KEYCLOAK

spring®

Full-stack OpenID solution

Jérôme Wacongne for eGastro GmbH

What we’ll build

- Keycloak in a multi-tenant setup
- 2 front-ends :
  - back-office with Vue.js: manage restaurants
  - Android mobile app with Flutter
- REST API
  - users: access user roles and relations to restaurants
  - realms, restaurants, menus and orders
- “Keycloak mapper” to add data from your APIs to tokens
- Keycloak admin API to create realms, roles, clients and users programmatically

UAA & Token

- « Authentication » : who (identity)
- « Authorization » : what can be done
- Token : grants from a “resource owner” to a “client”
  - authorization server identity (issuer)
  - resource-owner identity (subject)
  - client ID
  - Scope: filter to apply on resource owner grants
  - expires
  - …

OAuth2 OpenID	Keycloak	Spring-security
subject	subject	principal
Private claims	roles (realm & client)	GrantedAuthority
scope	scope	N/A

OAuth2 actors

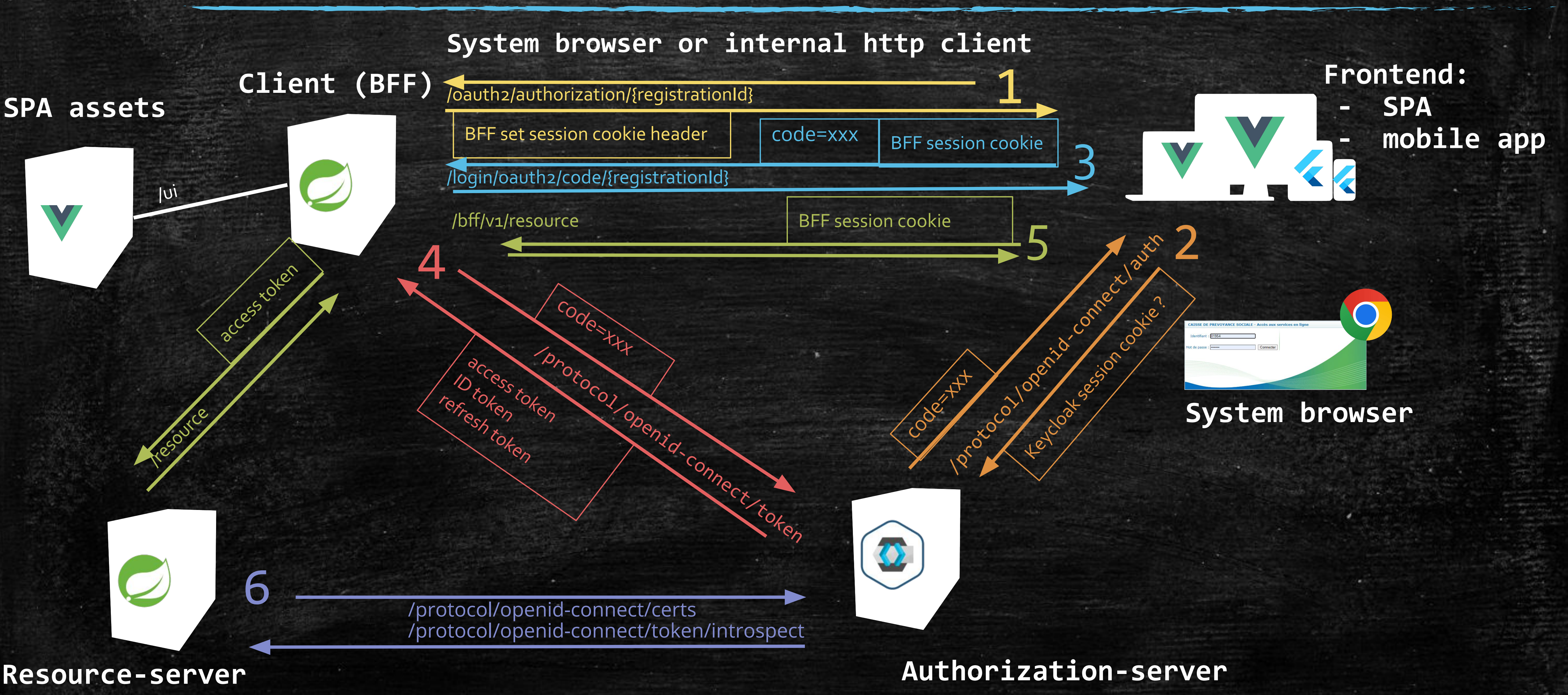
- **« Authorization server »:**
  - **<u>provides with identities</u>**
  - also known as *issuer* or *OpenID Provider* (*OP*)

