ChatGPT

Plano de Implementação da Plataforma Leitícia

1. Estrutura Técnica da Aplicação

Separação entre Frontend e Backend

Para um MVP com equipe reduzida, é recomendado separar claramente as responsabilidades de *frontend* (interface e experiência do usuário) e *backend* (regras de negócio, banco de dados e APIs). Essa separação facilita a manutenção e permite evoluir cada parte independentemente [1]. Por exemplo, o frontend pode ser composto por páginas HTML estáticas e scripts JavaScript que consumam dados através de chamadas a um backend via APIs REST; enquanto isso, o backend expõe esses serviços e gerencia o banco de dados. Essa divisão traz vantagens como menor carga no servidor e possibilidade de escalar ou alterar tecnologias de front e back separadamente [1]. Em contrapartida, requer um cuidado extra com desempenho em dispositivos limitados, já que mais processamento ocorre no cliente (navegador) [2].

No contexto da Plataforma Leitícia, podemos adotar um frontend leve, responsável por apresentar informações (páginas de cadastro, listagem de bancos de leite, formulários de solicitação de doação, etc.), e um backend que fornece APIs para essas funcionalidades (por exemplo, endpoints para criar novas doações, listar bancos de leite, autenticar usuários, etc.). Essa abordagem segue as tendências modernas de desenvolvimento web, empregando o padrão MVC ou arquiteturas similares para organizar o código em camadas de forma limpa.

Frameworks e Bibliotecas Recomendados

Visando uma solução **leve e escalável**, priorize tecnologias simples e bem suportadas. Dado que a equipe considerou "HTML, CSS e JS puros + PHP ou Node.js", pode-se optar por uma pilha clássica LAMP (Linux, Apache, MySQL, PHP) ou algo baseado em Node.js. Para rapidez no desenvolvimento em ambiente de hospedagem compartilhada, PHP é uma escolha pragmática – especialmente porque Hostinger (ambiente previsto para publicação) oferece suporte nativo a PHP em seus planos padrão, enquanto Node.js exigiria um VPS dedicado [3]. De fato, o Node.js não é suportado em planos compartilhados comuns (exigindo acesso root em VPS) [3]. Além disso, o próprio ambiente Hostinger é otimizado para aplicações PHP, como WordPress e outros CMS [4], o que indica menor fricção ao escolher PHP para o MVP.

Frontend: Utilize **HTML5** semântico para estruturar o conteúdo e facilite a acessibilidade. Para estilização, pode-se adotar CSS puro respeitando as diretrizes de identidade visual fornecidas (paleta de cores, tipografia, etc.). Caso seja necessário agilizar a responsividade, considere um framework CSS leve como **Bootstrap** ou **Bulma** – eles fornecem grid responsivo e componentes prontos, reduzindo o trabalho manual. Contudo, se a preferência é manter "CSS puro" para controle total do design, aplique técnicas de *Flexbox* e *Grid CSS* junto com media queries para garantir boa apresentação em diferentes telas. Em termos de JavaScript, pode-se usar **JS Vanilla** para interações simples (validação de formulários, máscaras de input, requisições AJAX) a fim de evitar overhead de frameworks pesados. Bibliotecas utilitárias mínimas como **Axios** (para requisições AJAX) ou **jQuery** (caso a equipe tenha familiaridade) são opcionais, mas não obrigatórias, dependendo da complexidade das interações.

Backend: Com PHP, uma possibilidade é começar simples usando uma abordagem estruturada com inclusão de scripts (por exemplo, páginas PHP que incluem arquivos de configuração, modelos e controladores básicos). Se desejado, pode-se adotar um micro-framework PHP como **Slim** ou **Lumen** (versão simplificada do Laravel) para organizar rotas e middleware de forma mais padronizada, sem adicionar muita complexidade. Esses micro-frameworks facilitam a criação de APIs REST com pouco código adicional. Alternativamente, se a equipe preferir Node.js pela familiaridade com JavaScript no servidor, seria preciso usar um serviço de VPS ou contêiner para hospedá-lo. Nesse caso, o framework mínimo recomendado é **Express.js** devido à sua simplicidade e larga adoção na construção de APIs RESTful. Vale ressaltar que Node.js em Hostinger requer configurações avançadas (ou uso de VPS) e não funciona em host compartilhado convencional [3], o que torna PHP a escolha mais simples para o escopo atual.

Independentemente da linguagem backend, mantenha as dependências reduzidas para facilitar a manutenção do MVP. Use apenas o necessário: por exemplo, bibliotecas de segurança (como para hashing de senhas – PHP `password_hash` ou a biblioteca bcrypt no Node), e evite pacotes muito pesados. Quanto ao formato de troca de dados, adote **JSON** nas APIs para facilidade de consumo pelo frontend JavaScript.

Integração com APIs Públicas e Privadas

A Plataforma Leitícia terá um diferencial ao integrar-se com serviços externos para ampliar suas funcionalidades. Dois eixos principais foram identificados: **bancos de leite** e **logística**.

Para **bancos de leite**, é viável integrar com dados públicos existentes. O Ministério da Saúde/Fiocruz mantém a Rede Brasileira de Bancos de Leite Humano (rBLH) com informações de 222 bancos e 217 postos de coleta em todo o Brasil [5]. Uma integração poderia consistir em consumir dados geográficos desses bancos para exibir ao usuário os pontos de doação próximos. Se não houver uma API pública direta disponibilizando essas informações, pode-se utilizar serviços de geocoding/mapas para localizá-los. Por exemplo, a API Google Places/Maps pode buscar por termos como "Banco de Leite Humano" dentro de uma localidade e retornar endereços e coordenadas [6]. Essa solução permitiria listar e mostrar num mapa os bancos de leite mais próximos do usuário. Outra opção é verificar se a rBLH-Fiocruz disponibiliza um webservice ou dataset; na ausência, um dataset estático poderia ser inserido no banco de dados da aplicação para consulta offline (atualizado periodicamente de fontes oficiais).

Para **logística**, o objetivo é facilitar o recolhimento do leite doado na casa da doadora ou o envio a um banco. Aqui entram integrações privadas: uma abordagem simples é usar a API de coleta dos Correios, que permite **agendar** buscas de encomendas em endereços fornecidos [7]. Essa API de Coleta Interativa dos Correios dá autonomia para programar coletas e acompanhar status [7] – no contexto do projeto, cada doação confirmada poderia acionar um pedido de coleta por meio dessa API, automatizando a logística de transporte do leite até o banco de leite escolhido. Alternativamente, pode-se integrar com serviços de logística privados ou startups (como serviços de entrega locais via motoboy ou apps de transporte). Por exemplo, plataformas de roteirização como a **RoutEasy** ou **Pathfind** oferecem APIs para otimizar rotas e gerenciar coletas [8]. Em um primeiro momento, no entanto, é recomendável implementar algo mais simples: gerar notificações aos responsáveis pela logística (por exemplo, enviar automaticamente um e-mail ou WhatsApp para o banco de leite/local logístico quando uma nova doação for solicitada, contendo os dados necessários para retirada). Isso garante a funcionalidade básica mesmo sem depender de uma integração complexa imediatamente.

Em ambos os casos, é crucial desenhar a aplicação já prevendo pontos de integração. Por exemplo, criar serviços/backend que encapsulam as chamadas externas (um módulo para “Logística” que possa

chamar a API dos Correios ou outro provedor) e tratar respostas/erros de forma robusta. Assim, se futuramente trocar o provedor (por exemplo, usar outra transportadora), a lógica interna da plataforma Leitícia não precisa mudar, apenas o módulo de integração.

Segurança de Dados e LGPD

Segurança e privacidade devem ser consideradas desde o início, alinhadas à **LGPD (Lei Geral de Proteção de Dados)** brasileira. Em termos práticos, isso significa:

- **Minimização de dados coletados:** Coletar apenas informações pessoais estritamente necessárias para os objetivos da plataforma (princípio da necessidade da LGPD) [9]. Por exemplo, para doadoras pode ser preciso nome, contato, endereço (para logística de coleta) e talvez dados de saúde relevantes (se houver triagem), mas qualquer dado irrelevante não deve ser solicitado. Nada de dados excessivos ou sem finalidade clara.

- **Consentimento e transparência:** Informar claramente às usuárias (doadoras) e demais usuários quais dados estão sendo coletados e para qual finalidade, obtendo consentimento explícito quando necessário. Deve haver uma política de privacidade acessível explicando o tratamento dos dados. As telas de cadastro podem incluir uma caixa de consentimento concordando com os termos LGPD.

- **Proteção e segurança técnica:** Implementar medidas aptas a proteger os dados pessoais contra acesso não autorizado, vazamentos ou perda [10]. Isso inclui:

- Uso de conexão segura (HTTPS) em toda a aplicação, especialmente em páginas de login e formulários, para criptografar os dados em trânsito.
- Armazenamento seguro de senhas (sempre usando **hash criptográfico forte** com *salt*, ex: `password_hash` do PHP ou bcrypt).
- Controle de acesso no backend para que cada usuário só acesse/altere seus próprios dados; áreas administrativas protegidas por autenticação forte.
- **Prepared Statements** nas interações com o banco de dados para prevenir SQL Injection. As bibliotecas modernas (PDO ou MySQLi no PHP, por exemplo) suportam *prepared statements*, o que é fundamental para evitar ataques de injeção de SQL [11] [12].
- Validação de entradas no backend (nunca confiar somente em validação JavaScript no cliente, sempre validar no servidor tipos de dados, formatos de e-mail, CEP etc., e sanitizar dados antes de usar).

- Backup regular do banco de dados e logs de auditoria de ações importantes (como doações realizadas, edições de cadastro), para poder detectar e responder a incidentes.

- **Anonimização/Eliminação:** Prever procedimentos para atender direitos dos titulares, como exclusão dos dados pessoais mediante solicitação. Por exemplo, se uma usuária solicitar a remoção de sua conta, o sistema deve permitir apagar ou anonimizar seus dados (mantendo apenas o histórico necessário de doações de forma agregada, se for o caso).

- **Segregação de ambientes e chaves de API:** Manter seguras as chaves de API de integrações externas (por exemplo, chave da API Google Maps ou credenciais da API dos Correios). Idealmente, tais credenciais ficam em arquivos de configuração no servidor, não expostas no front-end. No código versionado (GitHub), nunca incluir dados sensíveis diretamente – usar variáveis de ambiente ou arquivos de configuração que são omitidos do versionamento (`.gitignore`). Isso evita vazamento de segredos da aplicação.

Ao seguir esses princípios, a plataforma estará aderindo às boas práticas da LGPD, garantindo finalidade, necessidade, segurança e prevenção contra uso indevido dos dados pessoais coletados [13] [14].

Responsividade e Acessibilidade

Desde o início, o design deve ser **responsivo**, garantindo uso confortável em smartphones, tablets e desktops. Isso implica em desenvolver *layouts* fluidos ou adaptativos, usando consultas de mídia CSS para ajustar a disposição de elementos conforme a largura de tela. Por exemplo, componentes que em desktop aparecem lado a lado podem se empilhar verticalmente no mobile. Teste o protótipo em diferentes resoluções para assegurar legibilidade e funcionalidade (menus, botões e formulários devem funcionar bem ao toque, sem necessidade de zoom).

Além da responsividade, a **acessibilidade web** não deve ser negligenciada. Seguem algumas boas práticas aplicáveis: - Utilizar HTML semântico adequadamente (por exemplo, `<header>`, `<nav>`, `<main>`, `<footer>`, `<form>` e rótulos `<label>` nos campos de formulário). Isso auxilia tecnologias assistivas a interpretarem a página corretamente [15]. - Fornecer texto alternativo (*atributo* `alt`) para todas as imagens relevantes, de forma que usuários com deficiência visual (utilizando leitores de tela) entendam o conteúdo das figuras [15]. - Garantir contraste adequado entre texto e fundo, seguindo diretrizes WCAG. As diretrizes de identidade visual devem ser aplicadas com cuidado para não criar combinações de cores de baixo contraste que prejudiquem a leitura. - Assegurar que todas as funções são utilizáveis via teclado (navegação por **TAB**, *shift-tab* e **Enter**). Isso inclui links e botões evidentes ao foco do teclado, e evitar componentes que só possam ser acionados via mouse. - Evitar conteúdo muito animado ou que pisque, pois pode ser distractivo ou causar desconforto. Elementos multimídia não devem reproduzir automaticamente som; vídeos devem ter legendas. - Incluir instruções claras e mensagens de erro nos formulários. Se um formulário de cadastro falhar, por exemplo, destacar os campos com problema e fornecer feedback textual útil. - Testar a página com CSS desligado para verificar se a ordem e estrutura do conteúdo ainda fazem sentido (uma página bem estruturada deve ser compreensível linearmente mesmo sem estilo) [16]. - Testar também com JavaScript desligado: funcionalidades críticas (como navegação básica) devem ter *fallback* caso o JS não carregue. Por exemplo, links de navegação que dependem de dropdown em JS devem ainda assim ser acessíveis (um menu simples ou página de sitemap como alternativa).

Um site acessível tende a ser melhor para todos os usuários. Portanto, seguir padrões estabelecidos pelo W3C e diretrizes WCAG é um investimento de qualidade. Conforme uma referência de boas práticas, *"o site deve ser responsivo, ou seja, abrir em celulares, tablets, desktop"* e devemos considerar cenários como CSS desabilitado, evitando efeitos excessivos, etc. [16]. Aplicando essas recomendações, a Plataforma Leitícia será inclusiva e eficiente, oferecendo uma ótima experiência em diferentes contextos de uso.

2. Modelagem do Banco de Dados (MySQL)

Principais Entidades e Relacionamentos

Com base nos requisitos (cadastro de doadoras, registro de doações, integração com bancos de leite e logística), definimos as seguintes entidades principais no banco de dados MySQL:

- **Usuários**: armazenará informações das pessoas que utilizam a plataforma. Aqui incluirá principalmente as doadoras de leite materno, e possivelmente administradores ou operadores de bancos de leite. Campos típicos: nome, e-mail, senha (hash), telefone, endereço, tipo de

usuário (doadora, admin, etc.) e metadados como data de cadastro. Cada usuário pode realizar várias doações.

- **Bancos_de_Leite**: representará os bancos de leite ou postos de coleta parceiros. Guardará nome da instituição, endereço (logradouro, cidade, estado, CEP), contato (telefone, email) e talvez campos como horário de funcionamento. Esses dados podem vir pré-cadastrados de uma fonte oficial. Um banco de leite terá várias doações associadas a ele.
- **Solicitacoes_Doacao**: esta tabela registra as solicitações de doação de leite realizadas pelas usuárias (doadoras). Cada solicitação contém referência à doadora (usuário) que quer doar e ao banco de leite destinatário. Campos: data da solicitação, volume estimado (se informado a quantidade de leite disponível), status (ex.: "Pendente Coleta", "Em Transito", "Concluída", "Cancelada"), forma de entrega (ex.: "coleta domiciliar" ou "entrega pessoal"). Poderíamos separar solicitações e doações efetivadas em tabelas distintas, mas para um MVP podemos unificar: uma solicitação aprovada e concluída representa uma doação realizada. Cada registro se liga a um usuário (quem doa) e a um banco de leite (para onde vai o leite).
- **(Opcional) Logistica/Coletas**: caso queiramos detalhar as interações com serviços de logística, poderia haver uma tabela para acompanhar o agendamento de coleta. Por exemplo, armazena o ID de confirmação da coleta externa (número de pedido na API dos Correios ou outra transportadora), data/hora agendada da retirada e status da coleta (ex.: "Agendada", "Coletado", "Entregue no Banco"). No entanto, esse detalhe pode ser incorporado em campos na tabela de Solicitações de Doação (por exemplo, campos `cod_rastreio` e `data_coleta` na própria solicitação). Para simplicidade, vamos implementar como campos adicionais, evitando uma tabela separada agora.
- **(Opcional) Administradores/Operadores**: poderíamos diferenciar usuários administradores numa tabela à parte ou via um campo de papel na tabela de usuários. Optaremos por um campo na tabela de usuários (`tipo_usuario`), evitando tabelas adicionais. Assim, se `tipo_usuario = 'ADMIN'`, sabemos que aquele registro é um administrador com privilégios; se for `'DOADOR'`, é uma doadora comum; poderíamos ter `'BANCO'` para funcionários de bancos de leite caso haja acesso dedicado para eles.

Os relacionamentos principais são: - Usuário (doadora) **1 - N** Solicitações/Doações (uma usuária pode fazer várias doações). - Banco_de_Leite **1 - N** Solicitações/Doações (um banco de leite recebe muitas doações). - Usuário (administrador) terá permissões especiais, mas está no mesmo domínio de usuário com uma flag. Esses relacionamentos serão implementados via **chaves estrangeiras** no banco.

Script de Criação do Banco de Dados (SQL)

A seguir está o script SQL completo para criar as tabelas descritas, com seus campos, chaves primárias, estrangeiras e índices relevantes. As tabelas utilizam engine InnoDB (para suportar integridade referencial) e conjunto de caracteres utf8mb4 (para acomodar texto Unicode como nomes, endereços com acentos, emojis se necessário). Comentários são incluídos para explicar cada tabela e certos campos:

```
-- ------------------------------------------------------
-- DDL para Banco de Dados da Plataforma Leitícia
-- (MySQL 5.7+ sintaxe)
-- ------------------------------------------------------
CREATE DATABASE if not exists leiticia_db
    CHARACTER SET utf8mb4
    COLLATE utf8mb4_general_ci;
USE leiticia_db;
```

```sql
-- Tabela de Usuários: guarda informações das usuárias doadoras e admins
CREATE TABLE usuarios (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,   -- e-mail único para login/contato
    senha_hash VARCHAR(255) NOT NULL,     -- armazenamento do hash da senha
(ex: 60 caracteres para bcrypt)
    telefone VARCHAR(20),                 -- telefone de contato (com DDD)
    logradouro VARCHAR(150),              -- endereço: rua/av + número
    cidade VARCHAR(100),
    estado CHAR(2),
    cep VARCHAR(10),
    tipo_usuario ENUM('DOADOR','ADMIN') DEFAULT 'DOADOR',  -- define se é
doador comum ou admin/supervisor
    criado_em DATETIME DEFAULT CURRENT_TIMESTAMP,
    atualizado_em DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
) ENGINE=InnoDB;

-- Índice adicional para busca rápida por cidade (ex: listar doadoras de
certa cidade, se necessário)
CREATE INDEX idx_usuario_cidade ON usuarios(cidade);

-- Tabela de Bancos de Leite: lista dos bancos de leite parceiros/postos de
coleta
CREATE TABLE bancos_de_leite (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(150) NOT NULL,
    logradouro VARCHAR(150) NOT NULL,
    cidade VARCHAR(100) NOT NULL,
    estado CHAR(2) NOT NULL,
    cep VARCHAR(10),
    telefone VARCHAR(20),
    email_contato VARCHAR(100),
    criado_em DATETIME DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB;

-- Índice para busca por cidade/estado de bancos (útil para encontrar bancos
próximos)
CREATE INDEX idx_banco_localidade ON bancos_de_leite(cidade, estado);

-- Tabela de Solicitações/Doações: registra doações de leite solicitadas e
seu status
CREATE TABLE solicitacoes_doacao (
    id INT AUTO_INCREMENT PRIMARY KEY,
    usuario_id INT NOT NULL,        -- referência à doadora (usuária que doa)
    banco_id INT NOT NULL,          -- referência ao banco de leite de
destino
    data_solicitacao DATETIME DEFAULT CURRENT_TIMESTAMP,
    volume_ml INT,                  -- volume estimado de leite a ser doado
(em ml, se informado)
```

```
    status VARCHAR(20) NOT NULL DEFAULT 'PENDENTE',  -- ex: PENDENTE, EM
ANDAMENTO, CONCLUIDA, CANCELADA
    forma_entrega VARCHAR(20) DEFAULT 'COLETA',      -- COLETA (buscar na
doadora) ou ENTREGA (doadora leva ao banco)
    codigo_rastreio VARCHAR(50),    -- código de rastreio ou protocolo (se
integrado a transportadora/Correios)
    data_coleta DATETIME,
-- data/hora agendada para coleta (quando aplicável)
    data_conclusao DATETIME,        -- data/hora da conclusão da doação
(efetivada no banco)
    CONSTRAINT fk_solicit_usuario FOREIGN KEY (usuario_id)
        REFERENCES usuarios(id) ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT fk_solicit_banco FOREIGN KEY (banco_id)
        REFERENCES bancos_de_leite(id) ON DELETE RESTRICT ON UPDATE CASCADE
) ENGINE=InnoDB;

-- Índice para consultar solicitações por status (por exemplo, buscar todas
pendentes)
CREATE INDEX idx_solicit_status ON solicitacoes_doacao(status);
```

Explicação do script e regras de integridade: Começamos criando o banco de dados `leiticia_db` (caso não exista) com *charset* UTF-8. Em seguida:

- **usuarios:** Tabela que armazena os usuários (doadoras e administradores). A chave primária é `id`. Definimos campo `email` com índice único para evitar duplicidade [17] (e facilitar login por e-mail). O campo `senha_hash` guardará a senha encriptada – estamos adotando boas práticas de segurança ao não armazenar senhas em claro. Campos de endereço (`logradouro`, `cidade`, `estado`, `cep`) podem ser preenchidos pelas doadoras para viabilizar logística de coleta. O campo `tipo_usuario` distingue perfis; usamos um tipo enumerado simples (`'DOADOR'` ou `'ADMIN'`) para indicar privilégios. Os campos de timestamp (`criado_em` e `atualizado_em`) ajudam a auditar criação e última modificação do cadastro. Após a criação da tabela, sugerimos um índice em `cidade` caso haja necessidade de consultar usuárias por localização (não obrigatório, mas útil se o sistema listar doadoras por região).

