Event-Driven Architecture

Перевод статьи: Event-Driven Architecture (Herberto Graca)

Практика использования событий для разработки приложений существует с конца 1980-х годов. Мы можем использовать события в любом месте frontend'а или backend'а. При нажатии кнопки, при изменении данных или выполнении какого-либо действия.

Но что эта практика из себя представляет? Когда и как мы должны ее использовать? Каковы приемущества и недостатки?

Что? Где? Когда?

Как и классы, компоненты должны иметь высокую внутреннюю связность и низкую внешнюю зависимость. Когда компоненты должны взаимодействовать, например, компонент **A** должен активировать какую-либо логику компонента **B**, обычным способом выполнить это является заставить **A** вызывать метод класса, который принадлежит **B**. Однако, если **A** знает о существовании **B**, то это значит, что они связанны друг с другом (**A** зависит от **B**), что делает систему более сложной для внесения изменений и поддержки. События помогают избежать зависимоти между компонентами.

Более того, побочным эффектом использования событий и отделения компонентов друг от друга является, например, то что, если у нас есть команда, работающая только над компонентом **B**, она может изменить то, как компонент B реагирует на событие в компоненте **A**, даже не разговаривая с командой,

ответственной за компонент **A**. Это значит, что компоненты могут развиваться независимо и наше приложение становится более органичным.

Даже в пределах одного компонента, иногда у нас есть код, который должен быть выполнен как результат какого-то действия, но он не должен быть выполнен немедленно, в процесе обработки запроса и возврата ответа. Самым ярким примером является отправка электронной почты. В этом случае мы можем немедленно вернуть ответ пользователю, а электронное письмо отправить позже (асинхронно) и избежать ожидания пользователем отправки электронного письма.

Чтобы предотвратить превращение нашей кодовой базы в большую кучу спагетти-кода, мы должны ограничить использование событий четко определенными ситуациями. В соответствии с моим опытом, есть три случая, в которых можно использовать события:

1. Для отделения компонентов друг от друга;
2. Для выполнения асинхронных заданий;
3. Для отслеживания изменений состояния (audit log).

Отделение компонентов друг от друга

Когда компонент **A** выполняет логику, которая должна инициировать логику компонента **B**, то вместо того, чтобы вызывать ее напрямую, мы можем инициировать отправку Event'а (события) в Event Dispatcher (диспетчер событий). Компонент **B** будет прослушивать это конкретный Event в Event Dispatcher'е и будет выполнять необходимую логику всякий раз, когда этот Event происходит.

Это значит, что компоненты **A** и **B** зависят от Event'а и Event Dispatcher'а, и ничего не знают друг о друге, т.е. **A** и **B** отделены друг от друга.

В идеальном случае, ни Event, ни Event Dispatcher не должны принадлежать ни одному из компонентов:

- Event Dispatcher должен быть библиотекой, которая не зависит от нашего приложения;
- Но Event должен быть частью нашего приложения, но быть объявленным где-то вне любого из компонентов, чтобы компоненты оставались отделенными друг от друга. Event'ы являются общими для нескольких (или всех) компонентов и находятся в ядре приложения. Т.е. Event'ы должны быть частью того, что в DDD называется **Shared Kernel** (общее ядро). Таким образом компоненты будут зависеть от Shared Kernel, но не буду ничего знать друг о друге;

Однако, в монолитном приложении, для удобства, допустимо размещать Event'ы в компоненте, который их запускает.

Shared Kernel...

Заключает в явные границы некоторый набор элементов модели предметной области, который команды согласны сделать общедоступным. Shared Kernel должен быть как можно меньше. Этот общий набор (Shared Kernel) имеет особый статус и не должен быть изменен без консультации с другой командой.

Eric Evans 2014, Domain-Driven Design

Выполнение асинхронных заданий

Бывает так, что есть логика, которую необходимо выполнить и выполнение занимет определнное время, но мы не хотим заставлять пользователя ждать на протяжении этого времени. В таком случае нам необходимо выполнить эту

логику как асинхронное задание и мгновенно вернуть пользователю сообщение, что его запрос будет выполнен позже, асинхронно.

