My4S – A program-builder for creating web applications such as ERP, CRM, and others.

Avtor: Serhiy Chakhanyuk aka didSergiy4s.

Table of contens

What is My4S.

My4S is a freely distributed web application with open-source code. The server-side runs on Node.js (currently only on Windows).

My4S can become an excellent tool for small and medium-sized businesses, especially if you need a flexible and free ERP/CRM system. Here are several ways it can be used:

1. Financial and accounting management.

You can configure accounting registers to track income, expenses, taxes, and other financial operations. Built-in balance sheets help analyze the state of the budget.

2. Inventory management.

Thanks to the ability to work with different registers and directories, you can easily manage goods, monitor stock levels, automate purchases, and analyze sales.

3. Customer and sales management.

Use directories of contractors and document flow to maintain a client database, manage orders, automate invoicing, and analyze sales.

4. Business process automation.

My4S allows you to create documents and registers, enabling automation of key processes — from payroll calculations to scheduling meetings and tasks.

5. Customization for any tasks.

Since the system is flexible and open-source, you can adapt it to your needs: add new features, expand reporting, and integrate with other services.

I express my deep gratitude to the developers of Node.js, Firebird databases, Firebird Editor Pro, Visual Studio Code!

Core Principles:

• Use exclusively free software (except Windows).

• In Node.js, only one exe file is currently used, and this is a fundamental position! It is important that any third-party code used in the program is trustworthy or can be modified without the author's involvement.

• Ability to connect any amount of external code, both in Node.js and in everything that supports JavaScript. However, responsibility for its use lies with you!

Technologies:

• Database: Firebird 3.0 and above. For administration, you can use Firebird Editor Pro.

Editor: If you don't yet have a favorite JavaScript editor, try Visual Studio Code — it offers good support for Node.js. Node.exe can be restarted directly from the editor, which is very convenient for making changes and testing. Debug mode is also available!

• Client and server: Both are written in the same language — JavaScript.

Briefly about My4S

On the client side:

No need to install any programs! We use only a browser (tested on Google Chrome, Microsoft Edge, Opera, CCleaner Browser).

It is launched with a browser string "path:port/path_mydb/cf" — entry to the Constructor, or "path:port/path_mydb/db" — entry for working with the program (Enterprise).

Where:

path – the path to the My4S server (if on a local computer “127.0.0.1”, if in a corporate network then “servername with Node”. On a local computer and corporate network everything works without the internet! Otherwise, the standard path to the site or VPN).

port – a free port that the server will listen to (specified in “Start4s.js”). By default, mine is “1637”.

path_mydb – the name of the folder with a specific database (there may be several databases on the server).

Example: “**127.0.0.1:1637/mydb/cf**”.

On the server side:

Install Firebird 3.x (you can also use Firebird 4.x (5.x) — but the developers rushed ahead too quickly!!). I tested under 3.x, but I think it should work with others (4.x tested quickly — works, 5.x still raw!). For ERP program development, 3.x is quite sufficient for now, and later you can decide yourself!

Step-by-step instructions on how to set up a demo database and the main database are at the end of the description.

ODBC driver

For working with Firebird, I use the ODBC driver. Why not “node-firebird-driver-native”?

There are several reasons:

I try not to use code that I am not sure will work with Firebird x (I had experience with Firebird 2.5 and used a driver from the internet — then Firebird 2.8 was released and the driver stopped working!). So slow and steady wins the race!

Node.js (single-threaded) uses one CPU core when running! Therefore, I use my own method of working with the Firebird database — instead of the node-firebird-driver-native driver, I use DSN ODBC for Firebird and call 'child_process' asynchronously each time, then run a function written and invoked with "wscript.exe". Windows automatically distributes processes across available CPU cores.

I have written two files for "wscript.exe". One for writing "MyComand.js", the other for retrieving data "MySelect.js". These files are written in Jscript for Windows, and they are included in the distribution in the root folder “My4S”.

Пример кода:

```
let child_process = require('child_process');

            …

let MyWscrComm, MyWscrSel;

bason = require(require('MyBasePath').idx(BasonPath));

MyWscrComm = "wscript.exe MyComand.js " + bason.DSN + ' ' + proces;

MyWscrSel = "wscript.exe MySelect.js " + bason.DSN + ' ' + proces;

            …

let strExec = MyWscrComm + ' ' + CountLogs[mydbxx]++ + ' ' + mydbxx;
```

```
child_process.exec(strExec, () => { … }
```

Features of My4S.

After the release of Firebird 3.0 and above, the limits on database size (up to 128 TB) and tables (up to 64 TB) were significantly increased. Essentially, everything now depends only on the server's capabilities.

Because of this, I decided to create only one table for documents, directories, information registers, and possibly others.

For clarity, let's consider just one table — directories (CATS). But how can dozens or even hundreds of different directories such as nomenclature, contractors, etc. be placed into a single table? Each table has standard columns (id, id_n, name, actual, cod ...), and for all other columns we will use a blob (memo) field. All additional fields (columns) unique to each directory will be stored in a JSON string. I believe experienced programmers already understand my idea. NoSQL concepts have been around for a long time, and it is no coincidence that respected databases have begun to implement JSON.

To keep things consistent, all fields, names, and identifiers are written in English and limited to 31 characters.

Here is what we have:

id – a unique identifier of the table. Knowing the id, we can immediately access an element of any directory.

id_n – the name of the directory as defined in the Constructor (example: "contr", "nomencl", etc.).

name – the string of the directory element ("Gasoline A-95", "Some Beer", etc.), length defined in the Constructor.

cod – the string of the directory element (may contain barcode, office code, and other parameters separated by a delimiter such as a space or another symbol, length defined in the Constructor).

id_n + name and id_n + cod are paired indexes.

Attention! For the directory "Contractor Agreements" I would recommend the name "contract". This is not critical, but in some cases the program will work more accurately!

Almost the same applies to the information register (REGS), except instead of cod a date field is added for selecting periodic data.

With documents, everything is much more complex. We will use two linked tables: one for the header (DOCS) and another for the document rows (ROWS_DOCS).

Introduction to browser forms

Each table (directories, documents, information registers, etc.) will have a unified HTML template with fixed standard data, as well as a prepared HTML table (table) with a single row. For documents, two tables are provided: one for the document header (with a single row), and another for the tabular part (multi-row). This design eliminates the need for a form editor (IDE); it is sufficient to use a good HTML code editor (for example, Visual Studio Code).

Before opening a form, the template will be dynamically supplemented for each specific metadata object, using data recorded in blob fields or in JS files.

Database optimization

The program is designed to minimize database queries. Everything created in the Constructor is described by the properties of each metadata object and recorded into blob fields of the table. At the same time, this data is saved into a special JS file associated with the specific metadata. When opening a metadata form, the data is not loaded from the database but instead connected via SRC to this JS file. This file is updated each time the metadata is modified in the Constructor.

Such prepared JS files significantly reduce database queries, since the program works with preloaded data.

Capabilities of My4S

The My4S program is already prepared for creating ERP systems. For developing small applications such as family budget management or small warehouse accounting, deep knowledge of HTML (DOM) or programming languages is not even required — it is enough to study the My4S description.

The program supports:

• balance registers (simple and accounting, including the chart of accounts);

• cost write-off using the average method (FIFO and LIFO are not yet described — can be added independently);

• ready-made trial balance reports, both for balance registers and for accounting registers (with filters and saved settings);

• up to seven multi-row tables in a single document.

> 💡 Explanations for beginners:
>
> 1C8 / 1C77 — popular Russian business automation systems, with which the author compares My4S.
>
> Visionary programmers — enthusiasts who see the potential and help develop the project.
>
> Modules — functional blocks of the program (for example, accounting, warehouse).
>
> HTML templates — prepared layouts for displaying forms and interfaces in the browser.

The Future of My4S

I strive to catch up with the capabilities of 1C8, although this is not easy, considering that the system has been developed for more than twenty years (not counting 1C77). However, I hope that my initiative will attract a community of visionary programmers who will help turn this "ugly duckling" into a "beautiful swan."

At the moment, I am not aiming to create complete solutions such as "Accounting," "UTP," "UPP," and others. My goal is to develop a convenient tool for creating such programs.

My4S is a flexible system that allows you to easily:

• refine existing modules and create new ones,

• create new objects and describe their properties,

• add new tables and functions to the database,

• modify old and develop new HTML templates.

Database Recovery Mechanism

I have developed and successfully tested a database recovery mechanism after a critical failure. Its essence is to restore the database exactly up to the moment of failure.

Recovery process:

1. After creating a backup archive of the database, logging of all UPDATE and INSERT operations begins.

2. All SQL queries sent to the Firebird driver are recorded into files inside the logs folder.

3. In case of failure, the database is restored from the latest backup archive.

4. Then all saved files in the logs folder are processed in a loop and executed in Firebird, thereby restoring even the changes made in the Constructor!

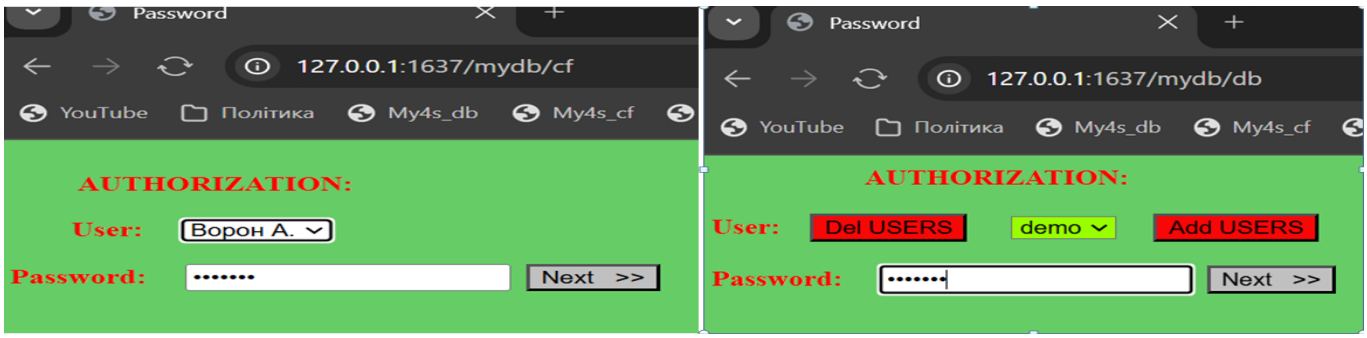
Attention! For now, I haven't inserted this mechanism into My4s. However, in the logs folders there is a folder called **bec_res**, which contains the necessary files that can be used to restore the database. In these files, you need to correctly specify the paths as on your server and also provide the correct password. This task will only be manageable for experienced programmers. Later, I'll come up with something (or maybe you'll suggest!) to make it convenient to restore the database directly in My4s.

I am not a security specialist, so I have not yet set myself the task of specifically protecting your data. If the program proves successful, I hope experts will either develop reliable solutions or advise how to implement them. For now, I know only one protection method — using OpenVPN with generated certificates. However, it is too early to think about this, since for studying the program you will be working on your own computer and can even do without the internet.

Prospects

Overall, My4S is a ready-made startup that lacks only one thing: for someone to believe in the program.

AUTHORIZATION.



Scr1

For entering the Constructor and the Enterprise, two different authorization approaches are used.

For entering the Constructor, the data is taken from the JSON file " ..\My4S\mydb\cf\Progv.us":

"Prog1": [

"Voron A.",

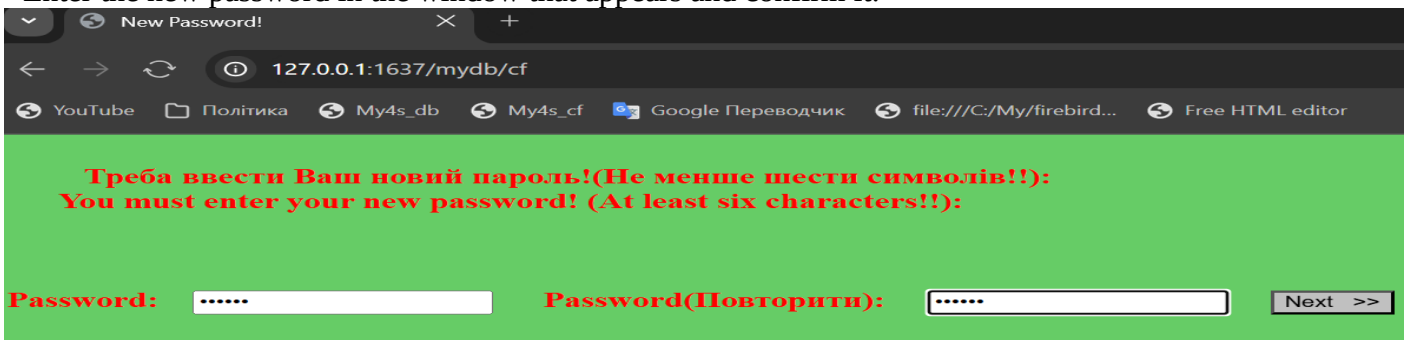
"767a06140f52c4cd3f81c9124be765ba038f895644e6d08bb0157c8467b2317b",

```
    ""

  ],

  " Prog2": [

    "Denis",

    "a73317a3d07ff37fa0837338a29e0c9bd6f488b788be6e98bfcd30af9edaf43c",

    ""

  ]

}
```

- ***Prog1***— `the main username displayed in the Enterprise.`
- **`"Voron A."`** — `representation, used exclusively for authorization.`

• The third element — a password encrypted using the Node.js 'crypto' module.
• The fourth element — not used yet, but may be useful in the future for configuring access rights.

To change the password:
- Replace the encrypted string in the JSON file with a temporary password (no longer than five characters).
- Log in to the Constructor.
- Enter the new password in the window that appears and confirm it.



Scr2

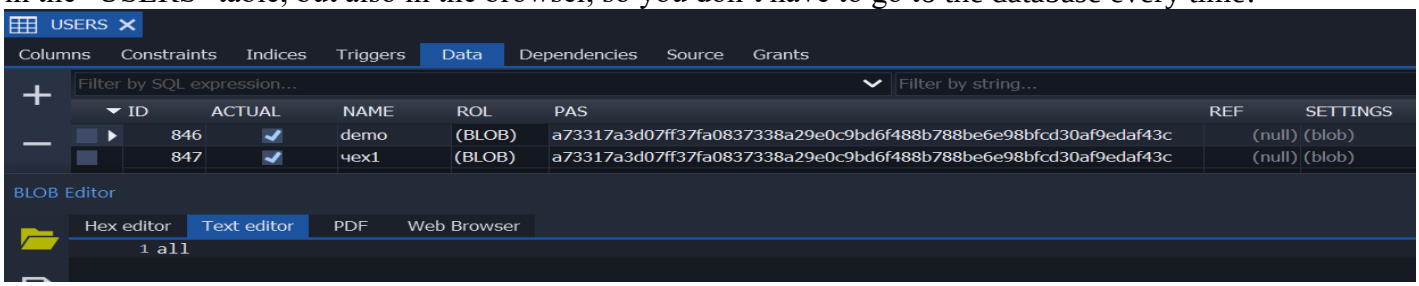
The new password will be encrypted and written into the JSON file.
If everything went well, a tab with information will appear (otherwise ERROR):

The new password has been accepted. Close the tab and enter the program with the new password!

In the demo version, I have temporarily removed the six-character limit!
By default, the password for " *Voron* A." = "11", and for "*Denis*" = "22".

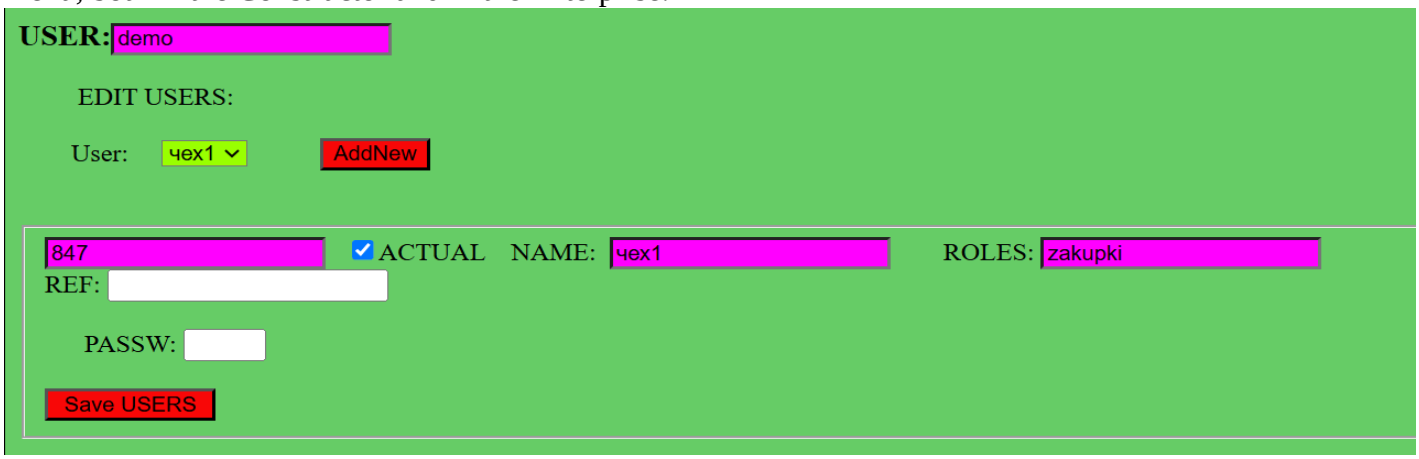
To enter the Enterprise, data is taken from the Firebird table “USERS”.
By default, the password for the user "demo" = "22". When entering the Enterprise database for the first time, the user field will be empty. You need to add a user with "ADD USERS". The user must know their name and type the suggested string, in this case "demo", then enter the password — and you are in the database! If you clear the browser history, you will need to add the user again. This is because user data will be stored not only

in the "USERS" table, but also in the browser, so you don't have to go to the database every time!



Scr3

For editing and adding new users in the Enterprise, there is an HTML form that is called from the "EditUsers" menu, both in the Constructor and in the Enterprise:



Scr4

Access rights management in the Enterprise

- Users with full rights ("all") can modify data but cannot modify other users with full rights.
- Programmers in the Constructor are the only ones who have the right to modify users with full rights.
- The chief accountant and their deputy can be granted full rights so as not to distract programmers when a new user needs to be added to the database.

Password change

1. Double-click in the "PASSW" field.
2. Enter a temporary password (no longer than five characters).
3. Save the changes by clicking "Save USERS".
4. To enter a new password, click "AddNew" and fill in all parameters.
5. Inform the user of the new password.
6. The user must log in with the temporary password, then enter and confirm the new one.
7. The new password will be known only to the user.

User linkage

In the REF field, it is planned to store the ID from the "Individuals" directory to link the user to a specific person.

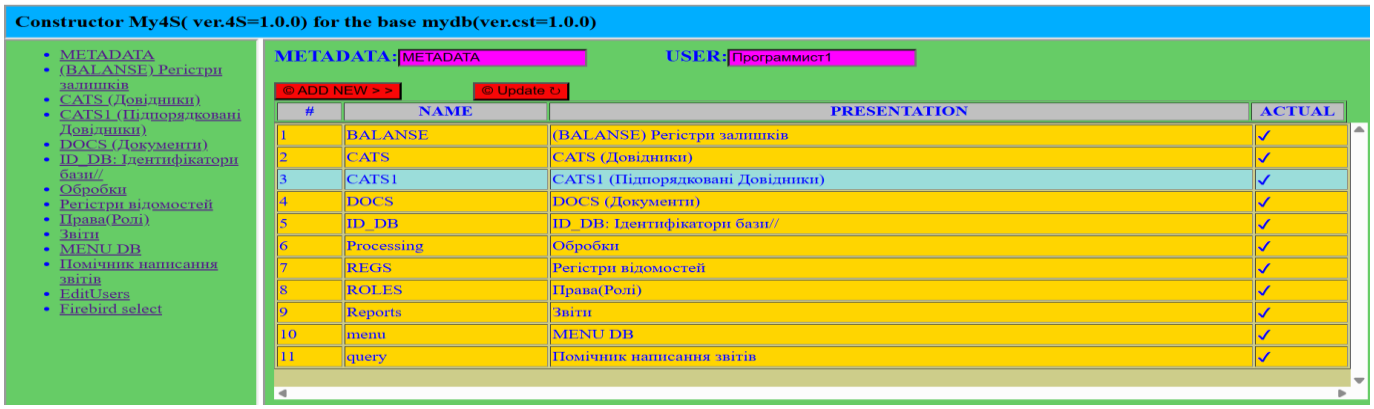
Initial base (do not confuse with demo)

The initial database (my4sNull) is a database where there is still no metadata in the Enterprise in the CATS, CATS1, DOCS, REGS tables. In the Designer, I have already entered some data in the METADATA table, which I hope is necessary for proper work, especially for accounting. And then you yourself should know what to do - this is your program!

Attention! When entering the Constructor of the initial database for the first time, the password is "11". Then you must set your new password with at least six characters. In the Enterprise, at the beginning there will be no users (Users). You need to enter yourself and your password in the Constructor first!

Entering the Constructor.

After logging into the Constructor, the initial form will appear with the main menu on the left.
METADATA:



Scr5

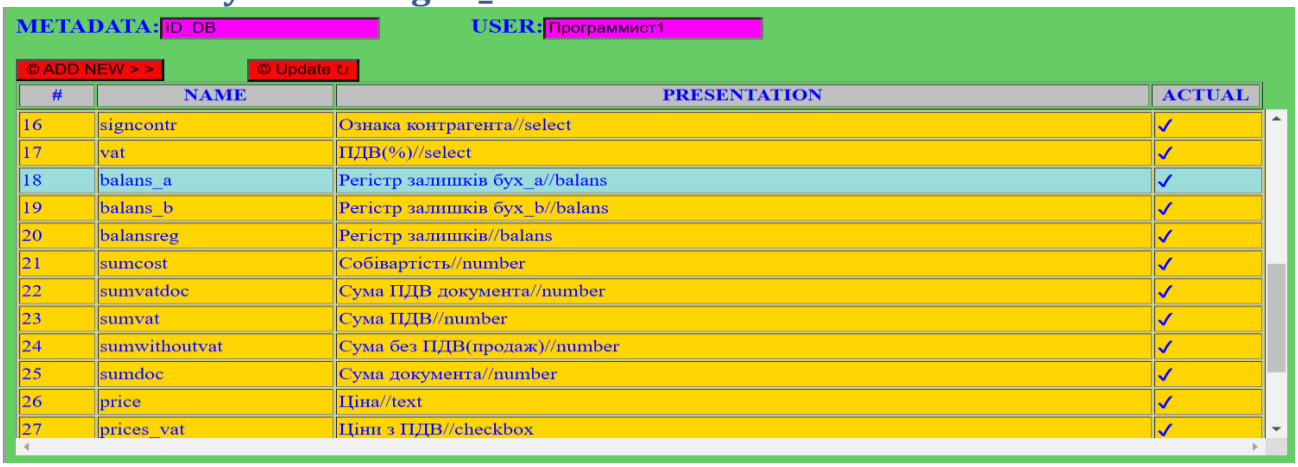
METADATA table and menu structure

The METADATA table almost completely corresponds to the menu on the left, except for the last two items. It is precisely based on this table that the menu you see is formed.

Adding new objects
If you add a new metadata object, it will automatically appear in the menu on the left.
However, for it to work fully, you will need to:
• Describe its properties and modules.
• If necessary, create new tables in Firebird.

For creating ERP programs (similar to 1C8), almost everything necessary is already available.
If additional functionality is required, it can be further developed!

Let us start by examining ID_DB.



