ChatGPT

Proposta Reformulada de Implementação do Portal FACODI (MVP Estático + Supabase)

Visão Geral do Projeto

FACODI (Faculdade Comunitária Digital) é uma plataforma de educação à distância inspirada nos currículos da UALG, que organiza conteúdo acadêmico em cursos, unidades curriculares (UCs) e playlists do YouTube [1]. O objetivo principal é **democratizar o acesso ao ensino superior** fornecendo trilhas de estudo gratuitas e open-source, com **conteúdo versionado em Markdown** e sincronizado com um banco de dados PostgreSQL (via Supabase) [2]. Nesta proposta reformulada, adotaremos uma arquitetura 100% **estática no front-end**, utilizando Hugo (gerador de site estático) para construir as páginas, enquanto todo o conteúdo dinâmico (descrições, vídeos, documentos) será carregado no navegador via chamadas à **API do Supabase**. Com isso, eliminamos a necessidade de um framework complexo (como Next.js) no MVP e mantemos a experiência de desenvolvimento o mais simples possível, sem abrir mão da elegância e organização.

Arquitetura Estática e Tecnologias do MVP

- **Frontend Estático (Hugo)**: Usaremos o Hugo para gerar um site estático a partir de arquivos Markdown. O Hugo permitirá compilar o conteúdo das páginas localmente (ou em CI) em HTML, servindo-o via seu servidor embutido durante o desenvolvimento e podendo ser hospedado em qualquer servidor web para produção. A estrutura de pastas do projeto já segue o padrão do Hugo (com diretórios `content/`, `layouts/`, `static/`, etc.), integrando o tema Doks para estilização e funcionalidades adicionais [3] [4]. Todo o **site "shell" (menus, navegação, estrutura das páginas)** estará contido nesses arquivos estáticos gerados, garantindo rapidez e simplicidade no deploy.

- **Backend como Serviço (Supabase)**: O Supabase será usado como backend *headless*. Ele provê:

- Um banco **PostgreSQL** gerenciado, onde armazenaremos todo o conteúdo acadêmico (cursos, UCs, tópicos, playlists, etc.) em tabelas – todas com nomes e colunas em **inglês** para padronização.
- APIs prontas (RESTful e client JS) para consultar esses dados diretamente do front-end estático. No navegador usaremos o cliente JavaScript do Supabase com a **chave anônima (anon key)** para ler os dados públicos.

- Possibilidade futura de autenticação e armazenamento de progresso (via Supabase Auth), embora **nesta fase MVP não haverá criação/autenticação de usuários**, mantendo o escopo inicial focado apenas em conteúdo público.

- **Integração Frontend-Backend via Fetch**: Como o site será estático, **todas as chamadas de conteúdo serão realizadas no lado do cliente** (browser) através da API do Supabase. Isso significa que as páginas HTML geradas pelo Hugo conterão apenas a estrutura básica e o código JavaScript necessário para buscar os dados e inserir o conteúdo. Por exemplo:

- Uma página de curso será entregue inicialmente com layout estático (cabecalho, rodapé, placeholders, etc.) e um identificador do curso; um script JavaScript irá, ao carregar a página, fazer um fetch ao Supabase (via `supabase.from('course').select(...)` ou similar) para obter detalhes atualizados daquele curso (nome, descrição, UCs relacionadas, etc.) e então renderizá-los no DOM.
- Da mesma forma, uma página de UC irá buscar via API o conteúdo da unidade curricular (ementa, pré-requisitos, resultados de aprendizagem, etc.) e também listas de vídeos (playlists do YouTube) ou tópicos associados.
- Esse modelo **client-side fetch** garante que o conteúdo exibido esteja sempre sincronizado com o banco de dados **sem precisar reconstruir o site estático a cada atualização de conteúdo** (basta que o conteúdo no Supabase seja atualizado). Além disso, simplifica a arquitetura, pois não há camada de servidor customizada – o browser comunica-se diretamente com o Supabase usando a anon key segura.

Obs.: Uma consequência dessa abordagem é que conteúdo textual carregado via JavaScript pode não ser imediatamente indexável por mecanismos de busca. No futuro, podemos mitigar isso gerando algumas partes do conteúdo estaticamente para SEO ou utilizando prerendering. No MVP, porém, priorizamos simplicidade e sincronização em tempo real sobre SEO.

Estrutura de Conteúdo em Markdown no Repositório

Todo o conteúdo acadêmico permanece versionado no repositório Git, em arquivos **Markdown** organizados de forma hierárquica conforme o currículo. Esta é a *fonte de verdade* (Single Source of Truth) do conteúdo editorial [5] – quaisquer alterações no conteúdo devem ser feitas nos `.md` e sincronizadas para o banco. A estrutura de pastas já proposta no repositório segue este modelo organizado [6] [7]:

- `content/courses/<curso>/<versão>/index.md` – Arquivo Markdown principal de um **Curso** (por exemplo, um curso `LESTI` versão `2024-2025`). Contém no frontmatter YAML informações como código do curso, título, grau (bacharelado, mestrado), total de ECTS, semestres, versão do plano, instituição, idioma e um resumo [8] [9]. O corpo do arquivo traz a descrição/apresentação do curso. Esse arquivo representa a página do curso no site.
- `content/courses/<curso>/<versão>/uc/<codigo_UC>/index.md` – Arquivo Markdown de uma **Unidade Curricular (UC)** específica dentro de um curso e plano de estudos. O frontmatter inclui campos como código da UC, título, descrição, ECTS, semestre, idioma, pré-requisitos (lista de códigos de UCs), resultados de aprendizagem (lista de itens texto), playlists do YouTube (lista de IDs de playlist com prioridades) e tópicos associados (lista de *slugs* de tópicos) [10] [11]. O corpo em Markdown contém a ementa detalhada, metodologia e bibliografia da UC. Este arquivo corresponde à página de detalhes da UC.
- `content/courses/<curso>/<versão>/uc/<codigo_UC>/<topico>.md` – Arquivos Markdown para **Tópicos** específicos dentro de uma UC (por exemplo, um tópico `estruturas-de-dados` dentro de `LESTI-ALG1`). No frontmatter, definem-se o slug do tópico, título (nome do tópico), um sumário ou descrição breve, eventualmente playlists do YouTube específicas do tópico, e tags relacionadas [12]. O corpo em Markdown traz o conteúdo explicativo do tópico (por exemplo, texto introdutório, exemplos etc.). Cada um desses arquivos será uma página ou seção dedicada a um tópico no site.

Além dos conteúdos, o repositório contém também arquivos de configuração e scripts para apoio: - **Menus e Navegação**: A estrutura de navegação (menus, sidebar) do site será definida no próprio código do Hugo. Como os cursos, UCs e tópicos já têm uma organização hierárquica no conteúdo, podemos aproveitar isso para gerar automaticamente menus. Por exemplo, podemos configurar o

Hugo para listar todos os cursos disponíveis na página inicial ou no menu principal. Cada curso, por sua vez, listará suas UCs, e dentro da página de uma UC pode-se listar os tópicos. Essa "carcaça" de navegação fica pré-definida no site estático (por meio de templates ou dados de menu do Hugo), garantindo uma **estrutura padrão para cada tipo de entidade** (curso, UC, tópico) já prevista no código base. Assim, temos páginas e seções prontas para cada entidade, e o JavaScript sabe onde injetar os dados correspondentes de forma consistente. - **Frontmatter YAML como Metadados**: Os campos definidos no frontmatter dos `.md` funcionam como metadados estruturados que serão utilizados tanto pelo Hugo (para eventualmente exibir títulos, ordenações, etc.) quanto pelo processo de sincronização com o banco. Todos os nomes de campos estão em inglês (por ex.: `title`, `description`, `language`), e devem permanecer assim para consistência com as colunas do banco de dados. Isso atende ao requisito de padronização de nomenclatura em inglês para tabelas e colunas. - **Arquivos Markdown no repositório vs. conteúdo dinâmico**: Embora no MVP o site carregue o conteúdo via fetch no cliente, **todos os arquivos Markdown são mantidos no repositório** para facilitar edição colaborativa, revisão via PR e histórico versionado. Ou seja, editamos o Markdown e o banco de dados será atualizado para refletir essas mudanças – mas a renderização ao usuário final virá do banco via API. Este arranjo nos dá o melhor dos dois mundos: edição simples e rastreável (no git) e entrega dinâmica atualizada.

Sincronização do Conteúdo com o Banco de Dados (Supabase)

Para manter o conteúdo do banco de dados alinhado com os arquivos Markdown (e vice-versa), implementaremos um fluxo de **sincronização automatizada**: - Sempre que mudanças forem feitas nos arquivos `content/` (por exemplo, adição de um novo curso ou atualização de uma ementa de UC) e essas mudanças forem enviadas ao repositório (push/merge na branch principal), um **GitHub Action** irá acionar um script de sync. O repositório já inclui configurações de CI para isso: o workflow `sync-md-to-supabase.yml` é disparado em push nos arquivos de conteúdo ou config [13]. Esse workflow executa o script (via `npm run sync:md:db`) com as credenciais do Supabase configuradas nos secrets do GitHub [14], populando/atualizando as tabelas conforme as mudanças nos `.md`. - O script de sincronização utiliza o cliente do Supabase no modo administrador (usando a **Service Key**, que tem permissão de escrita). A configuração `config/sync.config.json` mapeia os campos do frontmatter para colunas específicas no Postgres [15] e define onde gravar o conteúdo Markdown completo. Por exemplo, o campo `code` de uma UC no frontmatter é mapeado para a coluna `catalog.uc.code` no banco; o título da UC vai para `catalog.uc.name`; o array de `prerequisites` gera entradas relacionadas apropriadas; e o **corpo** do Markdown da UC é salvo em uma coluna de conteúdo (por exemplo, `catalog.uc_content.content_md`) [16] [17]. Esse mapeamento assegura que todos os dados relevantes do Markdown estejam refletidos no banco de forma estruturada e normalizada [18]. - O mecanismo de **Single Source of Truth** é mantido: os arquivos `.md` são a fonte primária, e o banco é essencialmente uma projeção dos dados para fins de consulta e indexação [5]. Dessa forma, evitamos discrepâncias – o que está no portal sempre veio de um conteúdo versionado. - Para garantir qualidade, há também um workflow de **validação de Markdown** (`validate-md.yml`) que pode executar linter ou verificações (por exemplo, campos obrigatórios no frontmatter como `code`, `plan_version` preenchidos) e impedir merges caso algo esteja inconsistente [5]. Isso protege a integridade do conteúdo antes mesmo da sincronização. - Embora não seja foco imediato do MVP, a arquitetura também prevê **sincronização reversa**: ações feitas no futuro portal (como aprovar uma nova playlist de vídeos via interface) deverão refletir de volta nos arquivos Markdown, possivelmente abrindo automaticamente um Pull Request no GitHub com aquelas alterações [5]. Isso garante que o repositório continue sendo a base fiel de todo o conteúdo, mesmo que edições ocorram fora dele.

