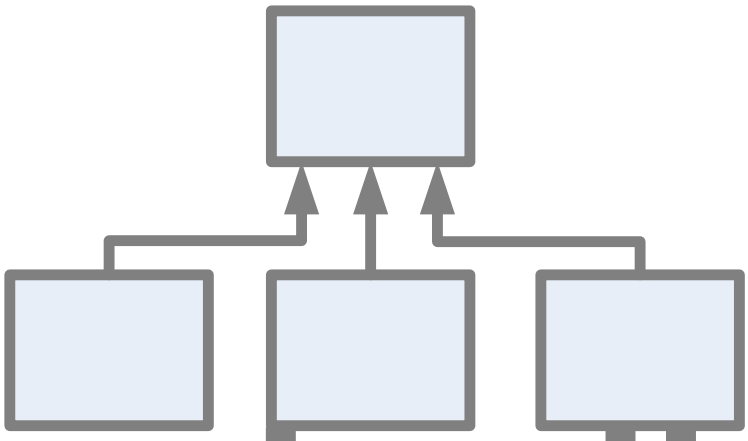


vyhodb

Getting started

Введение

“Vyhodb” это СУБД, которая использует сетевую модель данных, поддерживает ACID транзакции, написана на Java и предназначена для использования Java приложениями.

Данный документ даёт краткую информацию о vyhodb и её API. Документ предназначен для разработчиков и архитекторов прикладного программного обеспечения. Знание языка программирования java, а также основных классов JDK, является обязательным.

Каждая глава документа (за исключением первых двух) описывает тот или иной API и состоит из краткого описания API и примера кода:

API	Описание
Server API	Запуск/останов сервера; управление транзакциями.
Space API	Чтение/изменение данных vyhodb.
Indexes	Работа с индексами vyhodb.
RSI (Remote Service Invocation)	Удалённый вызов методов java объектов, выполняющихся внутри vyhodb сервера и имеющих доступ к Space API.
Functions API	Механизм обход данных vyhodb.
ONM API	Чтение/запись графа Java объектов из/в vyhodb.

Данный документ входит в пакет документации vyhodb, состав которого представлен в следующей таблице:

Документ	Описание
Getting Started	Быстрый старт. Документ даёт представление о vyhodb API на простых примерах без детального описания.
Developer Guide	Руководство разработчика. Описывает различные vyhodb API и их использование.
Functions Reference	Справочник по функциям Functions API
Administrator Guide	Руководство администратора. Описывает архитектуру vyhodb, её конфигурирование и администрирование.

Исходный код примеров

Исходный код примеров можно скачать отсюда http://www.vyhodb.com/doku.php?id=docs.

Для запуска примеров необходимо настроить среду разработки, а именно:

1. Включить в classpath среды разработки все jar архивы из каталога lib.
2. В каждом примере изменить значения статических полей **LOG, DATA** с тем, чтобы они указывали на абсолютные имена файлов хранилища vyhodb: **storage/vyhodb.log** и **storage/vyhodb.data**.

Дистрибутив vyhodb, установка и запуск

Установка JRE

Для работы vyhodb необходима версия JRE не ниже 7u64. Последнюю версию, а также инструкции по установке JRE можно найти вот здесь http://www.oracle.com/technetwork/java/javase/downloads/index.html.

Установка vyhodb

Последнюю версию дистрибутива vyhodb можно скачать вот здесь download. Дистрибутив представляет собой zip архив, с каталогом vyhodb-0.9.0.

Конфигурирование пути к JRE

После разархивирования, необходимо указать путь к JRE:

Windows:

1) Открыть файл **bin-cmd\set-env.cmd**
2) Установить значение переменной **JRE_HOME**: SET JRE_HOME=C:\Program Files\Java\jdk1.7.0_60\jre

Linux:

1) Открыть файл **bin-sh/set-env.sh**
2) Установить значение переменной **JRE_HOME**: JRE_HOME=/home/jdk1.7.0_79/jre

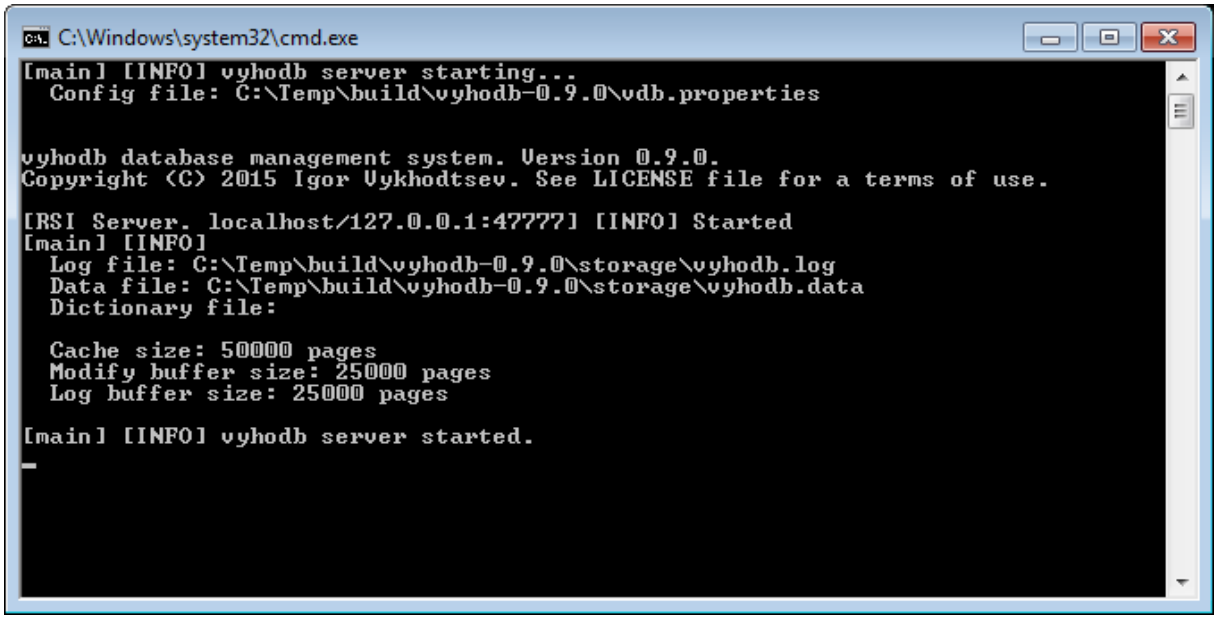
Запуск

Теперь можно запустить vyhodb сервер в standalone режиме. Для этого выполните скрипт:

Windows: **vdb-start.cmd**

Linux: **vdb-start.sh**

Ниже показан экран вывода при запуске vyhodb:



Для остановки vyhodb сервера используйте Ctrl+C или утилиты **vdb-close-remote** (расположены в каталогах **bin-cmd**, **bin-sh**).