Scr6

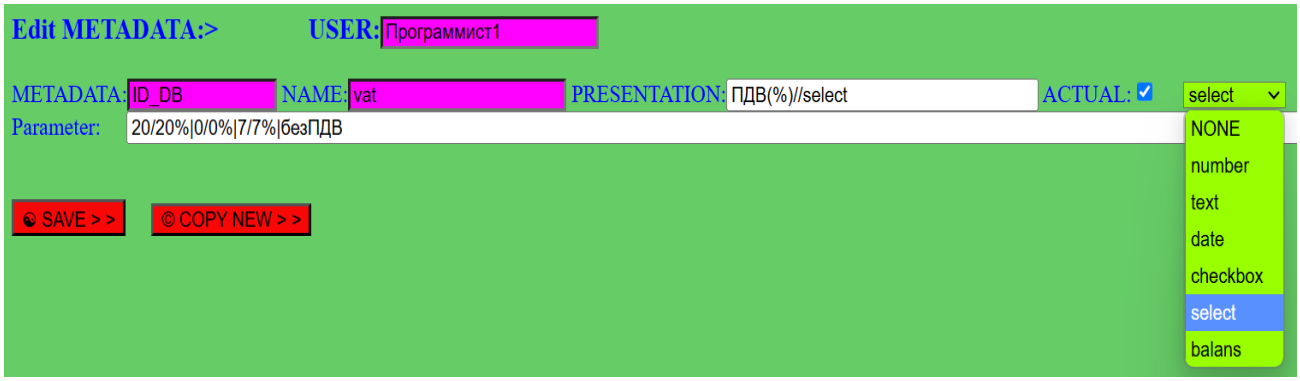
Main metadata identifiers

The program already provides predefined main identifiers (CATS, CATS1, DOCS, REGS, etc.), and it is better not to change them.
When creating a program, programmers can invent their own unique identifier names ("Name").

Restrictions on creating identifiers
• In large projects, different programmers may accidentally invent different identifiers for the same elements with identical properties.
• To avoid this, the use of new identifiers is only possible through the ID_DB table.
• If the required identifier is not there yet, it must be added.
• One may argue about the feasibility of such a solution, but I consider it the most practical.
• In the PRESENTATION column, after the description with //, the type of identifier is specified.
• Before each entry, you must select the type from the dropdown list.
• When saving, this type is automatically added to the line in PRESENTATION.

Additional properties of types date and select
A new property will appear: parameter. If necessary, new types can be added.
For date everything is obvious.
For select:
• You need to describe the dropdown HTML list as text (a replacement for enumeration).
• Each list element is separated by a vertical bar (|).
• Each element includes two parameters separated by a slash (/):
1. The first — the parameter itself.
2. The second — its representation.

You may notice that some types work like constants. Therefore, I do not have such metadata types as constants and enumerations!



Scr7

В Предприятии при выборе из выпадающего списка это будет выглядеть так:

Scr8

BALANSE (Balance registers):

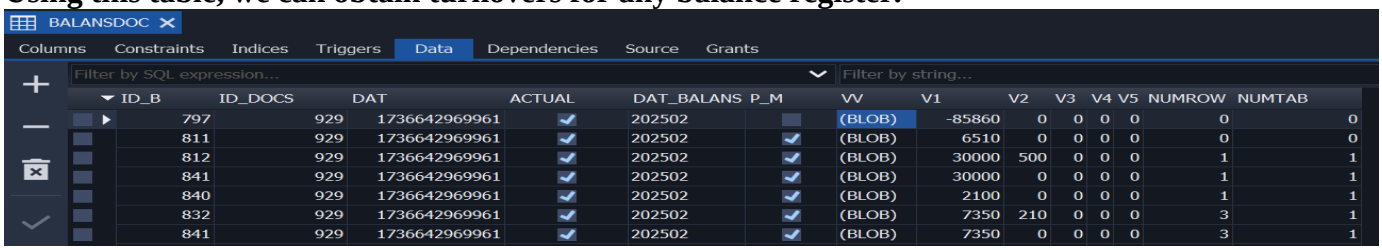
Balances in My4S and their support in Firebird

Balances can only be changed by documents. The Constructor provides mechanisms for creating movements of balance registers.
In Firebird, balance registers are supported by three tables:
BALANSOPT — stores unique combinations of dimensions for each balance register, each having a unique ID.
Example with nomenclature:
In all documents that modify a certain nomenclature, the same combinations of dimensions for each balance register are repeatedly used.
• To avoid storing these combinations in the database each time, they are replaced with unique IDs.

• This saves disk space and speeds up the system.

Scr9

ID – unique identifier.
ID_N – the name of the balance register.
W1…W5 – IDs of directory elements (we have already agreed that IDs are unique for any directory). For example: W1 is the ID of the nomenclature, W2 is the ID of the warehouse, etc., depending on how you configure it in the Constructor!
BALANSDOC — this table stores all movements of balance registers across all documents. Each document may contain any number of rows. One row equals one movement, as configured in the Constructor. Using this table, we can obtain turnovers for any balance register.

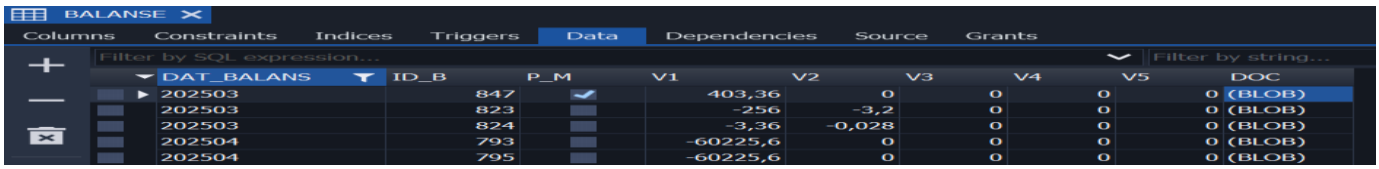


Scr10

ID_B – stores the ID of a unique combination of dimensions from the BALANSOPT table.
ID_DOCS – ID of the related document.

DAT – document date. In My4S, the built-in Firebird DATE type is rarely used; instead, the date is stored as a large number (BIGINT) generated in JavaScript. All queries are formed in JavaScript modules, so there is no need to waste machine time converting dates between JavaScript and Firebird! However, in two tables, DOCS and REGS, the DATE type is still used in the DATEFB column (without time). This is needed for filtering by date periods (day, week, month, quarter …). Firebird already has built-in functions for working with dates!
ACTUAL – flag indicating whether the movement is valid. If the document is marked invalid when saved, all movements linked to it also become invalid, and balances are reversed.
DAT_BALANS – 6 characters (year+month). Links the movement to a specific month and year.
P_M – indicates whether it is an inflow (true) or outflow (false).
VV – stores textual data used later, e.g., “act=Receipt of goods”.
V1…V5 – store resources, only NUMERIC type. Example: V1 is the movement amount, V2 is the quantity. Configured in the Constructor.
NUMROW – row number of the tabular section. If 0 – the document header.
NUMTAB – unique number of the tabular section for all movements of one document (up to 7 tabular sections).

BALANSE – this table stores balances at the beginning of each month, represented in the DAT_BALANS column.



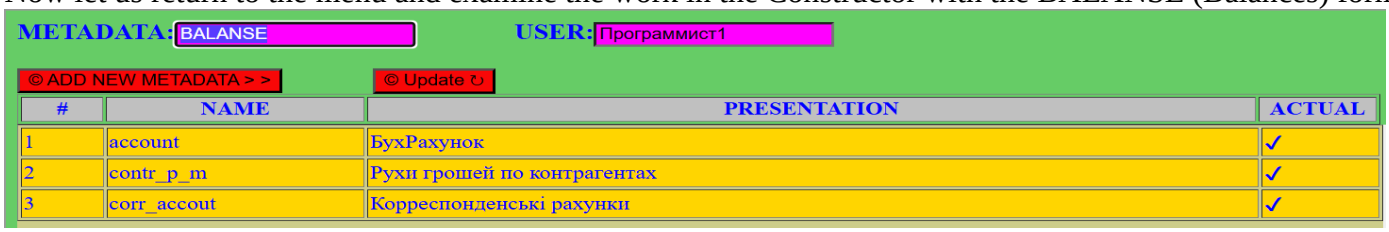
Scr11

Column types have already been described above, so I will not repeat them.
The beginning of the current month and the end of the previous month are essentially the same row in the table. If in the trial balance you set the period as the beginning and end of the month, the data will be quickly retrieved from this table row. However, if the start or end date (or both) fall between months, then to obtain bal-

ances for that date, the system will use the opening balance of the month plus the turnover from the BALANS-DOC table from the beginning of the month up to that date.

Important for accounting movements! The program does not support a direct link between debit and credit movements. Each movement exists independently, and the correctness and synchronization of movements are ensured by the programmer when describing movements in documents. When describing postings, simply record them in pairs — first debit, then credit (amounts with a minus sign) — and later reports will be able to calculate turnovers correctly! But these issues must be solved by programmers who will write the "Accounting for the country," while My4S allows you to build almost anything you want.

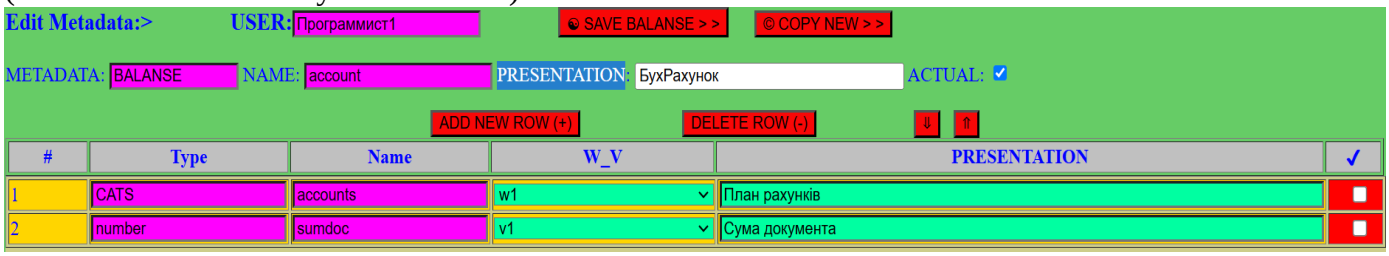
Now let us return to the menu and examine the work in the Constructor with the BALANSE (Balances) form.



#	NAME	PRESENTATION	ACTUAL
1	account	БухРахунок	✓
2	contr_p_m	Рухи грошей по контрагентах	✓
3	corr_accout	Корреспонденські рахунки	✓

Scr12

Pay special attention to the first row: account – this is a reserved name for accounting balances. If you plan to keep track of balances by accounting accounts, this row must be present exactly as in the demo version (PRESENTATION – at your discretion).



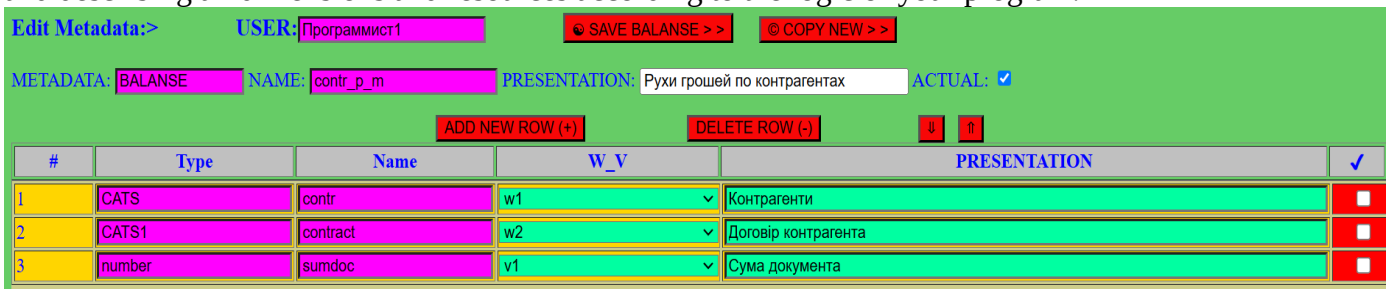
#	Type	Name	W_V	PRESENTATION	✓
1	CATS	accounts	w1	План рахунків	
2	number	sumdoc	v1	Сума документа	

Scr13

All other parameters will be automatically taken from the chart of accounts (Plan of accounts) — accounts (also a reserved identifier!).

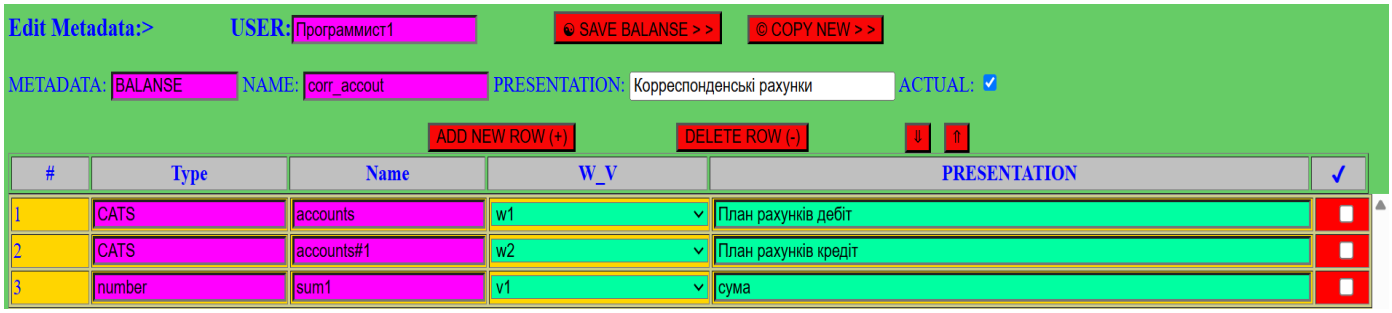
I want to emphasize that in My4S there is no separate register for accounting and other balance registers. Everything works as a single system of three tables, as described above. Accounting balances differ only in that the first dimension (W1) is always a reference to the chart of accounts (Plan of accounts). In combination with the reserved identifiers account and accounts, the code itself determines how to handle accounting totals.

After that, you can create any number of simple balance registers, assigning them your own invented identifiers and describing all dimensions and resources according to the logic of your program.



#	Type	Name	W_V	PRESENTATION	✓
1	CATS	contr	w1	Контрагенти	
2	CATS1	contract	w2	Договір контрагента	
3	number	sumdoc	v1	Сума документа	

Scr14

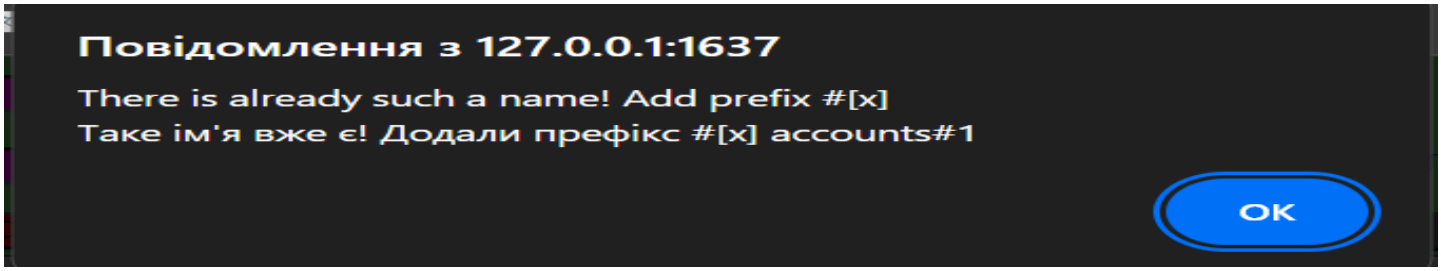


#	Type	Name	W_V	PRESENTATION	✓
1	CATS	accounts	w1	План рахунків дебіт	
2	CATS	accounts#1	w2	План рахунків кредіт	
3	number	sum1	v1	сума	

Scr15

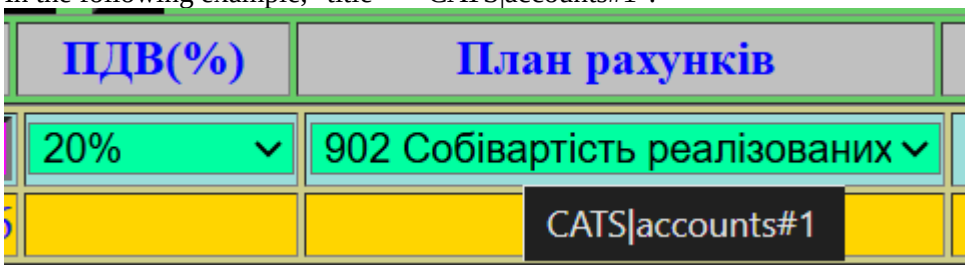
Why accounts and accounts#1.

You may wonder why accounts and accounts#1. This is another feature of mine. When in the Constructor you select a certain Type, you will see a ready-made list of identifiers for that type, and you will never see an identifier with the ending (#n). The reason is that this suffix (#n) is generated by the program itself. Before creating a new row, the program checks the Name of all rows for matches with the new identifier (ignoring the (#n) ending), and at the same time counts all matches. If necessary, the program will automatically append a new suffix to the identifier. Before doing so, it will issue a notification:



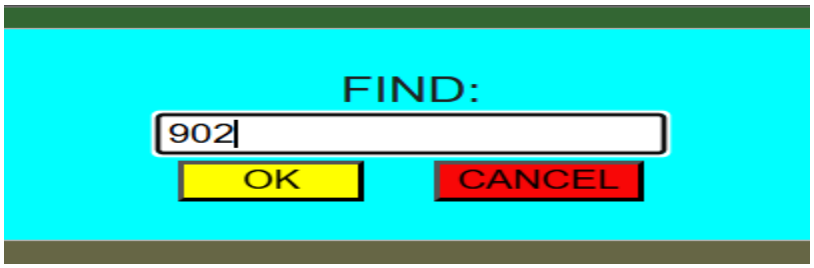
Scr16

This feature is used for searching in the Enterprise among different elements of directories (CATS) and more.
Each cell of an HTML table has a text parameter called "title".
If you assign a value to it, when hovering the cursor over that cell, the "title" text will appear below it.
This parameter is filled by the program at the moment the table is populated on the fly, based on the data recorded in the Constructor.
The parameter will be written into cells where there is the possibility of selecting from a directory.
The string will look like: METADATA|id_n.
In the following example, "title" = "CATS|accounts#1".



Scr17

And now, when searching for an element in the directory, the program knows in which directory to look for this element. To do this, the program discards the suffix (#1) when searching in the database (Select …) and in the filter you specify (WHERE).



Scr18

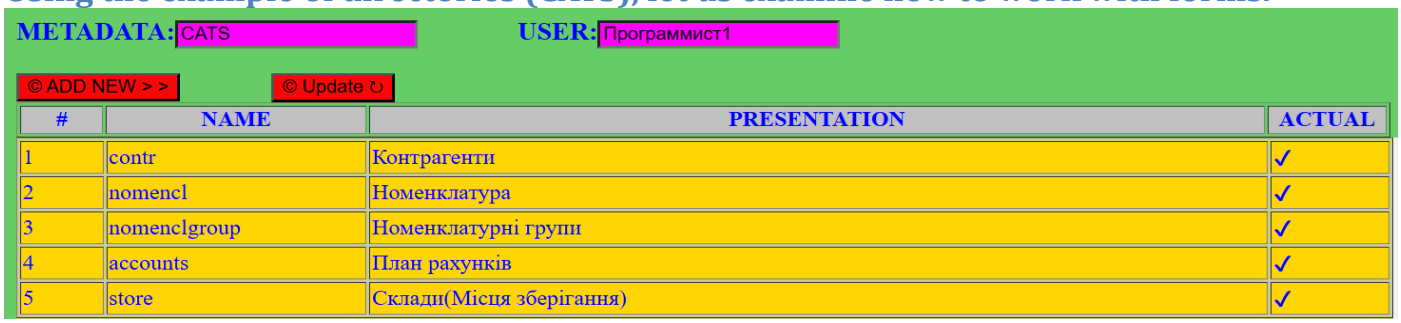
It returns a list of selected elements (no more than 50!) from which you choose the desired one,
and if there is only one element, it will immediately be placed into the selection cell!
In the search line, you can enter several parameters separated by spaces.
Parameters are strings, and the elements selected will be those in which all occurrences of the search string are found.
The search is performed by the element name or by its code.
There is a special rule: if the first character is a digit, the search is performed by the “cod” parameter; otherwise, by “name”!
Example: “Beer Dark 0.5” — in the search line you can type “beer dar 5” and you will get the correct result.
For barcode search, simply scan it into the search line, and the search will then be performed by the “cod” parameter.

An approximate example of a selection string when filtering from the Nomenclature:

Scr19

Working with the metadata configuration form.

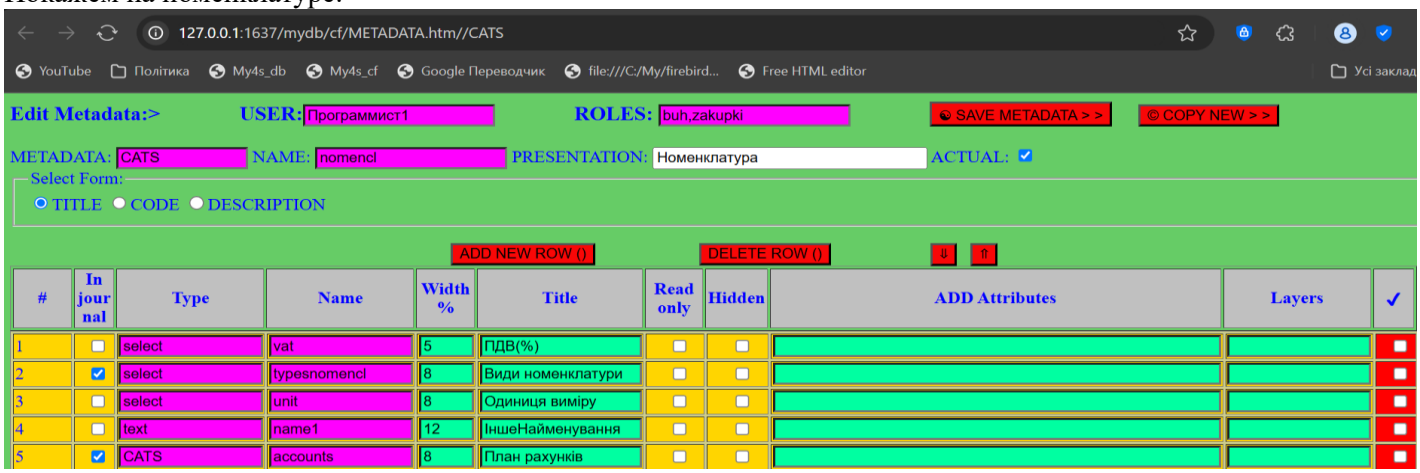
Using the example of directories (CATS), let us examine how to work with forms.



#	NAME	PRESENTATION	ACTUAL
1	contr	Контрагенти	✓
2	nomencl	Номенклатура	✓
3	nomenclgroup	Номенклатурні групи	✓
4	accounts	План рахунків	✓
5	store	Склади(Місця зберігання)	✓

Scr20

Для начала редактирования какого-либо справочника – сделайте двойной щелчок на нем.
Покажем на номенклатуре:

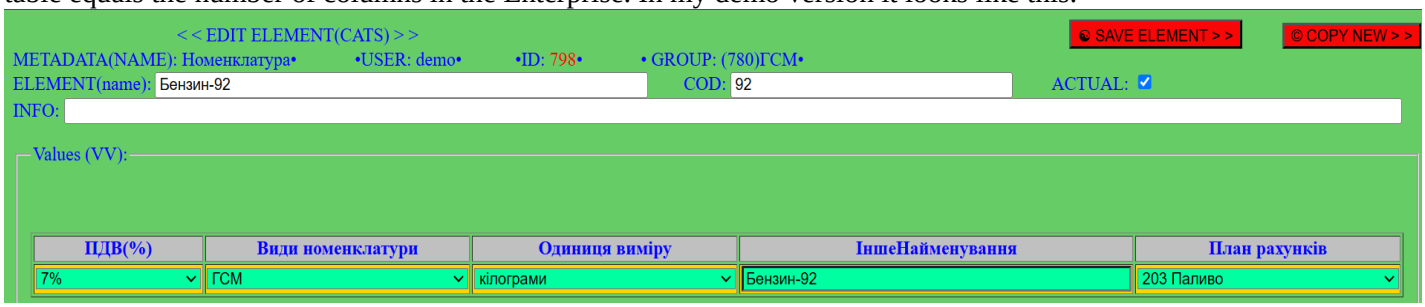


#	In journal	Type	Name	Width %	Title	Read only	Hidden	ADD Attributes	Layers	✓
1	☐	select	vat	5	ПДВ(%)	☐	☐			☐
2	☑	select	typesnomencl	8	Види номенклатури	☐	☐			☐
3	☐	select	unit	8	Одиниця виміру	☐	☐			☐
4	☐	text	name1	12	ІншеНайменування	☐	☐			☐
5	☑	CATS	accounts	8	План рахунків	☐	☐			☐

Scr21

Everything displayed in pink is closed for editing! Therefore, think carefully before creating any new objects.
Let me explain the “COPY NEW” button. When pressed, this form will be copied and created as a new form of metadata not yet saved in the database. In the “NAME” field, there will be a hint string “NEW METADATA!?”. You should replace it with the name of the metadata you have invented.
Everything marked in pink remains locked. However, you can delete a row and create a new one; with the new form you can experiment without consequences until you save it (SAVE). To delete rows, there is the “DELETE ROW()” button. By itself, pressing the button does nothing. You must indicate which rows to delete using the last column “checkbox,” where you mark the row(s) to be deleted, and then you can remove them. The checkbox is also used to move rows up or down with the corresponding buttons. But if you set two or more checkboxes, the move will not work!
Movement does not wrap around.
This is how it works in all forms of the Constructor and the Enterprise where there is a “checkbox” and the “COPY NEW” button!
Let us consider the Select Form: it contains radio buttons TITLE, CODE, DESCRIPTION. When I started, I thought of using all of them, but so far I have only needed TITLE. The others can be used to store some textual information in blob fields WW (CODE) and DESCRIPTION in the METADATA table.

