Учебный курс «Делаем фактографию»

Марчук А.Г., д.ф.-м.н., профессор

## Введение

Под фактографической системой мы будем понимать систему фиксации (записи) фактов. Фактами в нашем случае будут сущности реального мира (люди, документы, организационные системы, географические системы) отражаемые через информационные поля или отношения между ними. Например, человек может быть описан датой рождения, организацией, в которой он работает, местом проживания, фотографиями, на которых он отражен. Мы почти полностью выстроим аналог построений Semantic Web [], позволяющих выполнять как структуризацию данных, так и специфицировать эти данные через онтологию.

Следующее относительно независимое построение – хранилище документов. В предлагаемом подходе документ, это файл (в общем случае – файлы), который содержит контент (содержимое) документа и его информационная сущность, как правило, проявляется через просмотр.

Таким образом, появляется задача формирования и использования информационной базы, состоящей из хранилища документов и базы данных. Под формированием мы будем понимать два возможных сценария: автоматическую переработку наборов файлов и второй, редактирование информационной базы. Использование информационной базы предполагается через просмотровые/поисковые интерфейсы и сервисы.

Все указанные элементы информационных и программных построений будут опробованы в простой фактографической системы. От «сложного», комплексного решения построенный вариант будет отличаться заметным упрощением моделей, особенностей программной реализации и интерфейсов.

Курс рассчитан на изучение ряда идей и технологий, ориентированных на создание современных информационных систем. Для студентов необходимо содержательное знание C#/LINQ/.NET/ASP.NET/XML/HTML и опыт их использования.

## Предварительная проверка знаний и умений

Упомянутое содержательное знание требует конкретизации и проверки. Предложим несколько тестовых примеров, которые помогут вам определить самостоятельно готовность к восприятию следующего материала учебного курса.

Тесты на C#, LINQ

Необходимо в консольном приложении сделать следующие тесты:

1. Завести массив целых, заполнить его несколькими значениями. Произвести фильтрацию потока, оставив только четные, произвести функциональное преобразование потока, напр. возведение в квадрат, вычислить сумму элементов.
2. Задать длину последовательности, сгенерировать поток из этого числа псевдослучайных целых значений, убрать повторы, сделать функциональное преобразование потока, породив из случайного числа запись { id, name, age } в которой id – это случайное значение, name – строковое значение этого случайного значения, age – случайное значение в диапазоне 1-100. Далее, отсортировать поток записей по id, превратить в массив записей.
3. Делается процедура получения записи по идентификатору из массив записей предыдущего примера.
4. В технологии ASP.NET делается Web-приложение в котором строится страница со всеми фотографиями из некоторой директории.

## Тестовый набор данных Фототека

Для опробования разных подходов, нам потребуется простой генератор тестов, в котором по заданному параметру можно породить тестовую базу данных с некоторыми существенными для нас свойствами. Тест сделан намеренно максимально простым и легко воспроизводим в рамках практически любого решения. Тест сделан в парадигме связанных таблиц и приспособлен для максимально эффективной реализации на обычных SQL-платформах.

База данных состоит из трех таблиц: персон, фотографий, отражений. Параметр тестового набора - количество записей в таблице персон или просто количество персон npersons. Количество фотографий npersons \* 2. Количество отражений npersons \* 6. Все таблицы имеют идентификационное поле id, являющееся первичным ключом для таблицы, все ключи в одной таблице различны и могут быть целочисленными или строковыми. Уникальность идентификаторов поддерживается или на уровне одной таблицы или для всех записей всех таблиц в зависимости от поставленных для экспериментов задач. Таблица персон persons содержит три поля: целочисленный идентификатор id, строковое значение имени name, числовое (возможно целое или длинное целое или вещественное, это зависит от целей тестирования) значение возраста age. Таблица фотографий photos состоит из полей id и name, таблица отражений reflections связывает записи о персонах с записями о фотографиях и содержит поля id, reflected, indoc. Причем два последних поля представляют внешние ссылки (external keys) на соответственно таблицу персон (отражаемое) и таблицу фотографий (отражение в документе).

Нагрузочное тестирование и измерение проводятся по небольшому, но принципиальному спектру вопросов. Это - время загрузки, отдельно или вместе - время вычисления индексов. Это - время выборки записей по задаваемому случайным образом ключу, это - время выборки записей по задаваемому имени или части имени. Это - время вычисления "информационного портрета" персоны. Под этим вычислением понимается получение найденной по идентификатору персоны и всех фотографий, на которых отражена персона со случайно задаваемым первичным ключом. Времена измеряются в миллисекундах или в секундах, для больших значений. Для "быстрых" действий по выборкам, измерения производится на подходящем количестве испытаний и приводятся к миллисекундам на 1 тыс. испытаний.

Надо отметить, что тест "фототека" иногда используется в неполном виде. В качестве базы данных можно взять только одну таблицу, мы берем таблицу персон и это будет вполне осмысленная (тестовая) база данных. Эта база данных может рассматриваться как key-value хранилище, где ключом является идентификатор персоны, а значение - какой-то вариант полной записи об этой персоне.

## Основы RDF

RDF (Resource Description Framework) [] – способ структуризации данных, используемый в Semantic Web []. Существует несколько моделей RDF как системы структуризации данных. Чаще всего используется модель ориентированного графа. В этой модели имеются идентифицированные узлы, идентифицированные ребра и строковые константные значения. Исток ребра – всегда узел, сток (цель) ребра – может быть узел, может быть константой.

Изобразительные свойства такого построения достаточны для самых разных задач. Для иллюстрации, рассмотрим фрагмент сети, описывающий типовую ситуацию:

Рис 1. Фрагмент графа RDF

Овалы соответствуют узлам графа, прямоугольники – константам, подписи в овалах и надписи у дуг – их идентификаторы, константные значения изображены внутри прямоугольников. Суть структурирующего построения достаточно понятна из рисунка: есть идентифицированная персона Иванов 1988 года рождения. У Иванова есть отец и Иванов работает в организации с именем НГУ.

Рассмотрим разные виды представления таких графов. Первая – в виде триплетов:

Рис 2. Составные части графа RDF

В этом представлении, примитивами являются два вида троек «овал – стрелка – овал» и «овал – стрелка – прямоугольник». Это высказывания. Отдельные высказывания собираются в граф через склеивание одноименных овалов. Оба варианта называются триплетами (тройками) и слева направо их составляющие называются субъект, предикат, объект. В стандартизованном виде триплеты изображаются своими тремя частями, последовательно через пробел расположенными в виде строки:

<id1> <P1> <id2>. или

<id3> <P2> ”data”.

Еще есть XML-форма изображения RDF-графа. Суть ее в том, что множество высказываний разбивается на группы с одинаковым субъектом, а группы изображаются в виде записи специального формата, напр.:

<rdf:RDF>

<person rdf:about=“p3817”>

<name>Иванов Иван Иванович</name>

<birth>1988</birth>

<work-in rdf:resource=“o19302”/>

</person>

<org-sys rdf:about=“o19302”>

<name xml:lang=”ru”>НГУ</name>

<name xml:lang=”en”>NSU</name>

</org-sys>

</rdf:RDF>

Запись помещается в элемент с именем типа и идентификатором в атрибуте rdf:about. Поля записи – это элементы с именем предиката и либо константным значением в виде текста, либо ссылкой на объект через атрибут rdf:resource. База данных будет состоять из одного или нескольких RDF-файлов формата XML. Несколько файлов эквивалентны одному файлу с объединенным набором записей.

Такая форма похожа на традиционные построения средствами XML для реляционных баз данных. Действительно, имя записи соответствует имени таблицы, атрибуты rdf:about и ref:resource соответствуют первичному и внешнему ключу соответственно, имя поля соответствует имени колонки. Но RDF – шире, он позволяет 1) иметь элементы с отсутствующим типом; 2) иметь неопределенный набор полей; 3) иметь множество полей с одинаковым именем, но разными значениями, в частности, на разных языках.

## Построение тестового набора данных

Вот мы и добрались до создания кода. Желательно в этом месте создать решение (Solution) для Visual Studio, проекты в этом решении будут отдельными задачами, некоторые из которых будут использованы в других проектах. Первая задача – создание RDF-файла в формате XML, содержащего базу данных определенного параметром размера. Сделаем это в проекте GenerateRDF. Это будет вариант известного генератора Фототека []. Опуская детали, получающийся код довольно прост. Сначала мы формируем пустой набор записей:

// Загружаем основу

XElement dataset = XElement.Parse(

@"<?xml version='1.0' encoding='utf-8'?>

<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>

</rdf:RDF>

");

Этот фрагмент я сделал экзотически, через парсинг текста, потому что не умею внедрять пространство имен другим образом. Наверное, есть и более естественный способ типа:

XElement dataset = new XElement(“{http://www.w3.org/1999/02/22-rdf-syntax-ns#}RDF”, …);

Далее, мы наполняем этот элемент записями в соответствии с логикой построения Фототека:

Random rnd = new Random();

int npersons = 10;

int np = npersons;

int nf = npersons \* 2;

int nr = npersons \* 6;

dataset.Add(Enumerable.Range(0, np)

.Select(i => new XElement("person", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}about", "p"+i), new XElement("name", new XAttribute("{http://www.w3.org/XML/1998/namespace}lang", "ru"), "и"+i),

new XElement("age", "23"))));

dataset.Add(Enumerable.Range(0, nf)

.Select(i => new XElement("photo", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}about", "f" + i),

new XElement("name", "dsp" + i))));

dataset.Add(Enumerable.Range(0, nr)

.Select(i => new XElement("reflection", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}about", "r" + i),

new XElement("reflected", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}resource", "p" + rnd.Next(np))),

new XElement("indoc", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}resource", "f" + rnd.Next(nf))))));

Здесь стоит обратить внимание на внедрение русского спецификатора при задании имени персона (которое начинается с русской «и»). Финальный аккорд – сохранение полученного XML-элемента dataset в файле.

dataset.Save("../../../output.rdf");

Можно расслабиться и посмотреть на полученную базу данных. Поскольку это XML, то есть много способов это сделать. Можно посмотреть с помощью любого браузера. Можно посмотреть с помощью VisualStudio. В последнем случае, можно еще и редактировать этот текстовый файл. При редактировании нет гарантии, что XML будем оставаться правильным – это зависит от внимательности оператора. Итак, база данных может быть сгенерирована или создана с помощью текстового редактора. Она может быть загружена оператором

var xdb = XElement.Load(filepath);

Она может быть изменена с помощью текстового редактора или изменением объекта базы данных xbd с последующим сохранением в файле. К база денных также предъявляют требования возможности осуществления типовых запросов на выборку данных.

Первое требуемое существенное действие – поиск записи или записей по имени или части имени. Логично желать получить из базы данных все записи, совпадающие с заданным образцом имени, напр. задаем «Иванов», получаем всех Ивановых, имеющихся в базе данных. Что такое «получаем»? Получаем набор записей вида:

<person rdf:about="…">

<name xml:lang="ru">Иванов …</name>

…

</person>

У нас в данных Ивановых нет, но можно искать персоны, начинающиеся напр. на «и777». Напишем метод, находящий в наборе записей ту, имя которой похоже на образец:

// Поиск по образцу

Func<XElement, string, IEnumerable<XElement>> Search = (ds, sample) => ds.Elements()

.Where(r => r.Elements("name").Any(f => f.Value.StartsWith(sample)));

Другой способ получения информации из базы данных – выборка по значению идентификатора:

// Выборка записи по идентификатору

Func<XElement, string, XElement> GetById =

(ds, id) => ds.Elements().FirstOrDefault(r => r.Attribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}about").Value == id);

На имеющихся тестовых данных можно легко испытать работу этих функций. Но это только пробы. Для реальной и даже модельной работы с фактографической системой, нужно создать более развитую базу данных. Сначала функциональность базы данных будет минимальной, в дальнейшем будем наращивать функциональность и применять более изощренные решения.

Глядя вперед, мы создадим проекты RDFEngine (библиотека), TesterConsole (консоль) и FactographyView (веб-приложение).

## Первый шаг на пути к базе данных

Созданный файл формата RDF обладает структурирующими свойствами. Он текстовый, его можно посмотреть, создать «вручную», редактировать, легко загрузить в оперативную память и легко выгрузить, ну и использовать для обработки. Посмотрим, как можно использовать эту структуру для нашей тестовой информационной системы. Для этого, надо разобраться в том, как и какие данные мы будем извлекать из будущей базы данных.

Вообще-то для работы с RDF существует рекомендованный язык запросов Sparql []. Это своего рода “SQL” для триплетной базы данных. Формат довольно сложный как в понимании, так и в реализации. Мы попробуем решать наши задачи (изучения фактографии и фактографических систем) введя более простые формы, основанные на объектном представлении кусков графа. Это в чем-то похоже на классическую алгоритмику работы со структурными данными (списки, деревья, и др.), когда элементы структуры представляются в виде объектов, имеющих ссылки на другие элементы. Главное в этом представлении – понятие записи.

Рис 3. Выделение записи в графе RDF

На рисунке сделана попытка черным контуром выделить то, что мы будем называть записью. В запись входит фокусный узел (на рисунке p3817), имеющиеся у узла текстовые поля и исходящие объектные ссылки, напр. ссылка на o19302. Легко видеть, что полный RDF-граф может рассматриваться как множество записей с отождествлением адресатов ссылок записям с соответствующим идентификатором.

