МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра вычислительной математики

Курсовой проект

Разработка микросервисных масштабируемых приложений на языке Golang

Гашникова Анастасия Станиславовна

студент 3 курса 5 группы,
специальность «прикладная математика»

Эксперт-консультант от филиала кафедры:
Черник Дмитрий Викторович

Руководитель от кафедры:
Мандрик Павел Алексеевич

Минск, 2021

Постановка задачи

Создать контейнерезированное микросервисное приложение по выбранной тематике.

Требования к реализации:

1) от 3 разделенных по доменным областям сервисов, например:сервис управления пользователями, сервис для обработки логического состояния, сервис каталогизации/справочной информации
2) разделить хранение данных на несколько изолированных хранилищ(БД)
3) реализовать коммуникацию и хранение промежуточного состояния, используя либо систему обмена сообщениями(rabbitmq, service bus,etc…), либо протокол бинарного взаимодействия(gRPC)
4) подготовить каждое из приложений для запуска в изолированном окружении, используя любую систему для запуска и управления контейнерами(docker swarm, kubernetes)
5) реализовать автоматизацию на основе базовой (basic) или на основе токена авторизации(jwt)
6) удостовериться в корректности работы приложения, добавив автоматические юнит-тесты
7) протестировать конечную реализацию на наличие состояний гонки

Оглавление

ВВЕДЕНИЕ

Микросервисные приложения стали популярны в середине 2010-х и их применение продолжает набирать обороты. Микросервисная система основывается на выстраивании приложения из компонентов, выполняющих разные функции, взаимодействующие с использованием сетевых протоколов (в стиле REST с использованием, например, Protocol Buffers, который я использовала в своей работе).

Микросервисная архитектура имеет свои преимущества, такие как:

1.Простота развертывания

Можно развертывать только изменяющиеся микросервисы, независимо от остальной системы, что позволяет производить обновления чаще и быстрее.

2.Оптимальность масштабирования

Можно расширять только те сервисы, которые в этом нуждаются, то есть сервисы с наименьшей производительностью, оставляя работать остальные части системы на менее мощном оборудовании.

3.Устойчивость к сбоям

Отказ одного сервиса не приводит к остановке системы в целом. Когда же ошибка исправлена, необходимое изменение можно развернуть только для соответствующего сервиса — вместо повторного развертывания всего приложения. Правда, для этого еще на этапе проектирования микросервисов потребуется тщательно продумать связи между ними для достижения максимальной независимости друг от друга, а также заложить возможность корректного оповещения пользователя о временной недоступности определенного сервиса без ущерба для всей системы.

4.Возможность выбора технологий

Можно подбирать различные наборы технологий, оптимальные для решения задач, стоящих перед отдельными сервисами, а также возможность использовать несколько языков программирования

5.Небольшие команды разработки

При разработке микросервисов команды принято закреплять за конкретными бизнес-задачами (и сервисами, соответственно). Такие команды, как правило, показывают большую эффективность, а управлять ими легче.

6.Уменьшение дублирования функциональностей

Присутствует возможность повторного использования функциональности для различных целей и различными способами.

7.Упрощение замены сервисов при необходимости

Небольшие сервисы проще заменить на более подходящую версию или удалить вовсе — это несет значительно меньше рисков по сравнению с монолитным приложением.

8.Независимость развертывания

Каждый микросервис, как правило, использует собственное хранилище данных — поэтому изменение модели данных в одном сервисе не влияет на работу остальных.Каждый из микросервисов изолируется в отдельный контейнер доступный по сети другим микросервисам, а также имеет полный набор технологий.

Один микросервис может быть написан, например, на Java,другой-на Go.Тем не менее они смогут взаимодействовать.

Приложение было написано на языке Golang, который был разработан компанией Google на основе языка C беря из него всё самое лучшее и стремясь избавиться от недостатков. Так, Go может похвастаться высокой скоростью выполнения, которую он перенял у C и в то же время Go является достаточно простым и очень удобным для разработки приложений.

Особенностями языка являются простой и понятный синтаксис,легкость интеграции и поддержки кода, возможность динамического ввода данных, а также быстрота компиляции и мощный параллелизм, встроенные типы данных.

Отдельно стоит подчеркнуть его направленность на многопоточные приложения. Используя goroutines (горутины) – лёгковесные потоки, можно достаточно просто разрабатывать действительно хорошие многопоточные приложения.

Go не входит в список объектно-ориентированных языков программирования (ООП) и в нём отсутствует такой инструмент ООП, как классы. Но на замену классам в Go приходят интерфейсы позволяют сделать код немного проще и понятнее.

Это быстрый, статически типизированный, многопоточный язык, который прекрасно подходит для бэкенд-разработки и создания микросервисных приложений.

Глава 1

Написание микросервисов

В первую очередь необходимо обозначить архитектуру приложения: “разбить” приложение на микросервисы, подобрать базу данных для хранения, предусмотреть возможность изменения структуры базы с сохранением уже хранящихся в ней данных, а также построить соответствие между базой данных и микросервисами. Эти вопросы будут рассмотрены в данной главе.

Выбранная мною тема-прокат велосипедов.Первоначально пользователь входит в систему и происходит его авторизация, всего 2 роли пользователя:обычный юзер или администратор.

Администратор
У администратора есть доступ ко всем велосипедам, у него есть данные по всем велосипедам и по всем пользователям
Обычный пользователь.
У обычного пользователя есть доступ к списку только свободных велосипедов

1.1 Авторизация пользователей

Прежде чем зайти в систему, пользователь должен авторизоваться.Сначала мы устанавливаем соединения с базой данных. В ней мы создаем таблицу с параметрами пользователя ID, его именем, паролем,балансом(эти данные будут храниться в БД MySQL)(Приложение Б). Реализовываем интерфейс работы с БД(Приложение А),а также каждому пользователю присваивается jwt-токен.Токены создаются сервером, подписываются секретным ключом и передаются клиенту, который в дальнейшем использует данный токен для подтверждения своей личности. В токене авторизации зашифрованы ID этого пользователя и его роль(администратор,обычный юзер). При обращении к

сервису проката эта информация используется для разрешения доступа к тому или иному функционалу.(Приложение Г)

Пароль пользователя хэшируется(Алгоритм работает так, что для каждого текста генерируется уникальный хеш. И восстановить текст из хеша, перехватив сообщение, практически невозможно)

1.2.Проверка токена на валидность

Для данного микросервиса будем использовать промежуточное ПО JWT.

JWT предоставляет промежуточное ПО для аутентификации JSON Web Token (JWT).

Для действительного токена он устанавливает пользователя в контексте и вызывает следующий обработчик.

Для недействительного токена он отправляет ответ «401 - неавторизован».

Если заголовок авторизации отсутствует или недействителен, он отправляет «400 - неверный запрос».

Данный микросервис проверяет токен на валидность и возвращает роль и id пользователя(всего 2 вида пользователей:админ и обычный юзер).
Мы декодируем jwt token,(который был выдан пользователю при входе в систему),получаем из него ID и роль пользователя(Приложение В). Далее реализуем grpc handler и echo сервер

1.3 Прокат велосипедов

В данном микросервисе хранится вся информация о велосипедах: id, адрес, id пользователя, время последнего использования.Интерфейс работы с соединением с базой данных(Приложение Д).

Глава 2
Выбранные технологии

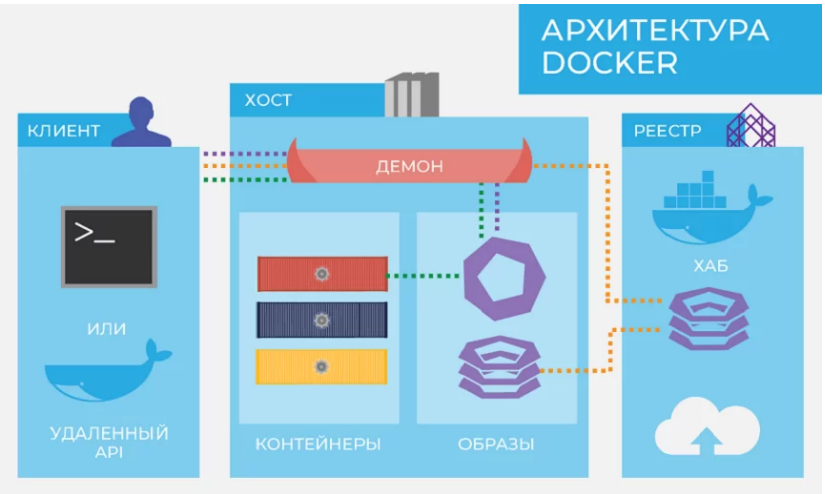
Было создано клиент-серверное приложение, что означает необходимость предоставить пользователю возможность регистрации и входа в систему, а также просмотр пользователем необходимых данных. Реализация данных функций будет рассмотрена в этой главе. А также возможность независимого развертывания микросервисов.

2.1 Настройка Docker в приложении

Docker — это ПО для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации. Он нужен для развертывания микросервисов, а также для их масштабирования и переноса в другие среды, где приложения смогут также гарантированно работать.

Контейнеризация похоже на виртуализацию, но это не одно и то же. Виртуализация работает как отдельный компьютер, со своим виртуальным оборудованием и операционной системой. При этом внутри одной ОС можно запустить другую ОС. В случае контейнеризации виртуальная среда запускается прямо из ядра основной операционной системы и не виртуализирует оборудование. Это означает, что контейнер может работать только в той же ОС, что и основная. При этом так как контейнеры не виртуализируют оборудование, они потребляют намного меньше ресурсов.

Прежде чем приступить к написанию микросервисов необходимо установить докер и писать приложение именно в нем, это упростит дальнейшее написание микросервисов. Перед началом работы необходимо разобраться с его архитектурой:



Основной принцип работы Docker — контейнеризация приложений. Контейнеры Docker получаются из образов Docker. Образ содержит в себе все необходимое для запуска и работы: операционную систему, runtime, и приложение, подготовленное к развертыванию.

Пользователь отдает команду докер-демону с использованием REST API, который осуществляет взаимодействие с контейнерами(создание, удаление, запуск). Далее докер-демон работает с образами, исходя из того, что запросил пользователь, для корректной работы докера использует докерфайл(Dockerfile), который есть у каждого образа. В файлах Dockerfile содержатся инструкции по созданию(сборке) образа. В нем указываются все программы, зависимости и образы, которые нужны для разворачивания образа. Количество докер-файлов равно количеству наших микросервисов.

Docker предоставляет такие возможности,как:

- Изоляция запущенных приложений. Можно отделить приложение от ОС хоста и обращаться с ним как с управляемым приложением.
- возможность разворачивать несколько контейнеров одновременно
- среда для запуска поставляется вместе с приложением, поэтому конфигурация среды не обязательна

- упаковка в один образ приложения и всех зависимостей (библиотеки и файлы настройки), это позволяет упростить перенос микросервисов в другую среду
- контейнер позволяет мигрировать приложение без необходимости заново прописывать разрешения, конфигурацию, нужно будет просто запустить готовый контейнер на новом хосте
- Docker позволяет запускать любое программное обеспечение, даже запуск небезопасного кода
- Docker позволяет фиксировать некоторое состояние файловой системы, и повторно использовать его в том случае, если это состояние является общим для нескольких образов.

