Распределенные информационно-аналитические системы

Лекция № 3.

Распределенные события и транзакции. Безопасность в распределенных системах

Профессор кафедры КБ-2: д.т.н. Шатовкин Р.Р.

Учебные цели:

Изучить основы научных знаний по распределенным событиям; распределенным транзакциям; безопасности в распределенных системах; промежуточных средах в Microsoft.NET Framework.

Учебные вопросы:

1. Распределенные события.

2. Распределенные транзакции.

3. Безопасность в распределенных системах.

4. Промежуточные среды в Microsoft.NET Framework.

1. Распределенные события

При разработке программного обеспечения достаточно часто возникает потребность получать извещения о каких-либо событиях, возникающих асинхронно, то есть в некоторые произвольные моменты времени.

В распределенных системах так же может возникнуть необходимость использования таких извещений, получаемых от удаленной системы.

Можно выделить ***два подхода к обработке событий***:

- тесно связанные события;
- слабо связанные события.

При тесно связанном событии происходит прямое уведомление одной стороны другой стороной. Хотя этот метод можно использовать, например, вместе с однонаправленным асинхронным вызовом, ему свойственен ***ряд недостатков, ограничивающих его применение в распределенных системах***:

- обе компоненты системы должны выполняться одновременно;
- для уведомления нескольких компонент об одном событии уведомляющей стороной должны использоваться механизмы для ведения списка получателей событий;
- затруднена фильтрация или протоколирование событий.

Поэтому в распределенных системах так же применяются слабо связанные события, когда источники события (издатели) не взаимодействуют напрямую с получателями событий (подписчиками). Промежуточная среда в этом случае должна предоставить сервис, позволяющий подписчику подписаться на какое-либо событие или отказаться от подписки, а издателю – инициировать событие для рассылки подписчикам (рисунок 1).

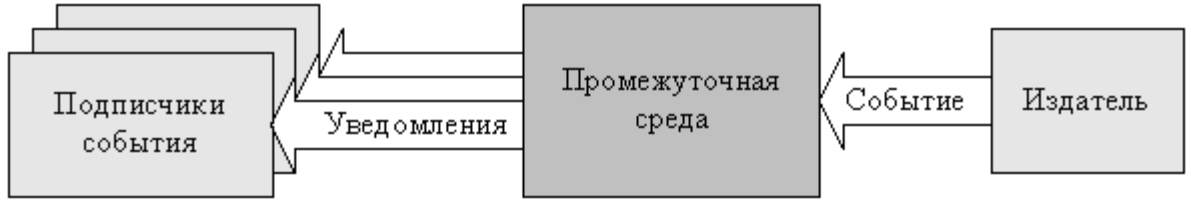

Рисунок 1 – Подписчики и издатели слабосвязанных событий

При использовании слабосвязанных событий подписчики, издатели и менеджер событий могут располагаться на различных компьютерах.

Само событие может быть реализовано как, напри-мер, вызов менеджером событий некоторого зарегистрированного метода удаленного объекта.

2. Распределенные транзакции

Транзакция – последовательность операций с какими-либо данными, которая либо успешно выполняется полностью, либо не выполняется вообще. В случае невозможности успешно выполнить все действия происходит возврат к первоначальным значениям всех измененных в течение транзакции данных (***откат транзакции***).

Транзакция должна обладать следующими качествами:

Атомарность. Транзакция выполняется по принципу «все или ничего».

Согласованность. После успешного завершения или отката транзакции все данные находятся в согласованном состоянии, их логическая целостность не нарушена.

Изоляция. Для объектов вне транзакции не видны промежуточные состояния, которые могут принимать изменяемые в транзакции данные. С точки зрения «внешних» объектов, до успешного завершения транзакции они должны иметь то же состояние, в котором находились до ее начала.

Постоянство (долговечность). В случае успешности транзакции сделанные изменения должны иметь постоянный характер (то есть сохранены в энергонезависимой памяти).

Транзакции являются основой приложений, работающих с базами данных, однако в распределенной системе может быть недостаточно использования только транзакций систем управления базами данных. Например, в распределенной системе в транзакции может участвовать несколько распределенных компонент, работающих с несколькими независимыми базами данных (рисунок 2).

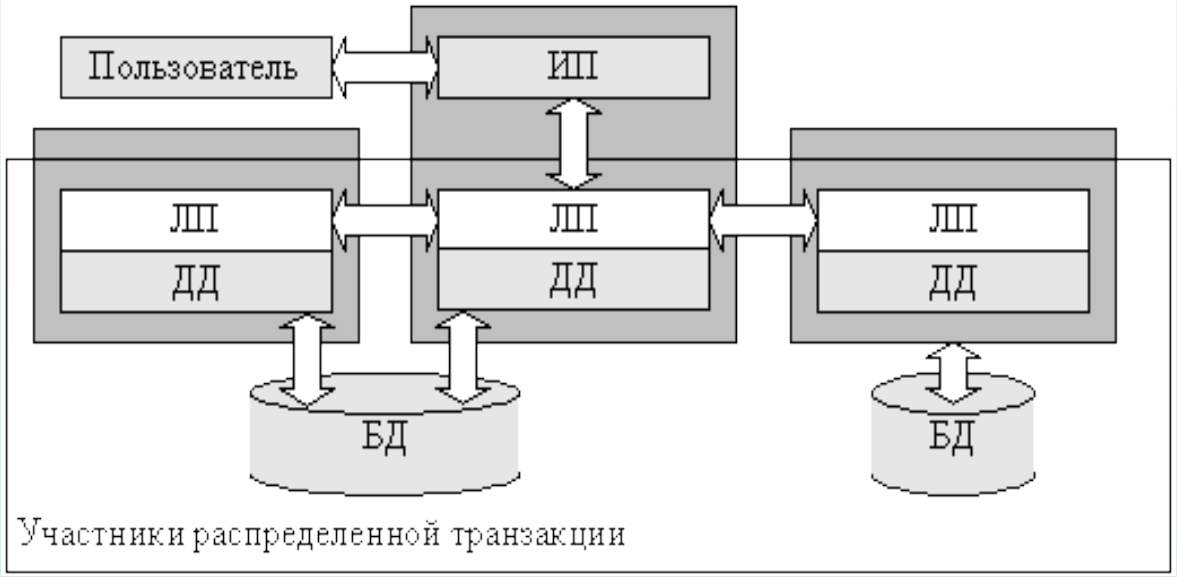

Рисунок 2 – Распределенная транзакция

Распределенной называется ***транзакция***, охватывающая операции нескольких взаимодействующих компонент распределенной системы. Каждая из этих компонент может работать с какими-либо системами управления базами данных или иными службами, например, использовать очереди сообщений, или даже работать с файлами.

