12 Интеграция посредством обмена сообщениями

О ЧЕМ РАССКАЗЫВАЕТСЯ В ЭТОЙ ГЛАВЕ?

- Интеграция ограниченных контекстов посредством обмена асинхронными сообщениями с использованием NServiceBus и Mass Transit
- Проектирование и моделирование систем сообщений на основе важных событий в предметной области
- Знакомство с особенностями работы фреймворков обмена сообщениями
- Создание архитектурных диаграмм
- Концептуальные различия между командами и событиями
- Теория и примеры применения стандартных шаблонов организации обмена сообщениями
- Поддержание потенциальной непротиворечивости
- Мониторинг ошибок в системах обмена сообщениями
- Мониторинг соблюдения соглашений об уровне обслуживания (Service Level Agreement, SLA) в системах обмена сообщениями

Загружаемые примеры кода для этой главы

Примеры программного кода для этой главы можно найти по адресу *www.wrox.com/go/domaindrivendesign* на вкладке Download Code (Загружаемый код). Примеры кода для главы 12 (и для других глав) доступны для загрузки отдельно, в виде файла с именем, соответствующим номеру главы.

Надеемся, что описание идей в предыдущей главе разожгло в вас интерес к возможностям разработки масштабируемых распределенных систем с сохранением всех выгод предметно-ориентированного проектирования. Цель этой главы — дать вам основополагающие навыки, с помощью которых вы сразу же сможете применять эти идеи на практике. Чтобы получить начальные умения, вы создадите приложение электронной коммерции, использующее механизм обмена со-

общениями на основе фреймворков NServiceBus и Mass Transit. Одновременно вы посмотрите, как внедрять в свои модели знания, полученные от специалистов в предметной области, используя приемы, которые делают предметные понятия в коде более явными. Вы также увидите, что одним из лучших таких приемов является именование сообщений в соответствии с событиями, происходящими в предметной области.

Все примеры в этой главе будут демонстрировать решение типичных задач, часто встречающихся на практике. С некоторыми из них вы уже познакомились в главе 11 «Введение в интеграцию ограниченных контекстов», например с увеличением надежности синхронных HTTP-вызовов служб, не контролируемых вами. Создание систем — лишь малая часть общей картины, поскольку эти системы необходимо также поддерживать и сопровождать. Поэтому в данной главе вы также узнаете, как решать проблемы с изменением форматов сообщений, касающиеся сразу нескольких групп, как контролировать производительность и ошибки в вашей системе обмена сообщениями и как реализовать горизонтальное масштабирование на множестве машин по мере увеличения потребностей предприятия.

Одно из самых больших преимуществ слабосвязанных систем заключается в возможности высокой скорости разработки несколькими группами программистов, как рассказывалось в предыдущей главе. В этой главе вы увидите, как на уровне реализации организовать файлы с исходными текстами и проекты в целях поддержания такой возможности. Но прежде чем опуститься на этот уровень, часто полезно создать общий проект системы, чтобы все группы разработчиков знали, какое место в системе занимают их ограниченные контексты. Поэтому часть данной главы также будет посвящена созданию архитектурных диаграмм, позволяющих наглядно представить важные решения и предметно-ориентированные бизнес-сценарии использования.

Системы обмена сообщениями имеют свои недостатки, и в этой главе будет показано, как обойти некоторые из них. Один из недостатков, в сравнении с вызовами HTTP, заключается в отсутствии стандартных форматов. Это может стать проблемой, если понадобится интегрировать два решения, использующие разные фреймворки обмена сообщениями. Данная проблема частично может быть решена с использованием шаблона обмена сообщениями, получившего название «Мост обмена сообщениями» (messaging bridge). В этой главе вы построите такой мост, соединяющий системы обмена сообщениями NServiceBus и Mass Transit.

Но прежде чем начинать писать код, вам следует познакомиться с некоторыми особенностями систем обмена сообщениями. Они помогут вам лучше понять, на что именно направлены примеры, обсуждаемые далее в этой главе.

