ChatGPT

Plano de Desenvolvimento do MVP de Biblioteca de Consentimento de Cookies

1. Checklist Técnico: Requisitos Mínimos e Boas Práticas

- **Consentimento Livre e Explícito:** Deve-se obter consentimento **prévio** para qualquer cookie **não essencial**, em conformidade com a LGPD [1]. Isso implica que cookies estritamente necessários podem ser habilitados por padrão (base legal de interesse legítimo), mas todos os demais devem permanecer **desativados até o usuário consentir**. Não usar caixas pré-marcadas nem inferir consentimento pelo uso contínuo do site [2]. O banner deve oferecer opção de **aceitar ou recusar**, garantindo escolha efetiva sem consequências negativas [3] [4].

- **Armazenamento Local (Cookie):** As preferências de consentimento do usuário serão armazenadas em um cookie do lado do cliente. Esse cookie deve ser persistente (ex: expirar em 6–12 meses) para lembrar a escolha do usuário, porém respeitando diretrizes de minimização (limitar duração ao necessário) [5]. Utilize atributos adequados (ex: `SameSite=Lax` ou `Strict`, `Secure` se HTTPS) para segurança. Não enviar dados a servidor; tudo é armazenado localmente no navegador do usuário.

- **Suporte a SSR (Next.js):** A biblioteca deve funcionar tanto em ambiente de *client-side* (React puro) quanto em *server-side rendering* (Next.js). Isso significa evitar referências diretas a `window` ou `document` durante a renderização inicial. No Next.js, a preferência de consentimento pode ser lida no servidor (via `cookies()` ou contexto do `NextRequest`) e passada para o React, evitando efeito "flash" do banner. Ferramentas como **react-cookie** (baseada em Universal Cookie) facilitam esse suporte, fornecendo o componente `CookiesProvider` e hooks de leitura/escrita que funcionam no cliente e servidor [6]. Caso não se use react-cookie, pode-se manualmente ler o cookie de consentimento em `getServerSideProps`/*middleware* e injetar no estado inicial do contexto.

- **Acessibilidade:** Seguir as diretrizes WAI-ARIA e boas práticas para que tanto o banner quanto o modal sejam navegáveis via teclado e utilizáveis por leitores de tela. Isso inclui:

- **Foco gerenciável:** Quando o banner aparecer, colocar o foco inicialmente no botão de ação (por exemplo, "Aceitar") ou em um elemento de texto descritivo. No modal de preferências, usar componentes de diálogo do MUI (que já implementam *focus trap* e rotação de foco).
- **Labels e Contrastes:** Fornecer labels claros nos botões (ex: "Aceitar todos", "Recusar não essenciais", "Salvar preferências") e nos toggles de categorias. Garantir contraste de cores adequado no tema padrão e nas customizações.
- **Elementos semânticos:** Utilizar componentes MUI apropriados (como `Dialog`, `DialogTitle`, `DialogContent`, `FormGroup`, `FormControlLabel`, `Switch`, etc.) que já possuem acessibilidade. Incluir aria-label/aria-describedby quando necessário (ex: descrever finalidade dos botões e switches).

- **Personalização via Props:** A biblioteca deve permitir **alta customização** sem necessidade de *fork*. Preferencialmente, expor propriedades React para textos (rótulos, descrições), cores/tema (permitir passar um tema MUI customizado ou sobrescrever estilos padrão) e até estrutura (ex.: permitir optar entre banner tipo barra inferior ou modal central, etc.). Essa personalização via props deve ser completamente *type-safe*: usar tipos com `Readonly<T>` para garantir que as configurações passadas não sejam alteradas dentro da lib, seguindo boas práticas de imutabilidade. Por exemplo, definir as categorias de cookie como uma lista `readonly` ou tupla constante, assegurando que a biblioteca trate-as como constantes.

- **Independência de Backend:** O MVP não envolve registro de consentimentos no servidor. Assim, toda a lógica ocorre no front-end. Isso significa que **não há dependências de APIs** para salvar ou buscar dados de consentimento. A vantagem é reduzir complexidade inicial e preocupações de compliance de armazenamento. No futuro, caso seja necessário logar consentimentos para auditoria, pode-se adicionar opcionalmente um callback ou integração, mas **fora do escopo do MVP**.

- **Carregamento Condicional de Scripts:** Nenhum script de terceiros (tracking, analytics, ads) deve ser carregado antes do consentimento. A biblioteca deve fornecer mecanismos para que, **após o consentimento**, os scripts sejam inseridos dinamicamente (por exemplo, criando elementos `<script>` somente se a categoria pertinente for aceita). Deve-se contemplar categorias típicas como *analytics*, *marketing/ads*, *funcionalidade*. Por exemplo, só injetar o Google Analytics se o usuário consentiu na categoria "Analytics". (Ver diagrama em seção 5.2 abaixo). Também deve ser possível reverter: se o usuário retirar o consentimento posteriormente, scripts futuros daquela categoria não devem carregar (e idealmente cookies existentes poderiam ser apagados, embora isso seja funcionalidade avançada pós-MVP).

- **Cookie Banner não bloqueador de UI:** Por usabilidade, o banner deve ocupar parte da tela (tipicamente rodapé ou topo) sem impedir a navegação básica (a não ser que a política da empresa exija um *modal* bloqueante inicial). O importante é que o usuário possa acessar informações de privacidade antes de consentir. Incluir link para a Política de Privacidade ou Política de Cookies no banner (ex.: "Saiba mais em nossa Política de Cookies") e/ou no modal, atendendo ao princípio de transparência da LGPD [7] [8].

- **Revogação e Preferências:** Implementar um **modal de preferências** de cookies onde o usuário possa ajustar sua decisão a qualquer momento. De acordo com a LGPD, deve existir um procedimento fácil e gratuito para o titular revogar consentimento posteriormente [9]. Assim, além do banner inicial, a biblioteca deve expor um componente ou método para reabrir as preferências (por exemplo, um botão "Gerenciar cookies" que o desenvolvedor possa colocar no rodapé do site após o usuário já ter escolhido). Este modal listará categorias de cookies com toggles para on/off, respeitando a escolha do usuário em tempo real.

- **Segurança e Integridade:** Embora sem backend, manter boas práticas de segurança no front-end: por exemplo, caso a biblioteca injete scripts de terceiros, fazê-lo de forma segura (preferir usar tags `<script>` com `src` hosteado confiável e atributos como `async` / `defer` quando aplicável). O cookie de consentimento em si não carrega dados sensíveis, mas ainda assim deve ser protegido contra acesso indevido (httpOnly não se aplica pois o script cliente precisa ler, mas SameSite ajuda a reduzir riscos).

- **Registro de Consentimento (futuro):** O MVP não registrará no servidor, porém é boa prática estruturar o código de forma que seja possível adicionar no futuro uma função callback ao

evento de consentimento (ex.: para enviar a informação de consentimento a um endpoint ou ao *Consent Management Platform* da empresa). Assim, arquitetar o estado de modo centralizado (contexto) torna mais fácil estender com logging futuro.

Resumindo, o checklist garante que o MVP atenda aos requisitos legais básicos (consentimento livre, possibilidade de revogar, transparência) e técnicos (compatibilidade SSR, usabilidade, segurança), servindo como base sólida para evoluções futuras.

