GETH-API Migration

Author: Vic Woeltjen
Approvers: @tommychheng @tonytran
Date: 12/01/2018

What is this feature?

Currently, geth-api is a geth node with some custom API attached to simplify common tasks. With services like Infura available, this is potentially surplus to what we need/wish to maintain.

geth-api exposes the following endpoints:

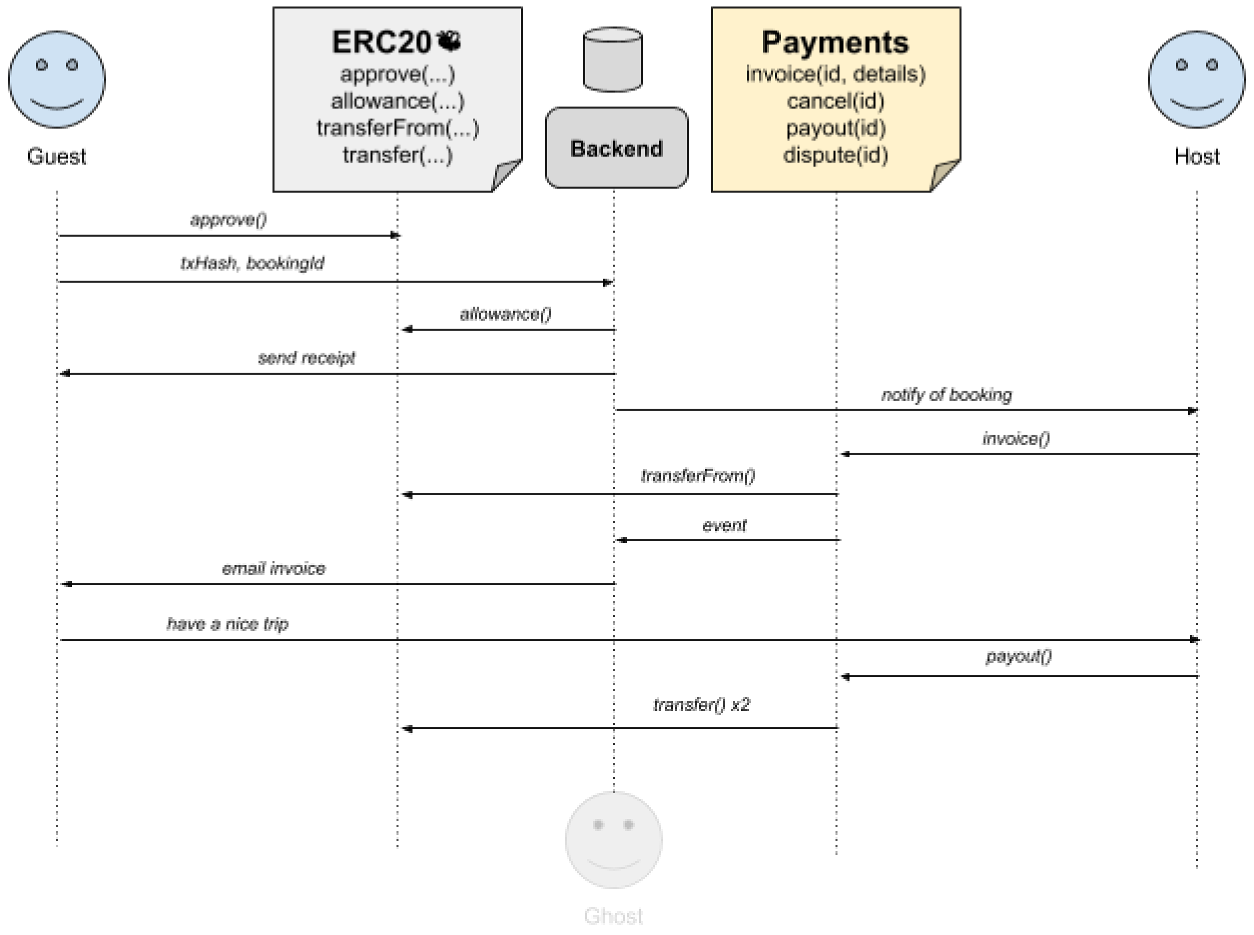
- /init_payment records information related to a forthcoming payment
- /pay moves funds (BEE tokens) from the purchaser to the payment contract
- /cancel_payment returns BEE funds to host/guest and cancels payment
- /dispatch_payment moves BEE funds from the payment contract to host/guest

Analogous endpoints are provided for ETH payments.