- **bancos_de_leite:** Tabela dos bancos de leite parceiros. Tem `id` auto-incremento como chave primária. Armazena nome e dados de localização/contato do banco. Não há campo de login aqui, pois funcionários/admins podem ser geridos via tabela de usuários (um eventual operador do banco de leite poderia ter um usuário com `tipo_usuario='ADMIN'` ou criamos futuramente um vínculo). Adicionamos um índice composto em `(cidade, estado)` para facilitar buscas de bancos por região (ex.: encontrar todos bancos no *estado X* ou *cidade Y* para mostrar ao usuário).

- **solicitacoes_doacao:** Tabela que registra as doações solicitadas/efetivadas. A PK é `id`. Os campos `usuario_id` e `banco_id` são chaves estrangeiras referenciando respectivamente `usuarios(id)` e `bancos_de_leite(id)`. Definimos **foreign keys** com *ON DELETE RESTRICT* para impedir remoção de usuários ou bancos que tenham doações associadas (protege integridade dos dados históricos), e *ON UPDATE CASCADE* para caso haja mudança de um id (situação rara, mas mantém consistência). Campos principais: `data_solicitacao` registra quando a doadora fez a solicitação no sistema; `volume_ml` é opcional para indicar quantos mL a doadora estima doar; `status` indica o estado do processo (ex.: inicia em "PENDENTE", depois

"EM ANDAMENTO" quando em coleta, "CONCLUIDA" quando o leite chega ao destino, ou "CANCELADA" se for cancelado). Poderíamos usar um ENUM para status, mas optamos por VARCHAR para flexibilidade de novos estados. `forma_entrega` diferencia se será coleta domiciliar ou se a doadora entregará pessoalmente no posto (útil para logística). Se integrando com serviço de entrega, `codigo_rastreio` armazena o código de rastreio fornecido (por ex., um código dos Correios ou ordem de serviço da transportadora) e `data_coleta` quando a coleta está agendada. Ao término, `data_conclusao` marca quando a doação foi finalizada/recebida. Incluímos também um índice em `status` para otimizar consultas administrativas, como "ver todas doações pendentes".

Todas as tabelas estão em InnoDB para suportar transações e integridade referencial. Os índices e restrições garantem performance e consistência: por exemplo, ao inserir uma solicitação, o banco assegurará que `usuario_id` e `banco_id` existam nas respectivas tabelas, e não será possível deletar, por exemplo, um `usuario` que ainda tenha doações pendentes vinculadas – isso evita * orphan records*. Em casos de remoção lógica (ex.: usuário solicitar exclusão via LGPD), o ideal seria um campo de flag (ativo/inativo) ao invés de deletar fisicamente; ou, se deletar fisicamente, primeiro remover ou reassociar seus registros de doação (nesse MVP, o fluxo de remoção deve ser cuidadosamente tratado pela aplicação para manter consistência).

Regras de Integridade Referencial e Índices

Já comentamos algumas regras implementadas via *foreign keys*. Recapitulando: - A integridade referencial garante que não teremos uma solicitação apontando para um usuário ou banco inexistente. As FKs `fk_solicit_usuario` e `fk_solicit_banco` cuidam disso. Com *ON DELETE RESTRICT*, se tentar remover um usuário que tenha doações solicitadas, o banco impedirá a operação – isso protege dados históricos (essencial para relatórios, accountability). Em cenários onde se desejaria cascata (por ex., deletar automaticamente todas solicitações de um usuário removido), poderíamos usar *ON DELETE CASCADE*, mas isso pode conflitar com LGPD (idealmente, apagar um usuário deveria também anonimizar as doações, em vez de removê-las do histórico). Para o MVP, manteremos restrict e trataremos remoções via lógica de aplicação. - *ON UPDATE CASCADE* permite que, se o ID de um usuário ou banco mudasse (o que praticamente não acontece, já que são auto-increment keys sem motivo para alterar), a referência na tabela de solicitações seria atualizada automaticamente. É mais uma medida de segurança para consistência, embora pouco acionada.

Sobre **índices**: além dos índices primários (PK) e das chaves únicas (email do usuário), adicionamos alguns índices secundários: - `idx_usuario_cidade` em `usuarios(cidade)` para agilizar consultas por localização de usuárias (por exemplo, se quisermos analisar de onde vêm mais doadoras, ou implementar alguma funcionalidade geográfica). - `idx_banco_localidade` em `bancos_de_leite(cidade, estado)` para poder filtrar bancos por cidade/estado rapidamente – isso será útil ao listar bancos próximos de uma doadora (filtro por estado ou cidade selecionada). - `idx_solicit_status` em `solicitacoes_doacao(status)` para auxiliar listagens administrativas, como “todas doações pendentes” ou “todas concluidas”. Como status é um campo de baixa cardinalidade, o índice ajuda quando há muitos registros, embora consultas por status junto com outras condições (ex.: por banco e status) possam exigir índice composto no futuro.

Esses índices são **sugestões** com base em possíveis consultas. Conforme a aplicação evoluir, devemos monitorar as consultas reais e ajustar índices (indexar colunas muito utilizadas em filtros ou junções). Índices em colunas como `email` (já unique index), ou `nome` não foram explicitados além do unique porque normalmente buscas por nome podem usar `LIKE` (nesse caso índices tipo *FULLTEXT* ou

similares seriam outra abordagem se necessário). Para o MVP, a estrutura acima é suficiente e equilibrada.

Comentários sobre Cada Tabela

(Obs: Os comentários também foram entremeados no script SQL acima como `-- comentário`, mas reiteramos aqui de forma mais descritiva para enfatizar o propósito de cada tabela.)

- **Tabela `usuarios`:** Armazena os dados dos usuários da plataforma. Inclui doadoras de leite e administradores. Cada registro tem informações pessoais e de contato. O propósito é identificar quem está usando a plataforma, autenticar o login (via email e senha_hash) e fornecer dados para contato e logística (endereço para coleta, telefone). A coluna `tipo_usuario` permite distinguir privilégios (ex.: um administrador poderia acessar um painel geral, enquanto uma doadora acessa apenas suas funcionalidades). Comentário de implementação: assegurar hashing de senha e talvez armazenar CPF se necessário (não listado acima, mas poderia ser um campo extra se houver necessidade de identificação única adicional, nesse caso também unique index).

- **Tabela `bancos_de_leite`:** Contém a lista de bancos de leite humanos ou postos de coleta parceiros. Serve para apresentar às doadoras as opções de onde seu leite pode ser enviado e armazenar dados de contato que podem ser usados pela logística (endereço de destino da entrega, ou para exibir num mapa). Essa tabela provavelmente será pré-populada com dados fornecidos pelas diretrizes do projeto (mockups ou material indicam parceiros?) e pode ser atualizada via script ou admin. É referenciada nas solicitações para vincular cada doação a um destino específico.

- **Tabela `solicitacoes_doacao`:** Registra cada interação de doação. Cada linha representa uma oferta de doação de leite feita por uma usuária doadora, destinada a um banco de leite específico. O propósito desta tabela é acompanhar o fluxo desde a solicitação até a conclusão. Ela agrega informações de status e logística, servindo como registro operacional (o "coração" da plataforma em termos de atividade). Com os campos de status e datas, poderemos gerar relatórios, por exemplo: quantas doações foram concluídas em determinado mês, quantas estão pendentes, tempo médio entre solicitação e conclusão, etc. Comentário: no MVP, a alteração de status provavelmente será manual (um admin ou funcionário marca quando foi coletado/concluído), mas podemos futuramente integrar isso com confirmações automáticas via API de logística (ex.: quando transportadora marca entrega efetuada, atualizar status para "CONCLUIDA").

Em suma, a modelagem busca ser **simples e clara**, cobrindo as entidades centrais para um MVP funcional. Está normalizada de forma básica (cada dado no lugar certo, evitando duplicação – por exemplo, os dados do banco de leite não se repetem em cada solicitação, apenas referenciamos via ID). Isso facilita manter consistência. Ainda assim, há espaço para expansão conforme novas funcionalidades surjam (por ex.: tabelas de mensagens, avaliações, histórico de estoque de leite, etc., poderiam ser adicionadas no futuro sem refazer as bases). Para o escopo atual, contudo, as três tabelas definidas suprem o fluxo principal (usuária se cadastra -> faz solicitação de doação -> doação é encaminhada a um banco de leite).

3. Organização do Projeto para Visual Studio Code e Deploy

Estrutura de Pastas do Projeto

Manter uma estrutura de pastas organizada é fundamental para que todos da equipe encontrem rapidamente o que precisam e para que o projeto seja facilmente mantido. Segue uma sugestão de estrutura de diretórios para o projeto web (considerando frontend separado do backend em grande medida, mas dentro do mesmo repositório):

```
leiticia-project/
├── index.php          # Página inicial (se homepage dinâmica) ou index.html
├── paginas/           # (Opcional) páginas HTML/PHP adicionais separadas
│   ├── cadastro.php   # Página de cadastro de usuária doadora
│   ├── login.php      # Página de login
│   ├── perfil.php     # Perfil da doadora (com lista de doações dela)
│   ├── admin/         # Subpasta para páginas admin protegidas
│   │   └── dashboard.php  # Exemplo de página administrativa
│   └── ...            # outras páginas
├── assets/            # Pasta de arquivos estáticos (assets gerais)
│   ├── css/
│   │   └── estilo.css # Arquivo CSS principal com identidade visual aplicada
│   ├── js/
│   │   └── script.js  # JS principal para interações front-end
│   ├── img/
│   │   └── ...        # Imagens do site (logo, ícones, etc.)
│   └── fonts/         # (se houver fontes customizadas da identidade visual)
├── includes/          # Scripts PHP incluíveis (não acessíveis diretamente)
│   ├── config.php     # Configurações gerais (ex: dados de conexão do DB)
│   ├── conexao.php    # Script de conexão com o BD (usado em outros scripts)
│   ├── funcoes.php    # Funções utilitárias (ex: sanitização, envio de
email)
│   └── ...            # outros includes (cabeçalho, rodapé se usar
templates)
├── api/               # (Opcional) Endpoints API REST (separados do front
tradicional)
│   ├── doacoes.php    # endpoint para criar/listar doações (retorna JSON)
│   └── ...            # outros endpoints
└── vendor/            # (Opcional) bibliotecas de terceiros via Composer ou
npm
    └── ...
```

Detalhes da organização:

- Colocamos um `index.php` na raiz, assumindo que vamos usar PHP para renderizar a home dinâmica (se fosse totalmente estática e sem necessidade de processamento inicial, poderia ser `index.html`). Nesse index, podemos mostrar um resumo da plataforma ou formulário inicial.

- A pasta `paginas/` contém as páginas da aplicação. Separar por funcionalidade ajuda. Por exemplo, todas páginas de usuário comum podem ficar diretamente, enquanto páginas

administrativas numa subpasta `admin/` (que podemos proteger via um controle de acesso no topo dos scripts, verificando `tipo_usuario`). Essa separação física também auxilia a aplicar restrições via .htaccess se necessário (ex: negar acesso se não logado).

- A pasta `assets/` agrupa os recursos estáticos. Dentro dela:

- `css/` terá os arquivos de estilo. Idealmente um único arquivo principal (`estilo.css`) compilando todo o CSS do site (facilita cache e carregamento; podemos usar SASS/SCSS se quiser modularizar durante desenvolvimento e compilar num arquivo final). As **diretrizes de identidade visual** fornecidas devem refletir aqui: por exemplo, definiremos variáveis SASS ou no começo do CSS para cores da paleta, fontes (se custom, colocadas em `assets/fonts` e referenciadas no CSS) e padrões de espaçamento. Isso garante consistência visual.
- `js/` para scripts JavaScript do front-end. Novamente, centralizar em um arquivo quando possível (ex: `script.js`). Se funcionalidades específicas exigirem, podemos ter vários arquivos (por exemplo, `maps.js` para funcionalidades de mapa). Utilize *module bundlers* ou importe via `<script>` conforme a complexidade – para um MVP, incluir diretamente alguns arquivos JS no HTML pode ser suficiente. Organize códigos JS em módulos lógicos (por funcionalidade) se crescerem.
- `img/` para imagens. Aqui entra o logo da plataforma, imagens usadas no layout e eventuais ícones. Nomeie os arquivos de forma clara e, se as diretrizes de identidade visual incluírem versões específicas de logos (ex: positivo, negativo, versões para web), mantenha-os.

- `fonts/` se forem utilizadas fontes customizadas (.woff, .ttf) ou ícones de fonte (como FontAwesome, se baixado localmente), mantê-los aqui.

- A pasta `includes/` para scripts do lado do servidor que não são acessados diretamente pelo navegador, mas sim *incluídos* nas páginas PHP. Por exemplo:

- `config.php`: pode definir variáveis de configuração, constantes (como as credenciais de banco de dados, chave de API, etc.). Recomenda-se manter neste arquivo as configurações e usá-lo no topo das outras páginas (`require 'includes/config.php';`).
- `conexao.php`: utiliza os parâmetros de `config.php` para estabelecer conexão com MySQL (usando mysqli ou PDO). Assim, qualquer página que precisar realizar queries primeiro inclui `conexao.php`. A conexão pode ser criada com `mysqli_connect` conforme exemplo da Hostinger [18], usando `localhost`, nome do BD, usuário e senha definidos no hPanel. Lembrando que na Hostinger, o host normalmente é `localhost` mesmo (quando PHP e MySQL estão no mesmo servidor) [17]. Alternativamente, se fosse usar PDO: `$pdo = new PDO("mysql:host=$servername;dbname=$database", $username, $password, [...]);`.
- `funcoes.php`: para funções PHP reutilizáveis, como sanitização de input, verificação de login (ex: função `verificaLogin()` que checa se `$_SESSION` do usuário está ativa, senão redireciona), formatação de datas, etc.

- Podemos ter `header.php` e `footer.php` aqui, se optarmos por montar as páginas de forma modular (incluindo um cabeçalho padrão e rodapé em todas). Isso ajuda a manter consistência do layout e facilita mudanças futuras (edita-se um header.php, reflete em todas as páginas).

