Разработка адаптеров на платформе .NET Framework/.NET Core

В данном документы описаны шаги для построения веб-сервисов, взаимодействующие через платформу X-Road версии 6.

Чтобы облегчить материал вам предоставляется решение, исходный код которого расположен по следующему адресу <https://github.com/tunduk-gov-kg/dotnet-adapters-demo>.

Решение разбито на следующие проекты:

* *Consumer.XRoad\_End*
* *Consumer.Xroad\_Start*
* *Producer.Data*
* *Producer.Service*
* *Producer.Xroad\_End*
* *Producer.Xroad\_Start*

Краткое описание проектов:

*Producer.Data* – проект .NET Core Library содержит доменные сущности и прилагаемые к ним конфигурации для взаимодействия с базой данных через ORM Entity Framework Core. Папка SQL\_Scripts содержит скрипт для создания таблиц в СУБД Postgres и тестовые данные.

*Producer.Service* – слой бизнес логики также является проектом .NET Core Library и содержит сервисы которые служат в качестве связующего звена между слоем представления и слоем данных.

*Хочется отметить, что эти два проекта в ни в коем образе не влияют на инфраструктуру Xroad.*

*Producer.Xroad\_End* – веб-сервис на Asp.NET Core версии 2.2, готовая реализация Producer – ДЕМО адаптера.

*Producer.Xroad\_Start* – это пустой шаблон проекта на Asp.NET Core 2.2 и именно с него мы и начнем обучение разработке адаптеров типа Producer.

*Consumer.XRoad\_End* – WinForms проект содержит готовую реализацию ДЕМО адаптера типа Consumer.

*Consumer.Xroad\_Start* – пустой шаблон проекта WinForms для разработки Consumer.

В конечном итоге у нас должна появиться две системы, взаимодействующие через систему X-Road. Первая система будет запрашивать данные о человеке по ПИН и выводить эти данные на форму. Вторая система будет являться источником этих данных и будет по запросу выдавать их.

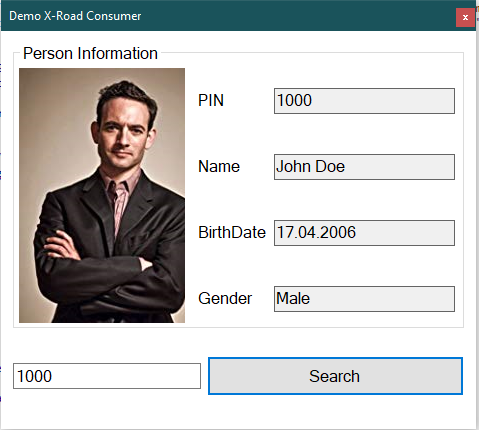


Рисунок 1 Программа для отображения данных о человеке

# Разработка Producer

Разработка провайдера услуг или так называемого Producer будет вестись на платформе .NET Core версии 2.

*Выбор платформы неограничен, потому что протокол SOAP описывает лишь спецификации к реализуемым веб-сервисам с помощью языка описания веб-сервисов WSDL.*

И так нам предстоит решить следующую задачу:

На сервере располагается база данных с информацией о людях. БД содержит таблицу Person с колонками:

* PIN – персональный идентификационный номер
* Name – ФИО человека
* Gender – пол человека
* BirthDate – дата рождения человека
* Photo – фотография человека

**Требуется реализовать SOAP сервис называемый GetPerson который будет выдавать информацию о человеке по входящему пин и зарегистрировать этот сервис в системе X-Road;**

Вам предоставляется тестовый проект называемый Producer.Xroad\_Start.

Для начала вам потребуется подключить библиотеку XRoadLib.AspNetCore версии 1.3.7 из Nuget репозитории.

Создадим папку DataContract где будут размещаться типы для сериализации в формат XML и для десериализации типов во входящих запросах. Эти типы будут далее описываться в XML схеме при генерации WSDL файла.

Для задания имени XML типа и пространства имен создаваемого класса его нужно пометить стандартным атрибутом XmlType и в качестве параметров указать TypeName = имя типа, Namespace = пространство имен.