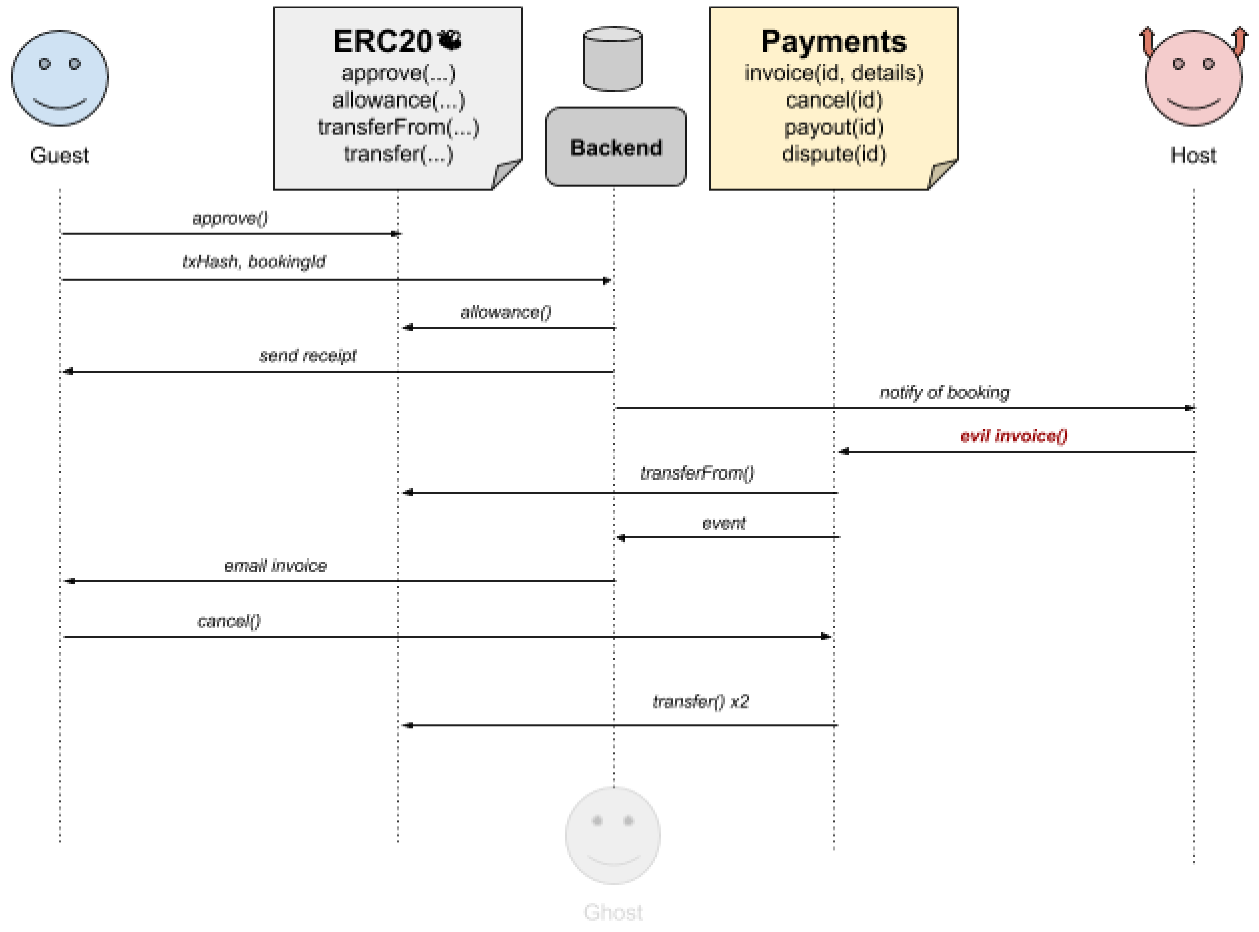
Who will use this?

