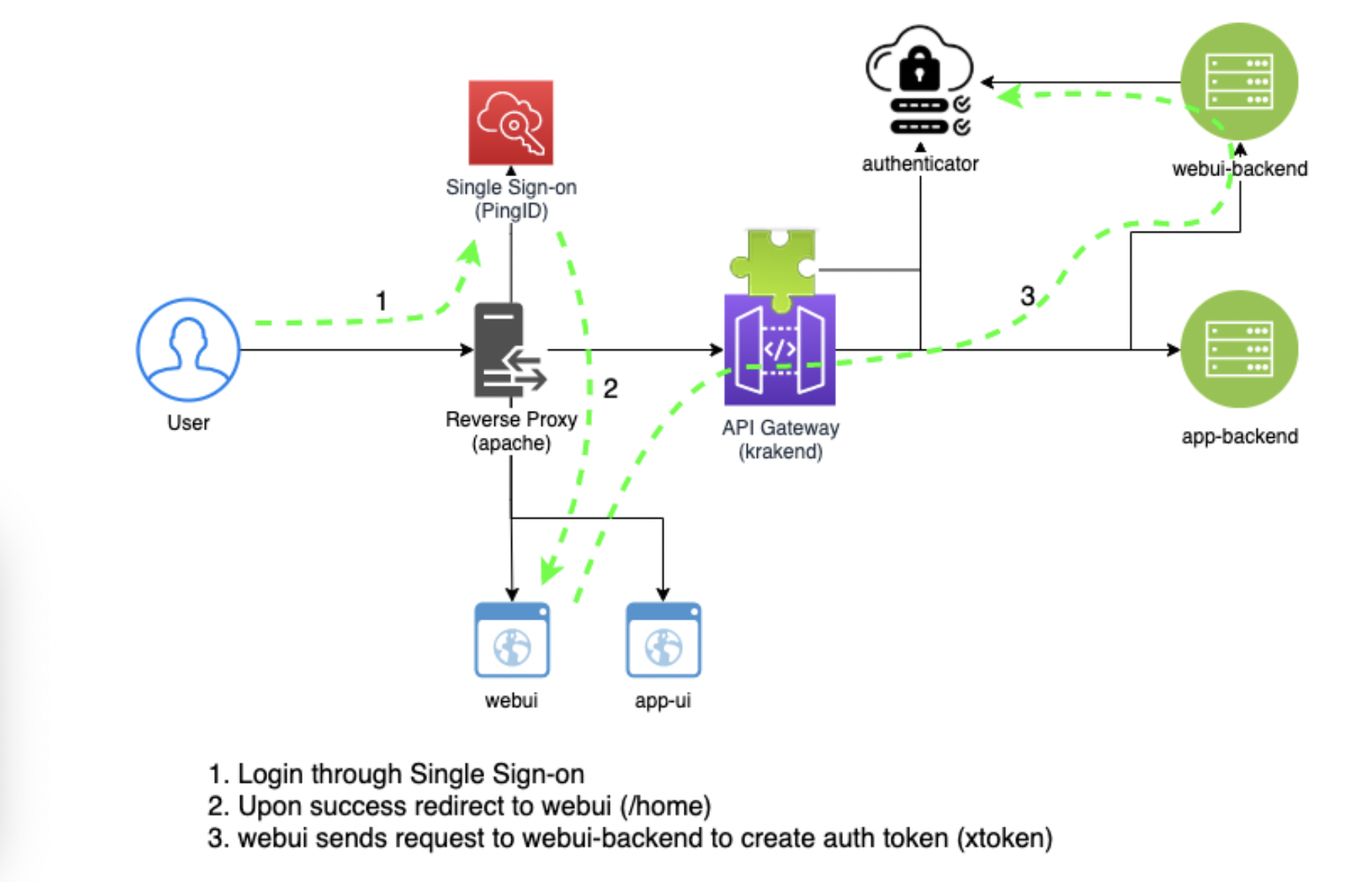
**oAuth2.0 for GNS2.0**

**Current Authentication Flow:**



1. User enters <https://gnsservices.paypalinc.com/> in the address bar of the browser.
2. Apache reverse proxy redirects it to PingID SSO

|  |
| --- |
| **apache-conf.yaml**  ...  PingFederateLoginPageURL <https://ssohrz.paypalinc.com/sp/startSSO.ping?PartnerIdpId=PPSSOALL01_OUD_2FA>  ... |

1. Post PingID authentication, the user is redirected to /home route of the webui frontend which results in a POST call to the endpoint /api/webui/login.
2. The post call is processed by the reverse proxy again, and the flow ends up hitting API Gateway (krakend) routing rules.

