Сервер обработки сообщений E-message

Оглавление

[Бизнес-аналитика 3](#_Toc110249293)

[Границы проекта 3](#_Toc110249294)

[Глоссарий 3](#_Toc110249295)

[Система из предметной области 3](#_Toc110249296)

[1. Бизнес-процессы 3](#_Toc110249297)

[2. Модель предметной области 4](#_Toc110249298)

[Системная аналитика 5](#_Toc110249299)

[Прецеденты и сценарии 5](#_Toc110249300)

[Требования проекта 6](#_Toc110249301)

[1. Атрибуты качества 6](#_Toc110249302)

[2. Функциональные требования 6](#_Toc110249303)

[Архитектура 7](#_Toc110249304)

[Интерфейс 7](#_Toc110249305)

[1. Методы User 8](#_Toc110249306)

[2. Методы Dialogue 8](#_Toc110249307)

[3. Методы Message 11](#_Toc110249308)

[Проектирование Серверного приложения 16](#_Toc110249309)

[1. Архитектура 16](#_Toc110249310)

[2. Разработка алгоритмов 17](#_Toc110249311)

[3. Кодирование и отладка 18](#_Toc110249312)

# 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

### Бизнес-процессы

#### Выход из диалога

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

#### 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. |

### Модель предметной области

#### Диаграмма классов

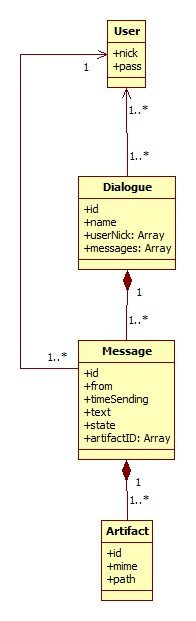


Рисунок 1 — диаграмма классов приложения

#### Диаграмма состояний

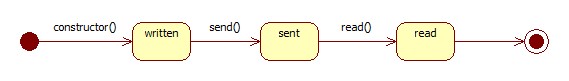


Рисунок 2 — диаграмма состояний сообщения

# Системная аналитика

## Прецеденты и сценарии

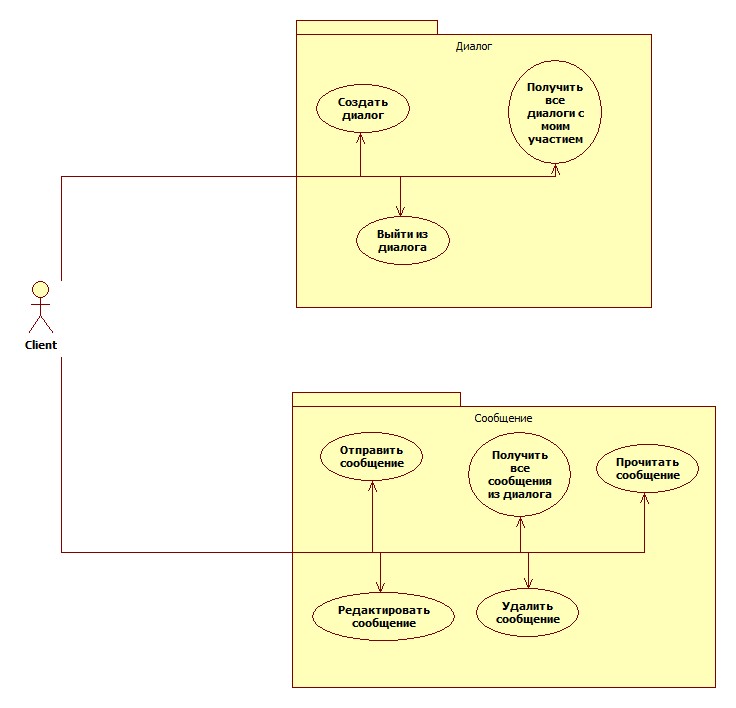


Рисунок 3 — диаграмма прецедентов приложения

## Требования проекта

### 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.

### 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

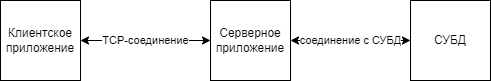


Рисунок 4 — схематическое представление структуры системы

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.