This is primarily a developer-facing change. However, moving toward more direct interactions with the blockchain improves the experience for users across all categories (guests, hosts, and administrators) *when* users are knowledgeable and invested/interested in the relevant technologies and wish to observe interactions with contracts directly.

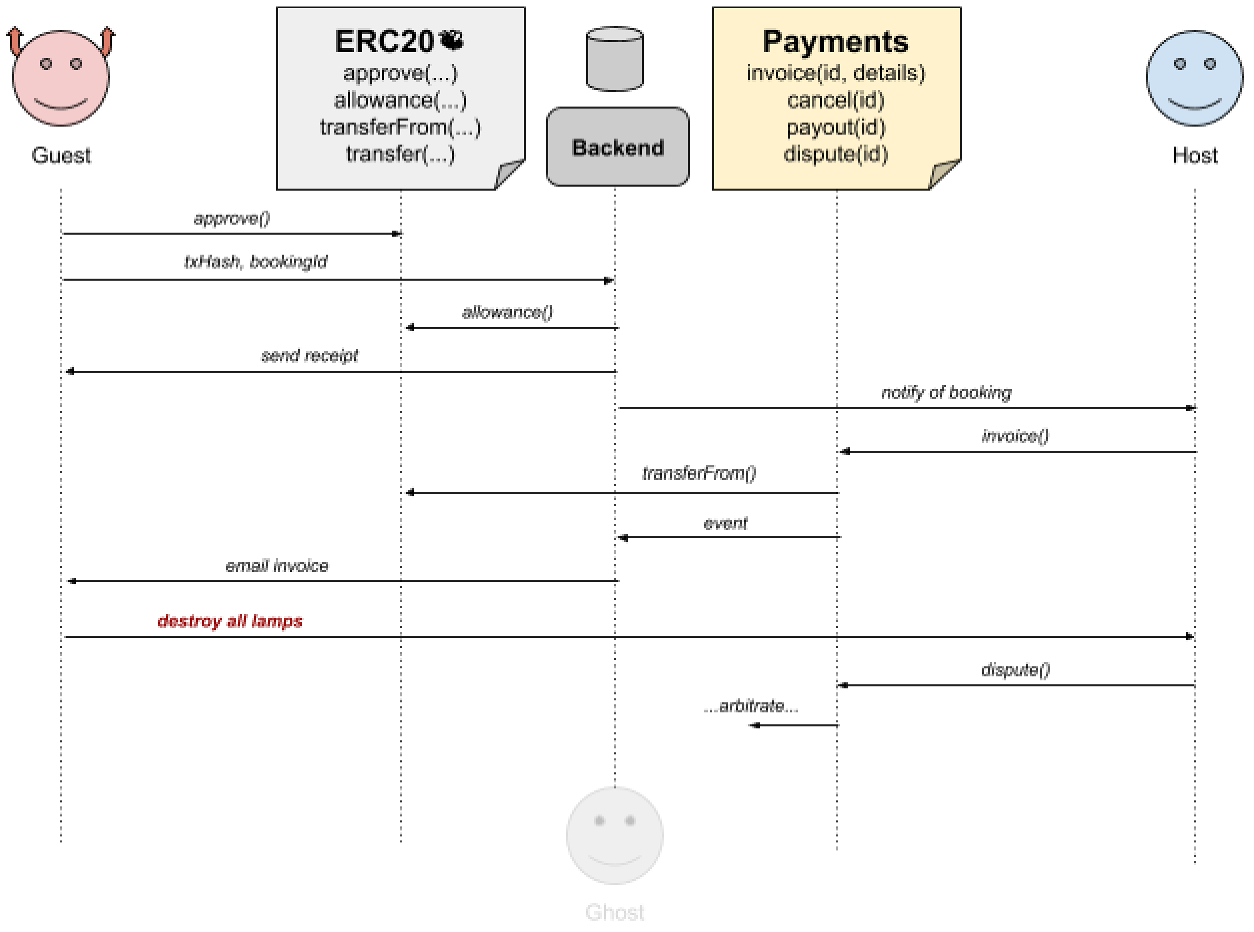
Implementation Overview



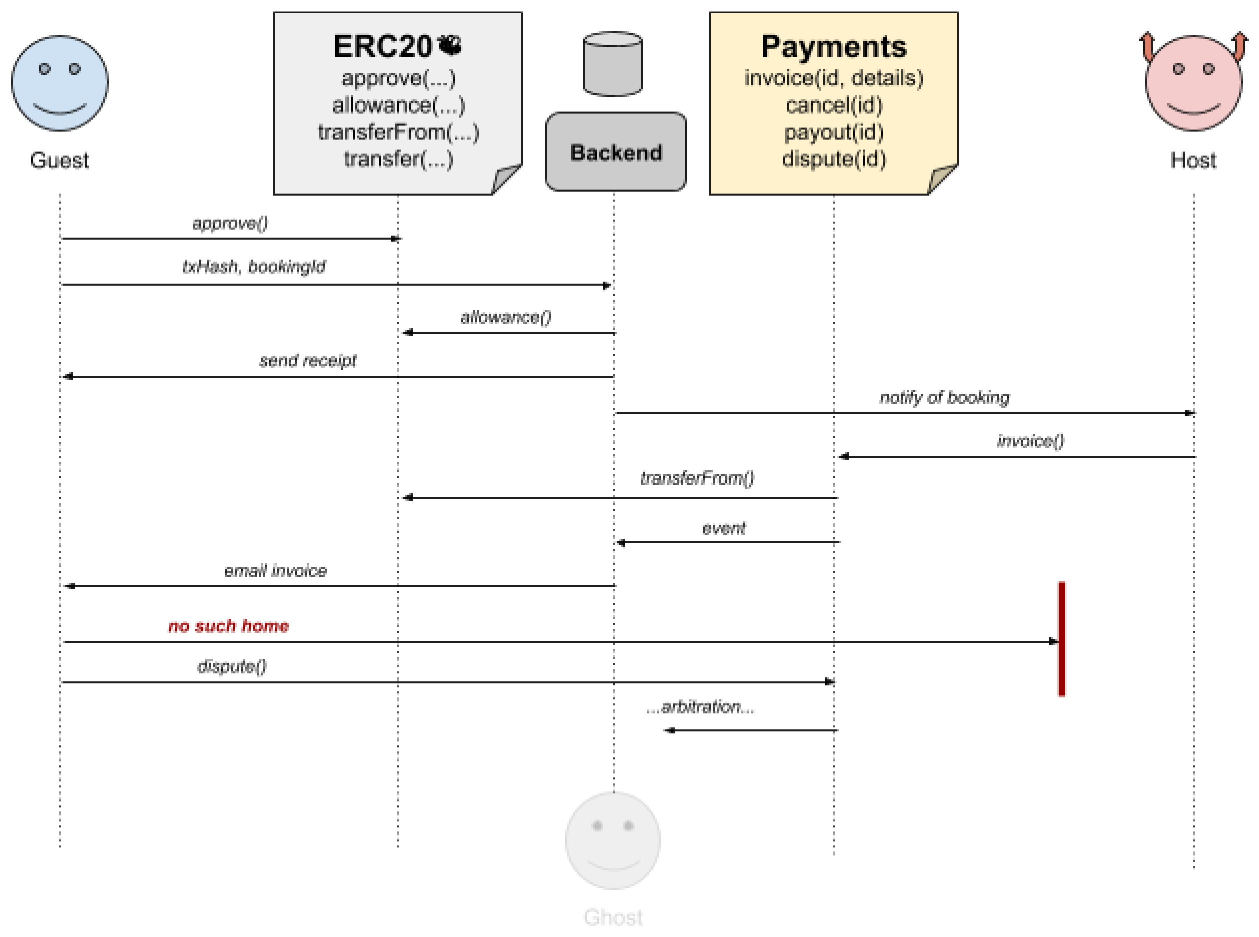
1. Guest buys a booking (e.g. via client), and approves transfer of funds to Payments
2. Guest sends transaction hash and booking info to the backend
3. Backend verifies that funds are approved, provides receipt, and notifies the host
4. Host submits an invoice to Payments with contract-executable details
5. Backend sees event from Payments, notifies Guest of invoice details
6. Guest goes and stays with host; all is fine.
7. Host initiates payout from Payments, which transfers payments/deposit/etc.



