# # Gameroom #

Business Through Smart Contracts

**Technical Walkthrough**

for Splinter v0.3.4

October 31, 2019

# Table of Contents

# Introduction

Distributed ledger technologies have the potential to revolutionize how businesses communicate and transact. At Cargill, we are leading the revolution with involvement in several open-source projects, including Hyperledger Sawtooth, Hyperledger Transact, and Splinter. This document outlines an example application, Gameroom, that uses technologies from each of these projects to demonstrate our vision of using smart contracts to enhance business and customer relationships.

Privacy and confidentiality between trading partners are important capabilities for (almost) all multi-party interactions. These capabilities are especially critical in distributed applications. As a result, the technology stack behind Gameroom differs from a common "shared-everything" blockchain design; instead, it sculpts the underlying blockchain-like distributed ledger technology into a sophisticated architecture that shares data only between the appropriate participants.

Parts of this walkthrough are written as a script, in several acts, for which we recommend a dramatic reading (as if performed on stage by very amateur actors). Each act is followed by a "behind the scenes" chapter that describes the complex underlying technology and explains what really happens when running the deceptively simple Gameroom application. A glossary at the end defines the terms used by Splinter and the Gameroom application.

**Note:** This document is based on Splinter version 0.3.4. As Splinter matures, some details may change (such as the exact format of messages).

## The Cast

Alice, an employee at Acme Corporation

Bob, a project leader at Bubba Bakery

Yoda, a VP at Yoyodyne Systems

Zixi, head of IT at Zymogen Industries

## The Setting

Two Splinter nodes are set up, one at Acme Corporation and one at Bubba Bakery. Alice and Bob have each registered as a Gameroom user with an email address, a private key, and a password. The Prequel explains the details of node setup and user registration.

# Prologue: An Initial Conversation

ALICE, sitting in an office chair at a desk. MACBOOK PRO. Aging desk PHONE. Alice picks up the receiver. Dials a phone number.

ALICE

Hi Bob.

Long pause. Alice rolls her eyes.

ALICE

Anyway, we need to get moving on setup -- yeah -- sure, it is a bit silly, but we need to prove how our companies can use this new technology. We'll use the gameroom to play a few games of --

Another pause. Alice sighs.

ALICE

Yes, we have to show that we can create a private and secure connection between our two companies.

How about I send you -- right, I'll create the gameroom and you'll see my invitation in your app.

Alice logs into her Mac. Starts up the BROWSER and clicks on the ACME GAMEROOM bookmark. Browser displays a network error.

ALICE

One second; I got this Mac from IT but it doesn’t have access to the corporate network -- yeah, I know.

Actually, just let me know if you don't get my invitation in a few minutes -- Fine. OK, bye.

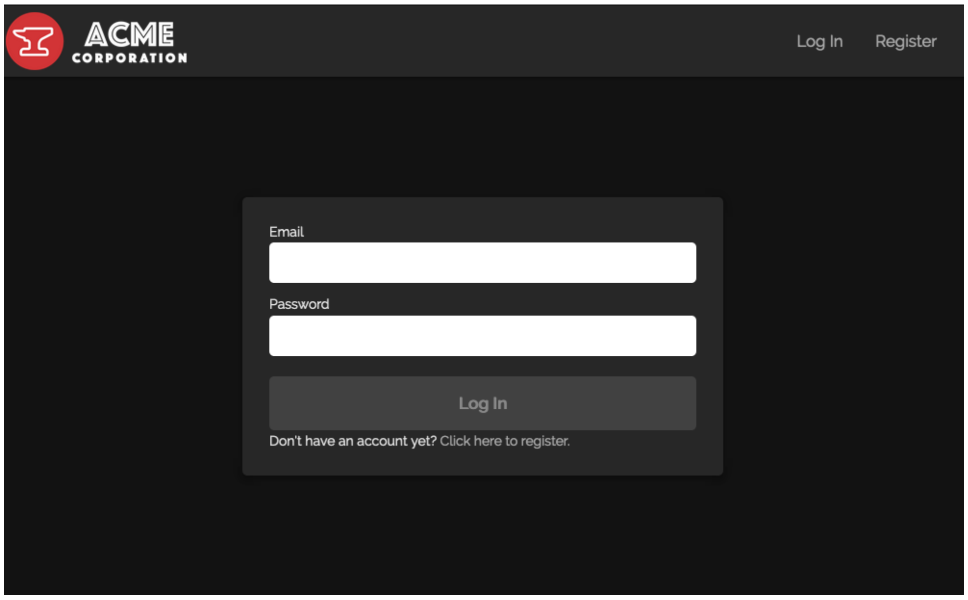
Alice hangs up the phone.

Alice pushes the Macbook aside. Reaches into her bag and heaves out a Windows notebook. It looks old. Alice opens it up. Logs in. Starts up the BROWSER and clicks on the ACME GAMEROOM bookmark. The app starts to load.

# Act Ⅰ: Alice and Bob Create a Gameroom

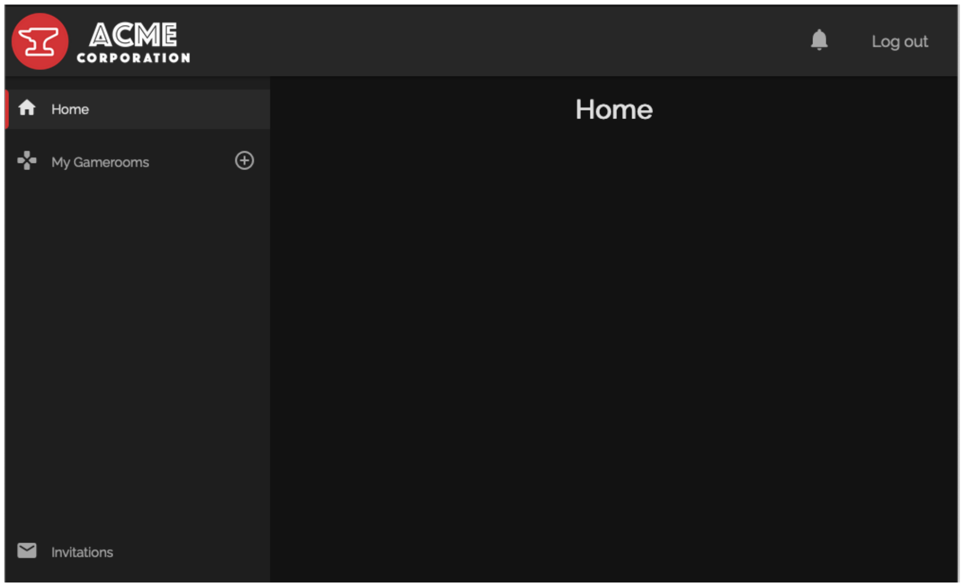
**Scene 1: Alice logs into Acme's Gameroom application**

Alice looks at the GAMEROOM APP LOGIN SCREEN in her browser.



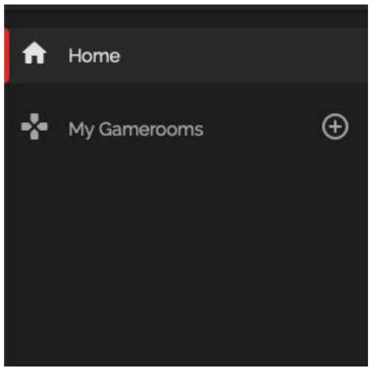
Alice enters her EMAIL and PASSWORD. Clicks LOG IN.

Success. The browser now displays the ACME GAMEROOM HOME SCREEN.

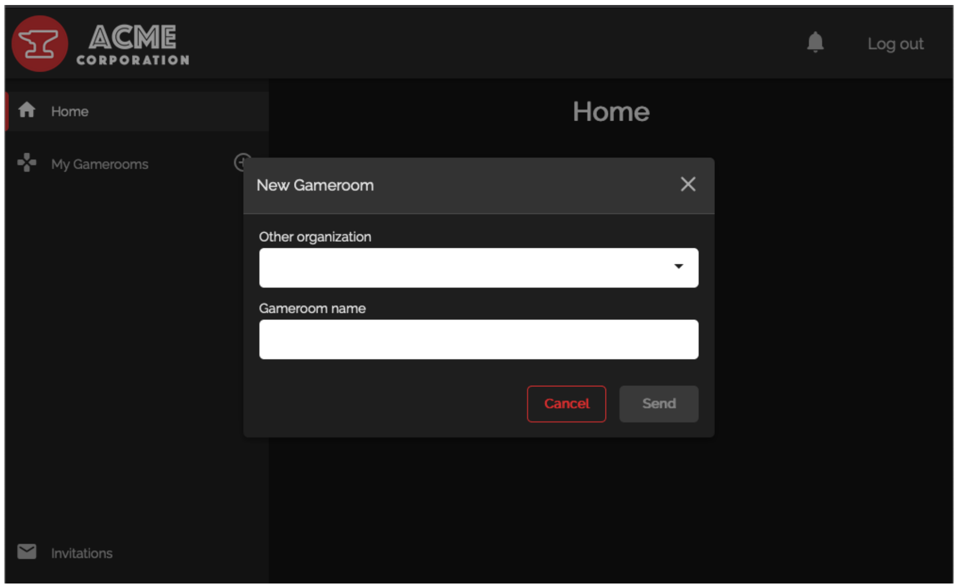

ACME CORPORATION
Log out
Home
My Gamerooms
Invitations
Home

## Scene 2: Alice creates a new gameroom

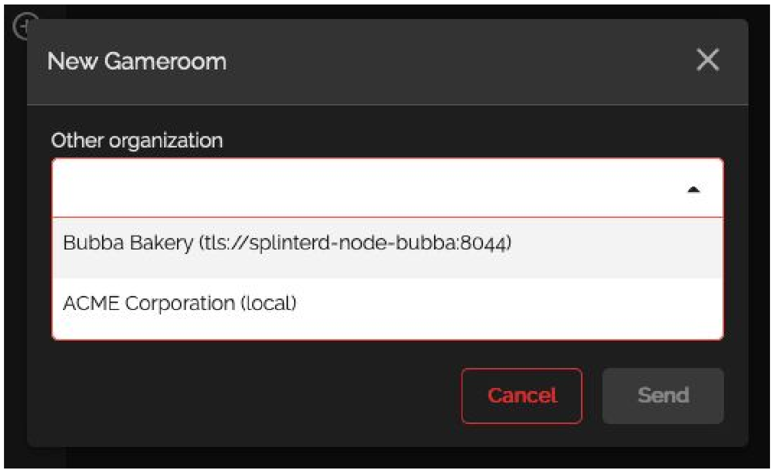
Alice sees an empty MY GAMEROOMS sidebar (no gamerooms exist yet). Alice creates a new gameroom by clicking on the **+** button next to My Gamerooms.



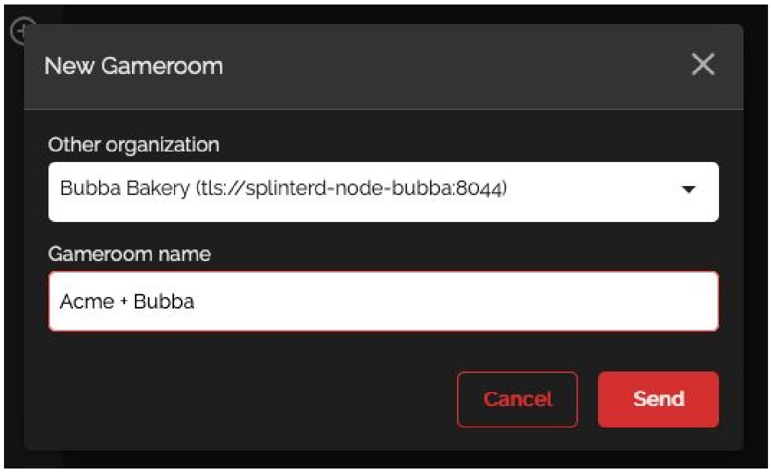
Alice sees the NEW GAMEROOM DIALOG.



Alice looks at the OTHER ORGANIZATION pulldown list. She selects BUBBA BAKERY.

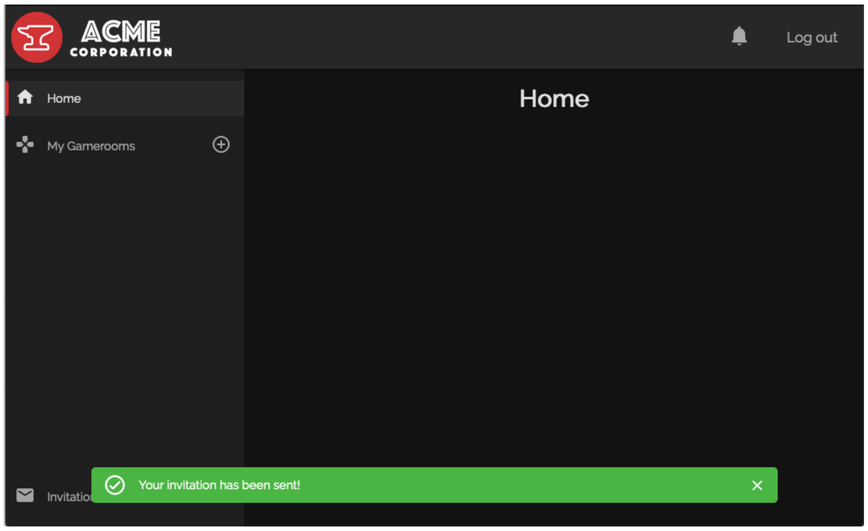


Next, she enters a NAME for the new gameroom: Acme + Bubba.



Alice clicks SEND.

The New Gameroom dialog is replaced with the Acme Gameroom home screen. A TOAST NOTIFICATION tells Alice that her invitation has been sent to Bubba Bakery.

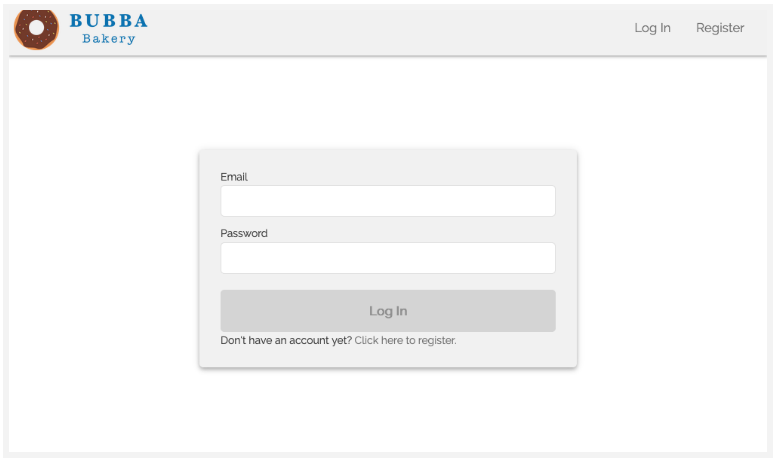


**INTERMISSION**

Live performances should include an intermission at this point, because there is a lot that just happened (see "Behind the Scenes: A Look at Act 1").

## Scene 3: Bob logs into Bubba Bakery's Gameroom application

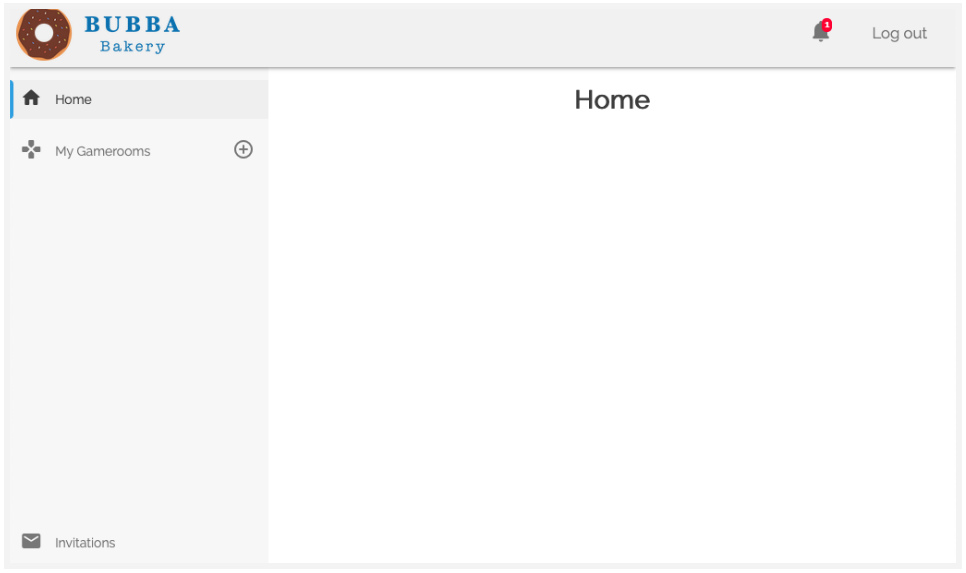
BOB, muttering to himself, opens a BROWSER and searches for "tic tac toe". Gets distracted by Wikipedia's list of games. Plays Quantum Tic Tac Toe Online for 20 minutes. Eventually hunts through his email for the right link and starts the BUBBA BAKERY GAMEROOM APP.



Don't have an account yet? Click here to register.

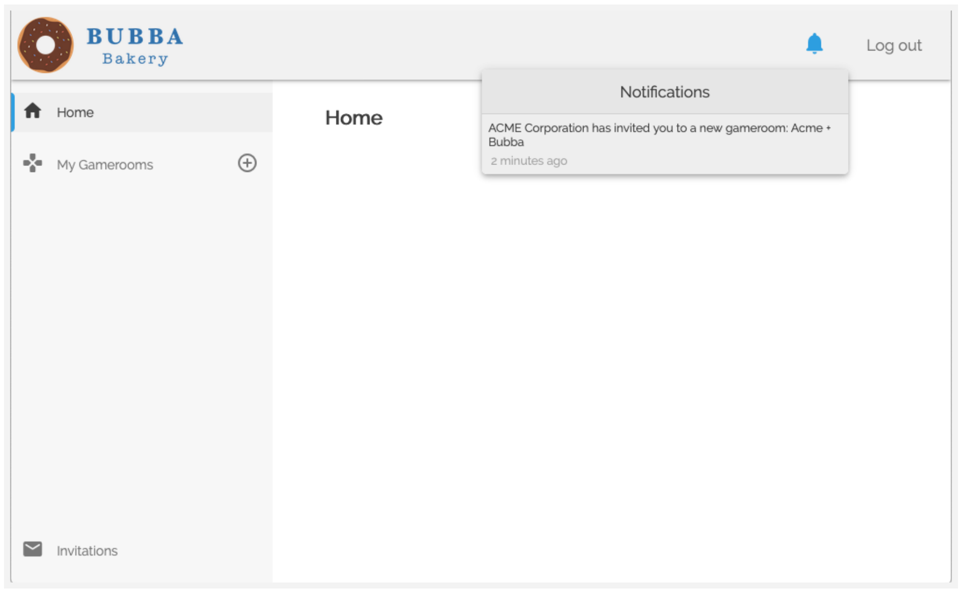
Bob logs in with his EMAIL and PASSWORD.

Success. The browser now displays the BUBBA BAKERY GAMEROOM HOME SCREEN.



## Scene 4: Bob checks his notifications

Bob sees that he has a notification and clicks on the NOTIFICATION ICON. The NOTIFICATION PANE shows an INVITATION from Alice.

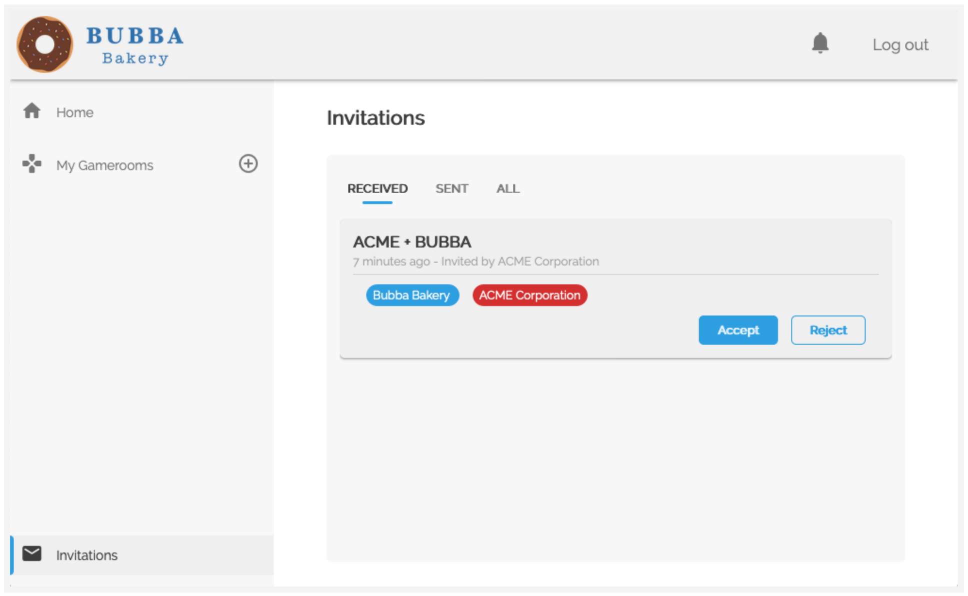


ACME Corporation has invited you to a new gameroom: Acme + Bubba
2 minutes ago

## Scene 5: Bob accepts Alice's gameroom invitation

Time passes.

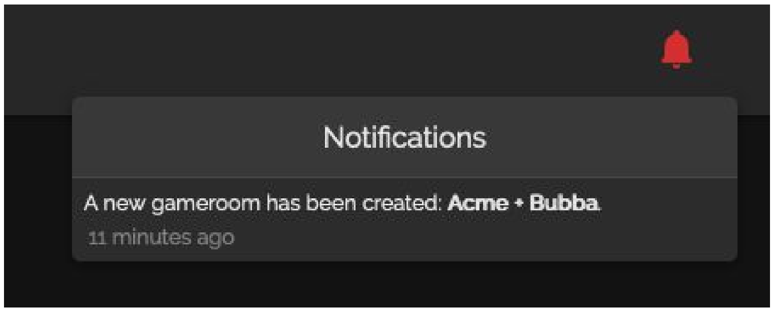
Eventually, Bob clicks the notification. The notifications pane disappears and the INVITATIONS TAB is shown. Bob clicks the ACCEPT button on Alice's invitation.



ACME + BUBBA
7 minutes ago - Invited by ACME Corporation
Bubba Bakery
ACME Corporation

## Scene 6: Alice sees that Bob accepted her invitation

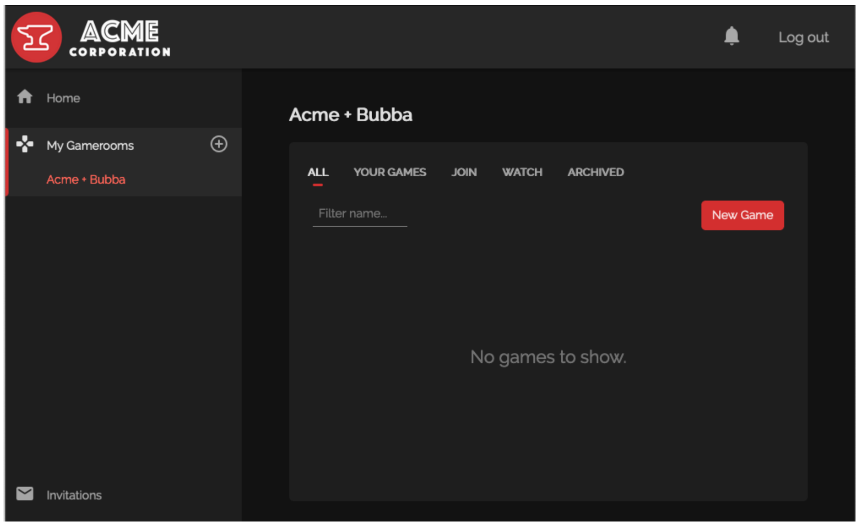
Alice notices that she has a notification and clicks on the notification icon. The NOTIFICATIONS PANE appears, with the happy news that Bob has accepted her invitation and that the new Acme + Bubba gameroom has been created.



A new gameroom has been created: **Acme + Bubba**

11 minutes ago

Alice clicks on the notification. The Notifications pane closes and Alice is redirected to the ACME + BUBBA GAMEROOM SCREEN.



No games to show.

Alice and Bob’s gameroom is ready. They can now play games.

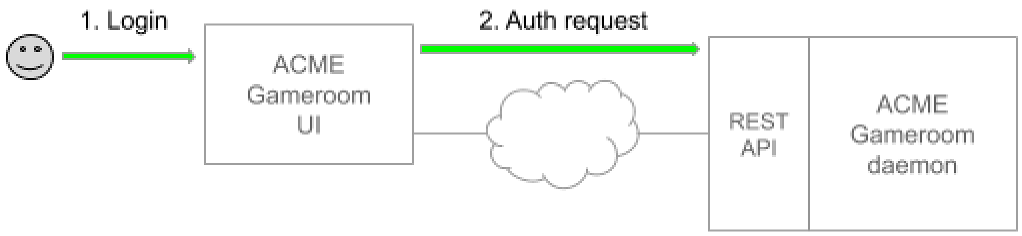
# Behind the Scenes: A Look at Act Ⅰ, Alice and Bob Create a Gameroom

This section explains what really happens during the apparently simple steps in Act Ⅰ. As you read this section, see the glossary for definitions of unfamiliar terms.

