Инструкция по развертыванию экземпляра ПО

Оглавление

1. Разработка в среде Ragtime, предварительные настройки

Прежде, чем начать разработку в среде Ragtime, установите следующие обязательные компоненты:

1. Postgre SQL Server
   Установка Postgre SQL Server для работы в среде Ragtime выполняется как обычно и не имеет каких-либо особенностей.
2. .Net Core SDK 3.1
   Скачайте и установите последнюю версию .Net Core SDK 3.1 с официального сайта Microsoft
   https://dotnet.microsoft.com/download/dotnet/3.1.
   Необходимо установить два пакета – Hosting bundle и Desktop runtime.
3. Redis Server
   Для работы приложения, созданного на основе Ragtime, необходимо установить и настроить Redis Server. Установка – стандартная. Настройки после установки:
   - Изменить `redis.windows-service.conf`, установить `notify-keyspace-events AKE`
   - Инсталлировать сервис командой `redis-server --service-install redis.windows-service.conf`
4. Настройка шаблонов Ragtime
   Для создания новых проектов в среде .Net используются шаблоны. Шаблон Ragtime необходимо зарегистрировать. Для этого необходимо скопировать во временную папку файл
   `ragtime.application.starter.3.6.6.nupkg`
   и выполнить команду установки шаблона
   `dotnet new -i ragtime.application.starter.3.6.6.nupkg`
   Важное замечание: здесь 3.6.6 это номер сборки Ragtime. В будущем, с выходом новых версий, этот номер может измениться. Используйте актуальную версию.
   После выполнения команды установки шаблона выполните команду
   `dotnet new -l`.
   Она отобразит список всех установленных шаблонов. Убедитесь, что среди них присутствует шаблон **`Ragtime starter application`** (короткое имя - **`Ragtime.Application.Starter`**).

2. Создание первого тестового приложения Ragtime

Создайте для тестового проекта отдельную папку. Важное условие – название папки должно быть идентификатором, т.е. содержать только латинские буквы, цифры и символ ‘_’. Первый символ не должен быть цифрой.
Примеры:

- MyFirstApplication
- Test_01

В дальнейшем описании предполагается, что папка проекта называется MyFirstApplication.

Выполните в этой папке команду

```
dotnet new ragtime.application.starter
```

В папке будут созданы следующие дочерние папки и файлы:

- **MyFirstApplication.sln** – файл решения .Net
- **MyFirstApplication.Main** – проект для размещения объектов
- **MyFirstApplication.Web** – проект оболочки веб-сервера
- **.eslintrc** – вспомогательный файл, изменять его вручную не следует
- **.gitattributes** и **.gitignore** – настройки хранения репозитория Git
- **_start.bat** – команда запуска приложения
- **Product.props** – единые настройки версий импортируемых пакетов, в т.ч. Ragtime

Создайте базу данных приложения с именем MyFirstApplication и укажите настройки подключения к ней в файле

```
MyFirstApplication.Web\appsettings.development.json
```

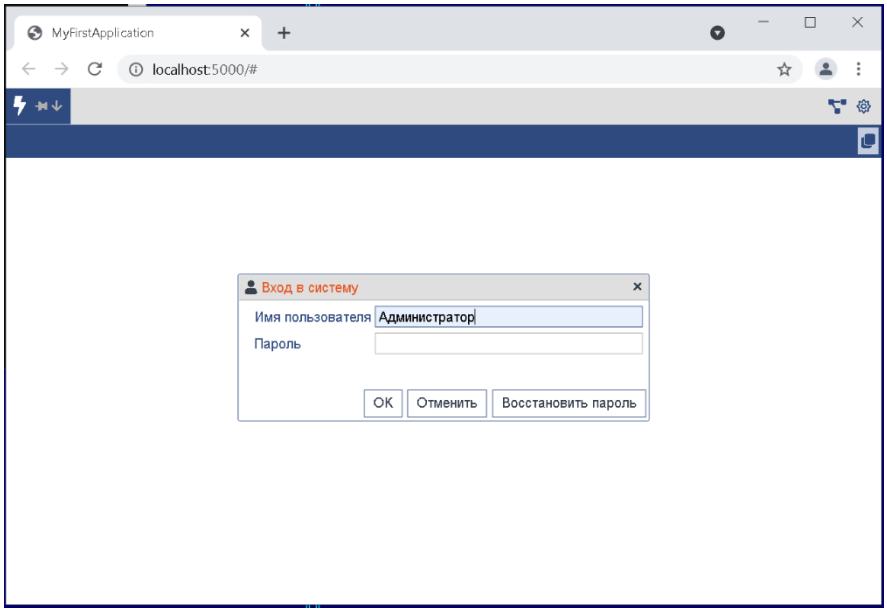
Это файл в формате json. Настройки подключения приложения к базе данных находятся в разделе data -> db-> Main и имеют следующий вид:

```
"Main": {
  "Server": "localhost", -- укажите здесь имя сервера баз данных
  "Database": "MyFirstApplication", -- укажите здесь имя базы данных
  "User": "postgres", -- укажите здесь имя владельца базы данных
  "Password": "111", -- укажите пароль этого пользователя
  "ProviderName": "PostgreSQL" -- имя SQL-адаптера
}
```

Постройте приложение. Для этого выполните в папке приложения команду

```
dotnet build
```

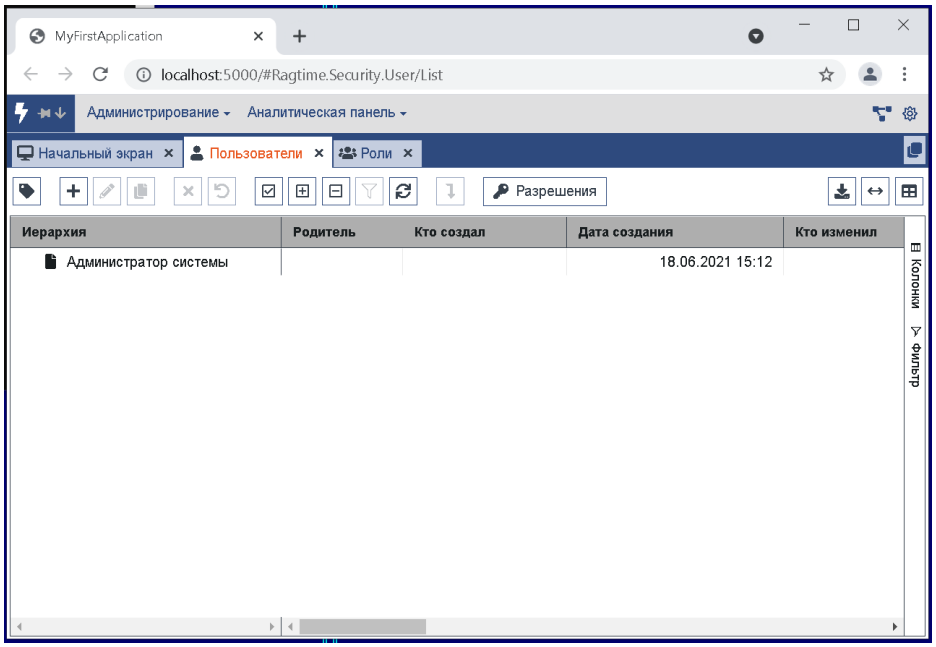
Первое тестовое приложение Ragtime готово к запуску. Запустите его командой _start.bat



Войдите с учетной записью «Администратор», без пароля.

Вам доступны следующие возможности:

- Смена пароля.
- Управление пользователями – добавление, удаление, назначение ролей.
- Управление ролями. Роль - это набор разрешений доступа к объектам приложения – справочникам, документам, регистрам и т.д. Создание этих объектов и их настройки безопасности будут рассмотрены в следующих примерах.



2.1. План разработки тестового приложения

В качестве примера мы создадим простейшее приложение по учету товаров на складе. Описание товаров будет храниться в справочнике, покупку и продажу товаров будем оформлять документом «Накладная», так мы сможем отслеживать каждую операцию по

движению товаров на складе. Этот документ будет делать записи в регистр «Обороты товаров», с помощью которого мы будем отслеживать актуальные остатки товарных позиций, а также вычислять прибыль от торговых операций. Результаты выведем в виде отчета.

3. Создание объекта на основе RefObject

В приложениях Ragtime основными объектами данных являются объекты, построенные на основе RefObject. Основные особенности таких объектов:

1. Эти объекты хранятся в базе данных. Ragtime самостоятельно обеспечивает сохранение, чтение, изменение, удаление и восстановление после удаления таких объектов.
2. Каждый объект имеет поле Ref типа Guid, которое является уникальным идентификатором объекта.
3. Каждый объект имеет свойство Presentation, оно предназначено для визуального отображения, например, ссылки на объект.
4. Объект может иметь поля, а также одну или несколько табличных частей, в том числе иерархических.
5. Объект может иметь редакции (версии).
6. Объект может иметь статусную модель – набор статусов и переходов между ними, обеспечивающих бизнес-логику приложения.

Для иллюстрации этих возможностей создадим несколько объектов на основе RefObject. Пусть первым объектом будет «Справочник единиц измерений».

3.1. Создание метаданных объекта

Для того, чтобы добавить в приложение объект RefObject, необходимо создать файл метаданных. В данном примере это будет файл ref_cmn_OKEI.metadata, который мы поместим в папку MyFirstApplication\MyFirstApplication.Main\Src\RefBooks

Содержимое файла:

```
<?xml version="1.0" encoding="utf-8" ?>
<RefObject xmlns="Ragtime.Metadata,Ragtime.Core"
 Id="3b879463-f03c-4940-a229-adfb0dfc3b97"
 ToolsetVersion="1,2"
 Namespace="RefBooks"
 Name="ref_cmn_OKEI" SchemaVersion="3">

 <Doc>Единицы измерения (ОКЕИ)</Doc>
 <Security Guard="ref_cmn_OKEI_Guard" />
 <Ui Caption="ОКЕИ" CaptionPlural="ОКЕИ" />

 <Field Name="Code" Ui.Caption="Код" Type="string" Storage.Size="10" Rules.IfEmpty="Error"
Rules.IfTooLong="Error" />
 <Field Name="Name" Ui.Caption="Наименование" Type="string" Storage.Size="30"
Rules.IfEmpty="Error" Rules.IfTooLong="Error" />
 <Field Name="FullName" Ui.Caption="Полное наименование" Type="string" Storage.Size="150"
Rules.IfTooLong="Error" />
 <Field Name="NationalSymbol" Ui.Caption="Нац. усл. обозн." Type="string" Storage.Size="30"
Rules.IfTooLong="Error" Ui.Description="Национальное условное обозначение" />

 <Field Name="Presentation" Type="string" Storage.Size="50" Rules.IfTooLong="Truncate" />

</RefObject>
```

Пояснения к содержимому

1. Файл метаданных имеет формат xml, корневой элемент имеет имя RefObject в пространстве `Ragtime.Metadata,Ragtime.Core`
2. Атрибут Id – уникальный Guid типа объекта, определяемого этим файлом метаданных.

3. Атрибут ToolsetVersion – версии инструментов Ragtime, на текущий момент 1,2.
4. Атрибут Namespace – пространство имен для создаваемого объекта.
5. Атрибут Name – имя создаваемого объекта.
6. Атрибут SchemaVersion – версия формата файла метаданных, на текущий момент – 3.
7. Атрибуты могут быть записаны как подузлы, например вместо Id="…" можно писать <Id>…</Id>. Порядок атрибутов и подузлов не анализируется и роли не играет.
8. Узел Doc позволяет программисту добавить к объекту краткую документацию. Пользователю приложения она не отображается.
9. Узел Security позволяет указать имя вспомогательного объекта, который будет отвечать за настройки безопасности создаваемого объекта.
10. Узел Ui позволяет указать наименования объекта для пользователя. Наименование во множественном числе отображается при просмотре списка объектов, в единственном – при отображении одного объекта.

Далее идет описание полей объекта. Для справочника единиц измерений мы добавили поля «Код», «Наименование», «Полное наименование», «Национальное условное обозначение». Кроме того, мы задали размер предопределенного поля Presentation.
Атрибуты полей:

1. Name – идентификатор соответствующего поля объекта.
2. Ui.Caption – заголовок поля, видимый пользователю.
3. Type – тип объекта. Определяет и тип, который будет использован в коде приложения, и топ поля в базе данных.
4. Storage.Size – размер хранимых данных, например, для строк.
5. Rules.* - набор атрибутов, описывающих типовые правила заполнения полей. Например, Rules.IfEmpty="Error" говорит о том, что поле должно быть обязательно заполнено до сохранения объекта в базу, Rules.IfTooLong="Error" – проверка, что значение поля не слишком длинное и, если слишком, то отображение ошибки. Аналогично, Rules.IfTooLong="Truncate" – проверка, что значение поля не слишком длинное и, если слишком, то усечение его до указанной в Storage.Size длины.

3.2. Создание кода объекта

Для иллюстрации минимально необходимой функциональности объекта добавим рядом с файлом метаданных файл ref_cmn_OKEI.cs с содержимым:

```
namespace RefBooks {
  using System;
  using System.Linq;
  using Ragtime;

  public partial class ref_cmn_OKEI: Ragtime.Security.IAny {

    partial void GetPresentation() {
      Presentation = Name;
    }

    partial void DoValidate(Func<ref_cmn_OKEI> getOld, Func<EntityDiff> getDiff) {
      if(Records().Any(_ => _.Code == Code && _.Ref != Ref))
        throw new ApplicationError($"Справочник \"{Metadata.Ui.Caption}\": Запись с кодом
\"{Code}\" уже существует");
    }

    partial void DoMakeNew(Guid sourceRef) {
    }
```

```
    partial void DoAfterStore() {
    }

    partial void DoAfterKill() {
    }

    partial void DoAfterRevive() {
    }

  }

  public class ref_cmn_OKEI_Guard: Ragtime.RefObject.RefObjectGuard {
    public ref_cmn_OKEI_Guard() : base("ref_cmn_OKEI", "Общее", "Единицы измерения", new
Guid("4B252298-7320-483F-942C-80A36DDA47D2")) { }
  }
}
```

Пояснения к содержимому

1. Пространство имен RefBooks, а также имена объектов `ref_cmn_OKEI` и `ref_cmn_OKEI_Guard` – те, которое указано в файле метаданных.
2. Объект `ref_cmn_OKEI_Guard` максимально прост – в базовый конструктор нужно передать имя защищаемого объекта, имя папки и название элемента в ней для страница настройки разрешений, а также уникальный Guid объекта-защитника.
3. Пустые методы `DoMakeNew`, `DoAfterStore` и т.д. представлены как примеры обработчиков событий, которые возникают, например, при создании нового объекта из существующего, после сохранения, после удаления или восстановления из удаленный.
4. Метод `GetPresentation` позволяет программисту создать текстовое описание объекта.
5. Метод `DoValidate` вызывается перед сохранением объекта и позволяет выполнить дополнительные проверки, необходимые для работы приложения. В данном случае выполняется проверка на уникальность поля «Код» и, если оно не уникально, пользователю выдается сообщение об ошибке – для этого возбуждается исключение типа ApplicationError.

