Feature Dynamics with Remote Config

How the frontend can stay flexible and adapt to feature changes without redeployment

Problems

- A constantly changing development backlog
- Production issues that are hard to trace even with observability tools (Sentry, logging, monitoring)
- Complex branching strategy and cherry-picking
- Environment files, vaults, and ABAC are not flexible enough
- Small feature changes still require a redeployment

Why the Old Approach Isn’t Enough

- **Development backlog changes frequently**
  - The product team often shifts priorities mid-sprint. Feature A suddenly gets delayed, while Feature B is accelerated. As a result, code must be quickly modified and rebuilt — even just to toggle a button on or off.
- **Production issues are hard to trace despite observability**
  - Errors appear only under certain user conditions. While Sentry and monitoring are in place, it still takes a long time to determine whether an error is caused by configuration or code. There’s no easy way to disable a feature directly in production.
- **Complex branching strategy & cherry-picking**
  - Even small features live in separate branches. When a hotfix is urgent, developers must cherry-pick interdependent commits. This makes review and merging prone to errors or conflicts.

Why the Old Approach Isn’t Enough (cont.)

- **Env files, vaults, and ABAC aren’t flexible enough**
  - Environment variables work well for database connections or API keys, but they don’t fit for controlling feature behavior at the UI level.
  - Vault updates or ENV changes require a service restart, or even an image rebuild. In practice, this is time-consuming since CI/CD pipelines (e.g., in Jenkins) run multiple stages (dependency checks, security scans, build steps) before the service comes back up.
- **Small changes still require redeployment**
  - Even minor updates, like changing notification text or disabling an upload form, still trigger the CI/CD pipeline. This wastes time and can disrupt other teams who are deploying at the same time.

Solution

- **Remote Config**
  - Feature configuration is loaded dynamically from a server or object storage
  - No frontend redeployment needed to modify features
  - Inspired by Firebase Remote Config
  - Provides greater flexibility for both development and operations

Basic Concept

- **Uneditable config** → remote (server, object storage)
- **Editable config** → local (cookie / override for QA/testing)
- **Merge strategy** → editable overrides uneditable
- **Async loading** → config can change at runtime

Implementation

- **Folder structure (example from this project):**
  - src/app/api/config → endpoint to fetch config
  - src/utils/featureConfig.js → main handler for decoding & merging config
  - src/constants/remoteConfigSources → fallback static config
  - src/components/ConfigInitializer.jsx → initial configuration setup
- **For full details, see the repository:**
  - https://github.com/dimaspandu/dynamic-frontend-remote-config

