстовляет собой следующие действия. Оборачивает запрос django в DRF запрос: DRF имеет специализированный класс Request, который обертывает класс django Request. Находит анализатор запросов и средство визуализации ответов. Проверка подлинности: проверяет учетные данные, связанные с запросом. Проверка разрешений: проверяет, может ли пользователь, связанный с запросом иметь доступ к этому представлению. Проверка тротлингов: проверяет, не было ли слишком много запросов в последнее время этим пользователем. Выполняет обработчик представления: выполняет действие, связанное с представлением (для например, создание ресурса, запрос к базе данных и т. д.). Обработка ответа: отображает ответ. Большинство предыдущих шагов выполняются с использованием настраиваемых классов. DRF настраивается создав словарь в settings.py нашего проекта под названием REST\_FRAMEWORK.

* DEFAULT\_PARSER\_CLASSES: Поддерживает JSON, формы и составные формы по умолчанию. Другие синтаксические анализаторы (например, YAML и MessageBuffer) доступны как пакеты сторонних сообществ.
* DEFAULT\_AUTHENTICATION\_CLASSES: Поддерживает сеанс на основе проверки подлинности и обычная проверка подлинности по протоколу HTTP по умолчанию. Проверка подлинности сеанса может упростить использование API в интерфейсе приложения. DRF поставляется с токеном класс аутентификации. Поддержка OAuth (1 и 2) доступна через пакеты сообщества.
* DEFAULT\_PERMISSION\_CLASSES: По умолчанию позволяет любому пользователю любые действий (включая операции обновления и удаления). DRF постовляется с коллекцией более строгих разрешений.
* DEFAULT\_TROTTLE\_CLASSES: Пусто по умолчанию. DRF предлагает простую схему для троттлинга, позволяя нам установить различные тарифы для анонимных запросов и запросов пользователей из коробки.
* DEFAULT\_RENDERER\_CLASSES: По умолчанию используется JSON и просматриваемый шаблон. Просматриваемый шаблон визуализации делает простой пользовательский интерфейс для просмотра и проверки.

Обновим settings.py.

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': (

'rest\_framework.authentication.TokenAuthentication',

'rest\_framework.authentication.SessionAuthentication',

),

'DEFAULT\_THROTTLE\_CLASSES': (

'rest\_framework.throttling.UserRateThrottle',

'rest\_framework.throttling.AnonRateThrottle',

),

'DEFAULT\_THROTTLE\_RATES': {

'user': '60000/min',

'anon': '30/min',

},

}

Когда анализатор DRF анализирует тело запроса, анализатор в основном возвращает Python словарь. Однако, прежде чем мы сможем выполнить какую-либо операцию с этими данными, нам необходимо убедиться, что данные действительны. В Django используется форма, а в DRF сериализаторы. DRF сериализатор очень похож на форму Django. Оба участвуют в получение данных проверки и подготовка моделей для вывода. Однако класс сериализатора не знает, как отображать свои данные, в отличие от формы Django. В представление DRF, визуализаторы отвечают за рендеринг результата в JSON. Подобно форме Django, сериализатор может быть создан для работы с произвольными данными или быть на основе модели Django. Кроме того, сериализатор состоит из коллекции полей, которые можно использовать для управления сериализацией. Когда сериализатор связан с моделью, DRF знает какое поле для чего использовать. Нужно создать сериализатор для треков.

class VoiceTrackSerializer(serializers.HyperlinkedModelSerializer):

# Пользователь который владеет треком

owner = serializers.PrimaryKeyRelatedField(queryset=get\_user\_model().objects.all())

class Meta:

# указание модели

# указание полей

}

Также нужно создать разрешения которые будут использоватся в представления для выявляния имеет ли пользователь доступ к запрашиваемому ресурсу. Первое разрешение проверяет имеет ли пользователь достаточно баланса для конвертации.

class UserBalanceTextPermission(BasePermission):

# сообщение об ошибке

def has\_permission(self, request, view):

# проверка баланса пользователя

Второе разрешение это проверка на обладание треком для того чтобы его мог удалить только пользователь создавший его.

class UserCanUseVoiceTrack(BasePermission):

# сообщение об ошибке

def has\_object\_permission(self, request, view, obj):

# Проверка является ли пользователь обладателем запрашиваемого обьекта.

Далее нужно создать сами представления. DRF представляет несколько CBV из коробки. Основное различие между DRF CBV и django CBV в том как они совмещают несколько операций. Для примера DRF предоставляет класс ListCreateAPIView, но django предоставляет ListView и CreateView отдельно. Так как ресурс api/v1/example будет предоставлять она и создание и получение списка.

