Technical Documentation-SSO System with Kong API Gateway

1. Deconstructing kong.yml

The `kong.yml` file is the declarative heart of the gateway. It defines every component, route, and policy.

1.1 Core Concepts

- **Services**: Represent your upstream applications. Kong needs to know where to send traffic for a given request. We have services for the frontend clients (frontend-service, second-app-service) and the backend authentication logic (auth-service).
- **Routes**: Define how requests are mapped to Services. A route matches incoming request criteria (like the path, e.g., `/profile`) and forwards it to its configured Service.
- **Plugins**: These are the workhorses of Kong. They are modules that execute during the request/response lifecycle to implement policies like authentication, rate-limiting, logging, and transformations. In this project, we heavily use the `pre-function` plugin for custom Lua logic and the `jwt` plugin.
- **Consumers**: Represent users of a Service. In our case, we use a single `shared-consumer` to associate the JWT secret with the gateway. The `key` (`shared-key`) must match the `iss` (issuer) claim inside the JWTs generated by the saml-auth-service.

1.2 Key Routes and Plugins

The logic is orchestrated by applying specific plugins to specific routes.

1. **Protected Routes (e.g., `/profile`, `/dashboard`, `/app2/settings`)**
   - **Routes**: `profile-route, dashboard-route, second-app-dashboard, etc.`
   - **Tag**: `protected-route`. This is a logical tag for organization.
   - **Plugin**: `jwt-cookie-auth-redirect` (`handler.lua`).
   - **Function**: This custom plugin is the primary gatekeeper.
     - It runs on every request to a protected route.
     - It checks for a valid `access_token` cookie.
     - **If valid**: It adds the `X-User-Email` header to the request and allows it to proceed to the upstream service (e.g., `frontend-service`).
     - **If invalid/missing**: It saves the current URL into a `return_after_auth` cookie and redirects the user to `/auth` to begin the login process.

2. **Authentication Start Route (`/auth`)**
    - **Route**: `auth-route` .
    - **Plugin**: `pre-function` (`login-referer-handler`).
    - **Function**: This plugin acts as a helper. If the `jwt-cookie-auth-redirect` plugin didn't set the `return_after_auth` cookie, this plugin uses the `Referer` header as a fallback to ensure the user is returned to the correct place after login. The request is then forwarded to the `auth-service` (Java app) to initiate the SAML handshake.
3. **Token Handling Route (`/`)**
    - **Route**: `root-route` .
    - **Plugin**: `pre-function` (`token-handler`).
    - **Function**: This is a critical step *after* a successful SAML login. The saml-auth-service redirects the user back to Kong with the JWT in a URL parameter (e.g., `/?token=...`). This plugin on the root path intercepts this request. It extracts the token from the URL, sets it as a secure, `HttpOnly` `access_token` cookie, and then redirects the user to the original URL stored in the `return_after_auth` cookie.
4. **Logout Route (`/custom-logout`)**
    - **Route**: `logout-route` .
    - **Function**: This route is called by the client applications to log out. It forwards the request to the `auth-service` to invalidate the server-side SAML session. The `auth-service` then redirects to `/kong-clear-cookies` .
5. **Cookie Clearing Route (`/kong-clear-cookies`)**
    - **Route**: `kong-clear-cookies-route` .
    - **Function**: This is an internal route handled entirely by Kong. It clears the `access_token` and `JSESSIONID` cookies from the browser and then redirects the user to their final destination (e.g., the application's home page).