Структура каталога vyhodb

Каталог/файл	Описание
bin-cmd	Утилиты командной строки (windows)
bin-sh	Утилиты командной строки (linux)
lib	API и системные jar архивы
services	Каталог для jar архивов RSI Service-ов
storage	Файлы базы данных (файл данных, файл журнала транзакций)
LICENSE	Лицензионное соглашение vyhodb
vdb.properties	Конфигурационный файл vyhodb
vdb-start.cmd	Скрипт запуска vyhodb (windows)
vdb-start.sh	Скрипт запуска vyhodb (linux)

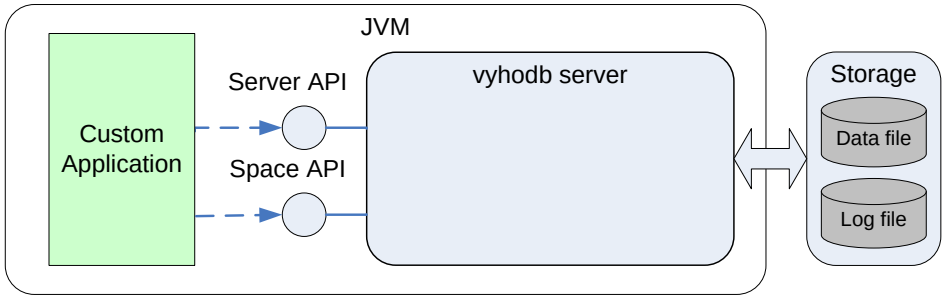
Режимы работы vyhodb

vyhodb поддерживает следующие режимы работы[1]:

1) Embedded
2) Standalone

Embedded

В этом режиме, приложение, использующее vyhodb, и сам vyhodb сервер работают внутри одной JVM. Приложение использует **Server API** для запуска vyhodb сервера и управления транзакциями. Для чтения и изменения данных, приложение использует **Space API**.

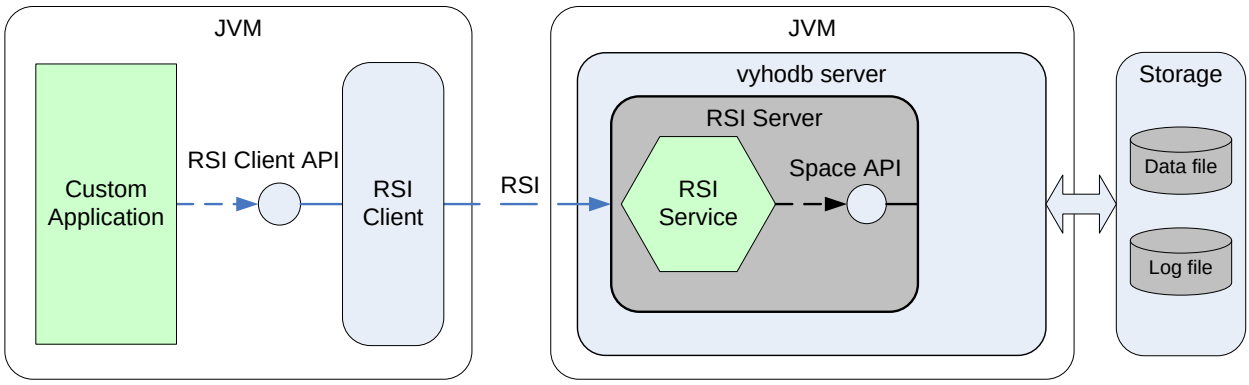


Во всех разделах данного документа, кроме раздела посвященного RSI, мы будем использовать данный режим работы.

Standalone

В этом режиме, приложение и vyhodb сервер работают в разных JVM. Для их взаимодействия используется так называемый механизм Remote Service Invocation (RSI).

Для запуска сервера vyhodb в standalone режиме используются командные файлы: **vdb-start.cmd, vdb-start.sh**.



В данном режиме, клиентское приложение имеет доступ к данным опосредованно, путем удалённого вызова методов RSI Service-ов, реализующих бизнес логику. RSI Service читает и изменяет данные, используя Space API.

[1] Также существует третий режим запуска - “Local”. Данный режим позволяет клиентскому приложению запустить vyhodb в своей JVM, но при этом использовать технологию RSI (Server API не доступен для данного режима). Более подробно данный режим описан в документах “Developer Guide”, “Administration Guide”.

Компонент RSI Server управляет жизненным циклом RSI Service-ов. Он также открывает новую транзакцию для каждого удаленного вызова и завершает её по окончании работы метода RSI Service-а.

Клиентское приложение, для удаленного вызова методов RSI Service-ов устанавливает сетевое соединение с RSI Server-ом используя RSI Client API.

Пример реализации RSI Service-а и использования RSI Client API приведены в разделе **RSI**.

Server API

Server API предназначен для запуска vyhodb сервера в embedded режиме, и управления транзакциями.