Но вы можете опустить эти аргументы и просто указать что данный класс просто является XmlType либо вовсе проигнорировать данный атрибут.

Далее вам следует унаследоваться от абстрактного класса XRoadSerializable иначе данный тип будет игнорироваться при генерации XSD схемы

Пример:

[XmlType(TypeName = "Person", Namespace = "http://producer.xroad.com")]

public class PersonModel : XRoadSerializable

Генерирует следующий тип XML который будет находиться в пространстве имен http://producer.xroad.com

<xsd:complexType name="Person">

Атрибут XRoadXmlElement наследует атрибут XmlElement и добавляет новые свойства такие как IsOptional и UseXop

Параметр IsOptional указывает что поле является опциональными и при сериализации поле со значением NULL не включается в XML.

Параметр UseXop указывает следует ли включать поле типа Stream в тело сообщения в виде закодированной base64 строки либо выносить в отдельную часть HTTP запроса. Тогда HTTP запрос помечается как multipart/related и разделяется на несколько условных частей разделяемых значением mime/boundary.

Пример:

[XRoadXmlElement(ElementName = "Photo", IsOptional = false, Order = 4)]

public Stream Photo { get; set; }

Создается обязательное поле Photo в типе Person идущий 4ым по очереди.

А вот так выглядит xml схема

<xsd:element xmime:expectedContentTypes="application/octet-stream" name="Photo" type="xsd:base64Binary"/>

Как вы заметили добавился атрибут expectedContentTypes со значением application/octet-stream это указывает что это поле будет содержать двоичные данные закодированные в base64.

Создаем в папке DataContract два типа:

[XmlType(TypeName = "Gender")]

public enum GenderEnum

{

Male = 1,

Female = 0

}

[XmlType(TypeName = "Person", Namespace = "http://sample.com")]

public class PersonModel : XRoadSerializable

{

[XRoadXmlElement(ElementName = "Pin", IsOptional = false, Order = 0)]

public string Pin { get; set; }

[XRoadXmlElement(ElementName = "Name", IsOptional = false, Order = 1)]

public string Name { get; set; }

[XRoadXmlElement(ElementName = "BirthDate", IsOptional = false, Order = 2)]

public DateTime BirthDate { get; set; }

[XRoadXmlElement(ElementName = "Gender", IsOptional = false, Order = 3)]

public GenderEnum Gender { get; set; }

[XRoadXmlElement(ElementName = "Photo", IsOptional = false, Order = 4)]

public Stream Photo { get; set; }

}

Создаем папку ServiceContract и обьявляем следующий интерфейс:

public interface IPersonSoapService

{

[XRoadService("GetPerson", AddedInVersion = 1)]

[XRoadTitle("RU","Сервис для выдачи персональных данных по пин")]

PersonModel GetPerson(string pin);

}

Атрибут XRoadService обязателен. В нем вы указываете название вашего сервиса и версию если вы предполагаете, что она будет меняться со временем.

Атрибут XRoadTitle не обязателен, он служит для добавления информации для разработчиков. При добавлении такого атрибута в wsdl появится следующий тег с внутренними полями

<wsdl:documentation>

<xrd:title xml:lang="RU">

Сервис для выдачи персональных данных по пин

</xrd:title>

</wsdl:documentation>

В целом wsdl будет выглядет следующим образом

<wsdl:portType name="PortTypeName">

<wsdl:operation name="GetPerson">

<wsdl:documentation>

<xrd:title xml:lang="RU">Сервис для выдачи персональных данных по пин</xrd:title>

</wsdl:documentation>

<wsdl:input message="tns:GetPerson"/>

<wsdl:output message="tns:GetPersonResponse"/>

</wsdl:operation>

</wsdl:portType>

В реализации сервиса вы можете прописать любое поведение которые вы учтете нужным это может быть формирование запроса на основе данных из базы данных либо делать какие-либо вычисления и т.д.