## Ⅰ-1. Behind scene 1: Alice logs into Acme's Gameroom UI

When a user logs in, the user interface (UI) component of the Gameroom client application works with the Gameroom REST API to check the user's email address and password. Each Gameroom daemon stores the user credentials in a local PostgreSQL database; user passwords are hashed so that they remain secret.

### Ⅰ-1.1. Acme UI sends authorization request to Gameroom REST API

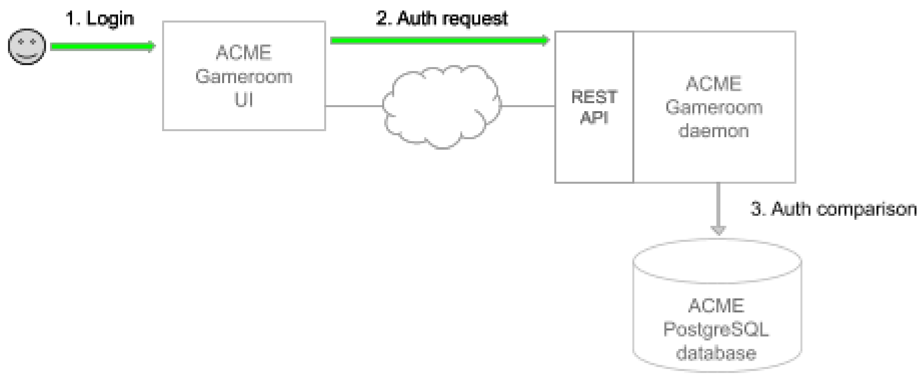


When Alice clicks `Log in`, the Acme Gameroom UI hashes the password, then sends an authorization request to the Acme Gameroom daemon, `gameroomd`. The request is handled by the Gameroom REST API, which is a part of `gameroomd`.

```
POST /users/authenticate
{
    email: "alice@acme.com",
    hashedPassword: "8e066d41...d99ada0d"
}
```

The UI does not reveal the user's password to the REST API because the password is used to encrypt signing keys (as described in section Ⅰ-2.3, step 5).

## I -1.2. Gameroom REST API authorizes login



When the Acme Gameroom REST API receives the authorization request for Alice, it re-hashes the password sent from the browser and compares the email and hashed password to Alice's entry in the Acme Gameroom daemon's local database. If they match, authentication was successful.

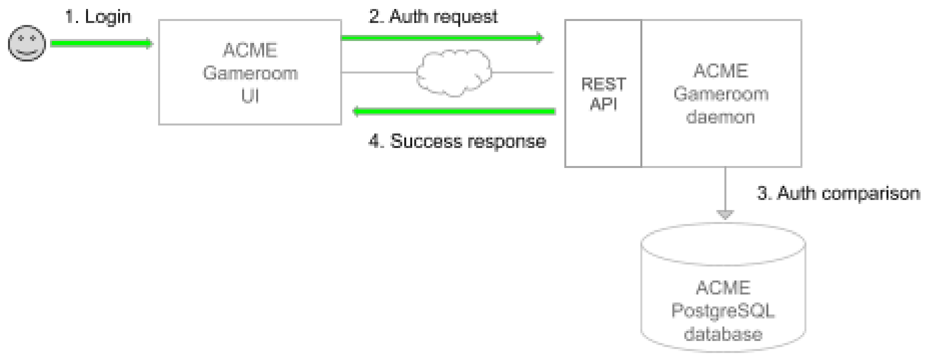
The `gameroom_user` table in the Gameroom database has the following schema:

```
CREATE TABLE IF NOT EXISTS gameroom_user (
  email                      TEXT        PRIMARY KEY,
  public_key                 TEXT        NOT NULL,
  encrypted_private_key      TEXT        NOT NULL,
  hashed_password            TEXT        NOT NULL
);
```

Alice's public and private key pair was created during registration and was added to the Acme Gameroom database (see The Prequel, section P.2). The database has the following entry:

| email | hashed_password | public_key | encrypted_private_key |
|---|---|---|---|
| alice@acme.com | 56ec82cb...480cad32 | 0384781f...5a7e4998 | {\"iv\":...cgXrm\"} |

## I -1.3. Gameroom REST API returns login success response



If the user authentication was successful, the Gameroom REST API sends a response to the Acme UI that contains Alice's public key and encrypted private key.

```
{
        email: “alice@acme.com”,
        public_key: “0384781f...5a7e4998”,
        encrypted_private_key: "{\"iv\":...cgXrm\"}",
}
```

Next, the UI must gather the information for the Acme Gameroom home screen that Alice will see after logging in.

## I -1.4. Acme UI requests a list of gamerooms

After a user has been authenticated, the UI gathers user-specific information for the home screen. First, it requests the list of existing gamerooms for that user. (At this point, no gamerooms exist.) Later, this walkthrough will describe what happens when there are gamerooms for the UI to display.

1. When Alice logs in, the Acme UI makes a call to the Gameroom REST API for the list of gamerooms.

   ```
   GET /gamerooms
   ```

2. This call returns an empty list, since there are no gamerooms in the Acme Gameroom's PostgreSQL database.

   ```
   {
          "data": [],
          "paging": {
                 "current": "/gamerooms?limit=100&offset=0",
                 "offset": 0,
                 "limit": 100,
                 "total": 0,
                 "first": "/gamerooms?limit=100&offset=0",
                 "prev": "/gamerooms?limit=100&offset=0",
                 "next": "/gamerooms?limit=100&offset=0",
                 "last": "/gamerooms?limit=100&offset=0"
          }
   }
   ```

## I -1.5. Acme UI requests a list of invitations

Next, the Acme Gameroom UI requests the list of gameroom invitations. In this scenario, Alice has no invitations, so the list is empty. Later, the walkthrough will show what happens when a user has unaccepted invitations.

1. When Alice logs in, the UI makes a call to the Gameroom REST API for the list of invitations (also called *circuit proposals*).

```
GET /proposals
```

2. Because Alice has no invitations, the Gameroom REST API returns an empty list.

```
{
	"data": [],
	"paging": {
		"current": "/proposals?limit=100&offset=0",
		"offset": 0,
		"limit": 100,
		"total": 0,
		"first": "/proposals?limit=100&offset=0",
		"prev": "/proposals?limit=100&offset=0",
		"next": "/proposals?limit=100&offset=0",
		"last": "/proposals?limit=100&offset=0"
	}
}
```

At this point, Alice sees the Acme Gameroom home screen with no existing gamerooms or invitations.

## Ⅰ-2. Behind scene 2: Alice creates a new gameroom

The Gameroom home screen includes a button to create a new gameroom. When a user clicks it, the UI requests the member list (possible other nodes) to use in the next dialog.

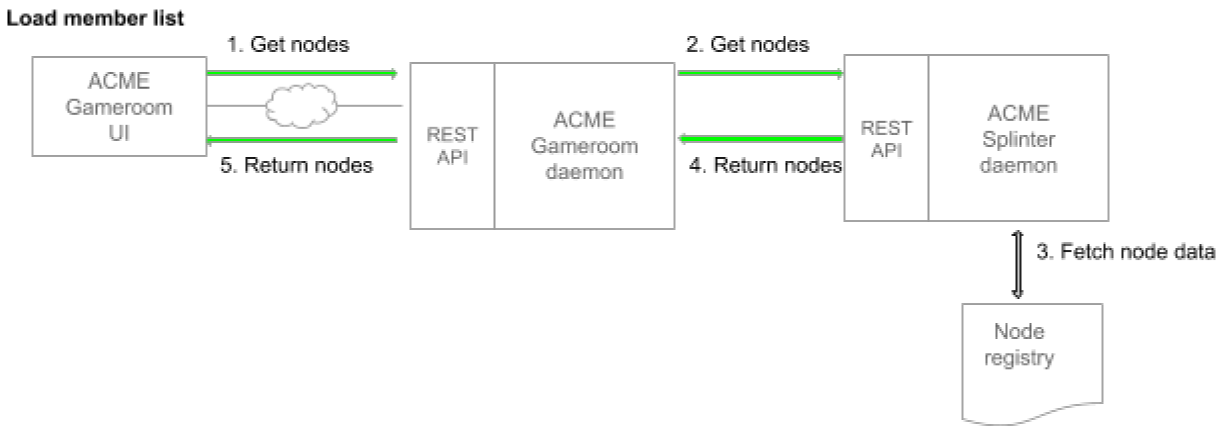
After the Acme UI has the member list, it displays the "New Gameroom" dialog, where Alice can use the members list to select her opponent (called *Other organization* in the dialog), and enter a name for the new gameroom. When she clicks **`Send`**, the Acme UI starts the process of sending Bob an invitation to the new gameroom.

A gameroom is enabled by a Splinter *circuit* that connects two or more systems, or *nodes*. A *node registry* stores a list of nodes that can participate in a circuit; the Splinter daemon, `splinterd`, can provide this list of nodes upon request. (The Gameroom example creates a node registry that includes the Acme and Bubba Bakery nodes.) Splinter uses the term *members* for the nodes that can be connected (or are connected) on a circuit.

A gameroom invitation is also called a *circuit proposal*. Each gameroom proposal requires a vote (an *approval*) from each member, which is handled by two-phase commit consensus and a *consensus proposal*. When Alice creates a new gameroom, her action automatically includes a vote from her organization (Acme Corporation) that approves the creation of that gameroom. Her invitation to Bob, at Bubba's Bakery, is actually a request for his organization's vote to approve the new circuit.

## I -2.1. Acme UI loads members list for New Gameroom dialog

First, the Acme Gameroom UI must load the list of members for the "New Gameroom" dialog. The general process looks like this:



1. The UI makes this REST API call to the Gameroom REST API.

```
GET /nodes
```

2. The Gameroom REST API sends a `GET` request to the `/nodes` endpoint in the Splinter REST API asking for the list of nodes.

3. The Splinter daemon, `splinterd`, fetches the list of nodes from the node registry and sends a response to the Gameroom REST API that includes the requested data. The "list of nodes" response looks like this:

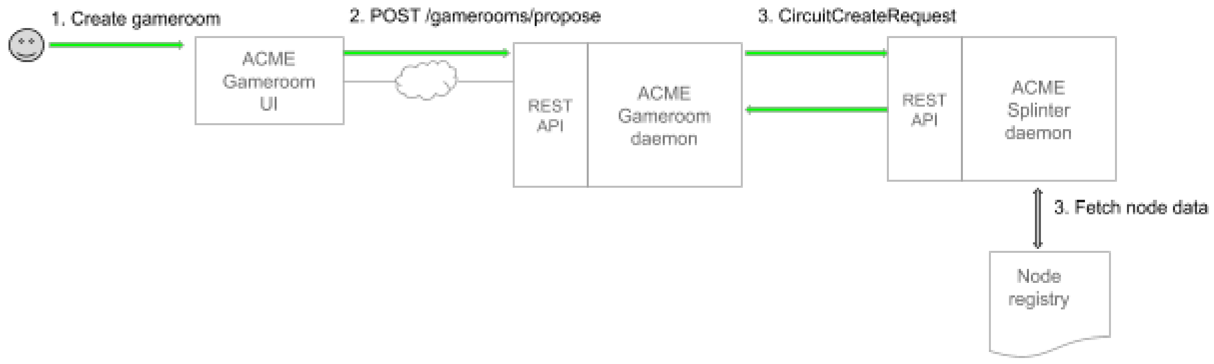
```
{
       "data": [
       {
              "identity": "bubba-node-000",
              "metadata": {
                     "organization": "Bubba Bakery",
                     "endpoint": "tls://splinterd-node-bubba:8044"
              }
       },
       {
              "identity": "acme-node-000",
              "metadata": {
                     "organization": "ACME Corporation"
                     "endpoint": "tls://splinterd-node-acme:8044",
              }
       }
       ],
       "paging": {
              "current": "/nodes?limit=100&offset=0",
              "offset": 0,
              "limit": 100,
              "total": 2,
              "first": "/nodes?limit=100&offset=0",
              "prev": "/nodes?limit=100&offset=0",
              "next": "/nodes?limit=100&offset=0",
              "last": "/nodes?limit=100&offset=0"
       }
}
```

4. The Gameroom REST API forwards the response to the Acme Gameroom UI, which uses the list of nodes to build the members list in the New Gameroom dialog.

## I -2.2. Acme UI sends Create Gameroom request to Gameroom REST API

In the New Gameroom dialog, Alice enters a unique name for the gameroom (`Acme + Bubba`) and selects Bubba Bakery from the **Other Organizations** list. Then she clicks **`Send`** to forward her invitation to Bob.

When Alice clicks on the **`Send`** button, the general process looks like this:



The UI sends a "create new gameroom" request to the Gameroom REST API that includes the gameroom name (also called an *alias*) and list of members in the proposed gameroom.  Each member entry includes the node ID, organization name, and endpoint for its Splinter REST API. The request (also called a *proposal*) looks like this:

```
POST /gamerooms/propose
{
        “alias”: “Acme + Bubba”,
        “members”: [
                {
                “identity”: “bubba-node-000”,
                “metadata”: {
                        “organization”: “Bubba Bakery”,
                        “endpoint”: “tls://splinterd-node-bubba:8044”,
                        }
                }],
}
```

## I -2.3. Gameroom REST API sends CircuitManagementPayload

When the Acme Gameroom REST API receives the proposal request, it uses that information to create a `CircuitManagementPayload`, which will eventually be sent to the Acme Splinter daemon. Before sending the proposal request, the Gameroom REST API asks the Gameroom UI to sign it with Alice's information.

1. The Gameroom daemon uses the information from the "create new gameroom" request to create a new `CircuitManagementPayload`.

   The following example shows a YAML representation of the `CircuitManagementPayload`.

   ```
   Application metadata:
   ---
   alias: Acme + Bubba // Gameroom name chosen by Alice
   scabbard_admin_keys:
     - <acme gameroomd public key>

   Circuit definition:
   ---
   circuit_id: gameroom::acme-node-000::bubba-node-000::<UUIDv4>
   authorization_type: Trust
   members:
     - node_id: acme-node-000
       endpoint: tls://splinterd-node-acme:8044
     - node_id: bubba-node-000
       endpoint: tls://splinterd-node-bubba:8044
   roster:
     - service_id: gameroom_acme-node-000
       service_type: scabbard
       allowed_nodes:
         - acme-node-000
       arguments:
         - peer_services:
             - gameroom_bubba-node-000
           admin_keys:
             - <acme gameroomd public key>
     - service_id: gameroom_bubba-node-000
       service_type: scabbard
       allowed_nodes:
         - bubba-node-000
       arguments:
         - peer_services:
             - gameroom_acme-node-000
           admin_keys:
             - <acme gameroomd public key>
   circuit_management_type: gameroom
   ```

```
application_metadata: <bytes of the application metadata described above>
persistence: Any
durability: None
routes: Any
```

```
Header:
---
Action: CIRCUIT_CREATE_REQUEST
requester: <public key of requester> // left empty by the REST API
payload_sha512: <sha512 hash of the circuit definition described above>
requester_node_id: acme-node-000
```

```
CircuitManagmentPayload:
---
header: <bytes of header described above>
circuit_create_request: <circuit definition described above>
signature: <signature of bytes of the header> // left empty by the REST API
```

Note that the Gameroom REST API does not fill in the `requester` field in the header or the signature in the `CircuitManagementPayload`.

2. Before the payload can be sent, the Acme UI must sign the bytes of the `CircuitManagementPayload` header. The Acme Gameroom REST API serializes the payload and sends the bytes as a response to the UI.

```
{
       "data":  {
             “Payload_bytes”: <bytes of the CircuitManagementPayload>
       }
}
```

3. After receiving the response from the Gameroom REST API, the Acme UI deserializes the `CircuitManagementPayload`. It adds the requester’s public key to the header (in this case, Alice is the requester), serializes the header, signs the header bytes, and adds the signature to the payload. Finally, the UI serializes the complete payload.

4. The Acme UI submits the bytes of the signed payload to the Gameroom REST API.

```
POST /submit
Content-Type: application/octet-stream

<bytes of the signed CircuitManagementPayload>
```

5. The Acme Gameroom REST API forwards the payload to the Acme Splinter REST API.

```
POST /admin/submit
Content-Type: application/octet-stream

<bytes of the signed CircuitManagementPayload>
```

6. The Acme Splinter REST API calls the Acme admin service to forward the proposed payload, a `CircuitManagementPayload` (described in the next section). The protobuf is represented in YAML format:

```
---
CircuitManagmentPayload:
    header: <bytes of header described above>
    circuit_create_request:
        circuit:
            gameroom::acme-node-000::bubba-node-000::<UUIDv4>:
                auth: trust
                members:
                    acme-node-000:
                      endpoints:
                          - tls://splinterd-node-acme:8044
                    bubba-node-000:
                      endpoints:
                          - tls://splinterd-node-bubba:8044
                roster:
                    gameroom_acme-node-000:
                       service_type: scabbard
                       allowed_nodes:
                          - acme-node-000
                       arguments:
                         - peer_services:
                             - gameroom_bubba-node-000
                           admin_keys:
                             - <acme gameroomd public key>
                    gameroom_bubba-node-000:
                       service_type: scabbard
                       allowed_nodes:
                          - bubba-node-000

                       arguments:
                         - peer_services:
                             - gameroom_acme-node-000
                           admin_keys:
                             - <acme gameroomd public key>
                persistence: any
                durability: none
                routes: require_direct
                circuit_management_type: gameroom
        signature: <signature of bytes of requested circuit definition>
```

7. The Acme admin service checks that the `CircuitManagementPayload` signature is valid by comparing it against the header bytes and the requester public key stored in the header.

8. Because the Acme and Bubba Bakery nodes are not yet peered (do not have an authorized connection on the Splinter network), the `CircuitManagmentPayload` is placed in the "pending payloads" queue for unpeered nodes.

### I -2.4. Acme node peers with Bubba Bakery node

Before the `CircuitManagementPayload` message can be validated, every member of the circuit must be connected (peered).

The admin service on the Acme Splinter node (which has the service ID `admin::acme-node-000`) uses a `PeerConnector` to request connections with the members. The `PeerConnector` joins a transport and a network in order to enable adding peers at runtime, without having knowledge of the underlying transport.

1. Acme's admin service calls `PeerConnector.connect` with the node ID and the endpoint listed in the proposed circuit. If the node is already connected, the peer connector returns "`Ok`". If the node is not connected, the peer connector creates the connection and, if successful, adds the connection to the Splinter network.

2. After the connection has been created, a message exchange starts for peer authorization (described in Appendix A).

3. When peer authorization succeeds, the Acme admin service is notified that the Bubba Bakery node (`bubba-node-000`) has been successfully authorized. Acme and Bubba are now peers.

4. The `CircuitManagmentPayload` is removed from the "pending payloads" queue and is passed to the admin service handler for pending circuit payloads.

## I -2.5. Splinter daemons use consensus to process the circuit request

At this point, the circuit proposal is ready to be validated and approved (voted on with two-phase commit consensus), as described in Appendix B.

During this process, the admin services on both nodes (`admin::acme-node-000` and `admin::bubba-node-000`) must agree that the `CircuitManagementPayload`, which includes `CircuitCreateRequest`, is a valid request. Consensus manages each node's approval of the proposal.

### I -2.5.1. Acme node validates CircuitManagmentPayload

1. The Acme admin service verifies that the `CircuitManagementPayload` and the included `CircuitCreateRequest` are valid.
    a. A `CircuitManagementPayload` request is valid if the following things are true:
        - The `CircuitManagementPayload` must contain a header and signature in bytes.
        - The header in the payload must contain an action enum value, the public key of the requester, and hash of the action associated with the payload.
        - The action in the `CircuitManagementPayload` must match the enum action in the payload.
        - The signature must be valid for the bytes of the header and the requester public key stored in the header.

    b. The provided payload (a `CircuitCreateRequest`) is valid if the following things are true:
        - The new circuit has a unique name (the node is not part of an existing circuit with that name). Circuit names do not need to be unique across all Splinter nodes. Two sets of nodes can use the same circuit name if there is no overlap in members in the circuit.
        - The circuit definition includes the node ID in the circuit member list.
        - For each service, every node in the service's allowed node list is also present in the circuit member list.
        - There is no other pending proposal for a circuit with the same name.
        - The requester is registered for the Splinter node whose ID is in the `requester_node_id` field of the `CircuitManagementPayload` header. The requester is identified by the public key of the person who requested the new gameroom (in this example, Alice).
        - The requester has permission to submit circuit proposals from that Splinter node.

        To verify the node's public key and proposal permission, the admin service checks the key registry and key permissions manager.

- The key registry provides a way to look up details about a public key used to sign a circuit proposal: the requester node ID (the "home node" of the requester and location of that user's public key) and arbitrary metadata (represented as key/value string pairs).
- The key permissions manager checks that a public key is authorized in a specific role.  In the case of "create circuit" requests, the signing public key must be authorized for the "proposal" role.

2. If the request is valid, the Acme admin service creates a `CircuitProposal` and stores it in the `AdminServiceShared.pending_changes` field. The protobuf is represented in YAML format.

```
---
CircuitProposal:
    proposal_type: CREATE
    circuit_id: gameroom::acme-node-000::bubba-node-000::<UUIDv4>:
    circuit_hash: <hash of circuit>
    circuit_proposal:
      circuit:
           gameroom::acme-node-000::bubba-node-000::<UUIDv4>:
               auth: trust
               members:
                   acme-node-000:
                      endpoints:
                          - tls://splinterd-node-acme:8044
                   bubba-node-000:
                      endpoints:
                          - tls://splinterd-node-bubba:8044
               roster:
                   gameroom_acme-node-000:
                       service_type: scabbard
                       allowed_nodes:
                          - acme-node-000
                       arguments:
                         - peer_services:
                             - gameroom_bubba-node-000
                           admin_keys:
                             - <acme gameroomd public key>
                   gameroom_bubba-node-000:
                       service_type: scabbard
                       allowed_nodes:
                          - acme-node-000


                       arguments:
```

```
                      - peer_services:
                          - gameroom_acme-node-000
                        admin_keys:
                          - <acme gameroomd public key>
              persistence: any
              durability: none
              routes: require_direct
    votes: []
    requester: <public key of requester>
    requester_node_id: acme-node-000
```

3. The Acme admin service creates a consensus proposal (a `Proposal` struct) with the following contents:
   - Proposal ID: the expected hash of the `CircuitManagementPayload` bytes
   - Summary: the expected hash of the created `CircuitProposal`
   - List of required verifiers

   An admin service running on a Splinter node does not have a fixed (static) list of required verifiers (services that must agree on a proposal through consensus). Instead, the admin service specifies the required verifiers as a list of admin service IDs that belong to the members of the proposed circuit, using a protobuf message called `RequiredVerifiers`. This list is stored in the consensus data of the consensus proposal.

   The following protobuf, which is represented in YAML format, shows the consensus proposal.

```
---
required_verifiers:
    verifiers:
          -  <admin::acme-node-000 as bytes>
          -  <admin::bubba-node-000 as bytes>

---
proposal:
    id: <hash of CircuitManagementPayload bytes>
    summary: <expected hash of the create CircuitProposal>
    consensus_data: <bytes of required verifiers>
```

### I -2.5.2. Acme node sends Circuit Create request to Bubba Bakery node

After the Acme node creates the `CircuitProposal`, the `CircuitManagmentPayload` is sent to the other members defined in the circuit. In this case, the only member is the admin service on the Bubba Bakery node.

1. First, the Acme admin service wraps `CircuitManagementPayload` in a series of messages to prepare it for sending across the Splinter network.

   a. The payload is wrapped in an `AdminMessage`, which is a service-level message. The protobuf is represented in YAML format.

   ```
   ---
   admin_message:
       message_type: PROPOSED_CIRCUIT,
       propose_circuit:
           circuit_payload: <circuit_managment_payload>
           expected_hash: <expected hash of CircuitProposal generated by payload>
           required_verifiers:< bytes of the required verifiers from proposal>
   ```

   b. The `AdminMessage` is then wrapped in an `AdminDirectMessage`, which enables the message to be sent over the Splinter network from Acme's admin service to the Bubba Bakery admin service (which has the service ID `admin::bubba-node-000`).

   ```
   ---
   admin_direct_message:
          circuit: admin
          sender: admin::acme-node-000
          recipient: admin::bubba-node-000
          payload: <serialized admin message>
          correlation_id: 6f04e471-f33a-4f9f-ad6f-5f80ab627133
   ```

   c. Next, the `AdminDirectMessage` is wrapped in a `CircuitMessage`, which is the envelope that wraps all circuit-specific messages, such as direct messages and service connections.

   ```
   ---
   circuit_message:
       message_type: ADMIN_DIRECT_MESSAGE
       payload: <serialized admin_direct_message>
   ```

   d. In order to hide circuits from the network layer, which can be used without circuits, the `CircuitMessage` is wrapped in a `NetworkMessage`.

   ```
   ---
   network_message:
       message_type: CIRCUIT
       payload: <serialized circuit_message>
   ```

2. The Acme admin service sends this message over the admin circuit to the Bubba Bakery Splinter node.

### I -2.5.3. Bubba Bakery node receives Circuit Create request from Acme node

The Bubba Bakery Splinter node receives the network message from the Acme node and starts the process of "unwrapping" the message with a series of dispatchers.

