Event-Driven Architecture

Перевод статьи: [Event-Driven Architecture](https://herbertograca.com/2017/10/05/event-driven-architecture/) ([Herberto Graca](https://herbertograca.com/))

Практика использования событий для разработки приложений существует с конца 1980-х годов. Мы можем использовать события в любом месте frontend’a или backend’a. 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 **А** должен активировать какую-либо логику компонента **В**, обычным способом выполнить это является заставить **А** вызывать метод класса, который принадлежит **В**. Однако, если **А** знает о существовании **В**, то это значит, что они связанны друг с другом (**А** зависит от **В**), 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 отсеживания изменений состояния (audit log).

**Отделение компонентов друг от друга**

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

Это значит, что компоненты **А** и **В** зависят от Event’a и Event Dispatcher’а, и ничего не знают друг о друге, т.е. **А** и **В** 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 забиет 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’a. Подробнее об этом далее.

**Слушатели VS Подписчики**

При реализации архитектуры, управляемой событиями, часто возникают споры по поводу использования слушивателей событий (Listeners) или подписчиков событий (Subscribers), поэтому вот мой взгляд на это: