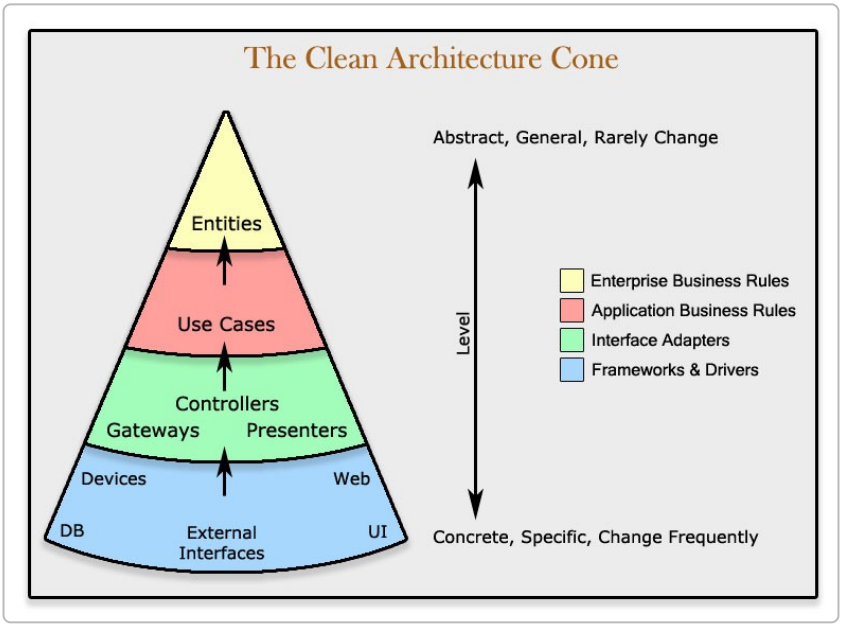
ChatGPT

Visão Geral da Arquitetura (DDD em Camadas)



Exemplo de diagrama de arquitetura limpa em camadas, destacando Domínio (entidades), Aplicação (casos de uso), Interface (apresentação) e Infraestrutura (mecanismos externos).

A arquitetura Domain-Driven Design (DDD) organiza o projeto em **camadas** bem definidas, cada uma com responsabilidades distintas. Isso ajuda a manter o código **modular, evolutivo e de fácil manutenção**, separando as preocupações de **regra de negócio**, **fluxo de aplicação**, **interface com o usuário** e **detalhes técnicos**. Em resumo [1]:

- **Camada de Domínio:** contém os **conceitos e regras de negócio** centrais. É o coração do sistema, independente de detalhes técnicos.
- **Camada de Aplicação:** representa os **casos de uso** ou fluxos de negócio do projeto, coordenando operações para atender requisitos específicos.
- **Camada de Interface (UI/Apresentação):** lida com as **informações de entrada e saída**, ou seja, interação com o usuário (no frontend, a UI do navegador).
- **Camada de Infraestrutura:** implementa os **mecanismos técnicos** do sistema (acesso a dados, APIs externas, persistência, etc.).

Uma **regra fundamental** da arquitetura em camadas é que camadas internas **não conhecem** as externas. Ou seja, as dependências de código **apontam sempre da camada externa para a interna** (nunca o contrário) [2]. A camada de domínio é a mais interna e isolada – ela **não depende de nenhuma outra camada**. Já as camadas externas podem depender das internas conforme necessário (por exemplo, a camada de aplicação pode usar o domínio; a UI pode chamar a aplicação; a infraestrutura pode conhecer interfaces do domínio para implementar repositórios). Isso segue o Princípio da Inversão de Dependências, garantindo baixo acoplamento. Em suma: **Domínio** não importa nada de ninguém, **Aplicação** importa

apenas Domínio, **UI** pode importar Aplicação/Domínio, e **Infra** (a mais externa) pode interagir com todas as anteriores.

No projeto em questão, essa arquitetura em camadas está refletida na estrutura de pastas sob `src/` (seguindo até certo ponto o estilo "Clean Architecture" ou “onion architecture”). Abaixo detalhamos cada camada – seu papel, quais arquivos atuais pertencem a ela, e diretrizes práticas do que deve (ou não) estar em cada uma.

Camada de Domínio (`src/features/**/domain`)

Papel: A camada de **Domínio** encapsula a lógica de negócio e os conceitos fundamentais da aplicação. Aqui ficam as **entidades** (objetos de negócio com identidade) e possivelmente **objetos-valor** ou **serviços de domínio**. Esta camada define *o que* o sistema faz em termos de negócio (regras, comportamentos), sem se preocupar *como* isso será exibido ou de onde vêm os dados. O domínio deve ser independente de detalhes de infra e UI – idealmente não há nenhuma dependência de frameworks, APIs externas ou manipulação de DOM nesta camada [2] .

No seu projeto, os arquivos de domínio atuais são as classes JavaScript que representam entidades do mundo do jogo League of Legends:

- `Champion.js` (em `features/champions/domain/`): representa um campeão. Provavelmente contém propriedades como nome, id, função (lane), etc., e poderia conter comportamentos relacionados ao campeão. Por exemplo, poderíamos ter métodos como `getSkins()` (embora buscar skins talvez seja responsabilidade de aplicação/infra, discutiremos depois) ou outras regras relacionadas a um campeão. Essa classe modela o conceito de Campeão no domínio do projeto.
- `Skin.js` (`features/skins/domain/`): representa uma skin (aparência) de um campeão, contendo dados como nome da skin, ID do campeão a que pertence, talvez URLs de imagem, raridade etc. Também é uma entidade de domínio. Regras específicas de skins (por exemplo, determinar se é uma skin lendária) poderiam ficar aqui como métodos ou propriedades calculadas, se fizer sentido.
- `Gallery.js` (`features/gallery/domain/`): possivelmente modela a galeria de skins (o carrossel). Pode armazenar a lista de skins a exibir e o estado atual do carrossel (slide ativo, etc.). Se a lógica de rotação automática (autoplay) ou transição de slides for complexa, parte dela poderia ser encapsulada aqui como regras de negócio do “carrossel” (por exemplo, métodos como `nextSlide()` para calcular o próximo slide ativo, ou restrições de quando mostrar certos elementos de UI). **Importante:** Mesmo sendo um carrossel algo visual, podemos tratar a sequência de skins e a regra de qual skin está ativa como uma lógica de domínio da funcionalidade *Galeria* – isto pode ser definido de forma agnóstica de UI (por ex., atualizar um índice ativo dentro do objeto Gallery), enquanto os detalhes de *apresentar* essa mudança ficam na camada UI.

Responsabilidades: Essas classes de domínio devem manter **toda lógica relevante de negócio**. Por exemplo, se houvesse uma regra de que *“ao selecionar um campeão, deve-se marcar internamente que esse campeão já foi visto em tal sessão”*, esse tipo de estado poderia ser parte do domínio do campeão (mas também poderia ser tratado na aplicação, conforme a natureza da regra). Outra regra de domínio poderia ser algo como *“um campeão só pode ter skins de determinados tipos”* – validações assim poderiam ser métodos no **Champion** ou **Skin** (embora neste projeto específico as regras de negócio sejam simples,

principalmente exibição). O ponto chave é: **lógica que representa o negócio ou as regras do jogo deve residir no domínio**, para que seja reutilizável e testável isoladamente do resto.

O que NÃO vai no Domínio: Código de interface (manipulação de HTML/DOM, eventos de clique, etc.) **não** pertence aqui. Também não devem existir chamadas de APIs, acesso a arquivos JSON, nem qualquer código dependente de detalhes de infraestrutura. Por exemplo, ler o arquivo `champions.json` ou formatar uma string para exibição na tela não são responsabilidade do domínio. O domínio **não conhece** JSON, fetch, localStorage, nem elementos do DOM – ele lida com *objetos e regras puras*. Assim, **Champion.js** e **Skin.js** não devem, por exemplo, criar elementos HTML ou fazer `fetch()` de dados; eles simplesmente armazenam dados e métodos de negócio. Essa independência torna o domínio **facilmente testável** (você pode instanciar um Champion ou Skin em um teste sem precisar de navegador ou servidor). Em suma, mantenha o domínio limpo, sem "poluir" com detalhes externos.

Camada de Aplicação (`src/features/**/application`)

Papel: A camada de **Aplicação** (às vezes chamada de *Application Services* ou *Use Cases*) orquestra as operações do sistema para realizar casos de uso específicos. Aqui ficamos os **serviços de aplicação** – funções ou classes que representam **ações/fluxos de negócio** completos do ponto de vista da aplicação. Eles utilizam o domínio para executar regras e interagem com a infraestrutura para obter ou persistir dados, mas de forma abstrata. A aplicação define *como* as funcionalidades são coordenadas para atender aos requisitos. Em outras palavras, um serviço de aplicação **recebe solicitações da interface (UI)**, possivelmente faz validações ou transformações simples, **chama objetos do domínio para executar as operações de negócio** e coordena o resultado [3]. É a "cola" entre UI, Domínio e Infraestrutura.

No seu projeto, os casos de uso atuais incluem:

- `getChampionsUseCase.js` (`features/champions/application/`): caso de uso para **obter a lista de campeões**. Provavelmente, quando a aplicação inicia ou quando precisamos popular o seletor de campeões, este use case é chamado. Ele deve recuperar os dados de campeões do repositório (camada de infra) e possivelmente convertê-los em objetos de domínio `Champion` antes de retorná-los. A responsabilidade aqui é centralizar a lógica de obtenção de campeões, para que a UI não precise saber de onde vem os dados (JSON, API etc.).
- `getSkinsByChampionUseCase.js` (`features/skins/application/`): caso de uso para **obter as skins de um determinado campeão**. Quando o usuário seleciona um campeão, a UI deve carregar as skins correspondentes – este use case cuida disso. Recebe uma identificação/nome do campeão (provavelmente passado pela UI através do controller), consulta o `skinsRepository` (infra) para pegar os dados das skins filtradas daquele campeão, possivelmente instancia objetos `Skin` no processo, e retorna essa lista de skins de volta para que a camada de apresentação possa exibí-las. Poderia também aplicar alguma lógica extra, e.g., ordenar as skins ou inserir alguma informação default (como uma skin "placeholder" caso não haja skins, etc.), se isso fizer parte do fluxo de negócio.
- `loadGalleryUseCase.js` (`features/gallery/application/`): caso de uso para **carregar a galeria de skins** (carrossel) com base no campeão selecionado. Esse use case pode funcionar em conjunto com o anterior: ele orquestra o processo completo de preparar a galeria. Por exemplo, poderia internamente chamar o `getSkinsByChampionUseCase` para obter as skins do campeão, depois usar a entidade `Gallery` do domínio para organizar essas skins no modelo de dados do

carrossel, e então retornar um objeto `Gallery` preenchido ou mesmo já acionar a atualização da UI da galeria. O design exato pode variar: ou o **championsSelectController** chama primeiro `getSkinsByChampionUseCase` e depois chama `loadGalleryUseCase` passando as skins, ou `loadGalleryUseCase` sozinho pode fazer ambos (pegar skins e preparar a galeria). O importante é que *toda a coordenação necessária para exibir a galeria do campeão selecionado* seja feita na camada de aplicação, não diretamente na UI.

Responsabilidades: A camada de aplicação deve **coordenar os passos** necessários para realizar cada ação do usuário ou processo do sistema. Ela atua como **orquestradora**: invoca repositórios da infra para obter dados, aciona métodos do domínio para aplicar regras de negócio, e retorna resultados prontos para a camada de apresentação. Muitas vezes, cada funcionalidade importante do sistema corresponde a um use case nesta camada. Por exemplo, no fluxo principal *"Usuário abre o select → seleciona campeão → carrossel carrega as skins"*, podemos identificar dois casos de uso: **listar campeões** (para popular o select) e **mostrar skins do campeão selecionado** (carregar galeria). Esses casos de uso são implementados pelos módulos de aplicação mencionados acima. A aplicação também pode gerenciar coisas como **estado de execução** (e.g., indicar que uma carga está em andamento), embora o feedback visual de "loading" seja da UI, o use case pode, por exemplo, disparar eventos ou retornar *promises* que a UI use para mostrar/ocultar loading.

Um detalhe de design: *serviços de aplicação não precisam ser classes orientadas a objeto; podem ser funções ou objetos simples*, contanto que cumpram seu papel. No seu projeto eles estão como módulos JS exportando funções (por exemplo, possivelmente uma função `getChampions()` dentro de `getChampionsUseCase.js`). Está ótimo assim.

O que NÃO vai na Aplicação: Lógica de apresentação (como manipular HTML) não deve estar aqui. A aplicação **não deve fazer diretamente nada na tela** – em vez disso, retorna dados para que a UI faça isso. Também não deve acessar bancos de dados ou arquivos *diretamente* – para isso existem os repositórios na camada de infraestrutura. Se o use case precisa de dados, ele deve chamar um método de repositório (por exemplo, `championsRepository.getAll()`), ao invés de ele próprio fazer um fetch. Ou seja, evite que a lógica de aplicação dependa de detalhes de armazenamento. Além disso, **evite regras de negócio complexas duplicadas aqui** – se algo é puramente regra de negócio, melhor mantê-lo no domínio. A camada de aplicação pode realizar **regras de negócio *específicas do caso de uso***, ou seja, lógica *aplicativa*. Por exemplo, decidir *"se não houver campeão selecionado, não carregar o carrossel"* é uma regra de fluxo de aplicação que pode ser controlada aqui ou mesmo na UI; já *"um campeão não pode ter mais de X skins"* seria regra de domínio.

Em resumo, usecases **podem conter alguma lógica** sim (são as "regras de negócio da aplicação" distintas das regras de negócio do domínio) [4] . Por exemplo, um use case poderia verificar *"o campeão selecionado é válido?"* ou *"já temos skins em cache para esse campeão?"* antes de decidir buscar dados. Mas eles não devem fazer cálculos de negócio que as entidades poderiam fazer, nem saber detalhes de UI. Também não é função deles formatar strings de exibição ou manipular elementos do DOM.

(Observação: Em DDD clássico, a camada de aplicação é fina e não contém estado próprio – é mais um coordenador mesmo. Aqui no frontend, você pode tratá-la de forma semelhante.)

Camada de Infraestrutura (`src/features/**/infra` + `src/shared/config` etc.)

Papel: A camada de **Infraestrutura** cuida de tudo que envolve detalhes técnicos externos: acesso a dados, chamadas de API, armazenamento, arquivos, integração com bibliotecas de terceiros, configurações de ambiente, etc. Basicamente, ela implementa os "mecanismos" necessários para que a aplicação funcione, **isolvendo esses detalhes das camadas de negócio**. No contexto deste projeto frontend, a infra estará principalmente nos **repositórios** que fornecem dados dos campeões e skins (que no seu caso vêm de arquivos JSON locais, mas poderiam vir de uma API REST, por exemplo). Também podemos considerar configurações como a do Cloudinary e scripts de build como parte de infraestrutura.

Arquivos de infraestrutura atuais no projeto:

- `championsRepository.js` (`features/champions/infra/`): provavelmente implementa a lógica para **obter os dados de campeões** da fonte de dados. Dado que há um `public/data/champions.json`, este repositório deve carregar esse JSON (via fetch ou import estático) e fornecer métodos para acessar os campeões. Por exemplo, pode exportar uma função `getAllChampions()` que faz fetch no `champions.json` e retorna um array de objetos (idealmente convertendo para instâncias de `Champion` do domínio, embora ele também possa retornar dados puros e deixar o use case montar os objetos de domínio). Se houver necessidade de pegar um campeão específico, poderia haver um método `getChampionById(id)` aqui também. O ponto é: **toda interação com a fonte de dados de campeões acontece aqui**, isolada do restante do sistema.
- `skinsRepository.js` (`features/skins/infra/`): similarmente, lida com **obtenção de dados de skins**. Provavelmente lê `public/data/skinColections.json`. Pode ter um método `getSkinsByChampion(championId)` que carrega o JSON de skins e filtra apenas as skins cujo campeão corresponda ao ID passado. Ou poderia carregar tudo de uma vez e filtrar – dependendo de como foi implementado. Assim como o repositório de campeões, ele pode também instanciar objetos `Skin` do domínio para cada skin, garantindo que a aplicação receba entidades ricas ao invés de objetos literais. (Novamente, isso é opcional – há designs onde o repositório retorna DTOs e o caso de uso os converte em entidades. O importante é que a decisão de onde converter seja consistente).
- `cloudinary.js` (`src/shared/config/`): provavelmente contém configurações ou funções utilitárias para integrar com o Cloudinary (talvez base URLs, presets, etc. para gerar URLs de imagens das skins). Este arquivo é parte de infraestrutura pois trata de detalhes de um serviço externo (armazenamento/imagens). Por exemplo, pode exportar uma URL base ou uma função `getCloudinaryURL(imgPath)` que formata o caminho completo. Camadas superiores podem usar essas funções, mas não precisam saber os detalhes (e.g., se no futuro mudar de Cloudinary para outro CDN, só este arquivo deve mudar).

Além disso, note que há um diretório `scripts/` no root do projeto contendo vários scripts (`sync-icons.js`, `upload-skins.js`, etc.). Esses scripts não fazem parte do aplicativo em si, mas são **ferramentas de infraestrutura** usadas durante o desenvolvimento (para preparar dados, fazer upload de ativos para Cloudinary, gerar JSON a partir de fontes externas, etc.). Eles interagem com APIs e arquivos, então conceitualmente pertencem à infraestrutura do projeto, embora estejam fora da pasta `src` principal. Eles não influenciam diretamente a estrutura em tempo de execução do front-end, então

mencionamos apenas por completude. O importante é entender que coisas como *geração de dados* e *uploads* são preocupações de infra (e é bom mantê-los separados do código de negócio, como você fez).

Responsabilidades: Em resumo, a infra **fornece serviços** para as camadas superiores obterem o que precisam do mundo externo ou realizar tarefas técnicas. Ela **implementa** interfaces ou contratos que o domínio/aplicação definem. Por exemplo, poderíamos definir (implicitamente) que existe um contrato `ChampionsRepository` com operações `getAll()` e `getById()`. A classe/objeto concreto em `championsRepository.js` implementa esse contrato lendo do JSON. Se amanhã os campeões vierem de uma API REST, você adaptaria este arquivo (ou criaria outra implementação) sem mudar o restante do sistema – essa é a ideia. A infra também pode incluir outros aspectos técnicos: se houvesse logs em arquivo, acesso a localStorage, workers web, etc., tudo entraria nesta camada.

O que NÃO vai na Infraestrutura: Não coloque regras de negócio aqui. O repositório **não deve decidir coisas além de carregar/entregar dados**. Por exemplo, determinar *quais campeões pertencem a determinada lane* é lógica de negócio – isso deveria estar no domínio ou aplicação (ou até pré-calculado no JSON). O repositório só deveria fornecer os dados brutos ou, no máximo, filtrados de acordo com parâmetros simples que a camada de aplicação pedir. Também não coloque lógica de apresentação (obviamente a infra não mexe no DOM ou visual). Em essência, a infraestrutura é "burra" em termos de conhecimento de negócio: ela sabe falar com fontes externas, mas não sabe *por que* – quem sabe o porquê é o domínio/aplicação. Por exemplo, o `skinsRepository` pode filtrar skins por campeão porque isso é basicamente uma busca de dados; mas ele não deveria, digamos, ordenar skins por raridade *a não ser* que isso seja apenas detalhe de dados (se raridade for puramente um campo no JSON, poderia até ordenar, mas se raridade tivesse significado de negócio, melhor a aplicação decidir).

Outra coisa: a infra pode lançar exceções ou retornar erros se algo falhar (ex: JSON não encontrado), mas a decisão de *o que fazer* com esse erro (tentar de novo? mostrar mensagem ao usuário?) cabe à camada de aplicação/UI. Mantenha a infra focada em **I/O e tecnologia**.

Camada de Interface de Usuário (UI) (`src/features/**/ui` + `src/ui` global)

Papel: A camada de **UI (User Interface)**, ou camada de Apresentação, gerencia tudo que é **interação com o usuário e apresentação visual**. No front-end, isso significa manipular o DOM, atualizar elementos HTML, ouvir eventos como cliques, mudanças em inputs, disparar animações, etc. Essa camada recebe as entradas do usuário, envia essas entradas para a camada de aplicação (casos de uso) processar e, então, pega os resultados (dados, estado) e atualiza a interface conforme necessário. Ela atua, portanto, como a "face" do sistema e também como o **controlador de fluxo de interação**.

No seu projeto, a camada de UI está dividida em **componentes por funcionalidade**, dentro de `src/features/*/ui`, e também há alguns scripts utilitários globais em `src/ui/`. Vamos separar por funcionalidade:

- **Champions UI:** Temos o `championsSelectView.js` e o `championsSelectController.js` (`features/champions/ui/`). Juntos, eles implementam o componente de seleção de campeão (o `<select>` e possivelmente elementos relacionados). O **Controller** é responsável por lidar com os eventos e coordenar ações, enquanto a **View** lida com a atualização visual do select. Por exemplo,

`championsSelectController` provavelmente: inicia carregando a lista de campeões (chamando `getChampionsUseCase` na aplicação), então passa esses dados para o `championsSelectView` para renderizar as `<option>` no `<select>` de campeões. Depois, ele escuta o evento de mudança (`change`) nesse select; quando o usuário seleciona um campeão, o controller captura qual campeão foi escolhido (talvez perguntando à View ou diretamente acessando o select), e então aciona o próximo passo da aplicação – possivelmente chamando `getSkinsByChampionUseCase` e/ou `loadGalleryUseCase` para obter as skins daquele campeão. Em seguida, ele deve passar o resultado para a parte de galeria (ver abaixo). Em suma, `championsSelectController.js` centraliza a lógica de *"quando o usuário escolher um campeão, o que fazemos"*. Já o `championsSelectView.js` cuida de detalhes como *popular o dropdown com nomes de campeões*, *exibir placeholder ou texto padrão*, *talvez desabilitar o select enquanto carrega*, etc. A View pode oferecer métodos que o controller chama, como `championsSelectView.renderOptions(listaCampeoes)` ou `championsSelectView.getSelectedChampion()` e `championsSelectView.showLoading()`. Essa separação facilita testes e manutenção: você poderia trocar o tipo de input (por exemplo, de um `<select>` para uma lista de botões) alterando principalmente a View, enquanto o Controller (e as camadas abaixo) permanecem quase iguais.

- **Gallery (Carrossel) UI:** Temos `galleryController.js` e `galleryView.js` (`features/gallery/ui/`). Esta dupla gerencia o carrossel de skins. Após o campeão ser selecionado (lá no championsSelectController), provavelmente o galleryController será informado de quais skins mostrar. Há algumas maneiras de coordenar isso: o championsSelectController poderia chamar diretamente um método em galleryController (por ex: `galleryController.loadSkins(skinsDoCampeao)`), ou poderia disparar um evento global que o galleryController escuta (uma abordagem mais desacoplada). Como o projeto é pequeno, é possível que o championsSelectController simplesmente **importe** o galleryController e invoque algo. Suponhamos que `galleryController.loadGallery(skins)` seja chamado – então dentro desse método, ele poderia usar a entidade de domínio `Gallery` para talvez armazenar estado (ex: definir o índice inicial, etc.), e então chamar `galleryView.render(skins)` para exibir as imagens. O **galleryView.js** seria responsável por manipular o DOM do carrossel: criar/remover slides (`<div>` ou `<img>` para cada skin), atualizar classes CSS para mostrar apenas o slide ativo, aplicar estilos de fade, etc. Também poderia controlar elementos como setas de navegação, indicador de slide atual, etc., mostrando ou escondendo conforme necessário. O **galleryController.js** provavelmente também lida com **eventos do carrossel**: por exemplo, clique no botão "Próximo" ou "Anterior" do carrossel, ou evento de swipe. Nesses casos, o controller captura o evento, atualiza o estado (por ex, incrementa o índice ativo, possivelmente usando um método de `Gallery` do domínio para garantir que loops ou limites sejam respeitados) e então pede à View para atualizar a interface (mostrar o novo slide e atualizar a div de info, etc.). Além disso, o galleryController poderia iniciar o **autoplay**: por exemplo, configurar um `setInterval` que a cada X segundos avança o slide automaticamente. A lógica de *quando* e *como* avançar pode residir aqui (ou parte no domínio Gallery), enquanto a View apenas reflete a mudança visual. Em resumo, a **UI da galeria** gerencia toda a interação de slideshow.

- **Outros componentes UI globais (`src/ui/`):** Vejo que você tem pastas como `navbar/initNavLinks.js`, `theme/initThemeSwitcher.js`, `toast/initToast.js`, `input/initInputSelect.js`. Esses parecem ser **scripts de inicialização de elementos da interface global**, fora do contexto de um recurso específico:

- `initNavLinks.js`: provavelmente adiciona comportamento nos links da navbar (por exemplo, scroll suave, ou marcar link ativo).
- `initThemeSwitcher.js`: configura o botão de alternância de tema (claro/escuro), aplicando classes CSS ou mexendo em `localStorage` para salvar preferência de tema.
- `initToast.js`: inicializa toasts de notificação (talvez configura para mostrar automaticamente certas mensagens ou fornece funções globais para disparar toasts). Pela descrição da funcionalidade "*Toast automático com info do campeão (1 por lane)*", imagino que esse módulo gerencia os toasts e possivelmente expõe uma função tipo `showChampionToast(champion)` que o championsSelectController pode chamar após selecionar um campeão. Ele poderia verificar "já mostramos toast para lane X?" – essa lógica poderia estar aqui ou no controller; discutiremos essa funcionalidade adiante.
- `initInputSelect.js`: talvez configura estilização ou eventos customizados para selects (pode estar relacionado ao componente de select customizado).

Esses arquivos no `src/ui` demonstram que além das **UI por feature**, há também **UI compartilhada** (componentes reutilizáveis ou configurações gerais). Eles também fazem parte da camada de apresentação. A distinção é apenas organizacional: você separou o código de UI específico de "champions" e "gallery" dentro de features, e manteve código UI genérico (nav, tema, toast, input genérico) em uma pasta `src/ui`. Isso é uma boa organização. O importante é que todos são parte da **última milha** antes do usuário: manipulam DOM, estilos, eventos, mas **não contêm lógica de negócio** complexa.

Responsabilidades: A camada de UI deve **apresentar os dados da maneira correta e fornecer meios de interação**. Isso inclui: renderizar componentes na inicialização (por exemplo, montar o menu, configurar o seletor de campeão com opções), reagir a eventos do usuário (cliques, mudanças, teclas, etc.), chamar os **casos de uso** apropriados na camada de aplicação quando necessário, e atualizar a interface conforme as respostas/resultado desses casos de uso. Também é responsável por UX: mostrar estados de loading, mensagens de erro ou sucesso (toasts, modais), feedback visual (como a animação de confete dourado que você mencionou), e garantir que a interface esteja consistente com o estado da aplicação.

Em termos práticos no seu projeto, aqui vão alguns exemplos de **como distribuir as funcionalidades novas listadas na UI**:

- *"Atualizar textos (títulos/parágrafos) conforme campeão"*: Isso é claramente responsabilidade da UI. Provavelmente há na página elementos de texto (como um título ou descrição) que devem mudar para refletir o campeão selecionado. Por exemplo, um `<h2>` com o nome do campeão e talvez um parágrafo com a lore/resumo. Esses textos provavelmente vêm do JSON de campeões (campos como lore, título etc.). A sequência seria: o championsSelectController, ao lidar com a seleção de campeão, já tem o objeto Champion selecionado (do domínio) – ele pode então passar esse objeto (ou os dados relevantes) para a UI atualizar os textos. Talvez via um método na championsSelectView, ou diretamente manipulando DOM se for simples. O ideal é encapsular: poderia haver métodos como `championsSelectView.updateChampionInfo(champion)` que atualiza vários campos de texto de uma vez. Novamente, a UI decide *onde colocar cada informação na tela*. Nenhuma outra camada sabe de `<h2>` ou `<p>` elementos – só a UI.
- *"Carregar imagens no carrossel"*: Também UI. O galleryView deve criar os elementos de imagem (`<img src="...">` ou backgrounds via CSS) para cada skin, usando as URLs fornecidas. A obtenção das URLs em si pode envolver a infra (por exemplo, usar uma base URL do Cloudinary configurada em `cloudinary.js` – esse valor foi provido pela infra, mas a UI usa para montar o

atributo `src`). Em alguns casos, a camada de aplicação poderia ter já montado URLs completas, mas isso é detalhe de implementação. De qualquer forma, inserir as imagens no DOM (ou trocar o `src` de imagens placeholder) é tarefa da UI (galleryView).

- *"Renderizar carrossel dinamicamente"*: Significa construir os slides de acordo com os dados recebidos. O galleryView provavelmente tem um template ou função que cria o HTML para cada slide (por exemplo, `<div class="carousel-item"> <img src="..."> <div class="info">Nome da Skin</div> </div>`). Ele faria isso iterando sobre a lista de skins fornecida pelo controller. Se usar algum mecanismo de templating ou simplesmente `innerHTML`/DOM API, pouco importa – é a UI montando a estrutura visual baseada nos dados.
- *"Fade + autoplay"*: Efeitos de transição e rotatividade automática dos slides são comportamento de interface. Provavelmente implementado no galleryView (via CSS classes para fade) e galleryController (usando `setInterval` para o autoplay). Por exemplo, o controller pode iniciar um interval que a cada N segundos chama `nextSlide()`, um método do controller ou do domain Gallery que atualiza o índice ativo, depois chama `galleryView.showSlide(indiceAtual)` para atualizar classes CSS. O fade pode ser puramente CSS (por exemplo, adicionando/removendo uma classe que dispara animação), ou controlado via JS (por exemplo, trocando opacidade manualmente). Mas tudo isso permanece na camada de UI. Nenhuma lógica de negócio depende do fade/autoplay – é puramente experiência do usuário.
- *"Estado de loading / placeholder"*: Também UI. Mostrar um spinner ou uma imagem placeholder enquanto as skins carregam é responsabilidade da UI. O championsSelectController, ao iniciar a carga das skins, pode pedir ao galleryView: `showLoadingPlaceholder()` (por exibir um spinner no carrossel) antes de iniciar o use case de skins. Quando o use case retornar as skins, o controller manda o view remover o placeholder e renderizar as skins. A decisão de quando exibir/ocultar loading vem do fluxo da aplicação (sabe quando iniciou/terminou), mas a implementação (mostrar/ocultar elemento) é feita na UI.
- *"Div de info sincronizada com slide ativo"*: Essa "div de info" provavelmente mostra detalhes da skin ou do campeão correspondentes ao slide atual. Por exemplo, se o slide mostra a skin "Battle Academia Lux", a div de info poderia mostrar "Battle Academia Lux – skin épica lançada em 2020...". Essa div pode estar dentro do galleryView, e sempre que o slide ativo muda, o galleryController informa o galleryView para atualizar o conteúdo dessa div. Os dados para preencher podem vir do objeto Skin (do domínio) associado ao slide ativo (que o controller sabe qual é). Portanto, é UI sincronizando exibição conforme estado fornecido pelo domínio/aplicação.
- *"Layout sem carrossel quando nenhum campeão estiver selecionado"*: Provavelmente, antes do usuário selecionar qualquer campeão, a área onde o carrossel e info apareceriam deve mostrar outra coisa (ou nada). A UI deve tratar esse estado "nenhum campeão ativo". Por exemplo, a galleryView inicialmente poderia renderizar um layout vazio ou um texto tipo "Selecione um campeão para ver as skins". O championsSelectController, ao inicializar a página, poderia garantir que o galleryView está em estado "vazio". Ou a galleryView por conta própria, se não recebe skins, exibe um estado padrão. Em qualquer caso, é uma lógica de apresentação: decidir mostrar uma imagem genérica ou esconder o carrossel por completo. Esse tipo de condicional pode ficar no controller ou na view – por exemplo, o controller poderia chamar `galleryView.clear()` quando nenhum campeão está selecionado, ou passar uma lista vazia de skins e a view internamente sabe que lista vazia significa mostrar estado vazio. O que não queremos é que *camadas de aplicação/domínio* se preocupem com isso – para elas, "nenhum campeão selecionado" pode nem ser uma noção, elas simplesmente não são invocadas até ter algo. A UI que lida com o "antes da seleção".
- *"Cards de pré-seleção via JS" & "Pré-seleção refletindo no carrossel"*: Pelo que entendi, além do `<select>`, talvez haja uma lista de **cards de campeões** (imagens ou botões) para seleção rápida,

ou destaque de alguns campeões. Esses cards seriam outro elemento interativo de UI. Provavelmente, para cada campeão ou para campeões de destaque, existe um card clicável. Ao clicar, o efeito deve ser o mesmo que selecionar no dropdown aquele campeão. Em termos de implementação, você poderia **reutilizar** a mesma lógica: o evento de clique no card pode simplesmente chamar a função do controller de seleção passando o campeão correspondente. Talvez valha a pena criar um `ChampionCardView` e `ChampionCardController`, ou incorporar essa funcionalidade no próprio championsSelectController/View. Depende de como você organiza: se os cards são parte da mesma página/componentes que o select, o championsSelectController pode gerenciar ambos. Por exemplo, no início ele pode gerar os cards (recebendo a lista de campeões, criar elementos para cada um via uma função de view) e adicionar event listeners de clique que apontam para o mesmo handler de seleção. Assim, seja via select ou via card, chama-se algo como `onChampionSelected(championId)`. Internamente, esse método do controller chamaria os use cases de skins/galeria como de costume. **Responsabilidade na camada UI**: gerar os cards no DOM (View), e capturar cliques (Controller). **Não** faria sentido duplicar a lógica de carregamento de skins para cards separadamente – deve ser reaproveitada. Portanto, planeje os controllers de forma que possam ser chamados por diferentes triggers.

- *"Animação de confete dourado"*: Uma animação cosmética de confete, provavelmente para celebrar algo (talvez ao selecionar um campeão lendário? Ou ao visualizar todas skins?). Suponhamos que seja para quando um campeão é selecionado, e se ele é especial (por exemplo, primeiro campeão selecionado ou um campeão da lane favorita do usuário). Independentemente do gatilho, isso é **puro front-end visual**. Pode ser implementada adicionando um elemento `<canvas>` ou `<div>` que reproduz confetes animados. Pode ser acionada no championsSelectController após um determinado evento. Por exemplo, *depois* de carregar a galeria, o controller pode verificar uma condição (tal como `if (champion.isLegendary())` – se houvesse tal propriedade – ou algo do tipo) e então chamar uma função utilitária, talvez em `shared/utils` ou em algum módulo específico, para disparar confetes na tela. Poderia haver um `confetti.js` (não listado, mas você pode criar) que encapsula essa lógica usando CSS ou um pequeno script. Novamente, nada disso vai para domínio ou aplicação – é puramente uma resposta visual. Se houver condição de negócio (como "campeão lendário"), identificar *qual campeão é lendário* é informação de domínio (poderia estar marcado no Champion.js, por exemplo), mas decidir jogar confete por causa disso é detalhe de UI/UX. Então o controller consulta o domínio para a propriedade e a UI executa a animação.
- *"Toast automático com info do campeão (1 por lane)"*: Esse requer um pouco de lógica e estado. A ideia parece ser que ao selecionar um campeão, aparece um toast com alguma informação do campeão, mas garantir que aparece no máximo um toast por *lane* (função) do campeão. Possivelmente, isso visa não spammar muitos toasts se o usuário ficar trocando campeões da mesma lane repetidamente. Implementação provável: cada campeão tem uma lane (top, mid, jungle, adc, support). Podemos manter um controle (por exemplo, um Set ou objeto) das lanes já apresentadas em toast nesta sessão. Onde manter isso? Como é estado de interface (uma decisão de *notificar o usuário*), faz sentido mantê-lo na camada UI, talvez dentro do `championsSelectController` ou no `initToast.js`. Por exemplo, o controller ao selecionar um campeão verifica: `if (!lanesToastShown.has(champion.role)) { Toast.show("Dica: Você escolheu um campeão da rota X..."); lanesToastShown.add(champion.role); }`. O `Toast.show` seria uma função do módulo de toast (UI), que cria/exibe o toast na tela. Assim, a lógica "1 por lane" fica no controller (camada de aplicação/UI – aqui eu diria UI, pois é sobre limitar notificações visuais), ou poderia ser delegada ao toast manager se quisermos. Mas não é parte da regra de negócio do domínio "campeão" – é comportamento de UX. Portanto, definitivamente não colocar nada disso em Champion.js ou similares. Mantém-se no front. Em resumo:

championsSelectController lida com toasts, chamando `initToast.js` (que talvez exporte uma função para disparar toasts).

- *"Badges no topo alternando mensagens"*: Parece ser algum elemento de UI no topo da página que mostra mensagens rotativas, possivelmente dicas ou informações (badges talvez referentes a lanes ou eventos). Exemplo hipotético: pequenos destaques que trocam a cada poucos segundos, tipo " Novo evento disponível!", depois " Confira as skins lendárias!", etc. Se for isso, a implementação é puramente de interface. Provavelmente há um elemento HTML (ou vários) para esses badges, e um script (poderia ser em `navbar` ou outro módulo) que cíclica as mensagens. Isso envolve temporização (`setInterval`) e manipulação de DOM (trocar texto/classe de um badge). Pertence totalmente à camada UI. Pode ser implementado isoladamente, talvez no `initNavLinks.js` ou um novo `initBadgeRotator.js`. O importante é manter separado: não misturar essa funcionalidade com, por exemplo, o galleryController. Cada componente de UI deve fazer uma coisa.

O que NÃO vai na UI: Regras de negócio ou acesso direto a dados. A UI *não deve* acessar o JSON de campeões por si só, nem instanciar repositórios. Se a UI fizesse `fetch('champions.json')` diretamente, estaria violando a separação (misturando apresentação com infraestrutura). Em vez disso, sempre peça à camada de aplicação (use case) para fornecer os dados. A UI tampouco deve conter lógica complexa de processamento de dados – por exemplo, determinar quais campeões pertencem a qual lane para então exibir separados deveria ser tarefa de aplicação (talvez um use case de "listar campeões por lane"). A UI apenas receberia algo já organizado. Em suma, **evite lógica de negócio na UI** [5]. Um exemplo do que *não* fazer: dentro de um event handler do controller fazer cálculos do tipo "se campeão tem mais de 3 skins, marcar um ícone especial". Se "mais de 3 skins" for algo relevante para negócio, deixe a aplicação ou domínio dizer (talvez um campo `champion.hasManySkins`). A UI apenas lê esse campo e decide exibir o ícone visualmente. Assim, a UI fica "burra" em relação ao negócio – ela só sabe mostrar o que pedem e enviar ações do usuário para quem sabe tratar.

Camada de App (Inicialização) (`src/app/main.js`)

Papel: A pasta `src/app` costuma ser o ponto de entrada da aplicação. No seu projeto, `main.js` dentro de `src/app` é provavelmente responsável por **inicializar toda a aplicação** no lado do cliente. Aqui é onde você junta as peças das camadas anteriores e efetivamente **dá o pontapé inicial**: configura event listeners principais, instancia controllers, carrega config global, etc. Em outras palavras, a camada App (ou *Bootstrap/ Composition Root*) amarra as dependências e inicia o fluxo.

No `main.js`, as seguintes ações provavelmente ocorrem (ou deveriam ocorrer):

- **Inicialização de controllers e views:** Você vai instanciar (caso estejam implementados como classes) ou importar e chamar inicializadores dos controllers de cada feature. Por exemplo, se `championsSelectController` expuser uma função `init()` para configurar o dropdown, aqui você chamaria. Ou você pode criar instâncias: `const champController = new ChampionsSelectController(...)`. Nesse caso, você pode passar dependências no construtor. Por exemplo, se o controller precisa de um use case ou view, você injeta aqui. Como seu projeto ainda não tem um sistema formal de injeção, pode ser que o controller mesmo importe o use case internamente. Mas em projetos maiores, o main (camada App) poderia instanciar os repositórios, passar para use cases, depois passar use cases para controllers. Isso permite trocar implementações facilmente. No escopo atual, isso pode ser "overengineering", mas vale ter em mente. Mesmo que não faça manualmente, o `main.js` ao menos **deve chamar o fluxo inicial**: e.g., carrega lista de

campeões e popula o select. Isso talvez já seja feito dentro do controller, mas algo deve acionar. Provavelmente, no `main.js` você chama algo como `championsSelectController.setup()` que internamente chama `getChampionsUseCase` e depois renderiza via view.

- **Configuração global da UI:** `main.js` pode invocar aqueles initializers globais. Por exemplo: `initNavLinks(); initThemeSwitcher(); initToast(); initInputSelect();`. Dessa forma, você habilita as funcionalidades gerais de UI (menu, tema, etc.) ao carregar a página. Esses módulos estão separados, mas precisam ser chamados. O arquivo HTML possivelmente inclui `main.js` como script principal, então é a lugar certo para ativar essas coisas.
- **Gerenciar dependências de módulos:** Se há alguma dependência entre features, o main pode resolvê-las. Por exemplo, comentei antes sobre como o championsSelectController informa o galleryController. Uma abordagem bacana é fazer isso via o main: ao inicializar, você pode inscrever um evento. Ex: o championsSelectController pode emitir um evento "championSelected" com dados do campeão; o galleryController, ao inicializar, registra um handler para esse evento para então carregar a galeria. Implementar um simples sistema de eventos custom (Pub/Sub) ou usar o DOM como event bus (por ex, dispatchEvent em document) são opções. Mas também pode ser simples: passar o galleryController para o championsSelectController. Exemplo: `new ChampionsSelectController(galleryController)`. Então dentro do champion controller, quando seleciona um campeão, ele pode chamar `this.galleryController.loadGallery(skins)`. Isso é **injeção de dependência manual** feita pelo main. Qual método preferir depende de gosto; para poucas interações, chamar diretamente métodos de outro controller é ok. Se quiser manter baixa a acoplagem, eventos são melhores. Independentemente disso, o `main.js` é onde você decide isso – ele conhece todas as partes e conecta quem precisa conversar.
- **Leitura de configurações .env (se houver)**: Você tem um `.env` no projeto (listado na estrutura). Possivelmente contém chaves ou flags (talvez a URL da API ou chave do Cloudinary). O main, ou algum config loader, poderia ler isso e configurar os módulos (no front via substituição no build ou uma configuração global). Embora específico de cada projeto, o `.env` provavelmente é usado pelo bundler, então talvez não precise de código para isso, mas vale citar que as configurações de ambiente acabam influenciando a infra (e.g., a URL da API guardada no .env seria usada no repository).
- **Iniciar a aplicação**: Por fim, main.js poderia chamar algo para iniciar a interface. Por exemplo, se você tem rotas (páginas diferentes), poderia inicializar baseado na rota atual. Mas supondo que este é um app single-page focado na galeria, o main simplesmente configura tudo para a página inicial.

O que NÃO vai em App/main: Regras de negócio não devem ser implementadas aqui. Ele não deve, por exemplo, filtrar campeões ou aplicar lógica – isso cabe às camadas adequadas. O main também não deve ser muito extenso; se começar a colocar muita lógica nele, considere mover para as camadas apropriadas. Pense no main como um **arranjo das peças**: ele monta o tabuleiro, mas não joga o jogo. Após inicializar, a lógica do programa deve "rodar sozinha" reagindo aos eventos do usuário. Ou seja, depois do setup inicial, o main fica praticamente parado (a não ser que haja alguma orquestração dinâmica, o que não parece ser o caso aqui).

Fluxo de Dados e Interação entre as Camadas

Vamos descrever agora, de forma prática, **como as camadas trabalham juntas** no principal fluxo do projeto: *o usuário seleciona um campeão e o carrossel mostra as skins desse campeão*. Esse fluxo exemplifica bem a separação de responsabilidades:

1. **Evento na UI (Camada de Interface):** Tudo começa com uma interação do usuário. Suponha que a página foi carregada e o dropdown de campeões está populado. O usuário clica e escolhe um campeão (por exemplo, "Ashe"). Esse evento de mudança no `<select>` é capturado pelo **ChampionsSelectController** (camada UI). O controller tinha, no setup, registrado um listener para o evento `change` do select.

2. **Chamada de Caso de Uso (Camada de Aplicação):** Ao interceptar a seleção, o **controller** extrai o identificador ou nome do campeão selecionado (por exemplo, `championId = "ashe"`). Em seguida, ele aciona a camada de aplicação para buscar as skins correspondentes. Ele pode chamar diretamente `getSkinsByChampionUseCase(championId)` – uma função/serviço da camada **Application**. Aqui ocorre a transição da camada de UI para a camada de Application: a UI está dizendo "usuário quer ver skins do campeão X, por favor obtenha os dados necessários". Note que o controller não sabe *como* isso será feito, ele apenas confia que o use case lhe dará o resultado. Antes de chamar o caso de uso, o controller também pode fazer tarefas de UI imediatas, como exibir um loading (ex: `galleryView.showLoading()`) para feedback visual enquanto dados são carregados.

3. **Busca de Dados no Repositório (Camada de Infraestrutura):** O **use case** `getSkinsByChampion` recebido o ID do campeão e agora precisa dos dados. Ele então chama o `skinsRepository` (camada Infra) para realmente obter as skins. Pode ser algo como `skinsRepository.fetchByChampion(championId)`. O repositório, por sua vez, realiza a operação técnica: talvez faz um fetch HTTP do arquivo JSON ou consulta um objeto carregado. Vamos supor que o JSON `skinCollections.json` contém um array de skins com um campo `championId`. O repositório então filtra esse array para apenas itens cujo `championId` seja "ashe". Ele então retorna esses dados (digamos, uma lista de objetos `{ id, name, imageUrl, ... }` para cada skin). Em alguns designs, o repository poderia aqui mesmo mapear esses objetos para instâncias da classe `Skin` do domínio antes de retornar, garantindo que a camada de aplicação receba **objetos de domínio** prontos. Isso é bom porque a partir daqui podemos usar métodos do domínio se precisarmos. Vamos supor que sim – o repository retorna uma lista de `Skin` objects.

4. **Criação de entidade de Domínio / Regras de Negócio (Camada de Domínio):** Durante a etapa anterior, ou logo após, pode haver envolvimento da camada de **Domínio**. Se o repository retornou dados puros, o **use case** poderia agora iterar e fazer `new Skin(dado)` para cada skin, efetivamente convertendo-os para entidades de domínio. Além disso, se houvesse alguma regra de negócio a aplicar nas skins obtidas, o use case faria agora. Por exemplo, imagine que há uma regra: "*Skins legacy (legado) não devem ser exibidas no carrossel*". O use case poderia filtrar fora skins marcadas como legado (um atributo possivelmente nos dados). Isso é uma **regra de aplicação específica**. Se fosse uma regra fundamental (tipo "*um campeão não pode ter skins legado disponíveis*"), talvez estivesse no domínio, mas isso é discutível. De todo modo, o use case prepara a

lista final de skins válidas. Podemos também imaginar que ele ordene as skins por data de lançamento ou raridade se a apresentação exigir (embora ordenação para apresentação poderia ser decidido na UI, faz sentido fazê-lo aqui também, já que é parte do "fluxo de negócio da aplicação: mostrar skins em certa ordem").

5. **Retorno do Caso de Uso (de Application para UI):** O use case então **retorna** a lista de skins (agora pronta) para o caller – que é nosso championsSelectController lá na UI. Note: nessa hora, o use case pode já ter envolvido também o **GalleryUseCase** ou `Gallery` do domínio. Por exemplo, talvez ao invés de retornar diretamente a lista, ele quis criar um objeto `Gallery` (domínio) para encapsular as skins. Se seguiu esse caminho, o retorno poderia ser `Gallery` contendo skins e estado inicial. Mas para simplificar, digamos que retornou apenas a lista de skins (domínio ou dados).

6. **Atualização da UI – Carregar Galeria:** Agora de volta no **ChampionsSelectController** (UI), com as skins em mãos, ele precisa atualizar a interface. O fluxo provavelmente delega isso ao **GalleryController**. Dependendo de como você implementou, aqui pode acontecer duas situações:

7. *Via chamada direta:* O championsSelectController invoca um método do galleryController, por exemplo `galleryController.showSkins(skins, champion)`. Passa a lista de skins e talvez o campeão selecionado (caso a galeria precise do nome do campeão para exibir). O galleryController então atua (próximo passo).

8. *Via evento:* Alternativamente, o championsSelectController dispara um evento do sistema (`event "champion-selected"`) com os dados, e o galleryController já estava ouvindo e reage chamando internamente seu método de carregar skins. Essa abordagem evita dependência direta entre os dois controllers. De qualquer forma, **agora o GalleryController entra em ação** com os dados das skins.

9. **Preparação do Carrossel (UI):** O **GalleryController** recebendo a lista de skins vai preparar a exibição. Primeiro, se havia um *placeholder ou loading* mostrado, ele manda a UI remover (e.g., `galleryView.hideLoading()`). Depois, ele pode instanciar/atualizar a entidade de domínio **Gallery** (por exemplo: `this.gallery = new Gallery(skinsList)`). Esse objeto `Gallery` poderia guardar a lista internamente e definir `currentIndex = 0` (primeiro slide). Talvez ele tenha também métodos para calcular o próximo/prev índice circularmente. Guardar esse objeto no controller permite que, ao avançar slides, o controller atualize o estado de forma consistente. **Observação:** Alternativamente, o controller nem precisa de uma entidade Gallery e pode simplesmente armazenar a lista e um índice em variáveis – mas usar o domínio `Gallery` é mais elegante se a lógica de navegação for complexa (e.g., diferentes modos de exibição). Vamos supor que use.
   Em seguida, o galleryController chama a **GalleryView** para de fato renderizar: `galleryView.renderGallery(skinsList)`. Esse método dentro do GalleryView vai construir o HTML dos slides: cria elementos para cada skin (imagens, etc.) dentro do contêiner do carrossel. Também pode atualizar a **div de info** com os dados do primeiro slide ativo imediatamente. Por exemplo, exibe o nome da primeira skin selecionada.
   A GalleryView pode também configurar classes CSS iniciais (deixar só o primeiro slide visível, etc.). Se há um elemento de paginação (tipo bolinhas ou contagem), ele também é atualizado aqui.

10. **Interface pronta para interação:** Agora o usuário vê o carrossel preenchido com as skins do campeão escolhido. A partir daqui, eles podem interagir com o carrossel (próximo, anterior, clicar

numa miniatura se houver). Cada interação dessas também segue o esquema: a UI (galleryController) escuta e, usando o estado/dados (Gallery do domínio, ou lista e índice), calcula o novo estado e só então reflete na view. Por exemplo, ao clicar “Próximo”, o galleryController faz `this.gallery.nextSlide()` (método do domínio que incrementa e retorna o novo índice, possivelmente lidando com loop no final), depois `galleryView.showSlide(newIndex)`. Esta função na view ajusta classes/styles para mostrar o slide correspondente e oculta os demais, e atualiza a div de info para os dados da skin nesse índice. Nada disso envolve pedir novos dados de infra – já está tudo carregado.

11. **Toasts e Efeitos visuais:** Após carregar as skins, o championsSelectController lembra de mostrar o **toast** se aplicável. Ele verifica a lane do campeão selecionado (ex: Ashe é *ADC*). Se ainda não exibiu nenhum toast para "ADC", ele chama algo como `Toast.show("Veja os atiradores dominando o late game!", {lane: "ADC"})`. O módulo `initToast.js` cuidará de realmente exibir na tela a notificação por alguns segundos. Marca-se internamente que "ADC toast já mostrado". Esse controle poderia estar em um objeto no controller (`shownLanes["ADC"]=true`) ou dentro do Toast (por ex., Toast module guarda estado global de lanes mostradas – mas preferencialmente no controller mesmo, mantendo Toast simples).
    Em paralelo, talvez o GalleryController ou championsSelectController dispare a **animação de confete**. Digamos que você decidiu: toda vez que um campeão é selecionado, solta confete. Então no final do handler de seleção, chame `launchConfetti()` (função utilitária global). Essa função manipula o DOM (por exemplo, adiciona um canvas e animates, ou usa uma lib de confetti). Isso ocorre já no final, sem bloquear o restante – só um efeito adicional de UI.

12. **Fluxo completo:** O usuário agora vê as skins, possivelmente um confete animado celebrando a seleção, e um toast no canto dando alguma informação. Ele pode repetir o processo escolhendo outro campeão, e o mesmo ciclo acontece: UI -> Application -> Infra -> Domain -> Application -> UI, respeitando cada camada. Note que, graças à separação, se amanhã a fonte de dados mudar (por exemplo, pegar campeões de uma API online), só a camada de **Infra** muda (o repository faz requisição HTTP em vez de ler JSON). As camadas acima nem percebem a diferença – o championsSelectController ainda chama `getChampionsUseCase`, que por sua vez chama o repositório, mas agora o repositório consulta a API. Do mesmo modo, se mudarmos algo na apresentação (por exemplo, um novo design para o carrossel), podemos alterar o GalleryView extensivamente sem tocar nos casos de uso ou entidades – pois eles não sabem nada sobre HTML. Essa flexibilidade é a vantagem da arquitetura em camadas bem definida [6] [7].

O que Deve ou Não Deve ir em cada Camada – Resumo e Melhores Práticas

Para reforçar, vamos listar **exemplos concretos** do que pertence ou não a cada camada, servindo como um guia de boas práticas para evoluir o projeto:

- **Domínio (Entidades, Regras de Negócio):**
- **Deve conter:** Lógica de negócio pura, válida independentemente da interface. Exemplo: cálculo de atributos de um campeão, verificação se uma skin é rara ou não, regras de associação (um campeão pertence a uma lane). Propriedades e métodos que refletem conceitos do domínio do jogo.

- **Não conter:** Chamadas de API, manipulação de JSON ou fetch, acesso ao DOM ou elementos visuais, nem dependências de bibliotecas de UI. Por exemplo, não fazer `document.querySelector` dentro de `Champion.js` nem `fetch('skins.json')` dentro de `Skin.js`. Também não colocar aqui funções utilitárias que não sejam do negócio (ex: formatar data para string – isso é util, mas se não for lógica de negócio, melhor em utils ou apresentação).
- **Exemplo Positivo:** Um método `Champion.isMarksman()` que retorna true se a role do campeão for ADC – regra de domínio (identificar arqueiros).
- **Exemplo Negativo:** Um método `Champion.renderCard()` que cria um bloco HTML para o campeão – isso seria UI acidental dentro do domínio (evitar!).

- **Aplicação (Use Cases, Serviços de Aplicação):**

- **Deve conter:** Coordenação de ações para atender casos de uso. Chamada a repositórios, invocação de métodos de domínio, aplicação de regras de negócio de *processo*. Por exemplo: "Obter skins do campeão X e ordenar por lançamento" – o *como ordenar* é decisão possivelmente da aplicação (a não ser que ordenação seja intrínseca do domínio). Realizar transformações simples em dados vindos da infra antes de enviar à UI (e.g., filtrar lista, montar DTO para a view). Também pode implementar verificações de permissões ou políticas de aplicação (embora em front isso seja mínimo).
- **Não conter:** Qualquer coisa de interface (nenhum `alert()`, nenhum manipular de CSS ou HTML). Também não deve *decidir layout*. Além disso, a aplicação não faz acesso direto a arquivo/HTTP – sempre use um repositório da infra para isso. Evite também duplicar lógica de domínio aqui – se o domínio já consegue decidir algo, deixe-o fazer lá.
- **Exemplo Positivo:** `getSkinsByChampionUseCase` chamando `skinsRepository.getByChampion(id)`, recebendo dados e instanciando objetos Skin, depois retornando-os. Talvez adicionando: *se nenhum skin encontrado, adicionar uma Skin "Default" indicando que o campeão não tem skins*. Essa adição é uma lógica de caso de uso (pode ser útil para UI mostrar algo em vez de vazio).
- **Exemplo Negativo:** Um use case que manipula o DOM para mostrar um resultado ou que faz `fetch` por conta própria. Ou por exemplo, um use case que calcula algo como dano de ataque do campeão (isso seria regra de jogo – devia estar no domínio).

- **Infraestrutura (Repositórios, Fontes de Dados, APIs, Configurações técnicas):**

- **Deve conter:** Código para acessar dados e recursos externos. Ler arquivos JSON (via fetch), fazer chamadas AJAX/HTTP, acessar localStorage, cookies, integrar com libs (ex: SDK do Cloudinary), quaisquer *drivers* ou implementações técnicas. Também adaptadores: por ex., converter o formato da API para o formato do domínio (DTO para Domain Model). Conexões com back-end, manipulação de resposta, tratamento básico de erro de conexão (ex: logar ou retornar erro).
- **Não conter:** Lógica de negócio ou decisões de fluxo. Não "adivinhar" o que a aplicação quer fazer – apenas cumprir solicitações. Exemplo, *não* aplicar filtros complexos a não ser que a aplicação peça (repository pode ter métodos específicos, mas não deve, por vontade própria, esconder dados). Também não interagir com nada de UI – não chamar alert, não manipular elementos, etc.

- **Exemplo Positivo:** `championsRepository.js` lendo `champions.json`, criando objetos Champion e retornando lista. Ou se fosse via API: fazer `fetch('/api/champions')`, parsear JSON, instanciar Champion ou ao menos normalizar campos, e entregar.

- **Exemplo Negativo:** `skinsRepository` decide que skins com `"isLegacy": true` não devem ser retornadas e filtra silenciosamente. Sem a camada de aplicação saber, isso poderia causar comportamentos inesperados. Melhor passar todas as skins e deixar a camada superior decidir exibir ou não. (A não ser que essa filtragem seja realmente puramente técnica, mas em geral, infra não omite dados de negócio).

- **UI/Apresentação (Views, Controllers, Components):**

- **Deve conter:** Tudo relacionado à **experiência do usuário**. Criação e atualização de elementos DOM, aplicação de estilos (diretamente ou via classes CSS), animações, coleta de inputs do usuário. Lógica de interação: ao clicar A fazer B, etc. Formatação de dados para apresentação – por exemplo, formatar uma data timestamp em string legível (isso pode ser considerado apresentação). Pequenas decisões de UI como "mostrar seção X expandida ou colapsada". Basicamente, implementar o design e comportamento visual desejado.
- **Não conter:** Regras de negócio ou acesso direto a dados que não tenham vindo via casos de uso. A UI não deve conter, por exemplo, cálculos financeiros, regras de validação complexas (essas deveriam estar no domínio ou aplicação). Também evitar cálculos pesados (se precisar processar muita coisa, considere mover para domínio/aplicação ou WebWorker dependendo do caso). E **nunca** quebrar a direção de dependência: UI não deve chamar algo do domínio *sem* passar pela aplicação (exceto talvez ler propriedades prontas do objeto de domínio, isso é ok). Por exemplo, importar diretamente `championsRepository` dentro de um controller para pegar dados iria contra a arquitetura (UI saltando a aplicação).
- **Exemplo Positivo:** `galleryView.js` manipulando o DOM para exibir slides e ocultar outros, adicionando classe `"fade"` para animação, ou construindo elementos `<img>` com src correto. Também um `navbar.js` que adiciona classe `"active"` no menu atual, etc.

- **Exemplo Negativo:** `championsSelectController` contendo dentro dele um objeto com todos os campeões e filtrando por código da lane para passar para a view. Por que negativo? Porque filtrar campeões por lane poderia ser um caso de uso (aplicação) ou um método do domínio (se “filtrar por lane” for recorrente). O controller deveria apenas pedir a lista certa de quem de direito, e não saber detalhes da estrutura interna dos dados. Outro exemplo: um componente React/Vue (se fosse o caso) contendo lógica de cálculo de preço ou de autorização – essas coisas deveriam vir prontas de camadas de negócio, não decididas no componente. No seu contexto, evitar colocar no controller decisões como “se champion.lane == 'JUNGLE' então …” a não ser que isso seja puramente para exibição (tipo escolher um ícone de árvore se for jungle – daí tudo bem, é detalhe visual). Mas se for algo que impacta lógica (como mostrar um confete apenas para a primeira seleção da lane – até isso arguivelmente é UI logic, então ok). Sempre se pergunte: *isso afeta a regra de negócio ou só a aparência?* Se só aparência, pode ficar na UI; se é regra do domínio, não.

- **App/Inicialização:**

- **Deve conter:** Instanciação e configuração inicial. Importação de módulos e chamadas de setup. Por exemplo, definir que repositório usar se houvesse múltiplos (injetar mocks vs produção),

inicializar controllers com dependências, chamar funções que iniciam componentes globais (menu, tema, etc.). Em resumo, wiring.

- **Não conter:** Lógica de negócio ou apresentação propriamente dita. Evite manipular DOM aqui além do mínimo necessário para iniciar (se é que precisa – muitas vezes o main nem toca no DOM, só delega para controllers). Também não fazer fluxos de aplicação completos – esses pertencem aos use cases e controllers. Se o main começar a ter "se isso, faz aquilo" fora do contexto de inicialização, pode estar errado.
- **Exemplo Positivo:** `main.js` chamando `initThemeSwitcher()` para aplicar o tema escolhido anteriormente (lendo talvez do localStorage) – isso é inicialização de UI global, faz sentido aqui.
- **Exemplo Negativo:** `main.js` contendo código para filtrar uma lista de skins e renderizar manualmente – isso estaria realizando lógica de aplicação e interface dentro do main, quando deveria delegar ao feature específico.

Evolução da Arquitetura e Estrutura

Seu projeto já está bem organizado, seguindo boas práticas de DDD no frontend. Ao continuar o desenvolvimento, aqui vão **sugestões para evoluí-lo mantendo a arquitetura limpa**:

- **Adicionar novas funcionalidades seguindo o padrão:** Sempre que for implementar algo novo, pergunte-se em qual **camada** essa funcionalidade pertence. Por exemplo, se quiser adicionar um **sistema de favoritos (marcar skins favoritas)**:
- Você criaria possivelmente uma entidade de domínio **Favorite** ou simplesmente um atributo no objeto Skin (ex: `isFavorite`).
- Na camada de aplicação, um use case `toggleFavoriteSkinUseCase` para marcar/desmarcar favorito, que poderia usar um repositório (talvez salvando em localStorage ou enviando a uma API) para persistir essa preferência.
- Na infraestrutura, implementaria o `favoritesRepository` que salva os IDs favoritos em localStorage ou arquivo.
- Na UI, adicionaria talvez um ícone de coração em cada card de skin: ao clicar, o GalleryController captura e chama o use case `toggleFavoriteSkin(skinId)`. Quando o use case devolver sucesso (ou o novo estado), o controller manda a GalleryView atualizar aquele ícone (cheio ou vazio). Mantendo esse padrão, a base do código continua organizada.
- **Reutilizar e isolar componentes UI comuns:** Você já tem um `shared/utils` (embora não listou arquivos, imagino que vá colocar funções úteis lá). Continuando, se notar que certas partes da UI são genéricas, você pode movê-las para algo compartilhado. Por exemplo, se futuramente houver outro select customizado além de champions, o `initInputSelect.js` pode ser generalizado para inicializar todos. Ou se criar um modal genérico para mostrar informações, mantenha-o desacoplado das regras de negócio, recebendo dados via parâmetros. Essa separação facilita usar componentes UI sem misturar lógica de domínio neles.
- **Manter** Domínio e Aplicação independentes de detalhes externos:** Conforme evolui, tente evitar que suas entidades de domínio ou casos de uso usem diretamente coisas como formatos específicos de dados ou bibliotecas. Por exemplo, se amanhã decidir usar Redux ou outra forma de gerenciar estado, tente integrar isso na camada de aplicação ou infra de forma a não "vazar" para o domínio. O domínio deve continuar sendo apenas JavaScript puro com suas regras, facilmente testável.
- **Implementar interfaces/contratos onde faz sentido:** Em JavaScript não temos interfaces nativas, mas podemos simular. Por exemplo, você pode definir que o `championsRepository` deve ter certos métodos. Se planeja talvez ter diferentes fontes (ex: offline vs online), poderia codificar de

forma que o use case não importe o repositório concreto, mas receba ele injetado. No seu nível de projeto, pode ser overkill; mas para escalar, considere um padrão de injeção simples:

- No `main.js`, instancie os repositórios: `const champRepo = new ChampionsRepositoryJson()` (uma classe você poderia criar).
- Instancie use cases passando esses: `const getChamps = new GetChampionsUseCase(champRepo)`.
- Instancie controllers passando use cases: `const champController = new ChampionsSelectController(getChamps, getSkinsByChampUseCase, galleryController)`.

  Essa abordagem explicita dependências e torna fácil trocar implementações (por ex, usar `ChampionsRepositoryAPI` no lugar de JSON sem mudar o controller). Enquanto o projeto for pequeno e as dependências via import direto não estiverem causando problema, não é obrigatório – mas ter isso em mente ajuda se o projeto crescer ou se você quiser escrever testes unitários passando *mocks* em vez de reais.
- **Organização por Feature (Screaming Architecture):** Você já está organizando por `features` (champions, skins, gallery), o que é ótimo. Isso significa que para adicionar um novo conjunto de funcionalidades, basta adicionar uma nova pasta em `features` com as subcamadas. Por exemplo, se quiser uma feature **"Profile"** (perfil do usuário), você criaria `features/profile/domain/Profile.js`, `application/updateProfileUseCase.js`, etc., e seguir o mesmo esquema. Isso mantém cada contexto isolado. Só tome cuidado com *dependências cíclicas* entre features – se uma feature precisa muito de algo de outra, talvez elas não deveriam ser separadas ou você deve extrair a parte comum para `shared`. No seu caso, "champions" e "skins" são interligadas. Você as separou, mas nota que um use case de gallery precisa de skins. Isso é resolvido chamando o use case de skins (aplicação) dentro do use case de gallery, ou tendo o gallery controller usar ambos. Não tem certo ou errado, mas prefira evitar que o **Domínio** de uma feature conheça outra. Entre Aplicações é mais aceitável (um use case pode chamar outro). Só não "importe bagunça": ex., não faça um import direto de `skinsRepository` dentro do gallery UI – passe sempre pela aplicação.
- **Documentação e Claridade:** Mantenha documentação como esta ( ) no projeto real, talvez em um arquivo README.md ou `docs/architecture.md`. Isso ajuda qualquer colaborador (ou você no futuro) a lembrar onde colocar cada coisa. Adicione comentários nos arquivos complexos explicando suas responsabilidades em 1-2 linhas.
- **Boas práticas de código dentro das camadas:** Além da arquitetura, continue seguindo princípios SOLID em miniatura: funções pequenas, responsabilidade única, nomes claros. Por exemplo, no GalleryView, tenha métodos específicos (ex: `createSlideElement(skin)` que retorna o elemento DOM de um slide) para separar a lógica de construção. No championRepository, se começar a crescer lógica, talvez separar leitura de arquivo e transformação em funções distintas. Assim, cada camada internamente também se mantém limpa.
- **Testabilidade:** Um benefício da sua estrutura é que você pode testar partes isoladamente. Considerar criar alguns testes unitários (se aplicável) para métodos de domínio (ex: testar se `Gallery.nextSlide()` faz o certo nos limites, ou se um método `Champion.getRoleIcon()` retornaria o ícone correto). Você poderia também testar use cases passando repositórios fake (já que é JS, você pode mockar import). Isso garante que a arquitetura não é só teórica, mas traz vantagens reais de confiabilidade.
- **Escalonamento e Performance:** Se futuramente precisar otimizar algo (por ex, pré-carregar imagens de skins para evitar lag no carrossel), você sabe bem onde implementar: pré-carregamento de imagens é infra (pode fazer no repository ou em uma camada de serviço adicional), ou pode ser

uma melhoria de UI (galleryView poderia criar objetos Image off-DOM para pré-carregar). Avalie sem misturar nas camadas erradas.

- **Evitar violações de dependência:** Sempre que sentir tentação de acessar algo diretamente de outra camada, lembre das **regras de dependência**. Como mencionado em uma discussão de arquitetura: *"não quebre a direção de dependência entre domínio e apresentação – ex.: nunca importe algo da apresentação dentro do domínio; nem coloque lógica de negócio dentro de componentes de UI"* [5]. Essa linha guia te manterá no caminho. Se precisar que o domínio notifique algo à UI, isso deve acontecer por meio da aplicação, usando eventos, callbacks ou retornos de função – *nunca* pelo domínio chamando função de UI. Inversamente, se a UI precisa de algo do domínio, deve pedi-lo via aplicação ou usar resultados que o caso de uso retornou.

Em conclusão, sua arquitetura proposta (camadas **domain → application → ui → infra** com features modulares) está bem alinhada com as boas práticas de DDD e Clean Architecture no front-end. Cada camada tem uma responsabilidade clara e os arquivos atuais já refletem isso de forma consistente. Seguindo as orientações acima, você deve conseguir implementar todas as funcionalidades listadas de forma organizada e sustentável. Lembre-se que o objetivo é **manter o código modular, isolando mudanças**: se precisar alterar a aparência, você atua na UI; se mudar uma regra de negócio, atua no domínio; se trocar a fonte de dados, atua na infra – minimizando impactos cruzados [8] [7]. Essa é a grande vantagem dessa arquitetura em camadas bem definida. Bom desenvolvimento!

---

[1] [2] Arquitetura .NET: modelando aplicações com Domain-Driven Design Estratégico | Alura
https://www.alura.com.br/conteudo/arquitetura-dotnet-aplicacoes-domain-driven-design-estrategico?srsltid=AfmBOoq0AqbafOL1xarxzGfiM-pH1IfcTsAgVncTGz7SzhvCxV2jjOj3

[3] Introduzindo a camada de aplicação – Robson Castilho
https://robsoncastilho.com.br/2014/04/16/introduzindo-a-camada-de-aplicacao/

[4] design patterns - Use cases vs Domain vs Repositories: Where should the business logic reside? - Stack Overflow
https://stackoverflow.com/questions/76821302/use-cases-vs-domain-vs-repositories-where-should-the-business-logic-reside

[5] [6] [7] [8] Frontend Architecture. Layers & Domain Mindset - DEV Community
https://dev.to/nefedov-dm/frontend-architecture-layers-domain-mindset-35a0