2. Roadmap de Desenvolvimento (MVP vs. Futuro)

Fase 0: Planejamento e Setup do Projeto – Iniciar configurando a estrutura do repositório como um monorepo (por exemplo, usando PNPM Workspaces ou Turborepo) caso planeje-se separar pacotes no futuro. No MVP, pode haver apenas um pacote principal (ex: `cookie-consent-lib`) e talvez um pacote de exemplo. Configurar ferramentas de build modernas como **tsup** (simples para libs TS) ou **Vite** (em modo library) para gerar bundles otimizados (CJS/ESM) [10] [11]. Adicionar o suporte ao TypeScript desde o início, com configurações de *lint* (ESLint + Prettier) alinhadas ao projeto existente do usuário. Executar `tsup` ou `vite build` deve produzir um pacote leve, tree-shakeable, sem dependências desnecessárias.

Fase 1: Implementação do Contexto de Consentimento (MVP) – Desenvolver o núcleo da biblioteca: - Criar o **Contexto React** (`ConsentContext`) com estado global das preferências de cookies do usuário. Implementar um Provider (`ConsentProvider`) que encapsula a lógica de: verificar cookie existente, definir estado inicial (ex.: `{ consentido: false, prefs: {analytics: false, marketing: false, ...} }`), e atualizar o cookie conforme o usuário interage. Conforme a implementação do usuário no AlertDialogContext, usar `useState` ou `useReducer` internamente e expor funções de manipulação no contexto (como `acceptAll()`, `rejectAll()`, `setPreference(categoria, valor)` e `openPreferences()` para exibir modal) com uso de `useCallback` [12] [13]. Assegurar que o contexto seja inicializado com base no cookie (no cliente) ou com dados injetados (no servidor, se possível). - Garantir que a escrita/leitura no cookie siga o consentimento dado. Pode-se usar **js-cookie** para facilidade: `Cookies.set(name, value, { expires: 365 })` grava um cookie de forma simples [14]. Como notado, js-cookie é leve, sem dependências e compatível com todos browsers [10]. Por outro lado, para SSR, integrar com **react-cookie**: envolver a aplicação em `<CookiesProvider>` e usar `useCookies` hook se preferir approach mais "React" [15] [6]. Nesse MVP, uma estratégia é usar js-cookie no cliente (simples) e, documentar que para SSR completo, o integrador pode opcionalmente usar `CookiesProvider` do react-cookie para hidratar o estado inicial (para evitar flash). - **Saída desta fase**: contexto funcionando e testado isoladamente (p. ex., com um componente de teste configurando `<ConsentProvider>` e exibindo valores do contexto para validar persistência em cookie). Nenhuma UI ainda, apenas lógica.

Fase 2: Componentes de UI do Banner e Modal (MVP) – Desenvolver os componentes visuais utilizando **Material-UI (MUI)**: - **Banner de Cookies:** um componente fixo (geralmente posicionado no rodapé ou header conforme design). Usar preferencialmente componentes do MUI como base: poderia ser um `Snackbar` ou um `Paper` com posicionamento fixo. O banner exibirá um texto curto ("Este site utiliza cookies…") + botões de ação. Boas práticas: um botão para "Aceitar todos" (estilo positivo), um botão "Recusar" (opcional, estilo secundário) e um botão "Preferências…" para abrir o modal. Caso deseje-se simplicidade no MVP, pode ter apenas "Aceitar" e "Configurações" e assumir que não aceitar implica recusa dos não essenciais. Integrar esses botões ao contexto: `onClick` de Aceitar chama `acceptAll()` do contexto (marcando todos não necessários como true, atualizar cookie e fechar banner), Recusar chama `rejectAll()` (marca não essenciais false, cookie e fecha banner). O banner só deve ser renderizado se o usuário ainda **não definiu** consentimento – ou seja, condicional em algo

como `!consentState.consentido`. O próprio Provider pode controlar isso e talvez inserir o banner automaticamente (ver fase 3). - **Modal de Preferências:** usar o componente `<Dialog>` do MUI para criar um modal centralizado. Dentro, incluir um `<DialogTitle>` ("Preferências de Cookies"), `<DialogContent>` com descrição e as categorias listadas. Para cada categoria de cookie (excluindo os estritamente necessários), fornecer um controle de toggle – pode ser um `FormControlLabel` com um `<Switch>` ou `<Checkbox>`. Por exemplo: `[ ] Analytics – permitir cookies de análise de desempenho`. O valor desses switches deve estar vinculado ao estado do contexto (ex: `context.prefs.analytics`). Alterações (onChange) chamam métodos do contexto para atualizar o estado (`setPreference("analytics", true/false)`). Opcional: incluir breve descrição de cada categoria ao lado do toggle, para cumprir requisito de informação clara [16] [17]. No rodapé do modal (`DialogActions`), um botão "Salvar" que fecha o modal (e marca consentido=true se ao menos uma preferência foi escolhida, ou simplesmente sempre marca consentido=true uma vez que o usuário interagiu). - **Renderização condicional:** O modal de preferências só aparece quando solicitado (ao clicar em "Configurações" no banner ou via método de contexto). O banner, por sua vez, esconde-se assim que o usuário toma uma decisão (qualquer que seja). - **Testes básicos:** Verificar a responsividade (MUI Dialog é responsivo; usar `fullScreen` prop em pequenos dispositivos se necessário para melhor UX). Verificar acessibilidade: tentar navegar via Tab no modal, fechar com Esc (MUI cuida disso). Testar se os toggles de categorias correspondem corretamente aos cookies esperados (por exemplo, marcar Analytics e salvar deve criar o cookie com essa preferência).

Saída da fase 2: UI integrada com contexto, possibilitando todo o fluxo: banner -> modal -> salvar preferências -> cookies setados.

Fase 3: Integração no Projeto Alvo e Ajustes (MVP) – Após criar a lib, integrá-la no projeto React/Next existente (do usuário, possivelmente MSQD): - **Publicação Local/NPM:** Publicar a biblioteca, seja em um registry privado (ex.: GitHub Packages ou Azure Artifacts) ou no npm público se open-source. Se "MSQD" refere-se a um repositório interno ou marketplace da organização, preparar o pacote para atender aos critérios dele (versão, changelog, licença, etc.) e realizar a publicação. - **Instalação no projeto do usuário:** Adicionar a dependência e envolver a aplicação com o `<ConsentProvider>` no `_app.tsx` (Next.js) ou no nível raiz. Verificar que o banner aparece como esperado em ambiente real. - **Teste de SSR:** Executar a aplicação Next.js em modo produção e garantir que nenhuma quebra ocorre no servidor. Idealmente, implementar no `_document` ou `_app` uma lógica para pegar o cookie de consentimento no request e passá-lo para o Provider (por exemplo, via `<ConsentProvider initialPrefs={...}>`). Se não, pelo menos confirmar que o banner piscando não prejudica a experiência. - **Ajustes Finais:** Coletar feedback de design (talvez ajustar estilos do banner para combinar com identidade visual do projeto – ex.: usar tema MUI do sistema). Verificar textos em português e ortografia. - **Documentação:** Redigir um README ou documentação interna explicando como usar a biblioteca – especialmente as props de customização, e exemplos de código de integração (parecido com o fornecido na seção 6 abaixo).

Saída fase 3: MVP da biblioteca funcionando no projeto, com documentação mínima.

