Сервер обработки сообщений E-message

Оглавление

Бизнес-аналитика

Границы проекта

Глоссарий

Сообщение — пересылаемый пользователями текст с дополнительной информацией: время отправки, отправитель.

Диалог — именованная совокупность пользователей, общающихся друг с другом через сообщения.

Система из предметной области

1. Бизнес-процессы

1.1.Выход из диалога

Имя процесса	**Реакция системы**
Выход пользователя из диалога	• Состоит ли указанный пользователь в диалоге? o Да, состоит: ▪ Указанный участник является последним? • Да: o Удалить диалог. • Нет: o Исключить пользователя; o Сохранить изменения. o Нет, отсутствует: ▪ Уведомить об ошибке

1.2.Отправка сообщения

Имя процесса	**Реакция системы**
Отправка сообщения пользователем	• Собрать сообщение, перевести в состояние **written;** • Сохранить сообщение; • Добавить сообщение в диалог; • Перевести сообщение в состояние **sent;** • Сохранить сообщение; • Отдать пользователю собранный дескриптор.

2. Модель предметной области

2.1.Диаграмма классов

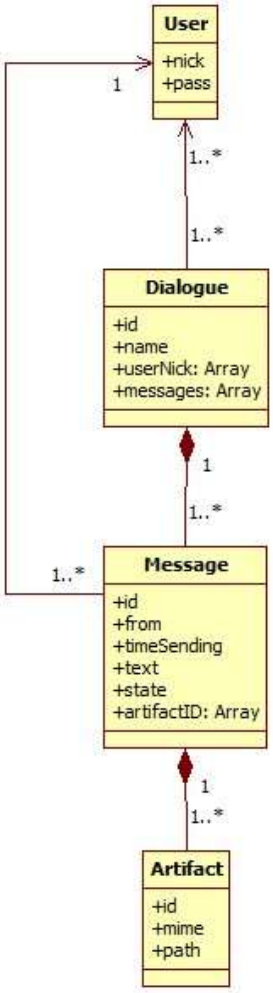

Рисунок 1 — диаграмма классов приложения

2.2.Диаграмма состояний

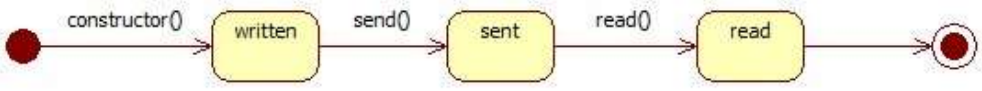

Рисунок 2 — диаграмма состояний сообщения

Системная аналитика

Прецеденты и сценарии

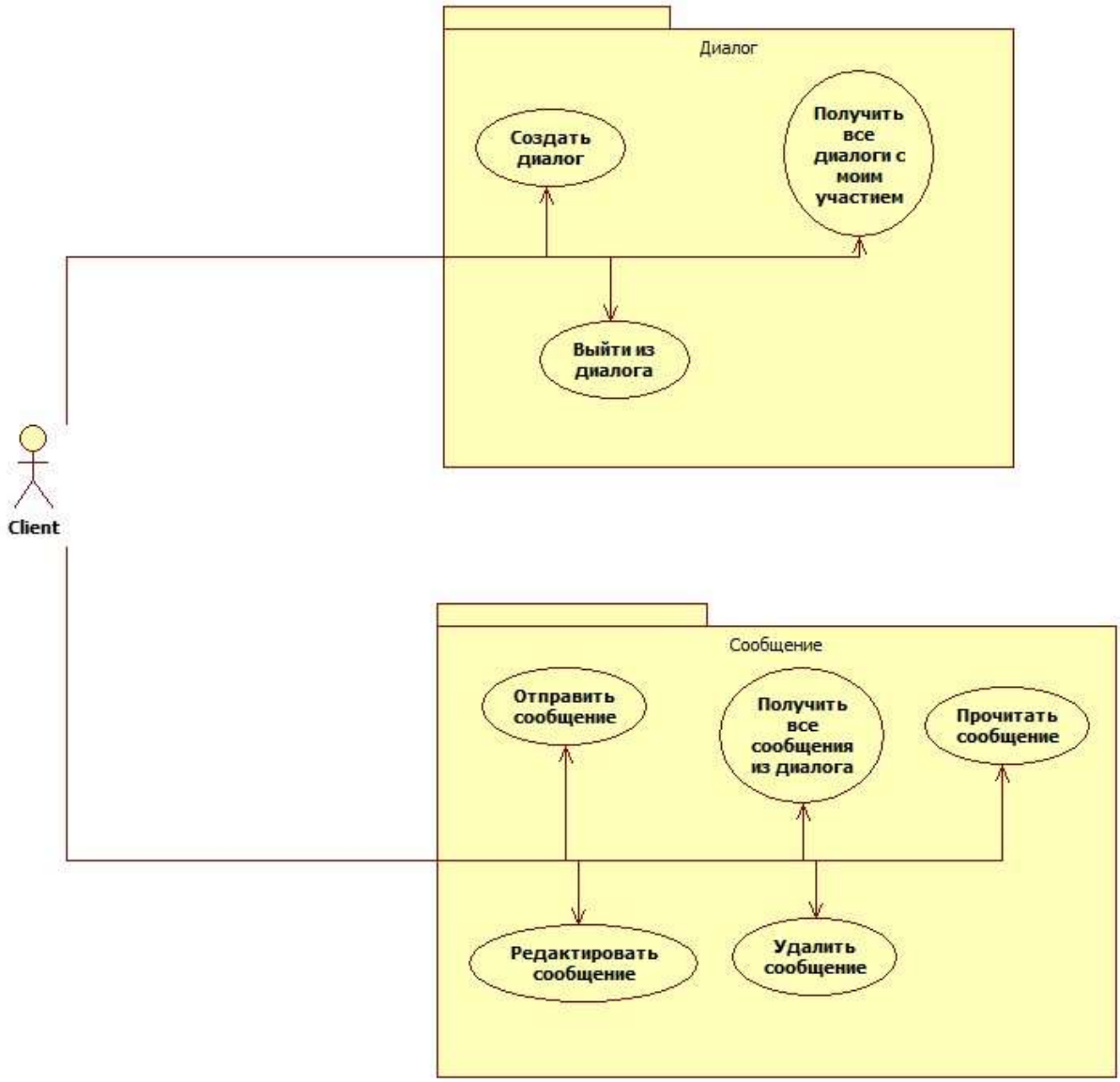

Рисунок 3 — диаграмма прецедентов приложения

Требования проекта

1. Атрибуты качества

Атрибуты.Сервер.Производительность.Многопоточность. Приложение должно эффективно распараллеливать обработку пользовательских запросов.

Атрибуты.Сервер.Производительность.Доступ к модели. Приложение должно кэшировать состояние модели в in memory database.

2. Функциональные требования

