ChatGPT

Instructions for Rebuilding the r3nt Contract Suite (Clean-Slate)

This document describes how to build the **r3nt** smart-contract suite from scratch while maintaining the core functionality (booking, deposits, rent handling, investor tokenisation) and addressing shortcomings in the original design. It is aimed at developers who will implement the new contracts using Solidity and OpenZeppelin libraries.

High-Level Goals

- **Unified booking flow:** Avoid duplicated logic across separate contracts. All bookings (short-term and tokenised) should be handled by the same per-property contract.
- **Per-unit booking:** Bookings reserve the **entire unit** (not per square metre). Each property (listing) manages its own calendar.
- **Integrated tokenisation:** Tokenising future rental income should be an optional feature of any booking—no separate `r3nt-SQMU` contract. Investors buy ERC-1155 shares representing claims on future cashflows.
- **Transparent deposit handling:** Deposits are escrowed in the listing contract, with a multi-sig release process and platform confirmation [1] [2].
- **Scalable rent streaming:** Use an accumulator pattern for distributing recurring rent to token holders [3], avoiding loops over all investors.
- **Modular architecture:** Separate concerns into Platform, ListingFactory, Listing clones, BookingRegistry and RentToken, with upgradeability via UUPS.

1. Platform (`Platform.sol`)

This contract owns global parameters and is the authority for creating new listings.

Responsibilities

- Store the USDC token address (assumed 6 decimals) and the platform fee configuration (basis points for tenant and landlord fees).
- Manage listing creation fees and view-pass pricing (if you want to charge for viewing listings).
- Hold the addresses of `ListingFactory`, `BookingRegistry` and `RentToken` so that the UI knows where to interact.
- Only the owner (platform multi-sig) may update global parameters or upgrade contracts via UUPS.
- Provide a function `createListing(address landlord, ListingParams params)` that calls `ListingFactory` to clone a new listing and emits `ListingCreated(listing, landlord)`.

Key State Variables

```
IERC20 public immutable usdc;      // 6-decimal payment token
uint16  public tenantFeeBps;       // fee taken from tenant's rent
```

```
uint16  public landlordFeeBps;     // fee taken from landlord's rent
address public listingFactory;
address public bookingRegistry;
address public rentToken;
// Additional config variables (e.g. viewPassPrice) as needed
```

Key Functions

- `setFees(uint16 tenantBps, uint16 landlordBps)` – update fee bps within a reasonable cap (e.g., <= 2000 bps) [1] .
- `setFactory(address)` / `setBookingRegistry(address)` / `setRentToken(address)` – update module addresses.
- `createListing(address landlord, ListingParams params)` – calls factory to deploy a listing clone and initialise it.
- `_authorizeUpgrade(address newImpl)` – override from UUPS, restricted to owner.

2. ListingFactory (`ListingFactory.sol`)

This minimal proxy factory clones the `Listing` implementation for each property.

Responsibilities

- Maintain the address of the `Listing` implementation.
- Deploy new clones via `createListing(address landlord, ListingParams params)` where `ListingParams` includes deposit amount, daily/weekly/monthly rates, etc.
- Use deterministic salts if you want predictable addresses.
- Emit `ListingCreated(address listing, address landlord)` for indexing.

Key Functions

- `createListing(address landlord, ListingParams calldata params) external returns (address listing)` – clones the implementation and calls `initialize(landlord, platform, bookingRegistry, rentToken, params)` on the new instance.
- `updateImplementation(address newImpl)` – owner-only; upgrade the template used for new listings.

3. BookingRegistry (`BookingRegistry.sol`)

The central calendar that tracks whether a unit is available or booked. Only manages **unit-level** occupancy.

Responsibilities

- Maintain a per-listing mapping of reserved dates (e.g., using a bitmap or mapping) [4] [5] .
- Provide `reserve(address listing, uint64 start, uint64 end)` and `release(address listing, uint64 start, uint64 end)` so that the listing contract can block/unblock dates.
- Enforce that only the listing contract (or Platform for administrative fixes) may call `reserve/release` for a given listing.

- Optionally expose `isAvailable(listing, start, end)` for off-chain checks.

4. RentToken (`RentToken.sol`)

An ERC-1155 token contract representing investor shares in a specific booking. Each booking is assigned a unique `tokenId`.

Responsibilities

- Maintain the URI template for metadata (e.g., `https://api.r3nt.xyz/booking/{id}.json`). The off-chain system generates the JSON files when bookings are created.
- Allow only authorised listing contracts to mint/burn tokens for their booking IDs. Use `AccessControl` with `MINTER_ROLE` granted to each listing.
- Optionally freeze transfers after funding to simplify rent streaming. A boolean `transfersLocked[tokenId]` can be toggled by the listing once all shares are sold.

Key Functions

- `mint(address to, uint256 id, uint256 amount)` – listing only.
- `burn(address from, uint256 id, uint256 amount)` – listing only.
- `lockTransfers(uint256 id)` – listing only; prevents secondary trading after funding.

