Министерство образования и науки Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Факультет информатики
Кафедра программной инженерии

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК

Руководитель ООП
д-р физ.-мат. наук, профессор
_____ О.А. Змеев
«01» июня 2016 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

РАЗРАБОТКА КОМПОНЕНТОВ КОРПОРАТИВНОЙ ИНФОРМАЦИОННОЙ СИСТЕМЫ УНИВЕРСИТЕТА

по основной образовательной программе подготовки магистров
«Управление проектами по разработке программного обеспечения»
направление подготовки
02.04.02 – Фундаментальная информатика и информационные технологии

Цыганков Алексей Александрович

Научный руководитель ВКР,
д-р физ.-мат. наук, профессор
_____ О.А.Змеев
подпись
«01» июня 2016 г.

Автор работы
студент группы №1447
_____ А.А.Цыганков
подпись

Томск 2016

РЕФЕРАТ

Выпускная квалификационная работа 44 с., 22 рис., 19 источников.

КОРПОРАТИВНАЯ ИНФОРМАЦИОННАЯ СИСТЕМА, СЕРВИС-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА, SOA, МИКРОСЕРВИСЫ, ОЧЕРЕДЬ СООБЩЕНИЙ

Объекты исследования – варианты реализации и взаимодействия компонентов информационной системы.

Цель работы – разработать компоненты корпоративной информационной системы университета.

Методы исследования – изучение существующих подходов к разработке компонентов информационной системы и их взаимодействия, разработка общих средств взаимодействия программных сервисов.

Результаты работы – разработаны компоненты корпоративной информационной системы университета на основе сервис-ориентированной архитектуры (SOA), разработаны повторно используемые библиотеки для упрощения взаимодействия между сервисами, разработаны адаптеры для интеграции с внешними системами.

Оглавление

Введение

Идея корпоративных информационных систем не нова. Существует огромное множество различных готовых информационных систем, доступных к внедрению в организацию. Ещё больше различных корпоративных информационных систем, созданных силами и средствами самих этих организаций для своих собственных нужд. Более того, в различных отделах одной и той же организации зачастую можно увидеть несколько информационных систем, которые достаточно часто никак не связаны друг с другом. В больших организациях это создаёт существенную проблему.

Решений этой проблемы две.

1. Создать новую единую информационную систему, включающую в себя возможности всех, которые уже используются в организации.
2. Попытаться связать уже имеющиеся информационные системы между собой.

Очевидный недостаток первого решения – большая стоимость. Каждая из используемых информационных систем могла создаваться годами с учетом опыта многих пользователей из различных организаций. Собрать этот опыт из нескольких систем в одну и повторить результаты – задача зачастую невероятная.

Недостаток второго решения заключается в том, что различные информационные системы, создаваемые обычно абсолютно независимо друг от друга различными командами разработчиков не предназначены для связывания друг с другом. Некоторые из них реализуют внешние интерфейсы различного рода, чтобы можно было взять данные из них для автоматической обработки. Некоторые реализуют различные стандартные протоколы обмена данными. Но очень редки случаи, когда несколько информационных систем действительно полноценно интегрируются друг с другом.

В университетах, как и в любой крупной организации, также применяются различные информационные системы в различных отделах. В

рамках данной работы решается проблема связывания этих систем путём разработки промежуточных компонентов на основе сервис-ориентированной архитектуры. Разрабатываемые компоненты интегрируются с существующими системами, объединяют их друг с другом и добавляют недостающую функциональность.

Работа выполняется в рамках Томского государственного университета и все примеры будут приведены на нём, но с небольшими изменениями та же работа применима и к любому другому университету.

1. Постановка задачи

Цель данной работы – разработать компоненты корпоративной информационной системы университета.

Перечислим эти компоненты и опишем основные функциональные требования, предъявляемые к ним.

ТГУ.Выпускники – это каталог всех выпускников университета. Данный компонент должен позволять пользователю:

1. просмотреть список всех выпускников по каждому факультету в отдельности;
2. осуществить поиск по всему списку студентов университета по различным критериям;
3. просмотреть данные о выпускнике, включая год и факультет, которые этот выпускник окончил, а также основную информацию о его деятельности после выпуска из университета;
4. возможность импортировать данные о выпускниках из файлов в формате xls;
5. возможность совершать рассылки по email по отдельным группам выпускников.

ТГУ.Сотрудники – это каталог всех сотрудников университета. Данный компонент должен позволять пользователю:

1. просмотреть список сотрудников в каждом из подразделений и отделов университета;
2. просмотреть основную информацию о сотруднике: его звания, место работы, фотографию;
3. просмотреть научные и другие достижения сотрудников, результаты его научной деятельности, в том числе список публикаций, причем

данные о публикациях и прочих достижениях берутся из внешней системы;

В случае, если пользователь является аутентифицированным сотрудником университета, то в рамках этого компонента необходимо предоставить сотруднику возможность принимать участие в различных бизнес-процессах, требующих авторизации. На момент написания работы, было выделено два таких процесса:

1. просмотреть представленные в понятном виде начисления по своей заработной плате;
2. заполнить и утвердить свой индивидуальный план.

ТГУ.Профили – это компонент для ведения личных и групповых блогов. Данный компонент должен позволять пользователю:

1. вести свой личный блог;
2. вести один блог на группу пользователей;
3. вести личный блог за другого человека, при этом не должно быть видно, кто именно ведёт этот блог;
4. использовать блог в качестве сайта отдела или некоторого структурного подразделения университета.

ТГУ.Новости – это агрегатор новостей со всех интернет-ресурсов ТГУ и других ресурсов, которые предоставляют свои новости в формате RSS. Данный компонент должен позволять пользователю:

1. выбрать те ресурсы ТГУ, с которых он хочет получать новости;
2. указать те ресурсы, не входящие в список ресурсов ТГУ, с которых он хочет получать новости;
3. получать новости из компонента ТГУ.Профили;
4. указать, новости каких из выбранных источников показывать в общей новостной ленте в данный момент.

ТГУ.Сообщения – это чат для общения между пользователями системы. Данный компонент должен позволять пользователю:

1. просмотреть список своих недавних собеседников;
2. найти нужного собеседника из всего множества пользователей системы;
3. вести обмен мгновенными сообщениями с любым пользователем системы.

ТГУ.Студенты – это каталог всех студентов ТГУ. Данный компонент должен позволять пользователю:

1. просмотреть список всех студентов в каждом из факультетов;
2. просмотреть базовую информацию о студенте: место учебы, год поступления, фотографию;
3. просмотреть научные и другие достижения студентов, результаты научной деятельности, в том числе список публикаций.

ТГУ.Уведомления – это компонент для рассылки уведомлений. Данный компонент должен позволять пользователю выбрать группу пользователей системы и разослать этой группе текстовое уведомление. Другим системам компонент должен предоставлять интерфейс для отправки текстовых уведомлений конкретным пользователям.

Карта инновационно-активной среды ТГУ – это сайт для демонстрации текущей деятельности и результатов работы в рамках проектов из программы повышения конкурентоспособности «5 – 100». Данный компонент должен предоставлять пользователю возможность:

1. заполнить данные о проекте из программы «5 – 100»;
2. выкладывать новости о работе проекта;
3. выкладывать проектную документацию;

4. вести список экспертных групп;
5. вести список инициатив по направлениям;
6. организовывать мероприятия в рамках программы «5 – 100»

Основные общие особенности всех описанных компонентов.

1. Высокая степень независимости. Каждый компонент может разрабатываться отдельно от других. Очень мало данных, которые необходимы более чем в одном компоненте.
2. Различные источники финансирования, и, как следствие, различные заинтересованные лица. Эти лица хотят получить необходимый им продукт, и при этом их не интересует развитие других компонентов. Объем финансирования одного компонента должен определять скорость развития только этого компонента, а не других.
3. Экспериментальный характер. Должна быть возможность исключить за ненадобностью или заменить любой из разрабатываемых компонентов без существенного ущерба остальной системе.

В связи с описанными особенностями было принято решение построить архитектуру системы на основе шаблона «Сервис-ориентированная архитектура».

2. Описание сервис-ориентированной архитектуры

1. Общее описание

Существует очень много различных толкований понятия сервис-ориентированной архитектуры. Мартин Фаулер пишет о том, что не существует единого определения, которое можно было бы назвать точным. При этом он приводит четыре различных расшифровки этого понятия, которыми обычно пользуются разработчики [1].

Для однозначности выделим основные признаки, которыми обладает сервис-ориентированная архитектура. Сервис-ориентированная архитектура (Service Oriented Architecture, SOA) подразумевает архитектуру, в которой вместо единого приложения создаётся несколько web-сервисов, которые обмениваются между собой данными по стандартным сетевым протоколам (например, http).

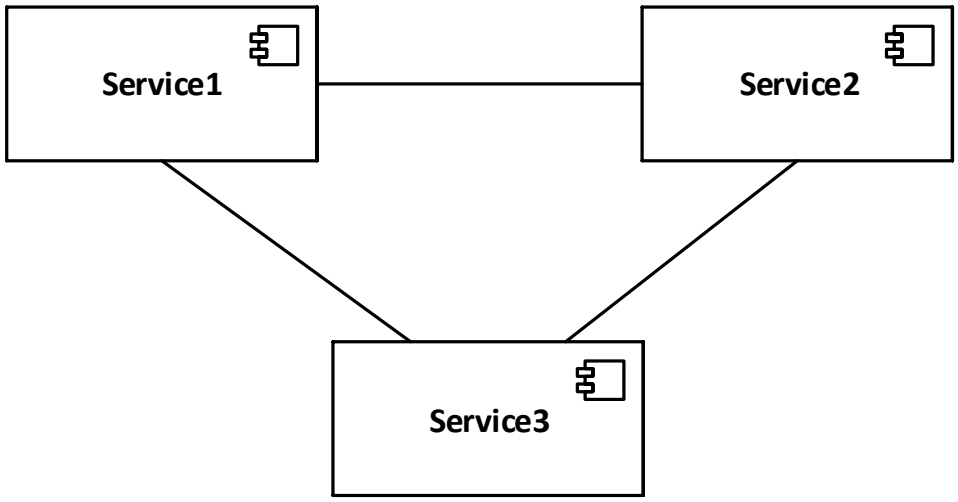


Рис. 1 Сервис-ориентированная архитектура

Web-сервисы зачастую противопоставляются монолитам. Монолит – это единое приложение, которое включает в себе все возможности системы целиком. Размеры монолита и web-сервиса зависят от размеров всей системы, но обычно размер web-сервиса очень небольшой. Считается, что для эффективной разработки один web-сервис должен быть такого размера, чтобы

полностью уместиться в сознание одного среднего разработчика, включая основные детали реализации. Если web-сервис превышает эти размеры, то его принято делить на несколько новых web-сервисов. Монолит в свою очередь фактически не ограничен в размерах. При этом в больших системах зачастую встречаются ситуации, когда один разработчик знает только некоторую часть этого приложения, и ни у кого в команде нет полной информации о всём приложении.

Web-сервисы размещаются на серверах независимо друг от друга. Под этим понимается несколько условий.

1. Каждый отдельный web-сервис не зависит от того, на каких серверах и в какой сети располагаются другие сервисы этой системы. Web-сервисы могут располагаться как на одном, так и на нескольких серверах, при этом они об этом могут и не знать. Единственное, что необходимо учитывать, это то, что при активном обмене данными между сервисами и высоким требованиям к скорости работы следует размещать сервисы как можно ближе друг у другу, чтобы время на сетевое взаимодействие было минимальным.
2. Каждый отдельный web-сервис разворачивается независимо от того, как разворачиваются другие сервисы. Другими словами, нет никакого порядка или других правил, которые бы регламентировали разворачивание сервисов относительно друг друга.

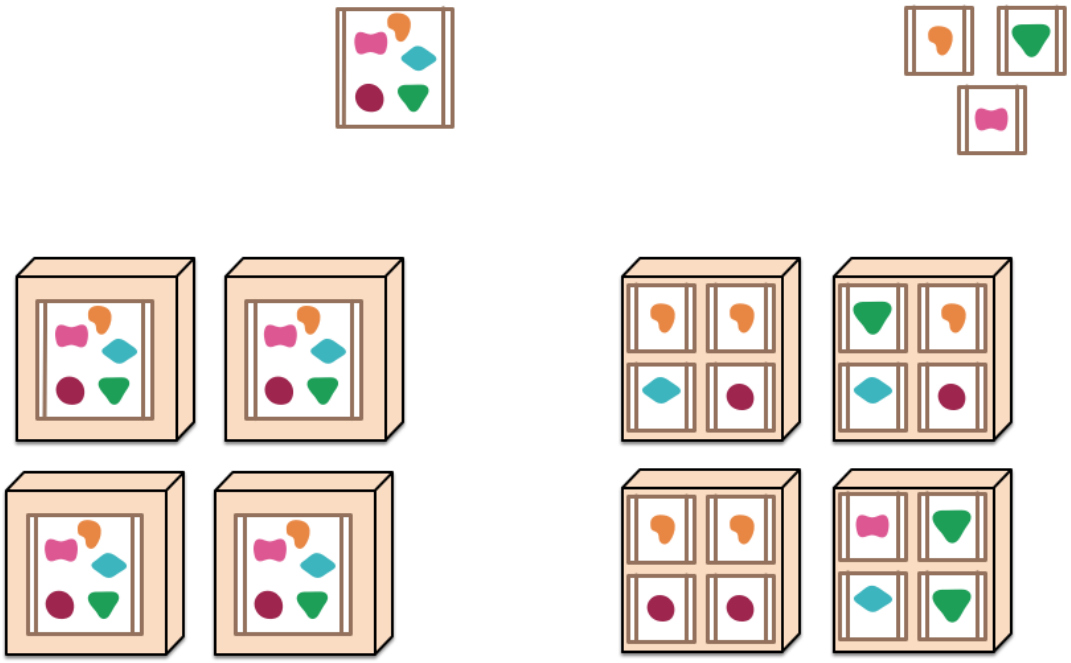


Рис.2 Сравнение монолита и сервисов SOA. Распределение между серверами.

Насколько бы ни были web-сервисы независимыми, они в любом случае должны обмениваться данными или отправлять друг другу команды. Общение между сервисами осуществляется по стандартным сетевым протоколам. Например, это может быть http, очередь сообщений или любое другое средство сетевого обмена данными. В частности, распространение получил формат REST API, который определяет правила взаимодействия по протоколу HTTP.

Однако, не любую систему разумно реализовывать по сервис-ориентированной архитектуре. Существуют системы, в которых данные сильно связаны друг с другом. Если представить модель предметной области в виде графа, где вершинами графа будут типы сущностей, то этот граф будет иметь высокую связность. В этом случае при попытке разделить систему на сервисы часть вершин графа попадёт в один сервис, а часть – в другой. Между сервисами окажется большое число рёбер, а это означает, что между сервисами будет чрезмерно большой обмен данными, а целостность данных придётся соблюдать не средствами БД, а программными средствами, что весьма трудоёмко и повышает вероятность возникновения ошибок. В таких случаях обычно рекомендуется не делить систему на сервисы, а оставлять её в

виде монолита. Возможно, удачно получится выделить в отдельные сервисы лишь небольшие независимые части.

2. Сравнение с микросервисной архитектурой

Часто обсуждают или описывают сервис-ориентированную архитектуру вместе с архитектурой, основанной на микросервисах. Как говорилось в первой части данный работы, у сообщества нет точного определения сервис-ориентированной архитектуры. В связи с этим одни считают, что SOA и микросервисы – это одно и то же, другие их противопоставляют, а третьи считают, что микросервисы – это частный случай SOA. Упомянутый ранее Мартин Фаулер вместе с Джеймсом Льюисом в своей статье Microservices [2] [3] также описывает эту проблему.

В другой статье Джеймс Льюис указывает на то, что микросервисы должны быть максимально маленькими и простыми, «полностью умещаться в голове», в то время как в SOA размеры сервисов не играют столь важную роль [4]. SOA нам говорит о том, что мы должны делить приложение на сервисы, согласно бизнес-требованиям. Обычно такие сервисы превращаются в отдельные сайты, которые обмениваются между собой данными, причем один такой сайт сам по себе может быть достаточно сложным и громоздким.

Согласно микросервисной архитектуре, подобные большие сервисы необходимо дробить ещё на несколько сервисов, чтобы каждый из них был очень простым, решал какую-то одну задачу. Однако бизнес-требования не позволяют плодить бесчисленное количество отдельных сайтов, ведь это будет неудобно конечному пользователю. Поэтому такие сервисы продолжают выглядеть как единый компонент, несмотря на то, что внутри он состоит из множества маленьких компонентов-микросервисов. Загружая одну страницу, пользователь может увидеть данные, полученные сразу от нескольких десятков микросервисов. При этом одни и те же микросервисы могут быть использованы в нескольких крупных сервисах.

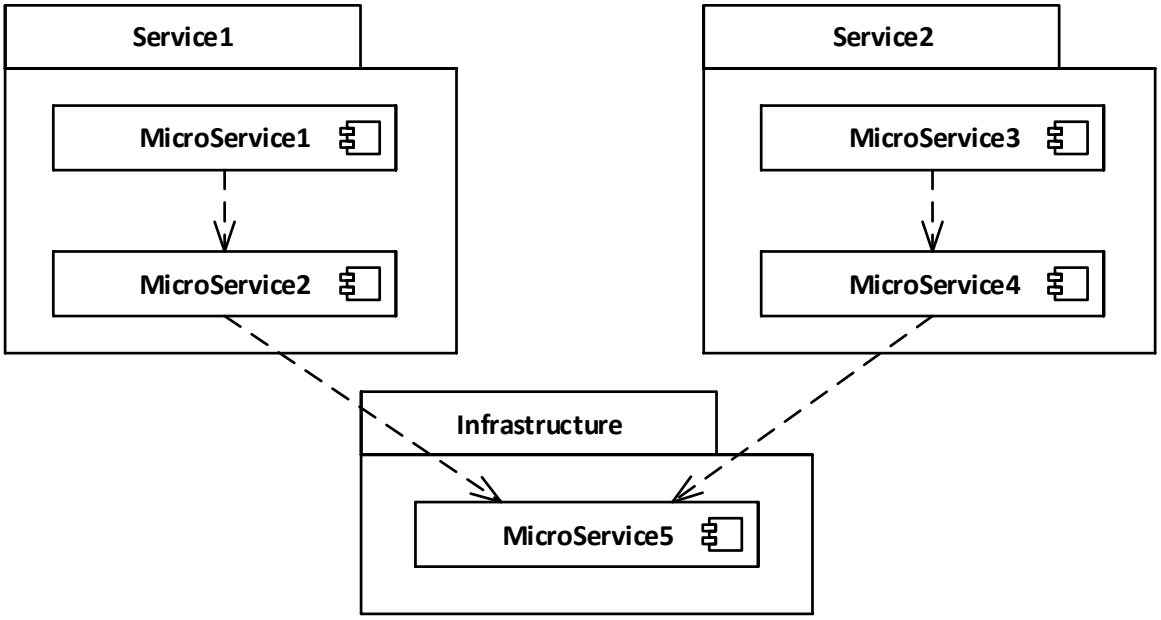


Рис. 3 Микросервисная архитектура

Подытожим сравнение SOA и микросервисной архитектуры. Микросервисы не нарушают ни одного принципа SOA, а напротив, только дополняют её [5]. Таким образом можно сказать, что микросервисная архитектура – это всё-таки частный случай SOA. При этом во всём приложении (или группе приложений) может применяться идея крупных сервисов на основе SOA, а при реализации отдельных сервисов могут применяться микросервисы.

