Содержание

1 STTS API v2.0 (STTS v1.4.0+) 2
1.1 Передача данных 2
1.1.1 HTTP заголовки (Content-Type / Accept) 2
1.1.2 Если заголовки не указаны 2
1.1.3 GET запросы 2
1.2 Получение данных 3
1.3 Запросы с авторизацией 3
1.4 Синхронные / асинхронные/ callback запросы 4
1.4.1 Асинхронное получение результата 4
1.4.2 Получение результата callback запроса 4
1.5 Методы информации о моделях 5
1.5.1 Список загруженных моделей с их параметрами (метод /system/get_models) 5
1.5.2 Получение списка всех установленных моделей с их параметрами (метод /system/get_models_all) 6
1.5.3 Структура данных правил корректора 7
1.5.4 Структура данных контекста модели распознавания 7
1.5.5 Получение данных оптимизации установленной модели (метод /system/get_model_data) . 8
1.5.6 Обновление данных оптимизации установленной модели (метод /system/set_model_data) 9
1.6 Методы распознавания речи 10
1.6.1 Формат фрагмента обработанных данных 10
1.6.2 Распознавание речи в файле (метод /chain) 11
1.6.3 Распознавание речи в потоке (онлайн) 12
1.6.3.1 Резервирование декодера (метод /asr/online) 12
1.6.3.2 Получение результатов потокового декодера (метод /online/results) 13
1.6.3.3 Передача данных (метод /online/feed) 13
1.7 Методы обработки речевого сигнала 14
1.7.1 Определение параметров речи (метод /vad) 14
1.7.1.1 Дополнительные параметры метода /vad 14
1.7.2 Разделение по дикторам (метод /diariz) 15
1.7.2.1 Дополнительные параметры метода /diariz 15
1.7.3 Определение языка речи (метод /lang) 17
1.8 Методы идентификации дикторов (метод /bio) 18
1.8.1 Формат фрагмента обработанных данных 18
1.8.2 Получение слепка диктора/эталона из аудиофайлов 18
1.8.3 Сравнение аудиофайлов со слепками эталонов 18
1.8.4 Сравнение слепков дикторов со слепками эталонов 19
1.8.5 Сравнение аудиофайлов и слепков дикторов со слепками эталонов 19
1.8.6 Поиск дикторов в аудиофайле 20
1.9 Методы Адаптации 21
1.9.1 Адаптация нейросетевой лингвистической модели (метод /mas) 21

1 STTS API v2.0 (STTS v1.4.0+)

1.1 Передача данных

API поддерживает gRPC, REST синхронные и асинхронные запросы, передачу данных в форматах JSON или MessagePack.

> ❗ При передаче данных в JSON - все бинарные данные должны быть закодированы `base64`

Сервис работает с форматом `pcm_s16le, mono, 8000 Hz, 16 bit, 128 kb/s`.

> ❗ Хранение и передача данных в формате отличном от вышеуказанного - может ухудшить качество распознавания.

Все методы, кроме онлайна/gRPC, принимают на вход данные любого формата, которые поддерживает ffmpeg, и переконвертируют их.

> ❗ Контейнер mp4 поддерживается только собранный с параметрами `-movflags +faststart`

1.1.1 HTTP заголовки (Content-Type / Accept)

```
# для передачи/получения данных в JSON
application/json

# для передачи/получения данных в MessagePack любой из следующих:
application/x-msgpack
application/vnd.msgpack
application/msgpack
```

1.1.2 Если заголовки не указаны

- формат входных данных из двух вариантов определяется автоматически
- формат выходных данных соответствует формату входных
- для GET запросов ответ выдаётся в формате JSON

1.1.3 GET запросы

Для GET запросов тип входных и выходных данных можно указать в url.

```
# для передачи и получения данных в JSON
<address>/<method>?format=msgpack
# для передачи и получения данных в MessagePack
<address>/<method>?format=json
```

1.2 Получение данных

Разметка в данных возвращается в секундах

Если запрос неуспешен - HTTP код ответа соответствует ошибке, в теле ответа - подробное описание ошибки.

1.3 Запросы с авторизацией

Для некоторых системных запросов необходима авторизация.

GET запрос с авторизацией

```
AUTH = "login:password"
URL = f"{HOST}/system/{METHOD}"
HEADERS = {"Authorization": f"Basic {b64encode(AUTH.encode()).decode()}"}
with urllib.request.urlopen(Request(URL, headers=HEADERS)) as f:
    result = json.loads(f.read().decode())
```

POST запрос с авторизацией

```
AUTH = "login:password"
URL = f"{HOST}/system/{METHOD}"
HEADERS = {"Authorization": f"Basic {b64encode(AUTH.encode()).decode()}"}
data = {
  "key": "value"
}
with urllib.request.urlopen(Request(URL, data=json.dumps(data).encode(), headers=HEADERS)) as f:
    result = json.loads(f.read().decode())
```

1.4 Синхронные / асинхронные/ callback запросы

По умолчанию запрос является синхронным: блокируется до полной обработки файла и возвращает весь результат.

