GeekBrains

Backend-разработка на Go. Модуль 2

Архитектурные паттерны монолитов и микросервисов

На этом уроке

1. Поговорим о базовых архитектурных паттернах, применяемых при разработке монолитных решений и решений с использованием микросервисов.
2. Научимся организации кода приложения на Go в зависимости от выбранной архитектуры.

Оглавление

Теория урока

- Эволюционный путь от монолита к микросервисам: вертикальное и горизонтальное масштабирование приложений
- Организация кода монолитных решений
- Архитектура микросервисных решений
- Основные шаблоны по взаимодействию бизнес-логики с хранилищем данных
  - Миграции
  - Active Record
  - Data Mapper
  - Unit of work
- MVC, MVP и MVVM
- Гексагональная архитектура
  - Слой предметной области: бизнес-логики, Domain Layer
  - Слой приложения (Application Layer)
  - Слой фреймворка (Framework Layer)
- Взаимодействие слоёв в гексагональной архитектуре: границы
  - Слой предметной области (бизнес-логики)
  - Слой приложения
  - Слой фреймворка
  - Шина команд
  - Зависимости (импорты) в гексагональной архитектуре
  - Жирные интерфейсы
- Чистая архитектура

Практическое задание

Глоссарий

Дополнительные материалы

Используемые источники

Теория урока

Это курс, посвящённый продвинутым практикам создания и развёртывания больших и сложных решений на Go. Во время прохождения курса вы изучите:

- развёртывание приложений (сервисов, микросервисов) в среде Kubernetes;
- взаимодействие между сервисами через протоколы OpenAPI, gRPC, GraphQL, а также посредством брокеров сообщений Kafka, NATS, RabbitMQ.

Научитесь проводить нагрузочное тестирование и оптимизировать работу с базами данных. Изучите практические инструменты для кэширования данных высокой степени доступности, используемых внутри сервисов. Вы также поймёте, как осуществлять полнотекстовый поиск в больших массивах данных, используя Elasticsearch.

В первом уроке мы поговорим о концептуальных, базовых основах построения архитектуры приложений, рассмотрим разные подходы к реализации слоёв абстракций различных архитектур на языке Go. Изучим принципы построения программного решения на Go «в целом» и для её отдельных частей. В последующих уроках этот базовый скелет пригодится нам, чтобы нарастить на него «мясо» и получить завершённую композицию из составляющих систему компонентов.

Эволюционный путь от монолита к микросервисам: вертикальное и горизонтальное масштабирование приложений

Когда-то давно все программы для компьютеров разрабатывались в виде отдельно запускаемых приложений. Каждое такое приложение объединяло в себе все свои функции и развёртывалось целиком, единым целым (комплектом для инсталляции). Большие приложения разбивались на различные компоненты и оформлялись в виде подключаемых библиотек, связанных через API вызова библиотечных функций.

Это был практически единственный способ разделить контекст разработки на части, с сохранением требуемой производительности решения. Скорость сетевого вызова была в то время ограниченной, а преобразование в какой-то промежуточный формат сообщения между программами сильно загружала процессор или диск. Инструментарий каждого такого решения был относительно небольшим, по сравнению с тем объёмом, что мы встречаем сегодня.

Шло время, решения обрастали новыми возможностями, росла внутренняя сложность решения на уровне кодовой базы и используемых баз данных. Возникали поистине огромные репозитории кода,

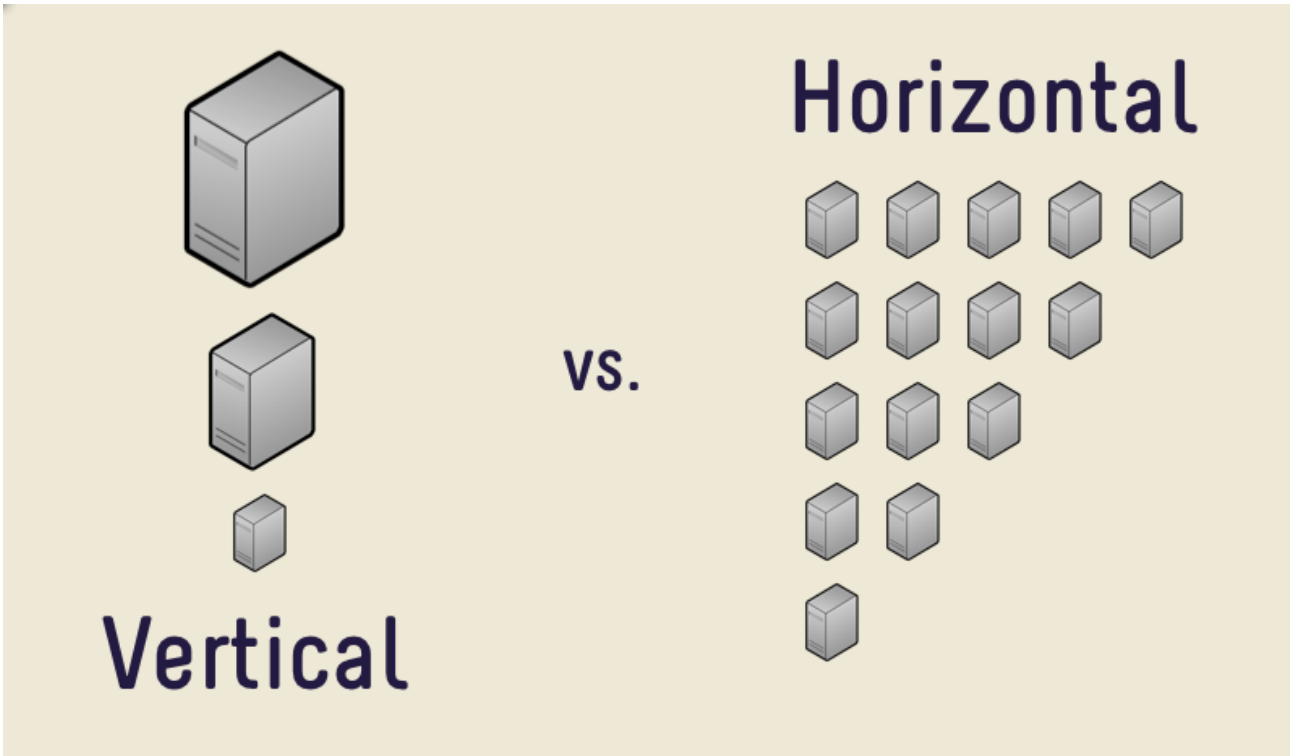
большие базы данных, относящиеся к одному решению. Эти решения долго компилировались (иногда днями) и с определённой периодичностью развёртывались в промышленную эксплуатацию. Процесс развёртывания существовал независимо от процесса создания решения и занимал дни или даже недели с момента завершения компиляции.

В то время использовался практически единственный путь масштабирования приложений — наращивание ресурсов вычислительной машины (сервера), на которых они запускались. Этот способ называется вертикальным масштабированием. В наше время он по-прежнему используется в некоторых ситуациях для масштабирования монолитных систем.

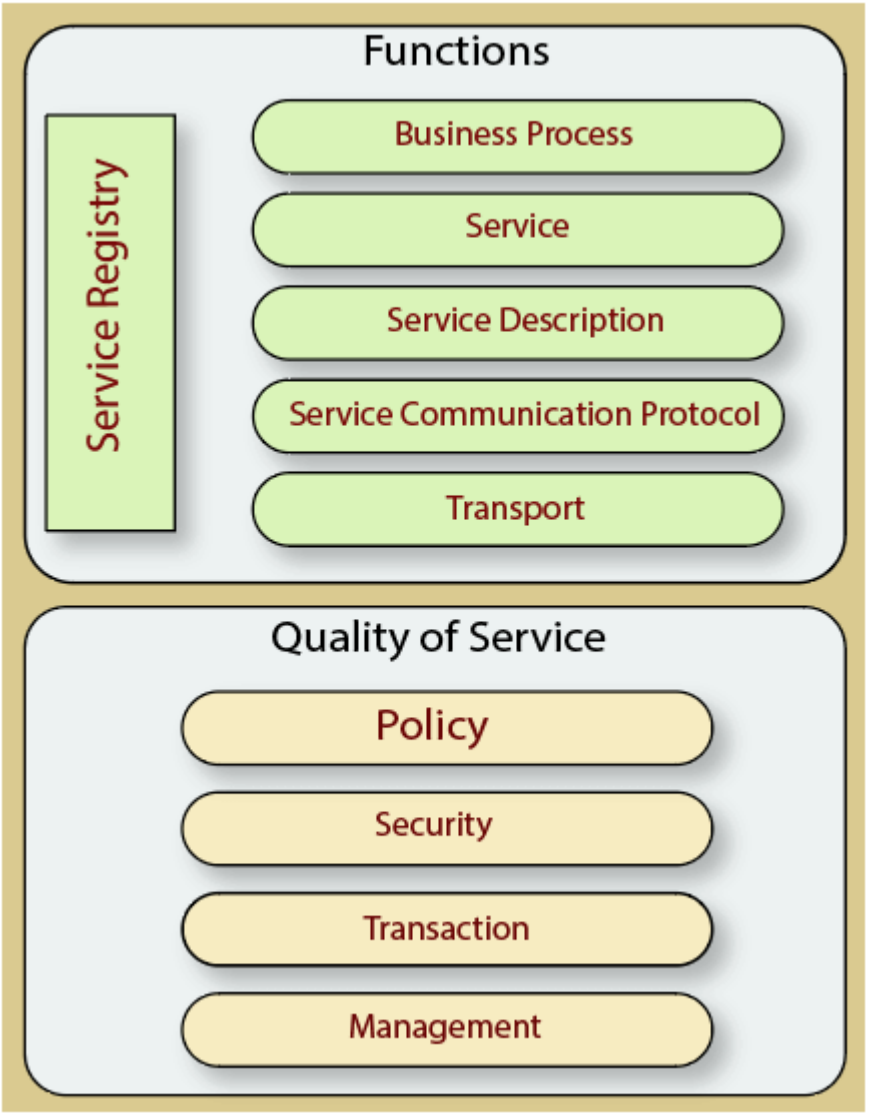
У вертикального масштабирования есть существенный недостаток — верхний предел роста и высокая совокупная стоимость владения при невысокой степени масштабирования решения. Иными словами, возрастающие затраты на вертикальное масштабирование и его дальнейшую поддержку — выход из строя компонентов с течением времени, модернизация и прочее — не увеличивает производительность решения до таких же показателей. Она растёт всё реже, в какой-то момент достигая потолка. Как правило, вертикальное масштабирование доходит до уровня покрытия ситуаций, относящихся к всплескам высокой нагрузки, и даже с некоторым запасом. Но так как такие ситуации всплеска нагрузки возникают нечасто, всё остальное время эти огромные мощности обычно простаивают и не оправдывают свою высокую стоимость.

Второй существенный недостаток вертикального масштабирования — необходимость единовременного внесения изменений для целей формирования очередного релиза системы, который развёртывается также в один момент времени. Вся связанная и несвязанная функциональность, как правило, к этому моменту уже готова. Такая стратегия релизов приводит к тому, что релизы выпускаются всё реже, часто задерживаются в ожидании готовности всего инструментария в целом, вызывая увеличение time to market.

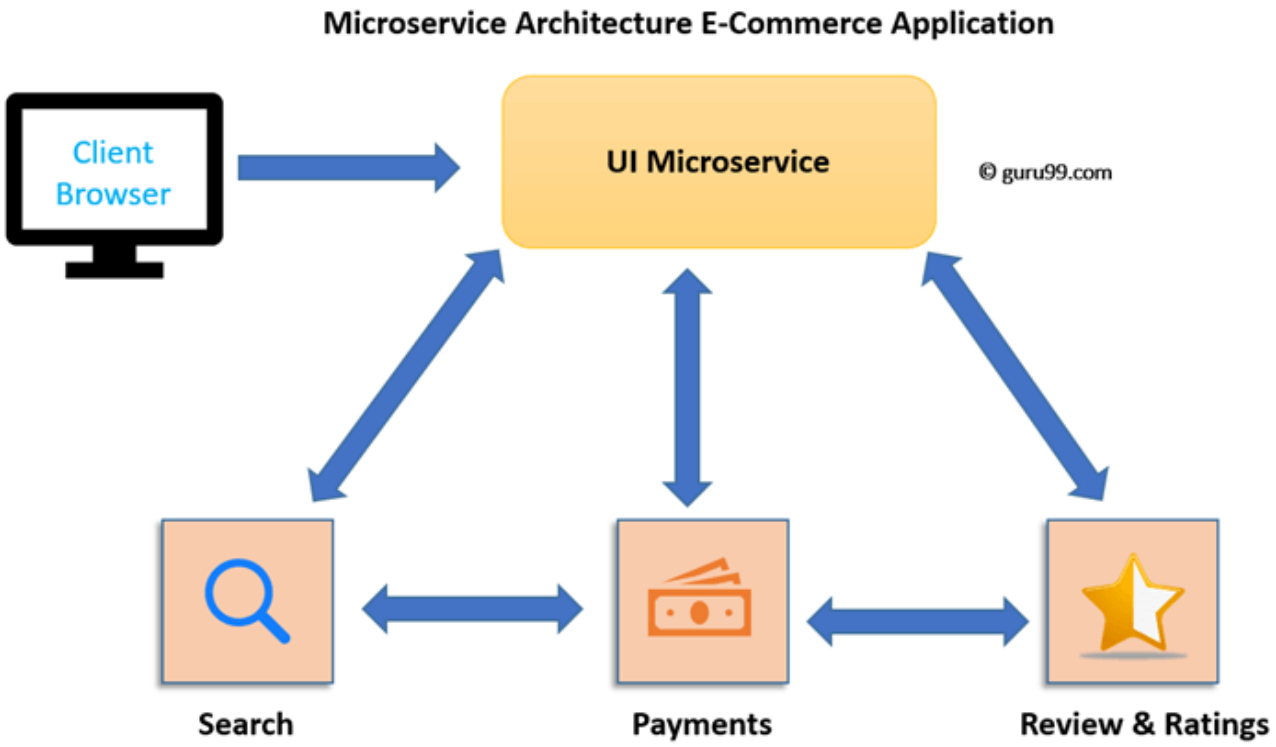
Пока система не достигла своих пределов вертикального масштабирования. Она обладает важными выгодными свойствами — высокой связанностью компонентов системы и соответствующей этому высокой производительностью решения в эксплуатации, относительной простотой в поддержке и развитии, меньшим объёмом трудозатрат. В таком решении легко поддержать принципы SOLID и ACID, нормальные формы представления реляционных отношений, отсутствие дублирования кода и т. д. Кодовая база такого решения обычно пишется в рамках одной экосистемы или фреймворка, что снижает порог входа и упрощает наём персонала поддержки.



Когда мы сталкиваемся с тем, что вертикально масштабированная система уже не может быть усовершенствована, чтобы повысить производительность, надо рассмотреть возможность перехода на модель горизонтального масштабирования. В этом случае компоненты системы изолируются в вид отдельных решений, каждое из которых запускается на одном или нескольких вычислительных устройствах (серверах), распределяя общую внешнюю нагрузку между этими экземплярами. Такое распределённое решение принято называть микросервисным, или просто разделённым на сервисы как в старых понятиях SOA.



SOA — это архитектурный паттерн в разработке программного обеспечения. В этом типе приложения компоненты предоставляют услуги другим компонентам по протоколу связи, обычно по сети. Принципы сервис-ориентации не зависят от какого-либо продукта, поставщика или технологии. Полная форма SOA — сервис-ориентированная архитектура. SOA упрощает взаимодействие компонентов программного обеспечения в различных сетях.



Микросервисы — это шаблон сервис-ориентированной архитектуры, где приложения создаются как совокупность различных наименьших независимых сервисных единиц. Это программный подход, который фокусируется на разложении приложения на однофункциональные модули с чётко определёнными интерфейсами.

Такие модули независимо разворачиваются и эксплуатируются небольшими группами, владеющими всем жизненным циклом службы.

Термин «микро» относится к размеру микросервиса, которым управляет одна команда разработчиков — от 5 до 10 разработчиков. В этой методологии большие приложения делятся на самые маленькие независимые единицы.

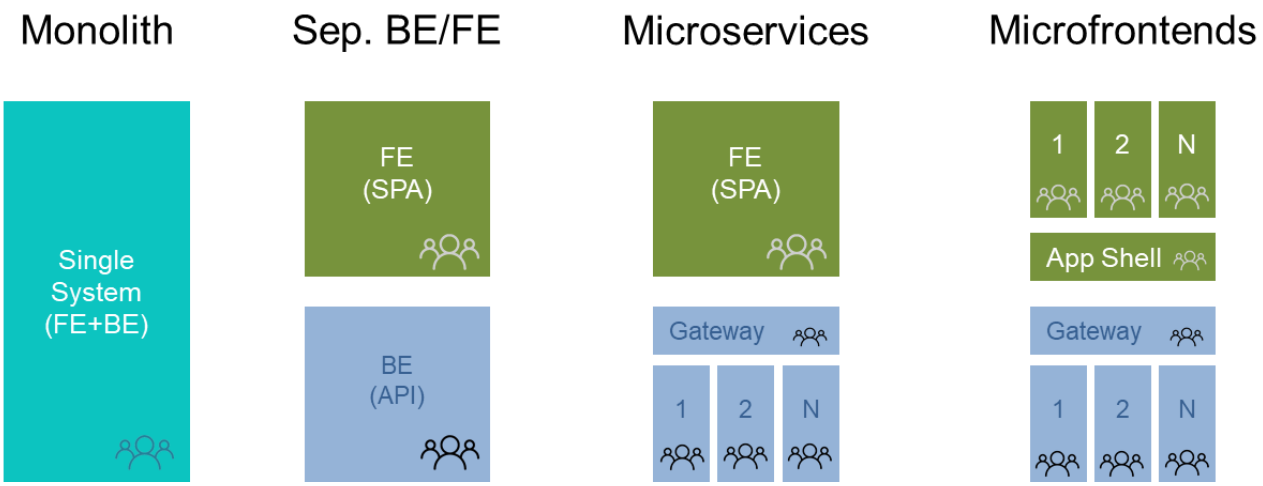
Главное преимущество микросервисов — изолированность контекста, кодовой базы и баз данных, возможность произвольного изменения и реконфигурации в пределах согласованного контракта взаимодействия с другими микросервисами в рамках системы. Такая архитектура позволяет реализовывать микросервисы на разных языках, экосистемах, фреймворках, постоянно обеспечивая следование за новыми технологиями и возможностями, применимыми к ситуации конкретного микросервиса, находить и использовать наилучшие из доступных инструментов их реализации.

С другой стороны, правильное разделение контекстов микросервисов становится критической и очень сложной задачей, если ситуация ещё не имеет реальных признаков, по которым можно произвести разделение независимых контекстов микросервисов. Попытки угадать контекст без наблюдения за реальной потребностью в нём часто приводит к ошибкам, исправление которых в микросервисной архитектуре обходится в несколько раз дороже аналогичных ошибок в монолитном решении.

Важно! Горизонтальное масштабирование только одного компонента монолитного решения, например, базы данных, в общем случае не решает основных проблем — единой кодовой базы, большого времени time to market, высокой совокупной стоимости владения. Такое масштабирование нельзя отнести к категории изменений монолитного решения, ведущих к микросервисной архитектуре. Оно лишь консервирует на некоторое время те проблемы, которые уже возникли.

Итак, поскольку до определённого момента монолитные решения имеют преимущества перед сложностью микросервисных решений, очевидно, что люди будут начинать делать сервисы в стиле монолитного решения. Необходимость перехода к микросервисной архитектуре, как правило, будет вызвана причинами достижения конкретного критического состояния в рамках монолитного решения, которое потребует срочных и кардинальных мер.

Проследим правильный путь от монолитного решения к микросервисному.

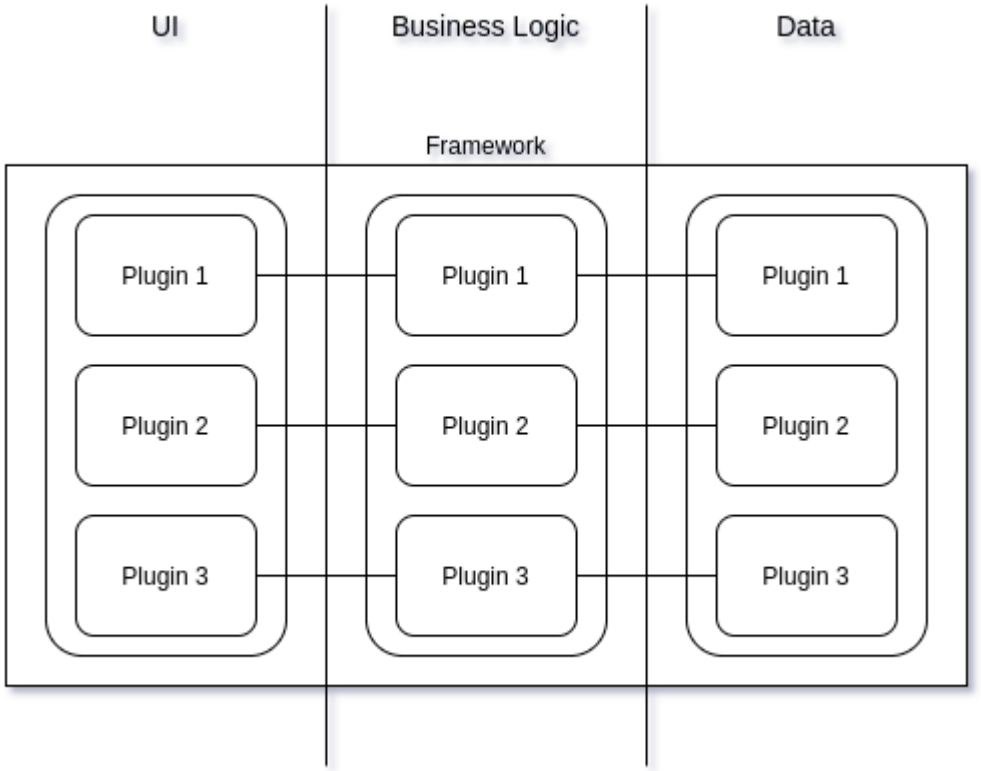


В монолитных решениях часть системы, относящаяся к интерфейсу пользователя, а именно визуализация информации и реагирование на команды пользователя, часто разрабатывается в рамках единой кодовой базы вместе с частью, относящейся к операциям по хранению, преобразованию, передаче информации между системами и базами данных. Первую часть принято именовать «фронтенд», вторую — «бэкенд». Примеры таких монолитных решений:

- системы, разработанные для ОС Windows или Linux/MacOS с графическим системным интерфейсом;
- системы на платформе 1С;
- старые версии SAP — даже с элементами трёхзвенной архитектуры, они всё ещё монолитны на уровне своих скриптовых языков;
- веб-сайты на PHP — там, где PHP реализует не только API, но и визуализацию вставками HTML/JS;
- прочее.

Первый шаг на пути к горизонтальному масштабированию монолитного решения — разделение кодовой базы на фронтенд и бэкенд-части, а также их независимое развёртывание на разных серверах. Отчасти это относится и к выделению сервера статического контента — картинок, скриптов и прочего.

Вторым шагом становится разбиение монолита на независимые модули внутри трёх звеньев:



Это всё ещё останется монолитным решением, но внутри оно уже будет подготовлено к выделению контекстов для отдельных внешних сервисов.

Третий шаг — оформление отдельных модулей системы в виде независимых небольших решений, у каждого из которых — свой жизненный цикл развития и поддержки. Общение между частями системы будет осуществляться через вызов функций, определённых контрактом взаимодействия (API).

Четвёртый шаг — переход от синхронных операций взаимодействия между сервисами «точка-точка» к асинхронным взаимодействиям посредством очередей сообщений.

Уже на этом моменте мы можем считать, что перешли к микросервисной архитектуре. Если же дополнительно к этому разбить фронтенд на модули с независимыми жизненными циклами, то получим микрофронтендную архитектуру в совокупности с микросервисной.

В любой архитектуре, так или иначе, приходится решать вопросы, как и где хранить данные, обрабатываемые сервисами. Это самое сложное. Вопросы API и бизнес-логики, с архитектурной точки зрения, не очень сложны, так как они по определению обладают поведением, связанным с событиями в будущем, а значит, гибкостью. А вот работа с состоянием сервисов тесно связана с хранением

данных прошлого, которые в целом не очень гибкие. И на стыке такого прошлого, настоящего и будущего возникает множество нюансов, которые позволяют классифицировать различные архитектурные паттерны. Об этом мы и поговорим дальше.

Организация кода монолитных решений

Монолитное решение на Go, как правило, состоит из таких компонент:

1. Внешнее API, то есть фронтенд и другие системы — Rest API, gRPC и т. д.
2. Функции для работы с СУБД, то есть SQL, NoSQL и прочие, включая пакеты и драйверы для конкретных систем хранения.
3. Механизмы кеширования данных СУБД в оперативной памяти.
4. Бизнес-логика обработки данных внутри системы, обработка потока входящих данных, генерация потоков исходящих данных.

В монолитном решении все данные — в прямом доступе у сервиса в полном объёме, модель данных едина для всех функциональных модулей монолита. Поэтому обеспечить атомарность и согласованность доступа к данным достаточно легко — через транзакции на уровне СУБД или виртуальные транзакции внутри бизнес-логики.

Весь код монолита разделяется на функции и методы, содержащиеся в разных пакетах и вызывающие друг друга напрямую. Для поддержания атомарности транзакций применяется общий контекст, передаваемый между этими функциями.

Как правило, все компоненты монолитного решения работают в рамках единого контекста с общим логгером, общей системой мониторинга показателей, одной базой данных в каждой подходящей категории, а именно в SQL, NoSQL, fulltext и прочих, и одной системой кеширования.

Рекомендуется осуществлять разбиение пакетов и функций Go по поведенческому признаку из подходящей предметной области.

Основная задача масштабирования монолитного решения при возрастании нагрузки — такая оптимизация компонентов системы, которая повышает производительность решения в узких местах. Это может быть как повышенный расход памяти самим решением, так и различные «тормоза», возникающие в разных местах системы.

Например, в какой-то момент монолит начинает «тормозить» по причине того, что данных в базе накопилось много, и транзакции замедлились. Тогда:

- либо производится рефакторинг, меняется модель данных или запросы к СУБД, оптимизируются транзакционные блокировки;

- либо оптимизируется работа с базой путём создания промежуточных кеширующих механизмов — в том числе индексов;
- либо оптимизируется сама СУБД посредством её настроек или переноса на другой движок;
- либо делается масштабирование СУБД (вертикальное или горизонтальное).

Иногда оптимизация монолитного решения из-за сильной внутренней связанности приводит к масштабному переписыванию большей части его кода.

Важно! В монолитном решении на первый план выходят принципы SOLID, ACID и DRY, а в части хранения данных мы стремимся к нормальной форме данных. Целостность данных в любой момент, дедупликация данных и кода, поддерживаемость всего кода минимальным числом разработчиков — первоочередные приоритеты при разработке монолитного решения.

Целостность данных во время одной бизнес-транзакции считается обязательным требованием к монолитной системе. Если его не соблюдать, то когда-нибудь возникнет ситуация неконсистентности, несогласованности сильно связанных нормализованных данных, в том числе и в других участках системы.

Значительную часть проблем в монолитном решении создают различного рода блокировки, возникающие при общем доступе к нормализованным (дедуплицированным) данным. Для их решения применяются неблокирующие транзакции с оптимистическими блокировками на уровне бизнес-логики или СУБД.

Оптимистическая блокировка осуществляется в момент перезаписи какой-либо сущности в СУБД таким образом, что сравнивается версия данных, уже находящаяся в БД с той версией, которую мы изменяли на уровне бизнес-логики. Если эти версии не совпали — значит, кто-то другой успел «вклиниться» и сделать запись раньше нас. Теперь надо повторить чтение из базы и бизнес-логику на обновлённых данных прежде, чем мы что-то запишем в СУБД.

Ещё одна проблема монолитных решений — сложность сочетания в едином способе работы с данными паттернов доступа к данным OLAP и OLTP. Как правило, монолитные решения делятся на OLTP-транзакционные решения и аналитические OLAP-решения.

Архитектура микросервисных решений

Причины и путь от монолитных решений к микросервисным, по которым проходят различные команды разработки, мы рассмотрели ранее. Сейчас поговорим о том, когда и как выстраивать систему микросервисов.

Первый шаг, который делают многие, попытка разделить монолит на функциональные области или области из модели данных. Так получается распределённый монолит, который всё ещё обладает отрицательными качествами монолита.

Качественный переход на уровень микросервисной архитектуры происходит лишь при переходе на Event Sourcing — обмен событийной информацией между сервисами. Но и это ещё не всё. Самое большое преимущество микросервисов — разделение команд. В монолитном приложении очень сложно поддерживать продуктивность разработчиков, поскольку их количество увеличивается, а унаследованный код накапливается. Разделение сервисов и предоставление каждой команде разработчиков контроля над собственными кодовыми базами позволяет им разрабатывать собственные продукты в своём темпе.

Если у вас всего одна команда разработчиков, микросервисы будут менее привлекательными. Тем не менее всё же есть некоторые преимущества:

- возможность изолированного рефакторинга частей кодовой базы, включая, возможно, переписывание их на разных языках;
- возможность индивидуально настраивать масштаб времени по выполнению различных частей кодовой базы.

Принцип микросервисной архитектуры — разделение сервисов на независимые бизнес-контексты с API, доступным внешнему миру. Каждый отдельный бизнес-контекст инкапсулирует все атомарные операции, из чего следует, что распределённые транзакции между сервисами свидетельствуют о неправильной микросервисной архитектуре решения, и сейчас сервисы составляют распределённый монолит.

Разработчикам намного проще реализовывать атомарные последовательно выполняемые операции, их код получается линейным и предсказуемым. Но часто атомарность тех или иных операций не требуется.

В большинстве задач по изменению данных внутри бизнес-логики можно определить некоторый лаг по времени, с которым те или иные данные синхронизируются по цепочке обновлений. Такая возможность позволяет формировать цепочки событий и несколько вызываемых этими событиями операций, распределённых во времени, вместо одной линейной атомарной операции. Но именно такие свойства нарушают линейность кода и создают значительные проблемы для разработчиков, повышая сложность и трудоёмкость задач разработки в рамках такой архитектуры.

Теорема CAP (теорема Брюера) — эвристическое утверждение, что в любой реализации распределённых вычислений обеспечивается не более двух из трёх следующих свойств:

1. Согласованность данных (англ. consistency) — во всех вычислительных узлах в один момент данные не противоречат друг другу.
2. Доступность (англ. availability) — любой запрос к распределённой системе завершается корректным откликом.

3. Устойчивость к разделению (англ. partition tolerance) — расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой секции.

С точки зрения теоремы CAP, распределённые системы в зависимости от пары практически поддерживаемых свойств из трёх возможных распадаются на три класса — CA, CP, AP.

В системе класса CA во всех узлах данные согласовываются, и обеспечивается доступность. Она (система) жертвует устойчивостью к распаду на секции. Такие системы возможны на основе технологического программного обеспечения, поддерживающего транзакционность в смысле ACID. В качестве примера таких систем приведём решения на основе кластерных систем управления базами данных или распределённую службу каталогов LDAP. К ним относятся монолитные системы, свойства которых мы рассмотрели ранее.

Система класса CP в каждый момент обеспечивает целостный результат и способна работать в условиях распада, но достигает этого в ущерб доступности: не выдавать отклик на запрос. Устойчивость к распаду на секции требует обеспечения дублирования изменений во всех узлах системы. В связи с этим отмечается практическая целесообразность использования распределённых пессимистических блокировок для сохранения целостности. Такими свойствами обладают распределённые монолитные решения и СУБД в режиме репликации.

В системе класса AP целостность не гарантируется, но выполняются условия доступности и устойчивости к распаду на секции. Хотя системы такого рода известны задолго до формулировки принципа CAP, например, распределённые веб-кеши или DNS, рост популярности решений с этим набором свойств связывается именно с распространением теоремы CAP.

Так, большинство NoSQL-систем принципиально не гарантируют целостности данных и ссылаются на теорему CAP как на мотив такого ограничения. Задачей при построении AP-систем становится обеспечение некоторого практически целесообразного уровня целостности данных. В этом смысле AP-системы воспринимаются как «целостные в конечном счёте» (англ. eventually consistent) или как «слабоцелостные» (англ. weak consistent). Теми же свойствами обладают и микросервисы с выделенными бизнес-контекстами.

Создание и вертикальное масштабирование монолитного решения требует относительно низкой компетенции от разработчиков. Как правило, так и получается, легаси-код монолитного решения часто выглядит не совсем хорошим в силу того, что компетенция разработчиков на этом этапе была недостаточной. В какой-то момент возникнет необходимость перейти на микросервисную архитектуру.

Переход от монолитного решения к микросервисам потребует большого и качественного скачка компетенций у разработчиков, увеличения численности и количества команд разработчиков, а также существенных усилий от них. Это важно учитывать при планировании развития своего продукта.

Основные шаблоны по взаимодействию бизнес-логики с хранилищем данных

Основной процесс (бизнес-логика) каждого отдельного сервиса в составе распределённого монолита или микросервисов, если смотреть в целом, занимается сбором некоторых данных из внешнего мира, их трансформацией и дальнейшим размещением во внешнем мире.

Архитектура каждого отдельного сервиса обычно содержит такие слои:

1. Слой внешних API. Возможен в нескольких вариантах: REST, RPC, gRPC, openAPI, WSDL, отслеживание изменений в файле, СУБД и т. д.
2. Слой аутентификации и авторизации запросов, приходящих из API.
3. Основной инструментарий системы, выполняющий обработку полученных через API данных и возвращающий, или передающий в другие сервисы, через API результаты этой обработки.
4. Слой хранения данных в оперативной памяти (кеши).
5. Слой хранения данных в долгосрочном (persistent) хранилище.

Практически в любом сервисе встречаются все слои или часть этих слоёв.

Начинающие разработчики во время набора опыта сталкиваются с необходимостью наилучшим образом структурировать свой код, разделять его на пакеты, организуя внутренние соглашения об использовании между ними. Требуется декомпозиция программного кода по некоторым принципам, которая позволяет:

- обеспечивать хорошую поддерживаемость: легко читать код, понимать, как он работает, быстро находить и исправлять проблемы;
- с минимальными сложностями (максимально быстро) вносить изменения в код.

В Go единственный инструмент изоляции инструментария и структурирования кода — пакеты, а в них — экспортируемые сущности и функции. Поэтому архитектура будет сильно зависеть от того, по каким принципам пройдёт линия разграничения между пакетами. На это влияет выбранный архитектурный шаблон и несколько принципов, которые отслеживаются на примере кода стандартных библиотек Go:

1. Пакет состоит из нескольких файлов. Один из них имеет имя, совпадающее с именем пакета. В этом файле хранится внешнее API пакета и внешняя документация.
2. В каждом отдельном файле пакета есть какая-либо изолированная законченная функциональность, например, относящаяся к конкретному типу или к определённому поведению для этого типа.

3. Пакеты, которые недоступны за пределами репозитория сервиса, размещаются в папке internal в корне репозитория.

4. Есть часто меняемые пакеты — нестабильные, и редко изменяемые пакеты — стабильные. Стабильность определяется предполагаемым количеством изменений в коде за единицу времени. Например, пакеты стандартной библиотеки максимально стабильны, и от них зависит весь инструментарий в ваших пакетах.

5. Зависимости между пакетами распространяются от менее стабильных пакетов к более стабильным. Менее стабильные пакеты импортируют много других более стабильных пакетов, а не наоборот.

6. Внешние зависимости из менее стабильных пакетов, то есть из разнообразных реализаций, разных типов или даже ещё не существующих пакетов, которые принимаются на вход функций и методов более стабильного пакета, оформляются в виде интерфейсных типов. Пример — пакет sort, который принимает на вход слайс из элементов вашего типа и функцию сравнения элементов между собой (менее стабильных), чтобы обеспечить их единообразную сортировку алгоритмом quick sort (самый стабильный инструментарий).

7. Важно, чтобы названия пакетов и файлов отражали поведенческие особенности инструментария. Не надо использовать классифицирующие названия типа db, api, common и прочими — они ничего не говорят о поведении и похожи на абстрактные классы.

Применяются эти принципы к уже конкретно выбранному архитектурному шаблону. Далее мы рассмотрим шаблоны Active Record, Data Mapper, Unit of Work, PAC, MVC/MVP, MVVM, гексагональной и чистой архитектуры. Узнаем, насколько в Go удобно использовать те или иные архитектурные шаблоны.

Вначале поговорим о сущностях в сервисе, хранимых данных и миграциях. Затем перейдём к слоистой архитектуре сервиса.

Миграции

Мы уже говорили выше, что каждый сервис состоит из компонент, отвечающих за его будущее поведение — API и бизнес-логика — и прошлое состояние (системы хранения данных). Бизнес-логика постоянно обновляется во время эволюционного развития сервиса. Конфликт между прошлым состоянием хранимых данных и будущим, которое ожидает обновлённая бизнес-логика, разрешается в миграциях данных в системах их хранения.

Миграции состоят из двух частей:

1. Миграция схемы данных для тех хранилищ, состояние данных в которых описываются схемой, например, Postgres, MySQL и т. д.

2. Миграция самих данных — необязательно в связи с изменением схемы данных.

Актуализация схемы хранения данных обязательно производится в автоматизированном виде. Создание схем данных для хранилищ на их языке DDL в виде отдельных скриптов миграции — плохой шаблон из-за дупликации кода и лишних сложностей по поддержанию разных представлений схемы данных как в коде сервиса, так и в коде скриптов. Современные платформы автоматизации не используют скриптовые миграции для модификации схем данных, в них это происходит автоматически.

И в самом деле, если у вас есть видение целевой схемы данных, а также текущая схема данных, то ничто не мешает запрограммировать универсальный алгоритм преобразования существующей схемы в целевую, вне зависимости от их конкретного содержимого сейчас.

При использовании автоматических миграций схем данных, а также для инициации миграции самих данных используется разница в схемах или реестры отметок о прохождении тех или иных миграций данных в конкретном хранилище. В этом тоже не возникает никаких сложностей или неоднозначностей, поскольку обычно миграция данных выполняется при добавлении элементов в схему данных, при кардинальной (несовместимой) смене типа хранимых данных в элементе или при удалении элемента из схемы. В остальных случаях изменения данных относятся к обычной бизнес-логике.

Active Record

Один из самых простых шаблонов, который успешно применяется в небольших сервисах.

Все архитектурные слои, то есть внешнее API, слой DAL и другие, строятся на базе одного (единственного) определения типа. В нём через теги определяется представление одной и той же сущности в подходящих слоях, и дополнительно к этому реализовываются различные интерфейсные типы. Посредством выделения одной сущности в отдельный пакет и разработки всех методов в этом же пакете можно подготовить полноценно абстрагированный тип для использования с различными внешними библиотеками API и СУБД, а также различными пакетами, реализующими бизнес-логику над такими сущностями.

Active Record описал Martin Fowler в своей книге Patterns of Enterprise Application Architecture. В Active Record объекты содержат и персистентные данные, и поведение, которое работает с этими данными. Active Record исходит из мнения, что обеспечение логики доступа к данным, как части объекта покажет пользователям этого объекта то, как читать и писать в базу данных.

Object Relational Mapping (объектно-реляционное отображение), обычно упоминающееся как аббревиатура ORM. Это техника, соединяющая сложные объекты приложения с таблицами в системе управления реляционными базами данных. Посредством ORM свойства и взаимоотношения этих объектов приложения с лёгкостью сохраняются и получаются из базы данных без прямого написания выражений SQL, а также с меньшим суммарным кодом для доступа в базу данных.

Active Record предоставляет несколько механизмов, наиболее важными из которых считаются способности:

- для представления моделей и их данных;
- представления связей между этими моделями;
- представления иерархий наследования через связанные модели;
- валидации моделей до того, как они станут персистентными в базе данных;
- выполнения операций с базой данных в объектно-ориентированном стиле.

В этом шаблоне каждая сущность, а также коллекция таких сущностей, обладает конкретным поведением, в том числе для целей внешнего взаимодействия:

- преобразования данных API <-> данные сущности (коллекции);
- преобразования данных СУБД <-> данные сущности (коллекции);
- выполнения CRUDL-операций — create, read, update, delete, list with filters — в конкретном хранилище данных;
- другие операции, связанные с сохранением данных и получением данных из системы хранения;
- подписки на события, происходящие в жизненном цикле сущности или коллекции;
- миграции данных.

Для выполнения этих целей часто используются библиотеки ORM, позволяющие по описанию структуры в коде, через теги полей или методы реализации требуемых интерфейсов, реализовывать преобразования данных между различными форматами и производить стандартные операции с хранилищем данных.

По такому шаблону в Go работают библиотеки ent, gorm и xorm и другие. Все они используют определение типа Go, чтобы преобразовывать значение в подходящий вид и использовать в операциях с СУБД.

Шаблон Active Record неплох. Многие разработчики его легко воспринимают, ведь он позволяет скрывать рутинное взаимодействие с СУБД при выполнении рядовых операций CRUDL, что сокращает дублирование кода по формированию индивидуальных запросов на языке СУБД. Автоматические миграции в этом шаблоне тоже хорошо реализуются.

Часто применяемые библиотеки: encoding/json, go-chi/render, govalidator, sqlx, ent, gorm, xorm и другие.

Data Mapper

Этот шаблон развивает и улучшает предыдущий паттерн Active Record и не требует использования ORM-инструментария, но и не запрещает его использование.

Преобразователь данных (Data Mapper) — это паттерн, который выступает посредником для двунаправленной передачи данных между:

- постоянным хранилищем данных — часто, реляционной базы данных;
- и представлением данных в памяти — слой домена, то что уже загружено и используется для логической обработки.

Цели паттерна:

1. Держать представление данных в памяти.
2. Служить постоянным хранилищем данных независимыми друг от друга и от самого преобразователя данных.

Слой состоит из одного или более мапперов или объектов доступа к данным, отвечающих за передачу данных. Реализации мапперов различаются по назначению. Общие мапперы обрабатывают всевозможные типы сущностей доменов, а выделенные — обрабатывают один или несколько конкретных типов.

Ключевым моментом этого паттерна, в отличие от Активной Записи (Active Records), считается то, что модель данных следует принципу единой обязанности SOLID.

Применение Data Mapper в Go обычно приводит к созданию трёх сущностей для одного объекта предметной области:

1. Тип, соответствующий объекту предметной области — обычно это структура.
2. Mapper — структура с методами, которые выполняют некоторые операции над объектами предметной области, то есть над коллекциями типа из пункта 1 или над объектами этих типов.
3. Объект, отвечающий за соединение с системой хранения данных, через который выполняются запросы к системе хранения.

Пример Data Mapper без использования ORM

Этот пример показывает самый широко используемый способ реализации паттерна Data Mapper в сервисах на Go.

Создадим таблицу СУБД, содержащую информацию о пользователях:

```
CREATE DATABASE data_mapper_post;
CREATE EXTENSION pgcrypto;

CREATE TABLE users (
  id uuid NOT NULL DEFAULT gen_random_uuid() PRIMARY KEY,
  password text NOT NULL,
  email text NOT NULL,
  name text NOT NULL,
  created_at timestamp without time zone
    NOT NULL DEFAULT (now() at time zone 'utc')
);
```

В качестве primary key используется автоматически генерируемый uuid, а created_at представляет собой отметку времени, когда появилась запись о юзере.

Создадим тип User, который представляет собой модель пользователя, с точки зрения предметной области. У него есть несколько полей, содержащих логин, электронную почту и прочее:

```
type User struct {
      ID        string
      Name      string
      Email     string
      CreatedAt time.Time
}
```

Создадим тип для реализации Mapper-объекта:

```
type UserMapper struct {
      DB *sql.DB
}
```

Маппер содержит инъекцию (dependency injection) объекта, ответственного за соединение с СУБД и выполнение запросов SQL.

Чтобы проиллюстрировать работу с пользователями, поработаем с тремя основными функциями предметной области:

1. Создать пользователя.
2. Найти пользователя по идентификатору.
3. Найти пользователя по электронной почте и паролю.

А чтобы избежать лишнего дублирования кода, создадим вспомогательные методы, позволяющие получить имена колонок полей и значения полей для подстановки в запросы к СУБД:

```
func (um *UserMapper) Columns() string {
	return "id, name, email, created_at"
}

func (um *UserMapper) Fields(u *User) []interface{} {
	return []interface{}{&u.ID, &u.Name, &u.Email, &u.CreatedAt}
}
```

Обратите внимание, что здесь в учебных целях заполнение результатов этих функций происходит явно. В реальном продуктивном коде предпочтительнее заполнять результаты функций посредством рефлексии, чтобы при изменении состава полей модели User не потребовалось исправлять и эти две функции.

Создадим все три метода предметной области:

```
// CreateUser добавляет пользователя в СУБД и возвращает заполненную модель
User для него
func (um *UserMapper) CreateUser(name, email, pw string) (*User, error) {
	u := &User{}
	err := um.DB.QueryRow(fmt.Sprintf(`
		INSERT INTO users(name, email, password)
		VALUES($1, $2, crypt($3, gen_salt('bf', 8)))
		RETURNING %s
	`, um.Columns()), name, email, pw).Scan(um.Fields(u)...)
	if err != nil {
		return nil, err
	}
	return u, nil
}

// FindUserByEmailPassword ищет пользователя по email и password
func (um *UserMapper) FindUserByEmailPassword(email, pw string) (*User, error) {
	return um.SelectUser(
		"lower(email) = lower($1) AND password = crypt($2, password)",
		email,
		w,
	)
}

// FindUser ищет пользователя по его id и если не находит, то возвращает nil
func (um *UserMapper) FindUser(id string) (*User, error) {
	return um.SelectUser("id = $1", id)
}
```

Для дедупликации кода мы также реализуем общую функцию, которая используется при поиске по email с password и по id:

```
func (um *UserMapper) SelectUser(c string, v ...interface{}) (*User, error) {
	u := &User{}
	err := um.DB.QueryRow(
		fmt.Sprintf(`SELECT %s FROM users WHERE %s`, um.Columns(), c),
		v...,
	).Scan(um.Fields(u)...)
	if err != nil {
		if err == sql.ErrNoRows {
			return nil, nil
		}
		return nil, err
	}
	return u, nil
}
```

Как видите, каждый тип занимается своим делом. User отвечает за состав данных, UserMapper — за формирование запросов к СУБД для этого типа, а коннект к СУБД — за прямое соединение с СУБД и выполнение запросов SQL.

Шаблон Data Mapper применяется в Go часто. У него нет серьёзных недостатков, с точки зрения принципов SOLID, и есть возможность быстро и независимо осуществлять замену одного из компонентов — модели, маппера или СУБД.

Ещё одним несомненным преимуществом считается то, что в качестве хранилища данных для маппера можно указать внутренний кеш или внешний кеширующий сервис, а не только СУБД. Таким образом, реализуется рекомендуемый М. Фаулером паттерн «коллекция объектов».

Важно! *Главная задача «коллекции объектов» — ведение реестра уникальных идентификаторов какой-либо сущности, и лишь затем — выполнение функции ускорения доступа путём кеширования.*

Data Mapper позволяет создавать тесты с подменой реальной СУБД на мок-объект.

В общем случае Data Mapper успешно занимается в том числе автоматической миграцией данных при апгрейде своей версии, а также актуализацией DDL-схемы в системе хранения данных, если там используется схема.

Unit of work

Этот шаблон напоминает о том, что некоторые последовательности операций шаблона Data Mapper всё-таки надо выполнять атомарно, в одной транзакции. И здесь возникают сложности.

Мы можем поддерживать целостность данных в СУБД при каждом изменении нашей объектной модели, не объединяя их в одну транзакцию, но это приведёт к множеству очень маленьких обращений к базе данных. Это в конечном счёте превратится в очень медленный процесс. Часто требуется как-то явно блокировать прочитанные нами объекты, чтобы избежать «грязного чтения».

Если открыть транзакцию для всего потока операций, но это может надолго блокировать ресурсы СУБД.

Единица работы — Unit of work (UOW) — используется для предотвращения частого обращения к базе данных при отправке множества небольших фрагментов информации. В этом паттерне идёт накопление множества мелких изменений для формирования единственного «патча» в системе хранения данных.

Паттерн Unit of Work, как правило, не считается полностью самостоятельным. Обычно он тесно связан с паттерном Identity Map, задача которого — сохранение карты созданных объектов, взятых из хранилища, чтобы гарантировать, что одна единица информации из хранилища представлена ровно одним экземпляром объекта данных в приложении. Это позволяет избежать конфликтов изменений, так как не допускает ситуации, когда два объекта, представляющие один и тот же элемент данных в хранилище, изменены по-разному.

Информация из Identity Map используется в методе commit() паттерна Unit of Work для вычисления разницы между исходными данными и накопленными изменениями.

Поскольку для вычисления разницы и, соответственно, определения, что и каким образом должно быть изменено в хранилище, надо знать, какие данные и как именно хранятся в объектах. Как правило, требуется реализация паттерна Metadata Mapping, описывающего связь между содержимым хранилища — между таблицами и столбцами базы данных — и классами, свойствами объектов (часть Data Mapper).