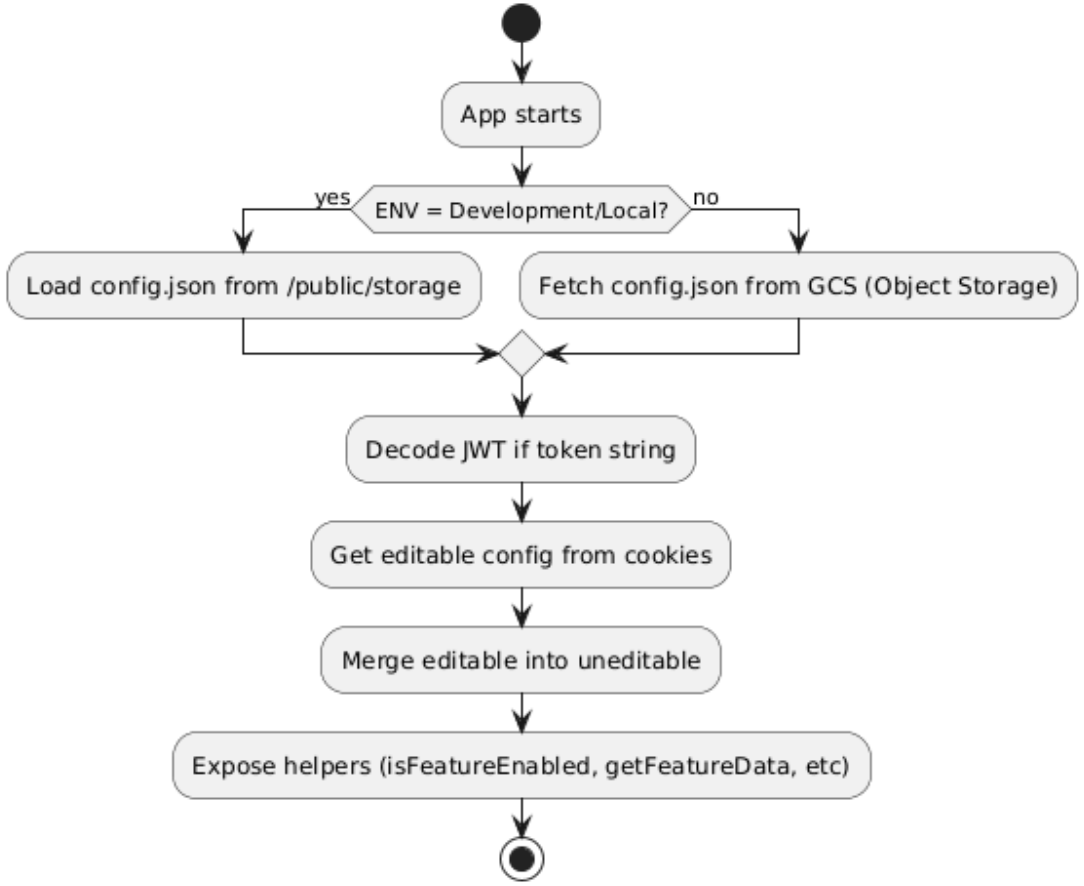
How It Works

- Frontend starts → loads config from /api/config
- If available, decode the JWT remote config
- Merge with cookie (editable config)
- Expose helper functions:
  - isFeatureEnabled(key)
  - getFeatureData(key)
  - getFeatureDataAlter(key)
  - getDecodeConfigAsync()
  - isFeatureEnabledAsync(key)