---

2. End-to-End User Flows

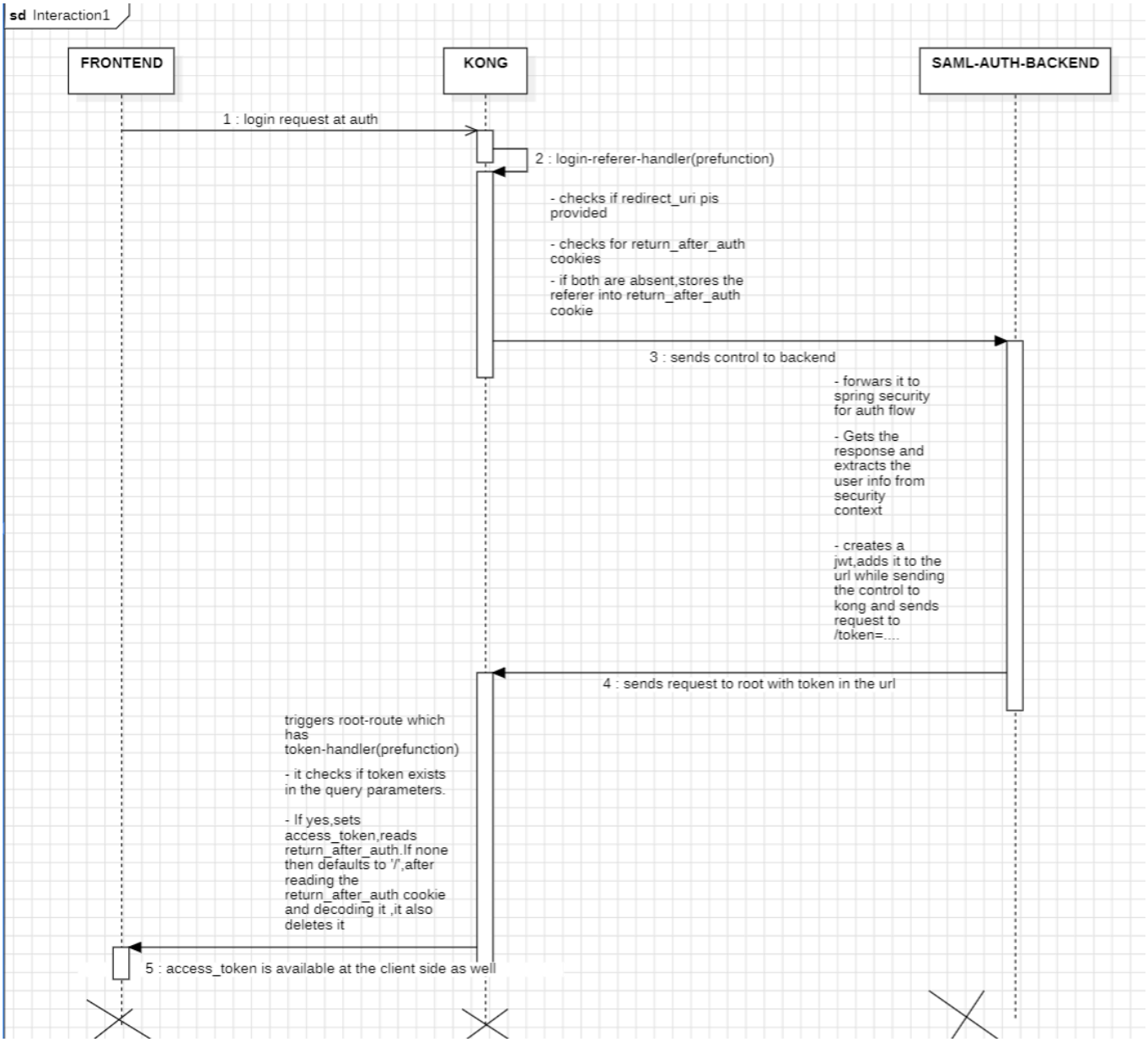
Flow 1: Login from an Unprotected Route (Homepage Button)

Goal: A new user on the homepage (`/`) clicks the "Login" button.

1. **User Action**: User clicks the "Login with Google" link on the `Home.js` component. This link points to `/auth` .
2. **Browser -> Kong**: Sends `GET /auth` .
3. **Kong (Routing & Plugin)**:
    - Kong matches the path to the `auth-route` .

- The `login-referer-handler` plugin (`kong.yml`) executes. It captures the `Referer` header (the homepage URL), Base64-encodes it, and stores it in a `return_after_auth` cookie. This ensures the user returns to their starting point.