Ниже приведен пример, в котором мы запускаем vyhodb в embedded режиме, открывает **Modify** транзакцию, и изменяем корневую запись (о записях и модели данных – см. следующий раздел). Далее открываем **Read** транзакцию и читаем измененные данные:

```
package com.vyhodb.started.server;

import java.io.IOException;
import java.util.Date;
import java.util.Properties;

import com.vyhodb.server.Server;
import com.vyhodb.server.TrxSpace;
import com.vyhodb.space.Record;

public class ServerAPI {

    public static final String LOG = "C:\\vyhodb-0.9.0\\storage\\vyhodb.log";
    public static final String DATA = "C:\\vyhodb-0.9.0\\storage\\vyhodb.data";

    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
        props.setProperty("storage.log", LOG);
        props.setProperty("storage.data", DATA);

        try(Server server = Server.start(props)) {
            modifyTrx(server);
            readTrx(server);
        }
    }

    private static void modifyTrx(Server server) {
        TrxSpace space = server.startModifyTrx();   // Starts modify transaction

        Record root = space.getRecord(0L);          // Retrieves root record
        root.setField("Current Time", new Date());  // Changes field

        space.commit();                             // Commits transaction
    }

    private static void readTrx(Server server) {
        TrxSpace space = server.startReadTrx();     // Starts read transaction

        Record root = space.getRecord(0L);          // Retrieves root record
        Date date = root.getField("Current Time");  // Gets field value
        System.out.println(date);

        space.rollback();                           // Rolls back transaction
    }
}
```

Обратите внимание, что в данном примере (также как и в последующих), необходимо корректно прописать пути к data и log файлам vyhodb сервера (константы LOG, DATA). По умолчанию данные файлы находятся в каталоге storage каталога vyhodb.

Результат:

```
vyhodb database management system. Version 0.9.0.
Copyright (C) 2015 Igor Vykhodtsev. See LICENSE file for a terms of use.

[main] [INFO]
  Log file: C:\vyhodb-0.9.0\storage\vyhodb.log
```

```
  Data file: C:\vyhodb-0.9.0\storage\vyhodb.data
  Dictionary file:

  Cache size: 50000 pages
  Modify buffer size: 25000 pages
  Log buffer size: 25000 pages

[main] [INFO] vyhodb server started.
Mon Sep 07 03:34:02 BRT 2015
[main] [INFO] vyhodb server closed.
```

Большая часть вывода на экран является сообщениями логирования vyhodb сервера, поэтому в последующих примерах они будут опущены. Т.е. значимым результатом для нашего примера является:

```
Thu Jul 09 07:28:10 BRT 2015
```

Space API

Space API предназначен для чтения и изменения данных vyhodb.

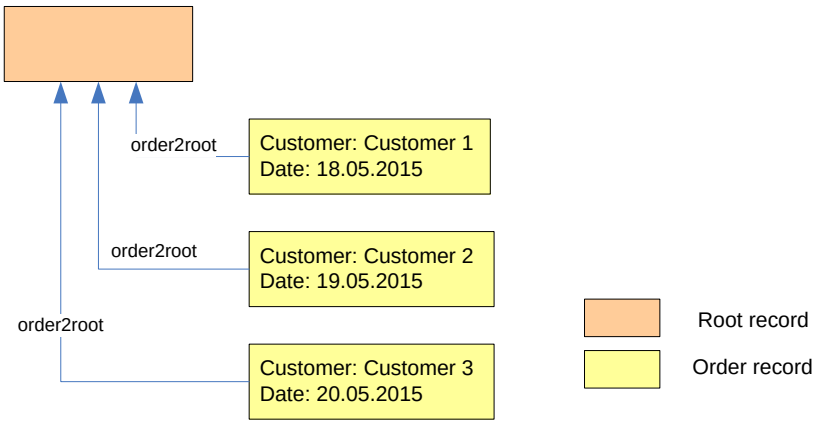
Vyhodb использует сетевую модель данных которая состоит из: записей, полей, линков:

- **Запись** – это основной элемент vyhodb. Он является контейнером для полей и имеет автоматически генерируемый идентификатор типа long . Запись с id==0 называется «корневой» и создаётся автоматически при создании новой базы данных vyhodb (именно эту запись мы изменяли в предыдущем примере).
- **Поле** – именованное значение, хранящееся внутри записи. Имена полей уникальны в пределах одной записи.
- **Линк** соединяет две записи и имеет имя. Запись, с которой создаётся линк является «дочерней», на которую создаётся линк – “родительской”. Со стороны родительской записи имеется возможность обхода всех дочерних записей по определенному имени линка.

Вся функциональность по работе с моделью данных vyhodb сосредоточена в двух интерфейсах: **com.vyhodb.space.Space** и **com.vyhodb.space.Record**.

Интерфейс **com.vyhodb.server.TrxSpace** выполняет сразу две роли, он одновременно является и пространством записей и активной транзакцией.

В следующем примере мы создадим структуру записей, показанных на диаграмме:



Пример:

```
package com.vyhodb.started.space;

import java.io.IOException;
import java.util.Date;
import java.util.Properties;

import com.vyhodb.server.Server;
import com.vyhodb.server.TrxSpace;
import com.vyhodb.space.Record;
import com.vyhodb.space.Space;

public class SpaceAPI {
    public static final String LOG = "C:\\vyhodb-0.9.0\\storage\\vyhodb.log";
    public static final String DATA = "C:\\vyhodb-0.9.0\\storage\\vyhodb.data";

    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
```

```
        props.setProperty("storage.log", LOG);
        props.setProperty("storage.data", DATA);

        try(Server server = Server.start(props)) {
            TrxSpace space = server.startModifyTrx();
            example(space);
            space.rollback();
        }
    }

    private static void example(Space space) {
        Record root = space.getRecord(0L);                  // Retrieves record by id

        Record order1 = space.newRecord();                  // Creates new record
        order1.setField("Customer", "Customer 2");          // Sets field
        order1.setField("Date", new Date("05/19/2015"));    // Sets field
        order1.setParent("order2root", root);               // Creates link to root record

        Record order2 = space.newRecord();
        order2.setField("Customer", "Customer 3");
        order2.setField("Date", new Date("05/20/2015"));
        order2.setParent("order2root", root);

        Record order3 = space.newRecord();
        order3.setField("Customer", "Customer 1");
        order3.setField("Date", new Date("05/18/2015"));
        order3.setParent("order2root", root);

        // Retrieves child records and iterates over them
        for (Record order : root.getChildren("order2root")) {
            System.out.println(order);
        }
    }
}
```

Как и в предыдущем примере, мы запускаем vyhodb сервер в embedded mode и открываем Modify транзакцию.