Функции.Сервер.Обслуживание_Соединений. Приложение должно слушать указанный в конфигурации порт, при этом иметь множество процессов, обслуживающих запросы клиентов.

Функция.Сервер.Процессы-обработчики. Количество процессов, обрабатывающих входящие запросы задаётся перед запуском сервера из конфигурационного файла.

Диалог.Удаление. Клиенты имеют право только выйти из диалога. Если в диалоге не осталось участников, то он автоматически удаляется со всеми содержащимися сообщениями.

Соообщение.Удаление. При удалении сообщения автоматически уничтожаются его артефакты при наличии.

Архитектура

Рисунок 4 — схематическое представление структуры системы

Система состоит из следующих компонентов:

- Клиентское приложение — отправляет по TCP-соединению текстовые запросы по формату, указанному в разделе **Интерфейс**. Разрыв соединения инициируется клиентом.
- Серверное приложение — получает запросы клиентов, обрабатывает их, отправляет результат обратно;
- СУБД — занимается сохранением изменений, внесённых в формальную модель предметной области.

Интерфейс

Интерфейс приложения построен на передаче от клиента по TCP-соединению. Структура типичного запроса:

Строка запроса
Пустая строка (разделитель)
Тело запроса

где:

- Строка запроса – строка с именем функции API. Синтаксически идентична атому Erlang;
- Пустая строка – нужна для разделения первой и последней частей. Используется UNIX-стиль разделения (\n);
- Тело запроса — часть, содержащая входные данные функции. Передаётся в формате JSON.
    - Так как разработан stateless-сервер, то практически каждый метод требует авторизации инициатора действия. Для этого передаются ник пользователя и его пароль.

1. Методы User

1.1.create_user

1.1.1. Описание

Метод позволяет сгенерировать нового пользователя в системе. Не требует аутентификации и авторизации.