Построим объектное представление графа через разбиение на записи. Мы видим, что запись определяется идентификатором, типом и «стрелками» двух видов: полями и объектными ссылками, т.е. ссылками, ведущими к другим записям. Поле, это имя свойства (идентификатор стрелки) и текстовая константа. Объектная ссылка это также имя свойства и ссылка на другую запись. Начальный вариант записи:

public class RRecord

{

public string Id { get; set; }

public string Tp { get; set; }

}

Но еще нужны «стрелки», назовем их свойствами . Два вида свойств определим через общий абстрактный класс и два варианта. Эти определения, совместно с расширением класса RRecord приведены ниже:

public abstract class RProperty

{

public string Prop { get; set; }

}

public class RField : RProperty

{

public string Value { get; set; }

}

public class RLink : RProperty

{

public string Resource { get; set; }

}

public class RRecord

{

public string Id { get; set; }

public string Tp { get; set; }

public RProperty[] Props { get; set; }

}

В принципе, это все. Будут еще расширения, но основа достаточно простая, поскольку простым является сам формат RDF.

Воспользуемся классом RRecord для создания примитивной in-memory базы данных. Логически, база данных должна представлять собой множество идентифицированных записей с некоторым протоколом и интерфейсом взаимодействия с внешним миром. Позволю себе назвать такую базу данных «движком» и предложить следующий вариант интерфейса (файл IEngine.cs в проекте RDFEngine):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Xml.Linq;

namespace RDFEngine

{

public interface IEngine

{

public void Clear();

public void Load(IEnumerable<XElement> records);

public void Build();

public RRecord GetRRecord(string id);

public IEnumerable<RRecord> RSearch(string searchstring);

// Константы для удобства

public static XName rdfabout = XName.Get("about", "http://www.w3.org/1999/02/22-rdf-syntax-ns#");

public static XName rdfresource = XName.Get("resource", "http://www.w3.org/1999/02/22-rdf-syntax-ns#");

}

}

Предполагается естественный протокол работы движка: очистка (если надо), загрузка, построение и дальше множество запросов поиска и выборок. Реализация интерфейса может быть разной, мы сейчас сделаем вариант, основанный на объектах класса RRecord. Сосредоточимся на проектировании класса REngine, наследующего интерфейс IEngine. Проектируем сверху вниз, обращаем внимание на комментарии:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Xml.Linq;

namespace RDFEngine

{

// Движок, основанный на объектном R-представлении есть набор записей, сформированных в Dictionary

public class REngine : IEngine

{

// База данных будет:

private IDictionary<string, RRecord> rdatabase;

// Очистка не нужна, поскольку она осуществляется автоматически при загрузке, но все же сделаем…

public void Clear() { rdatabase.Clear(); }

// Загружается поток записей в XML-представлении

public void Load(IEnumerable<XElement> records)

{

rdatabase = records.Select(x => new RRecord()

{

Id = x.Attribute(IEngine.rdfabout).Value,

Tp = x.Name.NamespaceName + x.Name.LocalName,

Props = x.Elements().Select<XElement, RProperty>(el =>

{

string prop = el.Name.NamespaceName + el.Name.LocalName;

XAttribute resource = el.Attribute(IEngine.rdfresource);

if (resource == null)

{ // Поле TODO: надо учесть языковый спецификатор, а может, и тип

return new RField() { Prop = prop, Value = el.Value };

}

else

{ // ссылка

return new RLink() { Prop = prop, Resource = resource.Value };

}

}).ToArray()

}).ToDictionary(rr => rr.Id);

}

// Ничего делать не надо, все сделано при загрузке

public void Build() { }

public RRecord GetRRecord(string id)

{

RRecord rr;

if (rdatabase.TryGetValue(id, out rr))

{

return rr;

}

return null;

}

// Поиск делаем перебором записей с проверкой

public IEnumerable<RRecord> RSearch(string searchstring)

{

return rdatabase

.Select(pair => pair.Value)

.Where(rr =>

{

return rr.Props.Any(p => p is RField && ((RField)p).Prop == "name" && ((RField)p).Value.ToLower().StartsWith(searchstring));

});

}

}

}

Теперь надо проверить работоспособность. Можно сделать ее в консольном приложении. Надо 1) задать количество персон; 2) создать объект движка; 3) проверить то, что одиночный поиск по части имени работает; 4) проверить одиночную выборку по идентификатору; 5) изменить количество персон, напр. на 10 тыс.; 6) измерить производительность выборок по идентификатору; 6) измерить производительность выборок по имени. Когда вы отладите написанные программы, вы выясните, что выборки выполняются практически мгновенно, поиск выполняется дольше, и время поиска линейно увеличивается при увеличении количества персон. Можно также выяснить, что до количества персон в 100 тыс., т.е. для большинства модельных постановок, временные характеристики работы движка являются приемлемыми. Код тестирующего приложения я приводить не буду, поскольку он простой, созидайте!

## Делаем простое Web-приложение

Мы построили базу данных персон, фотографий и отношений (отражение) между ними. Как посмотреть эту базу данных? Отбросим варианты перебора. Если просматривать не все подряд, то сначала надо задать имя персоны или его часть и найти кандидатов. Среди них визуально найти нужную персону и построить так называемый информационный портрет. Что такое информационный потрет персоны? Что у нас есть про персону в базе данных. Это ее имя и возраст. Еще мы знаем в каких фотографиях отражена данная персона. Собственно это все, что мы знаем. Попробуем свести все знания в единую композицию и гиперссылки перехода между разными страницами. Начинаем проект FactographyView. Объявим “Веб-приложение ASP.NET Core (модель-представление-контроллер)”. Если кто-то умеет делать Web-проекты по-другому, можно и по-другому. Но я буду вести по разработке именно в ASP.NET (MVC).

В порожденном по шаблону проекту, сделаем некоторые преобразования, которые нам нужны. Во-первых, изменим общее оформление страниц сайта изменив HTML-текст в Views/Shared/\_Layout.cshtml. Уберем всю внутренность элемента body и вставим в него только @RenderBody(). Что-нибудь вроде:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

</head>

<body>

@RenderBody()

</body>

</html>

Анализируя Startup.cs, мы выясняем, что по умолчанию запускается контроллер Index() из раздела Home и что некоторый дополнительный параметр означает идентификатор id. Чтобы следить за получающимся результатом, оформим индексную (начальную) страничку. Сначала мы в контроллере изменим реакцию на Index добавив строковый параметр id и передавая этот параметр в качестве модели в построитель Index.cshtml. Фрагменты изменений будут выглядеть приблизительно так:

public IActionResult Index(string id)

{

return View("Index", id);

}

@model string

<h1>FactographView</h1>

<div>@Model</div>

При запуске приложения, мы увидим заголовок, но не увидим идентификатора. Но если в адресной строке браузера изменить запуск, добавив после начального варианта запуска /Home/Index/p123, результат будет содержать и идентификатор p123. Для тех, кто не разбирается в нюансах клиент-серверных приложений поясню, что полная форма обращения к данному Web-приложению будет что-то типа <https://localhost:44305/Home/Index/p123>. А начальный запуск <https://localhost:44305> интерпретируется в <https://localhost:44305/Home/Index> в соответствии с заданным в Startup.cs шаблоном, где идентификатор будет пустой строкой. Если добавить идентификатор, то получится полная форма.

Теперь будем строить портрет. Нам понадобится база данных. И доступ к базе данных из контроллера, а может и из других мест. Самое простое решение сделать публичные статические поля в каком-нибудь классе. Давайте добавим класс информационной базы, там будет и база данных или движок и другие данные, необходимые для работы приложения. По аналогии с Database, назовем этот класс Infobase. В него мы пока поместим только движок. Не забываем указать для проекта использование проекта RDFEngine:

public class Infobase

{

public static RDFEngine.IEngine engine = null;

}

Движок надо инициализировать и загрузить, сделаем это в классе Startup, добавляя в обязательно и однократно исполняемое действие, напр. Configure инициализацию, загрузку и построение:

Infobase.engine = new RDFEngine.REngine();

Infobase.engine.Load(RDFEngine.PhototekaGenerator.Generate(100));

Infobase.engine.Build();

Приступаем к проектированию решения. Мы хотим сформировать страничку, на которой по заданному идентификатору персоны строится портрет этой персоны. Проектирование страницы начинается с проектирования контроллера. Контроллер принимает обращение извне, выделает из него параметры и передает в построитель вида (вьюер) некоторые данные (модель), построенные из параметров и базы данных. Модель будет использована в композиции.

Это действие назовем Person, соответственно контроллер будет обрабатывать запрос на персону с идентификатором id. Все, что мы умеем пока делать, это получать запись с таким идентификатором. Эту запись вычислить и передать построителю вида:

public IActionResult Person(string id)

{

RRecord model = Infobase.engine.GetRRecord(id);

return View("Person", model);

}

Создадим начальный вариант вьюера. Мы просто напишем, что это «Персона», идентификатор персоны не имеет «человеческого» значения, мы его выдавать не будем, зато выдадим список свойств:

@using RDFEngine

@model RRecord

@{

if (Model == null) return;

}

<h2>Персона</h2>

<div>

<table>

@foreach (var field in Model.Props)

{

RField f = (RField)field;

<tr><td>@f.Prop</td><td><b>@f.Value</b></td></tr>

}

</table>

</div>

Испытаем решение. Запустим сайт. Скорее всего, вы не увидите того, что ожидали, прост потому, что по умолчанию вызывается индексная страница. Чтобы вызвать персону, надо это сформулировать явно: <https://localhost:44305/Home/Person/p23>. В этом запуске вызывается действие (action) Person в контроллере Home и с параметром id=p23.

Замысел о том, чтобы в портрете персоны фигурировали фотографии, в которых персона отражена, не реализован. Это связано с тем, что у нас нет хорошего метода выявления таких фотографий. Посмотрим внимательно на граф. Хочется иметь определитель всех обратных стрелок для фокусного узла. То есть, информацию { узел, идентификатор обратной стрелки, запись, откуда идет стрелка }. Попробуем решить эту задачу тем, что в группе определений записи введем еще один вид «стрелки» свойства:

public class RInverse : RProperty

{

public string Source { get; set; }

}

Этот вариант будет означать, что на фокусный узел указывает стрелка с именем Prop и ведущая от узла с идентификатором Source.

Введя этот вариант, нам есть куда записывать в запись данные об обратных ссылках. Назовем такую запись расширенной. Превратим базу данных в набор расширенных записей. Это делается в модуле REngine. Потребуется несколько принципиальных добавлений. В метод Load() добавим словарь списков обратных ссылок:

Dictionary<string, List<RInverse>> inverseDic = new Dictionary<string, List<RInverse>>();

Action<string, RInverse> AddInverse = (id, ilink) =>

{

if (!inverseDic.ContainsKey(id)) inverseDic.Add(id, new List<RInverse>());

inverseDic[id].Add(ilink);

};

В этом словаре, в дальнейшем, созданные «стрелки» RInverse группируются по значению идентификатора «конца» стрелки. Это делается еще одним добавлением в LINQ-выражение в рамках обработки объектной ссылки RDF/XML. В этом месте добавляется:

// Создадим RInverse и добавим в inverseDic

AddInverse(resource.Value, new RInverse() { Prop = prop, Source = nodeId });

Сформированный таким образом словарь, далее используется для того, чтобы добавить обратные отношения в записи:

// Вставим наработанные обратные ссылки

foreach (var pair in inverseDic)

{

string id = pair.Key;

var list = pair.Value;

var node = rdatabase[id];

node.Props = node.Props.Concat(list).ToArray();

}

Такое изменение в структуре графа позволяет двигаться по графу не только по стрелкам, но и против стрелок. Нас интересует некоторое окружение фокусного узла, которое мы рассматриваем портретом. Модифицируем ранее построенный портрет персоны. Теперь запись стала уже расширенной, ее и распишем по вариантам RField, RLink, RInverse (файл Person.cshtml):

@foreach (var property in Model.Props)

{

if (property is RField)

{

…

}

else if (property is RLink)

…

{

}

else if (property is RInverse)

{

…

}

}

Здесь в один список сводятся поля, прямые ссылки и обратные ссылки. Что мы видим при испытании? Для данных тестовых данных, мы видим поля (наименование, значение) и обратные ссылки одного вида Reflected вместе со значениями идентификаторов узлов, от которых идет данная ссылка. Значения идентификаторов мало что дают пока мы не построим портрет записи с каким-то из этих идентификаторов. Если бы в данных были бы прямые ссылки, ситуация была бы аналогичная. Сделаем принципиальное решение: вместо идентификаторов внешних объектов, на портрет поместим гиперссылки, ведущие к этим объектам. По идее, мы должны бы строить портреты объекты каких-то других типов, но у нас есть только портрет персоны, так что именно им и воспользуемся. Заменим resource и source на гиперссылку <a href=”~/Home/Person/@(re)source”>текст гиперссылки</a>. Итоговое построение:

<table>

@foreach (var property in Model.Props)

{

if (property is RField)

{

RField f = (RField)property;

string value = f.Value;

<tr><td>@property.Prop</td><td><b>@value</b></td></tr>

}

else if (property is RLink)

{

RLink l = (RLink)property;

string resource = l.Resource;

<tr><td>@property.Prop</td><td><a href='/Home/Person/@(resource)'>прямая ссылка</a></td></tr>

}

else if (property is RInverse)

{

RInverse i = (RInverse)property;

string source = i.Source;

<tr><td>@property.Prop</td><td><a href='/Home/Person/@source'>обратная ссылка</a></td></tr>

}

}

</table>

Теперь, при испытании, мы видим, что получилась информационная система, в которой мы можем по ссылкам переходить от одних записей к другим. Есть некоторое несоответствие в типах изображаемых объектов изначально декларированному типу персоны, но это легко устраняется тем, что имя типа можно взять из поля Tp и напечатать. Мы получили портрет сущностей, записанных в базе данных. Создадим страницу Portrait и перенесем все полезное, что есть в странице Person. После этого будем совершенствовать портретную композицию.

