Сервер обработки сообщений E-message

Оглавление

[Бизнес-аналитика 3](#_Toc111025743)

[Границы проекта 3](#_Toc111025744)

[Глоссарий 3](#_Toc111025745)

[Система из предметной области 3](#_Toc111025746)

[1. Бизнес-процессы 3](#_Toc111025747)

[2. Модель предметной области 4](#_Toc111025748)

[Системная аналитика 5](#_Toc111025749)

[Прецеденты и сценарии 5](#_Toc111025750)

[Требования проекта 6](#_Toc111025751)

[1. Атрибуты качества 6](#_Toc111025752)

[2. Функциональные требования 6](#_Toc111025753)

[Архитектура 7](#_Toc111025754)

[Интерфейс системы 7](#_Toc111025755)

[1. Методы User 8](#_Toc111025756)

[2. Методы Dialogue 8](#_Toc111025757)

[3. Методы Message 11](#_Toc111025758)

[Проектирование Серверного приложения 17](#_Toc111025759)

[1. Архитектура 17](#_Toc111025760)

[2. Модель данных предметной области 19](#_Toc111025761)

[3. Разработка алгоритмов 20](#_Toc111025762)

[4. Кодирование и отладка 21](#_Toc111025763)

# Бизнес-аналитика

## Границы проекта

* Отсутствует обработка дополнительных вложений в сообщение;
* Состав участников диалога утверждается один раз. После создания конференции из неё можно будет только выйти;
* Не предоставляется готовый клиент;
* Предоставляемые соединения не защищаются — данные передаются открытым текстом.

## 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

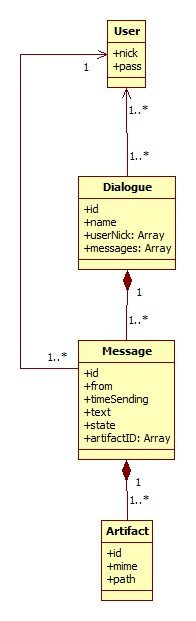


Рисунок — диаграмма классов приложения

#### Диаграмма состояний

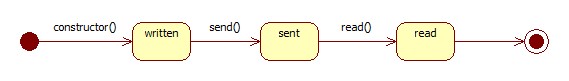


Рисунок — диаграмма состояний сообщения

# Системная аналитика

## Прецеденты и сценарии

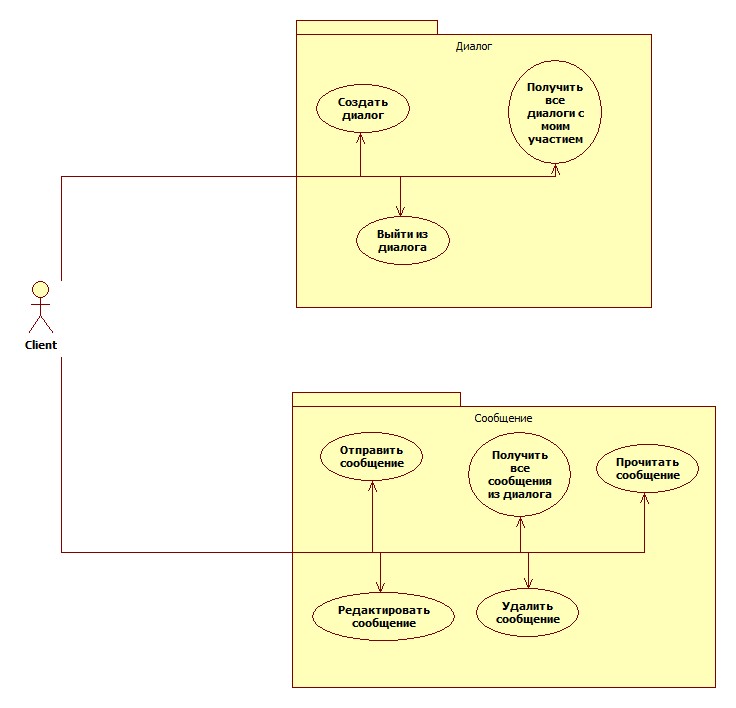


Рисунок — диаграмма прецедентов приложения

## 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

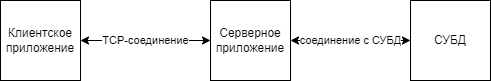


Рисунок — схематическое представление структуры системы

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 системы

Интерфейс приложения построен на передаче текстовой информации от клиента по TCP-соединению. Структура любого 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",  "id": 8  } |

##### Выход

При успешном исполнении может быть два варианта:

* Пользователь был последним в диалоге, тогда система удалит его полностью и вернёт:

|  |
| --- |
| ok |

* Пользователь не являлся единственным участником, тогда система вернёт обновлённую сущность диалога:

|  |
| --- |
| {  "id": 1,  "name": "name\_of\_dialogue",  "users": [  "creator",  ],  "messages": **null**  } |

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

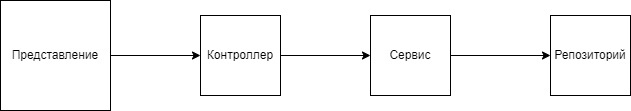


Рисунок — архитектура кода серверного приложения

Приложение состоит из компонентов:

* 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.