4. **Kong -> saml-auth-service**: Kong proxies the request to the Java `auth-service` .
5. **saml-auth-service (SAML Initiation)**: The `/auth` endpoint in `EnterpriseAuthController.java` is triggered. It redirects the browser to `/saml2/authenticate/google` , initiating the SAML flow via Spring Security.
6. **SAML Handshake**: The user is redirected to the IdP (Google), authenticates, and is redirected back to the saml-auth-service's ACS URL (`/login/saml2/sso/google`) with a valid SAML Assertion.
7. **saml-auth-service (JWT Generation)**:
   - Spring Security validates the SAML assertion. On success, it forwards the request to the configured success URL: `/token` (`SamlSecurityConfig.java`).
   - The `getTokenAndRedirect` method in `AuthController.java` generates a signed JWT and redirects the browser back to Kong's root, embedding the token in a query parameter: `http://localhost:8000/?token=<JWT_HERE>` .
8. **Browser -> Kong (Token Handling)**: The browser follows the redirect.
9. **Kong (Cookie Setting)**:
   - Kong matches the request to the `root-route` .
   - The token-handler plugin (`kong.yml`) executes. It extracts the token from the URL, sets it as a secure, `HttpOnly` `access_token` cookie, and clears the `return_after_auth` cookie.
   - It then issues a final redirect to the URL that was stored in the `return_after_auth` cookie (the homepage).
10. **Final State**: The user is back on the homepage, now authenticated. The React app's Header component will call the `/api/userinfo` endpoint, detect the user session, and update the UI to show a "Logout" button.

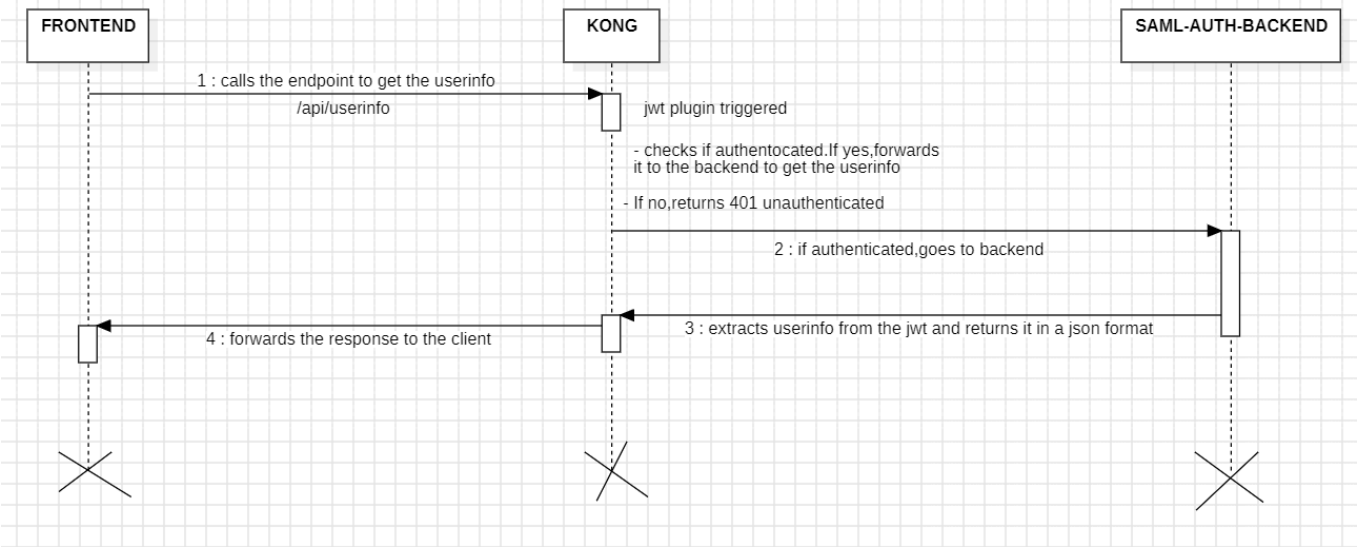


Flow 2: Fetching User Info

Goal: An authenticated user navigates to the `/profile` page.

1. **User Action**: User clicks the "View Profile" link in the UI.
2. **Browser -> Kong**: React Router handles the navigation client-side.
   The `Profile.js` component mounts and its useEffect hook triggers a call to ApiService.getUserInfo(). This sends a GET /api/userinfo request to Kong.
3. **Kong (Routing & Plugin)**:
   - Kong matches the path to the `api-route`.
   - The standard `jwt` plugin (`kong.yml`) attached to this route runs. It automatically finds the `access_token` cookie, validates its signature and expiration against the configured consumer secret.
   - Validation passes. Kong allows the request to proceed.

4. **Kong -> saml-auth-service**: Kong proxies the request to the `backend-api` service, which is the same Java application.
5. **saml-auth-service (User Info)**:
   - The `/api/userinfo` endpoint in `TokenApiController.java` is hit.
   - The controller code re-validates the JWT from the cookie and extracts the claims (email, name, etc.).
   - It returns a `200 OK` response with a JSON body containing the user's information.
6. **saml-auth-service -> Kong -> Browser**: The JSON response travels back through Kong to the browser.
7. **Final State**: The ApiService.js call resolves successfully. The Profile.js component receives the user data, updates its state, and renders the user's profile information on the screen.