При откате транзакции все эти операции должны быть отменены.

Для этого необходимо выполнение ***двух условий***:

- промежуточная среда должна поддерживать управление распределенными между несколькими компонентами транзакциями;

- компоненты распределенной системы не должны работать с какими-либо службами или ресурсами, которые не могут участвовать в транзакции.

Распределенные транзакции являются важнейшим элементом поддержания целостности данных в распределенной системе. Поэтому для более широкого их применения промежуточная среда может содержать механизмы, которые при необходимости (и определенных затратах времени на написание кода) позволят использовать в распределенных транзакциях внешние службы, не поддерживающие транзакции. Такой механизм называется ***компенсирующим менеджером ресурса*** (***compensating resource manager***). Компенсация в данном случае означает возврат ресурса к первоначальному состоянию при откате транзакции.

В настоящее время происходит формирование и стандартизация еще одного понятия, связанного с поддержкой целостности данных – ***хозяйственной деятельности*** (***business activity***) применительно к распределенным системам.

Деятельность обычно является отражением некоторого реального процесса, например, покупки в магазине: от оформления заказа до подтверждения доставки курьером. Деятельность может включать в себя транзакции (оформление заказа покупателя, заказ товара у поставщика, и так далее – до подтверждения доставки покупателем). В отличие от транзакции, время жизни которой предполагается коротким, деятельность может длиться в течение очень долгого времени (например, месяца). Деятельность может поддерживать отмену сделанных изменений (например, оформление возврата товара поставщика при отказе покупателя) путем использования компенсирующих задач.

В мире микросервисов транзакция распределяется между множеством сервисов, которые вызываются в некоторой последовательности для завершения всей транзакции.

На рисунке 3 представлена монолитная система Интернет-магазина, в которой используются транзакции.

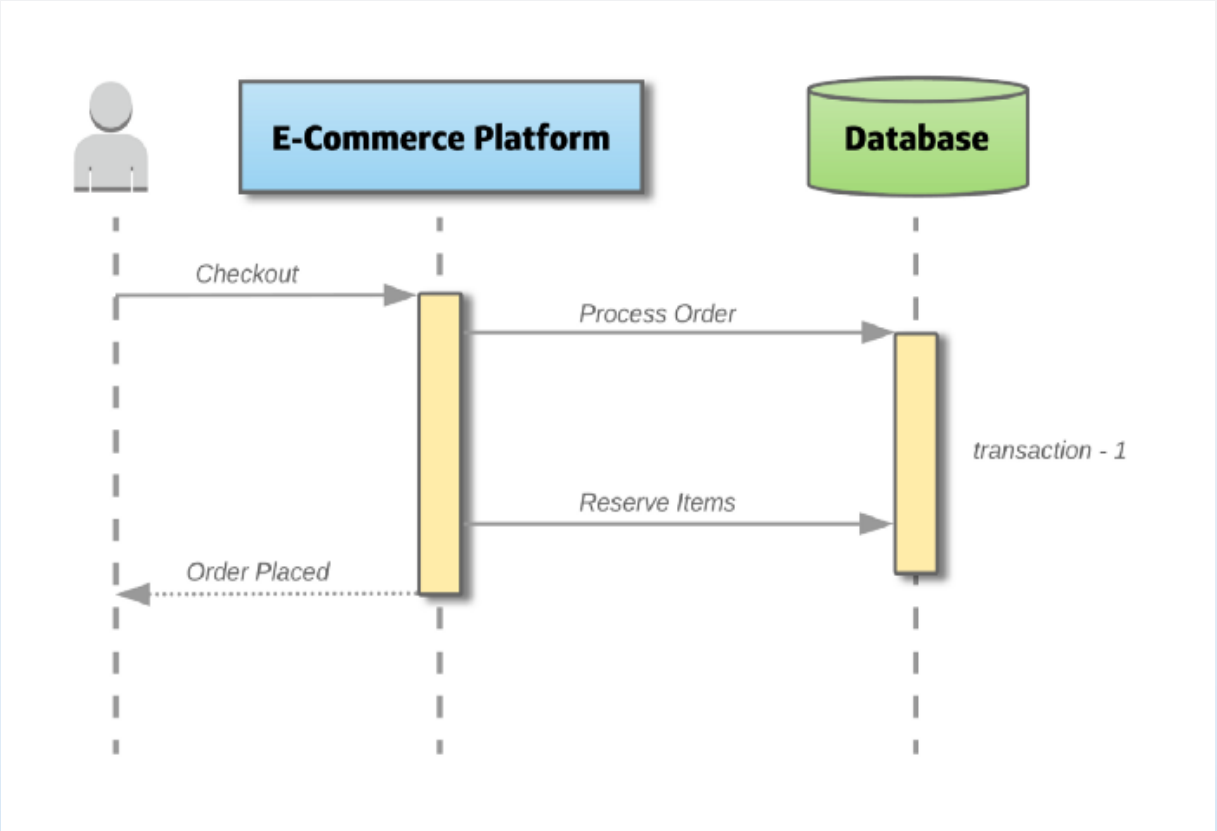

Рисунок 3 – Транзакция в монолите

Если в вышеприведенной системе пользователь отправляет к платформе запрос на заказ (Checkout), то платформа создает локальную транзакцию в базе данных, и эта транзакция охватывает множество таблиц базы данных, чтобы обработать (Process) заказ и зарезервировать (Reserve) товары со склада.

Если любой из этих шагов совершить не удастся, то транзакция может откатиться, что означает отказ как от самого заказа, так и от зарезервированных товаров. Этот набор принципов называется ***ACID*** (атомарность, согласованность, изоляция, долговечность) и гарантируется на уровне системы базы данных.

На рисунке 4 представлена декомпозиция системы Интернет-магазина, построенной из микросервисов.