RestAPI проекта будет предоставлять создание, чтение, изменение, и удаление обьектов VoiceTrack. Для создания VoiceTrack нужно использовать generics.CreateAPIView в качестве класса родителя. Псевдокод представлен ниже.

class VoiceTrackCreateView(generics.CreateAPIView):

# Обьявление классов разрешений

# Обьявление сериализатора

def get\_serializer(self, \*args, \*\*kwargs):

# инициализация владельца из запроса

return super().get\_serializer(\*args, \*\*kwargs)

Далее нужно добавить url паттерн в urls.py.

path('api/voice/create/', views.VoiceTrackCreateView.as\_view(), name='create\_track'),

Для операций удаления, обновления и представления VoiceTrack, создадим класс наследующийся от APIView и переопределим методы delete(), get() и retrive().

class VoiceTrackDetail(APIView):

# Обьявление разрешений

def get\_object(self, pk):

# получения обьекта по pk из запроса

def get(self, request, pk, format=None):

# возврещение обьекта в JSON формате

def delete(self, request, pk, format=None):

# удаление обьекта

def retrive(self, request, pk, format=None):

# изменение обьекта

Также нужно добавить url паттерн в urls.py.

path('api/voice/<uuid:pk>/', views.VoiceTrackDetail.as\_view(), name='detail\_track'),

Для предоставление всех созданные ранее обьектов VoiceTrack, создадим класс наследующийся от generics.ListCreateAPIView.

class VoiceTrackList(generics.ListCreateAPIView):

# обьявление разрешений

# обьявление сериализатора

def get\_queryset(self):

# получение обьектов

def list(self, request):

# возвращение обьектов в формате JSON

Также нужно добавить url паттерн в urls.py.

path('api/voice/list/', views.VoiceTrackList.as\_view(), name='list\_track'),

## Реализация пополнения баланса

Для начала нужно установить приложение django-paypal.

$ pipenv install django-paypal.

Обновить список приложений в settigns.py.

INSTALLED\_APPS = [

#...

'paypal.standard.ipn',

#...

]

Создайть экземпляр формы PayPalPaymentsForm в представлении. Необходимо заполнить словарь информацией, необходимой для завершения платежа, и передать ее через начальный параметр при создании формы PayPalPaymentsForm.

def view\_that\_asks\_for\_money(request):

# Описание того что должна делать кнопка платежа

paypal\_dict = {}

# Создание формы

form = PayPalPaymentsForm(initial=paypal\_dict)

# Отрисовка страницы с контекстом в виде формы

Добавим url птатрен в urls.py.

url(r'^paypal/', include('paypal.standard.ipn.urls')),

Всякий раз, когда IPN обрабатывается сигнал будет отправлен с результатом транзакции.

* Valid\_ipn\_received : Указывает на правильное, не повторяющееся сообщение IPN от PayPal. Обработчик будет получать paypal.standard.ipn.models.PayPalIPN в качестве отправителя. Нужно будет проверить атрибут payment\_status и business, чтобы убедиться, что счет, получающий платеж, является ожидаемым, а также другие атрибуты, чтобы знать, какие действия предпринять.
* Invalid\_ipn\_received : Отправляется, когда транзакция была помечена - из-за неудачной проверки с PayPal, например, или дубликат идентификатора транзакции.

def show\_me\_the\_money(sender, \*\*kwargs):

ipn\_obj = sender

if ipn\_obj.payment\_status == ST\_PP\_COMPLETED:

# Убедится, что адрес электронной почты получателя совпадает с предыдущим

if ipn\_obj.receiver\_email != "receiver\_email@example.com":

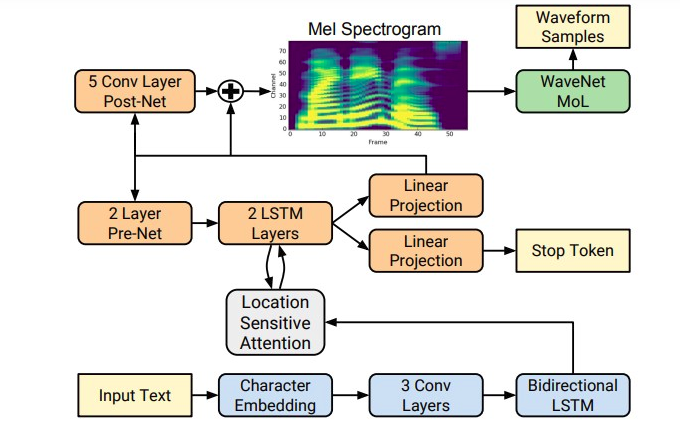
# не верная оплата

# проверка количества

# зачисление символов

## Реализация подсистемы конвертации

Для реализации конвертации текста в речь использовалась нейронная сеть Tacotron 2. Представленная Deepmind в 2017 году (рисунок 18).