- **« Resource servers »:**
  - **<u>provides with data</u>** (REST API), enforces access-control and data integrity
  - expects requests to be authorized with access tokens (which it validates)
  - can be stateless and as so, insensible to CSRF attacks

- **« Clients »:**
  - **<u>consumes data</u>** from the resource server(s) (either directly or as smart gateway proxying a single page or mobile application)
  - expects requests to be authorized with sessions => exposed to CSRF attacks
  - responsible for tokens acquisition and storage

« Authorization code » flow

applied to confidential client and SPA / mobile app



System browser or internal http client

SPA assets

Client (BFF)

Frontend:
- SPA
- mobile app

/ui

1 /oauth2/authorization/{registrationId}

BFF set session cookie header

code=xxx

BFF session cookie

3 /login/oauth2/code/{registrationId}

5 /bff/v1/resource

BFF session cookie

2 /protocol/openid-connect/auth

code=xxx

Keycloak session cookie ?

4 /protocol/openid-connect/token

code=xxx

access token
ID token
refresh token

access token

/resource

6 /protocol/openid-connect/certs
/protocol/openid-connect/token/introspect

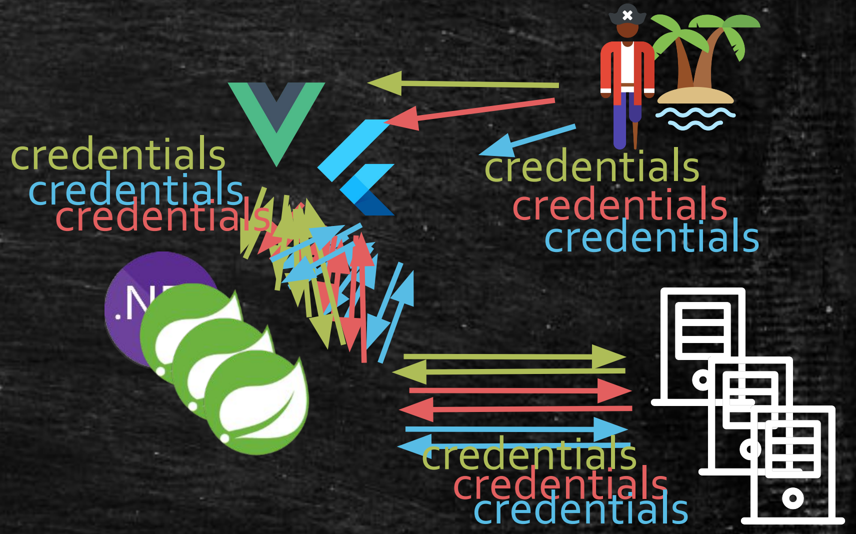
System browser

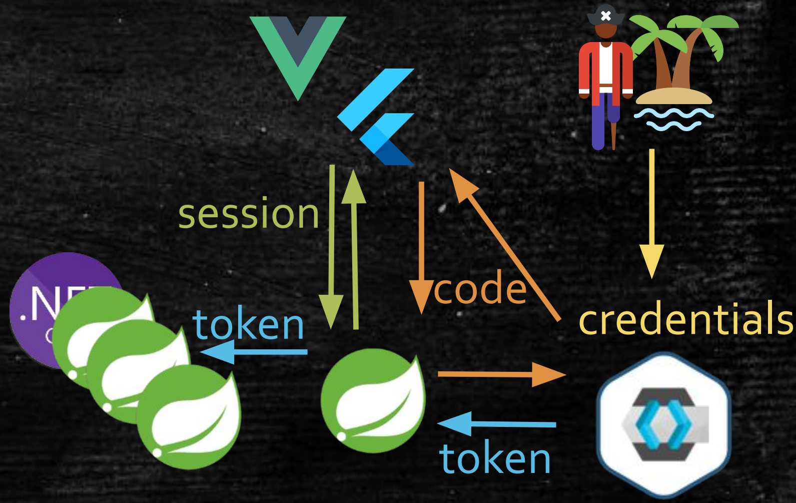
Resource-server

Authorization-server

Why using OpenID at all?