Далее, в методе **example**() мы создаём три записи, которые логически соответствуют трём заказам. У каждой записи устанавливаем два поля: “**Customer**”, “**Date**”. Создаём три линка (имя линка=”**order2root**”) с каждой созданной записи на «корневую» запись. Таким образом, созданные записи являются дочерними по отношению к корневой записи.

Далее, мы получаем дочерние записи с корневой записи (которые только что были созданы) и в цикле выводим их на экран. Результат работы (id могут отличаться):

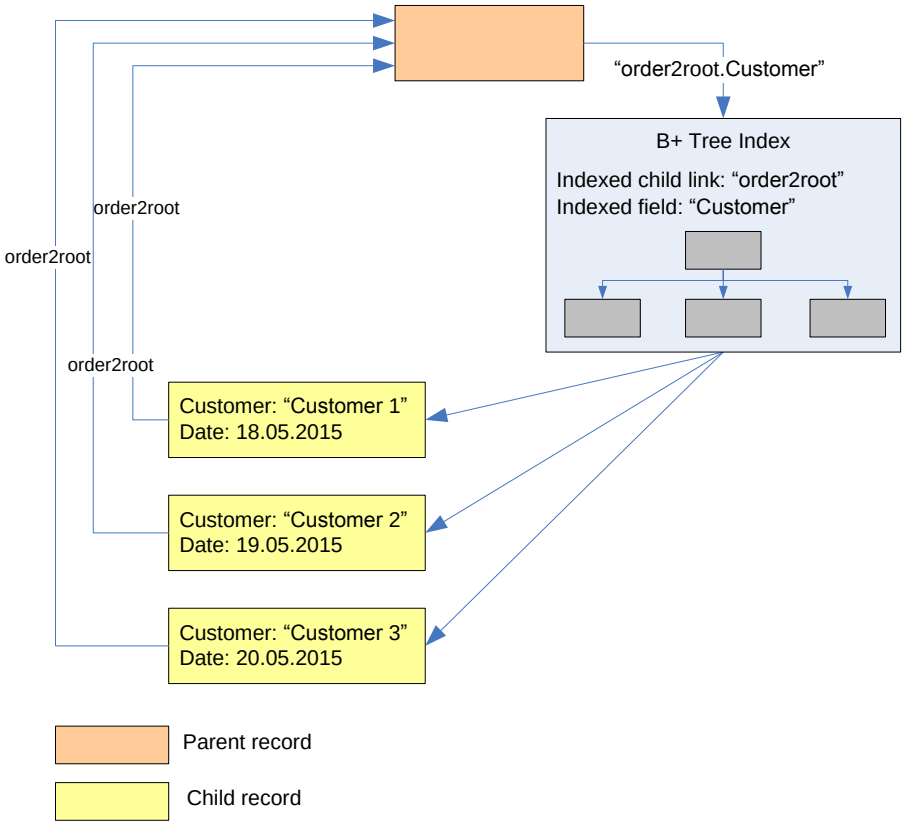
```
{Customer="Customer 2", Date="Tue May 19 00:00:00 BRT 2015"} id=523
{Customer="Customer 3", Date="Wed May 20 00:00:00 BRT 2015"} id=534
{Customer="Customer 1", Date="Mon May 18 00:00:00 BRT 2015"} id=545
```

Indexes

Индексы используются для быстрого поиска дочерних записей по значению индексируемого поля (или полей).

Индексы создаются со стороны родительской записи и индексируют поля дочерних записей, по значениям которым необходимо проводить поиск.

Ниже приведен пример, в котором на корневой записи создается индекс с именем “**order2root.Customer**”. Данный индекс индексирует поле “**Customer**” дочерних записей, ссылающихся на корневую запись по линку “**order2root**”. Создаваемые данные и место индекса показано на диаграмме:



Метод main() и директивы импорта аналогичны примеру из раздела Space API:

```
package com.vyhodb.started.index;

. . .

public class Indexes {

. . .

    private static void example(Space space) {
        Record root = space.getRecord(0L);

        // Creates records
        {
            Record order1 = space.newRecord();
            order1.setField("Customer", "Customer 2");
            order1.setField("Date", new Date("05/19/2015"));
            order1.setParent("order2root", root);

            Record order2 = space.newRecord();
```

```
            order2.setField("Customer", "Customer 3");
            order2.setField("Date", new Date("05/20/2015"));
            order2.setParent("order2root", root);

            Record order3 = space.newRecord();
            order3.setField("Customer", "Customer 1");
            order3.setField("Date", new Date("05/18/2015"));
            order3.setParent("order2root", root);
        }

        // Creates index which indexes "Customer" field on child records
        createIndex(root);

        // Creates search criteria
        Criterion criterion = new Equal("Customer 3");

        // Searches records and iterates over search result
        for (Record order : root.searchChildren("order2root.Customer", criterion)) {
            System.out.println(order);
        }
    }

    private static void createIndex(Record record) {
        String fieldName = "Customer";
        String linkName = "order2root";
        String indexName = "order2root.Customer";

        // Creates indexed field descriptor
        IndexedField indexedField = new IndexedField(
                fieldName,
                String.class,
                Nullable.NOT_NULL
        );

        // Creates index descriptor
        IndexDescriptor indexDescriptor = new IndexDescriptor(
                indexName,
                linkName,
                Unique.DUPLICATE,
                indexedField
        );

        // Creates index
        record.createIndex(indexDescriptor);
    }
}
```

Результат:

```
{Customer="Customer 3", Date="Wed May 20 00:00:00 BRT 2015"} id=534
```

За более подробной информацией об индексах необходимо обратиться к разделу “Indexes” документа “Developer Guide”.

RSI

Remote Service Invocation (RSI) – это механизм удалённого вызова методов RSI Service-а. RSI Service это java объект, который «живет» внутри vyhodb сервера (а более точнее внутри RSI Server-а, см. “Administration Guide”) и имеет доступ к Space API.

В данном разделе мы реализуем RSI Service и клиентское приложение, которое вызовет его методы.

