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:

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:

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.

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.

## 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, соответствующиц 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. (Такие отображения назовем проекциями)

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). Далее, мы ввели обратные свойства. В целом, начало построения соответствует нашему RRecord. Но на этом, дерево обрывается. Обрывается на ссылках Resource и Source. 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; }

}

Как нам воспользоваться этим расширением? Систему базы данных (реализации движка) мы пока трогать не будем. Кроме реализации другого варианта выборки портрета. Нам теперь нужна не просто выборка портрета, а построение дерева с корнем, идентифицированным заданным идентификатором. Назовем новый метод GetRTree. Его реализация приведена в REngine.cs, если чувствуете в себе силы, можете попробовать сделать свой вариант.

Можно испытать GetRTree также как испытывали GetRRecord:

var engine = new RDFEngine.REngine();

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

engine.Build();

var result = engine.GetRTree("p66", 0, null);

Console.WriteLine(result.ToString());

result = engine.GetRTree("p66", 1, null);

Console.WriteLine(result.ToString());

result = engine.GetRTree("p66", 2, null);

Console.WriteLine(result.ToString());

Правда результат вычисления дерева глубиной 2 будет запутанным, можно попробовать все это проанализировать.

Теперь мы попробуем использовать такое дерево в построении портретной страницы. Назовем действие ShowTree. В части контроллера, там действия аналогичны тем, которые были ранее.

=============== Это старый текст ===============

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>

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.