Для асинхронного запроса необходимо дополнить адрес спецификатором `/async`.
Результатом асинхронного варианта является идентификатор (строка) созданной задачи `task_id`,
по которому надо получать статус выполнения `/async/get_status` и полный результат `/async/result`.

Для callback запроса - дополнить адрес спецификатором `/callback`.
Результаты callback запроса отправляются на указанные в запросе ip:port по мере выполнения.

1.4.1 Асинхронное получение результата

запрос статуса задачи: **(/async/get_status)**

```
data = {
  "apikey":  str, # Персональный ключ
  "task_id": str, # Идентификатор, полученный из запроса .../async
}
with urllib.request.urlopen(f"{HOST}/async/get_status", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

результат:

```
{ "queue":   bool, # True - задача в очереди на исполнение
  "process": bool, # True - задача исполняется
  "done":    bool, # True - задача выполнена
}
```

запрос результата задачи: **(/async/result)**

```
data = {
  "apikey":  str, # Персональный ключ
  "task_id": str, # Идентификатор, полученный из запроса .../async
}
with urllib.request.urlopen(f"{HOST}/async/result", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

1.4.2 Получение результата callback запроса

ожидание результата на другом адресе:

```
from http.server import BaseHTTPRequestHandler, HTTPServer
import msgpack

class Handler(BaseHTTPRequestHandler):
  def do_POST(self):
    content_length = int(self.headers["Content-Length"])
    raw_data = self.rfile.read(content_length)
    result = msgpack.unpackb(raw_data, raw=False)
    print(result)
    return

HTTPServer(("", PORT), Handler).serve_forever()
```

1.5 Методы информации о моделях

1.5.1 Список загруженных моделей с их параметрами (метод /system/get_models)

Получить список доступных моделей можно с помощью GET или POST запросов.

запрос GET:

```
with urllib.request.urlopen(f"{HOST}/system/get_models?apikey={APIKEY}") as f:
    result = json.loads(f.read().decode())
```

запрос POST:

```
data = {
  "apikey": str, # Персональный ключ
}
with urllib.request.urlopen(f"{HOST}/system/get_models", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

формат результата:

```
{ model_type: {
    model_name: {
        model_params...
    }, ...
  }, ...
}
```

пример ответа:

```
{ "vad": {
    "common": {
      "emotion": True,
      "gender": True,
      "offline_workers": 8}},
  "asr": {
    "uzb_t48_4": {
      "language": "Uzbek",
      "offline_corrector": "uzb_foreign",
      "offline_workers": 8},
    "rus_f15_ph8b": {
      "language": "Russian",
      "offline_corrector": "rus_default",
      "offline_workers": 20,
      "online_corrector": "rus_noswear",
      "online_workers": 8},
    "rus_s28_3": {
      "language": "Russian",
      "online_corrector": "rus_numbers",
      "online_workers": 8}},
  "lang": {
    "big":   {"offline_workers": 10},
    "small": {"offline_workers": 2}},
  "diariz": {
    "gsm": {"offline_workers": 4}},
  "bio": {
    "mic_small": {"offline_workers": 4}},
}
```

1.5.2 Получение списка всех установленных моделей с их параметрами (метод /system/get_models_all)

> ❗ Метод только для on-prem, необходима авторизация.

None или отсутствие параметра в результатах - означает неприменимость у данного типа модели

запрос:

```
# необходима авторизация
with urllib.request.urlopen(f"{HOST}/system/get_models_all") as f:
    result = json.loads(f.read().decode())
```

результат:

```
{ "asr": {
    <имя_модели_asr>: {
      "language": str,  # Язык модели (название на английском)
      "offline": bool,  # [опционально] Загружена ли offline модель
      "online": bool,   # [опционально] Загружена ли online модель
      "contexts": list, # [опционально] Список имён контекстов
      "cp": bool,       # [опционально] Возможность дообучения нейролингвистической модели
      "elms": {         # [опционально] Список нейролингвистических моделей
        <имя_модели_elm>: {
          "status": bool, # Доступность нейролингвистической модели
          "msg":    str,  # Причина недоступности, если status=False
        }, ...
    }, ...
  },
  "corrector": {
    <iso3>: {             # Код языка корректора (формат ISO-639-3)
      "language": str,    # Язык корректора (название на английском)
      "rules": list,      # Доступные типы правил
      "rules_file": bool, # Заданы ли пользовательские правила
    }, ...
  },
  "vad": {
    <имя_модели_vad>: {
      "offline": bool, # Загружена ли offline модель
      "emotion": bool, # Загружена ли модель эмоций
      "gender": bool,  # Загружена ли модель пола/возраста
    }, ...
  },
  <имя_сервиса>: {
    <имя_модели_сервиса>: {
      "offline": bool, # Загружена ли offline модель
      "online": bool,  # [опционально] Загружена ли online модель
    }, ...
} }
```

1.5.3 Структура данных правил корректора

Для постобработки результатов распознавания применяется Корректор, в котором пользователь может задавать правила обработки.

У поддерживаемых языков есть пользовательские правила:
- `replace` замены, выполняют простую замену
- `abbrs` аббревиатуры (только для русского языка)

Ключ - слово, на которое происходит замена.
Значение - список сочетаний букв, которые необходимо преобразовать.

> ❗ Аббревиатуры обязательно передать используя только алфавит языка, разделяя буквы пробелами

пример:

```
{ "abbrs": {
    "МФЦ": ["м ф ц", "э м ф э ц э"],
    "МРЭО": ["м р э о"]
  },
  "replace": {
    "ауди": ["бмв"]
} }
```

1.5.4 Структура данных контекста модели распознавания

В моделях распознавания речи можно настраивать контексты, которые влияют на распознавание заданных слов и словосочетаний.

Для каждого элемента необходимо указать:
контекстную фразу `str`
или
списков из двух значений - `[фраза (str), вес фразы (float)]`

```
[ ["доброе утро", 2.0],
  ["солнце", 1.6],
  "дела"
]
```

1.5.5 Получение данных оптимизации установленной модели (метод /system/get_model_data)

> ❗ Метод только для on-prem, необходима авторизация.

Метод используется для последующего изменения данных, либо для переноса оптимизации модели на другой комплекс

запрос:

```
data = {
  "alphabet": {   # [опционально] Получение данных алфавита модели
    "model": str, # Имя asr модели
  },
  "context": {    # [опционально] Получение данных контекстного графа модели
    "model": str, # Имя asr модели
    "name": str,  # Имя контекстного графа
  },
  "elm": {        # [опционально] Получение данных нейросетевой лингвистической модели
    "model": str, # Имя asr модели
    "name": str,  # Имя нейросетевой лингвистиеской модели
  },
  "corrector": {  # [опционально] получение данных правил корректора
    "iso3": str,  # iso-код языка правил корректора (формат ISO-639-3)
} }
# необходима авторизация
with urllib.request.urlopen(f"{HOST}/system/get_model_data", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

результат:

```
{ "alphabet": {        # [опционально] При наличии ключа alphabet в запросе
    "data": str|None,  # Данные алфавита, При отсутствии файла - None
  },
  "context": {         # [опционально] При наличии ключа context в запросе
    "hash": str|None,  # хеш данных в системе, нужен при обновлениии/переносе данных
    "data": dict|None, # Данные контекста, При отсутствии файла - None
  },
  "elm": {             # [опционально] При наличии ключа elm в запросе
    "data": str|None,  # данные нейросетевой лингвистической модели в формате base64, При отсутствии файла -
None
  },
  "corrector": {       # [опционально] При наличии ключа corrector в запросе
    "hash": str|None,  # хеш данных в системе, нужен при обновлениии/переносе данных
    "data": dict|None, # Данные правил языка корректора, При отсутствии файла - None
} }
```

1.5.6 Обновление данных оптимизации установленной модели (метод /system/set_model_data)

> ❗ Метод только для on-prem, необходима авторизация.

запрос:

```
data = {
  "context": {
    "model": str,        # Имя модели распознавания
    "name":  str,        # Имя контекста
    "hash":  str,        # [опционально] Хэш оригинального контекста
    "data":  list[list], # Данные нового контекста
  },
  "elm": {
    "model": str,        # Имя модели распознавания
    "name":  str,        # Имя нейросетевой лингвистической модели
    "data":  str,        # Файл нейросетевой лингвистической модели, закодированный в base64
  },
  "corrector": {
    "iso3": str,         # Язык корректора
    "hash": str,         # [опционально] Хэш оригинальных правил языка корректора
    "data": dict,        # Данные новых правил языка корректора
} }
# необходима авторизация
with urllib.request.urlopen(f"{HOST}/system/set_model_data", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

результат:

```
{ "context": "done",   # [опционально] При наличии ключа context в запросе
  "elm": "done",       # [опционально] При наличии ключа elm в запросе
  "corrector": "done", # [опционально] При наличии ключа corrector в запросе
}
```

1.6 Методы распознавания речи

1.6.1 Формат фрагмента обработанных данных

```
{ "client_id": str,      # [опционально] id запроса, полученный от клиента
  "task_id": str,        # внутренний id запроса

  "speech": bool,        # была ли найдена речь во фрагменте
  "error": str,          # [опционально] если при распознавании данного фрагмента произошла ошибка