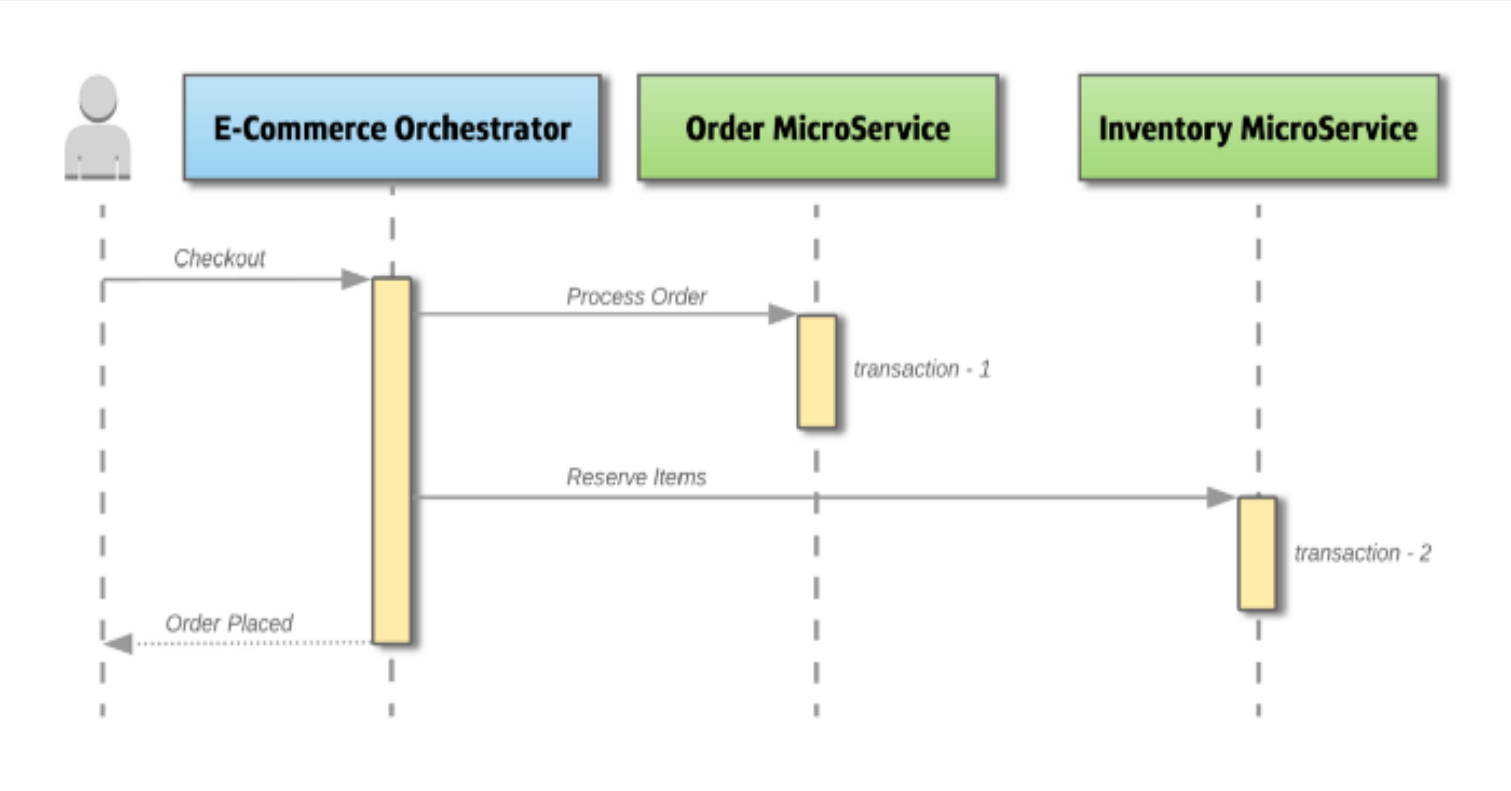

Рисунок 4 – Транзакции в микросервисе

Выполнив декомпозицию, мы создали микросервисы Order Microservice и Inventory Microservice, обладающие отдельными базами данных.

Когда от пользователя приходит запрос на заказ (Checkout), вызываются оба этих микросервиса, и каждый из них вносит изменения в свою базу данных. Поскольку теперь транзакция распространяется на несколько баз данных во множестве систем, она считается распределенной.

С внедрением микросервисной архитектуры базы данных утрачивают свою ACID-природу. В силу возможного распространения транзакций между множеством микросервисов и, следовательно, баз данных, приходится иметь дело с ***двумя ключевыми проблемами***:

***Поддерживание атомарности транзакции*.** Атомарность означает, что в любой транзакции могут быть завершены либо все шаги, либо ни одного. Если в вышеприведенном примере не удастся завершить операцию «заказать товары» в методе Inventory Microservice, то как откатить изменения в «обработке заказа», которые были применены Order Microservice?

***Обработка конкурентных запросов*.** Допустим, объект от любого из микросервисов поступает на долговременное хранение в базу данных, и в то же время другой запрос считывает этот же объект. Какие данные должен вернуть сервис – старые или новые?

В вышеприведенном примере, когда Order Microservice уже завершил работу, а Inventory Microservice как раз выполняет обновление, нужно ли включать в число запросов на заказы, выставленных пользователем, также и текущий заказ?

Современные системы проектируются с учетом возможных отказов, и одна из основных проблем при обработке распределенных транзакций хорошо сформулирована Патом Хелландом.

Как правило, разработчики просто не делают больших масштабируемых приложений, которые бы предполагали работу с распределенными транзакциями.

Возможные решения*.** Две вышеупомянутые проблемы весьма критичны в контексте проектирования и создания приложений на основе микросервисов. Для их решения применяется ***два следующих подхода:

- двухфазная фиксация;
- согласованность в конечном счете и компенсация / SAGA.

Двухфазная фиксация. Как понятно из названия, такой способ обработки транзакций предполагает ***два этапа***:

- фазу подготовки;
- фазу фиксации.

Важную роль в данном случае играет координатор транзакций, организующий жизненный цикл транзакции.

На подготовительном этапе все микросервисы, участвующие в работе, готовятся к фиксации и уведомляют координатора, что готовы завершить транзакцию. Затем на следующем этапе либо происходит фиксация, либо координатор транзакции выдает всем микросервисам команду выполнить откат.