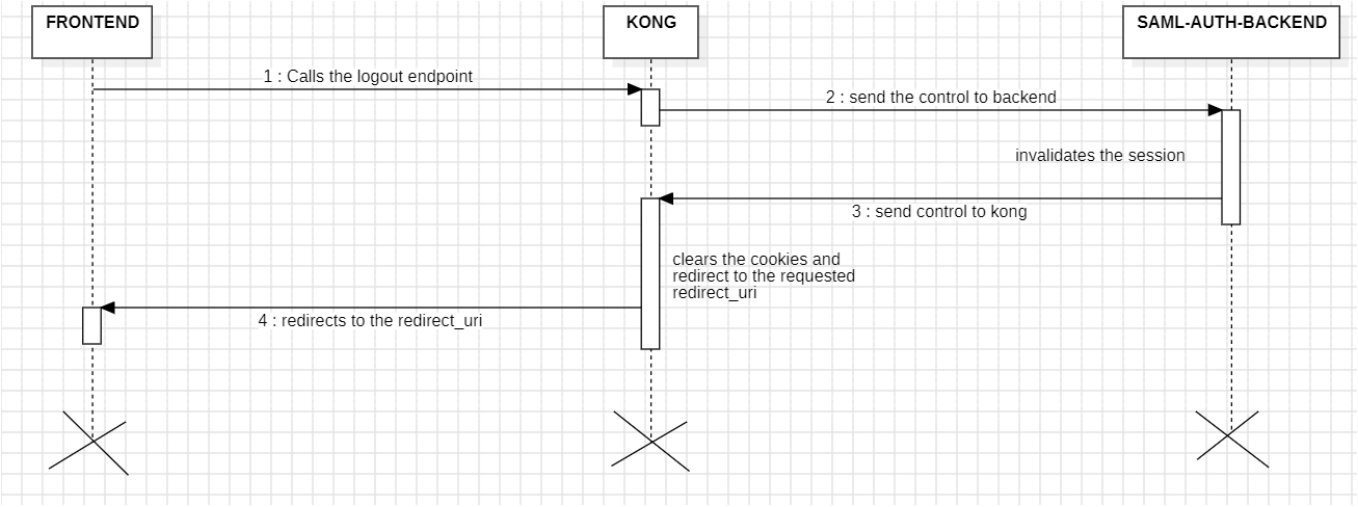
Flow 3: Logout

Goal: A logged-in user clicks the "Logout" button.

1. **User Action**: User clicks the logout link, for example in `Profile.js`. The link points to `/custom-logout?redirect_to=<homepage_url>`.
2. **Browser -> Kong**: Sends GET /custom-logout?redirect_to=....
3. **Kong -> saml-auth-service**: Kong matches this to the `logout-route` and proxies the request to the `auth-service`.
4. **saml-auth-service (Session Invalidation)**:
   - The `/custom-logout` method in `AuthController.java` is executed.
   - It invalidates the server-side HttpSession (which holds the SAML session details).
   - It then constructs a new URL pointing to an *internal* Kong endpoint: `http://localhost:8000/kong-clear-cookies?redirect_to=<original_redirect_param>`.

- It returns a `302 Redirect` to this new URL.

5. **Browser -> Kong (Cookie Clearing)**: The browser follows the redirect.
6. **Kong (Plugin Execution)**:
   - Kong matches the path to the `kong-clear-cookies-route`. This route is not associated with any service; it is handled entirely by its attached plugin.
   - The pre-function plugin executes. It generates `Set-Cookie` headers to immediately expire the `access_token` and `JSESSIONID` cookies in the browser.
   - It then issues a final `302 Redirect` to the URL specified in the `redirect_to` parameter.
7. **Final State**: The user is redirected to the homepage. All authentication cookies have been cleared from their browser, and the server-side session is gone. They are fully logged out.



Flow 4: Accessing a Protected Route

There are two primary scenarios:

1. The user is not authenticated (no valid JWT cookie).
2. The user is already authenticated (has a valid JWT cookie).

Scenario 1: Unauthenticated User (The Full Login Flow)

Goal: A user who is not logged in attempts to access `http://localhost:8000/profile`.