Funcionalidades Pós-MVP (Roadmap Futuro): - **Multi-idioma (i18n):** Estrutura para facilmente adicionar outros idiomas além do pt-BR. Ex: permitir prop `language` ou um contexto de tradução, ou simplesmente aceitar um objeto de strings traduzidas. Integrar com i18n do Next.js ou usar uma biblioteca como i18next se necessário. - **Mais categorias e descrição detalhada:** Incluir suporte a categorias customizadas definidas pelo integrador (além das padrão). Isso requer generalizar o contexto para talvez aceitar uma lista de categorias via props. - **Cookies de preferência granular:** Caso o projeto precise, diferenciar cookies de funcionalidade vs desempenho vs marketing explicitamente (a

ANPD cita categorias como necessárias, funcionais, desempenho, publicidade [18] [19]). No MVP usamos um modelo simples (ex.: analytics e marketing), mas futuramente expandir. - **Logging/Auditoria de consentimento:** Opcionalmente enviar um registro ao backend quando o usuário der ou mudar consentimento, para comprovação (como a LGPD exige prova de consentimento livre [20]). Poderia integrar com um API de analytics ou banco. - **Suporte a "opt-in tardio" de scripts:** Atualmente, no MVP, os scripts autorizados podem ser carregados imediatamente após consentir (via efeito ou callback). Em versões futuras, pode-se adicionar integração com gerenciadores de tag (GTM) ou disparar eventos globais que sistemas de tracking escutem para só então ativar (ex.: `window.dataLayer.push(consentGranted)`). - **UI aprimorada e temas:** Incluir vários layouts de banner (banner inferior vs modal inicial vs barra superior). Disponibilizar temas pré-definidos (claro/escuro) e melhora na acessibilidade visual (ex.: animações suaves, transição ao aparecer/desaparecer). - **Teste e Certificação:** Realizar testes com diferentes navegadores e dispositivos. Validar em ferramentas de acessibilidade (Lighthouse a11y, axe) e conformidade legal se possível. - **Publicação open-source (se for o caso):** Considerar abrir o código se for de interesse, para obter contribuições. Garantir que nenhuma informação sensível do projeto esteja no repositório antes de abrir.

Em termos de **cronologia**, as fases 1 a 3 correspondem ao MVP básico e poderiam ocorrer em 1 ou 2 sprints. As funcionalidades pós-MVP seriam planejadas conforme necessidade e prioridade, garantindo primeiro a conformidade legal e funcional, depois conforto e extensibilidade.

3. Proposta de Arquitetura de Código

Para tornar a biblioteca **modular, reutilizável e extensível**, sugere-se a seguinte estrutura de pastas e componentes:

```
cookie-consent-lib/
├── src/
│   ├── context/
│   │   └── ConsentContext.tsx    # Define o createContext, Provider e hooks
useConsent
│   ├── components/
│   │   ├── CookieBanner.tsx      # Componente do banner de consentimento
│   │   └── PreferencesModal.tsx  # Componente do modal de preferências
│   ├── hooks/
│   │   └── useConsent.ts         # (Opcional) Hook custom para consumir
contexto
│   ├── utils/
│   │   ├── cookieUtils.ts        # Funções para ler/escrever cookies (usando
js-cookie ou nativo)
│   │   └── scriptLoader.ts       # Função util para inserir scripts
condicionalmente
│   ├── types/
│   │   └── types.ts              # Tipos TypeScript (ex.:
ConsentPreferences, Category, etc.)
│   └── index.ts                  # Ponto de entrada, exporta componentes e
hooks
├── package.json
└── tsconfig.json
```

Contexto e Estado Global: `ConsentContext.tsx` exportará `ConsentProvider` e possivelmente dois hooks: `useConsentState` e `useConsentActions` (padrão separar estado de ações) ou um único `useConsent()` que retorna tudo. Dentro do Provider, usaremos `useReducer` para gerenciar ações (aceitar tudo, recusar tudo, togglar categoria, abrir/fechar modal). Isso facilita adicionar novos tipos de ação sem proliferar muitos `useState`. Por exemplo, uma interface para nosso estado:

```
interface ConsentPreferences {
  analytics: boolean;
  marketing: boolean;
  // ... outras categorias no futuro
}
interface ConsentState {
  consentido: boolean;
  preferencias: ConsentPreferences;
  modalAberto: boolean;
}
```

Inicialmente, `consentido` será falso até o usuário tomar alguma decisão. `preferencias` inicia com valores default (false para todas categorias não essenciais; cookies necessários não precisam constar pois são sempre ativos por definição). `modalAberto` controla exibição do modal.

As **ações** do reducer podem ser: `'ACEITAR_TUDO'` (marca todas prefs como true, consentido true), `'RECUSAR_TUDO'` (prefs false, consentido true), `'TOGGLE_CATEGORIA'` (inverte ou define uma categoria específica), `'ABRIR_MODAL'` e `'FECHAR_MODAL'`. O contexto proverá o `state` atual e um `dispatch` ou funções derivadas para cada ação.

Essa centralização facilita evolução e testes, e isola a lógica de consentimento (por exemplo, garantir que ao aceitar tudo ou recusar tudo o cookie seja escrito corretamente com todos valores). Conforme exemplo do gist, atualizar o cookie dentro de um `useEffect` que observa o state [21]. Assim qualquer mudança em `state.preferencias` ou `state.consentido` persiste no cookie JSON.

Componentes: O `CookieBanner.tsx` consome o contexto e decide renderizar ou não. Estrutura em JSX:

```
<Snackbar open={mostrarBanner} anchorOrigin={{ vertical: 'bottom',
horizontal: 'center' }}>
  <Paper sx={{ p:2, display: 'flex', alignItems: 'center', justifyContent:
'space-between' }}>
    <Typography>Este site utiliza cookies para melhorar a experiência.</
Typography>
    <Stack direction="row" spacing={1}>
      <Button variant="contained" color="primary" onClick={acceptAll}
>Aceitar todos</Button>
      <Button variant="outlined" color="inherit" onClick={declineAll}
>Recusar</Button>
      <Button variant="text" onClick={openPreferences}>Preferências</Button>
    </Stack>
```

```
  </Paper>
</Snackbar>
```

Acima é um exemplo: usamos MUI Snackbar/Paper para o layout (ou poderíamos usar `Alert` do MUI para estilizar). Os botões chamam funções do contexto (`acceptAll` etc.). Note que estamos utilizando componentes MUI padrão e estilo por props (`variant` , `color`). Isso facilita consistência visual e respeito ao tema global.

O `PreferencesModal.tsx` seria algo como:

```
<Dialog open={modalAberto} onClose={closeModal} aria-labelledby="cookie-pref-
title">
  <DialogTitle id="cookie-pref-title">Preferências de Cookies</DialogTitle>
  <DialogContent dividers>
    <FormGroup>
      {/* Exemplo de categoria Analytics */}
      <FormControlLabel control={
          <Switch checked={prefs.analytics} onChange={(e) =>
toggleCategory('analytics', e.target.checked)} />
        }
        label="Cookies Analíticos (medir uso do site)" />
      {/* Repetir para outras categorias */}
    </FormGroup>
    <Typography variant="body2" sx={{ mt: 1 }}>Você pode ajustar suas
preferências. Cookies necessários são sempre usados pois garantem
funcionalidade básica.</Typography>
  </DialogContent>
  <DialogActions>
    <Button onClick={closeModal} color="primary" variant="contained">Salvar
preferências</Button>
  </DialogActions>
</Dialog>
```

Isto demonstra a UI do modal: uma lista de switches para cada categoria configurável. Note a menção que cookies necessários sempre ativos – podemos simplesmente omitir eles da lista ou exibi-los como desabilitados. Ex.: "Cookies Necessários (sempre ativos)" sem switch.