- Simplicity
  - Everything related to authentication is centralized
- UX
  - Makes it possible to share user accounts across applications
  - SSO (makes it possible to share even user sessions across apps)
- Safety
  - User credentials are manipulated by a single actor (maintained by security experts)
- Cost
  - Authentication and user accounts are developed and hosted only once
  - Many existing solutions (with UI, MFA, connectors to many identity sources, …)
- Scalability
  - stateless resource servers are fault tolerant and easy to load-balance





Configuration

Client	Resource Server	Authorization server
● Get tokens (initiates OAuth2 “flows”: authorization_code (login), client_credentials, refresh_token) ● Store token	● Validates tokens (JWT decoder or introspection) ● Implements access control	● Issues tokens ● Exposes JWK-set or introspection endpoint
● Stateful for authorization code safety and tokens storage ● Needs protection against CSRF	● Can be stateless ● Insensible to CSRF	● Stateful (user authentication status) ● Needs protection against CSRF ● Needs CORS configuration
Responds with 302 (redirect to login) to unauthorized request (missing or invalid session)	Responds with 401 (unauthorized) to unauthorized requests	Depends on the provider

Backend For Frontend Pattern

- Why:
  - Single page and mobile apps can’t keep a secret => “public” OAuth2 clients
  - Frameworks and end-user devices are more exposed to attacks (JS, storage)
  - Cookies can be flagged with “secure”, “**http-only**” and “same-site”

- Solution:
  - “confidential” client OAuth2 on server
  - sessions with CSRF protection for exchanges between terminals and servers
  - client stores tokens in session and replaces the cookie with an access token before forwarding a request from a frontend to a REST API

Spring-cloud-gateway as BFF

- spring-cloud-gateway is a reactive application (webflux)
- SecurityWebFilterChain for an OAuth2 client with oauth2Login (authorization_code flow)
- filters:
  - TokenRelay
  - DedupeHeaders
  - StripPrefix
- predicates
- SecurityWebFilterChain for an OAuth2 resource server (resources not needing a session)

REST API as resource server

- accept tokens issued by the master realm
- implement role based access control
- unit-test access control
- enhanced Authentication with domain specific data
- advanced access control rules
- dynamic multi-tenancy (accept tokens from any realm)

Keycloak “mapper”

- Use Spring’s new RestClient
- Query the restaurants API to enrich tokens with current user “grants” for each restaurant, as saved in eGastro database

Vue.js Frontend

- Check the user status on the backend (Who am I? Until when is my identification valid? What am I granted with? ...)
- Redirect the user to the BFF login entry-point
- POST to the BFF logout endpoint and redirect the user to the authorization-server endpoint