В нашем случае мы нам нужно внедрить через конструктор интерфейс IPersonService и делегировать вызов GetPerson(pin)

public PersonModel GetPerson(string pin)

{

var person = \_personService.GetPerson(pin);

return person == null ? null : ConvertToModel(person);

}

Важно отметить, что в системе X-Road ошибки подразделяются на две группы:

Technical Issue и Non-Techincal Issue чем они отличаются вы можете прочитать здесь <https://github.com/nordic-institute/X-Road/blob/develop/doc/Protocols/pr-mess_x-road_message_protocol.md#annex-d-example-fault-messages>

В нашем случае если мы не сможем найти человека по заданному пин то мы возвращаем ответ с атрубтом nil=true;

И так предположим вы описали все модели сообщений и снабдили сервисы метаданными, дальше теперь нам потребуется чтобы библиотека XRoadLib сгенерировал WSDL файлы динамически.

Генерация WSDL происходит на основе метаданных, которыми мы снабдили сервисы и модели. Чтобы указать в какой сборке находятся эти элементы мы должны создать следующий класс PersonSoapServiceManager унаследованный от ServiceManager<XRoadHeader40>.

В конструкторе PersonSoapServiceManager вызывается базовый конструктор со следующими параметрами:

Указывается версия протокола 4.0 для текущей реализации X-Road 6

И DefaultSchemaExporter с указанием пространства имен и с указанием сборки которая будет являться источником метаданных.

В конечном итоге класс PersonSoapServiceManager будет выглядет следующим образом

public class PersonSoapServiceManager : ServiceManager<XRoadHeader40>

{

public PersonSoapServiceManager()

: base("4.0", new DefaultSchemaExporter("http://producer.xroad.com",

typeof(PersonSoapServiceManager).GetTypeInfo().Assembly))

{}

}

Далее открываем класс Startup.cs и регистрируем классы в IOC контейнере в методе ConfigureServices(IServiceCollection services):

services.AddScoped<IPersonSoapService, PersonSoapService>();

services.AddScoped<PersonSoapServiceManager>();

services.AddXRoadLib();

Обратите внимание что время жизни сервиса является Scoped это означает каждый сервис «живет» в контексте http запроса а затем он уничтожается. Если вы укажете его как Singleton то это может спровоцировать ошибки типа ObjectDisposedException. Более подробную информацию вы найдете здесь

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2>

Расширяющий метод AddXRoadLib() является обязательными так как он проводит дополнительные настройки для корректной работы библиотеки.

Открываем метод Configure (IApplciationBuilder app,IHostingEnvironment env)

Добавляем следующую строчку кода:

app.UseXRoadLib(routeBuilder =>

{

routeBuilder.MapWsdl<PersonSoapServiceManager>("");

routeBuilder.MapWebService<PersonSoapServiceManager>("");

});

Здесь мы настраиваем настройку маршрутизации. Расширяющему методу UseXRoadLib передается лямбда выражение где у аргумента routeBuilder есть такие методы как MapWsdl и MapWebService

MapWsdl – служит для задания адреса где будет размещаться wsdl файл для типа PersonSoapServiceManager (Пожалуйста не ставьте знак «/» перед указанием относительного адреса иначе будет выводиться ошибка тоже касается и метода MapWebService)

MapWebService – служит для задания адреса где будут обрабатываться SOAP запросы PersonSoapServiceManager.

*Если вам непонятен смысл ServiceManager то просто представьте себе что это всего лишь контейнер который содержит набор сервисов. Этот контейнер указывает в какой сборке искать метаданные для дальнейшей генерации wsdl.*