Сначала создадим контракт сервиса:

```
package com.vyhodb.started.rsi;

import java.util.Collection;
import java.util.Date;

import com.vyhodb.rsi.Implementation;
import com.vyhodb.rsi.Modify;
import com.vyhodb.rsi.Read;

@Implementation(className="com.vyhodb.started.rsi.ServiceImpl")
public interface Service {

    @Modify
    public long newOrder(String customerName, Date date);

    @Read
    public Collection<String> listCustomers();
}
```

Теперь реализуем RSI Service:

```
package com.vyhodb.started.rsi;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;

import com.vyhodb.space.Record;
import com.vyhodb.space.ServiceLifecycle;
import com.vyhodb.space.Space;

public class ServiceImpl implements Service, ServiceLifecycle {

    private Space _space;

    @Override
    public void setSpace(Space space) {
        _space = space;
    }

    @Override
    public long newOrder(String customerName, Date date) {
        Record root = _space.getRecord(0L);

        Record order = _space.newRecord();
        order.setField("Customer", customerName);
        order.setField("Date", date);
        order.setParent("order2root", root);

        return order.getId();
    }

    @Override
    public Collection<String> listCustomers() {
        ArrayList<String> result = new ArrayList<>();
```

```
        Record root = _space.getRecord(0L);
        for (Record order : root.getChildren("order2root")) {
            result.add((String) order.getField("Customer"));
        }

        return result;
    }
}
```

И в последнюю очередь реализуем класс клиентского приложения, который создаёт сетевое соединение и вызывает методы RSI Service-а:

```
package com.vyhodb.started.rsi;

import java.io.IOException;
import java.net.URISyntaxException;
import java.util.Collection;
import java.util.Date;

import com.vyhodb.rsi.Connection;
import com.vyhodb.rsi.ConnectionFactory;
import com.vyhodb.rsi.RsiClientException;

public class Client {

 public static final String URL = "tcp://localhost:47777";

    public static void main(String[] args) throws RsiClientException, IOException,
URISyntaxException {

        try(Connection connection = ConnectionFactory.newConnection(URL)) {

            Service service = connection.getService(Service.class);

            service.newOrder("Customer 3", new Date("05/20/2015"));
            service.newOrder("Customer 1", new Date("05/18/2015"));
            service.newOrder("Customer 2", new Date("05/19/2015"));

            Collection<String> customers = service.listCustomers();
            System.out.println(customers);
        }
    }
}
```

После того, как реализация готова, необходимо выполнить следующие шаги:

1) Упаковать классы Service и ServiceImpl в jar архив.
2) Остановить standalone vyhodb сервер, если он у вас работает.
3) Скопировать jar архив в каталог services
4) Запустить vyhodb сервер в standalone режиме. Для этого используются скрипты: vdb-start.cmd/vdb-start.sh
5) Запустить класс Client.

vyhodb сервер не поддерживает «горячего» развёртывания, поэтому необходима его перезагрузка.

Для вашего удобства, вместе с исходным кодом примеров, вы можете найти файл **vdb-started-rsi.jar** который уже содержит скомпилированные и упакованные классы из данного раздела. Для его запуска необходимо выполнить следующие шаги:

1) Скопировать **vdb-started-rsi.jar** в каталог **services**.
2) Запустить/перезагрузить vyhodb в standalone режиме.

3) Находясь в каталоге vyhodb выполнить следующую команду для запуска клиентского приложения:
    a. Windows: **java -cp lib/*;services/* com.vyhodb.started.rsi.Client**
    b. Linux: **java –cp lib/*:services/* com.vyhodb.started.rsi.Client**

Результат:

```
[Customer 3, Customer 1, Customer 2]
```

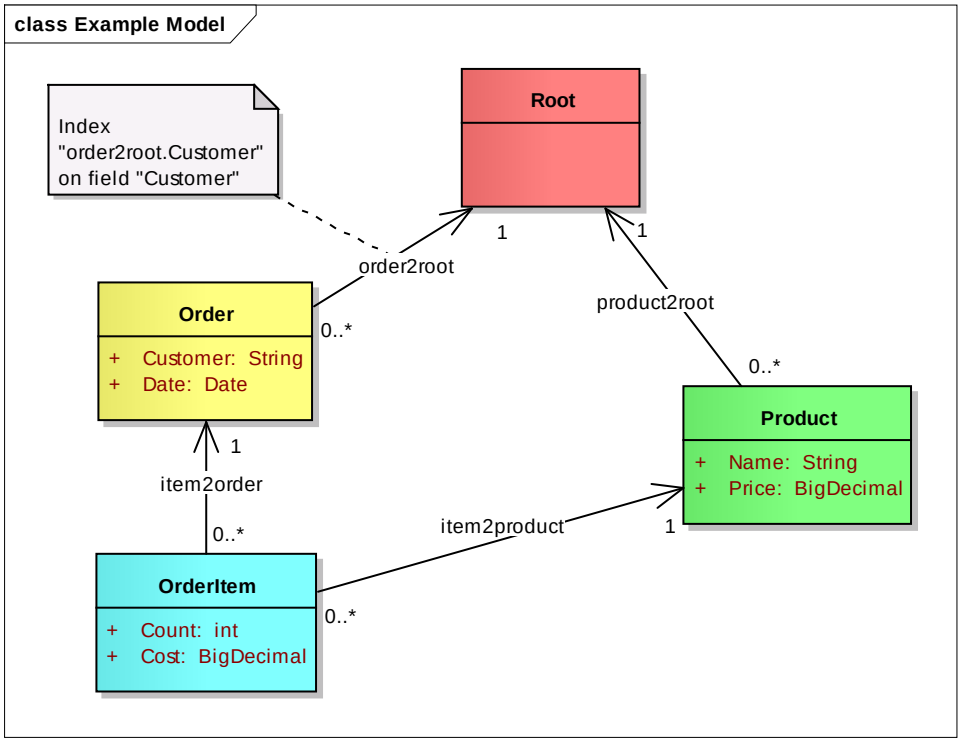
Functions API

Functions API используется для обхода записей.