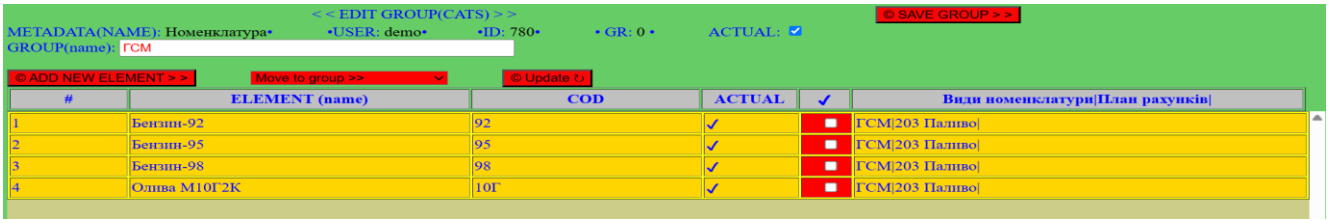
TITLE, as you may have noticed, is an HTML table (table). Each row describes one column in the tabular part of the table, showing how it will appear in the Enterprise when you open the Nomenclature directory. The number of rows in the table equals the number of columns in the Enterprise. In my demo version it looks like this:



Scr22

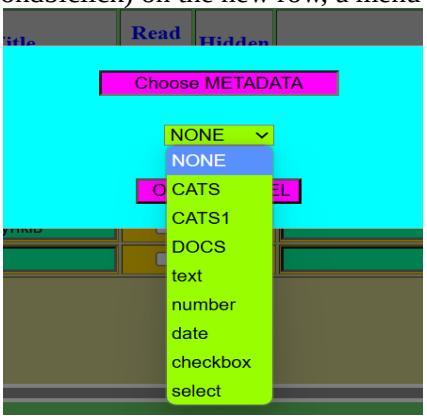
As I described earlier, what you see is an HTML template (ELEMENT.htm). The upper part is suitable for any directory, while the lower table will be defined in the Constructor to show how it looks for a specific directory. In this case, for the Nomenclature.
Let us return to the Constructor. Let us examine each column:

1. Sequential number
2. In journal – indicates whether to display information about the row in the journal. In the Enterprise, it will look like this:



Scr23

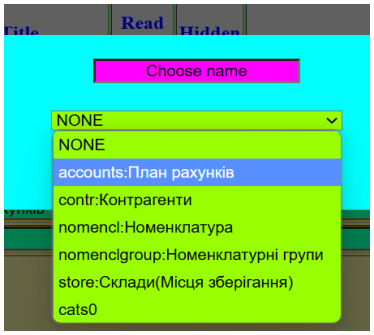
3. Type – can only be changed when adding a new row using ADD NEW ROW ().
When you double-click (ondblclick) on the new row, a menu will appear in the center:



Scr24

3. Name – depending on the choice made in the first menu, another menu will appear.
4.

For example, if you select CATS, then a list of all directories will be displayed.
cats0 – this is the transferred selection of the directory already in the Enterprise before starting the value selection.



Scr25

If you select CATS1 – this is a subordinate directory. It must always be paired with the main directory CATS.
First comes CATS and immediately after it CATS1, otherwise it will not work (e.g., Contractors, Contracts)!
If you select DOCS, then a list of all documents will appear.
Everything that comes after DOCS: text and further down the list – these are identifiers selected by type from the ID_DB table described earlier.

4. **Width %** – sets the width of the cell.
5. **Title** – the text of the table column header; if it does not fit, it will wrap inside the cell, expanding it vertically.
6. **Read only** – prohibits editing the cell content.
7. **Hidden** – determines whether the cell is visible.
8. **ADD Attributes** – you can additionally describe any attribute, except those listed above, according to HTML rules!

The full power of HTML is in your hands and knowledge. But as you can see, I have not yet needed to use this column's capabilities.

Quite a lot of what you can see in the demo version is recorded in the TabFix.css file, which you can also improve to your taste and style!

9. **Layers** – by default, all table forms stretch across the full width of the screen.

If there are few columns, this option is not needed.

When there are many columns, the browser will automatically adjust proportionally to the width you set (Width) so that all columns fit within the screen width.

In the Layers column, you write the name of the layer (just a string in any language).

If we want some columns to always be visible in every layer, we simply leave Layers empty.

The remaining columns are divided into layers, each with its own unique name (there can be several layers).

When the form is displayed in the Enterprise, a dropdown list of all layers will appear above the table on the left.

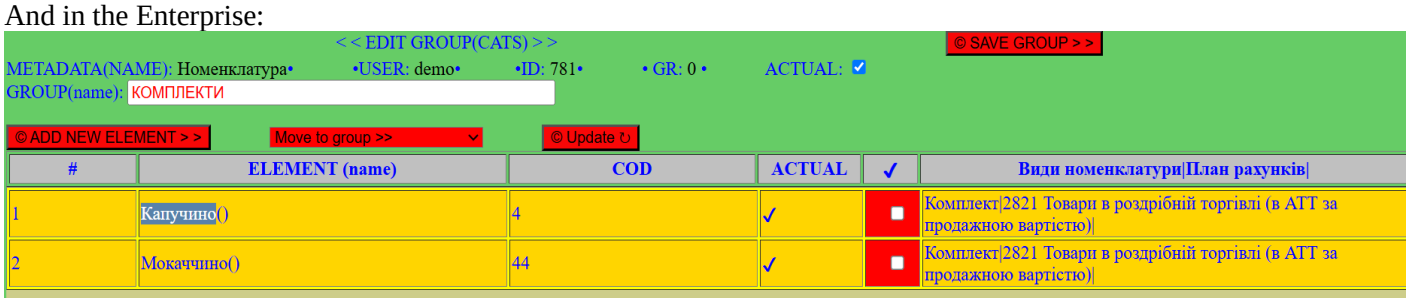
10. **Checkbox** – I have already explained earlier what it is used for.

Sets (Complects).

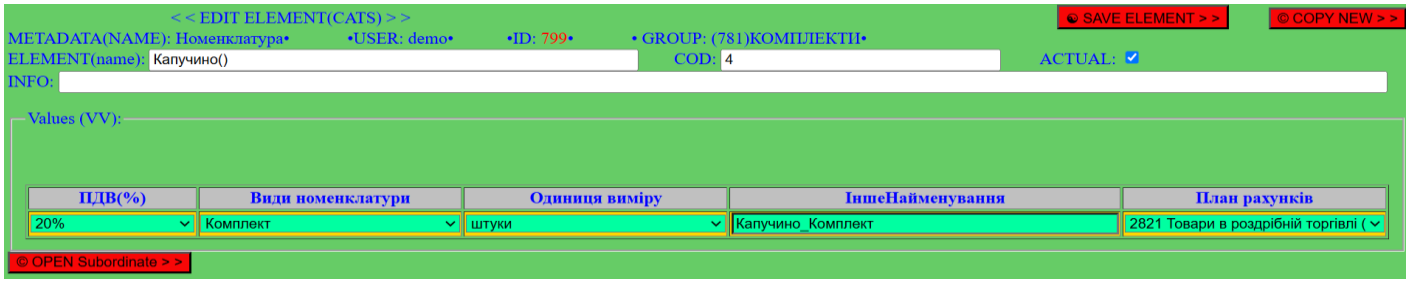
In My4S, a special subtype of directory called "Sets" is supported.
The indicator of a set in the Enterprise is the presence of opening and closing brackets "()" at the end of the directory element's name.
In the Constructor, a subordinate directory (CATS1) must be created and its columns defined.

Scr26

And in the Enterprise:

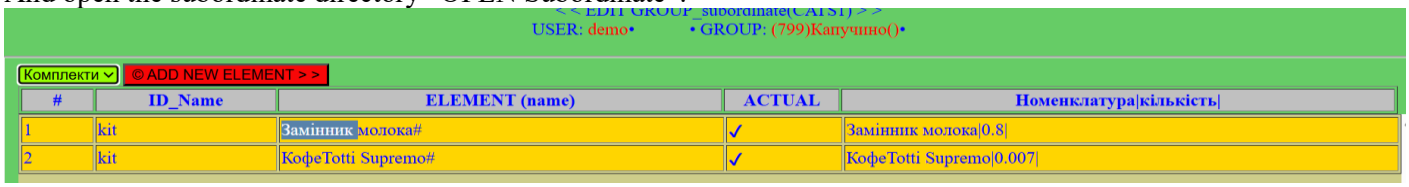


Scr27



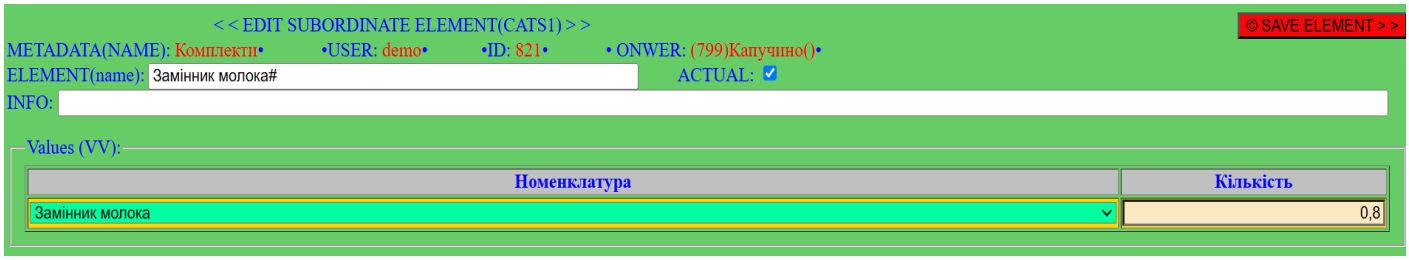
Scr27a

And open the subordinate directory “OPEN Subordinate”:



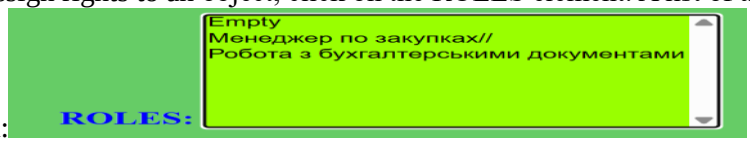
Scr28

And then edit or create a new element:



Scr29

At the end of the name of each element of a set in the subordinate directory, you must place the “#” symbol.
Let us clarify for all ROLES elements in the Constructor – if it is empty, this means full access for all users in the Enterprise to this form!
Remember, everything marked in pink cannot be changed (Read only). But there is an exception for some form elements. Consider the ROLES element. Formally, it is indeed Read only and therefore marked in pink.
However, we still need the ability to assign user rights to the object, allowing certain users to modify it
(and let us not forget: a user with the “all” right has full access to all metadata objects without restrictions, so there is no need to specify this right again!).
To assign rights to an object, click on the ROLES element. A list of all rights entered in the “Rights (Ролі)” menu will open:



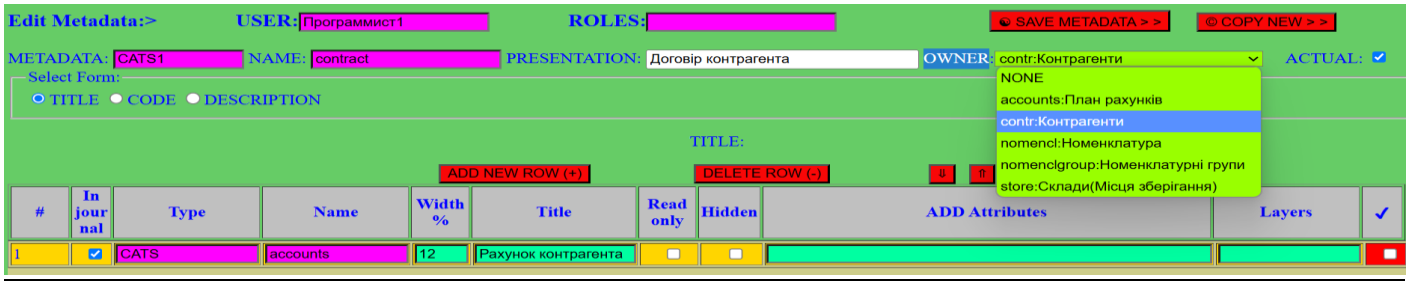
Scr30

If you need to correct or delete the list of already selected rights, click on Empty, and then click outside this small form – the list will be deleted!
Then click again on the ROLES element and, while holding down the Shift key, you can select several rights at once. After finishing the selection, click outside this small form – the ROLES field will display the list of selected rights separated by commas.

Almost everything described for directories (CATS) regarding working with forms will also apply to other forms in the Constructor.

Subordinate directory (CATS1).

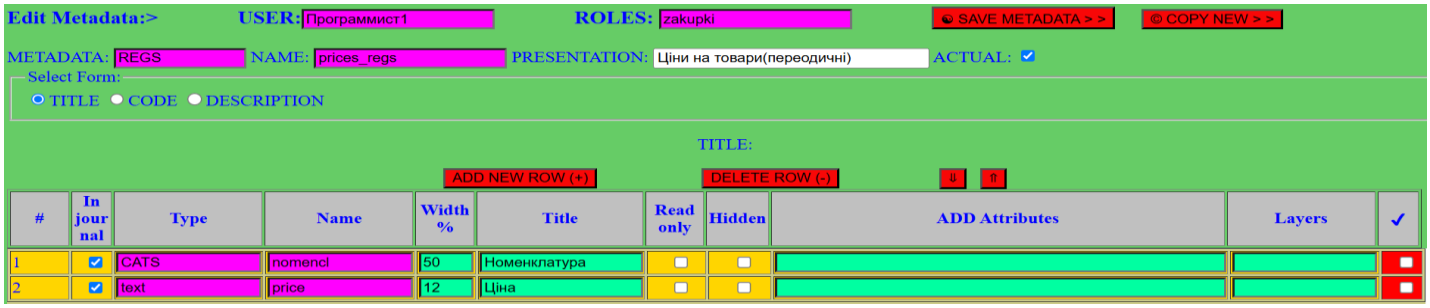
It is the same as CATS, except for one special feature – the OWNER, which is the name of the directory to which it is subordinate.



Scr31

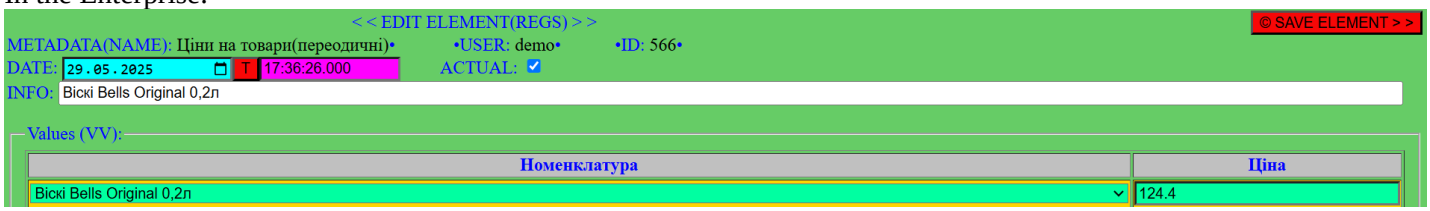
Information registers (REGS).

They are the same as CATS. For now, REGS differ from directories only in that,
in the Enterprise, when creating an element you can set a date,
and this automatically makes the information registers periodic!
In the Constructor:



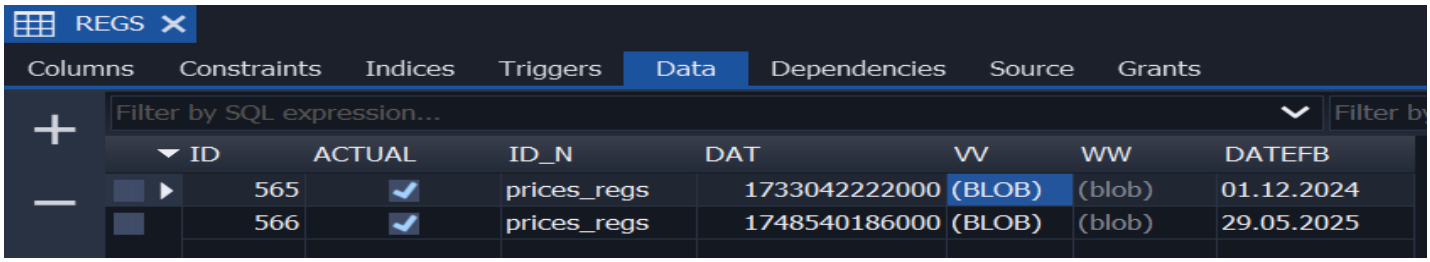
Scr32

In the Enterprise:



Scr33

In the firebird:



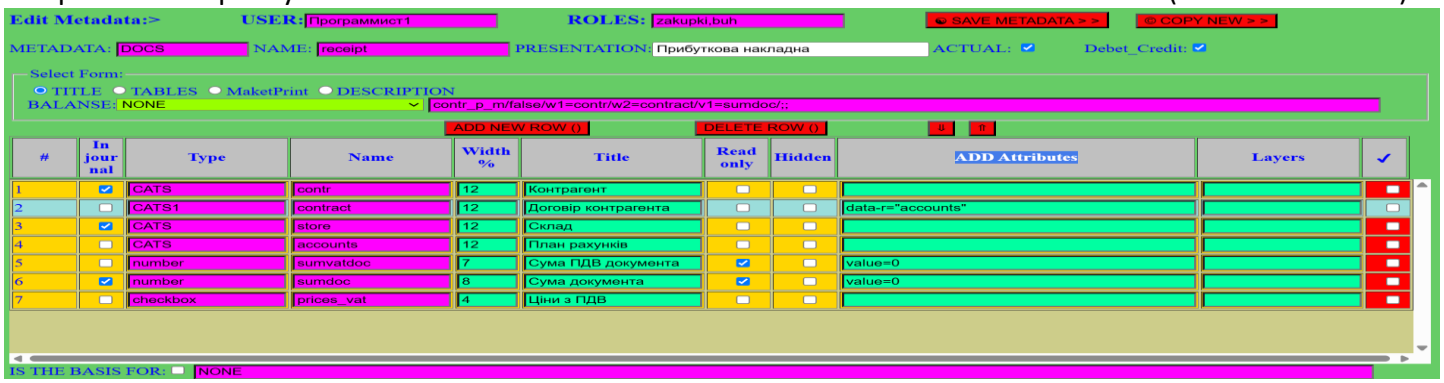
Scr34

Information registers are a separate object for storing data.
It is not possible to create a reference to them in metadata.
As you may have noticed, there is no such type in the Type selection!
Data can only be retrieved from Firebird by executing a (Select …) query!

Documents (DOCS).

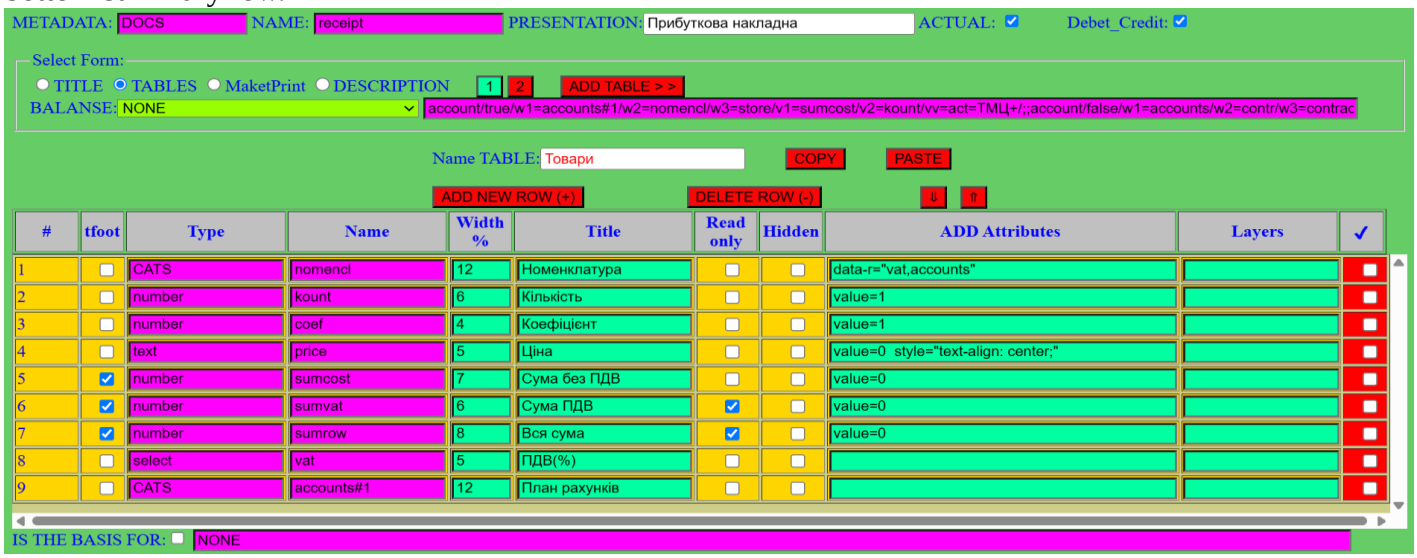
This is the most complex of the metadata objects. In addition to the header, it also has tabular sections (up to 7).

Documents describe all register movements.
It is possible to specify on the basis of which documents the current document is entered (IS THE BASIS FOR).



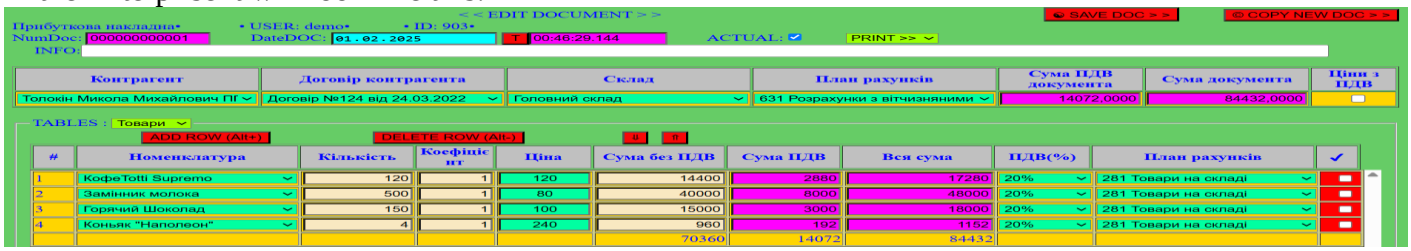
Scr35

Initial filling of tables, both in the header and in the tabular sections, is fully consistent with what was described above in CATS.
The only difference is that when filling tabular sections (TABLES), instead of the “In journal” column, a “tfoot” column appears.
This column specifies which columns in the Enterprise should be summed, and the total will be displayed in the bottom summary row.



Scr36

In the Enterprise it will look like this:



Scr37

Back to the Constructor:
Debet_Credit: indicates whether the document is primarily incoming (Debet) or outgoing (Credit).

If the checkbox is selected – it is incoming; otherwise, it is outgoing.
Select Form: a switch between the parts of the document we are currently working with.
Each document has a header (TITLE) and tabular sections (TABLES).
The header is a mandatory element, while tabular tables may or may not exist!
MaketPrint: this is a simple textarea field. When creating a new document, it is empty.
It is recommended to fill it at the very end of editing the document, preferably after testing the document in the Enterprise.
When opening this form, you will see two buttons: “Generate print” and “View print”.
The first button generates a text-based print form of the document in pure HTML code.
The second button shows how this form will look in the browser (only the structure).

In our case, since there are two tabular sections, two templates will be shown.

..txtt..