3. Enterprise Service Bus

Чтобы контролировать общение между сервисами, иногда прибегают к созданию Enterprise Service Bus (ESB), которая служит промежуточным слоем между всеми сервисами, которая контролирует прохождение всей информации, а также доступ к этой информации [6] [7]. ESB может реализовывать свой собственный протокол для обмена сообщениями или использовать один или несколько из стандартных протоколов, например, JMS или AMQP.

Во многих случаях ESB становятся очень сложными в реализации и в использовании. Особенно это касается случаев, когда в системе необходимо

интегрировать старые сервисы (legacy systems), которые создавались без учёта создания ESB. Для каждого такого сервиса необходимо создавать некий адаптер, который будет преобразовывать сообщения из формата, в котором сервис может отдавать информацию, в формат, которые требуется в ESB. В итоге стоимость таких систем и работ по интеграции с ней других компонентов может превышать выгоду. Об этом говорят и Мартин Фаулер с Джимом Веббером [2] [8], и практический опыт Томского государственного университета, связанный с попыткой использования этого подхода ранее.

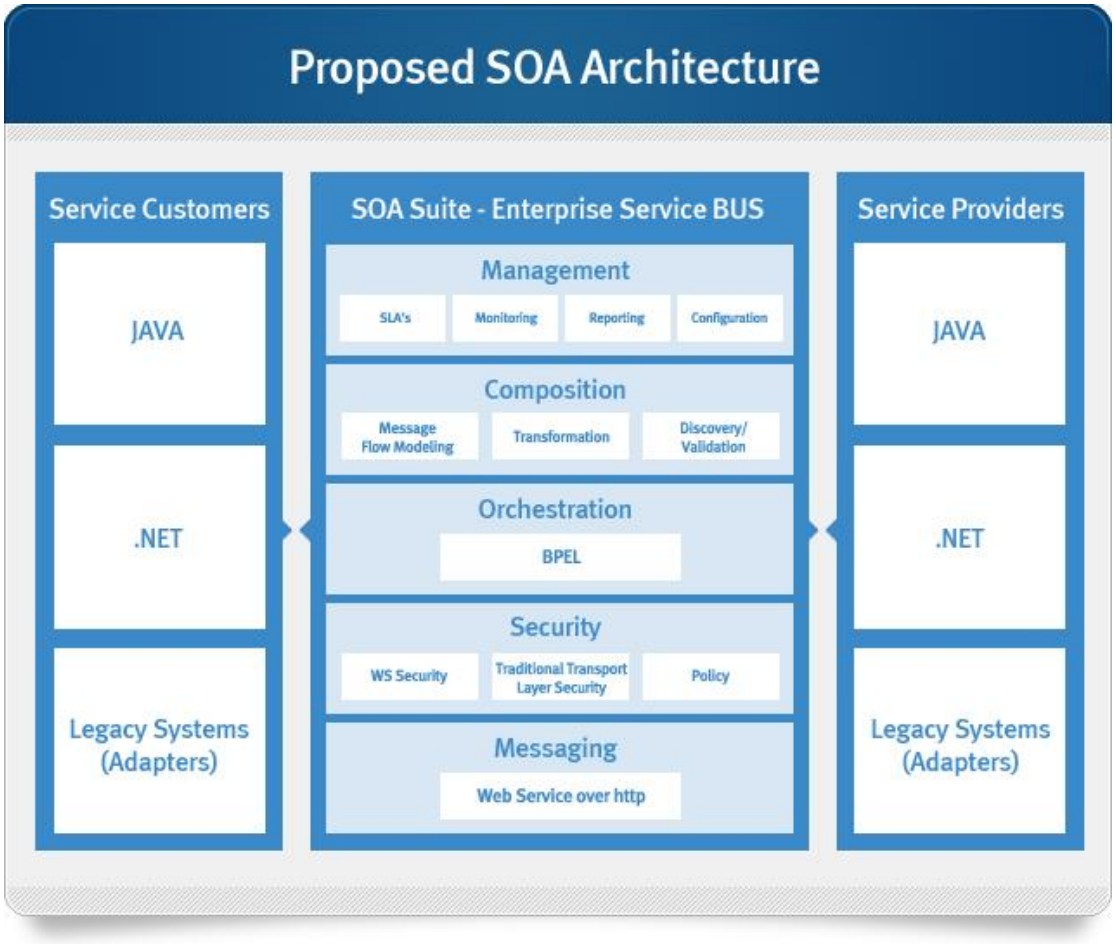


Рис. 4 Один из вариантов SOA-архитектуры [9]

На приведённом рисунке видно, что ESB не зависит от технологий, на которых реализованы подключаемые сервисы. А для legacy systems создаются адаптеры. Сама ESB выполняет функции управления (мониторинга), преобразования сообщений, контроль безопасности и маршрутизации сообщений.

Стоит отметить, что ESB часто ассоциируется с SOA и наоборот. Но включать ESB в само понятие SOA ошибочно, т.к. SOA – это лишь набор принципов, которым должны удовлетворять сервисы, в то время как ESB – это инструмент, с помощью которого можно связать эти сервисы. Более того, микросервисная архитектура вообще отрицает ESB из-за её сложности. Считается, что микросервисы должны отправлять сообщения друг другу напрямую по стандартным и, главное, очень простым и легким протоколам.

4. Enterprise Application Integration

Говоря о ESB нельзя не сказать об Enterprise Application Integration (EAI, интеграция приложений предприятия). EAI – это общая категория подходов для создания совместимости между различными несвязанными системами, которые составляют типичную инфраструктуру организации [10]. Обычно это понятие применяется тогда, когда в организации уже используются системы, которые не созданы для взаимной интеграции, но интеграция которых повысит общую эффективность работы организации. В этом случае реализуют промежуточные системы, которые транслируют сообщения между целевыми системами. Т.к. реализовывать связи между каждой парой систем трудозатратно (особенно если количество этих систем велико), то зачастую применяют ESB для уменьшения количества связей и количества адаптеров.

3. Описание разработанного решения

1. Межсервисное взаимодействие

Разрабатываемые в данной работе компоненты корпоративной информационной системы были разбиты на сервисы в соответствии с архитектурой SOA. На рисунке показаны не все компоненты информационной системы, а только те, в разработке которых принимал участие автор данной работы. Другие компоненты подключаются к указанным через адаптеры, которые реализуются отдельно для каждого конкретного случая.

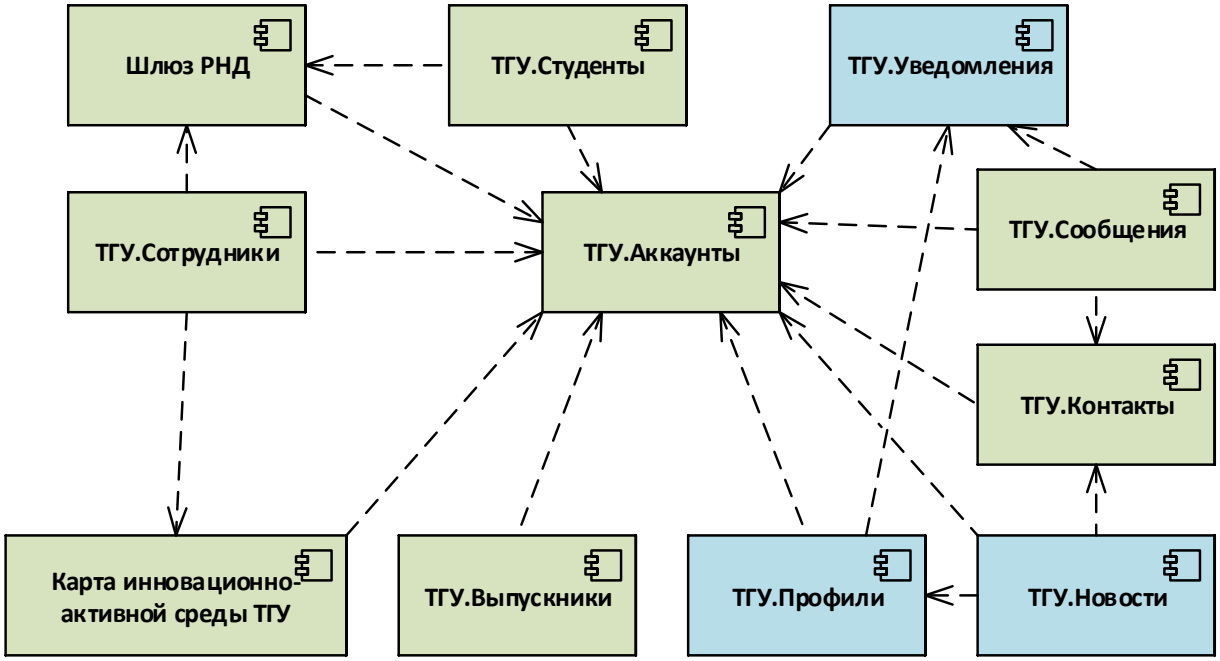


Рис. 5 Структура разрабатываемых компонентов. Зелёным обозначены компоненты, создаваемые исключительно автором данной работы. Синим показаны компоненты, проектируемые и контролируемые автором работы, но реализуемые при поддержке других разработчиков.

Несмотря на то, что сервисы в SOA разрабатываются и разворачиваются практически независимо друг от друга, тем не менее они интенсивно взаимодействуют, обмениваются данными. Взаимодействие сервисов должно осуществляться по стандартным протоколам. В данной работе использовались два основных способа: HTTP API и очередь сообщений, которая реализует протокол, AMQP.

1. HTTP API

HTTP API – наиболее простой и распространённый способ обмена данными между сервисами. Клиенты HTTP имеются в большинстве платформ, фреймворков, языков. Web-сервис сам по себе без дополнительных инструментов может предоставлять API, но практически в каждом web-фреймворке реализованы инструменты для упрощения процесса создания API.

Сервисы в данной работе реализованы на фреймворке ASP.NET [11], который в свою очередь основан на фреймворке .NET [12], в котором реализован HTTP-клиент. Также частью ASP.NET является фреймворк Web API [13], который значительно упрощает создание API, в том числе в формате RESTful API [14].

Однако, на чём бы ни были реализованы другие сервисы системы, взаимодействие по протоколу HTTP всегда будет простым. Но есть и обратная сторона: HTTP-запросы сравнительно медленны. Причем большая часть времени тратится не на полезную нагрузку (формирование и обработка данных, их сериализация, передача, десериализация), а на открытие и закрытие соединения, т.к. на каждый HTTP-запрос открывается и закрывается новое TCP-соединение.

Рассмотрим пример. В сервисе ТГУ.Сотрудники необходимо выводить список имён всех сотрудников отдела университета. В отделе могут работать десятки сотрудников. Но имя каждого сотрудника хранится не на ТГУ.Сотрудники, а на сервисе ТГУ.Аккаунты, как и все остальные данные об аккаунтах каждого пользователя системы. Чтобы вывести список имён сотрудников отдела, необходимо для каждого сотрудника отправить HTTP-запрос на ТГУ.Аккаунты для получения имени. Один запрос занимает в среднем 25мс. Выполняя запросы последовательно друг за другом при условии, что в отделе 100 сотрудников, понадобится 2500мс на то, чтобы получить все имена. Можно распараллелить запросы в нескольких потоках.

Был проведён опыт, и в этом случае общее время всех запросов стало равно 941мс.

Для многих сайтов выдвигается требование, чтобы среднее время обработки сервером запроса для одной страницы не превышало 100мс, а максимальное время обработки не превышало 500мс. Это же требование выдвигалось и ко всем разрабатываемым в данной работе сервисам. Очевидно, что, посылая для получения каждого имени отдельный HTTP-запрос, ограничение соблюсти невозможно.

Ещё одним способом ускорить обмен данными является изменение API на стороне сервиса ТГУ.Аккаунты таким образом, чтобы не приходилось создавать новый HTTP-запрос для получения каждого имени, а достаточно было бы отправить только один HTTP-запрос и получить все необходимые данные в ответе. Этот вариант полностью решает проблему долгого получения данных. В приведённом примере вместо 100 HTTP-запросов необходимо отправить только 1, а время обработки одного запроса увеличивается всего лишь до 30мс.

Но этот способ порождает новые проблемы. Для его реализации приходится подстраивать API всех сервисов под конкретные нужны каждого другого сервиса. При условии, что список сервисов в системе открытый (условно неограниченный), пришлось бы потратить слишком много ресурсов на создание и поддержку API. По сути для каждого зависимого сервиса пришлось бы писать своё собственное API. В целях экономии ресурсов было решено разрабатывать максимально простое и максимально универсальное API, которое могло бы быть использовано любыми другими сервисами для любых целей, даже если для достижения этих целей придётся сделать несколько HTTP-запросов.

Ещё одной особенностью, которую можно считать недостатком, является то, что после отправки HTTP-запросов необходимо ждать ответ сервера, который приходит только после полной обработки этого запроса. Это не всегда удобно. Иногда ответ сервера не требуется, нужно лишь передать

данные для последующей обработки. Например, в сервисе ТГУ.Выпускники реализуется массовая рассылка писем. Рассылка писем в системе осуществляется централизованно, через ТГУ.Аккаунты. При отправке писем от ТГУ.Выпускники к ТГУ.Аккаунты нет необходимости ждать подтверждения, что это письмо отправлено на почтовый сервер. Достаточно просто отправить данные. Эту и другие проблемы может решить очередь сообщений.

2. Очередь сообщений

Другим способом взаимодействия между Web-сервисами является очередь сообщений (Message Queue, MQ). Очередь сообщений – это целый класс программных средств, которые выполняют одну задачу: принимать информационные сообщения от одних программ и передавать эти сообщения другим программам в порядке FIFO (понятие очереди в классическом смысле: первое пришедшее от отправителя сообщение отправляется получателю также первым).

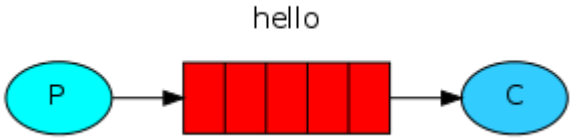


Рис. 6 Очередь сообщений с одним отправителем и одним получателем

На рисунке овал P означает сервис отправителя (producer), овал C – сервис получателя (consumer), красные прямоугольники – сообщения в очереди сообщений, стрелки – передачу сообщений.

Помимо задачи непосредственно передачи сообщения очередь сообщений также выполняет задачу маршрутизации сообщений и, как следствие, задачу балансировки нагрузки.

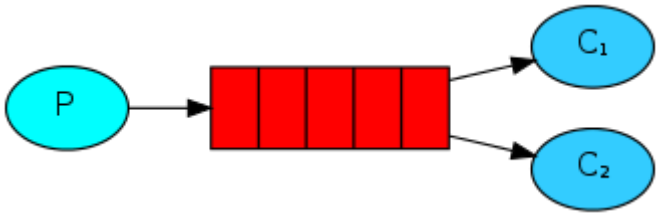


Рис. 7 Очередь сообщений с двумя получателями, которые получают сообщения поочередно, тем самым происходит простейшая балансировка нагрузки.

Многие из таких программных средств реализуют стандартный протокол AMQP (Advanced Message Queuing Protocol). AMQP – открытый стандарт для передачи сообщений между приложениями или организациями [15].

В рамках данной работы в качестве очереди сообщений был выбран RabbitMQ [16]. Он является типичным представителем этого класса программных средств, реализует все стандартные возможности очереди сообщений, реализует протокол AMQP, является одним из самых распространённых средств в своём классе.

Как уже было указано ранее, все сервисы в рамках данной работы реализуются на платформе .NET. На этой платформе имеется несколько клиентов для RabbitMQ. Среди них был выбран EasyNetQ [17], т.к. в нём наиболее просто реализована сериализация и десериализация объектов. В общем случае сообщения в очереди сообщений представляют собой текст. Однако интерфейс этого клиента разработан таким образом, что отправитель посылает в очередь сообщений объект некоторого класса, а получатель принимает этот объект того же класса, при этом в коде всё это выглядит очень компактно.

Все дальнейшие примеры будут основаны на реализации RabbitMQ, однако с некоторыми поправками могут быть распространены и на другие очереди сообщений.

Вернёмся к примеру про массовую рассылку писем, который был описан выше. В этом примере нам не нужно было ждать подтверждения об успешной отправке письма на почтовый сервер, а достаточно было просто отправить

данные. Очередь сообщений с этим справляется хорошо. Сообщение с данными отправляются в очередь и на этом связь отправителя и сообщения теряется. Отправитель работает дальше, а сообщение ждёт своей очереди, когда получатель сможет его обработать. Таким образом, если обработка сообщений занимает больше времени, чем их формирование, то очередь сообщений будет являться сглаживающим буфером, который позволит отправителю максимально быстро сформировать все необходимые сообщения и продолжить работу, а получателю позволит не потерять сообщения даже на пиковых нагрузках.

Однако, если всё-таки отправителю нужно получить какой-то ответ от получателя, необходимо использовать более сложную конфигурацию очереди сообщений, которая реализует по сути удалённый вызов процедур (RPC). Участвующие сервисы рассматриваются как клиент и сервер. Клиент отправляет сообщение-запрос в одну очередь, помечая его необходимым идентификатором. Сервер получает это сообщение, обрабатывает и отправляет сообщение-ответ в другую очередь, добавляя к нему идентификатор запроса. Клиент получает сообщение-ответ и сопоставляет его по идентификатору с запросом.