Предоставляемый слоями интерфейс не зависит от его внутренней реализации. Ниже прикреплены диаграммы, демонстрирующие принцип распараллеливания приложения:

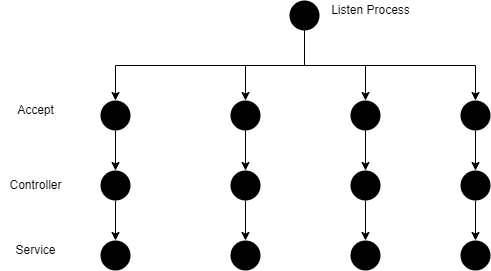


Рисунок — граф управляющих и информационных связей приложения. Показывает расщепление алгоритма работы программы на крупные блоки.

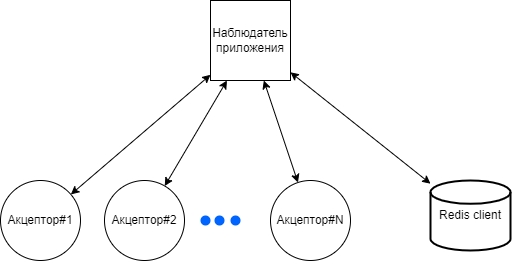


Рисунок — схематическое представление процессов системы. Демонстрирует Erlang-процессы из которых состоит приложение.

Типы процессов:

1. Наблюдатель приложения — занимается инициализацией дочерних процессов, прослушиванием входящих TCP-соединений, распределением их обработки между другими акцепторами;
2. Акцептор — процесс, обслуживающий клиентское соединение. Обслуживание происходит по принципу: «получить запрос, сформировать ответ, отправить ответ». Соединения закрываются по инициативе клиента;
3. Redis client — сторонний процесс Erlang, работа с которым осуществляется акцепторами по паттерну клиент-сервер. Через него происходят все операции с СУБД Redis.

### Модель данных предметной области

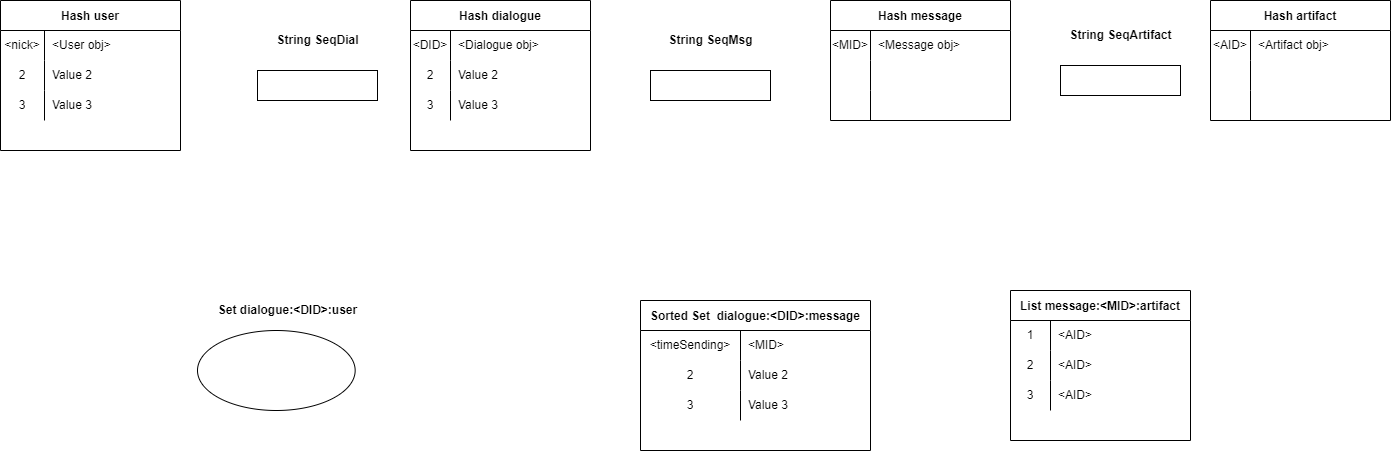


Рисунок — модель данных приложения в системе Redis

Строки с ключами формата Seq<Entity> содержат счётчик, значения которого берутся для формирования нового идентификатора конкретной сущности.

