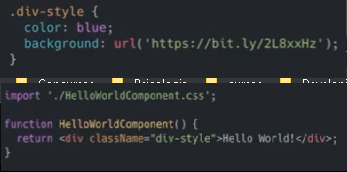
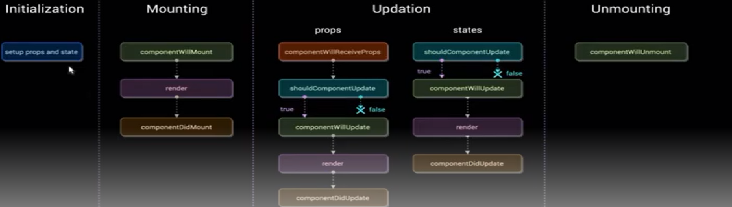
Existem 3 principais maneiras de estilização de componentes no reactJs:

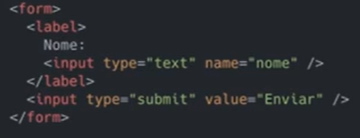
**Inline** a maneira mais básica de se estilizar um componente, utilizando a tag **style**, que são declaradas em CamelCase. Também é possivel jogar o style dentro da função principal de app. Os pontos positivos é que o Inline é **a maneira mais prática** de se estilizar, **permite ajustes rapidos** e **teste prático de novos estilos**. O ponto negativo é que quanto maior o código mais **difícil a** **manutenção do código**.

**Classes** é a segunda maneira de se estilizar, vale notar que é necessário colocar **className** ao se fazer dessa forma, e não **class**, isso é feito propositalmente. A class é a **maneira mais modular de criar estilizações**, porêm possui **difícil manutenção**, **pouca flexibilidade** e **pode causar conflitos de nome**.

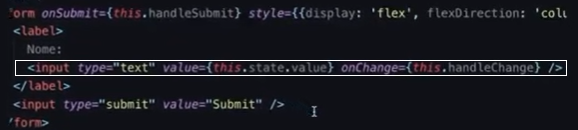
**CSS in JS** é um método que utiliza do npm para instalar componentes que permitem a edição direta do CSS dentro do JS, o seu grande forte é a **facil manutenção através a eliminação do arquivo CSS e a eslitilização dinâmica**, **performance**, **permite a injeção automatica de perfixos vendors**.

**Stateful** é quando o componente **usa estados**. Todo stateful possui o uso de state dentro do componente e tende a ser criado por Class, o React possui imbutido dentro de sua biblioteca diversos estados do ciclo de vida.

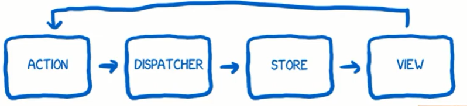
**Stateless** é quando o componente **não usa estados**. Ele não possui gerenciamento de estados no componente, sendo totalmente construido através de funções em JS. Esses tipos de componentes eles são em gerais usados para renderizar modos, icones e imagens, diminuindo o código.

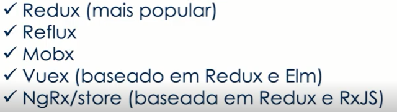
**Formulários** mantem estados internos, são feitos pelo <**form>**, em React podemos trabalhar de duas formas, através dos componentes **controlados** e **não-controlados**.

O controlado é **quando o estado**, **o ‘single-source-of-truth’ é o React**, nele tanto **select**, **input** ou **textarea** aceitam um atributo **value**, e nós podemos mudar esse valor usando o atributo **onChange**.

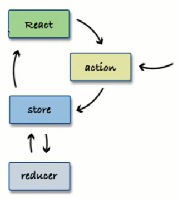


O componente não controlado, por sua vez, pega o **estado do DOM**, com a tag input **apenas servindo para a leitura**. A recomendação do React é sempre utilizar os componentes controlados, a única excessão à regra é o uso da tag Input mencionada já que ela funciona anpenas dessa forma.

**Flux** é uma arquitetura criada pelo facebook **para a comunicação entre componentes**, é um padrão de projetos para trafego de dados de maneira funcional.