Hooks utilitários: `useConsent()` poderia simplificar acesso, retornando algo como `{ consentido, preferencias, acceptAll, declineAll, openPreferences }` ao componente que precisar. Assim, se o desenvolvedor quiser saber no código se certo tipo de cookie foi consentido (por exemplo, para condicionalmente renderizar um `<GoogleAnalyticsScript />`), ele pode:

```
const { preferencias } = useConsent();
useEffect(() => {
  if(preferencias.analytics) loadGoogleAnalytics();
}, [preferencias.analytics]);
```

No MVP, também podemos fornecer um util `loadScript(src, id)` (em `scriptLoader.ts`) que injeta no DOM caso ainda não exista um `<script id="id">` presente. Esse util pode ser usado dentro de efeitos como acima ou internamente no contexto quando estado muda para true. Porém, é mais seguro deixar o desenvolvedor chamar explicitamente, para garantir que não se injete acidentalmente script antes do tempo.

Imutabilidade e Modularidade: Manter constantes para nomes de cookie, categorias, etc. por exemplo:

```
export const COOKIE_NAME = 'cookieConsent';
export const CATEGORIES = <const>['analytics','marketing','functional'];
type Category = typeof CATEGORIES[number];  // 'analytics' | 'marketing' |
'functional'
```

Isso aproveita o TS para garantir que só categorias válidas sejam usadas, e com `<const>` tornamos `CATEGORIES` readonly [22]. Designar que `ConsentPreferences` tem uma chave por categoria (pode derivar de generic ou mapear manualmente a interface).

Extensibilidade: A arquitetura separa contextos e componentes. Se no futuro quisermos adicionar, por exemplo, um componente `<ConsentManager>` que englobe banner+modal juntos, podemos facilmente compô-los. Também facilita testes unitários – o reducer pode ser testado isoladamente, assim como utils.

Em resumo, a arquitetura visa baixo acoplamento: os componentes de UI usam as funções do contexto, mas não se preocupam *como* o consentimento é armazenado (poderia mudar de cookie para localStorage, por exemplo, sem mudar o componente). O contexto é independente de MUI, só lida com dados; os componentes são independentes de cookie API, só lidos do contexto. Essa separação respeita princípios de reuso.

4. Bibliotecas Open-Source Reutilizáveis (com Links e Justificativas)

Para agilizar o desenvolvimento e garantir confiabilidade, o MVP pode se apoiar em algumas bibliotecas consagradas:

- **js-cookie** – Biblioteca leve (~800 bytes gz) para manipulação de cookies em JavaScript [10]. Permite criar, ler e apagar cookies facilmente com API simples. **Justificativa:** É amplamente utilizada (milhões de downloads semanais) e testada, evitando pitfalls de lidar com `document.cookie` manualmente (parsing de string, encoding). Por exemplo, `Cookies.set('consent', 'true', { expires: 365 })` grava um cookie de forma concisa [14]. Não possui dependências e suporta todos os navegadores modernos e legados, seguindo a RFC 6265 para cookies [10]. No nosso contexto, js-cookie pode ser usado dentro do contexto para persistir as preferências do usuário localmente.

- **React Cookie (react-cookie)** – Biblioteca que provê hooks React e contexto para cookies [11], construída sobre `universal-cookie`. **Justificativa:** Facilita integração com SSR, pois permite hidratar cookies no servidor e disponibilizar no cliente via `<CookiesProvider>`. Com ela podemos escrever `const [cookies, setCookie] = useCookies(['consent'])` e ter

reatividade no estado do cookie [15]. Embora nosso MVP implemente seu próprio contexto, o react-cookie poderia ser usado internamente ou pelo menos servir de inspiração. Ele também cuida de detalhes de performance (evitar re-renders desnecessários usando contexto de baixo nível). Se preferirmos não adicionar dependência direta, ainda podemos usar o **universal-cookie** (core do react-cookie) para SSR – criando instâncias que leem os cookies do `req.headers`.

- **React Cookie Consent** – (`react-cookie-consent` por Mastermindzh) Biblioteca open-source que exibe um banner simples de consentimento em React. **Justificativa:** Embora não a usemos diretamente (já que estamos construindo algo mais personalizado), ela serve de referência de boas práticas. Por exemplo, essa lib define um componente customizável e expõe utilidades como `getCookieConsentValue()` [23] e `resetCookieConsentValue()` [24]. Sabemos através do README que ela é *"uma barra de consentimento de cookies pequena, simples e customizável para aplicações React"* [25]. Ela também integra com js-cookie (reexporta a função Cookies do js-cookie) [26], confirmando que nossa escolha de js-cookie é alinhada ao padrão de mercado. Podemos nos inspirar em como essa biblioteca estiliza o banner e lida com opções (por exemplo, ela suporta um prop `debug` para sempre mostrar o banner durante desenvolvimento, o que poderíamos replicar [27]). Licença MIT e uso difundido (>600 stars no GitHub), então é confiável [25]. *Referência:* Mastermindzh/react-cookie-consent no GitHub.

- **cookies-next** – Uma alternativa focada em Next.js, que oferece utilidades como `hasCookie` e `setCookie` que funcionam tanto no client quanto no server (usando context do Next) [28] [29]. **Justificativa:** Caso precisemos acessar cookies no server side sem usar uma solução global, esta lib facilita (especialmente em rotas App Router do Next 13). No MVP, podemos não precisar se formos pelo caminho react-cookie ou leitura manual, mas é bom conhecer.

- **Material-UI (MUI)** – Biblioteca de componentes de UI React utilizada no projeto do usuário (já vemos no package.json várias dependências MUI). **Justificativa:** Além de já ser padrão no projeto (evitando adicionar outra lib de UI), o MUI oferece componentes acessíveis e responsivos prontos, como diálogos, botões, switches etc. Isso nos poupa de estilizar tudo do zero e garante consistência visual. Usaremos MUI intensamente conforme descrito. A biblioteca é mantida e madura, com licença MIT.

- **Testing Libraries:** (opcionais) Poderíamos mencionar **Testing Library (React)** e/ou **Jest** para escrever testes do comportamento do contexto e componentes. Dado que já estão configurados no projeto (vimos `@testing-library/react` e `jest` nas devDeps [30] [31]), podemos reutilizar. Escrever testes para garantir que, por ex, ao chamar acceptAll o cookie de consentimento é criado, ou que o banner desaparece após consentimento.

Todas as bibliotecas acima são open-source e amplamente utilizadas, o que reduz riscos de bugs e problemas de segurança. Além disso, seu uso acelera o desenvolvimento do MVP, permitindo foco na lógica específica de consentimento conforme LGPD. Por exemplo, em vez de implementar parsing de cookie ou mecanismos de contexto do zero, usamos **React-Cookie** que *"integra-se perfeitamente ao ecossistema React, oferecendo gerenciamento de estado e reatividade melhores"* [15] em comparação a soluções puramente manuais.

(Referências: js-cookie – NPM; react-cookie – NPM; react-cookie-consent – GitHub [25]; cookies-next – NPM; MUI – documentação oficial.)

5. Esboços Visuais (Diagramas)

Para ilustrar o funcionamento interno da biblioteca, apresentamos dois diagramas simples.