1.1.2. Тело запроса

```
{
    "nick":"some_user",
    "pass":"some_pass"
}
```

1.1.3. Выход

При успешном исполнении:

```
ok
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

2. Методы Dialogue

2.1.create_dialogue

2.1.1. Описание

Выполняет генерацию нового диалога с указанным перечнем участников.

2.1.2. Тело запроса

```
{
    "nick":"creator name",
    "pass":"creator_pass",
    "name":"name of dialogue",
    "userNicks":["creator",”user2”]
}
```

2.1.3. Выход

При успешном исполнении возвращается сгенерированный объект диалога:

```
{
    "id":1,
    "name":"name_of_dialogue",
    "users":["creator",”user2”],
    "messages":null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some reason"
}
```

2.2.get_dialogues

2.2.1. Описание

Выполняет поиск всех диалогов, в которых состоит инициатор запроса.

2.2.2. Тело запроса

```
{
    "nick":"some_user",
    "pass":"some_pass"
}
```

2.2.3. Выход

При успешном исполнении возвращается массив дескрипторов диалога:

```
{
  "arr": [
    {
      "id": 1,
      "name": "name_of_dialogue",
      "users": [
        "creator",
        "user2"
      ],
      "messages": null
    },
    {
      "id": 2,
      "name": "name_of_dialogue",
      "users": [
        "creator",
        "user5"
      ],
      "messages": null
    }
  ]
}
```


При возникновении ошибки:


```
{
    "type":"error",
    "msg":"some_reason"
}
```


2.3.quit_dialogue

2.3.1. Описание

Осуществляет выход пользователя из диалога. Если диалог станет пустым, то выполнится автоматическое удаление.

2.3.2. Тело запроса


```
{
    "nick":"some_user",
    "pass":"some_pass"
}
```


2.3.3. Выход

При успешном исполнении:

```
ok
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some reason"
}
```

3. Методы Message

3.1.send_message

3.1.1. Описание

Преобразует данные сообщения в сущность системы, добавляет сформированный дескриптор в модель, создаёт ссылку в указанном диалоге на новое сообщение.

3.1.2. Тело запроса

```
{
    "nick":"some_user",
    "pass":"some_pass",
    "text":"Test 123",
    "artifactID":null,
    "dialogueID":1
}
```

3.1.3. Выход

При успешной отработке функция возвращает дескриптор построенного сообщения:

```
{
      "id": 8,
      "from": "some_user",
      "timeSending": 1659251253428,
      "text": "Test 123",
      "state": "written",
      "artifactID": null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

3.2.get_message

3.2.1. Описание

Возвращает одно сообщение по его идентификатору.

3.2.2. Тело запроса

```
{
  "nick": "some user",
  "pass": "some_pass",
  "id": 8
}
```

3.2.3. Выход

При успешной отработке функция возвращает дескриптор сообщения:

```
{
      "id": 8,
      "from": "some_user",
      "timeSending": 1659251253428,
      "text": "Test 123",
      "state": "written",
      "artifactID": null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

3.3.get_messages

3.3.1. Описание

Возвращает все сообщения из указанного диалога.

3.3.2. Тело запроса

```
{
  "nick": "some user",
  "pass": "some_pass",
  "id": 8
}
```

3.3.3. Выход

При успешной отработке сервер возвращает массив дескрипторов сообщений:

```
{
  "arr": [
    {
      "id": 8,
      "from": "vasya",
      "timeSending": 1659251253428,
      "text": "Test 123",
      "state": "written",
      "artifactID": null
    },
    {
      "id": 9,
      "from": "vasya",
      "timeSending": 1659251484494,
      "text": "Test 123",
      "state": "written",
      "artifactID": null
    }
  ]
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

3.4.read_message

3.4.1. Описание

Помечает указанное сообщение как прочитанное.

3.4.2. Тело запроса

```
{
  "nick": "some user",
  "pass": "some_pass",
  "id": 8
}
```

3.4.3. Выход

При успешном выполнении возвращается обновлённый дескриптор сообщения:

```
{
	"id": 8,
	"from": "some_user",
	"timeSending": 1659251253428,
	"text": "Test 123",
	"state": "read",
	"artifactID": null
}
```

При возникновении ошибки:

```
{
    "type":"error",
    "msg":"some_reason"
}
```

3.5.change_text

3.5.1. Описание

Позволяет редактировать текст указанного сообщения.