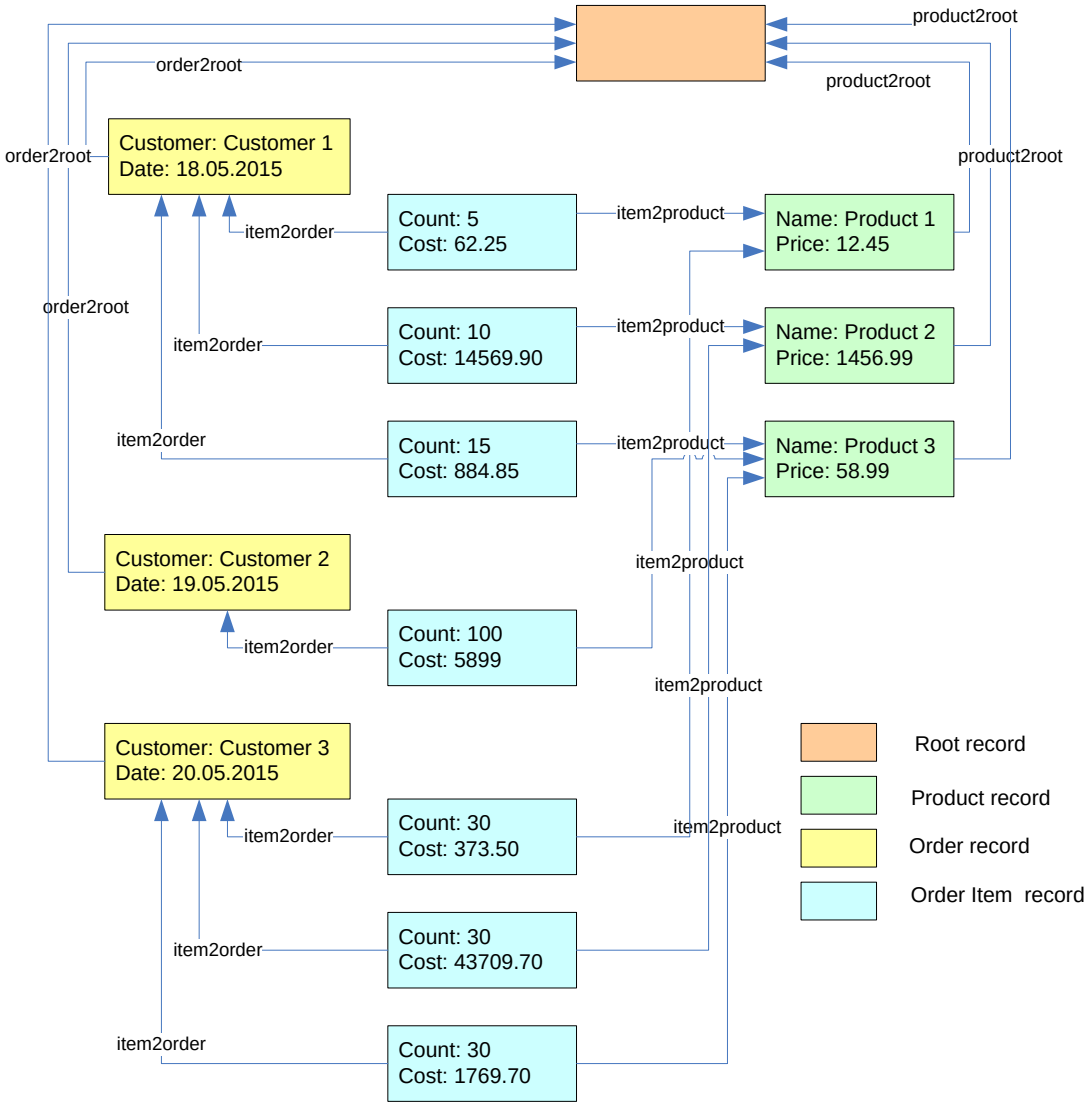
Основная идея: создаётся дерево функций, каждая их которых выполняет определенные действия. Функцией является любой объект, класс которого унаследован от **com.vyhodb.F**. После этого корневой функции передаётся запись и запускается процесс исчисления всего дерева функций. Подход напоминает функциональное программирование.

За более детальной информацией о **Functions API** необходимо обратиться к документам “Developer Guide”, “Functions Reference”.

В данном и следующем разделах мы будем использовать более сложную модель данных, показанную на следующей диаграмме:



Для создания тестовых записей в соответствии с вышеприведённой моделью, используется статический метод класса **com.vyhodb.utils.DataGenerator#generate()**, который создаёт следующие записи, поля и линки:



Рассмотрим пример, в котором мы выводим на экран записи “Order Item” ссылающиеся на продукт (“Product”) с определенным именем.

Пример реализован используя обычный подход с циклами **for-each** (метод **main**(), директивы импорта также константы опущены, так как они аналогичны примеру из раздела **Space API**):

```
package com.vyhodb.started.functions;

. . .
public class IntroForeach {

. . .

    private static void example(Space space) {
        String productName = "Product 2";

        // Generates sample data
        Record root = space.getRecord(0L);
        DataGenerator.generate(root);

        for (Record product : root.getChildren("product2root")) {
            if (productName.equals(product.getField("Name"))) {
                for (Record item : product.getChildren("item2product")) {
                    System.out.println(item);
                }
            }
        }
    }
}
```

Результат:

```
{Cost=14569.90, Count=10} id=600
{Cost=43709.70, Count=30} id=688
```

В методе example() мы видим нагромождение циклов **for-each**. И это причем для довольно-таки простого примера.

Посмотрим, как ту же самый пример можно реализовать, используя **Functions API**:

```
package com.vyhodb.started.functions;

import static com.vyhodb.f.CommonFactory.*;
import static com.vyhodb.f.NavigationFactory.*;
import static com.vyhodb.f.PredicateFactory.*;

. . .
public class IntroFunctions {

. . .

    private static void example(Space space) {
        String productName = "Product 2";

        // Generates sample data
        Record root = space.getRecord(0L);
        DataGenerator.generate(root);

        // Builds function tree
        F printf =
        childrenIf("product2root", fieldsEqual("Name", productName),
            children("item2product",
                printCurrent()
            )
        );

        // Evaluates function
        printf.eval(root);
    }
}
```

Результат:

```
{Cost=14569.90, Count=10} id=600
{Cost=43709.70, Count=30} id=688
```

Обратите внимание на директивы статического импорта:

```
import static com.vyhodb.f.factories.CommonFactory.*;
import static com.vyhodb.f.factories.NavigationFactory.*;
import static com.vyhodb.f.factories.PredicateFactory.*;
```

Которые мы используем для сокрытия имен классов, являющихся фабриками функций. Именно благодаря этому, код создания функции приобретает вид функционального языка программирования.