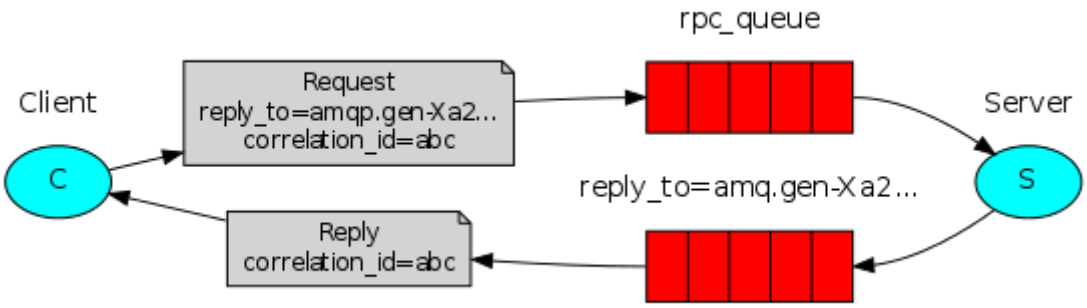


Рис. 8 Очередь сообщений в режиме RPC

Естественно, отправку запросов и приём сообщений в этом случае можно распараллелить, а серверов можно поставить не один, а несколько, чтобы распределить между ними запросы.

Кроме того, очередь сообщений более устойчива к недоступности сервиса. Далее приводится график, на котором изображена ситуация при перезагрузке сервиса, когда он был недоступен в течении нескольких десятков секунд. На оси ординат изображено количество сообщений в очереди в указанный момент времени. Здесь мы видим, что при недоступности сервиса сообщения накапливались в очереди, но не терялись. После возобновления работы сервиса он их сразу получил и обработал.

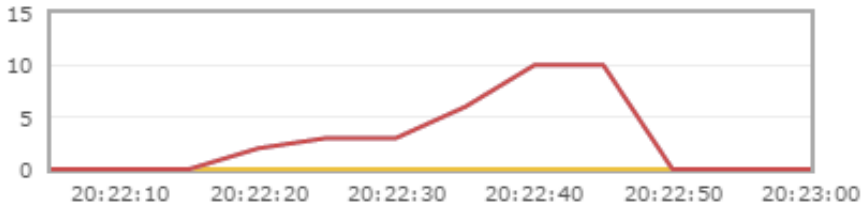


Рис. 9 Количество сообщений в очереди при кратковременной недоступности сервиса

Теперь посмотрим на скорость работы. Для этого был проведён опыт. Было отправлено 1000 запросов по HTTP и 1000 запросов по RabbitMQ в режиме RPC. Запросы отправлялись последовательно друг за другом, без распараллеливания, чтобы исключить возможное влияние систем обеспечения параллельного выполнения. Результаты опыта представлены на диаграмме. Очередь сообщений оказалась примерно в 2 раза быстрее, чем HTTP.

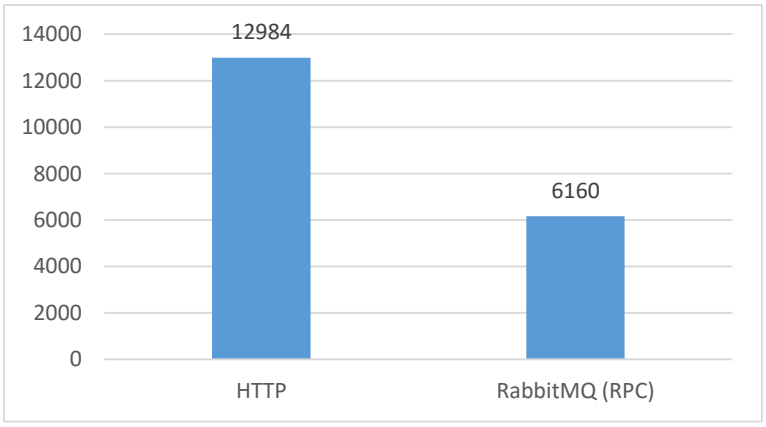


Рис. 10 Время отправки запросов и получения ответов по HTTP и RabbitMQ. Время указано в мс.

Однако у очереди сообщений есть и недостаток, который заключается в том, что её сложнее использовать. Для RabbitMQ имеется множество клиентов

для различных платформ, но они не встроены во фреймворки, и они не согласуются друг с другом. Сообщение, сформированное и отправленное одним клиентом может быть не прочитано другим клиентом. Очереди сообщений менее популярны, у сообщества меньше знаний о них. В то же время различные сервисы системы разрабатываются на различных платформах и различными командами, которые фактически не связаны друг с другом.

Для облегчения такого рода взаимодействия в данной работе было решено поддерживать оба способа. Это означает, что одна и та же функция доступна как по протоколу HTTP, так и через RabbitMQ.

На сервисах, создаваемых в рамках данной работы, в общем случае действует алгоритм, изображённый на Рис. 11. На рисунке видно, что вначале происходит попытка найти ответ на запрос в кэше. Если в кэше ответа нет, то происходит попытка отправить запрос через очередь сообщений. Если эта попытка оказалась неудачной, то происходит попытка отправить запрос через HTTP. Если всё-таки ответ получить удалось, то проверяется, необходимо ли сливать аккаунты. Подробнее о слиянии аккаунтов рассказано в той части этой главы, которая посвящена единому центру аутентификации. Если аккаунты необходимо слить, то они сливаются и весь алгоритм запроса начинается сначала, но уже с параметрами нового аккаунта.

Таким образом, если очередь сообщений по каким-то причинам перестанет работать, то сервисы продолжат взаимодействовать друг с другом, но по более медленному протоколу HTTP, тем самым надёжность системы в целом повышается.

Здесь описан общий вид алгоритма отправки запроса и обработки ответа, использующий два протокола и кэш. Однако на практике не во всех случаях имеется возможность использовать оба протокола. Иногда определённый запрос доступен только через HTTP, иногда только через очередь сообщений. Для этих случаев в алгоритме предусмотрена вариативность: можно для каждого запроса в отдельности указать только путь

через HTTP или только путь через очередь сообщений. Кроме того, время жизни кэша для каждого из запросов должно подбираться индивидуально. К примеру, имя человек меняет не часто, поэтому кэш запроса на получение имени можно установить достаточно большим (например, сутки), а запрос на наличие новых уведомлений кэшировать вообще нельзя, т.к. уведомления должны доходить до пользователя максимально быстро. Для этих целей в алгоритме отправки запросов для каждого запроса предусмотрено указание, нужно ли использовать кэш, и, если нужно, то каково будет его время жизни.

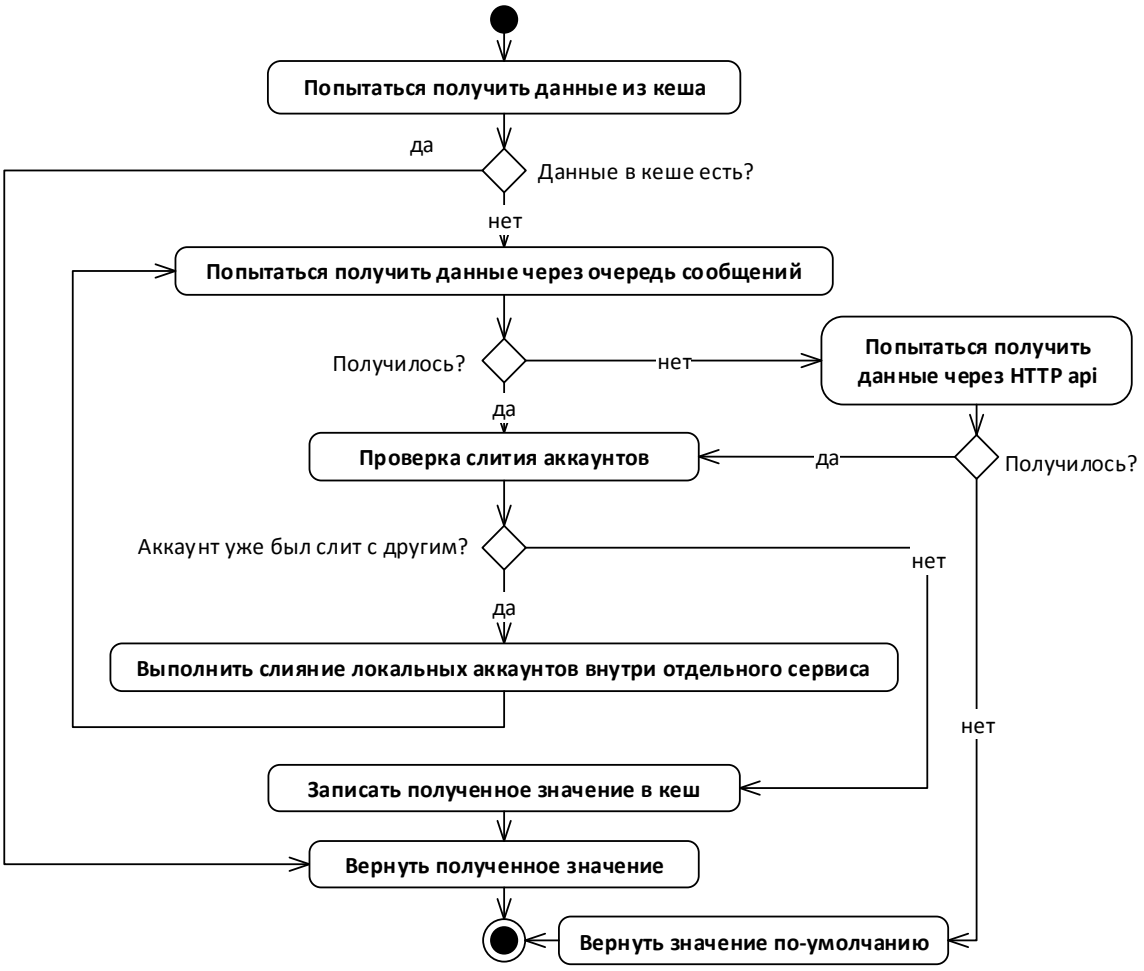


Рис. 11 Алгоритм отправки запроса к другому сервису и обработки ответа.

3. Выделение интерфейсных компонентов

Рассмотрим следующую ситуацию. Имеется два сервиса Customer1 и Customer2. Оба они используют одно и то же API третьего сервиса DataOwner.

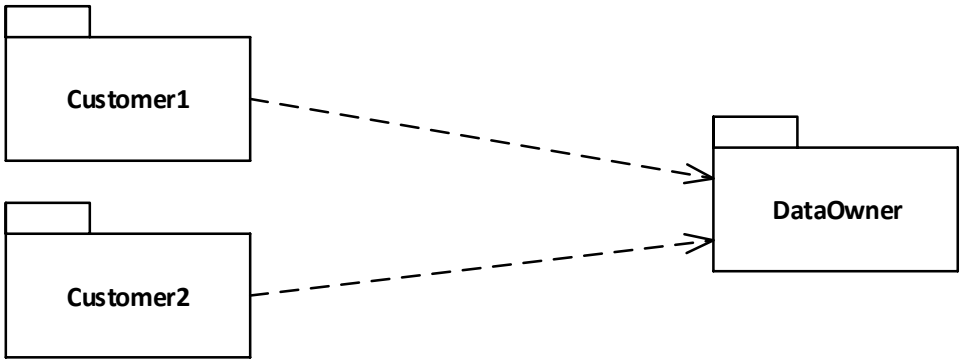


Рис. 12 Два сервиса используют API третьего сервиса

В этом случае всю логику отправки запросов и получения ответов (алгоритм описан выше) приходится дублировать в каждом из сервисов Customer1 и Customer2. Чтобы этого избежать, вся описанная логика была вынесена в отдельную библиотеку, которая подключается к каждому сервису, использующему это API, а также к сервису, который это API предоставляет (реализовано типовое решение Шлюз [18]).

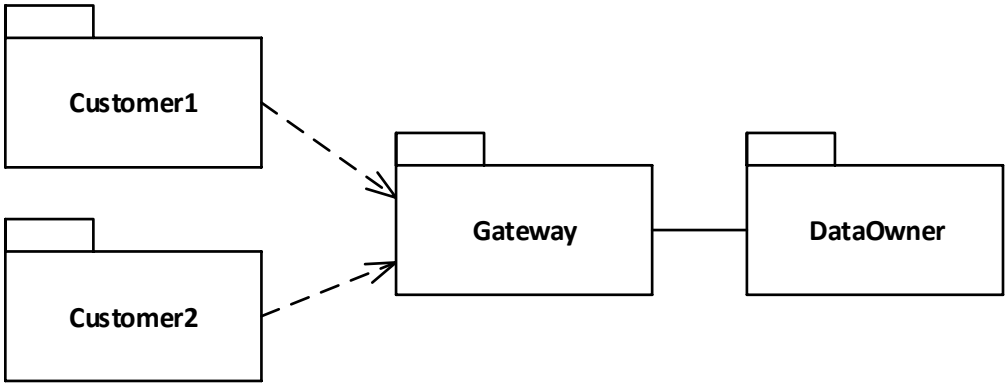


Рис. 13 Выделение логики отправки запросов и обработки ответов в отдельную библиотеку.

Рассмотрим следующую ситуацию. Теперь у нас есть два сервиса, предоставляющих каждый своё API: DataOwner1 и DataOwner2. Для каждого из этих сервисов создана своя собственная библиотека, обеспечивающая отправку запросов к API: Gateway1 и Gateway2. В этом случае снова возникает дублирование кода: один и тот же алгоритм используется в разных библиотеках, за исключением лишь деталей реализации API. Чтобы этого избежать, была создана библиотека BaseGateway, которая реализует типовой шаблон Супертип слоя [18]. В этой библиотеке содержится общая логика отправки запросов и обработки ответов, а детали реализации API разных

сервисов остаются в Gateway1 и Gateway2, которые наследуются от BaseGateway.

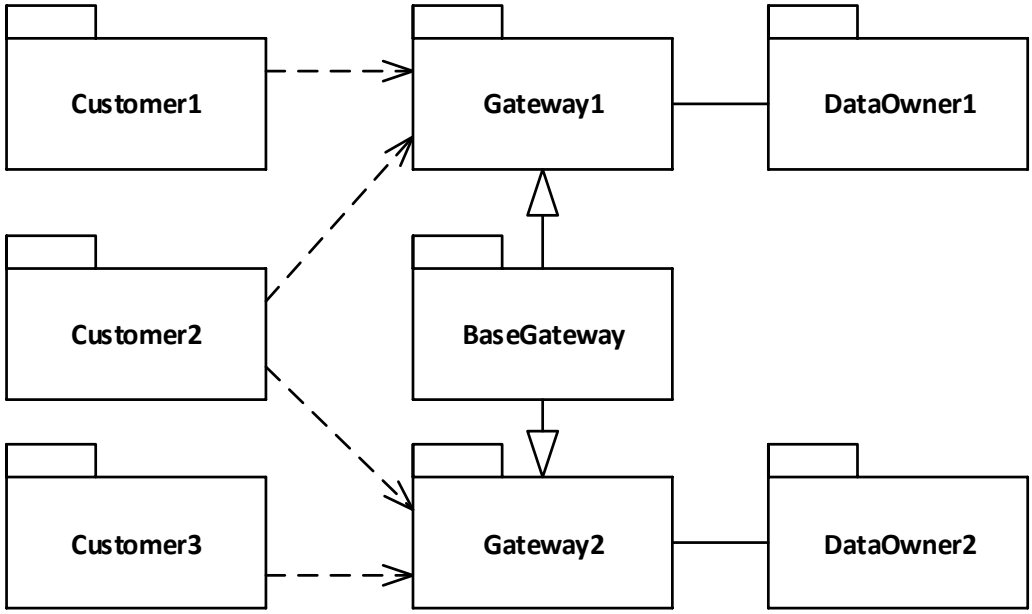


Рис. 14 Добавление супертипа слоя для библиотек, реализующих отправку и обработку запросов

Таким образом мы полностью исключили дублирование кода и во всех сервисах используем одну и ту же логику.

Такая организация кода имеет ещё одно преимущество. Как было указано ранее, для взаимодействия через RabbitMQ используется клиент EasyNetQ. Этот клиент сериализует объект некоторого класса, отправляет в текстовом виде в очередь сообщений, а на другой стороне принимает и десериализует этот объект в тот же класс. Проблема в том, что, чтобы сериализация, десериализация, отправка и приём сообщений работали корректно, необходимо, чтобы на одной и на другой стороне были одни и те же классы, в том числе и полные названия этих классов, включая имя сборки. Этого добиться можно лишь в том случае, если эти классы будут взяты из одной библиотеки, которая подключается к обоим сервисам: и к отправителю, и к получателю.

2. Единый центр аутентификации ТГУ.Аккаунты

Различные сервисы корпоративной информационной системы должны быть связаны общими аккаунтами. Если у пользователя уже есть аккаунт на одном сервисе, то, очевидно, этот же пользователь должен иметь возможность зайти под этим же аккаунтом и на другом сервисе. И с другой стороны: сервисы должны обмениваться между собой данными, в том числе и данными по конкретным пользователям. Чтобы синхронизировать данные о пользователях, необходимо иметь общие аккаунты на все сервисы. Для этих целей был создан единый центр аутентификации ТГУ.Аккаунты.