3.5.2. Тело запроса

```
{
  "nick": "some_user",
  "pass": "some pass",
  "id": 8,
  "text":"Napisal normal'noe soobsheniye."
}
```

3.5.3. Выход

При успешном выполнении возвращается обновлённый дескриптор сообщения:

```
{
	"id": 8,
	"from": "some_user",
	"timeSending": 1659251253428,
	"text": "Napisal normal'noe soobsheniye.",
	"state": "read",
	"artifactID": null
}
```


При возникновении ошибки:


```
{
    "type":"error",
    "msg":"some reason"
}
```


3.6.delete_message

3.6.1. Описание

Удаляет из системы указанное сообщение. Действие допустимо только для автора.

3.6.2. Тело запроса


```
{
  "nick": "some_user",
  "pass": "some pass",
  "messageID": 8,
  "dialogueID": 1
}
```


3.6.3. Выход

При успешном исполнении:

```
ok
```

При возникновении ошибки:


```
{
    "type":"error",
    "msg":"some reason"
}
```

Проектирование Серверного приложения

1. Архитектура

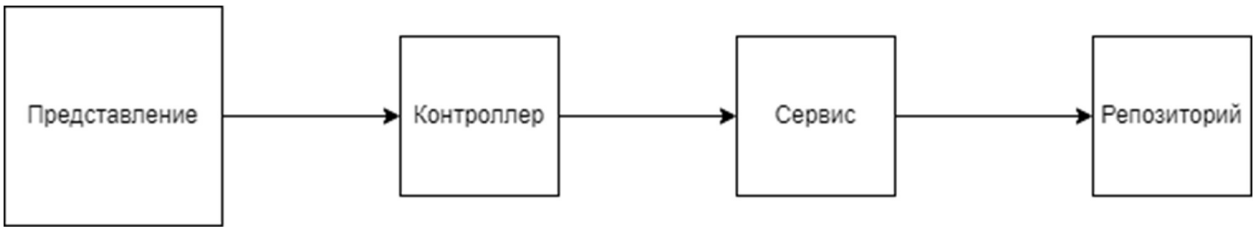

Рисунок 5 — архитектура кода серверного приложения

Приложение состоит из слоёв:

- Представление — слой, отвечающий за общение с клиентами, сериализацию сущностей модели в человекочитаемое представление;
- Контроллер — предоставляет интерфейс воздействия на модель слою представления. Отвечает за сложную логику обработки сущностей — каждая функция может вызвать одну или несколько команд слоя сервисов как для декларированной, так и для других сущностей.
- Сервис — модули данного слоя предоставляют перечень атомарных и не связанных друг с другом функций, направленных на изменение состояния одной декларированной сущности;
- Репозиторий — функции данного слоя занимаются чтением/записью сущностей модели в постоянную память и кэш.

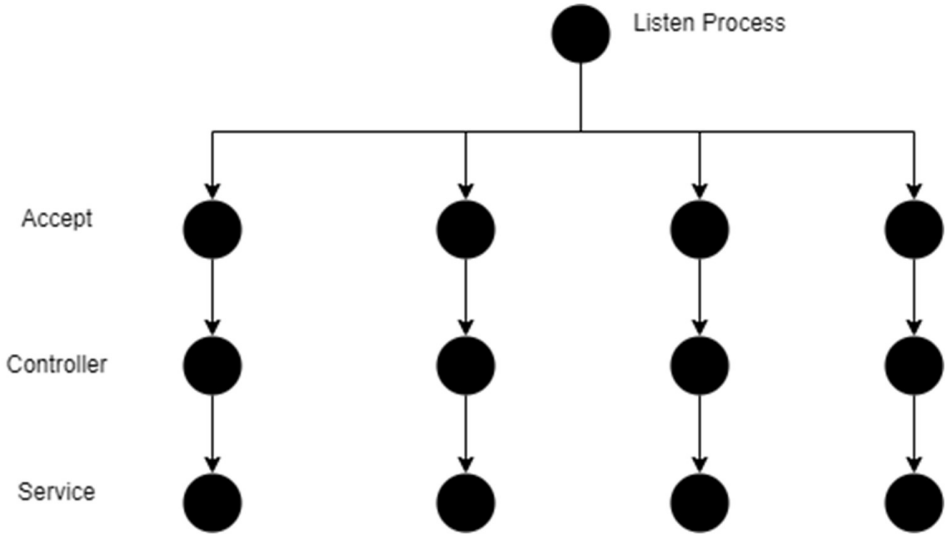

Рисунок 6 — граф управляющих и информационных связей приложения

Процесс, слушающий сокет, выполняет инициализацию приложения, создаёт N принимающих процессов, где N – число, прочитанное из конфигурационного файла.