|  |
| --- |
| **apache-conf.yaml**  ...  ProxyPass /api/ ${A2\_KRAKEND\_URL}/api/  ProxyPassReverse /api/ ${A2\_KRAKEND\_URL}/api/  ... |

1. API gateway processes the request and routes it to the webui-backend /login endpoint passing the username information in the HTTP header.

|  |
| --- |
| {  "endpoint": **"/api/webui/login**",  "method": "POST",  "output\_encoding": "no-op",  "concurrent\_calls": 1,  "headers\_to\_pass": [  "Content-Type",  "**PF\_AUTH\_USERNAME**"  ],  "backend": [  {  "host": [  "{{ .**env.webui\_backend\_url** }}"  ],  "url\_pattern": **"/login**",  "encoding": "no-op",  "method": "POST",  "disable\_host\_sanitize": true,  "extra\_config": {  }  }  ], |

1. webui-backend processes the POST request, and sends a POST request to the /api/login which is again processed by the krakend and routed to the authenticator’s /login endpoint:

|  |
| --- |
| **GNS2.0/infra/webui/backend/app.py**  @app.route('/login', methods=['POST'])  def login():  …  r = requests.post(  '{0}/api/login'.format(KRAKEND\_URL), headers=headers)  …  **GNS2.0/infra/krakend/config/krakend.json**  "endpoints": [  {  "endpoint": **"/api/login"**,  "method": "POST",  "output\_encoding": "no-op",  "concurrent\_calls": 1,  "headers\_to\_pass": [  "username",  "password",  "Authorization"  ],  "backend": [  {  "host": [  "{{ .**env.auth\_url** }}"  ],  "url\_pattern": **"/login**",  "encoding": "no-op",  "method": "POST",  "disable\_host\_sanitize": true,  "extra\_config": {  }  }  ], |

1. The authenticator creates a session for the user and generates a JWT token which is passed down as a xtoken attributue of the HTTP response body.

|  |
| --- |
| **GNS2.0/infra/authenticator/authenticator.py**  @app.route('/login', methods=['POST'])  def login():  …  try:  with Session.begin() as session:  newuser = User(username=username, ldap\_groups="|".join(ldapGroups))  **newuser.token = jwtHelper.generateToken(username)**  newuser.updateTimeout()  session.add(newuser)  …  return jsonify(username=newuser.username, **xtoken**=newuser.token,  ldap\_groups=newuser.ldap\_groups.split('|'),  expiry\_secs\_left=newuser.ticks2datetime()), 201 |

