GeekBrains

Архитектурные паттерны монолитов и микросервисов

Backend-разработка на Go. Модуль 2

Что будет на уроке

1. Базовые архитектурные паттерны, применяемые при разработке монолитных решений и решений с использованием микросервисов
2. Организация кода приложения на Go в зависимости от выбранной архитектуры

История монолита

Переход к микросервисам

Вертикальное масштабирование

Преимущества монолитов

Проблемы монолитов

Горизонтальное масштабирование

Преимущества микросервисов

Недостатки микросервисов

Вертикальное масштабирование

Преимущества и недостатки

Высокая связанность компонентов

Высокая производительность

Рост сложности систем

Верхний предел роста мощностей

Единовременный релиз компонентов

Рост time2market

Горизонтальное масштабирование

Преимущества и недостатки

Низкая связанность компонентов, независимый релиз компонентов, разные технологические стеки

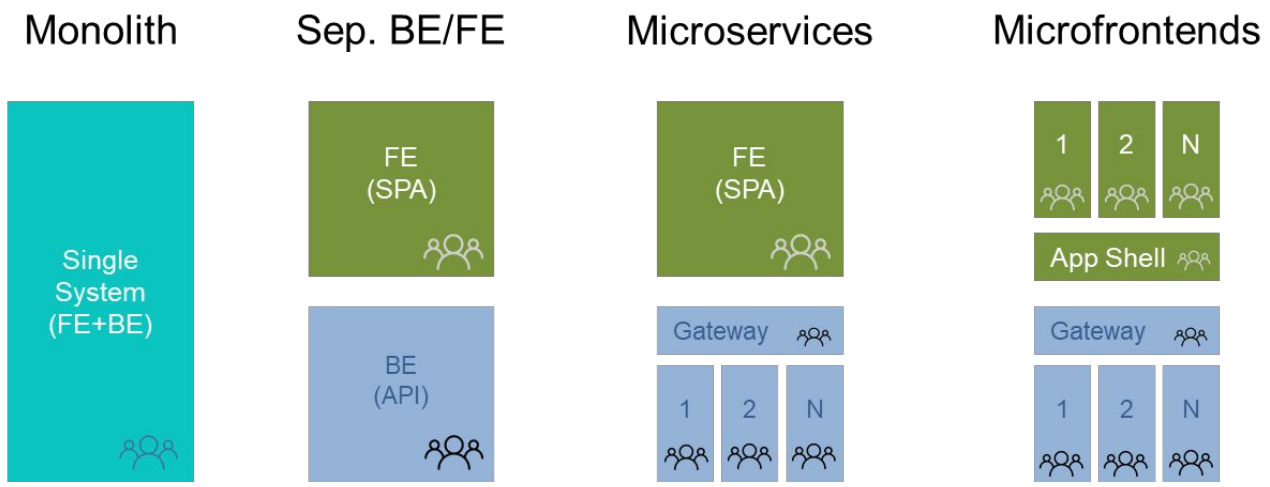
Высокая производительность отдельного компонента

Разные команды, компетенции и ЖЦ компонентов

Сложность изоляции контекста

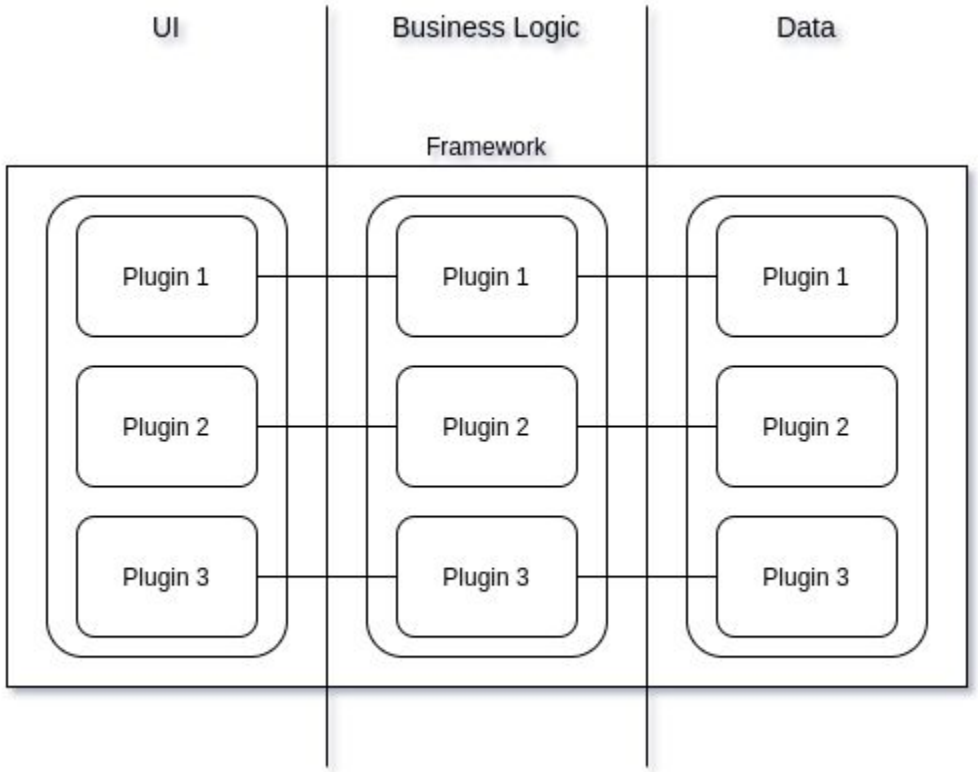
Низкая производительность взаимодействий между компонентами (требуется асинхронность)