1. Guest buys a booking (e.g. via client), and approves transfer of funds to Payments
2. Guest sends transaction hash and booking info to the backend
3. Backend verifies that funds are approved, provides receipt, and notifies the host
4. Host submits a **malicious** invoice to Payments with contract-executable details
5. Backend sees event from Payments, notifies Guest of invoice details
6. Guest sees that invoice details don’t match what Beenest said, cancels payment
7. Payments transfers funds back to host and guest to revert the deal



1. Guest buys a booking (e.g. via client), and approves transfer of funds to Payments
2. Guest sends transaction hash and booking info to the backend
3. Backend verifies that funds are approved, provides receipt, and notifies the host
4. Host submits an invoice to Payments with contract-executable details
5. Backend sees event from Payments, notifies Guest of invoice details
6. Guest goes and stays with host and **breaks a lamp**
7. Host initiates a dispute with Payments to initiate an arbitration flow
8. Arbitration decides what amount of deposit goes to the host



1. Guest buys a booking (e.g. via client), and approves transfer of funds to Payments
2. Guest sends transaction hash and booking info to the backend
3. Backend verifies that funds are approved, provides receipt, and notifies the host
4. Host submits an invoice to Payments with contract-executable details
5. Backend sees event from Payments, notifies Guest of invoice details
6. Guest tries to stay with host, but there is **no such home**
7. Guest initiates a dispute with Payments to initiate an arbitration flow
8. Arbitration decides what amount of booking price to refund to guest

List the security concerns and attack vectors

1. After the approve() call to the Token contract, there is no contract-level verification that the invoice() call looks like what the guest thought they were buying. Mitigations:
    a. *Require a cancellation deadline as part of the booking details.* This creates a period in which a malicious invoice() call can be effectively reverted.
    b. *Provide invoice details from the invoice call to the guest.* This creates an opportunity for the guest to verify the contract-level details.
    c. *Verify invoice details against database on backend.* This creates an opportunity to identify mismatched data and escalate visibility (e.g. notify internal security)
2. After the cancellation deadline is passed, the user cannot cancel upon finding that accommodations will be provided as agreed. Mitigations:
    a. *Allow guest to dispute a payment up until payout.* This provides recourse in the event of fraudulent listings, while protecting hosts from late cancellation.
    b. *Require a dispute deadline as part of the booking details.* Preventing payout until this deadline has passed ensures guests have opportunity to dispute.
3. The public methods of the Payments contract expose opportunities for non-guests/non-hosts to attempt to interfere maliciously. Mitigations:
    a. *Restrict cancel/payout/dispute to host/guest/owner.* This confines interactions with these calls to actors with known incentives and mitigation measures.
    b. *Disincentivize malicious invoicing.* When the cancellation flow works to prevent fraud, denial-of-service and other invoicing attacks just waste gas.
4. Both cancel and payout initiate token transfers from the payments contract to external users. These become potential targets for exploits. Mitigations:
    a. *Protect against re-entrancy.* Use a checks-effects-interaction flow to close invoices by identifier before interacting with other contracts (e.g. by transfer)
    b. *Use trusted contracts.* Accept addresses to token/arbitration contracts from the constructor. This avoids direct calls/transfers to user-provided contracts.
5. Host controls payment details, including deadlines, potentially allowing a host to submit an invoice which circumvents time-based guest protections. Mitigations:
    a. *Enforce minimum time windows.* If the contract defines suitable time windows for cancellation/disputes, the host is limited to invoices which provide those.
    b. *Detect and act on suspicious invoices internally.*

How is this an improvement over the existing system?

In removing the onlyOwner initPayment step of payments, this removes the need for a privileged back-end, allowing deprecation of the geth-api component with the following benefits:

- *Increased transparency.* All parts of the transaction which influence the movement of funds are visible on the blockchain, or remain private to the host/guest; there is no privileged black box to create uncertainty about the way the system works. Remaining back-end components augment this information with useful functionality, but guests and hosts are not strictly dependent on these.
- *Reduced maintenance.* In removing the geth-api component, we avoid future costs associated with maintaining that component.
- *Simplified troubleshooting.* Currently, when errors involving payments occur, the end-to-end flow to examine includes the client, backend, geth-api, and blockchain contracts. By moving payment to client-contract interactions, there are fewer possible points of failure, making it easier to audit flows both proactively and reactively.
- *Clearer decentralization.* Removing geth-api removes the apparent center of control around payments. The Payments contract acts as a common resource to facilitate following a shared payment protocol; authority is moved outward to the actors who select to use the protocol.

