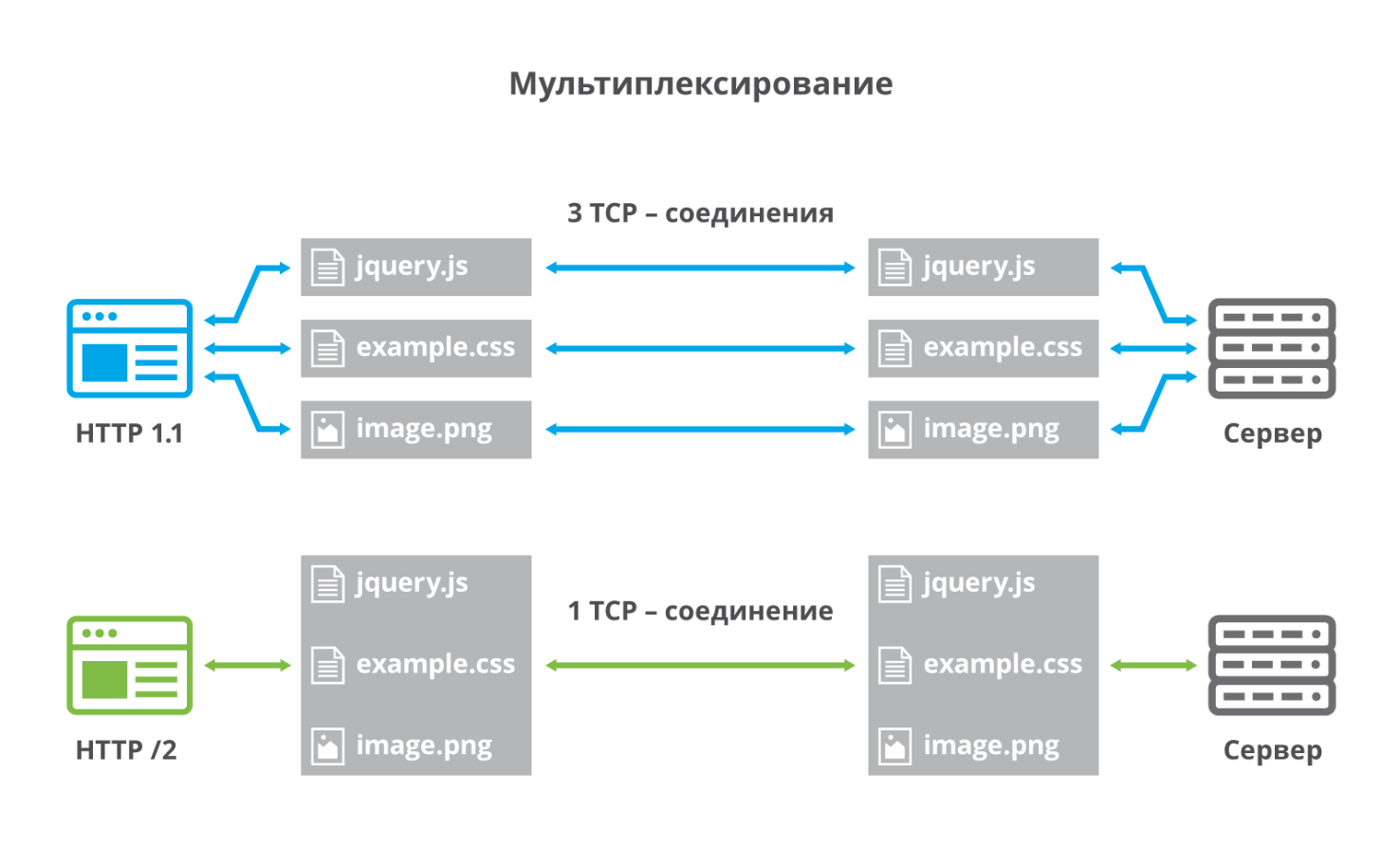
**Практическая работа № 10**

**Введение**

Развитие протокола HTTP может и не кажется стремительным, но оно продолжается. Так в 2014 был представлен HTTP/2. Его основная идея состоит в том, что протокол теперь работает с бинарным данным. А это значит, что отправляемы данные можно упаковывать гораздо более компактно. Также в HTTP/2, в отличие от HTTP/1.1 для каждого запроса не нужно устанавливать отдельно TCP соединение (это называется мультиплексирование). Есть и множество других нововведений, таких как, например, сжатие заголовков запроса, однако они уже не так важны.



**Теория**

1. **API RPC**

Как предшественник REST, RPC (удаленный вызов процедур) представляет собой программную архитектуру, восходящую к 1970-м годам. RPC позволяет вызывать функцию на удаленном сервере в определенном формате и получать ответ в том же формате. Не имеет значения, какой формат использует сервер, выполняющий запрос, и не имеет значения, локальный это сервер или удаленный. RPC позволяет вызывать функцию на сервере и получать результат в том же формате.

