ChatGPT

Integrating Safe{Core} Multisig Escrows into a WordPress dApp

Overview:
This guide outlines how to incorporate Gnosis Safe (Safe{Core}) multi-signature wallet functionality into a WordPress.com-based dApp (SQMU) using only HTML and vanilla JavaScript. We will use MetaMask for user wallet connections and the Safe{Core} SDK (via CDN scripts) for creating and managing Safe wallets as escrow accounts. The plan covers setting up the environment, automatically deploying a Safe wallet when a real estate transaction initiates, configuring flexible signer thresholds (e.g. 2-of-3 or 3-of-5 approvals), tracking the escrow status, collecting signatures from multiple parties, and releasing funds. If direct integration on WordPress.com is not feasible, an alternative lightweight front-end architecture is proposed.

1. Environment Setup on WordPress.com (HTML/JS Only)

WordPress.com supports adding custom HTML/JS in pages via a **Custom HTML** block. We will leverage this to inject the required scripts and our dApp logic:

- **Include Safe{Core} SDK libraries:** Load the Safe Protocol Kit and Safe API Kit via CDN. For example, use Unpkg or jsDelivr to include `@safe-global/protocol-kit` and `@safe-global/api-kit`. These provide the tools for Safe account creation, transaction proposals, and signature collection. The Safe SDK is designed in TypeScript but can be bundled for browser use; it supports standard Ethereum providers (EIP-1193) for signing [1] – which means we can use MetaMask's provider directly in the browser.
- **Include a Web3 provider library:** For convenience in handling MetaMask integration, include a lightweight library like **ethers.js** (via a CDN script tag). This helps create an EIP-1193 provider and signer from MetaMask. (MetaMask injects `window.ethereum` – we can wrap it with ethers for usability.)
- **Script execution in WordPress:** In the HTML block, after including the above scripts, add our custom JS logic inside a `<script>` tag. Ensure the script runs after the libraries load (you may use the script `defer` attribute or place custom script last). Since WordPress.com might restrict inline scripts, you might need to host the JS externally and include it via a `<script src="...">` tag.

Dependency summary: Safe{Core} SDK (Protocol Kit, API Kit), ethers.js (or Web3.js), MetaMask extension (for users). All are purely front-end and can be added to the WordPress page with `<script>` tags. If using ES Module imports, use a `<script type="module">` and import from CDN URLs. This setup avoids any build tools or server components, aligning with WordPress.com's static environment.

2. MetaMask Integration for Wallet Connection

The dApp will use MetaMask as the gateway to users' Ethereum accounts:

1. **Detect MetaMask:** In your script, check for `window.ethereum`. If present, MetaMask is installed. If not, prompt the user to install MetaMask.

2. **Request connection:** Use the EIP-1193 request method to connect accounts. For example:

```
await window.ethereum.request({ method: 'eth_requestAccounts' });
```

This will prompt the user to connect their wallet to the site. After approval, you can fetch `window.ethereum.selectedAddress` or use ethers to get the signer address.

3. **Create a signer object:** Since Safe{Core} can work with any EIP-1193 signer, we can use MetaMask's provider directly [1]. For instance, with ethers:

```
const provider = new ethers.BrowserProvider(window.ethereum);
const signer = await provider.getSigner();
```

This wraps the MetaMask provider in an ethers.js interface and obtains the active account signer. (Alternatively, Safe SDK's `Safe.init()` can accept an EIP-1193 provider directly as `provider` and manage the signer internally [1].)

MetaMask network: Ensure the user is on the correct network (e.g., Ethereum mainnet or testnet) that your dApp expects. You can request network changes via MetaMask if needed. The Safe SDK will require a chain ID or RPC URL for certain operations, which should match the MetaMask network context.

3. Safe Wallet Creation on Transaction Initiation (Escrow Setup)

When a buyer initiates a transaction (for example, pays an expression of interest), the dApp should automatically create a new Safe multisig wallet to serve as the escrow account for that deal. This involves deploying a new Safe smart contract:

1. **Gather participants' addresses:** Determine who the owners of the Safe (escrow) will be. For a real estate escrow, owners might include the buyer, seller, and a neutral escrow agent (or the dApp's admin). Collect their Ethereum addresses.
2. **Define the signature threshold:** Decide how many approvals are required to release funds. For example, in a 3-party escrow (buyer, seller, agent) you might require 2 of 3 signatures (e.g., buyer + seller) to approve release, or possibly all 3 for maximum security. Safe{Core} allows flexible k-of-n configurations by specifying the owners and a threshold value [2].
3. **Example:** For 3 owners, a `threshold: 2` means any 2 owners must sign to execute a transaction [2]. For 5 owners, `threshold: 3` would require three approvals, and so on.
4. **Initialize Safe deployment:** Use the Safe Protocol Kit to create a Safe account configuration and deployment transaction. For example:

```
const safeAccountConfig = {
  owners: [buyerAddress, sellerAddress, escrowAgentAddress],
  threshold: 2  // e.g. 2 of 3 required signatures
};
const safeSdk = await Safe.init({
  provider: window.ethereum,  // MetaMask provider (EIP-1193)
  signer: signer,            // ethers.js signer obtained from MetaMask
  predictedSafe: { safeAccountConfig }
});
const predictedSafeAddress = await safeSdk.getAddress();
```

The SDK can **predict the Safe's address** before deployment [3], which you can use to show the escrow wallet address to users immediately.

5. **Deploy the Safe contract:** Instruct the SDK to create the deployment transaction and execute it. The Protocol Kit's `createSafeDeploymentTransaction()` prepares the contract creation data [4]. This returns a transaction object containing the Safe Factory address, value (usually 0), and data to create the Safe. To deploy:
6. **With MetaMask:** The SDK can use the MetaMask signer to send the transaction. For example, using the Safe SDK's provider or an ethers provider, send the transaction object for confirmation. The Safe docs show getting an `externalSigner` from the SafeProvider to send the tx [5]. In practice, you can call `safeSdk.getSafeProvider().getExternalSigner()` or simply have the signer send the transaction via ethers:

```
const txResponse = await signer.sendTransaction({
  to: deploymentTx.to,
  data: deploymentTx.data,
  value: deploymentTx.value || 0
});
await txResponse.wait();  // wait for Safe deployment confirmation
```

7. On MetaMask, the user will confirm the creation (they pay the gas to deploy the Safe contract). Once mined, the Safe (escrow wallet) is live at the `predictedSafeAddress`.
8. **Post-deployment setup:** After deployment, re-initialize the Safe SDK for the newly deployed Safe address to interact with it going forward. For example, `await safeSdk.connect({ safeAddress: predictedSafeAddress })` to switch the SDK context to the new Safe [6] [7]. This allows querying the Safe's details and preparing escrow transactions.
9. **Fund the Safe (if needed):** If the buyer's initial payment wasn't already sent into the Safe, instruct the buyer to deposit the funds into the new Safe address. This could be a simple transfer transaction from the buyer to the Safe address, or the Safe creation transaction itself could carry an initial ETH value. In a typical flow, the buyer might transfer their deposit to the Safe once it's created (this step can be done via MetaMask as a normal transfer, or via a Safe transaction if the buyer is the sole deployer).

At this point, an **escrow Safe account** exists for the transaction. It has a unique address, a set of owners, and a signature threshold enforcing multi-party control.

4. Displaying Escrow Wallet Status in the Frontend

With the Safe (escrow) created, the dApp should present its status to all parties for transparency:

- **Escrow Wallet Address:** Show the Safe's address (users can independently verify it on Etherscan or a block explorer). This ensures transparency of where funds are held.
- **Signers and Threshold:** Display the list of Safe owners (participants' addresses) and how many signatures are required. The Safe SDK can fetch this data – e.g., `safeSdk.getOwners()` and `safeSdk.getThreshold()` return the array of owner addresses and the threshold number [7]. This confirms, for instance, "Owners: [A, B, C]; Required Approvals: 2 of 3".
- **Escrow Balance:** Fetch the Safe's balance (e.g., in ETH or relevant tokens) using a provider call. For ETH, you can call `provider.getBalance(safeAddress)` periodically to update the UI. This shows if the buyer's deposit has arrived in the Safe.

- **Deployment/confirmation status:** Indicate whether the Safe is fully deployed and funded. The SDK provides an `isSafeDeployed()` check [8] which should return true once the creation transaction is mined. You might use this to handle UI state (e.g., "Escrow Wallet is ready" message).
- **Pending transactions or required actions:** If there are any pending operations (like awaiting deposit or awaiting approvals for release), highlight them. Initially, right after creation, the main pending action might be "waiting for buyer to deposit funds" (if not already done). Later, it could be "waiting for signatures to release funds" once a withdrawal is initiated (covered in the next sections).