## Начинаем изучать онтологию

Понятие онтологии [] является принципиальным для построений SemanticWeb. Если коротко, то онтология – спецификация данных, сформированных в виде графа RDF. Вообще-то не только RDF и не только спецификация, но это уже детали…

Онтология это формальное описание, Мы воспользуемся формализмом OWL []. Рассмотрим пример, описывающий примененную в предыдущих примерах структуризацию:

<owl:Ontology

xmlns:owl ="http://www.w3.org/2002/07/owl#"

xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<owl:Class rdf:about="person">

<rdfs:label>Персона</rdfs:label>

</owl:Class>

…

<owl:DatatypeProperty rdf:about="name">

<rdfs:label xml:lang="ru">имя</rdfs:label>

</owl:DatatypeProperty>

…

<owl:ObjectProperty rdf:about="reflected">

<rdfs:label xml:lang="ru">отражаемое</rdfs:label>

<rdfs:inverse-label xml:lang="ru">отраж. в документе</rdfs:inverse-label>

</owl:ObjectProperty>

…

</owl:Ontology>

В этом фрагменте описываются понятия в виде классов, в виде свойств данных, в виде объектных свойств. В каждом элементе определено «человеческое» имя понятия, это позволяет идентификатор понятия не делать известным словом. Другие характеристики структуризации пока нам не существенны и не определены. Фактически, классы задают типы сущностей (овалов), объектные свойства и свойства данных задают типы стрелок. Более детальное описание этой онтологии дается в файле SimpleOntology.owl.

В данном варианте мы можем использовать онтологию для формирования словаря перевода онтологических идентификаторов в «человеческие» термины. Для каждого понятия есть его метка rdfs:label, задающая термин на том или ином языке. В данном случае, язык – только русский, но легко можно добавить термины и на других языках (английском, французском, арабском и т.д.). Для объектных свойств может быть задана и другая метка rdfs:inverse-label. Это связано с тем, что в зависимости от позиции наблюдателя противоположное отношение называется по-разному. Например, между персонами может быть установлено объектное отношение father, т.е. указатель на отца. Соответственно, со стороны истока стрелки, это «отец» персоны. А со стороны отца, противоположный объект это «ребенок».

Давайте применим онтологию для нашего проекта построения портрета. Мы закачаем файл онтологии в информационную систему (статическим объектом класса Infobase), потом превратим его в пару словарей и методы доступа:

private static XElement ontology = null;

private static Dictionary<string, string> labels\_ru;

private static Dictionary<string, string> inverse\_labels\_ru;

public static void LoadOntology(string path)

{

ontology = XElement.Load(path);

labels\_ru = ontology.Elements()

.SelectMany(el => el.Elements("{http://www.w3.org/2000/01/rdf-schema#}label"))

.Where(lab => lab.Attribute("{http://www.w3.org/XML/1998/namespace}lang")?.Value == "ru")

.ToDictionary(lab => lab.Parent.Attribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}about").Value,

lab => lab.Value);

inverse\_labels\_ru = ontology.Elements()

.SelectMany(el => el.Elements("{http://www.w3.org/2000/01/rdf-schema#}inverse-label"))

.Where(lab => lab.Attribute("{http://www.w3.org/XML/1998/namespace}lang")?.Value == "ru")

.ToDictionary(lab => lab.Parent.Attribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}about").Value,

lab => lab.Value);

}

public static string GetTerm(string id)

{

if (!labels\_ru.ContainsKey(id)) return id;

return labels\_ru[id];

}

public static string GetInvTerm(string id)

{

if (!inverse\_labels\_ru.ContainsKey(id)) return id;

return inverse\_labels\_ru[id];

}

Немножко громоздко, но зато универсально – можно подставлять любую онтологию. Теперь не забудем поместить онтологию в директорию wwwroot, загрузить онтологию в Startup’е. А сама терминология внедряется в Portrait.cshtml там, где есть свойство Prop. Только в прямых вариантах используется базовые термины GetTerm, а в обратном свойстве – обратные термины GetInvTerm. Еще тип Tp надо перевести. Если Вы нигде не ошиблись, то движение по информационной системе будет более осмысленно.

Однако, имеется существенное неудобство: от персоны к фотографии путь состоит из двух шагов и на портрете персоны нам не получается видеть фотографии, а на портрете фотографии, не получится увидеть персонажей.

Будем добиваться этого эффекта. Внимательно посмотрим на портрет персоны. В нем все хорошо, кроме того, что “обратная ссылка” не превращена в дальнейший путь к фотке. Попробуем это сделать. На графовом уровне вместо гиперссылки нужно раскрыть запись, на которую ссылка ведет. Точнее, откуда стрелка идет. В общем случае, есть два варианта: исходящая ссылка и входящая ссылка. В любом случае, «на том конце» находится запись. Прочитаем ее по идентификатору. Наше построение будет следующее. Если это прямая ссылка, то мы расписываем только поля в виде «имя поля : значение поля». Если это обратная ссылка, то мы прочитываем «обратную» запись, расписываем ее поля, потом «визуализируем» ссылку, если таковая найдется и она не совпадает с обратной. Некоторый итог преобразований вьюера находится в текущем состоянии Portrait.cshtml.

Если Вам самостоятельно удалось справиться с модернизацией, описанной в последнем абзаце – Вы герой! Если нет, не пугайтесь, рассмотренные вопросы довольно непривычны и не так просто в них ориентироваться.

## Простые портретные проекции RDF-графа

База данных, с которой мы работаем, представляет собой ориентированный граф структуры RDF. В «классическом» случае граф представляет собой множество триплетов, визуально выявить из которого информацию сложно. В качестве примера, рассмотрим фрагмент, соответствующий графу, фигурировавшему ранее в качестве иллюстрации.

<p3817> <father> <p2817>.

<o19302> <type> <org-sys>.

<p3817> <birth> “1988”.

<o19302> <name> “НГУ”.

<p3817> <name> “Иванов”.

<p3817> <type> <person>.

Понять связи объектов и к чему относятся значения полей, довольно затруднительно. Использование RDF/XML формата делает граф несколько более понятным:

<rdf:RDF>

<person rdf:about=“p3817”>

<name>Иванов Иван Иванович</name>

<birth>1988</birth>

<work-in rdf:resource=“o19302”/>

</person>

<org-sys rdf:about=“o19302”>

<name>НГУ</name>

</org-sys>

</rdf:RDF>

В нем информация располагается в виде произвольного расположения узлов, сопряженных с ними исходящих дуг и неявного, трудного для визуального восприятия, сопоставления ссылок rdf:resource с идентификаторами элементов rdf:about. В представленном случае, мы порождаем простую древовидную структуру для информационной группы. Это – введенная ранее «запись». В нашем случае, задача состоит в том, чтобы предложить древовидное построение, соответствующее группированию вокруг фокусного узла. Элементы этого древовидного построения можно будет в дальнейшем отображать на визуальную плоскость.

Принципиальным является построение дерева информационных элементов, начинающееся от фокусного узла. Напр. Корень дерева – идентификатор узла. Далее – тип узла. Следующий уровень «ветвей» - свойства. Как мы помним, есть свойства данных (datatype properties) и есть объектные свойства (object properties). Под портретом или портретной проекцией, мы будем понимать окрестность фокусного узла или какую-то ее часть. Окрестность формируется через смежные узлы и дуги.

Рис 4. Формирование портрета разного уровня

Портрет нулевого уровня формируется как пара овал-значение типа с добавленными полями (Datatype properties). Портрет первого уровня представляет собой портрет нулевого уровня с добавленной информацией о сущностях, на которые фокусный узел прямо ссылается. Эта информация состоит из имени предиката ссылки (ObjectProperty) и портрета нулевого уровня для ссылаемой сущности. Портрет второго уровня представляет собой портрет первого уровня с добавленной информацией о сущностях, которые прямо ссылаются на фокусный узел. Информация состоит из имени предиката обратной ссылки и портрета нулевого уровня для ссылающейся сущности.

Собственно это построение было создано в Portrait.cshtml. Повторим разработку с бОльшим пониманием того, что мы делаем. Для этого, заведем в котроллере Home действие Portrait1. При обработке идентификатора, являющегося параметром действия, мы вычислим объект, соответствующий портрету второго уровня и передадим его построителю (View) Portrait1.cshtml. Там мы его визуализируем.

Для начала, построим объект в текстовом виде. Для понятности, будем придерживаться нотации, близкой JSON, т.е. записи будем изображать набором пар ключ-значение, последовательности набором значений, помещенных в квадратные скобки. Разделителем будет запятая. Определим функцию, вычисляющую портрет по идентификатору:

// Фрагмент HomeConroller

private static string BuildPortraitText(string id, int level, string forbidden)

{

var rec = Infobase.engine.GetRRecord(id);

if (rec == null) return null;

string result = "{Id:" + rec.Id + ", Tp:" + rec.Tp + ", " +

rec.Props.Select<RDFEngine.RProperty, string>(p =>

{

if (p is RDFEngine.RField) return p.Prop + ":" + ((RDFEngine.RField)p).Value;

else if (level > 0 && p is RDFEngine.RLink && p.Prop != forbidden) return p.Prop + ":" + BuildPortraitText(((RDFEngine.RLink)p).Resource, 0, null);

else if (level > 1 && p is RDFEngine.RInverseLink) return p.Prop + ":" + BuildPortraitText(((RDFEngine.RInverseLink)p).Source, 1, p.Prop);

return null;

}).Where(s => s != null).Aggregate((a, s) => a + ", " + s) + "}";

return result;

}

Обратите внимание на параметр forbidden, он нужен для того, чтобы при визуализации записи, не визуализовать еще раз ту, запись, из котором мы «пришли» по обратной ссылке.

Потом применим ее при реализации действия Portrait1 и переданную модель визуализируем как текст напр.:  
// Фрагмент HomeConroller

public IActionResult Portrait1(string id)

{

var model = BuildPortraitText(id, 2, null);

return View("Portrait1", model);

}

@\* Текст Portrait1.cshtml \*@

@model string

<div>@Model</div>

Проверим работоспособность. Если надо, исправим ошибки. Чем это решение плохо? Тем, что структурированный портрет не имеет полезной html-разметки. Дальнейшее совершенствование портретной композиции может пойти по двум направлениям: построение модели в виде html-объекта или текста и использование его через @Html.Raw(htmltext) или построение модели в виде объекта и потом использование его для построения вида. Второй способ логичнее и соответствует принципам Model-View-Controller, поэтому на нем и сосредоточимся.

В какую структуру упаковать информацию, требующуюся для построения портрета? Конечно, в RRecord! Точнее, в расширенную запись RRecord. В конструкцию RRecord мы введем возможность непосредственной вставки узлов, для построения рекуррентного дерева. Это достигается там, что в дополнение к существующим вариантам «стрелок» RField, RLink, RInverseLink, мы вводим варианты прямой вставки узлов:

public class RDirect : RProperty

{

public RRecord DRec { get; set; }

}

public class RInverse : RProperty

{

public RRecord IRec { get; set; }

}

Опробуем новый подход. Аналогично предыдущему, введем действие Portrait2, а к нему функцию RRecord BuildPortrait(string id, int level, string forbidden):

private static RRecord BuildPortrait(string id)

{

return BP(id, 2, null);

}

private static RRecord BP(string id, int level, string forbidden)

{

var rec = Infobase.engine.GetRRecord(id);

if (rec == null) return null;

RRecord result\_rec = new RRecord()

{

Id = rec.Id,

Tp = rec.Tp,

Props = rec.Props.Select<RProperty, RProperty>(p =>

{

if (p is RField)

return new RField() { Prop = p.Prop, Value = ((RField)p).Value };

else if (level > 0 && p is RLink && p.Prop != forbidden)

return new RDirect() { Prop = p.Prop, DRec = BP(((RLink)p).Resource, 0, null) };

else if (level > 1 && p is RInverseLink)

return new RInverse() { Prop = p.Prop, IRec = BP(((RInverseLink)p).Source, 1, p.Prop) };

return null;

}).Where(p => p != null).ToArray()

};

return result\_rec;

}

public IActionResult Portrait2(string id)

{

var model = BuildPortrait(id, 2, null);

return View("Portrait2", model);

}

В построении портрета, попробуем следующую идею композиции: запись будем изображать «столбиком» пар ключ-значение, значение прямой ссылки будем раскрывать также столбиком, но уже в заданной ячейке. Также будем поступать с обратными ссылками. Ключи (имена стрелок) для прямой и обратной ссылки будем выделять цветом.