Если данные в хранилище не считаются независимыми, например, то потребуется реализация нескольких паттернов, отвечающих за сохранение информации о связях между данными. Это паттерны раздела Object-Relational Structural Patterns в каталоге паттернов.

Естественно, UOW не может обойтись без Data Mapper для преобразования самих данных.

Одна единица работы UOW включает в себя множество мапперов данных, а один маппер данных — множество единиц работы. Они ортогональны друг другу.

Обычно Unit of work встраивается как dependency injection в Data Mapper при его создании, по аналогии с инъекцией объекта СУБД. Тогда мы точно можем определить, в рамках какой бизнес-транзакции выполняется та или иная операция Data Mapper.

Один экземпляр UOW представляет одну активную бизнес-транзакцию. Установленный в Data Mapper конкретный экземпляр UOW означает, что вы участвуете в бизнес-транзакции, действия которой ещё не зафиксированы. Data Mapper и не должен знать, зафиксирована транзакция или ещё нет. Управление временем жизни UOW происходит на более высоком уровне. Например, для веб-приложения это обычно один входящий запрос на API.

Unit of work необязательно должен создавать одну транзакцию к СУБД. Речь идёт о создании бизнес-транзакции, которая реализуется:

- через трекинг изменений объектов в Data Mapper и формирование одной результирующей небольшой транзакции СУБД с гораздо меньшим числом запросов, чем это было бы при использовании только паттерна Data Mapper;
- через оптимистические блокировки посредством версий объектов внутри бизнес-транзакции.

Объект UoW, реализующий этот паттерн, отвечает за накопление информации о том, какие объекты входят в транзакцию и каковы их изменения относительно исходных значений в хранилище. Основная работа производится в методе commit(), который отвечает за вычисление изменений в сохранённых UOW-объектах и синхронизацию этих изменений с хранилищем (базой данных).

Пример Unit of work с общей транзакцией sql

Создадим пакет uow и тип UnitOfWork, который будет выполнять роль простейшего unit of work, объединяющего всю дальнейшую работу UserMapper в одну транзакцию СУБД:

```
package uow

import (
    "context"
    "database/sql"
    "fmt"
    "log"
    "runtime/debug"
    "time"
)

type UnitOfWork struct {
    db *sql.DB
    tx *sql.Tx
}

func (uow UnitOfWork) WithTx(ctx context.Context, f func(uow UnitOfWork) error)
(err error) {
    // если вызов — внутри UnitOfWork с существующей транзакцией, то выполняем
внутри неё
    if uow.tx != nil {
        if e := f(uow); e != nil {
            return e
        }
        return nil
    }

    // иначе — создаём и выполняем транзакцию
    return uow.WithBeginTx(ctx, f)
}
```

```
func (uow UnitOfWork) WithBeginTx(ctx context.Context, f func(uow UnitOfWork)
error) (err error) {
   var newTx *sql.Tx
   // стартуем новую транзакцию
   if tx, e := uow.db.BeginTx(ctx, nil); e != nil {
       return e
   } else {
       newTx = tx
   }

   nuow := uow
   nuow.tx = newTx

   // во время транзакции может произойти что угодно — обработаем это
   // в случае ошибки или паники — откатим её
   commit := false
   defer func() {
       if r := recover(); r != nil || !commit {
           if r != nil {
               log.Printf("!!! TRANSACTION PANIC !!! : %s\n%s", r,
string(debug.Stack()))
           }
           if e := newTx.Rollback(); e != nil {
               err = e
           } else if r != nil {
               err = fmt.Errorf("transaction panic: %s", r)
           }
       } else if commit {
           if e := newTx.Commit(); e != nil {
               err = e
           }
       }
   }()

   if e := f(nuow); e != nil {
       return e
   }

   commit = true
   return nil
}

func (uow UnitOfWork) Tx() *sql.Tx {
   return uow.tx
}
```

Немного исправим ранее созданный нами UserMapper и покажем на примере одной из функций, как она изменится при работе с unit of work:

```
type UserMapper struct {
   uow *UnitOfWork
}

// .....

func (um *UserMapper) SelectUser(ctx context.Context, c string, v
...interface{}) (*User, error) {
   u := &User{}
   if um.uow == nil {
       return nil, fmt.Errorf("unit of work is empty")
   }

   // такой вызов либо использует уже запущенную ранее в uow транзакцию, либо
запустит новую
   if err := um.uow.WithTx(ctx, func(uowtx UnitOfWork) error {
       // внутри этой функции нам ничто не мешает получить транзакцию WithTx
ещё раз
       // это полезно, если дальше вызываются какие-то универсальные функции
наподобие этой
       return uowtx.Tx().QueryRowContext(ctx,
           fmt.Sprintf(`SELECT %s FROM users WHERE %s`, um.Columns(), c),
           v...,
       ).Scan(um.Fields(u)...)
   }); err != nil {
       if err == sql.ErrNoRows {
           return nil, nil
       }
       return nil, err
   }

   return u, nil
}
```

Теперь любые вызовы SelectUser станут выполняться в контексте одной и той же транзакции unit of work, если она запущена ранее.

Шлюз таблицы БД и шлюз записи БД

Все запросы SQL рекомендуется изолировать от бизнес-логики путём выноса всех запросов SQL в соответствующие отдельные типы данных, чтобы не происходило смешивание кода Go и запросов SQL. Для этих целей М. Фаулер рекомендует использовать структуры, описывающие таблицы и отдельные записи, методы которых формируют напрямую запросы SQL. Слой работы с СУБД, и, в частности, Data Mapper, вызывает эти функции у соответствующих шлюзов.

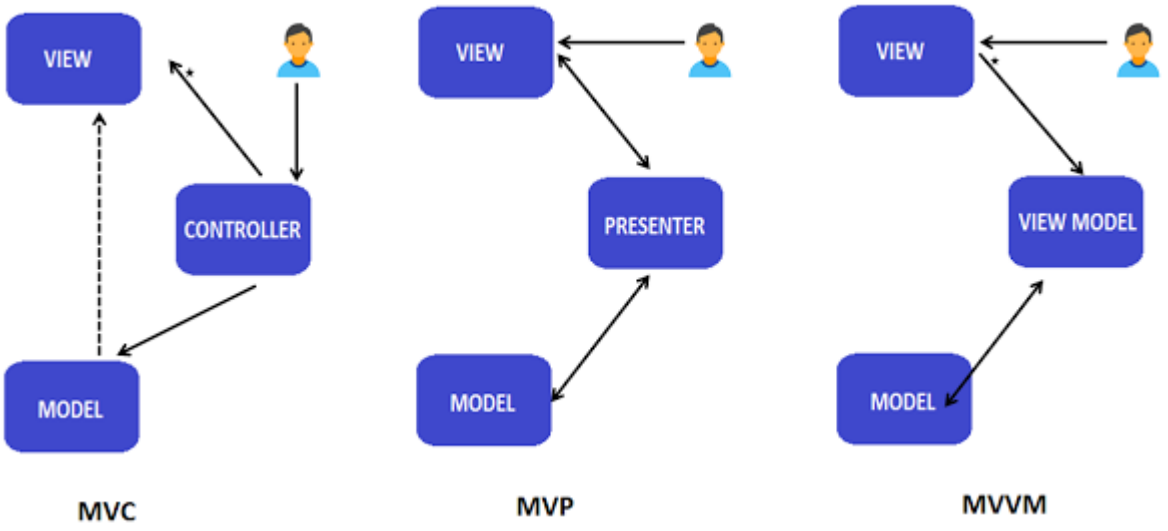
В примере выше метод (um *UserMapper) SelectUser надо перенести в новую структуру UserTableGateway:

```
package tgw

import (
    "context"
    "database/sql"
    "fmt"
)

type UserTableGateway struct{}

func (um UserTableGateway) tableName() string {
    return "users"
}

func (um UserTableGateway) Columns() string {
    return "id, name, email, created_at"
}

func (um UserTableGateway) Fields(u *User) []interface{} {
    return []interface{}{&u.ID, &u.Name, &u.Email, &u.CreatedAt}
}

func (um UserTableGateway) SelectUser(ctx context.Context, tx *sql.Tx, where
string, dest *User, v ...interface{}) error {
    return tx.QueryRowContext(ctx,
        fmt.Sprintf(`SELECT %s FROM %s WHERE %s`, um.Columns(), um.tableName(),
where),
        v...,
    ).Scan(um.Fields(dest)...)
}
```

MVC, MVP и MVVM

Это три самых популярных архитектурных дизайна систем в целом. Если в предыдущих разделах мы говорили про шаблоны архитектуры конкретных сущностей, а также операций над ними, то дальше речь пойдёт о структуре слоёв архитектуры в целом.

Слоистая архитектура в целом улучшает и исправляет недостатки простейших архитектурных паттернов: Active Record, Data Mapper, UoW.

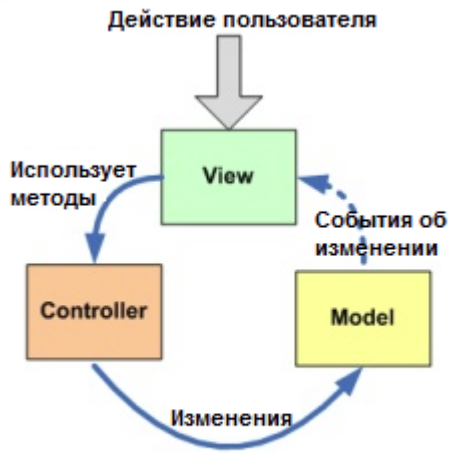


MVC MVP MVVM

Каждая буква в этих шаблонах означает определённый слой:

MVC

MVC — это фундаментальный паттерн, который нашёл применение во многих технологиях, дал развитие новым технологиям и каждый день облегчает жизнь разработчикам.



Model-View-Controller

Model — модель, состоит из модели данных и бизнес-логики работы с данными в системе.

Важно, чтобы модель не зависела от остальных частей продукта, а модельный слой ничего не знал об элементах дизайна и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая самой модели.

Модель обладает следующими признаками:

1. Модель — это бизнес-логика приложения.

2. Модель обладает знаниями о себе и не знает о контроллерах и представлениях.
3. Для некоторых проектов модель — просто слой данных: DAO, база данных, XML-файл.
4. Для других проектов модель — это менеджер базы данных, набор объектов или просто логика приложения.

View — представление, инструментарий визуализации данных, получаемых из модели.

В обязанности представления View входит отображение данных, полученных от модели. Однако, представление не влияет на модель напрямую. Можно говорить, что представление обладает доступом к данным «только на чтение».

Представление обладает следующими признаками:

1. В представлении реализуется отображение данных, которые получаются от модели любым способом.
2. В некоторых случаях представление имеет код, который реализует некоторую бизнес-логику.

Примеры представления: HTML-страница, json-объект.

Controller — медиатор между слоями View и Model, занимается трансформацией данных и запросов, полученных от пользователя, в управляющие воздействия на модель.

Признаки контроллера:

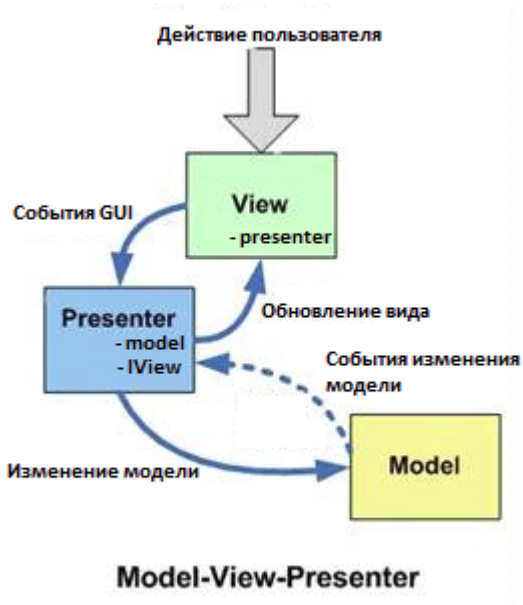
1. Контроллер определяет, какие представления отобразятся в конкретный момент.
2. События представления влияют только на контроллер. Контроллер влияет на модель и определяет другое представление.
3. Несколько представлений возможны только для одного контроллера.

Основная идея этого паттерна заключается в том, что контроллер и представление зависят от модели, а модель не зависит от этих двух компонент.

В Go используется нечасто, так как обычно view не имеет доступа к событиям и данным модели в обход контроллера. Взаимодействие с View производится в большинстве случаев по http, который не позволяет самостоятельно инициировать коммуникацию с view со стороны модели. Обычно MVC-решения на Go содержат отдельный контроллер CRUDL-операций, выполняющий роль склеивающего слоя между View и Model, а основной контроллер (отдельный), в свою очередь, содержит бизнес-логику. View в этой архитектуре содержит часть бизнес-логики, занимающейся интерпретацией данных модели, полученных через CRUDL-контроллер.