Вновь рассмотрим для примера систему Интернет-магазина (рисунок 5).

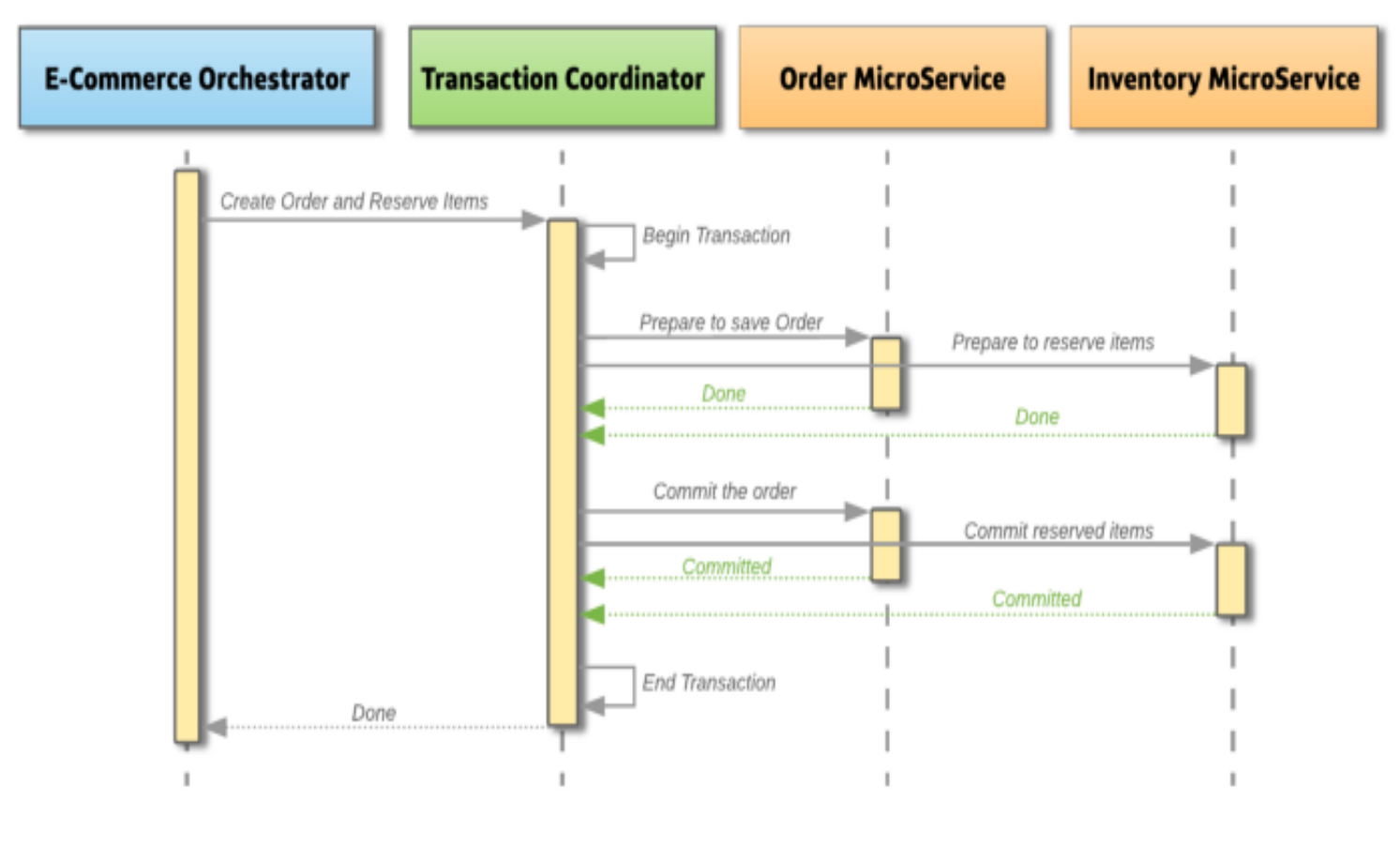

Рисунок 5 – Успешная двухфазная фиксация в микросервисной системе

В данном примере, когда пользователь направляет запрос на заказ, координатор Transaction Coordinator первым делом начинает глобальную транзакцию, обладая полной информацией о контексте.

Сначала он отправляет команду Prepare микросервису Order Microservice, чтобы создать заказ.

Затем отправляет команду Prepare к Inventory Microservice, чтобы зарезервировать товары.

Когда оба сервиса готовы внести изменения, они блокируют объекты от дальнейших изменений и уведомляют об этом Transaction Coordinator.

Как только Transaction Coordinator подтвердит, что все микросервисы готовы применить свои изменения, он прикажет этим микросервисам сохранить их, запросив фиксацию транзакции.

В этот момент все объекты будут разблокированы.

В сценарии отказа (рисунок 6) – если в любой момент отдельно взятый микросервис не успеет приготовиться, Transaction Coordinator отменит транзакцию и начнет процесс отката. На схеме Order Microservice по какой-то причине не смог создать заказ, но Inventory Microservice откликнулся, что готов создать заказ. Координатор Transaction Coordinator запросит отмену на Inventory Microservice, после чего сервис откатит все сделанные изменения и разблокирует объекты базы данных.