Рисунок 18 — Tacotron 2

Tacotron 2 представляет собой не одну сеть, а две: Feature prediction net и NN-vocoderWaveNet. Исходная статья, а также наше собственное видение проделанной работы позволяют считать первой скрипкой Feature prediction net, в то время как вокодер WaveNet играет роль периферийной системы. Tacotron2 — это архитектура sequence to sequence. Она состоит из кодировщика (encoder), создающего некоторое внутреннее представление о входном сигнале (символьных токенах), и декодировщика (decoder), который превращает это представление в мел-спектрограмму. Также крайне важным элементом сети является так называемый PostNet, призванный улучшить спектрограмму, сгенерированную декодером.

Первым слоем кодировщика является Embedding-слой. Он на основании последовательности натуральных чисел, представляющих символы, создает многомерные (512-мерные) векторы. Далее embedding-векторы подаются в блок из трёх одномерных свёрточных слоев. Каждый слой включает в себя 512 фильтров длиной 5. Это значение является хорошим размером фильтра в данном контексте, потому что захватывает некоторый символ, а также два предыдущих и два последующих его соседа. За каждым свёрточным слоем следует нормализация по мини-батчам и ReLU-активация.

Тензоры, полученные после свёрточного блока, подаются на двунаправленные LSTM-слои, по 256 нейронов в каждом. Результаты прямого и обратного прохода конкатенируются. Декодер имеет рекуррентную архитектуру, то есть на каждом последующем шаге используются выходные данные с предыдущего шага. Здесь ими будет являться один фрейм спектрограммы. Ещё одним важным, если не ключевым, элементом данной системы является механизм мягкого (обучаемого) внимания — относительно новый и набирающий всё большую популярность приём.

Нейросеть была обучена с использованием датасета LJSpeech dataset содержащим 24 часа аудио. Для тренировки использовался сервис Google Colab пердостовляющий бесплатный доступ к вычислительным ресурсам, а также TPU.

Модель загружается в память при старте celery здаачи и ожидает сообщения из очереди RabbitMQ.

# Тестирование

## Юнит тестирование

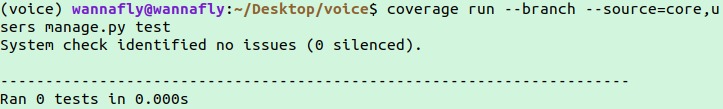
Coverage.py это самая популятный инструмент python для оценки покрытия тестами. Очень прост в установке и доступен из PyPi. Произведем установку.

$ pipenv install coverage

Теперь мы можем измерять покрытие тестами. Покрытие кода измеряет, какие строки кода были выполнены во время теста. В идеале, отслеживая охват кода, мы можем гарантировать, какой код тестируется, а какой нет. Проекты Django в основном Python, мы можем использовать Coverage.py для измерения покрытия кода. Два момента:

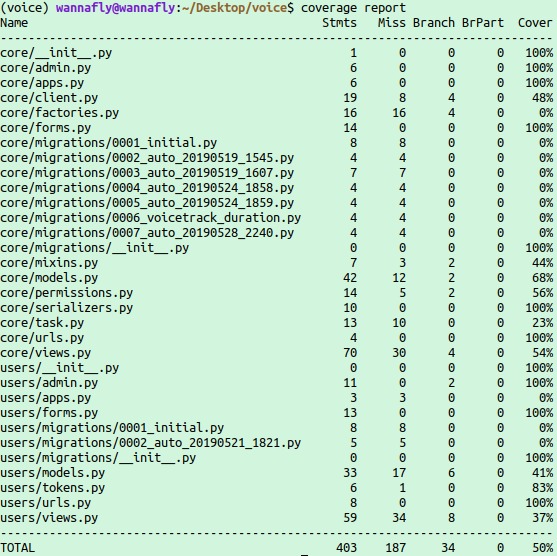
* Coverage.py не измерят покрытие шаблонов.
* Нетестированные CBV уже покрыты некоторыми тестами.

Для того чтобы определить покрытие проекта нужно запустить тесты с командой coverage. Сгенерировать отчет используся coverage report команду. Запустим юнит тестирование проекта.