Основная концепция RPC API аналогична концепции REST API. RPC API определяет правила взаимодействия и методы, которые клиент может использовать для взаимодействия с ним. Клиенты отправляют вызовы, которые используют «аргументы» для вызова этих методов. Однако в случае RPC API метод находится в URL-адресе. Аргументы, вызывающие методы, находятся в строке запроса. Чтобы проиллюстрировать это, вот как запрос RPC API сравнивается с запросом REST API:

* RPC: запрос RPC API может использовать POST / deleteSmth и иметь строку запроса, которая говорит {"id": 777}.
* REST: запрос REST API записывает этот запрос как DELETE / smth / 777.

1. **API RPC**

Как вариант архитектуры RPC, gRPC был создан Google для ускорения передачи данных между микросервисами и другими системами, которым необходимо взаимодействовать друг с другом. По сравнению с REST API, gRPC API уникальны в следующих отношениях:

* Protobuf вместо JSON
* Построен на HTTP/2 вместо HTTP/1.1
* Создание собственного кода вместо использования сторонних инструментов, таких как Swagger
* Передача сообщений в 7-10 раз быстрее
* Более долгая имплементация и реализация, чем REST

1. **Protobuf**

И REST API, и RPC API отправляют и получают сообщения с использованием форматов обмена сообщениями JSON или XML. Они также могут использовать другие форматы, но наиболее распространены JSON и XML. Проблема в том, что для определенных случаев использования JSON недостаточно быстр или легковесен при передаче данных между системами.

В отличие от REST и RPC, gRPC преодолевает проблемы, связанные со скоростью и весом, и предлагает большую эффективность при передаче сообщений, используя формат обмена сообщениями Protobuf. Вот несколько подробностей о Protobuf:

* Независимость от платформы и языка
* Сериализует и десериализует структурированные данные для передачи в двоичном формате.
* Поскольку он является сильно сжатым форматом, он не обеспечивает удобочитаемости JSON.
* Ускоряет передачу данных, удаляя множество обязанностей, которыми управляет JSON, поэтому он может сосредоточиться исключительно на сериализации и десериализации данных.
* Передача данных происходит быстрее, потому что Protobuf уменьшает размер сообщений и служит легким форматом обмена сообщениями.
* Формат сообщений жестко задан proto файлами как со стороны сервера, так и со стороны клиента.

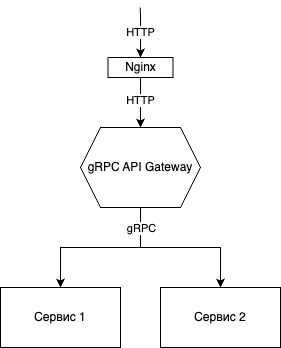
1. **gRPC API Gateway**

Однако, сегодня еще нельзя создать общедоступный сервис, который будет общаться с пользователями, используя HTTP/2 и gRPC. Чтобы решить эту проблему мы создадим API Gateway, который будет принимать запросы от пользователей в виде стандартных HTTP/1 запросов и дальше роутить их на нужные внутренние микросервисы с использованием gRPC и Protobuf.

**Задание**

Вам предлагается реализовать небольшую микросервисную архитектуру с использованием gRPC и API Gateway. Чтобы немного упростить задачу и не реализовывать сами сервисы, предлагается использовать сервисы из прошлых практик (как на Go, так и на Java), правда их необходимо будет перевести на работу с gRPC. Помимо API Gateway должно быть два и более микросервисов.

У вас получится следующая архитектура:



Обязательно соблюдать [Go Project Layout](https://github.com/golang-standards/project-layout).

Конфигурация сервисов должна задаваться с помощью ENV файла (порт, имя cookie и т.д.).

Не стоит забывать и про логирование ошибок, они должны писаться в поток стандартного вывода и в файл, который можно будет прочитать извне контейнера. Для каждого сервиса будет свой отдельный файл логов.

Приветствуется использование различных библиотек.

**Полезные ссылки**

1. [Go Tour](https://go.dev/tour/welcome/1)
2. [Go Project Layout](https://github.com/golang-standards/project-layout)
3. [Курс на Stepik по Go](https://stepik.org/course/54403/info)
4. [Популярные логгеры в Go](https://blog.logrocket.com/5-structured-logging-packages-for-go/)
5. [Библиотека для работы с unit-тестами](https://github.com/stretchr/testify)
6. [Стандартная библиотека net/http](https://pkg.go.dev/net/http)
7. [Статья про стандартные методы работы с HTTP запросами](https://www.alexedwards.net/blog/an-introduction-to-handlers-and-servemuxes-in-go)
8. [Статья про HTTP Middleware](https://www.alexedwards.net/blog/making-and-using-middleware)
9. [Официальный сайт gRPC](https://grpc.io/)
10. [Пример работы с gRPC от Google](https://github.com/GoogleCloudPlatform/golang-samples/tree/main/endpoints/getting-started-grpc)
11. [Простой пример создания gRPC API Gateway](https://blog.logrocket.com/guide-to-grpc-gateway/)
12. [Более сложный пример создания API Gateway](https://habr.com/ru/articles/496574/)

**Вопросы**

1. Какое главное отличие HTTP/2?
2. Расскажите, как использовать gRPC в Go?
3. Как передавать массивы с помощью gRPC?
4. Зачем нужен API Gateway в микросервисной архитектуре?
5. Как передавать опциональные поля по gRPC?