Implementation Details

Backend

The backend takes on additional responsibilities, most of which may be implemented as independent serverless jobs:

- Notify hosts that guest requests booking (already implemented, but may be augmented with allowance-checking)
- Listen for Invoice events from the Payments contract
    - Validate against the kind of bookings we expect
    - Notify guests to solicit cancellation (if there is a problem)
    - Provide an endpoint where this information is available on-demand
        - If notification fails, guest can proactively find this information

Data Format

Sketch implementation: https://github.com/woeltjen/gethless/blob/master/Payments.sol

Payments

- Constructor
    - token: address
    - arbitration: address
    - minimumCancelPeriod: uint64
    - minimumDisputePeriod: uint64
- Methods
    - invoice(id: bytes32, ...details)
        - purchaser: address
        - deposit: uint256
        - price: uint256
        - cancellationFee: uint256
        - cancelDeadline: uint64
        - disputeDeadline: uint64
    - cancel(id: bytes32) onlyPurchaserOrOwner beforeCancelDeadline
    - payout(id: bytes32) onlySupplier afterDisputeDeadline
    - dispute(id: bytes32) onlyParticipant beforeDisputeDeadline
- Events
    - Invoice
        - id: bytes32
        - supplier: address
        - purchaser: address
        - deposit: uint256
        - price: uint256
        - cancellationFee: uint256
        - cancelDeadline: uint64
        - disputeDeadline: uint64
    - Cancel
        - id: bytes32
        - supplier: address
        - purchaser: address
        - deposit: uint256
        - price: uint256
    - Payout
        - id: bytes32
        - supplier: address
        - purchaser: address
        - deposit: uint256
        - price: uint256
    - Dispute
        - id: bytes32
        - arbitration: address
        - disputant: address
        - supplier: address

        - purchaser: address
        - deposit: uint256
        - price: uint256

Frontend

Frontend interactions involving payments would need to be upgraded to interact with new contract implementations via web3. Guest-facing flows remain the same (the guest makes an approve() call about BEEs); host-facing flows may continue to be handled through an internal admin workflow until fully-tested, at which point a design and implementation of a host-facing confirmation flow.

Maintenance/Support Details

Backend

This reduces overall maintenance by removing the geth-api component and the geth dependency. New backend responsibilities can, for the most part, be delegated to serverless tasks with minimal component-level dependencies (e.g. on web3)

Frontend

This avoids adding new front-end dependencies. The amount of front-end code to maintain *may* increase around payments, but this should correlate to a larger reduction in back-end code.

Questions

Can initPay and pay be combined?

*Short answer is **yes**; it means we don't have any record of a purchase on-chain until the supplier approves, however.*

What prevents malicious actors from hijacking bookings?

This is disincentivized by the expectation that the guest will cancel

Can the guest calls be limited to just approve?

Yes.

What about cancellation fees et al?

These can be worked in. To disincentivize hijacking, we want a reasonable cancel-without-a-fee period baked in.

Task Breakdown

1. Prototype reimplementation of Payments
    a. Smart contract
    b. UI for testing
2. Verify reimplementation with test UI
3. Deploy new implementation
4. Modify backend logic for payments
    a. No initpay; just keep the bookingId/txHash
5. Implement backend components
    a. Invoice listener (watch, verify, notify)
    b. Other event
6. Integrate new Payments contract into front-end
    a. Update contract hash in settings
    b. Update calls to backend (if necessary)
7. Use new APIs from frontend, for hosts
    a. This is a change to make in admin
    b. May need new design for the payout step

Alternatives

Also briefly looked at issuing a Purchase Order as a one-use contract. This has potential benefits in terms of flexibility, but:

- Gas costs for issuing a new contract of necessary scale look prohibitive. Sketch contract came to about 850k in gas to deploy, which at current gas/eth prices is about 42¢. For comparison, invoice calls to the Payments contract are about 280k in gas.
- For practical purposes (to keep the guest interactions to a simple approve or transfer) we would want a geth node to support these. We liked deprecating that!