Implementation Guide: Linked Data Notifications on Solid Pods

Anshul Manapure

May 13, 2025

Contents

1 Introduction

This guide walks you through building an end-to-end Linked Data Notifications (LDN) system on Solid Pods. We cover environment setup, server and client implementations, and demonstrate how to deliver and consume JSON-LD notifications using HTTP, ACLs, and WebSockets.

2 Prerequisites

- **Node.js** v18 or higher (with built-in `fetch`)
- **npm** or **yarn**
- Two Solid Pods (e.g. on solidcommunity.net): one for the server, one for the client
- Client Credentials (ID + Secret) for both Pods
- Familiarity with Express.js, WebSockets, and Solid ACLs

3 Project Layout

```
project-root/
          client/
                auth.js
                client.js
                .env
          server/
              server.js
              .env
```

4 Data-Flow Diagram

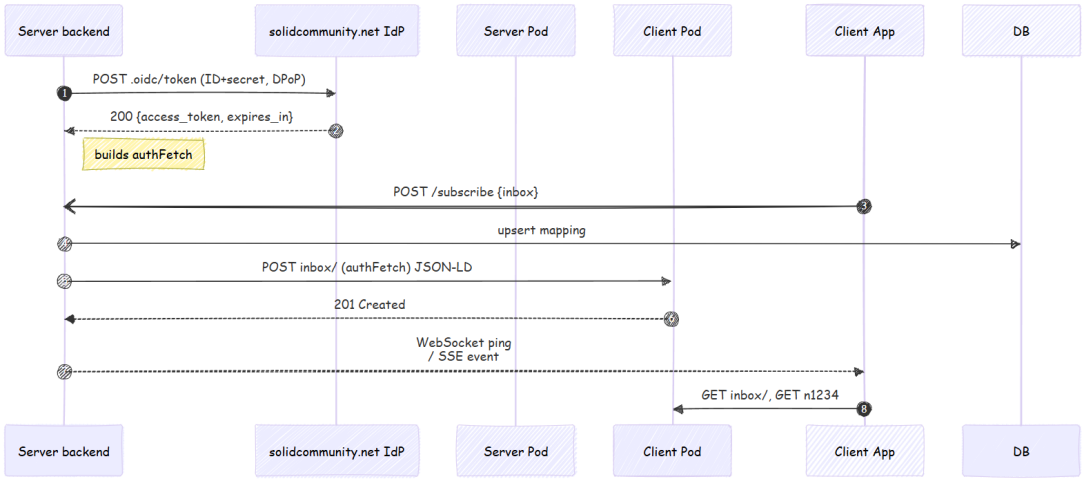

Figure 1: End-to-End LDN Sequence (steps 1–8)

5 Environment Configuration

5.1 Client `.env`

```
CLIENT_TOKEN_ID=<your-client-id>
CLIENT_TOKEN_SECRET=<your-client-secret>
CLIENT_INBOX_URL=https://yourname.solidcommunity.net/inbox/
SERVER_URL=http://localhost:4000
POLL_INTERVAL_MS=10000
```

5.2 Server `.env`

```
SERVER_PORT=4000
OIDC_ISSUER=https://solidcommunity.net
SERVER_POD_ROOT=https://yourserverpod.solidcommunity.net/files/
SERVER_TOKEN_ID=<your-server-id>
SERVER_TOKEN_SECRET=<your-server-secret>
```

6 Server Implementation

This section contains description of how each part of the server was built.

6.1 Authentication (Step 1)

Begin by logging the server into its own Solid Pod using the Server Credentials grant:

- We import `Session` from `@inrupt/solid-client-authn-node`, which manages OAuth2/DPoP for us.
- In `initSession()`, we create a new `Session()` and call `session.login({ OIDC`$Issuer, SERVER$
- If login fails, we exit immediately to avoid running unauthenticated operations.

6.2 Subscribe Endpoint (Steps 2–3)

Next, we expose an HTTP endpoint so clients can tell us where to post notifications:

- We use Express.js to create `POST /subscribe`. The request body must be JSON with an `inbox` URL.
- We simply store that URL in a local variable `registeredInbox` (we can write them to a database as well).
- We reply with HTTP 201 and include `session.info.webId` in the response body so the client knows which WebID to grant ACLs to.
- This ensures the server's WebID is recorded in the client's inbox ACL (Append permissions) before any notifications arrive.

6.3 File Monitoring & Overwrite (Step 4–5)

To detect updates to our payload (`weights.bin`) and publish it in the Server Pod, we follow these steps:

- Install and import `chokidar`, a filesystem-watcher library.
- Invoke:

```
// Watch only new changes to weights.bin
chokidar.watch('weights.bin', { ignoreInitial: true })
  .on('change', async () => {
    // Read the updated file
    const data = fs.readFileSync('weights.bin');
    // Overwrite the Pod resource
    await overwriteFile(
      `${process.env.SERVER_POD_ROOT}weights.bin`,
      data,
      {
        contentType: 'application/octet-stream',
        fetch: session.fetch.bind(session)
      }
    );
    // Trigger notification delivery
    notifyClient();
  });
```

- This approach ensures:
  1. The local `weights.bin` is monitored for modifications only (no initial copy).
  2. Each change reads the exact current bytes and uses `overwriteFile()` to PUT them into the Pod under `SERVER_POD_ROOT`.
  3. Immediately after the file is updated on the Pod, we call `notifyClient()` to send LDN notifications and real-time pings.