3.3. Добавление формы просмотра объекта

Для того, чтобы пользователь мог работать с объектом, Ragtime автоматически генерирует форму списка объектов. Внешний вид формы для работы с одним объектом задает программист. Для данного примеры сделаем это с помощью файла ref_cmn_OKEI.tsx, который расположим рядом с файлом метаданных. Содержимое файла:

```
import * as AppCommand from "Ragtime.AppCommand";
import * as Form from "Ragtime.Form";
import { Field } from "Ragtime.Ui.FieldGroup";
import { FieldGroup } from "Ragtime.Ui.FieldGroup";

import * as OKEI from "RefBooks.ref_cmn_OKEI";


export class ItemForm extends OKEI.ItemForm {
  behaviorOptions: Form.BehaviorOptions = {
    preferModal: true,
  };

  getMainContent() {
```

```
    return (
      <FieldGroup>
        <Field {...this.model.Code} />
        <Field {...this.model.Name} />
        <Field {...this.model.FullName} />
        <Field {...this.model.NationalSymbol} />
      </FieldGroup>
    );
  }
}

export var command: AppCommand.Options = { text: "ОКЕИ", id: "RefBooks.Cmn.OKEI", handler: ()
=> new OKEI.ListForm().show() };
```

Пояснения к содержимому

1. По файлу метаданных Ragtime генерирует для каждого RefObject отдельный модуль, которые можно импортировать по его имени. В данном случае это `RefBooks.ref_cmn_OKEI`. В этом модуле определена табличная форма для просмотра списка объектов, а также базовая (пустая) форма просмотра одного элемента.
2. Мы наследуемся от базовой формы `OKEI.ItemForm` и определяем метод `getMainContent` – этот метод будет возвращать элемент JSX с разметкой формы.
3. Для того, чтобы сделать просмотр объекта доступным для пользователя мы добавляем и экспортируем команду табличного просмотра. Эта команда, как видно из текста, создает и отображает пользователю форму табличного просмотра списка объектов.

Чтобы вывести команду просмотра списка объектов, модифицируем главное меню приложения, оно находится в файле `MyFirstApplication.Main\Src\App\Main.ts` Добавим в него новый раздел в меню и команду в нем:

```
function getCommandInterface() {
  return new CommandInterface()

    .folder("Справочники", f => {
        f.command("RefBooks.Cmn.OKEI");
    })
    .folder("Администрирование", f => {
      f.folder("Безопасность", subf => {
        subf.command("Ragtime.Security.Users");
```

3.4. Миграция базы данных

Прежде, чем объект можно будет использовать в приложении, необходимо внести изменения в структуру базы данных. Делается это с помощью миграций, которые должны быть расположены в отдельном проекте (таких проектов может быть несколько). В этом примере мы добавили в проект MyFirstApplication.Db папку Main_0001_Alice, а в нее – файл Transition.cs с содержимым

```
namespace Main_0001_Alice {
  using global::Ragtime.Data.DbUpgrade;
  using LinqToDB;
  using System.Linq;

  [Use("RefBooks.ref_cmn_OKEI", "refbook")]
  public partial class Transition : global::Ragtime.Data.DbUpgrade.Transition {

    public override void UpdateSchema() {
      refbook.Create();
    }
```

```
    public override void UpdateData() {

    }

  }

}
```

Пояснения к содержимому

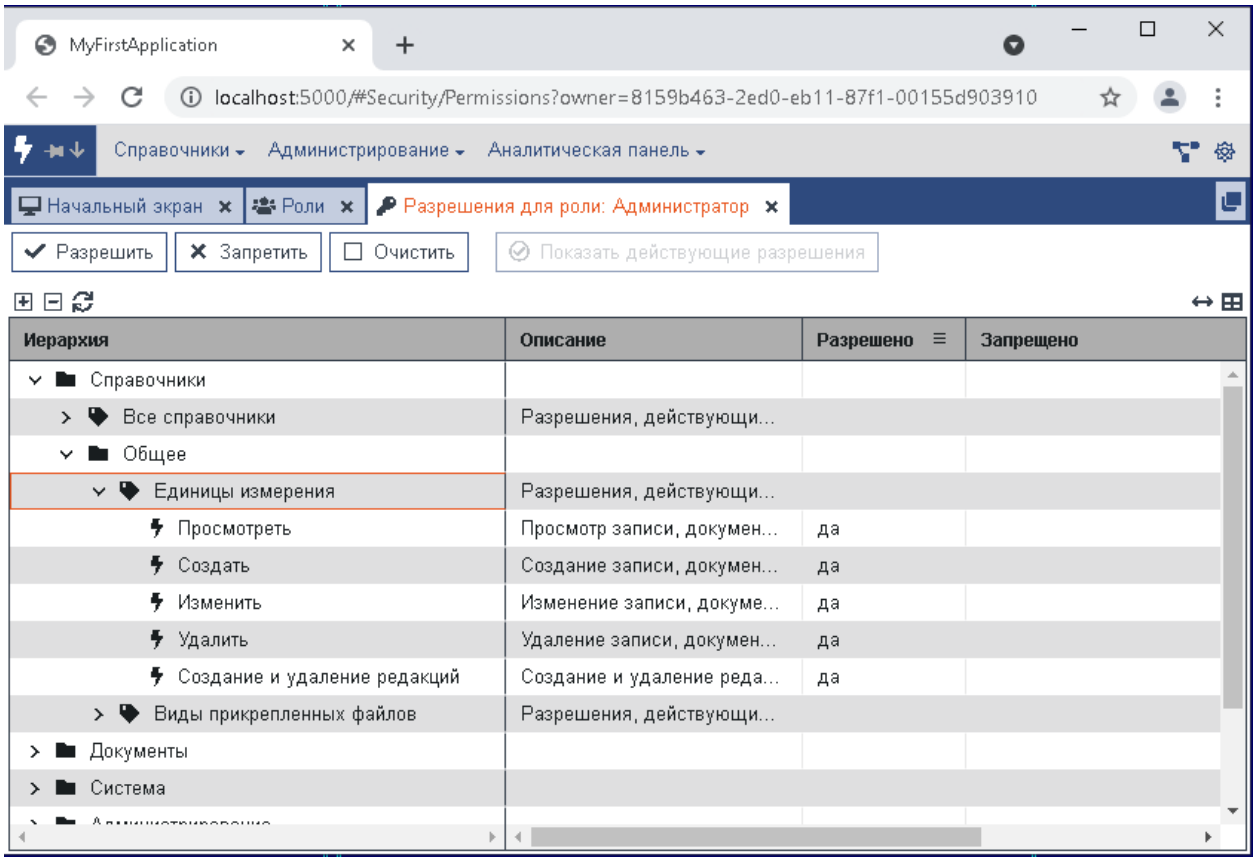
1. Миграции имеют номер и суффикс. Миграции с одинаковым суффиксом исполняются последовательно, наличие суффиксов позволяет нескольким программистам создавать миграции параллельно и избегать при этом конфликтов. В данном примере мы создали миграцию с номером 0001 и суффиксом Alice.
2. Внутри миграции мы указываем класс Transition, в его атрибутах указываем, какие именно прикладные объекты будут использованы в миграции.
3. Для изменения структуры данных необходимо переопределить метод `UpdateSchema`. В данном примере мы создаем в базе все необходимое для работы прикладного объекта `ref_cmn_OKEI`.

3.5. Сборка и запуск приложения

Собираем приложение командой dotnet build и запускаем его. В процессе запуска можно наблюдать вывод сведений об обновлении базы данных – исполнении миграций.
Обратите внимание, несмотря на то, что мы добавили команду в главное меню, эта команда не отображается. Причина этого в том, что справочник защищен настройками безопасности и для работы с ним в первую очередь нужно открыть к нему доступ.

3.6. Настройка безопасности для объекта ref_cmn_OKEI

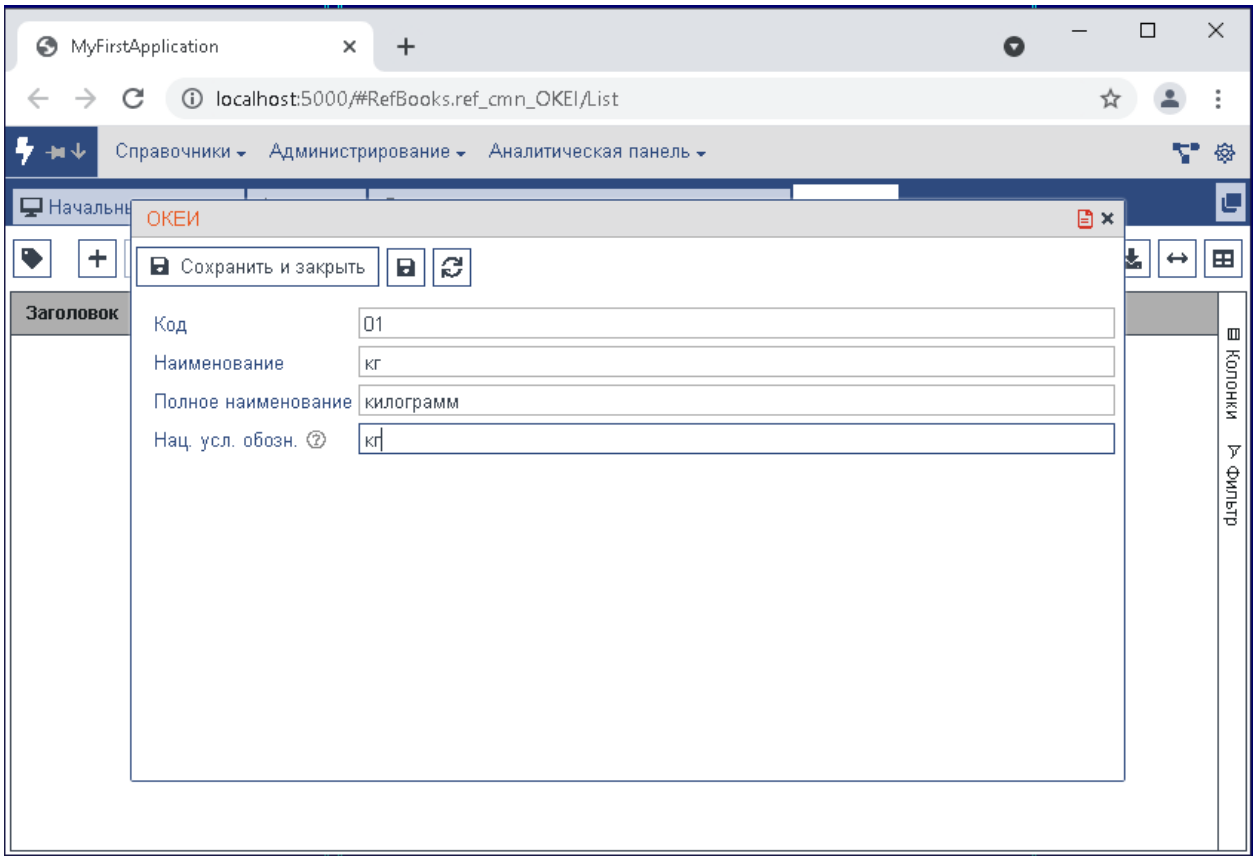
Открываем настройки разрешений для роли «Администратор»



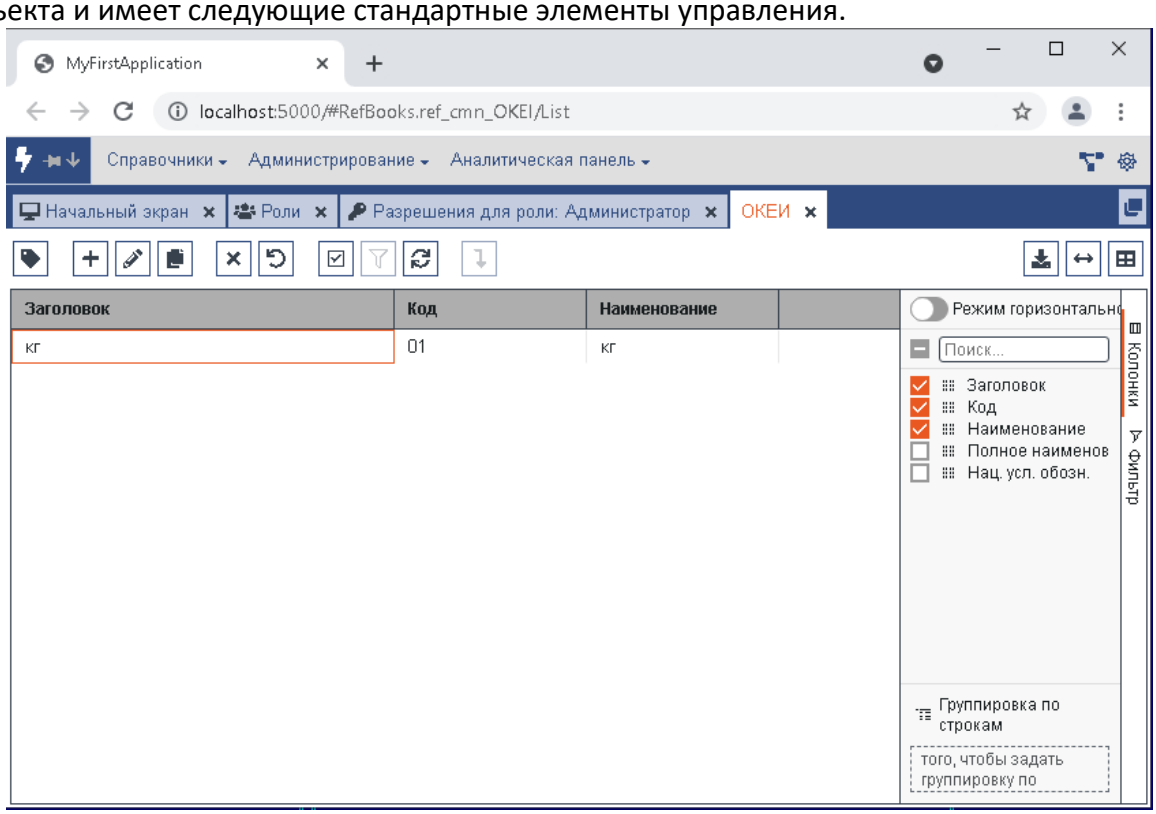
Видим, что в секции «Справочники» -> «Общее» появился новый справочник «Единицы измерения». Это результат работы созданного нами объекта `ref_cmn_OKEI_Guard`. Разрешим все операции с этим справочником с помощью кнопки «Разрешить». При этом в главном меню у нас сразу же появится меню «Справочники», а в нем – команда «ОКЕИ». Это результат добавления и экспорта команды в tsx-модуле и изменений в главном меню.