Demo

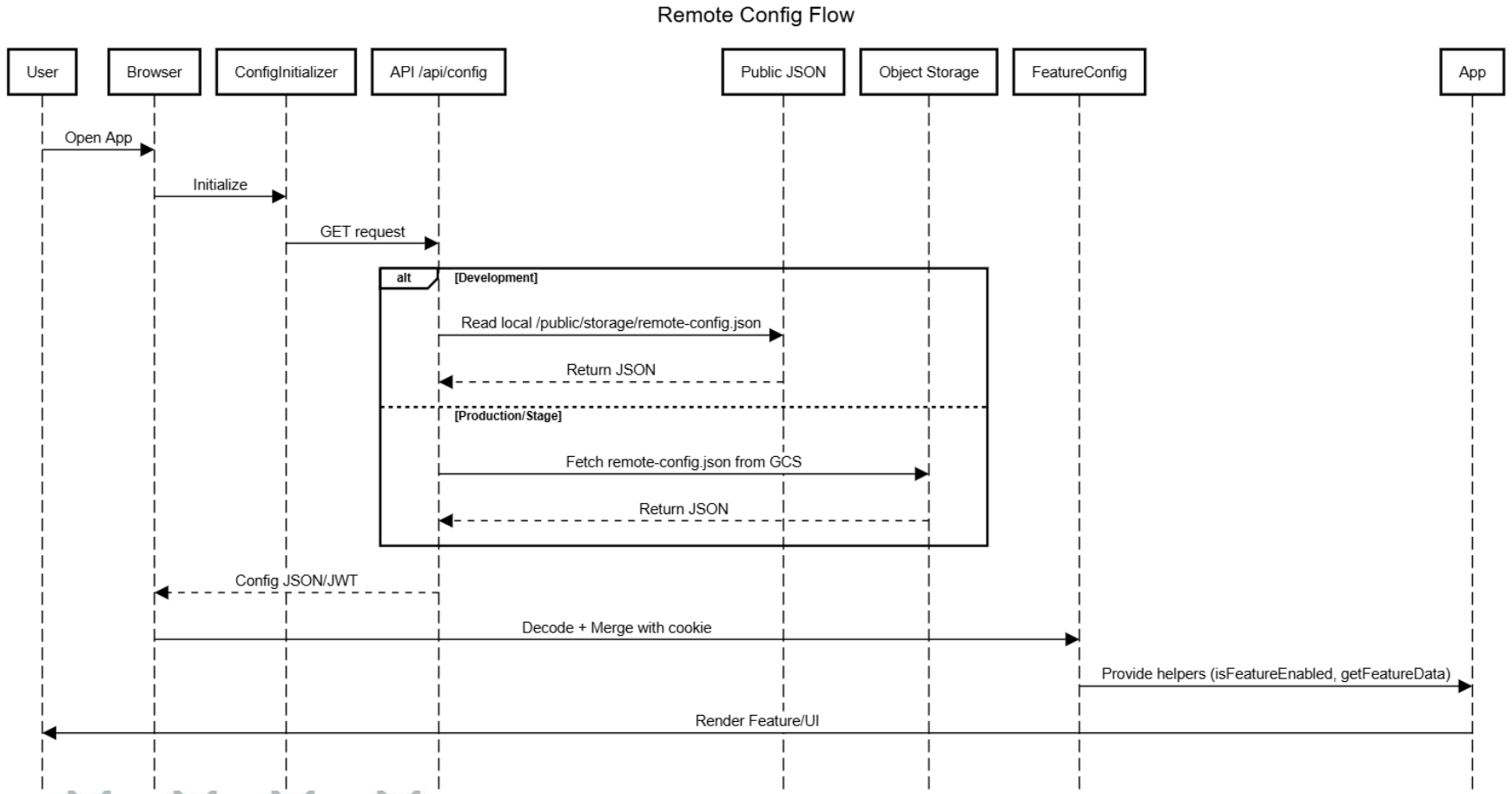
- **Homepage:**
  - https://dynamic-frontend-remote-config.netlify.app/
- **Subject Access Request (SAR):**
  - https://dynamic-frontend-remote-config.netlify.app/subject-access-request
- **Source Code (GitHub):**
  - https://github.com/dimaspandu/dynamic-frontend-remote-config

Activity Flow of Remote Config



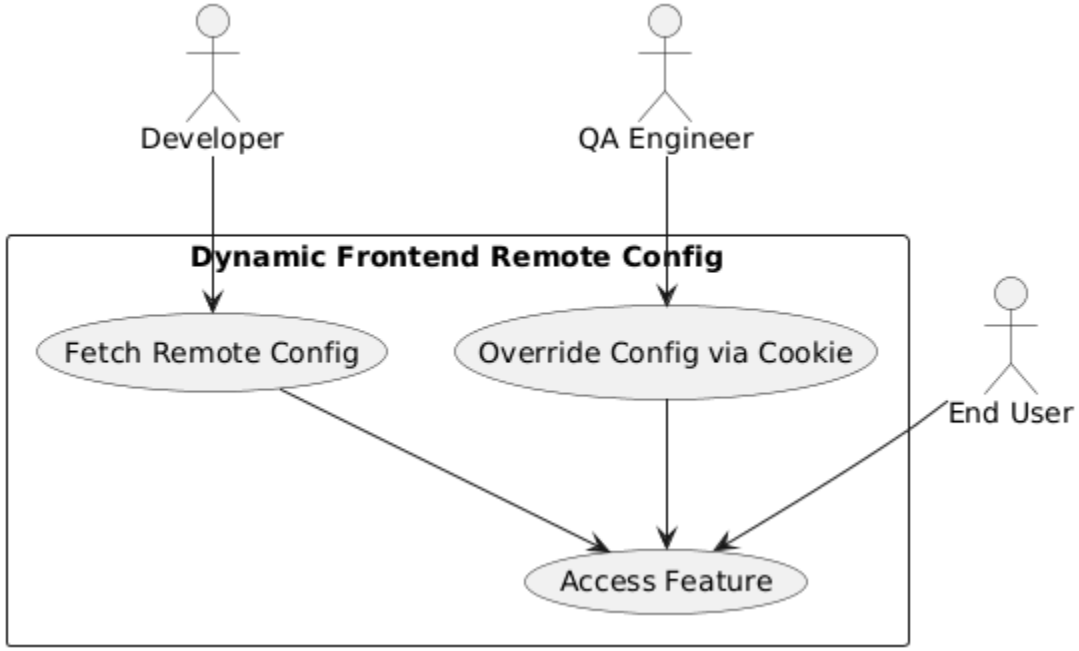
- The application starts configuration initialization.
- If ENV = Development/Local, config is loaded from /public/storage.
- If ENV = Stage/Production, config is fetched from GCS (Object Storage).
- JWT is decoded if a token is present.
- Editable config is read from cookies.
- Editable config is merged into uneditable config.
- Helpers (e.g., isFeatureEnabled) are exposed to the application.