1. A dispatcher takes the message and passes it to the correct message handler based on the message type of the message. Each dispatcher either handles the message or forwards the message onto the next dispatcher.
    a. The Bubba Bakery Splinter node passes the `NetworkMessage` to the network dispatcher.
    b. The network dispatcher unwraps the `NetworkMessage` to get the `CircuitMessage`, then sends it to the circuit dispatcher.
    c. The circuit dispatcher unwraps the `CircuitMessage` to get the `AdminDirectMessage`, then passes it to the circuit handler for this type of message, `AdminDirectMessageHandler`.
2. The `AdminDirectMessageHandler` checks whether the `AdminDirectMessage` is valid.

   An `AdminDirectMessage` message is valid if both the sender and the recipients of the message are admin services (the service ID of each is of the form `admin::<node_id>`).
3. If the `AdminDirectMessage` message is valid, the `AdminDirectMessageHandler` forwards it to the Bubba Bakery admin service.
4. The Bubba Bakery admin service takes the `AdminMessage` out of the `AdminDirectMessage` and inspects the `AdminMessage` to see if it contains `AdminMessage::ProposedCircuit`.

   If so, the admin service takes the `CircuitManagmentPayload` out of the `ProposedCircuit` message and passes it to `AdminServiceShared.pending_circuit_payloads.`

### I -2.5.4. Bubba Bakery node validates CircuitManagmentPayload

The Bubba Bakery admin service validates the `CircuitManagementPayload` using the same steps as in section I -2.5.1.

1. The admin service verifies that the `CircuitManagementPayload` and the included `CircuitCreateRequest` are valid. (For details, see section I -2.5.1, step 1.)
2. If the request is valid, the admin service creates a `CircuitProposal` and stores it in the `AdminServiceShared.pending_changes` field (see section I -2.5.1, step 2).
3. The admin service creates a consensus proposal (a `Proposal` struct) with the proposal ID, summary, and the list of required verifiers. For more information, see section I -2.5.1, step

3.

#### I -2.5.5. Acme and Bubba Bakery reach consensus

When the admin services have validated the proposal and consensus has reached agreement, consensus will notify the admin services to commit the proposal. See Appendix B for more information about consensus.

### I -2.6. Admin services commit pending circuit proposal

After the consensus notification, both admin services commit the `CircuitProposal`. Now the new circuit is officially pending, which means that the `CircuitProposal` is stored in the admin services' state but the circuit is not yet available for communication. A pending circuit proposal is also called an "open circuit proposal".

In the Gameroom example, the pending circuit ID specifies the application, the member nodes, and a version 4 UUID, as in this example:

```
gameroom::acme-node-000::bubba-node-000::<UUIDv4>
```

### I -2.7. Admin services notify authorization handler of pending circuit proposal

1. After the circuit proposal has been committed, the admin service on each node checks if there are any registered application authorization handlers for the circuit management type in the proposed circuit (`gameroom::acme-node-000::bubba-node-000::<UUIDv4>`). See The Prequel, section P.3, for more information on the registration process.

   An application authorization handler manages the voting strategy for the application and notifies the application of any events received from the admin service of the local Splinter node. This handler registers with an admin service for a specific circuit management type (also described in The Prequel, section P.3).

2. If there are any registered application authorization handlers for the proposed circuit management type, each admin service forwards the request to the local connected Gameroom application authorization handler.

   The notification is sent on a WebSocket connection.

```
{
   "eventType": "ProposalSubmitted",
   "message": {
     "proposal_type": "Create",
     "circuit_id": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
     "circuit_hash": "...",
     "circuit": {
        "circuit_id":"gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
        "authorization_type": "Trust",
        "members": [{
          "node_id": "acme-node-000",
          "endpoint": "tls://splinterd-node-acme:8044"
           },
          {
          "node_id": "bubba-node-000",
          "endpoint": "tls://splinterd-node-bubba:8044"
           }
        ],
        "roster": [{
            "service_id": "gameroom_acme-node-000",
            "service_type": "scabbard",
            "allowed_nodes": [ "acme-node-000" ],
            "arguments": {
              "peer_services": [ "gameroom_bubba-node-000" ],
              "admin_keys": [
                "<acme gameroomd public key>"
                ]
              }
            },
            {
            "service_id": "gameroom_bubba-node-000",
            "service_type": "scabbard",
            "allowed_nodes": [ "bubba-node-000" ],
            "arguments": {
              "peer_services": [ "gameroom_acme-node-000" ],
              "admin_keys": [
                "<acme gameroomd public key>"
                ]
              }
            }
        ],
        "circuit_management_type": "gameroom",
        "application_metadata": <metadata bytes defined by the application>,
        "persistence": "Any",
        "durability": "None",
        "routes": "Any"
       },
      "vote_record": [{}],
      "requester": "public_key_of_requester"
      "requester_node_id": "acme-node-000"
    }
}
```

## I -2.8. Gameroom daemons write notification to Gameroom database

When each Gameroom application authorization handler receives the gameroom proposal on the WebSocket connection, it parses the information and adds it to several tables in the Gameroom daemon's local database: `gameroom`, `gameroom_member`, `gameroom_service`, `gameroom_proposal`, and `gameroom_notification`.

### I -2.8.1. New `gameroom` table entry

First, the Gameroom application authorization handler adds a new entry to the `gameroom` table. This table contains the information about the circuit definition, including the data that was passed in the `application_metadata` field.

```
CREATE TABLE IF NOT EXISTS  gameroom (
  circuit_id               TEXT         PRIMARY KEY,
  authorization_type       TEXT         NOT NULL,
  persistence              TEXT         NOT NULL,
  routes                   TEXT         NOT NULL,
  durability               TEXT         NOT NULL,
  circuit_management_type  TEXT         NOT NULL,
  alias                    TEXT         NOT NULL,
  status                   TEXT         NOT NULL,
  created_time             TIMESTAMP    NOT NULL,
  updated_time             TIMESTAMP    NOT NULL
);
```

- `circuit_id`, `authorization_type`, `persistence`, `routes`, `durability`, and `circuit_management_type` are extracted directly from the circuit proposal message that is received from the Splinter daemon.
- `alias` is extracted by deserializing the application metadata in the circuit proposal message. The alias is the gameroom name that Alice entered when creating the gameroom in the Acme UI.
- `status` identifies the current status of the gameroom. In this case, it is set to `pending` because the proposal to create this gameroom has not yet been accepted.
- `created_time` is when the gameroom entry was introduced in the table.
- `updated_time` is when the gameroom entry was last updated.

At the end of the operation, the `gameroom` table looks like this:

| circuit_id | authorization_type | persistence | routes | durability |
|---|---|---|---|---|
| gameroom::acme-node-000::bubba-node-000::<UUIDv4> | Trust | Any | Any | None |

| circuit_management_type | alias | status | created_time | updated_time |
|---|---|---|---|---|
| gameroom | Acme + Bubba | pending | <time entry was created> | <time entry was updated> |

### I -2.8.2. New `gameroom_member` table entry

Next, the Gameroom application authorization handler adds a new entry to the `gameroom_member` table. This table contains the information about the members of the circuit.

```
CREATE TABLE IF NOT EXISTS gameroom_member (
  id                      BIGSERIAL   PRIMARY KEY,
  circuit_id              TEXT         NOT NULL,
  node_id                 TEXT         NOT NULL,
  endpoint                TEXT         NOT NULL,
  status                  TEXT         NOT NULL,
  created_time            TIMESTAMP    NOT NULL,
  updated_time            TIMESTAMP    NOT NULL,
  FOREIGN KEY (circuit_id) REFERENCES gameroom(circuit_id) ON DELETE CASCADE
);
```

- `circuit_id`, `node_id`, and `endpoint` are extracted directly from the circuit proposal message received from the Splinter daemon.
- `status` identifies the current status of the member. In this case, it is set to `pending` because the proposal to create the gameroom has not yet been accepted.
- `created_time` is when the gameroom member entry was introduced in the table.
- `updated_time` is when the gameroom member entry was last updated.

At the end of the operation, the `gameroom_member` table looks like this:

| id | circuit_id | node_id |
|---|---|---|
| `<auto generated id>` | `gameroom::acme-node-000::bubba-node-000::<UUIDv4>` | `acme-node-000` |
| `<auto generated id>` | `gameroom::acme-node-000::bubba-node-000::<UUIDv4>` | `bubba-node-000` |

| endpoint | status | created_time | updated_time |
|---|---|---|---|
| `tls://splinterd-node-acme:8044` | pending | <time entry was created> | <time entry was updated> |
| `tls://splinterd-node-bubba:8044` | pending | <time entry was created> | <time entry was updated> |

### I -2.8.3. New `gameroom_service` table entry

The Gameroom application authorization handler adds a new entry to the `gameroom_service` table, which contains the information about the services of the circuit.

```
CREATE TABLE IF NOT EXISTS gameroom_service (
  id                        BIGSERIAL   PRIMARY KEY,
  circuit_id                TEXT         NOT NULL,
  service_id                TEXT         NOT NULL,
  service_type              TEXT         NOT NULL,
  allowed_nodes             TEXT[][]     NOT NULL,
  arguments                 JSON []      NOT NULL,
  status                    TEXT         NOT NULL,
  created_time              TIMESTAMP    NOT NULL,
  updated_time              TIMESTAMP    NOT NULL,
  FOREIGN KEY (circuit_id) REFERENCES gameroom(circuit_id) ON DELETE CASCADE
);
```

- `circuit_id, service_id, service_type, arguments` and `allowed_nodes` are extracted directly from the circuit proposal message received from the Splinter daemon.
- `status` identifies the current status of the service. In this case, it is set to `pending` because the proposal to create the gameroom has not yet been accepted.
- `created_time` is when the gameroom service entry was introduced in the table.
- `updated_time` is when the gameroom service entry was last updated.

At the end of the operation, the `gameroom_service` table looks like this:

| id | circuit_id | service_id | service_type |
|---|---|---|---|
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | gameroom_acme-node-000 | scabbard |
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | gameroom_bubba-node-000 | scabbard |

| allowed_nodes | arguments | status | created_time | updated_time |
|---|---|---|---|---|
| {"acme-node-000"} | "peer_services": [ "gameroom_bubba-node-000" ], "admin_keys": …. | pending | <time entry was created> | <time entry was updated> |
| {bubba-node-000} | "peer_services": [ "gameroom_bubba-node-000" ], "admin_keys": …. | pending | <time entry was created> | <time entry was updated> |

### I -2.8.4. New `gameroom_proposal` table entry

The Gameroom application authorization handler adds a new entry to the `gameroom_proposal` table, which contains information about the gameroom proposal.

```
CREATE TABLE IF NOT EXISTS gameroom_proposal (
  id                       BIGSERIAL   PRIMARY KEY,
  proposal_type            TEXT         NOT NULL ,
  circuit_id               TEXT         NOT NULL,
  circuit_hash             TEXT         NOT NULL,
  requester                TEXT         NOT NULL,
  requester_node_id       TEXT         NOT NULL,
  status                   TEXT         NOT NULL,
  created_time             TIMESTAMP    NOT NULL,
  updated_time             TIMESTAMP    NOT NULL,
  FOREIGN KEY (circuit_id) REFERENCES gameroom(circuit_id) ON DELETE CASCADE
);
```

- `circuit_id`, `proposal_type`, `circuit_hash`, `requester` and `requester_node_id` are extracted directly from the circuit proposal message received from the Splinter daemon.
- `status` identifies the current status of the proposal. In this case, it is set to `pending` because the proposal to create the gameroom has not yet been accepted.
- `created_time` is when the gameroom proposal entry was introduced in the table.
- `updated_time` is when the gameroom proposal entry was last updated.

At the end of the operation, the `gameroom_proposal` table looks like this:

| id | circuit_id | proposal_type | circuit_hash |
|---|---|---|---|
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | Create | <hash of circuit definition> |

| requester | requester node id | status | created_time | updated_time |
|---|---|---|---|---|
| <public key of requester> | acme-node-000 | pending | <time entry was created> | <time entry was updated> |

### I -2.8.5. New `gameroom_notification` table entry

Finally, the Gameroom application authorization handler adds a new entry to the `gameroom_notification` table. This table contains information about events that the UI would like to notify the users about.

```
CREATE TABLE IF NOT EXISTS gameroom_notification (
  id                        BIGSERIAL   PRIMARY KEY,
  notification_type         TEXT         NOT NULL,
  requester                 TEXT         NOT NULL,
  requester_node_id         TEXT         NOT NULL,
  target                    TEXT         NOT NULL,
  created_time              TIMESTAMP    NOT NULL,
  read                      BOOLEAN      NOT NULL
);
```

- `notification_type` identifies the type of event that generated this notification (in this case, a `gameroom_proposal` event).
- `requester` identifies the public key of the user that generated the event (in this case, Alice's public key).
- `target` is the identifier for the resource that was affected by the event (in this case, the `circuit_id`).
- `created_time` is when the notification entry was introduced in the table.
- `read` identifies whether the user has seen that notification.

At the end of the operation, the `gameroom_notification` table looks like this:

| id | notification_type | requester | requester_node_id |
|---|---|---|---|
| <auto generated id> | gameroom_proposal | <Alice’s public key> | acme-node-000 |

| target | created_time | read |
|---|---|---|
| gameroom::acme-node-000::bubba-node-000::<UUIDv4> | <time entry was created> | f |

## I -2.9. Alice sees notification that gameroom invitation was sent

1. After the Acme Gameroom application authorization handler fills in the `gameroom_notification` table, the Acme Gameroom REST API uses a WebSocket connection to tell the Acme UI about the new notification.

```
{
   "namespace": "notifications",
   "action": "listNotifications"
 }
```

2. When the Acme UI receives that message, it sends a request to the Gameroom REST API to fetch a list of unread notifications from the database tables.

```
GET /notifications
```

3. The Acme Gameroom REST API responds with the list of unread notifications.

```
{
  "data": [
    {
      "id": <auto generated id>,
      "notification_type": "gameroom_proposal",
      "requester": <Alice’s public key>,
       “node_id”: “acme-node-000”,
      "target": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
      "timestamp": <time entry was created>,
      "read": false
    }
  ],
  "paging": {
       "current": "api/notifications?limit=100&offset=0",
       "offset": 0,
       "limit": 100,
       "total": 1,
       "first": "api/notifications?limit=100&offset=0",
       "prev": "api/notifications?limit=100&offset=0",
       "next": "api/notifications?limit=100&offset=0",
       "last": "api/notifications?limit=100&offset=0"
   }
}
```

4. The Acme UI updates its internal store with the new list of notifications. The notification that the user sees depends on whether they're the requester or an invitee.

    - For the requester (Alice), the Acme UI displays a toast notification saying that the invitation has been sent.

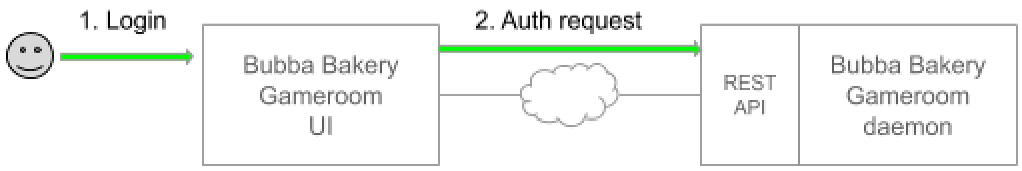
    - An invitee sees a bell notification icon with number (a red badge that shows the number of unread notifications). If an invitee is not logged in, the notification will appear on the Gameroom home screen when the user logs in. For example, when Bob logs in, the Bubba Bakery UI will request the list so it can display the notification icon and the number on the home screen.

## Ⅰ -3. Behind scene 3: Bob logs into Bubba Bakery's Gameroom application

When Bob logs in, the Bubba Bakery UI works with `gameroomd` and Gameroom REST API to check his user credentials and build the Bubba Bakery Gameroom home page. This process is almost identical to Alice's login process. The only difference is that the Bubba Bakery Gameroom home page will display a notification about his invitation from Alice.

### Ⅰ -3.1. Bubba Bakery UI sends authorization request to Gameroom REST API

This process is the same as the Acme process in section Ⅰ -1.1. For Bob, the general process looks like this:



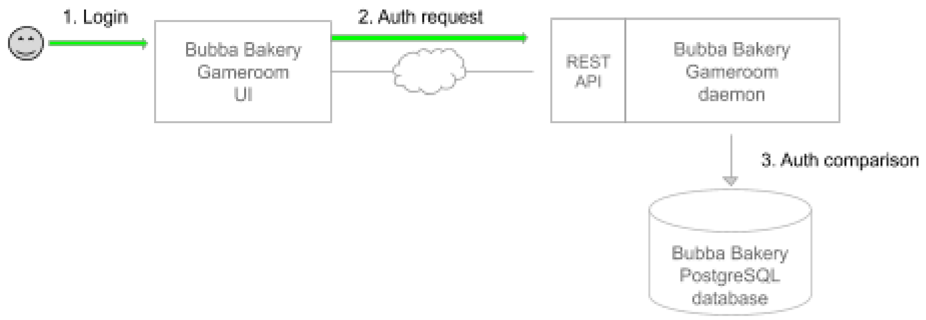
When Bob clicks `Log in`, the Bubba Bakery Gameroom UI hashes the password, then sends an authorization request to the Bubba Bakery Gameroom daemon, `gameroomd`. The request is handled by the Gameroom REST API, which is a part of `gameroomd`.

```
POST /users/authenticate
{
    email: "bob@bubbabakery.com",
    hashedPassword: "2b944c69...c11fcf9c"
}
```

As mentioned earlier, the UI does not reveal the user's password to the REST API because the password is used to encrypt signing keys.

## I -3.2. Bubba Bakery Gameroom REST API authorizes the login

This process is the same as the Acme process in section I -1.2. For Bob, the general process looks like this:



When the Gameroom REST API receives the authorization request for Bob, it re-hashes the password sent from the browser and compares the email and hashed password to Bob's entry in the Bubba Bakery Gameroom database. If they match, authentication was successful.

The `gameroom_user` table in the Gameroom database has the following schema:

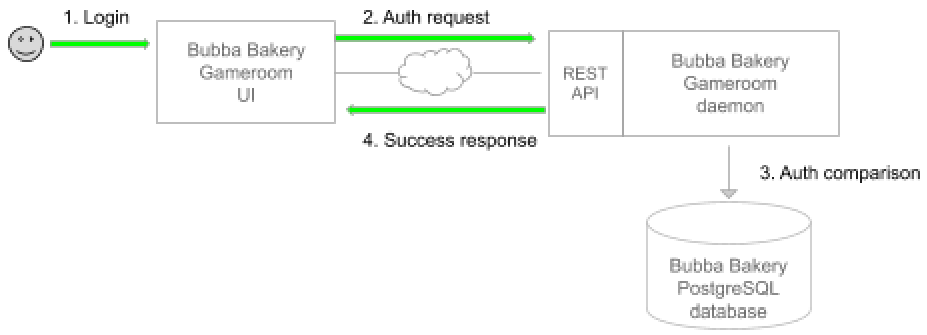
```
CREATE TABLE IF NOT EXISTS gameroom_user (
  email                      TEXT         PRIMARY KEY,
  public_key                 TEXT         NOT NULL,
  encrypted_private_key      TEXT         NOT NULL,
  hashed_password            TEXT         NOT NULL
);
```

Bob's public and private key pair was created before registration and was added to the Bubba Bakery Gameroom database when Bob registered (see The Prequel, section P.2). The database has the following entry:

| email | hashed_password | public_key | encrypted_private_key |
|---|---|---|---|
| bob@bubbabakery.com | 4c825b14...534bfc25 | b1834871...2914a7f4 | du+XOOyVy...nkO/NiHcn |

## Ⅰ-3.3. Bubba Bakery Gameroom REST API returns login success response

This process is the same as the Acme process in section Ⅰ-1.3. For Bob at Bubba Bakery, the general process looks like this:



If the user authentication was successful, the Gameroom REST API sends a response to the Bubba Bakery UI that contains Bob's public key and encrypted private key.

```
{
        email: “bob@bubbabakery.com”,
        publicKey: “b1834871...2914a7f4”,
        encryptedPrivateKey: "\"{\\\"iv\\\":...ZCyV\\\"}\"",
}
```

Next, the UI must gather information for the list of gamerooms, invitations, and notifications that Bob will see on the Bubba Bakery home page.

## Ⅰ-3.4. Bubba Bakery UI requests list of existing gamerooms

As part of building the Bubba Bakery home screen for Bob, the UI requests the list of Bob's gamerooms. This process is the same as the Acme process in section Ⅰ-1.4.

1. The Bubba Bakery UI makes a call to the Gameroom REST API for the list of existing gamerooms.

   ```
   GET /gamerooms
   ```

2. The Bubba Bakery Gameroom REST API returns an empty list, because there are no existing gamerooms in the Bubba Bakery Gameroom's PostgreSQL database.

   ```
   {
   	"data": [],
   	"paging": {
   		"current": "/gamerooms?limit=100&offset=0",
   		"offset": 0,
   		"limit": 100,
   		"total": 0,
   		"first": "/gamerooms?limit=100&offset=0",
   		"prev": "/gamerooms?limit=100&offset=0",
   		"next": "/gamerooms?limit=100&offset=0",
   		"last": "/gamerooms?limit=100&offset=0"
   	}
   }
   ```

## I -3.5. Bubba Bakery UI requests list of gameroom invitations

As part of building the Bubba Bakery home screen for Bob, the UI requests the list of Bob's invitations. This process is different from the Acme process in section I -1.5, because Bob has a new invitation from Alice.

1. The Bubba Bakery UI makes a call to the Gameroom REST API for Bob's list of invitations (also called *circuit proposals*).

```
GET /proposals
```

2. The Bubba Bakery Gameroom REST API returns a list that includes Alice's invitation.

```
{
       "data": [
       {
            "proposal_id": <auto-generated id>,
            "circuit_id": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
            "circuit_hash": <hash of circuit definition>,
            "members": [
                {
                    "node_id": "acme-node-000",
                    "endpoint": "tls://splinterd-node-acme:8044"
                },
                {
                    "node_id": "bubba-node-000",
                    "endpoint": "tls://splinterd-node-bubba:8044"
                }
            ],
            "requester": <Alice’s public key>,
             "requester_node_id": acme-node-000,
            "created_time": <time entry was created>,
            "updated_time": <time entry was updated>
        }

],
       "paging": {
             "current": "/proposals?limit=100&offset=0",
             "offset": 0,
             "limit": 100,
             "total": 1,
             "first": "/proposals?limit=100&offset=0",
             "prev": "/proposals?limit=100&offset=0",
             "next": "/proposals?limit=100&offset=0",
             "last": "/proposals?limit=100&offset=0"
       }
}
```

### Ⅰ-3.6. Bubba Bakery UI queries for unread notifications

While building the Bubba Bakery home screen, the UI also requests the list of Bob's unread notifications.

When the circuit proposal (Alice's invitation) was created, the Bubba Bakery admin service stored Bob's notification information in the Gameroom database, with the `read` field set to "false". For the details of how the Gameroom tables were updated during circuit creation, see section Ⅰ-2.8 and section Ⅰ-2.9.

1. The Bubba Bakery UI makes a call to the Bubba Bakery Gameroom REST API for the list of Bob's unread notifications.

```
GET /notification
```

2. The Gameroom REST API queries the `gameroom_notification` table and sends Bob's notifications to the Bubba Bakery UI. The notification for Alice's invitation looks like this:

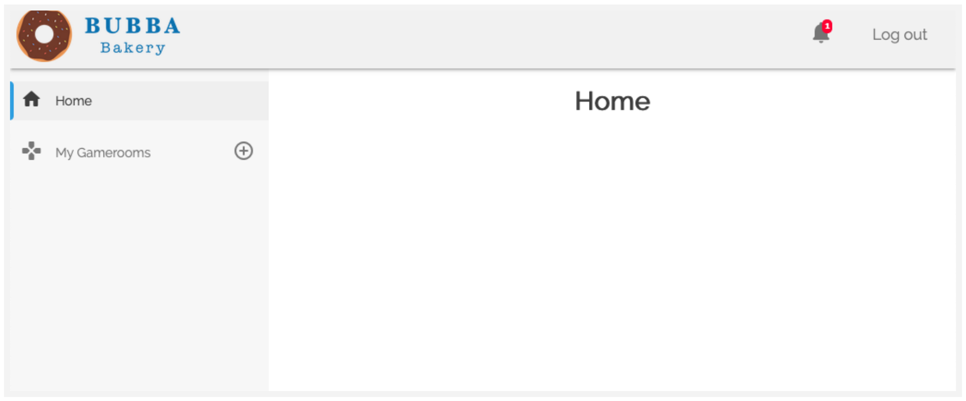
```
{
      "data": [
            {
                  “id”: <auto-generated id>,
                  “notification_type”: “gameroom_proposal”,
                  “org”: “”,
                  “requester”: <Alice’s public key>,
                  “node_id”: "acme-node-000",
                  “target”:
                        "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
                  “timestamp”: <time entry was created>,
                  “read”: <boolean; false means not read>,
            }
],
      “paging”: {
            "current": "/notifications?limit=100&offset=0",
            "offset": 0,
            "limit": 100,
            "total": 1,
            "first": "/notifications?limit=100&offset=0",
            "prev": "/notifications?limit=100&offset=0",
            "next": "/notifications?limit=100&offset=0",
            "last": "/notifications?limit=100&offset=0"

      },
}
```

At this point, the Bubba Bakery Gameroom UI has the information it needs to display Bob's home screen.

