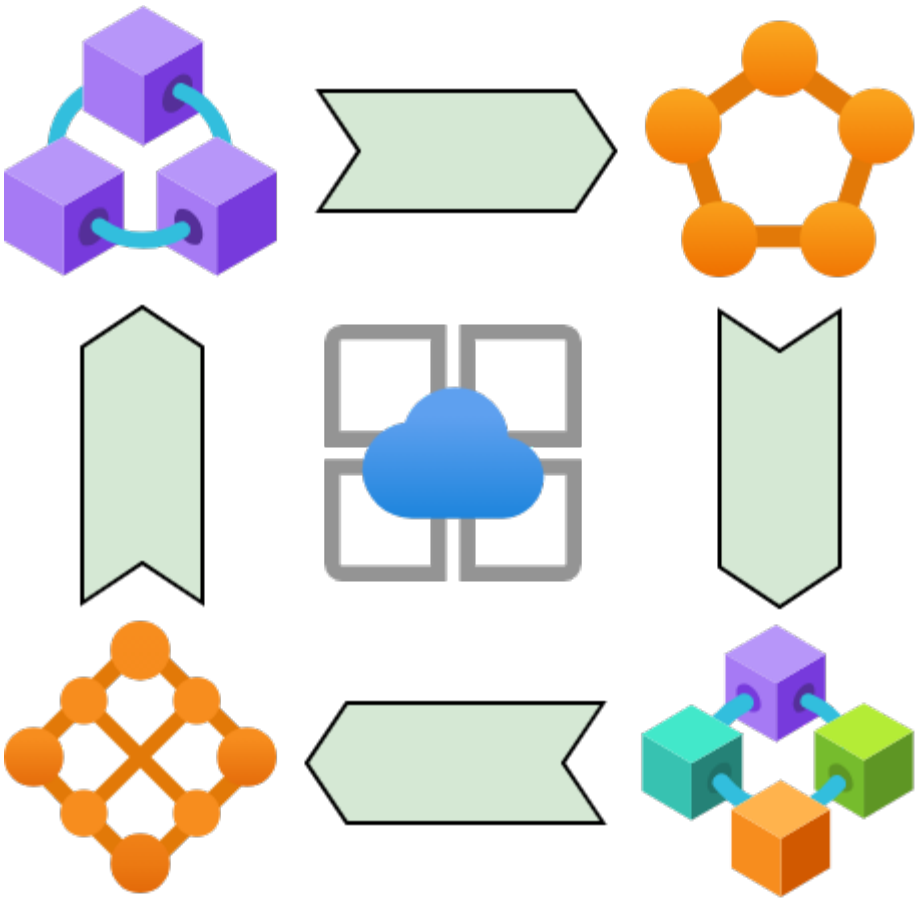
**А. А. Рындин, Э. Р. Саргсян**

# СОВРЕМЕННЫЕ СТАНДАРТЫ ИНФОРМАЦИОННОГО ВЗАИМОДЕЙСТВИЯ СИСТЕМ

**Учебное пособие**



**Воронеж 2021**

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Воронежский государственный технический университет»

**А. А. Рындин, Э. Р. Саргсян**

# СОВРЕМЕННЫЕ СТАНДАРТЫ ИНФОРМАЦИОННОГО ВЗАИМОДЕЙСТВИЯ СИСТЕМ

Учебное пособие

Воронеж 2021

УДК 681.3(075.8)
ББК 32.81я7
Р952

***Рецензенты:***
*кафедра информационных технологий управления*
*Воронежского гогсударственного университета*
*(зав. кафедрой д-р техн. наук, проф. М. Г. Матвеев);*
*д-р техн. наук, проф. О. Я. Кравец*

**Рындин, А. А.**

Р952 Современные стандарты информационного взаимодействия систем: учеб. пособие / А. А. Рындин, Э. Р. Саргсян; ФГБОУ ВО «Воронежский государственный технический университет». – Воронеж: Изд-во ВГТУ, 2021. 144 с.

ISBN 978-5-7731-0943-3

В учебном пособии приводится теоретический и практический материал согласно программе дисциплины, а также контрольные вопросы и задания. Представлены основные задачи и способы интеграции в современных системах. Описано множество инструментов, которые используются в процессе интеграции систем для выполнения задач передачи, защиты и анализа данных, а также инструменты для обработки сообщений и команд. Рассмотрены основные форматы и протоколы для передачи данных между системами.

Предназначено для студентов 1 курса магистратуры направления 09.04.02 «Информационные системы и технологии» (программа магистерской подготовки «Разработка Web-ориентированных информационных систем») при изучении дисциплины «Современные стандарты информационного взаимодействия систем».

Ил. 22. Табл. 5. Библиогр.: 27 назв.

**УДК 681.3(075.8)**
**ББК 32.81я7**

Научный редактор д-р техн. наук, проф. Я. Е. Львович

*Печатается по решению редакционно-издательского совета*
*Воронежского государственнного технического университета*

ISBN 978-5-7731-0943-3

# ВВЕДЕНИЕ

Множество сфер современной жизни, будь то экономика, технологии, производство или что-либо другое, неразрывно связаны с использованием информационных технологий, которые на данный момент являются уже обязательной частью нашей жизни. С помощью информационных технологий реализовываются системы производства товара, его транспортировки, связи с заказчиками, оплаты, интеграции и многое другое.

В связи с этим на данный момент становится всё больше специалистов в отраслях, связанных с информационными технологиями, при том совсем различного профиля. Это всё способствует развитию информационных технологий на совершенно новом уровне и подразумевает более глубокую интеграцию этих технологий в уже привычные системы. Зачастую подобные процессы требуют улучшений в архитектуре сети. Так как от качества подобных сетей напрямую зависит возможность работы приложений. Для этих целей используются различные способы оптимизации работы телекоммуникационных сетей и уменьшения затрат на их обслуживание.
Совокупность всех этих условий позволяет задавать современные стандарты информационного взаимодействия между системами.

Соответственно это требует выдвигать довольно высокие требования к IT специалистам, работающим в этих сферах, и к самим разрабатываемым технологиям. IT специалист должен обладать набором теоретических и эмпирических знаний, которые позволяют оценивать и классифицировать всю полученную информацию. Одной из особенностей данной отрасли является необходимость в постоянном обучении специалистов, так как применяемые технологии меняются из года в год и необходимо поддерживать высокий уровень знаний.

Курс «Современные стандарты информационного взаимодействия систем» ставит задачу формирования у учащихся необходимого минимума знаний, позволяющего воспринимать информацию о современных информационных технологиях, уметь воспользоваться ею для решения поставленных задач и владеть необходимыми навыками и технологиями, применение которых будет упрощать решение поставленных задач.

Учебное пособие состоит из 15 глав, каждая из которых затрагивает различные аспекты организации работы в рамках современных стандартов информационного взаимодействия систем. Помимо основного материала в каждой главе также присутствует набор контрольных вопросов, которые необходимы для проверки усвоения учебного материала учащимися.

Рисунки использованы с открытых источников интернета.

Пособие предназначено для студентов магистратуры направления 09.04.02 «Информационные системы и технологии», обучающихся по направлению «Разработка Web-ориентированных информационных систем».

# 1. ЗАДАЧИ И ВИДЫ ИНТЕГРАЦИИ ИНФОРМАЦИОННЫХ СИСТЕМ. ЧЕТЫРЕ СПОСОБА ИНТЕГРАЦИИ: ФАЙЛЫ, ОБЩАЯ БД, RPC, MESSAGES

Интеграция информационных систем (ИИС) — это процесс получения общего информационного пространства и организации поддержки процессов предприятий, необходимый для установки связей между информационными системами.

Основная задача ИИС разделяется на две части:

1. Интегрирование приложений.
2. Интегрирование данных.

Интеграция данных - процесс, применяемый к предприятиям и организациям, во время которого выполняются следующие задачи:

– агрегирование информации из различных информационных систем (ИС) предприятия;

– установка соответствия информации в разных системах (соответствие таблиц, записей и полей);

– синхронизация идентичных объектов в разных ИС.

При решении задачи интеграции необходимо провести стандартизацию нормативно-справочной информации (НСИ). НСИ – корпоративная информация, регламентирующая нормативно-справочные документы.

Для создания оптимальных и эффективных решений для интеграции необходимо иметь НСИ, которая удовлетворяет следующим свойствам:

– стандартизирована;

– структурирована;

– однозначна.

Интеграция приложений - процесс, который предназначен для установки и настройки взаимодействия ИС. Множество крупных компаний для решения данной задачи используют сервисную шину предприятия, так называемую Enterprise Service Bus (ESB). Это позволяет максимально сохранить текущее состояние каждый отдельно взятой ИС в предприятии и организовать процесс интеграции между данными ИС с помощью сервисной шины.

Интеграция приложений с помощью ESB - один из наиболее используемых инструментов, целью которого является создание общего информационного пространства внутри организации или предприятия, а также реализация возможности обмена информацией между всеми информационными системами предприятия без внесения каких-либо серьёзных изменений в их структуру.

## 1.1. Способы интеграции систем

Для решения задачи интеграции приложений используются следующие типы методов:

– обмен файлами;

– общая база данных;
– удалённый вызов;
– асинхронная передача сообщений.

**Обмен файлами**

Обмен файлами - один из самый используемых способов решения задачи интеграции. Имеет простую реализацию и поддерживает стандартные форматы обмена. Одним из примеров может быть формат CSV, который используется во множестве систем корпоративного назначения.

Однако у такого подхода есть недостатки. В случае если необходимо передавать большие и сложные структуры, то простые форматы обмена уже не пригодны. Возникающие в таких случаях специализированные форматы файлов должны «понимать» взаимодействующие системы, что ведет к жесткой зависимости систем друг от друга. Этот недостаток обычно преодолевают всевозможными утилитами конвертации данных.

**Общая база данных**

Данный подход концептуально очень прост - несколько информационных систем или приложений используют одну базу данных. Главный его недостаток - связь между интегрированными приложениями настолько тесная, что иногда невозможно заметить границу между ними (обычно так интегрируются продукты одного производителя). Примером такого подхода могут служить большинство ERP-систем, где различные модули системы используют одну базу [1].

**Удаленный вызов**

Стандарты на удаленный вызов процедур возникли два десятка лет назад, позволяя программному коду, который выполняется на одном компьютере, вызывать код на другом. Стандарты появлялись, развивались и угасали: RPC, CORBA, DCOM, RMI…, последним в этом ряду стал протокол SOAP, основа современных Web-сервисов. Собственно, в подходе к интеграции с использованием удаленных вызовов за эти годы ничего принципиально не изменилось — если приложению А что-то нужно от приложения Б, то А одним из перечисленных способов вызывает функцию приложения Б.

Основной недостаток удаленного вызова - требование работоспособности всех задействованных приложений в момент взаимодействия.

**Асинхронная передача сообщений**

Данный метод является одним из немногих, который целенаправленно создавался для процесса интеграции ИС. Идея концептуально проста и напоминает работу электронной почты. Когда приложению А необходимо вызвать какое-то действие в приложении Б, оно формирует соответствующее сообщение с данными и инструкциями и отправляет его посредством системы доставки сообщений. Слово «асинхронный» означает, что приложение А не должно ждать, пока сообщение дойдет до Б, будет обработано, сформирован ответ и т.п.

Недостаток данного подхода - высокая цена. Система гарантированной доставки на основе очередей сообщений обычно сама по себе недешева; единственным известным мне исключением является Microsoft Message Queue (MSMQ), компонент серверных операционных систем семейства Windows.

### 1.2. Контрольные задания

#### 1.2.1. Вопросы для самопроверки

1. Задачи интеграции информационных систем.
2. Интеграция данных.
3. Интеграция приложений.
4. Нормативно-справочная информация.
5. Способы интеграции систем: обмен файлами.
6. Способы интеграции систем: общая база данных.
7. Способы интеграции систем: удалённый вызов.
8. Способы интеграции систем: асинхронный обмен сообщениями.

## 2. ИНТЕГРАЦИЯ НА ОСНОВЕ ОБЩЕЙ БД. ПРЕДСТАВЛЕНИЯ, МАТЕРИАЛИЗОВАННЫЕ ПРЕДСТАВЛЕНИЯ, DB-ЛИНКИ

Интеграция информационных систем (ИИС) - это процесс получения общего информационного пространства и организации поддержки процессов предприятий, необходимый для установки связей между информационными системами.

Многообразие применяемых технологий и систем, разнообразие форматов данных, циркулирующих в информационных потоках, обилие аналитических и отчётных форм сделали актуальной задачу интеграции технологических и информационных объектов и сущностей, а также физические и виртуальные пространства и их взаимодействия преобразуются в единую информационно-управленческую среду.

В данной главе будет рассмотрен метод интеграции на основе общей базы данных. Будет проведен анализ преимуществ и недостатков данного метода, а также определены понятия представления, материализованного представления, DB-линков.

**Факторы, влияющие на интеграцию информационных систем**

Интеграция информационных систем – комплексная многоуровневая задача. Чтобы получить более четкое понимание поставленной задачи, необходимо провести анализ факторов, влияющих на интеграцию. Интеграция ИС в контексте данной работы рассматривается в рамках компании или организации [2].

1. Ускорение развития организации. Этот процесс требует чаще менять структуры данных, бизнес-процессы, дизайн, пользовательский интерфейс. Эти

компоненты довольно динамичны, и именно в таких областях, где динамизм - часть сути и природы системы, задача интеграции существенно усложняется.

2. Распределенность. Организации становятся все более крупными, а решаемые задачи все более комплексными, появляется логическая, организационная и географическая рассредоточенность.

3. Неоднородность. В крупном проекте почти никогда нет возможности придерживаться платформ и инструментов от одного производителя, поэтому в процессе интеграции необходимо учитывать необходимость поддержки нескольких платформ.

4. Наследственность. Невозможность полностью отказаться от наследия систем, морально устаревших технологий и аппаратного обеспечения, которые могут давать приемлемые показатели по надежности и производительности, но существенно усложняют интеграцию.

5. Хаотичность. Не всегда есть возможность полностью формализовать, специфицировать и структурировать данные, и часть модели остается слабо связанной, не поддающейся или слабо поддающейся машинной обработке, анализу, индексации.

6. Обусловленность. Информационные системы ограничены не только техническими рамками, но и человеческим фактором, особенностями законодательства, множеством других факторов, не зависящих от разработчиков.

7. Мобильность. Пользователи чаще обращаются к системе «на ходу», а взаимодействие ведется через каналы связи общего пользования в транспорте, дома и на улице, в общественных местах.

8. Безопасность. Этот фактор является логическим продолжением предыдущего (повышение мобильности). Передача данных через каналы общего пользования требует повышенного внимания к шифрованию.

Эти факторы в полной мере отражают сложность такой задачи, как интеграция систем. Существует также ряд параметров, влияющих на степень сложности интеграции. Ниже данные параметры будут рассмотрены, а также перечислены варианты минимизации их негативного влияния на процесс интеграции:

1. Концептуальная разница. Основана на том, что разработчики разных систем изначально приняли разные решения, предположения и допущения, которые концептуально не стыкуются между собой. Решается введением еще одного слоя абстракции, который концептуально не противоречит обоим подходам.

2. Технологическая разница. Основана на несовместимых форматах обмена данными, протоколах взаимодействия и интерфейсах. Решается написанием прослоек, брокеров и т. д.

3. Несовместимость лицензий. Решение данной проблемы индивидуально для каждого случая и решается на уровне организации.

Таким образом, задачу интеграции информационных систем можно обобщить так: необходимо интегрировать N информационных систем, характеризуемых описанными выше факторами, с минимизацией количества прослоек и интерфейсов между ними.

Есть различные способы решения поставленной задачи.

1. Стандартизация. Использование международных, государственных, отраслевых, корпоративных стандартов.

2. Интеграция на уровне брокеров.

3. Интеграция на уровне сервисов. Основана на фиксации интерфейсов и форматов данных с двух сторон и позволет наладить быструю отработку межкорпоративной бизнес-логики.

4. Интеграция на уровне пользователя. Неавтоматизированная интеграция, при которой пользователи перемещают данные между системами с помощью копирования, переноса файлов, т.е. вручную. Плохая практика интеграции, применяемая, однако, в ряде сложных случаев.

5. Динамическая интерпретация метаинформации. Это процесс создания модели предметной области из метамодели, метаданных и данных. Динамическая интерпретация происходит как на клиенте, так и на сервере в момент исполнения. Метамодель содержит спецификацию метаданных, которые необходимы для динамической интерпретации. Получение измененных метаданных приводит к перестройке модели предметной области в оперативной памяти.

6. Интеграция на уровне данных. Несколько приложений могут обращаться в одну базу данных или в несколько баз данных, связанных репликациями [3].

## 2.1. Интеграция на основе общей базы данных

Данный подход зачастую используется при клинт-серверной архитектуре приложений. В таком подходе интегрированная база данных, к которой имеют доступ все системы, является основным системообразующим фактором при реализации интеграции.

Интеграция на основе общей базы данных - один из наиболее распространенных подходов к интеграции корпоративных информационных систем. В качестве примера работы этого сценария часто рассматривается доступ нескольких приложений к общей выделенной базе данных. Однако на практике использование базы данных намного разнообразнее. Встречаются такие сценарии, как интерфейсная таблица, изменение состояний объекта, нотификация, а также ряд других.

Появление таких дополнительных сценариев обосновано особенностями общей базы данных, используемой несколькими приложениями. Простая схема для такого сценария представлена на рис. 2.1.

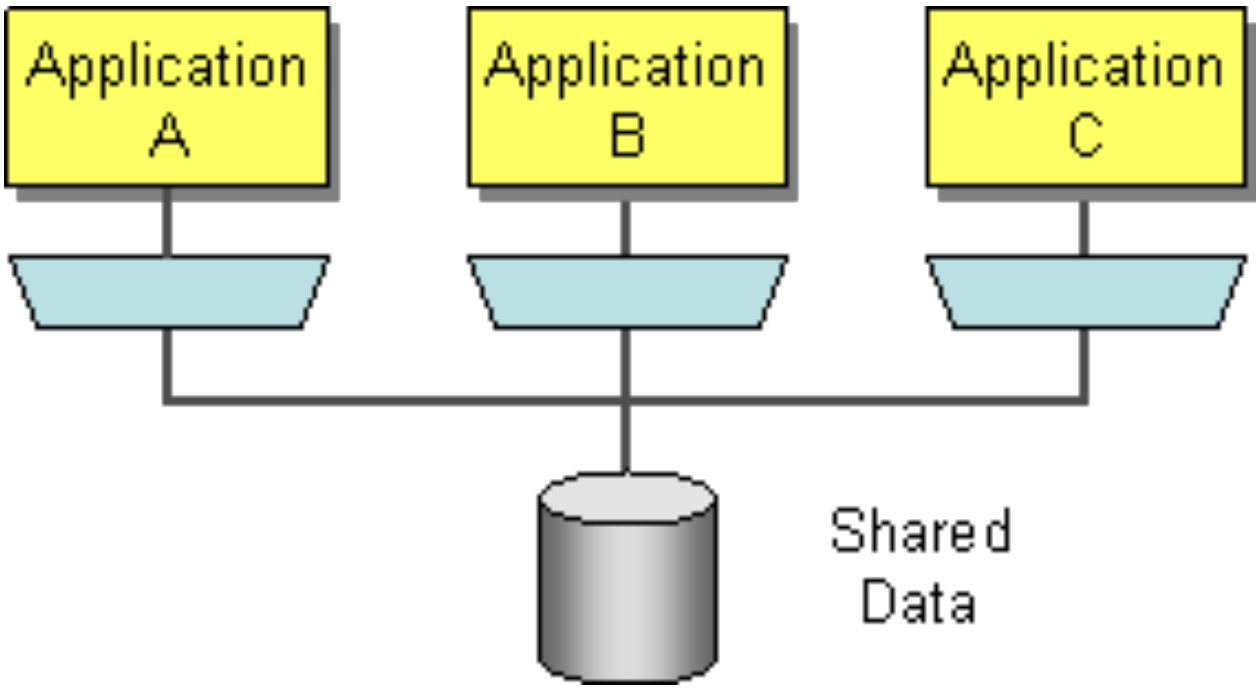



Рис. 2.1. Общая база данных, используемая несколькими приложениями

Считается, что данный способ интеграции - самый простой и наименее эффективный. В качестве аргументов можно привести следующее:

- изменения структуры базы данных требуют согласования со всеми использующими её приложениями;
- низкий уровень абстракции. Приложения работают не с логическими объектами, а с конкретными полями и записями;
- совместное использование базы данных требует большого запаса производительности, так как различные приложения создают различную нагрузку на сервер. Отладка совместного доступа усложнена и требует существенных трудозатрат;
- необходимость передачи больших объемов данных.

В ряде случаев данный подход имеет свои преимущества.

Сценарий интеграции на основе общей базы данных хорошо работает в условиях жестких финансовых ограничений: отсутствие ресурсов на создание реплик или ежедневных копий. При этом необходимо учитывать, что при интеграции на основе общей БД возрастут расходы на оборудование и лицензии СУБД.

Системы управления реляционными базами данных наиболее распространенный и наиболее эффективный инструмент последних десятилетий. Большинство разработчиков благодаря этому привыкли к ним и работают в рамках заложенных в системах ограничений. В качестве примера можно привести протокол доступа. Доступ к БД – это протокол, требующий установления соединения и обрабатывающий запросы в рамках этого соединения. Такие сервисы сложно масштабировать, это связано с разными

подходами к оптимизации для чтения и добавления данных, а также рядом других факторов.

В заключение можно сказать, что интеграция на основе общей базы данных – пригодный для практической реализации сценарий. Для ряда задач, например, общекорпоративных справочников – это один из немногих доступных подходов к интеграции, а ряд других задач можно решить с помощью репликации, а также с помощью улучшенных сценариев интеграции на основе той же концепции.

Для более полного понимания принципа работы с общей БД рассмотрим понятия представления, материализованного представления, DB-линков.

### 2.2. Представления, материализованные представления

Представление — представляет из себя поименованный запрос, который подставляется в запрос, использующий это представление, как подзапрос. Внешне выглядит как виртуальная таблица.

Отличается от обычной таблицы тем, что данные из представления не являются отдельными объектами, которые физически хранятся в базе данных, а всего лишь ссылаются на данные, которые физически расположены в других таблицах.

Исходя из этого, при изменении этих данных в физических таблицах все представления, которые построены на основании этой таблицы, будут также обновлены.

Типичным способом создания представлений для СУБД, поддерживающих язык запросов SQL, является связывание представления с определённым SQL-запросом. Так, для типичных СУБД, таких как PostgreSQL, Interbase, Firebird, Microsoft SQL Server, Oracle, представление может содержать:

1. Подмножество записей из таблицы БД, отвечающее определённым условиям.
2. Подмножество столбцов таблицы БД, требуемое программой.
3. Результат обработки данных таблицы определёнными операциями.
4. Результат объединения нескольких таблиц.
5. Результат слияния нескольких таблиц с одинаковыми именами и типами полей, когда в представление попадают все записи каждой из сливаемых таблиц.
6. Результат группировки записей в таблице.
7. Практически любую комбинацию вышеперечисленных возможностей.

Пример представления, реализованного на языке SQL, изображен на рис. 2.2.

```
SELECT *
   FROM Londonstaff;
```

```
==============  SQL Execution Log ============
|                                              |
| SELECT *                                     |
| FROM  Londonstaff;                           |
|                                              |
| ============================================ |
|   snum      sname         city         comm  |
| ------    ----------   -----------   ------- |
|   1001      Peel          London       0.1200 |
|   1004      Motika        London       0.1100 |
|                                              |
 ============================================
```

Рис. 2.2. Пример представления на языке SQL

Представления используются в запросах к БД тем же образом, как и обычные таблицы. В случае SQL-СУБД имя представления может находиться в SQL-запросе на месте имени таблицы (в предложении FROM). Запрос из представления обрабатывается СУБД точно так же, как запрос, в котором на месте имени представления находится подзапрос, определяющий это представление. При этом СУБД с развитыми возможностями оптимизации запросов перед выполнением запроса из представления могут проводить совместную оптимизацию запроса верхнего уровня и запроса, определяющего представление, с целью минимизации затрат на выборку данных.

Поскольку SQL-запрос, выбирающий данные представления, зафиксирован на момент его создания, СУБД получает возможность применить к этому запросу оптимизацию или предварительную компиляцию, что положительно сказывается на скорости обращения к представлению, по сравнению с прямым выполнением того же запроса из прикладной программы.

Существуют также специфические типы представлений. Некоторые СУБД имеют расширенные представления для данных, доступных только для чтения. Например, в СУБД Oracle и Microsoft SQL Server реализована концепция «материализованных представлений» — представлений, содержащих предварительно выбранные невиртуальные наборы данных,

совместно используемых в распределённых БД. Эти данные извлекаются из различных удалённых источников (с разных серверов распределённой СУБД). Целостность данных в материализованных представлениях поддерживается за счёт периодических синхронизаций или с использованием триггеров.

Материализованное представление — физический объект базы данных, содержащий результат выполнения запроса.

Материализованные представления позволяют многократно ускорить выполнение запросов, обращающихся к большому количеству (сотням тысяч или миллионам) записей, позволяя за секунды (и даже доли секунд) выполнять запросы к терабайтам данных. Это достигается за счет прозрачного использования заранее вычисленных итоговых данных и результатов соединений таблиц. Предварительно вычисленные итоговые данные обычно имеют очень небольшой объем по сравнению с исходными данными.

Целостность данных в материализованных представлениях поддерживается за счёт периодических синхронизаций или с использованием триггеров.

В общем случае представления доступны только для чтения данных, и их изменение или запись невозможна. Однако бывают исключения, например, в СУБД Oracle с помощью представлений можно редактировать данные, и эти изменения попадут в физическую таблицу. Однако здесь должны соблюдаться следующие условия:

– представление должно быть сформировано на основе одной физической таблицы;

– каждой записи в представлении должна соответствовать только одна запись в физической таблице, на основе которой построено это представление;

– в представлении присутствует первичный ключ физической таблицы, на основе которой построено это представление.

## 2.3. Database Links (DB-линки)

**Связь базы данных (database link)** - это одностороннее соединение локальной базы данных с удаленной базой данных. Связь всегда односторонняя. Пользователи удаленной базы не могут применять ее для подключения к локальной базе — вместо этого они должны создать отдельную связь базы данных. Пример связи баз данных изображен на рис. 2.3.

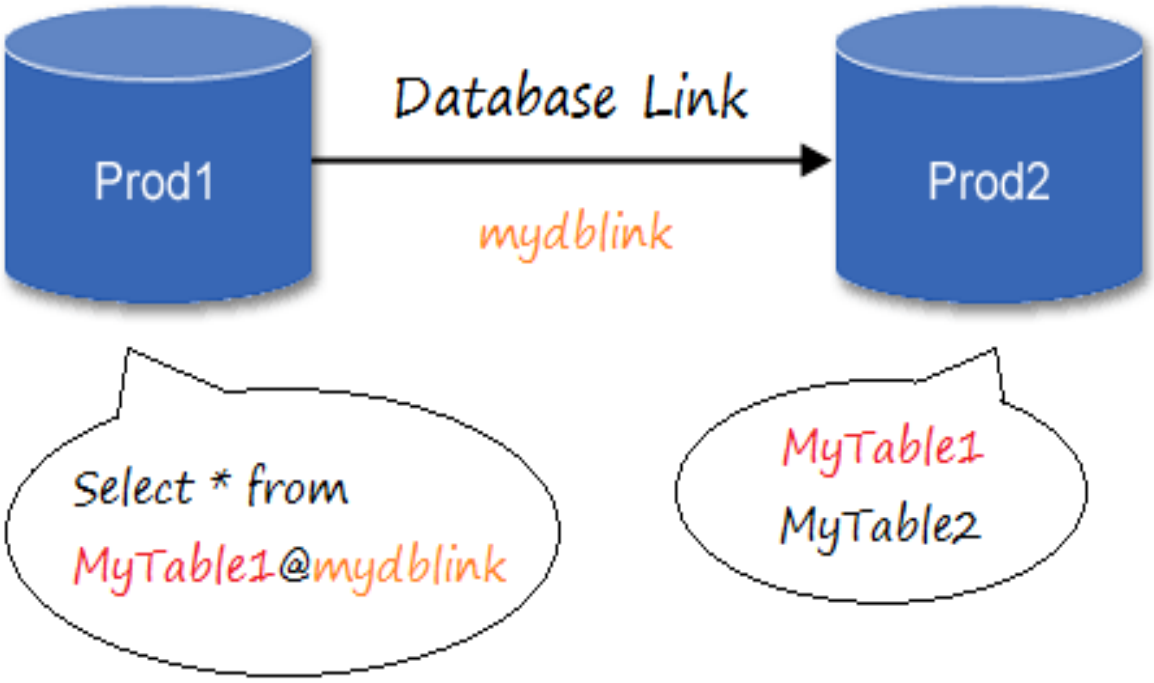



Рис. 2.3. Пример связи базы данных

Database Link позволяет получать доступ к разным базам данных через учетную запись пользователя удаленной базы; пользователь не обязан иметь учетную запись в удаленной базе данных. Привилегии в этой базе данных будут идентичны привилегиям пользовательской учетной записи, которая применялась для создания связи. Связи баз данных удобны, когда необходимо запросить таблицу в распределенной базе данных или даже вставить данные из таблицы другой базы в собственную локальную таблицу. Связи баз данных позволяют пользователям обращаться к множеству баз данных как к единой логической базе данных.

Если рассмотреть возможности применения Database Link в Oracle, то DB-линк - это объект SCHEMA в Oracle, который представляет собой мост для соединения с другой базой данных, помогающий вам иметь доступ к объектам другой базы данных.

Database Link также можно использовать для того, чтобы соединить Oracle с другим видом базы данных (MySQL, SQL Server и др.). В данном случае необходимо использовать сервис Oracle Heterogeneous.

В заключение можно сказать, что связь базы данных (Database Link) представляет собой удобное средство интеграции, которое применяется в том числе для соединения различных видов БД. Интеграция разных платформ, созданных с разными целями и философией, является одним из ключевых факторов, влияющих на степень сложности интеграции информационных систем.

### 2.4. Контрольные задания

#### 2.4.1. Вопросы для самопроверки

1. Интеграция информационных систем.
2. Факторы, влияющие на интеграцию информационных систем.
3. Параметры, влияющие на степень сложности интеграции.
4. Способы решения задачи интеграции информационных систем.
5. Интеграция на основе общей базы данных.
6. Представления.
7. Материализованные представления.
8. Database Link.

## 3. ИНТЕГРАЦИЯ И РАСПАРАЛЛЕЛИВАНИЕ БАНКОВ ДАННЫХ И NOSQL-ХРАНИЛИЩ. ПРИНЦИП ГЛОБАЛЬНОГО ID И МЕТОДЫ ГЕНЕРАЦИИ. ШАРДИНГ

Система управления базами данных (СУБД) - это программное обеспечение, состоящие из набора инструментов общего или специального назначения, для управления базой данных.

Существует несколько классификаций СУБД:

- по модели данных;
- по степени распределённости;
- по способу доступа к БД.

Среди СУБД, классифицируемых по модели данных, выделяют:

- иерархические;
- сетевые;
- реляционные (от англ. Relation - отношение);
- объектно-ориентированные;
- объектно-реляционные.

На практике наиболее используемы реляционные СУБД [4].

Реляционные СУБД - СУБД, управляющие реляционными базами данных. Реляционная модель ориентирована на представление данных в виде двумерных таблиц. Такое представление удобно для пользователей. Но реляционные базы данных не могут справляться с нагрузками, актуальными в наше время.

Три основных проблемы РСУБД:

- горизонтальное масштабирование при больших объёмах данных;
- производительность отдельного сервера;
- сложность изменения дизайн-логической структуры.

Всё это заставляет разработчиков приглядываться к альтернативам реляционных баз данных, используемым вот уже более тридцати лет.

В совокупности все эти технологии известны как «NoSQL базы данных».

В ходе выполнения курсовой работы были поставлены следующие задачи:

- освоение принципов работы Шардинга;
- освоение принципов работы NoSQL;
- освоение принципов глобального ID и методов его генерации.

Изучение методов интеграции и распараллеливания для банков данных и NoSQL-хранилищ.

**Что такое NoSQL**

NoSQL-способ структуризации данных заключается в избавлении от ограничений при хранении и использовании информации. Базы данных NoSQL, используя неструктуризированный подход, предлагают много эффективных способов обработки данных в отдельных случаях (например, при работе с хранилищем текстовых документов).

NoSQL-СУБД не используют реляционную модель структуризации данных. Существует много реализаций, рещающих этот вопрос по-своему, зачастую весьма специфично. Эти бессхемные решения допускают неограниченное формирование записей и хранение данных в виде ключ-значение.

В отличие от традиционных РСУБД, некоторые базы данных NoSQL, например, MongoDB, позволяют группировать коллекции данных с другими базами данных. Такие СУБД хранят данные как одно целое. Эти данные могут представлять собой одиночный объект наподобие JSON и вместе с тем корректно отвечать на запросы к полям.

NoSQL базы данных не используют общий формат запроса (как SQL в реляционных базах данных). Каждое решение использует собственную систему запросов [5].

Сравнение SQL и NoSQL.

Для того чтобы прийти к простому и понятному выводу, давайте проанализируем разницу между SQL- и NoSQL-подходами:

- структура и тип хранящихся данных: SQL/реляционные базы данных требуют наличия однозначно определённой структуры хранения данных, а NoSQL базы данных таких ограничений не ставят;
- запросы: вне зависимости от лицензии, РСУБД реализуют SQL-стандарты, поэтому из них можно получать данные при помощи языка SQL;
- масштабируемость: оба решения легко растягиваются вертикально (например, путём увеличения системных ресурсов);
- надёжность: когда речь заходит о надёжности, SQL базы данных однозначно впереди;
- поддержка: РСУБД имеют очень долгую историю. Они очень популярны, и поэтому получить поддержку, платную или нет, очень легко;

• хранение и доступ к сложным структурам данных: по своей природе реляционные базы данных предполагают работу со сложными ситуациями, поэтому и здесь они превосходят NoSQL-решения.

## 3.1. Шардинг

Масштабирование баз данных - самая сложная задача во время роста проекта. 90 % всех усилий обычно приходится как раз на работу, связанную с ростом объема данных и операций с ними. Один сервер базы данных в какой-то момент перестает справляться с нагрузкой. В этот момент и следует применять техники масштабирования.

В основе масштабирования данных лежит тот же принцип, что и в основе масштабирования Web-приложений. Это разделение данных на группы и выделение их на отдельные сервера. Существует две основные стратегии — репликация и шардинг.

Репликация позволяет создать полный дубликат базы данных. Так, вместо одного сервера их будет несколько.

Шардинг (иногда шардирование) - это другая техника масштабирования работы с данными. Суть его в разделении (партиционирование) базы данных на отдельные части так, чтобы каждую из них можно было вынести на отдельный сервер. Этот процесс зависит от структуры базы данных и выполняется прямо в приложении, в отличие от репликации [6]. Схема шардинга продемонстрирована на рис. 3.1.

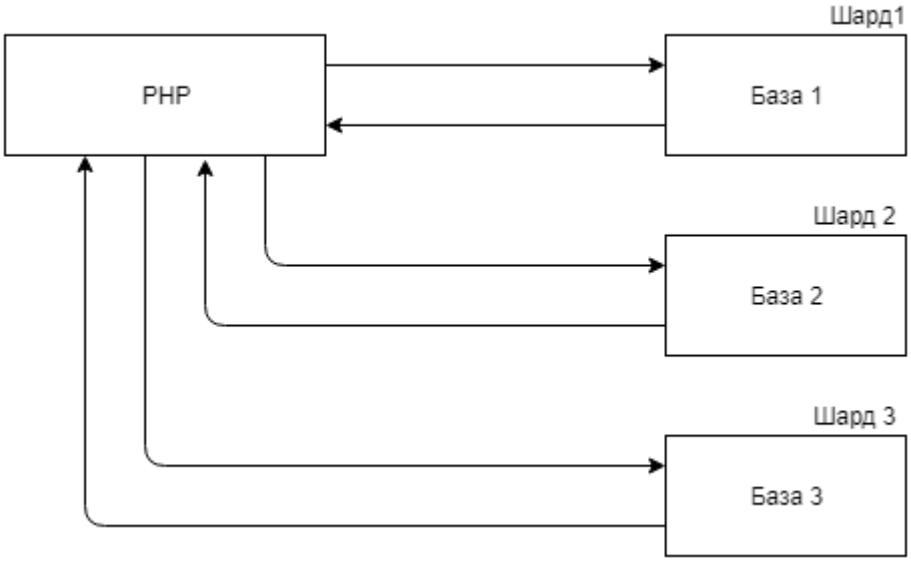



Рис. 3.1. Шардинг

В NoSQL базах данных шардинг, как и репликация, производится автоматически самой базой и пользовательское приложение обособленно от этих сложных механизмов.

Вертикальный шардинг - это выделение таблицы или группы таблиц на отдельный сервер. Например, в приложении есть такие таблицы:

- users - данные пользователей;
- photos - фотографии пользователей;
- albums - альбомы пользователей.

Таблица users остается на одном сервере, а таблицы photos и albums переносятся на другие. В таком случае в приложении необходимо прописать соответствующее соединение для работы с каждой таблицей отдельно.

Горизонтальный шардинг — это разделение одной таблицы на разные сервера. Это необходимо использовать для огромных таблиц, которые не умещаются на одном сервере. Разделение таблицы на куски делается по такому принципу:

- на нескольких серверах создается одна и та же таблица (только структура, без данных);

- в приложении выбирается условие, по которому будет определяться нужное соединение (например, четные на один сервер, а нечетные — на другой);

- перед каждым обращением к таблице происходит выбор нужного соединения.

Допустим, наше приложение работает с огромной таблицей, которая хранит фотографии пользователей. Например, имеются два сервера (обычно они называются шардами). Для нечетных пользователей мы будем работать с первым сервером, а для четных — со вторым. Таким образом, на каждом из серверов будет только часть всех данных.