Remote Config Request Flow



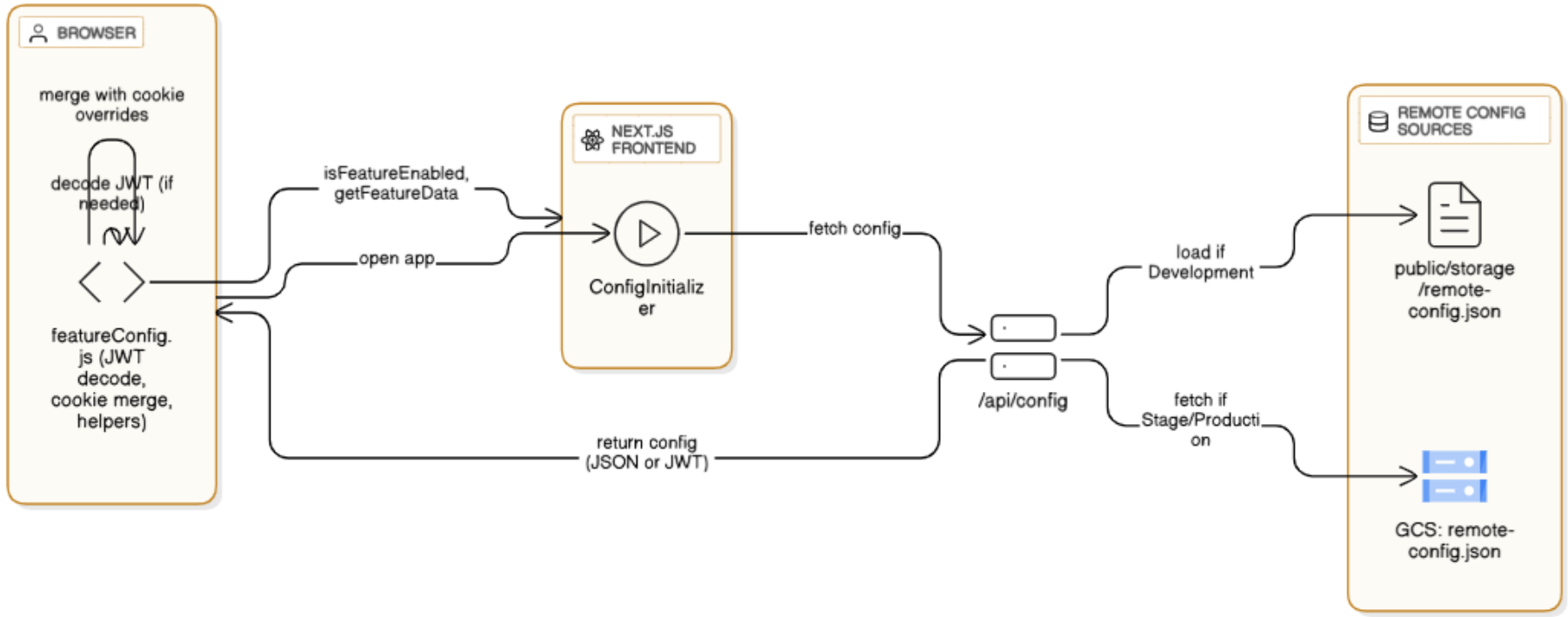
- Browser opens the application → runs ConfigInitializer.
- ConfigInitializer calls the /api/config API.
- Development: reads local JSON from public/storage.
- Stage/Production: fetches remote-config.json from GCS.
- JSON config is returned to the frontend.
- JWT is decoded and merged with cookies.
- Helpers (e.g., getFeatureData) are provided for rendering the UI.