### Методы User

#### create\_user

##### Описание

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

##### Тело запроса

|  |
| --- |
| {  "nick":"some\_user",  "pass":"some\_pass"  } |

##### Выход

При успешном исполнении:

|  |
| --- |
| ok |

При возникновении ошибки:

|  |
| --- |
| {  "type":"error",  "msg":"some\_reason"  } |

### Методы Dialogue

#### create\_dialogue

##### Описание

Выполняет генерацию нового диалога с указанным перечнем участников.

##### Тело запроса

|  |
| --- |
| {  "nick":"creator\_name",  "pass":"creator\_pass",  "name":"name\_of\_dialogue",  "userNicks":["creator",”user2”]  } |

##### 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"  } |

#### get\_dialogues

##### Описание

Выполняет поиск всех диалогов, в которых состоит инициатор запроса.

##### Тело запроса

|  |
| --- |
| {  "nick":"some\_user",  "pass":"some\_pass"  } |

##### 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"  } |

#### quit\_dialogue

##### Описание

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

##### Тело запроса

|  |
| --- |
| {  "nick":"some\_user",  "pass":"some\_pass"  } |

##### Выход

При успешном исполнении:

|  |
| --- |
| ok |

При возникновении ошибки:

|  |
| --- |
| {  "type":"error",  "msg":"some\_reason"  } |

### Методы Message

#### send\_message

##### Описание

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

##### Тело запроса

|  |
| --- |
| {  "nick":"some\_user",  "pass":"some\_pass",  "text":"Test 123",  "artifactID":**null**,  "dialogueID":1  } |

##### 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"  } |

#### get\_message

##### Описание

Возвращает одно сообщение по его идентификатору.

##### Тело запроса

|  |
| --- |
| {  "nick": "some\_user",  "pass": "some\_pass",  "id": 8  } |

##### 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"  } |

#### get\_messages

##### Описание

Возвращает все сообщения из указанного диалога.

##### Тело запроса

|  |
| --- |
| {  "nick": "some\_user",  "pass": "some\_pass",  "id": 8  } |

##### 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"  } |

#### read\_message

##### Описание

Помечает указанное сообщение как прочитанное.

##### Тело запроса

|  |
| --- |
| {  "nick": "some\_user",  "pass": "some\_pass",  "id": 8  } |

##### 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"  } |

#### change\_text

##### Описание

Позволяет редактировать текст указанного сообщения.

##### Тело запроса

|  |
| --- |
| {  "nick": "some\_user",  "pass": "some\_pass",  "id": 8,  "text":"Napisal normal'noe soobsheniye."  } |

##### 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"  } |

#### delete\_message

##### Описание

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

##### Тело запроса

|  |
| --- |
| {  "nick": "some\_user",  "pass": "some\_pass",  "messageID": 8,  "dialogueID": 1  } |

##### Выход

При успешном исполнении:

|  |
| --- |
| ok |

При возникновении ошибки:

|  |
| --- |
| {  "type":"error",  "msg":"some\_reason"  } |

# Проектирование Серверного приложения

### Архитектура

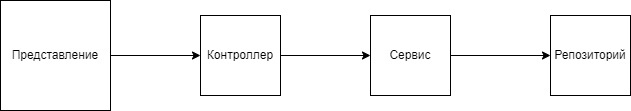


Рисунок 5 — архитектура кода серверного приложения

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.

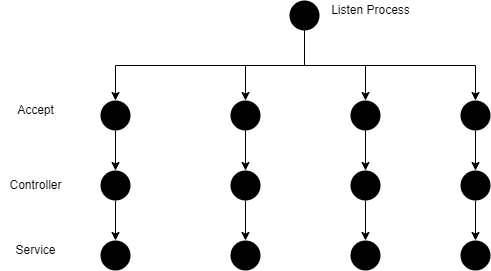


Рисунок 6 — граф управляющих и информационных связей приложения

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.