  "utterance": bool,     # [только в /online] окончание фразы, финальная гипотеза
  "last": bool,          # [только в /online, /callback] последний фрагмент

  "start": float,        # начало фрагмента
  "end": float,          # конец фрагмента

  "name": str,           # имя типа разметки; например: speech, __noise, online
  "nbest": [             # [опционально] варианты распознавания фрагмента
    { "score": float,    # confidence данного варианта распознавания (мера уверенности распознавания)
      "duration": float, # длительность речи
      "marking": [       # разметка фрагмента по словам (может быть пустым списком []):
        [ float,         # начало в секундах
          float,         # конец в секундах
          str,           # слово
          float          # confidence данного слова
        ], ...
      ],
      "draft": str,      # [опционально] текст варианта распознавания до применения корректора
      "text": str        # текст варианта распознавания
    }, ...
  ],
  "vad": {               # [опционально] описание разметки VAD, если в запросе указанно vad_results:true
    "duration": int,     # длительность фрагмента
    "marking": [         # разметка VAD:
      [ float,           # начало в секундах
        float,           # конец в секундах
        str              # тип разметки
        str|None,        # [опционально] эмоции в сегменте
        str|None,        # [опционально] пол/возраст в сегменте
      ], ...
    ]
} }
```

1.6.2 Распознавание речи в файле (метод /chain)

Поддерживаются следующие цепочки выполнения:

- vad -> asr -> corrector
- vad -> lang-> asr -> corrector
- lang -> vad -> asr -> corrector

запрос:

```
data = {
  "apikey": str,            # Персональный ключ
  "media" : bytes,          # аудио/видеофайл

  "client_id": str,         # id запроса от клиента, фигурирует в логах и результатах
  "priority": int,          # Приоритет запроса во входной очереди 3(высокий)..7(низкий) (по умолчанию 5)
  "vad_results": bool,      # Возвращать ли результат VAD (по умолчанию False)
  "make_utterance": bool,   # Ставить ли заглавную и точку при отключенной постобработке (по умолчанию False)
  "callback_url": str,      # Вызывается на каждый фрагмент; пример: "127.0.0.1:5000"

  "chain": {
    "_params": {            # [опционально] параметры цепочки и переходов между этамами
      "sequence": list,     # [опционально] последовательность выполнения этапов
      "lang_asr": {         # параметры перехода lang -> asr, обязательно при наличии такого перехода
        "asr_models": list, # список asr моделей для определения языков и распознавания
      }
    }
    "vad": {                # [опционально] При отсутствии - данные нарезаются по 1 минуте
      "model": str,         # Имя модели детектора речи
      # Все возможные параметры vad см. в разделе "Определение параметров речи"
    },
    "lang": {               # [опционально]
      "model": str,         # Имя модели определения языка
      "threshold": float,   # [опционально] общий порог разделения языков, (по умолчанию 0.5)
    },
    "asr": {
      "model": str,         # Имя модели распознавания

      # Далее опциональные параметры:
      "nbest": int,         # Сколько вариантов распознавания выводить (по умолчанию 1)
      "context': str,       #
      "extra_words': list,  #
      "extra_score': float, #
      "elm_model': str,     #
    },
    "corrector": bool|str|list,  # [опционально] Использовать ли постобработку (по умолчанию True)
} }
with urllib.request.urlopen(f"{HOST}/chain", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

результат:

Результатом является список фрагментов обработанных данных, описанный выше в разделе "Формат фрагмента обработанных данных".

1.6.3 Распознавание речи в потоке (онлайн)

Позволяет распознавать поток аудиоданных и получать результаты с минимальной задержкой.
Для открытия http-потока клиент должен поддерживать `Chunked transfer encoding (RFC 7230 section 4.1)`.

> ❗ Timeout бездействия онлайн-декодера - 3сек, после которых он автоматически освобождается с удалением результатов

1.6.3.1 Резервирование декодера (метод /asr/online)

запрос:

```
data = {
  "apikey": str,                # Персональный ключ
  "client_id": str,             # id запроса от клиента, фигурирует в логах и результатах

  "asr": {
    "model": str,               # Имя модели распознавания

    # Далее опциональные параметры:
    "nbest": int,               # Сколько вариантов распознавания выводить (по умолчанию 1)
    "context": str,             #
    "extra_words": list,        #
    "extra_score": float,       #

    "return_phrase": bool,      # Отдавать результат только финализированных фраз (по умолчанию True)
    "timeout_silence": float,           # Длительность паузы с начала сигнала, для финализации пустой фразы
(по умолчанию 10.0)
    "timeout_utterance_silence": float, # Длительность паузы в речи, для финализации фразы (по умолчанию 1.0)
    "timeout_utterance": float,         # Максимальная длительность непрерывной речи, для принудительной
финализации фразы (по умолчанию 120.0)
  },
  "corrector": bool|str|list,   # Использовать ли постобработку (по умолчанию True)
}
with urllib.request.urlopen(f"{HOST}/asr/online", json.dumps(data).encode()) as f:
    task_id = json.loads(f.read().decode())
```

результат:

Результатом резервирования является идентификатор задачи `task_id` для последующих вызовов.