2.2 Фреймворк gRPC для связи между микросервисами

gRPC- один из стандартов для удаленного вызова процедур, что же это означает?
За счет gRPC программы вызывают функции, которые могут находиться в другом адресном пространстве,именно поэтому мы его выбрали для коммуникации между микросервисами.
В gRPC клиентское приложение может напрямую вызывать метод серверного приложения на другом компьютере, как если бы это был локальный объект, что упрощает создание распределенных приложений и сервисов. Как и во многих системах RPC, gRPC основан на идее определения службы, определяя методы, которые могут вызываться удаленно с их параметрами и типами возвращаемых данных. На стороне сервера сервер реализует этот интерфейс и запускает сервер gRPC для обработки клиентских вызовов.
Клиенты и серверы gRPC могут работать и взаимодействовать друг с другом в различных средах - от серверов внутри Google до собственного ПК - и могут быть написаны на любом из поддерживаемых языков gRPC.

По умолчанию gRPC использует Protocol Buffers, зрелый механизм Google с открытым исходным кодом для сериализации структурированных данных (хотя его можно использовать с другими форматами данных, такими как JSON). Первым шагом при работе с буферами протокола является определение структуры данных, которые необходимо сериализовать в файле proto: это обычный текстовый файл с расширением .proto. Данные буфера протокола структурированы как сообщения, где каждое сообщение представляет собой небольшую логическую запись информации, содержащую серию пар имя-значение, называемых полями.

Затем, после того, как были указаны структуры данных, используется протокол компилятора буфера протокола, чтобы сгенерировать классы доступа к данным на языке из определения прототипа. Они предоставляют простые средства доступа для каждого поля, такие как name () и set_name (), а также методы для сериализации / синтаксического анализа всей структуры в / из необработанных байтов.

gRPC использует протокол protoc со специальным плагином gRPC для генерации кода из вашего proto-файла: вы получаете сгенерированный код клиента и сервера gRPC, а также код буфера обычного протокола для заполнения, сериализации и извлечения типов ваших сообщений. Ниже вы увидите пример этого.

Основная его специфика gRPC в том, что он сохраняет данные в бинарном виде, а ключами выступают числовые индексы, а не имена полей.Для передачи данных используются proto-запросы и proto-ответы. Что такое proto? Это язык разметки интерфейсов, декларативное описание нашей модели и контрактов в целом. Рассмотрим пример.Каждый сервис имеет имя, список методов (с ключевым словом RPC), входящие данные и результирующие данные. В message тоже есть имя и список полей, имя поля и индекс. И именно индекс используется при сериализация в формат protobuf.

Причины использования gRPC:

- Неэффективность протокола HTTP/1.1 -в данный момент это уже устаревший протокол, вместо которого используется HTTP/2.0(который использует gRPC), предлагающий мультиплексирование, приоритезацию потока, бинарное кадрирование,полное сжатие данных, контроль трафика
- Необходимость натягивать нашу модель данных и событий на REST+CRUD
- поддержка самых популярных основных языков программирования
- поддержка gRPC в публичных API от Google
- Protobuf в качестве инструмента описания типов данных и сериализации.

2.3 Использование баз данных

Для корректной работы приложения база данных должна поддерживать целостность данных даже при одновременном доступе большого количества пользователей. К тому же, необходимо, чтобы база была рассчитана на хранение больших объемов данных.

Поэтому для хранения данных в каждом сервисе была выбрана реляционная база данных MySQL.

Преимущества MySQL:

Одной из сильных сторон MySQL является ее архитектура. MySQL может применяться в среде клиент-сервер, что дает массу преимуществ как пользователям, так и разработчикам.

Простота в использовании. MySQL достаточно легко инсталлируется, а наличие множества плагинов и вспомогательных приложений упрощает работу с базами данных.
Обширный функционал. Система MySQL обладает практически всем необходимым инструментарием, который может понадобиться в реализации практически любого проекта.

Безопасность. Система изначально создана таким образом, что множество встроенных функций безопасности в ней работают по умолчанию.
Масштабируемость. Являясь весьма универсальной СУБД, MySQL в равной степени легко может быть использована для работы и с малыми, и с большими объемами данных.
Скорость. Высокая производительность системы обеспечивается за счет упрощения некоторых используемых в ней стандартов.
Открытость кода.Существует возможность добавлять в пакет нужные функции, расширяя его функциональность так, как требуется.
Основу MySQL составляет серверный процесс базы данных. Он выполняется на одном сервере.