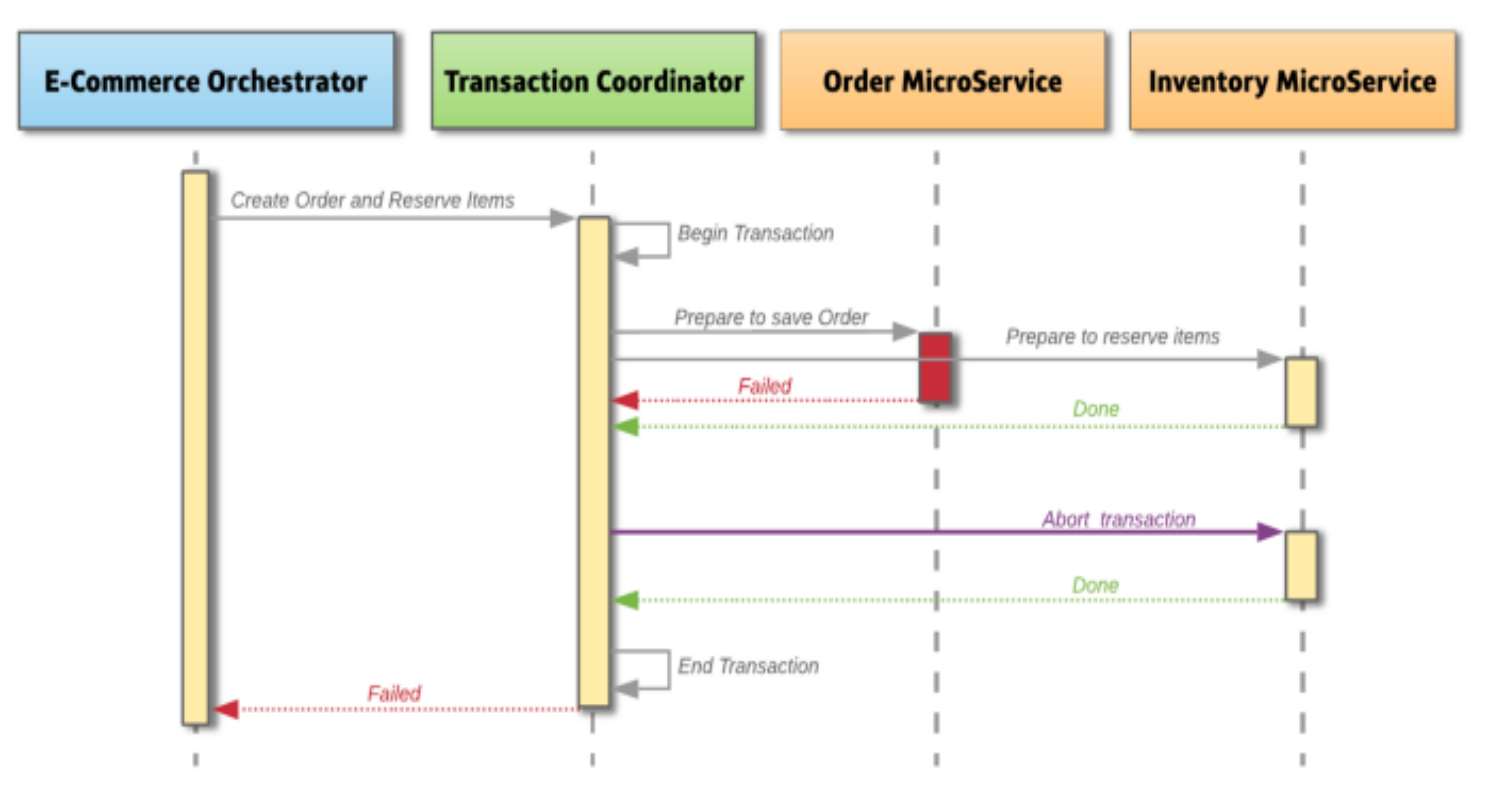

Рисунок 6 – Неудавшаяся двухфазная фиксация при работе с микросервисами

Такой подход гарантирует атомарность транзакции. Транзакция завершится либо в том случае, когда оба микросервиса сработают успешно, либо в случае, когда микросервисы не внесут никаких изменений.

Кроме того, данный подход позволяет изолировать чтение от записи, так как изменения в объектах не видны до тех пор, пока координатор транзакций не зафиксирует эти изменения.

Этот подход представляет собой синхронный вызов, при котором клиент будет уведомлен об успехе или неудаче.

Однако, двухфазные фиксации протекают довольно медленно по сравнению с операциями над одним микросервисом. Они сильно зависят от координатора транзакций, что может значительно замедлять работу системы в период высокой загруженности.

Другой серьезный недостаток заключается в блокировке строк базы данных. Блокировка может стать узким местом, затрудняющим производительность, причем, может возникнуть взаимная блокировка, где две транзакции намертво стопорят друг друга.

Согласованность в конечном счете и компенсация / SAGA. Одно из лучших ***определений согласованности*** в конечном счете дается на сайте microservices.io: ***каждый сервис публикует событие всякий раз, когда обновляет свои данные. Другие сервисы подписываются на события. При получении события сервис обновляет свои данные.***

При таком подходе распределенная транзакция выполняется как совокупность асинхронных локальных транзакций на соответствующих микросервисах. Микросервисы обмениваются информацией через шину событий.

Рассмотрим в качестве примера систему, работающую в Интернет-магазине (рисунок 7).

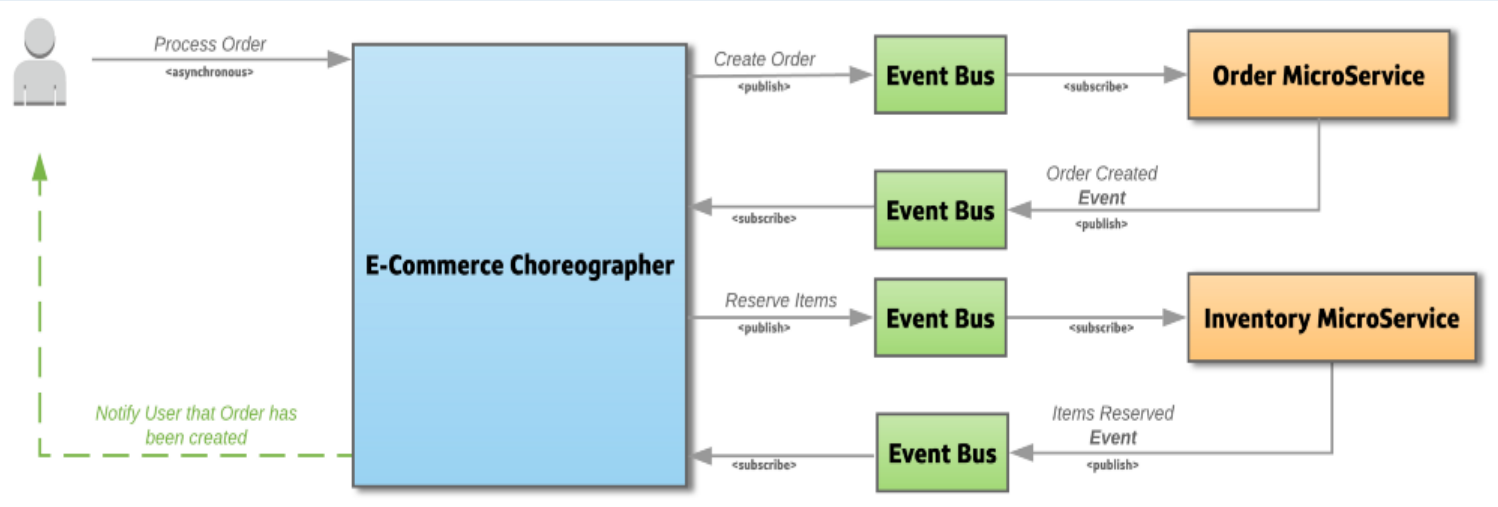

Рисунок 7 – Согласованность в конечном счете / SAGA, успешный исход

В данном примере клиент требует, чтобы система обработала заказ. При этом запросе Choreographer порождает событие Create Order (создать заказ), чем начинает транзакцию.