1.6.3.2 Получение результатов потокового декодера (метод /online/results)

> ❗ Запрос GET, параметры apikey и request_id передаются в URI

запрос: **(с использованием фреймворка requests)**

```
for res in requests.get(f"{HOST}/online/results?apikey={APIKEY}&task_id={TASK_ID}",
stream=True).iter_content(None):
  result = json.loads(res)
```

результат:

В chunked потоке возвращаются гипотезы фрагментов фраз.
Формат результата аналогичен результату метода `/chain`:

Получение `"utterance": True` означает:

- завершение фразы, финальная гипотеза

Получение `"last" : True` означает:

- последний пакет
- декодер освобождён

1.6.3.3 Передача данных (метод /online/feed)

> ❗ Параметры apikey и request_id передаются в URI

> ❗ данные передаются без заголовка, только в формате `pcm_s16le, mono, 8000 Hz, 16 bit, 128 kb/s`,

Все данные должны быть отправлены в одном соединении с использованием `chunked transfer encoding`:
Схематическое представление: `<длина_чанка_в_HEX>\0x0D\0x0A<содержание_чанка>\0x0D\0x0A`
Последний чанк строится по той же схеме: `0\0x0D\0x0A\0x0D\0x0A`

запрос: **(с использованием фреймворка requests)**

```
generator = (base64.b64encode(chunk) for chunk in split_wav(WAV))

r = requests.post(f"{HOST}/online/feed?apikey={APIKEY}&task_id={TASK_ID}", data=generator, stream=True)
```

запрос: **(реализация без использования фреймворка)**

```
from http.client import HTTPConnection
conn = HTTPConnection(HOST)
conn.connect()
conn.putrequest("POST", f"/online/feed?apikey={APIKEY}&task_id={TASK_ID}")
conn.putheader("Transfer-Encoding", "chunked")
conn.endheaders()
last_chunk = len(wav_chunked) - 1
for i, chunk in enumerate(wav_chunked):
  chunk_body = hex(len(chunk))[2:].encode() + b"\r\n" + chunk + b"\r\n"
  conn.send(chunk_body)
  if i == last_chunk:
    break
  time.sleep(SPLIT_SEC)
conn.send(b"0\r\n\r\n")
r = conn.getresponse()
print(r.status, r.reason, r.read())
```

1.7 Методы обработки речевого сигнала

1.7.1 Определение параметров речи (метод /vad)

запрос:

```
data = {
  "apikey": str,             # Персональный ключ
  "media": bytes,            # аудио/видеофайл

  "vad": {
    "model": str,            # Имя модели детектора речи

    # Далее опциональные параметры: (изменяют формат ответа)
    "segment_emotion": bool, # определять эмоции в каждом сегменте с речью (по умолчанию False)
    "segment_gender":  bool, # определять пол/возраст в каждом сегменте с речью (по умолчанию False)

    # Дополнительные параметры описаны ниже
} }
with urllib.request.urlopen(f"{HOST}/vad", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

формат результата:

```
[ [ float, # начало в секундах
    float, # конец в секундах
    str,   # тип разметки (речевой - "speech")
  ], ...
]
```

формат результата: **(с использованием emotion и/или gender)**

```
[ [ float,    # начало в секундах
    float,    # конец в секундах
    str,      # тип разметки (речевой - "speech")
    str|None, # эмоции в сегменте (None, если не указан в запросе)
    str|None, # пол/возраст в сегменте (None, если не указан в запросе)
  ], ...
]
```

1.7.1.1 Дополнительные параметры метода /vad

- **post.speech.threshold.prob** `float [0.0, 1.0] default: 0.65`
  Порог для определения речи по вероятности.
- **post.speech.threshold.energy** `float [0.0, 0.0001] default: 0.0`
  Порог для определения речи по энергии.
- **post.speech.expand** `float [0.0, 1.0] default: 0.2`
  Добавляет указанное кол-во секунд к выделенным как речь сегментам.
- **post.speech.min_dur** `float [0.1, 0.5] default: 0.1`
  Минимальная длительность речи в секундах для выделения ее в сегменте.
- **post.pause.min_dur** `float [0.1, 1.5] default: 0.8`
  Минимальная длительность пустого сигнала в секундах, для разделения сегментов речи.
- **predict.split_size** `int [30, 120] default: 60`
  Максимальная длительность одного сегмента речи.
  Может несколько превышать заданную длину, чтобы не обрезать сегмент речи посередине.
- **predict.min_signal_duration** `float [0.1, 1.0] default: 0.1`
  Минимальная длительность сегмента в секундах для выделения его в сигнале.

1.7.2 Разделение по дикторам (метод /diariz)

запрос:

```
data = {
  "apikey": str,                        # Персональный ключ
  "media" : bytes,                      # аудио/видеофайл

  "diariz": {
    "model":  str,                      # Имя модели распознавания

    # обязательно задать один из двух параметров:
    "diarization.clustering.clusters": 4,       # Заранее известное количество дикторов
    # "diarization.clustering.threshold": 0.64, # Мера уверенности, по которой разделяить дикторов

    # Далее опциональные параметры:
    "diarization.pre.asr": ASR_RESULTS, # результат распознавания от asr

    # Дополнительные параметры описаны ниже
} }
with urllib.request.urlopen(f"{HOST}/diariz", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

результат:

```
[ [ float, # начало в секундах
    float, # конец в секундах
    str,   # номер диктора
  ], ...
]
```

1.7.2.1 Дополнительные параметры метода /diariz

- **diarization.chunk_duration** `float [1.0, 30.0] default: 1.5`
  Длина аудио фрагмента (в секундах), который обрабатывается за один раз.
- **diarization.chunk_hop** `float [0.0, 5.0] default: 0.75`
  Шаг перекрытия между соседними фрагментами (в секундах).
  Помогает избежать "разрывов" на границах.
- **diarization.dvectors.l2_norm** `bool [True, False] default: False`
  Нормализует длину вектора признаков.
  Нормализация улучшает устойчивость к разной громкости, артефактам записи и другим вариациям.
  Подходит для любого алгоритма.
- **diarization.preprocess.enabled** `bool [True, False] default: False`
  Стандартизация векторов признаков для более стабильного поиска центроид.
  Полезно для алгоритма **kmeans**.
- **diarization.clustering** `default: agglomerative`
  Метод для сопоставления сегментов аудио дикторам:
  `agglomerative` – Иерархический "снизу вверх": объединяет ближайшие кластеры шаг за шагом.
  `kmeans` – Итеративный: случайные центроиды → перераспределение точек → обновление центров.
- **diarization.clustering.affinity** `default: cosine`
  Метод вычисления сходства между сегментами:
  `cosine` – Косинусная близость.
  `euclidean` – Евклидово расстояние.
- **diarization.clustering.linkage** `default: average`
  Метод объединения кластеров в **agglomerative** кластеризации:
  `single` – минимальное расстояние.
  `average` – среднее расстояние.
  `complete` – максимальное расстояние.

- **diarization.clustering.clusters** `int (1, 30) default: None`
  Ожидаемое количество кластеров (дикторов).
  При указании числа дикторов - параметр **diarization.clustering.threshold** игнорируется.
- **diarization.clustering.threshold** `default: 0.7`
  Порог сходства для объединения сегментов в один кластер диктора.
  Чем выше, тем строже разделение:

Left-aligned	agglomerative	kmeans
cosine	[-10, 10]	неприменим
euclidean	[0.2, 10.0]	неприменим

- **diarization.clustering.iters** `int [200, 3000] default: 1000`
  Число итераций алгоритма кластеризации **kmeans**.
- **diarization.overlap.affect_clustering** `bool [True, False] default: True`
  Учитывать ли перекрывающиеся сегменты при кластеризации.
- **diarization.overlap.enabled** `bool [True, False] default: False`
  Включение/отключение детектирования перекрытий (когда два диктора активны одновременно).
- **diarization.overlap.threshold** `float [0.0, 1.0] default: 0.6`
  Порог для определения перекрытий.
- **diarization.pre.merge_margin** `float [0.0, 0.8] default: 0.25`
  Соединяет короткие сегменты, полученные вадом, друг с другом, применяется до кластеризации.
- **diarization.post.merge_margin** `float [0.0, 1.0] default: 0.25`
  Максимальный промежуток (в секундах) между метками одного и того же диктора для их объединения в постобработке до исключения шумовых меток.
- **diarization.post.merge_margin_again** `float [0.0, 1.0] default: 0.25`
  Дополнительное правило для повторного объединения меток дикторов, после удаления шумовых меток.
- **diarization.post.remove_average_dur** `float [0.5, 3.0] default: 1.5`
  Удаление кластеров со средней длительностью короче значения параметра.
- **diarization.post.remove_short_min_dur** `float [0.0, 0.8] default: 0.0`
  Задаёт минимальную длительность сегмента (в секундах), при которой он не будет удалён на этапе постобработки результатов диаризации.
- **diarization.post.smooth_min_dur** `float [0.1, 1.5] default: 0.75`
  Удаляет сегменты, которые короче данного значения.
  Если после удаления есть метки с одинаковым диктором, то они объединяются в одного.
- **diarization.post.rename_mapping** `string of dict[str, str] default: None`
  Словарь для сопоставления индексов дикторов с именами.
  Словарь можно передать напрямую как строку в параметр.
  Не забывайте об экранировании и кавычках!
  Валидный JSON это только двойные кавычки, снаружи всю конструкцию надо обернуть в одинарные кавычки.
  Пример: `'{"1": "Operator", "2": "Customer"}'`