Основы обмена сообщениями

Приложения, опирающиеся на обмен сообщениями, в корне отличаются от традиционных нераспределенных объектно-ориентированных приложений. Приложения этого вида не только обладают новыми преимуществами, такими как устойчивость к отказам и масштабируемость, но и содержат проблемные места, такие как

модель асинхронного программирования, требующая совершенно иного мышления. Однако теория, представленная в предыдущей главе, и описание основ обмена сообщениями, о которых рассказывается в этом разделе, должны дать вам все необходимые знания для создания реактивных приложений, использующих асинхронные сообщения. Первое понятие, которое вам нужно освоить, — шина сообщений (message bus). Это связующее звено, обеспечивающее целостность системы.

Шина сообщений

Если в системе имеется единственный централизованный компонент, отвечающий за отправку всех сообщений, вся система может обрушиться, если этот компонент прекратит работу. Не меньшую проблему представляет случай, когда компонент слишком сложен для внедрения в каждую отдельную часть системы, чтобы можно было обеспечить ее масштабируемость в соответствии с потребностями предприятия. Решить эти проблемы можно с помощью *шины сообщений* (message bus), представляющей собой распределенную систему, имеющую агентов в каждом компоненте, который отправляет или принимает сообщения, и позволяющую избежать появления единственной центральной критической точки. Шина сообщений обеспечивает коллективную работу всех частей, как показано на рис. 12.1.

Использование шины сообщений позволяет устранить зависимости между ограниченными контекстами. Каждый ограниченный контекст — точнее, каждый компонент — зависит только от шины. Если компонент выходит из строя, это никак не отражается на других компонентах.

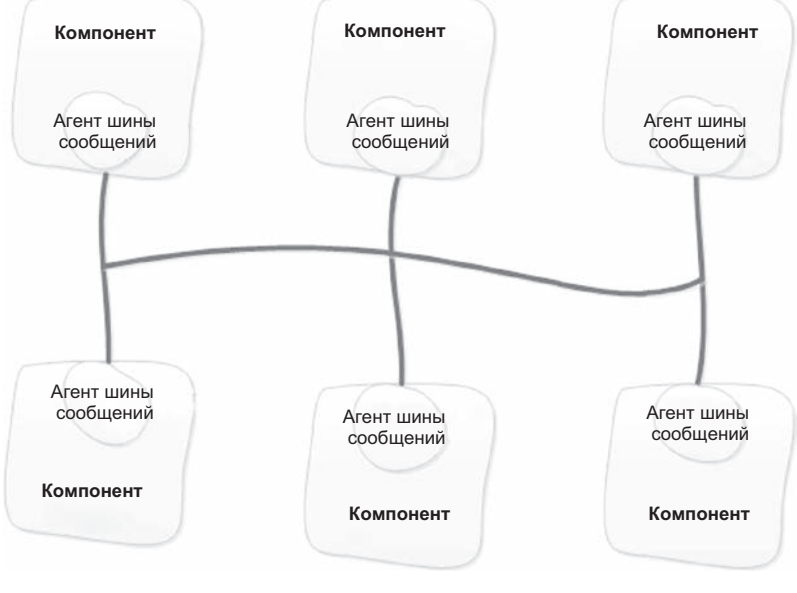

Рис. 12.1. Архитектура шины сообщений

ПРИМЕЧАНИЕ

Централизованный компонент, принимающий и рассылающий все сообщения, называют брокером сообщений (message broker), или просто брокером. Из-за проблем с надежностью и масштабируемостью предпочтение часто отдается шине сообщений. Однако современные брокеры поддерживают горизонтальное масштабирование и многие другие свойства шин сообщений. В качестве примера можно привести пользующийся заслуженной популярностью брокер Apache Kafka (*http://kafka.apache.org/*), созданный и используемый в LinkedIn (*https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying*). Большинство идей, описываемых в этой главе, также применимо к системам, использующим современные брокеры, такие как Kafka, хотя иногда может потребоваться приложить чуть больше усилий для решения дополнительных проблем.