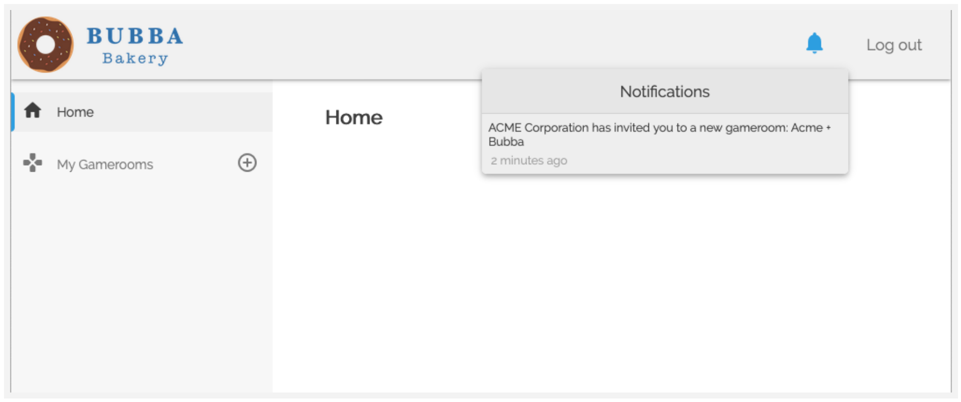
## I -4. Behind Scene 4: Bob checks his notifications

On Bob's Bubba Bakery home screen, the UI displays Bob's existing gamerooms on the left (none, at this point) and notifications in the upper right (as a bell icon).

Bob's public key is not listed as the requester on the `gameroom_proposal` notification, so the Bubba Bakery UI displays the notification icon with a red badge that indicates an unread notification.



1. When Bob clicks the bell icon, the UI shows his unread notifications.



ACME Corporation has invited you to a new gameroom: Acme + Bubba
2 minutes ago

2. When Bob clicks on his notification, the Bubba Bakery UI calls the Bubba Bakery Gameroom REST API to update the selected notification (to show that Bob has read it). After the update, this notification will no longer show up as a new notification in the UI.

```
PATCH /notifications/{notification_id}/read
```

   This call updates the notification's entry `read` field in the Bubba Bakery database's `gameroom_notification` table from false to true. For more information on this table, see section I -2.8.5.

3. After successfully updating the notification, the Bubba Bakery Gameroom REST API responds with the entire notification object.

```
{
  "data": [
    {
      "id": <auto generated id>,
      "notification_type": "gameroom_proposal",
      "requester": <Alice’s public key>,
       “requester_node_id”: “acme-node-000”,
      "target": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
      "timestamp": <time entry was created>,
      "read": true
    }
  ]
}
```

## I -5. Behind Scene 5: Bob accepts Alice’s invitation

When Bob accepts Alice's invitation, the Bubba Bakery UI sends his "yes" vote to the Gameroom REST API, which forwards it to Splinter REST API. After the vote is validated, the admin service creates the circuit in Splinter state and tells the Gameroom daemon that the circuit is available.

Next, the Bubba Bakery admin service notifies the Acme node that it's ready to create services. After the Acme node responds (described in "Behind Scene 6"), the Bubba Bakery admin service initializes its scabbard service for the new gameroom. Scabbard is the Splinter service for Gameroom that includes the Sawtooth Sabre transaction handler and Hyperledger Transact, using two-phase commit consensus to agree on state. Gameroom uses this service to store the XO smart contract and manage XO state.

Finally, the Gameroom daemon updates the gameroom status in its local database from "Accepted" to "Ready".

### I -5.1. Bubba Bakery UI submits Accept Invitation request to Gameroom REST API

When Bob clicks on the Accept button, the Bubba Bakery Gameroom UI sends a vote (also called a "circuit vote request") to the Gameroom REST API.

```
POST /proposals/vote
{
    "circuit_id": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
     "circuit_hash":"8cd2bfcf3f4259b9785a723e19b4bb4d5cc0206e",
    "vote": "Accept"
 }
```

## I -5.2. Gameroom REST API submits Proposal Accept request to Splinter REST API

1. When the Bubba Bakery Gameroom REST API receives the vote request, it uses that information to create a `CircuitManagementPayload`, which will eventually be sent to the Bubba Bakery Splinter daemon.

2. Before the payload can be sent, the Bubba Bakery UI must sign the bytes of the `CircuitManagementPayload` header.

   The following example shows a YAML representation of the `CircuitManagementPayload` that the Gameroom REST API creates.

   ```
   Circuit proposal vote:
   ---
   circuit_id: gameroom::acme-node-000::bubba-node-000::<UUIDv4>
   circuit_hash: <sha256 hash of the circuit definition of the proposed circuit>
   vote: Accept

   Header:
   ---
   Action: CIRCUIT_PROPOSAL_VOTE
   requester: <public key of requester> // left empty by the REST API
   payload_sha512: <sha512 hash of the circuit proposal vote described above>
   requester_id: <ID of the Splinter node that the requester is registered to>

    CircuitManagmentPayload:
    ---
   header: <bytes of header described above>
   circuit_proposal_vote: <circuit proposal vote described above>
   signature: <signature of bytes of the header> // left empty by the REST API
   ```

   Note that the Gameroom REST API does not fill in the `requester` field in the header or the `signature` field in the `CircuitManagementPayload`.

3. The Bubba Bakery Gameroom REST API serializes the payload and sends the bytes as a response to the UI.

   ```
   {
         "data":  { “Payload_bytes”: <bytes of the CircuitManagementPayload> }
   }
   ```

4. After receiving the response from the Gameroom REST API, the Bubba Bakery UI deserializes the `CircuitManagementPayload`. It adds the requester's public key to the header (in this case, Alice is the requester), serializes the header, signs the header bytes, and adds the signature to the payload. Finally, the UI serializes the complete payload.

5. The Bubba Bakery UI submits the bytes of the signed payload to the Gameroom REST API.

   ```
   POST /submit
   Content-Type: application/octet-stream

   <bytes of the signed CircuitManagementPayload>
   ```

6. The Bubba Bakery Gameroom REST API forwards the payload to the Bubba Bakery Splinter daemon.

   ```
   POST /admin/submit
   Content-Type: application/octet-stream

   <bytes of the signed CircuitManagementPayload>
   ```

7. The Splinter REST API responds with the status "`202 Accepted`" and the Bubba Bakery admin service processes the vote.

8. The Bubba Bakery REST API forwards the "`202 Accepted`" response to the Bubba Bakery UI.

## I -5.3. Bubba Bakery node votes "yes" (validates the vote)

1. The Bubba Bakery admin service (`admin::bubba-node-000`) receives the `CircuitManagmentPayload` containing a `CircuitProposalVote` from the Splinter REST API, and adds it to its pending circuit payloads queue.

```
---
CircuitProposalVote:
        circuit_id: gameroom::acme-node-000::bubba-node-000::<UUIDv4>
        circuit_hash: <hash of circuit>
         vote: ACCEPT
```

2. The Bubba Bakery admin service validates `CircuitManagementPayload` using the same validation process described earlier (see section I -2.5.1, step 1).

   It also validates the provided payload (a `CircuitProposalVote`) using vote-specific validation rules. `CircuitProposalVote` is valid if the following things are true:

   - There is a pending CircuitProposal in admin state with the same circuit ID as in the CircuitProposalVote
   - The hash of the stored pending CircuitProposal is the same as the `circuit_hash` in CircuitProposalVote
   - CircuitProposalVote vote field is set to either the Vote enum ACCEPT or REJECT
   - The public key belongs to a node in the circuit, and that node is allowed to submit the vote (see below).
   - CircuitProposal does not already contain a vote from that node

   To verify the node's public key and voting permission, the admin service checks the key registry and key permissions manager.

   - The key registry provides a way to look up details about a public key used to sign a circuit proposal: the requester node ID (the "home node" of the requester and location of that user's public key) and arbitrary metadata (represented as key/value string pairs).
   - The key permissions manager checks that a public key is authorized in a specific role. In the case of circuit proposal votes, the signing public key must be authorized for the role "voter".

3. If the request is valid, the admin service makes a copy of the existing `CircuitProposal`, adds a vote record to it, and stores it in `AdminServiceShared.pending_changes`. The protobuf is represented in YAML format.

```
---
CircuitProposal:
    proposal_type: CREATE
    circuit_id: gameroom::acme-node-000::bubba-node-000::<UUIDv4>:
    circuit_hash: <hash of circuit>
```

```
circuit_proposal:
  circuit:
      gameroom::acme-node-000::bubba-node-000::<UUIDv4>:
          auth: trust
          members:
              acme-node-000:
                 endpoints:
                     - tls://splinterd-node-acme:8044
              bubba-node-000:
                 endpoints:
                     - tls://splinterd-node-bubba:8044
          roster:
              gameroom_acme-node-000:
                  service_type: scabbard
                  allowed_nodes:
                     - acme-node-000
                  arguments:
                    - peer_services:
                        - gameroom_bubba-node-000
                      admin_keys:
                        - <acme gameroomd public key>
              gameroom_bubba-node-000:
                  service_type: scabbard
                  allowed_nodes:
                     - acme-node-000
                  arguments:
                    - peer_services:
                        - gameroom_acme-node-000
                      admin_keys:
                        - <acme gameroomd public key>
          persistence: any
          durability: none
          routes: require_direct
          circuit_management_type: gameroom
votes:
  - public_key: <voter’s public key>
    vote: ACCEPT
    voter_node_id: bubba-node-000
requester: <public key of requester>
requester_node_id: acme-node-000
```

4. The admin service creates a new consensus `Proposal` for the updated `CircuitProposal`. (See section  I -2.5.1, step 3, for the Proposal description.)

## I -5.4. Bubba Bakery node sends proposal accept vote to Acme node

1. After the Bubba node creates the updated `CircuitProposal`, the `CircuitManagmentPayload` is sent to the other members defined in the circuit -- specifically, the admin service on the Acme node (`admin::acme-node-000`).

2. The Acme admin service receives the `CircuitMangementPayload` containing the `CircuitProposalVote` (as described in section I -5.3, step 1), validates the payload (see section I -5.3, step 2), and creates an updated `CircuitProposal` (see section I -5.3, step 3).

3. The nodes use consensus to agree to accept or reject the circuit proposal. See Appendix B for more information on consensus agreement.

4. After consensus has completed its agreement on the proposal, it notifies the Bubba Bakery admin service that both nodes have accepted the proposal. The Bubba Bakery admin service then commits the updated `CircuitProposal`.

## I -5.5. Bubba Bakery admin service checks for approval and creates a circuit

1. When the `CircuitProposal` is committed, the Bubba Bakery admin service checks to see if it contains the required number of `ACCEPT` votes to be added to Splinter state (the `SplinterState` struct), where active circuits are stored.

   The `CircuitProposal` must have a `VoteRecord` with a vote of `ACCEPT` from every member of the proposed circuit definition, except for the requester (because submitting a circuit proposal counts an accept vote).

2. If a vote exists for every member, the Bubba Bakery admin service adds the circuit defined in the `CircuitProposal` to Splinter state. Once in Splinter state, the circuit is ready to accept service connections and be used for communication.

3. After the circuit has been created, the Bubba Bakery admin service creates the scabbard service using the service orchestrator (described in section I -5.8).

## I -5.6. Bubba Bakery admin service notifies Gameroom daemon of new circuit

1. The Bubba Bakery admin service notifies the Bubba Bakery application authorization handler that the circuit has been accepted.

```
{
   “eventType”: “ProposalAccepted”,
   “message”: {
     “proposal_type”: “Create”,
     “circuit_id”: “gameroom::acme-node-000::bubba-node-000::<UUIDv4>”,
     “circuit_hash”: “...”,
     “circuit”: {
       "circuit_id": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
       "authorization_type": "Trust",
       "members": [{
         "node_id": "acme-node-000",
         "endpoint": "tls://splinterd-node-acme:8044"
          },
         {
         "node_id": "bubba-node-000",
         "endpoint": "tls://splinterd-node-bubba:8044"
          }
       ],

       "roster": [{
           "service_id": "gameroom_acme-node-000",
           "service_type": "scabbard",
           "allowed_nodes": [ "acme-node-000"]
           "arguments": {
               "peer_services": [ "gameroom_bubba-node-000" ]
               "admin_keys": [ "<acme gameroomd public key>" ]
               }
           },
           {
           "service_id": "gameroom_bubba-node-000",
           "service_type": "scabbard",
           "allowed_nodes": [ "bubba-node-000"]
           "arguments": {
               "peer_services": [ "gameroom_acme-node-000" ]
               "admin_keys": [ "<acme gameroomd public key>" ]
               }
           }
       ],
       "circuit_management_type": "gameroom",
       "application_metadata": [...],
       "persistence": "Any",
          "durability": "None",
          "routes": "Any"
         },
       “vote_record”: [{
         “public_key”: “<publickeyofvoter>”,
         “vote”: “Accepted”
         "voter_node_id": “bubba-node-000”
       }],
       “requester”: “<publickeyofrequester>”
       “requester_node_id”: “acme-node-000”

     }
   }
```

2. When the application authorization handler receives this message, it updates the database to change the status of the proposal, gameroom, members and services from “Pending” to “Accepted”.

   At the end of the database transaction, the gameroom database has the following updates:

   - `gameroom` table:

| circuit_id | authorization_type | persistence | routes | durability |
|---|---|---|---|---|
| gameroom::acme-node-000::bubba-node-000::<UUIDv4> | Trust | Any | Any | None |

| circuit_management_type | alias | status | created_time | updated_time |
|---|---|---|---|---|
| gameroom | Acme + Bubba | accepted | <time entry was created> | <time status was updated> |

   - `gameroom_member` table:

| id | circuit_id | node_id | endpoint |
|---|---|---|---|
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | acme-node-000 | tls://splinterd-node-acme:8044 |
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | bubba-node-000 | tls://splinterd-node-bubba:8044 |

| status | created_time | updated_time |
|---|---|---|
| accepted | <time entry was created> | <time the status was updated> |
| accepted | <time entry was created> | <time the status was updated> |

- `gameroom_service` table:

| id | circuit_id | service_id | service_type |
|---|---|---|---|
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | gameroom_acme-node-000 | scabbard |
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | gameroom_bubba-node-000 | scabbard |

| allowed_nodes | arguments | status | created_time | updated_time |
|---|---|---|---|---|
| {"acme-node-000"} | "peer_services": [ "gameroom_bubba-node-000" ], "admin_keys": …. | accepted | <time entry was created> | <time status was updated> |
| {bubba-node-000} | "peer_services": [ "gameroom_bubba-node-000" ], "admin_keys": …. | accepted | <time entry was created> | <time status was updated> |

- `gameroom_proposal` table:

| id | circuit_id | proposal_type | circuit_hash |
|---|---|---|---|
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | Create | <hash of circuit definition> |

3. Finally, the application authorization handler updates the `gameroom_notification` table to tell the UI that the gameroom proposal has been accepted.

| id | notification_type | requester | requester_node_id |
|---|---|---|---|
| <auto generated id> | proposal_accepted | <Bob's public key> | bubba-node-000 |

| target | created_time | read |
|---|---|---|
| gameroom::acme-node-000::bubba-node-000::<UUIDv4> | <time entry was created> | false |

### I -5.7. Bubba Bakery admin service sends "ready to create services" to Acme

Before the Bubba Bakery admin service can initialize its scabbard service on the new circuit, it needs to know that the Acme Splinter node has also created the circuit on the Acme side (added the circuit that is defined in the `CircuitProposal` to Splinter state). The Acme process will be described in the next chapter (section I -6).

This information is required because when a Splinter service connects to its own Splinter node (the node that it is allowed to connect to), that Splinter node sends a message to the other connected Splinter nodes on the new circuit that the service is available. This message cannot be sent until the Splinter node (in this case, the Acme node) has created the circuit.

If the circuit was not yet created on the other Splinter node (or nodes), the message would be dropped. This node would not be able to communicate with the other node's service after the circuit is created, because it wouldn't know where that service exists.

1. To notify the Acme admin service that the Bubba Bakery node is ready to initialize its service, the Bubba Bakery admin service sends an `AdminMessage` with the message type `MEMBER_READY` and a "member ready" message that contains the circuit ID and Bubba Bakery's node ID.

```
---
admin_message:
    message_type: MEMBER_READY,
    member_ready:
        circuit_id: gameroom::acme-node-000::bubba-node-000::<UUIDv4>
        member_node_id: bubba-node-000
```

2. The Bubba Bakery admin service waits for Acme to respond with a "member ready" message. (The next section describes how the Bubba Bakery node initializes its services.)

## I -5.8. Bubba Bakery admin service initializes scabbard service

After the circuit is created (described in section I -5.5) and all members are ready to create services (covered in section I -5.7), the Bubba Bakery admin service makes a call to the service orchestrator to initialize the scabbard service for the new gameroom. As described above, scabbard is the Splinter service for Gameroom that includes the Sawtooth Sabre transaction handler and Hyperledger Transact, using two-phase commit consensus to agree on state. Gameroom uses this service to store the XO smart contract and manage XO state.

1. The Bubba Bakery admin service checks which services are allowed to run on its node. In this case, the Bubba Bakery node (`bubba-node-000`) is allowed to run the `scabbard` service with service ID `gameroom_bubba-node-000`.

2. The admin service creates a `ServiceDefinition` for `gameroom_bubba-node-000`, which is the scabbard service on the Bubba Bakery Splinter node.

```
ServiceDefinition {
    circuit: "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
    service_id: "gameroom_bubba-node-000",
    service_type: "scabbard",
}
```

3. The admin service passes the service definition, along with the arguments defined in the `CircuitProposal` for that service, to the service orchestrator's `initialize_service` method to initialize the scabbard service.

4. `ServiceOrchestrator` uses structs that implement the `ServiceFactory` trait to create new services. An orchestrator can have multiple factories. First, the orchestrator must determine which factory can create a scabbard service (in this case, the `ScabbardFactory`). Then the orchestrator creates a new instance of the scabbard service using that factory, the service definition, and the service arguments.

5. After the scabbard service has been created, the orchestrator starts the service and adds it to its internally managed list of services. When starting the service, the orchestrator creates a `StandardServiceNetworkRegistry` (used to register the service with the node) and passes it to the service; the scabbard service then registers, which provides it with a `StandardServiceNetworkSender` that it will use to send direct messages to other services.

## I -5.9. Bubba Bakery Gameroom daemon updates gameroom status in database

At this point, the new service is running and ready to receive smart contracts.

1. The Bubba Bakery admin service sends a CircuitReady notification to the Gameroom daemon's application authorization handler to let it know that the circuit is created and its services are ready.

```
{
   “eventType”: “CircuitReady”,
   “message”: {
     “proposal_type”: “Create”,
     “circuit_id”: “gameroom::acme-node-000::bubba-node-000::<UUIDv4>”,
     “circuit_hash”: “...”,
     “circuit”: {
       "circuit_id": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
       "authorization_type": "Trust",
       "members": [{
         "node_id": "acme-node-000",
         "endpoint": "tls://splinterd-node-acme:8044"
          },
         {
         "node_id": "bubba-node-000",
         "endpoint": "tls://splinterd-node-bubba:8044"
          }
       ],

       "roster": [{
           "service_id": "gameroom_acme-node-000",
           "service_type": "scabbard",
           "allowed_nodes": [ "acme-node-000"]
           "arguments": {
               "peer_services": [ "gameroom_bubba-node-000" ]
               "admin_keys": [ "<acme gameroomd public key>" ]
               }

           },
           {
           "service_id": "gameroom_bubba-node-000",
           "service_type": "scabbard",
           "allowed_nodes": [ "bubba-node-000"]
           "arguments": {
               "peer_services": [ "gameroom_acme-node-000" ]
               "admin_keys": [ "<acme gameroomd public key>" ]
               }

           }
       ],
       "circuit_management_type": "gameroom",
       "application_metadata": [...],
       "persistence": "Any",
       "durability": "None",
       "routes": "Any"
      },
     “vote_record”: [{
       “public_key”: “<publickeyofvoter>”,
       “vote”: “Accepted”
       "vote_node_id": "bubba-node-000"
     }],
     “requester”: “<publickeyofrequester>”
     "requester_node_id": "acme-node-000"
    }
 }
```

2. When the authorization handler receives the `CircuitReady` message, it changes the status of the gameroom in the database from “Accepted” to “Ready”.

3. The authorization handler creates a new WebSocket to listen for events from the scabbard service. These events are then captured, parsed, and uploaded into the gameroom database. The process of reading state-change updates from Splinter and uploading them to a local database is called "state delta export" and is done by the `XoStateDeltaProcessor`.

   The `XoStateDeltaProcessor` consumes `StateChangeEvent payloads`, such as this example:

```
      {
        “type”: “Set”,
        “message”: {
          “key”: “<xo_address>”
          “value” [<bytes>]
        }
      }
```

   The bytes in value field are deserialized into the following CSV-format representation of the XO game state:

```
      “<game-name>,<game-board>,<game-state>,<player1-key>,<player2-key>”
```

   For more information on the XO game state, see Appendix D.

At this point, the circuit (Alice and Bob's gameroom) is ready. Next, the Acme Gameroom daemon must submit the XO smart contract, which is the last step before the gameroom is ready for Alice and Bob to play games. See section  I -6.6 for an explanation of this process.

## I -6. Behind Scene 6: Alice sees that Bob accepted her invitation

Most of the steps this scene are similar to the Bubba Bakery steps described earlier. However, one activity is unique — because the Acme node (Alice's node) requested the new circuit, it is responsible for submitting the XO smart contract that will allow Alice and Bob to play tic tac toe in the new gameroom.

### I -6.1. Acme admin service receives CircuitProposalVote from Bubba Bakery admin service

When the Acme Splinter node receives the `CircuitMangementPayload` network message containing the Bubba Bakery `CircuitProposalVote` (sent in section I -5.4), it "unwraps" the message with a series of dispatchers. See section I -2.5.3 for the details of this process.

As described in section I -5.4, step 3, both nodes use consensus to agree on the circuit proposal, After they agree, the Acme node commits the `CircuitProposal`.

### I -6.2 Acme admin service checks for approval and creates circuit

When the `CircuitProposal` is committed, the Acme admin service checks that the Bubba Bakery node has voted "yes" (see section I -5.3), then creates the circuit (adds the new circuit to Splinter state). For the details of this process, see section I -5.5.

### I -6.3. Acme admin service notifies Acme Gameroom daemon of new circuit

Once the circuit has been created, the Acme admin service tells the Acme application authorization handler that the circuit has been accepted. When the application authorization handler receives this message, it updates the database to change the status of the proposal, gameroom, members and services from “Pending” to “Accepted”.  See section I -5.6 for the details of this process.

### I -6.4. Acme admin service tells Bubba Bakery that it is ready to create services

The Acme node notifies the Bubba Bakery node that it is ready to initialize the Acme scabbard service by sending an an `AdminMessage` with the message type `MEMBER_READY` and a "member ready" message that contains the circuit ID and Acme's Splinter node ID.

The Acme node waits for all members to report a "member ready" message before proceeding. For the details, see section I -5.7.

## I -6.5. Acme admin service creates scabbard service via service orchestration

After the Acme node learns that all members are ready to create services, the Acme admin service makes a call to the service orchestrator to initialize the scabbard service for the new gameroom. See section  I -5.8 for the details of this process.

## I -6.6. Acme Gameroom daemon submits Sabre transactions to add XO smart contract

The Acme Gameroom daemon submits the XO smart contract by using the scabbard REST API that is exposed by the Splinter daemon, `splinterd`.

1. When the Acme Gameroom daemon's application authorization handler receives the `CircuitReady` notification from the admin service, it subscribes to scabbard and starts listening for scabbard events. See Appendix C for the registration (event subscription) process.

   ```
   GET <circuit_id>/<service_id>/ws/subscribe
   ```

2. After a connection has been established, the application authorization handler prepares the XO (tic tac toe) business logic by submitting the XO smart contract to the Acme scabbard service.

   The Acme gameroom daemon gets the following information from the `CircuitProposal` that was just accepted (as described in section  I -5.9):

   - `circuit_id`: unique ID of the new circuit that includes a version 4 UUID, such as `gameroom::acme-node-000::bubba-node-000::<UUIDv4>`
   - `service_id`: ID of the scabbard service that is running on the local Splinter node; for example: `gameroom_acme-node-000`
   - `scabbard_admin_keys`: scabbard admin keys that are stored in the circuit proposal's application metadata

   The scabbard admin keys in Gameroom's application metadata specify who is allowed to add or modify smart contracts. When the circuit is initially defined (see section  I -2.3), the Gameroom daemon that creates the circuit definition (in this case, Acme's `gameroomd`) specifies its own public key as the scabbard admin. Since the Acme Gameroom daemon has the only key that's authorized to add smart contracts, it is responsible for adding the XO smart contract.

3. To add the XO smart contract, the Acme Gameroom daemon creates a series of transactions to set the permissions surrounding the smart contract and to submit the smart contract itself. For more information, see the Sawtooth Sabre documentation.