Микросервис Order Microservice слушает это событие и создает заказ – если эта операция прошла успешно, то он порождает событие Order Created (Заказ создан).

Координатор Choreographer слушает это событие и переходит к заказу товаров, порождая событие Reserve Items (зарезервировать товары).

Микросервис Inventory Microservice слушает это событие и заказывает товары; если это событие прошло успешно, то он порождает событие Items Reserved (товары зарезервированы). В данном примере это означает, что транзакция закончена.

Вся коммуникация на основе событий между микросервисами происходит через шину событий, а за ее организацию (хореографию) отвечает другая система – так решается проблема с излишней сложностью.

Если по какой-то причине Inventory Microservice не удалось зарезервировать товары (рисунок 8), он порождает событие Failed to Reserve Items (не удалось зарезервировать товары).

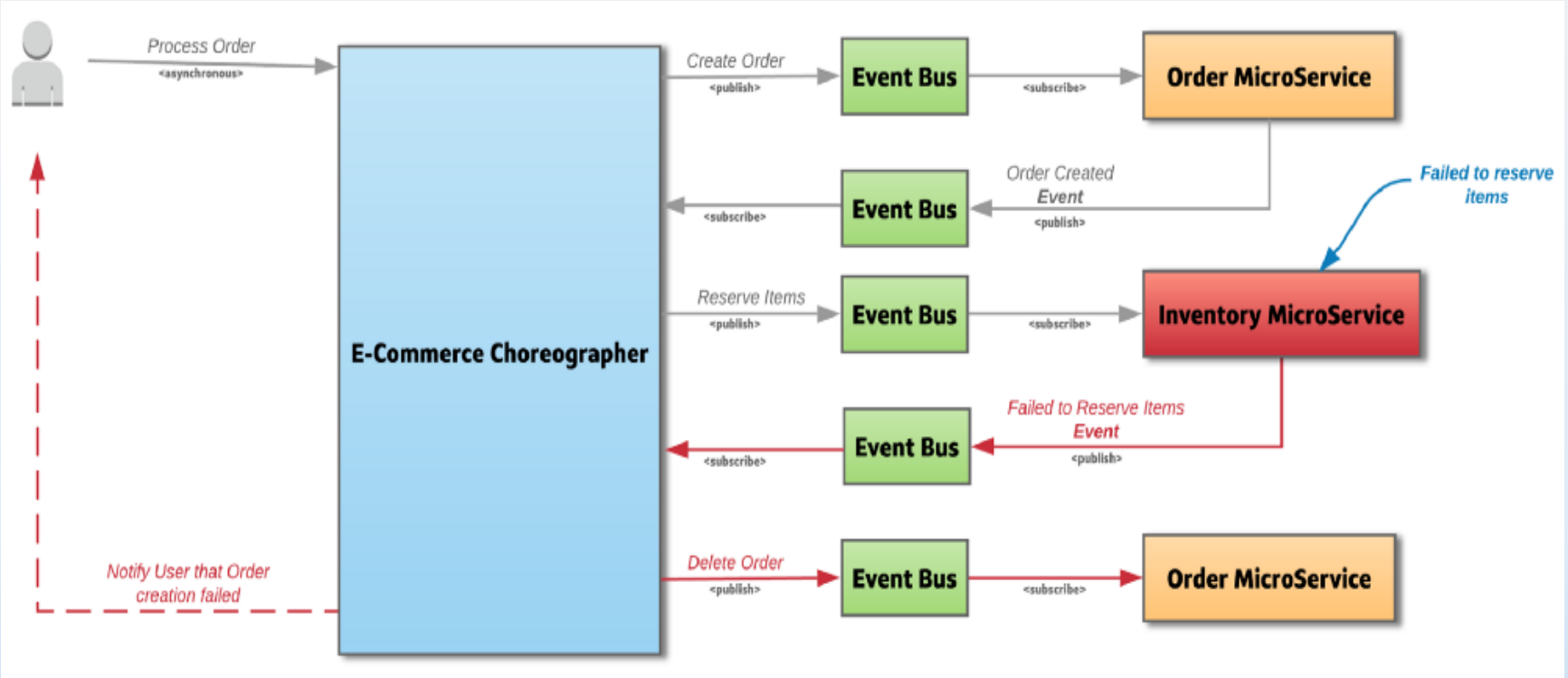

Рисунок 8 – Согласованность в конечном счете / SAGA, неудачный исход

Координатор Choreographer слушает это событие и запускает компенсирующую транзакцию, порождая событие Delete Order (удалить заказ).

Микросервис Order Microservice слушает это событие и удаляет ранее созданный заказ.

3. Безопасность в распределенных системах

Для обеспечения безопасности распределенной системы промежуточная среда должна обеспечивать поддержку ***трех функций, необходимых для создания безопасных систем***:

1. Проверка подлинности пользователя сервисов компоненты распределенной системы (аутентификация). Проверка подлинности может быть односторонней, когда только сервер убеждается в подлинности клиента, или двухсторонней, когда клиент так же убеждается в подлинности сервера.

2. Ограничение доступа к сервисам компоненты в зависимости от результатов аутентификации (авторизация). Для решения данной задачи промежуточная среда должна поддерживать ограничение доступа, основанное на так называемых ***ролях*** (***role based security***). Поскольку разработчики компоненты не могут обозначить уровни доступа через конкретных пользователей или групп пользователей системы, то они должны использовать некоторые абстрактные роли, которые при развертывании компоненты будут связаны администратором системы с учетными записями пользователей системы.