ПРИМЕЧАНИЕ

Как упоминалось в предыдущей главе, под компонентами в этой главе понимаются отдельные приложения, которые можно распространять и развертывать независимо друг от друга. Вы также можете встретить такие названия, как «автономные компоненты» (autonomous components), «распределенные компоненты» (distributed components), «компоненты, обменивающиеся сообщениями» (messaging components) и даже «микрослужбы» (micro services). К сожалению, сообщество разработчиков пока не нашло точного, однозначного и непротиворечивого названия.

Надежный обмен сообщениями

Отправка сообщений из одного ограниченного контекста в другой может стоить очень дорого в отсутствие гарантий доставки сообщений — клиент будет крайне недоволен, если вы сняли деньги с его кредитной карты, но не смогли организовать доставку купленного им товара из-за потери сообщения где-то в недрах системы. Это одна из причин, объясняющих потребность в надежной доставке сообщений. К сожалению, почти невозможно гарантировать, что сообщение всегда будет доставляться точно один раз. Если сообщение было отправлено, но подтверждение его получения не было принято, оно будет послано повторно. Но что же произойдет, если в действительности сообщение было принято, а подтверждение потерялось? Естественно, то же самое сообщение будет отправлено повторно, исходя из ошибочного предположения, что оно не было принято в первый раз.

Из-за проблем с надежностью доставки было разработано множество шаблонов надежного обмена сообщениями, каждый из которых имеет свои проблемы и компромиссы, в их числе: «доставка не менее одного раза» (at-least-once), «доставка не более одного раза» (at-most-once) и «доставка точно один раз» (only-once).

По мере знакомства с системами обмена сообщениями вы узнаете, что предпочтение обычно отдается шаблону «доставка не менее одного раза» (at-least-once), даже несмотря на риск обработать одно и то же сообщение дважды. Чтобы вы сумели избежать проблем, которые может вызвать такое поведение, например двукратное списание денег со счета клиента за один заказ, в этой главе будет описано, как объединить шаблон «доставка не менее одного раза» (at-least-once) с понятием идемпотентности сообщений.

ПРИМЕЧАНИЕ

Идемпотентными называют сообщения, которые могут посылаться многократно, но обрабатываться только один раз. Например, если сообщение, вызывающее списание денег со счета, посылается дважды, получатель обработает такое сообщение только один раз. Это часто достигается за счет присваивания каждому сообщению некоторого уникального идентификатора, например реквизитов платежа.

Доставка не менее одного раза предполагает повторную отправку сообщений в случае неудачи или отсутствия подтверждения со стороны получателя (из чего можно предположить, что он вышел из строя). Выполнение фактических операций, связанных с повторной отправкой сообщений, возлагается на шаблон «сохранить и передать» (store-and-forward).

Сохранить и передать

Отправленное сообщение может не достигнуть получателя. Например, в случае сетевых неполадок, выхода из строя аппаратного обеспечения или появления программной ошибки. Шаблон «сохранить и передать» (store-and-forward) решает многие подобные проблемы, сохраняя сообщение перед его отправкой. Если сообщение достигло получателя и было подтверждено, его локальная копия удаляется. Но если сообщение не достигло получателя, оно посылается повторно. Далее в этой главе вы увидите, как фреймворки обмена сообщениями принимают на себя большую часть сложностей, позволяя устанавливать правила, которые определяют частоту повторной отправки сообщений и продолжительность ожидания между повторами.

Для хранения сообщений большинство фреймворков использует очереди. Это означает, что, когда Служба А посылает сообщение Службе Б, сообщение будет помещено в очередь Службы Б. Когда Служба Б приступит к обработке сообщения, она извлечет его из очереди. Однако если Служба Б не завершит обработку сообщения, она вновь поместит его в очередь, чтобы повторить попытку позднее.