1.7.3 Определение языка речи (метод /lang)

запрос:

```
data = {
  "apikey": str,        # персональный ключ
  "media" : bytes,      # аудио/видеофайл

  "vad": {
      "model": str,     # Имя модели детектора речи
  },
  "lang": {
    "model":  str,      # имя модели распознавания

    # Далее опциональные параметры:
    "threshold": float, # общий порог разделения языков, (по умолчанию 0.5)
    "langs":  list(str|tuple(str,float)), # список языков, (по умолчанию все языки)
} }
with urllib.request.urlopen(f"{HOST}/lang", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

результат:

```
[ [ str,   # Код языка в формате ISO-639-3
    float, # степень соответствия
  ], ...
]
```

1.8 Методы идентификации дикторов (метод /bio)

> ❗ На данный момент новое api не реализовано, далее идёт описание совместимого api stt3

1.8.1 Формат фрагмента обработанных данных

```
{ "speaker_id":    int,         # [опционально] индекс слепка диктора из списка speakers
  "data":          bytes,       # [опционально] бинарный слепок диктора
  "win_score":     float,       # [опционально] степень схожести, с которой выиграл эталон
  "win_etalon_id": int,         # [опционально] индекс выигравшего эталона в списке etalons
  "scores":        list(float), # [опционально] список степеней схожести по всем эталонам
}
```

1.8.2 Получение слепка диктора/эталона из аудиофайлов

запрос:

```
data = {
  "apikey":        str,         # Персональный ключ
  "model":         str,         # Имя модели биометрии
  # данные
  "wavs" :         list(bytes), # Список бинарных данных аудиофайлов
  # опционально
  "vad_model":     str,         # Имя модели обнаружения речи, (по умолчанию None)
  "async":         bool,        # Асинхронный вариант, (по умолчанию False)
  "post_callback": str,         # Пример: "127.0.0.1/callback"
}
packed_data = msgpack.packb(data, use_bin_type=True)

r = requests.post(f"{HOST}/bio", data=packed_data)
assert r.status_code == 200
result = msgpack.unpackb(r.content, raw=False)
```

результат:

При отправлении любого количества файлов возвращается бинарный слепок:

```
[ {"data": bytes} ]
```

1.8.3 Сравнение аудиофайлов со слепками эталонов

запрос:

```
data = {
  "apikey":        str,         # Персональный ключ
  "model":         str,         # Имя модели биометрии
  # данные
  "wavs" :         list(bytes), # Список бинарных данных аудиофайлов
  "etalons" :      list(bytes), # Список бинарных данных слепков эталонов
  # опционально
  "vad_model":     str,         # Имя модели обнаружения речи, (по умолчанию None)
  "async":         bool,        # Асинхронный вариант, (по умолчанию False)
  "post_callback": str,         # Пример: "127.0.0.1/callback"
}
packed_data = msgpack.packb(data, use_bin_type=True)

r = requests.post(f"{HOST}/bio", data=packed_data)
assert r.status_code == 200
result = msgpack.unpackb(r.content, raw=False)
```

результат:

Возвращается бинарный слепок диктора и результат сравнения этого слепка со всеми слепками эталонов.

пример результата: **(аудиофайлы, 3 эталона)**

```
[ {"data": bytes, "win_score": 0.86, "win_etalon_id": 2, "scores": [0.50, 0.43, 0.86]} ]
```

1.8.4 Сравнение слепков дикторов со слепками эталонов

запрос:

```
data = {
  "apikey":        str,         # Персональный ключ
  "model":         str,         # Имя модели биометрии
  # Данные
  "speakers" :     list(bytes), # Список бинарных данных слепков дикторов
  "etalons" :      list(bytes), # Список бинарных данных слепков эталонов
  # опционально
  "async":         bool,        # Асинхронный вариант, (по умолчанию False)
  "post_callback": str,         # Пример: "127.0.0.1/callback"
}
packed_data = msgpack.packb(data, use_bin_type=True)

r = requests.post(f"{HOST}/bio", data=packed_data)
assert r.status_code == 200
result = msgpack.unpackb(r.content, raw=False)
```

результат:

Возвращается список результатов сравнений слепков дикторов со всеми слепками эталонов.

пример результата: **(2 слепка дикторов, 3 слепка эталонов)**