Переход от монолита к микросервисам



Переход от монолита к микросервисам

С чего начать?

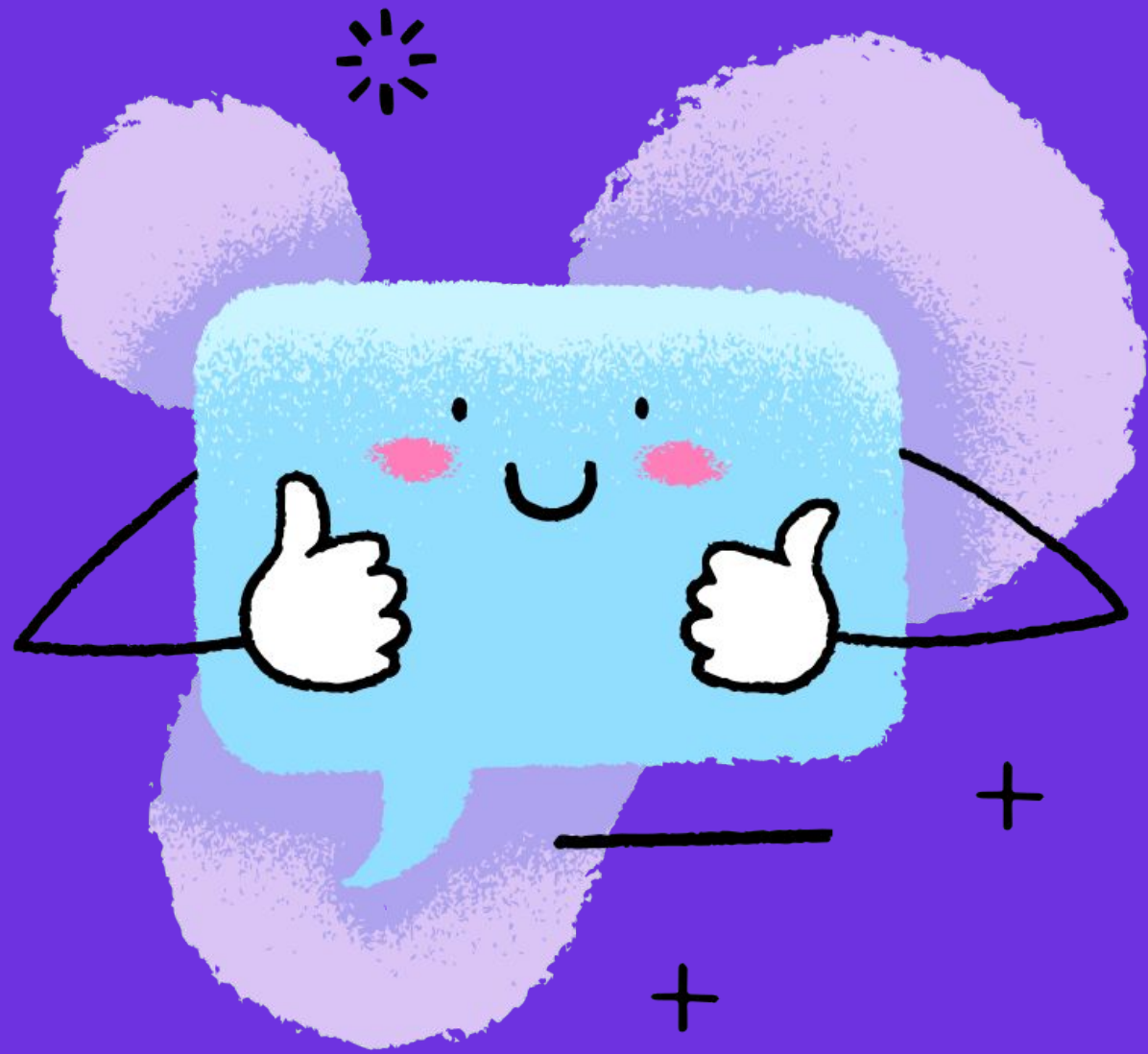


Переход от монолита к микросервисам

Начало

- Делим инструментарий на модули
- Выносим модули в отдельные сервисы с синхронным API
- Заменяем синхронное взаимодействие на асинхронное посредством очередей