Для хранения сообщений диалога была выбрана структура Sorted Set так как она позволяет извлекать сообщения в порядке их получения сервером без дополнительных затрат на сортировку.

Для связи сущностей Диалог и Пользователь было выбрано множество пользователей для каждого диалога. Структура «Set» удобна, так как позволяет проверять наличие пользователя в диалоге без лишних итераций. 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 пользователя;
  + Поиск всех участников диалога = O(n1), где n1 – мощность множества участников одного диалога.
  + Возникает необходимость синхронизации обоих коллекций;
  + Возрастают затраты памяти на поддержку состояния модели.

### Разработка алгоритмов

#### 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

#### Процессы

##### Наблюдатель приложения

Наблюдатель предоставляет внешней Erlang среде следующий интерфейс:

1. Старт:
   1. Описание:
   2. Тело запроса:
      1. Функция  
         e\_message:start(Path), где:  
         Path - путь до файла config.json;
      2. Функция   
         e\_message:start().   
         Путь по умолчанию "priv/etc/config.json".
   3. Выход:
      1. Функция возвращает атом ok.  
         Если во время инициализации приложения произойдёт критическая ошибка, то сервер отправит наверх сигналы выхода:
         1. {‘EXIT’, PID, acceptors\_start\_fail} — ни один из акцепторов не запустился;
         2. {‘EXIT’, PID, repo\_start\_fail} — клиент БД не запустился.
2. Стоп
   1. Описание.
      1. Наблюдатель сначала начнёт остановку каждого акцептора, если он не ответит в течение 5 секунд, то будет остановлен принудительно.  
         После отключается клиент Redis.
   2. Тело запроса:
      1. Кортеж Erlang:   
         {stop,From},   
         где From - PID отправителя.
   3. Выход:
      1. Сообщение ok.

##### Акцептор

Акцептор кроме API системы предоставляет внутренний интерфейс:

* Стоп
  1. Описание:
     1. Процесс постарается завершиться в момент, когда он не занят обслуживанием клиента.
  2. Тело запроса:
     1. Сообщенине  
        {stop,From},  
        где From - PID отправителя.
  3. Выход:
     1. Сообщение ok.

Акцептор в процессе своей работы вызывает коды слоёв **представления**, **контроллеров**, **сервисов**, **репозиториев**.

#### Слои приложения

Так как каждое действие слоя сводится к CRUD-операции интерфейс был построен по следующим принципам:

* На вход принимаются:
  + Объект;
  + Операция чтения — ключи поиска.
* Выходы:
  + Create-операции должны возвращать 1 персистентный объект;
  + Read-операции - фильтры, поэтому они возвращают пустой или заполненный список.  
    Если ничего не найдено, то пусть пользователь интерфейса решает является ли ситуация исключительной. Пустое множество результатов означает, что пользователь отправил плохие параметры поиска.
  + Update-операции - возвращают новый объект;
  + Delete – ok;

##### Слой представления

Слой представления состоит из блока receive, в котором для каждого метода API системы назначается нужная функция-обработчик. Код назначения обработчика:

|  |
| --- |
| %%обработка клиентских запросов  serve\_request(Socket, Request,Con)->  [Fun,ArgsJSON]=parseRequest(Request),  **case** Fun **of**  create\_user->  create\_user\_handler(ArgsJSON,Socket, Con);  create\_dialogue->  create\_dialogue\_handler(ArgsJSON,Socket, Con);  get\_dialogues->  get\_dialogues\_handler(ArgsJSON,Socket, Con);  quit\_dialogue->  quit\_dialogue\_handler(ArgsJSON,Socket, Con);  send\_message->  send\_message\_handler(ArgsJSON,Socket, Con);  get\_message->  get\_message\_handler(ArgsJSON,Socket, Con);  get\_messages->  get\_messages\_handler(ArgsJSON,Socket, Con);  read\_message->  read\_message\_handler(ArgsJSON,Socket, Con);  change\_text->  change\_text\_handler(ArgsJSON,Socket, Con);  delete\_message->  delete\_message\_handler(ArgsJSON,Socket, Con)  **end**. |

Пример функции обработчика:

|  |
| --- |
| get\_messages\_handler(ArgsJSON, Socket, Con)->  Args = ?json\_to\_record(get\_messages,ArgsJSON),  **#get\_messages**{nick = Nick, pass=Pass, id = DID}=Args,  **case** is\_authorised(Nick,Pass,Socket, Con) **of**  true->  D=dialogue\_controller:get\_dialogue(DID, Con),  io:format("TRACE server:get\_messages\_handler/3 D:~p~n",[D]),  **case** D **of**  {error,\_R}->  handle\_error(\_R,Socket);  D->  Res = dialogue\_controller:get\_messages(D, Con),  io:format("TRACE server:get\_messages\_handler/3 Messages:~p~n",[Res]),  handle\_request\_result(  Res,  **fun**(Y)-> parse:encodeRecordArray(Y,**fun**(X)->?record\_to\_json(message,X) **end**) **end**,  Socket)  **end**;  false->ok  **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:

|  |
| --- |
| %%Функция обработки read-операции репозитория.  %%Необходима, если с точки зрения бизнес-процесса результат:  %% 1) обязан быть найден  %% 2) должен быть единственным  extract\_single\_value(Transaction)->  **case** Transaction **of**  {error,\_R}->{error,\_R};  []->{error,not\_found};  [Res|\_]->Res  **end**.  %%Функция обработки read-операции репозитория.  %%Необходима, если с точки зрения бизнес-процесса результат:  %% 1) обязан быть найден  %% 2) допускается несколько объектов  extract\_multiple\_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, являющихся метафорой на действия с коллекцией данных;
* 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.

Модули репозитория поделены на 2 части:

* Код конкретного репозитория к Redis — функции, получающие данные от клиента Redis;
* Код абстрактного репозитория — обёртка над конкретным репозиторием.

Пример абстрактного репозитория сообщений:

|  |
| --- |
| -module(message\_repo).  -include("entity.hrl").  -export([read/2,  write/2,  update/2,  delete/2]).  %%create-операции должны возвращать 1 персистентный объект  %%read-операции - фильтры, поэтому они возвращают пустой или заполненный список  %%update-операции - возвращают новый объект  %%delete - ok  %%ошибка - {error, Reason}  %%Если произошла ошибка соединения с источником -> значит, что вся система становится бесполезной->  %%ошибка критическая и должна быть послана на самый верх  write(Message, Con)->  redis\_message:write(Con,Message).  read(ID, Con)->  redis\_message:read(Con,ID).  update(Message, Con)->  redis\_message:update(Con,Message).  %%Каскадно удаляются артефакты, так как вне сообщений они не имеют смысла  delete(**#message**{}=Message, Con)->  redis\_message:delete(Con,Message). |

Пример конкретного репозитория сообщений

|  |
| --- |
| -module(redis\_message).  -include("entity.hrl").  -include("jsonerl/jsonerl.hrl").  -export([write/2,  read/2,  update/2,  delete/2]).  %% на вход всегда принимаются:  %% дескриптор соединения, сущность  %% дескриптор соединения, ключи поиска  %%create-операции должны возвращать 1 персистентный объект  %%read-операции - фильтры, поэтому они возвращают пустой или заполненный список  %%update-операции - возвращают новый объект  %%delete - ok  write(Con,**#message**{}=Message)->  {ok,MID}=eredis:q(Con,["INCR","SeqMsg"]),  Commited = Message**#message**{id = binary\_to\_integer(MID)},  {ok,\_}=eredis:q(Con,["HSET",**atom\_to\_list**(message),MID,?record\_to\_json(message,Commited)]),  Commited.  read(Con,MID) **when** MID =/= -1 ->  {ok,JSON} = eredis:q(Con,["HGET",**atom\_to\_list**(message),MID]),  [?json\_to\_record(message,JSON)].  update(Con,**#message**{id = MID}=Message)->  {ok,\_}=eredis:q(Con,["HSET",**atom\_to\_list**(message),MID,?record\_to\_json(message,Message)]),  Message.  delete(Con,**#message**{id=MID})->  {ok,\_} = eredis:q(Con,["HDEL",**atom\_to\_list**(message),MID]),  ok. |

Пример абстрактного интерфейса транзакций:

|  |
| --- |
| -module(transaction).  -export([begin\_transaction/1,  abort\_transaction/0]).  %%Res || {error, Cause}  %%Fun/0 функция с телом транзакции.  %%Тело транзакции - набор действий языка среди которых есть операции, предоставляемые репозиторием.  %%Каждая функция репозитория - обращение к базе  %%Следовательно, заворачивая множество функций репозитория в транзакции мы заворачиваем и обращения к базе.  %%Транзакция исполняется если её не отменили.  begin\_transaction(Fun)->  **try**  redis\_transaction:begin\_transaction(Fun)  **catch**  throw:transaction\_aborted -> {error, transaction\_aborted}  **end**.  %%функция внутри должна послать сообщение базе об отмене транзакции и  %%выбросить исключение transaction\_aborted  abort\_transaction()->  redis\_transaction:abort\_transaction(). |

Пример интерфейса транзакций Redis:

|  |
| --- |
| -module(redis\_transaction).  -author("aleksandr\_work").  -export([begin\_transaction/1,  abort\_transaction/0]).  begin\_transaction(Fun)->  **case** Fun() **of**  {error,\_R}->{error,\_R};  Res -> Res  **end**.  abort\_transaction()->  **throw**(transaction\_aborted). |