Например, создание заказа в интернет-магазине может быть выполнено синхронно, а отправка уведомления о заказе на электронный почтовый ящик может быть осуществлена асинхронно.

В таком случае создается Event, который помещается в очередь и далее обработчик забирает его из очереди и выполняет, когда у системы появляются необходимые для этого ресурсы.

И в таком случае, не имеет никакого знаения то, относится ли исполняемая логика к одному и тому же или к разным ограниченным контекстам, в любом случае исполняемая логика разделена (синхронная часть отделена от асинхронной).

Отслеживание изменений состояний (audit log)

При традиционном способе хранения данных у нас есть сущности, содержащие некоторые данные. Когда данные в этих сущностях меняются, мы просто обновляем строку таблицы БД, чтобы сохранить новые значения. Проблема здесь в том, что мы не храним данные о том, что конкретно изменилось и когда. Однако, мы можем хранить события, содержащие изменения, в виде audit log'а. Подробнее об этом далее.

Слушатели VS Подписчики

При реализации Event-Driven Architecture'ы часто возникают споры по поводу использования слушивателей событий (Listeners) или подписчиков событий (Subscribers), поэтому вот мой взгляд на это:

- **Event Listener'ы** реагируют только на один Event и имеют несколько методов для реакции на него. Таким образом, мы должны назвать Listener в соответствии с имени Event'а, например, если у нас есть `UserRegisteredEvent`, также должен быть и `UserRegisteredEventListener`, что позволит легко узнать, какой Event слушает Listener. Реакции (методы) на Event'ы должны отражать то, что метод делает, например `.notifyNewUserAboutHisAccount()` и `.notifyAdminThatNewUserHasRegistered()`. Это обычный подход для большинства случаев, т.к. он позволяет Listener'у оставаться небольшим классом, который инкапсулирует одну обязанность (реагировать на конкретный Event). Кроме того, если у нас есть компонентная архитектура, каждый компонент (если необходимо) будет иметь свой собственный Listener для Event'а, который может быть сгенерирован любым из компонентов системы.
- **Event Subscriber'ы** реагируют на несколько Event'ов и имеют несколько методов для реакции на него. Процесс выбора имени для Subscriber'а более тяжелый, т.к. имя должно отражать единственную обязанность Subscriber'а. Использование Event Subscriber'ов должно быть менее распространенным подходом, особенно в компонентах, поскольку это может легко нарушить принцип единой ответственности (SRP). Примером правильного использования Event Subscriber'а может быть Subscriber для управления транзакциями. Например, у нас есть Event Subscriber с названием `RequestTransactionSubscriber`, который реагирует на такие Event'ы, как `RequestReceivedEvent`, `ResponseSentEvent` и `KernelExceptionEvent`. Реакцией на эти Event'ы будут, соответственно, начало, фиксация и откат транзакций, т.е. вызовы методов `.startTransaction()`, `.finishTransaction()` и `.rollbackTransaction()`. В данном случае Subscriber реагирует на несколько Event'ов, но все еще сосредоточен на выполнении единственной обязанности (SRP) – управлении транзакцией запроса.

Шаблоны

Мартин Фаулер определил 3 шаблона, относящихся к событиям:

- Событие-оповещение (Event Notification);
- Событие для переноса состояния (Event-Carried State Transfer);
- Накопление событий (Event-Sourcing).

Все эти шаблоны отражают следующие ключевый концепции:

1. Event'ы сигнализируют, что что-либо произошло (создаются после этого);
2. Event распространяется ко всем частям кода, которые ждут данного Event'а (несколько частей кода могут среагировать на один и тот же Event).

Event Notification

Пусть у нас есть ядро приложения с четко определнными компонентами. В идеальном случае, эти компоненты полностью разделены и ничего не знают друг о друге, но в тоже время часть функциональности одних компонентов требует выполнения определенной логики инкапсулированной в других компонентах.

Это наиболее распространенный случай, который был описан ранее: когда компонент **A** выполняет логику, которая должна инициировать логику компонента **B**, то вместо того, чтобы вызывать ее напрямую, мы можем инициировать отправку Event'а (события) в Event Dispatcher (диспетчер событий). Компонент **B** будет прослушивать это конкретный Event в Event Dispatcher'е и будет выполнять необходимую логику всякий раз, когда этот Event происходит.

