Типы интеграции

1. Внутренняя и внешняя.

Внутренняя - это когда к примеру, система технического учёта и система CRM были интегрированы между собой, чтобы оператор call-центра мог проверить техническую возможность проведения интернета человеку прямо во время регистрации заявки от него. Это обе наши внутренние системы,и в итоге не нужно было подключать каким-то образом третью сторону для этой интеграции.

А когда чуть позже понадобилось оператору карту показывать, то потребовалась внешняя интеграция (со сторонним сервисом - GoogleMap)

2. Простая и сложная (очень условное название)

Интеграция может быть с использованием no-code инструментов (например, Integromat), а может требовать разработчиков и написания кода. Вот тут классная подборка no-code инструментов.

Кстати, можно выделить еще ручную интеграцию. Например, сотрудник склада, переносящий данные из системы 1С:Склад в CRM, занимается интеграцией этих систем.

3. Классификация по целям

Интеграция может иметь разные цели. В основном, речь идёт про организацию сквозного бизнес-процесса (процесс заказа продуктов, например, требует работы нескольких систем). Или про организацию хранения данных (MDM-класс систем, DWH-системы и т.п.). Или про представление данных в одном окне (у нас оператор мог видеть dashboard с кучей информации из разных систем): всё это ради экономии времени, денег или предоставлении какой-то новой, ранее невозможной услуги (сервиса) или продукта

4. Интеграция может быть стихийной или плановой

Вам срочно надо объединить две системы, чтобы организовать процесс? О, неплохо бы ещё систему CRM к ним интегрировать? Делаем! И так год за годом. Куча систем, как клубок, наматывается: то там интегрируем, то тут. Плановая, соответственно, возникает, когда есть план.

5. Интеграции могут отличаться по структуре связи

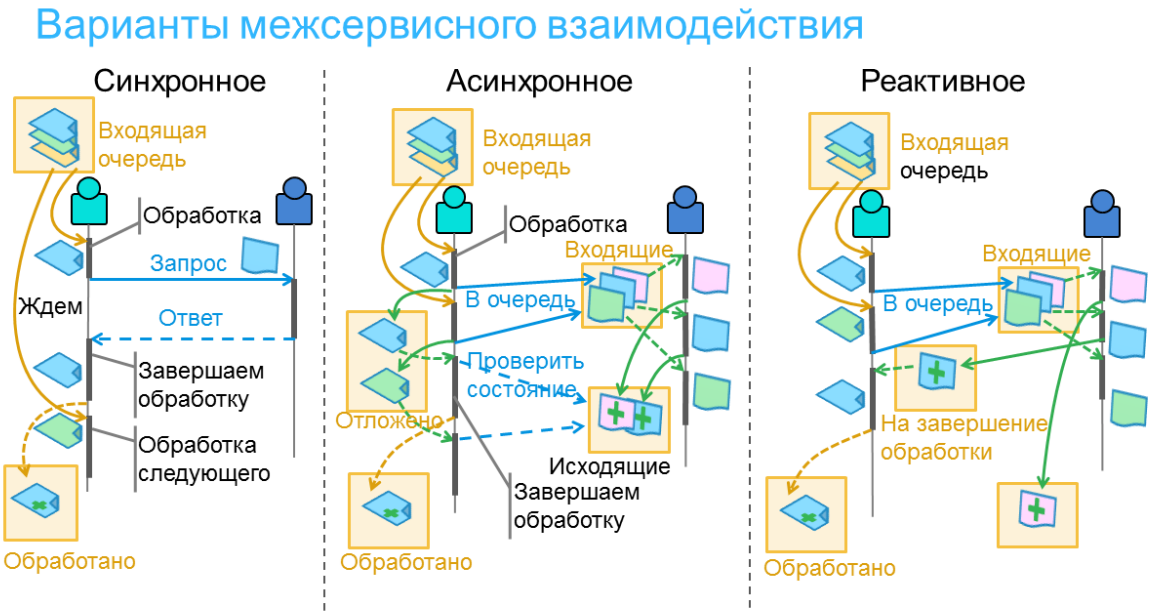
Взаимодействие систем по принципу **точка-точка** исторически появилось раньше остальных. Подразумевает этот принцип всего лишь ситуацию, когда каждая система попарно интегрируется с другими.
А интегрируется с Б, Б с В и др., т.е. напрямую. Система А знает, как обратиться к системам Б и В, завязана на их интерфейсы, знает адрес сервера приложений.