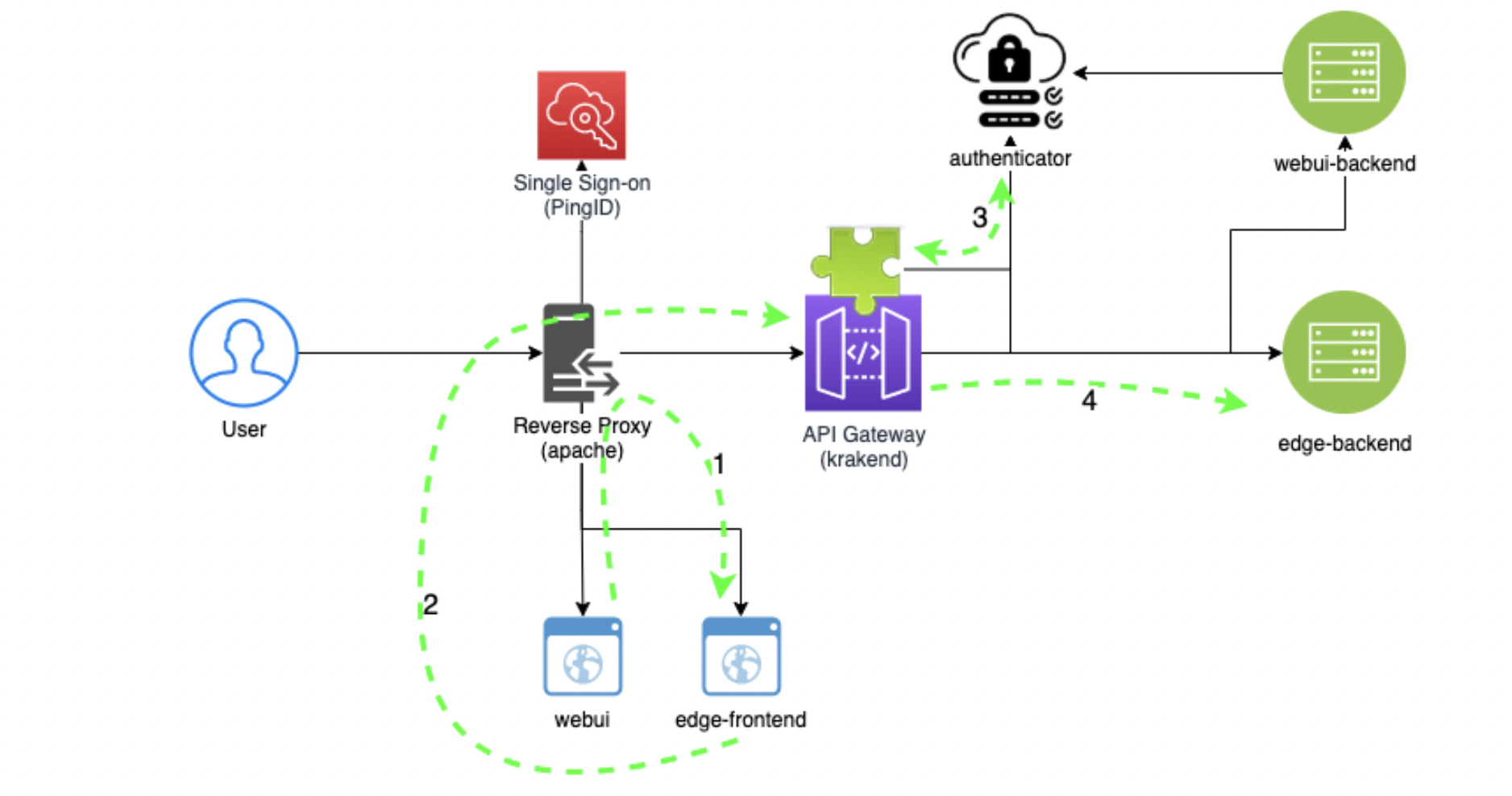
1. The HTTP response from the authenticator reaches webui-backend which processes the response body and sets the xtoken in HTTP cookie, and sends the response back to webui frontend.

|  |
| --- |
| **GNS2.0/infra/webui/backend/app.py**  @app.route('/login', methods=['POST'])  def login():  …  if r.ok:  res = r.json()  logging.error(res)  resp = jsonify(msg='Success')  resp.set\_cookie('username', res['username'], httponly=True)  **resp.set\_cookie('xtoken', res['xtoken'], httponly=True)**  return resp, 200  return jsonify(msg='Invalid credentials'), 401  … |

1. webui frontend processes the successful response, and redirects it to the home page of the webui i.e. list of GNS services.

|  |
| --- |
| **GNS2.0/infra/webui/ui/src/layouts/LoginProcess.jsx**  **…**  this.redirectURL = decodeURIComponent(queryParams.get('redirectURL') || '/home')  …  render() {  if (this.state.loggedIn) {  window.location.href = this.redirectURL  } |

**Current Authorization Flow:**

****

Every API call needs to be validated/authorized before reaching the corresponding backend service. Take Domain Parking, for example:

1. The user clicks on the Domain Parking tile on GNS Services home page.
2. Apache Reverse Proxy redirects it to the Domain Parking frontend.
3. Domain Parking UI’s landing page sends a GET request for the list of domains to /api/domainparking/domains

|  |
| --- |
| fetchDataFromBackend() {  trackPromise(  fetch(**this.backendURL(),** {  method: **'GET'**,  headers: { 'Content-Type': 'application/json' },  }))  .then(this.checkBackendErrors)  .then((res) => res.json())  .then((data) => {  this.setState({ domainsEntries:data})  })  .catch(this.handleError)  } |

1. Apache redirects the request to the API gateway
2. API gateway’s authenticator plugin sends a GET call to the authenticator to validate the request.

|  |
| --- |
| **GNS2.0/infra/krakend/plugin/auth.go**  func (r registerer) registerHandlers(ctx context.Context, extra map[string]interface{}, handler http.Handler) (http.Handler, error) {  …  **tokenHeader** := ""  tokenCookie, err := r.Cookie("xtoken")  if err == nil {  tokenHeader = tokenCookie.Value  }  …  …  rs, err := client.Do(rq)  if err != nil {  http.Error(w, err.Error(), http.StatusInternalServerError)  return  }  defer rs.Body.Close()  if rs.StatusCode != http.StatusOK {  http.Error(w, "Authentication failed", http.StatusUnauthorized)  return  }  **handler.ServeHTTP(w, r)**  …. |

1. Only when authenticator send OK status code, the previous GET request is forwarded to the application backend which can just exercise the business logic assuming that the request has been validated already.