Часто используемая библиотека — graphQL.

MVP



Похож на MVC, но вместо контроллера используется Presenter, который получает пользовательские запросы и данные от View и передаёт их в модель. MVP получает результат и возвращает его в View.

Признаки презентера:

1. Двухсторонняя коммуникация с представлением.
2. Представление (View) напрямую взаимодействует с презентером, вызывая соответствующие функции или события экземпляра презентера (через API).
3. Презентер взаимодействует с View путём использования специального интерфейса, реализованного представлением в соответствии с соглашением API.

Интерфейс View определяет набор функций и событий, необходимых для взаимодействия с пользователем. В случае сервисов на Go он описывается на уровне соглашения API с презентором.

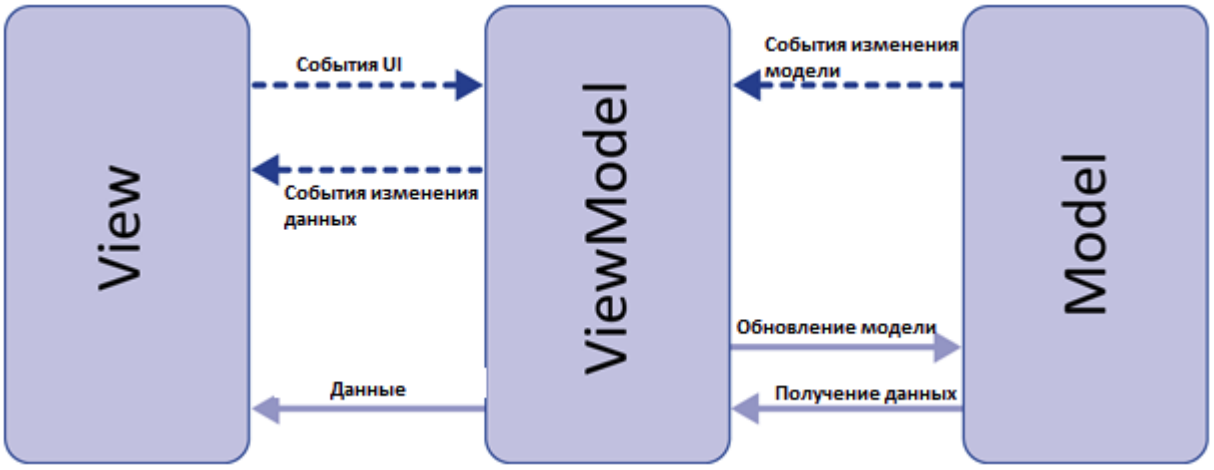
Этот шаблон часто используется в Go. Классический пример Presenter — http.Handler, возвращающий html или json, причём json может содержать и денормализованные данные.

В практическом плане этот паттерн говорит, что в слое Presenter может возникать и поддерживаться отдельная структура данных, в том числе не совпадающая со структурой данных, используемой в слое Model. В Go эти структуры находятся в разных пакетах. Реализация server-side rendering возможна также через html/templates.

В слое Model для работы с хранением данных используется паттерн Active Record или Data Mapper, и, конечно же, Unit of work.

Прямая работа с хранилищем данных внутри контроллера считается плохим архитектурным паттерном и приводит к проблеме FSTC — Fat Stupid Ugly Controllers.

MVVM



Такой подход позволяет связывать элементы представления со свойствами и событиями View-модели. Можно утверждать, что каждый слой этого паттерна не знает о существовании другого слоя.

Признаки View-модели:

1. Двухсторонняя коммуникация с представлением.
2. View-модель — это абстракция представления. Обычно означает, что свойства представления совпадают со свойствами View-модели.
3. View-модель не имеет ссылки на интерфейс представления (IView). Изменение состояния View-модели автоматически изменяет представление и наоборот, поскольку используется механизм связывания данных (Bindings).
4. Один экземпляр View-модели связан с одним отображением.

Используется, когда связывание данных возможно без ввода специальных интерфейсов представления, то есть не надо реализовывать IView. Яркий пример — VueJS.

В Go реализовать ViewModel очень сложно, поскольку невозможно языковыми средствами обеспечить связывание полей структур по аналогии, как это сделано в javascript. Поэтому такой шаблон реализуется на стороне фронтенда в части View и ViewModel, а значит, и часть бизнес-логики уходит туда. Бэкенду достаётся только слой Model. Модель на бэкенде всегда возвращает json только определённого вида, включая связи между сущностями модели.

Такой шаблон часто используется в сочетании с SPA (single page applications) на фронтенде.

Гексагональная архитектура

Гексагональная архитектура бэкенда на Go применяется при реализации компонентов Model или Presenter+Model из рассмотренных выше архитектурных дизайнов.

Гексагональная архитектура решает такие задачи:

1. Позволяет взаимодействовать с приложением как пользователю, так и другим программам, автоматическим тестам, скриптам пакетной обработки.
2. Даёт разрабатывать и тестировать приложение без каких-либо дополнительных устройств или баз данных.
3. Позволяет сравнительно легко проводить независимое изменение каждого слоя этой архитектуры, заменять реализации слоёв на альтернативные (более лучшие) или тестовые.

В целом, такая архитектура хорошо тестируется, масштабируется и поддерживается.

Хотя архитектура и называется гексагональной, что указывает на фигуру с определённым количеством граней, главное — множество граней этой фигуры. Каждая грань представляет собой «порт» доступа к нашему приложению или же его связь с внешним миром.

Порт — это некий проводник входящих запросов (или данных) к нашему приложению. Например, через HTTP-порт (запросы браузера, API) приходят HTTP-запросы, которые конвертируются в команды для приложения. Похожим образом действуют различные менеджеры очередей или всё, что взаимодействует с приложением по протоколу пересылки сообщений, например, AMQP. Всё это лишь порты, через которые мы можем попросить приложение сделать какие-то действия. Эти грани составляют множество «входящих портов». Другие порты используются для доступа к данным со стороны приложения, например, порт базы данных.

К чему мы стремимся при использовании правильной архитектуры:

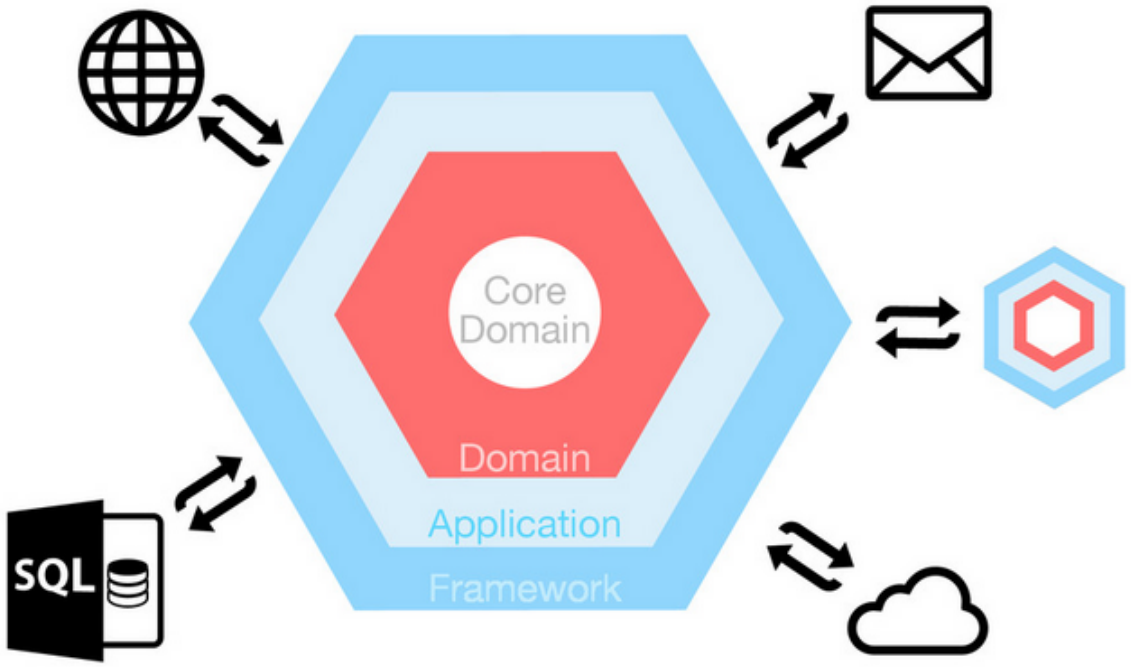
1. Изменения в одной части приложения почти не затрагивают других частей.
2. Добавление нового инструментария не требует каких-то масштабных изменений в коде.
3. Организация нового интерфейса для взаимодействия с приложением приводит к минимально возможным изменениям приложения.
4. Отладка включает меньше временных решений и хаков.
5. Тестирование происходит относительно легко.

Так как совершенного кода нет, слово «должно» стоит воспринимать, как то, к чему надо стремиться, но не более. Мы стараемся сделать наше приложение более простым в поддержке. Попытки же сделать его «идеальным» приведут к потерям времени и ментальной энергии.

Плохие архитектурные решения, принятые на ранних этапах разработки, в совокупности выльются нам большими проблемами. Незначительные, но плохие решения, принятые во время разработки, также приведут к проблемам. К счастью, обычно это не приводит к масштабным проблемам, к которым обычно приводят ошибки проектирования на ранних этапах конструирования.

Важно! *Хороший фундамент уменьшает скорость роста технического долга.*

Гексагональная архитектура — это слоёная архитектура, иногда называемая архитектурой портов и адаптеров. В рамках этой архитектуры есть концепция различных портов, которые используются (адаптируются) для использования с другими слоями.



Интерфейсы станут нашим главным средством для снижения связанности кода между слоями.

Задача кода — описывать в пределах слоёв и на их границах то, как должно происходить их взаимодействие. Так как слои действуют как порты и адаптеры для других слоёв, окружающих их, важно иметь правила взаимодействия между ними.

Слои взаимодействуют друг с другом, используя интерфейсы (порты) и реализации этих интерфейсов (адаптеры).

Вы можете найти в интернете различные изображения гексагональной архитектуры, на которых скрыты слои Application и Framework. Такие вариации считаются неполноценными, они не учитывают важные нюансы архитектуры, о которых мы поговорим далее.

Слой предметной области: бизнес-логики, Domain Layer

В самом центре нашего приложения — слой предметной области. Он также называется слоем бизнес-логики. Этот слой содержит в себе реализацию бизнес-логики предметной области и определяет, как внешние слои могут с ней взаимодействовать.

Бизнес-логика — это сердце нашего приложения. Описывается словом «устав» — правила, которым подчиняется ваш код.

Слой предметной области и бизнес-логика, реализованная в нём, определяют сущности, их поведение и различные ограничения вашего приложения. Это отличает ваше приложение от других и придаёт ему ценность.

Если ваше приложение содержит сложную бизнес-логику, различные варианты поведения, то у вас получится богатый слой предметной области. А если ваше приложение — небольшая надстройка над базой данных, так как многие приложения считаются таковыми, то этот слой будет «тоньше».

К бизнес-логике, как к ядру предметной области или к core domain, в слое предметной области часто примыкает дополнительная логика, например:

- события предметной области — domain events, события, которые выбрасываются в ключевых точках бизнес-логики;
- «сценарии использования» или use-cases — определение задач приложения.

Содержание слоя предметной области — тема для целой книги или серии книг, особенно если вы интересуетесь предметно-ориентированным проектированием (DDD). Эта методология детальнее описывает способы реализации приложения как можно ближе к терминам предметной области, а, стало быть, к бизнес-процессам, которые мы хотим перенести в код.

Каждый пакет из слоя предметной области может импортировать в основном только пакеты из стандартной библиотеки и пакеты, относящиеся к этому же слою (бизнес-логики).

Слой приложения (Application Layer)

Сразу за слоем предметной области «сидит» наш слой приложения. Этот слой занимается исключительно оркестрацией действий, производимых над сущностями из слоя предметной области. Он также считается адаптером запросов из слоя фреймворка и отделяет его от слоя предметной области.

Слой приложения отправляет на обработку события (domain events), которые произошли в слое предметной области.

Это внешний слой кода, составляющий наше приложение.

Снаружи слоя приложения есть ещё «слой фреймворка». Он содержит вспомогательный код нашего приложения, возможно, принимающий HTTP-запросы или отправляющий email. Но этот слой не само приложение.