У нас рекуррентно строится портрет расширенной записи. Поэтому есть смысл действие по построению записи «запроцедурить». В ASP.NET можно определять методы прямо в тексте .cshtml, естественно, в определяющей части. К сожалению, я не знаю способа использования razor-html-разметки в определяемых процедурах (наверное, такой способ есть, по крайней мере, раньше в ASPX он был и я им пользовался). Поступим следующим образом: будем формировать html в виде XML-объекта, потом превратим его в текст, потом конвертируем в разметку через @Html.Raw(text). Привожу вьюер полностью:

@using System.Xml.Linq

@using RDFEngine

@model RDFEngine.RRecord

@{

var m = Model;

Func<RDFEngine.RRecord, XElement> MkS = null;

MkS = rec =>

{

return new XElement("table",

new XElement("tr",

new XElement("td", new XAttribute("colspan", 3),

new XElement("span", rec.Tp + " ", new XElement("a", new XAttribute("href", "?id=" + rec.Id), "@")))),

rec.Props.Select(p =>

{

if (p is RField)

{

return new XElement("tr",

new XElement("td", p.Prop),

new XElement("td", ((RField)p).Value),

new XElement("td"));

}

else if (p is RDirect)

{

return new XElement("tr",

new XElement("td", p.Prop),

new XElement("td", MkS(((RDirect)p).DRec)),

new XElement("td"));

}

else if (p is RInverse)

{

return new XElement("tr",

new XElement("td", p.Prop),

new XElement("td", MkS(((RInverse)p).IRec)),

new XElement("td"));

}

return null;

}),

null);

};

}

@{

<div>

@Html.Raw(MkS(m).ToString())

</div>

}

В итоге, мы получаем что-то вроде:

|  |  |
| --- | --- |
| person [@](https://localhost:44305/Home/Portrait2?id=p45) | |
| name | и45 |
| age | 23 |
| reflected | |  |  | | --- | --- | | reflection [@](https://localhost:44305/Home/Portrait2?id=r92) | | | indoc | |  |  | | --- | --- | | photo [@](https://localhost:44305/Home/Portrait2?id=f106) | | | name | dsp106 | | |
| reflected | |  |  | | --- | --- | | reflection [@](https://localhost:44305/Home/Portrait2?id=r109) | | | indoc | |  |  | | --- | --- | | photo [@](https://localhost:44305/Home/Portrait2?id=f61) | | | name | dsp61 | | |
| reflected | |  |  | | --- | --- | | reflection [@](https://localhost:44305/Home/Portrait2?id=r133) | | | indoc | |  |  | | --- | --- | | photo [@](https://localhost:44305/Home/Portrait2?id=f46) | | | name | dsp46 | | |
| reflected | |  |  | | --- | --- | | reflection [@](https://localhost:44305/Home/Portrait2?id=r144) | | | indoc | |  |  | | --- | --- | | photo [@](https://localhost:44305/Home/Portrait2?id=f13) | | | name | dsp13 | | |
| reflected | |  |  | | --- | --- | | reflection [@](https://localhost:44305/Home/Portrait2?id=r253) | | | indoc | |  |  | | --- | --- | | photo [@](https://localhost:44305/Home/Portrait2?id=f25) | | | name | dsp25 | | |
| reflected | |  |  | | --- | --- | | reflection [@](https://localhost:44305/Home/Portrait2?id=r472) | | | indoc | |  |  | | --- | --- | | photo [@](https://localhost:44305/Home/Portrait2?id=f181) | | | name | dsp181 | | |

Снова отметим, что портрет сущности носит универсальный характер и не зависит от применяемой системы структуризации или онтологии.

## Формирование проекций RDF-графа

Сформируем еще одну специальную проекцию расширенной записи, основанную на табличном представлении записей. Идея заключается в том, чтобы поля и прямые ссылки записи поместить в таблицу (из одного рядка данных), а обратные ссылки сгруппировать по обратному предикату и типу, а далее, остаются записи определенного типа такие, что их можно уложить в единую таблицу по предыдущему принципу. Для понятности, на примере покажем закладываемые идеи. Пусть в базе данных есть фрагмент, похожий на уже рассмотренные фрагменты. В организации НГУ работают персоны Иванов и Петров различных годов рождения, см. Рис 5. Требуется сформировать портрет сущности «НГУ».

Рис 5. Пример графа RDF с одной организацией, двумя персонами и двумя отношениями

Для данного фрагмента дадим его представление в XML-виде:

<?xml version="1.0" encoding="utf-8"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<person rdf:about="p3817">

<name xml:lang="ru">Иванов</name>

<birth>1988</birth>

</person>

<person rdf:about="p3818">

<name xml:lang="ru">Петров</name>

<birth>1999</birth>

</person>

<org-sys rdf:about="o19302">

<name>НГУ</name>

</org-sys>

<participation rdf:about="r1111">

<participant rdf:resource="p3817" />

<in-org rdf:resource="o19302" />

<role>профессор</role>

</participation>

<participation rdf:about="r1112">

<participant rdf:resource="p3818" />

<in-org rdf:resource="o19302" />

<role>ассистент</role>

</participation>

</rdf:RDF>

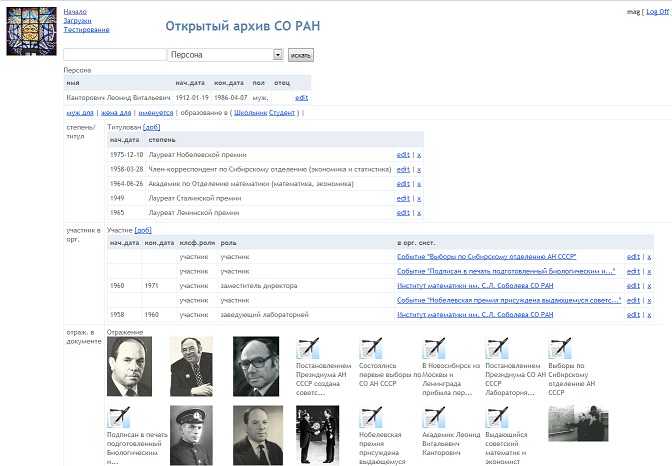
По сравнению с рисунком, здесь добавлены значения полей role в участии персон в организационной системе. Рассмотрим получающийся портрет для сущности НГУ. Портрет второго уровня сущности o19302 (НГУ) «захватит» все элементы фрагмента, а его раскладка в таблицы средствами Portrait2 может выглядеть следующим образом:

|  |  |
| --- | --- |
| org-sys [@](https://localhost:44305/Home/Portrait2?id=o19302) | |
| name | НГУ |
| in-org | |  |  | | --- | --- | | participation [@](https://localhost:44305/Home/Portrait2?id=r1111) | | | participant | |  |  | | --- | --- | | person [@](https://localhost:44305/Home/Portrait2?id=p3817) | | | name | Иванов | | birth | 1988 | | | Role | Профессор | |
| in-org | |  |  | | --- | --- | | participation [@](https://localhost:44305/Home/Portrait2?id=r1112) | | | participant | |  |  | | --- | --- | | person [@](https://localhost:44305/Home/Portrait2?id=p3818) | | | name | Петров | | birth | 1999 | | | role | ассистент | |

В чем недостатки развертки портрета, реализованной в Portrait2? Главные:

* не переведены служебные слова (мы знаем как это преодолеть)
* общая компоновка не является интуитивно понятной
* не достаточно экономно используется плоскость экрана – композиция тяготеет к вертикальному узкому построению.

Рассмотрим еще одну форму представления, которая, как кажется, больше соответствует интуитивно понятному размещению информации о портрете, приведем ее пример:

Рис. 6. Форма развертки портрета с горизонтальным размещением полей записей и списков полей записей

Можно не обращать внимания на последнюю колонку в таблицах, которая выделена для перехода к редактированию записи или ее уничтожению edit | x и другие элементы управления редактированием. В остальном, запись построена весьма логично – в виде полей и гиперссылок. Гиперссылками оформляются прямые ссылки из записи на другие элементы. В качестве текста гиперссылки используется поле name соответствующего элемента. Попробуем воспроизвести идею в варианте визуализации Portrait3.

Внимательное изучение картинки показывает, что новым свойством построения является упорядоченность полей и ссылок в отображении записи. Значения этих полей располагаются «под» надписями, являющимися общими для группы таблиц. Также можно заметить, что упорядоченность можно связать с типом сущности. Действительно, для «главной» таблицы портрета это просто перечисление полей и ссылок, которые могут появиться при его формировании в данном месте. Для вспомогательных таблиц, формирующихся по цепочке обратное свойство – группирующий тип – запись, это также «работает», но есть нюанс. Он заключается в том, что в наборе ссылок для записи должны присутствовать все исходящие дуги, а формирование вспомогательной таблицы осуществляется без одной – той по которой мы «пришли» к этой записи через обратную ссылку.

Подведу итог анализа. Для каждого типа сущностей нужно сформировать упорядоченный набор полей и прямых ссылок. Таким образом, мы пришли к необходимости создать более точную спецификацию данных, т.е. онтологию. И использовать ее для создания универсального табличного интерфейса.

## Создание универсального табличного интерфейса

Этот проект будет существенно более сложный, чем предыдущие. Для него рекомендую создать в нашем решении новый проект, в нем мы частично повторим уже пройденные моменты, но многое будет в новинку. Причем, с довольно сложной логикой построения и реализации. Напрягитесь. Новый проект, по-прежнему будет создаваться в технологии ASP.NET Core MVC. Назовем его ViewHTab. На что мы будем концентрировать внимание? Мы еще раз создадим модельные данные формата RDF, используем движок REngine, воспользуемся им для выборки из базы данных расширенной записи, повторим контроллер и вьюер Portrait. Это все мы уже делали, но важно повторить пройденное и сконцентрируемся на новые моменты, создавая интерфейс Portrait3. Для этого мы создадим объектную онтологию ROntology, выделим сложную обработку расширенной записи в отдельную модель и уже специально построенную модель мы визуализируем вьюером Portrait3.cshtml.

Быстро повторим то, что мы уже делали, доведя проект до работы с контроллером/вьюером Portrait. В начальном проекте очищаем Views/Shared/\_Layout.cshtml и Views/Home/Index.cshtml, проверяем работу. Добавляем в зависимости проект RDFEngine. Движок у нас будет в единственном числе, для большей доступности из разных мест проекта, создадим класс Infobase и поместим движок туда в виде статического объекта public static REngine engine; В Startup добавим создание объекта класса REngine и загрузку движка сгенерированной базой данных, напомню, что это делается просто:

Infobase.engine = new RDFEngine.REngine();

Infobase.engine.Load(RDFEngine.PhototekaGenerator.Generate(1000));

Далее, в разделе Home делаем контроллер Portrait и визуализатор к нему. Эти решения можно перенести из предыдущего проекта.

Теперь мы пойдем дальше. Рассмотрим общую схему «превращения» базы данных RDF в картинку.

Построение HTML

Спецификация данных

Создание модели

База данных

Объект, порождаемый базой данных по запросу, это расширенная запись, причем в иерархическом представлении, когда все необходимые части для портрета в объекте присутствуют и не надо выполнять дополнительные запросы. Создание модели – наиболее критическая часть для данного раздела. Как мы уже выяснили при анализе скриншота желаемого представления, из онтологии или спецификации берется порядок изложения полей и прямых ссылок записи и естественно-языковые названия классов, свойств, полей.

Сначала займемся моделью. Переводом R-записи в иерархическое представление на два уровня, у нас занималась функция BuildPortrait с параметром level=2. Эта функция была написана прямо в контроллере. В силу важности более глубокого «заглядывания» в данные, перенесем функции BuPo и BuildPortrai в класс REngine.

Теперь займемся созданием формирователя модели. Начнем с проектирования модели. Еще раз взглянув на картинку-образец, мы видим, что структура, адекватная html-образу портрета, будет состоять из идентификатора фокусной сущности, идентификатора типа сущности и набора элементов RProperty, соответствующих полям записи и конструкциям RDirect на месте прямых ссылок. Это все как обычно в расширенной записи. А вот обратные ссылки надо группировать по имени свойства и по имени типа ссылающейся записи. Введем эти свойства и эти группировки в модель P3Model:

/// Вспомогательный класс - группировка списков обратных свойств

public class InversePropType

{

public string Prop;

public InverseType[] lists;

}

/// Вспомогательный класс - группировка списков по типам ссылающихся записей

public class InverseType

{

public string Tp;

public RProperty[] list;

}

/// Класс модели

public class P3Model

{

public string Id, Tp;

public RProperty[] row;

public InversePropType[] inv;

}

Модели в текущей версии ASP.NET Core MVC принято размещать в разделе Models, так что поместим эти описания в него с именем файла P3Model.cs.

Теперь нам надо создать преобразование расширенной записи класса RRecord в модель класса P3Model. Задача не выглядит сложной и рекомендуем попробовать реализовать ее самостоятельно. Если не получается, то примитивное решение можно найти в демонстрационной процедуре Build0. Почему 0? Это потому, что она не решает полностью поставленных задач. Но с ней уже можно работать.

Под конец, напишем контроллер и вьюер Portrait3.

// Контроллер

public IActionResult Portrait3(string id)

{

var erec = Infobase.engine.BuildPortrait(id);

P3Model model = (new P3Model()).Build0(erec);

return View("Portrait3", model);

}

@\* Вьюер \*@

@using RDFEngine

@model P3Model

@{

var m = Model;

Func<RRecord, string> GetName = (RRecord rec) =>

{

return ((RField)rec.Props.FirstOrDefault(p => p is RField && p.Prop == "name"))?.Value;

};

}

<table border="1">

<tr>

<td colspan="@m.row.Length">@m.Tp @m.Id</td>

</tr>

<tr>

@foreach (var c in m.row)

{

<td>@c.Prop</td>

}

</tr>

<tr>

@foreach (var c in m.row)

{

if (c is RField)

{

<td>@(((RField)c).Value)</td>

}

else if (c is RDirect)

{

var rr = ((RDirect)c).DRec;

<td><a href='?id=@(rr.Id)'>@(GetName(rr))</a></td>

}

}

</tr>

</table>

@foreach (var p in m.inv)

{

foreach (var t in p.lists)

{

RRecord defrow = t.list[0];

<table border="1">

<tr>

<td colspan="@(t.list.Length)">

<span>@(p.Prop)</span>

<span>@(defrow.Tp)</span>

</td>

</tr>

<tr>

@foreach (var c in defrow.Props)

{

<td>@(c.Prop)</td>

}

</tr>

@foreach (var r in t.list)

{

<tr>

@foreach (var c in r.Props)

{

if (c is RField)

{

<td>@(((RField)c).Value)</td>

}

else if (c is RDirect)

{

var rr = ((RDirect)c).DRec;

<td><a href='?id=@(rr.Id)'>@(GetName(rr))</a></td>

}

else

{

<td>???</td>

}

}

</tr>

}

</table>

}

}

Код достаточно простой и легко пишется. Но… неправильный! А неправильный он в той части, которая не приведена, в методе public P3Model Build0(RRecord erec) для класса P3Model. Дело в том, что таблицы строятся в порядке появления полей или прямых ссылок в записях. А поля и ссылки могут в записях появляться в произвольном порядке, какие-то поля или какие-то ссылки могут отсутствовать. В итоге, построение визуальных таблиц обратных ссылок надо бы производить по-другому. Но тогда идея компактного интуитивно понятного представления была бы нарушена. Что же делать? Внимательный анализ показывает, что порядок полей и ссылок для записей должен быть определен внешним образом. Один из таких способов – определять его в формальной спецификации данных, т.е. в онтологии.