Ниже представлен еще один пример, в котором вычисляется сумма всех продаж (поле Cost у записей “OrderItem”) продукта с заданным именем:

```
package com.vyhodb.started.functions;

import static com.vyhodb.f.AggregatesFactory.*;
import static com.vyhodb.f.CommonFactory.*;
import static com.vyhodb.f.NavigationFactory.*;
import static com.vyhodb.f.PredicateFactory.*;
import static com.vyhodb.f.RecordFactory.*;

. . .

public class Sum {

. . .

    private static void example(Space space) {
        String productName = "Product 2";

        // Generates sample data
        Record root = space.getRecord(0L);
        DataGenerator.generate(root);

        // Builds function tree
        F sumF =
            composite(
                childrenIf("product2root", fieldsEqual("Name", productName),
                    children("item2product",
                        sum(getField("Cost"))
                    )
                ),
                getSum()
            );

        // Evaluates function
        BigDecimal sum = (BigDecimal) sumF.eval(root);
        System.out.println(sum);
    }
}
```

Результат:

```
58279.60
```

ONM API

ONM API (Object-to-Network model Mapping) предназначен для:

1) **ONM Reading.** Чтения графа java объектов из графа vyhodb записей.
2) **ONM Writing.** Обновления vyhodb записей на основе графа java объектов. Под обновлением понимается создание/удаление записей, изменение значений полей, создание/удаление линков.
3) **ONM Cloning.** Клонирования графа java объектов.

В этом документе мы рассмотрим лишь ONM Reading и ONM Writing. За более подробной информацией об ONM API необходимо обратиться к документу “Developer Guide”.

Маппинг между полями классов и полями/линками vyhodb записей задается либо аннотациями, либо внешним xml файлом. В наших примерах мы будем использовать аннотации.

Java классы

Создадим следующие классы, соответствующие нашей модели данных, описанной в предыдущем разделе **Functions API**. Исходный код классов, как и примеры этого раздела, находятся в пакете **com.vyhodb.started.onm.** Итак, нам необходимы классы:

1) **Root**
2) **Order**
3) **OrderItem**
4) **Product**

Root

```
package com.vyhodb.started.onm;

import java.util.ArrayList;
import java.util.List;

import com.vyhodb.onm.Children;
import com.vyhodb.onm.Id;
import com.vyhodb.onm.Record;

@Record
public class Root {

    @Id
    private long id = 0;

    @Children(linkName="order2root")
    private ArrayList<Order> orders = new ArrayList<>();

    public long getId() {
        return id;
    }

    public List<Order> getOrders() {
        return orders;
    }
}
```

Order

```
package com.vyhodb.started.onm;

import java.util.ArrayList;
import java.util.List;
```

```
import com.vyhodb.onm.Children;
import com.vyhodb.onm.Field;
import com.vyhodb.onm.Id;
import com.vyhodb.onm.Record;

@Record
public class Order {

    @Id
    private long id = -1;

    @Children(linkName="item2order")
    private ArrayList<OrderItem> items = new ArrayList<>();

    @Field(fieldName="Customer")
    private String customerName;

    public long getId() {
        return id;
    }

    public List<OrderItem> getItems() {
        return items;
    }

    public String getCustomerName() {
        return customerName;
    }

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }
}
```

OrderItem

```
package com.vyhodb.started.onm;

import com.vyhodb.onm.Field;
import com.vyhodb.onm.Id;
import com.vyhodb.onm.Parent;
import com.vyhodb.onm.Record;

@Record
public class OrderItem {

    @Id
    private long id = -1;

    @Parent(linkName="item2order")
    private Order order;

    @Parent(linkName="item2product")
    private Product product;

    @Field(fieldName="Count")
    private int count;

    public long getId() {
        return id;
    }

    public Order getOrder() {
        return order;
    }

    public void setOrder(Order order) {
        this.order = order;
    }
```

```
    public Product getProduct() {
        return product;
    }

    public void setProduct(Product product) {
        this.product = product;
    }

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }
}
```