Рисунок ­ Команда coverage без тестов

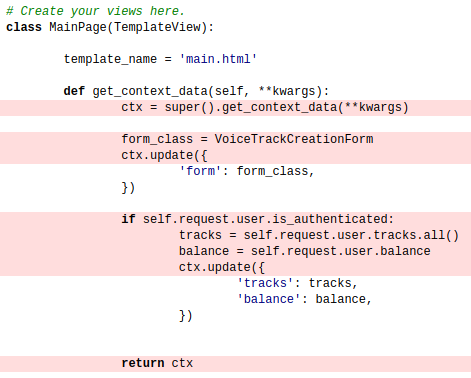
Эта команда говорит coverage выполнить команду test для записи покрытия. В команде присутствуют две опции:

* --branch: Для отслеживания двух путей i f, true и false.
* --source=croe, users: Для отслеживания только core и users модулей.

Рисунок ­ Результат покрытия кода тестами без тестов

Для того чтобы понять почему проект без тестов покрыт на 50% нужно взглянуть на html версию core/veiws.py.

$ coverage html

Рисунок ­ Html очтет покрытия

На рисунке показано что MainPage полностью покрыт за исключением MainPage.get\_context\_data(). Инструмент для оценки покрытия проверяет была ли выполнена конкретная строчка кода во время теста. На рисунке MainPage и его члены были выполнены. При запуске теста, django создаст корневой URLConf так же, как при запуске для разработки или продакшена. При создании корневого URLConf он импортирует другие ссылки Urlconf (например, core.urls.py ). Эти Urlconf, в свою очередь, импортируют свои представления: представлений формы, модели и другие модули. Эта цепочка импорта означает, что все, что находится на верхнем уровне модуля, будет покрыто, независимо от того, тестируется ли он. Определение класса MainPAge было выполненно. Однако сам класс не был создан, и ни один из его методов не был выполнен. Это также объясняет, почему тело MainPage.get\_contex\_data() не покрыто. Тело MainPage.get\_contex\_data() никогда не выполнялось. Это ограничение инструментов покрытия кода при работе с декларативным кодом, таким как MainPage .

Теперь напишем юнит тесты. Django помогает писать модульные тесты для тестирования отдельных единиц кода. Если код полагается на внешнюю службу, то мы можем использовать стандартную библиотеку unittest.mock, чтобы имитировать API, предотвращение запросов к внешним системам.

Для упращения тестирования мы будем использовать Factory boy. Для начала установим.

$ pipenv install factory-boy

Нам нужны фабрики для создания плоьзователя и VoiceTrack. Оба обьекта связанны. Создание всех связанных моделей для каждого теста с использованием менеджера занимают много строчек кода и отвлекают от цели тестов. Джанго предлагает поддержку из коробки для инструментов для тестов. factory boy поможет нам, позволяя использовать код, UserFactory и VoiceTrackFactory, который может создавать экземпляры модели во время выполнения на основе состояния текущего пользователя модель.

UserFactory бдует наследованна от класса factory boy DjangoModelFactory, ктоорый умеет обращатся с моделями django. Мы используем класс Meta, для того чтобы сказать UserFactory какую модель создавать. Также добавим атрибуты класса что бы сказать factory boy какие значения задавать модели.

class UserFactory(factory.DjangoModelFactory):

class Meta:

# Указание модели

# указание значения email в виде лямбды

email = factory.Sequence(lambda n: 'user%d' % n)

class VoiceTrackFactory(factory.DjangoModelFactory):

# указание значения text в виде лямбды

text = factory.Sequence(lambda n: 'foo%d' % n)

# Остольные поля

class Meta:

# Указание модели

Приступим к написанию юнит тестов. Юнит тестирование представления означает прямую передачу представленю запроса и утверждение, что ответ соответствует нашим ожиданиям. Поскольку мы передаем запрос непосредственно в представление, нам также необходимо напрямую передать любые аргументы, которые представление обычно получает из URL-адрес запроса. Разбор значений из путей URL-адресов является обязанностью запроса маршрутизация, которую мы не используем в юнит тесте представления.

class MainTestCase(TestCase):

""" Tests the MainPage view """

REQUEST = RequestFactory().get(path='/')

def test\_GET\_on\_day\_with\_no\_questions(self):

# получение response

response = views.MainPage.as\_view()(self.REQUEST,)