## 3.2. Работа с NoSQL

Подключение к СУБД Rethinkdb:

```
Conn = r.connect("localhost", 28015)
```

Пример создания базы:

```
r.db_create("food").run(conn)
```

Создание таблицы:

```
r.db("food").table_create("favorites").run()
```

Заполнение данными:

```
r.db("food").table("favorites").insert([
  { "person": "Maks", "Age": 26,
    "fav_food": [
      "banana",
      "cereal",
      "spaghetti"
    ]
  },
  { "person": "Maria", "Age": 14,
```

```
        "fav_food": [
          "cookies",
          "apples",
          "cake",
          "sandwiches"
        ]
      },
      { "person": "Igor", "Age": 22,
        "fav_food": [
          "grapes",
          "pie",
          "avocado"
        ]
      }
    ]).run()
```

Подключение к СУБД Mongodb:

```
public MongoDBContext(string ConnectionString, string User, string Password, string Database) {
    var mongoUrlBuilder = new MongoUrlBuilder(ConnectionString);
    server = MongoServer.Create(mongoUrlBuilder.ToMongoUrl());
    MongoCredentials credentials = new MongoCredentials(User, Password);
    database = server.GetDatabase(Database, credentials);}
```

Для добавления объекта в коллекцию необходимо получить коллекцию по имени:

```
var collection = database.GetCollection<T>(table);
```

Для добавления можно выполнить команду:

```
collection.Insert<T>(obj);
```

или (как и для изменения)

```
collection.Save<T>(obj);
```

MongoDb самостоятельно добавляет поле _id — уникальный параметр. Если при выполнении команды Save у объекта будет Id, существующий уже в коллекции, то выполнится апдейт этого объекта.

Для удаления по id создается запрос (Query). В данном случае это

```
var query = Query.EQ("_id", id),
```

и выполняется команда:

```
collection.Remove(query);
```

### 3.3. Принцип глобального ID и методы генерации

GUID (Globally Unique Identifier) - статистически уникальный 128-битный идентификатор. Его главная особенность - уникальность, которая позволяет создавать расширяемые сервисы и приложения без опасения конфликтов, вызванных совпадением идентификаторов. Хотя уникальность каждого отдельного GUID не гарантируется, общее количество уникальных ключей настолько велико (2128 или 3,4028×1038), что вероятность того, что в мире будут независимо сгенерированы два совпадающих ключа, крайне мала.

Генерация случайных знаков:

```
$length = 11;
$chars=
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXY
Z_-';
$uid = '';
while ($length-- > 0) {
$uid .= $chars[random_int(0, 63)];}
var_dump($uid); // 4rnQMtJ4HRw
```

Плюсы:

- управляемость длиной идентификатора;
- нет ограничений в длине идентификатора;
- возможность изменить сами символы в наборе символов;
- возможность изменить количество символов в наборе;
- за счёт нескольких вызовов random_int() уменьшается вероятность локальных коллизий.

Минусы:

- несколько вызовов random_int() негативно сказывается на производительности.

Побайтовая генерация ID:

```
$length = 64;
$uid = '';
while ($length-- > 0) {
$uid .= random_int(0, 1);}
$uid = bindec($uid);
$uid = dechex($uid);
$uid = hex2bin($uid);
$uid = base64_encode($uid);
$uid = str_replace(['=', '+', '/'], ['', '-', '_'], $uid);
var_dump($uid); // tDiGk9YyWAA
```

Плюсы:

- управляемость длиной идентификатора;

- более частый вызов random_int(), по сравнению с предыдущим вариантом, уменьшает вероятность локальных коллизий.

Минусы:

- теряется возможность изменить сами символы в наборе символов;
- теряется возможность изменить количество символов в наборе;
- идентификатор ограничен 64 битами;
- несколько вызовов random_int() негативно сказывается на производительности.

Случайные числа и временная метка:

```
$time = floor(microtime(true) * 1000);
$prefix = random_int(0, 0b111111111);
$suffix = random_int(0, 0b111111111);
$uid = 1 << (9 + 45 + 9);
$uid |= $prefix << (9 + 45);
$uid |= $time << 9;
$uid |= $suffix;
$uid = dechex($uid);
$uid = hex2bin($uid);
$uid = base64_encode($uid);
$uid = str_replace(['=', '+', '/'], ['', '-', '_'], $uid);
var_dump($uid); // vELDchIFvk0
```

Плюсы:

- уменьшается вероятность коллизии за счет использования временной метки.

Минусы:

- теряется возможность изменить сами символы в наборе символов;
- теряется возможность изменить количество символов в наборе;
- идентификатор ограничен 64 битами;
- затруднительно управлять длиной идентификатора;
- как и в случае с Snowflake, идентификаторы получаются схожими.

Случайные числа и плавающая временная метка:

```
$time = floor(microtime(true) * 1000);
$prefix_length = random_int(1, 18);
$prefix = random_int(0, bindec(str_repeat('1', $prefix_length)));
$suffix_length = 18 - $prefix_length;
$suffix = random_int(0, bindec(str_repeat('1', $suffix_length)));
$uid = 1 << ($suffix_length + 45 + $prefix_length);
$uid |= $prefix << ($suffix_length + 45);
$uid |= $time << $suffix_length;
$uid |= $suffix;
$uid = dechex($uid);
$uid = hex2bin($uid);
```

```
$uid = base64_encode($uid);
$uid = str_replace(['=', '+', '/'], ['', '-', '_'], $uid);
var_dump($uid); // 4WG5MmC3SQo
```

Плюсы:

- идентификаторы получаются менее схожими, чем в предыдущем примере, даже при использовании одной временной метки.

Минусы

- теряется возможность изменить сами символы в наборе символов;
- теряется возможность изменить количество символов в наборе;
- идентификатор ограничен 64 битами;
- затруднительно управлять длиной идентификатора;
- из-за непостоянной позиции временной метки коллизия все же возможна.

Генерация случайных байт:

```
$uid = random_bytes(8);
$uid = base64_encode($uid);
$uid = str_replace(['=', '+', '/'], ['', '-', '_'], $uid);
var_dump($uid); // BOjs1VmavxI
```

Плюсы:

- наиболее короткое и простое решение из всех;
- нет ограничений в длине идентификатора.

Минусы:

- теряется возможность изменить сами символы в наборе символов;
- теряется возможность изменить количество символов в наборе;
- усложнено управление длиной конечного идентификатора.

## 3.4. Интеграция и распараллеливание банков данных и NoSQL-хранилищ

Банк данных - автоматизированная информационная система централизованного хранения и коллективного использования данных. В состав банка данных входят одна или несколько баз данных, справочник баз данных, СУБД, а также библиотеки запросов и прикладных программ.

Банк данных (БнД) является современной формой организации хранения и доступа к информации. Существует множество определений банка данных. По наиболее распространенному мнению, банк данных – это система специальным образом организованных данных (баз данных), программных, технических, языковых, организационно-методических средств, предназначенных для обеспечения централизованного накопления и коллективного многоцелевого использования данных.

Поэтому в БнД предусматривается специальное средство приведения всех запросов к единой терминологии – словарь данных. Кроме того, используются специальные методы эквивалентных грамматических преобразований запросов

для построения оптимальных процедур их обработки, специальные методы доступа к одним и тем же данным различных пользователей при совпадении во времени поступивших запросов. Обычно со стороны внешних пользователей к банку данных формулируются следующие требования. Банк данных должен:

• удовлетворять актуальным информационным потребностям внешних пользователей, обеспечивать возможность хранения и модификации больших объемов многоаспектной информации, удовлетворять выявленным и вновь возникающим потребностям внешних пользователей;

• обеспечивать заданный уровень достоверности хранимой информации и ее непротиворечивость;

• обеспечивать возможность поиска информации по произвольной группе признаков;

• удовлетворять заданным требованиям по производительности при обработке запросов;

• обеспечивать выдачу информации пользователям в различной форме;

• обеспечивать простоту и удобство обращения внешних пользователей за информацией.

Банк данных включает в свой состав две основные компоненты: базу данных, которая есть не что иное, как даталогическое представление информационной модели предметной области, и систему управления базой данных (СУБД), с помощью которой и реализуются централизованное управление данными, хранимыми в базе, доступ к ним и поддержание их в состоянии, соответствующем состоянию предметной области. Структура банка данных изображена на рис. 3.2.

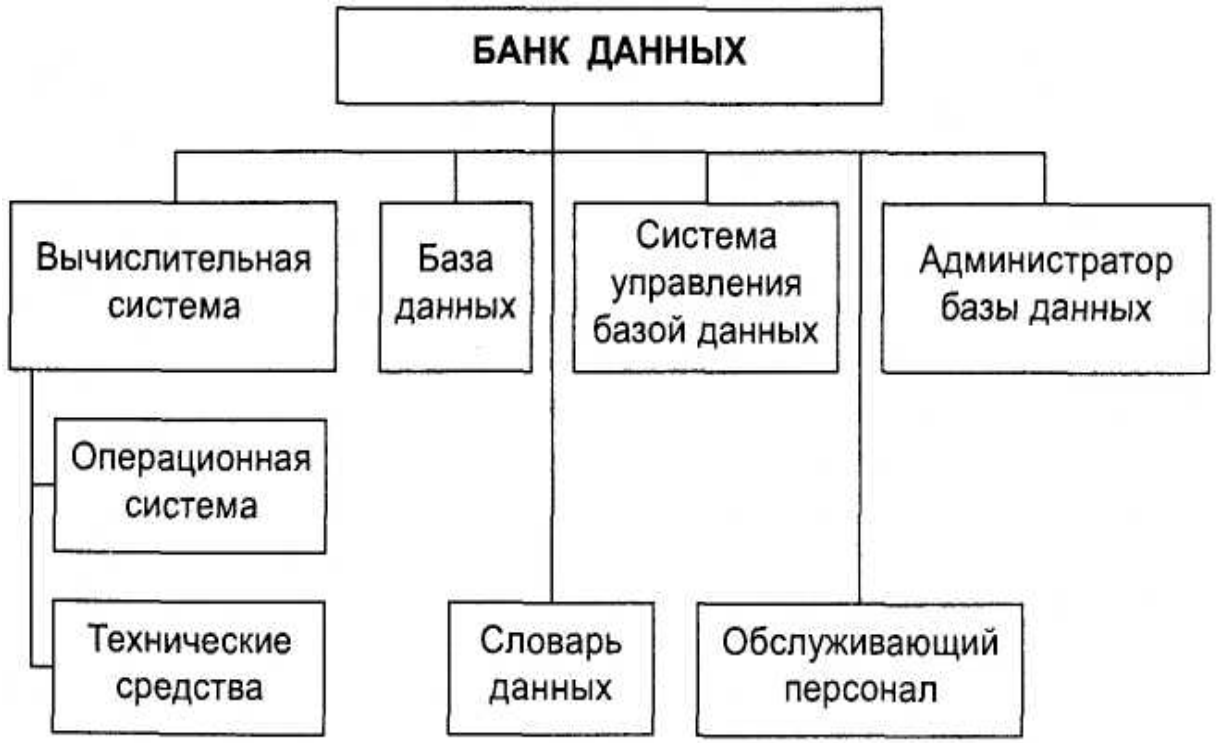



Рис. 3.2. Структура банка данных

NoSql хранилища бывают четырёх основных типов:

- структура «ключ-значение» — присутствует большая хеш-таблица, в которой хранятся ключи и значения;
- документоориентированное хранилище — хранилище состоит из элементов, у каждого из которых есть свой собственный тэгированный элемент;
- колоночное хранилище — в каждом блоке хранилища хранятся данные только из одной колонки;
- хранилище основаное на графах — база данных сетевого типа, которая использует узлы и рёбра для отображения и хранения данных.

База данных типа «ключ-значение»

Отсутствие схемы в базах данных «ключ-значение», например, Riak, — это как раз то, что вам нужно для хранения данных. Ключ может быть синтетическим или автосгенерированным, а значение может быть представлено строкой, JSON, блобом (BLOB, Binary Large Object, большой двоичный объект) и т.д.

Такие базы данных, как правило, используют хеш-таблицу, в которой находится уникальный ключ и указатель на конкретный объект данных. Существует понятие блока (bucket) — логической группы ключей, которые не группируют данные физически. В разных блоках могут быть идентичные ключи.

Документоориентированная база данных

Данные, представленные парами ключ-значение, сжимаются как хранилище документов схожим с хранилищем «ключ-значение» образом, с той лишь разницей, что хранимые значения (документы) имеют определённую структуру и кодировку данных. XML, JSON и BSON — некоторые из стандартных распространённых кодировок.

Тот факт, что такие базы данных работают без схемы, делает простой задачей добавление полей в JSON-документы без необходимости сначала заявлять об изменениях.

Couchbase и MongoDB — самые популярные документоориентированные СУБД.

Колоночная база данных

В колоночных NoSQL базах данных данные хранятся в ячейках, сгруппированных в колонки, а не в строки данных. Колонки логически группируются в колоночные семейства. Колоночные семейства могут состоять из практически неограниченного количества колонок, которые могут создаваться во время работы программы или во время определения схемы. Чтение и запись происходит с использованием колонок, а не строк.

В сравнении с хранением данных в строках, как в большинстве реляционных баз данных, преимущества хранения в колонках заключаются в быстром поиске/доступе и агрегации данных.

Графовая база данных

В графовой базе данных вы не найдёте строгого формата SQL или представления таблиц и колонок, вместо этого используется гибкое графическое представление, которое идеально подходит для решения проблем масштабируемости.

Графовые структуры используются вместе с рёбрами, узлами и свойствами, что обеспечивает безиндексную смежность. При использовании графового хранилища данные могут быть легко преобразованы из одной модели в другую.

Интеграция SQL и NoSQL

Объектно-реляционное отображение (Object-Relational Mapping, ORM) давно стало стандартной практикой при разработке объектно-ориентированных приложений, использующих SQL.

Являясь промежуточным слоем для преодоления несоответствия между SQL и объектной моделью приложения, ORM обеспечивает управление жизненным циклом объектов, абстракцию от деталей хранения, выполнение запросов в терминах объектной модели, генерацию схемы базы данных и т. д.

Тем не менее классические ORM-библиотеки, несмотря на развитый инструментарий и богатую функциональность, мало подходят для высоконагруженных систем. Наличие высокого уровня абстракции мешает применять низкоуровневые оптимизации и требует слишком больших накладных расходов (например, преобразование «объектного» диалекта SQL в реальный запрос, который будет выполнен на целевой СУБД).

Реализация интеграции SQL и NoSQL на уровне объектного отображения мотивирована простым фактом — в большинстве случаев при работе с любой СУБД требуется создание объектов на основе запрашиваемой информации, а также сохранение этих объектов в базе данных. Можно было бы предложить использование нескольких библиотек для реализации отображения внутри одного приложения, однако тогда снова появляется проблема различных интерфейсов и возникает необходимость вручную связывать между собой объекты, отображаемые различными библиотеками; кроме того, остаются проблемы с оптимизацией.

Классы в ООП, как и таблицы в SQL, обычно имеют фиксированный набор атрибутов. Для того чтобы иметь возможность выполнять отображение неструктурированных или квазиструктурированных данных, требуется поддержка динамических атрибутов, а также вложенных объектов и массивов.

Объекты, таким образом, должны представлять собой скорее документы, наделенные поведением, нежели объекты в привычном для ООП смысле. Конечно, в этом случае работать с объектной моделью становится несколько сложнее, но зато она покрывает как фиксированные, так и динамические схемы.

Кроме того, при этом естественнее реализуются проекции объектов (частичные объекты), у которых из базы данных были загружены не все поля. Это бывает полезно, когда требуется работа только с небольшим

подмножеством атрибутов и загрузка всех полей избыточна. Более того, поддержка динамических атрибутов в ряде случаев может заменить наследование и упростить модель [7].

Что же касается ограничений целостности, то целесообразно иметь средства для их проверки на уровне приложения, так как большинство NoSQL-систем не обладает подобной функциональностью, а СУБД SQL не всегда позволяют выразить специфичные для приложения условия.

Предложенный вариант организации отображения является, по сути, сервисной архитектурой в миниатюре — если рассматривать модуль отображения как сервис, обладающий стандартизованным интерфейсом и скрывающий низкоуровневые детали работы с данными.

Этот подход дает возможность упростить и ускорить разработку приложений, использующих несколько систем управления данными, и в то же время сохранить гибкость, необходимую для дальнейшей оптимизации под конкретные задачи.

### 3.5. Контрольные задания

#### 3.5.1. Вопросы для самопроверки

1. Масштабирование баз данных.
2. Шардинг.
3. NoSQL базы данных.
4. Принцип глобального ID.
5. Методы генерации глобального ID.
6. Банк данных.
7. Требования к банку данных.
8. Типы NoSQL хранилищ.
9. Типы баз данных.

## 4. СИСТЕМА ОБМЕНА СООБЩЕНИЯМИ APACHE KAFKA

Каждое предприятие работает на основе данных. Мы принимаем информацию, анализируем ее, манипулируем ею и создаём большой выходной поток данных.

Каждый байт данных имеет историю, которую нужно рассказать, что-то важное, что сообщит следующую вещь, которую нужно сделать. Чтобы узнать, что это такое, нам нужно получить данные, откуда они были созданы, и где их можно проанализировать. Мы видим это каждый день на таких сайтах, как Yandex, где наши клики по интересующим нас товарам превращаются в рекомендации, которые нам показываются немного позже.

Чем быстрее мы сможем это сделать, тем более гибкими и отзывчивыми могут быть наши организации. Чем меньше усилий мы тратим на перемещение данных, тем больше мы можем сосредоточиться на основной деятельности. Вот почему конвейер является критически важным компонентом предприятия, управляемого данными. То, как мы перемещаем данные, становится почти таким же важным, как и сами данные.

Прежде чем обсуждать специфику Apache Kafka, для нас важно понять концепцию публикации/подписки и почему это важно. Публикация/подписка - это шаблон, который характеризуется отправителем (издателем) фрагмента данных (сообщения), который не направляет его конкретно получателю. Вместо этого издатель как-то классифицирует сообщение, и этот получатель (подписчик) подписывается на получение определенных классов сообщений. Системы Pub/Sub часто имеют посредника, центральную точку публикации сообщений, чтобы облегчить этот процесс.

Многие варианты использования для публикации или подписки начинаются одинаково: с простой очереди сообщений или межпроцессного канала связи. Например, вы создаете приложение, которое должно отправлять информацию мониторинга куда-то, поэтому вы пишете из вашего приложения в приложение, которое отображает ваши метрики на приборной панели, и помещаете метрики.

Это простое решение простой проблемы, которая работает, когда вы начинаете с простейшего мониторинга. Вскоре вы решаете, что хотите анализировать свои показатели в течение более длительного периода времени, и это плохо работает на панели инструментов. Вы запускаете новый сервис, который может получать метрики, хранить их и анализировать их. Для поддержки этого вы изменяете свое приложение для записи метрик в обе системы. К настоящему времени у вас есть еще три приложения, которые генерируют метрики, и все они имеют одинаковые соединения с этими двумя службами. Ваш коллега считает, что было бы неплохо также провести активный опрос служб для оповещения, поэтому вы добавляете сервер в каждое приложение для предоставления метрик по запросу. Через некоторое время у вас появляется больше приложений, которые используют эти серверы для получения отдельных метрик и использования их для различных целей.

Технический опыт, накопленный при решении такой задачи, позволяет понять более очевидный выход для решения этой проблемы. Вы настраиваете одно приложение, которое получает метрики от всех приложений, и предоставляете сервер для запроса этих метрик для любой системы, которая в них нуждается. Это уменьшает сложность архитектуры. В этом и состоит система обмена сообщениями с подпиской.

В то же время, когда вы решали эту проблему, один из ваших коллег выполнял аналогичную работу с логами. Другой работал над отслеживанием поведения пользователей на веб-сайте пользовательского интерфейса и предоставлением этой информации разработчикам, занимающимся машинным

обучением, а также над созданием некоторых отчетов для управления. Вы все пошли по сходному пути построения систем, которые отделяют издателей информации от подписчиков этой информации.

Это, конечно, намного лучше, чем использование соединений от одной точки до другой, но есть много дублирования. Ваша компания поддерживает несколько систем для организации очередей данных, каждая из которых имеет свои собственные ошибки и ограничения. Вы также знаете, что скоро будет увеличение количества подписчиков для обмена сообщениями. Вы хотели бы иметь единую централизованную систему, которая позволяет публиковать общие типы данных, которые будут расти по мере роста вашего бизнеса.

Apache Kafka - это система обмена сообщениями «публикация/подписка», разработанная для решения этой проблемы. Её часто называют «распределенным журналом фиксации» или в последнее время «распределенной потоковой платформой». Журнал фиксации файловой системы или базы данных предназначен для обеспечения надежной записи всех транзакций, чтобы их можно было воспроизвести для последовательного построения состояния системы. Точно так же данные в Kafka хранятся в порядке и могут быть детерминированно прочитаны. Кроме того, данные могут быть распределены внутри системы, чтобы обеспечить дополнительную защиту от сбоев, а также значительные возможности для повышения производительности.

Единица данных в Kafka называется сообщением. Если вы обращаетесь к Kafka из базы данных, вы можете думать об этом как о строке или записи. Сообщение - это просто массив байтов в отношении Kafka, поэтому содержащиеся в нем данные не имеют определенного формата или значения для Kafka. Сообщение может иметь дополнительный бит метаданных, который называется ключом. Ключ также является байтовым массивом и, как и в сообщении, не имеет особого значения для Kafka. Ключи используются, когда сообщения должны быть записаны в разделы более контролируемым образом. Самая простая такая схема - создать непротиворечивый хэш ключа, а затем выбрать номер раздела для этого сообщения, взяв результат хэширования по модулю, общее количество разделов в теме. Это гарантирует, что сообщения с одним и тем же ключом всегда записываются в один и тот же раздел.

Для эффективности сообщения записываются в Kafka партиями. Партия - это просто набор сообщений, все из которых создаются для одной и той же темы и раздела. Отдельная передача туда и обратно по сети для каждого сообщения может привести к чрезмерным издержкам, а сбор сообщений в пакет уменьшает это. Конечно, это компромисс между задержкой и пропускной способностью: чем больше пакеты, тем больше сообщений можно обработать за единицу времени, но тем больше времени требуется для распространения отдельного сообщения. Пакеты также обычно сжимаются, обеспечивая более эффективную передачу и хранение данных за счет снижения вычислительной мощности.

Хотя сообщения являются просто байтовыми массивами для самой Kafka, рекомендуется, чтобы дополнительная структура или схема была наложена на содержимое сообщения, чтобы его можно было легко понять. Существует множество параметров для схемы сообщений, в зависимости от индивидуальных потребностей вашего приложения. Упрощенные системы, такие как Javascript Object Notation (JSON) и Extensible Markup Language (XML), просты в использовании и удобочитаемы. Однако им не хватает таких функций, как надежная обработка типов и совместимость между версиями схемы. Многие разработчики Kafka предпочитают использовать единый формат данных, который важен в Kafka, так как он позволяет разделить записи и чтения сообщений. Когда эти задачи тесно связаны, приложения, которые подписываются на сообщения, должны быть обновлены для обработки нового формата данных параллельно со старым форматом. Только тогда приложения, публикующие сообщения, могут быть обновлены для использования нового формата.

Сообщения в Kafka делятся на темы. Наиболее близкими аналогами для темы являются таблица базы данных или папка в файловой системе. Темы дополнительно разбиты на несколько разделов. Возвращаясь к описанию «журнала коммитов», раздел - это единственный журнал. Сообщения записываются в него только в виде дополнений и читаются по порядку от начала до конца. Обратите внимание, что поскольку тема обычно состоит из нескольких разделов, нет гарантии упорядочения времени сообщения по всей теме, только внутри одного раздела. Разделы - это также способ, которым Kafka обеспечивает избыточность и масштабируемость. Каждый раздел может быть размещен на отдельном сервере, что означает, что одна тема может быть масштабирована по горизонтали между несколькими серверами, чтобы обеспечить производительность, намного превышающую возможности одного сервера.

Термин “поток” часто используется при обсуждении данных в таких системах, как Kafka. Чаще всего поток считается отдельной темой данных, независимо от количества разделов. Это представляет собой единый поток данных, перемещаемых от производителей к потребителям. Этот способ обращения к сообщениям наиболее распространен при обсуждении потоковой обработки, то есть когда фреймворки, в том числе Kafka Streams, Apache Samza и Storm, работают с сообщениями в режиме реального времени. Этот метод работы можно сравнить с тем, как работают автономные фреймворки, например, Hadoop, предназначенные для работы с объемными данными в более позднее время.

Клиенты Kafka являются пользователями системы, и существует два основных типа: производители и потребители. Существуют также расширенные клиентские API-интерфейсы - Kafka Connect API для интеграции данных и Kafka Streams для потоковой обработки. Опытные клиенты используют производителей и потребителей в качестве строительных блоков и

обеспечивают функциональность более высокого уровня. Производители создают новые сообщения. В других системах публикации/подписки они могут называться издателями или авторами. Как правило, сообщение будет подготовлено к определенной теме. По умолчанию производителю все равно, в какой раздел записано конкретное сообщение, и он будет равномерно распределять сообщения по всем разделам темы. В некоторых случаях производитель будет направлять сообщения в определенные разделы. Обычно это делается с помощью ключа сообщения и разделителя, который сгенерирует хэш ключа и сопоставит его с конкретным разделом. Это гарантирует, что все сообщения, созданные с данным ключом, будут записаны в один и тот же раздел. Производитель может также использовать пользовательский разделитель, который следует другим бизнес-правилам для отображения сообщений на разделы.

Потребители читают сообщения. В других системах публикации/подписки эти клиенты могут называться подписчиками или читателями. Потребитель подписывается на одну или несколько тем и читает сообщения в том порядке, в котором они были созданы. Потребитель отслеживает, какие сообщения он уже использовал, отслеживая смещение сообщений. Смещение - это еще один бит метаданных - целочисленное значение, которое постоянно увеличивается, - которое Kafka добавляет к каждому сообщению при его создании. Каждое сообщение в данном разделе имеет уникальное смещение. Сохраняя смещение последнего использованного сообщения для каждого раздела либо в Zookeeper, либо в самом Kafka, потребитель может останавливаться и перезапускаться, не теряя своего смещения.

Потребители работают как часть группы потребителей, которая является одним или несколькими потребителями, которые работают вместе, чтобы использовать тему. Группа гарантирует, что каждый раздел потребляется только одним участником. Два потребителя работают с одного раздела каждый, в то время как третий потребитель работает с двух разделов. Сопоставление потребителя с разделом часто называется владением разделом потребителем.

Таким образом, потребители могут горизонтально масштабировать, чтобы использовать темы с большим количеством сообщений. Кроме того, в случае отказа одного потребителя оставшиеся члены группы будут перебалансировать расходуемые разделы, чтобы заменить отсутствующего участника.

### 4.1. Брокеры и Кластеры

Один сервер Kafka называется брокером. Посредник получает сообщения от производителей, присваивает им смещения и фиксирует сообщения в хранилище на диске. Он также обслуживает потребителей, отвечая на запросы выборки для разделов и отвечая сообщениями, которые были переданы на диск.

В зависимости от конкретного оборудования и его характеристик производительности, один брокер может легко обрабатывать тысячи разделов и миллионы сообщений в секунду.

Брокеры Kafka предназначены для работы в составе кластера. В кластере брокеров один брокер также будет выполнять функции контроллера кластера (выбирается автоматически из действующих членов кластера). Контроллер отвечает за административные операции, включая назначение разделов брокерам и мониторинг их сбоев. Раздел принадлежит одному посреднику в кластере, и этот посредник называется лидером раздела. Раздел может быть назначен нескольким посредникам, что приведет к репликации раздела. Это обеспечивает избыточность сообщений в разделе, так что другой брокер может взять на себя лидерство в случае сбоя брокера. Однако все потребители и производители, работающие в этом разделе, должны подключаться к лидеру.

Ключевой особенностью Apache Kafka является функция хранения, которая обеспечивает долговременное хранение сообщений в течение некоторого периода времени. Брокеры Kafka настраиваются с сохранением по умолчанию для тем, сохраняя сообщения в течение некоторого периода времени (например, 7 дней) или до тех пор, пока тема не достигнет определенного размера в байтах (например, 1 ГБ). Как только эти пределы достигнуты, сообщения считаются просроченными и удаляются, так что конфигурация хранения представляет собой минимальный объем данных, доступных в любое время. Отдельные темы также могут быть сконфигурированы с их собственными настройками хранения, так что сообщения хранятся столько времени, сколько они полезны. Например, тема отслеживания может сохраняться в течение нескольких дней, тогда как метрики приложений могут сохраняться только в течение нескольких часов. Темы также могут быть сконфигурированы как сжатые журналы, это означает, что Kafka сохранит только последнее сообщение, созданное с определенным ключом. Это может быть полезно для данных типа changelog, где интересно только последнее обновление.

## 4.2. Преимущества системы обмена Apache Kafka

Существует много вариантов систем обмена сообщениями с публикацией/подпиской, так что же делает Apache Kafka хорошим выбором?

- Несколько производителей.
- Несколько потребителей.
- Хранение данных на дисках.
- Масштабируемость.
- Высокая производительность
- Экосистема данных.

Рассмотрим каждый из них детальнее:

Несколько производителей - Kafka может беспрепятственно работать с несколькими производителями, независимо от того, используют ли эти клиенты много тем или одну и ту же тему. Это делает систему идеальной для агрегирования данных из многих внешних систем и обеспечения ее согласованности. Например, сайт, который обслуживает пользователей через несколько микросервисов, может иметь одну тему для просмотров страниц, которую все службы могут записывать в общий формат. Потребительские приложения могут затем получать один поток просмотров страниц для всех приложений на сайте без необходимости координировать потребление по нескольким темам, по одному для каждого приложения.

Несколько потребителей - в дополнение к нескольким производителям, Kafka предназначен для нескольких потребителей, чтобы читать любой поток сообщений, не мешая друг другу. Это отличает ее от многих систем массового обслуживания, где, когда сообщение используется одним клиентом, оно становится недоступным для любого другого. Несколько потребителей Kafka могут работать в составе группы и совместно использовать поток, гарантируя, что вся группа обрабатывает данное сообщение только один раз.

Хранение данных на дисках - не только Kafka может обрабатывать несколько потребителей, но и длительное хранение сообщений означает, что потребители не всегда должны работать в режиме реального времени. Сообщения сохраняются на диске и будут храниться с настраиваемыми правилами хранения. Эти параметры могут быть выбраны для каждой темы, что позволяет различным потокам сообщений иметь различную степень хранения в зависимости от потребностей потребителя. Долговременное хранение означает что, если потребитель отстает либо из-за медленной обработки, либо из-за увеличения трафика, нет никакой опасности потери данных. Это также означает, что обслуживание может выполняться для потребителей, когда приложения на короткое время отключаются от сети, не беспокоясь о том, что сообщения копируются на производителя или теряются. Потребителей можно остановить, а сообщения будут сохранены в Kafka. Это позволяет им перезапускать и обрабатывать сообщения, с которых они остановились, без потери данных.

Масштабируемость - гибкая масштабируемость Kafka позволяет легко обрабатывать любой объем данных. Пользователи могут начать с одного брокера в качестве доказательства концепции, расширить его до небольшого кластера разработки из трех брокеров и перейти к производству с большим кластером из десятков или даже сотен брокеров, который растет со временем по мере увеличения объема данных. Расширения могут выполняться, когда кластер подключен к сети, что не влияет на доступность системы в целом. Это также означает, что кластер из нескольких брокеров может обработать сбой отдельного брокера и продолжить обслуживание клиентов. Кластеры, которым необходимо допускать больше одновременных сбоев, могут быть настроены с более высокими коэффициентами репликации.

Высокая производительность - все эти функции объединяются, чтобы сделать Apache Kafka средством публикации/подписки. Производители, потребители и все брокеры могут быть легко масштабированы для обработки очень больших потоков сообщений. Это может быть реализовано, при этом обеспечивая задержку второго сообщения от первого.

Экосистема данных - многие приложения работают в среде, которую разработчик создает для обработки данных. Он определил входные данные в форме приложений, которые создают данные или иным образом вводят их в систему. Он определил выходные данные в форме метрик, отчетов и других продуктов данных. Он создает циклы, в которых некоторые компоненты считывают данные из системы, преобразуют их с использованием данных из других источников, а затем вводят их обратно в инфраструктуру данных для использования в другом месте. Это сделано для многочисленных типов данных, каждый из которых имеет уникальные качества содержимого, размера и использования. Apache Kafka передает сообщения между различными участниками инфраструктуры, обеспечивая согласованный интерфейс для всех клиентов. При соединении с системой для предоставления схем сообщений производителям и потребителям больше не требуется тесная связь или прямые соединения любого рода. Компоненты могут добавляться и удаляться по мере создания и ликвидации бизнес-кейсов, и производителям не нужно беспокоиться о том, кто использует данные или количество приложений-потребителей.

### 4.3. Зачем нам нужна Apache Kafka?

Отслеживание активности - первоначальный вариант использования Kafka, разработанный в LinkedIn, - это отслеживание активности пользователя. Пользователи веб-сайта взаимодействуют с внешними приложениями, которые генерируют сообщения о действиях, предпринимаемых пользователем. Это может быть пассивная информация, такая как просмотры страниц и отслеживание кликов, или более сложные действия, такие как информация, которую пользователь добавляет в свой профиль. Сообщения публикуются в одной или нескольких темах, которые затем используются приложениями на сервере. Этими приложениями могут быть создание отчетов, подача систем машинного обучения, обновление результатов поиска или выполнение других операций, которые необходимы для обеспечения богатого пользовательского опыта.

Обмен сообщениями - Kafka также используется для обмена сообщениями, где приложения должны отправлять уведомления (например, электронные письма) пользователям. Эти приложения могут создавать сообщения, не заботясь о форматировании или о том, как сообщения будут отправляться. Одно приложение может затем прочитать все сообщения, которые будут отправлены, и обрабатывать их последовательно, включая:

- форматирование сообщений (также называемое декорированием) с использованием общего внешнего вида;
- сбор нескольких сообщений в одно уведомление для отправки;
- применение пользовательских настроек к тому, как они хотят получать сообщения.

Использование одного приложения для этого исключает необходимость дублирования функций в нескольких приложениях, а также позволяет выполнять такие операции, как агрегирование, которые в противном случае были бы невозможны.

Метрики и логи - Kafka также идеально подходит для сбора метрик и журналов приложений и системы. Это тот случай использования, в котором существует возможность иметь несколько приложений, создающих один и тот же тип сообщения. Приложения регулярно публикуют метрики в теме Kafka, и эти метрики могут использоваться системами для мониторинга и оповещения. Они также могут быть использованы в автономной системе, такой как Hadoop, для выполнения более долгосрочного анализа, такого как прогнозы роста. Сообщения журнала могут публиковаться таким же образом и могут направляться в специальные системы поиска журналов, такие как Elastisearch или приложения для анализа безопасности. Еще одним дополнительным преимуществом Kafka является то, что, когда необходимо изменить целевую систему (например, пришло время обновить систему хранения журналов), нет необходимости изменять внешние приложения или средства агрегирования.

Журнал фиксации – поскольку Kafka основан на концепции журнала фиксации, изменения базы данных могут публиковаться в Kafka, и приложения могут легко отслеживать этот поток, чтобы получать живые обновления по мере их поступления. Этот поток журнала изменений также можно использовать для репликации обновлений базы данных в удаленную систему или для консолидации изменений из нескольких приложений в одном представлении базы данных. Долговременное хранение полезно здесь для предоставления буфера для журнала изменений, это означает, что он может быть воспроизведен в случае сбоя приложений-потребителей. В качестве альтернативы темы, сжатые журналом, можно использовать для обеспечения более длительного хранения, сохраняя только одно изменение на ключ.

Потоковая обработка - еще одной областью, которая предоставляет многочисленные типы приложений, является потоковая обработка. Хотя почти все использование Kafka можно рассматривать как потоковую обработку, этот термин обычно используется для обозначения приложений, которые предоставляют аналогичные функциональные возможности для сопоставления/сокращения обработки в Hadoop. Hadoop обычно опирается на агрегирование данных в течение длительного периода времени, будь то часы или дни. Потоковая обработка работает с данными в режиме реального времени так же быстро, как и сообщения. Структуры потоков позволяют пользователям создавать небольшие приложения для работы с сообщениями Kafka,

выполнения таких задач, как подсчет метрик, разбиение сообщений для эффективной обработки другими приложениями или преобразование сообщений с использованием данных из нескольких источников.

### 4.4. Настройка и запуск Apache Kafka

Apache Kafka - это Java-приложение, которое может работать во многих операционных системах: Windows, MacOS, Linux и другие. Шаги установки будут направлены на настройку и использование Kafka в среде Linux, так как это самая распространенная ОС, на которой он установлен. Это также рекомендуемая ОС для развертывания Kafka для общего пользования.

Шаг 1. Установка Java:

Перед установкой Zookeeper или Kafka вам потребуется настройка и функционирование среды Java. Это должна быть версия Java 8, и это может быть версия, предоставленная вашей ОС, или версия, прямо загруженная с java.com. Хотя Zookeeper и Kafka будут работать с выпуском Java во время выполнения, при разработке инструментов и приложений может быть удобнее иметь полный Java Development Kit (JDK). 17 шагов установки предполагают, что вы установили обновление 52 JDK версии 8 в /usr/java/jdk1.8.0_52.

Перейдите в каталог, в который необходимо выполнить установку. Введите:

```
cd directory_path_name
```

Например, для установки ПО в каталог /usr/java/directory введите:

```
cd /usr/java/
```

Переместите архивированный двоичный файл .tar.gz в текущий каталог.

Распакуйте tar-архив и установите Java

```
tar zxvf jre-8u52-linux-x64.tar.gz
```

Файлы Java устанавливаются в каталог jre1.8.0_52, созданный в текущем каталоге. В данном примере это каталог /usr/java/jre1.8.0_52. После завершения установки появляется сообщение Done (Готово).

Удалите файл .tar.gz, если необходимо сэкономить пространство на диске.