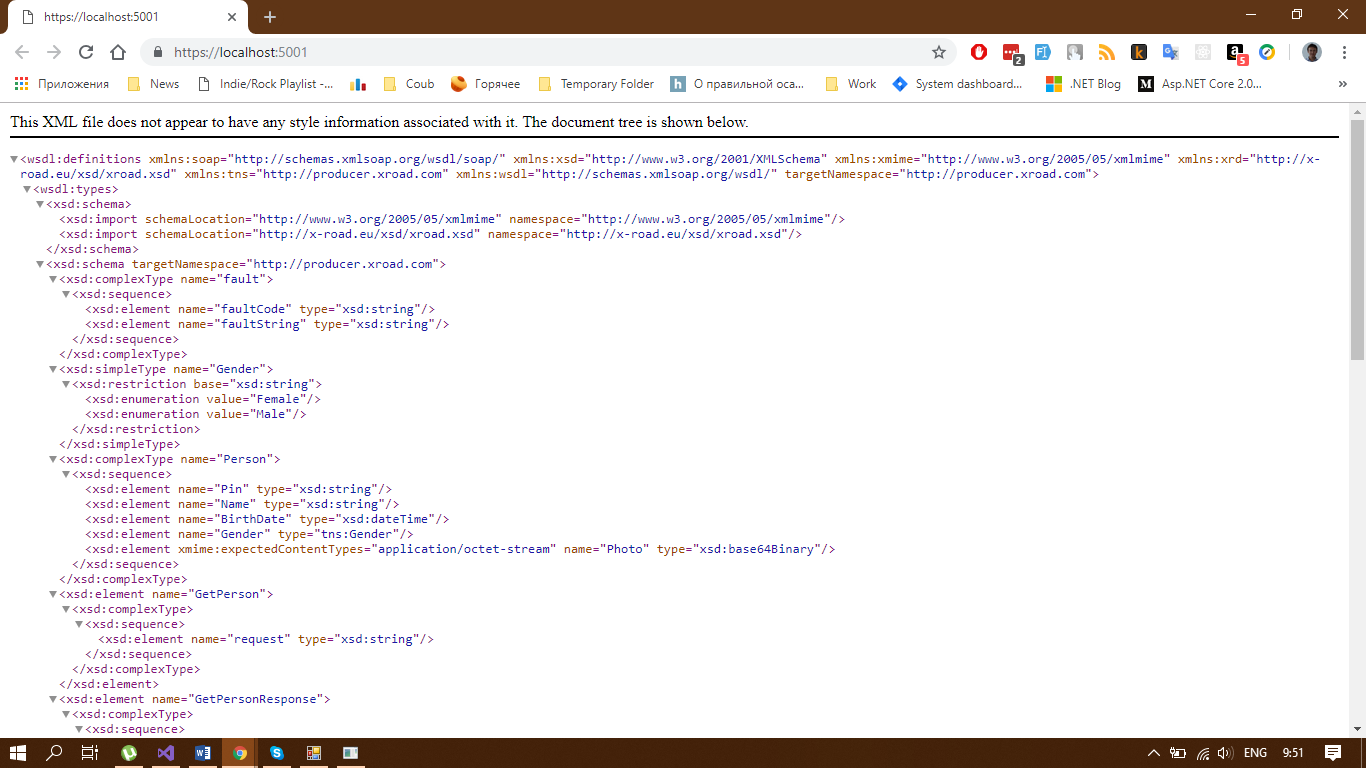
Прошу еще обратить внимания на класс DefaultSchemaExporter: при генерации wsdl файла вызывается метод ExportProtocolDefinition.

Этот метод принимает тип ProtocolDefinition, который содержит такие свойства как

* ProducerNamespace
* ContractAssembly
* TechNotesElementName
* PortTypeName
* BindingName
* PortName
* SoapAddressLocation
* И так далее.

Я настоятельно рекомендую наследоваться от класса DefaultSchemaExporter и переопределять метод ExportProtocolDefinition. В этом методе вы можете перезаписать свойства ProtocolDefinition и указать более близкие к реальности значения.

Если вы сделали все правильно согласно этому мануалу у вас запустится проект ASP.NET Core.



И в браузере вы увидите динамически сгенерированный wsdl.

Более подробную информацию о WSDL вы найдете здесь <https://www.w3.org/TR/2001/NOTE-wsdl-20010315>

На этом разработка Producer заканчивается.

# Разработка Consumer