Контрагент: ..contr..
Договір контрагента: ..contract..
Склад: ..store..
План рахунків: ..accounts..

#	Номенклатура	Кількість	Коефіцієнт	Ціна	Сума без ПДВ	Сума ПДВ	Вся сума	ПДВ(%)	План рахунків
..#..	..nomencl..	..kount..	..coef..	..price..	..sumcost..	..sumvat..	..sumrow..	..vat..	..accounts#1..

Сума без ПДВ: ..5..
Сума ПДВ: ..6..
Вся сума: ..7..

//TAB

..txtt..

Контрагент: ..contr..
Договір контрагента: ..contract..
Склад: ..store..
План рахунків: ..accounts..

#	Номенклатура	Кількість	Ціна	Сума без ПДВ	Сума ПДВ	Вся сума	ПДВ(%)	План рахунків
..#..	..nomencl..	..kount..	..price..	..sumcost..	..sumvat..	..sumrow..	..vat..	..accounts#1..

Сума без ПДВ: ..4..
Сума ПДВ: ..5..
Вся сума: ..6..

//TAB

Scr38

In the Enterprise, based on this template, the print form will be generated,
already filled with the document's data.
If the document has several tabular sections, the print form will be generated only for the table
whose tab is currently selected!

Прибуткова накладна #000000000004 > 2025-03-02 15:03:04.241(Товари)

Контрагент: БестТрейд
Договір контрагента: Договір №250 від 20.10.2020
Склад: Головний склад
План рахунків: 631 Розрахунки з вітчизняними постачальниками

#	Номенклатура	Кількість	Коефіцієнт	Ціна	Сума без ПДВ	Сума ПДВ	Вся сума	ПДВ(%)	План рахунків
1	Коньяк "Наполеон"	120	1	240	28800	5760	34560	20%	281 Товари на складі
2	Горілка Козацька рада "Оригінальна" 0,75 л	210	1	100	21000	4200	25200	20%	281 Товари на складі

Сума без ПДВ: 49800.00
Сума ПДВ: 9960.00
Вся сума: 59760.00

Scr39

Editing the textarea field

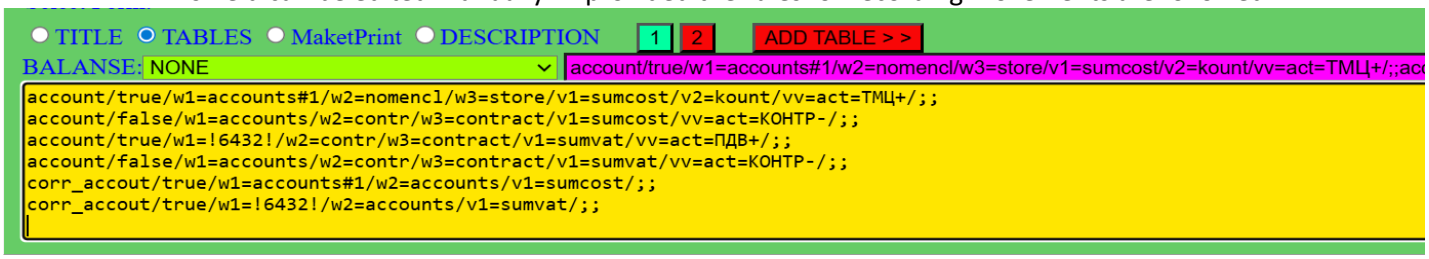
You can freely edit the text in the textarea field as you wish.
When saving the document in the Constructor, your changes will be recorded.
If the editing result does not satisfy you, you can cancel the changes by deleting all the text
in the field (Ctrl+A → Delete),
then click the "Generate print" button – a new (initial) text will be automatically generated.
To return to editing the document, simply click on its title.
In the Constructor this will be "..txtt..", in the Enterprise system – "Incoming invoice …".

Column "ADD Attributes"
Next to the parameter "Price" it says: value=0 style="text-align: center;".
For those familiar with HTML, this is obvious. For others: value — the default value set when opening the form.
style="text-align: center;" — defines alignment of the value in the center of the cell. This is part of the HTML standard,
not a custom development.

Attribute data-r="vat,accounts"
This is my own enhancement – data-r is a reserved identifier.
In the Enterprise, when selecting a nomenclature in a row, the program checks if such an identifier (data-r) exists.
If it does, the program checks whether the nomenclature has identifiers (vat,accounts).
If found, the program remembers these values.
And if it finds the same identifiers in the document row, they will be automatically filled in!
They can, of course, be changed later. If this is not clear right away, you will understand in the demo version.

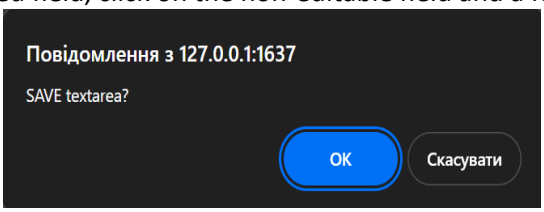
Tab "DESCRIPTION"
This is simply a textarea field where you can enter arbitrary text: Help, comments, hints, etc.
This text is not used anywhere except in this tab.
Description of register movements.
This stage is crucial in working with documents.
You should proceed to describing movements only after the document has been created in the Constructor.
In the header and tabular sections of the document, there is a BALANSE field with a dropdown list.

Next to it is a text field where lines describing register movements are automatically recorded (separated by ;;).
This field is visually highlighted and is not intended for direct editing.
However, by double-clicking on it, an additional textarea field will open, where all movements are displayed line by line.
This field can be edited manually — provided the rules for recording movements are followed.



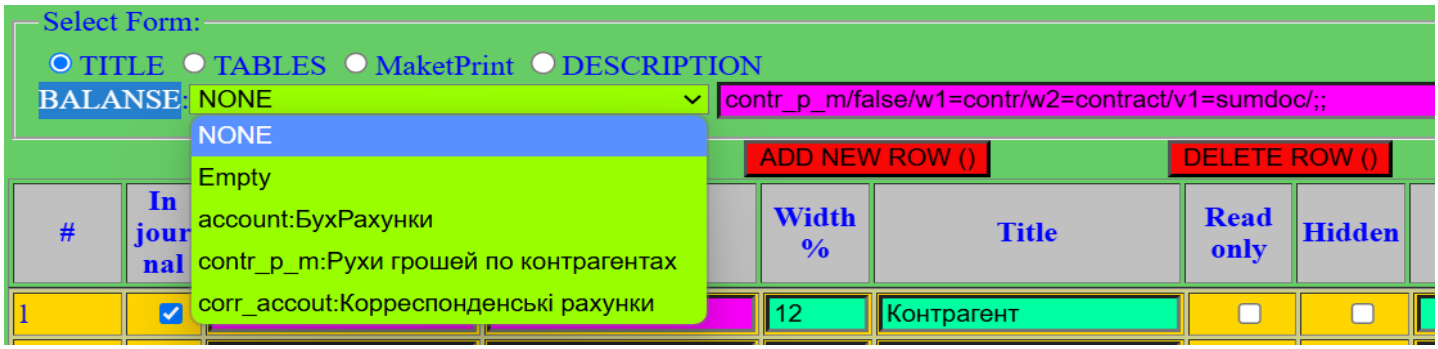
Scr40

To save the edited field, click on the non-editable field and a form will appear:



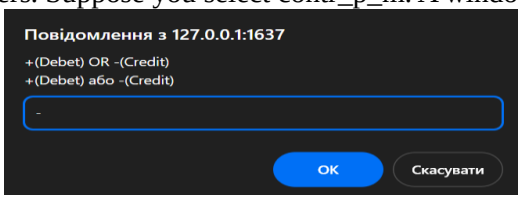
Scr41

If you click OK, the edited data will appear in the non-editable field as a single line separated by “;;”.
Now let’s see how this works:



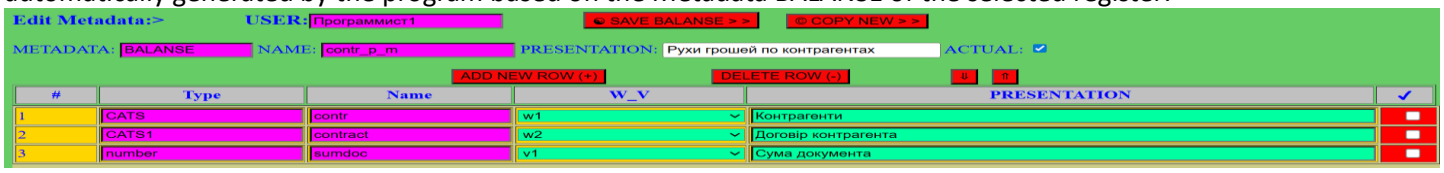
Scr42

In the BALANSE dropdown list, a menu of all actions for forming movements is presented.
If you click Empty, the non-edited line will be cleared, and it cannot be easily restored — you will have to fill in the register movements from scratch!
Next, a full list of balance registers will be presented, as described in Metadata BALANSE.
Let’s start with the balance registers. Suppose you select contr_p_m. A window for entering values will appear:



Scr43

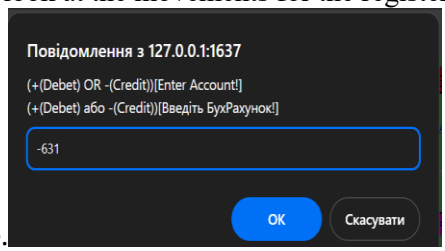
Here we must specify a single symbol +(Incoming) or –(Outgoing). If you enter any other symbol – it is an error!
Then press Enter or OK, or if you want to cancel the action – press Esc or Cancel.
If everything is correct, a new line will appear in the non-editable field
(contr_p_m/false/w1=contr/w2=contract/v1=sumdoc/;;)
automatically generated by the program based on the Metadata BALANSE of the selected register:



Scr44

Based on this line, the program will automatically generate register movements when saving the document in the Enterprise.
The number of such lines separated by “;;” determines how many register movements will be created for the given tab (TITLE or selected TABLES).

Now let’s look at the movements for the register “account:BuhRakhunky (Accounting Accounts)”. Here there is a small difference.



Scr45

Сначала также пишем «+» или «-», но сразу же после символа пишем (без пробела!) счёт,
по которому предположительно будет формироваться движение.
Из справочника счетов (В Предприятии) будут браться данные для автоматического формирования строки,
описывающей бухДвижения.
Что-то типа такого:

BALANSE: account:БухРахунки | account/true/w1=accounts#1/w2=nomencl/w3=store/v1=sumcost/v2=kount/\

```
account/true/w1=accounts#1/w2=nomencl/w3=store/v1=sumcost/v2=kount/vv=act=ТМЦ+/;;
account/false/w1=accounts/w2=contr/w3=contract/v1=sumcost/vv=act=КОНТР-/;;
account/true/w1=!6432!/w2=contr/w3=contract/v1=sumvat/vv=act=ПДВ+/;;
account/false/w1=accounts/w2=contr/w3=contract/v1=sumvat/vv=act=КОНТР-/;;
```

Scr46

Уже выше я описывал, что значит accounts и accounts#1, повторяться не буду.
Но что касается движений регистров, уточню.
Когда программа при записи документа построчно читает строки в табличных частях, то одновременно она видит всю шапку документа,
но другие табличные части (если они есть) не видит.
Поэтому, если в шапке документа есть какой-то идентификатор, в данном случае accounts,
то accounts#1 (любой #n) может быть в каждой табличной части, и они никак не будут конкурировать.
В момент создания документа в Конструкторе программа сама позаботится, чтобы правильно расставить префиксы #n.
А вот при описании движений регистров программа может иногда неправильно расставить идентификаторы с #n.
Потому доверяй, но проверяй (штучный интеллект тоже ошибается!)!
Вот для чего у вас есть возможность подкорректировать то, что создала программа.

А теперь разберём, что значит «!6432!».
Иногда мы можем наверняка знать бухгалтерский счёт для данного движения.
И тогда зачем забивать документ лишней колонкой со ссылкой на счёт?
Просто мы можем этот счёт прописать непосредственно в описании движений, обрамив этот счёт восклицательными знаками.

Строка «act=ТМЦ+/», здесь ТМЦ+(Приход) — краткое описание операции.
Я такое придумал, вы можете написать своё.
В Предприятии эта строка будет использована в печатной форме BALANSE, которая есть в каждом документе,
и показывает все движения регистров остатков документа.

Прибуткова накладна #000000000006 > 2025-07-03 20:18:57.000

БухРахунки(account)

w1/w2/w3/w4/w5	v1/v2/v3/v4/v5	ACT
281 Товари на складі / КофеТotti Supremo / Головний склад	14400/120/0/0/0/	ТМЦ+
631 Розрахунки з вітчизняними постачальниками / БестТрейд / Договір №258 від 20.10.2019	-14400/0/0/0/0/	КОНТР-
6432 Податкові зобов'язання непідтверджені / БестТрейд / Договір №258 від 20.10.2019	2880/0/0/0/0/	ПДВ+
631 Розрахунки з вітчизняними постачальниками / БестТрейд / Договір №258 від 20.10.2019	-2880/0/0/0/0/	КОНТР-
281 Товари на складі / Замінник молока / Головний склад	40000/500/0/0/0/	ТМЦ+
631 Розрахунки з вітчизняними постачальниками / БестТрейд / Договір №258 від 20.10.2019	-40000/0/0/0/0/	КОНТР-
6432 Податкові зобов'язання непідтверджені / БестТрейд / Договір №258 від 20.10.2019	8000/0/0/0/0/	ПДВ+
631 Розрахунки з вітчизняними постачальниками / БестТрейд / Договір №258 від 20.10.2019	-8000/0/0/0/0/	КОНТР-
281 Товари на складі / Горячий Шоколад / Головний склад	15000/150/0/0/0/	ТМЦ+
631 Розрахунки з вітчизняними постачальниками / БестТрейд / Договір №258 від 20.10.2019	-15000/0/0/0/0/	КОНТР-
6432 Податкові зобов'язання непідтверджені / БестТрейд / Договір №258 від 20.10.2019	3000/0/0/0/0/	ПДВ+
631 Розрахунки з вітчизняними постачальниками / БестТрейд / Договір №258 від 20.10.2019	-3000/0/0/0/0/	КОНТР-
281 Товари на складі / Коньяк "Наполеон" / Головний склад	400/2/0/0/0/	ТМЦ+
631 Розрахунки з вітчизняними постачальниками / БестТрейд / Договір №258 від 20.10.2019	-400/0/0/0/0/	КОНТР-
6432 Податкові зобов'язання непідтверджені / БестТрейд / Договір №258 від 20.10.2019	80/0/0/0/0/	ПДВ+
631 Розрахунки з вітчизняними постачальниками / БестТрейд / Договір №258 від 20.10.2019	-80/0/0/0/0/	КОНТР-

Scr47

Корреспонденські рахунки(corr_accout)

w1/w2/w3/w4/w5	v1/v2/v3/v4/v5	ACT
281 Товари на складі / 631 Розрахунки з вітчизняними постачальниками	14400/0/0/0/0/	
6432 Податкові зобов'язання непідтверджені / 631 Розрахунки з вітчизняними постачальниками	2880/0/0/0/0/	
281 Товари на складі / 631 Розрахунки з вітчизняними постачальниками	40000/0/0/0/0/	
6432 Податкові зобов'язання непідтверджені / 631 Розрахунки з вітчизняними постачальниками	8000/0/0/0/0/	
281 Товари на складі / 631 Розрахунки з вітчизняними постачальниками	15000/0/0/0/0/	
6432 Податкові зобов'язання непідтверджені / 631 Розрахунки з вітчизняними постачальниками	3000/0/0/0/0/	
281 Товари на складі / 631 Розрахунки з вітчизняними постачальниками	400/0/0/0/0/	
6432 Податкові зобов'язання непідтверджені / 631 Розрахунки з вітчизняними постачальниками	80/0/0/0/0/	

Рухи грошей по контрагентах(contr_p_m)

w1/w2/w3/w4/w5	v1/v2/v3/v4/v5	ACT
БестТрейд / Договір №258 від 20.10.2019	-83760/0/0/0/0/	

Scr48

For now, AST works only for accounting.
Expenses are always marked with a minus sign, not only in the report but also in Firebird tables!
When collapsing (aggregating) the data, there is no need to manipulate or adjust.

Operation of the mechanism for creating a document based on another in the Enterprise system

Now let's describe how the "IS THE BASIS FOR" mechanism works.
I tried to automate this process as much as possible.
If we decide that another document should be created based on the one being edited,
we check the box after IS THE BASIS FOR: and a window will open with a list of all documents in the database.
Select one or several documents (using Shift), then close the checkbox,
and your selected document(s) will be placed into the text field:

Scr49

If you need to correct the data, click the checkbox and select the documents again.
At this point, your task in the Constructor will be finished.

Let me briefly explain how this is implemented and what role the programmer plays here.
If the structures of the source and new documents are similar (including matching identifiers),
the system will automatically generate the document correctly.
You may only need to manually fill in one or two elements.
However, if the structures differ (for example, when creating a tax invoice),
programmer intervention is required.
All necessary conditions are provided to ensure your work is efficient.

Document check on opening

When opening a document, the system checks whether it is created based on another document.
In window.onload = function(), the following code is used:

```
            if (myForm.e_title.value === "BASIS") {

                if (typeof MyFunctBASIS == "function") {

                    MyFunctBASIS();

                } else doLoadDoc();
```

```
            }
```

The program checks whether such a function (MyFunctBASIS) exists in the document's context.
By default, this function does not exist, and the program will attempt to create the document based on another as best as it can.
MyFunctBASIS is a reserved function name.
Each object form has the ability to attach one or more modules,
which will be included in the context of this object using SRC:

```
    <script src="Myjsjs/DOCS/id_n__/my.js"></script>
```

my.js is entirely your function, and you are responsible for its correct operation.
Returning to the "based on" document: if your module contains the MyFunctBASIS function, it will be executed,
and with its help you will manipulate the document at the moment of its loading
(for a proper understanding of how to handle this, look at how it works in the doLoadDoc function in the All_db.js module regarding "BASIS"!).
There will be more than one basic function like MyFunctBASIS, which will allow you to manipulate the object both at the beginning of opening or saving, and at the end. I will describe this later.
If you get to know My4s better, you will be able to create such functions yourself and describe their processing (this is not a program where the main part is locked away in closed files like dll, exe, and others!).

About 4SON

Now it is time for me to confess that at the beginning I told you a small untruth about JSON in blob fields. If I had based the system on PostgreSQL, most likely it would have been true, but I used Firebird as the foundation, where built-in JSON support only appeared in version 5, while I started with version 3.

Therefore, I came up with my own method of storing data in blob fields and extracting it from there. It is a delimiter-based string. The thing is, putting JSON into a blob field is no problem (JSON.stringify), but retrieving data from it in a Select turned out to be quite problematic. I had to write a function similar to JSON.parse..

I even tried to do this after everything was already working with the delimiter string (me vs JSON!), and my function (Get_Blob) was already operational. But when I started writing my function (Get_JSON), I realized that the speed of retrieving data from JSON would be many times faster than from a delimiter string. And this function is used in all Select statements, and more than once!

So I decided to leave everything as it was, even though I understood that JSON would have been immediately clear to anyone reading this text, and I wouldn't have needed to explain anything, as I am trying to do now.

Here is a small example from the module text:

```
let strselect_ = "Select GET_BLOB(vv,'t_name') as acc, GET_BLOB(vv,'t_name_') as acc1 From
MDMD WHERE id = " + select_one;

strselect_ = strselect_.replace("MDMD", t.title.split("|")[0]);

strselect_ = strselect_.replace("t_name", t_name).replace("t_name", t_name);
```

If things are not yet fully clear (the example is taken out of context), the main point is to know that there is such a function in Firebird as **GET_BLOB(vv, 't_name')**. Here, *vv* is the blob field, and *'t_name'* is the name of the parameter in the string *vv* that we are searching for.

There can also be a third parameter – *separator*. If it is not provided, the default separator is used. In my case, the separator can consist of one to three arbitrary characters. For now, I have only needed three levels and, naturally, three separators.

```
const spr1 = "1~@", spr2 = "2~@", spr3 = "3~@";
```

The first **"1~@"** always goes by default!

Now let us move on to the description of the delimiter string – let us call it **4SON**. In the string, there can be multiple nested levels, each with its own delimiters. Here is the structure of **4SON** with two levels:

«name1=param1[=name11=param11spr2name12=param12spr2…]spr1name2=param2[=name21=param21spr2name22=param22spr2…]spr1…»

Do not be alarmed, everything will turn out to be much simpler! The main thing to understand is that there are **no spaces** at all. Parameters can contain any text, but the text cannot include a delimiter, and the text is not enclosed in quotes unless required by the content itself.

The type of the string (**string or number**) must already be known when writing a **Select**, or determined programmatically. If it is a number, then in the **Select** you will need to use the function **GET_BLOB4**, which returns a number.

I designed these delimiters so that the chance of them appearing naturally in the text would be minimal. In addition, the program in **Enterprise** strictly controls the text entered by operators to prevent the appearance of these delimiters!

To begin, let us look at a simple string with one level from the program:

«***nomencl=8001~@nomencl_=Замінник молока1~@store#1=7771~@store#1_=Головний склад1~@***»

Let me digress a little from the topic and explain what **"nomencl"** and **"nomencl_"** mean. The identifier without the final underscore (as named in the Constructor) is the **id of the directory** (remember – the id is unique for any directory, e.g. *nomencl=800*). The same identifier with an underscore at the end represents the **ready-made name of that directory element**. In some reports, this can significantly speed up the execution of a query (**Select…**).

Now let us return to **4SON** and continue. For example, let us take a string with simple single-character delimiters, which is located in the **blob field vv** with three levels:

a=000*aa=b1=111/b2=222/bb=c1=444&c2=555&/a1=333**

- The delimiter of the **first level** is "*".
- In the **second level**, the delimiter is "/".
- In the **third level**, the delimiter is "&".

To obtain the string of the first level, it is simple: GET_BLOB(vv, 'a', '*') = 000

And in the same way we can get the string: GET_BLOB(vv, 'a1', '*') = 333

To obtain the second level, first we get the string in which the second level is located:

GET_BLOB(vv, 'aa', '*') = `b1=111/b2=222/bb=c1=444&c2=555&/`

Here we can get the string: GET_BLOB(GET_BLOB(vv, 'aa', '*'), 'b2', '/') = 222

And to obtain **c1** and **c2** (this is already the third level with the delimiter "&"), we first get the string "bb" in which the third level is located:

GET_BLOB(GET_BLOB(GET_BLOB(vv, 'aa', '*'), 'bb', '/') = c1=444&c2=555&/

GET_BLOB(GET_BLOB(GET_BLOB(GET_BLOB(vv, 'aa', '*'), 'bb', '/'), 'c1', '&') = 444
GET_BLOB(GET_BLOB(GET_BLOB(GET_BLOB(vv, 'aa', ''), 'bb', '/'), 'c2', '&') = 555

The same **GET_BLOB** function that exists in Firebird is also present in the main program written in **JavaScript**. Here is an example that clearly demonstrates everything I tried to explain a little earlier:

```
let vv = "a=000*aa=b1=111/b2=222/bb=c1=444&c2=555&/*a1=333*";

let a = GET_BLOB(vv, 'a', '*'); //000

let a1 = GET_BLOB(vv, 'a1', '*'); //333

let aa = GET_BLOB(vv, 'aa', '*'); //b1=111/b2=222/bb=c1=444&c2=555&/

let b1 = GET_BLOB(aa, 'b1', '/'); //111

let b2 = GET_BLOB(aa, 'b2', '/'); //222

let bb = GET_BLOB(aa, 'bb', '/'); //c1=444&c2=555&

let c1 = GET_BLOB(bb, 'c1', '&'); //444

let c2 = GET_BLOB(bb, 'c2', '&'); //555

//А можно c2 получить одной строкой:

c2 = GET_BLOB(GET_BLOB(GET_BLOB(vv, 'aa', '*'), 'bb', '/'), 'c2', '&'); //555

//И даже так можно, если вы уверены, что 'bb=' один в строке

c2 = GET_BLOB(vv.split("bb=")[1], 'c2', '&'); //555
```

In fact, you do not need to know how **4SON** works, and the program will function correctly on its own. But I believe that programmers who wish to improve the program should understand it.

I fully realize that my program is still far from perfect. That is why I will be glad if someone helps me improve it, or at least enhances it for their own use.

🗀 Description of the Project File Structure in Node.js

Let us move on to the description of the files on which Node.js operates. All of them are located on the server (in the demo version – most likely on your computer) in the My4s folder. The location of this folder is chosen by you.