Слой фреймворка (Framework Layer)

Наше приложение «закутано» в слой фреймворка, который также называется инфраструктурным слоем — infrastructure layer. Этот слой содержит код, который использует ваше приложение, но не считается частью приложения. Обычно такой слой представлен вашим фреймворком или стандартной библиотекой Go, но включающий в себя любые сторонние библиотеки и любой другой код. Задача этого слоя — выполнять различные рутинные, часто повторяемые задачи для удовлетворения потребностей приложения.

Слой уровня фреймворка также представляет собой адаптер HTTP-запросов к нашему приложению. Например, мы можем поручить приём HTTP-запросов, сбор входящих данных и маршрутизацию запроса или данных к соответствующему http.Handler.

Таким образом, этот слой отделяет наше приложение от всех внешних запросов, например, сделанных из браузера. Именно этот слой считается адаптером запросов приложения.

Важно!

Не стоит воспринимать определения слоёв как какое-то очень жёсткое правило. Если у вас возникнет вопрос «а что если мне понадобится использовать стороннюю библиотеку из слоя фреймворка в слое предметной области», то ничего страшного. Основной идеей служит разделение обязанностей: когда мы определяем интерфейс, то делаем именно это. Поэтому сменить функциональную составляющую можно и позже.

Взаимодействие слоёв в гексагональной архитектуре: границы

Теперь обсудим, как слои взаимодействуют друг с другом.

Как уже говорилось выше, каждый слой регламентирует то, каким образом другие слои взаимодействуют с ним. Если конкретнее, то все слои ответственны за определение того, как с ними будет взаимодействовать каждый следующий внешний слой.

Инструментом для этого нам послужат интерфейсы. **На границе каждого слоя мы найдём интерфейсы**. Эти интерфейсы считаются портами для следующих слоёв, где реализуются адаптеры.

Например, слой приложения содержит реализацию интерфейсов (адаптеры к портам), объявленных в слое предметной области. Этот слой будет содержать код для других вещей, специфичных для нашего приложения.

Пройдём через границы каждого слоя, чтобы разобраться, как они связаны друг с другом.

Слой предметной области (бизнес-логики)

На границе слоя предметной области мы найдём средства, предоставляемые им для общения внешнему слою (слою приложения).

Например, наш слой предметной области содержит команды (сценарии использования). Слой предметной области объявляет, каким образом он может быть использован, а уже слой приложения использует эти определения как составляющую для выполнения объявленного сценария использования.

Пример с регистрацией пользователя

На границе слоя предметной области мы разместим следующий интерфейс:

```
package register

type User struct{
  Name string
  Password string
  Email string
}

type UserProvider interface{
  ProvideNewUser(u User) error
}

func RegisterUser(name, email, pass string, up UserProvider) error{
  u := User{name, pass, email}
  // ....
  return up.ProvideNewUser(u)
}
```

В этом примере в слое бизнес-логики мы объявляем интерфейс UserProvider. Через этот интерфейс наша бизнес-логика вызовет уведомление слоя приложения о вновь созданном пользователе (up.ProvideNewUser(u)) по окончании его регистрации в своём методе RegisterUser. Слой приложения реализует эту функцию в каком-либо из своих типов и передаст его объект на вход функции RegisterUser в параметр up.

Слой приложения

Итак, в этом слое мы должны реализовать обработку события регистрации нового пользователя, которое получили из слоя бизнес-логики. Допустим, нам надо отправить сообщение по email самому пользователю с приветствием после регистрации. Всё, что для этого требуется, есть в слое уровня

фреймворка. Именно в слое фреймворка удобнее всего разместить реализацию сервисов по отправке уведомлений.

Для этого в слое приложения определим интерфейс:

```
type Notifyer interface{
  Notify(to, message string) error
}
```

Чтобы в main создавать объект слоя приложения, в его параметры надо передать фреймворк, реализующий метод Notify этого интерфейса. После этого на уровне слоя приложения мы сможем вызвать эту функцию в момент выполнения метода ProvideNewUser:

```
type App struct{
  nf Notifyer
}

func (a *App) ProvideNewUser(u register.User) error{
  // ....
  return a.nf.Notify(u.Email,"Welcome to our service, "+u.Name)
}
```

Слой приложения определяет, как он будет взаимодействовать со слоем фреймворка. Даже больше, слой приложения определяет, как будет использоваться слой фреймворка без выстраивания прямой зависимости от него. Интерфейсы (порты) и их реализации (адаптеры) позволяют нам с лёгкостью сменить один адаптер на другой. Таким образом, мы не привязываемся намертво к слою фреймворка.

К интерфейсам, определённым «на границе» слоя приложения, подключатся интерфейсы всех внешних адаптеров уровня фреймворка: СУБД, системы хранения файлов, внешние сервисы, доступные по GRPC или SMTP, и т. д.

Слой фреймворка

Пока мы рассмотрели границу слоёв предметной области и приложения. Оба этих слоя взаимодействуют со слоями, которые находятся под нашим контролем. Слой предметной области взаимодействует со слоем приложения, слой приложения — со слоем фреймворка, а слой фреймворка — с внешним миром. С миром, заполненным различными протоколами. Как правило, это какие-то протоколы, основанные на TCP: HTTP или HTTPS, GRPC, SQL. Слой фреймворка содержит много кода — все библиотеки, которые мы используем. Не стоит забывать про код, написанный нами, например, хендлеры HTTP и реализации интерфейсов, объявленных во внутренних слоях. Здесь же располагаются те самые Data Mapper и Unit of work, рассмотренные выше.

Пример с запросом на регистрацию пользователя через HTTP-API

```
package framework

import (
    "encoding/json"
    "log"
    "net/http"
    "our_service/app"
)

func handler(w http.ResponseWriter, r *http.Request) {
    dec := json.NewDecoder(r.Body)
    defer r.Body.Close()
    var u app.User
    if err := dec.Decode(&u); err != nil {
       http.Error(w, err.String(), http.StatusBadRequest)
       return
    }
    if err := app.RegisterUser(u); err != nil {
       http.Error(w, err.String(), http.StatusInternalServerError)
    }
}

func ServeHTTP() {
    http.HandleFunc("/register_user", handler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}

// ......

//package app
// ....
type User struct {
    Name     string `json:"name"`
    Password string `json:"psw"`
    Email    string `json:"email"`
}

func (a *App) RegisterUser(u User) error {
    // ....
    return register.RegisterUser(u.Name, u.Email, u.Password)
}
```

Мы видим, как фреймворк обращается к методу слоя приложения и передаёт ему распознанного в запросе пользователя, а тот, в свою очередь, обращается к бизнес-логике. Такое разделение слоёв позволяет сделать совершенно другой адаптер внутри фреймворка, например, для протокола GRPC, который также вызовет метод RegisterUser в слое приложения. Слой приложения, где происходит вся дальнейшая логика, останется неизменным.

Пример сохранения нового пользователя в СУБД

```
package app

// ....
type App struct {
    //....
    nf  Notifyer
    usv UserSaver
}

type UserSaver interface {
    SaveUser(u register.User) error
}

func (a *App) ProvideNewUser(u register.User) error {
    // ....
    if err := a.usv.SaveUser(u); err != nil {
       return err
    }
    return a.nf.Notify(u.Email, "Welcome to our service, "+u.Name)
}

//....
// package dbuser
type UserMapper struct {
    db *sql.DB
}

func (um *UserMapper) SaveUser(u register.User) error {
    // um.db.Exec ...
}
```

Здесь мы видим, как слой приложения обращается к пакету dbuser, входящему в слой фреймворка, но не напрямую, а через интерфейс UserSaver. Пакет dbuser реализует этот интерфейс в виде шаблона Data Mapper, который мы обсуждали выше. Он будет использовать состояние пользователя, определённое в слое бизнес-логики (register.User), чтобы произвести преобразование его в сущность СУБД. Как видите, здесь мы в любой момент можем изменить реализацию UserMapper или подменить её на альтернативную, но остальной инструментарий в слое приложения или слое бизнес-логики не изменится.

Код уровня фреймворка, все импортируемые туда внешние пакеты и используемые внешние фреймворки предоставляют нам средства для взаимодействия с внешним миром, убирая необходимость снова и снова писать весь этот код.

В большинстве случаев нам не надо добавлять что-то на границе слоя фреймворка. Однако такие случаи бывают. Например, если мы делаем API для нашего приложения, у нас добавится забот на уровне HTTP. Обычно это реализация CORS, HTTP-кеширование, HATEOAS, аутентификация,

валидация и другие специфические для обработки HTTP-запросов проблемы. Все эти вещи, возможно, важны, для нашего приложения, но не для слоя предметной области или даже слоя приложения.

Шина команд

Отдельно стоит упомянуть концепцию команд и шины команд в гексагональной архитектуре. В однопоточном мире для решения задачи асинхронного выполнения команд, относящихся к некоторым use case (сценариям использования), используется шина команд.

В Go всё взаимодействие с внешним миром, так или иначе, осуществляется в независимых горутинах, создаваемых на каждый внешний запрос. Так устроена и стандартная библиотека Go, например, пакеты net и net/http, и различные внешние пакеты для работы с СУБД или GRPC.

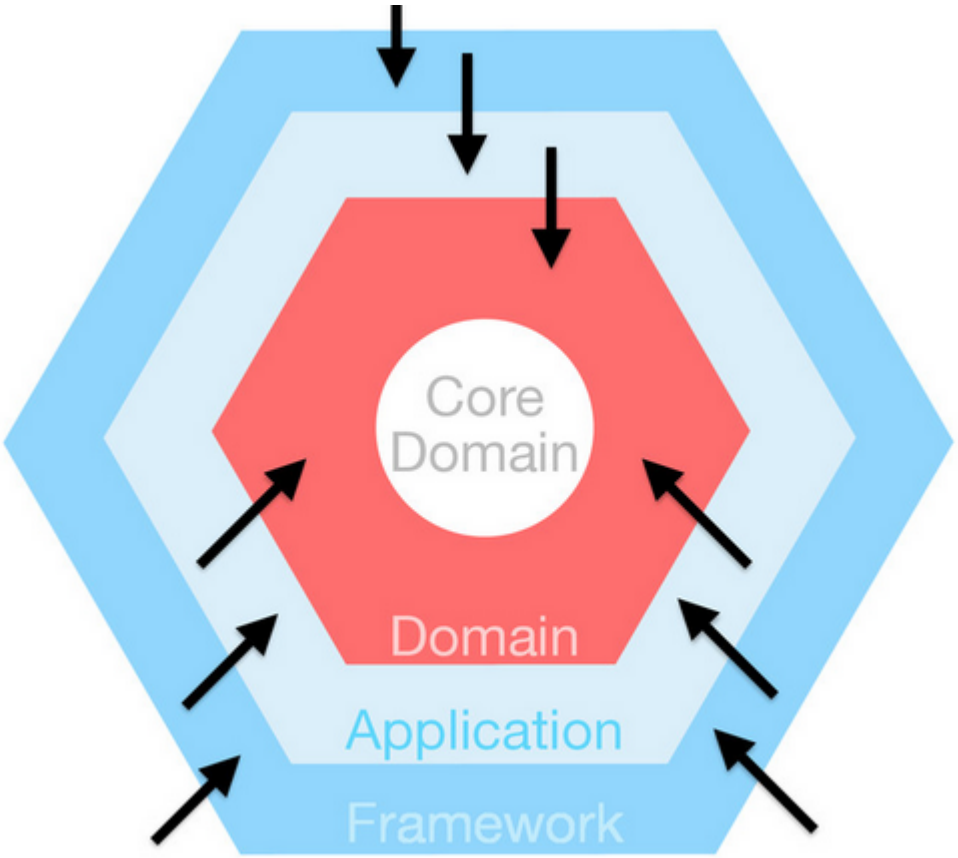
В роли шины команд в Go выступают каналы и горутины, которые используются внутри механизмов распараллеливания обработки входящих запросов, поэтому выполнение конкретного сценария использования возможно без применения шины команд. При желании обработку внутри сценариев использования можно легко распараллелить через пул воркеров или их цепочки.

Таким образом, шина команд в гексагональной архитектуре на Go именно как шина не реализуется.

Зависимости (импорты) в гексагональной архитектуре

В рамках гексагональной архитектуры мы придерживаемся одностороннего потока зависимостей: снаружи внутрь. Слой предметной области (центральный слой) не зависит от наружных слоёв и не импортирует их пакеты. А слой уровня приложения всегда зависит от слоя предметной области, импортируя его пакеты, но не от слоя фреймворка. Слой фреймворка зависит от слоя уровня приложения и импортирует его пакеты, но не зависит от пакета main. Кстати, однонаправленная зависимость в гексагональной архитектуре хорошо соответствует запрету циклических зависимостей в Go и невозможности импорта пакета main.