Regularly update this information or provide a "Refresh" button, as blockchain state (balance or confirmations) might change with time. The goal is to keep all parties informed about the escrow's status through the UI.

5. Handling Multi-Sig Transactions: Collecting Signatures for Fund Release

When it's time to release the escrow funds (e.g., after all real estate conditions are met), a transaction from the Safe must be executed – typically sending the funds to the seller (or back to buyer if deal is canceled). Safe's multi-signature scheme means this outbound transaction requires approvals from the required number of owners. The dApp should facilitate the proposal and collection of these signatures:

a. Proposing a Transaction to the Safe:
One of the Safe owners (say, the buyer or the designated escrow agent) will initiate the release by proposing a Safe transaction:

1. **Create the transaction details:** Using the Safe Protocol Kit (with the Safe connected to the escrow Safe address), create a transaction object representing the release. For example:

   ```
   const releaseTxData = {
     to: sellerAddress,
     value: ethers.parseEther("X"),  // the amount to release
     data: "0x"  // (empty data for a simple ETH transfer)
   };
   const safeTransaction = await safeSdk.createTransaction({ transactions:
   [releaseTxData] });
   ```

   This generates a Safe transaction object containing the details to transfer X amount to the seller. No funds move yet – it's just a proposal.
2. **Calculate transaction hash and signer's signature:** The Safe transaction has a unique hash (`safeTxHash`) that must be signed by the owners. The initiator signs it first:

   ```
   const safeTxHash = await safeSdk.getTransactionHash(safeTransaction);
   const signature = await safeSdk.signHash(safeTxHash);
   ```

   This cryptographically proves the initiator's approval. The Safe SDK's `signHash` uses the connected MetaMask signer to sign the Safe's hash off-chain.
3. **Propose to Safe Transaction Service:** Send the transaction proposal to Safe's Transaction Service via the API Kit. This makes the pending tx available to other owners for confirmation. For

example:

```
await safeApiKit.proposeTransaction({
  safeAddress: escrowSafeAddress,
  safeTransactionData: safeTransaction.data,
  safeTxHash: safeTxHash,
  senderAddress: initiatorAddress,
  senderSignature: signature.data
});
```

The Safe API Kit will post this to the transaction service, which acts as a coordination backend for multisig approvals [9] [10] . After this call, the transaction is "proposed" – essentially awaiting the remaining owners' signatures.

At this stage, the escrow transaction is **pending** with 1 signature collected (the proposer's). The Safe (on-chain) hasn't executed anything yet, and won't until enough signatures are gathered and someone calls execute.

b. Notifying and Collecting Other Signers' Approvals:
All other Safe owners need to know about the pending transaction and approve it:

1. **Retrieve pending transactions:** Using the Safe API Kit, the app can query for transactions awaiting confirmations on that Safe. For example:

   ```
   const pending = await
   safeApiKit.getPendingTransactions(escrowSafeAddress);
   ```

   This returns a list of pending tx proposals and how many signatures each has. The dApp should filter for the relevant transaction (there might be only one, the release). According to Safe docs, one can retrieve all pending transactions for the Safe [11] .
2. **Display pending transaction to signers:** Show the transaction details in the UI (e.g., "Release X ETH to [seller]"). Include information like how many signatures are required and how many have been collected so far. (The Safe API will indicate current confirmations, or the SDK's `safeTransaction.signatures` map can be used to count signatures [12] .) For example, display "1 of 3 approvals obtained – your approval needed."
3. **Each signer approves via MetaMask:** When a signer (who is an owner of the Safe) views the pending request, allow them to approve it with a click:
4. Ensure they have connected their MetaMask (the app should detect their account and confirm it's one of the Safe owners).
5. Use the Safe Protocol Kit with that signer to **sign the transaction hash**: e.g., `await safeSdk.signHash(safeTxHash)` (with the Safe SDK instance configured for the escrow Safe and using this signer's provider). This produces the signer's signature.
6. Submit the signature to the Safe service via `safeApiKit.confirmTransaction(safeTxHash, signature)` [13] . This adds the signer's approval to the off-chain store of confirmations. No blockchain transaction is sent in this step – it's an off-chain signature submission.
7. Update the UI to reflect the new approval count (for instance, from "1 of 3" to "2 of 3 approvals"). The Safe API can be polled or subscribe to events to detect new confirmations.
8. **Repeat for all required signers:** The transaction will remain pending until the threshold number of unique owners have all submitted signatures. Each signer can independently come to

the app and perform their confirmation. The Safe Transaction Service aggregates these. The dApp should handle scenarios such as one of the required signers not acting – possibly by sending reminders outside the app if necessary.

