Some calls to APIs can change things on the server, while others return data without changing anything. It is important to distinguish between these different types of calls when testing or designing an API.

1. Метод HTTP является **безопасным** (**safe**), если он не меняет состояние сервера. Другими словами, безопасный метод проводит операции "только чтение" (read-only). **GET**
2. **Не безопасный** тот который меняет состояние на сервере. Например загрузка файла на сервер
3. **Идемпотентный** - если повторный идентичный запрос, сделанный один или несколько раз подряд, имеет один и тот же эффект, не изменяющий состояние сервера*.* Например после удаления файла с сервера, повторное удаление этого файла не поменяет его состояние, так как файл уже был удален. Если мы редактируем запись, и вводим те же данные что и в первый раз, то состояние сервера не поменяется.

Другими словами, идемпотентный метод не должен иметь никаких побочных эффектов (side-effects), кроме сбора статистики или подобных операций. Корректно реализованные методы GET, PATCH, PUT и DELETE идемпотентны, но не метод POST (например при создании 1 и той же сущности у нее могут быть разные id).

***Например создание заказа***

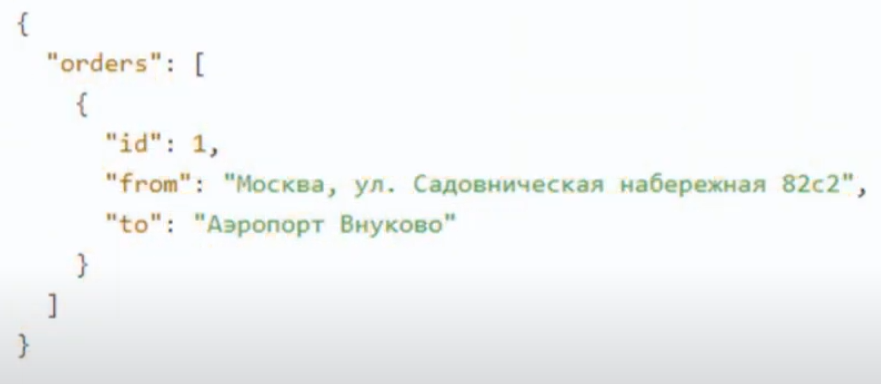
* При повторном вызове идемпотентного метода API создание заказа, заказ не будет создаваться повторно, но API может ответить как 200, так и 400. При обоих кодах ответа API будет идемпотентно, с точки зрения состояния сервера. Но это будет разное поведение с точки зрения клиента.



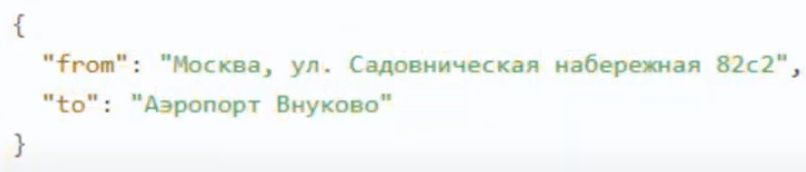
**Допустим есть приложение вызова такси**

**Глава 1 – про API**

* При запуски приложения отправляется запрос getOrders(), который посылает GET запрос на получение активного заказа. Если такой есть, то рисуем на UI



* Что бы создать заказ отправляем запрос postOrders() с введенными пользовательскими данными



* При возникновении серверной или сетевой ошибки рисуем сообщение об ошибки

**Глава 2 – блокирование кнопки**

У ряда пользователе возникла проблема, что к ним приехало 2 машины такси, и деньги списали за обе машины

Проблема была в том, что пользователи два раза подряд нажали вызвать такси, так как на стороне БД запросы по каким то причинам стали медленными, из за этого запросы на вызова такси стали тормозить

**Глава 3 – в подземном переходе**

Один из пользователей пытался заказать такси находясь в подземном переходе, скорость интернета была плохой и при нажатии кнопки заказа запросы был отправлен на сервер, но ответ не был получен. Приложение оповестило что произошла ошибка и разблакировало кнопку заказа

Решение было такое:

Перед созданием заказа в БД делается

SELECT \*

FROM orders AS a INNER JOIN users AS b

ON a.id = b.order\_id

WHERE b.user\_id = нужный нам пользователь AND a.from = ‘’ AND a.to = ‘’ AND время за последние 5 минут

Если такой заказ найден, то сервер дает ошибку 500

Но при таком варианте решения есть гонка между SELECT и INSERT в базу, при параллельных запросах одного и того же пользователя.