Use Case of Dynamic Frontend Remote Config



- Developer: fetches remote config from storage.
- QA Engineer: overrides config using cookies for testing.
- End User: accesses features according to the active config.
- All actors go through the same config mechanism.

High-Level Architecture



- Browser runs the Next.js application.
- ConfigInitializer triggers configuration retrieval.
- Config can be local JSON (dev) or from GCS (prod/stage).
- API Route /api/config acts as the bridge.
- Config is processed in featureConfig.js (JWT decoding, merging, helper exposure).
- Reduces the need for redeployment for minor feature changes.

Implementation

- **How Remote Config Works**
  - The default remote config code serves as the main source of truth for the frontend application.
  - In local development, this file is used directly to design configurations.
  - During the build/dev process, the file is converted into remote-config.json and stored in the /public/storage folder.
  - In production, remote-config.json is typically placed in object storage (e.g., Google Storage) and dynamically fetched by the application.
- For full details, see the repository:
  - https://github.com/dimaspandu/dynamic-frontend-remote-config/blob/main/src/constants/remoteConfigSources/index.mjs

Implementation (cont.)

- **Configuration Structure**
  - **ticket_code**: Internal ID for reference or tracking.
  - **enabled**: Determines whether the feature is globally active.
  - **editable_with_cookies**: If true, the feature can be overridden via cookies (useful for QA or debugging).
  - **data**: Main metadata for the feature (e.g., dropdown options, UI messages, validation rules).
  - **data_alter**: Shadow config used for schema migration without causing downtime or breaking live features.
- For full details, see the repository:
  - https://github.com/dimaspandu/dynamic-frontend-remote-config/blob/main/src/constants/remoteConfigSources/index.mjs

Implementation (cont.)

- **Usage in Code**
  - The frontend calls utilities like isFeatureEnabled("UI_MESSAGES") to check if a feature is active.
  - If active, the application reads the data content to determine behavior (e.g., displaying UI messages or validating file uploads).
  - If the data structure changes, the application can (or is directed to) read from data_alter so that both live (old) and new versions remain compatible.
- For full details, see the repository:
  - https://github.com/dimaspandu/dynamic-frontend-remote-config/blob/main/src/constants/remoteConfigSources/index.mjs