Принципы структурирования приложений на Go



GeekBrains

Слои

- API
- Аутентификация и авторизация
- Логика обработки данных
- Кеш
- Персистентное хранилище (СУБД, файлы)

Функции кода

- Обеспечивать хорошую поддерживаемость:

  другим программистам будет легко читать код, понимать, как он работает, быстро находить и исправлять проблемы
- С минимальными сложностями (максимально быстро) вносить изменения в код

Инструменты реализации архитектуры в Go

- Кроме пакетов, ничего нет
- Структура пакетов отражает архитектуру
- Запрет циклических зависимостей принуждает делать код правильным

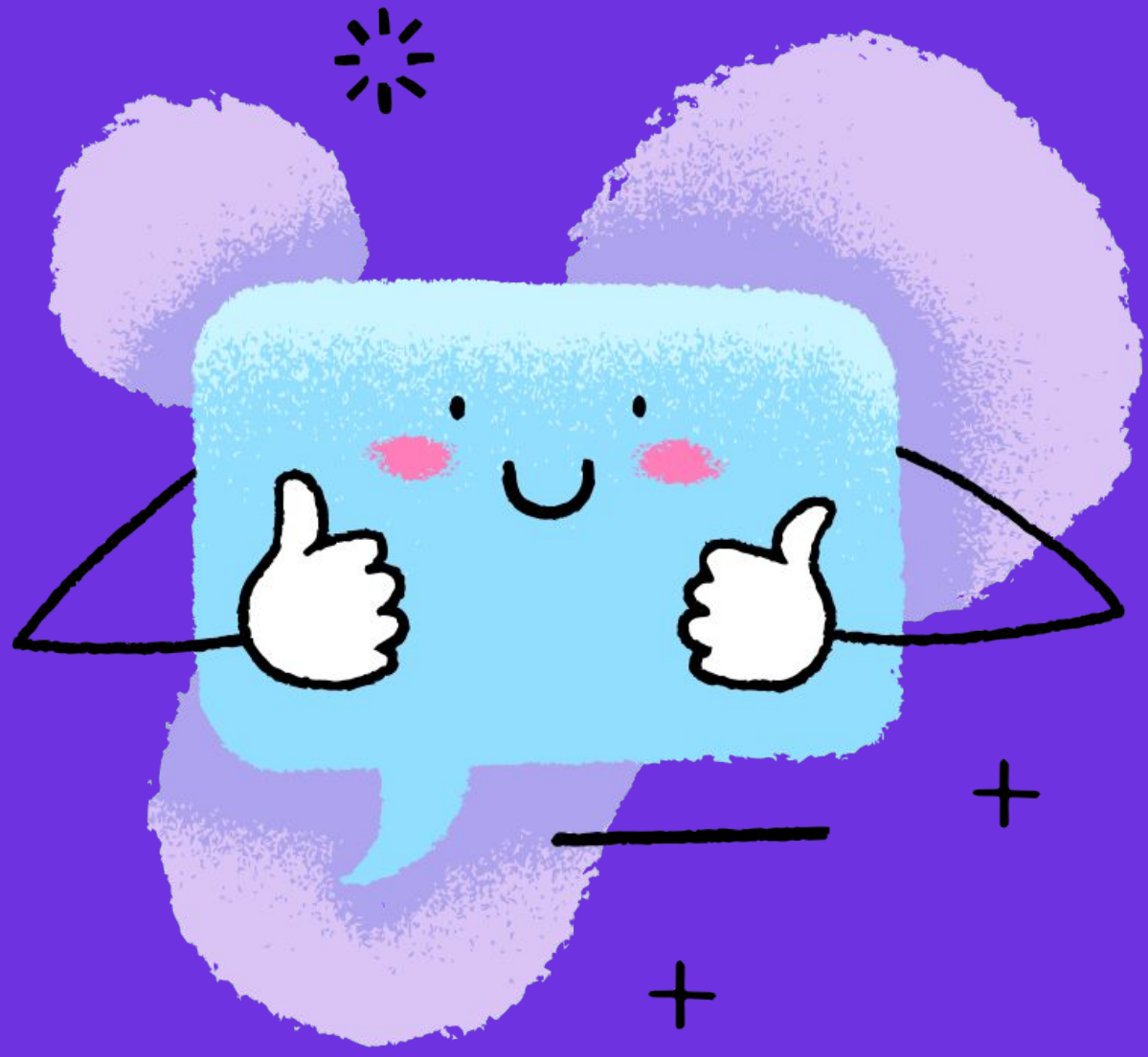
Частота изменений

- Часто изменяемые пакеты — нестабильные
- Редко изменяемые пакеты — стабильные
- Пакеты стандартной библиотеки максимально стабильны, и от них зависит вся функциональность в ваших пакетах
- Менее стабильные пакеты импортируют более стабильные пакеты, но не наоборот
- Инверсия зависимости через интерфейсы (пример: sort)