Выводы были такие: могут быть проблемы с сетью, БД может тормозить увеличивая окно гонки

**Глава 4 – лимиты на число активных заказов**

Обойти гонку помог такой алгоритм

1. Начинаем транзакцию
2. Делаем UPDATE таблицы заказов UPDATE active\_orders SET n=1 WHERE user\_id={user\_id} AND n=0; Если UPDATE изменил 0 записей, то отдаем 409 ошибку, и далее вставить объект заказов в другую таблицу
3. Совершить транзакцию

**Глава 5 – мультизаказ (ключ идемпотентности)**

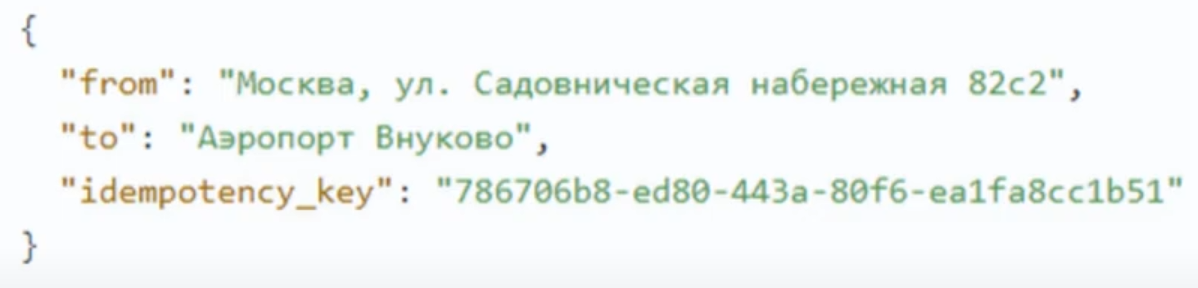
Сделать фичу, что бы юзер смог заказать 2 машины такси

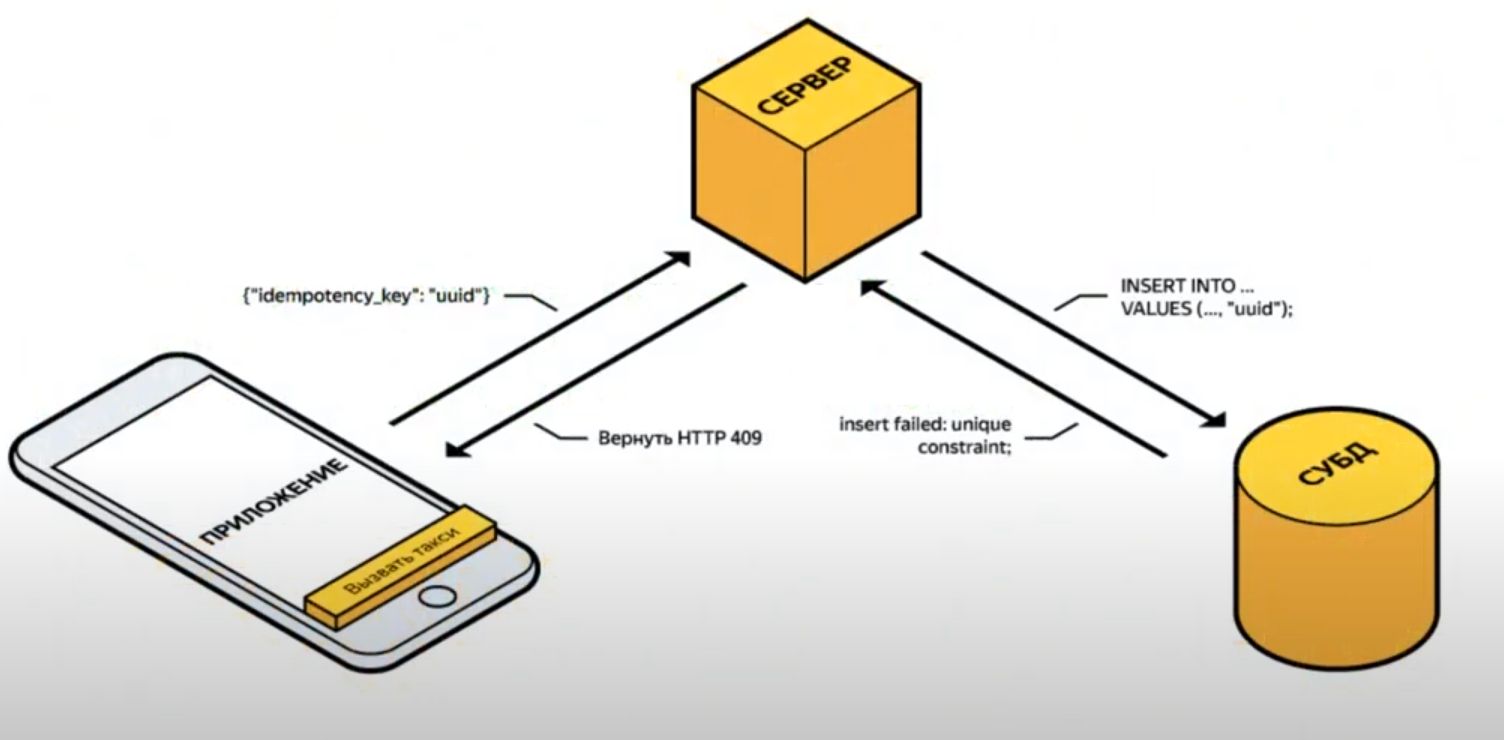
Так как ранее мы сделали защиту от дублей путем проверки есть ли у юзера активный заказ, то эта новая фича полностью ломает логику ее работы

На помощь приходит ключ идемпотентности

Вместе в неидемпотентным POST запросов мы теперьклиент шлет idempotency\_key с уникальным ключом uuid4. Ключи хранятся в течении 24 часов.

postOrders()





При повторной попытке создания заказа, клиент шлет тот же кдюч идемпотентности. На сервере ключ идемпотентности INSERT в БД, на которой есть ограничения на уникальность. Если запрос не был уникальным (то есть такой заказ уже создали), то возвращает 409 ошибку. Но можно упросить, и отдавать не 409, а 200, что бы не писать код на обработку клиентом 409 ошибки

**Глава 6 – баг при тестировании**

1. Пользователь создает заказ и запрос приходит на сервер, заказ создается
2. Пользователь обрывает интернет и ответ не получает, пользователь видит сообщение об ошибке
3. Пользователь меняет точку назначения
4. Пользователь включает интернет и создает заказ повторно

Actual Result: приложение не меняет ключ идемпотентности между запросами. Сервер обнаруживает что заказ с таким ключом уже есть и отжает код 200. На сервере создан заказ со старой точкой назначения, а клиент думает что заказ создан с новой точкой назначения

Решения:

1. Генерировать новый ключ идемпотентности (плохое решение, так как может создаться дубль)
2. Приложение не дает изменить параметры заказа и бесконечно пытается создать заказ, пока получает коды ответа 5xx или сетевые ошибки (выбранное решение)

**Глава 7 – полезное код ревью**

У выбранного решения нашли 2 проблемы:

***2 такси:***

1. Приложение отправляет запрос на создание заказа, запрос выполняется долго по каким то причинам
2. Пользователь видит что такси не заказывается, по этому он перезаходит в приложение
3. При входе в приложение, оно делает запрос getOrders() и создающегося заказа не получает, так как оно еще не создалось
4. Пользователь создает заказ повторно и он создается быстро, и к тому времени первый запрос на создание так же отработал

Actual Result: к пользователю приезжает 2 такси

***Приехало отмененное такси:***

1. Приложение отправляет запрос на создание заказа, но из за ошибки в сети пользователь не получает оповещение об успешном создании заказа
2. Пользователь удаляет заказ (реализовано как удаление строки из БД)
3. Приложение шлет повторный запрос на создание заказа, запрос успешно выполняется и создается заказ, так как ключ идепмотентности, который хранился до этого в БД больше не существует

Как решить эти проблемы:

1. Приложение хранит все создающиеся в данный момент заказы, даже между рестартами приложения. Приложение показывает их на UI, при этом продолжая попытки на создание заказа
2. Перейти с удаления записей таблицы заказов, на выставление поля deleted\_at = Null (так называемый soft delete) – тем самым проверка на ключ идемпотентности работало бы даже после удаления заказа
3. Отделить абстракцию обеспечения идемпотентности запросов от абстракции ресурсов и хранить использованные ключи идемпотентности ограниченное время отдельно от ресурсов и хранить 24 часа
4. Версионировать состояние списка заказов.
   1. **API GET /v1/orders** отдавало бы версию списка заказов. Это версия всего списка заказов пользователя, а не конкретного заказа.
   2. При создании заказа приложение передает в отдельном поле или заголовке If-Match версию, о которой он знает.
   3. Сервер атомарно с изменением увеличивает версию при любых изменениях заказов (создание, отмена, редактирование). То есть приложение в запросе к серверу говорит ему, какое состояние заказов оно знает. И если это состояние заказов (версия) расходится с тем, что хранится на сервере, то сервер отдает ошибку «заказы были изменены параллельно, перезагрузите информацию о заказах».

Версионирование решает обе найденные проблемы. Также стоит отметить, что версия может быть как числом (номером последнего изменения), так и хэшом от списка заказов: так, например, работает параметр fingerprint в Google Cloud API для изменения тегов инстансов.

**Глава 8 – выводы**

Любой API создания ресурсов обязательно должен быть идемпотентным. Кроме того важно синхронизировать знание о списке ресурсов на клиенте и сервере через версионирование этого списка.

**Глава 9 – Идемпотентность удаления**

В один день в телеграм приходит нотификация о том, что в API был код ответа 404. По логам нашли, что это случилось в API отмены заказа.

Отмена заказа делалась через запрос DELETE /v1/orders/:id. Внутри строка с заказом просто удалялась. В soft delete (выставление deleted\_at=now()) необходимости не было.

В данной ситуации приложение послало первый запрос на отмену, но он стаймаутил. Приложение, не уведомляя пользователя, сразу сделало перезапрос и получило 404: первый запрос уже выполнился и удалил заказ. Пользователь же увидел сообщение «неизвестная ошибка сервера».

***Оказывается, идемпотентным должны быть не только API создания, но и удаления ресурсов***

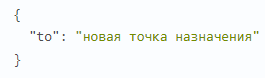
Вариант отдавать 200 всегда, даже если DELETE запрос в базе не удалил ничего. Но это создавало риск скрыть и пропустить возможные проблемы. Поэтому решили сделать soft delete и переделать API отмены:

1. из базы данных стали селектить все, даже уже отмененные заказы с данным id;
2. если заказ уже был удален, и это было в пределах последних n минут (то есть, на обычных перезапросах), то сервер стал отдавать 200;
3. в остальных случаях сервер отдает 410 с ошибкой «заказа не существует». Так же заменили 404 на 410 как более подходящий, так как код 404 означает, что это ошибка временная, а запрос можно потом повторить. Код 410 же означает, что ошибка постоянная, и повтор запроса выдаст тот же результат.

Больше подобных проблем с отменой заказа не всплывало.

**Глава 10 – Идемпотентность изменения**

В приложении пассажир может изменить точку B. При этом посылается запрос **PATCH /v1/orders/:id:**



Сервер же внутри просто выполняет UPDATE в базу:

UPDATE orders SET to={to} WHERE id={id}

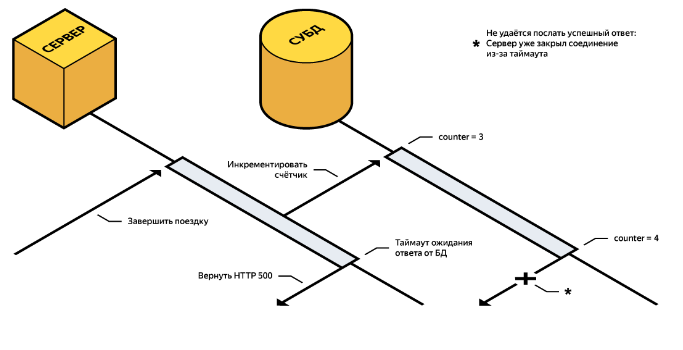
Тут все идемпотентнее некуда. Только есть момент, что при параллельном изменении и чтении/изменении тут могут быть гонки, но это уже совсем другая история.

**Глава 11 – А надо ли фиксить**

Проверил API завершения поездки: оно вызывается водительским приложением, когда водитель выполнил заказ. На сервере API помечает заказ выполненным и делает ряд действий, в том числе подсчет статистики. Среди считаемой статистики взгляд упал на метрику кол-ва завершенных заказов у пользователя. При вызове API счетчик завершенных заказов инкрементился запросом вида

UPDATE user\_counters SET orders\_finished = {orders\_finished+1} WHERE user\_id={user\_id}

Стало понятно, что при повторных вызовах API счетчик может увеличиться больше, чем на 1



Зачем вообще нужен счетчик, если можно каждый раз по базе считать общее число таких заказов?

1. во-первых, старые заказы уезжают в отдельные хранилища
2. во-вторых, счетчик используется в нагруженных API, где важно не делать лишние запросы в базу.

Создали задачу в таск-трекере на переделку расчета счетчика по следующему алгоритму:

1. при создании заказа счетчик никак не меняется;
2. в очереди заданий появляется новая процедура, которая фетчит все заказы пользователя из обоих хранилищ, рассчитывает метрику завершенных заказов и сохраняет ее в базу;
3. задание в очередь кладется из API завершения заказа: при повторных вызовах API в худшем случае несколько раз выполнится задание в очереди, что нестрашно.

Но зачем это делать? Редкое расхождение счетчиков приемлемо. И переделывать схему для точного подсчета метрики нецелесообразно для бизнеса на данном этапе.

**Заключение**

Важно, чтобы API были идемпотентными

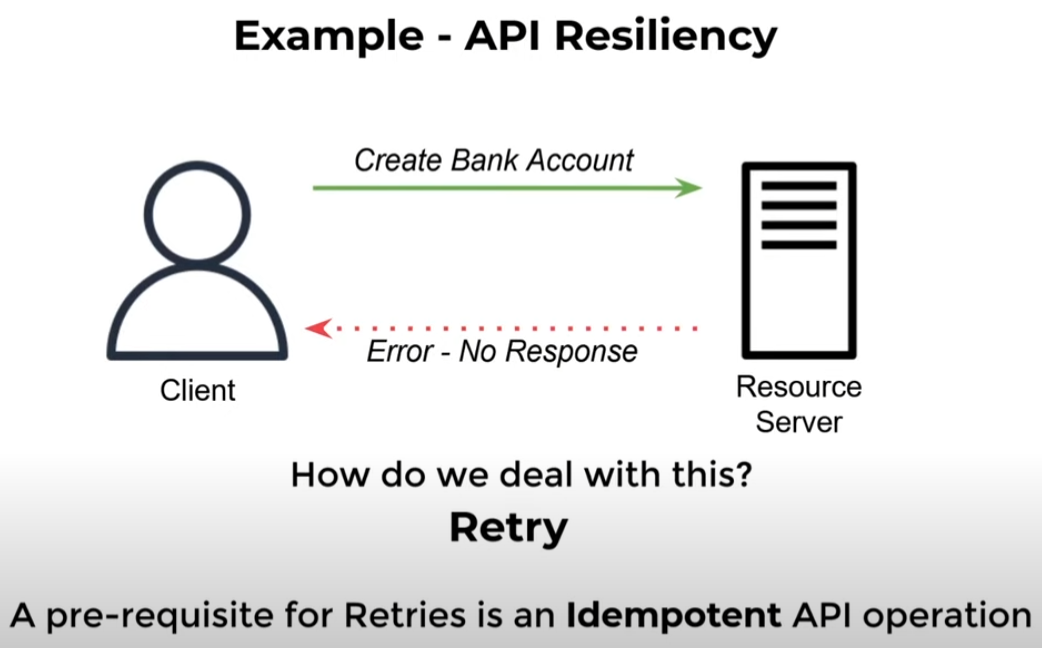
В Яндекс.Такси мы всегда думаем об идемпотентности наших API. В небольшом проекте допустимо было бы не тратить время на проработку редких случаев. Но Яндекс.Такси это десятки миллионов поездок ежемесячно. Поэтому у нас есть процедура дизайн-ревью архитектуры и API. Если что-то неидемпотентно, есть гонки, либо логические проблемы, то API не пройдет ревью. Для разработчиков это означает, что приходится внимательно относиться к деталям и продумывать множество граничных случаев. Это нетривиальная задача, и особенно сложно покрывать такие граничные случаи автотестами.