For now, we will skip the description of the files located in the root – some of them have already been mentioned, others will be considered later. Let us start with the folders:

🗀 **.vscode** This folder contains the **launch.json** file, which specifies the path to **node.exe** when running the application through **Run** in **VSCode**. If Node.js is already installed on the server, everything will work correctly. If you are using only the **node.exe** file (for example, from the root of **My4s**), you need to uncomment the line:

"runtimeExecutable": "c:/My/My4s/node.exe",

and specify the full path to **node.exe**.
Example content of **launch.json**:

```
{

    // Используйте IntelliSense, чтобы узнать о возможных атрибутах.
```

```
    // Наведите указатель мыши, чтобы просмотреть описания существующих атрибутов.

    // Для получения дополнительной информации посетите:
https://go.microsoft.com/fwlink/?linkid=830387

    "version": "0.2.0",

    "configurations": [

        {

            "type": "node",

            "request": "launch",

            "name": "Launch Program",

            //"runtimeExecutable": "c:/My/My4s/node.exe",

            "program": "${workspaceFolder}\\Start4s.js",

        }

    ]

}
```

🗁 **MySnipp.code-snippets** – A very useful file that allows you to configure your own dropdown hints (**IntelliSense**) in **VSCode**.

🗁 **cf** – Contains **.htm** file templates for working in the **Constructor**. 🗁 **db** – **.htm** templates for working in **Enterprise**.

🗁 **css** – **.css** styles. For now, only one file is used – **TabFix.css**. 🗁 **js** – **.js** scripts connected in HTML files via **src**. Used only on the client side. 🗁 **node_modules** – Contains **.js** modules used for routing. Used only on the server side.

🗁 **qq** – An empty folder, but it is used for exchanging information between **Firebird** and **Node.js**.

All the folders and files listed above are **static**. They work with any database, and there can be several databases on the server.

Example of a connection string in the browser: **127.0.0.1:1637/mydb/cf**

Here, **mydb** is the name of the database being connected to. It must match the name of the folder inside **My4s**. In my case, it is **mydb**, but you can set your own name.

🗀 **Database Folder Structure**
🗋 **MyBase.json** – Located in the root of the database folder. It contains the main settings.

```
{
    "version_cst": "1.0.0", //Версия Конструктора
    "name": "mydb",
    "DSN": "mydb9", //имя DSN подсоединения к драйверу ODBC Firebird как настроите в windows
    "FDB": "C:/fb/db4s/MYDB9.FDB", //полный путь к базе на сервере
    "FBK": "C:/fb/db4s/MYDB9.FBK", //полный путь к файлу becap на сервере
```

```
    "Logs": "C:/My/My4S/mydb/logs/", //полный путь к папке Logs на сервере конкретной базы
    "AllLogs": "C:/My/My/logs_mydb/", //полный путь к папке AllLogs,где хранятся сарые logs
    "FB": "'c:/Program Files/Firebird/Firebird_3_0/'" //полный путь к папке Bin Firebird
}
```

At the moment, only **"version_cst"** and **"DSN"** are used. The other parameters are reserved for the future.

*Automatically Generated .js Files

When metadata is saved in the **Configurator**, the program automatically creates or overwrites **.js** files. These files contain ready-made **SELECT... (Ajax)** queries and are connected to HTML forms using **src**.

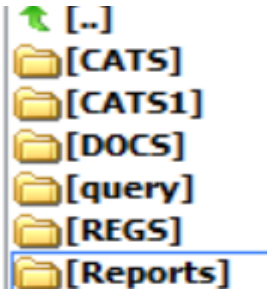
These files are stored on the **client side**, which significantly speeds up working with the database.

```
    <script src="Mydbdb/BalansXX.js"></script>

    <script src="Mydbdb/AccountXX.js"></script>

    <script src="Mydbdb/ID_DBXX.js"></script>
```

Internal Database Folders

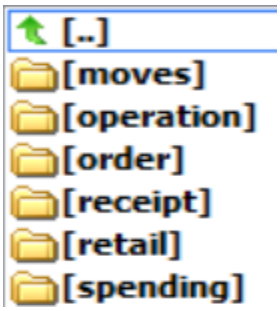
cf folder – .htm templates and other files for working in the Constructor.

Progv.us – A **JSON** file with user settings for the **Constructor**. Already described above. **Pass.htm** and **Pass2.htm** – Templates for entering and confirming a password. Universal for any database, but can be adapted if desired.
index.js – The routing file. Suitable for any database. **Conf4S.htm** – The main form opened in the browser. A universal form for all databases.
db folder – **.htm** templates and other files for working in **Enterprise**. All files in the root of this folder are suitable for working with any database.
Now let us examine the
Myjsjs folder. All data in this folder is generated automatically when working with metadata in the **Constructor**. The folder is multi-level. First come all the main metadata:



Scr50

And inside each folder there are subfolders named after all the metadata identifiers subordinate to the main folder. For now, let us look at one example: **DOCS**.



Scr51

The folder names correspond to my demo version of the database (**mydb**). In your database, they may be different! In each such folder, there will always be one file with the same name **myXX.js**. This file always con-

tains a single line: **xml_trXX=`[77,68,61,68,79,67,83,47….]** It is generated automatically, and you should not attempt to modify it!

If you accidentally change this file and the metadata stops loading correctly, you need to go into the **Configurator**, open the form where the metadata is described, toggle the **ACTUAL** checkbox on and off, and then save (**Save**) again. The file will be restored.

The variable **xml_trXX** stores ready-made data as if obtained from the database via **SELECT... (Ajax)**. The numbers separated by commas are character codes in the **Firebird** database encoding (in my demo it is *windows-1251*). After that, there is a function for re-encoding into **UTF-8** – **loadXMLString(xml_trXX)**.

Using this data, when the form is opened, the **.htm template** will be dynamically appended, and only then can the template be filled with data. This works the same way as I described for the **.js files** in the root of the database folder. The difference is that this file is strictly tied to the metadata.

In the same folder, you can also place **js modules** in which you add the required functionality for working with this metadata without breaking the initial settings. These file(s) will be connected to the metadata template using **src**.

How src Works for Me It is time to explain how **src** works in my setup. I will show this using the example of the template file **DOCS_db.htm**:

```
    <script src="Blob.js"></script>
    <script src="All_db.js"></script>
    <script src="Find4s.js"></script>
    <script src="Formuls.js"></script>
    <script src="print.js"></script>
    <script src="Myjsjs/DOCS/id_n__/myXX.js"></script>
    <script src="Mydbdb/BalansXX.js"></script>
    <script src="Mydbdb/AccountXX.js"></script>
    <script src="Mydbdb/ID_DBXX.js"></script>
    <script src="Myjsjs/DOCS/id_n__/my.js"></script>
```

The program is configured to correctly recognize routes to **src** files. I use three types of **src connections**:

1. **Direct reference to a .js file name.** The program always searches in the 🗀 **js** folder.
2. **Searching for a file in the folder of a specific metadata.** Example: `src="Myjsjs/DOCS/id_n__/myXX.js"`. Since the template is used for multiple metadata, it is necessary to specify where to look for the **.js** file. Therefore, the **id_n__ marker** is used. When preparing to send the form from the server to the client, the name of the required metadata identifier will be substituted here. In this way, you can connect not only the **myXX.js** file but also any other file located in this folder (possibly written by you). Example: `<script src="Myjsjs/DOCS/id_n__/my.js"></script>`.
3. **Connecting files located in the root folder of the database you are currently working with.** Example: `<script src="Mydbdb/BalansXX.js"></script>`. Here, **Mydbdb** is also a marker, which will be replaced with the name of the database.

As you can see, for each type of connection, you can place your own files in these folders and then use them when working with the given form.

 Very important information! When connecting these files, identical function names may be repeated. When such a function is called in the program code, the one that is located **lower in the src connection order** will be executed!

Experienced programmers already understand the idea. For beginners, let me explain: suppose you already have a program written by someone else, but you want to tweak something in it without breaking your code during updates. You write your own function in the **My.js** module, or copy a function from another module that you want to modify, and then connect your module in **src** last!

Summary:

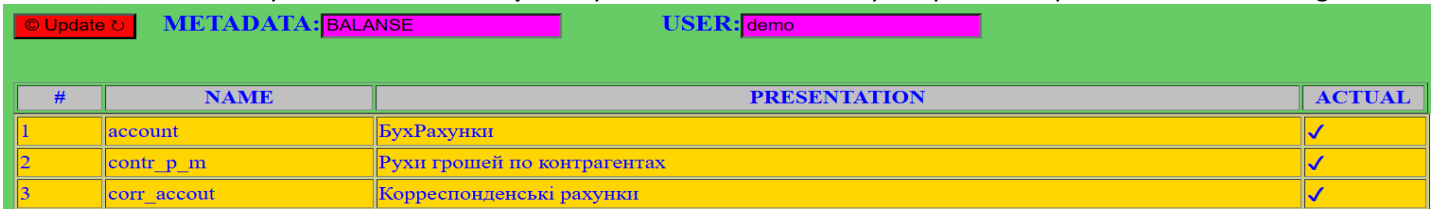
- In the 🗀 **js** folder, I recommend placing your modules that will be used in multiple HTML forms.

- For modules tied to a single metadata, place them in the folder of that metadata, where the **myXX.js** file already exists.

Report in My4s

Turnover Balanse Sheet

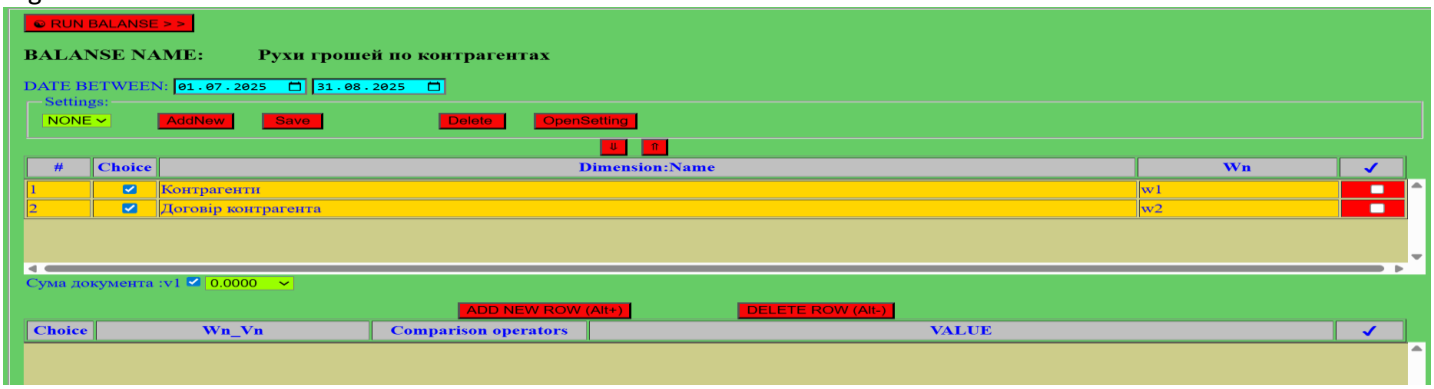
The **Turnover Balance Sheet** is a universal report for all balance registers. It does not require configuration in the **Constructor**, as it is already built into the **Enterprise** system and automatically adapts to the parameters of each register.



Scr52

Register Selection

The list displays all balance registers available in the database. For example, let us open the report for the **contr_p_m** register with a double-click.



Scr53

Main Parameters

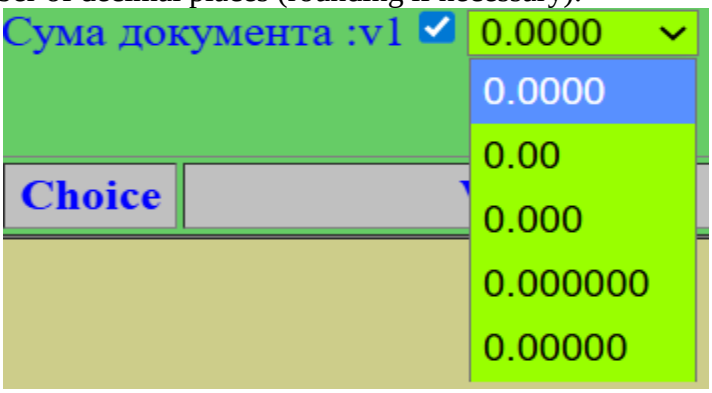
- **DATE BETWEEN** — specifies the start and end dates of the period.
- **Settings** — manages saving and loading of report settings (to be discussed below).

Top Table: Dimensions

- Displays all dimensions configured in the **Constructor**.
- The **Choice checkbox** — includes the dimension in the report.
- The last checkbox — allows changing the order of dimensions, affecting the hierarchy of the report (upper ones are primary, lower ones are subordinate).

Resources and Formatting

- Below is a list of resources with checkboxes.
- You can set the number of decimal places (rounding if necessary).



Scr54

🔍 Bottom Table: Filters

- Contains condition rows. The report will include only those records that meet all rows (**AND logic**).
- **Choice** — enables the row in the filter plan.
- **Wn_Vn** — a dropdown list of all dimensions and resources.

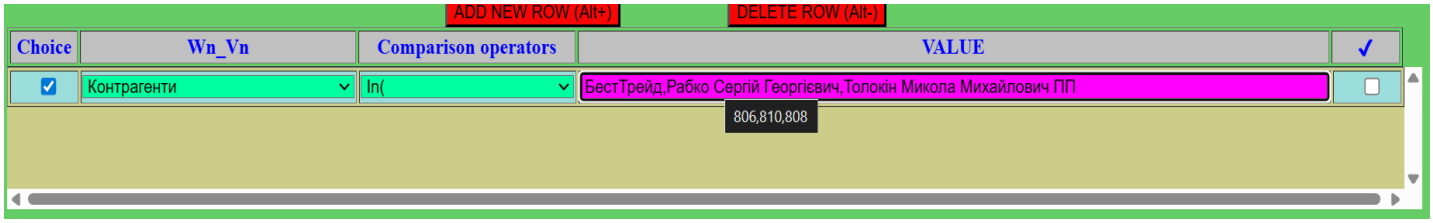
 Subordinate directories (for example, *Contracts*) are not yet supported in filters.

Scr55

Examples of Working with Filters

📌 **Filter by Dimensions** (for example, *Counterparties*):

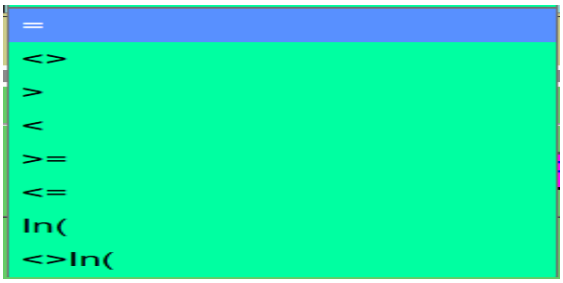
- **Comparison operators:**
  - **=** — one counterparty.
  - **<>** — exclude one.
  - **In(** — select a list of counterparties.
  - **<>In(** — exclude a list.
- **VALUE** — cannot be edited directly. A double-click opens the **FIND** window for selection.



Scr56

To clarify, in the **VALUE** column the names of counterparties will be displayed (separated by commas), but only their unique **IDs**, stored in the **TITLE** of the cell, will be included in the query. You can see them (although it is not necessary to know this) by hovering the cursor over the cell — for example: ***806, 810, 808***. This works the same way with any directory! In this example, we have only one resource, but in other registers there may be more (up to five), and for each one you can create a filter by adding a new row (**ADD NEW ROW (Alt+)**).

Now let us look at how to work with **filters by resources**. Suppose we select the resource *"Document Amount"*, then in the **Comparison operators** column we will get a different list.



Scr57

To remind you, in **My4s** resources are numbers. **VALUE** is a string that can be edited, which in the simplest case means entering a number.

- If you want to cut out a numeric range, enter in one row **">" number**, and then add another filter row with **"<" number**.
- If you use **In(** — write several numbers separated by commas (the decimal separator is always a dot).

If this is unclear, you will understand once you have at least my demo version.

But that is not all: for resource filters there is also the option to filter by **turnover** — simply enter a number, and the filter will apply only to the **end-begin** turnover column.

- If you want to filter by the **beginning of the period**, put the Latin letter **b** before the number — for example: **b250.75**.
- If you want to filter by the **end of the period**, put the Latin letter **e** before the number — for example: **e-28550.968** (a number less than zero!).

 Always use the minus sign if the number is negative.

Here is an example of a report with filters: *(example remains in original form)*

And here is an example of the printed report form with filters:

Рухи грошей по контрагентах

DATE BETWEEN: 2025-03-01 00:00:00 2025-07-31 23:59:59.999
Контрагенти = БестТрейд
Сума документа = -83760

Контрагенти / Договір контрагента	Сума документа		
	begin	end-begin	end
БестТрейд	100000.0000	-83760.0000	16240.0000
Договір №258 від 20.10.2019	100000.0000	-83760.0000	16240.0000
TOTAL	100000.0000	-83760.0000	16240.0000

0 / 83760.0000

Scr58

Take note of the dark row: **0 / 83760.0000** — the point is that in the **end-begin turnover column** not only the turnover is stored, but also hidden values (not printed, but viewable on the computer). Again, by using **TITLE** when hovering the cursor over the desired number, you will see two values separated by a slash.

- The first number is the **incoming (DEBIT)** for this row during the selected period.
- The second number is the **outgoing (CREDIT)**.

The key point is that you can also create filters based on these numbers!

- Simply put the Latin letter **d** before the number if you need the incoming (**DEBIT**, e.g. *d0*).
- Use **c** before the number for the outgoing (**CREDIT**, e.g. *c83760*).

All reports in **My4s** support **turnover breakdowns**. If you double-click on a turnover (**end-begin**), a list of all documents that contributed to this figure during the selected period will open.

METADATA: DOCS **USER:**demo **DOCUMENT:** AllDOCS

#	DOCUMENT	ACT.	INFORMATION
1	Прибуткова накладна #000000000006 > 2025-07-03 20:18:57.000	✓	БестТрейд|Головний склад|83760.0000|
2	Прибуткова накладна #000000000004 > 2025-03-02 15:03:04.241	✓	БестТрейд|Головний склад|60225.6000|

Scr59

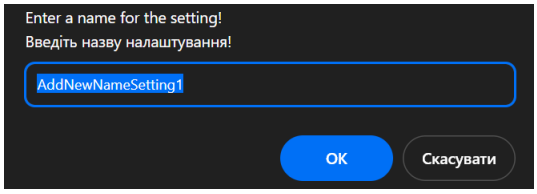
After that, there is the option to **double-click on a document** to open it, and you can even edit it, save it, and then re-run the report.
Now let us describe how **Settings** works.

Settings:

NONE AddNew Save Delete OpenSetting

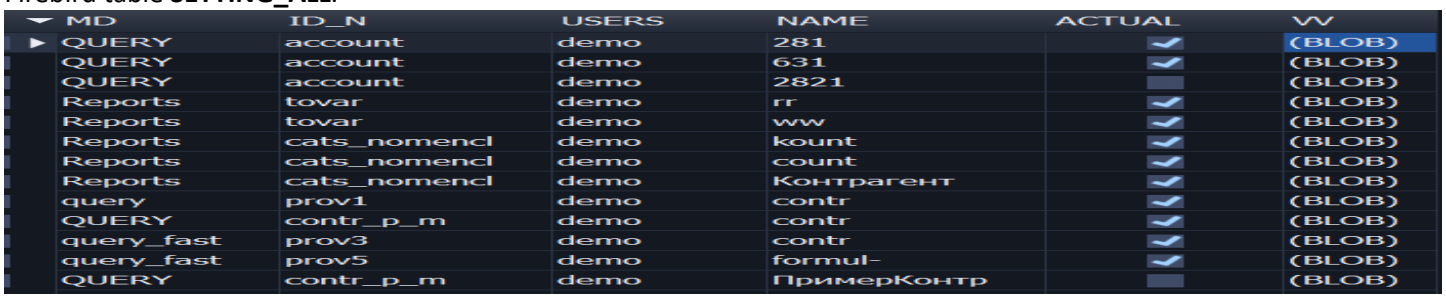
Scr60

When you create a filter and want to save it for future use, this is where **Settings** comes in handy. Click **AddNew** — a window will open.



Scr61

Enter the name of your setting. It will appear in the dropdown list. Next, click **Save**. The data will be written into the Firebird table **SETTING_ALL**.



MD	ID_N	USERS	NAME	ACTUAL	VV
QUERY	account	demo	281	✓	(BLOB)
QUERY	account	demo	631	✓	(BLOB)
QUERY	account	demo	2821		(BLOB)
Reports	tovar	demo	rr	✓	(BLOB)
Reports	tovar	demo	ww	✓	(BLOB)
Reports	cats_nomencl	demo	kount	✓	(BLOB)
Reports	cats_nomencl	demo	count	✓	(BLOB)
Reports	cats_nomencl	demo	Контрагент	✓	(BLOB)
query	prov1	demo	contr	✓	(BLOB)
QUERY	contr_p_m	demo	contr	✓	(BLOB)
query_fast	prov3	demo	contr	✓	(BLOB)
query_fast	prov5	demo	formul-	✓	(BLOB)
QUERY	contr_p_m	demo	ПримерКонтр		(BLOB)

Scr62

When you open the **Turnover Balance Sheet** report again, you can select your saved setting from the dropdown list by name. All configurations you have saved will be displayed. If you need to make changes, simply click **Save** again, and the updated setting will be stored.

How Settings Are Saved

It is important to understand: when you click **Save**, the settings are stored not only in the **Firebird** database but also locally in the browser (via **localStorage**). The report primarily works with data from the browser. You may have already noticed that I try to minimize database queries — all to ensure the system can comfortably serve not just dozens but thousands of clients simultaneously (and ideally even more!).

□ Browser Cleanup and Restoring Settings

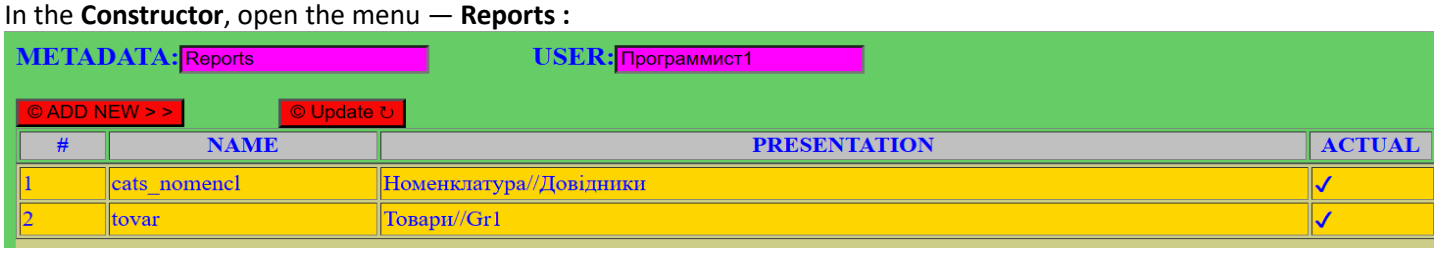
Sometimes the browser needs to be cleaned, and then your local **Settings** disappear. But we save them in the database for a reason! To restore them, there is the **OpenSetting** button — after clicking it, all your settings will be loaded from the database and written back into the browser cache. The **OpenSetting** button will be hidden if the settings are already present in the browser.

Universality of Settings

Settings are available in all reports created in **My4s** and work almost identically. Therefore, I will not describe them further.

Report Created in the Constructor

In the **Constructor**, open the menu — **Reports :**



Scr63

Here you can see the reports created through the **Reports** menu. To create a new one, click **ADDNEW**.
All reports are generated based on the combination of several tables using **UNION**. The use of **JOIN** is also possible, but in that case the SQL query (**SELECT...**) must be manually refined.

At the core lies the **DOCS.htm** document creation form, adapted for reports. At present, up to **seven table parts** are supported. The number of table parts corresponds to the number of **UNION** operations in the query plus one.
Ось переклад цього блоку англійською у «чистому» форматі з виділенням ключових термінів:

Creating a New Report

When creating a new report, you need to specify:

- **NAME** — a unique name for the report (note: after saving, changing it will be difficult).
- **PRESENTATION** — the display name.

Next, in the **TITLE** field, select the required **Firebird tables** from the **TABLES** list — as many times as needed to form the **UNION** query. After each selection, a line of the following type is automatically added to the **textarea**:

```
SELECT * FROM DOCS2 WHERE ID_N=`idd__` AND DAT BETWEEN dat8_ AND dat9_ AND 1=1;;
```

After choosing a table, an additional dropdown list with possible **ID_N** values will appear — this is needed to specify which document to work with (relevant for the **DOCS** and **DOCS2** tables).

In words, this may sound complicated, but when running the demo version everything becomes clear and visual. The system works automatically — the main thing is to correctly select the parameters.

Let us consider creating a report using the example of the already prepared report **“tovar”**.

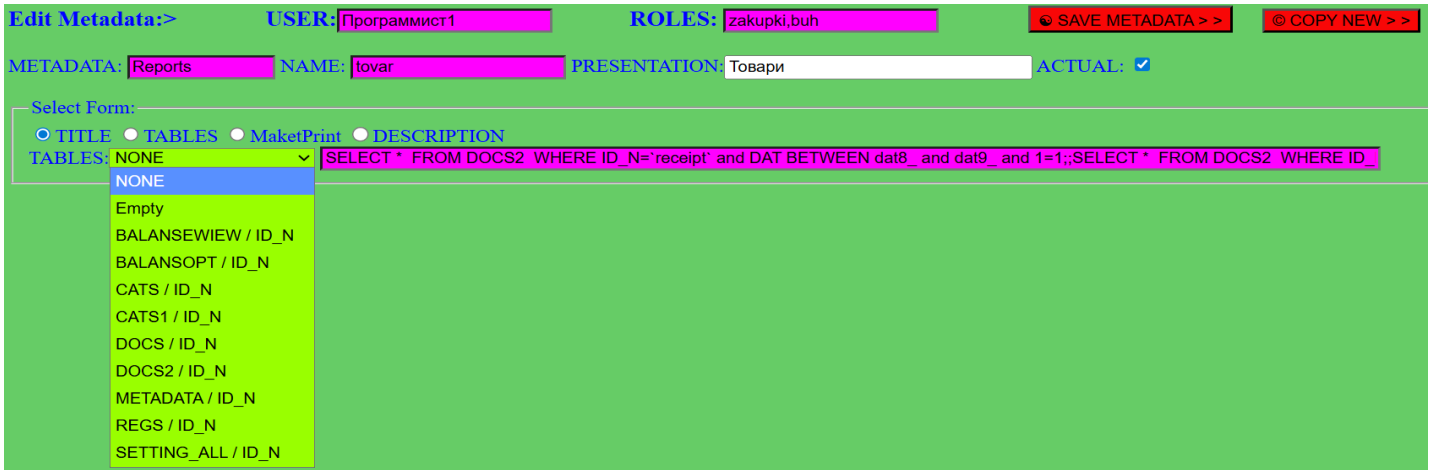
If you are not familiar with the principle of **UNION** in SQL, I recommend first reviewing the documentation — preferably for **Firebird**, although in other DBMS the mechanism works in a similar way.

The report is generated only for the following documents:

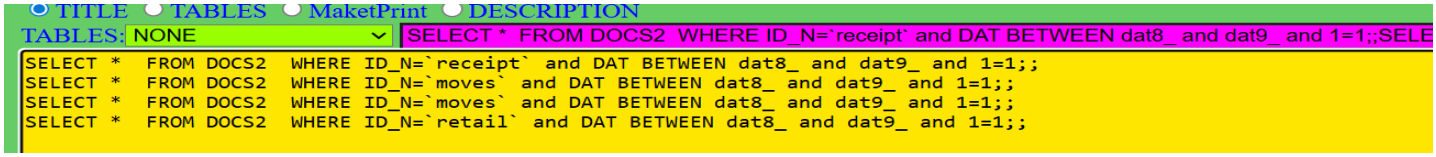
- **Incoming Invoice**
- **Transfer**
- **Retail Trade**

Special note: the **Transfer** document is used twice, since it contains two warehouses — the source and the recipient.

With each document selection, initial **SELECT queries** are added to the **textarea**, separated by the symbols **“;;”**. At the same time, table templates are automatically created — still empty, but ready to be filled.



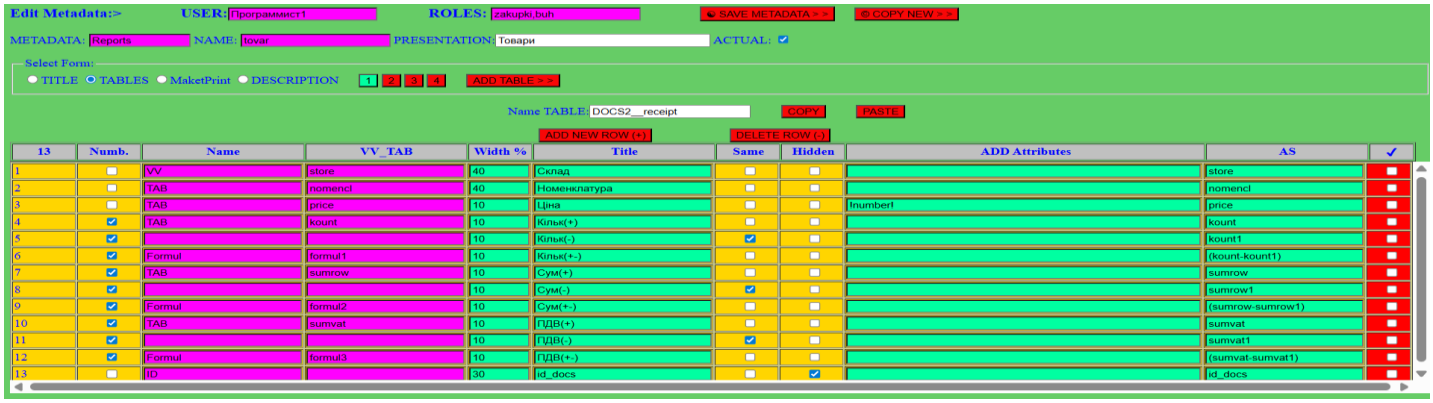
Scr64



Scr64a

As I mentioned earlier, **DOCS2** is a **VIEW** that combines two Firebird tables into one (**DOCS** and **ROWS_DOCS**).

Now, when you switch to the **TABLES** tab, you will see table **No. 1 out of 4** already filled. But this is actually a ready-made query — fully prepared for saving. In reality, the table will initially be empty.



Scr65

General Principles of Report Configuration

- **DOCS2__receipt** — this is the table name automatically generated by the program. You can change it, but it will not affect the operation of the report.
- Before filling the tables, it is important to plan the structure of the query in advance.

• **Table No. 1** is considered the main one — all others are adjusted to it. • The number of rows in all tables is always the same. • **Dimensions** go first, and the **Width** parameter is taken from the first row. Other dimensions (not resources) do not affect Width. • In the **Numb** column, you must check the box for all resources.

Working with the AS Column

The **AS** column is used similarly to the SQL query **SELECT ... AS ...**. It is filled automatically, but you can change it if necessary — in some cases this is even required.

Adding a New Row When adding a new row, always start with the **Name** field. In the dropdown list, the first two parameters are standard:

- **Empty** — means that the parameter is absent in the current table but exists in at least one of the others. All other fields (except **Name** and **VV_TAB**) must still be filled in.
- **Formul** — indicates that a formula will be used in the column. In the window that appears, you must specify the formula name. All formulas are defined in **Table No. 1**.

For formulas, the **AS** column has special significance — it is the formula itself. Inside the parentheses, use identifiers from other **AS** rows.

Parameters Name, VV, and TAB

- All other parameters in **Name** are taken from the structure of the corresponding Firebird table.
- Special attention should be paid to the parameters **VV** and **TAB** (at least one is always present). These are **BLOB fields** containing information in the **4SON format** (described earlier).
- When such a parameter is selected, the **VV_TAB** column becomes active, allowing you to choose specific parameters from the BLOB field.

Language and Formatting

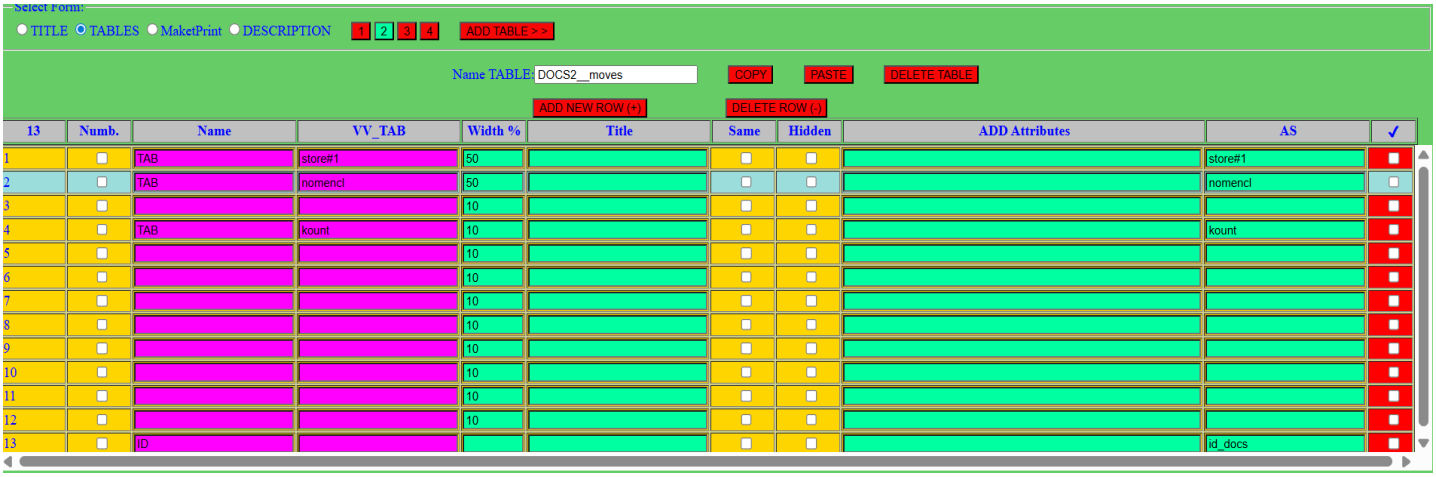
- All fields must be filled in using Latin characters, except for the **Title** column — here you can use arbitrary text.
- The **ADD Attributes** column is not used, as described earlier. It is simply a string intended for additional settings when executing the report.

Example of Use

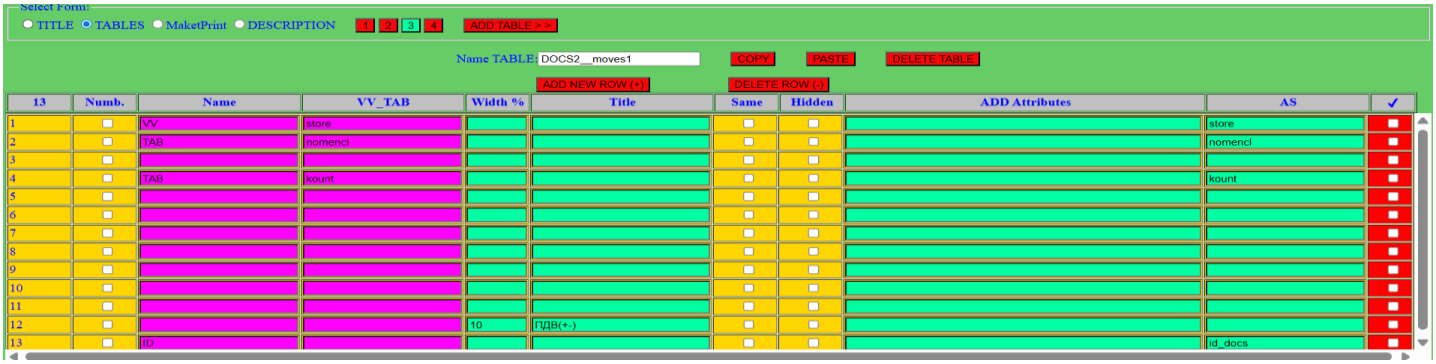
- If a dimension contains not a reference to a directory but a regular number (for example, *price*), you must tell the program that this is a number.
- To do this, in the **ADD Attributes** column write: **"!number!"**.
- Without this string, the report will work, but filtering by price will not be possible.

Other Notes

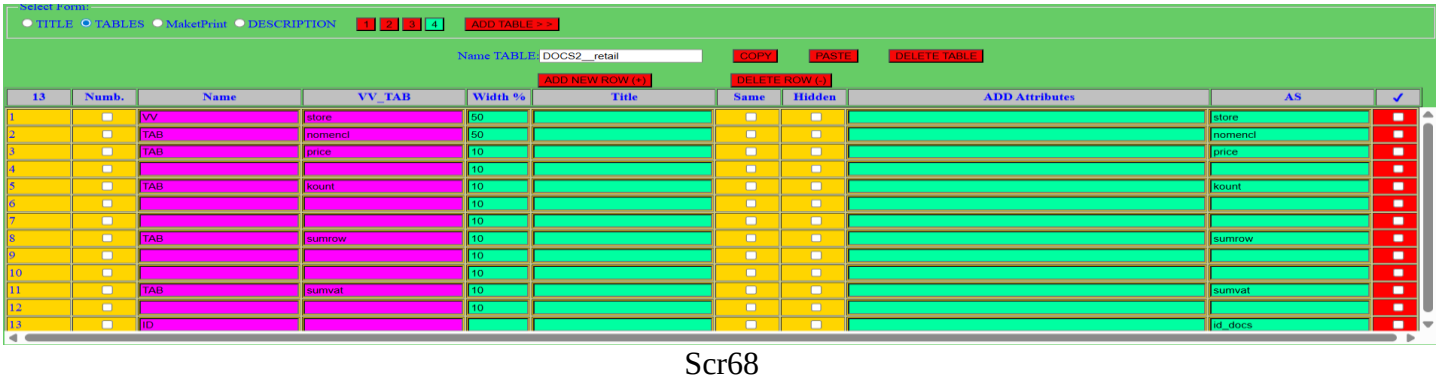
- A checkmark in the **Same** column means that the parameter will be placed in the same cell as the previous one (above) — i.e., in two levels (layers).
- For other tables, the main task is to configure and link rows with **Table No. 1** in the **Name** and **VV_TAB** columns. If there is no link, select **Empty**.
- The remaining columns do not play a role, regardless of what is written in them.



Scr66



Scr67



Scr68

This is how the report will look in the Enterprise:

Товари

DATE BETWEEN: 2025-07-01 / 2025-08-31

Склад / Номенклатура / Ціна	Кільк(+) Кільк(-)	Кільк(+-)	Сум(+) Сум(-)	Сум(+-)	ПДВ(+) ПДВ(-)	ПДВ(+-)
Головний склад	772.00 0.00	772.00	83760.00 0.00	83760.00	13960.00 0.00	13960.00
Горячий Шоколад	150.00 0.00	150.00	18000.00 0.00	18000.00	3000.00 0.00	3000.00
100	150.00 0.00	150.00	18000.00 0.00	18000.00	3000.00 0.00	3000.00
Замінник молока	500.00 0.00	500.00	48000.00 0.00	48000.00	8000.00 0.00	8000.00
80	500.00 0.00	500.00	48000.00 0.00	48000.00	8000.00 0.00	8000.00
Коньяк "Наполеон"	2.00 0.00	2.00	480.00 0.00	480.00	80.00 0.00	80.00
200	2.00 0.00	2.00	480.00 0.00	480.00	80.00 0.00	80.00
КофеTotti Supremo	120.00 0.00	120.00	17280.00 0.00	17280.00	2880.00 0.00	2880.00
120	120.00 0.00	120.00	17280.00 0.00	17280.00	2880.00 0.00	2880.00
TOTAL:	772.00 0.00	772.00	83760.00 0.00	83760.00	13960.00 0.00	13960.00

Scr69

As you may have noticed, in all my printed forms the **dimensions** are always displayed in a single column — the first one.

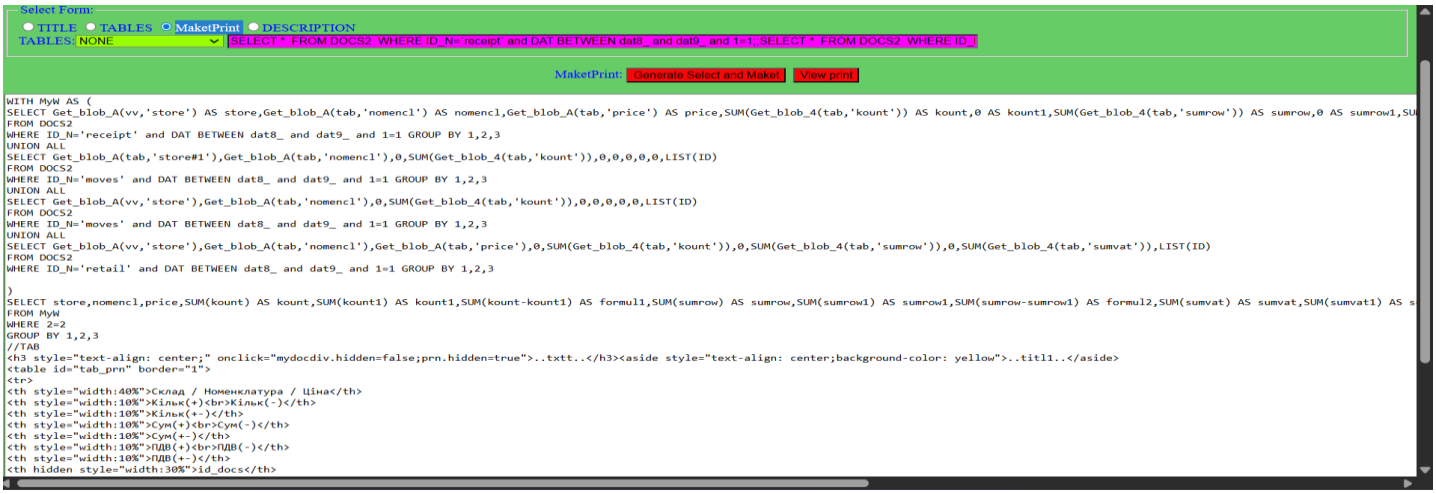
The reason is that the **SELECT... query** returns data as a single row, collapsed into the structure *"Warehouse / Item / Price"*. In the printed form, they are shown as separate rows with visual indentation (a long space).

Initially, I implemented the query so that it immediately returned the data in a ready-made form — exactly as you see on the screen. The query itself became significantly larger, but that was not critical. The problem arose elsewhere: the **Firebird server** began spending considerably more time executing such a query.

Therefore, I abandoned this idea and moved the logic of splitting into subordinate dimensions and calculating sums for each dimension to the **client side**. Yes, it required a lot of effort, but I wrote the code once, and now I apply it in all similar queries.

The report also supports **document breakdowns** (if it is a document-based report). For this, a hidden column **id_docs** is added. However, the breakdown works only for the **last dimension** — in this example, *Price* (via double-click).

Let us now look at the next tab — MaketPrint:

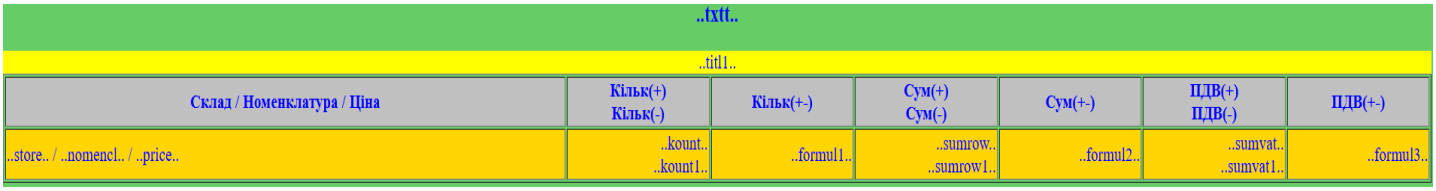


Scr70

Initially, the **textarea** field remains empty. Only after fully completing the setup of all tables can you click the **Generate Select and Maket** button — the program will automatically generate the SQL query and HTML code. If the tables are filled correctly, the report should successfully run in the **Enterprise** system.

If the report works, the text can be edited — provided you clearly understand what you are doing. Of course, experimentation is allowed: refine the text, save it, and run it again in **Enterprise**. If the result turns out unsuccessful — no problem. Simply delete all the text and click **Generate Select and Maket** again. Note: if even a single character remains in the **textarea**, the button will not work.

When the text is ready, you can view the report structure by clicking the **View print** button.



Scr71

As I mentioned earlier — in **Enterprise**, by clicking on the header of the printed form (in our case, *"Goods"*), you can return to the report settings. You can change the dates or filters and generate the report again.

You can also return to the report settings by pressing **Esc**, but there is a small difference — in addition to the report settings, the printed form will remain visible just below the settings! This works in almost all reports.

The next tab is **DESCRIPTION**. This is simply a **textarea field**. It does not affect the operation of the report. You can write something like *Help* there, or anything else you consider useful!

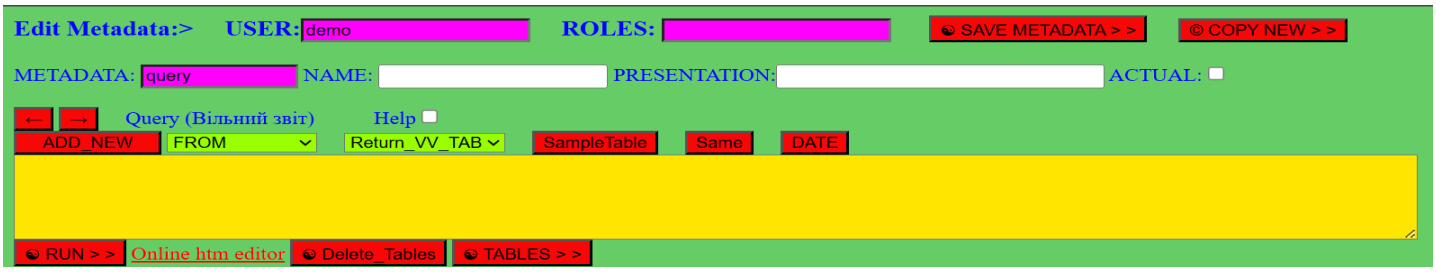
Next Report — Assistant for Creating Additional Reports

In the **Constructor**, open the menu — **Assistant for Creating Additional Reports**.

This constructor allows you to create almost the same reports as the previous one, but the time spent on building the report, I hope, will be significantly less. The reason is that the previous report requires a solid understanding of how **UNION** works in queries (**SELECT**).

In this report, however, you do not need to strictly understand or follow the rules of building a report using **UNION**. You simply need to make data selections from different tables (or even from just one!), and the program will automatically combine the table data according to the rules of **UNION**.

Let us now proceed to the description of creating a new report.



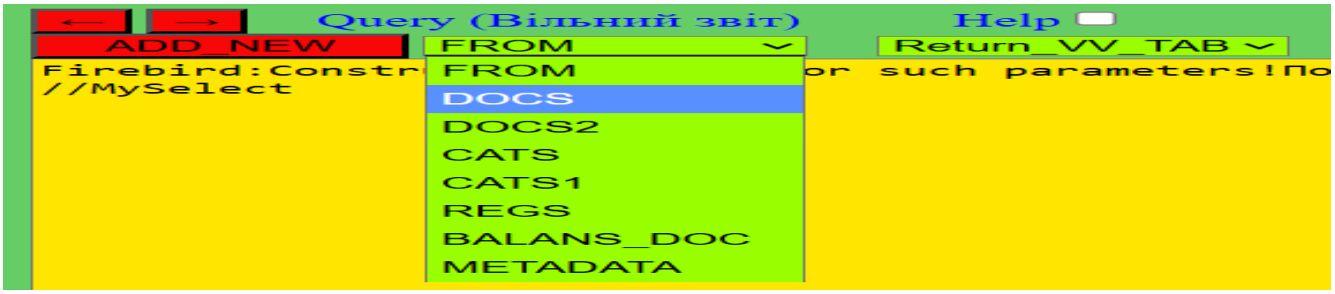
Scr72

Starting Work with a New Report

- **Document Header**: its completion was described earlier, so it is not considered here.
- **Help**: if you check this option, a simple **textarea field** will open where you can add comments or a description of the report at your discretion.
- **Left/Right Arrows**: mainly used to test the query at the end of report creation, with the **RUN>>** button.

Important Rules When Creating a Report

1. At the beginning, the **textarea field** must always be empty. It cannot be edited until all required data has been filled in.
2. Work begins by pressing the **ADD_NEW** button. After this, the field will display service text: *Firebird: Construct a request for such parameters! Build a query for such parameters: //MySelect* Next, you must proceed to selecting the tables and data that will be used in the report.