Команды и события

Иногда сообщения, такие как `PlaceOrder` (заказ размещен) из главы 11, отправляются с целью сообщить о том, что должно произойти. Такого рода сообщения часто называют *командами*. Команды образуют логическую связь между отправителем и получателем, потому что отправитель знает, как получатель должен обработать сообщение. Команды обрабатываются единственным получателем, что лишает систему гибкости. События, такие как `OrderCreated` (заказ создан), напротив, сообщают о происшедшем. События не образуют тесных связей как команды, потому что отправитель не знает, как получатель будет обрабатывать сообщение. Фактически отправитель даже не знает, кто обрабатывает сообщения. Это обусловлено особенностями шаблона «издатель/подписчик» (publish/subscribe).

Более слабая зависимость, обеспечиваемая шаблоном «издатель/подписчик», часто делает события более предпочтительными, чем команды. Одно из существенных преимуществ событий состоит в возможности добавлять новых подпис-

чиков без изменения существующего программного кода. Это позволяет добавлять совершенно новые ограниченные контексты, не меняя существующий код и не замедляя работу других групп разработчиков. В примере сайта электронной коммерции из предыдущей главы можно было бы добавить в поток обработки событий новый ограниченный контекст `Marketing`, представляющий отдел продвижения, просто подписав его на событие `OrderCreated`, как показано на рис. 12.2.

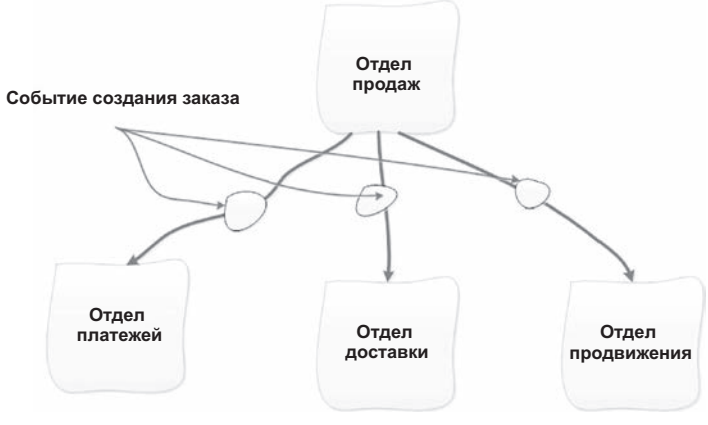

Рис. 12.2. Добавление нового подписчика на получение события не влечет изменения существующего кода

В Сети можно наткнуться на споры о способах именования команд и событий. Вы можете выбрать свой вариант, но по мнению большинства, команды должны получать имена, имеющие вид приказов, определяющие, что должно произойти — `PlaceOrder` (разместить заказ), `ChangeAddress` (изменить адрес), `RefundAccount` (вернуть деньги на счет), — а события должны получать имена, описывающие происшедшее в прошлом времени, — `OrderCreated` (заказ создан), `MovedAddress` (адрес изменен), `AccountRefunded` (деньги возвращены на счет).

Потенциальная непротиворечивость

Реализуя обмен сообщениями в системе, где каждый ограниченный контекст имеет собственную базу данных, есть риск оказаться в ситуации, нетипичной для систем, использующих большие транзакции. Примером может служить выполнение заказа без его оплаты. В системах, использующих транзакции, такое состояние вообще невозможно, потому что создание заказа является составной частью большой, атомарной операции, в рамках которой также осуществляется обработка оплаты. Если в ходе попытки оплатить заказ произойдет ошибка, заказ не будет создан. В решениях на основе сообщений, напротив, такое состояние вполне возможно, потому что они устраняют большие транзакции и являются потенциально непротиворечивыми.

Одним из важнейших аспектов потенциально непротиворечивых систем является создание благоприятного впечатления у пользователей. В непротиворечивых си-

стемах пользователи вынуждены ждать, пока завершится вся транзакция. После чего они точно будут знать, что заказ создан, платеж принят и доставка организована. Однако, как вы узнали в главе 11, большие транзакции не всегда хорошо масштабируются. В системах с потенциальной непротиворечивостью пользователи часто немедленно получают подтверждение, что намерение разместить заказ было принято, а позднее, после обработки платежа и организации доставки, им высылается подтверждение по электронной почте, что на первый взгляд может отрицательно сказаться на впечатлении пользователей.