Example Use Case

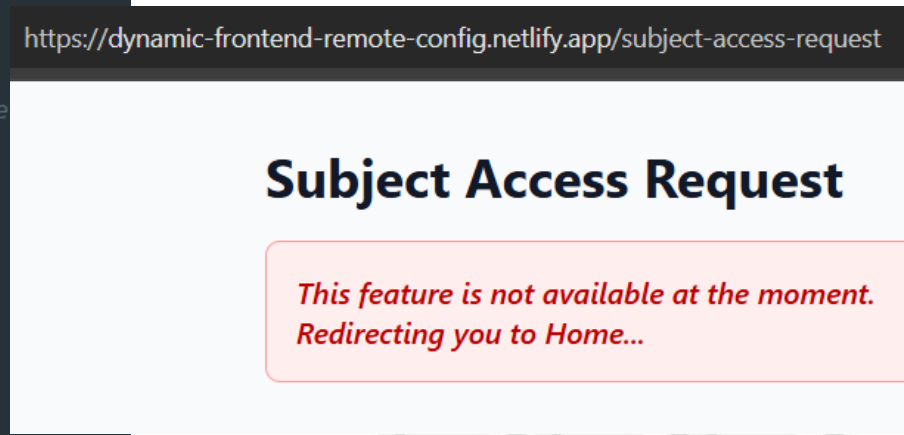
- **Feature: Subject Access Request (SAR)**
  - If SUBJECT_ACCESS_REQUEST.enabled = false → redirect to home
  - If true → display a form with dynamic fields (profileFields)

```
SUBJECT_ACCESS_REQUEST: {
  ticket_code: "F20-10007",
  enabled: false, // not released yet
  editable_with_cookies: false,
  data: {
    profileFields: {
      fullName: true,
      email: true,
      phoneNumber: true,
      address: false,
    },
  },
  data_alter: {},
},
```

```
useEffect(() => {
  async function checkFeature() {
    // Check if the SAR feature is enabled
    const available = await isFeatureEnabledAsync("SUBJECT_ACCESS_REQUEST");
    setEnabled(available);

    if (available) {
      // If enabled, load the feature config and extract profile fie
      const config = await getDecodeConfigAsync();
      setFields(config?.SUBJECT_ACCESS_REQUEST?.data?.profileFields
    } else {
      // If disabled, redirect to home after 3 seconds
      const timer = setTimeout(() => {
        router.push("/");
      }, 3000);
      return () => clearTimeout(timer);
    }
  }

  checkFeature();
}, [router]);
```



Case Study: QA Testing with Remote Config

- **Imagine QA is testing the Resend OTP feature.**
  - By default, the application sets a 30-minute cooldown for each OTP resend.
  - During testing, QA needs to resend OTP multiple times without waiting that long.
  - Developers can add a special configuration object, for example:

```
RESEND_OTP: {
  ticket_code: "F20-20001",
  enabled: true,
  editable_with_cookies: true,
  data: {
    cooldownSeconds: 1800 // default 30
  },
  data_alter: {}
}
```

Case Study: QA Testing with Remote Config (cont.)

- **With Remote Config:**
  - Because **editable_with_cookies**: **true**, QA can override cooldownSeconds via a JWT cookie to just 5 seconds.
  - Testing becomes faster without affecting other users in production.
  - After testing, the configuration is reverted to the default value (30 minutes).
- **Important Note:**
  - ***This configuration design must be isolated to QA browsers/accounts only, so it does not impact end users.***

Implementation Tips

- If a configuration only needs to be processed once (e.g., during login or app bootstrap), it can be placed in middleware so it doesn’t need to be checked on every page/router change.
- Developers are free to design the object structure as needed: some can be simple flags, others can be complex nested objects for various feature variations.
- Main principle: the more consistent the object structure, the easier it is to use the configuration across features.

Benefits

- **Fast** → no redeployment needed
- **Flexible** → features can be enabled/disabled per user or role
- **Secure** → the original config URL is loaded via API gateway/proxy and can be signed (JWT)
- **Adaptive** → supports A/B testing, gradual rollout, and feature flagging

Conclusion

- **Remote Config enables the frontend to:**
  - Adapt more quickly to changing requirements
  - Reduce release management overhead
  - Provide granular control over application behavior

Important Notes!!!

- **Remote Config must not be used for:**
  - Storing credentials (API keys, secrets)
  - Sensitive or confidential user data
  - Information that requires end-to-end encryption

Thank You

Dimas Pandu Pratama