Nx | Arquitetura em camadas, responsabilidades e relacionamentos

Estrutura de fundação

Existem dois estilos de monorepos: repositórios integrados e repositórios baseados em pacotes. A diferença é mais sobre a mentalidade do que os recursos usados e a escolha do estilo é um espectro - não um booleano.

Os baseados em pacotes

Os baseados em pacotes concentram-se na flexibilidade e na facilidade de adoção.

São uma coleção de pacotes que dependem uns dos outros por meio de arquivos package.json. Com essa configuração, você normalmente tem diferentes dependências para cada projeto. Ferramentas como Jest ou Webpack funcionam normalmente, já que tudo é resolvido como se cada pacote estivesse em um repositório separado e todas as suas dependências fossem publicadas no npm. Mover um pacote existente para um repositório baseado em pacotes é bem fácil, pois geralmente deixamos intocadas as ferramentas e configurações de build existentes. Já não podemos dizer o mesmo para a criação de novos pacotes usando este estilo de monorepo. Criar pacotes é tão difícil quanto criar um novo repositório, pois precisamos configurar tudo do zero.

Exemplo de estrutura

```
package-based/
├── packages/
├── nx.json
└── package.json
```

Lerna, Yarn, Lage, Turborepo e Nx (sem plugins) suportam esse estilo.

Integrados

Concentram-se na eficiência e na facilidade de manutenção.

Eles contém projetos que dependem uns dos outros por importação padrão. Normalmente, há uma única versão de cada dependência definida na raiz. É mais difícil adicionar um pacote existente a esse estilo de repositório porque as ferramentas de build deste pacote podem precisar ser modificadas. É fácil adicionar um novo projeto ao repositório porque todas as decisões de ferramentas já foram tomadas.

Exemplo de estrutura

```
package-based/
├── apps/
├── libs/
├── nx.json
└── package.json
```

Bazel e Nx (com plugins) suportam este estilo.

Tipos de bibliotecas

De uns anos para frente (4~5) até o momento, vi vários tipos de bibliotecas em monorepos, testando a aplicação de todos conforme os descobria. Estes tipos de bibliotecas são a forma com que separamos o código em camadas em um repositório com vários projetos. Com o tempo aprendi que muitos tipos de bibliotecas e também um número alto de aninhamento no repositório causa dificuldade não apenas de navegação mas também de entendimento…

Para manter um senso de organização em repositórios de médio e grande porte, recomenda-se a divisão entre 4 a 6 tipos de bibliotecas, sendo elas camadas com responsabilidades bem definidas e relacionamentos controlados. Trata-se de uma estrutura de diretórios com no máximo 2 níveis de aninhamento ou 3 em caso de agrupamento por escopos, contando a partir de apps, libs ou packages.

Para a maior parte dos casos, recomenda-se o uso de 4 tipos de bibliotecas, sendo suficiente para manter o código organizado em uma arquitetura de baixa complexidade.

1. Bibliotecas `feature`, para componentes com regras de negócio e aplicação (*smart*), geralmente são específicas de algum aplicativo, como páginas ou funcionalidades.
2. Bibliotecas `data-access`, para classes e funções relacionadas a acesso a dados e gerenciamento de estado, como código para acessar uma API back-end.
3. Bibliotecas `util`, para código sem regras e compartilhado por muitas bibliotecas.
4. Bibliotecas `ui`, para componentes sem regras (*dumb*) e compartilhados por algum escopo ou diversas bibliotecas no repositório.

Além destes 4, quando entendido em consenso, temos recomendações para outros 2.

1. Bibliotecas `domain`, para casos de uso com regras de negócios puras, dependentes apenas de entidades e abstrações para acesso a dados.

2. Bibliotecas `api` servem como porta de acesso para permitir que bibliotecas de diferentes escopos se comuniquem entre si.