Once the required number of signatures is reached (e.g., 2 of 3 have signed for a 2-of-3 Safe), the transaction is fully approved and ready to execute. The Safe service knows it has enough signatures, and this state can be detected via the API or by attempting execution.

6. Executing the Escrow Transaction (Fund Release)

After collecting the necessary approvals, the escrow transaction must be **executed on-chain** to actually transfer the funds. Execution can be triggered by any Safe owner (often the original proposer or a designated party) once the threshold is met:

1. **Prepare for execution:** The front-end should enable an "Execute Release" button when enough signatures are present (e.g., display a button to the signer who will execute, or to all owners with a note that only one needs to execute).
2. **Fetch the fully signed transaction:** Using the API Kit, retrieve the transaction including all signatures. For example:

   ```
   const safeTx = await safeApiKit.getTransaction(safeTxHash);
   ```

   This will return the transaction data along with the collected signatures.
3. **Call execute via Protocol Kit:** With the Safe SDK (Protocol Kit) connected as an owner, call the execute function to send the transaction on-chain. For example:

   ```
   const execResponse = await safeSdk.executeTransaction(safeTx);
   ```

   This will prompt MetaMask to confirm a transaction from the Safe. Essentially, one of the owner accounts will post a transaction to the Safe contract's `execTransaction` method with the packed transaction data and signatures. The Safe contract will then verify the signatures and, if valid, perform the action (releasing the funds). The Safe{Core} docs illustrate this step: the first owner fetches the transaction and calls executeTransaction with all signatures collected [14].
4. **MetaMask confirmation:** The executor (owner) will confirm the execution transaction in MetaMask. They will pay the gas fees for this final step (fees can also be paid from the Safe's funds if using a relay/4337 approach, but that is advanced usage – by default an owner pays gas).
5. **Funds transfer on-chain:** Once executed, the Safe contract moves the funds to the target (e.g., seller's address). The escrow is now closed. Update the UI: the Safe's balance will drop (likely to 0 if all funds released) and you can mark the escrow as completed.
6. **Post-execution checks:** The dApp can confirm the execution by listening for the transaction receipt or Safe event. The Safe Transaction Service will also mark the proposal as executed. It's good to display a success message and possibly a transaction hash link for transparency.

At this point, the multi-signature escrow flow is complete: the Safe wallet served its purpose as an escrow that required multiple parties to sign off.

Note: Safe transactions can also be executed through other means (Safe's web interface or CLI) if needed [15], but integrating it directly in the dApp via the SDK as above provides a seamless user experience.

7. Alternative Architecture if Direct Integration is Unfeasible

If WordPress.com's environment proves too restrictive for the above approach (for example, if adding custom scripts is limited or the Safe SDK bundle is too large for a `<script>` block), consider a minimal standalone front-end for the dApp logic:

- **Static Front-End dApp:** Develop the escrow workflow in a lightweight web app (using plain JS or a minimal framework) and host it separately (e.g., on IPFS, GitHub Pages, Netlify). This app can still use the Safe{Core} SDK and MetaMask, but since it's not constrained by WordPress.com, you can use modern bundling (Webpack/Vite) to include the Safe SDK modules. The output can be a single JS file that is optimized and loaded in a basic HTML page. You could then embed this app into the WordPress site via an `<iframe>` or a link (e.g., "Open Escrow DApp"). This separation keeps WordPress for content management while delegating blockchain interactions to a purpose-built client.
- **WordPress Plugin or React App:** If using WordPress.org or if WordPress.com supports it, you could create a custom plugin or use the WordPress REST API with a React front-end. For instance, Safe provides React Hooks for easier integration [16], which could be used in a React-based widget. However, on WordPress.com you likely cannot run a full React build without embedding it. A compromise is to use a small React app bundled as a script and include it as described above.
- **Leverage Safe Transaction Service APIs directly:** In scenarios where adding the entire Safe SDK is impractical, you could interact with Safe's REST APIs (for transaction proposals and confirmations) using fetch/AJAX from your front-end. You would use MetaMask/ethers to create and sign the transaction hash manually, then call the API endpoints to propose/confirm. This is essentially what the Safe SDK does under the hood. While possible, this requires manual implementation of some logic (like encoding transactions and computing safeTxHash according to Safe's contract). It's recommended only if you absolutely cannot use the provided SDK kits.