3. Защита данных, передаваемых между компонентами системы, от просмотра и изменения третьими сторонами. Для этого передаваемые между компонентами сообщения должны подписываться электронной подписью и шифроваться как клиентом, так и сервером.

Функции обеспечения безопасности могут обеспечиваться транспортным протоколом, используемым промежуточной средой, самой средой, или ими обеими в совокупности.

Механизмы защиты ресурсов в распределенных системах. Механизм защиты от несанкционированного доступа является частью механизмов распределенной системы и обычно входит в ее ядро безопасности.

Системы должны быть не только защищенными и безопасными, но и быть в состоянии продемонстрировать эти свойства пользователям. Это может быть сделано с помощью различных моделей безопасности, так как общепринятая модель безопасности распределенных систем отсутствует. Данное обстоятельство затрудняет обсуждение проблем в общем виде.

Существует множество различных подходов к построению механизмов защиты. Все они основаны на определенных моделях управления доступом. При этом используется абстрактная структура операционной системы или концепция безопасности, принятая для вооруженных сил США.

Модели управления доступом могут быть реализованы как ***модели систем дискреционного доступа*** (доступ к объекту или ресурсу предоставляется или не предоставляется любому пользователю по решению «владельца» объекта) или как ***модели систем недискреционного (мандаторного) доступа*** (доступ к объекту или ресурсам предоставляется или не предоставляется на основе категории секретности информации и допуска, присвоенного пользователю).

Известны и применяются следующие модели/механизмы управления доступом: матричная модель, решетчатая модель (модель Деннинга), модель Белла-ЛаПадула, модель информационного потока, механизм ядра безопасности.

***Матричная модель доступа*.** Модель основана на абстрактном представлении структуры системы. Она отображает политику управления доступом, состоящую в том, что права доступа каждого пользователя к каждому объекту определяются как входы матрицы. В распределенной системе объектами могут быть главные и другие ЭВМ, терминалы, доступ к которым контролируется, а также объекты внутри ЭВМ, например процессы, файлы, принтеры, доступ к которым традиционно контролируется централизованно системными средствами. Внутренние объекты ЭВМ могут быть включены в матрицу как объекты, обслуживающие главные ЭВМ. Матричная модель может быть реализована как модель системы дискреционного управления доступом.

Важная особенность матричной модели доступа состоит в том, что управление доступом к информации (файлам) осуществляется на основе категории секретности информации и категории допуска пользователей. При этом доступ к информации разрешается или не разрешается без учета семантики данной информации.

Благодаря общности, простоте и возможности реализации различными способами матричная модель управления доступом нашла широкое применение в распределенных системах.

Основной недостаток этой модели состоит в ее уязвимости к воздействию программ типа «троянский конь» и компьютерных вирусов. Другим ее недостатком является то, что она не учитывает семантику информации. Это серьезный недостаток, так как во многих реальных системах, например военных, банковских, коммерческих и медицинских, обрабатывается секретная и конфиденциальная информация.

***Модель Деннинга (Denning) или решетчатая модель*.** Эта модель является расширением матричной модели, где степень защиты зависит от определения потока защищенной информации.

В этой модели решетка представляет собой конечное множество частично упорядоченных элементов – такое, что для каждой пары элементов имеются верхняя граница и наибольшая нижняя граница.

С точки зрения безопасности это означает, что имеется множество частично упорядоченных классов доступа, из которых выбирается пара, соответствующая категории допуска пользователя и категории секретности информации.

Таким образом, данная модель учитывает категорию секретности, категорию допуска пользователя и правила определения категории секретности. Она построена с учетом присвоения категории секретности, принятой в министерстве обороны США.

Решетчатую модель можно использовать и в коммерческих системах. При этом объектам присваивается определенная категория секретности, а субъектам (пользователям) – категории допуска.

Данная модель может быть реализована как модель системы недискреционного управления доступом.

Модель Белла-ЛаПадула*.** Ее можно рассматривать как пример решетчатой модели. Ее авторы представили ядро безопасности системы как машину с конечным числом состояний, переход которой из одного состояния безопасности в другое определяется правилами безопасности. Эта модель является наиболее известной моделью системы управления доступом. В ней кроме объектов и субъектов матричной модели доступа учитываются и категории секретности информации, принятые для защищенных военных систем. Следовательно, каждый объект системы имеет определенную категорию секретности, а каждый субъект – допуск с указанием разрешенной секретности. С моделью связаны ***две основные аксиомы:

- пользователь не может считывать информацию, категория секретности которой выше разрешенного для него уровня секретности;

- пользователь не имеет права понизить категорию секретности информации.

Модель потока информации, предложенная Деннингом, также основана на концепции решетчатой модели. Однако в ней вместо перечня аксиом, определяющих права пользователя на доступ, требуется, чтобы любая передача информации определялась порядком обмена потоками информации между классами безопасности. Таким образом, основное внимание здесь обращено на поток информации от одного объекта к другому, а не на рассмотрение индивидуальных запросов на доступ к объекту. Преимущества такого доступа состоят в большей гибкости анализа потоков в каналах с памятью и большей детализации этого анализа.

***Механизм ядра безопасности*.** Под ядром безопасности понимается небольшое подмножество элементов системы, на которое возлагается ответственность за безопасность всей системы. Ставится условие, что это подмножество будет контролировать все возможные пути доступа в систему, действовать правильно и независимо, так что его действия не могут быть фальсифицированы.

Рашби (Rushby) и Рэндел (Randell) всесторонне исследовали возможности защиты распределенных систем с использованием концепции ядра безопасности. Они приняли, что распределенная система состоит из небольших ЭВМ, безопасность которых доказуема, и некоторого количества больших защищенных ЭВМ. ***Безопасность в такой системе достигается***:

- частично путем физического разделения отдельных ее элементов;
- частично за счет управления доступом к незащищенным элементам и связью между ними посредством ядра безопасности.

Концепция механизма ядра безопасности рассматривается как общая модель, однако ее реализация более тесно связана с ЭВМ. Это объясняется тем, что данный механизм построен на концепции монитора обращений в систему, позволяющей реализовать специфическую совокупность стратегии безопасности.