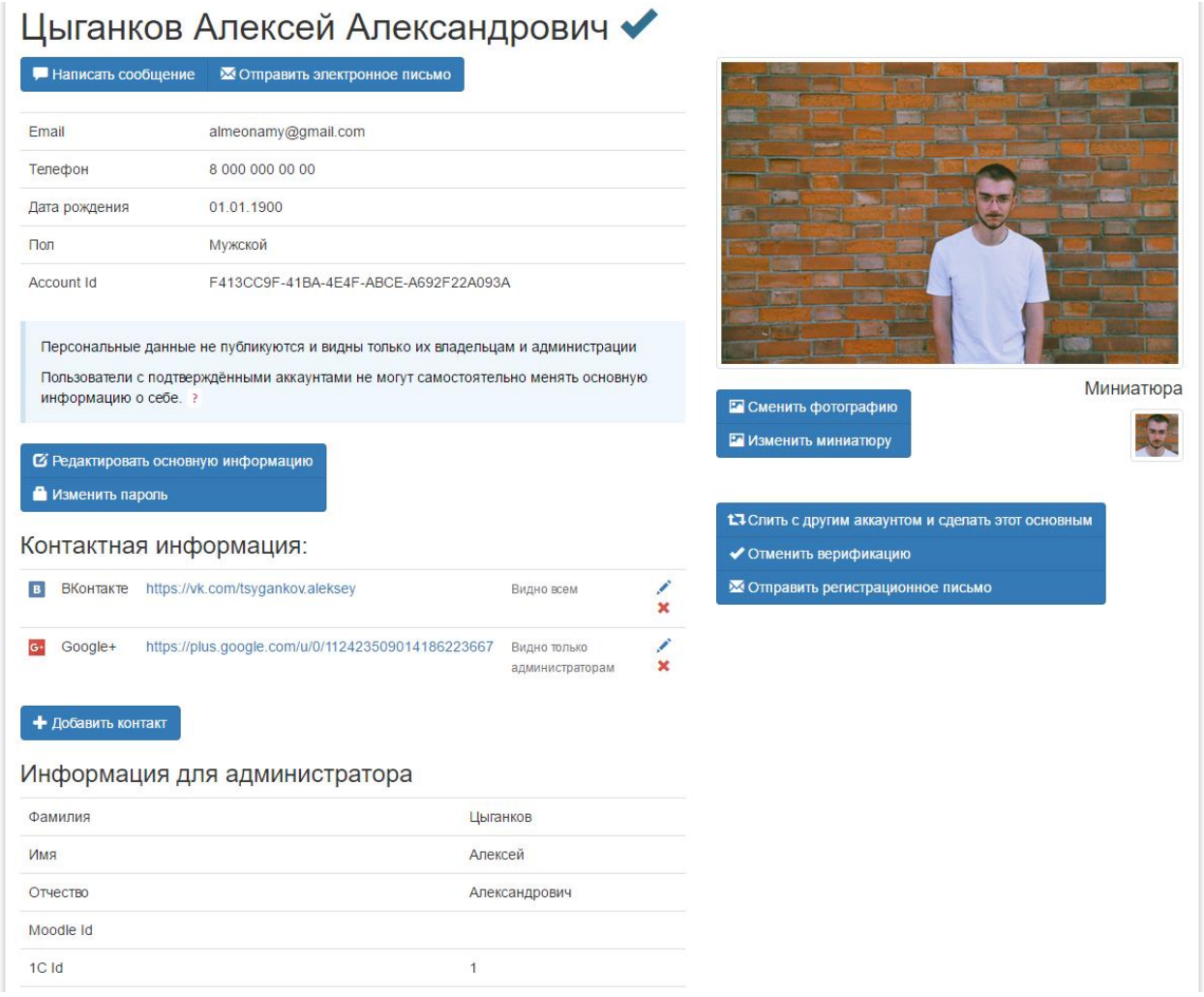


Рис. 15 Пример страницы аккаунта с основными данными с точки зрения администратора системы.

На этом сервисе хранятся все логины и пароли пользователей, а также общая информация, не привязанная ни к одному из остальных сервисов: фотография, контактные данные, идентификаторы в различных сервисах.

Кроме того, на некоторых сервисах необходима возможность удостовериться, что доступ к аккаунту имеет именно тот пользователь, о котором имеется информация. Для этих целей была создана система аутентификации. Пользователи подтверждают свои данные и единоличный доступ к аккаунту с помощью бумажного заявления. После этого аккаунт считается подтверждённым. Любой другой сервис может проверить любой аккаунт на подтверждённость и в зависимости от этого предоставлять пользователю какие-либо функции или нет. Например, на сервисе ТГУ.Сотрудники имеется возможность сотрудникам университета просмотреть информацию о своей заработной плате. Но в целях безопасности эта информация предоставляется только тем, кто подтвердил свой аккаунт.

Аутентификация на других сервисах осуществляется по протоколу OAuth. Для реализации этого протокола сначала регистрируются все сервисы системы, которые должны иметь возможность проходить аутентификацию через ТГУ.Аккаунты. Зарегистрировать свой сервис может любой, отправив заявку. После регистрации сервис получает свой идентификатор. Далее этот сервис реализует алгоритм, приведённый на рисунке. Далее приведён полный алгоритм действий по шагам (* звёздочками помечены шаги, которые необходимо реализовать на стороне подключаемого сервиса).

1. Пользователь кликает по ссылке "Выполнить вход". Запрос отправляется на сервер сервиса.
2. * Сервис возвращает перенаправление на страницу https://accounts.tsu.ru/Account/Login2/?applicationId=... Здесь необходимо указать идентификатор сервиса, который будет выдан при регистрации.
3. Пользователю показывается страница с полями для ввода логина и пароля.

4. Пользователь вводит свои логин и пароль, отправляет на сервер ТГУ.Аккаунты, где происходит их проверка.
5. ТГУ.Аккаунты возвращает перенаправление на страницу сервиса, которая указывается при регистрации. К адресу этой страницы добавляется временный token, который представляет собой строку из 255 символов.
6. * Сервис должен обменять временный token на AccessToken и AccountId. Для этого он направляет прямой POST-запрос (без участия клиента) на адрес https://accounts.tsu.ru/api/Account/?token=...&applicationId=...&secretKey=... Здесь необходимо будет указать временный token, который был получен на предыдущем шаге, идентификатор сервиса и секретный ключ, которые выдаются при регистрации. Секретный ключ используется для проверки подлинности сервиса.
7. В ответ ТГУ.Аккаунты возвращает AccessToken (строка из 255 символов) и AccountId (GUID) аутентифицированного пользователя. AccountId - это глобальный идентификатор пользователя, который используется в ТГУ.Аккаунты и других связанных с ним сервисах.
8. * Реализация этого шага напрямую зависит от целей и особенностей сервиса. Здесь приведены рекомендации для типичного использования. Полученные данные необходимо сохранить. По AccountId необходимо найти локального пользователя и считать его аутентифицированным. Если пользователь не найден среди локальных, то его необходимо зарегистрировать. После этого можно вернуть в браузер перенаправление на страницу пользователя.

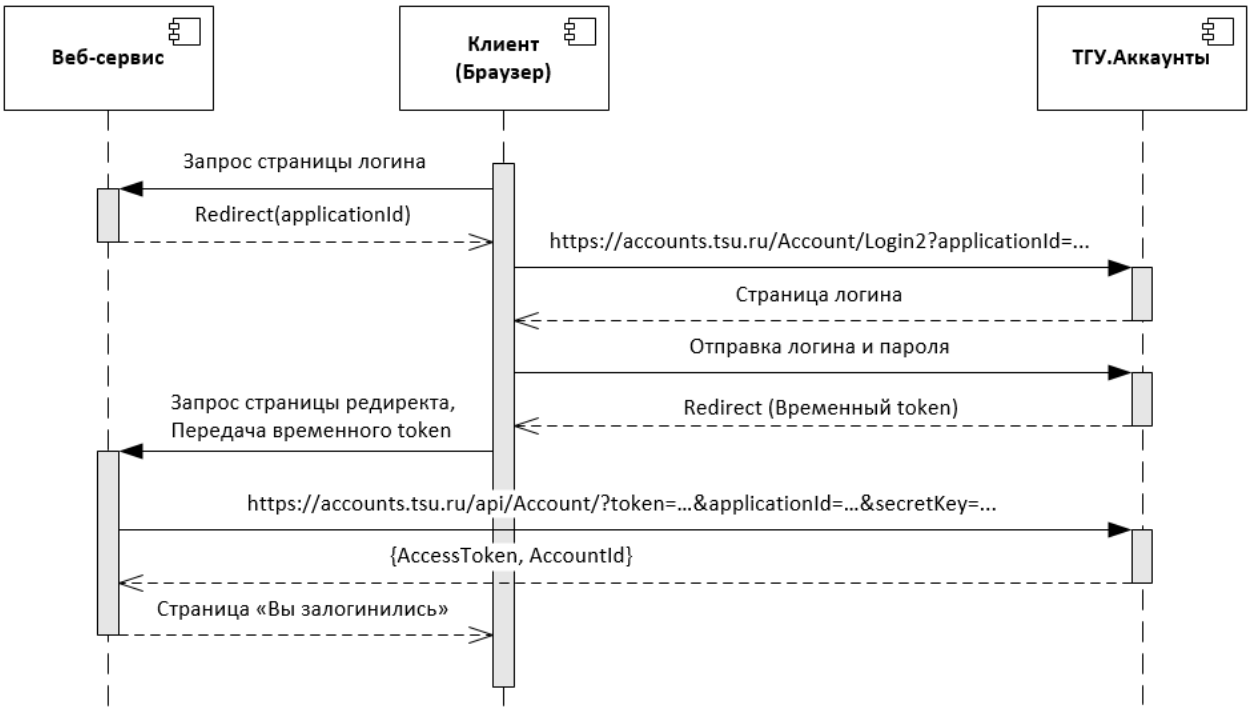


Рис. 16 Процесс аутентификации через сервис ТГУ.Аккаунты

Одним из ключевых принципов протокола OAuth является то, что логин и пароль пользователя хранятся и вводятся только в центре аутентификации. Ни один другой сервис ни при каких обстоятельствах не получает доступа к паролю пользователя, этим обеспечивается безопасность системы.

Большинство аккаунтов были зарегистрированы автоматически. Основными источниками послужили база данных отдела кадров с полным списком сотрудников, различные списки студентов и выпускников университета. Зачастую один и тот же человек попадал в несколько списков. Например, человек одновременно может являться выпускником бакалавриата, студентом магистратуры и сотрудником университета. Более того, различные списки студентов и выпускников были не согласованы, и один и тот же человек мог попасть в роли студента в два списка. Кроме того, пользователи не всегда подозревали о том, что для них уже автоматически зарегистрирован аккаунт, и пытались зарегистрировать себе аккаунт вручную. Всё это приводило к огромному количеству дубликатов – аккаунтов, принадлежащих одному пользователю. В условиях корпоративной информационной системы это недопустимо.

Для устранения этой проблемы была реализована возможность сливать аккаунты. В случае, если администратор системы находит дубликаты, он помечает один аккаунт как основной, а другой – дублирующим. При этом дублирующий аккаунт не удаляется совсем, т.к. на него могли ссылаться другие сервисы, но он удаляется из публичных списков. Если пользователь попытается зайти под дублирующим аккаунтом, система перенаправит его на основной. Если на основном аккаунте не хватало каких-то данных (фотографии, контактов), но эти данные были на дублирующем аккаунте, то они копируются на основной.

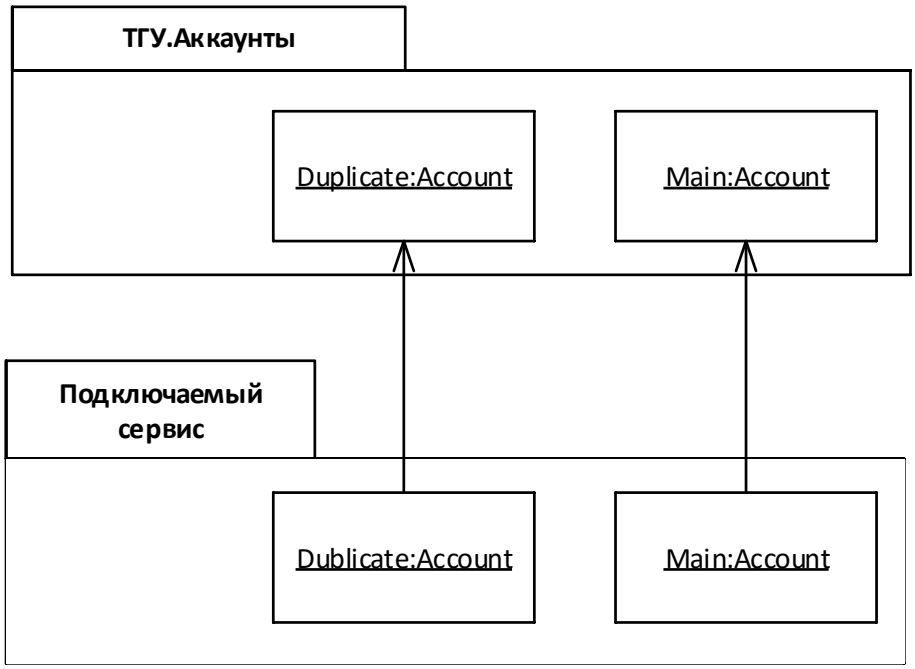


Рис. 17 Основной и дублирующий аккаунты на разных сервисах

Однако аккаунты необходимо сливать на всех сервисах, где они были зарегистрированы, но ТГУ.Аккаунты не могут делать это напрямую. Каждый сервис должен сам реализовать логику слияния аккаунтов в соответствии со своей спецификой. Если сервис посылает на ТГУ.Аккаунты запрос, касающийся дублирующего аккаунта, то система возвращает HTTP статус 498, означающий, что аккаунты слиты. При этом в теле ответа содержится идентификатор основного аккаунта. После этого сервис должен слить свои локальные аккаунты, используя старый и новый идентификаторы. Затем можно повторить запрос к ТГУ.Аккаунты, используя уже новый

идентификатор аккаунта. Подробно алгоритм отправки запроса и обработки ответа изображён на Рис. 11.

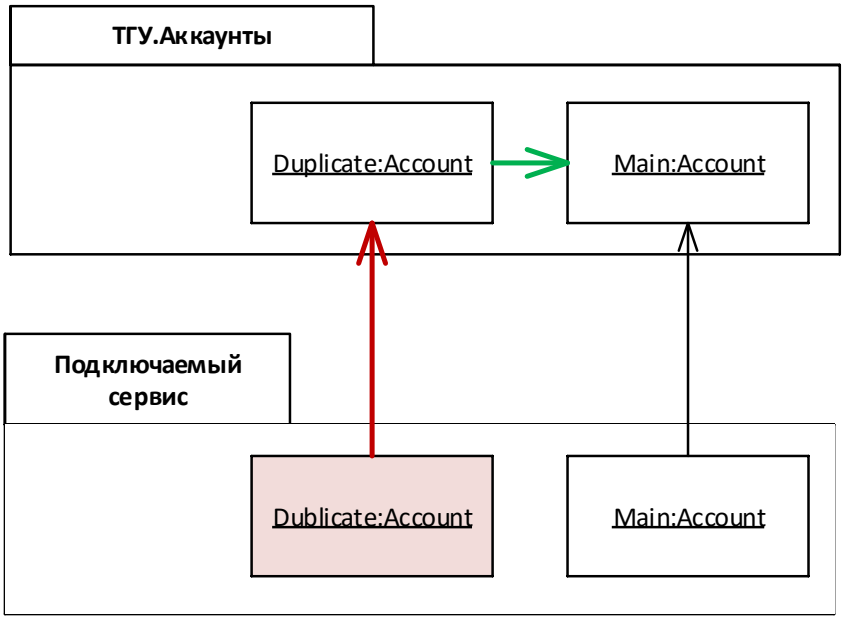


Рис. 18 Слитые аккаунты в разных сервисах. Красным помечены удаляемые элементы, зелёным - добавляемые. В зависимости от реализации алгоритма слияния в подключаемом сервисе данные из удаляемого объекта могут быть перекопированы в объект основного аккаунта.

Стоит отметить, что вся логика по обработке ответа ТГУ.Аккаунты о том, что нужно слить аккаунты, находится в одном месте, в библиотеке BaseGateway (Рис. 14), и используется повторно в каждом из подключаемых сервисов. Но логика непосредственно слияния реализуется в самом сервисе и зависит от его специфики.

3. ТГУ.Сотрудники

ТГУ.Сотрудники – это сервис, представляющий из себя каталог всех сотрудников университета с основной информацией о них. Источником данных является официальная информационная система отдела кадров. На сервисе не предусмотрена ручная регистрация. Регистрируются только те аккаунты, которые есть в информационной системе отдела кадров. Если из этой системы кто-то удаляется, он автоматически удаляется и с сервиса ТГУ.Сотрудники.

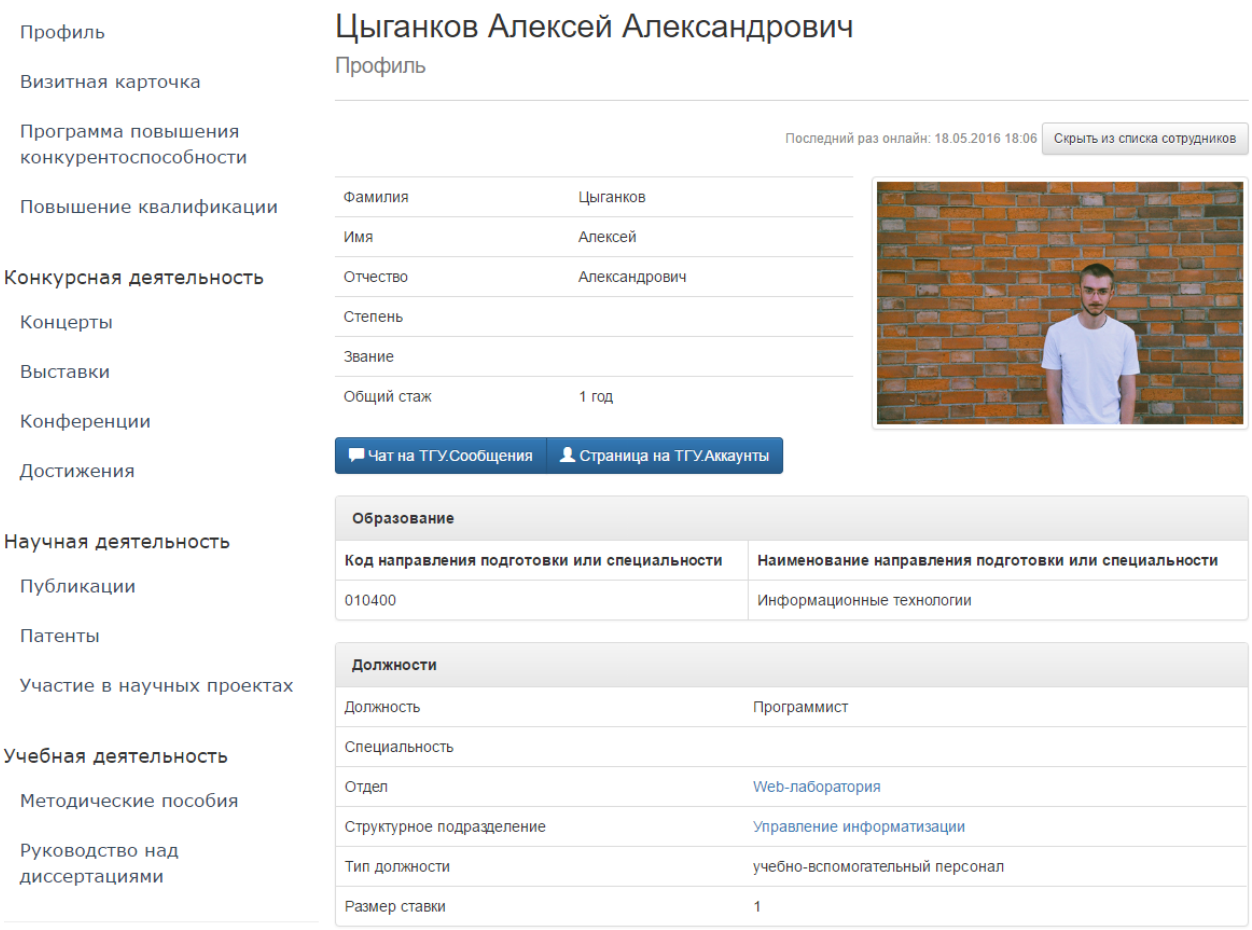


Рис. 19 Пример страницы сотрудника с основными данными о нём.

Однако информационная система отдела кадров не разрабатывается в данный момент и внести изменения в её код возможности нет, поэтому реализовать свои, наиболее подходящие способы синхронизации данных невозможно. Единственный способ синхронизации данных, который был предусмотрен в этой системе – это ручной экспорт xml-файлов с полными данными о каждом сотруднике и его должностях. Было решено раз в месяц получать такой файл и импортировать эти данные на ТГУ.Сотрудники. Для этого было реализовано типовое решение Адаптер [18].

При импорте данных очередного сотрудника могут возникать следующие ситуации.

1. Сотрудник ещё не имеет аккаунта ни на ТГУ.Сотрудники, ни на ТГУ.Аккаунты.
2. Сотрудник уже имеет аккаунт на ТГУ.Аккаунты, но не имеет на ТГУ.Сотрудники.

3. Сотрудник уже имеет аккаунт и на ТГУ.Сотрудники, и на ТГУ.Аккаунты.

Также возможен дополнительный вариант.

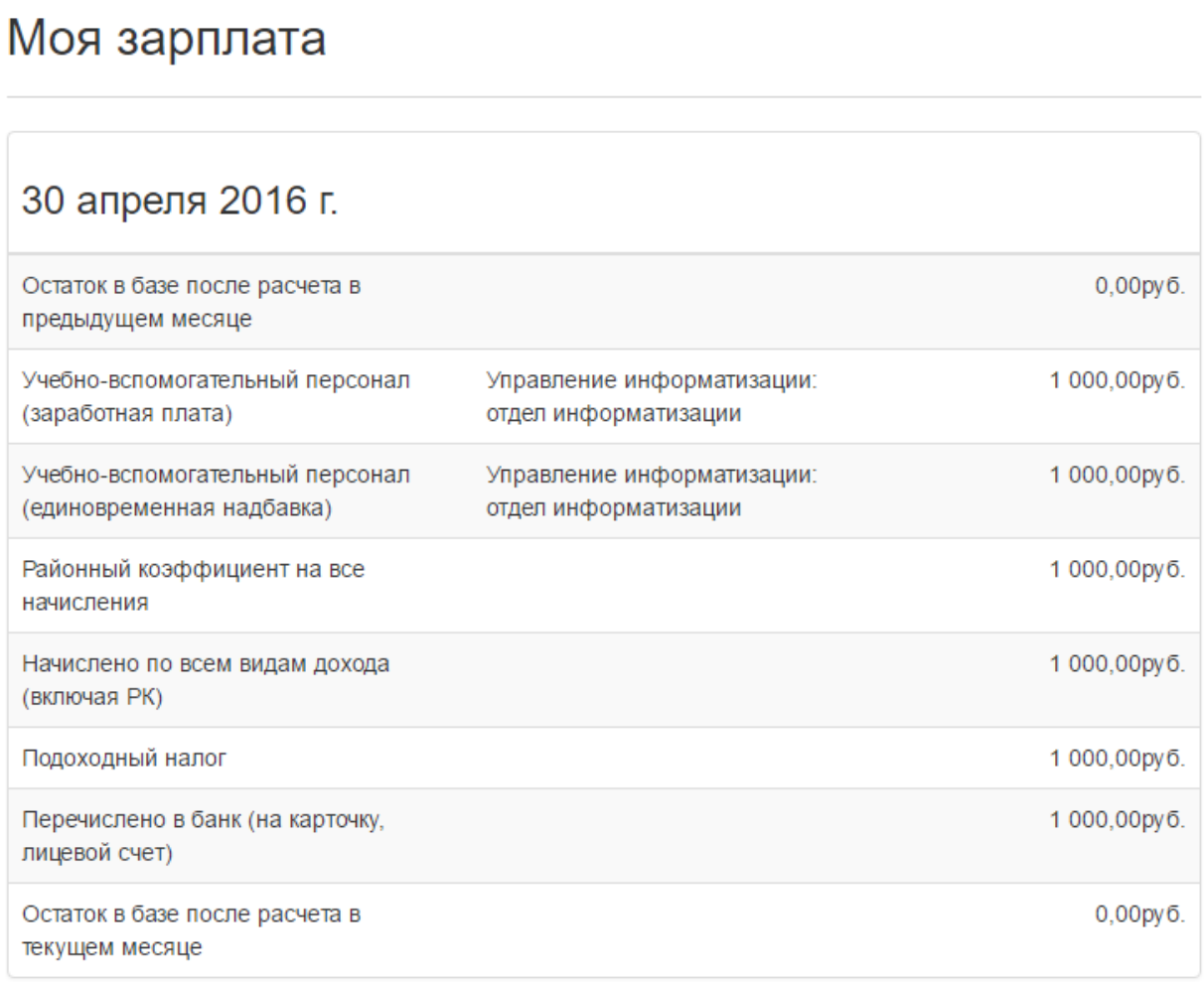
4. У пользователя есть аккаунт и на ТГУ.Сотрудники, и на ТГУ.Аккаунты, но его нет в информационной системе отдела кадров, т.е. он не является сотрудником университета.

Для начала необходимо определить, имеется ли аккаунт у сотрудника на ТГУ.Аккаунты. Поиск соответствующего аккаунта происходит по полному совпадению имени, фамилии, отчества и даты рождения. Если такого аккаунта нет, то он регистрируется. Если такой аккаунт найден, берётся его глобальный идентификатор AccountId и по нему ищется локальный аккаунт на ТГУ.Сотрудники. Если локальный аккаунт найден, то проверяются и обновляются все данные о нём. Если локальный аккаунт не был найден, то он регистрируется. Для того, чтобы удалять локальные аккаунты пользователей, которые перестали быть сотрудниками, перед процедурой импорта все локальные аккаунты помечаются флагом IsApproved = false. После того, как проверен очередной сотрудник и для него определен локальный аккаунт, для него этот флаг меняется на IsApproved = true. Таким образом после процедуры импорта необходимо удалить все локальные аккаунты, у которых флаг IsApproved = false. В итоге данные на сервисе ТГУ.Сотрудники полностью синхронизируются с информационной системой отдела кадров.

Ещё одной задачей на сервисе ТГУ.Сотрудники стала синхронизация данных о заработной плате сотрудников. Источником для этих данных является информационная система бухгалтерии университета. Также, как и в отделе кадров, эта система не разрабатывается в данный момент и нет возможности внести изменения в её код. Единственным способом получить из неё данные является ручной экспорт текстового файла, содержимое которого представлено в своём собственном формате, не предназначенном для машинного чтения, а не на одном из формальных языков. Кроме того, часть данных в этом файле (а именно статьи доходов/расходов и названия отделов)

представлены в зашифрованном виде. Расшифровать данные можно с помощью вспомогательных данных, содержащихся в файлах в формате .xls, сопоставив коды. Файлы .xls также не были предназначены для машинного чтения, хотя строго соответствовали своему формату.

Для решения задачи импорта данных о заработной плате был реализован адаптер, который читает и анализирует файл с данными о зарплате, а также вспомогательные файлы, сопоставляет коды и сохраняет результат на сервисе ТГУ.Сотрудники. Для чтения .xls формата была использована библиотека NPOI [19]. Данная процедура инициируется вручную раз в месяц, после начисления заработной платы сотрудникам.



Моя зарплата

30 апреля 2016 г.

Остаток в базе после расчета в предыдущем месяце		0,00руб.
Учебно-вспомогательный персонал (заработная плата)	Управление информатизации: отдел информатизации	1 000,00руб.
Учебно-вспомогательный персонал (единовременная надбавка)	Управление информатизации: отдел информатизации	1 000,00руб.
Районный коэффициент на все начисления		1 000,00руб.
Начислено по всем видам дохода (включая РК)		1 000,00руб.
Подоходный налог		1 000,00руб.
Перечислено в банк (на карточку, лицевой счет)		1 000,00руб.
Остаток в базе после расчета в текущем месяце		0,00руб.

Рис. 20 Пример страницы с данными о заработной плате сотрудников.

В целях безопасности, эти данные импортируются только для тех сотрудников, которые подтвердили свой доступ к аккаунту письменным

заявлением. Процесс подтверждения аккаунтов был описан в разделе, посвященном центру аутентификации ТГУ.Аккаунты. Кроме того, в целях безопасности все данные о зарплатах сотрудников хранится в базе данных в зашифрованном виде. Все данные шифруются с помощью алгоритма симметричного шифрования Rijndael (AES). Расшифровываются данные непосредственно перед выводом их пользователю. Таким образом, если злоумышленник получит доступ к базе данных сервиса, он не сможет получить сами данные, т.к. ключ для расшифровывания хранится в коде сервиса.

В итоге была решена задача синхронизации данных о заработной плате сотрудников между информационной системой бухгалтерии университета и сервисом ТГУ.Сотрудники.

Ещё одной задачей, которая решалась при разработке сервиса ТГУ.Сотрудники, была задача получения данных о сотруднике из системы Результативности научной деятельности (РНД). У этой системы, в отличии от предыдущих случаев, имеется HTTP API, с помощью которой можно получать данные в режиме реального времени в формате xml. Логика получения данных не была подстроена под нужды сервиса ТГУ.Сотрудники или других сервисов. В связи с этим было решено выделить её в отдельный сервис, названный Шлюз РНД (Рис. 5). Этот сервис инкапсулирует в себе всю логику работы с API РНД, а также выполняет функцию кэша. Наружу он предоставляет API, удобное для использования в других сервисах, в том числе на ТГУ.Сотрудники. Основные данные, получаемые от РНД – список публикаций, участие в конференциях, патенты, участие в научных проектах, методические пособия и т.п.

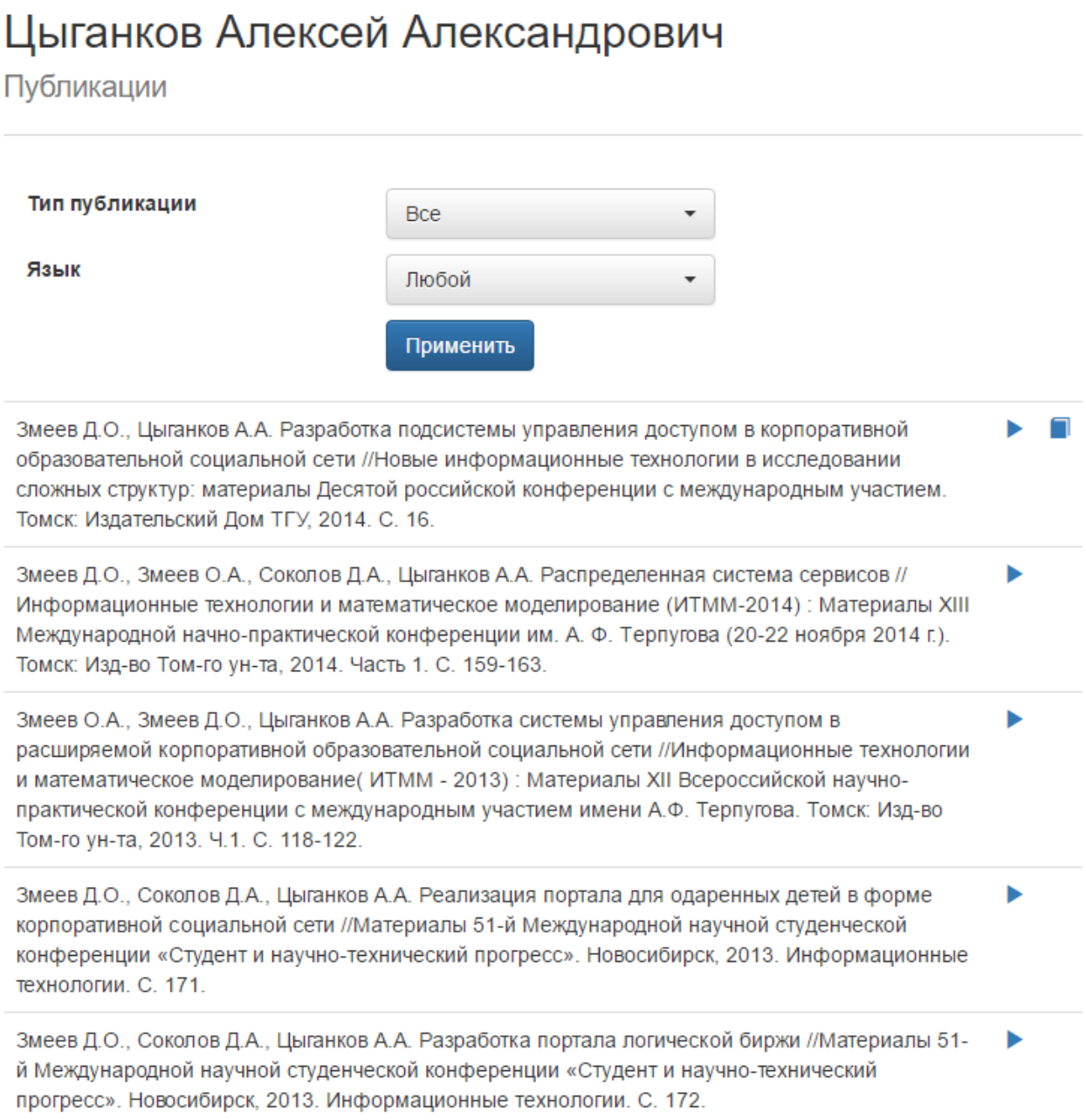


Рис. 21 Пример страницы со списком публикаций сотрудника.

4. ТГУ.Профили

Рассмотрим архитектуру сервиса ТГУ.Профили. Все остальные сервисы, созданные в рамках этой работы, были построены по тем же принципам и отличаются лишь особенностями предметной области. ТГУ.Профили – это сервис, который позволяет любому пользователю системы вести свой собственный блог или участвовать в групповом блоге, а также

позволяет вести страницу некоторой группы людей или структурного подразделения университета, заменяя им тем самым несложный информационный сайт.

На рисунке представлена упрощённая модель предметной области сервиса. Здесь центральную часть занимают классы Blog и User. Пользователь может являться владельцем нескольких блогов. Кроме того, пользователь может быть модератором блогов, участником или подписчиком. В блоке пользователь может писать посты (класс Post), к которым могут быть прикреплены изображения (класс Image). Кроме того, каждый пост может быть помечен несколькими тегами (класс Tag) для упрощения поиска постов по всему сервису. Пользователи могут комментировать любые посты (класс Comment), а к комментариям также прикреплять изображения и теги. Ко всем постам и комментариям пользователи могут добавить метку «Мне нравится» (класс Like). Кроме того, в блоге можно создавать альбомы изображений (класс Album). Для того, чтобы блог можно было использовать в качестве небольшого информационного сайта, к блогу можно прикреплять статически страницы (класс StaticPage).

Сервис реализован на фреймворке ASP.NET MVC на языке C#. В качестве базы данных использовался MS SQL Server. Для ORM использовался Entity Framework с подходом Code First.

Данные о пользователях, включая имя и фотографию, загружаются из сервиса ТГУ.Аккаунты. Межсервисное взаимодействие подробно описано в предыдущих частях.