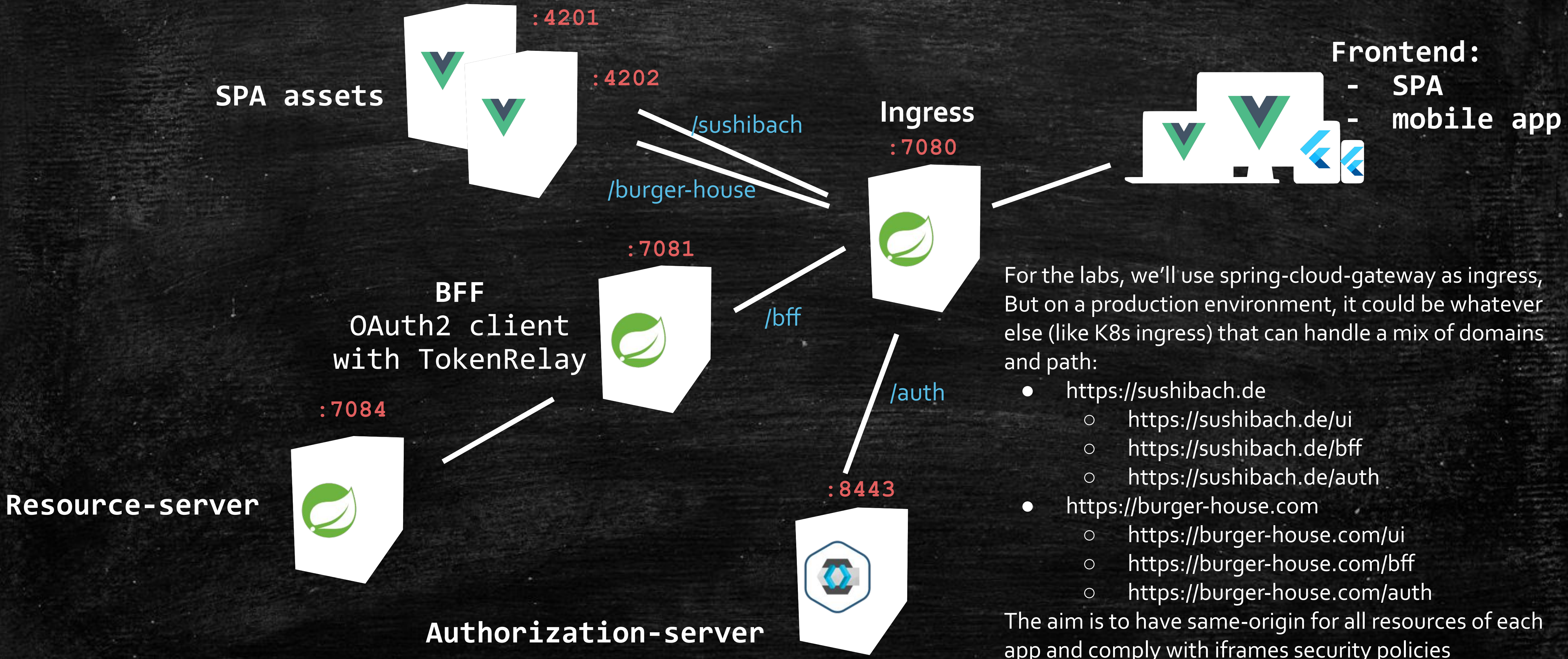
Flutter mobile frontend

- Add session and CSRF support to the http package (handle cookies)
- Check the user status on the backend (Who am I? Until when is my identification valid? What am I granted with? …)
- Redirect the user to the BFF login entry-point (ensure a session is opened) and follow to the authorization server authorization-code endpoint using system browser
- Intercept the callback with a deep link (“app” or “universal” link)
- Forward the authorization-code to the BFF
- POST to the BFF logout endpoint and then to the authorization-server endpoint

Work with Keycloak “admin” API

- Choose between two ways of authorizing requests:
  - the REST client acts on behalf of the resource owner who originated the request: forward the access token
  - the REST client acts in its own name: use a client registration with client_credentials
- Declaring and using @FeignClient with OAuth2
  - writing a RequestInterceptor to insert the access token from the security context
  - writing the configuration for using a client registration with client_credentials
- Declaring and using WebClient with OAuth2
  - writing a ServerOAuth2AuthorizedClientExchangeFilterFunction, using client_credentials without the context of an authorized user

Target architecture



For the labs, we'll use spring-cloud-gateway as ingress, But on a production environment, it could be whatever else (like K8s ingress) that can handle a mix of domains and path:

- https://sushibach.de
  - https://sushibach.de/ui
  - https://sushibach.de/bff
  - https://sushibach.de/auth
- https://burger-house.com
  - https://burger-house.com/ui
  - https://burger-house.com/bff
  - https://burger-house.com/auth

The aim is to have same-origin for all resources of each app and comply with iframes security policies

Ressources

- https://docs.spring.io/spring-security/reference/servlet/oauth2/index.html (servlets)
- https://docs.spring.io/spring-security/reference/reactive/oauth2/index.html (reactive applications like spring-cloud-gateway)
- https://github.com/ch4mpy/egastro
- https://github.com/ch4mpy/spring-addons
- https://dzone.com/articles/spring-oauth2-resource-servers
- https://quiz.c4-soft.com/ui/quizzes
- ch4mp@c4-soft.com