Звезда — это когда мы добавляем, к примеру, шину (ESB). Какую-то прослойку, которая

позволяет всем системам интегрироваться между собой. Это как единая точка. Шина — это тоже ПО. Звезда имеет много плюсов (например, если эта CRM система вам больше не нужна, меняем её на новую — всё не развалится). Но есть и минусы. Например, у вас есть единая точка отказа. Случись что-то с шиной — и не работает вообще всё (весь контур систем, которые интегрируются через эту шину).

Смешанная — это когда у нас на проекте есть шина, общая точка, где системы объединяются. Но и есть попарно интегрированные системы.

6. Синхронная или асинхронная интеграция



Синхронное взаимодействие

Синхронное взаимодействие — самое простое. Оно скрывает все детали удаленного вызова, что для вызывающего сервиса превращается в обычный вызов функции с получением ответа. Но очевидная проблема синхронности в том, что удаленный сервис может отвечать не очень быстро даже при простой операции — на время ответа влияет загруженность сетевой инфраструктуры, а также другие факторы. И все это время вызывающий сервис находится в режиме ожидания, блокируя память и другие ресурсы (хотя и не потребляя процессор). В свою очередь, блокированные ресурсы могут останавливать работу других экземпляров сервиса по обработке сообщений, замедляя тем самым уже весь поток обработки. А если в момент обращения к внешнему сервису у нас есть незавершенная транзакция в базе данных, которая держит блокировки в БД, мы можем получить каскадное распространение блокировок.

Другая проблема связана с падениями удаленных вызовов. Падение вызываемого сервиса еще укладывается в общую логику обработки — всякий вызов процедуры может породить ошибку, но эту ошибку можно перехватить и обработать. Но ситуация становится сложнее,

когда вызываемый сервис отработал успешно, но вызывающий за это время был убит в процессе ожидания или не смог корректно обработать результат. Поскольку этот уровень скрыт и не рассматривается разработчиками, то возможны эффекты типа резервов для несуществующих заказов или проведенные клиентом оплаты, о которых интернет-магазин так и не узнал.

И третья проблема связана с масштабированием. При синхронном взаимодействии один экземпляр вызывающего сервиса вызывает один экземпляр вызываемого, но который, в свою очередь, тоже может вызывать другие сервисы. И нам приходится существенно ограничивать возможность простого масштабирования через увеличение экземпляров запущенных сервисов, при этом мы должны проводить это масштабирование сразу по всей инфраструктуре, поддерживая примерно одинаковое число запущенных сервисов с соответствующей затратой ресурсов, даже если проблема производительности у нас только в одном месте.

Поэтому синхронное взаимодействие между сервисами и системами — зло. Оно ест ресурсы, мешает масштабированию, порождает блокировки и взаимное влияние разных серверов.

Асинхронное и реактивное взаимодействие

Асинхронное взаимодействие предполагает, что вы посылаете сообщение, которое будет обработано когда-нибудь позднее.

Есть два основных способа:

- обычное **асинхронное** взаимодействие, когда передающая система сама периодически опрашивает состояние документа;
- и **реактивное**, при котором принимающая система вызывает callback или отправляет сообщение о результате обработки заданному в исходном запросе адресату.

Оба способа вы можете увидеть на схеме вместе с очередями и логикой обработки, которая при этом возникает.

Какой именно способ использовать — зависит от способа связи. Когда канал однонаправленный, как при обычном клиент-серверном взаимодействии или http-протоколе, то клиент может запросить сервер, а вот сервер не может обратиться к клиенту — взаимодействие получается асинхронным.

Впрочем, такой асинхронный способ легко превращается в синхронный — достаточно в методе отправки сообщения поставить таймер с опросом результата. Сложнее превратить его в реактивный, когда внутри метода отправки находится опрос результата и вызов callback. И вот это второе превращение — далеко не столь безобидно, как первое, потому что использующие реактивную интеграцию рассчитывают на ее достоинства: пока ответа нет, мы не тратим ресурсы и ждем реакции. А оказывается, что где-то внутри все равно работает процесс опроса по таймеру…

В реактивном взаимодействии есть не только переключение потоков, но и скрытые очереди. А скрытые очереди хуже явных, потому что когда возникает дефицит ресурсов и возрастает нагрузка, все тайное становится явным. Особенно в интеграции, когда это используется для взаимодействия между узлами и сервисами, которые потенциально находятся на разных узлах.

7. Интеграция систем, подсистем, сервисов, микросервисов

Если вы интегрируете свою систему CRM с чужой системой MDM, то это интеграция систем.

Если сервису Яндекс.Кошелек вдруг нужны данные с сервиса Госуслуги, то перед нами интеграция сервисов.

Если наша система построена по SOA, то мы имеем распределенную систему, подсистемы которой интегрируются между собой (например, с помощью шины, о которой будет ниже). Для систем на микросервисах все тоже самое.

8. Паттерны интеграции по типу обмена данными

Речь идет о таких паттернах, как обмен файлами, обмен через общую базу данных, удалённый вызов процедур, обмен сообщениями.

9. Паттерны по методу интеграции

Никогда не встречала, чтобы кто-то всерьез пользовался этой классификацией, но так как она существует, поделюсь с вами:
• Интеграция систем по данным (data-centric)
• Объектно-центрический (object-centric)
• Функционально-центрический (function-centric) подход
• Интеграция на основе единой понятийной модели предметной области (concept-centric).

ЗАЧЕМ ИСПОЛЬЗОВАТЬ ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ?

Люди из сферы IT-индустрии прекрасно знают, насколько переменчивой может быть рабочая атмосфера на проектах. Каждый день ранее утвержденные требования могут пересматриваться, редактироваться и банально терять свою актуальность.

Все это приводит к созданию большого количества строчек программного кода, который в любом случае необходимо тщательно тестировать на предъявленные заранее требования.

Идеальным инструментом для подобных целей как раз и выступает интеграционное тестирование, позволяющее классифицировать программный код на блоки (модули). Интеграция проверки ПО очень важна, так как в релиз должен поступать исключительно работоспособный и качественный продукт, ликвидность которого в своей нише будет максимальной (так следует в теории).

ВИДЫ ПОДХОДОВ К ИНТЕГРАЦИОННОМУ ТЕСТИРОВАНИЮ

Есть сразу 4 основных типа и подхода к процессу интеграционного тестирования, которые стоит рассмотреть более детально.

Типы интеграционного тестирования:

- **Большой взрыв** (англ. Big bang approach);
- **Снизу вверх** (англ. Bottom-Up Approach);
- **Сверху вниз** (англ. Top-Down Approach);
- **Смешанный / сэндвич** (англ. Hybrid / Sandwich).

БОЛЬШОЙ ВЗРЫВ (АНГЛ. BIG BANG APPROACH)

Процесс разработки ПО предполагает, что созданные и запрограммированные модули и системные компоненты соединены между собой. При объединении эти модули тестируются как единое целое. После проведения юнит-тестов, модули также проверяются вместе, еще до образования целостной программной системы.

При интеграционном тестировании работы проводятся исключительно с отдельными модулями, а при проверке всего ПО тестируются все системные параметры.
Представленная ниже диаграмма красноречиво демонстрирует, что именно означает метод «большого взрыва» в процессе проведения интеграционных проверок.