- Pasta `api/` (opcional mas recomendada se seguirmos com um frontend desacoplado ou partes do sistema via AJAX): Podemos criar endpoints PHP que retornam JSON para certas ações. Por exemplo, `api/doacoes.php` poderia responder a GET (listando doações em formato JSON, talvez restrito por usuário se autenticado via token ou sessão) e POST (criando uma nova

solicitação de doação recebendo dados em JSON). Esse design facilita caso no futuro haja um app móvel consumindo as mesmas APIs. Cada arquivo em `api/` serviria a uma entidade ou grupo de funcionalidades. Importante: ao implementar, cuidar da segurança (ex: exigir token/API key ou verificar a sessão do usuário para evitar acesso indevido).

- Pasta `vendor/`: se usarmos **Composer** (PHP) para gerenciar dependências, ou se utilizarmos pacotes JS via **npm** (Node, caso aplicável ou só para ferramentas de front-end), esses arquivos ficam nesta pasta (no caso do Composer) ou em uma pasta `node_modules` (no caso de npm, que geralmente não vai pro servidor se for só build). Por exemplo, se decidirmos usar PHPMailer para envio de email, instalando via Composer, ele residirá em `vendor/`. No MVP, podemos até evitar dependências complexas, mas deixamos o diretório indicado.

Em ambiente de desenvolvimento local, pode-se servir o site usando o **Live Server** (para páginas estáticas) ou preferencialmente configurar um ambiente PHP local (XAMPP, Laragon, etc.) para testar as páginas PHP com backend. Se usar VS Code Live Server diretamente, ele não processa PHP, então para testar páginas dinâmicas seria necessário rodar o PHP embutido (`php -S localhost:8000`) ou usar extensões específicas de servidor PHP [19]. Uma opção conveniente é usar o recurso do VS Code chamado **PHP Server** (extensão) que inicia um servidor local interpretando PHP.

Ao preparar para o **deploy na Hostinger**, é importante adequar a estrutura: - No Hostinger, o diretório público é geralmente `public_html/`. Todo conteúdo acessível da web deve estar lá dentro. Logo, ao fazer upload do projeto, o conteúdo de `leiticia-project` deve ser colocado dentro de `public_html`. Por exemplo, `index.php`, as pastas `assets/`, `paginas/`, `includes/` etc., todas diretamente em `public_html` ou suas subpastas correspondentes. - Arquivos sensíveis (config) idealmente ficariam fora de `public_html` por segurança. No Hostinger, não há acesso a pasta acima de `public_html` em hospedagem compartilhada, então uma prática é proteger o acesso direto a esses arquivos com `.htaccess` (por exemplo, criar um `.htaccess` dentro de `includes/` negando qualquer requisição web, já que só precisam ser incluídos internamente). - Verificar nomes de arquivos e caminhos: Hostinger em ambientes Linux diferencia maiúsculas de minúsculas, então manter padrão (tudo minúsculo por exemplo) evita problemas de rota quebrada ao publicar.

Versionamento com GitHub (Fluxo Sugerido)

Usar o GitHub para versionamento trará muitos benefícios à equipe: histórico de mudanças, colaboração simultânea e backup central do código. Para uma equipe pequena, podemos adotar um fluxo Git simplificado: - **Repositório inicial:** Criar um repositório no GitHub para o projeto. Todos os membros clonam esse repo. - **Branching básico:** Podemos definir a branch principal como `main` (ou `master`). Desenvolvimentos maiores ou novas features podem ser feitos em branches separados para não quebrar o código principal. Por exemplo, se um desenvolvedor vai implementar a integração com API de mapas, pode criar a branch `feature/maps-api`. Após concluir e testar localmente as mudanças, ele abre um Pull Request para mesclar na `main`. Outro membro revisa (mesmo que seja só um olhar para ver se está ok) e então faz o *merge*. Em uma equipe muito pequena, o mesmo desenvolvedor pode mesclar, mas é bom hábito revisar antes. - **Commits frequentes e claros:** Cada membro deve fazer commits atomizados, ou seja, ao concluir uma parte lógica do trabalho (ex: "Criar tabela X", "Implementar validação do formulário de login") para facilitar identificar o que cada alteração faz. Escrever mensagens de commit descritivas é importante: no lugar de "update" ou "fix", prefira "Correção do SQL de inserção na tabela solicitacoes_doacao" ou "Estilos CSS para tela de perfil do usuário". - **Gitignore:** Configurar o arquivo `.gitignore` para evitar comitar arquivos sensíveis ou desnecessários. Por exemplo, ignore o `includes/config.php` se nele houver senhas (nesse caso, mantenha um exemplo de config sem credenciais no repo e cada dev tem o seu local). Ignore também

diretórios como `vendor/` (se for recriável via Composer) ou `node_modules/` (se usar npm) – eles podem ser reinstalados em vez de versionados, economizando espaço e evitando ruídos nos commits. - **Issues e Milestones:** No GitHub, aproveite a seção de *Issues* para registrar tarefas, funcionalidades a fazer ou bugs encontrados. É uma boa prática, mesmo em MVP, listar as funcionalidades e marcar quem está fazendo o quê, garantindo que nada seja esquecido e evitando duas pessoas trabalharem na mesma coisa sem saber. Para organização temporal, podem usar *Milestones* (ex: "MVP v1.0" agregando as issues básicas). - **Releases e deploy:** Quando estiver pronto para uma versão testável ou entrega, pode-se criar uma tag de release no GitHub (ex: v1.0.0). Isso ajuda a marcar pontos no histórico para referência. O código de produção (Hostinger) deve estar sincronizado com uma versão estável do repositório. Pode-se até integrar deploy contínuo se desejado, mas manualmente: clonar/puxar do GitHub para Hostinger (via Git pelo painel ou FTP) quando for lançar atualizações.

Ferramentas como **GitLens** (extensão do VS Code) podem ser utilizadas para melhorar a visibilidade do histórico dentro do editor. O GitLens mostra quem editou cada linha por último e permite navegar pelos commits facilmente [20]. Isso ajuda na revisão de código e entendimento das mudanças. Outra dica é habilitar o **GitHub Pull Requests** extension no VS Code, se quiserem revisar PRs dentro do editor.

Em resumo, adotar um fluxo de *feature branches* integrado na `main` (algo próximo ao GitFlow, mas simplificado) deve atender bem. Dado que o projeto é um MVP, evitar complexidade excessiva no fluxo (não há necessidade de muitas branches de longa duração). O importante é sempre ter o código funcionando na branch principal e fazer merges frequentes para evitar grandes desvios.

Extensões Úteis no Visual Studio Code

O Visual Studio Code, por si só, já oferece um bom ambiente para desenvolvimento web, mas certas extensões podem aumentar a produtividade e auxiliar na qualidade do código. Algumas recomendações de plugins para a equipe:

- **Live Server:** Já mencionado, esse plugin levanta um servidor local servindo os arquivos estáticos e recarrega o navegador automaticamente a cada mudança salva [21]. Embora seja excelente para trabalhar o HTML/CSS/JS em tempo real, lembre-se de que para visualizar páginas PHP ele não é suficiente (neste caso, usar a extensão PHP Server ou um servidor local). No entanto, para prototipagem do front-end, Live Server agiliza muito o *feedback loop* de desenvolvimento, evitando a necessidade de ficar apertando F5 no navegador manualmente.

- **GitLens:** Auxilia no controle de versão integrado ao VSCode, mostrando blame (quem alterou o que) em linhas de código, histórico de commits por arquivo, comparações, etc., diretamente no editor [20]. Isso facilita entender modificações e depurar quando algo que funcionava parou (basta ver o histórico do arquivo e talvez identificar o commit responsável).