O **action** é como um telegrafo, ele formata a mensagem a ser enviada. O **dispatcher** guarda todos os callbacks para diferenciar os stores. A **store** guarda todas as informações **e estados** e os **altera**, **apenas o store pode alterar as informações**. A **view** é um ‘middleware’ entre o store e a tela, é essa função que pede ao store a alteração de informações e passa os dados para visão abaixo delas. Ao lado estão seus principais usos.

**As bibliotecas baseadas em Flux servem para a comunicação entre componentes**, centralizam as informações, nas palavras do criador, “bibliotecas de Flux são como óculos, voce vai notar quando precisar deles.” – Dan Abramov

**Redux** é a principal implementação do flux, ele funciona de uma forma ligeiramente diferente. Ele possuo três princípios básicos:

**Single-source-truth**, ou seja possui uma única store.

**State** é **read-only**.

Mudanças são feitas **por pure functions**, o que significa que o estado é imutavel.

Em redux, as actions são como flux, porêm elas **não enviam a action em sí para o dispatcher**, **mas sim um objeto de action formatado**.

A store aqui cuida de toda a arvore de estados onde os **reducers que cuidam qual em descobrir qual estado ocorrerá mudanças**.

Como o redux não possui dispatcher, a store se conecta ao root reducer, que divide os estados em pequenos reducers para descobrir como lidar com esse estado, que **aqui são imutáveis**.

Em react, a camada view sofre três adições para conectar-se ao store:

**Provider**, um wrapper da arvore de componentes, que torna mais facil para o view se conectar a store usando o connect.

**Connect()** por sua vez é uma função estilo hook que faz a conexão com o selector.

**Selector** é uma função que fazemos que define **quais os estados do redux que queremos passar como procs**.

O **Redux é instalado através do Redux DevTools no chrome**, é uma extensão! Gritos internos. Aqui e guiou um pouco seu funcionamento, da para ver tanto pelo extensor quanto pelo inspector, perto do console.

O **root component** ela tem algumas responsabilidades, como a **criação de stores** e a **associação de reduxes a ela**.

Os **APIs HTTP** servem para conectar a um ou mais servidores HTTP, pelo **get** (para listagem**), post** (para criação), **delete** (deleção), **put** (para alteração), o **patch** também pode ser usado como put.

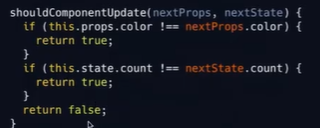
Também temos o Fetch API e o AXIOS.

A Fetch API é uma recente biblioteca nativa do browser, que nem todos os browser estão ainda adaptados a ela, oferece uma alternativa ao XMLHttpResquest() e o Jquery.ajax(), **suportando a service works**, sua principal diferença é que **o Fetch API não envia nem recebe cookies, precisando definir todos os adders corretamente para ter acesso**, **também não rejeita o status de erro de HTTP**.

O AXIOS é uma lib e HTTP API, uma outra forma de trabalhar com HTTP, é **cross-browser**, pode monitorar o progresso de um request, possui um melhor tratamento de erro e testagem.

**Imutabilidade** se refere a uma coleção que não pode ser alterada, novas coleções podem ser criadas apartir de uma anterior e uma mutação **(setter)** como um conjunto.

Novas coleções são criadas utilizando o maximo de instruções da coleção original, reduzindo a copia e melhora performance. Além disso, a programação é mais simples e o debugging simplificado. Se perfomance em React é requerido, utiliza-se os dados imutaveis, voce consegue utilizando o **shouldComponentUpdate** ou **React.PureComponent**.

ele funciona em geral através do uso de **comparação shallow**.

A imutabilidade é um pre-requisito do redux, utilizam **comparações** **rasas** e cuidam de **manipulação de dados de forma segura**, com **time-travel debugging**.

Os reducers dividem o objeto de estados em dominíos por uma chave, *combineReducers* checa mudanças usando a comparação rasa, fazendo interação com os reducers e criam um novo objeto de estado apartir dos estados retornados por cada reducer.

O connect assim gera componentes que fazem comparação rasa com o **estado root**, retornam um valor para a função mapStateToProps, verificando aqueles que precisam de uma operação de re-render.

**Redux + Rest**

**Redux middleware** preve uma camada entre o disparo de uma ação e o momento que ela atinge o reducer, utilizado para uma variedade de funções, entre elas os **APIs** **de serviço**.