5. Listing (`Listing.sol`)

This cloneable contract manages the entire lifecycle of a unit's bookings, deposit escrow, tokenisation and rent streaming. It is initialised per property via the factory.

Responsibilities

Booking

- Provide `book(RateType rt, uint256 units, uint64 start, uint64 end)` where `RateType` is an enum (e.g., DAILY, WEEKLY, MONTHLY). `units` is the number of nights/weeks/months. The function:
- Checks availability via `BookingRegistry.reserve(address(this), start, end)` [4].
- Computes `grossRent = units * rate[rt]` (6-decimal USDC). Applies tenant and landlord fees: tenant pays `grossRent + deposit + tenantFee`, landlord receives `grossRent - landlordFee` [1].
- Collects deposit and stores it on the contract's balance.
- Stores booking data in a mapping keyed by `bookingId`.
- Emits `BookingCreated` with tenant, dates, rent and deposit.

Deposit escrow and release

- When a booking is completed, the landlord calls `proposeDepositSplit(uint256 bookingId, uint16 tenantBps)` to suggest how much of the deposit should return to the tenant vs. landlord (e.g., 80% to tenant, 20% withheld).
- The platform (owner) confirms via `confirmDepositSplit(uint256 bookingId, bytes calldata signature)` (or multi-sig), which transfers deposit portions to tenant and landlord and marks the booking as concluded [2].

Tokenisation (optional)

- Landlord or tenant calls `proposeTokenisation(bookingId, uint256 totalShares, uint256 pricePerShare, uint16 feeBps, Period period)` specifying how many shares to issue, the price per share (in USDC) and the periodic frequency (weekly or monthly). A `Period` enum indicates the cadence of future rent distribution.
- The proposal is stored in the booking record with a flag `tokenised = false` until approved.
- The platform calls `approveTokenisation(bookingId)` to finalise the tokenisation parameters. This step could check that `totalShares * pricePerShare` is consistent with the remaining rent or with some discount model.
- Investors call `invest(bookingId, uint256 shares)` and transfer `shares * pricePerShare` USDC to the contract. The contract:
- Transfers USDC minus a platform fee to the proposer (landlord or tenant), enabling them to cash out future rental income [6].
- Mints `shares` of `RentToken` for `tokenId = bookingId` to the investor.
- Increments `booking.soldShares` and when `soldShares == totalShares`, locks transfers of that token ID.
- **Periodic rent distribution:** For each period defined in the booking (e.g., weekly or monthly), the tenant calls `payRent(bookingId, uint256 amount)` with the periodic rent. The contract updates an accumulator `accRentPerShare` as `acc += amount * 1e18 / totalShares` [3]. Each investor's claimable amount is `balance * accRentPerShare - userDebt[investor]`. `claim()` allows investors to withdraw their share and updates their `userDebt`. This pattern avoids iterating over all holders each time rent is paid.
- At the end of the stay, the contract burns all remaining shares for `bookingId` and clears tokenisation state.

Cancellations & defaults

- The platform can cancel a booking before funding (no shares sold) via `cancelBooking(uint256 bookingId)`, refunding rent and deposit to the tenant.
- After funding, if the tenant defaults on rent, the platform may call `handleDefault(uint256 bookingId, uint256 seizedAmount)` to allocate seized deposit or penalties to investors by updating the accumulator.

State Structures

```
enum Status { NONE, ACTIVE, COMPLETED, CANCELLED, DEFAULTED }
enum Period { NONE, WEEK, MONTH }

struct Booking {
    address tenant;
    uint64  start;
    uint64  end;
    uint256 rent;          // total rent (6d)
    uint256 deposit;       // escrowed amount
    Status  status;
    // Tokenisation
    bool    tokenised;
    uint256 totalShares;
    uint256 soldShares;
```

```
    uint256 pricePerShare;
    uint16  feeBps;       // platform fee on raise
    Period  period;
    address proposer;
    uint256 accRentPerShare;              // 1e18 scaled
    mapping(address => uint256) userDebt; // investor checkpoints
}

uint256 public nextBookingId;
mapping(uint256 => Booking) private bookings;
```

Key Functions (non-exhaustive)

```
function initialize(
    address landlord,
    address platform,
    address bookingRegistry,
    address rentToken,
    ListingParams memory params
) external initializer;

// Booking
function book(RateType rt, uint256 units, uint64 start, uint64 end) external;
function proposeDepositSplit(uint256 bookingId, uint16 tenantBps) external;
function confirmDepositSplit(uint256 bookingId, bytes calldata sig) external;
function complete(uint256 bookingId) external;
function cancelBooking(uint256 bookingId) external;

// Tokenisation
function proposeTokenisation(uint256 bookingId, uint256 totalShares, uint256
pricePerShare, uint16 feeBps, Period period) external;
function approveTokenisation(uint256 bookingId) external;
function invest(uint256 bookingId, uint256 shares) external;
function payRent(uint256 bookingId, uint256 amount) external;
function claim(uint256 bookingId) external;
function handleDefault(uint256 bookingId, uint256 seizedAmount) external;

// Getter
function getBooking(uint256 bookingId) external view returns (...);
```