Escopos de bibliotecas

Escopo é um diretório agrupador, eles podem representar diferentes conjuntos de funcionalidades ou APIs disponíveis. Cada biblioteca ou um grupo de bibliotecas pode ter seu próprio escopo, que define quais recursos são públicos, privados ou internos. Isso permite que os usuários da biblioteca acessem apenas as partes relevantes e evita a exposição de implementações internas. Isso é feito através do tipo API.

As principais diferenças a serem consideradas ao decidir entre abordagens de 4 ou 6 tipos pode variar dependendo das necessidades e características específicas do projeto, como:

1. **Complexidade e tamanho do projeto:** O uso de 4 tipos de bibliotecas pode ser mais adequado para projetos menores e menos complexos, nos quais a modularidade é alcançada com uma estrutura mais simples. Isso evita uma segmentação excessiva e reduz a sobrecarga de gerenciamento de várias bibliotecas. Por outro lado, projetos maiores e mais complexos podem se beneficiar do uso de 6 tipos de bibliotecas, pois isso permite uma maior granularidade e organização do código.
2. **Reutilização de código:** O uso de 6 tipos de bibliotecas, incluindo Bibliotecas Domain e Bibliotecas API, promove uma maior reutilização de código em diferentes partes do projeto. Isso é especialmente benéfico quando há casos de uso com regras de negócio puras que podem ser compartilhadas entre aplicativos ou quando diferentes bibliotecas precisam se comunicar entre si. Se a reutilização de código é uma prioridade, a abordagem com 6 tipos de bibliotecas pode ser mais apropriada.
3. **Flexibilidade e extensibilidade:** A inclusão de Bibliotecas Domain permite que a lógica de negócio seja encapsulada e independente de frameworks externos, facilitando a modificação e a expansão do projeto no futuro. Isso permite que diferentes aplicativos ou partes do código utilizem a mesma lógica de domínio, proporcionando uma maior flexibilidade e extensibilidade. Se o projeto requer uma estrutura que possa se adaptar a mudanças futuras com facilidade, o uso de 6 tipos de bibliotecas pode ser vantajoso.
4. **Manutenção:** O uso de uma maior quantidade de tipos de bibliotecas pode tornar a manutenção do código mais organizada e compreensível. Cada tipo de biblioteca tem sua responsabilidade bem definida, o que facilita a localização e a modificação de partes específicas do código. Isso torna o processo de depuração e correção de problemas mais eficiente. Se a manutenção do código é uma preocupação importante, a abordagem com 6 tipos de bibliotecas pode ser preferível.

Veremos cada um dos tipos de bibliotecas para esclarecer cada um deles.

Principais diferenças para adoção de 4 ou 6 tipos

Também preciso esclarecer sobre escopos, que são nada mais que diretórios agrupadores.

Antes disso, quero adiantar que abordarei sobre restrições de dependências entre elas e que as restrições podem variar dependendo da quantidade de tipos de bibliotecas adotados no repositório, darei minha perspectiva sobre 2 casos, 4 tipos e 6 tipos, como vimos acima.

Bibliotecas `feature`

Conhecida como biblioteca de recursos ou funcionalidades, devemos considerar implementações da interface do usuário inteligentes (smart components), que tenham acesso às fontes de dados e gerenciamento de estado da camada de acesso a dados.

Convenção de nomenclatura

feature (caso aninhado) ou **feature-*** exemplo: **feature-auth**

As bibliotecas do tipo Feature são componentes que possuem regras de negócio e aplicação específicas de um determinado aplicativo. Elas são desenvolvidas para abordar funcionalidades ou partes específicas de um sistema. Essas bibliotecas são projetadas para serem reutilizáveis em diferentes partes do aplicativo, como páginas, módulos ou funcionalidades.

A principal característica das bibliotecas do tipo Feature é encapsular as regras de negócio específicas de uma determinada funcionalidade. Elas contêm a lógica necessária para implementar e manipular as características relacionadas.

Ao criar uma biblioteca do tipo Feature, é importante considerar sua coesão, ou seja, o quanto ela está focada em uma única funcionalidade ou conceito. Isso garante que a biblioteca seja clara e específica em seu propósito, facilitando sua utilização e entendimento pelos desenvolvedores. Elas podem incluir, lógica de negócio, interfaces de usuário personalizadas, validações de formulários, entre outros elementos necessários para implementar a funcionalidade em questão. Essas bibliotecas são desenvolvidas com o objetivo de serem páginas da aplicação carregadas lentamente (lazy-loading) ou uma parte de uma aplicação maior, utilizada em diferentes partes, o que leva a uma maior eficiência no desenvolvimento.

Restrições de dependência

Para 4 principais tipos

Bibliotecas **feature** podem depender de qualquer tipo de biblioteca.

Para 6 tipos utilizados

Não é recomendado que uma biblioteca feature dependa diretamente de bibliotecas domain, mas que seja criado por exemplo um facade na camada de acesso a dados ou dependa de uma biblioteca do tipo API que exponha publicamente apenas o necessário para que os casos de uso sejam utilizados sem que haja dependência direta, pois quando permitimos importações diretas, qualquer coisa que está disponível pode ser utilizada, causando possíveis transtornos em quaisquer alterações, pois podem estar dependendo do recurso que está sendo alterado e manter retrocompatibilidade não é algo que queremos quando há alternativas sem danos ao negócio.

Bibliotecas `data-access`

Este tipo de biblioteca serve como uma camada de delegação no lado do cliente, contendo código de acesso a dados em APIs REST e todo tipo de arquivo relacionado ao gerenciamento de estado e estes, por convenção, residem no diretório **src/lib/+state**.

Convenção de nomenclatura

data-access (caso aninhado) ou **data-access-*** exemplo: **data-access-auth**

As bibliotecas do tipo Data Access são projetadas para facilitar o acesso a dados e o gerenciamento de estado em um aplicativo. Elas fornecem um conjunto de classes, métodos e utilitários que permitem recuperar, armazenar, atualizar e excluir informações de uma fonte de dados, como um banco de dados ou uma API back-end.

O objetivo principal das bibliotecas do tipo Data Access é abstrair as complexidades e detalhes técnicos envolvidos no acesso a dados. Elas oferecem uma camada de abstração que permite que os desenvolvedores interajam com os dados de maneira simplificada e consistente, independentemente da origem dos dados.

Essas bibliotecas geralmente fornecem abstrações e classes concretas para realizar operações comuns de acesso a dados, como consultas, inserções, atualizações e exclusões. Elas também podem oferecer recursos adicionais, como mapeamento objeto-relacional (ORM), gerenciamento de transações, cache de dados e tratamento de erros.

Ao utilizar uma biblioteca do tipo Data Access, os desenvolvedores podem se concentrar na lógica de negócio do aplicativo, em vez de se preocupar com os detalhes de como se conectar e interagir com a fonte de dados. Isso aumenta a produtividade e a eficiência do desenvolvimento, permitindo que eles se concentrem nas funcionalidades principais do aplicativo.

Outra vantagem das bibliotecas do tipo Data Access é a padronização do acesso a dados. Elas estabelecem convenções e melhores práticas para a interação com a fonte de dados, o que ajuda a garantir a consistência e a integridade dos dados manipulados pelo aplicativo. Além disso, a abstração fornecida por essas bibliotecas permite que seja mais

fácil alterar a fonte de dados no futuro, caso seja necessário migrar para um banco de dados diferente ou uma API atualizada.

Restrições de dependência

Para 4 principais tipos

Camadas de acesso a dados podem depender de outras camadas de acesso a dados ou camadas utilitárias.

Para 6 tipos utilizados

Neste caso, acesso a dados também pode depender do tipo domain no mesmo escopo e da camada API para acesso a domínios em outros escopos.

Bibliotecas `ui`

Abriga uma coleção de componentes de apresentação relacionados. Geralmente não dependem de serviços por injeção de dependências, todos os dados necessários devem vir de entradas (valores de atributos / props) e eventos para emissão de dados.

Convenção de nomenclatura

ui (caso aninhado) ou **ui-*** exemplo: **ui-dialogs**

As bibliotecas do tipo UI são projetadas para fornecer componentes de interface do usuário que podem ser compartilhados entre diferentes partes de um aplicativo ou até mesmo entre aplicativos distintos. Essas bibliotecas são frequentemente chamadas de "UI shared" ou "UI kits".

O objetivo principal das bibliotecas do tipo UI é promover a consistência e a eficiência no desenvolvimento da interface do usuário. Elas encapsulam componentes reutilizáveis, como botões, caixas de texto, menus, barras de navegação e outros elementos de interface comuns. Esses componentes são projetados para serem flexíveis, personalizáveis e facilmente integrados em diferentes partes do aplicativo.

Ao utilizar uma biblioteca do tipo UI, os desenvolvedores podem economizar tempo e esforço, pois não precisam criar cada componente de interface do zero. Eles podem simplesmente importar os componentes fornecidos pela biblioteca e utilizá-los em suas implementações. Isso resulta em uma padronização visual e de comportamento em todo o aplicativo, oferecendo uma experiência de usuário consistente.

As bibliotecas do tipo UI também podem incluir estilos de CSS, temas e padrões de design para garantir a coerência estética do aplicativo. Isso facilita a manutenção do visual do aplicativo e a aplicação de alterações de estilo de forma consistente em todo o sistema.

Além disso, essas bibliotecas geralmente fornecem uma abstração do código complexo e detalhado necessário para criar interações de interface do usuário. Os componentes encapsulam a lógica necessária para responder a eventos, interagir com o usuário e atualizar a interface conforme necessário. Isso ajuda a simplificar o desenvolvimento e a garantir uma base sólida para a construção de uma interface do usuário interativa e responsiva.

Uma das principais vantagens das bibliotecas do tipo UI é a reutilização de código. Ao usar os componentes fornecidos pela biblioteca em todo o aplicativo, as atualizações e correções em um único lugar se refletirão automaticamente em todos os locais onde esses componentes são utilizados. Isso reduz a duplicação de código, facilita a manutenção e melhora a consistência.

No entanto, é importante notar que as bibliotecas do tipo UI podem não ser adequadas para todas as situações. Em alguns casos, pode ser necessário criar componentes personalizados para atender às necessidades específicas do aplicativo. Além disso, a escolha de uma biblioteca do tipo UI deve levar em consideração a compatibilidade com as tecnologias e frameworks utilizados no projeto.

Restrições de dependência

Para 4 e 6 principais tipos

Bibliotecas UI podem depender de outras UIs ou utilitárias.

Bibliotecas `util`

Uma biblioteca de utilitários contém código de baixo nível usado por muitas bibliotecas. Frequentemente, não há código específico do framework e a biblioteca é simplesmente uma coleção de utilitários ou funções puras.

Convenção de nomenclatura

util (caso aninhado) ou **util-*** exemplo: **util-testing**

Elas oferecem funcionalidades genéricas e comuns que podem ser usadas em diferentes partes de um aplicativo ou até mesmo em vários aplicativos. Elas são projetadas para fornecer um conjunto de ferramentas e utilidades que facilitam tarefas comuns de programação, independentemente do domínio específico do aplicativo.

O objetivo principal das bibliotecas do tipo Útil é promover a reutilização de código e simplificar o desenvolvimento, fornecendo funcionalidades comuns que podem ser usadas em várias partes do aplicativo. Essas bibliotecas abrangem uma ampla gama de recursos e utilidades, como manipulação de strings, formatação de datas, cálculos matemáticos, conversões de dados, operações de arquivo, gerenciamento de configurações e muito mais.

Ao usar bibliotecas do tipo Útil, os desenvolvedores podem economizar tempo e esforço ao evitar a necessidade de implementar essas funcionalidades comuns repetidamente em diferentes partes do código. Isso resulta em um código mais limpo, mais conciso e mais legível, além de melhorar a manutenibilidade e a consistência do aplicativo.

Elas são projetadas para serem independentes do domínio do aplicativo, o que significa que podem ser utilizadas em diferentes projetos, independentemente do seu escopo ou propósito.

Essas bibliotecas normalmente são compostas por uma coleção de classes, funções e utilitários que são organizados de forma coesa e bem documentada. Também podem ser distribuídas como pacotes ou módulos separados, permitindo que os desenvolvedores escolham e utilizem apenas as funcionalidades que são relevantes para suas necessidades.

Além de fornecer funcionalidades genéricas, as bibliotecas do tipo Util também podem incluir recursos adicionais, como tratamento de erros, logging, serialização/deserialização de objetos, geração de código, criptografia e muito mais. Esses recursos adicionais ajudam a melhorar a qualidade, a segurança e o desempenho do aplicativo.

Bibliotecas `domain`

As bibliotecas do tipo **domain** são projetadas para encapsular as regras de negócio e a lógica pura de um domínio específico. Elas são focadas em representar conceitos, entidades e operações relacionadas ao núcleo da aplicação, independentemente da implementação técnica ou do ambiente em que o sistema está sendo executado.

Convenção de nomenclatura

domain (caso aninhado) ou **domain-*** exemplo: **domain-accounts**

O objetivo principal destas bibliotecas é separar e isolar a lógica de negócio do restante do código da aplicação. Elas encapsulam as regras de negócio complexas e fornecem uma interface clara e coesa para que as outras partes do sistema possam interagir com o domínio de negócio específico.

Ao utilizar bibliotecas do tipo **domain**, os desenvolvedores podem se concentrar no entendimento e na modelagem do domínio em que estão trabalhando. Isso permite uma melhor expressão da lógica de negócio e facilita a manutenção e a evolução do sistema ao longo do tempo.

As bibliotecas do tipo **domain** são tipicamente compostas por classes, interfaces e estruturas que representam as entidades e os conceitos do domínio. Elas encapsulam as operações e as interações entre essas entidades, bem como as regras e as restrições que governam o comportamento do domínio.

Essas bibliotecas podem conter métodos e algoritmos específicos do domínio, validações de dados, cálculos complexos, fluxos de negócio e outras operações relacionadas às regras e ao comportamento específico do domínio.

Uma vantagem significativa das bibliotecas do tipo **domain** é a reutilização de lógica de negócio entre diferentes partes do aplicativo ou até mesmo entre projetos diferentes. Ao separar as regras de negócio em uma biblioteca de domínio, é possível compartilhar e reutilizar a lógica em várias camadas da aplicação, como a camada de interface do usuário, a camada de acesso a dados e a camada de serviços.

Essas bibliotecas também podem ser testadas de forma independente, facilitando a realização de testes unitários e de integração para validar a lógica de negócio do domínio. Isso permite a detecção precoce de erros e garante que a lógica de negócio esteja funcionando corretamente.

Bibliotecas `api`

Essas bibliotecas atuam como pontos de entrada para acessar recursos e funcionalidades específicas do monorepo. Elas podem fornecer métodos e classes que expõem serviços, endpoints de API ou outros mecanismos de comunicação.

Convenção de nomenclatura

api (caso aninhado) ou **api-*** exemplo: **api-products**

Ao utilizar bibliotecas do tipo API em monorepos, é possível promover a reutilização de código e a consistência na comunicação entre os diferentes componentes do sistema. Elas permitem que os módulos do monorepo interajam uns com os outros de maneira padronizada, evitando duplicação de código e facilitando a manutenção e a evolução do sistema.

Quando diferentes partes do monorepo importam diretamente umas às outras, isso cria uma dependência rígida entre essas partes. Isso significa que qualquer mudança em uma parte pode afetar diretamente as outras partes que dependem dela. Essa dependência direta pode resultar em acoplamento excessivo, dificultando a manutenção e a evolução do código.

Ao usar bibliotecas do tipo API, cria-se uma camada de abstração entre os módulos. Cada módulo acessa a funcionalidade e os recursos de outros módulos por meio de uma interface definida pela biblioteca API. Isso permite que os módulos dependam apenas da interface da biblioteca, em vez de depender diretamente uns dos outros.