Ранее у нас уже использовалась простая онтология, в которой перечислялись имена классов сущностей, имена свойств (стрелок) и относящиеся к ним «человеческие» названия классов и свойств – «метки». Был использован формализм OWL, рекомендуемый консорциумом WWW [www.w3.org]. Рекомендуемый, не значит обязательный, сейчас мы создадим объектное представление онтологии и расширим его нужными средствами. Напомним, как это выглядело:

<owl:Ontology

xmlns:owl ="http://www.w3.org/2002/07/owl#"

xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<owl:Class rdf:about="person">

<rdfs:label xml:lang="ru">Персона</rdfs:label>

</owl:Class>

<owl:Class rdf:about="photo">

<rdfs:label xml:lang="ru">Фотография</rdfs:label>

</owl:Class>

…

Это же можно сделать средствами объектного представления записи, в нашем случае через объекты класса RRecord и классов группы RProperty. Cоздадим массив записей RRecord причем по тем же правилам, что и для значений при переводе записей RDF в объектное представление. Но теперь мы добавим в набор свойств записи ссылки RLink двух специальных видов, со свойствами Prop DatatypeProperty и ObjectProperty. Первый вариант будет определять наличие поля в записи, а второй – наличие прямой ссылки с соответствующим идентификатором предиката (стрелки), см. пример:

// ROntology - объектная онтология

RRecord[] rontology = new RRecord[]

{

new RRecord

{

Id = "person",

Tp = "Class",

Props = new RProperty[]

{

new RField { Prop = "Label", Value = "Персона" }

, new RLink { Prop = "DatatypeProperty", Resource = "name"}

, new RLink { Prop = "ObjectProperty", Resource = "father"}

}

}

, new RRecord

{

Id = "name",

Tp = "DatatypeProperty",

Props = new RProperty[] { new RField { Prop = "Label", Value = "Персона" }}

}

};

Теперь при развертке (проекции) записи какого-то класса (типа), мы берем из R-онтологии определение этого класса и формируем описанный в определении список полей и ссылок. При этом, для каждого такого поля или ссылки, мы можем заглянуть в его определение через эту же онтологию и в нем найти метку и, возможно, другие атрибуты «стрелки».

Оптимизация использования такой онтологии может потребовать добавление дополнительных структур. Например, целесообразно иметь хеш-словарь для элементов массива онтологий. Это для быстрого доступа к определению понятия по его имени. Другой момент, требующий ускорения – нахождение позиции (номера столбца) свойства в записи по его имени. Это потому что RDF не предписывает специальной упорядоченности полей в записи значения. Все эти оптимизации можно свести к дополнительному словарю и массиву словарей:

Вот такая модернизированная (хотя и неполная) онтология представлена классом ROntology.cs, помещенным в пакет RDFEngine. Не уверен, что читатель уже готов к проектированию онтологии, да еще в объектном представлении, так что принимайте код так как он есть и попробуйте им воспользоваться.

## Изучаем Blazor

Подозреваю, что читатели не слишком разобрались в сложном варианте построения портрета, представленном в предыдущем разделе. Сложности технологии, структурных и алгоритмических построений усилили друг друга и вряд и сформировали у вас ясное понимание. Поэтому придется разобрать все эти вопросы еще раз, но по отдельности и в более простых вариантах. Начнем с технологии. Мы использовали ASP.NET в варианте MVC. Сейчас мы познакомимся с другой веточкой ASP.NET – технологией Blazor.

В чем суть технологии клиент-сервер, реализованной в классическом варианте Model-View-Controller? Суть в том, что запрос (query), пришедший от браузера, получает контроллер, проанализировав его, делает выборку из данных и формирует модель данных, соответствующую желаемой форме ответа. В конце этой цепочки вызывается построитель (View) новой страницы, отсылаемой браузеру в качестве ответа (response). Проблема технологии в том, что объекты программы и визуального интерфейса проходят преобразование в разных модулях, проходя через сеть и форму отображения. При этом одному циклу «ввод/редактирование данных – request – работа с данными – response» соответствуют разные фрагменты в разных модулях на разных языках, иногда на разных компьютерах. А еще в цикле может присутствовать клиентская обработка (на еще одном языке программирования JavaScript). И чем логичнее и удобнее создается интерфейс, тем сложнее программирование для разработчика.

Существует много технологических решений и библиотек, делающие попытки упростить клиент-серверное программирование, технология Blazor – одна из них. Главная особенность этой технологии в том, что все этапы клиент-серверного взаимодействия «закрываются» единой средой программирования и единым языком программирования C#. Преодолеть необходимость использования HTML-разметки не удается, но происходит сильная «сшивка» C# и HTML через Razor. Технология довольно объемная, быстро ее не изучить, но познакомиться – можно. Существуют разные пособия по Блазору, но, как всегда, важно изучать официальную документацию <https://docs.microsoft.com/ru-ru/aspnet/core/blazor/> , с помощью которой можно понять что и как надо делать и что при этом происходит.

Во всех пособиях рекомендуют начинать с шаблонного решения. В VisualStudio создайте проект по шаблону Blazor Server, назовем его для определенности идентификатором BlazorServer. Запустите полученное приложение, посмотрите как работают вкладки. Запуск проекта осуществляется стандартными средствами (F5, <cntrl>F5). Важно сопоставить имеющиеся под единым меню одностраничные приложения Index, Counter, Fetch Data с их описаниями в файлах Pages/Index.razor, Pages/Counter.razor, Pages/FetchData.razor. Это и есть код приложений. Посмотрите на код счетчика. Там все более или менее понятно. Первая строчка – имя приложения, соответственно обращение к нему со стороны клиента будет что-нибудь вроде <https://localhost:44320/counter>

На пятой строке кода есть вставка значения переменной currentCount в html код. В 7-й строку описана кнопка, нажатие на которую вызовет запуск метода IncrementCount. Этот метод, вместе с переменной currentCout, имеется в разделе кода. После исполнения метода значение переменной увеличится на единичку. Изменение значения переменной вызовет перерисовку страницы. Просто и наглядно.

Оставим страницу со счетчиком в покое, будем экспериментировать на странице Index. Она вызывается по умолчанию, поскольку так сделана настройка. Поэкспериментируйте с индексной страницей. Обратите внимание, что если запустить приложение в стандартном (не отладочном режиме) напр. через <cntrl>F5, то синхронизация относительно изменений на странице будет происходить автоматически. Напр. уберите элемент SurveyPrompt и сделайте изменения в текстах и элементах начала. По сохранению изменений в файле, посмотрите на страницу в браузере и Вы увидите процесс синхронизации и появление нового варианта страницы.

Не «растекаясь мыслями по вариантам», будем изучать как выполнять изменения в данных. Сначала создадим данные. Пусть это будет простая запись, состоящая из строкового поля Name и числового поля Age. Тип записи введем в разделе кода страницы, там же определим экземпляр записи и зададим ему значение:

@code{

class Perso

{

public string Name { get; set; },

public int Age { get; set; }

}

Perso sample = new Perso { Name = "Иванов Иван Иванович", Age = 39 };

}

В визуализационную часть страницы поместим выдачу значения через выдачу его полей:

<div>Name: @sample.Name</div>

<div>Age: @sample.Age</div>

Это похоже на то, как было со счетчиком. Проверяем работу, если не ошиблись, то увидите то, что ожидается.

Следующий шаг: мы хотим подвергнуть экземпляр персоны редактированию, т.е. возможности изменения значения полей. Для этого добавим поля ввода данных.

<div> Имя: <input **@bind**="sample.Name" /> </div>

<div> Возраст: <input **@bind**="sample.Age" /> </div>

Принципиальным здесь является использование связывания bind элемента ввода и поля данных.

При испытании, Вы увидите, что формы ввода синхронизированы с полями переменной sample. Это означает, что изменение значения в поле ввода автоматически влечет изменение значения полей переменной и, наоборот, изменение значение переменной проявляется в выводе и вводе. Для демонстрации последнего, достаточно добавить код:

<div> <button **@onclick**="Clear" > Clear person </button></div>

@code{

void Clear() { sample.Name = ""; sample.Age = -1; }

}

И понажимать кнопки.

В дальнейшем, нам понадобится осуществлять байндинг и к элементам массива. Но возможно ли это? Заменим наше простое приложение Index на более сложное. Выбросим все, за исключение именующей строки (строка 1). Сформируем массив данных в виде двумерного массива строк. Цель приложения – редактирование отдельных полей массива или рядков массива. Массив данных (поместим в раздел кода):

string[][] rowsToEdit = new string[][]

{

new string[] { "colName1", "colName2", "colName3" },

new string[] { "colCont1a", "colCont2a", "colCont3a" },

new string[] { "colCont1b", "colCont2b", "colCont3b" },

new string[] { "colCont1c", "colCont2c", "colCont3c" }

};

Подразумеваем, что первый рядок – это рядок названий колонок, а следующие рядки – значения ячеек данных. Чтобы все делалось по порядку, сделаем отрисовщик этих данных:

<table border="1">

<tr>

@for (int j = 0; j < rowsToEdit[0].Length; j++)

{

<th>@rowsToEdit[0][j]</th>

}

</tr>

@for (int i = 1; i < rowsToEdit.Length; i++)

{

<tr>

@for (int j = 0; j < rowsToEdit[i].Length; j++)

{

<td>@rowsToEdit[i][j]</td>

}

</tr>

}

</table>

Обратите внимание на то, что первый рядок строится отдельно в виде заголовков и что цикл по рядкам далее начинается с 1.

Следующий этап – формирование кнопок для перевода рядка в режим редактирования. Мы просто добавим по одной колонке в каждый рядок и в новые ячейка поместим кнопки с некоторой командой. Команда будет EditRow(int i) – «перевести рядок i в режим редактирования». Эту команду надо определить. В коде вводим целую переменную rowi и метод будет присваивать текущее значение этой переменной:

<table border="1">

<tr>

@for (int j = 0; j < rowsToEdit[0].Length; j++)

{

<th>@rowsToEdit[0][j]</th>

}

<th></th>

</tr>

@for (int i = 1; i < rowsToEdit.Length; i++)

{

int ii = i;

<tr>

@for (int j = 0; j < rowsToEdit[ii].Length; j++)

{

<td>@rowsToEdit[ii][j]</td>

}

<td><a href="javascript:void(0)" **@onclick**="@(e =>EditRow(ii))">edit</a></td>

</tr>

}

</table>

@code{

int rowi = -1;

void EditRow(int i) { rowi = i; }

}

Можно убедиться, что приложение «дышит». По крайней мере, в состоянии «отзываться» на нажатие кнопки (гиперссылки) edit. Обратим Ваше внимание на странный оператор int ii = i;. Тут дело «темное», если этого не сделать, а использовать I, то атрибут **@onclick**="@(e =>EditRow(ii))" будет работать неправильно. Видимо, это ошибка реализации, придется это обстоятельство учитывать, пока ошибку не исправят.

Что надо сделать дальше? Надо сделать для изображения рядка два варианта: один – как сейчас, другой, в случае совпадения номера рядка с rowi, - вариант ввода данных. То есть, вместо нынешнего <tr>…</tr> в цикле по рядкам, нужно сделать разные варианты:

if (ii != rowi)

{

<tr>

@for (int j = 0; j < rowsToEdit[ii].Length; j++)

{

<td>@rowsToEdit[ii][j]</td>

}

<td><a href="javascript:void(0)" **@onclick**="@(e =>EditRow(ii))">edit</a></td>

</tr>

}

else

{

<tr>

@for (int j = 0; j < rowsToEdit[ii].Length; j++)

{

int jj = j;

<td><input **@bind**="@rowsToEdit[ii][jj]"/></td>

}

<td><input type="submit" **@onclick**="@(e =>Ok())" value="Ok"/></td>

</tr>

}

Обратим внимание на то, что используется (для разнообразия) другой вид кнопки. Также, по причинам, описанным ранее, вводится переменная jj.

## Онтология и онто-шаблоны

Мы будем пользоваться почти стандартным формализмом задания онтологии []. Что такое «почти» будет ясно позже и не составляет чего-то очень принципиального.

## Редактирование портрета

Вернемся к выполнению проекта создания универсального вьюера/редактора. Итак, постараемся создать редактор через создание вьюера с переводом выделенной записи в режим редактирования и организацией редактирования. Будем использовать Blazor.

Сначала повторим решение ViewHTab в обстановке созданного Blazor-проекта BlazorServer. Поскольку Blazor и ASP.NET MVC очень близкие «родственники», это сделать довольно просто. Возьмем из ViewHTab все, что нам надо и перенесем в новый Blazor-подпроект. Сначала очистим Index.razor. Чтобы не пропало наработанное, переименуем Index.razor в SimpleEditor.razor и добавим вход в навигационном меню. Проверим, что страница работает, создадим новый пустой вариант Index.razor. Перенесем Infobase.cs и его инициирование в Startup’е.

Теперь действия контроллера, формирователя модели и вьюера объединим в Index.razor. Все начинается с идентификатора сущности. Потом вычисляется модель, потом модель разрисовывается. Добавим лишь то, с чего удобнее начать – поиск сущности по имени или части имени.