Em resumo, sempre que o time editorial adicionar ou modificar conteúdo (via Git) o portal refletirá isso após o pipeline de sincronização. E, no futuro, se o portal permitir editar conteúdo, tais mudanças serão gravadas no banco e também retornadas ao repositório. Para configurar tudo isso inicialmente, precisamos definir as credenciais do Supabase como variáveis de ambiente e nos Secrets do repositório: - No ambiente de desenvolvimento/local, criar um arquivo `.env` ou `.env.local` contendo `SUPABASE_URL`, `SUPABASE_ANON_KEY` e `SUPABASE_SERVICE_KEY` (valores fornecidos para o projeto) [19]. Estes serão usados pelos scripts e pelo site para se conectar ao Supabase. - No GitHub, definir os *Actions Secrets* `SUPABASE_URL`, `SUPABASE_ANON_KEY` e `SUPABASE_SERVICE_KEY` com os mesmos valores [20], para que os workflows de CI possam usar as credenciais com segurança. *(Obs.: A chave de serviço é usada apenas no contexto de CI/servidor. No front-end do site estático, usaremos somente a anon key, que tem permissões restritas de leitura, evitando expor quaisquer credenciais sensíveis.)*

Modelagem do Banco de Dados (Tabelas em Inglês)

No Supabase (Postgres) iremos implementar o esquema de banco de dados seguindo a lógica do conteúdo, com nomenclatura em inglês para todas as tabelas e colunas. Podemos nos basear no mapeamento definido no config para criar as tabelas. A seguir uma proposta de estrutura (respeitando o design já delineado, mas traduzindo para nomes consistentes em inglês):

- **Schema** `catalog`: agrupa as entidades de cursos e UCs (unidades curriculares):
- Tabela `catalog.course` – armazena os cursos. Campos principais:
  - `code` (texto, PK): código do curso (ex: "LESTI").
  - `name` (texto): nome/título do curso (ex: "Licenciatura em ...").
  - `degree` (texto): grau acadêmico (ex: "bachelor", "master", etc.).
  - `ects_total` (inteiro): total de créditos ECTS do curso.
  - `duration_semesters` (inteiro): duração em semestres.
  - `plan_version` (texto): versão do plano curricular (ex: "2024/2025").
  - `institution` e `school` (texto): identificação da instituição e escola/faculdade.
  - `language` (texto): idioma principal do curso (ex: "pt").
  - `summary` (texto): resumo/descrição breve do curso.
  (Obs.: O conteúdo completo de apresentação do curso, caso extenso, pode ser armazenado separadamente – ver tabela `course_content`*.)*
- Tabela `catalog.course_content` – opcional, para armazenar o corpo em Markdown completo da apresentação do curso. Campos:
  - `course_code` (fk para `catalog.course.code`).
  - `content_md` (textão): conteúdo Markdown bruto do curso [21].
  (Essa separação evita colocar campos grandes de texto na tabela principal de curso, mantendo-a leve para consultas).
- Tabela `catalog.uc` – armazena as UCs (disciplinas). Campos principais:
  - `code` (texto, PK composto ou único): código da UC (ex: "LESTI-ALG1"). Este código incorpora o curso, mas podemos incluir também um campo separado `course_code` para referência explícita ao curso pai (ex: "LESTI"), e possivelmente `plan_version` se quisermos distinguir UCs por versões de plano.
  - `name` (texto): nome da UC (ex: "Algoritmos e Estruturas de Dados I").
  - `description` (texto): descrição/ementa breve.
  - `ects` (inteiro): créditos ECTS da UC.
  - `semester` (inteiro): semestre em que é lecionada.
  - `language` (texto): idioma da UC.

    - `prerequisites` (array de texto ou tabela relacionada): pré-requisitos (lista de códigos de UC que são pré-requisito). Poderemos armazenar isso como um array de strings na própria tabela `uc` ou, para normalização, ter uma tabela associativa para pré-requisitos. Como é MVP, um array de texto pode servir, mas a abordagem padronizada seria ter, por exemplo, uma tabela `catalog.uc_prerequisite` com colunas `uc_code` e `prereq_uc_code` registrando cada relação.
- Tabela `catalog.uc_content` – armazena o conteúdo Markdown completo da UC (ementa detalhada, etc.). Campos:
    - `uc_code` (fk para `catalog.uc.code`).
    - `content_md` (textão): conteúdo Markdown da UC [22].
- Tabela `catalog.uc_learning_outcome` – guarda os **resultados de aprendizagem** de cada UC de forma estruturada. Cada item da lista `learning_outcomes` do frontmatter gera uma entrada aqui. Campos:
    - `uc_code` (fk para `catalog.uc.code`).
    - `outcome` (texto): descrição do resultado de aprendizagem (por exemplo: "Compreender complexidade de algoritmos" [23]).
    - *Opcional:* um campo de ordem (para manter a sequência listada no Markdown).

- *(Obs.: As tabelas `catalog.uc` e `catalog.course` estão conceitualmente ligadas. Se necessário, podemos criar foreign keys entre `uc.course_code` e `course.code` para integridade referencial.)*

- **Schema** `subjects`: agrupa as entidades de tópicos e taxonomias:

- Tabela `subjects.topic` – armazena os **tópicos** (assuntos) que podem ser associados às UCs. Campos:
    - `slug` (texto, PK): identificador do tópico (ex: "estruturas-de-dados").
    - `name` (texto): nome/título do tópico (ex: "Estruturas de Dados").
    - `summary` (texto): breve resumo/descrição do tópico (se fornecido).
    (Um mesmo tópico pode teoricamente aparecer em várias UCs, então ele é entidade separada globalmente. Caso quiséssemos permitir tópicos duplicados por curso, poderíamos incluir contexto no slug, mas a proposta assume tópicos únicos por slug.)
- Tabela `subjects.topic_content` – corpo em Markdown do tópico (conteúdo explicativo). Campos:
    - `topic_slug` (fk para `subjects.topic.slug`).
    - `content_md` (textão): conteúdo Markdown do tópico.

- Tabela `subjects.topic_tag` – armazena **tags** associadas a tópicos (palavras-chave). Cada tag listada no frontmatter de um tópico gera uma entrada aqui. Campos:

    - `topic_slug` (fk para `subjects.topic.slug`).
    - `tag` (texto): tag/palavra-chave (ex: "listas", "pilhas" [24]).
    (Alternativamente, poderíamos ter uma tabela de tags únicas e uma tabela de relacionamento muitos-para-muitos, mas dado o escopo, salvar diretamente as tags por tópico é aceitável.)

- **Schema** `mapping`: contém tabelas de relacionamentos (*mapeamentos*) entre as entidades e outros recursos (como playlists de vídeos e associação UC<->Tópico):

- Tabela `mapping.uc_topic` – relaciona quais **tópicos** pertencem a qual UC (definido pelo array `topics` no frontmatter da UC). Campos:
    - `uc_code` (fk para `catalog.uc.code`).

    - `topic_slug` (fk para `subjects.topic.slug`).
      (Cada slug listado em `topics` de uma UC gera uma entrada aqui [25].)
- Tabela `mapping.uc_playlist` – relaciona **playlists de YouTube** a uma UC. Uma UC pode ter uma ou mais playlists de vídeos (listadas com prioridade no frontmatter). Campos:
    - `uc_code` (fk para `catalog.uc.code`).
    - `playlist_id` (texto): identificador da playlist do YouTube (ex: `"PL_abc123"` [26]).
    - `priority` (inteiro): ordem/prioridade de exibição (como especificado no frontmatter).
- Tabela `mapping.topic_playlist` – analogamente, relaciona **playlists** a um determinado **tópico**. Campos:
    - `topic_slug` (fk para `subjects.topic.slug`).
    - `playlist_id` (texto): ID da playlist do YouTube referente ao tópico.
    - `priority` (inteiro): ordem de prioridade.
      (Isso permite, por exemplo, que um tópico possua uma playlist específica de vídeos explicando aquele assunto, independente das playlists gerais da UC.)

Todas essas tabelas e colunas serão nomeadas em inglês, conforme descrito, para consistência. Os arquivos `.sql` antigos presentes no repositório podem ser descartados ou usados apenas como referência inicial, pois vamos recriar a estrutura do zero já com os nomes padronizados e alinhados ao conteúdo do Markdown. **Importante**: após criar essas tabelas no Supabase, habilitar as políticas de acesso (RLS) apropriadas. Para o MVP, podemos tornar todas essas tabelas legíveis publicamente (selecionar permissões para role `anon`), já que o conteúdo é aberto. A escrita ficará restrita à service key (via sincronização ou painel admin do Supabase). Assim, o front-end conseguirá fazer `select` nos dados sem exigir autenticação de usuário.

Implementação do Frontend Estático com Hugo

Com a estrutura de conteúdo estabelecida e o banco de dados sincronizado, o próximo passo é configurar o front-end Hugo para apresentar esses dados. Os principais pontos de implementação no front-end são:

Templates de Páginas e Navegação

O Hugo permitirá definir layouts/templates para cada tipo de página: - **Página inicial**: Pode exibir uma lista de todos os cursos disponíveis. Como o site é estático, podemos gerar essa lista de forma fixa com base nos cursos presentes no conteúdo (ou até mesmo codificar no template). Uma abordagem é usar os arquivos em `content/courses/` para construir essa lista. Alternativamente, poderíamos deixá-la vazia e preencher via fetch do Supabase (ex: buscar todos os cursos ordenados por nome). Porém, como a existência de cursos e seus títulos já está no conteúdo, pré-renderizar a lista na home pode ser mais rápido e amigável ao usuário. Por simplicidade, poderemos criar manualmente uma seção no template inicial que itera sobre os cursos (Hugo consegue iterar sobre páginas em `content/courses` se estruturado adequadamente) ou inserir os links dos cursos diretamente. Cada item da lista da página inicial linkará para a página do curso correspondente. - **Página de Curso**: Correspondente a `content/courses/<code>/<versao>/index.md`. Hugo gerará um HTML estático para cada curso, onde colocaremos o esqueleto da página – por exemplo, título do curso (que podemos inserir estaticamente do frontmatter), e se quisermos, alguns detalhes básicos como duração, ECTS etc. Entretanto, seguindo a diretriz de carregar conteúdo via API, poderemos colocar apenas o título e estrutura fixa, e usar JavaScript para preencher os demais detalhes dinâmicos (descrição completa, lista de UCs, etc.). Nesta página, também teremos seções ou links para as **Unidades Curriculares** do curso. Podemos aproveitar a estrutura de pastas: Hugo pode automaticamente saber as páginas filhas (as UCs) de cada curso e gerar links para elas, ou incluir um menu lateral com essas UCs. Porém, para manter simplicidade, podemos inserir um container vazio onde o JS irá listar as UCs obtidas via

Supabase (ex.: `select * from catalog.uc where course_code = 'LESTI'`). Uma alternativa intermediária: exibir estaticamente a lista de UCs (já que elas são conhecidas via conteúdo), mas ainda assim usar fetch para detalhes adicionais se necessário. De qualquer forma, a navegação (links) para as UCs de um curso deve estar presente. - **Página de UC (Unidade Curricular)**: Derivada de `content/courses/<curso>/<versao>/uc/<uc_code>/index.md`. Será gerada uma página estática por UC. Nela, podemos colocar estaticamente o título da UC (para que o usuário veja algo imediatamente e para SEO) e talvez alguns meta-dados básicos, mas deixaremos o grosso do conteúdo para ser carregado via JS. Por exemplo, a ementa detalhada, resultados de aprendizagem e playlists de vídeos seriam buscados no Supabase: - No carregamento da página UC, um script JavaScript identificará o código da UC (podemos inseri-lo em um `data-attribute` no HTML ou derivar da URL) e fará requisições ao Supabase: buscar na tabela `catalog.uc` os dados da UC (description, ects, semester, etc.), buscar em `catalog.uc_content` o conteúdo Markdown (que poderia ser renderizado em HTML no cliente – possivelmente usando alguma biblioteca Markdown -> HTML para exibir formatado), buscar em `catalog.uc_learning_outcome` a lista de resultados de aprendizagem, e buscar em `mapping.uc_playlist` as playlists de vídeo associadas. Com essas respostas, o JS preencherá as seções adequadas da página: descrição da disciplina, tabela de informações, lista de resultados de aprendizagem (pode ser um `<ul>`), e incorporar um player ou links para as playlists do YouTube. *(Observação: poderemos usar a API do YouTube ou simplesmente links embed dos vídeos; porém, dado que provavelmente teremos uma integração via Supabase Edge Functions para atualizar detalhes das playlists* [27] *, é provável que também armazenemos no banco informações atualizadas sobre vídeos, que poderíamos mostrar. No MVP, podemos apenas linkar/incorporar o playlist embed por ID.)* - A página de UC também deve apresentar os **Tópicos** relacionados. Conforme o frontmatter e tabelas, os tópicos vinculados a essa UC podem ser obtidos consultando `mapping.uc_topic` (todas as entradas onde `uc_code` = código da UC). Isso retornará uma lista de slugs de tópicos, que podemos então usar para obter os detalhes de cada tópico da tabela `subjects.topic` (ou diretamente fazer um join server-side via Supabase RPC, mas para simplicidade vários requests ou um request encadeado podem servir). O JS então pode gerar links na página para cada tópico (por exemplo, uma lista "Tópicos desta UC: Estruturas de Dados, Análise de Algoritmos..."). Esses links apontarão para as respectivas páginas de tópico. - **Página de Tópico**: a partir de `content/courses/<curso>/<versao>/uc/<uc_code>/<topic>.md`. Hugo gerará uma página por tópico (provavelmente com URL algo como `/courses/LESTI/2024-2025/uc/LESTI-ALG1/estruturas-de-dados/`). Essa página, similarmente, terá um template estático básico: por exemplo, um cabeçalho indicando o nome do tópico e a qual UC ele pertence (podemos inserir isso estaticamente se Hugo tem contexto do pai, ou incluir via JS). Ao carregar, o JS irá usar o slug do tópico para consultar o Supabase: obter de `subjects.topic` o nome e descrição do tópico, de `subjects.topic_content` o conteúdo Markdown (que será convertido para HTML no browser), e possivelmente listar as playlists ou materiais associados (consultando `mapping.topic_playlist` para playlists específicas do tópico, e `subjects.topic_tag` para exibir as tags associadas). Dessa forma, o conteúdo completo do tópico (texto e vídeos) é inserido dinamicamente. A página de tópico também deve permitir navegação de volta à UC e curso pai (links de hierarquia).

- **Elementos Comuns e Layout**: Utilizaremos os recursos do Hugo (layouts padrão, partials) para fatorar elementos comuns (como cabeçalho, rodapé, menu lateral). Por exemplo, podemos ter um partial de cabeçalho com o nome do projeto e menu principal de cursos. No menu principal poderemos listar todos os cursos (estaticamente ou via Hugo range sobre `content/courses`). Também incluir um seletor de idioma se já formos preparar o multi-idioma (por enquanto PT, mas com possibilidade de EN no futuro). O rodapé pode conter créditos, links etc. Esses elementos podem ser estilizados conforme a identidade desejada. *Vale destacar:* o repositório integrou o tema **Doks** (ou similar) que já traz um visual limpo e vários componentes, então podemos customizá-lo para atender nossas necessidades visuais mantendo a elegância e responsividade prontas.

Carregamento de Conteúdo via Supabase (Frontend)

Para implementar o fetch de conteúdo no navegador de forma organizada e **padronizada para cada entidade**, criaremos scripts JavaScript modulares: - Um arquivo JS principal (incluído no layout base do Hugo) que inicializa o cliente do Supabase usando `SUPABASE_URL` e `SUPABASE_ANON_KEY`. Essas variáveis podem ser inseridas no HTML gerado via Hugo através de variáveis de ambiente ou configurações. (Por exemplo, poderíamos definir no `config.toml` do Hugo valores para supabase URL e key, e usá-los em um script tag.) O Supabase fornece a função `createClient(url, anonKey)` para inicializar o cliente. *Importante:* usaremos **apenas a anon key** no front-end, garantindo que somente operações permitidas aos usuários anônimos ocorram. - Módulos/funções JS por tipo de página: - `loadCoursePage(courseCode, planVersion)`: responsável por buscar no Supabase os dados do curso (`from('course').select(...).eq('code', courseCode).eq('plan_version', planVersion)`) e suas UCs associadas (`from('uc').select(...).eq('course_code', courseCode)`), então atualizar o DOM da página do curso (por exemplo, inserindo a descrição do curso, populando uma lista de UCs com links). Poderemos também ordenar as UCs por semestre antes de exibir. Se algumas informações básicas do curso (nome, etc.) já estiverem no HTML estático, esse script pode complementar dados faltantes. - `loadUCPage(ucCode)`: fará fetch de `catalog.uc` por código (único) para obter detalhes da UC, de `catalog.uc_content` para o conteúdo (que poderá ser renderizado como HTML usando uma lib Markdown->HTML ou uma simples função se o conteúdo não tiver muita formatação complexa), de `catalog.uc_learning_outcome` para resultados de aprendizagem (construindo uma listagem), de `mapping.uc_playlist` para montar embeds ou links de playlists (podemos inserir um iframe embed do YouTube playlist iframes for each playlist ID, or list links), e de `mapping.uc_topic` + `subjects.topic` para obter os tópicos relacionados e exibir seus nomes como links. Esse script então preenche os elementos da página UC correspondentes. - `loadTopicPage(topicSlug)`: consulta `subjects.topic` (por slug) para dados do tópico, `subjects.topic_content` para conteúdo (Markdown -> HTML), `mapping.topic_playlist` para eventuais playlists do tópico (podendo exibir um player ou links), e `subjects.topic_tag` para obter tags e exibí-las (talvez como chips ou uma simples lista). Insere tudo nos lugares adequados do HTML. Também pode mostrar um link de volta para a UC relacionada; para isso, talvez tenhamos passado o contexto da UC de origem no link ou poderíamos, via Supabase, inferir em que UCs aquele tópico aparece (consultando `mapping.uc_topic` por topic_slug). Se for necessário, podemos exibir "Este tópico aparece na UC X do curso Y" utilizando esses dados. - **Padronização das Chamadas**: Ao projetar essas funções de carregamento, manteremos consistência nos nomes de tabelas e colunas (em inglês, conforme criado). Por exemplo, sempre buscaremos `name`, `description` como colunas padrão, etc. Isso facilita o reuso. Além disso, todas as funções utilizarão o mesmo cliente Supabase inicializado centralmente. Podemos também implementar tratamento de erros unificado (ex.: exibir mensagem se algum conteúdo não carregar). - **Performance e Experiência**: Como o conteúdo será carregado via rede ao abrir a página, devemos cuidar para que o usuário veja um *feedback* durante o carregamento (por exemplo, um spinner ou texto "carregando..."). Entretanto, dado que o site é estático e leve, os fetches ao Supabase (que está em nuvem) tendem a ser rápidos. Podemos também considerar usar a opção de *caching* do lado do cliente (Supabase JS permite caching básico ou a gente mesmo guardar em localStorage se necessário). Porém, inicialmente, a simplicidade é chave: requisitar e renderizar diretamente. - **Exemplo de Integração**: Para deixar claro, imagine a página `LESTI-ALG1` (UC de Algoritmos I). O HTML estático entregue ao navegador conterá o cabeçalho com "Algoritmos e Estruturas de Dados I" e placeholders para "Descrição", "ECTS", "Semestre", "Resultados de Aprendizagem", etc., possivelmente como `<div id="description">...</div>`, `<ul id="outcomes">...</ul>` e assim por diante. Assim que o script `loadUCPage("LESTI-ALG1")` rodar, ele fará, por exemplo:

```
const { data: uc } = await supabase.from('uc').select('name, description,
ects, semester, language').eq('code','LESTI-ALG1').single();
document.getElementById('description').innerText = uc.description;
document.getElementById('ects').innerText = uc.ects + ' ECTS';
...
```

E assim sucessivamente para preencher cada parte. Para o conteúdo em Markdown:

```
const { data: ucContent } = await
supabase.from('uc_content').select('content_md').eq('uc_code','LESTI-
ALG1').single();
// Convert Markdown to HTML (could use a library like marked.js or a simple
converter if needed)
document.getElementById('content').innerHTML =
markdownToHtml(ucContent.content_md);
```

Para playlists:

```
const { data: playlists } = await
supabase.from('uc_playlist').select('*').eq('uc_code','LESTI-
ALG1').order('priority');
playlists.forEach(pl => {
    // create iframe embed or link using pl.playlist_id
});
```

E assim por diante. **Todas as consultas usam nomes de colunas em inglês exatamente como definidas no banco**, garantindo aderência ao esquema padronizado.

Internacionalização (i18n)

Embora no MVP o foco seja conteúdo em Português, a estrutura já suporta múltiplos idiomas. O Hugo facilita isso mantendo, por exemplo, diretórios de conteúdo separados por idioma (`content/pt/`, `content/en/`, etc.) ou utilizando chaves de tradução. No nosso repositório, há referência a um arquivo `config/i18n.json` para sinalizar idiomas disponíveis [28]. Planejamos ativar PT e EN inicialmente (e talvez ES, FR conforme mencionado [29]). Para implementar: - Mantenha o atributo `language` no frontmatter de cursos/UCs/tópicos para indicar o idioma do conteúdo [30] [31]. - Separe conteúdos traduzidos em caminhos diferentes no repositório, se aplicável (por ex., poderíamos ter `content/en/courses/...` para versões em inglês dos mesmos cursos, ou decidir que certas UCs terão conteúdo apenas em PT no MVP). - Configure o Hugo para ter *multilingual mode*, assim ele gerará sites/páginas para cada idioma. No MVP, se só PT tiver conteúdo, podemos deixar EN desativado, mas já prevemos a expansão. - No front-end, o supabase também armazenará o conteúdo multilíngue (provavelmente a coluna `language` ajuda a filtrar). Por exemplo, se no futuro tivermos UCs em PT e EN, as tabelas podem conter ambos e a aplicação filtrará conforme o idioma atual selecionado. Como padrão, carregaremos apenas conteúdo `language = 'pt'` até termos traduções. - Os textos da interface (menus, botões) podem ser externalizados em arquivos de idioma (Hugo permite usar dicionários de i18n). Focaremos em PT por enquanto, mas deixar preparado.

Configurações Iniciais e Deploy

Para viabilizar tudo acima, as seguintes ações/tarefas iniciais devem ser realizadas pelo responsável (ou agente) na configuração do repositório e do Supabase:

1. **Configurar o Hugo no projeto**: Certificar-se que o Hugo está instalado e funcionando com o tema Doks (ver dependências no `package.json`). Ajustar configurações básicas em `config/_default/config.toml` ou similar: título do site, idiomas, possivelmente desabilitar funcionalidades do tema que não sejam necessárias inicialmente. Verificar se o comando de build (`hugo` ou via npm script) está ok e se gera os arquivos estáticos esperados. Também ajustar o baseURL se necessário para o ambiente de produção.
2. **Remover dependências do Next.js (se ainda existirem)**: Como optamos por não usar Next.js no MVP, qualquer código referente a ele pode ser limpo. Por exemplo, se há um pacote `web` no monorepo com uma app Next, ele não será mais utilizado. O foco será no conteúdo (`facodi-docs`) e possivelmente renomear esse pacote se for o caso. O readme mencionava comandos de Next (agora obsoletos) – podemos atualizá-lo para refletir o novo fluxo de desenvolvimento (por exemplo: usar `hugo serve` para rodar local, em vez de `pnpm dev --filter=web`).
3. **Criar as tabelas no Supabase**: Usando a interface do Supabase ou scripts SQL, criar os schemas e tabelas conforme descrito na seção de modelagem. Garantir o uso de nomes em inglês. Se os arquivos `.sql` antigos do repositório contêm algum esquema inicial (mesmo em português), refatorá-los para a nova nomenclatura pode acelerar este passo. Após criar, inserir dados iniciais correspondentes ao conteúdo já presente nos Markdown (ou simplesmente rodar a sincronização para povoar automaticamente).
4. **Atualizar os scripts de sincronização**: Revisar o arquivo `config/sync.config.json` para refletir os nomes exatos das tabelas/colunas criadas. Por exemplo, se alteramos algum nome (digamos `catalog.uc.name` em vez de `title` como no draft), alinhar aqui. Atualmente no config proposto, os nomes já estão em inglês e provavelmente consistentes [15], mas confirme se vamos usar exatamente os mesmos (ex.: decidimos por `name` ao invés de `title`, etc., então ajustar `mapFrontmatter` conforme). Depois, ajustar os scripts TypeScript em `scripts/` (como `utils/supabase.ts` e possivelmente um script principal de sync) para usar as novas tabelas. O snippet no plan mostra uma função `supabaseAdmin()` criando o client [32] – isso deve continuar funcionando, usando a service key.
5. **Inserir os dados iniciais**: Popular o banco com os cursos/UCs de exemplo já disponíveis. O jeito mais direto é rodar o workflow de sincronização: commitar os arquivos Markdown existentes (que são rascunhos iniciais de LESTI, ALG1 etc.) e deixar o GitHub Action criá-los no DB. Se preferir, pode rodar o script manualmente local (carregando as env vars e executando `npm run sync:md:db`). Verificar no Supabase se os registros foram criados corretamente (tabelas `course`, `uc`, etc. preenchidas).
6. **Testar o front-end localmente**: Iniciar o Hugo em modo desenvolvimento (por exemplo, `hugo server -D` ou via `npm run dev` se configurado). Isso deve servir o site estático na porta local. A essa altura, as páginas irão carregar, mas possivelmente vazias de conteúdo dinâmico até implementarmos os fetches. Então, incluir e testar os scripts JS de fetch:
7. Configurar o Supabase anon key no front-end: por exemplo, criar um arquivo JS `supabaseClient.js` exportando o cliente (`createClient(SUPABASE_URL, SUPABASE_ANON_KEY)`), e incluí-lo.
8. Implementar uma função simples para detectar o tipo de página atual. Podemos usar atributos no HTML ou o caminho para saber se é página de curso, UC ou tópico. Por exemplo, no template Hugo de curso, incluir `<body class="page-course" data-course="{{ .Params.code }}" data-plan="{{ .Params.plan_version }}">`. Para UC: `<body class="page-uc" data-`

`uc="{{ .Params.code }}">`. Para tópico: `<body class="page-topic" data-topic="{{ .Params.slug }}">`. Assim, nosso script global pode ler `document.body.classList` ou dataset e chamar a função correspondente (`loadCoursePage`, etc.).

9. Testar chamada por chamada: abrir uma página de curso, verificar se os dados do Supabase estão sendo trazidos corretamente e aparecendo. Ajustar quaisquer erros de consulta (por ex., verificar se os nomes de coluna batem 100% com os do banco). Console do navegador e supabase logs ajudarão.
10. Testar páginas de UC e tópico do mesmo modo. Por exemplo, a página `Algoritmos I` deve exibir sua descrição, lista de resultados de aprendizagem, etc., e links para `Estruturas de Dados` tópico, que ao clicar deve carregar a página de tópico com seu conteúdo.
11. **Personalizar elementos visuais**: Garantir que a apresentação esteja de acordo com a expectativa de elegância e simplicidade. O tema Doks fornece um visual base; poderemos ajustar cores, logotipo, e ocultar funcionalidades não usadas no MVP. Por exemplo, se o tema tem busca ou seção de blog, podemos desativar. Manteremos o design limpo, com tipografia legível e layout responsivo. Também, como highlight do projeto, incorporar de forma destacada os elementos chave: playlists de vídeo incorporadas (ex.: usando embed do YouTube player) e conteúdo Markdown formatado (talvez estilizando tabelas, listas, etc., se houver no conteúdo).
12. **Deploy**: Para publicar o site, podemos usar o mesmo esquema anterior pensado (Coolify em servidor próprio, ou mesmo GitHub Pages/Netlify já que é estático). O pipeline de CI pode ser adaptado para também fazer deploy do Hugo output. No entanto, para o MVP, até mesmo rodar manualmente o `hugo` e copiar os arquivos para um servidor Nginx é válido, dado que não há backend custom. A única consideração especial no deploy é preservar as variáveis de ambiente do Supabase para o front-end. Se for um deploy estático puro (como GH Pages), precisamos embutir o SUPABASE_URL e anon key no código JS durante o build (não podemos deixá-las só em env no servidor, pois não há execução server-side). Uma solução é colocar essas keys diretamente no código do cliente (são públicas de qualquer forma). Então, garantir que no build final os scripts tenham a URL e chave correta (poderíamos usar substituição de variáveis via Hugo templating). **Nunca** incluir a service key no front-end, apenas a anon.

Seguindo todos esses passos, teremos uma aplicação totalmente estática, simples e performática, com conteúdo rico carregado sob demanda do Supabase. A manutenção fica trivial: edita-se Markdown para conteúdo, e ajusta-se HTML/JS para mudanças de layout ou comportamento. O desenvolvimento local também é fácil (Hugo server + Supabase local ou cloud). E ainda assim, a solução é robusta para evoluir:

- **Extensibilidade**: Já teremos a base para adicionar autenticação de usuário e acompanhamento de progresso na próxima fase (basta habilitar o Supabase Auth e criar tabelas de progresso, sem alterar a arquitetura do front-end além de adicionar condições de login e salvamento de estado via API).
- **Escalabilidade de conteúdo**: Novos cursos ou traduções podem ser adicionados criando novos arquivos Markdown e deixando a automação sincronizar e o front-end listá-los.
- **Colaboração**: Por ser open-source e baseado em Markdown, outros contribuidores podem facilmente propor melhorias de conteúdo via pull requests [33], e as validações automatizadas garantirão consistência.

Considerações Finais

Em conclusão, a proposta reformulada abraça a filosofia "Jamstack": **frontend estático + conteúdo via API**. Isso atende aos requisitos do usuário de ter um site estático (rápido, fácil de hospedar) e ao mesmo tempo flexível para conteúdo dinâmico. Mantemos tudo simples em termos de stack (Hugo, Supabase, JavaScript puro ou minimalista no cliente) e organizado em termos de código (padrões

consistentes, nomes em inglês, conteúdo versionado). Com essa implementação, o agente desenvolvedor terá um guia claro para efetuar as configurações iniciais no repositório e no Supabase, pavimentando o caminho para um MVP funcional do FACODI. Conforme o próximo sprint, poderemos então incorporar funcionalidades adicionais (sistema de usuários, progresso, etc.) sobre essa base sólida e limpa.

Fontes e Referências Utilizadas:

- Documentação e estrutura proposta no repositório FACODI (conteúdo Markdown e configuração de sincronização) [34] [15], enfatizando a abordagem de Markdown como fonte primária e mapeamento para banco de dados.
- Detalhamento das convenções de conteúdo (frontmatter de cursos, UCs e tópicos) conforme o plano do projeto [8] [10].
- Diretrizes de melhor prática descritas no plano (single source of truth, workflows de CI para validação e sync) [5], adotadas e traduzidas em passos práticos nesta proposta.

---

[1] [29] [33] README.md
https://github.com/Monynha-Softwares/facodi.pt/blob/3bb5a7bde95ae51d6fa16b5a9c2aaa8d824db6a2/README.md

[2] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [30] [31] [32] [34] PLAN.md
https://github.com/Monynha-Softwares/facodi.pt/blob/3bb5a7bde95ae51d6fa16b5a9c2aaa8d824db6a2/PLAN.md

[3] [4] module.toml
https://github.com/Monynha-Softwares/facodi.pt/blob/3bb5a7bde95ae51d6fa16b5a9c2aaa8d824db6a2/config/_default/module.toml