**GIT FLOW:**

Para começar a fazer as alterações do fatecarona é preciso preparar o fluxo de desenvolvimento, o fatecarona está no git na branch “redux” foi como o grupo anterior deixou... para melhorar a organização do GIT eu vou usar a metodologia do GIT FLOW, onde tem 3 branchs principais:

**master**: é a branch principal onde fica a versão de produção

**develop**: é onde vai ser criado as features e conter as atualizações, no fim ele vai dar merge na máster.

**Hotfix**: é a branch para corrigir bug de produção, ela altera tanto a develop como a master, como o fatecarona não está em produção ela não será usada.

O GIT FLOW é usado baseado em FEATURES, que são pequenas mudanças isoladas, sabe os itens do design thinking? Cada item daqueles vai ser uma feature que vai nascer a partir da develop, ser realizada na branch “feature/nome\_da\_branch” e por fim ser mesclada na develop novamente, por exemplo o card do design thinking de “arruma senha”... é criado uma nova branch com nome “feature/arrumar\_senha”, todo o código é desenvolvido nessa branch isolada da develop e após ser concluída ela volta para a develop. Isso permite desenvolvimento paralelo de features já que ficam em ambientes isolados (também ajuda em **code coverage** já que ajuda a fazer testes unitários em cada feature isoladamente, mas não abordarei isso nesse momento).

**Padrões:**

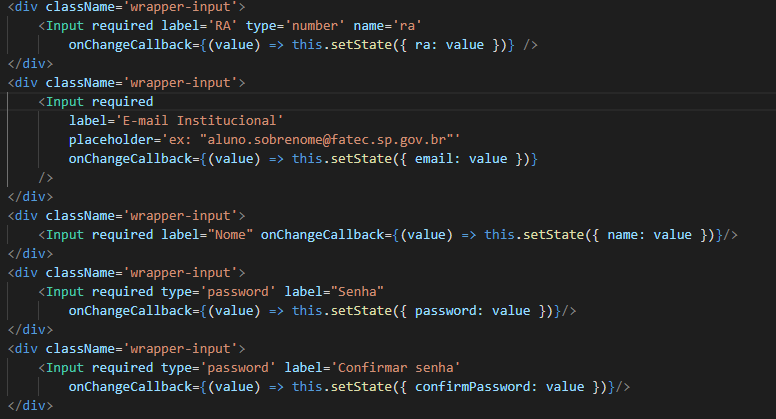
O código do WebApp não tem padrão, metade está em português e a outra metade em inglês, como tem mais trechos em português o código vai ser traduzido para inglês

**DRY (don’t repeat yourself):**

O DRY diz para retirarmos repetições no código usando abstrações, na feature “arrumar senha” por exemplo tínhamos esse código:

// print dos input repetido

Cheio de repetições de código e parâmetros, então decidi abstrair um input genérico que já vem com funcionalidades básicas necessárias no react com parâmetros default que é utilizado em todo lugar no webapp e ficou assim:



Caso tenha que mudar um layout de input do webapp, mudando apenas o componente “input” que criei altera em todos os trechos melhorando MUITO a manutenibilidade.

Dry não deve ser confundido com reduzir redundância no código, isso se faz com boas práticas e alguns code convetions.

Temos outro conceito chamado **WET (write everythinf twice)** que significa “escreva tudo duas vezes”... completamente diferente do DRY?

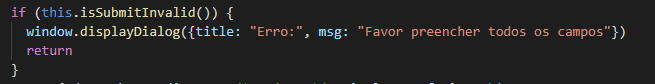
Não! O WET segue um principio que uma repetição de código pode fazer duas coisas diferentes parecerem iguais mas na verdade não são!

Tipo uma regra de negócio poder mudar no futuro, então com isso o WET fala que tudo bem ter uma repetição que isso não tem grande impacto na manutenibilidade e pode ajudar futuramente.

Então enquanto você estiver escrevendo algum código e ver que há apenas uma repetição, não é necessário abstrair de imediato, apenas com 3 ou mais.

**Mensagens de Erro:**

As mensagens de erro do Fatecarona são feitas a partir da lib do material design que sua referência é armazenada em uma variável global “window” de manipulação de DOM, em vários lugares do webapp temos código desse tipo:

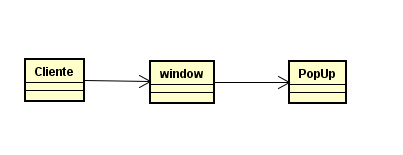


Essa é um popup básico do fatecarona usando diretamente na implementação de um componente chamado “GlobalDialog” bastante genérico, agora irei criar meu próprio popup com estilos pré-definidos... do jeito que está vou ter que alterar todos os “window.displayDialog” (que tem 41 resultados diferentes no projeto, vou literalmente alterar em 41 lugares diferentes ☹ vai dar um trabalhão)

Durante o survey as pessoas ficaram travadas no popup, ou ficaram esperando alguma coisa ou procuravam um botão de fechar e não acontecia nada, então vou criar minha própria “lib” de popup para ter um ponto único que posso alterar e customizar...

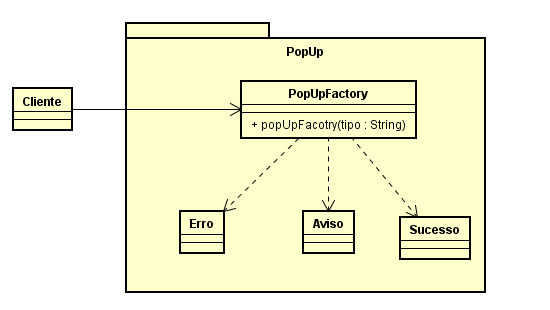
Criando o pacote PopUp:

O popUp atual está assim:



Cada pagina chama o popup passando todos os parâmetros.

Criando meu próprio pacote ficou assim:



O cliente passa um parâmetro “tipo” para o popupfactory, assim ele escolhe qual implementação irá usar já com parâmetros pré-definidos.

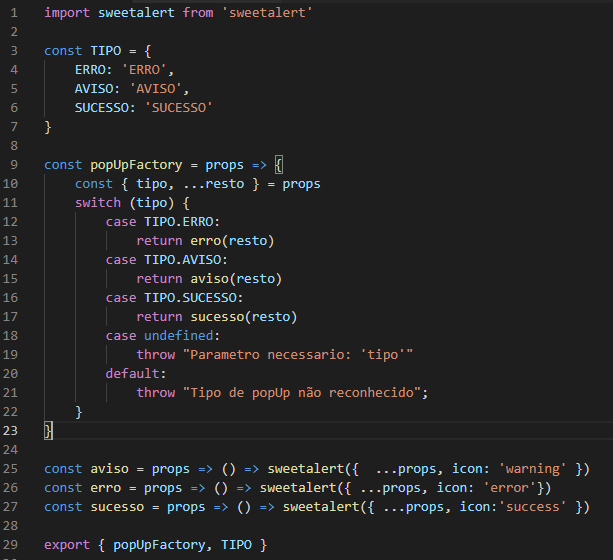
Benefícios dessa nova estrutura:

A aplicação não chama a classe que tem detalhes de implementação diretamente, assim o cliente não precisa conhecer os detalhes da implementação do popup que estão nas classes especialistas, caso você queira adicionar um ícone ou animação você teria que faze-lo no cliente por meio de parâmetros, com a nova estrutura os detalhes ficam nas classes especialistas o cliente só tem que pedir o tipo de popUp e o texto que será exibido.

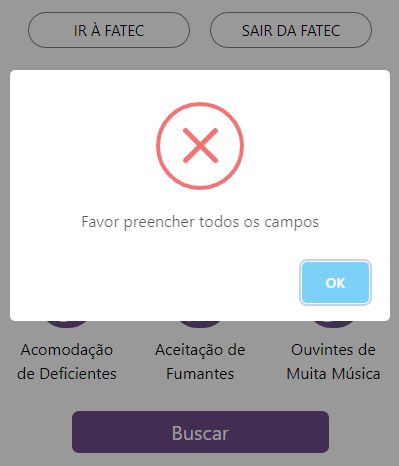
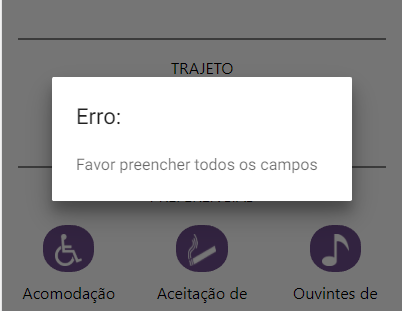
Forçando a barra, poderia fazer uma comparação com um dos princípios do SOLID, a inversão de dependência que diz para dependermos de uma abstração ao invés de uma implementação, antes dependíamos da implementação do displayDialog, agora de uma “abstração” Factory.

A manutenabilidade melhora também, lembra do DRY? Estamos repetindo o mesmo código de displayDialog em 41 lugares diferentes, muitas vezes passando o mesmo parâmetro, agora caso precise alterar o popup novamente no futuro, ele está concentrado em apenas um local.

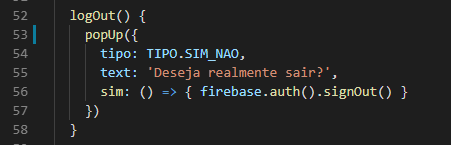
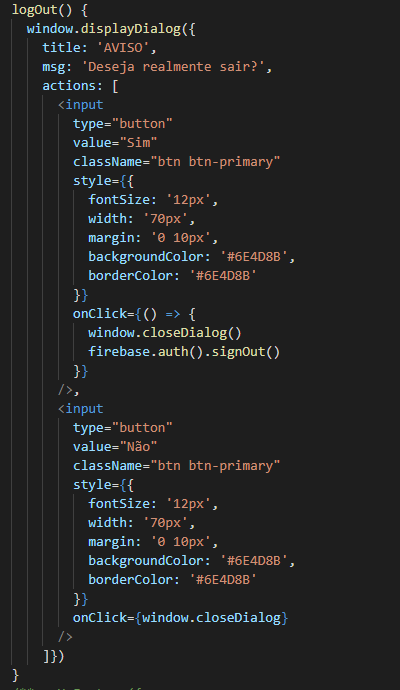
A estrutura básica fica assim(essa **não** é a versão final):



Mensagem de erro antes (pessoas não sabiam sair dela) e depois (sempre com botão de escape):



Antes \\ depois (falar da redução de codigo e esconder detalhes):



Esse é o código de uma view, da para notar a diferença entre responsabilidades, antes a view era responsável de montar o popup, hoje o responsável de montar o popup é o próprio popup.

**Código Final:**



Esse caso da mensagem de erro deu para explorar bastante conteúdo do paradigma funcional, vou explicar alguns aqui:

Note na linha 27, temos a seguinte linha

Const aviso = props => () => ...

Variável = função => função

Nessa linha temos uma função cujo a referência está em uma variável, isso é possível por que o javascript suporta **first class functions** que são funções que são tratadas como objetos e sua referencia pode ser armazenada dentro de uma variável.

Mas essa linha é mais interessante que isso, que além de salvar a referência de uma função em uma variável, essa função também retorna outra função como resultado! Olhe bem, temos a primeira função que recebe “props” como parâmetro e temos outra função interna que não recebe parâmetro algum “()” única coisa que a função wrapper faz é retornar outra, mas por que eu fiz isso?

Aproveitei o **first class functions** para fazer uso de outro conceito bem parecido chamado **higher order functions** que são funções que podem ser passadas como parâmetro que o javascript também suporta.

*Isso vai ser meio complicado de explicar, mas vamos lá.*

Olha a linha 17:

Temos um withRedirect(aviso(resto))(redirect)

A função aviso(resto) retorna outra função.

A função withRedirect recebe uma função por parâmetro e retorna outra função que por sua vez recebe uma string por parâmetro, isso é devido que ao final de um popup, você pode redirecionar o usuário para outra tela.

Logo em seguida que a função com apenas um parâmetro é retornada ela é imediatamente executada passando o parâmetro redirect (withRedirect(aviso(resto))(redirect))

A função aviso e withRedirect foram combinadas para fazer uma **composição de funções**.