Если Вы были внимательны и аккуратны, у Вас обязательно получится собрать предыдущие достижения воедино. Но, на всякий случай, я оставлю текущее состояние проекта во вкладке Viewer.

Для менее внимательных, а также тех, кто считает, что «повторение – мать учения», объясню подробнее. При этом, я не буду использовать ряд уже пройденных моментов чтобы проект стал совсем простым. Я назвал проект MagBlazor. Ставим исполняемым проектом этот новый и убеждаемся, что Index, Counter и FetchData работают.

Сначала спроектируем объект, который будет нам статически предоставлять то, что требуется в разных местах (полезных страниц может быть несколько). Назовем его по-прежнему Infobase и поместим в него, для начала только движок. Придется также добавить ссылку на проект RDFEngine. В прошлый раз мы изрядно «намусорили» в ни в чем не повинном Startup, эти полезные для отладки данные мы перенесем в движок. Теперь и вставка в Main (файл Program.cs) станут очень простыми:

// файл Infobase.cs

namespace MagWABlazor

{

public class Infobase

{

public static RDFEngine.IEngine engine = null;

public static RDFEngine.ROntology rontology = null;

}

}

// Вставка в Main

// Инициирование объектов Infobase

Infobase.engine = new RDFEngine.REngine();

((RDFEngine.REngine)Infobase.engine).Load();

Infobase.engine.Build();

Infobase.rontology = new RDFEngine.ROntology();

Надо испытать работу базы данных. Для этого, в индексной странице поместим форму поискового запроса и список результатов запроса.

@page "/"

@using RDFEngine

<h1>страница /Home</h1>

<div>

<input **@bind**="searchsample" />

</div>

@if (!string.IsNullOrEmpty(searchsample))

{

var query = Infobase.engine.RSearch(searchsample);

foreach (RRecord rec in query)

{

<div>

<span>@rec.Tp</span> &nbsp;

<a href="javascript:void(0)" **@onclick**="@(e =>BuildPortrait(rec.Id))">@rec.GetName()</a>

</div>

}

}

@code{

private string searchsample = "";

private void BuildPortrait(string id)

{

}

}

Испытаем код, а если Вы его не писали, внимательно на него посмотрим. Что здесь написано? Есть поле ввода, связанное (bind) с переменной searchsample. И если searchsample не пустая, то делаем вычисление RSearch, а его результаты визуализируем построчно. Визуализируется тип найденного айтема и его имя, причем на имени «навешена» гиперссылка, запускающая процедуру построения портрета. Пока процедура – пустая. Остальное – работает. Только имейте ввиду, что задавить нужно начальную часть имени имеющихся в базе данных объектов. Тестовую базу данных можно посмотреть в файле REngine и выяснить, что есть только три именованных объекта и если задать образец “и”, то будет найден Иванов.

Расширим страницу Index.razor новыми элементами обработки. Идея заключается в том, что если не задан поисковый образец, то может быть задана модель фокусного объекта. Эта модель «разрисовывается» под поисковой частью. Для лучшего понимания, пойдем шагами. Сначала введем модель типа RRecord, а визуализируем ее в виде строки model.ToString(), это строку выдадим на страницу через элемент <pre>…</pre>. Итак, добавим в код модель и в коде же породим модель в BuildPortrait:

private RRecord model = null;

private void BuildPortrait(string id)

{

model = Infobase.engine.GetRRecord(id);

if (model != null) searchsample = null;

}

А построитель портрета обусловим тем, что модель ненулевая

// После поискового сегмента

else if (model != null)

{

<pre>@model.ToString()</pre>

}

Это была портретная проба. На самом деле, в нашем решении портрет будет устроен сложнее. Ранее, мы пытались понять его конструкцию, в этот раз оставим такие попытки и просто используем класс P3Model и вспомогательные классы определенные в модели P3Model. Для тех, кто выполняет проект самостоятельно, указываю, что надо создать систему определений, аналогичную P3Model.cs, используемую в более ранних проектах. Минимальными изменениями, т.е. изменением типа модели и метода вычисления модели:

private P3Model model = null;

private void BuildPortrait(string id)

{

RRecord rec = Infobase.engine.GetRRecord(id);

model = new P3Model().Build(rec);

if (model != null) searchsample = null;

}

Теперь снова создадим построитель визуального образа окрестности RDF-графа, выделенного и сформатированного P3Model. Если кратко повторить структуру реализуемой конструкции, то это горизонтальные таблицы с одним или несколькими рядками значений, соответствующих полям и прямым ссылкам записей. Сначала идет опорная запись – это собственно запись раскрываемого айтема. Потом идет группирование по обратным свойствам и по типам обратных айтемов и изображение этих самых записей обратных отношений.

|  |  |  |  |
| --- | --- | --- | --- |
| in-org participation | | | |
| role | from-date | participant |  |
| профессор |  | [Иванов](javascript:void(0)) | [ред](javascript:void(0)) |
| ассистент | 2008 | [Петров](javascript:void(0)) | [ред](javascript:void(0)) |

Отдельная таблица состоит из первого рядка, в котором для основной записи указан тип записи и идентификатор записи, а в «обратной» таблице (как на рисунке), указано имя обратного отношения и тип записи. Следующий рядок именует колонки и формируется из имен свойств полей и прямых ссылок. Далее идет некоторое количество рядков со значениями (обратных) записей, имеющих тот же контекст по обратному отношению и типу. Понятно, что для основной (опорной) таблицы, имеется только один рядок со значениями. Зато для обратных таблиц изображаются все колонки (поля и прямые ссылки) согласно онтологии.

Еще важным отличием обратных таблиц является то, что в них отсутствует одна колонка. Это обратное отношение, ведущее от обратного элемента к опорному элементу. Оно не информативно, но легко восстанавливается в данных, поскольку зафиксировано имя обратного отношения.

Пожалуйста, постарайтесь разобраться с логиком и кодом получающихся построений – это важно для продвижения дальше по материалу.

Приведу полный текст построителя таблиц, поскольку он дальше будет искажаться для задания редактирующих областей и полей.

var m = model;

<table border="1">

<tr>

<td colspan="@(m.row.Length+1)">@m.Tp @m.Id</td>

</tr>

<tr>

@foreach (var c in m.row)

{

<td>@c.Prop</td>

}

<td></td>

</tr>

<tr>

@foreach (var c in m.row)

{

if (c is RField)

{

<td>@(((RField)c).Value)</td>

}

else if (c is RDirect)

{

var rr = ((RDirect)c).DRec;

if (rr != null)

{

<td><a href="javascript:void(0)" **@onclick**="@(e => BuildPortrait(rr.Id))">@(rr.GetName())</a></td>

}

else

{

<td></td>

}

}

else

{

<td>???</td>

}

}

<td>

<a href="javascript:void(0)" **@onclick**="@(e =>SetEditItem(model.Id, null))">ред</a>

</td>

</tr>

</table>

@foreach (var p in m.inv)

{

foreach (var t in p.lists)

{

RRecord defrow = t.list[0];

<table border="1">

<tr>

<td colspan="@(defrow.Props.Length)">

<span>@(p.Prop)</span>

<span>@(defrow.Tp)</span>

</td>

</tr>

<tr>

@foreach (var c in defrow.Props)

{

if (c.Prop == p.Prop) { continue; }

<td>@(c.Prop)</td>

}

<td></td>

</tr>

@foreach (var r in t.list)

{

<tr>

@foreach (var c in r.Props)

{

if (c.Prop == p.Prop) { continue; }

if (c is RField)

{

<td>@(((RField)c).Value)</td>

}

else if (c is RDirect && ((RDirect)c).DRec != null)

{

var rr = ((RDirect)c).DRec;

if (rr != null)

{

<td><a href="javascript:void(0)" **@onclick**="@(e => BuildPortrait(rr.Id))">@(rr.GetName())</a></td>

}

else

{

<td></td>

}

}

else

{

<td>???</td>

}

}

<td><a href="javascript:void(0)" **@onclick**="@(e =>SetEditItem(r.Id, p.Prop))">ред</a></td>

</tr>

}

</table>

}

}

Наверное, было бы хорошо запроцедурить построение таблицы, но пока мы не будем отвлекаться. Так что придется «потерпеть» повтор.

Скорее всего, среда разработки будет «ругаться» на отсутствие определения метода SetEditItem. В этом месте, я бы посоветовал пока поставить «заглушку»: private void SetEdit(string eid, string forbidden) { }

Если Вы сделали все правильно, то страница будет «работать»: искать айтем по имени, изображать айтем, обеспечивать переходы по гиперссылкам на другие айтемы (напр. НГУ). Также в портрете есть «кнопки» для перехода к редактированию. Посмотрите еще раз где и как эти ссылки (ред) поставлены и Вы можете увидеть, что мы хотим подвергать редактированию имеющиеся в портрете записи. Это либо опорную запись, либо какую-нибудь из обратных записей. По нажатию на гиперссылку

**onclick**="@(e =>SetEditItem(r.Id, p.Prop))"

Будет запускаться процедура SetEdit с передачей ей идентификатора записи и имени запрещенного свойства, т.е. свойства, которое ведет на опорный айтем. Для опорной таблицы, второй параметр будет null.

Итак, мы подошли к редактированию. Предлагается следующий подход. В портрете редактируется единственная запись или не редактируется ни одной. Этот подход постараемся зафиксировать тем введем переменную страницы RRecord erecord. Если эта переменная null, то ничего не редактируется, если не null, значит это запись, которая подвергается редактированию. Если это обратная запись, то еще одна переменная страницы string forbidden определяет имя свойства, связывающего редактируемую запись с опорной. Если forbidden == null, то редактируется опорная запись. Идентификатор редактируемой запись erecord.Id. В случае редактирования опорной записи, этот идентификатор должен совпадать с model.Id.

Запись erecord должна быть копией записи базы данных. Только если редактирование будет утверждено, будет внесено новое значение элемента и скорректированы хеш-таблицы (см. далее). Соответственно, никакая кнопка не должна работать пока идет редактирование. Это касается и гиперссылок перехода к другим айтемам.

Образ редактируемой записи (в данном подходе) предлагается поместить в тот же рядок, а котором находился образ оригинала записи. Сделаем указанные изменения. Мы пока не будем редактировать, а лишь воспроизведем редактирующую обстановку.

Введем указанные переменные, наметим переход в режим редактирования:

// редактируемая запись и идентификатор обратного свойства

private RRecord erecord = null;

private string forbidden;

private void SetEditItem(string eid, string forbidden)

{

erecord = new RRecord() { Id = eid };

this.forbidden = forbidden;

}

Сделаем альтернативный вариант рисования рядка. В рядок мы поместим одну ячейку, в которой буду две кнопки ok, cancel. Альтернативный вариант будет рисоваться если идентификатор записи совпадет с erecord.Id. Вот код генерации рядка данных:

@if (erecord != null && m.Id == erecord.Id)

{

<td colspan="@(m.row.Length+1)" style="border:dashed;">Edit Record</td>

}

else

{

// Это код, который был, теперь он используется в варианте «без редактирования»

@foreach (var c in m.row)

{

if (c is RField)

{

<td>@(((RField)c).Value)</td>

}

}

…

}

Обратим внимание на то, что альтернативную ячейку мы оконтуриваем пунктиром. Попробуйте что получится. Теперь пора обеспечить режим редактирования записи. Условием нахождения в режиме редактирования мы установили erecord != null. Добавим в новый код редактирования две кнопки: ok и cancel. И используем условие во всех процедурах обработке onclick. То есть, в существующем методе SetEditItem действие обусловим тем, что erecord == null, а в новых методах Ok и Cancel сбросим erecord в null. Испытаем новшества. Если все сделано правильно, то при нажатии команды редактирования для опорной записи, вместо полей это записи появится пунктирная ячейка с двумя кнопками. Теперь гиперссылки (ведущие к другим айтемам или перевода в режим редактирования) перестанут работать. Но если нажать кнопку Ok или Cancel картинка восстановится. Но если нажать гиперссылку перевода в режим редактирования одну из обратных записей, кнопки не появятся, но блокировка состоится. Это логично, поскольку мы не нарисовали этих кнопок. Воспроизведем адаптированный вариант этого поля, все должно работать правильно.

@if (erecord != null && r.Id == erecord.Id)

{

<td colspan="@(m.row.Length + 1)" style="border:dashed;">

<table border="1">

@\* Здась будут рядки записи, предназначенной для редактирования \*@

<tr>

<td colspan="2">

<button **@onclick**="Ok">ok</button>

<button **@onclick**="Cancel">cancel</button>

</td>

</tr>

</table>

</td>

}

else …

А теперь просто расположим еще одну таблицу в месте, которое помещено комментарием. Эта таблица будет аналогична общему табличному изображению, но будет построена «по вертикали». То есть, будет две колонки, в первой колонке будут названия полей, во второй колонке – значения полей. Количество полей будет тем же, что и в базовом варианте.

@\* Это рядки записи, предназначенной для редактирования \*@

@foreach (var c in m.row)

{

<tr>

<td>@c.Prop</td>

@if (c is RField)

{ <td width="100%">@(((RField)c).Value)</td> }

else if (c is RDirect)

{

var rr = ((RDirect)c).DRec;

if (rr != null)

{

<td><a href="javascript:void(0)" **@onclick**="@(e =>BuildPortrait(rr.Id))">@(rr.GetName())</a></td>

}

else

{

<td></td>

}

}

</tr>

}

После каждой даже небольшой группы изменений, проверяем решение на работоспособность. Это позволяет вовремя заметить ошибку и устранить ее. Еще раз попереживаем, что нет запроцедуривания и скопируем полученную вставку в рядок обратной таблицы. Сделаем необходимые изменения.

Теперь будем реализовывать редактирование выделенной записи. Идея заключается в том, чтобы нажатием на поле значения переводить эту ячейку в режим ввода (элемент <input>), а после ввода или редактирования, убирать этот режим. И соответственно, применим @bind. Это подходит для текстовых полей RField. Что делать с объектными ссылками, подумаем позже. Как всегда, начнем с программирования @onclick, который «навешиваем» на <td></td> . В процедуру обработки события поместим идентификатор свойства этого поля. Получившаяся ячейка для поля:

<td width="100%" **@onclick**="@(e => EditField(c.Prop))">

@if (c.Prop == editfieldprop)

{

<input **@bind**="fieldvalue" />

}

else

{

<span>@(((RField)c).Value)</span>

}

</td>

Наибольшие изменения при переводе в режим редактирования будут в коде. Надо: а) в методе SetEditItem написать код копирования фрагмента модели в erecord; б) в методе EditField зафиксировать в erecord «накопившееся» в fieldvalue значение в указанном свойстве; в) изменить значение редактируемого свойства editfieldprop и редактируемого значения fieldvalue. Возможный код этих изменений:

private void SetEditItem(string eid, string forbidden)

{

if (erecord != null) return; // Не реагировать на гиперссылку если не закончено редактирование какой-то записи

if (eid == model.Id)

{

erecord = new RRecord { Id = model.Id, Tp = model.Tp, Props = model.row.Select(p =>

{

if (p is RField) return new RField { Prop = ((RField)p).Prop, Value = ((RField)p).Value };

else return p;

}).ToArray() };

}

else

{

RRecord re = model.inv

.SelectMany(iv => iv.lists)

.SelectMany(t => t.list)

.FirstOrDefault(r => r.Id == eid);

if (re != null)

erecord = new RRecord

{

Id = re.Id,

Tp = re.Tp,

Props = re.Props.Select(p =>

{

if (p is RField) return new RField { Prop = ((RField)p).Prop, Value = ((RField)p).Value };

else return p;

}).ToArray()

};

else erecord = null;

}

editfieldprop = null;

this.forbidden = forbidden;

}

private string editfieldprop = null;

private string fieldvalue = "Hello!"; // на всякий случай - неправильное значение

private void EditField(string prop)

{

if (editfieldprop != null)

{

var pr = erecord.Props.Where(p => p is RField)

.Cast<RField>()

.FirstOrDefault(f => f.Prop == editfieldprop);

pr.Value = fieldvalue;

}

editfieldprop = prop;

// Перепишем значение в fieldvalue

fieldvalue = erecord.Props.Where(p => p is RField)

.Cast<RField>()

.FirstOrDefault(f => f.Prop == prop)

.Value;

}

Добейтесь работоспособности страницы. Портрет должен переходить в режим редактирования как для опорной таблицы, так и для обратных. После редактирования «наработанные» значения пропадают. Теперь можно заняться фиксацией результата редактирования. Накопленное во временной записи значение надо записать в базу данных. Это может быть опорная запись, но может быть и обратная. В обоих случаях запись состоит из полей и прямых ссылок. И находится она по входу, определяемому идентификатором записи. В принципе, вначале достаточно заменить старое значение на новое. Правда в этом новом значении сначала надо преобразовать прямые RDirect в косвенные RLink. Поскольку запись не исчезает, все ссылки RLink, ведущие к ней также сохранятся. Но вот с прямыми ссылками надо дополнительно поработать. Одна прямая ссылка для обратной записи обязательно сохранится. Это та ссылка, по которой (в обратном направлении) мы пришли к опорной записи. Но могут быть и другие ссылки. Они могут измениться в результате редактирования. То есть, указать на другую запись. Соответственно, изменятся индексы обратных ссылок и для предыдущей записи по ссылке и последующей. Эти изменения можно отследить при фиксации изменений в базе данных.

Прямую ссылку можно разрушить и можно создать. Разрушить прямую ссылку можно нажав на команду возле нее, напр. на крестик. Создать прямую ссылку можно, когда есть пустое поле ссылки и надо отработать onclick.

<a href="javascript:void(0)" **@onclick**="@(e => DeleteLink())">x</a>

По этой команде формируется структура поиска айтема. При этом, мы переходим в режим searchsubitem.

Начну с уничтожения связи. Крестик я сделал. Его обработка вроде совсем простая. Надо убрать значение прямой ссылки. Пока из редактируемого результата, а при фиксации, - из базы данных. Первое действие делается в методе DeleteLink.

private void DeleteLink(string prop)

{

var erec = new RRecord

{

Id = erecord.Id,

Tp = erecord.Tp,

Props = erecord.Props.Select(p => p.Prop != prop ? p : new RDirect { Prop = prop, DRec = null }).ToArray()

};

erecord = erec;

}

Теперь надо отработать поиск и фиксацию в этом поле. Для перевода в режим ввода, пользуемся традиционным onclick в пустой ячейке. Запускается метод SearchLink, который активирует режим SearchLink записью свойства, которое ищется, в переменной searchlinkprop. Это, при следующей «перерисовке» должно внедрить поисковый элемент в активированной ячейке. Поиск – точно такой же, как и в самом начале страницы. Поиск оканчивается выбором подходящего варианта. Этот выбор фиксирует прямую ссылку в (копии) редактируемой записи

private void FixLink(string prop, string id, string tp, string name)

В теле этого метода происходит замена пустой прямой ссылки на полученную тройку Id, Tp, name и место «прикрепления» в виде имени свойства.

Для переноса решения на опорную таблицу ничего особенного делать не надо, надо воспроизвести html-формы в разделе опорной таблицы.

Следующая забота: зафиксировать отредактированную запись в базе данных. Это надо сделать в обработчике Ok(). Мы сформированную запись подаем движку для фиксации в базе данных.

И так далее… Подозреваю, что Вы уже потеряли мысль и обучение перестает быть эффективным. Доведенное до определенного конца решение мною дописано и перемещено на страницу ViewEdit. Можете его изучить, enjoy!

## Использование компонентов в Blazor

Полученное (или не полученное) в предыдущем разделе решение обладает сложной логикой, что существенно препятствует его пониманию. Попробуем воспользоваться компонентным подходом, имеющимся в блазоре для сведения большой задачи к множеству задач поменьше.

В этом месте целесообразно почитать документацию блазора, особенно касающуюся компонентов. Нам вполне пригодятся наработки по базе данных, онтологии и модели, все это сохранится если писать новое приложение в уже сформированном ранее проекте. Нам, в основном, надо написать новую страницу. Проще всего, стереть наработанное из старой страницы Index.razor и будем создавать ее заново.

Сначала изучим и попробуем создание и использование компонента. Одним из важных компонентов предыдущего проекта был поисковый элемент. На нем и поэкспериментируем. Поисковый элемент состоит из поля ввода/редактирования, сопряженного с переменной searchsample. Изменение значения этой переменной вызывает обращение к базе данных на поиск записей с таким же (похожим) именем. Найденные кандидаты изображаются списком для выбора. Выбор варианта заканчивает поисковые действия тем, что выбранная запись куда-то уходит, а сама форма переводится в начальное состояние.

Создадим папку Components, а в ней создадим блазор-компонент Search. Начальный вариант компонента будет совсем простым:

@using RDFEngine

<div>

<input **@bind**="searchsample" />

</div>

@if (!string.IsNullOrEmpty(searchsample))

{

IEnumerable<RRecord> query = null;

query = Infobase.engine.RSearch(searchsample);

foreach (RRecord rec in query)

{

<div>

<span>@rec.Tp</span> &nbsp;

<a href="javascript:void(0)" **@onclick**="@(e =>SearchResultClick(rec))">@rec.GetName()</a>

</div>

}

}

@code{

private string searchsample = "";

private void SearchResultClick(RRecord rec) { }

}

Теперь на пустой странице Index.razor разместим экземпляр этого компонента:

@page "/"

<**MagBlazor.Components.Search** />

Испытаем, убедимся, что он работает. Если не лень, можно размножить компонент и убедится, что каждый из экземпляров работает и работает автономно. Пока не работает нажатие на гиперссылку выбора варианта, так мы ее и не запрограммировали. Компоненту можно передать параметры по простой схеме. Допустим, нам надо «индивидуализировать» экземпляры компонента Search тем, что выводить результаты поиска в виде:

Результаты:

Вариант

Вариант

…

Передадим компоненту параметр заголовка. Для этого определим параметр и используем его в композиции

…

<div>@title</div>

…

[Parameter]

public string title { get; set; } = "";

А в основной странице зададим параметр

<MagBlazor.Components.Search title="Результаты:" />

Передача значений в компонент решает какие-то проблемы, но проблему возвращения результатов работы компонента родительскому элементу не решают. Было бы хорошо передать компоненту функцию, которая в какой-то момент сработает на данных компонента и вернет нужное значение. Такая функция (делегат) существует – это EventCallback. Так сказать, «обратный звонок». Определяется делегат в компоненте, напр.

[Parameter]

public EventCallback onSomethingHappend { get; set; }

Кроме того, надо внутри компонента выполнить «обратный звонок». Это делается как-то так:

protected async Task InternalHandler(T rec)

{

await onSomethingHappend.InvokeAsync(rec);

}

Здесь значение заданного типа посылается «наружу» через универсального делегата. Следующий момент – надо в какой-то момент времени запустить внутренний обработчик с вычисленным значением, напр. можно привязать к событию (onclick и др,).

<a href="…" **@onclick**="@(e => InternalHandler(rec))">click me!</a>

И наконец, во внешней программе, надо задать параметр-делегат onSomethingHappend. Задается напр. через лямбда-функцию, что-нибудь типа:

<**Search** onSomethingHappend ="@(arg => DoSomethingWithResult((T)arg)"**>**</**Search**>

Потренируемся на нашем компоненте Search. А заодно, будем доводить его до возможности полезного использования в нашем проекте. Итак, надо определить ровно 4 момента: параметр- делегат в компоненте, асинхронное выполнение метода, заданного делегатом, привязка запуска этого метода к событию в элементе, внешнее использование результата, полученного в компоненте через задание параметра-делегата. Три момента из четырех реализуются в компоненте, один – в вызывающем построении. Полный текущий тест компонента Search.razor:

@using RDFEngine

<div>

<input **@bind**="searchsample" />

</div>

@if (!string.IsNullOrEmpty(searchsample))

{

IEnumerable<RRecord> query = null;

query = Infobase.engine.RSearch(searchsample);

foreach (RRecord rec in query)

{

<div>

<span>@rec.Tp</span> &nbsp;

**<a href="javascript:void(0)" @onclick="@(e =>SearchResultClick(rec))">@rec.GetName()</a>**

</div>

}

}

@code{

**[Parameter]**

**public EventCallback onSelected { get; set; }**

private string searchsample;

**private async Task SearchResultClick(RRecord rec)**

**{**

**await onSelected.InvokeAsync(rec);**

**}**

}

Строчки, ответственные за возврат результата, выделены размером и полужирным. Полный текст текущего экспериментального варианта индексной страницы:

@using RDFEngine

@page "/"

**<MagBlazor.Components.Search onSelected="(rec) => UtilizeRecord((RRecord)rec)" />**

@code{

private string look = "";

private void UtilizeRecord(RRecord rec)

{

look = rec.Id;

}

}

<hr />

<div>@look</div>

Испытаем решение и выясним, что мы на правильном пути. В дальнейшем мы вернемся к компоненту Search, добавляя свойства, а пока перейдем к другим компонентам.

Нас будет интересовать построение таблицы. Рекомендую посмотреть на реализацию вьюера с разделе «Создание универсального табличного интерфейса». Там код построения таблицы во многом повторяется, так что есть смысл его выделить в компонет. Назовем его DrawTable. Заведем компонент с таким именем, перенесем в него 2-е решение по построению таблицы (оно более гибкое) из раздела. Студия сделает несколько красных подчеркиваний. Будем их устранять. Во-первых, компоненту не известны типы RField и RDirect. Просто добавим @using RDFEngine. Потом есть неизвестные переменные. Это defrow, t.list и p.Prop. По текстам предыдущих решений установим: defrow вычисляется из t.list и что тип t.list это массив RRecord, а p.Prop это строка. Добавим в код:

@code {

[Parameter]

public string pProp { get; set; }

[Parameter]

public RRecord[] tlist { get; set; }

}

Поменяем в тексте компонента t.list на tlist, p.Prop на pProp, GetName(rr) на rr.GetName(). А еще добавим в начало компонента @{ RRecord defrow = tlist[0]; }.

Теперь попробуем компонент в разработке.

## Полнотекстовый поиск

Некоторые считают, что полнотекстовый поиск это что-то «заоблачное». В основе полнотекстового поиска лежат довольно простые идеи, реализуемые простыми алгоритмами. Мы (поверхностно) познакомимся с этими идеями и алгоритмами.

Пусть есть множество документов, имеющих текстовое наполнение. Мы в этом разделе будем рассматривать документы как наборы слов. Для нас не будет существенным порядок слов и число повторов отдельных слов. Пусть есть поисковый образец, состоящий из нескольких слов. Нашей задачей будет нахождение документов, в которых встречаются слова из поискового образца. И чем больше слов встретилось, тем точнее документ соответствует образцу. Используя пары: документ (какая-то идентификация документа)-слово {d, w} преобразуем множество документов (и содержащихся в них слов) в таблицу. Далее, сгруппируем таблицу по словам, получим для каждого слова набор документов, в которых есть такое слово, превратим набор в множество. Получим хеш-словарь по которому мы (очень быстро) по заданному слову из образца находим все документы, в которых оно содержится. Применим эту процедуру для всех слов образца и построим таблицу {s, d}, где s – слово из образца (sample). Сгруппируем таблицу по документам. Теперь для каждого помещенного в таблицу документа будет вычислен набор слов, которые в нем присутствуют. Упорядочим документы по числу попавших из образца слов. Верхняя часть таблицы будет решением поставленной задачи.

Все очень просто! Но почему же поисковые системы состязаются своими решениями? Вопрос в оптимизациях. Во-первых, важно получить максимально релевантные решения (соответствующие запросу), в этих решениях должно быть как можно меньше «мусору». Во-вторых, оптимизация ведется в алгоритмах решения задач, особенно для задач, в которых хеш-таблица не помещается в ОЗУ или ее не рационально туда помещать.

Мы применим лишь одну оптимизацию, без которой поиск по словам будет неэффективным. Мы будем делать нормализацию слов, т.е. перевод всех слов в именительный падеж единственного числа. Делается это через лингвистические словари, мы воспользуемся словарем Зализняка. В упакованном виде его упрощенный вариант находится в директории Resources в виде текстового файла zaliznyak\_shortform.txt. Посмотрим на содержимое файла, там записаны строчки вида:

словарь словаря словарю словарь словарем словаре словари словарей словарям словари словарями словарях

словацкий словацкого словацкому словацкого словацкий словацким словацком словацкая

словеса словес словесам словеса словесами словесах

Формат очень простой, а поэтому удобный для использования в обработке. В одной строчке находятся: на первом месте нормальная форма конкретного слова, далее через запятую, другие формы (включая и нормальную). Создадим из набора строк хеш-словарь, отображающий слово в нормальную форму.

IEnumerable<object> record\_flow = …

var query = record\_flow

=============== Это старый текст, его читать не надо ===============

Изложенный в предыдущем разделе подход хорош тем, что он универсален, не зависит он наличия онтологии либо ее аналогов. В то же время, в этом подходе есть существенные недостатки:

* Порядок полей не определен;
* Глубина раскрытия дерева фиксирована;
* Все поля именуются вне зависимости от того, нужно это делать или нет, именование фиксировано используемым словарем;
* Отсутствует управление тем, что раскрывать нужно, что не нужно.

Для чего это нужно? Определенный порядок полей нужен в композициях, претендующих на понятность и логичность. Предлагается ввести в рассмотрение структуру (формат, шаблон), которая сможет и будет преодолевать указанные недостатки. Если немного подумать, то лучше опять же RRecord предложить трудно. Пусть есть значение класса RRecord и нам требуется сделать выборку из базы данных другого объекта этого же класса в соответствии с этим форматом. Строим выборку последовательно. Смотрим поле Id. Если оно определено, то это одна ситуация, если не определено, то – другая. В первой ситуации, по семантике RDF существует единственная запись с этим идентификатором. Или не существует вообще. Во второй ситуации, выделять нужно все элементы, удовлетворяющий дальнейшим условиям. Рассмотрим запрос с заданным Id. Если в базе данных есть запись с таким Id, то это выборка именно этого элемента. Но при условии выполнения других условий. Если тип в формате задан, то он задан для проверки, если не задан, то он берется из записи. Далее, выполняется анализ полей. Поле может быть задано или не задано, а если задано, то в формате еще может быть определено значение. Если поле не задано, то оно не берется в выходную запись. Если поле задано и его значение в формате не определено, то в выходную запись берется поле и его значение. Если поле задано и его значение в формате тоже задано, то поле используется только для проверки и не попадает в результирующую запись.

Аналогично рассматриваются прямые ссылки. Прямая ссылка определяется идентификатором предиката (также как и поля). Если в формате записи есть предикат прямой ссылки, то …

Запись о персоне в нашем случае состоит из идентификатора записи, идентификатора типа и двух полей. В более общем случае, полей может быть несколько. Давайте используем эту информацию для того, чтобы сформировать портрет. Мы каким-то удобным образом сформируем композицию. Например в виде пар имя атрибута или поля, значение атрибута или поля. Например, так:

<table>

<tr>

<td>id</td>

<td>@Model.Attribute("id").Value</td>

</tr>

<tr>

<td>type</td>

<td>@Model.Attribute("type").Value</td>

</tr>

@foreach (var field in Model.Elements("field"))

{

<tr>

<td>@field.Attribute("prop").Value</td>

<td>@field.Value</td>

</tr>

}

</table>

Но можно композицию сформировать и «в строке», т.е. как линейную развертку полей. Например, можно поля изображать парами имя\_поля: значение\_поля, а последовательность полей в виде перечисления через запятую отдельных полей. Для атрибутов нет большого смысла явно называть имена атрибутов (они всегда одинаковые), можно воспользоваться оформительскими возможностями HTML, напр. использую гиперссылку. Один из возможных вариантов такого портрета:

<div>

<a href="">@Model.Attribute("type").Value</a>

@foreach (var field in Model.Elements("field"))

{

<span>@field.Attribute("prop").Value</span>

<span>: </span>

<span>@field.Value</span>

}

</div>

Запись стала гораздо компактнее. Мы в дальнейшем воспользуемся и подобной компактной записью и табличным построением.

А пока сделаем свободную композицию. Нам какая информация важна? Тип записи, имя записи и записи, доступные через отношения. Мы стремимся к тому, чтобы персона сопровождалась набором фотографий, на которой персона изображена, а фотография сопровождалась списком персон, которые изображены на фотографии. Соответственно, элементы списков фотографий и персон можно снабдить гиперссылками, ведущими к портрету сущности. В данном построении предполагается два вида страниц: персоны и фотографии. Создадим страницу портрета персоны Person. Для этого сначала создадим действие контроллера, потом в этом действии вычислим модель, потом «нарисуем» страницу с данными из модели. Бросаем заниматься индексом Index, работаем с портретом персоны Person. Фрагмент, включаемый в HomeController.cs и вьюер Person.cshtml

public IActionResult Person(string id)

{

XElement model = Infobase.engine.GetRecordBasic(id, false);

return View("Person", model);

}

@using System.Xml.Linq

@model XElement

@{

if (Model == null) return;

}

<div><b>Персона</b></div>

<div>

<table>

@foreach (var field in Model.Elements("field"))

{

<tr><td>@field.Attribute("prop").Value</td><td><b>@field.Value</b></td></tr>

}

</table>

</div>

Проверьте, как работает страница. Не забудьте, что страница активируется запуском типа <https://localhost:44305/Home/Person/p23>

или

<https://localhost:44305/Home/Person?id=p23>

Изображена только часть желаемой информации. Другая часть – отражения. Чтобы войти в тему, сделайте изменение в акции контроллера Index, надо установить в параметрах формирования модели Infobase.engine.GetRecordBasic(id, false); второй параметр в true. Это означает, что мы хотим, чтобы кроме полей и прямых ссылок, в результат попали обратные ссылки. Посмотрите как изменится вид модели при запуске Home/Index/p24. Надеюсь, вы не убрали вывод XML-текста модели.

Кстати, когда Вам надоест вручную исправлять адресную строку, можно задействовать для этого «стартовую» страницу, обработав отсутствие идентификатора и послав не построение страницу (напр.) Stend. В эту страницу будете потом вставлять все осмысленные и тестовые вызовы. Я вот сейчас это сделаю.

Возвратимся к разработке. Расширенная модель для какого-то элемента будет выглядеть как-то так:

<record id="p25" type="person">

<field prop="name" xml:lang="ru">и25</field>

<field prop="age">23</field>

<inverse prop="reflected">

<record id="r261" />

</inverse>

<inverse prop="reflected">

<record id="r276" />

</inverse>

<inverse prop="reflected">

<record id="r306" />

</inverse>

</record>

То есть, появились обратные ссылки на отношения с идентификаторами r261, r276, r306. В принципе, эти записи можно было бы «раскрыть», т.е. сформировать для них модель в промежуточном представлении. Это можно сделать в программе контроллерного действия Person. Что-нибудь такое:

# Приложение 1. Промежуточное представление в XML-виде

Запись в промежуточном XML-представлении является XML-элементом с атрибутами id и type, значением которых являются идентификатор узла и тип узла (изображен желтым). Подэлементами записи являются текстовые поля и прямые ссылки. Расширенная запись включает в себя также обратные ссылки, напр. p1, p2. Запись p3817 в промежуточном представлении будет выглядеть:

<record id=’p3817’ type=’person’>

<field prop=’name’>Иванов</field>

<field prop=’birth’>1988</field>

<direct prop=’work-in’><record id=’o19302’/></direct>

<direct prop=’father’><record id=’p2317’/></direct>

<!—Следующие поля действительны для расширенного представления -->

<inverse prop=’p1’><record id=’id1’/></inverse>

<inverse prop=’p2’><record id=’id2’/></inverse>

</record>

Обратите внимание на оформление прямой и обратной ссылок через указания направления, свойства и вырожденной записи цели или источника. Такое оформление позволяет в дальнейшем, расширить запись до более развернутой древовидной композиции за счет расписывания этих ссылок.

Сначала – движок. Мы определим минимальные требования к движку через интерфейс. Движок – это база данных для будущих приложений. База данных у нас будет формироваться на основе RDF-модели и предоставлять несколько методов.

public interface IEngine

{

public void Clear();

public void Load(IEnumerable<XElement> records);

public IEnumerable<XElement> Search(string searchstring);

public XElement GetRecord(string id, bool addinverse);

}

База данных очищается. Потом – загружается. Внутри загрузки она еще и строится. После этого, база данных готова выполнять поиск и выборку записей. Поясним операции на рисунке и в последующих текстовых конструкциях.

Теперь создадим простой XML-движок, реализующий интерфейс IEngine. Движок выполним в виде класса XMLEngine, который конструируется на основе XML-представления RDF-документа. Поскольку реализация даже самого простого варианта движка не тривиальна, текст реализации в это месте приводиться не будет, мы оставим вам удовольствие разобраться в нем самостоятельно по проекту.

В тексте класса XMLEngine есть понятные комментарии о том, как класс устроен и реализован. Определенный нюанс имеется в словаре ссылочных элементов subelementsByResource. Это множество всех подэлементов записей, имеющих определенный rdf:resource, ключами выступают значения этого атрибута. Для чего такая структура? На самом деле, как XElement используются ссылки на элементы в структуре rdf (это общее свойство элементов классов при присваивании значений). У этих элементов есть имя, которое соответствует имени предиката и родительский элемент (свойство Parent), который является записью, содержащую данную ссылку. Напр. в записи

<person rdf:about="…">

<name xml:lang="ru">Иванов …</name>

<father rdf:resource="aaa123…" />

…

</person>

В словарь попадет ссылка на подэлемент father, причем попадет под ключом aaa123…

Для чего такая сложность? Так мы во вспомогательных структурах не будем порождать сложных элементов, а обойдемся ссылками.

В библиотеку RDFEngine, для удобства отладки, добавим генератор набора Фототека для того, чтобы иметь «под рукой» базу данных произвольного объема. Генератор сформируем в виде статической функции, помещенный к какой-то класс, я использовал вновь введенный public class PhototekaGenerator

public static IEnumerable<XElement> Generate(int npersons)

{

Random rnd = new Random();

int np = npersons;

int nf = npersons \* 2;

int nr = npersons \* 6;

IEnumerable<XElement> recordFlow =

Enumerable.Range(0, np)

.Select(i => new XElement("person", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}about", "p" + i),

new XElement("name", new XAttribute("{http://www.w3.org/XML/1998/namespace}lang", "ru"), "и" + i),

new XElement("age", "23")))

.Concat(

Enumerable.Range(0, nf)

.Select(i => new XElement("photo", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}about", "f" + i),

new XElement("name", "dsp" + i))))

.Concat(

Enumerable.Range(0, nr)

.Select(i => new XElement("reflection", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}about", "r" + i),

new XElement("reflected", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}resource", "p" + rnd.Next(np))),

new XElement("indoc", new XAttribute("{http://www.w3.org/1999/02/22-rdf-syntax-ns#}resource", "f" + rnd.Next(nf))))));

;

return recordFlow;

}

Теперь, как и обещалось, создадим консольный проект TesterConsole для тестирования и опробуем созданную библиотеку (классов). В него добавим ссылку на библиотеку RDFEngine ( в дух местах: проекте и в заголовках using). Далее будет создание движка, загрузка записями через генератор, проверка поиска и выборки.

int npersons = 100;

// Создадим движок

RDFEngine.XMLEngine engine = new XMLEngine();

engine.Load(PhototekaGenerator.Generate(npersons));

engine.Build();

// Попробуем поиск

foreach (var rec in engine.Search("и9"))

{

Console.WriteLine(rec.ToString());

}

Console.WriteLine();

// Попробуем выборку

int key = npersons \* 2 / 3;

var record = engine.GetRecordBasic("p" + key, false);

Console.WriteLine(record.ToString());

Можно испытать движок на производительность, напр. по выборке

// Испытание на производительность

Random rnd = new Random();

System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();

npersons = 10\_000;

engine.Clear();

engine.Load(PhototekaGenerator.Generate(npersons));

engine.Build();

Console.WriteLine("new generation ok.");

Console.WriteLine("====");

int nprobes = 1000;

sw.Start();

for (int i = 0; i < nprobes; i++)

{

key = rnd.Next(npersons);

var r = engine.GetRecordBasic("p" + key, false);

}

sw.Stop();

Console.WriteLine($"duration of {nprobes} probes for {npersons} persons: {sw.ElapsedMilliseconds} ms.");

Console.WriteLine("====");

При испытании можно выяснить, что 1) выборка работает (очень) быстро – порядка 6 мс. на 1000; 2) выборка работает на объемах до 1 млн. персон (в базе данных около 25 млн. триплетов); 3) поиск выполняется медленнее, для 10 тыс. персон, 1 поиск по имени занимает около 30 мс. В принципе, характеристики для многих случаев достаточны.