5.1 Comunicação entre Componentes e Contexto

Legenda: Diagrama ilustrando a comunicação entre os componentes do sistema de consentimento e o contexto global. O `ConsentContext` *armazena o estado global (preferências e status de consentimento) e expõe métodos para alteração. O componente Banner consome o contexto para ler/exibir a mensagem e chama métodos (como* `acceptAll`*) quando o usuário interage, atualizando o estado no contexto. O componente Modal de Preferências também consome o contexto: lê as preferências atuais para marcar os toggles e, ao alterar alguma preferência, invoca ações do contexto (ex.:* `toggleCategory`*). Ambos os componentes estão dentro do* `<ConsentProvider>` *para ter acesso ao contexto.*

Acima, vemos que o **ConsentContext** é o ponto central: quando o Banner ou o Modal disparam ações (setar tudo, ou mudar categoria), o contexto atualiza seu estado e salva no cookie. Como os componentes estão inscritos (via hook useContext), eles reagem às mudanças – por exemplo, se "aceitar todos" for clicado no Banner, o estado `preferencias` vira true para todas categorias e `consentido=true`, então o Banner detecta que já não precisa mais ser mostrado (pode se ocultar), e possivelmente o Modal nem chega a abrir nesse caso. Por outro lado, se o usuário abre o Modal e salva preferências customizadas, o Banner igualmente pode sumir pois já houve interação. Essa comunicação fluída é gerenciada pelo React Context API.

5.2 Fluxo de Carregamento Condicional de Scripts

Legenda: Diagrama mostrando como o sistema lida com carregamento de scripts de acordo com categorias consentidas. O ConsentContext armazena flags de cada categoria (por exemplo, `analyticsConsent` *e* `adsConsent`*). Se uma categoria estiver com consentimento true, o sistema carrega o script externo correspondente; caso esteja false, o script não é carregado.*

No diagrama, exemplificamos duas categorias comuns: - **Analytics**: se o usuário consentiu cookies de Analytics, o script do Google Analytics (ou outra ferramenta de análise) é injetado na página. Caso contrário, permanece bloqueado. - **Marketing/Ads**: se consentido, carrega-se o script de anúncios/marketing (por exemplo, pixel de redes sociais ou ad trackers). Se não consentido, nada é feito e esses serviços ficam desativados.

Implementacionalmente, isso pode ocorrer via um efeito no contexto: quando `preferencias.analytics` muda para true, chamamos uma função `loadScript('ga.js')`. Alternativamente, o integrador do lib pode usar um hook como no exemplo do código (seção 6) para carregar condicionalmente. O importante é que **por padrão o MVP não insere nenhum script de terceiro**, ele apenas fornece os meios para fazê-lo no momento certo. Assim garantimos conformidade – nenhum cookie de terceiros de tracking é criado até haver consentimento explícito [1] .

6. Exemplos de Código TypeScript dos Blocos Principais

A seguir, trechos de código TypeScript ilustrando partes chave da implementação, seguindo boas práticas de codificação do projeto:

6.1 Contexto de Consentimento (ConsentContext.tsx)

```
import { createContext, useContext, useReducer, useEffect, ReactNode } from
'react';
import Cookies from 'js-cookie';

// Tipos de estado e ações
interface ConsentPreferences {
  analytics: boolean;
  marketing: boolean;
}
interface ConsentState {
  consentido: boolean;
  prefs: ConsentPreferences;
}
type ConsentAction =
  | { type: 'ACEITAR_TUDO' }
  | { type: 'RECUSAR_TUDO' }
  | { type: 'SET_CATEGORIA'; categoria: keyof ConsentPreferences; valor:
boolean }
  | { type: 'CARREGAR_INICIAL'; state: ConsentState };

// Estado inicial padrão (tudo false exceto consentido)
const DEFAULT_PREFS: ConsentPreferences = { analytics: false, marketing:
false };
const initialState: ConsentState = { consentido: false, prefs:
DEFAULT_PREFS };

// Contextos
const ConsentStateContext = createContext<ConsentState |
undefined>(undefined);
const ConsentDispatchContext = createContext<React.Dispatch<ConsentAction> |
undefined>(undefined);

// Reducer para manipular estado conforme ação
function consentReducer(state: ConsentState, action: ConsentAction):
ConsentState {
  switch(action.type) {
    case 'ACEITAR_TUDO':
      return { consentido: true, prefs: Object.fromEntries(
                Object.keys(state.prefs).map(cat => [cat, true])
              ) as ConsentPreferences };
    case 'RECUSAR_TUDO':
      return { consentido: true, prefs: Object.fromEntries(
                Object.keys(state.prefs).map(cat => [cat, false])
              ) as ConsentPreferences };
    case 'SET_CATEGORIA':
      return {
        consentido: state.consentido,
        prefs: { ...state.prefs, [action.categoria]: action.valor }
      };
```

```
    case 'CARREGAR_INICIAL':
      return action.state;
    default:
      return state;
  }
}

// Provider do contexto
export function ConsentProvider({ children }: { children: ReactNode }) {
  // Carrega preferências salvas do cookie, se existir
  const saved = Cookies.get('cookieConsent');
  const parsed = saved ? JSON.parse(saved) : null;
  const [state, dispatch] = useReducer(consentReducer, parsed ? parsed :
initialState);

  // Atualiza cookie a cada mudança de estado (exceto carregamento inicial)
  useEffect(() => {
    // Não salvar até que o usuário tenha consentido ou recusado
explicitamente
    if(state.consentido) {
      Cookies.set('cookieConsent', JSON.stringify(state), { expires: 365 });
    }
  }, [state]);

  return (
    <ConsentStateContext.Provider value={state}>
      <ConsentDispatchContext.Provider value={dispatch}>
        {children}
      </ConsentDispatchContext.Provider>
    </ConsentStateContext.Provider>
  );
}

// Hooks para consumir contexto
export function useConsentState() {
  const context = useContext(ConsentStateContext);
  if(context === undefined) {
    throw new Error('useConsentState must be used within ConsentProvider');
  }
  return context;
}
export function useConsentDispatch() {
  const context = useContext(ConsentDispatchContext);
  if(context === undefined) {
    throw new Error('useConsentDispatch must be used within
ConsentProvider');
  }
  return context;
}
```

Explicação: O `ConsentProvider` acima inicializa o estado a partir de um cookie existente (se disponível). Usamos `Cookies.get` de **js-cookie** para ler; o cookie armazena um JSON string do nosso estado. Em seguida, definimos um reducer simples: - `'ACEITAR_TUDO'` e `'RECUSAR_TUDO'` percorrem as chaves de `state.prefs` e setam todas para true ou false (mantendo `consentido: true` para marcar que usuário já escolheu). - `'SET_CATEGORIA'` ajusta uma categoria específica (usado no toggle do modal). - `'CARREGAR_INICIAL'` poderia ser acionado se quisermos carregar estado inicial de forma assíncrona ou via props (no nosso caso, já fizemos parse antes do reducer, então poderíamos nem precisar dessa action, mas deixamos para clareza).

O `useEffect` grava o cookie toda vez que `state` muda e `state.consentido` for true. Assim evitamos gravar enquanto o usuário não interagiu (embora poderíamos gravar as preferências parciais também, optamos por gravar quando finalize). **Obs:** Poderíamos gravar mesmo antes de consentido=true para salvar preferências parciais em tempo real; mas para MVP, considerar gravar apenas após clique "Salvar" ou Aceitar/Recusar.

Os hooks `useConsentState` e `useConsentDispatch` permitem acessar estado e despachar ações facilmente nos componentes. Isso segue o mesmo padrão do React Context oficial (lançando erro se usado fora do provider).