**Advantages of the current authorization model:**

1. It’s quite simple. At present the entire authorization workflow is encapsulated in a krakend plugin and is transparent to the backend developers who only need to write the business logic.

**Disadvantages of the current authorization model:**

1. API gateway needs to validate each and every request through a POST call to authenticator. While authentication (loggin in) is a non frequent activity, API calls validation is not. This can slow down the routing process of API gateway as GNS services scale. How will this play out? A lot of activity for application1 can result in higher latencies for application2.
2. Authenticator becomes a single point of failure for GNS services. Any problem with authenticator service after the user is successful in logging in renders other applications unavailable.
3. What if we want to expose a certain features of an application for some users while hiding for others? Currently we work around this problem by sending a GET request to /api/validate/token from UI and based on the user roles or permissions present in the response, we choose to hide or display the corresponding feature of the application. This is not a cleaner approach and results in unnecessary calls (for example at load time without the feature being used in first place). UI should simply make backend calls passing down the JWT token and display appropriate error when 401 is received instead of proactively hiding the features on UI. We should progressively make UI `dumber`.
4. The backend validating an API call should be faster than the current model because there is no overhead of extra HTTP call from API gateway to authenticator. I am not saying that authenticator’s validation part is slow. If we assume that authenticator and backend should take time validating a JWT token, we still save a HTTP call.

**Proposed changes:**

1. The current login process for GNS services can stay as the user credentials are validated by the PingID SSO before the flow eventually reaches to the /login endpoint of authenticator which can just rely on PF\_AUTH\_USERNAME to generate the JWT token.
2. The validation for the API calls can be moved to backend implementation. It is quite easy to adapt the existing implementation for the REST handlers with decorator pattern. Sample code example in go as below:

|  |
| --- |
| func **decorator**(endpoint func(http.ResponseWriter, \*http.Request)) http.Handler {  return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {  fmt.Println("Authorizing.")  /\*  Add the JWT validation here  /\*  **endpoint( w, r )**  })  }  func home(w http.ResponseWriter, r \*http.Request) {  fmt.Fprintf(w, "Hello there, thanks for hitting the home page!”)  }  func handleRequests() {  http.Handle("/", **decorator(home)**)  log.Fatal(http.ListenAndServe(":8080", nil))  } |

1. Currently the JWT token is passed as xtoken HTTP cookie while making REST calls to the application backend. This needs to be slightly changed and JWT token should be passed as the Authorization header to conform to RFC6749. This is optional and using the xtoken cookie is also fine as long as it has the expiry date which it already has,.
2. Add a new endpoint for the applications willing to use the oAuth2.0 client credentials framework of GNS Services. Sample Request and Responses below:

|  |
| --- |
| **Request:**  POST <https://gnsservices.paypalinc.com/oauth2/token> >  Content-Type='application/x-www-form-urlencoded'&  **client\_id**=aSdxd892iujendek523uedj&  **client\_secret**= 2YotnFZFEjr1zCsicMWpAA&  **grant\_type**=client\_credentials      **Response (Positive):**  HTTP/1.1 200 OK  Content-Type: application/json    {  "access\_token":"eyJz9sdfsdfsdfsd",  "token\_type":"Bearer",  "expires\_in":3600  }          **Response (Negative):**  HTTP/1.1 400 Bad Request  Content-Type: application/json;charset=UTF-8    {  "error":"invalid\_request|invalid\_client|invalid\_grant|unauthorized\_client|unsupported\_grant\_type|"  }    Description of error codes in accordance with RFC6749:  invalid\_request: Missing a required paramemter (for example, grant\_type)  invalid\_client: The client has failed authentication.  invalid\_grant: The client doesn't have access to the scope to access GNS Services.  unauthorized\_client: The client is not authorized for client\_credentials grant type.  unauthorized\_grant\_type: Returned if the client uses any other grant\_type than client\_credentials. |

Later on the client application needs to pass this Bearer token as the Authorization header in the GET calls. Example below:

|  |
| --- |
| curl --location --request GET '[https://gnsservices.paypalinc.com/**someService/someResource**](https://gnsservices.paypalinc.com/someService/someResource)' \  --header 'Authorization: Bearer abc' \  --header 'Content-Type: application/json' \  --data-raw '{  "name": "some random data"  }' |

The response to the above GET call can be either `someResource` or 401 unauthorized.

Conclusion:

1. Caching in krakend – explore instead of backend
2. Pure OAuth2 ( service accounts ) and multiple tokens (rolling over expiry period)
3. Cookies vs header