1. **Initial Request**
   - The user's browser sends a `GET` request for `/profile` to the Kong gateway.

2. **Kong Route Matching**
    - Kong receives the request and matches the path `/profile` to the `profile-route` defined in kong.yml.
    - This route is tagged with `protected-route`, which triggers the associated authentication plugin.
3. **Custom Plugin Execution (`jwt-cookie-auth-redirect`)**
    - The `access` function in the plugin's handler.lua file is executed.
    - The `get_cookie("access_token")` function is called. It inspects the request's `Cookie` header but finds no `access_token` cookie.
    - Since the cookie is `nil`, the `if cookie then...` block is skipped. The user is considered unauthenticated.
4. **Preparing for Redirect**
    - The plugin's code now prepares to redirect the user for login.
    - It captures the full URL the user was originally trying to access: `http://localhost:8000/profile`.
    - It Base64-encodes this URL to ensure it can be safely stored in a cookie.
    - It generates a `Set-Cookie` header to create a new cookie named `return_after_auth` containing the encoded URL. This cookie is temporary (`Max-Age=300`) and `HttpOnly` for security.
5. **The Redirect for Login**
    - The plugin executes `kong.response.exit(302, "", { ["Location"] = "/auth" })`.
    - This immediately halts the original request for `/profile` and sends an **HTTP 302 Redirect** response back to the browser. The browser is instructed to navigate to the `/auth` endpoint.
6. **Starting the SAML Handshake**
    - The browser follows the redirect and sends a new `GET` request to `/auth`.
    - Kong matches this to the `auth-route`, which proxies the request to the backend saml-auth-service.
    - The Java service receives the request and initiates the SAML 2.0 protocol, redirecting the user's browser to the external Identity Provider (e.g., Google) for authentication.
7. **User Authentication & JWT Generation**
    - The user authenticates successfully with the Identity Provider.
    - The IdP redirects the user back to the saml-auth-service with a valid SAML Assertion.
    - The Java service validates this assertion, extracts the user's details (email, name), and generates a signed JSON Web Token (JWT).
8. **Returning the Token to Kong**
    - The saml-auth-service redirects the user's browser back to Kong's root path (`/`), embedding the newly generated JWT in a URL query parameter:

`http://localhost:8000/?token=<THE_JWT>`.

9. **Kong Sets the Authentication Cookie**
    - The browser follows this redirect. Kong receives the request for `/`.
    - This request matches the `root-route` in kong.yml. The `pre-function` plugin named `token-handler` attached to this route is executed.
    - The plugin extracts the JWT from the `token` query parameter.
    - It generates a `Set-Cookie` header to create the final `access_token` cookie, marking it as `HttpOnly`.
    - It reads the `return_after_auth` cookie to find the original URL (`http://localhost:8000/profile`), then clears that temporary cookie.
    - It issues a final **HTTP 302 Redirect** back to the original URL.
10. **Final Access to Protected Route**
    - The browser follows the final redirect and sends a `GET` request for `/profile` again. This time, the request includes the valid `access_token` cookie.
    - The flow now proceeds as described in Scenario 2.

Scenario 2: Authenticated User

Goal: A user who is already logged in (has a valid `access_token` cookie) accesses `http://localhost:8000/profile`.

1. **Initial Request**
    - The user's browser sends a `GET` request for `/profile` to Kong. The request includes the `access_token` cookie.
2. **Kong Route Matching**
    - Kong matches the path `/profile` to the `profile-route`.
3. **Custom Plugin Execution (`jwt-cookie-auth-redirect`)**
    - The `access` function in handler.lua is executed.
    - `get_cookie("access_token")` successfully finds and returns the JWT string from the cookie.
    - The `if cookie then...` block is entered.
    - The JWT is decoded and validated. The plugin checks that the token's signature is valid, its expiration time (`exp`) is in the future, and its issuer (`iss`) is `shared-key`.
    - The validation succeeds.
4. **Augmenting the Request**
    - The plugin extracts the user's email from the JWT payload.

- It adds a new header to the request being sent to the upstream service: `X-User-Email: <user's email>`.

5. **Granting Access**
   - The plugin's `access` function finishes by executing `return`. This signals to Kong that the request is authorized.
   - The redirect logic at the end of the file is never reached.
6. **Proxy to Upstream**
   - Kong forwards the (now augmented) request to the configured upstream `frontend-service`. The React application receives the request and can now render the protected profile page.