Главная отличительная особенность данного шаблона в том, что **Event переносит минимальное количество информации**. Т.е. такое количество

информации, которое позволит Listener'у среагировать на данный Event, например, ID сущности и, может быть, дата и время создания события.

Приемущества:

- *Повышение устойчивости*. Например, компонент **A** выполняет свою логику и отправляет в очередь Event, который необходим для выполнения дополнительной логики в компоненте **B**. И даже, если эта логика не будет выполнена в **B** сразу (из-за бага, недостатка ресурсов и т.п.), то она будет выполнена в будущем, когда проблемы в компоненте **B** будут устранены.
- *Уменьшенная задержка*, если событие помещено в очередь, то пользователю не нужно ждать выполнения этой логики;
- Команды могут развивать компонент независимо друг от друга, что делает работу команд разработки более простой, быстрой, надежной и органичной.

Недостатки:

- Если использовать этот шаблон не по назначению, то это может превратить кодовую базу в спагетти-код.

Event-Carried State Transfer

Пусть у нас опять есть ядро приложения с четко определнными компонентами. Но в этот раз компонентам необходимы данные, которыми владеют другие компоненты. Наиболее естественным способом получения этих данных является вызов методов классов других компонентов, и это означает, что запрашивающий компонент будет знать о запрашиваемом компоненте, т.е. компоненты будут связаны друг с другом!

Другим путем передачи необходимых данных является создание Event'а, когда компонент, который владее данными изменяет их. **Этот Event будет нести**

измененные данные вместе с собой. И далее компонент, который заинтересован в этих данных будет прослушивать необходимый Event, и в качестве реакции на его получение, будет сохранять локальную копию этих данных. При таком подходе компонент имеет локальную копию данных другого компонента и обновляет ее при получении соотетствующего Event'а, т.е. компоненту не нужно обращаться за необходимыми ему данными в компоненту-владельцу, т.к. он располагает локальной копией этих данных.

Приемущества:

- *Повышение устойчивости*. Например, компонент **A** может продолжать функционировать, используя, необходимую ему, локальную копию данных, которые принадлежать компоненту **B**, даже если **B** недоступен (из-за бага и т.п.);
- *Уменьшенная задержка*, т.к. компоненту **A** не нужно запрашивать необходимые ему данные у компонента **B**, а вместо этого можно использовать локальную копию этих даных.

Недостатки:

- Несколько копий одних и тех же данных, которые должны использоваться только для чтения;
- Повышение сложности реализации компонента, т.к. теперь ему необходимо поддерживать локальную копию данных (в большенстве случаев это достаточно стандартная логика), которыми обладает другой компонент.

Возможно, этот шаблон и не нужен, если оба компонента выполняются в одном и том же процессе, что обеспечивает быструю связь между компонентами, но даже тогда может быть интересно использовать это шаблон для повышения независимости и обслуживаетости компонентов, или в качестве подготовки к разделению этих компонентов на различные микросервисы, когда-нибудь в

будущем. Все зависит от того, каковы наши текущие потребности, будущие потребности и как далеко мы хотим/должны зайти приследуя цель отделения компонентов друг от друга.

Event Sourcing

Пусть у нас есть сущность (Entity), которая находится в исходном состоянии. Будучи Entity, она имеет свою индивидуальность, т.е. представляет определнный объект/понятие реального мира (предметной области). На протяжении своего существования данные (состояние) этой Entity изменяются, и, традиционно, текущее состояние Entity просто хранится в виде строки в БД.

Transaction log (лог транзакции)

И в большенстве случаев это подходит, но что делать, если нам необходимо знать, как Entity пришла в данное состояние (т.е. например, мы хотим знать все зачисления и списания определнного банковского счета)? В данном случае это невозможно, т.к. в БД хранится только текущее состояние Entity.

Используя Event Sourcing вместо сохранения состояние Entity, мы фокусируемся на **сохранении изменений состояния Entity и вычислении текущего состояния на основе этих изменений**. Каждое изменение состояния – это Event, который сохраняется в определенной таблице БД. И когда нам необходимо получить текущее состояние Entity, мы вычисляем его на основе сохраненных Event'ов.