- **Prettier – Code Formatter:** Uma extensão de formatação de código automática. Configurada adequadamente, ela formata o código ao salvar, padronizando indentação, aspas, pontuação etc. Funciona para HTML, CSS, JavaScript e também tem suporte a PHP com plugin [22] [23]. Isso ajuda a manter um estilo consistente, especialmente útil com múltiplos desenvolvedores editando os mesmos arquivos. O Prettier pode ser configurado via `.prettierrc` para adaptar às preferências (por exemplo, usar aspas simples vs duplas, etc.).

- **PHP Intelephense**: Extensão que oferece *autocompletion* e verificação de erros para PHP. Facilita ao sugerir funções, destacar problemas de sintaxe, e navegação para definições. Especialmente

útil se a equipe não tiver um IDE completo para PHP; o VS Code com Intelephense supre grande parte das funcionalidades encontradas em IDEs como PHPStorm.

- **PHP Server**: Permite rodar um servidor PHP embutido do VS Code facilmente (similar ao Live Server, mas para PHP) [19]. Com ele, você clica com botão direito em um arquivo PHP e escolhe "PHP Server: Serve project" e ele inicia um servidor local acessível (geralmente em http://localhost:3000 ou porta similar) interpretando o PHP. Isso é muito útil para testar a aplicação sem precisar configurar Apache/Nginx localmente.

- **SQLite / MySQL extensions**: Existem extensões para gerenciar bancos de dados diretamente no VSCode, como *MySQL Management Tool* ou *vscode-database*. Podem ser úteis para executar consultas de teste ou visualizar tabelas sem sair do editor. Alternativamente, usar a interface do **phpMyAdmin** que a Hostinger fornece ou um cliente dedicado como MySQL Workbench pode ser considerado.

- **ESLint** (se for usar Node/JS intensivamente no front): Ajuda a manter padrão de código e detectar erros de JS. Se o projeto JS for pequeno e simples, talvez não seja crítico, mas em geral é bom para qualidade do código JS.

- **CSS IntelliSense / CSS Peek:** Extensões que ajudam no HTML/CSS. O CSS Peek, por exemplo, permite *Ctrl+Clique* sobre um nome de classe no HTML e ir direto para a definição dessa classe no CSS [24]. Isso agiliza o styling. Já o IntelliSense for CSS class names sugere nomes de classes existentes enquanto você digita no atributo class do HTML [25], evitando erros de digitação e acelerando o processo.

Todos esses plugins visam reduzir o tempo gasto em tarefas repetitivas ou propensas a erro e melhorar a eficiência do time.

Orientações para Publicação na Hostinger

Publicar o MVP na Hostinger requer alguns passos e cuidados práticos:

- **Configuração do ambiente MySQL:** Primeiro, criar o banco de dados MySQL pelo painel da Hostinger (hPanel). No menu "Banco de dados MySQL", criar uma nova base com nome, usuário e senha [26]. Anotar esses detalhes. No Hostinger, o hostname do MySQL geralmente será `localhost` (quando o script PHP roda no mesmo host do BD) [17], portanto normalmente não precisa alterar isso. Em casos de planos específicos, Hostinger pode fornecer um host do tipo `mysqlxxxx.hostinger.com` – verificar no painel, mas via de regra é localhost para hospedagem compartilhada.

- **Upload dos arquivos:** Há diversas formas. Pode usar o gerenciador de arquivos do hPanel para enviar os arquivos zipados e extrair lá, ou configurar o Git via SSH (Hostinger permite git push deployment em alguns planos) – para simplicidade inicial, o upload manual pode ser ok. Certifique-se de colocar os arquivos dentro de `public_html`. Por exemplo, `index.php` vai em `public_html/index.php`. Se o projeto estiver em GitHub, pode baixar um zip do repositório ou usar um cliente FTP (ex: FileZilla) para enviar todos os diretórios e arquivos.

- Atenção: não sobrescrever pastas padrão do hostinger como `public_html` (colocar dentro dela).

- Após upload, verifique permissões de arquivos (arquivos PHP geralmente 644, pastas 755, que é padrão ao enviar).

- **Ajuste das configurações de conexão no código:** Edite `includes/config.php` ou equivalente com as credenciais do BD criadas no painel. Exemplo:

```
<?php
$db_host = "localhost";
$db_name = "nomeDaBase";
$db_user = "usuarioDaBase";
$db_pass = "senhaDefinida";
?>
```

O Hostinger usa `localhost` como host se a aplicação PHP estiver local [17]. Em seguida, no `conexao.php`, usar essas variáveis:

```
$conn = mysqli_connect($db_host, $db_user, $db_pass, $db_name);
if (!$conn) {
    die("Connection failed: " . mysqli_connect_error());
}
mysqli_set_charset($conn,"utf8");
```

Isso estabelecerá a conexão. Para testar, você pode criar um arquivo `test_conn.php` temporário que inclui a conexão e faz `echo "conectado"` para verificar no navegador (lembrar de remover depois).

- **Importação do banco de dados:** Utilize o phpMyAdmin do Hostinger (acessível pelo hPanel) para importar o script SQL das tabelas. No phpMyAdmin, selecione o BD criado e use a aba "Importar" para enviar o arquivo `.sql` com as tabelas (como gerado na seção anterior). Se o script de criação de DB estiver incluso, remova a linha `CREATE DATABASE` e o `USE` ou simplesmente assegure que está importando dentro do DB correto. Depois de rodar, confira se as tabelas `usuarios`, `bancos_de_leite`, `solicitacoes_doacao` aparecem. **Dica:** pode também inserir alguns dados iniciais (se não foram inseridos via script) – por exemplo, popular a tabela `bancos_de_leite` com alguns registros reais ou fictícios de bancos de leite para teste, e talvez criar um usuário admin manualmente (inserindo na tabela `usuarios` um registro com email e senha hash conhecida) para conseguir logar inicialmente.

- **Configurações específicas da Hostinger (HTTPS e PHP):** Ative o SSL (Hostinger oferece certificado gratuito via Let's Encrypt pelo hPanel). Assim, use URLs `https://` para o site, garantindo segurança. No código PHP de conexão, se estiver usando SMTP para emails ou cURL para APIs, certifique-se de que as extensões estão habilitadas (Hostinger geralmente já habilita cURL, OpenSSL, etc.). Caso necessite ajustar versões de PHP, no hPanel há opção para selecionar a versão do PHP – escolha uma versão estável recente (PHP 8.x, por exemplo, pois tem melhorias de performance e segurança).

- **Testes pós-deploy:** Depois de subir tudo, acesse o domínio (ou subdomínio provisório fornecido pela Hostinger) e teste: carregue a página inicial, realize um cadastro de teste, veja se os dados aparecem no banco, etc. Erros comuns a verificar:

- Path errados (caso tenha usado caminhos absolutos incorretos, mas usando relativos como mostrado na estrutura geralmente funciona).
- Permissão de escrita: se a aplicação precisar salvar arquivos (ex: foto de perfil, etc.), a pasta destino (por ex. `uploads/`) deve ter permissão de escrita (CHMOD 775/777 conforme necessidade). No nosso escopo, talvez não tenha upload de arquivos, mas fica a nota.
- Erros de conexão com BD: se aparecer "Connection failed..." no browser, rever credenciais ou se o servidor `localhost` é o certo. Como mencionado, Hostinger usualmente usa localhost mesmo [17].