6.2 Componente Banner (CookieBanner.tsx)

```
import { useConsentState, useConsentDispatch } from '../context/
ConsentContext';
import { Snackbar, Paper, Typography, Button, Stack } from '@mui/material';

export function CookieBanner() {
  const { consentido } = useConsentState();
  const dispatch = useConsentDispatch();

  // Só exibe banner se usuário ainda não consentiu/recusou
  if(consentido) return null;

  const aceitarTudo = () => dispatch({ type: 'ACEITAR_TUDO' });
  const recusarTudo = () => dispatch({ type: 'RECUSAR_TUDO' });
  const abrirPreferencias = () => dispatch({ type: 'ACEITAR_TUDO' }); // Em
MVP podemos tratar abrir modal como aceitar tudo por enquanto ou ajustar
lógica

  return (
    <Snackbar open={!consentido} anchorOrigin={{ vertical: 'bottom',
horizontal: 'center' }}>
      <Paper elevation={3} sx={{ p: 2, maxWidth: 600, mx: 'auto' }}>
        <Typography variant="body1">
          Utilizamos cookies para melhorar sua experiência em nosso site.
Você pode aceitar todos ou personalizar suas preferências.
        </Typography>
        <Stack direction="row" spacing={2} justifyContent="flex-end" mt={1}>
          <Button onClick={recusarTudo} variant="outlined">Recusar</Button>
          <Button onClick={aceitarTudo} variant="contained">Aceitar todos</
Button>
```

```
          <Button onClick={abrirPreferencias} variant="text">Preferências</
Button>
        </Stack>
      </Paper>
    </Snackbar>
  );
}
```

Explicação: Este banner verifica `consentido` do contexto: se já for true, retorna null (não exibe nada). Caso contrário, renderiza um Snackbar posicionado no bottom-center contendo um Paper com texto e botões. Usamos MUI `<Stack>` para alinhar os botões à direita com espaçamento.

Os botões disparam dispatch de ações do contexto: - `Recusar`: dispatch `{ type: 'RECUSAR_TUDO' }` – isso marcará consentido=true e prefs all false. - `Aceitar`: dispatch `{ type: 'ACEITAR_TUDO' }` – marcará all true. - `Preferências`: aqui no MVP chamamos `ACEITAR_TUDO` só para fechar banner e marcar consentido (pois abrir modal após interagir já implica consentimento inicial). **Mas isso seria alterado** para em vez disso dispatch uma ação para abrir modal (ex: `OPEN_MODAL`). No código acima está simplificado assumindo MVP sem modal completo.

Após qualquer ação, como `consentido` vira true, o componente irá se desmontar (Snackbar some).

(Nota: Em versão final, teríamos `dispatch({ type: 'ABRIR_MODAL' })` e o contexto controlaria `modalAberto`. Para simplificar, omitimos essa parte no snippet.)

6.3 Carregamento Condicional de Script (util/scriptLoader.ts e uso)

`scriptLoader.ts`:

```
export function loadScript(id: string, src: string, attrs:
Record<string,string> = {}) {
  if(document.getElementById(id)) return; // já carregado
  const script = document.createElement('script');
  script.id = id;
  script.src = src;
  script.async = true;
  // define quaisquer atributos adicionais
  for(const [key, value] of Object.entries(attrs)) {
    script.setAttribute(key, value);
  }
  document.body.appendChild(script);
}
```

Uso no contexto ou em componentes:

```
import { useConsentState } from '../context/ConsentContext';
import { useEffect } from 'react';
import { loadScript } from '../utils/scriptLoader';
```

```
export function AnalyticsLoader() {
  const { prefs } = useConsentState();

  useEffect(() => {
    if(prefs.analytics) {
      loadScript('ga-script', 'https://www.googletagmanager.com/gtag/js?
id=UA-XXXXX-Y');
      // Exemplo: depois de carregar gtag, inicializar se necessário
      window.dataLayer = window.dataLayer || [];
      function gtag(){ dataLayer.push(arguments); }
      gtag('js', new Date());
      gtag('config', 'UA-XXXXX-Y');
    }
  }, [prefs.analytics]);

  return null;
}
```

No exemplo acima, `AnalyticsLoader` é um componente que poderia ser usado pelo app integrador para carregar o GA quando `prefs.analytics` for true. Ele utiliza nosso `useConsentState` para saber das prefs. Quando a dependência muda para true, ele injeta o script do GA (evitando duplicar por id) e configura o tracker.

No MVP, poderíamos incorporar algo assim diretamente no contexto (ex: usar `useEffect` dentro do `ConsentProvider` para monitorar mudanças e carregar scripts automaticamente). No entanto, deixar isso para o integrador pode dar mais controle. Uma abordagem intermediária: expor componentes utilitários, como um `<ConsentGate category="analytics">` que renderiza children somente se consentido. Mas para MVP, mantemos simples.

6.4 Uso da Biblioteca no Next.js (exemplo de integração)

No arquivo `_app.tsx` do projeto Next.js do usuário:

```
import { ConsentProvider, CookieBanner } from 'cookie-consent-lib';
import type { AppProps } from 'next/app';

function MyApp({ Component, pageProps, cookieConsentState }: AppProps & {
cookieConsentState: any }) {
  return (
    <ConsentProvider initialState={cookieConsentState}>
      {/* ... resto do layout global ... */}
      <Component {...pageProps} />
      <CookieBanner />
    </ConsentProvider>
  );
}

// Exemplo de hidratação SSR (opcional):
MyApp.getInitialProps = async ({ ctx }) => {
```

```
  const cookies = ctx.req?.headers.cookie;
  let cookieConsentState = null;
  if(cookies) {
    const match = cookies.split(';').find(c =>
c.trim().startsWith('cookieConsent='));
    if(match) {
      try { cookieConsentState =
JSON.parse(decodeURIComponent(match.split('=')[1])); } catch {}
    }
  }
  return { cookieConsentState };
};

export default MyApp;
```

Explicação: Envolvemos o aplicativo com `ConsentProvider` e inserimos `CookieBanner` no layout global para aparecer em todas páginas (apenas quando necessário). Passamos um `initialState` (suponto que adaptamos o Provider para aceitar via props) obtido no server em getInitialProps – aqui fazemos parse manual do cookie. Isso evitará que, se o usuário já deu consentimento prévio, o banner pisque no SSR. Se não usar getInitialProps, o banner vai sumir apenas após hidratação (pequeno flash), o que pode ser aceitável também.

O importante é mostrar que a integração é direta: uma vez instalada a biblioteca, o desenvolvedor apenas coloca o Provider e o Banner. O Modal de preferências nem precisa ser colocado manualmente se o Provider já gerenciá-lo internamente (podemos decidir renderizar `<PreferencesModal />` dentro do Provider quando `state.modalAberto` for true, semelhante ao que foi feito com `<AlertDialog />` no código do usuário [13] [32]). Essa simplicidade de integração é fundamental.

7. Internacionalização (i18n)

Embora a primeira versão seja em português (pt-BR) fixo, a arquitetura deve facilitar a internacionalização futura: - **Separação de Texto vs Lógica:** Todos os rótulos, textos informativos e mensagens exibidas ao usuário devem ficar fora do código lógico. Em vez de strings embutidas, usar props ou um objeto de tradução. Exemplo: o componente `CookieBanner` pode receber via props as strings `message`, `acceptLabel`, `declineLabel`, `preferencesLabel`. Para o MVP, podemos definir padrões em português, mas ainda permitir override. Assim, em um futuro update, poderíamos fornecer versões pré-traduzidas ou integrar com i18n do Next. - **Contexto de idioma ou uso de library:** Uma opção futura é integrar **react-i18next** ou similar, mas não é necessário se apenas expomos props customizáveis. Como boa prática, podemos armazenar as strings padrão num objeto constante, e talvez criar um tipo para as chaves de texto:

```
interface ConsentTexts {
  bannerMessage: string;
  acceptAll: string;
  declineAll: string;
  preferences: string;
  modalTitle: string;
  // ... etc
}
```

```
const DEFAULT_TEXTS: ConsentTexts = {
  bannerMessage: "Utilizamos cookies ...",
  acceptAll: "Aceitar todos",
  // ...
};
```

E no Provider permitir `<ConsentProvider texts={customTexts}>` para mesclar com DEFAULT_TEXTS. - **Pluralização e Formatação:** Provavelmente desnecessário aqui (textos são simples). Mas se houvesse necessidade, mantendo em mente usar bibliotecas ou API Intl do JS. - **Idiomas da LGPD:** Português será o foco, mas eventualmente pode-se querer inglês para usuários internacionais. O design preparado com props torna isso trivial para o integrador (basta passar traduções via props ou contexto). - **Evitar hardcode no código:** Já no MVP, mesmo que não exponhamos tudo via API pública, podemos internamente definir as strings em um só lugar, para facilitar manutenção e futura extração de tradução.