Хранилище событий является основным источником истины, и состояние системы расчитывается на его основе. Для программистов лучшим примером

является система управления версиями. Журнал всех фиксаций является хранилищем событий.

Greg Young 2010, CQRS Documents

Удаление событий

Если у нас есть Event изменения сосотояния Entity, который является ошибкой, мы не можем просто удалить такой Event, т.к. это изменит историю изменений состояния Entity, что противоречит самой идее Event Soursing'а. Вместо этого мы создаем Event, который отменяет Event, который мы хотели удалить. Этот процесс называется Реверсирующей Транзакцией (Reversal Transaction) и он не только возвращает Entity в нужное состояние, но и оставляет след, который показывает, что Entity находилась в этом состоянии в определенный момент времени.

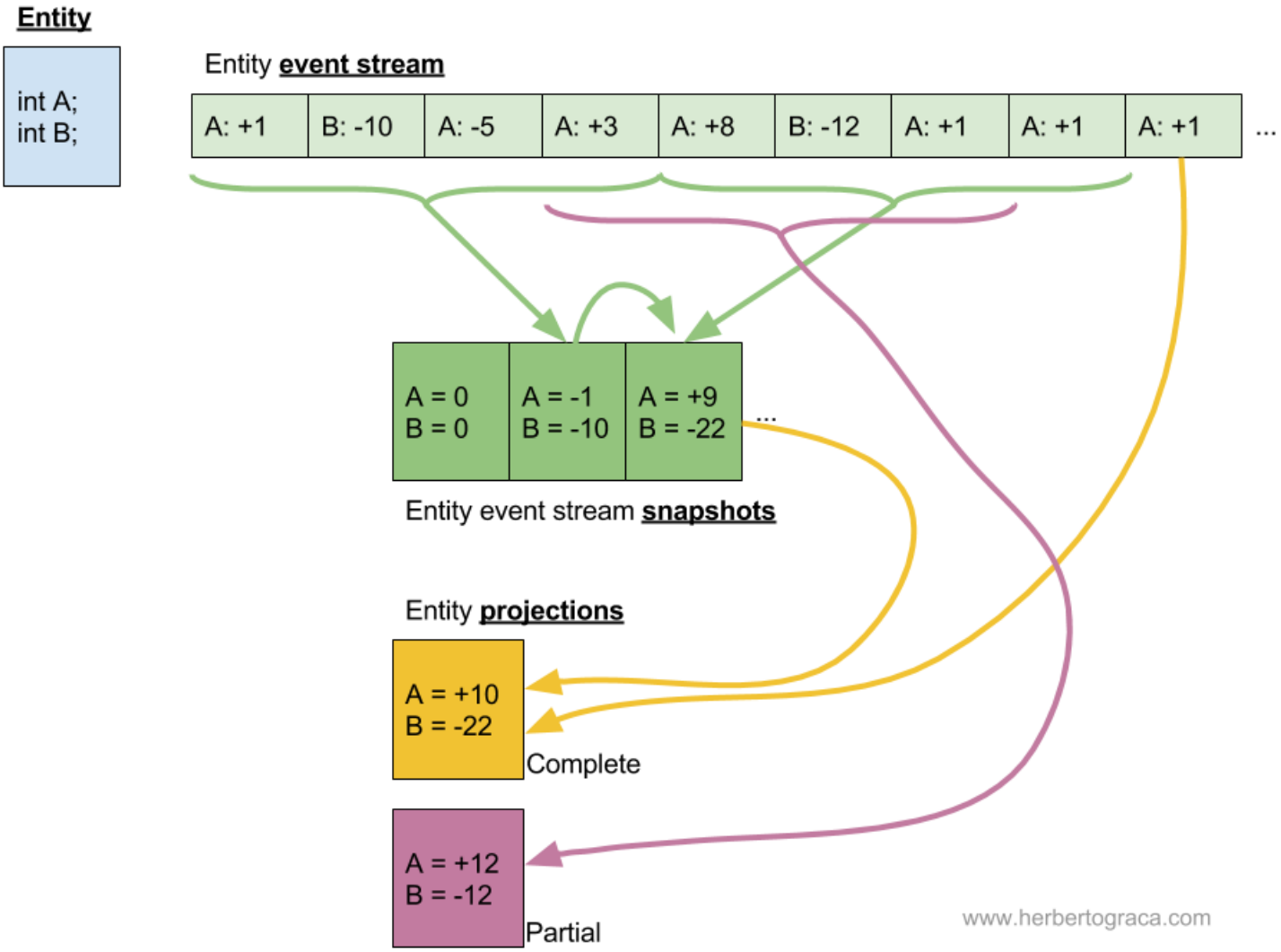
Snapshot'ы

Если у нас имеется множество Event'ов изменения состояния той или иной Entity, то вычисление текущего состояния этой Entity будет затратным. Для решения этой проблемы мы может через каждые «X» Event'ов создавать Snapshot состояния Entity в этот момент времени. Таким образом, когда нам понадобится состояние Entity нам потребуется выполнить вычисления по Event'ам до последнего snapshot'а.

Projections

В Event Sourcing'е есть концепция прогнозов (Projections), которая представляет собой расчет состояния Entity "с" и "до" определнного момента. Это значит, что Snapshot и текущее состояние Entity подподает под определение Projection, но

наиболее значимым в концепции прогнозов является то, что мы можем анализировать изменения состояния Entity в течении определнного периода времени, что дает нам возможность делать предположения о будующих изменения состояния этой Entity, и это может быть чрезвычайно ценной возможностью для бизнеса.



Достоинства и недостатки:

Event Sourcing может быть очень полезен как для бизнеса, так и для процесса разработки:

- можно использовать журнал Event'ов для восстановления прошлых состояний;
- можно исследовать альтернативные истории, вводя гипотетические Event'ы;

- Можно использовать поток Event'ов в целях debug'а.

Но не все на самом деле так хорошо и надо быть в курсе скрытых проблем:

- **Внешние изменения**