4. The Acme Gameroom daemon bundles these transactions into a batch, serializes the batch, and submits the serialized batch to the scabbard service on its local Splinter node:

   ```
   POST /scabbard/<circuit_id>/<service_id>/batches
   <serialized batch>
   ```

   For more information about batches, see "Transactions and Batches" in the Sawtooth Architecture documentation.

5. When the Acme scabbard service receives this batch, it must agree with the Bubba Bakery scabbard service to submit the smart contract. The Acme scabbard service performs the following steps:
   a. Deserializes the batch
   b. Shares the batch with the other scabbard services in the circuit (in this case, the Bubba Bakery scabbard service)
   c. Creates a consensus proposal to send to the scabbard services of the other nodes (in this case, Bubba Bakery's scabbard service) to agree on the batch
   d. Waits for consensus to agree on the batch, then commits it to scabbard state. For information on consensus, see Appendix B.

After the scabbard services on both nodes have committed the smart contract, the Acme Splinter node is done setting up the gameroom.

### I -6.7. Both Gameroom daemons update gameroom status in database

The application authorization handler listens for scabbard events using its state delta processor, `XoStateDeltaProcessor`. When the state delta processor receives an event with the address of the uploaded XO contract it then sets the status of the gameroom to “`circuit_active`”

```
StateChangeEvent containing contact address
{
  “type”: “Set”,
  “message”: {
    “key”: “<xo_contract_address>”
    “value” [..]
  }
}
```

At this point, the state delta processor will begin listening for XO game state changes (defined in Appendix D).

### I -6.8. Acme Gameroom daemon notifies Acme UI

After the state delta processor sets the status of the gameroom to “`circuit_active`”, the Acme Gameroom application authorization handler adds a new entry to the `gameroom_notification` table:

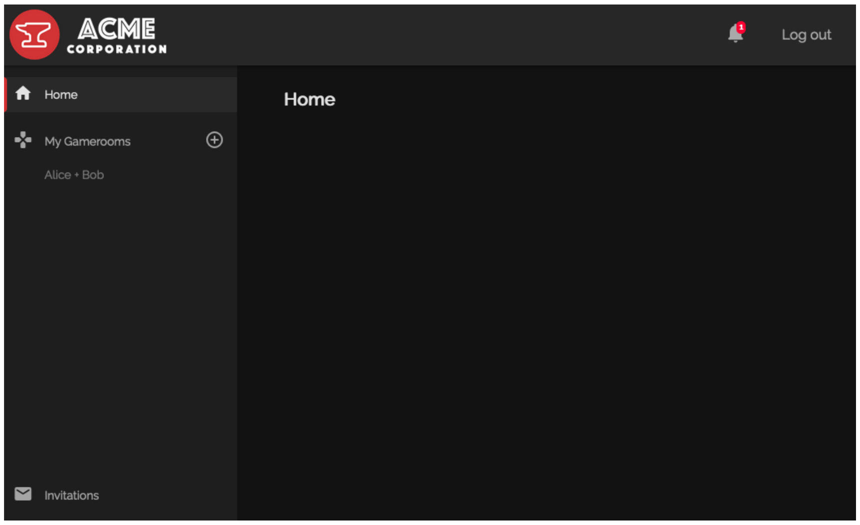
| id | notification_type | requester | requester_node_id |
|---|---|---|---|
| `<auto generated id>` | `circuit_active` | `<Alice’s public key>` | `acme-node-000` |

| target | created_time | read |
|---|---|---|
| `gameroom::acme-node-000::bubba-node-000::<UUIDv4>` | `<time entry was created>` | `false` |

This notification is pushed to the Acme UI in the same way as the “`gameroom_proposal`” notification. (See section I -2.9.)

## I -6.9. Alice sees notification that new gameroom is ready

After the notification is pushed to the Acme UI, Alice sees a new notification. The new gameroom also appears on the dashboard menu.

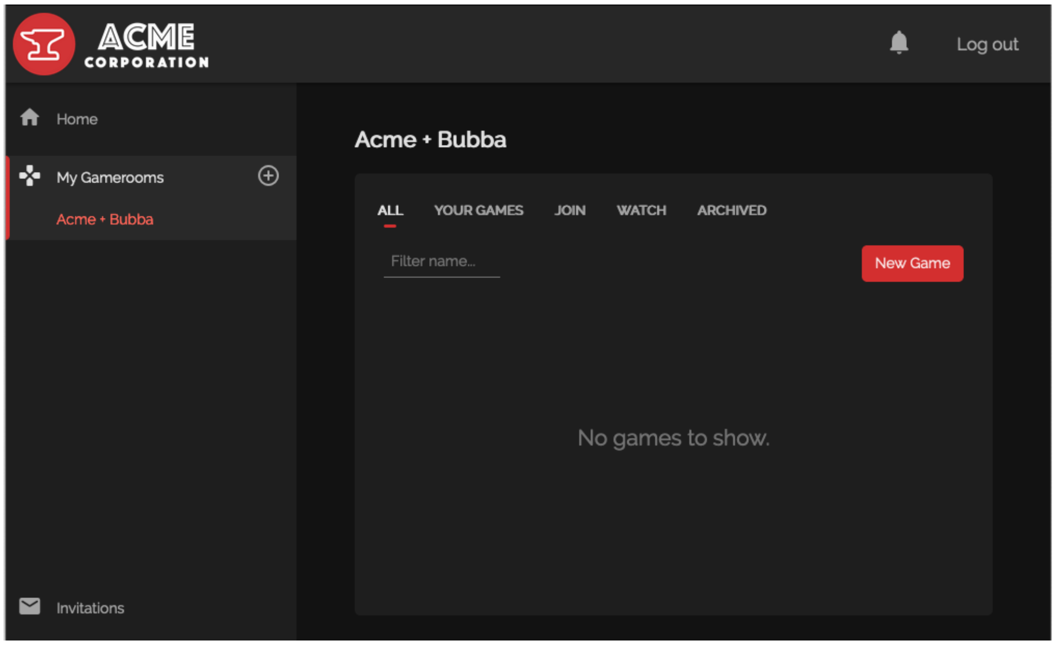


When Alice clicks on the notification, she sees the details page for the new gameroom. (See Behind Scene 4, Bob Checks his Notifications.)

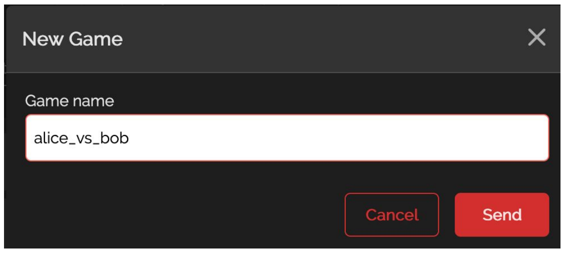
# Act Ⅱ: Alice and Bob Play XO

## Scene 1: Alice creates a new XO game

Alice returns from lunch and unlocks her computer. The GAMEROOM TAB is still displayed.



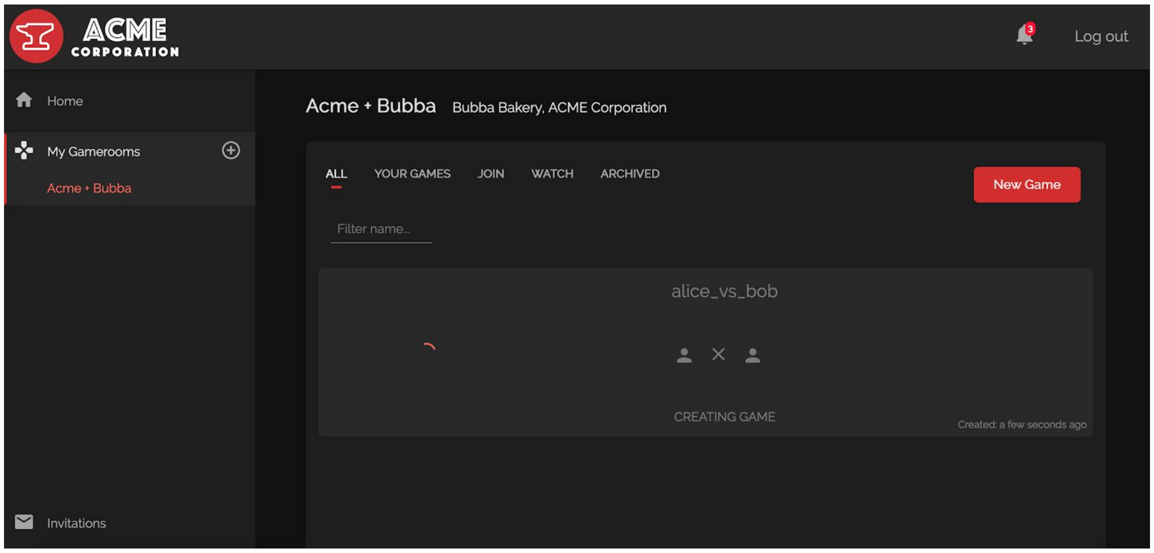
Alice wants to play an XO game. She clicks **New Game**. Alice sees the NEW GAME DIALOG.



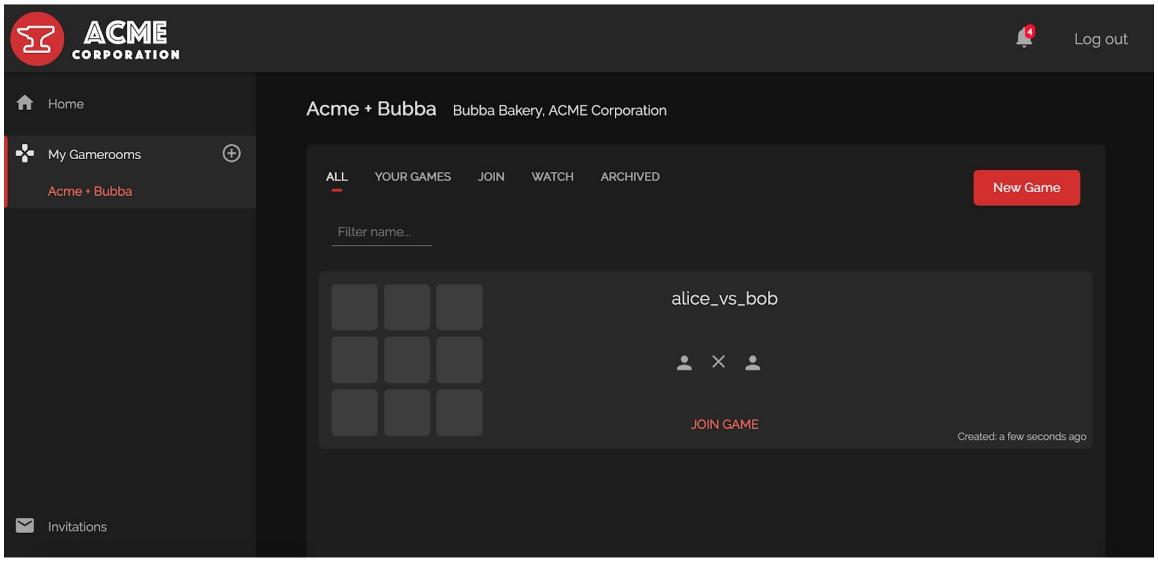
alice_vs_bob

Alice enters "**alice_vs_bob**" as the game name. She clicks the SEND BUTTON.

The Gameroom application starts creating the game. Alice sees a new "alice_vs_bob" entry with a SPINNER and the message "**CREATING GAME**".

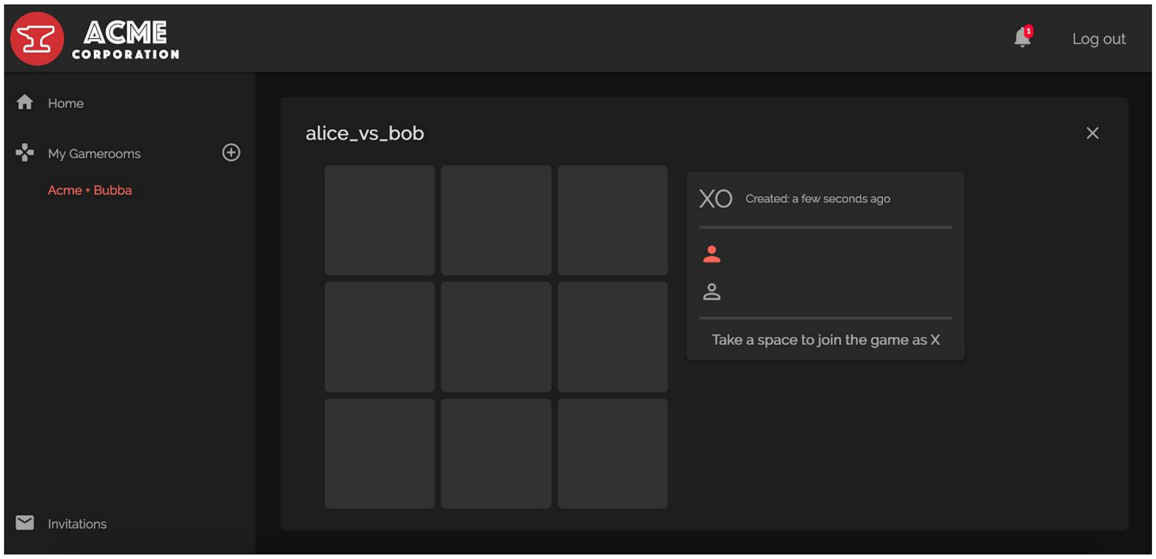


A short period of time passes. When the game has been fully created, Alice sees a blank game board and the message "**JOIN GAME**".



## Scene 2: Alice makes the first move

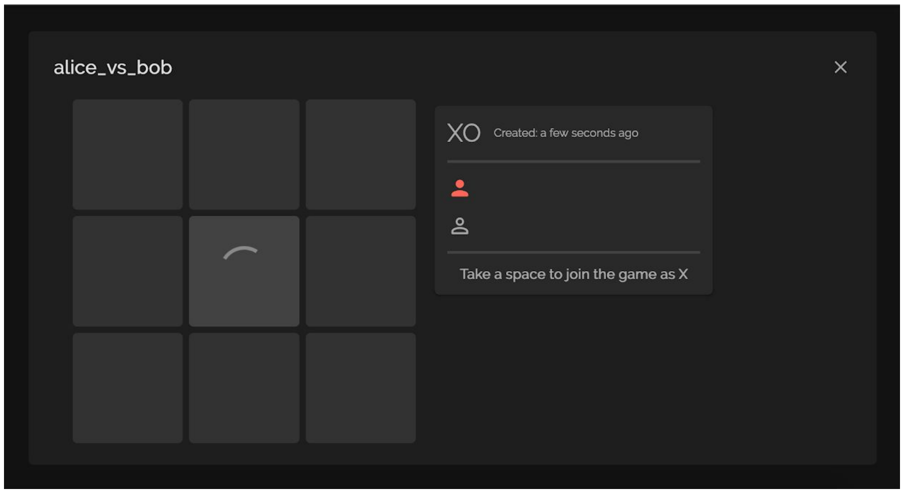
Alice clicks **JOIN GAME** on the newly created game. The **"alice_vs_bob"** game board appears.



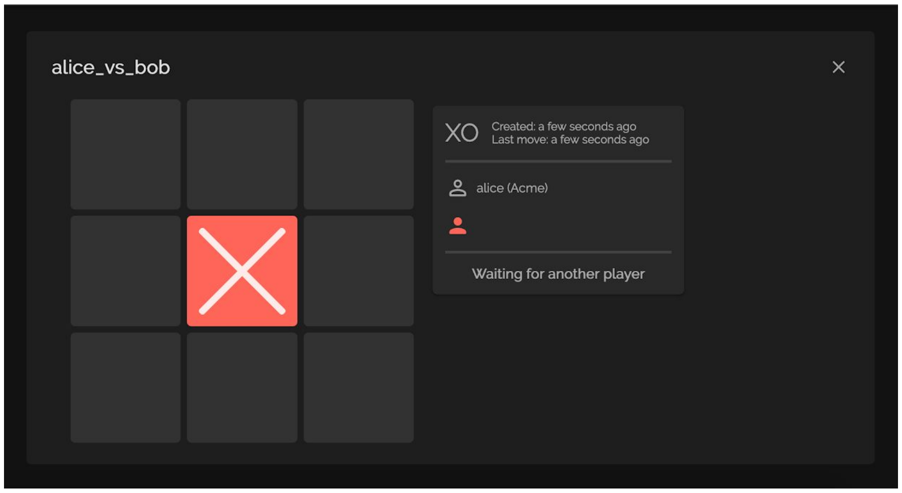
Alice stretches her hands and rubs her neck, preparing for her first move.

She clicks the center spot on the board.

While the Gameroom application processes her move, Alice sees a spinner in the center spot.


alice_vs_bob
XO
Created: a few seconds ago
Take a space to join the game as X

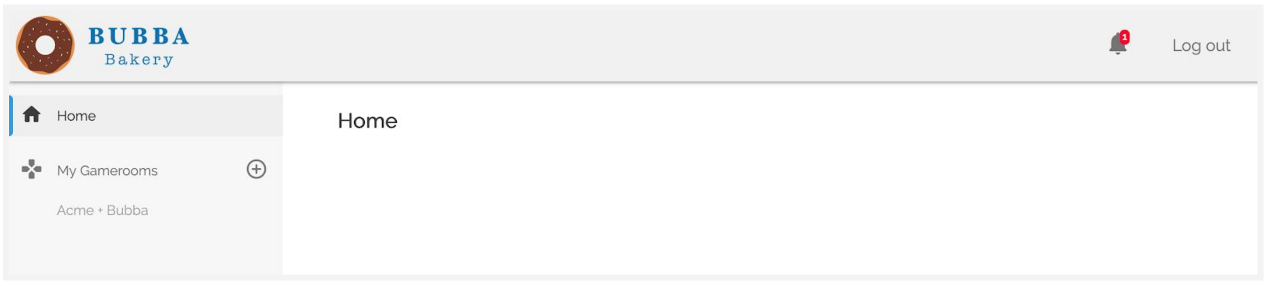
Time passes. The spinner disappears and an X appears in its place. Alice's first move has been accepted.


alice_vs_bob
XO
Created: a few seconds ago
Last move: a few seconds ago
alice (Acme)
Waiting for another player

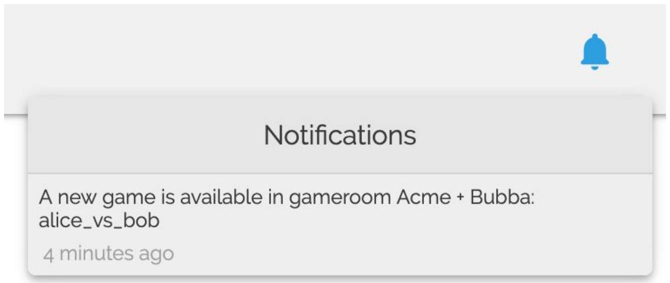
Now she must wait for Bob’s first move.

## Scene 3: Bob takes a turn

Eventually, Bob sees a RED NUMBER 1 on his bell icon, which means that he has a notification.



Bob clicks on the bell icon. He sees that the game **alice_vs_bob** is available in the **Acme + Bubba** gameroom.

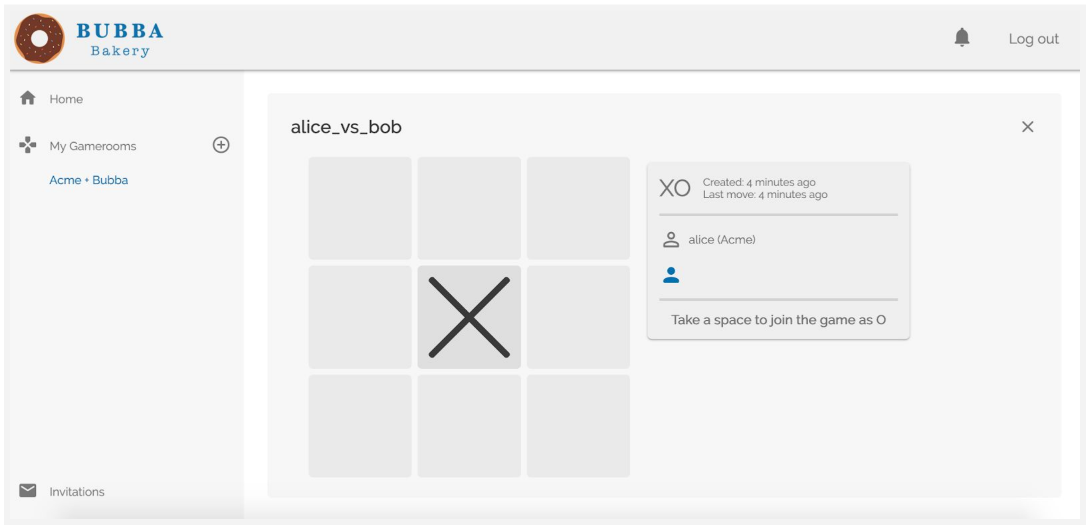

Notifications

A new game is available in gameroom Acme + Bubba: alice_vs_bob

4 minutes ago

He clicks on the notification text. The **alice_vs_bob** game board is displayed.

Bob sees that Alice has taken the center space.



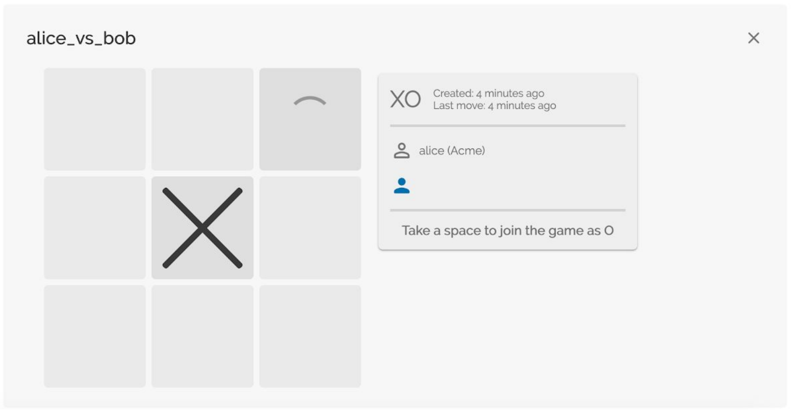
Created: 4 minutes ago
Last move: 4 minutes ago

alice (Acme)

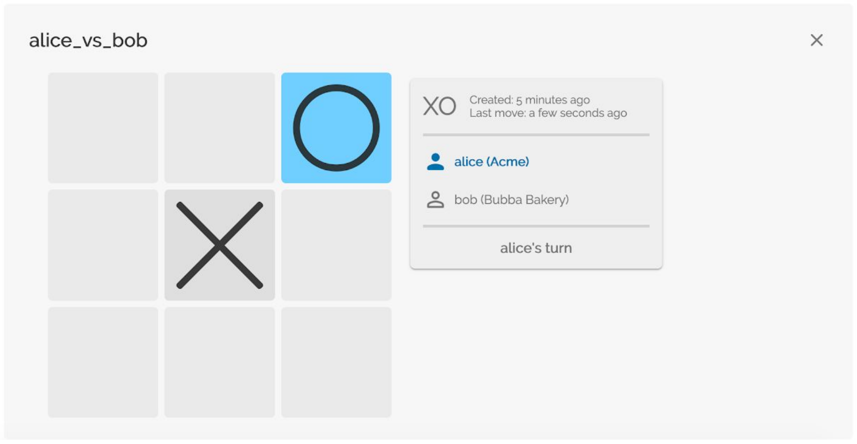
Take a space to join the game as O

Muttering to himself, Bob makes his first move: he takes the top right corner.

While the Gameroom application processes his move, Bob sees a spinner in that space.



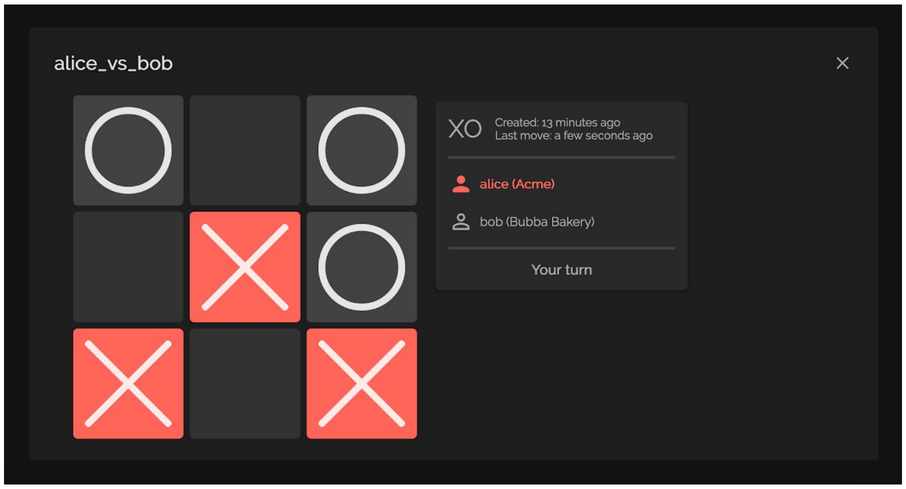
Soon, the spinner disappears. Bob's first move has been accepted.



The game continues, slowly, as Alice and Bob carefully analyze each move.

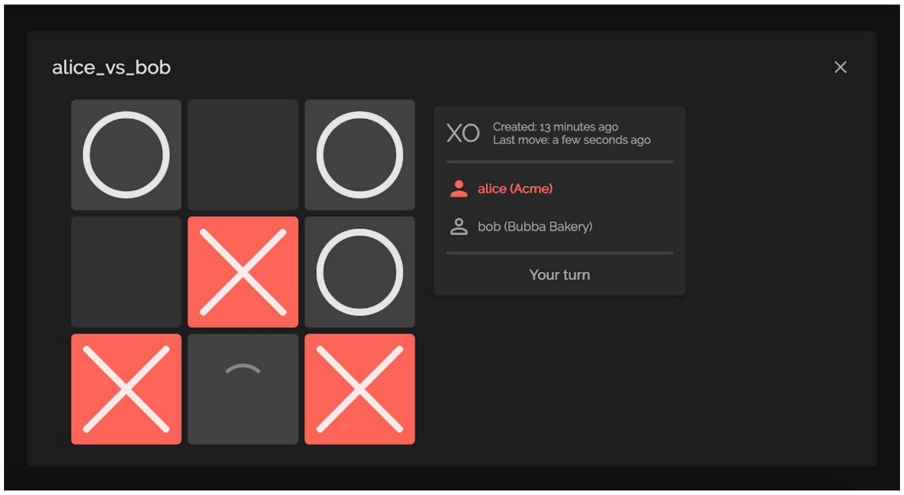
## Scene 4: Alice wins the game

It's the last move of the game.

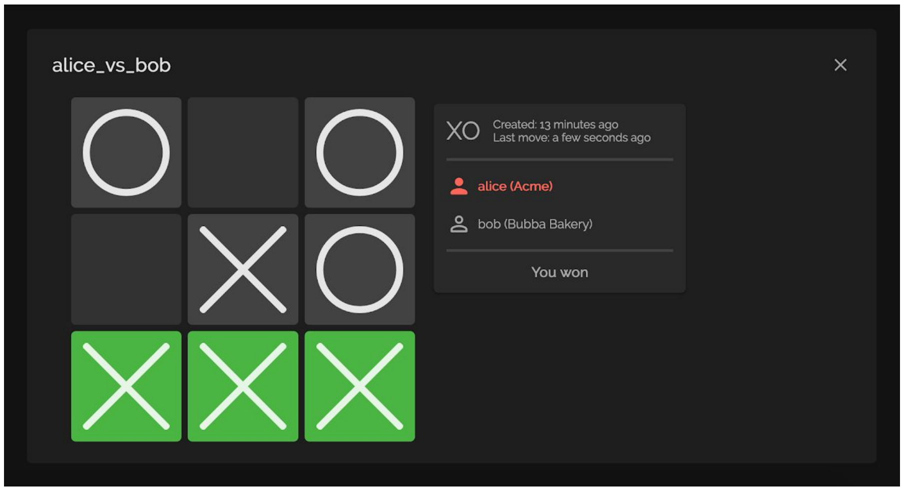


Alice is biting her nails. Bob wipes his forehead.

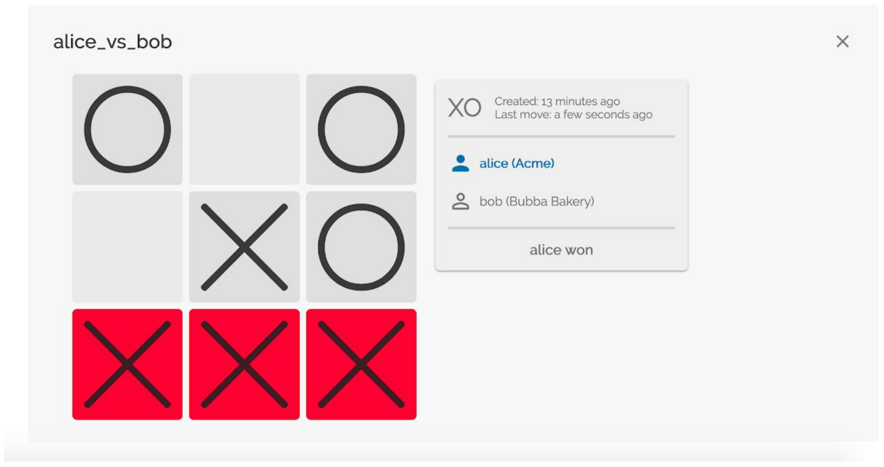
Alice clicks on the winning spot. Suspense builds while she watches the spinner.



The spinner disappears. Alice wins the game!



Bob sees Alice's winning move as a row of red Xs.



**Scene 5: The triumph of Alice**

In Alice's office, we hear CHEERS IN THE BACKGROUND.

Alice turns around and sees a crowd of co-workers who are celebrating her win.

**Scene 6: The tragedy of Bob**

In Bob's office, there's a loud crash, then a scream. CUT TO BLACK.

# Behind the Scenes: A Look at Act Ⅱ, Alice and Bob Play XO

This section describes what really happens during Act Ⅱ. Although the actions of creating a game and making moves are different, the underlying functions are similar to the process of creating a gameroom.

## Ⅱ-1. Behind Scene 1: Alice creates a new XO game

In Scene 1, Alice uses the Gameroom UI to create a new game named `alice_vs_bob` in the `acme + bubba` gameroom. This section explains how the new game request is handled.

### Ⅱ-1.1. Acme client sends 'create game' request to Gameroom REST API

1. When Alice clicks **Create** in the New Game screen, the Acme client creates an XO transaction request payload for the creation request. (See Appendix D.2 for information about the XO transaction request format.)

   ```
   alice_vs_bob,create,
   ```

2. The Acme client wraps this transaction request payload in a `sabre_payload` message.

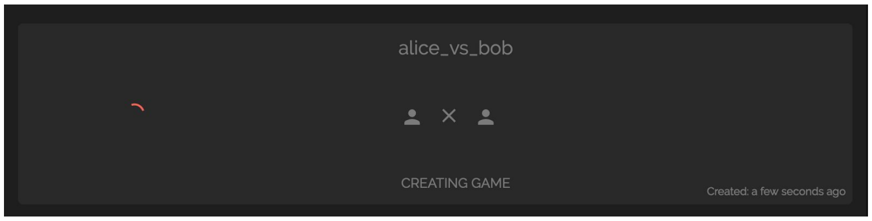
   ```
   ---
   sabre_payload:
     action: "EXECUTE_CONTRACT"
     execute_contract:
       name: xo
       version: 0.3.3
       inputs:
         - 5b7349
         - 00ec00
         - 00ec01
         - 00ec02
       outputs:
         - 5b7349
       payload: b"alice_vs_bob,create,"
   ```

3. The Acme client bundles the Sabre payload into a batch, then serializes the batch into an array of bytes. (For more information, see "Transactions and Batches" in the Sawtooth Architecture documentation.)

4. The Acme client sends the batch to the Acme Gameroom REST API.

   ```
   POST /gamerooms/<circuid_id>/batches
   Content-Type: application/octet-stream

   <bytes of the batch containing the Sabre XO transaction>
   ```

5. After the transaction is sent, the Acme Gameroom UI displays a spinner and the message "Creating Game", which shows that the game is in a pending state.



This state continues until the UI is notified that the new game has been committed to state (described in section Ⅱ-1.6).

## Ⅱ-1.2. Acme Gameroom REST API sends ‘create' transaction to Acme scabbard service

The Acme gameroom daemon passes the serialized batch (which contains the XO transaction) to the Acme scabbard service, using the scabbard service's existing REST API route (as described in section Ⅰ-6.6, step 4).

```
POST /scabbard/<circuit_id>/<service_id>/batches
<serialized batch>
```

## Ⅱ-1.3. Scabbard services use consensus to commit the new game

When the Acme scabbard service receives this batch, it commits the batch using a similar process as adding the XO smart contract (described in section Ⅰ-6.6, step 5).

1. The first steps are the same: The Acme scabbard service deserializes the batch, then shares the batch with the Bubba Bakery scabbard service  so that both nodes can use consensus to agree to commit the new game.

2. Next, the scabbard services use the Sabre transaction handler to use the XO smart contract to execute the transactions in the batch.

3. The remaining steps are the same as before: After the nodes agree on state using two-phase commit consensus, both scabbard services commit the new XO game to scabbard state in their local database.

## Ⅱ-1.4. Scabbard services notify Gameroom daemons of state change

After the new XO game has been committed to scabbard state, both scabbard services send the new XO game state to their gameroom daemon via a WebSocket connection.

An XO game state is defined as a string of comma-separated values:

```
“<game-name>,<board-state>,<game-state>,<player1-key>,<player2-key>”
```

The message to the gameroom daemon looks like this:

```
  {
    “eventType”: “Set”,
    “message”: {
        “key”: “<xo game address>”,
        “value”: b“alice_vs_bob,---------,P1-NEXT,,”
    }
}
```

Note that the values for `player1-key` and `player2-key` are empty. These fields are not set until a player makes the first move. (This is a design choice for the XO smart contract; it's not an inherent limitation of Splinter or the Gameroom application.)

## Ⅱ-1.5. Gameroom daemons update gameroom status in database

When a Gameroom daemon receives the message from its scabbard service, the daemon stores the state change in its local database.

1. Because this is a new game, the Gameroom daemon creates a new entry in the `xo_games` table in the database.

   The `xo_games` table has this definition:

```
xo_games (
  id                       BIGSERIAL    PRIMARY KEY,
  circuit_id               TEXT         NOT NULL,
  game_name                TEXT         NOT NULL,
  player_1                 TEXT         NOT NULL,
  player_2                 TEXT         NOT NULL,
  game_status              TEXT         NOT NULL,
  game_board               TEXT         NOT NULL,
  created_time             TIMESTAMP    NOT NULL,
  updated_time             TIMESTAMP    NOT NULL,
  FOREIGN KEY (circuit_id) REFERENCES gameroom(circuit_id) ON DELETE CASCADE
);
```

At the end of this operation, the `xo_games` table has the following entry:

| id | circuit_id | game_name | player_1 |
|---|---|---|---|
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | alice_vs_bob | |

| player_2 | game_status | game_board | created_time | updated_time |
|---|---|---|---|---|
| | P1-NEXT | --------- | <time entry was created> | <time entry was created> |

2. The gameroom daemon also adds a new notification to the `gameroom_notification` table, which indicates that a new game was created.

| id | notification_type | requester | requester_node_id |
|---|---|---|---|
| <auto generated id> | new_game_created:alice_vs_bob | <Alice’s public key> | acme-node-000 |

| target | created_time | read |
|---|---|---|
| gameroom::acme-node-000::bubba-node-000::<UUIDv4> | <time entry was created> | f |

## Ⅱ-1.6. Gameroom REST APIs tell clients that XO game is committed

1. After the Acme and Bubba Bakery Gameroom daemons fill in the `gameroom_notification` tables, the Gameroom REST API uses a WebSocket connection to tell each UI about the new notification.

```
{
   "namespace": "notifications",
   "action": "listNotifications"
}
```

2. When the UI receives that message, it sends a request to the Gameroom REST API to fetch a list of unread notifications from the database tables.

```
GET /notifications
```

3. The Gameroom REST API responds with the list of unread notifications.

```
{
  "data": [
    {
      "id": <auto generated id>,
      "notification_type": "new_game_created:alice_vs_bob",
      "requester": <Alice’s public key>,
      “node_id”: “acme-node-000”,
      "target": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
      "timestamp": <time entry was created>,
      "read": false
    }
  ],
  "paging": {
       "current": "api/notifications?limit=100&offset=0",
       "offset": 0,
       "limit": 100,
       "total": 1,
       "first": "api/notifications?limit=100&offset=0",
       "prev": "api/notifications?limit=100&offset=0",
       "next": "api/notifications?limit=100&offset=0",
       "last": "api/notifications?limit=100&offset=0"
   }
}
```

4. Once the UI receives this notification, it asks the Gameroom REST API to fetch the list of games in the `Acme + Bubba` gameroom.

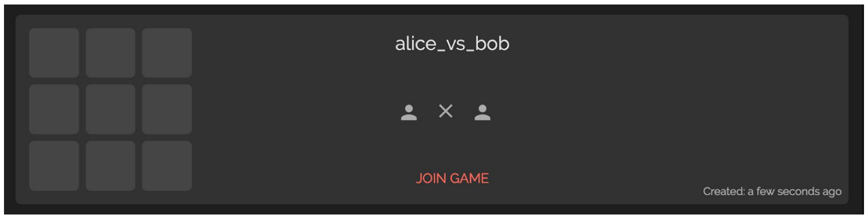
```
GET /xo/gameroom::acme-node-000::bubba-node-000::<UUIDv4>/games
```

5. The Gameroom REST API responds with a list of games that includes the status of each game. At this point, only the new `alice_vs_bob` game exists; no moves have been made.

```
{
   "data": [
        {
           "circuit_id": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
           "game_name": "alice_vs_bob",
           "player_1": "",
           "player_2": "",
           "game_status": "P1-NEXT",
           "game_board": "---------",
           "created_time": <time entry was created>,
           "updated_time": <time entry was created>
       }
   ],
   "paging": {
       "current": "api/xo/gameroom::bubba-node-000::acme-node-000::<UUIDv4>/games?limit=100&offset=0",
       "offset": 0,
       "limit": 100,
       "total": 0,
       "first": "api/xo/gameroom::bubba-node-000::acme-node-000::<UUIDv4>/games?limit=100&offset=0",
       "prev": "api/xo/gameroom::bubba-node-000::acme-node-000::<UUIDv4>/games?limit=100&offset=0",
       "next": "api/xo/gameroom::bubba-node-000::acme-node-000::<UUIDv4>/games?limit=100&offset=0",
       "last": "api/xo/gameroom::bubba-node-000::acme-node-000::<UUIDv4>/games?limit=100&offset=0"
   }
}
```

6. The Acme Gameroom UI checks that the `alice_vs_bob` game is present in the list of games received from the REST API. Because the game is in the list, the UI can now show the game in a "committed" state (ready to play because the "create game" transaction has been committed).

7. The Acme Gameroom UI replaces the spinner with a blank game board.



Alice can now click on the game board to start playing XO.

## Ⅱ-2. Behind Scene 2: Alice makes the first move

Each game move is handled the same way as the "create game" process described in section Ⅱ-1. This section summarizes these steps.

1. The Acme client submits the "take a space" transaction.

   a. When Alice clicks the middle square in the XO board, the Acme client creates an XO transaction request payload for taking the 5th space. See Appendix D.2 for information on XO game moves and the game board.

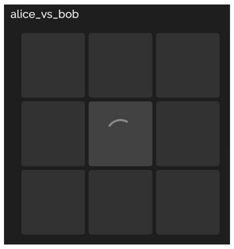
   ```
   alice_vs_bob,take,5,
   ```

   b. As with game creation, the Acme client wraps this transaction request payload in a `sabre_payload` message (see section Ⅱ-1.1, step 2 for the message details).

   c. The Acme client bundles the Sabre payload into a batch, then serializes the batch into an array of bytes. (For more information, see "Transactions and Batches" in the Sawtooth Architecture documentation.)

   d. The Acme client posts the batch to the Acme Gameroom REST API (see the details in section Ⅱ-1.1, step 4).

   e. After the transaction is sent, the Acme Gameroom UI displays a spinner in the center square until it is notified that the game has been updated in state.



2. The Acme Gameroom REST API forwards the XO ‘take’ transaction to the scabbard service (see the details in section Ⅱ-1.2).

3. The Acme and Bubba Bakery scabbard services use consensus (defined in Appendix B) to commit the move (as described in section Ⅱ-1.3).

4. After Alice's first move has been committed to scabbard state, the scabbard services send the new state to their gameroom daemons via a WebSocket connection (as described in section Ⅱ-1.4).

This message includes an updated game board that has Alice's X in the center square of the game board and Alice's public key in the player1 field.

```
    {
      “eventType”: “Set”,
      “message”: {
         “key”: “<xo game address>”,
         “value”: b“alice_vs_bob,----x----,P2-NEXT,<Alice’s public key>,”
      }
  }
```

Note that the last field (the key for player2) is empty. That field will be set when Bob makes his first move.

5. When each Gameroom daemon receives the message, it updates the `alice_vs_bob` entry in the `xo_games` table in the database (this entry was created in section Ⅱ-1.5).

   At the end of the operation, the `xo_games` table looks like this:

| id | circuit_id | game_name | player_1 |
|---|---|---|---|
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | alice_vs_bob | <Alice’s public key> |

| player_2 | game_status | game_board | created_time | updated_time |
|---|---|---|---|---|
| | P2-NEXT | ----x---- | <time entry was created> | <time entry was updated> |

   The gameroom daemon also adds a new notification to the `gameroom_notification` table to indicate that the game was updated.

| id | notification_type | requester | requester_node_id |
|---|---|---|---|
| <auto generated id> | game_updated:alice_vs_bob | <Alice’s public key> | acme-node-000 |

| target | created_time | read |
|---|---|---|
| gameroom::acme-node-000::bubba-node-000::<UUIDv4> | <time entry was created> | f |

6. The Gameroom REST APIs tell the clients that Alice's move has been committed and the XO game state has been updated. This process is the same as in section II-1.6, but the notification details contain information about Alice's move.

    a. After the Acme Gameroom daemon handler fills in the `gameroom_notification` table, the Acme Gameroom REST API uses a WebSocket connection to tell the Acme UI about the new notification (see the details in section Ⅱ-1.6, step 1).

    b. When the Acme UI receives that message, it asks the Gameroom REST API to fetch a list of unread notifications from the database tables (using the same `GET /notifications` request as in section Ⅱ-1.6, step 2).

    c. The Acme Gameroom REST API responds with the list of unread notifications, as described in section Ⅱ-1.6, step 3. At this point, however, the notification type is `game_updated`.

    ```
    {
      "data": [
        {
          "id": <auto generated id>,
          "notification_type": "game_updated:alice_vs_bob",
          "requester": <Alice’s public key>,
           “node_id”: “acme-node-000”,
          "target": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
          "timestamp": <time entry was created>,
          "read": false
        }
      ],
      "paging": {

         … [SNIP] …

       }
    }
    ```

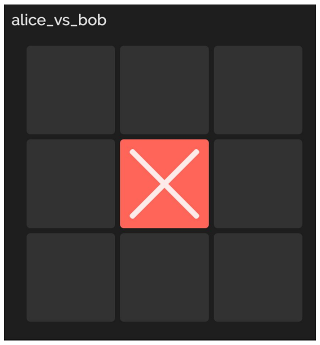
    d. Once the UI receives this notification, it sends a request to the Acme Gameroom REST API to fetch the list of games in the `Acme + Bubba` gameroom (using the same `GET` request as in section Ⅱ-1.6, step 4).

    e. The Acme Gameroom REST API returns the list of games and game data. At this point, the `alice_vs_bob` game data shows that Alice is player_1, player_2 must move next, and Alice's X is in the center square of the game board.

    ```
    {
       "data": [
            {
               "circuit_id": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
               "game_name": "alice_vs_bob",
               "player_1": "<Alice’s public key>",
               "player_2": "",
    ```

```
                "game_status": "P2-NEXT",
                "game_board": "----x----",
                "created_time": <time entry was created>,
                "updated_time": <time entry was updated>
            }
        ],
        "paging": {

          ... [SNIP] ...

        }
    }
```

f. The Acme Gameroom UI now shows that Alice’s move has been committed by replacing the spinner in the center square with a red X.



## Ⅱ-3. Behind Scene 3: Bob takes a turn

In Act Ⅱ, Scene 3, Bob notices his "new game" notification, clicks on it, and is redirected to the game page. The notification process is the same as in Act Ⅰ, section Ⅰ-3, when Bob saw Alice's invitation for the new gameroom. When Bob joins the game, the process is similar to Alice's first move in Act Ⅱ, section Ⅱ-2. This section summarizes the process and highlights the differences.

1. The Bubba Bakery client gets a notification of a new game in the Acme + Bubba gameroom (see the details in section Ⅰ-3.6)

2. Bob checks his notifications as described in section Ⅰ-4.

3. When Bob clicks on his "alice_vs_bob" game notification, he joins the XO game with Alice.

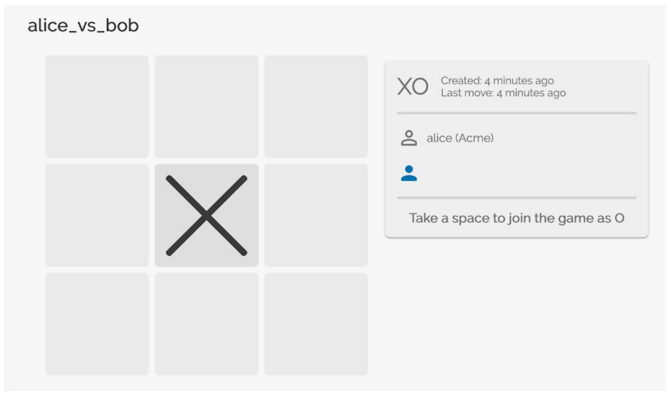
   a. The Bubba Bakery UI makes a call to the Gameroom REST API for the list of existing games in the `Acme + Bubba` gameroom.

   ```
   GET /xo/<circuitID>/games
   ```

   b. The Bubba Bakery Gameroom REST API responds with a list of games. The game data shows the status after Alice's first move, as described in section Ⅱ-2, step 6e.

   ```
   {
      "data": [
           {
              "circuit_id": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
              "game_name": "alice_vs_bob",
              "player_1": "<Alice’s public key>",
              "player_2": "",
              "game_status": "P2-NEXT",
              "game_board": "----x----",
              "created_time": <time entry was created>,
              "updated_time": <time entry was updated>
         }
      ],
      "paging": {

        … [SNIP] …

      }
   }
   ```

c. The Bubba Bakery UI then displays the game board and related information for the `alice_vs_bob` game. Bob sees Alice's X in the center square.



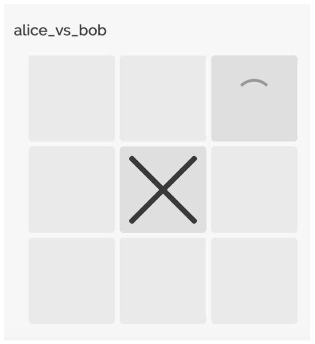
4. When Bob clicks the top right square on the XO board, the Bubba Bakery client starts the process of handling his "take a square" request.

   a. The Bubba Bakery client creates an XO transaction request payload for taking the 3rd square.

   ```
   alice_vs_bob,take,3
   ```

   b. The Bubba Bakery client wraps this transaction request payload in a `sabre_payload` message (as described in section Ⅱ-1.1, step 2), bundles the Sabre payload into a batch and serializes it (see section Ⅱ-1.1, step 3), then posts it to the Bubba Bakery Gameroom REST API (see section Ⅱ-1-1, step 4).

   After the transaction is sent, the Bubba Gameroom UI displays a spinner until it is notified that the game has been updated in state (described in section Ⅱ-x.x).



5. The Bubba Bakery REST API forwards the XO 'take' transaction to the scabbard service (see the details in section Ⅱ-1.2).

6. The Bubba Bakery and Acme scabbard services use consensus (defined in Appendix B) to commit the move, as described in section Ⅱ-1.3. For Bob's move, Bubba Bakery's scabbard service starts the process.

7. After Bob's move has been committed to scabbard state, the scabbard services send the new state to the gameroom daemon via a WebSocket connection (as described in section Ⅱ-1.4).

   This message includes an updated game board that has Bob's O in the top right square (3rd space) of the game board and Bob's public key in the player_2 field.

```
  {
    “eventType”: “Set”,
    “message”: {
    “key”: “<xo game address>”,
    “value”: b“alice_vs_bob,--o-x----,P2-NEXT,<Alice’s public key>,<Bob’s public key>”
    }
}
```

8. When each Gameroom daemon receives the message, it updates the `alice_vs_bob` entry in the `xo_games` table in the database (this entry was created in section Ⅱ-1.5).

   At the end of the operation, the `xo_games` table looks like this:

| id | circuit_id | game_name | player_1 |
|---|---|---|---|
| <auto generated id> | gameroom::acme-node-000::bubba-node-000::<UUIDv4> | alice_vs_bob | <Alice’s public key> |

| player_2 | game_status | game_board | created_time | updated_time |
|---|---|---|---|---|
| <Bob's public key> | P1-NEXT | --o-x---- | <time entry was created> | <time entry was updated> |

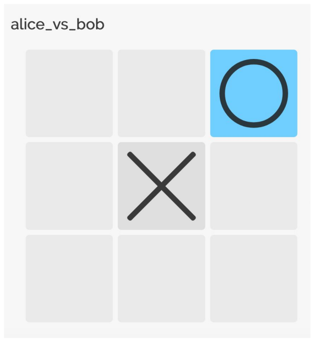
   As with Alice's first move, the gameroom daemon also adds a "game updated" notification to the `gameroom_notification` table (see section Ⅱ-2, step 5).

9. The Gameroom REST APIs notify the Gameroom daemons that the XO game’s state has been updated. This process is the same as for Alice's first move (see section Ⅱ-2, step 6), but the notification details contain information about Bob's move.