Em resumo, a lib será "locale-ready": pronta para tradução sem grandes mudanças de arquitetura, atendendo a demanda multi-idiomas quando surgir.

8. Personalização e Extensibilidade para Projetos Diferentes

Um dos objetivos principais é tornar a biblioteca **altamente personalizável** para se adequar a diferentes aplicações React/Next. As estratégias incluem:

- **Theming (Tema Visual):** Como usamos Material-UI, aproveitamos o sistema de tema. O componente do banner e modal devem suportar o tema global do aplicativo. Por exemplo, se o projeto do usuário já define um tema MUI com certas cores primárias/secundárias, nossos botões automaticamente usarão essas cores ao usar `color="primary"`. Para personalizações adicionais, permitir props de estilo:
- Prop para classes ou sx: ex: `<CookieBanner containerSx={{ backgroundColor: '#123' }} />` ou `<CookieBanner PaperProps={{ sx: { ... }}} />`.
- Suporte a modo escuro/claro – deixar isso a cargo do tema do aplicativo ou detectar preferências do sistema via MUI.

- Responsividade custom: garantir que nosso layout (snackbar, dialog) funcionem em mobile (talvez usar `DialogProps={{ fullScreen: isMobile }}` se quisermos).

- **Textos e Conteúdo:** Conforme seção i18n, todos os textos serão configuráveis via props. Além disso, permitir inserir conteúdo extra se necessário. Por exemplo, poderia haver uma prop para `<PreferencesModal info={<MyCustomInfoComponent/>}>` caso o projeto queira incluir um texto explicativo adicional ou link para política detalhada dentro do modal. Outra ideia: prop para URL da política de privacidade que o banner usará no link "Saiba mais".

- **Categorias de Cookies:** Diferentes projetos podem ter diferentes categorias (ou quantidades). No MVP talvez fixamos “analytics” e “marketing” como categorias de exemplo. Mas pensando em extensibilidade, podemos:

- Implementar o contexto de forma genérica usando um tipo de categoria genérico paramétrico (ex: `ConsentProvider<Category extends string>`). Contudo, isso complica um pouco o uso. Uma alternativa pragmática: definir um conjunto padrão mas permitir **desabilitar** ou renomear categorias via props. Ex:

`<ConsentProvider categories={{ analytics: true, marketing: false, funcional: false }}>` onde o valor boolean indica padrão inicial (ou ativo/inativo). Ou passar um array de objetos: `categories=[{ id:'analytics', label:'Análise', default: false }, ...]`.

- Para MVP, documentar que atualmente as categorias X e Y existem, e futuras serão adicionadas. Mas já projetar o código para não ser difícil incluir mais. Por exemplo, ao invés de ter lógica estática só para 'analytics' e 'marketing', iterar sobre `Object.keys(state.prefs)` quando for renderizar toggles ou salvar cookie. Isso já nos desacopla do número específico de categorias.

- **Readonly & Type Safety:** Se usarmos uma lista constante de categorias, como mencionado, o TS nos ajuda a garantir consistência. E marcando-a `readonly` evitamos modificações. Por exemplo:

```
const CATEGORIES = ['analytics', 'marketing'] as const;
type Category = typeof CATEGORIES[number];
```

  A interface `ConsentPreferences` então poderia ser gerada a partir disso, mas TS não faz facilmente objeto a partir de union. Podemos manualmente definir e manter sincronizado com CATEGORIES.

- **APIs de Extensão:** Pensando adiante, podemos expor ganchos para extensões. Talvez permitir ao desenvolvedor passar callbacks: `onConsentGiven` (chamado quando usuário aceita), `onPreferencesSaved(preferences)` (chamado quando salva específicas). Isso permite integrar com outras partes do app (ex.: notificar um analytics interno). No MVP não é obrigatório, mas já podemos ter no radar e talvez incluir stub desses callbacks (que por padrão não fazem nada).

- **Composição:** Como a lib está em React, usuários podem optar por não usar nosso Banner e criar o próprio usando nosso contexto. Isso é ótimo – então garantir que o contexto e hooks sejam bem documentados para quem quiser UI custom. Nossa arquitetura suporta isso, pois `ConsentProvider` e hooks são independentes dos componentes. Assim, temos reuso máximo: use a UI pronta *ou* só use o contexto e crie sua UI.

- **Imutabilidade para segurança:** Ao usar `Readonly` em props e estruturas, garantimos que dentro da biblioteca não vamos sem querer modificar objetos do usuário. Por exemplo, se passarem um objeto de textos, podemos fazer `const texts = useMemo(()=> ({ ...DEFAULT_TEXTS, ...props.texts }), [props.texts])` para mesclar, sem alterar o original. Isso evita efeitos colaterais. Também torna claro para usuários que aquelas props não serão alteradas internamente.

- **Evitar dependências desnecessárias:** Por fim, para ser fácil de integrar em qualquer projeto, manter a biblioteca leve. Não incluir grandes libs a não ser as já esperadas (React, MUI). js-cookie pesa ~2KB, react-cookie ~4KB, o que é aceitável. Focar em código próprio simples. Isso garante que projetos diferentes não tenham impacto de performance.

Em suma, a biblioteca será pensada como um **kit flexível**: dá para usar do jeito básico (drop-in com padrão) ou adaptar profundamente (passando props e compondo com seus próprios componentes). Essa extensibilidade vai ao encontro de cenários reais, onde cada aplicação pode ter um requisito

ligeiramente diferente de texto, estilo ou fluxo, mas a base de lógica de consentimento permanece igual.

9. Armazenamento Local e Ausência de Backend

Reforçando um ponto do checklist: nesta implementação inicial, **não haverá persistência de consentimento em servidor**. Todas as informações permanecem no navegador do usuário, principalmente via cookies. Consequências e detalhes:

- **Cookie como Fonte de Verdade:** O cookie de consentimento (ex: nome `cookieConsent`) é o armazenamento primário. Ele contém um JSON com as preferências do usuário. Exemplo de valor: `{"consentido": true, "prefs": {"analytics": false, "marketing": true}}`. Ao ler esse cookie, o app sabe o que pode ou não carregar. Como cookies são enviados ao servidor em cada requisição por padrão, *tecnicamente* o backend teria acesso a essa info, mas não pretendemos usá-la no MVP. E podemos marcar o cookie com `SameSite=Lax` para que seja enviado só para nosso domínio mesmo.

- **Sem Logs de Consentimento:** Não manteremos registro de timestamp ou user id atrelado ao consentimento, já que não há backend. A LGPD não obriga um envio ao servidor, apenas que possamos comprovar consentimento se requerido – mas isso pode ser atendido pelo próprio cookie no navegador do usuário e pelas configurações do sistema. Para uma futura necessidade de auditoria, ver roadmap.