Events

- `BookingCreated(uint256 indexed bookingId, address indexed tenant, uint64 start, uint64 end, uint256 rent, uint256 deposit)`
- `DepositSplitProposed(uint256 indexed bookingId, uint16 tenantBps)`
- `DepositReleased(uint256 indexed bookingId, uint256 tenantAmount, uint256 landlordAmount)`
- `TokenisationProposed(uint256 indexed bookingId, address proposer, uint256 totalShares, uint256 pricePerShare, uint16 feeBps, Period period)`

- `TokenisationApproved(uint256 indexed bookingId)`
- `SharesMinted(uint256 indexed bookingId, address indexed investor, uint256 shares)`
- `RentPaid(uint256 indexed bookingId, uint256 amount, uint256 newAccRentPerShare)`
- `Claimed(uint256 indexed bookingId, address indexed investor, uint256 amount)`
- `BookingCancelled(uint256 indexed bookingId)`
- `BookingCompleted(uint256 indexed bookingId)`

These events enable off-chain indexers (subgraphs) to reconstruct all bookings, investor positions and rent flows.

Design Notes

- **No global arrays** – use mappings keyed by `bookingId` to avoid gas-heavy iteration. Emit events to index bookings.
- **All values in 6 decimals (USDC)** – follow the original contract pattern [1].
- **Multi-sig deposit release** – use the existing multi-sig signature verification from `Listing.sol` (ERC-7913). Offload signature verification into a library or inline it inside Listing [2].
- **UUPS upgradeability** – each module should inherit from `Initializable` and `UUPSUpgradeable`. Only the owner (platform multi-sig) may upgrade implementations.
- **Roles & permissions** – restrict functions based on sender: tenant (only their booking), landlord, platform. Investors can only invest and claim.
- **Integration with off-chain metadata** – the `RentToken` URI should point to JSON that describes the booking (dates, rent amount, landlord, tenant). These files are generated off-chain when a booking is created and uploaded to IPFS or your API.

6. Implementation Plan

1. **Deploy** `RentToken` – grant `MINTER_ROLE` to the `ListingFactory` or `Platform` so clones can mint.
2. **Deploy** `BookingRegistry` – record addresses of listings allowed to reserve dates.
3. **Deploy** `Listing` **implementation** – do not call `initialize` yet. Set up default parameters (rent rates) in `ListingParams` struct.
4. **Deploy** `ListingFactory` – set `listingImplementation` and allow the Platform to call it.
5. **Deploy** `Platform` – set `usdc` address, fees, and addresses for the factory, registry and rent token.
6. **Listing creation** – platform calls `createListing(landlord, params)`, clones Listing and initialises it with addresses. The new listing is now ready for bookings and tokenisation.
7. **Front-end & subgraph** – update UI and subgraph to interact with the unified booking and tokenisation flows.

7. Rationale & Links to Old Implementation

The original `r3nt.sol` handled bookings, fees and deposits [1], while `r3nt-SQMU.sol` handled tokenisation and investor flows separately [6]. This duplication increased maintenance burden and risked inconsistent logic. By merging both flows into `Listing.sol`, we remove the need for two different booking paths and provide a single entry point for tenants, landlords and investors.

The deposit escrow and release logic from the old `Listing.sol` (multi-sig proposals and confirmations) [2] is preserved inside the new `Listing` but now handles the entire booking lifecycle. The calendar functionality from `BookingRegistry.sol` [5] remains a separate module but no longer needs to track square-metre occupancy. Lastly, the rent streaming pattern from `RentDistribution.sol` [3] is integrated into `Listing` to support periodic rent payments to investors without additional contracts.

---

[1] [4] r3nt.sol
https://github.com/NP-Vincent/r3nt/blob/620676b76fae0945a8a25d1441942d5cd686820e/contracts/r3nt.sol

[2] Listing.sol
https://github.com/NP-Vincent/r3nt/blob/620676b76fae0945a8a25d1441942d5cd686820e/contracts/Listing.sol

[3] RentDistribution.sol
https://github.com/NP-Vincent/r3nt/blob/620676b76fae0945a8a25d1441942d5cd686820e/contracts/RentDistribution.sol

[5] BookingRegistry.sol
https://github.com/NP-Vincent/r3nt/blob/620676b76fae0945a8a25d1441942d5cd686820e/contracts/BookingRegistry.sol

[6] r3nt-SQMU.sol
https://github.com/NP-Vincent/r3nt/blob/620676b76fae0945a8a25d1441942d5cd686820e/contracts/r3nt-SQMU.sol