### Модель данных предметной области

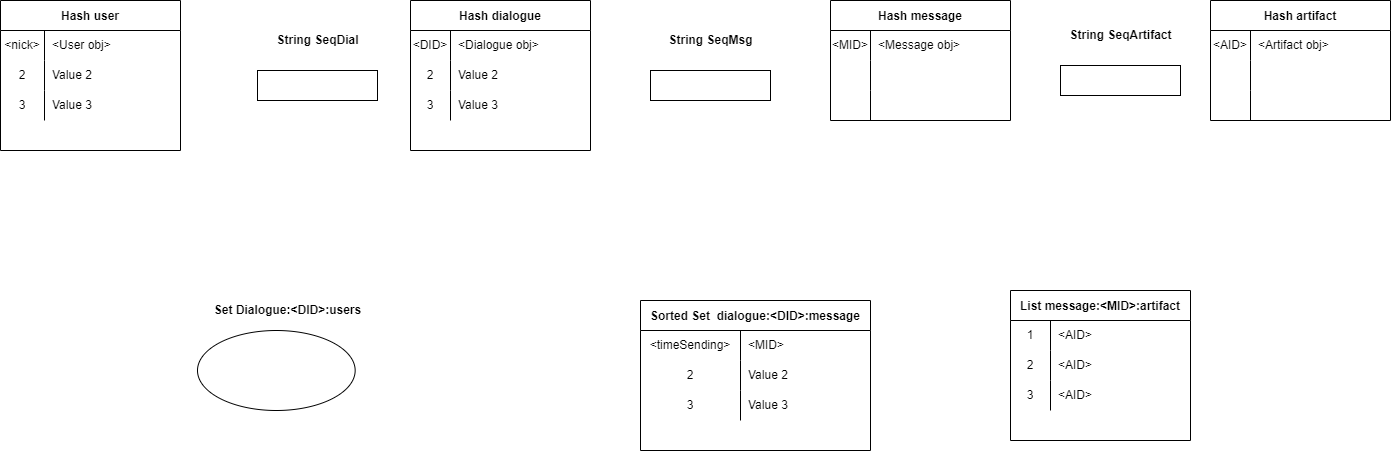


Рисунок 7 — модель данных приложения в системе Redis

Для связи сущностей Диалог и Пользователь было выбрано множество пользователей для каждого диалога.

Такое решение обеспечивает экономию памяти, но взамен:

* Операция поиска всех диалогов для одного пользователя будет иметь сложность O(n) (n - число всех диалогов), так как нужно перебирать список пользователей для каждого диалога;
* Сложность поиска всех участников диалога будет соответствовать O(n1) (n1 – мощность множества пользователей диалога). В данном случае n1 будет в разы меньше n.

Прочие доступные решения:

* Сделать множество диалогов для каждого пользователя:
  + Поиск всех диалогов пользователя = O(n), где n – мощность множества диалогов;
  + Поиск всех участников диалога = O(n1), где n1 – количество всех пользователей.
* Сделать два множества: множество участников диалога, множество диалогов пользователя:
  + Поиск всех диалогов пользователя = O(n), где n – мощность множества диалогов;
  + Поиск всех участников диалога = O(n1), где n1 – мощность множества участников.
  + Возникает необходимость синхронизации обоих коллекций;
  + Возрастают затраты памяти на поддержку состояния модели.

### Разработка алгоритмов

#### Перезапись диалога в базу

Есть два способа обновления дерева сообщений для одного диалога:

* получить коллекцию сообщений, удалить дерево, записать новое  
  где n – количество записей дерева
* проверить каждый ключ,если нет, то получить дескриптор сообщения и добавить ключ в дерево   
  где n – количество записей дерева, x – количество элементов, которых нет в базе, но они присутствуют в памяти сервера

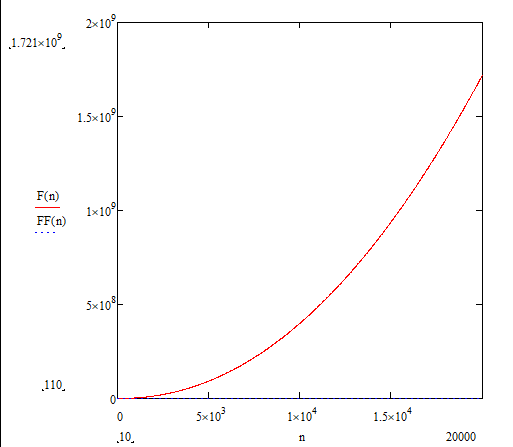


Рисунок — график функций трудоёмкости: F(n) – первый способ, FF(n) – второй способ (x = 20000).

#### 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.

### Кодирование и отладка

#### 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**. |

#### 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). |

#### 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}). |