Каждый принимающий процесс занимается обслуживанием клиентских запросов — вызов функции контроллера, соответствующей поступившему сообщению.

2. Разработка алгоритмов

2.1. Алгоритм сериализации массива записей

При разработке возникла потребность преобразования списка записей Erlang в JSON-массив. Имеющийся инструментарий мог заниматься сериализацией списков примитивных данных.

Была написана функция с хвостовой рекурсией. Алгоритм:

- Рекурсивный обход исходной последовательности:
  - Сколько элементов в списке:
    - Несколько:
      - Расщепить на голову и хвост;
      - Сформировать список результат по принципу: к имеющемуся списку добавить новую голову — голова исходного массива + ”,”.
    - Один:
      - Преобразовать последний элемент, добавить его в качестве головы результата;
      - Выполнить reverse.
- Добавить в начало и конец спец. символы для JSON.

Ниже прикреплена реализация алгоритма:

```
%%Конвертирует массив Erlang-записей в JSON-массив объектов
%%Data - массив записей
%%Encode - callback-функция, конвертирующая каждую запись из списка
%%Encode(type:atom, Rec), где: type - имя записи, Rec - запись
encodeRecordArray(Data,Encode)->
  JSON_List=encodeRecordArray(Data,[],Encode),
  ["{\"arr:\"[" | [JSON_List|"]}"]].

encodeRecordArray([H],Res,Encode)->
  JSON=Encode(H),
  lists:reverse([JSON|Res]);
encodeRecordArray([H|T],Res, Encode)->
  JSON= Encode(H),
  Put=[JSON|","],
  encodeRecordArray(T,[Put|Res], Encode).
```

Так как написанная функция является функцией высшего порядка за счёт получения на вход callback-функции, занимающейся преобразованием каждой записи входной последовательности, то итоговая реализация является обобщённой.

3. Кодирование и отладка

3.1.Слой представления

//рассказать про организацию обработки запросов и построения процессов

В Erlang при работе с TCP-соединениями существует логическое разделение процеесов на:

- Слушающий процесс — вызывает функцию `gen_tcp:listen()`. Занимается назначением обработчиков;
- Принимающий процесс — вызывает функцию `gen_tcp:accept()`. После может получать данные от клиента, выполнять их обработку, посылать ответы.

Одному слушающему процессу может соответствовать множество принимающих. За счёт этой возможности осуществляется параллельная обработка клиентских соединений:

- В процессе инициализации приложение запрашивает ListenSocket;
- Если всё прошло успешно, то генерируется N принимающих процессов, где N – параметр, прочтённый из конфигурации;
- Каждый принимающий процесс работает в цикле:
  Ожидание запроса-> Обработка данных запроса-> Отправка ответа -> Ожидение запроса…
  Цикл прерывается при прерывании соединения клиентом.

Ниже прикреплён фрагмент кода, демонстрирующий скелет слоя работы с соединениями:

```
start() ->
  db:start db(),
  {ok,Text_Bin}=file:read_file("priv/etc/config.json"),
  Conf=?json to record(config,Text Bin),
  #config{port = Port,acceptors_quantity = N}=Conf,
  case  gen_tcp:listen(Port,[{active, false}]) of
    {ok, ListenSocket}->
      io:format("INFO server:start/0 Server started. Port=~w~n",[Port]),
      start_servers(N,ListenSocket),
      io:format("INFO server:start/0 Started ~w acceptors~n",[N]),
      %%замораживает listen-процесс
      timer:sleep(infinity);
    {error, Reason}->
      io:format("FATAL Can't listen port.~n~p~n",[Reason])
  end.

start_servers(0,_)-> ok;
start_servers(Num, ListenSocket)->
  spawn(?MODULE,wait_request,[ListenSocket]),
  io:format("INFO server:start_servers/2 Acceptor#~w spawned~n",[Num]),
  start servers(Num-1,ListenSocket).

wait_request(ListenSocket)->
  case gen_tcp:accept(ListenSocket) of
```

```
    {ok, Socket} ->
      loop(Socket),
      wait_request(ListenSocket);
    {error, Reason}->
      io:format("ERROR server:wait_request Socket ~w [~w] can't accept session.
Reason:~p~n",[ListenSocket, self(),Reason])
  end.

%%функция-цикл работы потока-акцептора
loop(Socket)->
  inet:setopts(Socket,[{active,once}]),
  receive
    {tcp,Socket,Request}->
      io:format("INFO server:loop/1 Socket ~w [~w] received request ~n",
[Socket, self()]),
      process_request(Socket,Request),
      loop(Socket);
    {tcp_closed,Socket}->
      io:format("INFO server:loop/1 Socket ~w closed [~w]~n",[Socket,self()]),
      ok
  end.

%%обработка клиентских запросов
process_request(Socket, Request)->
  [Fun,ArgsJSON]=parseRequest(Request),
  case Fun of
    create_user->
      create_user_handler(ArgsJSON,Socket);
    create_dialogue->
      create_dialogue_handler(ArgsJSON,Socket);
    get_dialogues->
      get_dialogues_handler(ArgsJSON,Socket);
    quit_dialogue->
      quit_dialogue_handler(ArgsJSON,Socket);
    send_message->
      send_message_handler(ArgsJSON,Socket);
    get_message->
      get_message_handler(ArgsJSON,Socket);
    get_messages->
      get_messages_handler(ArgsJSON,Socket);
    read_message->
      read_message_handler(ArgsJSON,Socket);
    change_text->
      change_text_handler(ArgsJSON,Socket);
    delete_message->
      delete_message_handler(ArgsJSON,Socket)
  end.
```

3.2.Слой сервисов

В данном слое выполняется логика обработки сущностей, а также функции обобщённой обработки результатов транзакций репозитория.

Пример обобщённых функций, обрабатывающих результаты транзакций чтения записей:

```
%%Обобщенная функция, обрабатывающая результат транзакции,
%%возвращающей результат mnesia:read.
%%Но с точки зрения бизнес-модели результат обязан быть единственным.
extract_single_value(Transaction)->
  case Transaction of
    {error, R}->{error, R};
    []->{error,not_found};
    [Res|_]->Res
  end.

%%аналог предыдущей, но предназначена для
%%чтений с несколькими результатами
extract_values(Transaction)->
  case Transaction of
    {error,_Reason}->{error,_Reason};
    []->{error,not found};
    Res->Res
  end.
```

Пример функции, обрабатывающий отправку сообщения в диалог:

```
%%Сохранить в БД сообщение
%%Добавить полученный ID в диалог
%%Сохранить диалог
%%Вовзращает персистентное сообщение
add_message(D,M)->
  Fun=
    fun()->
      M_Persisted = message_repo:write(M),
      D_Updated = dialogue:add_message(D,M_Persisted),
      dialogue_repo:update(D_Updated),
      message_repo:update(message:send(M_Persisted)),
      message_repo:read(M_Persisted#message.id)
    end,
  T = transaction:begin_transaction(Fun),
  service:extract_single_value(T).
```

3.3.Слой репозитория

Репозиторий – абстракция для получения данных. Слой предоставляет два интерфейса сервисам:

- Интерфейс CRUD-операций в БД для сущностей;
- Интерфейс транзакций для CRUD-операций.

Такой подход позволяет сервисам в рамках одной транзакции выполнять несколько запросов на чтение/запись к СУБД, что положительно сказывается на быстродействии и надёжности приложения.

Слой репозитория по совместительству является обёрткой над СУБД Mnesia, тому послужило несколько причин:

- Mnesia позволяет легко строить распределённую на разных физических узлах базу;

- Mnesia совмещает в себе функционал in memory db и базы, хранящей информацию на внешнем диске;
- Mnesia является встроенным приложением в Erlang\OTP.

Пример интерфейса транзакций:

```
%% API
-export([begin_transaction/1, abort_transaction/1]).

begin_transaction(Fun)->
  case mnesia:transaction(Fun) of
    {atomic,  Res}-> Res;
    {aborted, _Reason}-> {error,_Reason}
  end.

abort_transaction(Reason)->
  mnesia:abort(Reason).
```

Пример функций репозитория сообщений:

```
write(Message)->
  ID = seq:get counter(seq),
  Commited=Message#message{id=ID},
  mnesia:write(Commited),
  [Obj| ]=mnesia:read(message,ID),
  Obj.

read(ID)->
  mnesia:read(message,ID).

update(Message)->
  mnesia:write(Message).

%%Каскадно удаляются артефакты, так как вне сообщений они не имеют смысла
delete(#message{id=MID})->
  mnesia:delete({message, MID}).
```