```
[ {"speaker_id": 0, "win_score": 0.99, "win_etalon_id": 1, "scores": [0.87, 0.99, 0.32]},
  {"speaker_id": 1, "win_score": 0.68, "win_etalon_id": 0, "scores": [0.68, 0.39, 0.05]},
]
```

1.8.5 Сравнение аудиофайлов и слепков дикторов со слепками эталонов

запрос:

```
data = {
  "apikey":        str,         # Персональный ключ
  "model":         str,         # Имя модели биометрии
  # данные
  "wavs" :         list(bytes), # Список бинарных данных аудиофайлов
  "speakers" :     list(bytes), # Список бинарных данных слепков дикторов
  "etalons" :      list(bytes), # Список бинарных данных слепков эталонов
  # опционально
  "vad_model":     str,         # Имя модели обнаружения речи, (по умолчанию None)
  "async":         bool,        # Асинхронный вариант, (по умолчанию False)
  "post_callback": str,         # Пример: "127.0.0.1/callback"
}
packed_data = msgpack.packb(data, use_bin_type=True)

r = requests.post(f"{HOST}/bio", data=packed_data)
assert r.status_code == 200
result = msgpack.unpackb(r.content, raw=False)
```

результат:

Возвращается список результатов сравнений слепков дикторов со всеми слепками эталонов.
Последним в списке будет результат сравнения слепка диктора из аудиофайлов со всеми слепками эталонов.

пример результата: **(аудиофайлы, 2 слепка дикторов, 3 слепка эталонов)**

```
[ {"speaker_id": 0, "win_score": 0.99, "win_etalon_id": 1, "scores": [0.87, 0.99, 0.32]},
  {"speaker_id": 1, "win_score": 0.68, "win_etalon_id": 0, "scores": [0.68, 0.39, 0.05]},
  {"data": bytes, "win_score": 0.86, "win_etalon_id": 2, "scores": [0.50, 0.43, 0.86]},
]
```

1.8.6 Поиск дикторов в аудиофайле

запрос:

```
data = {
  "apikey":       str,   # Персональный ключ
  "model":        str,   # Имя модели биометрии
  # данные
  "wav" :         bytes, # Бинарные данные аудиофайла
  "speakers" : {         # Словарь имён и бинарных данных слепков дикторов
    "dictor_a":   bytes,
    "dictor_b":   bytes,
    ...
  },
  # задаётся один из двух параметров:
  "vad_model":     str,  # Имя модели обнаружения речи, (по умолчанию None)
  "marking":       list, # Результат распознавания /file, для выравнивания таймингов дикторов
  # опционально
  "async":         bool, # Асинхронный вариант, (по умолчанию False)
  "post_callback": str,  # Пример: "127.0.0.1/callback"
}
packed_data = msgpack.packb(data, use_bin_type=True)

r = requests.post(f"{HOST}/bio", data=packed_data)
assert r.status_code == 200
result = msgpack.unpackb(r.content, raw=False)
```

результат:

Возвращается список таймингов звучащих дикторов с именами и вероятностями.
Результатом является список списков вида:

```
[ [ int,  # начало (фреймы)
    int,  # конец (фреймы)
    str,  # имя диктора
    float # вероятность
  ], ...
]
```

пример результата:

```
[ [0, 216000, "unknown", -100.0],
  [216000, 360000, "dictor_b", 0.6084472741931677],
  [360000, 432000, "unknown", -100.0],
  [432000, 456000, "dictor_a", 0.5523388385772705],
]
```

1.9 Методы Адаптации

Для увеличения точности распознавания заданной тематики адаптируют или создают новую лингвистическую модель распознавания речи.

1.9.1 Адаптация нейросетевой лингвистической модели (метод /mas)

Позволяет:
- создать модель на большом количестве тематических данных.
- дообучить имеющуюся модель на маленьком наборе тематических данных.

Для дообучения модели следует задать параметры (иначе обучения начнётся с нуля):
`"tl": True`
`"size": 1024`

запрос:

```
data = {
  "apikey": str,         # Персональный ключ
  "mas": {
    "model": "train",
    "elm": {
      "model": str,      # Имя модели распознавания
      "name": str,       # Имя нейросетевой лингвистической модели
      "size": int,       # Размер модели (256/512/1024), для дообучения только 1024
      "files": [         # Файлы с обучением
        {
          "name": str,   # имя фйла,
          "content": str # даннные, закодированные в base64
        }, ...
      ]

      # Далее опциональные параметры:
      "tl": bool,        # Использовать ли дообучение (по умолчанию False)
      "epochs": int,     # Количество эпох, положительное значение
} } }
with urllib.request.urlopen(f"{HOST}/mas", json.dumps(data).encode()) as f:
    result = json.loads(f.read().decode())
```

результат:

```
{ "elm": "done" }
```