Функциональная обособленность

- В каждом отдельном файле пакета есть изолированная законченная функциональность, относящаяся к конкретному типу или к определённому поведению для этого типа
- Названия пакетов и файлов отражают поведенческие особенности, не надо использовать классифицирующие названия типа db, api, common и т. д.
- А вот папки, в которые объединяются пакеты, называются как угодно
- Особенная папка: internal

Архитектуры кода на Go



GeekBrains

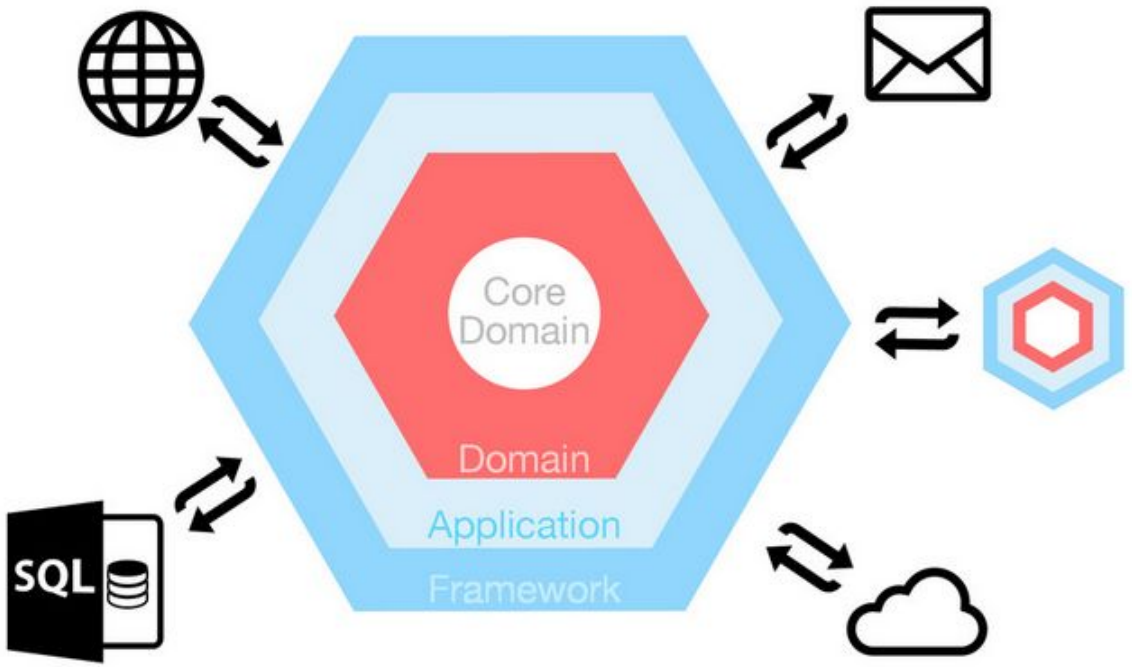
«Никакая» архитектура

- Иногда называется «линейной»
- Код пишется в той же последовательности, в которой он потом выполняется
- Архитектура получается «случайно» в результате выделения общих паттернов или дедупликации кода

MVC, MVP и MVVM

- MVC почти неприменим, так как view обычно не имеет доступа к событиям и данным модели в обход контроллера
- MVP — частый шаблон в Go, например, Presenter — http.Handler
- MVVM — реализовать ViewModel очень сложно, поскольку невозможно языковыми средствами обеспечить связывание полей структур по аналогии, как это сделано в javascript

Hexagonal (Clean)



Hexagonal (Clean)

- На границе каждого слоя мы найдём интерфейсы
- Интерфейсы — это порты для внешних слоёв, где реализуются адаптеры
- Интерфейс определяется там, где используется
- Принимаем интерфейс, возвращаем конкретный тип

Hexagonal (Clean)

Ссылка

Hexagonal (Clean)

- Сначала пишем бизнес-логику. От неё зависит всё (её все импортируют)
- Потом пишем реализацию API
- Об API имеет смысл договариваться, когда уже есть конкретный слой бизнес-логики
- Закрыть фасадом (интерфейсом) бизнес-логику не получится из-за циклической инверсии зависимостей, запрещённой в Go

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.

Спасибо!

Каждый день вы становитесь лучше :)