Случаются ли таймауты, перезапросы, дубли, когда у приложения нет миллионов пользователей? К сожалению, да. Описанные ситуация типичны для распределенных систем: сетевые ошибки происходят регулярно, железо регулярно выходит из строя и т. д. Хорошо спроектированная система считает подобные ошибки нормальным поведением и умеет их компенсировать.

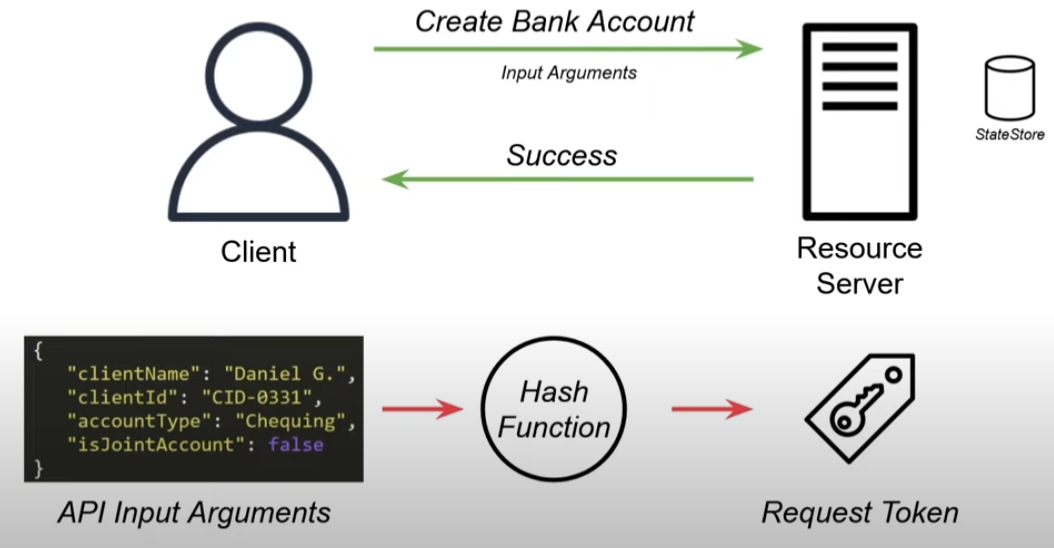
**Пример 2**

Допустим юзер хочет открыть счет в банке. Он отправляет запрос на создание счета, и по каким то причинам не получает ответа (оборвалась сеть, сервер был загружен, …)

Решение простое, нужно послать запрос повторно. Но что если первый запрос на создание счета завершился успешно? И при повторном запросе на создание счета не создастся еще 1 такой же счет