Вы можете обратить потенциальную непротиворечивость на пользу предприятию, спросив у заинтересованных лиц, что должно происходить в потенциально непротиворечивых состояниях. Иногда таким способом можно вскрыть новые деловые правила или возможности. Для получения дополнительной информации мы настоятельно рекомендуем прочитать статью Уди Дахана «Race Conditions Don't Exist» (*http://www.udidahan.com/2010/08/31/race-conditions-dont-exist/*).

ПРИМЕЧАНИЕ

Большинство шаблонов обмена сообщениями из рассматриваемых в этой главе было формализовано Грегором Хопом и Бобби Вульфом в их весьма авторитетной книге «Enterprise Integration Patterns»[1]. Описания всех этих и многих других шаблонов доступны бесплатно на веб-сайте книги (*http://www.eaipatterns.com/*).

Создание приложения электронной коммерции с применением NServiceBus

Теперь у вас есть возможность получить практический опыт создания реактивной системы обмена сообщениями. Сначала вы создадите систему электронной коммерции, с помощью которой пользователи смогут совершать покупки через Интернет. Вы будете использовать команды, события и другие хорошо известные шаблоны, чтобы в полной мере оценить расширенные возможности масштабируемости и надежности, предоставляемые системами обмена сообщениями. Когда вы будете готовы, вашей первой задачей будет загрузить фреймворк NServiceBus, чтобы обеспечить свой компьютер всем необходимым для работы с примерами.

ПРИМЕЧАНИЕ

Чтобы опробовать примеры из этой главы, нужно установить бесплатную версию Visual Studio Express 2013 for Web (*http://www.visualstudio.com/products/visual-studio-express-vs*). При желании можно также использовать одну из полных версий Visual Studio.

Установить фреймворк NServiceBus совсем не сложно: просто загрузите и запустите установочный файл. Для работы с примерами в этой главе выбирайте версию 4.3.3

[1] *Хоп Г., Вульф Б.* Шаблоны интеграции корпоративных приложений. — М.: ООО «И. Д. Вильямс», 2015. — *Примеч. пер.*

(*https://github.com/Particular/NServiceBus/releases/download/4.3.3/Particular.NServiceBus-4.3.3.exe*). Она содержит все необходимое, включая службу очередей сообщений Microsoft Message Queuing (MSMQ) и координатора распределенных транзакций Distributed Transactions Coordinator (DTC). Как вы увидите ниже, вам потребуется также настроить ссылки на сборки (assemblies) NServiceBus в своих проектах.

ПРИМЕЧАНИЕ

Работая с примерами из этой главы, помните, что вы можете задавать любые вопросы или делиться своими мыслями на дискуссионном форуме Wrox.

Проектирование системы

Прежде чем запустить среду разработки, часто полезно графически изобразить, что именно вы собираетесь создать, особенно когда некоторые понятия являются для вас новыми. Визуальное представление того, как разные компоненты укладываются в единую картину, поможет лучше понять, что именно вы в действительности создаете. Проектирование приложения электронной коммерции в этой главе будет выполнено в три этапа. Один шаг — создание диаграммы контейнеров, изображающей группы в приложении, выбранные технологии и протоколы взаимодействий. Другой шаг — создание диаграммы компонентов, отображающей логические связи между ограниченными контекстами. Но первый и самый важный шаг — знакомство с предметной областью.

Предметно-ориентированное проектирование

Как вы узнали в первой части книги, самое важное в разработке программного обеспечения — это, пожалуй, определиться с целью его создания, чтобы суметь спроектировать систему, представляющую ценность для тех, кто будет ее использовать. Поэтому, проектируя систему, большую часть времени следует тратить на создание единого языка и выявление скрытых идей в сотрудничестве со специалистами в предметной области. Некоторые практики DDD рекомендуют начинать с выявления важнейших событий, возникающих в предметной области. Этот этап известен под названием «штурм событий» (Event Storming) и как нельзя лучше подходит для приложений, опирающихся на обмен сообщениями (*http://ziobrando.blogspot.co.uk/2013/11/introducing-event-storming.html*).