   When the Gameroom REST APIs return the list of games and game data, the `alice_vs_bob` game data shows that Bob is player_2, Alice has the next move, and each player has made one move on the game board.

```
{
   "data": [
        {
           "circuit_id": "gameroom::acme-node-000::bubba-node-000::<UUIDv4>",
           "game_name": "alice_vs_bob",
           "player_1": "<Alice’s public key>",
           "player_2": "<Bob’s public key>",
           "game_status": "P1-NEXT",
           "game_board": "--o-x----",
           "created_time": <time entry was created>,
           "updated_time": <time entry was updated>
       }
   ],
   "paging": {

     ... [SNIP] ...

   }
}
```

10. The Bubba Bakery UI now shows that Bob's move has been committed by replacing the spinner in the upper right corner with a blue O.



## Ⅱ-5. Behind Scene 5: Alice wins the game

After each move in the XO game, the XO smart contract checks the current game state to determine if the move resulted in a win or a tie. If not, the game state is updated to show which player moves next. (See Appendix D for the XO execution rules and game state values.)

1. Before Alice's last move, the `xo_games` table in the database looks like this:

| id | circuit_id | game_name | player_1 |
|---|---|---|---|
| `<auto-generated id>` | `gameroom::acme-node-000::bubba-node-000::<UUIDv4>` | `alice_vs_bob` | `<Alice's public key>` |

| player_2 | game_status | game_board | created_time | updated_time |
|---|---|---|---|---|
| `<Bob's public key>` | `P1-NEXT` | `o-o-xox-x` | `<time entry was created>` | `<time entry was updated>` |

   After Alice clicks the winning spot, the move is approved and committed to state, and the Gameroom daemons update the game state. Now, the `xo_games` table in the database looks like this:

| id | circuit_id | game_name | player_1 |
|---|---|---|---|
| `<auto-generated id>` | `gameroom::acme-node-000::bubba-node-000::<UUIDv4>` | `alice_vs_bob` | `<Alice's public key>` |

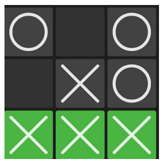
| player_2 | game_status | game_board | created_time | updated_time |
|---|---|---|---|---|
| `<Bob's public key>` | `P1-WIN` | `o-o-xoxxx` | `<time entry was created>` | `<time entry was updated>` |

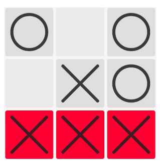
   The `P1-WIN` game status means that neither player can make any more moves.

2. Each Gameroom daemon receives the notification of the game state change, as described for Alice's first move (see section Ⅱ-2) and Bob's first turn (see section Ⅱ-3).

3. Each Gameroom daemon notifies the Gameroom UI to update the game board with Alice's winning move on the game board, as described in those earlier sections.

   The Acme UI shows a green row to indicate that Alice has won. The Bubba Bakery UI shows the row in red to let Bob know that he has lost.





# Act Ⅲ: Alice Creates Gamerooms with Yoda and Zixi

**Scene 1: The ketchup packet**

[The scene starts at the cliffhanger ending of ACT 2.]

NARRATOR (voice-over)

Bob is furious that he lost the XO game to Alice. He stands up -- jumps up, really -- and puts his foot on a KETCHUP PACKET on the floor. He stumbles into a MOBILE WHITEBOARD, grabbing the whiteboard as he falls and pulling it on top of himself. The whiteboard breaks his nose and his computer.

Bob screams in pain.

**Scene 2: Alice and Yoda set up a gameroom**

NARRATOR (voice-over)

Alice doesn't want to stop playing tic tac toe. She asks her friend Yoda, the VP of Yoyodyne Systems, to set up the Gameroom application at his company.

They start a competitive series of XO games in the "Alice vs Yoda" gameroom, playing from early morning until when Yoda leaves work.

IMAGE: Alice, hunched over her computer at sunset. CUT TO BLACK.

**Scene 3: Alice sets up a gameroom with Zixi**

NARRATOR (voice-over)

Alice decides that she needs more gamerooms. She arranges for Zixi, who works at Zymogen Industries, to install the Gameroom application and join the "Alice vs Zixi" gameroom. They play XO from dinner time until midnight.

Zixi has no idea that Alice has multiple gamerooms. She can only sees the "Alice vs Zixi" gameroom in her view of the Gameroom application.

FADE OUT: Alice, hunched over her computer in a dark office.

## **Scene 4: Alice's addiction**

FADE IN: Alice, hunched over her computer at dawn.

NARRATOR *(voice-over)*

Alice can't stop playing XO. She sets up gamerooms with people all over the globe. She lives in her office. She starts stealing other people's lunches from the office refrigerator. She drinks vending machine coffee at all hours. She doesn't sleep.

Finally, her co-workers stage an intervention. Alice goes to an addiction rehab center for a month.

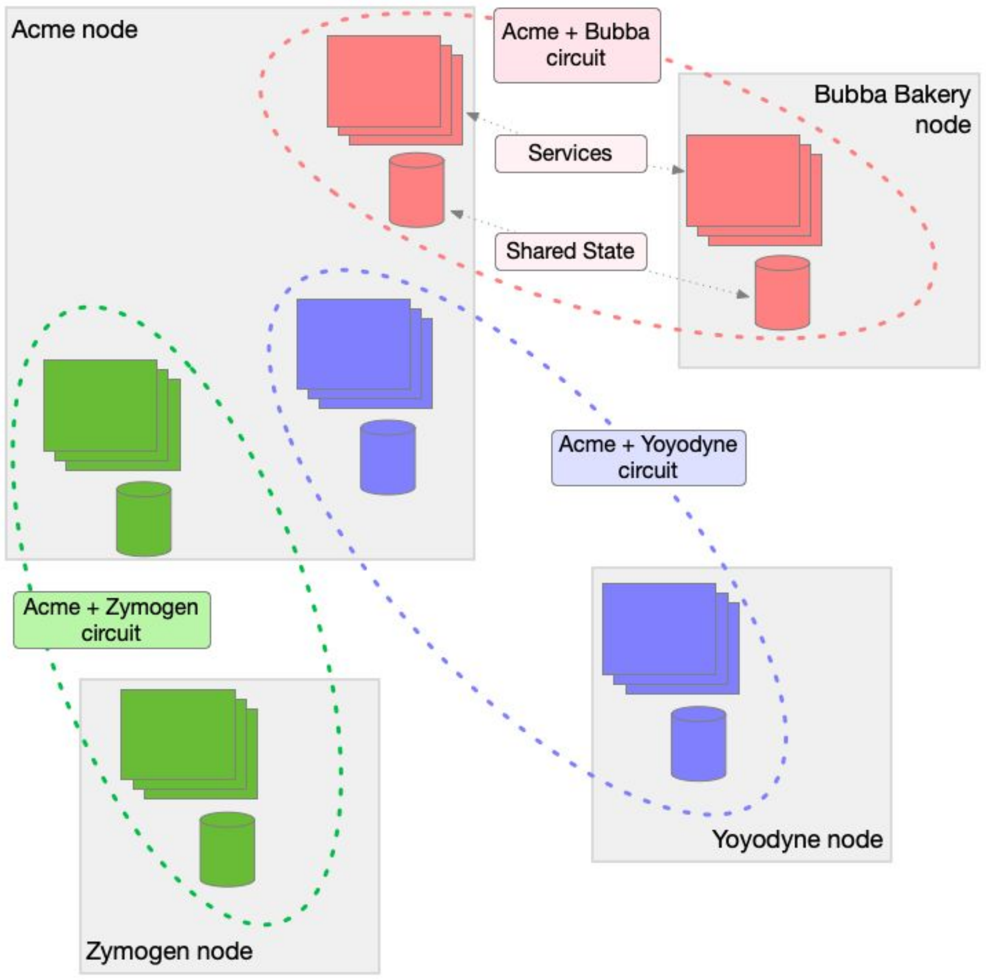
When Alice returns to work, she never plays online games. She also has an uncontrollable twitch when she sees an X.

CUT TO BLACK.

## Behind the Scenes: A Look at Act Ⅲ, Alice creates Gamerooms with Yoda and Zixi

At the end of Act Ⅲ, Alice has three gamerooms. Her gamerooms with Yoda and Zixi are created the same way as the first gameroom with Bob in Act Ⅰ. The game creation and XO gameplay transactions are the same as in Act Ⅱ.

This section summarizes how Splinter manages circuits, services, and shared state to keep each gameroom private and confidential.



A Splinter application, such as Gameroom, provides a set of distributed services that can communicate with each other across a Splinter circuit. In Gameroom, the Splinter software manages two-party private communication and network-wide multi-party shared state, all managed with consensus.

- A **circuit** is a virtual network within the broader Splinter network that defines a visibility domain and securely enforces privacy scope boundaries.
- **Services** provide applications with a REST API to dynamically create new circuits, based on business need. A service is an endpoint within a circuit that sends and receives private messages.
- **Connections** are dynamically constructed between nodes as circuits are created.

The existence of a circuit is confidential: Participants can see only the gamerooms that they have been invited to or have joined[1]. Alice sees her three gamerooms, but the other participants see only their one gameroom with Alice. If Yoda and Zixi set up a Yoyodyne + Zymogen gameroom, Alice wouldn't see it in her list of gamerooms.

Likewise, participant actions are private to the circuit. The transactions to create a gameroom, start a new game, or make an XO move are private to the participants in a gameroom. Shared state (a database updated by smart contracts) is visible only to the services within a circuit.

[1] The example Gameroom application handles gameroom access at the node level. For example, any user on the Acme node can view and join an Acme gameroom, including Alice's three gamerooms with Bob, Yoda, and Zixi. However, other applications could choose to restrict participation at the user level.

# The Prequel: Setting Up the Gameroom Application

Before Act 1 starts, sysadmins installed the Gameroom application on Alice and Bob's corporate networks, and both people are registered as Gameroom users.

This section describes the installation and user-registration process. It also describes how the Gameroom application registers the Gameroom daemon for admin service events.

## P.1. Running the Gameroom Demo with Docker

Gameroom is a demo Splinter application that allows you to set up dynamic two-party circuits (called "gamerooms") and play tic tac toe with shared state, as managed by two-phase commit consensus between the parties.

**Note**: This demo uses the Sabre smart contract engine provided in Sawtooth Sabre and the XO smart contract provided in the Hyperledger Sawtooth Rust SDK.

This example application includes a docker-compose file that sets up Splinter nodes for two imaginary organizations: Acme Corporation and Bubba Bakery. Both nodes are created on the same system so that this example is easy to run. For a proof-of-concept or production network, however, each node should be on a separate system.

**Prerequisites**: This demo requires Docker Engine and Docker Compose.

1. Clone the splinter repository.

2. To start Gameroom, run the following command from the Splinter root directory:

   ```
   $ docker-compose -f examples/gameroom/docker-compose.yaml up
   ```

3. Get Alice's and Bob's private keys to use in the web application. To display these keys, run `bash` using the `generate-key-registry` image, then read the private key.

   For example, to get Alice's private key, use these commands:

   ```
   $ docker-compose -f examples/gameroom/docker-compose.yaml \
   run generate-key-registry bash

   root@<container-id>:/# cat /key_registry/alice.priv; echo ""
   Alice's-private-key-value
   root@<container-id>:/#
   ```

4. In a browser, navigate to the Gameroom web application UI for each organization:
   - Acme UI: http://localhost:8080
   - Bubba Bakery UI: http://localhost:8081
5. When you are finished, shut down the demo.
   a. Enter CONTROL-C in the terminal window where you ran `docker-compose.yaml up`.

   ```
   ^C Gracefully stopping... (press Ctrl+C again to force)
   Stopping gameroomd-acme                      ... done
   Stopping gameroomd-bubba                     ... done
   Stopping gameroom-app-acme                   ... done
   Stopping splinterd-node-acme                 ... done
   Stopping splinterd-node-bubba                ... done
   Stopping db-acme                             ... done
   Stopping db-bubba                            ... done
   Stopping gameroom-app-bubba                  ... done
   $
   ```

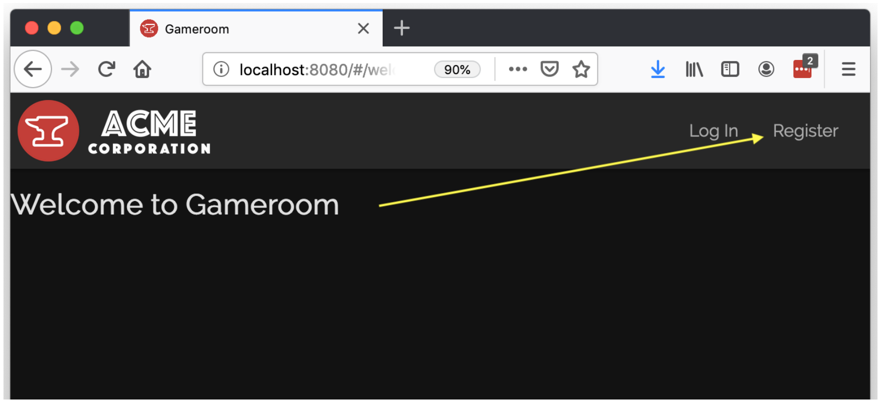
   b. Then shut down the docker containers with the following command:

   ```
   $ docker-compose -f examples/gameroom/docker-compose.yaml down
   ```

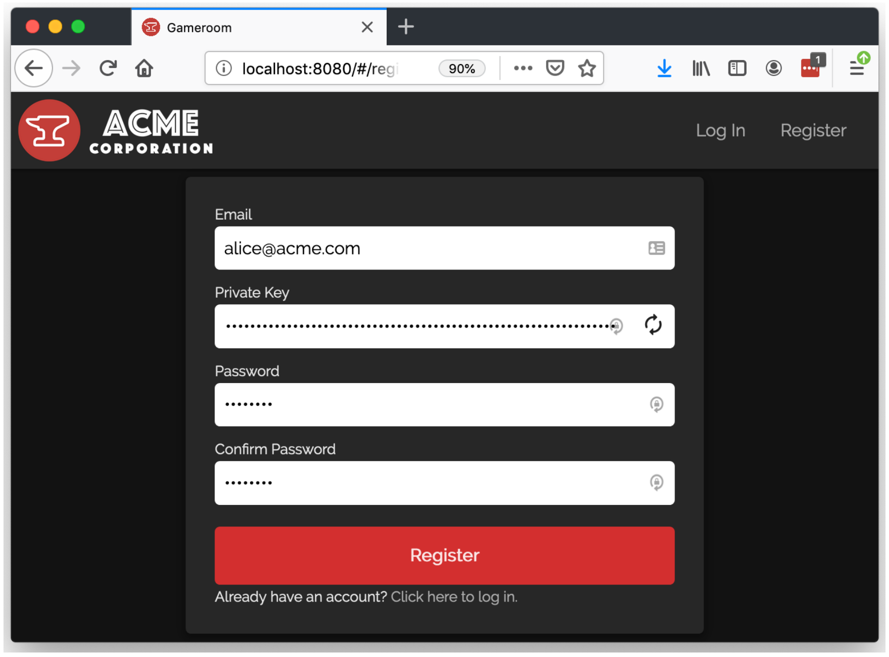
## P.2. Registering a User in the Gameroom UI

Each new user must register with the Gameroom application by specifying an email address, providing their private key, and setting a password to use when logging in.

When Alice navigates to the Gameroom application in her browser, the UI welcome page includes an option to register.



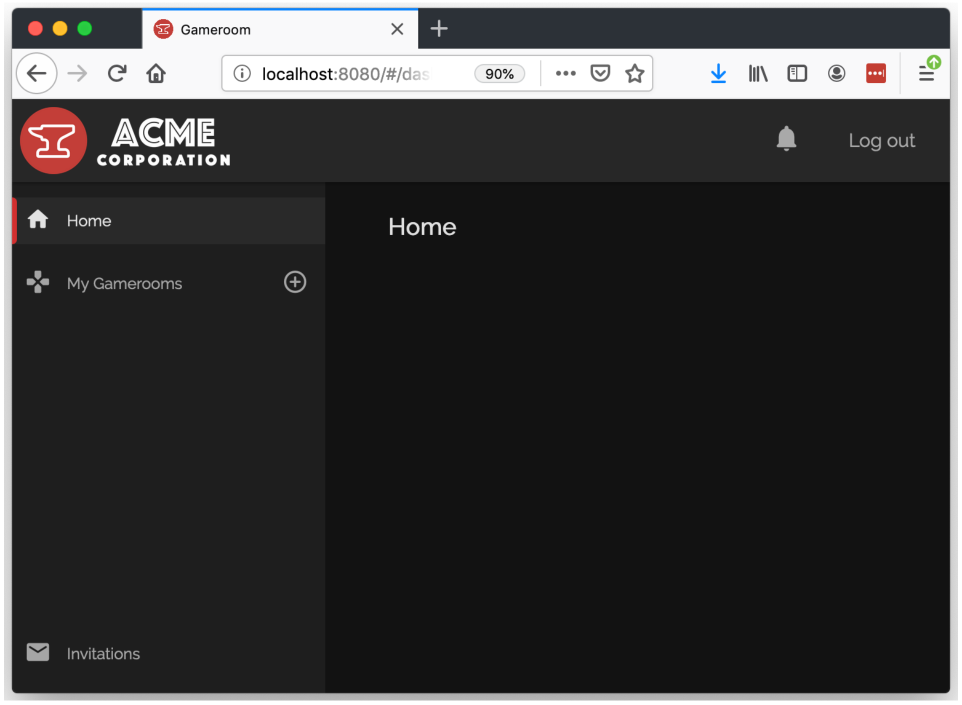
The Register page lets Alice enter her email address, private key, and password. The Gameroom demo docker-compose file generates private keys for Alice and Bob. See section P.1, step 3 to learn how to display these private keys.



Email
alice@acme.com
Private Key
••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
Password
••••••••
Confirm Password
••••••••

Already have an account? Click here to log in.

After Alice registers, she is automatically logged in. The Acme Gameroom UI displays the home page.



When a new user registers, the Gameroom daemon adds a new entry for that user to the `gameroom_user` table in the local Gameroom database. The `gameroom_user` table has the following schema:

```
CREATE TABLE IF NOT EXISTS gameroom_user (
  email                      TEXT        PRIMARY KEY, I
  public_key                 TEXT        NOT NULL,
  encrypted_private_key      TEXT        NOT NULL,
  hashed_password            TEXT        NOT NULL
);
```

For example, an entry for a new user looks like this:

| `email` | `hashed_password` | `public_key` | `encrypted_private_key` |
|---|---|---|---|
| `user@example.com` | `56ec82cb...480cad32` | `0384781f...5a7e4998` | `{\"iv\":...cgXrm\"}` |

This information is used for authorization when a user logs in, as described for Alice in section I -1.2 and for Bob in section I -3.2.

## P.3. Registering the Gameroom daemon for admin service events

The Gameroom application needs to receive notifications for admin service events (described in Appendix C) so that it can react appropriately to circuit proposal events and other admin events.

To see these events, the Gameroom daemon (`gameroomd`) must register an application authorization handler for circuits with a specific circuit management type. This handler manages the voting strategy for the application and notifies the application of any events received from the admin service on the local Splinter node.

As part of the event registration, the application authorization handler must specify the circuit management type. The `circuit_management_type` string in the circuit definition briefly describes the purpose of the circuit. For example, the Gameroom application uses the type `gameroom` for its circuits (see the `CircuitManagementPayload` definition in section I -2.3).

When an event occurs (such as a new circuit proposal or vote), each admin service uses a WebSocket connection to notify its application authorization handler about the event. In order to receive WebSocket notifications, each application authorization handler must send a registration request to its Splinter node's REST API.

For example, the Acme and Bubba Bakery Gameroom daemons would send this registration request:

```
GET /ws/admin/register/gameroom
```

See Appendix C for more information on circuit events.

# Appendix A: Peer Authorization

This appendix describes the peer authorization process that occurs as part of creating a circuit.

To be able to communicate on a Splinter network, each node and service involved in the proposed circuit must go through authorization. Each node must authorize with the other node (or nodes) involved in the circuit; each service must authorize with its own node. After the node or service is authorized, its peer ID (the node ID or service ID) is used to prove its identity.

Before a node or service is authorized, it can send only authorization messages (in a specific order). If it sends any other messages before the connection is authorized, those messages will be dropped.

## A.1 The Authorization Process

When the admin service on the first node (the node where the circuit request originated) requests connections with the other members' nodes and services (as described in section I -2.4), it starts the process of authorizing the nodes and services on those nodes.

1. First, the node or service requesting authorization is given a temporary peer ID with the following format:

   ```
   temp-<uuid>
   ```

2. Next, the node or service sends a `ConnectRequest` message wrapped in an `AuthorizationMessage`.

   The `ConnectRequest` specifies whether the authorization should be bidirectional (both sides) or unidirectional (one side only).

   - Connecting Splinter nodes should use bidirectional authorization, because each node must be authorized with the other node.
   - A Splinter service can use unidirectional authorization if it does not require the node to authorize itself with the service.

   The following example shows a bidirectional authorization request from a node.

   ```
   ---
   ConnectRequest:
         handshake_mode: BIDIRECTIONAL

   ---
   AuthorizationMessage:
         message_type: CONNECT_REQUEST
        payload: <bytes of connect request>
   ```

3. When a Splinter node receives a `ConnectRequest`, it responds with a ConnectResponse that includes a list of supported authorization types. Currently, the only supported authorization type is Trust, which means that the specified node or service (as identified by the peer ID) will be accepted as valid without any proof.

```
---
ConnectResponse:
       accepted_authorization_types: [Trust]
```

4. When the node or service requesting authorization receives the `ConnectResponse`, it checks the list of accepted authorization types for a matching, supported `authorization` type. If both sides support Trust authorization, this node or service will send a `TrustRequest` message that includes its peer ID (either a node ID or service ID).

```
---
TrustRequest:
       identity: <ID for the node or service>
```

5. When the node that is being connected to receives the `TrustRequest`, it changes the temporary peer ID to the actual peer ID (the node or service ID).

6. Next, this node sends an empty `AuthorizedMessage` to the connecting node or service to signify that it is now authorized to communicate on the Splinter network.

## A.2 Authorization Callbacks

When a new circuit is being created, the admin service may need to create a new connection to Splinter nodes that are not currently connected. This is done using the `PeerConnector`, as described in section  I -2.4. Before the admin service completes authorization, any `AdminDirectMessage` it sends will be dropped. This section describes how authorization callbacks are used to notify a node or service (such as the admin service) when the authorization process is complete.

The `AuthorizationInquisitor` interrogates the authorization status for a given peer ID, and includes a callback registration function to notify the caller of changes in peer authorization.

The `AuthorizationInquisitor` provides two methods:

- `is_authorized` checks whether a specific peer ID is registered
- `register_callback`, which takes a boxed `AuthorizationCallback`, requests notification when a peer's authorization status changes

```
pub trait AuthorizationInquisitor: Send {
   /// Register a callback to receive notifications about peer
      /// authorization statuses.
   fn register_callback(
               &self,
               callback: Box<dyn AuthorizationCallback>,
   ) -> Result<(), AuthorizationCallbackError>;

   /// Indicates whether or not a peer is authorized.
   fn is_authorized(&self, peer_id: &str) -> bool;
}
```

An `AuthorizationCallback` is a trait that must implement an `on_authorization_change` function that is called by the `AuthorizationInquisitor` when a peer's authorization status change. It takes the peer ID of the node or service whose authorization status has changed and the new `PeerAuthorizationState` (either `Authorized` or `Unauthorized`).

```
pub enum PeerAuthorizationState {
   Authorized,
   Unauthorized,
}

/// A callback for changes in a peer's authorization state.
pub trait AuthorizationCallback: Send {
   /// This function is called when a peer's state changes to Authorized
/// or Unauthorized.
   fn on_authorization_change(
               &self,
               peer_id: &str,
               state: PeerAuthorizationState,
   ) -> Result<(), AuthorizationCallbackError>;
}
```

The admin service is passed an `AuthorizationInquisitor` on startup. Then the admin service registers an `AuthorizationCallback` that will remove pending payloads from the `unpeered_payload` queue and move them the pending circuit payload queue once all required members have successfully peered and authorized.

# Appendix B: Consensus

Consensus is used to reach agreement between multiple parties.

Within Splinter, consensus refers to a library that contains consensus algorithm implementations (called "consensus engines") and a single interface for using those algorithms. Splinter services are typically the consumers of this interface.

In the Gameroom application, both the admin service and the scabbard service use a consensus algorithm called *two-phase commit*, which is a basic consensus algorithm that requires all participating parties to agree. If any party disagrees, the consensus proposal (the item being considered) is rejected. The Gameroom example uses two-phase commit for items such as circuit proposals, proposal validation, and transactions to add a smart contract.

## B.1. Consensus Interface

The consensus interface defines the relationship between a service and a consensus engine.