Доступ из приложений к данным базы осуществляется посредством процесса базы данных. Клиентские программы не могут получить доступ к данным самостоятельно, даже если они работают на том же компьютере, на котором выполняется серверный процесс.

Такое разделение клиентов и сервера позволяет построить распределенную систему. Можно отделить клиентов от сервера посредством сети и разрабатывать клиентские приложения в среде, удобной для пользователя. Например, можно реализовать базу данных под UNIX и создать клиентские приложения, которые будут работать в системе Microsoft Windows.

Несколько клиентов подсоединяются к серверу по сети. MySQL ориентирована на протокол TCP/IP – это может быть локальная сеть или Интернет.

Благодаря тому, что манипулирование данными сосредоточено на сервере, СУБД не приходится контролировать многочисленных клиентов, получающих доступ в совместно используемый каталог сервера, и MySQL может поддерживать целостность данных даже при одновременном доступе большого количества пользователей.

2.4 JWT-токен

JSON Web Token (JWT) — это открытый стандарт для создания токенов доступа, основанный на формате JSON.

Как правило, используется для передачи данных для аутентификации в клиент-серверных приложениях. Токены создаются сервером, подписываются секретным ключом и передаются клиенту, который в дальнейшем использует данный токен для подтверждения своей личности. Создание jwt-токена((Приложение Д).

Токен JWT состоит из трех частей:

1. заголовок (header)
2. полезная нагрузка (payload)
3. подпись или данные шифрования

Первые два элемента — это JSON объекты определенной структуры. Третий элемент вычисляется на основании первых и зависит от выбранного алгоритма (в случае использования не подписанного JWT может быть опущен).

Токены могут быть перекодированы в компактное представление (JWS/JWE Compact Serialization): к заголовку и полезной нагрузке применяется алгоритм кодирования Base64-URL, после чего добавляется подпись и все три элемента разделяются точками («.»).

Заголовок

В заголовке указывается необходимая информация для описания самого токена.

Обязательный ключ здесь только один:

• alg: алгоритм, используемый для подписи/шифрования (в случае не подписанного JWT используется значение «none»).

Необязательные ключи:

• typ: тип токена (type). Используется в случае, когда токены смешиваются с другими объектами, имеющими JOSE заголовки. Должно иметь значение «JWT».

• cty: тип содержимого (content type). Если в токене помимо зарегистрированных служебных ключей есть пользовательские, то данный ключ не должен присутствовать. В противном случае должно иметь значение «JWT»

Полезная нагрузка

В данной секции указывается пользовательская информация (например, имя пользователя и уровень его доступа), а также могут быть использованы некоторые служебные ключи.

Все они являются необязательными:

• iss: чувствительная к регистру строка или URI, которая является уникальным идентификатором стороны, генерирующим токен (issuer).

• sub: чувствительная к регистру строка или URI, которая является уникальным идентификатором стороны, о которой содержится информация в данном токене (subject). Значения с этим ключом должны быть уникальны в контексте стороны, генерирующей JWT.

• aud: массив чувствительных к регистру строк или URI, являющийся списком получателей данного токена. Когда принимающая сторона получает JWT с данным ключом, она должна проверить наличие себя в получателях — иначе проигнорировать токен (audience).

• exp: время в формате Unix Time, определяющее момент, когда токен станет не валидным (expiration).

• nbf: в противоположность ключу exp, это время в формате Unix Time, определяющее момент, когда токен станет валидным (not before).

• jti: строка, определяющая уникальный идентификатор данного токена (JWT ID).

Схема работы

Как правило, при использовании JSON токенов в клиент-серверных приложениях реализована следующая схема:

1. Клиент проходит аутентификацию в приложении (к примеру, с использованием логина и пароля).

2. В случае успешной аутентификации, сервер отправляет клиенту access- и refresh-токены.

3. При дальнейшем обращении к серверу, клиент использует access-токен. Сервер проверяет токен на валидность и предоставляет клиенту доступ к ресурсам.