3.7. Работа с новым справочником

Выполните команду главного меню «Справочники» -> «ОКЕИ». На экране появится табличный просмотр списка записей. Добавим в справочник первую запись.



Внешний вид формы определен нами в методе `getMainContent`. Сохраним запись. Табличный просмотр Ragtime сгенерировал для нас автоматически на основе метаданных объекта и имеет следующие стандартные элементы управления.



1. Команды добавления, редактирования и копирования записей.

2. Команды удаления и восстановления из удаленных.
3. Команды групповой пометки записей, фильтрации и обновления списка.
4. Справа расположены выпадающие настройки полей, выводимых на просмотр, а также фильтров, ограничивающих выборку данных.

Все эти элементы управления являются типовыми для всех объектов RefObject. При необходимости, в соответствии с требованиями бизнес-логики приложения, программист может как менять поведение этих команд, удалять их и/или добавлять.

4. Связи объектов, созданных на основе RefObject

Объекты, созданные на основе RefObject, могут ссылаться друг на друга, и ниже мы покажем, как это работает. Но сначала добавим в проект еще один справочник «Товары». Первоначально запись этого справочника будет иметь всего два поля – Код и Наименование. Действия по его созданию аналогичны предыдущему примеру:

1. Добавляем файл метаданных ref_cmn_Mat.metadata

```
<?xml version="1.0" encoding="utf-8" ?>
<RefObject
 Id="b183f792-a4e8-4b08-907b-267725d86457"
 Namespace="RefBooks"
 Name="ref_cmn_Mat"
>

 <Doc>Материалы</Doc>
 <Security Guard="ref_cmn_Mat_Guard" />
 <Ui Caption="Товар" CaptionPlural="Товары" />

 <Field Name="Code" Ui.Caption="Код" Type="string" Storage.Size="10" Rules.IfEmpty="Error"
Rules.IfTooLong="Error" />
 <Field Name="Name" Ui.Caption="Наименование" Type="string" Storage.Size="255"
Rules.IfEmpty="Error" Rules.IfTooLong="Error" />

 <Field Name="Presentation" Type="string" Storage.Size="255" Rules.IfTooLong="Truncate" />

</RefObject>
```

2. Добавляем код, в т.ч. объект безопасности – файл ref_cmn_Mat.cs

```
namespace RefBooks {
  using System;
  using System.Linq;
  using Ragtime;

  public partial class ref_cmn_Mat: Ragtime.Security.IAny {

    partial void GetPresentation() {
      Presentation = Name;
    }

    partial void DoValidate(Func<ref_cmn_Mat> getOld, Func<EntityDiff> getDiff) {
      if(Records().Any(_ => _.Code == Code && _.Ref != Ref))
        throw new ApplicationError($"Справочник \"{Metadata.Ui.Caption}\": Запись с кодом
\"{Code}\" уже существует");
    }
  }

  public class ref_cmn_Mat_Guard: Ragtime.RefObject.RefObjectGuard {
    public ref_cmn_Mat_Guard() : base("ref_cmn_Mat", "Общее", "Товары", new Guid("b000e293-
2c19-479b-8cf7-745227c245a6")) { }
  }
}
```

3. Добавляем форму ввода/просмотра/редактирования записи – файл ref_cmn_Mat.tsx

```
import * as AppCommand from "Ragtime.AppCommand";
import * as Form from "Ragtime.Form";
import { Field } from "Ragtime.Ui.FieldGroup";
import { FieldGroup } from "Ragtime.Ui.FieldGroup";

import * as Mat from "RefBooks.ref_cmn_Mat";


export class ItemForm extends Mat.ItemForm {
  behaviorOptions: Form.BehaviorOptions = {
    preferModal: true,
  };
```

```
  getMainContent() {
    return (
      <FieldGroup>
        <Field {...this.model.Code} />
        <Field {...this.model.Name} />
      </FieldGroup>
    );
  }
}

export var command: AppCommand.Options = { text: "Товары", id: "RefBooks.Cmn.Mat", handler: ()
=> new Mat.ListForm().show() };
```

4. Добавляем миграцию базы данных – файл Transition.cs в папке Main_0002_Alice

```
namespace Main_0002_Alice {
  using global::Ragtime.Data.DbUpgrade;
  using LinqToDB;
  using System.Linq;

  [Use("RefBooks.ref_cmn_Mat", "refbook")]
  public partial class Transition : global::Ragtime.Data.DbUpgrade.Transition {

    public override void UpdateSchema() {
      refbook.Create();
    }

    public override void UpdateData() {
    }
  }
}
```

5. Добавляем команду в главное меню

```
    .folder("Справочники", f => {
        f.command("RefBooks.Cmn.OKEI");
        f.command("RefBooks.Cmn.Mat");
    })
```

Итак, новый справочник «Товары» готов к использованию (не забудьте разрешить доступ к нему в настройках безопасности), но, на текущий момент, в нем явно не хватает важного атрибута – единицы измерения. Добавим это поле как ссылку на справочник единиц измерений ref_cmn_OKEI, который был создан на предыдущем шаге. Для этого:

1. Добавим поле в метаданные:

```
<Field Name="OkeiRef" Ui.Caption="Ед.изм." RefTo="ref_cmn_OKEI" Rules.IfEmpty="Error" />
```

1. Добавим поле в форму просмотра

```
<Field {...this.model.Okei} />
```

2. Изменим визуальное представление объекта

```
    partial void GetPresentation() {
      Presentation = $"{Name} ({Okei.Name})";
    }
```

3. Добавим миграцию базы данных

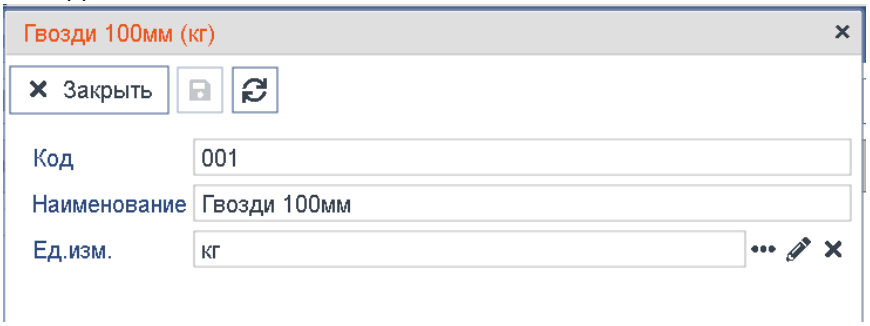
```
namespace Main_0003_Alice {
  using global::Ragtime.Data.DbUpgrade;
  using LinqToDB;
  using System.Linq;

  [Use("RefBooks.ref_cmn_Mat", "refbook")]
  public partial class Transition : global::Ragtime.Data.DbUpgrade.Transition {

    public override void UpdateSchema() {
      refbook.AddField("OkeiRef");
    }
  }
}
```

Важные моменты, на которые нужно обратить внимание:

1. В метаданных имя ссылочного поля оканчивается на «Ref», а вместо атрибута «Type» используется атрибут «RefTo».
2. В миграции имя поля также указывается с суффиксом «Ref».
3. Во всех остальных местах – и в коде c#, и в typescript имя поля указывается без суффикса. Связано это с тем, что везде, кроме миграции и метаданных ссылочное поле это объект с типом, который указан в «RefTo». Этот объект никогда не бывает null. Проверить, заполнено ли ссылочное поле или нет можно, проверив его Ref. Например:

```
If(mat.Okei.Ref == Guid.Empty) throw new ApplicationError("Не указана единица измерения");
```

Соберем и запустим проект. В карточке Товара добавилось новое поле, которое отличается от обычных полей данных:



1. Значение поля нельзя ввести вручную. При попытке ввода откроется форма поиска, в которой нужно выбрать подходящее значение из имеющихся.
2. Поле имеет дополнительные элементы управления – кнопки поиска, просмотра/редактирования, а также команду очистки значения.

5. Возможности объектов на основе RefObject

Данный пример демонстрирует некоторые базовые возможности объектов на базе Refobject такие, как:

1. Табличные части
2. Автоматическая нумерация
3. Вычисляемые поля
4. Альтернативный дизайн формы просмотра
5. Статусная модель и настройки безопасности для нее
6. Протоколирование истории изменений
7. Использование вложений
8. Редакции
9. Обсуждения

Для демонстрации этих возможностей создадим новый объект – документ «Накладная». Этот документ будет выполнять простейшие функции учета товаров, у него будет Тип – покупка или продажа, а также строки товаров.

5.1. Метаданные документа

Создадим новый файл метаданных doc_WayBill.metadata со следующим содержимым

```
<?xml version="1.0" encoding="utf-8" ?>
<RefObject
 Id="0A5C7CC1-52C3-400A-B0E8-AED6CABFCDB0"
 Namespace="Docs"
 Name="WayBill"
>

 <Doc>Накладные на покупку и продажу товара</Doc>
 <Security Guard="WayBill_Guard" />
 <Storage Name="doc_WayBill" />
 <Ui Caption="Накладная" CaptionPlural="Накладные" />

 <Field Name="DocNo" Type="string" Storage.Size="50" Ui.Caption="№ документа"
Rules.IfTooLong="Error" >
 <Numerator Id="9B196858-2272-4191-82FA-F8763131EFDA"
 InitialValue="1"
 AllowReuse="true"
 AllowUserNums="false"
 ContinueUserNum="false"
 AllowDup="false"
 Dimensions="{NumberPostfix()}"
 Format="{number:D6}/{NumberPostfix()}" >
 </Numerator>
 </Field>
 <Field Name="DocDate" Ui.Caption="Дата документа" Type="DateTime" Ui.Format="shortDate" />
 <Field Name="Type" Ui.Caption="Тип накладной" Type="WayBillType" />
 <Field Name="TotalSumma" Ui.Caption="Сумма" Type="decimal" Storage.Type="Money"
Ui.Format="fixedPoint" Ui.Precision="2" Storage.Stored="true" Expression="_ =>
_.calcTotalSum()"/>

 <Field Name="Presentation" Type="string" Storage.Size="255" Rules.IfTooLong="Truncate" />

 <TablePart Name="MatLine" Doc.Summary="Товары">
 <FieldGroup Caption="Свойства товара">
 <Field Name="MatRef" Ui.Caption="Товар" RefTo="RefBooks.ref_cmn_Mat"
Rules.IfEmpty="ErrorIfNotDraft"/>
 <Field Name="OkeiRef" Ui.Caption="Ед.изм." Expression="Mat.OkeiRef" />
 </FieldGroup>
 <FieldGroup Caption="Расчеты">
```

```
 <Field Name="Quantity" Ui.Caption="Количество" Type="decimal" Storage.Type="decimal"
Storage.Param="18" Storage.Param2="5" Ui.Format="fixedPoint" Ui.Precision="5" />
 <Field Name="Price" Ui.Caption="Цена" Type="decimal" Storage.Type="Money"
Ui.Format="fixedPoint" Ui.Precision="2" Totals="false" Storage.Stored="true" />
 <Field Name="Summa" Ui.Caption="Сумма" Type="decimal" Storage.Type="Money"
Ui.Format="fixedPoint" Ui.Precision="2" Storage.Stored="true" Expression="_ => _.calcSum()"/>
 </FieldGroup>
 </TablePart>

</RefObject>
```

Пояснения к содержимому

1. В заголовке документа мы добавили узел Storage. Его назначение – указать, как должна называться таблица объекта в базе данных
   <Storage Name="doc_WayBill" />
2. Для поля DocNo добавлен дополнительный вложенный узел, который описывает алгоритм автоматической генерации номера документа.
   `<Numerator Id="9B196858-2272-4191-82FA-F8763131EFDA"`
   - InitialValue="1" – начальное значение, первый номер в серии номеров
   - AllowReuse="true" – разрешено или нет повторное использование номеров
   - AllowUserNums="false" – разрешены или нет пользовательские номер
   - ContinueUserNum="false" – если номер введен пользователем, следует ли продолжать нумерацию с него
   - AllowDup="false" – разрешены ли повторы номеров
   - Dimensions="{NumberPostfix()}" – разрез формирования номера, в данном случае ежегодно будет начинаться новая серия номеров
   - Format="{number:D6}/{NumberPostfix()}" > -- отображаемый текст номера, а данном случае часть номера формируется функцией NumberPostfix, см. код ниже
   - </Numerator> - Наличие нумератора у поля, помимо прочено, означает, что, помимо поля DocNo у объекта будет еще одно поле DocNo_NumberRef, в котором будет хранится Ref (уникальный идентификатор) присвоенного номера. При копировании документа это поле необходимо очищать, чтобы скопированному документу присвоился новый номер (см. ниже)
3. Для поля «Тип накладной» использован пользовательский тип данных – перечисление (enum):
   `<Field Name="Type" Ui.Caption="Тип накладной" Type="WayBillType" />`
   Описание перечисления находится в отдельном файле enum_WayBillType.metadata. Его содержимое достаточно тривиально:
   ```
   <?xml version="1.0" encoding="utf-8" ?>
   <Enum xmlns="Ragtime.Metadata,Ragtime.Core"
    Id="F515ABDD-D008-4D4F-BEF2-640D8BCFE754"
    Namespace="Docs"
    Name="WayBillType">

    <Doc Summary="Тип накладной" />

    <EnumMember Name="Buy" Caption="Приход" Value="0" />
    <EnumMember Name="Sale" Caption="Расход" Value="1" />

   </Enum>
   ```
4. Для поля «Сумма» в голове документа указано, что это поле является вычисляемым. Поведение вычисляемых полей будет рассмотрено ниже

```
<Field Name="TotalSumma" Ui.Caption="Сумма" Type="decimal" Storage.Type="Money"
Ui.Format="fixedPoint" Ui.Precision="2" Storage.Stored="true" Expression="_ =>
_.calcTotalSum()"/>
```

У вычисляемого поля имеется атрибут Storage.Stored, который указывает, будет ли поле храниться в базе данных или будет исключительно виртуальным, доступным только в составе объекта. Поле, которое хранится в базе, можно использовать в запросах.