Метод большого взрыва

К слову, подобный подход имеет несколько преимуществ и даже недостатков.

Преимущества:

1. Весьма удобен в использовании при тестировании небольших систем.
2. Быстрое нахождение ошибок, а значит, существенная экономия время, которое может быть потрачено на разработку и доработку уже используемого функционала.

Недостатки:

1. Так как модули завязаны на одной системе, порой очень трудно найти источник дефектов.
2. Если в системе используется много модулей, может уйти достаточно времени, чтобы пересмотреть все реализованные функциональности.

СНИЗУ ВВЕРХ (АНГЛ. BOTTOM-UP APPROACH)

Подобный подход подразумевает проверку низкоуровневых систем для начала: вместе и по отдельности. Другими словами процесс тестирования начинается с внутреннего уровня и постепенно доходит до наиболее критичных позиций.

Представленная далее схема красноречиво свидетельствует о том, что модули верхнего ранга не могут быть интегрированы в ПО до тех пор, пока не будет завершено тестирование модулей нижнего порядка.

Подход снизу вверх

При подходе «снизу вверх» может использоваться ***Драйвер***, который выступает в роли специального «соединителя» между модулями нижнего и верхнего уровней.

Преимущества:

1. Создание отдельных модулей может совершаться при применении метода интеграционного тестирования по схеме «снизу вверх», так как тестирование самых критических моментов начинается с тестов модулей нижнего порядка.
2. Если определенный модуль перестает функционировать, его ошибка может быть сразу же исправлена.
3. Требуется минимальное время на идентификацию и устранение ошибок.

Недостатки:

1. Общее время проверки всех модулей довольно долгое, так как продукт не может быть представлен в релиз пока не пройдет тестирование от самого нижнего до самого верхнего модулей.
2. Если ПО содержит много небольших модулей мелкого уровня, которые очень сложны в своей имплементации, то для завершения процесса тестирования может потребоваться больше времени.

СВЕРХУ ВНИЗ (АНГЛ. TOP-DOWN APPROACH)

Это полная противоположность вышеописанной методике. Суть подхода «сверху вниз» заключается в первостепенном тестировании всех верхних модулей, и только затем QA специалист может приступать к проверке работоспособности нижестоящих модулей.

Большинство модулей нижнего уровня тестируются по отдельности, а затем выполняются проверки в совокупности реализованных модулей. По аналогии с подходом «снизу вверх»,

данный метод также зависим от вызова специальной связующей функции под названием ***«Функция-заглушка»*** (англ. Stubs).
Заглушки – это специальные логические операторы с коротким программным кодом, которые применяются для приема входных данных нижними модулями от модулей верхнего уровня при интеграционном тестировании.

Подход сверху вниз

Преимущества:

1. Легко обнаружить неисправности или ошибки в работе системы.
2. В первую очередь проверяются важные модули, а лишь потом модули нижнего порядка.
3. По сравнению с другими подходами, время на тестирование интеграции очень коротко.

Недостатки:

1. Если в модули нижнего уровня заложена важная логика, она не может быть протестирована в первую очередь, пока не завершится работа над проверкой верхних порядков.
2. Использование «заглушек»становится обязательным на всех последующих проектах.

СМЕШАННЫЙ / СЭНДВИЧ (АНГЛ. HYBRID/SANDWICH APPROACH)

Данный подход еще принято именовать как ***тестирование смешанной интеграции***. Логики подходов «сверху вниз» и «снизу вверх» максимально объединены в этот подход. А значит, его запросто можно считать смешанным, своего рода гибридным методом в интеграционном тестировании.

Итак, самый верхний модуль тестируется отдельно, при этом модули нижнего уровня интегрируются и проверяются с модулями верхнего уровня.

Преимущество:

- Идеально подходит для больших проектов, работа над которыми длится очень долгое время.

Недостаток:

- Цена подобного тестирования очень высока, так как данный подход включает в себя сразу несколько модулей проведения интеграционного тестирования.