***Монитор обращений в систему*.** Концепция монитора обращений в систему объединяет субъекты, объекты, базу данных монитора (называемую также санкционирующей, или разрешающей базой данных) и механизм контроля. К числу субъектов относятся пользователи, процессы или потоки заданий, запрашивающие доступ к объектам. Объектами являются файлы, программы, терминалы и магнитные ленты с записанной на них информацией. В мониторе обращения субъекты получают доступ к объектам на основании разрешений (содержащихся в базе данных монитора). Каждое обращение к информации или изменение разрешений должно проходить через монитор обращений

Общая концепция монитора обращений хорошо согласуется с представлениями о ядре безопасности системы и многоуровневых абстрактных машинах. При этом ядро безопасности реализует абстракцию монитора обращений и является последним иерархическим уровнем системы, ответственным за все относящиеся к безопасности операции.

4. Промежуточные среды в Microsoft .NET Framework

Понятие промежуточной среды, обеспечивающей сервисы высокого уровня для инкапсуляции удаленного взаимодействия, появилось в середине 90-х годов, когда выяснилось, что для создания распределенных систем необходима некоторая независящая от приложения и операционной среды «прослойка».

Среда CLR так же может рассматриваться как некоторая «промежуточная» среда для выполнения программ на управляемом коде. Поэтому закономерно использовать .NET Framework в качестве основы для создания распределенных приложений.

В настоящий момент в .NET Framework Class Library присутствует ***поддержка четырех промежуточных сред для построения распределенных систем***:

- Среда Microsoft Message Queuing (MSMQ) поддерживает обмен сообщениями между программными компонентами на основе очередей.

- Среда Microsoft Enterprise Services основана на разработанной ранее фирмой Microsoft среде COM+, которая позволяет использовать удаленные объекты и распределенные транзакции в локальной сети.

- Среда ASP .NET Web Services позволяет организовать удаленный вызов на основе общепринятых стандартов, базирующихся на языке XML.

- Среда .NET Remoting была разработана как универсальная промежуточная среда для использования удаленных объектов.

В версии .NET Framework 3.0 предполагается ввести технологию ***WCF (Windows Communication Foundation)***, объединяющую все упомянутые технологии построения распределенных систем. Кроме указанных технологий, приложения на .NET Framework могут использовать, например, удаленные вызовы на основе стандарта XML-RPC при подключении дополнительных библиотек.

В целом, под платформой Microsoft.NET Framework следует понимать интегрированную систему (инфраструктуру) средств разработки, развертывания и выполнения сложных (как правило, распределенных) программных систем.

Платформа .NET состоит из нескольких основных компонентов (рисунок 9).

Рисунок 9 – Платформа Microsoft.NET

Операционные системы корпорации Microsoft (Windows 2000/XP/ME/CE) представляют собой базовый уровень платформы MS.Net.

Серверы MS.Net (.Net Enterprise Servers) являются программными продуктами корпорации Microsoft, использование которых позволяет снизить сложность разработки сложных программных систем. В числе готовых для применения серверы Application Center 2000, Exchange Server 2000, SQL Server и др.

Сервисы MS.Net (.Net Building Block Services) представляют собой готовые «строительные блоки» сложных программных систем, которые могут быть использованы через Интернет как сервисные услуги. Набор таких сервисов MS.Net планируется последовательно расширять.

Примером имеющегося сервиса платформы MS.Net является Microsoft Passport, позволяющий установить единое имя пользователя и пароль на всех сайтах, поддерживающих аутенфикацию через Passport.

Интегрированная среда разработки приложений Visual Studio.NET (VS.Net) – верхний уровень MS.Net – обеспечивает возможность создания сложного программного обеспечения на основе платформы и продолжает в этом плане ряд разрабатываемых корпорацией Microsoft средств разработки профессионального программного обеспечения.

Подсистема MS.NET Framework является ядром платформы MS.Net, обеспечивая возможность построения и исполнения .Net приложений (рисунок 10).

Рисунок 10 – Архитектура MS.NET Framework

В состав MS.NET Framework входит:

- общеязыковая среда выполнения (Common Language Runtime или CLR);
- библиотеки классов подсистемы MS.NET Framework.

Базовый уровень подсистемы MS.NET Framework составляет общеязыковая среда выполнения (Common Language Runtime или CLR).

Общеязыковая среда выполнения (Common Language Runtime, CLR) выполняет следующие функции:

- активизирует исполняемый код;
- выполняет для него проверку безопасности;
- располагает этот код в памяти и исполняет его;
- обеспечивает сборку мусора.

По своему функциональному назначению в составе библиотек классов могут быть выделены:

- набор базовых классов, обеспечивающих, например, работу со строками, ввод-вывод данных, многопоточность и т.п.;

- набор классов для работы с данными, предоставляющих возможность использования SQL-запросов, ADO.Net и обработки XML данных;

- набор классов Windows Forms, позволяющих создавать обычные Windows-приложения, в которых используются стандартные элементы управления Windows;

- набор классов Web Forms, обеспечивающих возможность быстрой разработки Web-приложений, в которых используется стандартный графический интерфейс пользователя;

- набор классов Web Services, поддерживающих создание распределенных компонентов-сервисов, доступ к которым может быть организован через Интернет.

Здесь набор ***базовых классов*** (класс – это реализация некоторого множества интерфейсов) обеспечивает, например, работу со строками, ввод-вывод данных, многопоточность.

Набор классов для работы с данными предоставляют возможность:

- использования SQL-запросов, ADO.Net;

- обработки XML данных и т.д.

Для обеспечения возможности многоязыковой разработки программного обеспечения, программный код, получаемый после компиляции программы платформы MS.Net, представляется на общем промежуточном языке (Common Intermediate Language или CIL).

Сборки (файлы на CIL) перед своим исполнением с помощью JIT-компилятора (Just-In-Time compilers) переводятся с программного кода на промежуточном языке (CIL-кода) в машинный (native) код платформы исполнения.

Выводы

В ходе лекции рассмотрены следующие вопросы:

- *распределенные события;*
- *распределенные транзакции;*
- *безопасность в распределенных системах;*
- *промежуточные среды в Microsoft.NET Framework.*

Задание на самостоятельную работу

1. Конспект лекций.

Вид и тема следующего занятия

Практическое занятие №3. Основы в ASP.NET Core (ч. 3)