**Решение 1 – наивное**

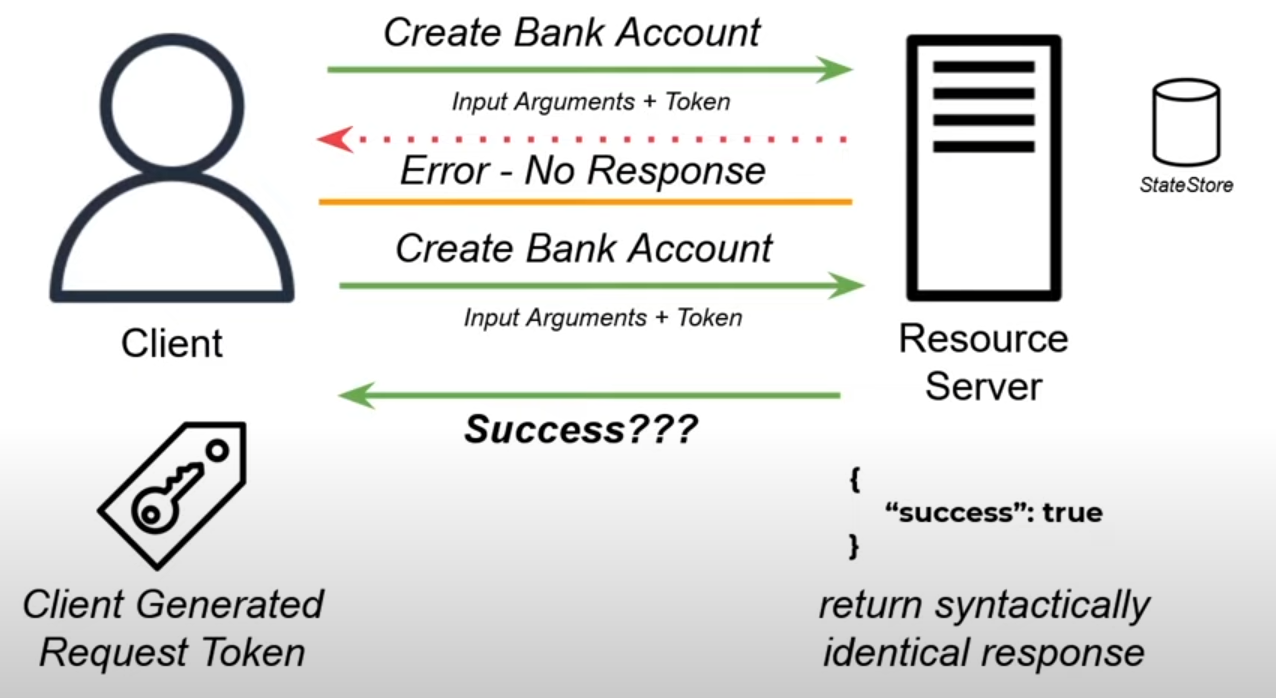


Все данные для создания счета будут преобразовываться hash функцией и это будет уникальным ключом. Этот токен потом хранится в БД. И если юзер отправит запрос у которого будет такой же hash, в течении N времени, то отклонить этот запрос

Но что если пользователь ХОТЕЛ создать 2 одинаковых счета? У него это не получится

**Решение 2 – подходящее**

Заместо того что бы сервер хэшировал данные запроса пользователя и сверял их, мы возлагаем эту ответственность на пользователя



Теперь при каждом запросе пользователь в тело запроса будет вставлять uuid, который будет является уникальным ключом. Все любые повторные отправки в течении N времени будут осуществляться с этим же ключом, если клиент не получил из за сбоев сети или других проблем ответ.

Юзеру же нужно ВСЕГДА возвращать 200 статус, даже не смотря на то, что он уже слал такой запрос. Ведь если юзер в начале не получил ответа, и при повторном запросе ему вернуть что аккаунт уже создан, его это смутит. По этому мы возвращаем сообщение об успешном создании заказа (такой же ответ, как и при первом запросе пользователя на создание)

Если же ответ был получен, то при повторном вызове API, формируется новый uuid токен и создается новый аккаунт