5. Добавлена табличная часть MatLine. Это означает, что у объекта WayBill будет доступно свойство MatLines с типом List<WayBill.MatLine>. В нашем примере каждый экземпляр MatLine это одна строка накладной
   ```
   <TablePart Name="MatLine" Doc.Summary="Товары">
   ```
6. Некоторые поля MatLine объединены в группы. Это объединение не влияет на поведение объекта с точки зрения кода, но изменяет его отображение пользователю, см. ниже
   ```
   <FieldGroup Caption="Свойства товара">
   ```
7. Поле «Ед.изм», которое является свойством товара, вынесено как отдельное поле. Его значение определяется с помощью выражения:
   ```
   <Field Name="OkeiRef" Ui.Caption="Ед.изм." Expression="Mat.OkeiRef" />
   ```

5.2. Код документа

Создадим файл для кода документа doc_WayBill.cs со следующим содержимым:

```
namespace Docs {
 using System;
 using System.Linq;
 using Ragtime;

 public partial class WayBill : Ragtime.Security.IAny {
 partial void GetPresentation() {
 string wbType = "";
 switch(Type) {
 case WayBillType.Buy: wbType = "Покупка"; break;
 case WayBillType.Sale: wbType = "Продажа"; break;
 }
 Presentation = $"{wbType} {DocNo} от {DocDate}";
 }

 [Numerator]
 public string NumberPostfix() {
 return DocDate.Year.ToString();
 }

 partial void DoMakeNew(Guid sourceRef) {
      DocNo = "";
      DocNo_NumberRef = Guid.Empty;
    }

 partial class Calculation {
 decimal calcTotalSum() {
 return MatLines.Sum(_ => _.Summa);
 }

 }

 partial class MatLine {

 partial class Calculation {
 decimal calcSum() {
 return Math.Round(Price * Quantity,2);
 }
```

```
 }

 }

 }

 public class WayBill_Guard : Ragtime.RefObject.RefObjectGuard {
 public WayBill_Guard() : base("WayBill", "Документы", "Накладные", new Guid("AC451051-
CBA2-4F28-8723-8D6BCFAE089E")) { }
 }

}
```

Пояснения к содержимому

1. Для работы нумератора добавлен метод NumberPostfix, который помечен атрибутом Numerator

```
[Numerator]
public string NumberPostfix() {
 return DocDate.Year.ToString();
}
```

2. В нашем примере накладная, которая была создана из другой путем копирования, доблна иметь свой собственный номер. С этой целью мы очищаем поля DocNo и DocNo_NumberRef в обработчике DoMakeNew.
3. Для обслуживания вычисляемых полей, которые есть и в голове документа, и в его строках, добавлены классы WayBill.Calculation и WayBill.MatLine.Calculation соответственно.
4. В этих классах добавлены методы расчета вычисляемых полей. Важное замечание – данные, к которым имеет доступ класс Calculation это не данные объекта, а их копия, созданная для расчетов.

5.3. Форма документа

Создадим файл doc_WayBill.tsx со следующим содержимым

```
import * as AppCommand from "Ragtime.AppCommand";
import * as Form from "Ragtime.Form";
import { Field } from "Ragtime.Ui.FieldGroup";
import { FieldGroup } from "Ragtime.Ui.FieldGroup";
import { Block, BlockType } from "Ragtime.BlockForm";
import { EditGrid } from "Ragtime.Ui.EditGrid";
import { TableFieldModel } from "Ragtime.ViewModel";

import * as WayBill from "Docs.WayBill";


export class ItemForm extends WayBill.ItemForm {
 behaviorOptions: Form.BehaviorOptions = {
 preferModal: false,
 };

 protected *getBlocks(): Iterable<Block> {

 yield {
 type: BlockType.Important,
 markup: () =>
 <FieldGroup columns={2} compact={true}>
 <Field {...this.model.DocNo} columns={1} compact={true}/>
 <Field {...this.model.DocDate} columns={1} compact={true}/>
 </FieldGroup>
 };

 yield {
```

```
 type: BlockType.Main,
 caption: "Основные реквизиты",
 markup: () =>
 <FieldGroup >
 <Field {...this.model.Type} />
 <Field {...this.model.TotalSumma}/>
 </FieldGroup>
 };

 yield {
 type: BlockType.Regular,
 caption: withLineCount("Состав", this.model.MatLine),
 markup: () =>
 <EditGrid id="MatLine" {...this.model.MatLine} fullSize />
 }

 function withLineCount(caption: string, table: TableFieldModel<any>):
KnockoutObservable<string> {
 return ko.pureComputed(() => {
 let count = 0;
 if (table.value() != null)
 count = table.value().length;
 return `${caption} (${count ? count : "-"})`;
 });
 }
 }
}

export var command: AppCommand.Options = { text: "Накладные", id: "Docs.WayBill", handler:
() => new WayBill.ListForm().show() };
```

Пояснения к содержимому

Самое важное отличие от предыдущих примеров в том, что вместо метода getMainContent мы определили метод getBlocks. Это альтернативный метод генерации разметки, в котором можно генерировать разметку по частям. В нашем примере мы определили три блока:

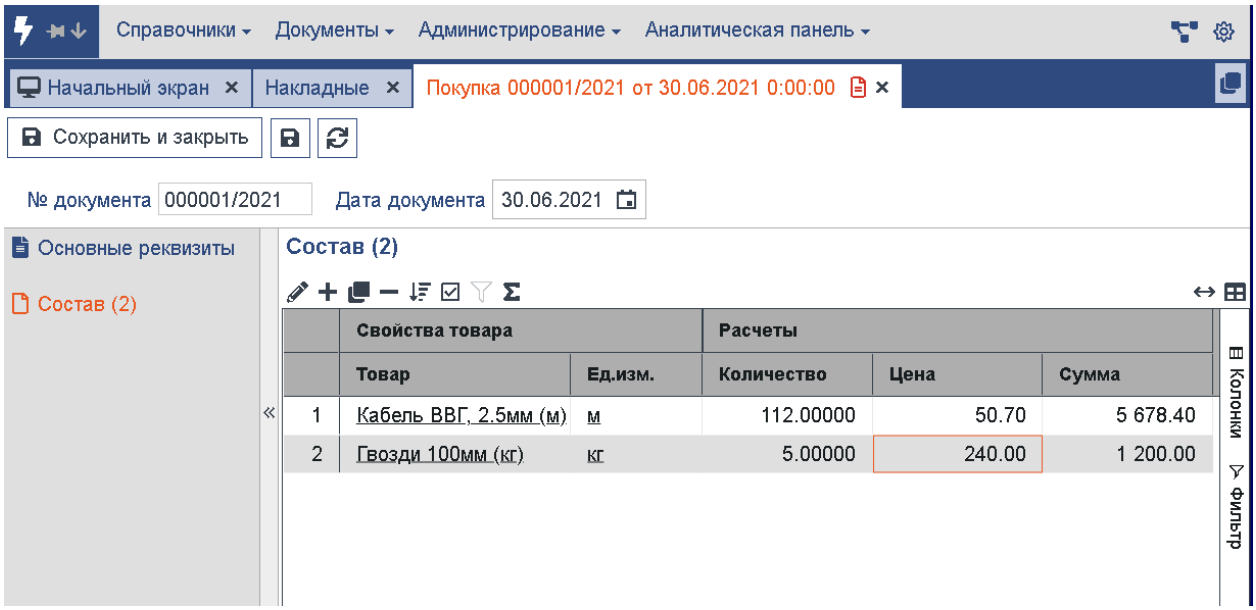
1. BlockType.Important – ототображается в строке в верхней части формы
2. BlockType.Main – первая страница формы, в нее мы помечтили (немногочисленные) атрибуты из головы документа
3. BlockType.Regular – дополнительная страница формы, на которой мы разместили табличну часть документа

Чтобы документ был доступен из главного меню, нужно добавить в него соответствующую команду:

```
.folder("Документы", f => {
  f.command("Docs.WayBill");
})
```

Теперь проект можно собрать и запустить и, как обычно, для нового объекта необходимо выдать разрешения в настройках безопасности.

Итоговый результат выглядит следующим образом:



5.4. Статусная модель

Итак, мы создали простейший документ «Накладная». Фактически, он может находиться в двух состояниях – «накладной нет» и «накладная есть». Для реализации продвинутой бизнес-логики этого явно недостаточно, поэтому объекты, созданные на базе RefObject, могут поддерживать статусную модель. Для пример предположим, что наша накладная проходит следующие этапы:

1. Черновик. Пользователь создал документ, внес в него все данные или их часть, сохранил документ и, возможно еще будет его редактировать.
2. Когда все данные введены, пользователь направляет документ в специальный отдел, где ответственный сотрудник проверяет ассортимент накладной – правильно ли указаны товарные позиции. Если все правильно, то документ переходит в состояние «Товары проверены».
3. Затем документ поступает в следующий отдел, который занимается контролем цен. Если и здесь все правильно, то документ переходит в состояние «Цены проверены», иначе – возвращается обратно.
4. После проверки цен документ поступает к руководителю, который подписывает документ, разрешая тем самым его исполнение (например, отгрузку или прием товара).
5. И, наконец, когда отгрузки или прием товара завершены, документ переходит в состояние «Исполнен».

Отразим эти этапы в виде статусной модели документа, которую добавим в метаданные накладной:

```
<RefObject …>
…
     <StateModel>
     <State Name="Draft" Value="0" Ui.Caption="Черновик" Ui.Forward="CheckMat" />
     <State Name="MatChecked" Value="10" Ui.Caption="Товары проверены" Ui.Forward="CheckPrice"
    Ui.Backward="MatCheckedToDraft" />
     <State Name="PriceChecked" Value="15" Ui.Caption="Цены проверены" Ui.Forward="Sign"
    Ui.Backward="PriceCheckedToDraft" />
     <State Name="Signed" Value="20" Ui.Caption="Подписана" Ui.Forward="Execute"
    Ui.Backward="SignedToDraft" />
     <State Name="Executed" Value="30" Ui.Caption="Исполнена" />
```

```
    <Transition Name="CheckMat" From="Draft" To="MatChecked" Ui.Caption="Проверить товары" />
    <Transition Name="CheckPrice" From="MatChecked" To="PriceChecked" Ui.Caption="Проверить
цены" />
    <Transition Name="Sign" From="PriceChecked" To="Signed" Ui.Caption="Подписать" />
    <Transition Name="Execute" From="Signed" To="Executed" Ui.Caption="Исполнить" />
    <Transition Name="MatCheckedToDraft" From="MatChecked" To="Draft" Ui.Caption="Вернуть на
черновик" />
    <Transition Name="PriceCheckedToDraft" From="PriceChecked" To="Draft" Ui.Caption="Вернуть
на черновик" />
    <Transition Name="SignedToDraft" From="Signed" To="Draft" Ui.Caption="Вернуть на
черновик" />
    <Transition Name="PriceCheckedToMatChecked" From="PriceChecked" To="MatChecked"
Ui.Caption="Вернуть на 'Товары проверены'" />
    <Transition Name="SignedToMatChecked" From="Signed" To="MatChecked" Ui.Caption="Вернуть
на 'Товары проверены'" />
    <Transition Name="SignedToPriceChecked" From="Signed" To="PriceChecked"
Ui.Caption="Вернуть на 'Цены проверены'" />
    </StateModel>

</RefObject>
```

Синтаксис статусной модели максимально прост – в ней описаны возможные статусы и переходы между ними.

Чтобы статусная модель объекта стала доступна, необходимо сделать два действия:

1. Создать и исполнить миграцию базы данных

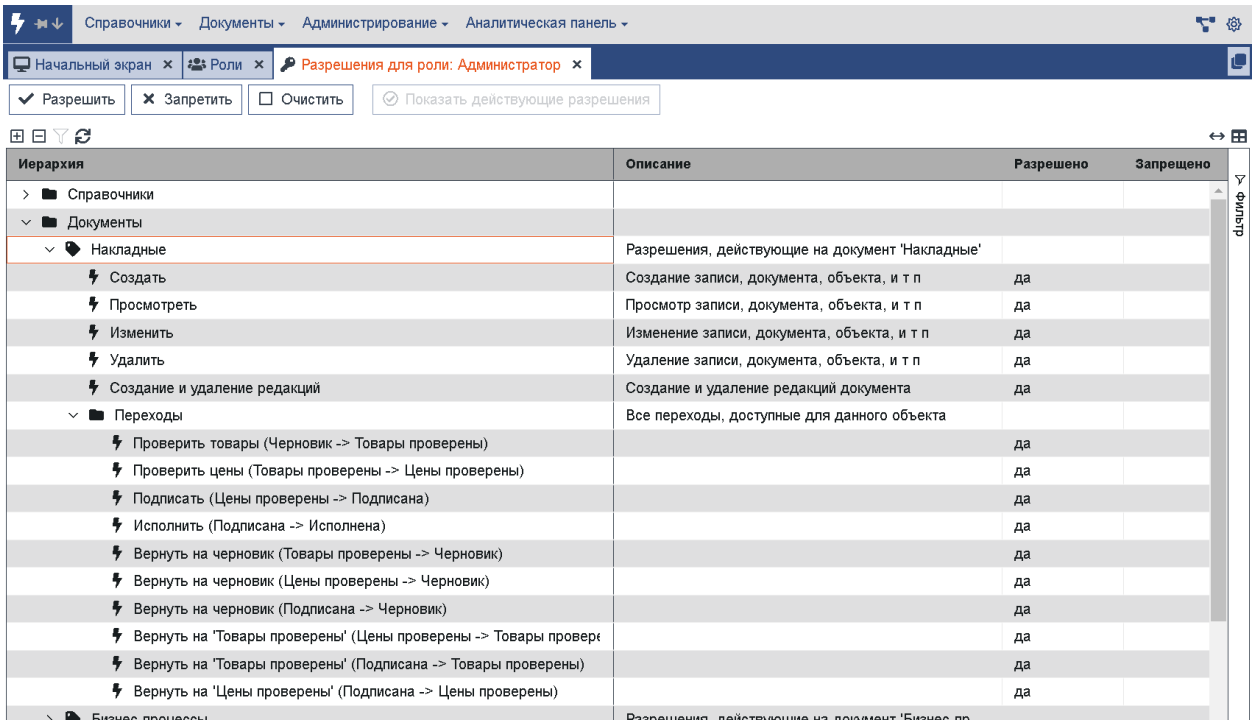
```
namespace Main_0005_Alice {
  using global::Ragtime.Data.DbUpgrade;
  using LinqToDB;
  using System.Linq;

  [Use("Docs.WayBill", "doc")]
  public partial class Transition : global::Ragtime.Data.DbUpgrade.Transition {

    public override void UpdateSchema() {
      doc.AddStateModel();
    }
  }
}
```

2. Создать объект безопасности, с помощью которого можно будет разрешить или запретить пользователям те или иные переходы между статусами.