# проверка кода статуса

# проверка имени шаблона

# проверка содержимого шаблона

## Создание интегрированного тестирования

Интегрированные тесты использует тот же класс TastCase, что и юнит тест. Интегрированный тест покажет нам, может ли наш проект направить запрос в представление и вернуть правильный ответ. Запрос теста интеграции должен пройти через все промежуточное фильтры и маршрутизацию URL, с которой настроен проект. Чтобы помочь нам написать интеграционные тесты, Django предоставляет TestCase.client. TestCase.client это инструмент который позволяет отсылать HTTP запросы к нашему проекту. Client также предоставляет нам методы типа login(), для начала сесии аунтефикации. Мы будем использовать фабрики factory boy. Напишем интегрированные тесты. Для начала протестируем создание и активацию пользователей.

class UserActiveManagerTestCase(TestCase):

def testConfirmedSubscribersForMailingList(self):

# Создание нескольких неактивных пользователей

confirmed\_users = [UserFactory(is\_active=True) for n in range(3)]

# Создание нескольких активных пользователей

confirmed\_users = [UserFactory(is\_active=false) for n in range(3)]

# Получение всех пользователей

# получение активных и неактивных пользователей

# проверка на совпадение количества активных и неактивных

Далее протестируем создание VoiceTrack. Для этого также будем использовать ранее созданную фабрику.

class VoiceTrackCreationTestCase(TestCase):

def test\_calling\_create\_queues\_confirmation\_email\_task(self):

# создание обьекта

track = VoiceTrackFactory(text='foo', owner=UserFactory())

# Проверка наличия обьекта в базе данных

## Тестирование RestAPI

API обеспечивают ценность для наших пользователей, позволяя им автоматизировать их взаимодействие с сервисом. DRF помогает нам автоматизировать тестирование нашего кода, предоставляя замену всем необходимым инструментам dajngo.

* APIRequestFactory для django RequestFactory.
* APIClient для django Client.
* APITestCase для dajngo TestCase.

APIRequestFactory и APIClient облегчают отправку запросов, отформатированных для нашего API. Например, они предоставляют простой способ установки учетных данных для запроса, который не полагается при аутентификации на основе сеанса. В противном случае эти два класса служат той же цели, что и их эквиваленты Django по умолчанию.

class TestAPI(APITestCase):

# Метод для инициализации

def setUp(self):

# инициализация клиента

# инициализация фабрики

# инициализация представления

# инициализация фабрики

# инициализация uri

# инициализация пользователя

# инициализация токена

# тестирование списка треков

def test\_list(self):

# запрос с токеном авторизации

# проверка статуса ответа

# тестирование созданных стаска треков

def test\_listing\_all\_my\_voice\_lists(self):

# создание треков с помощью фабрики

# запрос с токеном авторизации

# проверка статуса ответа

# парсинг и проверка количества треков

# проверка совпадения текста и id

# Заключение

В ходе работы над преддипломной практикой было разработано приложение для конвертации текста в речь.

В ходе разработки автоматизированной системы, были выполнены следующие функции: предоставление возможности конвертации, по средством формы через клиент и RestAPI, а также хранение, загрузку, удаление и проигрывание ранее созданных записей.

Разработанная программа корректно работает с данными, выдаёт сообщения об ошибках, уведомления об успехах, в удобном для пользователя режиме.

При разработке программы учитывались все требования, которыми должен обладать проект. Программа проста в использовании, не требует от пользователя глубоких знаний в сфере компьютерных технологий, имеет простой интерфейс.

Было проведено тестирование программы на корректную работу всех ее функций.

Программа полностью соответствует условиям и готова к размещению в глобальной сети интернет.

Преимущество приложения заключается в том, что оно позволяет производить конвертацию в приятную речь как прямого текста так и текстовых документов.

# Cписок использованных источников

1. Git [Электронный ресурс]. – 2018. – Режим доступа: https://git-scm.com – Дата доступа: 05.10.2018.
2. Ларман, К. Применение UML 2.0 и шаблонов проектирования / К. Ларман – С.: Издательский дом «Вильямс», 2013. – 1059 с.
3. Самоучитель Python [Электронный ресурс]. – 2018. – Режим доступа: https://pythonworld.ru/samouchitel-python – Дата доступа: 15.10.2018.
4. Свейгат, Э. Автоматизирование рутинных задач с помощью Python / Э. Свейгат – С.: Издательский дом «Вильямс», 2016. – 581 с.
5. J. Sotelo, S. Mehri, K. Kumar, J. F. Santos, K. Kastner, A. Courville, and Y. Bengio, “Char2Wav: End-to-end speech synthesis,” in Proc. ICLR, 2017.
6. S. O. Arik, G. F. Diamos, A. Gibiansky, J. Miller, K. Peng, ¨ W. Ping, J. Raiman, and Y. Zhou, “Deep voice 2: Multi-speaker neural text-to-speech,” CoRR, vol. abs/1705.08947, 2017.