  Когда наши Event'ы запускают обновления во внешних системах, мы не хотим повторно запускать эти обновления при воспроизведении Event'ов в процессе создания прогнозов. На этом этапе мы можем просто отключить внешние обновления, когда мы находимся в "режиме воспроизведения" Event'ов (можно инкапсулировать эту логику в Gateway'е).

- **Внешние запросы**

  Когда в результате Event'а осуществляется запрос к внешней системе, например, получение рейтингов фондовых облигаций, что произойдет, когда мы воспроизведем такой Event, чтобы создать прогноз? Мы могли бы получить те же рейтинги, которые использовались, когда Event был создан, может быть, годы назад. Таким образом, либо внешнее приложение может дать нам эти значения, либо нам нужно сохранить их в нашей системе, чтобы мы могли имитировать удаленный запрос (и опять же такую логику можно реализовать в Gateway'е).

- **Изменения в коде приложения**

  Мартин Фаулер выделяет 3 типа изменений кода: новые функции, исправления ошибок и временная логика. Реальная проблема возникает при воспроизведении Event'ов, которые должны воспроизводиться с различными правилами бизнес-логики в разные моменты времени, т. е. например, в прошлом году налоговые расчеты отличаются от этого года. В таких случаях подойдет шаблон проектирования **Strategy**.

Поэтому я советую соблюдать осторожность и следовать следующим правилам, когда это возможно:

- Event'ы должны быть как можно проще (глупее), т.е. они должны знаять только об изменении состояния. Таким образом, мы можем безопасно воспроизвести любой Event и ожидать, что результат будет таким же, даже если бизнес-правила изменились (хотя нам нужно будет сохранить устаревшие бизнес-правила, чтобы мы могли применять их при воспроизведении прошлых Event'ов);
- Взаимодействие с внешними системами не должно зависеть от Event'ов. В результате мы сможем безопасно воспроизводить Event'ы без опасения повторного запуска внешней логики, и нам не нужно будет гарантировать, что ответ от внешней системы будет такой же, как и когда Event был создан первоначально.

И, конечно, как и любой другой подход, нам не нужно использовать Event-Driven Architecture везде. Мы должны использовать его там, где это имеет смысл, где это приносит нам преимущество и решает больше проблем, чем создает их.

Заключение

И опять же, этот подход используется для повышения внутренней связности и снижения внешней зависимости.