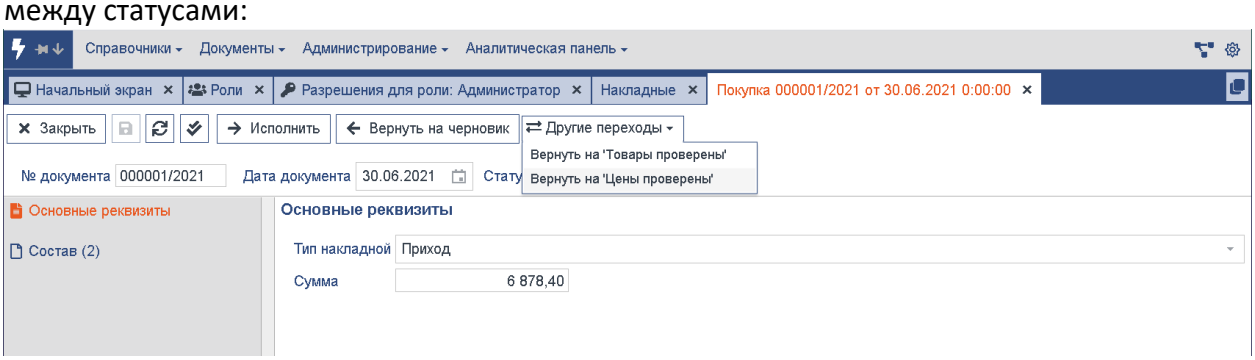
```
  public class WayBill_Guard: Docs_Guard {
    public WayBill_Guard() : base("WayBill", "Накладные", new Guid("AC451051-CBA2-4F28-
8723-8D6BCFAE089E")) { }

    protected override Ragtime.Metadata.StateModel GetStateModel() {
      return WayBill.GetStateModel();
    }
  }
```

   Базовый класс Docs_Guard объекта безопасности для документов со статусной моделью будет описан отдельно. Его назначение сгенерировать дополнительные элементы в модель безопасности на основе переходов статусной модели документа.

Теперь проект можно собрать и запустить. Обратите внимание, в настройках безопасности появилась возможность дать разрешение или запретить любой из переходов между статусами документа. В тестовых целях выдадим текущему пользователю полные права:



Теперь в форме просмотра накладной добавились дополнительные команды – переходы между статусами:



5.5. Дополнительные возможности объекта RefObject

Объекты, созданные на основе RefObject, поддерживают несколько важных механизмов, для использования которых прикладному программисту требуется выполнить минимальное количество действий:

5.5.1. Протоколирование истории изменений

Для подключения к объекту истории достаточно добавить в метаданные объекта узел
`<TrackHistory>true</TrackHistory>`
и добавить соответствующую миграцию базы данных (см. ниже)

5.5.2. Использование вложений

К объекту можно прикреплять вложения. В нашем примере это могут быть отсканированные копии бумажных документов поставщика. Для подключения этой функции в метаданные объекта необходимо добавить узел
`<UseAttachments>true</UseAttachments>`

5.5.3. Редакции

Редакции – это механизм хранения версий документов. Если, по каким-либо причинам, в документ требуется внести изменения, то механизм редакций позволяет создать новую

версию документа, внести в нее необходимые изменения и сохранить. При этом старая версия документа никуда не исчезает и ее в любой момент можно посмотреть. При необходимости новую версию можно удалить, при этом произойдет возврат к предыдущей версии документа. Количество редакций документа не ограничено.
Для подключения этой функции в метаданные объекта необходимо добавить узел
`<UseEditions>true</UseEditions>`

и добавить соответствующую миграцию базы данных (см. ниже). Кроме того, в метаданных документа необходимо указать, на каком статусе можно создавать новые редакции. Если разрешение на создание новой редакции у статуса не указано, то считается, что новую редакцию на этом статусе создавать нельзя.

5.5.4. Обсуждения

Обсуждения – это текстовые сообщения, которые можно оставлять с привязкой к каждому конкретному объекту на базе RefObject. В нашем примере, например, это может быть обсуждение, связанное с согласованием цен или списка товарных позиций.
Обсуждения подключаются добавлением в метаданные объекта узла
`<UseDiscussion>true</UseDiscussion>`

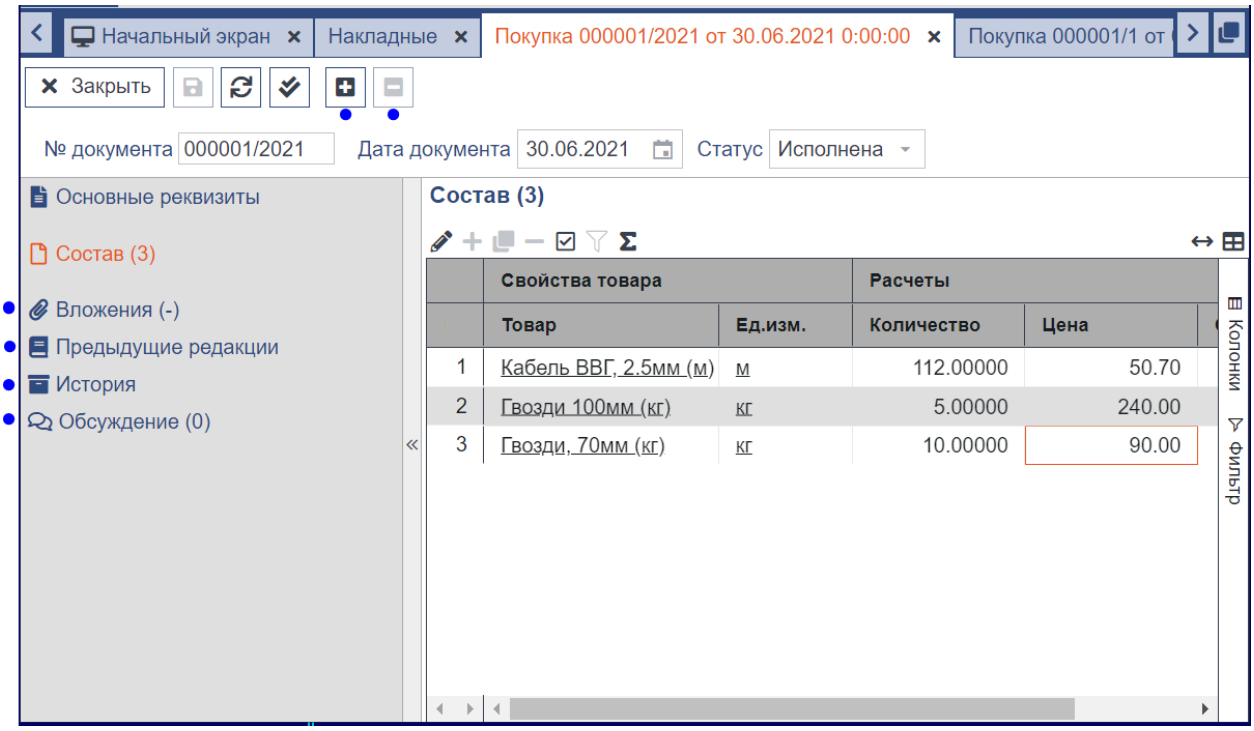
5.6. Миграция базы для поддержки дополнительных возможностей

Некоторые из возможностей, описанных выше – история и редакции – требуют внесения изменений в баз данных. Миграция для них будет выглядеть как:

```
namespace Main_0006_Alice {
  using global::Ragtime.Data.DbUpgrade;
  using LinqToDB;
  using System.Linq;

  [Use("Docs.WayBill", "doc")]
  public partial class Transition : global::Ragtime.Data.DbUpgrade.Transition {

    public override void UpdateSchema() {
      doc.AddTrackHistory();
      doc.AddEditions();
    }
  }
}
```

5.7. Тестирование изменений

После всех внесенных изменений соберем и запустим проект. Откроем форму накладной, в ней мы увидим все добавленные нам новые механизмы (помечены синими точками):



6. Регистры

Регистры Ragtime - это механизм, позволяющий без особых усилий организовать аггрегирование данных в желаемых разрезах. В качестве примера добавим в проект регистр «Обороты материалов». Документы «Накладная» будут делать записи в этот регистр, позволяя, тем самым, отслеживать итоговое количество товаров, а также вычислять доход от торговли.

6.1. Добавление регистра

Добавим в проект файл MatBalance.metadata с содержимым:

```
<?xml version="1.0" encoding="utf-8" ?>
<AccReg
 Id="7dde82fb-0a80-490c-b1ac-11f4c3d5c6bc"
 Type="BalanceAndTurnover"
 Namespace="Regs"
 Name="MatBalanceReg">

 <Doc>Обороты товаров</Doc>
 <Security Disabled="true" />
 <Ui Caption="Обороты товаров (регистр)" />

 <Field Name="Period" Rules.Granularity="Day" />

 <Field Name="MatRef" Ui.Caption="Товар" RefTo="RefBooks.ref_cmn_Mat" Rules.IfEmpty="Error"
Default="true" />
 <Field Name="OkeiRef" Ui.Caption="Ед.изм." Expression="Mat.OkeiRef" />

 <Field Name="Quantity" Ui.Caption="Количество" Type="decimal" Storage.Type="decimal"
Storage.Param="18" Storage.Param2="5" Ui.Format="fixedPoint" Ui.Precision="5" Default="true"/>
 <Field Name="Summa" Ui.Caption="Сумма" Type="decimal" Storage.Type="Money"
Ui.Format="fixedPoint" Ui.Precision="2" Storage.Stored="true" Default="true"/>


 <Dimensions Fields="MatRef" />
 <Measures Fields="Summa,Quantity"/>

 <Index Name="I0" Fields="MatRef" />

</AccReg>
```

Комментарии к содержимому:

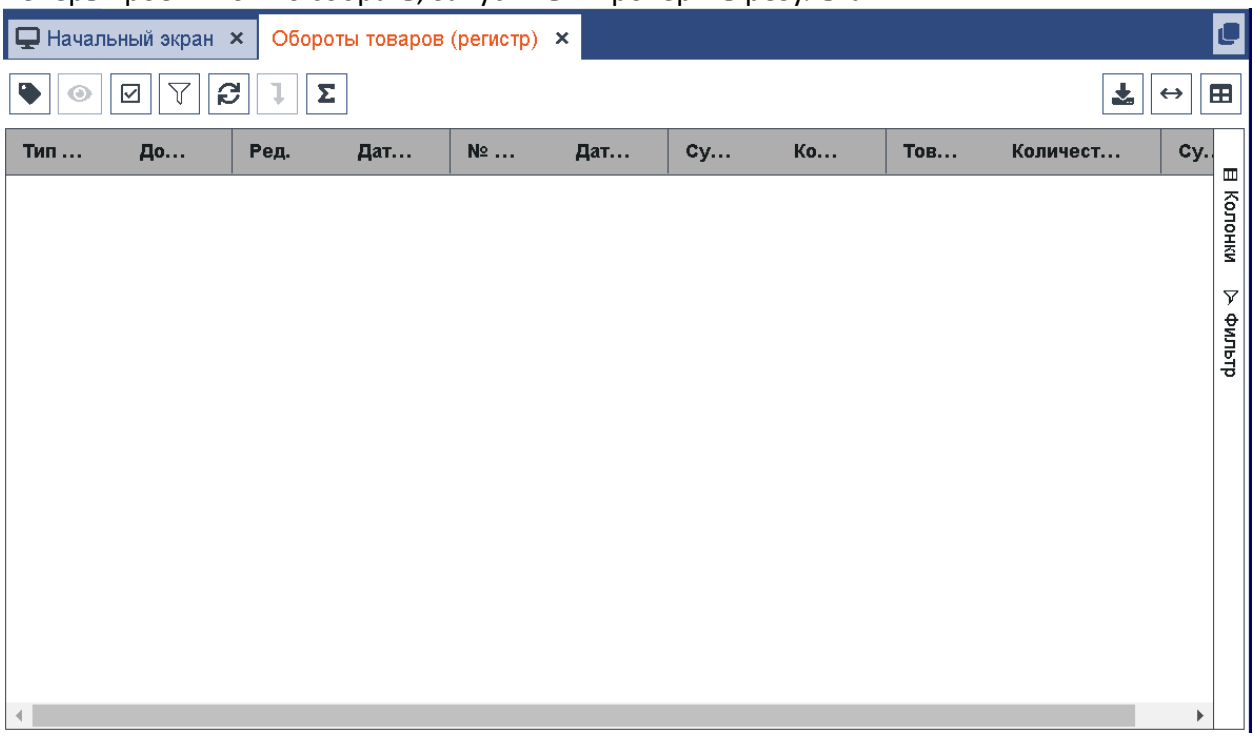
- Регистр Ragtime это объект с типом AccReg. В данном примере тип регистра Type="BalanceAndTurnover" – регистр оборотов.
- Period - предопределенное поле с типом «Дата», его атрибут Rules.Granularity определяет, в каком разрезе по времени будут подсчитаны итоги по регистру.
- Dimensions – список полей, в разрезе которых ведется подсчет итогов.
- Measures - список полей, по которым ведется подсчет итогов.

6.2. Просмотр регистра

Добавим команду просмотра регистра, файл MatBalance.tsx с содержимым:

```
import * as AppCommand from "Ragtime.AppCommand";
import * as MatBalance from "Regs.MatBalanceReg";


export var command: AppCommand.Options = {
 text: "Обороты материалов (регистр)",
```

```
 id: "Regs.MatBalance",
 handler: () => new MatBalance.ListForm().show(),
 label: { },
};
```
а также команду в главное меню
```
 .folder("Регистры", f => {
 f.command("Regs.MatBalance");
 })
```
Последнее необходимое действие – миграция базы данных:
```
namespace Main_0007_Alice {
 using global::Ragtime.Data.DbUpgrade;
 using LinqToDB;
 using System.Linq;

 [Use("Regs.MatBalanceReg", "reg")]
 public partial class Transition : global::Ragtime.Data.DbUpgrade.Transition {
 public override void UpdateSchema() {
 reg.Create();
 }
 }
}
```
Теперь проект можно собрать, запустить и проверить результат:



6.3. Заполнение регистра

На предыдущем шаге мы добавили в документ «Накладная» статусную модель. Сделаем так, чтобы при переходе на статус «Исполнена» накладная делала записи в регистр «Обороты материалов». Приходная накладная будет увеличивать остатки товара и уменьшать остаток денежных средств, расходная накладная – наоборот, будет уменьшать остатки товара и увеличивать остатки денежных средств.