Шаг 2. Установка Zookeeper

Apache Kafka использует Zookeeper для хранения метаданных о кластере Kafka, а также сведений о клиентских клиентах. Хотя можно запустить сервер ZooKeeper с помощью сценариев, содержащихся в дистрибутиве Kafka, установить полную версию Zookeeper из дистрибутива тривиально.

Kafka была тщательно протестирована со стабильной версией 3.4.6 Zookeeper, которую можно загрузить с сайта apache.org по адресу http://bit.ly/2sDWSgJ.

В следующем примере Zookeeper устанавливается с базовой конфигурацией в /usr/local/zookeeper, хранящий свои данные в /var/lib/zookeeper:

```
# tar -zxf zookeeper-3.4.6.tar.gz
# mv zookeeper-3.4.6 /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cat > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk1.8.0_52
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```

Теперь вы можете проверить правильность работы Zookeeper в автономном режиме, подключившись к клиентскому порту и отправив четырехбуквенную команду srvr:

```
# telnet localhost 2181
Trying ::1...
Connected to localhost.
Escape character is '^]'.
srvr
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x0
Mode: standalone
Node count: 4
Connection closed by foreign host.
#
```

Чтобы настроить серверы Zookeeper в кластер, они должны иметь общую конфигурацию, в которой перечислены все серверы, и каждому серверу нужен файл myid в каталоге данных, в котором указан идентификационный номер сервера. Если имена узлов серверов в ансамбле - zoo1.example.com, zoo2.example.com и zoo3.example.com, файл конфигурации может выглядеть следующим образом:

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=zoo1.example.com:2888:3888
server.2=zoo2.example.com:2888:3888
server.3=zoo3.example.com:2888:3888
```

В этой конфигурации initLimit - это количество времени, которое позволяет последователям соединяться с лидером. Значение syncLimit ограничивает количество несинхронизированных подписчиков с лидером. Оба значения являются числом единиц измерения tickTime, что делает initLimit 20 * 2000 мс или 40 секунд. Конфигурация также перечисляет каждый сервер в кластере. Серверы указываются в формате server.X = имя хоста: peerPort: leaderPort со следующими параметрами:

X – идентификационный номер сервера. Это должно быть целое число, но оно не должно быть нулевым или последовательным;

hostname - имя хоста или IP-адрес сервера;

peerPort - TCP-порт, через который серверы в кластере связываются друг с другом;

leaderPort - порт TCP, по которому осуществляется выбор лидера. Клиенты должны иметь возможность подключаться к кластеру только через clientPort, но члены кластера должны иметь возможность общаться друг с другом через все три порта.

В дополнение к общему файлу конфигурации каждый сервер должен иметь файл в каталоге данных Dir с именем myid. Этот файл должен содержать идентификационный номер сервера, который должен соответствовать файлу конфигурации. После завершения этих шагов серверы запустятся и будут общаться друг с другом в кластере.

Шаг 3. После настройки Java и Zookeeper вы можете установить Apache Kafka. Текущий выпуск Kafka можно скачать по адресу http://kafka.apache.org/downloads.html. В следующем примере Kafka устанавливается в /usr/local/kafka, настроенный для использования сервера Zookeeper, запущенного ранее, и для хранения сегментов журнала сообщений, хранящихся в /tmp/kafka-logs:

```
# tar -zxf kafka_2.11-0.9.0.1.tgz
# mv kafka_2.11-0.9.0.1 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk1.8.0_52
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

После запуска брокера Kafka мы можем убедиться, что он работает, выполнив несколько простых операций с кластером, создав несколько сообщений и использовав те же сообщения.

Создание и проверка топика:

```
# /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic myTopic
Created topic "myTopic".
# /usr/local/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic myTopic Topic:myTopic PartitionCount:1 ReplicationFactor:1 Configs:
 Topic: myTopic Partition: 0 Leader: 0 Replicas: 0 Isr: 0
#
```

Создать сообщения в тестовой теме:

```
# /usr/local/kafka/bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic myTopic
MyTopic Message 1
MyTopic Message 2
^D
#
```

Прочесть сообщения из тестовой темы:

```
# /usr/local/kafka/bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic myTopic --from-beginning
MyTopic Message 1
MyTopic Message 2
^C
Consumed 2 messages
#
```

Конфигурация брокера:

Пример конфигурации, поставляемой с дистрибутивом Kafka, достаточен для запуска автономного сервера в качестве подтверждения концепции, но его будет недостаточно для большинства установок. Существует множество опций конфигурации для Kafka, которые контролируют все аспекты настройки. Многие параметры могут быть оставлены с настройками по умолчанию, так как они касаются аспектов настройки брокера Kafka, которые не будут применимы, пока у вас не будет конкретного варианта использования для работы и конкретного варианта использования, который требует настройки этих параметров.

Шаг 4. Создание приложения для опубликования сообщений

Когда Kafka установлена и готова к использованию, мы можем приступить к написанию приложений, которые будут генерировать сообщения и помещать их в Kafka. В качестве примера мы будем писать приложение на языке Java.

Клиент-производитель Kafka состоит из следующих API:

• класс KafkaProducer предоставляет метод send для асинхронной отправки сообщений в тему. Подпись send () выглядит следующим образом:

```
producer.send(new ProducerRecord<byte[],byte[]>(topic,
partition, key1, value1) , callback);
```

• ProducerRecord - производитель управляет буфером записей, ожидающих отправки;

• обратный вызов - предоставленный пользователем обратный вызов для выполнения, когда запись была подтверждена сервером (ноль означает отсутствие обратного вызова);

• класс KafkaProducer предоставляет метод сброса, обеспечивающий фактическое завершение всех ранее отправленных сообщений. Синтаксис метода очистки следующий:

```
public void flush();
```

• класс KafkaProducer предоставляет метод partitionFor, который помогает в получении метаданных раздела для данной темы. Это может быть использовано для пользовательского разбиения. Суть этого метода заключается в следующем:

```
public Map metrics().
```

Возвращает карту внутренних метрик, поддерживаемых производителем;

• public void close () - класс KafkaProducer предоставляет блоки методов close, пока все ранее отправленные запросы не будут выполнены.

## 4.5. Kafka API

1. API продюсера

Центральной частью API производителя является класс Producer. Класс Producer предоставляет возможность подключить брокер Kafka в своем конструкторе следующими методами.

Класс продюсера

Класс продюсера предоставляет метод send для отправки сообщений в одну или несколько тем с использованием следующих подписей.

```
public void send(KeyedMessaget<k,v> message)
- sends the data to a single topic,par-titioned by key using either sync or async producer.
public void send(List<KeyedMessage<k,v>>messages)
- sends data to multiple topics.
Properties prop = new Properties();
prop.put(producer.type,”async”)
ProducerConfig config = new ProducerConfig(prop);
```

Существует два типа производителей - Sync и Async.

Класс Producer предоставляет метод close для закрытия соединений пула производителей со всеми брокерами Kafka.

Настройки конфигурации

Основные параметры конфигурации API производителя приведены в табл. 4.1.

Таблица 4.1

Основные параметры конфигурации API

| | Настройки конфигурации | Описание конфигурации |
|---|---|---|
| | client.id | определяет приложение производителя |
| | producer.type | Синхронно или асинхронно |
| | acks | Критерии по запросам производителя |
| | retries | Если запрос производителя не удался, автоматически повторите попытку столько раз |
| | bootstrap.servers | Список брокеров |
| | linger.ms | Регулирует количество запросов |
| | serialize.key | Ключ для интерфейса сериализатора. |
| | serialize.value | Значение для интерфейса сериализатора. |
| | batch.size | Размер батча |
| 0 | buff.mem | Размер буфера |

ProducerRecord API

ProducerRecord - это пара ключ/значение, которая отправляется в кластер Kafka. Конструктор класса ProducerRecord для создания записи с парами разделов, ключей и значений с использованием следующей подписи.

```
public ProducerRecord (string topic, int partition, k key, v value)
public ProducerRecord (string topic, k key, v value)
public ProducerRecord (string topic, v value)
```

где

- topic - определенное пользователем название темы, в которое будет добавлена запись;
- partition - количество разделов;
- key - ключ, который будет включен в запись;
- value - запись содержимого.

Пример кода, посылающего сообщения в Kafka:

```
import java.util.Properties;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
```

```
//Создаем класс с именем “SimpleExampleProducer”
public class SimpleExampleProducer {

  public static void main(String[] args) throws Exception{
    // проверяем, указан ли топик
    if(args.length == 0){
      System.out.println("Введи имя топика”);
      return;
    }
    //Считываем имя топика в переменную
    String nameOfTopic = args[0].toString();
    // создаем объект Properties для настройки параметров
    Properties kafkaProperties = new Properties();
    // указываем Kafka сервер
    kafkaProperties.put("bootstrap.servers", “localhost:9092");
    //устанавливаем политику подтверждений для всех
    kafkaProperties.put("acks", “all");
    // указываем количество попыток досылки сообщения
    kafkaProperties.put("retries", 0);
    //Устанавливаем размер батча
    kafkaProperties.put("batch.size", 16383);
    // устанавливаем количество запросов меньше нуля
    kafkaProperties.put("linger.ms", 1);
    // устанавливаем размер буфера памяти и параметры сериализации
    kafkaProperties.put("buff.mem", 33553432);
    kafkaProperties.put("serialize.key",
      "org.apache.kafka.common.serializa-tion.StringSerializer");
    kafkaProperties.put("serialize.value",
      "org.apache.kafka.common.serializa-tion.StringSerializer");
   // применяем параметры и создаем продьюсера
    Producer<String, String> kafkaProducer = new KafkaProducer
      <String, String>(kafkaProperties);
    // посылаем несколько сообщений в Kafka
    for(int i = 0; i < 10; i++)
      kafkaProducer.send(new ProducerRecord<String, String>( nameOfTopic,
        Integer.toString(i), Integer.toString(i)));
          System.out.println(“Сообщение успешно отправлено”);
          kafkaProducer.close();
  }
}
```

Соберем и запустим приложение.
Если все настроено верно, то мы увидим в консоли следующее:

Сообщение успешно отправлено

To check the above output open new terminal and type Consumer CLI command to receive messages.

>> bin/kafka-console-consumer.sh --zookeeper localhost:2181 —topic <topic-name> —from-beginning

1
2
3
4
5
6
7
8
9
10

Шаг 5. Создание простого клиента, вычитывающего сообщения из Kafka.

API KafkaConsumer используется для приема сообщений из кластера Kafka. Конструктор класса KafkaConsumer определен ниже.

public KafkaConsumer(java.util.Map<java.lang.String,java.lang.Object> configs)

где configs – объект класса Properties, в котором указаны параметры, аналогичные приведенным выше для публикующего приложения.

Класс KafkaConsumer имеет следующие важные методы, которые перечислены в табл. 4.2.

Таблица 4.2

Основные методы класса KafkaConsumer

| № | Название метода | Описание |
|---|---|---|
| 1 | assignment() | Получить лист партишенов, закрепленных за потребителем |
| 2 | subscription() | Подписаться на данный список тем, чтобы получить динамически подписанные разделы |
| 3 | subscribe(java.util.List<java.lang.String> topics, ConsumerRe-balanceListener listener) | |
| 4 | unsubscribe() | Отписаться от тем из данного списка разделов |

Окончание табл. 4.2

| | | |
|---|---|---|
| 5 | subscribe(java.util.List<java.lang.String> topics) | Подпишитесь на данный список тем, чтобы получить динамически подписанные разделы. Если данный список тем пуст, он обрабатывается так же, как и отмена подписки |
| 6 | subscribe(java.util.regex.Pattern pattern, ConsumerRebalanceLis-tener listener) | Шаблон аргумента относится к шаблону подписки в формате регулярного выражения, а аргумент слушателя получает уведомления из шаблона подписки |
| 7 | assign(java.util.List<TopicParti-tion> partitions) | Ручное назначение списка разделов |
| 8 | poll() | Получить данные для тем или разделов, указанных с помощью одного из API подписки или назначения |
| 9 | commitSync() | Смещение коммитов, возвращаемое в последнем опросе для всех подписанных списков тем и разделов |
| 10 | seek(TopicPartition partition, long offset) | Получить текущее значение смещения, которое потребитель будет использовать в следующем методе poll() |
| 11 | resume() | Возобновить приостановленные разделы |
| 12 | wakeup() | Восстановить потребителя |

ConsumerRecord API

API ConsumerRecord используется для получения записей из кластера Kafka.

Класс ConsumerRecord используется для создания записи потребителя с определенным именем темы, количеством разделов и парами <ключ, значение>.

| |
|---|
| public ConsumerRecord(string topic,int partition, long offset,K key, V value) |

где:

• topic - определенное пользователем название темы, в которое будет добавлена запись;

• partition - количество разделов;

• key - ключ, который будет включен в запись;

• value – содержимое записи.

API APIConsumerRecords действует как контейнер для ConsumerRecord. Этот API используется для хранения списка ConsumerRecord для каждого раздела для определенной темы. Его конструктор определен ниже.

```
public ConsumerRecords(java.util.Map<TopicPartition,java.util.List
<ConsumerRecord>K,V>>> records)
```

где:

• TopicPartition – карта с топиками и партишенами;

• Records – лист записей.

В классе ConsumerRecords определены методы. Они перечислены в табл. 4.3.

Таблица 4.3

Методы ConsumerRecords

| | Название метода | Описание |
|---|---|---|
| | count() | Количество записей по всем темам. |
| | partitions() | Набор разделов с данными в этом наборе записей |
| | iterator() | Итератор позволяет циклически проходить через коллекцию, получать или перемещать элементы. |
| | records() | Получить список записей для данного раздела |

Ниже представлен пример кода приложения для получения сообщений из Kafka.

```
import java.util.Properties;
import java.util.Arrays;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;
public class SimpleConsumer {
  public static void main(String[] args) throws Exception {
    // проверяем, указан ли топик
    if(args.length == 0){
      System.out.println("Введите имя топика");
      return;
    }
    // забираем имя топика
    String kafkaTopicName = args[0].toString();
    // настраиваем параметры аналогично продьюсеру
    Properties properties = new Properties();

    properties.put("bootstrap.servers", "localhost:9092");
    properties.put("group.id", "test");
    properties.put("enable.auto.commit", "true");
```

```
        properties.put("auto.commit.interval.ms", "1000");
        properties.put("session.timeout.ms", "30000");
        properties.put("key.deserializer",
          "org.apache.kafka.common.serializa-tion.StringDeserializer");
        properties.put("value.deserializer",
          "org.apache.kafka.common.serializa-tion.StringDeserializer");
        // Создаем объект KafkaConsumer
        KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer
          <String, String>(properties);

        // подписываемся а топик
        consumer.subscribe(Arrays.asList(kafkaTopicName))
        // (необязательно) выводим имя топика
        System.out.println("я подписан на топик " + kafkaTopicName);
        int i = 0;
        while (true) {
          ConsumerRecords<String, String> records = kafkaConsumer.poll(100);
          // пробегаем по всем записям
          for (ConsumerRecord<String, String> record : records)
          // печатаем в консоль оффсет и пары ключ - значение
          System.out.printf("оффсет = %d, ключ = %s, значение = %s\n",
            record.offset(), record.key(), record.value());
        }
      }
    }
```

Скомпилируем и запустим приложение. Вывод представлен ниже:

```
я подписан на топик Hello-Kafka
оффсет = 3, ключ = null, значение = тест123
```

## 4.6. Контрольные задания

### 4.6.1. Вопросы для самопроверки

1. Что такое Kafka?
2. Преимущества системы Kafka.
3. Экосистема данных.
4. Что такое журнал фиксации?
5. Что такое отслеживание активности?
6. Что такое обмен сообщениями?
7. Что такое потоковая обработка?

# 5. ИНТЕГРАЦИЯ НА ОСНОВЕ РАСПРЕДЕЛЕННЫХ ОБЪЕКТНЫХ СИСТЕМ: COM/DCOM, CORBA, .NET. ДОСТОИНСТВА, НЕДОСТАТКИ

Распределённая система является системой, для которой отношения местоположений элементов (или групп элементов) играют существенную роль с точки зрения функционирования системы, а, следовательно, и с точки зрения анализа и синтеза системы.

Для распределённых систем характерно распределение функций, ресурсов между множеством элементов (узлов) и отсутствие единого управляющего центра, поэтому выход из строя одного из узлов не приводит к полной остановке всей системы. Типичной распределённой системой является Интернет.

На сегодняшний день выделяются три различные технологии, поддерживающие концепцию распределенных объектных систем. Это технологии COM / DCOM, CORBA и .NET [8].

## 5.1. CORBA

В конце 1980-х и начале 1990-х годов многие ведущие фирмы-разработчики были заняты поиском технологий, которые принесли бы ощутимую пользу на все более изменчивом рынке компьютерных разработок. В качестве такой технологии была определена область распределенных компьютерных систем. Необходимо было разработать единообразную архитектуру, которая позволяла бы осуществлять повторное использование и интеграцию кода, что было особенно важно для разработчиков. Цена за повторное использование кода и интеграцию кода была высока, но никто из разработчиков в одиночку не мог воплотить в реальность мечту о широко используемом, языково-независимом стандарте, включающем в себя поддержку сложных многосвязных приложений. Поэтому в мае 1989 была сформирована OMG (Object Managment Group). Как уже отмечалось, сегодня OMG насчитывает более 700 членов (в OMG входят практически все крупнейшие производители ПО, за исключением Microsoft).

Вот небольшой список достоинств и недостатков использования технологии CORBA.

В CORBA есть ряд существенных достоинств:

1. Язык IDL поддерживает разнообразные программные языки, операционные системы, сети и объектные системы. IDL позволяет отделить описание интерфейса от его реализации.

2. CORBA – сетевая архитектура по определению, эта идея лежит в основе его развития. Объектно-ориентированные интерфейсы CORBA легко определять, создавать и использовать.

3. Каждый сервер может содержать много объектов. Связь между отправителем и адресатом осуществляется напрямую. Объекты могут быть разных размеров.

4. CORBA хорошо сочетается с разнообразным промежуточным ПО, включая OLE языки.

5. Интеграция с другими распространенными технологиями: базами данных, системами обработки сообщений, системами обработки пользовательского интерфейса и другими.

6. Существует протокол IIOP, который позволяет взаимодействовать различным ORB по TCP/IP. CORBA сервисы обеспечивают ряд дополнительных возможностей: транзакции, события, query и т. д. Одновременная поддержка статических и динамических интерфейсов. Возможность включения в распределенную среду Web-клиентов и серверов, в частности, через Java-реализации CORBA.

В CORBA обеспечиваются достаточно развитые функциональные возможности, но в ней отсутствует поддержка безопасности данных.

Нешифрованный трафик в CORBA позволяет производить атаки типа перехвата информации (eavesdropping) и «человек посередине» (man-in-the-middle), и для обеспечения такого трафика требуется наличие открытого порта в корпоративном брандмауэре для каждой службы. Это противоречит обычным корпоративным политикам безопасности. (Между прочим, этот недостаток CORBA был основной предпосылкой возникновения SOAP).

Для коммерческой инфраструктуры электронной коммерции отсутствие поддержки безопасности и версий является просто непреодолимым препятствием – многие потенциальные потребители из области электронной коммерции отказались от CORBA только из-за этого.

Также следующие недостатки идут не на пользу CORBA:

1. Нет передачи параметров «по значению».
2. Отсутствует динамическая загрузка компонент-переходников.
3. Нет именования через URL.

К основным достоинствам CORBA можно отнести межъязыковую и межплатформенную поддержку. Хотя CORBA-сервисы и отнесены к достоинствам технологии CORBA, их в равной степени можно одновременно отнести и к недостаткам CORBA, ввиду практически полного отсутствия их реализации.

## 5.2. COM / DCOM

COM (Component Object Model) – модель компонентного объекта, которая является технологическим стандартом от компании Microsoft, предназначенная для создания программного обеспечения на основе взаимодействующих компонентов объекта, каждый из которых может использоваться во многих программах одновременно. Стандарт воплощает в

себе идеи полиморфизма и инкапсуляции объектно-ориентированного программирования.

DCOM (Distributed COM) – расширение Component Object Model для поддержки связи между объектами на различных компьютерах по сети.

Данные модели построены по принципу клиент-сервер. Сильно упрощая, можно сказать, что сервер экспортирует функции, которые клиент может вызывать, вынуждая сервер выполнить то или иное действие. Если взаимодействие между клиентом и сервером подразумевает обмен данными, эти данные передаются в качестве параметров функций. При необходимости клиент также может экспортировать функции, которые могут быть вызваны сервером.

В модели COM рекомендуется, чтобы функции, экспортируемые сервером и клиентом, возвращали результат типа HRESULT, по которому можно судить об успешном или неуспешном выполнении функции. Даже если логика работы функции подразумевает возвращение какого-то результата, его рекомендуется возвращать через параметры, передаваемые по ссылке, а тип самой функции должен быть HRESULT. В модели DCOM использование HRESULT – это уже не рекомендация, а обязательное требование. Некоторые функции стандартных интерфейсов, доставшиеся DCOM в наследство от COM, тем не менее, имеют тип, отличный от HRESULT. Но это допускается только для стандартных интерфейсов в целях совместимости, а все функции новых серверов должны иметь тип HRESULT.

Обычно ProgID состоит из имени разработчика, имени класса и номера версии, разделённого точками. Имя класса отражает его функциональное предназначение. При регистрации класса в системный реестр заносится и ProgID, и CLSID, поэтому, зная один из этих идентификаторов, можно узнать и второй. Для этого можно использовать системные функции CLSIDFromProgID и ProgIDFromCLSID.

В силу того, что DCOM отличается от COM тем, что в DCOM введена поддержка связи между объектами на различных компьютерах по сети, и достоинства и недостатки DCOM будут справедливы и для COM.

Преимуществом DCOM является, по мнению Карен Буше, аналитика The Standish Group, значительная простота использования. Если программисты пишут свои Windows-приложения с помощью ActiveX (предлагаемого Microsoft способа организации программных компонентов), то операционная система будет автоматически устанавливать необходимые соединения и перенаправлять трафик между компонентами, независимо от того, размещаются ли компоненты на той же машине или нет.

Можно перечислить следующие преимущества.

1. Независимость от языка.
2. Динамический/статический вызов.
3. Динамическое нахождение объектов.
4. Масштабируемость.
5. Открытый стандарт (контроль со стороны TOG).

6. Множественность Windows-программистов.

Однако у DCOM есть и ряд недостатков. "На самом деле это решение до сих пор ориентировано исключительно на системы Microsoft", – считает Буше. Изначально DCOM создавалась под Windows. Хорошо известно, что Microsoft заключила соглашение с компанией Software AG, предмет которого – перенос DCOM на другие платформы. Впрочем, по мнению Буше, значение этой работы достаточно ограниченно, поскольку Microsoft уже успела внести ряд существенных изменений в Windows-версию DCOM.

Можно перечислить следующие недостатки:

1. Сложность реализации.
2. Зависимость от платформы.
3. Нет именования через URL.
4. Нет проверки безопасности на уровне выполнении ActiveX компонент.
5. Отсутствие альтернативных разработчиков.

## 5.3. .NET

Следующим шагом в развитии COM/DCOM стандартов является появление платформы .NET.

.NET Framework – программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду.

Считается, что платформа .NET Framework явилась ответом компании Microsoft на набравшую к тому времени большую популярность платформу Java компании Sun Microsystems (ныне принадлежит Oracle).

Хотя .NET является патентованной технологией корпорации Microsoft и официально рассчитана на работу под операционными системами семейства Microsoft Windows, существуют независимые проекты (прежде всего это Mono и Portable.NET), позволяющие запускать программы .NET на некоторых других операционных системах.

В настоящее время .NET Framework получает развитие в виде .NET Core, изначально предполагающей кроссплатформенную разработку и эксплуатацию.

Относительно недавно появившаяся технология Microsoft .NET имеет много достоинств по сравнению с более ранними технологиями.

Можно выделить следующие достоинства у данной технологии:

- единые средства API для разработки программ на разных языках;
- простота стыковки разноязыковых модулей;
- многие тысячи готовых к употреблению классов, реализующие различные алгоритмы, сокращают сроки разработки новых программ и повышают надежность этих программ;

- установка программ под .NET не требует программ-инсталляторов, делается простое копирование программы в нужную папку. Как следствие, при установке не вносятся ни какие записи в реестр Windows, поэтому после удаления таких программ в реестре не остается «мусор».

Новая технология имеет и ряд недостатков, потому что она была не обкатана на момент появления относительно уже используемых ранее технологий:

- замедление при выполнении программ;
- привязанность некоторых архитектурных решений .NET к C++ - подобным языкам;
- необходимость изменения стандартов для многих языков программирования.

Естественно, что все преимущества .NET, которые мы перечислили выше, не могут быть абсолютно бесплатными. Как и у любой другой архитектуры, у .NET есть свои недостатки.

Самым ощутимым недостатком является существенное замедление выполнения программ. Это неудивительно, так как между исходным языком и машинным кодом вводится дополнительный уровень MSIL.

Однако промежуточное представление .NET с самого начала проектировалось с прицелом на компиляцию времени исполнения.

Другая проблема .NET заключается в том, что при ее создании основной упор был сделан на C++/Java-подобные языки.

Это ограничивает возможности интеграции некоторых языков с более богатыми возможностями, особенно с принципиально отличающимися языками, такими как функциональные языки или устаревшие языки.

Наконец, наблюдается и движение с противоположной стороны: уже сегодня стандарты некоторых языков программирования претерпевают значительные изменения для того, чтобы эти языки могли быть поддержаны в .NET.

## 5.4 . Контрольные задания

### 5.4.1. Вопросы для самопроверки

1. Что такое распределённая система?
2. Технология CORBA.
3. Спецификация OMG.
4. Спецификация инфраструктуры технологией CORBA.
5. Модель COM.
6. Расширение DCOM.
7. Преимущества и недостатки DCOM.
8. Платформа .NET.
9. Достоинства и недостатки .NET.

# 6. ЯЗЫК XML - ВВЕДЕНИЕ, ОСНОВНЫЕ КОНСТРУКЦИИ, МЕТОДЫ ВАЛИДАЦИИ

Расширяемый язык разметки (XML) - это язык разметки, который определяет набор правил для кодирования документов в формате, удобном для чтения и считываемом компьютером. Спецификация консорциума World Wide Web XML 1.0 1998 года и ряд других связанных спецификаций - все из них являются бесплатными открытыми стандартами - определяют XML.

Цели разработки XML подчеркивают простоту, универсальность и удобство использования в Интернете. Это текстовый формат данных с сильной поддержкой через Unicode для разных человеческих языков. Хотя дизайн XML ориентирован на документы, этот язык широко используется для представления произвольных структур данных, таких как используемые в веб-сервисах.

Были разработаны сотни форматов документов с использованием синтаксиса XML, включая RSS, Atom, SOAP, SVG и XHTML. XML-форматы на основе стали по умолчанию для многих инструментов офисно-производительности, в том числе Microsoft Office (Office Open XML), OpenOffice.org и LibreOffice (OpenDocument), и Apple, «s iWork. XML также предоставил базовый язык для протоколов связи, таких как XMPP, Приложения для Microsoft .NET Framework используют XML-файлы для конфигурации, а списки свойств представляют собой реализацию хранилища конфигурации, построенного на XML.

Многие отраслевые стандарты данных, такие как Health Level 7, OpenTravel Alliance, FpML, MISMO и модель обмена национальной информацией, основаны на XML и богатых функциях спецификации схемы XML. Многие из этих стандартов довольно сложны, и спецификация нередко состоит из нескольких тысяч страниц. XML широко используется для поддержки различных форматов публикации.

## 6.1. Язык XML

XML широко используется в сервис-ориентированной архитектуре (SOA). Отдельные системы взаимодействуют друг с другом путем обмена сообщениями XML. Формат обмена сообщениями стандартизирован как схема XML (XSD). Это также называется канонической схемой. XML широко используется для обмена данными через Интернет.

XML - это язык разметки, который используется для хранения и транспортировки данных. XML не зависит от платформы и программного обеспечения (языка программирования). Вы можете написать программу на любом языке на любой платформе (операционной системе) для отправки, получения или хранения данных с использованием XML.

XML предназначен для хранения данных, при этом не выполняя с ними никаких операций.

Эта заметка представляет собой заметку Tove от Jani, хранящуюся в формате XML:

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Приведенный выше XML довольно информативен:

- имеется информация об отправителе;
- имеется информация о получателе;
- имеется заголовок;
- имеется тело сообщения.

Но все же приведенный выше XML ничего не делает. XML - это просто информация, завернутая в теги.

Кто-то должен написать часть программного обеспечения для отправки, получения, хранения или отображения.

Разница между XML и HTML

XML и HTML были разработаны с разными целями:

- XML был разработан для переноса данных;
- HTML был разработан для отображения данных;
- XML-теги не предопределены, как HTML-теги.

Зачем нам нужен XML? Так как существуют системы с разными операционными системами, имеющие данные в разных форматах. Передача данных между этими системами является сложной задачей, поскольку данные необходимо преобразовать в совместимые форматы, прежде чем их можно будет использовать в другой системе. С XML так легко передавать данные между такими системами, что XML не зависит от платформы и языка. XML - это простой документ с данными, который можно использовать для хранения и передачи данных между любыми системами независимо от их аппаратной и программной совместимости.

## 6.2. Основные конструкции

В общем случае XML-документы должны удовлетворять следующим требованиям:

- в заголовке документа помещается объявление XML, в котором указывается язык разметки документа, номер его версии и дополнительная информация;

- каждый открывающий тэг, определяющий некоторую область данных в документе, обязательно должен иметь своего закрывающего "напарника", т. е., в отличие от HTML, нельзя опускать закрывающие тэги;
- в XML учитывается регистр символов;
- все значения атрибутов, используемых в определении тэгов, должны быть заключены в кавычки;
- вложенность тэгов в XML строго контролируется, поэтому необходимо следить за порядком следования открывающих и закрывающих тэгов.

Для того чтобы обеспечить проверку корректности XML-документов, необходимо использовать анализаторы, производящие такую проверку и называемые верифицирующими.

## 6.3. Цели разработки XML

Спецификация XML устанавливает следующие цели для XML: это ссылки на расширяемый язык разметки для рекомендаций W3C (XML) 1.0 .

1. Использование XML через Интернет должно быть простым. Пользователи должны иметь возможность просматривать документы XML так же быстро и легко, как документы HTML.
2. XML должен поддерживать широкий спектр приложений. XML должен быть полезен для широкого круга разнообразных приложений: создание, просмотр, анализ содержимого и т. д.
3. XML должен быть совместим с SGML. Большинство людей, вовлеченных в работу по XML, происходят из организаций, которые имеют большой, в некоторых случаях ошеломляющий, объем материала в SGML.
4. Должно быть легко писать программы, которые обрабатывают документы XML.
5. Количество дополнительных функций в XML должно быть сведено к абсолютному минимуму, в идеале ноль. Дополнительные функции неизбежно вызывают проблемы совместимости, когда пользователи хотят обмениваться документами, а иногда приводят к путанице и разочарованию.
6. XML-документы должны быть удобочитаемыми и достаточно понятными. Если у вас нет браузера XML и вы откуда-то получили кусок XML, вы должны иметь возможность просмотреть его в своем любимом текстовом редакторе и выяснить, что означает контент.
7. Дизайн XML должен быть подготовлен быстро. Усилия по стандартизации общеизвестно медленны. XML был необходим немедленно и был разработан как можно быстрее.
8. Дизайн XML должен быть формальным и лаконичным. Во многих отношениях это следствие правила 4, по сути, это означает, что XML должен быть выражен в EBNF и должен соответствовать современным инструментам и методам компилятора. Написание правильного синтаксического анализатора

SGML требует обработки множества редко используемых и сложных для анализа функций языка. XML нет.

9. XML-документы должны быть просты в создании. Хотя со временем появятся сложные редакторы для создания и редактирования содержимого XML, они появятся не сразу.

10. Краткость в разметке XML имеет минимальное значение. Несколько языковых функций SGML были разработаны для минимизации объема ввода, необходимого для ручного ввода документов SGML.

Конструкции языка

Содержимое XML-документа представляет собой набор элементов, секций CDATA, директив анализатора, комментариев, спецсимволов, текстовых данных. Рассмотрим каждый из них подробней.

Элементы данных

Элемент - это структурная единица XML-документа. Заключая слово rose в тэги <myFlower> </myFlower>, мы определяем непустой элемент, называемый <myFlower>, содержимым которого является rose.

Любой непустой элемент должен состоять из начального, конечного тэгов и данных, между ними заключенных. Например, следующие фрагменты будут являться элементами:

<myFlower>rose</myFlower>

<myCity>Novosibirsk</myCity>

а эти - нет:

<myRose>

<myFlower>

myRose

Набором всех элементов, содержащихся в документе, задается его структура и определяются все иерархическое соотношения. Плоская модель данных превращается с использованием элементов в сложную иерархическую систему со множеством возможных связей между элементами.

<myCountry id="Russia">

<myCities-list>

<myCity>

<myTitle>Новосибирск</myTitle>

<myState>Siberia</myState>

<myUniversities-list>

<myUniversity id="2">

<myTitle>Новосибирский Государственный Технический Университет</myTitle>

<noprivate/>

<myAddress URL="www.nstu.ru"/>

<myDescription>очень хороший институт</myDescription>

</myUniversity>

<myUniversity id="2">
<myTitle>Новосибирский Государственный Университет</myTitle>
<noprivate/>
<myAddress URL="www.nsu.ru"/>
<myDescription>тоже не плохой</myDescription>
</myUniversity>
</myUniversities-list>
</myCity>
</myCities-list>
</myCountry>

Производя в последствии поиск в этом документе, программа клиента будет опираться на информацию, заложенную в его структуру, используя элементы документа. Т.е. если, например, требуется найти нужный университет в нужном городе, используя приведенный фрагмент документа, то необходимо будет просмотреть содержимое конкретного элемента <myUniversity>, находящегося внутри конкретного элемента <myCity>. Поиск при этом, естественно, будет гораздо более эффективен, чем нахождение нужной последовательности по всему документу.

Например, прочитав фрагмент <myCity>Holliwood</myCity>, мы можем догадаться, что речь в этой части документа идет о городе, а вот во фрагменте <restaurant>Holliwood</restaurant> - о забегаловке.

В случае если элемент не имеет содержимого, т.е. нет данных, которые он должен определять, он называется пустым. Примером пустых элементов в HTML могут служить такие тэги HTML, как <br>, <hr>, <img>;. Необходимо только помнить, что начальный и конечный тэги пустого элемента как бы объединяются в один, и надо обязательно ставить косую черту перед закрывающей угловой скобкой (например, <empty/>;).

Комментарии

Комментариями является любая область данных, заключенная между последовательностями символов <!-- и -->. Комментарии пропускаются анализатором и поэтому при разборе структуры документа в качестве значащей информации не рассматриваются.

Атрибуты

Если при определении элементов необходимо задать какие-либо параметры, уточняющие его характеристики, то имеется возможность использовать атрибуты элемента. Атрибут - это пара "название" = "значение", которую надо задавать при определении элемента в начальном тэге. Пример:

<color RGB="true">#ff08ff</color>
<color RGB="false">white</color>

или

<author id=0>Ivan Petrov</author>

Примером использования атрибутов в HTML является описание элемента <font>:

<font color=¦white¦ name=¦Arial¦>Black</font>

Директивы анализатора

Инструкции, предназначенные для анализаторов языка, описываются в XML-документе при помощи специальных тэгов - <? и ?>;. Программа клиента использует эти инструкции для управления процессом разбора документа. Наиболее часто инструкции используются при определении типа документа (например, <? Xml version=¦1.0¦?>) или создании пространства имен.

CDATA

Чтобы задать область документа, которую при разборе анализатор будет рассматривать как простой текст, игнорируя любые инструкции и специальные символы, но, в отличии от комментариев, иметь возможность использовать их в приложении, необходимо использовать тэги <![CDATA] и ]]>.

## 6.4. Методы валидации и проверки на корректность

Проверка XML - это процесс проверки документа, написанного на XML (расширяемый язык разметки), чтобы убедиться, что он правильно сформирован и «действителен» в том смысле, что он соответствует определенной структуре. Правильно оформленный документ следует основным синтаксическим правилам XML, которые одинаковы для всех документов XML. Действительный документ также соблюдает правила, определенные конкретной схемой DTD или XML. Автоматизированные инструменты - валидаторы - могут выполнять тесты на корректность и многие другие валидационные тесты, но не такие, которые требуют человеческого суждения, например, правильное применение схемы к набору данных.

Проверка DTD.

DTD обеспечивают базовый контроль над именами элементов и атрибутов и общей структурой документа XML. DTD обычно просты в написании и понимании. Следующий документ XML может быть проверен с помощью краткого DTD, который следует:

```
1 | <? xml version = "1.0"?>
2 | <! DOCTYPE Список СИСТЕМА "list.dtd">
3 | < List name = "Список фруктов">
4 | < Item > Apple </ Item >
5 | < Item > Banana </ Item >
6 | < Item > Pear </ Item >
7 | </ List >
```

```
1 | <! - list.dtd ->
2 | <! Список элементов  ( Item + )>
```

3 | <! Список ATTLIST
4 | имя CDATA # ПРЕДПОЛАГАЕТСЯ >
5 | <! Элемент ЭЛЕМЕНТ (#PCDATA )>

Во второй строке приведенного выше XML-документа DOCTYPE объявление связывает документ с местоположением DTD для проверки документа. DOCTYPE декларация не является единственным способом проверить с помощью DTD, но это общий один.

Проверка XML-схемы

XML-схема является значительно более мощным механизмом проверки, чем DTD, поскольку она добавляет типы данных и более сложные структурные ограничения. Пространства имен полностью поддерживаются в XML-схеме.

Одним из наиболее интуитивно выгодных аспектов XML-схем является то, что они выражаются в виде XML-документов. Наличие схемы, выраженной в XML, означает, что информация в схеме программно доступна через те же стандартные интерфейсы XML, с которыми вы, вероятно, уже работаете.

Спецификация XML-схемы разделена на три части. Первая часть называется учебником и описывает базовое использование XML-схемы. Вторая часть посвящена механизму структуры или модели содержимого. И последний раздел описывает типы данных.

Структура XML-схемы.

Схема XML использует несколько объектно-ориентированный подход к описанию модели содержимого документа XML. Простые типы объединяются в более сложные типы, а элементы определяются в терминах этих типов. XML-схема поддерживает элементарное наследование для сложных типов, позволяя производным типам расширять или ограничивать базовые типы.

Вот пример документа XML-экземпляра, который ссылается на XML-схему:

1 | <? xml version = "1.0"?>
2 | < List name = "Список фруктов"
3 | xmlns = http: //liquidhub.com/SimpleList
4 | xmlns: xsi = http: //www.w3.org/2001/XMLSchema-instance
5 | xsi: schemaLocation = "http://liquidhub.com/SimpleList list.xsd">
6 | < Item > Apple </ Item >
7 | < Item > Banana </ Item >
8 | < Item > Груша </ Item > </ List >

На строках четыре и пять в приведенном выше документе XML атрибут schema Location является распространенным методом для связывания экземпляра документа в XML с его схемой. Хотя это удобно для тестирования, вы не захотите доверять schema Location документам, созданным вне вашего контроля. Как правило, вы используете API-интерфейс вашей реализации XML, чтобы предоставить достоверную XML-схему для проверки.

Вот пример XML-схемы:

1 | <! - list.xsd ->

```
2 | < schema xmlns = "http://www.w3.org/2001/XMLSchema"
3 | xmlns: lh = "http://liquidhub.com/SimpleList"
4 | targetNamespace = "http://liquidhub.com/SimpleList"
5 | elementFormDefault = "qualified">
6 | < complexType name = "SimpleList">
7 | < последовательность >
8 | < element name = "Item" type = "string"
9 | maxOccurs = "неограниченный" />
10 | </ последовательность >
11 | < attribute name = "name" type = "string" />
12 | </ complexType >
13 | < element name = "List" type = "lh: SimpleList" />
14 | </ schema >
```

После множества стандартных схем создается complex Type определение для нашего Simple List типа. Этот тип состоит из последовательности одного или нескольких Item элементов и name атрибута. Наконец, корневой List элемент объявлен глобально как тип Simple List.

RelaxNG Validation. DTD и XML Schema - не единственные проверочные игры в городе. RelaxNG был создан как ответ на сложность XML-схемы. Схема XML провела много времени в процессе стандартизации и подвергается критике за чрезмерную разработку. RelaxNG - это мощный механизм проверки, который намного проще, чем XML-схема.

Как и XML-схемы, схемы RelaxNG выражаются в виде документов XML, хотя RelaxNG также имеет компактный формат, который не является XML. Вы можете сравнить два формата в эквивалентных схемах RelaxNG ниже:

```
1 | <! - list.rng ->
2 | < element  name = "List"  xmlns = "http://relaxng.org/ns/structure/1.0">
3 | < attribute  name = "name">
4 | < text />
5 | </ attribute >
6 | < oneOrMore >
7 | < element  name = "Item">
8 | < text />
9 | </ element >
10 | </ oneOrMore >
11 | <
```

```
1 | # list.rng (компактный формат)
2 | Список элементов {
3 | имя атрибута {  текст  },
4 | элемент Item {  text  } +
5 | }
```

Наборы инструментов для выполнения проверки RelaxNG широко доступны для Java и несколько доступны для .NET. К сожалению, RelaxNG вряд ли когда-нибудь станет частью служб Microsoft XML.

XML-схема имеет элементы аннотации, предназначенные для хранения документации или дополнительной информации о приложении. Элемент appInfo является идеальным местом для встраивания утверждений Schematron в вашу схему. Напишите класс-оболочку проверки, который может извлечь и выполнить эти утверждения после успешной проверки схемы.

Schematron - это метод проверки, основанный на утверждениях XPath относительно древовидных структур XML. Реализации проверки Schematron доступны в виде таблиц стилей XSL. Schematron предназначен для дополнения других типов проверки.

Вот пример схемы Schematron с одним утверждением:

```
1 | < schema xmlns = "http://www.ascc.net/xml/schematron">
2 | < pattern name = "No Duplicate Items">
3 | < rule context = "Item">
4 | < assert test = "not (previous-sibling :: Item =.)">
5 | Дублирование предметов не допускается!
6 | </ assert >
7 | </ rule >
8 | </ pattern >
9 | </ schema >
```

В XSL-реализации Schematron вы подаете схему Schematron, подобную приведенной выше, в XSL-преобразование скелета Schematron, которое создает второе проверочное XSL-преобразование. Это сгенерированное проверочное XSL-преобразование запускается для экземпляра XML, что приводит к списку любых ошибок валидации. Ошибки форматируются любым способом, указанным в схеме Schematron, включая простой текст или XML. Шаблон сводится к следующему: создайте простую грамматику XML, которую можно обрабатывать с помощью XSL, чтобы создать преобразование XSL, которое делает что-то полезное для других экземпляров XML.

XML Schema и RelaxNG, безусловно, могут проверять все, что может DTD, но DTD остаются актуальными из-за их широкого использования. Отрасли, которые рано стали использовать XML-спецификации, создали DTD.

Выбор механизма проверки будет, естественно, зависеть от доступных инструментов. Разработчики Java имеют наименьшее количество препятствий для использования любого из методов проверки, обсужденных здесь. Когда у вас есть выбор методов проверки, вы должны учитывать относительную выразительность и простоту использования каждого из них. График методов проверки изображен на рисунке.

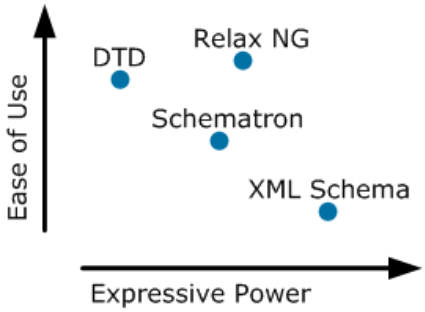


Рис. График методов проверки

Рассматривая этот график, учтите, что объем усилий, необходимых для реализации более выразительных методов проверки, может окупиться с меньшими затратами на кодирование.

Relax NG действительно удачное место для простоты использования и выразительной силы. Даже если вы являетесь разработчиком .NET, то, что Microsoft вряд ли внедрит Relax NG в свои основные XML-сервисы, не означает, что вам следует полностью отказаться от Relax NG.

Schematron было сложно разместить на графике. Schematron может выражать контент-зависимые правила через элементы и атрибуты, что не может сделать ни один из других методов. С другой стороны, Schematron не так хорошо подходит для основных ограничений модели контента, как другие. Schematron лучше всего использовать в качестве дополнительной мощности для других методов проверки.

Мало кто будет спорить с размещением DTD и XML Schema на графике. DTD имеет преимущество и недостаток в том, что он быстрый и легкий. DTD широко используется и может значительно упростить код в ваших приложениях без особых усилий. Но как только вы увидите, что предлагает XML-схема, с DTD трудно быть довольным.

## 6.5. Контрольные задания

### 6.5.1. Вопросы для самопроверки

1. Определение языка XML.
2. Требование к XML-документам.
3. Что такое формально-правильный XML-документ?
4. Цели XML-спецификаций.
5. Конструкции языка XML.
6. Проверка XML и DTD.

# 7. ВВЕДЕНИЕ В XSLT. XSLT-ПРОЦЕССОРЫ. ЯЗЫК ОПИСАНИЯ СХЕМ

Представьте на минуту, что у вас есть набор данных в XML. Возможно, вы предпочли хранить их таким образом из-за гибкости XML. Вы знаете, что сможете использовать их на разных платформах и практически с любым языком программирования. Но иногда вам придется менять форму вашего XML. Например, вам может понадобиться перевести данные в другой XML-формат, чтобы хранить их в другой системе, или в другую форму для представления или еще для чего-нибудь.

Например, у вас могут быть данные в XML, которые вы хотите опубликовать в Интернете, что означает перевод их в HTML. Конечно, вы могли бы вручную просмотреть документ и сделать в нем необходимые изменения, или, может быть, вы подумали о том, чтобы загрузить XML в DOM, а затем вручную создать документ вывода.

Преобразования расширяемого языка таблиц стилей (XSLT) предоставляют способ для автоматического перевода XML-данных из одной формы в другую. Целевая форма - это обычно другой XML-документ, но не обязательно; вы можете преобразовать XML практически во что угодно, просто создав таблицу стилей XSLT и обработав данные [9].

## 7.1. Язык XSLT

XSLT – это расширяемый язык таблиц стилей. Язык XSL фактически состоит из двух частей: языка преобразований и языка форматирования. Язык, предназначенный для выполнения преобразований, позволяет конвертировать структуры документов в различные формы, в то время как язык форматирования используется для оформления и определения стилей документов различными способами. Обе части языка XSL могут функционировать совершенно независимо одна от другой, поэтому их можно рассматривать в качестве независимых языков разметки.

Язык XSLT применяется для обработки документов, внесения изменений и необходимых дополнений в разметку. Его можно применять для преобразования XML-кода в отформатированный HTML-код.

Язык XSLT обеспечивает доступ к содержимому XML-документов, а также применяется для создания новых документов на их основе.

Для выполнения XSLT-преобразований используется два документа: преобразуемый документ и таблица стилей, определяющая само преобразование. В данном случае речь идет об XML-документах.

## 7.2. XPath

XPath (XML Path Language) — язык запросов к элементам XML-документа. Разработан для организации доступа к частям документа XML в

файлах трансформации XSLT и является стандартом консорциума W3C. XPath призван реализовать навигацию по DOM в XML. В XPath используется компактный синтаксис, отличный от принятого в XML. В 2007 году завершилась разработка версии 2.0, которая теперь является составной частью языка XQuery 1.0. В декабре 2009 года началась разработка версии 2.1, которая использует XQuery 1.1.

XML имеет древовидную структуру. В самостоятельном XML-документе всегда имеется один корневой элемент, в котором допустим ряд вложенных элементов, некоторые из которых тоже могут содержать вложенные элементы. Также могут встречаться текстовые узлы, комментарии и инструкции. Можно считать, что XML-элемент содержит массив вложенных в него элементов и массив атрибутов.

У элементов дерева бывают элементы-предки и элементы-потомки. Каждый элемент дерева находится на определённом уровне вложенности. Элементы упорядочены в порядке расположения в тексте XML, и поэтому можно говорить об их предыдущих и следующих элементах. Это очень похоже на организацию каталогов в файловой системе.

Строка XPath описывает способ выбора нужных элементов из массива элементов, которые могут содержать вложенные элементы. Начинается отбор с переданного множества элементов, на каждом шаге пути отбираются элементы, соответствующие выражению шага, и в результате оказывается отобрано подмножество элементов, соответствующих данному пути.

## 7.3. XSLT–процессоры

Весь написанный нами XSL-код исполняет XSL-трансформатор (он же XSL-процессор).

XSLT содержит много разных модулей, среди которых и расширенная работа со строками, и регулярные выражения, и пользовательские XPath-функции, и генерация случайного числа. Все зависит лишь от поддержки каждого модуля трансформаторами.

Существует достаточно большое количество различных XSLT - процессоров. Рассмотрим самые популярные из них, а именно: libxslt, Xalan, Saxon и MSXML.

Libxslt – самый широко используемый и, наверное, самый старинный процессор, который базируется на другой библиотеке libxml. Библиотека libxslt является примером того, насколько сильным может быть open-source продукт, даже если он создаётся в основном одним человеком.

Библиотека libxslt изначально создавалась для поддержки XSLT-преобразований в проекте Gnome.

В libxslt реализованы почти все модули EXSLT, в том числе функция node-set(), поэтому на этом процессоре временные деревья можно использовать без каких-либо опасений.

Xalan – это очень известный XSLT-процессор, созданный в рамках Apache XML Project для языковых платформ Java и C++. Xalan, как и остальные продукты Apache XML Project, поставляется с открытым исходным кодом и открытым API, что делает его очень привлекательным для интеграции в другие приложения.

По сути дела, Xalan Java и Xalan C++ – это библиотеки, позволяющие использовать XSLT-преобразования в собственных проектах. Xalan Java поддерживает набор интерфейсов TrAX, определяющий стандартные модели и методы преобразования XML-документов в Java-программах.

Как результат, разработанный и поддерживаемый им процессор считается образцом соответствия стандарту XSLT. В целом, Saxon можно описать как выдающийся продукт от выдающегося человека.

MSXML — набор XML-библиотек Microsoft, в состав которого входит и XSL-трансформатор. Поддерживается только XSLT 1.0 и нет поддержки EXSLT.

Вместо этого предлагается пользоваться расширением самого Microsoft, в котором есть аналогичная функция node-set(). В общем, это лучше, чем ничего, но использовать тот же код на других трансформаторах, понятно, не получится.

1. Практический пример

Рассмотрим простой следующий XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<employees>
  <employee>
    <name>Петя</name>
    <position>Программист</position>
    <salary caption="$">400</salary>
  </employee>
   <employee>
    <name>Вася</name>
    <position>Тестировщик</position>
    <salary caption="$">500</salary>
  </employee>
</employees>
```

Если открыть XML в браузере, то отобразится тот же самый текст, который приведен выше, вместе со всеми тегами и служебной информацией (рис. 7.1):

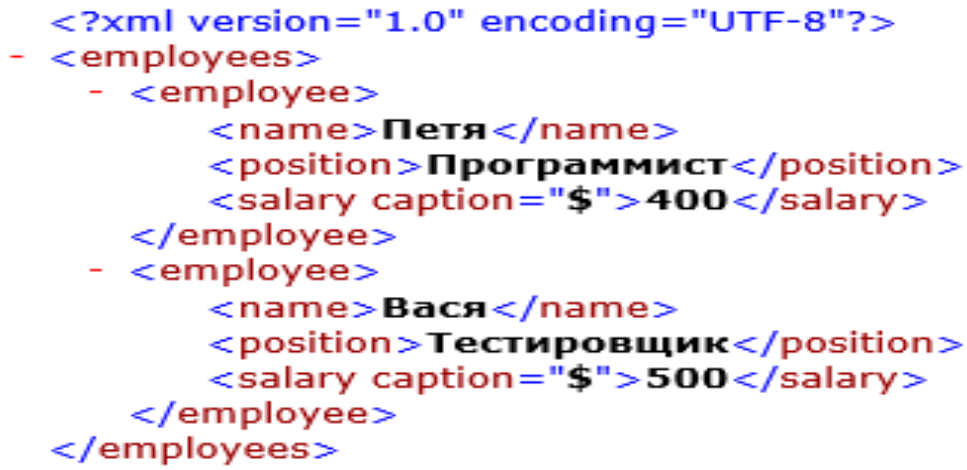



Рис. 7.1. Отображение XML

Данная информация является лишней, так как нам интересно лишь отображение значений тегов. Эта задача решается легко и просто: необходимо к XML-файлу добавить шаблон преобразования - XSL-файл.

Требуется привести XML к следующему виду:

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet type='text/xsl' href='1.xsl'?>
<employees>
 <employee>
  <name>Петя</name>
  <position>Программист</position>
  <salary caption="$">400</salary>
 </employee>
  <employee>
  <name>Вася</name>
  <position>Тестировщик</position>
  <salary caption="$">500</salary>
 </employee>
</employees>
```

Также требуется создать XSL:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml">
<xsl:template match="/">
<table border="1">
<tr bgcolor="#CCCCCC">
<td align="center"><strong>Имя</strong></td>
<td align="center"><strong>Должность</strong></td>
<td align="center"><strong>Зарплата</strong></td>
</tr>
<xsl:for-each select="employees/employee">
<tr bgcolor="#F5F5F5">
<td><xsl:value-of select="name"/></td>
<td><xsl:value-of select="position"/></td>
<td align="right"><xsl:value-of select="salary"/><xsl:value-of
select="salary/@caption"/></td>
</tr>
</xsl:for-each>
</table>
</xsl:template>
</xsl:stylesheet>
```

Теперь при открытии в браузере никакой лишней информации отображено не будет (рис. 7.2):

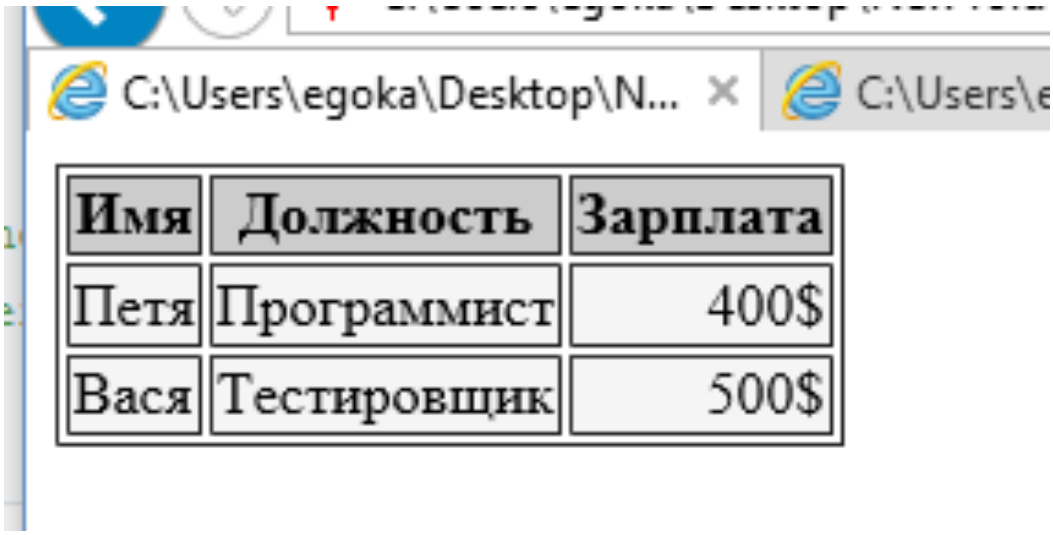



Рис. 7.2. Отображение таблицы

Для того чтобы браузер выполнил необходимое преобразование, нужно в XML-файле указать ссылку на XSL-файл - <?xml-stylesheet type='text/xsl' href='1.xsl'?>.

Рассмотрим теперь текст XSL-файла. Первая строка файла содержит тег элемента xsl:stylesheet. Атрибуты элемента - номер версии и ссылка на пространство имен. Эти атрибуты элемента xsl:stylesheet являются обязательными. В нашем случае пространство имен - это все имена элементов и их атрибутов, которые могут использоваться в XSL-файле. Для XSL-файлов ссылка на пространство имен является стандартной.

Элемент xsl:value-of помогает нам получить содержание элемента. При помощи того же самого элемента можно вывести значение атрибута элемента.

Ссылка на атрибут элемента выглядят следующим образом - //employee /@name. Имя элемента и имя атрибута разделены парой символов "/@". В остальном синтаксис тот же самый, что и для ссылки на содержание элемента.

### 7.4. Контрольные задания

#### 7.4.1. Вопросы для самопроверки

1. Что такое XSLT?
2. Что такое язык преобразования?
3. Что такое язык форматирования?
4. Что такое XPath?
5. Структура XPath.
6. XSLT процессоры.
7. Что такое MSXML?

## 8. JSON: КАК УСТРОЕН, ГДЕ ИСПОЛЬЗУЕТСЯ, ЗАЧЕМ НУЖЕН

JSON (с англ. JavaScript Object Notation) - текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми. Формат JSON был разработан Дугласом Крокфордом.

Как можно понять из названия, JSON произошел из JavaScript - языка програмирования, но он доступен для использования на многих языках, включая Python, Ruby, PHP и Java, в англоязычных странах его в основном произносят как Jason, то есть как имя ДжЭйсон, в русскоязычных странах ударение преимущественно ставится на “о” — ДжисОн.

Легкочитаемый и компактный JSON предлагает хорошую альтернативу XML и требует гораздо меньше форматирования.

## 8.1. Структура JSON

JSON-текст представляет собой (в закодированном виде) одну из двух структур:

– набор пар ключ: значение. В различных языках это реализовано как объект, запись, структура, словарь, хеш-таблица, список с ключом или ассоциативный массив. Ключом может быть только строка (регистрозависимая: имена с буквами в разных регистрах считаются разными), значением — любая форма;

– упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

Это универсальные структуры данных: как правило, любой современный язык программирования поддерживает их в той или иной форме. Они легли в основу JSON, так как он используется для обмена данными между различными языками программирования.

В качестве значений в JSON могут быть использованы:

– объект — это неупорядоченное множество пар ключ: значение, заключённое в фигурные скобки { }. Ключ описывается строкой, между ним и значением стоит символ «:» [10]. Пары ключ: значение отделяются друг от друга запятыми (рис. 8.1);

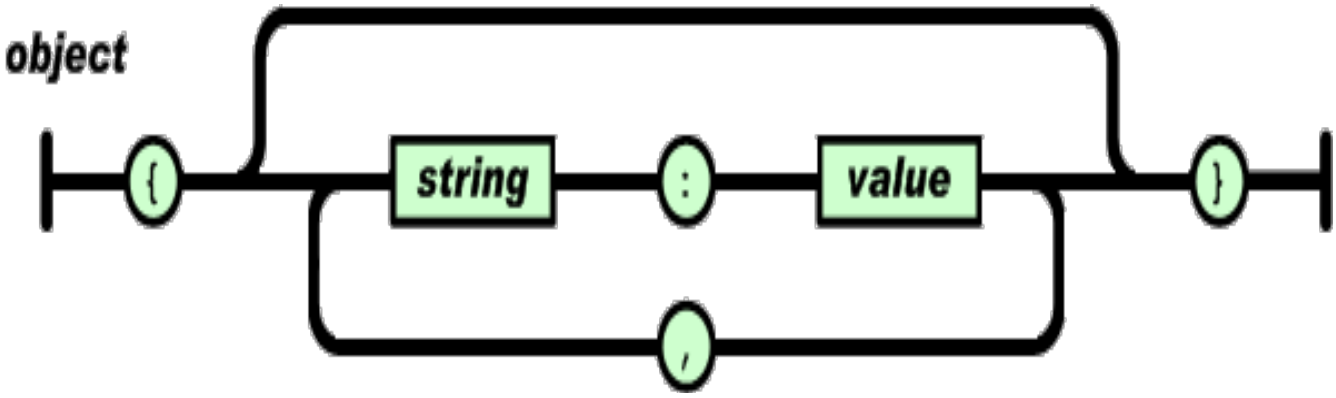



Рис. 8.1. Структура JSON с объектом.

– массив (одномерный) — это упорядоченное множество значений. Массив заключается в квадратные скобки [ ]. Значения разделяются запятыми (рис. 8.2);

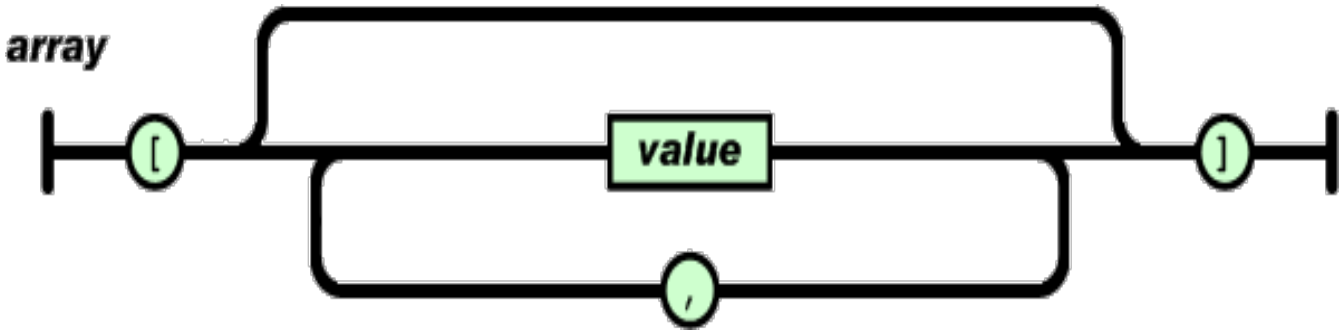



Рис. 8.2. Структура JSON с массивом.

– число (рис. 8.3);

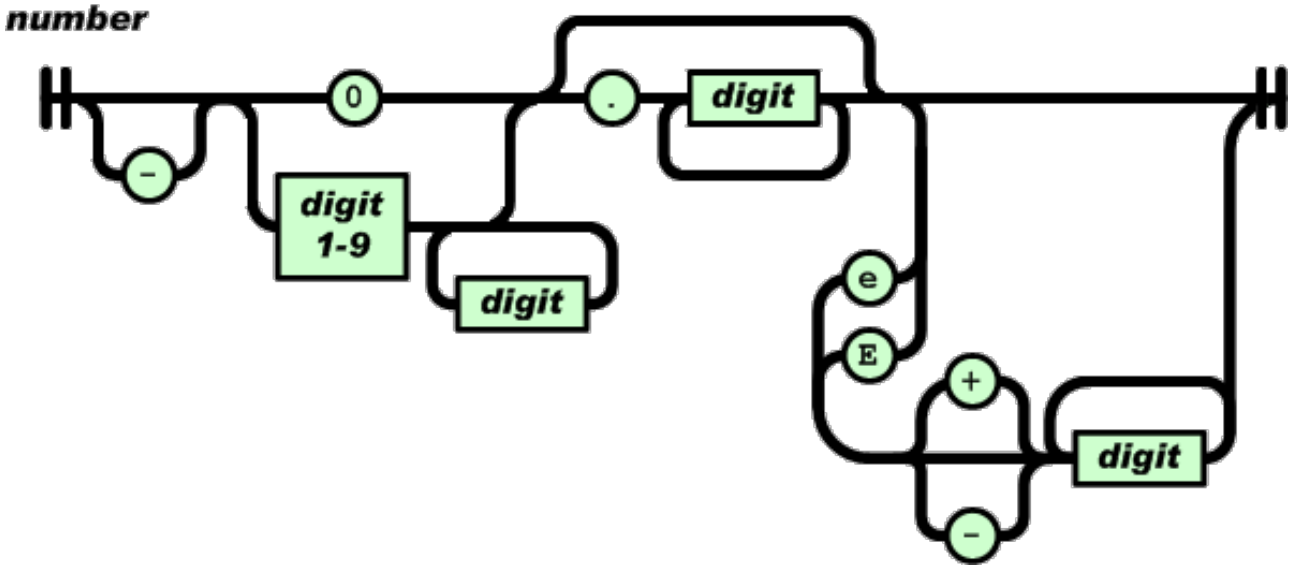



Рис. 8.3. Структура JSON с числом

– литералы (true, false и null);

– строка — это упорядоченное множество из нуля или более символов юникода, заключённое в двойные кавычки. Символы могут быть указаны с использованием escape-последовательностей, начинающихся с обратной косой черты «\» (поддерживаются варианты \', \", \\, \/, \t, \n, \r, \f и \b), или записаны шестнадцатеричным кодом в кодировке Unicode в виде \uFFFF (рис. 8.4).

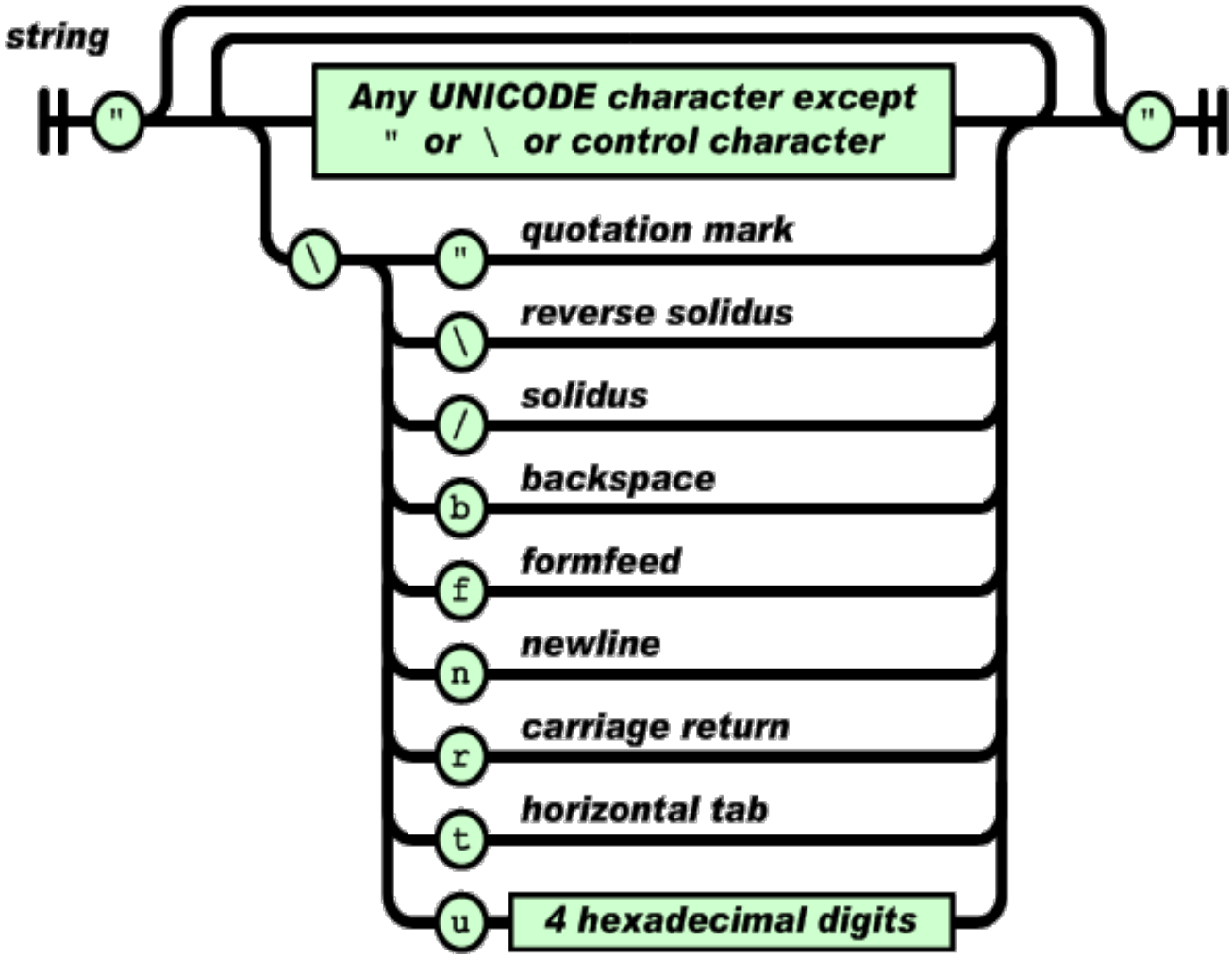



Рис. 8.4. Структура JSON со строкой

Строка очень похожа на одноимённый тип данных в языках C и Java. Число тоже очень похоже на C- или Java-число, за исключением того, что используется только десятичный формат. Пробелы могут быть вставлены между любыми двумя синтаксическими элементами.

## 8.2. Пример JSON

Так выглядит JSON объект:

```
{
“first_name” : “John”,
“last_name” : “Kanefield”,
“location” : “RiverSide”,
“online” : true,
“followers” : 987
}
```

Этот пример показывает то, что этот формат в основном устанавливается двумя фигурными скобками, которые выглядят так { }, а данные с ключевыми значениями находятся между ними. Большинство используемых данных в JSON заключаются в JSON объекты.

Хоть в .json файлах мы обычно видим формат нескольких строк, JSON также может быть написан в одну сплошную строку.

{ “first_name” : “John”, “last_name”: “Kanefield”, “online” : true, }

Работа с JSON в многострочном формате зачастую делает его более читабельным, особенно когда вы пытаетесь справиться с большим набором данных.

## 8.3. Работа с комплексными типами в JSON

JSON может содержать вложенные объекты в JSON формате, в дополнение к вложенным массивам. Такие объекты и массивы будут передаваться как значения ключам и обычно будут состоять из пар ключевых значений [11].

### 8.3.1. Вложенные объекты

В users.json файле ниже для каждого из четырех пользователей (“John”, “jesse”, “drew”, “jamie”) есть вложенный JSON объект, передающий значение для каждого из пользователей, со своими собственными вложенными ключами “username” и “location”, которые относятся к каждому из пользователей. Первый вложенный JSON объект подсвечен ниже:

```
{
“John” : {
“username” : “JohnKanefield”,
“location” : “Indian RiverSide”,
“online” : true,
“followers” : 987
},
“jesse” : {
“username” : “JesseOctopus”,
“location” : “Pacific RiverSide”,
“online” : false,
“followers” : 432
},
“drew” : {
“username” : “DrewSquid”,
“location” : “Atlantic RiverSide”,
“online” : false,
“followers” : 321
```

```
},
“jamie” : {
“username” : “JamieMantisShrimp”,
“location” : “Pacific RiverSide”,
“online” : true,
“followers” : 654
}
}
```

В примере выше фигурные скобки используются везде для формирования вложенного JSON объекта с ассоциированными именами пользователей и данными локаций для каждого пользователя. Как и любое другое значение, используя объекты, двоеточие использовалось для разделения элементов.

### 8.3.2. Вложенные массивы

Данные также могут быть вложены в формате JSON, используя JavaScript массивы, которые передаются как значения. JavaScript использует квадратные скобки [ ] для определения массива. Массивы - это упорядоченные коллекции и могут включать в себя значения разнящихся типов данных [12].

В этом примере первый вложенный массив подсвечен:

```
{
“first_name” : “John”,
“last_name” : “Kanefield”,
“location” : “RiverSide”,
“websites” : [
{
“description” : “work”,
“URL” : “https://www.digitalriverside.com/"
},
{
“desciption” : “tutorials”,
“URL” : “https://www.digitalriverside.com/community/tutorials"
}
],
“social_media” : [
{
“description” : “twitter”,
“link” : “https://twitter.com/digitalriverside"
},
{
“description” : “facebook”,
“link” : “https://www.facebook.com/DigitalriversideCloudHosting"
```

```
},
{
“description” : “github”,
“link” : “https://github.com/digitalriverside"
}
]
}
```

Ключи “websites” и “social_media”, каждый используют массив для вложения информации о сайтах пользователя и профайлов в социальных сетях. Мы знаем, что это массивы из-за квадратных скобок.

## 8.4. JSON на практике в информационной системе

Есть большое количество вариантов использования JSON в современных программных продуктах. Рассмотрим один из наиболее встречающихся вариантов.

Например, возьмем мобильный сервис для «ВКонтакте», который позволяет прослушивать аудиокниги с возможностью покупки книги, а также оформления подписки на сервис. Он основан на клиентской части, сервере приложений и базе данных.

Данный сервис делает запросы через API Вконтакте для получения личной информации о пользователе, такие как: имя, фамилия, дата рождения, электронная почта, уникальный идентификатор пользователя, присваемый социальной сетью для формирования профиля, VKPay для оплаты подписки или покупки книги. Помимо этого, пользователи имеют возможность оставлять отзывы и оценки на книги, из этих оценок формируется рейтинг всех книг в сервисе. Также происходит взаимодействие с большим количество данных через REST API сервера приложений. Клиентская часть «общается» с базой данных, получая данные о книгах, авторах и жанрах. С помощью фильтрации и сортировки пользователь может быстро найти нужные ему книги.

Все описанные выше взаимодействия происходят с использованием JSON, на все запросы к различным API клиент получает ответ в формате JSON, примеры которого были описаны выше.

## 8.5. Контрольные задания

### 8.5.1. Вопросы для самопроверки

1. Что такое JSON?
2. Структура JSON.
3. Что такое вложенный JSON объект?
4. Что такое вложенный JSON массив?
5. Способы использования JSON формата.

# 9. Rabbit MQ

Система обмена сообщениями – это некий сервис, программа, предназначенная для передачи информации в реальном времени. Под информацией может предполагаться все что угодно, но как правило это текстовые рассылки.

Существуют 2 вида обмена: синхронный и асинхронный.

При синхронном обмене сообщениями отправитель и получатель дожидаются друг друга для передачи каждого сообщения, и операция отправки считается завершенной только после того, как получатель закончит прием сообщения.

При асинхронном обмене сообщениями не происходит никакой координации между отправителем и получателем сообщения. Для завершения операции отправки отправителю не требуется дожидаться приема сообщения процессом-получателем. При отправке нового сообщения, отправителю неизвестно, получено ли его предыдущее сообщение, направленное этому же или, возможно, другому получателю.

Поэтому, если канал связи между отправителем и получателем не сохраняет порядок передаваемых по нему сообщений, получатель может принимать сообщения в другом порядке, нежели они были переданы отправителем.

## 9.1. RabbitMQ

RabbitMQ – это сервис сообщений для организации приложений. Простыми словами – это программное обеспечение, в котором могут быть определены очереди, и к которому могут подключаться различные приложения и передавать/получать сообщения [13].

RabbitMQ реализует AMQP (Advanced Message Queueing Protocol). Данный протокол позволяет не задумываться над тем, где находятся получатели сообщений, сколько их, от кого надо ждать сообщение, когда оно будет доставлено получателю.

Преимущества:

- легок в использовании – он достаточно просто устанавливается, настраивается, имеет подробную документацию;
- работает на всех основных операционных системах;
- имеет клиентов для многих платформ разработки;
- открытый ресурс – репозиторий находится на github, он имеет открытую лицензию.

## 9.2. Взаимодействие объектов в протоколе AMQP

Взаимодействием объектов в протоколе AMQP изображено на рис. 9.1.

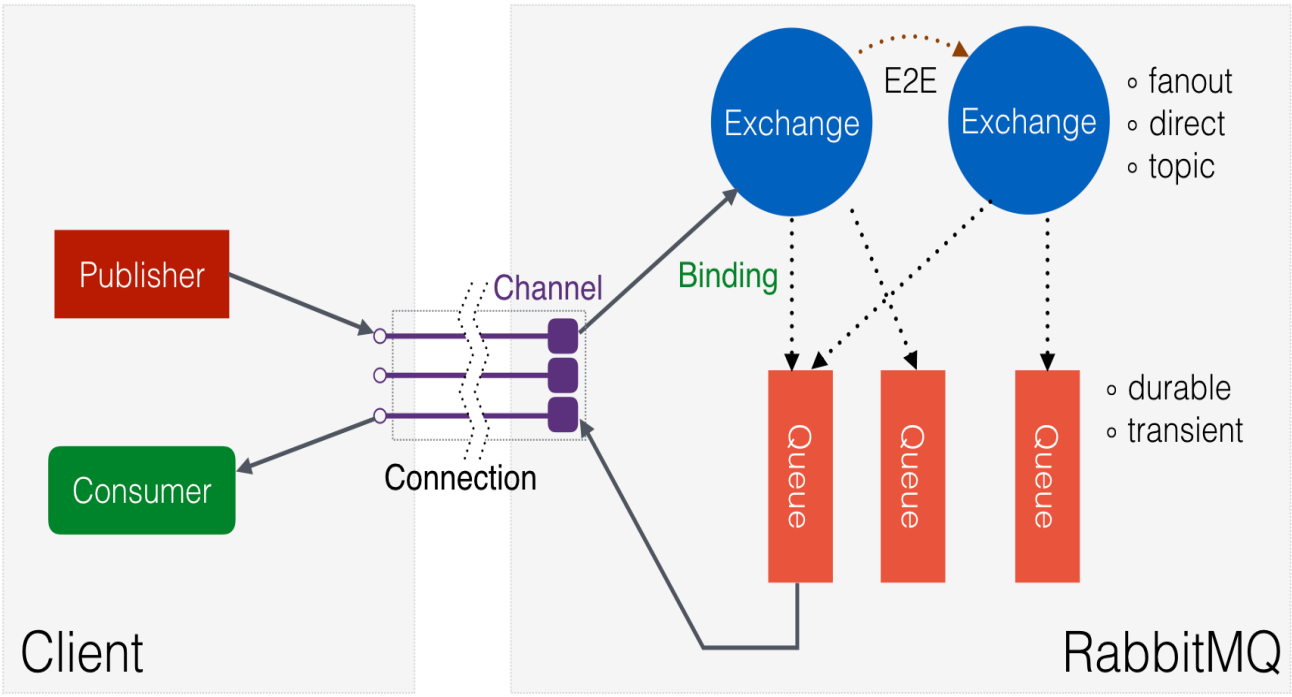



Рис. 9.1. Взаимодействие объектов

Есть клиентская и серверная сторона. В первую очередь клиент открывает AMQP соединение. Далее, чтобы иметь возможность выполнять какие-либо команды, ему необходимо создать хотя бы один канал. Каждый канал создает отдельный процесс, соответственно, если имеется какая-то интенсивная работа, то желательно распараллелить нагрузку на несколько каналов.

Далее рассмотрим, что такое обменники, очереди и их связи.

Очередь (queue) – представляет собой обычное хранилище, к которому могут подключаться потребители (consumers) и забирать из него сообщения.

Обменник (exchange) – это маршрутизатор сообщений, который не хранит в себе сообщения, а сразу доставляет их по очередям. Для того чтобы обменник понимал, в какие очереди нам нужно доставить сообщение, существуют привязки (binding) между ним и очередью. Таких связей может быть несколько и с разными параметрами. Один exchange можно привязать к нескольким очередям, также как и одну очередь можно привязать к нескольким обменникам.

В RabbitMQ есть расширение AMQP протокола «exchange two exchange binding», которое позволяет связать два обменника между собой, тем самым можно строить более сложную маршрутизацию сообщений.

Существуют четыре основных типа обменников: fanout, direct, topic, headers (о них чуть позже).

Также стоит отметить, что как обменники, так и очереди могут быть durable (т.е. постоянными, информация о них будет записана на диск и будет восстановлена после рестарта сервера), transient (временными, информация хранится в оперативной памяти и не будет сохранена после рестарта). Если создается связь между двумя durable-объектами, она также будет сохранена и восстановлена. Пример реализации обменников изображен на рис. 9.2.

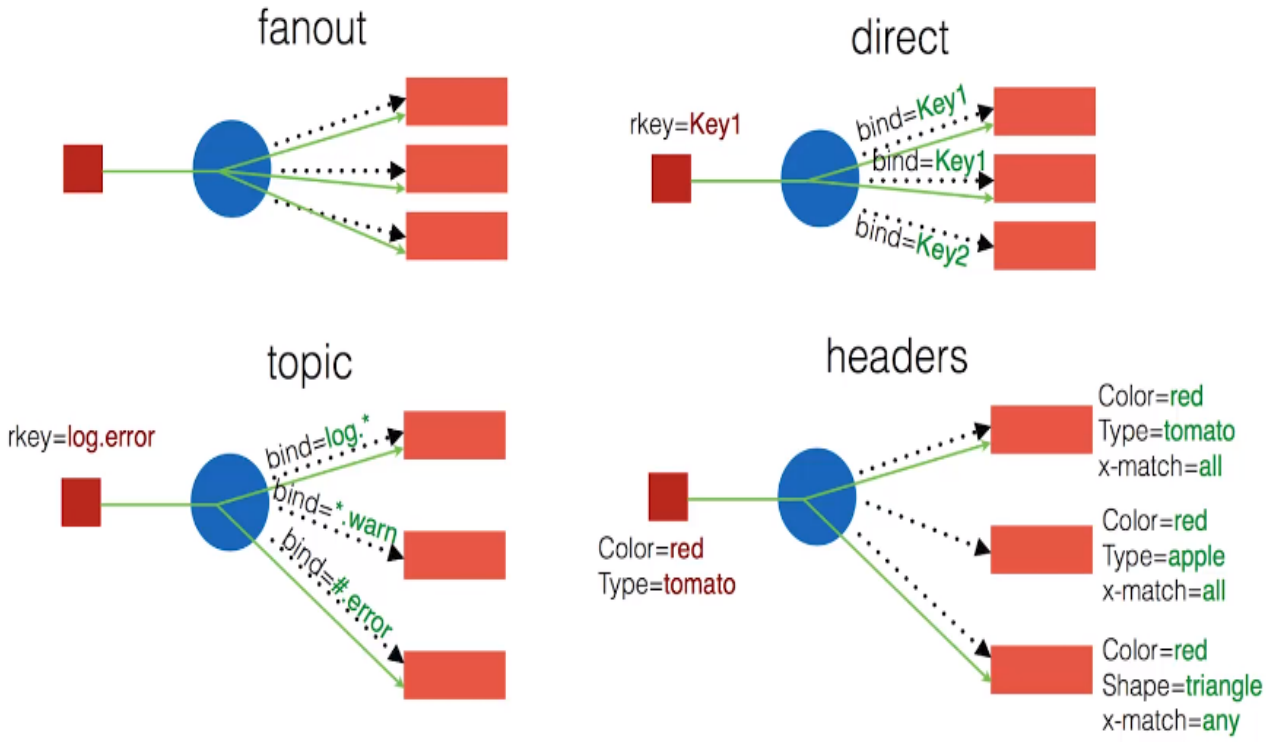



Рис. 9.2. Обменники

Как функционируют обменники? Рассмотрим их от самого простого к сложному.

Самый простой fanout. Связи не имеют параметров, а для того чтобы сообщение попало в очередь, достаточно просто наличие связи (сообщение попадает во все очереди). Этот обменник самый быстрый из всех существующих.

Далее идет direct. При создании связи здесь уже указывается строковый ключ, который используется при поиске кандидатов на получение сообщений. У нас публикуется сообщение, у сообщения есть ключ маршрутизации. Соответственно, если этот ключ будет равен ключу связи, то конечная очередь получит сообщение (здесь у нас получит сообщение первая и вторая очереди).

Следующий обменник – topic. Здесь ключ рассматривается не как строка, а уже как набор токенов, соединенных точкой. Мы можем в качестве токена использовать символ *, это означает, что нам подойдет любое его значение. Но 8 распространяется только на один токен. Если нам нужно сказать, что нам подойдут любые значения любых токенов, то мы можем использовать символ # (в данном примере у нас сообщение имеет 2 токена log и error, что соответствует 1 и 3 очередям) [14].

Последний обменник – headers. Обменники такого типа уже не смотрят на ключ-маршрутизации, а используют заголовки. (Сообщение наше содержит 2 заголовка color и type. При создании связей такого типа помимо заголовков указывается поле x-match, которое говорит, что нам нужно использовать оператор И, если указано ключевое слово all, или оператор ИЛИ, если указано any. 1 и 2 – требует совпадение всех параметров (1 подходит, 2 нет), 3 – требует совпадение хотя бы одного параметра (3 подходит)).

Панель управления RabbitMQ изображена на рис. 9.3.

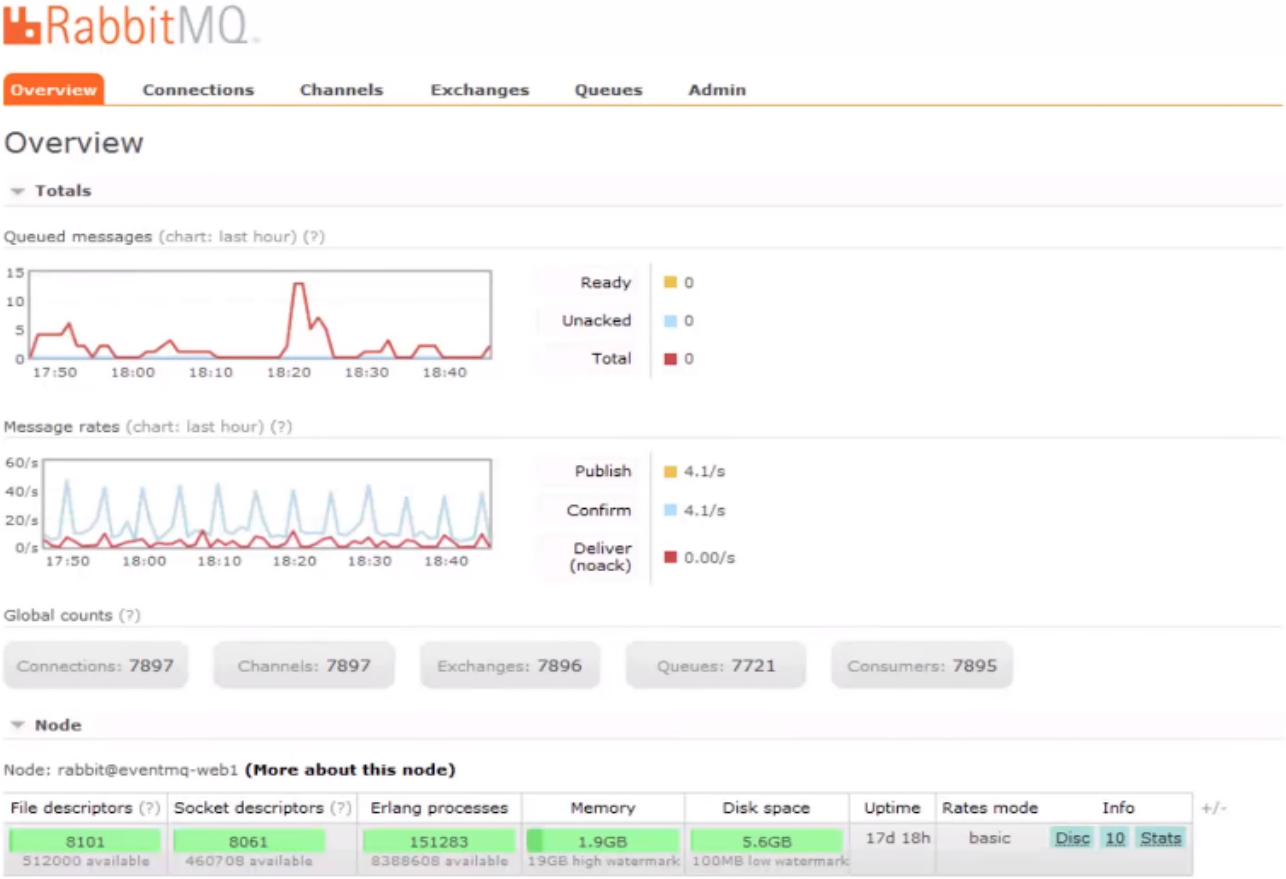



Рис. 9.3. Панель управления

Так выглядит панель управления RabbitMQ:

- меню для переключения режимов;
- общее количество сообщений в кластере;
- скорость поступления и потребления сообщений из кластера;

- статистика по подключениям, каналам, обменникам, очередям и потребителям;
- информация по всем кластерам.

Кластер – это сразу несколько сервисов, у которых общие пользователи, настройки и даже очереди.

## 9.3. Аналоги

Kafka – брокер, разработанный в рамках фонда Apache. Написан на Java. Спроектирован как распределённая, масштабируемая система, обеспечивающая наращивание пропускной способности как при росте числа и нагрузки со стороны источников, так и количества систем-подписчиков. Подписчики могут быть объединены в группы. Поддерживается возможность временного хранения данных для последующей пакетной обработки. Одной из особенностей является применение техники, сходной с журналами транзакций, используемыми в системах управления базами данных.

Apache ActiveMQ — это message broker, который полностью реализует Java Message Service. Он обеспечивает кластеризацию, хранение сообщений с возможностью использовать различные базы данных, кэширование и ведение журналов.

**RabbitMQ– написан на языке Erlang.

## 9.4. Пример использования RabbitMQ

Для примера разберем простейшее приложение на SpringBoot. Суть его заключается в следующем: после того как пользователь перейдет по определенной ссылке, в RabbitMQ будет посылаться сообщение, которое будет отправляться одному из слушателей. Слушатель же просто будет выводить сообщение в лог.

Итак, в первую очередь устанавливаем RabbitMQ. Далее разбираемся с конфигурацией SpringBoot. Открываем конфигурационный файл pom.xml и добавляем нужные нам для работы dependencies (springframework.amqp, springframework.boot).

AMQP – это протокол для передачи сообщений между компонентами системы.

Конфигурационный файл RabbitMQ изображен на рис. 9.4.

```
<groupId>com.cchgeu</groupId>
<artifactId>rabbitmq</artifactId>
<version>1.0-SNAPSHOT</version>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.4.RELEASE</version>
</parent>

<dependencies>

    <dependency>
        <groupId>org.springframework.amqp</groupId>
        <artifactId>spring-rabbit</artifactId>
        <version>1.4.5.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

</dependencies>
```

Рис. 9.4. Конфигурационный файл

Далее для работы с RabbitMQ потребуются следующие бины:

- ConnectionFactory – для соединения;
- AmqpAdmin – для автоматического объявления очередей, обменов и привязок;
- RabbitTemplate – для отправки сообщений;
- MyQueue – сама очередь.

Конфигурация для бинов изображена на рис. 9.5.

```java
@EnableRabbit
@Configuration
public class RabbitConfiguration {
    Logger logger = Logger.getLogger(RabbitConfiguration.class);

    @Bean
    public ConnectionFactory connectionFactory() {
        CachingConnectionFactory connectionFactory =
                new CachingConnectionFactory( hostname: "localhost");
        return connectionFactory;
    }

    @Bean
    public AmqpAdmin amqpAdmin() {
        RabbitAdmin rabbitAdmin = new RabbitAdmin(connectionFactory());
        rabbitAdmin.setAutoStartup(true);
        return rabbitAdmin;
    }

    @Bean
    public RabbitTemplate rabbitTemplate() {
        return new RabbitTemplate(connectionFactory());
    }

    @Bean
    public Queue myQueuel() { return new Queue( name: "query-example"); }
}
```

Рис. 9.5. Конфигурация для бинов

В данном примере рассматривается одна очередь и 2 слушателя. Наглядно представлено это следующим образом на рис. 9.6.

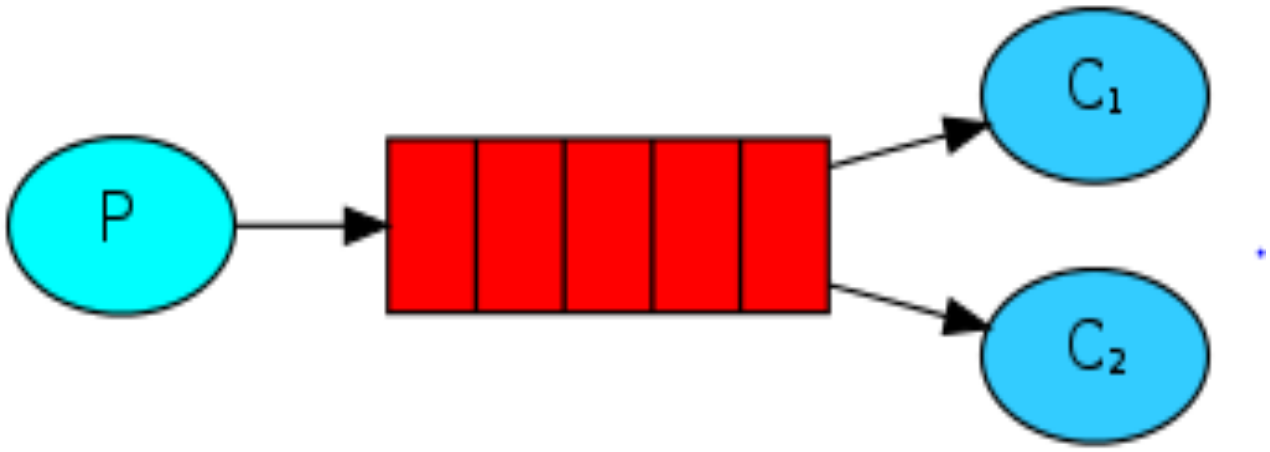



Рис. 9.6. Работа очереди

Пример программной реализации роли слушателя в RabbitMQ изображен на рис. 9.7.

```
@Component
public class RabbitMqListener {
    Logger logger = Logger.getLogger(RabbitMqListener.class);
    Random random = new Random();

    @RabbitListener(queues = "query-example")
    public void worker1(String message) throws InterruptedException {
        logger.info("worker 1 : " + message);
        Thread.sleep( millis: 100 * random.nextInt( bound: 20));
    }

    @RabbitListener(queues = "query-example")
    public void worker2(String message) throws InterruptedException {
        logger.info("worker 2 : " + message);
        Thread.sleep( millis: 100 * random.nextInt( bound: 20));
    }
}
```

Рис. 9.7. Роль слушателя

Для эмуляции работы используется Thread.sleep.

В приложении роль продюсера исполняет контроллер и он изображен на рис. 9.8. Этот контроллер как раз и будет посылать сообщение в RabbitMQ.

```
@Controller
public class SampleController {
    Logger logger = Logger.getLogger(SampleController.class);

    @Autowired
    AmqpTemplate template;

    @RequestMapping("/emit")
    @ResponseBody
    String queue1() {
        logger.info("Emit to queue");
        for (int i = 0; i < 10; i++)
            template.convertAndSend( s: "query-example",  o: "Message " + i);
        return "Emit to queue";
    }
}
```

Рис. 9.8. Роль продюсера

Все готово. Остается запустить приложение и перейти в браузере по адресу http://localhost:8080/emit. Если все сделано правильно, то в консоли можно наблюдать результат, изображенный на рис. 9.9.

```
2019-05-01 17:24:42.541  INFO 7472 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2019-05-01 17:24:42.542  INFO 7472 --- [           main] com.cchgeu.rabbit.Configuration          : Started Configuration in 3.049 seconds (JVM running for 3.52)
2019-05-01 17:24:50.160  INFO 7472 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherServlet'
2019-05-01 17:24:50.160  INFO 7472 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization started
2019-05-01 17:24:50.175  INFO 7472 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in 15 ms
2019-05-01 17:24:50.190  INFO 7472 --- [nio-8080-exec-1] com.cchgeu.rabbit.SampleController       : Emit to queue
2019-05-01 17:24:50.249  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 1 : Message 0
2019-05-01 17:24:50.249  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 2 : Message 1
2019-05-01 17:24:50.361  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 2 : Message 2
2019-05-01 17:24:50.651  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 1 : Message 3
2019-05-01 17:24:50.870  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 2 : Message 4
2019-05-01 17:24:51.752  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 1 : Message 5
2019-05-01 17:24:51.871  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 2 : Message 6
2019-05-01 17:24:51.954  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 1 : Message 7
2019-05-01 17:24:52.184  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 2 : Message 8
2019-05-01 17:24:53.185  INFO 7472 --- [cTaskExecutor-1] com.cchgeu.rabbit.RabbitMqListener       : worker 2 : Message 9
```

Рис. 9.9. Результат работы RabbitMQ

Масштабируемость является одной из основных проблем нашего времени, а обмен сообщениями является неотъемлемой частью решения. Наконец, все сводится к программному обеспечению брокера сообщений для управления и контроля обмена сообщениями между приложениями, процессами и потоками. Посредники сообщений могут помочь решить проблемы масштабируемости и архитектурные проблемы, такие как связывание.

RabbitMQ - одно из самых мощных программных брокеров с открытым исходным кодом, которое широко используется в таких технологических компаниях, как Mozilla, VMware, Google, AT & T и так далее. RabbitMQ - это настраиваемая платформа для обмена сообщениями, разработанная и поддерживаемая хорошо осведомленным и преданным сообществом.

Брокеры сообщений и очередь сообщений

В последнее время программные системы эволюционировали кардинально. Приложения должны взаимодействовать с другими приложениями, эти приложения могут быть внутренними и внешними по отношению к самому приложению. Для одного и того же приложения у нас могут быть клиенты другого типа, такие как браузеры, мобильные клиенты и так далее. Следовательно, нам абсолютно необходим уровень связи между внутренними приложениями и между приложениями и клиентами. Нам нужно доставлять разные сообщения разным приложениям или клиентам. Доставка сообщений может быть узким местом, если коммуникационный уровень не масштабируется. Преследование масштабируемых систем для коммуникационного уровня приводит нас к брокерам сообщений и очередям сообщений. Давайте теперь обсудим, что такое брокеры сообщений и очереди сообщений.

Message Broker - это архитектурный шаблон, который может принимать сообщения от нескольких адресатов, определять правильный адресат и направлять сообщение по правильному маршруту.

Посредники сообщений позволяют системам иметь дело с обменом сообщениями и маршрутизацией, обеспечивая взаимодействие между компонентами. Как только приложения реализуют шаблон посредника сообщений, это уменьшает связь между компонентами приложения.

Посредники сообщений централизованы в архитектурном смысле для контроля и управления всеми сообщениями. Следовательно, все входящие и исходящие сообщения отправляются через Message Brokers, которые анализируют и доставляют сообщения по назначению.

Посредники сообщений решают следующие проблемы на уровне коммуникации:

- преобразование сообщений в альтернативные форматы;
- маршрутизация сообщений в пункты назначения;
- поддержка различных типов шаблонов для отправки сообщений;
- получение и реагирование на события;
- выполнение агрегации сообщений;
- сохранение состояния сообщения;
- обеспечение получения и отправки сообщения;
- разъединение целевых программных систем.

Многие задачи брокера сообщений нуждаются в очереди сообщений для обмена или передачи данных в пункт назначения.

Очередь сообщений

Очередь - это основная структура данных, стоящая за функционированием очереди сообщений. Операции очереди сообщений аналогичны операциям структуры данных очереди, таким как операции enqueue и dequeu. Операция постановки в очередь приводит к добавлению элемента в конец очереди. Операция удаления очереди приводит к удалению элемента с начала очереди.

Очереди сообщений обеспечивают одновременные и асинхронные операции для масштабирования приложений. В очереди сообщений сообщения ожидают, пока приложение не получит сообщение.

Различные типы стандартов и протоколов определяют спецификации очереди сообщений. Некоторые протоколы открыты для всех; однако некоторые протоколы закрыты. RabbitMQ использует расширенный протокол очереди сообщений (AMQP), который определяет политики очередей сообщений.

RabbitMQ имеет следующие функциональные возможности для решения проблем обмена сообщениями:

- гарантирует, что сообщения отправлены и получены;

- направляет сообщения в правильные места назначения;
- сохраняет состояние сообщений;
- поддерживает несколько транспортных протоколов (AMQP, MQTT, STOMP, HTTP);
- поддерживает кластеризацию;
- высоко масштабируемый, надежный и доступный;
- расширяемый с помощью плагинов;
- поддерживает клиентов практически на любом языке;
- поддержка большого сообщества также обеспечивает коммерческую поддержку.

## 9.5. Установка RabbitMQ

Установка RabbitMQ ничем не отличается от других программ в разных операционных системах. Операционные системы на основе Unix могут создавать RabbitMQ из исходного кода, а Microsoft Windows может запускать стандартные установщики MSI. Установочные файлы RabbitMQ можно найти на веб-странице загрузки веб-сайта RabbitMQ.

Единственной предпосылкой для установки RabbitMQ является среда исполнения Erlang, поскольку RabbitMQ работает на виртуальной машине Erlang. Поэтому мы должны установить Erlang перед установкой RabbitMQ. Erlang можно загрузить с веб-страницы загрузки Erlang.

RabbitMQ работает как на 32-разрядных, так и на 64-разрядных компьютерах из одного пакета. Erlang устанавливается как 32-битный, так и 64-битный. Таким образом, RabbitMQ может быть легко установлен в операционной системе Windows. Давайте установим эти вещи для запуска RabbitMQ.

Во-первых, мы должны установить среду исполнения Erlang в Windows. Erlang имеет установщики Windows для 32-разрядных и 64-разрядных систем.

Мы можем легко загрузить соответствующий бинарный файл на наш компьютер и установить Erlang, используя его.

После установки среды исполнения Erlang мы выполнили требования установки RabbitMQ. Следующим шагом является загрузка и установка бинарного файла RabbitMQ для соответствующей версии Windows.

Мы можем найти соответствующий установщик Windows для RabbitMQ с помощью веб-страницы загрузки RabbitMQ, затем нам нужно просто щелкнуть и установить RabbitMQ на наш компьютер с Windows. Помимо установки с помощью программы установки, мы можем установить с помощью бинарного файла Windows, который находится на веб-странице загрузки RabbitMQ. Следующих инструкций будет достаточно для установки RabbitMQ без установщика:

- скачать бинарный файл для двоичных файлов RabbitMQ Windows;
- извлечь загруженный ZIP-файл RabbitMQ в нашу локальную папку.

Установить RabbitMQ на компьютер с Windows можно обоими способами. Обратите внимание, что вы можете добавить каталог бинарных файлов RabbitMQ в системный путь Windows в настройках переменных system/environment.

Установка RabbitMQ довольно проста, и запуск RabbitMQ похож на его установку. Некоторые менеджеры пакетов в Linux, Mac OS X и установщике Windows добавляют параметры конфигурации в конфигурацию операционной системы для автоматического запуска. В таком случае нам не нужно запускать команду RabbitMQ вручную; однако, если мы устанавливаем RabbitMQ вручную, нам нужно запускать команды RabbitMQ вручную.

Если мы используем установщик Windows RabbitMQ, установщик уже создает конфигурации для автоматического запуска. Поэтому нам не нужно запускать RabbitMQ вручную; однако всякий раз, когда мы хотим контролировать состояние сервера, нам просто нужно выполнить следующую команду в папке sbin RabbitMQ:

```
rabbitmqctl status
```

Если вы установили RabbitMQ вручную на Windows, то должны запустить следующую команду, чтобы запустить RabbitMQ (вы должны запустить эту команду с правами администратора). Кроме того, вы можете установить сервер RabbitMQ в качестве службы Windows:

```
rabbitmq-server
```

**Конфигурация RabbitMQ:**

Конфигурация является одной из важнейших частей для администрирования RabbitMQ. Благодаря отличной конфигурации RabbitMQ может эффективно отправлять и получать сообщения между приложениями, процессами и потоками.

Существует три способа настройки RabbitMQ:

- использовать переменные среды RabbitMQ, которые располагаются в переменных среды операционной системы;
- через файл конфигурации, предоставленный RabbitMQ;
- использовать параметры времени выполнения. Такое разнообразие конфигурации дает полный контроль над RabbitMQ на стороне сервера и операционной системы.

Настройку можно разделить на:

- общая конфигурация RabbitMQ;
- переменные среды RabbitMQ;
- файл конфигурации;
- параметры времени выполнения.

Общая конфигурация RabbitMQ

Переменные среды RabbitMQ - это один из способов конфигурации RabbitMQ. Каждая операционная система имеет свой собственный набор переменных среды для каждого пользователя. Хотя операционные системы

имеют возможность иметь переменные среды, способ изменения переменных среды несколько отличается в разных операционных системах.

В Windows мы должны использовать переменные среды System Properties для изменения переменных среды RabbitMQ. Мы можем получить доступ к переменным среды, перейдя в Настройки | Панель управления | Системные свойства | Расширенный | Переменные среды, в которых мы используем каналы для отображения переходов. Теперь мы получаем доступ к переменным окружения. Хотя RabbitMQ предоставляет нам множество различных переменных среды, мы рассмотрим наиболее важные из них.

1. Конфигурирование при помощи переменных среды.

RabbitMQ предоставляет нам множество отличных переменных среды для управления всеми частями своего движка.

RABBITMQ_BASE: эта переменная в основном находит каталог RabbitMQ. В этом каталоге есть база данных и файлы журналов.

RABBITMQ_CONFIG_FILE: Хотя файл конфигурации RabbitMQ имеет местоположение по умолчанию, вы можете изменить его местоположение с помощью этой переменной среды.

RABBITMQ_LOGS: RabbitMQ поддерживает разные уровни журналов. Всякий раз, когда RabbitMQ создает файл журнала, он имеет местоположение по умолчанию; однако вы можете изменить его местоположение с помощью этой переменной среды.

RABBITMQ_NODE_IP_ADDRESS: RabbitMQ связывается со всеми сетевыми интерфейсами как свойство по умолчанию. Поскольку RabbitMQ дает нам полный контроль над сетевыми интерфейсами, мы можем легко изменить его связывающую сеть, используя эту переменную, такую как 127.0.0.1.

RABBITMQ_NODE_PORT: RabbitMQ имеет порт по умолчанию 5672; однако иногда мы сталкиваемся с портами, поэтому мы должны изменить порты, которые связывает RabbitMQ. Мы можем изменить порт привязки RabbitMQ, используя эту переменную.

RABBITMQ_PLUGINS_DIR: RabbitMQ имеет много очень полезных плагинов, которые будут включены через RabbitMQ. RabbitMQ имеет местоположение по умолчанию для этих плагинов с кодировкой Erlang; однако вы можете изменить его местоположение.

ABBITMQ_SASL_LOGS: это расположение файлов журналов библиотек поддержки системных приложений сервера RabbitMQ.

RABBITMQ_SERVER_START_ARGS: параметры Erlang используются для команды erl при вызове сервера RabbitMQ. Эта переменная не будет переопределять RABBITMQ_SERVER_ERL_ARGS.

Специфичное для Windows расположение по умолчанию

Значения по умолчанию переменных среды RabbitMQ связаны с другими переменными среды RabbitMQ. В табл. 9.1 показано расположение Windows по умолчанию.

Таблица 9.1

Расположение по-умолчанию

| Имя переменной | Расположение |
|---|---|
| RABBITMQ_BASE | %APPDATA%\RabbitMQ |
| RABBITMQ_CONFIG_FILE | %RABBITMQ_BASE%\rabbitmq |
| RABBITMQ_LOGS | %RABBITMQ_LOG_BASE%\%RABBITMQ_NODENAME%.log |
| RABBITMQ_LOG_BASE | %RABBITMQ_LOG_BASE%\log |
| RABBITMQ_MNESIA_BASE | %RABBITMQ_BASE%\db |
| RABBITMQ_MNESIA_DIR | %RABBITMQ_MNESIA_BASE%\%RABBITMQ_NODENAME% |
| RABBITMQ_PLUGINS_DIR | %RABBITMQ_BASE%\plugins |
| RABBITMQ_SASL_LOGS | %RABBITMQ_LOG_BASE%\%RABBITMQ_NODENAME%-sasl.log |

Переменные среды RabbitMQ сильно зависят от переменных среды операционной системы. Например, имя компьютера в Unix и имя хоста в Windows устанавливают переменные среды RABBITMQ_SERVICENAME и RABBITMQ_NODENAME.

2. Конфигурирование при помощи конфигурационного файла.

В Windows по умолчанию вы можете найти файл конфигурации в следующей папке:

| C:\Program Files (x86)\RabbitMQ_Server\etc\rabbitmq.config |
|---|

Переменные среды RabbitMQ в основном предоставляют контроль расположения файлов и каталогов, в то время как файл конфигурации RabbitMQ предоставляет управление механизмом, таким как аутентификация, производительность, ограничение памяти, ограничение диска, обмены, очереди, привязки и так далее.

RabbitMQ имеет много конфигурационных переменных. Ниже представлены несколько из них:

- auth_mechanisms - используется для поддержки различных типов механизмов аутентификации. Вы можете изменить другой тип механизма аутентификации, используя эту переменную;
- default_user - используется как пользователь по умолчанию для доступа к серверу RabbitMQ с помощью клиента RabbitMQ. Переменная default_user просто определяет имя пользователя по умолчанию;
- default_pass - похожа на переменную default_user, поскольку она просто определяет пароль пользователя по умолчанию;

- default_permission - похожа на default_user и default_pass. Эта переменная описывает права пользователя по умолчанию;
- disk_free_limit - используется для управления размером диска для хранения сообщений на диске. Эта переменная определяет размер свободного диска для оповещения администратора сервера RabbitMQ;
- Heartbeat - переменная конфигурации, которая указывает временной интервал между ударами. Удар - это пакет, отправляемый от посредника клиенту и обратно, чтобы посредник мог понять, подключен ли клиент по-прежнему или нет, и держать линию открытой, где какое-то сетевое оборудование может отключить его из-за неактивности. В других протоколах, таких как старый и проверенный временем Internet Relay Chat (IRC), этот прием также назывался пинг-понг;
- hipe_compile - как свойство по умолчанию, RabbitMQ компилируется с компилятором Erlang по умолчанию; однако мы можем скомпилировать с помощью высокопроизводительного компилятора Erlang. Сервер RabbitMQ компилируется при запуске. Результаты компиляции хипа при более позднем запуске; однако Hipe Compile обеспечивает увеличение производительности на 20% - 40% при работе с брокерами сообщений. С помощью переменной hipe_compile мы можем контролировать, будет ли RabbitMQ компилироваться с помощью высокопроизводительного компилятора Erlang или нет;
- log_levels - у нас есть журналы для контроля и отслеживания приложения для каждого из приложений. Журналы имеют разные уровни, чтобы показать сообщения журнала в соответствии с его уровнем журнала, то есть ошибки, предупреждения и информацию. С помощью этой переменной вы можете выбрать уровень ведения журнала;
- tcp_listeners - это то же самое, что серверные приложения, такие как FTP-сервер, почтовый сервер и т. д. Сервер RabbitMQ связывается с IP-адресом и портом операционной системы. Вы можете изменить его порт привязки и IP с помощью переменной tcp_listeners;
- ssl_listeners - всякий раз, когда клиенты прослушивают сервер с помощью SSL, сервер RabbitMQ использует другой IP-адрес и порт. Переменная ssl_listeners просто определяет IP и порт клиентских соединений SSL;
- vm_memory_high_watermark - размер свободной памяти достаточно важен для сервера RabbitMQ. RabbitMQ предупреждает о проблеме с памятью с помощью заданного коэффициента свободной памяти в vm_memory_high_watermark.

3. Runtime параметры..

RabbitMQ предоставляет переменные среды и переменные конфигурации для настройки RabbitMQ при запуске сервера RabbitMQ. В дополнение к этим конфигурациям RabbitMQ позволяет нам изменять параметры, которые были

установлены в переменных среды и переменных конфигурации во время выполнения, используя параметры времени выполнения.

Мы можем использовать инструмент командной строки для управления брокером RabbitMQ для изменения параметров во время работы.

Управление параметрами

Управление параметрами - это способ настройки RabbitMQ путем установки значений параметров. Мы можем изменить параметры, используя команду set_parameter в rabbitmqctl. Более того, мы можем изменить различные типы компонентов RabbitMQ с помощью данного атрибута component_name.

Управление политикой

Управление политикой - это настройка значений политики RabbitMQ. RabbitMQ дает нам возможность изменять свои политики для очередей сообщений во время выполнения, а его политики применимы для обмена и очередей. Вы можете установить новые политики, используя «set_policy», тогда как вы можете очистить все политики, используя «clear_policy».

Управление памятью

Управление памятью - это конфигурация значений памяти RabbitMQ. Управление памятью может осуществляться через параметры файла конфигурации RabbitMQ. Но иногда нам приходится менять порог памяти на более низкое значение для многих клиентов. RabbitMQ дает возможность изменить порог памяти, используя параметр во времени выполнения set_vm_memory_high_watermark.

## 9.6. Архитектура и обмен сообщениями

Сервер RabbitMQ решает проблемы обмена сообщениями. Но в чем смысл сообщений? Иногда термин «обмен сообщениями» путают с сообщениями в режиме реального времени, такими как сообщения чата, сообщения SMS и т. д. Эти системы также имеют систему обмена сообщениями в своих подсистемах. Однако речь идет о несколько иной проблеме.

Обмен сообщениями в данном контексте означает, что сервер просто принимает сообщения от производителя и отправляет их потребителям с помощью компьютерной техники. В системах обмена сообщениями мы используем некоторую архитектуру, связанную с обменом сообщениями и элементами. Кроме того, у нас есть различные функциональные возможности элементов в системе обмена сообщениями. Следующий список характеризует основные функции приложения:

- обмен сообщениями и варианты использования сообщений;
- корпоративные сообщения;
- архитектура программного обеспечения для обмена сообщениями;
- концепции обмена сообщениями;
- расширенный протокол очереди сообщений (AMQP).

Обмен сообщениями и варианты его использования

Обмен сообщениями просто определяется как связь между производителем сообщения и потребителем сообщения. Посредник сообщений определяется как модуль, управляющий потоком сообщений. Управлять действиями не так просто, поэтому для реализации этой функции обмена сообщениями брокерам сообщений необходимо много навыков.

Прежде чем говорить о функциях Message Broker, нам нужно знать о проблемах, которые возникают у нас с сообщениями. Проблемы меняются по отношению к области программного обеспечения системы; однако большинство проблем одинаковы в разных типах функций обмена сообщениями программных систем. Давайте перечислим все распространенные проблемы обмена сообщениями и способы их решения с помощью брокеров сообщений.

Сцепление программных систем - в настоящее время связь обычно называется выражением зависимости между двумя модулями по отношению друг к другу. Связывание или, точнее, запутывание кода, плохо, потому что усложняет обслуживание программного обеспечения. Любое изменение в зависимом компоненте может привести к изменениям, ошибкам, обновлениям версии и так далее. Тесная связь может быть на уровне кода и на уровне сервиса/архитектуры. Решения существуют как для кода, так и для архитектурной связи. Связь на уровне кода может быть решена с помощью внедрения зависимости. Архитектурная связь может быть решена с помощью брокеров сообщений. Нам нужно создать абстракцию между модулями для решения проблем с сообщениями. На следующем снимке экрана показаны соединенные модули обмена сообщениями, которые взаимодействуют с другими модулями без использования Message Broker. Всякий раз, когда вы изменяете один метод модуля, вы должны распространить это изменение на другие модули.

Message Broker - это отличное решение для хорошо известной проблемы программных систем с высокой связью, которые передаются между модулями. Message Broker создает абстракцию между модулями, поэтому функции обмена сообщениями контролируются и управляются самим Message Broker. Модули не знают об отправке или получении сообщений; они только отправляют свои сообщения нужному получателю через Message Broker. Message Broker направляет эти сообщения в нужный модуль и преобразует их в соответствующий формат обмена сообщениями.

Как следствие, Message Broker определяется как промежуточное программное обеспечение для обмена сообщениями, которое просто выполняет переход модулей обмена сообщениями с высокой связью на модули обмена сообщениями с низкой связью, создавая промежуточный уровень между модулями.

Множество удобных платформ – в настоящее время все программные системы используют разные типы технологий, такие как Java Platform, .Net

Framework и так далее. Кроме того, мобильность и Интернет дает нам возможность добавлять новых клиентов в наши программные системы. Поэтому мы должны объединить эти технологические стеки в архитектуре программного обеспечения и соединить их друг с другом; например, подключение Java Platform к .Net Framework. Тогда у нас есть еще одна проблема в нашей программной системе.

Так как сообщение не имеет привязки к конкретному языку или к конкретномой платформе, то Message Broker является решением этой проблемы.

Масштабируемость - современная проблема разработки программного обеспечения. Многие научные статьи были направлены на решение этой известной проблемы. Масштабируемость не является простой проблемой и не имеет единого решения. Хотя масштабируемость отличается от финансовых программных систем до веб-приложений реального времени, у них есть похожая проблема: связь между модулями или процессами. Затем речь идет о Message Brokers, который решает эту проблему масштабируемости.

**Корпоративные сообщения**

Корпоративные приложения, такие как приложения для управления взаимоотношениями с клиентами, приложения для бизнес-аналитики, приложения для управления проектами, приложения для управления персоналом и так далее, должны быть интегрированы с другими корпоративными приложениями. Более того, мы должны гарантировать, что все системы должны быть масштабируемыми.

Посредники сообщений в основном пытаются решить подобные проблемы в корпоративных приложениях. В корпоративном обмене сообщениями мы должны гарантировать, что сообщение отправлено и получено, поскольку каждое из сообщений очень важно для надежности нашей системы. Брокеры сообщений имеют функцию постоянного хранения всех сообщений для удовлетворения такого рода требований. В результате корпоративные сообщения не сильно отличаются от других типов приложений обмена сообщениями. Таким образом, мы можем решать проблемы с корпоративными сообщениями, используя Message Brokers.

## 9.7. RabbitMQ и программные продукты

У RabbitMQ есть клиенты для Java, Python, C#, Ruby и так далее. Используя эти клиенты, приложения для обмена сообщениями могут быть реализованы самым простым способом.

Сообщество RabbitMQ и его главная компания-спонсор Pivotal предоставляют официальную клиентскую библиотеку для Java под названием RabbitMQ Java Client. Клиентская библиотека обеспечивает как публикацию сообщений, так и получение сообщений. Более того, клиентская библиотека поддерживает как синхронный, так и асинхронный прием. Детали будут объяснены в следующих темах.

Если мы посмотрим на основные пакеты Java-клиента RabbitMQ, то увидим три пакета:

- com.rabbitmq.client предоставляет классы и интерфейсы для AMQP-соединений, каналов и описания фреймов проводного протокола;
- com.rabbitmq.tools предоставляет классы и методы для неосновных утилит и инструментов администрирования;
- com.rabbitmq.utility предоставляет вспомогательные классы, которые в основном используются при реализации библиотеки.

Вы можете найти каждый элемент AMQP в пакете клиента, среди них Connection, Channel, Exchange, Queue.

Connection – это интерфейс в пакете клиента. Интерфейс подключения напрямую относится к элементу подключения AMQP. Таким образом, интерфейс Connection также охватывает функции элемента Connection AMQP. Мы можем создать экземпляр Connection через класс ConnectionFactory, как показано в следующем коде:

```
// создаем ConnectionFactory
ConnectionFactory connectionFactory = new ConnectionFactory();
// указываем хост
connectionFactory.setHost("127.0.0.1");
//указываем пользователя
connectionFactory.setUsername("admin");
// указываем пароль
connectionFactory.setPassword("password");
// Создаем объект Connection, с помощью которого будет осуществляться взаимодействие
Connection connection = connectionFactory.newConnection();
```

Класс ConnectionFactory имеет атрибуты, которые относятся к имени хоста, порту, имени пользователя, паролю и виртуальному хосту. Мы можем установить каждый из обязательных атрибутов, и тогда мы готовы создать наше соединение. Кроме того, мы можем установить каждый атрибут, используя стандарты URI, следующим образом:

```
// создаем ConnectionFactory
ConnectionFactory connectionFactory = new ConnectionFactory();
// указываем параметры в формате URL
connectionFactory.setUri("amqp://admin:password@127.0.0.1")
// Создаем объект Connection, с помощью которого будет осуществляться взаимодействие
Connection connection = connectionFactory.newConnection();
```

После завершения работы с подключением его обычно закрывают. Для этого предусмотрен метод close();

```
// Закрываем подключение
connection.close();
```

Канал - это еще один интерфейс в пакете клиента. Поскольку интерфейс подключения относится к элементу подключения AMQP, канал относится к элементу канала AMQP. Основная роль канала - служить логическим соединением внутри сетевого соединения с брокером сообщений. Экземпляры канала являются потокобезопасными.

Экземпляр Channel может быть инициализирован через экземпляр Connection, как показано в следующем примере кода:

```
// Создаем connection
Connection connection = connectionFactory.newConection();
//Создаем в connection канал
Channel ch = connection.createChannel();
```

Канал отвечает за отправку сообщения, получение сообщения, выполнения операции с очередями и многое другое. Канал не будет доступен, если эти операции не пройдены.

Exchange - основной элемент AMQP, который модерируют очереди с заданными функциями. Обмены также доступны в API клиента Java RabbitMQ. Основной обязанностью является получение сообщений от производителей и отправка их в соответствующие очереди, которые выражены правилами. Хотя они имеют такое значение в AMQP 0.9.1, они не существуют в спецификации AMQP 1.0.

```
// Создаем connection
Connection connection = connectionFactory.newConection();
// Создаем в connection канал
Channel ch = connection.createChannel();
// Создаем в канале Exchange
ch.exchangeDeclare("mastering.rabbitmq","fanout");
```

Queue – очереди. Очереди являются наиболее важной частью Message Brokers и AMQP. Всякий раз, когда новый потребитель или подписчик сообщений подключается к Exchange, RabbitMQ создает очередь для соответствующего обмена с указанным именем.

Каналы отвечают за общие операции с очередями. Следовательно, их можно объявлять, связывать, отменять привязку, очищать и удалять очереди с помощью методов Channels, как предусмотрено в Java API RabbitMQ. Следующий простой пример кода показывает, как очереди привязаны к данным обменам:

```
// Создаем в канале Exchange
ch.exchangeDeclare("mastering.rabbitmq","fanout");
// Создаем очередь в канале и получаем ее имя
String name = ch.queueDeclare().getQueue()
// Привязываем очередь к обмену. В качестве ключа маршрутизации укажем пустую строку
ch.queueBind(name, "mastering.rabbitmq","");
```

Описанных выше методов достаточно, чтобы уже отправить сообщение. Пример отправки представлен ниже:

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;

public class MessageSender {

  private final static String NAME ="mastering.rabbitmq";

  public static void main(String[] argv)throws Exception {
    ConnectionFactory connectionFactory =new ConnectionFactory();
    connectionFactory.setHost("127:0:0:1");
    Connection connection = factory.newConnection();
    Channel ch = connection.createChannel();
    ch.queueDeclare(NAME,false,false,false,null);
    String message ="Это мое первое сообщение в RabbitMQ";
    ch.basicPublish("",NAME,null,message.getBytes());
    System.out.println("Мы отправили в RabbitMQ сообщение: "+ message);

    ch.close();
    connection.close();
  }
}
```

Мы смогли отправить сообщение и готовы перейти к тому, как получаем сообщение, которое доставляется из подключенной очереди.

Использование сообщений из RabbitMQ аналогично, но не идентично публикации. Во-первых, мы инициализируем соединение через экземпляр ConnectionFactory и объявляем очередь, связанную с нашим получателем и отправителем. Затем приходит ответ от отправителя, то есть часть получаемого сообщения. Принимающая часть может быть реализована синхронным или асинхронным способом. Синхронно мы блокируем текущий поток для прослушивания доставки сообщений; асинхронным способом поток не может быть заблокирован, поэтому всякий раз, когда доставляется сообщение, потребительский метод вызывается мгновенно.

Пример синхронизированного получателя сообщений показан ниже:

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.QueueingConsumer;

public class MessageReciever {
```

```
  private final static String NAME="mastering.rabbitmq";

  public static void main(String[]argv) throws Exception {
    ConnectionFactory connectionFactory=new ConnectionFactory();
    connectionFactory.setHost("127.0.0.1");
    Connection conn= connectionFactory.newConnection();
    Channel ch = conn.createChannel();
    ch.queueDeclare(NAME, false, false, false, null);
    QueueingConsumer cons=new QueueingConsumer(ch);
    ch.basicConsume(NAME, true, cons);

    while(true) {
      QueueingConsumer.Delivery del=consumer.nextDelivery();
      String message=newString(del.getBody());
      System.out.println("Received Message:"+ message);
    }
  }
}
```

Пример асинхронного получения сообщения:

```
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.QueueingConsumer;

public class MessageReciever {

  private final static String NAME="mastering.rabbitmq";

  public static void main(String[] argv) throws Exception {
    ConnectionFactory connectionFactory=new ConnectionFactory();
    connectionFactory.setHost("127.0.0.1");
    Connection conn = connectionFactory.newConnection();
    Channel ch = conn.createChannel();
    ch.queueDeclare(NAME, false, false, false, null);

    ch.basicConsume(NAME, false, new DefaultConsumer(channel) {
      @Override
      public void handleDelivery(String tag, Envelope env,
      AMQP.BasicProperties props, byte[] message) throws IOException {
        String messageStr = new String(message);
        System.out.println("Получено сообщение: " + messageStr);
      }
    });
  }
}
```

### 9. 8. Контрольные задания

#### 9.8.1. Вопросы для самопроверки

1. Что такое Rabbit MQ?
2. Взаимодействие объектов в Rabbit MQ протоколом AMQP.
3. Аналоги Rabbit MQ.
4. Способы настройки Rabbit MQ.
5. Что такое Message Broker?
6. Какую проблему решает сервер Rabbit MQ?
7. У каких языков программирования есть поддержка Rabbit MQ?

## 10. МИКРОСЕРВИСНАЯ АРХИТЕКТУРА

Микросервисная архитектура - вариант сервис-ориентированной архитектуры программного обеспечения, направленный на взаимодействие нескольких, возможно небольших, слабо связанных и легко изменяемых модулей — микросервисов, получивший распространение в середине 2010-х годов в связи с развитием практик гибкой разработки.

Единого определения для микросервисов не существует. Существует несколько пунктов, которые определяют суть микросервисной архитектуры:

● сервисы в микросервисной архитектуре (MSA) часто представляют собой процессы, которые обмениваются данными по сети для достижения цели с использованием независимых от технологий протоколов, таких как HTTP;

● сервисы в микросервисной архитектуре развертываются независимо друг от друга;

● сервисы могут быть реализованы с использованием разных языков программирования, баз данных, аппаратной и программной среды, в зависимости от того, что подходит лучше всего;

● сервисы небольшого размера с поддержкой обмена сообщениями, ограниченные контекстами, автономно разработанные, независимо развертываемые, децентрализованные, построенные и выпущенные с автоматизированными процессами.

Микросервис не является слоем в монолитном приложении (например, веб-контроллер или интерфейс). Скорее, это отдельная часть бизнес-функции, которая может через свои внутренние компоненты реализовывать многоуровневую архитектуру. С точки зрения стратегии, архитектура микросервисов, в сущности, следует философии Unix: «Делай одно и делай это хорошо».

Мартин Фаулер описывает архитектуру, основанную на микросервисах, как обладающую следующими свойствами:

● подходит для постоянного процесса разработки программного обеспечения. Изменение только небольшой части приложения требует

перестройки и повторного развертывания только одного или небольшого количества сервисов;

● придерживается принципов, таких как детализированные интерфейсы (для независимо развертываемых служб), бизнес-ориентированная разработка (например, проектирование на основе домена).

Архитектура микросервисов обычно применяется для облачных приложений, бессерверных вычислений и приложений, использующих развертывание легких контейнеров. По словам Фаулера, из-за большого количества (по сравнению с монолитными реализациями приложений) услуг для эффективной разработки, поддержки и эксплуатации таких приложений необходимы децентрализованная непрерывная доставка и DevOps с целостным мониторингом услуг.

DevOps - это набор практик, сочетающих разработку программного обеспечения (Dev) и ИТ-операции (Ops). Он направлен на сокращение жизненного цикла разработки систем и обеспечение непрерывной разработки с высоким качеством программного обеспечения. DevOps дополняет гибкую разработку программного обеспечения; несколько аспектов DevOps пришли из Agile методологии.

Следствием (и обоснованием) применения этого подхода является то, что отдельные микросервисы могут быть индивидуально масштабированы. В монолитном подходе приложение, поддерживающее три функции, должно было бы масштабироваться полностью, даже если бы только одна из этих функций имела ограничение ресурса. В случае микросервисов необходимо масштабировать только микросервис, поддерживающий функцию с ограниченными ресурсами, что обеспечивает преимущества оптимизации ресурсов и затрат.

## 10.1. История

Еще в 2005 году Питер Роджерс ввел термин «микро-веб-сервисы» ("Micro-Web-Services") во время презентации на конференции Web Services Edge. Вопреки общепринятому мышлению он выступил за «REST-сервисы» и высказал мысль о том, что «Программные компоненты - это микро-веб-сервисы». Также он говорил: «Микро-сервисы создаются с использованием Unix-подобных конвейеров. Сервисы могут вызывать другие сервисы».

Сложные сервисные сборки абстрагируются за простым URI-интерфейсом. Любой сервис с различной степенью детализации может быть открыт. Он описал, как хорошо спроектированная платформа микросервисов применяет основополагающие принципы архитектуры Web и REST сервисов вместе с Unix-подобным планированием и конвейерами, чтобы обеспечить радикальную гибкость и улучшенную простоту в обслуживании.

Работа Роджерса началась в 1999 году с исследовательского проекта Dexter в Hewlett Packard Labs , цель которого состояла в том, чтобы сделать код

менее хрупким и сделать масштабные, сложные программные системы устойчивыми к изменениям. В конечном итоге этот путь исследований привел к разработке ресурсно-ориентированных вычислений (ROC), обобщенной вычислительной абстракции, в которой REST является специальным подмножеством.

В 2007 году Ювал Леви в своих работах призвал к созданию систем, в которых каждый класс представлял собой услугу. Леви понял, что это требует использования технологии, которая может поддерживать такое гранулярное использование сервисов, и расширил Windows Communication Foundation (WCF), чтобы сделать это, взяв каждый класс и рассматривая его как сервис, поддерживая при этом условную модель программирования классов.

Семинар архитекторов программного обеспечения, проведенный недалеко от Венеции в мае 2011 года, использовал термин «микросервис», чтобы описать то, что участники рассматривали как общий архитектурный стиль, который многие из них недавно изучали. В мае 2012 года та же группа определила «микро-сервисы» в качестве наиболее подходящего названия. Джеймс Льюис представил некоторые из этих идей в качестве тематического исследования в марте 2012 года на 33-й степени в Кракове в Micro services - Java, Unix Way, как и Фред Джордж примерно в то же время. Адриан Кокрофт, бывший директор по облачным системам в Netflix, описал этот подход как «мелкозернистую SOA», впервые применил стиль в масштабах сети, как и многие другие, упомянутые в этой статье - Джо Уолнес, Дэн Норт, Эван Ботчер и Грэм Тэкли.

Микросервисы - это специализация подхода для реализации сервис-ориентированных архитектур (SOA), используемых для создания гибких, независимо развертываемых программных систем. Подход на основе микросервисов - это первая реализация SOA, которая последовала за внедрением DevOps и становится все более популярной для построения постоянно развертываемых систем.

В феврале 2020 года в Отчете об исследовании рынка облачных микросервисов прогнозировалось, что объем глобального рынка микросервисной архитектуры увеличится в среднем на 21,37 % с 2019 по 2026 год и достигнет 3,1 млрд. долларов к 2026 году.

## 10.2. Преимущества

Преимущество разложения приложения на различные меньшие сервисы многочисленны:

- модульность. Это упрощает понимание, разработку, тестирование и устойчивость приложения к разрушению архитектуры. Это преимущество часто обсуждается в сравнении со сложностью монолитных архитектур;

● масштабируемость: поскольку микросервисы внедряются и развертываются независимо друг от друга, то есть работают в независимых процессах, их можно контролировать и масштабировать независимо;

● интеграция унаследованных систем: микросервисы рассматриваются как жизнеспособное средство для модернизации существующего монолитного программного приложения. Имеются сообщения об опыте нескольких компаний, которые успешно заменили (частично) свое существующее программное обеспечение на микросервисы или находятся в процессе.

Процесс модернизации программного обеспечения устаревших приложений выполняется с использованием поэтапного подхода.

Распределенная разработка: она распараллеливает разработку, позволяя небольшим автономным группам независимо разрабатывать, развертывать и масштабировать свои соответствующие службы. Это также позволяет архитектуре отдельного сервиса появляться посредством непрерывного рефакторинга.

Микросервисные архитектуры обеспечивают непрерывную интеграцию, непрерывную доставку и развертывание.

Критика и проблемы.

Микросервисный подход подвергается критике по ряду вопросов:

● услуги формируют информационные барьеры;

● межсервисные вызовы по сети имеют более высокую стоимость с точки зрения задержки в сети и времени обработки сообщений, чем внутрипроцессные вызовы в рамках процесса монолитного обслуживания;

● тестирование и развертывание являются более сложными;

● перемещать обязанности между службами сложнее. Это может включать в себя взаимодействие между различными командами, переписывание функциональности на другом языке или встраивание его в другую инфраструктуру. Однако микросервисы могут быть развернуты независимо от остальной части приложения, в то время как команды, работающие над монолитами, должны синхронизироваться для совместного развертывания;

● просмотр размера сервисов в качестве основного механизма структурирования может привести к слишком большому количеству сервисов, когда альтернатива внутренней модульности может привести к упрощению конструкции. Это требует использования приложений, помогающих понять общую архитектуру приложений и взаимозависимости между компонентами;

● двухфазные фиксации рассматриваются в качестве антишаблонов в архитектурах на основе микросервисов, поскольку это приводит к более тесному взаимодействию всех участников транзакции;

● разработка и поддержка многих сервисов более сложная задача, если они построены с использованием различных инструментов и технологий. Это особенно проблематично, если инженеры часто перемещаются между проектами;

● протокол, обычно используемый с микросервисами (HTTP), был разработан для общедоступных служб и поэтому не подходит для работы внутренних микросервисов, которые часто должны быть безупречно надежными;

● само понятие микросервиса вводит в заблуждение, поскольку существуют только сервисы. Нет четкого определения, когда служба запускается или перестает быть микросервисом.

Архитектура вводит дополнительную сложность и новые проблемы, с которыми приходится иметь дело, такие как задержка в сети, резервное копирование/доступность/согласованность (BAC), балансировка нагрузки и отказоустойчивость.

Все эти проблемы должны решаться в масштабе. Согласно выводам из отчета State of Microservices 2020, обслуживание и устранение неполадок по-прежнему являются одной из самых больших проблем для групп разработчиков, создающих микросервисы.

Сложность монолитного приложения не исчезнет, если его повторно реализовать в виде набора микросервисных приложений. Возникнут другие места, где проявляется сложность, увеличивается сетевой трафик и, как следствие, снижается производительность. Кроме того, приложение, состоящее из любого количества микросервисов, имеет большее количество точек интерфейса для доступа к соответствующей экосистеме, что увеличивает сложность архитектуры.

Различные принципы организации (такие как HATEOAS, документация интерфейса и модели данных, полученные с помощью Swagger и т. д.) были применены для уменьшения влияния такой дополнительной сложности.

Микросервисы могут быть реализованы на разных языках программирования и могут использовать разные инфраструктуры. Следовательно, наиболее важными технологическими решениями являются способ взаимодействия микросервисов друг с другом (синхронный, асинхронный, интеграция пользовательского интерфейса) и протоколы, используемые для обмена данными (RESTful HTTP, обмен сообщениями, GraphQL и т.д.).

В традиционной системе большинство технологических решений, таких как язык программирования, влияют на всю систему. Поэтому подход к выбору технологий совсем другой.

## 10.3. Сервисная сетка

В архитектуре программного обеспечения сервисная сетка - это выделенный уровень инфраструктуры для облегчения связи между сервисами или микросервисами, часто с использованием прокси-сервера.

Наличие такого выделенного уровня связи может обеспечить ряд преимуществ, таких как обеспечение наблюдаемости при обмене данными, обеспечение безопасных соединений или автоматизация повторных попыток и отката для неудачных запросов.

В сервисной сетке каждый экземпляр сервиса связан с экземпляром обратного прокси-сервера, называемого прокси-сервером сервиса или sidecar-прокси. Экземпляр службы и прокси-сервер sidecar совместно используют контейнер, а контейнеры управляются инструментом оркестровки контейнеров, таким как Kubernetes, Nomad, Docker Swarm или DC/OS [15].

Прокси-серверы отвечают за связь с другими экземплярами службы и могут поддерживать такие функции, как обнаружение службы (экземпляра), балансировка нагрузки, аутентификацию и авторизацию, безопасную связь и другое.

## 10.4. Сравнение платформ

Реализация микросервисной архитектуры очень сложна. Существует много проблем, которые необходимо учитывать для любой микросервисной архитектуры. Netflix разработал микросервисную инфраструктуру для поддержки своих внутренних приложений. Многие из этих инструментов были популяризированы с помощью Spring Framework - они были повторно реализованы как инструменты на основе Spring под эгидой проекта Spring Cloud. Одним из примечательных аспектов экосистемы Spring Cloud является то, что все они основаны на технологиях Java [16].

## 10.5. Контрольные задания

### 10.5.1. Вопросы для самопроверки

1. Что такое микросервисная архитектура?
2. Определения сути микросервисной архитектуры.
3. Преимущества микросервисной архитектуры.
4. Распределённая разработка.
5. Проблемы микросервисной архитектуры.
6. Сервисная сетка.
7. Примеры платформ с микросервисной архитектурой.

# 11. СИСТЕМА REST

## 11.1. Что такое REST

REST (REpresentational State Transfer) - это архитектурный стиль для распределенных систем, который был впервые представлен Роем Филдингом в 2000 году в своей знаменитой диссертации.

Как и любой другой архитектурный стиль, REST также имеет свои 6 ограничений, которые должны быть выполнены для того, чтобы сервис назывался RESTful.

Основные принципы REST:

● клиент-сервер (Client–server) - необходимо отделять проблемы пользовательского интерфейса от проблем хранения данных для улучшения пользовательского интерфейса на различных платформах. За счет упрощения компонентов сервера улучшается масштабируемость;

● нет сохранения состояния (Stateless) - каждый запрос от клиента к серверу должен содержать всю информацию, необходимую для понимания запроса, и не может использовать какой-либо сохраненный контекст на сервере. Поэтому состояние сеанса полностью сохраняется на клиенте;

● кэшируемость (Cacheable) - ограничения Cache требуют, чтобы данные в ответе на запрос были неявно или явно помечены как кешируемые или не кешируемые. Если ответ кешируется, клиентскому кешу предоставляется право повторно использовать эти данные ответа для последующих эквивалентных запросов;

● унифицированный интерфейс (Uniform interface) - благодаря применению принципа общности разработки программного обеспечения к интерфейсу компонента упрощается общая архитектура системы и улучшается видимость взаимодействий. Чтобы получить унифицированный интерфейс, необходимо несколько архитектурных ограничений для управления поведением компонентов;

● многоуровневая система (Layered system). Стиль многоуровневой системы позволяет архитектуре состоять из иерархических уровней, ограничивая поведение компонента таким образом, что каждый компонент не может «видеть» за пределами непосредственного уровня, с которым они взаимодействуют;

● код по требованию (Code on demand) - это необязательное требование. REST позволяет расширить функциональность клиента путем загрузки и выполнения кода в форме апплетов или скриптов. Это упрощает работу клиентов за счет уменьшения количества функций, необходимых для предварительной реализации.

## 11.2. Ресурсы

Ключевая абстракция информации в REST - это ресурс. Любая информация может быть ресурсом: документ или изображение, временная служба, набор других ресурсов, не виртуальный объект (например, человек) и так далее.

Состояние ресурса в любой конкретной временной отметке называется представлением ресурса. Представление состоит из данных, метаданных, описывающих данные и гипертекстовые ссылки, которые могут помочь клиентам в переходе к следующему желаемому состоянию.

Гипертекст (или гипермедиа) означает одновременное представление информации и элементов управления, так что информация становится доступной, благодаря которой пользователь (или сервис) получает выбор и выбирает действия. Помните, что гипертекст не обязательно должен быть HTML (или XML, или JSON) в браузере. Сервисы могут переходить по ссылкам, когда они понимают формат данных и типы отношений [17].

Кроме того, представления ресурсов должны быть самоописательными. Таким образом, на практике вы в конечном итоге создадите множество пользовательских типов медиа - обычно один тип медиа, связанный с одним ресурсом.

Каждый тип носителя определяет модель обработки по умолчанию. Например, HTML определяет процесс рендеринга для гипертекста и поведение браузера вокруг каждого элемента. Он не имеет никакого отношения к методам ресурсов GET / PUT / POST / DELETE /…, за исключением того факта, что некоторые элементы медиа-типа будут определять модель процесса, которая выглядит как «якорные элементы с атрибутом href, создают гипертекстовую ссылку, которая при выборе вызывает запрос поиска (GET) по URI, используя атрибут href в кодировке CDATA».

**Ресурсные методы**

Еще одна важная вещь, связанная с REST, - это методы ресурсов, которые будут использоваться для выполнения желаемого перехода. Большое количество людей ошибочно связывают методы ресурсов с методами HTTP GET/PUT/POST/DELETE.

Рой Филдинг никогда не упоминал никаких рекомендаций относительно того, какой метод использовать в каких условиях. Если вы решите, что HTTP POST будет использоваться для обновления ресурса - вместо того, чтобы большинство людей рекомендовало HTTP PUT - все в порядке, и интерфейс приложения будет RESTful.

В идеале все, что необходимо для изменения состояния ресурса, должно быть частью ответа API для этого ресурса, включая методы, и в каком состоянии они будут выходить из представления.

Еще одна вещь, которая поможет вам при создании RESTful API, заключается в том, что результаты API на основе запросов должны быть представлены списком ссылок со сводной информацией, а не массивами исходных представлений ресурсов, поскольку запрос не заменяет идентификацию ресурсов.

В архитектурном стиле REST данные и функциональность считаются ресурсами и доступны с использованием унифицированных идентификаторов ресурсов (URI). На ресурсы воздействуют с помощью набора простых, четко определенных операций. Клиенты и серверы обмениваются представлениями ресурсов с использованием стандартизированного интерфейса и протокола - обычно HTTP.

Архитектура REST позволяет поставщикам API доставлять данные в различных форматах, таких как простой текст, HTML, XML, YAML и JSON, что является одной из его самых любимых функций. Благодаря растущей популярности REST, легкий и читаемый человеком формат JSON также быстро завоевал популярность, так как он отлично подходит для быстрого и безболезненного обмена данными [18].

JSON означает JavaScript Object Notation. Это простой для анализа и легкий формат обмена данными. Несмотря на свое название, JSON полностью не зависит от языка, поэтому его можно использовать с любым языком программирования, а не только с JavaScript. Его синтаксис является подмножеством стандарта ECMA-262, 3-е издание. Файлы JSON состоят из наборов пар имя/значение и упорядоченных списков значений, которые представляют собой универсальные структуры данных, используемые большинством языков программирования. Поэтому JSON может быть легко интегрирован с любым языком.

Чтобы увидеть разницу между XML и JSON, рассмотрим пример кода из документов API Crowd Server компании Atlassian, который позволяет запрашивать и принимать данные в форматах XML и JSON:

```
<?xml version="1.0" encoding="UTF-8"?>
<authentication-context>
  <username>my_username</username>
  <password>my_password</password>
  <validation-factors>
    <validation-factor>
      <name>remote_address</name>
      <value>127.0.0.1</value>
    </validation-factor>
  </validation-factors>
</authentication-context>
```

Вот как выглядит приведенный выше XML-код в JSON:

```
{
  "username" : "my_username",
```

```
  "password" : "my_password",
  "validation-factors" : {
    "validationFactors" : [
      {
        "name" : "remote_address",
        "value" : "127.0.0.1"
      }
    ]
  }
}
```

Как видите, JSON является более легким и менее многословным форматом, а также его легче читать и писать. В большинстве случаев он идеально подходит для обмена данными через Интернет. Тем не менее XML по-прежнему имеет некоторые преимущества. Например, XML позволяет размещать метаданные в тегах, а также лучше обрабатывать смешанный контент, особенно когда массивы смешанных узлов требуют подробных выражений.

## 11.3. Создание веб-службы RESTful

Для рассмотрения REST-приложения создадим веб-службу с помощью фреймворка Spring на языке Java.

Служба будет представлять из себя сервис, который будет принимать запросы HTTP GET на http://localhost:8080/greeting.

Данный сервис ответит JSON-представлением приветствия в формате JSON:

{"id":1,"content":"Hello, World!"}

Вы можете настроить приветствие с необязательным name параметром в строке запроса:

http://localhost:8080/greeting?name=User

Значение параметра name переопределяет значение по умолчанию World и отражается в ответе:

{"id":1,"content":"Hello, User!"}

Для этого необходимо:

- JDK 1.8 или более поздние версии;
- Gradle 4+ или Maven 3.2+.

Вы также можете импортировать код прямо в вашу IDE:

- Spring Tool Suite (STS);
- IntelliJ IDEA.

В данном листинге показан pom.xml файл, который создается при выборе Maven:

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

<<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
<<modelVersion>4.0.0<</modelVersion>
<<parent>
<<groupId>org.springframework.boot<</groupId>
<<artifactId>spring-boot-starter-parent<</artifactId>
<<version>2.2.2.RELEASE<</version>
<<relativePath/> <<!-- lookup parent from repository -->
<</parent>
<<groupId>com.example<</groupId>
<<artifactId>rest-service<</artifactId>
<<version>0.0.1-SNAPSHOT<</version>
<<name>rest-service<</name>
<<description>Demo project for Spring Boot<</description>
<<properties>
<<java.version>1.8<</java.version>
<</properties>
<<dependencies>
<<dependency>
<<groupId>org.springframework.boot<</groupId>
<<artifactId>spring-boot-starter-web<</artifactId>
<</dependency>
<<dependency>
<<groupId>org.springframework.boot<</groupId>
<<artifactId>spring-boot-starter-test<</artifactId>
<<scope>test<</scope>
<<exclusions>
<<exclusion>
<<groupId>org.junit.vintage<</groupId>
<<artifactId>junit-vintage-engine<</artifactId>
<</exclusion>
<</exclusions>
<</dependency>
<</dependencies>
<<build>
<<plugins>
<<plugin>
<<groupId>org.springframework.boot<</groupId>
<<artifactId>spring-boot-maven-plugin<</artifactId>
<</plugin>
<</plugins>

```
        <</build>
    <</project>
```
В следующем листинге показан build.gradle файл, который создается при выборе Gradle:
```
    plugins {
        id 'org.springframework.boot' version '2.2.0.RELEASE'
        id 'io.spring.dependency-management' version '1.0.8.RELEASE'
        id 'java'
    }
    group = 'com.example'
    version = '0.0.1-SNAPSHOT'
    sourceCompatibility = '1.8'
    repositories {
        mavenCentral()
    }
    dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-web'
        testImplementation('org.springframework.boot:spring-boot-starter-test')
{
            exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
        }
    }
    test {
        useJUnitPlatform()
    }
```
Теперь необходимо создать класс представления ресурса.

Служба будет обрабатывать GET-запросы /greeting необязательно с name параметром в строке запроса. GET-запрос должен возвращать HTTP код 200 (OK) ответ с JSON в теле HTTP, которое представляет собой приветствие. Он должен выглядеть следующим образом:
```
    {
      "id": 1,
      "content": "Hello, World!"
    }
```
id представляет собой уникальный идентификатор для приветствия, а content - это текстовое представление приветствия.

Чтобы смоделировать представление приветствия, создайте класс представления ресурса (Greeting.java):
```
    package com.example.restservice;
    public class Greeting {
        private final long id;
        private final String content;
        public Greeting(long id, String content) {
            this.id = id;
```

```
        this.content = content;
    }
    public long getId() {
        return id;
    }
    public String getContent() {
        return content;
    }
}
```

В нашем примере будет использоваться библиотека Jackson для автоматической сортировки экземпляров типа Greeting в JSON. Jackson включен по умолчанию веб-стартером.

Теперь необходимо создать контроллер ресурсов.

В подходе Spring к созданию RESTful веб-сервисов HTTP-запросы обрабатываются контроллером. Эти компоненты идентифицируются аннотациями @RestController и GreetingController, показанными в следующем листинге (GreetingController.java). Данный контроллер обрабатывает GET-запросы /greeting, возвращая новый экземпляр Greeting класса:

```
package com.example.restservice;
import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class GreetingController {
    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();
    @GetMapping("/greeting")
    public Greeting greeting(@RequestParam(value = "name", defaultValue
= "World") String name) {
    return new Greeting(counter.incrementAndGet(), String.format(template,
name));
    }
}
```

Ключевым отличием между традиционным контроллером MVC и контроллером веб-службы RESTful является способ создания тела ответа HTTP. Этот контроллер веб-службы RESTful заполняет и возвращает Greeting объект, данные которого будут записаны непосредственно в ответ HTTP как JSON.

Этот код использует @RestController аннотацию Spring, которая помечает класс как контроллер, где каждый метод возвращает объект домена вместо представления. Это сокращение для включения обоих @Controller и @ResponseBody.

Greeting объект должен быть преобразован в формат JSON. Благодаря поддержке конвертера HTTP-сообщений Spring вам не нужно делать это преобразование вручную.

@SpringBootApplication - это удобная аннотация, которая добавляет следующее:

● @Configuration: помечает класс как источник определений компонентов для контекста приложения;

● @EnableAutoConfiguration: говорит Spring Boot начать добавлять bean-компоненты на основе настроек пути к классам, других bean-компонентов и различных настроек свойств. Например, если spring-webmvc находится в пути к классам, эта аннотация помечает приложение как веб-приложение и активирует ключевые поведения, такие как настройка a DispatcherServlet;

● @ComponentScan: говорит Spring искать другие компоненты, конфигурации и сервисы в com/example пакете, позволяя ему найти контроллеры.

Вы можете запустить приложение из командной строки с помощью Gradle или Maven. Вы также можете создать один исполняемый файл JAR, который содержит все необходимые зависимости, классы и ресурсы, и запустить его. Создание исполняемого файла jar позволяет легко доставлять, создавать версии и развертывать службу как приложение на протяжении всего жизненного цикла разработки в разных средах и т.д.

Если вы используете Gradle, вы можете запустить приложение с помощью ./gradlew bootRun. В качестве альтернативы вы можете создать файл JAR с помощью ./gradlew build, а затем запустить файл JAR следующим образом:

java-jar build / libs / gs-rest-service-0.1.0.jar.

Если вы используете Maven, вы можете запустить приложение с помощью ./mvnw spring-boot:run. Кроме того, вы можете создать JAR-файл с помощью ./mvnw clean package, а затем запустить JAR-файл следующим образом:

java -jar target / gs-rest-service-0.1.0.jar.

Служба должна быть запущена в течение нескольких секунд.

Теперь, когда сервис запущен, посетите http://localhost:8080/greeting, где вы должны увидеть:

{"id":1,"content":"Hello, World!"}

Укажите name-параметр строки запроса, посетив страницу http://localhost:8080/greeting?name=User.

Обратите внимание, как значение content изменяется с “Hello, World!” на “Hello, User!”, как показано в следующем листинге:

{"id":2,"content":"Hello, User!"}

Это изменение демонстрирует, что @RequestParam в GreetingController работает должным образом. Параметру name может быть дано значение по умолчанию “World”, или его можно явно определить через строку запроса.

Обратите также внимание, как id атрибут изменился с 1 на 2. Это доказывает, что вы работаете с одним и тем же GreetingController экземпляром в нескольких запросах и что его counter поле увеличивается при каждом вызове, как и ожидалось.

### 11.4. Контрольные задания

#### 11.4.1. Вопросы для самопроверки

1. Что такое REST?
2. Основные принципы REST.
3. Что такое ресурс в REST?
4. Какие существую методы работы с ресурсами?
5. Какие форматы данных поддерживает REST?
6. Что такое веб-служба и для чего она нужна?

## 12. ПРОТОКОЛ ОБМЕНА СООБЩЕНИЯМИ SOAP

SOAP (сокращение от Simple Object Access Protocol ) - это спецификация протокола обмена сообщениями для обмена структурированной информацией при реализации веб-сервисов в компьютерных сетях. Его целью является обеспечение расширяемости, нейтральности, многословия и независимости. Он использует XML Information Set для своего формата сообщений и протоколы прикладного уровня, чаще всего Hypertext Transfer Protocol (HTTP), хотя некоторые устаревшие системы взаимодействуют через Simple Mail Transfer Protocol (SMTP) для согласования и передачи сообщений.

SOAP позволяет разработчикам вызывать процессы, работающие в разных операционных системах (таких как Windows, macOS и Linux), для аутентификации, авторизации и обмена данными с использованием расширяемого языка разметки (XML). Поскольку веб-протоколы, такие как HTTP, установлены и работают во всех операционных системах, SOAP позволяет клиентам вызывать веб-службы и получать ответы независимо от языка и платформ.

### 12.1. Характеристики

SOAP имеет три основных характеристики:

- расширяемость;
- нейтральность;
- независимость.

В качестве примера того, что могут делать процедуры SOAP, можно рассмотреть приложение, которое отправляет запрос SOAP на сервер с

включенными веб-службами, такими как база данных. Затем сервер возвращает ответ SOAP (документ в формате XML с результирующими данными). Поскольку сгенерированные данные поступают в стандартизированном формате, пригодном для машинного анализа, запрашивающее приложение может затем интегрировать их напрямую.

Вот как выглядит обычное SOAP-сообщение. Пример взят из документов W3C SOAP и содержит конверт SOAP, блок заголовка и тело:

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
<n:alertcontrol xmlns:n="http://example.org/alertcontrol">
     <n:priority>1</n:priority>
     <n:expires>2001-06-22T14:00:00-05:00</n:expires>
   </n:alertcontrol>
  </env:Header>
  <env:Body>
   <m:alert xmlns:m="http://example.org/alert">
     <m:msg>Pick up Mary at school at 2pm</m:msg>
   </m:alert>
  </env:Body>
</env:Envelope>
```

## 12.2. Какая основная причина использовать SOAP?

В краткосрочной и среднесрочной перспективе SOAP, вероятно, будет по-прежнему использоваться для веб-служб уровня предприятия, которые требуют высокой безопасности и сложных транзакций. API-интерфейсы для финансовых услуг, платежных шлюзов, программного обеспечения CRM, управления идентификацией и телекоммуникационных услуг являются широко используемыми примерами SOAP.

Одним из наиболее известных API-интерфейсов SOAP является общедоступный API-интерфейс PayPal, позволяющий принимать платежи через PayPal и кредитные карты, добавлять кнопку PayPal на свой веб-сайт, разрешать пользователям входить в систему через PayPal и выполнять другие действия, связанные с PayPal.

Поддержка устаревших систем - еще один частый аргумент в пользу использования SOAP. У популярных веб-сервисов, которые существуют уже некоторое время, может быть много пользователей, которые все еще подключаются к своим сервисам через свой SOAP API, который был лидером рынка до того, как REST приобрел популярность. Например, Salesforce предоставляет как SOAP, так и REST API, чтобы каждый разработчик мог интегрировать Salesforce со своей собственной платформой так, как им удобно.

SOAP может быть отличным решением в ситуациях, когда вы не можете использовать REST.

Кроме того, SOAP может быть отличным решением в ситуациях, когда вы не можете использовать REST. Хотя в наши дни большинство поставщиков веб-услуг хотят обмениваться запросами и ответами без сохранения состояния, в некоторых случаях вам может потребоваться обрабатывать операции с состоянием. Это происходит в сценариях, в которых цепочка операций должна действовать как одна транзакция, например, в случае банковских переводов.

Хотя API-интерфейсы SOAP по умолчанию не имеют состояния, SOAP поддерживает операции с состоянием, которые могут быть реализованы с использованием спецификаций WS-* (веб-сервисов), основанных на основных стандартах XML и SOAP [19].

## 12.3. Основные отличия REST от SOAP

● REST поддерживает различные форматы: text, JSON, XML; SOAP - только XML.

● REST работает только по HTTP(S), а SOAP может работать с различными протоколами.

● REST может работать с ресурсами. Каждый URL - это представление какого-либо ресурса. SOAP работает с операциями, которые реализуют какую-либо бизнес логику с помощью нескольких интерфейсов.

● SOAP на основе чтения не может быть помещена в кэш, а REST в этом случае может быть закэширован.

● SOAP поддерживает SSL и WS-security, в то время как REST - только SSL.

● SOAP поддерживает ACID (Atomicity, Consistency, Isolation, Durability). REST поддерживает транзакции, но не один из ACID не совместим с двух фазовым коммитом.

## 12.4. Контрольные задания

### 12.4.1. Вопросы для самопроверки

1. Что такое SOAP?
2. Основные характеристики SOAP.
3. Пример общедоступных SOAP интерфейсов.
4. Различие между SOAP и REST.

## 13. ВВЕДЕНИЕ В GraphQL

GraphQL — это язык запросов и манипулирования данными с открытым исходным кодом для API , а также среда выполнения для выполнения запросов с существующими данными, разработанная Facebook. Он создавался как более эффективная альтернатива REST для разработки и использования программных интерфейсов приложений.

GraphQL обладает множеством достоинств, например:

1. Вы получаете информацию именно в том объеме, в котором запрашиваете. В отличие от REST (Representational state transfer), ответ на запрос не будет содержать ненужных данных.

2. Вам будет необходима всего одна конечная точка, никаких дополнительных версий для единственного API (**A**pplication **P**rogramming **I**nterface).

3. GraphQL - сильно типизированный язык, что позволяет предварительно оценить корректность запроса в рамках системы типов этого синтаксиса до исполнения. Это позволяет разрабатывать более мощные API.

Для создания схем в GraphQL используется собственный язык Schema Definition Language (SDL). SDL обладает интуитивно понятным синтаксисом и универсален для любой используемой технологии [20].

**Введите GraphQL**

GraphQL использует 1 конечную точку для всего.

Самая базовая структура graphql может быть:

```
{
    me {
        name
    }
}
```

который даст следующий ответ:

```
{
"me": {
"name": "Luke Skywalker"
}
}
```

**Преимущества**

1. Клиент может контролировать, какие данные он хочет.

2. Клиент может легко сказать, какие данные он будет получать.

3. Поскольку существует одна конечная точка, ее очень легко поддерживать и для клиента.

Клиент запрашивает различные данные с сервера GraphQL через запросы. Формат ответа описывается в запросе и определяется клиентом, а не сервером: они называются запросами, указанными клиентом.

Структура данных не жестко запрограммирована, как в традиционных API REST. Это делает получение данных с сервера более эффективным для клиента.

Например, клиент может запросить связанные ресурсы без определения новых конечных точек API. С помощью следующего запроса GraphQL мы можем запросить поля пользователя и связанный ресурс друзей.

```
{
 user(id: 1) {
  name
   age
  friends {
        name
   }
 }
}
```

В REST API на основе ресурсов это будет выглядеть примерно так:
*GET /users/1 and GET /users/1/friends*

или же
*GET /users/1?include=friends.name*
Ключевые понятия языка запросов GraphQL:

- иерархическая;
- продукт-ориентированный;
- Strong-типирование;
- клиентские запросы;
- интроспективный.

В 2000 году Рой Филдинг, специалист по информатике, представил миру самую первую идею REST API. Этот программный архитектурный стиль был разработан, чтобы помочь разработчикам столкнуться со многими препятствиями, такими как доступ и использование API. Вскоре после этого REST стал популярным благодаря различным преимуществам.

Тем не менее, когда дело доходит до REST API, все еще существуют огромные недостатки.

Например, REST будет использовать один URL для определенного ресурса. Итак, у нас есть 2 сценария:

1. Запрос прост. В этом случае REST работает и отлично уменьшает вывод.
2. Но когда запрос сложный или идет с огромной суммой, количество запросов в REST должно увеличиваться. Следовательно, требуется гораздо больше усилий и запросов для разрешения этих запросов. И никто не доволен этим.

Другими словами, REST может возвращать настроенный набор данных, но для этого нужно строго следовать некоторым соглашениям. Между тем возврат настроенного набора данных является функцией по умолчанию для GraphQL.

Использование REST похоже на использование старого способа заказа пиццы, доставки продуктов и вызова химчистки для получения одежды. Вы должны сделать три телефонных звонка, чтобы связаться с этими 3 поставщиками услуг.

Благодаря удобному и гибкому подходу многие крупные и мелкие организации, такие как Facebook, Github, Amazon и т. д., используют этот новый язык запросов для своих общедоступных API.

## 13.1. Что может сделать GraphQL?

Имеется 4 основные особенности GraphQL:

- иерархические - запросы выглядят точно так же, как и данные, которые они возвращают;

- заданные клиентом запросы - клиент имеет право диктовать, что выбрать с сервера;

- строго типизированный - вы можете проверить синтаксически запрос и в системе типов GraphQL перед выполнением. Это также помогает использовать мощные инструменты, улучшающие опыт разработки, такие как GraphQL;

- Introspective - вы можете запросить систему типов, используя сам синтаксис GraphQL. Это отлично подходит для разбора входящих данных в строго типизированных интерфейсах и не требует разбора и ручного преобразования JSON в объекты.

Прежде всего, при использовании GraphQL размер запроса клиента уменьшается. Таким образом, вы можете сэкономить массу времени и усилий, затрачиваемых на выполнение запросов. Нам нужен только один URL для всей выборки и изменения данных при работе на сервере GraphQL.

Кроме того, этот язык запросов способен обнаруживать форму отправленных данных и получать подробную информацию о необходимых данных, а также об ошибках от API через свою систему.

Что еще? Что ж, GraphQL может автоматически создавать несколько основных форм кода, когда разработчики пишут свои запросы.

И поскольку эта структура разработки API создается типами и полями, во-первых, она может извлекать данные для разных конечных точек, а во-вторых, получать доступ к любой возможности данных только из одной конечной точки. В результате клиенты получат точные данные, которые они запрашивали в начале. Эта особенность делает GraphQL выдающимся языком запросов.

Некоторые другие дополнительные преимущества при использовании GraphQL вместо REST или других языков запросов:

- самодокументированность;
- поддержка развивающихся API со временем;
- поддержка функциональности в реальном времени.

Во-первых, с точки зрения кэширования в REST URL-адреса являются глобально уникальными идентификаторами для ресурсов, следовательно, клиент может использовать это для создания кэша. Однако в GraphQL нет URL-подобного примитива для данного ресурса.

Во-вторых, если ваш проект небольшой, вам не нужно использовать GraphQL для выполнения задач извлечения, потому что этот язык довольно сложный. Это все равно, что использовать кувалду, чтобы сломать орех.

И последнее, но не менее важное: если вы хотите использовать GraphQL, подготовьтесь к обучению. Разработчики должны владеть знаниями языка определения схемы, знакомиться с новыми определениями (например, преобразователем, схемой и т. д.). Если ваша команда должна выполнить проект в течение короткого времени и ограниченных ресурсов, тогда REST будет более практичным [21].

Преимущества, которые имеют значение для начинающих в GraphQL:

- поддерживаемость: единая конечная точка API для сервера всех потребностей данных, против распространения конечных точек API в REST;
- одиночная поездка в оба конца: все данные выбираются в одном цикле и в несколько раундов в REST. В нашем примере выше, чтобы клиент REST мог получить все продукты в каждой категории, ему необходимо сначала получить категории магазина из одной конечной точки API, а затем для каждой категории получить свои продукты из другой конечной точки API (известной как 'n + 1');
- никаких перегрузок: клиент полностью контролирует выбор только тех данных, которые ему нужны, в отличие от REST API, на стороне сервера мы должны балансировать между минимизацией нескольких циклов (проблема 'n + 1') и перегрузкой; мы можем устранить проблему 'n + 1', которую описали выше, если создадим API, который возвращает категории магазина, а также вернет все продукты, но другой набор клиентов, которым нужны категории магазина, будет перегружен.

Часто при использовании GraphQL люди используют Relay, если они используют React; вы можете размещать компоненты React вместе с их объявлением данных, что приводит к автоматическому извлечению данных с помощью Relay и автоматическому заполнению компонентов React извлеченными данными, что устраняет необходимость в записи запросов AJAX.

Кроме того, ретрансляция упрощает использование GraphQL с помощью множества функций, таких как повторное получение дополнительных/измененных данных предыдущих запросов, разбиение на страницы для данных с множеством записей и т. д.

### 13.2. Контрольные задания

#### 13.2.1. Вопросы для самопроверки

1. Что такое GraphQL?
2. Преимущества GraphQL.
3. Ключевые понятия языка запросов GraphQL.
4. Пример организаций, использующих GraphQL.
5. Проблемы GraphQL.

## 14. ОНТОЛОГИИ И МОДЕЛИ ДАННЫХ. ПРИНЦИПЫ ОНТОЛОГИЧЕСКОГО ОПИСАНИЯ ПРЕДМЕТНОЙ ОБЛАСТИ. ЯЗЫКИ ОПИСАНИЯ ОНТОЛОГИЙ

Технологии БД сформировались как значимая ветвь информационных технологий в конце 60-х – начале 70-х гг. XX века. Начали производиться системы управления базами данных (СУБД) общего назначения, становились востребованными методологии проектирования БД и реализующий их инструментарий. В соответствии со сложившимися методологиями проектирования БД начальным этапом этого процесса является формирование спецификации абстрактного представления предметной области (ПО) с помощью подходящих выразительных средств. Этот этап называется концептуальным моделированием ПО, а его результат – концептуальной схемой предметной области (КС).

Концептуальная модель – абстрактное описание предметной области, независимое от аспектов реализации систем, в рамках которых оно используется, определяющее концептуальную структуру и поведение сущностей в предметной области.

### 14.1. Онтологический инжиниринг

В середине 90-х гг. в крупных корпорациях, для которых проблемы обработки информации стали критическими, появилось понятие «управление знаниями». Оно подразумевает создание базы знаний, которая содержит объединенные знания всех сотрудников, необходимые, например, для поддержания процесса принятия решений. При этом стало очевидным, что основным узким местом является работа со знаниями (сохранение, поиск, интеграция и пр.), накопленными специалистами компании, так как именно знания обеспечивают преимущество перед конкурентами.

Появление знаний как информационных объектов для обработки на компьютерах определило переход от БД к БЗ. Раздел искусственного

интеллекта, изучающий базы знаний и методы работы со знаниями, называется инженерией знаний.

Онтология – это спецификация на концептуальном уровне. Данный термин произошел от названия раздела философии, где изучаются проблемы бытия.

В контексте информационных технологий этот термин приобрел другой смысл, а именно онтология – это попытка всеобъемлющей и детальной формализации некоторой области знаний с помощью концептуальной схемы. Онтология включает в себя словарь указателей на термины ПО и логические выражения, описывающие соотношение между ними, т.е. онтология – это сеть, содержащая термины и понятия и показывающая взаимосвязь между ними.

Концепт – это шаблон, содержащий множество правил, определяющих форму экземпляра, т.е. то, каким образом может быть построен экземпляр. Концепты могут иметь атрибуты – имена или структуры полей записи. Атрибуты характеризуют размер или тип информации, содержащейся в поле. Отношениями между концептами называются зависимости между экземплярами онтологии. Обычно отношением является атрибут, ссылающийся на другой экземпляр [22].

## 14.2. Средства семантического описания данных

На сегодняшний день имеется много средств семантического описания данных, многие из которых считаются достаточно выразительными для задач семантического моделирования данных. В качестве примера можно привести модель описания ресурсов (Resource Definition Framework), диаграммы Сущность-Связь (Entity-Relationship model). Преимущества онтологий перед другими механизмами описания семантики предметной области, например, такие как RDFS, ER-диаграммами заключается в том, что:

1. Ограничения традиционных моделей данных. Под термином «модель данных», согласно Когаловскому М.Р., понимается инструмент моделирования, т.е. является совокупностью понятий для описания данных, для описания структуры данных. «Модель предметной области» представляет собой визуальное представление сущностей предметной области и отношений между ними, т.е. спецификацию модели предметной области, является результатом моделирования.

2. Модель Entity Relationship ER – является основой, из которой могут быть порождены три существующие модели данных: сетевая модель, реляционная модель и модели набора сущностей, представляя данные более строго и естественно и одновременно обеспечивая независимость данных от приложений (ER-модель основывается на теории множеств и реляционной теории). С тех пор было предложено множество расширений ER-схем, чтобы обеспечить более мощные средства выражения семантики данных: механизмы

задания иерархии подклассов классов сущностей, некоторых семантических ограничений типа «часть-целое», реификаций как классов сущностей, благодаря которым можно было распознавать общие характеристики сущностей различных классов. Примеры таких моделей - "semantic data modeling", "extended ER modeling", "hyper-semantic data modeling", "OMT approach" и др [23].

Ограничения ER-модели и её расширений в том, что они, описывая семантику «сущностей», позволяют интерпретировать данные одним единственным способом.

3. Архитектура ANSI/SPARC. Повторное использование знаний в различных контекстах невозможно без наличия механизмов, позволяющих фиксировать различное понимание этих знаний. Идея разработки такого механизма была представлена частично в ANSI/SPARC-архитектуре баз данных.

Эта архитектура включает три уровня:

1) логический уровень (называемый «концептуальной схемой»), который является промежуточным уровнем и основой данной архитектуры;

2) внутреннее представление базы данных описывает способ, по которому концептуальная схема может быть реализована в терминах объектов физического уровня: файлов, индексов, хэш-таблиц и т.д.;

3) на верхнем уровне концептуальной модели можно определить множественное «внешнее представление». Оно будет состоять из выборок и комбинаций элементов концептуальной схемы и представлять видение схемы для каждого конкретного пользователя этого приложения. Например, база данных, содержащая административную информацию о сотрудниках организации, должна содержать два различных представления данных: для финансового отдела и для самих научных сотрудников [24].

Языки описания онтологий

Одним из важных моментов в создании онтологической модели ПО является выбор конкретного языка описания. Цель таких языков – сделать машинное представление данных более похожим на положение вещей в реальном мире, существенно повысить выразительные возможности концептуального моделирования слабо структурированных данных, дать возможность указывать дополнительную машинно-интерпретируемую семантику ресурсов.

Существуют традиционные языки спецификации онтологий: Ontolingua, CycL, языки, основанные на дескриптивных логиках (такие как LOOM), языки, основанные на фреймах (OKBC, OCML, FLogic). Специально для обмена онтологиями через Web были созданы языки RDF, RDFS, DAML+OIL, OWL.

## 14.3. Язык RDF

Спецификация RDF (Resource Description Framework) состоит из двух частей: модели представления информации в WWW и синтаксиса для представления этой информации в конкретном цифровом виде для дальнейшего хранения, передачи и обработки. Базовыми понятиями RDF являются ресурсы, отношения, также называемые предикатами и свойствами, и утверждения, представляющие собой триплет «субъект-предикат-объект».

Субъект – это описываемый ресурс.

Предикат – это тот аспект ресурса, который описывается (другими словами, свойство ресурса).

Объект – это уже конкретное значение отношения (значение свойства ресурса).

Все ресурсы и отношения (свойства) в RDF идентифицируются с помощью URI, что позволяет описывать с помощью RDF как интернет-ресурсы, так и ресурсы, которые не могут быть получены через Интернет (человек, автомобиль, город и т.д.).

Например, если информация о том, что Иванов Иван является автором ресурса http://www.example.org/, расположена по адресу http://www.example.org/creator#, то в терминах RDF в нотации N-Triple это описание должно быть выражено следующим образом:

<http://www.example.org/creator#person>

<http://www.example.org/creator#/name>

<http://www.example.org/creator#И. Иванов>.

Использованная нотация N-Triple не является стандартной формой записи RDF-описаний – рекомендованный синтаксис для записи RDF является RDF/XML, который более удобен для автоматической обработки, хотя и более сложен для восприятия человеком.

Запись троек в XML-документах производится иначе, чем в нотации N-Triple. Например, записи XML-документа, который используется для передачи данных из таблицы «Сведения о количестве земельных участков за 2011 г.» (таблица).

Таблица

| № п/п | Наименование кадастрового района | Количество земельных участков в ГКН | Количество земельных участков поставлено на учет | Количество земельных участков снято с учета | Количество земельных участков, по которым проведен учет изменений |
|---|---|---|---|---|---|
| 1 | Мичуринский | 12636 | 845 | 35 | 7747 |
| 2 | Клевцовский | 3289 | 1375 | 987 | 41942 |
| 3 | Астраханьский | 15423 | 1455 | 140 | 186342 |
| 4 | Красноармейский | 15233 | 1506 | 3173 | 23426 |
| 5 | Мариупольский | 7444 | 5123 | 108 | 126292 |

XML-документ для передачи данных таблицы будет выглядеть следующим образом:

<<?xml version="1.0" encoding="windows-1251" standalone="no" ?>
<<!-- File Name: regions.xml -->
<<?xml-stylesheet type="text/css" href="ST01.css" ?>
<<!DOCTYPE Region-list
[
<<!ELEMENT Region-list (REGION*)>
<<!ELEMENT REGION (ID,NAME_CR,GKN,REGISTERED,REMOVAL,CHANGES) >
<<!ELEMENT ID ANY>
<<!ELEMENT NAME_CR ANY>
<<!ELEMENT GKN ANY>
<<!ELEMENT REGISTERED ANY>
<<!ELEMENT REMOVAL ANY>
<<!ELEMENT CHANGES ANY>
] >
<<Region-list>
<<REGION >
<<ID>1<</ID>
<<NAME_CR>Мичуринский<</NAME_CR>
<<GKN>12636<</GKN>
<<REGISTERED>845<</REGISTERED>
<<REMOVAL>35<</REMOVAL>
<<CHANGES>7747<</CHANGES>
<</REGION>
<<REGION >
<<ID>2<</ID>
<<NAME_CR>Клевцовский<</NAME_CR>
<<GKN>3289<</GKN>
<<REGISTERED>137<</REGISTERED>
<<REMOVAL>987<</REMOVAL>
<<CHANGES>4194<</CHANGES>
<</REGION>
<<REGION >
<<ID>3<</ID>
<<NAME_CR>Астраханьский<</NAME_CR>
<<GKN>15423<</GKN>
<<REGISTERED>1455<</REGISTERED>
<<REMOVAL>140<</REMOVAL>
<<CHANGES>18634<</CHANGES>
<</REGION>
…….
<</Region-list>

Далее рассматривается, как будет записан RDF-граф, приведенный в примере таблицы.

Субъектом в данном случае будет «кадастровый район» (тэг <REGION> в XML-документе). Предикатами являются уникальный номер региона в таблице (тэг <ID>), наименование (тэг <NAME_CR>), количество земельных участков в ГКН (тэг <GKN>), количество земельных участков, поставленных на учет (тэг <REGISTERED>), количество земельных участков, снятых с учета (тэг <REMOVAL>), и количество участков, по которым произведен учет изменений(<CHANGES>).

Задача RDF – это описание ресурсов в Сети, например, таблица находится по адресу http://www.example.org/cadastral-regions#.

<<rdf:RDF>

xmlns: regions=”http://www.example.org/cadastral-regions#”

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

<<regions: region rdf:about=” http://www.example.org/cadastral-regions#region1”>

<<regions:id>1<</regions:id>

<<regions:name_cr>Мичуринский<</regions:name_cr>

<<regions:gkn>12636<</regions:gkn>

<<regions:registered>845<</regions:registered>

<<regions:removal>35<</regions:removal>

<<regions:changes>7747<</regions:changes>

<</regions:region>

<<regions:region rdf:about=” http://www.example.org/cadastral-regions#region2”>

<<regions:id>2<</regions:id>

<<regions:name_cr>Клевцовский<</regions:name_cr>

<<regions:gkn>3289<</regions:gkn>

<<regions:registered>137<</regions:registered>

<<regions:removal>987<</regions:removal>

<<regions:changes>4194<</regions:changes>

<</regions:region>

<<regions:region rdf:about=” http://www.example.org/cadastral-regions#region3”>

<<regions:id>3<</regions:id>

<<regions:name_cr>Астраханьский<</regions:name_cr>

<<regions:gkn>15423<</regions:gkn>

<<regions:registered>1455<</regions:registered>

<<regions:removal>140<</regions:removal>

<<regions:changes>18634<</regions:changes>

<</regions:region>

……..

<</rdf:RDF>

## 14.4. Язык OWL

Другая спецификация – OWL (Web Ontology Language – язык онтологии Web), это язык, созданный для описания онтологии Web. Он совместим с языком RDF, т.е. каждый OWL-документ является также и RDF-документом. Базовыми понятиями OWL являются класс, свойство, экземпляр класса, ограничение на свойство, мощность свойства, инверсное свойство и многие другие, с помощью которых можно явно описывать различные предметные области.

Базовым элементом языка OWL является класс, определяемый как owl: Class, следовательно, любой OWL класс должен быть задан как экземпляр класса OWL класс. Например, если нужно определить класс Student (Студент), то необходимо определить тройку Student rdf: type owl: Class, которая в XML синтаксисе будет выглядеть следующим образом: <owl:Class rdf:ID= “Student”/>.

В языке OWL также существует два предопределенных класса:

- класс owl:Thing (сущность), который обозначает множество всех индивидов;
- класс owl:Nothung, обозначающий пустое множество.

Наследование классов в языке OWL задается с помощью конструкции rdfs:subClassOf, а тот факт, что один класс является дочерним классом другого, означает, что все экземпляры дочернего класса являются экземплярами родительского класса.

Свойства в OWL делятся на два класса:

- объектные свойства – это экземпляры класса owl:ObjectProperty. Свойства типов данных связывают индивидов с так называемыми значениями типов данных;
- свойства типов данных – это экземпляры класса owl:DataTypeProperty.

Язык OWL позволяет описывать различные характеристики классов и свойств, которые обычно задаются как разного рода ограничения на структуру связей между своими экземплярами. Эти ограничения выражаются в виде предопределенных соотношений, называемых в OWL аксиомами.

## 14.5. Контрольные задания

### 14.5.1. Вопросы для самопроверки

1. Что такое концептуальная модель?
2. Что такое модель «сущность-связь»
3. Что такое онтология?
4. Что такое концепт?

5. Преимущества онтологий перед другими механизмами описания семантики предметной области.
6. Уровни архитектуры ANSI/SPARC.
7. Язык RDF и его структура.
8. Язык OWL и его структура.

# 15. Middleware ДЛЯ ОБЩЕЙ ШИНЫ (IBM Message Broker И АНАЛОГИ) - ПРИНЦИПЫ ПОСТРОЕНИЯ, РЕШАЕМЫЕ ЗАДАЧИ

По мере развития любой компании появляются новые бизнес-процессы, требующие автоматизации, усложняются схемы взаимодействия IT-систем. Таким образом, по прошествии нескольких лет многие IT-директора сталкиваются с проблемой: в состав используемого ПО входит целый набор «проверенных временем» систем, но при этом взаимодействие между ними реализовано лишь частично, плохо структурировано, не подчинено единому стандарту, а необходимость создания новой интеграции IT-систем почти всегда требует использования собственных разработок или приобретения еще одного дорогостоящего программного продукта.

Кроме того, нередко, ввиду отсутствия обратной совместимости, перевод какой-либо системы на новую версию влечет за собой необходимость модификации ПО, реализующего связь с другими подсистемами. Все это неизбежно находит отражение в возрастающем объеме инвестиций в IT-блок организации, т.к. для покрытия требований бизнеса необходимо внедрение новых IT-систем и, как следствие, поиск и обучение дорогостоящих технических специалистов.

Для решения данной проблемы используются различные архитектуры построения приложений. Самая распространенная – сервисная шина предприятия [25].

## 15.1. Архитектура ESB

Сервисная шина предприятия (ESB) - это интегрированная платформа, предоставляющая базовые сервисы взаимодействия и связи для сложных программных приложений через управляемый событиями и основанный на стандартах механизм обмена сообщениями или шину, построенную на основе технологий продуктов промежуточного программного обеспечения. Платформа ESB предназначена для изоляции канала между сервисом и транспортным каналом и используется для удовлетворения требований сервис-ориентированной архитектуры (SOA). Модель сервисной шины предприятия представлена на рис. 15.1.

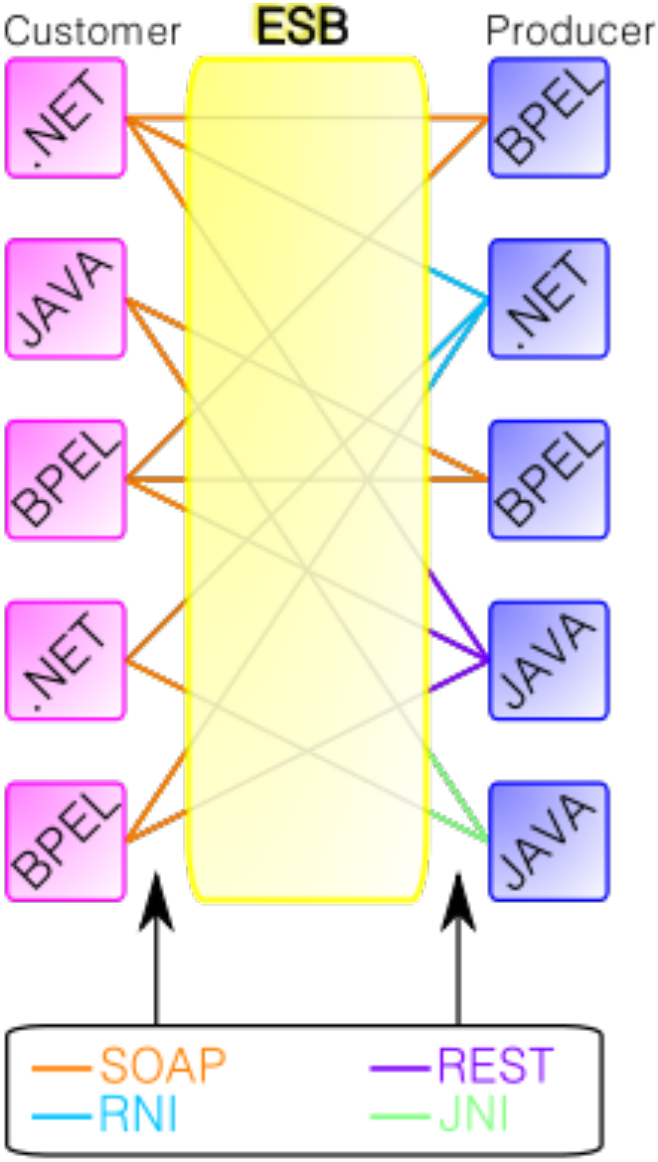


Рис. 15.1. Сервисная шина предприятия

Мнения относительно точного определения ESB отличаются, потому что этот термин часто относится к базовой программной инфраструктуре ESB.

ESB включает в себя следующие основные компоненты:

– архитектурная платформа;

– программный продукт;

– пакет программных продуктов.

ESB обеспечивает концептуальный уровень для установленной системы обмена сообщениями предприятия, который позволяет архитекторам интеграции применять преимущества обмена сообщениями без написания кода. В отличие от традиционных методов интеграции корпоративных приложений (EAI), таких как монолитный концентратор или стек спиц, ESB основан на простых функциях, разделенных как элементарные части с распределенным развертыванием и совместным использованием по мере необходимости.

Кроме того, ESB имеет структурные элементы SOA и SOA 2.0 на основе метрик, которые обеспечивают гибкость и возможность использования нескольких транспортных сред. Большинство провайдеров ESB интегрируют значения SOA с учетом независимых форматов сообщений.

## 15.2. Функции ESB

Сервисная шина предприятий выполняет следующие функции:

– связи;

– взаимодействий сервисов;

– интеграции;

– обеспечения качества сервиса;

– безопасности;

– моделирования;

– интеллектуальные функции инфраструктуры.

Впервые ESB была представлена в 2002 году и к настоящему времени получила широкое применение. Преимуществом сервисной шины, главным образом, является относительная дешевизна, простота в использовании, надежность продукта.

Основываясь на современных требованиях, сервисная шина предприятий должна выполнять следующие основные задачи:

– осуществлять маршрутизацию и передачу сообщений между сервисами;

– отделять представление сервисов для потребителя от ее реализации;

– инкапсулировать технические особенности взаимодействия сервисов;

– обеспечивать взаимодействие сервисов и управление ими в масштабе предприятия.

Эти задачи реализуются заменой непосредственных связей между потребителем и поставщиком сервиса. Иными словами, замена связи «точка-точка» сервисной шиной предприятия [26].

Брокер сообщений – это промежуточный компьютерный программный модуль, который переводит сообщение из формального протокола обмена сообщениями отправителя в формальный протокол обмена сообщениями получателя. Брокеры сообщений – это элементы в телекоммуникационных или компьютерных сетях, где программные приложения обмениваются данными путем обмена формально определенными сообщениями. Брокеры сообщений являются строительным блоком промежуточного программного обеспечения, ориентированного на сообщения (MOM), но обычно не являются заменой традиционного промежуточного программного обеспечения, такого как MOM и удаленный вызов процедур.

Брокер сообщений – это архитектурный шаблон для проверки, преобразования и маршрутизации сообщений. Он обеспечивает связь между

приложениями, сводя к минимуму взаимное понимание того, что приложения должны иметь прямые связи друг к другу, чтобы иметь возможность обмениваться сообщениями, эффективно реализуя разделение.

Основная задача брокера – принимать входящие сообщения из приложений и выполнять над ними определенные действия. Посредники сообщений могут отделять конечные точки, отвечать определенным нефункциональным требованиям и облегчать повторное использование промежуточных функций. Например, брокер сообщений может использоваться для управления очередью рабочей нагрузки или очередью сообщений для нескольких получателей, обеспечивая надежное хранение, гарантированную доставку сообщений и, возможно, управление транзакциями.

Следующие примеры представляют сценарии, которые могут быть обработаны брокером:

– направление сообщения одному или нескольким получателям;

– преобразование сообщения в альтернативное представление;

– выполнение агрегации сообщений, разложив сообщения на несколько сообщений и отправив их по назначению, а затем перекомпоновав ответы в одно сообщение для возврата пользователю;

– взаимодействие с внешним хранилищем, чтобы дополнить сообщение или сохранить его;

– вызов веб-сервисов для получения данных;

– ответ на события или ошибки;

– предоставление контента и маршрутизация сообщений по шаблону.

Принцип работы брокера сообщений изображен на рис. 15.2.

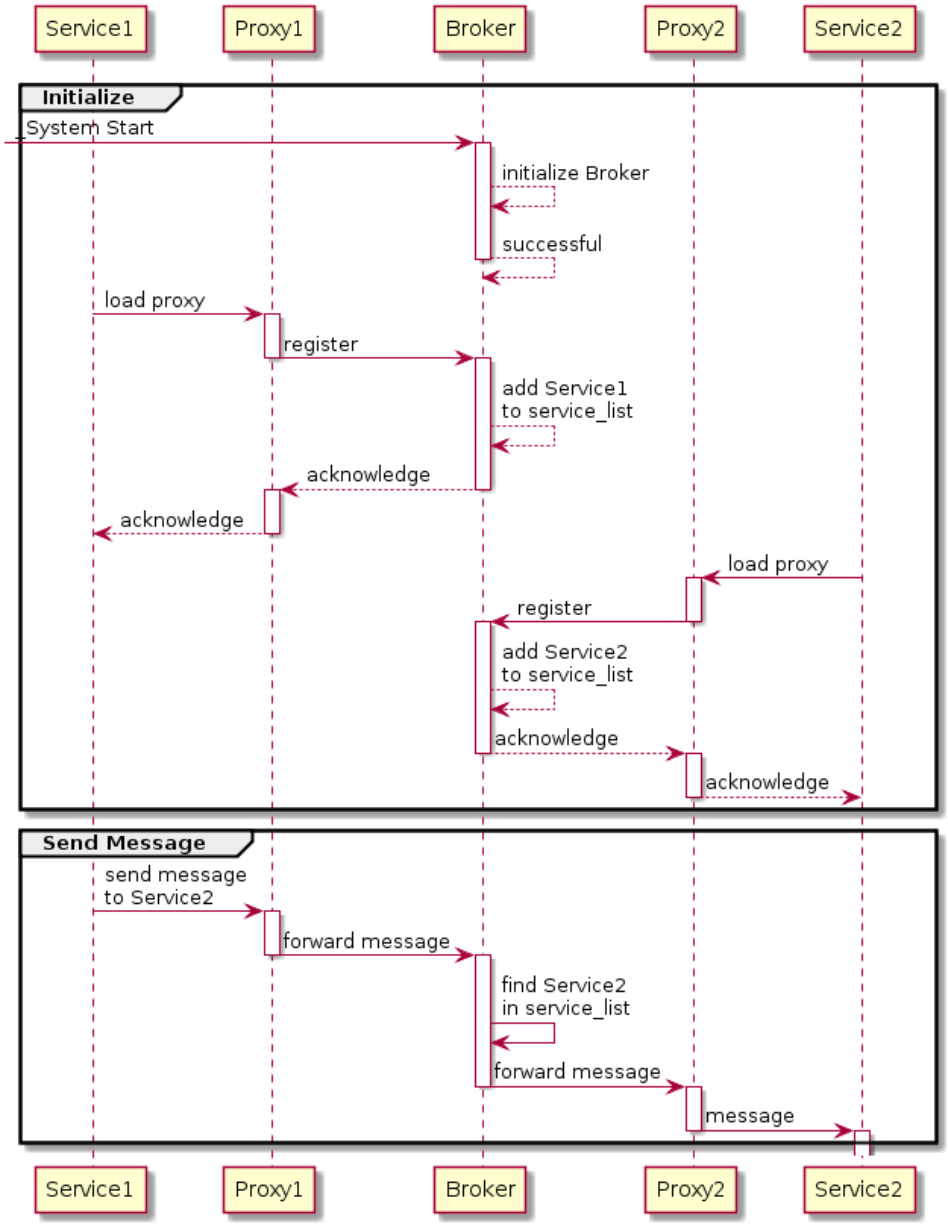



Рис. 15.2. Принцип работы брокера сообщений

### 15.3. Промежуточное программное обеспечение

RabbitMQ — популярный брокер сообщений, и у него есть много мощных функций. Документация на веб-сайте RabbitMQ отличная и имеется много книг. RabbitMQ написан на Erlang, не широко используемом языке программирования, но хорошо приспособленном для таких задач. Конфигурация RabbitMQ устанавливается в файле rabbitmq.config и содержит множество настраиваемых параметров.

RabbitMQ делится на компоненты:

– producer – клиент, который создает сообщения;

– consumer – клиент, который получает сообщения;

– queue – неограниченная по размеру очередь, которая хранит сообщения;

– exchange – компонент, который позволяет переправлять отправляемые в него сообщения на различные очереди.

RabbitMQ — простой в использовании, поддерживает огромное количество платформ для разработки. Он хорошо масштабируется при добавлении большего числа серверов.

RabbitMQ обладает следующими свойствами:

– сообщения, опубликованные в очереди (через обменные пункты);

– несколько потребителей могут подключиться к очереди;

– брокер сообщений распространяет сообщения среди всех доступных потребителей;

– сообщение может быть переадресовано, если потребитель не сработает;

– заказ на доставку гарантирован для очередей с одним потребителем (это невозможно, если очередь имеет несколько потребителей).

Apache Kafka

Apache Kafka — один из важных компонентов этой экосистемы. Разработанная корпорацией LikedIn и названная в честь знаменитого писателя служба обмена сообщениями Kafka обладает такими ценными качествами, как скорость работы, масштабируемость, способность секционировать и множество раз фиксировать одни и те же данные в памяти. Перечислим основные отличия Kafka от традиционных систем обмена сообщениями:

– служба Kafka изначально создавалась и позиционируется как распределенная программа – следовательно, она приспособлена к масштабированию;

– система обладает отличной производительностью – как в случае публикации сообщений, так и в случае подписки на них;

– Kafka сохраняет сообщения на диске и, таким образом, может использоваться для пакетной передачи данных;

– Kafka имеет высокую производительность и гарантирует нулевое время простоя и нулевую потерю данных.

Kafka имеет схожие компоненты с компонентами RabbitMQ, однако имеет и уникальные компоненты, например:

– Topic – поток сообщений, относящихся к определенной категории, называется темой. Topic разбит на разделы. Каждый такой раздел содержит сообщения в неизменной упорядоченной последовательности. Раздел реализован как набор файлов сегментов одинакового размера;

– Partition – раздел потока сообщений.

Пример работы брока сообщений Kafka изображен на рис. 15.3.

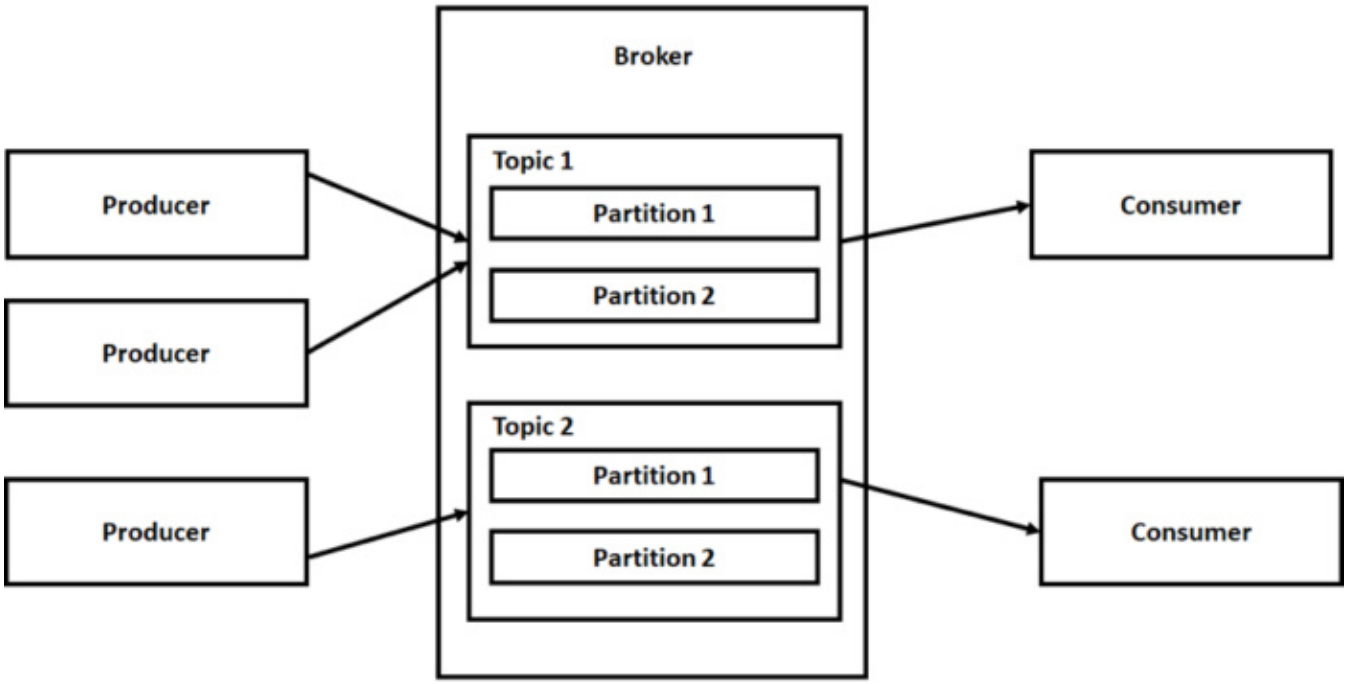



Рис. 15.3. Работа брокера Kafka

## 15.4. IBM Message Broker

IBM Message Broker - это решение для ESB, поддерживающее широкий спектр коммуникационных протоколов и форматов сообщений. Оно обеспечивает выполнение преобразований, маршрутизации и дополнения с использованием различных технологий. Ключевой компонент WebSphere Message Broker - встроенный инструментарий разработки программного обеспечения. Инструментарий WebSphere Message Broker является плагином Eclipse, основанным на IBM Rational Application Developer.

Основные преимущества:

– ориентирован на технологическую интеграцию;

– высокопроизводительная транзакционная обработка потока электронных документов в виде сообщений;

– поддерживаемые протоколы: MQ, JMS, HTTP(S), WebServices, MQe, Pub/Sub, TCP, File/(S)FTP, JDBC/ODBC и т.д.;

– сервис форматирования сообщений – домены сообщений: XML, Text, Binary, Tag delimited, стр-ры Java/C/COBOL и т.д.;

– концепция визуального конструирования процессов обработки – message flows – поток обработки сообщения из узлов (nodes-готовые обработчики) – соединение терминалов обработчиков (входные и выходные точки обработчиков) и определение параметров;

– программный язык для работы с сообщениями – ESQL- расширение процедурного SQL, Java, C/C++ .NET, PHP;

– вставка собственных компонент и парсеров на Java и C;

– контроль доступа к среде исполнения.

### 15.5. Контрольные задания

#### 15.5.1. Вопросы для самопроверки

1. Что такое сервисная шина предприятия?
2. Основные компоненты ESB.
3. Основные функции ESB.
4. Основные задачи современной сервисной шины предприятия.
5. Промежуточное программное обеспечение. Его назначение и примеры.
6. Что такое IBM Message Broker?

## 16. HTTP И БЕЗОПАСНОСТЬ. ПРИНЦИПЫ РАБОТЫ ЧЕРЕЗ HTTP, ЗАЩИТА И УГРОЗЫ, HTTPS, КРОСС-ДОМЕННЫЕ ЗАПРОСЫ

Более чем за три десятка лет Интернет проник во все области деятельности человечества: его используют для того, чтобы читать книги, смотреть видео, любоваться изображениями, узнавать погоду, слушать музыку. Почти весь бизнес так или иначе использует Сеть для передачи информации о сотрудниках, поступлениях товаров на склады и перевода денежных средств. Большая часть данных, которая передаётся через Сеть, использует протокол HTTP в качестве контейнера. Каждый раз, когда вы заходите на сайт, ваш браузер посылает до нескольких десятков HTTP запросов. HTTP используется для загрузки файлов из сети, программы скачивают обновления, используя этот протокол, даже интернет-радио не обходится без него.

Но так ли безопасна работа в Интернете? У всего бывают недостатки, в том числе и у протокола HTTP. При его использовании пользователи подвергаются ряду угроз, которые будут рассмотрены ниже.

Но и прогресс в области обеспечения безопасности также не стоит на месте. Одним из основных достижений в этой области является расширение протокола, названное HTTPS, которое также будет рассмотрено далее.

### 16.1. Принципы работы через HTTP

HTTP - это протокол, позволяющий получать различные ресурсы, например, HTML-документы. Протокол HTTP лежит в основе обмена данными в Интернете. HTTP является протоколом клиент-серверного взаимодействия, что означает инициирование запросов к серверу самим получателем, обычно

веб-браузером. Полученный итоговый документ будет реконструирован из различных субдокументов, например, из отдельно полученного текста, описания структуры документа, изображений, видеофайлов, скриптов и многого другого.

Хотя HTTP был разработан еще в начале 1990-х годов, за счет своей расширяемости в дальнейшем он все время совершенствовался. HTTP является протоколом прикладного уровня, который чаще всего использует возможности другого протокола - TCP (или TLS - защищённый TCP) - для пересылки своих сообщений, однако любой другой надежный транспортный протокол теоретически может быть использован для доставки таких сообщений. Благодаря своей расширяемости он используется не только для получения клиентом гипертекстовых документов либо изображений и видео, но и для передачи контента серверам, например, с помощью HTML-форм. HTTP также может быть использован для получения только частей документа с целью обновления веб-страницы по запросу.

Клиенты и серверы взаимодействуют, обмениваясь индивидуальными сообщениями (а не потоком данных). Сообщения, отправленные клиентом, обычно веб-браузером, называются запросами, а сообщения, отправленные сервером, называются ответами. Место HTTP в иерархии сетевых и веб-технологий изображено на рис. 16.1.

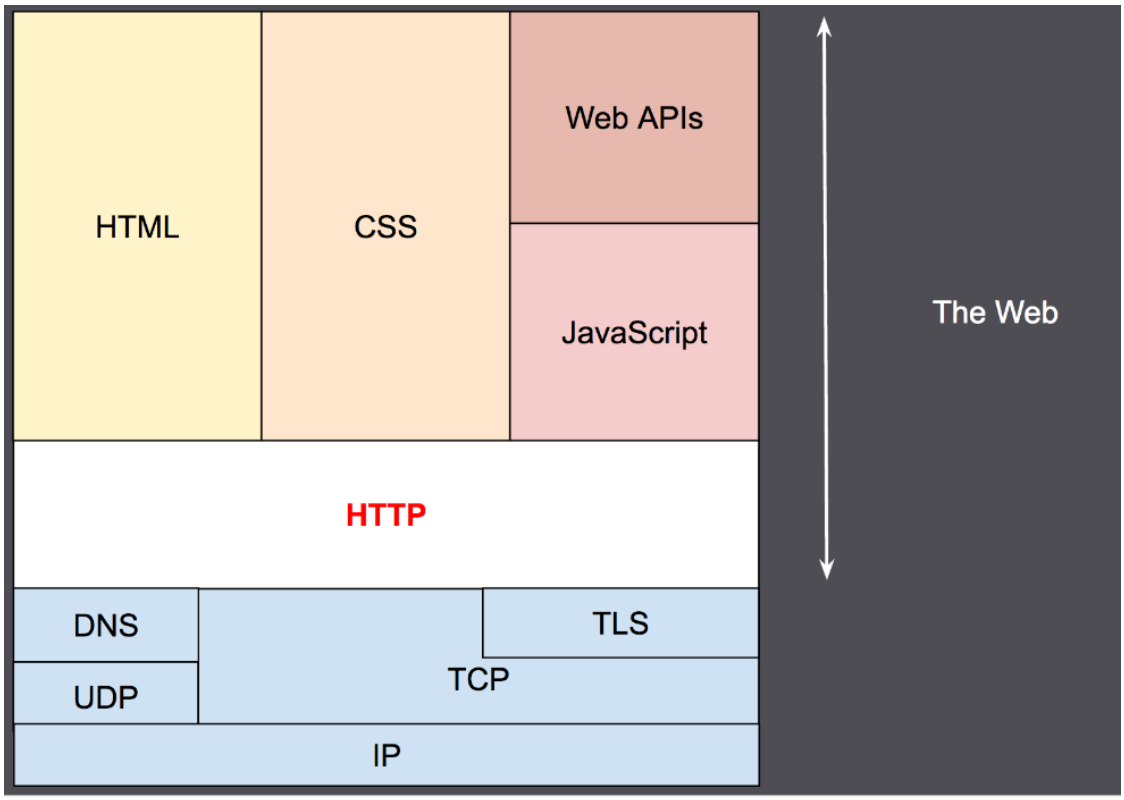



Рис. 16.1. Место HTTP в иерархии сетевых и веб-технологий

HTTP - это клиент-серверный протокол, то есть запросы отправляются какой-то одной стороной — юзер-агентом (user-agent) (либо прокси вместо него). Чаще всего в качестве юзер-агента выступает веб-браузер, но им может быть кто угодно, например робот, путешествующий по Сети для пополнения и обновления данных индексации веб-страниц для поисковых систем.

Каждый индивидуальный запрос (англ. request) отправляется серверу, который обрабатывает его и возвращает ответ (англ. response). Между этими запросами и ответами существуют многочисленные посредники, называемые прокси, которые выполняют различные операции и работают как шлюзы или кэш, например. Пример организации сообщений по HTTP изображено на рис. 16.2.

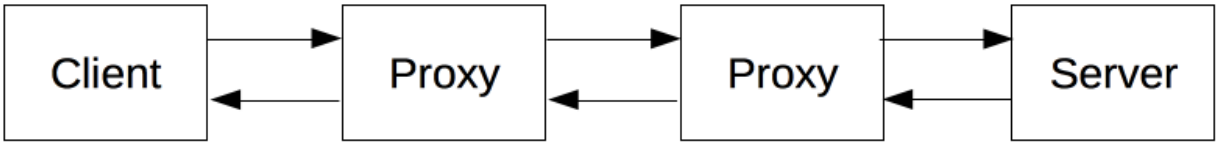



Рис. 16.2. Пример организации сообщения по HTTP

В реальности, между браузером и сервером гораздо больше различных устройств-посредников, которые играют какую-либо роль в обработке запроса: роутеры, модемы и так далее [27].

Механизмов безопасности передачи данных в HTTP практически нет, а то, что есть, трудно назвать механизмом, отвечающим за безопасность.

По сути, протокол HTTP не предусматривает никаких механизмов защиты данных пользователя и веб-серверов, если не считать базовой схемы аутентификации клиентов на сервере, кодирования информации и положений безопасности в стандарте HTTP 1.1.

## 16.2. Идентификация клиентов на сервере. Безопасность информации, хранящейся на HTTP сервере

Если на вашем сервере хранится информация, которая должна быть доступна только определенному кругу людей, то вы можете использовать базовый метод аутентификации клиентов на сервере, этот метод не гарантирует безопасность данных на вашем сервере. Но стоит заметить, что протокол HTTP никак не препятствует шифровать ваши данные на сервере и использовать другие методы для повышения безопасности хранения данных.

Самый существенный недостаток базовой аутентификации на сервере и в то же время самая большая брешь в безопасности HTTP сервера заключается в том, что логин и пароль пользователя передаются в HTTP сообщении в незашифрованном виде, поэтому вам стоит использовать дополнительные методы шифрования при использовании базовой аутентификации.

Брешь в безопасности не ограничивается тем, что кто-то может перехватить HTTP запросы или HTTP ответы с незашифрованными данными, но и в том, что пользователь может вместо желаемого ресурса попасть на вредоносный клон, в котором он пройдет аутентификацию, тем самым оставив свои учетные данные злоумышленнику.

## 16.3. Безопасность и логи HTTP сервера

Сервер обычно записывает в специальный файл все запросы пользователя и ответы на них, то есть ведет лог. На некоторых сайтах лог запросов может представлять интерес для злоумышленников. Поэтому реализации серверов должны позаботиться о безопасности лога HTTP запросов.

Протокол HTTP - это универсальный протокол передачи данных, а это значит, то по данному протоколу можно передавать, в принципе, любую информацию. Также у HTTP протокола нет никаких средств или механизмов, которые могли бы отрегулировать содержимое HTTP сообщения (HTTP объект). Собственно, передача содержимого и безопасность передачи содержимого лежит на клиентском и серверном программном обеспечении, поэтому к небезопасным полям заголовка HTTP сообщения можно отнести: Server, Via, Referer, From.

Подмена DNS-адресов и HTTP протокол

Система DNS (Domain Name System) преобразует доменное имя (например, www.test.com) в его IP-адрес (например, 192.168.0.1) и наоборот. Данная атака использует технологию отправки фальшивых ответов на DNS-запросы жертвы. Атака основывается на двух основных методах.

1. Подмена DNS ID (DNS ID Spoofing).

Заголовок пакета DNS-протокола содержит идентификационное поле для соответствия запросов и ответов. Целью подмены DNS ID является посылка своего ответа на DNS-запрос до того, как ответит настоящий DNS-сервер. Для выполнения этого нужно спрогнозировать идентификатор запроса. Локально это реализуется простым прослушиванием сетевого трафика. Однако удаленно выполнить эту задачу гораздо сложнее. Существуют различные методы:

– проверка всех доступных значений идентификационного поля. Не очень практично, поскольку общее количество возможных значений составляет 65535 (размер поля 16 бит);

– посылка нескольких сотен DNS-запросов в правильном порядке. Очевидно, что этот метод не очень надежен;

– нахождение сервера, генерирующего прогнозируемые идентификаторы (например, увеличивающиеся на 1). Этот тип уязвимости присущ некоторым версиям Bind и системам Windows 9x.

В любом случае необходимо ответить до настоящего DNS-сервера. Этого можно достичь, например, выполнив против сервера атаку типа "отказ в обслуживании".

Атака требует выполнения четырех шагов:

– атакующий шлет DNS-запрос от имени www.attaquant.com к DNS-серверу домена cible.com, как показано на рис. 16.3.

–


Рис. 16.3. DNS-запрос посланный к ns.cible.com

– целевой DNS-сервер перенаправляет запрос к серверу домена attaquant.com;

– злоумышленник прослушивает запросы на предмет получения идентификаторов (в нашем примере ID равен 100);

– злоумышленник фальсифицирует IP-адрес, соответствующий имени. В примере машина-жертва это www.spoofed.com, адрес которой должен быть 192.168.0.1.

Злоумышленник посылает DNS-запрос на разрешение имени www.spoofed.com серверу ns.cible.com. И сразу же шлет группу фальсифицированных DNS-ответов (передавая в качестве IP-адреса один из адресов злоумышленника - 10.0.0.1) на свой же запрос с подмененным IP-адресом источника на адрес одного из DNS-серверов домена spoofed.com. Пример подмены DNS ID изображен на рис. 16.4.

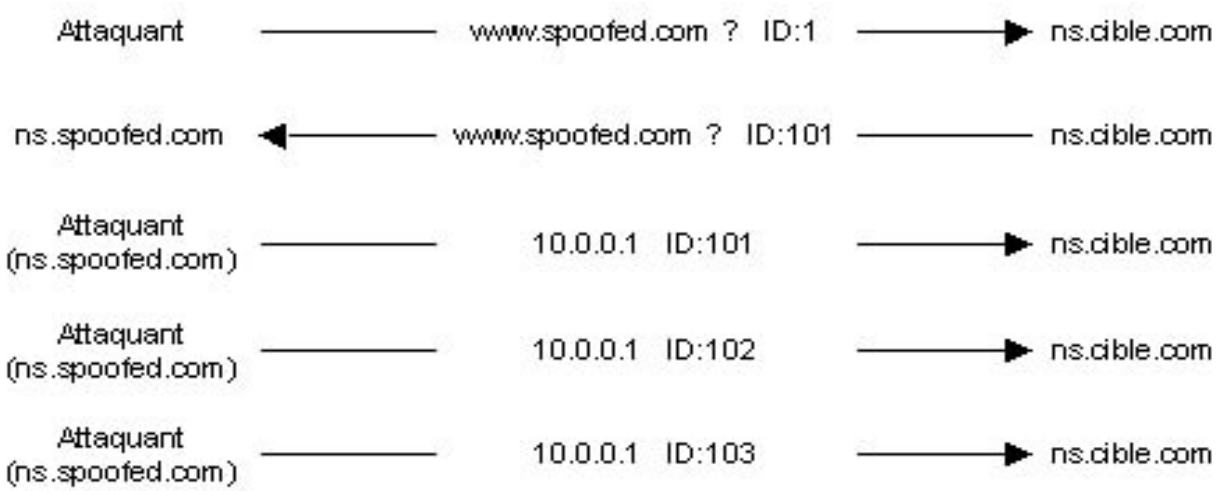



Рис. 16.4. Подмена DNS ID

В результате кэш целевого DNS-сервера будет содержать соответствие, необходимое злоумышленнику и клиентам, запрашивающим адрес www.spoofed.com, будет сообщен адрес машины злоумышленника. На ней может быть размещена копия настоящего сайта, с помощью которого злоумышленник может красть конфиденциальную информацию.

2. Изменение DNS Cache Poisoning.

DNS-сервера используют кеш для хранения результатов предыдущих запросов в течение некоторого времени. Это делается, чтобы избежать постоянных повторов запросов к авторизированным серверам соответствующих доменов. Второй вариант атаки, направленной на подмену DNS, заключается в изменении кэша DNS сервера.

Пример: используем те же данные, что и в предыдущем примере. Ключевые моменты этого варианта атаки:

– послать DNS-запрос на разрешение имени www.attaquant.com DNS-серверу домена cible.com;

– целевой DNS-сервер шлет запрос на разрешение имени www.attaquant.com DNS-серверу злоумышленника;

– DNS-сервер злоумышленника шлет ответ с фальсифицированными записями, что позволяет задавать имени соответствие с IP-адресом злоумышленника. Например, сайт www.cible.com может иметь фальсифицированную DNS-запись, соответствующую IP-адресу сайта www.attaquant.com.

## 16.4. Межсайтовый скриптинг

XSS (англ. Cross-Site Scripting — «межсайтовый скриптинг») — тип атаки на веб-системы, заключающийся во внедрении в выдаваемую веб-системой страницу вредоносного кода (который будет выполнен на компьютере пользователя при открытии им этой страницы) и взаимодействии этого кода с веб-сервером злоумышленника. Является разновидностью атаки «Внедрение кода». Схема данной атаки представлена на рис. 16.5.

Специфика подобных атак заключается в том, что вредоносный код может использовать авторизацию пользователя в веб-системе для получения к ней расширенного доступа или для получения авторизационных данных пользователя.

Вредоносный код может быть вставлен в страницу как через уязвимость в веб-сервере, так и через уязвимость на компьютере пользователя.

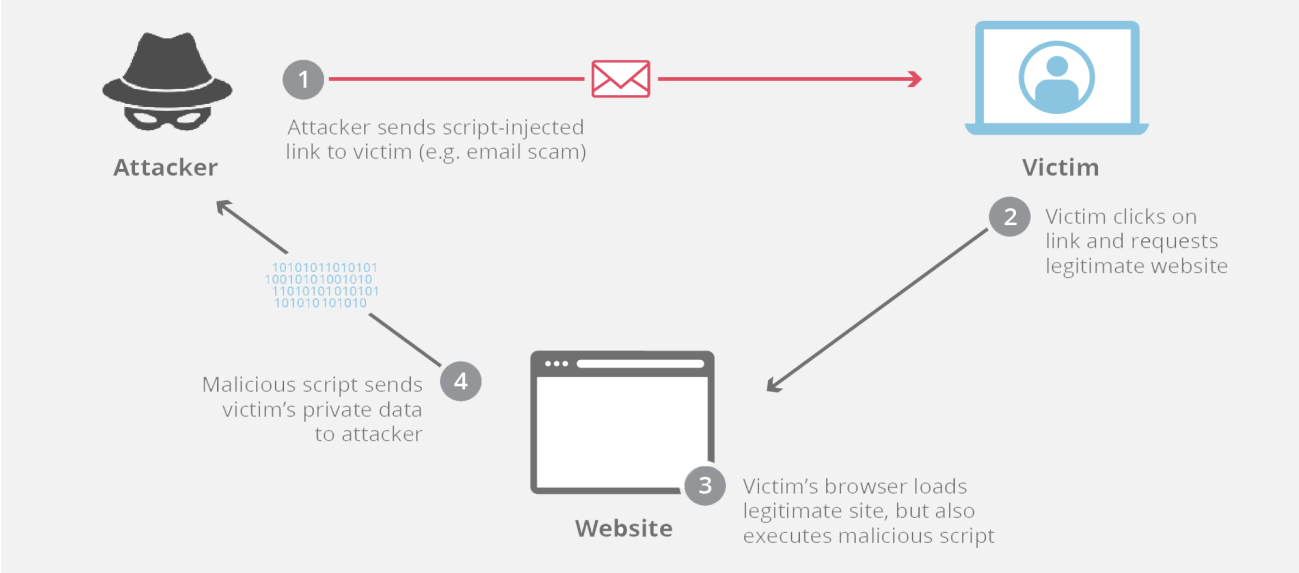



Рис. 16.5. Схема XSS атаки

## 16.5. Кросс-доменные запросы

Межсайтовая подделка запроса, или же CSRF (англ. Cross Site Request Forgery — «межсайтовая подделка запроса», также известна как CSRF) — вид атак на посетителей веб-сайтов, использующий недостатки протокола HTTP (схема атаки представлена на рис. 16.6). Если жертва заходит на сайт, созданный злоумышленником, от её лица тайно отправляется запрос на другой сервер (например, на сервер платёжной системы), осуществляющий некую вредоносную операцию (например, перевод денег на счёт злоумышленника).

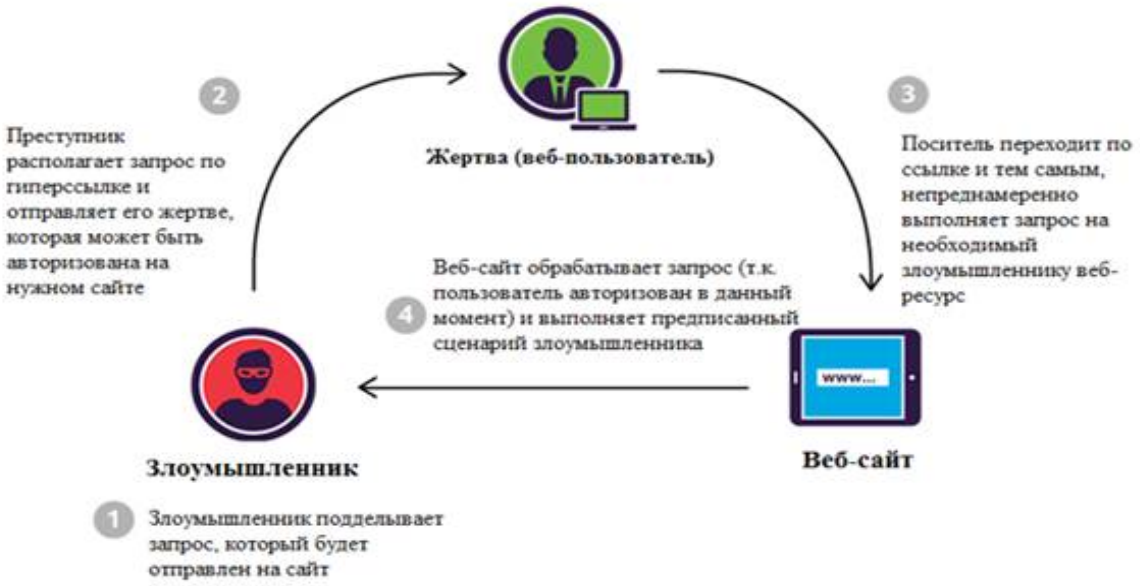



Рис. 16.6. Схема атаки с использованием кросс–доменных запросов

Данный тип атак, вопреки распространённому заблуждению, появился достаточно давно: первые теоретические рассуждения появились в 1988 году, а первые уязвимости были обнаружены в 2000 году. А сам термин ввёл Peter Watkins в 2001 году.

Основное применение CSRF — вынуждение выполнения каких-либо действий на уязвимом сайте от лица жертвы (изменение пароля, секретного вопроса для восстановления пароля, почты, добавление администратора и т. д.). Также с помощью CSRF возможна эксплуатация отражённых XSS, обнаруженных на другом сервере.

Расширение HTTP - HTTPS

HTTPS (аббр. от англ. HyperText Transfer Protocol Secure) — расширение протокола HTTP для поддержки шифрования в целях повышения безопасности. Данные в протоколе HTTPS передаются поверх криптографических протоколов SSL или TLS. В отличие от HTTP с TCP-портом 80, для HTTPS по умолчанию используется TCP-порт 443.

Протокол был разработан компанией Netscape Communications для браузера Netscape Navigator в 1994 году.

HTTPS не является отдельным протоколом. Это обычный HTTP, работающий через шифрованные транспортные механизмы SSL и TLS. Он обеспечивает защиту от атак, основанных на прослушивании сетевого соединения, — от снифферских атак и атак типа man-in-the-middle, при условии, что будут использоваться шифрующие средства и сертификат сервера проверен и ему доверяют.

Традиционно на одном IP-адресе может работать только один HTTPS-сайт. Для работы нескольких HTTPS-сайтов с различными сертификатами применяется расширение TLS под названием Server Name Indication(SNI).

## 16.6. Контрольные задания

### 16.6.1. Вопросы для самопроверки

1. Что такое HTTP?
2. Для чего используется TCP?
3. Привести пример организации сообщения по HTTP протоколу.
4. Безопасность в HTTP протоколе.
5. Для чего используется DNS адресса?
6. Что такое XSS и для чего используется?
7. Что такое CSRF и для чего используется?
8. Расширение HTTP протокола.

## ЛАБОРАТОРНЫЕ РАБОТЫ

Лабораторная работа № 1

Установка и настройка Kafka

Цель работы: изучение основных понятий в Apache Kafka и получение практических навыков установки и настройки этого программного обеспечения.

Задание на лабораторную работу:

1. Ознакомьтесь с теоретической частью, изложенной в данном пособии.
2. Установите Zookeeper.
3. Установите Kafka.
4. Проверьте работоспособность, отправив сообщение через командную строку ОС.
5. Получите отправленное сообщение с помощью командной строки ОС.
6. Сделайте вывод.

Лабораторная работа № 2

Разработка приложений для работы с Apache Kafka

Цель работы: углубленное изучение аспектов использования Apache Kafka в современных приложениях.

Задание на лабораторную работу:

1. Ознакомьтесь с теоретической частью, изложенной в данном пособии.
2. Создайте приложение для отправки сообщений согласно выбранному варианту.
3. Создайте первое приложение для получения сообщений согласно выбранному варианту.
4. Создайте второе приложение для получения сообщений согласно выбранному варианту.
5. Сделайте вывод.

Лабораторная работа № 3

Установка и настройка RabbitMQ

Цель работы: изучение основных понятий в RabbitMQ и получение практических навыков установки и настройки этого программного обеспечения.

Задание на лабораторную работу:

1. Ознакомьтесь с теоретической частью, изложенной в данном пособии.
2. Установите RabbitMQ.
3. Проверьте работоспособность установленного программного обеспечения.
4. Сделайте вывод.

Лабораторная работа № 4

Разработка приложений для работы с RabbitMQ

Цель работы: углубленное изучение аспектов использования RabbitMQ в современных приложениях.

Задание на лабораторную работу:

1. Ознакомьтесь с теоретической частью, изложенной в данном пособии.
2. Создайте приложение для отправки сообщений согласно выбранному варианту.
3. Создайте первое приложение для получения сообщений согласно выбранному варианту используя метод синхронного вычитывания сообщений.
4. Создайте второе приложение для получения сообщений согласно выбранному варианту, используя асинхронный метод вычитывания сообщений.
5. Сделайте вывод.

| Вариант | Что публикует приложение | Что делает первое приложение для получения | Что делает второе приложение для получения |
|---|---|---|---|
| 1 | Публикует логи | Вычитывает логи и выводит их в консоль | Считает количество логов по категориям (логи с ошибками, логи с полезной информацией) |
| 2 | Публикует статьи | Вычитывает статьи и выводит их в консоль | Считает статьи по категориям (научные, разлекательные) |
| 3 | Публикует метрики | Вычитывает метрики и выводит их в консоль | Считает сумму, максимальное, минимальное и среднее значение публикуемых метрик |
| 4 | Публикует текстовые файлы | Вычитывает содержимое файла | Сохраняет файлы на диске |

Лабораторная работа № 5

Создание RESTful приложения

Цель работы: изучение основных понятий REST и получение практических навыков создания RESTful приложения на языке Java.

Задание на лабораторную работу:

1. Ознакомьтесь с теоретической частью, изложенной в данном пособии.
2. Установите IDE (например, IntelliJ IDEA).
3. Напишите простое RESTful приложение.
4. Добавьте в свое приложение дополнительный параметр.
5. Сделайте вывод.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК


1. Агальцов, В.П. Базы данных: в 2-х т. Т. 2: Распределенные и удаленные базы данных: учебник / В.П. Агальцов. - М.: ИД ФОРУМ, НИЦ ИНФРА-М, 2013. - 272 с.
2. Агальцов, В.П. Базы данных: 2-х т. Т. 1: Локальные базы данных: учебник / В.П. Агальцов. - М.: ИД ФОРУМ, НИЦ ИНФРА-М, 2013. - 352 с.
3. Советов, Б.Я. Базы данных: теория и практика: учебник для бакалавров / Б.Я. Советов, В.В. Цехановский, В.Д. Чертовской. - М.: Юрайт, 2013. - 463 с.
4. Дейт, К. Дж. Введение в системы баз данных / К. Дж. Дейт. – Спб.: Вильямс, 2018. – 1314 с.
5. Грофф, Дж. SQL Полное руководство / Дж. Грофф, П. Вайнберг, Э. Оппель. – Спб.: Вильямс, 2017. – 942 с.
6. Дейт К. Руководство по реляционной СУБД DB2 / К. Дейт. - М.: Финансы и статистика, 2008. - 320 с.
7. Дейт К. Дж. Введение в системы баз данных / К. Дж. Дейт. - 6-е изд. - М.: Вильямс. 2006. – 317 с. - ISBN 5-8459-0788-8.
8. Куцевич, Н.И. Интеграция информационных систем предприятия [Электронный ресурс] / НОУ ИНТУИТ: https://www.intuit.ru/studies/courses/13862/1259/lecture/24012
9. Старыгин А. XML. Разработка WEB-приложений / А. Старыгин. – СПб: БХВ-Петербург, 2006. – 559 с.
10. Sai Srinivas Sriparasa, JavaScript and JSON Essentials / Sai Srinivas Sriparasa – Packt Publishing, 2013. – 120p.
11. JSON [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/JSON.
12. Введение в JSON [Электронный ресурс]: Режим доступа: https://www.json.org/json-ru.html.
13. Трескин М. Введение в AMQP [Электронный ресурс] / М. Трескин. - Режим доступа: https://habr.com/ru/post/64192/.
14. Понимание AMQP, Режим доступа: http://spring-projects.ru/understanding/amqp.
15. Гагрина, Л.Г. Введение в архитектуру программного обеспечения [Электронный ресурс] / Студенческие реферативные статьи и материалы: https://studref.com/329522/informatika/patterny_metodu_integratsii
16. Кузнецов С. Когда, как, что и зачем стоит интегрировать? [Электронный ресурс] / База знаний о корпоративных хранилищах данных: https://www.prj-exp.ru/integration/why_should_integrate.php
17. Дейт К. Руководство по реляционной СУБД DB2 / К. Дейт. - М.: Финансы и статистика, 2008. - 320 с.

18. Дейт К. Дж. Введение в системы баз данных / К. Дж. Дейт. - 6-е изд. - М.: Вильямс. 2006. – 317 с. - ISBN 5-8459-0788-8.

19. Кириллов В.В. Основы проектирования реляционных баз данных: учеб. пособие / В.В. Кириллов. - СПб.: ИТМО, 2008. - 90 с.

20. Когаловский М.Р. Энциклопедия технологий баз данных / М.Р. Когаловский. – М.: Финансы и статистика, 2005. - ISBN 5-279-02276-4.

21. Конноли Т. Базы данных. Проектирование, реализация и сопровождение. Теория и практика / Т. Коноли, Л. Бегг, А. Страчан. - 3-е изд. М.: Вильямс, 2005.

22. Леонтьев В.П. ПК: универсальный справочник пользователя / В.П. Леонтьев. - М., 2005. – 251 с. - ISBN: 5-8459-0527-3.

23. Коннолли Т. Базы данных. Проектирование, реализация и сопровождение. Теория и практика / К. Бегг, А. Страчан. - 2-е изд. – СПб.: Вильямс, 2007. – 1120 с. - ISBN 5-8459-0109-2.

24. Корнеев В.В. Базы данных. Интеллектуальная обработка информации / А.Ф. Гареев, С.В. Васютин, В.В. Райх. - 2-е изд. – М.: Изд-во Молгачева С.В., 2005. – 494 с. - ISBN 5-89251-100-6.

25. Мейер М. Теория реляционных баз данных / М. Мейер. - М.: Мир, 2007. - 608 с.

26. Spring Remoting, Режим доступа: https://www.baeldung.com/spring-remoting-amqp.

27. Кононов Р. Очередь сообщений (Message Queue) [Электронный ресурс] / Р. Кононов. Режим доступа: https://habr.com/ru/post/165981.

**ОГЛАВЛЕНИЕ**

**Учебное издание**

**Рындин** Александр Алексеевич
**Саргсян** Эрик Ромович

# СОВРЕМЕННЫЕ СТАНДАРТЫ ИНФОРМАЦИОННОГО ВЗАИМОДЕЙСТВИЯ СИСТЕМ

Учебное пособие

Редактор Г. В. Биндюкова

Подписано в печать 26.04.2021.
Формат 60x84/16. Бумага для множительных аппаратов.
Уч.-изд. л. 9,0. Усл. печ. л. 8,4. Тираж 350 экз.
Зак №

ФГБОУ ВО «Воронежский государственный технический университет»
394026 Воронеж, Московский просп., 14

Участок оперативной полиграфии издательства ВГТУ
394026 Воронеж, Московский просп., 14