4. В случае, если access-токен становится не валидным, клиент отправляет refresh-токен, в ответ на который сервер предоставляет два обновленных токена.

5. В случае, если refresh-токен становится не валидным, клиент опять должен пройти процесс аутентификации.

JSON Web Token (JWT) — это открытый стандарт для создания токенов доступа, основанный на формате JSON.

Преимущества JSON Web Token (JWT) над куки:

• При использовании куки сервер должен хранить информацию о выданных сессиях, в то время как использование JWT не требует хранения дополнительных данных о выданных токенах: все, что должен сделать сервер — это проверить подпись.

• Сервер может не заниматься созданием токенов, а предоставить это внешним сервисам.

• В JSON токенах можно хранить дополнительную полезную информацию о пользователях. Как следствие — более высокая производительность. В случае с куки иногда необходимо осуществлять запросы для получения дополнительной

информации. При использовании JWT эта информация может быть передана в самом токене.

• JWT делает возможным предоставление одновременного доступа к различным доменам и сервисам.

ЗАКЛЮЧЕНИЕ

В ходе выполнения проекта были рассмотрены и решены следующие задачи:

1) создание контейнерезированного микросервисного приложения по выбранной тематике(приложение по прокату велосипедов).
2) создано 3 сервиса: сервис авторизации пользователей, сервис проверки jwt-токена, сервис проката велосипедов
3) использование БД MySQL для разделенного хранения данных
4) реализована коммуникация и хранение промежуточного состояния,используя протокол бинарного взаимодействия (gRPC)
5) реализована автоматизацию на основе токена авторизации(jwt)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. https://golang.org/doc/effective_go
2. https://echo.labstack.com/guide/
3. https://jmoiron.github.io/sqlx/
4. https://github.com/golang-standards/project-layout
5. https://ru.wikipedia.org/wiki/MySQL
6. MySQL documentation. – Mode of access: https://dev.mysql.com/doc/

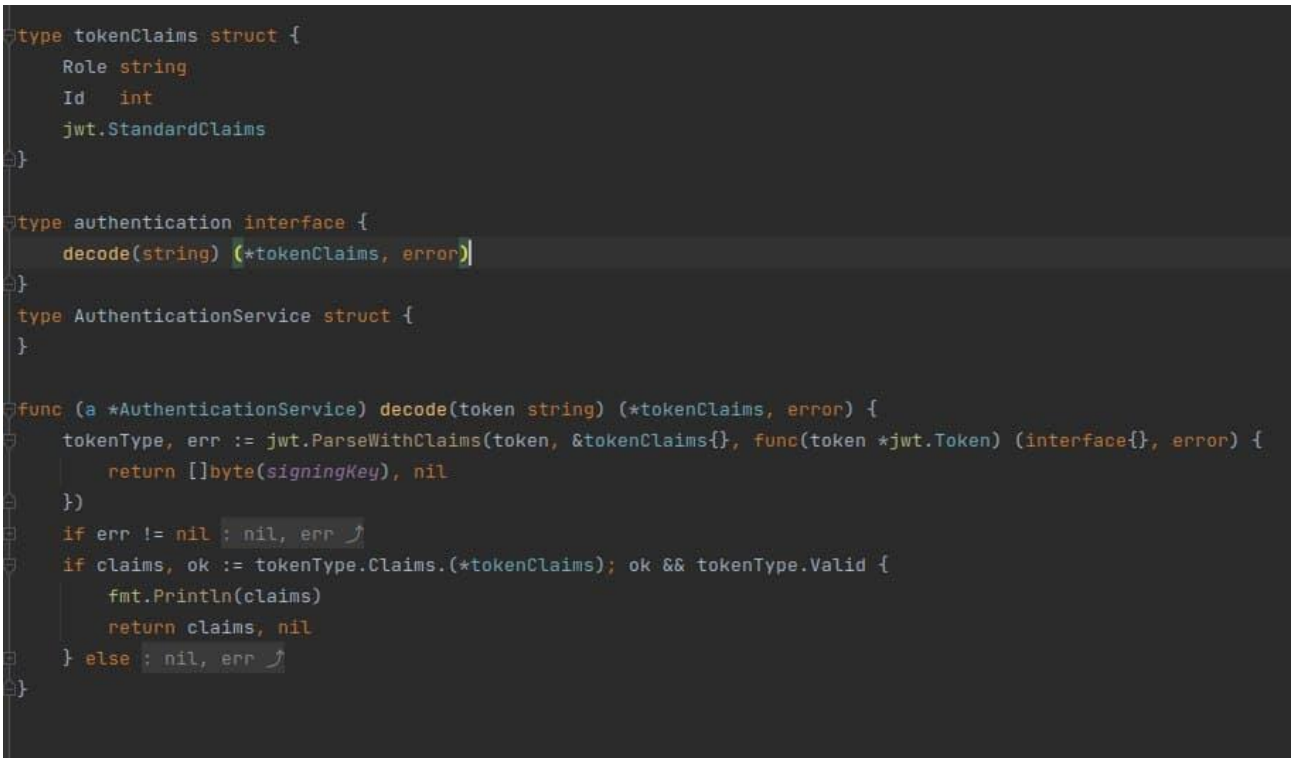
ПРИЛОЖЕНИЕ А

```
var config = mysql.Config{
    User:      "nastya",
    Passwd:    "1234",
    Net:       "tcp",
    Addr:      "localhost:3306",
    DBName:    "BikeService",
    Collation: "",
}

type repository interface {
    CreateUser(*User) (string, error)
    GetByName(string) (*User, error)
    GetAll() ([]*User, error)
    DeleteUser(int) (string, error)
    SetBalance(id int, addMoney int) (string, error)
}

type Repository struct {
    db *sql.DB
}

func NewRepository() repository {
    db, err := sql.Open( driverName: "mysql", config.FormatDSN())
    if err != nil { return nil }
    return &Repository{db: db}
}
```