Прежде, чем заполнять регистр, расширим статусную модель накладной и добавим в нее переход, обратный переходу Executed. Это нужно для демонстрации поведения регистра при переходах между статусами и между редакциями.

```
<State Name="Executed" Value="30" Ui.Caption="Исполнена" Ui.Backward="Unexecute"
NewEditionAllowed="true" />
```

```
<Transition Name="Unexecute" From="Executed" To="Signed" Ui.Caption="Отменить" />
```

Теперь добавим в накладную обработчики переходов, которые будут сохранять записи в регистр:

```
    partial void DoMotionForExecuteTransition(IMotions motions, string comment) {
      var recs = motions.MatBalanceReg.Records = new List<Regs.MatBalanceReg>();
      foreach(var line in MatLines) {
        recs.Add(new Regs.MatBalanceReg() {
          Period = DocDate,
          ItemType = Type == WayBillType.Buy ? Ragtime.AccReg.ItemType.Receipt :
Ragtime.AccReg.ItemType.Expence,
          Mat = line.Mat,
          Quantity = line.Quantity,
          Summa = -line.Summa,
        });
      }
    }

    partial void DoMotionForUnexecuteTransition(IMotions motions, string comment) {
      motions.MatBalanceReg.Delete();
    }
```

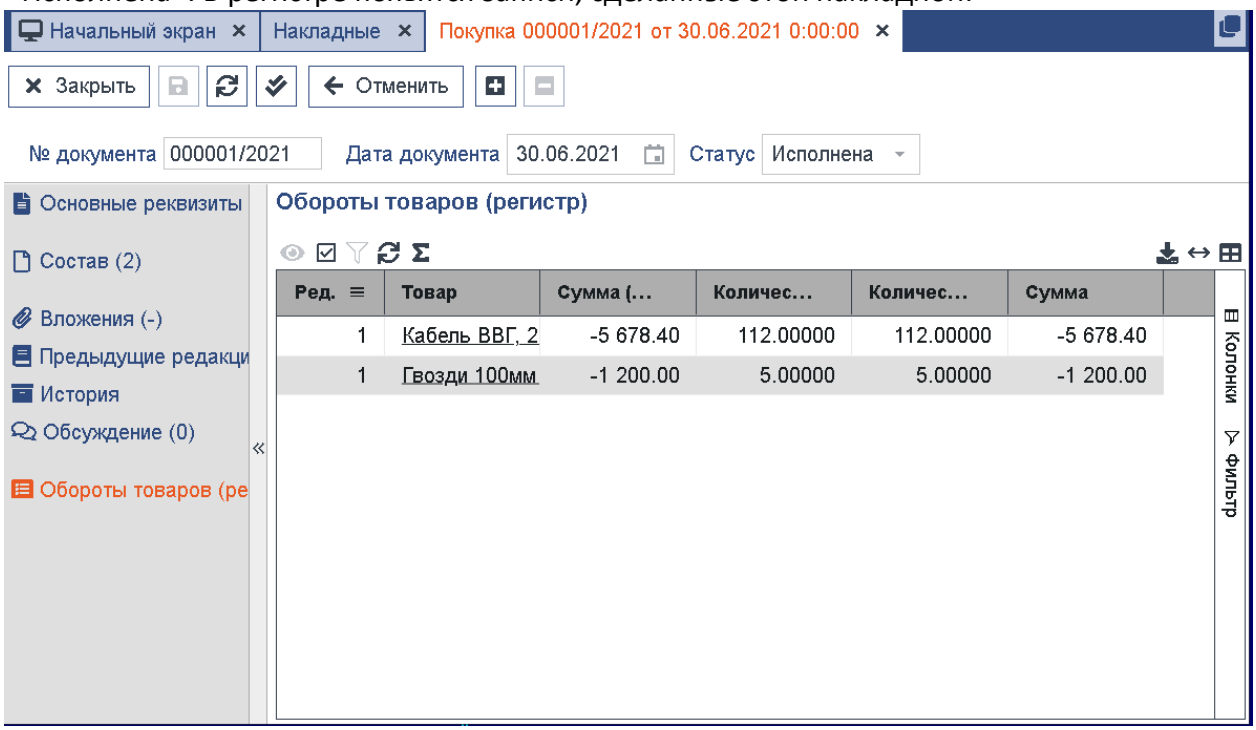
Имя обработчика имеет вид DoMotionFor<ИмяПерехода>Transition. Для двух переходов – Execute и Unexecute мы реализовали два обработчика DoMotionForExecuteTransition и DoMotionForUnexecuteTransition. Обработчик обратного перехода в нашем случае максимально прост – он удаляет все движение, которое было создано при прямом переходе.

Обработчик прямого перехода сохраняет в регистр записи по каждой строке накладной.

И, наконец, необходимо указать, что документ накладная подключена к регистру. Для этого в метаданные накладной добавляем строку:

<Motion Type="Regs.MatBalanceReg" />

Проверим работу обработчиков. Для этого создадим накладную и доведем ее до статуса «Исполнена». В регистре появятся записи, сделанные этой накладной:



Если выполнить переход «Отменить», то эти записи исчезнут.

6.4. Регистр и редакции

Важным моментом является то, что при записи движения мы не заботимся о том, какая именно по счету у нас редакция документа, и какие записи были сделаны предыдущими редакциями. Мы сохраняем движение ровно в том виде, в котором оно должно быть в итоге, и механизм регистров сам определяет, какие для этого нужно добавить проводки. Для иллюстрации создадим новую редакцию накладной и внесем в нее изменения – добавим еще одну строку товаров, а по одной из строк изменим цену. После этого доведем вторую редакцию до статуса «Исполнена».

Состав накладной первой редакции:

	Свойства товара		Расчеты		
	Товар	Ед.изм.	Количество	Цена	Сумма
1	Кабель ВВГ, 2.5мм (м)	м	112.00000	50.70	5 678.40
2	Гвозди 100мм (кг)	кг	5.00000	240.00	1 200.00

Состав накладной второй редакции:

	Свойства товара		Расчеты		
	Товар	Ед.изм.	Количество	Цена	Сумма
1	Кабель ВВГ, 2.5мм (м)	м	112.00000	50.70	5 678.40
2	Гвозди 100мм (кг)	кг	5.00000	260.00	1 300.00
3	Гвозди, 70мм (кг)	кг	7.00000	195.00	1 365.00

Регистр:

Обороты товаров (регистр)

Ред.	Товар	Сумма (...	Количес...	Количес...	Сумма
1	Кабель ВВГ, 2	-5 678.40	112.00000	112.00000	-5 678.40
1	Гвозди 100мм	-1 200.00	5.00000	5.00000	-1 200.00
2	Гвозди 100мм	-100.00	0.00000	0.00000	-100.00
2	Гвозди, 70мм	-1 365.00	7.00000	7.00000	-1 365.00

Как мы видим, вторая редакция добавила дельту по второй товарной позиции и новую строку для третьей товарной позиции, которой не было в первой версии накладной.

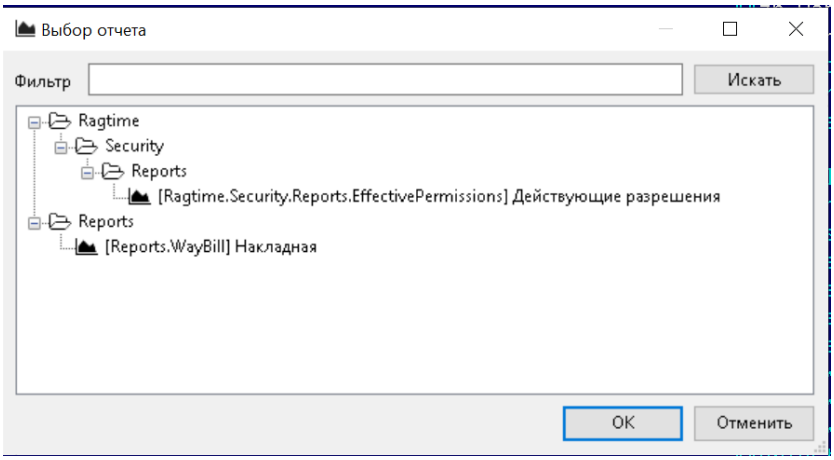
7. Создание отчетов

Отчеты и печатные формы – обязательная часть любого бизнес-приложения. Для создания отчетов в приложении на базе Ragtime, в первую очередь, необходимо подготовить и запустить сам дизайнер отчетов. Для этого нужно добавить новый проект с кодом:

```
namespace ReportDesigner {
 using System;
 using System.Reflection;
 using System.IO;
 using System.Windows.Forms;
 using WF = System.Windows.Forms;

 static class Program {
 [STAThread]
 static void Main() {
 WF.Application.ThreadException += (s, e) => MessageBox.Show(e.Exception.Message, "Ошибка",
MessageBoxButtons.OK, MessageBoxIcon.Error);
 WF.Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchException);
 WF.Application.EnableVisualStyles();

 Ragtime.ReportDesigner.Application.ShowLogo();
 try {
 var dir = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
 Ragtime.DesktopApplication.Application.Initialize(new Ragtime.DesktopApplication.Settings {
 ConfigBasePath = Path.GetFullPath(Path.Combine(dir, "../../MyFirstApplication.Web"))
 });
 }
 finally {
 Ragtime.ReportDesigner.Application.HideLogo();
 }
 var report = Ragtime.ReportDesigner.Application.SelectReport();
 if(report != null)
 Ragtime.ReportDesigner.Application.Run(report);
 }
 }
}
```

Пояснения к содержимому

1. Дизайнер отчетов – локальное приложение UI на базе Windows.Forms.
2. Блок try{} читает настройки основного приложения MyFirstApplication.Web.
3. Функция Ragtime.ReportDesigner.Application.SelectReport просматривает все отчеты в проекте и отображает их список.
4. Ragtime.ReportDesigner.Application.Run отображает редактор для выбранного отчета.



В список зависимостей этого проекта нужно добавить все проекты, которые будут содержать отчеты. В нашем случае это один проект, MyFirstApplication.Main

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">
 <Import Project="../Product.props" />
 <PropertyGroup>
 <OutputType>WinExe</OutputType>
 <Description>Редактор отчетов</Description>
 <EnableDefaultNoneItems>false</EnableDefaultNoneItems>
 <UseWindowsForms>true</UseWindowsForms>
 </PropertyGroup>

 <ItemGroup>
 <PackageReference Include="Ragtime.DesktopApplication" Version="$(RagtimeVersion)" />
 <PackageReference Include="Ragtime.ReportDesigner" Version="$(RagtimeVersion)" />
 </ItemGroup>

 <ItemGroup>
 <ProjectReference Include="..\MyFirstApplication.Main\MyFirstApplication.Main.csproj" />
 </ItemGroup>
</Project>
```

Кроме того, проект MyFirstApplication.ReportDesigner необходимо добавить в решение, чтобы обеспечить его сборку вместе с остальными проектами.

7.1. Добавление отчета

Создадим для отчета файл метаданных WayBill.metadata с содержимым

```
<Report Id="BBEF9AD8-376D-49C7-B5B5-4297D4488186" Namespace="Reports" Name="WayBill">

 <Ui Caption="Накладная" Group="Документы" />

 <Parameter Name="WayBillRef" RefTo="Docs.WayBill" Ui.Caption="Накладная" Required="true" />

 <DataClass Name="WayBillHead" Doc="Шапка накладной">

 <Field Name="DocNo" Type="string" Ui.Caption="№ документа" />
 <Field Name="DocDate" Ui.Caption="Дата документа" Type="DateTime" />
 <Field Name="Type" Ui.Caption="Тип накладной" Type="Docs.WayBillType" />
 <Field Name="TotalSumma" Ui.Caption="Сумма" Type="decimal" />
 <Field Name="Presentation" Type="string"/>
 <Field Name="WayBillRows" Type="WayBillContent[]" Ui.Caption="Строки накладной" />

 <DataClass Name="WayBillContent" Doc="Строка накладной">
 <Field Name="MatRef" Ui.Caption="Товар" RefTo="RefBooks.ref_cmn_Mat" />
 <Field Name="OkeiRef" Ui.Caption="Ед.изм." Expression="Mat.OkeiRef" />
 <Field Name="Quantity" Ui.Caption="Количество" Type="decimal" />
 <Field Name="Price" Ui.Caption="Цена" Type="decimal" />
 <Field Name="Summa" Ui.Caption="Сумма" Type="decimal" />
 </DataClass>

 </DataClass>

</Report>
```

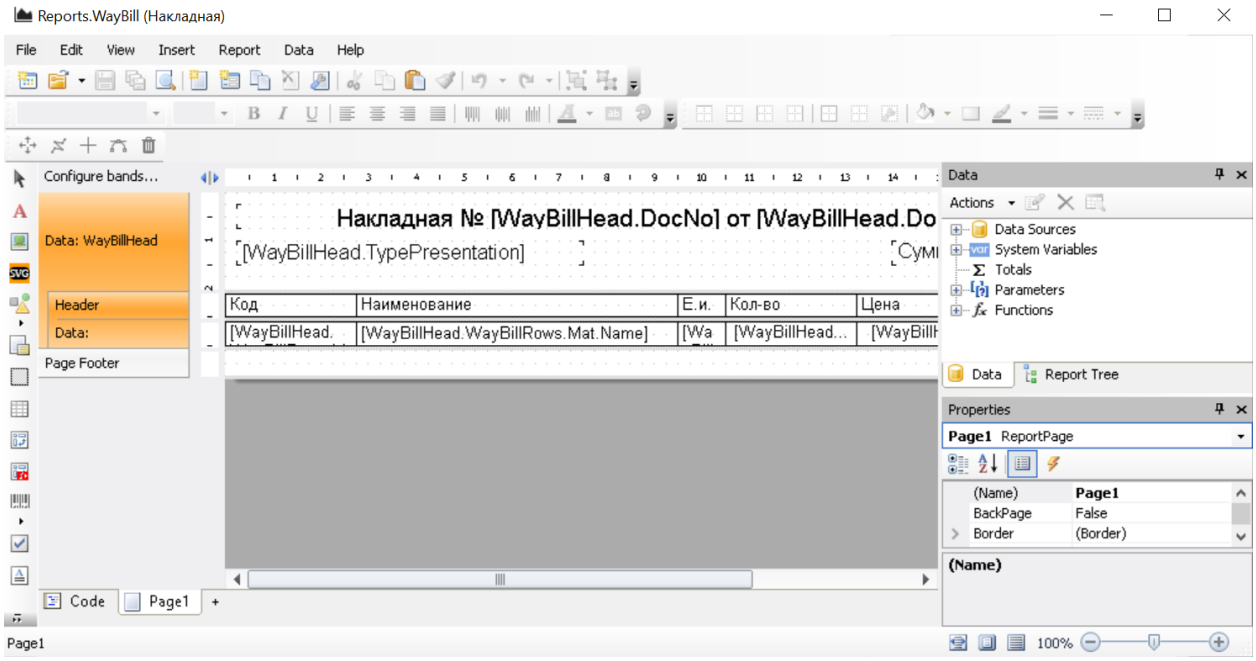
Пояснения к содержимому

1. Для определения отчета мы создали элемент Report. Его состав аналогичен составу RefObject, основные отличия связаны с тем, что Report не хранится в базе данных, а каждый раз генерируется заново.
2. Report может содержать узлы Parameter. Эти узлы описываю параметры, которые будут отображены при исполнении отчета.
3. Данные отчета описываются узлами DataClass. В данном примере мы описали головной узел WayBillHead, который содержит дочерние подузлы WayBillContent.

4. Поскольку данный отчет будет просто отображать содержимое накладной, то его поля мы скопировали один-в-один, с той лишь разницей, что удалили из них атрибуты проверок и хранения – для отчета эти атрибуты не имеют смысла.

7.2. Редактирование форм отчета

Для создания или редактирования печатной формы необходимо собрать решение и запустить редактор отчетов MyFirstApplication.ReportDesigner.exe. Редактор обнаружит созданный нами отчет и даст возможность изменить его:



Включим в отчет шапку и строки накладной и сохраним его.

7.3. Заполнение данных отчета

Для заполнения отчета в момент исполнения реализуем его метод DoPrepare:

```
#pragma warning disable 1591
#pragma warning disable 1573
namespace Reports {
 using System;
 using System.Linq;
 using Ragtime;
 using Ragtime.Data;
 using LinqToDB;
 using System.Collections.Generic;
 using FastReport;

 partial class WayBill {

 partial void DoPrepare(FastReport.Report report) {

 var wb = GetParameters().WayBill;

 report.ReportInfo.Name = wb.Presentation;

 var records = new List<WayBillHead>() {
 new WayBillHead(){
```

```
 DocDate = wb.DocDate,
 DocNo = wb.DocNo,
 Presentation = wb.Presentation,
 TotalSumma = wb.TotalSumma,
 Type = wb.Type,
 WayBillRows = wb.MatLines.Select(_ => new WayBillHead.WayBillContent(){
 Mat = _.Mat,
 Okei = _.Okei,
 Price = _.Price,
 Quantity = _.Quantity,
 Summa = _.Summa,
 }).ToArray(),
 }
 };
 SetData_WayBillHead(records, null);
 }
 }
}
```

Пояснения к содержимому

1. Параметры исполнения отчета, заполненные пользователем, доступны нам через метод GetParameters. В данном примере мы указали, что в качестве параметра должна быть выбрана накладная – ее мы и используем для заполнения данных отчета.
2. Само заполнение сводится к тому, что мы создаем List<WayBillHead> и добавляем к нему один элемент (мы печатаем одну накладную). После этого мы вызываем метод SetData_WayBillHead. На этом заполнение данных отчета завершено.

7.4. Проверка отчета

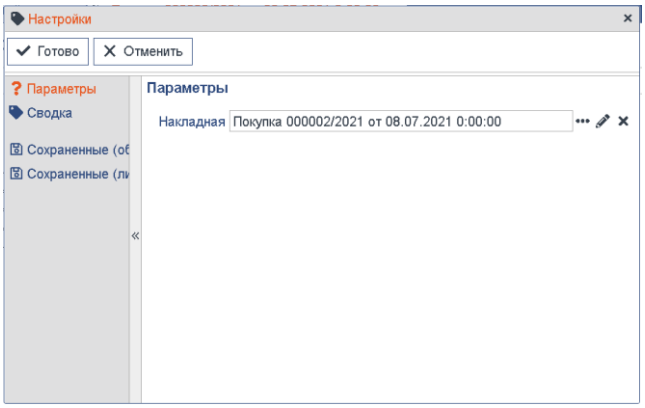
Чтобы проверить работу отчета мы добавили создали для него отдельную команду и добавили его в главное меню:

```
import * as WayBill from "Reports.WayBill";
import * as AppCommand from "Ragtime.AppCommand";

export var command: AppCommand.Options = {
 text: "Накладная",
 id: "Reports.WayBill",
 handler: () => {
 new WayBill.Report().show();
 },
 label: {},
};
```

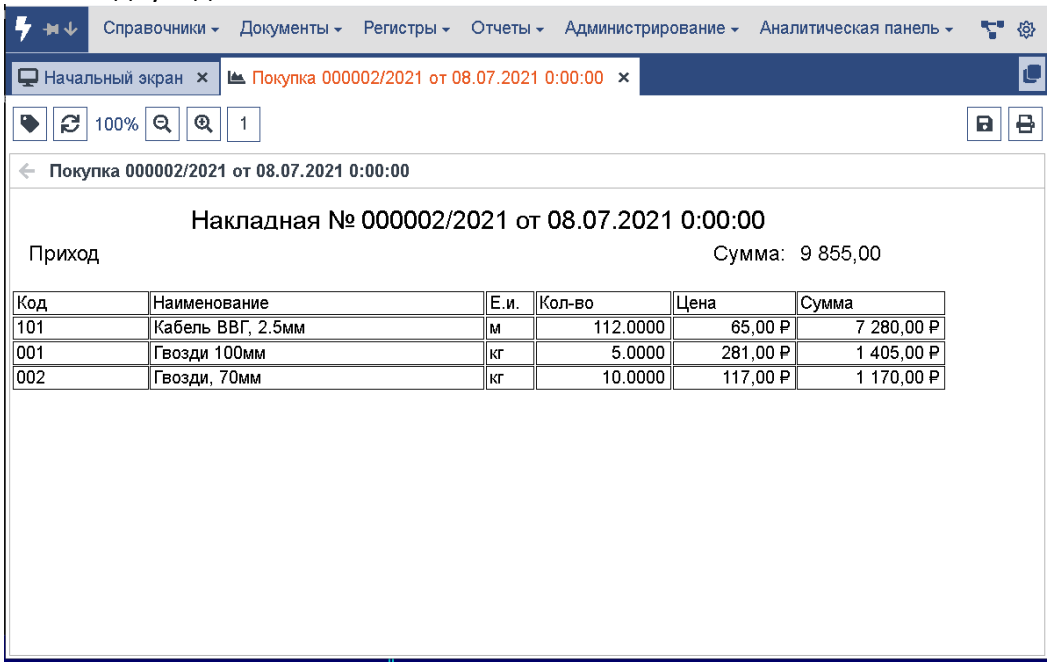
Новый пункт в главном меню:

```
.folder("Отчеты", f => {
 f.command("Reports.WayBill");
 })
```

Исполним отчет. Первое, что будет отображено на экране это форма выбора параметров отчета:



Выберем накладную для печати и исполним отчет:



Этот отчет можно распечатать, сохранить в виде файлов xlsx, docx, mht или pdf.
При повторном исполнении отчета окно выбора параметров не показывается. Для изменения параметров отчета необходимо воспользоваться командой «Настройки».

7.5. Альтернативные способы исполнения отчета

В примере выше отчет представлял собой печатную форму накладной. Естественно печатать его непосредственно из самой накладной. Реализуем эту печать, добавив в форму просмотра накладной следующий код:

```
import * as Dialog from "Ragtime.Dialog";
import * as WayBillReport from "Reports.WayBill";
import { ToolbarSettings } from "Ragtime.Ui.TableViewer";
import { Command } from "Ragtime.Command";
```

```
…
 setupToolbar(t: ToolbarSettings) {
 super.setupToolbar(t);
 t.addSection().addCommand(this.cmdPrint);
 }

 cmdPrint = new Command(this, {
 icon: "fa fa-print",
 hint: "Печать",
 handler: () => this.doPrint(),
 });

 async doPrint(): Promise<void> {
 if (this.modified()) {
 if (!(await Dialog.askConfirmation("Документ не сохранен. Сохранить?"))) return;
 await this.write();
 }

 var params = { WayBillRef: this.model.Ref.value(), WayBillPresentation:
this.model.Presentation.value(), };
 var r = new WayBillReport.Report();
 r.show(params);
 }
```

Пояснения к содержимому

1. Для несохраненного документа мы отображаем пользователю подтверждение на сохранение документа, для этого мы используем методы модуля Ragtime.Dialog.
2. Модуль Reports.WayBill необходимо импортировать для получения доступа к методам отчета по накладной.
3. Метод setupToolbar позволяет нам изменить панель элементов управления. В данном случае мы добавляем в него новую команду печати
4. При исполнении отчета мы используем заранее подготовленные параметры. Поэтому запрос параметров у пользователя не производится и отчет выполняется для текущей накладной.

7.6. Отчет по регистру

Регистры оборотов позволяют собирать данные в различных разрезах. Рассмотрим вывод этих данных на печать. Для этого создадим отчет на базе регистра «Обороты товаров». Метаданные отчета:

```
<Report Id="0A4F623F-E73A-47C6-AA27-BDDE2E75D67A" Namespace="Reports" Name="MatRests">

 <Ui Caption="Обороты товаров" Group="Документы" />

 <Parameter Name="FromDate" Type="DateTime" Ui.Caption="С" Required="true" />
 <Parameter Name="ToDate" Type="DateTime" Ui.Caption="По" Required="true" />

 <DataClass Name="MatRestLine" Doc="Строка отчета">

 <Field Name="MatRef" Ui.Caption="Товар" RefTo="RefBooks.ref_cmn_Mat" />
 <Field Name="StartQuantity" Ui.Caption="На начало" Type="decimal" />
 <Field Name="QuantityIn" Ui.Caption="Приход" Type="decimal" />
 <Field Name="QuantityOut" Ui.Caption="Расход" Type="decimal" />
 <Field Name="EndQuantity" Ui.Caption="На конец" Type="decimal" />

 </DataClass>

</Report>
```

Для заполнения отчета будем выполнять следующие операции:

1. Соберем из регистра остатки на начало, остатки на конец и обороты.
2. Соберем общий список строк отчета, для каждого товара – одна строка отчета.
3. Дозаполним остальные поля строк отчета.

Код, который выполняет эти действия:

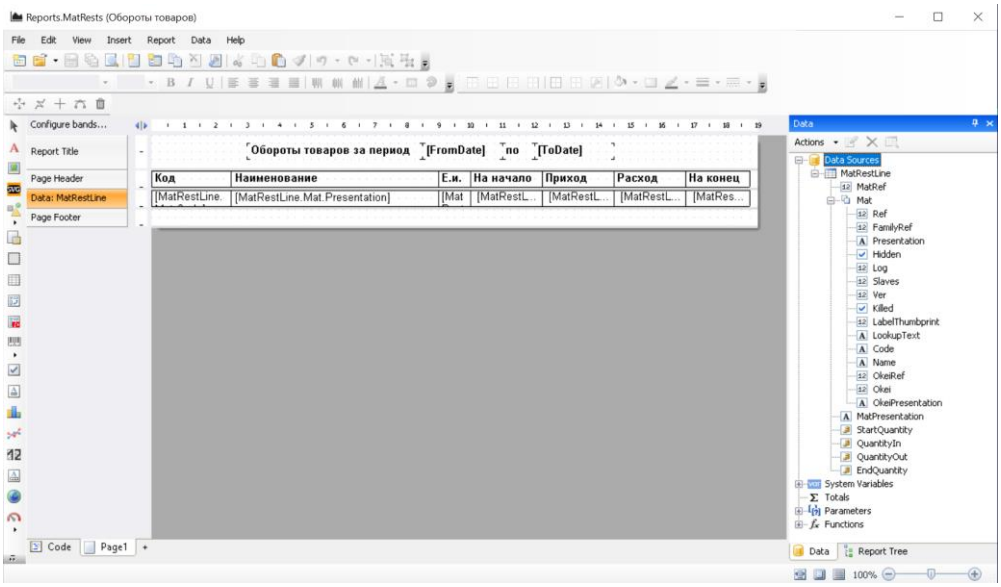
```
partial class MatRests {


partial void DoPrepare(FastReport.Report report) {

var fromDate = GetParameters().FromDate;
var toDate = GetParameters().ToDate;

report.ReportInfo.Name = $"Обороты товаров с {fromDate} по {toDate}";


Regs.MatBalanceReg.Balance b;

b = new Regs.MatBalanceReg.Balance();
b.AtStart(fromDate);
var startRests = b.Run().ToList();

b = new Regs.MatBalanceReg.Balance();
b.AtEnd(toDate);
var endRests = b.Run().ToList();

var t = new Regs.MatBalanceReg.Turnover();
t.From(fromDate);
t.ToEnd(toDate);
var turns = t.Run().ToList();

var rows = turns.Select(_ => _.MatRef)
.Union(startRests.Select(_ => _.MatRef))
.Union(endRests.Select(_ => _.MatRef))
.ToDictionary(_ => _, _ => new MatRestLine() { MatRef = _, });

startRests.ForEach(_ => rows[_.MatRef].StartQuantity = _.Quantity);
endRests.ForEach(_ => rows[_.MatRef].EndQuantity = _.Quantity);
turns.ForEach(_ => {
rows[_.MatRef].QuantityIn = _.QuantityReceipt;
rows[_.MatRef].QuantityOut = _.QuantityExpence;
});