Product

```
package com.vyhodb.started.onm;

import com.vyhodb.onm.Field;
import com.vyhodb.onm.Id;
import com.vyhodb.onm.Record;

@Record
public class Product {

    @Id
    private long id = -1;

    @Field(fieldName="Name")
    private String name;

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

ONM Reading

Для чтения графа java объектов используется “Functions API”. Создаётся функция, которая обходит vyhodb записи и для каждой посещаемой записи создаёт java объект и устанавливает связи между созданными java объектами.

Во время обхода графа vyhodb записей, информация об маппинге храниться в объекте класса **com.vyhodb.onm.Mapping**, который необходимо заранее создать.

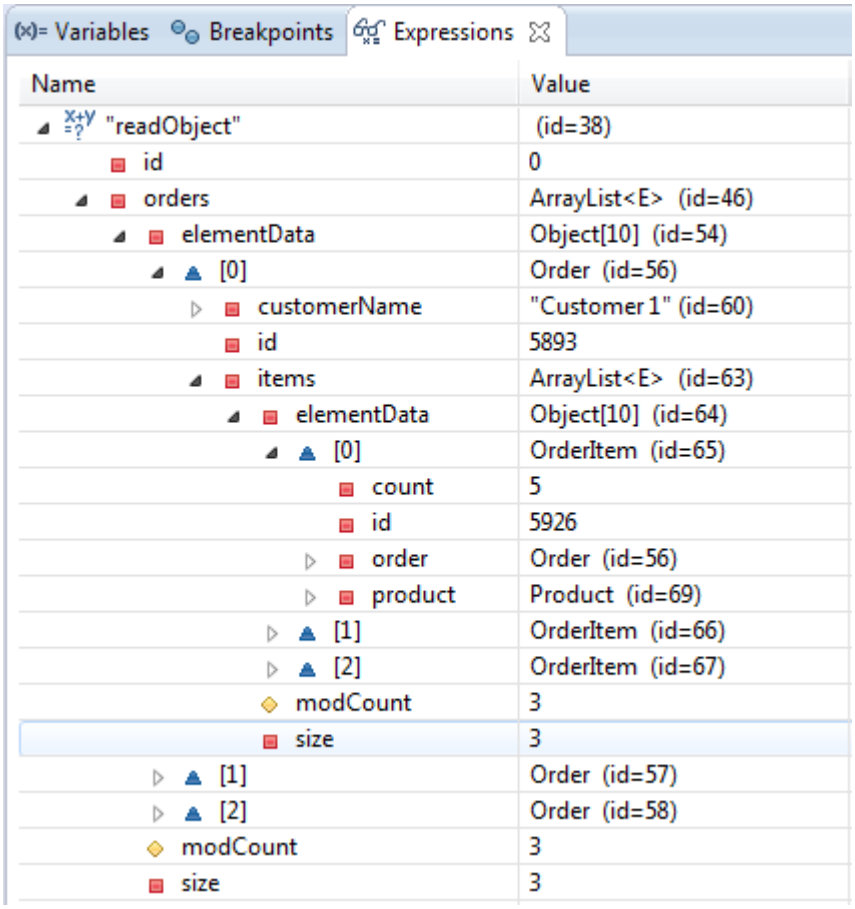
Для обхода используются те же самые функции, что были использованы в разделе “Functions API” так называемые навигационные функции: children(), parent(), index(). Разница состоит в том, что дерево навигационных функций должно быть “обёрнуто” ONM функцией startRead(), которая, выполняя закулисные действия, добавляет дополнительную обработку посещаемых vyhodb записей (создание java объектов в нашем примере).

Пример (метод main() и статические константы не показаны):

```
package com.vyhodb.started.onm;

import static com.vyhodb.f.NavigationFactory.*;
```

```
import static com.vyhodb.onm.OnmFactory.*;

import java.io.IOException;
import java.util.Properties;

import com.vyhodb.f.F;
import com.vyhodb.onm.Mapping;
import com.vyhodb.server.Server;
import com.vyhodb.server.TrxSpace;
import com.vyhodb.space.Record;
import com.vyhodb.space.Space;
import com.vyhodb.utils.DataGenerator;

public class OnmRead {

. . .

    private static void example(Space space) {
        // Generates sample data
        Record rootRecord = space.getRecord(0L);
        DataGenerator.generate(rootRecord);

        // Creates Mapping
        Mapping mapping = Mapping.newAnnotationMapping();

        // Builds ONM Reading function
        F readF =
            startRead(Root.class, mapping,
                children("order2root",
                    children("item2order",
                        parent("item2product")
                    )
                )
            );

        // Evaluates and reads object graph
        Root readObject = (Root) readF.eval(rootRecord);

        System.out.println(readObject);
    }
}
```

Результат выполнения лучше всего увидеть, запустив приложение в Debug режиме и проинспектировав граф объектов readObject. Ниже представлен скриншот среды Eclipse:



ONM Writing

Для записи графа Java объектов в vyhodb space используется класс **com.vyhodb.onm.Writer** а именно, его статический метод:

```
public static void write(Mapping mapping, Object rootObject, Space space)
```

Данный метод обходит переданный ему граф java объектов, начиная с объекта **rootObject**, используя java reflection и информацию о маппинге (объект **mapping**). И для каждого java объекта выполняет определенные действия: создаёт новую запись, изменяет значения полей/линков, удаляет запись. За более подробной информацией о механизме обхода java объектов и изменения записей обращайтесь к разделу “ONM API” документа “Developer Guide”.

В следующем примере мы читаем граф объектов, изменяем его и сохраняем изменения. Изменения заключаются в CustomerName у первого объекта Order и добавлении нового объекта OrderItem. В завершении, мы повторно читаем граф объектов, чтобы показать, что vyhodb записи были изменены/добавлены.

Метод main() а также статические константы соответствуют примеру из раздела “Space API” и опущены для удобства чтения:

```
package com.vyhodb.started.onm;

import static com.vyhodb.f.NavigationFactory.*;
import static com.vyhodb.onm.OnmFactory.*;

import java.io.IOException;
import java.util.Properties;
```

```
import com.vyhodb.f.F;
import com.vyhodb.onm.Mapping;
import com.vyhodb.onm.Writer;
import com.vyhodb.server.Server;
import com.vyhodb.server.TrxSpace;
import com.vyhodb.space.Record;
import com.vyhodb.space.Space;
import com.vyhodb.utils.DataGenerator;

public class OnmWrite {

. . .

    private static void example(Space space) {
        // Generates sample data
        Record rootRecord = space.getRecord(0L);
        DataGenerator.generate(rootRecord);

        // Builds ONM Read function
        Mapping mapping = Mapping.newAnnotationMapping();
        F readF =
            startRead(Root.class, mapping,
                children("order2root",
                    children("item2order",
                        parent("item2product")
                    )
                )
            );

        // Reads object graph
        Root readRoot = (Root) readF.eval(rootRecord);

        // Changes objects in read graph
        modify(readRoot);

        // ONM Writing
        Writer.write(mapping, readRoot, space);

        // Reads object graph again to illustrate ONM Writing result
        Root updatedRoot = (Root) readF.eval(rootRecord);
        System.out.println(updatedRoot);
    }

    private static void modify(Root root) {
        Order order = root.getOrders().get(0);
        Product product = order.getItems().get(0).getProduct();

        // Changes order's customer
        order.setCustomerName("Onm Customer");

        // Adds item
        OrderItem item = new OrderItem();
        item.setCount(1000000);
        item.setOrder(order);
        item.setProduct(product);
        order.getItems().add(item);
    }
}
```

Результат покажем через инспектирование графа updatedRoot: