

Estudo de Viabilidade Técnica

1. Otimização do Carregamento (PDF de 22MB)

Viabilidade: Alta.

Desafio: Carregar 22MB de uma vez em 4G/Mobile trava o navegador.

Solução Técnica:

- **Não carregue o arquivo inteiro:** A pior abordagem seria enviar o arquivo completo para o src de um visualizador.
- **Abordagem Recomendada (PDF.js com Range Requests):** Utilizar a biblioteca **Mozilla PDF.js**. Ela permite renderizar o PDF no navegador (client-side) e, se o servidor Django estiver configurado corretamente (aceitando *HTTP Range Requests*), o navegador baixará apenas os bytes necessários para exibir a página atual, não os 22MB de uma vez.
- **Abordagem Alternativa (Pre-renderização):** Usar uma biblioteca Python (como pdf2image) no backend para converter cada página do PDF em uma imagem (WebP ou JPEG) e servir essas imagens sob demanda. Isso é mais leve para o processador do celular do aluno, mas perde a capacidade de selecionar texto nativamente. *Recomendo começar com PDF.js.*

2. Registrar a última página lida (Bookmark)

Viabilidade: Alta.

Solução Técnica:

- Criar um modelo no Django (UserProgress) relacionando User, Documento e CurrentPage.
- No Frontend, usar JavaScript para disparar um evento (fetch/AJAX) para a API do Django toda vez que o aluno mudar de página (ou a cada X segundos).
- Ao abrir o app, o Django consulta o banco e o Frontend inicia o visualizador na página retornada.

3. Recurso de TTS (Text-to-Speech)

Viabilidade: Média/Alta (depende da qualidade do PDF).

Solução Técnica:

- **Extração:** O PDF precisa ter texto selecionável (não pode ser uma imagem escaneada). Usaremos Python (pdfplumber ou pypdf) para extrair o texto de cada página e salvá-lo no PostgreSQL.
- **Reprodução:** Usar a **Web Speech API** (nativa do navegador). É gratuita e não requer API externa (Google/AWS).

- *Como funciona:* O usuário clica em "Ouvir". O Frontend pega o texto daquela página (vindo do banco de dados ou extraído na hora pelo PDF.js) e dita para o navegador: `speechSynthesis.speak()`.

4. Pesquisa de Palavra-chave

Viabilidade: Alta.

Solução Técnica:

- Não pesquise no arquivo PDF em tempo real (seria lento).
- **Indexação:** Ao fazer o upload do PDF no sistema administrativo, um script Python deve extrair todo o texto, página por página, e salvar no PostgreSQL.
- **Busca Full-Text:** O PostgreSQL possui recursos nativos poderosos de busca textual (Full Text Search com tsvector).
- **Fluxo:** O aluno digita a palavra -> Django busca no Postgres -> Retorna lista de páginas onde a palavra ocorre -> Aluno clica e vai para a página.

5. Layout Responsivo (Paisagem/Retrato - 1 ou 2 páginas)

Viabilidade: Alta.

Solução Técnica:

- Isso é resolvido 100% no Frontend (JavaScript + CSS).
- O visualizador deve ter um "modo spread" (página dupla).
- **Lógica:** O JS detecta a largura da tela (`window.innerWidth`).
 - Se mobile ou tablet portrait: Renderiza 1 canvas do PDF.
 - Se desktop ou tablet landscape: Renderiza 2 canvas lado a lado (Página N e N+1).

Tecnologias e Pacotes Recomendados

Backend (Python/Django)

1. **Django:** Framework principal.
2. **Django REST Framework (DRF):** Para criar a API que comunica o Frontend com o Banco (salvar página, buscar texto).
3. **PostgreSQL:** Banco de dados (obrigatório para busca textual eficiente).
4. **pdfplumber** ou **PyPDF2:** Para extrair texto do PDF para o banco de dados (para busca e TTS).

5. **Whitenoise:** Para servir arquivos estáticos de forma eficiente.

Frontend (Interface de Leitura)

1. **Mozilla PDF.js:** A biblioteca padrão da indústria para renderizar PDFs na web. É o que o Firefox usa.
 2. **HTML5/CSS3/JavaScript (ES6):** Para a lógica de virar página.
 3. **Bootstrap 5** ou **Tailwind:** Para a responsividade da interface (botões, menus).
-

Plano de Implementação

Aqui está um roteiro passo a passo para construir seu MVP (Produto Mínimo Viável).

Fase 1: Backend e Modelagem de Dados

1. **Configurar Django + PostgreSQL:**
 - Iniciar projeto Django.
 - Configurar conexão com Postgres.
2. **Criar Models:**
 - Documento: (título, arquivo_pdf, data_upload).
 - PaginaConteudo: (FK Documento, numero_pagina, texto_conteudo, vetor_busca). *Aqui armazenaremos o texto extraído.*
 - LeituraProgresso: (FK User, FK Documento, ultima_pagina_lida, data_atualizacao).
3. **Script de Processamento (Ingestão):**
 - Criar um signal ou método save() no modelo Documento. Quando o admin subir o PDF, o Python usa pdfplumber para ler página por página, extrair o texto e salvar na tabela PaginaConteudo.
 - Criar índice de busca (GinIndex) no campo de texto para a pesquisa ser instantânea.

Fase 2: API (Django REST Framework)

Criar os seguintes endpoints (URLs):

1. GET /api/documento/{id}/config: Retorna metadados (total de páginas, título).
2. GET /api/progresso/{doc_id}: Retorna a página onde o aluno parou.

3. POST /api/progresso/{doc_id}: Recebe o número da página atual e salva no banco.
4. GET /api/busca/?q=termo: Retorna as páginas que contêm o termo.
5. GET /api/pagina/{doc_id}/{num_pag}/texto: Retorna o texto puro daquela página (para o TTS).

Fase 3: Frontend (O Leitor)

Esta é a parte crítica para a experiência do usuário ("folhear").

1. Integração PDF.js:

- Criar uma página HTML com um `<canvas id="pdf-render">`.
- Carregar o documento via PDF.js: `pdfjsLib.getDocument(url)`.

2. Lógica de Renderização:

- Criar função `renderPage(num)`.
- Implementar botões "Anterior" e "Próximo".
- Adicionar áreas de toque (lado esquerdo da tela volta, lado direito avança) para mobile.

3. Responsividade (1 vs 2 páginas):

- Criar lógica:

```
if (window.innerWidth > 1024 && !isPortrait) {  
    // Renderiza Pagina X no Canvas 1 e X+1 no Canvas 2  
}  
else {  
    // Renderiza Pagina X no Canvas 1  
}
```

1. Sincronização de Progresso:

- Ao renderizar a página, chamar a API POST `/api/progresso` (com *debounce* para não chamar 100 vezes se o usuário folhear rápido. Chamar apenas após 2 segundos parado na página).

Fase 4: Recursos Avançados (TTS e Busca)

1. Implementar Busca:

- Criar um input de busca.
- Ao submeter, chamar API Django.

- Exibir lista de resultados. Ao clicar, chamar `renderPage(resultado.numero_pagina)`.

2. Implementar TTS:

- Adicionar botão "Ouvir Página".
- O JS pega o texto (via API ou extraído da camada de texto do PDF.js).
- Executa:

```
let utterance = new SpeechSynthesisUtterance(textoDaPagina);  
utterance.lang = 'pt-BR';  
window.speechSynthesis.speak(utterance);
```

E a parte que simula a ação de folhear uma página de livro. Isso é possível?

Sim, é perfeitamente possível e esse efeito é conhecido tecnicamente como **"Page Flip"** ou **"Turn.js effect"** (referência à biblioteca antiga que popularizou isso).

Para implementar isso num cenário moderno (sem depender do antigo jQuery) e integrado com o PDF.js (que renderiza o seu PDF de 22MB), a abordagem muda um pouco.

Aqui está como viabilizar o efeito de "folhear" realístico:

1. A Biblioteca Recomendada: StPageFlip

Não tente programar a física do papel virando do zero (cálculo de curvas de Bezier, sombras, gradientes). Use uma biblioteca pronta e leve.

Recomendo a **StPageFlip** (também conhecida como page-flip).

- **Por que esta?** É escrita em JavaScript puro (Vanilla JS), não tem dependências pesadas, suporta touch (dedo) e mouse, e lida bem com layout de uma página (retrato) ou duas páginas (paisagem).

2. O Desafio Técnico: PDF.js + Efeito Flip

O grande "pulo do gato" aqui é que você **não pode** converter o PDF inteiro de 22MB em imagens de uma vez para colocar no visualizador, senão o navegador vai travar por falta de memória RAM.

A Estratégia de Implementação:

1. **O Container:** Você cria o elemento HTML do "Livro".
2. **O Renderizador (PDF.js):** Ele funciona "sob demanda".

3. A Integração:

- O usuário abre o app.
- O App carrega apenas a Capa e a Página 1 usando PDF.js.
- Quando o usuário começa a arrastar a borda da página (evento flip), o JavaScript intercepta isso e manda o PDF.js renderizar a **próxima página** num <canvas> escondido.
- A biblioteca StPageFlip pega esse canvas e aplica a distorção visual (a curva do papel).

3. Exemplo de Lógica (Frontend)

Imagine que você terá um HTML assim:

```
<!-- Container do Livro -->

<div id="book">

  <!-- As páginas serão injetadas aqui dinamicamente -->

  <div class="my-page" data-density="hard"> <!-- Capa Dura -->

    <canvas id="canvas_capa"></canvas>

  </div>

  <div class="my-page">

    <canvas id="canvas_pag1"></canvas>

  </div>

  <!-- ... -->

</div>
```

E no JavaScript:

```
import { PageFlip } from 'page-flip';

// Configuração do Flip

const pageFlip = new PageFlip(document.getElementById('book'), {

  width: 400, // largura base

  height: 600, // altura base

  showCover: true,

  maxShadowOpacity: 0.5 // Sombra realista quando a página curva
```

```
});
```

```
// Integração com PDF.js
```

```
function renderPdfPageToCanvas(pdfDoc, pageNum, canvasId) {  
    pdfDoc.getPage(pageNum).then(function(page) {  
        // Renderiza o PDF dentro do Canvas da página do livro  
        var viewport = page.getViewport({scale: 1.5});  
        var canvas = document.getElementById(canvasId);  
        var context = canvas.getContext('2d');  
        // ... código padrão de renderização do PDF.js ...  
    });  
}
```

```
// Carregar páginas dinamicamente (Lazy Loading)
```

```
pageFlip.on('flip', (e) => {  
    // O usuário virou a página.  
    // Vamos carregar a próxima página do PDF que ainda não foi renderizada  
    // para economizar memória.  
    let proximaPagina = e.data + 2;  
    renderPdfPageToCanvas(meuPdfDoc, proximaPagina, 'id_novo_canvas');  
});
```

4. Cuidados Importantes (UX/UI)

Para o seu projeto, considere estes pontos cruciais sobre o efeito de folhear:

1. Mobile (Smartphones):

- **Problema:** Em telas muito pequenas (celular em pé), o efeito de folhear (pegar na pontinha da página e arrastar) é muitas vezes frustrante e compete com o gesto de "scroll" do sistema. Além disso, ocupa pixels preciosos com a animação 3D.
- **Recomendação:** Detecte se é mobile.

- *Desktop/Tablet*: Ative o efeito Page Flip.
- *Smartphone*: Desative o efeito e use um "Slider" simples (arrastar para o lado faz a página deslizar chapada, sem curvar). É mais usável para estudar.

2. Performance:

- Como seu PDF tem 22MB, pode ter muitas páginas. Use o conceito de "Janela Deslizante". Mantenha no DOM (na tela) apenas a página atual, a anterior e a próxima. Apague as páginas distantes da memória do navegador, senão o app ficará lento após folhear 50 páginas.

3. Zoom:

- O efeito de folhear complica o Zoom (pinça). Geralmente, quando o usuário dá zoom para ler um texto pequeno, o efeito de folhear deve ser bloqueado temporariamente para que ele possa arrastar o documento ampliado sem virar a página acidentalmente.

Resumo da Viabilidade

É Viável e Visualmente Impressionante.

A combinação tecnológica seria:

- **Backend**: Python/Django servindo o arquivo (Range Requests).
- **Renderização**: PDF.js (converte PDF em imagem/canvas).
- **Animação**: Biblioteca StPageFlip (pega o canvas e aplica a física de papel).