A `Proposal` is the entity that consensus agrees on; it contains a summary of the underlying data that a service would like to commit, as well as information that may be useful to consensus. The `Proposal` is defined as a protobuf:

```
message Proposal {
  // The proposal’s identifier, which is a hash of `previous_id`,
  // `proposal_height`, and `summary`
  bytes id = 1;
  // The identifier of the proposal’s immediate predecessor
  bytes previous_id = 2;
  // The number of proposals preceding this one (used for ordering
  // purposes)
  uint64 proposal_height = 3;
  // A summary of the data this proposal represents
  bytes summary = 4;
  // Opaque data that is provided by the consensus algorithm
  bytes consensus_data = 5;
}
```

A message sent between consensus engines is called a `ConsensusMessage`, and is defined by the following protobuf:

```
message ConsensusMessage {
  // An opaque message that is interpreted by the consensus algorithm
  bytes message = 1;
  // ID of the service that created this message
  bytes origin_id = 2;
}
```

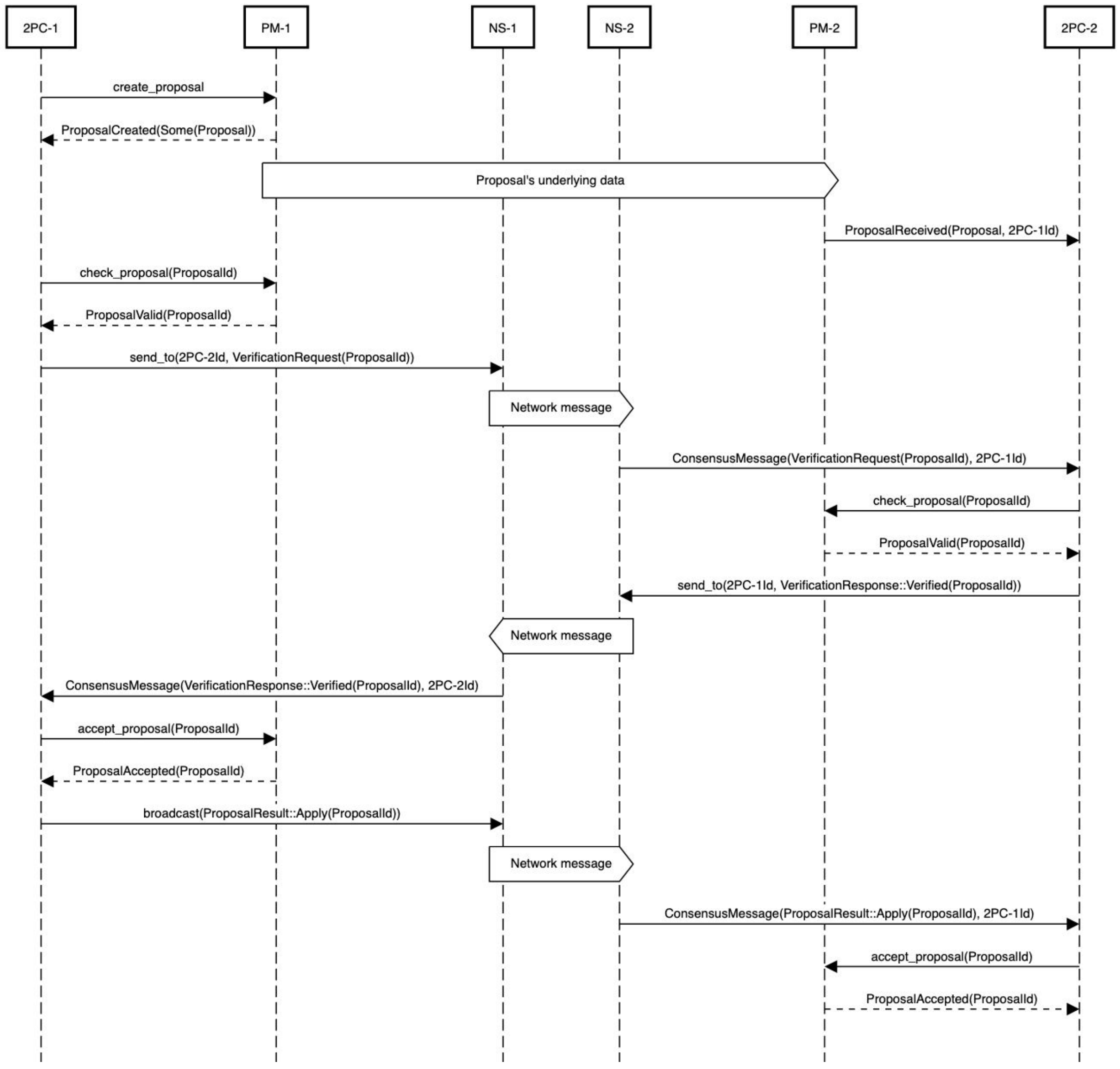
A service that uses consensus must implement two Rust traits for the consensus engine to interact with: the `ProposalManager` trait, which manages the `Proposals` that consensus decides on, and the `ConsensusNetwork` trait, which an engine uses to send messages to other nodes' consensus engines.

The consensus algorithm itself is implemented using the `ConsensusEngine` trait.

## B.2. Two-Phase Commit

Two-phase commit (2PC) is a basic consensus algorithm that requires agreement from all parties in order to accept a proposal.

The following diagram summarizes the operation of this algorithm. It shows the activities on two nodes for the consensus engines (2PC-1 and 2PC-2), the proposal managers (PM-1 and PM-2), and the consensus network senders (NS-1 and NS-2).


2PC-1
PM-1
NS-1
NS-2
PM-2
2PC-2
create_proposal
ProposalCreated(Some(Proposal))
Proposal's underlying data
ProposalReceived(Proposal, 2PC-1Id)
check_proposal(ProposalId)
ProposalValid(ProposalId)
send_to(2PC-2Id, VerificationRequest(ProposalId))
Network message
ConsensusMessage(VerificationRequest(ProposalId), 2PC-1Id)
check_proposal(ProposalId)
ProposalValid(ProposalId)
send_to(2PC-1Id, VerificationResponse::Verified(ProposalId))
Network message
ConsensusMessage(VerificationResponse::Verified(ProposalId), 2PC-2Id)
accept_proposal(ProposalId)
ProposalAccepted(ProposalId)
broadcast(ProposalResult::Apply(ProposalId))
Network message
ConsensusMessage(ProposalResult::Apply(ProposalId), 2PC-1Id)
accept_proposal(ProposalId)
ProposalAccepted(ProposalId)

### B.2.1. TwoPhaseMessage Types

Two-phase commit has three message types that are sent between its consensus engines: PROPOSAL_VERIFICATION_REQUEST, PROPOSAL_VERIFICATION_RESPONSE, and PROPOSAL_RESULT. The following TwoPhaseMessage protobuf defines these message types.

```
message TwoPhaseMessage {
   enum Type {
         UNSET_TYPE = 0;
         PROPOSAL_VERIFICATION_REQUEST = 1;
         PROPOSAL_VERIFICATION_RESPONSE = 2;
         PROPOSAL_RESULT = 3;
   }

   enum ProposalVerificationResponse {
         UNSET_VERIFICATION_RESPONSE = 0;
         VERIFIED = 1;
         FAILED = 2;
   }

   enum ProposalResult {
         UNSET_RESULT = 0;
         APPLY = 1;
         REJECT = 2;
   }

   Type message_type = 1;

   bytes proposal_id = 2;

   ProposalVerificationResponse proposal_verification_response = 3;
   ProposalResult proposal_result = 4;
}
```

To send a message to one of its peers, the two-phase commit engine constructs the TwoPhaseMessage protobuf, serializes it into bytes, and passes it to the ConsensusNetworkSender, which will then wrap it in a ConsensusMessage and relay it to one or more peers.

When a two-phase engine receives a consensus message, it extracts and deserializes the TwoPhaseMessage protobuf, then handles the message.

## B.2.2. Startup

When a service using two-phase commit starts up, it creates the consensus engine and runs it in a new thread.

## B.2.3. Proposal Creation

The two-phase commit consensus engine can create new proposals when it  is not already performing consensus on a proposal. To create a new proposal, the consensus engine requests a new proposal from the Splinter service using the `ProposalManager.create_proposal()` method.

- If the service has data for consensus to agree on, it will create a proposal for that data and send it to consensus as a `ProposalCreated(Some(Proposal))` update.
- If the service does *not* have data for consensus, it will send a `ProposalCreated(None)` update to consensus, and consensus will ask again after a brief timeout.

After sending the new proposal to the consensus engine, the service sends the data (the item to be decided on) to the other services in the circuit. The other services send the new proposal to their respective consensus engines as a `ProposalReceived(Proposal, PeerId)` update, where the `PeerId` is the ID of the consensus engine that created the proposal.

## B.2.4. Coordinator and Initial Verification

A `ProposalManager`  is a Rust trait that must be implemented for a Splinter service and is used by the consensus engine to create, check, accept, and reject proposals

When a two-phase commit engine determines that it is the coordinator for a new proposal, it first asks its service to verify the proposal using the `ProposalManager.check_proposal()` method.

If the proposal is valid, the proposal manager replies with a `ProposalValid(ProposalId)` update; if it is invalid, it will reply with a `ProposalInvalid(ProposalId)` update.

- In the case of an invalid proposal, the coordinator will simply reject the proposal by calling `ProposalManager.reject_proposal()` and instruct its peers to do the same by broadcasting a `ProposalResult::REJECT` message.
- In the case of a valid proposal, the coordinator will request verification from the other verifying peers.

### B.2.5. Verification

To request verification from the verifying peers, the coordinator broadcasts a `PROPOSAL_VERIFICATION_REQUEST` for the proposal using the service's `ConsensusNetworkSender.broadcast()` method.

When each verifying consensus engine receives the `PROPOSAL_VERIFICATION_REQUEST` from the coordinator, it verifies the proposal itself by calling its service's `ProposalManager.check_proposal()` method and waiting for a response.

- If the verifier receives a `ProposalValid` update from its proposal manager, it will send a `ProposalVerificationResponse::VERIFIED` message to the coordinator using its service's `ConsensusNetworkSender.send_to()` method.
- If the verifier receives a `ProposalInvalid` update, it will send a `ProposalVerificationResponse::FAILED` message to the coordinator.

### B.2.6. Proposal Result and Commit/Reject

If the coordinator receives a `ProposalVerificationResponse::FAILED` response, the consensus engine tells the `ProposalsManager.reject_proposal`, which will roll back any changes being stored in the service.

If the coordinator receives a `ProposalVerificationResponse::VERIFIED`, the consensus engine checks whether it has received a verification response from every peer. If the engine has received all verification requests, it accepts the proposal and calls `ProposalsManager.accept_proposal`, which will commit the pending changes in the Splinter service.

The coordinator then sends a message about the `ProposalResult` to its peers, with either an `APPLY` or `REJECT` result. This notifies the other peers they should also accept or reject the proposals.

# Appendix C: Circuit Proposal Events

During Gameroom setup (see The Prequel), each node's Gameroom application authorization handler is registered as an authorization handler for the Gameroom application. This handler receives messages (via a WebSocket connection) about circuit proposal events.

1. The application authorization handler, which is part of the Gameroom daemon, sends a request to the Splinter REST API to register as an authorization handler for the Gameroom application. The request is a WebSocket handshake request that looks like this:

   ```
   GET /ws/admin/register/gameroom
   Upgrade: websocket
   Connection: Upgrade
   Sec-Websocket-Version: 13
   Sec-Websocket-key:  13
   ```

2. If the request is successful, the server sends a response indicating that the protocol will change from HTTP to WebSocket. The response looks like this:

   ```
   HTTP/1.1 101 Switching Protocols
   Upgrade: websocket
   Connection: Upgrade
   Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
   ```

3. After the protocol has been upgraded, the Gameroom Application Authorization Handler receives messages (via the WebSocket connection) about circuit proposal events related to the Gameroom application. The event types include:

- `ProposalSubmitted`
- `ProposalRejected`
- `ProposalAccepted`
- `ProposalVote`
- `CircuitReady`

These event messages are serialized JSON.

## C.1. `ProposalSubmitted` event

The serialized JSON message for a `ProposalSubmitted` event looks like this:

```
{
  "eventType": "ProposalSubmitted",
  "message": {
    "proposal_type": "Create",
    "circuit_id": "my_circuit",
    "Circuit_hash":  "8e066d41911817a42ab098eda35a2a2b11e93c753bc5ecc3ffb3e99ed99ada0d",
    "circuit": {
    "circuit_id": "my_circuit",
    "roster": [
      {
        "service_id": "scabbard_123",
        "service_type": "scabbard",
        "allowed_nodes": [
          "acme_corp"
         ]
       }
    ],
    "members": [
      {
        "node_id": "Node-123",
        "endpoint": "127.0.0.1:8282”
      }
    ],
    "authorization_type": "Trust",
    "persistence": "Any",
    "routes": "Any",
    "circuit_management_type": "gameroom",
    "application_metadata": []
  },
  "votes": [],
  "requester": "<requester public key>"
  "requester_node_id": <node id of the node the requester is registered"
  }
}
```

## C.2. `ProposalRejected` event

The serialized JSON message for a `ProposalRejected` event looks like this:

```
{
  "eventType": "ProposalRejected",
  "message": {
    "proposal_type": "Create",
    "circuit_id": "my_circuit",
    "circuit_hash":  "8e066d41911817a42ab098eda35a2a2b11e93c753bc5ecc3ffb3e99ed99ada0d",
    "circuit": {
    "circuit_id": "my_circuit",
    "roster": [
      {
        "service_id": "scabbard_123",
        "service_type": "scabbard",
        "allowed_nodes": [
          "acme_corp"
         ]
       }
    ],
    "members": [
      {
        "node_id": "Node-123",
        "endpoint": "127.0.0.1:8282”
      }
    ],
    "authorization_type": "Trust",
    "persistence": "Any",
    "routes": "Any",
    "circuit_management_type": "gameroom",
    "application_metadata": []
  },
  "votes": [{
      “public_key”: “<publickeyofvoter>”,
      “vote”: “Rejected”
      "voter_node_id": “ <node id of the node the requester is registered>”
    }],
  "requester": "<requester public key>"
  "requester_node_id": <node id of the node the requester is registered>"

  }
}
```

## C.3. `ProposalAccepted` event

The serialized JSON message for a `ProposalAccepted` event looks like this:

```
{
  "eventType": "ProposalAccepted",
  "message": {
    "proposal_type": "Create",
    "circuit_id": "my_circuit",
    "circuit_hash":  "8e066d41911817a42ab098eda35a2a2b11e93c753bc5ecc3ffb3e99ed99ada0d",
    "circuit": {
    "circuit_id": "my_circuit",
    "roster": [
      {
        "service_id": "scabbard_123",
        "service_type": "scabbard",
        "allowed_nodes": [
          "acme_corp"
         ]
       }
    ],
    "members": [
      {
        "node_id": "Node-123",
        "endpoint": "127.0.0.1:8282”
      }
    ],
    "authorization_type": "Trust",
    "persistence": "Any",
    "routes": "Any",
    "circuit_management_type": "gameroom",
    "application_metadata": []
  },
  "votes": [{
      “public_key”: “<publickeyofvoter>”,
      “vote”: “Accepted”
      "voter_node_id": “ <node id of the node the requester is registered>”
    }],
  "requester": "<requester public key>"
  "requester_node_id": <node id of the node the requester is registered>"
  }
}
```

## C.4. `ProposalVote` event

The serialized JSON message for a `ProposalVote` event looks like this:

```
 {
  "eventType": "ProposalVote",
  "message": {
    "proposal_type": "Create",
    "circuit_id": "my_circuit",
    "circuit_hash":
"8e066d41911817a42ab098eda35a2a2b11e93c753bc5ecc3ffb3e99ed99ada0d",
    "circuit": {
    "circuit_id": "my_circuit",
    "roster": [
      {
        "service_id": "scabbard_123",
        "service_type": "scabbard",
        "allowed_nodes": [
          "acme_corp"
         ]
       }
    ],
    "members": [
      {
        "node_id": "Node-123",
        "endpoint": "127.0.0.1:8282”
      }
    ],
    "authorization_type": "Trust",
    "persistence": "Any",
    "routes": "Any",
    "circuit_management_type": "gameroom",
    "application_metadata": []
  },
  "votes": [{
      “public_key”: “<publickeyofvoter>”,
      “vote”: “Accepted”
      "voter_node_id": “ <node id of the node the requester is registered>”
    }],
  "requester": "<requester public key>"
  "requester_node_id": <node id of the node the requester is registered>"
  },
}
```

# Appendix D: XO Smart Contract Specification

The XO smart contract allows users to play the simple board game tic tac toe (also known as "Noughts and Crosses" or "X's and O's").

## D.1. XO State Entries

An XO state entry consists of the UTF-8 encoding of a string with exactly four commas, which has the following format:

```
<game-name>,<game-board>,<game-state>,<player1-key>,<player2-key>
```

- `<game-name>` is the name of the game, as a non-empty string that does not contain the character |.
- `<game-board>` represents the game board as a 9-character string (called "the board string") that contains only `O`, `X`, or `-`.
- `<game-state>` is one of the following: `P1-NEXT`, `P2-NEXT`, `P1-WIN`, `P2-WIN`, or `TIE`. (`P1` and `P2` stand for "player 1" and "player 2".)
- `<player1-key>` and `<player2-key>` are the (possibly empty) public keys associated with the game's players.

In the event of a hash collision (two or more state entries sharing the same address), the colliding state entries are stored as the UTF-8 encoding of the following string, with entries sorted alphabetically:

```
<a-entry>|<b-entry>|...
```

### D.1.1. State Addressing

XO data is stored in state using addresses generated from the XO "family name" (explained below) and the name of the game being stored.

In particular, an XO address consists of the first 6 characters of the SHA-512 hash of the UTF-8 encoding of the string "xo" (which is "`5b7349`"), plus the first 64 characters of the SHA-512 hash of the UTF-8 encoding of the game name.

For example, the XO address for a game called "mygame" could be generated as follows:

```
>>> hashlib.sha512('xo'.encode('utf-8')).hexdigest()[:6] +
hashlib.sha512('mygame'.encode('utf-8')).hexdigest()[:64]
'5b7349700e158b598043efd6d7610345a75a00b22ac14c9278db53f586179a92b72fbd'
```

## D.2. XO Transaction Payload

An XO transaction request payload consists of the UTF-8 encoding of a string with exactly two commas, which is formatted as follows:

```
<name>,<action>,<space>
```

- `<name>` is the game name, as a non-empty string not containing the character `|`. If `<action>` is *create*, the new name must be unique.
- `<action>` is the game action: *create*, *take*, or *delete*.
- `<space>` is the location on the board, as an integer between 1-9 (inclusive), if `<action>` is *take*.

## D.3. XO Transaction Header

Each XO transaction must include a header with the required inputs and outputs, plus the XO "family name" and version.

### D.3.1. Inputs and Outputs

The inputs and outputs for an XO transaction are just the state address generated from the transaction game name.

### D.3.2. Dependencies

XO transactions have no explicit dependencies.

### D.3.3. Family Name and Version

Each smart contract has a "family name", which identifies the smart contract type, and a version number. The term "family" comes from the XO transaction family (and transaction processor) in Hyperledger Sawtooth, which is an off-chain version of the XO business logic.

- family_name: “xo”
- family_version: “`1.0`”

## D.4. XO Execution

When a running XO smart contract receives a transaction request and a state dictionary, it checks the validity of the request. A valid transaction request payload has a game name, an action, and (if the action is *take*) a space.

Next, the XO smart contract checks whether the transaction (the requested action) is valid, then updates the state entry according to the specified action.

- If the action is *create*, the transaction is invalid if the game name is already in state dictionary. Otherwise, the smart contract will store a new state entry with board `---------` (a blank board), game state `P1-NEXT`, and empty strings for both player keys.

- If the action is *delete*, the transaction is invalid if the game name is not in the state dictionary. Otherwise, the smart contract will delete the state entry for the game.

- If the action is *take*, the transaction is invalid if the game name is not in the state dictionary. Otherwise, there is a state entry under the game name with a board, game state, player-1 key, and player-2 key.
    1. If the game name is in the state dictionary, the transaction is invalid if one of the following is true:
        - The game state is `P1-WIN`, `P2-WIN`, or `TIE`
        - The game state is `P1-NEXT`, the player-1 key is not null, and the player-1 key is different from the transaction signing key
        - The game state is `P2-NEXT`, the player-2 key is not null, and the player-2 key is different from the transaction signing key
        - The specified ("space-th") character in the board string has already been claimed (is not `-`).
    2. Otherwise, the smart contract will update the state entry as follows:
        a. Player keys: If the player-1 key is null (the empty string), it will be updated to the key with which the transaction was signed. If the player-1 key is not null and the player-2 key is null, the player-2 key will be updated to the signing key. Otherwise, the player keys will not be changed.
        b. Board: If the game state is `P1-NEXT`, the board will be updated with an `X` (player 1's character) in the specified space. That is, the updated board will be the same as the initial board, except with the "space-th" character replaced by the character `X`. If the game state is `P2-NEXT`, the same action occurs with an `O` (player 2's character).
        c. Game state: The smart contract updates the game state based on the contents of the board string. In this description, the first three characters of the board string represent the *first row*, the next three characters are the *second row*, and the last three characters are the *third row*.

           A character has a win on the board if any of the following is true:
            - If any row consists of the same character.
            - If the same character appears in a column (all the rows have the same first or second or third character).
            - If the same character appears in a diagonal line (the first character/first row, second character/second row, and third character/third row are the same; or the third character /first row, second character/second row, and first character/third row are the same).

3. Then the smart contract checks for a tie:
    - If `X` has a win on the board and `O` doesn't, the updated state will be `P1-WINS`.
    - If `O` has a win on the board and `X` doesn't, the updated state will be `P2-WINS`.
    - Otherwise, if the updated board has no empty spaces (does not contain `-`), the updated state will be `TIE`.
    - Otherwise, the game continues and the other player takes a turn. If the initial state was `P1-NEXT`, the updated state will be `P2-NEXT`. Conversely, if the initial state was `P2-NEXT`, the updated state will be `P1-NEXT`.

# Glossary

admin circuit

Splinter circuit that automatically includes the admin services of all connected nodes. This circuit is used to send administrative messages for operations such as circuit creation.

admin service

Splinter service that handles administration tasks. In the Gameroom application, the admin service is part of the Splinter daemon (`splinterd`) that runs on each node.

Each admin service has a service ID in the form `admin::{nodeID}`. For example, the service ID for Gameroom's Acme admin service is `admin::acme-node-000`.

alias

User-supplied name for a circuit. The Gameroom UI calls this a "gameroom name".

application authorization handler

Part of an application that handles notifications for pending circuit proposals and commit protocol updates. The application authorization handler also determines how voting is handled for the application, such as waiting for the client to submit a manual vote or accepting all received proposals.

The application authorization handler must register with the admin service (using the Splinter REST API) for a specific circuit management type, so that the admin service knows which circuit proposals are controlled by this handler.

circuit

Splinter connection between organizations (nodes) that provides private communication, as managed by services on each node. A client application might use a different term; for example, the Gameroom application calls this a "gameroom".

In addition, all nodes can connect to an admin circuit that handles administration functions.

circuit management type

String (stored in a circuit definition) that indicates which application authorization handler will handle this circuit's change proposals. An application authorization handler uses this string when registering as a handler with the node's admin service.

circuit proposal
: Circuit that has been requested but is not final. A circuit proposal, which is stored in the admin service, contains the pending circuit definition and the votes for or against the proposal. The pending circuit in the proposal cannot be used for communication until the circuit is approved and the accepted proposal is committed.

circuit roster
: Set of services that are authorized to communicate over the circuit.

client
: Short term for a client application for Splinter. A client application usually includes a user interface (UI) and a server-side daemon with application-specific handlers. For example, the Gameroom client has a web-based browser interface and a Gameroom daemon, `gameroomd`.

consensus
: Splinter component that is used by services to agree on shared state.

consensus proposal
: Encapsulation of data that services want to agree on (like a transaction), plus consensus-specific information such as ID and ordering information.

Gameroom
: Example multi-party Splinter application (also called a "distributed application") that creates circuits with specific members. Note that the capital G marks the application name; an individual circuit is called a *gameroom* (with a lower-case g).

gameroomd
: Gameroom daemon; part of the example Gameroom application that provides the Gameroom REST API and Gameroom application authorization handler.

invitation
: Gameroom application's term for a circuit proposal that contains a pending circuit.

member
: Splinter node that is a proposed or actual participant in a circuit.

peer nodes
: Splinter nodes that have an authorized (authenticated) connection to each other. Peering is a trusted connection between nodes.

peer services
: Splinter services that share an isolated portion of state on a circuit.

pending circuit
: Proposed circuit (defined in a circuit proposal) that is waiting for approval and is not yet ready for use. The Gameroom application uses the term "invitation" and marks proposed gamerooms with the status "Pending".

scabbard
: Splinter service that includes the Sawtooth Sabre transaction handler and Hyperledger Transact, using two-phase commit consensus to agree on state. This application-specific service is specifically configured to work with the example Gameroom application.

scabbard REST API
: Endpoints exposed by the Splinter REST API that allow interactions with a scabbard service (for operations such as adding batches).

service
: Portion of a daemon that handles administration or application-specific functions, such as the Splinter daemon's admin service or the Gameroom daemon's scabbard service. A service has a service ID that is specified in the circuit definition.

service orchestrator
: Splinter component that is used by the admin service to initialize new services when a circuit is created.

splinterd
: Splinter daemon that includes a Splinter REST API and an admin service.

state delta export
: Process of reading state-change updates from Splinter and uploading them to a local database. An application provides this functionality in a state delta processor (or state delta export process). For example, the Gameroom application registers for XO smart contract updates and uses the `XoStateDeltaProcessor` to process the information.

two-phase commit
: Basic consensus algorithm that requires all participating parties to agree. If any party disagrees, the consensus proposal (the item being considered) is rejected.