6.4 Notification Delivery & Real-Time Push (Steps 4–6)

Once the file is in place, we notify subscribed clients:

- We build a JSON-LD notification object conforming to ActivityStreams: it has `@context`, `type:"Update"`, `actor` set to our server WebID, `object` pointing to the Pod URL, and a timestamp.
- We use session.fetch(registeredInbox, method: 'POST', headers, body:JSON.stringify(notification)) to POST into the client's inbox container.
- On HTTP 201 success, we log that the notification was accepted.
- To support instant client updates, we also run a WebSocket server (using the `ws` package). After each successful notification POST, we call `broadcastPing()`, which sends a simple `"ping"` message to all connected clients.

- Clients listening on that WebSocket path immediately re-list their inbox to retrieve the new notification resource.

6.5 Putting It All Together

1. *Startup:* Call `initSession()` to authenticate and bind `fetch`.
2. *Express Server:* Listen on port `SERVER_PORT`, expose `/subscribe` and `/simulate-update`.
3. *Watcher:* Invoke `chokidar.watch()` on the local `weights.bin`.
4. *Notification Flow:* On file change:
   (a) Overwrite the Pod file via `overwriteFile()`.
   (b) POST the JSON-LD notification to the client's inbox.
   (c) Broadcast a WebSocket "ping" for real-time alerting.

This design ensures the server never loses control of its own Pod, uses standard LDP operations, and delivers notifications both persistently (in the inbox container) and in real-time (via WebSocket).

7 Client Implementation

In this section we describe how the client app is built, step-by-step, to subscribe to notifications and consume them in real time.

7.1 6.1 Authentication

First, the client must authenticate to its own Solid Pod using the Client Credentials grant:

- Import `Session` from `@inrupt/solid-client-authn-node`.
- In `loginWithClientCreds()`, construct a new `Session()` and call:

```
// client/auth.js
const session = new Session();
await session.login({
  oidcIssuer:   process.env.OIDC_ISSUER,
  clientId:     process.env.CLIENT_TOKEN_ID,
  clientSecret: process.env.CLIENT_TOKEN_SECRET
});
```

- On success, `session.info.isLoggedIn` is `true`, and `session.fetch` attaches DPoP tokens to every HTTP request.
- Return the logged-in `session` so that `client.fetch = session.fetch.bind(session);` can be set.

7.2 6.2 Subscribe & ACL Grant (Steps 2–3)

Once authenticated, the client registers its inbox with the server and grants the server write access:

1. **Subscribe API Call**

```
// client/client.js
const res = await session.fetch(`${SERVER_URL}/subscribe`, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ inbox: CLIENT_INBOX_URL })
});
const serverWebId = await res.text();
```

2. **Fetch Container + ACL** Use `getSolidDatasetWithAcl()` to retrieve both data and ACL:

```
import { getSolidDatasetWithAcl } from '@inrupt/solid-client
  ';
const dsWithAcl = await getSolidDatasetWithAcl(
  CLIENT_INBOX_URL,
  { fetch: session.fetch }
);
```

3. **Obtain or Create ACL**
   - `getResourceAcl(dsWithAcl)` returns the container's current ACL.
   - If none exists, call `createAcl(dsWithAcl)` to get a writable ACL object.

4. **Set Permissions** Grant `Append` (and optionally `Read`) on both resource and default scopes:

```
import {
  setAgentResourceAccess,
  setAgentDefaultAccess,
  saveAclFor
} from '@inrupt/solid-client';

let acl = /* resource ACL from above */;
acl = setAgentResourceAccess(acl, serverWebId, {
  read:   true,
  append: true,
  write:  false,
  control:false
});
acl = setAgentDefaultAccess(acl, serverWebId, {
  read:   true,
  append: true,
  write:  false,
  control:false
});
await saveAclFor(dsWithAcl, acl, { fetch: session.fetch });
```

5. This ensures the server's WebID can `POST` new notifications without altering the client's own rights.

7.3 6.3 Polling & Listing (Step 7)

To discover new notifications, the client issues a `GET` on the inbox container:

- Use `getSolidDataset()` to fetch the container:

```
import { getSolidDataset } from '@inrupt/solid-client';
const ds = await getSolidDataset(CLIENT_INBOX_URL, {
  fetch: session.fetch
});
```

- Extract contained resource URLs with `getContainedResourceUrlAll()`:

```
import { getContainedResourceUrlAll } from '@inrupt/solid-
   client';
const urls = getContainedResourceUrlAll(ds).sort();
urls.forEach(url => console.log(url));
```

- Wrap this in a timer (`setInterval`) for periodic polling, if WebSockets are unavailable.

7.4 6.4 WebSocket Listener (Steps 6→7)

For near-instant updates, the client also listens for a "ping" from the server:

- Create a WebSocket connection to `ws://<server_host>:9000/ws`:

```
// client/client.js
import WebSocket from 'ws';
const ws = new WebSocket(`${SERVER_URL.replace(/^http/, 'ws')
   }/ws`);
```

- On receiving the message `"ping"`, trigger a re-list:

```
ws.on('message', message => {
  if (message === 'ping') {
    console.log('       Ping received     refresh inbox');
    listInbox();  // call the polling function above
  }
});
```

- Handle `open`, `close`, and `error` events to display connection status to the user.

Summary By combining OAuth2/DPoP login, ACL grants, LDP container polling, and WebSocket pushes, the client reliably receives and displays linked-data notifications as soon as they arrive.

8 Testing & Validation

1. Start the server: `node server/server.js`
2. Start the client: `node client/client.js` → choose *Subscribe*
3. Simulate update:

   ```
   curl -X POST http://localhost:4000/simulate-update
   ```

4. Observe:
   - Server logs file overwrite, notification sent, WS ping
   - Client logs WS ping and new inbox URL