Scr73

Selecting a Table for the Report

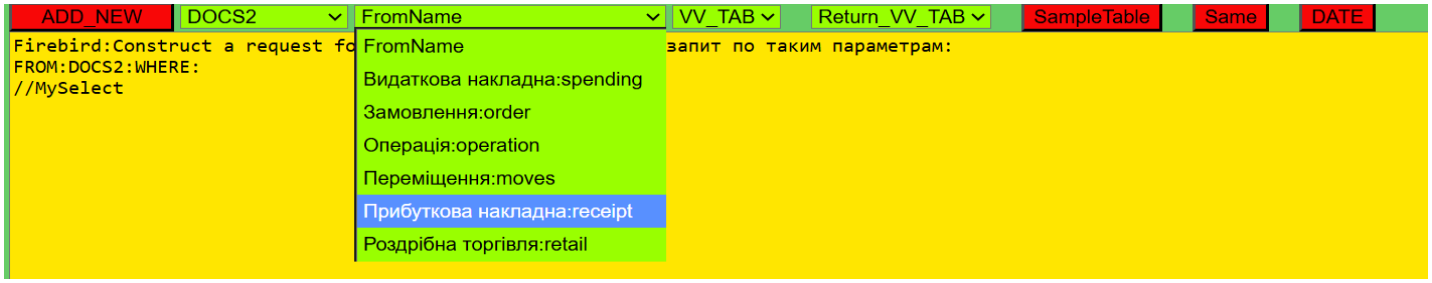
In the **FROM** dropdown list, you need to choose the desired table from **Firebird**.

All tables were described earlier, but it is worth explaining separately the **BALANS_DOC** view.

- **BALANS_DOC** is a **VIEW** that provides access to all document movements.
- You can study its structure in more detail using the **Firebird editor**.

For example, let us select the **DOCS2** view. After this, additional dropdown lists will appear:

- **FromName**
- **VV_TAB**



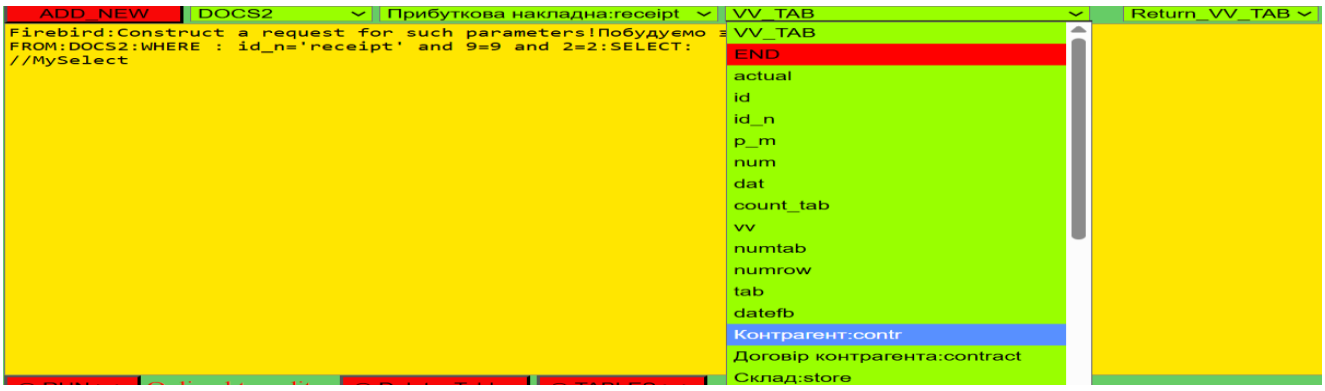
Scr74

Working with Dropdown Lists

- In the **FromName** list, all documents available in the database are displayed.
- For example, select the document **receipt** (*Incoming Invoice*).
- After selecting the document, the **VV_TAB** dropdown list is automatically filled.
- It will contain all parameters of the selected table, including those located in the document's **Blob fields**.

Thus, the sequence of actions is as follows:

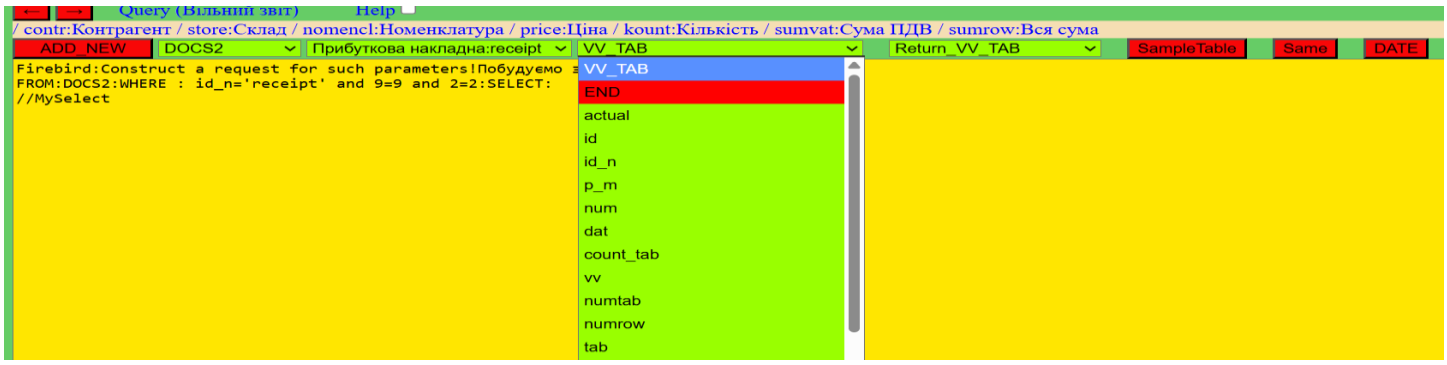
1. Select a document in the **FromName** list.
2. Wait for the automatic filling of the **VV_TAB** list.
3. Use the parameters from **VV_TAB** for further report configuration.



Scr75

Here is where the real programmer's work begins. You should already know which parameters you will need from this document. Then, select these parameters one by one from the dropdown list.

After each selection, the chosen parameters will be removed from the dropdown list and will appear in a special control field.



Scr76

Completing Parameter Selection At this stage, it is important to be careful:

- If a mistake is made, you will have to start over.
- To do this, you can clear the **textarea field** (Ctrl+A → Delete) or simply reload the page.

If everything is done correctly:

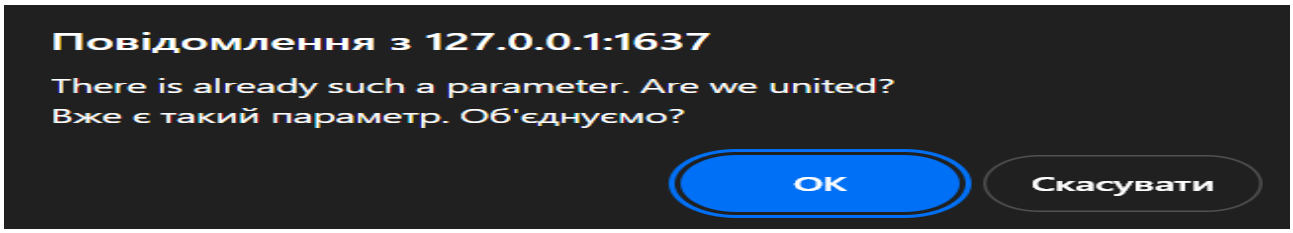
- At the end of all selections, you must choose the **END** item from the list (highlighted in red).
- This signals to the program that the selection stage is complete.
- After this, a new line will appear in the **textarea** above the `//MySelect` string, containing a preliminary set of data you selected earlier:

```
FROM:DOCS2:WHERE : id_n='receipt' and dat BETWEEN dat8_ and dat9_ and 2=2:SELECT
:Get_blob_A(vv,'contr') AS contr,Get_blob_A(vv,'store') AS
store,SUM(Get_blob_4(tab,'kount')) AS kount,
```

Do not be alarmed — this line will ultimately allow the program to correctly form the final **SELECT**.

If your query uses multiple tables, then again go to the **FROM** list, select a new table, and repeat the process just as with the first table. But now there is an important nuance: we are preparing a query using **UNION**.

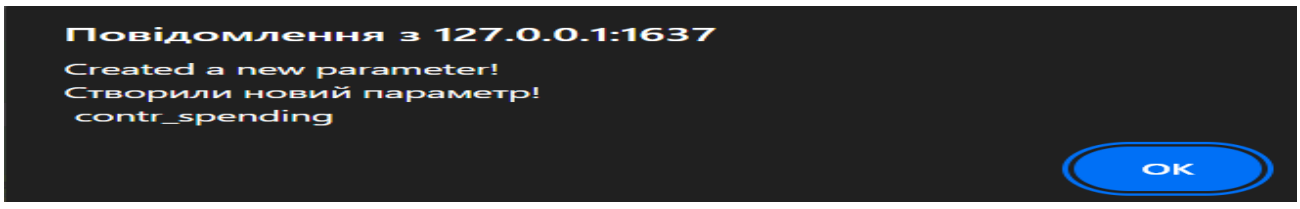
With each parameter selection, the program analyzes the parameter name for matches with names from previous tables. If a match is found, a window will appear.



Scr77

If you click **OK**, the program will remember that these parameters must be combined when creating the final **SELECT** using **UNION**.

If you click **Cancel**, a new parameter will be created and a window will appear.



Finalizing Parameter Selection

- A new parameter receives a name created from the new name and the table name, joined with an underscore. This ensures the program correctly builds **SELECT** statements with **UNION**.
- Important: if documents are involved in the report and a breakdown in the printed form is required, you must specify the **document id** as the last parameter when selecting document fields.

Return_VV_TAB List After selecting from the tables, proceed to the final step — open the **Return_VV_TAB** list. In addition to the standard items (**END**, **formul_**, **ALL**), it will display all selected parameters in the order they were added.

- If the parameters are already arranged correctly (dimensions first, then resources) and formulas are not needed, you can select **ALL** — then all parameters will automatically be included in the **Return** line.
- If the order is different or formulas are required, parameters must be selected manually, one by one, in the order they should appear in the printed form (dimensions first, then resources).
- To insert a formula into a column, select the **formul_** parameter. Note: if a parameter is used only inside a formula and is not needed in the printed form, it should not be selected.

Example After completing parameter selection, choose **END**. A new line will appear in the **textarea** above the `//MySelect` string with a preliminary set of data:

```
Return: Select : contr,store,nomencl,price,SUM(kount) AS kount,SUM(sumvat) AS
sumvat,SUM(sumrow) AS sumrow,SUM(kount_spending) AS kount_spending,SUM(sumvat_spending)
AS sumvat_spending,SUM(sumrow_spending) AS sumrow_spending,SUM(formul_1) AS formul_1
```

After `//MySelect`, the program will generate the full **SELECT**, for example:

```
WITH q_all AS (
  SELECT Get_blob_A(vv,'contr') AS contr,Get_blob_A(vv,'store') AS
store,Get_blob_A(tab,'nomencl') AS nomencl,Get_blob_T(tab,'price') AS price,
         SUM(Get_blob_4(tab,'kount')) AS kount,SUM(Get_blob_4(tab,'sumvat')) AS
sumvat,SUM(Get_blob_4(tab,'sumrow')) AS sumrow,
         NULL AS price_spending,NULL AS kount_spending,NULL AS sumvat_spending,NULL AS
sumrow_spending
  FROM DOCS2 WHERE id_n='receipt' AND dat BETWEEN dat8_ AND dat9_ AND 2=2
  GROUP BY contr,store,nomencl,price,price_spending
  UNION ALL
  SELECT Get_blob_A(vv,'contr') AS contr,Get_blob_A(vv,'store') AS
store,Get_blob_A(tab,'nomencl') AS nomencl,
         NULL AS price,NULL AS kount,NULL AS sumvat,NULL AS sumrow,
         Get_blob_T(tab,'price') AS price_spending,
         SUM(Get_blob_4(tab,'kount')) AS kount_spending,SUM(Get_blob_4(tab,'sumvat')) AS
sumvat_spending,SUM(Get_blob_4(tab,'sumrow')) AS sumrow_spending
  FROM DOCS2 WHERE id_n='spending' AND dat BETWEEN dat8_ AND dat9_ AND 2=2
  GROUP BY contr,store,nomencl,price,price_spending
)
SELECT contr,store,nomencl,price,price_spending,
       SUM(kount) AS kount,SUM(sumvat) AS sumvat,SUM(sumrow) AS sumrow,
       SUM(kount_spending) AS kount_spending,SUM(sumvat_spending) AS
sumvat_spending,SUM(sumrow_spending) AS sumrow_spending,
       SUM(formul_1) AS formul_1
FROM q_all
GROUP BY contr,store,nomencl,price,price_spending;
```

Editing Formulas At the formula refinement stage, edit the line `SUM(formul_1) AS formul_1` to reflect the required calculation. For example:

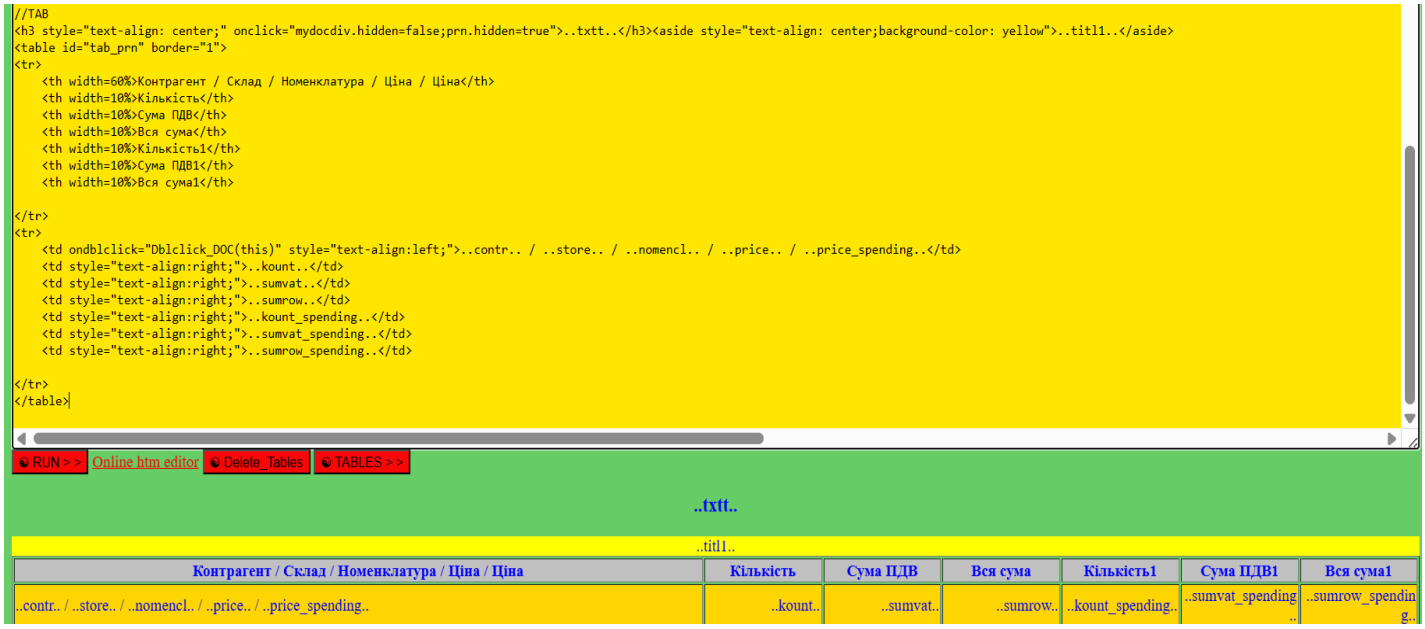
```
SUM(kount - kount_spending) AS formul_1
```

- The number of formulas can be unlimited.
- The program automatically creates them using a counter: **formul_1**, **formul_2**, **formul_3**, etc.

Once all parameters are correctly filled in, the **SELECT** block can be finalized. The query will then execute properly and return the expected result.

Generating HTML Code Next, generate the HTML code for the printed form output:

- Click the **SampleTable** button, and the program will automatically create the HTML code.
- To view the structure of the printed form, click the **TABLES** button — the structure of the printed form will appear at the bottom of the screen.



Scr79

The **table header** will look the same as in **Enterprise**. If you need to change something, open the line with the tag and edit it (for example, you can adjust the column width using `width=?`). After that, click the **TABLES** button again, and the changes will appear in the structure.

The **Delete_Tables** button hides the table structure.

You can also make changes in the lines with tags, but for this you need a good understanding of how **HTML** works. The main rule: do not change parameter names between two dots (for example, `..kount..`), otherwise the report will stop working.

Any column can have an event attached. However, you must write the event handler yourself in a connected module, using:

Html:
```
<script src="Myjsjs/METADATA__/id_n__/my.js"></script>
```

If you add a new event but the program cannot find the corresponding function, it will result in an error. In this way, you can connect your own event handling for the required column.

I already have an example: on the first dimension column there is an event:

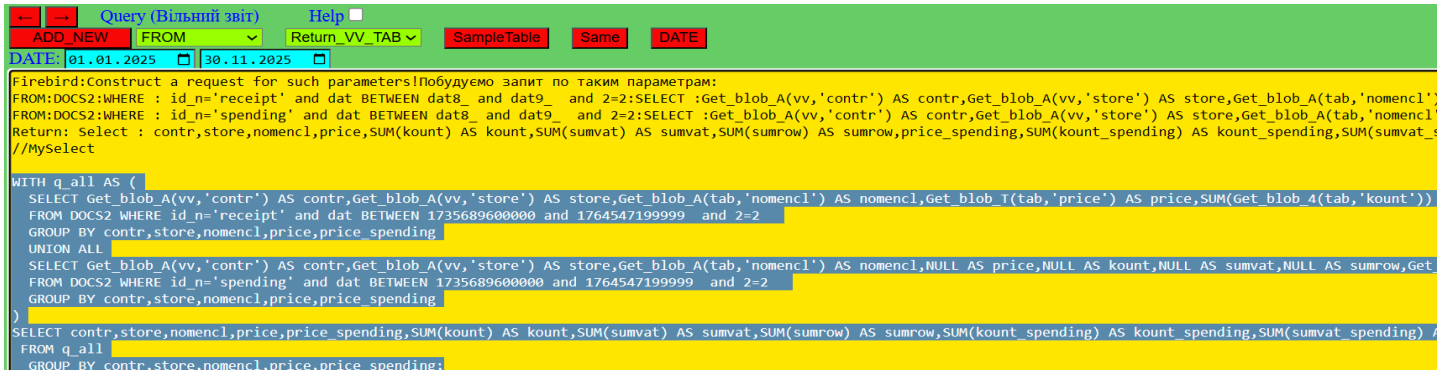
Html:
```
ondblclick="Dblclick_DOC(this)"
```

By default, it opens the breakdown of documents linked to the row. If you do not need this breakdown, the event can be deleted or replaced with your own.

Now about the **RUN>>** button. It allows you to test the operation of the **SELECT query** directly in the report constructor.

- If the query uses date filtering (`dat BETWEEN dat8_ and dat9_`), you must first fill in these values, otherwise an error will occur.

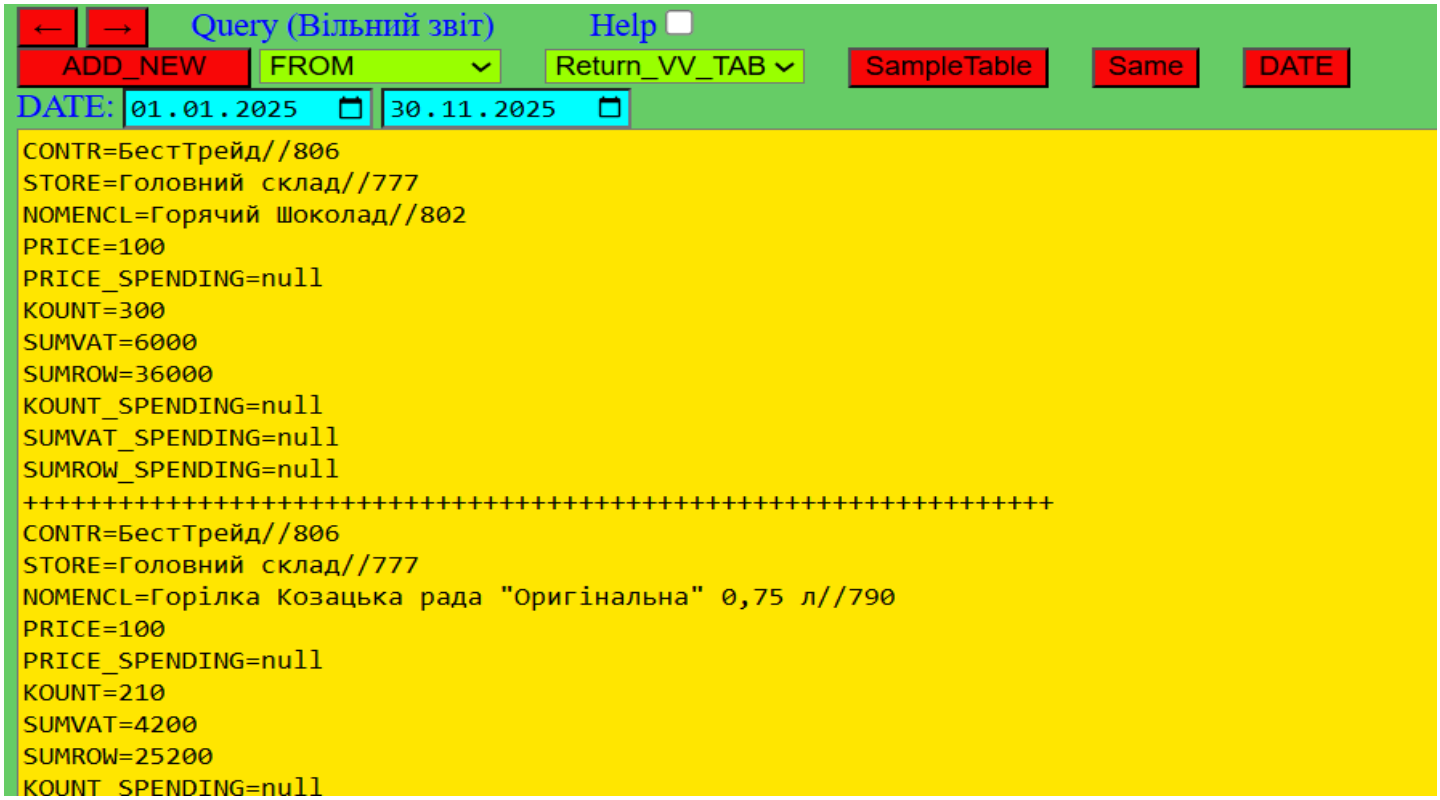
- To do this, click the **DATE** button and in the window that appears specify the start and end dates.
- After that, highlight the plain text of the query.



Scr80

Only then can you click the **RUN>>** button.

- If **"undefined"** appears, it means something is wrong with the query.
- Otherwise, the **textarea field** will display the query result, approximately in the following form:



Scr81

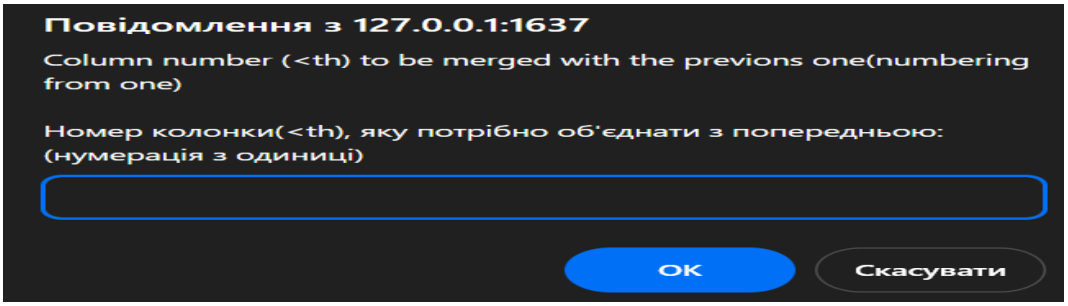
Do not worry: when outputting the printed form in **Enterprise**, the program will process all data correctly. This means only one thing — your query works.

If you press the **left arrow** button several times, you can return the **textarea field** to its initial state, until the date values reappear in the query (`dat BETWEEN dat8_ and dat9_`).

 Checking with the **RUN>>** button is not mandatory. Experienced **My4s** users will recognize when the query is correct. If it turns out to be incorrect, in **Enterprise** it simply will not work — then you return to the Constructor and fix it.

Now about the **Same** button (in the same column). When a report has many columns, the browser tries to fit them all on the screen, and the result is not always convenient.

After pressing the **Same** button, a window will open where you need to specify the tag number that you want to merge with the previous column.



Scr82

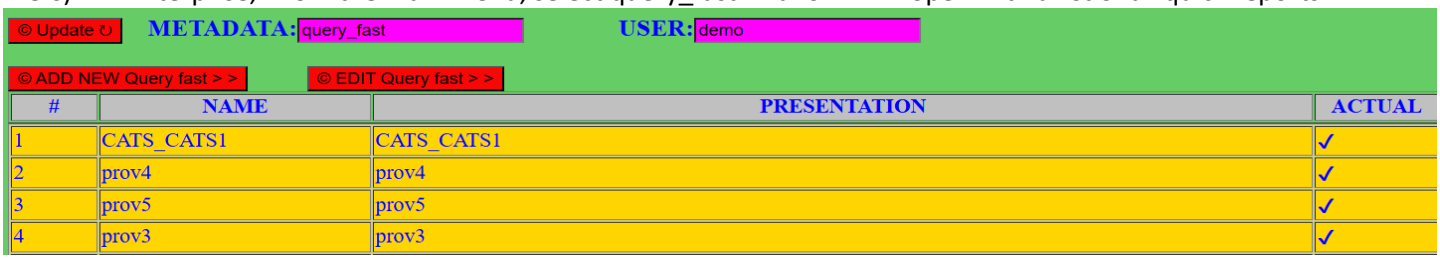
Automation will merge not only in `<th>` but also in `<td>`. You can press **Same** as many times as you want to merge columns — even across multiple levels! The merged column will be highlighted.

It is important that in the **Return** setup the columns are arranged in the correct order so they can be merged properly.

At the end, you can save everything you have done using the **SAVE METADATA** button. Then you can go to **Enterprise**, select **Additional reports** in the main menu, find your report, and test it. If something is wrong, return to the Constructor, edit the report, save it again, and check it in **Enterprise**.

Query_fast Reports (Quick Reports)

These reports largely coincide with the previous report “Assistant for Writing Additional Reports.” To create and edit these reports, the Constructor is not required — they are generated directly within the “Enterprise” system. This feature allows each user (for example, an accountant) to independently create their own reports. Naturally, programmers must first train them to do so. Therefore, a repeated description is not necessary. The only difference is that two items are missing from the table list: BALANS_DOC and METADATA (I believe work with these tables should be left to programmers). In “Enterprise,” from the main menu, select query_fast — a form will open with a list of all quick reports.

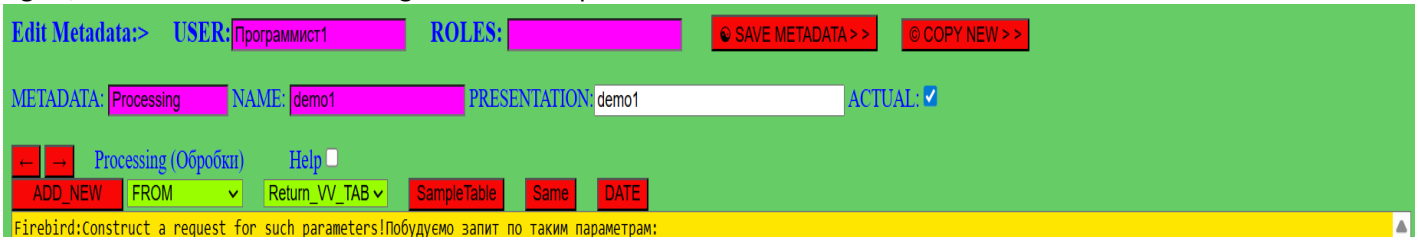


Scr83

If the list contains a large number of rows, you can use the standard Ctrl+F combination to find the desired one. To run a report, double-click the corresponding row. Ready-made reports have no access restrictions. To create a new report, click the ADD NEW … button — a form will open, similar to the previous report. A detailed description can be found there as well. To edit an existing report, select its row and click the EDIT … button. All users can use these reports. However, only the following can edit them: • users (user) who created the report; • users with full rights (all). Important: if a report is edited by a user with full rights (all), the primary owner still remains the user who originally created it.

Processing

Again, I used “Assistant for Writing Additional Reports” as the basis.



Scr84

Almost always, a report is required for processing. This is convenient because it allows you to immediately verify the correctness of its execution. If, in the final version, displaying the print form on the screen turns out to be unnecessary, simply delete all the contents from the textarea starting with “//TAB” and below. Another

scenario is also possible: a report may not be needed at all when preparing the processing. In this case, you describe all processing procedures independently in the module my.js (or another module connected via src). Then, information in the textarea is not required — it can be completely deleted or not filled in at all. In the connected module <**script src="Myjsjs/DOCS/id_n__/my.js"></script**> there must be a function **MyProcessing**(zr), and it will depend on you how the processing will work.

Let's summarize reports and processing.

In the "Enterprise" system, the operation of reports and processing is managed by a single HTML template **MyReport.htm** and connected modules (in addition to the standard ones):

Html:
```
<script src="Report_db.js"></script>
<script src="Myjsjs/METADATA__/id_n__/myXX.js"></script>
<script src="Mydbdb/query_fast.js"></script>
<script src="Myjsjs/METADATA__/id_n__/my.js"></script>
```

I will not provide a detailed description of their operation — the code is fully open. Experienced programmers can easily understand the logic and adapt it to their tasks if necessary.

I will only highlight a few points that help avoid changes in the **Report_db.js** module. In most cases, adjustments should be made in **my.js**, which is specifically intended for this purpose.

In **Report_db.js**, several "reference points" are provided that direct code execution to the **my.js** module. These are convenient for implementing custom modifications without breaking the base structure:

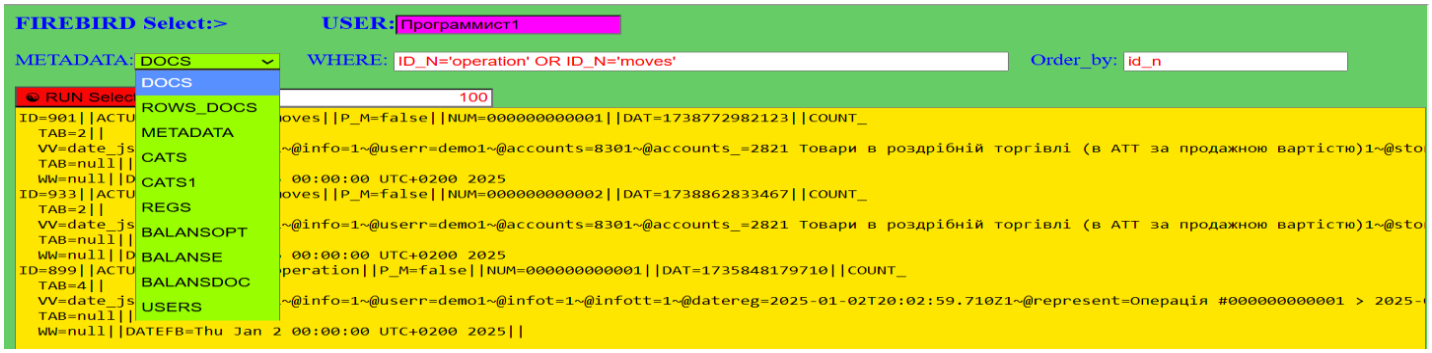
Javascript:
```
if (typeof MyBeginRUN === "function") MyBeginRUN(strselect_); // My.js
```

This code checks for the existence of the **MyBeginRUN** function in the context of report or processing execution. Without such a check (if the function is not found), the program will terminate with a critical error. The **MyBeginRUN** function can be used to dynamically modify the **strselect_** string before it is executed in Firebird. If no changes are needed, the function simply does not exist.

Next reference function:
Javascript:
```
if (typeof MyENDRUN === "function") MyENDRUN(zr); // My.js
```
The **MyENDRUN(zr)** function allows you to process the result already obtained from Firebird (if you suddenly need it).
For processing:
Javascript:
```
if (typeof MyProcessing === "function") MyProcessing(zr); // My.js
```
Next:
javascript
```
if (typeof MyENDRUNrun === "function") MyENDRUNrun(tab_prn); // My.js
```
Here you can attempt to modify the appearance of the print form before it is displayed on the screen!

Firebird Select Report

In the constructor, there is another metadata report called "Firebird select." I wrote this report almost at the very beginning of creating My4s, when I did not yet have Firebird Editor Pro. It allowed me to quickly view the contents of Firebird tables. I have kept this report, as it might still be of interest to someone.



Scr85

I will not describe the operation of the report. Those familiar with Firebird will easily understand it.

Let's sum up briefly what was written above.

I realize that what is written may not be immediately clear in some places. However, this description is quite sufficient for you to decide whether you want to continue familiarizing yourself with My4s. If you do want to continue, then you should first install the demo version and go through the description using specific examples. That way, many things will become clearer.

Installation guide for My4s on your computer or laptop with Windows

You should already have an archive with the distribution Distrib_My4s.zip transferred from GitHab on your computer!
WARNING! The demo database and the empty database (my4sNull) work for me with WIN1251 encoding.
I created My4s primarily for my country — Ukraine.
This does not mean that the system will not work with other coding, but it will take some effort to configure it (artificial intelligence will help you!).

But in a zero base it will be easier - I tried to write everything there in English. You can simply translate it into your language wherever you need it!

If you have Windows 10 or higher installed (it should also work on Windows 8.1, but this has not been tested), the following steps can be performed without problems.

1. **Creating folders and the database**

   - Create a folder on your computer to store Firebird databases (for example, *FB*).
   - Inside it, create a folder for the demo database (for example, *demoFB*).
   - Move the files *MYDB9.FDB* and *MYDB3.fbk* from the distribution into this folder.
   - Copy the file *fbclient.dll* from the Firebird folder into the demo database folder.

2. **Copying additional files**

   - Copy the *My4s* folder from the distribution to a convenient location on your computer.

3. **Installing Firebird**

   - Install a stable version of Firebird 3.x (this is sufficient for learning; it is lighter in size).
   - Versions 4.x or 5.x can also be used, but for initial study, 3.x is recommended.

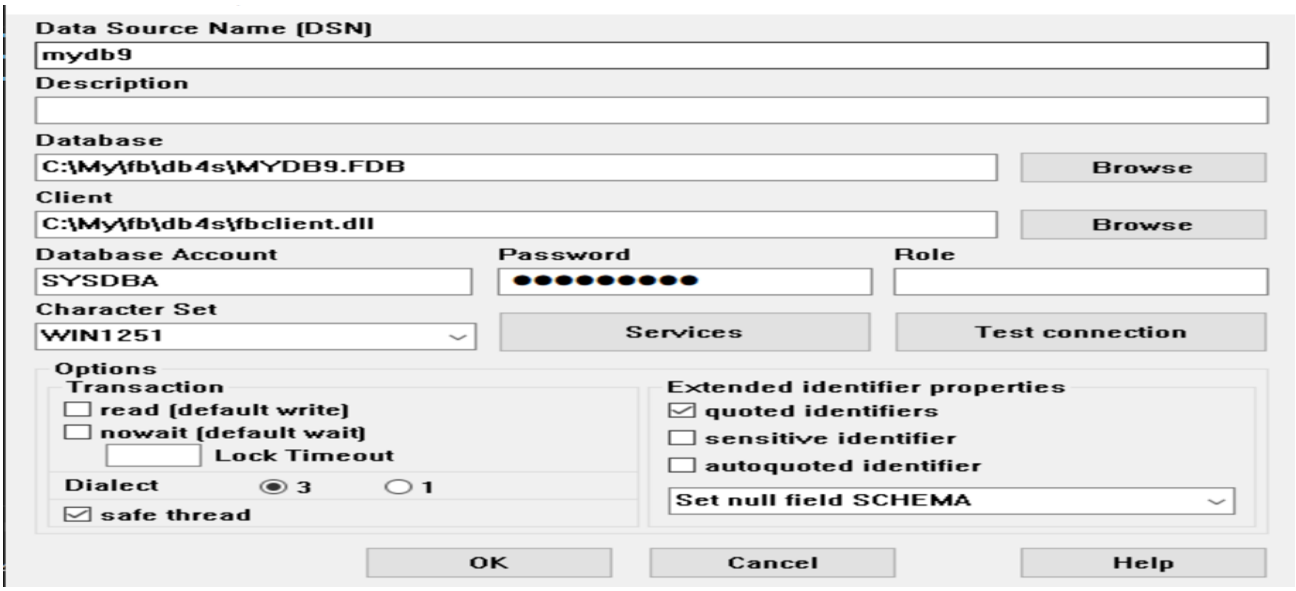
4. **Installing the ODBC driver**

   - Download the driver *Firebird_ODBC_2.0.5.156_x64.exe* (or a newer version if available).
   - Install the driver.

5. **Connecting the driver in Windows**

   - Type "odbc" in the Windows search bar.
   - Open *ODBC Data Source*.

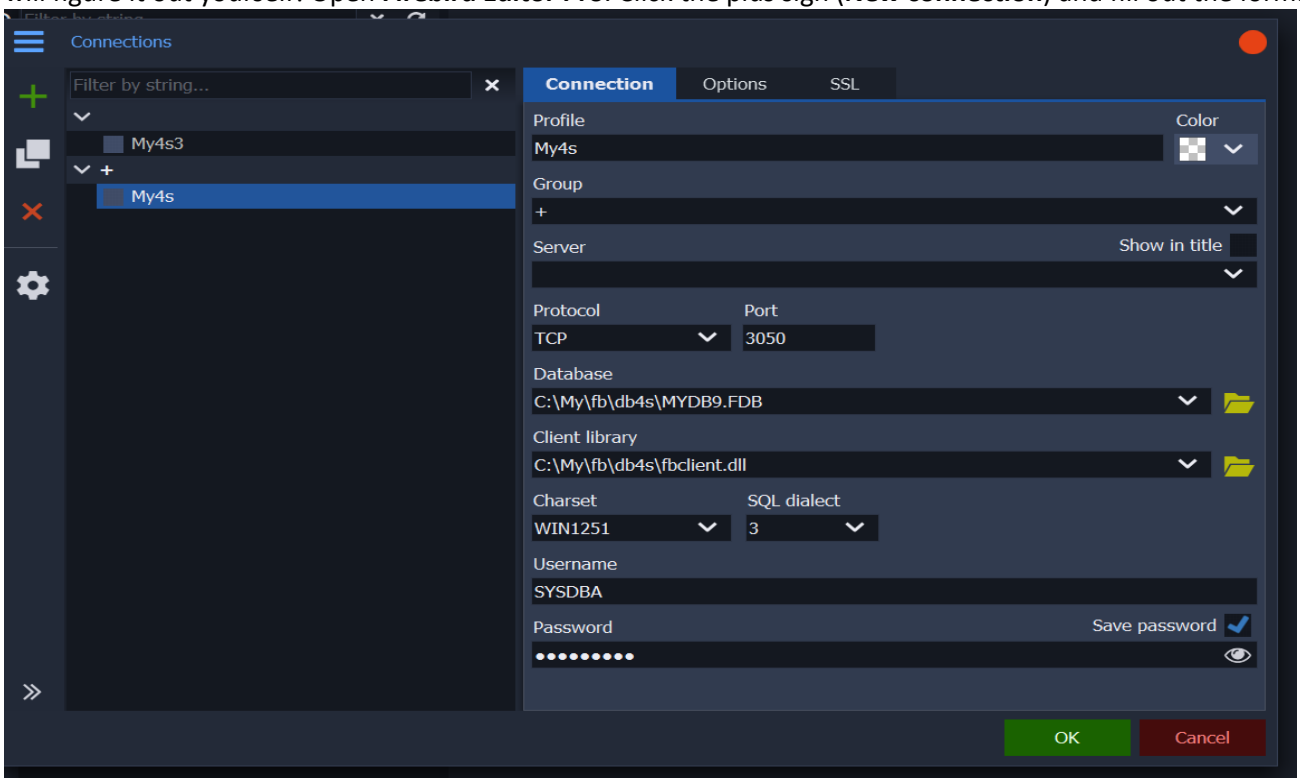
- Add a new DSN (*Add…*).
- Open the *Setup* window to configure the connection.



Scr86

Fill it out as shown in the screenshot. For working with the demo database, the DSN must be **mydb9** and the Character Set must be **WIN1251** (I already have this specified in the settings). Next, correctly fill in **database** and **client**. The password is **masterkey**, and everything else should be entered as in the screenshot. Then click **Test connection** – if there is no error, click **OK**.

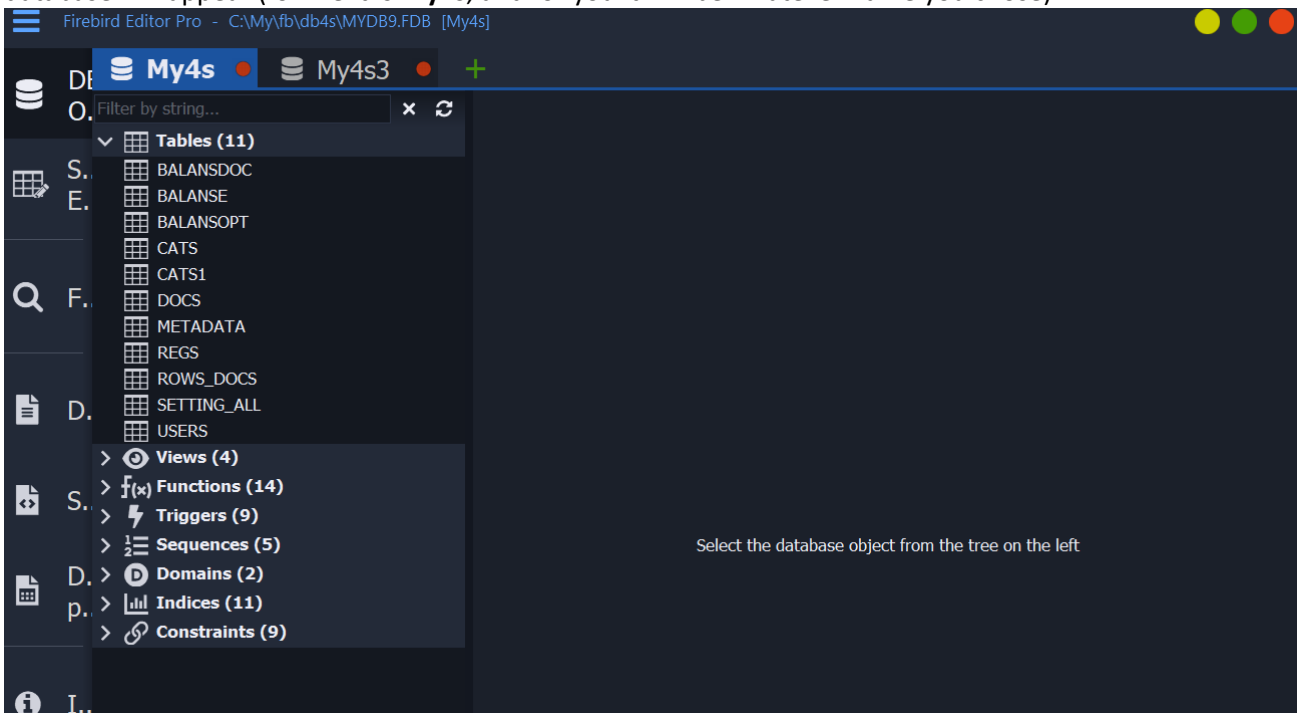
Install **Firebird Editor Pro**. If you already have another editor, then you are probably an experienced Firebird user and will figure it out yourself! Open **Firebird Editor Pro**. Click the plus sign (**New connection**) and fill out the form.



Scr87

For me, it is **My4s**, but you can name it as you like, for example **demo4s**. In **Database**, select the file with the demo database, and in **Client library**, select the file *fbclient.dll*. For the demo database, the standard password is **masterkey**. Everything else should be filled in as shown in the screenshot. Click **OK**. If you have entered everything correctly, the demo

database will appear (for me it is **My4s**, and for you it will be whatever name you chose).



Scr88

Next, install Node.js — the latest stable version (LTS). If you don't care about the options, you can choose the default installation. After that (if not already installed), set up the stable version of Visual Studio Code. Launch the editor and open the folder (it should already have been copied from the distribution) **My4s** via the menu *File → Open Folder*. On the left, a navigation column will appear with all the contents of the **My4s** folder. You will be able to open any files, view their contents, and make changes if necessary. With a single click, you can update the files and immediately see the results in the browser (without having to start My4s from scratch again — just reload the page related to your changes!).

Visual Studio Code is a powerful text editor. In our case, its main advantage lies in strong support for **Node.js** and the ability to debug modules. In addition, the editor supports various interface languages, which makes working with it more convenient. You can find a detailed description on the internet. Debug mode is also available!

Working with My4s on Windows 7

If your computer is running Windows 7 — don't worry. In the *Windows7* distribution you will find:

- **Node.js version 12** (compatible with Windows 7)
- A folder with a ready-to-use **VS Code**, which will also work on Windows 7

Steps for setting up the database

1. For getting acquainted with My4s, the demo version is sufficient.
2. If you want to create your own database from scratch:
    - In the distribution, there is a folder *my4sNull* containing:
        - the initial database *mydb3.FDB*
        - the backup copy *mydb3.fbk*
    - In the *FB* folder on your computer (or another folder with a different name), create a new folder *MyFb*.
    - Move the files from the *my4sNull* distribution into this folder.
3. In the *My4s* folder, there is already a template for the initial database in the *mydb3* folder.
4. In Windows settings, add a new DSN *mydb3* for the ODBC driver.
5. In Firebird Editor Pro, create a new connection (for example, *My4s3*, as described above).

Verification of operation

- If everything has been done correctly, you will be able to work with two databases in the browser at once:

    - Demo database: **127.0.0.1:1637/mydb/cf** or **127.0.0.1:1637/mydb/db**
    - Your database: **127.0.0.1:1637/mydb3/cf** or **127.0.0.1:1637/mydb/db**
- The default password is **masterkey**. You can change it to any other.

💡 If it doesn't work the first time, don't be upset. That's normal—I didn't get it right immediately either! Just reread the description carefully.

For programming "from scratch" or working with the demo database on a local computer, it is quite sufficient to use VS Code — since you can directly run **Node.exe** through the *Run* menu.

But what should you do once you have already created your program, debugged it, and placed the **My4s** folder on the server so that many users can work with the database via the internet or a local network?

For this purpose, on the server where **Node.exe** is installed (remember: in My4s the server is just a single file, **Node.exe**, which must be located in the *My4s* folder), a special batch file **Start_Node.bat** is provided.

Its contents consist of just one line: **node.exe Start4s.js**

When launched, **Node.exe** receives the file **Start4s.js**, which:

- listens to the port (in this case, **1637**),
- contains the main routing,
- connects other files from the *My4s* folder as needed.

It is enough to run this batch file — and you will be able to connect to the database directly from the browser!

Important: on the server, it is not necessary to install the entire Node.js package. The key requirement is to have the correct **Node.exe** file of the release needed for operation.

Attention! You cannot run Node simultaneously from VS Code and from the batch file if you are working on the same computer!

Contact with the author:

Email address for contact: sergey2429@i.ua

I will be glad if you find any inaccuracies, bugs, or—God forbid—errors. Be sure to describe under what conditions this occurs! If you improve any part of the program yourself, describe in detail what you did. And if I consider that it truly improves the program, I will gladly update it and credit you as the author of that part!

License and usage

My4S can be used free of charge in any environment, including but not limited to: personal, academic, commercial, governmental, corporate, non-profit, and commercial applications. "Free of charge" means there is no fee for installing and using My4S. Anyone who obtains a copy of this software has the right to use it without restrictions, including: use, copying, modification, publication, distribution, and granting access to other users with the same rights.

Copyright

All programs created with My4S must indicate that they were developed using this software.

Disclaimer of warranties

The software is provided "AS IS", without any warranties, express or implied, including but not limited to: warranties of merchantability, fitness for a particular purpose, or non-infringement of

third-party rights. The authors and copyright holders are not liable for any claims, damages, or other obligations arising from the use of this software.

Would you like me to format the entire license section (usage, copyright, disclaimer) into a clean Markdown block so it can be directly inserted into your documentation?