События в предметной области

Создавая ограниченные контексты, которые интегрируются посредством общих команд и событий, вы получаете отличную возможность определить, как будут отображаться важные события предметной области в события, возникающие в программной системе. Этот шаблон используется многими известными практиками DDD для явного выражения предметных понятий в создаваемой системе сообщений. Для этого необходимо сначала выявить важные события в предметной области. Их часто называют предметными событиями.

В первой части книги «Принципы и приемы предметно-ориентированного проектирования» было показано множество способов приобретения знаний о предметной области, включая и события. Обычно это происходит во время сеансов переработки знаний совместно со специалистами в предметной области, на неформальных встречах и даже в обеденных перерывах. В предыдущей главе мы видели, что события в предметной области возникают в любом случае, даже в отсутствие программной системы. Вот некоторые из них: «заказ размещен», «оплата получена» и «доставка организована». Этот этап может принести пользу при создании системы событий, если особое внимание уделить подобным предметным событиям.

После того как предметные события выявлены, они становятся частью единого языка (UL) и можно приступать к их объединению, чтобы сформировать законченные сценарии использования. Лучший способ сохранить это знание и поделиться им с другими — отобразить последовательности событий на диаграмме компонентов.

Диаграммы компонентов

Нарисовав схему высокоуровневой логики до начала работы над программным кодом, вы получаете преимущество, позволяющее эффективнее создавать компоненты, потому что вы понимаете, что делаете. И в этом существенную помощь вам окажет диаграмма компонентов. Лучшее время для начала создания диаграмм компонентов — сеансы переработки знаний со специалистами в предметной области. Вы можете вместе набросать основные схемы из прямоугольников и стрелок, отображающие предметные события и процессы. Когда затем вы приступите к программированию, вы уже будете понимать, что именно собираетесь создать, и иметь единый язык с терминологией, необходимой для моделирования системы.

Диаграммы компонентов не имеют формальной структуры, они просто отражают логические связи, или взаимодействия между определенными компонентами. Не следует подниматься на слишком высокий уровень, отображая технологические варианты, точно так же не следует слишком углубляться в детали и показывать имена классов и методов. Пример хорошей диаграммы можно увидеть на рис. 12.3, она показывает потоки сообщений между ограниченными контекстами в будущем приложении электронной коммерции.

Число диаграмм компонентов ограничивается только необходимостью. В данном примере создана единственная диаграмма компонентов, сообщающая порядок размещения заказа. Вы можете взять на вооружение следующее соглашение: диаграмма компонентов должна создаваться для каждого высокоуровневого сценария использования с последующей оценкой ее значимости для вас.

Диаграммы контейнеров

В некоторый момент, после нескольких встреч со специалистами в предметной области, когда начнет появляться понимание предметной области, вы сможете приступить к созданию системы. Вы должны будете реализовать требования предприятия в виде работающей распределенной программной системы, пред-

ставляющей ценность для бизнеса. Сбалансировать функциональные предметные и нефункциональные технические требования вам поможет создание диаграмм контейнеров.

Как будут взаимодействовать разные части приложений? Как проверить, что система обеспечивает необходимый уровень отказоустойчивости и масштабируемости? Как гарантировать понимание всеми разработчиками в группе, что они делают, чтобы они не пошли неправильным путем? Эти вопросы очень важны для большинства программных проектов, а ответить на них вам поможет диаграмма контейнеров. Диаграммы этого вида показывают, как группируются разные части системы, как они взаимодействуют и какие технологии выбраны для их реализации.

Рис. 12.3. Диаграмма компонентов, отображающая предметные события

ПРИМЕЧАНИЕ

Ключевыми решениями в выборе технологий считаются такие решения, которые оказывают существенное влияние на разработку, распространение или поддержку проекта. Обычно эти решения трудно изменить в последующем. Как правило, к таким ключевым решениям относятся: выбор операционной системы, языка программирования и среды выполнения, веб-серверов, промежуточных программных компонентов и основных прикладных фреймворков, таких как веб-фреймворки или фреймворки поддержки параллельного выполнения.