SetData_MatRestLine(rows.Values, null);
}
}
```

Пояснения к содержимому

1. Класс Balance регистра обеспечивает доступ к остаткам. Остатки можно брать на начало или на конец периода. В нашем примере в качестве периода выбран день, т.е. мы можем получить остатки на начало либо на конец дня.
2. Класс Turnover регистра обеспечивает доступ к оборотам.

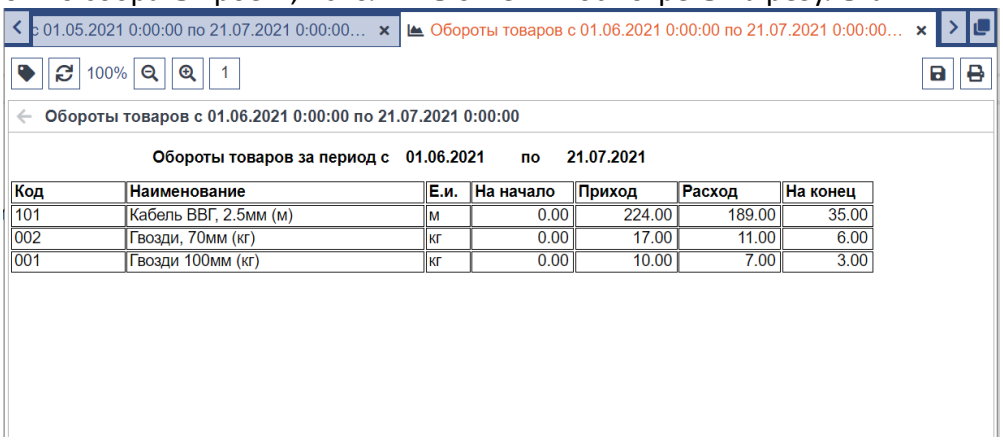
7.7. Проверка отчета

Создайте печатную форму для этого отчета, например такую:



Обратите внимание, в метаданных отчета мы указали только поле Mat, ссылку на элемент справочника товаров. Ragtime самостоятельно добавил в отчет все свойства этого элемента.

Теперь можно собрать проект, исполнить отчет и посмотреть на результат:



7.8. Второй отчет – прибыли и убытки

Наше приложение представляет собой максимально упрощенный вариант учета торговли. Естественно было бы получить информацию о том, какие у нас доходы и расходы и какая, в итоге, прибыль. Для получения такого отчета за период мы будем использовать данные регистра «Обороты товаров», но этих данных нам недостаточно – нужно еще как-то учесть стоимость товаров, которые находятся на складе в остатках. Для этого поступим максимально упрощенным способом – добавим товару свойство «Учетная цена» и будем использовать ее для определения суммы остатков на начало и на конец периода. Это даст нам возможность рассчитать прибыль по формуле:

Прибыль = «Сумма на конец периода» - «Сумма на начало периода» + «Сумма продаж» - «Сумма покупок»

Прибыль будем считать как в целом, так и по каждому товару в отдельности.

7.9. Добавление учетной цены

Добавим поле в метаданные справочника товаров:

```
<RefObject …
 Name="ref_cmn_Mat"
>
…
 <Field Name="Price" Ui.Caption="Учетная цена" Type="decimal" Storage.Type="Money"
Ui.Format="fixedPoint" Ui.Precision="2" />
…
</RefObject>
```

А также добавим миграцию для создания этого поля в базе данных:

```
namespace Main_0008_Alice {
 using global::Ragtime.Data.DbUpgrade;
 using LinqToDB;
 using System.Linq;

 [Use("RefBooks.ref_cmn_Mat", "refbook")]
 public partial class Transition : global::Ragtime.Data.DbUpgrade.Transition {
 public override void UpdateSchema() {
 refbook.AddField("Price");
 }
 }
}
```

Чтобы пользователь мог заполнить значение – добавляем поле в форму просмотра товара:

```
getMainContent() {
 return (
 <FieldGroup>
 <Field {...this.model.Code} />
 <Field {...this.model.Name} />
 <Field {...this.model.Okei} />
 <Field {...this.model.Price} />
 </FieldGroup>
 );
}
```

7.10. Создание отчета

Следующий шаг – создание собственно отчета. Создадим метаданные отчета:

```
<Report Id="5EB64151-61F5-4F95-9689-34690C312DF6" Namespace="Reports" Name="MatBal">

       <Ui Caption="Баланс" Group="Документы" />

       <Parameter Name="FromDate" Type="DateTime" Ui.Caption="С" Required="true" />
       <Parameter Name="ToDate" Type="DateTime" Ui.Caption="По" Required="true" />

       <DataClass Name="BalLine" Doc="Строка отчета">
              <Field Name="Code" Ui.Caption="Код товара" Expression="Mat.Code" />
              <Field Name="MatRef" Ui.Caption="Наименование товара"
RefTo="RefBooks.ref_cmn_Mat" />
              <Field Name="OkeiRef" Ui.Caption="Ед.изм." Expression="Mat.OkeiRef" />
              <Field Name="Price" Ui.Caption="Учетная цена" Expression="Mat.Price" />
              <Field Name="StartQuantity" Ui.Caption="Кол-во на начало" Type="decimal" />
              <Field Name="SummaOut" Ui.Caption="Расходы" Type="decimal" />
              <Field Name="SummaIn" Ui.Caption="Доходы" Type="decimal" />
              <Field Name="EndQuantity" Ui.Caption="Кол-во на конец" Type="decimal" />
</DataClass>

</Report>
```

И модуль для заполнения этого отчета данными:

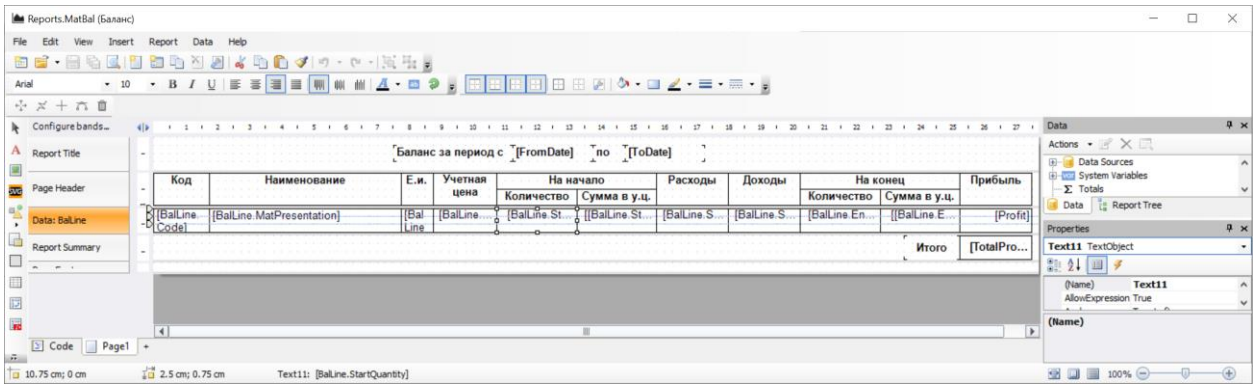
```
#pragma warning disable 1591
#pragma warning disable 1573
namespace Reports {
  using System;
  using System.Linq;
  using Ragtime;
  using Ragtime.Data;
```

```
  using LinqToDB;
  using System.Collections.Generic;
  using FastReport;

  partial class MatBal {

    partial void DoPrepare(FastReport.Report report) {

      var fromDate = GetParameters().FromDate;
      var toDate = GetParameters().ToDate;

      report.ReportInfo.Name = $"Баланс за период с {fromDate} по {toDate}";


      Regs.MatBalanceReg.Balance b;

      b = new Regs.MatBalanceReg.Balance();
      b.AtStart(fromDate);
      var startRests = b.Run().ToList();

      b = new Regs.MatBalanceReg.Balance();
      b.AtEnd(toDate);
      var endRests = b.Run().ToList();

      var t = new Regs.MatBalanceReg.Turnover();
      t.From(fromDate);
      t.ToEnd(toDate);
      var turns = t.Run().ToList();

      var rows = turns.Select(_ => _.MatRef)
        .Union(startRests.Select(_ => _.MatRef))
        .Union(endRests.Select(_ => _.MatRef))
        .ToDictionary(_ => _, _ => new BalLine() { MatRef = _, });

      startRests.ForEach(_ => rows[_.MatRef].StartQuantity = _.Quantity);
      endRests.ForEach(_ => rows[_.MatRef].EndQuantity = _.Quantity);
      turns.ForEach(_ => {
        rows[_.MatRef].SummaOut = _.SummaReceipt;
        rows[_.MatRef].SummaIn  = _.SummaExpence;
      });

      SetData_BalLine(rows.Values, null);
    }
  }
}
```

Этот код полностью аналогичен предыдущему отчету, но здесь мы используем не количества, а суммы, которые есть в реестре.

7.11. Печатная форма отчета

Печатная форма в нашем примере будет выглядеть так:



В полях «На начало\сумма в у.ц.» и «На конец\сумма в у.ц.» мы будем использовать выражения вида [[BalLine.StartQuantity] * [BalLine.Price]] и [[BalLine.EndQuantity] * [BalLine.Price]]

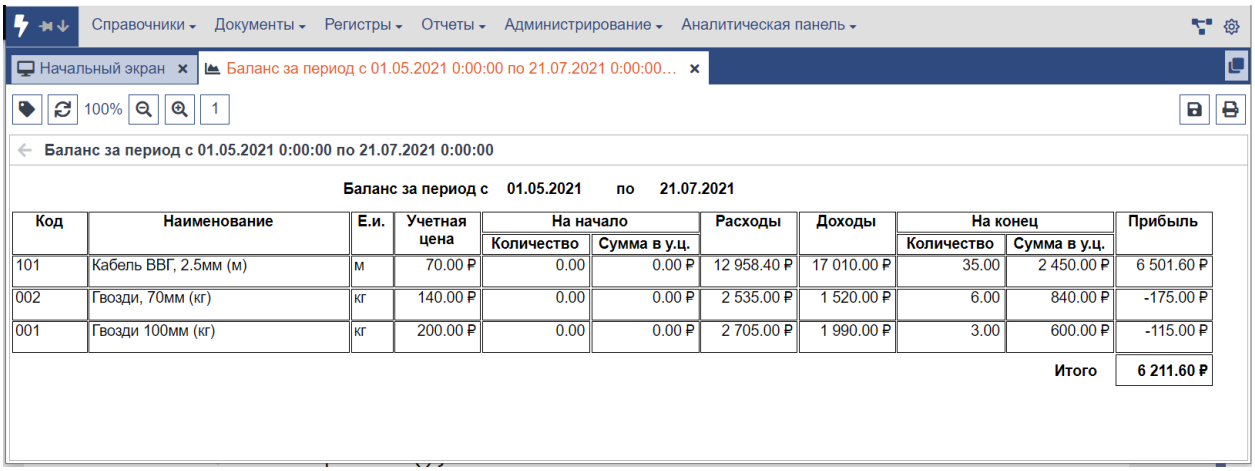
Прибыль по строке и общую прибыль рассчитаем прямо в коде отчета:

```
private void Text28_BeforePrint(object sender, EventArgs e)
    {
      Profit = ((decimal)Report.GetColumnValue("BalLine.EndQuantity") *
                (decimal)Report.GetColumnValue("BalLine.Price")) -
               ((decimal)Report.GetColumnValue("BalLine.StartQuantity") *
                (decimal)Report.GetColumnValue("BalLine.Price")) +
               (decimal)Report.GetColumnValue("BalLine.SummaIn") -
               (decimal)Report.GetColumnValue("BalLine.SummaOut");
    }

    private void Text28_AfterData(object sender, EventArgs e)
    {
      TotalProfit = TotalProfit + Profit;
    }
```

Для отображения отчета необходимо добавить соответствующую команду и пункт в главном меню:

```
import * as MatBal from "Reports.MatBal";
import * as AppCommand from "Ragtime.AppCommand";

export var command: AppCommand.Options = {
  text: "Баланс",
  id: "Reports.MatBal",
  handler: () => {
    new MatBal.Report().show();
  },
  label: {},
};


    .folder("Отчеты", f => {
      f.command("Reports.WayBill");
      f.command("Reports.MatRests");
      f.command("Reports.MatBal");
    })
```

Теперь проект можно собрать и запустить. Отчет будет выглядеть примерно так:


Справочники
Документы
Регистры
Отчеты
Администрирование
Аналитическая панель
Начальный экран
Баланс за период с 01.05.2021 0:00:00 по 21.07.2021 0:00:00…
100%
1
Баланс за период с 01.05.2021 0:00:00 по 21.07.2021 0:00:00
Баланс за период с 01.05.2021 по 21.07.2021
Код
Наименование
Е.и.
Учетная цена
На начало
Количество
Сумма в у.ц.
Расходы
Доходы
На конец
Количество
Сумма в у.ц.
Прибыль
101
Кабель ВВГ, 2.5мм (м)
м
70.00 ₽
0.00
0.00 ₽
12 958.40 ₽
17 010.00 ₽
35.00
2 450.00 ₽
6 501.60 ₽
002
Гвозди, 70мм (кг)
кг
140.00 ₽
0.00
0.00 ₽
2 535.00 ₽
1 520.00 ₽
6.00
840.00 ₽
-175.00 ₽
001
Гвозди 100мм (кг)
кг
200.00 ₽
0.00
0.00 ₽
2 705.00 ₽
1 990.00 ₽
3.00
600.00 ₽
-115.00 ₽
Итого
6 211.60 ₽

8. Перепроведение документов

В примере ниже мы покажем, как можно заново выполнить запись в регистр для выбранных документов или для всех документов одного типа. Это может понадобиться в том случае, если в процессе работы приложения потребовалось изменить состав записей, которые документ делает в регистр.

В нашем примере, на 6-м шаге, имелся вот такой код:

```
    partial void DoMotionForExecuteTransition(IMotions motions, string comment) {
      var recs = motions.MatBalanceReg.Records = new List<Regs.MatBalanceReg>();
      foreach(var line in MatLines) {
        recs.Add(new Regs.MatBalanceReg() {
          Period = DocDate,
          ItemType = Type == WayBillType.Buy ? Ragtime.AccReg.ItemType.Receipt :
Ragtime.AccReg.ItemType.Expence,
          Mat = line.Mat,
          Quantity = line.Quantity,
          Summa = -line.Summa,
        });
      }
    }
```

И накладные, которые уже были доведены до статуса «Исполнена», сделали запись в регистр с отрицательной суммой. Исправим знак записываемой регистром суммы:

```
          Summa = line.Summa,
```

Теперь, чтобы изменения отразились в базе данных, добавим миграцию:

```
namespace Main_0009_Alice {
  using global::Ragtime.Data.DbUpgrade;
  using LinqToDB;
  using System.Linq;


  public partial class Transition : global::Ragtime.Data.DbUpgrade.Transition {

    public override void UpdateData() {
      Ragtime.RefObject.RepostJob.Schedule<Docs.WayBill>(null, true);
      Ragtime.AccReg.RecalcTotalsJob.Schedule<Regs.MatBalanceReg>();
    }

  }

}
```

Пояснения к содержимому

Вызов

```
Ragtime.RefObject.RepostJob.Schedule<Docs.WayBill>(null, true);
```

создает внутреннюю задачу Ragtime, которая для каждого указанного документа (в данном случае список документов null, что означает – для всех) выполнит полное удаление записей из регистра, а затем, шаг за шагом, вызовет все переходы документа по его статусам, как записано в истории документа. Второй параметр отвечает за пересчет итогов по регистру. Чтобы сэкономить время мы указали, что не требуется делать пересчет после каждого перепроведенного документа. Вместо этого мы указали, что итоги по регистру нужно пересчитать отдельно:

```
Ragtime.AccReg.RecalcTotalsJob.Schedule<Regs.MatBalanceReg>();
```