ПРИЛОЖЕНИЕ Б

	id	name	password	balance	role
1	2	vi	$2a$10$SI6SkNORi2fH6Xh.IgEgKOwDpEX52b7OrlIt9UW5SY91YWJcEngxm	0	user

ПРИЛОЖЕНИЕ В



```
type tokenClaims struct {
    Role string
    Id   int
    jwt.StandardClaims
}

type authentication interface {
    decode(string) (*tokenClaims, error)
}
type AuthenticationService struct {
}

func (a *AuthenticationService) decode(token string) (*tokenClaims, error) {
    tokenType, err := jwt.ParseWithClaims(token, &tokenClaims{}, func(token *jwt.Token) (interface{}, error) {
        return []byte(signingKey), nil
    })
    if err != nil { return nil, err }
    if claims, ok := tokenType.Claims.(*tokenClaims); ok && tokenType.Valid {
        fmt.Println(claims)
        return claims, nil
    } else { return nil, err }
}
```

ПРИЛОЖЕНИЕ Г

```go
type tokenClaims struct {
    Role string
    Id   int
    jwt.StandardClaims
}
type tokenMaker interface {
    CreateToken(int, string) (string, error)
}

type TokenMaker struct {
    repository
}

func (t *TokenMaker) CreateToken(Id int, Role string) (string, error) {
    token := jwt.NewWithClaims(jwt.SigningMethodES256, &tokenClaims{
        StandardClaims: jwt.StandardClaims{
            ExpiresAt: time.Now().Add(1 * time.Hour).Unix(),
            IssuedAt:  time.Now().Unix(),
        }, Role: Role, Id: Id,
    })
    return token.SignedString([]byte(signingKey))
}
```

Приложение Д

```go
type repository interface {
    CreateBike(address string) (string, error)
    RentBike(userId, bikeId int) (string, error)
    ReturnBike(userId, bikeId int, newAddress string) (int, error)
    GetAll() ([]*Bike, error)
}
type Repository struct {
    db *sql.DB
}
```