- Ajuste do timezone do PHP para America/Sao_Paulo (se relevante) via `date_default_timezone_set` no config, para que as datas registradas estejam corretas localmente.

- **Manutenção contínua:** Utilize o Git para controlar versões. Possivelmente, não queremos subir arquivos `.git` para o servidor. Em vez disso, pode-se usar o próprio hostinger para puxar do repositório (Hostinger tem opção de Git deployment onde você aponta seu repositório e ele permite dar um "Pull" do código). Isso facilita atualizar o site: faz alterações localmente, *commit/push* no GitHub, e do painel Hostinger atualiza. Se não for usar esse recurso, manter um controle manual das alterações e subir via FTP os arquivos modificados a cada atualização, tomando cuidado para não sobrescrever configurações locais (por isso, separar config em um arquivo não versionado é útil – evita que ao dar deploy você sobrescreva a configuração de produção com uma vazia ou de teste).

Em síntese, a publicação envolve replicar no servidor o que foi testado em local, ajustando as configurações de produção. Uma vez no ar, é bom também habilitar ferramentas de monitoramento de erro (Hostinger loga erros PHP em `error_log`; pode ser acessado via FTP). Considere ativar no `config.php` do hostinger `error_reporting(0)` e `display_errors(0)` para não mostrar erros ao usuário final (por segurança), mas mantenha log para depurar.

Boas Práticas Adicionais e Dicas para Equipe Reduzida

Para finalizar, algumas orientações gerais que ajudarão no sucesso do projeto, dado o time enxuto: - **Documentação interna:** Comentem o código de forma esclarecedora (sem exagero, mas explicando trechos não triviais). Um novo membro deve conseguir entender a lógica olhando código e comentários. Por exemplo, no PHP, comentar a lógica de atualização de status da doação; no JS, comentar uma função que calcula distância até o banco de leite (se houver). - **Divisão de tarefas:** Aproveitando a modularização, membros podem trabalhar em partes diferentes sem conflito – ex.: um cuida do front-end (HTML/CSS/JS), outro do back-end (PHP, SQL). Depois juntam as peças. Utilize stubs/mockups: frontender pode criar formulário e em vez de funcionalidade real de envio, colocar um placeholder que o backender depois conecta ao banco. - **Performance simples:** Embora MVP não demanda otimizações prematuras, faça o básico – compressão de imagens (não subir imagens muito pesadas para ícones/logos), minificação de CSS/JS se possível (ou pelo menos não escrever código JS/CSS muito repetitivo). Hostinger tem limite de recursos, então páginas enxutas carregam mais rápido. - **Responsividade e teste cross-browser:** Testem nos principais navegadores (Chrome, Firefox, Safari, Edge) e em mobile (se possível, em dispositivos reais ou usando DevTools emular). Pequenas diferenças de CSS/JS podem surgir e consertar cedo evita suporte ruim depois. - **Backup:** Mantenham backup do banco de dados (podem versionar um dump inicial do SQL no GitHub, mas após produção, backup real via Hostinger periodicamente). Para código, o GitHub já é backup. - **Escalabilidade futura:** A solução apontada é leve (PHP/MySQL em host compartilhado). Isso deve suportar o MVP e até um número moderado de usuários. Se o projeto crescer, considere então migração para um servidor mais robusto ou otimizações (ex.: cache de páginas, CDN para arquivos estáticos, etc.). Desde já, escrever código

claro e separar responsabilidades facilita escalar posteriormente (por exemplo, poder trocar o banco MySQL por outro, ou migrar de PHP para Node microserviços, etc., tendo uma base organizada). - **Conformidade LGPD contínua:** Mantenha registro do consentimento dos usuários (pode ser um campo `aceitou_termos` com timestamp em `usuarios`). Tenha um processo para atender pedidos de informação ou exclusão de dados. Isso demonstra compromisso e evita problemas legais futuros.

Seguindo esse plano detalhado – da arquitetura técnica até a implantação – a equipe terá um guia passo a passo para construir a Plataforma Leitícia de forma consistente com as expectativas dos stakeholders (conforme mockups e identidade visual fornecidos) e boas práticas de desenvolvimento de software. O resultado será um MVP funcional, seguro e preparado para evoluções, mesmo tendo sido feito por uma equipe pequena e com recursos limitados, pois focamos em soluções simples, eficientes e de fácil manutenção. Boa sorte e bom desenvolvimento!

Referências Utilizadas:

- Vantagens da separação frontend/backend [1].
- Considerações sobre Node.js vs PHP em ambiente Hostinger [3] [4].
- Exemplo de API de logística (Correios) para agendamento de coletas [7].
- Princípios de privacidade e segurança da LGPD (minimização de dados, proteção) [9] [10].
- Uso de prepared statements para evitar SQL injection [11] [12].
- Boas práticas de acessibilidade e responsividade [16].
- Sugestões de extensões VS Code (GitLens, Live Server) [20] [21].
- Conexão PHP/MySQL na Hostinger (configurações de host e exemplo de código) [17] [18].

---

1 2 php - Separar ou não o frontend do backend? - Stack Overflow em Português
https://pt.stackoverflow.com/questions/176196/separar-ou-n%C3%A3o-o-frontend-do-backend

3 O Node.js é compatível com a Hostinger? | Central de Ajuda da Hostinger
https://support.hostinger.com/pt/articles/1583661-o-node-js-e-compativel-com-a-hostinger

4 node.js - can i host nodejs react project on hostinger shared hosting? - Stack Overflow
https://stackoverflow.com/questions/63750754/can-i-host-nodejs-react-project-on-hostinger-shared-hosting

5 Banco de Leite Humano — Ministério da Saúde
https://www.gov.br/saude/pt-br/acesso-a-informacao/acoes-e-programas/banco-de-leite-humano

6 Overview | Places API - Google for Developers
https://developers.google.com/maps/documentation/places/web-service/overview

7 API Coleta — Correios
https://www.correios.com.br/atendimento/developers/apicoleta

8 Pathfind | Gestão logística otimizada e dinâmica
https://www.pathfind.com.br/

9 10 13 14 Conheça os 10 Princípios da Lei Geral de Proteção de Dados Pessoais — Laboratório Nacional de Computação Científica - LNCC
https://www.gov.br/lncc/pt-br/centrais-de-conteudo/campanhas-de-conscientizacao/campanha-lgpd/2024/conheca-os-10-principios-da-lei-geral-de-protecao-de-dados-pessoais

11 12 17 18 26 Guia para aprender como conectar PHP com MySQL
https://www.hostinger.com/br/tutoriais/como-conectar-php-com-mysql

15 16 Acessibilidade Web: Boas práticas para uma internet mais igualitária
https://abale.com.br/acessibilidade-web-boas-praticas-para-uma-internet-mais-igualitaria/

19 How do I configure vscode live server to process php files properly (I ...
https://stackoverflow.com/questions/62255030/how-do-i-configure-vscode-live-server-to-process-php-files-properly-im-using-w

20 21 24 25 10 Must-Have VSCode Extensions for Web Development | by GautamManak | Medium
https://medium.com/@gautammanak1/10-must-have-vscode-extensions-for-web-development-44b0d129ae56

22 Prettier - Code formatter - Visual Studio Marketplace
https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode

23 Essential VS Code Extensions for PHP & Laravel Development
https://dev.to/nasrulhazim/essential-vs-code-extensions-for-php-laravel-development-ah