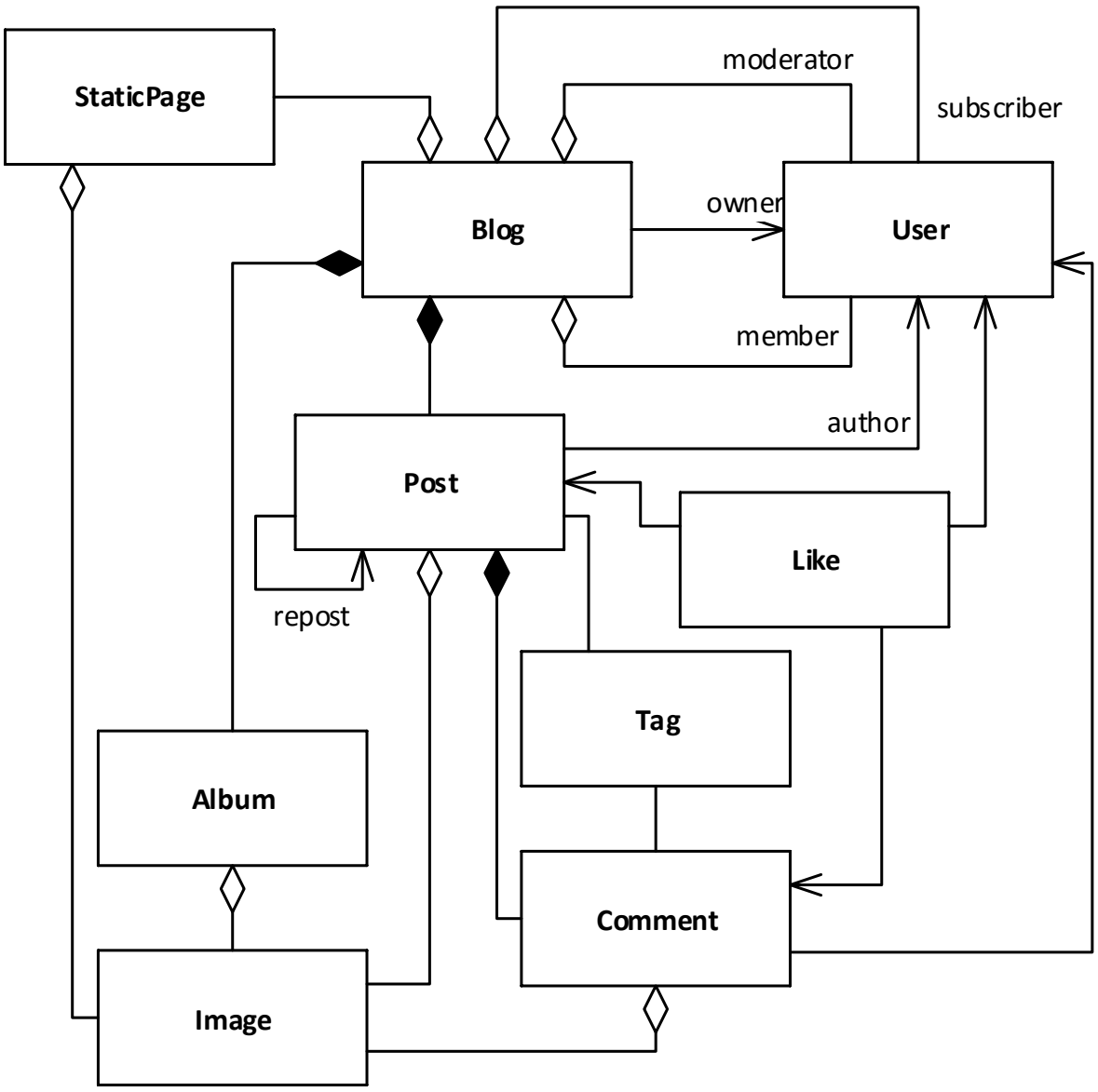


Рис. 22 Упрощённая модель предметной области сервиса ТГУ.Профили

Заключение

В рамках данной работы были созданы следующие сервисы корпоративной информационной системы университета: ТГУ.Аккаунты, ТГУ.Сотрудники, ТГУ.Выпускники, ТГУ.Студенты, Шлюз РНД, ТГУ.Профили, ТГУ.Новости, ТГУ.Сообщения, ТГУ.Уведомления, ТГУ.Контакты, Карта инновационно-активной среды ТГУ.

Разработанные сервисы интегрированы с существующими в университете информационными системами с помощью адаптеров. Разработаны библиотеки для упрощения взаимодействия между сервисами. Создан единый центр аутентификации ТГУ.Аккаунты, реализующий протокол OAuth. Решена задача слияния аккаунтов.

Все поставленные задачи выполнены, цель достигнута.

Список литературы

1. ServiceOrientedAmbiguity [Электронный ресурс] // Martin Fowler: [сайт]. [2005]. URL: http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html (дата обращения: 02.05.2016).
2. Microservices [Электронный ресурс] // Martin Fowler: [сайт]. [2014]. URL: http://martinfowler.com/articles/microservices.html (дата обращения: 02.05.2016).
3. Микросервисы [Электронный ресурс] // Хабрахбар: [сайт]. [2015]. URL: https://habrahabr.ru/post/249183/ (дата обращения: 02.05.2016).
4. HOW BIG SHOULD A MICRO-SERVICE BE? [Электронный ресурс] // The Bovine Synchrotron, James Lewis's blog: [сайт]. [2013]. URL: http://bovon.org/archives/350 (дата обращения: 02.05.2016).
5. What are Microservices? [Электронный ресурс] // Martin Fowler: [сайт]. URL: http://martinfowler.com/microservices/#what (дата обращения: 02.05.2016).
6. What is an ESB? [Электронный ресурс] // MuleSoft: [сайт]. URL: https://www.mulesoft.com/resources/esb/what-esb (дата обращения: 03.05.2016).
7. ESB на практике [Электронный ресурс] // IBM Bluemix: [сайт]. [2005]. URL: http://www.ibm.com/developerworks/ru/library/0509_flurry1/ (дата обращения: 03.05.2016).
8. Does My Bus Look Big in This? [Электронный ресурс] // InfoQ: [сайт]. [2008]. URL: http://www.infoq.com/presentations/soa-without-esb#anch24626 (дата обращения: 03.05.2016).

9. Service-Oriented Architecture (SOA) [Электронный ресурс] // urlIntegration: [сайт]. URL: http://www.urlintegration.com/?page_id=752 (дата обращения: 03.05.2016).
10. Understanding Enterprise Application Integration - The Benefits of ESB for EAI [Электронный ресурс] // MuleSoft: [сайт]. URL: https://www.mulesoft.com/resources/esb/enterprise-application-integration-eai-and-esb (дата обращения: 03.05.2016).
11. // ASP.NET: [сайт]. URL: http://www.asp.net/ (дата обращения: 15.05.2016).
12. //.NET: [сайт]. URL: https://www.microsoft.com/net/default.aspx (дата обращения: 15.05.2016).
13. Learn About ASP.NET Web API [Электронный ресурс] // ASP.NET: [сайт]. URL: http://www.asp.net/web-api (дата обращения: 15.05.2016).
14. Architectural Styles and the Design of Network-based Software Architectures [Электронный ресурс] // UCI: [сайт]. [2000]. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата обращения: 15.05.2016).
15. AMQP is the Internet Protocol for Business Messaging [Электронный ресурс] // AMQP: [сайт]. URL: https://www.amqp.org/about/what (дата обращения: 15.05.2016).
16. // RabbitMQ: [сайт]. URL: https://www.rabbitmq.com/ (дата обращения: 15.05.2016).
17. // EasyNetQ: [сайт]. URL: http://easynetq.com/ (дата обращения: 15.05.2016).
18. Фаулер М. Архитектура корпоративных программных приложений. Москва: "Вильяме", 2006. 544 с.

19. NPOI [Электронный ресурс] // CodePlex: [сайт]. URL: https://npoi.codeplex.com/ (дата обращения: 18.05.2016).

Отчет о проверке № 1

ФИО: Tsygankova Yana
дата выгрузки: 10.06.2016 11:14:15
пользователь: sauerstoff9@gmail.com / ID: 3039693

Информация о документе

№ документа: 6
Имя исходного файла: Разработка компонентов корпоративной информационной системы университета.docx
Размер текста: 1452 кБ
Тип документа: Не указано
Символов в тексте: 50649
Слов в тексте: 6594
Число предложений: 400

Оригинальность: 99.12%
Заимствования: 0.88%
Цитирование: 0%

Информация об отчете

Дата: Отчет от 10.06.2016 11:14:15 - Последний готовый отчет
Комментарии: не указано
Оценка оригинальности: 99.12%
Заимствования: 0.88%
Цитирование: 0%

Источники

Доля в тексте	Источник	Ссылка	Дата	Найдено в
0.27%	[1] не указано	http://window.edu.ru	раньше 2011 года	Модуль поиска Интернет
0.26%	[2] Рабочая программа конференции.	http://tsu.ru	25.12.2014	Модуль поиска Интернет
0.18%	[3] http://www.inf.tsu.ru/library/DiplomaWorks/CompScience/2010/	http://inf.tsu.ru	раньше 2011 года	Модуль поиска Интернет
0.14%	[4] не указано	http://window.edu.ru	раньше 2011 года	Модуль поиска Интернет
0.1%	[5] не указано	http://osp.ru	раньше 2011 года	Модуль поиска Интернет

Текст отчета

Министерство образования и науки Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)[2]
Факультет информатики
Кафедра программной инженерии[3]
ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
д-р физ.-мат. наук, профессор
_____ О.А. Змеев
«_____»_____2016 г.
МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
РАЗРАБОТКА КОМПОНЕНТОВ КОРПОРАТИВНОЙ ИНФОРМАЦИОННОЙ СИСТЕМЫ УНИВЕРСИТЕТА
по основной образовательной программе подготовки магистров
«Управление проектами по разработке программного обеспечения»
направление подготовки
02.04.02 – Фундаментальная информатика и информационные технологии
Цыганков Алексей Александрович
Научный руководитель ВКР,
д-р физ.-мат. наук, профессор
_____ О.А. Змеев
«_____»_____2016 г.
Автор работы
студент группы № 1447
_____ А.А. Цыганков
Томск - 2016
РЕФЕРАТ

Выпускная квалификационная работа 44 с., 22 рис., 19 источников.

КОРПОРАТИВНАЯ ИНФОРМАЦИОННАЯ СИСТЕМА, СЕРВИС-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА, SOA, МИКРОСЕРВИСЫ, ОЧЕРЕДЬ СООБЩЕНИЙ

Объекты исследования – варианты реализации и взаимодействия компонентов информационной системы.

Цель работы – разработать компоненты корпоративной информационной системы университета.

Методы исследования – изучение существующих подходов к разработке компонентов информационной системы и их взаимодействия, разработка общих средств взаимодействия программных сервисов.

Результаты работы – разработаны компоненты корпоративной информационной системы университета на основе сервис-ориентированной архитектуры (SOA), разработаны повторно используемые библиотеки для упрощения взаимодействия между сервисами, разработаны адаптеры для интеграции с внешними системами.

Оглавление

Введение

Идея корпоративных информационных систем не нова. Существует огромное множество различных готовых информационных систем, доступных к внедрению в организацию. Ещё больше различных корпоративных информационных систем, созданных силами и средствами самих этих организаций для своих собственных нужд. Более того, в различных отделах одной и той же организации зачастую можно увидеть несколько информационных систем, которые достаточно часто никак не связаны друг с другом. В больших организациях это создаёт существенную проблему.

Решений этой проблемы две.

Создать новую единую информационную систему, включающую в себя возможности всех, которые уже используются в организации.

Попытаться связать уже имеющиеся информационные системы между собой.

Очевидный недостаток первого решения – большая стоимость. Каждая из используемых информационных систем могла создаваться годами с учетом опыта многих пользователей из различных организаций. Собрать этот опыт из нескольких систем в одну и повторить результаты – задача зачастую невероятная.

Недостаток второго решения заключается в том, что различные информационные системы, создаваемые обычно абсолютно независимо друг от друга различными командами разработчиков не предназначены для связывания друг с другом. Некоторые из них реализуют внешние интерфейсы различного рода, чтобы можно было взять данные из них для автоматической обработки. Некоторые реализуют различные стандартные протоколы обмена данными. Но очень редки случаи, когда несколько информационных систем действительно полноценно интегрируются друг с другом.

В университетах, как и в любой крупной организации, также применяются различные информационные системы в различных отделах. В рамках данной работы решается проблема связывания этих систем путём разработки промежуточных компонентов на основе сервис-ориентированной архитектуры. Разрабатываемые компоненты интегрируются с существующими системами, объединяют их друг с другом и добавляют недостающую функциональность.

Работа выполняется в рамках Томского государственного университета и все примеры будут приведены на нём, но с небольшими изменениями та же работа применима и к любому другому университету.

Постановка задачи

Цель данной работы – разработать компоненты корпоративной информационной системы университета.

Перечислим эти компоненты и опишем основные функциональные требования, предъявляемые к ним.

ТГУ.Выпускники – это каталог всех выпускников университета. Данный компонент должен позволять пользователю:

просмотреть список всех выпускников по каждому факультету в отдельности;

осуществить поиск по всему списку студентов университета по различным критериям;

просмотреть данные о выпускнике, включая год и факультет, которые этот выпускник окончил, а также основную информацию о его деятельности после выпуска из университета;

возможность импортировать данные о выпускниках из файлов в формате xls;

возможность совершать рассылки по email по отдельным группам выпускников.

ТГУ.Сотрудники – это каталог всех сотрудников университета. Данный компонент должен позволять пользователю:

просмотреть список сотрудников в каждом из подразделений и отделов университета;

просмотреть основную информацию о сотруднике: его звания, место работы, фотографию;

просмотреть научные и другие достижения сотрудников, результаты его научной деятельности, в том числе список публикаций, причем данные о публикациях и прочих достижениях берутся из внешней системы;

В случае, если пользователь является аутентифицированным сотрудником университета, то в рамках этого компонента необходимо предоставить сотруднику возможность принимать участие в различных бизнес-процессах, требующих авторизации. На момент написания работы, было выделено два таких процесса:

просмотреть представленные в понятном виде начисления по своей заработной плате;

заполнить и утвердить свой индивидуальный план.

ТГУ.Профили – это компонент для ведения личных и групповых блогов. Данный компонент должен позволять пользователю:

вести свой личный блог;

вести один блог на группу пользователей;

вести личный блог за другого человека, при этом не должно быть видно, кто именно ведёт этот блог;

использовать блог в качестве сайта отдела или некоторого структурного подразделения университета.

ТГУ.Новости – это агрегатор новостей со всех интернет-ресурсов ТГУ и других ресурсов, которые предоставляют свои новости в формате RSS. Данный компонент должен позволять пользователю:

выбрать те ресурсы ТГУ, с которых он хочет получать новости;

указать те ресурсы, не входящие в список ресурсов ТГУ, с которых он хочет получать новости;

получать новости из компонента ТГУ.Профили;

указать, новости каких из выбранных источников показывать в общей новостной ленте в данный момент.

ТГУ.Сообщения – это чат для общения между пользователями системы. Данный компонент должен позволять пользователю:

просмотреть список своих недавних собеседников;

найти нужного собеседника из всего множества пользователей системы;

вести обмен мгновенными сообщениями с любым пользователем системы.

ТГУ.Студенты – это каталог всех студентов ТГУ. Данный компонент должен позволять пользователю:

просмотреть список всех студентов в каждом из факультетов;

просмотреть базовую информацию о студенте: место учебы, год поступления, фотографию;

просмотреть научные и другие достижения студентов, результаты научной деятельности, в том числе список публикаций.

ТГУ.Уведомления – это компонент для рассылки уведомлений. Данный компонент должен позволять пользователю выбрать группу пользователей системы и разослать этой группе текстовое уведомление. Другим системам компонент должен предоставлять интерфейс для отправки текстовых уведомлений конкретным пользователям.

Карта инновационно-активной среды ТГУ – это сайт для демонстрации текущей деятельности и результатов работы в рамках проектов из программы повышения конкурентоспособности «5 – 100». Данный компонент должен предоставлять пользователю возможность:

заполнить данные о проекте из программы «5 – 100»;

выкладывать новости о работе проекта;

выкладывать проектную документацию;

вести список экспертных групп;

вести список инициатив по направлениям;

организовывать мероприятия в рамках программы «5 – 100»

Основные общие особенности всех описанных компонентов.

Высокая степень независимости. Каждый компонент может разрабатываться отдельно от других. Очень мало данных, которые необходимы более чем в одном компоненте.

Различные источники финансирования, и, как следствие, различные заинтересованные лица. Эти лица хотят получить необходимый им продукт, и при этом их не интересует развитие других компонентов. Объем финансирования одного компонента должен определять скорость развития только этого компонента, а не других.

Экспериментальный характер. Должна быть возможность исключить за ненадобностью или заменить любой из разрабатываемых компонентов без существенного ущерба остальной системе.

В связи с описанными особенностями было принято решение построить архитектуру системы на основе шаблона «Сервис-ориентированная архитектура».

Описание сервис-ориентированной архитектуры

Общее описание

Существует очень много различных толкований понятия сервис-ориентированной архитектуры. Мартин Фаулер пишет о том, что не существует единого определения, которое можно было бы назвать точным. При этом он приводит четыре различных расшифровки этого понятия, которыми обычно пользуются разработчики [1].

Для однозначности выделим основные признаки, которыми обладает сервис-ориентированная архитектура. Сервис-ориентированная архитектура (Service Oriented Architecture, SOA)[4] подразумевает архитектуру, в которой вместо единого приложения создаётся несколько web-сервисов, которые обмениваются между собой данными по стандартным сетевым протоколам (например, http).

Рис. 1 Сервис-ориентированная архитектура

Web-сервисы зачастую противопоставляются монолитам. Монолит – это единое приложение, которое включает в себе все возможности системы целиком. Размеры монолита и web-сервиса зависят от размеров всей системы, но обычно размер web-сервиса очень небольшой. Считается, что для эффективной разработки один web-сервис должен быть такого размера, чтобы полностью уместиться в сознание одного среднего разработчика, включая основные детали реализации. Если web-сервис превышает эти размеры, то его принято делить на несколько новых web-сервисов. Монолит в свою очередь фактически не ограничен в размерах. При этом в больших системах зачастую встречаются ситуации, когда один разработчик знает только некоторую часть этого приложения, и ни у кого в команде нет полной информации о всём приложении.

Web-сервисы размещаются на серверах независимо друг от друга. Под этим понимается несколько условий.

Каждый отдельный web-сервис не зависит от того, на каких серверах и в какой сети располагаются другие сервисы этой системы. Web-сервисы могут располагаться как на одном, так и на нескольких серверах, при этом они об этом могут и не знать. Единственное, что необходимо учитывать, это то, что при активном обмене данными между сервисами и высоким требованиям к скорости работы следует размещать сервисы как можно ближе друг у другу, чтобы время на сетевое взаимодействие было минимальным.

Каждый отдельный web-сервис разворачивается независимо от того, как разворачиваются другие сервисы. Другими словами, нет никакого порядка или других правил, которые бы регламентировали разворачивание сервисов относительно друг друга.

Рис.2 Сравнение монолита и сервисов SOA. Распределение между серверами.

Насколько бы ни были web-сервисы независимыми, они в любом случае должны обмениваться данными или отправлять друг другу команды. Общение между сервисами осуществляется по стандартным сетевым протоколам. Например, это может быть http, очередь сообщений или любое другое средство сетевого обмена данными. В частности, распространение получил формат REST API, который определяет правила взаимодействия по протоколу HTTP.

Однако, не любую систему разумно реализовывать по сервис-ориентированной архитектуре. Существуют системы, в которых данные сильно связаны друг с другом. Если представить модель предметной области в виде графа, где вершинами графа будут типы сущностей, то этот граф будет иметь высокую связность. В этом случае при попытке разделить систему на сервисы часть вершин графа попадёт в один сервис, а часть – в другой. Между сервисами окажется большое число рёбер, а это означает, что между сервисами будет чрезмерно большой обмен данными, а целостность данных придётся соблюдать не средствами БД, а программными средствами, что весьма трудоёмко и повышает вероятность возникновения ошибок. В таких случаях обычно рекомендуется не делить систему на сервисы, а оставлять её в виде монолита. Возможно, удачно получится выделить в отдельные сервисы лишь небольшие независимые части.

Сравнение с микросервисной архитектурой

Часто обсуждают или описывают сервис-ориентированную архитектуру вместе с архитектурой, основанной на микросервисах. Как говорилось в первой части данный работы, у сообщества нет точного определения сервис-ориентированной архитектуры. В связи с этим одни считают, что SOA и микросервисы – это одно и то же, другие их противопоставляют, а третьи считают, что микросервисы – это частный случай SOA. Упомянутый ранее Мартин Фаулер вместе с Джеймсом Льюисом в своей статье Microservices [2] [3] также описывает эту проблему.

В другой статье Джеймс Льюис указывает на то, что микросервисы должны быть максимально маленькими и простыми, «полностью умещаться в голове», в то время как в SOA размеры сервисов не играют столь важную роль [4]. SOA нам говорит о том, что мы должны делить приложение на сервисы, согласно бизнес-требованиям. Обычно такие сервисы превращаются в отдельные сайты, которые обмениваются между собой данными, причем один такой сайт сам по себе может быть достаточно сложным и громоздким.

Согласно микросервисной архитектуре, подобные большие сервисы необходимо дробить ещё на несколько сервисов, чтобы каждый из них был очень простым, решал какую-то одну задачу. Однако бизнес-требования не позволяют плодить бесчисленное количество отдельных сайтов, ведь это будет неудобно конечному пользователю. Поэтому такие сервисы продолжают выглядеть как единый компонент, несмотря на то, что внутри он состоит из множества маленьких компонентов-микросервисов. Загружая одну страницу, пользователь может увидеть данные, полученные сразу от нескольких десятков микросервисов. При этом одни и те же микросервисы могут быть использованы в нескольких крупных сервисах.

Рис. 3 Микросервисная архитектура

Подытожим сравнение SOA и микросервисной архитектуры. Микросервисы не нарушают ни одного принципа SOA, а напротив, только дополняют её [5]. Таким образом можно сказать, что микросервисная архитектура – это всё-таки частный случай SOA. При этом во всём приложении (или группе приложений) может применяться идея крупных сервисов на основе SOA, а при реализации отдельных сервисов могут применяться микросервисы.

Enterprise Service Bus

Чтобы контролировать общение между сервисами, иногда прибегают к созданию Enterprise Service Bus (ESB), которая служит промежуточным слоем между всеми сервисами, которая контролирует прохождение всей информации, а также доступ к этой информации [6] [7]. ESB может реализовывать свой собственный протокол для обмена сообщениями или использовать один или несколько из стандартных протоколов, например, JMS или AMQP.

Во многих случаях ESB становятся очень сложными в реализации и в использовании. Особенно это касается случаев, когда в системе необходимо интегрировать старые сервисы (legacy systems), которые создавались без учёта создания ESB. Для каждого такого сервиса необходимо создавать некий адаптер, который будет преобразовывать сообщения из формата, в котором сервис может отдавать информацию, в формат, которые требуется в ESB. В итоге стоимость таких систем и работ по интеграции с ней других компонентов может превышать выгоду. Об этом говорят и Мартин Фаулер с Джимом Веббером [2] [8], и практический опыт Томского государственного университета, связанный с попыткой использования этого подхода ранее.

Рис. 4 Один из вариантов SOA-архитектуры [9]

На приведённом рисунке видно, что ESB не зависит от технологий, на которых реализованы подключаемые сервисы. А для legacy systems создаются адаптеры. Сама ESB выполняет функции управления (мониторинга), преобразования сообщений, контроль безопасности и маршрутизации сообщений.

Стоит отметить, что ESB часто ассоциируется с SOA и наоборот. Но включать ESB в само понятие SOA ошибочно, т.к. SOA – это лишь набор принципов, которым должны удовлетворять сервисы, в то время как ESB – это инструмент, с помощью которого можно связать эти сервисы. Более того, микросервисная архитектура вообще отрицает ESB из-за её сложности. Считается, что микросервисы должны отправлять сообщения друг другу напрямую по стандартным и, главное, очень простым и легким протоколам.

Enterprise Application Integration

Говоря о ESB нельзя не сказать об Enterprise Application Integration (EAI, интеграция приложений предприятия). EAI – это общая категория подходов для создания совместимости между различными несвязанными системами, которые составляют типичную инфраструктуру организации [10]. Обычно это понятие применяется тогда, когда в организации уже используются системы, которые не созданы для взаимной интеграции, но интеграция которых повысит общую эффективность работы организации. В этом случае реализуют промежуточные системы, которые транслируют сообщения между целевыми системами. Т.к. реализовывать связи между каждой парой систем трудозатратно (особенно если количество этих систем велико), то зачастую применяют ESB для уменьшения количества связей и количества адаптеров.

Описание разработанного решения

Межсервисное взаимодействие

Разрабатываемые в данной работе компоненты корпоративной информационной системы были разбиты на сервисы в соответствии с архитектурой SOA. На рисунке показаны не все компоненты информационной системы, а только те, в разработке которых принимал участие автор данной работы. Другие компоненты подключаются к указанным через адаптеры, которые реализуются отдельно для каждого конкретного случая.

Рис. 5 Структура разрабатываемых компонентов. Зелёным обозначены компоненты, создаваемые исключительно автором данной работы. Синим показаны компоненты, проектируемые и контролируемые автором работы, но реализуемые при поддержке других разработчиков.

Несмотря на то, что сервисы в SOA разрабатываются и разворачиваются практически независимо друг от друга, тем не менее они интенсивно взаимодействуют, обмениваются данными. Взаимодействие сервисов должно осуществляться по стандартным протоколам. В данной работе использовались два основных способа: HTTP API и очередь сообщений, которая реализует протокол, AMQP.

HTTP API

HTTP API – наиболее простой и распространённый способ обмена данными между сервисами. Клиенты HTTP имеются в большинстве платформ, фреймворков, языков. Web-сервис сам по себе без дополнительных инструментов может предоставлять API, но практически в каждом web-фреймворке реализованы инструменты для упрощения процесса создания API.

Сервисы в данной работе реализованы на фреймворке ASP.NET [11], который в свою очередь основан на фреймворке .NET [12], в котором реализован HTTP-клиент. Также частью ASP.NET является фреймворк Web API [13], который значительно упрощает создание API, в том числе в формате RESTful API [14].

Однако, на чём бы ни были реализованы другие сервисы системы, взаимодействие по протоколу HTTP всегда будет простым. Но есть и обратная сторона: HTTP-запросы сравнительно медленны. Причем большая часть времени тратится не на полезную нагрузку (формирование и обработка данных, их сериализация, передача, десериализация), а на открытие и закрытие соединения, т.к. на каждый HTTP-запрос открывается и закрывается новое TCP-соединение.

Рассмотрим пример. В сервисе ТГУ.Сотрудники необходимо выводить список имён всех сотрудников отдела университета. В отделе могут работать десятки сотрудников. Но имя каждого сотрудника хранится не на ТГУ.Сотрудники, а на сервисе ТГУ.Аккаунты, как и все остальные данные об аккаунтах каждого пользователя системы. Чтобы вывести список имён сотрудников отдела, необходимо для каждого сотрудника отправить HTTP-запрос на ТГУ.Аккаунты для получения имени. Один запрос занимает в среднем 25мс. Выполняя запросы последовательно друг за другом при условии, что в отделе 100 сотрудников, понадобится 2500мс на то, чтобы получить все имена. Можно распараллелить запросы в нескольких потоках. Был проведён опыт, и в этом случае общее время всех запросов стало равно 941мс.

Для многих сайтов выдвигается требование, чтобы среднее время обработки сервером запроса для одной страницы не превышало 100мс, а максимальное время обработки не превышало 500мс. Это же требование выдвигалось и ко всем разрабатываемым в данной работе сервисам. Очевидно, что, посылая для получения каждого имени отдельный HTTP-запрос, ограничение соблюсти невозможно.

Ещё одним способом ускорить обмен данными является изменение API на стороне сервиса ТГУ.Аккаунты таким образом, чтобы не приходилось создавать новый HTTP-запрос для получения каждого имени, а достаточно было бы отправить только один HTTP-запрос и получить все необходимые данные в ответе. Этот вариант полностью решает проблему долгого получения данных. В приведённом примере вместо 100 HTTP-запросов необходимо отправить только 1, а время обработки одного запроса увеличивается всего лишь до 30мс.

Но этот способ порождает новые проблемы. Для его реализации приходится подстраивать API всех сервисов под конкретные нужны каждого другого сервиса. При условии, что список сервисов в системе открытый (условно неограниченный), пришлось бы потратить слишком много ресурсов на создание и поддержку API. По сути для каждого зависимого сервиса пришлось бы писать своё собственное API. В целях экономии ресурсов было решено разрабатывать максимально простое и максимально универсальное API, которое могло бы быть использовано любыми другими сервисами для любых целей, даже если для достижения этих целей придётся сделать несколько HTTP-запросов.

Ещё одной особенностью, которую можно считать недостатком, является то, что после отправки HTTP-запросов необходимо ждать ответ сервера, который приходит только после полной обработки этого запроса. Это не всегда удобно. Иногда ответ сервера не требуется, нужно лишь передать данные для последующей обработки. Например, в сервисе ТГУ.Выпускники реализуется массовая рассылка писем. Рассылка писем в системе осуществляется централизованно, через ТГУ.Аккаунты. При отправке писем от ТГУ.Выпускники к ТГУ.Аккаунты нет необходимости ждать подтверждения, что это письмо отправлено на почтовый сервер. Достаточно просто отправить данные. Эту и другие проблемы может решить очередь сообщений.

Очередь сообщений

Другим способом взаимодействия между Web-сервисами является очередь сообщений (Message Queue, MQ). Очередь сообщений – это целый класс программных средств, которые выполняют одну задачу: принимать информационные сообщения от одних программ и передавать эти сообщения другим программам в порядке FIFO (понятие очереди в классическом смысле: первое пришедшее от отправителя сообщение отправляется получателю также первым).

Рис. 6 Очередь сообщений с одним отправителем и одним получателем

На рисунке овал P означает сервис отправителя (producer), овал C – сервис получателя (consumer), красные прямоугольники – сообщения в очереди сообщений, стрелки – передачу сообщений.

Помимо задачи непосредственно передачи сообщения очередь сообщений также выполняет задачу маршрутизации сообщений и, как следствие, задачу балансировки нагрузки.

Рис. 7 Очередь сообщений с двумя получателями, которые получают сообщения поочередно, тем самым происходит простейшая балансировка нагрузки.

Многие из таких программных средств реализуют стандартный протокол AMQP (Advanced Message Queuing Protocol). AMQP – открытый стандарт для передачи сообщений между приложениями или организациями [15].

В рамках данной работы в качестве очереди сообщений был выбран RabbitMQ [16]. Он является типичным представителем этого класса программных средств, реализует все стандартные возможности очереди сообщений, реализует протокол AMQP, является одним из самых распространённых средств в своём классе.

Как уже было указано ранее, все сервисы в рамках данной работы реализуются на платформе .NET. На этой платформе имеется несколько клиентов для RabbitMQ. Среди них был выбран EasyNetQ [17], т.к. в нём наиболее просто реализована сериализация и десериализация объектов. В общем случае сообщения в очереди сообщений представляют собой текст. Однако интерфейс этого клиента разработан таким образом, что отправитель посылает в очередь сообщений объект некоторого класса, а получатель принимает этот объект того же класса, при этом в коде всё это выглядит очень компактно.

Все дальнейшие примеры будут основаны на реализации RabbitMQ, однако с некоторыми поправками могут быть распространены и на другие очереди сообщений.

Вернёмся к примеру про массовую рассылку писем, который был описан выше. В этом примере нам не нужно было ждать подтверждения об успешной отправке письма на почтовый сервер, а достаточно было просто отправить данные. Очередь сообщений с этим справляется хорошо. Сообщение с данными отправляются в очередь и на этом связь отправителя и сообщения теряется. Отправитель работает дальше, а сообщение ждёт своей очереди, когда получатель сможет его обработать. Таким образом, если обработка сообщений занимает больше времени, чем их формирование, то очередь сообщений будет являться сглаживающим буфером, который позволит отправителю максимально быстро сформировать все необходимые сообщения и продолжить работу, а получателю позволит не потерять сообщения даже на пиковых нагрузках.

Однако, если всё-таки отправителю нужно получить какой-то ответ от получателя, необходимо использовать более сложную конфигурацию очереди сообщений, которая реализует по сути удалённый вызов процедур (RPC). Участвующие сервисы рассматриваются как клиент и сервер. Клиент отправляет сообщение-запрос в одну очередь, помечая его необходимым идентификатором. Сервер получает это сообщение, обрабатывает и отправляет сообщение-ответ в другую очередь, добавляя к нему идентификатор запроса. Клиент получает сообщение-ответ и сопоставляет его по идентификатору с запросом.

Рис. 8 Очередь сообщений в режиме RPC

Естественно, отправку запросов и приём сообщений в этом случае можно распараллелить, а серверов можно поставить не один, а несколько, чтобы

распределить между ними запросы.

Кроме того, очередь сообщений более устойчива к недоступности сервиса. Далее приводится график, на котором изображена ситуация при перезагрузке сервиса, когда он был недоступен в течении нескольких десятков секунд. На оси ординат изображено количество сообщений в очереди в указанный момент времени. Здесь мы видим, что при недоступности сервиса сообщения накапливались в очереди, но не терялись. После возобновления работы сервиса он их сразу получил и обработал.

Рис. 9 Количество сообщений в очереди при кратковременной недоступности сервиса

Теперь посмотрим на скорость работы. Для этого был проведён опыт. Было отправлено 1000 запросов по HTTP и 1000 запросов по RabbitMQ в режиме RPC. Запросы отправлялись последовательно друг за другом, без распараллеливания, чтобы исключить возможное влияние систем обеспечения параллельного выполнения. Результаты опыта представлены на диаграмме. Очередь сообщений оказалась примерно в 2 раза быстрее, чем HTTP.

Рис. 10 Время отправки запросов и получения ответов по HTTP и RabbitMQ. Время указано в мс.

Однако у очереди сообщений есть и недостаток, который заключается в том, что её сложнее использовать. Для RabbitMQ имеется множество клиентов для различных платформ, но они не встроены во фреймворки, и они не согласуются друг с другом. Сообщение, сформированное и отправленное одним клиентом может быть не прочитано другим клиентом. Очереди сообщений менее популярны, у сообщества меньше знаний о них. В то же время различные сервисы системы разрабатываются на различных платформах и различными командами, которые фактически не связаны друг с другом.

Для облегчения такого рода взаимодействия в данной работе было решено поддерживать оба способа. Это означает, что одна и та же функция доступна как по протоколу HTTP, так и через RabbitMQ.

На сервисах, создаваемых в рамках данной работы, в общем случае действует алгоритм, изображённый на Рис. 11. На рисунке видно, что вначале происходит попытка найти ответ на запрос в кэше. Если в кэше ответа нет, то происходит попытка отправить запрос через очередь сообщений. Если эта попытка оказалась неудачной, то происходит попытка отправить запрос через HTTP. Если всё-таки ответ получить удалось, то проверяется, необходимо ли сливать аккаунты. Подробнее о слиянии аккаунтов рассказано в той части этой главы, которая посвящена единому центру аутентификации. Если аккаунты необходимо слить, то они сливаются и весь алгоритм запроса начинается сначала, но уже с параметрами нового аккаунта.

Таким образом, если очередь сообщений по каким-то причинам перестанет работать, то сервисы продолжат взаимодействовать друг с другом, но по более медленному протоколу HTTP, тем самым надёжность системы в целом повышается.

Здесь описан общий вид алгоритма отправки запроса и обработки ответа, использующий два протокола и кэш. Однако на практике не во всех случаях имеется возможность использовать оба протокола. Иногда определённый запрос доступен только через HTTP, иногда только через очередь сообщений. Для этих случаев в алгоритме предусмотрена вариативность: можно для каждого запроса в отдельности указать только путь через HTTP или только путь через очередь сообщений. Кроме того, время жизни кэша для каждого из запросов должно подбираться индивидуально. К примеру, имя человек меняет не часто, поэтому кэш запроса на получение имени можно устано

вить достаточно большим (например, сутки), а запрос на наличие новых уведомлений кэшировать вообще нельзя, т.к. уведомления должны доходить до пользователя максимально быстро. Для этих целей в алгоритме отправки запросов для каждого запроса предусмотрено указание, нужно ли использовать кэш, и, если нужно, то каково будет его время жизни.

Рис. 11 Алгоритм отправки запроса к другому сервису и обработки ответа.

Выделение интерфейсных компонентов

Рассмотрим следующую ситуацию. Имеется два сервиса Customer1 и Customer2. Оба они используют одно и то же API третьего сервиса DataOwner.

Рис. 12 Два сервиса используют API третьего сервиса

В этом случае всю логику отправки запросов и получения ответов (алгоритм описан выше) приходится дублировать в каждом из сервисов Customer1 и Customer2. Чтобы этого избежать, вся описанная логика была вынесена в отдельную библиотеку, которая подключается к каждому сервису, использующему это API, а также к сервису, который это API предоставляет (реализовано типовое решение Шлюз [18]).

Рис. 13 Выделение логики отправки запросов и обработки ответов в отдельную библиотеку.

Рассмотрим следующую ситуацию. Теперь у нас есть два сервиса, предоставляющих каждый своё API: DataOwner1 и DataOwner2. Для каждого из этих сервисов создана своя собственная библиотека, обеспечивающая отправку запросов к API: Gateway1 и Gateway2. В этом случае снова возникает дублирование кода: один и тот же алгоритм используется в разных библиотеках, за исключением лишь деталей реализации API. Чтобы этого избежать, была создана библиотека BaseGateway, которая реализует типовой шаблон Супертип слоя [18]. В этой библиотеке содержится общая логика отправки запросов и обработки ответов, а детали реализации API разных сервисов остаются в Gateway1 и Gateway2, которые наследуются от BaseGateway.

Рис. 14 Добавление супертипа слоя для библиотек, реализующих отправку и обработку запросов

Таким образом мы полностью исключили дублирование кода и во всех сервисах используем одну и ту же логику.

Такая организация кода имеет ещё одно преимущество. Как было указано ранее, для взаимодействия через RabbitMQ используется клиент EasyNetQ. Этот клиент сериализует объект некоторого класса, отправляет в текстовом виде в очередь сообщений, а на другой стороне принимает и десериализует этот объект в тот же класс. Проблема в том, что, чтобы сериализация, десериализация, отправка и приём сообщений работали корректно, необходимо, чтобы на одной и на другой стороне были одни и те же классы, в том числе и полные названия этих классов, включая имя сборки. Этого добиться можно лишь в том случае, если эти классы будут взяты из одной библиотеки, которая подключается к обоим сервисам: и к отправителю, и к получателю.