Why this alternative? WordPress.com might restrict large script inclusion or certain operations. A separate static app gives you full control and can still be "lightweight" – e.g., a few hundred KB bundle. Users would experience it as part of your site (especially if embedded or styled to match). The friction remains low: they just connect MetaMask and use the escrow features, without worrying about the underlying complexity.

8. Tooling and Libraries Recap

To implement the above plan, ensure you have the following tools and libraries ready:

- **MetaMask** – Ethereum wallet extension for user accounts (no direct library needed; interacts via `window.ethereum`).
- **Safe{Core} Protocol Kit** – for Safe account deployment, transaction creation, and execution (available via npm packages or CDN).
- **Safe{Core} API Kit** – for interacting with the Safe Transaction Service (proposing and confirming transactions off-chain) [9] [13].
- **ethers.js (or web3.js)** – to interface with MetaMask and the Ethereum network in the browser. Ethers is used in examples for its simplicity.
- **Network RPC Endpoint** – The Safe SDK might need an RPC URL for certain operations (e.g., when initializing if not using `window.ethereum` directly). Make sure to use the appropriate network (Ethereum mainnet, or a testnet like Sepolia for testing). Safe's transaction service supports multiple networks – ensure the `chainId` in API Kit is set accordingly (e.g., 1 for

mainnet, 11155111 for Sepolia, etc.). You may need to obtain an API key for Safe's transaction service for higher rate limits [17] (Safe docs provide a link for API key instructions).

All these can be integrated in a purely front-end fashion. No backend or database is required, as the Safe contract and Safe transaction service hold the necessary state (the contract holds funds and requires signatures; the service holds pending tx data and signatures off-chain).

9. Step-by-Step Summary

To conclude, here is a high-level step-by-step flow of the implementation:

1. **Setup** – Add Safe SDK and ethers scripts to WordPress. Connect MetaMask and get the current user's signer.
2. **Initiate Escrow** – When a transaction starts (e.g., buyer commits to deal), deploy a new Safe wallet with the buyer, seller, and others as owners (threshold as needed). Use Safe Protocol Kit to deploy the Safe [2]. Display the Safe escrow address and info to users.
3. **Fund Escrow** – Buyer deposits funds into the Safe (if not done during creation). Confirm deposit by checking Safe balance.
4. **Show Escrow Status** – Continuously show escrow wallet details: owners, required signatures, balance, and any pending actions.
5. **Propose Release** – At closing, one party creates a release transaction (to send funds to seller) via Safe SDK and signs it. Propose this to the Safe service with their signature [10] [18].
6. **Collect Signatures** – Notify other parties of the pending release. Each party uses the dApp to fetch the pending tx [11] and submit their signature approval via MetaMask [13]. The dApp updates the count of collected signatures.
7. **Execute Release** – After reaching the required threshold, allow an owner to execute the transaction. Fetch the fully signed tx and call Safe's execute function to perform the on-chain transfer [14]. MetaMask confirms and the funds move out of the Safe to the recipient.
8. **Close Escrow** – Mark the escrow as completed in the UI, possibly disabling further actions on that Safe. The Safe wallet can be left as an archive of the transaction (with history accessible via Safe's interface or blockchain explorers).

By following this plan, you leverage Safe{Core} to provide a secure, multi-signature escrow wallet within a WordPress-hosted dApp, ensuring that real estate transaction funds are only released with the required approvals. This design offers trust-minimization (no single party controls the funds) and transparency, all while using a lightweight front-end approach suitable for WordPress.com.

Sources: The implementation details above are based on the official Safe{Core} documentation and guides for the SDK Starter Kit, Safe Protocol/API Kits, and multisig transaction flows [1] [2] [9] [11] [13] [14]. These resources provide further technical insight and example code for integrating Safe functionality into web applications.

---

[1] [2] [3] [4] [5] [6] [7] [8] Safe Deployment – Safe Docs
https://docs.safe.global/sdk/protocol-kit/guides/safe-deployment

[9] [10] [11] [13] [14] [15] [17] [18] Execute transactions – Safe Docs
https://docs.safe.global/sdk/protocol-kit/guides/execute-transactions

[12] Transaction signatures – Safe Docs
https://docs.safe.global/sdk/protocol-kit/guides/signatures/transactions

16 Starter Kit – Safe Docs
https://docs.safe.global/sdk/starter-kit