Ранее мы говорили, что интерфейсы — это основной механизм инкапсуляции изменений. Они позволяют нам регламентировать, как слои будут общаться друг с другом, избегая лишней связанности между ними. Если мы посмотрим на зависимости, то, по сути, придём к той же идее. Сейчас разберёмся.



Зависимости проще представить, когда мы прописываем данные или логику «внутрь». Когда на сервер приходит HTTP-запрос, у нас должен быть какой-то код, который обработает его, иначе ничего не произойдёт. Следовательно, внешний HTTP-запрос зависит от нашего слоя фреймворка, который используется для интерпретации запроса. Если фреймворку удалось интерпретировать запрос и определить маршрут к соответствующему обработчику, то этому обработчику потребуется что-то, чтобы выполнить действия. Без слоя уровня приложения ему нечего делать. Слой уровня фреймворка зависит от слоя уровня приложения. Чтобы слою уровня приложения было кем руководить, ему требуется слой предметной области. Главное — иметь возможность выполнить требуемое действие. Слой предметной области зависит в основном от поведения и ограничений, определённых в нём же.

Когда мы пытаемся проследить путь запроса, совершённого извне, внутрь нашего приложения, то зависимости проглядываются весьма явно. Внешние слои зависят от внутренних, но они могут полностью игнорировать существование более глубоких слоёв приложения. Слой знает только то, какой метод вызвать и какие данные туда передать. Детали реализации безопасно инкапсулированы там, где требуется. То, как мы использовали интерфейсы, хорошо это показывает.

Теперь проследим обратный путь.

Когда мы выходим «наружу», всё становится немного сложнее. Разберёмся, что делает наше приложение в ответ на какой-то запрос, то есть как происходит обработка запроса и формирование ответа.

Нашему слою предметной области, скорее всего, понадобится доступ к базе данных, чтобы загрузить оттуда сущности. Следовательно, слой предметной области зависит от какого-то хранилища данных. Слой уровня приложения после завершения какой-то задачи отправит уведомление пользователю. Если мы используем email для доставки уведомлений, используя AWS SES, то, можно сказать, что наш слой уровня приложения имеет зависимость от SES для организации доставки уведомлений.

Как вы могли заметить, в нашей концепции прослеживается зависимость внутренних слоёв от того, что содержится во внешних. Посмотрим, как инвертируется направление зависимостей.

Слово «инвертировать» связано с принципом Dependency Inversion или принципом инверсии зависимостей. Буква D в SOLID. Здесь нам снова потребуются интерфейсы.

Воспользуемся интерфейсами для инверсии зависимостей. С ними мы можем регламентировать, как будут взаимодействовать наши слои. И нас не интересует, как реализуются эти интерфейсы в других слоях. Таким образом, конкретный слой диктует внешним слоям, что он хочет, не завися от конечной реализации. Это и есть инверсия зависимостей.

Наш слой предметной области может объявить интерфейс репозитория наших сущностей. Этот интерфейс реализуется в одном из внешних слоёв, скорее всего, в слое уровня фреймворка. Однако из-за объявления интерфейса мы отделили нашу бизнес-логику от конкретного способа хранения данных. Это даёт возможность поменять слой хранения наших моделей при тестировании или же сменить его, если потребуется. Будет здорово, если нашему приложению понадобится гибкое масштабирование.

С нотификатором (сервисом уведомлений) похожая ситуация. Наш слой уровня приложения знает только то, что ему требуется что-то для отправки уведомления. Для этого мы и сделали интерфейс Notifyer. Приложению не надо знать, как мы отправим эти уведомления. Оно только определяет, как с ним взаимодействовать. Таким образом, интерфейс Notifyer объявится в слое уровня приложения и реализуется в слое уровня фреймворка. Благодаря этому, направление зависимости изменилось. Мы инвертировали его — сказали внешним слоям, как собираемся их использовать. Поскольку у нас есть возможность в любое время менять реализацию интерфейсов, то, можно сказать, что наши слои разделены.

Подведём итоги.

1. Интерфейсы используются для обозначения направления потока логики внутрь и наружу.
2. Мы использовали инверсию зависимостей, чтобы сохранить одинаковое направление зависимостей.

3. Отделили внутренние слои от внешних, продолжая ими пользоваться.

4. Мы написали идиоматический код на Go и не создали каких-либо монструозных интерфейсов с множеством методов, как это обычно бывает при неправильном разделении слоёв в целом или одного конкретного слоя на пакеты.

5. Всё выглядит лаконично, хорошо поддерживается и легко масштабируется.

Жирные интерфейсы

Обратимся к одной важной теме в контексте гексагональной архитектуры на Go. Часто между бизнес-логикой и слоем работы с СУБД создаётся огромный, жирный интерфейс с десятками и сотнями методов. Этот интерфейс также имплементируется в виде одного-единственного Data Mapper, который выполняет все эти методы на одной и той же СУБД. Идея такого объединения выглядит довольно логичной — мы объединяем все методы работы с СУБД в одном объекте.

Важно понять, что эта предпосылка исходит из позиции страха перед необходимостью поиска подходящих методов в разных интерфейсах при программировании того или иного инструментария, использующего инверсии зависимостей. Вместо того чтобы рассматривать слои архитектуры, разбитые внутри себя на различные пакеты, мы пытаемся уменьшить количество пакетов и сущностей для работы с конкретной СУБД, и хотим иметь возможность заменять их целиком, например, на тестовые моки. Интуитивно кажется, что так будет проще — единственный репозиторий для работы со всеми сущностями СУБД и бизнес-логики, всё лежит в одном месте, в одном пакете и даже одном типе. Если надо что-то поменять, то меняем это в одном месте.

Это желание возникает само по себе и не имеет никакого отношения к слоистой архитектуре. По сути, это обычный DTO (data transfer object), «завёрнутый» в огромный интерфейс. Насмотренность на большие объекты DTO в других языках программирования, СУБД или архитектурных стилях приводит к желанию абстрагировать слой работы с данными в единое целое — огромный жирный интерфейс.

При использовании гексагональной архитектуры так делать нельзя. Порты (интерфейсы) лежат внутри бизнес-логики, а адаптеры — снаружи. Чтобы этот шаблон работал должным образом, крайне важно создавать порты в соответствии с потребностями ядра приложения, а не просто имитировать или повторять API инструментов из внешних слоёв.

Важно, чтобы каждый интерфейс содержал минимальное количество методов, относящееся к конкретному поведению в рамках той или иной функции слоя бизнес-логики, приложения или фреймворка, который требуется в конкретном месте.

В результате небольшие интерфейсы оказываются разбросанными по разным пакетам внутри одного слоя. И это хорошо. Проблема дублирования мелких интерфейсов решается через их эмбеддинг друг в друга, но только там, где такой микс действительно необходим. Точно так же организована стандартная библиотека Go. Это считается идиоматическим стилем программирования на Go.

Чистая архитектура

Чистая архитектура объединяет в себе несколько видов ранее созданных архитектур (гексагональная, DCI, BCE). Ключевая из них — гексагональная архитектура.

Так или иначе, все архитектуры похожи. Они все преследуют одну цель — разделение задач. Эта цель достигается путём деления программного обеспечения на уровни. Каждая имеет хотя бы один уровень для бизнес-правил и ещё один для пользовательского и системного интерфейсов.

Все архитектуры способствуют созданию систем, обладающих следующими характеристиками:

1. Независимость от фреймворков. Архитектура не зависит от наличия какой-либо библиотеки. Это позволяет рассматривать фреймворки как инструменты, вместо того, чтобы стараться втиснуть систему в их рамки.

2. Простота тестирования. Бизнес-правила тестируются без пользовательского интерфейса, базы данных, веб-сервера и других внешних элементов.

3. Независимость от пользовательского интерфейса. Пользовательский интерфейс изменяется, не затрагивая остальной системы. Например, веб-интерфейс заменяется консольным интерфейсом, не изменяя бизнес-правил.

4. Независимость от базы данных. Вы можете поменять Oracle или SQL Server на Mongo, BigTable, CouchDB или что-то ещё. Бизнес-правила не привязаны к базе данных.

5. Независимость от внешних агентов. Ваши бизнес-правила ничего не знают об интерфейсах, ведущих во внешний мир.

По сути, чистая архитектура уточняет и слегка дополняет структуру внутри каждого слоя. С точки зрения нашего урока, эти отличия не играют принципиальной роли.

Круги на схеме выше лишь схематически изображают основную идею: иногда требуется больше четырёх кругов. Фактически нет такого правила, утверждающего, что кругов должно быть именно четыре. Но всегда действует правило зависимостей. Зависимости в исходном коде всегда направлены внутрь. По мере движения внутрь уровень абстракции и политик увеличивается. Внешний круг включает низкоуровневые конкретные детали. По мере приближения к центру программное обеспечение становится более абстрактным и инкапсулирует более высокоуровневые политики. Самый внутренний круг считается самым обобщённым и располагается на самом высоком уровне.

Обычно через границы данные передаются в виде простых структур. При желании используются простейшие структуры или объекты передачи данных — Data Transfer Objects; DTO. Данные можно также передавать в вызовы функций через аргументы. Или упаковывать их в ассоциативные массивы или объекты. Главное, чтобы через границы передавались простые, изолированные структуры данных. Не надо хитрить и передавать объекты сущностей или записи из базы данных. Структуры данных не должны нарушать правило зависимостей.

Например, многие фреймворки для работы с базами данных возвращают ответы на запросы в удобном формате. Их можно назвать «представлением записей». Такие представления записей не

передаются через границы внутрь. Это нарушает правило зависимостей, потому что заставляет внутренний круг знать что-то о внешнем круге.

Итак, при передаче через границу данные всегда будут принимать форму, наиболее удобную для внутреннего круга.

Следование этим простым правилам не требует больших усилий и позволит сохранить душевный покой в будущем. Разделив программное обеспечение на уровни и соблюдая правило зависимостей, вы создадите систему, которую легко протестировать — со всеми вытекающими из этого преимуществами.

Когда какой-либо из внешних элементов системы, например, база данных или веб-фреймворк, устареет, вы без суеты замените его.

Практическое задание

1. Создайте каркас приложения из пакетов, структур и интерфейсов для приложения, которое через REST-API позволяет управлять каталогом пользователей большой компании.
    1.1. Каталог состоит из пользователей и их окружений — проектов, организаций, корпоративных групп и сообществ.
    1.2. Одно окружение включает в себя много пользователей.
    1.3. Один пользователь входит в несколько окружений.
    1.4. Система позволяет:
        1.4.1. Добавлять пользователей и окружения.
        1.4.2. Назначать и убирать пользователей из окружений.
        1.4.3. Искать юзеров по имени или по названию окружения, куда они входят.
        1.4.4. Осуществлять поиск окружений по их названию или по именам входящих в них пользователей.
    1.5. Реализовывать полную функциональность не надо. Достаточно показать методы у подходящих структур (без тела).
    1.6. В выполненном задании будет проверяться:
        1.6.1. Корректность группировки пакетов и слоёв в иерархии папок проекта.
        1.6.2. Наличие интерфейсов и их размеры на границах слоёв.
        1.6.3. Имплементация интерфейсов.

1.6.4. Наличие общих, связующих все слои, механизмов при запуске приложения.

2. Покажите, что умеете использовать шаблоны Data Mapper и Unit of Work внутри пакетов.

Глоссарий

1. **Архитектурный слой** — набор пакетов, содержащих функциональность одного типа — фреймворк (адаптер), приложение или бизнес-логика.

2. **Гексагональная архитектура** — архитектура портов и адаптеров, свод правил разделения исходного кода на функциональные слои так, чтобы внешние адаптеры — API, СУБД и прочие — можно было подменять, исправлять или улучшать независимо от внутренних слоёв и бизнес-логики приложения.

Дополнительные материалы

1. Catalog of patterns of enterprise application architecture.

2. Р. Мартин. Чистая архитектура. Искусство разработки программного обеспечения.

Используемые источники

1. Р. Мартин. Чистая архитектура. Искусство разработки программного обеспечения.

2. М. Фаулер. Шаблоны корпоративных приложений.

3. Э. Эванс. Предметно-ориентированное проектирование.

4. К. Ричардсон. Микросервисы. Паттерны разработки и рефакторинга.