Единый центр аутентификации ТГУ.Аккаунты

Различные сервисы корпоративной информационной системы должны быть связаны общими аккаунтами. Если у пользователя уже есть аккаунт на одном сервисе, то, очевидно, этот же пользователь должен иметь возможность зайти под этим же аккаунтом и на другом сервисе. И с другой стороны: сервисы должны обмениваться между собой данными, в том числе и данными по конкретным пользователям. Чтобы синхронизировать данные о пользователях, необходимо иметь общие аккаунты на все сервисы. Для этих целей был создан единый центр аутентификации ТГУ.Аккаунты.

Рис. 15 Пример страницы аккаунта с основными данными с точки зрения администратора системы.

На этом сервисе хранятся все логины и пароли пользователей, а также общая информация, не привязанная ни к одному из остальных сервисов: фотография, контактные данные, идентификаторы в различных сервисах.

Кроме того, на некоторых сервисах необходима возможность удостовериться, что доступ к аккаунту имеет именно тот пользователь, о котором имеется информация. Для этих целей была создана система аутентификации. Пользователи подтверждают свои данные и единоличный доступ к аккаунту с помощью бумажного заявления. После этого аккаунт считается подтверждённым. Любой другой сервис может проверить любой аккаунт на подтверждённость и в зависимости от этого предоставлять пользователю какие-либо функции или нет. Например, на сервисе ТГУ.Сотрудники имеется возможность сотрудникам университета просмотреть информацию о своей заработной плате. Но в целях безопасности эта информация предоставляется только тем, кто подтвердил свой аккаунт.

Аутентификация на других сервисах осуществляется по протоколу OAuth. Для реализации этого протокола сначала регистрируются все сервисы системы, которые должны иметь возможность проходить аутентификацию через ТГУ.Аккаунты. Зарегистрировать свой сервис может любой, отправив заявку. После регистрации сервис получает свой идентификатор. Далее этот сервис реализует алгоритм, приведённый на рисунке. Далее приведён полный алгоритм действий по шагам (* звёздочками помечены шаги, которые необходимо реализовать на стороне подключаемого сервиса).

Пользователь кликает по ссылке "Выполнить вход". Запрос отправляется на сервер сервиса.

* Сервис возвращает перенаправление на страницу https://accounts.tsu.ru/Account/Login2/?applicationId=... Здесь необходимо указать идентификатор сервиса, который будет выдан при регистрации.

Пользователю показывается страница с полями для ввода логина и пароля.

Пользователь вводит свои логин и пароль, отправляет на сервер ТГУ.Аккаунты, где происходит их проверка.

ТГУ.Аккаунты возвращает перенаправление на страницу сервиса, которая указывается при регистрации. К адресу этой страницы добавляется временный token, который представляет собой строку из 255 символов.

* Сервис должен обменять временный token на AccessToken и AccountId. Для этого он направляет прямой POST-запрос (без участия клиента) на адрес https://accounts.tsu.ru/api/Account/?token=...&applicationId=...&secretKey=... Здесь необходимо будет указать временный token, который был получен на предыдущем шаге, идентификатор сервиса и секретный ключ, которые выдаются при регистрации. Секретный ключ используется для проверки подлинности сервиса.

В ответ ТГУ.Аккаунты возвращает AccessToken (строка из 255 символов) и AccountId (GUID) аутентифицированного пользователя. AccountId - это глобальный идентификатор пользователя, который используется в ТГУ.Аккаунты и других связанных с ним сервисах.

* Реализация этого шага напрямую зависит от целей и особенностей сервиса. Здесь приведены рекомендации для типичного использования. Полученные данные необходимо сохранить. По AccountId необходимо найти локального пользователя и считать его аутентифицированным. Если пользователь не найден среди локальных, то его необходимо зарегистрировать. После этого можно вернуть в браузер перенаправление на страницу пользователя.

Рис. 16 Процесс аутентификации через сервис ТГУ.Аккаунты

Одним из ключевых принципов протокола OAuth является то, что логин и пароль пользователя хранятся и вводятся только в центре аутентификации. Ни один другой сервис ни при каких обстоятельствах не получает доступа к паролю пользователя, этим обеспечивается безопасность системы.

Большинство аккаунтов были зарегистрированы автоматически. Основными источниками послужили база данных отдела кадров с полным списком сотрудников, различные списки студентов и выпускников университета. Зачастую один и тот же человек попадал в несколько списков. Например, человек одновременно может являться выпускником бакалавриата, студентом магистратуры и сотрудником университета. Более того, различные списки студентов и выпускников были не согласованы, и один и тот же человек мог попасть в роли студента в два списка. Кроме того, пользователи не всегда подозревали о том, что для них уже автоматически зарегистрирован аккаунт, и пытались зарегистрировать себе аккаунт вручную. Всё это приводило к огромному количеству дубликатов – аккаунтов, принадлежащих одному пользователю. В условиях корпоративной информационной системы это недопустимо.

Для устранения этой проблемы была реализована возможность сливать аккаунты. В случае, если администратор системы находит дубликаты, он помечает один аккаунт как основной, а другой – дублирующим. При этом дублирующий аккаунт не удаляется совсем, т.к. на него могли ссылаться другие сервисы, но он удаляется из публичных списков. Если пользователь попытается зайти под дублирующим аккаунтом, система перенаправит его на основной. Если на основном аккаунте не хватало каких-то данных (фотографии, контактов), но эти данные были на дублирующем аккаунте, то они копируются на основной.

Рис. 17 Основной и дублирующий аккаунты на разных сервисах

Однако аккаунты необходимо сливать на всех сервисах, где они были зарегистрированы, но ТГУ.Аккаунты не могут делать это напрямую. Каждый сервис должен сам реализовать логику слияния аккаунтов в соответствии со своей спецификой. Если сервис посылает на ТГУ.Аккаунты запрос, касающийся дублирующего аккаунта, то система возвращает HTTP статус 498, означающий, что аккаунты слиты. При этом в теле ответа содержится идентификатор основного аккаунта. После этого сервис должен слить свои локальные аккаунты, используя старый и новый идентификаторы. Затем можно повторить запрос к ТГУ.Аккаунты, используя уже новый идентификатор аккаунта. Подробно алгоритм отправки запроса и обработки ответа изображён на Рис. 11.

Рис. 18 Слитые аккаунты в разных сервисах. Красным помечены удаляемые элементы, зелёным - добавляемые. В зависимости от реализации алгоритма слияния в подключаемом сервисе данные из удаляемого объекта могут быть перекопированы в объект основного аккаунта.

Стоит отметить, что вся логика по обработке ответа ТГУ.Аккаунты о том, что нужно слить аккаунты, находится в одном месте, в библиотеке BaseGateway (Рис. 14), и используется повторно в каждом из подключаемых сервисов. Но логика непосредственно слияния реализуется в самом сервисе и зависит от его специфики.

ТГУ.Сотрудники

ТГУ.Сотрудники – это сервис, представляющий из себя каталог всех сотрудников университета с основной информацией о них. Источником данных является официальная информационная система отдела кадров. На сервисе не предусмотрена ручная регистрация. Регистрируются только те аккаунты, которые есть в информационной системе отдела кадров. Если из этой системы кто-то удаляется, он автоматически удаляется и с сервиса ТГУ.Сотрудники.

Рис. 19 Пример страницы сотрудника с основными данными о нём.

Однако информационная система отдела кадров не разрабатывается в данный момент и внести изменения в её код возможности нет, поэтому реализовать свои, наиболее подходящие способы синхронизации данных невозможно. Единственный способ синхронизации данных, который был предусмотрен в этой системе – это ручной экспорт xml-файлов с полными данными о каждом сотруднике и его должностях. Было решено раз в месяц получать такой файл и импортировать эти данные на ТГУ.Сотрудники. Для этого было реализовано типовое решение Адаптер [18].

При импорте данных очередного сотрудника могут возникать следующие ситуации.

Сотрудник ещё не имеет аккаунта ни на ТГУ.Сотрудники, ни на ТГУ.Аккаунты.

Сотрудник уже имеет аккаунт на ТГУ.Аккаунты, но не имеет на ТГУ.Сотрудники.

Сотрудник уже имеет аккаунт и на ТГУ.Сотрудники, и на ТГУ.Аккаунты.

Также возможен дополнительный вариант.

У пользователя есть аккаунт и на ТГУ.Сотрудники, и на ТГУ.Аккаунты, но его нет в информационной системе отдела кадров, т.е. он не является сотрудником университета.

Для начала необходимо определить, имеется ли аккаунт у сотрудника на ТГУ.Аккаунты. Поиск соответствующего аккаунта происходит по полному совпадению имени, фамилии, отчества и даты рождения. Если такого аккаунта нет, то он регистрируется. Если такой аккаунт найден, берётся его глобальный идентификатор AccountId и по нему ищется локальный аккаунт на ТГУ.Сотрудники. Если локальный аккаунт найден, то проверяются и обновляются все данные о нём. Если локальный аккаунт не был найден, то он регистрируется. Для того, чтобы удалять локальные аккаунты пользователей, которые перестали быть сотрудниками, перед процедурой импорта все локальные аккаунты помечаются флагом IsApproved = false. После того, как проверен очередной сотрудник и для него определен локальный аккаунт, для него этот флаг меняется на IsApproved = true. Таким образом после процедуры импорта необходимо удалить все локальные аккаунты, у которых флаг IsApproved = false. В итоге данные на сервисе ТГУ.Сотрудники полностью синхронизируются с информационной системой отдела кадров.

Ещё одной задачей на сервисе ТГУ.Сотрудники стала синхронизация данных о заработной плате сотрудников. Источником для этих данных является информационная система бухгалтерии университета. Также, как и в отделе кадров, эта система не разрабатывается в данный момент и нет возможности внести изменения в её код. Единственным способом получить из неё данные является ручной экспорт текстового файла, содержимое которого представлено в своём собственном формате, не предназначенном для машинного чтения, а не на одном из формальных языков. Кроме того, часть данных в этом файле (а именно статьи доходов/расходов и названия отделов) представлены в зашифрованном виде. Расшифровать данные можно с помощью вспомогательных данных, содержащихся в файлах в формате .xls, сопоставив коды. Файлы .xls также не были предназначены для машинного чтения, хотя строго соответствовали своему формату.

Для решения задачи импорта данных о заработной плате был реализован адаптер, который читает и анализирует файл с данными о зарплате, а также вспомогательные файлы, сопоставляет коды и сохраняет результат на сервисе ТГУ.Сотрудники. Для чтения .xls формата была использована библиотека NPOI [19]. Данная процедура инициируется вручную раз в месяц, после начисления заработной платы сотрудникам.

Рис. 20 Пример страницы с данными о заработной плате сотрудников.

В целях безопасности, эти данные импортируются только для тех сотрудников, которые подтвердили свой доступ к аккаунту письменным заявлением. Процесс подтверждения аккаунтов был описан в разделе, посвященном центру аутентификации ТГУ.Аккаунты. Кроме того, в целях безопасности все данные о зарплатах сотрудников хранится в базе данных в зашифрованном виде. Все данные шифруются с помощью алгоритма симметричного шифрования Rijndael (AES). Расшифровываются данные непосредственно перед выводом их пользователю. Таким образом, если злоумышленник получит доступ к базе данных[1] сервиса, он не сможет[1] получить сами данные, т.к. ключ для расшифровывания хранится в коде сервиса.

В итоге была решена задача синхронизации данных о заработной плате сотрудников между информационной системой бухгалтерии университета и сервисом ТГУ.Сотрудники.

Ещё одной задачей, которая решалась при разработке сервиса ТГУ.Сотрудники, была задача получения данных о сотруднике из системы Результативности научной деятельности (РНД). У этой системы, в отличии от предыдущих случаев, имеется HTTP API, с помощью которой можно получать данные в режиме реального времени в формате xml. Логика получения данных не была подстроена под нужды сервиса ТГУ.Сотрудники или других сервисов. В связи с этим было решено выделить её в отдельный сервис, названный Шлюз РНД (Рис. 5). Этот сервис инкапсулирует в себе всю логику работы с API РНД, а также выполняет функцию кэша. Наружу он предоставляет API, удобное для использования в других сервисах, в том числе на ТГУ.Сотрудники. Основные данные, получаемые от РНД – список публикаций, участие в конференциях, патенты, участие в научных проектах, методические пособия и т.п.

Рис. 21 Пример страницы со списком публикаций сотрудника.

ТГУ.Профили

Рассмотрим архитектуру сервиса ТГУ.Профили. Все остальные сервисы, созданные в рамках этой работы, были построены по тем же принципам и отличаются лишь особенностями предметной области. ТГУ.Профили – это сервис, который позволяет любому пользователю системы вести свой собственный блог или участвовать в групповом блоге, а также позволяет вести страницу некоторой группы людей или структурного подразделения университета, заменяя им тем самым несложный информационный сайт.

На рисунке представлена упрощённая модель предметной области сервиса. Здесь центральную часть занимают классы Blog и User. Пользователь может являться владельцем нескольких блогов. Кроме того, пользователь может быть модератором блогов, участником или подписчиком. В блоке пользователь может писать посты (класс Post), к которым могут быть прикреплены изображения (класс Image). Кроме того, каждый пост может быть помечен несколькими тегами (класс Tag) для упрощения поиска постов по всему сервису. Пользователи могут комментировать любые посты (класс Comment), а к комментариям также прикреплять изображения и теги. Ко всем постам и комментариям пользователи могут добавить метку «Мне нравится» (класс Like). Кроме того, в блоге можно создавать альбомы изображений (класс Album). Для того, чтобы блог можно было использовать в качестве небольшого информационного сайта, к блогу можно прикреплять статически страницы (класс StaticPage).

Сервис реализован на фреймворке ASP.NET MVC на языке C#. В качестве базы данных использовался MS SQL Server. Для ORM использовался Entity Framework с подходом Code First.

Данные о пользователях, включая имя и фотографию, загружаются из сервиса ТГУ.Аккаунты. Межсервисное взаимодействие подробно описано в предыдущих частях.

Рис. 22 Упрощённая модель предметной области сервиса ТГУ.Профили

Заключение

В рамках данной работы были созданы следующие сервисы корпоративной информационной системы университета: ТГУ.Аккаунты, ТГУ.Сотрудники, ТГУ.Выпускники, ТГУ.Студенты, Шлюз РНД, ТГУ.Профили, ТГУ.Новости, ТГУ.Сообщения, ТГУ.Уведомления, ТГУ.Контакты, Карта инновационно-активной среды ТГУ.

Разработанные сервисы интегрированы с существующими в университете информационными системами с помощью адаптеров. Разработаны библиотеки для упрощения взаимодействия между сервисами. Создан единый центр аутентификации ТГУ.Аккаунты, реализующий протокол OAuth. Решена задача слияния аккаунтов.

Все поставленные задачи выполнены, цель достигнута.

Список литературы

1.

ServiceOrientedAmbiguity [Электронный ресурс] // Martin Fowler: [сайт]. [2005]. URL: http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html (дата обращения: 02.05.2016).

2.

Microservices [Электронный ресурс] // Martin Fowler: [сайт]. [2014]. URL: http://martinfowler.com/articles/microservices.html (дата обращения: 02.05.2016).

3.

Микросервисы [Электронный ресурс] // Хабрахбар: [сайт]. [2015]. URL: https://habrahabr.ru/post/249183/ (дата обращения: 02.05.2016).

4.

HOW BIG SHOULD A MICRO-SERVICE BE? [Электронный ресурс] // The Bovine Synchrotron, James Lewis's blog: [сайт]. [2013]. URL: http://bovon.org/archives/350 (дата обращения: 02.05.2016).

5.

What are Microservices? [Электронный ресурс] // Martin Fowler: [сайт]. URL: http://martinfowler.com/microservices/#what (дата обращения: 02.05.2016).

6.

What is an ESB? [Электронный ресурс] // MuleSoft: [сайт]. URL: https://www.mulesoft.com/resources/esb/what-esb (дата обращения: 03.05.2016).

7.

ESB на практике [Электронный ресурс] // IBM Bluemix: [сайт]. [2005]. URL: http://www.ibm.com/developerworks/ru/library/0509_flurry1/ (дата обращения: 03.05.2016).

8.

Does My Bus Look Big in This? [Электронный ресурс] // InfoQ: [сайт]. [2008]. URL: http://www.infoq.com/presentations/soa-without-esb#anch24626 (дата обращения: 03.05.2016).

9.

Service-Oriented Architecture (SOA) [Электронный ресурс] // urlIntegration: [сайт]. URL: http://www.urlintegration.com/?page_id=752 (дата обращения: 03.05.2016).

10.

Understanding Enterprise Application Integration - The Benefits of ESB for EAI [Электронный ресурс] // MuleSoft: [сайт]. URL: https://www.mulesoft.com/resources/esb/enterprise-application-integration-eai-and-esb (дата обращения: 03.05.2016).

11.

// ASP.NET: [сайт]. URL: http://www.asp.net/ (дата обращения: 15.05.2016).

12.

//.NET: [сайт]. URL: https://www.microsoft.com/net/default.aspx (дата обращения: 15.05.2016).

13.

Learn About ASP.NET Web API [Электронный ресурс] // ASP.NET: [сайт]. URL: http://www.asp.net/web-api (дата обращения: 15.05.2016).

14.

Architectural Styles and the Design of Network-based Software Architectures [[1] Электронный ресурс] // UCI: [сайт]. [2000]. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm ([5] дата обращения: 15.05.2016).

15.

AMOP is the Internet Protocol for Business Messaging [Электронный ресурс] // AMOP: [сайт]. URL: https://www.amqp.org/about/what (дата обращения: 15.05.2016).

16.

// RabbitMQ: [сайт]. URL: https://www.rabbitmq.com/ (дата обращения: 15.05.2016).

17.

// EasyNetQ: [сайт]. URL: http://easynetq.com/ (дата обращения: 15.05.2016).

18.

Фаулер М. Архитектура корпоративных программных приложений. Москва: "Вильяме", 2006. 544 с.

19.

NPOI [Электронный ресурс] // CodePlex: [сайт]. URL: https://npoi.codeplex.com/ (дата обращения: 18.05.2016).