Existem varios métodos de uso, o que pode gerar uma visão de “gambiarra” algumas vezes. O professor mostrou diversos métodos de se utilizar para a mesma coisa como forma de entender seu funcionamento. Três diferenças do applyMiddleware() do Redux:

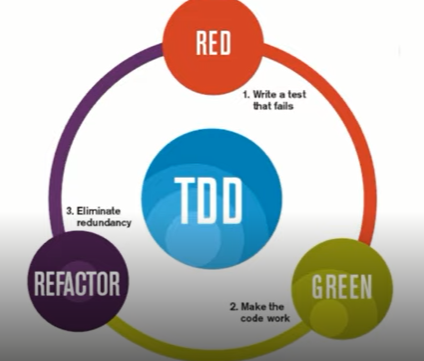
Ela só expoe um subconjunto da Store API para o middleware: dispatch e getState().

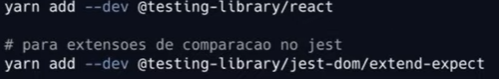
Fica difícil saber se o store.dispatch(action) vai realmente percorrer a cadeia de middleware de novo.

E ela opera em cima de createStore() ao invez da store em sí.

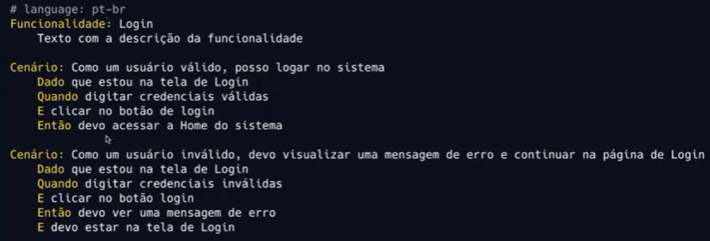
O **redux-thunk**, vem de pensar, é uma função que chama outra função. É uma biblioteca, instalada através do comando *yarn add redux-thunk*.

O **redux-saga**, que fica a critério de conceito para quem quiser aprender mais ok.

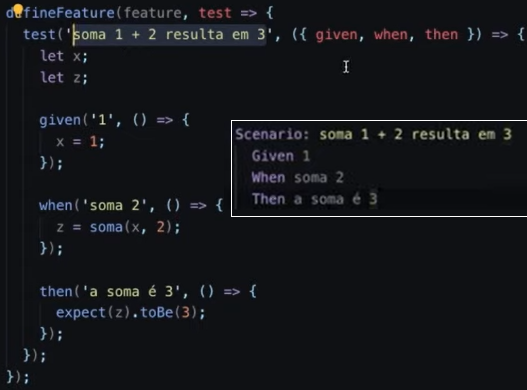
TDD, **Test-Driven Development**, funciona para antecipar erros no nivel de desenvolvimento, objetiva **escrever um código para testar nosso próprio código**, é um teste escrito antes da funcionalidade, **não descarta a presença de um tester**, sendo uma garantia de controle em grandes produtos que mostra falhas antes da produção. Pode ser usado para testes unitários ou testes end-to-end.

**Jest** é um aplicativo de linguagem JS que serve para testagem, que utiliza o Jasmine.

Note que para utilizar as bibliotecas de testagem é necessário instalá-las com o comando ao lado.

BDD, **Behavior-Driven Development**, um tipo de teste de especificação que reune com o teste automatizado e premissa de teste. Nos traz uma **sintaxe Gherkin**, uma sintax de steps para definir cenários, descrevendo cada funcionalidade por feature (caso de uso).

A imagem ao lado esta traduzida, **mas é sempre feito em ingles**, utiliza-se palavras reservadas que descrevem o que acontece.

Existem duas bibliotecas para BDD, a **Jest-cucumber**  e a **Chai**. São ferramentas e tecnicas que auxiliam no uso de testes. Um exemplo de teste esta ao lado.

Debugging é o processo de encontrar e reduzir defeitos de um software, **significa depuração em portugues**. É feito de cabeça, através de logs, ou ferramentas de teste e debugging. Existem três ferramentas principais, **Chrome DevTools**, **Redux DevTools** e **React DevTools**.

Tratamento de erros em programação é a correção e solução de erros dentro de um software, um bom tratamento de erro traz resiliencia ao software e segurança.