- **No backend calls:** As funções da biblioteca não fazem fetch/XHR. Isso simplifica muito, pois não precisamos considerar falha de rede, tempo de resposta ou servidores. O consentimento é imediato e local.

- **Possibilidade de Sincronização Manual:** Se algum projeto quiser registrar no backend, pode usar os callbacks citados ou escutar eventos do contexto para então ele mesmo enviar um fetch. Mas isso fora do escopo MVP.

- **Reset de Consentimento:** Sem backend, a única forma de "esquecer" um consentimento é o usuário limpar os cookies ou alguma funcionalidade nossa de reset. Poderíamos oferecer `resetConsent()` no contexto que simplesmente apaga o cookie (`Cookies.remove(name)`) e reseta estado local, causando o banner aparecer de novo. Isso pode ser útil em desenvolvimento (prop `debug` semelhante à react-cookie-consent, que ignora cookie e sempre mostra banner para testes [27]).

Essa abordagem local-first é alinhada com performance e privacidade: os dados de consentimento não saem do dispositivo do usuário, evitando exposição indevida. Apenas certifique-se de informar na política de privacidade que as preferências ficam armazenadas em cookie no navegador do usuário.

10. UI com Material-UI: Responsividade e Acessibilidade

Todos os elementos de interface serão construídos com componentes do **MUI v5**, o que traz várias vantagens: - **Acessibilidade Nativa:** MUI já implementa ARIA roles e teclado para componentes complexos como Dialog, Snackbar, Switch, Button. Por exemplo, o `<Dialog>` gerencia foco automaticamente, `<Switch>` vem com role checkbox e pode ser lido por leitor de tela indicando on/

off. Isso não nos exime de conferir, mas adianta bastante. Também há suporte a <kbd>Tab</kbd> e <kbd>Esc</kbd> out-of-the-box no Dialog. - **Responsividade:** MUI usa o sistema de breakpoints. Podemos facilmente tornar o modal de preferências em tela cheia no mobile adicionando prop `fullScreen` quando `useMediaQuery(theme.breakpoints.down('sm'))` é true, por exemplo. O banner em si, por ser basicamente um flex container, já se ajusta – mas devemos testar se em telas pequenas o texto não extrapola. Podemos usar `<Typography variant="body2">` no mobile vs `body1` no desktop, ou simplesmente permitir quebra de linha. O stack de botões rola automaticamente se exceder largura, ou podemos fazer os botões se empilharem verticalmente em xs telas (usando `<Stack direction={{ xs: 'column', sm: 'row' }}>`). - **Estilização Consistente:** Seguindo o design system do usuário, utilizaremos o tema global. No código de exemplo usamos `color="primary"` no botão "Aceitar todos". Isso aplicará a cor primária definida no tema (que geralmente é verde ou azul conforme branding). Caso precisemos personalizar, MUI permite overrides via Theme ou passando `sx` prop. No MVP, manteremos o estilo simples, mas limpo (Paper com elevação leve para destaque, etc.). - **Iconografia e Feedback:** Poderíamos incluir ícones (por exemplo, um ícone de informações ao lado de "Preferências" ou um ícone de check no botão aceitar). Como o projeto já inclui `@mui/icons-material`, temos acesso fácil a ícones se desejado, sem dependência extra. - **Validação Visual:** Graças ao MUI, alcançamos boa UI sem muito CSS custom. Importante é checar o contraste: por ex, se o Paper for branco e os botões padrão, provavelmente ok. Se o site estiver em dark mode, nosso banner deve se adaptar (o Paper vai ficar no default do tema escuro). Testaremos ambos.

Acessibilidade Extra: Além do que o MUI fornece, cuidaremos de detalhes: - Fornecer um texto descritivo no banner sobre a finalidade ("melhorar experiência..."). Manter esse texto conciso para usuários de leitor de tela não se perderem, mas suficiente para contexto. - `aria-labelledby` e `aria-describedby` no Dialog para amarrar o título e a descrição. O DialogTitle tem `id`, usamos no Dialog via prop. - No foco inicial: ao abrir o modal, garantir que o foco esteja no primeiro toggle ou botão dentro dele, para navegação suave. MUI por padrão foca no DialogTitle, podemos ajustar se necessário. - **Teclas de atalho:** Não implementaremos no MVP, mas podemos documentar que <kbd>Esc</kbd> fecha o modal (nativo do Dialog), e o usuário pode navegar com Tab nos toggles. Se quisermos ser bem completos, poderíamos permitir fechar banner com <kbd>Esc</kbd> também (usando Snackbar's ability or by adding keydown listener).

Finalmente, testaremos o componente com ferramentas como Lighthouse ou axe-core para verificar se há atributos ARIA faltando. Ex: a Stack com botões deve ter `aria-label`? Provavelmente não, pois cada botão já tem texto visível.

Em conclusão, usando MUI atendemos aos requisitos de UI responsiva e acessível "by construction". Claro que, atendendo às **diretrizes da ANPD**, precisamos assegurar texto claro e opções destacadas para o usuário [3] [4] – o que foi feito dando botões bem identificados e não escondendo a recusa. O resultado será uma interface amigável que inspira confiança de que o site respeita a privacidade do usuário, cumprindo LGPD em letra e espírito.

Referências: A implementação acessível e responsiva de cookie banners tem sido recomendada em diversas fontes. Por exemplo, a própria orientação brasileira reforça que deve-se permitir **recusa sem prejuízo** e apresentar informações de forma **clara e destacada** [3] [2]. Nossa UI cumpre isso, colocando todas as opções lado a lado, sem texto confuso. Além disso, cookies não essenciais só são usados se consentidos [1], garantindo conformidade. Em suma, o MVP proposto cobre os requisitos técnicos e legais iniciais, provendo uma base reutilizável que pode evoluir conforme as necessidades de projetos React/Next e regulações de privacidade futuras.

---

1 Brazil's Guidance on Cookies - Securiti.ai
https://securiti.ai/blog/brazil-anpd-guidance-on-cookies/

2 3 4 5 7 8 9 16 17 18 19 20 LGPD & COOKIES - ANPD lança “Guia orientativo: Cookies e Proteção de Dados Pessoais”
https://goadopt.io/blog/anpd-cookies-guia-orientativo-cookies-protecao-dados-pessoais/

6 15 Comparing React-Cookie with other cookie management libraries - Managing Cookies in React Applications with React-Cookie | StudyRaid
https://app.studyraid.com/en/read/11468/359568/comparing-react-cookie-with-other-cookie-management-libraries

10 11 14 5 Top JavaScript Cookie Libraries | by Nipuni Arunodi | Bits and Pieces
https://blog.bitsrc.io/5-top-javascript-cookie-libraries-329ae3150cfb

12 13 22 32 AlertDialogContext.tsx
file://file-4prhmXTMhznTkbRorocid5

21 A React cookie consent using hooks and context · GitHub
https://gist.github.com/daankauwenberg/bf0daf4d4a9a157a078ba4ec4559e3ab

23 24 25 26 27 GitHub - Mastermindzh/react-cookie-consent: A small, simple and customizable cookie consent bar for use in React applications.
https://github.com/Mastermindzh/react-cookie-consent

28 29 Cookie consent in NextJs. This article discusses how to add... | by Afzal Imdad | Medium
https://afzalimdad9.medium.com/cookie-consent-in-nextjs-4ba144c5cc60

30 31 package.json
file://file-2J1V6cBS6kywUHDyZGs9TZ